# RRB-Trees: Efficient Immutable Vectors

Phil Bagwell Tiark Rompf

EPFL, Lausanne, Switzerland
{first.last}@epfl.ch

September, 2011

EPFL-REPORT-169879

**Abstract.** Immutable vectors are a convenient data structure for functional programming and part of the standard library of modern languages like Clojure and Scala. The common implementation is based on wide trees with a fixed number of children per node, which allows fast indexed lookup and update operations. In this paper we extend the vector data type with a new underlying data structure, Relaxed Radix Balanced Trees (RRB-Trees), and show how this structure allows immutable vector concatenation, insert-at and splits in $O(logN)$ time while maintaining the index, update and iteration speeds of the original vector data structure.

## 1 Introduction

Immutable data structures are a convenient way to manage some of the problems of concurrent processing in a multi-core environment. Building an efficient immutable analogue to the ubiquitous mutable array, i.e. an indexable ordered sequence of values, is not an easy task as a naive immutable version will have an unacceptable linear cost for updating individual elements. The immutable vector data structure as pioneered by the programming language Clojure [Hickey, 2006] strikes a good balance between read and write performance and supports many commonly used programming patterns in an efficient manner. In Clojure, immutable vectors are an essential part of the language implementation design. Ideal Hash Tries (HAMTs) [Bagwell, 2001] were used as a basis for immutable hash maps and the same structure, 32-way branching trees, was used for immutable vectors. The resultant design provides efficient iteration and extension in constant time, indexing in $log_{32}N = \frac{1}{5}lgN$ time and updates in $\frac{32}{5}lgN$ time. Using 32-wide arrays as tree nodes makes the data structure cache friendly. An indexed update incurs only $\frac{1}{5}lgN$ indirect memory accesses, meaning that, for practical purposes, programmers can consider all the operations as "effectively constant time". However, vector concatenation, splits and inserts at a given index are not easily supported by the structure. Our aim with this work is to extend the underlying vector structure to support concatenation and splits in $O(logN)$ rather than linear time without compromising the performance of the existing operations.

Such a data structure would lend itself to parallelization for common types of comprehensions, for example *filter*. A vector can be split into partitions that then can be evaluated in parallel. In the case of filter, the resulting sub-vectors can then be concatenated to return a result vector without linear copying. In this way the benefits of parallel processing are not lost in assembling the results.

Although the present work was targeted at Scala the data structure is applicable in other language environments such as Clojure, C, C++ and so on. A specialised implementation could lead to string data structure that is well matched to the demands of web page generation.

In the remainder of this paper the term vector will be used to refer to the 32-way branching data structure found in Scala and Clojure.

### 1.1 Previous Work

Previous work has led to immutable data structures that offer an improved solution to the problem of concatenation, notably Ropes [Boehm et al., 1995], 2-3 finger trees [Hinze and Paterson, 2006], and B-Trees [Bayer and McCreight, 1972]. However, each has limitations. With Ropes, a data structure originally created to support the concatenation of strings, the aim is achieved by simply creating a binary tree that has the two sub-strings as branches. With the addition of the two string sizes to the node, efficient indexing can be performed after concatenation. Splits can be performed by creating a split node above the Rope with the values of the upper and lower split bounds. However the performance degrades as repeated concatenations and splits are performed. The index times become $s + lg\ c$ where $c$ is the number of concatenations and $s$ is the number of splits along a Rope tree path. Balancing is required to preserve worst case performance. Without copying, splits will lead to memory leakage as the excluded parts of the original string are not collectible when no longer referenced.

2-3 finger trees achieve a $lgN$ time for indexing and update while at the same time maintaining an amortized constant time for adding items to the vector. Concatenation can be accomplished in $lgN$ time too. Although attractive, using this data structure for vectors does compromise the $\frac{1}{5}lgN$ time for index, making it theoretically 5 times slower. Data structures discussed by Okasaki [1999] also differ in constant factors.

Here we introduce a new data-structure RRB-Trees, Relaxed Radix Balanced Trees, that extends the vector structure whilst keeping its basic performance characteristics and allowing efficient concatenation, splits and inserts.

### 1.2 Vectors

For illustration purposes all the structures will be presented using 4-way branching tree examples. Except where specifically stated the principles can be applied to any m-way tree structure, including the 32-way one of interest for immutable vectors. In figure 1 a basic vector structure is illustrated using this convention. A 32-way version can be imagined by mentally replacing each array of 4 branches with one of 32.

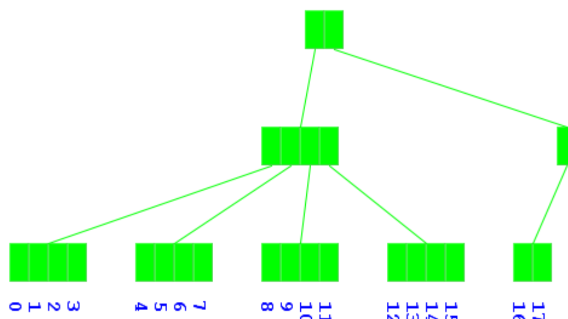


**Fig. 1.** Basic Vector Structure

In developing immutable hash maps for Clojure, lead developer Rich Hickey used Hash Array Mapped Tries (HAMT) [Bagwell, 2001] as their basis. These use a 32-way branching structure to implement a mutable hash map. The immutable version, pioneered by Clojure, was created understanding that only the tree path needed to be rewritten as items were added or removed from the map. The same 32 way branching structure was then used to provide an immutable vector. In this case the 'key' to an item is its index in the vector rather than a hash of the key. Again immutability was achieved by copying and updating only the tree path for updates and additions, with the 32-way branching leading to $\frac{1}{5}lgN$ index performance.

The choice of 32 for $m$ in the m-way branching of vectors follows from a trade-off between the different use cases for the structure. Increasing the branch factor improves index and iteration performance while tending to slow down updating and extension. As $m$ increases the indexing cost are in principle proportional to $log_m N$ while the update costs are proportional to $m\ log_m N$. However in practice the memory cache line, 64-128 bytes in modern processors, makes the cost of copying small blocks of this size relatively cheap. As can be seen from figure 2, $m = 32$ represents a good balance between index and update costs, particularly as common use cases tend to use indexing and iteration far more than updates.

Choosing $m$ to be a power of two enables shifts to be used to extract the branching value from the index rather than the slightly more expensive modulus. Although an important consideration in the past, modern computer architectures make this a marginal advantage today.

The graph, figure 2, demonstrates the advantage of using the m-way structure over that of a binary or 2-3 finger tree. Index times are a little over four times faster using a 32-way data-structure while update times are similar. The theoretical five time advantage is diluted by the caching of the upper tree nodes.

**Fig. 2.** Time for Index and Update (nano-seconds)

### 1.3 Concatenation

Two vectors are shown in figure 3. The naive approach to concatenate these into one vector requires a "shift left" and copying of all the nodes from the right hand vector into the appropriate positions in the concatenated vector, a process linear in the size of the right hand vector. Alternatively one can traverse the right hand vector and add the items to the left vector, again a linear process. It will be shown that the proposed RRB-Tree structure allows efficient concatenation.

## 2 RRB-Trees

RRB-Trees extend the given vector structure by relaxing the fixed branching factor $m$. In doing so it is crucial to prevent the tree representation from degenerating into a linear linked list and to maintain $lgN$ tree height.

### 2.1 Balanced Trees

One class of balanced tree structures maintain a relation between the maximum and minimum branching factor $m_m$ and $m_l$ at each level. These give corresponding maximum height $h_m$ and least height $h_l$ needed to represent a given number of items $N$.

Then $h_l = log_{m_m} N$ and $h_m = log_{m_l} N$
or as $h_l = \frac{1}{lg m_m} lgN$ and $h_m = \frac{1}{lg\ m_l} lgN$

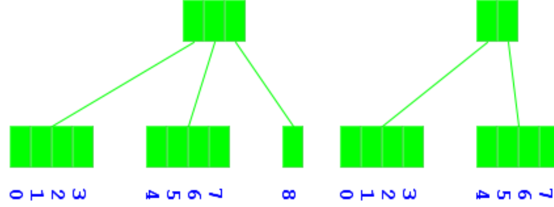


**Fig. 3.** Two Vectors to Concatenate

Trees that are better balanced will have a height ratio, $h_r$, that is closer to 1, perfect balance.

$$h_r = \frac{lg\ m_l}{lg\ m_m}$$

The closer $m_l$ approaches $m_m$ the more perfectly the tree is balanced. For a B-Tree $m_l = \frac{1}{2}m_m$ so

$$h_r = \frac{lg\frac{1}{2}m_m}{lg\ m_m}$$

$$h_r = \frac{lg\ m_m - 1}{lg\ m_m}$$

$$h_r = (1 - \frac{1}{lg\ m_m})$$

As $m_m$ gets large B-Trees approach perfect balance, the well know characteristic.

### 2.2 Relaxed Radix Search

In the case of vectors the branch factor $m$ is always 32 and the trees are perfectly balanced. The constant $m$ allows the radix search to find the child node directly during indexing. When two such trees are concatenated this constraint must be relaxed in order to avoid the linear cost of copying. Less than $m$ items or sub-trees may be stored in a node. However doing this implies that radix search can no longer be used in a simple manner.

The radix search relies on the fact that at a given level there are expected to be exactly $m^{h-1}$ items in each sub-tree thus the index $i$ will be found in sub-tree $\lfloor i/(m^{h-1}) \rfloor$ of that node. The convention is that the index and node offsets are zero based. If there are less than the expected number of items then another method must be used, which we will call relaxed radix search.

In B-Trees the keys are found by storing the key range in the parent nodes and performing a linear or binary search at the parent node to find the sub-tree containing the required key. RRB-trees are similar to B-Trees in this respect, however the index ranges, rather than keys, are stored at the parent node in an array, but only at those nodes that contain sub-trees with nodes that are not $m$ slots wide.

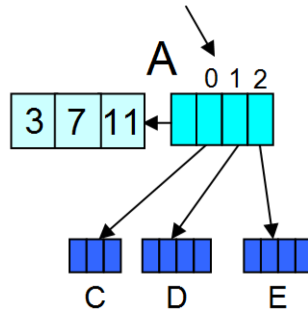


**Fig. 4.** RRB Tree

Figure 4 illustrates the basic structure of an RRB Tree. The tree node A comprises an array of pointers that point to sub-trees, C, D and E. Associated with this array is the range array that contains the accumulated total number of leaf items in these sub-trees. For convenience the pointer and its associated range value will be referred a slot. In the example slot 0 points to node C and is said to contain 3 sub-trees, in this case leaf items.

Suppose the item at index position 3, namely the first item in node D, is to be retrieved. An integer divide of the index by 4 would select slot 0. However, the index is found to be equal or greater than the range value stored at that slot, 3, so the next slot, 1 must be tried. Here the index is less than the range so the indexing continues in slot 1's sub-tree node D. Before doing so, the previous slot, 0, range, 3, is subtracted from the index to give a new zero base index of 0 for the continued search. The desired item is found at zeroth position of node D.

In general then, if $m_l$ is close to $m$ then a radix search at the parent node will fall close to the desired slot. For example if $m_l = m - 1$ then, worst case for a tree of height 2, it will only contain $(m-1)^2$ items. Indexing to the $m^{th}$ slot the sub-tree would be expected to be found in the chosen slot, however some of the time the next slot must be tested.

While indexing, before following a sub-tree, the sub-tree range value must be compared to ascertain which sub-tree to follow (no backtracking is necessary). The range values are in fact the count of actual items in the leaves in and to

the left of that slot. Two possible range values may be checked rather than just indexing to the correct path directly. This extra test is relatively cheap to make on modern machines as reading the first value will cause a cache line to be loaded and the next few adjacent values are brought into the cache at the same time as the first one. Carrying out a short linear search has a very small overhead. Furthermore, if all possible indexes are considered and the nodes are randomly $m$ or $m-1$, then we would expect the radix search to succeed $\frac{3}{4}$ of the time.

The average number of items in a slot is $m - \frac{1}{2}$. Starting with the first slot, the probability that the item will not be found in the slot is $\frac{0.5}{m}$. For the second slot it will be $\frac{1.0}{m}$, the third $\frac{1.5}{m}$ and so on to the $m^{th}$ $\frac{0.5m}{m}$. Summing the series the average probability of a miss is 0.25.

### 2.3 Cache Line Awareness and Binary Search

Understanding that cache line loads give this benefit to radix searches with a short linear search, it may be expected that a binary or even a simple linear search at a node could be attractive. A binary search would be desirable as it requires fewer constraints in an eventual concatenation algorithm. However, a binary search with $m = 32$ may cause several cache misses with the attendant cache line loads and cache prefetching cannot be easily employed. Empirical testing shows that the relaxed radix search gives an overall indexing speed that is nearly three times faster than a binary or purely linear search at the node (see benchmarks).

### 2.4 Concatenation

Figure 5 illustrates the concatenation of two RRB-Trees. For simplicity the case is considered where $m = 4$, however the same principles can be applied for greater values. The process starts at the bottom of the right edge of the left hand tree and bottom of the left edge of the right hand.

B-Trees ensure balancing by constraining the node branching sizes to range between $m$ and $2m$. However, as mentioned earlier B-Trees nodes do not facilitate radix searching. Instead the initial invariant chosen was that of allowing the node sizes to range between $m$ and $m-1$. This defines a family of balanced trees starting with well known 2-3 trees, 3-4 trees and (for m=32) 31-32 trees. With this invariant balancing is ensured and radix branch search achieved in the majority of cases. Occasionally a few step linear search is needed after the radix search to find the correct branch.

The extra steps required increase at the higher levels. The least number of leaf items in the tree is given by $(m-1)^h$. The maximum number of leaf items is $m^h$, The worst case number of extra steps at the top is given by the maximum less the minimum divided by the slot size at that level.

$$\frac{m^h - (m-1)^h}{m^{(h-1)}}$$

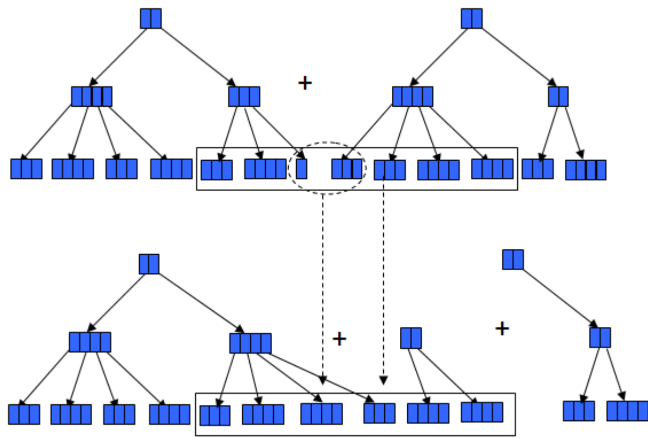

**Fig. 5.** Vector Concatenation, 3-4 tree bottom level

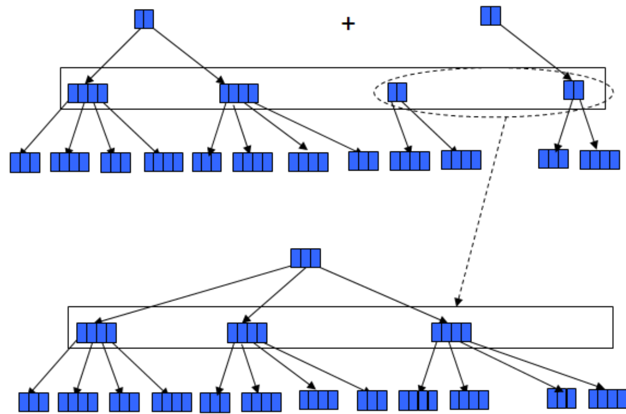

**Fig. 6.** Vector Concatenation, 3-4 tree first level

or 4.69 where $m = 32$ and $h = 5$. Assuming a random distribution for node sizes between $m - 1$ and $m$ then the expected worst case would be 2.49 on average.

In figure 5 the concatenation/balancing algorithm is illustrated. Although the principle is demonstrated with a 3-4 tree it can be applied to any of the family including the case when m=32. First move down the right hand edge of the left tree and the left hand edge of the right tree. Move up one level. In order to have a well formed tree the nodes enclosed in the box must be rebalanced to conform to the invariant.

This requires that branch entries or items must be copied from the right to the left until each slot count lies between $m - 1$ and $m$. Further the sequential order must be maintained. In the example, the item in the 3rd node and the three items in the 4th node need to be copied into a new node that now meets the invariant. However, in general several or even all the nodes may need to be copied before they meet the invariant.

In the worst case $m^2$ nodes must be copied at each balancing. It is always possible to redistribute a slot with 1 to $m$ entries amongst $m$ slots to meet the invariant. The worst case cost of concatenation is therefore proportional to $\frac{m^2}{lgm} lgN$ or $O(logN)$. This is a constant factor times the cost of completing a simple immutable update to a value in a vector.

The first four of the resulting nodes form a new right hand node to the left tree while the remaining two nodes are carried up to the next level to be included in the balancing at that level, as shown in figure 6. Here the right hand two nodes are combined to form a new 4 way node and the new next level up node is created to complete the concatenation. In general, this process is repeated until the top of the tree is reached.

Note that only these modified tree edge nodes are rewritten. The rest of the tree is untouched and concatenation results in a new immutable vector.

### 2.5 Summing over two levels

Further inspection suggested a less constrained approach that achieves the desired relaxed radix search directly and reduces the concatenation cost by a factor of around three on average.

Considering all the slots at a node, if $a$ slots contain a total of $p$ sub-tree branches then the maximum extra linear search steps $e$ is given by $e = a - (\lfloor \frac{p-1}{m} \rfloor + 1)$. Only $\lfloor \frac{p-1}{m} \rfloor + 1$ slots, each slot containing exactly $m$ items, are required for perfect radix searching while there are actually $a$. Now the balancing step is carried out just as before but individual nodes can have any number of sub-trees as long as $e$, the extras search steps is not exceeded.

An example of balancing using this constraint, with $e = 1$, is shown in figure 7. At the bottom here are 6 nodes to be considered for balancing and the total number of items is 16. in this case $e = 6 - \lfloor \frac{16-1}{4} \rfloor - 1$ or 2, one more than the allowed radix search error. Balancing is required. Starting from the left any slots with $m$ entries are skipped. If all nodes are of this size the invariant is met automatically and the typical use case is concatenating standard vectors with

all nodes of size $m$ except the right hand one. Then from the first small one the process continues to the right until the total of all slots from the small one are enough to reduce the slot count by one (the ones enclosed by the ellipse). Moving along, slots are copied to reduce the number of nodes.

Since the node at this level is going to be copied anyway the opportunity is taken to copy in enough slots to make the total up to $m$ or 4 in this case. Since the total possible number of slots is $2m$, $m$ from the left and $m$ from the right, this ensures no more than $m$ are left on the right to carry up and the node will typically be skipped at the next level in any re-balancing so there is no extra work. Now we carry the remainder up to the next level and repeat the exercise.

In figure 8, the first level node on the right hand side of the left tree only has two slots. Here there are 4 nodes to consider with a total of 11 sub-nodes. In this case $e = 4 - \lfloor \frac{11-1}{4} \rfloor - 1$ or 1. No balancing is required so the top 4 slot node can be constructed and the concatenation is complete.

In general it can be shown that skipping over slots with $m - \frac{e}{2}$ or greater entries is sound. Once the smallest one has been reached there will always be enough nodes to copy in to the right to reduce to the correct value of $e$.

Suppose the small slot is the $n^{th}$ one and that the number of slots to give direct radix searching is $r$ calculated from the total number of sub-nodes as described above. Given this then $r + e$ gives the final Each of the nodes to the left of the $n^{th}$ will be between $m - \frac{e}{2}$ and $m$ and thus skipped. There can be a maximum of $2m$ slots to be balanced. Hence even if the small slot is the $2m^{th}$ one then the minimum total number of sub nodes/items to its left will be $2m(m - \frac{e}{2})$ or $2m^2 - em$. But there are $2m - e$ slots needed to give direct radix searching therefore the total number of sub nodes/items must be $m(2m - e)$ or $2m^2 - em$. The same as is needed to allow the small slot at the $2m^{th}$ position.

In practice to reduce the copying work all the new node sizes are first computed and the actual copying is done just once.

So far it has been assumed that the maximum error $e$ is 1 but it can be made larger. A larger allowed radix search error causes the time to index a value to increase while the concatenation cost reduces. With $m = 32$ empirical testing shows that setting $e = 2$ gives a good compromise. Small impact on the index time while retaining low cost to concatenate. The resulting tree will be very close to perfect balance as on average $m_l + 2 = m = 32$. The slight increase in depth contributes to the increased index times.

With the current balancing strategy the worst case situation can arrive that the smallest slot is the left most one and all the ones to the right of it are $m$ except the last two. In this case balancing will cause shuffling of all 2m slots giving a worst case copy cost of $2m(m - e)$, still constant. The process of shuffling described is a heuristic that has been found to provide the performance characteristics desired. The worst case just described could easily be avoided by first checking which small slot is cheapest to start with. Alternative algorithms, for example based on dynamic programming that try to achieve a globally optimal shuffling could be used as well, however the chosen algorithm performs well in practice.

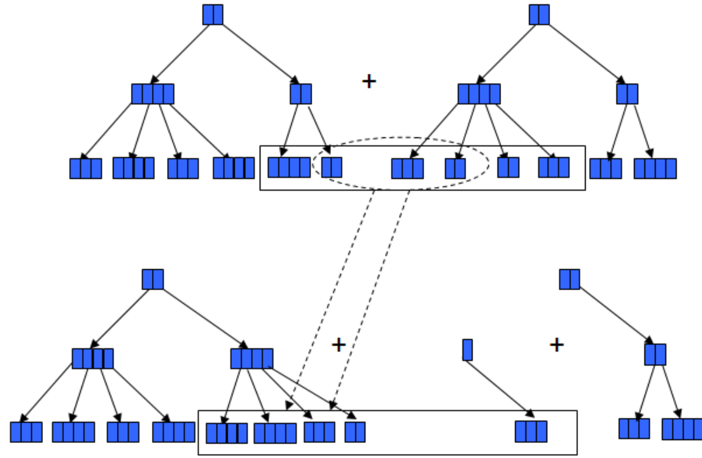


**Fig. 7.** Sum 2 Levels, 3-4 tree bottom level

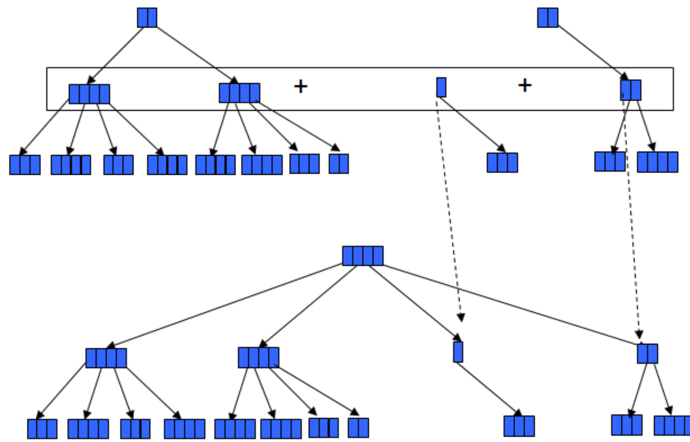


**Fig. 8.** Sum 2 Levels, 3-4 tree first level

## 3 Benchmark Results

Performance evaluation of RRB-Trees is performed in several empirical setups. The two most relevant attributes are the impact of concatenation on indexing performance and the time required to concatenate RRB-Trees.

### 3.1 Index Performance Benchmark

Table 1 tabulates the benchmarks for the per index cost of indexing through vectors of different sizes and having undergone varying amounts of concatenation. A random size is chosen for the two test vectors to be concatenated. Then each of the vectors is created by either creating a regular vector or dividing the size randomly into two sub-sizes and repeating the size partitioning. Whether the size is re-partitioned or not is chosen randomly based on probability $p$ set for each benchmark. Results are for $p = 1,\ 0.25,\ 0.17,\ 0.125$ and 0. If the probability $p$ is chosen then there will be a probability $p$ that no further subdividing of size will be done and a regular vector will be created. Once two sub-vectors have been created they are concatenated. This concatenation process continues at the next level up until a vector is finally constructed of the initial size required. In this way vectors are created that are regular, $p = 1$ or the result of many small random vectors concatenated, $p = 0$ or somewhere in between. A large number of trials was made and the results averaged in each vector size range.

The summary performance factor gives a guide to the cost of using RRB-Tree structures using regular vectors as the basis of comparison.

It is worth mentioning that updates and iterations have negligible speed penalties. In both cases the range information does not need to be updated and iterations only follow the branch references, behaving in the same way as a regular vector.

The last column reports the index times for RRB-Trees containing almost no regular vector parts using a binary search at the node branching instead of the relaxed radix search. These index times are over five times longer than the regular vector while nearly 3 times longer than using radix search.

### 3.2 Concatenation Costs

Test vectors were created as for the Index tests and the total number of memory locations copied during the final concatenation recorded. This includes the tree nodes, leaf items, range arrays and temporary size arrays created during the concatenating and balancing process. The results can be seen in Table 2. These can be compared to the cost to complete a simple value update, $\frac{32}{5} lg\ N$ or 160 for a 5 level tree.

### 3.3 Observation

Notice that as the size passes through boundaries 10, 15 and 20 index speed and concatenation costs reflect the extra tree level added.

**Table 1.** Comparison of Index Times nano-secs

| n | $2^n$ | RegV $p = 1$ | RRB $p = 0.25$ | RRB $p = 0.17$ | RRB $p = 0.12$ | RRB $p = 0$ | Bin Srch $p = 0$ |
|---|---|---|---|---|---|---|---|
| 10 | 1024 | 23 | 45 | 46 | 45 | 46 | 93 |
| 11 | 2048 | 28 | 45 | 43 | 46 | 47 | 98 |
| 12 | 4096 | 28 | 44 | 82 | 47 | 47 | 116 |
| 13 | 8192 | 41 | 39 | 45 | 50 | 48 | 118 |
| 14 | 16384 | 27 | 50 | 46 | 46 | 66 | 130 |
| 15 | 32768 | 27 | 53 | 56 | 57 | 64 | 155 |
| 16 | 65536 | 31 | 53 | 57 | 62 | 66 | 161 |
| 17 | 131072 | 35 | 53 | 59 | 58 | 66 | 175 |
| 18 | 262144 | 33 | 58 | 60 | 61 | 66 | 189 |
| 19 | 524288 | 34 | 66 | 59 | 61 | 67 | 200 |
| 20 | 1048576 | 34 | 63 | 64 | 71 | 80 | 225 |
| 21 | 2097152 | 38 | 63 | 62 | 74 | 82 | 230 |
| 22 | 4194304 | 38 | 61 | 69 | 69 | 82 | 243 |
| 23 | 8388608 | 37 | 59 | 54 | 66 | 82 | 258 |
| | Average | 32 | 54 | 57 | 58 | 65 | 171 |
| | **Factor** | **1.00** | **1.66** | **1.77** | **1.79** | **2.00** | **5.27** |

**Table 2.** Copy costs of Concatenation RRB-Tree

| n | $2^n$ | RegV $p = 1$ | RRB $p = 0.25$ | RRB $p = 0.17$ | RRB $p = 0.12$ | RRB $p = 0$ |
|---|---|---|---|---|---|---|
| 10 | 1024 | 76 | 272 | 273 | 307 | 307 |
| 11 | 2048 | 152 | 484 | 534 | 531 | 572 |
| 12 | 4096 | 160 | 225 | 260 | 220 | 240 |
| 13 | 8192 | 173 | 319 | 318 | 393 | 489 |
| 14 | 16384 | 194 | 508 | 556 | 547 | 757 |
| 15 | 32768 | 226 | 786 | 871 | 808 | 1009 |
| 16 | 65536 | 315 | 1164 | 1149 | 1145 | 1304 |
| 17 | 131072 | 313 | 619 | 567 | 648 | 795 |
| 18 | 262144 | 325 | 570 | 793 | 616 | 887 |
| 19 | 524288 | 326 | 808 | 871 | 1006 | 1191 |
| 20 | 1048576 | 371 | 1114 | 1149 | 1345 | 1410 |
| 21 | 2097152 | 456 | 1417 | 1687 | 1617 | 1674 |
| 22 | 4194304 | 464 | 755 | 923 | 1106 | 1183 |
| 23 | 8388608 | 473 | 871 | 938 | 1100 | 1631 |

[a)] Costs include all element copies - items, sub-tree references and sizes

# 4 Splits and Insert At

The split operation can be more easily understood as being the result of removing a left and right slice from the vector.

A right slice is accomplished by simply passing down the edge defined by the right slice index and making it the right hand edge of the tree. Nodes to the right of the index are simply dropped when the nodes on the index are copied to make the new right edge of the tree.

Similarly a left slice is accomplished by passing down the edge defined by the left slice index and making it the left hand edge. Nodes to the left of the index are dropped and nodes to the right are shifted left when copying to create the left edge of the tree. Trees created in this way may not meet the invariant outlined above. There may be one more than $e$ extra slots for a given level. However, a subsequent concatenations will restore the invariant. By taking this lazy approach redundant balancing is avoided. Splits have the same cost as an update namely $\frac{m}{lg\ m} lgN$.

# 5 Implementation Considerations

The original Scala and Clojure Vector implementation uses internal nodes with an array carrying the 32 way branch. In this implementation the array is increased by one when a node is converted to an RRB Tree node. The zeroth element carries a reference to the array containing range values. There is no overhead on vectors that have not been concatenated. Further by using a separate range array, rather than a range/pointer object the speed of iteration, a common use case, is unaffected.

After concatenation on some of the nodes the zeroth element will point to the array of range values. Since only internal nodes carry this extra element the extra memory overhead is close to one in $m^2$ or 1 in 1024 and can be considered negligible.

A typical vector after concatenation will contain a mix of nodes with standard 32 way sub-trees and RRB-Tree nodes with range values. In the JVM implementation the object type can be used to determine whether to use the optimised standard vector index method or the slower RRB tree index method.

There is no speed loss or memory overhead on the standard vectors when the concatenation and slice capability is unused.

## 5.1 Constant Time Addition

The Clojure vector implementation includes an optimization to allow constant time append of single elements. The last 32-wide block of elements is kept outside the tree structure so that it can be accessed in constant time, without going through multiple tree levels. To append an element (on the right) to a Clojure Vector, only this max. 32-wide array has to be copied.

It is possible to extend this model to multiple levels and to varying positions. In the Scala implementation, vectors have a notion of *focus*, which identifies a single 32-wide block that is accessible in constant time. Every update operation (be it at the left end, the right end, or any other position), will put the target block in focus. Updates to a block 'in focus' will copy only that particular 32-wide block. Moreover, all the tree nodes from the root to the block in focus are kept in a *display*, i.e. a constant-time stack. Moving the focus to an adjacent 32-block incurs only one indirection through the display, possibly copying the node one level up. Indexed reads also use the display to minimize the number of indirections.

When putting a position in focus, the downward pointers are replaced by null entries. As long as the focus remains within that block, only the bottom array needs to be copied. If the focus moves to an adjacent block, display slot 1 needs to be copied, at the old focus position the pointer is put back in and the newly focused one is nulled.

The design decision behind this model is to optimise for spacio-temporal access locality. The assumption is that sequentially rewriting parts of a Vector (starting from an arbitrary position) is a common operation; the same holds for reading elements close to the last update.

## 6 Conclusions

The RRB-Tree based vector seems to provide a viable extension to the existing 32-way immutable vector used in Clojure and Scala to provide $O(logN)$ concatenation and splits while maintaining the basic costs associated with other operations. For practical purposes they can be considered constant time.

The data structure could also be attractive as the basis of a string implementation or for in-memory database structures using any language.

This ability to partition and concatenate vectors is highly desirable when performing typical parallel comprehensions.

Although a very satisfactory heuristic has been discovered for the node shuffling, further research could study alternative algorithms better optimised to particular use cases.

# Bibliography

P. Bagwell. Ideal hash trees. Technical report, EPFL, 2001.

R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta informatica*, 1(3):173–189, 1972.

H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.

R. Hickey. The Clojure programming language, 2006. URL http://clojure.org/.

R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.