HyperLogLog in Practice

Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm

Stefan Heule
Marc Nunkesser
Alexander Hall

Contents

i. Problem Definition & Applications

Problem Description

- Count-distinct problem or Cardinality Estimation Problem
- Formal Definition:
  - Given a stream of N elements with repetitions
  - Find the number of distinct elements D
  - Using M storage units, where M << D
- Memory requirement tends to be proportional to the cardinality
- Accuracy depends on the algorithm and the underlying data structure
- Set Membership can be inferred by changes in cardinality

Exact Solution

- Store unique elements in Array Data Structure → $A_j = X, \; j \in [0, n]$
- Advantages:
  - Accuracy
  - Cardinality is the Array length → $O(1)$
- Disadvantages:
  - Slow Lookup → $O(n)$, insertion time depends on cardinality
  - Required memory scales with cardinality → $\sum A_j$
- Possible Improvements:
  - Use lossless compression in order to reduce data size
  - Use Set / Tree data structure (B-tree) with lookup time $O(1) \; / \; O(logn)$

Approximate Solution

- Define:
  - Cardinality → n
  - Hashing function with uniform distribution $h(x) \rightarrow D, D \epsilon [0, S]$
- Add each hash value to a Set Data Structure
  - Lookup time → O(1)
  - Cardinality calculation → O(1)
- Collision probability depends on hash value range
- Accuracy is retained if: S >> n
- Memory requirements are still proportional to cardinality

Problem Domain

- Time-series data streams lead to large, transient datasets
- Not feasible or otherwise necessary to maintain the exact original dataset:
  - Cardinality is unknown
  - Memory constraints
  - Sliding window calculations
- Small errors can be tolerated:
  - Eventual consistency
  - Multiple data sources
- Latency versus Accuracy

Applications: Web Analytics & Security

Problem Definition

- Count unique sessions or pageviews per device/browser for given time periods.
- Detect possible spamming or malicious requests.

Solution Modelling

- Combine session ID and URL → Request ID.
- Count unique Request ID.
- Low cardinality of Request ID and high request rate → throttling?
- Calculation happens over a sliding time window.

Applications: Network Security

Problem Definition

- Detect and possibly block malicious network traffic[6]
- Very high data rate: 40+ Gb/s
- Packet inter-arrival time must remain low (~8ns)
- Attack detection is not the main role of the router, memory should remain low

Solution Modelling

- Flow is sequence of packets identified by the tuple
  `(Source-IP, Source-Port, Destination-IP, Destination-Port, Protocol)`
- Calculate the total packets using different destination ports over a sliding time window

ii. Cardinality Estimation Algorithms

Concepts & Theory - Part I

- Multiset
  - Modification of Set that allows multiples instances of its elements.
  - Number of instances per element is named *multiplicity*.
  - Cardinality is the number of unique elements.
- Uniform distribution:
  - Probability of random variable within the interval **[a,b]** is independent of the variable.
- K-th Order Statistic
  - Equal to kth-smallest value of a statistical sample
  - First order statistic is the sample minimum
- Probabilistic Algorithms / Data Structures
  - Employ randomness as part of their logic
  - Examples: Bloom Filter, Linear Counting, Skip List, HyperLogLog

Concepts & Theory - Part II

- Estimator
  - Rule for calculating an estimate of a given quantity based on observed data
- Bias
  - Expected relative error of an estimator
  - **bias = 0.01** means an overestimation of **1%** compared to the exact value
- Standard Error
  - Standard deviation of the ratio of estimated quantity compared to the exact value
- Harmonic Mean
  - Type of average function, better suited for average of rates
  - Expressed as the reciprocal of the arithmetic mean of the reciprocals

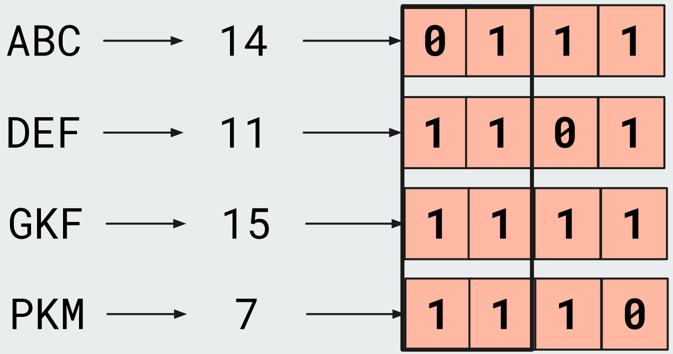
$$H = (\sum x_i^{-1} / n)^{-1}$$

Cardinality Observables - Part I

- Consider the uniformly distributed hashed values of a Multiset (S-values)
- **Bit-pattern Observables**
  - Series of bits (bit strings) of predefined length (e.g. 32-bit)
  - Pattern of bits occurring at the beginning of the S-values
  - Observe the minimum index of a 1-bit
  - By convention we use **LSB order**

Cardinality Observables - Part II

- Assume hashing function of 4 bits, LSB order in bit strings:
    - **50%** of hashed values will display the pattern **1BBB**
    - **25%** of hashed values will display the pattern **01BB**
- Hashing **4** unique elements, it is likely that at least one S-value will start with **01**
- Inverting the expectation:
    - If the first 1-bit index is **2**, it is likely we encountered **~4 unique values**.



Cardinality Observables - Part II

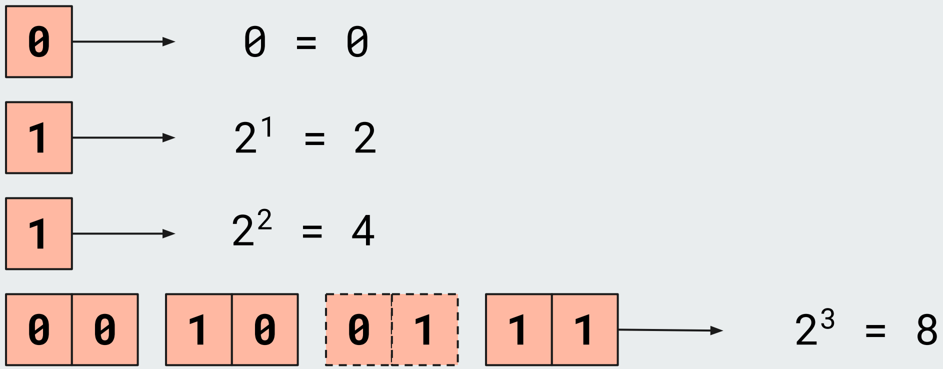
- **Order-Statistic Observables**
  - Hash function provides uniformly distributed real values in **[0, 1]**
  - The minimum value in an ideal multiset does not depend on:
    - Replication structure of the data
    - Ordering
  - Indication on the number of distinct values of the multiset
    - The minimum of independent uniform values on [0, 1] has more chances of being small if cardinality is large
  - MinCount[9]

Available Probabilistic Algorithms

- Flajolet - Martin (1985)
- Linear Counting (Whang et al - 1990)
  - Bit-map of size *m* - initialized to zero values
  - Hash function generates bit map address which is set to one
  - Cardinality is computed by: `n = -m * ln(ZeroCount / m)`
- LogLog (Durand & Flajolet - 2003)
  - Improvements to the Flajolet - Martin algorithm
- HyperLogLog (Flajolet et al - 2007)

Probabilistic Counting - An Example

- Approximate Counting (Morris - 1977)
  - Count large number of events using small amount of memory
  - Counts powers of two - only stores the exponent
  - Incremented by Pseudo-random event: “coin flip” times the current counter value



iii. HyperLogLog and Improvements

Introduction to HyperLogLog - Part I

- Computational Algorithm presented in Flajolet et al (2007)
- Contains the analytical mathematical proofs:
  - The estimator is asymptotically unbiased.
  - The standard error constant values used for bias correction.
- Applies hash function on the original data stream
- Uses bit-pattern observables
- Divides the hash values input stream into m substreams
  - Specific bit interval is used for partitioning
- Emulates m experiments in parallel with a single hash function
  - As opposed to using multiple hash values per original input element
- Estimate of the cardinality is a suitable average of the substream observables
  - Quality should improve due to averaging effects as substreams increase
- Solution is named **Stochastic Averaging**

Introduction to HyperLogLog - Part II

- If each of a m random variables has standard deviation σ
    - Their arithmetic mean has deviation $\sigma / \sqrt{m}$
- Accuracy Characteristics (m = 2048, hashing to 32 bit values):
    - Cardinalities close to and exceeding one billion (10^9)
    - Typical accuracy of 2% error → $1.04 / \sqrt{m}$
    - 1.5 kilobyte of storage
- Substream counters are also referred as registers
- Use of harmonic mean since it is less sensitive to outliers.
- Suggests a correction formula depending on estimate range (E):
    - Small Range ($\leq 2.5m$): Use Linear Counting
    - Large Range ($> 2^{32} / 30$)

HyperLogLog - Computational Algorithm (I)

- Let $h : D \to [0, 1] \equiv \{0, 1\}^{\infty}$ hash data from domain D to the binary domain.
- Let $\rho(s)$, for $s \in \{0, 1\}^{\infty}$, be the position of the leftmost 1-bit ($\rho(0001 \cdots) = 4$).
- Algorithm HYPERLOGLOG (input M : multiset of items from domain D).
- Assume $m = 2^b$ with $b \in Z > 0$
- Initialize a collection of m registers, `M[1], . . . , M[m]`, using a value of $-\infty$
- `for v in M:`

```
    x ← h(v)
    j ←1 + x_1 x_2 ... x_b
    w ← x_{b+1} x_{b+2} …
    M[j] = max(M[j], ρ(w))
```

$$x \leftarrow h(v)$$
$$j \leftarrow 1 + x_1 x_2 \ldots x_b$$
$$w \leftarrow x_{b+1} x_{b+2} \ldots$$
$$M[j] = \max(M[j], \rho(w))$$

- Compute indicator function $Z = (\sum 2^{-M[j]})^{-1}$
- Compute estimator $E = \alpha_m m^2 Z$ with α_m as a bias-correction constant.

HyperLogLog - Computational Algorithm (II)

- Assume hashing function *h* with 8-bit output range and 4 registers ($m = 2^2$, $b = 2$)
- Let v = ‘abcdef’
- x = h(v) = 142 = 01110001
- j = ‘**01**110001’$_{1..2}$ = ‘01’ = 2
- w = ‘01**110001**’$_{3..8}$ = ‘110001’
- ρ(w) = ρ(‘**1**10001’) = 1
- M[2] = 1
- x = h(‘ghij’) = 178 = **01001**101
- j = 2, ρ = 3, M[2] = max(M[2], 3), M[2] = 3

Example Implementation - Part I

```
def compute_hash(value):
  return zlib.crc32(value.encode()) % (1 << 32)

def to_binary_representation(integer):
  return '{0:b}'.format(integer)

def to_lsb_binary(integer):
  returnto_binary_representation(integer)[::-1]

def convert_to_base_10(value):
  return int(value, 2)
```

```
def lsb_bit_range(number, start, end):
  return to_lsb_binary(number)[start:end]

def indicator_function(registers):
  return 1. / sum(2 ** -register for
register in registers)

def find_first_one_bit_index(lsb_binary):
  return lsb_binary.index('1') + 1
```

Example Implementation - Part II

```
def add(registers, b, value):
    x = compute_hash(value)
    j = convert_to_base_10(lsb_bit_range(x, 0, b)[::-1])
    w = lsb_bit_range(x, b, -1)
    rho = find_first_one_bit_index(w)
    changed = rho > registers[j]
    registers[j] = max(rho, registers[j])
    return changed

def count(registers):
    a_m = 0.72134
    Z = indicator_function(registers)
    return a_m * (m ** 2) * Z

def merge(hll_1, hll_2):
    return map(max, hll_1, hll_2)
```

Example Implementation - Part III

```
def initialize_hll(b):
    m = 2 ** b
    return [-math.inf] * m

B = 11
hll_registers = initialize_hll(B)

for n in range(1, 10**7 + 1):
    if add(registers=hll_registers b=B, value=str(n)):
        cardinality = count(hll_registers)
        if n % 100 == 0:
           print(
               'n={}, hyperloglog count={}, error={}'.format(
               n, cardinality, abs(n - cardinality) / n)
           )
```

HyperLogLog - Operations and Complexity

- Add
  - Time Complexity $O(1)$
- Merge
  - $HLL_{union}[j] = Max(HLL_1[j], HLL_2[j])$
  - Dependent on register count: Complexity $O(m)$
- Count
  - Time Complexity $O(m)$
- Space
  - $O(m)$
  - Total size depends on the hashing function output range S
  - Register Size $\log_2(S\text{-bits} - \log_2 m)$

HyperLogLog++ - Part I

- HyperLogLog weaknesses:
  - As cardinality approaches 2^L, where L is the number of hash output bits, number of collisions increases.
  - Zero cardinality for n `<< mlogm`
- Improvements to the original HyperLogLog algorithm
  - Use a 64-bit hashing function, instead of 32-bit
  - Initialize registers to zero to avoid zero cardinality for n `<< mlogm`
  - Large range correction no longer necessary due to the 64-bit range shift
  - Empirical bias correction
  - Sparse/dense representation
- Improved Accuracy for cardinalities larger that 2^{32}
- Small increase in memory requirements

HyperLogLog++ - Part II

- **Small Cardinality Estimation**
  - Linear Counting below $2.5m$
  - Empirical bias correction until $5m$
    - Calculation of the mean difference of raw estimate minus the cardinality
    - Use k-nearest neighbor interpolation
      (200 cardinalities as interpolation points)
- **Sparse Representation**
  - Stores pairs $(index, \rho(w))$ with size threshold $6m$ bits
  - Represented as sorted list of integers, by concatenating bit patterns:

    `<INDEX><RHO>`
  - Variable-Length Encoding
  - Difference Encoding
    - Store consecutive differences for the second element and so forth

Sliding HyperLogLog

- Problem: allow calculations over a sliding time window for unknown intervals
- Store a List of Possible Future Maxima for each register
- LPFM entries `[Timestamp, R`$_i$`]`
  - R_i is the `ρ(w)` value of the HyperLogLog algorithm.
  - Timestamp is the Unix epoch time (seconds)
- Requires approximately 5 additional bytes per LPFM element.
- Query: estimate cardinality given an interval **[t - w, t]**
  - Retrieve all LPFM entries for which the timestamp is within the interval.
  - Maintain the maximum R_i value for each LPFM.
  - Apply stochastic averaging on the computed maximum register values.
- Remove LPFM entries based on the timestamp as the window advances

iv. Implementations and Examples

Elasticsearch

- Elasticsearch Cardinality Aggregations
- Cardinality Aggregation uses HyperLogLog++ (based on the Google paper).
- Example: Count unique IPs

```
POST /logs/_search?size=0
{
    "aggs" : {
        "type_count" : {
            "cardinality" : {
                "field" : "source_ipv6"
            }
        }
    }
}
```

Elasticsearch

- Configurable precision with the `precision_threshold` option
- Increasing the threshold requires more memory, in order to improve accuracy.
- The threshold defines a unique count below which counts are expected to be close to accurate.
  - Maximum supported value is 40000, with a default value of 3000.
  - Memory: `Threshold * 8 bytes`
- Pre-computed hashes:
  - Value hashing can be performed using an ES plugin.
  - Beneficial for large string and high-cardinality fields.
  - Field hash value is stored in the document.
  - Aggregation is applied on the hash field.

Redis - Part I

- Redis is an in-memory data structure store
- HyperLogLog structures are available since version **2.8.9**[10]
- Largely based on HyperLogLog++ with further improvements
- **Sparse representation**, optimized for storing large number of registers set to zero:
    - Lossless compression with run-length encoding
- **Dense representation:**
    - Redis string of 12288 bytes in order to store **16384 6-bit counters**
- 6-bit registers
    - With 64-bit hash values ($m = 2^{14}$) 50 bits remain that require $\mathbf{2^6}$ storage bits
- Maximum memory is **12 kB**, standard error **0.81%** (since 16384 registers are used)
- 64-bit hash function (MurmurHash2)

Redis - Part II

- Introduced curve fitting resulting in a four-order polynomial for error correction in range 40960-72000
  - Linear counting: `Cardinality = m * ln(m / total-registers-with-initialization-value)`
  - Finally using τ raw estimator (Ertl - 2017)
- `PFADD key element [element ...]`
  - Adds the given elements to the HLL structure.
  - Returns 1 if any of the internal registers was modified.
- `PFCOUNT key [key]`
  - Returns the approximated cardinality of the specified HLL or the combined cardinality by merging multiple HLL structures.
- `PFMERGE destkey source-key [source-key …]`
  - Merge multiple HLL structures to a new one.

Druid

- Column-oriented, distributed data store
  - High performance analytics data store for event-driven data
- Cardinality Aggregator
- Fast, Cheap, and 98% Right: Cardinality Estimation for Big Data
  - Murmur 128 hashing function
  - Stores intermediate HLL format in a column
- How We Scaled HyperLogLog: Three Real-World Optimizations
  - Register compaction:
    - Offset + positive differences in the registers
  - Faster cardinality calculations was to use lookups for register values
  - Dense/sparse storage

Libraries

PostgreSQL extension adding HLL as native datatype:
https://github.com/citusdata/postgresql-hll

C#: https://github.com/Microsoft/CardinalityEstimation

Golang: https://github.com/axiomhq/hyperloglog

Python: https://github.com/ekzhu/datasketch

Erlang: https://github.com/GameAnalytics/hyper

References

[1] HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm
[2] Probabilistic Counting Algorithms for Data Base Applications
[3] Streaming Algorithm
[4] Elasticsearch: The Definitive Guide [2.x] » Aggregations » Approximate Aggregations » Finding Distinct Counts
[5] Approximate counting algorithm
[6] How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?
[7] Sliding HyperLogLog: Estimating cardinality in a data stream
[8] HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm
[9] Order statistics and estimating cardinalities of massive data sets
[10] Redis new data structure: HyperLogLog