Abstract

The count-distinct problem, or estimation of the number of unique elements in large dataset, has drawn some attention lately. The elements might represent IP addresses of packets passing through a router, unique visitors to a website, elements in a large database, motifs in a DNA sequence, or elements of RFID/sensor networks [1].

HyperLogLog is a probabilistic algorithm for the count-distinct problem approximating the number of unique elements in a multiset (a multiset is a generalization of the concept of a set that, unlike a set, allows multiple instances of the multiset's elements [2]). Calculating the exact cardinality of a multiset requires an amount of memory proportional to the cardinality, which is impractical for very large data sets. Further, the exact calculation is not easy to parallelise. Probabilistic cardinality estimators, such as the HyperLogLog algorithm, use significantly less memory than this, at the cost of obtaining only an approximation of the cardinality. The HyperLogLog algorithm is able to estimate cardinalities of >10^9 with a typical accuracy of 2%, using 1.5kB of memory [3].

In this paper we present a summary of the HyperLogLog algorithm, highlight a selection of applications where it proves useful and provide with a Java code implementation. Further, while the full theory behind the algorithm and the proof of the details is beyond the intended scope, we point out a few interesting intuitive considerations by the authors.

HyperLogLog algorithm

The basic idea of the HyperLogLog algorithm is the observation that the cardinality of a multiset of uniformly-distributed random numbers can be estimated by calculating the maximum number of trailing zeros in the binary representation of each number in the set. If the maximum number of trailing zeros observed is \( n \), an estimate for the number of distinct elements in the set is \( 2^n \) [3]. Intuitively we can understand this by looking at the following table. First column is a 8-bit random hash pattern (x indicates that the bit can be set to either 0 or 1), second column is number of trailing zeros and the third column indicates the probability of finding such a bit pattern.
Table 1: 8-bit random hashes and their probabilities of occurrence. x denotes any bit value (0 or 1).

<table>
<thead>
<tr>
<th>Hash bit pattern</th>
<th># trailing zeros</th>
<th>probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxxxx1</td>
<td>0</td>
<td>50%</td>
</tr>
<tr>
<td>xxxxxx10</td>
<td>1</td>
<td>25%</td>
</tr>
<tr>
<td>xxxx100</td>
<td>2</td>
<td>12.5%</td>
</tr>
<tr>
<td>xxx1000</td>
<td>3</td>
<td>6.25%</td>
</tr>
<tr>
<td>xxx10000</td>
<td>4</td>
<td>3.125%</td>
</tr>
<tr>
<td>xx100000</td>
<td>5</td>
<td>1.5625%</td>
</tr>
<tr>
<td>x1000000</td>
<td>6</td>
<td>0.78125%</td>
</tr>
<tr>
<td>10000000</td>
<td>7</td>
<td>0.390625%</td>
</tr>
</tbody>
</table>

In the HyperLogLog algorithm, a hash function is applied to each element in the original multiset (an element can be a string, a number, etc.), to obtain a multiset of uniformly-distributed random numbers with the same cardinality as the original multiset. The cardinality of this randomly-distributed set can then be estimated using the algorithm of counting zeros in the binary representation above.

**Accuracy**

The simple estimate of cardinality obtained using the algorithm above has the disadvantage of a large variance. In the HyperLogLog algorithm, the variance is minimised by splitting the multiset into numerous subsets, calculating the maximum number of trailing zeros in the numbers in each of these subsets, and using a harmonic mean to combine these estimates for each subset into an estimate of the cardinality of the whole set. The splitting is achieved by assigning the first \( p \) bits of the hash to select a subset (Note: \( p \) needs to be large enough in order to obtain sufficient averaging over subsets - Flajolet et. al. suggest the values \( p \in [4, 16] \) corresponding to \( m = 2^p \) subsets). The remaining \( (32-p) \) bits are used as the actual hash to estimate the cardinality. Note that long (64 bit) integers can be used if cardinalities with order of magnitude larger than \( 10^9 \) are expected.

Flajolet et. al. use harmonic means for averaging. As noted by Chassaing and Gérin [6], such means have the effect of taming probability distributions with slow-decaying right tails, and here they operate as a variance reduction device, thereby appreciably increasing the quality of estimates. The algorithm needs to maintain a collection of registers, each of which is at most \( \log_2 \log_2 N + O(1) \) bits, when cardinalities \( \leq N \) need to be estimated.

The main conclusion of Flajolet et. al. is that cardinalities till values over \( N = 10^9 \) can be estimated with a typical accuracy of 2% using 1.5kB of storage [4]. The values of the estimate returned are expected to be approximately Gaussian, due to an averaging effect and the Central Limit Theorem (this property is supported by the simulations in Ref. [4]). We discuss the exceptions to this case in the following subsection concerning the small and large range correction formulas.
Small range corrections

In Ref. [4], extensive simulations demonstrate that the asymptotic regime is practically attained at the cardinality value \( n = (5/2)m \), when \( m \geq 16 \). In contrast, for \( n < (5/2)m \), nonlinear distortions start appearing. Consider the probabilistic properties of random allocations, i.e., let us have \( m \) bins into which \( n \) balls are thrown at random. Let us define the random variable \( X \) as \( X_i = 1 \) if bin \( i \) is empty and \( X_i = 0 \) if bin \( i \) is occupied. The probability that bin \( i \) is empty after the first ball has been thrown is \( \frac{m-1}{m} \). After \( n \) balls have been thrown the probability that bin \( i \) is still empty is \( \left( \frac{m-1}{m} \right)^n \). Thus, the expected number of empty bins is given by

\[
E(X) = V = m \left( \frac{m-1}{m} \right)^n = m \left( 1 - \frac{n}{mn} \right)^n = m \left( 1 - \frac{1}{n} \right)^n \approx me^{-\mu}
\]

in the limit where \( n \) grows large. Inverting the equation an estimate for \( \mu \). Thus the expected value of distinct records \( C \) is obtained by

\[
C = m \log(m/V)
\]

Large range corrections

Hash collisions occur for large \( n \), namely, when \( n \) approaches \( 2^L \), where \( L=32 \) for 32-bit hash (number of distinct possible hashes). The same probabilistic allocation model can be used for hash collisions. Expected number of occupied bins (bits with value 1) can be estimated by the logic from previous subsection as \( 2^L (1 - e^{-\Lambda}) \) where \( \Lambda = n/2^L \). Inverting the equation the approximate equation for \( n \) is obtained (and used by Flajolet et al. [4]):

\[
n = -2^L \log(1 - C/2^L)
\]

In the next section, we describe the details of implementation of the algorithm.

Algorithm description

The algorithm works as follows (note that we describe the 32 bit hash implementation of Flajolet et al.). As noted in [5], when using a 64 bit hash function there’s no need for large range correction in practise (hash collisions don’t take place before the cardinality is of the order of magnitude of \( 2^{64} \approx 10^{19} \)).

Initialization

Let \( h : D \rightarrow \{0, 1\}^{32} \) hash data from \( D \) to binary 32–bit words.
Let \( \rho(s) \) be the position of the leftmost 1-bit of \( s \): e.g., \( \rho(1 \cdots) = 1, \rho(0001 \cdots) = 4, \rho(0^k) = K+1 \).
Define \( a_{16} = 0.673; \ a_{32} = 0.697; \ a_{64} = 0.709; \ a_m = 0.7213/(1 + 1.079/m) \) for \( m \geq 128 \).
Let \( M \) be a multiset of items \( v \) from domain \( D \).
Aggregation

Assign \( m = 2^p \) with \( p \in [4, 16] \).
Initialize a collection of \( m \) registers \( R_i = 0 \) for \( i \in [1, m] \).

For each \( v \in M \) do:
- set \( x = h(v) \);
- set \( u = b_1 b_2 b_3 ... b_p \) (integer determined by first \( p \) bits of \( x \));
- set \( w = b_{p+1} b_{p+2} b_{p+3} ... b_L \);
- set \( R(u) = \max(R(u), \rho(w)) \);

Computation

Compute the “raw” HyperLogLog cardinality estimate as \( C = a_m m^2 (\sum_{i=1}^{m} 2^{-R(u)})^{-1} \).

If \( C < \frac{1}{2} m \) then \{small range correction\}
- Let \( V \) be the number of registers equal to 0;
- If \( V \neq 0 \) then set \( C^* = m \log(m/V) \) else set \( C^* = C \);

If \( C \leq \frac{1}{16} 2^{32} \) then \{intermediate range—no correction\}
- Set \( C^* = C \);

If \( C > \frac{1}{16} 2^{32} \) then \{large range correction\}
- Set \( C^* = 2^{32} \log(1 - C/2^{32}) \);

Return cardinality estimate \( C^* \) with typical relative error \( \pm 1.04/\sqrt{m} \).

Implementation

The full Java implementation of HyperLogLog algorithm is available in the Appendix. For parallel execution the merge functionality is also implemented - after each individual HyperLogLog instance has processed its respective set of data, the merge can be used to capture the overall cardinality estimate over all data sets. We ran this on a single laptop with parallel implementation using the Java ThreadPoolExecutor and sample data sets in order to test the merge functionality of the code.

Summary

In this paper we discussed probabilistic cardinality estimation and specifically, the HyperLogLog algorithm. We highlighted a few theoretical insights of the derivation of the algorithm and its boundaries. We also presented a Java implementation of the code with a variety of 64 bit hash functions. We tested the code by generating 10 million random strings (alpha-numeric characters) of length 4-8 and comparing the estimate given by the HyperLogLog algorithm (with the parameter \( m = 12 \) ) to the exact counts (using a hash map) and found its accuracy consistent with the predictions.
References

1. Count-distinct problem
2. Multiset
3. HyperLogLog
6. Amazon Redshift APPROXIMATE COUNT DISTINCT statement.

Appendix - source code

[Source code highlighting by http://markup.su/highlighter/]

```java
package com.done.algorithm;

import java.nio.ByteBuffer;
import java.security.MessageDigest;

/**
 * Implementation of HyperLogLog algorithm. Original paper:
 * @author Joonas Asikainen <joonas.asikainen@d1-solutions.com>
 */
public class HyperLogLog {

    public static enum HashAlgorithm {
        MD2,
        MD5,
        SHA1,
        SHA256
    };

    private final int p;
    private final int m;
    private final long c;
    private final double am;
    private final int[] rhos;
    private final HashAlgorithm hashAlgorithm;
    private final MessageDigest md;
    private int count = 0;

    public HyperLogLog(int p) throws Exception {
        this(p, HashAlgorithm.MD5);
    }
```
public HyperLogLog(int p, HashAlgorithm hashAlgorithm) throws Exception {
    if (p < 4 || p > 16) {
        throw new Exception("Parameter p out of range [4, 16]" simplest);
    }
    this.p = p;
    this.m = 0x01 << p;
    this.c = (long) 0x01 << (64 - p);
    this.rhos = new int[m];
    switch (m) {
        case 16:
            am = 0.673;
            break;
        case 32:
            am = 0.697;
            break;
        case 64:
            am = 0.709;
            break;
        default:
            this.am = 0.7213 / (1 + 1.079 / m);
    }
    this.hashAlgorithm = hashAlgorithm;
    switch (hashAlgorithm) {
        case SHA1:
            this.md = MessageDigest.getInstance("SHA-1");
            break;
        case SHA256:
            this.md = MessageDigest.getInstance("SHA-256");
            break;
        case MD2:
            this.md = MessageDigest.getInstance("MD2");
            break;
        default:
            this.md = MessageDigest.getInstance("MD5");
    }
}

public int aggregate(String value) throws Exception {
    final long hsh = getHash(value);
    final int idx = (int) (hsh & (m - 1));
    final long reg = (hsh >>> p);
    final int rho = Long.numberOfTrailingZeros(reg & (c - 1)) + 1;
    rhos[idx] = rho > rhos[idx] ? rho : rhos[idx];
    return (++count);
}

public int getCount() {
    return count;
}

public void reset() {
}
```java
for (int i = 0; i < m; i++) {
    rhos[i] = m;
}

public double getCardinality() {
    int v = 0;
    int min = rhos[0];
    int max = rhos[0];
    double raw = 0;
    for (int i = 0; i < m; i++) {
        if (rhos[i] == 0) {
            v++;
        }
        min = rhos[i] < min ? rhos[i] : min;
        max = rhos[i] > max ? rhos[i] : max;
        raw += java.lang.Math.pow(2.0, -(double) rhos[i]);
    }
    raw = (double) (am * java.lang.Math.pow((double) m, 2.0)) / raw;
    final double result;
    if (2 * raw <= 5 * m) {
        if (v == 0) {
            result = raw;
        } else {
            result = ((double) m * java.lang.Math.log((double) m / (double) v));
        }
    } else {
        result = raw;
    }
    return java.lang.Math.floor(result);
}

public int merge(HyperLogLog[] others) throws Exception {
    int result = 0;
    for (HyperLogLog other : others) {
        result += merge(other);
    }
    return result;
}

public int merge(HyperLogLog other) throws Exception {
    if (!isCompatible(other)) {
        throw new Exception("Incompatible HyperLogLog instances (this.p = " +
        this.p + " vs. other.p = " + other.p + ").");
    }
    for (int i = 0; i < m; i++) {
        final int rho = other.getRho(i);
        if (this.rhos[i] < rho) {
            this.rhos[i] = rho;
        }
    }
    return 1;
}
```
private boolean isCompatible(HyperLogLog other) {
    return (this.p == other.getP())
        && this.hashAlgorithm == other.getHashAlgorithm());
}

private int getP() {
    return p;
}

private HashAlgorithm getHashAlgorithm() {
    return hashAlgorithm;
}

private long getHash(String value) {
    md.reset();
    final byte[] bytes = md.digest(value.getBytes());
    return ByteBuffer.wrap(bytes).getLong();
}

private int getRho(int i) {
    return rhos[i];
}

@Override
public String toString() {
    return "HyperLogLog\" + p + \", m=\" + m + \", c=\" + c + \", am=\" + am + \\
        \", hashAlgorithm=\" + hashAlgorithm + \", count=\" + count + \", cardinality=\" + \\
        getCardinality() + '\";"
}
}