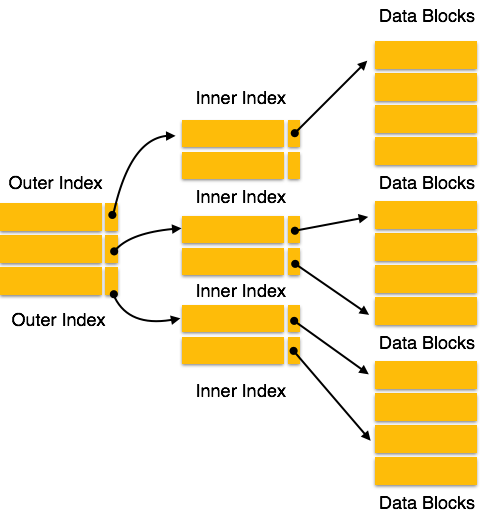
**Multilevel Index**

Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



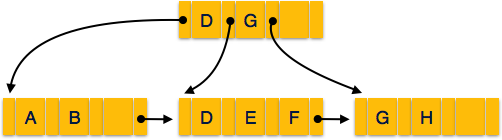
Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

B+ Tree

A B+ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B+ tree denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, the leaf nodes are linked using a link list; therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

Every leaf node is at equal distance from the root node. A B+ tree is of the order **n** where **n** is fixed for every B+ tree.



**Internal nodes** −

* Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
* At most, an internal node can contain **n** pointers.

**Leaf nodes** −

* Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values.
* At most, a leaf node can contain **n** record pointers and **n** key values.
* Every leaf node contains one block pointer **P** to point to next leaf node and forms a linked list.

B+ Tree Insertion

* B+ trees are filled from bottom and each entry is done at the leaf node.
* If a leaf node overflows −
  + Split node into two parts.
  + Partition at **i = ⌊(m+1)/2⌋.**
  + First **i** entries are stored in one node.
  + Rest of the entries (i+1 onwards) are moved to a new node.
  + **i*th*** key is duplicated at the parent of the leaf.
* If a non-leaf node overflows −
  + Split node into two parts.
  + Partition the node at **i = ⌈(m+1)/2⌉**.
  + Entries up to **i** are kept in one node.
  + Rest of the entries are moved to a new node.

B+ Tree Deletion

* B+ tree entries are deleted at the leaf nodes.
* The target entry is searched and deleted.
  + If it is an internal node, delete and replace with the entry from the left position.
* After deletion, underflow is tested,
  + If underflow occurs, distribute the entries from the nodes left to it.
* If distribution is not possible from left, then
  + Distribute from the nodes right to it.
* If distribution is not possible from left or from right, then
  + Merge the node with left and right to it.

## Multi Level Indexes

The indexing described so far involve ordered index files. A binary search is applied to the index to locate pointers to disk blocks or records in the file having a specific index field value.

Remember from before, a binary search requires log2bi block accesses for an index with bi blocks. Each step of the binary search reduces the part of the index file we search by a factor of 2. (Each step divides the search space by 2, or halves the search space)

With multilevel indexes, the idea is to reduce the part of the index that we continue to search by a larger factor, the blocking factor of the index, where the bfri is greater than 2. The blocking factor of the index, bfri is called the **fan-out**, or fo of the multilevel index.

Searching a multilevel index requires approximately (logfobi) block accesses which is a smaller number than for a binary search if the fan-out is larger than 2. If the fan out is equal to 2, there is no difference in the number of block accesses.

The index file is called the first level of a multilevel index. It is an ordered file with a distinct value for each key value K(i). Therefore we can create a ***primary index*** for the first level. This index to the first level is called the second level of the multilevel index.

The second level is a primary index, therefore we can use block anchors, and the second level has ***one entry*** ***for each block*** in the first level index. The blocking factor for all other levels is the same as for the first level, because the size of each index entry is the same. Each entry has one field value, and one block address.

If the first level has r1 entries, and the blocking factor (which is also the fan out) for the index is bfri = fo, then the first level needs ⎡r1/fo⎤ blocks, which is therefore the number of entries r2 needed at the second level of the index.

If necessary, this process can be repeated at the second level. The third level, which is an index for the second level, has an entry for each second level block, so the number of third level entries is r3 = ⎡r2/fo⎤.

We only require a second level only if the first level needs more than one block of disk storage, and we require a third level only if the second level requires more than one block as well.

This process can be repeated until all the index entries at some level t fit in a single block.

The block at the tth level is the top-level index. Each level reduces the number of entries from the previous level by a factor of fo (the index fan out or blocking factor of the index).

A multilevel index with r1 first level entries will have approximately t levels, where:

t = ⎡logfo(r1)⎤.

The multilevel indexes can be used on any type of index, primary, clustering, or secondary, as long as the first level index has distinct values for K(i), and fixed length entries.

**Example of a two level Primary Index:**



**Example Question from Text:**

Assume the dense secondary index from example 2 is converted into a multilevel index. The index blocking factor is bfri = 68 index entries per block. This value is also the fan out for the multilevel index. The number of first level blocks, b1 was calculated as 442.

How many second level blocks?

b2 = ⎡b1/fo⎤ = ⎡442/68⎤ = 7 blocks.

How many third level blocks?

b3 = ⎡b2/fo⎤ = ⎡7/68⎤ = 1 block.

Because all of the index entries at the third level can fit into one block, the third level is the top level of the index, and t = 3. (Remember: t is the number of levels required)

To access a record by searching the multilevel index, we must access one block at each level, plus one block from the data file, so we need t + 1 = 3 + 1 = 4 block accesses. In example 2, using a single level index with a binary search, we required 10 block accesses.

In this example, a dense secondary index was used, meaning there was a index entry for each record in the table. You could also have a multilevel primary index which is non dense (for example if the first level is a primary index).

In this case, we must access the data block from the file before we can determine whether the record being searched for is in the file. With a dense index, this can be determined by accessing the first index level, without having to access the data file, since there is an entry in the index for each record in the file.

Using multi-level indexes, we can reduce the number of block accesses required to search for a record given its indexing field value. Insertions and deletions are still a problem because all the index levels are physically ordered files.