

重构 大数据统计

|| 杨旭 著 ||

内 容 简 介

大数据的统计计算是进行数据探索和分析挖掘的基础,在实际应用中会遇到两个问题:一个是需要多少资源;另一个是计算时间,它关系到数据探索分析的效率和效果。人们都希望花更少的钱,并且希望计算时间更短,但对于某个确定的计算过程,它们是成反比的。本书作者就是从统计计算的算法入手,重构其计算过程,从而同时降低资源使用量和计算时间。本书提出了一套完整的关于大数据统计的计算理论,包括常用的各种统计量和统计方法。基于本书内容开发的数据分析工具已经在阿里巴巴集团内部的多个部门使用,并取得显著效果。另外,本书还提供大量的示例程序代码帮助读者进一步了解算法细节,便于将书中的方法运用于实际计算。

本书适合对大数据分析感兴趣的读者阅读,本书前面章节比较容易理解,包含了常用统计量的计算;后面的各章节需要读者具备一些基础知识。建议读者根据自己的兴趣和工作需要,选择相应的内容进行参考。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

重构大数据统计 / 杨旭著. —北京:电子工业出版社, 2014.8

(大数据丛书. 阿里巴巴集团技术丛书)

ISBN 978-7-121-22500-0

I. ①重… II. ①杨… III. ①数据处理 IV. ①TP274

中国版本图书馆 CIP 数据核字(2014)第 030832 号

策划编辑: 刘 皎

责任编辑: 李利健

印 刷: 北京中新伟业印刷有限公司

装 订: 河北省三河市路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×980 1/16 印张: 25.25 字数: 502 千字

版 次: 2014 年 8 月第 1 版

印 次: 2014 年 8 月第 1 次印刷

定 价: 79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线:(010) 88258888。

第 1 章

基本概念

本章将简要介绍一些最基本的概念：数据类型、变量、总体、样本、参数和统计量，并通过具体的例子和说明介绍本书常用的分布式计算思想，为读者深入阅读本书作铺垫。

1.1 数据类型

我们接触到的数据有很多，例如：用户姓名、性别、交易金额、商品单价、用户评分、交易时间等。按照所采用的计量尺度不同，可以分为三类：名义数据、有序数据和数值型数据。

1. 名义数据

名义数据（Nominal Data）是指对事物分类的结果不区分顺序，但有分类尺度计量形成的数据。

各个名义数据间无大小、高低和等级之分，唯一可行的是对发生的频数进行计算。例如，用户姓名和性别都为名义数据。名义数据可以用数字表示，例如，1 表示男，0 表示女。显然，这里的 1 并不意味着比 0 大。

2. 有序数据

有序数据（Ordinal Data）是指对事物分类的结果有顺序、有分类尺度计量形成的数据。

该类型数据可以进行排序操作，也可以对发生的频数进行计算。例如：用户评分（好、中、差），受教育水平（小学、初中、高中、大学及以上）。有序数据也可用数值表示，例如：对评分用 3 表示好，2 表示中，1 表示差；对受教育水平用 1 表示小学，2 表示初中，3 表示高中，4 表示大学及以上，其中的 4 意味着比 2 受教育水平更高。其数值计算结果也没有意义，例如：1+1+1=3 不能说明 3 个差评等于一个好评；2+2=4 不能说明受了两次初中教育相当于大学毕业。

3. 数量数据

数量数据（Quantitative Data）是按自然单位、度量衡单位、价值单位对事物进行测量的结果，该结果表现为具体的数值，取值为实数，可以进行所有的计算（求和、平均值等），包括前

两种数据类型的排序和计算发生的频数。例如：购买商品的个数、交易金额等。

上述三种数据类型的关系如图 1-1 所示。



图 1-1

1.2 总体和样本

说明事物某种特征的概念，称为变量（因素或者元）。例如：灯泡的寿命、购物时间、物品单价、物品个数等。

个体是由一个或多个变量（多元或多个因素）构成的。例如：某个灯泡的寿命是 1200 小时；一条网购记录为“用户名称：张三；购物时间：2013-9-1；物品单价：99.99；物品个数：10”。

包含所研究的全部个体的集合，称为总体。

对于所要研究的总体，通过观测或试验而得到的个体集合 X_1, X_2, \dots, X_n ，称为样本。这里的 X_i 称为第 i 个样本， n 称为样本大小或样本容量（Sample Size）。

通常，我们用表格来记录个体的集合，表格的行数对应个体的数量，每一列对应一个变量。

1.3 参数和统计量

用来描述总体特征的概括性数字度量，称为参数（Parameter）。

例如：某工厂生产的一批灯泡，把它们看作一个总体，灯泡的平均使用寿命就是一个重要的参数。但需要测试整批灯泡的寿命，才可以得到这个参数，得到参数的同时，这批灯泡也就都费掉了。我们能否只拿一小部分的灯泡来测试，从而估计出这个参数呢？这就需要下面的概念：样本统计量。

用来描述样本特征的概括性数字度量（简单地说，就是由样本计算出来的量），称为统计量（Statistic）。

例如：在上面的例子中，从这一批灯泡中抽样出 20 个，测试并计算其平均使用寿命为 1200 小时，则整批灯泡的平均使用寿命应该在 1200 小时“附近”。如何精确描述和确定这个“附近”值？这需要统计推断的一个重要内容是：参数估计。相关内容将在后面详细介绍。

1.4 分布式计算

分布式计算（Distributed Computing）是将大的计算任务（需要巨大的计算能力或需处理巨大的数据量）分解成许多小的部分，并分配给许多计算机进行处理，最后将所有小部分的计算结果综合起来得到最终的结果。

我们所说的大数据（如几百亿条的交易记录）一般就存储在由很多机器构成的分布式存储系统里，而这些机器同时也是我们进行数据分析的计算资源。如何通过分布式计算进行高效的统计计算，就是本书的重点。

用数学计算的例子来解释分布式计算，例如，要计算下面的式子：

$$a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + a_9 + a_{10} + a_{11} + a_{12} + a_{13} + a_{14} + a_{15}$$

利用加法结合律，我们有

$$(a_1 + a_2 + a_3 + a_4 + a_5) + (a_6 + a_7 + a_8 + a_9 + a_{10}) + (a_{11} + a_{12} + a_{13} + a_{14} + a_{15})$$

对于新的表达式，每个括号内的内容可以分别计算，最后将 3 个中间结果再进行加法计算，得到最终结果。通过括号将原本的计算问题分成 3 个小部分，每个小部分都可以独立进行计算，它是分布式计算能够处理大问题、提高计算速度的关键。而括号间的加号代表汇总过程，它保证我们可以得到原问题的正确结果。

上面的例子说明了数据的求和可以通过分布式方法进行计算。特别地，当每个 a_i 都取 1 的时候，就相当于求数据的总个数，每个小部分独立地计算出各自包含的数据的个数，再汇总到一起，就是数据的总个数。

分布式计算可以求得数据的总个数和数据的和。那么对于数据的平方和呢？下式：

$$a_1^2 + a_2^2 + a_3^2 + a_4^2 + a_5^2 + a_6^2 + a_7^2 + a_8^2 + a_9^2 + a_{10}^2 + a_{11}^2 + a_{12}^2 + a_{13}^2 + a_{14}^2 + a_{15}^2$$

也使用加法结合律，有

$$(a_1^2 + a_2^2 + a_3^2 + a_4^2 + a_5^2) + (a_6^2 + a_7^2 + a_8^2 + a_9^2 + a_{10}^2) + (a_{11}^2 + a_{12}^2 + a_{13}^2 + a_{14}^2 + a_{15}^2)$$

对于新的表达式，每个括号内的数据都是总体数据的一个小部分，可以独立进行计算；3 个括号内的计算完成后，就可进行汇总，即进行括号间的加法运算，得到全部数据的平方和结果。类似的，我们可以使用分布式计算得到数据的立方和、数据的四次方和。

我们再看一下另外两个非常熟悉的统计量：最大值和最小值。它们该如何通过分布式计算得到？仍以数据 a_i 为例进行说明，数据 a_i 的最大值表示为：

$$\max(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15})$$

这个表达式意味着计算时，从左边的两个数据 a_1 、 a_2 开始，选出最大的，然后和下一个数 a_3 进行比较，留下最大的数，再比较下一个，……，直到比较完最后一个数。由于整体数据的最大值一定包含该数据的任意个小部分的最大值，且所有部分的最大值中最大的就是整体数据的最大值。有如下等价的表达式：

$$\max\{\max(a_1, a_2, a_3, a_4, a_5), \max(a_6, a_7, a_8, a_9, a_{10}), \max(a_{11}, a_{12}, a_{13}, a_{14}, a_{15})\}$$

根据这个表达式，每个小部分分别求出最大值，然后所有小部分的最大值中最大的就是我们要求的结果。

同样也适用于计算最小值

$$\begin{aligned} & \min(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}) \\ & = \min\{\min(a_1, a_2, a_3, a_4, a_5), \min(a_6, a_7, a_8, a_9, a_{10}), \min(a_{11}, a_{12}, a_{13}, a_{14}, a_{15})\} \end{aligned}$$

综上所述，使用分布式计算可以很容易求得数据的总个数、和、平方和、立方和、四次方和、最大值和最小值。对于其他统计量（如方差、峰度等），该如何计算呢？接下来的章节就会回答这个问题。

第 2 章

单变量基本统计量

首先，要介绍我们最熟悉的数量统计量（均值、方差等），然后介绍适用于全部数据类型的频数统计量（频数信息、众数），最后介绍顺序统计量。

除了基本概念的简介，为了加深大家对基本概念的理解，本章还对常见的两个问题进行了深入讨论：样本方差为何除以 $n-1$ ？数据分布与标准差的关系如何？

本章的重点是如何高效地计算这些统计量，要点如下。

- 尽量减少数据读取的次数。在流式计算中，数据只有一次读取机会；在分布式存储系统中，每次读取数据都需要占用磁盘 I/O 和网络带宽，同时需要伴随着时间的消耗。
- 利用各统计量在数值上的关系，只计算少量的统计量，其他的通过与已计算出的统计量间的关系得到。
- 每次计算需要尽可能地获取更多的统计量。在对计算量影响非常小的情况下，同时计算出用户还可能需要的统计量，能减少实际应用中大规模计算的次数。

2.1 数量统计量

数量统计量是只适合数量类型数据的统计量，是我们最常见的统计量。

对于数量类型的数据样本 X_1, X_2, \dots, X_n ，其数量统计量的定义如下。

1. 均值（Mean）

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

用来描述数据取值的平均位置。

2. 方差 (Sample Variance)

$$S_n^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

3. 标准差 (Standard Deviation)

$$S_n = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2}$$

标准差在数值上等于方差的开平方，它们都是用来反映数据取值的离散（变异）程度。标准差的量纲与数据的量纲相同。

4. 变异系数 (Coefficient of Variation, CV)

$$\frac{S_n}{\bar{X}}$$

标准差和均值的比值称为变异系数。它是一个无量纲的量，用来刻画数据的相对分散性。

5. 标准误 (Standard Error)

$$\frac{\sqrt{\frac{1}{(n-1)} \sum_{i=1}^n (X_i - \bar{X})^2}}{\sqrt{n}} = \sqrt{\frac{1}{n(n-1)} \sum_{i=1}^n (X_i - \bar{X})^2}$$

标准误是由样本的标准差除以样本个数的开平方计算得到的。标准误代表的就是样本均数对总体均数的相对误差。

6. k 阶原点矩 (Moment)

$$\frac{1}{n} \sum_{i=1}^n X_i^k$$

一阶原点矩就是均值。

7. k 阶中心矩 (Central Moment)

$$\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^k$$

由均值定义可知，一阶中心矩恒等于 0。

下面介绍的两个统计量可以用来描述样本数据分布形状。

8. 偏度 (Skewness)

$$\sqrt{n} \sum_{i=1}^n (X_i - \bar{X})^3$$

$$\left[\sum_{i=1}^n (X_i - \bar{X})^2 \right]^{\frac{3}{2}}$$

偏度是用来刻画数据对称性的指标。关于均值对称的数据，其偏度系数为 0；若左侧数据比较分散，则偏度系数小于 0；若右侧数据比较分散，则偏度系数大于 0。

9. 峰度 (Kurtosis)

$$\frac{n \sum_{i=1}^n (X_i - \bar{X})^4}{\left[\sum_{i=1}^n (X_i - \bar{X})^2 \right]^2} - 3$$

峰度可以描述样本数据分布形态相对于正态分布的陡缓程度。

- 若 Kurtosis=0，则与正态分布的陡缓程度相同。
- 若 Kurtosis>0，则比正态分布的高峰更加陡峭，表现为尖顶峰。
- 若 Kurtosis<0，则比正态分布的高峰显得平缓，表现为平顶峰。

注：如果是正态分布，那么偏度为 0，峰度也为 0。

2.1.1 样本方差为何除以 $n-1$

统计上的标准解释为：取分母 $n-1$ ，可使样本方差的期望等于总体方差，即这种定义的样本方差是总体方差的无偏估计。

可以通过推导知道为什么是“无偏估计”。

设总体的均值和方差分别为 μ 和 σ^2 ，从中随机抽取 n 个样本 X_1, X_2, \dots, X_n ，则有

$$E(X_i) = \mu, \quad D(X_i) = E([X_i - E(X_i)]^2) = \sigma^2$$

对于样本均值 $\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$ ，有

$$E(\bar{X}) = E\left(\frac{1}{n} \sum_{i=1}^n X_i\right) = \frac{1}{n} \sum_{i=1}^n E(X_i) = \frac{1}{n} \sum_{i=1}^n \mu = \mu$$

即样本均值为总体均值的无偏估计。

接下来，我们先推导两个重要的中间结果 $E(\bar{X}^2)$ 和 $E(X_i^2)$ ，然后，计算方差无偏估计。

对于样本方差 $D(\bar{X})$ ，我们有如下两个等式：

$$D(\bar{X}) = E([\bar{X} - E(\bar{X})]^2) = E(\bar{X}^2 - 2\bar{X}E(\bar{X}) + E(\bar{X})^2) = E(\bar{X}^2) - E(\bar{X})^2 = E(\bar{X}^2) - \mu^2$$

$$D(\bar{X}) = D\left(\frac{1}{n} \sum_{i=1}^n X_i\right) = \frac{1}{n^2} D\left(\sum_{i=1}^n X_i\right) = \frac{1}{n^2} \sum_{i=1}^n D(X_i) = \frac{1}{n^2} \sum_{i=1}^n \sigma^2 = \frac{\sigma^2}{n}$$

由于上面两个等式的左端相同，所以右端的两个式子相等，我们可以得到：

$$E(\bar{X}^2) - \mu^2 = \frac{\sigma^2}{n}$$

即

$$E(\bar{X}^2) = \frac{\sigma^2}{n} + \mu^2$$

接下来，我们关注 $E(X_i^2)$ ，将其进行恒等变换有：

$$\begin{aligned} E(X_i^2) &= E([X_i - E(X_i)]^2 + 2X_iE(X_i) - E(X_i)^2) \\ &= E([X_i - E(X_i)]^2 + 2X_i\mu - \mu^2) \\ &= E([X_i - E(X_i)]^2) + 2\mu E(X_i) - \mu^2 \\ &= \sigma^2 + 2\mu \cdot \mu - \mu^2 \\ &= \sigma^2 + \mu^2 \end{aligned}$$

使用上面的推导结果 $E(\bar{X}^2) = \frac{\sigma^2}{n} + \mu^2$ 和 $E(X_i^2) = \sigma^2 + \mu^2$ ，则有

$$\begin{aligned}
E\left(\frac{1}{n-1}\sum_{i=1}^n[X_i - \bar{X}]^2\right) &= \frac{1}{n-1} \cdot E\left(\sum_{i=1}^n[X_i - \bar{X}]^2\right) \\
&= \frac{1}{n-1} \cdot E\left(\sum_{i=1}^n[X_i^2 - 2X_i\bar{X} + \bar{X}^2]\right) \\
&= \frac{1}{n-1} \cdot \left\{E\left(\sum_{i=1}^n X_i^2\right) - 2 \cdot E\left(\sum_{i=1}^n (X_i\bar{X})\right) + E\left(\sum_{i=1}^n \bar{X}^2\right)\right\} \\
&= \frac{1}{n-1} \cdot \left\{E\left(\sum_{i=1}^n X_i^2\right) - 2 \cdot nE(\bar{X}^2) + nE(\bar{X}^2)\right\} \\
&= \frac{1}{n-1} \cdot \left\{\sum_{i=1}^n E(X_i^2) - nE(\bar{X}^2)\right\} \\
&= \frac{1}{n-1} \cdot \left\{\sum_{i=1}^n (\sigma^2 + \mu^2) - n\left(\frac{\sigma^2}{n} + \mu^2\right)\right\} \\
&= \frac{1}{n-1} \cdot \{n(\sigma^2 + \mu^2) - (\sigma^2 + n \cdot \mu^2)\} \\
&= \sigma^2
\end{aligned}$$

由此式可知，样本方差 $\frac{1}{n-1}\sum_{i=1}^n[X_i - \bar{X}]^2$ 为总体方差 σ^2 的无偏估计，自然也说明了样本方差

中为什么要除以 $n-1$ 。

我们也可以从另一个角度来理解“除以 $n-1$ ”。这里引用《数理统计讲义》（王兆军、邹长亮编著，南开大学）中的解释。

样本方差（Sample Variance）

$$S_n^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

我们注意到样本方差的分母为 $n-1$ ，称之为 S_n^2 的自由度（degree of freedom），其解释如下：

- 有 n 个样本，它们都可以自由变化，由于有一个已用于估计总体均值，所以还有 $n-1$ 个可自由变化。
- 从 S_n^2 的定义不难看出，它是由 n 个数 $X_1 - \bar{X}, X_2 - \bar{X}, \dots, X_n - \bar{X}$ 的平方求和得到的，而

这 n 个数有一个约束 $\sum_{i=1}^n (X_i - \bar{X}) = 0$ ，故其自由度只有 $n-1$ 。

- 如果把 \bar{X} 代入 $\sum_{i=1}^n (X_i - \bar{X})^2$ ，则可知它是一个如下形式的二次型：

$$\sum_{i=1}^n \sum_{j=1}^n a_{ij} X_i X_j \quad (a_{ij} = a_{ji}), \text{ 且不难验证矩阵 } \mathbf{A} = (a_{ij}) \text{ 的秩为 } n-1.$$

2.1.2 数据分布与标准差的关系

前面定义标准差时，定性地提到：标准差用来反映数据取值的离散（变异）程度。但我们没有具体的概念，与均值相差 1 个标准差的范围内，会有多少个样本数据？相差 2 个标准差会圈定更大的范围，包含更多的数据，但会有多少？如果相差 k 个标准差会怎样？下面的定理可以定量地回答我们的疑问。

1. 契比雪夫（Chebyshev）定理

对于参数 $k > 1$ ，与均值相差小于 k 个标准差的样本数据，与样本总量的比例至少为：

$$1 - \frac{1}{k^2}$$

等价地，与均值相差大于或等于 k 个标准差的样本数据，占样本总量的比例不超过：

$$\frac{1}{k^2}$$

下面列举几个具体的 k 值进行说明。

- 与均值相差小于 2 个标准差的样本数据，数目至少为样本总数的 $3/4$ 。
- 与均值相差小于 3 个标准差的样本数据，数目至少为样本总数的 $8/9$ 。
- 与均值相差小于 4 个标准差的样本数据，数目至少为样本总数的 $15/16$ 。
- 与均值相差小于 k 个标准差的样本数据，数目至少为样本总数的 $1 - 1/k^2$ 。

下面举几个例子。

例 1：商品 A 的平均售价为 500 元，标准差为 30 元，则我们可以得出结论：商品 A 至少 $15/16 \times 100\% = 93.75\%$ 的售价在 (380, 620) 范围内，即在与均值 (500 元) 相差小于 4 个标准差 (30 元) 的范围。

例 2：商品 B 的平均售价为 500 元，标准差为 20 元，则我们可以得出结论：商品 B 至少 $24/25 \times 100\% = 96\%$ 的售价在 (400, 600) 范围内，即在与均值 (500 元) 相差小于 5 个标准差 (20 元) 的范围。

例 3：有 90 人参加的 100 分制考试，平均分是 80 分，标准差是 10 分，可得出结论：与平均 (80 分) 相差 3 个标准差 (10 分) 以上，即得分不超过 50 分的人，数目最多为 $90 \times 1/9 = 10$ 个。

这个契比雪夫（Chebyshev）定理由下面的同名不等式推导而来，过程很简捷，详见下面的

证明，希望能帮助读者加深理解契比雪夫定理。

2. 契比雪夫 (Chebyshev) 不等式

设随机变量 X 存在有限均值 $E(X)$ 和方差 $D(X)$ ，则有对任意 $\varepsilon > 0$ ，有

$$P\{|X - E(X)| \geq \varepsilon\} \leq \frac{D(X)}{\varepsilon^2}$$

设 $\varepsilon = k \cdot \sqrt{D(X)}$ ，因为 $\sqrt{D(X)}$ 为标准差， ε 即为 k 倍标准差，代入上面的不等式，则有

$$P\{|X - E(X)| < k \cdot \sqrt{D(X)}\} \geq 1 - \frac{D(X)}{(k \cdot \sqrt{D(X)})^2} = 1 - \frac{1}{k^2}$$

这样便证明了契比雪夫 (Chebyshev) 定理。

也许有人还会有疑问：契比雪夫 (Chebyshev) 定理给出的边界估计是否大了些？

我们看如下的例子：对给定的一个正数 $k > 1$ ，设分布 X 的取值范围为 3 个数 $\{-1, 0, 1\}$ ，且其分布概率为：

$$\begin{cases} P\{X = -1\} = \frac{1}{2k^2} \\ P\{X = 0\} = 1 - \frac{1}{k^2} \\ P\{X = 1\} = \frac{1}{2k^2} \end{cases}$$

则 X 的均值为 0，标准差为 $1/k$ 。在距均值小于 k 个标准差的范围内，即区间 $(-1, 1)$ ，只含有 0，而 0 所占的比例恰为 $1 - 1/k^2$ 。这说明契比雪夫 (Chebyshev) 定理给出的边界已不能再改进了。

2.1.3 新的计算公式

在进行基本统计量计算前，我们先关注在公式中经常出现的如下求和：

$$\sum_{i=1}^n (X_i - \bar{X})^2$$

$$\sum_{i=1}^n (X_i - \bar{X})^3$$

$$\sum_{i=1}^n (X_i - \bar{X})^4$$

每个公式求和之前都要知道 \bar{X} ，而知道 \bar{X} 就意味着要读取一遍数据，所以，每个求和都需要读两遍数据，能否只读一遍就完成计算呢？

先考查 $\sum_{i=1}^n (X_i - \bar{X})^2$ 对其进行恒等变换，有

$$\begin{aligned}\sum_{i=1}^n (X_i - \bar{X})^2 &= \sum_{i=1}^n (X_i^2 - 2X_i\bar{X} + \bar{X}^2) \\&= \sum_{i=1}^n X_i^2 - 2\bar{X} \sum_{i=1}^n X_i + \sum_{i=1}^n \bar{X}^2 \\&= \sum_{i=1}^n X_i^2 - 2\bar{X} \cdot n\bar{X} + n\bar{X}^2 \\&= \sum_{i=1}^n X_i^2 - n\bar{X}^2\end{aligned}$$

最终的表达式由 3 部分构成： $\sum_{i=1}^n X_i^2$ 、 \bar{X} 和 n 。显然，这 3 个量都可以通过读取一遍数据得到，

然后再通过上面的式子计算出 $\sum_{i=1}^n (X_i - \bar{X})^2$ 。

通过恒等变换能否解决 $\sum_{i=1}^n (X_i - \bar{X})^3$ 和 $\sum_{i=1}^n (X_i - \bar{X})^4$ 的计算呢？我们继续推导：

$$\begin{aligned}\sum_{i=1}^n (X_i - \bar{X})^3 &= \sum_{i=1}^n (X_i^3 - 3X_i^2\bar{X} + 3X_i\bar{X}^2 - \bar{X}^3) \\&= \sum_{i=1}^n X_i^3 - 3\sum_{i=1}^n X_i^2\bar{X} + 3\sum_{i=1}^n X_i\bar{X}^2 - \sum_{i=1}^n \bar{X}^3 \\&= \sum_{i=1}^n X_i^3 - 3\bar{X} \sum_{i=1}^n X_i^2 + 3\bar{X}^2 \sum_{i=1}^n X_i - n\bar{X}^3 \\&= \sum_{i=1}^n X_i^3 - 3\bar{X} \sum_{i=1}^n X_i^2 + 3\bar{X}^2 \cdot n\bar{X} - n\bar{X}^3 \\&= \sum_{i=1}^n X_i^3 - 3\bar{X} \sum_{i=1}^n X_i^2 + 2n\bar{X}^3\end{aligned}$$

类似地推导：

$$\begin{aligned}
\sum_{i=1}^n (X_i - \bar{X})^4 &= \sum_{i=1}^n (X_i^4 - 4X_i^3\bar{X} + 6X_i^2\bar{X}^2 - 4X_i\bar{X}^3 + \bar{X}^4) \\
&= \sum_{i=1}^n X_i^4 - 4\sum_{i=1}^n X_i^3\bar{X} + 6\sum_{i=1}^n X_i^2\bar{X}^2 - 4\sum_{i=1}^n X_i\bar{X}^3 + \sum_{i=1}^n \bar{X}^4 \\
&= \sum_{i=1}^n X_i^4 - 4\bar{X}\sum_{i=1}^n X_i^3 + 6\bar{X}^2\sum_{i=1}^n X_i^2 - 4\bar{X}^3\sum_{i=1}^n X_i + n\bar{X}^4 \\
&= \sum_{i=1}^n X_i^4 - 4\bar{X}\sum_{i=1}^n X_i^3 + 6\bar{X}^2\sum_{i=1}^n X_i^2 - 4\bar{X}^3 \cdot n\bar{X} + n\bar{X}^4 \\
&= \sum_{i=1}^n X_i^4 - 4\bar{X}\sum_{i=1}^n X_i^3 + 6\bar{X}^2\sum_{i=1}^n X_i^2 - 3n\bar{X}^4
\end{aligned}$$

由这些推导可知，只要读取一遍数据，同时计算出下面 5 个量：

$$n, \bar{X} = \frac{1}{n}\sum_{i=1}^n X_i, \sum_{i=1}^n X_i^2, \sum_{i=1}^n X_i^3 \text{ 和 } \sum_{i=1}^n X_i^4$$

就可得到

$$\sum_{i=1}^n (X_i - \bar{X})^2, \sum_{i=1}^n (X_i - \bar{X})^3 \text{ 和 } \sum_{i=1}^n (X_i - \bar{X})^4$$

有了上面的结果，我们可以期望更大的结果：读取一遍数据，计算出所有的基本统计量。

首先，考虑具体实现的情况，确定读数据时要得到的基本量。

- 在大部分情况下，大数据中包含的数据个数也是未知的，我们也将数据总数 n 作为一个基本量，在计算的过程中获得。
- 对于大规模数据，需要将其分成很多小块，分别交给不同的计算节点，从而实现并行计算，每个计算节点得到基本量后，还要进行汇总，对于计算节点很多的情况下，还需将汇总过程分层进行。数据的和、平方和、立方和与四次方和容易计算和汇总在一起；均值在汇总的过程中，需要考虑到各自的数据个数，进行转换计算时没有数据和的汇总简单直接。

综上所述，我们选择数据总个数、数据的和、平方和、立方和与四次方和作为基本量，将它们分别记为： n 、 sum 、 sum2 、 sum3 、 sum4 ，即

$$\text{sum} = \sum_{i=1}^n X_i$$

$$\text{sum2} = \sum_{i=1}^n X_i^2$$

$$\text{sum3} = \sum_{i=1}^n X_i^3$$

$$\text{sum4} = \sum_{i=1}^n X_i^4$$

下面是关键的一步，用这 5 个基本量计算出所有该数据的数量统计量。

1. 样本总数

$$\text{count} = n$$

2. 均值 (Mean)

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i = \frac{\text{sum}}{n}$$

注：我们在下面也会用到上式的一个简单变形

$$\text{sum} = n \bar{X}$$

3. 方差 (Variance)

$$S_n^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2 = \frac{1}{n-1} \left(\sum_{i=1}^n X_i^2 - n\bar{X}^2 \right) = \frac{1}{n-1} (\text{sum2} - \text{sum} \cdot \bar{X})$$

4. 标准差 (Standard Deviation)

$$S_n = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2} = \sqrt{\text{variance}}$$

5. 变异系数 (Coefficient of Variation, CV)

$$\frac{S_n}{\bar{X}} = \frac{\text{standardDeviation}}{\text{mean}}$$

6. 标准误 (Standard Error)

$$\frac{\sqrt{\frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2}}{\sqrt{n}} = \frac{\text{standardDeviation}}{\sqrt{n}}$$

7. 2 阶原点矩 (Moment2)

$$\frac{1}{n} \sum_{i=1}^n X_i^2 = \frac{\text{sum2}}{n}$$

8. 3 阶原点矩 (Moment3)

$$\frac{1}{n} \sum_{i=1}^n X_i^3 = \frac{\text{sum3}}{n}$$

9. 4 阶原点矩 (Moment4)

$$\frac{1}{n} \sum_{i=1}^n X_i^4 = \frac{\text{sum4}}{n}$$

10. 2 阶中心矩 (Central Moment2)

$$\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2 = \frac{1}{n} \left(\sum_{i=1}^n X_i^2 - n\bar{X}^2 \right) = \frac{1}{n} (\text{sum2} - \text{sum} \cdot \bar{X})$$

11. 3 阶中心矩 (Central Moment3)

$$\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^3 = \frac{1}{n} \left(\sum_{i=1}^n X_i^3 - 3\bar{X} \sum_{i=1}^n X_i^2 + 2n\bar{X}^3 \right) = \frac{1}{n} (\text{sum3} - 3\bar{X} \cdot \text{sum2} + 2 \cdot \text{sum} \cdot \bar{X}^2)$$

12. 4 阶中心矩 (Central Moment4)

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^4 &= \frac{1}{n} \left(\sum_{i=1}^n X_i^4 - 4\bar{X} \sum_{i=1}^n X_i^3 + 6\bar{X}^2 \sum_{i=1}^n X_i^2 - 3n\bar{X}^4 \right) \\ &= \frac{1}{n} (\text{sum4} - 4\bar{X} \cdot \text{sum3} + 6\bar{X}^2 \cdot \text{sum2} - 3 \cdot \text{sum} \cdot \bar{X}^2) \end{aligned}$$

13. 样本偏度 (Skewness)

$$\begin{aligned} \frac{\sqrt{n} \sum_{i=1}^n (X_i - \bar{X})^3}{\left[\sum_{i=1}^n (X_i - \bar{X})^2 \right]^{\frac{3}{2}}} &= \frac{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^3}{\left(\frac{1}{n} \right)^{\frac{3}{2}} \left[\sum_{i=1}^n (X_i - \bar{X})^2 \right]^{\frac{3}{2}}} = \frac{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^3}{\left[\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2 \right]^{\frac{3}{2}}} \\ &= \frac{\text{centralMoment3}}{\text{centralMoment2} \cdot \sqrt{\text{centralMoment2}}} \end{aligned}$$

14. 样三本峰度 (Kurtosis)

$$\frac{n \sum_{i=1}^n (X_i - \bar{X})^4}{\left[\sum_{i=1}^n (X_i - \bar{X})^2 \right]^2} - 3 = \frac{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^4}{\frac{1}{n^2} \left[\sum_{i=1}^n (X_i - \bar{X})^2 \right]^2} - 3 = \frac{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^4}{\left[\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2 \right]^2} - 3$$
$$= \frac{\text{centralMoment4}}{\text{centralMoment2} \cdot \text{centralMoment2}} - 3$$

至此，我们实现了只读一遍数据，便获得了 14 种常用的统计量。

2.1.4 代码实现

首先，计算总记录数 n 和如下 4 个求和：

$$\text{sum} = \sum_{i=1}^n X_i, \quad \text{sum2} = \sum_{i=1}^n X_i^2, \quad \text{sum3} = \sum_{i=1}^n X_i^3, \quad \text{sum4} = \sum_{i=1}^n X_i^4$$

对于分布式系统，每个计算节点只计算一部分数据，需要汇总过程，才能得到最终的结果。对于我们关注的 n 、 sum 、 sum2 、 sum3 和 sum4 ，汇总过程很简单，直接加和就可以，下面代码中的函数 `calculate(MeasureCalculator mc)`就是用于汇总过程的。

参考 Java 代码如下：

```
public class MeasureCalculator {

    public long count;
    public double sum;
    public double sum2;
    public double sum3;
    public double sum4;

    public MeasureCalculator() {
        count = 0;
        sum = 0.0;
        sum2 = 0.0;
        sum3 = 0.0;
        sum4 = 0.0;
    }
}
```

```

        count = 0;
    }

    public void calculate(double d) {
        sum += d;
        sum2 += d * d;
        sum3 += d * d * d;
        sum4 += d * d * d * d;
        count++;
    }

    public void calculate(MeasureCalculator mc) {
        this.sum += mc.sum;
        this.sum2 += mc.sum2;
        this.sum3 += mc.sum3;
        this.sum4 += mc.sum4;
        this.count += mc.count;
    }
}

```

有了 5 个基本量后, 就可以利用前面推导的公式, 计算出 14 种统计量, 参考 Java 代码如下:

```

mean = sum / count;

if (1 == count) {
    variance = 0.0;
} else {
    variance = (sum2 - mean * sum) / (count - 1);
}

standardDeviation = Math.sqrt(variance);
cv = standardDeviation / mean;
standardError = standardDeviation / Math.sqrt(count);

moment2 = sum2 / count;
moment3 = sum3 / count;
moment4 = sum4 / count;

centralMoment2 = (sum2 - mean * sum) / count;
centralMoment3 = (sum3 - 3 * sum2 * mean + 2 * sum * mean * mean) / count;
centralMoment4 = (sum4 - 4 * sum3 * mean + 6 * sum2 * mean * mean - 3 * sum
* mean * mean * mean) / count;

skewness = centralMoment3 / (centralMoment2 * Math.sqrt(centralMoment2));

```

```
kurtosis = centralMoment4 / (centralMoment2 * centralMoment2) - 3;
```

2.2 频数统计量

在一组数据中，有的数据只出现一次，有的数据能出现多次，我们称每个数据出现的次数为频数（Frequency）。

一组数据中出现次数最多的数是众数（Mode），如果出现次数最多的数有两个或者两个以上，则取其中最小的那个。

频数统计最典型的使用场景是投票选举，在黑板上写下所有候选人的名字，并在旁边预留一个位置计数，如果候选人的名字在黑板上写不下，最终就无法给出所有候选人的票数。

频数计算的一般步骤如下。

- （1）建立一个空的对应表（包含两列：数据值和累计计数值）。
- （2）读取一个数据，判断该数据是否在对应的表中：
 - 1）若存在，找到对应表中数据的位置，并使其累计计数值增加 1。
 - 2）若不存在，则将该数据新增到对应的表中，并初始化其累计计数值为 1。
- （3）若还有数据没有统计，则转到第（2）步继续操作。
- （4）输出对应表的数据和频数。

注：上面的算法需要有足够的空间保存对应表，否则计算无法完成。

对于大规模的数据，可以通过如下方式计算出全部数据的频数信息。

- （1）将数据分块，每块数据由一个计算节点负责，每个计算节点将每条数据通过 Hash 函数确定其应该被发送到哪个汇总节点。
- （2）每个汇总节点被分到的不同数据的个数不会很多，采用上面一般的计算频数的方法即可。
- （3）将每个汇总节点输出的数据和频数信息直接拼接在一起，就得到了整个大数据集的频数信息。

注：在这个算法中，Hash 函数的使用非常重要，保证了相等的数据会分到同一个汇总节点，而每个汇总节点需要处理的不同数据的个数也会比较均衡。MapReduce 编程模式中的典型例子 wordcount 就是用的这种思想。

在实际应用中，人们关注频数信息的那些样本往往有大量的数据是重复出现的，不同数据的个数较少。针对这样的情况，我们推荐更高效的算法，其核心思想是设定对应表的容量（一般为

1000 或 10000)，如果不同数据的个数超过容量，则可以终止计算，这样可以保证我们得到那些不同数据个数较少的样本的频数信息，而且计算的效率也会很高。算法分为下面两个阶段：

- 将数据分块，每块数据由一个计算节点负责，计算出这部分数据的频数信息。
- 汇总每个计算节点的频数信息，得到整个大数据集的频数信息。

在第一阶段中，每个计算节点使用带容量限制的频数计算方法，步骤如下。

(1) 建立一个空的对应表（包含两列：数据值和累计计数值），设定表的容量。

(2) 读取一个数据，判断该数据是否在对应的表中：

1) 若存在，找到对应表中数据的位置，并使其累计计数值增加 1。

2) 若不存在，则先判断对应表中已有数据的个数是否大于表的容量：

① 若小于，则将该数据新增到对应表中，并初始化其累计计数值为 1。

② 若大于或等于，则说明没有空间完成数据的频数统计，应该退出，直接跳到第（4）步，输出“超出容量，无法计算”。

(3) 若还有数据没有统计，则转到第（2）步继续操作。

(4) 输出对应表的数据和频数，或输出“超出容量，无法计算”。

在第二阶段，若有一个计算节点的计算结果是“超出容量，无法计算”，则整个样本频数计算也返回这一结果。否则，逐个将各节点的频数信息进行汇总，一旦在汇总过程中出现不同数据的个数大于容量，则立即停止计算。

参考 Java 代码如下：

```
public class FrequencyCalculator {

    private final int capacity;
    private TreeMap<Object, Long> treeMap = new TreeMap<Object, Long>();

    public FrequencyCalculator() {
        this.capacity = 1000;
    }

    public FrequencyCalculator(int capacity) {
        this.capacity = capacity;
    }

    public Object getMode() {
        if (null != treeMap) {
            Object mode = null;
            long modeCount = 0;
            Iterator<Entry<Object, Long>> it = treeMap.entrySet().iterator();
            while (it.hasNext()) {
                Entry<Object, Long> e = it.next();
```

```
        if (modeCount < e.getValue().longValue()) {
            mode = e.getKey();
            modeCount = e.getValue().longValue();
        }
    }
    return mode;
} else {
    throw new RuntimeException("不同元素个数超过" + capacity + ", 本方法不能计算众数!");
}
}

public void calc(Object obj) {
    if (null != treeMap) {
        if (treeMap.containsKey(obj)) {
            treeMap.put(obj, treeMap.get(obj) + 1);
        } else {
            if (treeMap.size() > this.capacity) { //Too many different items!
                treeMap = null;
            }
            treeMap.put(obj, new Long(1));
        }
    }
}

public void calc(double d) {
    calc(new Double(d));
}

public void calc(long k) {
    calc(new Long(k));
}

public void calc(int k) {
    calc(new Integer(k));
}

public void calc(boolean b) {
    calc(new Boolean(b));
}

public void calc(FrequencyCalculator freq2) {
    if (null != treeMap) {
        Iterator<Entry<Object, Long>> it = freq2.treeMap.entrySet().iterator();
```

```

        while (it.hasNext()) {
            Entry<Object, Long> e = it.next();
            Object obj = e.getKey();
            if (null != treeMap) {
                if (treeMap.containsKey(obj)) {
                    treeMap.put(obj, treeMap.get(obj) + e.getValue().longValue());
                } else {
                    if (treeMap.size() > this.capacity) { //Too many different items!
                        treeMap = null;
                    }
                    treeMap.put(obj, e.getValue().longValue());
                }
            } else {
                break;
            }
        }
    }

    @Override
    public String toString() {
        java.io.CharArrayWriter cw = new java.io.CharArrayWriter();
        java.io.PrintWriter pw = new java.io.PrintWriter(cw, true);
        if (null != treeMap) {
            pw.println("共有" + treeMap.size() + "个不同元素:");
            Iterator<Entry<Object, Long>> it = treeMap.entrySet().iterator();
            while (it.hasNext()) {
                Entry<Object, Long> e = it.next();
                pw.print(e.getKey());
                pw.print(" : ");
                pw.println(e.getValue());
            }
            pw.println("众数: " + getMode());
        } else {
            pw.println("不同元素个数超过" + capacity);
        }
        return cw.toString();
    }
}

```

注：其中的函数 `public void calc(FrequencyCalculator freq2)` 可以用来逐一汇总全部计算节点的频数信息。

使用函数的例子如下。

```
public static void main(String[] args) throws Exception {
    double[] data = new double[]{1.2, 1.2, 3, 5, 7, 4, 9, 9, 9, 10.5, 98, 1000006};
    calcFrequency(data, 1000);

    FrequencyCalculator fsi = new FrequencyCalculator(100);
    for (double d : data) {
        fsi.calc(d);
    }
    System.out.println(fsi);
    FrequencyCalculator fsi2 = new FrequencyCalculator(100);
    for (double d : data) {
        fsi2.calc(d + 2);
    }
    fsi.calc(fsi2);
    System.out.println(fsi);
}
```

运行结果为：

共有 9 个不同元素：

```
1.2 : 2
3.0 : 1
4.0 : 1
5.0 : 1
7.0 : 1
9.0 : 3
10.5 : 1
98.0 : 1
1000006.0 : 1
众数： 9.0
```

共有 15 个不同元素：

```
1.2 : 2
3.0 : 1
3.2 : 2
4.0 : 1
5.0 : 2
6.0 : 1
7.0 : 2
9.0 : 4
10.5 : 1
11.0 : 3
12.5 : 1
98.0 : 1
```



```
100.0 : 1
1000006.0 : 1
1000008.0 : 1
众数 : 9.0
```

2.3 次序统计量

设 X_1, X_2, \dots, X_n 为样本，把 X_1, X_2, \dots, X_n 由小到大排列成 $X_{(1)} X_{(2)} \cdots X_{(n)}$ ，则称 $(X_{(1)}, X_{(2)}, \dots, X_{(n)})$ 为次序统计量， $X_{(i)}$ 称为第 i 个次序统计量。

由次序统计量的定义，可以定义极小值 (Min) 为 $X_{(1)}$ ，极大值 (Max) 为 $X_{(n)}$ ，极差 (Range) 为 $X_{(n)} - X_{(1)}$ 。

分位数 (Quantile)：对于给定的 $p \in [0, 1]$ ， p 分位数为 $X_{1+[(n-1)p]}$ ，其中， $[x]$ 表示不超过 x 的最大整数。

中位数 (Median)：是指将统计样本中的各个值按大小顺序排列起来，形成一个数列，处于数列中间位置的值就称为中位数，也就是 0.5 分位数，即 $X_{1+[\frac{n-1}{2}]}$ 。

q -分位数 (q -Quantile)：将排序好的数列分为 q 份，处于 $q-1$ 个分割点和端点位置的值。

我们常见的有百分位数 (Percentile) 和四分位数 (Quartile)。百分位数就是将排序好的数列分为 100 份，处于 101 个分割点和端点位置的值，即为 $p = 0, 0.01, 0.02, \dots, 0.98, 0.99, 1$ 所对应的分位数。四分位数就是将排序好的数列分为四份，相应的五个分割点和端点都有特殊的意义：

- 第 0 个四分位数实际为通常所说的最小值 (MINimum)。
- 第 1 个四分位数 (1st Quartile、Lower Quartile 或者 Q1)，为 0.25 分位数。
- 第 2 个四分位数实际为通常所说的中位数。
- 第 3 个四分位数 (En: 3rd Quartile、Upper Quartile 或者 Q3)，为 0.75 分位数。
- 第 4 个四分位数实际为通常所说的最大值 (MAXimum)。

还有一个在实际应用中非常重要的统计量：秩 (Rank)。

对数据 X_1, X_2, \dots, X_n 进行排序，则有 $X_{i_1}^{(1)} < X_{i_2}^{(2)} < \dots < X_{i_n}^{(n)}$ ，其中， $X_i^{(k)}$ 表示数据 X_i 排在第 k 个位置，即数据 X_i 的位置编号为 k ，则称 X_i 的秩为 X_1, X_2, \dots, X_n 中所有与 X_i 相等的数据的位置编号的平均值。

显然，若 $X_i = X_j$ ，则 X_i 与 X_j 的秩相等。对于特殊情况，若数据 X_1, X_2, \dots, X_n 中任何两个值都不等，则 X_i 的秩即为其位置编号。

例 1：对于数据 {31, 7, 131, 32, 57, 24}，因为 $7 < 24 < 31 < 32 < 57 < 131$ ，则这 6 个数均不相等，它们的秩即为其位置编号，结果如表 2-1 所示。

表 2-1

数据	31	7	131	32	57	24
----	----	---	-----	----	----	----

秩	3	1	6	4	5	2
---	---	---	---	---	---	---

例 2：对于数据{32, 7, 131, 32, 57, 24}，因为 $7 < 24 < 32 = 32 < 57 < 131$ ，则 32 的秩为位置编号 3 和 4 的平均值，这 6 个数的秩如表 2-2 所示。

表 2-2

数据	31	7	131	32	57	24
秩	3.5	1	6	3.5	5	2

下面，我们首先介绍次序统计量如何通过分布式排序算法计算次序统计量。对于大数据，排列全部数据要比只读取一遍数据花费更多的时间和 CPU、内存、网络等资源。我们深入研究一下，会发现的确有一些次序统计量其实不需要全局排序。

2.3.1 通过排序方法计算次序统计量

在实际应用中，我们使用并行正则采样排序(Parallel Sorting by Regular Sampling, 简称 PSRS)算法。该算法除了具有排序的高效性外，在计算过程中，各计算节点所使用的内存也比较均衡，且有严格的上界，可以避免个别计算节点使用过多的内存，从而导致机器出现内存溢出(out of memory)问题。

假设共 n 个数据，记为 X_0, X_1, \dots, X_{n-1} ，使用的计算节点数为 p 个。采用 PSRS 排序的具体步骤如下。

(1) 将有 n 个数据均匀分割为 p 份，每个计算节点读取一份数据到内存，第 i 个计算节点读取到的数据个数为：

$$n_i = \begin{cases} \left\lceil \frac{n}{p} \right\rceil + 1 & , \quad i < n \% p \\ \left\lfloor \frac{n}{p} \right\rfloor & , \quad i \geq n \% p \end{cases}$$

其中， $n \% p$ 表示 n 除以 p 所得的余数。

记第 i 个计算节点包含的数据为 $X_0^i, X_1^i, \dots, X_{n_i-1}^i$ 。

(2) 每个计算节点对内存中的 n_i 个数据 $X_0^i, X_1^i, \dots, X_{n_i-1}^i$ 进行排序，得到 $Y_0^i, Y_1^i, \dots, Y_{n_i-1}^i$ 。

(3) 每个计算节点从 $Y_0^i, Y_1^i, \dots, Y_{n_i-1}^i$ 中采样出 $p-1$ 个数据，设

$$w = \left\lfloor \frac{n}{p^2} \right\rfloor$$

则采样出的 $p-1$ 个数据为 $Y_{w-1}^i, Y_{2w-1}^i, Y_{3w-1}^i, \dots, Y_{(p-1)w-1}^i$ 。

(4) 将每个计算节点采样出的有序数据 $Y_{w-1}^i, Y_{2w-1}^i, Y_{3w-1}^i, \dots, Y_{(p-1)w-1}^i$ ，共 $p(p-1)$ 个，汇总到第 0 个节点。对这些采样数据进行排序，设得到有序数据为

$$S_0, S_1, \dots, S_{p(p-1)-1}$$

从中选取 $p-1$ 个数据：

$$S_{(p-1)-1}, S_{2(p-1)-1}, \dots, S_{(p-1)(p-1)-1}$$

称其为主元，记作：

$$V_0, V_1, \dots, V_{p-1}$$

(5) 第 0 个计算节点将其计算出来的 $p-1$ 个主元分发给其他的计算节点。

(6) 每个计算节点根据其收到的 $p-1$ 个主元，将自己保存的有序数据 $Y_0^i, Y_1^i, \dots, Y_{n_i-1}^i$ 分为 p 段，且满足第 j ($j = 0, 1, 2, \dots, p-1$) 段数据的值均不大于 V_j 。

(7) 每个计算节点将其第 i 段的数据发送给第 i 个计算节点，从而使第 i 个计算节点含有所有计算节点的第 i 段数据 ($i = 0, 1, 2, \dots, p-1$)。

(8) 每个计算节点将上一步得到的 p 段有序数据序列进行多路归并排序，得到有序数列。到这里，按节点序号的顺序及各计算节点内数据顺序，即得到了全部 n 个数据的排序。

(9) 每个计算节点将有序数据输出。在实际应用中，经常将第 i 个计算节点的数据写到第 i 个文件里，这样按文件名的顺序逐个访问数据就是有序的。

注：由算法的第 (7) 步可知，每个计算节点接收其他节点传来的数据。可以证明，每个计算节点最多包含的数据量为 $2n/p$ ，即最多为初始数据量的 2 倍。所以，各计算节点间的负载虽有差异，但还在可以接受的范围。

对于 q -分位数的计算，我们可以在前面排序算法的基础上，保持前 8 个步骤不变，然后修改之后的操作，添加以下 3 个步骤。

(1) 每个计算节点统计出当前包含数据的个数，设为 m_i ，并汇总到第 0 个计算节点，然后由第 0 个计算节点将汇总结果 m_0, m_1, \dots, m_{p-1} 分发到各个计算节点。

(2) 利用 q -分位数，求第 k 个分位值在排序数列中位置的计算公式为：

$$idx = \left\lfloor \frac{(n-1) \cdot k}{q} \right\rfloor, \quad k = 0, 1, 2, \dots, q$$

对于第 i 个计算节点所包含的位置范围为：

$$\left[\sum_{j=0}^{i-1} m_j, \left(\sum_{j=0}^i m_j \right) - 1 \right]$$

在其中查找出所有包含的全部分位 k ，相应地计算出在整体排序数列中的位置，转化为第 i 个计算节点数据队列的相对位置，并取出对应的数据值，即为分位值。

(3) 每个计算节点将所计算出的分位值汇总到第 0 个计算节点，然后，由第 0 个计算节点输出。

计算秩 (Rank) 值的核心部分是排序，得到各个数据排序后的位置，并由这些位置信息得到每个数据的秩值，最后将这些秩值按照数据排序前的顺序输出。

设有 n 个数据，记为 X_0, X_1, \dots, X_{n-1} ，使用的计算节点数为 p 个。

定义一个结构体： $S_i = \{X_i, i, r\}$ ，其中第一项为 X_i 的值，第二项为其在原始数列中的位置，第三项为计算后返回来的秩值。再定义结构体 S_i 和 S_j 间的大小关系的判断步骤如下：

- 如果 $X_i < X_j$ ，则 $S_i < S_j$ 。
- 如果 $X_i > X_j$ ，则 $S_i > S_j$ 。
- 如果 $X_i = X_j$ ，且 $i < j$ ，则 $S_i < S_j$ 。
- 如果 $X_i = X_j$ ，且 $i > j$ ，则 $S_i > S_j$ 。
- 如果 $X_i = X_j$ ，且 $i = j$ ，则 $S_i = S_j$ 。

具体计算步骤如下。

(1) 将 n 个数据结构体均匀分割为 p 份，每个计算节点读取一份数据到内存。第 i 个计算节点读取到的数据个数为：

$$n_i = \begin{cases} \left\lceil \frac{n}{p} \right\rceil + 1 & , \quad i < n \% p \\ \left\lfloor \frac{n}{p} \right\rfloor & , \quad i \geq n \% p \end{cases}$$

记第 i 个计算节点包含的数据为 $S_0^i, S_1^i, \dots, S_{n_i-1}^i$ 。

(2) 每个计算节点对内存中的 n_i 个数据 $S_0^i, S_1^i, \dots, S_{n_i-1}^i$ 进行排序，得到 $T_0^i, T_1^i, \dots, T_{n_i-1}^i$ 。

(3) 每个计算节点从 $T_0^i, T_1^i, \dots, T_{n_i-1}^i$ 中采样出 $p-1$ 个数据，设

$$w = \left\lfloor \frac{n}{p^2} \right\rfloor$$

则采样出的 $p-1$ 个数据为 $T_{w-1}^i, T_{2w-1}^i, T_{3w-1}^i, \dots, T_{(p-1)w-1}^i$ 。

(4) 将每个计算节点采样出的有序数据 $T_{w-1}^i, T_{2w-1}^i, T_{3w-1}^i, \dots, T_{(p-1)w-1}^i$ ，共 $p(p-1)$ 个，汇总到第0个节点。对这些采样数据进行排序，设得到有序数据为

$$U_0, U_1, \dots, U_{p(p-1)-1}$$

从中选取 $p-1$ 个数据：

$$U_{(p-1)-1}, U_{2(p-1)-1}, \dots, U_{(p-1)(p-1)-1}$$

称其为主元，记作：

$$V_0, V_1, \dots, V_{p-1}$$

(5) 第0个计算节点将其计算出来的 $p-1$ 个主元分发给其他的计算节点。

(6) 每个计算节点根据其收到的 $p-1$ 个主元，将自己保存的有序数据 $T_0^i, T_1^i, \dots, T_{n_i-1}^i$ 分为 p 段，且满足第 j ($j = 0, 1, 2, \dots, p-1$) 段数据的值均不大于 V_j 。

(7) 每个计算节点将其第 i 段的数据发送给第 i 个计算节点，从而使第 i 个计算节点含有所有

计算节点的第*i*段数据 ($i = 0, 1, 2, \dots, p-1$)。

(8) 每个计算节点将上一步得到的*p*段有序数据序列进行多路归并排序, 得到有序数列 $Y_0^i, Y_1^i, \dots, Y_{m_i-1}^i$, m_i 为该有序数列的长度。到这里, 按节点序号的顺序及各计算节点内的数据顺序, 即可得到全部*n*个数据的排序。

(9) 得到每个计算节点的 m_i , 并汇总到第 0 个计算节点, 然后由第 0 个计算节点将汇总结果 m_0, m_1, \dots, m_{p-1} 分发到各个计算节点。

(10) 每个计算节点将每个数据结构体 Y_k^i 中的第三项 (即秩值项) 赋初始值为:

$$\left(\sum_{j=0}^{i-1} m_j \right) + k + 1$$

(11) 在每个计算节点内调整秩 (Rank) 值。对每个相邻的且其第一项数值相等的结构体集合求其秩 (Rank) 值的平均, 然后赋此平均值给该集合的各数据结构体的秩值项。

(12) 在相邻的计算节点间调整秩值。第*i*个计算节点上与 $Y_{m_i-1}^i$ 的第一项数值 x_b^i 相等的最大的数据结构体集合为 $Y_{m_i-b_i}^i, \dots, Y_{m_i-1}^i$, 由上一步的操作可知, 它们的秩值相同, 记为 r_b^i ; 第*i*+1个计算节点上与 Y_0^{i+1} 的第一项数值 x_a^{i+1} 相等的最大的数据结构体集合为 $Y_0^{i+1}, \dots, Y_{a_i-1}^{i+1}$, 由上一步的操作可知, 它们的秩值相同, 记为 r_a^{i+1} 。

第*i*个计算节点将 x_b^i 、 r_b^i 和 b_i 传给第*i*+1个计算节点。同时, 接收第*i*+1个计算节点的 x_a^{i+1} 、 r_a^{i+1} 和 a_{i+1} 。如果 $x_b^i \neq x_a^{i+1}$, 则不需要做任何调整。反之, 如果 $x_b^i = x_a^{i+1}$, 则第*i*个计算节点需将 $Y_{m_i-b_i}^i, \dots, Y_{m_i-1}^i$ 各自的秩值调整为:

$$\frac{r_b^i \cdot b_i + r_a^{i+1} \cdot a_{i+1}}{b_i + a_{i+1}}$$

同时, 第*i*+1个计算节点需将 $Y_0^{i+1}, \dots, Y_{a_i-1}^{i+1}$ 各自的秩值也调整为:

$$\frac{r_b^i \cdot b_i + r_a^{i+1} \cdot a_{i+1}}{b_i + a_{i+1}}$$

(13) 每个计算节点根据数据结构体 Y_k^i 中的第二项 (即在初始数据中的位置*q*), 计算出其初始化时属于计算节点的标号*j*, 对应关系如下:

$$j = \begin{cases} \left\lfloor \frac{q}{\left\lfloor \frac{n}{p} \right\rfloor + 1} \right\rfloor, & q < \left(\left\lfloor \frac{n}{p} \right\rfloor + 1 \right) \cdot (n \% p) \\ (n \% p) + \left\lfloor \frac{q - \left(\left\lfloor \frac{n}{p} \right\rfloor + 1 \right) \cdot (n \% p)}{\left\lfloor \frac{n}{p} \right\rfloor} \right\rfloor, & q \geq \left(\left\lfloor \frac{n}{p} \right\rfloor + 1 \right) \cdot (n \% p) \end{cases}$$

按 $j = 0, 1, 2, \dots, p-1$, 将自己保存的有序数据 $Y_0^i, Y_1^i, \dots, Y_{m_i-1}^i$ 分为*p*段。

(14) 每个计算节点将其第*i*段的数据发送给第*i*个计算节点, 从而使第*i*个计算节点仅包含初始化时的数据。

(15) 每个计算节点将数据结构体按第二项进行排序, 这样就得到了与初始时相同的顺序。而每个结构体的第三项 (秩值项) 都已在前面的步骤中被计算出来。

(16) 每个计算节点将秩值输出。

2.3.2 不需排序就可计算的次序统计量

我们在实际计算极大值和极小值的时候,并不需要全部的数据排列好来得到全部的次序统计量。下面进一步扩展极值的概念,介绍最大的前 k 个值和最小的前 k 个值。

能否通过读取一遍数据同时获取最大的前 K 个值和最小的前 K 个值呢?当处理海量数据时,将全部数据存放在内存中的代价很大,能否只用最小的内存来实现呢?

答案是肯定的。可以通过同时维护两个队列,一个是按从大到小的顺序排列,每次读取到新的数据后按顺序加入到队列中,然后只保留前 k 个数,将多出的数据删除。这样,这个数列最多需容纳 $k+1$ 个数,而且,最终数列的第一个数就是全部数据的极大值。另一个队列是按照从小到大的顺序排列,也是每次读取到新的数据后按顺序加入到队列中,然后只保留前 k 个数,将多出的数据删除,而且,最终数列的第一个数就是全部数据的极小值。

参考 Java 代码如下:

```
public void calcOrderStatistics(double[] data) {
    int small_k = 3;
    int large_k = 3;
    PriorityQueue<Double> priQueue_s
        = new PriorityQueue<Double>(small_k + 1, new DoubleComparator(-1));

    PriorityQueue<Double> priQueue_l
        = new PriorityQueue<Double>(large_k + 1, new DoubleComparator(1));

    for (double d : data) {
        if (priQueue_l.size() < large_k) {
            priQueue_l.add(d);
        } else {
            priQueue_l.add(d);
            priQueue_l.poll();
        }
        if (priQueue_s.size() < small_k) {
            priQueue_s.add(d);
        } else {
            priQueue_s.add(d);
            priQueue_s.poll();
        }
    }

    int large = priQueue_l.size();
    int small = priQueue_s.size();
}
```

```
double[] kLargeReal = new double[large];
double[] kSmallReal = new double[small];
for (int i = 0; i < small; i++) {
    kSmallReal[small - i - 1] = priQueue_s.poll();
}
for (int i = 0; i < large; i++) {
    kLargeReal[large - i - 1] = priQueue_l.poll();
}

double min = kSmallReal[0];
double max = kLargeReal[0];
double range = max - min;

System.out.print("Min : ");
System.out.println(min);
System.out.print("Max : ");
System.out.println(max);
System.out.print("Range: ");
System.out.println(range);
System.out.println();
System.out.println("Top " + kLargeReal.length + " Values:");
for (double d : kLargeReal) {
    System.out.println(d);
}
System.out.println();
System.out.println("Bottom " + kSmallReal.length + " Values:");
for (double d : kSmallReal) {
    System.out.println(d);
}
}

class DoubleComparator implements Comparator {

    int sortKey = 1;

    DoubleComparator(int sortKey) {
        this.sortKey = sortKey;
    }

    public int compare(Object o1, Object o2) {
        return (sortKey) * (((Double) o1).compareTo((Double) o2));
    }
}
```

2.3.3 基于频数信息计算次序统计量

在很多情况下，数据不同元素的个数会远小于数据的总数，若已求出频率，则可以直接对频率中的元素排序，由其结果可知在整体数据都参加排序时，某个位置对应的数值。具体步骤如下。

(1) 对频率对按元素由小到大的顺序排序（若已排好序，可省略此步），设排好序的频率对为(items[i], counts[i])。

(2) 对每个元素 items[i]计算 cntScan[i]，其数值上等于 count[0],count[1],...,count[i-1]的和，实际意义是将全部数据由小到大排序，第一次出现元素 items[i]时对应的位置。为方便编程，设位置的序号从 0 开始。

(3) 对于要求的分位数，首先计算出对应排好序的数列的位置 k ，利用 cntScan，使用二分法搜索出相应的数据元素编号 j ，则 items[j]即为所求。

下面为由频率信息计算 q -分位数的 Java 参考代码。

```
public class Quantile {

    public final Class dataType;
    public final int q;
    public Object[] items = null;

    /**
     * 构造函数
     *
     * @param q          分位数个数，即指定 q-分位数
     * @param dataType   数据类型
     */
    Quantile(int q, Class dataType) {
        this.q = q;
        this.dataType = dataType;
        this.items = new Object[q + 1];
    }

    public Object getQuantile(int k) {
        if (k < 0 || k > q) {
            throw new RuntimeException();
        }
        return items[k];
    }

    public static Quantile fromFreqSet(int q, Class dataType, TreeMap<Object,
Long> freq) {
        Quantile qtl = new Quantile(q, dataType);
        long n = 0;
```

```

        long[] cntScan = new long[freq.size()];
        Object[] item = new Object[freq.size()];
        int k = 0;
        Iterator<Entry<Object, Long>> it = freq.entrySet().iterator();
        while (it.hasNext()) {
            Entry<Object, Long> e = it.next();
            item[k] = e.getKey();
            n += e.getValue().longValue();
            cntScan[k] = n - 1;
            k++;
        }
        if (n <= 0) {
            throw new RuntimeException();
        }
        double t = 0.0;
        for (int i = 0; i <= q; i++) {
            t = 1.0 * i / q;
            if (t < 0.0) {
                qtl.items[i] = item[0];
            } else if (t >= 1.0) {
                qtl.items[i] = item[item.length - 1];
            } else {
                qtl.items[i] = item[getItemIndex(cntScan, (int) ((n - 1) * t))];
            }
        }

        return qtl;
    }

    private static int getItemIndex(long[] cntScan, long k) {
        int low = 0;
        int high = cntScan.length - 1;
        int cur;
        if (k > cntScan[high]) {
            throw new RuntimeException();
        }
        if (k <= cntScan[0]) {
            return 0;
        }
        while (low < high - 1) {
            cur = (low + high) / 2;
            if (k <= cntScan[cur]) {
                high = cur;
            } else {

```

```

        low = cur;
    }
}
return high;
}

@Override
public String toString() {
    java.io.CharArrayWriter cw = new java.io.CharArrayWriter();
    java.io.PrintWriter pw = new java.io.PrintWriter(cw, true);
    for (int i = 0; i <= q; i++) {
        pw.println(i + "-th element of " + q + "-Quantile : " + items[i]);
    }
    return cw.toString();
}

public String __repr__() {
    return toString();
}
}

```

2.3.4 中位数、众数和均值的关系

中位数、众数和均值都是描述数据集中趋势的统计量，它们各有特点。例如，对于某种商品的各种售价，中位数为处在中间的价格，大于和小于中位数的价格各为一半；众数为众多价格中出现频数最多的那个价格；而均值在大部分情况下，数值上不会等于其中的任何一个价格，但是将所有价格都放在数轴上，均值则刚好位于平衡点，即所有价格的重心上，该点两侧的力矩是相等的，恰好使数轴保持平衡。

当数据为单峰的对称分布时，其中位数、众数与均值是相同的。但如果是单峰的偏态分布，则在均值的两侧，数据的个数不同。显然，中位数会在数据个数较多的一侧；由于均值位于平衡点，两侧的力矩相等，则数据个数较多的一侧，每个点相对于均值的力矩（即距离）要小一些。也就是说，数据较多的一侧分布在较小的区间里，更容易出现频数较大的数据（众数）。所以中位数和众数会出现在均值的同侧。

下面利用皮尔森（K. Pearson）经验公式给出更加准确的关系描述。

中位数（median）一般介于均值（mean）和众数（mode）之间，且众数近似地等于 3 倍的中位数减去 2 倍的均值，即

$$\text{mode} = 3 \text{ median} - 2 \text{ mean}$$

还可以进一步得到如下两个公式。

- 将等式两端同时减去mean，得到

$$\text{mode} - \text{mean} = 3 (\text{median} - \text{mean})$$

这说明众数与均值的距离约等于中位数与均值距离的 3 倍。

- 将等式两端同时减去median，得到

$$\text{mode} - \text{median} = 2 (\text{median} - \text{mean})$$

这说明众数与中位数的距离约等于中位数与均值距离的 2 倍。

单变量数据的分布

当数据只有几十个的时候，我们扫一眼，心中就有了大概的印象；当数据达到 1 千条、1 万条、……、1 亿条、10 亿条的时候，使用一些形象化的工具就会给我们很大的帮助。本章首先介绍针对大数据的直方图算法，其特点有如下两个。

- 对数据计算一次，便可得到总体的直方图，并可以选择区间看局部数据的直方图，而且在查看总体及各部分直方图的过程中，不需要再次进行计算，在用户的客户端可以没有停顿地进行操作。
- 在直方图的计算过程中，对目标数据只需读取一遍。

基于上面直方图的计算结果，可以得到统计上常用的经验分布函数，对其分位数及百分位数的估计，以及统计上判别数据分布是否符合相同分布的 PP 图和 QQ 图。

第 2 章和本章介绍的内容都是针对单变量数据的统计量计算，且可以通过读取一次目标数据计算得到。我们定义了一个类，将这些结果汇总到一起便于计算，从而获取统计量。

3.1 直方图

直方图（Histogram）是用一组无间隔、等宽、底端对齐的直条表现数据分布特征的统计图形，每个直条代表相应区间的数据频数。

常用直方图对数据分布进行描述。如图 3-1 所示为对某数据分布的直方图表示，区间的宽度为 80。由图 3-1 可知，数据多集中在[480,560)和[0,80)两个区间。

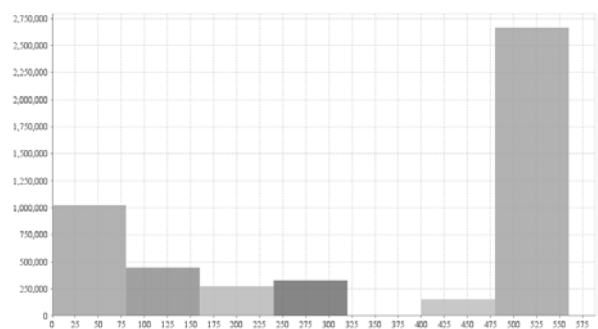


图 3-1

我们将区间的宽度设为 20，可以看到数据较多的两个区间[500,520)和[0,20)优势非常明显，如图 3-2 所示。

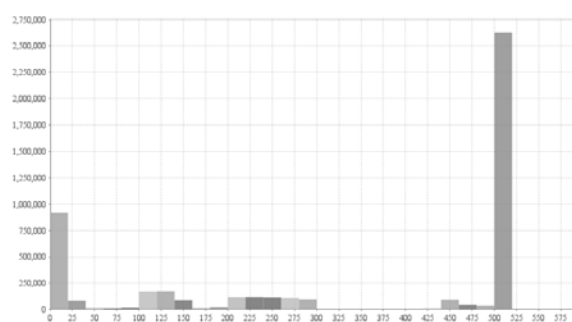


图 3-2

接下来，我们专注于数据最多的区间[500,520)，看一下这个区间的分布，在区间[510,512)上集中了绝大部分数据，如图 3-3 所示。

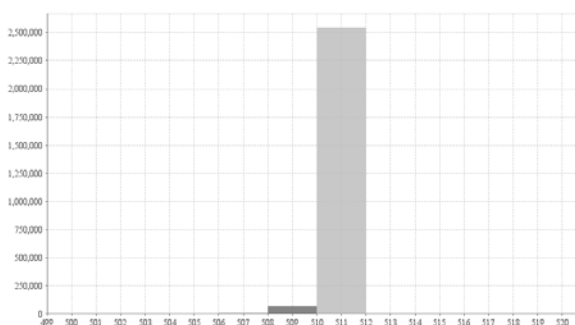


图 3-3

在同样的区间，将间隔调整为 0.1，我们得到了更清晰的认识，如图 3-4 所示。

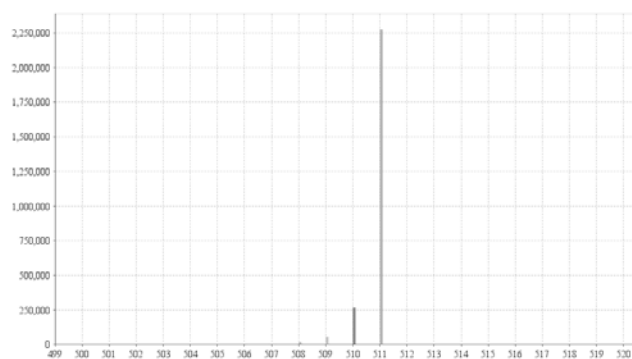


图 3-4

然后，我们将目光转向区间 $[0,20)$ ，其分布与区间 $[500,520)$ 完全不同，是对数分布的形状，如图 3-5 所示。

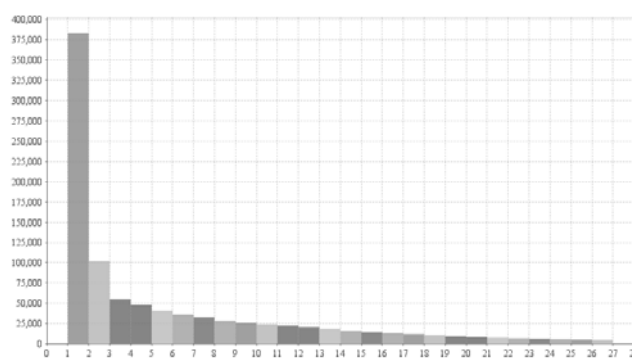


图 3-5

再将间隔调整为 0.1，可以看到数据都在整数点附近，如图 3-6 所示。

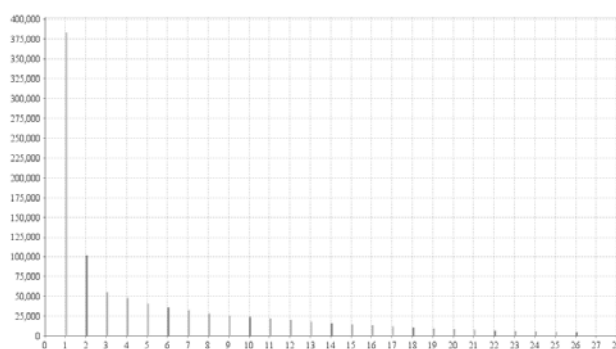


图 3-6

通过上面的例子可知直方图对了解数据分布的作用。不同的粒度（区间宽度）会给我们带来不同的信息，而找出其中感兴趣的若干区域，各个区域的分布也可能各有特点。

3.1.1 直方图的计算

计算直方图的一般步骤如下。

- (1) 计算数据的最大值和最小值，得到极差，即数据的最大值减去最小值。
- (2) 确定直方图的组数，然后以此组数去除极差，可得到直方图每组的宽度，即组距。
- (3) 确定各组的界限值，分组时应把所有的数据表都包括在内。
- (4) 统计各组的频数。

当数据量比较小时，每次执行计算的时间很短，用户可以连续变换显示粒度，切换到感兴趣的区间，而没有明显的停顿感觉，但数据量增大，计算时间变长，使用体验就会下降。对于存在分布式存储系统的大数据，每次计算需要用几分钟才能看到新的直方图结果。

对于大数据，能否用最少的数据读取量给用户流畅的直方图查看功能？答案是肯定的，而且数据仅读取一次！

新算法的核心思想是：计算一个粒度非常小的直方图（称为基本直方图），每次用户的直方图计算请求都会变成将基本直方图的若干相邻区间合并为一个新的显示区间。

新算法分如下两个部分。

- 通过自适应的方式读取一遍数据，得到基本直方图。
- 由基本直方图计算不同区间和不同粒度的直方图。

下面将详细介绍这两部分内容。

1. 计算基本直方图

在计算基本直方图时，首先要确定基本直方图的建议组数，建议组数并不是要求计算出来的基本直方图的组数与其完全相等，而是希望在一个量级上。实际的计算中，建议组数常取 10000，得到的基本直方图的分组数会在 10000 和 100000 之间。

然后关注另一个重要问题：数据只读取一遍。在一般的直方图算法中，需要统计读取一遍数据，得到极差（最大值减去最小值），才能确定数据范围和组距，然后需要再读取一遍数据，得到结果。若数据只能一次读取，需要去掉事先计算数据范围和组距这一步，通过当前已读取的数据，确定当前的数据范围和组距，在增加新数据时，自适应地调整数据范围和组距。

若要做到自适应，组距的选取很关键。不断新增数据时，数据范围也会不断变大，组数会超过我们可以处理的个数，这时就要通过增大组距的方式，将组数降下来。这个增大的新组距一定要是原来组距的整数倍，这样，某个新分组的数据个数是若干个小分组的数据个数的加和；新组距还要考虑到分组边界为比较整齐的数，例如：0.1、0.002、1、100 等，这样符合用户一般的习惯。对数值型的数据，我们选取基本组距为 10^k ，其中的 k 为整数，新的组距一定为原组距的 10^m 倍，其中的 m 为正整数。

自适应基本直方图计算的步骤如下。

- (1) 设定建议组数 N 。
- (2) 读取一个或若干个数据，建立一个初始分组。

(3) 读取新的数据（一个或多个）。

(4) 如果现在的区间划分不能包含全部的新数据，则需要按如下步骤进行调整。

1) 组距不变，增加分组个数，使之能包含全部新的数据。

2) 若分组个数超过 $10N$ ，则需对组距升级（乘以 10、100、1000...），同时也要调整区间划分的左边界，使之成为新组距的倍数，最终得到的新区间划分要满足：分组个数大于或等于 N ，且小于 $10N$ ，并确定每个新分组的频数。

(5) 判断每个新数据所属的分组，增加相应分组的频数。

(6) 若还有数据需要处理，则回到第（3）步。

(7) 得到基本直方图的区间划分和频数。

注：可以使每次获取的新的数据量多一些，从而减少区间划分调整的次数，提高运行效率。

举例：对于分组 $[10, 10.1)$, $[10.1, 10.2)$, $[10.2, 10.3)$, \dots , $[99.9, 100)$ ，相应的频数为 $\{2, 2, 2, \dots, 2\}$ ，建议组数 $N=10$ 。下面我们读取新的数据，并相应地调整分组和频数。

(1) 读到一个新数据 10.15，因为它属于已有的区间 $[10.1, 10.2)$ ，则分组不变，相应的频数为 $\{2, 3, 2, \dots, 2\}$ 。

(2) 再读到一个新数据 9.85，则分组需要增加 $[9.8, 9.9)$ 和 $[9.9, 10)$ ，其中， $[9.9, 10)$ 是因为直方图的区间是连续的，而增加的空区间最终的分组为 $[9.8, 9.9)$, $[9.9, 10)$, $[10, 10.1)$, $[10.1, 10.2)$, $[10.2, 10.3)$, \dots , $[99.9, 100)$ ，相应的频数为 $\{1, 0, 2, 3, 2, \dots, 2\}$ 。

(3) 再读到一个新数据 -1.0，则分组需要按 10^2 升级为 $[0, 10)$, $[10, 20)$, $[20, 30)$, \dots , $[90, 100)$ ，相应的频数为 $\{1, 201, 200, \dots, 200\}$ ；然后增加 $[-10, 0)$ ，最终的分组为 $[-10, 0)$, $[0, 10)$, $[10, 20)$, $[20, 30)$, \dots , $[90, 100)$ ，相应的频数为 $\{1, 1, 201, 200, \dots, 200\}$ 。

对于分布式计算，每个计算节点都会负责处理一部分数据，可执行上面的自适应算法得到各部分数据的基本直方图划分，然后，我们需要一个汇总过程，得到对于全部数据的基本直方图划分，计算步骤如下。

(1) 获取各部分数据基本直方图划分的左右边界，比较得出左边界的最小值和右边界的最大值，将它们作为全部数据的取值范围。

(2) 根据此全部数据的取值范围，及建议组数 N ，计算出针对全部数据的组距，从而得到对于全部数据的区间划分。

(3) 对每部分的数据基本直方图划分，将其每个分组位置和频数汇总加入到全部数据区间划分中的对应分组。

(4) 经过上面三步，得到全部数据基本直方图的区间划分和频数。

2. 按需要的区间和粒度生成直方图

在实际计算中，我们通常选取建议的组数为 10000，计算出的基本直方图会有几万个分组，用户无法直接从中获得信息，还需以直方图的方式展示出来，展示的内容如下。

-
- 用户可以选择一个感兴趣的数据范围，默认为全体数据。
 - 默认的显示一般是显示十几个区间，每个区间包含的基本区间个数要尽量相等。
 - 在当前绘图的精度下，最细粒度的显示。
 - 可以进行更细粒度或更粗粒度的显示，即放大（zoom in）和缩小（zoom out）效果。
 - 按用户输入的粒度进行展示。
 - 对于长尾类型的，可以选择不显示某些区间，便于查看那些频数小的区间。
 - 显示的每个区间所包含的频数，及在当前显示范围内所占总频数的百分比。

上面的各种操作最终都会转化为这样一个问题：对于一个建议左边界值、建议右边界值和建议步长，求显示直方图。

具体计算步骤如下。

(1) 计算显示步长，它需要满足两个条件：是基本直方图的组距的正整数倍，等于或接近建议步长。

(2) 计算显示左边界，同样要满足两个条件：是显示步长的整数倍，等于或小于建议左边界值。

(3) 计算显示右边界，同样要满足两个条件：是显示步长的整数倍，等于或大于建议右边界值。

(4) 由显示步长、显示左边界和显示右边界，得到显示区间划分。

(5) 对每个显示区间，计算属于该区间的基本直方图分组的频数和，作为该显示区间的频数。

(6) 输出整个显示区间划分和频数。

3.1.2 算法实现

将前面介绍的基本直方图计算及区间合并的功能包装成类 `XInterval` 的方法和函数。详细的 Java 代码如下。

```
import java.math.BigDecimal;
import java.math.BigInteger;

public class XInterval implements Cloneable {

    private final static int DefaultMagnitude = 1000;
    private long start;
    private BigDecimal step;
    private int n;
    private long[] count = null;
    private int magnitude; // magnitude < n <= 10 * magnitude

    private XInterval(long start, BigDecimal step, long[] count, int magnitude)
    {
```

```

        if ((10 * (long) magnitude > Integer.MAX_VALUE) || (magnitude < 1)) {
            throw new RuntimeException();
        } else {
            this.magnitude = magnitude;
        }
        this.start = start;
        this.step = step;
        this.n = count.length;
        this.count = count;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        XInterval sd = (XInterval) super.clone();
        sd.count = this.count.clone();
        return sd;
    }

    public double getLeftBound() {
        return BigDecimal.valueOf(start).multiply(step).doubleValue();
    }

    public double getStep() {
        return this.step.doubleValue();
    }

    public long[] getCount() {
        return this.count.clone();
    }

    @Override
    public String toString() {
        StringBuilder sbd = new StringBuilder();
        sbd.append("start=" + start + ", step=" + step + ", n=" + n + ", magnitude="
+ magnitude + '\n');
        for (int i = 0; i < n; i++) {
            sbd.append("count[" + i + "] = " + count[i] + "\n");
        }
        return sbd.toString();
    }

    public void countValue(double val) {
        while (!hasValidIntervalVal(val)) {
            upgrade();
        }
    }

```

```

    }
    long k = toIntervalVal(val);
    if ((k >= start) && (k < start + n)) {
        // val 在区间内
        count[(int) (k - start)]++;
        return;
    } else {
        // val 不在区间内，需要重新设计区间分划
        long min = start;
        long max = start + n;
        if (k < start) {
            min = k;
        } else {
            max = k + 1;
        }

        upgrade(min, max);

        long kk = toIntervalVal(val);
        count[(int) (kk - start)]++;
        return;
    }
}

public void countValue(double[] vals) {
    if (null == vals || vals.length == 0) {
        return;
    }

    double minVal = vals[0];
    double maxVal = vals[0];
    for (int i = 0; i < vals.length; i++) {
        if (minVal > vals[i]) {
            minVal = vals[i];
        }
        if (maxVal < vals[i]) {
            maxVal = vals[i];
        }
    }

    while (!hasValidIntervalVal(minVal)) {
        upgrade();
    }

    while (!hasValidIntervalVal(maxVal)) {

```

```

        upgrade();
    }

    long min = toIntervalVal(minVal);
    long max = toIntervalVal(maxVal);
    if ((min >= start) && (max < start + n)) {
        // val 在区间内
        for (int i = 0; i < vals.length; i++) {
            long k = toIntervalVal(vals[i]);
            count[(int) (k - start)]++;
        }
        return;
    } else {
        // val 不在区间内，需要重新设计区间分划
        min = Math.min(min, start);
        max = Math.max(max + 1, start + n);

        upgrade(min, max);

        for (int i = 0; i < vals.length; i++) {
            long k = toIntervalVal(vals[i]);
            count[(int) (k - start)]++;
        }
        return;
    }
}

private void upgrade() {
    this.step = this.step.multiply(new BigDecimal(10));
    long startNew = divideInt(start, 10);
    long endNew = divideInt(start + n + 9, 10);
    int nNew = (int) (endNew - startNew);
    long[] countNew = new long[nNew];
    for (int i = 0; i < n; i++) {
        countNew[(int) ((i + start) / 10 - startNew)] += count[i];
    }
    this.start = startNew;
    this.n = nNew;
    this.count = countNew;
}

private void upgrade(long min, long max) {

    long scale = getScale4Upgrade(min, max);

```

```

        this.step = this.step.multiply(new BigDecimal(scale));
        long startNew = divideInt(min, scale);
        long endNew = divideInt(max + scale - 1, scale);
        int nNew = (int) (endNew - startNew);
        long[] countNew = new long[nNew];
        for (int i = 0; i < n; i++) {
            countNew[(int) ((i + start) / scale - startNew)] += count[i];
        }
        this.start = startNew;
        this.n = nNew;
        this.count = countNew;
    }

    private long getScale4Upgrade(long min, long max) {
        if (min > max) {
            throw new RuntimeException();
        }
        long scale = 1;
        int i = 0;
        long k = 0;
        long start = 0;
        long end = 0;
        for (; i < 20; i++) {
            start = divideInt(min, scale);
            end = divideInt(max + scale - 1, scale);
            k = end - start;
            if (k <= this.magnitude * 10) {
                break;
            }
            scale = scale * 10;
        }
        return scale;
    }

    private long divideInt(long k, long m) {
        if (k >= 0) {
            return k / m;
        } else {
            return (k - m + 1) / m;
        }
    }

    private static BigInteger getIntervalValBD(double val, BigDecimal stepBD)

```

```

{
    BigDecimal valBD = new BigDecimal(val);
    BigInteger kBD = valBD.divide(stepBD).toBigInteger();
    if (valBD.subtract(stepBD.multiply(new BigDecimal(kBD))).signum() < 0)
    {
        kBD = kBD.subtract(BigInteger.ONE);
    }
    return kBD;
}

private boolean isValidIntervalVal(double val) {
    BigInteger k = getIntervalValBD(val, this.step);
    return BigInteger.valueOf(k.longValue()).subtract(k).signum() == 0;
}

private long toIntervalVal(double val) {
    BigInteger k = getIntervalValBD(val, this.step);
    if (BigInteger.valueOf(k.longValue()).subtract(k).signum() != 0) {
        // 有精度损失
        throw new RuntimeException();
    }
    return k.longValue();
}

public static XInterval createXInterval(double[] vals) {
    return createXInterval(vals, XInterval.DefaultMagnitude);
}

public static XInterval createXInterval(double[] vals, int magnitude) {
    if (null == vals || vals.length == 1) {
        throw new RuntimeException();
    }

    double minVal = vals[0];
    double maxVal = vals[0];
    for (int i = 0; i < vals.length; i++) {
        if (minVal > vals[i]) {
            minVal = vals[i];
        }
        if (maxVal < vals[i]) {
            maxVal = vals[i];
        }
    }
    XInterval xi = getEmptyInterval(minVal, maxVal, magnitude);
}

```

```

        xi.countValue(vals);

        return xi;
    }

    public static XInterval getEmptyInterval(double min, double max) {
        return getEmptyInterval(min, max, XInterval.DefaultMagnitude);
    }

    public static XInterval getEmptyInterval(double min, double max, int
magnitude) {
        if (Double.NEGATIVE_INFINITY < min && min <= max && max <
Double.POSITIVE_INFINITY) {
            int k = -300; //double 类型的最小精度
            if (0 != min && 0 != max) {
                int k1 = (int) Math.log10(Math.abs(min) + Math.abs(max));
                k = Math.max(k, k1 - 19); //long 型数据大约 19 个有效数字

                if (min != max) {
                    int k2 = (int) (Math.log10(max - min) - Math.log10 (magnitude));
                    k = Math.max(k, k2);
                }
            }
            BigDecimal stepBD = new BigDecimal(1);
            if (k > 1) {
                stepBD = BigDecimal.TEN.pow(k - 1);
            } else if (k <= 0) {
                stepBD = new BigDecimal(1).divide(BigDecimal.TEN.pow(1 - k));
            }

            return new XInterval(getIntervalValBD(min, stepBD).longValue(),
stepBD, new long[]{0}, magnitude);

        } else {
            throw new RuntimeException();
        }
    }

    public static XInterval mergeXInterval(XInterval ia, XInterval ib) throws
CloneNotSupportedException {
        if (ia.magnitude != ib.magnitude) {
            throw new RuntimeException("Two merge XInterval must have same magnitude!");
        }
        XInterval x = null;

```

```
XInterval y = null;
if (ia.step.subtract(ib.step).signum() > 0) {
    x = (XInterval) ia.clone();
    y = (XInterval) ib.clone();
} else {
    x = (XInterval) ib.clone();
    y = (XInterval) ia.clone();
}

while (x.step.subtract(y.step).signum() > 0) {
    y.upgrade();
}

long min = Math.min(x.start, y.start);
long max = Math.max(x.start + x.n, y.start + y.n);

x.upgrade(min, max);
y.upgrade(min, max);

for (int i = 0; i < x.n; i++) {
    x.count[i] += y.count[i];
}

return x;
}
}
```

3.1.3 已知数据频数的情况下求直方图

从前面的内容我们知道，当数据中相同的数值很多，且出现的不同数值个数有限（如小于10000）时，我们只需读取一遍数据就可以得到数据频数。有数据频数后，我们可以直接由它计算直方图。

在已知数据频数的情况下，对于给定的显示步长、建议左边界值和建议右边界值，求显示直方图，步骤如下。

- （1）计算显示左边界，同样要满足两个条件：是显示步长的整数倍，等于或小于建议左边界值。
- （2）计算显示右边界，同样要满足两个条件：是显示步长的整数倍，等于或大于建议右边界值。
- （3）由显示步长、显示左边界和显示右边界，得到显示区间划分。
- （4）对每个显示区间，计算属于该区间的数据的频数和，作为该显示区间的频数。

(5) 输出整个显示区间划分和频数。

3.1.4 日期类型直方图

日期类型的数据特点如下。

- 从秒到分钟，需要将属于此分钟的 60 个秒区间统一起来；从分钟到小时，也需要将属于同一小时的 60 个分钟区间整合；从小时到天，需要将同一天的 24 个小时区间整合。
- 在分布式系统里存储的时候，是用毫秒记录的。从 1970 年 1 月 1 日起的毫秒的数量表示日期。也就是说，例如，1970 年 1 月 2 日是在 1 月 1 日后的 $1000 \times 60 \times 60 \times 24 = 86400000$ 毫秒。同样，1969 年 12 月 31 日是在 1970 年 1 月 1 日前的 86400000 毫秒。我们用有符号的长整型记录这些毫秒值，所以日期可以在 1970 年 1 月 1 日之前，也可以在这之后。

长整型表示的最大正值和最大负值为

$$\frac{2^{63}}{86400000 \times 365} = 292471208.68$$

所以，该日期表示方法可表示为 1970 年之前的 2.9 亿年，到 1970 年后的 2.9 亿年，即公元前 2.9 亿年到公元 2.9 亿年，这适合绝大多数人的时间要求。

- 人们常以 10 秒钟、半小时等作为直方图的区间。

日期类型的直方图的实现过程与前面普通的直方图基本相同，主要的区别在于区间升级时的尺度是不等的。另外，由于时间用整数表示，相对于浮点数的表示和计算的误差处理要简单一些，程序实现也做了相应的优化。

参考 Java 代码如下：

```
public class IntervalDateCalculator implements Cloneable {

    public static final long[] stepDate = new long[]{
        1, // 1 millisecond
        10,
        10 * 10,
        10 * 10 * 10, // 1 sec
        10 * 10 * 10 * 10,
        10 * 10 * 10 * 10 * 6, // 1 min
        10 * 10 * 10 * 10 * 6 * 10,
        10 * 10 * 10 * 10 * 6 * 10 * 3, // half an hour
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2, // 1 hour
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6,
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2, // half a day
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2 * 2, // 1 day
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2 * 2 * 10, // 10 day
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2 * 2 * 100, // 100 day
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2 * 2 * 1000, // 1000 day
    };
```

```

        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2 * 2 * 10000, //10000 day
        10 * 10 * 10 * 10 * 6 * 10 * 3 * 2 * 6 * 2 * 2 * 100000, //100000 day
    };

    private final static int DefaultMagnitude = 1000;
    long start;
    long step;
    int n;
    long[] count = null;
    int magnitude; // magnitude < n <= 10 * magnitude

    private IntervalDateCalculator(long start, long step, long[] count, int
magnitude) {
        if ((10 * (long) magnitude > Integer.MAX_VALUE) || (magnitude < 1)) {
            throw new RuntimeException();
        } else {
            this.magnitude = magnitude;
        }
        this.start = start;
        this.step = step;
        this.n = count.length;
        this.count = count;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        IntervalDateCalculator sd = (IntervalDateCalculator) super.clone();
        sd.count = this.count.clone();
        return sd;
    }

    public long getLeftBound() {
        return start * step;
    }

    public long getStep() {
        return this.step;
    }

    public long[] getCount() {
        return this.count.clone();
    }

    @Override
    public String toString() {

```

```
        StringBuilder sbd = new StringBuilder();
        sbd.append("start=" + start + ", step=" + step + ", n=" + n + ", magnitude="
+ magnitude + '\n');
        for (int i = 0; i < n; i++) {
            sbd.append("count[" + i + "] = " + count[i] + "\n");
        }
        return sbd.toString();
    }

    public void calculate(long val) {
        long k = toIntervalVal(val);
        if ((k >= start) && (k < start + n)) {
            // val 在区间内
            count[(int) (k - start)]++;
            return;
        } else {
            // val 不在区间内，需要重新设计区间分划
            long min = start;
            long max = start + n;
            if (k < start) {
                min = k;
            } else {
                max = k + 1;
            }

            upgrade(min, max);

            long kk = toIntervalVal(val);
            count[(int) (kk - start)]++;
            return;
        }
    }

    public void calculate(long[] vals) {
        if (null == vals || vals.length == 0) {
            return;
        }

        long minVal = vals[0];
        long maxVal = vals[0];
        for (int i = 0; i < vals.length; i++) {
            if (minVal > vals[i]) {
                minVal = vals[i];
            }
        }
    }
}
```

```
        if (maxVal < vals[i]) {
            maxVal = vals[i];
        }
    }

    long min = toIntervalVal(minVal);
    long max = toIntervalVal(maxVal);
    if ((min >= start) && (max < start + n)) {
        // val 在区间内
        for (int i = 0; i < vals.length; i++) {
            long k = toIntervalVal(vals[i]);
            count[(int) (k - start)]++;
        }
        return;
    } else {
        // val 不在区间内，需要重新设计区间分划
        min = Math.min(min, start);
        max = Math.max(max + 1, start + n);

        upgrade(min, max);

        for (int i = 0; i < vals.length; i++) {
            long k = toIntervalVal(vals[i]);
            count[(int) (k - start)]++;
        }
        return;
    }
}

public void calculate(Date date) {
    calculate(date.getTime());
}

public void calculate(Date[] dates) {
    long[] vals = new long[dates.length];
    for (int i = 0; i < dates.length; i++) {
        vals[i] = dates[i].getTime();
    }
    calculate(vals);
}

private long getLargerStep() {
    for (int i = 0; i < stepDate.length; i++) {
        if (stepDate[i] > this.step) {
```

```
        return stepDate[i];
    }
}

if (this.step * 10 > this.step) {
    return this.step * 10;
} else {
    return Long.MAX_VALUE;
}
}

void upgrade() {
    long stepNew = getLargerStep();
    long scale = stepNew / this.step;
    if (scale * this.step != stepNew) {
        throw new RuntimeException();
    }
    long startNew = divideInt(start, scale);
    long endNew = divideInt(start + n + 9, scale);
    subUpgrade(scale, startNew, endNew);
}

private void upgrade(long min, long max) {
    long scale = getScale4Upgrade(min, max);
    long startNew = divideInt(min, scale);
    long endNew = divideInt(max + scale - 1, scale);
    subUpgrade(scale, startNew, endNew);
}

private void subUpgrade(long scale, long startNew, long endNew) {
    this.step = this.step * scale;
    int nNew = (int) (endNew - startNew);
    long[] countNew = new long[nNew];
    for (int i = 0; i < n; i++) {
        countNew[(int) (divideInt(i + start, scale) - startNew)] += count[i];
    }
    this.start = startNew;
    this.n = nNew;
    this.count = countNew;
}

private long getScale4Upgrade(long min, long max) {
    if (min > max) {
        throw new RuntimeException();
    }
}
```

```
        min *= this.step;
        max *= this.step;
        long s = this.step;
        long k = 0;
        long start = 0;
        long end = 0;
        for (int i = 0; i < 20; i++) {
            start = divideInt(min, s);
            end = divideInt(max + s - 1, s);
            k = end - start;
            if (k <= this.magnitude * 10) {
                break;
            }
            s = getLargerStep();
        }
        if (s == (s / this.step) * this.step) {
            return s / this.step;
        } else {
            throw new RuntimeException();
        }
    }

    private static long divideInt(long k, long m) {
        if (k >= 0) {
            return k / m;
        } else {
            return (k - m + 1) / m;
        }
    }

    private long toIntervalVal(long val) {
        return divideInt(val, this.step);
    }

    public static IntervalDateCalculator create(long[] vals) {
        return create(vals, IntervalDateCalculator.DefaultMagnitude);
    }

    public static IntervalDateCalculator create(long[] vals, int magnitude) {
        if (null == vals || vals.length == 1) {
            throw new RuntimeException();
        }

        long minVal = vals[0];
```

```
        long maxVal = vals[0];
        for (int i = 0; i < vals.length; i++) {
            if (minVal > vals[i]) {
                minVal = vals[i];
            }
            if (maxVal < vals[i]) {
                maxVal = vals[i];
            }
        }
        IntervalDateCalculator xi = getEmptyInterval(minVal, maxVal, magnitude);
        xi.calculate(vals);

        return xi;
    }

    public static IntervalDateCalculator getEmptyInterval(long min, long max)
    {
        return getEmptyInterval(min, max, IntervalDateCalculator.DefaultMagnitude);
    }

    public static IntervalDateCalculator getEmptyInterval(long min, long max,
int magnitude) {
        if (Double.NEGATIVE_INFINITY < min && min <= max && max < Double.
POSITIVE_INFINITY) {
            long k;
            long s = 1;
            long start = 0;
            long end = 0;
            for (int i = 0; i < stepDate.length; i++) {
                s = stepDate[i];
                start = divideInt(min, s);
                end = divideInt(max + s - 1, s);
                k = end - start;
                if (k <= magnitude * 10) {
                    break;
                }
            }

            return new IntervalDateCalculator(divideInt(min, s), s, new long[]
{0}, magnitude);

        } else {
            throw new RuntimeException();
        }
    }
}
```

```

    }

    public static IntervalDateCalculator merge(IntervalDateCalculator ia,
IntervalDateCalculator ib) throws CloneNotSupportedException {
        if (ia.magnitude != ib.magnitude) {
            throw new RuntimeException("Two merge XInterval must have same magnitude!");
        }
        IntervalDateCalculator x = null;
        IntervalDateCalculator y = null;
        if (ia.step > ib.step) {
            x = (IntervalDateCalculator) ia.clone();
            y = (IntervalDateCalculator) ib.clone();
        } else {
            x = (IntervalDateCalculator) ib.clone();
            y = (IntervalDateCalculator) ia.clone();
        }

        while (x.step > y.step) {
            y.upgrade();
        }

        long min = Math.min(x.start, y.start);
        long max = Math.max(x.start + x.n, y.start + y.n);

        x.upgrade(min, max);
        y.upgrade(min, max);

        for (int i = 0; i < x.n; i++) {
            x.count[i] += y.count[i];
        }

        return x;
    }
}

```

3.2 经验分布

设 X_1, X_2, \dots, X_n 是总体 X 的样本, $X \sim F(x)$, 则称

$$F_n(x) = \begin{cases} 0, & x < X_{(1)} \\ k/n, & X_{(k)} < x < X_{(k+1)} \\ 1, & x > X_{(n)} \end{cases}$$

为经验分布函数 (Empirical Distribution Function)。这里的 $F_n(x)$ 是一个跳跃函数, 其跳跃点

是样本观测值，在每个跳跃点处的跳跃度均为 $1/n$ 。

Glivenko 于 1933 年证明了对于经验分布函数有以下结果：

$$P\left\{\lim_{n \rightarrow \infty} \sup_{-\infty < x < \infty} |F_n(x) - F(x)| = 0\right\} = 1$$

这个结果表明：当 n 充分大时，经验分布函数与总体分布函数的差异很小，实际中可以用 $F_n(x)$ 近似代替 $F(x)$ 。

在大数据量的情况下，我们直接使用几十亿个数据来定义经验分布函数，在计算时间和存储空间上会面临很大的问题。能否用少量的数据来进行描述，且具有很高的精度呢？

利用基本直方图的算法，我们可得到足够多（ m 个，一般取 $10000 < m < 100000$ ）的细分区间，每个区间内包含数据的个数 m_i 来近似。数学表示如下：

$$F_n(x) = \begin{cases} 0, & x < X_{(1)} \\ \frac{1}{n} \sum_{i=1}^k m_i, & X_1 + (k-1) \cdot \frac{X_{(n)} - X_{(1)}}{m-1} x < X_1 + k \cdot \frac{X_{(n)} - X_{(1)}}{m-1} \\ 1, & \end{cases}$$

其中， $n = \sum_{i=1}^{m-1} m_i$ ， $1 \leq k \leq m-1$ 。

由此计算公式可知，具体的计算步骤如下。

(1) 求数据的基本直方图，得到区间的起始位置和步长。

(2) 对于基本直方图的细分区间，包含 m_i 个数据，计算出数据总数 $n = \sum_{i=1}^{m-1} m_i$ ，并计算出

每个比例值 $\frac{1}{n} \sum_{i=1}^k m_i$ ，然后保存。这样就得到了经验分布。

(3) 利用计算出的经验分布及上面的计算公式，可对任何 x 计算经验分布值。

参考 Java 代码如下：

```
public class EmpiricalCDF {

    int n;
    double[] f;
    double[] x;

    public EmpiricalCDF(SummaryResultColsrc) throws Exception {

        if (src.hasFreq()) {
            initFreq(src.getFrequencyOrderByItem(), src.dataType);
        } else {
```

```

        initIC(src.getIntervalCalculator());
    }
}

public EmpiricalCDF(IntervalCalculator ic) {
    initIC(ic);
}

public EmpiricalCDF(ArrayList<Entry<Object, Long>> freq, Class dataType) throws
Exception {
    initFreq(freq, dataType);
}

private void initIC(IntervalCalculator ic) {
    double start = ic.getLeftBound().doubleValue();
    double step = ic.getStep().doubleValue();
    long[] counts = ic.getCount();
    this.n = counts.length;
    this.f = new double[n];
    this.x = new double[n];
    double s = 0.0;
    for (int i = 0; i < n; i++) {
        s += counts[i];
    }
    f[i] = s;
    x[i] = start + step * i;
}

for (int i = 0; i < n; i++) {
    f[i] /= s;
}
}

private void initFreq(ArrayList<Entry<Object, Long>> freq, Class dataType)
throws Exception {
    n = freq.size();
    this.f = new double[n];
    this.x = new double[n];
    long count = 0;
    for (Entry<Object, Long> e : freq) {
        count += e.getValue().longValue();
    }
    double s = 0;
    for (int i = 0; i < n; i++) {
        Entry<Object, Long> e = freq.get(i);
        s += e.getValue().longValue();
    }
}

```

```

        f[i] = Math.min(1.0, s / count);
        if (dataType == Double.class) {
            x[i] = ((Double) e.getKey()).doubleValue();
        } else if (dataType == Long.class) {
            x[i] = ((Long) e.getKey()).longValue();
        } else {
            throw new Exception("Error in datatype of column!");
        }
    }
}

public double calculate(double val) {
    if (val < x[0]) {
        return 0;
    } else if (val >= x[n - 1]) {
        return 1;
    } else {
        int b0 = 0;
        int b1 = n - 1;
        while (1 < b1 - b0) {
            int m = (b0 + b1) / 2;
            if (x[m] > val) {
                b1 = m;
            } else {
                b0 = m;
            }
        }
        return this.f[b0];
    }
}

public XYSeries getXYSeries() {
    XYSeries xyseries = new XYSeries("Empirical Distribution Function");
    int n = this.f.length;
    for (int i = 0; i < n; i++) {
        xyseries.add(this.x[i], this.f[i]);
    }
    return xyseries;
}
}

```

3.3 近似分位数和近似百分位数

求出分位数、百分位数的数据传输量和计算量相当于进行一次排序操作，需要消耗较多的资源，花费更长的时间。但很多时候，人们并不需要一个非常精确的解，只是希望通过分位数、百分位数对数据有一个大致的印象。这里我们利用直方图的计算结果，近似地估计出分位数和百分位数，并可以给出估计值与真实值的误差，以及真实值所在的区间。

针对直方图计算所求得的细分区间，每个细分区间都有其包含的数据个数，我们虽然不知道第 i 个区间内 n_i 个数据的大小关系，但我们知道对于任何两个区间，右边区间中的任何元素会大于左边区间中的任何元素。进一步可以确定，如果进行整体排序，排序后的第 m 个元素位于第 k 个区间，当且仅当

$$\sum_{i=0}^k n_i m < \sum_{i=0}^{k+1} n_i$$

此时，对于百分位点 p ，其相对于数据整体排序后的第 m_p 个元素，由上面可以估计出它在第 k_p 个区间。接下来，由所在细分区间的中点可作为百分位数的估计，区间长度的一半便是估计的误差，区间的边界即为真实值所在的区间。

在实际计算的时候，我们很容易得到数据的最大值和最小值，也可将其加入到百分位数的估计中，使两端的 0 和 100 百分位数更精确。

近似分位数计算的具体实现代码如下：

```
public class ApproximateQuantile {

    private final int numQuantile;
    private int[] kthIndex = null;
    private double start;
    private double step;
    private double min;
    private double max;

    public ApproximateQuantile(int numQuantile, double startInterval, double
stepInterval, long[] countsInterval) {
        this(numQuantile, startInterval, stepInterval, countsInterval,
Double.NaN, Double.NaN);
    }

    public ApproximateQuantile(int numQuantile, double startInterval, double
stepInterval, long[] countsInterval, double min, double max) {
        this.numQuantile = numQuantile;
        this.start = startInterval;
        this.step = stepInterval;
        this.min = min;
```

```

        this.max = max;
        kthIndex = new int[numQuantile + 1];
        kthIndex[0] = 0;
        long total = 0;
        for (int i = 0; i < countsInterval.length; i++) {
            total += countsInterval[i];
        }
        double s = total * 1.0 / numQuantile;
        long t1 = 0;
        long t2 = countsInterval[0];
        int k = 1;
        int idxCounts = 0;
        while (idxCounts < countsInterval.length && k <= numQuantile) {
            if (t1 <= s * k && s * k <= t2) {
                kthIndex[k] = idxCounts;
                k++;
            } else {
                t1 += countsInterval[idxCounts];
                idxCounts++;
                if (idxCounts < countsInterval.length) {
                    t2 += countsInterval[idxCounts];
                } else {
                    break;
                }
            }
        }
        while (k <= numQuantile) {
            kthIndex[k] = countsInterval.length - 1;
            k++;
        }
    }

    public int getNumQuantile() {
        return this.numQuantile;
    }

    public double getQuantile(int k) {
        if (0 == k && !Double.isNaN(this.min)) {
            return this.min;
        } else if (k == numQuantile && !Double.isNaN(this.max)) {
            return this.max;
        } else {
            double r = this.start + this.step * (kthIndex[k] + 0.5);
            if (!Double.isNaN(this.max)) {

```

```
        r = Math.min(this.max, r);
    }
    if (!Double.isNaN(this.min)) {
        r = Math.max(this.min, r);
    }
    return r;
}

}

public double getError(int k) {
    if ((0 == k && !Double.isNaN(this.min)) || (k == numQuantile && !Double.
isNaN(this.max))) {
        return 0.0;
    } else {
        return this.step / 2;
    }
}

public double getLeftBound(int k) {
    if (0 == k && !Double.isNaN(this.min)) {
        return this.min;
    } else if (k == numQuantile && !Double.isNaN(this.max)) {
        return this.max;
    } else {
        return this.start + this.step * kthIndex[k];
    }
}

public double getRightBound(int k) {
    if (0 == k && !Double.isNaN(this.min)) {
        return this.min;
    } else if (k == numQuantile && !Double.isNaN(this.max)) {
        return this.max;
    } else {
        return this.start + this.step * (kthIndex[k] + 1);
    }
}

public double getLowerLimit() {
    if (!Double.isNaN(this.min)) {
        return this.min;
    } else {
        return this.start;
    }
}
```

```

    }

    public double getUpperLimit() {
        if (!Double.isNaN(this.max)) {
            return this.max;
        } else {
            return this.start + this.step * kthIndex[numQuantile];
        }
    }

    @Override
    public String toString() {
        StringBuilder sbd = new StringBuilder();
        sbd.append("Approximate " + this.numQuantile + "-Quantile:\n");
        for (int k = 0; k <= numQuantile; k++) {
            sbd.append(k + "%\t: " + this.getQuantile(k));
            sbd.append("\t in range [ " + this.getLeftBound(k) + " , " +
this.getRightBound(k) + " )\n");
        }
        return sbd.toString();
    }
}

```

百分位数的计算是分位数个数为 100 的分位数计算，其实现可以通过调用分位数计算获得，具体代码如下：

```

public class ApproximatePercentile {

    private final ApproximateQuantile aqt;

    public ApproximatePercentile(double startInterval, double stepInterval,
long[] countsInterval) {
        this(startInterval, stepInterval, countsInterval, Double.NaN, Double.
NaN);
    }

    public ApproximatePercentile(double startInterval, double stepInterval,
long[] countsInterval, double min, double max) {
        this.aqt = new ApproximateQuantile(100, startInterval, stepInterval,
countsInterval, min, max);
    }

    public double getPercentile(int k) {
        return aqt.getQuantile(k);
    }
}

```

```
public double getError(int k) {
    return aqt.getError(k);
}

public double getLeftBound(int k) {
    return aqt.getLeftBound(k);
}

public double getRightBound(int k) {
    return aqt.getRightBound(k);
}

@Override
public String toString() {
    return this.aqt.toString();
}
}
```

3.4 PP、QQ 概率图

由直方图可以看到数据的分布情况,如果我们希望进一步确定其属于什么分布呢?对于均匀分布,很好确认,只要看直方图的各个直方的顶端是否在一条直线上即可。如果是常见的正态分布呢?大致是一个“钟”型,还要对称。但这两个条件还不充分,尤其是第一条。就像几何上,我们判断点是否在一条直线上很直观,但要判断是否在一条曲线上,就很不直观。我们能一眼判断出均匀分布,是因为它可以转化为一个点共线的问题,容易直观判断。

PP、QQ 概率图 (Probability Plot) 就是这样的变换,它可以将判断是否服从正态分布也用点是否共线形象地表示出来。

使用 QQ 图时,当所有的点都在一条直线上时,说明它是服从正态分布的,该直线的截距和斜率体现了它具体服从什么样的均值和方差的正态分布。对于 PP 图,要求更严格,需要确定具体的正态分布 (指定均值和方差)。当所有的点都在对角线上时,说明它服从正态分布。显然,QQ 图的要求更低,实际应用中,QQ 图的应用更多。

1. QQ 图

QQ 图 (Q 代表分位数 Quantile) 是一种为比较两个概率分布所绘制的散点图。具体实现时,选择一系列概率值,对每个概率值分别计算其在两个分布上的分位数作为画图点的坐标。其特点如下。

- 两个分布相同时,每个点的横/纵坐标相同,所有的点位于对角线上。
- 对于两个相同的分布,其中一个进行平移或伸缩变换,则所有的结果点还是共线。这个特点适合用来做正态性的验证。

在一系列概率值的选取上，我们采用较常用的方式，若需要 n 个画图点，我们取 n 个概率值 $1/(n+1), 2/(n+1), \dots, n/(n+1)$ ，并分别对两种分布利用逆累积概率函数或样本数据的近似分位数函数来计算相应的分位数。关于概率值选取的详细讨论，可参见 Wiki (http://en.wikipedia.org/wiki/Q-Q_plot)。

2. PP 图

PP 图(P 代表分位数 Probability)是一种为比较两个概率分布所绘制的散点图。具体实现时，选择一系列采样点，对每个采样点分别计算其在两个分布上的累计概率函数值，作为画图点的坐标。PP 图的特点如下。

- 两个分布相同时，每个点的横/纵坐标相同，所有的点位于对角线上。
- 所有的结果点在 $[0, 1] \times [0, 1]$ 范围内。

在采样点的选择上，用户可以指定具体的采样点列；或者用户只输入需要画出的点的总数 n ，由程序确定具体的点列。程序会用类似 QQ 图选取概率的方法，取 n 个概率值 $1/(n+1), 2/(n+1), \dots, n/(n+1)$ ，并对其中一种分布利用逆累积概率函数或样本数据的近似分位数函数来计算相应的分位数作为采样点。

在实现的时候，我们可以分别定义 x 轴和 y 轴上的分布，可以通过分布名称+参数确定，也可以是由样本数据得到的经验分布。

每个轴的定义代码如下：

```
public class ProbabilityAxis {

    int type = -1;
    DistributionFuncName funcName = null;
    double[] params = null;
    IntervalCalculator ic = null;

    public ProbabilityAxis(DistributionFuncName funcName, double[] params) {
        this.type = 0;
        this.funcName = funcName;
        this.params = params;
    }

    public ProbabilityAxis(IntervalCalculator ic) {
        this.type = 1;
        this.ic = ic;
    }

    public double[] getQuantiles(int n) {
        switch (type) {
            case 0: {
                IDF idf = new IDF(this.funcName, this.params);
                double[] vals = new double[n];
```

```

        for (int i = 0; i < n; i++) {
            vals[i] = idf.calculate((i + 1.0) / (n + 1.0));
        }
        return vals;
    }
    case 1: {
        ApproximateQuantile aq = new ApproximateQuantile(n + 1,
ic.getLeftBound(), ic.getStep(), ic.getCount());
        double[] vals = new double[n];
        for (int i = 0; i < n; i++) {
            vals[i] = aq.getQuantile(i + 1);
        }
        return vals;
    }
    default:
        throw new RuntimeException();
}
}

public double[] getProbabilities(double[] samples) {
    switch (type) {
        case 0: {
            CDF cdf = new CDF(this.funcName, this.params);
            int n = samples.length;
            double[] vals = new double[n];
            for (int i = 0; i < n; i++) {
                vals[i] = cdf.calculate(samples[i]);
            }
            return vals;
        }
        case 1: {
            EmpiricalCDF ecdf = new EmpiricalCDF(this.ic);
            int n = samples.length;
            double[] vals = new double[n];
            for (int i = 0; i < n; i++) {
                vals[ i] = ecdf.calculate(samples[i]);
            }
            return vals;
        }
        default:
            throw new RuntimeException();
    }
}
}
}

```

概率图的计算程序如下：

```
public class ProbabilityPlot {

    public static XYSeries PPPlot(ProbabilityAxis xAxis, ProbabilityAxis yAxis,
double[] samples) throws Exception {

        int n = samples.length;
        double[] xvals = xAxis.getProbabilities(samples);
        double[] yvals = yAxis.getProbabilities(samples);

        XYSeries xyseries = new XYSeries("Probability-Probability Series");
        for (int i = 0; i < n; i++) {
            xyseries.add(xvals[i], yvals[i]);
        }
        return xyseries;
    }

    public static XYSeries PPPlot(ProbabilityAxis xAxis, ProbabilityAxis yAxis,
int n) throws Exception {

        double[] samples = xAxis.getQuantiles(n);
        return PPPlot(xAxis, yAxis, samples);
    }

    public static XYSeries QQPlot(ProbabilityAxis xAxis, ProbabilityAxis yAxis,
int n) throws Exception {

        double[] xvals = xAxis.getQuantiles(n);
        double[] yvals = yAxis.getQuantiles(n);

        XYSeries xyseries = new XYSeries("Quantile-Quantile Series");
        for (int i = 0; i < n; i++) {
            xyseries.add(xvals[i], yvals[i]);
        }
        return xyseries;
    }
}
```

3.5 单变量的基本统计信息

由前面对整个统计信息的分析可知，对大数据只读取一遍就可以得到非常丰富的信息，我们将其称为单变量的基本统计信息。

实际计算过程中，数据以表格的方式存放，表中的一列对应一个变量，对每一列分别记录其统计量。

```
public class SummaryResultCol {

    /**
     * 数据类型
     */
    public Class dataType;
    /**
     * 全部数据个数，包括缺失值个数、空值个数、正负无穷的数值个数
     */
    public long countTotal;
    /**
     * 正常计算范围的数据个数，下面的各种统计量都是针对此范围的数据进行计算的
     */
    public long count;
    /**
     * 缺失值个数，用户没有输入该值
     */
    public long countMissValue = 0;
    /**
     * 空值个数，用户输入了，但并不不是一个数
     */
    public long countNaNValue = 0;
    /**
     * 值为正无穷的数据个数
     */
    public long countPositiveInfinity = 0;
    /**
     * 值为负无穷的数据个数
     */
    public long countNegativInfinity = 0;
    /**
     * 数据的和
     */
    public double sum;
    /**
     * 数据的平方和
     */
    public double sum2;
    /**
     * 数据的立方和
     */
    public double sum3;
```

```
    /**
     * 数据的四次方和
     */
    public double sum4;
    /**
     * 最小值对应的双精度浮点值
     */
    double minDouble;
    /**
     * 最大值对应的双精度浮点值
     */
    double maxDouble;
    /**
     * 极差所对应的双精度浮点值
     */
    double rangeDouble;
    /**
     * 最小值
     */
    public Object min;
    /**
     * 最大值
     */
    public Object max;
    /**
     * 极差
     */
    public Object range;
    /**
     * 均值
     */
    public double mean;           // = sum / count
    /**
     * 方差
     */
    public double variance;       // = (sum2 - mean * sum) / (count - 1)
    /**
     * 标准差
     */
    public double standardDeviation; // = sqrt(variance)
    /**
     * 变异系数 Coefficient of Variation
     */
    public double cv;             // = standardDeriation / mean
```

```

    /**
     * 标准误
     */
    public double standardError;        // = standardDeriation / sqrt(count)
    /**
     * 偏度 Skewness
     */
    public double skewness;              // = centralMoment3 / [ centralMoment2 *
sqrt(centralMoment2) ]
    /**
     * 峰度 Kurtosis
     */
    public double kurtosis;              // = centralMoment4 / [ centralMoment2 *
centralMoment2 ] - 3
    /**
     * 2 阶原点矩
     */
    public double moment2;                // = sum2 / count
    /**
     * 3 阶原点矩
     */
    public double moment3;                // = sum3 / count
    /**
     * 4 阶原点矩
     */
    public double moment4;                // = sum4 / count
    /**
     * 2 阶中心矩
     */
    public double centralMoment2;         // = ( sum2 - mean * sum ) / count
    /**
     * 3 阶中心矩
     */
    public double centralMoment3;         // = ( sum3 - 3 * sum2 * mean + 2 * sum
* mean * mean ) / count
    /**
     * 4 阶中心矩
     */
    public double centralMoment4;         // = ( sum4 - 4 * sum3 * mean + 6 * sum2
* mean * mean - 3 * sum * mean * mean * mean ) / count
    /**
     * 数据 TopN
     */
    public Object[] topItems = null;

```

```
    /**
     * 数据 BottomN
     */
    public Object[] bottomItems = null;
    /**
     * 频数信息，若不同数值个数较少时，会计算出此信息
     */
    TreeMap<Object, Long> freq = null;
    /**
     * 众数，在频数信息存在的前提下，可计算出此值
     */
    private Object mode = null;
    /**
     * 直方图数据
     */
    HistoData histoData = null;
    /**
     * 基本直方图区间划分及区间内元素个数
     */
    IntervalCalculator itvcalc = null;

    SummaryResultCol() {
    }

    /**
     * 获取频数信息
     *
     * @return 频数信息
     * @throws Exception
     */
    public TreeMap<Object, Long> getFrequencyMap() throws Exception {
        getFreq_Internal();
        return this.freq;
    }

    /**
     * 获取按数值排序的频数信息
     *
     * @return 按数值排序的频数信息
     * @throws Exception
     */
    public ArrayList<Entry<Object, Long>> getFrequencyOrderByItem() throws
Exception {
        getFreq_Internal();
```

```

        if (null == this.freq) {
            return null;
        }
        return new ArrayList<Entry<Object, Long>>(this.freq.entrySet());
    }

    /**
     * 获取按计数个数排序的频数信息
     * @return 按计数个数排序的频数信息
     * @throws Exception
     */
    public ArrayList<Entry<Object, Long>> getFrequencyOrderByCount() throws
Exception {
        getFreq_Internal();
        if (null == this.freq) {
            return null;
        }
        ArrayList<Map.Entry<Object, Long>> list = new ArrayList<Map.Entry<
Object, Long>>(this.freq.entrySet());
        Collections.sort(list, new Comparator<Map.Entry<Object, Long>>() {

            public int compare(Entry<Object, Long> arg0, Entry<Object, Long> arg1)
{
                return (int) (arg0.getValue() - arg1.getValue());
            }
        });
        return list;
    }

    /**
     * 获取百分位数信息
     *
     * @return 百分位数信息
     * @throws Exception
     */
    public Percentile getPercentile() throws Exception {
        if (!hasFreq()) {
            return null;
        }
        getFreq_Internal();
        Quantile qtl = Quantile.fromFreqSet(100, this.dataType, this.freq);
        Percentile pct = new Percentile(this.dataType);
        pct.items = qtl.items;
        pct.median = pct.items[50];
    }

```



```
pct.Q1 = pct.items[25];
pct.Q3 = pct.items[75];
pct.min = pct.items[0];
pct.max = pct.items[100];

return pct;
}

/**
 * 获取 q-分位数信息
 *
 * @param q    分位值个数
 * @return q-分位数信息
 * @throws Exception
 */
public Quantile getQuantile(int q) throws Exception {
    if (!hasFreq()) {
        return null;
    }
    getFreq_Internal();
    return Quantile.fromFreqSet(q, this.dataType, this.freq);
}

/**
 * 获取近似百分位数
 *
 * @return 近似百分位数
 * @throws Exception
 */
public ApproximatePercentile getApproximatePercentile() throws Exception {
    if (hasFreq()) {
        return new ApproximatePercentile(getPercentile());
    } else {
        return new ApproximatePercentile(this.getHistogram().
getIntervalCalculator(), this.minDouble, this.maxDouble);
    }
}

/**
 * 获取负值个数
 *
 * @return 负值个数
 * @throws Exception
 */
}
```

```
public long getNegativeValueCount() throws Exception {
    if (this.count != 0) {
        if (hasFreq()) {
            getFreq_Internal();
            long countNegativeValue = 0;

            Iterator<Entry<Object, Long>> it = this.freq.entrySet().
iterator();

            while (it.hasNext()) {
                Entry<Object, Long> e = it.next();
                if (Double.parseDouble(String.valueOf(e.getKey())) < 0) {
                    countNegativeValue += e.getValue().longValue();
                }
            }
            return countNegativeValue;
        } else {
            long countNegativeValue = 0;
            IntervalCalculator ic = this.getHistogram().getIntervalCalculator();
            double left = ic.getLeftBound().doubleValue();
            double step = ic.getStep().doubleValue();
            int intervalCount = 0;
            while (left + intervalCount * step <= 0) {
                intervalCount++;
            }
            for (int i = 0; i < intervalCount; i++) {
                countNegativeValue += ic.count[i];
            }
            return countNegativeValue;
        }
    } else {
        return 0;
    }
}

//此处略去一些代码

}
```