

A COOL AND PRACTICAL ALTERNATIVE TO TRADITIONAL HASH TABLES

ULFAR ERLINGSSON, MARK MANASSE, FRANK MCSHERRY

MICROSOFT RESEARCH – SILICON VALLEY
MOUNTAIN VIEW, CALIFORNIA, USA

ABSTRACT

Recent advances in the theoretical literature have proposed interesting modifications to traditional hash tables. The authors of these papers propose hash tables which

- a) have a guaranteed constant time for a lookup
- b) have amortized constant time for an insertion
- c) require table size only slightly larger than the space for the elements

Previous hash table technologies have offered at most two of these three.

Moreover, these hash tables do no dynamic memory allocation (except when the tables have to be resized), are easy to code, admit efficient concurrent implementations, and can be tuned (even dynamically).

This is enabled by one simple idea: allow elements in the hash table to have multiple potential locations by using multiple hash functions, and allow elements to be relocated among potential locations to make room for new elements. Such relocations can cascade, inducing a search-tree to find an empty slot, and a sequence of relocations to allow the insertion of a new key.

INTRODUCTION

Hash tables are a useful abstraction in many applications, allowing for highly efficient expected time to maintain an association between keys and data. Traditional hash tables provide for reasonably fast lookup, reasonably fast insertion, and reasonably efficient use of memory. Some known variants give up some of these to improve on others—perfect hashing, for example, uses more memory to allow expected rapid table construction, and constant time lookup, while open-address hashing allows some lookups and insertions to be slow, while using little memory.

Recent developments [1,2,3] have proposed a new class of hash tables which provide for highly space-efficient tables, offering constant-time lookup and amortized constant-time insertion.

In this paper, we present a simple generalization of these schemes, and note that some of these are well-suited to modern memory hierarchies, and to distributed settings.

THE NEW ALGORITHM, BASIC VERSION

In the simplest version of this scheme, a hash table is stored in an array of n elements. Each element e has a set of locations L_e at which it might be found in the table. If L_e consists of a single location, we have standard hashing. This version of standard hashing will fail to insert colliding elements; this can be expected to start happening when the number of elements inserted exceeds $O(\sqrt{n})$. In other traditional schemes, we would insert an element at the first unoccupied location in L_e ; this describes many variations on chained hashing, where the list of locations includes all possible overflow locations in the order in which we would probe them. In the new schemes, we consider L_e 's which contain at least two unrelated locations. If any of these locations are unoccupied, we use the first unoccupied location to hold element e . If none are vacant, we create a space, by finding an element g currently located in some location in L_e which can be relocated to some different location in L_g ; this may entail relocating a chain of other elements. More concretely, we find a list of elements $g_1, g_2, g_3, \dots, g_k$, such that:

- 1) There is a vacant location $v \in L_{g_k}$.
- 2) The current location of L_{g_i} is also in $L_{g_{i+1}}$.
- 3) L_e contains the current location of L_{g_1} .

Having found such a list, we attempt to move g_k to v , each g_i to the initial location of g_{i+1} , and then (if these moves succeeded, which they will in the absence of concurrent updates) insert e at the initial location of g_1 . If we cannot find a set of relocations in a reasonable amount of time, insertion fails. In the presence of concurrency, we don't need to undo any moves which succeeded, we merely try again to find a new list of g 's (probably of different length) and repeat the attempt to make space.

The techniques for finding a list of g 's range from performing a breadth-first search of the elements currently occupying the positions in L_e to performing a random walk, picking the element currently occupying a random position in L_e . In the event that all L_e are of size two, these two searches are the same; this is the algorithm proposed in [2]. If each L_e has at least two unrelated locations, and at least three locations, the tree will have a branching factor of at least two.

In any case, lookup is straightforward: check each location in L_e and see if e is there. If all the L_e are of some small maximum size, this is manifestly a constant-time operation.

PREVIOUS WORK

If each L_e has exactly two independently chosen random locations, the construction of the previous yields Pagh and Rodler's Cuckoo Hashing technique [2]. The algorithm works in this setting, but without good space efficiency; once the table is about half full (that is, roughly $n/2$ elements have been inserted), insertions are likely to start failing. This is due to a threshold property of random graphs: each of the sets can be viewed as an edge in a graph. The theory of random graphs is well-understood: most properties exhibit a threshold, where random graphs with sufficiently low edge probability don't exhibit the property, but once the edge probability exceeds the threshold value, the graph is very likely to exhibit the property. In this case, the property is the existence of cycles, for which the threshold occurs at $n/2$ edges. Once cycles exist, they exist in profusion, and the largest cycles are quite large. When an edge is introduced linking two nodes already in the same cycle, an irresolvable conflict is produced; again, this happens when the table is 50% full.

Other papers explore expanded sets of choices for location sets, which we describe below.

Fotokis *et al* [1] take L_e to be of size 4, all independent, except that each is in a different quarter of the table. By doing this, they demonstrate a load factor of approximately 97% before the first irresolvable conflict is produced. They provide a theoretical justification for this. It should be recognized that this is quite good: with four selections for each element, each drawn from a table of size $n/4$, we expect that the probability that a cell has no elements which map to it is $\sim(1-4/n)^n$, which is approximately e^{-4} , that is, roughly .02. We know that the load can't possibly exceed n without conflict, and so, even without considering the graph theory, we can't expect to exceed a load of 98%. The paper further considers extensions to larger sets of random choices, to drive the load factor still higher.

In contrast, Panigrahy [3] makes one additional simplifying assumption: he also chooses 4 locations, but assumes that the locations are pairwise related. To do so, we divide the array into bins of size 2, or bins with 2 cells, with locations $2x$ and $2x+1$ forming bin x . By so doing, we reduce the number of hash computations needed to 2 instead of 4. Lookups also take advantage of locality; memory systems are more efficient when looking at consecutive memory locations, due to caches, prefetching, and reduced pressure on the translation look-aside buffers (TLBs). In so doing, he proves an attainable load factor of at least 87%; that is, below that 87% load, we're almost certain to be able to insert all of our elements into the table, and above that we're very likely to encounter an irresolvable collision. This paper then considers increasing the size of bins, demonstrating theoretically an increase in attainable load factor.

We see that four independent locations are superior to two pairs of independent locations, but impose a greater burden, increasing the computation of pseudo-random values, and adding pressure on the memory system.

WHAT HAVE WE DONE?

The two papers mentioned above extend cuckoo hashing in different ways: one extends the number of hash functions used (and, to simplify the analysis but with impaired capacity, splits the table into disjoint ranges for different hash functions); the other suggests using bins of size larger than 1, and two hash functions. We can unify these by considering cuckoo hashing using a two-parameter family: a bin size, and a set of hash functions. Standard cuckoo hashing results from the parameter choice (1, 2), that is a bin of size one, and two hash functions. Fotokis-style hashing is, in this parametrization, (1, 4) (or more generally, (1, d)). Panigrahy-style hashing is (2, 2) (or more generally (k, 2)). We have experimentally tried a variety of choices; (2, 3), for instance, achieves the same load factor as (1, 4) but with one fewer hash function evaluation, and (assuming that a length 2 bin fits in a cache line, as it should for reasonable element sizes) fewer cache misses. We have not considered these choices analytically, except to observe that the existing proofs serve to establish lower bounds. We also note that increasing load sets a lower bound on the

Number of hash functions	4 hash func	97%	99%	99.9%	
	3 hash func	91%	97%	98%	99.9%
	2 hash func	49%	86%	93%	96%
	1 hash func	0.06%	0.6%	3%	12%
		1 cell	2 cells	4 cells	8 cells
		Number of cells per hash bucket			

Figure 1: Achievable load, 99% of the time, with 64K cells and fixed amount of work per insert

expected number of locations that must be inspected to complete an insertion.

We favor setting the bin size to be as large as practical without increasing pressure on the memory hierarchy stressed by a given table size. For tables that fit in memory, but not in L1 cache, this is likely to be the size of a cache line, and aligned with the cache line boundaries. This may not always be true: sufficiently large tables in user space will mean that most access to memory will cause a TLB fault; it may be profitable to increase the size of a bin to amortize the cost of that as well. For very large tables, which we expect to be disk-resident, it may make sense to increase the bin size to fill the expected unit of reading and writing to disk. Nominally, this is a sector, but current disks and controllers typically make considerably larger operations efficient.

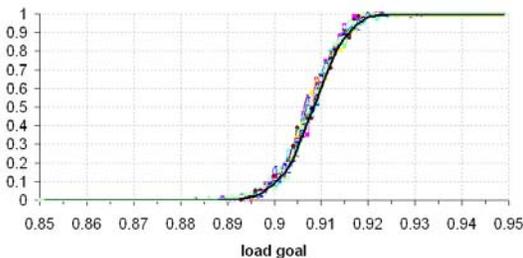


Figure 2: Prob[couldn't achieve load] for (1,3)

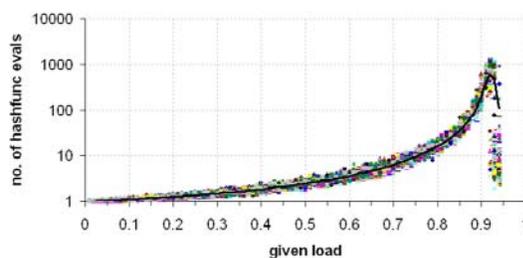


Figure 3: Amount of work per insert for (1,3)

The three figures above show some results from our experiments. Figure 1 reports on tables with 64K cells, while Figures 2 and 3 report on tables with 4K cells. As the latter two figures demonstrate, for any given L_c , the amount of work per inserted key remains reasonable until high load, and there is a sharp cutoff load at which point it becomes infeasible to insert additional keys. The total work required to fill this type of hash table up to its supported load, is amortized constant per inserted key.

Experimentally, we find (8, 2) to be almost as space-efficient as (1,4) and (2, 3); if elements can be represented in 8 bytes, this requires a line size of 64 bytes, which matches a Pentium 4 L1 cache line.

We have also considered the impact of allowing correlated-but-misaligned consolidation of elements. In this variant, each hash function would map into the full range, and the probability of unused locations is reduced, but we haven't run the experiments. That is, bins would overlap, with the advantage that fewer cells would be unreachable due to having no hashes evaluate to a particular bin, but the advantages of cache line reuse would be diminished.

In a distributed setting, or for disk-based hash tables, it would also be interesting to consider much larger bins, related, say, to the size of disk pages or network transmissions. Interesting variants for the distributed setting might also want to consider hash functions which make it more likely that elements will be relocated to other bins residing on the same host, to reduce network traffic during relocation.

WHY IS THIS USEFUL?

Many applications use hash tables. In the Web-focused research we do, we run experiments on extremely large collections of web pages, where we use hash tables to store information about each server, website, or page. In some cases we track the collection of phrases on pages, looking for collisions between documents. In such cases, our data sets place billions, or even trillions, of entries into our hash tables. In these cases, our performance is highly limited by the memory hierarchy: tables that fit in the memory of a computer are much faster than those which have to access disk

frequently. Compact representations of the hash tables are a boon.

Databases use hash tables to implement hash joins, at the very least, so this may come in handy there.

Lots of programs want to keep a bounded size cache of recent actions. Using these hash tables, we can make that cache associative, and keep as large a set as possible in a fixed amount of memory.

WHY ELSE IS THIS COOL?

These hash tables are highly robust. For instance, a hash table constructed using a small number of hash functions is still valid if viewed with more hash functions; this means that we can dynamically increase the number of hash functions to stretch the capacity of the table, if needed. Also, the tables can be valid whether keys are present in one or more of their possible locations, meaning that the relocation operation doesn't need to delete a key from an old location. Furthermore, as we move items, the table is always in a consistent state, as long as moving an item from one location to another is atomic (or even copying an item). The atomic actions are small: by finding a plan for moving elements, and then executing it in reverse the tables are always consistent, and, as described earlier, if we encounter a conflict in the middle of executing a plan, we just leave the elements where they are, and pick a new plan.

We have started experimenting with optimizing lookups by trying to put elements where the first hash function points, for those elements where lookups are frequent. We probabilistically decide to try to improve the placement of elements, so that we don't need to track access frequency, but can just say that one-quarter of the time, we try to move an element to a more-preferred location.

The hash tables are easily expanded, when the load approaches the maximum: add more table space, and expand the range of the hash functions to map into the old and new space, either by adding new hash functions mapping into the new space (or mapping to both old and new space), or by reinterpreting bins to include old and new locations. We can easily change hash functions, without fully rehashing the table all at once: expand the sets L_e by adding new hash functions, but mark the old locations in L_e as lookup-only, so that, over time, elements migrate to new hash locations. When all elements have migrated (we can keep a counter of the unmigrated elements, and trigger a forced migration when that counter becomes small enough) we can retire the lookup-only hash functions.

We can continue to break the table space into bins, but use hash functions which select a random starting element of the bin, to make operations even faster most of the time. Having done this, it's now easy to grow the size of bins, should that seem to be a good idea: we can merge adjacent pairs of bins effortlessly. This can be used to make the initial insertion of elements faster, but to allow higher utilization as the table becomes full.

CONCLUSION

Few algorithms have proven themselves as broadly useful as hash tables. With over a half-century of history, it is remarkable that new avenues remain, offering practical improvements in performance and efficiency.

REFERENCES

1. D. Fotakis et al., Space Efficient Hash Tables With Worst Case Constant Access Time, in 20th Annual Symposium on Theoretical Aspects of Computer Science, pp.271-283, Berlin, Germany, 2003.
2. R. Pagh and F. F. Rodler, Cuckoo Hashing, in 9th Annual European Symposium on Algorithms, v.2161 of Lecture Notes in Computer Science, pp. 121-133, Springer-Verlag, 2001.
3. R. Panigrahy, Efficient Hashing with Lookups in Two Memory Accesses, in 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 830-839, Vancouver, British Columbia, Canada, 2005.