Concordia University
Department of Computer Science and Software Engineering
COMP 6521 – Advanced Database Technology and Applications
Winter 2016

Implementation Project

Team:
Baitassov Ulan 40001592 u_baitas@encs.concordia.ca
Adejugba Adeola 40011859 a_adejug@encs.concordia.ca
Sajjad Ashraf 40012126 a_sajjad@encs.concordia.ca

REPORT

Description of the program:

It was required to design secondary dense index on attribute **AGE**. For a large file with billions of records, there can be many people of the same age scattered randomly in the data file. Even to store the SIN and AGE would result in big index file. So we decided to break data file in blocks and stored block indexes in a way to make it efficient.

Our program is using ***secondary dense index on buckets*** which contain indexes of the blocks in the data file. Buckets are using the ***sparse index on the Data file*.**

There are 82 buckets in total. Each bucket is referred to a certain age. For example buckets [0] is for age 18, and contain indexes to the blocks that have age 18 in them. Some portion of buckets are stored in the main memory and are sorted. Other overflow buckets are stored in the index file. Every overflow bucket has pointer to previous overflowed bucket, it helps to easily locate the chain of overflow buckets for a given age.

Below you can see, the data structure design:

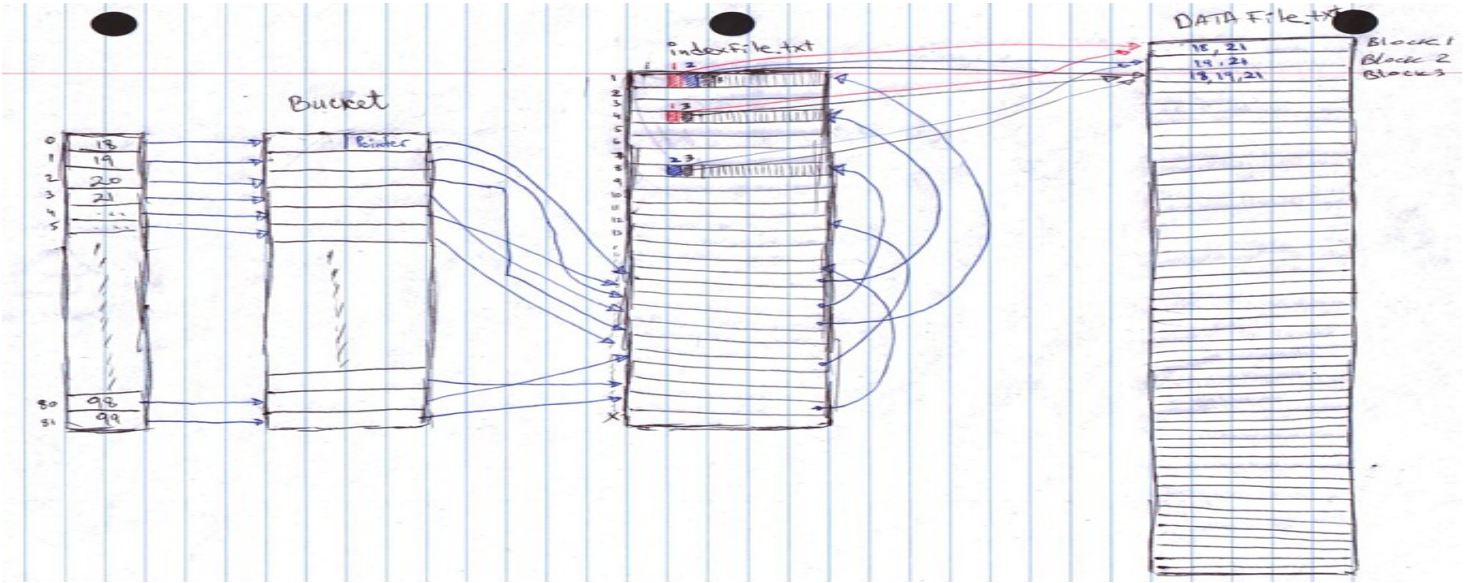

Figure 1: Data Structure Design illustrating the process followed to create the index file.

The program is written in one Main class with main function that runs step by step by calling several methods to construct the index file and search for records.

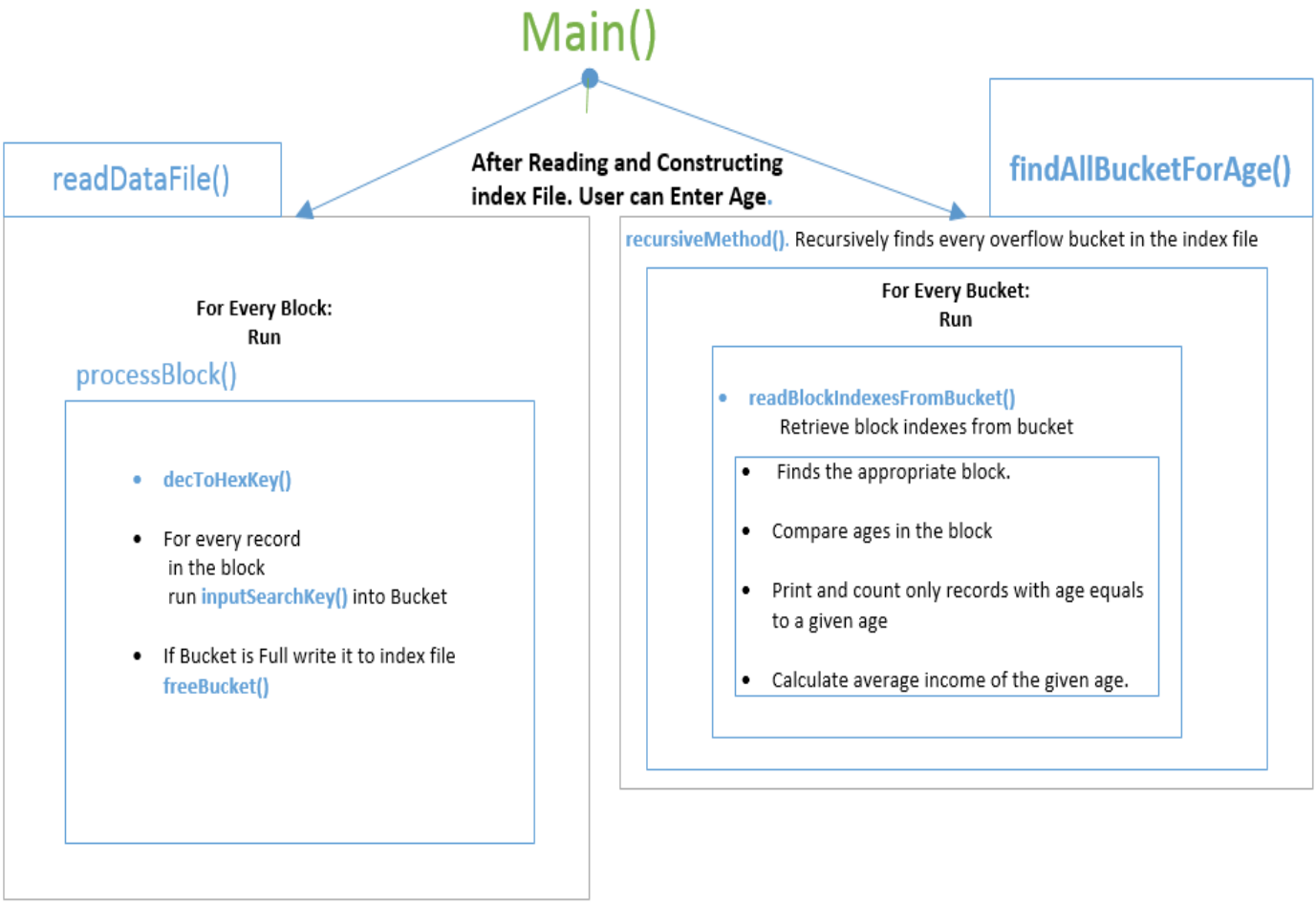

Figure 2: Illustrates the connection of some methods used.

At first, the program reads entire data file to construct the index file. The data file is divided by blocks, each 4000 bytes. Program reads one block at a time and process it. The process include reading the records from the block and filling the buckets with current block index according to the age of the record.

Bucket size is the same as blocks that is 4000 bytes. If the bucket of any age is filled, and there are still more blocks to read from the data file, then program writes the bucket to index file and create new one and stores previous buckets address as a pointer in the first 2-10 bytes. The very first bucket has pointer to nowhere which we indicated as “00000000”.

Here is the bucket structure:

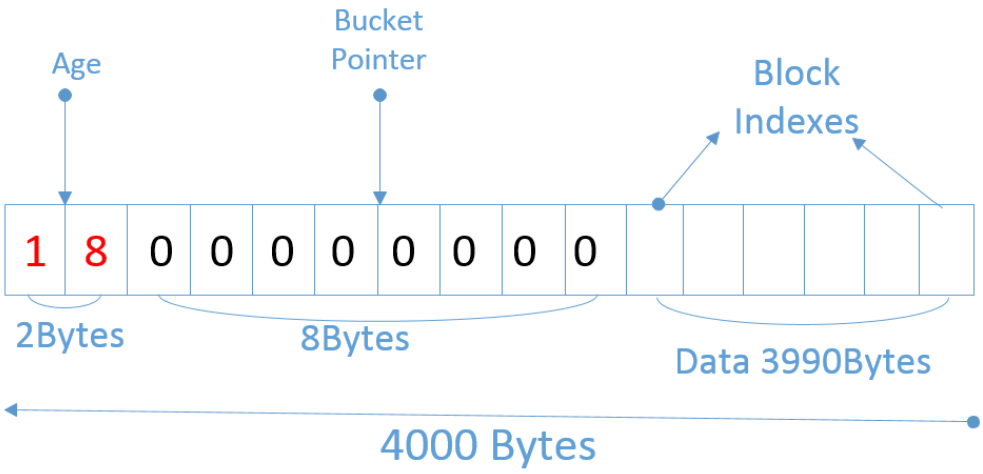

Figure 3: Diagram to illustrate how we implemented the Bucket structure.

After it finished constructing the index file, user can enter the age to print all records of a given age, count number of people of that age and compute the average income salary. First, when the age is entered, the program finds the bucket related to this age in the main memory and retrieves the pointer. If the pointer hasn't got all zeros, then it points to another overflow bucket in the index file. So, we decode this value and retrieve next bucket and repeat the same operation, until we reach the very first bucket which has pointer as "00000000". After finding all buckets, program reads block indexes from buckets and retrieves the blocks from data file. For every block, it searches through the whole records in the block and compares the given age to the record's age. If it equals, we print the record in the "Output.txt", increment countPeople and add salary to sum (later we computed the average salary dividing the sum by countPeople).

We converted the block index number in hexadecimal number because this way we can store more blocks using less bytes.

FFF (3bytes) = 4095 (4 bytes)
FFFF (4bytes) = 65 535 (5 bytes)
FFFFF (5 bytes) = 1 048 575 (7 bytes)

This approach decreased the index file size. For 5 GB of data, index file size was 230 MB.

Description of methods in the program:

We only used one Main class with several methods.
Description of each of the method in class Main is as follows.

Methods	Description
• **public static void** readDataFile()	This method reads the data from a file by blocks
• **public static void** processBlock()	This method process each block that was read
• **public static void** decToHexKey()	This method converts the decimal block index into hexadecimal
• **public static void** InputSearchKeyIntoBucket()	This method puts block index into appropriate bucket number according to age
• **public static void** freeBucket(**int** bucketNumber)	This method writes the bucket to the file.
• **public static void** writeToIndexFile()	This method writes the all buckets in memory into a "IndexZero.txt" file
• **public static void** findAllBlocksForAge(**int** new_age)	This method is used to initiate the search for all records of a given age.
• **public static boolean** recursiveMethod(**byte**[] block, **int** new_age)	This method is recursive method to search through whole overflow buckets till the very first bucket of a given age.
• **public static void** readBlockIndexesFromBucket(**byte**[] block, **int** new_age)	This method retrieves the block indexes from bucket.
• **public static byte**[] findDataBlock(**int** new_index, String new_filename)	This method finds block in the data file and reads it.
• **public static byte**[] findBucket(int new_index)	This method finds bucket in the index file and reads it.
• **public static int** decodePointer(String new_pointer)	This method converts the hexadecimal string to integer value
• **public static void** readRecordsFromBlock(**int** pos, **int** new_age)	This method reads all records from block which satisfy age comparison.
• **public static void** main(String[] args)	This is the main method in which all the

Difficulties and Limitation Faced

Finalizing on the best approach was quite tedious. The considered data file size was very large (2TB) so looking for the best method to implement it with less time, minimum disk I/O and less space should be the best approach.

The first difficulty we faced was how many bytes to give each block index in index

file. If the data file was very large, then we should have allocated more bytes to each block index which would result in big index file size. This would have effect on hard disk space and performance as well. As we have more data to read. The solution we found was to convert the decimal value of the block index to hexadecimal value. For 7 bytes decimal value, we only used 5 bytes hexadecimal value. Also, we calculate the number of blocks required and assign the byte length to block index according to data file size.

The second difficulty was memory management. Firstly, we wanted to write the buckets in sorted way, however as buckets were filling up, we needed to write them in order to prevent the memory shortage. So we came up with bucket pointer saving solution. Now, when we write bucket in the index file, we save its address in the next the bucket.

The third problem is printing the result. After we found the blocks where records of the given age are located, then we tried to print record data in "System.out.println();" the console wouldn't show all data. So we decided to write in the txt file. However, the writing to the disk takes time, so now the search is slower than it was. We are still working on solution and hope till demo presentation we solve the issue.

Division of tasks in the group

Baitassov Ulan 40001592
Did analysis on how to implement the project. That is, the most proffered data structure to use to implement the project work. Implemented part of the project, troubleshoots the code to ensure its working according to specifications and also worked on description of the classes used in the program.

Adejugba Adeola: 40011859
Analyzed the project implementation and suggested a step by step method to solve the problem to meet up with the required specifications. Designed each of the diagram to better illustrate how we implemented the project work. Also worked on the documentation.

Sajjad Ashraf 40012126
Worked on part of the program and contributed to the process to follow to implement the project. Debugging of the code to ensure that the program is working without faults or bugs.

Results and Analytics

1) **Analyzing Result using 2MB memory size on 5MB data file.**

```
file size = 4996186112 bytes.
49961861 records.
number of blocks required to READ = 1249046
Program started...
Index File has been constructed...
Time taken = 45899 ms
Number of I/O WRITE = 59627
Number of blocks to store the index = 59709
Total number of I/O = 1308673
1. For one age:
2. Print average income for every 10 ages:
1
Enter the age 18-99: 18
Number of people of age 18 = 616184
The Average yearly Income = 97095.22012418369
Time taken = 69663 ms
Number of I/O = 489380
1. For one age:
2. Print average income for every 10 ages:
```

Figure 4: Pictorial view showing what it displays when running the program using 5GB data. Limiting the memory size to 2MB.

Step by step explanation of how Figure 4 output was derived.

The data file size is = **4996186112 bytes ~ 5GB** which is 49961861 records.

- File size divided by the total record on each line which is 100 byte gives **49961861 records.**

In our program 1 block is 4000 bytes.

- Number of blocks required to read is = **4996186112/4000 = 1249046 blocks.**

We read each block at a time, so it means **1249046** I/O reads to read the data file.
Each time we write the overflow bucket in the index file, we increment the I/O.
Therefore, number of I/O writes is **59627** means this many overflow buckets were written in the index file. It is **238MB**.

- Limiting the memory to 2MB memory with 5GB data file size, it takes **45899ms** to create the index file which is approximately **46 seconds.**
- Calculating the index file size = Number of I/O which is 59627 * 4000(number of block size) = 238508000 **~ 238MB**

We got the total number of I/O by adding the number of read blocks (1249046) to number of blocks to store the index (59709)

- The total number of I/Os = n of reads + n of writes to construct the index = 1249046+59709 = **1308673.**

Now, when we ran the program to find the records of age 18. It gives us **616184** which is the number of people of age 18 the program finds. Each time it finds the record with age 18, it also adds the salary to variable sum. (Later it divides the sum by number of people to find the average salary). To calculate the number of I/Os, we increment the IO whenever the bucket or block is read. This gives us **489380 for the total I/O to locate age 18.**

- Average yearly income generated for age 18 is **97095.22012418369 as seen in**

Figure 4.

- The time it takes to generate age 1s **69663ms** which approximately 69 seconds.

2) Analyzing Result using 5MB memory size on 5MB data file.

```
file size = 4996186112 bytes.
49961861 records.
number of blocks required to READ = 1249046
Program started...
Index File has been constructed...
Time taken = 55087 ms
Number of I/O WRITE = 59627
Number of blocks to store the index = 59709
Total number of I/O = 1308673
1. For one age:
2. Print average income for every 10 ages:
1
Enter the age 18-99: 18
Number of people of age 18 = 616184
The Average yearly Income = 97095.22012418369
Time taken = 125452 ms
Number of I/O = 489380
1. For one age:
2. Print average income for every 10 ages:
```

Figure 5: Pictorial view showing what it displays when running the program using 5GB data. Limiting the memory size to 5MB.

*The time performance for the program using 5MB memory size does not really differs from 2MB because our program reads and write in blocks. it also processes data block by block.