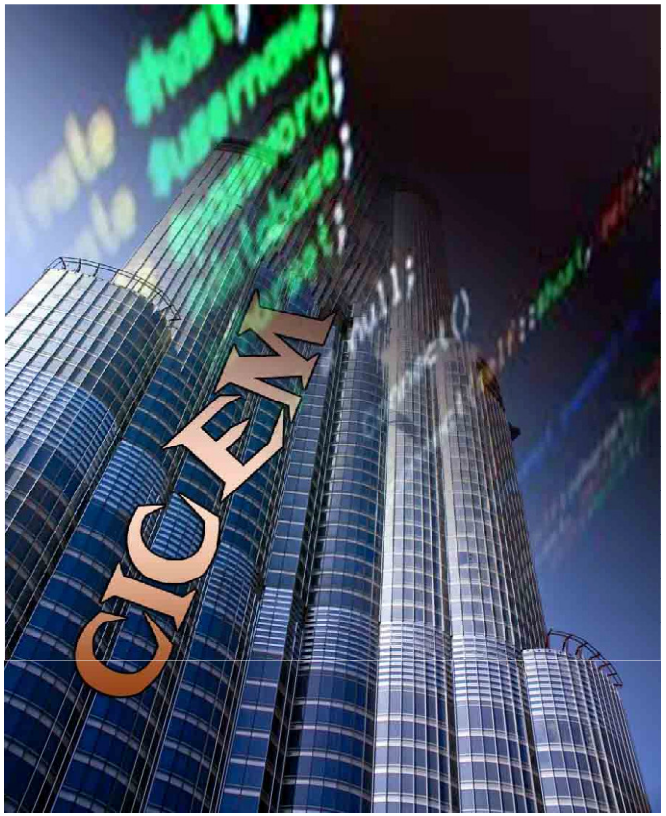

Practical Perfect Hashing for very large Key-Value Databases

Amjad Daoud, Ph.D.

http://iswsa.acm.org/mphf

Practical Perfect Hashing for very large Key-Value Databases

Abstract

This presentation describes a practical algorithm for perfect hashing that is suitable for very large KV (key, value) databases. The algorithm was recently used to compute MPHFs for a keyset with 74 billion keys.

Dictionary (Key, Value) Data Structures

- Trie's (Text retRieval data structure) can be used deterministically to find the closest match in a dictionary of words. Unsuccessful searches terminate quickly. Searches are usually tolerant and allows mistyped queries and fuzzy searches.
- Storage requirements are huge; and to be practical must be compressed sacrificing speed.
- DAWGs can be used to compress a trie using shared links for similar keys but cannot store other information as values

Dictionary Data Structures: Traditional Hashing

- Hash tables are one of the most powerful ways for searching data based on a key. The main issue with traditional hash function is the hash value collisions

- Here's an example of a hash table that uses separate chaining. To look up a word, we run it through a hash function, H() which returns a number. We then look at all the items in that "bucket" to find the data. The search algorithm must recheck the key against a list of elements sharing the same hash. This can cause some major performance issues and too much storage space.

- See **Hashing Animations Applet**

- If we could guarantee that there were no collisions, we could throw away the keys of the hash table. Very useful for URLs on the entire Internet, or parts of the human genome.

Dictionary Data Structures: Minimal perfect hashing

- Perfect hashing is builds a hash table with no collisions and O(1) search time; all of the keys must be known in advance. ***Minimal*** implies no empty slots.
- Hash functions of the form F(d, key), uses a small table G of parameters or displacements to find the unique position for the key. F(0, key), can be used to index the table G.
- The function F will always returns a the correct location of the key if it is known for sure that it is in the table; no false positives. Usually faster than Bloomier Filters; Bloom Filter Animation
- Requires more space than Bloom Filters on average 2 bits/key(very close to theoretical limit).

Dictionary Data Structures: Minimal perfect hashing MapOrderSearch algorithm

- How do we find the G table in linear time? **By exploiting randomness.**
- **Mapping Step**, map the keys into buckets using a simple hash.
- **Ordering Step**, we process the buckets ***largest first***
- **Searching Step** try to place all the keys it contains in an empty slot of the value table using F(d=1, key). If that is unsuccessful, we use different random displacements d. Randomness is key here and it is fast since the table is largely empty and much larger than the size of patterns being processed. All patterns of size 1 are fitted directly and negative displacements are used.

Minimal perfect hashing References

- Implementing the MOS Algorithm II CACM92 , and Amjad M Daoud Ph.D. Thesis 1993 at VT
- An example mphf in C for the unix dictionary.
- The code ported to Python; download as http://iswsa.acm.org/mphf/mphf.py. For the javascript port: download as http://iswsa.acm.org/mphf/mphf.js
- The algorithm is used to compute Google page rank Google Page Rank in C#
- Better and more scalable algorithm for perfect hashing Perfect Hash Functions for Large Web Repositories, The Seventh International Conference on Information Integration and Web Based Applications & Services (iiWAS2005)

Many Derivative Open Source Implementations

1. Fuzzy Tolerant Search with DWAGs and MPHF
2. CMPH Library cmph.sourceforge.com,
3. MPHF in C#
4. http://burtleburtle.net/bob/ha sh/perfect.html (splits keys into buckets by a first h1, sorts buckets by size, maps them in decreasing order so table[hash1(key)] ^ hash2(key) causes no collision).
5. The algorithm is used to compute **Google page rank** Google Page Rank in C#;

Minimal perfect hashing Python Code

- import sys
- import math
- # first level simple hash ... used to disperse patterns using random d values
- def hash(d, str):
- if d == 0: d = 0x811C9DC5
- # Use the FNV-1a hash
- for c in str:
- d = d ^ ord(c) * 16777619 & 0xffffffff # FNV-1a
- return d

Minimal perfect hashing Python Code

- # create PHF with MOS(Map,Order,Search), g is specifications array
- def CreatePHF(dict):
- # size = len(dict) for minimal perfect hash
- size = nextprime(len(dict)+len(dict)/4)
- print "Size = %d" % (size)
- gsize = nextprime(int(size/(4*math.log(size,2)))) #c=4 corresponds to 4 bits/key
- print "G array size = %d" % (gsize)
- sys.stdout.flush()

Minimal perfect hashing Python Code

```
#Step 1: Mapping
    patterns = [ [] for i in range(gsize) ]
    g = [0] * gsize #initialize g
    values = [None] * size #initialize values

    for key in dict.keys():
        patterns[hash(0, key) % gsize].append( key )
```

Minimal perfect hashing Python Code

```
# Step 2: Sort patterns in descending order and process
    patterns.sort( key= len, reverse=True )
    for b in xrange( gsize ):
        pattern = patterns[b]
        if len(pattern) <= 1: break

        d = 1
        item = 0
        slots = []
```

Minimal perfect hashing Python Code

```
# Step 3: rotate patterns and search for suitable displacement
        while item < len(pattern):
            slot = hash( d, pattern[item] ) % size
            if values[slot] != None or slot in slots:
                d += 1
                if d < 0 : break
                item = 0
                slots = []
            else:
                slots.append( slot )
                item += 1

        if d < 0:
           print "failed"
           return

        g[hash(0, pattern[0]) % gsize] = d
        for i in range(len(pattern)):

            values[slots[i]] = dict[pattern[i]]
```

Minimal perfect hashing Python Code

```
# Process patterns with one key and use a negative value of d
 freelist = []
 for i in xrange(size):
     if values[i] == None: freelist.append( i )

 for b in xrange(b+1,gsize ):
     pattern = patterns[b]
     if len(pattern) == 0: break
     #if len(pattern) > 1: continue;
     slot = freelist.pop()
     # subtract one to handle slot zero
     g[hash(0, pattern[0]) % gsize] = -slot-1
     values[slot] = dict[pattern[0]]

        print "PHF succeeded"
 return (g, values)
```

Minimal perfect hashing Python Code

- # Look up a value in the hash table, defined by g and V.
- def lookup(g, V, key):
- d = g[hash(0,key) % len(g)]
- if d < 0: return V[-d-1]
- return V[hash(d, key) % len(V)]

Minimal perfect hashing Python Code

```
# main program

#reading keyset size is given by num
DICTIONARY = "/usr/share/dict/words“

dict = {}
line = 1
for key in open(DICTIONARY, "rt").readlines():
    dict[key.strip()] = line
    line += 1
    if line > num: break

(g, V) = CreatePHF( dict )

#printing phf specification
print g
```

Minimal perfect hashing Python Code

- #fast verification for few (key,value) count given by num1
- num1 = 5
- print "Verifying hash values for the first %d words"% (num1)
- line = 1
- for key in open(DICTIONARY, "rt").readlines():
- line = lookup(g, V, key.strip())
- print "Word %s occurs on line %d" % (key.strip(), line)
- line += 1
- if line > num1: break

Hashing Animations and Videos

- Hashing Animations Applet
- MIT 6.046J Introduction to Algorithms