

第一章 绪论

背景:

基数(cardinality, 也译作势), 是指一个集合中不重复元素的个数。这里的集合和我们学过的严格定义的集合不同, 允许存在重复元素, 被称作多重集合(multi-set), 如给定这样的一个集合 $\{1, 2, 3, 1, 2\}$, 它有 5 个元素, 基数是 3。

基数计数(cardinality counting), 则是指计算一个集合的基数。基数计算的场景很广泛, 例如计算网站的访问 uv, 计算网络流量网络包请求 header 中的源地址的 distinct 数来作为网络攻击的重要指标。要实现基数计数最直接想到的方式就是通过字典/HashSet, 每条数据流入后直接保存相应的 key, 最后统一集合的 size 就得到集合的基数。但是, 这种方法的空间复杂度很高, 在面对大数据的场景下做这样的统计代价很高。在近几十年有学者提出了很多基数估算的算法, 在容许一定的误差的情况下, 基于统计概率进行估算, 本文就来介绍其中比较有名的几个基数估计算法。

挑战:

在面对大数据场景下, 我们使用近似的算法来估计集合的基数, 我们需要考虑一下内容:

1. 如何在有效的时间复杂度内得出结果;
2. 如何在有效的空间复杂度内得出结果;
3. 在时空复杂度有效的前提下, 如何节省时间和空间;
4. 如何降低集合估计算法的错误率, 保证结果在可控的范围内;

以上这些内容是我们必须要考虑到的, 但是大数据场景下的基数估计问题所面临的挑战远不止这些。

本文介绍的解决方法:

本文给出三种多重集合基数估计算法:

1. 使用 HyperLogLog 进行基数估计;
2. 使用 BloomFilter (布隆过滤器) 进行基数估计;
3. 使用水库抽样算法来进行基数估计;

本文的贡献:

相较于已有的 HyperLogLog 和 BloomFilter, 本文提出了一种基于水库抽样算法的基数估计算法, 具体来说就是通过水库算法来抽样, 然后根据样本特性来估计整体集合的基数。但是就目前的情况来看, 这种方法得到的结果存在较大的误差。不过, 本文提出的方法也是一种思路, 我相信只要用心去优化, 一定可以将误差降低到可以接受的范围内。

章节安排:

第一章 绪论

第二章 基于 HyperLogLog 的基数估计算法介绍

第三章 基于 BloomFilter 的基数估计算法介绍

第四章 基于水库抽样算法的技术估计算法介绍

第五章 实验

第六章 结论

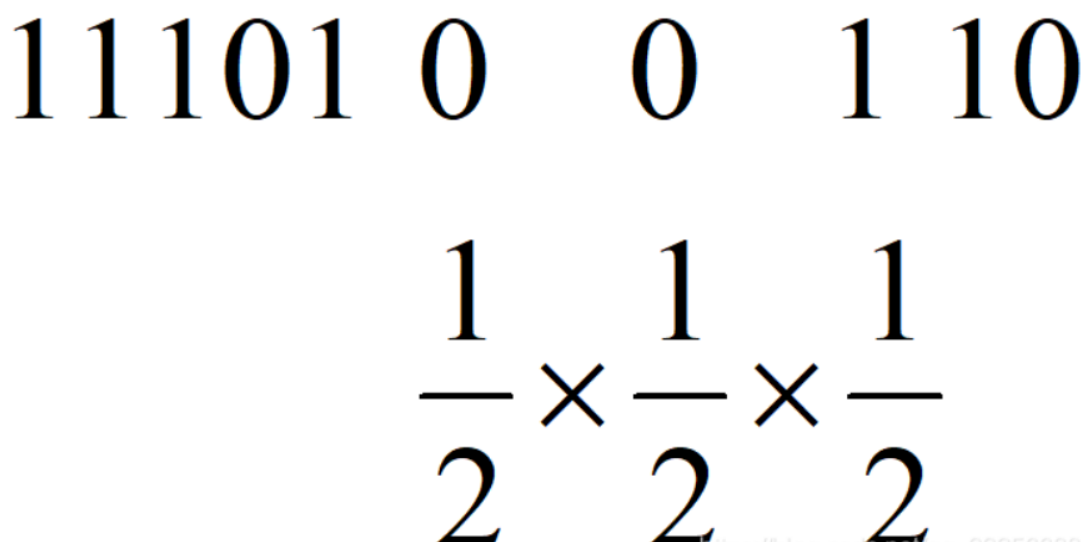
参考文献

附录

第二章 基于 HyperLogLog 的基数估计算法介绍

基本思想：

HyperLogLog 本质来源于生活中一个小的发现，假设你抛了很多次硬币，你告诉在这次抛硬币的过程中最多只有两次扔出连续的反面，让我猜你总共抛了多少次硬币，我敢打赌你抛硬币的总次数不会太多，相反，如果你和我说最多出现了 100 次连续的反面，那么我敢肯定扔硬币的总次数非常的多，甚至我还可以给出一个估计，这个估计要怎么给呢？其实是一个很简单的概率问题，假设 1 代表抛出正面，0 代表反面：



The diagram illustrates the HyperLogLog estimation process. It shows a coin toss sequence "11101 0 0 1 10" and the calculation of the probability of a run of three zeros as $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2}$.

图 2-1 抛硬币举例

上图中以抛硬币序列“1110100110”为例，其中最长的反面序列是“00”，我们顺手把后面那个 1 也给带上，也就是“001”，因为它包括了序列中最长的一串 0，所以在序列中肯定只出现过一次，而它在任意序列出现出现且仅出现一次的概率显然是上图所示的三个二分之一相乘，也就是八分之一，所以我可以给出一个估计值，你大概总共抛了 8 次硬币。

很显然，上面这种做法虽然能够估计抛硬币的总数，但是显然误差是比较大的，很容易受到突发事件（比如突然连续抛出好多 0）的影响，HyperLogLog 算法研究的就是如何减小这个误差。

之前说过，HyperLogLog 算法是用来计算基数的，这个抛硬币的序列和基数有什么关系呢？比如在数据库中，我只要在每次插入一条新的记录时，计算这条记录的 hash，并且转换成二进制，就可以将其看成一个硬币序列了，如下 (0b 前缀表示二进制数)：

$$\text{hash}(\text{record}) = 0b1110100110$$

图 2-2 hash 举例

根据上面抛硬币的启发我可以想到如下的估计基数的算法（伪代码描述）：

输入：一个集合

输出：集合的基数

算法：

`max = 0`

对于集合中的每个元素：

`hashCode = hash(元素)`

`num = hashCode` 二进制表示中最前面连续的 0 的数量

if `num > max`:

`max = num`

最后的结果是 2 的 $(\text{max} + 1)$ 次幂

举个例子，对于集合 {ele1, ele2}，先求 $\text{hash}(\text{ele1}) = 0b00110111$ ，它最前面的连续的 0 的数量为 2（又称为前导 0），然后求 $\text{hash}(\text{ele2}) = 0b10010000111$ ，它的前导 0 数量为 0，我们始终只保存前导零数量的最大值，所以最后 max 是 2，我们估计的基数就是 2 的 $(2+1)$ 次幂，即 8。

为什么最后的 max 要加 1 呢？这是一个数学细节，具体要看论文，简单的理解的话，可以像之前抛硬币的例子那样理解，把最长的一串零的后面的一个 1 或者前面的一个 1“顺手”带上进行概率估计。

显然这个算法是非常不准确的，但是这个想法还是很有启发性的，从这个简单的想法跟随下文一步一步优化即可得到最终的高精度的 HyperLogLog 算法。

分桶：

最简单的一种优化方法显然就是把数据分成 m 个均等的部分，分别估计其总数求平均后再乘以 m ，称之为分桶。对应到前面抛硬币的例子，其实就是把硬币序列分成 m 个均等的部分，分别用之前提到的那个方法估计总数求平均后再乘以

m，这样就能一定程度上避免单一突发事件造成的误差。

具体要怎么分桶呢？我们可以将每个元素的 hash 值的二进制表示的前几位用来指示数据属于哪个桶，然后把剩下的部分再按照之前最简单的想法处理。

还是以刚刚的那个集合 {ele1, ele2} 为例，假设我要分 2 个桶，那么我只要去 ele1 的 hash 值的第一位来确定其分桶即可，之后用剩下的部分进行前导零的计算，如下图：

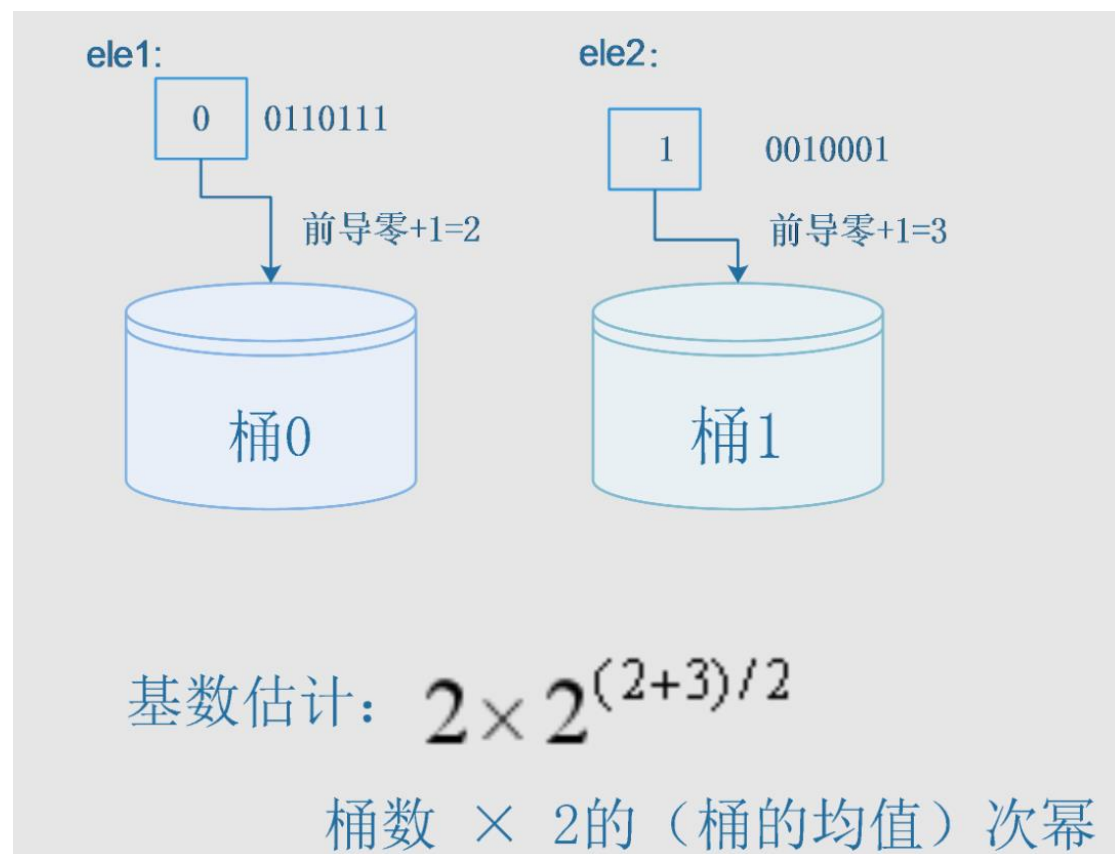


图 2-3 分桶举例

到这里，其实我们介绍了 LogLog 算法的基本思想，LogLog 算法是在 HyperLogLog 算法之前提出的一个基数估计算法，HyperLogLog 算法其实就是 LogLog 算法的一个改进版。LogLog 算法完整的基数计算公式如下：

$$DV_{LL} = \text{constant} * m * 2^{\overline{R}}$$

图 2-4 LogLog 算法公式

其中 m 代表分桶数，R 头上一道横杠的记号就代表每个桶的结果（其实就是桶中数据的最长前导零+1）的均值，相比我之前举的简单的例子，LogLog 算法

还乘了一个常数 **constant** 进行修正，这个 **constant** 下文具体介绍。

调和平均数：

前面的 LogLog 算法中我们是使用的是平均数来将每个桶的结果汇总起来，但是平均数有一个广为人知的缺点，就是容易受到大的数值的影响，一个常见的例子是，假如我的工资是 1000 元一个月，我老板的工资是 100000 元一个月，那么我和老板的平均工资就是 $(100000 + 1000)/2$ ，即 50500 元，显然这离我的工资相差甚远，我肯定不服这个平均工资。

用调和平均数就可以解决这一问题，调和平均数的结果会倾向于集合中比较小的数， x_1 到 x_n 的调和平均数的公式如下：

$$H_n = \frac{1}{\frac{1}{n} \sum_{i=1}^n \frac{1}{x_i}} = \frac{n}{\sum_{i=1}^n \frac{1}{x_i}}$$

图 2-5 调和平均数公式

再回到前面的 LogLog 算法，从前面的举的例子可以看出，影响 LogLog 算法精度的一个重要因素就是，hash 值的前导零的数量显然是有很大的偶然性的，经常会出现一两数据前导零的数目比较多的情况，所以 HyperLogLog 算法相比 LogLog 算法一个重要的改进就是使用调和平均数而不是平均数来聚合每个桶中的结果，HyperLogLog 算法的公式如下：

$$DV_{HLL} = \text{constant} * m^2 * \left(\sum_{j=1}^m 2^{-R_j} \right)^{-1}$$

图 2-6 HyperLogLog 算法公式

其中 **constant** 常数和 **m** 的含义和之前的 LogLog 算法公式中的含义一致，**R_j** 代表(第 **j** 个桶中的数据的最小前导零数目+1)，为了方便理解，我将公式再拆解一下：

$$DV_{HLL} = const * m * \left(\frac{m}{\sum_{j=1}^m \frac{1}{2^{R_j}}} \right)$$

每个桶的估计值

所有桶估计值的调和平均数

图 2-7 HyperLogLog 算法公式

constant 常数的选择与分桶的数目有关，具体的数学证明请看论文，这里就直接给出结论：假设 m 为分桶数， p 是 m 的以 2 为底的对数，则：

```
switch (p) {
  case 4:
    constant = 0.673 * m * m;
  case 5:
    constant = 0.697 * m * m;
  case 6:
    constant = 0.709 * m * m;
  default:
    constant = (0.7213 / (1 + 1.079 / m)) * m * m;
}
```

图 2-8 constant 常数的计算

如果理解了之前的分桶算法，那么很显然分桶数只能是 2 的整数次幂。如果分桶越多，那么估计的精度就会越高，统计学上用来衡量估计精度的一个指标是“相对标准误差” (relative standard deviation, 简称 RSD), RSD 的计算公式这里就不给出了，百科上一搜就可以知道，从直观上理解，RSD 的值其实就是 ((每次估计的值) 在 (估计均值) 上下的波动) 占 (估计均值) 的比例。RSD 的值与分桶数 m 存在如下的计算关系：

$$RSD = \frac{1.04}{\sqrt{m}}$$

图 2-9 RSD 的计算

至此，我们介绍了利用 HyperLogLog 进行基数估计的原理和一些重要的细节，具体的证明过程请参阅论文。

第三章 基于 BloomFilter 的基数估计算法介绍

介绍:

布隆过滤器 (Bloom Filter) 是一个叫做 Bloom 的人于 1970 年提出的。我们可以把它看作由二进制向量 (或者说位数组) 和一系列随机映射函数 (哈希函数) 两部分组成的数据结构。相比于我们平时常用的 List、Map、Set 等数据结构, 它占用空间更少并且效率更高, 但是缺点是其返回的结果是概率性的, 而不是非常准确的。理论情况下添加到集合中的元素越多, 误报的可能性就越大。并且, 存放在布隆过滤器的数据不容易删除。

概括地说, 布隆过滤器可以告诉我们“某样东西一定不存在或者可能存在”, 也就是说布隆过滤器说这个数不存在则一定不存在, 布隆过滤器说这个数存在但是却可能不存在。

原理介绍:

当一个元素加入布隆过滤器中的时候, 会进行如下操作:

1. 使用布隆过滤器中的哈希函数对元素值进行计算, 得到哈希值 (有几个哈希函数得到几个哈希值)。
2. 根据得到的哈希值, 在位数组中把对应下标的值置为 1。

当我们需要判断一个元素是否存在于布隆过滤器的时候, 会进行如下操作:

1. 对给定元素再次进行相同的哈希计算。
2. 得到值之后判断位数组中的每个元素是否都为 1, 如果值都为 1, 那么说明这个值在布隆过滤器中, 如果存在一个值不为 1, 说明该元素不在布隆过滤器中。

举个例子:

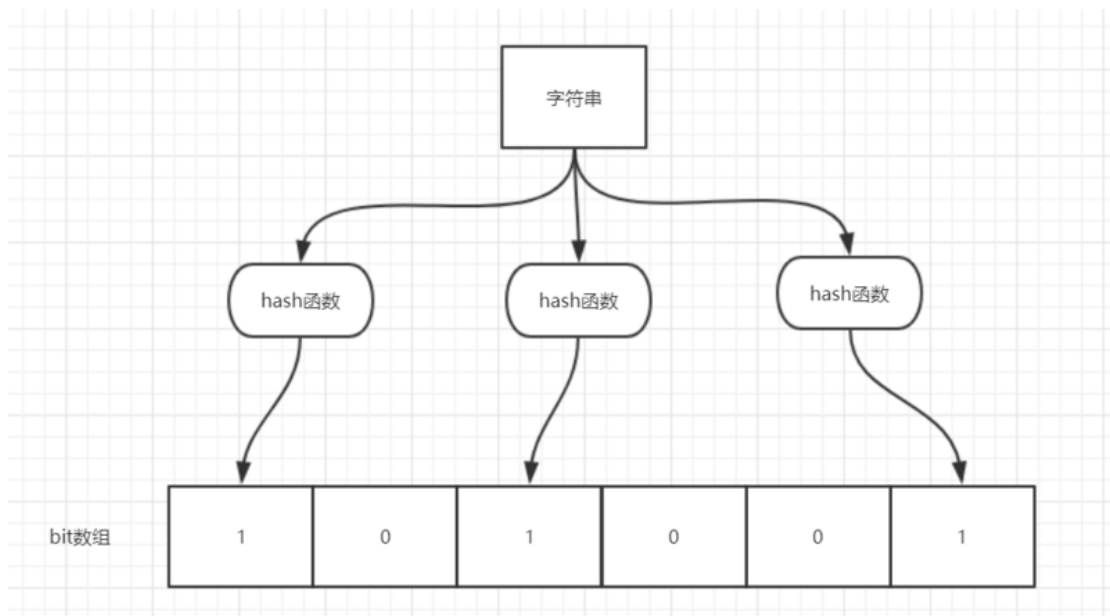


图 3-1 举例

如图所示，当字符串存储要加入到布隆过滤器中时，该字符串首先由多个哈希函数生成不同的哈希值，然后在对应的位数组的下表的元素设置为 1（当位数组初始化时，所有位置均为 0）。当第二次存储相同字符串时，因为先前的对应位置已设置为 1，所以很容易知道此值已经存在（去重非常方便）。

如果我们需要判断某个字符串是否在布隆过滤器中时，只需要对给定字符串再次进行相同的哈希计算，得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

不同的字符串可能哈希出来的位置相同，这种情况我们可以适当增加位数组大小或者调整我们的哈希函数。

综上，我们可以得出：布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不在，那么这个元素一定不在。

错误率推导：

假设 Hash 函数以等概率条件选择并设置 Bit Array 中的某一位， m 是该位数组的大小， k 是 Hash 函数的个数，那么位数组中某一特定的位在进行元

素插入时的 Hash 操作中没有被置位的概率是： $1 - \frac{1}{m}$ 。那么在所有 k 次

Hash 操作后该位都没有被置“1”的概率是： $\left(1 - \frac{1}{m}\right)^k$ 。如果我们插入了 n

个元素，那么某一位仍然为“0”的概率是： $\left(1 - \frac{1}{m}\right)^{kn}$ ；因而该位为“1”的

概率是： $1 - \left(1 - \frac{1}{m}\right)^{kn}$ 。现在检测某一元素是否在该集合中。标明某个元素是否在集合中所需的 k 个位置都按照如上的方法设置为“1”，但是该方法可能会使算法错误的认为某一原本不在集合中的元素却被检测为在该集合中

(False Positives)，该概率由以下公式确定：

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

其实上述结果是在假定由每个

Hash 计算出需要设置的位 (bit) 的位置是相互独立为前提计算出来的，不难看出，随着 m (位数组大小) 的增加，假正例 (False Positives) 的概率会下降，同时随着插入元素个数 n 的增加，False Positives 的概率又会上升，对于给定的 m, n，如何选择 Hash 函数个数 k 由以下公式确定：

$\frac{m}{n} \ln 2 \approx 0.7 \frac{m}{n}$ ，此时 False Positives 的概率为： $2^{-k} \approx 0.6185^{m/n}$ 。而对于给定的 False Positives 概率 p，如何选择最优的位数组大小 m 呢，

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

上式表明，位数组的大小最好与插入元素的个数成线性关系，

对于给定的 m, n, k，假正例概率最大为： $\left(1 - e^{-k(n+0.5)/(m-1)}\right)^k$ 。

第四章 基于水库抽样算法的技术估计算法介绍

水库抽样算法介绍:

问题描述: 输入一组数据, 大小未知, 要求输出这组数据中 K 个均匀抽取的数据, 即从 N 个元素中随机的抽取 k 个元素, 其中 N 无法确定, 保证每个元素抽到的概率相同。

设 k 为要抽样的个数, N 为总体个数位置, n 为当前遍历的元素的位置, $pool$ 为用来保存抽到的样本大小为 k 的数组。则具体步骤如下:

1. $n \leq k$, 把当前值放入 $pool$ 中, 构成初步样本;
2. $n > k$, 生成一个随机数 p , 如果 $p < k/n$, 那么把 $pool$ 中的任意一个数替换为第 n 个数。如果 $p \geq k/n$, 继续保留前面的数。
3. 直到数据流结束, 返回此 k 个数。

正确性证明:

1) 当 $n \leq k$ 时, 出现在 $pool$ 中的每个元素概率都是相同的, 都为 1

2) 当 $n = k+1$ 时, 计算前 k 个元素在 $pool$ 的概率

(a). 前 k 个元素在 $pool$ 中的元素概率都为 1

(b). 由假设得, 第 $k+1$ 个元素被选中的概率为: $k/(k+1)$, $pool$ 任意元素被替换的概率为 $(k/(k+1)) * (1/k) = 1/(k+1)$, 没被替换(即选中)的概率为 $1 - 1/(k+1) = k/(k+1)$ 。

由 $a*b = 1 * k/(k+1) = k/(k+1)$, 前 k 个元素和 $k+1$ 元素被选中的概率都为 $k/(k+1)$ 。

3) 当 $n > k+1$ 时, 计算前 $n-1$ 个元素在 $pool$ 的概率

(a). 前 $n-1$ 个元素在 $pool$ 中被选择的的概率为 $k/(n-1)$

(b). 由假设得, 第 n 个元素被选中的概率为: k/n , $pool$ 任意元素被替换的概率为 $(k/n) * (1/k) = 1/n$, 没被替换(即选中)的概率为 $1 - 1/n = (n-1)/n$ 。

由 $a*b = (k/(n-1)) * ((n-1)/n) = k/n$, 前 k 个元素和 $k+1$ 元素被选中的概率都为 k/n 。

因此假设成立, 所以到数据结束时, 所有元素的抽到的概率都为 k/N 。

用样本来估计总体的基数：

通过水库抽样算法来进行抽样，我们可以得到一个大小为 k 的样本集合，并且可以确信每个样本都是随机抽样得来的。只要样本的大小 k 与全部元素的数目 N 满足一定关系时，我们就可以通过样本的结构来估计整体的结果，并且有理由相信，得到的结果是有说服力的。

具体来说，如果大小为 k 的样本集合中不重复元素的数目为 m ，则大小为 N 的整体中，不重复元素的个数的估计值为： $(m/k)*N$ 。

第五章 实验

准备测试数据：

1. 首先获取到一份有 5000 多个成语的文本文件（记为 lib1），以及一份有 20000 多个不重复元素的文本文件（记为 lib2）。
2. 然后从第一份文件的 5000 多个词语中进行随机抽取，抽取 100000 此得到一份大小为 100000 的有重复元素的数据集。
3. 抽取 500000、1000000、3000000、5000000 次，得到多份测试数据集。
4. 对第二份文件重复以上过程，得到多份数据集。

对 HyperLogLog 进行测试：

(1) 预期错误率为 0.1

① 测试数据集大小为 100000

基于 lib1 的测试结果如下：

```
HLL结果：3732
准确结果：5024
耗时：61毫秒
```

基于 lib2 的测试结果如下：

```
HLL结果：27288
准确结果：28753
耗时：64毫秒
```

结论：样本本身的特征对 HyperLogLog 的估计结果影响较大，当样本的真实基数较大时，HyperLogLog 的估计结果更加准确。

② 测试数据集大小为 500000

基于 lib1 的测试结果如下：

```
HLL结果：3732
准确结果：5024
耗时：166毫秒
```

基于 lib2 的测试结果如下：

```
HLL结果: 29520  
准确结果: 33725  
耗时: 420毫秒
```

结论：增大数据集可以提高 HyperLogLog 的正确率。

③测试数据集大小为 3000000

基于 lib1 的测试结果如下：

```
HLL结果: 3732  
准确结果: 5024  
耗时: 626毫秒
```

基于 lib2 的测试结果如下：

```
HLL结果: 29520  
准确结果: 33727  
耗时: 490毫秒
```

结论：当数据集中的不重复元素不变时，增大数据集基本不能提高 HyperLogLog 的正确率。

(2) 预期错误率为 0.01 且测试数据集大小为 3000000

基于 lib1 的测试结果如下：

```
HLL结果: 5018  
准确结果: 5024  
耗时: 452毫秒
```

基于 lib2 的测试结果如下：

```
HLL结果: 34229  
准确结果: 33727  
耗时: 449毫秒
```

结论：HyperLogLog 的估计结果受参数影响较大，降低预期错误率，可以有效提高估计结果的准确率，并且执行时间并不会明显增加。

对 BloomFilter 进行测试：

(1) 预期错误率为 0.1

① 测试数据集大小为 100000

基于 lib1 的测试结果如下：

```
BloomFilter结果: 5024  
准确结果: 5024  
耗时: 139
```

基于 lib2 的测试结果如下：

```
BloomFilter结果: 28748  
准确结果: 28753  
耗时: 185
```

结论：由于布隆过滤器本身的特性，样本本身的特征对 BloomFilter 的估计结果有影响，当样本的真实基数较大时，BloomFilter 的估计结果准确性下降。

② 测试数据集大小为 500000

基于 lib1 的测试结果如下：

```
BloomFilter结果: 5024  
准确结果: 5024  
耗时: 372
```

基于 lib2 的测试结果如下：

```
BloomFilter结果: 33725  
准确结果: 33725  
耗时: 314
```

③ 测试数据集大小为 3000000

基于 lib1 的测试结果如下：

```
BloomFilter结果: 5024  
准确结果: 5024  
耗时: 1794
```


基于 lib2 的测试结果如下：

```
BloomFilter结果: 33727  
准确结果: 33727  
耗时: 1559
```

(2) 预期错误率为 0.01 且测试数据集大小为 3000000

基于 lib1 的测试结果如下：

```
BloomFilter结果: 5024  
准确结果: 5024  
耗时: 2401
```

基于 lib2 的测试结果如下：

```
BloomFilter结果: 33727  
准确结果: 33727  
耗时: 2355
```

结论：增加预期错误率会使得程序执行时间大幅度增大。

(3) 预期错误率为 0.1 且预期数据为 100000 的前提下，实际数据大小为 3000000：

基于 lib1 的测试结果如下：

```
BloomFilter结果: 5024  
准确结果: 5024  
耗时: 1292
```

基于 lib2 的测试结果如下：

```
BloomFilter结果: 33709  
准确结果: 33727  
耗时: 1265
```

结论：如果实际数据量大于预期数据量，则 BloomFilter 的估计准确率会下降。

对抽样估计法进行测试：

(1) 抽样率为 0.1 且测试数据集大小为 100000

基于 lib1 的测试结果如下：

```
准确结果：5024  
抽样法：43170  
耗时：55毫秒
```

基于 lib2 的测试结果如下：

```
准确结果：28753  
抽样法：67680  
耗时：39毫秒
```

结论：样本本身的特征对抽样法的估计结果有影响，当样本的真实基数较大时，BloomFilter 的估计结果更准确。

(2) 抽样率为 0.3 且测试数据集大小为 100000

基于 lib2 的测试结果如下：

```
准确结果：28753  
抽样法：52010  
耗时：48毫秒
```

结论：提高抽样率可以大幅度提高结果的准确性。

(3) 抽样率为 0.3 且测试数据集大小为 10000

```
准确结果：6811  
抽样法：8096  
耗时：40毫秒
```

结论：抽样法在数据集较小时的准确率更高。

第六章 结论

通过上面的实验，我们可以得到以下结论：

1. 样本本身的特征对 HyperLogLog 的估计结果影响较大，当样本的真实基数较大时，HyperLogLog 的估计结果更加准确。
2. 增大数据集可以提高 HyperLogLog 的正确率。
3. 当数据集中的不重复元素不变时，增大数据集基本不能提高 HyperLogLog 的正确率。
4. HyperLogLog 的估计结果受参数影响较大，降低预期错误率，可以有效提高估计结果的准确率，并且执行时间并不会明显增加。
5. 由于布隆过滤器本身的特性，样本本身的特征对 BloomFilter 的估计结果有影响，当样本的真实基数较大时，BloomFilter 的估计结果准确性下降。
6. 增加预期错误率会使得程序执行时间大幅度增大。
7. 如果实际数据量大于预期数据量，则 BloomFilter 的估计准确率会下降。
8. 样本本身的特征对抽样法的估计结果有影响，当样本的真实基数较大时，BloomFilter 的估计结果更准确。
9. 提高抽样率可以大幅度提高结果的准确性。
10. 抽样法在数据集较小时的准确率更高。
11. 相较于 HyperLogLog 和 BloomFilter 来说，抽样法得到的结果的准确率更低，且没有办法保证误差在一定范围内。
12. 平均执行时间排序：抽样法 < HyperLogLog < BloomFilter。

参考文献：

[1] *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*

[2] M. Mitzenmacher. *Compressed Bloom Filters*. *IEEE/ACM Transactions on Networking* 10:5 (2002), 604—612.

[3] <https://blog.csdn.net/u012397189/article/details/52181005>

附录：

本文的代码实现：<https://github.com/Zhang-Qing-Yun/cardinality-estimation>