CS 4320/5320 - Homework 2

Submitted by:
Saicharan Mujumdar: ssm268
Akhil Umesh Mehendale: aum5
Prashanth Basappa: pb476

1. Written Part

1.1 Indexing Basics

(a)

1. Yes
2. Yes
3. No
4. Yes

(b)

1. No
2. Yes
3. No
4. No

- Advantage of tree based indexes over hash based indexes:

- Tree based indexes can be used to match equality and comparison (range) conditions. Whereas a hash based indexes can be used to only find equality indexes.
- Tree based indexes can be used even if a prefix of the composite index is matched. Whereas a hash index can be used only when all the selection condition uses all the attributes that are used in the composite index keys.

1.2 Tree-based Indexing

(a)
The number of pointers at each index node will be fanout (f). Since in a b+ tree of order d each node at 2d keys and 2d + 1 pointers.

8 * (2d + 1) + 20 (2d) = 560
d = 9.85
f has to rounded off to the lowest natural number. Hence the order d will be:
d = floor(9.85) = 9

Fanout = Number of pointers at each node = 2d + 1 = 19

Height of tree = $\log_f$ (N) where N = Number of data entries in all leaf pages.

To index one record we need a key and pointer i.e. 20 + 8 = 28 bytes. So number of key-pointer pairs in one leaf page is:

560/28 = 20
N = Number of records / number key-value pairs per leaf page
N = 10000 / 20 = 500

Level of tree = ciel ($\log_{19}$ (10000/ 20) + 1) = 4

(b)
Since nodes at each level were filled as much as possible:
at level 4 = ciel (number of leaf nodes) = 10000/20 = 500 nodes
level 3 = ciel (500/19) = 27 nodes
level 2 = ciel (27 /19) = 2 nodes
level 1 = ciel (2/19) = 1 nodes

(c)
Order of tree d:
2d (10) + (2d + 1) (8) <= 560
d = 15

Fanout f = 2d + 1 = 31

Height of tree = $\log_f$ (N) where N = Number of data entries in all leaf pages.

To index one record we need a key and pointer i.e. 10 + 8 = 18 bytes. So number of key-pointer pairs in one leaf page is:

560/18 = 31
N = Number of records / number key-value pairs per leaf page
N = 10000 / 31 = ciel (322.58) = 323

Level of tree = ciel ($\log_{31}$ (10000/ 31) + 1) = 3

(d)

If page is 70 % full that means the page size is 560 * 0.7 = 392

The number of pointers at each index node will be fanout (f). Since in a b+ tree of order d each node at 2d keys and 2d + 1 pointers.

8 * (2d + 1) + 20 (2d) <= 392
d = 6.85
f has to rounded off to the lowest natural number. Hence the order d will be:
d = floor (6.85) = 6

Fanout = Number of pointers at each node = 2d + 1 = 13

Height of tree = $\log_f$ (N) where N = Number of data entries in all leaf pages.

To index one record we need a key and pointer i.e. 20 + 8 = 28 bytes. So number of key-pointer pairs in one leaf page is:

392/28 = 14
N = Number of records / number key-value pairs per leaf page
N = 10000 / 14 = ciel (714.28) = 715

Level of tree = ciel ($\log_{13}$ (10000/ 14) + 1) = 4

1.3 Hash based Indexing

a. Nothing can be said about the last entry into the index: it can be any of the data because there is no information about any overflow pages. So we do not know which element was added last in the entries of the index.

b. If the last item that was inserted had a hash code h0 (keyvalue) = 00 then it caused a split, otherwise, any value could have been inserted at any of the levels. This is ambiguous because there is no info about any overflow pages.

c. The last data entry which caused a split satisfies the condition
h0 (keyvalue) = 00. As there are no overflow pages for any of the other buckets.

(d) Level = 0

H1	h0	Primary Pages				Overflow Pages
000	00	32	8	24		
001	**01**	9	25	41	17	
010	10	14	18	10	30	
011	11	31	35	7	11	
100	00	44	36	4		

Split counter is the same. Next=1. A new bucket is not created. Level =0 is same.

(e) Level = 0

H1	h0	Primary Pages				Overflow Pages			
000	00	32	8	24					
001	01	9	25	41	17				
010	**10**	14	18	10	30				
011	11	31	35	7	11	15			
100	00	44	36	4					
101	01								

Split counter is incremented by 1. Next=2. A new bucket is created. Level =0 is same.

(f)

Level = 0

H1	h0	Primary Pages				Overflow Pages
000	**00**	32	8	24		
001	01	9	25	41	17	
010	10	14	18	10	30	
011	11	31	35	7	11	

Split counter is decremented by 1. Next=0. No new bucket was created. The last split got removed.

(g)
The following constitutes the minimum list of entries to cause two overflow pages in the index:

- 19, 43, 67, 99, 123 (5 entries).

The first insertion causes a split and causes an update of Next= 2. The insertion of 123 causes a subsequent split and Next is updated to 3 which points to this bucket. This overflow chain will not be redistributed until three more insertions (a total of 8 entries) are made. So, the maximum number of entries that can be inserted into this bucket before a split occurs that reduces the length of this overflow chain is 8. In principle if we choose data entries with key values where its binary form ends in a 011, we can take the maximum number of entries that can be inserted to reduce the length of the overflow chain to be greater than any arbitrary number. This is because the initial index has 31(binary 11111), 35(binary 10011), 7(binary 111) and 11(binary 1011). So by an appropriate choice of data entries as mentioned above we can make a split of this bucket because just two values (7 and 31) to be redistributed to the new bucket.

1.4 Indexing Costs

(a)

The cost of the query will using a clustered B+ tree indexed on title will be **8000** I/Os.
An additional index along with title won't be of any use as we still have to use the index title first to traverse the tree. However, a new B+ tree indexed on dname would significantly reduce the number of I/Os.

(b)

Since the tree index is on title we need to read the entire file. While reading we filter only a subset off the record that satisfy the where condition. Then we sort and project this data subset and compute group by.

Cost of reading file = 8000 I/Os
Cost of sorting (filtered data) = 2 * (8000 * 1/4) * (10/100) $\log_4$ (ciel ((2000) /5) + 1) = 1600

Total Cost = 9600 I/Os

(c)

Since there is a clustered b+ index on dname, we can traverse the index and filter the dnames which satisfy the where condition. Then we write those filtered records and sort them to get group by.

Cost of index traversal = 8000 * ¼ * 10/100 = 200
Cost reading filtered records = 8000 * 10/100 = 800
Cost of writing filtered records into temporary table = 800 * ¼ = 200
Cost of sorting = 2 * (200) $\log_4$ (ciel (200/5) + 1) = 1600

Total = 2800 I/Os

Having an additional index on title will help as in that case it will just be an index only scan.

(d)

Since there the index is on <dname, title> we can perform an index only scan to get the relevant record and the sort those records to get the solution. To sort we need to store filtered records into temporary table.

Cost of index traversal = 8000 * 2/4 * 10/100 = 400
Cost of writing filtered records into temporary table = 8000 * 10/100 * ¼ = 200
Cost of sorting = 2 * (200) $\log_4$ (ciel (200/5) + 1) = 1600

Total cost = 2200 I/Os

(e)
Since there the index is on <title, dname> we can perform an index only scan to get the relevant result. The result is already sorted as the first key of the index is title.

A clustered B+ index on <title, dname> supports an index-only scan costing
Total Cost = 8000* 2/4 = **4000 I/Os.**