基于 MapReduce 的潜在人际关系发现方法设计与实现

邵华松 [1)]

[1)] 澳门城市大学，数据科学学院，澳门特别行政区，999078

摘 要 在数据科学领域中，人际关系发现作为一个重要研究方向，需要对数量庞大的关系数据进行分析，而单机计算模式难以满足经济性、时效性方面的要求。本研究针对目前单机计算模式在人际关系发现方向的不足，通过阅读相关文献并结合实践探索，研究并实现了一种分布式的、基于 MapReduce 的潜在人际关系发现方法，重点研究以下内容：

(1) 基于 Docker 的完全分布式 Hadoop 环境搭建。

(2) Hadoop Streaming 工具理解与应用。

(3) MapReduce 原理与具体实现。

通过集成以上研究内容，本研究在基于 Docker 的完全分布式 Hadoop 环境下，利用 Hadoop Streaming 工具，结合 Python 语言编写的 Mapper 与 Reducer 高效地完成海量关系数据分析并实现潜在人际关系的发现。

关键词 大数据；人际关系；Docker；Hadoop；MapReduce

Design and Implementation of a Method for Discovering Potential Interpersonal Relationship Based on MapReduce

Shao Huasong[1)]

[1)]Department of Data Science, City University of Macau, 999078, Macao Special Administrative Region

Abstract

In the field of data science, interpersonal relationship discovery, as an important research direction, requires the analysis of a huge amount of relationship data, and the single-computer computing model can hardly meet the requirements in terms of economy and timeliness.

This study addresses the shortcomings of the current single-computer computing model in the direction of interpersonal relationship discovery. By reading relevant literature and combining practical exploration, a distributed, MapReduce-based approach to potential interpersonal relationship discovery is studied and implemented, focusing on the following:

(1) Docker-based fully distributed Hadoop environment construction.

(2) Understanding and application of Hadoop Streaming tools.

(3) MapReduce principles and concrete implementation.

By integrating the above research contents, this study efficiently completes massive relationship data analysis and achieves potential interpersonal relationship discovery under a fully distributed Hadoop environment based on Docker, using the Hadoop Streaming tool, combined with Mapper and Reducer written in Python language.

Keywords Big Data; Interpersonal Relationships; Docker; Hadoop; MapReduce

收稿日期：2021 年 12 月 13 日； 最终修改稿收到日期：2021 年 12 月 13 日

1 引言

1.1 研究目的与意义

网络技术的迅速发展推动着网络社交平台的进步，显著降低了人际关系的建立成本，并产生了海量的数字化人际关系数据。数字化人际关系数据实质上是一张庞大的图，通常以邻接矩阵的形式存储，其中包含着诸多在相关领域中具有重要意义的信息。如在社交商业领域中，潜在人际关系具有较高的商业利用价值。

目前，主流的潜在人际关系发现方法主要基于单机计算模式。在实际应用过程中，基于单机计算模式的潜在人际关系发现方法在数据量较小的数据集上表现优秀；但在面对海量数据时难以满足经济性与时效性方面的要求。

本研究拟采用虚拟化技术与分布式计算相结合的思路，实现基于 MapReduce 的潜在人际关系发现方法，在经济性与时效性方面满足相关领域的要求。

基于 MapReduce 的潜在人际关系发现方法技术难点在于以下两点：Hadoop 环境的搭建、MapReduce 编程逻辑与编程实现。针对以上两点，本研究采用 Docker 搭建完全分布式 Hadoop 环境，并利用 Hadoop Streaming 工具降低 MapReduce 逻辑理解难度与编程实现复杂度。

本研究采用基于 Docker 搭建的完全分布式 Hadoop 环境，利用 Hadoop Streaming 工具，结合 Python 语言编写的 Mapper 与 Reducer，构建一个可快速部署的潜在人际关系发现方法，实现从海量数据中分布式提取潜在人际关系。本研究最终实现的方法所采用框架与工具成熟度高、设计逻辑简单，具有较高的替换性与扩展性，研究成熟后可进行大规模推广并应用于人际关系研究的其他方向。

1.2 研究内容及技术路线

本研究代码开发以 Python 语言为主，开发环境为 macOS11.4(m1)，Docker 镜像环境为 Ubuntu16.04(arm)。基于 Docker 实现完全分布式 Hadoop 搭建；基于 Python 实现 Mapper 与 Reducer 编写；基于 Hadoop Streaming 实现 MapReduce 过程。本研究技术路线如图 1 所示。

本研究内容如下：

(1) 完全分布式 Hadoop 搭建。利用 Docker 区别于虚拟机的容器引擎特性，实现可快速部署的完全分布式 Hadoop 小集群搭建。

(2) Mapper 与 Reducer 编写。基于 Python 语言，结合 MapReduce 运算逻辑编写符合规范的 Mapper 与 Reducer。

(3) Hadoop Streaming 调用。利用 Hadoop 自带的 Hadoop Streaming 工具，结合 Python 语言编写的 Mapper 与 Reducer 实现 MapReduce 过程。

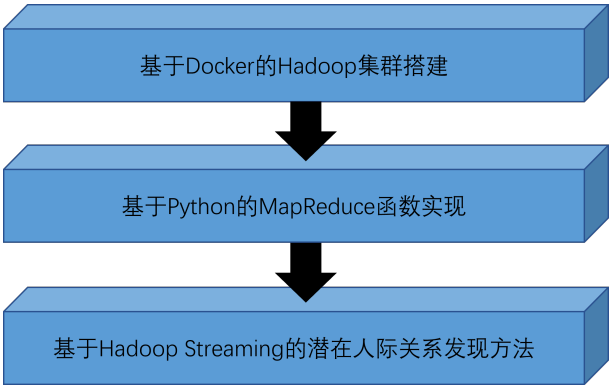

图 1 技术路线图

2 基于 Docker 的完全分布式 Hadoop 环境搭建

2.1 Docker 配置

Docker 是一个可以实现虚拟化的、使用沙箱机制的开源应用容器引擎，其特性允许将应用与相关依赖包打包至一个可移植的容器中，以便于发布到采用 Linux、macOS 或 Windows 操作系统的机器上。本研究基于虚拟化与便捷可移植性的需求，采用 Docker 进行 Hadoop 环境搭建，而非使用传统虚拟机软件。

2.1.1 Docker 具体配置过程

本研究 Docker 镜像采用 16.04 版本 Ubuntu，逐步配置 Hadoop 依赖环境，具体过程如下：

(1) 镜像拉取与启动。首先通过 Docker 的 pull 命令拉取位于 DockerHub 上的开源 Ubuntu16.04 镜像；随后通过 Docker 的 run 命令启动容器进行进一步配置。如表 1 所示：

表 1 镜像拉取与启动命令

命令	含义
docker pull ubuntu:16.04	拉取 ubuntu:16.04 镜像
docker run -it ubuntu:16.04	启动 ubuntu:16.04 容器

(2) 系统更新与 JDK 安装。首次启动容器时通过 apt-get 命令更新系统，随后进行 JDK 安装；JDK 版本选择与 Hadoop3.2.1 相符合的 JDK8。如表 2 所示：

表 2　表说明系统更新与 JDK 安装命令

命令	含义
apt-get update	更新系统
apt-get install java-1.8.0-openjdk	安装 JDK8

(3) ssh 服务端与客户端配置。通过 apt-get 安装 ssh 服务端与客户端并配置免密登陆，编辑环境变量实现容器启动时自动开启 ssh。如表 3 所示：

表 3　ssh 服务端与客户端配置命令

命令	含义
apt-get install openssh-server	安装 ssh 服务端
apt-get install openssh-client	安装 ssh 客户端
ssh-keygen -t rsa -P ""	生成密钥
cat id_rsa.pub » authorized_keys	设置免密登陆

2.1.2　Hadoop 配置

Hadoop 是一个由 Apache 基金会所开发的分布式系统基础架构，支持忽略底层地分布式程序开发并充分利用集群的算力进行高速运算和存储。该框架最核心的设计为 HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，而 MapReduce 则为海量的数据提供了计算。本研究基于稳定性与高效性的需求，选择配置 3.2.1 版本 Hadoop。

2.1.3　Hadoop 具体配置过程

本研究采用 3.2.1 版本 Hadoop，逐步配置完全分布式 Hadoop 集群，具体过程如下：

(1) Hadoop 下载与安装。通过 wget 命令从 Apache 基金会网站下载 Hadoop3.2.1 并解压。

(2) 配置环境变量。配置 JDK 路径、HDFS 节点、Hadoop 路径等。如图 2：

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-arm64
export JRE_HOME=${JAVA_HOME}/jre
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib
export PATH=${JAVA_HOME}/bin:$PATH
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_LIBEXEC_DIR=$HADOOP_HOME/libexec
export JAVA_LIBRARY_PATH=$HADOOP_HOME/lib/native:$JAVA_LIBRARY_PATH
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HDFS_DATANODE_USER=root
export HDFS_DATANODE_SECURE_USER=root
export HDFS_SECONDARYNAMENODE_USER=root
export HDFS_NAMENODE_USER=root
export YARN_RESOURCEMANAGER_USER=root
export YARN_NODEMANAGER_USER=root
```

图 2　具体环境变量配置

(3) 配置 Hadoop 相关设置。配置详见附录 1。

3　MapReduce 编程逻辑与编程实现

3.1　MapReduce 编程逻辑

MapReduce 是由 Google 公司研究并提出的一种面向大规模数据处理的并行计算模型。该模型允许编程人员编写非分布式并行编程的程序实现软件的分布式运行，其核心为 Map(映射) 函数与 Reduce(归约) 函数。

3.1.1　Map 过程

Map 过程的输入为经分片后的大量原始数据，在本研究中，Map 任务由 Hadoop 集群的 Master 分派到每个 Worker 节点上。Map 过程将原始数据解析成一批键值对并输出。

3.1.2　Reduce 过程

Reduce 过程的输入为经过 Shuffle 的 Map 过程输出的键值对，在本研究中，Reduce 任务由 Hadoop 集群的 Master 分派到每个 Worker 节点上。Reduce 过程将输入键值对按键对值进行归约合并，形成最终统计输出。

3.2　MapReduce 编程实现

本研究原始数据输入为 “< 用户, 人际关系集合 >” 的形式，如图 3 所示，从中寻找具有潜在人际关系的用户，输出形式为 “< 用户, 潜在人际关系用户 < 共同人际关系 >>”。

```
0   1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36
1   0,5,20,135,2409,8715,8932,10623,12347,12846,13840,13845,14085,20075,21556,22939,23528,28193,29724,
2   0,117,135,1220,2755,12453,24539,24714,41456,45046,49927,6893,13795,16659,32828,41878
3   0,12,41,55,1532,12636,13185,27552,38737
```

图 3　原始数据形式

3.2.1　Map 函数实现

以输入 “0 1,2,3” 为例，该输入代表用户 0 与用户 1、用户 2、用户 3 已经存在人际关系。Map 函数将实现在用户现有人际关系中构建所有可能的潜在人际关系组合作为键，用户本身作为值，输出对应键值对的功能；并为防止出现潜在人际关系与实际人际关系冲突的现象，Map 函数还将输出以用户与其本身人际关系作为键，以一个全局不存在用户作为值的键值对，供 Reduce 函数在归约时进行判断。本研究基于 Python3+ 实现 Map 函数，主要使用 sys 库，具体代码详见附录 2。

3.2.2　Reduce 函数实现

Reduce 函数将对所有潜在关系进行归约，并以共同人际关机数量进行排序。本研究基于 Python3+ 实现 Reduce 函数，主要使用 sys 库，具体代码详见

附录 3。

4 基于 MapReduce 的潜在人际关系发现方法实现

Hadoop Streaming 是 Hadoop 发行版附带的实用程序。该实用程序允许指定任何可执行文件或脚本作为 Mapper 或 Reducer 来创建和运行 MapReduce 作业。

在 Hadoop Streaming 过程中，Mapper 和 Reducer 将从 stdin 逐行读取输入并将输出发送到 stdout，其将创建一个 MapReduce 作业，将作业提交到适当的集群，并监视作业的进度直至完成。当为映射器指定了可执行文件时，每个映射器任务都会在映射器初始化时将可执行文件作为单独的进程启动。当映射器任务运行时，它会将其输入转换为行并将这些行提供给进程的标准输入。同时，映射器从进程的标准输出中收集面向行的输出，并将每一行转换为键值对，作为映射器的输出收集。默认情况下，直到第一个制表符的行的前缀是键，该行的其余部分将是值。

当为 Reducer 指定了可执行文件时，每个 Reducer 都会将可执行文件作为单独的进程启动，然后初始化 Reducer。当 Reducer 运行时，它将其输入键值对转换为行并将这些行提供给进程的标准输入。同时，Reducer 从进程的标准输出中收集面向行的输出，将每一行转换为键值对，作为 Reducer 的输出收集。

4.1 方法实际运行

基于 MapReduce 的潜在人际关系发现方法运行主要由以下步骤构成：

(1) 基于 Docker 的 Hadoop 集群启动。启动三个节点，分别命名为 Master、Slave1 与 Slave2，达到完全分布式 Hadoop 集群要求；随后在 Master 节点中启动 Hadoop 服务。

(2) 基于 Hadoop Streaming 的 MapReduce 计算。对附录 2 及附录 3 中所展示的 Mapper 与 Reducer 赋予权限；向 HDFS 上传原始数据文件；调用 Hadoop Streaming 工具启动 MapReduce 进行潜在人际关系发现。运行过程如图 4 所示：

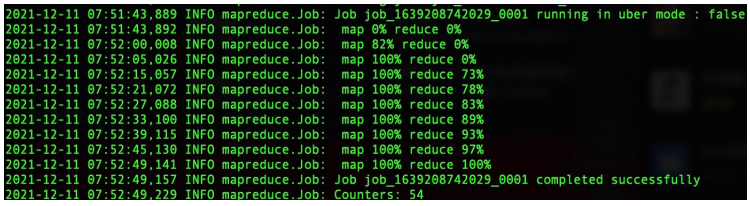

图 4 MapReduce 运行过程

(3) 检查运行结果。部分结果如图 5 所示：

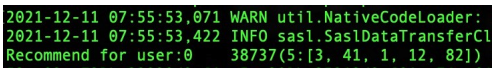

图 5 潜在人机关系发现结果 (部分)

致　谢　感谢应作斌教授的悉心教导

附录 1.

* **附录内容**

core-site.xml:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://master:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/root/hadoop/tmp</value>
  </property>
</configuration>
```

hdfs-site.xml:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>2</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/root/hadoop/hdfs/name</value>
  </property>
  <property>
    <name>dfs.namenode.data.dir</name>
    <value>/root/hadoop/hdfs/data</value>
  </property>
</configuration>
```

mapred-site.xml:

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
  <property>
    <name>mapreduce.application.classpath</name>
      <value>
        /usr/local/hadoop/etc/hadoop,
        /usr/local/hadoop/share/hadoop/common/*,
        /usr/local/hadoop/share/hadoop/common/lib/*,
        /usr/local/hadoop/share/hadoop/hdfs/*,
        /usr/local/hadoop/share/hadoop/hdfs/lib/*,
        /usr/local/hadoop/share/hadoop/mapreduce/*,
        /usr/local/hadoop/share/hadoop/mapreduce/lib/*,
        /usr/local/hadoop/share/hadoop/yarn/*,
        /usr/local/hadoop/share/hadoop/yarn/lib/*
      </value>
  </property>
</configuration>
```

yarn-site.xml:

```
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
      <value>master</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

附录 2.

* **附录内容**

```
'''
@author ShaoHuasong
@desc A mapper for hadoop-streaming
@date 2021/11/28
@pyversion 3
'''
import sys
for line in sys.stdin:
  line = line.strip()
  data = line.split("\t")
  if len(data)==2:
    user = data[0]
    friends = data[1].split(',')
    for firstPointer in range(len(friends)):
      print(user + "," + friends[firstPointer] + "\t" + "-1")
      for secondpointer in range(len(friends)):
        if firstPointer!=secondpointer:
          print(friends[firstPointer] + "," +
          friends[secondpointer] + "\t" + user)
```

附录 3.

* **附录内容**

```
'''
@author ShaoHuasong
@desc A reducer for hadoop-streaming
@date 2021/11/28
@pyversion 3
'''
import sys
currentUser1 = None
currentUser2 = None
user1 = None
user2 = None
mutualCount = []
outputStream = []
for line in sys.stdin:
  line = line.strip()
  data = line.split()
  user12 = data[0].split(',')
  user1 = int(user12[0])
```

```
user2 = int(user12[1])
mutualUser = int(data[1])
if(currentUser1 == user1):
    if(currentUser2 == user2):
        mutualCount.append(mutualUser)
    elif(-1 in mutualCount):
        mutualCount.clear()
        currentUser2 = user2
        mutualCount.append(mutualUser)
    else:
        outputStream.append([currentUser2,
        mutualCount.copy()])
        mutualCount.clear()
        currentUser2 = user2
        mutualCount.append(mutualUser)
else:
    if(currentUser1 == None):
        currentUser1 = user1
        currentUser2 = user2
        mutualCount.append(mutualUser)
        outputStream.append(currentUser1)
    else:
        if(-1 in mutualCount):
            if(len(outputStream)==1):
                currentUser1 = user1
                currentUser2 = user2
                outputStream.clear()
                mutualCount.clear()
                mutualCount.append(mutualUser)
                outputStream.append(currentUser1)
        else:
            output = "Recommend for user:" +
            str(outputStream[0]) + "\t"
            recommendationInOrder =
            sorted(outputStream[1:],
            key=lambda mutualNum:
            len(mutualNum[1]),reverse=True)
            for data in recommendationInOrder:
                output = output + str(data[0]) + "(" +
            str(len(data[1])) + ":" + str(data[1]) + ") "
            print(output)
            currentUser1 = user1
            currentUser2 = user2
            outputStream.clear()
            mutualCount.clear()
            mutualCount.append(mutualUser)
            outputStream.append(currentUser1)
        else:
            outputStream.append([currentUser2,
            mutualCount.copy()])
            output = "Recommend for user:" +
            str(outputStream[0]) + "\t"
            recommendationInOrder =
            sorted(outputStream[1:],
            key=lambda mutualNum:
            len(mutualNum[1]), reverse=True)
            for data in recommendationInOrder:
                output = output + str(data[0]) + "(" +
                str(len(data[1])) + ":" + str(data[1])
                + ") "
            print(output)
            currentUser1 = user1
            currentUser2 = user2
            mutualCount.clear()
            outputStream.clear()
            mutualCount.append(mutualUser)
            outputStream.append(currentUser1)
if(-1 in mutualCount):
    if(len(outputStream)==1):
        pass
    else:
        output = "Recommend for user:" +
        str(outputStream[0]) +
        "\t"
        recommendationInOrder = sorted(outputStream[1:],
        key=lambda mutualNum: len(mutualNum[1]),
        reverse=True)
        for data in recommendationInOrder:
            output = output + str(data[0]) + "(" +
            str(len(data[1])) + ":" + str(data[1]) + ") "
        print(output)
else:
    outputStream.append([currentUser2,
    mutualCount.copy()])
    output = "Recommend for user:" +
    str(outputStream[0]) +
    "\t"
    recommendationInOrder = sorted(outputStream[1:],
    key=lambda
    mutualNum:len(mutualNum[1]),reverse=True)
    for data in recommendationInOrder:
        output = output + str(data[0]) + "(" + str(len(data[1]))
        + ":" + str(data[1]) + ") "
    print(output)
```

First S. Huasong
Born on 25/08/1999, Date Science Master's degree in reading, main in Machine Learning.

Background

By integrating the above research contents, this study efficiently completes massive relationship data analysis and achieves potential interpersonal relationship discovery under a fully distributed Hadoop environment based on Docker, using the Hadoop Streaming tool, combined with Mapper and Reducer written in Python language.