

一、实验过程

1. MapReduce 热门商品统计

热门商品和商家的统计与 Word Count 逻辑非常类似，此处直接使用 Word Count 作业中定义的 IntSumReducer、IntWritableDecreaseComparator、OrderRecordWriter 和 OrderOutputFormat 类来实现 Reduce 和格式化输出，并重新定义 ItemFollowedMapper 和 MerchantFollowedMapper 类来实现对符合要求的商品与商家的过滤。

在 ItemFollowedMapper 类的 map()函数中，对 time_stamp 和 action_type 做判断，将符合要求的 item_id 发出。

```
1. public void map(Object key, Text value, Context context) throws IOException,
   InterruptedException {
2.     String line = value.toString();
3.     String[] tokens = line.split(",");
4.     if (tokens[0].equals("user_id")) return;
5.     if (!tokens[5].equals("1111")) return;
6.     if (tokens[6].equals("0")) return;
7.     keyIntWritable.set(Integer.parseInt(tokens[1]));
8.     context.write(keyIntWritable, valueIntWritable);
9. }
```

在 MerchantFollowedMapper 类的 setup()函数中，读取 info 文件，将年龄小于 30 的用户的 user_id 加入集合 ageU30Set。

```
1. public void setup(Context context) throws IOException, InterruptedException
   {
2.     conf = context.getConfiguration();
3.     BufferedReader fis = new BufferedReader(new FileReader("user_info_format
   1.csv"));
4.     String line = null;
5.     while ((line = fis.readLine()) != null) {
6.         String[] tokens = line.split(",");
7.         if (tokens.length < 2) continue;
8.         if (tokens[0].equals("user_id")) continue;
9.         if (!(tokens[1].equals("1") || tokens[1].equals("2") || tokens[1].eq
   uals("3"))) continue;
10.        ageU30Set.add(Integer.parseInt(tokens[0]));
11.    }
12. }
```

在 map()函数中，对 time_stamp 和 action_type 做判断，并且判断 user_id 是否在 ageU30Set 之中，将符合要求的 merchant_id 发出。

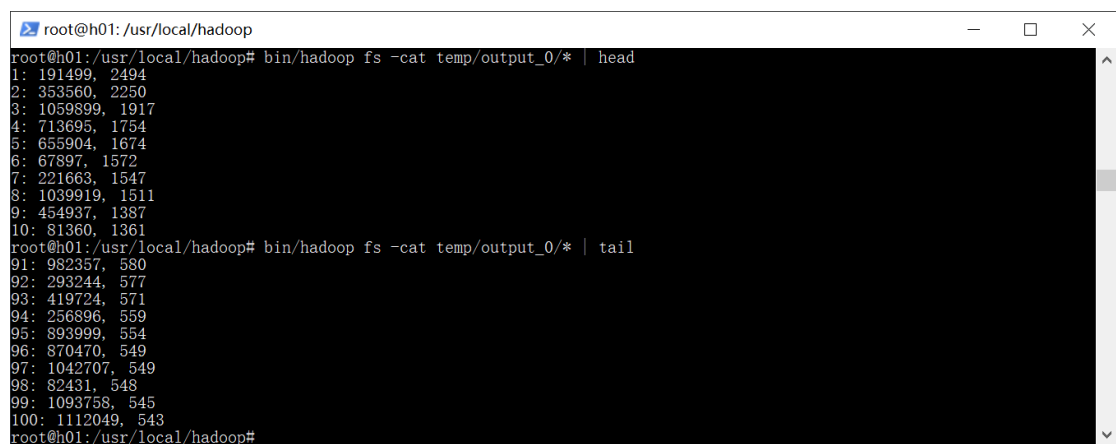
```

1. public void map(Object key, Text value, Context context) throws IOException,
   InterruptedException {
2.     String line = value.toString();
3.     String[] tokens = line.split(",");
4.     if (tokens[0].equals("user_id")) return;
5.     if (!tokens[5].equals("1111")) return;
6.     if (tokens[6].equals("0")) return;
7.     if (!ageU30Set.contains(Integer.parseInt(tokens[0]))) return;
8.     keyIntWritable.set(Integer.parseInt(tokens[3]));
9.     context.write(keyIntWritable, valueIntWritable);
10. }

```

在 main()函数中依次分别执行以 ItemFollowedMapper 和 MerchantFollowedMapper 类为 Mapper、IntSumReducer 类为 Reducer 的 job，再对两者的中间结果分别执行以 InverseMapper 为 Mapper、IntWritableDecreaseComparator 为 SortComparator 的 sortJob。

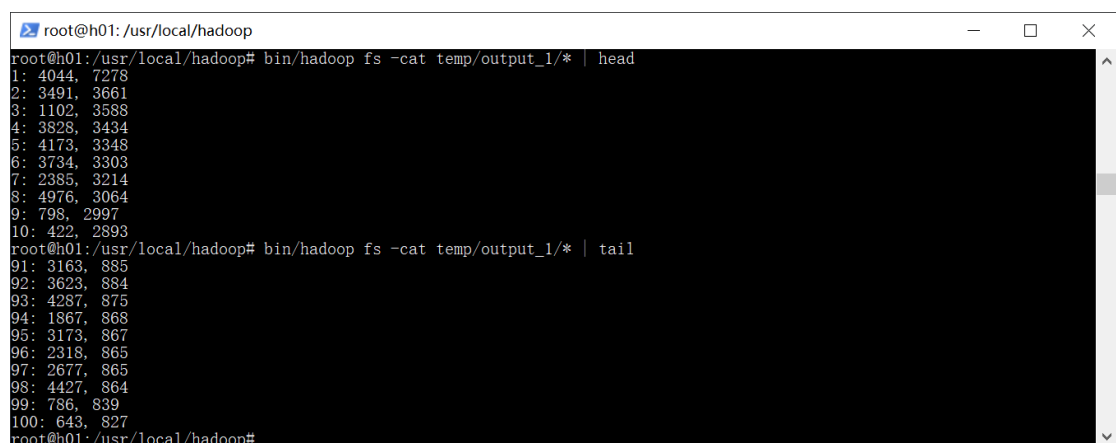
MapReduce 程序即编写完成。在本地集群中运行结果如下。



```

root@h01: /usr/local/hadoop
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_0/* | head
1: 191499, 2494
2: 353560, 2250
3: 1059899, 1917
4: 713695, 1754
5: 655904, 1674
6: 67897, 1572
7: 221663, 1547
8: 1039919, 1511
9: 454937, 1387
10: 81360, 1361
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_0/* | tail
91: 982357, 580
92: 293244, 577
93: 419724, 571
94: 256896, 559
95: 893999, 554
96: 870470, 549
97: 1042707, 549
98: 82431, 548
99: 1093758, 545
100: 1112049, 543
root@h01: /usr/local/hadoop#

```



```

root@h01: /usr/local/hadoop
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_1/* | head
1: 4044, 7278
2: 3491, 3661
3: 1102, 3588
4: 3828, 3434
5: 4173, 3348
6: 3734, 3303
7: 2385, 3214
8: 4976, 3064
9: 798, 2997
10: 422, 2893
root@h01: /usr/local/hadoop# bin/hadoop fs -cat temp/output_1/* | tail
91: 3163, 885
92: 3623, 884
93: 4287, 875
94: 1867, 868
95: 3173, 867
96: 2318, 865
97: 2677, 865
98: 4427, 864
99: 786, 839
100: 643, 827
root@h01: /usr/local/hadoop#

```

2. Spark 热门商品统计

采用 IntelliJ IDEA、Scala 语言和 sbt 构建工具作为开发环境，采用本地 Docker 集群中的 Spark 2.4.0 作为运行环境。由于 Spark 2.4.0 依赖于 Hadoop 2.7.0，而先前实验中

所用的 Hadoop 版本为 3.2.1，两者无法兼容，因此需要重新配置安装集群环境，步骤较为繁琐，不在此赘述。

在 IDEA 中，安装 Scala 插件后即可创建 Scala 工程。在 build.sbt 文件中增加 Spark 相关依赖项即可在源代码中导入、调用 Spark 相关库。

```
1. libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
```

新建 MostPopular 类，并在其中定义 main() 函数，作为程序主入口。首先读取 info 文件为 RDD，通过 map() 分割数据行，通过 filter() 过滤出年龄小于 30 的用户，通过 map() 提取 user_id 列，最后通过 collect().toSet 将 RDD 转换为集合，赋值给 userU30。

```
1. val infoFile = "file:///usr/local/spark-2.4.0/input/user_info_format1.csv"
2. val conf = new SparkConf().setAppName("Most Popular Item");
3. val sc = new SparkContext(conf)
4. val info = sc.textFile(infoFile).cache()
5. val userU30 = info
6.   .map(line => line.split(","))
7.   .filter(line => line.size >= 2 && !line(0).equals("user_id") && (line(1).equals("1") || line(1).equals("2") || line(1).equals("3")))
8.   .map(line => line(0))
9.   .collect()
10.  .toSet
```

而后读取 input 文件为 RDD，通过 map() 分割数据行，通过 filter() 过滤出在双十一有加购物车、购买或收藏操作的用户，通过 map() 提取 item_id 列，通过 reduceByKey() 以 item_id 为分组计数，通过 sortBy() 排序，最后通过 take() 提取结果的前 100 项，赋值给 itemTop100。

```
1. val inputFile = "file:///usr/local/spark-2.4.0/input/user_log_format1.csv"
2. val input = sc.textFile(inputFile).cache()
3. val itemTop100 = input
4.   .map(line => line.split(","))
5.   .filter(line => !line(0).equals("user_id") && line(5).equals("1111") && !line(6).equals("0"))
6.   .map(line => (line(1), 1))
7.   .reduceByKey { case (x, y) => x + y }
8.   .sortBy(_._2, false)
9.   .take(100)
```

以类似方式处理得到 merchantTop100，仅在 filter() 函数的参数中增加对 user_id 是否在 userU30 集合中的判断。

```
1. val merchantTop100 = input
2.   .map(line => line.split(","))
```

```

3.    .filter(line => !line(0).equals("user_id") && line(5).equals("1111") && !line(6).equals("0") && userU30.contains(line(0)))
4.    .map(line => (line(3), 1))
5.    .reduceByKey { case (x, y) => x + y }
6.    .sortBy(_._2, false)
7.    .take(100)

```

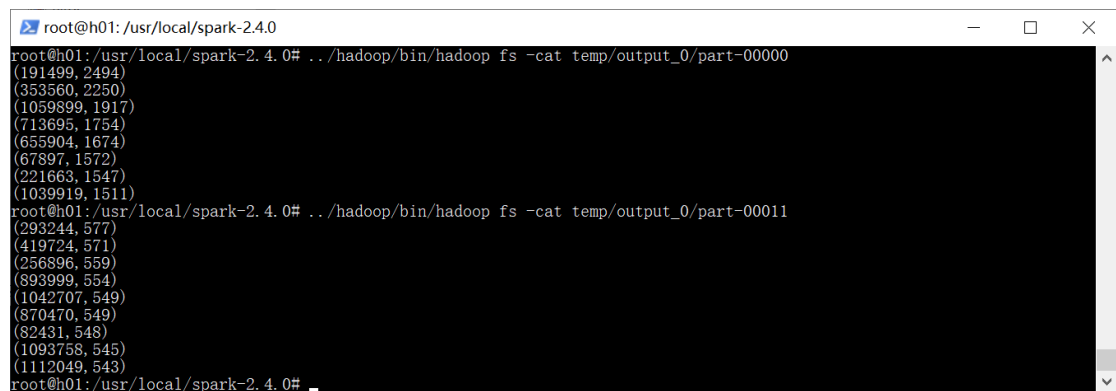
需要注意，经过 take()函数处理后的 RDD 不再是 RDD，而是 Array，需要通过 sc.parallelize()将其重新转化为 RDD 后通过 saveAsTextFile()保存到文件。

```

1. sc.parallelize(itemTop100).saveAsTextFile("temp/output_0")
2. sc.parallelize(merchantTop100).saveAsTextFile("temp/output_1")

```

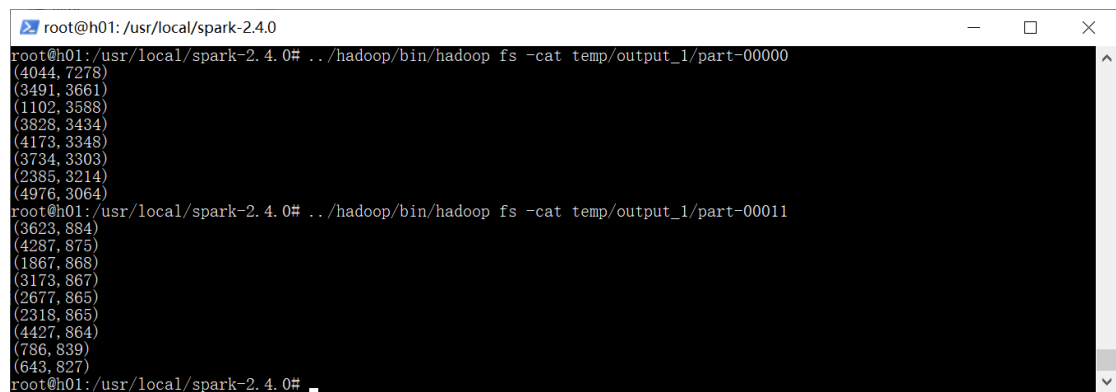
Spark 程序即编写完成。在本地集群中运行结果如下。



```

root@h01: /usr/local/spark-2.4.0
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_0/part-00000
(191499, 2494)
(353560, 2250)
(1059899, 1917)
(713695, 1754)
(655904, 1674)
(67897, 1572)
(221663, 1547)
(1039919, 1511)
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_0/part-00011
(293244, 577)
(419724, 571)
(256896, 559)
(893999, 554)
(1042707, 549)
(870470, 549)
(82431, 548)
(1093758, 545)
(1112049, 543)
root@h01: /usr/local/spark-2.4.0#

```



```

root@h01: /usr/local/spark-2.4.0
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_1/part-00000
(4044, 7278)
(3491, 3661)
(1102, 3588)
(3828, 3434)
(4173, 3348)
(3734, 3303)
(2385, 3214)
(4976, 3064)
root@h01: /usr/local/spark-2.4.0# ./hadoop/bin/hadoop fs -cat temp/output_1/part-00011
(3623, 884)
(4287, 875)
(1867, 868)
(3173, 867)
(2677, 865)
(2318, 865)
(4427, 864)
(786, 839)
(643, 827)
root@h01: /usr/local/spark-2.4.0#

```

3. Spark 用户画像统计

新建 BuyerInfo 类，并在其中定义 main()函数，作为程序主入口。首先读取 input 和 info 文件为 RDD。

```

1. val inputFile = "file:///usr/local/spark-2.4.0/input/user_log_format1.csv"
2. val infoFile = "file:///usr/local/spark-2.4.0/input/user_info_format1.csv"
3. val conf = new SparkConf().setAppName("User Info")
4. val sc = new SparkContext(conf)
5. val input = sc.textFile(inputFile).cache()

```

```
6. val info = sc.textFile(infoFile).cache()
```

对 input 通过 map()分割数据行, 通过 filter()过滤出在双十一有购买操作的用户, 通过 map()提取 user_id 列, 最后通过 collect().toSet 将 RDD 转换为集合并去重, 赋值给 buyer。

```
1. val buyer = input
2.   .map(line => line.split(","))
3.   .filter(line => line(5).equals("1111") && line(6).equals("2"))
4.   .map(line => line(0))
5.   .collect()
6.   .toSet
```

对 info 通过 map()分割数据行, 通过 filter()过滤出在双十一有购买操作的用户, 通过 map()将缺失值设为“未知”, 通过 map()提取 gender 列, 通过 reduceByKey()以 gender 为分组计数, 最后通过 sortBy()排序, 赋值给 gender。

```
1. val gender = info
2.   .map(line => line.split(","))
3.   .filter(line => buyer.contains(line(0)))
4.   .map(line => if (line.size < 3) Array[String](line(0), "0", "2") else line
   )
5.   .map(line => (line(2), 1))
6.   .reduceByKey { case (x, y) => x + y }
7.   .sortBy(x => x._2, false)
```

对 gender 通过 map()提取 count 列, 通过 reduce()求和得到在双十一有购买操作的用户总数, 赋值给 buyerSum。

```
1. val buyerSum = gender
2.   .map(line => line._2)
3.   .reduce(_ + _)
```

对 gender 通过 map()计算分组数量和总数之比, 得到比例, 赋值给 genderProp。

```
1. val genderProp = gender
2.   .map(line => (line._1, line._2.toDouble/buyerSum.toDouble))
```

以类似方式对 info 处理得到年龄段比例, 赋值给 ageProp。

```
1. val age = info
2.   .map(line => line.split(","))
3.   .filter(line => buyer.contains(line(0)))
4.   .map(line => if (line.size < 2 || (line.size >= 2 && line(1).equals("")))
   Array[String](line(0), "0") else line)
```

```

5.  .map(line => (line(1), 1))
6.  .reduceByKey { case (x, y) => x + y }
7.  .sortBy(x => x._2, false)
8.  val ageProp = age
9.  .map(line => (line._1, line._2.toDouble/buyerSum.toDouble))

```

最后保存到文件。

```

1. genderProp.saveAsTextFile("temp/output_0")
2. ageProp.saveAsTextFile("temp/output_1")

```

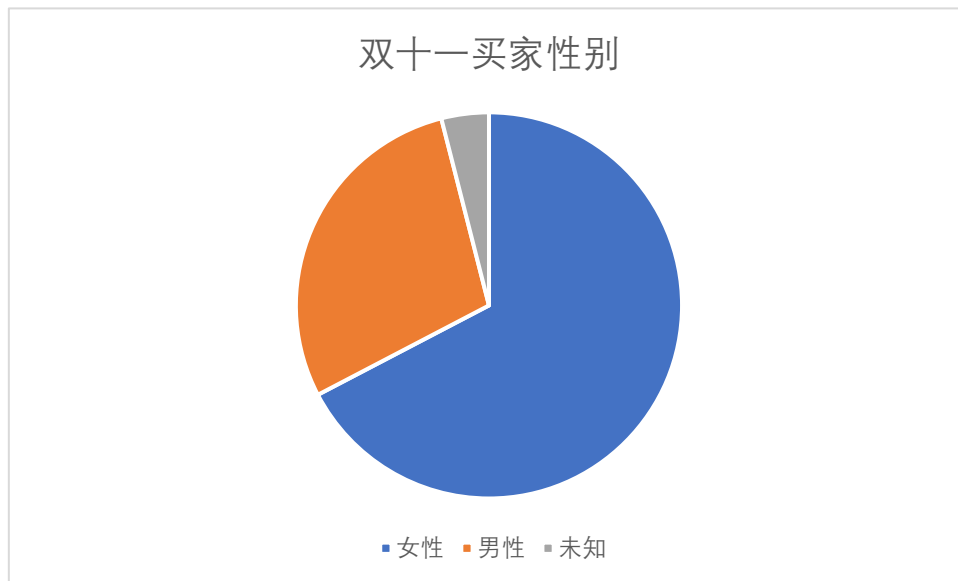
Spark 程序即编写完成。在本地集群中运行结果如下。

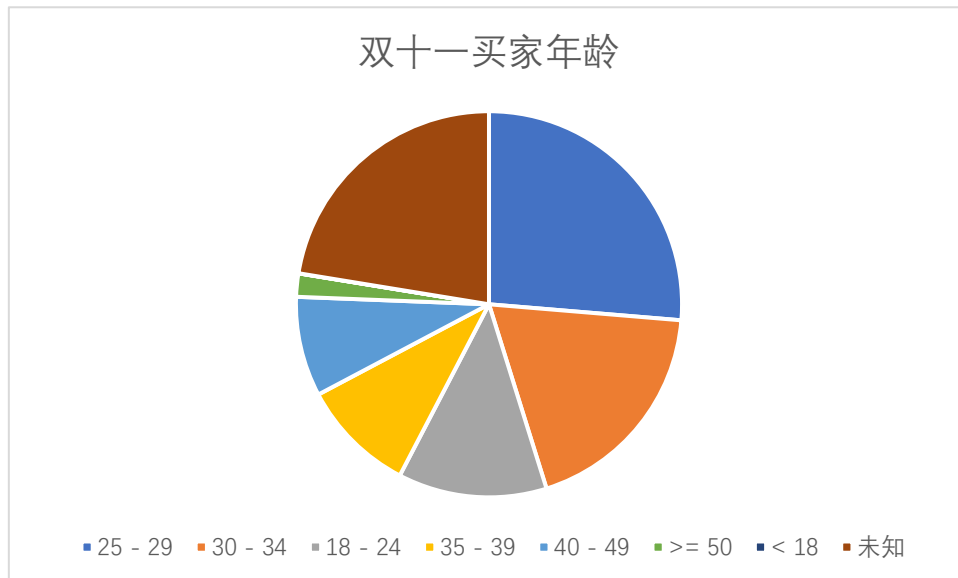
```

root@h01: /usr/local/spark-2.4.0
root@h01:/usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat temp/output_0/*
(0.0.6734045312021124)
(1.0.2868425395478228)
(2.0.03975292925006483)
root@h01:/usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat temp/output_1/*
(3.0.26322936558455334)
(0.0.22427564419925974)
(4.0.18858240799679374)
(2.0.12464577881509772)
(5.0.09613362566895349)
(6.0.08360798736355707)
(7.0.016483956904071482)
(8.0.0029846523799420045)
(1.5.6581087771412404E-5)
root@h01:/usr/local/spark-2.4.0#

```

可视化结果如下。





4. Spark SQL 用户画像统计

在 build.sbt 文件中增加 Spark SQL 所需依赖项。

```
1. libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.4.0"
```

定义 SparkSession。

```
1. val spark = SparkSession
2.   .builder()
3.   .appName("Spark SQL Buyer Info")
4.   .config("spark.some.config.option", "some-value")
5.   .getOrCreate()
6. import spark.implicits._
```

以 csv 格式读取 input 和 info 文件为 DataFrame。

```
1. val inputDF = spark
2.   .read.format("csv")
3.   .option("inferSchema", "true")
4.   .option("header", "true")
5.   .load("file:///usr/local/spark-2.4.0/input/user_log_format1.csv")
6. val infoDF = spark
7.   .read.format("csv")
8.   .option("inferSchema", "true")
9.   .option("header", "true")
10.  .load("file:///usr/local/spark-2.4.0/input/user_info_format1.csv")
```

对 inputDF 通过 filter() 过滤出在双十一有购买操作的用户, 通过 select() 提取 user_id 列, 通过 distinct() 去重, 通过 join() 连接 infoDF 得到带有上述用户 gender 和 age_range

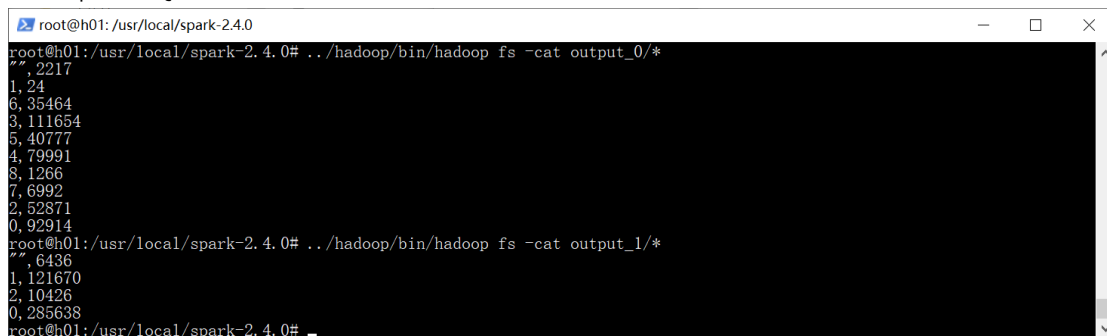
的 DataFrame，赋值给 buyer1111。

```
1. val buyer1111 = inputDF
2.   .filter($"time_stamp" === 1111 && $"action_type" === 2)
3.   .select("user_id")
4.   .distinct()
5.   .join(infoDF, "user_id")
```

对 buyer1111 通过 groupBy().count()按列分组计数，最后通过 write()保存到文件。

```
1. buyer1111
2.   .groupBy("age_range")
3.   .count()
4.   .write
5.   .format("csv")
6.   .mode("overwrite")
7.   .save("output_0")
8. buyer1111
9.   .groupBy("gender")
10.  .count()
11.  .write
12.  .format("csv")
13.  .mode("overwrite")
14.  .save("output_1")
```

Spark SQL 程序即编写完成。在本地集群中运行结果如下。



```
root@h01: /usr/local/spark-2.4.0
root@h01: /usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat output_0/*
", 2217
1, 24
6, 35464
3, 111654
5, 40777
4, 79991
8, 1266
7, 6992
2, 52871
0, 92914
root@h01: /usr/local/spark-2.4.0# ../hadoop/bin/hadoop fs -cat output_1/*
", 6436
1, 121670
2, 10426
0, 285638
root@h01: /usr/local/spark-2.4.0#
```

此处得到的结果为不同性别、年龄段买家的计数，简单处理即可得到对应的比例，结果与 Spark 程序结果相同，不再展示可视化结果。

本地集群运行时，发现尽管使用 DataFrame 的 Spark SQL 代码更加简洁，但运行时间远远长于使用 RDD 的 Spark 程序。猜测可能是因为 Spark SQL 中 join 操作的复杂度较高。

二、问题总结、解决方案及其他思考

实验过程中的所有问题及其解决方案已在实验过程中叙述，此处不再重复。

本次实验首先需要安装配置 Spark 环境，有了 Hadoop 安装配置的经验后，该项任务不算困难，但仍会花去很多时间。

由于 Spark 推荐使用 Scala 语言编程，且听说 Scala 语言程序较为简洁，本次实验中我采用了 Scala 语言。Scala 语言需要一定时间上手，但在习惯其匿名表达式的用法、且理解 RDD 或 DataFrame 的转化过程后，可发现 Scala 语言确实非常高效，编写程序的过程类似于构造 Pipeline 的过程，非常流畅。