CSCE-608 Database Systems
Fall 2020
Project # 2

Name: Sun Yul Lee
UIN: 824007724

Part 1
Implementation: B+Tree

I used Java to implement B+ Trees.

Part 1. Data Generation

To generate records which only contains a search key with no other attributes, I created an array of integer with the size of 10000. I used Random package of Java to generate random integer values between 100000 and 200000. The integer values are sorted and not duplicated.

Part 2. Building B+trees

In this part, I created class Node and class BPTree.

- Node
  - Attributes
    IS_LEAF – indicates whether the node is a leaf or not
    parent – indicates which node points this node
    keys – holds keys in the node
    ptrs – holds pointers in the node
    occupied – number of keys in the node

  - Functions
    Node(int) – create an object of Node. It initializes all attributes of the object.
    Node(Node) – copy an object of Node.
    insert() – insert key and pointer in the node.
    delete_key() – delete key and corresponding pointer in the node.

- BPTree
  - Attributes
    order – order of the tree.
    root – the root node of the tree.

  - Functions
    BPTree() – create an object of B+ Tree. It set order value of the object and construct the tree.
    construct() – construct the tree. It creates dense tree or sparse tree based on the input.
    search() – search a key in the tree.
    range_search() – search keys in the given range.
    insert() – insert a new key in the tree.
    delete_key() – delete a key in the tree.

Part 3. Operations on B+trees

- Search

  To implement search, I created function search() in BPTree.Java. The function takes the key and the node as input. Initially, the root of the tree is passed as the node in input. First, it checks whether the input node is leaf or not. If the node is a leaf, it searches for a key of the node which is the same as the input key. If there is a key, it returns the position of the key in the node. If there is no such key, it returns 0. If the input node is not a leaf it uses recursion. It checks the keys in the node. If there is a key which is greater than the input key, it finds the first key which is greater than the input key. Then it passes the input key and the node which is pointed by the pointer at the position of found key to the function search(). If there is no such key, it passes the input key and the node which is pointed by the last pointer of the input node to the function search().

- Range Search

  To implement range search, I created function range_search() in BPTree.Java. The function takes the begin key, end key, and node as input. Initially, the root of the tree is passed as the node in input. First, it checks whether the input node is leaf or not. If the node is a leaf, it searches for the first key which is greater than the begin key. From that position, it searches all keys which is smaller than the end key. If all keys in the node are smaller than the end key, it checks the node which is pointed by the last pointer of the node. Then it returns the list of keys. If the node is not a leaf, it uses recursion. It checks the keys in the node. If there is a key which is greater than the begin key, it finds the first key which is greater than the input key. Then it passes the begin key, end key, and the node which is pointed by the pointer at the position of found key to the function range_search(). If there is no such key, it passes the begin key, end key, and the node which is pointed by the last pointer of the input node to the function range_search().

- Insertion

  To implement insertion, I created function insert() in BPtree.Java. The function takes the key and the node as input. Initially, the root of the tree is passed as the node in input. First, it checks whether the input node is leaf or not. If the node is a leaf, it checks whether the node has a space for new key. If there is a space, it inserts the key in the node and returns 0 and null node. If there is no space in the node, it creates a new leaf. Then it re-arranges the keys in the node and a new key in the original node and the new node. It set the last pointer of the original node to the new node. If the original node is the root, it creates a new root. And it set the original node and the new node as children of the new root. The key in the new root is the first key in the new node Then the function returns 0 and null node. If the original node is not the root, the function returns the first key of the new node and the new node. If the input node is not a leaf, it uses recursion. First, it checks the keys in the node. If there is a key which is greater than the input key, it finds the first key which is greater than the input key. Then it passes the input key and the node which is pointed by the pointer at the position of found key to the function insert(). If there is no such key, it passes the input key and the node which is pointed by the last pointer of the input node to the function insert(). If returned node is null, the function returns 0 and null node. If the returned node is not null, it checks whether there is a space for a new key in the node. If there is a space, it insert a new key in the node and returns 0 and null node. If there is no space, it performs the same process when the input node is a leaf. It creates a new leaf, re-arranges the keys in the original node and a new key in the original node and the new node, and sets the last pointer of the original node to the new node. If the original node is the root, it creates a new root, sets the original node and the new node as children of the new root, and inserts the first key of the new node in the new root. Then the function returns 0 and null node. If

the original node is not the root, the function returns the first key of the new node and the new node.

- Deletion
  To implement deletion, I created function delete_key() in BPTree.Java. The function takes the key and the node as input. Initially, the root of the tree is passed as the node in input. First, it checks whether the input node is leaf or not. If the node is a leaf, it deletes the input key from the node. If there is no such key in the node, it returns false. If it deletes the key, it checks whether the node is root or has less pointers than the minimum number of pointers. If it is, the function returns false. If it is not, the function returns true. If the input node is not a leaf, it uses recursion. First, it checks the keys in the node. If there is a key which is greater than the input key, if there is a key which is greater than the input key, it finds the first key which is greater than the input key. Then it passes the input key and the node which is pointed by the pointer at the position of found key to the function delete_key(). If there is no such key, it passes the input key and the node which is pointed by the last pointer of the input node to the function delete_key(). Let the passed node as p. If the function returns false, then it returns false. If the function returns true, it checks adjacent sibling. If p is pointed by the last pointer of the input node, the adjacent sibling is the node on the left side of p. Else, the adjacent sibling is the node on the right side of p. If the sibling node has more than the minimum number of pointers and it is on right side, it moves first pointer and first key of the sibling node to p. if the sibling node has more than the minimum number of pointers and it is on left side, it moves the last key and second pointer from the back of the sibling node to p. it also update the sequence pointer. Then it returns false. If the adjacent sibling does not have more than the minimum number of pointers, it combines p and the adjacent sibling. it also updates the sequence pointers. After that, if the input node is the root with one pointer p, p becomes a new root and it returns false. If the input node has at least the minimum pointers or it is the root, it returns false. Else it returns true.

Experiments: B+Tree

(a) Random package of Java was used to generate random integers between 100000 and 200000. Set was used to avoid duplication. Then the array of the integers was created and sorted.

(b) Then with the generated keys, 2 dense B+ trees and 2 sparse B+ trees were created. One of each type has order of 13 and the other has order of 24. The message after construction is in Figure 1(a). To build a dense tree, first I made a list of full leaf nodes with the keys. If the last node has less keys than the minimum number of keys, the adjacent node gives required node to the node so that keys in the node is not less than the minimum number of keys. Then, I made a loop that creates a list of parent nodes which take the nodes in the previous list as pointers. The parent nodes also had the keys as full as possible. In the loop, I did not consider the case where the last non-leaf node has less keys than the minimum number of keys since the case in the experiments do not have such case. The loop was finished when the number of parent nodes become one. Then the parent nodes became a root. The process of creating a sparse node was very close to this process. First, I made a list of sparse leaf nodes with the keys. If the last node has less keys than the minimum number of keys, each key in the node was distributed into previous nodes. Then the loop that creates a list of parent nodes construct the tree until it reached the root. Instead of creating the parent nodes

which had the keys as full as possible, in this process, the keys were as sparse as possible.

```
------------------------------------------------------------
                    Project 2

    Please type one of the options:
        Tree - B+Tree
        Hash - Join based on Hashing
        Exit - Close the program

------------------------------------------------------------

    Your choice:
tree
root: [125411, 151327, 176643, 0, 0, 0, 0, 0, 0, 0, 0, 0]
--- Dense B+Tree of order 13 has created ---
root: [105902, 111791, 117755, 123903, 129752, 135868, 141892, 148080, 154486, 160547, 166307, 172067, 178237, 184054, 189893, 195819, 0, 0, 0, 0, 0, 0, 0, 0]
--- Dense B+Tree of order 24 has created ---
root: [135617, 171809, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
--- Sparse B+Tree of order 13 has created ---
root: [119960, 140474, 161224, 181254, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
--- Sparse B+Tree of order 24 has created ---
```

Figure 1(a) – Message after construction of four trees.

(c) The output of one test run for all below sections are saved in Result.txt file.

1. I applied two randomly generated insertion operations on each of the dense trees. The message after the first insertion operation on the dense tree with the order of 13 is shown in Figure 1(b). The figure shows that my implementation works properly. The insertion key was 137800. It searched the root. The key was between 125411 and 151327. It searched the pointer between two keys. Now the key was between 136265 and 138237. It searched the pointer between these two keys. Now the key was between 137734 and 137942. It searched the pointer between these two keys. Now the search reached a leaf. Since the node is full, it re-arranged the keys into two nodes. Then it sent the first key in the second node to its parent node. Since the parent node was also full, it re-arranged the keys into two nodes. This process repeated until there was a space in the parent node.

```
--- Insert in Dense B+Tree of order 13 ---
--- key: 137800
[125411, 151327, 176643, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Location: 1
[127192, 128874, 130731, 132474, 134376, 136265, 138237, 140038, 141760, 143545, 145481, 147394, 149246]
Location: 6
[136383, 136563, 136739, 136885, 136972, 137069, 137161, 137343, 137499, 137631, 137734, 137942, 138071]
Location: 11
[137734, 137763, 137787, 137838, 137848, 137861, 137866, 137893, 137909, 137916, 137927, 137930, 137941]
--- Above node is a leaf ---
Before: [137734, 137763, 137787, 137838, 137848, 137861, 137866, 137893, 137909, 137916, 137927, 137930, 137941]
After: [137734, 137763, 137787, 137800, 137838, 137848, 137861, 0, 0, 0, 0, 0, 0]
       [137866, 137893, 137909, 137916, 137927, 137930, 137941, 0, 0, 0, 0, 0, 0]
Before: [136383, 136563, 136739, 136885, 136972, 137069, 137161, 137343, 137499, 137631, 137734, 137942, 138071]
After: [136383, 136563, 136739, 136885, 136972, 137069, 137161, 0, 0, 0, 0, 0, 0]
       [137343, 137499, 137631, 137734, 137866, 137942, 138071, 0, 0, 0, 0, 0, 0]
Before: [127192, 128874, 130731, 132474, 134376, 136265, 138237, 140038, 141760, 143545, 145481, 147394, 149246]
After: [127192, 128874, 130731, 132474, 134376, 136265, 137343, 0, 0, 0, 0, 0, 0]
       [138237, 140038, 141760, 143545, 145481, 147394, 149246, 0, 0, 0, 0, 0, 0]
Before: [125411, 151327, 176643, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
After: [125411, 138237, 151327, 176643, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 1(b) – Result of the first insertion on dense tree with the order of 13.

2. I applied two randomly generated deletion operations on each of the sparse trees. The message after the first deletion operation on the sparse tree with the order of 13 is shown in Figure 1(c). Key searching process works the same as that in insertion process. When it reached a leaf. It searched a key which is the same as searching

key. In the test, randomly generated search key did not exist in the tree. Therefore, the result showed that there is no such key.

```
--- Delete in Sparse B+Tree of order 13 ---
--- key: 197195
[135617, 171809, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Location: 2
[176314, 180753, 185063, 189565, 194041, 198582, 0, 0, 0, 0, 0, 0, 0]
Location: 5
[194516, 195073, 195609, 196077, 196578, 197363, 197933, 198582, 0, 0, 0, 0, 0]
Location: 5
[196658, 196803, 196836, 196892, 196970, 197130, 197290, 197363, 0, 0, 0, 0, 0]
Location: 6
[197130, 197137, 197170, 197185, 197244, 197265, 197271, 0, 0, 0, 0, 0, 0]
--- Above node is a leaf ---
There is no such key
```

Figure 1(c) – Result of the first deletion on sparse tree with the order of 13.

I ran test several times to find a case where there is a matching key for deletion. The message in the Figure 1(d) shows the process of deletion. It shows the deletion operation on sparse tree with the order of 13. It searched the matching key at a leaf node. Then it deleted the key from the node. Since the tree is sparse, the node can be merged with its adjacent sibling. Then it removed the empty node and updated the parent node.

```
--- Delete in Sparse B+Tree of order 13 ---
--- key: 115257
[135204, 171371, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Location: 0
[104300, 108797, 113203, 117638, 122002, 126400, 130471, 135204, 0, 0, 0, 0, 0]
Location: 3
[113735, 114388, 114860, 115354, 115876, 116523, 117074, 117638, 0, 0, 0, 0, 0]
Location: 3
[114921, 114977, 115035, 115125, 115191, 115234, 115294, 115354, 0, 0, 0, 0, 0]
Location: 6
[115234, 115239, 115240, 115249, 115257, 115259, 115270, 0, 0, 0, 0, 0, 0]
--- Above node is a leaf ---
Before: [115234, 115239, 115240, 115249, 115257, 115259, 115270, 0, 0, 0, 0, 0, 0]
After: [115234, 115239, 115240, 115249, 115259, 115270, 0, 0, 0, 0, 0, 0, 0]
Before: [115234, 115239, 115240, 115249, 115259, 115270, 0, 0, 0, 0, 0, 0, 0]
After: [115234, 115239, 115240, 115249, 115259, 115270, 115294, 115297, 115308, 115313, 115322, 115330, 115334]
Before: [115294, 115297, 115308, 115313, 115322, 115330, 115334, 0, 0, 0, 0, 0, 0]
After: Empty
Before: [114921, 114977, 115035, 115125, 115191, 115234, 115294, 115354, 0, 0, 0, 0, 0]
After: [114921, 114977, 115035, 115125, 115191, 115234, 115354, 0, 0, 0, 0, 0, 0]
```

Figure 1(d) – Result of the deletion on sparse tree with the order of 13.

3. I applied five additional randomly generated insertion or deletion operations on each of the dense and sparse trees. All the insertion and the deletion worked properly. The whole result of the testing is saved in Result.txt file.

4. I applied five randomly generated search operations on each of the dense and sparse trees. The message after the first search operation on the dense tree with the order of 13 is shown in Figure 1(e). The search process worked the same as that in insertion or deletion operations. It kept searching until it reached at a leaf. Then

it searched a key in a leaf which is the same as the input key. In the test, randomly generated search key did not exist in the tree. Therefore, the result showed that there is no such key.

```
--- Search in Dense B+Tree of order 13 ---
--- key: 177033
[114429, 125411, 138237, 151327, 163862, 163862, 176643, 0, 0, 0, 0, 0, 0]
Location: 7
[178422, 180184, 181933, 183721, 185451, 187314, 189223, 191002, 192674, 194516, 196215, 198285, 0]
Location: 0
[176747, 176912, 177077, 177190, 177365, 177462, 177547, 177681, 177832, 177984, 178152, 178237, 178295]
Location: 2
[176912, 176919, 176930, 176936, 176940, 176947, 176988, 176989, 176999, 177016, 177044, 177048, 177050]
--- Above node is a leaf ---
There is no such key
```

Figure 1(e) – Result of the first search operation on dense tree with the order of 13.

The fourth search operation on dense tree with the order of 13 showed the key found scenario. The message after the operation is shown in Figure 1(f). When it reached a leaf, there were the same key in the tree. It printed the location of key in the node, index of 4 in this case.

```
--- Search in Dense B+Tree of order 13 ---
--- key: 171658
[114429, 125411, 138237, 151327, 163862, 163862, 176643, 0, 0, 0, 0, 0, 0]
Location: 6
[163862, 165827, 167637, 169372, 171146, 172896, 174790, 0, 0, 0, 0, 0, 0]
Location: 5
[171273, 171391, 171463, 171594, 171767, 171874, 171976, 172090, 172204, 172332, 172444, 172614, 172702]
Location: 4
[171594, 171616, 171625, 171646, 171658, 171662, 171678, 171685, 171715, 171729, 171734, 171755, 171757]
--- Above node is a leaf ---
There is a key at 4
```

Figure 1(f) – Example of key found scenario on dense tree with the order of 13.

5. I applied one range search operation on each of the dense and sparse trees. I set the begin key and the end key to 150000 and 150030 for all operations. The message after the range search operation on the dense tree with the order of 13 is shown in Figure 1(g). It used the begin key to do search operation. When it reached a leaf, it searched a key which is greater than the begin key. From that key, it found all keys which is smaller than the end key.

```
--- Range Search in Dense B+Tree of order 13 ---
--- keys: 150000, 150030
[114429, 125411, 138237, 151327, 163862, 163862, 176643, 0, 0, 0, 0, 0, 0]
Location: 3
[138237, 140038, 141760, 143545, 145481, 147394, 149246, 0, 0, 0, 0, 0, 0]
Location: 13
[149412, 149617, 149733, 149838, 149981, 150106, 150253, 150381, 150483, 150661, 150824, 150966, 151200]
Location: 5
[149981, 150002, 150003, 150025, 150033, 150039, 150041, 150069, 150075, 150086, 150094, 150097, 150104]
--- Above node is a leaf ---
-- Results ---
[150002, 150003, 150025]
```

Figure 1(g) – Result of the range search operation on dense tree with the order of 13.

In the test, I had a output which showed a scenario where the operation search the keys in next node. In the range search operation on sparse tree with the order of 13, last key in the node was smaller than the end key. So, it checked the node which

```
--- Range Search in Sparse B+Tree of order 24 ---
--- keys: 150000, 150030
[119960, 140474, 161224, 181254, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Location: 2
[142032, 143545, 145114, 146884, 148408, 150106, 151839, 153524, 155145, 156833, 158268, 159775, 161224, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Location: 5
[148582, 148684, 148863, 148942, 149052, 149175, 149353, 149453, 149650, 149773, 149869, 150002, 150106, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Location: 11
[149869, 149874, 149877, 149879, 149904, 149918, 149929, 149935, 149945, 149953, 149964, 149981, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
--- Above node is a leaf ---
[150002, 150003, 150025, 150033, 150039, 150041, 150069, 150075, 150086, 150094, 150097, 150104, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
-- Results ---
[150002, 150003, 150025]
```

the sequence pointer of the node was pointing. The message after the operation is shown in Figure 1(h).

Figure 1(h) – Result of the range search operation on sparse tree with the order of 24

Part 2
Implementation: Join based on Hashing

Like implementing B+ Trees, I used Java to implement hash_based algorithm.

Part 1. Data Generation

To generate tuples for a relation S, I created a class Tuple which can hold two values, one integer value and one String value. Then, I created an array of integer with the size of 5000. I used Random package of Java to generate random integer values between 10000 and 50000. The integer values are not duplicated. After generating the array of integer value, I created a list of Tuple and used the integer array to set integer value of Tuples. A String value of Tuple was set as "si" where i is an index of the Tuple. The Tuples were then stored in the object of virtual disk.

Part 2. Virtual Disk I/O

In this part, I created class VDisk for virtual disk and VMM for virtual main memory.

- VDisk
  - Attributes
    block_size – maximum number of tuples that one block can hold.
    num_blocks – number of blocks in virtual main memory. I used this value in VDisk to calculate the number of buckets (num_blocks - 1).
    data_R – holds tuples of R in blocks.
    data_S – holds tuples of S in blocks.
    hash_R – holds tuples of R in buckets.
    hash_S – holds tuples of S in buckets.

  - Functions
    VDisk() – create an object of virtual disk. It initializes all attributes of the object.
    write() – write tuples into blocks.

read() – read a block from the disk. This function is used for sending a block from virtual disk to virtual main memory before hashing. A block which is read is removed from the disk.
hash_write() – write a block into buckets.
bucket_read() – read a bucket from disk with the given hash index. This function is used for sending a bucket of small relation(R) to virtual main memory.
hashed_block_read() – read a block in a bucket from disk. This function is used for sending a block of bucket in large relation(S) to virtual main memory.
is_empty() – check whether block of the relation exists or not.
bucket_size() – return the number of blocks in a bucket with given hash index.
print_hash() – print the number of blocks in the buckets. (Used for experiment)

- VMM
  - Attributes

    block_size – maximum number of tuples that one block can hold.
    num_blocks – number of blocks in virtual main memory.
    data – holds tuples in blocks

  - Functions

    VMM() – create an object of virtual main memory. It initializes all attributes of the objects.
    hash_size() – return number of total buckets (num_blocks - 1).
    write_at() – write a block at a certain position of the virtual main memory.
    read_at() – read a block from a certain position of the virtual main memory.
    hash_function() – hash function for calculating the hash index.
    hash() – hash the tuples in first block into the rest blocks. It returns full blocks after hashing.
    natural_join() – perform natural join
    clear() – clear all data in the virtual main memory.

I implemented the I/O process between virtual disk and virtual main memory in the function for the natural join. In the function, every time when the data block moves from virtual disk to virtual main memory or from virtual main memory to virtual disk, the number of disk I/O is increment. In the phase one of the two-pass join algorithm, function read() in VDisk is used to read a data block in the virtual disk. Then, function write_at() in VMM is used to write a data block in the virtual main memory. After hashing, function hash() is used to read full blocks from the virtual main memory. Function hash_wirte() in VDisk is used to write the blocks in the virtual disk. Function read_at() in VMM is used to read blocks which are not full after hashing from the virtual main memory and function hash_write() is used to write the blocks in the virtual disk. In phase two of the two-pass join algorithm, function bucket_read() in VDisk is used to read a bucket of blocks of small relation from the virtual disk. Then function write_at() in VMM is used to write each block in a bucket in the virtual main memory. Function hashed_block_read() is used to read a block in a bucket of large relation from the virtual disk. Function write_at() is used to write the block in the virtual main memory.

Part 3. Hash Function

I use Attribute B value to calculate the hash value. First, I calculated the number of 1 of B in a binary form. Then, I use modulo to calculate the hash value.

Part 4. Join Algorithm

The join algorithm is implemented in a function natural_join() in Project2.java. I implemented the two-pass join algorithm. In the first phase of the algorithm, the program first transfers each block of relation R from the virtual disk to the virtual main memory. The block is written at the first block of the virtual main memory. Then the virtual main memory performs hashing to distribute tuples in the first block to the rest blocks. After hashing, full blocks are read from the virtual main memory and the blocks are written in the virtual disk. When there is no block of relation R in the virtual disk, the remaining blocks in the virtual main memory are sent to the virtual disk. The same process is done to the relation S. In the second phase of the algorithm, the program first transfers one bucket of relation R from the virtual disk to the virtual main memory. The blocks are stored from the second block of the virtual main memory. Then the program transfers one block of the bucket with the same hash index in relation S from the virtual disk to the virtual main memory. The block is stored in the first block of the virtual main memory. Next, the virtual main memory performs natural join between the tuples in the first block and the tuples in the rest blocks. This process is done for each block in the bucket with the same hash index in relation S. When the process is done, the virtual main memory read the next bucket of R from the virtual disk. This process is repeated until all buckets in relation R and S perform the natural join.

Experiment: Join based on Hashing

To test my implementation, first I created 1000 tuples in relation R. the values of the attribute B were randomly picked from the values of the attribute B in the relation S. Then I ran my natural join algorithm to compute the natural join of the relation R and the relation S. I printed the number of disk I/O's used in the algorithm. Then I randomly picked 20 values of attribute B from the relation R. I used the values to print all tuples in the join whose values of the attribute B are the selected values. For the second part of this process, I created new relation R of 1200 tuples. The values of the attribute B were randomly picked between 20000 and 30000. Then I ran my natural join algorithm to compute the natural join of the relation R and the relation S. I printed the number of disk I/O's used in the algorithm and all tuples in the join.

To check my implementation, I printed the number of blocks in each bucket after hashing. The result of the hashing is shown in the Figure 2(a) and Figure 2(b). In the figures, the number of hashed blocks were not distributed well. The tuples were likely to have a hash index in the middle. For relation R, the number of I/O's was 255 after hashing. The ideal number of I/O's is 2B(R) where B(R) is the number of blocks in R. Since there were 1000 tuples in R and 8 tuples per block, the ideal number of I/O's is 1000 / 8 * 2 = 250. The number of I/O's in my implementation was slightly different from the ideal number of I/O's. It seemed reasonable since my hash function does not uniformly distribute the tuples. It makes some tuples remain in the virtual main memory after hashing because the blocks are not full. My implementation does additional I/O's to move the remaining un-full blocks from the virtual main memory to the virtual disk. It caused my hashing has more I/O's. The similar result occurred for the relation S. Since there were 5000 tuples in S, the ideal number of I/O's is 5000 / 8 * 2 = 1250. My implementation has 1513 – 255 = 1258 I/O's which was a little more than the ideal I/O's. Thus, my implementation seems work well for the first phase of two-pass join algorithm based on hashing.

Figure 2(a) – Number of blocks in each bucket in R.

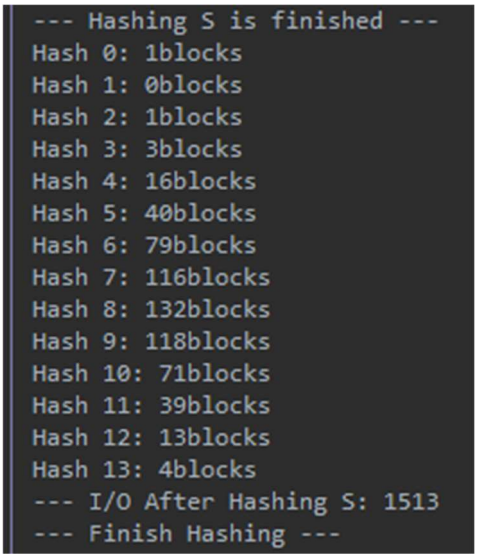

Figure 2(b) – Number of blocks in each bucket in S.

In the second phase of the algorithm, the whole bucket of each hash index in the smaller relation passes to the virtual main memory and each block of the bucket with the hash index in the larger relation passes to the virtual main memory to perform the natural join. To check the performance of my implementation, I printed the number of I/O's after natural join and number of tuples in the join. The result of the natural join is shown in the Figure 2(c). The ideal number of I/O's for two-pass join algorithm based on hashing is 3B(R) + 3B(S) where B(R) is the number of blocks in R and B(S) is the number of blocks in S. Since there were 5000 tuples in S and 1000 tuples in R, the ideal number of I/O's was (5000 / 8 * 3 + 1000 / 8 * 3) = 2250. My implementation performed 3308 I/O's which is 1058 more than the ideal I/O's. In my implementation, the tuples are not uniformly distributed in each bucket since my hash function does not perform well. So, there were buckets which have more blocks than the blocks for storing a bucket in the virtual main memory. In my implementation, the first block of the virtual main memory is used for holding a block from the larger relation and the rest blocks of the

virtual main memory are used for holding a bucket from the smaller relation. Since there are 15 blocks in the virtual main memory, 14 blocks are used to hold a bucket. In Figure 2(a), buckets with hash index of 6, 7, 8, 9, and 10 have more blocks than 14 blocks. The virtual main memory cannot hold whole blocks in those buckets. Therefore, in my implementation, the program separates bigger bucket to fit in the virtual main memory and send each separated bucket. To compare all tuples in the bucket in R and S, if there is a bucket which has more blocks, compared block of the larger relation is written back to the virtual disk and is written back to the virtual main memory again to compare with the tuples in the next separated bucket. For example, the bucket with hash index 6 in R has 17 blocks. To perform the natural join, first, 14 blocks of the bucket are sent to the virtual main memory. Then each block in the bucket with hash index 6 in S is sent to the virtual main memory. The virtual main memory performs the natural join and send the compared block of S back to the bucket in S in the virtual disk. After comparing all blocks, the rest blocks in the bucket is sent to the virtual main memory and the same process is performed. In this case, there is no more block in the bucket in R, the block from S is not written back to the virtual disk. In my implementation, the number of I/O's for each bucket in the second phase is B(R) + (2(⌊$B(R)$ / 14⌋) + 1) * B(S). With this formula, the number of I/O's with the values from Figure 2(a) and Figure 2(b),

I/O's = 0 + (0 + 1) * 1 + 0 + (0 + 1) * 0 + 0 + (0 + 1) * 1 + 1 + (0 + 1) * 3 + 3 + (0 + 1) * 16 + 8 + (0 + 1) * 40 + 17 + (2 + 1) * 79 + 26 + (2 + 1) * 116 + 26 + (2 + 1) * 132 + 22 + (2 + 1) * 118 + 16 + (2 + 1) * 71 + 7 + (0 + 1) * 39 + 2 + (0 + 1) * 13 + 2 + (0 + 1) * 4 = 1795

In Figure 2(b), the number of I/O's after hashing S was 1513. Since 1513 + 1795 = 3308 and the Figure 2(c) displayed that the number of I/O's after natural join was 3322, It seems that my implementation worked properly.

```
--- Begin Natural Join ---
--- Finish Natural Join ---
Number of disk I/O: 3308
Number of tuples in join: 1000
--- Selected B values: [12230, 13551, 14935, 15278, 15989, 17519, 20563, 22036, 24479, 26725, 29156, 29774, 31590, 32660, 35023, 39629, 40729, 42300, 45011, 49541]
------ Natural Join ------
A: r137     B: 12230     C: s261
A: r37     B: 13551     C: s415
A: r476     B: 13551     C: s415
A: r497     B: 14935     C: s583
A: r880     B: 15278     C: s616
A: r427     B: 15989     C: s705
A: r349     B: 17519     C: s872
A: r290     B: 20563     C: s1251
A: r868     B: 22036     C: s1439
A: r300     B: 24479     C: s1736
A: r162     B: 26725     C: s2019
A: r168     B: 29156     C: s2295
A: r331     B: 29774     C: s2361
A: r378     B: 31590     C: s2581
A: r639     B: 32660     C: s2731
A: r994     B: 35023     C: s3050
A: r385     B: 39629     C: s3662
A: r336     B: 40729     C: s3792
A: r551     B: 42300     C: s3985
A: r845     B: 45011     C: s4356
A: r125     B: 49541     C: s4947
```

Figure 2(c) – The result of the natural join.

Then I randomly picked 20 B values without duplication from the relation R and printed the tuples in the join whose B values are picked values. The result is shown in Figure 2(c). The program printed out 20 tuples. Since I picked B values without duplication and B values of R were picked from B values of S, it seemed reasonable that there were 20 tuples in the join.

In the second part of the experiment, I created a relation R of 1200 tuples. Relation S generated in the first part of the experiment was reused in this part. The result of the natural join is displayed in the figures below. The output of the experiment was copied in Result.txt file.

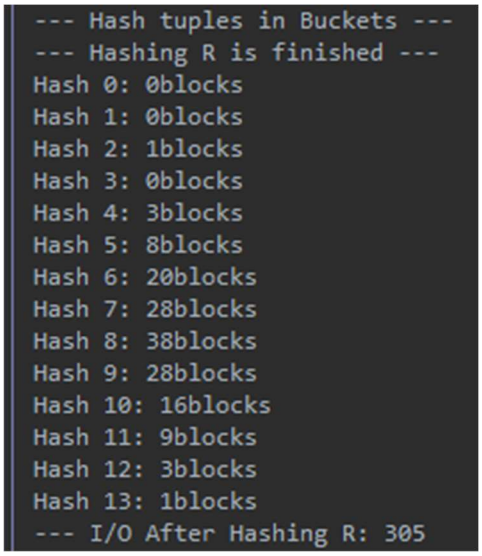

Figure 2(d) - Number of blocks in each bucket in R.

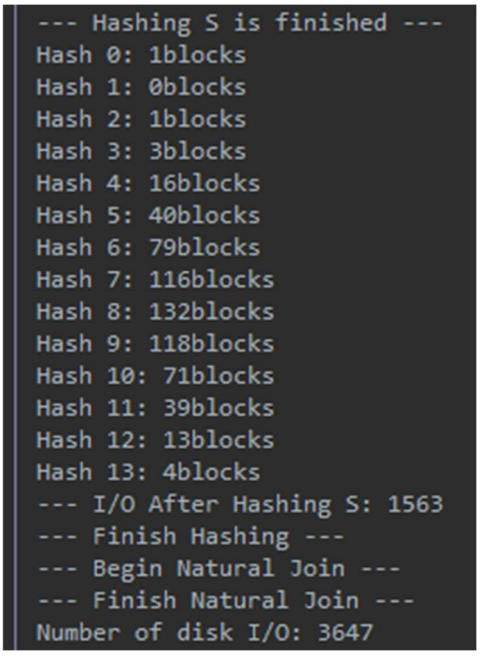

Figure 2(e) - Number of blocks in each bucket in S.

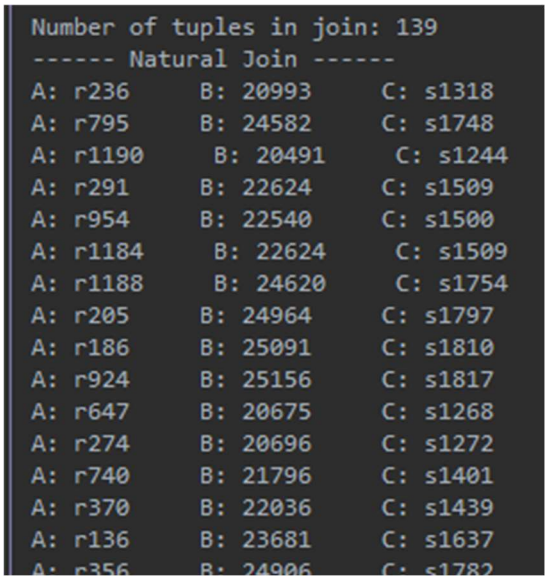

Figure 2(f) – The result of the natural join.

Discussion

At first when I implement the B+Tree, I did not understand the meaning of dense and sparse. After few studies, I understood the meaning of dense and sparse, but I encountered a new difficulty: how to create those trees. To create trees, I used small examples to find a pattern of tree format and finally could make a function to create dense tree and sparse tree. The next difficulty that I encountered was that my hash function does not perform well. In the class, we assume that there is enough space in main memory to hold all blocks in a bucket of small relation. But with the hash function that I used in the project does not uniformly distribute tuples in buckets and some bucket in small relation are bigger than the size of the virtual main memory. So, the pure two-pass algorithm did not work for natural join in this case since not all tuples are compared. To solve the problem, I put a handling code for the case when the size of bucket in small relation is greater than the size of virtual main memory. I divided large bucket into pieces which can be fit into the virtual main memory. Then it passes the tuples piece by piece into the virtual main memory. For the larger relation, if there is a piece of bucket of small relation in virtual disk, all blocks are sent to the virtual main memory for natural join and written back to the virtual disk. If there is no piece, it works as two-pass algorithm. As implementing B+Tree and two-pass join algorithm based on hashing, I learned how the processes in each topic work. In implementation of B+Tree, I learned how to manage data in the form of B+Tree. In addition, I learned two types of B+Tree. In implementation of two-pass join algorithm, I learned how data transfer works between disk and main memory. Further, I learned how two-pass algorithm works to optimize the computation between tuples. As I work on the project, I could revisit the concepts that I learned in the class. Overall, through the project, I earned experience in developing index structures and efficient algorithms for relational algebraic operations.