《数据科学与工程算法》项目报告

报告题目：数据流统计算法

姓　　名：郭夏辉

学　　号：10211900416

完成日期：2024.05.01

摘要 [中文]：

数据科学技术的发展过程中出现了很多有趣的问题，处理和分析大规模数据在面向流动的、源源不断到达的数据时相较于传统的面向静态数据的数据挖掘有显著的差异。面对动态的数据，需要新的技术来提炼其中的有效信息，本文介绍的数据流算法就是来应对这类问题的。在此背景下，以 Count Min Sketch 为代表的一系列算法可以较好地完成随机频繁项挖掘工作。本次实验我们面对的就是这样一个数据流场景，针对较大的数据规模之数据我们显然不能在有限资源、有限查询时间前提下像以往那样通过一些精确求解算法来挖掘。为了更好地实现成员查询、频度查询和 Top-k 查询这三项任务，也为了将三种功能的实现整合进一种结构，更为了寻求一种空间占用尽可能小的结构，我尝试了四种彼此之间有所联系，但是也存在差异的结构——Count-Min Sketch+MinHeap、Count-Min Sketch(conservative update) +MinHeap、Count-Min Log Sketch+MinHeap 和 SpaceSaving，通过实验结果发现以 SpaceSaving 为核心的结构不仅估计结果最为准确，而且代码实现简单、空间复杂度最小，全面有效地完成了本次实验。

Abstract [English]

In the development process of data science technology, many interesting problems have emerged. Handling and analyzing large-scale data in the context of streaming, continuously arriving data poses significant differences compared to traditional data mining approaches oriented towards static data. Facing dynamic data requires new techniques to extract meaningful information. The data stream algorithms described in this article are designed to address such challenges. In this context, a series of algorithms represented by Count Min Sketch can effectively accomplish the task of mining random frequent items.

In this experiment, we are confronted with a data streaming scenario where, given the large scale of data, we obviously cannot employ traditional precise solving algorithms under limited resources and query time constraints. In order to better implement the tasks of membership query, frequency query, and Top-k query, and to integrate the implementation of these three functions into a single structure, as well as to seek a structure with minimal space consumption, I attempted four structures that are related to each other but also have differences — Count-Min Sketch + MinHeap, Count-Min Sketch (conservative update) + MinHeap, Count-Min Log Sketch + MinHeap, and SpaceSaving.

Through experimental results, it was found that the structure centered around SpaceSaving not only provides the most accurate estimation results but also has simple code implementation and the smallest space complexity. It effectively completed this experiment comprehensively.

一、 项目概述（阐明项目的科学价值与相关研究工作，描述项目主要内容）

传统的数据挖掘或者数据分析主要面向的是静态数据，即数据存在磁盘上，可以多次访问。如今在很多应用场景中，这个前提可能不再成立。例如，在电信领域，网络流量经由交换机进行转发，不仅到达的速度快，而且数据量大；在科学研究领域，大型的科学仪器产生的数据也具有类似特征。举一个具体的例子，在智慧城市应用中，城市中部署了海量的物联网设备，这些设备也会持续不断地收集交通、人流的数据。以路径导航为例，所有终端需要及时更新当前位置信息，综合当前路况信息，实现路径的动态调整。对于导航平台来说，每时每刻有海量的终端同时提交服务请求，导航平台如何才能及时处理这些服务请求，并准确返回路况信息。在这类的应用中，数据不仅持续不断到达，而且数据规模庞大，具有非常典型的动态特征。传统的静态数据处理方法显得力不从心，亟需新的技术分析挖掘这些动态数据。本文介绍的数据流算法就是来应对这类挑战的。

数据流，顾名思义指的是流动的、源源不断到达的数据。实际应用中，数据流往往具有特征包括：

1. 数据总量不受限制：由于数据源源不断的到达，很难准确估算数据量的大小；
2. 数据到达速率快：很多实际应用中，数据产生的速率都非常快。比如，欧洲核子研究中心的大型强子对撞机每秒钟可以产生 40E 字节的数据；
3. 数据到达次序不受约束：事先无法预知数据的到达次序，比如，网络运营商网络中的 IP 数据包，网络交换机在接收到数据包之前，无法预知到达的数据包的内容；
4. 除非刻意保存，每个数据只能“看”一次：前面已经提到，随着数据持续不断地到达累积的数据量越来越大，很少有系统能把海量的数据都放在内存中，并且因为数据到达的速度过快所以为数据查询和分析预留的时间很短，否则便会出现数据阻塞。学过计算机系统的我们也知道多遍读取数据需要多次从磁盘中加载这些数据到内存中，但是这样导致了时间消耗的显著增大，在现实中是不可取的。

基于以上对流数据的介绍，传统的静态算法已不能胜任这些场景，动态的流数据算法在有限资源约束的情形下，需要能够处理海量的实时流数据。由于流数据模型具有与传统数据模型不同的特点，对基于流数据模型的数据处理提出了新的挑战，主要包括以下几个方面：

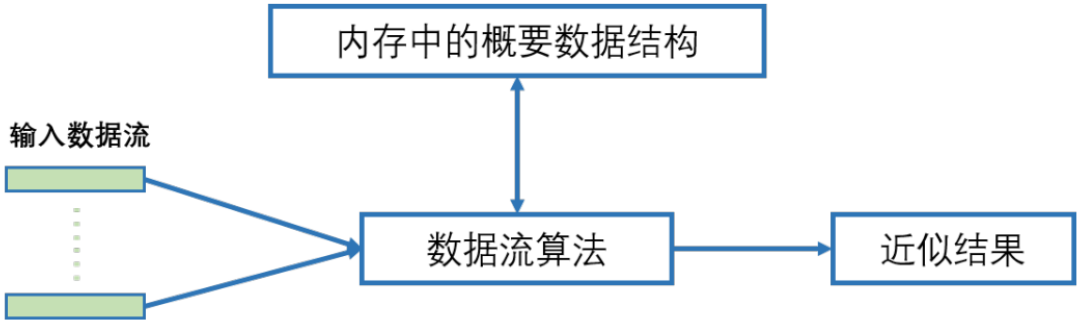
1. 实时性 能够实时、连续地输出查询结果。对于数据流上的任意一个元素，数据流算法均能够很快完成处理，否则由于数据不断到达，时间延迟不断积累，将会导致数据拥塞;
2. 低空间复杂度 前面提到，流数据模型下数据流的规模在理论上是无限制的，为了保证算法高效稳定运行，需要降低算法的空间开销。比如，交换机检测异常流量时，交换机的存储空间非常有限，即使异常流量检测算法的空间消耗是线性的，交换机也不具备把所有数据加载在内存中的条件;
3. 结果的近似准确性 流数据模型下数据规模大、速率快，因此对于一些复杂问题，不太可能通过数据的一次遍历就获得准确的答案。幸运的是，实际应用中往往不要求精确的查询结果。比如：在 DDOS 攻击检测中，仅需要计算指向某个目标地址的数据包数量的量级。如果精确的数据包的数量是 100，即

使算法给出的估计值是 200，也不会对服务质量产生太大的影响;

4. 适应性 在很多应用中，涉及到对多个流数据的处理。例如在传感器网络中，各个传感器节点同时采集数据，并且发送到中心主机中进行集中处理。传感器节点发送的速率和外部环境紧密相关，当某特定事件发生时，邻近传感器节点需要发送更多的数据。在这种多流的应用场景中，设计具备适应性的算法，根据各个数据流的变化及时调整算法参数，从而提高性能也是一个问题。

至今，流数据管理仍然是数据库领域的研究热点之一。主要包括从数据流中计算和挖掘各种统计量，如最大值（max）、最小值（min）、和（sum）、平均值（avg），以及计算中位数、分位数、频繁元素等稍复杂的统计量。

综上所述，在很多数据流应用中，计算资源和通讯资源是有限的，在内存中保存全部高速达到、总量不限的流元素是不切实际的。因此，在流数据查询和分析中，设计单遍扫描算法（One-pass Algorithm）实时地给出近似的查询结果是常见的处理方式。近似算法的关键在于设计一个远小于数据规模的数据结构——概要数据结构（Synopsis Data Structure）。概要数据结构不仅能够反映原始输入流数据的特征，而且它是可以加载在内存中的。一个经典的数据流模型算法的框架结构如下图所示。在这个架构中，若干数据流进入模型，每个流都是无限和连续的，流数据的时间戳随着流数据的不断到达而逐步递增。当“看到”一个新元素时，概要数据结构需要及时更新，反映流数据的新变化；当需要输出结果时，可以基于概要数据结构返回结果。对于不同的流数据算法，其概要数据结构也各不相同。直方图、抽样、小波、哈希等是常用的概要数据结构。



在实际的应用场景下，源于 Misra-Gries 算法，诞生了以 Count Sketch 为代表的一系列算法来进行频繁项挖掘的工作。然而，Misra-Gries 算法无法给出每个元素具体的频数估计，但是 Count Sketch 这个随机频繁项挖掘算法却可以做到。Count-Min Sketch 在 Count Sketch 基础上放弃了频数的无偏估计但是获得了更为有效的频数估计方法。Conservative Update 是 Count-Min Sketch 算法的一种微小但十分有效的优化策略，可以提高 Count-Min Sketch 算法的准确性和效率，特别是处理大规模和动态的数据流。SpaceSaving 算法是一种用于处理数据流的统计算法，主要用于解决 Top-K 频繁项查询问题。这种算法的主要优点是可以在有限的内存空间中处理大规模的数据流，同时还可以准确地估计数据的频度。Count-Min-Log sketch 采用 Conservative Update 策略的同时，在 Count-Min Sketch 基础上减小了 counter 的大小，减少空间浪费的同时进一步提升了算法的空间复杂度。本项目的目标是让我们设计并实现一个空间占用尽量小的数据结构，用于处理带有时间戳的用户对电影的评分信息。我将使用 Count-Min Sketch、Conservative Update 优化的 Count-Min Sketch、Count-Min Log Sketch 和 SpaceSaving 等数据流统计算法来实现成员查询、频度查询和 Top-k 查询这样的功能。

二、 问题定义（提供问题定义的语言描述与数学形式）

2.1 近似算法和频繁项估计

因为存储空间和时间的限制，在数据流上做到精确计算是非常困难的，备选方案是仅计算真实值 $\phi(\sigma)$ 的一个近似估计，这种算法被称之为近似算法。

其中， (ϵ, δ) 近似算法的定义如下所示：

给定输入的流数据 σ 和精确输出 $\varepsilon(\sigma)$,近似算法的输出记为A(σ)。该算法被称之为 (ϵ, δ)-近似算法,如果该算法的输出结果满足

$$Pr[|A(\sigma)-\varepsilon(\sigma)|<c\,\varepsilon(\sigma)]>1-\delta$$

这里给出的是相对版 (ϵ, δ)-近似算法, 一个 (ϵ, δ)-近似算法输出的结果可能会出现 $|A(\sigma)-\varepsilon(\sigma)|<c\,\varepsilon(\sigma)$,但是这种情况发生的概率不会超过 δ。通常偏差很大的概率上界 δ 是很小的一个值,这意味着 (ϵ, δ)-近似算法输出值以概率 $(1-\delta)$ 成为一个比较好的近似值。

在流数据管理领域，频繁项估计有很多的应用，比如 DDOS 攻击需要统计指向每个目标地址的流量包的频数。接下来，以频繁项估计为例来介绍流数据场景下的近似算法设计。

在数据流 $\sigma =< a_1, a_2, \cdots, a_m >, a_i \in [n]$ 中,定义一个频数向量 $f = (f_1, f_2, \cdots, f_n)$，其中 n 中不同元素的个数, f_i 为元素 a_i 的频数。

得到元素的频数后,可以找出满足需求的元素,比如频繁项。该问题分为两类:

大多数问题:如果 $\exists\, a_i : f_i > \frac{m}{2}$,则输出 a_i ,否则输出 $\varnothing$ 。

频繁项:给定一个参数k,输出频繁元素集合{ $a_i : f_i > \frac{m}{k}$ };或者,给定一个参数 ψ, 输出频繁元素集合 $a_1 : f_i > \psi m.$

2.2 本次实验要解决的三大问题

其实实验要求里就很好地阐明了我们要解决的三大类问题：

（1）成员查询：对给定查询时间戳和电影 id，查询该电影在该时间戳及之前是否曾被评分过；

（2）频度查询：对于给定查询时间戳和电影 id，查询该电影在该时间戳及之前被评分的总次数；

（3）Top-k 查询：对给定正整数 k 和查询时间戳，查询在该时间戳及之前被评分次数最多的前 k 个电影。

我们只要结合已有的算法进行相关的应用即可。

三、 方法（问题解决步骤和实现细节）

我首先来介绍一下自己使用的几个算法，再来结合代码来讨论我的实现细节。

3.1.1Count-Min Sketch

在课本和这篇文章("An improved data stream summary: the count-min sketch and its applications" by Cormode and Muthukrishnan, 2004.)中，都有介绍Count-Min Sketch 算法，这里我就以课本为主吧

CM Sketch 是一个宽度为 w 、深度为 d 的计数器数组：

$$C[1,1], C[1,2], ..., C[d,w]$$

初始化时，每个元素均为 0 。另有 d 个哈希函数：

$$h_i : \{1,2,...,n\} \to \{1,2,...,w\}, 1 \le i \le d.$$

是从两两互相独立的哈希函数族中随机均匀抽取得到的。一旦 w 和 d 确定下来，CM Sketch 所需空间便确定了。进一步地，可以用 dw 个计数器和 d 个哈希函数来表示该数据结构。该算法的伪代码如算法 5.5所示。

考虑一个维度为 n 的向量 **a** ，它在时刻 t 的状态为 $\mathbf{a}(t) = [a_1(t), a_2(t), ..., a_i(t), ..., a_n(t)]$ 。初始时， $\mathbf{a}(0)$ 中的所有元素都是 0 。假设第 t 次更新的量是 (i_t, c_t) ，那么：

$$a_{i'}(t) = \begin{cases} a_{i'}(t-1) + c_t & i' = i_t \\ a_{i'}(t-1) & i' \neq i_t \end{cases}$$

Algorithm 5.5: CM sketch 算法

输入: 数据流，查询元素 a
输出: 元素 a 出现的频数

1 **初始化**：$C[1 \cdots d][1 \cdots w] \leftarrow 0, w = \frac{2}{\epsilon}, d = \lceil log(1/\delta) \rceil$; 选择$d$个独立的哈希函数$h_1, h_2, \cdots, h_d : [n] \to [w]$
2 **处理**：(j, c)，其中$c = 1$;
3 **for** i = *1 to* d **do**
4 $\quad C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c$;
5 **return** $\widehat{f}_a = min_{1 \le i \le d} C[i][h_i(a)]$;

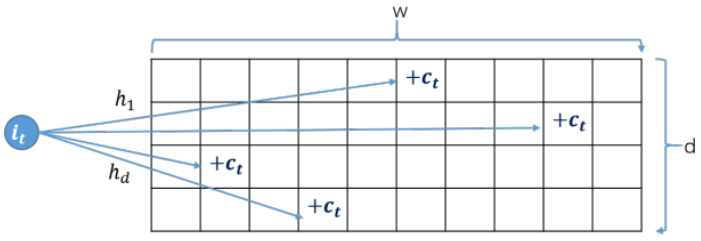


图 5.7 CM Sketch 第 t 次更新示例

具体地，如图 5.7 所示，当更新 (i_t, c_t) 到达时， c_t 被增加到 CM Sketch 数组中每行的某个计数器中，且第 i 行计数器是由哈希函数 h_i 确定的。可按如下方法进行更新：

$$\forall\, 1 \le j \le d : C[j, h_j(i_t)] \leftarrow C \quad [j, h_j(i_t)] + c_t.$$

特别的，在某些情况下，c_t 严格为正，这就意味着 CM Sketch 数组中的元素只会增加。而在 Count sketch 算法中，c_t 可正可负，经过多次更新后，计数数组中的元素可能大于 0，也可能小于 0。前者对应的是收银机模型，后者对应的是十字转盘模型。

对于给定的元素 a 和哈希函数 h_i，定义随机变量 X_i 表示在第 i 个哈希函数上其他元素对元素 a 频数估计上的贡献大小。

首先，对于 $j \in [n] \setminus \{a\}$，定义如下随机变量表示元素 j 在第 i 个哈希函数上与元素 a 冲突与否：

$$Y_{i,j} = \begin{cases} 1, & \text{如果} h_i(j) = h_i(a); \\ 0, & \text{否则.} \end{cases}$$

当 $Y_{i,j} = 1$ 时，元素 j 对于第 i 个哈希函数对元素 a 的频数估计起了作用，即元素 j 和元素 a 在第 i 个哈希函数上发生了冲突。所以，定义在第 i 个哈希函数上其他元素对元素 a 频数估计上的贡献为：

$$X_i = \sum_{j \in [n] \setminus \{a\}} f_j Y_{i,j}.$$

根据期望的线性性质，得到：

$$E[X_i] = X_i = \sum_{j \in [n] \setminus \{a\}} \frac{f_j}{k} = \frac{\|f\|_1 - f_a}{k} = \frac{\|f_{-a}\|_1}{k}.$$

因为 $f_j \geq 0$，所以 $X_i \geq 0$，运用马尔科夫不等式得到如下的尾概率：

$$P[X_i \geq \epsilon \|f\|_1] \leq P[X_i \geq \epsilon \|f_{-a}\|_1] \leq \frac{\|f_{-a}\|_1}{k\epsilon \|f_{-a}\|_1} \doteq \frac{1}{2}.$$

上面计算的概率是对于一个哈希函数而言，实际上有 d 个相互独立的哈希函数。若 $\widehat{f_a} - f_a \geq x$，则 $\min\{X_1, \cdots, X_d\} \geq x$。所以，

$$\begin{aligned} & P[\widehat{f_a} - f_a \geq \epsilon \|f_{-a}\|_1] \\ & = P[\min\{X_1, \cdots, X_d\} \geq \epsilon \|f_{-a}\|_1] \\ & = \Pi_{i=1}^{d} P[X_i \geq \epsilon \|f_{-a}\|_1] \leq \frac{1}{2^d}. \end{aligned}$$

可以适当选择 d 的值，使得上式概率至多为 δ。所以可以证明，至少以 $(1-\delta)$ 的概率下式成立。

$$f_a \leq \widehat{f_a} \leq f_a + \epsilon \|f_{-a}\|_1.$$

所以，算法　需要的计数器个数为：$M = O(\frac{\log 1/\delta}{\epsilon})$。

综上所述，结合文献，w: hash 表的大小(映射区间); d: hash 表的数量时，它们的大小近似和空间复杂度近似如下：

$$w = O(\frac{1}{\epsilon}), d = O(ln(\frac{1}{\delta}))\text{空间复杂度:}O(\frac{1}{\epsilon}ln(\frac{1}{\delta}))$$

3.1.2Conservative Update

在这篇文章(Estan, Cristian, and George Varghese. "New directions in traffic measurement and accounting." Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications. 2002.)中介绍了 Conservative Update 方法，CU sketch 只是对 CM 做了一项非常微小(slight)但却十分高效的改动，称为保守更新。即在每次插入元素时，只对 mapped counters 中最小的那个(或那些，即有多个最小)counter(s)进行增加操作，而查询操作与 CM 完全相同。

虽然 CU sketch 的准确率很高，但它有一个缺点在于不支持 delete 操作。因为当插入某个元素一段时间后，此时该元素的 mapped counters 中的最小者不一定就是原先插入时的最小者，若贸然删除不仅会影响到该元素，还会影响到共享该 counter 的其他元素。因此在实际应用中还是 CM 的出场率更高一些。

3.1.3SpaceSaving

这篇文章(Metwally, Ahmed, Divyakant Agrawal, and Amr El Abbadi. "Efficient computation of frequent and top-k elements in data streams." International conference on database theory. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.)提出了 SpaceSaving 算法。SpaceSaving 算法是一种近似计数算法，它在大多数情况下能够提供准确的结果。然而，它并不是完全准确的，因为它使用了一些近似技巧来降低内存使用。具体来说，SpaceSaving 算法通过牺牲一定的精度来减少内存消耗。在实际应用中，SpaceSaving 算法通常能够提供足够准确的结果，尤其是对于大规模数据集。但是，对于某些特定的数据分布或极端情况，可能会出现一些误差。因此，在使用 SpaceSaving 算法时，需要根据具体情况评估其准确性是否满足需求。具体做法如下所示：

如果元素在集合中，将其对应的计数器自增；

如果元素不在集合中且集合未满，就将元素加入集合，计数器设为 1；

如果元素不在集合中且集合已满，将集合内计数器值最小的元素移除，将新元素插入到它的位置，并且在原计数值的基础上自增。（这里维护计数值最小的元素可以用传统的堆）

可见，Space Saving 算法构建在 Misra-Gries 算法的基础上。这样操作的好处是，所有计数器的和一定等于数据流的总元素数 m（因为不需要做减法，只需要自增），且那些没有被移除过的元素的计数值是准确的。容易分析得出：

集合中最小的计数值 min 一定不会大于 $m / k = \varepsilon m$，同时能够保证找出所有频率大于 εm 的元素；元素出现频率的估计误差同样在 εm 的范围内，不过会偏高；

Space Saving 算法也有假阳性的问题，特别是在非频繁项集中位于流的末尾时。

3.1.4Count-Min Log Sketch

在这篇文章(Pitel, Guillaume, and Geoffroy Fouquier. "Count-min-log sketch: Approximately counting with approximate counters." arXiv preprint arXiv:1502.04885 (2015).)中，Count-Min Log Sketch 算法被提出来了。

CML sketch 的思想在于减小 counter 的大小：在传统的 sketch 中我们需要把所有的 counter 都设为可能出现的最大值，因为我们不知道哪个 counter 会被多次 hash 到。而大多数 counter 由于记录的是低频项，因此并不需要那么大的空间，这就造成了空间浪费。因此 CML 每次只以 x^(-c)的概率增加 counter 的计数，其

中 c 为对当前需要插入元素的估计值，x 为大于 1 的 log base，且增加计数时采用了 CU 策略。对应的插入和查询算法修改如下：

Algorithm 1 Count-Min-Log Sketch UPDATE

Input: sketch width w, sketch depth d , log base $b > 1$, independent hash functions $h_{1..d} : U \to \{1 \dots w\}$

function INCREASEDECISION(c)
  return True with probability b^{-c}, else False
end function
function UPDATE(e)
  $c \leftarrow \min_{1 \le k \le d} sk[k, h_k(e)]$
  if INCREASEDECISION(c) **then**
    for $k \leftarrow 1 \dots d$ **do**
      if $sk[k, h_k(e)] = c$ **then**
        $sk[k, h_k(e)] \leftarrow c + 1$
      end if
    end for
  end if
end function

Algorithm 2 Count-Min-Log Sketch QUERY

Input: sketch width w, sketch depth d , log base $b > 1$, independent hash functions $h_{1..d} : U \to \{1 \dots w\}$

function POINTVALUE(c)
  if $c = 0$ **then**
    return 0
  else
    return b^{c-1}
  end if
end function
function VALUE(c)
  if $c \le 1$ **then**
    return POINTVALUE(c)
  else
    $v \leftarrow$POINTVALUE($c + 1$)
    return $\frac{1-v}{1-x}$
  end if
end function
function QUERY(e)
  $c \leftarrow \min_{1 \le k \le d} sk[k, h_k(e)]$
  return VALUE(c)
end function

在计算元素的估计值时，因为插入时是按 x^(-c)概率进行，即每个计数相当于原先的 x^c 次，c 的范围为[0,c]，令 v = x^c，等比数列求和可得结果为 (1-v) / (1-x)，最后取所有 counter 的最小值

3.2 我对问题的理解和解决方法思考

介绍完了我用到的数据流处理算法，结合本次项目实际的三个任务，我来谈一下自己的理解和解决方法思考。

问题一（成员查询）和问题二（频度查询）的本质是一样的，只不过问题一侧重的是有没有在这个时间戳之前被评分过（估计值是不是 0？），问题二侧重的是这个时间戳之前被评分的次数（估计值的具体数值到底是什么？）。问题三（Top-k 查询）其实就是在前两个问题基础上的进一步引申，它要在所有的估计值数据中选择 k 个最大的作为频繁项。

如果要说对解决方法的思考，我觉得问题一和问题二直接使用 3.1.1-3.1.4 中的各个随机频繁项挖掘算法即可，但是问题三怎么做呢？笨办法肯定是把概要数据结构生成的 n 个估计值遍历并排序，但是即便采用最优秀的排序算法，多次 Top-k 查询的时间消耗也是难以忍受的，我们能不能寻求一种合适的方法动态完成 Top-k 查询这个工作，使得每次查询的时间消耗是 O(1)?这就让我想到了学习数据结构时学过的堆这个结构——保证父结点要比子结点大/小。假如我们每次查询的 k 值是不固定的，但是预先假定最大的 k 值为 k'，在 CMS 算法的基础上维护一个最大大小为 k'的最小堆便能动态地解决问题三，具体而言是这样的：在堆里查找该元素，如果找到，把堆里的计数器也增 1，并调整堆；如果没有找到，把这个元素的次数跟堆顶元素比较，如果大于堆顶元素的出现次数，则把堆顶元素替换为该元素，并调整堆。

值得注意的是，CMS 和 CMLS 以及 CUS 算法这样以 CMS 算法为基础的算法都需要结合最小堆这个数据结构才能在一个时间复杂度比较理想的情况下解决三个问题，**我不禁思考能不能用一种结构同时完成三种操作？**我觉得自己的瓶颈主要在于估计值集合的规模可能比较大，每次做排序算法可能时间效率很低。但是，再次了解算法之后我发现 SpaceSaving 算法通过固定 counter 的大小，虽然牺牲了一些准确性，**但是它的 counter 规模大幅缩小，我每次只需要在 counter 的元素中找 k 个最大的，这样做一方面减少了每次排序的成本，另一方面估计频数的成本也大幅降低，因此直接使用 SpaceSaving 算法排序 k 个得到的估计值在时间上消耗不会那么大，完全可以基于 SpaceSaving 算法整合一种结构完成三种操作。**

3.3 我具体是如何解决问题的

这一块我将结合自己的代码来阐述怎样一步一步利用算法、解决问题的。

3.3.1 CountMinSketch 算法的框架

我根据 CMS 算法的流程，利用 hashlib 库的方法，实现它还是十分容易的：

```
from array import array
from math import log, e, ceil
import hashlib
```

```
class CountMinSketch(object):
    def __init__(self, w=None, d=None, delta=None, epsilon=None,
is_conservative_update=False):
        """
        w: hash 表的大小(映射区间); d: hash 表的数量
        如果 w 和 d 设置了 delta 和 epsilon 就没必要设置了
        delta: 查询错误的最大概率 epsilon: 查询错误的最大偏离值
        w = ceil(e/epsilon)
        d = ceil(ln(1.0/delta))
        "An improved data stream summary: the count-min sketch and its
applications" by Cormode and Muthukrishnan, 2004.
        """
```

首先，它会根据输入的参数来确定 hash 表的大小 w 和数量 d。如果 w 和 d 已经提供，就直接使用。如果没有提供，那么就根据 delta 和 epsilon 来计算 w 和 d。delta 是查询错误的最大概率，epsilon 是查询错误的最大偏离值。计算公式是 `w = ceil(e / epsilon)` 和 `d = ceil(log(1.0 / delta))`。

然后，它会创建一个二维数组 `self.table`，这个数组的维度是 d 行 w 列。每个元素的初始值都是 0。这个二维数组用于存储数据流中元素的频率信息。

最后，它会设置是否使用保守更新的标志 `is_conservative_update`。

这里有个值得注意的地方，结合原始的论文，我发现 w 的取值一般是 e/epsilon 而不是 1/epsilon。

```
        if w is not None and d is not None:
            self.w = w
            self.d = d
        elif delta is not None and epsilon is not None:
            self.w = int(ceil(e / epsilon))
            self.d = int(ceil(log(1. / delta)))
        else:
            raise Exception("Incomplete parameters. Please provide w&d or
delta&epsilon.")

        # 数组元素的类型是 long
        self.table = [array('l', (0 for _ in range(self.w))) for _ in
range(self.d)]
        self.is_conservative_update = is_conservative_update

    def _hash(self, x):
```

它会对输入的元素 x 进行 hash，生成 d 个 hash 值。这里使用的是 MD5 hash 函数。

这个方法返回一个生成器，每次产生一个 hash 值。

下面这行代码首先对输入的元素 x 进行内置的 hash 处理，然后将 hash 值转换为字符串，并编码为字节串。接着，使用 MD5 算法对这个字节串进行 hash，得到一个 MD5 hash 对象：

```
        md5 = hashlib.md5(str(hash(x)).encode())
        for i in range(self.d):
```

在每次循环中，将循环变量 i 转换为字符串，并编码为字节串，然后更新 MD5 hash 对象。这样做的目的是为了生成不同的 hash 值：

```
            md5.update(str(i).encode())
```

下面这行代码首先将 MD5 hash 对象转换为 16 进制字符串，然后将这个字符串转换为整数。最后，对这个整数进行模运算，模数为 w，w 是 Count-Min Sketch 中 hash 表的大小。这样做的目的是为了将 hash 值映射到 hash 表的范围内。yield 关键字表示这个方法是一个生成器，每次调用都会生成一个新的 hash 值：

```
            yield int(md5.hexdigest(), 16) % self.w
```

总的来说，我的代码的功能是为输入的元素 x 生成 d 个 hash 值，这些 hash 值会被用于 Count-Min Sketch 算法中的数据插入和查询操作。**我的代码不仅可以面向整数数字，还可以面向字母、小数等数据，健壮性还是非常强的。**

因为 conservative update 其实就只是在 CMS 更新时候进行的，所以只用添加特判即可：

```
    def add(self, x, v=1):
```

它会将元素 x 添加到 Count-Min Sketch 中，增加元素 x 的频率。这里的 v 是元素 x 出现的次数。 如果设置了保守更新，那么只有当前位置的计数等于查询结果时，才会更新计数。否则，直接更新计数。

```
        # 元素 x 出现了 v 次
        if self.is_conservative_update:
            min_count = self.query(x)
            for table, i in zip(self.table, self._hash(x)):
                if table[i] == min_count:
                    table[i] += v
        else:
            for table, i in zip(self.table, self._hash(x)):
                table[i] += v

    def query(self, x):
```

它会查询元素 x 在 Count-Min Sketch 中的估计频率。这个频率是通过取所有 hash 位置的最小值得到的：

```
        # 元素 x 的估计出现次数
        return min(table[i] for table, i in zip(self.table, self._hash(x)))

    def __getitem__(self, x):
        return self.query(x)

    def __setitem__(self, x, v):
        for table, i in zip(self.table, self._hash(x)):
            table[i] = v
```

3.3.2 SpaceSaving 算法的框架

```
import pandas as pd
class SpaceSaving(object):
    def __init__(self, k):
        self.counters = {}
        self.k = k

    def add(self, x):
```

首先检查元素 x 是否已经在 self.counters 中。self.counters 是一个字典，用于存储元素及其对应的频率。如果元素 x 已经在 self.counters 中，那么就将其频率加 1；如果元素 x 不在 self.counters 中，首先检查如果添加元素 x 后，self.counters 的大小是否会超过 self.k，self.k 是 SpaceSaving 数据结构的大小限制。如果 self.counters 的大小会超过 self.k，那么就找出 self.counters 中频率最小的元素，赋值给 min_counter，然后删除 self.counters 中的 min_counter 元素，并将其频率加 1 后赋值给新元素 x。如果 self.counters 的大小不会超过 self.k，直接将新元素 x 添加到 self.counters 中，其频率设置为 1。其实这就是 SpaceSaving 算法论文中所介绍的流程。

```
        if x in self.counters:
            self.counters[x] += 1
        else:
            if len(self.counters) + 1 > self.k:
                min_counter = min(self.counters, key=self.counters.get)
                self.counters[x] = self.counters.pop(min_counter) + 1
            else:
                self.counters[x] = 1
```

确实，就如此简洁明了！我觉得 SpaceSaving 真的是一个优雅的算法。

3.3.3 Count-Min-Log Sketch 算法框架

```
from array import array
from math import log, e, ceil, pow
import hashlib
import random


class CountMinLogSketch(object):
    def __init__(self, w=None, d=None, delta=None, epsilon=None,
exp=1.00026):
        if w is not None and d is not None:
            self.w = w
            self.d = d
        elif delta is not None and epsilon is not None:
            self.w = int(ceil(e / epsilon))
            self.d = int(ceil(log(1. / delta)))
        else:
            raise Exception("Incomplete parameters. Please provide w&d or
delta&epsilon.")
```

```
        self.exp = exp
        self.table = [array('l', (0 for _ in range(self.w))) for _ in
range(self.d)]

    def increaseDecision(self, c):
        return random.random() * pow(self.exp, float(c)) < 1

    def pointValue(self, c):
        if c == 0:
            return 0
        return pow(self.exp, float(c-1))

    def value(self, c):
        if c <= 1:
            return self.pointValue(c)
        else:
            v = self.pointValue(c + 1)
            return (1 - v) / (1 - self.exp)
    def _hash(self, x):
        md5 = hashlib.md5(str(hash(x)).encode())
        for i in range(self.d):
            md5.update(str(i).encode())
            yield int(md5.hexdigest(), 16) % self.w

    def add(self, x, v=1):
        # 元素 x 出现了 v 次
        for _ in range(1, v+1):
            c = min(table[i] for table, i in zip(self.table, self._hash(x)))
            if self.increaseDecision(c):
                for table, i in zip(self.table, self._hash(x)):
                    if table[i] == c:
                        table[i] += 1

    def query(self, x):
        # 元素 x 的估计出现次数
        c = min(table[i] for table, i in zip(self.table, self._hash(x)))
        return int(self.value(c))

    def __getitem__(self, x):
        return self.query(x)
```

其实从本质上来说 CMLS 算法和 CMS 算法还是很类似的，只不过依据论文介绍的算法，加入了 increaseDecision 这样的模块。

3.3.4 主程序框架

我个人认为，这个实验只用做一个流的频繁项挖掘工作即可，并不需要考虑多线程、分布式这样的情况。客观来说本次实验的 2.5kw 数据量很庞大，而且依据时间的分布也较为均匀，如果适当地划分为多块，然后对每一块数据进行单流频繁项挖掘、快与块之间的结果具有单调性和可加性这样优良的性质，我完全可以通过 MapReduce 编程框架进行分布式处理，但苦于自己资源有限，无法租赁过多的云实例，所以这个计划并没有落实。最终，我只实现了对于所有的 2.5kw 条数据单流频繁数据项挖掘。

值得注意的是，为了模拟数据以数据流的形式随时间增量式获取，也为了减少数据集中无关列（userID,rating 这两列和本次实验无关）从而减少空间占用、优化内存利用情况，我首先进行的操作便是读取与筛选(utils/dataparsing.py)：

```
import pandas as pd
import os

if __name__ == '__main__':
    print(os.getcwd())
    df1 = pd.read_csv('../data/ratings.csv')
    df1.sort_values(by='timestamp', inplace=True)
    df1 = df1[['timestamp', 'movieId']]
    df1.to_csv('../data/data.csv', index=False)
```

以下内容主要为了方便调试，只取了 100 条的小数据集：

```
    df2 = pd.read_csv('../data/data.csv')
    data_mini = df2.head(100)
    data_mini.to_csv('../data/data_mini.csv', index=False)
```

在对数据集做出初步处理之后，我的正式实验便开始了，为了方便调试，我使用了 argparse 包，并将很多参数给形式化了：

```
…
import argparse
…
parser = argparse.ArgumentParser()
parser.add_argument('--k', default=10, type=int, help='Max query k value
in Top-K')
parser.add_argument('--p', default=1e-2, type=float, help='query
probability')
parser.add_argument('--filepath', default='data/data.csv', type=str,
help='input file path')
parser.add_argument('--delta', default=1e-6, type=float, help='delta')
parser.add_argument('--eps', default=(math.e / 1000), type=float,
help='epsilon')
parser.add_argument('--ssk', default=10000, type=int, help='k value of
SpaceSaving')
parser.add_argument('--is_conservative_update', default=False,
type=bool, help='Conservative Update')
parser.add_argument('--logexp', default=1.00026, type=float,
```

```
help='CountMinLogSketch exp value')
parser.add_argument('--head', default=1000000, type=int, help='how much
lines do you want to read from the file?')
args = parser.parse_args()
```

与之相关的数据结构及变量初始化如下所示：

```
from model.CountMinSketch import CountMinSketch
from model.MinHeap import MinHeap
from model.SpaceSaving import SpaceSaving
from model.CountMinLogSketch import CountMinLogSketch
import math as math
import pandas as pd
import random

df = pd.read_csv(args.filepath)
if args.head != 0:
    df = df.head(args.head)

ans_count = {}  # 用于记录各电影在此之前真实出现的次数
cms = CountMinSketch(delta=args.delta, epsilon=args.eps,
is_conservative_update=args.is_conservative_update)
cmls = CountMinLogSketch(delta=args.delta, epsilon=args.eps,
exp=args.logexp)
ss = SpaceSaving(args.ssk)
hp = MinHeap(args.k)
hp_log = MinHeap(args.k)

is_query = False
k = args.k
query_times12, query_times3 = 0, 0
sum_error_task1, sum_error_task2, sum_error_task3 = 0, 0, 0
sum_error_task1_log, sum_error_task2_log, sum_error_task3_log = 0, 0, 0
sum_error_task1_ss, sum_error_task2_ss, sum_error_task3_ss = 0, 0, 0
```

因为我们是按照时间戳为单位进行处理的，所以理论上来说一条一条去读、去处理是不科学的。首先我要将数据集合根据 timestamp 分组聚集：

```
for index, (timestamp, group_data) in enumerate(df.groupby('timestamp')):
```

而且，我为了模拟查询，想法是这样的：

1. 首先我不能有多少个时间戳就进行多少次查询（这样消耗的时间太大了，也没有必要，与事实情况也不符合，我们在现实中往往只在一些时间有查询需求），所以我选择随机抽了一些时间戳作为需要查询的样本;

2.接下来就是我们怎样“随机”抽取一些时间戳作为进行查询的点呢？如果要使用分组抽样、等距抽样等方法似乎并不能保证“随机”，因为我们不能预先知道数据流的大小。如果使用水库抽样，这样只能保证在一个很大的数据流扫了一遍之后得到一个固定大小的样本集（水库），但是我们在水库最终形成之前压根不能确定一个样本到底是不是最终还能保留在水库中，既然都不确定，那么我们就

无法肯定的进行查询操作，但是数据流事实情况来看只能扫描一次，所以第二遍扫描其实是不符合现实情况的.但是我们如果让每一个时间戳独立同分布——就是说它们彼此之间是不是要查询是独立的，每个点都有 p 的概率被抽到要查询，这样理论来说是符合要求的;
3.公平起见，如果某个时间戳被抽中要去做查询操作了，我将 task1 到 3 都运行了一遍，而不是只在这个 task 做某一个任务；
4.结合课本上关于 Misra-Gries 算法是否替换这个过程中随机数的利用，我的结构大概是这样的：

```
for index, (timestamp, group_data) in enumerate(df.groupby('timestamp')):
    is_query = False
    # print("Timestamp:", timestamp)
    random_num = random.randint(1, 1000000)
    if random_num <= int(1000000 * args.p):
        is_query = True
……
if is_query:
    for _, row in group_data.iterrows():
        query_times12 += 1
……
```

生成某部电影到底被评分了多少次其实维护一个字典 ans_count 即可。对于每一步到来的电影评价信息，我们都要将其加入对应的 CMS,CMLS,SS 结构中：

```
for index, (timestamp, group_data) in enumerate(df.groupby('timestamp')):
    is_query = False
……
    for _, row in group_data.iterrows():
        movie_id = row['movieId']
        cms.add(movie_id)
        cmls.add(movie_id)
        ss.add(movie_id)
        ans_count[movie_id] = ans_count.get(movie_id, 0) + 1
```

其中 CMS 和 CMLS 维护 Top-k 问答要依赖一个小根堆，根据老师的意见，我没有使用 heapq 这样的内置库，而是自己实现了一个可以存储二元组的小根堆（排序依据是各个二元组的第二个元素），小根堆的代码见 model/MinHeap.py. 结合我在实验报告 3.2 节的论述，我以 CMS 算法为例来展现如何维护堆这个结构：

```
if movie_id in hp.index_map:
    hp.insert([movie_id, 1])
else:
    frequent_hat = cms.query(movie_id)
    if len(hp.heap) < args.k:
        hp.insert([movie_id, frequent_hat])
    elif frequent_hat > hp.getmin():
        root = hp.heap[0]
        hp.heap[0] = [movie_id, frequent_hat]
```

```
            hp.index_map[movie_id] = 0
            hp.adjust_down(0)
            del hp.index_map[root[0]]
```

至于被抽到要进行查询、算法准确率衡量的代码，我在此不赘述了，代码如下所示，其实注释我觉得写得还算明白。不过值得强调的一点是 Top-k 的 k 值我认为每次不一样才更加符合实际情况，而且为了表述方便我在程序运行前便确定了 k 的最大值 k',然后每次随机地在[1, k']这个区间内抽一个整数作为 k 值即可。

```
if is_query:
    for _, row in group_data.iterrows():
        query_times12 += 1
        movie_id = row['movieId']
        frequent_hat = cms.query(movie_id)
        frequent_hat_log = cmls.query(movie_id)
        if movie_id in ss.counters:
            frequent_hat_ss = ss.counters[movie_id]
        else:
            frequent_hat_ss = 0

        # task1: 成员查询：对给定查询时间戳和电影 id，查询该电影在该时间戳及之前是否曾被评分过
        if frequent_hat > 0 and ans_count[movie_id] == 0:
            # 此时算法认为该电影在该时间戳及之前被评分过
            # 并且假设该电影此前并没有被评分过(预测失败)
            sum_error_task1 += 1

        # 根据 CMS 算法的原理，不可能出现算法认为该电影在该时间戳及之前未被评分过但实际被评分过的情况

        if frequent_hat_log > 0 and ans_count[movie_id] == 0:
            sum_error_task1_log += 1

        if frequent_hat_ss > 0 and ans_count[movie_id] == 0:
            sum_error_task1_ss += 1

        # task2: 频度查询：对于给定查询时间戳和电影 id，查询该电影在该时间戳及之前被评分的总次数
        sum_error_task2 += (frequent_hat - ans_count[movie_id]) ** 2
        sum_error_task2_log += (frequent_hat_log - ans_count[movie_id]) ** 2
        sum_error_task2_ss += (frequent_hat_ss - ans_count[movie_id]) ** 2

    # task3: Top-k 查询：对给定正整数 k 和查询时间戳，查询在该时间戳及之前被评分次数最多的前 k 个电影。
    query_times3 += 1
```

```
    sum_error_task3_tmp = 0
    sum_error_task3_tmp_log = 0
    sum_error_task3_tmp_ss = 0

    k = random.randint(1, args.k)

    topk_ans_true = sorted(ans_count.items(), key=lambda x: x[1],
reverse=True)[:k]
    topk_ans_true = [[x[0], x[1]] for x in topk_ans_true]

    topk_ans_hat = sorted(hp.heap, key=lambda x: x[1], reverse=True)[:k]
    topk_ans_hat_log = sorted(hp_log.heap, key=lambda x: x[1], reverse=True)[:k]

    topk_ans_hat_ss = sorted(ss.counters.items(), key=lambda x: x[1],
reverse=True)[:k]
    topk_ans_hat_ss = [[x[0], x[1]] for x in topk_ans_hat_ss]

    for p, q in zip(topk_ans_true, topk_ans_hat):
        sum_error_task3_tmp += (p[1] - q[1]) ** 2

    for p, q in zip(topk_ans_true, topk_ans_hat_log):
        sum_error_task3_tmp_log += (p[1] - q[1]) ** 2

    for p, q in zip(topk_ans_true, topk_ans_hat_ss):
        sum_error_task3_tmp_ss += (p[1] - q[1]) ** 2

    sum_error_task3_tmp = sum_error_task3_tmp / k
    sum_error_task3_tmp_log = sum_error_task3_tmp_log / k
    sum_error_task3_tmp_ss = sum_error_task3_tmp_ss / k

    sum_error_task3 += sum_error_task3_tmp
    sum_error_task3_log += sum_error_task3_tmp_log
    sum_error_task3_ss += sum_error_task3_tmp_ss
```

四、 实验结果（验证提出方法的有效性和高效性）

首先来说一下各结构的构建与更新时间，因为我的数据是源源不断输入进去的，这就意味着这些结构的更新过程也是动态的，我统计了一下，发现更新一个元素的时间在 1s 左右，而且随着 sketch 逐步被填充有扩大趋势，但不超过 3s。构建时间更快了，整体来说基本都在 1s 之内。任务的精确度我主要通过 MSE 这个指标来衡量，而且为了避免很多高频项频数是一样的但是预测和实际差距很大导致 MSE 较大这样的情况，所以在 task3 中只要前 k 名频数是一样的我就认为对 MSE 的增长无影响。

至于空间占用情况，其实主要还应该结合前面理论部分的讲解，我就以 CMS

和 SpaceSaving 这两个最经典的例子来介绍吧。前者的空间复杂度：

$$w = O(\frac{1}{\epsilon}), d = O(ln(\frac{1}{\delta}))空间复杂度:O(\frac{1}{\epsilon}ln(\frac{1}{\delta}))$$

后者的空间复杂度就是 O(k),这里的 k 值就是 SpaceSaving 结构的超参数,k=O(1/ε)。

Top-k 查询中的 k 最大值被我设置为 25，抽取概率 p=1e-3(每个时间戳有 0.05%的概率被抽)而且我读取了所有的 2500w 条数据。

本来我是打算读取所有的 2500w 条数据的，但是在 delta=1e-6, epsilon=(math.e / 1000) ,没有开启 conservative_update 情况下,k value of SpaceSaving 为 10000 时到 1360185605 这个时间戳（数据集按时间戳排序的第 16505242 条数据）之后就内存溢出了（自己的电脑配置比较差）：

```
timestamp:1360185605,query_times:12500
CMS:
task1 MSE:0.0 task2 MSE:2189122.291686893 task3 MSE:26890.924443820804
CMLS:
task1 MSE:0.0 task2 MSE:199466.25898058253 task3 MSE:6362.814848999
SpaceSaving:
task1 MSE:0.0 task2 MSE:50.9818567961165 task3 MSE:0.0
Traceback (most recent call last):
  File "D:\dase.algo\project\1\main.py", line 43, in <module>
    for index, (timestamp, group_data) in enumerate(df.groupby('timestamp')):
  File "C:\Users\tom\AppData\Roaming\Python\Python310\site-packages\pandas\core\groupby\ops.py", line 727, in get_iterator
    yield from zip(keys, splitter)
  File "C:\Users\tom\AppData\Roaming\Python\Python310\site-packages\pandas\core\groupby\ops.py", line 1239, in __iter__
    yield self._chop(sdata, slice(start, end))
  File "C:\Users\tom\AppData\Roaming\Python\Python310\site-packages\pandas\core\groupby\ops.py", line 1264, in _chop
    mgr = sdata._mgr.get_slice(slice_obj, axis=1 - self.axis)
  File "pandas\_libs\internals.pyx", line 861, in pandas._libs.internals.BlockManager.get_slice
  File "pandas\_libs\internals.pyx", line 842, in pandas._libs.internals.BlockManager._get_index_slice
  File "pandas\_libs\internals.pyx", line 654, in pandas._libs.internals.NumpyBlock.getitem_block_index
  File "pandas\_libs\internals.pyx", line 661, in pandas._libs.internals.NumpyBlock.getitem_block_index
  File "pandas\_libs\internals.pyx", line 616, in pandas._libs.internals.SharedBlock.__cinit__
  File "pandas\_libs\internals.pyx", line 894, in pandas._libs.internals.BlockValuesRefs.add_reference
MemoryError
```

在不减少参数的前提下，就让我浅浅来分析一下跑出来的过程中的一些结果吧，因为原理上其实是相通的。

```
timestamp:965326248,query_times:1900
CMS:
task1 MSE:0.0 task2 MSE:11230.594171483623 task3 MSE:2.190459649122806
CMLS:
task1 MSE:0.0 task2 MSE:4048.8612716763005 task3 MSE:664.7162761474691
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:968418851,query_times:2000
CMS:
task1 MSE:0.0 task2 MSE:15089.143540669857 task3 MSE:2.69403854037267
CMLS:
task1 MSE:0.0 task2 MSE:4783.532695374801 task3 MSE:753.3927350704646
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:974655791,query_times:2100
CMS:
task1 MSE:0.0 task2 MSE:19209.755965292843 task3 MSE:4.110208434388247
CMLS:
task1 MSE:0.0 task2 MSE:4806.863557483731 task3 MSE:845.9845020466577
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
```

```
timestamp:1100007642,query_times:4900
CMS:
task1 MSE:0.0 task2 MSE:303149.8187615955 task3 MSE:440.59077860905313
CMLS:
task1 MSE:0.0 task2 MSE:34752.57838589981 task3 MSE:3380.5914716712837
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:1103744882,query_times:5000
CMS:
task1 MSE:0.0 task2 MSE:316858.61925501435 task3 MSE:545.4784752042028
CMLS:
task1 MSE:0.0 task2 MSE:36095.969512893986 task3 MSE:3458.628074974747
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:1106245041,query_times:5100
CMS:
task1 MSE:0.0 task2 MSE:330602.81939723546 task3 MSE:628.0805585140275
CMLS:
task1 MSE:0.0 task2 MSE:37149.58984817584 task3 MSE:3506.4767715907733
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0

timestamp:1244395778,query_times:9900
CMS:
task1 MSE:0.0 task2 MSE:1309766.6570953277 task3 MSE:7080.528423728515
CMLS:
task1 MSE:0.0 task2 MSE:115150.2461304788 task3 MSE:5142.501414682809
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.23253290901200638 task3 MSE:0.0
timestamp:1249069496,query_times:10000
CMS:
task1 MSE:0.0 task2 MSE:1333231.2774466863 task3 MSE:7159.459428994091
CMLS:
task1 MSE:0.0 task2 MSE:118032.77303080348 task3 MSE:5177.513210172862
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.5586989301357076 task3 MSE:0.0
timestamp:1253148792,query_times:10100
CMS:
task1 MSE:0.0 task2 MSE:1357588.3371114912 task3 MSE:7268.775583115231
CMLS:
task1 MSE:0.0 task2 MSE:120079.18648417451 task3 MSE:5215.866370661442
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.8721842030225264 task3 MSE:0.0
```

那如果我加入 conservative_update 机制来优化 CMS 成为 CUS 呢？运行过程中的几个点的结果如下：

```
timestamp:962985050,query_times:1900
CMS:
task1 MSE:0.0 task2 MSE:523.3463473053893 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:3837.7552095808383 task3 MSE:1021.1789687777633
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:966863724,query_times:2000
CMS:
task1 MSE:0.0 task2 MSE:778.543645409317 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:4089.5305291723203 task3 MSE:1140.7597522265294
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:974606379,query_times:2100
CMS:
task1 MSE:0.0 task2 MSE:1005.0919340849956 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:4395.21487424111 task3 MSE:1287.343615148861
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0

timestamp:1100107705,query_times:4900
CMS:
task1 MSE:0.0 task2 MSE:23664.523235397402 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:30265.431113192088 task3 MSE:3178.6451875600915
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:1104260610,query_times:5000
CMS:
task1 MSE:0.0 task2 MSE:25010.8692734845 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:31712.878875520593 task3 MSE:3246.9006366396534
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:1106394737,query_times:5100
CMS:
task1 MSE:0.0 task2 MSE:26959.651194695325 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:33748.800160054874 task3 MSE:3322.310481434972
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
```

如果我调整 CMS 的超参数 delta, epsilon 呢，实验的结果如何呢？因为自己之前设置的都比较大（delta=1e-6, epsilon=(math.e / 1000)），所以我以下实验都是往小了调。

delta=1e-6, epsilon=(math.e / 100)时：

```
timestamp:833899449,query_times:100
CMS:
task1 MSE:0.0 task2 MSE:1419.0049261083743 task3 MSE:2.1815530355161057
CMLS:
task1 MSE:0.0 task2 MSE:1691.64039408867 task3 MSE:2.354148022905961
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:835973568,query_times:200
CMS:
task1 MSE:0.0 task2 MSE:5308.031941031941 task3 MSE:8.526351270490322
CMLS:
task1 MSE:0.0 task2 MSE:5852.22113022113 task3 MSE:28.497941499490953
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
```

可以看到最开始的阶段错误就十分显著了

delta=1e-4, epsilon=(math.e / 1000)时：

```
timestamp:867319471,query_times:900
CMS:
task1 MSE:0.0 task2 MSE:0.6916183447548762 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:3437.333684765419 task3 MSE:176.26704474269619
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:892908580,query_times:1000
CMS:
task1 MSE:0.0 task2 MSE:0.625772705658583 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:3416.0603899191633 task3 MSE:199.47441092613965
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
timestamp:916626828,query_times:1100
CMS:
task1 MSE:0.0 task2 MSE:79.82357966680905 task3 MSE:0.0
CMLS:
task1 MSE:0.0 task2 MSE:3467.089705254165 task3 MSE:217.7768505922679
SpaceSaving:
task1 MSE:0.0 task2 MSE:0.0 task3 MSE:0.0
```

通过观察，我们可以发现如下规律：

1. 在数据流输入的比较小时候，三种算法的表现都很不错，但是随着输入数据的增多，三种算法都表现了不同程度的表现恶化。其中 CMLS 算法由于浮点数精度和随机性模拟等问题，从最开始就有积累偏差。初期 CMLS 算法的表现要明显劣于其他两种算法，尤其是 task2 任务上;但是随着时间推移、数据流经量的扩大，CMLS 和 CMS 算法有一个比较独特的“反超”现象——即 CMLS 的准确性表现渐渐地要好于 CMS，CMS 在后期成为了准确率表现最差的算法；
2. SpaceSaving 算法即便也有准确率表现退化，但是并不明显，整体来说表现最好，效果最为稳定，而且时间复杂度明显比 CMLS 和 CMS 算法要好。
3. 加入 conservative update 优化的 CMS 算法可以显著增大频数的估计准确率，进而提升准确率，可以看到确实是有明显效果的。
4. 缩小 delta, epsilon 这两个参数可以降低程序的运行时间与内存消耗，但是会显著增大估计值的误差，降低算法的准确率。

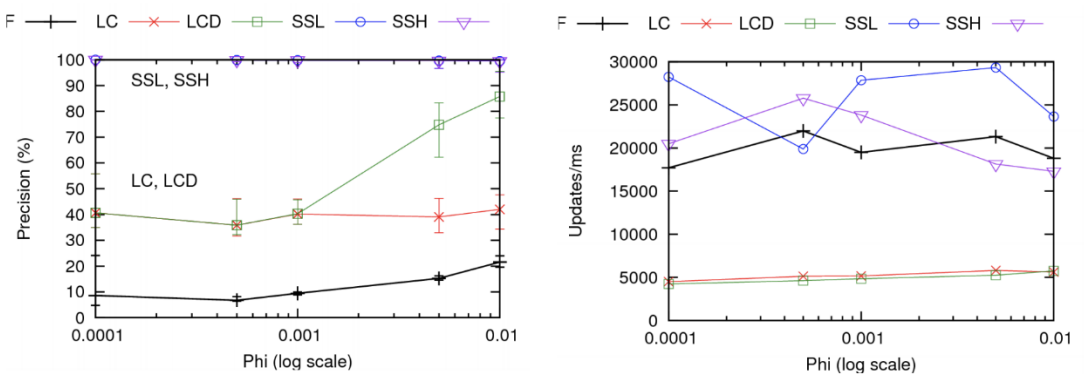
这些实验现象与我们对理论的了解是吻合的，总体来说实际结果印证了理论知识。通过具体的分析，我们能看到每次查询过程中关于小根堆、排序是时间消耗的决定性因素，更新 CMS,CMLS,SS 数据结构所消耗的时间并不是决定性的。在实验过程中，其实我们的空间复杂度都是可以接受的，都满足流场景下对算法的要求，可以在有限的内存空间中处理大规模的数据流。

五、 结论（对使用的方法可能存在的不足进行分析，以及未来可能的研究方向进行讨论）

在本项目中，我成功地设计并实现了一个空间占用尽量小的数据结构，用于处理带有时间戳的用户对电影的评分信息。通过对比 Count-Min Sketch+MinHeap、Count-Min Sketch(conservative update) +MinHeap、Count-Min Log Sketch+MinHeap 和 SpaceSaving 这四种结构，我发现 SpaceSaving 算法不仅代码实现容易，而且空间复杂度最低，甚至整体的准确性表现最佳。以 SpaceSaving 算法为核心的一整套结构足以满足成员查询、频度查询和 Top-k 查询等功能。通过给定数据集的实验，我发现理论和实际吻合程度较好，也感受到了这些算法在处理大规模数据流时具有很高的效率和准确性。

实验结果显示，这些算法也有一些局限性。首先，它们都假设数据流是静态的，即数据流的分布不会随时间变化。如果数据流的分布发生了变化，这些算法的性能可能会下降。其次，这些算法都需要预先设定一些参数，如哈希函数的数量和数组的大小，这些参数的设定可能会影响算法时间和空间上的性能。在未来的研究中，我计划探索更多的数据流统计算法，如 Sketching、Sampling 和 Histogram 等，以提高查询的精度和效率，还计划研究如何动态地调整算法的参数，以适应数据流的变化。

如果要进一步延申，Space Saving 算法在贴近实际应用的 Zipfian 数据集上的 benchmark 如下图所示，可见与其他算法相比，无论在准确率方面还是效率方面都几乎是最优的。



在大数据相关的组件中，Space Saving 算法应用有很多。比如说在 Apache Kylin 中的 Top-N 近似预计算特性；又比如 ClickHouse 函数库中的 anyHeavy()函数，它能够返回数据集中任意一个频繁项。特别地，它们使用的都是并行化的 Space Saving 算法，能够显著提升多线程环境下的计算效率。