The case for Learned Index Structures

Ceachi Bogdan
bogdan.ceachi@my.fmi.unibuc.ro
University of Bucharest – November 29, 2018

Introduction

In the present paper "The Case for Learned Index Structure," the author wants to present a new way through which we can visualize the components that make up a software system.

The main topic is about indexing issues that can help improve data management systems, but also various programming tasks by introducing machine learning concepts. The replacement of the base components from a data management system with “learned models” has a great implications for future systems. The data structures used for indexing (B-trees or HashMaps) can be replaced with machine learning (neural networks) models.

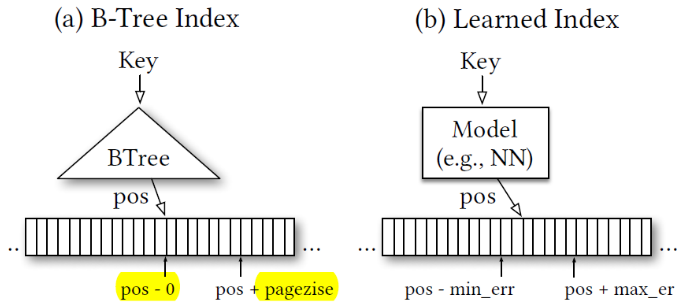
For example, either the function f(key) == pos, where the key parameter is the key that you want to be searched for, and the result pos parameter is the position of the key inside a sorted array.We can see f as a data structure/algorithm or more generally as a function that receives an input and returns a result, but also as a machine learning model. From here we can see that ML models presents a learning potentials, but also as a exploitation of existing data models in the real world.

The author demonstrates that traditional data structures can be replaced by so-called “**ML learned indexes”**, showing three types of such indexes

1) **Range index** (by receiving a key as input, the model will “predict”/return the location of a value within a key-sorted set)
2) **Point index** (by receiving a key as input, the model will return only one unsorted position)
3) **Existence index** (by receiving a key as input, the model will return if the key exists or not)

Thus, range and point indexes can be seen as regression models, and existence index as a classification task.

Range index:



For example: *" consider a B-Tree index in an analytics in-memory database(i.e, read-only) over the sorted primary key column. In this case, the B-Tree provides a mapping from a look-up key to a position inside a sorted array of records with the guarantee that the key of the record at that position is the first key equal or higher than the look-up key. The data has to be sorted". "....For efficiency reasons it is common not to index every single key of the sorted records, rather only the key of every n-th record i.e., the first key of a page".*

Thus, for this example, we visualize the memory region as a single array not physical pages witch are located in different memory regions.

"... Thus, the B-Tree is a model, or in ML terminology, a regression tree: it maps a key to a position with a min- and max-error (a min-error of 0 and a max-error of the page size), with a guarantee that the key can be found in that region if it exists.".

".. The B-Trees only provides this guarantee over the stored date, not for all possible data. For new data, B-Trees need to be re-balanced, or in machine learning terminology, re-trained, to still be able to provide the same eror guarantees".

From here we can come to the conclusion that if we want to run the model for each key and while we keep all the predictions, when we want a new search for a key that exists, it must fall within these limits.

As a result, the machine learned model: *"has the potential to transform the log(n) cost of a B-Tre lookup into a constant operation).*

Range Index Models are CDF Models:

Range Index Models is "a model that predicts the position given a key inside a sorted array effectively approximates the cumulative distribution function (CDF)".
Model: f(key) -> pos
This is equivalent to modelling the (CDF):

$$p = F(key) * N$$

p = predictedPos
F(key) = P(X <= Key)
N = total number of keys

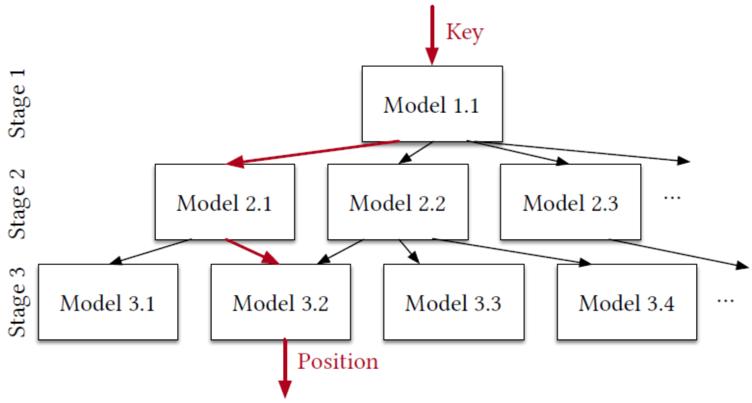
This observation, implies that indexing literally requires learning data distribution. A B-Tree "learns" the data distribution, by building a regression tree. A linear regression model would learn the data distribution by minimizing the (squared) error of a linear function.

The Recursive Model Index:

"... As can be seen, the learned index dominates the B-Tree index in almost all configurations by being up to 1.5 – 3x faster".

Solution is to layer models in a recursive regression model:

"... we build a hierarchy of models, where at each stage the model takes the key as an input and based on it, picks another model, until the final stage predicts the position. ".



- Different models at the same stage may map to the same model at the next stage
- We can think of this model like a professional who knows very well some certain keys".
- The entire index (all stages) can be represented as a sparse matrix.

Hybrid Indexes = build mixtures of models, so we can use traditional B-Trees at the bottom stage if the date is particulary hard to learn.

Algorithm 1: Hybrid End-To-End Training

```
Input: int threshold, int stages[], NN_complexity
Data: record data[], Model index[][]
Result: trained index
M = stages.size;
tmp_records[][];
tmp_records[1][1] = all_data;// get the entire dataSet
for i ← 1 to M do                      // for every model per stage
    for j ← 1 to stages[i] do
        index[i][j] = new NN trained on tmp_records[i][j]; // train the model [i][j] with the tmp_records[i][j] data
        if i < M then // if we are not on the last stage
            for r ∈ tmp_records[i][j] do // for every records in timp_records[i][j]
                p = index[i][j](r.key) / stages[i + 1]; // get the (prediction) // get the next model from the next stage to send data
                tmp_records[i + 1][p].add(r);// add all keys wich fall into the model p
for j ← 1 to index[M].size do // for every model, on the last stage
    index[M][j].calc_err(tmp_records[M][j]);// we calculate for model j the error for the set of data that we get
    if index[M][j].max_abs_err > threshold then // if absolute min/max-error is above a predefined threshold
        index[M][j] = new B-Tree trained on tmp_records[M][j]; // the model [m][j] will be a B-Tree trained on the tmp_records[M][j] data
return index;
```

Search Strategies and Monotonicity:

The authors develop a new based quaternary search.

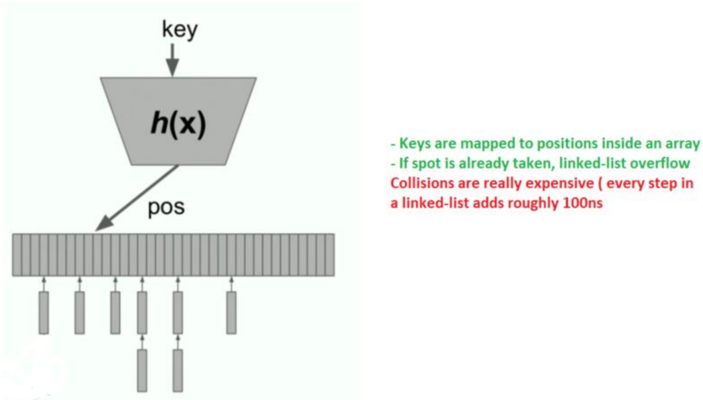
In a quaternary search, instead of picking one new middle point (as binary search), we are using three points (we are dividing the dataset into four quarters). The three initial middle points for the quaternary search are set to be $pos - \sigma, pos, pos + \sigma.$ where $pos.$ is the predicted position.

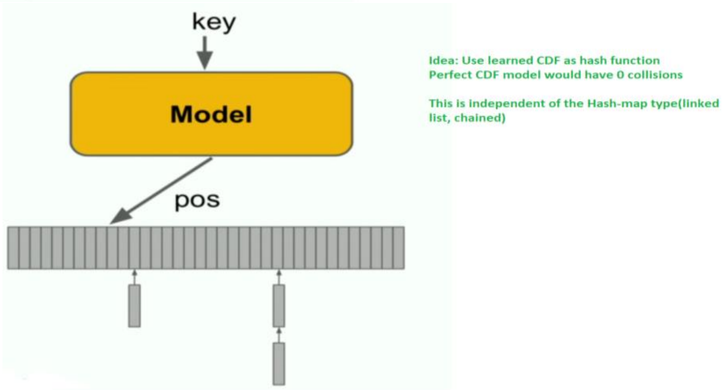
Point index:

Hash Maps are used to prevent too many distinct keys from mapping to the same position inside the map (conflict). In case of a conflict, separate chaining Hash-maps would create a linked-list to handle the conflict.

"... To our knowledge it has not bee explored if it is possible to learn models wich yield more efficient point indexes".

The Hash-Model Index:





*"Surprisingly, learning the CDF of the key distribution is one potential way to learn a better hash function... we can scale the CDF by the targeted size M of the hash-map and use h(K) = F(K) * M, with key K as our hash-function. If the model F perfectly learned the CDF, no conflicts would exist"*

So it seems, with this strategy, the learned models can reduce the number of conflicts by up to 77% over our datasets by learning the empirical CDF at a reasonable cost; the execution time is the same as the model execution time.

Existence Index:

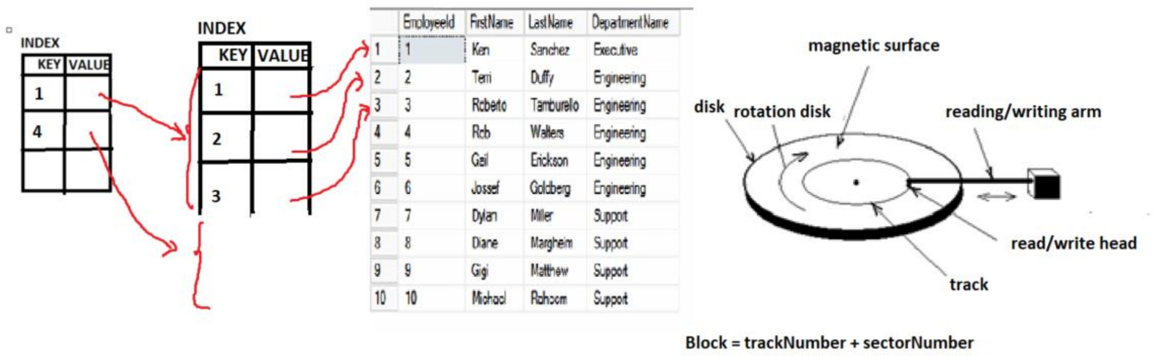
- The most commonly ones are **Bloom filters;**

Why use a bloom filter?

- If you want to be able to know if a certain element is not in the set. Wich allows you to avoid extra work for elements not in the set
- Allows us to store binary data, instead of keys, which takes less space

Bloom filters guarantee that there are no false negatives (if a Bloom filter says the key is not there, then definitely isn't there. They are highly space efficient.
From an ML perspective, we can think of a Bloom-filter as a classifier, which outputs the probability that a given key is in the database.

Creating B-Tree Index Structure



	EmployeeId	FirstName	LastName	DepartmentName
1	1	Ken	Sanchez	Executive
2	2	Terri	Duffy	Engineering
3	3	Roberto	Tamburello	Engineering
4	4	Rob	Walters	Engineering
5	5	Gail	Erickson	Engineering
6	6	Jossef	Goldberg	Engineering
7	7	Dylan	Miller	Support
8	8	Diane	Margheim	Support
9	9	Gigi	Matthew	Support
10	10	Michael	Raheem	Support

Its original purpose is to reduce the time being spent in computer hard drive by minimizing storage I/O operations as much as possible. The technique has served very well in computer fields such as database and file system. With the time being, big-data and NoSQL distributed database systems (due to cheap hardware and internet growth) B-Tree and its variants are playing more important role than ever for data storage.

Definition and Properties:

A **B-Tree (or M-way branching) T** is a tree with root (root[T]) which has the following 5 properties:

Each node x has the following fields:

a. n[x] = the current number of keys stored in x,
b. the n[x] keys, are stored in ascending order: $key_1[x] \le key_2[x] \le ... \le key_{n[x]}[x]$;
c. the Boolean value `leaf[x]`, which is `true` if x is a leaf node and `false` if the node is an internal one.

2. If x is an internal node, he contains `n[x]+1` pointers to his sons $c_1[x], c_2[x], ..., c_{n[x]+1}[x]$.
   The leaves nodes have no sons, so their fields c_i are undefined.

3. The keys **key[x]** separates key fields in each sub-tree: if $k_1, k_2 ... k_{n[x]+1}$ is a key system stored in a root sub-tree $c_i[x]$, then:

$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le ... \le key_{n[x]}[x] \le k_{n[x]+1};$$

4. Each leaf have the same depth, which is the height **h** of the B-tree
5. There is a lower and a higher limit of the number of keys that can be contained in a node. These margins can be expressed by a fixed integer $t \ge 2$, called the minimum degree of the B-Tree:
   a. Each node, with the exception of the root, must have at least t-1 keys and consequently at least t sons; if the tree is not null, then the root must have at least one key;
   b. Each node can have at most **2t-1 keys**; therefore, any internal node may have no more than 2 sons; a node with **2t-1** keys will be called a full node**.**

   Therefore $t-1 \le n[x] \le 2t-1$ and

$$t \le nr.sons \le 2t$$ (number of sons or pointers)

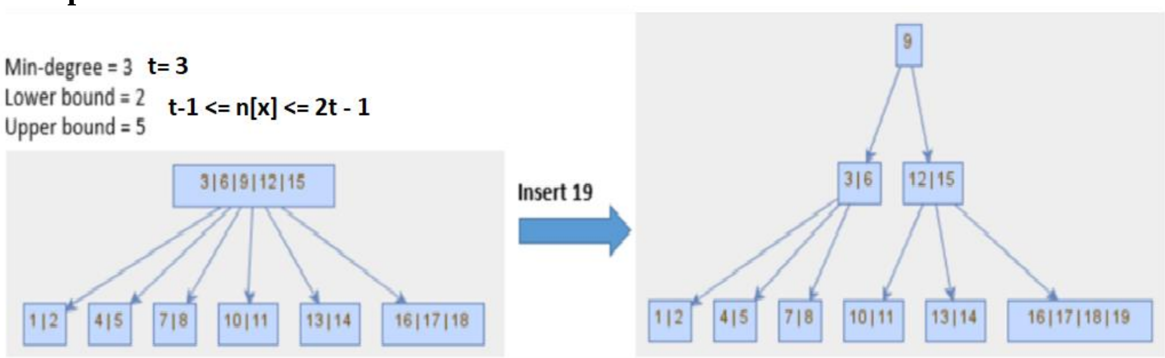
! The simplest B-Tree occurs when $t=2$. Any internal node can have **2,3** or **4** sons, (and **1,2** or **3** keys) .

The number of disks access is proportional to the B-Tree height.

Property.

If $n \ge 1$, then, for any **B**-Tree **T** with **n** keys of height **h and the minimum degree t** ≥ 2, we have : $h \le \log_t(n+1)/2$

Examples of Insertion:



Examples of Deletion:

If we want to delete a key from the tree, we can get the following cases:

Case - 1: if x is a leaf node and x has >= t keys then

Just delete the key from node-x;

Case - 2: The node x containing the target key is a leaf and x has exactly (t-1) keys then:

If x has a sibling with at least t keys, then move x's parent key into x and move the appropriate extreme from x's sibling into the open slot in the parent node. Then delete the target key.

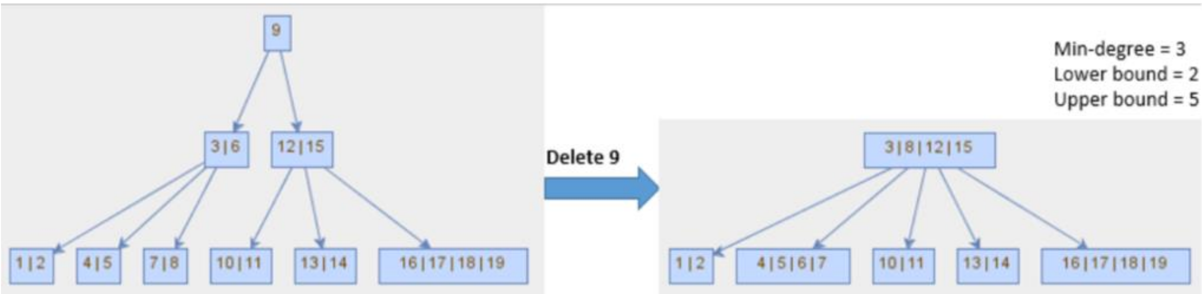
If x's siblings also have (t-1) keys, merge with one of it's sibling by bringing down the parent key as the median key. Then delete the target key.

Case – 3: If the node x containing the target key is an internal node.

If the target key's left child has at least t keys then it's largest value can be moved to the parent, to replace the target key.

If the target key's right child has at least t keys, then it's smallest value can be moved to the parent to replace the target key.

If none of the target key's children have at least t keys then the children must be merged into one and the key could be removed.



General concepts for implementing B-Tree operations

High level view:

- Right/left simbling of a node = are the nodes on it's right and left site at the same level
- Predecessor = is a leaf node within the subtree on the left side of the key and it contains key whose value is the largest one within that subtree.
- Successor = is a leaf node

How to search for a key:

```
Key-Search (searchedKey)
    currentProcessedNode = rootNode
    while ( currentProcessedNode is not NULL)
        currentIndex = 0
        while(( currentIndex < key number of currentProcessedNode AND
                (searchedKey > currentProcessedNode.Keys[currentIndex]))
            currentIndex++
        end while
        if(( currentIndex < key number of currentProcessedNode) AND
            (searchedKey == currentProcessedNode.Keys[currentIndex]))
            searchedKey is found
            return it
        currentProcessedNode = Left / Right Child of the currentProcessedNode
    end while
    return NULL
```

How to Split-Node:

```
Split – Node (parentNode, splittedNode)
        create a new node
        Leaf[new-node] = Leaf[splitted-node] (The new node must have the same leaf info)
        Copy right half of the keys from splitted-node to the new node
        if(Leaf[splittedNode] is FALSE) then
            Copy right half of the child pointers from splittedNode to the new node
        end if
        Move some of parent children to the right accordingly
        parentNode.keys[relevant index] = splittedNode.keys[the right-most index]
```

How to instert a key to a node:

```
Insert-Key-To-Node(currentNode, insertedKey)
    if(Leaf[currentNode] == TRUE) then
        put inserted-key in the node in the ascending order
        Return (We are done)
    end if
    Find the childNode where insertedKey belong
    if(total number of keys in childNode == UPPER BOUND) then
        Split-Node(currentNode, childNode)
        return Insert-Key-To-Node(currentNode, insertedKey)
    end if
    Insert-Key-To-Node(childNode, insertedKey)
```

Hot to insert a key into B-Tree:

```
Insert-Key(inserted key)
    if( rootNode is NULL) then
        Allocate for rootNode
        Leaf[rootNode] = TRUE
    end if

    if(total number of keys in rootNode == UPPER BOUND) then
        create a new node
        assign rootNode to be the child pointer of the new node
        Split-Node(new-node,new-node.children[0])
    end if
    Insert-Key-Node(new-node, inserted-key)
```

How to delete a node:

```
Delete-Key-From-Node(parent-node, current-node, deleted-key)
    if(Leaf[current-node] == TRUE) then
        Search for deleted-key in current-node
        if(deleted-key not found) then
            return (not found)
        end if
        if(total number of keys in current-node > LOWER BOUND) then
            Remove the key in current-node
            return (done)
        end if
        Get left-sibling-node  and right-sibling-node of current-node
        if(Left-sibling-node is found total number of keys in right-sibling-node > LOWER BOUND) then
            Remove deleted-key from currentNode
            Perform left rotation
            return (done)
        end if
        if(left-sibling-node is not NULL) then
            Merge current-node with left-sibling-node
        else
            Merge current-node with right-sibling-node
        end if
        Return Rebalance-Btree-Upward(current-node)
        end if
    Find predecessor-node of current-node
    Swap the right most key of predecessor-node and deleted-key of current-node
    Delete-Key-From-Node(predecessor-parent-node, predecessor-node, deteled-key)
```

How to Rebalance-BTree:

```
Rebalance-Btree-Upward(current-node)
    Create-Stack
    for each step of the path from root-node to current-node then
        Stack.push(step-node)
    end for
    While(Stack is not empty) then
        step-node = Stack.pop()
        if(total number of keys in step-node < LOWER BOUND) then
            Rebalance-Btree-At-Node(step-node)
        else
            return (done)
        end if
    end while
```

```
Rebalance-Btree-At-Node(step-node)
    if(step-node is NULL OR step-node is root-node) then
        return (done)
    end if
    Get left-sibling-node and right-sibling-node of step node
    if(left-sibling-node is found AND total number of keys in left-sibling-node > LOWER BOUND)
        Remove deleted-key from step-node
        Perform right rotation
        Return (done)
    end if
    if(right-sibling-node is found AND total number of keys in right-sibling-node > LOWER BOUND)
        Remove deleted-key from step-node
        Perform left rotation
        Return (done)
    end if
    if(left-sibling-node is not NULL) then
        Merge step-node with left-sibling-node
    else
        Merge step-node with right-sibling-node
    end if
```

```
Delete-Key(deleted-key)

    Delete-Key-From-Node(NULL, root-node, deleted-key)
```

Bibliography

- **The Case of Learned Index Structures** Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dead, Neoklis Polyzotis
- **Introduction to Algorithms Third Edition** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein