How Implementations of the Persistent Data Type HAMT with Varying Node Sizes Compare in Performance When Implemented in Python

Computer Science

Extended Essay

Word count: 3957

gps320

May 2018 Examination Session

Abstract

Researchers recently implemented the HAMT (Hash Array Mapped Trie) in many programming languages and optimized it for performance and memory efficiency, replacing traditional hash tables in many cases. The HAMT has many possible implementation differences that can have significant effects on its performance, such as separation of data types, bitmap compression, and node size. The purpose of this investigation is to explore **how the size of the nodes in the data structure Hash Array Mapped Trie affect the performance of the search and insert functions on it**. To investigate this, I implemented the HAMT data structure in Python and tested it for insert and search times with varying node sizes using the PyPy Python interpreter. I initially hypothesize that as node size increases, speeds for both insert and search increase, but that the greater node size increases memory overhead, making the middle node size the most effective. My test results refute both of these claims. HAMT implementations in Python with a node size of 32 perform best on insert and search tests and memory use decreases as node size increases. I conclude that HAMT implementations that have bitmap compression with a node size of 32 are optimal when written in Python.

Table of Contents

Introduction

Data structures are ways to represent sets of data so that they can be used efficiently. They allow us to group data abstractly, without worrying about the details of the implementation (Miller, 2014). There are many ways to represent data, with different advantages and disadvantages. Programs written in different programming languages use the data structures provided by their language. Optimizations to base data structures, then, are highly desirable, because they increase the efficiency of all applications that use them.

Some data structures are designed to be persistent, meaning that they are immutable and that all previous versions of the data structure are available after modifications if the previous version remains referenced. The data persists, after modifications, often in a tree structure tied together with pointers (Driscoll, 1989). This type of data structure is used in functional programming languages. The benefits of functional programming are numerous, among them are better structure and debugging potential (Hughes, 1989; Okasaki, 1999).

The Hash Array Mapped Trie is a persistent data type that is useful for mapping keys to values efficiently. Phil Bagwell first conceptualizes it with compression in 2000 in his paper "Fast and space efficient trie searches." Recently, research into the data structure HAMT has made it more memory efficient and quick. The CHAMP variant of HAMT (Steindorfer et al., 2015) increases its performance significantly. The increased use of the data structure also makes optimizations desirable, and structural elements such as node size have not fully been explored. Languages such as Javascript, Ruby, Haskell, Clojure, Scala, and OCaml have implementations of the HAMT. In Clojure and Scala, HAMT is the default Hash Map (Schuck, 2015).

One of many ways HAMTs are represented differently is in the size of the nodes. This paper explores **how implementations of persistent data type HAMT with varying node sizes compare in performance when implemented in Python**. I implemented and benchmarked the HAMT data structure in Python for search speeds, insert speeds, and memory usage with a variety of node sizes. This paper will give background knowledge necessary to understand the HAMT data structure, detail the methods used to benchmark it, and draw conclusions from the results.

Background

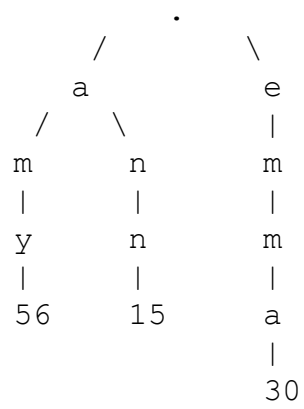
The basic purpose of the data structures mentioned in this paper is to map keys to values. Keys are any kind of data that corresponds to a value, which is another piece of data. Imagine needing to keep track of people's ages and wanting to store the data in some form. The keys would be the names that can be used to access the values, in this case age in years.

Tries

A tree is a branching data structure. It has nodes that contain pointers to sub-nodes that can also be considered trees. There are many kinds of trees, and a commonly used example of a tree is the binary tree. Big-O notation describes the amount of computational time it takes to do something as n increases (Miller, 2014). Using Big-O notation, most operations on binary trees take only $O(\lg(n))$ time to complete in the worst case. Binary trees have many good properties, but a major and obvious drawback of them is the large amount of NULL pointers at the end of the tree (Cormen, 2009, pg. 244).

Tries are a type of tree introduced by Brandais in which each level of the tree defines only part of the key so that checking can be done more efficiently (Brandais, 1959). If the key is a string, for example, then the algorithm checks the a single character at each level of the tree against subnodes. Data is only stored on the final leafs of the tree (Pitts, 2000). Below is a diagram of a trie that takes a string as the key, and an integer as a value. Each letter is a node that is entered if the next character in the name matches it.

Diagram 1- Trie with names as keys and ages as values



Names(keys)	Ages(values)
amy	56
ann	15
emma	30

Because of the structure, when a tree (or trie) is updated, it only requires the reconstruction of one node. The rest of the tree, tied together by pointers, can persist. This means that the new tree returned shares most of its data with the old tree, and the old tree is left intact. The functional way of thinking about this structure eliminates side-effects in an effective way, with comparable performance to mutable data structures.

Another interesting kind of tree is the ternary search tree, which combines elements of the trie and the binary search tree. Each node has three daughter nodes, and checking for parts of the key is done similarly to Binary Search Trees, where values are compared in size and a branch is taken based on this comparison. This model has the drawback of being complex in the

different ways in which it can be arranged based on inputs, and in the need for many nodes (Bentley, 1999).

Hashing

Hashing is using mathematical operations to convert large keys into smaller ones for use in direct indexing of an array. Hashtables are data structures that hold values in a list and use hashing functions to map keys to the values (Guttag, 2013, pg. 137). Python dictionaries are implemented as hash tables, and the data structure HAMT can potentially replace them in Python if a persistent data structure is needed. Dictionaries similarly map keys to values, but internally have a very different structure. The benefit of hash tables is a O(1) time for indexing any value. However, hash tables come with drawbacks, such as slow copying for persistence, their memory overhead, and their need to deal with hash collisions. Hash collisions happen when a key is mapped to the same integer as another existing key. The simple resolution is to use the next available slot in one direction (Cormen, 2009).

Hash Array Mapped Tries

The basic idea behind HAMT is to hash incoming keys inside the data structure, and map those into a trie. While the idea of the hashing values for indexing in a tree already existed (Bentley), Bagwell proposed compressing nodes using a bitmap to save a lot of memory, eliminating many NULL value placeholders. The bitmap stores which indexes in the array would be filled, and then is used with an ordered list of only values. Each bit in the integer represents a filled slot, and the `insert` function does bitwise operations on it to compute the actual index in the compressed array. When inserting a new key value pair, it resizes the list by inserting it in the correct index, and then uses a mask to flip the bit in the bitmap to a one.

With this compression, HAMT has very little memory overhead. Bagwell also details how to write good hashing functions for the HAMT in his paper the following year, titled “Ideal hash trees” (2001). Hashing introduces a new layer of complexity, besides in dealing with collision resolution. For example, the question of whether or not to store the keys in hashed form, in the normal key form, or not at all arises. Further, the way in which key-value pairs and subnodes are stored and checked in each node becomes an interesting problem. Keys are often stored in the array next to the value or in the first half of the array, as is in the Python implementation.

Recently, the HAMT data structure has grown in popularity and become more optimized. In 2015, Steindorfer and Vinju introduce the CHAMP variant mentioned in the introduction in the paper “Optimizing hash-array mapped tries for fast and lean immutable JVM collections”. It discusses ways to store data in the nodes to make them more efficient and use less memory. One of the most important ideas is to separate the sub-nodes and the data into different parts of the array, which requires the use of two bitmaps, one for nodes, and one for data points. They then propose that the two bitmaps be combined into one large 32 bit one. This eliminates the need to check the type of the data when searching and inserting, previously a costly operation, while adding necessary bitwise operations and offset-based indexing. Another proposed way to change the HAMT design using specialized Java classes is demonstrated in the paper “Code Specialization for Memory Efficient Hash Tries” (Steindorfer, 2014).

Pypy

Because Python (as implemented by CPython) is an interpreted language, it is highly optimized for provided data types written in C compared to data types written in Python. The

tests I wrote running in Python show that the dictionaries show that dictionaries perform magnitudes better than HAMTs do. I used a Python implementation called PyPy to make better comparisons between HAMTs and Python dictionaries. PyPy uses a JIT (just-in-time) compiler, and provides much better performance than Python.

Methods

To determine the effect of changing the node sizes in the HAMT data structure, I implemented HAMT in Python 3. I chose Python because there is no published testing of the HAMT data structure in Python, and research using Python would therefore demonstrate concepts behind HAMT in a different environment. This is meaningful because while the purpose of the implementation is to test node size, it also replicates and strengthens previous research. I initially hypothesize that as node size increases, speeds for both insert and search increase due to fewer node transversals, but that the greater node size increases memory overhead, making the middle node size the most effective.

Implementation

First, I implemented an uncompressed version of HAMT in Python 3. It uses the Python list because it requires the nesting of arrays among values, and the Python `array.array` data type can only hold data that is all of the same type (Lenski, 2008). Using small integers directly as keys simplifies the problem. The HAMT implementation uses them directly as hash codes, meaning that there are no hash collisions.

The `insert` function takes a key and a value and inserts the pair into the HAMT. It stores keys and values in nodes, with keys in the first half and values in the second half. Keys

and values are integers, so at each node level the function uses groups of bits to index. For example, for a HAMT with node size 4, the insert function first takes the first two bits and finds the correct index for the key in the first node. If there is a subtree at that index, it enters the subtree and recurs. In the case that there is no subtree, it simply inserts the key and value into that index. When it attempts to insert a value into an index that already holds a value, it creates a new subnode, placing both the old value and the new one in the new node. Below is the insert function used in the uncompressed HAMT implementation. The insert function copies nodes that need to be changed, and returns a completely new HAMT object, thus making the HAMT implementation a persistent one. The Python `list.copy()` method copies the list itself and its necessary pointers, but not data within it, letting the old HAMT and the new HAMT with inserted data share the majority of the tree structure.

The search function follows from the insert function. I reused the code that retrieves the index of the key at each depth with a simple loop that follows the subnodes defined by the key through the tree until it reaches the value, assuming the key and value have already been inserted.

Next, I implemented the HAMT with the method of compressing each node with a bitmap explained in the background section. This memory optimization eliminates Python `None` placeholders and is especially effective for larger node sizes, which introduced too large of a memory overhead in the unoptimized version.

My implementation stores the bitmap in the first index of each node, so all other indexes are offset by one. To get a better understanding of the insert function, see the code snippets in Figure 1 and Figure 2 in the appendix. Figure 1 is the insert function from the HAMT

implementation without compression, and Figure 2 is a snippet from the insert function with compression.

Testing and Benchmarking

The methods used to benchmark insert and search speeds of the HAMT implementations are important, because they must be designed to test how node sizes affect performance alone. They are documented here for repeatability and variability.

Two testing methods are used to check the behavior of the insert function. The first is to create small HAMT structures by hand using a list of keys, and check them against a HAMT with the list of keys inserted by the implementation. The second method is to insert a long list of keys in different orders, and to check that the resulting two HAMTs are consistent. Both of these methods are critical to discovering bugs in the insert method, especially in how it handles allocating new nodes and inserting both the old data and the new data. Search is tested by comparing keys to the values retrieved to make sure they are correct.

Python's `random.randint` function generates random integer keys for testing between 0 and 100,000 inclusive, to minimize repeats. I chose to make values be the corresponding key plus one to make testing convenient. Benchmarking functions I wrote measure the amount of time taken to perform the operation, repeated many times to obtain valid results. The time it takes to insert the data into a HAMT only once is too small to be used in comparisons. The benchmarking functions use the Python `timeit.timeit` function, which by default disables Python's garbage collector while it is timing to measure only computational time (Python documentation). To measure the performance of data structures in PyPy, the benchmarking functions explicitly re-enable Python's garbage collector.

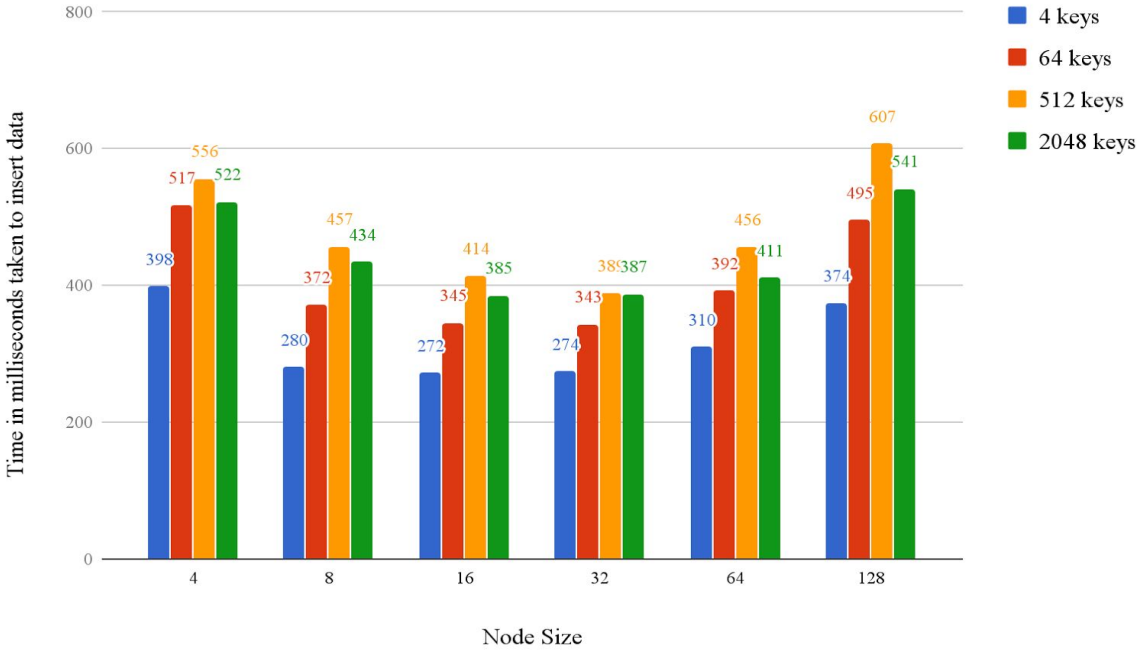
To measure memory usage of the data structures, I used the Python `sys.getsizeof` function. For HAMTs, a function recursively follows the paths of each node to call `getsizeof` on each Python list. Memory use has to be measured running in vanilla Python, because PyPy does not support `getsizeof`.

Results

The benchmarking functions print the results out to the console. Graphs 1 and 2 below (see corresponding tables in the appendix) compare the speeds of insert and search functions on the compressed HAMT implementation with varying node sizes and data set sizes. The data from the compressed HAMT implementation running in PyPy will be used, because it is the best scenario in which to make generalizations about node size. Note that these graphs do not demonstrate the effects of varying amounts of data to the reduced number of iterations in tests with larger amounts of data.

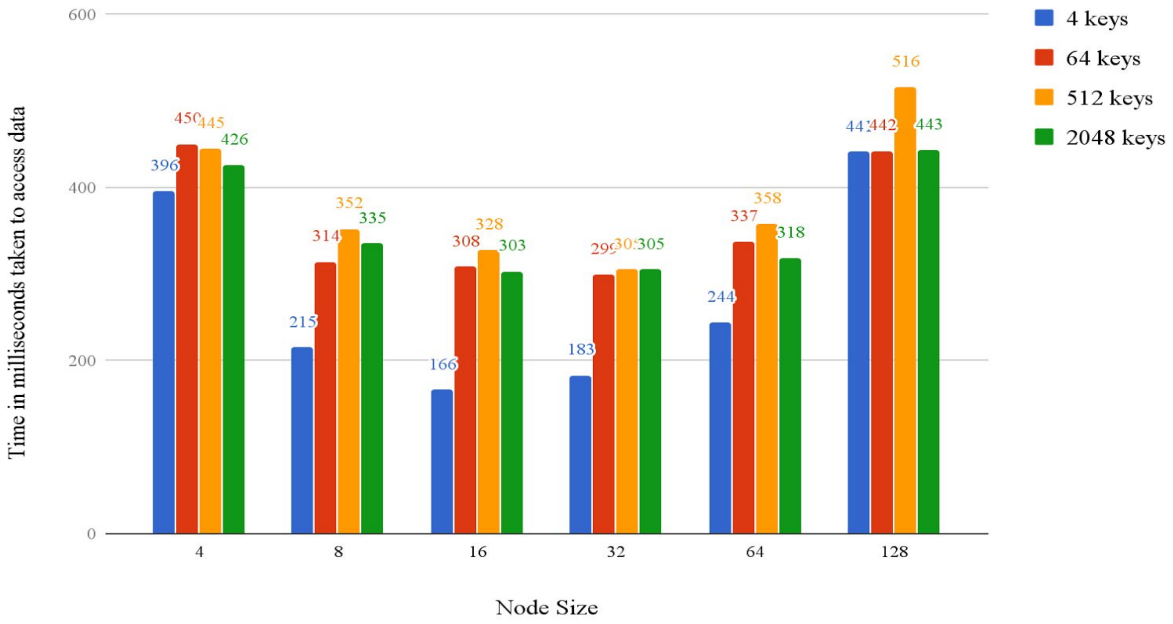
Graph 1

Time in milliseconds taken to insert data into compressed HAMT with varying node sizes in PyPy



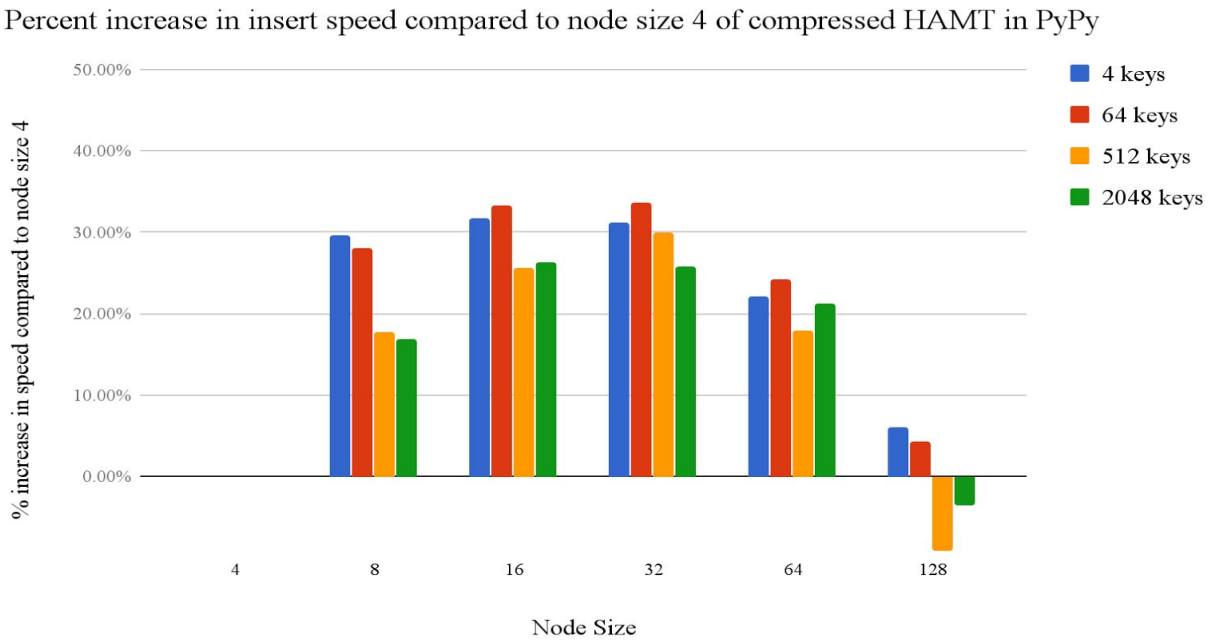
Graph 2

Time in milliseconds taken to access data in compressed HAMT with varying node sizes in PyPy

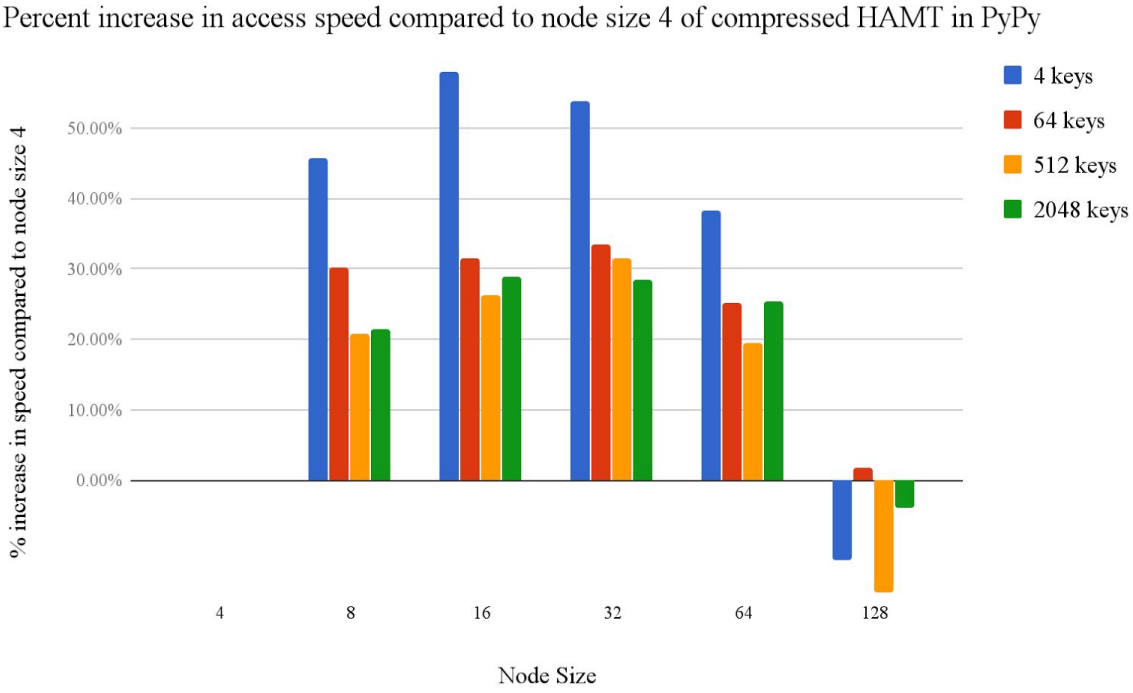


Graphs 3 and 4 below highlight the best node sizes for different sizes of data because they show data relative to the smallest node size.

Graph 3



Graph 4



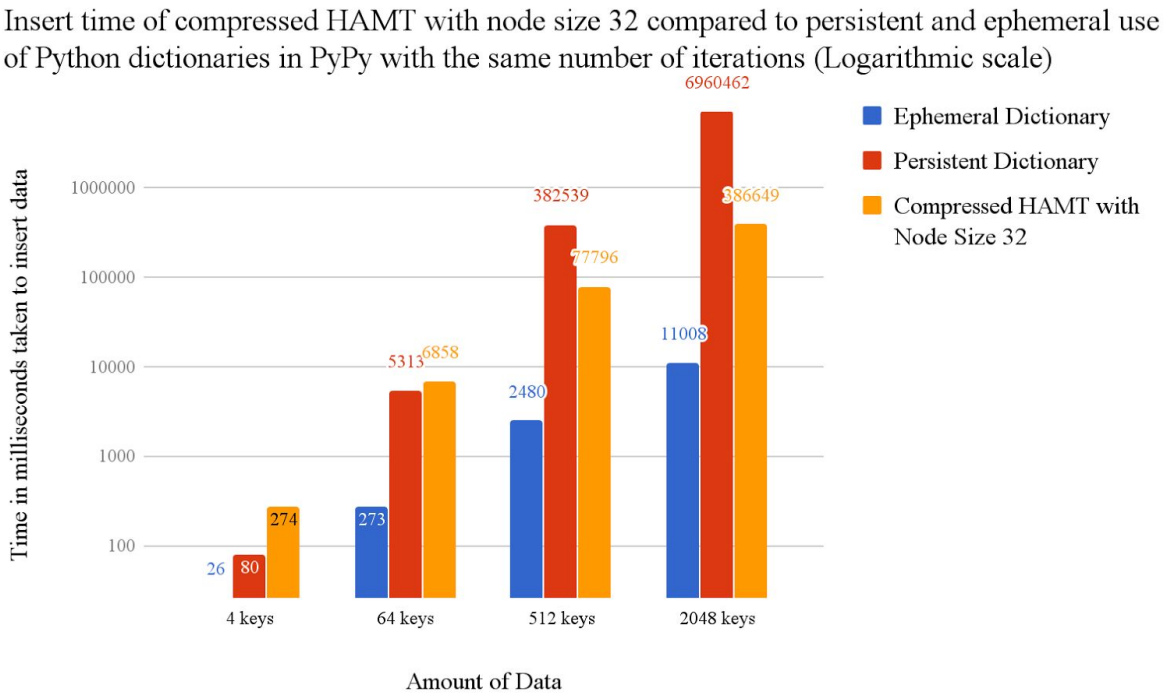
Both Graphs 3 and 4 indicate an optimum node size of 32, although 16 is close. For insert speeds, 32 is the best node size, because it is where the cost of copying larger nodes and the benefit of fewer re-allocations balanced out. As node sizes increase, a larger node and its pointers has to be copied with each insert. On the flip side, larger node sizes mean fewer allocations of new subnodes, because more node slots increase the number of keys each node can potentially contain. Having more space in each node leads to fewer node copies and fewer re-inserts of old keys because of fewer collisions in placing values. Similarly, 32 is the optimal node size for search because the benefit of traversing fewer nodes to reach the data and the cost of computations on larger bitmaps reached a balance.

The increases in speed after node size 8 are less dramatic than the big increase in speed after node size 4, but minor optimizations can add up over time in applications, especially in ones that heavily use the data type. This supports the hypothesis that larger node sizes have better performance. The data suggests that 32 is the most efficient node size for HAMTs in Python, refuting my hypothesis that insert and search speeds would continue to increase with larger node sizes. The hypothesis did not take into account the cost of copying larger nodes, or the computations on larger bitmaps made necessary by the bitmap compression technique. Now that insert and search speeds have been discussed, the paper will move on to memory usage of HAMTs with different node sizes.

After determining the best node size, it is meaningful to check its performance against Python dictionaries and compare because functional data structures can be useful in non-functional languages, such as C# (Sturm, 2011). I tested Python dictionaries as intended, using mutations to insert the data, and persistently, by copying the dictionary with each insert.

Graph 5 below compares the use of Python dictionaries normally through mutation (ephemeral) and persistently through copying before each insert. To compare different amounts of data, the number of iterations is uniformly 1000 for each test, unlike previous testing. Note the logarithmic scale used due to the high variance in the data.

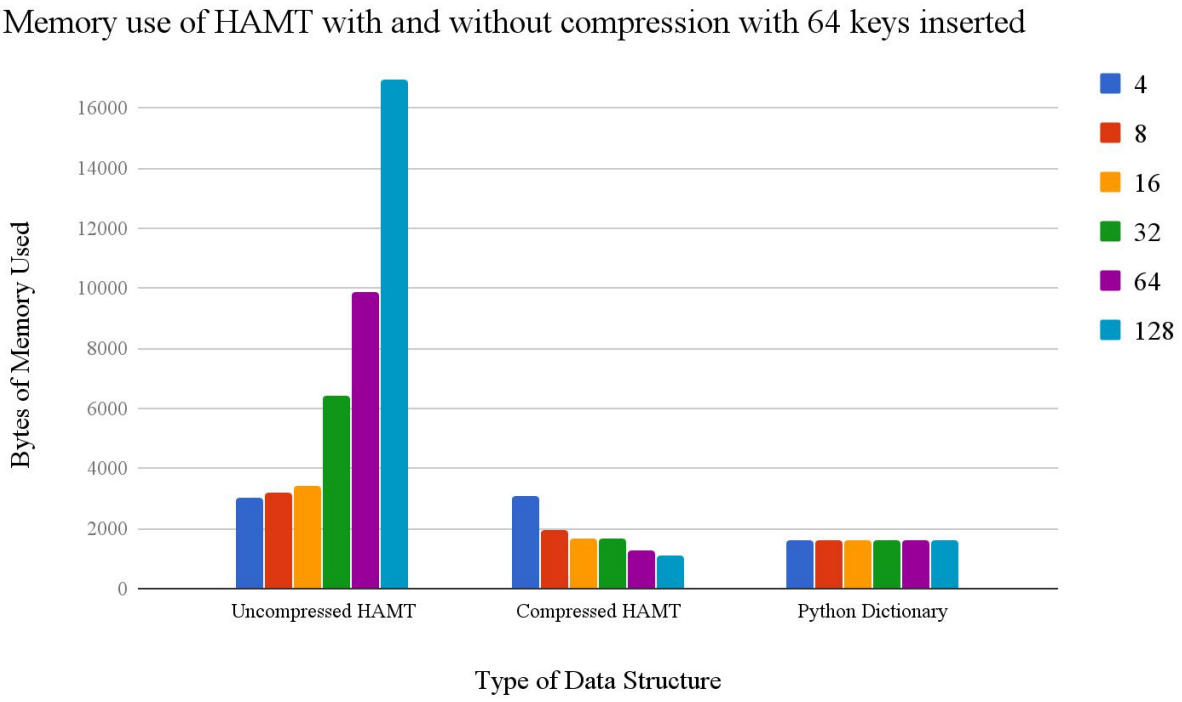
Graph 5



Looking at Graph 5, the ephemeral dictionary is still much faster than the HAMT in most cases, and the gap in speed increases as the amount of data increases. With 64 points, the HAMT is about 25 times slower, and about 30 times slower with 512 points of data. An important part of their performance is their memory use. Graph 6 below compares the data structures when containing 64 points of data measured in bytes. 64 points of data is fairly representative of all

amounts. Node size does not vary for the Python dictionary, because dictionaries do not have nodes.

Graph 6



Graph 6 displays memory usage of HAMTs with and without compression. Memory usage for the uncompressed HAMT goes up as expected, while the compressed HAMT's memory usage actually goes down as node size increases. This refutes the second part of the hypothesis, that larger node sizes would introduce a memory overhead. The reason that the uncompressed HAMT uses more memory is the extra space wasted by NULL pointers in larger node sizes. In comparison, the compressed HAMT uses much less memory compared to the uncompressed HAMT and sometimes even less than Python dictionaries. Memory usage of the compressed HAMT is much better than expected, especially for large node sizes. Larger node

sizes save space in the compressed HAMT because there are fewer nodes, which is more efficient because there are fewer bitmaps and other Python allocations for lists.

Overall, the compressed HAMT data structure when implemented in Python performs best with a node size of 32, maximizing insert and search speeds while using less memory compared to lower node sizes. The decrease in memory use in HAMTs with greater node sizes does not justify the significantly lower insert and search speeds.

Implications

Using HAMTs

The HAMT trades access and insert speeds for persistence. If old versions need to be tracked, all the previous versions are available with minimal memory use because most of the data is shared between them (Pitts, 2000). In addition, they can be used in functional programming, and introduce many safety and testing features. If compared to the persistent use of dictionaries in Graph 5, HAMT is by far faster for insert time in larger data sets, so that a persistent use of dictionaries gives far less performance with datasets larger than 64 and with the drawback that the entire dictionary has to be copied. In terms of memory use, as mentioned before, HAMT with compression and a large node size is very close to dictionaries as well. Using dictionaries persistently, the search time is unchanged compared to HAMT because accessing values does not alter the data structure, and is still about 25 times faster for dictionaries due to only needing to hash the key and directly index. The results imply that in PyPy, the HAMT is a viable choice as a persistent data type.

Limitations and Possibilities

One limitation of this investigation is the lack of further optimizations possible in the HAMT implementation written. These are used in HAMT implementations in other languages, so it would have been better to test with these optimizations. A major one mentioned in the background section is the CHAMP version, which separates key value pairs and subnodes. Instance checks are slow and the CHAMP algorithm eliminates the need for them (Steindorfer, 2015). It seems likely that this optimization would not interfere with comparing node sizes, because it is only a reordering of the list, but it certainly would have speeded up search time for comparisons with dictionaries. In addition, if my implementation were to be optimized further, it could be published and used by Python developers.

An expansion to this project could be to test HAMT implementations in other languages. While testing HAMT in Python is good for demonstrating concepts in another language and making comparisons within Python, it would have been beneficial to test HAMT implementations with varying node sizes in another language, such as Haskell or Clojure. This could yield results that could determine whether or not the currently used node size is the most efficient in that language. Given the results pointing in the direction of 32 being the optimal node size, it is possible that a different node size may benefit one of these languages.

Concluding words

The primary goal of this investigation is to determine how different node sizes affect the performance of the HAMT when implemented in Python. The HAMT is implemented in Python without and with the bitmap compression technique with varying node sizes, and compared running in the Python interpreter PyPy. The initial hypothesis is not supported by the data in

insert and access speeds observed in Graphs 1-3, or by the data for memory use shown in Graph 4. HAMT data structures that use compression with bitmaps were shown to use less memory with higher node sizes, and be fastest with a node size of 32. Thus, it is concluded that HAMT implementations with the node size of 32 perform better than HAMT implementations with other node sizes in Python.

HAMT is then considered as an alternative to the Python dictionary when used persistently in PyPy, and found to be effective compared to the dictionary when there is a need to add to the data structure often. Python dictionaries are much faster when being searched or being modified ephemerally.

If this research were to be extended, HAMT in other languages could be tested with different node sizes and compared to these results. It could potentially optimize a HAMT implementation in a language that uses the HAMT as a hash table further, thus optimizing any program written in the language that uses it.

References

Bagwell, Phil. Fast and space efficient trie searches. No. LAMP-REPORT-2000-001. 2000.

Bagwell, Phil. Ideal hash trees. No. LAMP-REPORT-2001-001. 2001.

Bentley, Jon, Don Knuth, and Doug McIlroy. "Programming pearls: A literate program." Communications of the ACM 29.6 (1986): 471-483.

Bentley, Jon, and Bob Sedgewick. “Ternary Search Trees.” Computer Science Department, Princeton University, 1999, www.cs.upc.edu/~ps/downloads/tst/tst.html. Accessed 22 Aug. 2017.

Brandais, R. 1959. File searching using variable length keys. In Proceedings of Western Joint Computer Conference, Volume 15 (1959), pp. 295–298.

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). Introduction to Algorithms (1st ed.). Massachusetts Institute of Technology. pp. 198–296. ISBN 978-0-262-03384-8.

Driscoll, James R., et al. "Making data structures persistent." Journal of computer and system sciences 38.1 (1989): 86-124.

Guttag, John V. Introduction to computation and programming using Python. Mit Press, 2013.

Hughes, John. "Why functional programming matters." The computer journal 32.2 (1989): 98-107.

Lenski, Dan. “Python List vs. Array - when to use?” Stack overflow, 6 Oct. 2008, stackoverflow.com/questions/176011/python-list-vs-array-when-to-use. Accessed 23 Aug. 2017.

Miller, Bradley N., and David L. Ranum. Problem solving with algorithms and data structures

using Python. Decorah, IA, Brad Miller, David Ranum, 2014.

Okasaki, Chris. Purely functional data structures. Cambridge University Press, 1999.

Pitts, Robert I. “Trie.” Trie, Boston University, 2000, www.cs.bu.edu/teaching/c/tree/trie/. Accessed 25 June 2017.

Pypy documentation. http://doc.pypy.org/en/latest/introduction.html. Accessed 23 Sep. 2017.

Python documentation. https://docs.python.org/3/contents.html. Accessed 20 June 2017.

Sturm, Oliver. Functional Programming in C#. John Wiley and Sons. (2011). 167-170

Schuck, Peter, and Cliff Rodgers. Leveling up Clojure's Hash Maps. Bendyworks, 18 Dec. 2015, bendyworks.com/blog/leveling-clojures-hash-maps. Accessed 21 Aug. 2017.

Schuck, Peter, and Cliff Rodgers. Implementation of HAMT in Clojure. Github repository. https://github.com/bendyworks/lean-map. Accessed 23 Aug. 2017.

Steindorfer, Michael J., and Jurgen J. Vinju. "Optimizing hash-array mapped tries for fast and lean immutable JVM collections." ACM SIGPLAN Notices. Vol. 50. No. 10. ACM, 2015.

Steindorfer, Michael, and Jurgen Vinju. "Code Specialization for Memory Efficient Hash Tries." (2014).

Suzanne, Thibault. Implementing HAMT for OCaml. Gagallium, 2 Feb. 2013, gallium.inria.fr/blog/implementing-hamt-for-ocaml/. Accessed 22 Aug. 2017.

Appendix

To access the source code used, visit my github repository at:

https://github.com/oflatt/hamtpython

Figure 1- Insert Function in Uncompressed HAMT in Python

```
def insert(self, key, value):
        done = False
        # make a new list that shares its data with the old one
        newhead = self.head.copy()
        # use l as the current node
        l = newhead
        depth = 0

        while not done:
            # get the needed bits for this depth
            index = (key >> depth*self.numbits) & (self.nodesize-1)
            # If there is no key in the current index
            if l[index] == None:
                l[index] = key
                l[index + self.nodesize] = value
                done = True

            # if there already is a key at the index and a subnode is needed
            elif isinstance(l[index], int):
                oldkey = l[index]
                oldval = l[index + self.nodesize]
                # if it has the same value, no insert is needed
                if oldval == value:
                    done = True
                else:
                    # key value now notes that it is a subnode
                    l[index] = 's'
                    l[index + self.nodesize] = [None] * self.nodesize * 2
                    # adds the old key and value to the newly created subnode
                    depth += 1
                    l = l[index + self.nodesize]
                    oldindex = (oldkey >> depth*self.numbits) & (self.nodesize-1)
                    l[oldindex] = oldkey
                    l[oldindex + self.nodesize] = oldval
                    # loops and tries to insert the new key and value into the new node

            # otherwise there must be a subnode, and we go down it, increasing depth
            else:
                depth += 1
```

```
            l = l[index + self.nodesize]
            # now test the next depth, loop

        # after the loop, return the new hamt with inserted data
        return Hamt(self.nodesize, self.numbits, newhead)
```

Source: https://github.com/oflatt/hamtpython/blob/master/hamt.py

Figure 2- Inserting a key and value into a node in a compressed HAMT in Python

```
# We place the key and value in the node if it's not occupied
if not (bitmap >> index) & 1:
        #set the bit in the bitmap to 1
        mask = 1 << index
        l[0] = bitmap | mask
        # now insert in right place, growing list
        l.insert(listindex, key)
        #can't use secondlistindex because not the list has changed size
        l.insert(len(l)-lengthafter, value)
        done = True
```

Source: https://github.com/oflatt/hamtpython/blob/master/compressedhamt.py

Table 1- Time in milliseconds taken to insert data into compressed hamt with varying node sizes in PyPy (rounded)

		Node Size					
Iterations	Number of Keys	4	8	16	32	64	128
100000	4 keys	398	280	272	274	310	374
5000	64 keys	517	372	345	343	392	495
500	512 keys	556	457	414	389	456	607
100	2048 keys	522	434	385	387	411	541

Table 2- Time in milliseconds taken to access data in a compressed hamt with varying node sizes in PyPy (rounded)

		Node Size					
Iterations	Number of Keys	4	8	16	32	64	128
100000	4 keys	396	215	166	183	244	441
5000	64 keys	450	314	308	299	337	442
500	512 keys	445	352	328	305	358	516
100	2048 keys	426	335	303	305	318	443

Table 3- Insert time of compressed HAMT in milliseconds with node size 32 compared to persistent and ephemeral use of Python dictionaries in PyPy with the same number of iterations (Logarithmic scale)

		Type of Data Structure		
Iterations	Number of Keys	Ephemeral Dictionary	Persistent Dictionary	Compressed HAMT with Node Size 32
100000	4 keys	26	80	274
5000	64 keys	273	5313	6858
500	512 keys	2480	382539	77796
100	2048 keys	11008	6960462	386649

Table 4- Memory use of data structures with 64 values inserted and varying node sizes

	Node Size					
	4	8	16	32	64	128
Uncompressed HAMT	2992	3200	3444	6424	9864	16960
Compressed HAMT	3096	1960	1680	1640	1248	1088
Python Dictionary	1588	1588	1588	1588	1588	1588