Hash Array Mapped Tries

24 May 2016

A Hash Array Mapped Trie (HAMT) is a structure for organizing arbitrary data in a broadly-branching tree. HAMTs are commonly used to build immutable Hash Maps in functional programming languages. By using a value's hash code to represent a unique "path" into the tree, we can build a Hash Map on top of a tree, rather than on an Array-based table as is more commonly seen. The structure is more complex than a simple Hash Table, but provides a few key benefits, including:

- Ability to grow the map indefinitely without re-sizing or chaining (no re-hashing penalties)
- Ability to share repeated structure between similar trees

This last point is very powerful from the perspective of copying or modifying – we can represent a modified copy of a Hash Trie by duplicating any changed values but sharing the remaining (unchanged) structure with the previous tree. We get the conceptual benefits of an immutable structure but the efficiency of a traditional mutable collection.

Hash Tries have been getting a lot of attention over the last several years as a means of implementing efficient immutable data structures, especially for functional programming languages. I learned about them through exploring Clojure, which uses HAMTs as the basis for its immutable hash-maps. They also make an appearance in Scala, Haskell, and Elixir.

HAMT Structure

To implement this data structure, we'll rely on a few key tools:

1. A Hashing Algorithm for uniquely differentiating pieces of data. Many languages already provide this – in Ruby you can access an object's hashcode by calling `#hash` on it, or you can use a hashing function like the `SHA1` implementation including in the `digest` library.
2. A trie with very high branching factor – this lets us store lots of data in a very shallow (and speedy) structure.
3. Bitwise operations to "consume" the data's hash code in small chunks, turning a hash code into a "path" to the data's location in the trie.

So what does all this look like in practice? Let's look at an example creating a HAMT of order 32.

Within the trie, each level can store 3 things:

1. A key
2. An associated value
3. Connections to up to 32 nested child trees

When we want to insert a key-value pair, we'll use the key's hash code to choose a path through the trie until we find an empty position to insert it.

For retrieval, we simply do the same thing in reverse – hash the key, find the pathway through the trie represented by this hashcode, and check tree nodes until we either find the desired key or "bottom out" at the end of the tree.

Insertion Algorithm

Let's walk through the insertion process in more detail.

To insert a piece of data, we need to find an appropriate path in the trie in which to place it. As we'll see, this path is ultimately determined by the key's hash value.

As we walk down the trie, we'll be looking for 3 possible cases:

1. The current trie node is empty, so we can insert our new key and value here
2. The current trie node is not empty, but its key is equal to the one we are trying to insert, so we can overwrite its value
3. The current trie node is not empty, and its key value is not equal to ours, so we need to go deeper in the trie.

Insertion Case 1

Consider inserting a new K/V pair into an empty trie. We'll insert the key "pizza" with the value "yum". Our trie is empty so far, so the root tree node has no key and value, so we can insert our pair there.

Pretty easy so far.

Insertion Case 2

Let's get the second easy case out of the way – overwriting that K/V pair. We can insert the key "pizza" again, this time with the value of "real yum".

We find that the root node is not empty, but its key is equal to the one we're trying to insert, so we simply change the value.

Also pretty easy.

Insertion Case 3

Here is where things start to get more interesting.

Let's insert the key "calzone" with the value "aw yiss".

We first check the current (root) node – it does have a key and value, and the key is *not* the one we're trying to insert. We need to go deeper into the trie to find a place for our new pair.

To insert a key, we first need to generate its hash value. Again, in Ruby, we can use one of the hashing functions included in the Digest library. This gives us a (large) numeric value representing a unique digest of that piece of data.

For example:

```
[1] pry(main)> require "digest"
=> true
[2] pry(main)> Digest::SHA1.hexdigest("calzone").to_i(16)
=> 334703588949583183218034173573122019749278332384
```

To walk the trie and an appropriate location for this element, we'll "consume" this hash-code in 5-bit chunks.

Why 5 bits at a time?

This is determined by the branching factor of the tree – with an order-32 trie, we have 32 possible children from each node in the tree. A 5-bit hash-code chunk allows us to concisely represent all 32 possible child branches using a single bitmap. `(=2 ** 5 == 32)`

To get the numeric value of the first 5 bits of our hashcode, we can bitwise `AND` it with a 5-bit number containing all "on" bits:

```
[13] pry(main)> 31.to_s(2)
=> "11111"
[14] pry(main)> Digest::SHA1.hexdigest("calzone").to_i(16) & 31
=> 0
```

This tells us that the "right-most" 5 bits of the number `334703588949583183218034173573122019749278332384` ("calzone"'s hash code) are `00000`, or 0.

This tells us the position in the current node's children array to insert this element.

Thus we can move to the `0` th subtree under our current one and retry our insertion algorithm. In our case, “calzone” is only the second element to be inserted in the trie, so the =0=th child of the “pizza” node will be empty, and we can insert our data there.

Consuming the hash code

We mentioned that we would “consume” the key’s hash code in 5-bit chunks. This helps us fully exploit the wide branching factor of the trie to insert a lot of elements in a fairly shallow data structure.

If we simply re-use the same 5 (rightmost) bits that we used in the previous example, we effectively turn our trie into a collection of 32 linked lists, since all elements that share an initial 5-bit value will stack up on one another in a chain.

We would prefer to get more of a “zig-zag” effect, and we can achieve this by making sure we use a different 5-bit chunk at each layer in the trie.

To do this, we’ll use another bit-wise operator, the **right shift**.

A bitwise shift simply takes the bits that make a number and slides them in one direction or another.

In the case of a left shift, we move the existing bits to the left, usually padding them with 0’s on the righthand side.

For example:

```
[20] pry(main)> 15.to_s(2)
=> "1111"
[21] pry(main)> (15 << 4).to_s(2)
=> "11110000"
```

In our case, we just want to consume the next 5 bits of our hash code value, so we can use a right shift of 5 bits.

Consider our “calzone” example from before:

```
[24] pry(main)> (Digest::SHA1.hexdigest("calzone").to_i(16) >> 5) & 31
=> 15
```

We now get a completely different subtrie index, helping us avoid the “stacking” behavior we would get if we just re-used the existing one. As we walk down the trie, we want to use this technique to shift off 5 bits at each layer.

Retrieval Algorithm

The retrieval process is effectively the same. We'll simply retrieve the located value rather than inserting one. Consider the same 3 cases:

1. The current tree node is empty – this means we have "bottomed out", so our key must not exist in the trie
2. The current tree node contains the key you're searching for, so retrieve its value.
3. The current tree node is not empty, but doesn't contain the key we're looking for. Use another 5-bit slice of the hash code to identify the next step to take into the trie.

HAMT Performance

The strength of the HAMT is its wide branching factor. The 32-bit factor is common because it can be manipulated efficiently on 32-bit processors, but you could in theory use an even larger factor if needed.

This branching factor allows us to store a large amount of keys and values in a relatively shallow tree which will still be very quick to traverse.

For example in just 6 layers, we could store `33,554,432` (`32 ** 5`, assuming the root only stores 1 pair) keys and values.

Technically, the retrieval performance of our Tree will be logarithmic, as opposed to the Constant-time performance offered by traditional Hash Map implementations. However since the log base is so large, the growth flattens out very quickly and in practice isn't much worse off than a traditional hash table.

Other Considerations – Structural Sharing

We mentioned the ability of our tries to potentially share duplicated structure with other tries. This is a common approach to creating immutable or "persistent" hash maps and is used in several functional languages like Clojure, Scala, and Frege.

The goal for this technique is to preserve every intermediate state of the Map (i.e. they "persist"). Thus each operation on the map should generate a new map value rather than modifying an existing one in place.

This would be problematic if we had to completely copy every node in the trie each time we changed anything. But because of the trie's nested structure, we have a better option.

Whenever we need to change the trie, we duplicate the node in question as well as all the nodes within its path to the root.

Thus we get a new root node (this represents the "new" Map produced by our operation), and a new path to the internal node that was actually changed.

The nodes that we copy can continue referring to the other existing nodes so that those don't have to be copied. In practice this allows us to produce a "copy" of the entire trie by actually copying only a handful of nodes.

Further Reading

- The data structure was invented by Phil Bagwell, and you can find the original paper on it here.
- Rich Hickey discussing HAMTs and other Clojure internals
- ClojureWest Talk about Optimizing Clojure's Persistent Data Structures