Data Stream Algorithms III

Joaquim Madeira

Version 0.1 – December 2018

Overview

- Counting the Number of Distinct Items

Set of distinct items

- Given a stream: $\sigma = \langle a_1, a_2, \dots, a_m \rangle$
- The set of distinct items is

$$D = \{ j : f_j > 0 \}$$

- f_j is the number of occurrences of item j

$$f_1 + f_2 + \dots + f_n = m$$

Number of distinct items

- The DISTINCT-ELEMENTS problem
  - $\#D$ = ?
- Cardinality estimation
  - Output a (ϵ, δ)-approximation of $\#D$
- It is impossible to solve this problem in sublinear space, restricted to using
  - A deterministic algorithm – i.e., $\delta = 0$
  - An exact algorithm – i.e., $\epsilon = 0$
- Use a randomized approximation alg. !!

Application examples

- Web site gathering statistics on how many unique users it has seen in each given month
  - Amazon : unique login name for registered users
  - Google : IP address from which a query is issued
  - Yahoo! : count users viewing each of its pages
    - A data stream for each page

Application examples

- How many different Web pages does each user request in a week ?
- How many distinct items have been sold in the last week ?
- How many different words are found among the Web pages being crawled at a site?
  - Low or high numbers might indicate fake pages

What if the data set is known ?

- We can do an exact count !
- Which trivial approaches do you know ?
- Memory space ?
- Running time ?

Bitmap and bit counting

- Bitmap of size *N* – the size of the universe !!

- Initialize to 0s
- Scan the data set
- Set the i-th bit to 1, when the i-th item is observed
- Count the number of 1s in the bitmap

Sorting and eliminating duplicates

- Array of size *m* – the size of the data set !

- Sort and eliminate duplicates
- Count the number of elements

- Linux: `sort -u file.txt | wc –l`

- Impractical approach for current huge data sets !!

Hashing and counting

- Build an hash table with one pass over the data
- It eliminates duplicates without sorting !!
- Count the number of hash table elements

- To achieve fast insertion and avoid (too many) collisions, requires a large hash table
  - Load factor ?
  - Might not fit in main memory !!

Tasks

- Implement the exact count strategies
- Compare their performance

- Bitmap vs Sorting vs Hashing

- Which Python features can we use ?

- Use the provided test files
  - Random integer values

Approximate solutions

- Exact count methods are expensive !!
  - Memory space
  - Run time

- Relax the need for an exact solution
- And use less memory space

- How to proceed ?

A naïve algorithm

- Keep in main memory a list of all distinct items seen so far
  - They are at most n
- Use an efficient data structure
  - Hash table / search tree / …
- Fast :
  - Checking if an item exists
  - Adding a new item

A naïve algorithm

- No problem :
  - Number of distinct items not too large
    - The list fits in main memory
  - We get an exact answer !
    - The number of distinct list elements

- Problems when :
  - Number of distinct items is too large !
  - Need to process many streams simultaneously !

Solutions ?

- Use more computers
  - Each processes one or a few data streams
- Store part of the data structure in secondary memory
  - Processing is disk block oriented
  - Ensure many accesses to the current block in memory

Hashing and counting – Again !

- Build an hash table with one pass over the data
- It eliminates duplicates without sorting
- BUT NOW, do not solve collisions !!
  - Don’t insert any element that collides with an existing one

- Cardinality estimate = number of table elements

- Such an estimate can be improved !
  - Statistical correction factor

Task

- Implement the hashing and counting strategy

- The hash table stores zeroes and ones
- Count the number of ones

- Compare its performance and accuracy

- Use the provided test files
  - Random integer values

Bloom Filters

- Create a Bloom Filter – size ?
- Insert the data set elements with one pass
- BUT, query the BF before every insertion and do not insert those already in the BF
- Count the number of actually inserted elements

- Card. estimate = number of inserted elements
  - It cannot be an over-estimate – why ?
- Such an estimate can be improved !
  - Statistical correction factor

Task

- Implement a Bloom Filter for cardinality estimation
- Compare its performance and accuracy
- Use the provided test files
- Use 4 hash functions
- Filter size can be 6 times the data size

Solutions ?

- Estimate the number of distinct items
  - And use much less memory than their number !
- Accept that the count may have a little error
- But limit the probability that the error is large !

Algorithm

- Flajolet & Martin : Probabilistic counting algorithms for data base applications
  - 1985
  - www.sciencedirect.com/science/article/pii/0022000085900418

- Alon, Matias & Szegedy : The space complexity of approximating frequency moments
  - 1999
  - http://www.sciencedirect.com/science/article/pii/S0022000097915452

Algorithm

- Pick a hash function *h(a)* that maps each of the n items to, at least, $log_2\, n$ bits
  - h(a) is a 2-universal hash function
  - When applied to the same argument, it always produces the same result
  - Length of the bit-string has to be large enough
  - Number of possible hash results larger than the possible number of distinct items
  - 64 bits can be used to hash URLs

Algorithm

- For each stream token a_k : $zeros(a_k)$ is the number of trailing zeros in $h(a_k)$
  - $zeros(p) = max\{\, i : 2^i \text{ divides } p \,\}$
  - Tail length
  - The position / índex of the rightmost 1
- Record $R = max\ zeros(a_k)$
- Estimate the number of distinct stream tokens as 2^R
  - Or $2^{R + 1/2}$

Intuition – Why it seems to work

- *h(a)* hashes a with equal probability to any of *n* values
- Then, *h(a)* is a sequence of $\log_2 n$ bits

- About $1/2^r$ of all elements have a tail with *r* zeros
  - About 50% hash to ****0
  - About 25% hash to ***00
  - …

Intuition – Why it seems to work

- If the longest tail so far is $r = 2$
  - I.e., item hash ending ***00
- We have probably seen about 4 distinct items so far
- It takes to hash about 2^r stream elements, before we see one element with a zero-suffix of length r

Intuition – Why it seems to work

- It can be shown that the probability of finding a tail of r zeros
  - Goes to 1, if $k >> 2^r$
  - Goes to 0, if $k << 2^r$
  - Where k is the number of distinct elements seen so far in the stream
- Thus, 2^R will almost always be around k !!
  - Can use a statistical correction factor

Better approaches

- Use many hash functions h_i and compute many estimates R_i
- Combine those samples to get a better estimate

- How to ?

- Average ?
  - What if there is one very large estimate ?

Better approaches

- Median ?
  - But, all estimates are a power of 2 !
  - Do not get a close estimate, if the number of distinct elements lies betwen two powers of 2

- Solution
  - Partition the sample estimates into small groups
  - Take the average of each group
  - Take the median of the averages

Space requirements

- Do not store the elements seen in the stream
- Keep in main memory one integer per hash function
  - *O(log n)* for h(a)
  - *O(log log n)* for zeros(a)
- If processing just one data stream, use very many hash functions
  - Hundreds ? Thousands ? Millions ?

Run time

- The time it takes to compute hash values is the significant limitation on the number of hash functions used !!

Simple example

- σ = 3, 1, 4, 1, 5, 9, 2, 6, 5
- Determine
  - The tail length for each stream element
  - The estimate of the number of distinct elements
- For the hash functions
  - $h_1(x) = (2\,x + 1)\ mod\ 32$
  - $h_2(x) = (3\,x + 7)\ mod\ 32$
  - $h_3(x) = (4\,x)\ mod\ 32$
  - Treat the results as 5-bit binary integers

Questions

- Do you see any problems with the choice of the previous hash functions ?

- Any advice when using hash functions of the form

$$h(x) = (a\,x + b)\ mod\ 2^k$$

Tasks

- Implement the algorithm
- Test it using different data sets
- Compare the obtained results with the real count of distinct elements
  - Look for possible statistical correction factors

Other algorithms

- Several algorithms have been proposed for cardinality estimation
  - Try to find other algorithms using Google !
- The HyperLogLog algorithm has been considered as the best approach
  - It refines the original ideas of Flajolet & Martin

The HyperLogLog algorithm

- Flajolet, Fusy, Gandouet & Meunier : HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm
  - 2007
  - http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
- Near-optimal probabilistic algorithm

HyperLogLog – Practical variant

Require: Let $h : \mathcal{D} \to \{0,1\}^{32}$ hash data from domain $\mathcal{D}$.
Let $m = 2^p$ with $p \in [4..16]$.

Phase 0: Initialization.

1: Define $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$,
2: $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$.
3: Initialize m registers $M[0]$ to $M[m-1]$ to 0.

HyperLogLog – Practical variant

Phase 1: Aggregation.

for all $v \in S$ **do**

$\quad x := h(v)$

$\quad idx := \langle x_{31}, \ldots, x_{32-p} \rangle_2$ { First p bits of x }

$\quad w := \langle x_{31-p}, \ldots, x_0 \rangle_2$

$\quad M[idx] := \max\{M[idx], \varrho(w)\}$

end for

HyperLogLog – Practical variant

Phase 2: Result computation.

$E := \alpha_m m^2 \cdot \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$ { The "raw" estimate }

if $E \leq \frac{5}{2}m$ **then**

  Let V be the number of registers equal to 0.

  if $V \neq 0$ **then**

    $E^* := \textsc{LinearCounting}(m, V)$

  else

    $E^* := E$

  end if

else if $E \leq \frac{1}{30}2^{32}$ **then**

  $E^* := E$

else

  $E^* := -2^{32} \log(1 - E/2^{32})$

end if

return E^*

The Kane et al. algorithm

- Kane, Nelson & Woodruff : An optimal algorithm for the distinct elements problem
  - 2010
  - http://dl.acm.org/citation.cfm?doid=1807085.1807094
- Seems to close a line of theoretical research on this problem
- Are there any new, recent developments / algorithms ?

HyperLogLog in Practice

- Heule, Nunkesser & Hall : HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm
  - Google, Inc. !!
  - 2013
  - http://dl.acm.org/citation.cfm?id=2452456

- Improvements to HyperLogLog !
- Empirical evaluation !

HyperLogLog in Practice

- At Google, various data analysis systems estimate the cardinality of very large data sets every day

- E.g., to determine the number of distinct search queries over a time period

Requirements

- Accuracy
  - Accurate estimates, for a fixed amount of memory
  - Near exact results, for small cardinalities
- Memory efficiency
  - Efficient memory use
  - Memory usage adapted to cardinality

Requirements

- Estimate large cardinalities
  - Cardinalities well beyond 1 billion occur daily
  - Estimate them with reasonable accuracy
- Practicality
  - The algorithm should be implementable and maintainable.

Practicality

- The algorithm by Kane et al. meets the memory space lower bound

- BUT, it is complex and an actual implementation and its maintenance seems out of reach in a practical system

HyperLogLog in practice

- HyperLogLog++ fulfills the requirements
- 64 bit hash codes allow the algorithm to estimate cardinalities well beyond 1 billion !
- Significantly better accuracy
- Adaptive use of memory

2017 – Experimental survey of 12 algs.

Cardinality Estimation: An Experimental Survey

Hazar Harmouch

Felix Naumann

umn, also known as the zeroth-frequency moment. Cardinality estimation itself has been an active research topic in the past decades due to its many applications. The aim of this paper is to review the literature of cardinality estimation and to present a detailed experimental study of twelve algorithms, scaling far beyond the original experiments.

First, we outline and classify approaches to solve the problem of cardinality estimation – we describe their main idea, error-guarantees, advantages, and disadvantages. Our experimental survey then compares the performance all twelve cardinality estimation algorithms. We evaluate the algorithms' accuracy, runtime, and memory consumption using synthetic and real-world datasets. Our results show that

2017 - Genomics

Bioinformatics, 33(9), 2017, 1324–1330
doi: 10.1093/bioinformatics/btw832
Advance Access Publication Date: 5 January 2017
Original Paper

Sequence analysis

ntCard: a streaming algorithm for cardinality estimation in genomics data

Hamid Mohamadi[1,2,*], Hamza Khan[1,2] and Inanc Birol[1,2,*]

[1]Canada's Michael Smith Genome Sciences Centre, British Columbia Cancer Agency, Vancouver, BC, V5Z 4S6,

2017 – Elephant Flows

3738 IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 25, NO. 6, DECEMBER 2017

Cardinality Estimation for Elephant Flows: A Compact Solution Based on Virtual Register Sharing

Qingjun Xiao, *Member, IEEE, ACM*, Shigang Chen, *Fellow, IEEE*, You Zhou, *Member, IEEE*, Min Chen, *Member, IEEE*, Junzhou Luo, *Member, IEEE*, Tengli Li, and Yibei Ling, *Senior Member, IEEE*

Reference

- J. Leskovec, A. Rajaraman & J. Ullman, Mining of Massive Datasets, 2nd Ed., Cambridge University Press, 2014
  - Chapter 4 : Mining data streams
  - http://www.mmds.org/