

期末考试

- 闭卷考试
- 8个题目（简答题、算法题）
- 复习材料：乐学上所有PPT
- 重点内容：MG Sketch, 3种Min-Hash sketch、Min-hash采样及其应用, Morris Counter, 蓄水池采样, Bloom filter, 指数分布等

1. 第一章

1.1 Key-Value pairs

Data element $e \in D$ has key and value ($e.key, e.value$)

Data access

Distributed data/parallel computation



- GPUs, CPUs, VMs, Servers, wide area, devices
- Distributed data sources
 - Distribute for faster/scalable computation

Challenges: Limit data movement, Support updates to D

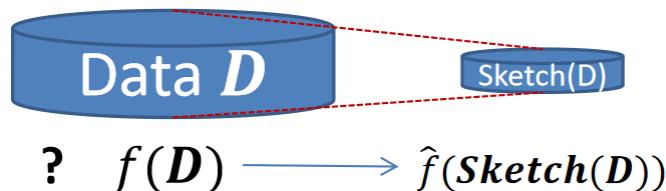
Data streams



- Data read in one (or few) sequential passes
- Can not be revisited (IP traffic)
 - **I/O efficiency** (sequential access is cheaper than random access)

Challenges: "State" must be much smaller than data size, Support updates to D

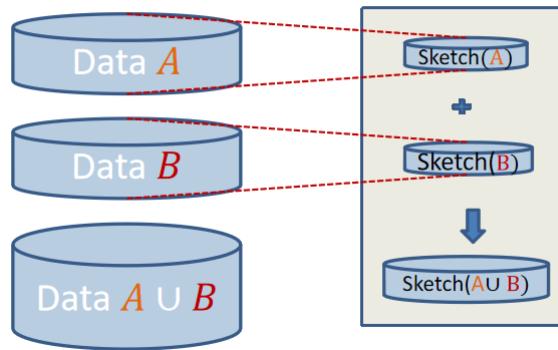
1.2 Summary Structures (Sketches)



这张幻灯片讲解了**概要结构 (Sketches) **的概念，用于处理大型数据集时的简化和估算。数据集 D 是原始数据，而 $f(D)$ 表示我们希望计算的数据统计或属性。然而，直接处理 D 可能会因为其体积过大而在存储、传输和查询时效率低下，因此我们引入了 $\text{Sketch}(D)$ ，这是 D 的简化表示，作为其“代理”。通过对 $\text{Sketch}(D)$ 应用估计器 $\hat{f}()$ ，可以近似计算 $f(D)$ ，从而在不访问整个数据集的情况下获得我们需要的统计结果。常见的示例包括随机样本、投影和直方图。Sketch 方法的优点在于，它在支持多种查询的同时减少了数据处理的时间和资源消耗。

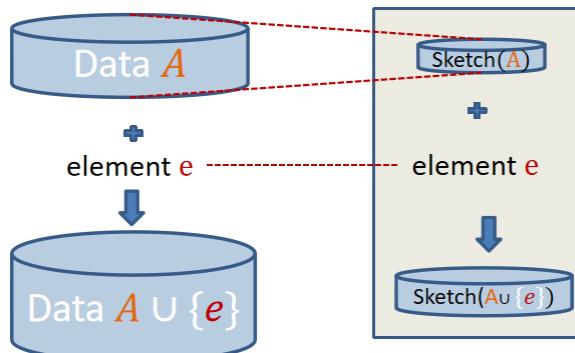
1.2.1 Composable sketches

▪ Sketch($A \cup B$) from Sketch(A) and Sketch(B)

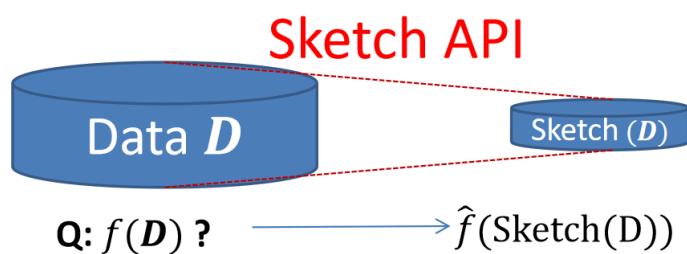


1.2.2 Streaming sketches

▪ Sketch($A \cup \{e\}$) from Sketch(A) and element $\{e\}$



1.2.3 Sketch API



- Initialization Sketch(\emptyset)
 - Estimator specification $\hat{f}(\text{Sketch}(D))$
 - Merge two sketches Sketch($A \cup B$) from Sketch(A) and Sketch(B)
 - Process an element $e = (e.key, e.val)$: Sketch($A \cup e$) from Sketch(A) and e
 - Delete e
- optional*
- Seek to optimize **sketch-size** vs. **estimate quality**

这张幻灯片介绍了 **Sketch API** 的工作流程，展示如何通过概略数据 (sketch) 进行数据估计。对于一个数据集 D ，我们可以用 $\text{Sketch}(D)$ 来创建数据的摘要，并通过估计器 $\hat{f}(\text{Sketch}(D))$ 来近似计算出我们需要的属性 $f(D)$ 。在 Sketch API 中，主要步骤包括初始化一个空的 Sketch、指定估计器、合并多个 Sketch、处理单个元素（如添加或删除），这些操作帮助管理和更新 Sketch。API 的目标是在 **sketch 大小和估计质量之间** 找到平衡，以确保有效的存储和准确的计算。

1.2.4 example

Easy sketches: min, max, sum, ...

Element values: 32, 112, 14, 9, 37, 83, 115, 2,

Exact, composable, Sketch is just a single register s :

Sum

- Initialize: $s \leftarrow 0$
- Process element e : $s \leftarrow s + e.\text{val}$
- Merge s, s' : $s \leftarrow s + s'$
- Delete element e : $s \leftarrow s - e.\text{val}$
- Query: return s

Max

- Initialize: $s \leftarrow 0$
- Process element e : $s \leftarrow \max(s, e.\text{val})$
- Merge s, s' : $s \leftarrow \max(s, s')$
- Query: return s

No delete support

max:不支持删除操作，因为删除元素后无法准确地维护最大值。

1.3 Frequent Keys-MG

MG算法 (Misra-Gries Algorithm) 是一种用于 **频繁元素检测** 的流处理算法，特别适用于大规模数据流中识别**出现次数较高的元素**。该算法在有限的空间内运行，能够在不保存所有数据的情况下，近似找出数据流中的频繁元素。

Find the keys that occur very often.

Zipf law: Frequency of i^{th} heaviest key $\propto i^{-s}$
Say top 10% keys in 90% of elements

幻灯片中还提到了 **zipf 定律**，即第 i 个频繁键的出现频率 $\propto i^{-s}$ ，这意味着少数键占据大部分频率。例如，前 10% 的键可能在 90% 的元素中出现。**zipf 定律**在许多场景中适用，例如城市人口分布、社交网络中的用户行为等。

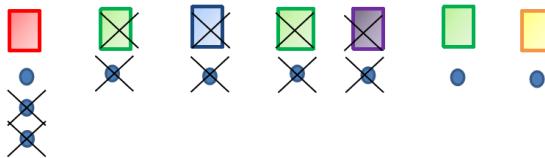
在这张幻灯片中， s 是 **zipf 定律**中的指数参数，它控制了频率分布的陡峭程度。



Sketch size parameter k : Use (at most) k counters indexed by keys. Initially, no counters

Processing an element with key x

- If we already have a counter for x , increment it
- Else, If there is no counter, but there are fewer than k counters, create a counter for x initialized to 1.
- Else, decrease all counters by 1. Remove 0 counters.



Query: #occurrences of x ?

- If we have a counter for x , return its value
- Else, return 0.

Clearly an under-estimate.

What can we say precisely?

$$\begin{aligned} n &= 6 \text{ #distinct} \\ k &= 3 \text{ #structure size} \\ m &= 11 \text{ #element} \end{aligned}$$

Lemma: Estimate is smaller than true count by at most $\frac{m-m'}{k+1}$

m' : Sum of counters in structure; m : #elements in stream; k : structure size

We charge each “missed count” to a “decrease” step:

- If key in structure, any decrease in count is due to “decrease” step.
- Element processed and not counted results in decrease step.

We bound the number of “decrease” steps

Each decrease step removes k “counts” from structure, together with input element, it results in $k+1$ “uncounted” elements.

$$\Rightarrow \text{Number of decrement steps} \leq \frac{m-m'}{k+1}$$

18

我们限制“减少”步骤的次数

为了保证误差的上限，算法限制了“减少”步骤的次数。具体说明如下：

- 每次减少步骤会从数据结构中移除 k 个“计数”，同时还包含一个输入元素，这样总共会产生 $k+1$ 个未计入的元素。
- 因此，每次减少步骤实际上减少了 $k+1$ 个未被计数的元素。由此可以推出减少步骤的总次数应满足

$$\leq \frac{m - m'}{k + 1}$$

估计准确性：某个元素的频率估计值最多比真实频率低

$$\frac{m - m'}{k + 1}$$

其中：

m' : Sum of counters in structure; m : #elements in stream; k : structure size

这个误差界限表示估计的频率值最多比真实频率低

$$\frac{m - m'}{k + 1}$$

频率条件: 对于频率远大于

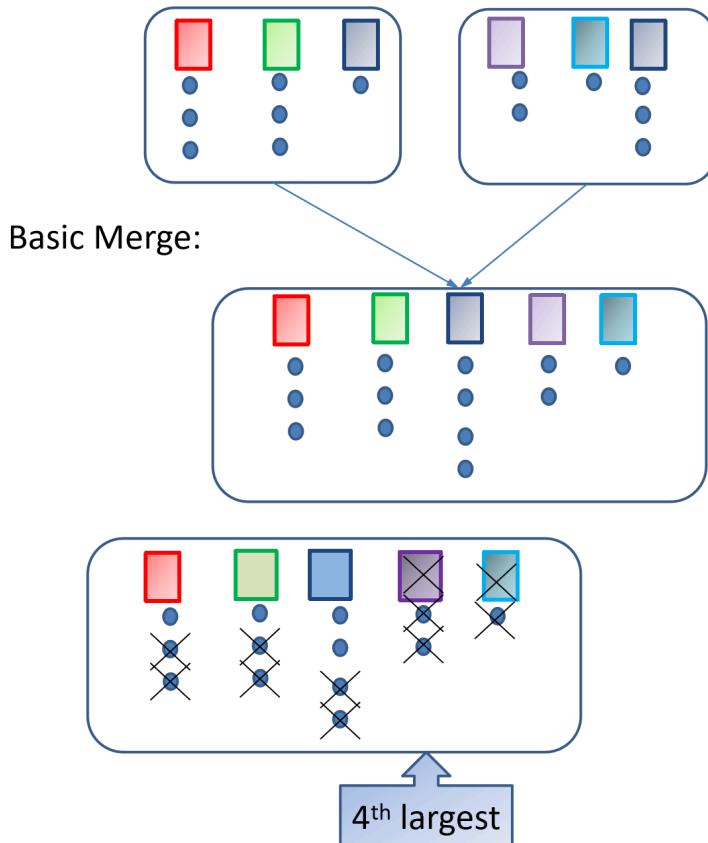
$$\frac{m - m'}{k + 1}$$

的元素 x , MG sketch 能够提供良好的频率估计。这意味着它对在数据流中出现频率较高的元素有较好的效果。

误差界限的比例关系: 误差界限与 k 成反比, 即随着 k 的增加, 误差会减小。这反映了 sketch 大小和估计质量之间的权衡。

Zipf 定律: MG sketch 有效的原因在于典型的频率分布通常符合 Zipf 定律, 即数据流中只有少数元素会非常流行, 大部分元素的出现频率较低。

Merging two Misra Gries Sketches



Reduce since there are more than $k = 3$ counters :

- Take the $(k + 1)^{\text{th}} = 4^{\text{th}}$ largest counter
- Subtract its value (2) from all other counters
- Delete non-positive counters

1. **声明 1**: 最终合并后的 sketch 至多有 k 个计数器。

- **证明**: 通过从所有计数中减去第 $(k + 1)$ 大的计数值, 这样做可以保证只有最大的 k 个计数可以保持为正数。因此, 最终合并后的 sketch 不会超过 k 个计数器。

2. **声明 2**: 对于每个键 (key), 合并后的 sketch 计数最多比真实计数少 $\frac{m-m'}{k+1}$ 。

- 这意味着, 在合并后的 sketch 中, 每个元素的估计值的误差界限最多为 $\frac{m-m'}{k+1}$, 其中 m 是总元素计数, m' 是其他元素计数的总和。

声明2的证明:

1. **声明**: $m'_1 + m'_2 - m' \geq R(k + 1)$

- 其中, m'_1 和 m'_2 是两个部分中的计数, m' 是合并后的计数结果, R 是在 `reduce` 步骤中丢弃的第 $(k + 1)$ 大计数的值。

2. **证明**: 在 `reduce` 步骤中, 计数值 R 从每一个前 $k + 1$ 个最大计数器中都被减去, 因此被减去的总量至少是 $R(k + 1)$ 。此外, 在较小的计数器中可能会减去更多的值。

1. **声明**: 对于每个键 (key), 合并后的 sketch 计数最多比真实计数少 $\frac{m-m'}{k+1}$ 。

2. **证明思路**: 键 x 的“计数”在合并过程中可能会丢失于 `part1`、`part2` 或 `reduce` 组件中。因此, 我们需要把这些部分的计数误差上界加起来。

3. **Part 1**:

- 总元素数: m_1
- 在结构中的计数: m'_1
- 丢失的计数上限: $\leq \frac{m_1-m'_1}{k+1}$

4. **Part 2**:

- 总元素数: m_2
- 在结构中的计数: m'_2
- 丢失的计数上限: $\leq \frac{m_2-m'_2}{k+1}$

5. **Reduce 组件**: 在合并过程中, 每个键的“reduce”丢失的计数上限是 R , 即第 $(k + 1)$ 大的计数值。

“Count missed” of one key is at most

$$\frac{m_1-m'_1}{k+1} + \frac{m_2-m'_2}{k+1} + R \leq \frac{1}{k+1}(m_1 + m_2 - m') = \frac{m-m'}{k+1}$$

1.4 Set membership-Bloom Filters

1. 结构

Bloom Filter 的结构通常由一个 **位数组** 和多个 **哈希函数** 组成:

- 位数组**: 一个长度为 m 的二进制数组, 所有元素初始时都设置为 0。
- 哈希函数**: 一组独立的哈希函数, 假设有 k 个哈希函数。每个哈希函数将输入元素映射到数组中的一个位置。

2. 插入元素

当我们把元素 x 插入到 Bloom Filter 中时，按照以下步骤操作：

1. 使用 **多个哈希函数** $h_1(x), h_2(x), \dots, h_k(x)$ 计算元素 x 的哈希值。每个哈希函数会输出一个整数，通常是一个数组索引。
2. 对于每个哈希函数的结果，将对应的位数组位置设置为 1。

举个例子，假设我们有 3 个哈希函数 h_1, h_2, h_3 ，并且位数组的长度是 10：

- 假设元素 x 通过哈希函数 h_1, h_2, h_3 映射到的位置分别为索引 2, 5, 7。
- 那么我们将位数组的第 2、5、7 个位置标记为 1。

3. 查询元素

查询时，我们需要判断某个元素 x 是否存在于 Bloom Filter 中。具体操作如下：

1. 使用 **相同的哈希函数** h_1, h_2, \dots, h_k 对查询元素 x 进行哈希，得到 k 个哈希值。
2. 对于每个哈希值 $h_i(x)$ ，检查位数组的对应位置是否为 1。
 - 如果所有位置都为 1，那么元素 **可能在集合中**。
 - 如果有任何一个位置为 0，那么元素 **肯定不在集合中**。

广泛应用：布隆过滤器在许多应用中都非常流行，尤其适合**快速判断元素是否属于一个集合**。

概率性数据结构：布隆过滤器是一种概率性的数据结构，允许一定的错误率。

减少存储需求：它将每个键的表示大小减少到少量的位（通常是每个键 8 位），大大节省了存储空间。

可能出现误报，但不会漏报：布隆过滤器可能会错误地报告一个不存在的元素为存在（误报），**但不会漏掉实际存在的元素**。

大小与误报率的权衡：布隆过滤器的大小和误报率之间存在权衡关系。增大过滤器大小可以降低误报率，但会消耗更多的存储空间。

可组合性：布隆过滤器具有可组合性，可以将多个布隆过滤器合并以检查集合的联合。

依赖独立的随机哈希函数：布隆过滤器的分析依赖于使用独立的随机哈希函数。在实际应用中，这些哈希函数的效果较好，但理论上仍存在一些问题。

Hash solution: Probability of a **false positive**

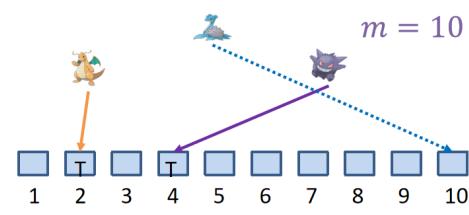
m : Structure size; n number of distinct keys inserted

$b = \frac{m}{n}$, number of bits we use in structure per distinct key in data

Probability ϵ of **false positive** for x :

Probability of $h(x)$ hitting an occupied cell:

$$\epsilon = \Pr_{h \sim H} [S[h(x)] = T] \approx \frac{n}{m} = \frac{1}{b}$$



Example:

$$\epsilon = 0.02 \Rightarrow b = 50$$

m : Structure size ; k : number of hash functions ; n number of distinct keys inserted

Probability of $h_i(x)$ NOT hitting a particular cell j :

$$\Pr_{h \sim H} [h_i(x) \neq j] = \left(1 - \frac{1}{m}\right)$$

Probability that cell j is F is that none of the nk "dart throws" hits cell j :

$$\Pr_{h \sim H} [S[j] = F] = \left(1 - \frac{1}{m}\right)^{kn}$$

A **false positive** occurs for x when all k cells $h_i(x)$ for $i = 1, \dots, k$ are T:

$$\varepsilon = \prod_{i=1, \dots, k} \left(1 - \Pr_{h \sim H} [h_i(x) = F]\right) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

* Assume $k \ll m$ so $h_i(x)$ for different $i = 1, \dots, k$ are very likely to be distinct

34

m : 布隆过滤器的总位数 (结构大小)。

k : 使用的哈希函数数量。

n : 插入的不同键的数量。

概率计算过程

1. 某个哈希函数不命中特定单元的概率:

- 对于某个元素 x 和特定单元 j , 哈希函数 $h_i(x)$ 不命中该单元 j 的概率是:

$$\Pr_{h \sim H} [h_i(x) \neq j] = \left(1 - \frac{1}{m}\right)$$

2. 单元 j 保持为 F (未被设置为 T) 的概率:

- 单元 j 保持为 F 的条件是, 所有 $n \times k$ 次“投掷”(即哈希映射) 都没有命中该单元。
- 这个概率为:

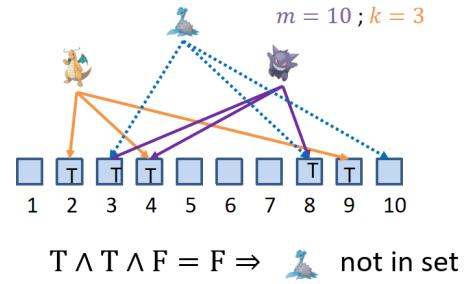
$$\Pr_{h \sim H} [S[j] = F] = \left(1 - \frac{1}{m}\right)^{kn}$$

3. 误报概率 (ε) :

- 当查询一个不存在的元素 x 时, 如果所有 k 个哈希函数对应的位置都为 T, 就会出现误报。
- 误报概率为:

$$\varepsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

- 这里假设 $k \ll m$, 即不同的哈希函数映射到的单元非常可能是独立的, 从而使得计算结果更准确。



m : Structure size ; k : number of hash functions ; n number of distinct keys inserted
 False positive probability:

$$\varepsilon \leq \left(1 - \left(1 - \frac{1}{m}\right)^{\frac{kn}{m}}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k = \left(1 - e^{-\frac{k}{b}}\right)^k$$

$$\lim_{i \rightarrow \infty} \left(1 - \frac{1}{i}\right)^i = \frac{1}{e}$$

$$\left(1 - \frac{1}{m}\right)^{\frac{kn}{m}} = \left(\left(1 - \frac{1}{m}\right)^{\frac{m}{m}}\right)^{\frac{kn}{m}} \approx \left(\frac{1}{e}\right)^{\frac{kn}{m}} = e^{-\frac{kn}{m}}$$

1.5 Simple Counting-Morris Counter

Morris Counter 是一种 概率性计数算法，相比传统的计数器（需要随着计数增长线性增长存储空间），它在空间效率上非常高效。Morris Counter 的核心思想是 在每一步概率性地决定是否增加计数器的值，从而用对数空间表示大数。

Morris Counter [Morris 1978]



Probabilistic stream counter: Maintain $\log n$ instead of n , use $\log \log n$ bits

| | |
|--|---|
| <ul style="list-style-type: none"> ▪ Initialize: $s = 0$ ▪ Increment: Increment s with probability 2^{-s} ▪ Query: Return $2^s - 1$ | $n = 10^9$, Exact: $\log_2 10^9 \approx 30$ bits $\log_2 \log_2 10^9 \approx 5$ bits |
|--|---|

| | |
|---------------------------------------|---|
| Stream: | 1, 1, 1, 1, 1, 1, 1, 1, 1, |
| Count n: | 1, 2, 3, 4, 5, 6, 7, 8, |
| $p = 2^{-x}$: | 1 $\frac{1}{2}$ $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{4}$ $\frac{1}{8}$ $\frac{1}{8}$ |
| Counter x: | 0 1 1 2 2 2 2 3 3 |
| Estimate \hat{n}: | 0 1 1 3 3 3 3 7 7 |

43

Morris Counter 的工作原理

1. 基本思路:

- Morris Counter 使用 $\log \log n$ 位来近似计数 n 而不是 $\log n$ 位。
- 通过概率性增量来实现更节省空间的计数。

2. 初始化:

- 将计数器 s 初始化为 0。

3. 增量:

- 每当遇到一个新元素时，以概率

$$2^{-s}$$

来增加 s 的值。这意味着 s 增加的频率随着计数增加而减少，从而实现对大数的近似。

4. 查询:

- 查询当前计数的估计值时，返回

$$2^s - 1$$

这是对实际计数的一个近似值。

Morris Counter 的无偏性 (Unbiasedness)，即其计数估计值如何在数学期望上接近真实值。

无偏性目标：证明每次增量操作后估计值的期望增量始终为 1。

分析：假设当前计数器值为 s ，增量概率为 2^{-s} 。

- 增量后，估计值的期望增量为：

$$2^{-s}((2^{s+1} - 1) - (2^s - 1)) + (1 - 2^{-s}) \times 0 = 2^{-s} \times 2^s = 1$$

- 这表明，每次增量操作的期望增量为 1，从而证明了 Morris Counter 的无偏性。

方差分析：

估计方差：设 X_n 是输入为 n 时的随机变量，对应 Morris Counter 的计数值。

- 估计的方差公式为：

$$\text{Var}[\hat{n}] = E[(\hat{n} + 1)^2] - E[\hat{n} + 1]^2$$

- 通过归纳法可以得出：

$$E[2^{2X_n}] = \frac{3}{2}n^2 + \frac{3}{2}n + 1$$

- 最终得到方差近似为 $\frac{1}{2}n^2$ ，且变异系数 CV 约为 $\frac{1}{\sqrt{2}}$ 。

通过平均减少方差：

平均法：使用 k 个独立的无偏估计 Z_i ，每个具有相同期望 μ 和方差 σ^2 。

- 将多个估计值取平均，期望不变，方差减少：

$$\text{Var}(\hat{n}') = \frac{\sigma^2}{k}$$

- 这使得变异系数 CV 缩减为 $\frac{\sigma}{\mu\sqrt{k}}$ ，即方差因 k 的增加而减少。

Morris Counter 通过独立计数器减少方差：

具体方法：使用 k 个独立的计数器 y_1, y_2, \dots, y_k ，计算每个计数器的估计值 $Z_i = 2^{y_i} - 1$ ，并将所有估计值取平均。

- 得到的 NRMSE 等于 CV，约为 $\frac{1}{\sqrt{2k}}$ 。

- **存储需求**：为了实现误差 ε ，需要 $k \log \log n = \frac{1}{2}\varepsilon^{-2} \log \log n$ 位。

通过基数改变减少方差：

基数改变方法：由 Morris 和 Flajolet 提出，通过改变计数基数从 2 变为 $1 + b$ ，使得估计为 $(1 + b)^s - 1$ 。

- 增量过程：以概率 $\Delta b^{-1}(1 + b)^{-s}$ 增加 s 。

- **效果**：当 b 趋近于 0 时，准确性提高，但计数器大小也会增加。

Weighted Morris Counter:

- **功能**: 加权 Morris Counter 可以处理带权重的值，并通过基参数 b 来调整精度和存储大小。
- **初始化**: 将计数 s 初始化为 0。
- **估计**: 返回 $(1 + b)^s - 1$ 作为估计值。
- **添加或合并**:
 - 对于值 V 或合并到另一个 Morris 计数器，增加 s 以使估计值增至不超过 V 。
 - 定义增量 $\Delta = V - Z$ ，并以概率 $\frac{\Delta}{b(1+b)^s}$ 增加 s 。
- **结果**: 方差满足 $\text{Var}[\hat{n}] \leq bn(n+1)$ ，并通过选择 $b = \varepsilon^2$ 可以控制变异系数 (CV)。

Morris Counter 适用于需要节省空间且对计数精度要求不高的场景。例如：

- **网络流量计数**: 在大规模流量监控中，只需要近似计数即可，不需要精确的计数。
- **用户行为分析**: 在一些简单的统计任务中，例如用户点击量、访问量等场景，可以用近似值估算。
- **物联网 (IoT) 数据处理**: 在资源受限的 IoT 设备中，空间效率尤为重要，Morris Counter 可以提供一种轻量化的计数方式。

2. 第二章

2.1 MinHash Sketch

The MinHash Sketch Variety

Sketch maintains k hash values s_1, s_2, \dots, s_k

k-mins sketch: Use k “independent” hash functions: h_1, h_2, \dots, h_k
Track the respective minimum s_1, s_2, \dots, s_k for each function.

Bottom-k sketch: One hash function: h
Track the k smallest values s_1, s_2, \dots, s_k

!!Same as our distinct sampler
Keep only hash values...

k-partition sketch: Use a single hash function: h'
Use the first $\log_2 k$ bits of $h'(x)$ to map x uniformly to one of k parts. Call the remaining bits $h(x)$.

For $i = 1, \dots, k$: Track the minimum hash value s_i of the elements in part i .

- All sketch types are the same for $k = 1$.
- Sketch depends only on set of distinct keys and h . Distribution only depends on n
- All sketches correspond to distinct sampling schemes

三种 MinHash Sketch 方法

1. k-mins sketch:

- 使用 k 个独立的哈希函数 h_1, h_2, \dots, h_k 。
- 对每个哈希函数，记录最小值 s_1, s_2, \dots, s_k ，即每个函数的最小哈希值。

2. Bottom-k sketch:

- 使用一个哈希函数 h 。
- 保留该哈希函数生成的 k 个最小值 s_1, s_2, \dots, s_k 。
- 与去重采样方法类似，仅保留最小的 k 个哈希值。

3. k-partition sketch:

- 使用单个哈希函数 h' 。
- 先使用 h' 的前 $\log_2 k$ 位将元素均匀地映射到 k 个部分之一。
- 对于每个部分 i ，记录该部分中的最小哈希值 s_i 。

注意事项

- 当 $k = 1$ 时，这些方法效果相同。
- 这些 sketch 类型仅依赖于不同键集合和哈希函数，且分布依赖于不同键的数量 n 。
- 这些方法代表了不同的采样策略，以有效进行相似性估计和数据去重。

2.1.1 k-mins MinHash Sketch

| k-mins $k = 3, n = 6$ | | | | | | |
|-----------------------|------|------|------|------|------|------|
| x | | | | | | |
| $h_1(x)$ | 0.45 | 0.35 | 0.74 | 0.21 | 0.14 | 0.92 |
| $h_2(x)$ | 0.19 | 0.51 | 0.07 | 0.70 | 0.55 | 0.20 |
| $h_3(x)$ | 0.10 | 0.71 | 0.93 | 0.50 | 0.89 | 0.18 |

$(s_1, s_2, s_3) = (0.14, 0.07, 0.10)$

- **k-mins Sketch 原理:**

- 使用 k 个独立的哈希函数 h_1, h_2, \dots, h_k 。
- 追踪每个哈希函数的最小值 s_1, s_2, \dots, s_k 。

- **处理流程:**

- 对于每个元素键 x :
 - 计算每个哈希函数的值 $h_i(x)$, 并更新最小值 $s_i = \min(s_i, h_i(x))$ 。
- **计算复杂度:** 每当 sketch 被修改或查询时, 计算复杂度为 $O(k)$, 因为我们需要计算 k 个最小哈希值。

2.1.2 k-partition MinHash Sketch

| $k\text{-partition } k = 3, n = 6$ | | | | | | |
|--|------|------|------|------|------|------|
| x | | | | | | |
| $i(x)$ | 2 | 3 | 1 | 1 | 2 | 3 |
| $h(x)$ | 0.19 | 0.51 | 0.07 | 0.70 | 0.55 | 0.20 |
| $(s_1, s_2, s_3) = (0.07, 0.19, 0.20)$ | | | | | | |

- **k-partition Sketch 原理:**

- 使用单个哈希函数 h' , 将键 x 均匀分配到 k 个部分之一。
- 哈希函数的前 $\log_2 k$ 位用于确定分区编号 i , 剩余位用于计算哈希值 h 。
- 对于每个部分 i , 我们记录该部分中的最小哈希值 s_i 。

- **处理流程:**

- 对于每个元素键 x :
 - 计算分区编号 i 和对应哈希值 h 。
 - 更新该分区的最小值 $s_i = \min(s_i, h)$ 。
- **计算复杂度:** 每次测试或更新的计算复杂度为 $O(1)$, 因此可以高效处理大数据。

2.1.3 Bottom- k Min-Hash Sketch

Bottom- k $k = 3, n = 6$

| | | | | | | |
|--------|------|------|------|------|------|------|
| x | | | | | | |
| $h(x)$ | 0.19 | 0.51 | 0.07 | 0.70 | 0.55 | 0.20 |

$$(s_1, s_2, s_3) = (0.07, 0.19, 0.20)$$



1. 处理流程:

- 使用一个哈希函数 h 生成哈希值。
- 对每个新元素 x , 如果 $h(x)$ 比当前存储的第 k 小的哈希值 s_k 还小, 且不在已有的哈希值列表中, 那么用 $h(x)$ 替换掉当前第 k 小的值并重新排序。

2. 计算复杂度:

- 使用有序列表或优先队列来维护 s_1, s_2, \dots, s_k 。
- 测试是否需要更新 (是否 $h(x) < s_k$) 的时间复杂度是 $O(\log k)$ 。
- 如果需要更新, 有序列表的插入或优先队列的更新操作也需要 $O(\log k)$ 。

3. 性能优化: 由于更新 sketch 的次数 (#modification) 远小于处理的不同元素的数量 (#distinct elements), 因此该方法可以高效处理大量数据。

2.1.4 Composability of MinHash Sketches

Merge s', s'' to obtain s

- **k -mins:** entry-wise min (per hash function) $s_i \leftarrow \min \{s'_i, s''_i\}$
- **k -partition:** entry-wise minimum (per part) $s_i \leftarrow \min \{s'_i, s''_i\}$
- **Bottom- k :** $\{s_1, \dots, s_k\} = \text{bottomk}\{s'_1, \dots, s'_k, s''_1, \dots, s''_k\}$

1. k -mins 方法

- **描述:** k -mins 表示“前 k 个最小哈希值”。
- **操作:** 对于每个哈希函数, 取 s' 和 s'' 中对应位置的最小值, 作为合并后的 sketch s 中的值。
 - 公式表示: $s_i \leftarrow \min\{s'_i, s''_i\}$

2. k-partition 方法

- **描述:** `k-partition` 将每个集合划分为 k 个部分，分别计算每个部分中的最小值。
- **操作:** 每个部分的哈希值之间，取最小值作为合并后的 sketch。
 - 公式表示: $s_i \leftarrow \min\{s'_i, s''_i\}$

3. Bottom-k 方法

- **描述:** `Bottom-k` 是指从两个集合的哈希值中，选择 k 个最小的值作为合并后的 MinHash sketch。
- **操作:** 取 s' 和 s'' 的所有哈希值，然后从中选出最小的 k 个值，形成合并后的 sketch。
 - 公式表示: $\{s_1, \dots, s_k\} = \text{bottomk}\{s'_1, \dots, s'_k, s''_1, \dots, s''_k\}$

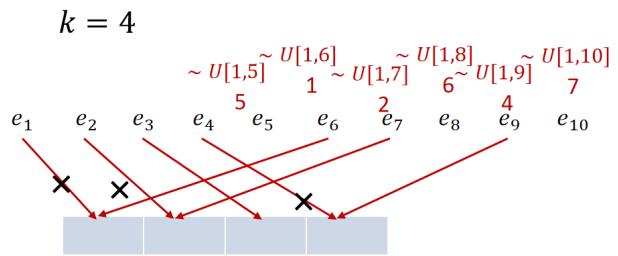
2.2 Reservoir Sampling

Streaming sketch Input: Stream of (unique) data elements: a_1, a_2, \dots

Initialize: Declare empty array $S[1], \dots, S[k]$ of elements; initialize counter $t \leftarrow 0$

Process element a :

- $t \leftarrow t + 1$ /* increment counter*/
- **If** $t \leq k$, $S[t] \leftarrow a$.
- **Else:**
 - Choose $i \sim U[1, \dots, t]$
 - **If** $i \leq k$, $S[i] \leftarrow a$



- **目标:** 在数据流环境下，随机采样 k 个数据项，使得每个数据项都有相等的概率被选中。
- **输入:** 一个数据流，包含一系列唯一的元素 a_1, a_2, \dots 。
- **输出:** 一个大小为 k 的样本，包含数据流中均匀随机选出的 k 个元素。

- 声明一个空数组 $s[1], \dots, s[k]$ ：用于存储采样的 k 个元素。
- 初始化计数器 t 为 0：用于跟踪数据流中的元素位置。

处理每个元素的步骤

对于每一个到来的元素 a ，按照以下步骤进行处理：

1. **增加计数器**：将 t 增加 1，表示当前处理的是数据流的第 t 个元素。
2. **判断计数器 t 是否小于等于 k** ：
 - 如果 $t \leq k$ ：将当前元素 a 添加到蓄水池中，存放在 $s[t]$ 的位置上。这是因为前 k 个元素一定会被放入样本中。
 - 如果 $t > k$ ：进入“替换”步骤。
3. **替换步骤**（当 $t > k$ 时）：
 - 随机选择一个索引 i 从 $[1, \dots, t]$ 中。
 - 如果选中的索引 i 在 $[1, \dots, k]$ 的范围内（即在蓄水池中），则用当前元素 a 替换 $s[i]$ 的元素。
 - 如果选中的索引 i 超出 $[1, \dots, k]$ 的范围，则跳过，不做任何替换。

假设我们设定的采样大小 $k = 4$ ，并且依次处理了 10 个元素 e_1, e_2, \dots, e_{10} 。以下是每个元素的处理情况：

1. 前四个元素 e_1, e_2, e_3, e_4 被直接放入蓄水池 $s[1]$ 到 $s[4]$ 的位置。
2. 第五个元素 e_5 到达时，生成一个随机索引（如 5 ），不在 $[1, 4]$ 范围内，因此不替换任何元素。
3. 第六个元素 e_6 到达时，生成的随机索引是 1 ，在 $[1, 4]$ 范围内，因此用 e_6 替换 $s[1]$ 中的元素。
4. 类似地，第七个元素 e_7 替换 $s[2]$ ，第八个元素 e_8 没有替换任何元素，依次进行。

通过这种方式，每个元素都有机会被选入蓄水池，并且保持在采样结果中的概率是相等的。

Reservoir sampling: Correctness

Stream of size n , sample size k .

Lemma: Each element a_i is included with probability $\min\{1, \frac{k}{n}\}$

Proof by induction: When $n \leq k$ all elements are included.

- Assume claim holds for some $n \geq k$. Consider processing a_{n+1}
- Element a_{n+1} has probability $\frac{k}{n+1}$ of entering sample
- Element a_i for $i \leq n$ had (by induction hypothesis) probability $\frac{k}{n}$ of being in the sample before processing a_{n+1} . It remains in sample with probability $\frac{n}{n+1} \Rightarrow$ it is included with probability $\frac{k}{n} \cdot \frac{n}{n+1} = \frac{k}{n+1}$

引理 (Lemma)

每个元素 a_i 被包含在样本中的概率为 $\min\{1, \frac{k}{n}\}$ 。

证明思路：归纳法

通过数学归纳法来证明这个结论。

1. **初始情况：**当 $n \leq k$ 时，所有元素都会被包含在样本中，因为数据流的大小不超过样本容量 k ，因此每个元素被选中的概率为 1。这与 $\min\{1, \frac{k}{n}\}$ 的结果一致。
2. **归纳假设：**假设对于某个 $n \geq k$ ，结论对前 n 个元素成立，即每个元素 a_i 被包含在样本中的概率为 $\frac{k}{n}$ 。
3. **处理第 $n + 1$ 个元素：**考虑处理第 $n + 1$ 个元素 a_{n+1} 。
 - **进入样本的概率：**第 $n + 1$ 个元素被选入样本的概率为 $\frac{k}{n+1}$ 。这是因为我们在处理 a_{n+1} 时，有 k 个样本槽位，并且我们随机选择其中一个来替换，概率为 $\frac{k}{n+1}$ 。
 - **前 n 个元素保留在样本中的概率：**对于前 n 个元素中的某个元素 a_i ，根据归纳假设，它被包含在样本中的概率为 $\frac{k}{n}$ 。当处理第 $n + 1$ 个元素时，该元素留在样本中的概率为 $\frac{n}{n+1}$ （即不被替换的概率）。

因此，元素 a_i 最终被包含在样本中的概率为：

$$\frac{k}{n} \cdot \frac{n}{n+1} = \frac{k}{n+1}$$

2.3 MinHash Sampling

2.3.1 方法

1. k-mins Sketch

- **采样策略:** `k-mins sketch` 使用了 k 个独立的哈希函数 h_1, h_2, \dots, h_k 。对于每个哈希函数，我们取该函数计算得到的最小哈希值作为特征，即每个 h_i 对应一个最小哈希值 s_i 。
- **对应的采样方式:** 这种方式相当于对每个哈希函数进行独立采样，每个函数都从数据集合中采样出它对应的最小哈希值。这种策略可以捕捉集合在不同哈希函数下的多个最小值，因此能更细致地反映集合的特征。

2. Bottom-k Sketch

- **采样策略:** `Bottom-k sketch` 使用单一哈希函数 h ，计算集合中每个元素的哈希值，并从所有哈希值中选择最小的 k 个值。这里的 "Bottom-k" 表示取哈希值最小的 k 个。
- **对应的采样方式:** 这种采样方式相当于从所有元素的哈希值中筛选出前 k 个最小值。由于只使用一个哈希函数，所以不会像 `k-mins` 那样分布在多个哈希函数上，但依然能代表集合中的整体特征。

3. k-partition Sketch

- **采样策略:** `k-partition sketch` 也是使用一个单一的哈希函数 h' ，但是将该哈希函数的结果的前 $\log_2 k$ 位作为分区，将集合划分为 k 个子集合。然后在每个子集合中选取最小的哈希值作为特征。
- **对应的采样方式:** 这种采样策略通过分区的方式，确保每个分区中选出一个最小哈希值，从而形成一种“分布式采样”或“分区采样”。这种方式确保了集合中不同部分的代表性，使得集合在不同区域上的特征都得以保留。

2.3.2 应用

1. k-mins Sketch

应用场景:

- **文本或网页去重:** `k-mins sketch` 可以通过比较不同文本或网页的 MinHash 签名来近似计算它们的相似度，从而判断是否为重复内容。
- **海量集合相似性检测:** 例如，在推荐系统中，可以用它来计算用户兴趣集合之间的相似性。
- **多样本比较:** `k-mins sketch` 可以对多个集合进行相似性分析，因为它提供了多哈希函数下的独立采样，这样的特征丰富度可以带来更高的精度。

优势:

- `k-mins sketch` 的独立哈希函数使得它的特征采样更丰富，能更准确地近似表示集合。
- 适合需要精度较高的相似性估计，因为多个哈希函数带来了较低的采样偏差。

劣势:

- 需要多个哈希函数，计算和存储成本较高。
- 当需要对大量集合进行实时比较时，计算资源消耗大。

适用场景总结: 适合高精度要求的场景，尤其在数据量不算特别大，或计算资源相对充裕的情况下。

2. Bottom-k Sketch

应用场景：

- **频繁项检测：**在大规模数据流分析中，`Bottom-k sketch` 可以用来找到出现频率较高的元素，因为它可以高效地提取前 k 个最小哈希值，从而代表频繁出现的元素。
- **近似查询：**用于近似集合查找和相似集合检索。在数据库或索引系统中，通过 `Bottom-k sketch` 可以快速找到与查询集合相似的集合。
- **空间有限的情况下集合比较：**`Bottom-k sketch` 在只使用单一哈希函数的情况下，可以通过前 k 个最小值来表示集合，节省空间和计算成本。

优势：

- 只需要一个哈希函数，计算和存储成本较低，适合高效、大规模的数据场景。
- 适合实时性要求高的场景，比如实时检测频繁出现的模式。

劣势：

- 相对 `k-mins sketch`，特征采样的多样性不足，可能影响精度。
- 对于非常高精度要求的场景可能不够理想，因为只使用一个哈希函数带来了一定的误差。

适用场景总结：适合在大规模数据流和实时性要求高的场景，尤其在内存或计算资源有限的情况下，如网络监控、异常检测、数据流分析等。

3. k-partition Sketch

应用场景：

- **分布式数据分析：**`k-partition sketch` 的分区特性适合分布式环境。每个分区只需要存储最小哈希值，适合在分布式系统中对大规模数据进行局部采样和全局分析。
- **局部相似性检测：**适用于需要保持分区间均衡性的场景。由于 `k-partition sketch` 将集合划分为多个部分并取每个部分的最小哈希值，可以用于识别集合中具有局部特征的相似性。
- **数据去重与查询优化：**在数据去重或数据查询中，`k-partition sketch` 可以用来分区存储数据，从而提升查询效率。特别适合需要将大集合分割为小部分处理的场景。

优势：

- 分区采样方式可以确保不同区域的代表性，适合在分布式或分区环境中处理大规模数据。
- 相比于 `k-mins sketch`，计算成本低于多哈希函数，适合资源有限的分布式系统。

劣势：

- 对分区的划分有一定要求，如果分区选择不当，可能导致局部偏差。
- 适合特定场景下的分区均衡采样，对于不需要分区的应用场景可能带来额外的计算复杂度。

适用场景总结：适合分布式环境和分区均衡性要求高的场景，例如在大规模分布式系统中对局部特征相似性进行分析。

k-mins sketch：适用于需要高精度的相似性检测，适合数据量相对中等且计算资源充足的场景。

Bottom-k sketch：适合**大规模、实时性要求高的场景**，具有更好的空间效率和计算效率。

k-partition sketch: 适用于分布式数据或需要分区处理的场景，通过分区采样在分布式环境下获得更均衡的样本。

3. 题目

3.1 指数分布

已知指数分布的概率密度函数为 $f(x)=ne^{-nx}$, $x \geq 0$, $f(x)=0$, $x<0$ 。请证明指数分布式的无记忆性。也即证明: $\forall t, y \geq 0$, $\Pr[X - y > t | X > y] = \Pr[X > t]$ 。

$$\Pr(X>t) = \int_t^\infty ne^{-nx} dx = -e^{-nx} \Big|_t^\infty = e^{-nt}$$
$$\Pr(X-y>t | X>y) = \frac{\Pr(X>y+t)}{\Pr(X>y)} = \frac{-e^{n(y+t)}}{-e^{ny}} = e^{-nt}$$

3.2 可合并的Sketch算法

请简要回答为什么需要设计可合并的 Sketch 算法？可合并的 Sketch 算法主要是用于什么场景？

可合并的 Sketch 算法是为了解决大规模数据流处理和分布式系统中的近似查询问题而设计的。这些算法能够对数据流进行压缩和摘要，以便在有限的内存和有限的通信带宽条件下处理大量的数据。

可合并的 Sketch 算法主要用于以下场景：

数据流处理: 在数据流处理系统中，数据以高速率持续到达，无法全部存储在内存中。可合并的 Sketch 算法通过在有限的内存中维护摘要信息，例如频率估计、矩估计等，能够对数据流进行实时查询和分析。

分布式系统: 在分布式系统中，数据通常分布在多个节点上，而节点之间的通信成本较高。可合并的 Sketch 算法允许在分布式环境下对数据进行分布式计算和聚合，从而减少数据传输量和通信开销。

网络监测和流量分析: 可合并的 Sketch 算法可以用于网络监测和流量分析，例如统计网络中不同类型的流量、识别网络中的异常行为或研究网络拓扑结构等。

3.3 MG算法

给定数据流 $D=(1,2,5,1,4,2,3,3,2,4,5,2)$, 假设 $k=3$, 请详细描述 Misra-Gries 算法在该数据流上的运行步骤。

```
T=0: MG{[]}
T=1: 输入1, MG{1:1}
T=2: 输入2, MG{1:1, 2:1}
T=3: 输入5, MG{1:1, 2:1, 5:1}
T=4: 输入1, MG{1:2, 2:1, 5:1}
T=5: 输入4, MG{1:1}
T=6: 输入2, MG{1:1, 2:1}
T=7: 输入3, MG{1:1, 2:1, 3:1}
T=8: 输入3, MG{1:1, 2:1, 3:2}
T=9: 输入2, MG{1:1, 2:2, 3:2}
T=10: 输入4, MG{2:1, 3:1}
T=11: 输入5, MG{2:1, 3:1, 5:1}
T=12: 输入2, MG{2:2, 3:1, 5:1}
```

3.4 Morris counter

请解释 Morris 计数算法的基本原理？它为什么能够做到只用 $O(\log n)$ 的空间来对 n 个数据进行计数？

Morris计数器的原理简单来说是记录一个计数器 s , 每次以

$$2^{-s}$$

的概率给 $s+1$, 最终返回的估计结果是

$$2^s - 1$$

。对于最终的估计量来说，我们所记录的 s 是估计量的log结果；而当我将 s 这个结果存储到内存中时，我还需要再进行一步log得出我最终所需的位数，所以最后的空间只需要 $\log n$ 。

3.5 Minhash

MinHash 算法有哪三种实现方式？它们各自有哪些优缺点？

Minhash算法的三种实现方式分别是：k-mins sketch、Bottom-k sketch、k-partition sketch。

K-mins算法是使用 k 个hash函数，每个哈希函数都对序列求一遍哈希，取出每个哈希函数求出的最小值代表该序列的最小哈希，优点是结果可信度高，缺点是浪费空间；

Bottom-k是只用一个哈希函数，对序列求哈希，取出前 k 个最小的代表该序列的最小哈希，优点是操作简单，节省空间；结果精度较低，容易受哈希冲突影响。

k-partition综合上述两种方法，先分成 k 个部分然后再用一个函数做哈希，取出每个部分中最小的哈希值，优点是结果可信度比较好，也比较节省空间，缺点是操作繁琐，大批量数据很麻烦。

关于复杂度（当一个元素到来时的时间开销）：

k-mins是 $O(k)$ ，无论sketch是否更新。

k-partition是 $O(1)$, test or update

bottom-k最低是 $O(\log k)$, to test if an update is needed; 对于to update的情况，取决于实现方式，如果是sorted list，则 $O(k)$ ，如果是优先队列，则 $O(\log k)$ 。

test的意思就是单纯地比个大小。update是如果比完大小需要更换。对于k-partition，更换只涉及一个元素；对于bottom-k, 更换完还涉及 k 个bottom的重新排序。

此外，关于modified次数（所有数据到来的情况下时间开销），三者虽然都近似为 $O(k \ln n)$ ，但具体而言，k-mins是 $k \ln n$ ，k-partition是 $k \ln(n/k)$ ，而bottom-k的小于 $k \ln n$ 小于 $k \ln n$ 。

3.6 K-mins Minhash

给定一个包含 n 个不同元素的集合 A ，请证明在采用 k-mins MinHash 算法来构造集合 A 的 k-mins sketch 的过程中，sketch 发生更新的期望次数为 $O(k \ln n)$ 。

假设我们已经有一个包含 k 个最小哈希值的 sketch，并且我们正在插入第 $i+1$ 个元素的哈希值。考虑新哈希值

$$h_{i+1}$$

会更新现有的 k 个最小值的概率。

- 如果

$$h_{i+1}$$

小于当前的第 j 个最小哈希值，那么它将替换第 j 个最小哈希值。

- 假设当前 k 个最小哈希值是

$$h_1, h_2, \dots, h_k$$

, 这些哈希值的排序是从小到大的。如果

$$h_{i+1}$$

是从新的元素的哈希值中抽取的, 那么

$$h_{i+1}$$

小于当前最小哈希值的概率是

$$\frac{k}{i+1}$$

, 因为在所有前 i 个哈希值中, 选择 k 个最小哈希值的期望概率是

$$\frac{k}{i+1}$$

(每个新的哈希值有 k 个位置可能被它替换)。

因此, 每个新的哈希值更新 k 个最小值的期望次数是

$$\frac{k}{i+1}$$

假设我们有 n 个元素, 总共有 n 次插入操作, 每次插入操作的期望更新次数为

$$\frac{k}{i+1}$$

因此, 总的期望更新次数为:

$$\mathbb{E}[\text{总更新次数}] = \sum_{i=1}^n \frac{k}{i}$$

`\mathbb{E}` 用于表示期望值符号。

`\text{总更新次数}` 用于在期望符号中显示文本“总更新次数”。

`\sum_{i=1}^n \frac{k}{i}` 表示从 $i=1$ 到 n 的求和, 每一项为 k / i

这个求和式是一个调和级数, 已知调和级数的上界为 $\ln n$, 即:

$$\because \sum_{i=1}^n \frac{1}{i} = O(\ln n)$$

$$\therefore \mathbb{E}[\text{总更新次数}] = O(k \ln n)$$

3.7 k-mins Minhash

给定集合 $A=\{a, b, c, d, e, h\}$, $B=\{a, c, e, f, g, m, n\}$ 。假设采用 k-mins MinHash 算法来处理集合 A 与 B。令 $k=4$, 得到集合 A 和集合 B 的 k-mins sketch 分别为 $S(A)=(0.22, 0.11, 0.14, 0.22)$, $S(B)=(0.18, 0.24, 0.14, 0.35)$

1. 请计算 A 和 B 的 Jaccard 相似性。
2. 请根据 $S(A)$ 和 $S(B)$ 来计算集合 A 与 B 合并后的 k-mins sketch, 即 $S(A \cup B)$ 。
3. 请基于 $S(A \cup B)$ 估计集合 A 和集合 B 的 Jaccard 相似性。

4. 在该问题中，基于 k-mins MinHash 算法的估计方差为多少？

1. 计算集合 A 和集合 B 的 Jaccard 相似性

Jaccard 相似性定义为两个集合的交集大小除以它们的并集大小：

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

根据题目给定的信息，集合 $A = \{a, b, c, d, e, h\}$ 和集合 $B = \{a, c, e, f, g, m, n\}$ 。

首先计算 $A \cap B$ 和 $A \cup B$ ：

- $A \cap B = \{a, c, e\}$, 即集合 A 和 B 的交集包含元素 a, c, e 。
- $A \cup B = \{a, b, c, d, e, h, f, g, m, n\}$, 即集合 A 和 B 的并集包含 10 个元素： $a, b, c, d, e, h, f, g, m, n$ 。

因此，Jaccard 相似性为：

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{3}{10} = 0.3$$

2. 计算集合 A 和集合 B 合并后的 k-mins sketch $S(A \cup B)$

集合 $S(A)$ 和 $S(B)$ 分别是集合 A 和集合 B 的 k-mins sketch。给定的 $k = 4$ ，我们有：

$$S(A) = (0.22, 0.11, 0.14, 0.22)$$

$$S(B) = (0.18, 0.24, 0.14, 0.35)$$

要计算 $S(A \cup B)$ ，我们首先计算 $S(A)$ 和 $S(B)$ 中每一位置的最小值，即将 $S(A)$ 和 $S(B)$ 中每个位置上的哈希值取最小值。具体地， $S(A \cup B)$ 的每个元素是 $S(A)$ 和 $S(B)$ 相同位置的哈希值的最小值：

$$S(A \cup B) = (\min(0.22, 0.18), \min(0.11, 0.24), \min(0.14, 0.14), \min(0.22, 0.35))$$

$$S(A \cup B) = (0.18, 0.11, 0.14, 0.22)$$

因此，集合 $A \cup B$ 的 k-mins sketch 是：

$$S(A \cup B) = (0.18, 0.11, 0.14, 0.22)$$

3. 基于 $S(A \cup B)$ 估计集合 A 和集合 B 的 Jaccard 相似性

根据 k -mins MinHash 算法的性质，集合 A 和集合 B 的 Jaccard 相似性估计值是它们的 k -mins sketch 中相同位置的最小值的相等比例。也就是说，计算 $S(A)$ 和 $S(B)$ 中每个位置相等的比例，作为 A 和 B 的 Jaccard 相似性估计值。

我们计算 $S(A)$ 和 $S(B)$ 中相同位置的哈希值是否相等：

- 第一个位置：0.22 和 0.18 不相等。
- 第二个位置：0.11 和 0.24 不相等。
- 第三个位置：0.14 和 0.14 相等。
- 第四个位置：0.22 和 0.35 不相等。

所以， $S(A)$ 和 $S(B)$ 相等的比例是 $1/4$ ，即 0.25。

因此，基于 $S(A \cup B)$ 估计的 Jaccard 相似性为：

$$\text{Jaccard Estimation}(A, B) = \frac{1}{4} = 0.25$$

4. 估计方差

MinHash 算法的方差是由以下公式给出的：

$$\text{Var}(\hat{J}) = \frac{1 - \text{Jaccard}(A, B)}{k}$$

其中， \hat{J} 是估计的 Jaccard 相似性， k 是使用的哈希函数的个数。根据之前的计算：

- 真实的 Jaccard 相似性是 $\text{Jaccard}(A, B) = 0.3$
- $k = 4$

因此，方差为：

$$\text{Var}(\hat{J}) = \frac{1 - 0.3}{4} = \frac{0.7}{4} = 0.175$$

3.8 k-mins Minhash/图

给定一个有向图 $G=(V, E)$ ，以及一个节点 v 。在图 G 中可以到达节点 v 的集合定义为 $\text{Reach-1}(v)=\{u \in V | u \text{ 在图 } G \text{ 中可达 } v\}$ 。对于集合 $\text{Reach-1}(v)$ ，我们可以采用 k -mins MinHash 算法来生成该集合的一个 k -mins sketch，记为 $S(v)$ 。

1. 请设计一个算法计算图中所有节点的 k -mins sketch，即对于任意的 v 计算所有的 $S(v)$ 。

在图论中，**Reach-1集合** (Reach-1 Set) 是指能够通过路径到达某个特定节点的所有节点的集合。对于给定的一个节点 v 和图 $G = (V, E)$ ，Reach-1集合 $\text{Reach-1}(v)$ 是图中所有能够到达节点 v 的节点的集合。

整体算法: 对于图 G 中的每个节点 v , 我们执行以下操作:

- 反向遍历图来计算 $\text{Reach-1}(v)$ 。
- 使用 k -mins MinHash 算法计算该集合的 k -mins sketch $S(v)$ 。

2. 请描述如何用 $S(u)$ 和 $S(v)$ 来计算 $|\text{Reach-1}(v) \cup \text{Reach-1}(u)|$?

Jaccard Similarity

Similarity measure of two sets

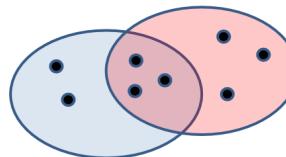
Features N_1 of document 1

Ratio of size of intersection
to size of union:

Features N_2 of document

2

$$J(N_1, N_2) = \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|}$$



$$J = \frac{3}{8} = 0.375$$

1. **比较 MinHash sketch:** 我们可以通过比较 $S(u)$ 和 $S(v)$ 中相同位置的哈希值, 来估算 $\text{Reach-1}(v)$ 和 $\text{Reach-1}(u)$ 的交集比例。
2. **交集比例:** 计算 $S(u)$ 和 $S(v)$ 中相同位置哈希值相等的比例, 记为 p_{equal} , 即:

$$p_{\text{equal}} = \frac{\text{Number of positions where } S(u) \text{ and } S(v) \text{ are equal}}{k}$$

3. **估算 Jaccard 相似性:** 通过以下公式估算 Jaccard 相似性:

$$\hat{J}(u, v) = p_{\text{equal}}$$

4. **计算并集的大小:** 利用 Jaccard 相似性和集合的大小关系, 可以估算 $|\text{Reach-1}(v) \cup \text{Reach-1}(u)|$:

$$|\text{Reach-1}(v) \cup \text{Reach-1}(u)| = \frac{|\text{Reach-1}(v)| \cdot |\text{Reach-1}(u)|}{\hat{J}(u, v)}$$

其中, $\hat{J}(u, v)$ 是通过 k -mins sketch 计算得到的 Jaccard 相似性。

3.9 bloom-filter

给定一个集合 D , 假设你已经构造了一个长度 $m=2b$ 比特的 bloom filter。你的同学也想用你构造的这个 bloom filter 来处理“元素 x 是否在 D 中”的查询, 但是你同学只有 b 比特的空间, 请问他该如何实现他的目标? 注意: 假设我们不允许他重新针对集合 D 构造一个长度为 b 的 bloom filter, 而是要求在你构造的长度为 $2b$ 的 bloom filter 的基础上去实现。

通过将原始长度为 $2b_2b_2b$ 的 Bloom Filter 划分为两部分，其中 **前 bbb 比特** 用于查询，**后 bbb 比特** 保留不使用，我们能够在同学只有 bbb 比特的情况下实现查询元素是否在集合 DDD 中的目标。这个方法的关键在于 **保持哈希函数一致性**，确保查询时能准确地映射到 Bloom Filter 的前 bbb 个比特，进而进行有效的查询。