
Count-Min Sketches: Theory & Applications

Nagiza F. Samatova, samatova@csc.ncsu.edu

Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

Theory: Count-Min (CM)

An Improved Data Stream Summary: The Count-Min Sketch and its Applications

Graham Cormode* and S. Muthukrishnan**

Abstract. We introduce a new sublinear space data structure—the *Count-Min Sketch*—for summarizing data streams. Our sketch allows fundamental queries in data stream summarization such as point, range, and inner product queries to be approximately answered very quickly; in addition, it can be applied to solve several important problems in data streams such as finding quantiles, frequent items, etc. The time and space bounds we show for using the CM sketch to solve these problems significantly improve those previously known — typically from $1/\epsilon^2$ to $1/\epsilon$ in factor.

Theory for CM Applications

Question(s)	Algorithm(s)
Set membership	Basic Bloom filter
<ul style="list-style-type: none">• Set cardinality• Multi-set cardinality (union, intersection)• Similarity between sets	HyperLogLog
<ul style="list-style-type: none">• Point query• Top-K• Heavy hitters• Range queries• Histograms• Quantiles• Median• Inner product	Count-Min (CM)

CM Real-World Applications

- **Compressed sensing**
- **Networking**
- **Databases**
- **NLP**
- **Security**
- **Machine Learning**

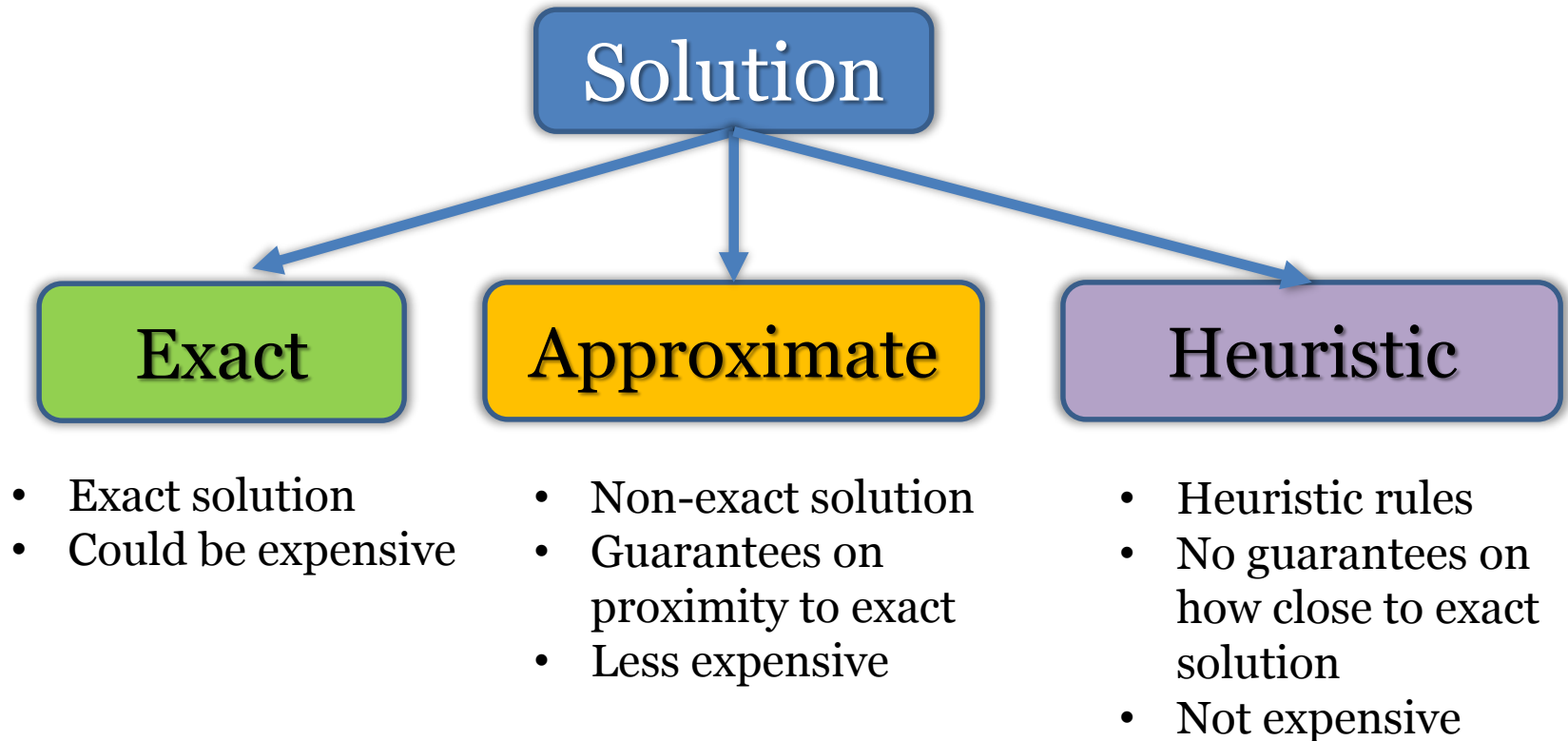


NOT the focus of this lecture!

General Theory

APPROXIMATION

Algorithm Types



CSC 505: Approximation Algorithm

Optimization Problem: A decision problem that asks for an **optimum** (**maximum** or **minimum**) value of its parameter k .

Examples:

- Minimum vertex cover
- Maximum clique
- Maximum independent set

$k = A(n)$ - the value of k for an algorithm A on an input of size n
 $k^* = Opt(n)$ - the optimum value of k for an input of size n

Definitions:

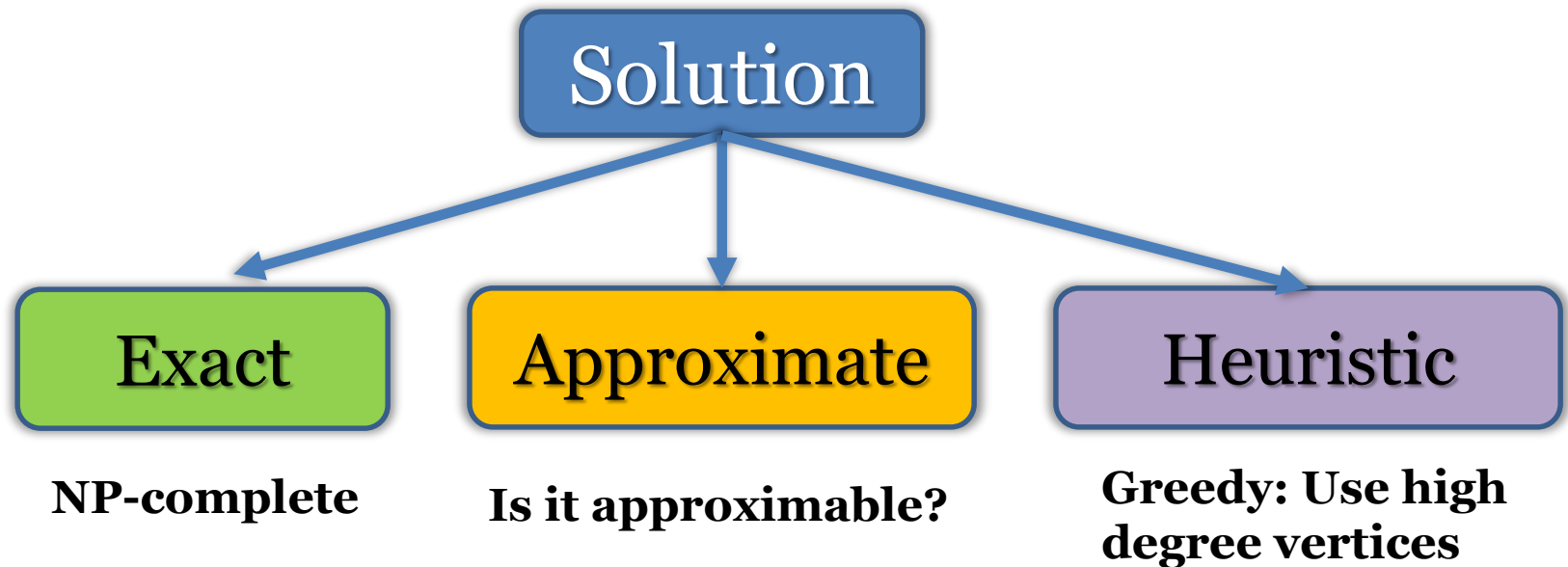
A problem has an **approximation ratio** of $\rho(n)$ for **any** input of size n :

$$\max \left(\frac{k}{k^*}, \frac{k^*}{k} \right) \leq \rho(n)$$

An algorithm $A(n)$ is called **$\rho(n)$ -approximation** algorithm.

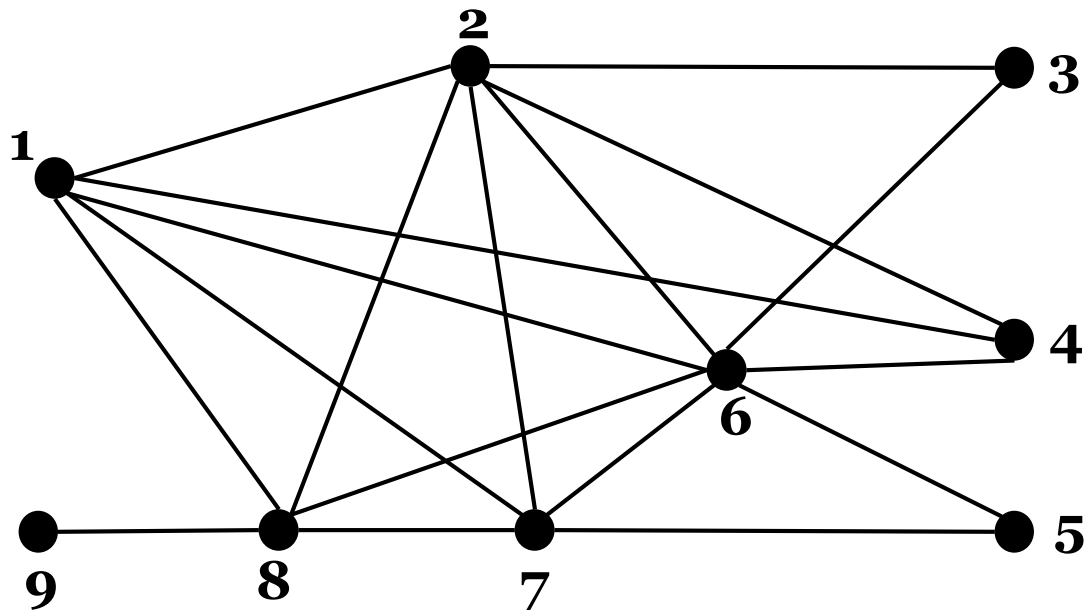
Ex: Minimum Vertex Cover in Graph

We want to find a **minimum** set of vertices (nodes) such that every edge in the graph is attached to at least one vertex in this minimum set.



What are the real-world applications for the VC problem?

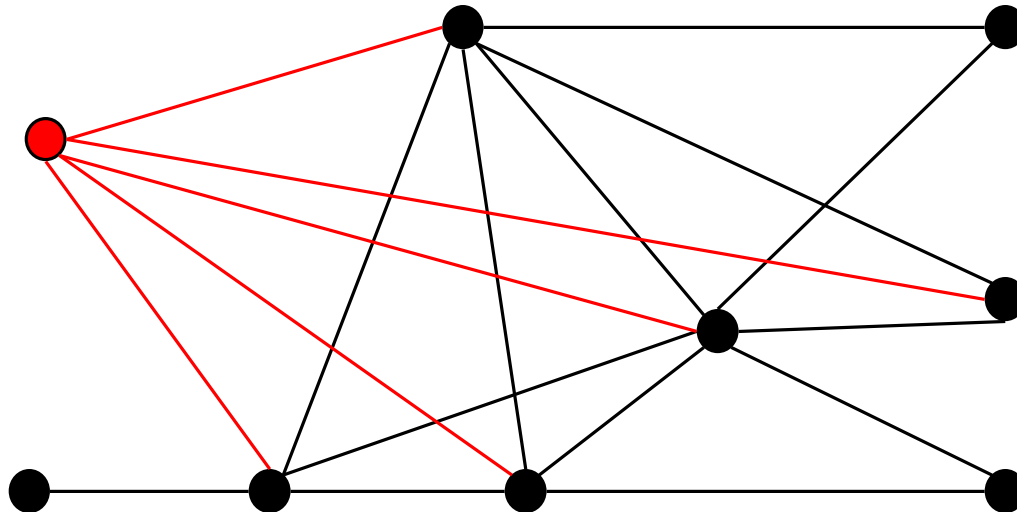
What is the size of the minimum VC in this graph?



COVER EDGES WITH VERTICES from VC

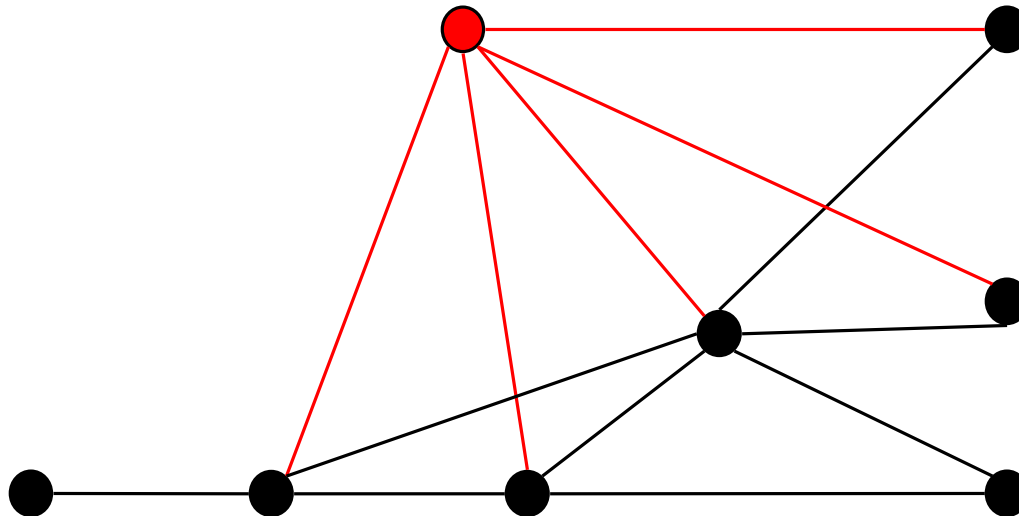
Upper Bound: High Degree Greedy Heuristic

$k = 1?$



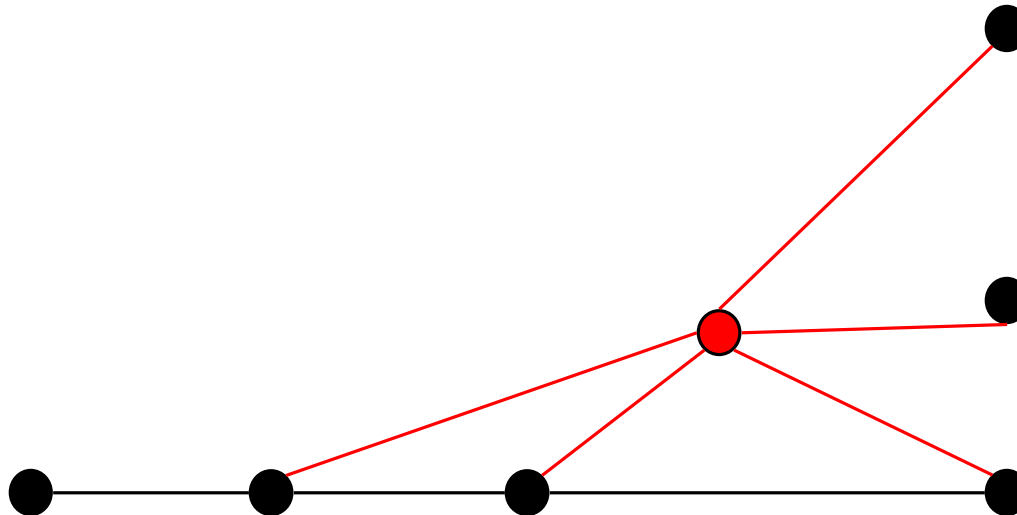
Upper Bound: High Degree Greedy Heuristic

$k = 2?$



Upper Bound: High Degree Greedy Heuristic

$k = 3?$



Upper Bound: High Degree Greedy Heuristic

$$k = 4?$$



Upper Bound: High Degree Greedy Heuristic

$$k = 5?$$



Polynomial Time Vertex Cover Approximation

Algorithm:

VertexCover($G = (V, E)$)

while $E \neq \emptyset$

 Select an arbitrary edge $(u, v) \in E$

 Add both u and v to the vertex cover, S

 Delete all edges from E that are incident on either u or v

$$|S| \leq 2k$$

Off by a multiplicative factor of 2 at most

Claim: If G has a vertex cover of size k then $k \geq \frac{1}{2}|S|$.

Proof:

Note that S is a vertex cover.

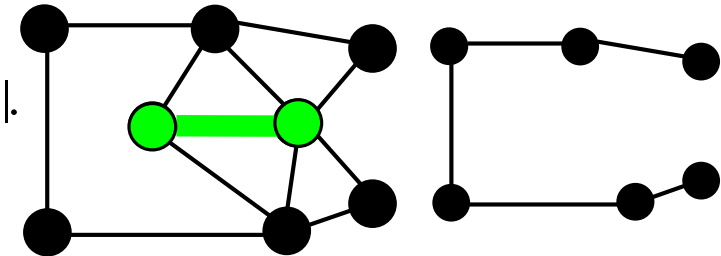
Consider the edges selected by the algorithm.

No two edges share a vertex.

Any cover of these edges must include at least one vertex per edge.

Therefore, minimum vertex cover size k must be at least half of $|S|$:

$$k \geq \frac{1}{2} |S|$$



General TSP is NOT Approximable (unless P=NP)

Theorem: If $P \neq NP$, then for any **constant** $\rho \geq 1$, there is no **polynomial-time ρ -approximation** algorithm for a general TSP problem.

An approximation scheme is a **polynomial time approximation** scheme if for **any fixed** $\varepsilon > 0$, the $(1 + \varepsilon)$ -approximation algorithm runs in time polynomial in the size n of its instance.

Hint: Show that HAM-CYCLE can be solved in polynomial time.

$$c(u, v) = \begin{cases} 1, & \text{if } (u, v) \in E \\ \rho |V| + 1, & \text{otherwise} \end{cases}$$

If the graph has Hamiltonian cycle, it corresponds to a TSP path of cost $|V|$. Any non-Hamiltonian TSP path has cost at least $(1 + \rho) * |V|$ because at least one of the edges will be of cost $\rho |V| + 1$, i.e., cost $\geq (|V| - 1) + \rho |V| + 1 = (1 + \rho) * |V| > |V|$. Thus, we have a poly-time algorithm to decide an NP-complete problem, HAM-CYCLE. Contradiction, if we assume that P is not equal to NP.

CSC 505: Approximation Summary

- **Problems: NP-hard**
 - **Some are approximable: Minimum Vertex Cover, MAX-CUT**
 - **Some are not: Travel Salesman Problem (TSP), Max CLIQUE**
- **Approximation: A polynomial function of finite problem size**
 - **Approximation constant: multiplicative factor**
- **Approximation algorithm (if it exists):**
 - **Deterministic**
 - **Always guarantees to find the solution that is optimal up to the approximation constant/function**

Data Streaming Algorithms

RANDOMIZED APPROXIMATION

Data Streaming Model

- **INPUT:**

- massively long input stream defined as a *sequence* of

$$S = \langle s_1, s_2, \dots, s_m \rangle,$$

- where each element is drawn from the universe $U = \{1, 2, \dots, n\}$, i.e. $s_i \in U$; m is the stream length and n is the universe size

- **GOAL:**

- Compute some functions $\phi_{approx}(S)$ that are estimates or approximations to their true value $\phi_{true}(S)$

- **Algorithmic REQUIREMENTS:**

- $Space = O(\log m + \log n)$
- No more than p passes ($p=1$ or small) over the data

Bad News....

- Many even basic functions can provably not be computed *exactly* using *sub-linear* space with any *deterministic* algorithm
- → we resort to **RANDOMIZED** streaming algorithms:
 - *Approximate* solution
 - *Sub-linear* space
 - *Non-deterministic*

Randomized Approximation Algorithm

- **Notation:**
 - Let $A(S)$ denote the output of a **randomized** streaming algorithm A on input stream S .
 - Let $\phi(S)$ be the function that A is supposed to compute.

- **Approximation:**

- The algorithm $A(S)$ (ϵ, δ) -approximates **multiplicatively** $\phi(S)$ if

$$\Pr \left[\left| \frac{A(S)}{\phi(S)} - 1 \right| > \epsilon \right] \leq \delta$$

- The algorithm (ϵ, δ) -approximates **additively** $\phi(S)$ if


$$\Pr[|A(S) - \phi(S)| > \epsilon] \leq \delta$$

- **GOAL:** To devise randomized streaming approximation algorithms with provable sub-linear space requirement



WWW.PHDCOMICS.COM

Cash Register vs. Turnstile Streaming Model

- **Cash Register Model:**  **our focus**
 - Stream elements only "arrive" and
 - $\phi(S)$ only gets incrementally updated
- **Turnstile Model:**
 - Stream elements may also "depart" the streaming multiset of items
 - and $\phi(S)$ can get incremented or decremented

Variation of the Basic Formulation

INPUT: $\mathcal{S} = \langle s_1, s_2, \dots, s_m \rangle$ over $U = [n] = \{1, 2, \dots, n\}$, i.e. $s_i \in U$;
 m is the stream length and n is the universe size
GOAL: Compute some functions $\phi_{approx}(\mathcal{S})$ that are estimates or approximations to their true value $\phi_{true}(\mathcal{S})$

NEW INPUT: $\vec{f} = \langle f_1, f_2, \dots, f_n \rangle$ where f_j is the frequency of j in the stream, i.e., the number of times the stream element with the value of j appears in the stream.

NEW GOAL: Compute some functions $\phi_{approx}(\vec{f})$ that are estimates or approximations to their true value $\phi_{true}(\vec{f})$ such that:

$$\Pr[|\phi_{approx}(\vec{f}) - \phi_{true}(\vec{f})| > \epsilon] \leq \delta$$

Query Problems

$S = \langle s_1, s_2, \dots, s_m \rangle$ over $U = [n] = \{1, 2, \dots, n\}$

$\vec{f} = \langle f_1, f_2, \dots, f_n \rangle$ where f_j is the frequency of j in the stream

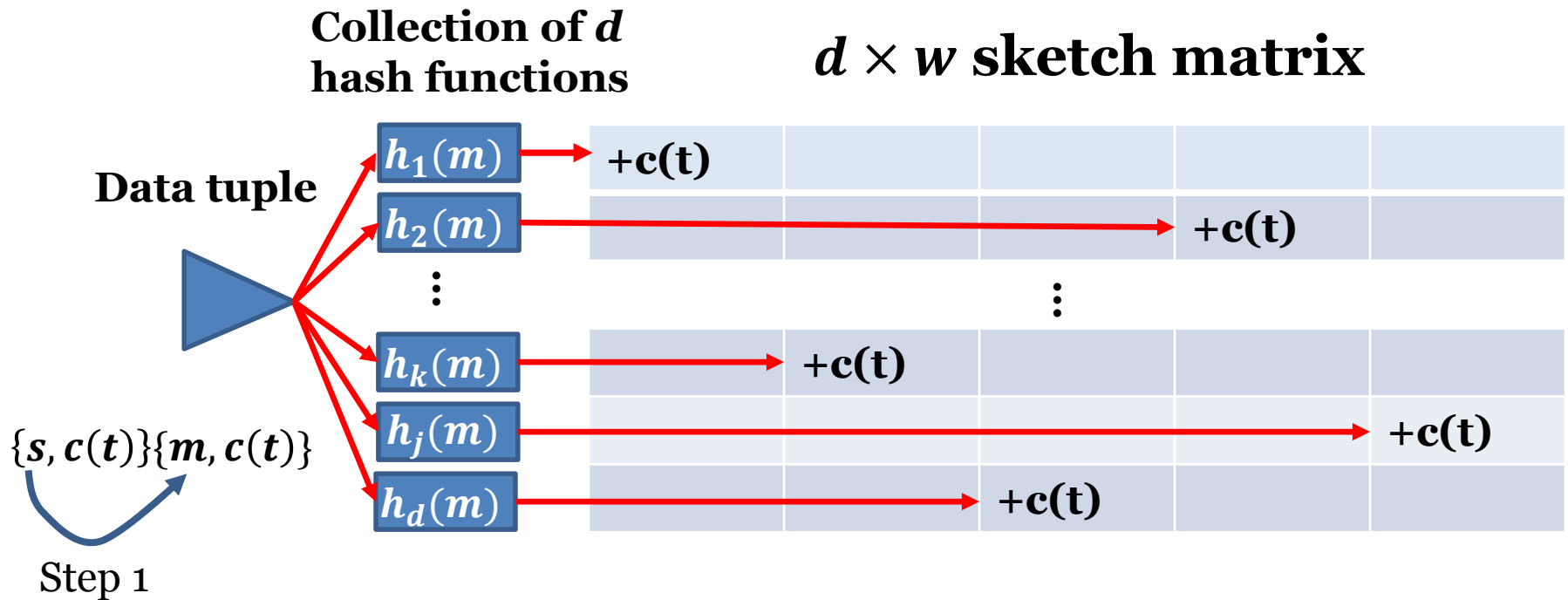
Point Query: For $j \in [n]$, estimate f_j

Range Query: For $i, j \in [n]$, estimate $f_i + f_{i+1} + \dots + f_j$

Quantile Query: For $\alpha \in [0, 1]$, find j with $f_1 + \dots + f_j \approx \alpha \times m$

Heavy Hitter Query: For $\alpha \in [0, 1]$, find all j with $f_j \geq \alpha \times m$

Step 1: Build Count-Min Sketch Matrix



Definition: Pairwise independent hash function

A hash function $h(x)$ mapping from range $[n]$ to range $[w]$ is said to be **pairwise independent**, if for all $x_1 \neq x_2$ in range $[n]$, and all y_1 and y_2 in range $[w]$, the following property holds:

$$\Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = \Pr[h(x_1) = y_1] \Pr[h(x_2) = y_2] = \frac{1}{w} \times \frac{1}{w} = \frac{1}{w^2}$$

Lemma: Probability of collision

Lemma: If a hash function $h(x)$ mapping from range $[n]$ to range $[w]$ is *pairwise independent*, then the following property holds:

$$\Pr[h(x) = h(y)] = \frac{1}{w}$$

Proof: By definition of *pairwise independence* and *marginalization*:

$$\begin{aligned}\Pr[h(x) = h(y)] &= \sum_{j=1}^w \Pr[h(x) = j \wedge h(y) = j] = \\ &= \sum_{j=1}^w \frac{1}{w^2} = w \times \frac{1}{w^2} = \frac{1}{w}\end{aligned}$$

Step 2: Point Query w/ Count-Min Sketch

Point Query: For $j \in [n]$, estimate f_j

1. A set H of d *pairwise independent hash functions*

$$h_1, h_2, \dots, h_d: [n] \rightarrow [w]$$

2. As we observe the stream, we maintain $d \times w$ counters, where

$c_{i,j}$ = the number of elements s in the stream with $h_i(s) = j$

3. For any $x \in [n]$, $c_{i,h_i(x)}$ is an over-estimate for f_x :

$$\text{true value} \longrightarrow f_x \leq \widetilde{f}_x = \min(c_{1,h_1(x)}, \dots, c_{d,h_d(x)}) \longleftarrow \text{estimated value}$$

(Note: A red arrow points from the tilde symbol to the estimated value label, and a blue arrow points from the true value label to the inequality.)

Question: How good is an estimate?

INPUT: $S = \langle s_1, s_2, \dots, s_m \rangle$

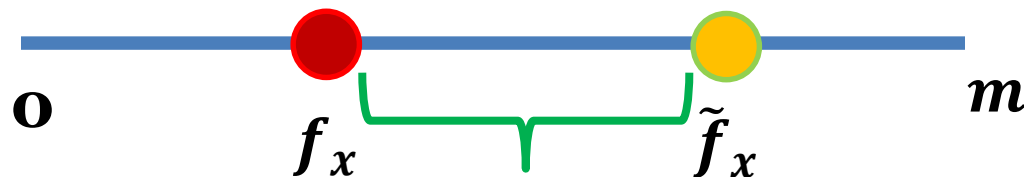
Definition: The algorithm (ϵ, δ) -approximates *additively* $\phi(\vec{f})$ if

$$\Pr[|\phi_{approx}(\vec{f}) - \phi_{true}(\vec{f})| > \epsilon] \leq \delta$$

Lemma 1: $\tilde{f}_x \geq f_x, \forall x \in [n]$

Lemma 2: $\tilde{f}_x \leq m, \forall x \in [n]$

Lemma 3: $\Pr[\tilde{f}_x - f_x \leq \epsilon \times m] \geq 1 - \delta$
if $w = 2/\epsilon$ and $d = \log_2 \delta^{-1}$



Proof of Lemma 3: One hash $f()$ ($d=1$)

1. Define random variables Z :

$$Z_x = \sum_{\forall y \in S: y \neq x: h(y) = h(x)} f_y$$

collision

2. Then the sketch matrix counter:

$$c_{h(x)} = \widetilde{f}_x = f_x + Z_x$$

3. Define collision indicator variable: $I_y = 1$ if $h(y) = h(x)$ and 0 otherwise

$$I_y = \begin{cases} 1, & h(y) = h(x) \\ 0, & \text{otherwise} \end{cases}$$

$$Z_x = \sum_{\forall y \in S: y \neq x} f_y \times I_y$$

Proof of Lemma 3: One hash $f()$ (Cont.)

4. Estimate the mean of Z_x :

$$E(Z_x) = \sum_{\forall y \in S: y \neq x} f_y \times E(I_y)$$

collision

5. But because h is a pairwise independent hash function, then:

$$E(I_y) = \Pr(h(y) = h(x)) = \frac{1}{w}$$

6. Then the estimated mean of

$$E(Z_x) = \sum_{\forall y \in S: y \neq x} f_y \times \frac{1}{w} = \frac{1}{w} \times \sum_{\forall y \in S: y \neq x} f_y \leq \frac{m}{w}$$

Note: $\sum_{\forall y \in S} f_y = m$

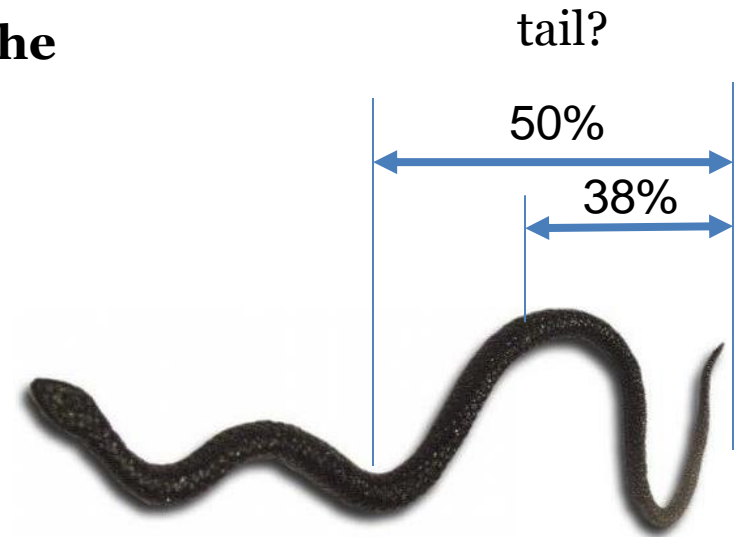
$$E(Z_x) \leq \frac{m}{w}$$

Markov Inequality

Suppose a random nonnegative variable Z such that: $\Pr\{Z \geq 0\} = 1$
What is the probability that the value falls in the tail of the distribution?

If the estimated mean $E(Z)$ is the expected value and a is the value where the “tail” starts then Markov Inequality gives you the estimate:

$$\Pr\{Z \geq a\} \leq E(Z)/a$$



$$\frac{1}{e} = \frac{1}{2.71} = 0.3679$$

To sum up...

$$\Pr\{Z \geq a\} \leq E(Z)/a$$

$$E(Z_x) \leq \frac{m}{w}$$

$$a = \epsilon \times m: \Pr\{Z_x \geq \epsilon \times m\} \leq E(Z_x)/(\epsilon \times m)$$

$$\Pr\{Z_x \geq \epsilon \times m\} \leq \frac{\frac{m}{w}}{\epsilon \times m} = \frac{1}{w \times \epsilon} = 1/2 \text{ for } w = \frac{2}{\epsilon}$$

$$\Pr\{Z_x \geq \epsilon \times m\} \leq 1/2 \text{ for } w = \frac{2}{\epsilon}$$

$$c_{h(x)} = \widetilde{f}_x = f_x + Z_x$$

General Case for Lemma 3: d hash $f()$'s

1. Define random variables Z_1, \dots, Z_d :

$$Z_{i,x} = \sum_{\forall y \in S: y \neq x: h_i(x) = h_i(y)} f_y, \forall i \in [d]$$

2. Then the sketch matrix counter: $C_{i,h_i(x)} = f_x + Z_{i,x}$

3. Since each $Z_{i,x}$ is independent:

$$\Pr\{Z_{i,x} \geq \epsilon \times m, \forall 1 \leq i \leq d\} \leq \left(\frac{1}{2}\right)^d = \delta \text{ for } w = \frac{2}{\epsilon}$$

$$\Pr\{Z_{i,x} \geq \epsilon \times m, \forall 1 \leq i \leq d\} \leq \delta$$

for $w = \frac{2}{\epsilon}$ and $d = \log_2 \delta^{-1}$

4. Therefore, with probability $1 - \delta$ there exists a value j such that:

$$Z_{j,x} \leq \epsilon \times m$$

Therefore, the point query estimate...

$$\begin{aligned} f_x &\leq \tilde{f}_x = \min(c_{1,h_1(x)}, \dots, c_{d,h_d(x)}) \\ &= \min(f_x + Z_{1,x}, f_x + Z_{2,x}, \dots, f_x + Z_{j,x}, \dots, f_x + Z_{d,x}) \\ &\leq f_x + \epsilon \times m \end{aligned}$$

Theorem: We can find an estimate \tilde{f}_x for f_x that satisfies,

$$f_x \leq \tilde{f}_x \leq f_x + \epsilon \times m$$

*with probability $1 - \delta$ while only
using $O(w \times d) = O(\epsilon^{-1} \times \log \delta^{-1})$ memory*

Reminder:

$$\begin{aligned} &\Pr\{Z_{i,x} \geq \epsilon \times m, \forall 1 \leq i \leq d\} \leq \delta \\ &\text{for } w = \frac{2}{\epsilon} \text{ and } d = \log_2 \delta^{-1} \end{aligned}$$

Query Problems

$S = \langle s_1, s_2, \dots, s_m \rangle$ over $U = [n] = \{1, 2, \dots, n\}$

$\vec{f} = \langle f_1, f_2, \dots, f_n \rangle$ where f_j is the frequency of j in the stream

Point Query: For $j \in [n]$, estimate f_j

Range Query: For $i, j \in [n]$, estimate $f_i + f_{i+1} + \dots + f_j$

Quantile Query: For $\alpha \in [0, 1]$, find j with $f_1 + \dots + f_j \approx \alpha \times m$

Heavy Hitter Query: For $\alpha \in [0, 1]$, find all j with $f_j \geq \alpha \times m$

Range Queries

- **Data has some sort of natural ordering**
- **Data has an empirical distribution (or histogram) , e.g.:**
 - prices paid for a particular advertising unit on a website
 - number of seconds spent on a given page before taking an action
- **Range query questions:**
 - What fraction of users spent less than 10 seconds on this page?
 - What is the median sale price of this advertising unit?

Range Query: Approach #1

Range Query: For $i, j \in [n]$, estimate $f_i + f_{i+1} + \cdots + f_j$

Naive Approach: By adding multiple point query estimates

$$sum_e = \tilde{f}_i + \widetilde{f_{i+1}} + \cdots + \tilde{f}_j$$

But the error accumulates with the increase of the range!

Why?: 1. Each point query estimate will contribute up to $\epsilon \times m$:

$$f_x \leq \tilde{f}_x \leq f_x + \epsilon \times m$$

2. The maximum size of the range is n : because $i, j \in [n]$

3. The estimate for the range query:

$$sum_{true} \leq sum_e \leq sum_{true} + n \times \epsilon \times m$$

Can we get a much tighter bound?

Naive Approach:

$$sum_{true} \leq sum_e \leq sum_{true} + n \times \epsilon \times m$$

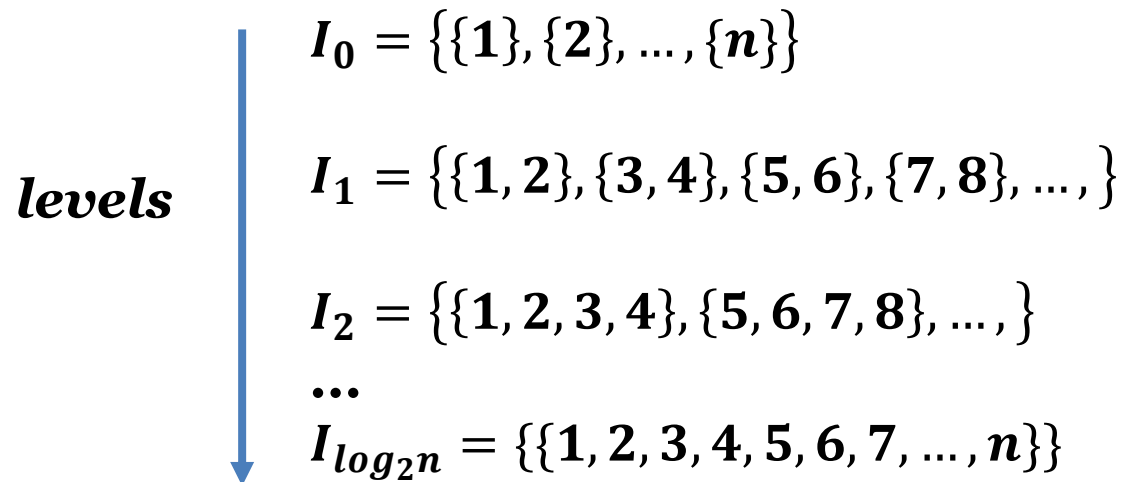
Dyadic Interval Approach:

$$sum_{true} \leq sum_e \leq sum_{true} + 2(\log_2 n) \times \epsilon \times m$$

Dyadic Intervals/Ranges

Define $\log_2 n$ partitions of $[n]$ where:

- (a) each partition I has a power-of-two length $l = 2^r$
- (b) its *start index* is $1(\bmod l)$
- (c) each point in $[n]$ is a member of exactly one dyadic range of each length: $2^0, 2^1, \dots, 2^{\log_2 n}$



The two parameters define the start and end points of the interval:

$$[2^{\text{level}} \times \text{start}, 2^{\text{level}} \times (\text{start} + 1) - 1]$$

Programmatically...

Dyadic Interval with two parameters (*level*, *start*):

$$[2^{\textit{level}} \times \textit{start}, 2^{\textit{level}} \times (\textit{start} + 1) - 1]$$

```
public int min() { return (1 << level)* start; }  
public int max() { return (1 << level)*( start + 1) - 1; }
```

Lemma: On the union of dyadic ranges

Lemma: Any range within $[n] = \{0, \dots, n\}$ has a unique, non-overlapping, complete covering consisting of at most $2 \times \log_2 n$ dyadic ranges.

Proof: (Exam question)

Corollary: Any range within $[n]$ has an estimate for its true value:

$$sum_{true} \leq sum_e \leq sum_{true} + 2(\log_2 n) \times \epsilon \times m$$

Example: The range $[12; 27]$ for $n = 64$ is covered by the union of dyadic ranges:
 $[12; 27] = [12; 12] \cup [13; 16] \cup [17; 24] \cup [25; 26] \cup [27; 27]$



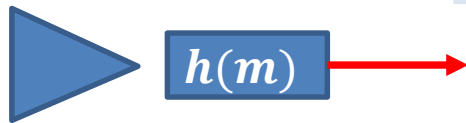
Range Queries with Count-Min

- **Goal:** Use Count-Min to reduce the number of point queries required to compute the range query to reduce the error
- Use **multiple** Count-Min sketches that store all data at different **resolutions/dyadic ranges**:
 - **1st sketch**: the basic sketch that stores the point queries
 - **2nd sketch**: halves the resolution by combining the first and second elements into one counter, and so on.
 - **3rd sketch**: combines the first two elements of the 2nd sketch
 - The resolution is halved until the last sketch contains only two elements

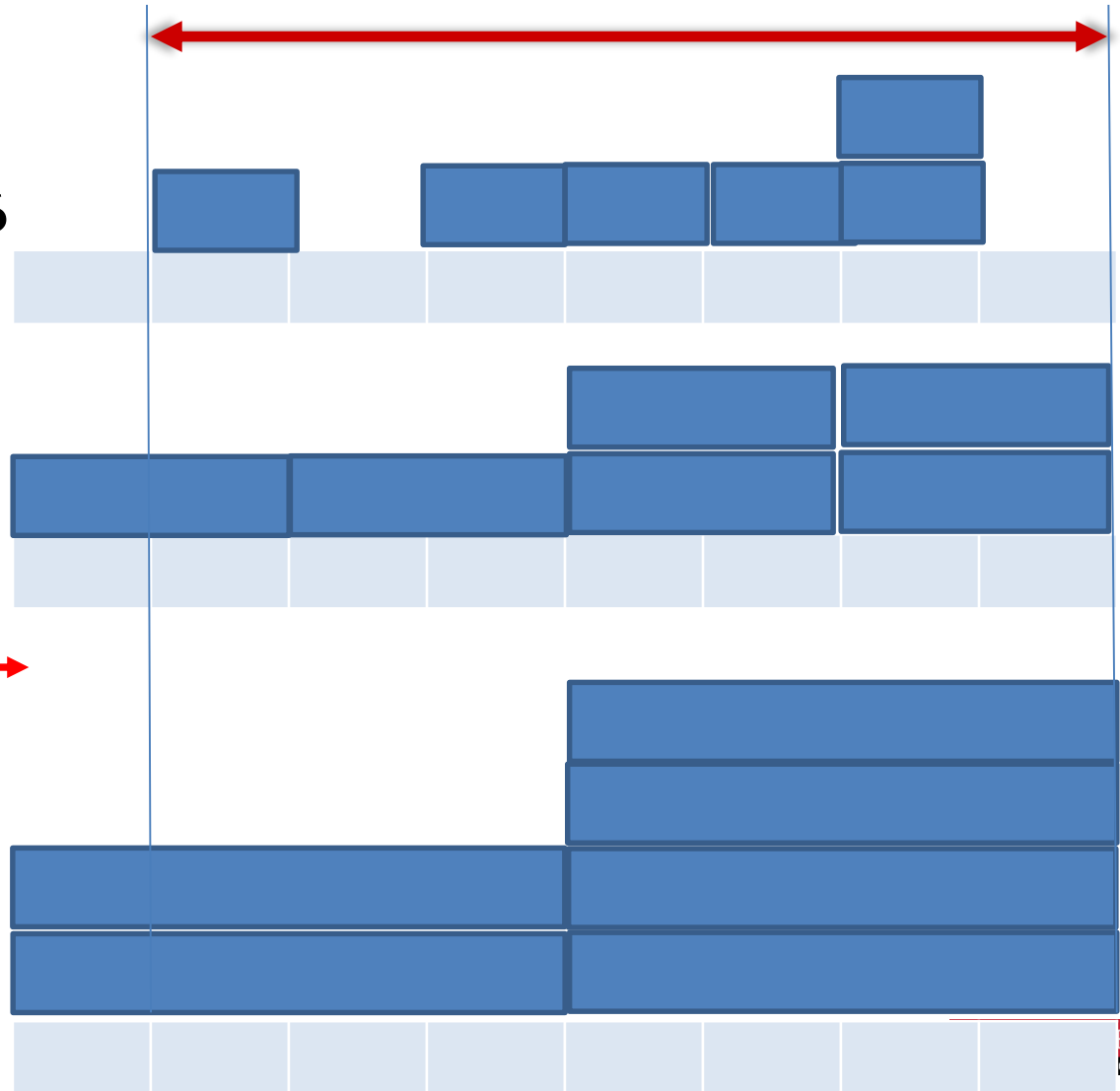
Range query with Count-Min illustration

Naïve: $1+0+1+1+1+2=6$

Data tuple



Dyadic: $1+1+4=6$



Finding covering dyadic intervals

These two parameters define the start and end points of the interval as:

$$[2^{level} \times start, 2^{level} \times (start + 1) - 1]$$

```
public int min() { return (1 << level)* start; }  
public int max() { return (1 << level)*( start + 1) - 1; }
```

- Subintervals computed recursively by finding the largest dyadic interval that fits *within* the current interval.
- This interval is added to the final output and potentially generates two further intervals, the interval to the left of the range and the interval to the right of the range.
- These ranges are recursively divided in the same way until all points have been included in one range:

Dyadic intervals

Halving resolution allows any range query to be divided into at most $2 \times \log_2 n$ intervals, where n is the total size of the domain.

e.g. if the domain is the space of 32-bit integers, then at most 64 subintervals are needed to compute any range query:

$O(1)$ compared to $O(n)$ compute time of the naïve implementation

```
public class DyadicInterval {
    public int level = 0;
    public int start = 0;
    public DyadicInterval() { }
    public DyadicInterval( int level, int start ) {
        this.level = level;
        this.start = start;
    }
}
```

```

public static List <DyadicInterval> createIntervals(
int a, int b) {
    ArrayList < DyadicInterval > output = new ArrayList <
DyadicInterval >();
    Stack < Integer > left = new Stack < Integer >();
    Stack < Integer > right = new Stack < Integer >();
    left.push(Math.min(a, b));
    right.push(Math.max(a, b) + 1);
    while( left.size() > 0) {int l=left.pop();int r=right.pop();
        for( int k = 32; k >= 0; k--) {
            long J = (1L << k);
            long L = (int)( J*Math.ceil((double) l/( double) J));
            long R = L + J - 1;
            if( R < r) {
                //Segment fits in the range
                output.add( new DyadicInterval(( int) k,( int)( L/ J)));
                if( L > l) {left.push( l);  right.push(( int) L); }
                if( R+1< r) {left.push(( int)( R + 1)); right.push( r);}
                break;
            }
        }
    }
}

```

source: Chapter 10, Real-time analytics book github

Implementing Count-Min Ranges

```
public class CountMinRange {
    ArrayList<CountMinSketch<Integer>> sketches =
        new ArrayList<CountMinSketch<Integer>>();
    long N = 0;
    int max = Integer.MAX_VALUE;
    int min = Integer.MIN_VALUE;
    public CountMinRange(int width, int[] seeds) throws IOException{
        for (int i=0; i<32; i++)
            sketches.add(new CountMinSketch<Integer>(width, seeds));
    }
}
```

Update step

Update step adds the count to the appropriate start parameter of each of the different levels

```
public void increment(int value, long n) {  
    if(value < min) min = value;  
    if(value > max) max = value;  
    for (int i=0; i<32; i++) {  
        sketches.get(i).increment(value/(1<<i),n);  
    }  
    N+=n;  
}
```

Range calculations

```
public long range (int a, int b) {  
    long count = 0;  
    for(DyadicInterval d: DyadicInterval.createIntervals(a,b) {  
        count+=sketches.get(d.level).get(d.start);  
    }  
    return count;  
}
```

Query Problems

$S = \langle s_1, s_2, \dots, s_m \rangle$ over $U = [n] = \{1, 2, \dots, n\}$

$\vec{f} = \langle f_1, f_2, \dots, f_n \rangle$ where f_j is the frequency of j in the stream

Point Query: For $j \in [n]$, estimate f_j

Range Query: For $i, j \in [n]$, estimate $f_i + f_{i+1} + \dots + f_j$

Quantile Query: For $\alpha \in [0, 1]$, find j with $f_1 + \dots + f_j \approx \alpha \times m$

Heavy Hitter Query: For $\alpha \in [0, 1]$, find all j with $f_j \geq \alpha \times m$

Quantiles computation via Range Query

- The computation of quantiles, such as the median, is accomplished via a *binary search* of the space.
- Output: the closest value to the appropriate quantile

Example Quantile Query:

Query the number of people in the top 10% of income

Reduce to Range Query:

$$\text{range}(i, j) = (\frac{9}{10} \times n, n) \in [n]$$

Quantiles computation: Java Code

- The computation of quantiles, such as the median, is accomplished via a *binary search* of the space.
- Output: the closest value to the appropriate quantile

```
public int quantile(double q) {  
    if(q == 0.0) return min;  
    if(q == 1.0) return max;  
    int lo = min;  
    int hi = max;  
    while (lo <= hi) {  
        int mid = lo + (hi - lo)/2;  
        double val = frequency(min, mid);  
        if (val < q) hi = mid - 1;  
        else if (val > q) lo = mid + 1;  
        else return mid;  
    }  
    return lo; // Return the nearest value  
}
```

Query Problems

$S = \langle s_1, s_2, \dots, s_m \rangle$ over $U = [n] = \{1, 2, \dots, n\}$

$\vec{f} = \langle f_1, f_2, \dots, f_n \rangle$ where f_j is the frequency of j in the stream

Point Query: For $j \in [n]$, estimate f_j

Range Query: For $i, j \in [n]$, estimate $f_i + f_{i+1} + \dots + f_j$

Quantile Query: For $\alpha \in [0, 1]$, find j with $f_1 + \dots + f_j \approx \alpha \times m$

Heavy Hitter Query: For $\alpha \in [0, 1]$, find all j with $f_j \geq \alpha \times m$

Top-K and “Heavy Hitters”

A common Count-Min application is to maintain lists of frequent items.

- **Top-K:** a list of the k-most common items in the data stream
- **Heavy Hitters:** list of items with frequencies higher than some predetermined value $\alpha \times m$

Basic implementation uses a Count-Min sketch to store the *frequencies* and a *heap structure* to hold the top values.

Basic implementation

Comparator for **PriorityQueue**

```
public class SketchComparator < T extends Serializable >
    implements Comparator < T > {
    CountMinSketch < T > sketch;
    public SketchComparator( CountMinSketch < T > sketch) {
        this.sketch = sketch;
    }

    public int compare( T o1, T o2) {
        return (int)(sketch.get( o1) - sketch.get( o2));
    }
}
```

Top-K list

To implement a Top-K list, all you need is

1. to increment an incoming element
2. add it to the *priority queue*.
3. If the queue is larger than k (it can be at most $k + 1$ elements), remove the smallest element to return it to size k :

Top-K implementation

```
public class TopList<E extends Serializable> {
    CountMinSketch<E> sketch;
    PriorityQueue<E> heap;
    int k = Integer.MAX_VALUE;

    public TopList(int k, CountMinSketch<E> sketch) {
        this.sketch = sketch;
        this.k = k;
        this.heap = new PriorityQueue<E>(k + 1, new
            SketchComparator<E>(sketch));
    }

    public void add(E element) {
        sketch.increment(element);
        heap.add(element);
        while (heap.size() > k) heap.remove();
    }
}
```

Heavy Hitter

- Heavy Hitter implementation is identical to top-K except that elements are removed from the queue if they do not meet the frequency requirements.
- This requires the maintenance of a counter of all elements seen thus far.

Heavy Hitter implementation

```
public class TopList<E extends Serializable> {
    CountMinSketch<E> sketch;
    PriorityQueue<E> heap;
    double f = 1.0; int N = 0;
    public HeavyHitters(double f, CountMinSketch <E> sketch) {
        this.sketch = sketch;
        this.heap = new PriorityQueue<E>(11,
            new SketchComparator < E >( sketch));
        this.f = f;
    }
    public void add( E element) {
        N++;
        sketch.increment( element);
        heap.add( element);
        while( heap.size() > 0 && f<=(double)sketch.get(
            heap.peek()))/( double) N)
            heap.remove();
    }
}
```

Acknowledgements

All Java codes are from the github for the book, "Real-time analytics", chapter 10