Theodore Kim
JinZhao Su
CS-UY 3083: Introduction to Databases
Homework #6

Problem 1

1a.
Value 58 is compared to root(40)
Compare value 58 to child B of root(48,62 [STOP])
Compare value 58 to child H of B(48,50,52,54,56,58[STOP])
Follows the pointer 58 from H to find the data

1b.
Value 32 is compared to root(40)
Compare value 32 to A(10,18,32)
Compare value 32 to child F of A(32,34,36,38)
Follows the pointer 32 from F to find the data

1c.
Value 24 is compared to root(40)
Compares value 24 to A(10,18,32)
Compares value 24 to child E of A(18,20,22,24[FOUND],26,28,30)
Goes through the values 26,28,30 to find the files
Compares the leftmost, which is the value 42 to:
Compares value 42 to child F of A(32,34,36,38)
Goes through the values 32,34,36,38 to find the files
Compares value 42 to child G of B(40,42[STOP])
Goes through the value 40 to find the files

1d.
Value 24 is compared to root(40)
Compares value 24 to A(10,18,32)
Compares value 24 to child E of A(18,20,22,24[FOUND],26,28,30)
*Since the B+ tree is a primary index, the data file is already sorted. Since it is sorted, you can now just continue with the item until the index reaches a=42. So it will continue to load the files 26,28,30,32,34,36,38,40 and stop once it reaches 42.

2a.
Value 19 would be compared and inserted to D.
Node D = (10,11,12,14,16) #This is okai since Size(D)<=7
Node A will remain the same

2b.
Value 19 would be compared and inserted to E. However, node E is already at size 7, which is at maximum capacity. What we can and will do is get all the data of node E (18,20,22,24,26,28,30) and turn it into two nodes: E and E’.

2c.
Node E = (18,19,20,22)
Node E’= (24,26,28,30)
Node A would have to be updated(parent of Node E and E’)
Node A = (10,18,24,32) #old one was (10,18,32)

2d.
Value 59 would be compared and inserted to H.
Node H != (48,50,52,54,56,58,59,60) because then Size(H)=8, which violates the law that Size(node)<=7.
Split Node H into Node H and Node H’
Node H = (48,50,52,54)
Node H’ = (56,58,59,60)
Node B would have to be updated(parent of Node H and H’)
Node B = (48,56,62,74,82,100,200,300)
#old Node B was (48,62,74,82,100,200,300)

Problem 2: Consider a B+ tree with a root, two levels of internal nodes, and leaf nodes (height of 4), each leaf has 10 children, each leaf has 10 values. Each node on its own block, each data block in r has 4 tuples.

1. How many different values of a does r have?

There are 10^4 (10,000) values of *a* that *r* has as there are 4 levels of 10 nodes in the B+ tree.

2. What is the numerical value of the number of blocks r occupies?

The relation, *r*, occupies 2,500 data blocks as there are a total of 10,000 nodes (tuples) represented by this B+ tree as the tuples belonging to *r*. As a single data block holds 4 tuples, the number of data blocks used by *r* is 10,000 / 4 = **2,500**.

3. Consider the execution of `SELECT * FROM r WHERE a = 'foo'`, what is the worst case execution time of the query using a sequential scan in terms of block access time (t_S) and block transfer time (t_T).

In the worst case, every single block is checked to perform the comparison of a = 'foo'. Therefore, all 2,500 blocks would need to be transfered to make that comparison individually, but only a single seek would need to be performed to reach the initial position on disk (as the rest is sequential), so the total execution time is **$2{,}500 * t_T + t_S$**.

4. How long will it take to execute the query using the index?

Using the B+ tree index, the correct node can be found by accessing O(h) index blocks (where *h* is the height of the tree, in this case 4) and the actual datablock that contains the data. The overall runtime is $\mathbf{(4 + 1)(t_T + t_S)}$

Problem 3:

1. Give the number of transfers and number of seeks for the pseudo code in terms of b_t, b_s and n_A.

To implement this pseudocode, first the SELECT query is processed so that each tuple of the takes relation is scanned for whether or not it adheres to the WHERE condition. Then a new query is executed to fetch the name corresponding to the *id* of the fetched students (i.e. the number of students who got an A in CS 3083). The first query requires only a single seek (as the blocks are contiguous), while the the second "round" of queries requires one seek per query. The first query required a transfer for every in the relation (to perform the comparison), and the second round of queries transferred each entry in the student relation to search for a single id. Therefore, the final expression for the # of transfers and seeks are as follows:

# of transfers = $n_A * b_S + b_t$
# of seeks = $1 + n_A$

(NOTE: This is all assuming linear selection)

2. Write a query that finds the names of students who got an ‘A’ in ‘CS 3083’ with formula for the number of block transfers and seeks

SELECT name **FROM** takes **NATURAL JOIN** student **WHERE** course_id = “CS 3083” **AND** grade = “A”

This query will join the student and takes relations together then scan the joined relation for the conditions. Using a Block Nested Loop Join, the block of both relations are paired and iterated over to check for the JOIN condition. In the worst case, the # of transfers and seeks are as follows:

# of transfers = $b_s * b_t + b_s$
# of seeks = $2 * b_s$

3. Comment on what your answers to (1) and (2) imply about the nested loop application style used above, in which a query is instantiated and executed multiple times within a loop.

Using a JOIN statement query is more efficient as, the first example, wastes block transfers iterating through the second relation searching for the entry that satisfies the JOIN parameter (in this case, the *id* attribute). Therefore the pseudocode implementation is wasting n_A times more transfers than the JOIN implementation.