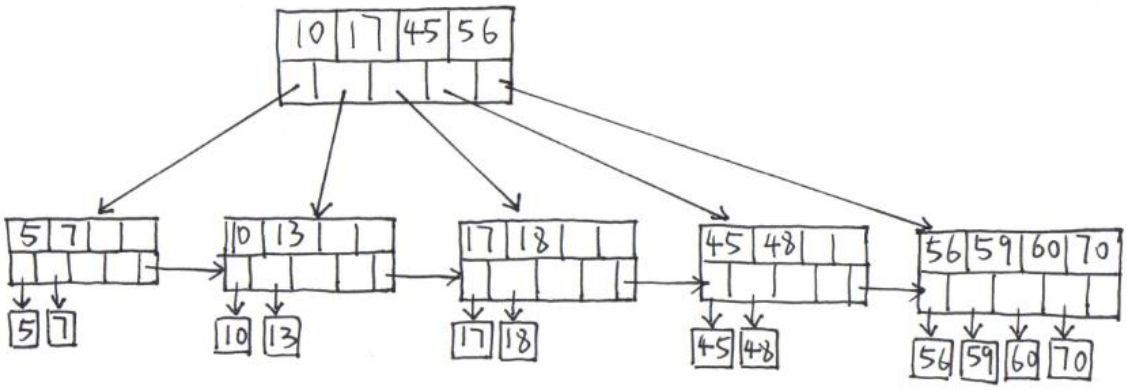
1.[40 points] Consider the following B+tree for the search key “age”. Suppose the degree d of the tree = 2, that is, each node (except for root) must have at least two keys and at most 4 keys.



a) Describe the process of finding keys for the query condition “age >= 10 and age <= 50”. How many blocks I/O’s are needed for the process?

Answer:

1. Read the root node, find the 2nd leaf node.
2. Read the 2nd leaf node, find the start point 10.
3. Then sequential traversal of leaves until 50, so we read 3rd 4th and 5th leaf node.
4. The end point 50 is less than the first data in 5th leaf node, so we stop.

Finally, to complete this searching, we cost 5 read 0 write => 5 block I/O’s.

b) Draw the updated B+tree after inserting 14, 16, and 15 into the tree. Show the tree after all insertions. How many blocks I/O’s are needed for the process?

Answer:

When insert 14:

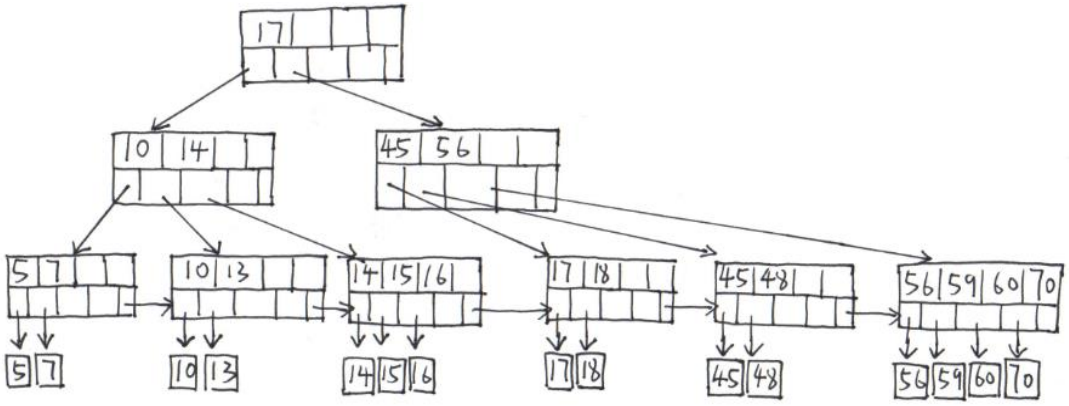
We cost 2 read(root node and leaf node start with 10) and 1 write => 3 block I/O’s.

When insert 16:

We cost 2 read(root node and leaf node start with 10) and 1 write => 3 block I/O’s.

When insert 15:

We cost 2 read(root node and leaf node start with 10) and 5 write(create new leaf node start with 10, create new leaf node start with 14, create new internal node start with 10, create new internal node start with 45 and create new root node start with 17) => 7 block I/O’s.



Finally, to complete this searching, we cost 6 read 7 write => 13 block I/O’s.

c) Draw the updated tree after deleting all odd ages (one by one) from the leaf nodes of the tree obtained in part b. Follow this order in deletion from leaf to right: 5, 7, … Show the tree after all deletions. How many blocks I/O's are needed for the process?

Answer:

When delete 5:

We cost 4 read(root node, internal node starts with 10, leaf node starts with 5, leaf node starts with 10) and 2 write(create new leaf node with 7, 10, and 13 and create new root node with 14, 17, 45, and 56) => 6 block I/O's.

When delete 7:

We cost 2 read(root node and leaf node start with 7) and 1 write(update leaf node with 7) => 3 block I/O's.

When delete 13:

We cost 3 read(root node, leaf node starts with 10 and leaf node starts with 14) and 3 write(update leaf node with 10, 14, update leaf node with 15, 16 and update root node with 15, 17, 45 and 56) => 6 block I/O's.

When delete 15:

We cost 4 read(root node, leaf node starts with 15, leaf node starts with 10 and leaf node starts with 17) and 2 write(create new leaf node with 10, 14 and 16 and update root node with 17, 45, 56) => 6 block I/O's.

When delete 17:

We cost 4 read(root node, leaf node starts with 17, leaf node starts with 10 and leaf node starts with 45) and 3 write(update leaf node with 10, 14, create new leaf node with 16, 18 and update root node with 16, 45, 56) => 7 block I/O's.

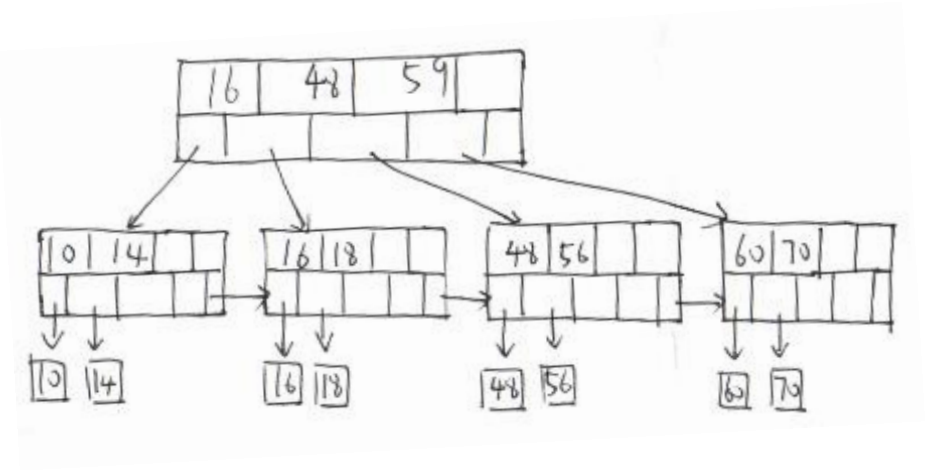
When delete 45:

We cost 4 read(root node, leaf node starts with 45, leaf node starts with 16 and leaf node starts with 56) and 3 write(create new leaf node with 48, 56, create new leaf node with 59, 60 and 70 and update root node with 16, 48, 59) => 7 block I/O's.

When delete 59:

We cost 2 read(root node and leaf node start with 59) and 1 write(update leaf node with 60, 70) => 3 block I/O's.

So finally, we get



In total, there are 23 read and 15 write => 38 block I/O's.

2.[60 points] Consider natural-joining tables R(a, b) and S(a, c). Suppose we have the following scenario.

i. R is a clustered relation with 20,000 blocks and 200,000 tuples.
ii. S is a clustered relation with 10,000 blocks and 200,000 tuples
iii. S has a clustered index on the join attribute a
iv. V(S, a) = 100 (recall that V(S, a) is the number of distinct values of a in S)
v. 102 pages available in main memory for the join.
vi. Assume the output of join is given to the next operator in the query execution plan (instead of writing to the disk) and hence the cost of writing the output is ignored

Describe the steps (including input, output at each step, and their sizes) in each of the following join algorithms. What is the total number of block I/O's needed for each algorithm? Which algorithm is most efficient?

a) Nested-loop join with R as the outer relation (B(R) = 20,000 and B(S) = 10,000 and M = 100)

R ⋈ S
for each 100 blocks b_r of R do
    for each block b_s of S do
        for each tuple r in b_r do
            for each tuple s in b_s do
                if r and s join then output (r, s)

Cost:

- Read R once: cost B(R)
- Outer loop runs B(R)/100 times, and each time need to read R: cost B(R)B(S)/100
- Total cost: B(R) + B(R)B(S)/100 = 2,020,000 Block I/O's.

b) Nested-loop join with S as the outer relation (B(R) = 20,000 and B(S) = 10,000 and M = 100)

S ⋈ R
for each 100 blocks b_s of S do
    for each block b_r of R do
        for each tuple s in b_s do
            for each tuple r in b_r do
                if r and s join then output (r, s)

Cost:

- Read S once: cost B(S)
- Outer loop runs B(S)/100 times, and each time need to read R: cost B(S)B(R)/100
- Total cost: B(S) + B(S)B(R)/100 = 2,010,000 Block I/O's.

c) Sort-merge join (assume only 100 pages used for sorting and 101 pages for merging). When the number of runs of a relation is too large for merging, the runs will be further merged first. Select the relation with larger number of runs for further merging if both have too many runs.

a. Load 100 blocks of R each time, sort them and send them back to disk. This will create 200 runs, each of size is 100. Cost = 2 B(R) = 40,000
b. Load 100 blocks of S each time, sort them and send them back to disk. This will create 100 runs, each of size is 100. Cost = 2 B(S) = 20,000

Since the number of runs of R > 101 – 1 and S > 101 – 1, we cannot merge them directly. So, we merge them in ⌈200/100⌉ = 2 runs and ⌈100/100⌉ = 1 runs respectively.

c. Load 200 runs of R , and merge them and send them back to disk, it generates 2 runs and each run size is 10,000 pages. Cost = 2 B(R) = 40,000

d. Load 100 runs of S , and merge them and send them back to disk, it generates 1 runs and each run size is 10,000 pages. Cost = 2 B(S) = 20,000

e. Merge 2 runs from R and 1 runs from S, Join all sorted runs. Cost = B(R) + B(S) = 30,000

So total cost: 5B(R) + 5B(S) = 150, 000 Block I/O's

d) Simple sort-based join (same assumption as above)

a. Load 100 blocks of R each time, sort them and send them back to disk. This will create 200 runs each of size 100. Cost = 2 B(R) = 40,000

Since the number of runs of R > M, we cannot load them at a time. So, we merge them in ⌈200/100⌉ = 2 parts.

b. Load 200 runs of R, and merge them, it generates 2 run and each run size is 10,000 pages. Cost = 2 B(R) = 40,000

c. Merge 2 runs from R into 1 run, each run size is 20, 000 pages, and write it back to disk. Cost = 2 B(R) = 40,000

d. Load 100 blocks of S each time, sort them and send them back to disk. This will create 100 runs each of size 100. Cost = 2 B(S) = 20,000

e. Load 100 runs of S, and merge them, it generates 1 run and the run size is 10,000 pages. Cost = 2 B(S) = 20,000

f. Join all sorted runs: Cost = B(R) + B(S) = 30,000

So total cost: 7B(R) + 5B(S) = 190, 000 Block I/O's.

e) Partitioned-hash join (assume 101 pages used in partitioning of relations and no hash table used for lookup in joining buckets)

a. Hash R into 100 buckets and send them back to disk: Cost = 2 B(R) = 40,000 200 blocks/bucket

b. Hash S into 100 buckets and send them back to disk: Cost = 2 B(S) = 20,000 100 blocks/bucket

c. Join corresponding buckets from R and S Cost: B(S) + B(R) = 30,000

So total cost: 3B(R) + 3B(S) = 90, 000 Block I/O's.

f) Index join (ignore the cost of index lookup) (B(R) = 20,000, B(S) = 10,000, T(R) = 200,000, T(S) = 200,000, V(S, a) = 100 and M = 100)

a. Load R into 100 pages, iterate R by tuples. Cost = B(R) = 20,000

b. For each tuple in R, fetch corresponding tuples from S. Cost = T(R)B(S)/V(S, a) = 20,000,000

c. Join R and S

Since R is clustered, so total cost: B(R) + T(R)B(S)/V(S, a) = 20,020,000 Block I/O's

Based on what number of these total blocks I/O would cost in each algorithm, the partitioned-hash algorithm is the most efficient.