

Fall 2020: CSCE 608 Database Systems

Project Report for Course Project II

Submitted By:

Flavia Ratto

130004853

Overview:

In this project, we were asked to implement two of the key algorithms which have been widely used in database management systems.

1. Implementation of B+ trees;
2. Implementation of hash-based algorithms for relational algebraic operations

Both the parts involve the following to be done –

- Data Generation
- Algorithm Implementation
- Experiments

B+ Trees:

In this part of the project, I have implemented B+ Trees in which the order of the tree is the parameter given by the user. The data which consists of only search key and no other attribute is randomly generated. Using this randomly generated data, the algorithm will be able to create a **dense and sparse tree** for the order provided.

On this B+ tree, various operations can be performed. This includes –

- Search
- Range Search
- Insertion
- Deletion

Finally, these operations have been tested on the B+ tree algorithm I have implemented using randomly generated data for creating both dense and sparse trees of different orders. The algorithm displays/prints the results of all these experiments.

Join Based on Hashing:

In this part of the project, I have implemented the two-pass join algorithm based on hashing. It involved two relations $R(A, B)$ and $S(B, C)$ as input and I have computed the natural join $(R(A, B) \bowtie S(B, C))$ on these two tables using **two-pass join algorithm based on hashing**.

The data for the tables is also randomly generated and stored in the disk. We are to consider a virtual **main memory M of 15 blocks**. Each block in the main memory is assumed to hold up to **8 tuples**. The virtual disk is assumed to be unlimited. However, the join operation can only be performed in the main memory. So, my algorithm is implemented in such a way that it takes into consideration the main memory.

I have generated data for relation $S(B, C)$ and stored it in the virtual disk. Also, I have created two tables $R_1(A, B)$ for experiment 1 and $R_2(A, B)$ for experiment 2. I have created procedures for reading from the virtual disk to the main memory and writing from the main memory to the virtual disk. Then, I have implemented an algorithm to hash the tuples to the appropriate buckets and perform natural join operation based on hashing.

The implemented is tested using two different experiments. The result of the join operation is printed as well as the number of disk I/O operations that the algorithm uses is computed.

B+ Trees:

Generating Data –

The B+ tree that I would be creating in this project requires me to generate 10,000 different records from the range of 100,000 and 200,000. These 10,000 different records are the trees for the B+ Tree. I have created the procedure ***getRandom(int min, int max, int num)*** in order to generate the data as per our requirement.

Building B+ Trees –

Using the 10,000 records generated, I have 2 procedures implemented to build a dense B+ tree ***createDenseTree(List key, int m)*** and sparse B+ tree ***createSparseTree(List key, int m)***. These procedures take the order m and the random keys (output of getRandom) as input. The keys are given as input in sorted order. The dense B+ tree is a tree which has all its non-root nodes as full as possible, while the sparse B+ tree has all its nodes are filled to its minimum capacity.

In order to build the dense tree of order m, I started by creating **leaf nodes** filled with **m keys** each. The 10000 keys are distributed among these leaf nodes in an ascending order. These leaf nodes are **doubly linked list** nodes. These leaf nodes then have internal parent nodes created which are also filled to **m** keys. This continues till we finally have just one parent node created which is the root node for this B+ tree.

In a similar way to build a sparse tree of order m, I started by creating **leaf nodes** filled with $\lfloor (m + 1)/2 \rfloor$ **keys** each. The 10000 keys are distributed among these leaf nodes in an ascending order. These leaf nodes are also **doubly linked list** nodes. These leaf nodes then have **internal parent nodes** created which are filled to $\lceil (m+1)/2 \rceil - 1$ **keys**. This continues till we finally have just one parent node created which is the root node for this B+ tree.

I have a ***printtreeroot()*** function which I use to print the whole B+ tree in a **Depth First Search format**.

Implement B+ Tree operations –

I have implemented the following operations in this part: Insertion, search, range search and deletion operations.

1. Search –

This function ***search(int key)*** takes the key value as input and traverses the tree from root node to leaf using the ***searchInNode(Node node, int key)*** function to find the node where the key value should exist. Then, I loop through the node to check if the key exists in the node. If it exists, I will print that the key exists, else I will print that the key does not exist.

2. Range Search –

The range search function is implemented as ***search(int keyStart, int keyEnd)***. This function takes two keys as input. Value of keyStart is smaller than keyEnd. First, I start by traversing the tree to find the leaf node where key start could exist in the tree. Then, I check all the leaf node from there and include all the keys that fit within this range in the result. I will stop when I find a key value

greater than keyEnd or I reach the last leaf node. The result is a list of all keys within the range. If no key exists in the given range, I print that as the output.

3. Insertion –

I have implemented the function called **insert_d(int key)** in order to insert a key in the B+ tree. The key value is taken as input. First, I check if the input key already exists in the tree by calling the search function **search(int key)** implemented above. I am only allowing unique values in the B+ tree. Therefore, if the input key value already exists in the tree I print that the key already exists in the tree and return from the function.

If the key is not already in the tree, I will go ahead with inserting it. First, I traverse through the tree using the function **searchInNode(Node node, int key)** from the root node to the leaf to find the correct node that I want to insert the key. Then, I find the exact location where I would insert the key in the index node.

Then, I check if the insertion in the leaf node caused the number of keys in the leaf to be more than m. If **leaf overflow** happens, I call the **splitLeafNode(Node node)** function to split this leaf node into two leaves and recursively call **splitInternalNode** function in order to restructure the parent (internal) nodes.

The operation ends when the tree is restructured appropriately with no overflow nodes and with the correct parent nodes.

4. Deletion –

In the delete function also that I have implemented **delete(int key)**, I traverse through the tree from the root to the leaf to find the node where the key value would exist. If the key does not exist in that node, I print that the key does not exist in the tree. However, if the key value exists in the tree, I delete the key and check for the number of keys in the node.

If the number of keys go below the minimum requirement, I check if I can **borrow keys** from the left or right sibling without causing them to go below minimum capacity.

However, I have chosen not to implement the procedure of **Coalescing**. This procedure is very costly and not very practical. As coalescing would cause you to combine the leaf nodes, sometimes to full capacity and then also restructure the internal parent nodes. However, it is very much possible that a new insertion that would happen soon enough would need you to split this same merged node and restructure everything again. This uses up a lot of I/O's and is very impractical. In practice, if deletion causes the nodes capacity to go below minimum that is okay as it might be filled up soon enough. Also, having nodes below minimum capacity would not harm the tree structure.

Both the insertion and deletion procedures when executed, will print all the nodes involved in the operation, with their before and after state.

Join by Hashing:

Generating Data –

The relation S (B, C) is supposed to contain 5000 tuples, where B is the key attribute and C could be of any type. Therefore, I generated 5,000 random integers in the range of 10,000 to 50,000 using **getRandom(int min, int max, int num)**. Since, B is the key attribute, the random integers are all unique. The values for C

are just 5,000 randomly generated integers using the function ***getRandomNonUnique(int min, int max, int num)***.

For experiment 1, the relation R (A, B) is supposed to contain 1,000 tuples, where B is the attribute whose values are to be picked randomly from S (B, C). Therefore, I generated 1,000 random integers from the B values I have generated for S (B, C). I have called the function ***getRandomFromTable(int num, List B)*** for this. Here, B need not be all unique. Hence, I have allowed duplicates if generated by the function. The values for C are just 1,000 randomly generated integers. Additionally, I also need to generate 20 random integers from B for printing the result of the natural join. Here, I am picking 20 random unique values from the B values of R1.

For experiment 2, the relation R (A, B) is supposed to contain 1,200 tuples, where B is the attribute whose values are to be picked randomly from integers between 20,000 and 30,000. Therefore, I generated 1,200 random integers in this range using the function ***getRandomNonUnique(int min, int max, int num)***, also allowing duplicates. The values for C are again just 1,200 randomly generated integers.

All these relations are stored in disk files as **Table_S, Table_R1 and Table_R2**.

Implementing Virtual Disk and Main Memory –

We are said to assume we have a main memory M of 15 blocks and an unlimited virtual disk. Each block in the main memory can hold 8 tuples of the relation. The total tuples that the main memory can hold is $15 * 8 = 120$. Therefore,

- Relation S consists of $B(S) = \frac{5000}{8} = 625$ blocks.
- Relation R1 in experiment 1 consists of $B(R1) = \frac{1000}{8} = 125$ blocks.
- Relation R2 in experiment 2 consists of $B(R2) = \frac{1200}{8} = 150$ blocks.

The relations are **read and written in blocks** by the program. The program has procedures to read and write the relations to and from the main memory using the **java.io** packages in JAVA. Also, when these procedures are called I take into account the main memory restriction and limit the amount of data that should exist in the main memory at a time in order to perform further operations on it. No more than 15 blocks of data is present in the main memory at a time.

Implementing Hash function –

Here, in order to implement hashing I have used the **hashCode()** method in JAVA. This method returns an integer value. Objects that are equal (according to their equals()) will return the same hash code. If this method is invoked on the same object more than once during an execution of the application, it will consistently return the same hash value. This value needs not remain consistent from one execution of an application to another execution of the same application. However, it is not necessary that objects that are different will generate different hash codes.

Hence, this hashCode() method is appropriate for implementing join by hashing. I apply this hashCode method on all values of my **join attributes** in both the tables. Therefore, all the join attributes that have the same value would return the same integer value when hashCode() is applied to it. I then modulo divide this integer value by M and take the remainder as output. This value ranges between 0 to M-1. Then based on this value, I assign the tuples to buckets which also has indices from 0 to M-1. Therefore, I am using **M buckets** in this algorithm.

























Implementing the join algorithm –

Here, the hash-based join is a **two-pass algorithm**. The general framework of the two – pass hash – based algorithm is as below –

1. Phase 1: Making hash buckets
2. Phase 2: Bucketwise operation

So, for **phase 1** as described in the previous section I first create $M=15$ hash bucket files which are present in the disk. I load the tuples of the table in blocks up to the capacity of the main memory, i.e. 15 blocks. I apply the implemented hash function as described in the previous part on the join attribute, which is B in our case. Accordingly, every tuple gets a bucket index computed for it. According to this index, I write the tuples to its appropriate bucket file in blocks to the bucket files in the disk. Once this is completed for these 15 blocks, the main memory is cleared, and the next 15 blocks of the relation are read, and the process continues. This is continued till the entire relation has been read, computed a bucket index and written to the disk in its respective hash bucket file.

The **hash bucket files** are created as below in the disk. The files prefixed Hash_R_1_ are for Relation R1, Hash_R_2_ are for Relation R2 and Hash_S_ are for Relation S –

 Hash_R_1_7	
 Hash_R_1_8	
 Hash_R_1_9	
 Hash_R_1_10	
 Hash_R_1_11	
 Hash_R_1_12	
 Hash_R_1_13	
 Hash_R_1_14	
 Hash_R_2_0	
 Hash_R_2_1	
 Hash_R_2_2	
 Hash_R_2_3	

For **phase 2**, I have to implement **natural join on the two tables**. The tables R1 and R2 are smaller tables as compared to table S. Table R1 and R2 has 125 and 150 blocks respectively. If they are hashed into 15 buckets (M), each bucket would have on an average 9 and 10 blocks respectively. This is **less than the main memory limitation** of $M = 15$ blocks. Therefore, each bucket of table R1 or R2 can be loaded into the main memory completely. Since, **M is large enough to hold the smaller bucket**, I can apply **one pass algorithm** on the pairs of buckets R_i and S_i .

Here, I read the entire bucket R_i in the main memory in blocks. Then, I load the tuples in S_i in blocks in the main memory. For every tuple t in S_i , I find the tuples in R_i that have the same join attribute B – value. I join them with t and write it in the result file. This is done till all the tuples in the S_i bucket are loaded and checked for. This entire process is then again done for the next pair of R_i and S_i buckets. The reason that join operations take place in bucket pairs is because, the hash function that is implemented will hash objects that are equal to the same index. Hence, a B-attribute value say x in Table R will be hashed to a bucket with index say 2 of R files, then the same B-attribute value x in table S will also be hashed to a bucket with index 2 of S files. This happens during the entire execution of the application.

Hence, the result output gives us the list of tuples which is the result of the natural join of the two tables R and S.

Analysis:

B + Trees –

A lot of experiments have been conducted for B+ tree. In order to conduct these experiments, I have created 2 functions *dense(int m)* and *sparse(int m)* in the main class. These functions take the order of the tree as input and generates random data for keys. It then initializes a B+ tree and creates a dense or sparse tree as per the function called of the specified order. I have also printed the tree. The operations as per the experiments asked for us to conduct have been called in the function. Some additional experiment have also be included in the function. The results of this function is written in a separate text file which generates when the program runs.

The main method calls the function *dense(13)*, *dense(24)*, *sparse(13)* and *sparse(24)* and produces 4 output text files.

In the **dense** B+ trees, almost all the nodes are of full capacity. **Insertion** when performed on these nodes in almost all cases **caused the nodes to overflow** hence requiring consequent splitting and restructuring. On the other hand, **deletion** operation when performed on these nodes would happen without any problem of the key in the **node going below minimum capacity**.

However, in the **sparse** B+ tree almost all nodes are of minimum capacity. **Insertion** when performed on these nodes caused **no problem of overflow**. But, **deletion** caused nodes to have the number of keys **going below minimum capacity**. In some cases, the nodes would borrow keys from siblings and restructure the internal parent nodes.

We had a very **high number of records** (10,000) and the values were in the range of 100,000 to 200,000. Having a tree structure to arrange this huge amount of data made all the operations that we performed – search, range search, insertion and deletion very **quick**. Basically, B+ trees **speed up the search process**.

Join Based on Hashing –

We are asked to conduct two experiments. On running this program, the table files, hash bucket files and result files are generated. Also, the required output is printed on the console along with the IO computation. The output file is created as below –

```
} 1_Table_R1_Table_S_join      ✓  
} 2_Table_R2_Table_S_join      ✓
```

i) Experiment 1

Here, since R has 1000 tuples in which the join attribute is picked from the join attribute in S, we will have **1000 tuples in the resultant output**. The B-value in S are all unique.

In order to print the result, we are asked to **pick 20 random B** values. I am assuming that we have to pick 20 random unique B values from the **R1 table's B-value**. Therefore, I get **at least 20 tuples printed**.

However, if I were to pick the random 20 B values from the B values in table S, I could possibly get 0 tuples. This is because the table R might have not picked B values from the ones we randomly picked for printing.

Number of tuples in Table S: 5000

Number of blocks in Table S B(S): 625

Number of tuples in Table R1: 1000
Number of blocks in Table R1 B(R1): 125

Calculating number of disk I/O's

Reading Table S from disk for hashing: B(S)
Reading Table R1 from disk for hashing: B(R1)
Writing to disk after hashing Table S: B(S)
Writing to disk after hashing Table R1: B(R1)
Reading all bucket pair in main memory for computing natural join: B(S) + B(R1)
Total Number of Disk I/O's by the algorithm: **$3*(B(R1) + B(S))$**

ii) Experiment 2

1200 tuples are generated here for table R. These values are in the range of 20,000 and 30,000, allowing duplicates. This is within the range of the B - values for table S. The **result** of the join in this case could be **between 0 tuples to 1200 tuples**. 0 if none of the tuples in R match with S. 1200 tuples if all the 1200 tuples in R have a match in S.

Number of tuples in Table S: 5000
Number of blocks in Table S B(S): 625
Number of tuples in Table R2: 1200
Number of blocks in Table R2 B(R2): 150

Calculating number of disk I/O's

Reading Table S from disk for hashing: B(S)
Reading Table R2 from disk for hashing: B(R2)
Writing to disk after hashing Table S: B(S)
Writing to disk after hashing Table R2: B(R2)
Reading each bucket pair in main memory for computing natural join : B(S) + B(R2)
Total Number of Disk I/O's by the algorithm: **$3*(B(R2) + B(S))$**

Discussion:

- This project was quite a challenging one for me in terms of understanding the concepts and very intricate details of the algorithms as well and programming for it. It gave me a better and deeper understanding of the algorithms.
- I gained an experience of generating data as per the requirement and working with large amount of data. I got to develop algorithms from scratch which made me understand the workflow of the algorithms better.
- In B+ tree, I had to create trees of any order m of two types – dense and sparse. All this was done while taking care of the minimum and maximum number of keys requirement. I was able to create high order trees like m = 13 and 24 which I had to experiment with in this project.
- It was a bit of a challenge for me to understand an optimal way to create the two trees as there isn't really an algorithm for creating these trees. But then the approach suggested by the professor in class was really very helpful for me to implement a function to create these trees.
- Also, printing the results and presenting the output in a presentable way for B+ tree was a bit difficult. So, I had to spend some time on finding a way to do so. And I also decide on creating

output text files to save the results of all the experiments conducted for a particular type of tree of order m .

- Also, a lot of details had to be taken care of while implementing insertion and deletion, like node overflows in leafnode and internal nodes, changing parent keys and children nodes, etc.
- Since we are working with huge data and high order tree, it was difficult to know if all the operations are happening correctly. Therefore, I started with testing the procedures for less number of keys and low order trees and then increased the number of keys and the order of the tree. Also, printing the output at various intermediate steps helped a lot in understanding how the algorithm's flow is and where the error is actually taking place. This really helped debugging the program.
- Also, in the join by hashing part of the project I learned about choosing the write hash function and implanting it for the range of my algorithm.
- I also learned to simulate the disk I/O's and have my program read from the files in disk and write them back in the disk all in Blocks.
- Also, learning about what the hashCode function does and how we use that in the phase two of the algorithm to perform join was something I understood better when I implemented the algorithm and seen intermediate results. The reasons to why we consider bucket pairs of the same index at a time was much clearer when I learned more of the hash function.
- Hence, it was better understood why hashing the keys in buckets made it much better to perform join operation of tables that are larger than the main memory.

Overall, there were quite some challenges I faced while implementing these algorithms. But I got to learn a lot of new things in the entire process and made my concepts and understanding a lot clearer and more practical.

Thank you!