## Lecture  15 — Multilevel Index: B+ Trees

# Multilevel Index: B$^+$ Trees

If instead of having just a flat index file, we decide to have multiple levels, then it means instead of performing a simple binary search on the index, we must traverse a tree. At first that may appear counterproductive, or at least, not helpful, since we would still have to traverse a bunch of blocks to get to the item we would like to find. That is true if we pick a binary tree, but, as the title of this section gives away, that's not what we will do.

Thinking back on how linear search works, we check an item to see if it matches and we reduce the remaining space to check by 1. Not efficient. If we do a binary search, we reduce the remaining space to check by half of its current size. Much better. Hypothetically we could improve on that if we could reduce the remaining space to search by more than that.

This is called the *fan out* of the multilevel index, and instead of dividing the remaining space to search by 2 like in binary search, we divide it $n$-ways where $n$ is the fan out. Then we reduce the record space by a lot. Searching the multi-level index requires about $log_{fo}($ number of blocks $)$ which is an improvement if the value for $fo$ is larger than 2, which it almost certainly is. In fact, we would like to see the fan out is in the order of 50 - 200 [EN11].

If our fan out is five, for example, when we need to find some data then it means we can at each step eliminate 80% of the remaining search space, which will obviously bring us to the answer much faster than if we eliminated only 50% of the search space. This pattern can be repeated at multiple levels of the index, which allows us to narrow in on the target quickly.

A common organization for files is called ISAM – Indexed Sequential Access Method. The IBM version of this has a two level index which maps as closely as possible to the hard drive underlying structure [EN11]. MySQL has its own implementation called MyISAM but it is terrible... Please do not use it; use InnoDB instead.

Let us examine a very simple approach for multi-level indexing. The lowest (or first, or base) level consists of blocks that contain pointers to the second level. The second level has blocks that contain pointers to the actual file.

Whether the file has multilevel index or not, in an index sequential file the performance does get slower as more entries are added. Reorganization can help, but it is expensive. Instead we might wish to use a balanced tree structure, specifically, a B$^+$ tree. As you have learned about the subject in a data structures & algorithms course in the past (and likely in an operating systems course too...), we will have just a quick refresher on how it all works:

A B$^+$ Tree structure, formally, has the following characteristics [Sta14]:

1. The tree is made up of nodes (have children) and leaves (have no children).

2. Each node contains at least one key identifying a file record, and more than one pointer to child nodes or child leaves.

3. Each node has some maximum number of keys.

4. The keys in a node are stored in non-decreasing order.

And a B$^+$ Tree of degree $d$ has the following properties [Sta14]:

1. Every node has at most $2d - 1$ keys and $2d$ children ($2d$ pointers).

2. Every node other than the root has at least $d - 1$ keys and $d$ pointers. Each internal node except the root is at least half full and has at least $d$ children.

3. All leaves appear on the same level.

4. A non-leaf node with $k$ pointers contains $k - 1$ keys.

To find something in a B$^+$ Tree, the general algorithm is fairly simple [Sta14]:

1. Start at the root node.

2. If the key is in the current node, it is found, and the algorithm terminates.

3. If the key is less than the smallest key in this node, follow the leftmost pointer; go to step 2.

4. If the key is greater than the largest key in this node, follow the rightmost pointer; go to step 2.

5. If the key is between the values of two adjacent keys, follow the pointer between them; go to step 2.

Insertion into the B$^+$ Tree is more complicated, because of the rules that keep the tree balanced [Sta14]:

1. Search the tree for the key. If it is not found, then we are at least looking at the block where it would be if it were there.

2. If this node has fewer than $2d - 1$ keys (that is, it is not full), insert the key into this node in the proper sequence. The algorithm terminates.

3. If the node is full, split this node around the median key into two new nodes with $d - 1$ keys each. Promote the median key to the higher level. If the new node is less than the median key, insert it in the left-hand new node; otherwise into the right.

4. The promoted node is inserted into the parent node in order, splitting the parent if the parent is already full.

5. If the process of promotion reaches the root node and the root is full, then splitting and promotion occurs and the height of the tree increases by 1.

Deletion is a little bit similar to insertion (but a little bit complicated) [SKS11].

1. Search the tree for the key. If it is not found, then there is nothing to do and the algorithm terminates.

2. Remove the target (and remove it from the parent nodes if it appears at intermediate levels as well), shifting the remaining entries if needed to ensure there are no gaps.

3. If the current node is the root and there is only one remaining child, then the child is the new root and delete the current node.

4. Otherwise, if the current node is less than half full then:
   (a) If an immediately adjacent node is exactly half full, then coalesce the current node with that adjacent node and update the pointers of the parent.
   (b) Otherwise, redistribution is necessary: move an item from the adjacent node so that the current node is at least half full and update the pointers of the parent.
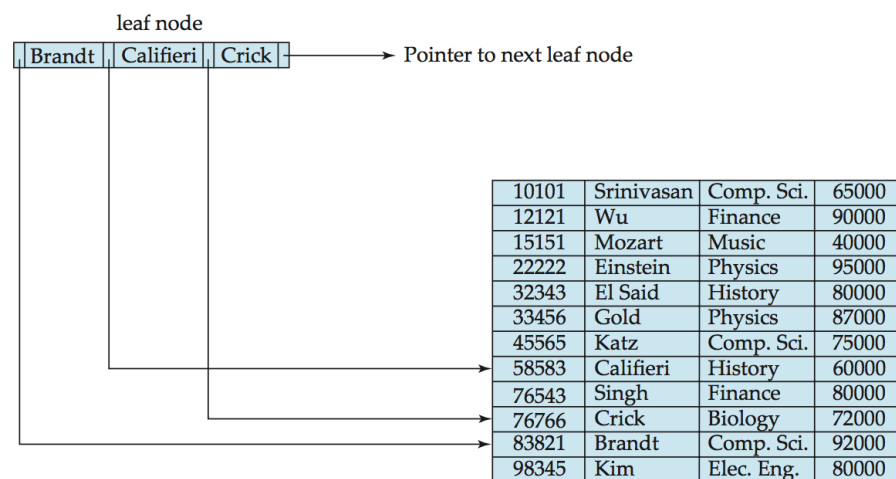
This glosses over, to some extent, how the actual deletion and insertions are performed. In particular, deletion requires some thought as to how to actually write the algorithm. The textbook contains a more extensive algorithm in pseudocode for both insertion and deletion.

There are some differences versus a general B-Tree which are worth calling out. We require that leaf nodes be at least half full (unless it is the root node and there isn't much in the file). The last pointer in a leaf node chains to the next leaf node to allow relatively quick access if we need to move through the data sequentially. So the last pointer is reserved and can't point directly to data. Non-leaf nodes are sometimes called internal nodes. Unlike your typical B-Tree, the items in the non-leaf nodes are duplicates (repeats) of the ones below. Why? The pointers in the non-leaf nodes point to leaf nodes, not the actual data.
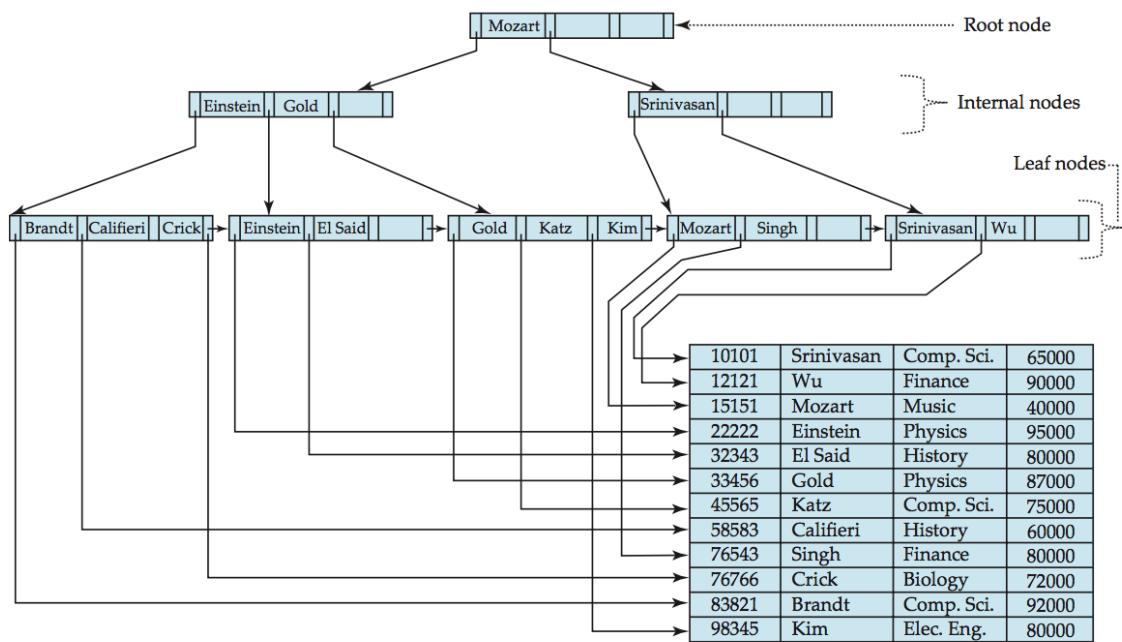
Why do we like to use trees? Well, here's an example from [SKS11]. Assume a node is 4 KB, we are searching a field of length 32 bytes, and pointers to disk blocks are 8 bytes. This means we can fit about 100 entries in a block. If the file to be queried contains 1 million records, then a lookup requires only $\lceil log_{100}(1\,000\,000) \rceil = 3$ accesses. This is the worst case, though, since the root node may be in the buffer frequently due to its frequent access. If we had to do this with a regular binary tree, the $\lceil log_2(1\,000\,000) \rceil = 20$ accesses and that is also the worst case for binary search. It is true that insertion and deletion are computationally expensive, but since the disk is the slow part of the system it is preferable to spend more CPU cycles computing the rebalancing of the tree than it is to spend more time accessing the disk.

Overall it seems like the benefit of much faster access to data outweighs the negatives of the B$^+$ Tree structure: duplicate data, more computational complexity to insert/delete, et cetera.

And now let us see an example of applying that to the "instructor" relation. First, a leaf node [SKS11]:



And then the complete tree for the instructor relation [SKS11]:

| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 80000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri | History | 60000 |
| 76543 | Singh | Finance | 80000 |
| 76766 | Crick | Biology | 72000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |

Even though name is non-unique (there's a lot of people with the name Wang/Wong, according to Wikipedia, in 2007 there were almost 93 million in China alone...!) this could easily be a secondary index and that would probably make sense.

Just for our interest, if this were a regular B-Tree and not a $B^+$ Tree then the storage would look something like this [SKS11]:
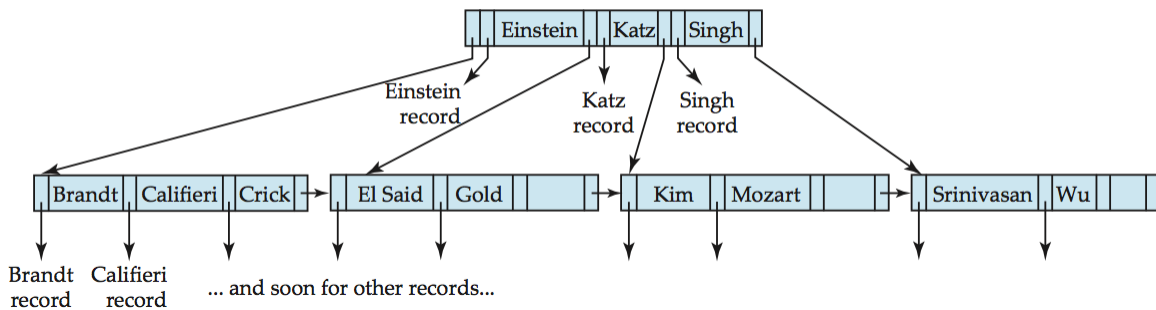


## Small Considerations in Indexing

**Indexing Strings**   Strings can cause a bit of a small problem when it comes to $B^+$ Trees because they can be of variable length or rather long, causing a pretty "tall" tree and losing the benefits of the structure. The suggested solution in [SKS11] is called prefix compression: store a prefix of the search key value that is sufficient to distinguish the sub-trees. Thus we could store in a non-leaf node "Jans" instead of "Jansen" if the closest two subtrees are under the subtrees are on the left "Janet" and on the right "Janson" (note different spelling) [SKS11].

**$B^+$ Tree File Organization**   Just as we have made use of a $B^+$ Tree organization for the index file, the actual file containing records could be stored using $B^+$ Trees as well.

**Covering Index.** A *covering index* is one that stores the value of some attributes (other than things we search on) alongside the pointers to the record [SKS11]. This increases the size of the index, but allows us to fulfill some queries without having to even go to the data blocks of the file at all. The advantage, of course, is that this can be much faster – if the data we need is at hand in the index, we just take it and use it and we need not load the record from disk. The downside is that it increases the size of the index, means fewer items can fit in one index block, and requires additional overhead to keep it up to date.

To give an example, it might be sensible to have in the covering index for employees, their last name alongside the regular index search key (ID). This would mean that queries where only the name is required, or name and ID, don't need to go into the file blocks at all.

There are limits to the size an index entry can be, so it's not possible (or sensible) to have all fields of a record in the index entry. Even if you could, it would pretty much defeat the point of having index blocks at all...

## Multiple Key Indexes

All discussion so far has covered the simple scenario where one index on one attribute is being used for a query. However, that is not the only way. Sometimes an index will be constructed on two attributes, or, we can combine two (or more) indexes on a single attribute.

**Indexes on Multiple Attributes.** Let us imagine that we have a set of addresses. It is likely that we may want a composite key on the attributes of (city, province) since city names can be duplicated: there is a Waterloo in Ontario and there is a Waterloo in Quebec (to say nothing of Belgium...). If such an index is defined, then the search key in the tree structures will be the combination of both attributes.

This is helpful in situations where we know that we are going to search on particular combinations of a certain attribute frequently. The city-province example is just one of many where we are likely to perform queries that hit both attributes. Another example might be first and last name.

It may help to think of composite keys as just sort of smashing together the two attributes (perhaps with some separator like a dash) so you end up with Waterloo-ON as the key. From there it works mostly like the regular search.

Assuming there is no index on city alone, if we wanted to find all addresses with the name Waterloo, this is possible. If the select query says to select all addresses where city equals "Waterloo", our search term in the tree structure is then something like "Waterloo-%". Seems simple enough.

**Combining Multiple Indexes.** Let us imagine that employees have an index defined on the fields ID, department, and salary. If we wish to do a select that finds the IDs of all employees with the department name "development" and salary above $100 000, there are three strategies for carrying out this query [SKS11]:

1. Use the index on the department to find all records for the development department. Check each record to see if the salary is above $100 000.

2. Use the index on salary to find all records for employees with a salary above $100 000. Check each record to see if the department equals development.

3. Use the index on the department to find all records for the development department. Use the index on salary to find all records for employees with a salary above $100 000. Compute the intersection of the pointers to find out which ones satisfy both criteria.

Which approach is best? It depends a lot on our data, but at a first glance only the third option makes use of both indexes, but it does require the computation of the intersection. The first approach may be very efficient if the number of employees in the development department is quite small. Similarly, the second approach is quite

efficient if the number of employees making over $100 000 is quite small. The third approach is perhaps best if neither the department nor the number of people with a large salary is small...

Even then, the third strategy may be a poor choice if there are a lot of employees in development, and a lot of employees with a salary about $100 000, but only a small number that fulfill both criteria (and do you really want to work there?). In that case we have to do a lot of work to produce a small number of results when one of the other approaches might have been faster.

This idea that there are three different ways to carry out the requested operation is actually an interesting and large topic. We will therefore take the next few lectures to discuss how the database server understands a query, optimizes it, and carries it out.

## References

[EN11]   Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[SKS11]  Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.

[Sta14]   William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.