Algorithms and Applications for the Estimation of Stream Statistics in Networks

Aviv Yehezkel

Ph.D. Research Proposal

Supervisor: Prof. Reuven Cohen

Overview

- Motivation
- Introduction
  - Cardinality Estimation Problem
  - Weighted Problem
- A Unified Scheme for Generalizing Cardinality Estimators to Sum Aggregation
- Efficient Detection of Application Layer DDoS Attacks by a Stateless Device
- Future Research
  - Combining Cardinality Estimation and Sampling
  - Estimation of Set Intersection
- New Results

Motivation for Sketching

We often need to monitor network links for quantities such as:

- Elephant flows (traffic engineering, billing)
- Number of distinct flows, average flow size (queue management)
- Flow size distribution (anomaly detection)
- Per-flow traffic volume (anomaly detection)
- Entropy of the traffic (anomaly detection)
- …

Motivation for Sketching

Network monitoring at high speed is challenging

- Packets arrive every 25ns on a 40 Gbps (OC-768) link
- Has to use SRAM for per-packet processing
- Per-flow state too large to fit into SRAM
- Classical solution of sampling is not accurate due to the low sampling rate dictated by the resource constraints

Sketching

- Main idea:
  - Use a small fixed size storage to store only the "most important" information about the stream elements, a **summary** of the data **= the sketch**
  - Process the stream of data (packets) in one pass
  - No need to store per-flow states for each flow
  - Employ probabilistic algorithm on the sketch to get an accurate estimation of the wanted quantity

- **What sketch to store?**
- **What algorithm to use to get accurate estimations? = unbiased & small variance**

Introduction

- In this research we present sketch-based algorithms for the estimation of different stream statistics
- We analyze their efficiency and statistical performance (bias and variance), and use these algorithms to develop new applications for various networking tasks
- We begin with the **“cardinality estimation problem”** and study its generalized **weighted estimation problem**
- We use our weighted estimator for developing new schemes that allow a stateless Network layer appliance:
  - To determine in real-time the Application layer load imposed on its end server
  - To detect Application layer attacks

Cardinality Estimation Problem

Motivation

Given a very long stream of elements with repetitions,

How many are distinct?

Cardinality of a Stream

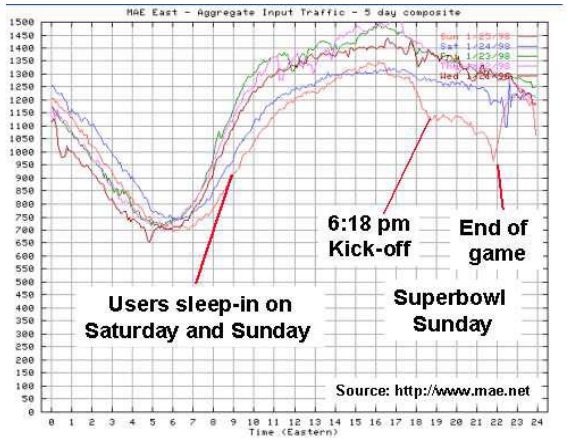
- Let M be a stream of elements with repetitions
  - N is the number of elements called the size
  - n the number of distinct elements called cardinality

$$N = 8$$
$$n = 4$$

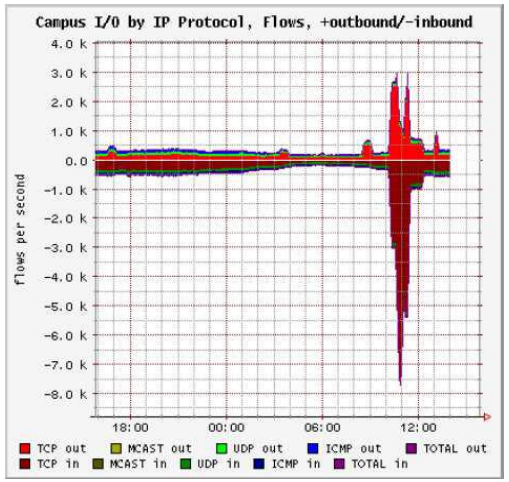
Element	Multi
C	1
D	3
B	3
Z	1

- **The problem:**
  compute the cardinality n in one pass and with small fixed memory

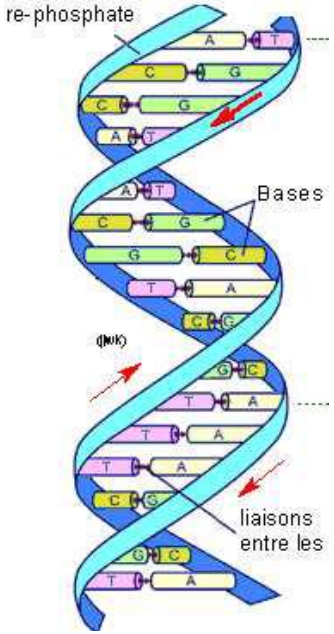
Many Applications



Traffic analysis



Attacks detection



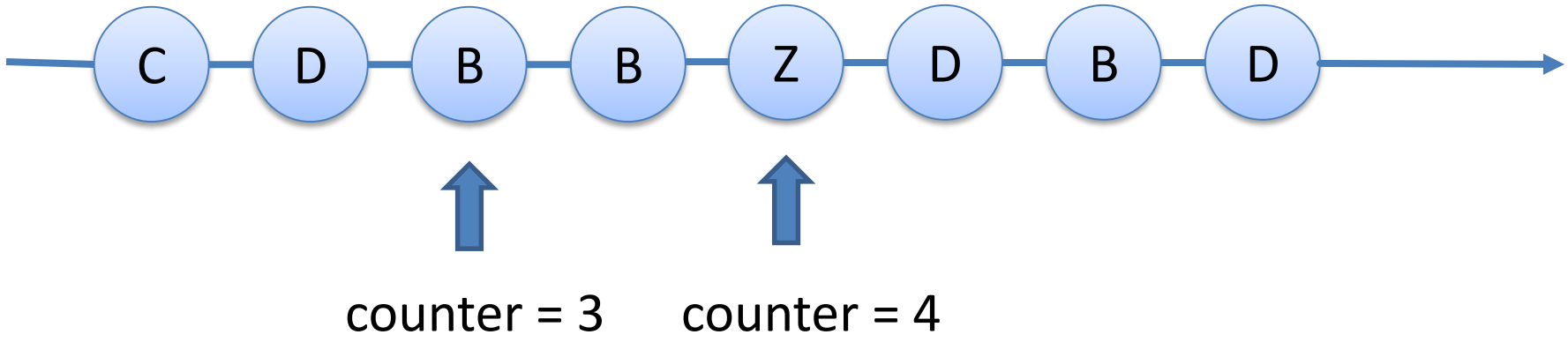
Genetics

Linguistic

and more…

Exact Solution

- Maintain distinct elements already seen



- One pass, but memory in order of n
- Lower bound: $\Omega(n)$ memory needed

Data Sampling

- Only a small sample of the data is collected (marked in red), and then analyzed

- Sensitive to the arrival order and to the repetition pattern

- Low accuracy

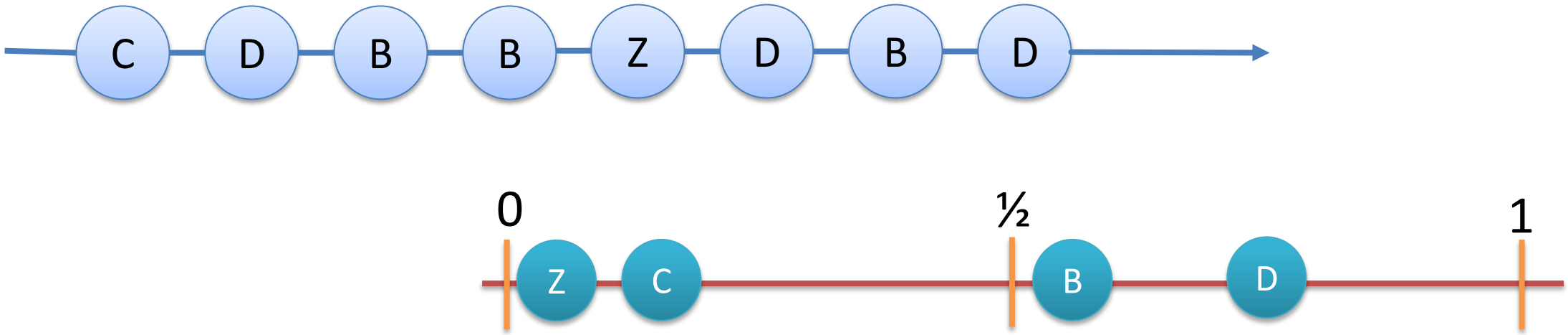
Sketch-based Solution

- Main idea:
  - relax the constraint of exact value of the cardinality
  - An estimate with good precision is sufficient for the applications

- Several algorithms:
  - Probabilistic counting
  - HyperLogLog
  - Linear Counting
  - Min Count
  - ….

Sketch-based Solution

- Elements of M are hashed to random variables in (0,1)



- Idea: use the maximum\minumum to estimate the cardinality
  - One pass
  - Constant memory

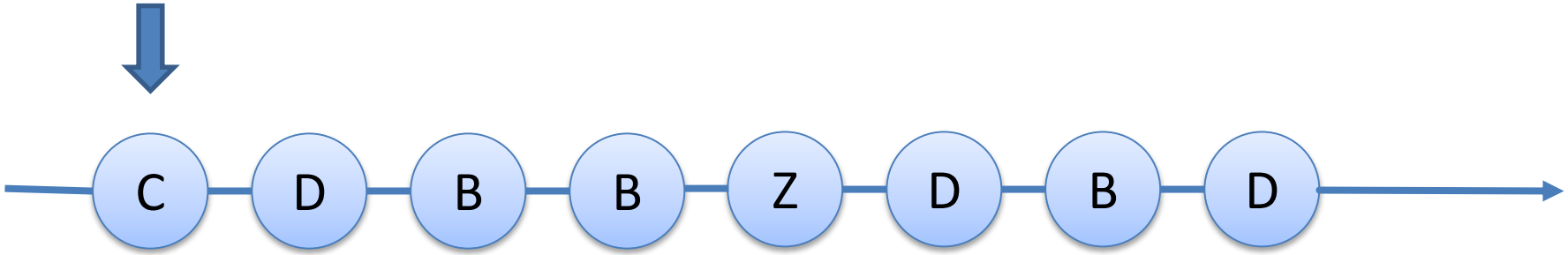
Sketch-based Solution

- Elements of M are hashed to random variables in (0,1)

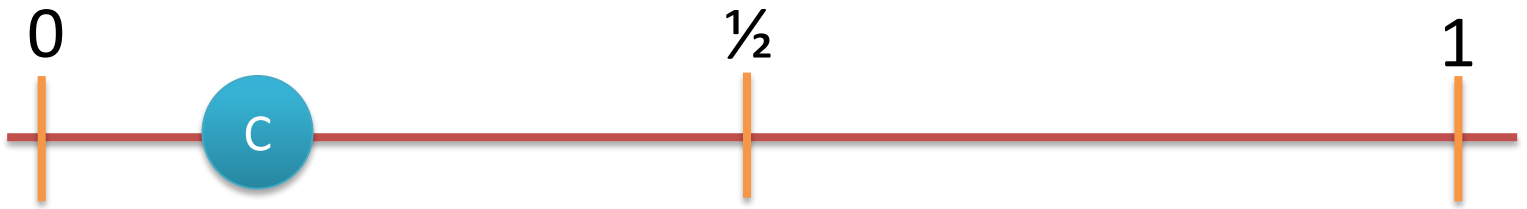
- Intuition:
  - If there are 10 distinct elements,
  - Expect the hash values to be spaced about $\frac{1}{10}$th apart from each other

- $\mathbb{E}(max) = \frac{n}{n+1}$

Sketch-based Solution

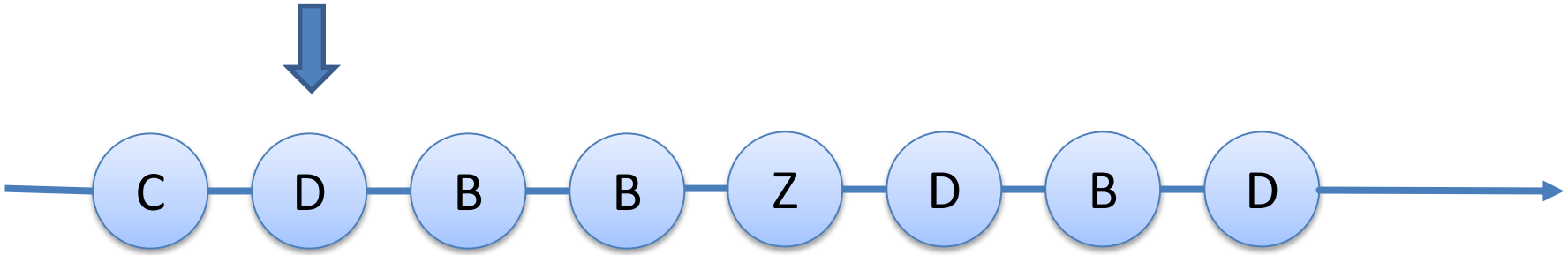


$h(C) = 0.347$

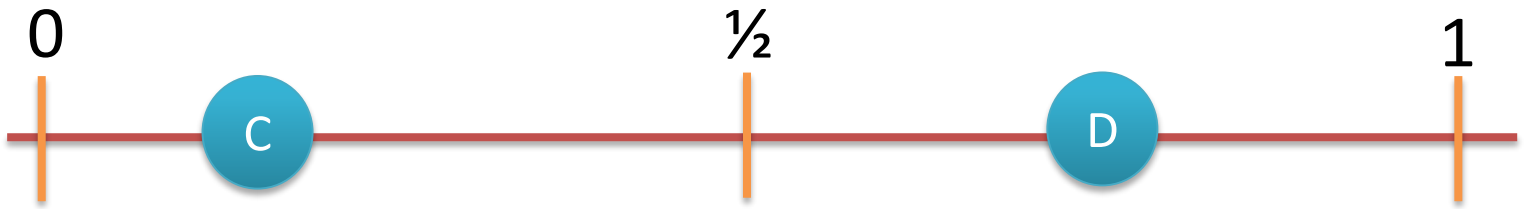


$$h^+ = 0.347$$

Sketch-based Solution

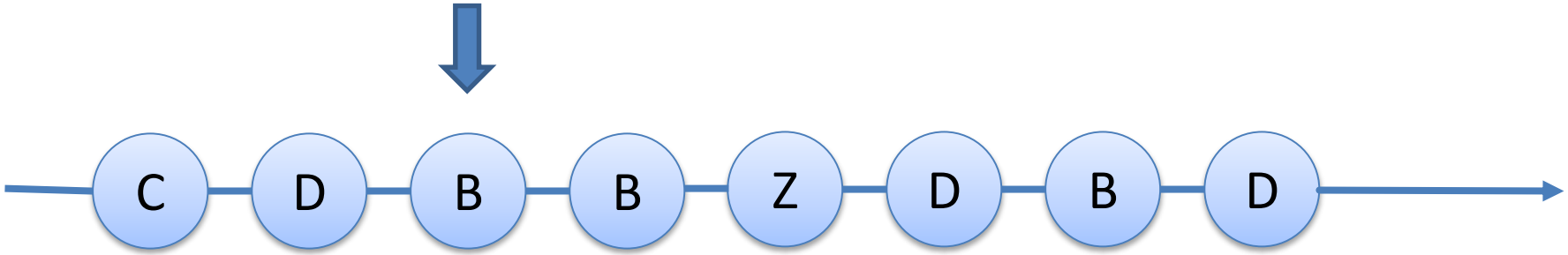


$h(D) = 0.773$

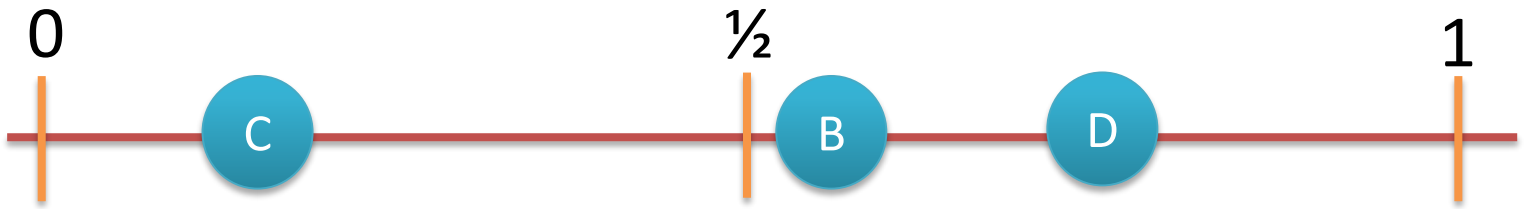


$$h^+ = 0.773$$

Sketch-based Solution

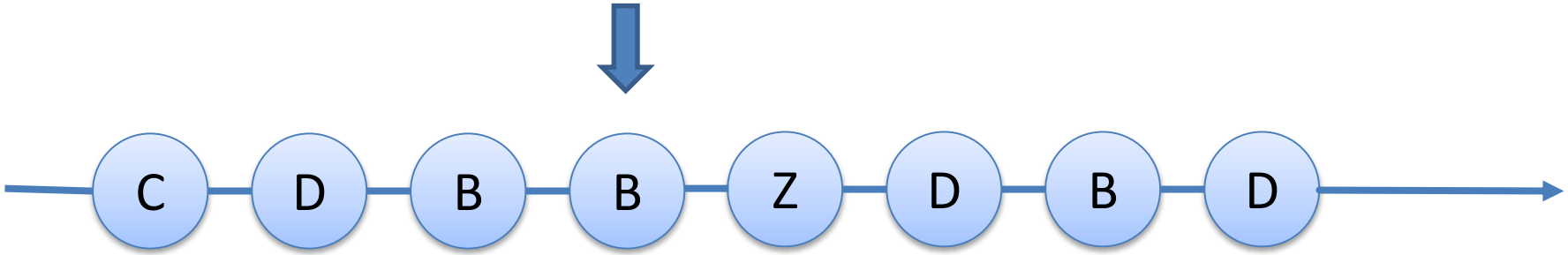


$h(B) = 0.512$

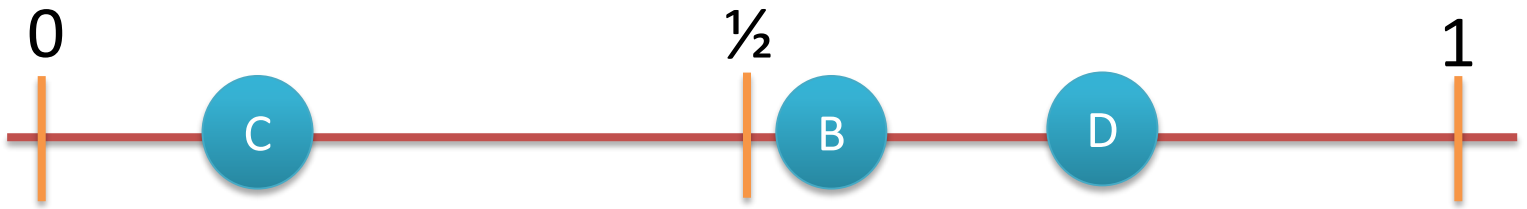


$$h^+ = 0.773$$

Sketch-based Solution

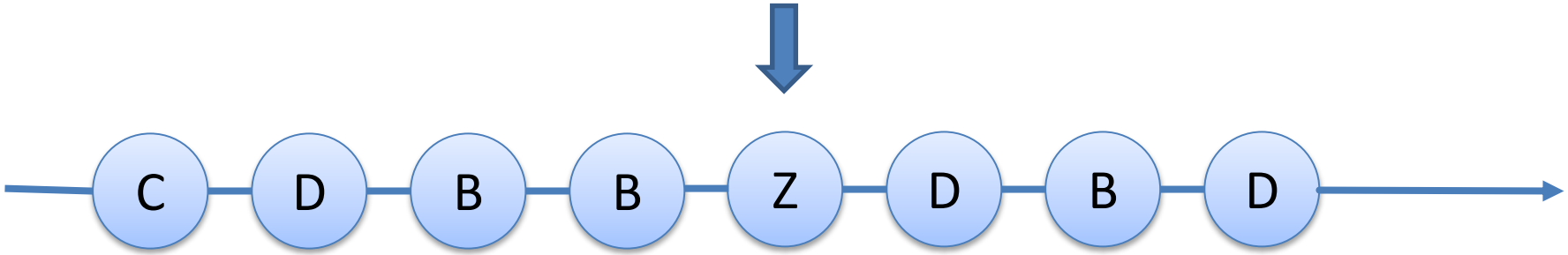


$h(B) = 0.512$

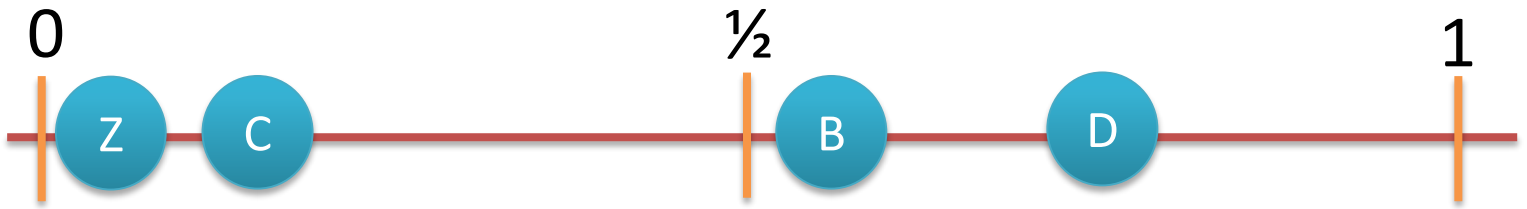


$h^+ = 0.773$

Sketch-based Solution

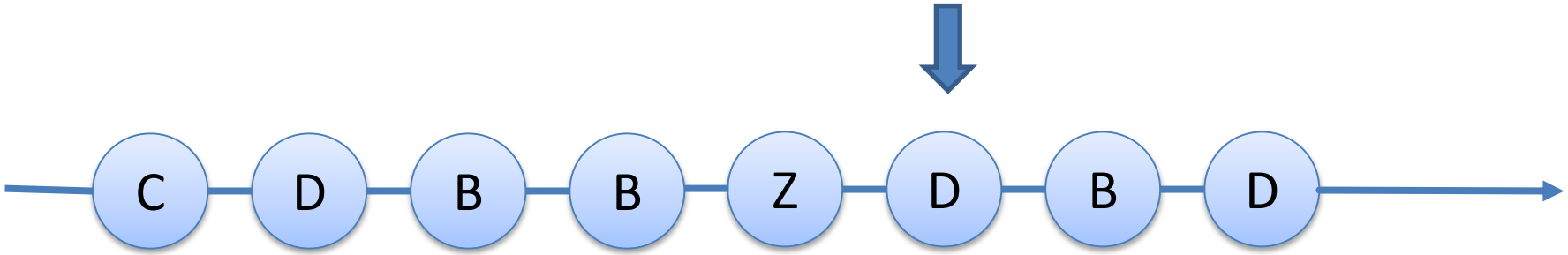


$h(Z) = 0.139$

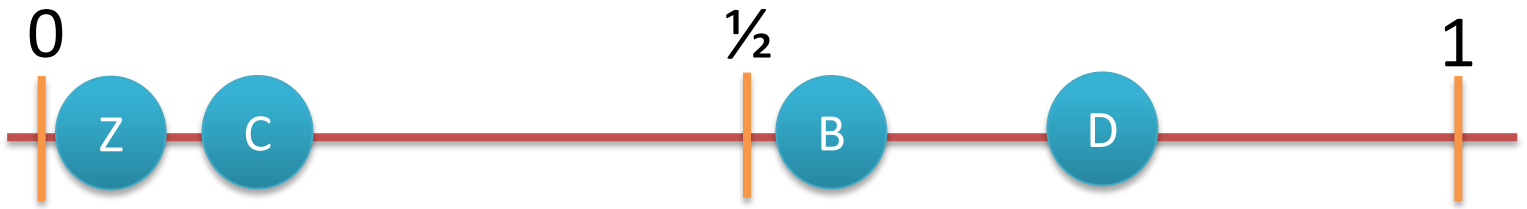


$$h^+ = 0.773$$

Sketch-based Solution

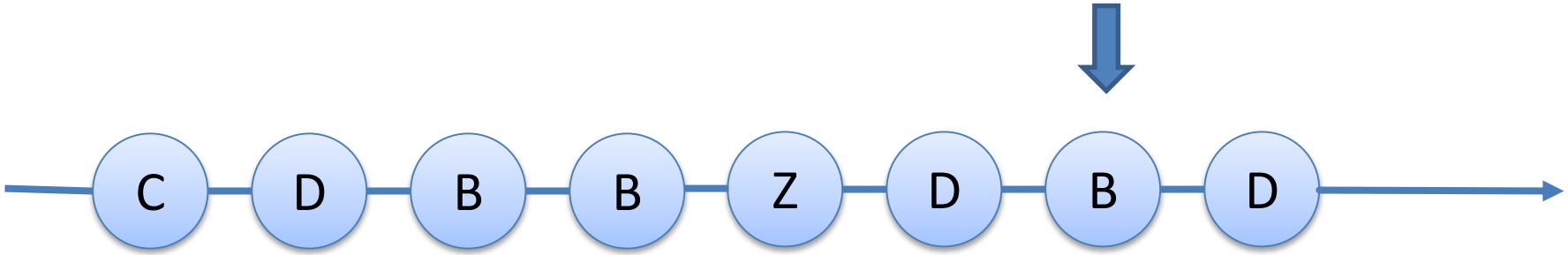


$h(D) = 0.773$

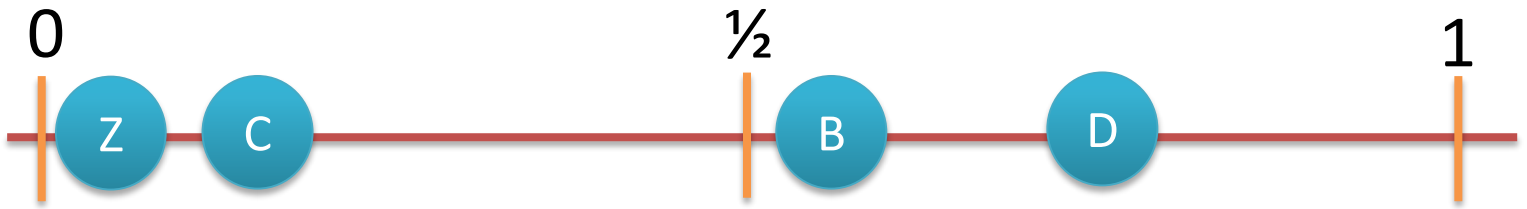


$$h^+ = 0.773$$

Sketch-based Solution

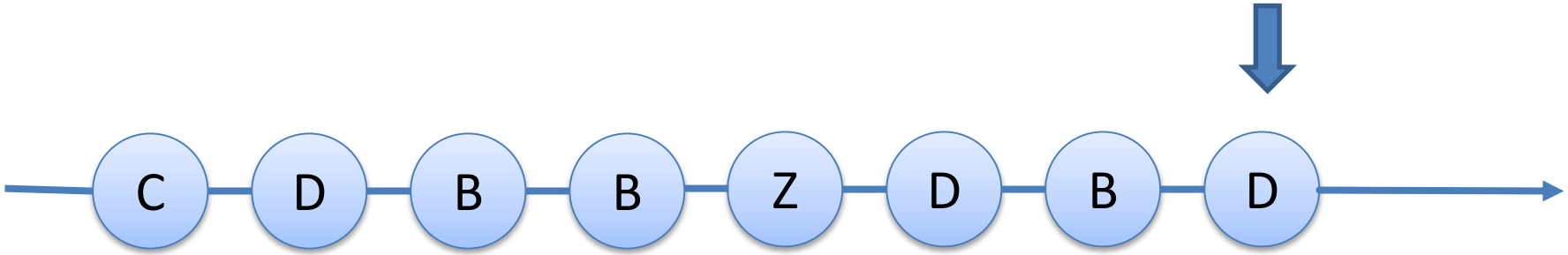


$h(B) = 0.512$

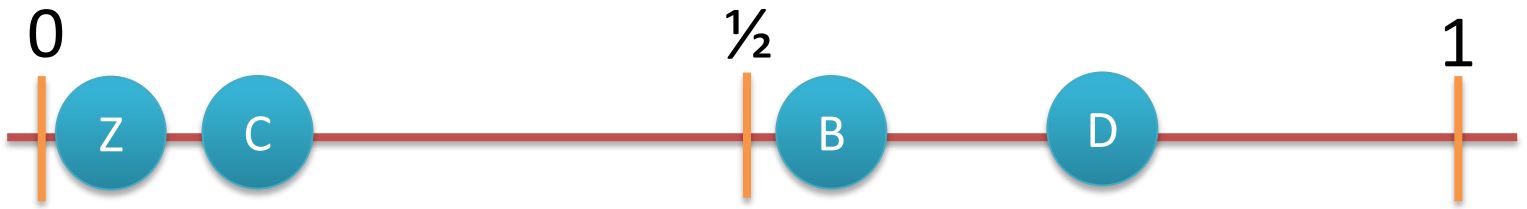


$$h^+ = 0.773$$

Sketch-based Solution

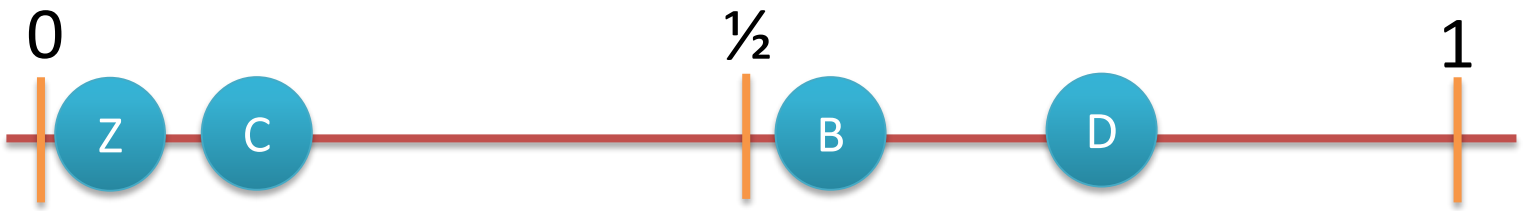


$h(D) = 0.773$



$$h^+ = 0.773$$

Sketch-based Solution



- $\mathbb{E}(max) = \frac{n}{n+1} = 0.773$
- Estimated cardinality = 3.405
- Actual cardinality = 4

Chassaing Algorithm

- Simulate m different hash functions
  - m maxima $h_1^+, h_2^+, \ldots, h_m^+$

- Estimate = $\dfrac{m-1}{\sum(1-h_k^+)}$

Chassaing Algorithm

- $h_k^+ \sim \frac{n}{n+1}$
- $\sum(1 - h_k^+) \sim \sum \frac{1}{n+1} = \frac{m}{n+1}$
- Therefore,
  - Estimate = $\frac{m-1}{\sum(1-h_k^+)} \sim n$

Chassaing Algorithm

- Relative error $\approx 1/\sqrt{m}$ for a memory of m words
- Minimal variance unbiased estimator (MVUE)

Formal Definition

Instance:

A stream of elements $x_1, x_2, \ldots, x_s$ with repetitions, and an integer m

Let n be the number of different elements, denoted by $e_1, e_2, \ldots, e_n$

Objective:

Find an estimate $\hat{n}$ of n, using only m storage units, where $m \ll n$

Min/Max Sketches

- Use m different hash functions
- Hash every element x_i to m uniformly distributed hashed values $h_k(x_i)$
- Remember only the minimum/maximum value for each hash function h_k
- Use these m values to estimate n

Generic Max Sketch Algorithm

Algorithm 1

1. Use m different hash functions
2. For every h_k and every input element x_i, compute $h_k(x_i)$
3. Let $h_k^+ = \max\{h_k(x_i)\}$ be the maximum observed value for h_k
4. Invoke $ProcEstimate(h_1^+, h_2^+, \ldots, h_m^+)$ to estimate n

Other Estimation Techniques

- Bit pattern sketches
  - The elements are hashed into a bit vector and the sketch holds the logical OR of all hashed values

- Bottom-m sketches
  - maintain the *m* minimal values

Weighted Cardinality Estimation Problem

Weighted Sum of a Stream

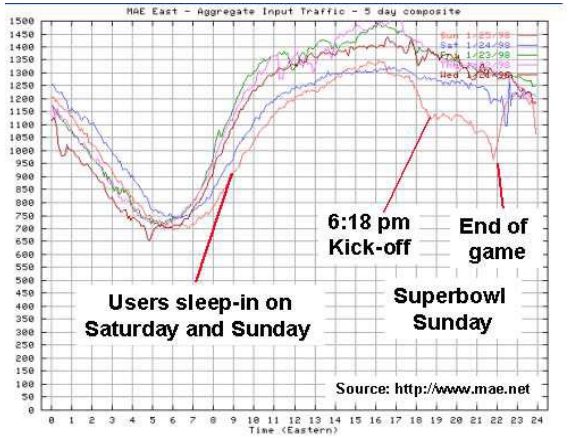
- Each element is associated with a weight
- The goal is to estimate the weighted sum w of the distinct elements

Element	Weight
C	0.5
D	0.25
B	1
Z	1.25

$$w = \sum w\_i = \\ = 0.5 + 0.25 + 1 + 1.25 = 3$$

Application Example

- Stream of IP packets received by a server $x_1, x_2, \ldots, x_s$
- Each packet belongs to a flow (connection) $e_1, e_2, \ldots, e_n$
- Each flow e_j imposes a load w_j on the server
- The weighted sum $w = \sum w_j$ represents the total load imposed on the server



Formal Definition

Instance:

A stream of weighted elements $x_1, x_2, \ldots, x_s$ with repetitions, and an integer m

Let n be the number of different elements, and let w_j be the weight of e_j

Objective:

Find an estimate $\hat{w}$ of $w = \sum w_j$, using only m storage units, where $m \ll n$

A Unified Scheme for Generalizing Cardinality Estimators to Sum Aggregation

Our Contribution

- A unified scheme for generalizing any min/max estimator for the unweighted cardinality estimation problem to an estimator for the weighted cardinality estimation problem.

Previous Works

- Cohen, 1997
  - Exponential distribution $h(e_j) \sim Exp(w_j)$
  - Minimal hash value is stored, $h^+ = \min(h(e_j))$
    - $h^+ \sim Exp(\sum w_j) = Exp(w)$
  - **Can be obtained as a direct result of our scheme**

- Jeffrey Considine, Feifei Li, George Kollios, and John W. Byers, 2004
  - Bit pattern sketch
    - Integer weights only
    - Storage is not fixed
  - **Our scheme does not assume integer weights and uses fixed memory**

Observation

- All min/max sketches can be viewed as a two step computation:
  1. Hash each element uniformly into (0, 1)
  2. Store only the minimum/maximum observed value

The Unified Scheme

- In the unified scheme we only change step (1) and hash each element into a Beta distribution.

- The parameters of the Beta distribution are derived from the weight of the element.

Beta Distribution

- Defined over the interval (0, 1)

- Has the following probability and cumulative density functions (PDF and CDF):

$$P[X = x \in (0,1)] = \frac{\Gamma(\alpha+\beta)}{\Gamma(\beta)\,\Gamma(\alpha)} x^{\alpha-1}(1-x)^{\beta-1}$$

$$P[X \leq x] = \int_0^x \frac{\Gamma(\alpha+\beta)}{\Gamma(\beta)\,\Gamma(\alpha)} x^{\alpha-1}(1-x)^{\beta-1} dx,$$

Beta Distribution

- Known (and useful) identities:

$$\text{Beta}\,(1,1) \sim \text{U}(0,1)$$

and

$$\text{Beta}\,(\alpha,\beta) \sim 1 - \text{Beta}\,(\beta,\alpha)\,.$$

Beta Distribution

Lemma:

Let $z_1, z_2, \ldots, z_n$ be independent RVs, where $z_i \sim Beta(w_i, 1)$

Then,

$$\max\{z_i\} \sim Beta(\sum w_i, 1)$$

Corollary

- For every hash function,

$$h_k^+ = \max\{h_k(x_i)\} \sim \max\{U(0,1)\}$$
$$\sim \max\{Beta(1,1)\} \sim \mathrm{Beta}(\mathrm{n}, 1)$$

- Thus, estimating the value of n by Algorithm 1, is equivalent to estimating the value of α in the $\mathrm{Beta}(\alpha, 1)$ distribution of h_k^+

The Unified Scheme

For estimating the weighted sum:

- Instead of associating each element with a uniform hashed value
  - $h_k(x_i) \sim U(0,1)$

- We associate it with a RV taken from a Beta distribution
  - $h_k(x_i) \sim Beta(w_j, 1)$
  - w_j is the element's weight

Generic Max Sketch Algorithm - Weighted

Algorithm 2

- Use m different hash functions
- For every h_k and every input element x_i:
    1. compute $h_k(x_i)$
    2. transform to $\hat{h_k}(x_i) \sim Beta(w_j, 1)$
- Let $h_k^+ = \max\{\hat{h_k}(x_i)\}$ be the maximum observed value for h_k
- Invoke $ProcEstimate(h_1^+, h_2^+, \ldots, h_m^+)$ to estimate the value of w

The Unified Scheme

- Practically, if

$$h_k(x_i) \sim U(0,1)$$

- Then,

$$h(x_i)_k^{1/w_j} \sim Beta(w_j, 1)$$

Distributions Summary

Unweighted	$h_k^+ \sim Beta(n, 1)$
Weighted	$h_k^+ \sim Beta(\mathrm{w} = \sum w_j, 1)$

The Unified Scheme

- The same algorithm that estimates n in the unweighted case can estimate w in the weighted case

- $ProcEstimate()$ is exactly the same procedure used to estimate the unweighted cardinality in Algorithm 1

The Unified Scheme Lemma

Estimating w by Algorithm 2 is equivalent to estimating n by Algorithm 1.

Thus, Algorithm 2 estimates w with the same variance and bias as that of the underlying procedure used by Algorithm 1.

Stochastic Averaging

- Presented by Flajolet in 1985
- Use 2 hash functions instead of m
- Overcome the computational cost at the price of negligible statistical efficiency in the estimator's variance

Stochastic Averaging

- Use 2 hash functions:
  1. $H_1(x_i) \sim \{1,2,\dots,m\}$
  2. $H_2(x_i) \sim U(0,1)$

- Remember the maximum observed value of each bucket

- The generalization to weighted estimator is similar

Generic Max Sketch Algorithm (Stochastic Averaging)

Algorithm 3

1. Use 2 different hash functions
    1. $H_1(x_i) \sim \{1,2,\ldots,m\}$
    2. $H_2(x_i) \sim U(0,1)$
2. For every input element x_i compute $H_1(x_i)$ and $H_2(x_i)$
3. Let $h_k^+ = \max\{H_2(x_i) \mid H_1(x_i) = k\}$ be the maximum observed value in the k'th bucket
4. Invoke $ProcEstimateSA(h_1^+, h_2^+, \ldots, h_m^+)$ to estimate n

Corollary (Stochastic Averaging)

- $\mathrm{b_k} = |\{H_1(x_i\} = k\}|$ = size of k'th bucket
  - $b_k = \frac{n}{m} \pm O\left(\sqrt{\frac{n}{m}}\right)$

- For every hash function,

$$h_k^+ = \max\{H_2(x_i) \mid H_1(x_i) = k\} \sim \mathrm{Beta}(\mathrm{b_k}, 1) \sim \mathrm{Beta}\left(\frac{n}{m}, 1\right)$$

- Thus, estimating the value of $\frac{\mathrm{n}}{\mathrm{m}}$ by Algorithm 3, is equivalent to estimating the value of α in the $\mathrm{Beta}(\alpha, 1)$ distribution of $\mathrm{h_k^+}$

The Unified Scheme (Stochastic Averaging)

For estimating the weighted sum:

- Instead of associating each element with a uniform hashed value
  - $H_2(x_i) \sim U(0,1)$

- We associate it with a RV taken from a Beta distribution
  - $H_2(x_i) \sim Beta(w_j, 1)$
  - w_j is the element's weight

- $b_k = \sum_{\{H_1(x_i\}=k\}} w_j$ is the sum of the elements in the k'th bucket
  - $b_k = \frac{w}{m} \pm O(\sqrt{\frac{1}{m}\sum w_j^2})$

Generic Max Sketch Algorithm - Weighted (Stochastic Averaging)

Algorithm 4

1. Use 2 different hash functions
    1. $H_1(x_i) \sim \{1,2,\dots,m\}$
    2. $H_2(x_i) \sim U(0,1)$
2. For every input element x_i:
    1. compute $H_1(x_i)$ and $H_2(x_i)$
    2. transform to $\hat{H_2}(x_i) \sim Beta(w_j, 1)$
3. Let $h_k^+ = \max\{\hat{H_2}(x_i) \mid H_1(x_i) = k\}$ be the maximum observed value in the k'th bucket
4. Invoke $ProcEstimateSA(h_1^+, h_2^+, \dots, h_m^+)$ to estimate w

The Unified Scheme

- Practically, if

$$H_2(x_i) \sim U(0,1)$$

- Then,

$$H_2(x_i)^{\frac{1}{w_j}} \sim Beta(w_j, 1)$$

Distributions Summary

Unweighted	$h_k^+ \sim Beta\left(\frac{n}{\mathrm{m}}, 1\right)$
Weighted	$h_k^+ \sim Beta(\frac{w}{m} = \frac{\sum w_j}{m}, 1)$

The Unified Scheme

- The same algorithm that estimates n in the unweighted case can estimate w in the weighted case

- $ProcEstimateSA()$ is exactly the same procedure used to estimate the unweighted cardinality in Algorithm 3

The Unified Scheme Lemma

Estimating w by Algorithm 4 is equivalent to estimating n by Algorithm 3.

Thus, Algorithm 4 estimates w with the same variance and bias as that of the underlying procedure used by Algorithm 3.

Stochastic Averaging – Effect on Variance (Unweighted)

- Brings computational efficiency at the cost of a delayed asymptotical regime (Lumbroso, 2010)
  - When *n* is sufficiently large, the variance of each bucket size b_k is negligible
  - How large *n* should be to obtain negligible variance of b_k in the unified scheme?

- When the **normalized standard deviation** of each b_k is $< 10^{-3}$, there is negligible loss of statistical efficiency
  - For example, when $\mathrm{n} = 10^6$ and $m = 10^3 \Longrightarrow Var\left[\frac{b_k}{E[b_k]}\right] \approx \frac{m}{n} = 10^{-3}$

Stochastic Averaging – Effect on Variance (Weighted)

- Assuming that $\frac{\sum w_j^2}{w^2} = 10^{-6} \implies$
  - The normalized standard deviation = $Var\left[\frac{b_k}{E[b_k]}\right] \approx \frac{\sum w_j^2}{w^2} m = 10^{-3}$
- However, other choices of the weights may **"delay"** this bound for bigger values of *n*

Stochastic Averaging – Effect on Variance (weighted) Random Distribution of Weights

- Assume that the weights w_j are drawn from a random distribution
- Using the variance definition:

$$\frac{\sum_{j=1}^{n} w_j^2}{w^2} = \frac{n\mathbb{E}\left[w_j^2\right]}{n^2\mathbb{E}^2\left[w_j\right]} = \frac{\mathbb{E}\left[w_j^2\right] - \mathbb{E}^2\left[w_j\right] + \mathbb{E}^2\left[w_j\right]}{n\mathbb{E}^2\left[w_j\right]}$$
$$= \frac{1}{n}\left(1 + \frac{\mathrm{Var}\left[w_j\right]}{\mathbb{E}^2\left[w_j\right]}\right).$$

Therefore,

$$\mathrm{Var}\left[b_k/\mathbb{E}\left[b_k\right]\right] = \frac{\sum_{j=1}^{n} w_j^2}{w^2} m = \frac{m}{n}\left(1 + \frac{\mathrm{Var}\left[w_j\right]}{\mathbb{E}^2\left[w_j\right]}\right).$$

The unified scheme can deal with unbounded number of weights as long as:

1. Weights are positive
2. $Var[w_j]\ /E^2[w_j]$ is a small constant

Transformation Between Distributions

- Each element is hashed $h(x_i) \sim U(0,1)$
- Then,
  - Some estimators transform $h(x_i)$ into another distribution
    - For example, HyperLogLog (Geometrical)
  - The unified scheme transforms $h(x_i)$ into a Beta distribution
    - $\hat{h}(x_i) \sim Beta(w_j, 1)$
- Inverse-Transform Method:

$$u \sim U(0,1) \quad \Rightarrow \quad F^{-1}(u) \sim D$$

where,

- F is the CDF of distribution D
- F is monotonically non-decreasing function
- F^{-1} is the inverse function

Transformation Between Distributions

- In general, $h(x_i)$ is transformed into $\hat{h}(x_i) = F^{-1}\big(h(x_i)\big)$
  - Inverse-Transfom Method

- The estimator may keep the original uniform hashed value
  - Without transformation
  - In this case, $F(x) = x$

source	transformation
$u \sim \mathrm{U}(0,1)$	$u^{\frac{1}{\alpha}} \sim \mathrm{Beta}\,(\alpha, 1)$
$u \sim \mathrm{U}(0,1)$	$(1-(1-u)^{\frac{1}{\beta}}) \sim \mathrm{Beta}\,(1, \beta)$
$u \sim \mathrm{U}(0,1)$	$\lfloor \log_{1-p}(1-u) \rfloor = \left\lfloor \frac{\ln(1-u)}{\ln(1-p)} \right\rfloor \sim \mathrm{Geom}\,(p)$
$u \sim \mathrm{U}(0,1)$	$-\ln u^{1/w} \sim \mathrm{Exp}\,(w)$

Table 2.1: Distribution Transformation Examples

The Unified Scheme

- The desired distribution is $Beta\ (w_j, 1)$
  - CDF: $G_{\max}\ (x) = \ x^{w_j}$
  - CDF inverse: $G_{max}^{-1}(u)\ = \ u^{1/w_j}$
- $G_{max}^{-1}\big(h(x_i)\big) = \ h(x_i)^{\frac{1}{w_j}} \sim Beta\ \big(w_j, 1\big)$
  - Inverse-Transform Method

To sum up:

$$h_k(x_i) \sim U(0,1) \implies h_k(x_i)^{1/w_j} \sim Beta\big(w_j, 1\big)$$

Weighted Generalization for Continuous U(0,1) with Stochastic Averaging

- Chassaing estimator
- Minimal variance unbiased estimator (MVUE)
- The estimator uses uniform variables
  - No transformation is needed, $\mathrm{F}^{-1}(u) = u$
- Estimate = $\dfrac{m(m-1)}{\sum(1-h_k^+)}$
- Standard error = $1/\sqrt{m}$
- Storage size $32 * m$ bits

To generalize this estimator

Estimate = $\dfrac{m(m-1)}{\sum(1-h_k^+)}$

But now,

$$h_k^+ = \max\{\hat{h_k}(x_i)\} = \max\{h_k(x_i)^{1/w_j}\}$$

Weighted Generalization for Continuous U(0,1) with m hash functions

- Maximum likelihood estimator

- The estimator uses exponential random variables with parameter 1
  - $F^{-1}(u) = -\ln u \sim Exp(1)$

- Estimate = $\frac{m}{\sum h_k^+}$
  - where $h_k^+ = \max\{-\ln(h_k(x_i))\}$

- Standard error = $1/\sqrt{m}$
- Storage size $32 * m$ bits

Weighted Generalization for Continuous U(0,1) with m hash functions

To generalize this estimator

Estimate = $\frac{m}{\sum h_k^+}$

But now,

$$h_k^+ = \max\{-\ln(\hat{h_k}(x_i))\} = \max\left\{-\ln(h_k(x_i)^{\frac{1}{w_j}})\right\}$$

This generalization is identical to the algorithm presented by Cohen, 1995

Weighted HyperLogLog with Stochastic Averaging

- Best known algorithm in terms of the tradeoff between precision and storage size
- The estimator uses geometric random variables with success probability ½
  - $F^{-1}(u) = \lfloor -\log_2 u \rfloor \sim Geom\,(1/2)$

- Estimate = $\frac{\alpha_m m^2}{\sum 2^{-h_k^+}}$
  - where $h_k^+ = \max\{\lfloor -\log_2 H_2(x_i) \rfloor \mid H_1(x_i) = k\}$

- Standard error = $1.04/\sqrt{m}$
- Storage size $5 * m$ bits

Weighted HyperLogLog with Stochastic Averaging

To generalize this estimator

Estimate = $\frac{\alpha_m m^2}{\sum 2^{-h_k^+}}$

But now,

$$h_k^+ = \max\{\lfloor -\log_2 H_2(x_i)^{1/w_j} \rfloor \quad | \quad H_1(x_i) = k\}$$

- **The extended algorithm offers the best performance, in terms of statistical accuracy and memory storage, among all the other known algorithms for the weighted problem**

Conclusion

- We showed how to generalize every min/max sketch to a weighted version
- **The scheme can be used for obtaining known estimators and new estimators in a generic way**

- The proposed unified scheme uses the unweighted estimator as a black box, and manipulates the input using properties of the Beta distribution

- We proved that estimating the weighted sum by our unified scheme is statistically equivalent to estimating the unweighted cardinality

- In particular, we showed that the new scheme can be used to extend the HyperLogLog algorithm to solve the weighted problem
- **The extended algorithm offers the best performance, in terms of statistical accuracy and memory storage, among all the other known algorithms for the weighted problem**

Efficient Detection of Application Layer DDoS Attacks by a Stateless Device

DoS and DDoS

Denial of Service Attack (DoS)

- Malicious attempt to make a server or a network resource unavailable to users
- The most common type is flooding the target resource with external requests.
  - The overload prevents/slows the resource from responding to legitimate traffic

Distributed Denial of Service Attack (DDoS)

- DoS attack where the attack traffic is launched from multiple distributed sources.
- A DDoS attack is much harder to detect
  - Multiple attackers to defend against

Application DDoS Attacks

- Seemingly legitimate and innocent requests whose goal is to force the server to allocate a lot of resources in response to every single request
- Can be activated from a small number of attacking computers
- Examples:
  - HTTP request attacks:
    - Legitimate, heavy HTTP requests are sent to a web server, in an attempt to consume a lot of its resources.
    - Each request is very short, but the server needs to work very hard to serve it.
  - HTTPS/SSL request attacks
    - Work against certain SSL handshake functions, taking advantage of the heavy computation use by SSL
  - DNS request attacks
    - The attacker overwhelms the DNS server with a series of legitimate or illegitimate DNS requests

Application DDoS Attacks

Application DDoS attacks are more difficult to deal with than classical DDoS:

- The **traffic pattern** is indistinguishable from legitimate traffic
- The number of **attacking machines** can be significantly smaller
  - Typically, it is enough for the attacker to send only hundreds of resource intensive requests, instead of flooding the server with millions of TCP SYNs, as in a volumetric DDoS attack

DDoS Protection Architecture

- Mostly multi-tier:

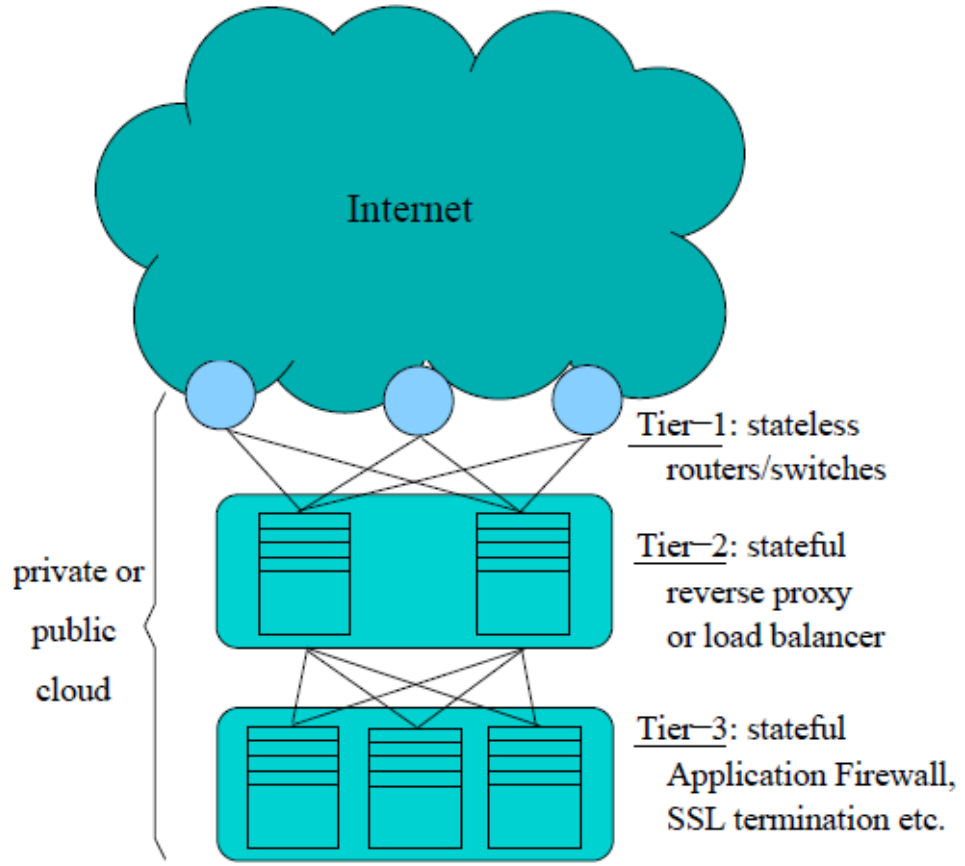


Figure 1.1: A multi-tier DDoS protection architecture

DDoS Protection Architecture

- As strong as its weakest link
  - Often this weakest link is tier-2 or 3
  - Will be the first to collapse in a targeted Application layer DDoS attack.

- It is generally assumed that Application layer attacks cannot be detected by the first tier devices, but only by tier-2 and tier-3 devices, which are stateful, this is because:
  - Many devices
  - Does not have **flow awareness,** cannot perform per-flow tasks
  - Dedicated to **fast performance,** its processing tasks must be simple and cheap
  - Lacks deep knowledge of the **end applications,** and is unable to keep track of the association between **packets-flows-applications**

Previous Work

- Stateless devices usually estimate the load imposed on a remote server by estimating the number of distinct flows
  - Cardinality estimation problem

- Can detect anomalies when the number of distinct flows becomes **suspiciously high**
  - Possibly DDoS attack
  - Alternative: monitor the entropy of selected attributes in the received packets and compare to pre-computed profile

- Previously proposed schemes have considered all flows as imposing the **same load**
  - This is clearly not true in a realistic case where high-workload requests require significantly more server efforts than simple ones
  - We solve this problem by preclassifying the incoming flows and associating them with different weights according to their load

Our Contribution

- We show how a tier-1 stateless device can acquire significant Application layer information and detect Application layer attacks

- Early detection will afford better overall protection
  - Triggers the opening of more tier-2 and tier-3 devices
  - Triggers the invocation of special tier-1 packet-based filtering rules, which will reduce the load

Basic Scheme

- Main idea:
  - classify incoming flows according to the load each of them imposes on the server
  - flows that impose different loads should be mapped in advance into different TCP/UDP ports
    - Consequently, a stateless router that receives a packet can look at the Protocol field and the destination port number in the packet's header in order to know the load imposed on the server by the flow to which the packet belongs
  - The total load imposed on the end server during a specific time interval is

$$w = \sum_{l=1}^{C} w_l * n_l$$

    - C is the number of weight classes
    - n_l is the number of flows belonging to class l
  - **execute an algorithm that estimates the number of flows for each class.**

Basic Scheme

Formally,

1. preclassify the flows according to their loads $w_1, w_2, \ldots, w_C$;
2. estimate the number of flows for each class $\widehat{n_l}$;
3. return $\widehat{w} = \sum_{l=1}^{C} w_l \cdot \widehat{n_l}$.

The total load imposed on the end server during a specific time interval is

$$w = \sum_{l=1}^{C} w_l * n_l$$

- C is the number of weight classes
- n_l is the number of flows belonging to class l

The problem of measuring the total load imposed on the web server during a specified time is now translated into the problem of estimating the number of flows for each class of weights.

HyperLogLog

Algorithm 3.1 *The HyperLogLog algorithm for estimating the number of flows*

1. *Initialize* m *registers:* $C_1, C_2, \ldots, C_m$ *to 0.*
2. *For each monitored packet whose flow ID is* x_i *do:*
   (a) *Let* $\rho = \lfloor -\log_2 (h_1(x_i)) \rfloor$ *be the leftmost 1-bit position of the hashed value.*
   (b) *Let* $j = h_2(x_i)$ *be the bucket for this flow.*
   (c) $C_j \leftarrow \max \{C_j, \rho\}$.
3. *To estimate the value of* n *do:*
   (a) $Z \leftarrow \left(\sum_{j=1}^{m} 2^{-C_j}\right)^{-1}$ *is the harmonic mean of* 2^{C_j}.
   (b) *return* $\alpha_m m^2 Z$, *where*
   $\alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u}\right)\right)^m du\right)^{-1}$.

Example: HTTP

Assign the same TCP port to all HTTP requests that impose the same load on server:

- Requests that require a lot of processing can be assigned to port 8090 (weight w_1)
- Requests that require slightly less are assigned to port 8091 **(with weight $w_2 < w_1$)**
- And so on…

Implementation

- Straightforward for every Application layer protocol that admits a **one-to-one** mapping to a **TCP or a UDP port**
  - Each TCP or UDP flow is associated with one application layer instance

- However, not the case for HTTP, because of **"persistent connection"** property.
  - Allows the client to send multiple HTTP requests over the same TCP connection (flow)
  - Cannot tell in advance which or how many requests will be sent over the same connection

- The solution we propose is to map all light requests to **one port,** and to map each heavier request to its **own port**
  - The weight associated with the light requests will take into account their resource consumption and the possibility that multiple light requests may **share** the same connection

Enhanced Scheme

- Main idea:
  - Instead of solving the cardinality estimation problem once per each class, the enhanced scheme solves the weighted cardinality estimation problem
  - The total load is estimated directly, without estimating the number of flows in each class

- **The enhanced scheme with m/C storage units performs better (has much better variance) than any configuration of the basic scheme, even if the latter uses factor C more storage units.**
  - Moreover, the enhanced scheme is agnostic to the distribution of the weights and does not need a priori information about the distribution of the weight classes

Weighted HyperLogLog

Algorithm 3.2 *A generalization of Algorithm 3.1 for estimating the weighted cardinality of the flows*

1. *Initialize* m *registers:* $C_1, C_2, \ldots, C_m$ *to 0.*
2. *For each monitored packet whose flow ID is* x_i *with weight* w_j *do the same as Algorithm 3.1, except that* $\rho = \left\lfloor -\log_2 \left(h_1(x_i)\right)^{\frac{1}{w_j}} \right\rfloor$.
3. *To estimate the weighted sum do:*
   (a) $Z \leftarrow \left(\sum_{j=1}^{m} 2^{-C_j}\right)^{-1}$ *is the harmonic mean of* 2^{C_j}.
   (b) *return* $\alpha_m m^2 Z$, *where*
   $\alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u}\right)\right)^m du\right)^{-1}$.

Basic Scheme vs. Enhanced Scheme

- **Minimal variance of basic scheme = $\frac{w^2}{m-2C} > \frac{w^2}{m-2}$ = variance of enhanced scheme**
- The enhanced scheme has smaller variance than the minimal variance of the basic scheme
- When the number of different classes $C > \frac{m}{2}$, then the variance of the basic scheme is infinite.
  - Moreover, even if there are only a few classes, and the statistical inefficiency can be tolerated, the basic scheme needs a priori information on the distribution of the weights, while the enhanced scheme does not.
- **The enhanced scheme with m/C storage units performs better (has much better variance) than any configuration of the basic scheme, even if the latter uses factor C more storage units.**
  - as long as the number of weight classes satisfy $C > \frac{m}{2}$, and this requirement is satisfied because m is usually very small.

Basic Scheme vs. Enhanced Scheme

- **Minimal variance of basic scheme = $\frac{w^2}{m-2C} > \frac{w^2}{m-2}$ = variance of enhanced scheme**

$$\widehat{n_l} \sim \mathcal{N}\left(n_l, \frac{{n_l}^2}{m_l - 2}\right).$$

$$\mathrm{Var}\left[\widehat{w}\right] = \mathrm{Var}\left[\sum_{l=1}^{C} w_l \cdot \widehat{n_l}\right] =$$

$$\sum_{l=1}^{C} w_l^2 \cdot \mathrm{Var}\left[\widehat{n_l}\right] = \sum_{l=1}^{C} \frac{w_l^2 \cdot n_l^2}{m_l - 2}.$$

$$J\left(m_1, m_2, \ldots, m_C\right) = \sum_{l=1}^{C} \frac{w_l^2 \cdot n_l^2}{m_l - 2} \quad \text{where} \quad m = \sum_{l=1}^{C} m_l.$$

The optimal choice can be found using Lagrange multipliers:

$$\widetilde{J}\left(m_1, m_2, \ldots, m_C, \lambda\right) = \sum_{l=1}^{C} \frac{w_l^2 \cdot n_l^2}{m_l - 2} - \lambda\left(m - \sum_{l=1}^{C} m_l\right).$$

Using derivatives and solving the equation yield:

$$\frac{\partial \widetilde{J}}{\partial m_l} = -\frac{w_l^2 \cdot n_l^2}{(m_l - 2)^2} + \lambda = 0.$$

$$(m_l - 2) = w_l \cdot n_l / \sqrt{\lambda}.$$

Estimating the Load Variance

- Main idea:
  - The weighted algorithm is useful for performing management tasks
    - Adding a virtual machine to a web server
    - Adjusting the load balancing criteria, etc…
  - Not useful for detecting an extreme and sudden increase in the load imposed on the server due to an **Application layer attack.**

- Definitions:
  - n(t) = number of active flows sampled at time *t* over the last *T* units of time
  - w(t) = weighted sum of these flows

Estimating the Load Variance

- $\widehat{w(t)}$ is a random variable that estimates the weighted sum of the flows sampled during time interval $[t - T, t]$
- Unbiased estimator, we get that

$$\mathbb{E}\left[\widehat{w(t)}\right] = w(t) = \sum w_j(t),$$

$$\mathbb{E}\left[\widehat{w(t)^2}\right] = \sum w_j(t)^2.$$

$$\begin{aligned}\mathrm{Var}\left[\widehat{w(t)}\right] &= \mathbb{E}\left[\widehat{w(t)^2}\right] - \mathbb{E}\left[\widehat{w(t)}\right]^2 \\ &= \sum w_j(t)^2 - (\sum w_j(t))^2 \\ &= \sum w_j(t)^2 - w(t)^2.\end{aligned}$$

Load Variance

Algorithm 3.3 *Estimating the variance of the weighted sum of the flows sampled at* $[t-T, t]$

1. *Estimate* $w(t)$ *using Algorithm 3.2;*
2. *Estimate* $\sum w_j(t)^2$ *using Algorithm 3.2; this time, associate each flow with the square of its original weight, i.e.,* w_j^2*;*
3. *Compute and return* $\mathrm{Var}\left[\widehat{w(t)}\right]$ *using Eq. 3.4.*

- **Variance can be affected not only by excessive load imposed by a few connections originated by an attacker, but also by an excessive number of new legitimate connections.**
- **To distinguish between the two cases, we normalize the variance by dividing it by the number of flows *n*.**

Normalized Load Variance

Algorithm 3.4 *Estimating the normalized variance of the weighted sum of the flows sampled at* $[t-T, t]$

1. *Estimate* $n(t)$ *using Algorithm 3.1;*
2. *Estimate* $\mathrm{Var}\left[\widehat{w(t)}\right]$ *using Algorithm 3.3;*
3. *Return* $\frac{\sum w_j(t)^2 - w(t)^2}{n(t)}$.

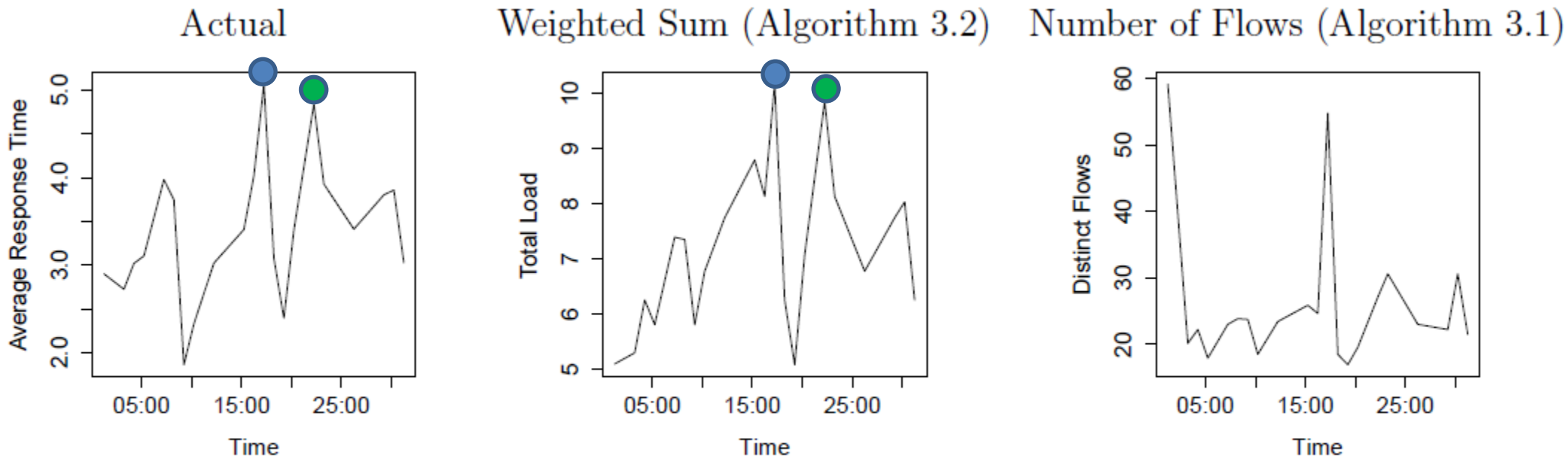
Simulation Results

Detecting the load imposed on a server

- We study the requests received by the main web server of the Technion campus
- Assign to each request a weight that represents the load it imposes on the server
- Compare the results of the weighted scheme to the results of two benchmarks:
- **Actual:**
  - Determines the real load imposed on the web server during every considered time interval by computing the server's average response time.
  - Actual is expected to outperform our scheme
  - Of course, such a scheme cannot employed by a stateless intermediate device
- **Number of Flows:**
  - Uses HyperLogLog to estimate the number of distinct flows during each time period.
- How to determine in advance the load imposed on the server by every request?
  - Because we do not have access to the server, but only to its log files, we assign weights according to the average size of the response file sent by the server to each request

Simulation Results

Detecting the load imposed on a server



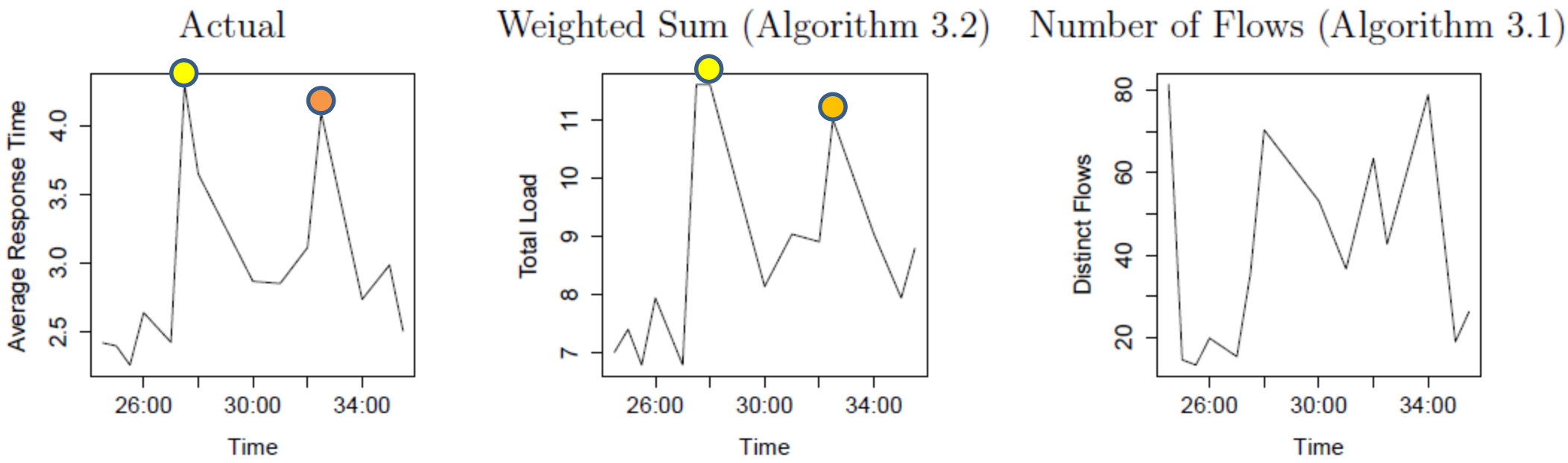
(a) Trace 1: 30 minutes with $\Delta = 60$ seconds

We can see a strong correlation between the load estimated by our scheme and Actual:

- For example, Actual shows a temporary heavy load on the server after 17 minutes, a load that is clearly detected by our scheme (in blue)
- Another peak, at t = 22, is also detected by our scheme (in green)

Simulation Results

Detecting the load imposed on a server



(b) Trace 2: 10 minutes with $\Delta = 30$ seconds

We can see a strong correlation between the load estimated by our scheme and Actual:

- Actual shows temporary heavy loads on the server at t = 28 (yellow) and t = 32 (orange), both clearly detected by our scheme as well.

Simulation Results

Detecting the load imposed on a server

- For mathematical corroboration, we measured the Pearson correlation coefficient between Actual and our scheme.
- Let S_0 be the vector of the values of Actual, and S_1 be the vector of the values of our scheme. Then:

$$\mathrm{Cor}\,[S_1, S_0] = \frac{\mathrm{Cov}\,[S_1, S_0]}{\sqrt{\mathrm{Var}\,[S_1]\,\mathrm{Var}\,[S_0]}}.$$

- This ratio varies between 1 and −1:
  - the closer it is to either (−1) or to 1, the stronger the correlation between the variables;
  - the closer it is to 0, the weaker the correlation.
- **Actual vs. our scheme:**
  - In the first trace we find that the correlation coefficient is **0.85,** which indicates a very strong correlation between Actual and our scheme.
  - In the second trace, the correlation coefficient is **0.92,** indicating even stronger correlation

Simulation Results

Detecting the load imposed on a server

- We then measured the Pearson correlation coefficient between Actual and Number-of-Flows

- In contrast to the strong correlation between our scheme and Actual, we can see that the correlation between Number-of-Flows and Actual **is very weak**
  - In the first trace, the correlation coefficient is only 0.38
  - In the second trace, the correlation coefficient is 0.23

- More specifically,
  - In the first trace, the peak after 22 minutes **is not identified** by the Number-of-Flows scheme
  - Moreover, the Number-of-Flows scheme identifies **false heavy loads,** for example after 1 minute

Simulation Results

Detecting Application Layer DDoS Attacks

- We use Wireshark to capture video sessions from YouTube, and manually add three Application DDoS attacks to the original data:
  - **a)** **attack-1** is represented by 30 downloads of a 1-minute video stream starting at 10:00;
  - **b)** **attack-2** is represented by 40 downloads of a 1-minute video stream starting at 20:00;
  - **c)** **attack-3** = 50 downloads of a 1-minute video stream starting at 06:00.

- We estimate the load variance and the normalized load variance every $\Delta = 60$ seconds, for $T = 1$ minute.

Simulation Results

Detecting Application Layer DDoS Attacks

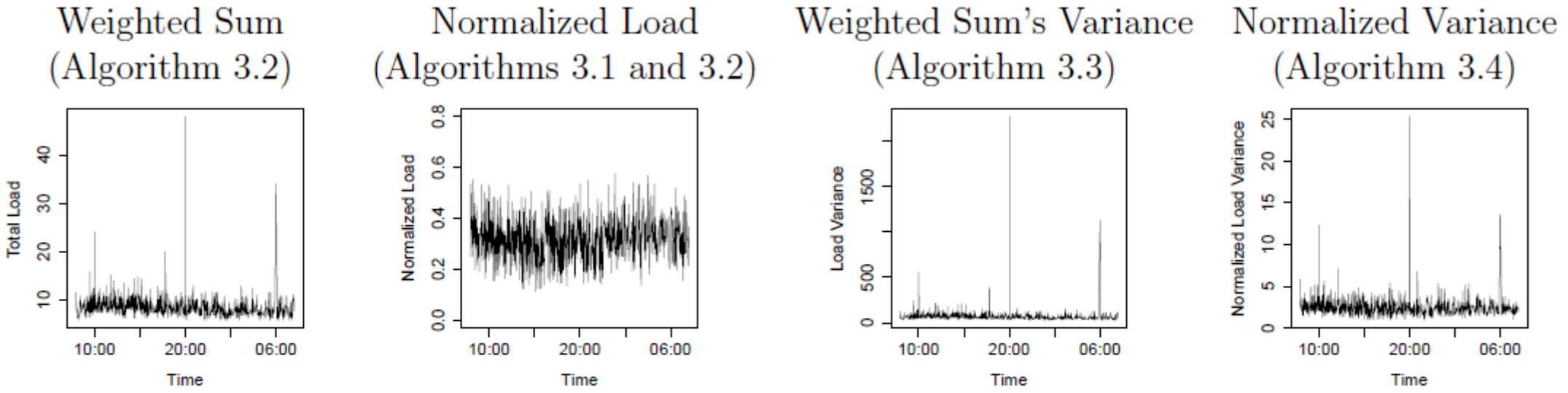


Figure 3.2: The performance of the various schemes for detecting Application layer attacks

One can easily see that Normalized Load scheme does not detect any of the attacks.

- This scheme is able to detect only attacks created by a small number of connections that generate a lot of traffic.
- Although all the attacks added to our log files were triggered by only 30-50 connections, they nonetheless had only a slight effect on the average amount of traffic per connection.

The three other schemes successfully detect the three attacks.

Simulation Results

Detecting Application Layer DDoS Attacks

- For comparing their performance, we define the minimal relative error as

$$\lambda = \frac{X_{\text{attack}} - X_{\text{false}}}{X_{\text{false}}},$$

- where,
  - X_{attack} is the minimal value computed by the scheme during an attack
    - i.e., the minimal value among its values at 10:00, 20:00 and 06:00
  - X_{false} is the maximal value of the scheme during normal times

Simulation Results

Detecting Application Layer DDoS Attacks

- For weighted HyperLogLog:
  - X_{attack}= 23.85 (for the attack at 10:00)
  - The maximal value at a normal time is X_{false} = 20.21 (at 18:00)

- Thus, the minimal relative error is $\lambda = \frac{23.85 - 20.21}{20.21} = 0.18$

- **Therefore, for this algorithm the minimal value representing an attack is (only) 18% larger than the nearest normal value**

Simulation Results

Detecting Application Layer DDoS Attacks

- λ allows to compare the performance of the schemes and their false-detection rates
  - As λ grows, the scheme more accurately distinguish between attack and normal traffic

- For Algorithm 3.3: $\lambda = \frac{525.69-420.18}{420.18} = 0.25$;
- For Algorithm 3.4: $\lambda = \frac{12.32-7.13}{7.13} = 0.73$.

- Among the three schemes, the Normalized Load Variance has the best performance.
  - Largest λ
  - **In particular, the two other schemes detect a false attack at 18:00**

Simulation Results

Detecting Application Layer DDoS Attacks

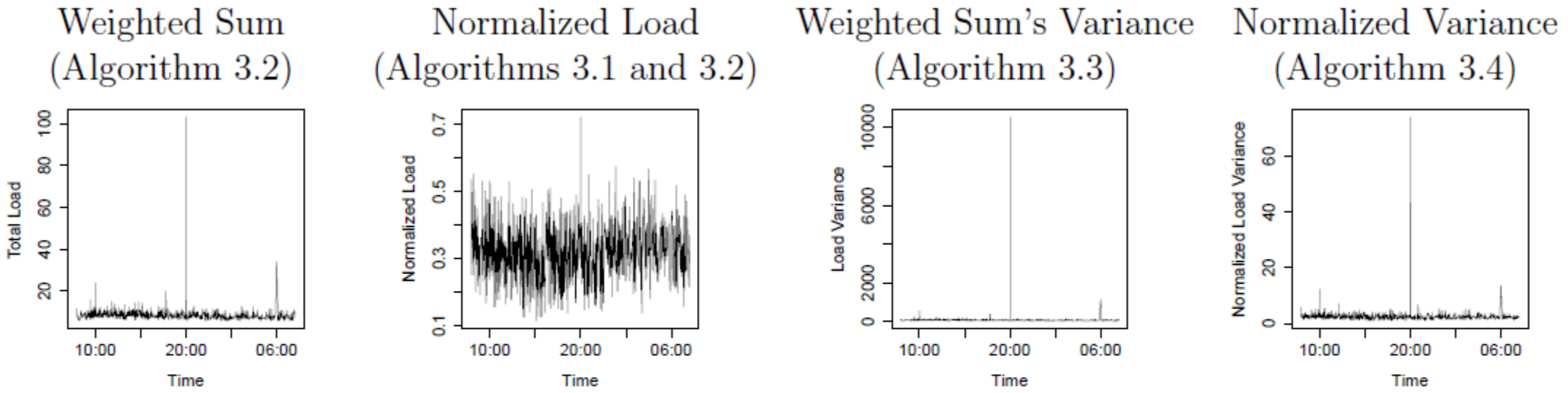


Figure 3.3: The various schemes after changing the second attack

In this scenario we change attack-2 to consist of 100 downloads of a 1-minute video stream starting at 20:00

The main different between the two figures is that this time the Normalized Load scheme successfully detects attack-2 (at 20:00). However, this scheme still does not detect the two other attacks: at 10:00 and at 6:00.

In terms of the other schemes, the Normalized Load Variance scheme still has the best performance, because the other two schemes detect a false attack at 18:00.

A Sliding Window Extension

- For continuous monitoring, one needs to estimate the total load every Δ seconds for a window of the last T minutes

- Instead of *m* buckets for a single estimator, one needs to maintain $\frac{T}{\Delta} * m$ buckets
  - For instance, for $T = 10$, $\Delta = 1$, if a packet is received at *t* = 15, its sketch is compared to the maximum obtained for the intervals [6, 15], [7, 16], . . . , [15, 24]

- The sliding window extension was proposed by Chabchoub (2010) for the unweighted HyperLogLog algorithm to avoid this penalty

A Sliding Window Extension

- The procedure stores not only the current maximum value, but all hashed values that may be maximum in any future window of time

- To this end, the algorithm maintains for each bucket a sorted list of the received packets, each consisting of a pair (R_k, t_k)
  - t_k is the arrival time of the packet
  - R_k is the hashed value of the packet's flow ID

List Update Procedure

LFPM (List of Possible Future Maximums), and it is updated for each packet as follows:

Procedure LFPM Update
For each monitored packet received at t do:

1. Let R be the hashed value of the flow ID to which this packet belongs.
2. Keep the record $[R, t]$, because every new record received at t can be a maximum during $[t, t+T]$.
3. Remove all the records $[*, t']$ for every $t' < t + \Delta - T$, because none of them can be a maximum in any relevant time window.
4. Remove all the records $[R', *]$ for every $R' < R$. Again, none of these records can be a maximum during any relevant interval, because all the relevant intervals contain the time t.

List Update Procedure

Time	1	2	3	4	5	6	7
Hashed value	0.3	0.5	0.2	0.1	0.7	0.4	0.8

Time	LFPM
1	(0.3,1)
2	(0.5,2)
3	(0.5,2) ; (0.2,3)
4	(0.5,2) ; (0.2,3) ; (0.1,4)
5	(0.7,5)
6	(0.7,5) ; (0.4,6)
7	(0.8,7)

Figure 3.4: An example of the execution of Procedure LFPM Update

A Sliding Window Extension

- We use the LFPM idea to extend the weighted HyperLogLog algorithm to get a sliding window version of the weighted problem
- The stochastic averaging process of splitting the packets into *m* buckets remains unchanged; namely, a different LFPM is maintained for each bucket
- Clearly, when a packet is received, only its matched list (according to its bucket) is updated
- To estimate the weighted cardinality at time t:
  - Extract the relevant packets from the LFPM, and compute the highest C_j among them
    - Independently for each bucket
  - The rest of the estimation is as specified in the weighted HyperLogLog algorithm

A Sliding Window Extension

Algorithm 3.5 *A sliding window extension of Algorithm 3.2*

1. *Initialize the m LFPMs to be empty.*
2. *For each monitored packet whose flow ID is x_i and flow weight is w_j do:*
   (a) *set ρ and j as specified in Algorithm 3.2.*
   (b) *update $LFPM_j$ as specified in Procedure LFPM Update.*
3. *To determine $\sum_{j=1}^{n} w_j$ at time t for all the flows detected during $[t-T, T]$ do:*
   (a) *compute C_j for each $LFPM_j$.*
   (b) *return $\widehat{w}$ as specified in Algorithm 3.2.*

The update procedure of the LFPM list does not affect the computation of the maximal hash values.
It is simply an efficient method for computing its exact value at any time, by storing only a short list of packets.

Therefore, the extended algorithm has the same bias and variance as the weighted HyperLogLog

Analysis of Memory Cost

- For HyperLogLog = $m * \log\left(\log\left(\frac{n}{m}\right)\right)$ bits
- What is the LFPM's penalty on memory?

Lemma 3.1

Assume that n active flows are sampled during some time interval and m buckets are used. Let L_n be the size of a single LFPM at the end of the considered interval. Then, $\mathbb{E}[L_n] = H_n \approx \ln n$, and $\mathrm{Var}[L_n] = H_n - \sum_{i=1}^{n} \frac{1}{i^2} \approx \ln n$, where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the Harmonic series.

Lemma 3.2

Assume that n active flows are sampled during some time interval and m buckets are used. Let L_n be the size of a single LFPM at the end of the considered interval, and L_n^m be the total required memory for storing the m LFPMs. Then, for every $\alpha > 0$, the following holds:

$$\Pr\left(L_n^m > m\ln(n/m) + \alpha\sqrt{m\ln(n/m)}\right) < e^{-\alpha/6}.$$

Conclusion

- **We show that both the basic scheme and the enhanced scheme allow the stateless device to estimate the total load imposed on the Application layer of a server**
  - We compare the performance of the enhanced scheme and the basic scheme and show that the enhanced scheme provides a much better variance
  - However, they **do not detect** an extreme and sudden increase in the load due to an attack

- We present two additional schemes, that use the enhanced scheme as a building block
  - The first scheme estimates the variance of the weighted sum of the flows
  - The second estimates the normalized variance of the weighted sum of the flows

- **We show that the latter one is the best for detecting Application layer attacks**

Future Research

Combining Cardinality Estimation and Sampling

- We will study the problem of estimating the number of distinct elements in a stream when only a small sample of the stream is given

- Real-world applications of cardinality estimation algorithms are required to process **large volumes of monitored data,** making it impractical to collect and analyze the entire stream
  - For example, IP packets over a high-speed link; 100 Gbps link creates a 1 TB log file in < 1.5 minutes
  - Must sample and process only a small part of the stream
  - sFlow

- Sampling techniques provide **greater scalability,** but they also make it more difficult to infer the characteristics of the full stream

Data Sampling

- Only a small sample of the data is collected (marked in red), and then analyzed

- The sample stream is

D , B , D , D

- Sample cardinality = 2
- **What is the cardinality of the full (unsampled) stream?**
  - Is it = 2? (equal to the sample cardinality)
  - Is it = 2*2 = 4? (inverse to the sampling rate)
  - Something else?

Combining Cardinality Estimation and Sampling

- We will present and analyze a generic algorithm for combining every cardinality estimation algorithm with a sampling process

- Notations:
  - X full (unsampled) stream
  - Y sample stream
  - n = cardinality of full stream
  - n_s = cardinality of sample stream

- The proposed algorithm will consist two main steps:
  a) Cardinality estimation of the sampled stream using any known cardinality estimator
  b) Estimation of the sampling ratio n/n_s

Good-Turing Frequency Estimation

- Useful in language-related tasks where one needs to determine the probability that a word will appear in a document

- Given a sample Y of a bigger stream X
- Notations:
  - P_i = the probability that an element of X appears in the sample i times
  - $|E_i|$ = the number of elements that appear exactly i times in the sample

- Good-Turing claims that $\widehat{P_i} = (i+1)\frac{|E_{i+1}|}{l}$ is a consistent estimator for the probability P_i

- For P_0, $\widehat{P_0} = |E_1|/l$
  - The probability that an element of X **does not appear** in the sample at all **(unseen element)**

- In other words, the hidden mass can be estimated by the relative frequency of the elements that appear exactly once in the sample *Y*.

The Proposed Algorithm

- To estimate n_s in step (a), any cardinality estimation algorithm can be used
- To estimate n/n_s in step (b), we note that $P_0 = \frac{n - n_s}{n}$ and thus $\frac{1}{1 - P_0} = n/n_s$.
- Therefore, the problem of estimating n/n_s is reduced to estimating the probability P_0 of unseen elements
  - Using Good-Turing
  - Need to find the number of elements that appear exactly once in the sampled stream

The Proposed Algorithm

Algorithm 4.1
(cardinality estimation with sampling)

(a) Estimate the number n_s of distinct elements in the sample Y by invoking a cardinality estimation procedure (Procedure 1) on this sample using m storage units.

(b) Determine the ratio n/n_s by computing $\frac{1}{1-\widehat{P_0}}$, where $\widehat{P_0} = |E_1|/l$. The value of $|E_1|$ is computed precisely and l is known.

(c) Return $\widehat{n} = \widehat{n_s} \cdot \widehat{n/n_s}$ as an estimator for the cardinality of the entire stream X.

To compute the value of $|E1|$ precisely, one should keep track of all the elements in *Y* and ignore each previously encountered element.
This is done using $O(l)$ storage units.

Combining Cardinality Estimation and Sampling

- We intend to show that the proposed algorithm does not affect the asymptotic unbiasedness of the original estimator.
- We will also analyze the sampling effect on the asymptotic variance of the estimators.
- Another goal is to **reduce** the memory cost
  - Uses $O(l)$ storage units, which is linear in the sample size
  - We hope to reduce this cost by estimating the value of $\frac{|E_1|}{l}$, instead of computing it precisely

Estimation of Set Intersection

- Given: two sets of elements A and B.
- Goal: estimate the size of their intersection $|A \cap B|$
  - Well known problem, arises in network monitoring, database systems, data integration and information retrieval
  - Many application scenarios are in the context of Network Functions Virtualization (NFV)

- For example, the sets might include IP addresses of packets passing through a router, and our goal is to determine if the two sets are similar to each other

- Can be found exactly
  - **Does not scale if storage is limited**

Estimation of Set Intersection

- Jaccard similarity

$$\rho = \frac{|A \cap B|}{|A \cup B|}.$$

- Three previously proposed schemes:
  1. Using inclusion-exclusion principle $|A \cap B| = |A| + |B| - |A \cup B|\,.$
  2. Using Jaccard similarity (1) $\widehat{|A \cap B|} = \widehat{\rho} \cdot \widehat{|A \cup B|}.$
  3. Using Jaccard similarty (2) $\widehat{|A \cap B|} = \frac{\widehat{\rho}}{\rho + 1}(\widehat{|A|} + \widehat{|B|}).$

- **No previous analysis of them**

Estimation of Set Intersection

- We will analyze the schemes
  - Efficiency
  - Statistical performance (bias and variance)
  - Compare between them

- We will then compute the optimal variance of any unbiased estimator
  - Cramer-Rao bound
  - The variance of any unbiased estimator is at least as high as the inverse of Fisher information

Estimation of Set Intersection

- We will present a new estimator
  - Maximum Likelihood (ML) method
  - We hope to prove analytically, and/or using simulations, that our new ML estimator outperforms all previously known schemes

- We will address the problem of estimating the cardinality of more than two sets

New Results

Analysis of Proposed Algorithm

- We showed that the proposed algorithm does not affect the asymptotic unbiasedness of the original estimator and analyzed the sampling effect on the asymptotic variance of the estimators.

Theorem 1

Algorithm 1 estimates n with mean value n and variance $\frac{n^2}{l}\frac{P_0(1-P_0)+P_1}{(1-P_0)^2}+\frac{n^2}{m}$, namely, $\widehat{n} \to \mathcal{N}\left(n, \frac{n^2}{l}\frac{P_0(1-P_0)+P_1}{(1-P_0)^2}+\frac{n^2}{m}\right)$, where l is the sample size, and m is the storage size used for estimating n_s. In addition, P_0 and P_1 satisfy:

1. $\mathbb{E}[P_0] = \frac{1}{n}\sum_{i=1}^{n} e^{-P\cdot f_i}$.
2. $\mathbb{E}[P_1] = \frac{P}{n}\sum_{i=1}^{n} f_i \cdot e^{-P\cdot f_i}$

where f_i is the frequency of element e_i in X.

New Algorithm with Subsamplimg

- The proposed algorithm computes $|E_1|$ **precisely**
  - Uses $O(l)$ storage units, linear in sample size.
- We reduce this cost by approximating the value of $|E_1|$ using a subsample *U* of *Y*

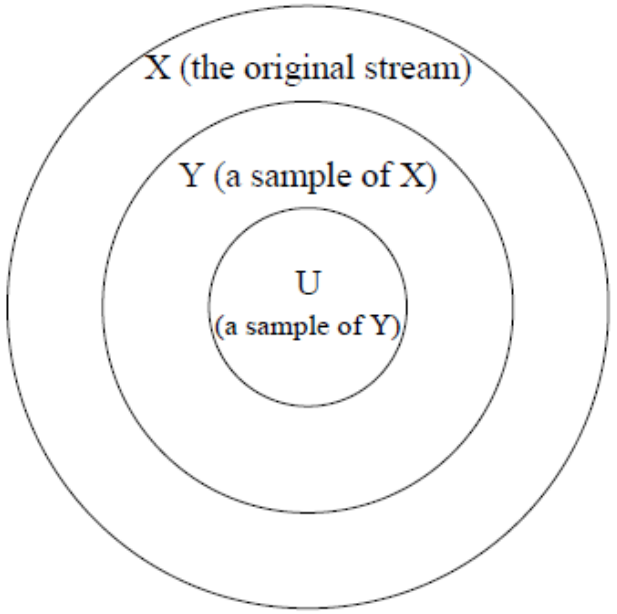


Figure 1: The relationship between X, Y and U

New Algorithm with Subsamplimg

Algorithm 2
(cardinality estimation with sampling and subsampling)
Same as Algorithm 1, except that in step (b) the ratio $|E_1|/l$ is estimated by invoking Procedure 2 using only $u \ll l$ storage units.

Procedure 2:

1. Uniformly subsample u elements from the sampled stream Y. Let this subsample be U.
2. Compute (precisely) the number $|U_1|$ of elements that appear only once in U.
3. Return $\widehat{P_0} = |U_1|/u$.

New Algorithm with Subsamplimg

- Uniform subsampling using one-pass reservoir sampling
- Running-time complexity:
  - $O(l + u \cdot \log l) = O(l)$
  - Similar to proposed algorithm
- Main advantage is in storage:
  - Algorithm 2: $m + u$ units
  - Algorithm 1: $m + l$ units, where $u \ll l$

Analysis of New Algorithm

- We showed that the proposed algorithm does not affect the asymptotic unbiasedness of the original estimator and analyzed the sampling effect on the asymptotic variance of the estimator.

Theorem 2

Algorithm 2 estimates n with mean value n and variance $\frac{n^2}{u}\frac{2P_0(1-P_0)+P_1}{(1-P_0)^2}+\frac{n^2}{m}$, namely, $\widehat{n} \to \mathcal{N}\left(n, \frac{n^2}{u}\frac{2P_0(1-P_0)+P_1}{(1-P_0)^2}+\frac{n^2}{m}\right)$. In addition, P_0 and P_1 can be estimated as described in Theorem 1.

Analysis of New Algorithm

- In the analysis, Procedure 1 = HyperLogLog
- Asymptotic relative efficiency (ARE) = $\frac{n^2}{m} \,/\, Var(\hat{n})$
- Generalization for any cardinality estimation procedure:

Theorem 3

Algorithm 2 estimates n with mean value n and variance $\frac{n^2}{u}\frac{2P_0(1-P_0)+P_1}{(1-P_0)^2} + \frac{1}{ARE}\frac{n^2}{m}$, namely, $\widehat{n} \to \mathcal{N}\left(n, \frac{n^2}{u}\frac{2P_0(1-P_0)+P_1}{(1-P_0)^2} + \frac{1}{ARE}\frac{n^2}{m}\right)$, where ARE *is the asymptotic relative efficiency of Procedure 1. In addition, P_0 and P_1 can be estimated as described in Theorem 1.*

Questions?

Thank You