Ruby Under a Microscope

Learning Ruby Internals Through Experiment

Pat Shaughnessy

Preview of one chapter - May 2012

Discussion/feedback:
http://patshaughnessy.net/ruby-under-a-microscope
@pat_shaughnessy

Table of Contents

Introduction 1

- Why bother to study Ruby internals? 1
- My approach in this book: theory and experiment 2

How Hashes Scale From One To One Million Elements 3

- Theory: Hash tables in Ruby 4
- Experiment 1: Retrieving a value from hashes of varying sizes 9
- Theory: How hash tables expand to accommodate more values 10
- Experiment 2: Inserting one new element into hashes of varying sizes 14
- Theory: Why Hashes will be faster in Ruby 2.0 18
- Experiment 3: Inserting one new element into hashes of varying sizes, for Ruby 2.0 20
- Theory: How Ruby implements hash functions 22
- Experiment 4: Using objects as keys in a hash 26
- Theory: How Ruby saves order information in hashes 30
- Experiment 5: Iterating over elements inserted into a Hash 32
- Alternate theories: Hashes in JRuby 33
- Alternate theories: Hashes in Rubinius 36
- Conclusion 38
- Appendix: Experiment Code 39

Introduction

Why bother to study Ruby internals?

Everyday you need to use your car to drive to work, drop your kids off at school, etc., but how often have you ever thought about how your car actually works internally? When you stopped at a red light on your way to the grocery store last weekend were you thinking about the theory and engineering behind the internal combustion engine? No, of course not! All you need to know about your car is which pedal is which, how to turn the steering wheel and a few other important details like shifting gears, turn indicator lights, etc.

At first glance, studying how Ruby is implemented internally is no different: why bother to learn how the language was implemented when all you need to do is use it? Who cares how the Ruby Array or Hash objects work internally, for example; all I need to know is some basic usage - how to add something to an array, how to get it back again later.

Well, in my opinion there are a few good reasons why you should take the time to study the internal implementation of Ruby:

- You'll become a better Ruby developer. By studying how Ruby works internally, you can become more aware of how Matz and the rest of the Ruby core team intended the language to be used. You'll learn what works well, what performs fast - and what doesn't. You'll be a better Ruby developer by using the language as it was intended to be used, and not just in the way you prefer to use it.

- You can learn a lot about computer science. Beyond just appreciating the talent and vision of the Ruby core team, you'll be able to learn from their work. While implementing the Ruby language the core team had to solve many of the same computer science problems that you might have to solve in your job or open source project. This isn't true of the car driving analogy, obviously; learning about the mechanical and electrical engineering details of your car's engine won't help you become a better driver.

- It's fun! I find learning about the algorithms and data structures Ruby uses internally fascinating, and I hope you will too.

My approach in this book: theory and experiment

“It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong.” - Richard Feynman

In Ruby Under A Microscope I’m going to teach you how Ruby works internally. I’ll use a series of simple, easy to understand diagrams that will show you what is happening on the inside when you run a Ruby program. Like a physicist or chemist, I’ve developed a **theory** about how things actually work based on many hours of research and study. I’ve done the hard work of reading and understanding Ruby’s internal C source code so you don’t have to. My goal is that some of these diagrams come back into your mind the next time you use a particular feature of Ruby.

But like any good scientist, I know that theory is worthless without some hard evidence to back it up. Therefore after explaining some aspect of Ruby internals, some feature or behavior of the language, I’ll perform an **experiment** to prove that my theory was correct. To do this I’ll use Ruby to test itself! I’ll run some small test Ruby scripts and see whether they produce the expected output, whether they run as fast or as slowly as I expect, whether Ruby actually behaves the way my theory says it should.

I assume you are a Ruby developer who uses the language every day and who is interested in learning more about how the language works internally. However, I don’t expect you to be fluent in the C programming language that Matz and the Ruby core team used to build the interpreter. I won’t walk through the C code step by step, explaining all of the in’s and out’s of the C coding details. If you’re interested in that sort of thing, then your best resource will be the Ruby Hacking Guide, originally written in Japanese and partially translated into English.

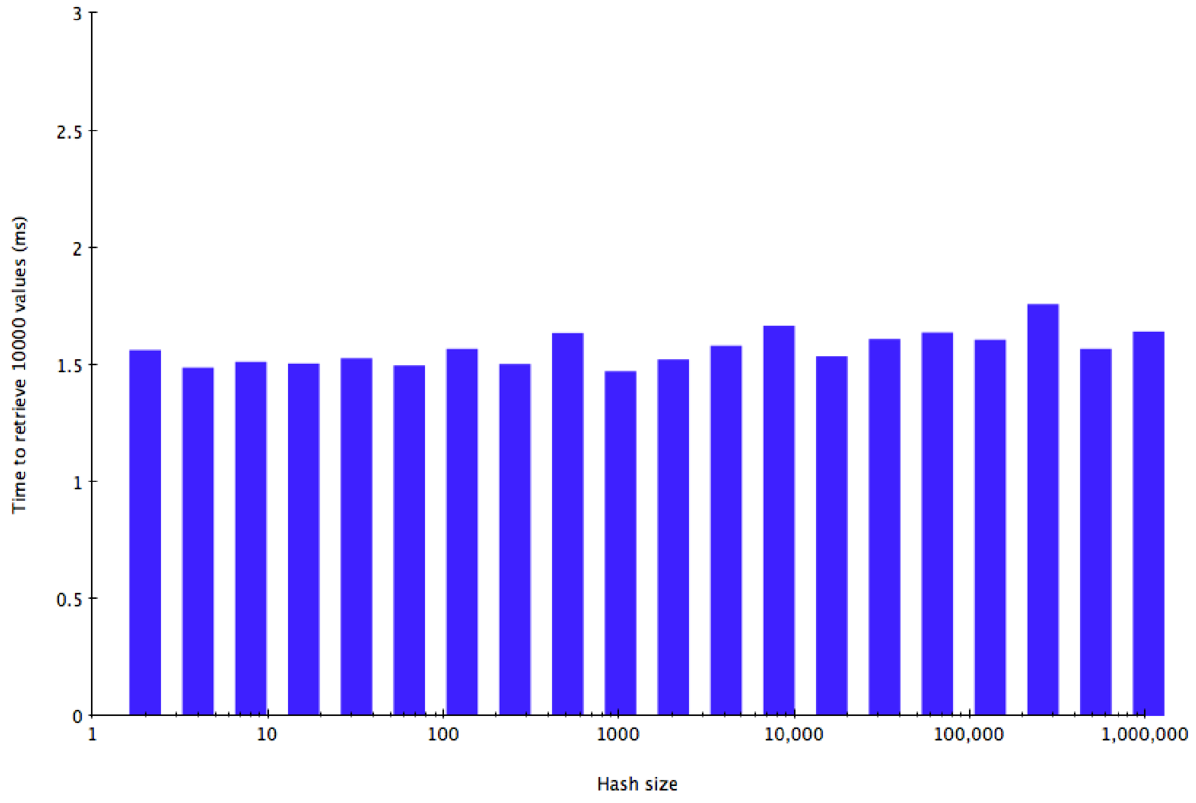
For those people familiar with C, however, I will show a few vastly simplified snippets of C code to give you a more concrete sense of what’s going on inside Ruby. I’ll also indicate which MRI C source code I found the snippet in; this will make it easier for you to get started studying the MRI C code yourself if you ever decide to. Like this paragraph, I’ll display this information on a yellow background.

If you’re not interested in the C code details, just skip over these yellow sections.

How Hashes Scale From One To One Million Elements

You probably know very well how to use the Hash object in your Ruby programs, but do you know what the most remarkable and important feature of Ruby's Hash object is? What makes the Hash object interesting is not that it can save and lookup values using keys, but that it can do it quickly… *regardless of how many elements it has*.

Here's some data proving this is true:



This chart shows how long it takes Ruby 1.9 to search for and retrieve values from a hash, for hashes of different sizes. The y-axis indicates how long it took my Ruby test code to retrieve 10,000 values from a hash, in milliseconds. Along the x-axis I show the size of the hash using a logarithmic scale - in other words the number of other keys Ruby had to search through to find the key I asked for. Clearly the Ruby Hash object is very fast; on my laptop it can search for and find a key in hash 10,000 times in about 1.5ms. Doing the math, on average it takes Ruby only 0.15 *microseconds* to find a given key and return the value.

But what’s really amazing about this is not just that Ruby is fast, it’s that Ruby is equally fast for a hash containing a million keys as it is for a hash containing just one key! What’s remarkable about this chart is that it’s more or less flat.

If you think about this for a minute, the Hash object is really a mini search engine: somehow Ruby can take any key, search for it very quickly among possibly thousands or even millions of other keys, and then return just the single value that corresponds to that key. Similarly, when you save a new key/value pair into a hash Ruby first quickly determines whether a value for that key is already present and overwrites it if there is one. How does it do this? Does Ruby build a search index of some kind?

In this chapter I’ll explain what a hash table is and how it uses a hash function to group data elements into different bins, allowing Ruby later to search for them very, very quickly. I’ll also explain how the hash tables expand to accommodate any number of keys and values. Along the way I’ll perform a series of experiments to provide some data, some evidence that Ruby actually uses hash tables and hash functions to implement the Hash object internally. Finally, I’ll explore how hashes save information about order in the hash table - do I get them back in the same order that I inserted them?

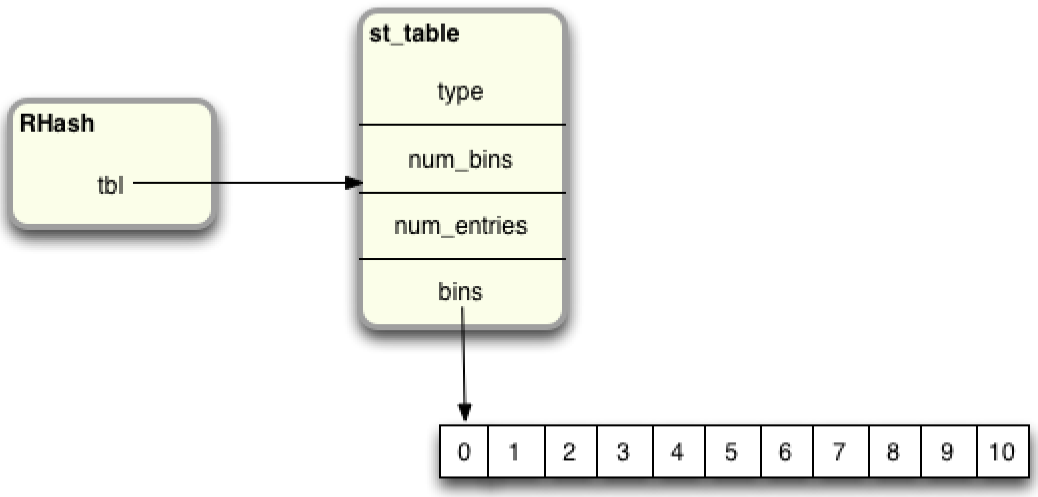
$$\nabla \cdot \vec{E} = \frac{\rho}{\varepsilon_0} = 4\pi k \rho$$

$$\nabla \cdot \vec{B} = 0$$

Theory: Hash tables in Ruby

After studying the MRI C code Ruby uses to implement the Hash object, I very quickly found that Ruby uses something called a “hash table” to save the keys and values you save in any hash. Hash tables are a commonly used, well known, old concept in computer science. They organize values into groups or “bins” based on an integer value calculated from each value called a “hash.” Later when you need to search for and find a value, by recalculating the hash value you can figure out which bin the value is contained in, speeding up the search.

Here’s a high level diagram showing a single hash object and its hash table:



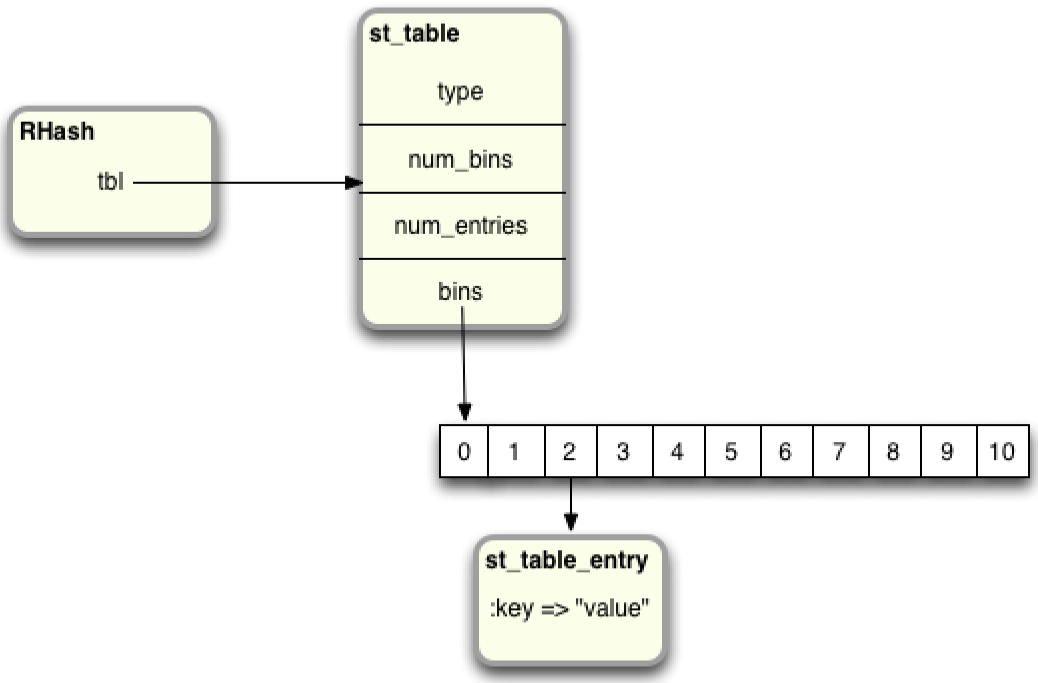
On the left is the “RHash” structure; this is short for “Ruby Hash.” All of the other important object types used inside of Ruby are represented by similar structures called “RFile,” “RArray,” “RValue,” etc. Each of these structures, including RHash, contains a set of internal, system values about that object that Ruby needs to keep track of.

On the right, I show the hash table used by this hash, represented by the “st_table” structure. This C structure contains the basic information about the hash table, such as the number of entries saved in the table, the number of bins and a pointer to the bins. Each RHash structure contains a pointer to a corresponding st_table structure. Finally, I show some empty bins on the lower right. Ruby 1.8 and Ruby 1.9 initially create 11 bins for a new, empty hash.

The best way to understand how a hash table works is by stepping through an example. Let’s suppose I add a new key/value to a hash called “my_hash:”

```
my_hash[:key] = "value"
```

While executing this line of code, Ruby will create a new structure called an “st_table_entry” and will save it into the hash table for “my_hash:”



Here you can see Ruby saved the new key/value pair under the third bucket, #2. Ruby did this by taking the given key, the symbol ":key" in this example, and passing it to an internal hash function that returns a pseudo-random integer:

```
some_value = internal_hash_function(:key)
```

Next, Ruby takes the hash value, "some_value" in this example, and calculates the modulus by the number of bins… i.e. the remainder after dividing by the number of bins:

```
some_value % 11 = 2
```

In this diagram I imagine that the actual hash value for ":key" divided by 11 leaves a remainder of 2. Later in this chapter I'll explore the hash functions that Ruby actually uses in more detail.

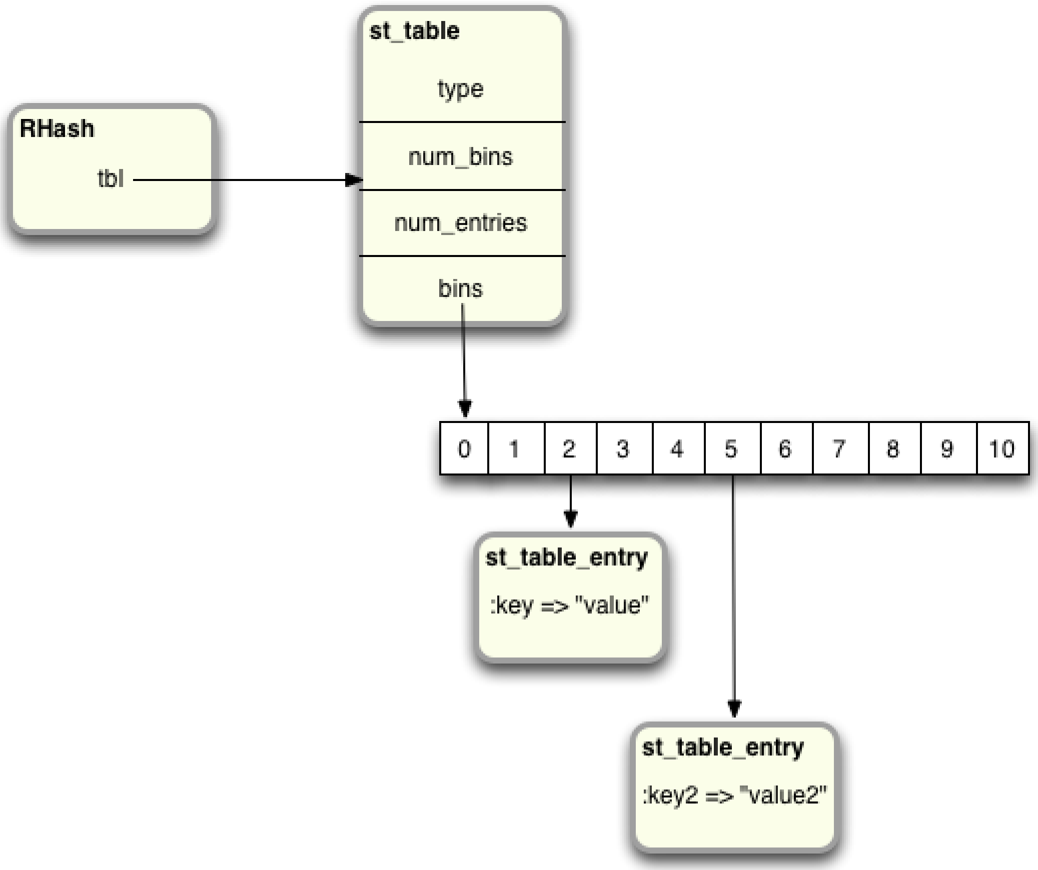
Now let's add a second element to the hash:

```
my_hash[:key2] = "value2"
```

And this time let's imagine that the hash value of ":key2" divided by 11 yields a remainder of 5:

```
internal_hash_function(:key2) % 11 = 5
```

Now you can see Ruby places a second “st_table_entry” structure under bin #5, the sixth bin:



The benefit of using a hash table comes later, when you ask Ruby to retrieve the value for a given key:

```
puts my_hash[:key]
=> "value"
```

If Ruby had saved all of the keys and values in an array or linked list, then it would have to iterate over all the elements in that array of list, looking for :key. This might take a very long time, depending on how many elements there were. But using a hash table Ruby can jump straight to the key it needs to find by recalculating the hash value for that key. It simply calls the hash function again:

```
some_value = internal_hash_function(:key)
```

… redivides the hash value by the number of bins and obtaining the remainder, the modulus:

```
some_value % 11 = 2
```

… and now Ruby knows to look in bin #2 for the entry with a key of :key. In a similar way, Ruby can later find the value for :key2 by repeating the same hash calculation:

```
internal_hash_function(:key2) % 11 = 5
```

Believe it or not, the C library used by Ruby to implement hash tables was originally written back in the 1980’s by Peter Moore from the University of California at Berkeley, and later modified by the Ruby core team. You can find Peter Moore’s hash table code in the C code files “st.c” and “include/ruby/st.h”. All of the function and structure names use the naming convention “st_" in Peter’s hash table code.

Peter Moore’s hash table code plays a very important and central role in Ruby internals. It’s used not only by the Hash object, but in many other places also, for example to keep track of what methods are defined in each Ruby object class or module. In other words, Ruby uses Peter Moore’s hash table code to track its own internal data, and not only your data that you save in Hash objects.

Meanwhile, the definition of the “RHash” structure that represents every Ruby Hash object can be found in the include/ruby/ruby.h file. Along with RHash, here you’ll find all of the other primary object structures used in the Ruby source code: RString, RArray, RValue, etc.

Experiment 1: Retrieving a value from hashes of varying sizes

My first experiment will create hashes of wildly different sizes, from 1 element to 1 million elements and then measure how long it takes to find and return a value from each of these hashes. You can find my complete test code script in the appendix or on Github in case you want try this yourself. For now, here are the important bits of the test code. First, I create hashes of different sizes, based on powers of two, by running this code for different values of "exponent":

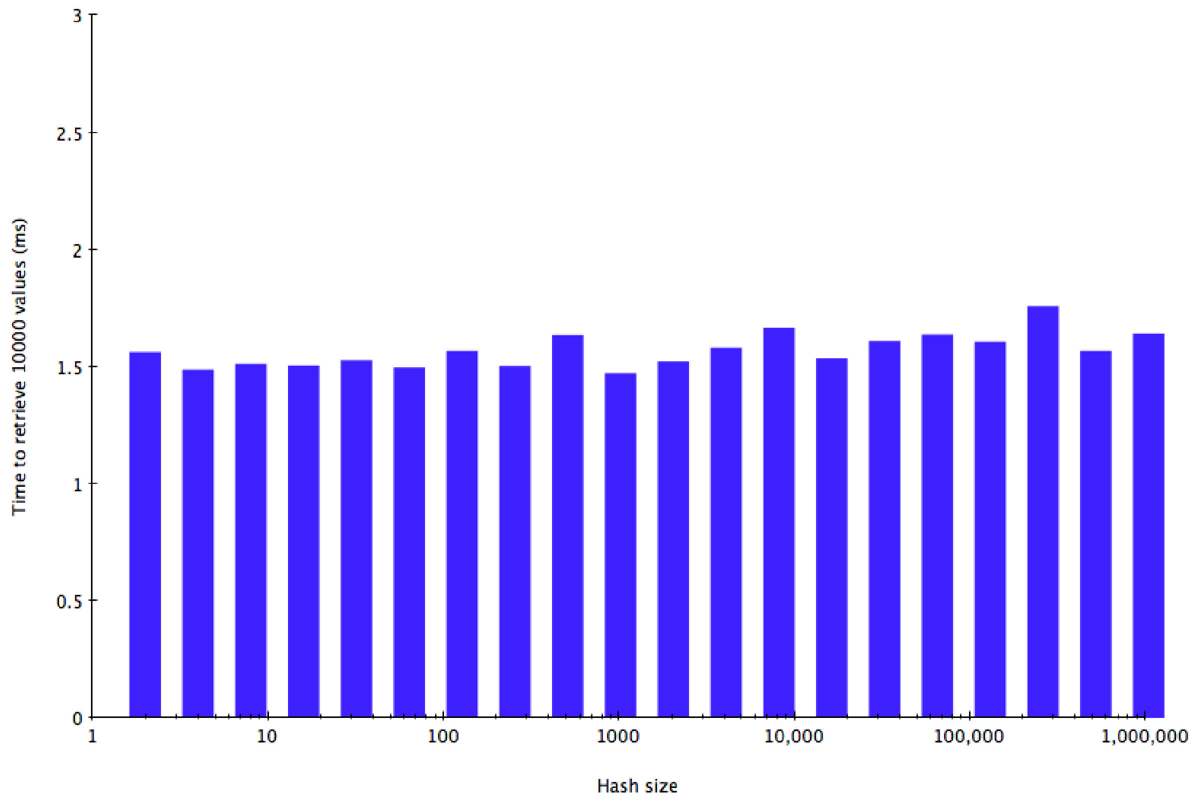
```
size = 2**exponent
hash = {}
(1..size).each do |n|
  index = rand
  hash[index] = rand
end
```

Here both the keys and values are random floating values. Then I measure how long it takes to find one of the keys, the "target_key", 10,000 times using the benchmark library:

```
Benchmark.bm do |bench|
  bench.report("retrieving an element from a hash with #{size} elements 10000 times") do
    10000.times do
      val = hash[target_key]
    end
  end
end
```

The results: small or very large Ruby hashes are equally fast!

I already showed this chart above; the results are remarkable: using a hash table internally, Ruby is able to find and return value from a hash containing over a million elements just as fast as it takes to return one from a small hash:



Clearly the hash function Ruby uses is very fast, and once Ruby identifies the bin containing the target key, it is able to very quickly find the corresponding value and return it. As I said above, what's remarkable about this is that the values in this chart are more or less flat.

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}$$

$$\nabla \times \vec{B} = \frac{\vec{J}}{\varepsilon_0 c^2} + \frac{1}{c^2}\frac{\partial \vec{E}}{\partial t}$$

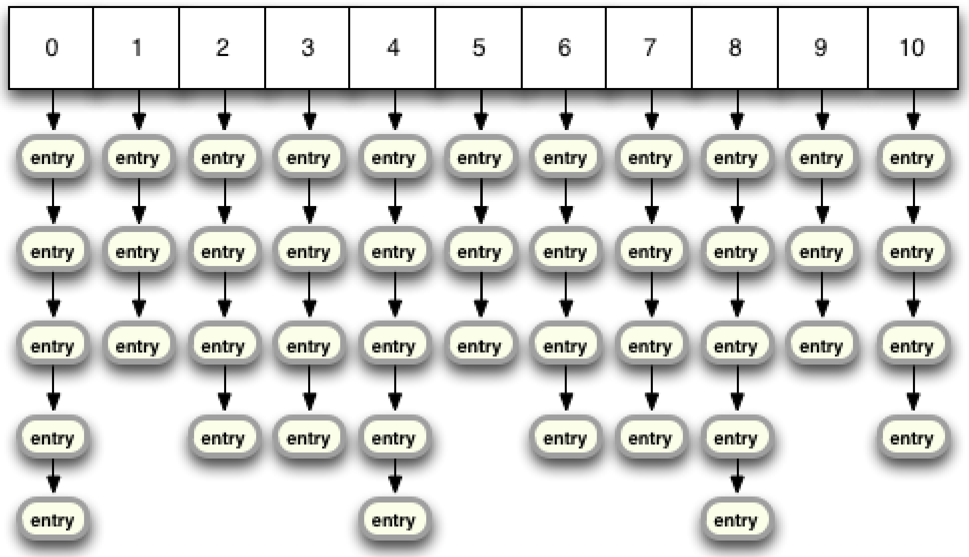
Theory: How hash tables expand to accommodate more values

You might be thinking ahead at this point, asking yourself: If there are millions of st_table_entry structures, why does distributing them among 11 bins help Ruby search quickly? Even if the hash function is fast, and even if Ruby distributes the values evenly among the 11 bins in the hash table, Ruby will still have to search among almost 100,000 elements in each bin to find the target key if there are a million elements overall.

Something else must be going on here. It seems to me that Ruby must add more bins to the hash table as more and more elements are added. Let's take another look at how Ruby's internal hash table code works. Continuing with the example from above… suppose I keep adding more and more elements to my hash:

```
my_hash[:key3] = "value3"
my_hash[:key4] = "value4"
my_hash[:key5] = "value5"
my_hash[:key6] = "value6"
... etc ...
```

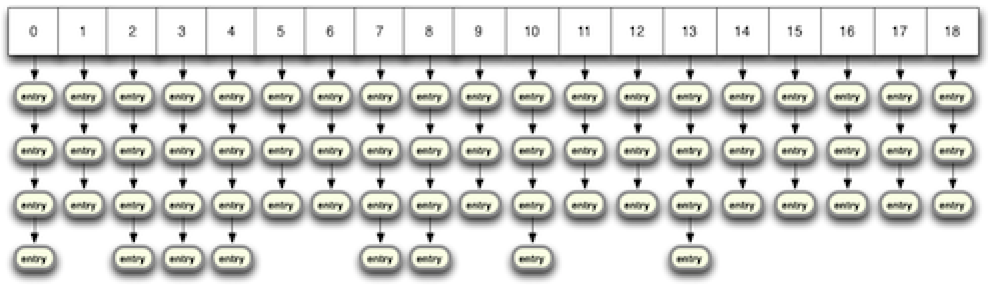
As I add more and more elements, Ruby will continue to create more st_table_entry structures and add them to different bins, depending on the modulus of the hash value for each key:



Ruby uses a linked list to keep track of the entries in each bin: each st_table_entry structure contains a pointer to the next entry in the same bin. As you add more entries to the hash, the linked list for each bin gets longer and longer.

To keep these linked lists from getting out of control, Ruby measures something called the “density” or the average number of entries per bin. In my diagram above, you can see that the average number of entries per bin has increased to about 4. What this means is that the hash value modulus 11 has started to return repeated values for different keys and hash values. Therefore, when searching for a target key, Ruby might have to iterate through a small list, after calculating the hash value and finding which bin contains the desired entry.

Once the density exceeds 5, a constant value in the MRI C source code, Ruby will allocate more bins and then “rehash”, or redistribute, the existing entries among the new bin set. For example, if I keep adding more key/value pairs, after a while Ruby will discard the array of 11 bins, allocate an array of 19 bins, and then rehash all the existing entries:



Now in this diagram the bin density has dropped to about 3.

By monitoring the bin density in this way, Ruby is able to guarantee that the linked lists remain short, and that retrieving a hash element is always fast - now after calculating the hash value Ruby just needs to step through 1 or 2 elements to find the target key.

You can find the “rehash” function - the code that loops through the st_table_entry structures and recalculates which bin to put the entry into - in the st.c source file at around line 316 in Ruby 1.8.7:

```
static void
rehash(table)
  register st_table *table;
{
  register st_table_entry *ptr, *next, **new_bins;
  int i, old_num_bins = table->num_bins, new_num_bins;
  unsigned int hash_val;

  new_num_bins = new_size(old_num_bins+1);
  new_bins = (st_table_entry**)Calloc(new_num_bins, sizeof(st_table_entry*));

  for(i = 0; i < old_num_bins; i++) {
    ptr = table->bins[i];
    while (ptr != 0) {
      next = ptr->next;
      hash_val = ptr->hash % new_num_bins;
      ptr->next = new_bins[hash_val];
      new_bins[hash_val] = ptr;
      ptr = next;
    }
  }
  free(table->bins);
  table->num_bins = new_num_bins;
  table->bins = new_bins;
}
```

The “new_size” method call here returns the new bin count, for example 19. Once Ruby has the new bin count, it allocates the new bins and then iterates over all the existing st_table_entry structures - all the key/value pairs in the hash. For each st_table_entry Ruby recalculates the bin position using the same modulus formula: hash_val = ptr->hash % new_num_bins. Then it saves each entry in the linked list for that new bin. Finally Ruby updates the st_table structure and frees the old bins.

In Ruby 1.9 and Ruby 2.0 the rehash function is implemented somewhat differently, but works essentially the same way.

Experiment 2: Inserting one new element into hashes of varying sizes

One way to test whether this rehashing or redistribution of entries really occurs is to measure the amount of time Ruby takes to save one new element into an existing hash of different sizes. As I add more and more elements to the same hash, at some point I should see some evidence that Ruby is taking extra time to rehash the elements.

I'll do this by creating 10,000 hashes, all of the same size, indicated by the variable "size":

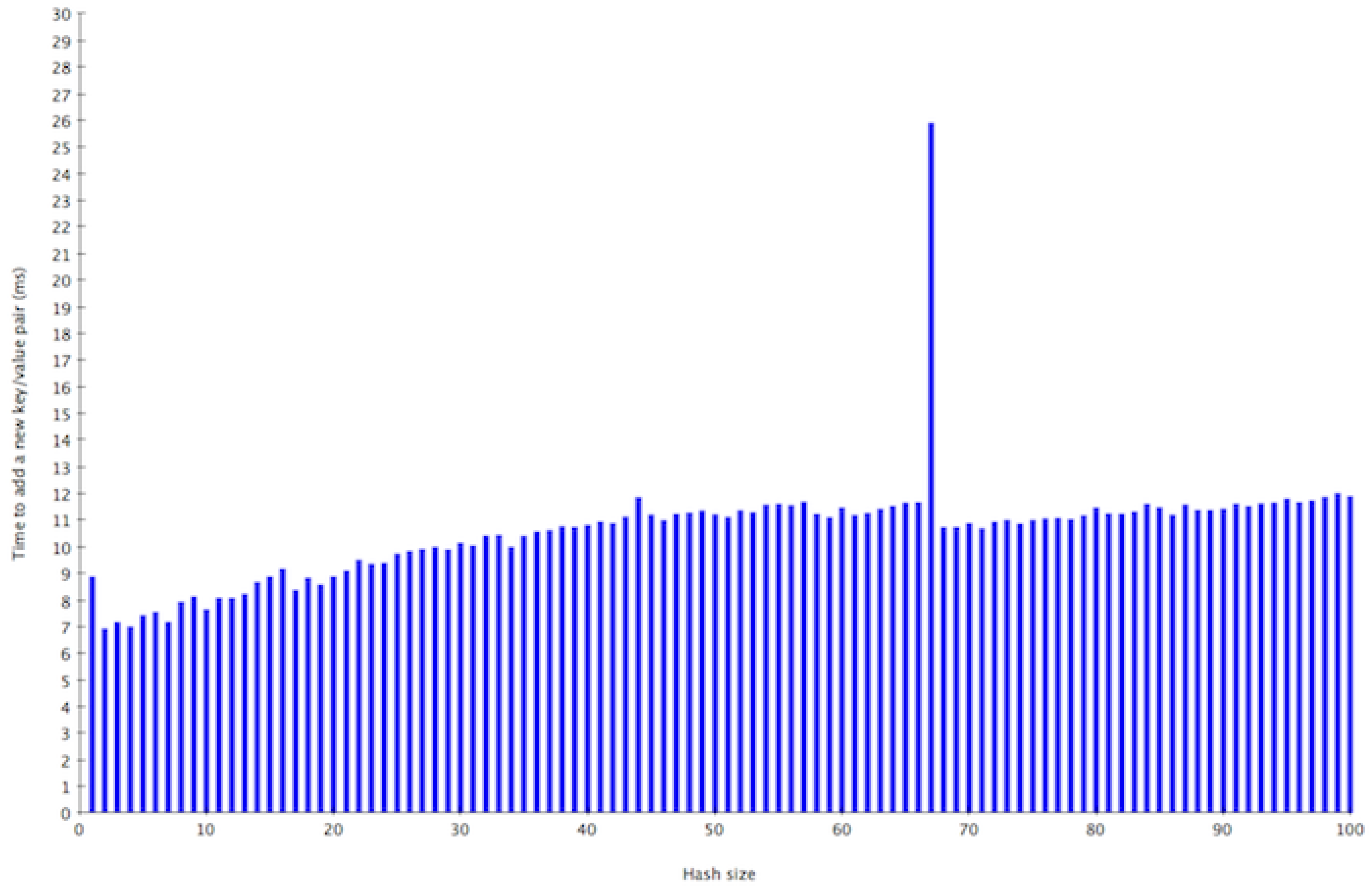
```
hashes = []
10000.times do
  hash = {}
  (1..size).each do |x|
    hash[rand] = rand
  end
  hashes << hash
end
```

Once these are all setup, I can measure how long it takes to add one more element to each hash - element number size+1:

```
Benchmark.bm do |bench|
  bench.report("adding element number #{size+1}") do
    10000.times do |n|
      hashes[n][size] = rand
    end
  end
end
```

The results: inserting the 67th element takes much more time!

What I found was surprising! Here's the data for Ruby 1.8:

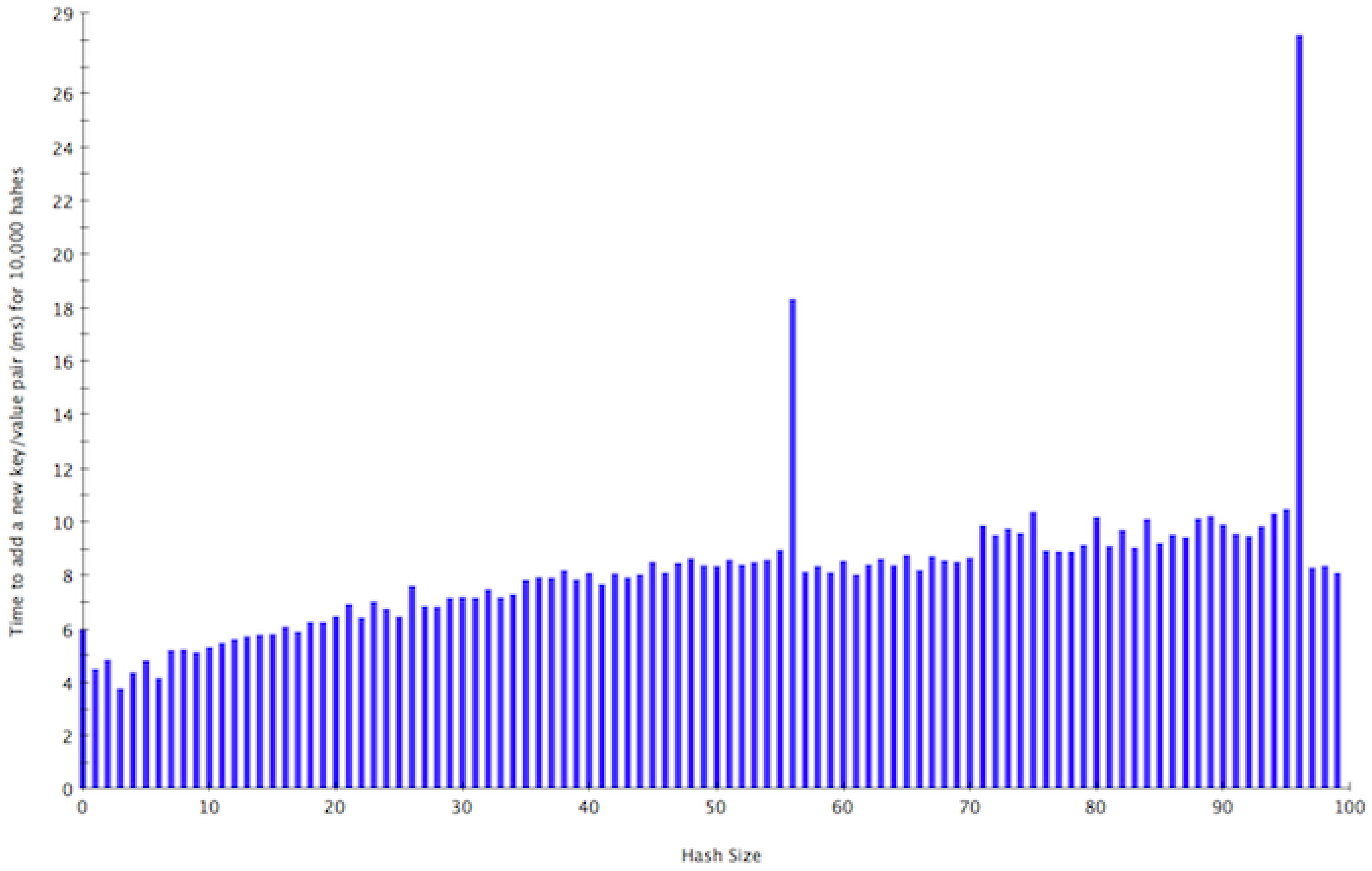


Interpreting these data values from left to right:

- It takes about 9ms to insert the first element into an empty hash (10000 times).
- Then it takes about 7ms to insert the second element into a hash containing one value (10000 times).
- Then as the hash size increases from 2, 3, up to about 60 or 65 the amount of time required to insert a new element slowly increases.
- Finally we see it takes around 11ms or 12ms to insert each new key/value pair into a hash that contains 64, 65 or 66 elements (10000 times).
- Then we see a huge spike! Inserting the 67th key/value pair takes over twice as much time: about 26ms instead of 11ms for 10000 hashes!
- Finally after inserting the 67th element, the time required to insert additional elements drops to about 10ms or 11ms, and then slowly increases again from there.

What’s going on here? Well, the extra time required to insert that 67th key/value pair is spent by Ruby reallocating the bin array from 11 bins to 19 bins, and then reassigning the st_table_entry structures to the new bin array.

Here’s the same graph for Ruby 1.9 - you can see this time the bin density threshold is different. Instead of taking extra time to reallocate the elements into bins on the 67th insert, Ruby 1.9 does it when the 57th element is inserted. Later you can see Ruby 1.9 performs another reallocation after the 97th element is inserted.



If you’re wondering where these magic numbers come from, 57, 97, etc., then take a look at the top of the “st.c” code file for your version of Ruby. You should find a list of prime numbers like this:

```
/*
Table of prime numbers 2^n+a, 2<=n<=30.
*/
static const unsigned int primes[] = {
    8 + 3,
    16 + 3,
    32 + 5,
    64 + 3,
    128 + 3,
    256 + 27,
    512 + 9,
...etc...
```

This C array lists some prime numbers that occur near powers of two. Peter Moore's hash table code uses this table to decide how many bins to use in the hash table. For example, the first prime number in the list above is 11, which is why Ruby hash tables start with 11 bins. Later as the number of elements increases the number of bins is increased to 19, and later still to 37, etc.

Ruby always sets the number of hash table bins to be a prime number to make it more likely that the hash values will be evenly distributed among the bins, after calculating the modulus - after dividing by the prime number and using the remainder. Mathematically, prime numbers help here since they are less likely to share a common factor with the hash value integers, in case a poor hash function often returned values that were not entirely random. If the hash values and bin counts shared a factor, or if the hash values were a multiple of the bin count, then the modulus might always be the same… leading to table entries being unevenly distributed among the bins.

Elsewhere in the st.c file, you should be able to find this C constant:

```
#define ST_DEFAULT_MAX_DENSITY 5
```

… which defines the maximum allowed density, or average number of elements per bin. Finally, you should also be able to find the code that decides when to perform a bin reallocation by searching for where that ST_DEFAULT_MAX_DENSITY constant is used in st.c. For Ruby 1.8 you'll find this code:

```
 if (table->num_entries/(table->num_bins) > ST_DEFAULT_MAX_DENSITY) {
   rehash(table);
```

So Ruby 1.8 rehashes from 11 to 19 bins when the num_entries/11 is greater than 5… i.e. when it equals 66… when you insert the 67th element.

For Ruby 1.9 and Ruby 2.0 you'll find this code instead:

```
 if ((table)->num_entries > ST_DEFAULT_MAX_DENSITY * (table)->num_bins) {
   rehash(table);
```

You can see Ruby 1.9 rehashes for the first time when num_entries is greater than 5*11, or when you insert the 57th element.

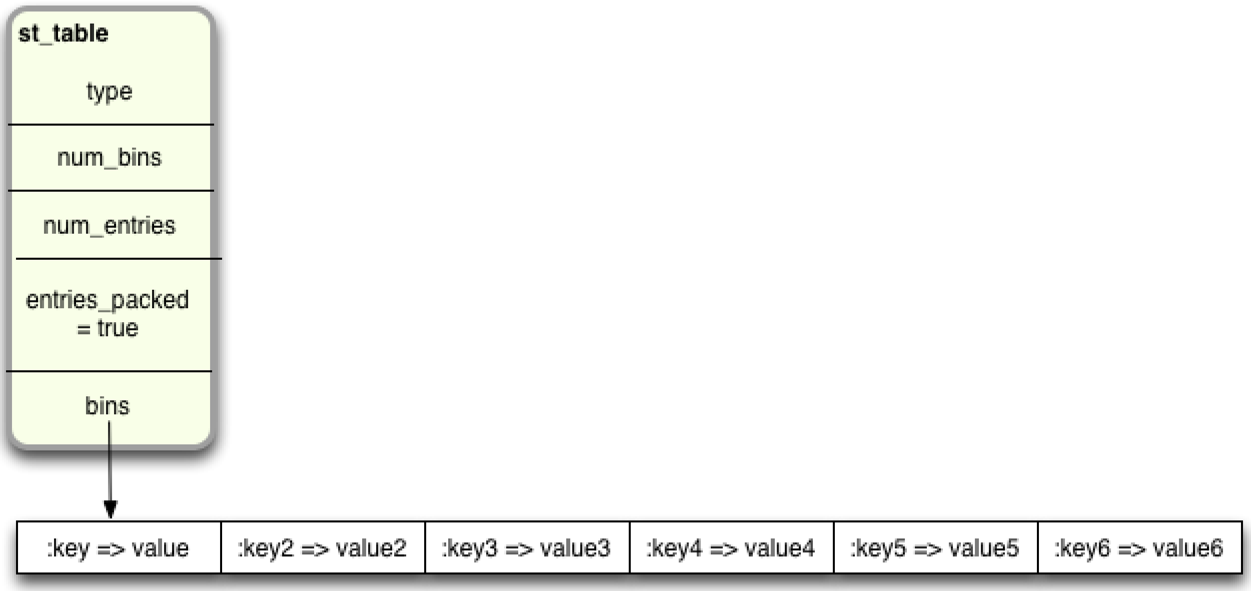
$$\oint \vec{E}\cdot d\vec{A} = \frac{q}{\varepsilon_0}$$

$$\oint \vec{B}\cdot d\vec{A} = 0$$

Theory: Why Hashes will be faster in Ruby 2.0

Above I showed graphs for Ruby 1.8 and Ruby 1.9 - but what about Ruby 2.0? Does it work any differently? Does it reallocate bins in the same way? While reading the Ruby 2.0 source code I noticed a lot of code changes related to allocating bins, which made me suspect that Hashes will work even faster in Ruby 2.0. Let's take a look….

Hashes in Ruby 2.0 will use somewhat different data structures to save keys and values in a hash table. Instead of allocating bins and then assigning the key/value entries (st_table_entry structures) to the bins, Ruby 2.0 will instead save the key/value data right inside the memory normally allocated for the bins. Here's what this looks like:



In the MRI source code these entries are referred to as "packed." Here you can see that the keys and values are saved right inside the bins on the lower right, and that the st_table has a new value in it called "entries_packed." This value is set to true whenever the keys and values are saved like this. When this is false, it indicates that the data elements are saved in st_table_entry structures as usual.

But wait a minute! If there are no buckets, then this isn't a hash table at all, is it? That's right: in Ruby 2.0 the Hash object is actually implemented as an array internally, and not as a hash table!

However, this is only true for small hashes - hashes whose elements all fit into the memory space normally used for the bin array. Once a hash's elements no longer fit into the bin array memory, the key/value data will be copied back into new st_table_entry structures and assigned to the hash table bins as usual.

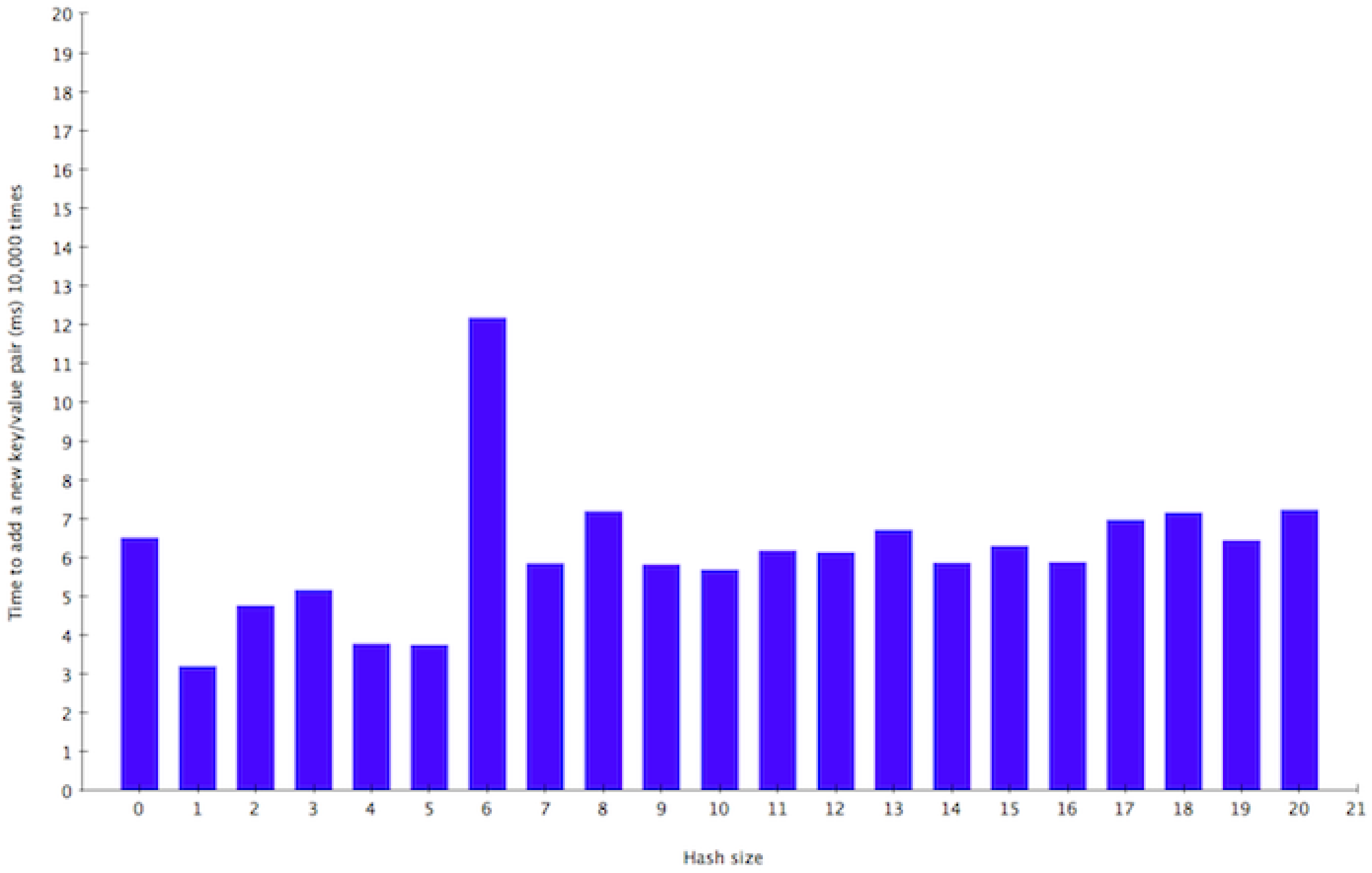
The optimization here really isn't about saving memory as much as time/speed. In Ruby 2.0 the Ruby core team has decided that for small hashes it's actually faster just to save the keys and values in an array, and to look for the target key by simply iterating through the array. In this way, Ruby avoids calling the hash function entirely for the target key, which should be fast, but could possibly be slow if the key was a large string or array. Ruby also saves time by avoiding the need to create and setup the st_table_entry structures when you insert elements.

Experiment 3: Inserting one new element into hashes of varying sizes, for Ruby 2.0

For this test I'll re-run the same code I used in Experiment #2, but this time using "ruby-head" build; the code from the master branch intended for an upcoming release of Ruby 2.0. And I'll focus on the data for smaller hash sizes, to see if I can find any evidence of this optimization.

The results: inserting the first six elements is faster!

Here are the results:



Again here the y-axis shows the amount of time required to insert one new element into an existing hash. Along the x-axis I measure this time for hashes of different sizes, starting with zero (an empty hash). Here's how I interpret the results:

Here's how I interpret the results:

- It takes about 6ms to insert the first element, 10,000 times.
- Then, it takes only about 3ms to insert the second element, 10,000 times.
- After that, it takes Ruby 2.0 between about 4ms or 5ms to insert elements number 3, 4, 5 and 6.
- Then we see a spike: to insert the seventh element (10,000 times) Ruby requires about 12ms.
- Finally, after that, inserting new elements takes on the average about 6ms to 7ms.

What does this mean?

- Inserting the first element takes a bit longer, but in Ruby 2.0 the bin array is actually not created at all for empty hashes. You can view this as another optimization: empty hashes have an RHash structure and an st_table structure, but no bin array at all. This allows Ruby to create new, empty hashes faster at the price of requiring a bit more time to insert the first element.
- Then inserting elements up to #6 is very fast, since they are saved directly into the bin array, and Ruby 2.0 doesn't need to allocate new st_table_entry structures or call the hash function at all.
- But only 6 elements fit into the array. Therefore when you insert the seventh element Ruby 2.0 has to create 7 st_table_entry structures, call the hash function for each key and assign them to the bins. In other words, Ruby 2.0 has to setup the hash table for the first time when you insert the seventh element!
- After that things work the same way they do for Ruby 1.9.

If you're interested in learning more about how Ruby 2.0's new hash-as-array optimization works, then search for the word "packed" in the st.c file. For example, in Ruby 2.0 the st_table structure contains a value called "entries_packed" which indicates whether or not the hash is really an array. There is also a function called "add_packed_direct" which inserts a new key/value into the hash array (and not in the hash table). Finally if you search for a constant called MAX_PACKED_HASH you'll find the calculation that determines that 6 hash elements should be saved as an array, while 7 or more hash elements should be saved in a hash table.

One final note here: this might seem like an unnecessary and unimportant optimization. After all, once you have 7 elements in a hash the optimization no longer applies. But think about how often Ruby developers use hashes to save options for method calls and in other ways that only require a few key/

value pairs be stored in the hash. Small hashes are used quite frequently in Ruby applications - most hashes are small hashes - and therefore this optimization will have a big impact.

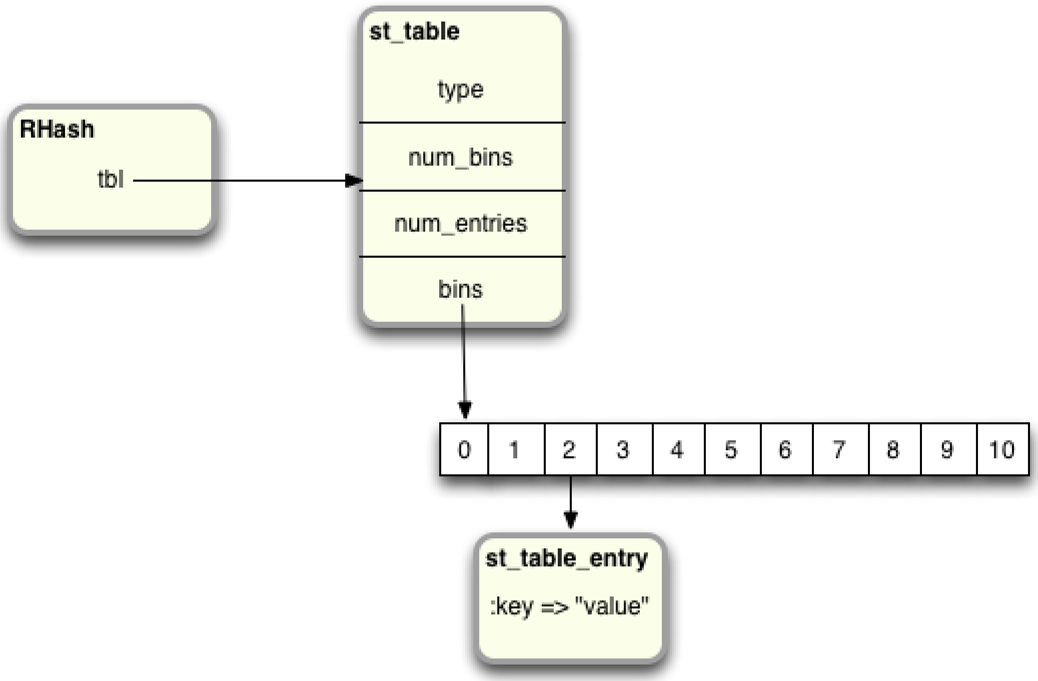
$$\oint \vec{E} \cdot d\vec{s} = -\frac{d\Phi_B}{dt}$$

$$\oint \vec{B} \cdot d\vec{s} = \mu_0 i + \frac{1}{c^2}\frac{\partial}{\partial t}\int \vec{E} \cdot d\vec{A}$$

Theory: How Ruby implements hash functions

Now let's take a closer look at the actual hash function Ruby uses to assign keys and values to bins in hash tables. If you think about it, this function is central to the way the Hash object is implemented - if this function works well then Ruby hashes will be fast, but a poor hash function would in theory cause severe performance problems. And not only that, as I mentioned above, Ruby uses hash tables internally to store its own information, and not only the data values you save in hash objects. Clearly having a good hash function is very important!

First let's review again how Ruby uses hash values. Remember that when you save a new element - a new key/value pair - in a hash, Ruby assigns it to a bin inside the internal hash table used by that hash object:



Again, the way this works is that Ruby calculates the modulus of the key's hash value by the number of bins:

```
bin index = internal_hash_function(key) % bin count
```

Or in this example:

```
2 = hash(:key) % 11
```

The reason this works well for Ruby is that Ruby's hash values are more or less random integers for any given input data. You can get a feel for how Ruby's hash function works by calling the "hash" method for any object like this:

```
$ irb
ruby-1.9.3-p0 :001 > "abc".hash
 => 3277525029751053763
ruby-1.9.3-p0 :002 > "abd".hash
 => 234577060685640459
ruby-1.9.3-p0 :003 > 1.hash
 => -3466223919964109258
ruby-1.9.3-p0 :004 > 2.hash
 => -2297524640777648528
```

Here even similar values have very different hash values. Note that if I call "hash" again I always get the same integer value for the same input data:

```
ruby-1.9.3-p0 :001 > "abc".hash
 => 3277525029751053763
ruby-1.9.3-p0 :002 > "abd".hash
 => 234577060685640459
```

Here's how Ruby's hash function actually works for most Ruby objects:

- When you call "hash" Ruby finds the default implementation in the "Object" class. You, of course, are free to override this if you really want to.
- The C code used by the Object class's implementation of the hash method gets the C pointer value for the target object - i.e. the actual memory address of that object's RValue structure. This is essentially a unique id for that object.
- Ruby then passes it through a complex C function - the hash function - that mixes up and scrambles the bits in the value, producing a pseudo-random integer in a repeatable way.

For string and arrays it works differently. In this case, Ruby actually iterates through all of the characters in the string or elements in the array and calculates a cumulative hash value; this guarantees that the hash value will always be the same for any instance of a string or array, and will always change if any of the values in that string or array change.

Finally, integers and symbols are another special case - for them Ruby just passes their values right to the hash function.

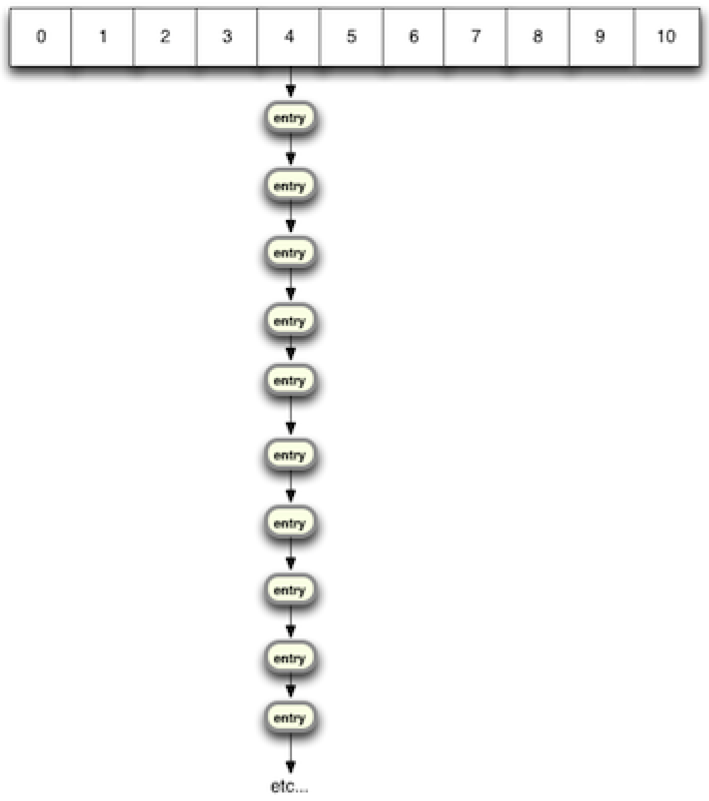
Ruby 1.9 and 2.0 actually use something called the "MurmurHash" hash function, which was invented by Austin Appleby in 2008. The name "Murmur" comes from the machine language operations used in the algorithm: "multiply" and "rotate." If you're interested in the details of how the Murmur algorithm actually works, you can find the C code for it in the st.c Ruby source code file, around line 1028. Or you can read Austin's web page on Murmur: http://sites.google.com/site/murmurhash/.

Also, Ruby 1.9 and Ruby 2.0 initialize MurmurHash using a random seed value which is reinitialized each time you restart Ruby. This means that if you stop and restart Ruby you'll get different hash values for the same input data. It also means if you try this yourself you'll get different values than I did above. However, the hash values will always be the same within the same Ruby process.

Since hash values are pseudo-random numbers, once Ruby divides them by the bin count, e.g. 11, the remainder values left over (the modulus values) will be a random number between 0 and 10. This means that the st_table_entry structures will be evenly distributed over the available bins as they are saved in the hash table. Evenly distributing the entries ensures that Ruby will be able to quickly search for and find any given key, since the number of entries per bin will always be small. (On the average it will always be less than the maximum density of 5, which I showed earlier.)

But imagine if Ruby's hash function didn't return random integers - imagine if instead it returned the same integer for every input data value. What would happen?

In that case, every time you added any key/value to a hash it would always be assigned to the same bin. Then Ruby would end up with all of the entries in a single, long list under that one bin, and with no entries in any other bin:



Now when you tried to retrieve some value from this hash, Ruby would have to look through this long list, one element at a time, trying to find the requested key. In this scenario loading a value from a Ruby hash would be very, very slow.

Experiment 4: Using objects as keys in a hash

Now I'm going to prove this is the case - and illustrate just how important Ruby's hash function really is - by using objects with a poor hash function as keys in a hash. Let's repeat Experiment 1 and create many hashes that have different numbers of elements, from 1 to a million:

```
size = 2**exponent
hash = {}
(1..size).each do |n|
  index = rand
  hash[index] = rand
end
```

But instead of calling "rand" to calculate a random key values, this time I'll create a new, custom object class called "KeyObject" and use instances of that class as my key values:

```
class KeyObject
end

size = 2**exponent
hash = {}
(1..size).each do |n|
  index = KeyObject.new
  hash[index] = rand
end
```

This works essentially the same way as Experiment 1 did, except that Ruby will have to calculate the hash value for each of these "KeyObject" objects instead of the random floating point values I used earlier.

After re-running the test with this KeyObject class, I'll then proceed to change the KeyObject class and override the "hash" method, like this:

```
class KeyObject
  def hash
    4
  end
end
```

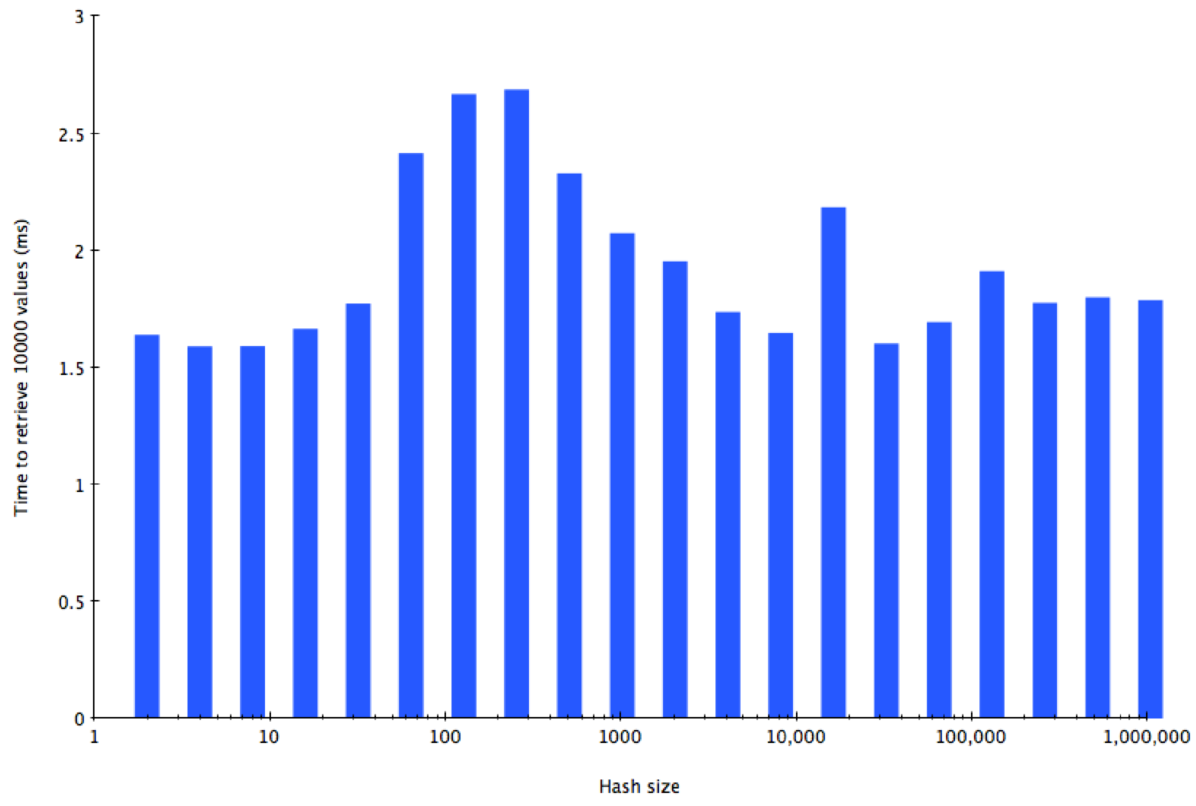
I’ve purposefully written a very poor hash function - instead of returning a pseudo-random integer, this hash function always returns the integer 4, regardless of which KeyObject object instance you call it on. Now Ruby will always get 4 when it calculates the hash value, and it will have to assign all of the hash elements to bin #4 in the internal hash table, like in the diagram above. Let’s see what happens….

The results: a poor hash function has a dramatic effect on performance!

Running the test with an empty “KeyObject” class:

```
class KeyObject
end
```

… I get results similar to Experiment 1:

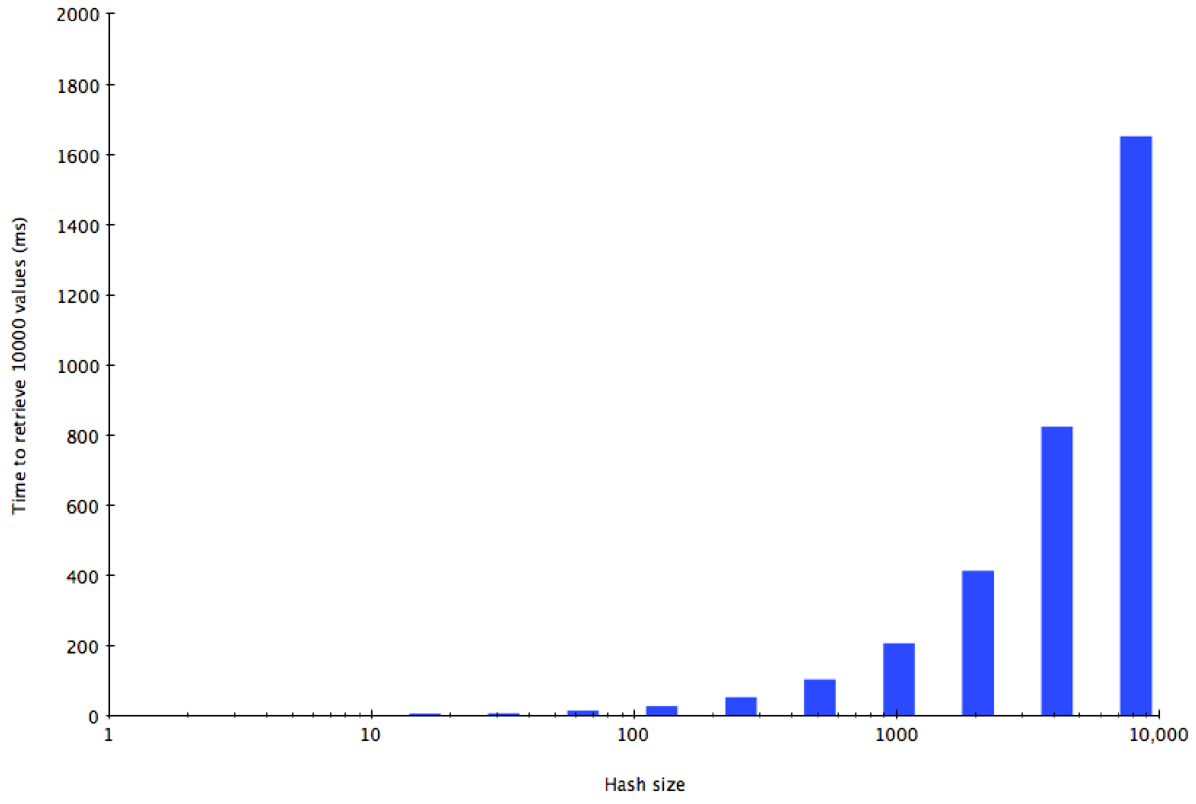


Using Ruby 1.9 I again see that Ruby takes about 1.5ms to 2ms to retrieve 10,000 elements from a hash, this time using instances of the KeyObject class as the keys.

Now let's run the same code, but this time with the poor hash function in KeyObject:

```
class KeyObject
  def hash
    4
  end
end
```

Here are the results:



Wow - very different! Pay close attention to the scale of the graph. On the y-axis I show milliseconds and on the x-axis again the number of elements in the hash, shown on a logarithmic scale. But this time notice that I have 1000s of milliseconds - or actual seconds - on the y-axis! With 1 or a small number of elements, I can retrieve the 10,000 values very quickly - so quickly that the time is too small to appear on this graph. In fact it takes about the same 1.5ms time.

But when the number of elements increases past 100 and especially 1000, the time required to load the 10,000 values increases linearly with the hash size. For a hash containing about 10,000 elements it takes over 1.6 full seconds to load the 10,000 values. If I continue the test with larger hashes it would take minutes or even hours to load the values.

Again what's happening here is that all of the hash elements are saved into the same bin, forcing Ruby to search through the list one key at a time.

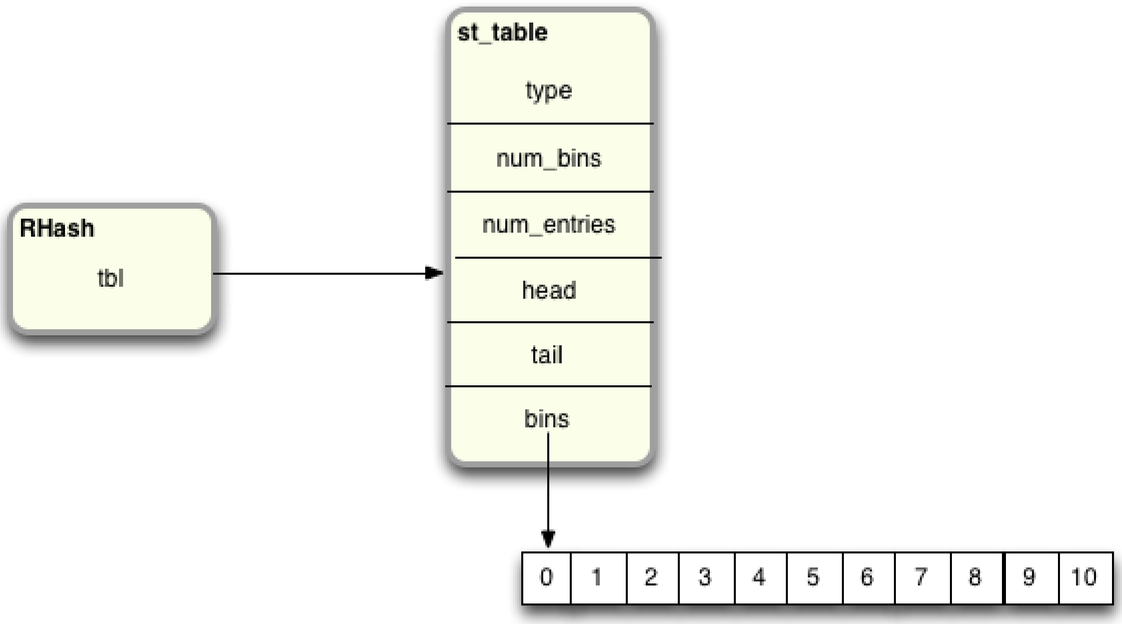
$$\nabla \cdot \vec{E} = \frac{\rho}{\varepsilon_0} = 4\pi k \rho$$

$$\nabla \cdot \vec{B} = 0$$

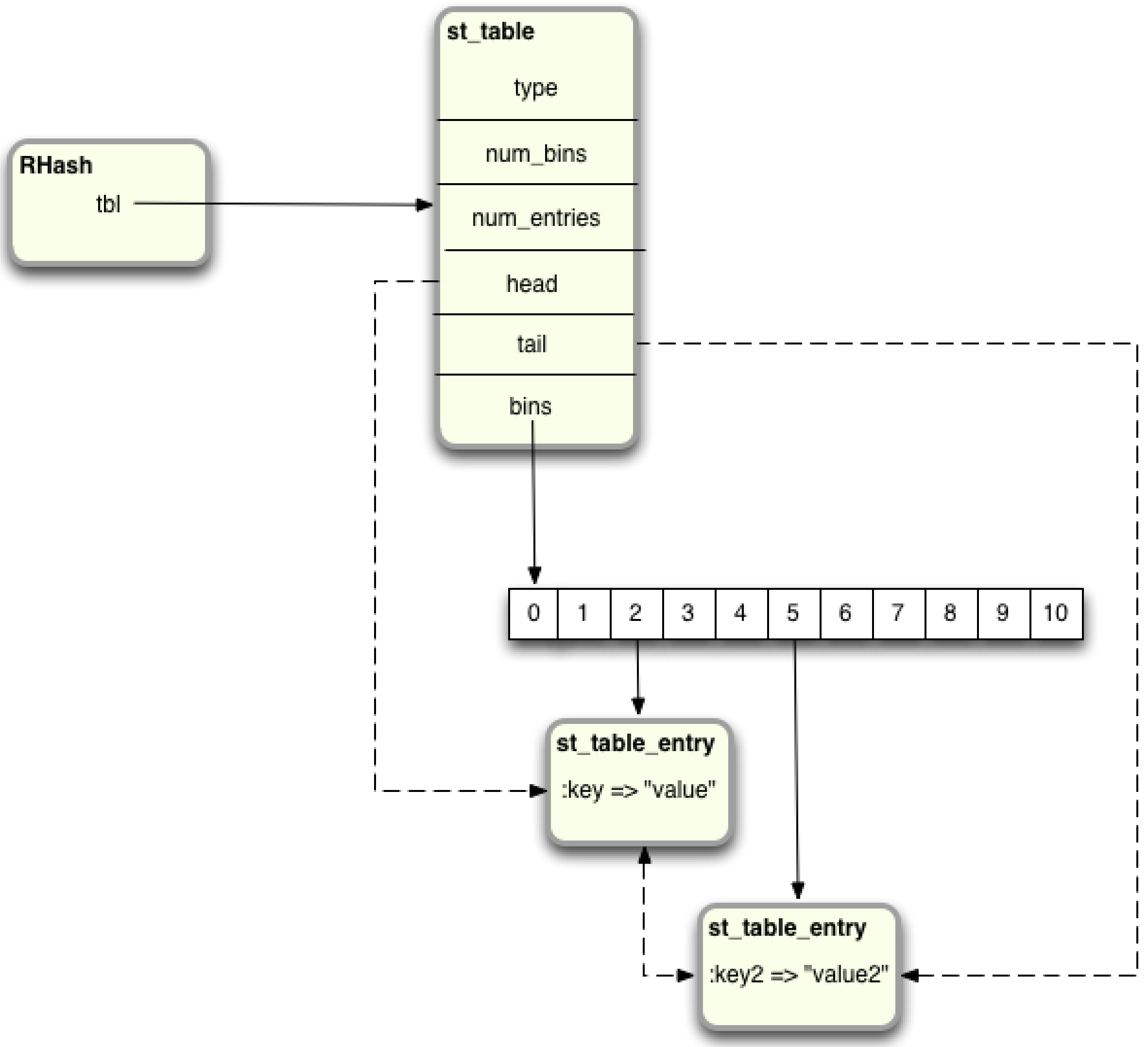
Theory: How Ruby saves order information in hashes

There’s one subtle but interesting detail about my diagrams so far you might not have noticed. If you look, there’s no information saved in the hash table about what order I saved the keys and values in. That is, looking at the hash table diagrams, how would Ruby know whether I inserted :key first, or :key2?

Aside from the Ruby 2.0 hash as array optimization, one other major code change I noticed between the different versions of MRI Ruby had to do with saving order information in the hash table. Investigating a bit, I began to learn that Ruby 1.8 and Ruby 1.9 behave differently with regard to order and iterating over hash elements. I discovered that in Ruby 1.9 and 2.0 the st_table structure contains two additional data values called “head” and “tail:”



The head and tail values are pointers to st_table_entry structures, forming a linked list that tracks the order entries were saved into the hash. In my previous example, the linked list would look like this:



The dashed lines indicate the linked list. When the first value is added to the hash table, the “head” and “tail” pointers are set to point to it. Then when the second value is added, a linked list is formed from the head to the first value and from the first to the second value. The tail pointer is then set to the second value. I’m not showing them here, but there are also new pointers added to the st_table_entry structure to support the linked list as well.

Why do all of this extra work? The linked list allows Ruby 1.9 and Ruby 2.0 to record the order you added values to the hash, and then to use that information to return the values back to you in the same order. That is, if I call one of the iterator related methods, such as “each” or “each_value” or “each_key,” etc., Ruby 1.9 and Ruby 2.0 will look for the “head” pointer in the st_table structure, and then start iterating through the st_table_entry structures using the linked list pointers.

Ruby 1.8, however, will simply iterate over the bins in the bin array, and then through the st_table_entry structures in the order it happens to find them in the hash table. While this order is not random - it has to do with the hash value of each key - since the hash values appear to be random values the order you get the elements back while iterating will appear to be random also.

Experiment 5: Iterating over elements inserted into a Hash

For this test I just need to create an empty hash:

```
hash = {}
```

… and then insert two values into it:

```
hash['one'] = "This should be returned first"
hash['two'] = "This should be returned second"
```

… and finally if I iterate over the values I can see what order they are returned in:

```
hash.each_value { |val| puts val }
```

First, running the test using Ruby 1.8:

```
$ ruby -v
ruby 1.8.7 (2011-06-30 patchlevel 352) [i686-darwin11.2.0]
$ ruby experiment5.rb
This should be returned second
This should be returned first
```

You can see here that actually these two elements are returned in the *wrong* order: first the value for :two is returned and then the value for :one. It turns out for Ruby 1.8 the hash values for these two keys, 'one' and 'two', happen to occur in the wrong order.

Now let’s re-run the same code for Ruby 1.9 (or Ruby 2.0):

```
$ ruby -v
ruby 1.9.3p0 (2011-10-30 revision 33570) [x86_64-darwin11.2.0]
$ ruby experiment5.rb
This should be returned first
This should be returned second
```

This time I get the values in the proper order.

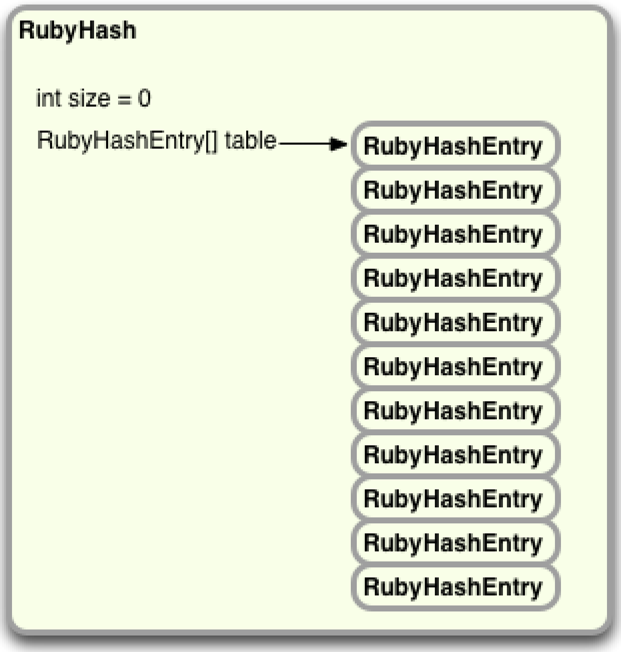
Remember that for the Hash object iterating through the keys in order is a secondary feature. The most important feature of a Hash is to be able to quickly retrieve a value for any given key. As I discussed above, this is done by calculating the hash value and the modulus by the number of bins, and not by using the head/tail linked list. Ruby only uses the linked list when you call “each” or one of the other iterator methods. Ruby also uses the list when executing the “shift” method.

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t}$$

$$\nabla \times \vec{B} = \frac{\vec{J}}{\varepsilon_0 c^2} + \frac{1}{c^2}\frac{\partial \vec{E}}{\partial t}$$

Alternate theories: Hashes in JRuby

It turns out JRuby implements hashes more or less the same way MRI Ruby does. Of course, the JRuby source code is written in Java and not C, but the JRuby team chose to use the same underlying hash table algorithm that MRI uses. Since Java is an object oriented language, unlike C, JRuby is able to use actual Java objects to represent the hash table and hash table entries, instead of memory structures. Here’s what a hash table looks like internally inside of a JRuby process:

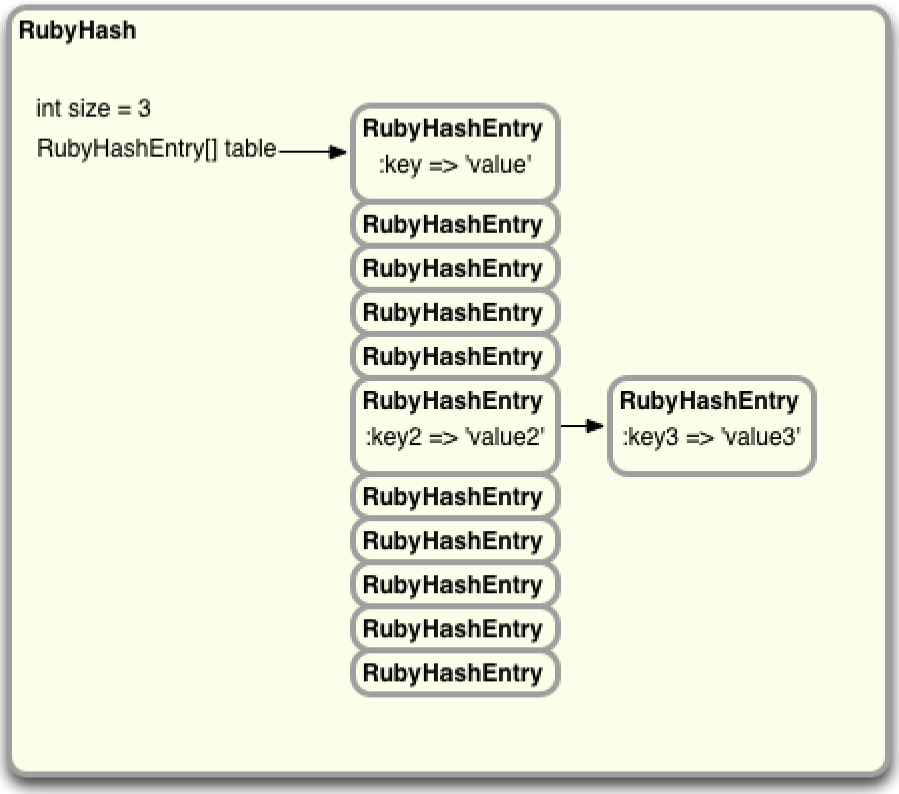


Here instead of the C RHash and st_table memory structures, we have a Java object called “RubyHash”. And instead of the bin array and st_table_entry structures we have an array of Java objects called “RubyHashEntry.” The RubyHash object contains an instance variable called “size” which keeps track of the number of elements in the hash, and another instance variable called “table”, which is the RubyHashEntry array.

JRuby allocates 11 empty RubyHashEntry objects when you create a new hash; these form the hash table bins. Then later as you insert elements into the hash, JRuby fills in these objects them with keys and values. Inserting and retrieving elements works the same was as in MRI: JRuby uses the same formula to divide the hash value of the key by the bin count, and uses the modulus to find the proper bin:

```
bin index = internal_hash_function(key) % bin count
```

As you add more and more elements to the hash, JRuby forms a linked list of RubyHashEntry objects as necessary when two keys fall into the same bin - just like MRI:



And JRuby also tracks the density of entries - the average number of RubyHashEntry objects per bin - and allocates a larger table of RubyHashEntry objects as necessary, rehashing the entries.

If you're interested, you can find the Java code JRuby uses to implement hashes in the src/org/jruby/RubyHash.java source code file. I found it easier to understand than the original C code from MRI, mostly because in general Java is a bit more readable and easier to understand than C is, and because it's object oriented. The JRuby team was able to separate the hash code into different Java classes, primarily RubyHash and RubyHashEntry.

The JRuby team even used the same identifier names as MRI in some cases; for example you'll find the same "ST_DEFAULT_MAX_DENSITY" value of 5, and JRuby uses the same table of prime numbers that MRI does: 11, 19, 37, etc., that fall near powers of two. This means that JRuby will show the same performance pattern MRI does for reallocating bins and redistributing the entries.

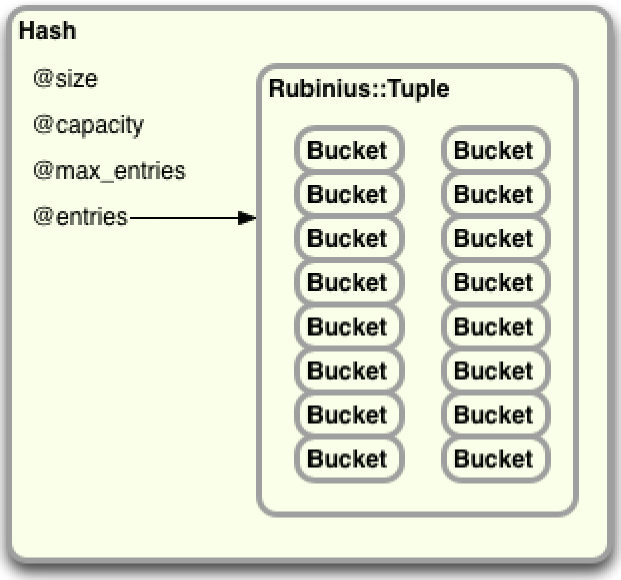
$$\oint \vec{E} \cdot d\vec{A} = \frac{q}{\varepsilon_0}$$

$$\oint \vec{B} \cdot d\vec{A} = 0$$

Alternate theories: Hashes in Rubinius

At a high level, Rubinius uses the same hash table algorithm as MRI and JRuby - but using Ruby instead of C or Java. This means the Rubinius source code is about 10 times easier to understand than either the MRI or JRuby code, and is a great way to learn more about hash tables if you're interested in getting your hands dirty without learning C or Java.

Here's how hashes look inside of Rubinius:



Since this is just plain Ruby, in Rubinius your Ruby objects are actually implemented with a real Ruby class called "Hash". You'll see it has a few integer attributes, such as @size, @capacity and @max_entries, and also an instance variable called @entries which is the bin array that actually contains the hash data. Rubinius implements the bin array using a Ruby class called "Rubinius::Tuple", which is a simple storage class similar to an array. Rubinius saves each hash element inside a Ruby object called "Bucket", saved inside of the @entries Rubinius::Tuple array.

One difference you'll see in the Rubinius hash table implementation is that it uses simple powers of two to decide how many hash bins to create, instead of prime numbers. Initially Rubinius uses 16 Bucket objects. Whenever Rubinius needs to allocate more bins, it just doubles the size of the bin array - "@entries" in the code above. While theoretically this is less ideal than using prime numbers, it simplifies the code substantially and also allows Rubinius to use bitwise arithmetic to calculate the bin index, instead of having to divide and take the remainder/modulus.

You'll find the Rubinius hash implementation in source code files called kernel/common/hash18.rb and kernel/common/hash19.rb - Rubinius has entirely different implementations of hashes depending on whether you start in Ruby 1.8 or Ruby 1.9 compatibility mode. Here's a snippet from hash18.rb, showing how Rubinius finds a value given a key:

```
def [](key)
  if item = find_item(key)
    item.value
  else
    default key
  end
end

... etc ...

# Searches for an item matching +key+. Returns the item
# if found. Otherwise returns +nil+.
def find_item(key)
  key_hash = key.hash

  item = @entries[key_index(key_hash)]
  while item
    if item.match? key, key_hash
      return item
    end
    item = item.link
  end
end

... etc ...

# Calculates the +@entries+ slot given a key_hash value.
```

```
def key_index(key_hash)
  key_hash & @mask
end
```

You can see the key_index method uses bitwise arithmetic to calculate the bin index, since the bin count will always be a power of two for Rubinius, and not a prime number. Trust me, this code is *much* easier to understand than the corresponding C code in MRI Ruby.

Conclusion

The more I look at the Ruby Hash object, the more I'm impressed. On the surface it seems to be very obvious and straightforward: you insert values and keys, and later you can get them back again. What could be simpler? But looking into the details of how Ruby implements hashes lead me to discover a great deal of knowledge:

- First, I learned about hash tables and hash functions: what they are and how they work.
- Then, I realized how scalable Ruby hashes really are. I can now write Ruby code that saves a large data set into a hash, while being confident that I will later be able to quickly and efficiently search for any given object.
- I saw how the Ruby core team has improved - and is still improving - how the Ruby Hash object works. I can't wait to see what they come up next!
- But most importantly, it was a lot of fun!

Appendix - Experiment Code

Experiment 1

Find this code on Github.

```
require 'benchmark'
ITERATIONS = 10000
(1..20).each do |exponent|

  size = 2**exponent
  hash = {}
  target_index = 0
  (1..size).each do |n|
    index = rand
    hash[index] = rand
    target_index = index if n == size/2
  end

  GC.start
  Benchmark.bm do |bench|
    bench.report("retrieving an element from a hash with #{size} elements #{ITERATIONS} times") do
      ITERATIONS.times do |n|
        val = hash[target_index]
      end
    end
  end
end
```

Experiments 2 and 3

Find this code on Github.

```
ITERATIONS = 10000
(0..99).each do |size|
  puts "Creating #{ITERATIONS} hashes with #{size} elements."
  hashes = []
  ITERATIONS.times do
    hash = {}
    (1..size).each do |x|
      hash[rand] = rand
    end
    hashes << hash
  end

  require 'benchmark'
  GC.start
  Benchmark.bm do |bench|
    bench.report("adding element number #{size+1}") do
      ITERATIONS.times do |n|
        hashes[n][size] = rand
      end
    end
  end
end
```

Experiment 4

Find this code on Github.

```
require 'benchmark'
ITERATIONS = 10000

class KeyObject
  def hash
    4
  end
end

(1..20).each do |exponent|

  size = 2**exponent
  hash = {}
  target_index = 0
  (1..size).each do |n|
    hash_index = KeyObject.new
    hash[hash_index] = rand
    target_index = hash_index if n == size/2
  end

  GC.start
  Benchmark.bm do |bench|
    bench.report("retrieving an element from a hash with #{size} elements #{ITERATIONS} times") do
      ITERATIONS.times do |n|
        val = hash[target_index]
      end
    end
  end
end
```