Assume an ordered file whose ordering field is a key. The file has 1000000 records of size 1000 bytes each. The disk block is of size 4096 bytes (unspanned allocation). The index record is of size 32 bytes. How many disk block accesses are needed to retrieve a random record when searching for the key field

1. Using no index ?
2. Using a primary index ?
3. Using a multilevel index ?

Solution: (1) Without the index, we can run a binary search to find a given value of the ordered field. In the worst case, this takes $\lceil \log_2 b \rceil$ where b is the number of blocks to store the data file. To compute b, compute first the blocking factor (i.e. number of data records per block):

$bf = \lfloor (4096/1000) \rfloor = 4$

Note that we used the floor function because of unspanned allocation (i.e. records must be store completely in the block). Then,

$b = \lceil (1000000/4) \rceil = 250000$

data blocks. Then, the number of data blocks to access is at most

$\lceil \log_2 250000 \rceil$ = 18.

(2) In a primary index, we have as many index records as data blocks we have, i.e. 250000. Let us first compute how many blocks we need to store the index. The blocking factor for the index is

$bf_i = \lfloor (4096/32) \rfloor = 128$

Then, we need

$b_i = \lceil (250000/128) \rceil = 1954$

blocks to store the index. Since the index file is ordered by definition, we can run a binary search to find the data block of the required value. Then, we need at most

$\lceil \log_2 1954 \rceil + 1 = 12$

block accesses to get to the data. Note that the "+1" comes from the access to read the data whereas the other eleven come from the search for the pointer to the data block in the index file.

(3) To construct a multilevel index, we construct an index of the index file constructed in (2). Since this index file is ordered according to a key value, the index we are going to construct is a primary index and, thus, it has as many entries as blocks in the index in (2), i.e. 1954. Since the blocking factor for the index is 128, we need $\lceil (1954/128) \rceil = 16$ blocks to store the new index. This new index is called a level 2 index, whereas the one built in (2) is a level 1 index. Now, let us build a level 3 index, i.e. an index of the level 2 index file. Again, the index we are going to construct is a primary index and, thus, it has as many entries as blocks in the level 2 index, i.e. 16. Since the blocking factor for the index is 128, we need only one block to store the new index. And we are done.

To access a data entry, we need to read the block of the level 3 index, find the pointer to the appropriate block of the level 2 index and read this block, find the pointer to the appropriate block of the level 1 index and read this block, find the pointer to the appropriate data block and read this block. So, we need 4 block accesses.

B-tree and B+-tree.

$B = 4096$ bytes, $P = 16$ bytes, $K = 64$ bytes, node fill percentage=70 %.

For both B-trees and B+-trees:

1. Compute the order p.
2. Compute the number of nodes, pointers and key values in the root, level 1, level 2 and leaves.
3. If the results are different for B-trees and B+-trees, explain why this is so.

Solution:

B-tree:

(1) From the structure of a node in the B-tree, we have

$p * P + (p - 1) * (P + K) = B * 70\%$

where $P = 16$ is the size of a pointer (either the block pointer or the record pointer), and $K = 64$ is the size of the search key field. $B = 4096$ is the block size. Accordingly, we get

$p * 96 = 2947$ thus $p = 30$.

(2) At the root level, we have one node, 30 pointers and 29 key entries. Note that the number of key entries is one less than the number of pointers. At level one, we have 30 nodes, because for each pointer from the root node there is one node at the level one. Then there are $30 * 29 = 870$ key entries, and $30 * 30 = 900$ pointers respectively. At level two, there are $30 * 30 = 900$ nodes, $900 * 29 = 26100$ key entries, and $900 * 30 = 27000$ pointers. At the leaf level, we have $900 * 30 = 27000$ nodes, $27000 * 29 = 783000$ key entries, and $27000 * 30 = 810000$ pointers. In summary, the values are as follows:

Level	nr. nodes	nr. key entries	nr. pointers
Root	1	29	30
Level 1	30	870	900
Level 2	900	26100	27000
Leaf	27000	783000	810000

B+-tree:

(1) With B+-trees we distinguish the internal nodes from the leaf nodes. For an internal node, we have

$p * P + (p - 1) * K = B * 70\%$

where $P = 16$ is the size of a block pointer, and $K = 64$ is the size of the search key field. $B = 4096$ is the block size. Accordingly, we get

$p * 80 = 2931$ thus $p = 36$.

On the other hand, p_{leaf} is calculated as follows:

$p_{leaf} * (P + K) + P = B * 70\%$

where $P = 16$ is the size of a pointer (either the block pointer or the record pointer), and $K = 64$ is the size of the search key field. $B = 4096$ is the block size. Accordingly, we get

$p_{leaf} * 80 = 2851$ thus $p_{leaf} = 35$.

(2) The number of nodes, key entries and the pointers for a B+-tree can be obtained analogously to those of B-tree:

Level	nr. nodes	nr. key entries	nr. pointers
Root	1	35	36
Level 1	36	1260	1296
Level 2	1296	45360	46656
Leaf	46656	1632960	

Note that the number of key entries of the leaf nodes is calculated with $46656 * 35 = 1632960$, where 46656 is the number of nodes and 35 is p_{leaf}. Moreover, the value of p_{leaf} as 35 is incidentally one less than p, which is 36 in this example. However, there is no such a correlation in general.

(3) One observation is that the p value is larger for B+-tree than B-tree, given the same value of the block size. The reason is that there are no pointers at the key entries of the internal nodes of B+-trees. As a result, nodes in B+-trees have larger fan-out. Accordingly, there are more key entries at the leaf nodes of B+-trees.