A 10-MINUTE DESCRIPTION OF HOW JUDY WORKS AND WHY IT IS SO FAST

By Doug Baskins, doug@fc.hp.com
October 16, 2001

As the inventor of the Judy algorithm I've been asked repeatedly, "What makes Judy so fast?" The answer is not simple, but finally I can share some of the details. Let's see if I can give you a good understanding in 10 minutes. (The Judy data structures are very well described in another paper, the *Judy Shop Manual*, but it took me about three hours to read!)

A Judy tree is generally faster than and uses less memory than contemporary forms of trees such as binary (AVL) trees, b-trees, and skip-lists. When used in the "Judy Scalable Hashing" configuration, Judy is generally faster then a hashing method at all populations. (See also http://www.hp.com/go/judy: information library: application notes: scalable hashing)

Expanse, **population**, and **density** are not commonly used terms in tree search literature, so let's define them here:

- **Expanse** is a range of possible keys, such as: 256..511

- **Population** is the count of keys contained in an expanse, such as, 260, 499, 500 = 3.

- **Density** is used to describe the sparseness of an expanse of keys; density = population / expanse. A density of 1.0 means that all possible keys are set or valid in that expanse.

Node and **branch** are used interchangeably in this document.

Key and **index** are used interchangeably. A Judy tree is thought of as an unbounded Judy array at the API level. The expanse of JudyL or Judy1 arrays are bounded by the expanse of the word (32[64]-bits) used for the index/key. A JudySL array is only bounded by the length of the key string that can be stored in the machine.

A (CPU) **cache-line fill** is additional time required to do a read reference from RAM, when a word is not found in cache. In today's computers the time for a cache-line fill is in the range of 50..200 machine instructions. Therefore a cache-line fill should be avoided when fewer than 50 instructions can do the same job. (Modern machines tend to pipeline writes to RAM. They often take no additional time in the Judy design.)

Some of the reasons Judy outperforms binary trees, b-trees, and skip-lists:

- Judy rarely compromises speed/space performance for simplicity (Judy will never be called simple except at the API).

- Judy is designed to avoid cache-line fills wherever possible. (This is the main design criteria for Judy.)

- A b-tree requires a search of each node (branch), resulting in more cache-line fills.

- A binary-tree has many more levels (about 8X), resulting in more cache-line fills.

- A skip-list is roughly equivalent to a degree-4 (4-ary) tree, resulting in more cache-line fills.

- An “expanse”-based digital tree (of which Judy is a variation) never needs balancing as it grows.

- A portion (8 bits) of the key is used to subdivide an expanse into sub-trees. Only the remainder of the key need exist in the sub-trees, if at all, resulting in key compression.

The Achilles heel of a simple digital tree is very poor memory utilization, especially when the N in N-ary (the degree or fanout of each branch) increases. The Judy tree design was able to solve this problem. In fact a Judy tree is more memory-efficient than almost any other competitive structure (including a simple linked list). A highly populated linear array[] is the notable exception.

From a speed point of view Judy is chiefly a 256-ary digital tree or trie (per D. Knuth Volume 3 definitions). A degree of 256-ary is a somewhat “magic” N-ary for a variety of reasons -- mostly because a byte (the least addressable memory unit) is 8 bits. Also a higher degree means reduced cache-line fills per access. You see the theme here -- avoid cache-line fills like the plague.

It is interesting to note that an early version of Judy used branch widening (sometimes called a level-compressed tree). Branch widening opportunities occur primarily in the upper level(s) of the tree. Since a tree is a hierarchy, the upper branches are likely to be in cache, thus branch widening did not significantly reduce the number of actual cache-line fills. Branch widening was removed in later versions of Judy. (However, Judy was also tuned to use as few instructions as possible when an access was likely to be in the cache.)

The presence of a CPU cache in modern machines has changed many of the ways to write a performance algorithm. To take advantage of a cache, it is important to leverage as much as possible. In a Judy tree, the presence of a cache results in 1..3 (or more) fewer cache-line fills per access than would be possible without a cache.

As a digression, note that a hash method loses the advantages of a cache as the size of the hash table approaches or exceeds the size of the cache. With very large hash tables the cache is no help at all. Also, hash methods often use a linked list to handle collisions (synonyms) and typically use a slow hash algorithm (greater than 50ns) or suffer from numerous collisions. “Judy Scalable Hashing” is an effective replacement for common hash methods. (See also http://www.hp.com/go/judy: information library: application notes: scalable hashing)

With an expanse of 2^{32} (or 256^4), a maximum of 4 cache-line fills would be required for a worst-case highly populated 256-ary digital tree access. In an expanse of 2^{64} (or 256^8), 8 cache-line fills would be the worst case. In practice, Judy does much better than this. The reason is (in part) due to the fact "density" of the keys is seldom the lowest possible number in a "majority" of the sub-expanses. It takes high density combined with high population to increase the depth of a Judy tree. It would take a long time to explain why. The short version is an analogy with sand. It takes a lot of sand to build a tall sand pile. (In a 64-bit Judy, it would probably require more RAM than exists on this planet to get it to have 8 levels.)

Judy adapts efficiently to a wide range of population counts and population types. To support this flexibility, in 32[64]-bit Judy there are approximately 25[85] major data structures and a similar number of minor structures. I am going to only describe a few of them so you can infer how density is synergistic with compression.

From a memory consumption (size) point of view, a Judy tree shares (does not duplicate) common digits of a key in a tree. This form of key compression is a natural outcome from using a digital tree. This would be very awkward to do in trees balanced by population and, as far as I know, has never been done. Each pointer traversed in a Judy tree points to ever smaller sub-expanses, while decoding another 8 bits of the key. (In a pure digital tree, the keys are not stored in the tree, they are inferred by position.)

Now let me try to describe the top of a small Judy tree and the bottom of a highly populated one. A Judy tree with a population of zero is simply a NULL pointer. A Judy (JudyL) tree with a population of one is a root pointer to a 2-word object containing a value and associated key.

A tree with a population of two points to a 4-word object with 2 values and 2 sorted keys. A tree with a population of three points to an 8-word object with a count word + 3 values and 3 sorted keys.

This continues until the population grows to 32 keys. At this point an actual tree structure is formed with a "compressed" 256-ary node (branch) that decodes the first byte of each key. The value 32 was chosen because this is where a tree structure requires an equivalent number of cache-line fills. All objects below this top branch contain keys that are shortened by at least the first byte.

There are three kinds of branches. Two are 1-cache-line fill objects to traverse, and one is a 2-cache-line fill object to traverse. In every path down the tree and at all populations, a maximum of one of the 2-cache-line fill branches is used. This means it is sometimes possible to have 1 additional (the branch design often subtracts 1) cache-line fill than you would expect from a pure 256-ary branch traversal in an otherwise complete Judy tree.

On the other extreme, a highly populated tree where the key has been decoded down to 1 byte, and the density of a 256-wide sub-expanse of keys grows to greater than 0.094 (25 keys / 256 expanse), a bitmap of 32 bytes (256 bits) is formed from an existing sorted array of 24 1-byte keys. (I am leaving out the handling of the values.) This results in a key using about an average of

1.3 (32/25) bytes of memory (up from 1.0). Note that increasing the density (population) at this point does NOT require more memory for keys. For example, when the density reaches 0.5 (population = 128 / expanse = 256), the memory consumed is about 2 bits ((32/128)*8) per key + some overhead (2.0+ words) for the tree structure.

Notice that to insert or delete a key is almost as simple as setting or clearing a bit. Also notice, the memory consumption is almost the same for both 32- and 64-bit Judy trees. Given the same set of keys, both 32- and 64-bit Judy trees have remarkably similar key-memory, depth, and performance. However, the memory consumption for 64-bit Judy is higher because the pointers and values (JudyL) are double the size.

In this short writeup it wasn't possible to describe all the data structure details such as: Root, JPM, narrow and rich pointers, linear, bitmap and uncompressed branches, value areas, immediate indexes, terminal nodes (leafs), least compressed form, memory management, fast leaf searches and counting trees.

Well I cannot describe Judy in 10 minutes -- what possessed me? I hope you understand some of what I have said and question me on the rest -- particularly those doubts. I will try to elaborate on parts where I get many questions.