

# 程序设计实习（实验班-2024春）

## 哈希方法

授课教师：姜少峰

助教：冯施源 吴天意

Email: [shaofeng.jiang@pku.edu.cn](mailto:shaofeng.jiang@pku.edu.cn)

**Hash: 均匀映射**

# 哈希函数 (hash functions)

- 哈希函数  $h : [n] \rightarrow [m]$  通常将某个较大的域  $[n]$  映射到某个较小的域  $[m]$  上
- 一般要求：  
 $[n] = \{1, \dots, n\}$
- “容易”计算哈希值：时空等资源占用少
- 较为平均/冲突较少

冲突：  $x \neq y$  但是  $h(x) = h(y)$   
注意到，若  $x = y$  则必有  $h(x) = h(y)$

如何做到尽可能均衡：  
随机哈希

# 随机哈希

- $h : [n] \rightarrow [m]$  是随机哈希，指对于每个  $i \in [n]$ ， $h(i)$  是  $[m]$  上的均匀随机分布
- 一般要求：对  $i \neq j$ ，那么  $\Pr[h(i) = h(j)] \leq 1/m$
- 术语：每个  $j \in [m]$  称为 bucket
- 典型行为：  $\forall j \in [m], \mathbb{E}[|h^{-1}(j)|] = n/m$

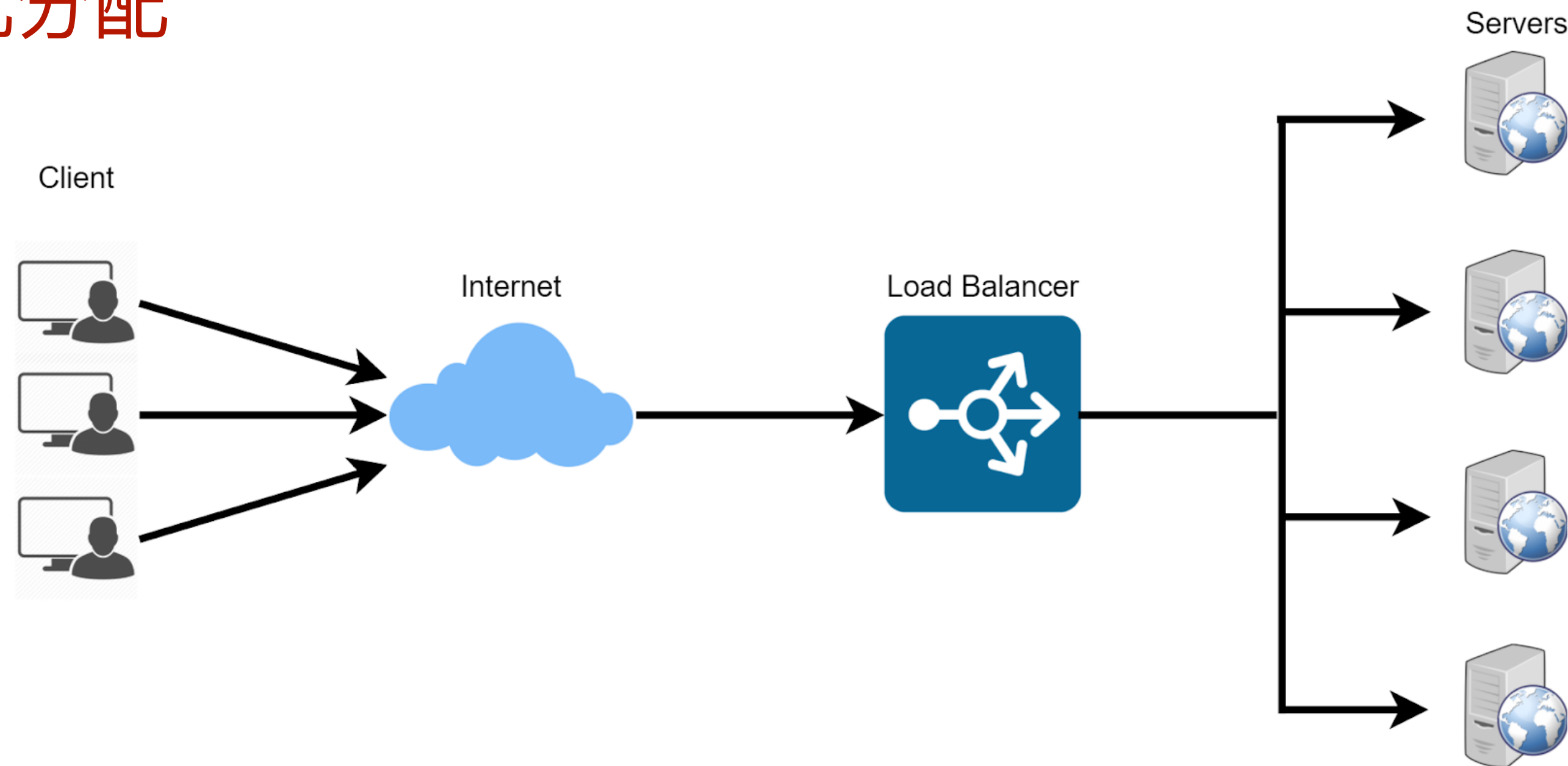
事实上这里  $\leq$  其实是  $=$ ，但应用上一般只用  $\leq$

一般反函数的定义：  $h^{-1}(j) := \{i \in [n] : h(i) = j\}$

随机哈希是与采样并列的另一大类随机算法设计工具

# 例子：分布式环境的负载均衡

- 问题：请求任意到达，如何设计均衡器，使得m台服务器处理的请求个数平均？
- 直观办法：分配任务前询问服务器当前负载，然后安排给负载最低的
- 高效方法：随机分配




效率低：需要与服务  
器额外通信、同步

# \* 随机哈希平衡性的高概率分析

- Claim: 设有 $n$ 个球均匀扔进 $n$ 个桶里, 1号桶 $1 - 1/n$ 概率会有 $O(\log n)$ 个球

**Chernoff bound (大误差版)** . 设 $X_1, \dots, X_n \in [0,1]$ 是独立随机变量。

令 $X = X_1 + \dots + X_n$ 。则对 $t \geq 1$ 有 与 $t \in (0,1)$ 的版本稍有不同!

$$\Pr[|X - \mu| \geq t\mu] \leq 2 \exp(-O(t\mu))$$

- 设 $X_i \in \{0,1\}$ 指示第 $i$ 次是否扔进了1号桶。 $X$ 代表1号桶球数,  $\mu = \mathbb{E}[X] = 1$

$$\Pr[|X - \mu| > \log n \cdot \mu] \leq \exp(-\log n)$$

# 再论负载均衡：互联网规模的哈希

- 旧场景：n个文件要分布式存储在m台机器上，要做负载均衡
- 新需求：服务器经常需要新增
- 考虑变成m + 1台机器：
  - 按照之前的随机hash则需要re-hash（在两个随机哈希种同一个元素大概率会对应不同bucket）

即从 $h : [n] \rightarrow [m]$ 改成  
 $h' : [n] \rightarrow [m + 1]$



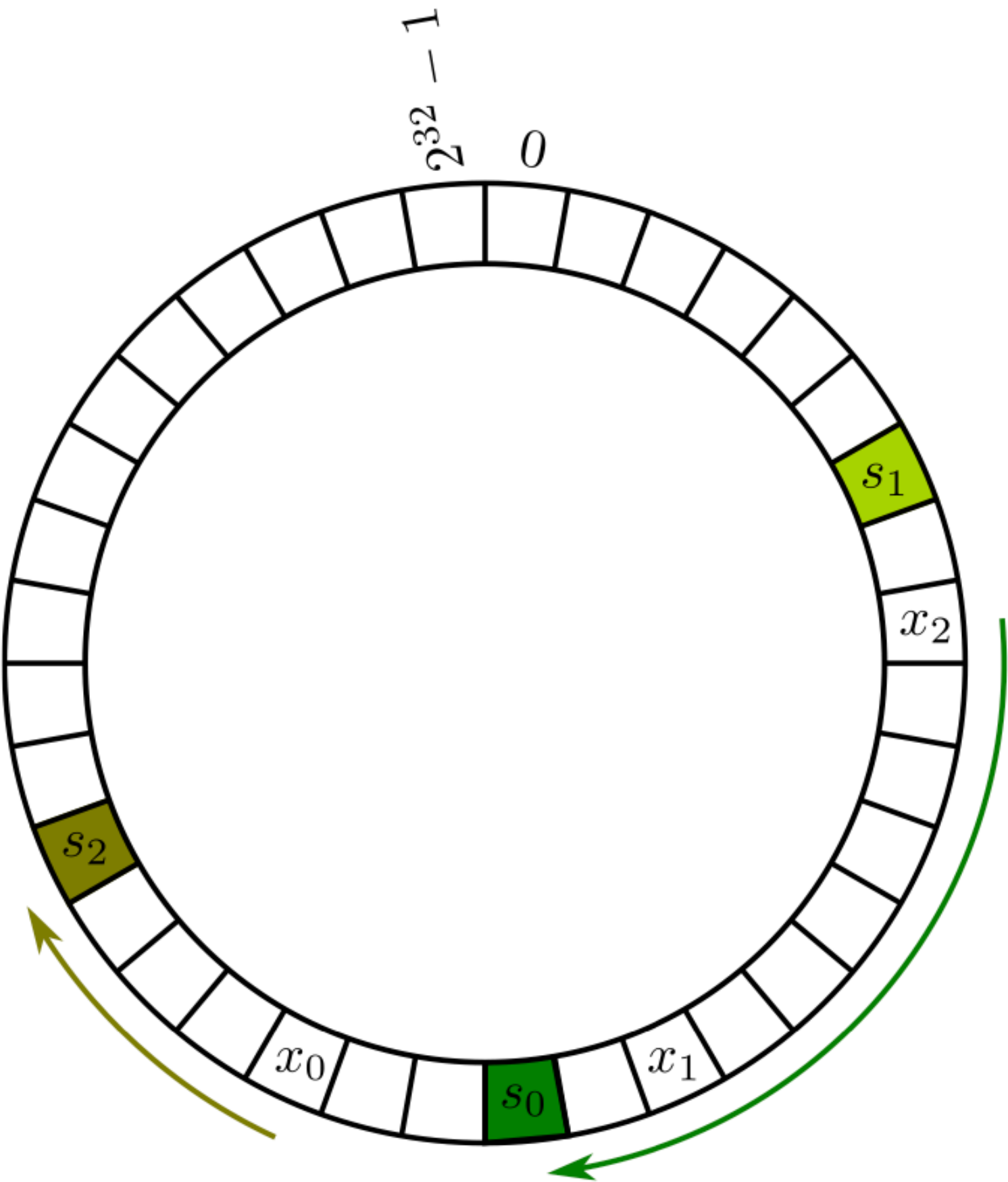
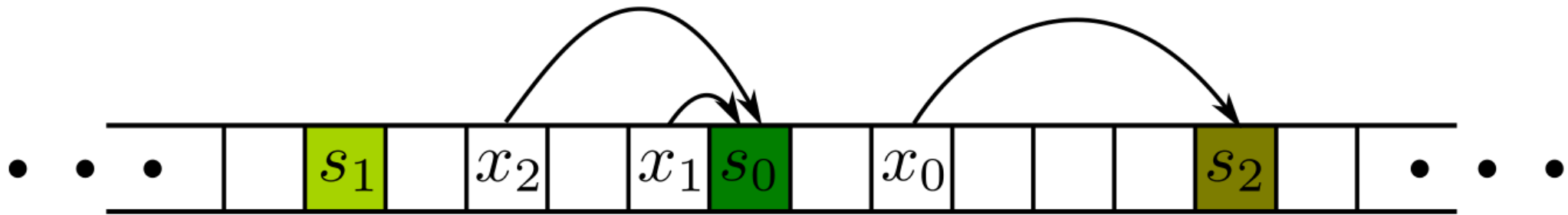
# “避免”Re-hash: Consistent Hashing

- 不只将文件hash、也将机器编号放在一起进行hash
  - 因此需要考虑一个域足够大的随机hash  $h : \text{int} \rightarrow \text{int}$  对应某个机器s的h(s)
- 对文件x, 先做 $h(x)$ , 然后往后找, 找到第一个非空的、对应某机器s的bucket
- 将文件存入机器s

# 示意图

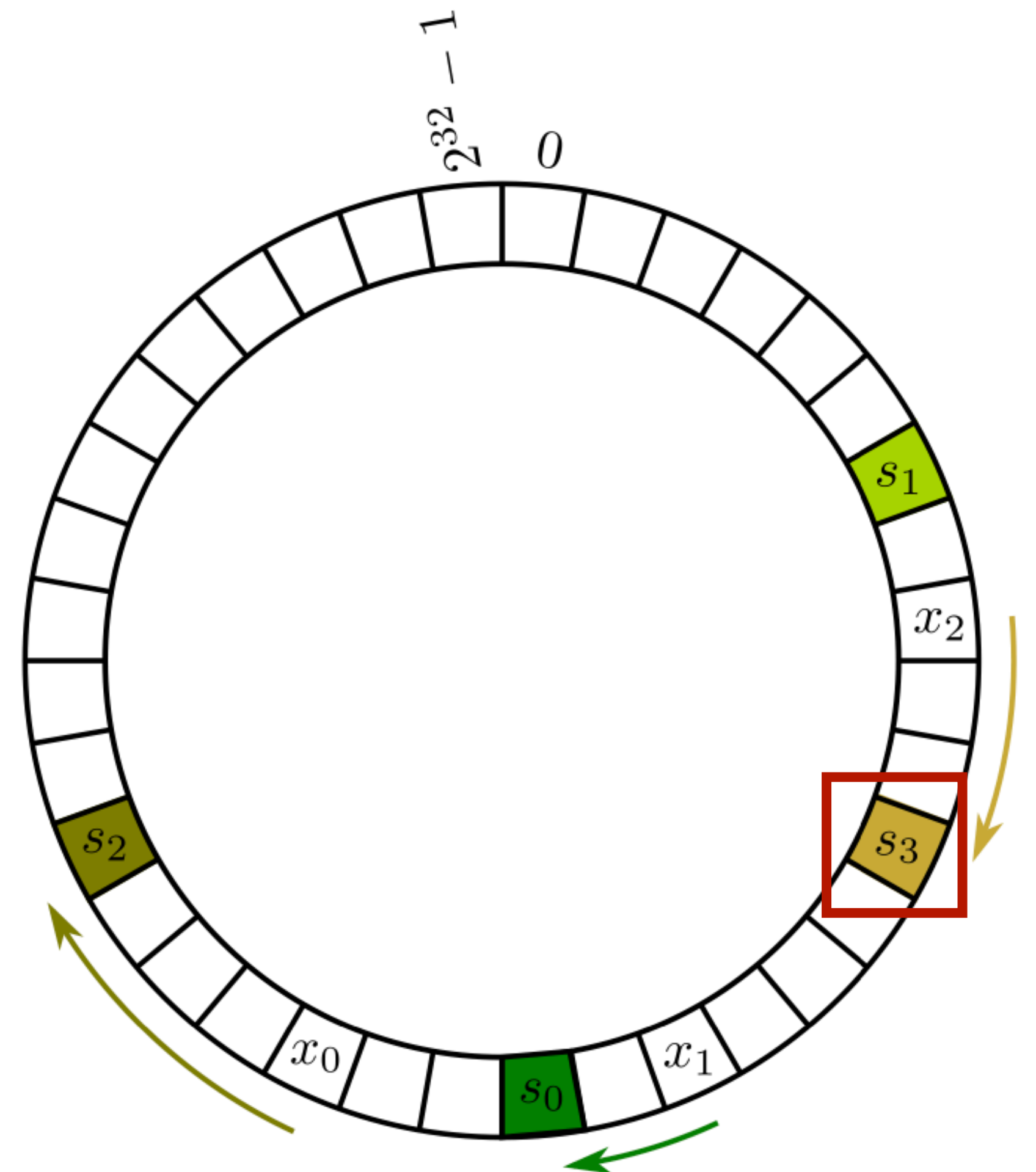
为方便处理边缘情况，可以把整个值域拼接起来，此时总是向顺时针方向找

算法示意图



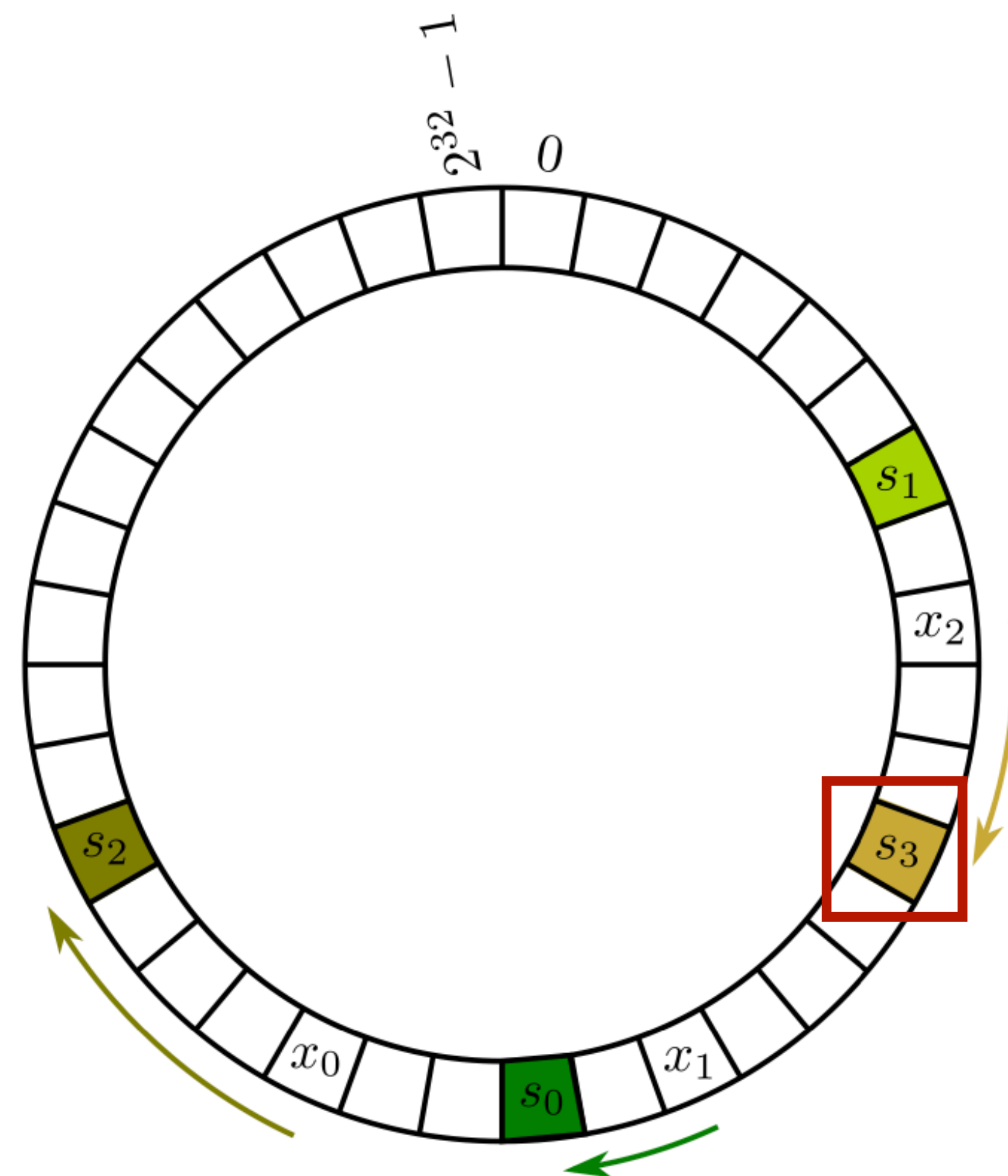
# 处理新增

- 如果新增一个服务器s呢?
- 只需要动很小的局部
- 由于hash是随机的，典型情况负载依然是均衡的
  - 将整个圆环分成了均匀间隔的部分




# 高效维护

- 可以维护（位置，文件/服务器id）的有序表
- 需要支持增删和查询后继的操作





- 第一次出现在[Karger et al., STOC 1997]
- 1998: Akamai成立  直到今天，仍是做CDN/负载均衡/网络安全等的巨头，以consistent hashing起家
- 1999年，apple.com是《星战魅影危机》预告片独家发行商，一放出来就瘫痪了，唯独Akamai的CDN上的copy可以正常下载

- 2000年代： p2p网络的核心问题是跟踪每个文件的位置（从哪里下载）
  - DHT（distributed hash table）
  - Consistent hashing是重要实现方法（但需额外的分布式路由、存储等算法）
  - 用在了BitTorrent等协议中
- [Stoica et al., ToN 2003]



2006: Amazon implements its internal Dynamo system using consistent hashing [1]. The goal of this system is to store tons of stuff using commodity hardware while maintaining a very fast response time. As much data as possible is stored in main memory, and consistent hashing is used to keep track of what's where.

This idea is now widely copied in modern lightweight alternatives to traditional databases (the latter of which tend to reside on disk). Such alternatives generally support few operations (e.g., no secondary keys) and relax traditional consistency requirements in exchange for speed. As you can imagine, this is a big win for lots of modern Internet companies.

- [1] = Dynamo: Amazon's highly available key-value store. DeCandia et al., SIGOPS 2007



# 哈希方法经典应用

# Point Query问题

- 问题设定：
  - 输入 $N$ 个 $[n]$ 上的整数，输入结束后给定 $i$ ，要求（近似）回答 $i$ 出现的次数
- 比如 $(1,1,2,2,3)$ ，最后1出现2次，2出现2次，3出现1次
- Trivial做法：暴力维护，最坏需要线性空间，但可以得到精确解

线性于输入的不同元素的个数

# Count-min Sketch

- Count-min sketch是一个数据结构，支持对于任何误差参数 $0 < \epsilon < 1$ :
  - 插入 $N$ 个 $[n]$ 上的元素后，给定一个 $x \in [n]$ ，估计 $x$ 共出现了多少次
  - 具体来说：以大概率满足 $C \leq \hat{C} \leq C + \epsilon \cdot N$ 
    - 大 =  $1 - 1/\text{poly}(n)$
    - $\hat{C}$ 是估计量， $C_x$ 是 $x$ 插入次数
- 总共使用空间 $\frac{\text{poly log } n}{\epsilon}$ ，且单次查询和插入单个元素时间 $\frac{\text{poly log } n}{\epsilon}$

# Count-min Sketch

## 大体思路

- 大致算法：
  - 构造一个随机哈希函数  $h : [n] \rightarrow [m]$
  - 将输入元素均匀映射到  $m$  个 bucket，对每个 bucket  $j \in [m]$  记元素个数  $C[j]$
- 如果没冲突，那么最后对  $x \in [n]$  查询输出  $C[h(x)]$  就是精确解
- 冲突的影响：多个元素的 count 累加在一起被报告了出来

$m \ll n$  是待定参数

$m \ll n$  因此必然有  
(很多) 冲突

只会造成对结果的高估!

# 数学期望分析

回忆：设计随机算法的三大步骤的第一步

- 设上述算法返回 $C[h(x)]$ ，真实的 $x$ 的出现次数是 $C_x$

$\mathbb{I}(\text{cond})$ 是指示函数，条件 $\text{cond}$ 满足  
返回1否则返回0

$$\begin{aligned}\mathbb{E}[C[h(x)]] &= \mathbb{E} \left[ C_x + \sum_{y \neq x} \mathbb{I}(h(y) = h(x)) \cdot C_y \right] = C_x + \sum_{y \neq x} \mathbb{E}[\mathbb{I}(h(y) = h(x)) \cdot C_y] \\ &= C_x + \sum_{y \neq x} \Pr[h(x) = h(y)] \cdot C_y \leq C_x + N/m\end{aligned}$$

回忆 $N$ 是插入的总元素数

由于算法只会高估，因此只要  
(对期望) 给上界

这步放缩看似很激进，但当  
 $C_x$ 相对于 $N$ 很小时基本是紧的

# 常数概率

## 设计随机算法的三大步骤的第二步



Markov inequality: 若 $X \geq 0$ , 有 $\Pr[X \geq a] \leq \mathbb{E}[X]/a$

- 设 $X := C[h(x)] - C_x \geq 0$ , 那么 $\mathbb{E}[X] \leq N/m$

- 对 $X$ 用Markov, 有 $\Pr[C[h(x)] - C_x > \epsilon \cdot N] \leq \frac{N/m}{\epsilon \cdot N} \leq \frac{1}{\epsilon m}$

取 $m = \frac{2}{\epsilon}$ 得到1/2概率

# 提升成功率：多次试验取最小值

设计随机算法的三大步骤的第三步

- 由于 $C[h(x)]$ 总是高估，因此可以多次试验取最小值
  - 每次以至少0.5概率高估不超过 $\epsilon N$ 的additive error
- 那么独立运行 $O(\log n)$ 次取min可以 $1 - 1/\text{poly}(n)$ 概率不超过 $\epsilon N$  additive error

# Count-mim Sketch: 完整算法

- 初始化:
  - 设置  $T = O(\log n)$  个独立的随机哈希  $h^{(i)} : [n] \rightarrow [m]$ , 这里  $m = 2/\epsilon$
  - 初始化  $T$  个  $m$  元 counter  $C^{(i)}[1 \dots m] = 0 \quad (1 \leq i \leq T)$
- 插/删  $x \in [n]$  时: 

count-min也支持删除

  - 对每个  $i = 1, \dots, T$ , 将  $C^{(i)}[h^{(i)}(x)]$  增/减 1
- 查询  $x \in [n]$  时: 返回  $\min_{1 \leq i \leq T} C^{(i)}[h^{(i)}(x)]$



# Count-min的可合并性

- 注意到count-min的counter是可加的

注意：需要使用同样的一组hash  $h^{(i)}$  才能保证counter数组  $C^{(i)}$  可以对应相加

- 因此，设分别有对A和B的count-min，counter相加可得  $A \cup B$  数据的count-min
- 类似地，counter相减可以得到从A上删除B数据集的count-min
- 这种可合并性质对于并行计算十分友好

# 应用：Count-min做数据流Heavy Hitter

至多有k个

- k-Heavy hitter (k-HH): A是长度N元素在[n]上的数组，求出现次数  $\geq N/k$  的元素
- 应用
  - A是亚马逊/京东物品成交/访问记录，要快速得到当日最流行爆款商品
  - A是路由器上的IP访问日志，快速找到异常流量防DoS攻击
- 这些应用都是大数据/小内存，因此我们考虑数据流算法
- 数据流：数组A中元素以任意顺序以数据流方式给出，算法在流结束时给出k-HH

相关问题：返回top-k出现次数元素  
k-HH一定在top-k，但是反之不然！

一般关注空间复杂度；  
处理全流总时间一般追求  $npoly \log n$

数据流可以理解成无随机访问，类似于磁带

# 近似k-HH

- 考虑近似k-HH：给定  $0 < \epsilon < 1$ ，要求：

真正的k-HH都被返回了

- 出现至少  $N/k$  次的元素必须返回

被返回的可能不够  $N/k$ ，但也必须足够heavy

- 返回的所有元素出现不少于  $N/k - \epsilon N$  次

# Count-min做Streaming近似k-HH

- 先考虑用 $O(n)$ 空间暴力做法：维护数组A，找到出现  $\geq N/k$  次的元素

此处把count-min当黑盒，不需要再次考虑hash冲突等问题！

- 数据流算法：将暴力维护的数组A替换成count-min sketch，即：
  - 维护误差 $\epsilon$ 的count-min sketch
  - 插入每个元素后，利用count-min sketch来查询当前元素的出现次数
  - 把count-min sketch回答  $\geq N/k$  的元素存下来

实际出现次数其实是  $\geq N/k - \epsilon N$

# 如果不知道N呢?

- 刚才的算法都是假定数据流开始前就知道数据流长度N的；如果不知道呢？
- 额外维护当前插入的总元素个数 $M$
- 维护一个表L，每插入一个元素 $x$ ，查询count-min，发现 $x$ 超过 $M/k$ 次就存入L
- 每次插入后，扫描L中元素，删除不够 $M/k$ 次的元素

此处“出现次数”依然需要查询count-min

注意判重：若发现已经有同样的 $x \in [n]$ 在表L内，只需要保留一份  
该优化是为了保证总空间复杂度

# 数据流近似k-HH：完整算法

- 输入参数：误差  $0 < \epsilon < 1$ ，整数  $k \geq 1$ ，输入元素取值范围  $n$
- 初始化：  $M = 0$ ，一个在  $[n]$  上的 count-min 数据结构  $P(\epsilon)$ ，一个集合  $L$
- 在数据流中，当  $x \in [n]$  被插入时：
  - $M++$ ；将  $x$  插入  $P(\epsilon)$ ；若  $P.\text{count}(x) \geq M/k$ ，将  $x$  插入  $L$
  - 检查  $L$  中所有元素  $y$ ，如果  $P.\text{count}(y) < M/k$ ，则把  $y$  从  $L$  删除
- 数据流结束时，返回  $L$

要求元素去重

重复元素会被去除

任何时候  $L$  中元素  $\text{count} \geq M/k - \epsilon M$ ，因此设

$$\epsilon \leq \frac{1}{2k}, \text{ 则有 } |L| = O(k)$$

# 如何实现随机哈希

# 完全随机哈希带来的问题

- 在count-min中我们要求使用（完全）随机的哈希函数
- 对于 $n$ 个不同数字的输入，需要至少 $\Omega(n)$ 个数来存储哈希值

具体来说，每个输入生成一个随机数，  
需要都存下来

- 这部分导致的存储代价在追求空间复杂度的情况下是不可忽略的



# 解决方案

- 实际中常用“足够随机”（又快速）的哈希，如MD5等
  - 作业中给出的代码也是用的这种类型的哈希
- 具有理论保证的：universal hashing

# Universal Hashing

- 要构造使用小空间的随机  $h : [n] \rightarrow [m]$ , 使得

$$\forall x \neq y \in [n], \quad \Pr[h(x) = h(y)] \leq \frac{1}{m}$$

- 算法:

- 不失一般性, 假设  $m$  为素数 (即把  $m$  替换成最小素数  $m' \geq m$ )
- 设  $k = \lceil \log_m n \rceil$ , 取  $k$  个  $\{0, \dots, m-1\}$  上的均匀独立随机数  $\{a_i\}_{i=1}^k$

- 给定  $x \in [n]$ , 把  $x$  表成  $m$ -进制数, 定义  $h(x) := \sum_i a_i x_i \bmod m$

至多  $k$  位

$x_i$  是  $m$ -进制表示第  $i$  位

# 分析

1. 不失一般性，假设 $m$ 为素数（即把 $m$ 替换成最小素数 $m' \geq m$ ）
2. 设 $k = \lceil \log_m n \rceil$ ，取 $k$ 个 $\{0, \dots, m-1\}$ 上的均匀独立随机数 $\{a_i\}_{i=1}^k$
3. 给定 $x \in [n]$ ，把 $x$ 表成 $m$ -进制数，定义 $h(x) := \sum_i a_i x_i \bmod m$

- 若 $x \neq y$ 则存在 $1 \leq i \leq k$ 使得 $x_i \neq y_i$ ，那么

$$\Pr[h(x) = h(y)] = \Pr \left[ a_i \equiv - (x_i - y_i)^{-1} \sum_{j \neq i} a_j (x_j - y_j) \bmod m \right] = \frac{1}{m}$$

$m$ 是素数， $x_i - y_i \not\equiv 0 \bmod m$ 因此一定可逆

$a_i$ 取任何 $\{0, \dots, m-1\}$ 上任何单个值的概率是 $1/m$

# 总结

随机数在 $0, \dots, m - 1$ 上

- 对于任意的整数 $n$ 和 $m$ ，存在一个使用 $O(\log_m n)$ 个随机数的函数 $h : [n] \rightarrow [m]$

$$\forall x \neq y \in [n], \quad \Pr[h(x) = h(y)] \leq \frac{1}{m}$$

- 这个函数虽然无法保证 $h(x)$ 和 $h(y)$ 独立，但是能确保期望意义上的“负载均衡”
- 这对于count-min等应用已经足够了

检查一下count-min的分析

# 实际编程？

- 实际编程一般不采用上述“严格”解决方案，因为过程相对复杂/“常数”比较大
- 相反，采用一些类似MD5的哈希函数，即直接hash到int64上
  - 虽然未必有最坏情况保证，但实际上来看非常有效
- 我们在作业的样例代码中提供了一个这样的快速哈希函数，可供参考
  - 基于：<https://github.com/rurban/fast-hash/blob/master/fasthash.c>
  - See also <https://code.google.com/archive/p/fast-hash/>

# 作业七： k-HH的数据流算法

分值： 2分

- <http://cssyb.openjudge.cn/24hw7/>
- 用刚刚介绍的基于count-min sketch的方法解决数据流近似k-HH问题
- 为了采用数据流输入和控制空间，此题需要使用提供的交互库
  - 数据流算法一般分为三个过程：初始化、更新和查询
  - 数据流开始之前运行初始化；数据流插删元素运行更新；数据流结束运行查询
- Deadline： 3月20日

# 数据流精确找Majority

即出现至少一半次数的元素

- 如果仅仅找majority，那么可以用更简单的确定性方法找到精确解
- 观察：考虑二进制表示，如果一个元素 $x$ 出现超过一半，那么单看所有元素的某一位 $t$ ， $x_t$ 出现也是过半的
- 比如 $\{110, 110, 110, 111\}$ ，那么在所有3位中，110对应位置的值出现过半
- 算法：维护总元素个数 $c$ ，及对 $1 \leq t \leq \log n$ ， $c_t$ 代表第 $t$ 位1的个数

支持插入和删除！

- 输出：
$$\sum_{i=1}^{\log n} 2^{i-1} \cdot \mathbb{I}(c_i \geq c/2)$$

# 一种只支持插入的Majority算法

- 初始化  $\text{counter} = 0$ ,  $\text{current} = \text{NULL}$

for  $i = 1$  to  $n$

    if  $\text{counter} == 0$ :  $\text{current} = A[i]$ ,  $\text{counter}++$

    else if  $A[i] == \text{current}$ :  $\text{counter}++$

    else:  $\text{counter}--$

return  $\text{current}$



# 数据流近似Inner Product

## Count-min的另一个应用

$$\langle a, b \rangle := \sum_{i=1}^n a_i b_i$$

- 问题：给出两个在 $[n]$ 上的frequency vector  $a$ 和 $b$ ，估计 $\langle a, b \rangle$ 
  - frequency vector:  $n$ 维向量，统计每个 $i \in [n]$ 出现多少次
- 应用场景：数据库Join操作结果集大小预估
- Join：两个表中都有某个列，将列元素相等的所有行做笛卡尔积 (Cartesian product)
  - Cartesian product:  $(a, b, c) \times (x, y, z) = (ax, ay, az, bx, by, bz, cx, cy, cz)$
  - 计算Join很费时，需要预估每次Join的代价

把两张表“拼起来”看

# 例子：根据学号求Join

| 学号 | 参与社团 |
|----|------|
| 1  | A    |
| 1  | B    |
| 2  | A    |

| 学号 | 修课 | 成绩  |
|----|----|-----|
| 1  | X  | 100 |
| 1  | Y  | 95  |
| 2  | X  | 90  |
| 2  | Y  | 80  |

| 学号 | 参与社团 | 修课 | 成绩  |
|----|------|----|-----|
| 1  | A    | X  | 100 |
| 1  | A    | X  | 100 |
| 1  | B    | Y  | 95  |
| 1  | B    | Y  | 95  |
| 2  | A    | X  | 90  |
| 2  | A    | Y  | 80  |

- 结果集大小计算：
  - 根据学号求frequency vector  $a = (2,1)$ ,  $b = (2,2)$
  - $\langle a, b \rangle = 6$

# 算法

- 维护对应于a和b的count-min sketch

下标a和b对应于针对a和b的sketch

- 对  $1 \leq i \leq T$ , 直接对bucket求inner product, 得到  $\hat{C}^{(i)} := \sum_{j=1}^m C_a^{(i)}[j] \cdot C_b^{(i)}[j]$

- 然后求  $\hat{C} := \min_{1 \leq i \leq T} \hat{C}^{(i)}$  作为最后估计

须使用同一组hash functions  $\{h^{(i)}\}_i$ , 否则对应counter相乘无意义

- 最后的保证是  $\langle a, b \rangle \leq \hat{C} \leq \langle a, b \rangle + \epsilon \cdot \|a\|_1 \|b\|_1$

用与count-min类似的分析可以得到如何选取参数, 以及如何达到类似count-min的空间复杂度

# Bloom Filter 布隆过滤器

## 高效的集合数据结构

- 维护一个数据结构，对应于一个集合S：
  - 全集 $[N]$ ，一系列插入、删除( $i, \pm 1$ )操作，询问某 $i \in [N]$ 是否在S里
- Bloom filter: 以 $O(n)$ 空间， $O(1)$ 时间，以大概率 $1 - \delta$ “成功”回答询问

这里 $n$ 是 $S$ 最大元素个数  
优势： $O$ 里面的常数特别小，实际确实省空间

- 成功概率的意义：“在”一定回答Yes，“不在”回答Yes的概率是  $\leq \delta$

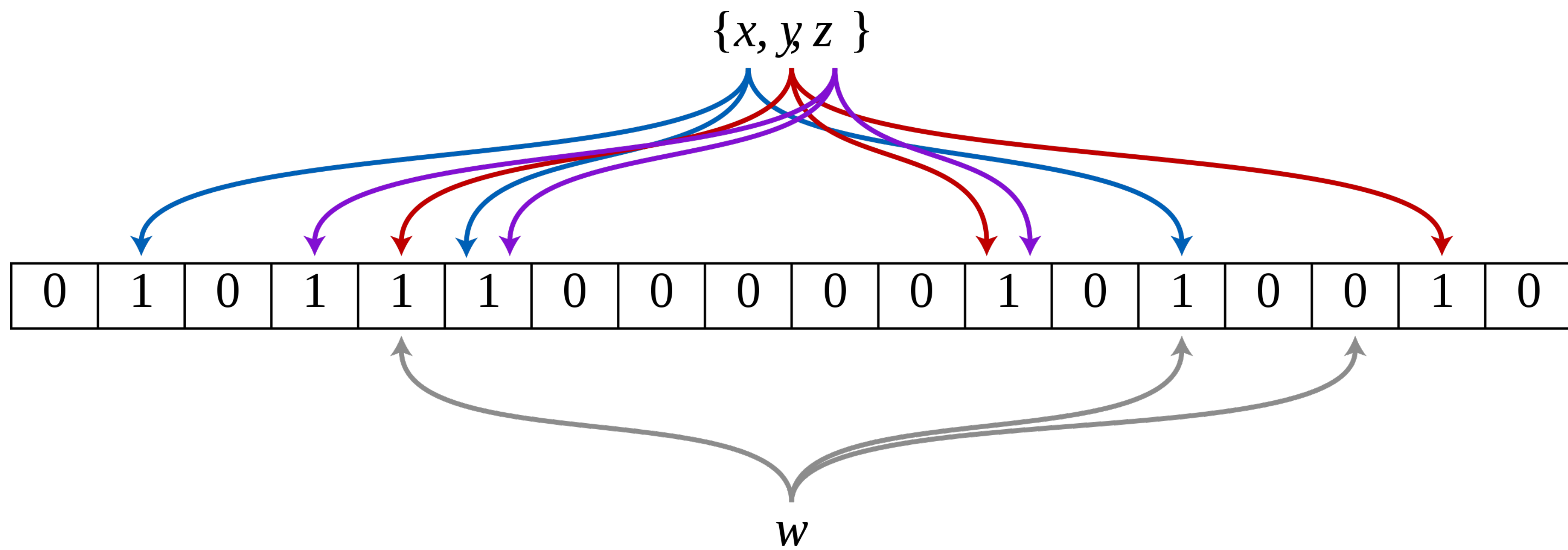
等价的（逆否）命题：回答No则一定不在

# Bloom Filter

## 具体算法

- 算法：
  - 维护一个长度是 $m$ 的0-1数组 $A$ ，用 $T$ 个随机哈希函数 $h^{(i)} : [N] \rightarrow [m]$
  - 插入 $x$ 时对所有 $1 \leq i \leq T$ ，把所有 $A[h^{(i)}(x)]$ 设成1
  - IsMember( $x$ ): 若对**所有** $1 \leq i \leq T$ ， $A[h^{(i)}(x)]$ 都是1才回答Yes，否则No

若 $x$ 确实出现，那么不会回答错



# Bloom Filter

T的选取：若x不出现，仍回答Yes的概率？

- 对于每个i，考察 $A[h^{(i)}(x)] = 0$ 的概率（的下界）
- 设 $t = h^{(i)}(x)$
- 当插入某个 $y \neq x$ 时，T个哈希函数都可能把 $A[t]$ 设成1
  - 那么都没设成1的概率  $\geq (1 - 1/m)^T$  不严格：因为每个位被set的概率可能是不独立的，但这样算也足够“准确”
- 有n个这样的y曾被插入，因此最终 $A[t] = 0$ 的概率

$$\geq (1 - 1/m)^{Tn} \approx \exp(-nt/m)$$

# 参数选取与时空复杂度

- 最后，只有当  $\forall 1 \leq i \leq T$ ，都有  $h^{(i)}(x) = 1$  才会（错误地）回答Yes
- 这个概率至多是  $(1 - \exp(-nT/m))^T$
- 如何选取参数  $m$  和  $T$ ？

- 设  $m$  和  $n$  都是定值，对  $T$  求导，最后可以得到  $T = \frac{m}{n} \ln 2$  概率最小

- 令失败概率  $= \delta$ ，则可以解得  $m = \frac{n \ln \delta^{-1}}{(\ln 2)^2}$ ， $T = \log_2(\delta^{-1})$

空间  $O(nT)$ ，时间  $T$

$$\frac{m}{n} \approx -1.44 \log_2 \delta$$



# 与传统的字典结构的比较

- Bloom filter不需要存储关键字/数据内容本身
  - 即使不支持快速查询的数组/链表，都需要存储数据本身来实现查找！
- 每次query可在严格 $O(1)$ 时间完成，该代价是一个不依赖于数据的、绝对的常数
  - 对于硬件实现Bloom filter算法非常有帮助

尤其对字符串/图片等数据可以大幅提升性能

# 与Count-min Sketch比较

- “有0则返回No”这一步类似于count-min里面的取min
- count-min可以理解成一种计数类型的Bloom filter
- Bloom filter需要线性空间，count-min的空间是亚线性的
  - count-min的亚线性是因为允许一个 $\epsilon N$ 的additive error，而这个additive error对Bloom filter的set membership查询是没意义的

# 广泛的实际应用

- The servers of [Akamai Technologies](#), a [content delivery](#) provider, use Bloom filters to prevent "one-hit-wonders" from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.<sup>[12]</sup>
- Google [Bigtable](#), [Apache HBase](#) and [Apache Cassandra](#) and [PostgreSQL](#)<sup>[13]</sup> use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.<sup>[14]</sup>
- The [Google Chrome](#) web browser previously used a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).<sup>[15][16]</sup>
- Microsoft [Bing \(search engine\)](#) uses multi-level hierarchical Bloom filters for its search index, [BitFunnel](#). Bloom filters provided lower cost than the previous Bing index, which was based on [inverted files](#).<sup>[17]</sup>
- The [Squid Web Proxy Cache](#) uses Bloom filters for cache digests.<sup>[18]</sup>
- [Bitcoin](#) used Bloom filters to speed up wallet synchronization until privacy vulnerabilities with the implementation of Bloom filters were discovered.<sup>[19][20]</sup>
- The [Venti](#) archival storage system uses Bloom filters to detect previously stored data.<sup>[21]</sup>
- The [SPIN model checker](#) uses Bloom filters to track the reachable state space for large verification problems.<sup>[22]</sup>
- The [Cascading](#) analytics framework uses Bloom filters to speed up asymmetric joins, where one of the joined data sets is significantly larger than the other (often called Bloom join in the database literature).<sup>[23]</sup>
- The [Exim](#) mail transfer agent (MTA) uses Bloom filters in its rate-limit feature.<sup>[24]</sup>
- [Medium](#) uses Bloom filters to avoid recommending articles a user has previously read.<sup>[25]</sup>
- [Ethereum](#) uses Bloom filters for quickly finding logs on the Ethereum blockchain.