CS265 PROJECT – COMPARISION OF BTREE IN SCALA VS C

ABSTRACT

Traditional database systems are written in System programming language - C. C is highly performant as it provides full control over every byte of data movement and memory allocation and deallocation. While very powerful C is also challenging to get right due to numerous details the programmers need to. Usually systems written in C have longer time to market and are known for difficult to trace runtime errors. In contrast a Strong Statically Typed Functional language such as Scala is relative easier to code in, and can provide compile time error checking. This paper studies the design and performance of an immutable BTree written in Scala to a mutable BTree written in C. Performance experiments shows that building a BTree in Scala has a bigger memory footprint and is slower when compared with C, but the range queries run more efficiently in Scala.

1. INTRODUCTION

The rapid growth of internet in 1990's was also a major historic event in programming languages and propelled the growth of new "Rapid Application Development" (RAD) languages. These languages usually came with an IDE and a Garbage Collector. One of the most notable languages of this era is Java – which introduced the concept of "Write Once, Run Anywhere" – by inventing the JVM (Java Virtual Machine). The JVM is an abstract computing machine that can run a Java program. A concrete JVM executes the Java bytecodes and comes with a Garbage Collector.

As the growth of Java and object oriented languages continued, there was an increased awareness of Functional languages. As programming languages evolution continued there was an increasing trend for bringing functional language constructs into mainstream programming, as they make code easier to reason about and easier to parallelize. The first version of Scala was released in 2003.

Scala is a functional programming language with a strong statically typed system. Scala code runs on a JVM, is Object oriented and Functional. Some of the functional constructs available in Scala include Lazy evaluation, Type inference, Immutability and Pattern matching. Its advance type system supports higher order functions, higher kinded types, covariance, contravarience. Scala's Type System is Static, which means it will help to find bug statically i.e. at compile time.

As the growth of new powerful programming languages continues, System Programming continues to predominantly be done in C. C is an imperative programming languages that provides low-level access to memory, with language constructs that map efficiently to machine code. While C can be highly performant , it also has the possibility of runtime errors (leading to core dumps) and Segfaults. It also lacks support for simple data structures such as List, Maps, Sets – which are available in all more advanced programming languages and needed for even the simplest of application development. All of these lead to longer application development and test cycles and slows down the time to market.
With all the work put into modern languages, developers should be able leverage these languages and the frameworks built using these languages, to write highly performant code. This will reduce the occurrences of runtime errors and reduce the time to market these applications.

This paper investigates the performance of the database index – BTree - implemented in Scala by comparing its performance to BTree implemented in C. We study the characteristics of the memory profile and range search

of the Tree for various data loads. One of the main differences in the two implementations is that the Scala tree is Immutable, why the C tree is mutable. Otherwise the two implementations have the same design approach.

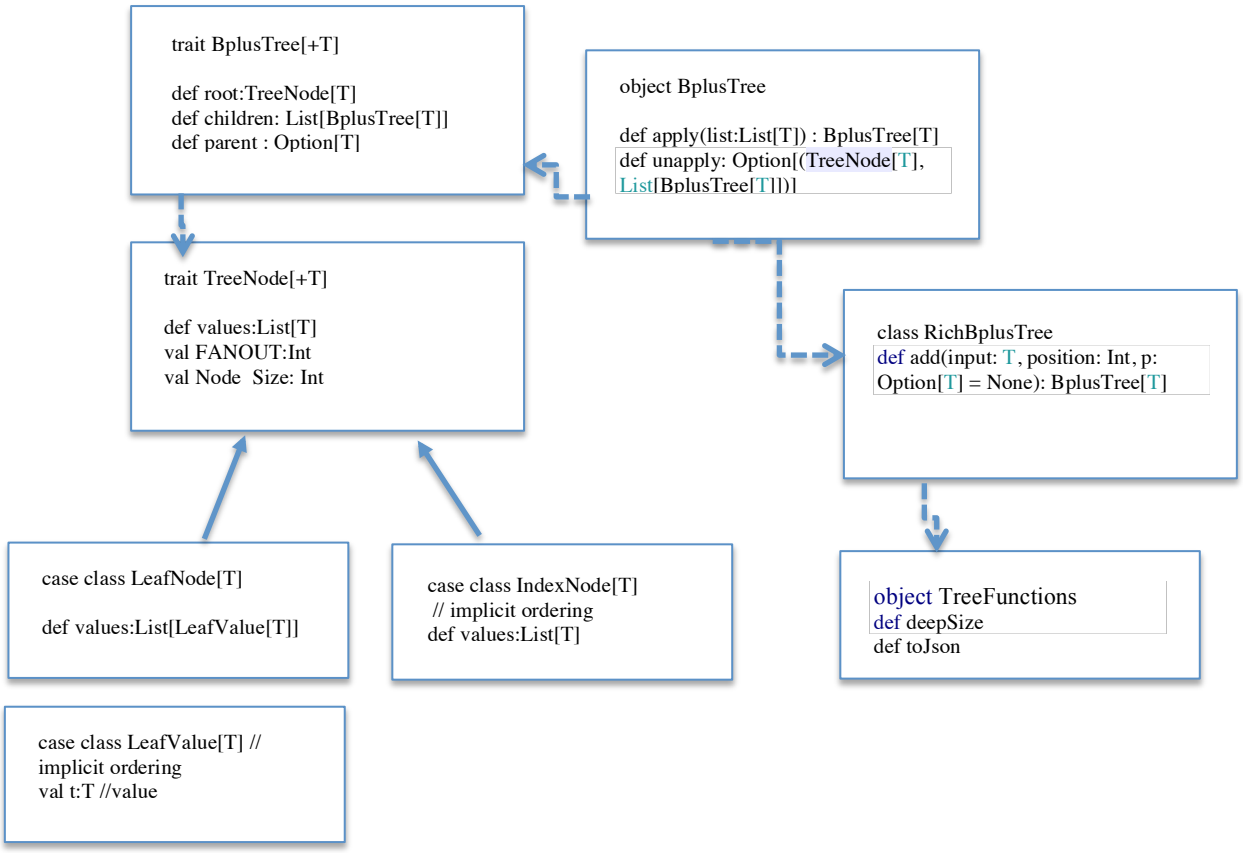
In the following sections of the paper we will study the design of the BTree in Scala and C and then discuss the results of the performance experiments and finally conclude with avenues for future work.

2. DESIGN

In this section we first present the design of the Scala Tree and the insert and search operations on this Tree, followed by same for C.

2.1 Scala

Scala tree is implemented as an immutable and Typed tree. Due to this typed property the same implementation can be used for building a String, Int, or any type of Tree – as long as the Type has a ordering defined for it.



The trait BplusTree is an interface and defines all operations that a concrete Tree must implement. Its convariant in the type T – This allows an EmptyTree (BplusTree[Nothing]) to be a BplusTree[T].
The root of the tree is a TreeNode, which can be either a LeafNode or an IndexNode. Both LeafNode and IndexNode have implicit ordering defined , which is the same ordering as that of type T. Note the LeafNodes of

the tree are not linked unlike a traditional Tree where all the LeafNodes are connected by a pointer to the next node.

New instance of the BplusTree are created via the object BplusTree calling its various apply functions. Operation on the Tree are defined via RichBplusTree . A BplusTree will be implicitly converted to a RichBplusTree and make all operations defined on the RichBplusTree available on the BplusTree instance.

TreeFunctions contain utility methods such as finding the size (memory) of the Tree , printing a JSON representation etc are defined here.

A simple socket client server is used to run the Tree in the server process and execute range queries via the client.

2.1.1 Insert

Inserting a new element in the Tree is defined by the add function of the RichBplusTree.

2.1.1.1. Adding element(s) to an EmptyTree :

When the tree is being created for the first time – it is defined as an add operation on an EmptyTree. The code below shows creating tree from a list of elements. The list of elements can contain one or more elemnts. Notice how we add elements by folding over an EmptyTree. Each addition created a newTree , akin to adding an item to a List, Map in Scala standard API.

```
def apply[T](input: List[T])(implicit ord: Ordering[T]): BplusTree[T] = {

  input.foldLeft(BplusTree.empty[T], 1) {
    case ((res, pos), in) =>
      {
        val r = (TypedTree(in, pos, res), pos + 1)
        r
      }
  }._1

}
```

2.1.1.1.1 EmptyTree

Is a singleton, which in scala can be easily achived by defining it as a case object and defining a function in BplusTree object that returns this singleton

// in BplusTree object

```
def empty[T]: BplusTree[T] = EmptyTree


/** Singleton representation of an empty Tree
 */
case object EmptyTree extends BplusTree[Nothing] {
  val root: TreeNode[Nothing] = LeafNode(List.empty)

  val children: List[BplusTree[Nothing]] = Nil

  override def parent: Option[Nothing] = None
}
```

2.1.1.1.2 TypedTree

Is a case class that represents a Tree created by adding an element to a baseTree.

(a) When the baseTree is an EmptyTree , its results in a Tree whose root is a LeafNode and has no children
(b) When the baseTree is not an EmptyTree, its calls the add function (defined in following section) on the RichBplusTree.

```
/** Create a tree from base tree.
 */
case class TypedTree[T](elem: T, pos: Int, baseTree: BplusTree[T])(implicit ord: Ordering[T]) extends BplusTree[T] {

  lazy private val newTree = baseTree match {

    case a @ EmptyTree => new BplusTree[T] {
      val root = LeafNode[T](List(LeafValue(elem, pos)))

      val children = Nil

      val parent: Option[T] = None
    }

    case _ => baseTree.add(elem, pos)

  }

  val root: TreeNode[T] = newTree.root

  val children: List[BplusTree[T]] = newTree.children

  override def parent: Option[T] = None

   override def toString() = s"Total children = ${children.size} and root size is ${root.values.size}"
}
```

2.1.1.1. Adding element(s) to non-empty Tree – RichBplusTree.add

RichBplusTree defines a recursive add operation , that adds element to the appropriate LeafNode and conditionally split and promote and create IndexNode. An important point to note is that each of the following opeations results in a new Tree. The original Tree that is present before the add , is left untouched and will eventually get garbage collected.

Add operation Steps: when the root is an IndexNode:

Step 1 - Find the Leaf that should contain this new element.
Step 2 – Add the element to this leaf Tree (this add is the same add)
Step 3 – Remove the old leaf from the Tree. This returns a newTree without that LeafNode

Step 4 – Add back newLeaf to the Tree returned from operation 3. This step involves finding the parent that should contain this new LeafNode, adding the newLeaf, promote and dedupe , updating parents and finally ensure the tree is balanced. (some parts of the code is shown below in the function addNodeAndRebalanceTree)

Step 5 – If a node is not the target – we copy it back in place in the new tree. This ensure that we always have all the Children of the Tree.

Top level add operation :

```
def add(input: T, position: Int, p: Option[T] = None): BplusTree[T] = {

 tree match {
  // only is a leaf
  case BplusTree(r: LeafNode[T], Nil) => {

   val toAdd: LeafValue[T] = LeafValue(input, position)
   val newLeaf = (toAdd :: r.leafValues).sorted

   if (newLeaf.size <= r.NODE_SIZE) {
    BplusTree(LeafNode(newLeaf /*, next = r.next*/ ), Nil, p)
   } else {
    // need to split and create new root
    val (rootNode, leafNodeLeftValues, leafNodeRightValues) = splitNodeAndAddInput(r.leafValues, input, position)
    val newRoot = IndexNode(rootNode)
    val rightLeafNode: LeafNode[T] = LeafNode(lValues = leafNodeRightValues /*, next = r.next*/ )
    val leftLeafNode = LeafNode(lValues = leafNodeLeftValues /*, next = Option(rightLeafNode)*/ )
    val rightTree = BplusTree(rightLeafNode, Nil, newRoot.values.headOption)
    val leftTree = BplusTree(leftLeafNode, Nil, newRoot.values.headOption)
    BplusTree(newRoot, List(leftTree, rightTree), p)
   }

  }

  // Tree with root and children
  case t2 @ BplusTree(r: IndexNode[T], c: List[BplusTree[T]]) => {
   // walk to the leaf node and build the tree bottom up.

   //println(s"adding input ${input} at position ${position}")
   val (originalTree, targetLeaf) = findTargetLeafNode(input)
   val updatedLeaf = targetLeaf.add(input, position, targetLeaf.parent)
   val newTree = addLeafAndRebalanceTree(originalTree, updatedLeaf, targetLeaf)
   newTree

  }

 }
}

// the function that adds and ensure the final tree is balanced.
private def addNodeAndRebalance(treeToAddAndRebalance: BplusTree[T], nodeToAdd: BplusTree[T]): BplusTree[T] =
{

 val heightNodeToAdd = nodeToAdd.height()

  def addToThisTree(thisTree: BplusTree[T]): BplusTree[T] = {
   thisTree match {

     case EmptyTree                    => nodeToAdd

     case BplusTree(lr: LeafNode[T], Nil) => thisTree

     case BplusTree(ir: IndexNode[T], c) if (c.size == 0) => { // this happends when the only child was removed
```

```
        // simply verufy that the parents of both thisTree and toAdd are same and make this node the child.
        //println("added nodeTOAdd here ... in without child")
        if (thisTree.parent == nodeToAdd.parent) {
          nodeToAdd match {
            case BplusTree(ir: IndexNode[T], ic) => promoteAndMerge(thisTree, nodeToAdd)
            case BplusTree(lr: LeafNode[T], Nil) => BplusTree(thisTree.root, List(nodeToAdd), thisTree.parent)
          }
        } else if (nodeToAdd.parent.isDefined && thisTree.root.values.contains(nodeToAdd.parent.get)) { // this tree is
parent of this nodeToAdd
          BplusTree(thisTree.root, List(nodeToAdd), thisTree.parent)
        } else {
          sys.error(s"Found a tree that has no children and also not a valid root for nodeTOAdd. Exiting .....")
        }

      }

      case BplusTree(ir: IndexNode[T], c) if (c.size > 0 && !TreeFunctions.isLeafNode(c.head)) => {

        // find the node of the parent.
        //
        val searchedAndMayBeUpdatedChildren: List[BplusTree[T]] = thisTree.orderedChildren.map {
          thisChildTree =>
            {
              addToThisTree(thisChildTree)
            }
        }
        // finally rebalance root tree  ??
        val topLevelTree = BplusTree(ir, searchedAndMayBeUpdatedChildren, thisTree.parent)
        topLevelTree.rebalanceThisTree

      }

      case BplusTree(ir: IndexNode[T], c) if (c.size > 0 && TreeFunctions.isLeafNode(c.head)) => {

        val foundMatch = nodeToAdd.parent.map { pp => ir.values.contains(pp) }.getOrElse(false)

        if (foundMatch) { // found the match

          // first check that the parent has same number of indexNodes as children ( as one of the should have been removed
earlier and created a hole

          if (ir.values.size >= c.size) {

            // now check if the nodeToAdd has the same height as that of the other children of the tree
            val heightChild1 = c.head.height()
            if (heightChild1 == heightNodeToAdd) { // if same height simply add to list of children and we are done
              // val updatedChildren = updateNextReference(c, nodeToAdd, ListBuffer())

              BplusTree(thisTree.root, (nodeToAdd :: c), thisTree.parent)
            } else if (heightChild1 < heightNodeToAdd) {
              val newTree = promoteAndMerge(thisTree, nodeToAdd)
              newTree
            } else {
              sys.error(s"Dont know what to do, height of children of this tree = ${heightChild1}  is greater than height of
nodeToAdd =${heightNodeToAdd}")
            }
          } else {
            //TreeFunctions.pretty_print(thisTree)
            sys.error(s"parent already has all the children - no Hole to fill. Error total index nodes ${ir.values.size} and
children =${c.size}!!!!")
```

```
        }
      } else { // not the match, check children
        val nr: ListBuffer[List[T]] = ListBuffer(thisTree.root.values)
        val nc: ListBuffer[List[BplusTree[T]]] = ListBuffer.empty[List[BplusTree[T]]]

        thisTree.orderedChildren.map {
          thisTreeChild =>
            {
              val updatedThisChild = addToThisTree(thisTreeChild)
              if (updatedThisChild == thisTreeChild) {
                nc += List(updatedThisChild)
              } else if (updatedThisChild.height() == thisTreeChild.height()) {
                nc += List(updatedThisChild)
              } else {
                println(s"heights are diff inputChildHeight =${} and returnedChildHeight = ${}")
                nr += updatedThisChild.root.values
                nc += updatedThisChild.children
              }
            }
        }
        // finally rebalance root tree
        BplusTree(nr.toList.flatten, nc.toList.flatten, thisTree.parent)
      }
    }
  }
 }

 addToThisTree(treeToAddAndRebalance)
}
```

2.1.2 Search

Range search involves finding the parent and sibling nodes for the given input. The reason we need to find the siblings at the parent level is to ensure that we don't miss any qualifying values that may be present in a LeafNode of a different parent.

Search Steps:

Step 1 - find all qualifying parent and sibling nodes for the given criteria. This returns a list of BplusTree
Step 2 – Traverse the list of qualifying trees , walk to their leaves and collect the qualifying values.

Code snippet for these provided below.

```
/** Finds all siblings that are greater than the pareent
 *
 *  @param originalTree
 *  @tparam T
 */
def findParentAndSiblingsTree[T](greaterThanEqual: T, lessThan: T, originalTree: BplusTree[T])(implicit ord:
Ordering[T]): List[BplusTree[T]] = {

 originalTree match {

   case BplusTree(ir: IndexNode[T], c) => {
```

```
    val firstIndex = ir.values.indexWhere(p => ord.gteq(p, greaterThanEqual))

    if (c.nonEmpty && isLeafNode(c.head)) {

      val rootValues = ir.values.slice(firstIndex, ir.values.size)
      val filteredC = c.filter(c => c.root.values.exists(p => ord.gteq(p, greaterThanEqual) && ord.lt(p, lessThan)))

      if (filteredC.nonEmpty) {
        BplusTree(rootValues, filteredC, originalTree.parent) :: Nil
      } else {
        Nil
      }

    } else {
      c.flatMap {
        findParentAndSiblingsTree(greaterThanEqual, lessThan, _)
      }
    }
  }

  case BplusTree(lr: LeafNode[T], Nil) => List(originalTree)

 }

}


def allLeafNodesValues[T](greaterThanEqual: T, lessThan: T, trees: List[BplusTree[T]])(implicit ord: Ordering[T]):
List[T] = {

 trees match {

  case BplusTree(lr: LeafNode[T], Nil) :: Nil => {

   lr.values.filter { yy => ord.gteq(yy, greaterThanEqual) && ord.lt(yy, lessThan) }
  }

  case BplusTree(ir: IndexNode[T], c) :: tl if c.nonEmpty && TreeFunctions.isLeafNode(c.head) => {

   val selected = trees.flatMap {
    currTree =>

     val resultOfThisTree = currTree.children.flatMap(c => c.root.values.filter { yy => ord.gteq(yy, greaterThanEqual)
&& ord.lt(yy, lessThan) })
     resultOfThisTree
   }

   selected

  }

  case r => {
   println(s"Invalid search - should always return Index Node with LeafNode as children or LeafNodes, but found
${r}, returning a Nil")
   Nil
  }

 }}
```

2.2 C

In contrast to Scala the C tree consists of a header that defines structure and operations on tree and a utils that has helper functions .

The following snippet from header file shows the design of the tree in C.

```
// the Tree
typedef struct root {
        void* root_ptr;
        NodeType ptr_type;
} root;

// types of nodes.
typedef enum NodeType {
    LEAF,
    NON_LEAF,
} NodeType;

// the Index node
typedef struct bt_node{
        int* values;
        void** children;
        int len;
        //int child_len;
        NodeType childType;
        struct bt_node* parent;
}bt_node;

// Leaf node.
typedef struct leaf_node{
        int* values; // array of values for leaf node and keyValues for non-leaf node.
        int* positions;
        int len; // total number of values in the array
        struct leaf_node* next; // used to link leaf nodes - not used for non-leaf nodes
        struct leaf_node* prev; // same as above
        struct bt_node* parent;
}leaf_node;
```

2.2.1 Insert

Adding elements to a tree is done by iterating over the input and adding one element to the tree at a time. The state of the tree is stored in the pointer that is passed to the function as a parameter. Each addition mutates the state of the Tree stored in that pointer.

Steps to add an entry :

Step 1: Search for the leaf node that will store this new element
Step 2: If this is the first time , the allocate memory for a LeafNode and add element
Step 3 : Add the element to this new leafNode , update the number of elements store , maintaining the next and prev pointer.
Step 4: If not null, check if the leafnode had space to take a new entry. If so , find the right position – so as to maintain the sorted nature of the tree – and add the entry , updating the required fields
Step 5 : If leafNode cannot take this new entry, split the node and promote upwards, continuing to split and promote till all nodes are balanced
Step 6: LeafNode and Index Nodes that are not impacted by this entry are left untouched. This is one of the biggest difference between a Scala and C implementation. In scala all nodes not impacted atleast require a copy and add of that node. While in C those nodes are not even visited.

Add function in C :

```
void add(int n, int p, root** r){

        leaf_node* toAdd = search_node(n, *r, true);
        if(toAdd == NULL){ // first time= root creation.
                // create a new leaf node
                leaf_node* new_node = malloc(sizeof(leaf_node));
                if(new_node == NULL){
                        printf("Out of memory \n");
                        exit(-1);
                }
                new_node->len = 1;
                int* v = calloc(max_nodes, sizeof(int)*max_nodes);
                int* pos = calloc(max_nodes, sizeof(int)*max_nodes);
                if(v == NULL){
                        printf("Out of memory \n");
                        exit(-1);
                }
                new_node->values = v;
                new_node->positions = pos;
                new_node->values[0] = n;
                new_node->positions[0] = p;
                new_node->prev = NULL;
                new_node->next = NULL;
                new_node->parent = NULL;
                root* new_r = malloc(sizeof(root));
                new_r->root_ptr = new_node;
                new_r->ptr_type = LEAF;
                *r = new_r;
        }
        else{
                int ll = toAdd->len;
                if(ll < max_nodes){
                        add_to_leaf(toAdd->values, toAdd->positions, ll, n, p);
                        toAdd->len = ll+1;
                }
                else{
                        branch_leaf(n, p, toAdd, *r);
                }

        }
```

```
}
```

2.2.2 Search

Range search in C tree involved finding the first leafNodes that satisfy the greaterThan and lessThan criteria, and the walking from the greaterThan node till lessThan node filtering leafNode at each step.

Search steps:
Step 1 : search the tree to find the leaf node that is greaterThan a value
Step 2: search the tree to find the leaf node that is lessThan a value
Step 3: Walk from leafNode in step 1 , until the leaf node in step 2 , following the pointers in LeafNode and collect the qualifying values.

```
// search function
int* btree_search(int less_than, int greater_than_equal, column* col, int* qCount){

   printf("searching bTree on colName :%s \n",  col->name);
          int* res_arr = NULL;
          int count = 0;

          column_index* ci = col->index;
          root* bTree = (root*)ci->index;
   if(bTree == NULL){
      printf("searching bTree which is :%p, on colName :%s exiting ..\n", (void*)bTree, col->name);
      exit(-1);
   }
 if((less_than != -1) && (greater_than_equal != -1)){
                      leaf_node* start_node = search_node(greater_than_equal, bTree, true);
                      leaf_node* end_node = search_node(less_than, bTree, true);
                      struct leaf_node* current_node = start_node;

                      while((current_node != NULL) && (current_node != end_node)){
                                count = filter_data_from_node(current_node,less_than, greater_than_equal,
&res_arr, count);
                       current_node = current_node->next;
                      }

                      if(current_node == end_node){
                        count =  filter_data_from_node(current_node,less_than, greater_than_equal,
          &res_arr, count);
       }
          }

          else{
                      exit(-1);
          }

          *qCount = count;

   return res_arr;
}

 // filter and collect function
```

```
int filter_data_from_node(leaf_node* current_node,int less_than, int greater_than_equal,
   int** res_arr, int count)
{
   int c = count;
   int nl = current_node->len;
   int* values    = current_node->values;
   int* positions = current_node->positions;
   int* r_arr = *res_arr;
   int incr_by = INDEX_PAGE_SIZE;
   int r=0;
   for(int i=0; i<nl; i=i+incr_by){
      if((i+incr_by) > nl){
         incr_by = (nl - i);
      }
      for(r=i; r<i+incr_by; r++){
         int curr_val = values[r];
         // printf("curr_val is %i, greater_than_equal is %i and less_than %i\n",
         //     curr_val, greater_than_equal, less_than);
         if((curr_val >= greater_than_equal) && (curr_val < less_than)){
           // printf("found a match %i , c is :%i\n", curr_val, c);
            r_arr = realloc(r_arr, sizeof(int)*(c+1));
            r_arr[c] = positions[r];
            c++;
         }
      }
   }
   *res_arr = r_arr;
   return c;

}
```

3. EVALUATION

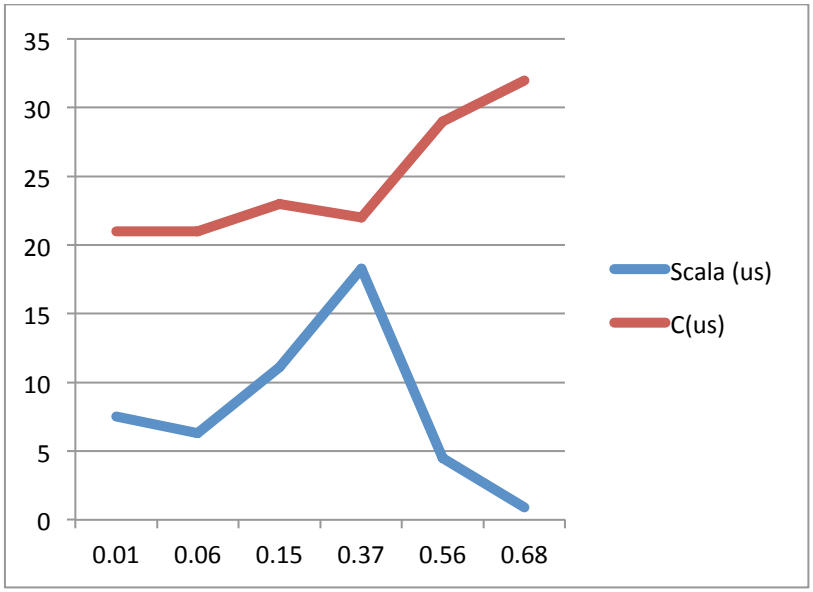
The experimental platform consists x86_64 machine with 4 cores running Intel(R) Xeon(R) CPU E5-1607 model, 32GB of memory. The operating system is Ubuntu 12.04.2 LT. The scala version used is 2.10.3 and the sbt version is 0.13.1. The JVMs were configured to run with 12G of heap and Concurrent MarkSweep garbage collection strategy. For compiling C gcc standard c99 compiler was used. The Fanout of the Tree was kept at 65 for all experiments.

In order to study the performance characteristics of the Tree , varying data sizes (100 to 100k integers) were loaded and queries were run for different selectivity. The graphs (a) through (d) below show the comparison of range queries in C vs Scala tree for different selectivity and different data loads. Graph (e) shows the size comparison of the two trees and graph (f) compares tree build times and table (g) presents the details of cache-misses and instructions for scala Tree (only). There is trend in the query graphs which show that scala querying is consistently better than C. But the size of the C tree is 10 time smaller than the scala tree and the tree build time for C stays pretty much constant, while for scala the increase is almost proportional to the increase in data size.

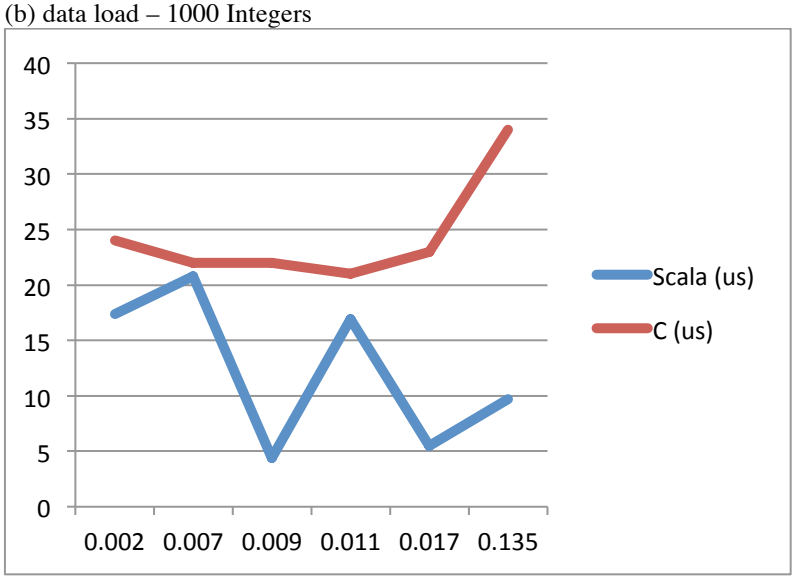
Studying the number of cache-misses, instructions for building the tree in scala – we can see that the number of instructions executed is really large for scala tree . This can be directly attributed to the way the scala tree is built – where each element added results in the whole tree being copied , so as the number of elements added increases so does the amount of copying and hence the garbage. This also triggers large GC cycles attributing to

overall slow down the whole build process. In order to speed up the build times we need to reduce the amount of work down – which will reduce the number of instructions and bring down garbage collection cycles. One way to do this would be to "Bulk Load" maintaing an immutable tree , or change the tree to be mutable just like C.

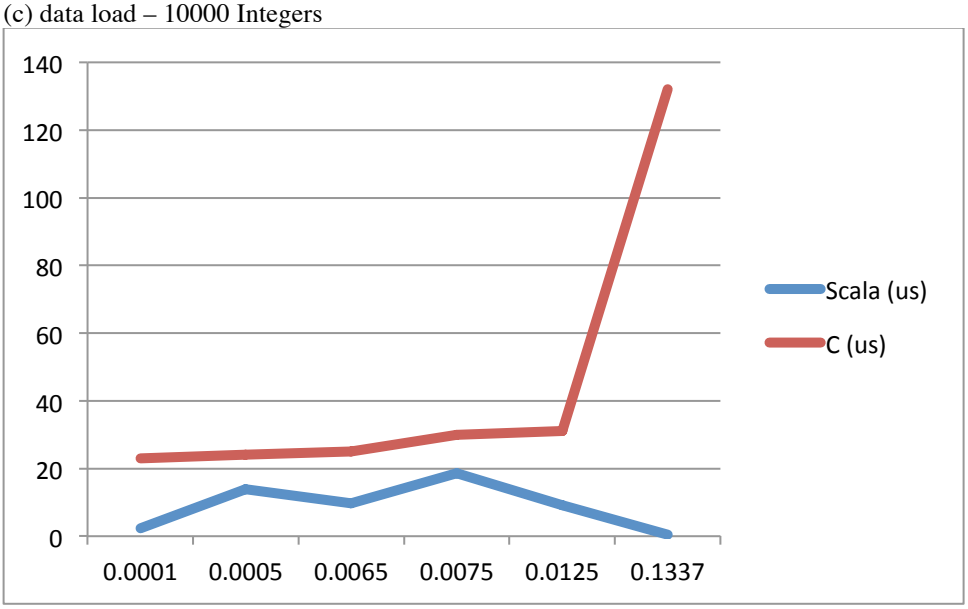
(a) data load – 100 Integers



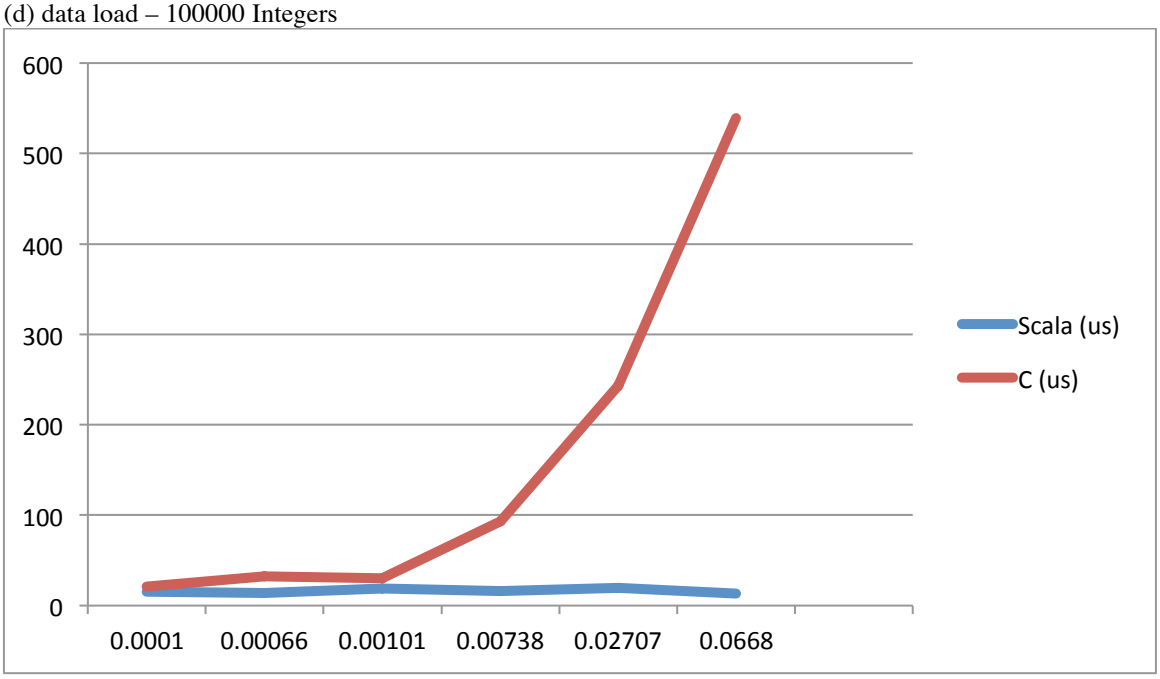
(b) data load – 1000 Integers



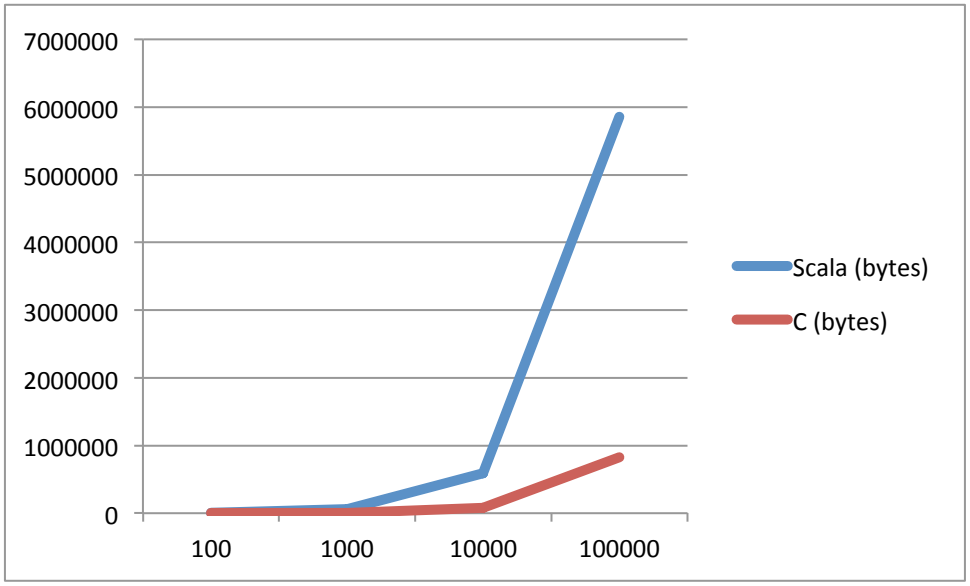
(c) data load – 10000 Integers



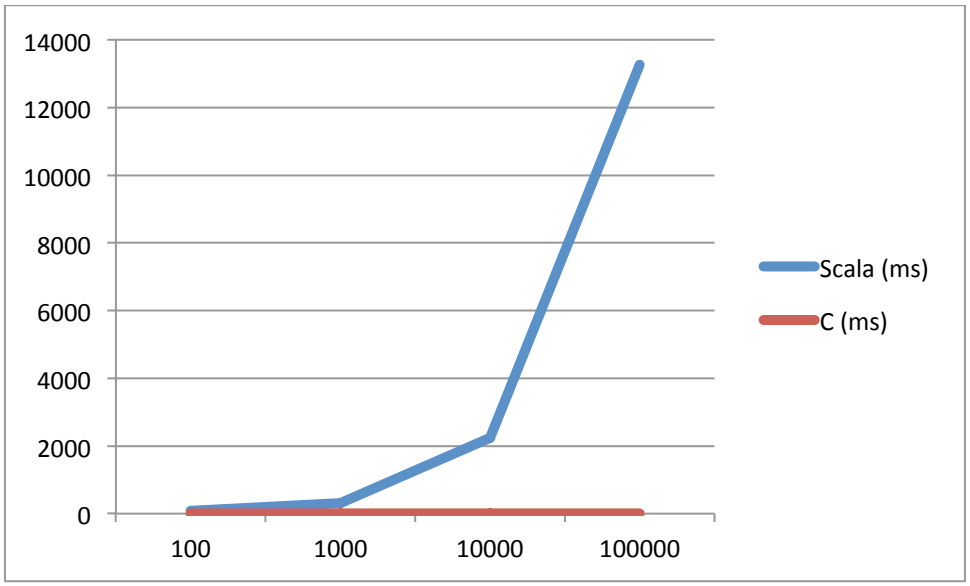
(d) data load – 100000 Integers



(e) size comparision for varying loads.



(f) tree build times for varying loads



(g) scala cache-misses , instructions and other details with increasing load.

Size	cache-misses	instructions	branch-misses	cycles
100	190,601,206	10,469,243,786	17,305,644	13,922,795,976
1000	191,140,952	12,438,019,088	44,806,872	16,209,715,057
10000	204,974,081	27,043,487,866	101,786,401	28,128,126,009
100000	1,475,611,523	1,561,597,988,157	518,463,606	885,821,817,558

4. CONCLUSION

In this paper we have studied the characteristics of C and Scala Tree and compared their memory profile and range query performances. We can see that while the memory footprint of scala tree is really large when compared with C, the range queries in scala almost always outperform C.

The current implementation of scala Tree is immutable, which lends itself to lot of copying operations (and hence garbage) triggering large GC cycles. One thing that would be interesting to study would be the performance of mutable Scala Tree, built using a TreeBuffer (very similar to the ListBuffer found in standard scala API). It will also be interesting to compare the performance of a C , Scala and Java implementations. Since Java allows for unboxed types (int, double). Given the strong query performance of range queries in scala , another interesting area of study is the performance of scans in C and Scala.

We need to continue to invest in these new and powerful programming languages. If applications written in these languages can be made to perform like C, then application development can be much faster and less error prone.

5. REFERNCES

1. wikipedia - https://en.wikipedia.org/wiki/Programming_language
2. scala api - http://www.scala-lang.org/api/2.10.3/#package
3. Christoph Koch and others - Building Efficient Query Engines in a High-Level Language
4. Purley function data structure - https://www.cs.cmu.edu/~rwh/theses/okasaki.pdf