Stream Algorithms: Hyperloglog

Richard Anderson
February 9, 2021
Announcements

• HW6
  • Implement count min and test on provided dataset

• Course material – streaming algorithms
  • Today: Count number of distinct elements: Hyperloglog
  • Thursday: Determine the second moment
    • $\Sigma f_x^2$
    • Alon, Matias, Szegedy
Algorithms for data streams

- Data items received one at a time, N is number of items received
- Computation performed on each data item
- Memory is limited to being much less than N
  - Memory in thousands
  - Data in millions
- Motivation
  - Large scale computation
  - Estimates are often good enough
Results so far

- Easy things that require little memory
  - Computing the maximum, sample a random element
- Impossible things (provably require $\Omega(n)$ memory)
  - Is there an element with frequency $\geq n/3$, find the median
- Identification and approximate estimates of high frequency items
Warmup

- Suppose we want to track $j$ smallest items in a stream
  - Space complexity
  - Time complexity
Count distinct problem

- FB Distinct visitors per week
- How many distinct IP addresses accesses a website
Solutions

• Sort, remove duplicates
  • sort –u foo.txt | wc
• Hashing
• Bloom Filter

```csharp
public int CountUnique(List<string> strings){
    Dictionary<string, int> dict = new Dictionary<string, int>();
    foreach (string str in strings)
        dict.TryAdd(str, 1);
    return dict.Count;
}
```
Estimator

• Assign each distinct element a random value, in [0, 1)
  • By hashing, of course
  • Each copy of the same item has the same hash value
• Compute the minimum value: min_val
  • Only remember a single data item
• If the number of distinct items is K, the expected value of min_val is 1/K
• Report the estimate 1/min_val
Improve the estimator

• Track the J smallest values
  • J values need to be maintained
  • Compute \( \text{min}_\text{val}_j \)
• Expected value of \( \text{min}_\text{val}_j \) is \( J/K \)
• Report estimate of \( J/\text{min}_\text{val}_j \)

• Probabilistically, this is a far more robust estimator
Hyperloglog

• Estimate number of distinct elements
  • Estimate cardinalities of $> 10^9$ with accuracy 2% using 1.5 kB of memory
• Derived from a 1984 theoretical result
• Suggested practical applications in early 2000s
• Derivatives are now used as a practical tool by large internet companies
  • `APPROX_COUNT_DISTINCT` in BigQuery
  • Reddit, to count unique views of posts

• From this discussion
  • Key ideas
  • Basic algorithm
  • But the analysis still contains some magic
Log Log idea

• What is the probability that a random number has exactly k consecutive one’s (in binary) in low order bits
  • 1010001010010100100100100111
  • Define $\rho(x)$ as consecutive ones at end of hash(x)

• Estimate the cardinality of $\{x_1, x_2, \ldots, x_M\}$ as $2^Q$ where
  $Q = \max\{\rho(x_1), \rho(x_2), \ldots, \rho(x_M)\}$

• $\rho(x) = k$ with probability $2^{-(k+1)}$

• Q is an estimate of log M, so Q can be stored in log log M bits
Intuition

• If you have one million items, one of them is going to have a hash that ends in: 01111111111111111
Strengthening the estimate

• Risk of over estimating with a very unlikely hash
• If we have $2^k$ distinct items, we expect to have items of that have hashes that end with $j$ consecutive 1’s for $j \leq k$
• We will need to track all the $\rho$ values, which we will do by keep a bitwise-or
Some bit hacking

- $r(x)$ is the number of trailing 1s in the binary representation of $x$
- $R(x) = 2^{r(x)}$
- $R(x) = \sim x \& (x+1)$
# Probabilistic Counting Trace

<table>
<thead>
<tr>
<th>x</th>
<th>( r(x) )</th>
<th>( R(x) )</th>
<th>sketch</th>
</tr>
</thead>
<tbody>
<tr>
<td>011000100110001111010011110111011</td>
<td>2</td>
<td>100</td>
<td>000000000000000000000000100</td>
</tr>
<tr>
<td>011001110010001100011111000000101</td>
<td>1</td>
<td>10</td>
<td>000000000000000000000000110</td>
</tr>
<tr>
<td>000100010001110001101101101100111</td>
<td>2</td>
<td>100</td>
<td>000000000000000000000000110</td>
</tr>
<tr>
<td>010001000111011110000000111011111</td>
<td>5</td>
<td>100000</td>
<td>000000000000000000000000100110</td>
</tr>
<tr>
<td>011010000010110001011100100100010</td>
<td>0</td>
<td>1</td>
<td>000000000000000000000000100111</td>
</tr>
<tr>
<td>001101111011000000000101001010101</td>
<td>1</td>
<td>10</td>
<td>000000000000000000000000100111</td>
</tr>
<tr>
<td>011010001100011101010101111111100</td>
<td>0</td>
<td>1</td>
<td>000000000000000000000000100111</td>
</tr>
<tr>
<td>000110000100010111110010111011011</td>
<td>3</td>
<td>1000</td>
<td>00000000000000000000000011111</td>
</tr>
<tr>
<td>000110001100111001110010000111111</td>
<td>6</td>
<td>100000</td>
<td>000000000000000000000000101111</td>
</tr>
<tr>
<td>010001011100010010110100111111100</td>
<td>0</td>
<td>1</td>
<td>000000000000000000000000101111</td>
</tr>
</tbody>
</table>

\[
R(\text{sketch}) = 10000_2 = 16
\]
Probabilistic Counting

- Returns the smallest value not seen
- It can be shown that this off by a factor of 0.77351
  - Established mathematically, and verified experimentally
- Typically off by a binary order of magnitude

```java
public long R(long x) {
    return ~x & (x+1);
}

public long estimate (iterable<string> stream) {
    long sketch;
    for (s : stream)
        sketch |= R(Hash(s));
    return R(sketch);
}
```
Next idea – M independent experiments

• M independent hash functions and average: work, but expensive

• Stochastic averaging
  • Divide stream into $2^m$ independent streams
  • Use probabilistic counting on each stream, yielding $2^m$ sketches
  • Compute mean = average number of trailing bits in each sketch
  • Return $2^{\text{mean}} / .77531$
Constructing the independent experiments

• Assume we have a j bit has function (so hashing to \([0..2^{j-1}]\))
• Use the first m bits to divide into substreams
• Use the remaining j-m bits as a hash function (into \([0..2^{j-m-1}]\))

1010100010101010001010
0100010101010101001111
0101010101011101110011
1001010100110010010001
0010010101000100100010
1101000101000101001011
1010001011101101110010
1001001011101101110101
1010110111011000100111
Probabilistic Counting Algorithms

• Flajolet-Martin, 1983
• Use of M words to achieve relative accuracy of $0.78/\sqrt{M}$
• Validated through experimentation
  • Theory doesn’t answer questions such as performance with real hash functions or what are the implementational constants
• Many versions now available with modified techniques
  • E.g., different mechanisms for averaging estimates across substreams, harmonic means vs. geometric means