When Cycles are Cheap, Some Tables Can Be Huge

Bin Fan, Dong Zhou, Hyeontaek Lim, Michael Kaminsky[†], David G. Andersen
Carnegie Mellon University, [†]Intel Labs

Abstract

The goal of this paper is to raise a new question: What changes in operating systems and networks if it were feasible to have a (type of) lookup table that supported billions, or hundreds of billions, of entries, using only a few bits per entry. We do so by showing that the progress of Moore's law, continuing to give more and more transistors per chip, makes it possible to apply formerly ludicrous amounts of brute-force parallel computation to find space-savings opportunities.

We make two primary observations: First, that some applications can tolerate getting an incorrect answer from the table if they query for a key that is not in the table. For these applications, we can discard the keys entirely, using storage space only for the values. Further, for some applications, the value is not arbitrary. If the range of output values is small, we can instead view the problem as one of *set separation*. These two observations allow us to shrink the size of the mapping by brute force searching for a "perfect mapping" from inputs to outputs that (1) does not store the input keys; and (2) avoids collisions (and thus the related storage). Our preliminary results show that we can reduce memory consumption by an order of magnitude compared to traditional hash tables while providing competitive or better lookup performance.

1 Introduction

Large, fast lookup tables are fundamental building blocks for routers, directory services, caches, and more; in all of these cases, their not-inconsiderable cost in memory is judged worthwhile for the size of data that must be stored. Through this paper, we invite the reader to day-dream: What additional building blocks and services may be created if we were able to reduce this cost by over an order of magnitude, building lookup tables that are, in the best case of a single-bit value, perhaps 50 times smaller than before? Based upon the new approach we outline in this paper, a single commodity server can hold tens of millions of lookup entries in L3 cache (versus the hundreds of megabytes previously required), or fit hundreds of billions of entries in memory, instead of spilling to flash or disk.

Although we demonstrate this technique using only a simple hash table, it potentially applies to many scenarios:

- Why use hierarchy in IPv6 routing? Perhaps we could simply store a forwarding entry to every device in the world.
- Why use hierarchical metadata servers for distributed filesystems? Or, why not have every node cache the location of every fragment of data?
- Perhaps these content-based routing people aren't crazy after all.

The space saving arises from the core observation that for many applications, a general mapping from keys to values is not necessary. Rather, these problems should be more accurately viewed as *set separation*, if (1) the range of possible values is very small (e.g., the output ports on a router, the sub-databases to direct a query); and (2) it's OK to return seemingly-valid answers for keys that were never inserted into a set. Not storing keys (exploiting property 2) has obvious space saving advantages. Intuitively, storing a set separation requires less space than mapping to arbitrary values because when the set of possible values is small, each value repeats frequently in a normal lookup table, which leads to space inefficiency.

This sounds like a theory problem, so we take a typically boneheaded systems approach that just happens to work: We *brute-force* it, and we made the code do the brute-forcing *really* fast.

To brute-force find a set separator for a set of n keys $k_1, k_2, ..., k_n$, start with a parameterizable hash function $H_i(key)$ that produces a different value for the same key if the parameter i is changed. (One could trivially think of doing this by just appending the bits of i to the bits of *key* and passing them to a strong hash function, but we have a faster technique.)

Then, brute force until $H_i(k_1)$ indicates that k_1 is in the proper set, $H_i(k_2)$ similarly for k_2, and so on, observing that you're looking for one value of i by which H_i works for all n keys. This is an exponential search. Once such a value of i is found, we store it so that we can later look up keys using the right hash function.

Why this might be the right tradeoff: Moore's law has reserved its greatest rewards for trivially parallel, predictable control-flow, no-memory-referencing computations such as these. As we show later in this paper, such computation can leverage both instruction level parallelism, SIMD, and GP-GPUs, as necessary. Memory capacity and bandwidth, in contrast, has not advanced as rapidly as this very heart of the CPU. Therefore, set separation, viewed properly, has become an excellent opportunity to trade up-front computation (during table construction) for space.

The benefit of encoding the set separation into hash functions is twofold. First, by only storing the indexes (i) of the hash functions, we do not have to explicitly store the keys at all; storing the values plus some fixed overhead means that we can approach the information theoretical lower bound (i.e., the minimum amount of bits required). Second, by dramatically reducing the overall table size, the set separator-based table can then provide a boost in performance—a table that previously would only fit on disk, can fit in memory, and a table that previously would only fit in memory, might even fit in CPU L3 cache.

Needless to say, encoding the keys and values into hash functions also creates as many problems as it solves—how to handle updates, negative lookups, and even to answer what in the world one might do with such a large lookup table in the first place. Our goal in this paper is not to solve all of these problems, but to place them on the table as problems worth solving.

2 Applications

Set separation has many potential applications where high query performance and low memory use are critical. The following is an incomplete listing of scenarios where using set separation may offer benefits.

Network routers A router or switch looks up the destination address of each packet, selecting an output port according to its *forwarding information base* (FIB). To sustain high-speed forwarding, these routers are equipped with only a small amount of fast memory, which means FIB must be compact. The number of output ports on most individual switching chips is relatively low. This combination of requirements make set separation an attractive idea for FIB implementation.

Set separation can also provide benefits in content-based routing [8] where each distinct data item (or even individual packet) can be named. Particularly for "flat" naming designs, where, e.g., content is named by its hash, set separation may enable routers to support an order of magnitude more content forwarding entries using the same amount of memory.

Distributed storage systems Large-scale distributed storage services such as GFS [7] and HDFS [1] employ a directory service to provide the addresses of individual data blocks in the storage cluster. It has become commonplace to store billions of blocks served by tens of thousands of storage nodes [6]. Although tens of thousands is larger than the number of values for which set separation is directly optimal, the scale of this problem makes it an attractive place to apply set separation as a building block for improved scaling. The high memory efficiency and fast query processing of set separation might allow a directory service to avoid sharding directory information across multiple nodes, which helps achieve consistency as well as robustness upon node failures.

Accelerating table joins Performing table joins efficiently is important to process complex SQL-like queries in database systems. In many cases, joins are performed between a large table (fact table) and small tables (dimension tables). If the joining key does not have any null value or mismatches, and the small table contains only a small number of values (e.g., from booleans up to, perhaps, country codes), then set separation may help accelerate table joins: Each dimension table is compressed as a set separator and is quickly broadcasted to the database nodes that possess a fact table fragment for local joins.

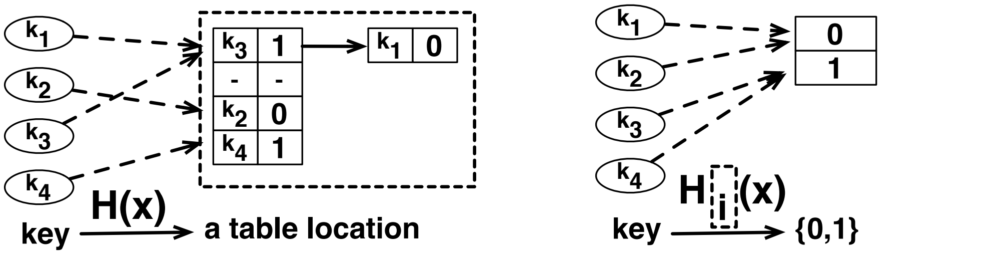
3 How it Works

Set separation problem Formally, the set separation problem is to: Construct a mapping such that for each element in set S of size N elements, the mapping returns a value in $\{1, 2, \ldots, K\}$. When the set S is large and the value K is small (e.g., 4 or 16), this mapping is equivalent to dividing the set S into K disjoint subsets. This section first briefly discusses conventional data structures that can handle set separation. The following subsection shows how directly solving the set separation problem results in a faster and more memory-efficient solution.

3.1 Conventional Solutions

Hash tables The simplest set separator is a hash table that maps each key in S to a unique location and stores its associated value there (see Figure 1a). This straightforward approach wastes space for two reasons: First, it stores the key associated with each value, both for retrievals and to resolve collisions. If two keys with different values map to the same hash table bucket, the lookup procedure must be able to identify the correct value to return, typically by comparing the stored key against the key being looked up.

Second, hash tables tend to allocate more space than the total amount of data they can hold. Simple hashing schemes such as linear probing typically use only half of their allocated space before unresolvable collisions occur; advanced open addressing hashing schemes such as *Cuckoo hashing* [14] or *d-left hashing* [12] are more space efficient (e.g., $> 90\%$), but they have to resolve



(a) Hash table: the table is stored

(b) Set separation: the hash function index i is stored

Figure 1: Illustration of using a hash table and set separator to store the mapping from four keys (k_1, k_2, k_3, k_4) to the set $\{0,1\}$. The dashed box is the part to store to represent this mapping.

frequent collisions because they assign each key multiple candidate locations.

(Minimal) Perfect hashing One way to eliminate the necessity of storing keys is to get rid of hash collisions. In order to do so, a common approach is to build an index based on a (minimal) perfect hashing scheme which guarantees that any given key in S is mapped to a *distinct* location. Such an index can be highly compressed, for example, both ECT [11], as used in SILT [10], and CHD [3] use fewer than 2.5 bits per key to store the index. In addition to the index, these schemes must also store the value of each key ($\log_2 K$ bits). Furthermore, both are too slow: ECT is optimized for indexing data in external storage such as SSDs, and its lookup latency is one to seven microseconds per lookup; CHD is faster, but is still much slower than one would like, particularly with a large number of items (see Section 4).

Observation: Both schemes assign each key a location to store the value bits explicitly. This is good when the number of possible values is large, but for set separation where K is small, giving each key a separate location adds unnecessary overhead. The information theoretic lower bound of the hashing approach requires $1.44 + \log_2 K$ bits per key, whereas the set separation approach requires only $\log_2 K$ bits, though no practical scheme quite achieves either of these limits. On the other hand, updating a value in hashing is fast and easy, as only one location in memory must be modified, without affecting other keys or values.

Bloom filters A Bloom filter [4] is a compact data structure used to represent a set of keys for set-membership tests. It achieves high space efficiency by allowing false positives, and has many applications [5]. For example, BUFFALO [15] proposed building a switch using a combination of K Bloom filters to represent K different output ports. They lookup in the mapping from key to port (the set id) by checking the candidate address for membership in all K Bloom filters. One inefficiency in this approach is that a query may see positive results from multiple Bloom filters, so the system must store additional information to resolve these false positives.

Observation: Building a switch is fundamentally a problem of set separation; using multiple binary membership tests as an approximation is less space efficient. Typically, it requires 8 or more bits per key to ensure the false positive rate is smaller than 1%.

3.2 Our Approach

Our proposal is to divide the entire set of input keys into many small *groups*. Each group is small enough (e.g., roughly 16 keys in our implementation) to ensure that a brute-force search can efficiently find a hash function that generates the correct mapping results for all the items in this group. The group size is independent of K (the number of sets/values).

Step 1: Partition keys into groups. To calculate the group id of a given key, one obvious way is to compute a hash of this key modulo the total number of groups. This approach is simple, but some groups will be significantly more loaded than the average group [13]. While a small imbalance is acceptable, very large groups require much longer to find the right hash functions using a brute-force search because the time grows exponentially as the group size increases.

To help uniformly distribute keys across groups, we instead first partition the entire input set into much larger buckets (e.g., 1024 keys per bucket in expectation). Within each bucket we spend 0.5 bits per key to refine the group partition using a two-level hashing scheme similar to [9]. In this paper, we omit the details of how this load-balanced group partitioning works.

Step 2: Search for a hash function in each group. For each group, the search process iterates over a parameterized hash function family $H_i(\cdot)$ where $i \in \{1,2,\ldots\}$ is the parameter. Starting from $i = 0$, for each key x in this group we verify if $H_i(x)$ equals the given mapping result of x; if any key x fails, we reject the current hash function H_i, switch to the next hash function H_{i+1} and start over the test for all the keys in this group (not only the keys making H_i fail). Once a hash function H_i passes all the tests for the current group, we store the index i for this group and continue to the next group (see Figure 1b). Once a maximum number of functions have been tested without success, a fallback mechanism is triggered to handle this group, e.g., we can store this group explicitly without adding too much overhead because this situation is very rare according to the theory and our experiments.

Why it saves space Given a hash function H_i, the probability to pass the test of all 16 keys in a group is $p = (1/K)^{16}$. Thus, the number of tested functions is a random variable with a Geometric distribution. Its information entropy is calculated by:

$$\frac{-(1-p)\log_2(1-p) - p\log_2 p}{p} \approx 16\log_2 K,$$

which indicates that each group must use $16\log_2 K$ bits on average to store the hash function index i. Note that this is the achievable lower bound because it is also the total number of bits required to store the values only without keys. In other words, we encode the values of each key in the hash function index i, and the keys are no longer needed, which is the source of space saving.

Why the search is fast The expected number of functions tested for each group is $1/p = K^{16}$. However, we can further speed up the brute-force search by the following observations:

- When $K > 2$, instead of looking for one hash function that outputs the right value in $\{1, \ldots, K\}$ for each key, we search for $\log_2 K$ independent hash functions, each only outputs value in $\{0, 1\}$, and the j-th hash function is responsible for the j-th bit for the mapping results. In this way, the expected total number of functions to test is reduced to $\log_2 K \cdot 2^{16}$ from K^{16}.
- The brute-force search within each group can benefit from SIMD (vector) instructions and GP-GPUs; furthermore, each $\approx$ 16-key group is independent and amenable to parallelization.

4 Vague Numbers That Show That This Idea Is Not Entirely Crazy

To test the feasibility of our idea, we built a preliminary implementation of set separation to evaluate its memory consumption as well as performance in terms of construction and lookup speed. Our current implementation consists of about 1000 lines of C++. We conducted our experiments on a commodity server equipped with two Intel Xeon E5-2680 processors and 64 GiB of DRAM. This machine runs GNU/Linux 3.2.0 x86_64, and we used gcc 4.6.3 to compile the code. All experiments used a single thread.

Table 1 compares our set separation with three different implementations: STL `unordered_map`, Google `SparseHash` [2] (a memory-efficient version of `unordered_map`), and a hash table using CHD [3] (minimal perfect hashing for indexing). To report total memory use, we used `glibc`'s `mallinfo()` feature, excluding initial memory allocation and unallocated space to accurately measure each implementation's use.

Set separation uses the least memory across all scenarios. Regardless of the key length, it requires 2.03 bits/item when the value is boolean. Both set separation and CHD have memory use independent of key length, shrinking their memory consumption considerably. As a result, set separation uses 50×–100× less memory than `unordered_map` and `SparseHash`, and its use of set separation instead of value storage enables it to use 50% less memory than the state-of-the-art CHD.

For construction, set separation is modestly slower than the other schemes, and roughly 3× slower than the STL `unordered_map`. We believe that in a full implementation, the construction throughput of set separation can be improved by exploiting parallelism.

Finally, the fastest lookups come from STL `unordered_map` when the dataset is small, and from set separation when the dataset is larger, likely due to the dataset fitting better in cache. CHD, the closest competitor to set separation in memory efficiency, always has 2×–3× slower lookup speed than set separation. We believe, but have not measured, that the hash table performance will degrade also with larger key sizes due to both memory use and the need to perform full key comparison. Neither set separation nor CHD should suffer this slowdown.

Storing 1 billion keys proved surprisingly challenging for many of the schemes. STL `unordered_map` aborted, as it required too much memory (more than our available 64 GiB) during construction. `SparseHash` completed the construction, but it consumed about 16 GiB memory. The original CHD code (from the authors) was unable to handle 1 billion keys, and ran out of memory on construction; we therefore modified it to create a set of CHDs, each for a partition (about 1 million keys) of the entire input data. Set separation used a proportionally larger amount of memory without requiring partitioning, and had slightly decreased construction speed.

5 Discussion

The primitive as we have discussed it is not perfect: It requires a lot of solid systems support around it to actually build working systems with. We discuss briefly two such

		`unordered_map`	`SparseHash`	CHD	SetSep
Baseline:	size (MiB)	625.32	266.62	6.04	**3.88**
16 Million items,	constr. (M items/sec)	**2.58**	1.48	1.05	0.84
64-bit key, 1-bit value	lookup (M items/sec)	**9.68**	3.44	3.39	9.15
Larger Keys:	size (MiB)	869.46	342.63	6.04	**3.88**
16 Million items,	constr. (M items/sec)	**2.52**	1.32	1.04	0.84
20B key, 1-bit value	lookup (M items/sec)	6.42	3.12	3.22	**8.62**
More Keys:	size (MiB)	Out of memory	15941.46	377.68	**242.43**
1 Billion items,	constr. (M items/sec)	–	0.87	**1.02**	0.70
64-bit key, 1-bit value	lookup (M items/sec)	–	1.99	1.34	**3.97**

Table 1: Comparing single-thread performance of set separation and hash tables, averaged from 5 runs.

areas in which support is particularly needed.

Support for update and delete Set separation is designed and optimized for high-performance lookups with near-optimal memory consumption. Updating the value of a single key requires, first, being able to obtain the full keys of the other elements in the group (perhaps via lookup on SSD); then update must brute-force search for a new hash for the affected group, but the rest of the set separator can remain unchanged. Inserting new keys can work in a similar way: find the group where the key should appear and recompute the group's hash function. Once the number of insertions reaches the point where a single group has too many keys, finding a hash function might become computationally infeasible. At this point, one could split the group or re-constructing the entire separator, similar to rehashing in hash tables. With deletion, there is flexibility to defer the update. The separator will return correct results for the remaining keys, but the old value for the deleted key. Alternatively, the deletion process could assign an unused value to the deleted key and recompute that group's hash.

As noted above, recomputing the separator requires the original keys. The in-memory set separator itself does not store any keys explicitly, as noted above, so the system will have to keep a copy of the original key-value pairs—most likely on disk. By batching these updates, we believe it is possible to construct systems in which fetching this data need not be on a performance critical path.

Look up non-existent keys Unlike conventional hash tables or Bloom filters, our set separator always returns meaningless mapping results for non-existent keys instead of errors to indicate that the keys are not present. These false positives mean that the target application must have an out-of-band mechanism to handle the incorrect results (or be able to ignore them).

One idea to help reduce, but not eliminate, the false positives is to place a Bloom filter in front of the set separator as a membership test. If the Bloom filter has a false positive, the separator will return an invalid answer for the not-present key.

If the application has an out-of-band mechanism to detect incorrect answers, the system can either insert that key with a "not found" value (similar to the deletion mechanism described above) or potentially keep a lookaside table or Bloom filter of keys that are known not to exist.

Several of the possible application scenarios for compact set separation can tolerate (i.e., detect) incorrect results for missing keys. For example, if a switch sees a packet with a previously unknown destination and sends it out the wrong port, network messages will notify the switch that it was incorrect at which point it can ask its neighbors to where that destination address should be forwarded. If the set separator is used to locate blocks in a distributed storage service, individual storage nodes will notify the directory service if they do not have the block. Typically, however, clients will only be looking up blocks known to exist, and the "not found" case will be rare.

6 Conclusion

Cycles become cheap
Theory falls to brute-force hack
Tables big become

World-changing applications needed. Apply inside.

References

[1] Hadoop. http://hadoop.apache.org/, 2011.

[2] Google SparseHash. https://code.google.com/p/sparsehash/, 2012.

[3] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In *Proceedings of the 17th European Symposium on Algorithms*, ESA '09, pages 682–693, 2009.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] A. Broder, M. Mitzenmacher, and A. Broder. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, volume 1, pages 636–646, 2002.

[6] A. Fikes. Storage architecture and challenges. *Talk at the Google Faculty Summit*, 2010.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[8] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proc. CoNEXT*, Dec. 2009.

[9] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact data structures for faster packet processing. In *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, pages 246–255. IEEE, 2007.

[10] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[11] H. Lim, D. G. Andersen, and M. Kaminsky. Practical batch-updatable external hashing with sorting. In *Proc. Meeting on Algorithm Engineering and Experiments (ALENEX)*, Jan. 2013.

[12] M. Mitzenmacher and B. Vocking. The asymptotics of selecting the shortest of two, improved. In *Proc. the Annual Allerton Conference on Communication Control and Computing*, volume 37, pages 326–327, 1999.

[13] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[14] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[15] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. CoNEXT*, Dec. 2009.