Implementation and Analysis of Fusion Trees

July 6, 2022

Gopesh Gaba (2020MCB1236) ,
Ishaan Chhabra (2020MCB1238) ,
Rohan Kumar (2020MCB1247)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Simran Setia

Summary: In this paper we will discuss the theory behind fusion trees and also how to implement it, followed by an analysis of it's time and space complexity

---

1. Introduction

In 1990, Michael Fredman and Dan Wilfred introduced a new sub-logarithmic data structure for searching: The Fusion Tree. The idea behind Fusion trees was mostly geared toward proving that it was possible to surpass the $\Omega(logn)$ lower bound on BST operations by using word-level parallelism and without regard to wall-clock runtime costs.
This fusion tree stores n w-bits statistically and performs predecessor/successor problems in $O(log_w n)$ time, and requires O(n) space, where n is the number of elements stored. Two essential bit operations utilized by fusion trees are parallel comparison and sketching.

Fusion Trees

Essentially the structure of Fusion Trees is a B-Tree with branching factor $w^{1/c}$ where w is the word size and c is some constant greater than 1. This means that the height of the tree will be $log_w n$.

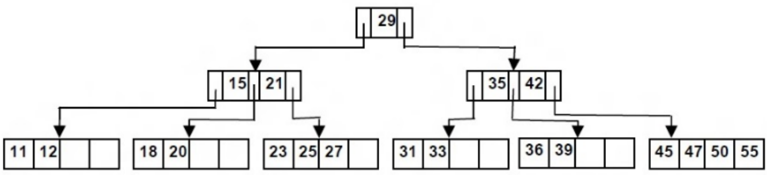

Figure 1: Fusion Tree is essentially a B-Tree

The main problem arises when we try to make searching a node take O(1) time. As each node would have $O(w^{1/c})$ keys and each key has w bits, we would be required to read atleast $O(w^{1+1/c})$ bits. But we can only read w-bits in constant time, so achieving $O(1)$ time seems impossible.
This problem can be solved, using $k^{O(1)}$ preprocessing, where k is the number of keys present in the node. We have to distinguish the set of keys in one node with less than w-bits. We achieve this with SKETCHING and PARALLEL COMPARISONS.

2. Sketching

We wish to reduce the size of the bits being compared, in order to obtain w-bits for comparison in total. As there are w bits in a word, and there are $w^{1/c}$ keys at most in a node, there can only be at most $w^{1/c} - 1$ bits which matter when trying to compare all the keys. These bits are called IMPORTANT BITS.
Sketching refers to the extraction of these important bits from the specified word, i.e sketch(X) refers to extracting of the important bits from the word X. Let there be r important bits per key.Then sketch(X) is the r-bit vector whose i-th bit = b_ith bit of word X. We can see that sketch(X_0) < sketch(X_1) ..... < sketch(X_{k-1})

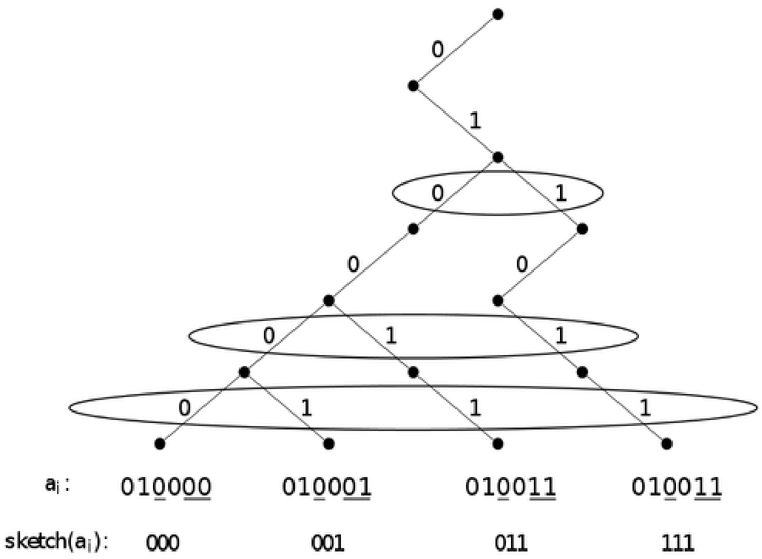

Figure 2: Sketch of Node

We tend towards APPROXIMATE SKETCHING instead of exact sketch because exact sketch cannot be computed in O(1). Prefect sketch requires the distinguishing bits to be compressed which a computation expensive operation. Approximate sketch doesn't compress the distinguishing bits tightly, rather it adds a fixed pattern of zeroes between every two distinguishing bits. This is done by multiplication with a predetermined constant. The result will be of order $O(r^{c-1})$ where r is count of distinguishing bits. Let Mask be the mask that will filter out distinguishing bits and m be a predetermined constant. x' = x AND Mask. After bitwise AND, x' will have only distinguishing bits. result = x' * m. Right shift result until it is r^{c-1} bits long. Result will then be the approximate sketch

3. Node Searching (Desketchifying)

Let us search query q. We can check where sketch(q) fits in sketch(X_0) < sketch(X_1)..... < sketch(X_{k-1}). But that is not necessarily where q fits in $X_0 < X_1$ ... $< X_{k-1}$. So we will use parallel comparison for this.

Parallel Comparison

Parallel comparison is used to find position of a value within the set of keys in a node in constant time. A node will have at most r + 1 keys. The sketches of all those keys can be concatenated to form a word. This word is used to compare query value q directly with all the keys at once rather than comparing one by one.

1. Let X_0, X_1, ... ,X_r be the keys.
2. Then the sketch of a node is 1sketch(X_0)1sketch(X_1)... 1sketch(X_r).
3. Convert sketch(a) to sketch(q)' = 0sketch(q)0sketch(q)... 0sketch(q).
4. After subtracting value at step 3 from value at step 2, we will get a number that will be like : 0_ _ _ _ _ _ _ _0_ _ ... _ _ _0_ _ _ _ _ _ _ _1_ _ _ _ _ _ _ _1_ _ _ _ _ _ _ _1. Call it diff.
5. There are 0's where sketch of X_i is less than sketch(q) and 1 when X_i is greater than sketch(q). The dashes represent garbage bits where sketch values used to be. It will always be a series of 0's followed by a series of 1's since sketch operation maintains order of keys.The position where the series of 0 turns into 1 is the position

where sketch(q) should be placed. So sketch(X_i) < sketch(q) < sketch(X_{i+1}), when bit of diff for sketch(X_i) = 0, and for sketch(X_{i+1}) is 1

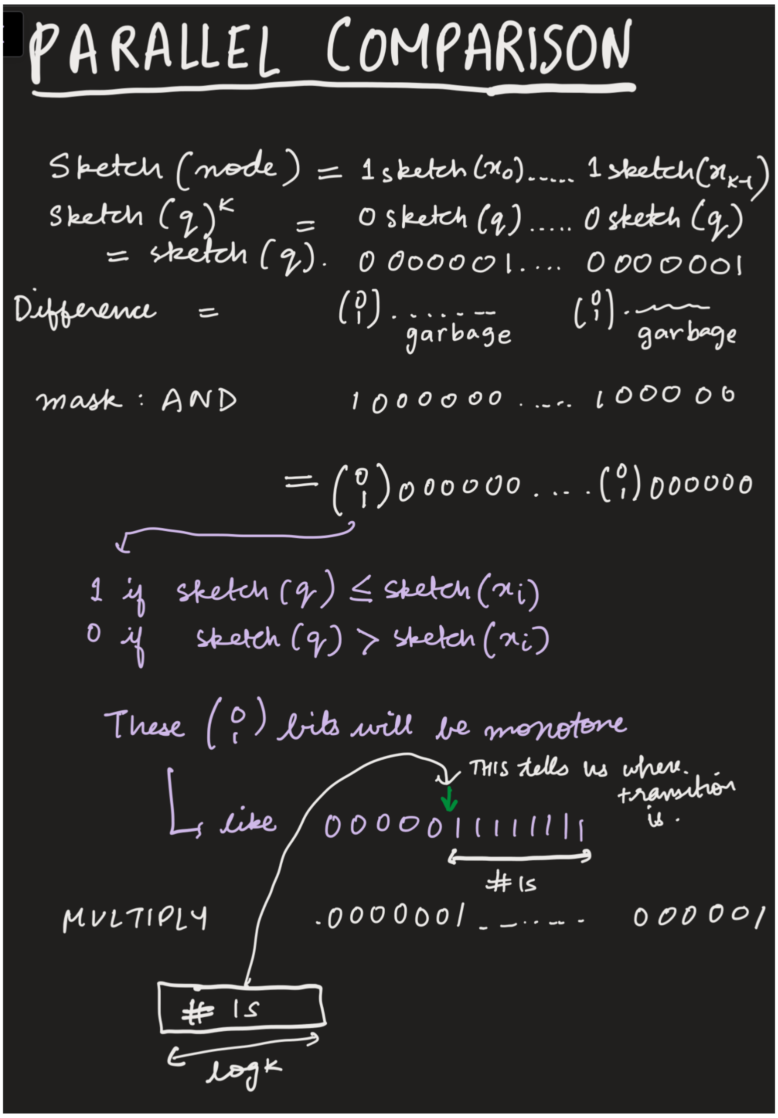

Figure 3: Parallel Comparison Demonstration

Note that parallel comparison gives index such that sketch of k is tightly bound with the sketches of keys, not the keys itself.
Now let us find where q fits among the keys. We will use parallel comparison to find X_i such that sketch(X_i) <= sketch(q) < sketch(X_{i+1}). The longest common prefix is the lowest common ancestor between q and either X_i or X_{i+1} , whichever is the longest or lowest. Let node y be where q fell off(the keys).
If bit y+1 = 1: e = y011111....
If bit y+1 = 0: e = y100000.... .
Then find sketch(e) among sketch(X_i)s. Finally, the predecessor and successor of sketch(e) among sketch(X_i)s will be equal to the predecessor and successor of q among X_is.

4. Time Complexity

The time complexity for searching an element or finding it's predecessor and successor is $O(log_w N)$, where N is the number of elements in total and w is the word size being used to store the data .
For inserting an element, time complexity will be $O(log_2 w)$ and for deleting an element time complexity will be $O(log_2 w)$ as well.
The space complexity of our constructed tree will be of $O(N)$.

5. Algorithms

Pseudo-algorithms can be written in LaTeX with the `algorithm` and `algorithmic` packages.

Algorithm 1 Parallel Comparison(root rt, node p, int data)

```
sketch = SKETCHAP( rt, p, data)
sketch_long = sketch*(p->mask_q)
res = p->node_sketch - sketch_long
res = res AND p->mask_sketch
i = 0
while ((1 « i) < res) do
   i++
end while
i++
sketch_len = p->n*p->n*p->n + 1
return  (p->n - (1/sketch_len))
```

Algorithm 2 Approximate Sketching

```
SKETCHAP(self, node, x)
xx = x & node.mask_b
res = xx * node.m
res = res & node.mask_bm
return  res
```

Algorithm 3 Initiating Node

```
initiateNode(self, node)
if node.key_count! = 0 then
   node.b_bits = self.getDiffBits(node.keys)
   node.m_bits, node.m = self.getConst(node.b_bits);
   node.mask_b = self.getMask(node.b_bits)
end if
initialize temp
for i in range(len(node.b_bits)) do
   temp.append(node.b_bits[i] + node.m_bits[i])
end for
node.mask_bm = self.getMask(temp)
r3 = int(pow(node.key_count, 3))
node.node_sketch = 0
sketch_len = r3 + 1
node.mask_sketch = 0
node.mask_q = 0
for i in range(node.key_count) do
   sketch = self.sketchApprox(node, node.keys[i])
   temp = 1 « r3
   temp |= sketch
   node.node_sketch «= sketch_len
   node.node_sketch |= temp
   node.mask_q |= 1 « i * (sketch_len)
   node.mask_sketch |= (1 « (sketch_len - 1)) « i * (sketch_len)
end for
return
```

Algorithm 4 Finding Successor

```
successor(self, k, node = None)
if  node == NULL then
  node = self.root
end if
if node.key_count == 0 then
  if node.isLeaf  then
    return  -1
  else
    return  self.successor(k, node.children[0])
  end if
end if
if node.keys[0] >= k then
  if not node.isLeaf then
    res = self.successor(k, node.children[0])
    if res == 1 then
      return  node.keys[0]
    else
      return  min(node.keys[0], res)
    else
      return  node.keys[0]
    end if
  end if
end if
if node.keys[node.key_count - 1] < k then
  if node.isLeaf then
    return  -1
  else
    return  self.successor(k, node.children[node.key_count])
  end if
end if
pos = self.parallelComp(node, k)
if pos >= node.key_count then
  print(node.keys, pos)
  dump = input()
end if
if pos == 0 then
  pos += 1
end if
x = max(node.keys[pos - 1], node.keys[pos])
common_prefix = 0
i = self.w
while i >= 0 and (x & (1 « i)) == (k & (1 « i)) do
  common_prefix |= x & (1 « i)
  i -=1
end while
if i == -1 then
  return  x
end if
temp = common_prefix | (1 « i)
pos = self.parallelComp(node, temp)
if node.isLeaf then
  return  node.keys[pos]
else
  res = self.successor(k, node.children[pos])
  if res == -1 then
    return  node.keys[pos]
  else
```

6. Conclusion

Fusion Trees are a modification of B-trees where each node contains some special information that helps speed up the search. Fusion Trees use word-level parallelism to do searches faster by packing multiple numbers into a single machine word and using individual operations to speed up processing. Sketching is used to reduce the number of bits in the input numbers to a point where parallel processing with a machine word is possible. All these operations help Fusion Trees to obtain an exceptional time complexity of $O(log_w N)$ while searching. Fusion tree are used extensively in database systems. Fusion tree and Van Emde boas tree are used together such that when word size is large, fusion tree is used and when word size is smaller, Van Emde Boas tree is used. This is because fusion trees are better at solving predecessor and successor problems when the universe is large.

7. Bibliography and citations

While doing this project we referred to different researches and following are some sources which we referred to:
[1] Michael L. Fredman and Dan E. Willard. Surpassing the information Theoretic bound with Fusion Trees. *Journal of Computer and System Sciences*, pages 47:424–436, 1993
[2] Introduction to Algorithms 3rd edition, by Cormen, Leiserson, Rivest, and Stein
[3] https://iq.opengenus.org/fusion-tree/
[4] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/calendar-and-notes/MIT6_851S12_L12.pdf