AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 10 : DATA PARALLEL HASHING

BY FRED ANNEXSTEIN, PHD

10. HASHING

CHAPTER OBJECTIVES

In this chapter we study the problem of table lookups in parallel. Here we show techniques for building hash tables and bloom filters, which are effective data structures for doing parallel lookups.

DICTIONARIES

A dictionary is a data structure consisting of a collection of key-value pairs. Given a universe U of possible key elements, we wish to maintain a subset $S \subseteq U$ so that inserting, deleting, and searching in S is efficient. We define a dictionary interface as the following four methods:

- create(): initialize a dictionary with $S = \varnothing$.
- insert(u): add key element $u \in U$ to S, or add key-value pair to S.

· delete(u): delete u from S (if u is currently in S).

· lookup(u): is u in S ? If yes, then return the associated value.

The challenge for building dictionaries is that the Universe U can be extremely large so defining an array of size | U | is infeasible.

There are numerous applications for dictionaries including file systems, databases, content delivery, distributed caches, compilers, checksums, P2P networks, associative arrays, cryptography, and distributed web caching.

HASHING

A hash function is defined as any function that maps each element of the universe to a natural number up to n. So a hash function $h : U \rightarrow \{0,1, 2, \ldots, n-1\}$ can be constructed and implemented by creating a table array A of length n, and each element u is accessed using the table array element A[h(u)].

Collisions occur when the hash function maps more than one element to the same target address, that is, when $h(u) = h(v)$ but $u \neq v$. From probability theory (see the birthday

problem) we know that a collision is expected after only $2\sqrt{n}$ random insertions into the table.

There is a popular technique called chaining that is used to deal with potential collisions. In chaining we have that each table entry A[i] stores a pointer to a linked list containing all of the elements u with h(u) = i.

AD-HOC HASH FUNCTION

It is possible to create a reasonably effective hash function using modular arithmetic. In the following function we hash an arbitrary string str to a number modulo n, by processing each character and repeatedly applying a linear map and taking the result modulo n, as follows:

```
def hash(str, n):
   hash = 0
   for c in str:
       hash = (31 * hash) + ord(c)
   return hash % n

>>> hash("to be or not to be", 1000)
814
```

This type of function is known as deterministic hashing. Deterministic hashing suffers from the fact that if the universe is

larger than say n^2 items, then we can always find a set n items that all collide with each other hashing to same location. Thus, with deterministic hashing, in the worst case we can have Θ(n) time per lookup operation. It is an interesting question whether there is any deterministic hashing function that is good enough for longterm practice.

HASHING PERFOMANCE

It some sense, the ideal hash function is one in which the hash function maps the elements uniformly at random to the n possible hash slots. When using chaining, the worst case running time depends on length of the longest chains. On average the length of any chain across the n possible table entries is exactly m / n. So when n ≈ m, we have constant time expected for each execution cycle with O(1) complexity per insert, lookup, or delete.

By assuming our hash function behaves randomly we know from probability theory that with extremely high probability, as n gets large, that some chain is likely to be larger than a constant, independent of n. In fact the size can be shown to be asymptotically at least $\Omega(\ln n / \ln \ln n)$.

So we see that there is a significant challenge to implement a hash function that achieves O(1) per operation, especially when the data distribution of keys is unknown.

If we use an approach that uses randomization (random numbers) for a hashing algorithm then there is no worst-case, in the sense that any adversary that was choosing the distribution of elements to hash, could have knowledge of the randomized algorithm, but could not create or select a worst case input since they do not have knowledge of the random choices that the algorithm makes!

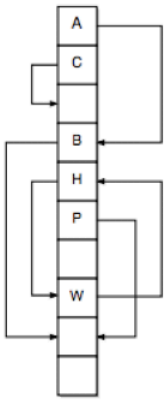
CUCKOO HASHING

There is a popular technique that has been used to avoid problems with long chains. The idea is called Cuckoo Hashing and the idea is to allocate k>1 hash tables that each has an associated, but independent, hash function.

To insert a new item, we map the item to one of the k possible locations determined by the hash functions. As long as one of the k slots has an opening, we then insert item into an open slot. Otherwise we see that there are k-collisions items—all hashed to filled-up slots. Now the idea is that we bump

out one of these colliding items (like a cuckoo bird does to another nest) so that we insert the new item into that slot. Following the bump we need to reinsert the bumped-out element back into the table. We do this by go back to the beginning of the procedure checking the remaining k-1 hash functions applied to the recently bumped item. Once the recently bumped item is placed in an open slot, the process ends. We need to consider the chance the algorithm fails by going into a loop. We would like to know that the chance of failure is small and how it depends on the size of n.

Cuckoo Hash into 1-D table




```
procedure insert(x)
  if T[h1(x)] = x or T[h2(x)] = x then return;
  pos ← h1(x);
  loop n times {
    if T[pos] = NULL then { T[pos] ← x; return};
    x ↔ T[pos];
    if pos= h1(x) then pos← h2(x) else pos← h1(x);}
  rehash(); insert(x)
end
```

Figure 2. Cuckoo hashing insertion procedure and illustration. The arrows show the alternative position of each item in the dictionary. A new item would be inserted in the position of A by moving A to its alternative position, currently occupied by B, and moving B to its alternative position which is currently vacant. Insertion of a new item in the position of H would not succeed: Since H is part of a cycle (together with W), the new item would get kicked out again.

k	h(k)	h'(k)
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3

1. table for h(k)										
	20	50	53	75	100	67	105	3	36	39
0										
1					100	67	67	67	67	100
2										
3								3	3	36
4										
5										
6		50	50	50	50	50	105	105	105	50
7										
8										
9	20	20	20	20	20	20	53	53	53	75
10										

2. table for h'(k)										
	20	50	53	75	100	67	105	3	36	39
0										3
1							20	20	20	20
2										
3									36	39
4			53	53	53	53	50	50	50	53
5										
6				75	75	75	75	75	75	67
7										
8										
9						100	100	100	100	105
10										

The two figures above represent two approaches to implementing Cuckoo Hashing. The first shows the implementation as a single table, whereas the second figures shows how the implementation can take advantage of multiple tables.

MEMBERSHIP TESTING

A membership test tells us if an item is in a given set. This may be an easier problem than hashing since we do not ask for a value to be returned. Rather in a membership test we are given a set S of m elements from large universe U, we require a fast algorithm for answering the question “is x in S?”.

Here our goal is to save space by allowing a small number of false positives results. That is the algorithm is allowed to

(infrequently) report that an item is in the set S, even though it is not.

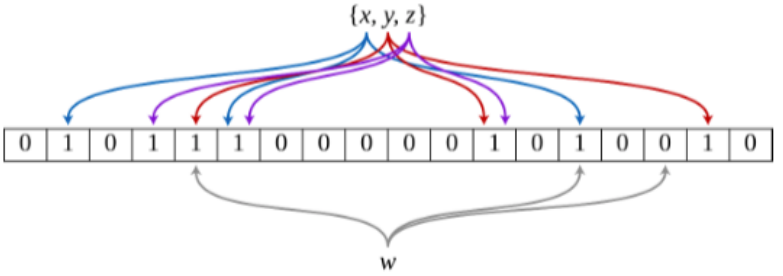
There are a number of applications where it is useful to tradeoff a large amount of space for the infrequent false positives.

For example, the problem of forbidden passwords, where you would like to quickly report to a user that their chosen password is not permitted. False positives are not a large burden in this example, since users can easily generate a new password attempt. Another popular application for this type of membership testing is for distributed caching and content delivery from a set of distributed content servers.

FINGERPRINTING

A fingerprinting algorithm is a procedure that maps an arbitrarily large data item (such as a computer file) to a much
shorter bit string, its fingerprint, that uniquely identifies the original data for all practical purposes, just as
human fingerprints uniquely identify people for practical purposes.

We can use the idea of a fingerprint is to determine membership from a key value which is a short b-bit string computed from the input. If we assume randomness in our



fingerprints, then the probability of colliding with a fingerprint from a given key is $1 - 1/2^b$.

Thus if we want to use the fingerprint to asses membership, then the probability of false positive is 1 minus the probability of missing collisions with all the forbidden keywords. We can express this false positive probability as $=$ $1 - (1 - 1/2^b)^m \leq 1 - e^{-m/2^b}$.

Therefore, if b is at least logarithmic in n then we can drive the probability of false positive probability below a given constant. In particular, if $b = 2 \log m$ bits we have a false positive probability that is $1 - (1 - 1/m^2)^m < 1/m$.

BLOOM FILTER

A Bloom filter is an array of m bits representing a set of n elements - with array set to 0 initially.

We can choose k independent hash functions with range {1, 2, …, m}. Assuming that each hash function maps each item in the universe to a random number uniformly over the range over the range {1, 2, …, m}.

For each element x in S, the bit in the array is set to 1, for $1 \leq i \leq k$,

Above is an example of a Bloom filter, representing the set $\{x, y, z\}$. The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set $\{x, y, z\}$, because it hashes to one bit-array position containing 0. For this figure, $m = 18$ and $\mathrm{k} = 3$

A bit in the bloom filter array may be set to 1 multiple times for different elements.

To check membership of y in S, check whether are all set to 1. If not, y is definitely not in S; Else, we conclude that y is in S, but sometimes this conclusion is wrong (false positive).

We will assume that kn < m. After all members of S have been hashed to Bloom filter, the probability that any specific bit is still 0 is

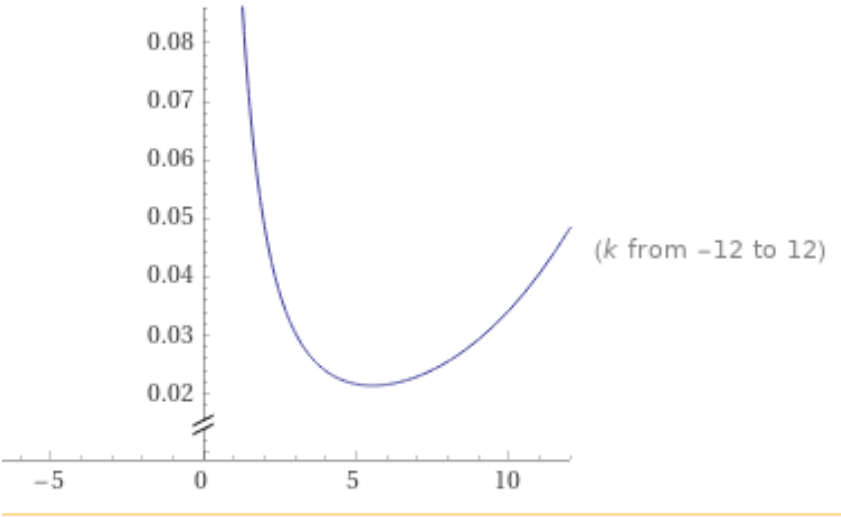
$$(1 - 1/n)^{km} \approx e^{-km/n} = p$$

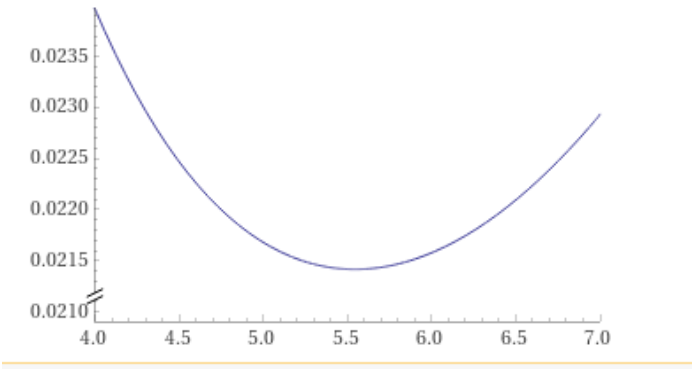
For a non member, it may be found to be a member of S (all of its k bits are nonzero) with false positive probability is $(1 - e^{-km/n})^k = (1 - p)^k$

CHOOSING THE OPTIMAL K VALUE

There are competing forces when attempting to find the optimal value for k. We find that increasing value of k results in lowering the probability that a random bit is set, and increasing the value of k results in lowering the probability of a false positive.

Here is a plot of the false positive probability, as a function of k when n = 8m, which is when we restrict range to one byte per item.





Above here is the same graph at a finer resolution.

OPTIMAL NUMBER OF HASH FUNCTIONS

Using calculus we have $k_{opt} = \ln 2(m/n)$ the false positive rate is $(1-p)^{k_{opt}} = 0.5^{k_{opt}} = 0.6185^{m/n}$

We must choosing an integer for the near optimal k, and we can do so as follows.

$$m/n = 6 \quad k = 4 \quad p_{\text{error}} = 0.0561$$
$$m/n = 8 \quad k = 6 \quad p_{\text{error}} = 0.0215$$
$$m/n = 12 \quad k = 8 \quad p_{\text{error}} = 0.003\,14$$
$$m/n = 16 \quad k = 11 \quad p_{\text{error}} = 0.000\,458$$