Compare data structures for metric labels (collectd)

Authors: srivasta@google.com
Experiment Timeframe: 2019/04

Problem Statement

1. **Why**: Trying to determine the best data structure to store metric identity labels and values for collectd internal metric storage. Label keys are supposed to be unique (duplicate labels are very likely an error). The data structure needs to be performant in testing for label uniqueness, and also since every metric value needs to store the associated set of identity labels, memory efficiency can not be ignored. While the common case is for a metric to have a handful or a dozen labels, there is a potential for the number of labels to be much larger in some edge cases.
2. **Importance**: Using the correct data structure for implementing label key-value pairs is fairly critical, since every single read and write plugin will need to implement and handle the labels for metric identity. Since every metric gathered by collectd will have associated labels associated with it memory bloat will tend to blow up. Creating and dispatching the metric will need to process labels, so speed is not unimportant either.
3. **Symptoms**: This is a new feature, and not really a problem, but the SLIs of interest are speed of the hash function, number of collisions, and randomness of the distribution
4. **Baseline**: The baseline is essentially the linked list.

	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insert	Delete	Access	Search	Insert	Delete	
Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
AVL Tree	O(lg(n))	O(lg(n))	O(lg(n))	O(lg(n))	O(lg(n))	O(lg(n))	O(lg(n))	O(lg(n))	O(n)
Hash Table		O(1)	O(1)	O(1)		O(n)	O(n)	O(n)	xO(n)[1]

5. **Reproducibility**: Since this is not a bug, this is not applicable here.

[1] In practice, the hash table has to be larger than the expected number of keys to prevent rampant collision

Hypothesis

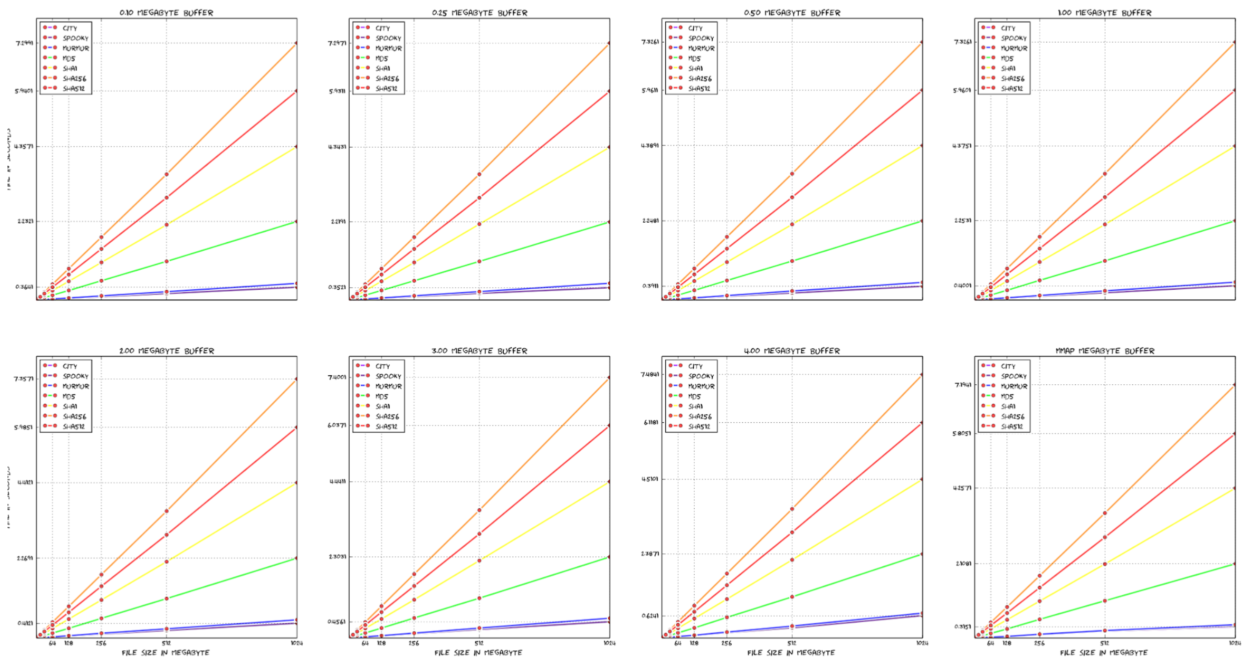
1. **Hypothesis**: One of the popular string hashing algorithms offers a better fit for storage of metric key/value parts. The most common trait is efficient collision detection, and traversal.
2. **Tradeoffs**: The tradeoffs here are in speed versus memory and code complexity. There is already an AVL tree implementation in collectd that can be reused. String has functions have better average case order complexity, but in case of collisions we need a secondary mechanism, and that adds to the complexity of implementing a hash table. A larger table reduces the possibility of any potential collisions, but that comes at the cost of increased memory usage. While the expected common case is for a small number of label key-value pairs, there is no predefined upper bound on the number.
3. **Alternatives**: The AVL tree seems like the best alternative to use here. The speed advantage This may help us in the future. Capturing why we chose a given alternative to try is good, as it helps us evaluate if things have changed in a way that makes an alternative more appealing. Sometimes, it's fine to say simple things like "we chose to try the easier approach first". You may need to describe what you did *not* do or why simpler approaches don't work. Mention other things to watch out for (if any).

Experimental Data

	Lowercase		UUID		Numbers(1..216553)		Random
	Time (ns)	Collisions	Time (ns)	Collisions	Time (ns)	Collisions	
DJB2	156	7	437	6	93	0	Good
DJB2a	158	5	443	6	91	0	Good
FNV-1 (32bit)	184	1	730	5	92	0	Excellent
FNV-1a (32 bit)	152	4	504	4	86	0	Outstanding
SDBM	148	4	484	6	90	0	Good
CRC32	250	2	946	0	130	0	Worse
murmur2	145	6	259	5	92	0	Outstanding
SuperFastHash	164	85	344	4	118	18742	Outstanding
LoseLose	338	215178					Terrible

AVL Tree	300	2					Perfect

[Details for the speed tests here.]



1. **Analysis**:
   a. Collisions
      i. **collisions rare**: FNV-1, FNV-1a, DJB2, DJB2a, SDBM
      ii. **collisions common**: SuperFastHash, Loselose
   b. Distribution
      i. **outstanding distribution:** Murmur2, FNV-1a, SuperFastHas
      ii. **excellent distribution:** FNV-1
      iii. **good distribution:** SDBM, DJB2, DJB2a
      iv. **horrible distribution:** Loselose
   c. Cityhash, Murmer3, and FNV-1a are all excellent hash functions.
   d. The AVL tree is slow, compared to the hash functions (160% as expensive as FNV-1), but it is already implemented, and collisions are not an issue. Remember that we also need to fold down the 32bithash into the array size, which means collision probability increases; and we'll have to implement storage for the hash table collision overflow. The complexity, and the much worse worst case behaviour for the hash functions make the AVL tree the right choice.

Decision & Conclusion

1. **Results**: The hash functions are indeed faster, and fairly well behaved, but that speed comes at the expense of complexity and wasted memory for a (sparse) hash table. Given that the complexity of hash table implementation overwhelms the speed advantages, and that the AVL tree exists already, it does not make sense to use a hash table.
2. **Next Steps**: Use the already implemented AVL tree for storage
3. **Accuracy**: The data was not adversarial: it was not designed to cause maximal collisions for the hash functions, so the observed behavior is not the worst case scenario for the hash functions.
4. **Caveats**: If the usage in practice were to involve a large number of labels the complexity of the hash table might become more palatable. In practice the space wasted on the sparse hash table might not be as onerous as initially estimated. However, the back-end systems are unlikely to be able to support very many identity determining labels either, so this is moot.

Reconsideration Guidelines

If a hash table implementation is added to collectd core (or a C++ std::map interface is added), then the cost and complexity equation changes. At that point further investigation of the impact of hash table memory usage might be justified. .

References

1. http://bigocheatsheet.com/
2. https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed
3. https://cp-algorithms.com/string/string-hashing.html
4. http://code.google.com/p/smhasher/wiki/MurmurHash3
5. http://www.cse.yorku.ca/~oz/hash.html
6. https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function
7. https://opensource.googleblog.com/2011/04/introducing-cityhash.html
8. https://github.com/aappleby/smhasher
9. https://create.stephan-brumme.com/fnv-hash/
10. http://www.isthe.com/chongo/tech/comp/fnv/
11. http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-16.html
12. https://stackoverflow.com/questions/12392278/measure-time-in-linux-time-vs-clock-vs-getrusage-vs-clock-gettime-vs-gettimeof

All English Words mirrors

- https://web.archive.org/web/20070221060514/http://www.sitopreferito.it/html/all_english_words.html
- https://drive.google.com/file/d/0B3BLwu7Vb2U-dEw1VkUxc3U4SG8/view?usp=

sharing