Application Note: Scalable Hashing

6/25/2001

Problem

How can you use Judy to create a scalable hash table with outstanding performance and automatic scaling, while avoiding the complexity of dynamic hashing?

Solution

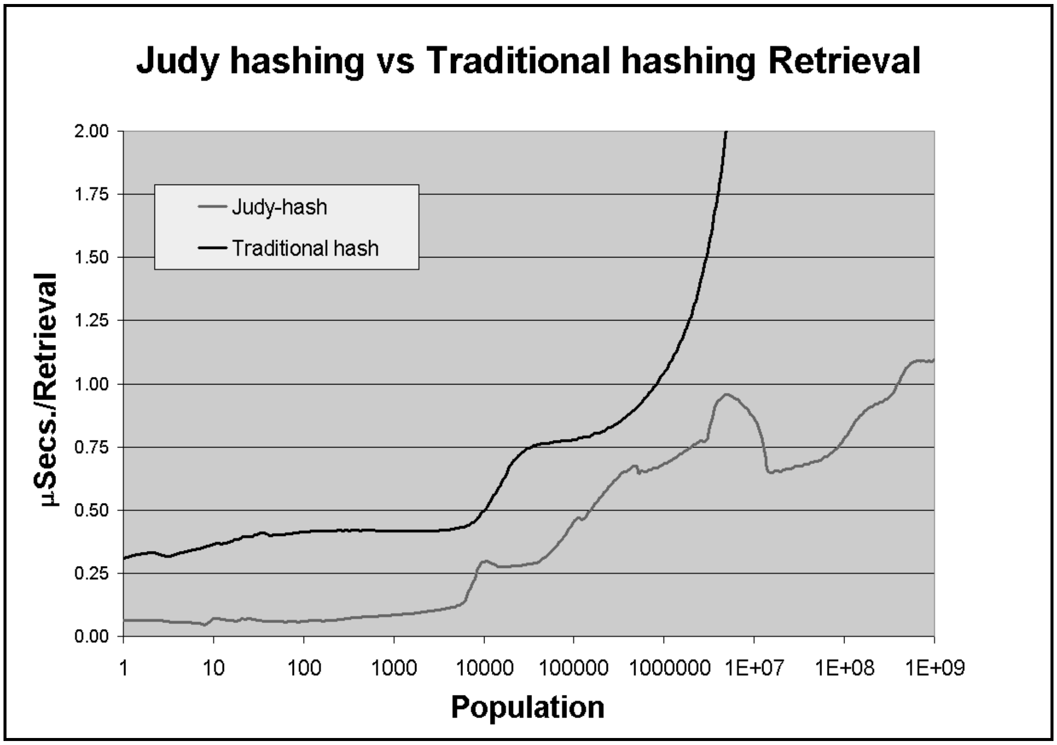
Create a hash-Judy hybrid to build on the strengths of each method. There are three parts to this solution:

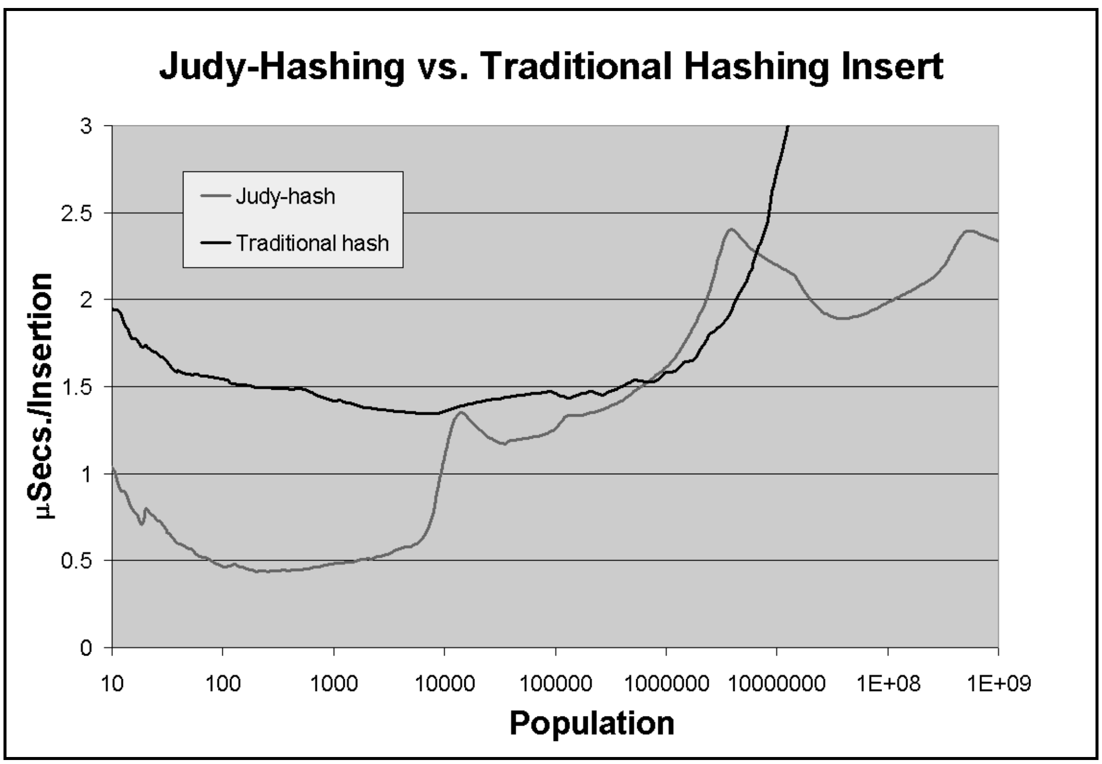
1. Keep the hash table small so it resides in CPU cache.
2. Use JudyL for handling hash table synonym chains to take advantage of Judy's excellent performance and scaling.
3. Don't waste CPU time on a complex hashing algorithm.

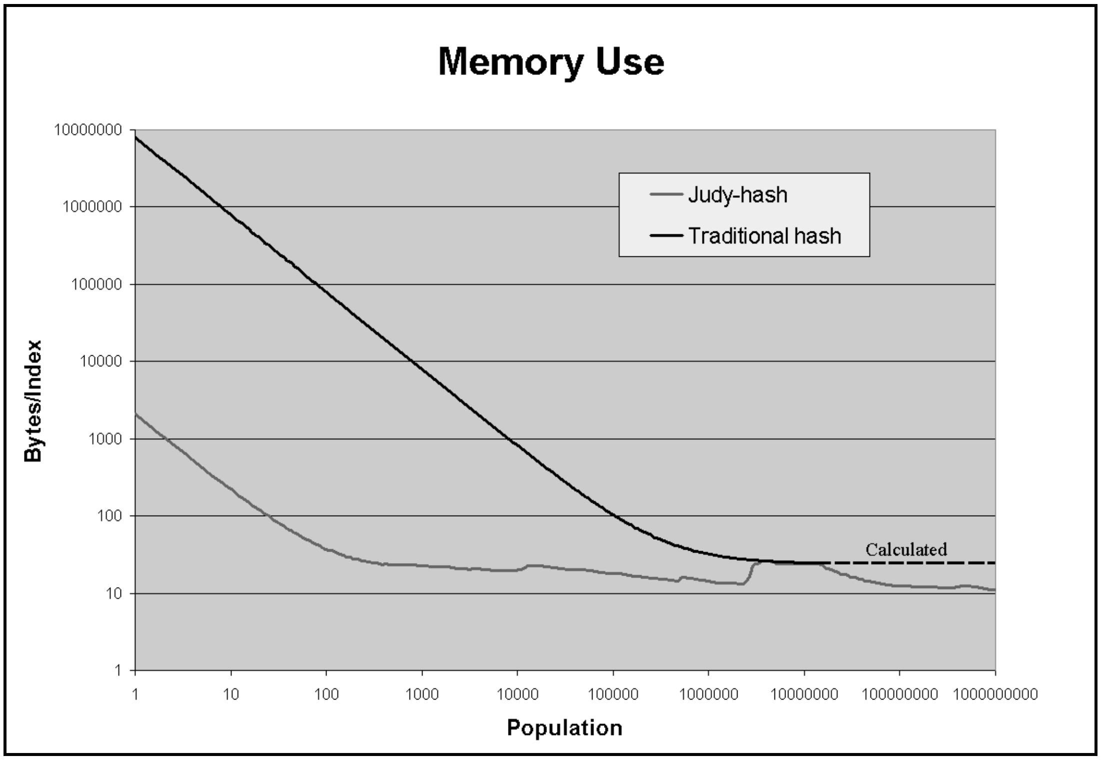
Create a small hash table with 2^8 or 2^{16} (256 or 65,536) buckets. Make the hash table size (i.e. the number of buckets) a power of two. This will enable you to use a mask instead of the mod function to determine the bucket index (which is much faster than a mod of a non-power of 2). In particular, 2^8 or 2^{16} optimizes JudyL performance.

Each bucket contains a pointer to a JudyL array. Use `JLG()` (the `JudyLGet()` macro) to retrieve the data from each bucket (the JudyL array). For up to a synonym population of 31 indexes, the JLG() macro is "inlined" and retrieves data from the Judy arrays without making a function call.

By using Judy to handle the collision chains, this hash algorithm doesn't have the performance degradation you see with long chains. This hybrid also typically outperforms a pure hash solution because the collision chain indexes are located in contiguous memory, sometimes even the same cache line.


Judy hashing vs Traditional hashing Retrieval
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Judy-hash
Traditional hash
µSecs./Retrieval
1
10
100
1000
10000
100000
1000000
1E+07
1E+08
1E+09
Population


Judy-Hashing vs. Traditional Hashing Insert
3
2.5
2
1.5
1
0.5
0
Judy-hash
Traditional hash
µSecs./Insertion
10
100
1000
10000
100000
1000000
10000000
1E+08
1E+09
Population



The previous graphs compare the Judy-hash hybrid to a hash table. The hash table was built with 1000003 (a prime number) buckets and a linked list for handling collisions. The Judy-hash hybrid used 256 buckets and JudyL arrays for the initial bucket fill as well as collisions (as described above). In both cases a single set of unique random numbers was used to populate the tables. The benchmarks were run on a 550 MHz PA 8600 System (N Class) using about 12 GBytes of RAM.

Example Code

This code provides outstanding performance at low populations due to being able to get away with a very simple hashing algorithm.

If this code is compiled using `cc -DHASHSIZE=1 hash.c` …, then the hash table size will be 1, the performance will revert to a simple JudyL array, and the memory usage will also appear nearly flat from 1 to 1000.

Using a hash table larger than 256 will degrade the performance at the high end by roughly 30%.

If you want to use your own hashing algorithm, modify the highlighted lines in the code below.

```
 // Sample program to show how to use Judy as a collision
 // handler within a Hash table.
 //
 //      cc -DHASHSIZE=256 hash.c ..

#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>

#define JUDYERROR_SAMPLE 1     // use default Judy error handler
#include <Judy.h>

// Needed for timing routines
#include <sys/time.h>

// Start of timing routines ========================================

struct timeval TBeg, TEnd;
#define STARTTm  gettimeofday(&TBeg, NULL)
#define ENDTm    gettimeofday(&TEnd, NULL)

#define DeltaUSec \
         (  ((double)TEnd.tv_sec * 1000000.0 + (double)TEnd.tv_usec) \
          - ((double)TBeg.tv_sec * 1000000.0 + (double)TBeg.tv_usec) )

// End of timing routines   ========================================

// Define Hash table size if not in compile line ===================
// Set HASHSIZE 1  for straight Judy

#ifndef HASHSIZE
#define HASHSIZE (1 << 8)     // hash table size 256
#endif

// Seed for pseudo-random counter ==================================

#define INITN         123456         // first Index to store

static uint32_t               // Placed here for INLINE possibility
Random(uint32_t Seed)  // produce 2^32 -1 numbers by different counting


{
       if ((int32_t)Seed < 0) { Seed += Seed; Seed ^= 16611; }
       else                   { Seed += Seed; }
       return(Seed);
}



// Hash Table ======================================================

Pvoid_t JArray[HASHSIZE] = { NULL }; // Declare static hash table

int main(int argc, char *argv[])
{
       Word_t Count;
       Word_t Index;
       Word_t *PValue;
       Word_t NumIndexes = 10000;  // default first parameter

       if (argc > 1) NumIndexes = strtoul(argv[1], NULL, 0);

//     Load up the CPU cache for small measurements:
       for (Count = 0; Count < HASHSIZE; Count++) JArray[Count] = NULL;

       printf("Begin storing %lu random numbers in a Judy scalable hash array\n",
              NumIndexes);

       Index   = INITN;
       STARTTm;
       for (Count = 0; Count < NumIndexes; Count++)
       {
              Index = Random(Index);

              JLI(PValue, JArray[Index % HASHSIZE], Index/HASHSIZE);
```

```
                *PValue += 1; // bump count of duplicate Indexes
        }
        ENDTm;

        printf("Insertion of %lu indexes took %6.3f microseconds per index\n",
                NumIndexes, DeltaUSec/NumIndexes);

        Index = INITN;  // start the same number sequence over
        STARTTm;
        for (Count = 0; Count < NumIndexes; Count++)
        {
                Index = Random(Index);

                JLG(PValue, JArray[Index % HASHSIZE], Index/HASHSIZE);

                if (*PValue != 1)
        printf("%lu dups of %lu\n", *PValue - 1, Index);
        }
        ENDTm;

        printf("Retrieval of %lu indexes took %6.3f microseconds per index\n",
        NumIndexes, DeltaUSec/NumIndexes);

        return(0);
}
```

