22组大作业演示

用时：下载2分钟+解压1分40秒+上传11分50秒+数据处理3分19秒+其他=19分

问题描述

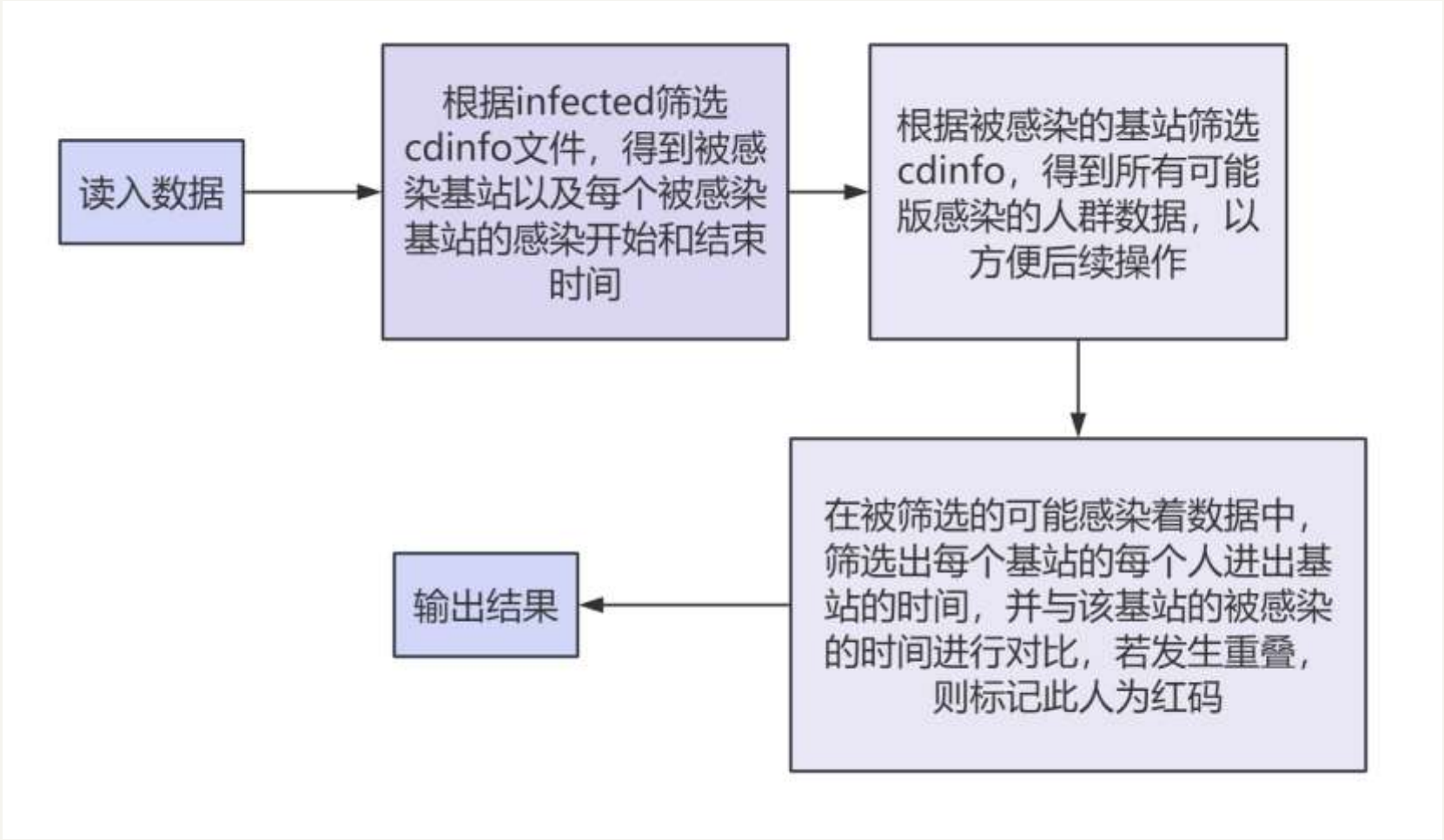
实验要求：从下发数据中找到需要标红码的人员列表，即与感染人员同时间在一个基站的手机列表，用于精确标红码，按号码升序导出到redmark+组号.txt文件中，文件格式和infected.txt相同。

本次实验我们主要利用大数据技术支持新冠疫情的控制与管理，通过分析手机漫游信息和感染者的信息，找出与感染者有接触风险的人员，并将其标注为红码，以促使密切接触者进行核算筛查。

整体思路流程

本次大作业，我们小组选择自行搭建**Spark集群**进行数据的处理，选取**On Yarn**模式运行，实际上使用了组内5台机器使用**WSL**在**校园网环境下**运行，代码采用**Scala**编写。

在接收到数据后，首先将其解压并上传至HDFS，上传完毕后调用Jar包对数据进行处理，输出redmark后下载至本地并提交。



大作业工作流程

1.实验过程

核心代码

2.执行过程记录

照片记录

3.总结分析

总结分析

1.核心代码

我们采用五台电脑，并行处理数据，最大限度发挥了计算资源，并使用 spark-submit 命令设置了相应参数。

```
--conf spark.default.parallelism=120 \    #设置 Spark 应用程序的默认并行度为 120，即每个任务默认使用的分区数。

--conf spark.executor.memory=4g \  #设置每个 Executor 的内存为 4GB。

--conf spark.executor.cores=4 \  #设置每个 Executor 的 CPU 核心数为 4。

--conf spark.cores.max=96 \  #设置集群的最大 CPU 核心数为 96。

--num-executors 5  #设置应用程序运行时使用的 Executor 数量为 5。
```

1.核心代码

1.读入数据

```
val cdinfo = spark.read.csv(args(0)).toDF("id", "timestamp", "type", "personId")
val infected = spark.read.csv(args(1)).toDF("personId")
```

2.提取感染人群的手机号，并将其**广播**到所有Spark集群的工作节点中；

```
val infectedPersonIds = infected.select("personId").as[String].collect().toSet
val infectedPersonIdsBC = spark.sparkContext.broadcast(infectedPersonIds)
```

使用广播变量的原因：
默认情况下，task执行的算子中，使用了外部的变量，每个task都会获取一份变量的副本，有多少task就会有多少副本，并且都需要从driver端序列化发送到执行端。如果task太多，或者变量太大，都会造成网络IO和内存不必要的频繁读写，加上序列化和反序列化，会严重影响程序性能。
广播变量的好处：不是每个task一份变量副本，而是变成每个节点的executor才一份副本。这样的话，就可以让变量产生的副本大大减少。

1.核心代码

3.过滤cdinfo，只保留手机号为感染人群的数据，减少数据传输，方便后续分析被感染基站的感染时间

```
// 过滤每个分区的数据，减少数据传输
val filteredCdinfo = cdinfo.filter(row => infectedPersonIdsBC.value.contains(row.getString(3)))
```

4.将上一步过滤所得数据按照基站编号分组，并提取每个基站内的感染开始时间与结束时间

```
val groupedCdinfo = filteredCdinfo.groupBy("id").agg(
  collect_list(when(col("type") === 1, col("timestamp")).cast("long")).alias("startTimes"),
  collect_list(when(col("type") === 2, col("timestamp")).cast("long")).alias("endTimes")
)
```

1.核心代码

5.**持久化上一步得到的数据，放置重复运算浪费性能——提升代码性能的关键**

```
// 持久化 groupedCdinfo 数据
groupedCdinfo.persist(StorageLevel.MEMORY_AND_DISK_SER)
```

因为 Spark 程序执行的特性，即**延迟执行和基于 Lineage 最大化的 pipeline**，当 Spark 中由于对某个 RDD 的 Action 操作触发了作业时，会基于 Lineage 从后往前推，找到该 RDD 的源头 RDD，然后从前往后计算出结果。

很明显，如果对某个 RDD 执行了多次 Transformation 和 Action 操作，每次 Action 操作出发了作业时都会重新从源头 RDD 出计算一遍来获得 RDD，再对这个 RDD 执行相应的操作。当 RDD 本身计算特别复杂和耗时时，这种方式性能是非常差的，此时必须考虑对计算结果的数据进行持久化。

数据持久化（或称为缓存）就是将计算出来的 RDD 根据配置的持久化级别，保存在内存或磁盘中，以后每次对该 RDD 进行算子操作时，都会直接从内存或者磁盘中提取持久化的 RDD 数据，然后执行算子操作，而不会从源头处重新计算一遍该 RDD。

1.核心代码

6.将分组后的数据收集起来转换为一个Map，并广播到所有工作节点

```
// 显式指定 collect 后的类型
val baseStationInfectedTimes: Map[String, (Seq[Long], Seq[Long])] = groupedCdinfo.collect().map { row =>
  val baseId = row.getString(0)
  val startTimes = row.getAs[Seq[Long]]("startTimes")
  val endTimes = row.getAs[Seq[Long]]("endTimes")
  (baseId, (startTimes, endTimes))
}.toMap

val baseStationInfectedTimesBC = spark.sparkContext.broadcast(baseStationInfectedTimes)
```

7.在cdinfo中筛选曾进过被感染基站的人群数据，并**持久化该数据**，以方便后续操作

```
// 仅在每个分区内过滤数据
val potentiallyInfectedPeople = cdinfo.filter { row =>
  val baseId = row.getString(0)
  baseStationInfectedTimesBC.value.contains(baseId)
}

potentiallyInfectedPeople.persist(StorageLevel.MEMORY_AND_DISK_SER)
```

8.将上一步所得数据按照基站编号与手机号分组，提取出基站内某人的进出时间

```
val groupedPotentiallyInfected = potentiallyInfectedPeople.groupBy("id", "personId").agg(
  collect_list(col("timestamp").cast("long")).alias("times")
)
```

1.核心代码

9.定义、注册UDF函数用于标记红码人群——核心算法

```
def isInfected(baseId: String, times: Seq[Long]): Boolean = {
  val sortedTimes = times.sorted
  if (sortedTimes.nonEmpty && baseStationInfectedTimesBC.value.contains(baseId)) {
    val (startTimes, endTimes) = baseStationInfectedTimesBC.value(baseId)
    for (i <- startTimes.indices) {
      val startTime = startTimes(i)
      val endTime = endTimes(i)
      for (j <- sortedTimes.indices by 2) {
        if (j + 1 < sortedTimes.length) {
          val entryTime = sortedTimes(j)
          val exitTime = sortedTimes(j + 1)
          if ((startTime <= entryTime && endTime >= entryTime) ||
            (startTime <= exitTime && endTime >= exitTime) ||
            (startTime >= entryTime && endTime <= exitTime)) {
            return true
          }
        }
      }
    }
  }
  false
}

// 注册UDF函数
val isInfectedUDF = udf(isInfected _)
val finalInfected = groupedPotentiallyInfected.filter(isInfectedUDF(col("id"), col("times")))
finalInfected.show()
```

该UDF函数具体判断方式为：**首先提取某人所在的感染基站编号，找到该基站的感染时间数据，并与此人进出感染基站的时间进行比对**，如果发生重叠(进、出时刻在感染之间；感染在进出时间段之间)，则标记此人为红码，否则不标记。

我们的算法可以完美应对同一个人进出多次、不完全包含于感染时段的特殊情况。

1.核心代码

10.按手机号升序排列结果并输出

```
val finalTask = finalInfected.select("personId").distinct().sort("personId")
finalTask.coalesce(1).write.text(args(3))
```

五台电脑执行实验：

2.执行过程记录

2.执行过程记录

下载数据：

2.执行过程记录

解压数据：

2.执行过程记录

上传数据至HDFS：

2.执行过程记录

开始处理数据：

2.执行过程记录

数据处理完毕：

2.执行过程记录

下载结果：

3.总结分析

在本次大作业中，本小组自行搭建了spark集群，使用组内五名成员的电脑作为集群节点进行数据处理。具体的任务实现方式中，使用scala编程语言编写了分布式并行的任务执行程序，实现了基于基站数据与感染名单生成红码人员文件的作业任务。

在整个大作业完成过程中，本小组成员查阅了大量spark集群与scala编程的相关资料，最大限度优化了执行算法，达到减少计算资源的浪费、I\O操作的频率的目的，极大地提升了集群运算速度。此外，在最终正式执行任务前，本小组多次合作运行代码进行模拟测试，并使用python代码本地处理模拟数据，进行结果对照，据此对代码进行多次调试，确保了代码的稳定性与正确性。

通过本次大作业，小组成员丰富了spark知识储备，提升了自身的实践能力，增强了团队合作意识与攻坚克难精神，让我们对未来再次面对难题更具信心。

最后，本小组最终完成任务，有赖于所有小组成员齐心协力、共同奋战的每一个夜晚，感谢所有成员的团结与努力。

感谢老师的指导