Data Stream Algorithms III

Joaquim Madeira

Version 0.4 – December 2024

Overview

- Counting the Number of Distinct Items

MOTIVATION

Set of distinct items

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

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

- f_j is the number of occurrences of item j

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

Number of distinct items

- ϵ (epsilon) refers to the **error tolerance** of the approximation. It indicates how much the estimated count can differ from the true count, typically as a fraction of the true count. For example, an $\epsilon = 0.01$ means the estimate can be off by 1% from the actual count.
- δ (delta) refers to the **confidence level** of the approximation. It is the probability that the approximation is within the error tolerance. For instance, if $\delta = 0.05$, there is a 95% chance that the approximation error is within ϵ.

- 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., $\delta = 0$ d - delta
  - An exact algorithm – i.e., $\epsilon = 0$ e - error parameter
- 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

DIRECT APPROACHES

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 !!

O(m logm + m) = O(m logm)

m being the number of distinct elements
which in the worst case is the previous m

Hashing and counting

- Build a 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

APPROXIMATIVE METHODS

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 a 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 (em teoria vai dar um cardinalidade menor)
- 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 ? (Everyone with a brain understands 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

A **2-universal hash function** is a type of hash function with a specific property that ensures a low probability of collisions. Specifically, for a hash function h to be 2-universal, it must satisfy the following condition:

For any two distinct inputs a and b,

$$\Pr[h(a) = h(b)] \leq \frac{1}{m}$$

where m is the size of the range of the hash function.

- 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 / index 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
- *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 ? Nonsense. Just some

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

Obtain the number of trailing zeros for each of the hash functions

  - $h_1(x) = (2\,x + 1)\ mod\ 32$
  - $h_2(x) = (3\,x + 7)\ mod\ 32$
  - $h_3(x) = (4\,x)\ mod\ 32$
  - Represent 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

THE HYPERLOGLOG ALGORITHM

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.

- **Hash function h:**
  - A hash function is used to map data from a domain $\mathcal{D}$ to a 32-bit binary representation ($\{0,1\}^{32}$).
  - This ensures uniform distribution of data.
- **Parameter $m = 2^p$:**
  - m is the number of registers (small memory units) used by the algorithm. It must be a power of 2.
  - The parameter p determines the precision of the algorithm and lies in the range $[4, 16]$.

2. Phase 0: Initialization

Step 1: Define α_m (Scaling Factor)

- The value of α_m depends on m, and it is a constant used in the final estimation formula.
- For small values of m:
  - $\alpha_{16} = 0.673$,
  - $\alpha_{32} = 0.697$,
  - $\alpha_{64} = 0.709$.
- For larger values ($m \geq 128$):
  - $\alpha_m = \frac{0.7213}{1+1.079/m}$.
  - This formula ensures accuracy for larger m.

Step 2: Initialize Registers

- Create m registers ($M[0], M[1], \ldots, M[m-1]$).
- All registers are initialized to 0. These registers will eventually store information about the positions of leading zeros in the hashed values.

HyperLogLog – Practical variant

rho (ρ)

primeiros p bits -> indexar
ultimos p bits -> trailing zero bullshit

Phase 1: Aggregation.

4: **for all** $v \in S$ **do** (Iterate through all elements of the dataset S)

5: $x := h(v)$ get binary representation with 32 bits

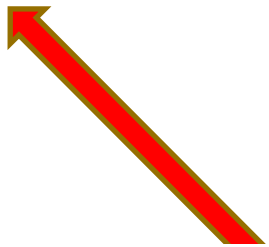
6: $idx := \langle x_{31}, \ldots, x_{32-p} \rangle_2$ { First p bits of x }

7: $w := \langle x_{31-p}, \ldots, x_0 \rangle_2$ remaining 32 - p bits

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

9: **end for**

update with the max value of current value and the new value



The add operation consists of computing the hash of the input data v with a hash function h, getting the first b bits (where b is $\log_2(m)$), and adding 1 to them to obtain the address of the register to modify. With the remaining bits compute $\rho(w)$ which returns the position of the leftmost 1, where leftmost position is 1 (in other words: number of leading zeros plus 1). The new value of the register will be the maximum between the current value of the register and $\rho(w)$.

HyperLogLog – Practical variant

Phase 2: Result computation.

10: $E := \alpha_m m^2 \cdot \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$ { The "raw" estimate }
11: **if** $E \leq \frac{5}{2}m$ **then**
12: Let V be the number of registers equal to 0.
13: **if** $V \neq 0$ **then**
14: $E^* := \text{LinearCounting}(m, V)$
15: **else**
16: $E^* := E$
17: **end if**
18: **else if** $E \leq \frac{1}{30}2^{32}$ **then**
19: $E^* := E$
20: **else**
21: $E^* := -2^{32} \log(1 - E/2^{32})$
22: **end if**
23: **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

RECENT PUBLICATIONS

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

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*

2019 – Deep Cardinality Estimation

Deep Unsupervised Cardinality Estimation

Zongheng Yang[1], Eric Liang[1], Amog Kamsetty[1], Chenggang Wu[1], Yan Duan[3], Xi Chen[1,3], Pieter Abbeel[1,3], Joseph M. Hellerstein[1], Sanjay Krishnan[2], Ion Stoica[1]

[1]UC Berkeley [2]University of Chicago [3]covariant.ai

ABSTRACT

Cardinality estimation has long been grounded in statistical tools for density estimation. To capture the rich multivariate distributions of relational tables, we propose the use of a new type of high-capacity statistical model: deep autoregressive models. However, direct application of these models leads to a limited estimator that is prohibitively expensive to evaluate for range or wildcard predicates. To produce a truly usable estimator, we develop a Monte Carlo integration scheme on top of autoregressive models that can efficiently handle range queries with dozens of dimensions or more.

2020 – Improved Memory Efficiency

IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 28, NO. 2, APRIL 2020

Estimating Cardinality for Arbitrarily Large Data Stream With Improved Memory Efficiency

Qingjun Xiao, *Member, IEEE, ACM*, Shigang Chen, *Fellow, IEEE*, You Zhou, and Junzhou Luo, *Member, IEEE, ACM*

***Abstract*— Cardinality estimation is the task of determining the number of distinct elements (or the cardinality) in a data stream, under a stringent constraint that the input data stream can be scanned by just one single pass. This is a fundamental problem with many practical applications, such as traffic monitoring of high-speed networks and query optimization of Internet-scale database. To solve the problem, we propose an algorithm named HLL-TailCut, which implements the estimation standard error $1.0/\sqrt{m}$ using the memory units of four or three bits each, whose cost is much smaller than the five-bit memory units used by HyperLogLog, the best previously known cardinality estimator. This makes it possible to reduce the memory cost of HyperLogLog by 20%∼45%. For example, when the target estimation error is 1.1%, state-of-the-art HyperLogLog needs 5.6 kilobytes memory. By contrast, our new algorithm only needs 3 kilobytes memory consumption for attaining the same accuracy.**

2022 – Learned Cardinality Estimation

Learned Cardinality Estimation: An In-depth Study

Kyoungmin Kim[†] Jisung Jung[†] In Seo[†] Wook-Shin Han[†*] Kangwoo Choi[‡] Jaehyok Chong[‡]
Pohang University of Science and Technology (POSTECH), Korea[†] SAP Labs, Korea[‡]

ABSTRACT

Learned cardinality estimation (CE) has recently gained significant attention for replacing long-studied traditional CE with machine learning, especially for deep learning. However, these estimators were developed independently and have not been fairly or comprehensively compared in common settings. Most studies use a subset of IMDB data which is too simple to measure their limits and determine whether they are ready for real, complex data. Furthermore, they are regarded as black boxes, without a deep understanding of why large errors occur.

In this paper, we first provide a taxonomy and a unified workflow of learned estimators for a better understanding of estimators. We next comprehensively compare recent learned CE methods that support joins, from a subset of tables to full IMDB and TPC-DS datasets. Under the experimental results, we then demystify the black-box models and analyze critical components that cause large errors. We also measure their impact on query optimization. Finally,

2022 – Benchmark Evaluation

Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation

Yuxing Han[1,#], Ziniu Wu[1,#], Peizhi Wu[2], Rong Zhu[1,*], Jingyi Yang[2], Liang Wei Tan[2], Kai Zeng[1], Gao Cong[2], Yanzhao Qin[1,3], Andreas Pfadler[1], Zhengping Qian[1], Jingren Zhou[1], Jiangneng Li[1], Bin Cui[3]
[1]Alibaba Group, [2]Nanyang Technological University, [3]Peking University

ABSTRACT

Cardinality estimation (CardEst) plays a significant role in generating high-quality query plans for a query optimizer in DBMS. In the last decade, an increasing number of advanced CardEst methods (especially ML-based) have been proposed with outstanding estimation accuracy and inference latency. However, there exists no study that systematically evaluates the quality of these methods and answer the fundamental problem: *to what extent can these methods improve the performance of query optimizer in real-world settings, which is the ultimate goal of a CardEst method.*

In this paper, we comprehensively and systematically compare the effectiveness of CardEst methods in a real DBMS. We establish a new benchmark for CardEst, which contains a new complex real-world dataset STATS and a diverse query workload STATS-CEB

2022 – More Accuracy

More accurate cardinality estimation in data streams

Jie Lu,[1,✉] Hongchang Chen,[2] Zheng Zhang,[1,3] and Jichao Xie[1]

[1] PLA Information Engineering University, Zhengzhou, China

Many sketches based on estimator sharing have been proposed to estimate cardinality with huge flows in data streams. However, existing sketches suffer from large estimation errors due to allocating the same memory size for each estimator without considering the skewed cardinality distribution. Here, a filtering method called SuperFilter is proposed to enhance existing sketches. SuperFilter intelligently identifies high-cardinality flows from the data stream, and records them with the large estimator, while other low-cardinality flows are recorded using a traditional sketch with small estimators. The experimental results show that SuperFilter can reduce the average absolute error of cardinality estimation by over 81% compared with existing approaches.

2023 – Better Cardinality Estimators

Better Cardinality Estimators for HyperLogLog, PCSA, and Beyond

Dingyu Wang
wangdy@umich.edu
University of Michigan

Seth Pettie
pettie@umich.edu
University of Michigan

ABSTRACT

Cardinality Estimation (aka Distinct Elements) is a classic problem in sketching with many applications in databases, networking, and security. Although sketching algorithms are fairly simple, analyzing the cardinality *estimators* is notoriously difficult, and even today the analyses of state-of-the-art sketches like HyperLogLog and PCSA are not very accessible.

In this paper we introduce a new class of estimators called τ-*Generalized-Remaining-Area* estimators, as well as a dramatically simpler approach to analyzing estimators. The estimators of Durand and Flajolet [11], Flajolet et al. [15], and Lang [24] can be seen as τ-GRA estimators for integer values of τ. By using *fractional* values of τ we derive improved estimators for HyperLogLog and PCSA whose variance comes very close to the Cramér-Rao lower bounds

May 2024 – UltraLogLog

UltraLogLog: A Practical and More Space-Efficient Alternative to HyperLogLog for Approximate Distinct Counting

Otmar Ertl
Dynatrace Research
Linz, Austria

ABSTRACT

Since its invention HyperLogLog has become the standard algorithm for approximate distinct counting. Due to its space efficiency and suitability for distributed systems, it is widely used and also implemented in numerous databases. This work presents UltraLogLog, which shares the same practical properties as HyperLogLog. It is commutative, idempotent, mergeable, and has a fast guaranteed constant-time insert operation. At the same time, it requires 28% less space to encode the same amount of distinct count information, which can be extracted using the maximum likelihood method. Alternatively, a simpler and faster estimator is proposed, which still achieves a space reduction of 24%, but at an estimation speed comparable to that of HyperLogLog. In a non-distributed setting

REFERENCE

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/