

大数据计算基础

第八组大作业

目录

01 思路介绍

02 代码解析

03 结果展示



01

思路介绍



01 思路介绍

基于 Spark SQL的数据处理和计算功能

- 利用了 Spark SQL 提供的分组、聚合、连接等操作，实现了高效且直观的数据处理和计算

利用Spark和HDFS的分布式计算能力

- 处理大规模的数据集，并在大规模数据上实现高性能的感染传播模拟

基于人员在基站内的进出记录，通过分析每个手机号码在每个基站内的停留时间段，来推断人员之间的接触情况，从而模拟病毒在人群中的传播过程

读取并处理原始
数据

计算出每个用户
在每个基站停留
的时间段

读取感染者列表，
筛选出与感染者
有过接触的用户



02

代码解析



1.导入所需的包和类

使用了scala的特质混入(trait mixin)系统, 自动给主类生成独一无二的日志器

```
import org.apache.logging.log4j.scala.Logging
import org.apache.spark.sql._
import org.apache.spark.sql.functions.{col, explode}
import org.apache.spark.sql.types._
import sun.misc.Signal

import java.util.concurrent.Semaphore

case class CPEL(cell: Long, phone: Long, folding: List[(Long, Long)])
case class CellLogRecord(cell: Long, timestamp: Long, enterOrLeave: Short, phone: Long)
```

导入了所需的包, 包括日志库Log4j、Apache Spark的SQL相关类和函数、信号处理相关类等, 并且自定义类CPEL、CellLogRecord

自定义类说明

```
case class CellLogRecord(cell: Long, timestamp: Long, enterOrLeave: Short, phone: Long)
```

数据结构与基站漫游信息数据完全吻合，用于接收原始文件数据

```
case class CPEL(cell: Long, phone: Long, folding: List[(Long, Long)])
```

一个CPEL实例 Cell Phone Enter Leave

cell	phone	folding
100	10000	[(100000,200000),(300000,400000),...,(800000,900000)]

经过处理（详见4）得到，用于记录某一用户在某一基站停留的时间段

2. 读取数据集

```
val ds = spark.read.schema(StructType(Seq(  
  StructField("cell", LongType, nullable = false),  
  StructField("timestamp", LongType, nullable = false),  
  StructField("enterOrLeave", ShortType, nullable = false),  
  StructField("phone", LongType, nullable = false))  
)).csv(cdinfoFile).as[CellLogRecord]
```

这段代码读取了一个CSV文件，并将其解析成一个Spark Dataset。这里使用了StructType来定义数据集中每一列的数据类型，然后将其转换成一个CellLogRecord类型的Dataset。

3.数据处理

foldLeft方法

```
val spans = ds.groupByKey(x => (x.cell, x.phone)).mapGroups(
  (group, infos) => {
    CPEL(group._1, group._2, infos.toList.sortBy(_.timestamp).foldLeft((Nil: List[(Long, Long)], None: Option[Long], 0)) {
      case ((mergedList, None, 0), e) if e.enterOrLeave == 1 =>
        (mergedList, Some(e.timestamp), 1)
      case ((mergedList, Some(lastEnter), 1), e) if e.enterOrLeave == 2 =>
        ((lastEnter, e.timestamp) :: mergedList, None, 0)
      case ((mergedList, maybeEnter, stack), e) if e.enterOrLeave == 1 =>
        (mergedList, maybeEnter, stack + 1)
      case ((mergedList, maybeEnter, stack), e) if e.enterOrLeave == 2 =>
        (mergedList, maybeEnter, stack - 1)
    })._1)
  }
).cache()
```

这段代码对上一步中读取的数据集进行了一些处理。使用groupByKey方法以(cell, phone)为键进行分组使用mapGroups方法对每个分组进行处理。

具体地，将每个分组中的数据按照时间戳排序，然后将每个电话进入和离开的时间记录成一个元组，最后将这些元组记录成一个List，构成一个CPEL类型的对象。这里使用了foldLeft方法对数据进行了累积处理，并使用了case语句进行匹配。

处理后，spans中的元素是CPEL实例



关于foldLeft()

foldLeft()方法包含3个要素：

1. 初始值
2. 操作
3. 返回值


具体地，foldLeft()会设定一个初始状态，然后从左往右遍历列表中的每个元素，对每个元素进行匹配，根据匹配结果对状态进行相应的转换。遍历完毕后，返回已经被转换多次的状态。

关于foldLeft()

在本例中

处理对象：在同一基站和手机号下的漫游信息元组的列表

返回值：从漫游信息中整合出的时间段列表



```
[(x1.cell, x1.timestamp,
x1.enterOrLeave, x1.phone),
(x2.cell, x2.timestamp,
x2.enterOrLeave, x2.phone),
.....,
(xn.cell, xn.timestamp,
xn.enterOrLeave, xn.phone)]
```

以上所有的cell, phone相同

初始值：一个元组

(Nil: List[(Long, Long)], None: Option[Long], 0)

各元素意义：

1. 列表：存放时间段元组(进入时间，离开时间)
2. Option：存放进入时间（未匹配到离开时），否则为None
3. 栈深度：初始时置为0，当有进入事件时+1，有离开事件时-1

关于foldLeft()

操作

操作由4个case完成，4个case意义对应如下（实际上只有前2个起作用）：

正常情况下（不会有多于1个进入事件）：

1. 当栈深度为0时，若遇到进入事件，将此事件时间放入Option，并将栈深度置为1
2. 当栈深度为1时，若遇到离开事件，将此事件（离开）时间和Option中存放的（进入）时间组成一个时间段，并加入列表，将栈深度置为0

只有这两种情况

实际上只由1，2即可处理完所有情况，若出现多于1个进入事件（可能由手机号重合导致），可以不采取措施。

如此，则保证列表中存放若干个匹配好的时间段

最后，列表中存放的就是该用户在该基站下停留的所有时间段

4. 输出日志信息

```
val sortedSpan = spans.select(  
    $"cell".as("spanCell"),  
    $"phone",  
    explode($"folding").as("folding")  
).sort($"spanCell").cache()  
logger.error(s"对 cdinfo.txt 预处理完成, 共 ${sortedSpan.count()} 条记录")  
  
for (i <- 5 to 0 by -1) {  
    Thread.sleep(1000)  
    logger.error(s"$i 秒后开始读取 infected.txt")  
}
```

输出日志信息，方便查看程序运行时中间结果和运行阶段

5. 读取另一个数据集

```
val infected = spark.read.schema(StructType(Seq(  
  StructField("infectedPhone", LongType, nullable = false)  
)))  
).csv(infectedFile)
```

这段代码读取了另一个CSV文件，并将其解析成一个Spark DataFrame。这里只有一列数据，即感染电话号码。

6.数据处理

```
val infectedSpans = spans.join(infected, $"phone" ===  
    $"infectedPhone", "cross").as[CPEL]
```

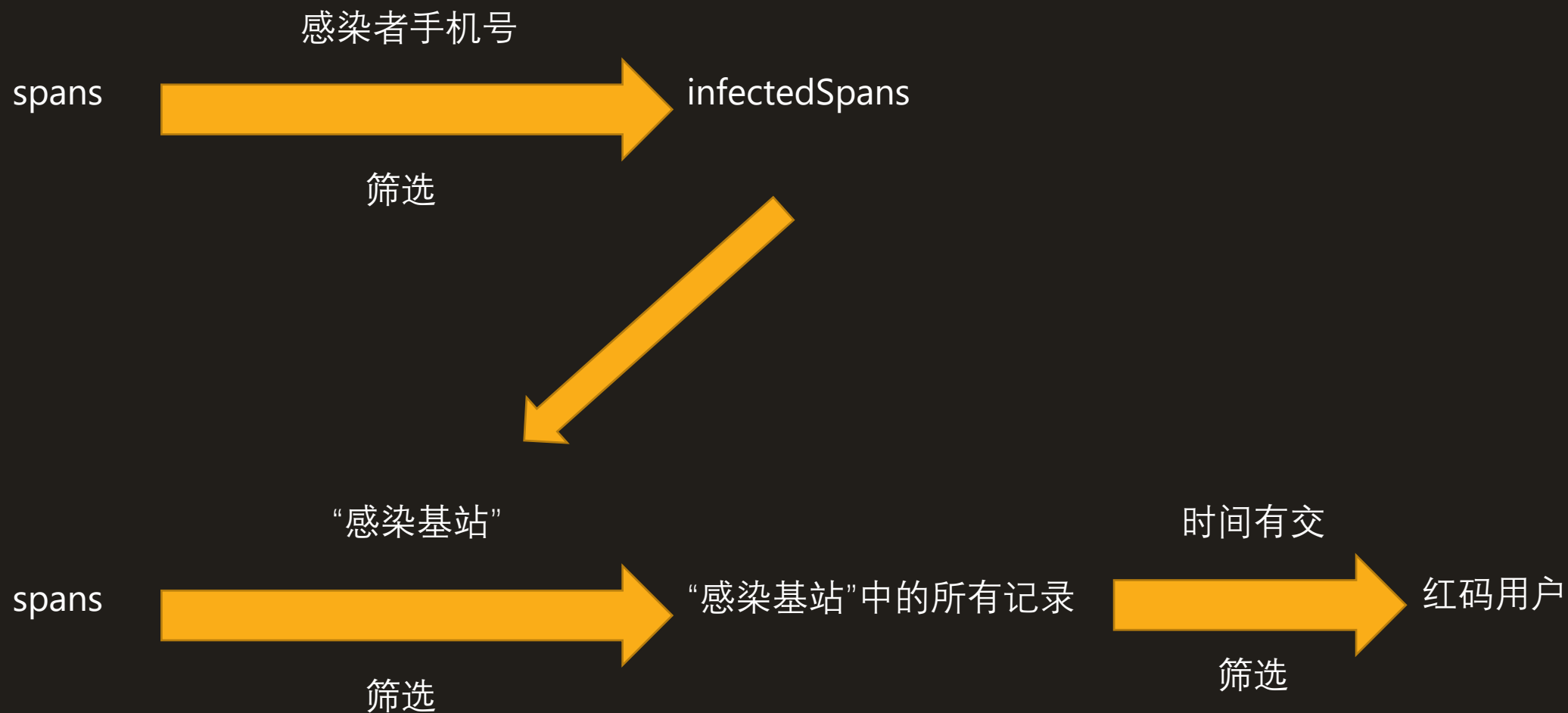
这段代码将上一步中处理得到的CPEL类型的数据集与感染电话号码进行连接，得到所有感染电话号码的位置跨度信息，并将其转换成CPEL类型的对象。

7.数据处理

```
val result = infectedSpans.select(
    $"cell",
    explode($"folding").as("infectedFolding")
).join(
    sortedSpan,
    $"cell" === $"spanCell", "cross"
).filter(
    "(folding._2 >= infectedFolding._1) AND (folding._1 <= infectedFolding._2)"
).select($"phone").distinct().sort($"phone").cache()
```

这段代码对上一步中得到的数据集进行了进一步处理，将感染电话号码的位置跨度信息与所有电话号码的位置跨度信息进行连接，得到所有与感染电话号码有交集的电话号码，并将其去重后缓存。这里使用了explode方法对嵌套的List进行展开，并使用filter方法过滤出有交集的电话号码。

6、7中的关系示意



8.输出结果

```
logger.error(s"共有 ${result.count()} 人需要标记为红码")  
result.show()  
  
result.write.text(resultFile)
```

这段代码输出红码电话号码数量，并输出这些电话号码，
且将电话号码输出到文件

9. 信号处理

```
val semaphore = new Semaphore(0)

Signal.handle(new Signal("INT"), _ => {
  logger.info("收到 Ctrl+C 信号, 退出")
  semaphore.release()
})

semaphore.acquire()
```

只在收到 Ctrl+C 信号后才让主线程退出，方便在输出结果后继续查看spark网页端浏览状态



03

结果展示

