Appendix for

Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation

Mikkel Thorup *Yin Zhang*

AT&T Labs—Research, Shannon Laboratory
180 Park Avenue, Florham Park, NJ 07932, USA.
(mthorup,yzhang)@research.att.com

June 19, 2003 8:14 PM

A Second moment estimation

Let $\mathcal{S} = (a_1, w_1), (a_2, w_2), ..., (a_s, w_s)$ be a data stream, where each key a_i is a member of $[u]$. Let $v_a = \sum_{i:\ a_i=a} w_i$ be the total weights associated with key $a \in [u]$. Define, for each $j \geq 0$,

$$F_j = \sum_{a\in[u]} v_a^j$$

The second moment, F_2, is of particular interest, since it arises in various applications.

A.1 Second moment estimators

Classic estimator The classic method for estimating F_2 by Alon *et. al.* [1] uses n counters c_i ($i \in [n]$) and n independent 4-universal hash functions s_i that map $[u]$ into $\{-1, 1\}$. When a new data item (a, w) arrives, all n counters are updated using $c_i += s_i(a) \cdot w$ ($i \in [n]$). F_2 is then estimated as $X_{\text{classic}} = \sum_{i\in[n]} c_i^2/n$. Following the analysis in [1], we have $\mathrm{E}\left[X_{\text{classic}}\right] = F_2$ and $\mathrm{Var}\left[X_{\text{classic}}\right] = \sum_{a\neq b} 2\, v_a^2 v_b^2 = 2(F_2^2 - F_4)/n$.

Count sketch based estimator Recently, Charikar *et. al.* [2] described a data structure called *count sketch* for keeping track of frequent items in a data stream. We can adapt count sketch to make second moment estimation. Using this method, we need n counters c_i ($i \in [n]$) and two independent 4-universal hash functions $h : [u] \rightarrow [n]$ and $s : [u] \rightarrow \{-1, 1\}$. When a new data item (a, w) arrives, a single counter $c_{h(a)}$ is updated using $c_{h(a)} += s(a) \cdot w$. F_2 is then estimated as $X_{\text{count\_sketch}} = \sum_{i\in[n]} c_i^2$. We can prove that $\mathrm{E}\left[X_{\text{count\_sketch}}\right] = F_2$ and $\mathrm{Var}\left[X_{\text{count\_sketch}}\right] = 2(F_2^2 - F_4)/n$. Therefore, $X_{\text{count\_sketch}}$ achieves the same variance as X_{classic} with substantially lower update cost.

Fast count sketch based estimator An alternative way of implementing the count sketch scheme is to use $2n$ counters c_i ($i \in [2n]$) and a 4-universal hash function $h : [u] \rightarrow [2n]$. When a new data item (a, w) arrives, w is directly added to the counter $c_{h(a)}$: $c_{h(a)} += w$. In the end F_2 is estimated using the alternating sum $X_{\text{fast\_count\_sketch}} = \sum_{i\in[n]}(c_{2i} - c_{2i+1})^2$. $X_{\text{fast\_count\_sketch}}$ achieves the same variance as $X_{\text{count\_sketch}}$, but is faster because the direct update of a counter based on a single hash value is much simpler. However, such simplicity comes at the cost of doubling the space.

Our new estimator Here we present a new estimator that achieves the same speed and variance as the fast count sketch based estimator without having to double the space. Instead of using $2n$ counters, our new method uses $m = n + 1$ counters c_i ($i \in [m]$), and a 4-universal hash function $h : [u] \to [m]$. The update algorithm is exactly the same as that of the fast count sketch based estimator: when a new data item (a, w) arrives, w is added to the counter $c_{h(a)}$: $c_{h(a)}$+= w. But the estimation formula is quite different. We use

$$X_{\text{new}} = \frac{m}{m-1}\sum_{i\in[m]} c_i^2 - \frac{1}{m-1}(\sum_{i\in[m]} c_i)^2$$

Note that we do not worry about the cost of adding up counters done in the end. Hence, it is not considered a problem to have a more complex sum for this. Below we prove that $\mathrm{E}[X_{\text{new}}] = F_2$ and $\mathrm{Var}[X_{\text{new}}] = 2(F_2^2 - F_4)/(m-1) = 2(F_2^2 - F_4)/n$.

A.2 Analysis of second moment estimator

Notations: For any possibly identical $a, b \in [u]$, let $a \sim b$ denote $h(a) = h(b)$, and $a \nsim b$ denote $h(a) \neq h(b)$. Let $a \overset{\sim}{\neq} b$ denote $a \sim b$ but $a \neq b$.

Clearly, we have

$$\begin{aligned} X_{\text{new}} &= \frac{m}{m-1}\sum_{a\sim b} v_a v_b - \frac{1}{m-1}\sum_{a,b} v_a v_b \\ &= \frac{m}{m-1}\left(\sum_a v_a^2 + \sum_{a \overset{\sim}{\neq} b} v_a v_b\right) - \frac{1}{m-1}\left(\sum_a v_a^2 + \sum_{a \overset{\sim}{\neq} b} v_a v_b + \sum_{a\nsim b} v_a v_b\right) \\ &= F_2 + \sum_{a \overset{\sim}{\neq} b} v_a v_b - \frac{1}{m-1}\sum_{a\nsim b} v_a v_b \qquad (1) \end{aligned}$$

Define

$$X_{a,b} = \begin{cases} 1 & \text{if } a \sim b \\ -\frac{1}{m-1} & \text{otherwise} \end{cases}$$

(1) becomes

$$X_{\text{new}} = F_2 + \sum_{a\neq b} v_a v_b X_{a,b} \qquad (2)$$

If h is 2-universal, then for any distinct $a, b \in [u]$, we have

$$\mathrm{E}[X_{a,b}] = 0 \qquad (3)$$

$$\mathrm{E}\left[X_{a,b}^2\right] = \frac{1}{m-1} \qquad (4)$$

In addition, if h is 4-universal, then for any distinct $a, b \in [u]$ and distinct $c, d \in [u]$ such that $\{a, b\} \neq \{c, d\}$, we have

$$\mathrm{E}[X_{a,b}X_{c,d}] = \mathrm{E}[X_{a,b}]\,\mathrm{E}[X_{c,d}] =_{(3)} 0 \qquad (5)$$

Theorem 1 *If h is 2-universal, then*

$$\mathrm{E}[X_{\text{new}}] = F_2 \qquad (6)$$

Proof

$$\mathrm{E}[X_{\text{new}}] =_{(2)} F_2 + \mathrm{E}\left[\sum_{a\neq b} v_a v_b X_{a,b}\right] = F_2 + \sum_{a\neq b} v_a v_b \mathrm{E}[X_{a,b}] =_{(3)} F_2$$

■

Theorem 2 *If h is* 4*-universal, then*

$$\mathrm{Var}[X_{\text{new}}] = \frac{2}{m-1}(F_2^2 - F_4) \tag{7}$$

Proof

$$\begin{aligned}
\mathrm{Var}[X_{\text{new}}] &= \mathrm{E}\left[(X_{\text{new}} - F_2)^2\right] =_{(2)} E(\sum_{a\neq b} v_a v_b X_{a,b})^2 && (8)\\
&= 2\sum_{a\neq b} v_a^2 v_b^2 \mathrm{E}\left[X_{a,b}^2\right] + \sum_{\substack{a\neq b \wedge c\neq d\\ \{a,b\}\neq\{c,d\}}} v_a v_b v_c v_d \mathrm{E}[X_{a,b}X_{c,d}] && (9)\\
&=_{(4)\,(5)} 2\sum_{a\neq b} v_a^2 v_b^2 \frac{1}{m-1} = \frac{2}{m-1}(F_2^2 - F_4) && (10)
\end{aligned}$$

■

A.3 Dealing with long hash keys

Our constructions of 4-universal hash functions are most efficient when the hash key length coincides with single or double machine word length. If the hash keys are longer, we show here that we can first hash them into a smaller domain $[2^{64}$ using 2-universal hashing with little degradation on the accuracy of the final estimator.

Theorem 3 *Let h be a* 2*-universal hash function from* $[u]$ *into* $[u']$*. Let* $X = \sum_{a\sim b} v_a v_b$*. We have*

$$\mathrm{E}[|X - F_2|] \le \frac{1}{u'}(\sum_a |v_a|)^2 \tag{11}$$

Proof Let

$$Y_{a,b} = \begin{cases} 1 & \text{if } a \sim b\\ 0 & \text{otherwise}\end{cases}$$

Since h is 2-universal, for any distinct $a, b \in [u]$, we have

$$\mathrm{E}[Y_{a,b}] = 1/u' \tag{12}$$

Clearly,

$$X = \sum_a v_a^2 + \sum_{a \not\approx b} v_a v_b = F_2 + \sum_{a\neq b} v_a v_b Y_{a,b} \tag{13}$$

Therefore,

$$\mathrm{E}[|X - F_2|] \le_{(13)} \sum_{a\neq b} |v_a v_b| \mathrm{E}[Y_{a,b}] =_{(12)} \sum_{a\neq b} |v_a v_b| \frac{1}{u'} \le \frac{1}{u'}(\sum_a |v_a|)^2$$

■

If the mass of $\{|v_a|\}$ is distributed on $n' \ll 2^{64}$ keys, which is almost certainly the case in any foreseeable future, then we have

$$(\sum_a |v_a|)^2 / F_2 \le n' \ll 2^{64}$$

Therefore, if we choose u' to be 2^{64}, then by (11) and Markov's Inequality, $|X - F_2|/F_2$ will be small with very high probability.

A.4 Performance evaluation

The code for second moment estimation can be found in §B.6. We used $m = 2^{15}$, giving us a relative standard error below $\sqrt{2/(m-1)} \approx 2^{-7} < 1\%$. Table 1 compares the instruction count and the running times for performing 10 million hash computation and counter updates. We see that the additional overhead is very small. Even in the worst case when all packets are of the minimum IP packet size of 40 bytes (320 bits), the counter update part can easily keep up with $320 \times 10^7/0.68 = 4.7 \times 10^9$ bits per second on the slower computer, which is nearly twice as fast as OC48 speed (2.48 Gbps). With enough buffering, if the average IP packet size is above 85 bytes (680 bits), which is generally the case in today's Internet, we can even keep up with OC192 speed (10 Gbps).

	C-level	running time (sec)	
algorithm	instructions	computer A	computer B
Table32	8	0.30	0.55
StreamUpdate2nd	11	0.50	0.68

Table 1: Instruction count plus running times for performing 10 million hash computations and counter updates on computer A (400 MHz SGI R12k processor running IRIX64 6.5) and B (900 MHz Ultrasparc-III processor running Solaris 5.8).

B Code

B.1 Common data types and macros

```
typedef unsigned char       INT8;
typedef unsigned short      INT16;
typedef unsigned int        INT32;
typedef unsigned long long  INT64;
typedef INT64               INT96[3];

// different views of a 64-bit double word
typedef union {
  INT64 as_int64;
  INT16 as_int16s[4];
} int64views;

const INT64 LowOnes = (((INT64)1)<<32)-1;

#define LOW(x)  ((x)&LowOnes)     // extract lower 32 bits from INT64
#define HIGH(x) ((x)>>32)         // extract higher 32 bits from INT64
```

B.2 Tabulation based hashing for 32-bit keys using 16-bit characters

```
/* tabulation based hashing for 32-bit key x using 16-bit characters.
 * T0, T1, T2 are precomputated tables */
inline INT64 Table32(INT32 x, INT64 T0[], INT64 T1[], INT64 T2[]) {
  INT32 x0, x1, x2;
  x0 = x&65535;
  x1 = x>>16;
  x2 = x1 + x2;
  x2 = compress32(x2);  // optional compression
  return T0[x0] ^ T1[x1] ^ T2[x2];
} // 8 + 4 = 12 instructions

/* optional compression */
inline INT32 compress32(INT32 i) {
  return 2 - (i>>16) + (i&65535);
} // 4 instructions
```

The code uses 12 instructions (8 without compression), including 3 lookups.

B.3 Tabulation based hashing for 64-bit keys using 16-bit characters

```
/* tabulation based hashing for 64-bit key x using 16-bit characters.
 * T0, T1, T2, T3, T4, T5, T6 are precomputated tables */
inline INT64 Table64(int64views x, INT64 *T0[], INT64 *T1[], INT64 *T2[],
                          INT64 *T3[], INT64 T4[], INT64 T5[], INT64 T6[])
{
  INT64 *a0, *a1, *a2, *a3, c;

  a0 = T0[x.as_int16s[0]];  a1 = T1[x.as_int16s[1]];
  a2 = T2[x.as_int16s[2]];  a3 = T3[x.as_int16s[3]];

  c = a0[1] + a1[1] + a2[1] + a3[1];
  c = compress64(c);   // optional compression

  return
    a0[0] ^ a1[0] ^ a2[0] ^ a3[0] ^
    T4[c&2097151] ^ T5[(c>>21)&2097151] ^ T6[c>>42];
} // 32 + 5 instructions



/* optional compression */
inline INT64 compress64(INT64 i) {
  const INT64 Mask1 = (((INT64)4)<<42) + (((INT64)4)<<21) + 4;
  const INT64 Mask2 = (((INT64)65535)<<42) + (((INT64)65535)<<21) + 65535;
  const INT64 Mask3 = (((INT64)32)<<42) + (((INT64)32)<<21) + 31;

  return Mask1 + (i&Mask2) - ((i>>16)&Mask3);
} // 5 instructions
```

The code uses 37 instructions (32 without compression) including 7 lookups.

B.4 CW Trick for 32-bit keys with prime $2^{61} - 1$

```
const INT64 Prime = (((INT64)1)<<61) - 1;

/* Computes ax+b mod Prime, possibly plus 2*Prime,
   exploiting the structure of Prime.  */
inline INT64 MultAddPrime(INT32 x, INT64 a, INT64 b) {
  INT64 a0,a1,c0,c1,c;
  a0 = LOW(a)*x;
  a1 = HIGH(a)*x;
  c0 = a0+(a1<<32);
  c1 = (a0>>32)+a1;
  c  = (c0&Prime)+(c1>>29)+b;
  return c;
} // 12 instructions

/* CWtrick for 32-bit key x with prime 2^61-1 */
inline INT64 CWtrick(INT32 x, INT64 A, INT64 B, INT64 C, INT64 D) {
  INT64 h;
  h = MultAddPrime(MultAddPrime(MultAddPrime(x,A,B),x,C),x,D);
  h = (h&Prime)+(h>>61);
  if (h>=Prime) h-=Prime;
  return h;
} // 12*3 + 5 = 41 instructions
```

The code uses 41 instructions including 6 multiplications.

B.5 CW trick for 64-bit keys using prime $2^{89} - 1$

```
const INT64 Prime89_0  = (((INT64)1)<<32)-1;
const INT64 Prime89_1  = (((INT64)1)<<32)-1;
const INT64 Prime89_2  = (((INT64)1)<<25)-1;
const INT64 Prime89_21 = (((INT64)1)<<57)-1;

/* Computes (r mod Prime89) mod 2^64, exploiting the structure of Prime89 */
inline INT64 Mod64Prime89(INT96 r) {
  INT64 r0, r1, r2;

  // r2r1r0 = r&Prime89 + r>>89
  r2 = r[2];  r1 = r[1]; r0 = r[0] + (r2>>25);  r2 &= Prime89_2;

  return (r2 == Prime89_2 && r1 == Prime89_1 && r0 >= Prime89_0) ?
         (r0 - Prime89_0) : (r0 + (r1<<32));
} // 7 instructions (worst case)

/* Computes a 96-bit r s.t.    r mod Prime89 == (ax+b) mod Prime89
   exploiting the structure of Prime89. */
inline void MultAddPrime89(INT96 r, INT64 x, INT96 a, INT96 b) {
  INT64 x1, x0, c21, c20, c11, c10, c01, c00;
  INT64 d0, d1, d2, d3;
  INT64 s0, s1, carry;

  x1 = HIGH(x);  x0 = LOW(x);

  c21 = a[2]*x1;  c11 = a[1]*x1;  c01 = a[0]*x1;
  c20 = a[2]*x0;  c10 = a[1]*x0;  c00 = a[0]*x0;

  d0 = (c20>>25)+(c11>>25)+(c10>>57)+(c01>>57);
  d1 = (c21<<7);
  d2 = (c10&Prime89_21) + (c01&Prime89_21);
  d3 = (c20&Prime89_2) + (c11&Prime89_2) + (c21>>57);

  s0 = b[0] + LOW(c00) + LOW(d0) + LOW(d1);
  r[0] = LOW(s0);  carry = HIGH(s0);

  s1 = b[1] + HIGH(c00) + HIGH(d0) + HIGH(d1) + LOW(d2) + carry;
  r[1] = LOW(s1);  carry = HIGH(s1);

  r[2] = b[2] + HIGH(d2) + d3 + carry;
} // 59 instructions

/* cW trick for 64-bit key x with prime 2^89-1 */
inline INT64 CWtrick89(INT64 x, INT96 A, INT96 B, INT96 C, INT96 D) {
  INT96 r;
  MultAddPrime89(r,x,A,B);
  MultAddPrime89(r,x,r,C);
  MultAddPrime89(r,x,r,D);
  return Mod64Prime89(r);
} // 59*3 + 7 = 184 instructions
```

The code uses 184 instructions including 18 multiplications. Note that `Mod64Prime89` only produces the 64 least significant bits of the answer.

B.6 Second moment estimation

```
#define NumCounters 32768      // (1<<15)
INT64 Counters[NumCounters];

// precomputed tables whose hash strings only use 15 least significant bits
INT64 *T0, *T1, *T2;
```

```
inline void StreamUpdate2nd(INT32 ipaddr, INT32 size) {
  Counters[Table32(ipaddr,T0,T1,T2)] += size;
} // 3 instructions plus those in Table32.

double StreamEstimate2nd() {
  int i;
  INT64 c;
  double sum = 0, sqsum = 0;
  for (i = 0; i < NumCounters; i++) {
    c = Counters[i];
    sum += c;
    sqsum += c*c;
  }
  // sqsum*NumCounters/(NumCounters-1) - sum*sum/(NumCounters-1)
  return sqsum + (sqsum - sum*sum)/(NumCounters-1);
}
```

The code for updating the counters adds 3 instructions to those in Table32, including one additional lookup.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. System Sci.*, 58(1):137–147, 1999.

[2] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th ICALP, LNCS 2380*, pages 693–703, 2002.