湖 南 农 业 大 学

全日制普通本科生毕业设计

基于 Hadoop 的聚类算法研究

Research on Clustering Algorithms Based on Hadoop

学生姓名：刘润尘

学　　号：200941842112

年级专业及班级：2009 计算机科学与技术(1)班

指导老师及职称：陈义明　副教授

学　　院：信息科学技术学院

湖南·长沙

提交日期：2013 年 5 月

湖南农业大学全日制普通本科生毕业设计

诚 信 声 明

本人郑重声明：所呈交的本科毕业设计是本人在指导老师的指导下，进行研究工作所取得的成果，成果不存在知识产权争议。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体在文中均作了明确的说明并表示了谢意。本人完全意识到本声明的法律结果由本人承担。

毕业设计作者签名：

年　　月　　日

目　　录

基于 Hadoop 的聚类算法研究

学　　生：刘润尘

指导老师：陈义明

（湖南农业大学信息科学技术学院，长沙 410128）

摘　要：本文主要研究了聚类算法向 Hadoop 分布式并行计算平台的移植和性能评估。首先，本文简述了 MapReduce 编程范型和 Aapche Hadoop 平台相关背景，并详细讨论了几种常见的聚类算法的特性和研究背景。根据算法特性，本文选择 K 均值聚类算法和模糊 K 均值聚类算法实现了在 Hadoop 平台上的移植，并对移植结果进行了验证；随后在一个分布式集群上，结合模拟数据集，通过几组对比实验对于移植后的聚类算法性能进行了系统的评估；实验结果显示移植后的算法取得了较好的加速比。

关键词：数据挖掘；聚类算法；大数据；Hadoop；贝叶斯原理；性能评价

Research on Clustering Algorithm Based on Hadoop

Student: Liu Runchen

Tutor: Chen Yiming

(College of Information Science and Technology, Hunan Agricultural University, Changsha 410128, China)

Abstract: This paper mainly researched the performance and application of clustering algorithms transplanted to Hadoop distributed parallel processing platform. First, this paper briefly introduced MapReduce programming paradigm with Apache Hadoop and its related extension, and then discussed the feature and research background of common clustering algorithms in detail. According to the feature of discussed algorithms, this paper chose to implement K-Means clustering algorithm and Bayesian Clustering algorithm on Hadoop platform, and verified the result of the implementation. After that, the performance of the implementations is systematically evaluated by several use case scenario and simulated datasets on a distributed parallel computing cluster. The result of experiments proved the correctness of the transplantation and tested algorithms achieved a good scale up factor running on

Hadoop cluster.

Keywords: data mining; clustering algorithm; massive data; Hadoop; Bayesian principle; performance evaluation;

1 前言

人工智能的研究从 1943 年开始，到它成为一门真正的科学，只用了仅仅几十年的时间[1]。人工智能的机器学习领域中的数据挖掘是一项相对较新的技术，但已经在各行各业得到了广泛的应用；社交网络运用数据挖掘提供各类推荐信息，利用用户习惯实现广告的定向投放；搜索引擎中结果的优化、内容分类也是其典型应用；在其他领域，如天气预报、地震预测等问题也可以通过机器学习的方法来解决。

现在，随着信息化在各行各业不断普及，以及计算和存储性能的不断提高，数据挖掘需要处理的信息量越来越大，往往需要大型集群进行处理。信息化所带来的一种新资源就是大量的数据。首先是云计算的普及，这样一种对于计算、存储和网络资源的透明访问机制极大的促进了需处理数据量的膨胀。与此同时，高度抽象的并行计算模型使得对计算资源的整合变得更加容易[2]。这样一来，硬件性能、算法性能直接与商业利益联系在一起，形成了一个不断发展壮大的生态圈。因此对于数据挖掘算法的研究有着广阔的需求和应用前景。

Hadoop是Apache基金会推出的一个开源集群运算框架，它使用了HDFS分布式文件系统作为支撑，由一个JobTracker支持多个TaskTracker而运行作业，可以可靠而高效的运行在几百台到几千台计算机组成的分布式集群上[3]；Hadoop平台能够自动的将运用MapReduce编程范型编写的作业并行执行，从而减少执行时间。除了Apache Hadoop以外，Hadoop也被Intel等公司进行了特别定制，甚至在CPU级别上进行优化支持。在一次实验中，Yahoo!使用一个有着 910 个节点的Hadoop集群使用了大约 200 秒对 100 亿行数据进行了排序[4]；

聚类算法在数据挖掘领域有着广泛的应用，它的原理是根据各数据点之间的距离度量，无监督的将输入数据分类。聚类算法是一系列算法的总称，针对不同的应用，可以选择合适的聚类算法进行建模。常见聚类方式有划分聚类、层次聚类、模糊聚类、基于图论、密度、网格以及基于贝叶斯理论的聚类。而模糊聚类、量子聚类以及基于粒度聚类这样的新型算法，往往能解决经典聚类算法难以克服的缺陷。

本文讨论了在 Hadoop 集群上大数据聚类算法的移植、改进和评价，使得移植后的聚类算法能满足大数据计算的需求。在随后的第 2 部分中，将介绍 Apache Hadoop

的基本原理；第 3 部分讨论了几种常见的聚类算法的实现，并总结了聚类算法目前的研究背景和关键问题；第 4 部分讨论了聚类算法向 Hadoop 平台的移植，并对移植后的算法进行了检验；在第 5 部分，作者结合一组测试数据，讨论移植后聚类算法在一个全分布式 Hadoop 集群上的性能；最后，在第 6 部分，作者针对本文进行的研究和本文尚未完全讨论的问题做了一个总结。

2 Hadoop 简介

2.1 MapReduce 编程范型

Google 公司为适应大数据的发展推出了 Google MapReduce 编程范型。用户定义一个 map 函数用于产生中间键值对集和 reduce 函数合并所有的中间数值，整个计算过程围绕着对于键值对的处理。许多实际问题都可以用这个模型来表示。用这个方式编写的程序都能够直接在大型集群上并行执行。运行环境将控制划分数据、机器故障、任务调度和内部通信等细节。

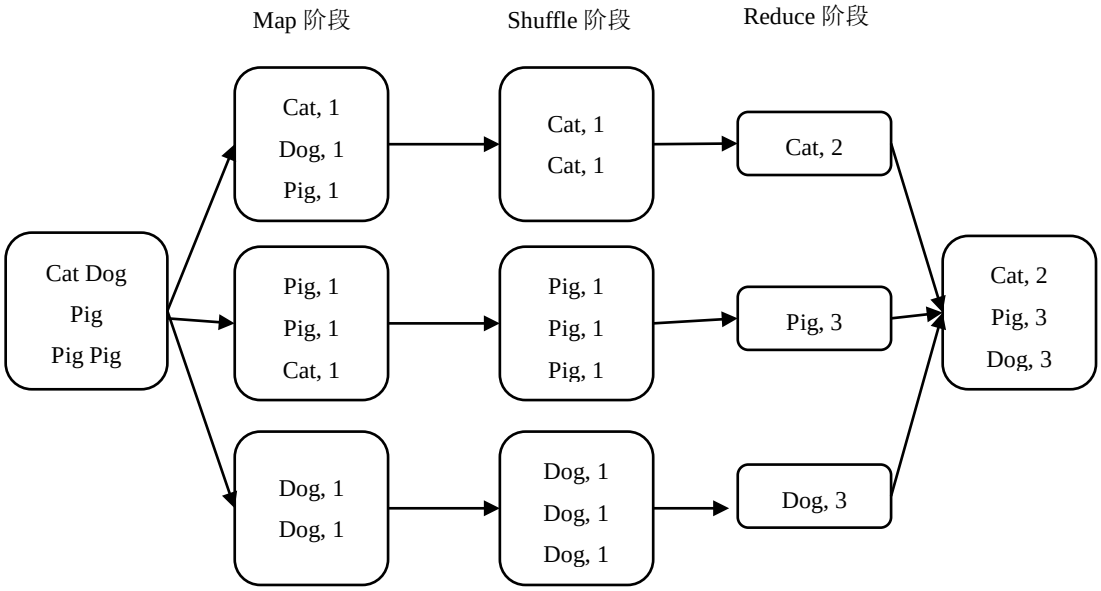

图 1　样例 MapReduce 数据流

Fig 1　Sample MapReduce Data flow

以一个计算文本中单词出现次数的小程序为例，现在给出对于 MapReduce 中 Mapper 和 Reducer 函数的描述。算法 1 中描述了单词计数程序中的 Mapper 函数，算法 2 中描述了单词计数程序中的 Reducer 函数；对它们的定义和计算都围绕着键值对展开。

算法 1：Mapper(key; value)

```
for word∈value do
    Emit_Intermediate(word, 1)
```

算法 2：Reducer(key; values)

result←0

for v∈values do

    result←result+v

Emit(key, result)

输入数据 key、value 依次经过 Map、Reduce 阶段从而产生最终结果。算法 1 描述了 Map 过程中所调用的 Mapper，本例中 key 是行在文件中的偏移量、value 是对应行包含的字符串；在这里每行字符串被分解为单词；对于所产生的每个单词，产生一个中间键值对使得单词作为 key，value 固定为整数 1。算法 2 描述了 Reduce 过程调用的Reducer,此时已经由一个Shuffle阶段,将中间键值对按照key相同的所有value组成一个数组values,成为reduce 的输入。Reducer 的输入key 是某个单词,而values是一个含有一串整数的数组，其中的每个数字代表着 key 在输入文件中的出现；这时reduce 所要做的就是计算数组中所用数字的合，并输出含有 key 和这个合的键值对。假设有一个输入数据包含几行英文单词,则上述算法的执行过程可以用图1表示出来。

Apache 基金会的开源项目 Hadoop 很好的实现了 MapReduce 的模型，是这个模型的主要应用，能够合理的满足对大量数据计算的要求。Hadoop 项目有几个主要的子项目，他们支持着 Hadoop 的各方面应用：

（1）HDFS（Hadoop Distributed File System，Hadoop 分布式文件系统）：主要关注于提高数据吞吐量。

（2）HBase：一个分布式的列数据库，使用 HDFS 作为下层存储，支持 MapReduce 批处理运算和点查询。

（3）Hive：一个分布式数据仓库，管理在 HDFS 上的数据，并提供一个基于 SQL 的查询语言。

（4）Mahout：一个可伸缩的机器学习和数据挖掘库，实现了聚类、分类等基于批处理的协同过滤。

2.2 HDFS 文件系统

与现有的其他分布式文件系统不同，HDFS是高度容错的，并且运行在廉价的普通商品计算机上，对于需要大量数据的应用HDFS是非常有效的。HDFS特别针对硬件故障做了优化，使用了一个很简单的数据完整性模型，并且能够比较容易的跨平台。HDFS运行在Linux平台上，利用Java实现,使用唯一的名称节点（NameNode）和多个数据节点（DataNode）。名称节点管理命名空间和客户端的连接，执行文件系统中的基本操

作；每个数据节点通常运行在一台单独的主机上，管理本机的数据存储，为客户端提供读写的数据[5]。这是一个主从结构，其逻辑架构如图 2 所示。

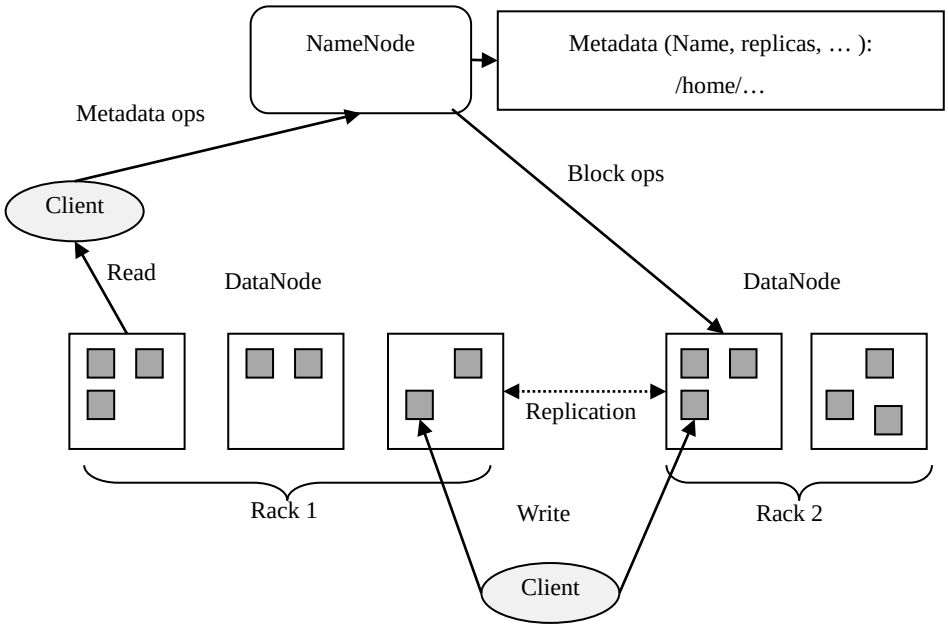

图 2　HDFS 逻辑架构

Fig 2　Logical Architecture of HDFS

HDFS 与 Linux FS（Linux File System，Linux 文件系统）相似，使用目录树组织文件，但是 HDFS 的目录树与 Linux FS 有一定的区别，如图 3、图 4 所示。

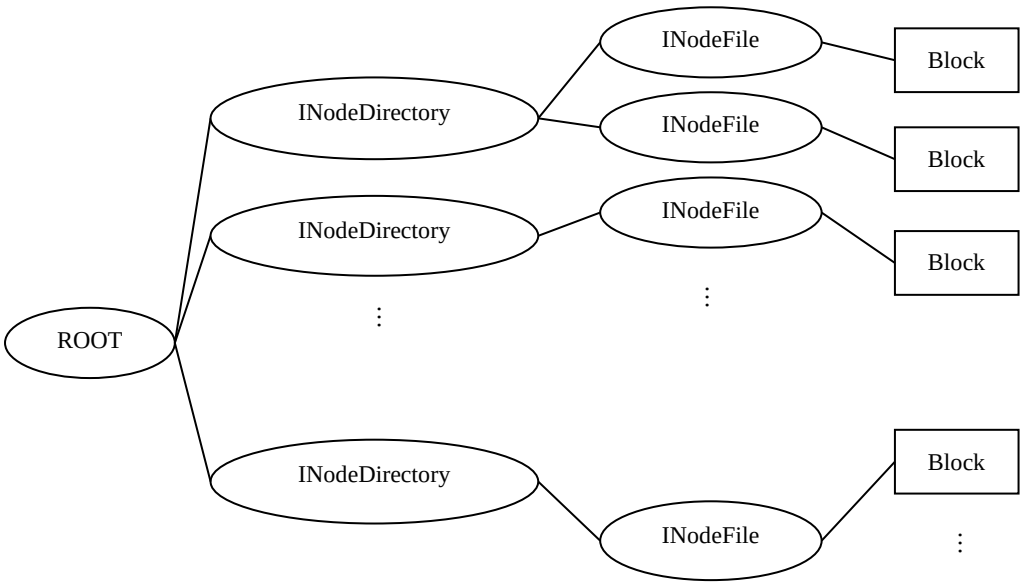

图 3　HDFS 目录树

Fig 3　Directory Tree of HDFS

HDFS 在 Linux FS 上的改动，可以从两个方面总结出来，分别是：第一，HDFS 节约名称节点内存空间，Dentry 与 INodeFile 没有子节点；第二，HDFS 节约名称节点

外存空间，INodeDirectory 没有以一组 Block 的形式存储在外存。

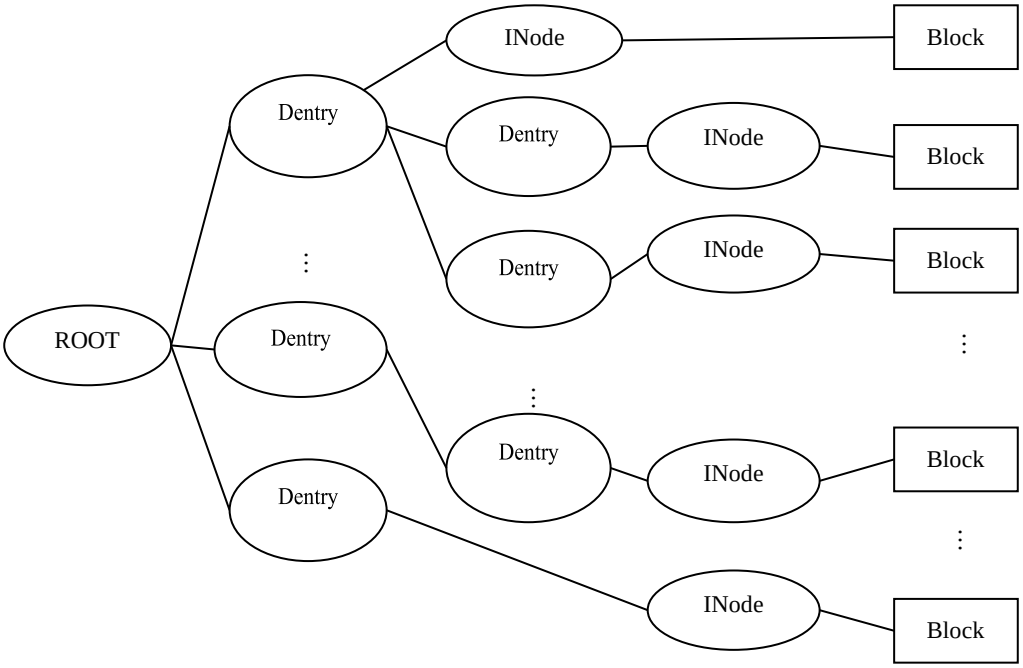

图 4　Linux FS 目录树

Fig 4　Directory Tree of Linux FS

2.3　Hadoop 节点管理

Hadoop 建立在 HDFS 之上，有一个 JobTracker 进程和多个 TaskTracker 进程。JobTracker 用于管理，运行于集群中的任意一台计算机上，而 TaskTracker 具体执行分配下来的任务（Task），必须运行于数据节点。图 5 描述了它们之间的关系。

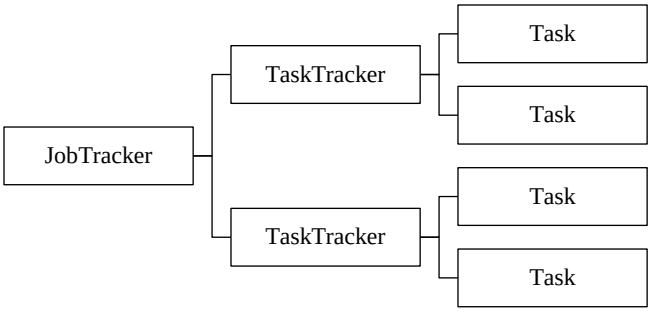

图 5　Hadoop 任务分配原理

Fig 5　Principle of Hadoop task assignment

实际上，在节点增多的情况下，一般会使用更多的备用节点，同时使用如 ZooKeeper 等工具，以保证在一个 JobTracker 出现故障时系统能够迅速恢复正常运行，这也体现了 Hadoop 良好的容错性。

2.4　Hadoop 功能特性

Apache Hadoop 是一个分布式批处理系统，并采用了 Google MapReduce 编程范型，因此对于能够执行的任务有一定的限制。首先，任何任务必须定义 Mapper 和 Reducer；

一般来说，Reducer 必须等待所有的 Mapper 完成后才开始执行；在 Mapper 和 Reducer 之间有一个 Combiner 用于预先合并部分数据，以便减少网络传输开销，通常可以直接指所使用的Reducer类;任务的输出文件数通常决定于Reducer的数量。由于Hadoop是一个分布式系统，在各个节点之间要尽量避免通信，因此不适合实现需要共享公共变量的算法。

Hadoop 的设计中十分注重系统的可靠性和容错性。对于 Map 阶段和 Reduce 阶段的中间结构，都会固化到 HDFS 中，这样一来，每当节点故障，JobTracker 都会加载固化的中间结果，重新运行丢失的计算。这样也使得 MapReduce 的正确执行建立在 HDFS 的可靠性之上，只要 HDFS 能够保证文件不丢失，就可以保证 Hadoop 正确执行 MapReduce 过程。

评价并行计算系统，常将Hadoop与其他系统对比性能差异。Dawei Jiang等人从多个角度对Hadoop进行了细致的评价，发现在合适调整系统内部参数以后，可以提高系统性能 2.5 到 3.5 倍[6]。基于这个原因，也促使了多种多样的修改版本Hadoop以便更有效的完成特定任务；Cloudera，EMC，IBM，INTEL，华为等公司都提供了自己的专用版本，每个版本都有各自的特点。Cloudera最早将Hadoop商用，使用Cloudera Manager可以迅速的将Hadoop集群配置完毕，并进行监控。EMC的版本则不需要单独的NameNode服务器，元数据分散存储，还支持NFS（Network File System，网络文件系统）直接访问HDFS，提供了与旧应用的兼容。IBM推出的Hadoop的发行版，在平台管理，安全认证，作业调度算法以及DB2 集成上做了增强。Intel与华为的版本则对各自的硬件提供了独到的支持。当然，如果能结合不同公司对Hadoop架构的增强，可以更高的提高对于不同硬件的兼容性。这也是开源软件的一大需求。

2.5 Hadoop 流处理

通用版本的Apache Hadoop并不支持在线聚集（Online Aggregation）和连续查询（Continuous Query）。为了解决这些问题，Neil等人在提出了一个修改版本HOP（Hadoop Online Prototype），在原有的Hadoop源码中添加了缓存控制包，主要包括了一个JBuffer类为输入输出提供缓存，BufferController类传输一个任务的输出作为另一个任务的输入,还有一个SnapshotManager类提供输入快照的支持[7]。这样一来，数据就可以在操作之间流动，扩展了MapReduce的批处理编程模型，还可以提高作业完成时间和系统利用率，使MapReduce程序可以用于实时监控和数据流处理。HOP一定程度的保留了Hadoop的容错性，可以运行未经修改的MapReduce程序。图 6 展示了Hadoop Online的数据流与Hadoop批处理的区别。

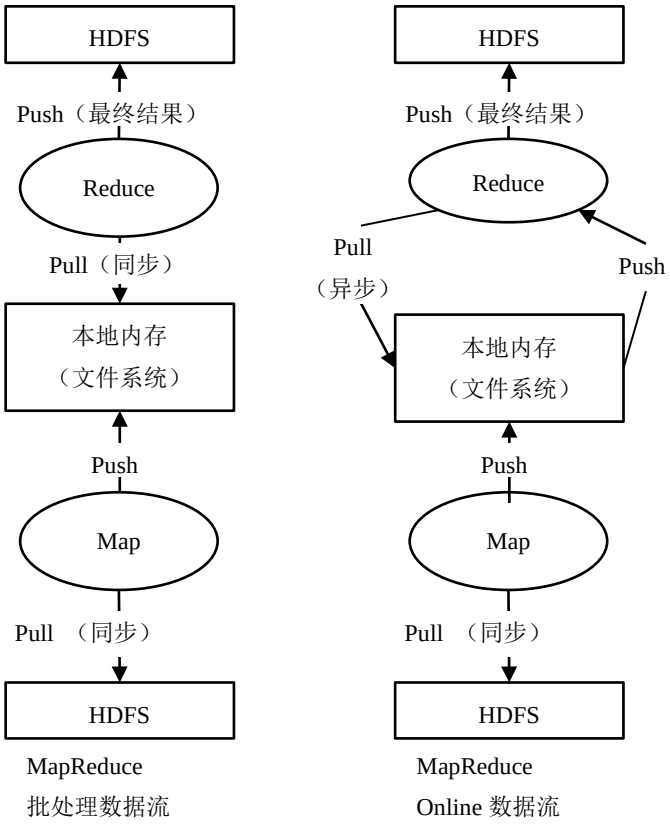

图 6　Hadoop Online Prototype 与 Hadoop 的数据流区别

Fig 6　Differences of Hadoop Online Prototype and Hadoop in Data Stream

3　聚类分析

聚类分析是一个比较老的机器学习研究领域，尚未有公认的定义，Everitt在 1974 年给出：同一类簇的任意两点间的距离小于不同类簇的任意两个点间的距离；类簇是一个包含密度相对较高的多维空间中的连通区域，被密度相对较低的点集的区域分隔。然而，这些定义只能提供一个理解，而不能准确的定义聚类。例如，在模糊聚类中，同一个元素可以属于不同的类。聚类算法可以用来解决很多问题，如对大量恒星光谱数据进行分类，找出不同的恒星类型。利用聚类可以实现了在不知道文档内容的情况下对其进行分类[8]。本文中将讨论几个无监督聚类算法（Unsupervised Clustering Algorithm），即输入数据没有人为设置的分类。

聚类分析的通常步骤为：从输入数据特征集中提取输入向量，聚类过程，结果评估[9]。结果评估主要是分析其有效性。在聚类算法中，初始化方式、距离度量、收敛条件都会对结果产生重要的影响。通过聚类模型不同，聚类算法可以分为几大类：层次聚类、划分聚类、密度聚类、贝叶斯聚类等。每种聚类算法的思路各有不同，但都有着相似的地方。

另外，为了方便研究，若不加其他说明，本文所有的讨论都将在一个二维的欧几

里得空间上进行。

3.1 层次聚类算法

层次聚类将初始数据进行聚集或者分裂。自底向上时，每一次迭代都将相近的类合并，直到满足一定终止条件或者所有类合为一个。同样的，这个算法也可以自顶向下执行，每次将现有的类分裂。

设 E 为一个对象集，C 是一个聚类集，下面是一个简单的自底向上层次聚类算法的伪代码。

算法 3：Hierarchical_Clustering(E; C)

```
for c∈C do
    c←(e∈E and e∉C)
repeat
    for ci∈C do
       for cj∈C do
          findClosest(ci,cj)
    ci←ci∪cj
    C←C-cj
until count(C)==1 or converged(C)
```

在这个算法中，首先初始化每一个对象为一个新的聚类，然后进入算法的主循环，当聚类没有收敛或者剩余聚类大于一个时，每次根据距离度量找出最近的聚类，并将其合并。findClosest(ci,cj)函数用于找到最近的两个聚类，count(C)表示 C 的元素数量，converged(C)是一个定义在 C 上的判别函数，判断当前聚类是否收敛。对于距离度量公式的选择，常见的有使用两个聚类c_i、c_j之间元素距离的最大值

$$\max\{d(x,y)|x \in c_i, y \in c_j\},$$

使用两个聚类之间元素距离的最小值

$$\min\{d(x,y)|x \in c_i, y \in c_j\},$$

和使用两个聚类之间元素距离的平均值

$$\frac{1}{|c_i| \cdot |c_j|} \cdot \sum_{x \in c_i} \sum_{y \in c_j} d(x,y).$$

显然，对应不同地具体问题，合理采用相应的距离度量，是计算成功的关键。

3.2 划分聚类算法

最常见的划分聚类算法是由 Lloyd 提出的 K 均值（K-Means）算法。设$E=\{e_1,e_2,e_3,\ldots,e_n\}$为一个含n个元素的集，$C=\{c_1,c_2,c_3,\ldots,c_k\}$为k个聚类质心（Cluster Centroid），标准的 K 均值算法要找出如何将n个元素分配到k个聚类中，从而使所有聚类误差方程最小

$$\operatorname{argmin}\left\{\sum_{i=1}^{k}\sum_{e_j\in c_i}(e_j-\mu_i)^2\,\middle|\,c_i\in C\right\},$$

其中聚类质心μ_i是k个聚类中元素的平均值。从一个初始的k个均值开始，算法通过循环两个阶段执行。首先是分配阶段（Assignment Step），根据元素和每个聚类质心的距离度量，将元素分配到最接近它的聚类中

$$c_i=\left\{e_l\middle|(e_l-\mu_i)^2\le\left(e_l-\mu_j\right)^2\forall 1\le j\le k\right\},$$

然后是更新阶段（Update Step），根据分配阶段的结果，重新计算各个聚类的质心

$$\mu_i=\frac{1}{|c_i|}\cdot\sum_{e_j\in c_i}e_j,$$

当分配阶段的结果不再改变的时候，算法完成。E 为元素集，C 为聚类集，下面是这个算法实现的伪代码。

算法 4：K_Means(E; C)

```
for c∈C do
    c←(e∈E and e∉C)
repeat
    for e∈E do
        L(ei)←argminDistance(e, c∈C)
    for c∈C do
        updateCluster(c)
until maxIteration() or converged(C)
```

L(e)函数表示元素 e 所属的聚类。首先初始化 C，这个初始化可以是将元素随机分配到各个聚类，然后计算出聚类质心。在这之后进入算法的主要循环，首先每次对于每个元素，根据距离公式计算出到各个聚类质心的距离，然后修改 L(e)使其指向距离最短的聚类；所有的计算完成之后，L 已经全部更新，这时计算新的划分下的聚类质心。当满足一定结束条件的时候结束算法，输出 L 和 C。

Khaled Alsabti 等人提出了一个利用树和空间分区的优化，并且认为可以很大程

度的降低时间复杂度。

3.3 模糊聚类算法

用$\mathrm{P}(h)$表示 h为真的概率，$\mathrm{P}(D|h)$表示当h为真时，D出现的概率。可以用这些符号表示后验概率$\mathrm{P}(h|D)$，也是贝叶斯定理（Bayesian Theorem）的形式

$$\mathrm{P(h}|D) = \frac{\mathrm{P}(D|h)\cdot \mathrm{P}(h)}{\mathrm{P}(D)},$$

由此可知当$\mathrm{P}(D)$增加时$\mathrm{P}(h)$不变，$\mathrm{P}(h|D)$则会下降，可以表示为 $\mathrm{P(h}|D) + \mathrm{P(h}|\overline{D}) = \mathrm{P}(h)$。由贝叶斯定理可以得出全概率公式，若事件$\{\mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3, \ldots, \mathrm{A}_n\}$相互独立，且 $\sum_{i=1}^{n} P(A_i) = 1$，对于事件B则有

$$\mathrm{P}(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i).$$

利用贝叶斯定理寻找聚类,可以看作是求一个最大似然估计(Maximum Likelyhood Probability）的过程。在这个基础上，首先讨论贝叶斯学习算法。在机器学习中，通过贝叶斯定理（Bayesian Theorem）可以得到在考虑所有已知条件的情况下，在一个假设空间 H 中找到最优假设$\mathrm{h} \in \mathrm{H}$，给定观测数据D。通过使用贝叶斯定理计算每个待选假设的后验概率，这样得到的最优假设被称为最大后验估计（Maximum A Posteriori， 简称 MAP）假设$\mathrm{h}_{\mathrm{MAP}}$，即

$$\mathrm{h}_{\mathrm{MAP}} = \frac{\underset{\mathrm{h} \in \mathrm{H}}{\mathrm{argmax}} P(D|h)P(h)}{P(D)},$$

在这个等式中，$\mathrm{P}(D)$是一个常数，因此可以省去，即可得

$$\mathrm{h}_{\mathrm{MAP}} = \underset{h \in H}{\mathrm{argmax}} P(D|h)P(h).$$

对于聚类算法,可以不考虑$P(h)$的变化。因此，把上式中$P(h)$省去，可以得到一个更简单形式的表达式，即最大似然（Maximum Likelihood，简称 ML）估计

$$\mathrm{h}_{\mathrm{ML}} = \underset{h \in H}{\mathrm{argmax}} P(D|h).$$

除此之外,利用全概率公式进行聚类分析,可以将输入数据作为一个混合分布T建模，得到表达式 1

$$T(d) = \sum_{i=1}^{k} T(d|A_i)T(A_i),$$

其中数据点$\mathrm{d} \in \mathrm{D}$，用随机变量A表示T中的k种成分，$T(A_i)$则是每种成分的权值。

3.3.1 模糊 K 均值聚类算法

在计算对象到聚类的距离时，有时候一个对象与其他几个聚类都相隔较近，如果

只将对象分配到一个聚类中并不是最好的分配。因此结合贝叶斯概率公式，允许计算一个对象属于某个聚类的概率，在这个基础上设计模糊聚类算法，使得同一个数据点可以属于多个聚类。这个算法需要从混合分布T中还原出k种成分，也就是找到k个给定聚类的质心。据此导出表达式 2

$$p_{ij} = T\left(C_i^{(t)} \middle| d_j\right)$$

与表达式

$$p_{ij} = \alpha \cdot T\left(d_j \middle| C_i^{(t)}\right) T\left(C_i^{(t)}\right).$$

其中p_{ij}代表数据点d_j属于一个聚类$C_i^{(t)}$的概率，$T\left(C_i^{(t)}\right)$ 表示聚类的权值，$T\left(d_j \middle| C_i^{(t)}\right)$则是聚类$C_i^{(t)}$中包含$d_j$的概率，$C_i^{(t)}$的上标$(t)$用于标志迭代次数；给定常数$\alpha$。

每次聚类的过程类似于 EM（Expectation Maximization）算法，先由表达式 2 计算出新的p_{ij}，这时完成聚类的分配阶段；再通过p_{ij}计算$C_i^{(t+1)}$，这时完成聚类的更新阶段，有表达式 3

$$C_i^{(t+1)} \leftarrow \sum_{p_{ij} \in C_i^{(t)}} p_{ij},$$

每次迭代过程中逐步求精，在一般情况下，能够收敛于一个局部最优点。将这个模糊 K 均值算法用伪代码表示出来，如下所示。

算法 5：Fuzzy_K_Means(P; C)

```
for c∈C do
    c←(p∈P and p∉C)
repeat
    for p∈P do
        for c∈C do
            L(p,c)←assignment(p,c)
    for c∈C
        updateCluster(c)
until maxIteration or converged(C)
```

首先是初始化步骤，先将 C 中的聚类信息初始化，给予一个假设的已知值。随后进入主循环，每次计算数据点来自每个聚类的可能性，然后计算更新后的聚类，即表达式 3 中的$C_i^{(t+1)}$，进入下次迭代。assignment(p,c)函数根据表达式 2 返回点 p 存在于某个聚类中的可能性。直到达到最大迭代次数 maxIteration，或者满足收敛条件 converged(C)，算法结束。

3.3.2 层次贝叶斯聚类算法

若将 MAP 推理过程与层次聚类算法的思想结合起来，可以利用最大似然估计，在每次迭代时选择需要合并的聚类，算法的描述如下。

对于数据集D，设第i步的分配方案为$\aleph_i$，对于聚类c可以找到函数关系$\aleph_{i+1} = \aleph_i - \{c_x, c_y\} + \{c_x \cup c_y\}$，则在第$i+1$步时，$P(\aleph_i|D)$为一个确定的常数。显然有表达式

$$\underset{\aleph_{i+1}}{argmax} P(\aleph_{i+1}|D) = \underset{\aleph_{i+1}}{argmax} log \frac{P(\aleph_{i+1}|D)}{P(\aleph_i|D)}.$$

给定一定的初始条件利用上式可以求得每一步的分类方案[10]。

3.4 图论聚类算法

首先给出一些图论的基本定义。图G的边连通度$k(G) = \beta$表示使图G成为非连通图最少需要移除的边的数量，同时图G也称为β-连通图。割是图中的一个边集，使得移除这个边集产生非连通图。最小割（mincut）则是所有割中包含边数最少的。因此一个非平凡图的割S是最小割当且仅当$|S| = k(G)$。当这样一条通路存在时，距离$d(v,u)$是顶点v和u之间的一条最短通路的长度，否则$d(v,u) = \infty$；通路的长度是其中所包含的边的数量。连通图的直径$diam(G)$是图G中任意顶点v、u之间，最大的$d(v,u)$。一个顶点v的度表示为$deg(v)$，指它所连接的边的数量；一个图的最小度表示为$\delta(G)$。

Jain 在 1999 年提出了一个图论聚类算法：构造一颗最小生成树，通过删除最小生成树的最长边来形成类。2007 年 Li 在一篇论文中描述了一个基于最大θ距离子树的算法 MDS_CLUSTERS，通过移除 MST 中所有长度大于θ的边，所得的每个顶点集为一个聚类。同是 1999 年的另一篇论文中，Erez Hartuv 和 Ron Shamir 描述了一个基于图的连通性，利用高连通图进行分裂聚类的算法。高连通图（Highly Connected Graph，HCG）的定义为：一个非平凡图G是高连通图当

$$k(G) > \frac{n}{2}.$$

一个高连通子图（Highly Connected Subgraph，HCS）是一个子图$H \subseteq G$，当H是高连通图。高连通子图算法的伪代码描述如算法 6 所示。

在算法 6 中，removeClusteredVertices(G)过程从 G 中移除已经聚类的顶点和对应的边；removeVertices(d,G)从 G 中移除所有的顶点满足表达式$deg(v) < d$；singletonAdoption(G)将 G 中孤立点根据邻近关系加入聚类，以此增加算法的有效性。原作者认为，这个算法有效地提高了在多种数据集上图聚类算法的性能。

```
算法 6：HCS_Clustering(D; G)
V←vertex(G)
E←edge(G)
function HCS(G)
    H,h,G←mincut(G)
    if highlyConnected(G) then
        return G
    else
        HCS(H)
        HCS(h)
for d∈D do
    H←removeClusteredVertices(G)
    H←removeVertices(di,H)
    repeat
        HCS(H)
        singletonAdoption(H)
        removeClusteredVertices(H)
    until clusters(H)==∅
```

3.5 其他聚类算法

有时候目标类簇可能呈现出多种多样的空间形态，此时通过一个密度函数，以密度的高低来分辨类簇的界限。这样的密度聚类算法能够很好的克服传统聚类形态的局限性，找到一些空间分布不规则的特殊聚类。

还有基于网格的聚类算法，将聚类空间限定为有限的网格，牺牲精度以提高处理速度。Shickuta 描述了应用于大型数据集的网格聚类算法 GRIDCLUS，并认为这个算法在内存需求和行为上比传统的层次聚类算法更优。这个算法分为这样几步：建立网格结构、计算块密度、块排序、识别聚类质心、遍历邻接块。

与模拟退火结合的核模糊无监督聚类算法，使用Xie-Beni指标作为目标函数，建立在马尔可夫链蒙特卡罗上，避免了常见的核重合问题，有着更好的有效性[11]。

3.6 聚类算法的关键问题

任何机器学习都可以定义为通过经验E在度量P下完成任务T的一个程序。从这个广义的定义出发，本文讨论几个聚类算法研究中的关键问题[12]。

3.6.1 距离度量

对于在欧几里得空间上进行的聚类，为了表示数据空间中两点之间的差异，总是需要采用一定的距离度量公式；使用不同的距离度量公式将对聚类的过程和结果产生直接的影响，因为距离度量定义了空间的属性，实际上也组成了聚类算法的归纳偏置（Inductive Bias）。现在取两个数据点(x_1, y_1)和(x_2, y_2)，假设求它们之间的距离d，从而简单定义二维空间中的距离公式。

欧几里得距离（Euclidean Distance）也称欧氏距离，是最常见的距离公式，它的形式是

$$\mathrm{d} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

曼哈顿距离（Manhattan Distance）起源于对于城市区块距离的度量，公式为

$$\mathrm{d} = |x_1 - x_2| + |y_1 - y_2|.$$

明可夫斯基距离（Minkowski Distance）是一个一般形式的距离度量，并且有一变量p，当p取不同值的时候所得的距离度量都是不同的，其公式为

$$\mathrm{d} = \sqrt[p]{(x_1 - x_2)^p + (y_1 - y_2)^p}.$$

切比雪夫距离（Chebyshev Distance）是一个较为有趣的距离，它的结果是国际象棋的国王在一个棋盘中两点间行进的步数，它也是明可夫斯基距离中当p趋向于无穷大时的结果

$$\mathrm{d} = \lim_{p \to \infty} \sqrt[p]{(x_1 - x_2)^p + (y_1 - y_2)^p} = max\{|x_1 - x_2|, |y_1 - y_2|\}.$$

在不同领域的研究中，有时候使用向量空间来表示数据，有一种球形空间聚类算法借鉴了这种方式。这种表示方法更加注重于对象之间的角度差异，同样适合于文本的处理，对于向量$\vec{x}$和$\vec{y}$有余弦相似度（Cosine Similarity）

$$\mathrm{d} = \frac{\vec{x} \cdot \vec{y}}{|x||y|}.$$

3.6.2 终止条件

聚类算法是一个迭代过程，因此终止条件直接影响着结果的好坏。有两种基本的终止条件。方法一，将终止条件简单的定义为聚类对象的分配不再变化。显然，这个方法的计算量较大，而且存储要求也比较多；对于噪声较高的数据，也容易出现收敛过慢的现象。为了提高实际性能，需要采用不同的评价方式。

方法二，比较相邻两次迭代聚类质心的位移，如果这个位移小于一个阀值μ则判定聚类收敛。这个方式的缺陷在于，第一，质心没有考虑到聚类的形态变化；第二，μ的大小很难确定。但是与检查每个聚类对象的分配相比，只计算聚类质心的位移可

以极大的减少计算量。

进一步，还可以考虑在两次迭代中聚类的抽象相似性，称为方法三：通过提取聚类的特征参数，综合对比。通常来说，这样可以得到比方法一和方法二更好的结果。实际上，聚类终止条件就构成了聚类的度量。

3.7 聚类分析的相关应用

聚类分析的研究已经存在了很长的时间，也得到了很多领域的大量应用，如动植物生态研究、基因学研究、医疗图像处理、Web搜索优化、社会学分析以及各类科学研究的辅助计算[13]。

例如，单一层次聚类可以被用于文档搜索结果的总结和浏览[14]。Krishna Kummamuru等人开发的DisCover算法所产生的分类考虑到了文档覆盖率、简洁性、节点的可预测性、到达时间等主要性能指标，在用户调查和几个横向对比实验中取得了很好的结果。Tapas Kanungo等人也采用了二叉树的方式对样本空间进行过滤，实现了对于图像聚类的优化[15]。通过简单的修改，K均值算法还可以间断运行，这样能够适应某些大型计算过程[16]，另外，还可以通过聚类来分析学生的学术表现[17]。

监督学习需要一组人为给出结果的训练样本，从而限制了监督学习的应用领域，而无监督学习不需要给出分类结果。聚类分析是一种常见的统计学习（Statistical Learning）方法，主要实现的是无监督学习。与其他机器学习方法相比，聚类分析的算法实现比较简单，而且计算过程通常是可划分的，便于并行的处理数据。另外相对复杂的算法，如支持向量机（Support Vector Machine）也可以在Hadoop实现[18]。

4 聚类算法向 Hadoop 移植

4.1 数据类型

Hadoop 中提供了多种数据类型供输入输出使用，主要如表 1 所示。

表 1 Hadoop 内置的 Writable 包类

Table 1 Writable Wrapper Classes in Hadoop

类名	基础类型	序列化大小
BooleanWritable	boolean	1
ByteWritable	byte	1
IntWritable	int	4
VIntWritable	int	1~5
FloatWritable	float	4
LongWritable	long	8
VLongWritable	long	1~9
DoubleWritable	double	8
Text	-	-
BytesWritable	-	-

续表 1

类名	基础类型	序列化大小
ObjectWritable	-	-
GenericWritable	-	-
MD5Hash	-	-
NullWritable	-	-

上表中有几种 Writable 类是通过扩展 Java 原有的基础类型的来的，其他的则更为复杂。这些 Writable 类都有一个 get()方法和 set()方法，另外，通过继承和实现接口，Hadoop 允许用户自定义多种多样的 Writable 类用于输入输出。

4.2 K 均值聚类算法

4.2.1 算法设计

本文首先实现的是朴素的 K 均值算法，以下称 NKM 算法。一次 MapReduce 过程并不能很好的表示 K 均值聚类算法的过程。K 均值算法是迭代进行的，每次迭代都更新一次聚类列表。若把每个迭代都看作一个 Map 和 Reduce 的任务，那么通过重复执行 MapReduce，可以逐步实现 K 均值聚类算法，如图 7 所示。

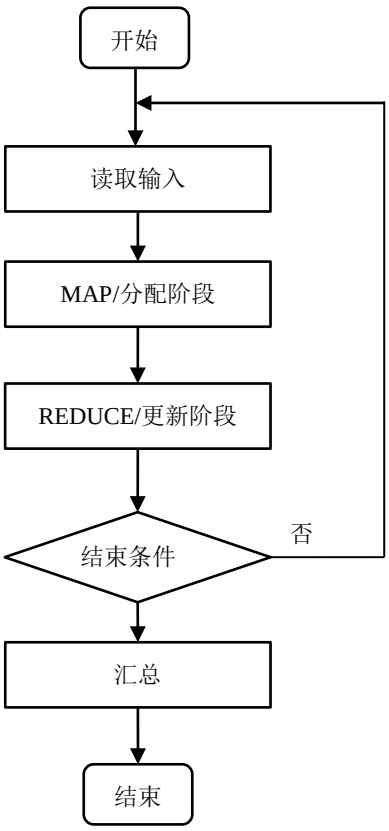

图 7 基于 Hadoop 的 K 均值算法流程图

Fig 7 flowchart of K-Means algorithm based on Hadoop

在 NKM 中，读取输入的数据来自于两个目录，一个是点集目录，另一个是聚类质

心目录，这两个目录是在程序启动参数中确定的；聚类质心目录的序号会随着每次迭代而改变，从而保存每次计算的历史轨迹。

4.2.2 自定义 Writable 类

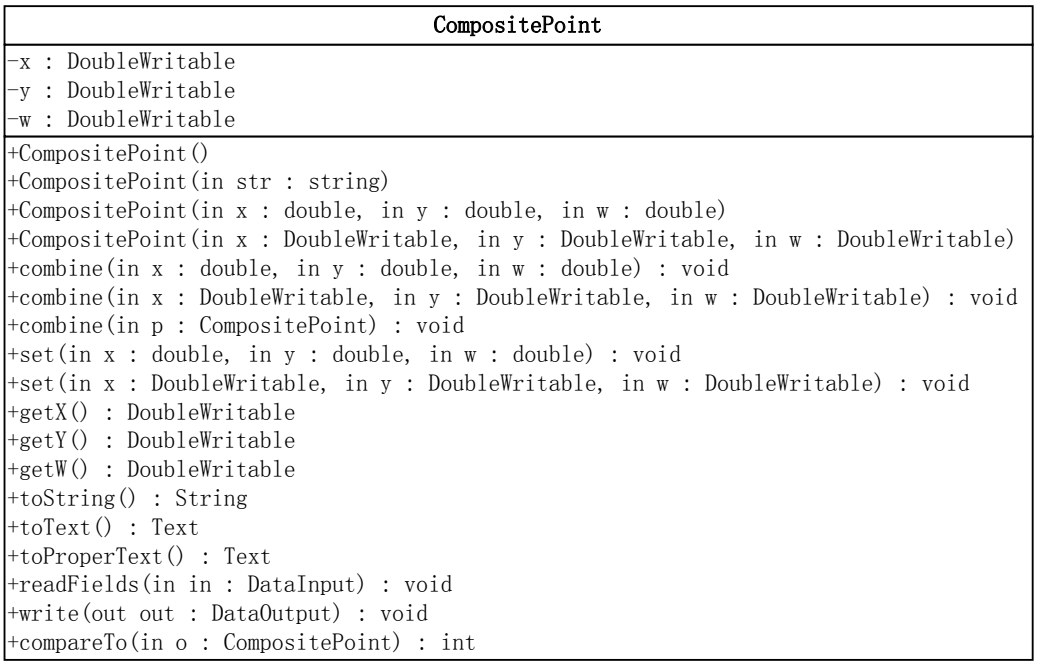

图 8　CompositePoint 的类图

Fig 8　Class Diagram of CompositePoint

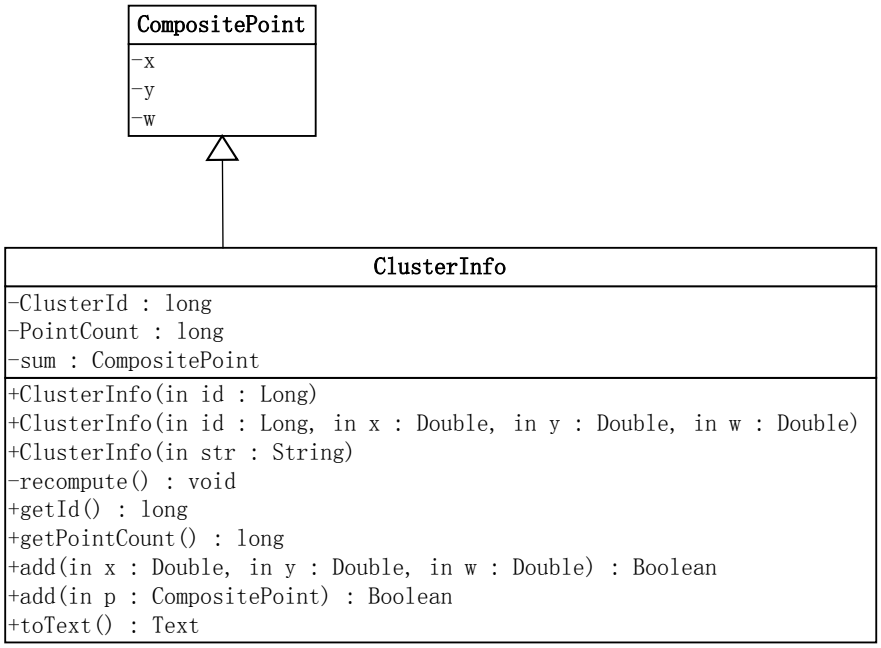

图 9　ClusterInfo 的类图

Fig 9　Class Diagram of ClusterInfo

Hadoop 允许用户定义类用于输入输出，同时自定义的类必须一个特定的基类如 WritableComparable 或 RawComparator。在 NKM 算法中，需要定义一个代表数据点的 CompositePoint 类和一个保存聚类信息的 ClusterInfo 类，图 8 是 CompositePoint

的类图。CompositePoint 类包含了数据点的坐标信息(x,y)以及权值 w，支持多种构造函数。其中的 combine()方法用于将当前点与输入点组合；set()方法用于修改点信息；getX()，getY()和 getW()方法用于取得点信息；toString()，toProperText()和 toText()方法将数据点转化为一个字符串输出。

以数据点类 CompositePoint 为基础，可以扩展一个子类 ClusterInfo 用于存储和表示聚类及其信息，其类图如图 9 所示。ClusterInfo 类继承了 CompositePoint 类，根据需要添加了新的参数和方法，并重写了 toText()方法。其中，ClusterId 用于保存当前聚类的标识符，PointCount 保存当前聚类中点的数量，sum 用于保存聚类的质心；相对应的 getId()和 getPointCount()用于获取这两个参数；add()方法实现了向聚类中添加数据点的功能。

应用这些自定义的数据类型和 Hadoop 原有的数据类型，可以很好的对 NKM 算法进行描述。

4.2.3 NKM 的距离度量

距离度量方式的采用对聚类算法的影响是至关重要的，因为它定义了数据空间的属性，本文的 NKM 算法采用了三种可选的距离度量，分别是欧几里得距离，曼哈顿距离和$\mathrm{p}=-1$的明可夫斯基距离。定义的 Distance 类图如图所示。

Distance
-measure : int -INFINITE : double -VERY_FAR : double -MEASURE_EUCLIDEAN : int -MEASURE_MINKOWSKI : int -MEASURE_MANHATTAN : int
+setMeasure() : void +compute(in cluster : ClusterInfo, in point : CompositePoint, in measure : int) : double +compute(in cluster : ClusterInfo, in point : CompositePoint) : double +compute(in c1 : ClusterInfo, in c2 : ClusterInfo) : double

图 10　Distance 类的类图

Fig 10　Class Diagram of Distance

在这个类中，setMeasure()方法用于修改距离度量，compute()方法用于返回计算结果，使用设定的距离度量；另外 compute()方法还有一个重载方法用于计算当前给定距离度量下的结果。

4.2.4 NKM 的 Mapper 类

Mapper 完成的是将输入数据转换为中间键值对的工作,它对应着本算法的分配步骤。作为 K 均值算法的 Map 部分，要扩展一个 Mapper 通用类，有着四个参数分别定义输入键、输入值、输出键和输出值。现给出定义如下：

（1）输入键：LongWritable；输入值：Text

（2）输出键：LongWritable；输出值：CompositePoint

在这里，根据所选择的输入数据方式，输入值是扫描文件的每一行，输入键表示输入值在文件中的位移量。在 Map 方法中，需要完成的任务是根据距离度量，将数据点分配给聚类。

首先，在 Map 方法初始化阶段将读取配置参数中的聚类目录路径，据此通过 Load 方法加载聚类信息到本地缓存，用一个 ClusterInfo 类型的数组 clist 保存。有了这个信息，K 均值聚类算法的分配步骤可以进行，算法 7 是 Mapper 方法的伪代码描述。

算法 7：NKM_Mapper(key; value)

```
clist←load()
cmin←INFINITE
cp←parseCompositePoint(value)
for c∈clist do
    cmin←findMinDistance(cp, c, cmin)
output(getId(cmin), cp)
```

算法十分简洁，首先通过 load()函数加载聚类信息列表，利用这个列表接下来对数据点进行分配。对于输入值 value，从中读取数据点 cp，然后找到离点 cp 最近的聚类 cmin，最后通过 output()过程，输出所分配聚类的编号和数据点本身，它们组成了中间键值对，随后通过 Shuffle 阶段的处理以后将成为 Reduce 阶段的输入。

4.2.5 NKM 的 Reducer 类

Reducer 完成的是处理中间键值对产生输出的工作。在这个算法中，用到了两个不同的 Reducer，一个用于产生每次迭代中新的 Centroid 文件，一个用于最后汇总输出划分结果。首先给出 Reducer 输入输出的接口数据类型：

（1）输入键：LongWritable；输入值：CompositePoint

（2）输出键：LongWritable；输出值：ClusterInfo

这里的输入输出键都保存了聚类标识符，输入数组中的元素为 CompositePoint，最后的结果通过 ClusterInfo 类型输出。每次迭代中使用的 Reducer 类算法的伪代码描述见算法 8。

算法 8：NKM_Reducer(key; values)

```
cluster←key
for p∈values do
```

```
    cluster←cluster+point
Output(key, cluster)
```

在 Reduce 阶段，NKM 算法接收已经按照 key 值合并的分配结果，包括了作为键的聚类标识符和作为值的一个属于此聚类的数据点的数组。首先按照 key 值新建聚类；随后，将分配到的数据点依次加入聚类中；最后，将所得的新聚类标识符作为键，而此新聚类的描述向量作为值输出;每个输出都形成下一次聚类迭代中使用的 Centroid 文件。

当已经判断收敛并结束迭代以后，将执行一个总结性的 MapReduce 过程，这个过程使用与之前相同的 Mapper 类，和一个特殊的 Reducer 类。这个 Reducer 类将数据点的分配情况汇总输出，它的伪代码表示如算法 9 所示。

```
算法 9：NKM_Last_Reducer(key; values)
cluster←new ClusterInfo(key)
for p∈values do
    output(key, p)
    cluster←cluster+p
output(key, cluster)
```

这个总结性的 Reducer 接收的键值对于算法 8 中相同。不同的是，此时每次向新聚类中加入数据点的时候，都将 Mapper 分配结果按照标识符为键、数据点为值的方式输出，最后再输出得到的新聚类信息。

4.2.6 NKM 的驱动与实现

NKM 的流程较普通的 MapReduce 来说更为复杂，因为一次算法中要多次的执行一个 MapReduce 工作。算法驱动类将在 Hadoop 平台的 JobTracker 上运行，首先将引导进行迭代过程，直到一个终止条件满足。每次迭代都是一个 MapReduce 工作，需要进行初始化，设置工作参数、Mapper 类和 Reducer 类。

结合上述内容，接下来将描述本文 NKM 算法的工作流程。启动 Hadoop，首先，将实验数据和 NKM 实例上传到 HDFS;实验数据包含了一个数据点文件目录和一个聚类信息目录。根据实际需求，配置好 NKM 的初始化参数，接下来启动计算。

NKM 的算法驱动类首先根据设定好的参数进行初始化，确定收敛阀值、最大迭代此时等；然后，NKM 驱动将向 JobTracker 发布每次迭代的工作，等待 JobTracker 返回结果。此时，JobTracker 首先将任务分配到各计算节点的 TaskTracker，让他们调用 Mapper 类。每一个 TaskTracker 中可以同时运行多个 Mapper，并且每个 Mapper

的运行都是异步的；在每个计算节点上运行的 Map 任务都会被分配到处理一部分的数据点文件；此时，TaskTracker 开始正式运行任务。NKM 的 Mapper 首先从任务信息中读取设定好的聚类信息目录，然后运行 load()函数，从 HDFS 的相应目录下文件列表中读取所需的聚类信息，把这个数据存储在本地缓存中。随后，Map 阶段开始，每个 Mapper 不断的接收输入数据，从中读取数据点，然后根据距离函数和聚类信息，计算出分配结果，并将这个分配结果输出为一系列中间键值对。

接下来，当所有的 Mapper 的输出完成以后将进行 Shuffle 阶段，即使得拥有相同键的值合并为一个数组，形成 Reducer 的输入，随后这个数据通过默认的 Partitioner 使用一个哈希表分配给相应的 Reducer 进行 Reduce 过程。

NKM 算法中的 Reducer 从输入数据中获取含有聚类标识符的键，此时数值列表的元素为分配所得的数据点，随后根据这个数据点重新计算当前的聚类信息，再产生输出，此时一次迭代就完成了。

每次迭代完成以后，JobTracker 将返回 NKM 驱动类，检查是否满足算法终止条件，否则重新创建 MapReduce 工作进行下一次迭代，使用本次工作输出的聚类信息目录作为新的参考目录。

当满足算法终止条件后，NKM 驱动将进入概括阶段。在这个阶段将进行一次 MapReduce 过程，输出每个数据点对于聚类的分配结果。此时算法完成。输出文件有每次迭代产生的聚类信息，以及最终得分配结果，误差小于初始化设定的收敛阀值。

4.2.7 小规模测试数据结果

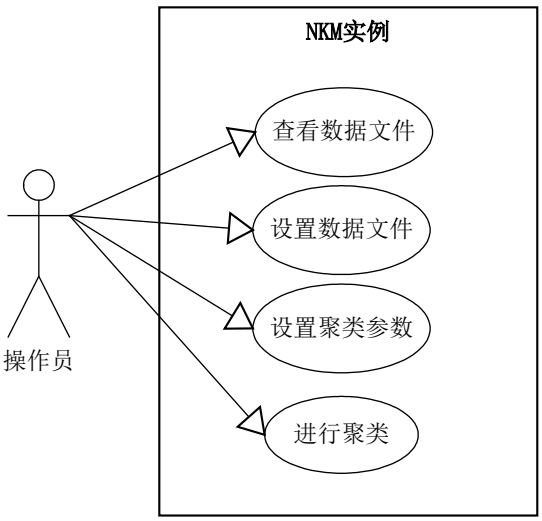

图 11 NKM 测试实例的用例图

Fig 11 Use Case Diagram of NKM Test Instance

下面讨论在一个伪分布式配置的 HOP 上运行一个随机生成的测试数据集。简单的分析 NKM 的功能，主要是对一个一定格式的输入数据，在给不同的参数下得出聚类结

果；对比在不同的输入数据属性、不同的聚类初始化条件、以及不同的距离度量的情况下 NKM 实例的测试结果。可以做一个测试系统的用例图如图 11 所示，用例图描述了用户与系统的功能交互。

输入数据集包含一组数据点文件，称为简单数据集，本章节所有的实验都在这个数据集上进行，共 32 行，每行包括用空格分隔的数据点描述一组浮点数 x、y、w，其中 x、y 是数据点的坐标，w 是数据点的权值；另外还有一组聚类初始化数据文件，每行使用 4 个浮点数 k、x、y、s 共描述了 7 个聚类，也将在第一次聚类迭代过程中被加载作为分配阶段的依据，其中 x、y 是聚类的初始化质心，k 是聚类的标识符，而 s 是聚类包含的数据点数量。测试所使用的计算机主要参数如表 2 所示。

表 2　测试用例使用的计算机描述

Table 2　Description of the Computer Used in the Test Instance

组件	性能指标
处理器	Intel Core i5 520M 2.4GHz x64 核心数 2 线程数 4
内存	8192MB DDR3 频率 532.1Mhz 双通道
硬盘	SAMSUNG SSD PM810 256GB 传输速率 230MBps
操作系统	Ubuntu 12.04 LTS x64
Hadoop 版本	Hadoop Online Prototype release 0.2

输入数据包含随机产生的在平面矩形区间{(0,0),(100,100)}的 32 个数据点，权值均设置为 1.0;这个数据集中初始化聚类质心的位置、数据点分布情况如图 12 所示，其中左图的十字符号表示了数据点，而右图的圆表示初始化聚类质心的位置。

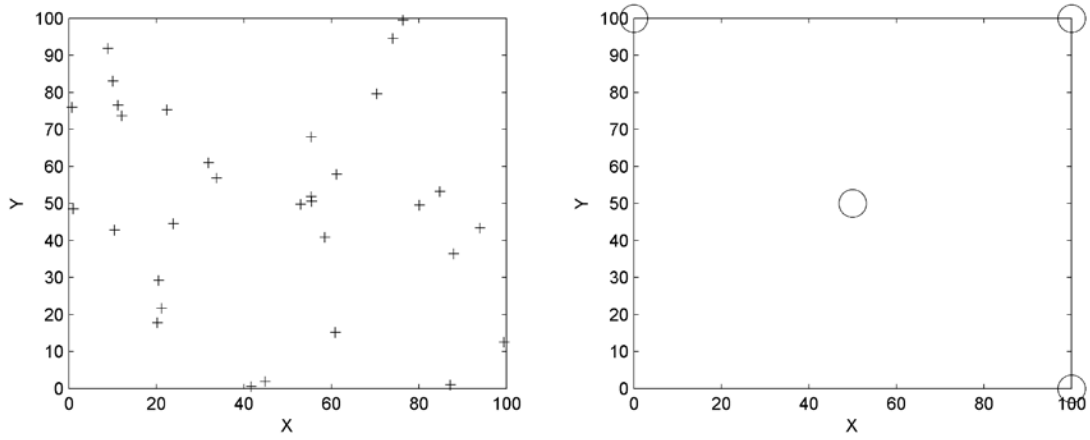

图 12　数据点和初始化聚类的分布情况

Fig 12　Distribution of Data Points and Initial Cluster

使用上述测试数据集，本文利用控制变量法，实验测量了在不同距离度量下 NKM 算法的性能表现；分别测试了欧几里得距离、明可夫斯基距离和曼哈顿距离，如表 4.2.2、表 4.2.3、表 4.2.4 所示，表的每行代表一次迭代中的聚类结果，每个聚类用一个含有三个参数的实数向量(x,y,r)表示，(x,y)是聚类质心的坐标，r 则是聚类的

半径，是聚类加权方差；可以通过这些实验数据分析聚类的性能指标。

表 3　NKM 算法使用欧几里得距离的聚类过程

Table 3　Clustering Progress of NKM Algorithm with Euclidean Distance

迭代	聚类 1	聚类 2	聚类 3	聚类 4
0	100.00,0.00,　0.00	0.00,100.00,　0.00	50.00,50.00,　0.00	100.00,100.00,　0.00
1	92.13,23.38,　107.57	10.93,79.36,　144.47	44.24,42.04,　570.22	75.20,97.00,　50.16
2	88.92,32.68,　129.40	9.51,74.95,　80.52	40.53,38.12,　406.51	73.60,91.18,　23.58
3	88.92,32.68,　110.11	9.51,74.95,　79.79	39.53,36.14,　378.84	69.07,85.36,　53.71
4	88.92,32.68,　110.11	9.51,74.95,　79.79	37.98,34.58,　347.97	67.49,79.87,　82.99
5	88.92,32.68,　110.11	12.32,73.20,　105.49	38.45,32.56,　322.02	67.49,79.87,　80.33
6	88.92,32.68,　110.11	12.32,73.20,　106.25	38.45,32.56,　321.18	67.49,79.87,　80.33

上表描述的实验中，初始化条件使用阀值 0.2，最大迭代次数 15。

表 4　NKM 算法使用明可夫斯基距离的聚类过程

Table 4　Clustering Progress of NKM Algorithm with Minkowski Distance

迭代	聚类 1	聚类 2	聚类 3	聚类 4
0	100.00,0.00,　0.00	0.00,100.00,　0.00	50.00,50.00,　0.00	100.00,100.00,　0.00
1	75.84,15.94,　358.26	9.51,74.95,　423.26	46.90,46.47,　1000.73	75.20,97.00,　86.41
2	62.81,17.44,　357.40	9.63,70.93,　192.25	52.97,43.91,　797.82	73.60,91.18,　54.69
3	53.64,10.07,　423.86	12.11,69.82,　289.03	58.77,48.60,　596.28	73.60,91.18,　53.80
4	53.64,10.07,　404.67	14.28,67.68,　332.19	57.27,50.09,　524.06	73.60,91.18,　53.80
5	53.64,10.07,　404.67	13.95,63.83,　496.59	65.42,50.74,　384.69	73.60,91.18,　53.80
6	53.64,10.07,　404.67	15.60,63.25,　546.05	68.58,50.12,　357.74	73.60,91.18,　53.80
7	59.05,8.14,　321.87	16.03,60.05,　635.61	68.58,50.12,　368.73	73.60,91.18,　53.80
8	66.81,6.21,　238.66	16.33,57.03,　700.21	68.58,50.12,　368.73	73.60,91.18,　53.80
9	66.81,6.21,　244.07	16.33,57.03,　696.15	68.58,50.12,　368.73	73.60,91.18,　53.80

上表描述的实验中，初始化条件使用阀值 0.2，最大迭代次数 15。

表 5　NKM 算法使用曼哈顿距离的聚类过程

Table 5　Clustering Progress of NKM Algorithm with Manhattan Distance

迭代	聚类 1	聚类 2	聚类 3	聚类 4
0	100.00,0.00,　0.00	0.00,100.00,　0.00	50.00,50.00,　0.00	100.00,100.00,　0.00
1	92.13,23.33,　124.82	10.93,79.35,　189.45	44.24,42.04,　702.36	75.20,97.00,　55.57
2	84.91,30.18,　199.30	9.51,74.95,　99.14	39.17,　39.65,　469.46	73.60,91.18,　29.56
3	84.91,30.18,　181.36	9.51,74.95,　91.20	38.00,37.64,　434.26	69.07,85.36,　71.10
4	84.91,30.18,　181.36	9.51,74.95,　91.20	36.22,36.08,　398.86	67.49,79.87,　109.10
5	84.91,30.18,　181.36	9.51,74.95,　91.20	36.22,36.08,　401.75	67.49,79.87,　105.18

上表描述的实验中，初始化条件使用阀值 0.2，最大迭代次数 15。

在第 3 部分，本文已经讨论过距离公式定义了数据空间的属性，从上述三个实验中可以看出，同一结束条件下，使用不同的距离公式对于聚类过程所带来的影响。

为了更直观的观察实验结果，现将上述 3 个实验的聚类结果使用散点图表示出来。使用的十字符号表示聚类质心，其他符号表示数据点；图中的图标描述 C1、C2、C3、

C4 分别是 4 个聚类的标识，CEN 则代表聚类质心；相对于初始化位置，在结果中的聚类质心都发生了较大的移动，且算法均在满足了收敛条件后结束迭代，可以证明，所得聚类结果达到了局部最优解。

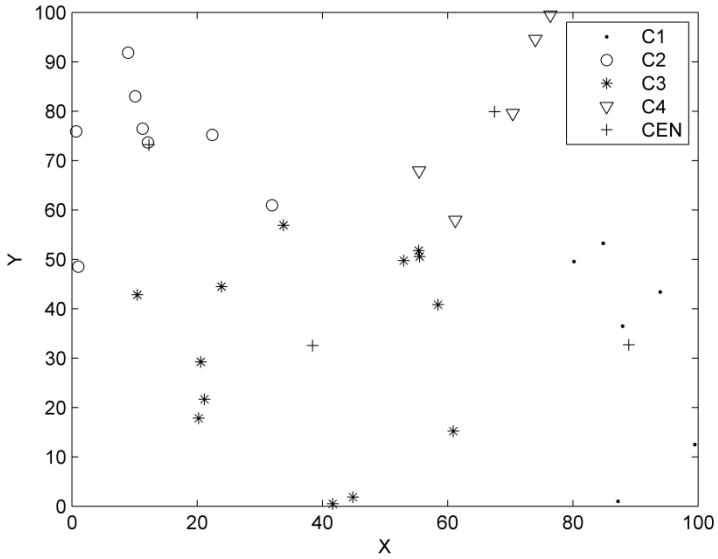

图 14　NKM 使用欧几里得距离的聚类结果

Fig 14　NKM Clustering Result Using Euclidean Distance

从图上可以看到，4 个聚类的分布较为紧密，然而聚类之间的分隔并不明显。可以看出数据点按照最近距离的原则被分配到了聚类中心；观察各聚类中心的邻域，已经在阀值内收敛。

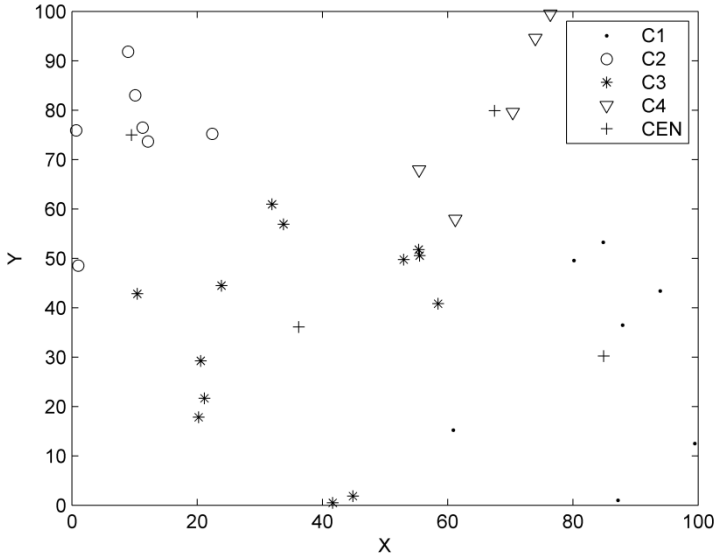

图 15　NKM 使用曼哈顿距离的聚类结果

Fig 15　NKM Clustering Result Using Manhattan Distance

使用曼哈顿距离可以得到矩形的聚类，这在特殊的情况下可以得到很好的应用。

可以看出，从聚类质心出发，数据点被分配在了 4 个大小不等的矩形当中。

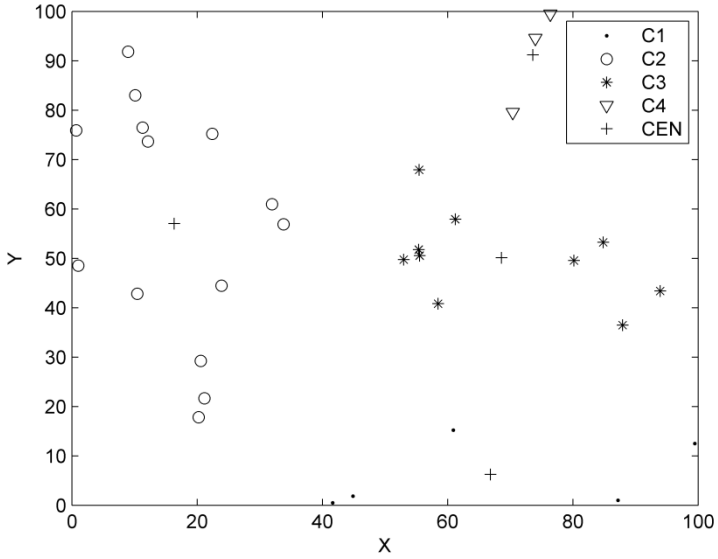

图 16　NKM 使用明可夫斯基距离的聚类结果

Fig 16　NKM Clustering Result Using Minkowski Distance

从聚类结果可以看出，本实验中使用明可夫斯基距离使得聚类之间的分隔比较明显，这很可能是因为在明可夫斯基距离公式下聚类质心的移动比较特殊。为了理解这个过程是如何进行的，下面绘制聚类质心移动的路径。

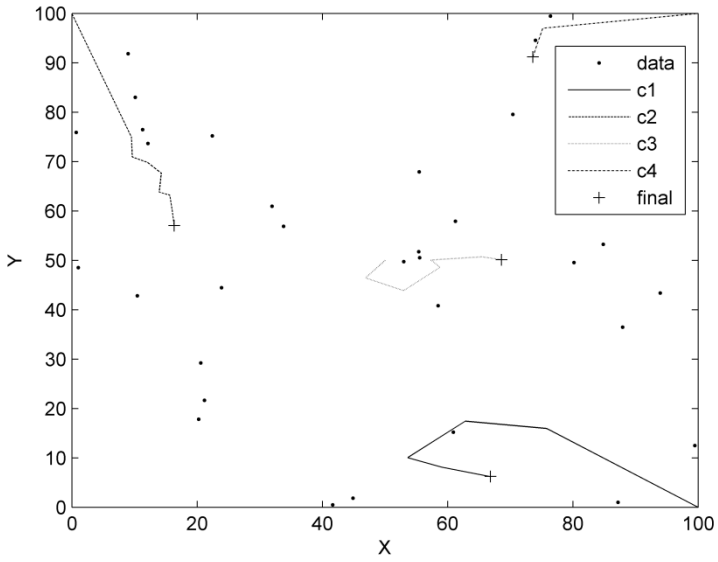

图 17　NKM 使用明可夫斯基距离的聚类轨迹

Fig 17　NKM Cluster Tracking Using Minkowski Distance

图 17 中的 c1、c2、c3、c4 分别是 4 个聚类质心移动的轨迹，而+号标志的位置则是算法终止时质心的位置。还可以看出这 4 个质心之间的相互作用，当 c2 逐步逼近最终位置，c1 和 c3 则改变了移动的方向被排斥。

经验证，NKM 与原始的 K 均值聚类算法将产生一致的结果。综上所述，本文在简单数据集上共进行了 3 次 NKM 算法的测试。接下来对结果进行分析。

4.2.8 NKM 性能评价

在不同的初始条件下进行实验，如上文所示，得到了一组在 Hadoop 上的 NKM 算法性能数据，见表 6。在当前设定下，这一组实验最终都达到了收敛阀值，聚类终止。表中的尺寸是聚类中每个数据点到聚类质心距离的加权和。

表 6　不同初始条件下 NKM 算法性能对照

Table 6　Performance Comparison of NKM Algorithm with Different Defaults

收敛阀值	最大迭代数	距离度量	初始尺寸	收敛尺寸	总迭代	总用时（s）	平均用时(s)
0.2	15	欧几里得	881.42	617.87	7	105	15.00
0.2	15	明可夫斯基	1868.66	1362.75	10	145	14.50
0.2	15	曼哈顿	821.2	779.49	6	85	14.17

从上表中可以看出,虽然在使用不同的距离度量所使用的迭代次数不一样，聚类结果的聚类尺寸都会比初始尺寸有所下降。虽然采用了不同的距离度量，平均每次迭代的使用时间约为 15s。

Dunn 指标是一个用于对聚类结果进行分析的通用指标，用v_D表示，在一个聚类集C上，其表达式为

$$v_D = \underset{c_i \neq c_j \wedge c_i, c_j \in C}{\operatorname{argmin}} \left\{ \frac{\delta(c_i, c_j)}{\underset{c_k \in C}{\operatorname{argmax}} \Delta(c_k)} \right\},$$

其中$\delta(c_i, c_j)$表示两个聚类的距离，设$d(\mathrm{x}, y)$为两个元素的距离度量，有

$$\delta(c_i, c_j) = \underset{x \in c_i, y \in c_j}{argmin} d(\mathrm{x}, y),$$

$\Delta(c_k)$表示一个聚类的直径，按照聚类中两个元素的最大距离计算。利用上述Dunn指标，可对聚类结果进行标准化的评价[19]，然而在NKM实现中，为了提高效率，采用了比较简单的“聚类尺寸”来度量聚类性能。

4.2.9 NKM 实现的优化

总结测试结果，将 K 均值算法移植到 Hadoop 平台上是可行的，使其可以利用并行处理和大数据所带来的优势。然而，为了保证算法的收缩性，需要考虑数据量很大的情况和对于分布式系统做出优化。

首先，ClusterInfo 作为一个聚集类型，必须能够保证内存的占用比较低，解决这个问题的方法在于，每次向 ClusterInfo 中添加数据点的时候，不是直接保存数据点，而是修改 ClusterInfo 中的聚类概要信息；因为在算法的计算中不需要从聚类中取出数据点信息。

其次，因为Hadoop是一个基于网络的分布式系统，而Mapper会产生大量的中间数据，因此在Map阶段完成以后的Shuffle过程会占用大量的网络资源。为了解决这个问题，Hadoop提供了Combiner，在Map后端执行，使得一些中间数据首先被合并为比较小的数据集，再将Combiner的输出分配到Reduce过程中。因此，可以为NKM算法编写一个Combiner从而减少网络传输开销。

每次迭代之间，都会有一个检查是否收敛的过程，原始的方式是检查每个数据点的分配是否发生变化。显然，当数据点很多的时候，这种方法将十分不理想。NKM采用的是计算两次迭代中聚类质心的移动距离，当小于一定阀值以后判定收敛。

进行上述优化以后的NKM算法，可伸缩性已经得到的显著的提高。

4.3 模糊K均值聚类算法

4.3.1 算法设计

模糊K均值算法简称FKM。在K均值算法中的数据点只能被分配到某一个聚类中，但是从概率论的角度来说，这不是在已知条件下的一个最好的假设。模糊K均值通过允许一个数据点在一定概率下属于某个聚类，可以使聚类算法的应用和性能得到扩展。

从NKM出发设计FKM，算法的流程类似，最大的区别在于分配阶段的具体算法。因此，通过对NKM实现的修改，可以得到一个简单的FKM算法实现。

通过修改以后，FKM与NKM有几个显著的不同。第一，因为采用了模糊计算，FKM对于数据点的分配不再完全由距离度量决定，最终分配的结果还取决于所使用的概率公式。第二，FKM会产生比NKM更多的中间结果，因此对此的优化显得极为重要。第三，FKM的聚类过程的稳定性会下降，因此在同样的收敛阀值下FKM可能需要更多的迭代次数。

4.3.2 FKM的Mapper类

FKM在Map阶段的运算与NKM差异较大，首先给出FKM的Mapper伪代码。

```
算法10：FKM_Mapper(key; value; clist)
cp←Parse_CompositePoint(value)
for c∈clist do
    P(c)←Measure_Probability(cp, c)
for c∈clist do
    if P(c)>0 then
       Output(Get_Id(c), Assigned(cp, P(c)))
```

FKM的Mapper部分算法与Naïve K-Means聚类的过程不同。首先从value中读出

数据点，然后进行 Measure_Probability(cp, c)过程，在这个过程中，根据表达式 2 计算数据点属于每个聚类的概率，保存在 P(c)中。

随后，根据 P(c)的结果，对每个聚类，函数 Assigned(cp, P(c))按照数学期望原理返回加权数据点，最后由 Output(key, value)函数输出，这些键值对将进入一个与 NKM 相似的 Reduce 过程。

4.3.3 小规模测试数据结果

为了对比测试FKM与NKM的性能差异,在这里将使用与4.2.4节相同的测试数据。相类似的，在 FKM 中也采用了不同的距离度量作为计算概率分布的基础函数，因此分别实验得出在不同距离度量下的聚类信息；根据 FKM 的特性，本实验对于聚类收敛阀值和最大迭代次数做了调整。下面列举使用不同距离度量时 FKM 的聚类结果。

表 7 FKM 算法使用欧几里得距离的聚类过程

Table 7 Clustering Progress of FKM Algorithm with Euclidean Distance

迭代	聚类 1	聚类 2	聚类 3	聚类 4
0	100.00,0.00, 0.00	0.00,100.00, 0.00	0.00,100.00, 0.00	100.00,100.00, 0.00
1	78.32,16.81, 141.65	10.68,73.91, 161.74	44.39,45.76, 566.96	75.94,82.67, 100.35
2	74.76,19.85, 171.96	12.17,68.83, 127.84	42.04,42.98, 322.33	73.20,76.33, 105.00
3	74.61,20.14, 171.53	12.66,68.08, 146.41	40.06,41.16, 282.57	71.91,72.60, 135.43

上表所描述的实验中，初始化参数最大迭代数设定为 10，收敛阀值为 5。

表 8 FKM 算法使用明可夫斯基距离的聚类过程

Table 8 Clustering Progress of FKM Algorithm with Minkowski Distance

迭代	聚类 1	聚类 2	聚类 3	聚类 4
0	100.00,0.00, 0.00	0.00,100.00, 0.00	0.00,100.00, 0.00	100.00,100.00, 0.00
1	70.13,15.26, 326.91	14.14,70.94, 458.27	45.33,47.06, 1144.75	72.79,74.61, 305.49
2	65.97,17.68, 358.39	14.91,64.21, 357.08	46.81,43.11, 644.61	63.77,72.65, 387.56
3	63.54,16.39, 362.48	15.08,63.36, 400.80	50.97,40.74, 570.16	59.49,72.23, 473.39
4	61.42,13.73, 352.90	15.36,61.57, 440.57	56.60,41.04, 535.70	57.30,72.13, 495.66

上表所描述的实验中，初始化参数最大迭代数设定为 10，收敛阀值为 10。

表 9 FKM 算法使用曼哈顿距离的聚类过程

Table 9 Clustering Progress of FKM Algorithm with Manhattan Distance

迭代	聚类 1	聚类 2	聚类 3	聚类 4
0	100.00,0.00, 0.00	0.00,100.00, 0.00	0.00,100.00, 0.00	100.00,100.00, 0.00
1	74.21,16.02, 184.70	11.67,71.87, 229.53	44.83,46.34, 668.14	74.43,79.65, 149.15
2	69.90,18.82, 220.81	13.24,66.34, 180.53	44.67,43.40, 382.57	70.24,74.70, 159.87
3	69.16,18.65, 223.61	13.87,65.06, 216.26	45.27,41.25, 336.68	67.91,72.40, 203.23

上表所描述的实验中，初始化参数最大迭代数设定为 10，收敛阀值为 5。

为了直观的观察聚类质心的最终位置，接下来给出使用上述数据绘制的散点图。图中图例注释的 data 为数据点，FKM 为本节实验的聚类终点质心，NKM 为用于对照的

NKM 算法所得结果的质心。因为对于 FKM 算法中概率计算权值的设定偏向于更明确的划分，FKM 实验所得结果与 NKM 应该比较接近，而又有一定的不同。

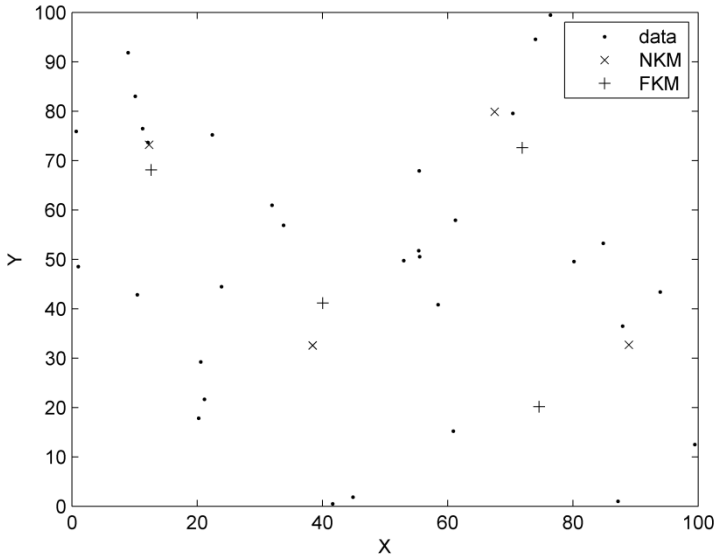

图 18　FKM 使用欧几里得距离的聚类结果

Fig 18　FKM Clustering Result Using Euclidean Distance

上图的结果与 NKM 对应结果对比，差异值约为 8 到 24 个单位，FKM 产生的结果更加的集中于中心点。

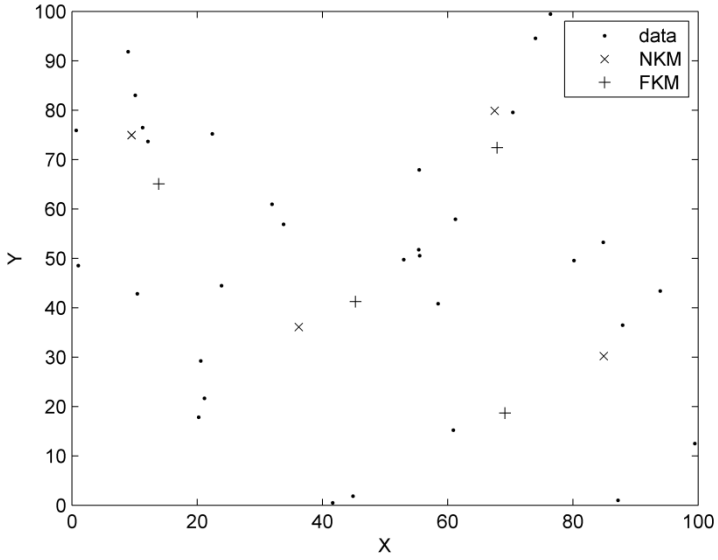

图 19　FKM 使用曼哈顿距离的聚类结果

Fig 19　FKM Clustering Result Using Manhattan Distance

上图的结果与 NKM 对应结果相比，聚类质心均有约 8 到 26 个单位的偏移。同时，因为使用了模糊算法，曼哈顿距离所得到的划分已经不再是一个矩形区域，而是一个类似矩形的分布函数图像。

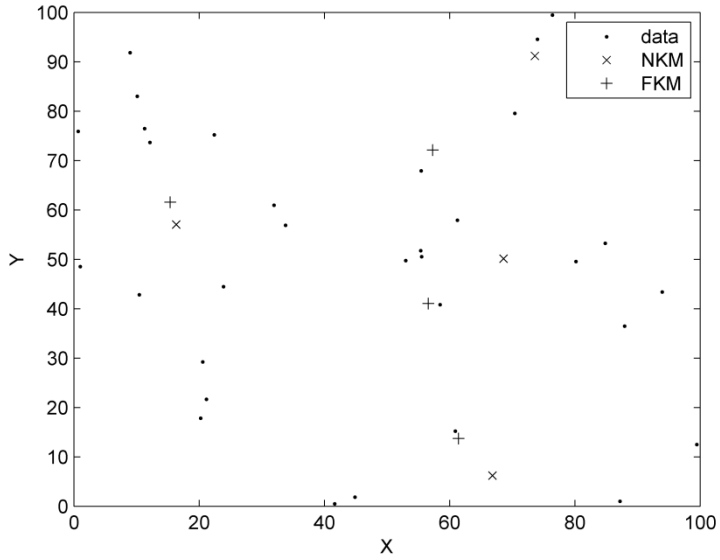

图 20　FKM 使用明可夫斯基距离的聚类结果

Fig 20　FKM Clustering Result Using Minkowski Distance

同样的，为了与 NKM 算法进行对比，在这里将 FKM 使用明可夫斯基距离的质心移动轨迹表示出来。这里的轨迹与图 17 相比有类似的地方，但细节有区别。可以看出，距离度量的选用对聚类过程有着重大的影响。

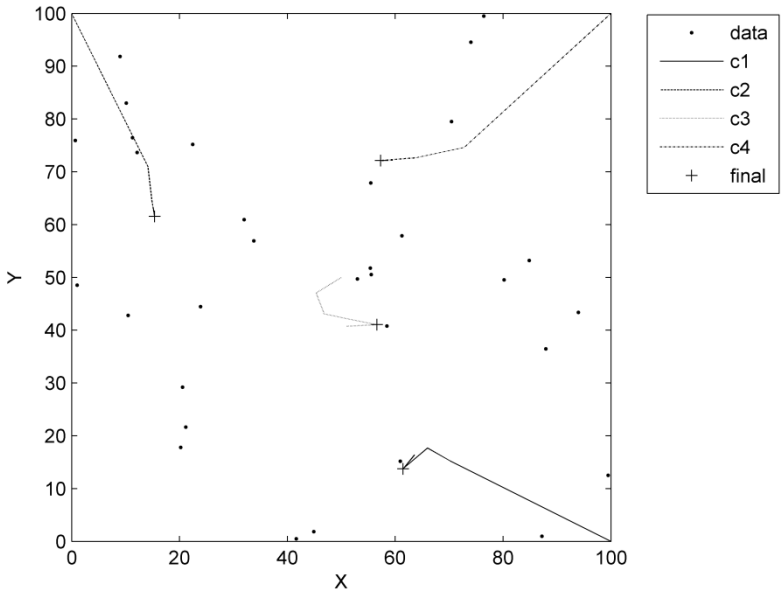

图 21　FKM 使用明可夫斯基距离的聚类轨迹

Fig 21　FKM Cluster Tracking Using Minkowski Distance

为了更快的收敛，上述实验中的阀值都较大，当减少阀值的时候，会使收敛速度减慢，并可能出现无法收敛的情况。下面进行了几组实验，改变了初始设定中的阀值，以验证阀值的变化对于迭代次数和聚类结果的影响。图 22 中显示的是 FKM 使用曼哈顿距离的聚类结果在阀值为 5、1、0.2 时的对比。

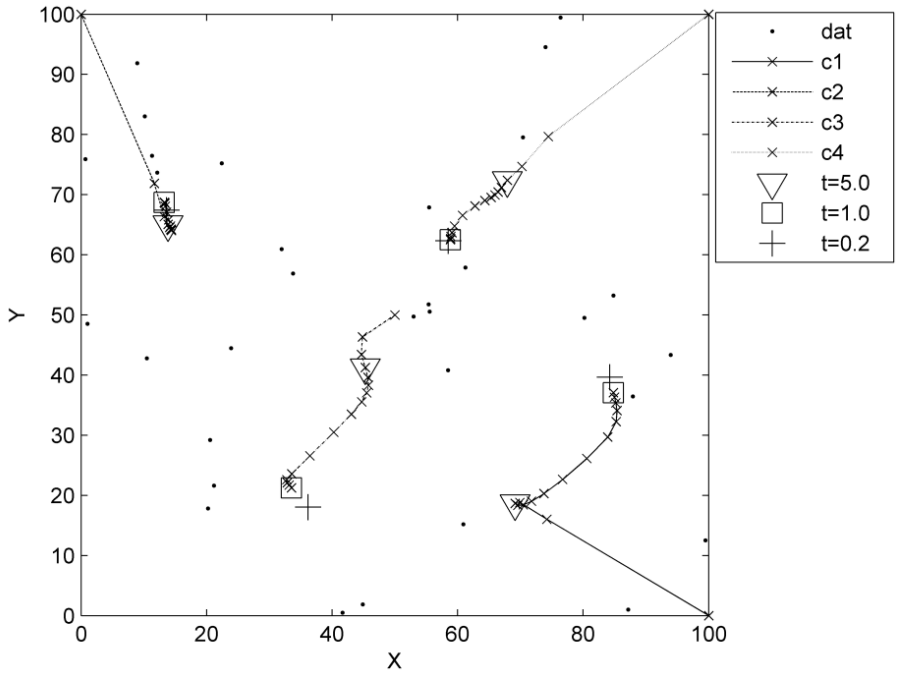

图 22　明可夫斯基距离下使用不同阀值的 FKM 聚类轨迹

Fig 22　FKM Cluster Tracking Using Minkowski Distance with Different Threshold

图 22 的图例中 dat 表示数据点，c1、c2、c3、c4 表示 4 个聚类的移动轨迹，最后 t=5.0、t=1.0、t=0.2 分别表示阀值设定为这些数值时的最终结果。在本次实验中，设定阀值为 0.2 时所有聚类没有在最大迭代数内收敛可以看出，随着阀值不断减小，聚类质心产生了一定程度的移动。由于设定阀值的目的在于在一个接近局部最优点的位置停止聚类，可以看出在本次实验中的若设定阀值为 1.0，可以得到比较好的效果。

经验证，FKM 与移植前模糊 K 均值聚类算法将产生类似的结果。另外，改变初始设定，本节还使用 FKM 算法进行了多组实验，这些实验的结果将在 4.3.4 节中讨论。

4.3.4　FKM 性能评估

综合分析上述实验的结果和所产生的性能数据，可以总结于表 10 中。

表 10　不同初始条件下 FKM 算法性能对照

Table 10　Performance Comparison of FKM Algorithm with Different Defaults

收敛阀值	最大迭代数	距离度量	初始尺寸	收敛尺寸	收敛	总迭代	总用时（s）	平均用时（s）
5	10	欧几里得	970.70	735.94	是	4	60	15.00
1	20	欧几里得	970.70	661.40	是	12	177	14.75
0.2	20	欧几里得	970.70	653.66	否	21	316	15.04
10	10	明可夫斯基	2235.42	1824.83	是	5	75	15.00
1	20	明可夫斯基	2235.42	1717.23	否	21	320	15.23
0.2	20	明可夫斯基	2235.42	1717.23	否	21	324	15.42
5	10	曼哈顿	1231.52	979.78	是	4	61	15.25
1	20	曼哈顿	1231.52	838.35	是	16	245	15.31
0.2	20	曼哈顿	1231.52	834.03	否	21	327	15.57

对比表 6，FKM 各实验结果的初始尺寸均较 NKM 有明显的增加，这是应该在计算聚类尺寸的时候，FKM 会考虑距离比较远的数据点，加上一个适当的权值。当设定的收敛阀值逐渐缩小，或者是总迭代数不断提高时，聚类收敛尺寸也在下降，但还未达到使用 NKM 实验中的值；因为聚类尺寸最终都有显著的减少，可以判断 FKM 算法是合理有效的。

对于模糊 K 均值算法进行评价可以同样采用 Dunn 指标作为基础。在这里，在两点之间的距离时考虑到一个非负浮点数的权值，在性能指数中使用数学期望，即可令 Dunn 指标函数处理 FKM 所得到的归属矩阵$k \times n$。

4.3.5 FKM 性能优化

本文中使用的FKM算法容易受到离群值的影响，Nikhil R. Pal等人提出了一个对于模糊K均值算法的改进措施，考虑数据点在聚类中的典型性，可以改善离群值对聚类质心带来的偏移[20]。然而，这个优化并未考虑到可伸缩性，而本文的重点在于使移植后的算法能够处理海量数据。应当从其他思路入手，寻找优化的途径。

4.4 流数据 K 均值算法

4.4.1 算法设计

Georgios Christopoulos 提出了一个修改后的 K 均值算法能够支持数据流。在每次迭代过程以后，将上次结果所得的 k 个质心加权以后附加到新的输入中来，从而简单的将算法扩展为支持数据流的输入。这个算法的数据流如图所示：

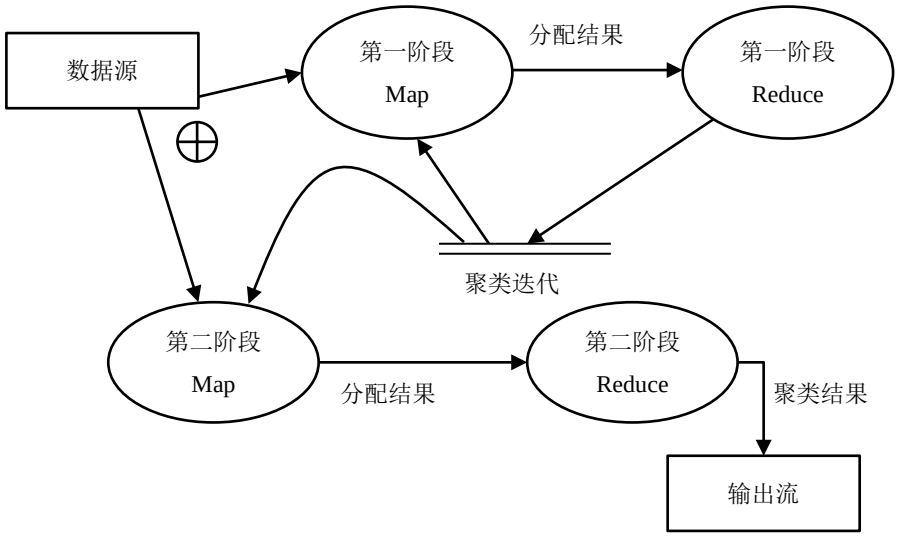

图 22　流数据 K 均值算法的数据流图

Fig 22　Data flow Diagram of Data Stream K-Means Algorithm

4.4.2 算法性能评估

运用几个大型的实际数据集，Georgios在SoftNet提供的一个 13 个节点的HOP集

群上验证了这个算法设计的有效性。他指出，这个算法可以处理大量的随着时间变化的数据，并且利用了Hadoop的强大的可伸缩性的优势，通过使用HOP所提供的在线聚集和连续查询功能，有几个实际的好处：第一，能够查看早期的聚类结果有时候可以在任务运行的早期得到需要的结果，以便于结束任务并节约计算资源[21]；第二，在线算法可以用于事件监控。为了进行对比，在此引用Georgios在不同节点数量情况下所得的加速比数据；节点采用的是 4GB内存和四核心处理器。在这个实验中，聚类阀值为 0.5，最大迭代次数为 15 次，输入数据为 10 个 100MB的数据文件[22]。

表 11　在线 K 均值算法在不同节点数量下的加速比

Table 10　Scale up of Online K-Means Algorithm with Different Number of Nodes

节点数量	运行时间（s）	加速比
4	22979.22	1.00
8	10910.37	2.11
12	7645.79	3.01

表 11 中加速比以节点数为 4 个的实验结果作为基准 1,计算得出通过提高节点的数量，所得到的性能的提升比例，一般来说这个比例函数是一个节点数量的线性函数的高阶无穷小，从上表中的数据来看，在线 K 均值算法在计算节点数变化的加速比β可以拟合一个节点数量n的对数函数$f(n) = \log n - C$。另作参考，在 12 个节点计算的情况下，改变输入数据的数量，可以得到另一组性能数据，见表 12。

表 12　在线 K 均值算法在不同数据量下的计算效率

Table 12　Efficiency of Online K-Means Algorithm with Different Amount of Data

数据量（MB）	运行时间（s）	吞吐量（MB/s）
300	3096.10	0.09
600	5294.25	0.11
1024	7645.79	0.13

从表 12 中可以看出，在线 K 均值算法的吞吐量随着数据量的增大而提高，在表中数据所表示的范围中，吞吐量增量保持稳定。

5　全分布式测试

本节将对在一个完全分布式的 Hadoop 集群上，利用一个模拟生成的数据集，称为数据集 DS,对于上述朴素 K 均值算法 NKM 和模糊 K 均值算法 FKM 进行多方面的性能评估。通过控制变量法和一些基准参数，在不同的聚类环境设定、不同的数据量大小的情况下，算法的性能出现什么变化，以及聚类结果的优劣。

5.1　数据描述

为了测试系统的处理性能，需要使用不同数据量大小的测试用例。数据集 DS 包含了在几个不同规模的测试用例；这些测试用例使用了同样的混合分布作为母函数，

因此在它们之上的测试具有一定的可比性。

5.1.1 混合分布和初始位置

根据表达式 1，定义数据集 DS 的混合分布为$T(p)$，其中k表示成分数量，n为样本量，p为样本，A_i为成分随机变量，$T(A_i)$为成分先验概率，$T(p|A_i)$为在A_i上样本p的概率分布。A_4是一个覆盖(0, 0)到(1000, 1000)的矩形平均分布，A_1、A_2、A_3是三个类高斯分布，他们的平均值和标准差在 X 轴和 Y 轴上的分量μ_i^x、μ_i^y和σ_i^x、σ_i^y见表 13。

表 13 混合分布 T 的成分参数

Table 13 Component Parameters of Mixture T

成分	μ_i^x	μ_i^y	σ_i^x	σ_i^y	$T(A_i)$
A_1	400	200	80	40	0.16
A_2	300	600	80	60	0.44
A_3	60	60	700	700	0.36
A_4	–	–	–	–	0.04

根据上表所示参数，可以导出样本点坐标$p(x, y)$生成公式

$$p \leftarrow \begin{cases} x = -2\sigma_i^x \cdot \cos 2\pi\phi \cdot \sqrt{\log \phi} + \mu_i^x \\ y = -2\sigma_i^y \cdot \sin 2\pi\phi \cdot \sqrt{\log \phi} + \mu_i^y \end{cases}.$$

利用上式，可以在确定了参数n的情况下生成对应的测试用例。为了更直观的表示数据集 DS 的混合分布，现利用此分布产生一个较小的用例，图 23 表示的是这个用例的数据点按照混合分布中的类高斯成分标识所得散点图，其中图例所示 gus1、gus2、gus3 是混合分布中的类高斯成分A_1、A_2、A_3，noise 则表示A_4；图中还标出了随后实验中使用的两种聚类质心的初始化位置 def1 和 def2，它们各包含 3 个质心。

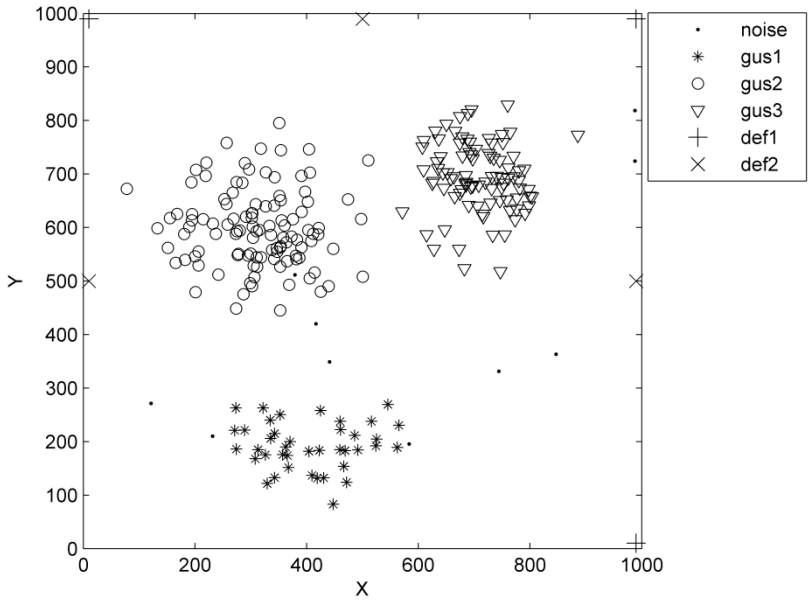

图 23 模拟测试数据混合分布

Fig 23 Hybrid Distribution of Simulated Test Data

在上图的图例中，def1 和 def2 分别表示两个聚类初始化位置，详细信息见表 14 所示。

表 14　测试用例详细描述

Table 14　Details of the Test Case

名称	聚类 1	聚类 2	聚类 3
def1	990, 990	10, 990	990, 10
def2	10, 500	500, 990	990, 500

表 14 中的分别列出了 3 个聚类的 2 种初始化质点位置，在 def1、def2 中，3 个聚类都初始化在混合分布的边缘部分，包围了大部分样本点；def1 和 def2 的初始化位置相互间隔

5.1.2　测试用例

根据数据量不同，测试用例分为 DS0、DS1、DS2、DS3 四个，均使用了图 23 所示的混合分布，它们的详细信息如表 15 所示。

表 15　测试用例详细描述

Table 15　Details of the Test Case

名称	行数（样本数n）	总大小（MB）	文件数
DS0	2500	0.1	4
DS1	25000	1	4
DS2	250000	10	4
DS3	2500000	100	4

5.2　评测平台配置

利用现有的硬件资源，对于算法性能的评价将搭建一个具有 6 个节点的 Hadoop 分布式并行计算集群，每个节点均使用了统一的硬件和软件配置，其中各节点的系统配置如表 16 所示。

表 16　节点计算机描述

Table 16　Description of the Node Computer

组件	性能指标
处理器	Intel Core 2 Duo E7500 2.93GHz x64 核心数 2
内存	3980MB DDR3
硬盘	HDD SATAII 178GB
操作系统	Ubuntu 12.04 LTS x64
Hadoop 版本	Apache Hadoop release 0.20.2

这 6 个节点分别命名为 node1，node2，…，node6；根据 Hadoop 硬件架构的特点，选取 node1 作为 JobTracker 以及 HDFS 的名称节点，其他 5 个节点作为 TaskTracker 和 HDFS 的数据节点，节点处于一个有着 100Mbps 传输速率的以太网的同一网段中，如图 24 所示。根据实际测试，该网络环境较为理想，平均延迟小于 10ms，在网络资

源使用集中的情况下，可能发生拥堵。

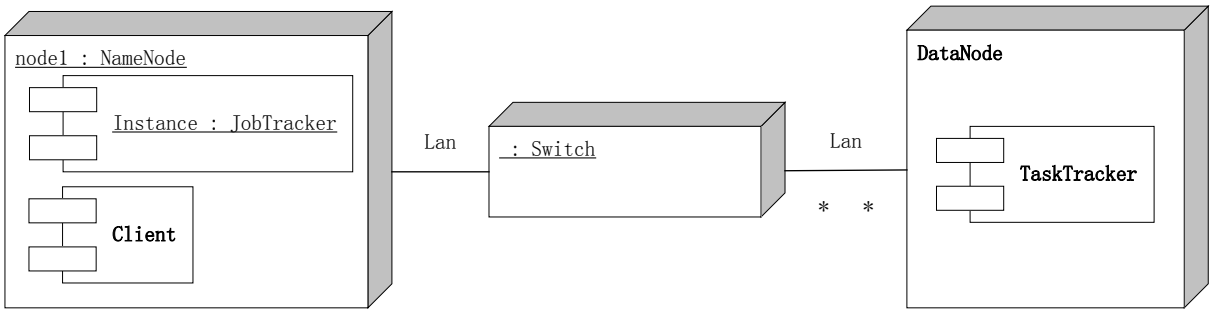

图 24 测试系统的部署图

Fig 24 Deployment Diagram of the Test System

现在首先在同一网段中配置好 6 个节点计算机，检查硬件。无误，在每台计算机上安装官方发行的 Ubuntu 12.4 LTS x64 版，安装时确定磁盘分区。当安装完成以后，在各节点的 hosts 文件中添加每个节点的名称映射。随后，在各节点设置好专用账号和 ssh 免密码连接，检查是否成功。随后，在每台计算机上安装 jdk1.6，配置好环境变量，此时完成安装 Hadoop 的准备工作。

接下来，在每个节点上安装 Apache Hadoop 0.20.2 官方发行版，并配置相应的 xml 文件，定义 JAVA_HOME 为 Java 的安装路径。在 core-site.xml 的 configuration 下加入：

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/hadoop/hadoop-${user.name}</value>
        <description>Temporary directories for hadoop.</description>
    </property>
</configuration>
```

在 hdfs-site.xml 的 configuration 下加入：

```
<configuration>
    <property>
        <name>dfs.replication</name>
```

```
        <value>1</value>
    </property>
</configuration>
```

在 mapred-site.xml 的 configuration 下加入：

```
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
</configuration>
```

这样一来就完成了 Hadoop 的基本配置。随后，在 Hadoop 目录下执行

```
$ bin/hadoop namenode  -format
$ bin/start-hdfs.sh
$ bin/start-mapred.sh
$ bin/hadoop dfsadmin  -report
```

若命令正常完成，则表示 Hadoop 集群已经成功配置并启动，现将 Hadoop 关闭。

5.3 性能评估

5.3.1 测试方法与过程

测试开始前，已经完成了所有的系统和软件配置。接下来，首先将组成集群的计算机启动，等待启动完毕以后，使用指定的用户登陆 Ubuntu 系统。

在 node1 上启动 HDFS，等待过程完成；启动 JobTracker 和 TaskTracker，等待启动过程完成，约 5 分钟。

当 Hadoop 平台进入正常运行模式，开始进行实验。每次测试中，首先将实验所用的测试用例和聚类初始化信息文件上传到 HDFS；随后，配置并上传所使用的 NKM 聚类或者 FKM 聚类 Jar 包；等待计算过程完毕，从 HDFS 中下载实验输出数据和日志文件，保存在本地测试编号文件夹内。

5.3.2 测试结果

在数据集 DS 上，使用 NKM 算法进行 16 次实验，使用 FKM 算法 12 次实验，得到了 2 组实验结果，对于实验中 NKM 算法和 FKM 算法的性能描述分别见表 17、表 18。所有的实验中，收敛阀值均设定为 1.00，最大迭代数为 15，采用欧几里得距离度量。实验变量为计算节点数量和初始化聚类位置，每次实验结果中记录了算法运行时间和

实际迭代次数。

表 17　NKM 算法的数据集 DS 模拟测试结果对照

Table 17　Comparison of the Simulated Test Results on Dataset DS with NKM

节点数	测试用例	初始聚类	总迭代	总用时（s）	平均用时（s）
1	DS0	def1	5	77	15.40
1	DS0	def2	11	215	19.54
5	DS0	def1	5	119	23.80
5	DS0	def2	11	264	24.00
1	DS1	def1	4	126	31.50
1	DS1	def2	11	357	32.45
5	DS1	def1	4	111	27.75
5	DS1	def2	11	353	32.09
1	DS2	def1	4	320	80.00
1	DS2	def2	12	935	77.91
5	DS2	def1	4	186	46.50
5	DS2	def2	12	515	42.91
1	DS3	def1	4	2353	588.25
1	DS3	def2	12	6893	574.41
5	DS3	def1	4	1032	258.00
5	DS3	def2	12	2382	198.50

从表 17 中可以看出，在数据量较小的时候，因为没有网络传输开销，单一节点的运算速度更快；随着计算量的不断增大，并行处理的优势不断增强，此时主要由计算量决定运算时间。

表 18　FKM 算法的数据集 DS 模拟测试性能对照

Table 18　Comparison of the Simulated Test Performance on Dataset DS with FKM

节点数	测试用例	初始聚类	总迭代	总用时（s）	平均用时（s）
1	DS0	def1	5	81	16.20
1	DS0	def2	13	254	19.53
5	DS0	def1	5	130	26.00
5	DS0	def2	13	356	27.38
1	DS1	def1	5	177	35.40
1	DS1	def2	12	424	35.33
5	DS1	def1	5	147	29.40
5	DS1	def2	12	290	24.16
1	DS2	def1	5	702	140.40
1	DS2	def2	13	1781	137.00
5	DS2	def1	5	315	63.00
5	DS2	def2	13	711	54.69

将表 18 中数据与表 17 对比可以发现，使用 FKM 算法进行测试时，计算节点数的增加对于性能提升的影响更为显著，这是因为 FKM 算法的计算量要比 NKM 大，而且更为集中。

考察各组测试聚类的最终结果，以 DS0 为例，现将使用 DS0 所进行的几项测试的聚类结果标识出来，并用图例加以区分，其中 mix 是数据集使用的混合分布实例，nkm1 是使用 NKM 算法在初始化聚类 def1 下的聚类结果，nkm2 则是使用 def2 的聚类结果，以此类推；在单个节点上进行的实验结果和多个节点上的结果基本一致，因此不再区分；在不同的样本量上的聚类结果类似，因此不再做 DS1、DS2、DS3 上的聚类结果图，在表 19 中共同列举这 4 个测试用例的聚类结果

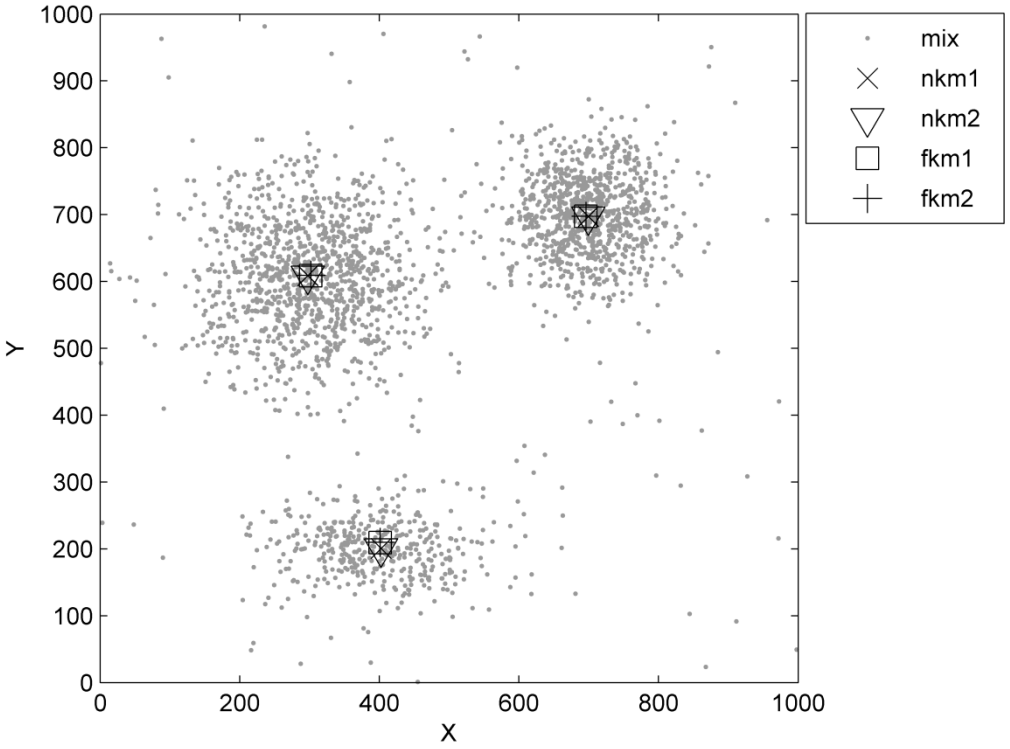

图 25　测试用例 DS0 上的聚类结果对比

Fig 25　Comparison of Clustering Results on Test Case DS0

从图中可以看出，在 4 种不同配置下，都得到了类似的聚类结果，距离小于 5 个单位。

表 19　数据集 DS 上聚类结果

Table 19　Clustering Results on Dataset DS

测试用例	聚类算法	初始化	聚类质点 1	聚类质点 2	聚类质点 3
DS0	NKM	def1	402.44, 200.61	297.59, 608.78	699.34, 697.24
DS0	NKM	def2	402.44, 200.61	297.59, 608.78	699.34, 697.24
DS0	FKM	def1	695.91, 697.30	301.87, 608.83	401.45, 209.38
DS0	FKM	def2	401.20, 209.78	301.95, 609.00	696.07, 697.28
DS1	NKM	def1	702.03, 698.63	298.56, 603.26	404.56, 200.96
DS1	NKM	def2	404.47, 201.04	298.57, 603.30	702.03, 698.63
DS1	FKM	def1	699.14, 697.91	302.85, 603.34	401.59, 210.03
DS1	FKM	def2	401.36, 210.39	302.93, 603.50	699.27, 697.89
DS2	NKM	def1	700.91, 698.52	296.57, 602.64	405.63, 200.69

续表 1

测试用例	聚类算法	初始化	聚类质点 1	聚类质点 2	聚类质点 3
DS2	NKM	def2	405.53, 200.72	296.57, 602.66	700.94, 698.48
DS2	FKM	def1	698.38, 697.77	300.92, 602.44	402.94, 209.11
DS2	FKM	def2	402.71, 209.49	300.99, 602.60	698.52, 697.75
DS3	NKM	def1	701.14, 698.91	296.78, 602.90	405.68, 200.88
DS3	NKM	def2	405.59, 200.94	296.79, 602.94	701.17, 698.88

上表中的每项测试初始化收敛阀值为 1.00，最大迭代数为 15，并采用欧几里得距离度量。

观察表 19 可以发现，聚类结果中的聚类 1 和聚类 3 在不同的设置下会改变位置，而聚类 2 保持不变，不过这与聚类结果的准确性无关，因为聚类标识符作为一个人为设定的标志，并不会作为分类依据。综合考虑，在数据集 DS 上 FKM 和 NKM 算法均有很高的准确性。

5.3.3 评价指标

接下来，综合上述实验结果，本文将从两个角度讨论在集群上所带来的性能提升。

第一个方面是在全部5个计算节点启用时的实验结果在不同算法和不同测试用例上得到的性能提升情况。分别考虑 NKM 和 FKM 使用了聚类初始化位置 def1 和 def2 的测试结果，可以得到性能折线图，图的横轴以数据量的对数（log/MB）为单位，纵轴是每次迭代的平均计算时间。在图 26 中共有 5 条折线，其中 liner 是一条参考线，它表示当数据量为 0.1MB 时用时 1s 的一个线性函数$y = 10x$，其他 4 条则是使用 NKM 算法和 FKM 算法所得的性能变化。

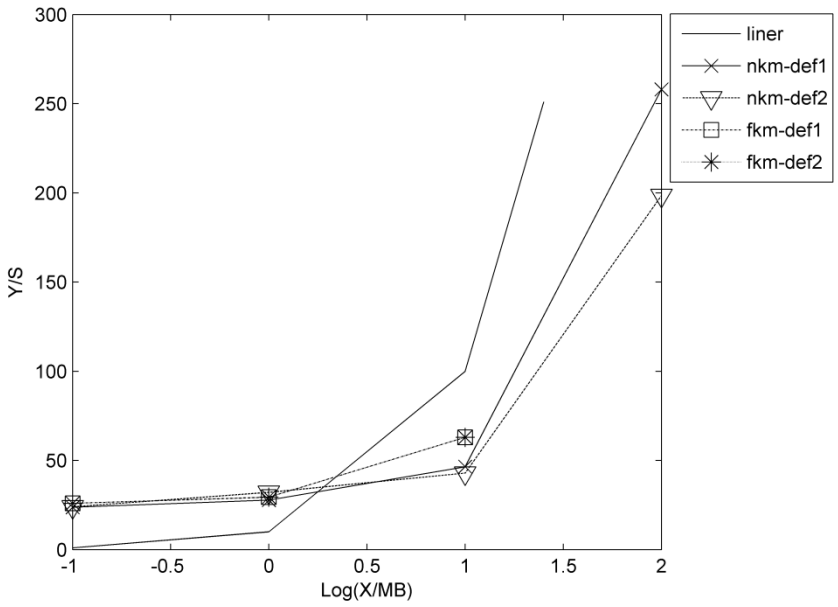

图 26 数据集 DS 上所测算法性能变化

Fig 26 Chang in Performance of Tested Algorithms on Dataset DS

从上图中可以看出，随着数据量的增大，NKM 和 FKM 的计算性能增强；算法实测时间复杂度在实验范围内小于线性增长速度。取 NKM 在 def1 初始条件下的结果，研究其计算效率，见表 20 所示。

表 20　NKM 算法在不同数据量下的计算效率

Table 20　Efficiency of NKM Algorithms with Different Amount of Data

数据量（MB）	含样本量	运行时间（s）	吞吐量（MB/s）
0.1	2500	119	0.000
1	25000	111	0.009
10	250000	186	0.053
100	2500000	1032	0.096

第二个方面是考虑在同等实验条件下，通过增加计算节点数量达到性能提升的目的。因此，对于 NKM 算法选定作为参考的测试用例 DS3，对于 FKM 算法选定 DS2，它们各自在 def1 和 def2 初始化的情况下的运行时间见图 27 所示。

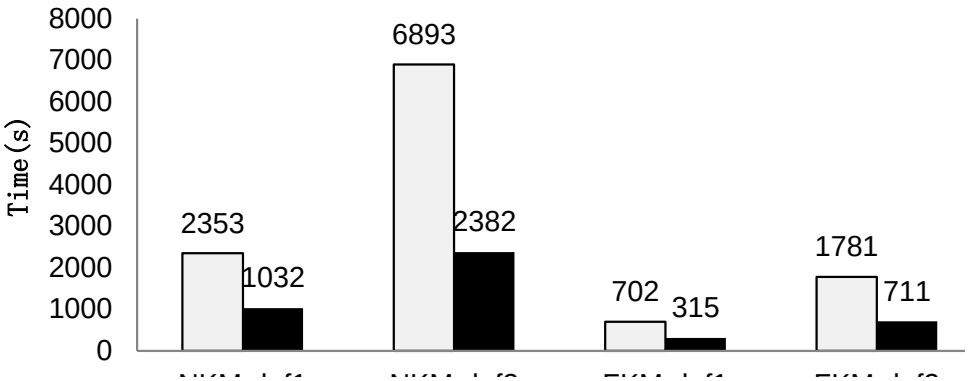

图 27　制定测试用例上所测算法的运行时间

Fig 27　Time Consumption of Tested Algorithms on Designated Test Case

参照在同样实验条件下使用 1 个计算节点的耗时，集群上的加速比以 5 个计算节点耗时为分子，1 个计算节点耗时为分母来计算，见表 21。

表 21　所测算法在 5 个计算节点下的加速比

Table 21　Scale up of Tested Algorithms Using 5 Nodes

所测算法	初始化	总迭代数	加速比
NKM	def1	4	2.28
NKM	def2	12	2.89
FKM	def1	5	2.22
FKM	def2	13	2.50

从上表中可以看出，本文所设计的 NKM 和 FKM 算法在 Hadoop 集群上实测取得了明显的效果，取得最大加速比 2.89，达到了理论最大值 5.00 的 58%。根据现有实验不难推论，当数据量增加的时候，加速比会越来越接近最大值，因为固定损耗时间在总运行时间中的比重会不断下降。利用这个原理，可以通过一个表达式计算理论最大

加速比$S_{max}^{(a)}$，

$$S_{max}^{(a)} = \frac{T_{max}^{(min)} - T_{min}^{(min)}}{T_{max}^{(max)} - T_{min}^{(max)}},$$

其中，$T_i^{(j)}$表示算法使用 j 个计算节点在测试用例 DSi 上的运行时间，a代表得到实验结果所使用的算法；最后求得将数据带入可得

$$\begin{cases} S_{max}^{(NKM)} \approx 3.15 \\ S_{max}^{(FKM)} \approx 3.34 \end{cases}.$$

显然，这个估计值将随着数据量的增大而不断逼近正确值；同时，通过上述结果，可以得出，适当的加大数据量，本文所讨论的 NKM 算法和 FKM 算法在一个具有 5 个节点的集群上可以分别达到 3.15 和 3.34 的加速比。

6 结论

机器学习已经得到了广泛的应用，然而对于越来越大的数据量，对于一种具有可伸缩性，能够利用廉价集群处理机器学习问题的算法将出现很大的需求。本文的研究目的是如何将机器学习与大数据的并行计算结合起来。从这个出发点，首先简单介绍了 Hadoop 平台，并广泛的讨论了几种聚类以及相关原理。

原始的聚类算法大多数用于单机运行，而本文将 K 均值聚类算法 NKM 和模糊 K 均值聚类算法 KFM 结合 MapReduce 编程范型重新设计以后，使得它们可以运行在 Hadoop 集群之上。通过一系列的实验结果，利用各类图形和数据表，本文验证了移植后算法的有效性和准确性。在一个拥有 5 个计算节点的 Hadoop 集群上，本文通过一个模拟测试用例集对移植后的算法进行了大量的实验；最后，通过分析实验结果，得出在当前测试用例下所测 NKM 算法加速比为 2.89，FKM 的加速比为 2.50；除此之外还对在更大数据情况下的加速比做出了合理预测。

今后的工作主要有，第一，对于已经成功移植的两种算法，可以继续寻求如何进行其他的优化，使得它们在多个节点的集群上的运行效率能够提高；第二，其他尚未移植到 Hadoop 环境中的聚类算法中，将选取与实际问题最相关的进行移植；第三，本文中对于聚类算法的讨论主要使用了二维欧几里得空间作为数据空间，对于实际应用中的其他数据空间算法的性能，还需要进一步的研究；第四，本文对于数据流挖掘的研究还未展开，接下来的工作是讨论具体算法的实现和一个实时数据挖掘系统的设计。

参考文献

[1] Russell S J, Norvig P, Davis E, et al. Artificial intelligence: a modern approach[M]. Prentice hall, Englewood Cliffs: 2010.

[2] Wang G, Butt A R, Pandey P, et al. Using realistic simulation for performance analysis of mapreduce setups[C]. Proceedings of the 1st ACM workshop on Large-Scale system and application performance. ACM, 2009: 19-26.

[3] White T. Hadoop: The definitive guide[M]. O'Reilly Media, Inc., 2012.

[4] Owen O'Malley. TeraByte Sort on Apache Hadoop[J]. Yahoo!, 2008.

[5] Dhruba Borthakur. The Hadoop Distributed File System Architecture and Design[M]. The Apache Software Foundation, 2007.

[6] Jiang D, Ooi B C, Shi L, et al. The performance of mapreduce: An in-depth study[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 472-483.

[7] Condie T, Conway N, Alvaro P, et al. Online aggregation and continuous query support in mapreduce[C]. ACM SIGMOD. 2010: 1115-1118.

[8] Sharma A. A Survey On Different Text Clustering Techniques For Patent Analysis[J]. International Journal of Engineering, 2012, 1(9).

[9] Shrestha D. Text mining with Lucene and Hadoop: Document clustering with feature extraction[J]. Wakhok University, 2009.

[10]Henao R, Lucas J E. Efficient hierarchical clustering for continuous data[J]. arXiv preprint arXiv:1204.4708, 2012.

[11] 曲福恒, 胡雅婷, 马驷良. 基于模拟退火的无监督核模糊聚类算法[J]. 吉林大学学报 (理学版), 2009, 47(2): 317-322.

[12] Langley P. The changing science of machine learning[J]. Machine Learning, 2011, 82(3): 275-279.

[13] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[14] Anaya-Sánchez H, Pons-Porrata A, Berlanga-Llavori R. A document clustering algorithm for discovering and describing topics[J]. Pattern Recognition Letters, 2010, 31(6): 502-510.

[15] Nazeer K A A, Sebastian M P. Improving the Accuracy and Efficiency of the k-means Clustering Algorithm[C]. Proceedings of the World Congress on Engineering. 2009, 1: 1-3.

[16] 黄志华, 温步瀛, 王国乾. 可间断运行的 K-means 聚类算法[J]. 计算机应用研究, 2009, 26(6): 2053-2055.

[17] Oyelade O J, Oladipupo O O, Obagbuwa I C. Application of k means clustering algorithm for prediction of students' academic performance[J]. arXiv, 2010: 1002,2425.

[18] 潘兴江.Hadoop 平台下基于 SVM 主动学习的分类算法研究[D]. 广州:华南理工大学,2011.

[19] Aliguliyev R M. Performance evaluation of density-based clustering methods[J]. Information Sciences, 2009, 179(20): 3583-3602.

[20] Havens T C, Chitta R, Jain A K, et al. Speedup of fuzzy and possibilistic kernel c-means for large-scale clustering[C].Fuzzy Systems (FUZZ), IEEE, 2011: 463-470.

[21] Pansare, Niketan, Vinayak R. Borkar, Chris Jermaine, Tyson Condie. Online aggregation for large mapreduce jobs[J]. PVLDB 4, no. 11 (2011): 1135-1145.

[22] Georgios Christopoulos. Fast, Parallel Stream Clustering using Hadoop Online[D]. IEEE, 2011.

[23] Sean Owen, Robin Anil, et al. Mahout in Action[M]. Manning Publications, 2012.

[24] Trevor Hastie, Robert Tibshirani, et al. The Element of Statistical Learning, Second Edition[M]. Springer, 2008.

[25] Christopher M. Bishop. Pattern Recognition and Machine Learning[M]. Springer, 2006.

[26] Dhruba Borthakur, Joydeep Sen Sarma, et al. Apache Hadoop Goes Realtime at Facebook[C]. ACM, 2011.

[27] Jianwu Wang, Daniel Crawl, Ilkay Altintas. Kepler + Hadoop : A General Architecture Facilitating Data-Intensive Applications in Scientific Workflow Systems[C]. ACM, 2009.

[28] 周品. Hadoop 云计算实战[M].清华大学出版社, 2012.

[29] 王敬昌.基于 Hadoop 分布式计算架构的海量数据分析[J].数字技术与应用,2010,07:6-7.

[30] 曹风兵.基于 Hadoop 的云计算模型研究与应用[D].重庆:重庆大学,2011.

[31] Tziortziotis N, Blekas K. A Model Based Reinforcement Learning Approach Using On-Line Clustering[C]. Tools with Artificial Intelligence (ICTAI), IEEE, 2012, 1: 712-718.

[32] Barbakh W, Fyfe C. Online clustering algorithms[J]. International Journal of Neural Systems, 2008, 18(03): 185-194.

[33] Ene A, Im S, Moseley B. Fast clustering using MapReduce[C]. Proceedings of the 17th ACM SIGKDD, 2011: 681-689.

致　谢

本论文是在陈义明老师的悉心指导和热情关怀下完成的。从选题到开题报告，从写作提纲，到一遍又一遍地指出每稿中的具体问题，严格把关，循循善诱。陈老师是我一名非常值得尊敬的老师，治学严谨，学识渊博，思想深邃，视野雄阔，为我营造了一种良好的精神氛围。

再次诚挚的感谢陈老师在 Hadoop 集群的配置过程中提供的指导和帮助。本文中测试所使用的集群主要是由陈老师调试完成的，若没有陈老师的精心指导，这个过程可能会变得相当的困难。

同时，我必须感谢湖南农业大学信科院的老师们，你们在日常教学以及课余时间为我解答的诸多问题，使我受益匪浅；更重要的是，我在老师们的指导下知识和内心不断的成长，使我成为了一个名副其实的学生。谢谢我的同学们，是你们的陪伴获得了多姿多彩的课余生活。与你们交流，也总能让我学到新的东西。

再次感谢信科院的全体老师和同学们，在我遇到问题的时候为我热心的提供帮助，感谢大家！

附录

附录 1：NKM 算法关键代码

附录 2：FKM 算法关键代码

附录 1

下面给出 NKM 算法关键部分，Mapper 类、Reducer 类和驱动 run()； load()方法与 checkConv()方法的具体代码没有给出。

```
public class NaiveKmeans extends Configured implements Tool {
    public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, LongWritable, Text> {
        public void configure(JobConf job) {
            cenPath = job.get("Info.CentroidFile");
            measure = Integer.parseInt(job.get("Info.Measure"));
            try {
                this.load();
            } catch (IOException e) {
            }
        }
        public void map(LongWritable key, Text value,
                OutputCollector<LongWritable, Text> output,
                Reporter reporter) throws IOException {
            Scanner sn = new Scanner(value.toString());
            CompositePoint cp = new CompositePoint();
            double md = Distance.VERY_FAR;
            double di = 0;
            long k = 0;
            if (!sn.hasNextDouble()) return;
            cp.set(sn.nextDouble(), sn.nextDouble(), sn.nextDouble());
            for (int i=0; i<clist.length; i++) {
                di = Distance.compute(clist[i], cp, measure);
                if (di < md) {
                    md = di;
                    k = clist[i].getId();
                }
            }
            output.collect(new LongWritable(k), new Text(cp.toString()+" "+(md*cp.getW().get())));
        }
    }
    public static class Reduce extends MapReduceBase
    implements Reducer<LongWritable, Text, LongWritable, Text> {
        public void configure(JobConf job) {
            try {
                this.load();
            } catch (IOException e) {
            }
        }
        public void reduce(LongWritable key, Iterator<Text> values,
```

```
            OutputCollector<LongWritable, Text> output,
            Reporter reporter) throws IOException {
        double s = 0;
        double x = 0;
        double y = 0;
        double w = 0;
        double sum = 0;
        CompositePoint cp = null;
        CompositePoint cluster = null;
        while (values.hasNext()) {
            Scanner sn = new Scanner(values.next().toString());
            cp = new CompositePoint(sn.nextDouble(), sn.nextDouble(), sn.nextDouble());
            sum += sn.nextDouble();
            w = cp.getW().get();
            x += cp.getX().get() * w;
            y += cp.getY().get() * w;
            s += w;
        }
        cluster = new CompositePoint(x/s, y/s, s);
        output.collect(key, new Text(cluster.toString()+" "+sum));
    }
}
public static class Combiner extends MapReduceBase
implements Reducer<LongWritable, Text, LongWritable, Text> {
    public void reduce(LongWritable key, Iterator<Text> values,
            OutputCollector<LongWritable, Text> output,
            Reporter reporter) throws IOException {
        double s = 0;
        double x = 0;
        double y = 0;
        double w = 0;
        double sum = 0;
        CompositePoint cp = null;
        CompositePoint cluster = null;
        while (values.hasNext()) {
            Scanner sn = new Scanner(values.next().toString());
            cp = new CompositePoint(sn.nextDouble(), sn.nextDouble(), sn.nextDouble());
            sum += sn.nextDouble();
            w = cp.getW().get();
            x += cp.getX().get() * w;
            y += cp.getY().get() * w;
            s += w;
        }
        cluster = new CompositePoint(x/s, y/s, s);
```

```
            output.collect(key, new Text(cluster.toString()+" "+sum));
        }
    }

    public static class LastReduce extends MapReduceBase
    implements Reducer<LongWritable, Text, LongWritable, Text> {
        public void reduce(LongWritable key, Iterator<Text> values,
                OutputCollector<LongWritable, Text> output,
                Reporter reporter) throws IOException {
            CompositePoint cp = null;
            ClusterInfo ci = new ClusterInfo(key.get());
            //output.collect(key, new Text("BEGIN CLUSTER"));
            while (values.hasNext()) {
                cp = new CompositePoint(values.next().toString());
                ci.add(cp);
                output.collect(key, cp.toText());

            }
            //output.collect(key, ci.toText());
        }
    }
    public int run(String[] args) throws Exception {
        int iter = 0;
    long stime = System.currentTimeMillis();
        System.out.println(COMMON_STAMP+"Job Started at "+stime+".");
        if (!parseArgument(args)) {
            return printUsage();
        }

        try {
           PrintStream ps=new PrintStream("/home/lrc007/hop/stats.txt");
           std = System.out;
           System.setOut(ps);
       } catch (FileNotFoundException e) {
           e.printStackTrace();
       }
        System.out.println(COMMON_STAMP+"Job Started at "+stime+".");

    do {
        long t1 = System.currentTimeMillis();
        conf = new JobConf(getConf(), NaiveKmeans.class);
        if (pipe) {
            conf.setFloat("mapred.snapshot.frequency", snapFreq);
            conf.setBoolean("mapred.map.pipeline", pipe);
```

```
        }
        conf.setNumMapTasks(numMapper);
        conf.setNumReduceTasks(numReducer);
        conf.setMapperClass(MapClass.class);
        conf.setCombinerClass(Combiner.class);
        conf.setReducerClass(Reduce.class);
        conf.set("Info.Iteration", String.valueOf(iter));
        conf.set("Info.Measure", String.valueOf(similarityMeasure));
        if (iter == 0) {
             conf.set("Info.CentroidFile", cenPath);
        } else {
             conf.set("Info.CentroidFile", cenPath+String.valueOf(iter-1));
        }
        FileInputFormat.setInputPaths(conf, new Path(inPath));
        FileOutputFormat.setOutputPath(conf, new Path(cenPath+String.valueOf(iter)));
        clist = this.load(conf.get("Info.CentroidFile"));
        JobClient.runJob(conf);
        System.out.println(COMMON_STAMP+"Time Consumed:
"+(System.currentTimeMillis()-t1)/1000+"s");
        iter++;
    } while (iter <= maxIteration && !checkConv(cenPath+String.valueOf(iter-1),convergence));

    //Summarize Points
    long t1 = System.currentTimeMillis();
    conf = new JobConf(getConf(), NaiveKmeans.class);
        if (pipe) {
             conf.setFloat("mapred.snapshot.frequency", snapFreq);
             conf.setBoolean("mapred.map.pipeline", pipe);
        }
        conf.setNumMapTasks(numMapper);
        conf.setNumReduceTasks(1);
        conf.setMapperClass(MapClass.class);
        conf.setReducerClass(LastReduce.class);
        conf.set("Info.Iteration", String.valueOf(iter));
        conf.set("Info.Measure", String.valueOf(similarityMeasure));
        if (iter == 0) {
             conf.set("Info.CentroidFile", cenPath);
        } else {
        conf.set("Info.CentroidFile", cenPath+String.valueOf(iter-1));
        }
        FileInputFormat.setInputPaths(conf, new Path(inPath));
        FileOutputFormat.setOutputPath(conf, new Path(outPath));

    JobClient.runJob(conf);
```

```
    //Summarize Points

        long durtime = System.currentTimeMillis() - stime;
        System.setOut(std);
        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new NaiveKmeans(), args);
        System.exit(res);
    }

}
```

附录 2

下面给出 FKM 算法关键代码即 FKM 的 Mapper 类，与 NKM 的主要区别在于 Mapper 类，因此其他部分不再书写。

```
public class FuzzyKmeans extends Configured implements Tool {
static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, LongWritable, Text> {
        public void configure(JobConf job) {
            cenPath = job.get("Info.CentroidFile");
            measure = Integer.parseInt(job.get("Info.Measure"));
            try {
                this.load();
            } catch (IOException e) {
            }
        }

        public void map(LongWritable key, Text value,
                OutputCollector<LongWritable, Text> output,
                Reporter reporter) throws IOException {
            Scanner sn = new Scanner(value.toString());
            CompositePoint cp = new CompositePoint();
            double k = 0;
            double md = 0;
            double di[] = new double[clist.length];
            double rec[] = new double[clist.length];
            int j=0;
            if (clist.length == 0) return;
            if (!sn.hasNextDouble()) return;
            cp.set(sn.nextDouble(), sn.nextDouble(), sn.nextDouble());
            for (int i=0; i<clist.length; i++) {
                di[i] = Distance.compute(clist[i], cp, measure);
```

```
                rec[i] = di[i];
                if (di[j]<di[i])
                    j = i;
            }
            k = di[j];
            for (int i=0; i<clist.length; i++) {
                di[i] = k-di[i];
                if (i == j) continue;
                md += Math.pow(di[i], 4)*cp.getW().get();
            }
            if (md <= NEAR_ZERO)
                md = NEAR_ZERO;
            for (int i=0;i<clist.length; i++) {
                if (i == j) continue;
                double p = Math.pow(di[i], 4)*cp.getW().get()/md;
                if (p <= NEAR_ZERO) continue;
                output.collect(new LongWritable(clist[i].getId()), new Text(cp.toString()+"
"+rec[i]));
            }

        }
    }
}
```