尚硅谷大数据项目之电商分析系统

(作者：章鹏)

官网：[www.atguigu.com](http://www.atguigu.com/) 版本：V1.0

# 第一章 项目框架

## 项目概述

###### 本项目来源于企业级电商网站的大数据统计分析平台， 该平台以 Spark 框架为核心， 对电商网站的日志进行离线和实时分析。

该大数据分析平台对电商网站的各种用户行为（ 访问行为、购物行为、广告点击行为等）进行分析，根据平台统计出来的数据， 辅助公司中的 PM（产品经理）、数据分析师以及管理人员分析现有产品的情况， 并根据用户行为分析结果持续改进产品的设计，以及调整公司的战略和业务。最终达到用大数据技术来帮助提升公司的业绩、营业额以及市场占有率的目标。

本项目使用了 Spark 技术生态栈中最常用的三个技术框架， Spark Core、Spark

SQL 和 Spark Streaming， 进行离线计算和实时计算业务模块的开发。实现了包括用户访问 session 分析、页面单跳转化率统计、热门商品离线统计、广告流量实时统计

4 个业务模块。通过合理的将实际业务模块进行技术整合与改造， 该项目几乎完全涵盖了 Spark Core、Spark SQL 和 Spark Streaming 这三个技术框架中大部分的功能点、知识点，学员对于 Spark 技术框架的理解将会在本项目中得到很大的提高。

图 1-1 Spark 计数框架

* 1. **项目框架**
     1. **项目整体框架**

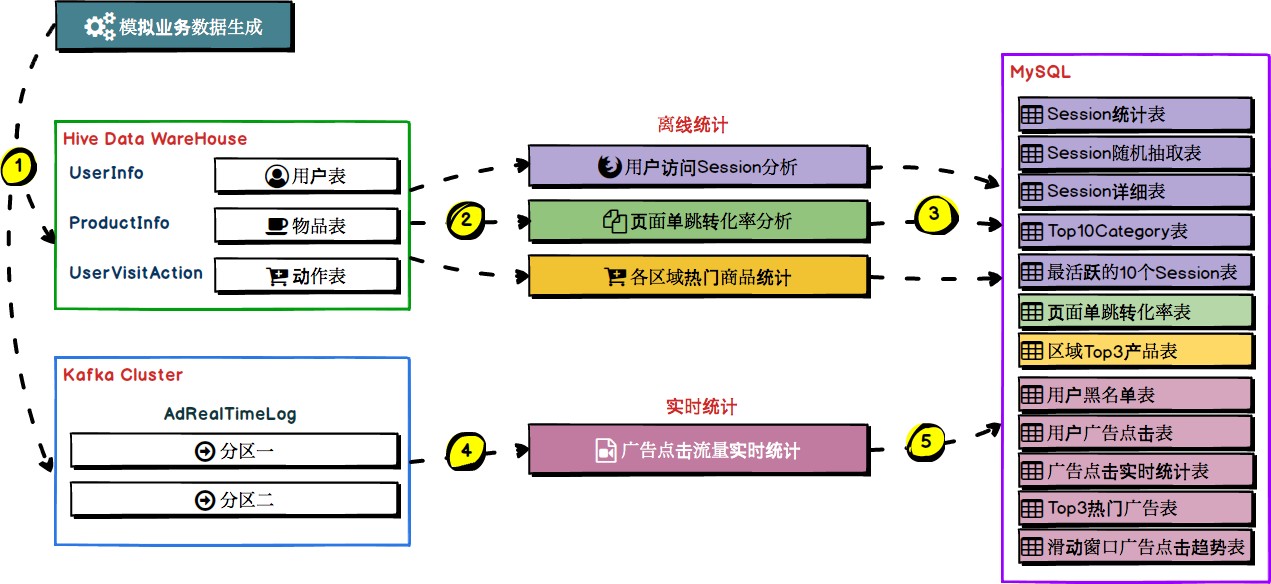


图 1-2 项目整体框架

本项目分为离线分析系统与实时分析系统两大模块。

在离线分析系统中，我们将模拟业务数据写入 Hive 表中，离线分析系统从 Hive 中获取数据，并根据实际需求（用户访问 Session 分析、页面单跳转化率分析、各区域热门商品统计） 对数据进行处理，最终将分析完毕的统计数据存储到 MySQL 的对应表格中。

在实时分析系统中，我们将模拟业务数据写入 Kafka 集群中， 实时分析系统从Kafka broker 中获取数据， 通过 Spark Streaming 的流式处理对广告点击流量进行实时分析，最终将统计结果存储到 MySQL 的对应表格中。

* + 1. **离线日志采集宏观流程**

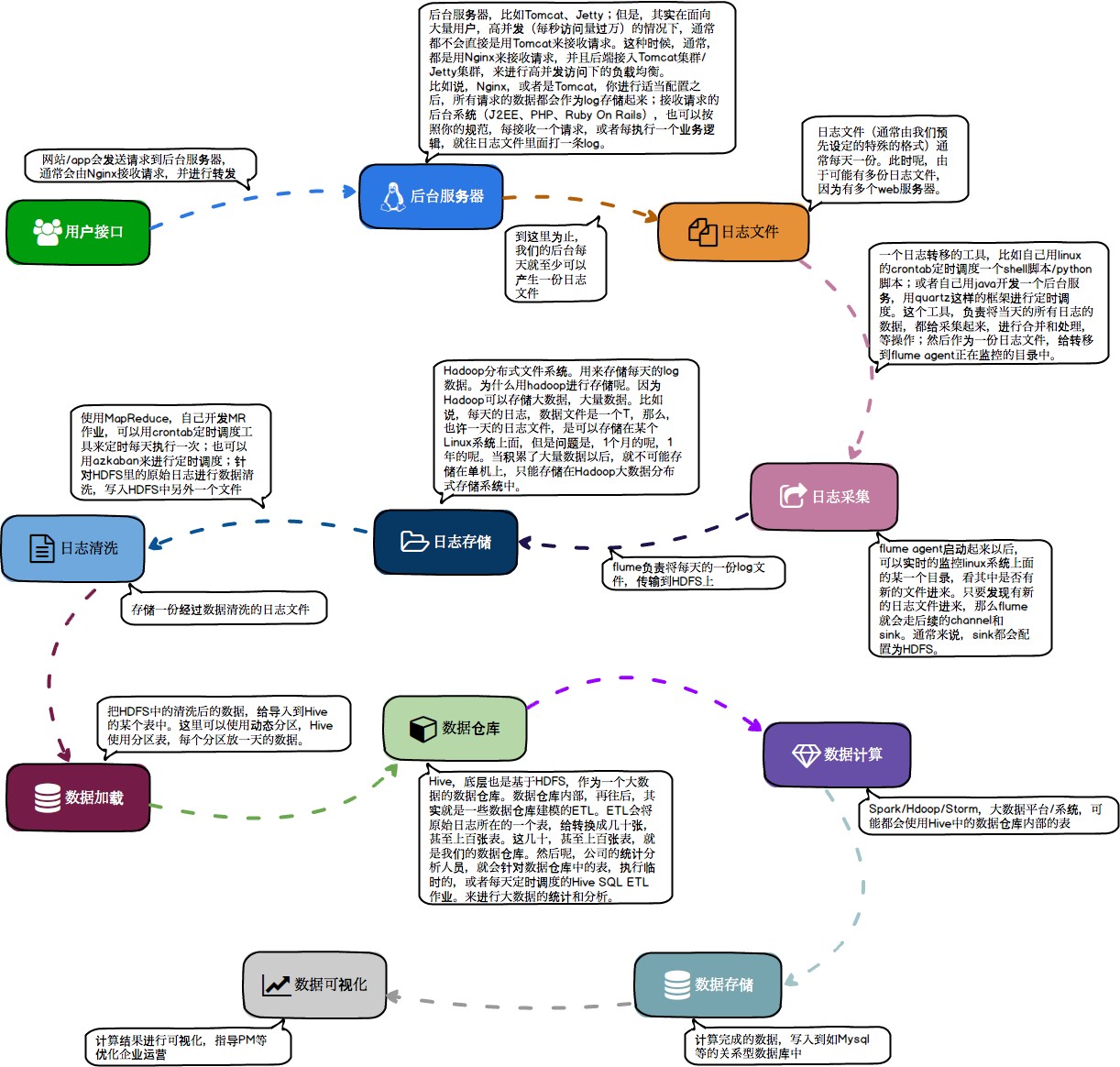


图 1-3 离线日志采集宏观流程

* + 1. **实时日志采集宏观流程**

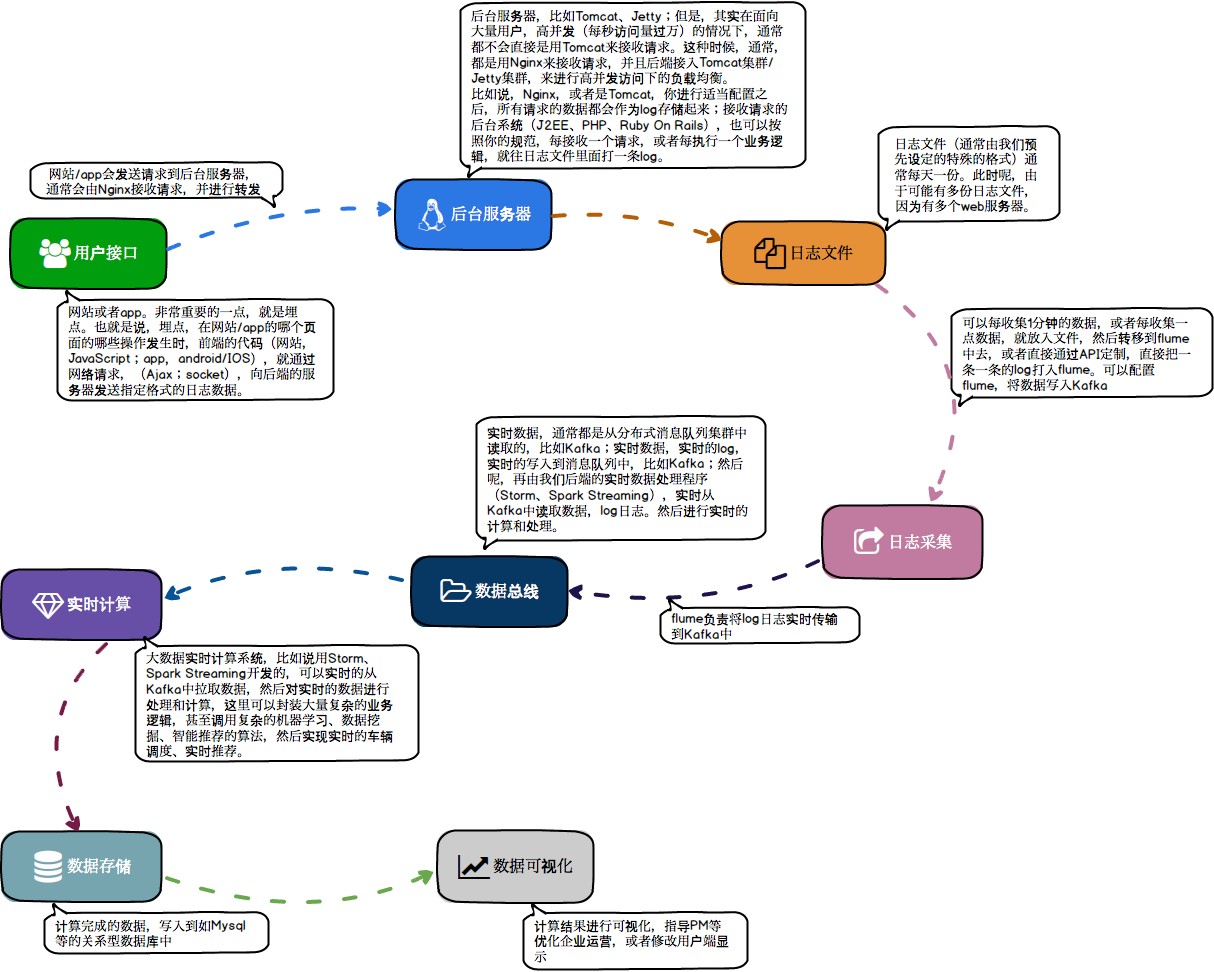


图 1-4 实时日志采集宏观流程

* + 1. **离线/实时日志采集框架**

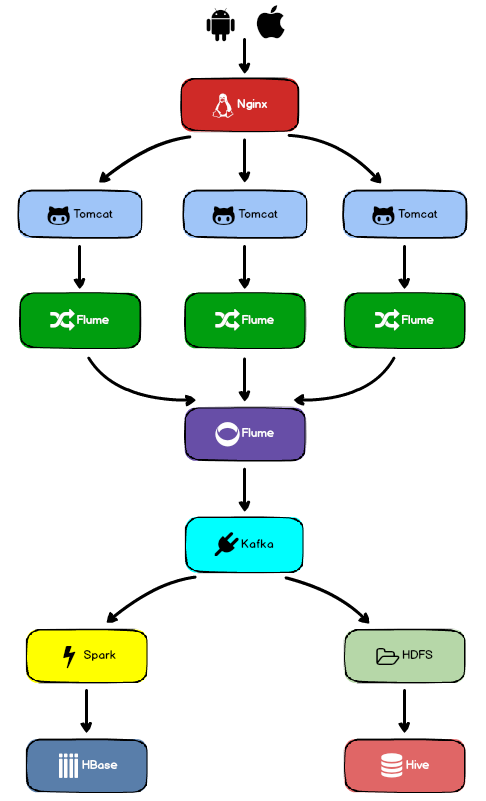


图 1-5 离线/实时日志采集框架

图 1-5 是一个企业级的日志处理框架， 这一框架实现了对日志信息进行采集、汇总、清洗、聚合、分析的完整过程， 并将日志数据分别存储到了离线和实时数据处理模块中， 使得分析系统可以通过离线和实时两个角度对数据进行分析统计，并根据统计结果指导业务平台的改良和优化。

* 1. **数据分析**

**1.3.1 离线数据分析**

* + 1. user\_visit\_action

user\_visit\_action 表，存放网站或者 APP 每天的点击流数据。通俗地讲，就是用

户对网站/APP 每点击一下， 就会产生一条存放在这个表里面的数据。



图 1-6 user\_visit\_action 表

user\_visit\_action 表中的字段解析如表 1-1 所示：

表 1-1 user\_visit\_action表字段解析

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| date | 日期，代表这个用户点击行为是在哪一天  发生的 |
| user\_id | 用户 ID，唯一地标识某个用户 |
| session\_id | Session ID，唯一地标识某个用户的一个访  问 session |
| page\_id | 页面 ID，点击了某些商品/品类，也可能是搜索了某个关键词，然后进入了某个页面，  页面的 id |
| action\_time | 动作时间，这个点击行为发生的时间点 |
| search\_keyword | 搜索关键词，如果用户执行的是一个搜索行为，比如说在网站/app 中，搜索了某个关键词，然后会跳转到商品列表页面； |
| click\_category\_id | 点击品类 ID，可能是在网站首页，点击了某个品类（美食、电子设备、电脑） |
| click\_product\_id | 点击商品 ID，可能是在网站首页，或者是在商品列表页，点击了某个商品（比如呷哺  呷哺火锅 XX 路店 3 人套餐、iphone 6s） |
| order\_category\_ids | 下单品类 ID，代表了可能将某些商品加入了购物车，然后一次性对购物车中的商品下了一个订单，这就代表了某次下单的行为中， 有哪些商品品类，可能有 6 个商品，但是就  对应了 2 个品类，比如有 3 根火腿肠（食品  品类），3 个电池（日用品品类） |
| order\_product\_ids | 下单商品 ID，某次下单，具体对哪些商品下的订单 |
| pay\_category\_ids | 付款品类 ID，对某个订单，或者某几个订单，进行了一次支付的行为，对应了哪些品类 |
| pay\_product\_ids | 付款商品 ID，支付行为下，对应的哪些具体的商品 |
| city\_id | 城市 ID，代表该用户行为发生在哪个城市 |

* + 1. user\_info

user\_info 表，是一张普通的用户基本信息表；这张表中存放了网站/APP 所有注册用户的基本信息。

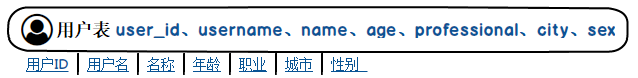


图 1-7 user\_info 表

user\_i 表中的字段解析如表 1-1 所示：

表 1-2 user\_info表字段解析

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| user\_id | 用户 ID，唯一地标识某个用户 |
| username | 用户登录名 |
| name | 用户昵称或真实姓名 |
| age | 用户年龄 |
| professional | 用户职业 |
| city | 用户所在城市 |
| sex | 用户性别 |

* + 1. product\_info

product\_info 表， 是一张普通的商品基本信息表； 这张表中存放了网站/APP 所有商品的基本信息。



图 1-8 product\_info 表

表 1-3 product\_info表字段解析

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| proudct\_id | 商品 ID，唯一地标识某个商品 |
| product\_name | 商品名称 |
| extend\_info | 额外信息，例如商品为自营商品还是第三  方商品 |

**1.3.2 在线数据分析**

程序每 5 秒向 Kafka 集群写入数据，格式如下： 格式 ： timestamp province city userid adid

表 1-4 在线数据分析

|  |  |
| --- | --- |
| 字段名称 | 取值范围 |

|  |  |
| --- | --- |
| timestamp | 当前时间毫秒 |
| userId | 0 – 99 |
| provice/city | 1 – 9  ((0L," 北京"," 北京"),(1L," 上海"," 上海"),(2L," 南京"," 江苏省  "),(3L,"广州","广东省"),(4L,"三亚","海南省"),(5L,"武汉","湖北省  "),(6L,"长沙","湖南省"),(7L,"西安","陕西省"),(8L,"成都","四川省  "),(9L,"哈尔滨","东北省")) |
| adid | 0 - 19 |

**1.4 项目需求**

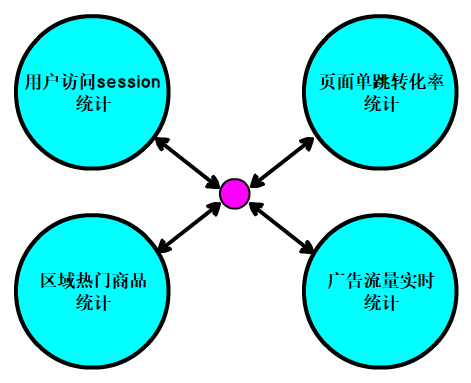


图 1-9 需求概述

* + 1. **用户访问 session 统计**

用户在电商网站上，通常会有很多的访问行为，通常都是进入首页， 然后可能点击首页上的一些商品，点击首页上的一些品类，也可能随时在搜索框里面搜索关键词，还可能将一些商品加入购物车，对购物车中的多个商品下订单，最后对订单中的多个商品进行支付。

用户的每一次操作，其实可以理解为一个 action，在本项目中，我们关注点击、搜索、下单、支付这四个用户行为。

用户 session，是在电商平台的角度定义的会话概念，指的就是，从用户第一次进入首页，session 就开始了。然后在一定时间范围内，直到最后操作完（ 可能做了几十次、甚至上百次操作），离开网站，关闭浏览器，或者长时间没有做操作，那么 session 就结束了。

以上用户在网站内的访问过程，就称之为一次 session。简单理解，session 就是某一天某一个时间段内，某个用户对网站从打开/进入，到做了大量操作，到最后关闭浏览器。的过程，就叫做 session。

session 实际上就是一个电商网站中最基本的数据和大数据。那么面向消费者/

用户端的大数据分析（ C 端） ， 最基本的就是面向用户访问行为/用户访问 session

的分析。

该模块主要是对用户访问 session 进行统计分析，包括 session 聚合指标计算、按时间比例随机抽取 session、获取每天点击、下单和购买排名前 10 的品类、并获取 top10 品类中排名前 10 的 session。该模块可以让产品经理、数据分析师以及企业管理层形象地看到各种条件下的具体用户行为以及统计指标，从而对公司的产品设计以及业务发展战略做出调整。主要使用 Spark Core 实现。

* + 1. **页面单跳转化率统计**

该模块主要是计算关键页面之间的单步跳转转化率，涉及到页面切片算法以及页面流匹配算法。该模块可以让产品经理、数据分析师以及企业管理层看到各个关键页面之间的转化率，从而对网页布局，进行更好的优化设计。主要使用 Spark Core 实现。

* + 1. **区域热门商品统计**

该模块主要实现每天统计出各个区域的 top3 热门商品。该模块可以让企业管理层看到电商平台在不同区域出售的商品的整体情况， 从而对公司的商品相关的战略进行调整。主要使用 Spark SQL 实现。

* + 1. **广告流量实时统计**

网站 / app 中经常会给第三方平台做广告，这也是一些互联网公司的核心收入来源；当广告位招商完成后，广告会在网站 / app 的某个广告位发布出去， 当用户访问网站 / app 的时候， 会看到相应位置的广告， 此时， 有些用户可能就会去点击那个广告。

我们要获取用户点击广告的行为，并针对这一行为进行计算和统计。

用户每次点击一个广告以后，会产生相应的埋点日志；在大数据实时统计系统中，会通过某些方式将数据写入到分布式消息队列中（ Kafka）。

日志发送给后台 web 服务器（ nginx），nginx 将日志数据负载均衡到多个 Tomcat 服务器上， Tomcat 服务器会不断将日志数据写入 Tomcat 日志文件中，写入后，就会被日志采集客户端（比如 flume agent）所采集，随后写入到消息队列中（ kafka）， 我们的实时计算程序会从消息队列中（ kafka）去实时地拉取数据，然后对数据进行实时的计算和统计。

这个模块的意义在于， 让产品经理、高管可以实时地掌握到公司打的各种广告的投放效果。以便于后期持续地对公司的广告投放相关的战略和策略，进行调整和优化；以期望获得最好的广告收益。

该模块负责实时统计公司的广告流量， 包括广告展现流量和广告点击流量。实

现动态黑名单机制， 以及黑名单过滤； 实现滑动窗口内的各城市的广告展现流量和广告点击流量的统计； 实现每个区域每个广告的点击流量实时统计； 实现每个区域

top3 点击量的广告的统计。主要使用 Spark Streaming 实现。

**第二章 预备知识**

* 1. **Scala**
     1. **Scala 操作符**

表 2-1 Scala操作符

|  |  |
| --- | --- |
| 操作符 | 说明 |
| :: | 该方法被称为 cons，意为构造，向队列的头部追加数据，创造新的列表。用法为 x::list, 其中 x 为加入到头部的元素，无论 x 是列表与否，它都只将成为新生成列表的第一个元素，也就是说新生成的列表长度为 list 的长度  ＋1(btw, x::list 等价于 list.::(x)) |
| :+ | :+方法用于在尾部追加元素 |
| +: | +:方法用于在头部追加元素，和::很类似，但  是::可以用于 pattern match ，而+:则不行 |
| ++ | 该方法用于连接两个集合 |
| ::: | 该方法只能用于连接两个 List 类型的集合 |

* + 1. **拉链操作**

###### 把一对集合 A 和 B 的包含的元素合成到一个集合中：

val prices = List(5.0, 20.0, 9.95)

val quantities = List(10, 2, 1)

prices zip quantities

运行结果：

List[(Double, Int)] = List((5.0, 10), (20.0, 2), (9.95, 1))

###### 这个方法之所以叫“拉链（ zip）”,是因为它就像拉链的齿状结构一样将两个集合结合在一起。

注意： 如果一个集合比另一个集合端， 那么结果中的对偶数量和较短的那个集合的元素数量相同。

zipAll 方法可以让你指定较短列表的缺省值。

zipWithIndex 方法返回对偶的列表， 其中每个对偶中第二个组成部分是每个元素的下标。

* 1. **SparkCore**

**2.2.1. Spark RDD 持久化**

Spark 非常重要的一个功能特性就是可以将 RDD 持久化在内存中，当对 RDD 执行持久化操作时，每个节点都会将自己操作的 RDD 的 partition 持久化到内存中， 并且在之后对该 RDD 的反复使用中，直接使用内存的 partition。这样的话，对于针对一个 RDD 反复执行多个操作的场景， 就只要对 RDD 计算一次即可，后面直接使用该 RDD， 而不需要反复计算多次该 RDD。

巧妙使用 RDD 持久化，甚至在某些场景下，可以将 Spark 应用程序的性能提高

10 倍。对于迭代式算法和快速交互式应用来说， RDD 持久化是非常重要的。

例如，读取一个有着数十万行数据的 HDFS 文件，形成 linesRDD，这一读取过程会消耗大量时间，在 count 操作结束后， linesRDD 会被丢弃， 会被后续的数据覆盖，当第二次再次使用 count 时，又需要重新读取 HDFS 文件数据， 再次形成新的

linesRDD， 这回导致反复消耗大量时间， 会严重降低系统性能。

如果在读取完成后将 linesRDD 缓存起来，那么下一次执行 count 操作时将会直接使用缓存起来的 linesRDD， 这会节省大量的时间。

要持久化一个 RDD， 只要调用其 cache()或者 persist()方法即可。在该 RDD 第一次被计算出来时，就会直接缓存在每个节点中， 而且 Spark 的持久化机制还是自动容错的，如果持久化的 RDD 的任何 partition 丢失了，那么 Spark 会自动通过其源RDD，使用 transformation 操作重新计算该 partition。

cache()和 persist()的区别在于，cache()是 persist()的一种简化方式，cache()的底层就是调用的 persist()的无参版本，同时就是调用 persist(MEMORY\_ONLY)，将输入持久化到内存中。如果需要从内存中清除缓存， 那么可以使用 unpersist()方法。

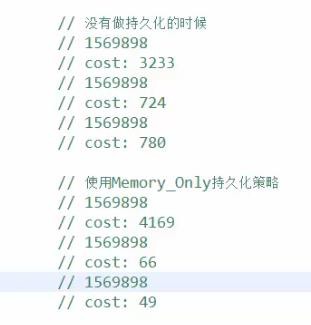
Spark 自己也会在 shuffle 操作时，进行数据的持久化，比如写入磁盘，主要是为了在节点失败时，避免需要重新计算整个过程。

表 2-2 Spark持久化级别

|  |  |
| --- | --- |
| 持久化级别 | 含义 |
| MEMORY\_ONLY | 以非序列化的 Java 对象的方式持久化在 JVM 内存中。如果内存无法完全存储 RDD 所有的  partition，那么那些没有持久化的 partition 就会在下一次需要使用它们的时候，重新被计算 |
| MEMORY\_AND\_DISK | 同上，但是当某些 partition 无法存储在内存中时，  会持久化到磁盘中。下次需要使用这些 partition |

|  |  |
| --- | --- |
|  | 时，需要从磁盘上读取 |
| MEMORY\_ONLY\_SER | 同 MEMORY\_ONLY，但是会使用 Java 序列化方  式，将 Java 对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大  CPU 开销 |
| MEMORY\_AND\_DISK\_SER | 同 MEMORY\_AND\_DISK，但是使用序列化方式  持久化 Java 对象 |
| DISK\_ONLY | 使用非序列化 Java 对象的方式持久化，完全存  储到磁盘上 |
| MEMORY\_ONLY\_2  MEMORY\_AND\_DISK\_2  等等 | 如果是尾部加了 2 的持久化级别，表示将持久  化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可 |

以下为对一个 156 万行大小为 168MB 的文本文件进行处理， textFile 后只进行

count 操作，持久化与不持久化的结果如下：

* + 1. **Spark 共享变量**

Spark 一个非常重要的特性就是共享变量。

默认情况下，如果在一个算子的函数中使用到了某个外部的变量，那么这个变量的值会被拷贝到每个 task 中，此时每个 task 只能操作自己的那份变量副本。如果多个 task 想要共享某个变量， 那么这种方式是做不到的。

Spark 为此提供了两种共享变量，一种是 Broadcast Variable（ 广播变量），另一种是 Accumulator（ 累加变量）。Broadcast Variable 会将用到的变量，仅仅为每个节点拷贝一份， 更大的用途是优化性能， 减少网络传输以及内存损耗。Accumulator 则可以让多个 task 共同操作一份变量， 主要可以进行累加操作。Broadcast Variable 是共享读变量，task 不能去修改它，而 Accumulator 可以让多个 task 操作一个变量。

* + - 1. 广播变量

广播变量允许程序员在每个机器上保留缓存的只读变量， 而不是给每个任务发送一个副本。 例如，可以使用它们以有效的方式为每个节点提供一个大型输入数据集的副本。Spark 还尝试使用高效的广播算法分发广播变量， 以降低通信成本。

Spark action 被划分为多个 Stages，被多个“shuffle”操作（ 宽依赖）所分割。Spark 自动广播每个阶段任务所需的公共数据（一个 Stage 中多个 task 使用的数据） ， 以这种方式广播的数据以序列化形式缓存，并在运行每个任务之前反序列化。 这意味着，显式创建广播变量仅在跨多个阶段的任务需要相同数据或者以反序列化格式缓存数据很重要时才有用。

Spark 提供的 Broadcast Variable 是只读的，并且在每个节点上只会有一个副本， 而不会为每个 task 都拷贝一份副本，因此，它的最大作用，就是减少变量到各个节点的网络传输消耗，以及在各个节点上的内存消耗。此外，Spark 内部也使用了高效的广播算法来减少网络消耗。

可以通过调用 SparkContext 的 broadcast()方法来针对每个变量创建广播变量。然后在算子的函数内， 使用到广播变量时， 每个节点只会拷贝一份副本了，每个节点可以使用广播变量的 value()方法获取值。

* + - 1. 累加器

累加器（ accumulator）：Accumulator 是仅仅被相关操作累加的变量，因此可以在并行中被有效地支持。它们可用于实现计数器（如 MapReduce）或总和计数。

Accumulator 是存在于 Driver 端的，从节点不断把值发到 Driver 端，在 Driver 端计数（ Spark UI 在 SparkContext 创建时被创建， 即在 Driver 端被创建，因此它可以读取 Accumulator 的数值） ， 存在于 Driver 端的一个值，从节点是读取不到的。

Spark 提供的 Accumulator 主要用于多个节点对一个变量进行共享性的操作。

Accumulator 只提供了累加的功能，但是却给我们提供了多个 task 对于同一个变量并行操作的功能，但是 task 只能对 Accumulator 进行累加操作，不能读取它的值， 只有 Driver 程序可以读取 Accumulator 的值。

自定义累加器类型的功能在 1.X 版本中就已经提供了，但是使用起来比较麻烦， 在 2.0 版本后， 累加器的易用性有了较大的改进， 而且官方还提供了一个新的抽象类： AccumulatorV2 来提供更加友好的自定义类型累加器的实现方式。

官方同时给出了一个实现的示例： CollectionAccumulator 类， 这个类允许以集合的形式收集 spark 应用执行过程中的一些信息。例如，我们可以用这个类收集 Spark 处理数据时的一些细节，当然，由于累加器的值最终要汇聚到 driver 端，为了避免

driver 端的 outofmemory 问题，需要对收集的信息的规模要加以控制，不宜过大。

代码清单 2-1 自定义AccumulatorV2

class SessionAggrStatAccumulator extends AccumulatorV2[String, mutable.HashMap[String, Int]] {

// 保存所有聚合数据

private val aggrStatMap = mutable.HashMap[String, Int]()

// 判断是否为初始值

* 1. **SparkSQL**

override def isZero: Boolean = { aggrStatMap.isEmpty

}

// 复制累加器

override def copy(): AccumulatorV2[String, mutable.HashMap[String, Int]] = { val newAcc = new SessionAggrStatAccumulator

aggrStatMap.synchronized{ newAcc.aggrStatMap ++= this.aggrStatMap

}

newAcc

}

// 重置累加器中的值

override def reset(): Unit = { aggrStatMap.clear()

}

// 向累加器中添加另一个值

override def add(v: String): Unit = { if (!aggrStatMap.contains(v))

aggrStatMap += (v -> 0) aggrStatMap.update(v, aggrStatMap(v) + 1)

}

// 各个 task 的累加器进行合并的方法

// 合并另一个类型相同的累加器

override def merge(other: AccumulatorV2[String, mutable.HashMap[String, Int]]): Unit =

{

other match {

case acc:SessionAggrStatAccumulator => {

(this.aggrStatMap /: acc.value){ case (map, (k,v)) => map += ( k -> (v + map.getOrElse(k,

0)) )}

}

}

}

// 获取累加器中的值

// AccumulatorV2 对外访问的数据结果

override def value: mutable.HashMap[String, Int] = { this.aggrStatMap

}

}

**2.3.1 RDD、DataFrame 与 DataSet**

* + 1. **RDD**

RDD，全称为 Resilient Distributed Datasets，即分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、 可分区、里面的元素可以并行计算的集合。在

Spark 中，对数据的所有操作不外乎创建 RDD、转化已有 RDD 以及调用 RDD 操作

进行求值。每个 RDD 都被分为多个分区， 这些分区运行在集群中的不同的节点上。

RDD 可以包含 Python、Java、Scala 中任意类型的对象， 甚至可以包含用户自定义的对象。RDD 具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升查询速度。

RDD 支持两种操作： transformation 操作和 action 操作。RDD 的 transformation 操作是返回一个新的 RDD 的操作，比如 map 和 filter()，而 action 操作则是向驱动器程序返回结果或者把结果写入外部系统的操作，比如 count()和 first()。

* + 1. **DataFrame**

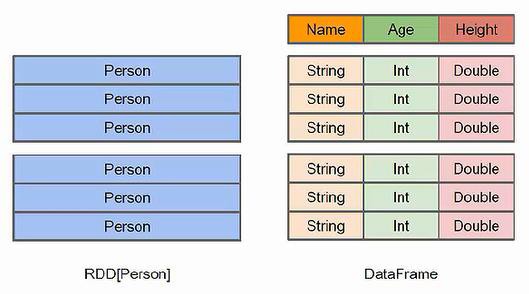
DataFrame 是一个分布式数据容器。相比于 RDD，DataFrame 更像传统数据库中的二维表格，除了数据之外，还记录数据的结构信息，即 schema。同时，与 Hive 类似，DataFrame 也支持嵌套数据类型（ struct，array 和 map）。从 API 易用性的角度上看， DataFrame API 提供的是一套高层的关系操作，比函数式的 RDD API 要更加友好，门槛更低。由于与 R 和 Pandas 中的 DataFrame 类似， Spark DataFrame 很好地继承了传统单机数据分析的开和体验。

图 2-1 RDD 与 DataFrame 对比

如上图所示，左侧的 RDD[Person]虽然以 Person 为类型参数，但是 Spark 框架本身不了解 Person 类的内部结构。而右侧的 DataFrame 却提供了详细的结构信息， 使得 SparkSQL 可以清楚地知道该数据集中包含那些列， 每列的名称是什么。

DataFrame 多了数据的结构信息，即 schema。RDD 是分布式的 Java 对象的集合。

DataFrame 是分布式的 Row 对象的集合。DataFrame 处理提供了比 RDD 更为丰富的算子以外，更重要的是提升了执行效率、减少数据读取以及执行计划的优化，比如

filter 下推、裁剪等。

DataFrame 是 DataSet 的特例， DataFrame = DataSet[Row]，所以可以通过 as 方法将 DataFrame 转换为 DataSet。Row 是一个类型， 跟 Car、Person 这些类型一样， 所有的表结构信息都用 Row 来表示。

* + 1. **DataSet**

DataSet 是 DataFrame API 的一个拓展，是 Spark 最新的数据抽象。DataSet 具有

用户友好的 API 风格， 既具有类型安全检查也具有 DataFrame 的查询优化特性。

DataSet 支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。

样例类被用来在 DataSet 中定义数据的结构信息，样例类中每个属性的名称直接映射到 DataSet 中的字段名称。

DataSet 是强类型的。比如可以有 DataSet[Car]， DataSet[Person]

DataFrame 只知道字段， 但是不知道字段的类型， 所以在执行这些操作的时候是没有办法在编译的时候检查是否类型失败的，比如你可以对一个 String 类型进行加减法操作，在执行的时候才会报错， 而 DataSet 不仅仅知道字段，而且知道字段类型，所以有更为严格的错误检查。就更 JSON 对象和类对象之间的类比。

* + 1. **DataSet 与 RDD 互操作**

介绍一下 Spark 将 RDD 转换成 DataFrame 的两种方式：

1.通过反射获取 Schema：使用 case class 的方式，不过在 scala 2.10 中最大支持

22 个字段的 case class，这点需要注意；

2.通过编程获取 Schema：通过 spark 内部的 StructType 方式，将普通的 RDD 转换成 DataFrame。

代码清单 2-2 DataSet与RDD互操作

object SparkRDDtoDF {

// 1. 通过编程获取 Schema

def rddToDF(sparkSession:SparkSession):DataFrame = {

//设置 schema 结构

val schema = StructType( Seq(

StructField("name",StringType,true)

,StructField("age",IntegerType,true)

)

)

val rowRDD = sparkSession.sparkContext

.textFile("file:/E:/scala\_workspace/z\_spark\_study/people.txt",2)

.map( x => x.split(",")).map( x => Row(x(0),x(1).trim().toInt)) sparkSession.createDataFrame(rowRDD,schema)

}

// 2. 通过反射获取 Schema

case class Person(name:String,age:Int)

def rddToDFCase(sparkSession : SparkSession):DataFrame = {

//导入隐饰操作，否则 RDD 无法调用 toDF 方法

import sparkSession.implicits.\_

val peopleRDD = sparkSession.sparkContext

.textFile("file:/E:/scala\_workspace/z\_spark\_study/people.txt",2)

.map( x => x.split(",")).map( x => Person(x(0),x(1).trim().toInt)).toDF() peopleRDD

}

def main(agrs : Array[String]):Unit = {

## RDD、DataFrame 与 DataSet 之间的转换

val conf = new SparkConf().setMaster("local[2]") conf.set("spark.sql.warehouse.dir","file:/E:/scala\_workspace/z\_spark\_study/") conf.set("spark.sql.shuffle.partitions","20")

val sparkSession = SparkSession.builder().appName("RDD to DataFrame")

.config(conf).getOrCreate()

// 通过代码的方式,设置 Spark log4j 的级别sparkSession.sparkContext.setLogLevel("WARN") import sparkSession.implicits.\_

//use case class convert RDD to DataFrame

//val peopleDF = rddToDFCase(sparkSession)

// 通过编程的方式完成 RDD 向

val peopleDF = rddToDF(sparkSession) peopleDF.show()

peopleDF.select($"name",$"age").filter($"age">20).show()

}

}

#### DataFrame/DataSet 转 RDD

val rdd1=testDF.rdd val rdd2=testDS.rdd

#### RDD 转 DataFrame

import spark.implicits.\_ val testDF = rdd.map {line=>

(line.\_1,line.\_2)

}.toDF("col1","col2")

#### RDD 转 DataSet

###### 参见 2.3.2 节 DataSet 与 RDD 互操作。

1. **DataFrame 转 DataSet**

import spark.implicits.\_ val testDF = testDS.toDF

#### DataSet 转 DataFrame

import spark.implicits.\_

//定义字段名和类型

case class Coltest(col1:String,col2:Int)extends Serializable val testDS = testDF.as[Coltest]

## 用户自定义聚合函数（UDAF）

#### 弱类型 UDAF 函数

###### 通过继承 UserDefinedAggregateFunction 来实现用户自定义聚合函数。

代码清单 2-3 弱类型UDAF

/\*\*

\* 用户自定义聚合函数

\*/

class GroupConcatDistinctUDAF extends UserDefinedAggregateFunction {

/\*\*

\* 聚合函数输入参数的数据类型

\*/

override def inputSchema: StructType = StructType(StructField("cityInfo", StringType) :: Nil)

/\*\*

* 聚合缓冲区中值的类型
* 中间进行聚合时所处理的数据类型

\*/

override def bufferSchema: StructType = StructType(StructField("bufferCityInfo", StringType) :: Nil)

/\*\*

* 函数返回值的数据类型

\*/

override def dataType: DataType = StringType

/\*\*

* 一致性检验，如果为 true，那么输入不变的情况下计算的结果也是不变的

\*/

override def deterministic: Boolean = true

/\*\*

* 设置聚合中间 buffer 的初始值
* 需要保证这个语义：两个初始 buffer 调用下面实现的 merge 方法后也应该为初始 buffer 即如果你初始值是

1，然后你 merge 是执行一个相加的动作，两个初始 buffer 合并之后等于 2，不会等于初始 buffer 了。这样的初始值就是有问题的，所以初始值也叫"zero value"

\*/

override def initialize(buffer: MutableAggregationBuffer): Unit = { buffer(0)= ""

}

/\*\*

* 用输入数据 input 更新 buffer 值,类似于 combineByKey

\*/

override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {

// 缓冲中的已经拼接过的城市信息串

var bufferCityInfo = buffer.getString(0)

// 刚刚传递进来的某个城市信息

val cityInfo = input.getString(0)

// 在这里要实现去重的逻辑

// 判断：之前没有拼接过某个城市信息，那么这里才可以接下去拼接新的城市信息

if(!bufferCityInfo.contains(cityInfo)) { if("".equals(bufferCityInfo))

bufferCityInfo += cityInfo else {

// 比如 1:北京

// 1:北京,2:上海

bufferCityInfo += "," + cityInfo

}

buffer.update(0, bufferCityInfo)

}

}

/\*\*

* 合并两个 buffer, 将 buffer2 合并到 buffer1. 在合并两个分区聚合结果的时候会被用到, 类似于

reduceByKey

* 这里要注意该方法没有返回值，在实现的时候是把 buffer2 合并到 buffer1 中去，你需要实现这个合并细节

\*/

override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = { var bufferCityInfo1 = buffer1.getString(0);

val bufferCityInfo2 = buffer2.getString(0);

for(cityInfo <- bufferCityInfo2.split(",")) { if(!bufferCityInfo1.contains(cityInfo)) {

if("".equals(bufferCityInfo1)) { bufferCityInfo1 += cityInfo;

} else {

bufferCityInfo1 += "," + cityInfo;

}

}

}

buffer1.update(0, bufferCityInfo1);

}

/\*\*

* 计算并返回最终的聚合结果

\*/

override def evaluate(buffer: Row): Any = { buffer.getString(0)

}

}

#### 强类型 UDAF 函数

###### 通过继承 Aggregator 来实现强类型自定义聚合函数。

代码清单 2-4 强类型UDAF

// 定义 case 类

case class Employee(name: String, salary: Long) case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {

/\*\*

\* 计算并返回最终的聚合结果

\*/

def zero: Average = Average(0L, 0L)

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

/\*\*

\* 根据传入的参数值更新 buffer 值

\*/

def reduce(buffer: Average, employee: Employee): Average = { buffer.sum += employee.salary

buffer.count += 1 buffer

}

/\*\*

\* 合并两个 buffer 值,将 buffer2 的值合并到 buffer1

\*/

def merge(b1: Average, b2: Average): Average = { b1.sum += b2.sum

b1.count += b2.count b1

}

/\*\*

\* 计算输出

\*/

def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count

/\*\*

* 设定中间值类型的编码器，要转换成 case 类
* Encoders.product 是进行 scala 元组和 case 类转换的编码器

\*/

def bufferEncoder: Encoder[Average] = Encoders.product

/\*\*

* 设定最终输出值的编码器

\*/

def outputEncoder: Encoder[Double] = Encoders.scalaDouble

}

* + 1. **开窗函数**

###### 开窗函数与聚合函数一样，都是对行的集合组进行聚合计算。

开窗用于为行定义一个窗口（ 这里的窗口是指运算将要操作的行的集合） ， 它对一组值进行操作，不需要使用 GROUP BY 子句对数据进行分组，能够在同一行中同时返回基础行的列和聚合列。

开窗函数的调用格式为： **函数名(列) OVER(选项)**

###### 第一大类： 聚合开窗函数  聚合函数(列) OVER (选项)，这里的选项可以是

PARTITION BY 子句，但不可是 ORDER BY 子句。

第二大类： 排序开窗函数  排序函数(列) OVER(选项)， 这里的选项可以是ORDER BY 子句， 也可以是 OVER（PARTITION BY 子句 ORDER BY 子句）， 但不可以是 PARTITION BY 子句。

代码清单 2-5 开窗函数main函数

def main(args: Array[String]): Unit = {

val sparkConf = new SparkConf().setAppName("score").setMaster("local[\*]")

**———————————————**

**尚硅谷大数据项目之电商分析平台**

### ——————————————

val sparkSession = SparkSession.builder().config(sparkConf).getOrCreate()

import sparkSession.implicits.\_

val scoreDF = sparkSession.sparkContext.makeRDD(Array(Score("a1", 1, 80),

Score("a2", 1, 78),

Score("a3", 1, 95),

Score("a4", 2, 74),

Score("a5", 2, 92),

Score("a6", 3, 99),

Score("a7", 3, 99),

Score("a8", 3, 45),

Score("a9", 3, 55),

Score("a10", 3, 78))).toDF("name", "class

", "score") scoreDF.createOrReplaceTempView("score") scoreDF.show()

}

#### 聚合开窗函数

###### OVER 关键字表示把聚合函数当成聚合开窗函数而不是聚合函数。SQL 标准允许将所有聚合函数用做聚合开窗函数。

sparkSession.sql("select name, class, score, count(name) over() name\_count from score")

###### 查询结果如下所示：

+----+-----+-----+----------+

|name|class|score|name\_count|

+----+-----+-----+----------+

| a1|

| a2|

| a3|

| a4|

| a5|

| a6|

| a7|

| a8|

| a9|

| a10|

1| 80|

1| 78|

1| 95|

2| 74|

2| 92|

3| 99|

3| 99|

3| 45|

3| 55|

3| 78|

10|

10|

10|

10|

10|

10|

10|

10|

10|

10|

+----+-----+-----+----------+

在上边的例子中，开窗函数 COUNT(\*) OVER()对于查询结果的每一行都返回所有符合条件的行的条数。OVER 关键字后的括号中还经常添加选项用以改变进行聚合运算的窗口范围。如果 OVER 关键字后的括号中的选项为空，则开窗函数会对结果集中的所有行进行聚合运算。

###### 开窗函数的 OVER 关键字后括号中的可以使用 PARTITION BY 子句来定义行的分区来供进行聚合计算。与 GROUP BY 子句不同，PARTITION BY 子句创建的分区是独立于结果集的，创建的分区只是供进行聚合计算的，而且不同的开窗函数所创建的分区也不互相影响。下面的 SQL 语句用于显示按照班级分组后每组的人数：

sparkSession.sql("select name, class, score, count(name) over(partition by class) name\_count from score").show()

查询结果如下所示：

+----+-----+-----+----------+

|name|class|score|name\_count|

+----+-----+-----+----------+

| a1|

| a2|

| a3|

| a6|

| a7|

| a8|

| a9|

| a10|

| a4|

| a5|

1| 80|

1| 78|

1| 95|

3| 99|

3| 99|

3| 45|

3| 55|

3| 78|

2| 74|

2| 92|

3|

3|

3|

5|

5|

5|

5|

5|

2|

2|

+----+-----+-----+----------+

###### OVER(PARTITION BY class)表示对结果集按照 class 进行分区，并且计算当前行所属的组的聚合计算结果。在同一个 SELECT 语句中可以同时使用多个开窗函数， 而且这些开窗函数并不会相互干扰。比如下面的 SQL 语句用于显示每一个人员的信息、所属城市的人员数以及同龄人的人数：

sparkSession.sql("select name, class, score, count(name) over(partition by class) name\_count1 from score").show()

sparkSession.sql("select name, class, score, count(name) over(partition by score)

name\_count2 from score").show()

* + - 1. **排序开窗函数**

对于排序开窗函数来讲，它支持的开窗函数分别为：ROW\_NUMBER（ 行号）、

###### RANK（排名）、DENSE\_RANK（ 密集排名）和 NTILE（ 分组排名）。

sparkSession.sql("select name, class, score, row\_number() over(order by score) rank from score").show()

sparkSession.sql("select name, class, score, rank() over(order by score) rank from score").show()

sparkSession.sql("select name, class, score, dense\_rank() over(order by score) rank from score").show()

sparkSession.sql("select name, class, score, ntile(6) over(order by score) rank from

score").show()

执行的结果如下：

+----+-----+-----+----+

|name|class|score|rank|

+----+-----+-----+----+

| a8|

| a9|

| a4|

| a2|

| a10|

| a1|

| a5|

| a3|

| a6|

| a7|

3| 45| 1|

3| 55| 2|

2| 74| 3|

1| 78| 4|

3| 78| 5|

1| 80| 6|

2| 92| 7|

1| 95| 8|

3| 99| 9|

3| 99| 10|

+----+-----+-----+----+

+----+-----+-----+----+

|name|class|score|rank|

+----+-----+-----+----+

| a8|

| a9|

| a4|

| a2|

| a10|

| a1|

| a5|

| a3|

| a6|

| a7|

3| 45| 1|

3| 55| 2|

2| 74| 3|

1| 78| 4|

3| 78| 4|

1| 80| 6|

2| 92| 7|

1| 95| 8|

3| 99| 9|

3| 99| 9|

+----+-----+-----+----+

+----+-----+-----+----+

|name|class|score|rank|

+----+-----+-----+----+

| a8|

| a9|

| a4|

| a2|

| a10|

| a1|

| a5|

| a3|

| a6|

| a7|

3| 45| 1|

3| 55| 2|

2| 74| 3|

1| 78| 4|

3| 78| 4|

1| 80| 5|

2| 92| 6|

1| 95| 7|

3| 99| 8|

3| 99| 8|

+----+-----+-----+----+

+----+-----+-----+----+

|name|class|score|rank|

+----+-----+-----+----+

| a8|

| a9|

| a4|

| a2|

| a10|

| a1|

| a5|

| a3|

| a6|

| a7|

3| 45| 1|

3| 55| 1|

2| 74| 2|

1| 78| 2|

3| 78| 3|

1| 80| 3|

2| 92| 4|

1| 95| 4|

3| 99| 5|

3| 99| 6|

+----+-----+-----+----+

###### 看到上面的结果了吧， 下面来介绍下相关的内容。我们得到的最终结果是按照

score 进行升序显示的。

对于 row\_number() over(order by score) as rownum 来说，这个排序开窗函数是按

score 升序的方式来排序，并得出排序结果的序号。

对于 rank() over(order by score) as rank 来说，这个排序形容函数是按 FSalary 升序的方式来排序，并得出排序结果的排名号。这个函数求出来的排名结果可以并列， 并列排名之后的排名将是并列的排名加上并列数（简单说每个人只有一种排名，然 后出现两个并列第一名的情况，这时候排在两个第一名后面的人将是第三名，也就 是没有了第二名，但是有两个第一名）。

对于 dense\_rank() over(order by score) as dense\_rank 来说，这个排序函数是按

score 升序的方式来排序，并得出排序结果的排名号。这个函数与 rank()函数不同在

于，并列排名之后的排名只是并列排名加１ （ 简单说每个人只有一种排名， 然后出现两个并列第一名的情况，这时候排在两个第一名后面的人将是第二名，也就是两个第一名，一个第二名）。

对于 ntile(6) over(order by score)as ntile 来说，这个排序函数是按 FSalary 升序的方式来排序，然后 6 等分成 6 个组， 并显示所在组的序号。

排序函数和聚合开窗函数类似，也支持在 OVER 子句中使用 PARTITION BY 语句。例如：

sparkSession.sql("select name, class, score, row\_number() over(partition by class order by score) rank from score").show()

sparkSession.sql("select name, class, score, rank() over(partition by class order by score) rank from score").show()

sparkSession.sql("select name, class, score, dense\_rank() over(partition by class order by score) rank from score").show()

sparkSession.sql("select name, class, score, ntile(6) over(partition by class order by score)

rank from score").show()

需要注意一点， 在排序开窗函数中使用 PARTITION BY 子句需要放置在

ORDER BY 子句之前。

* 1. **SparkStreaming**
     1. **Dstream transformation 算子概览**

表 2-3 Spark持久化级别

|  |  |
| --- | --- |
| 算子名称 | 说明 |
| map | 对每个传入的元素，返回一个新的元素 |
| flatMap | 对每个传入的 u 安苏，返回一个或多个元素 |
| fliter | 对传入的元素返回 true 或 false，返回 false  的元素将会被过滤掉 |
| union | 将连个 Dsrteam 进行合并 |
| count | 返回元素的个数 |
| reduce | 对所有 value 进行聚合 |
| countByValue | 对元素按照值进行分组，对每个组进行计  数，最后返回<K, V>的格式 |
| reduceByKey | 对 Key 对应的 values 进行聚合 |
| cogroup | 对两个 Dstream 进行连接操作 |
| join | 对两个 Dstream 进行 join 操作，每个链接起来的 pair，作为新 Dstream 的 RDD 的一个元素 |
| transform | 对数据进行转换操作 |

|  |  |
| --- | --- |
| updateStaeByKey | 对每个 key 维护一份 state，并进行更新 |
| window | 对滑动窗口数据进行操作 |

* + 1. **Dstream updataStateByKey 算子**

updateStateByKey 操作， 可以让我们为每一个 key 维护一根 state，并持续不断地更新该 state。

1. 首先，要定义一个 state，可以是任意的数据类型；
2. 其次，要定义 state 更新函数—— 指定一个函数如何使用之前的 state 和新值来更新 state；

对于每个 batch，Spark 都会为每个之前已经存在的 key 去应用一次 state 更新函数，无论这个 key 在 batch 中是否有新的数据。如果 state 更新函数返回 none，那么

key 对应的 state 就会被删除。

当然，对于每一个新出现的 key，也会执行 state 更新函数。

注意， updateStateByKey 操作，要求必须开启 Checkpoint 机制。

代码清单 2-6 updateStateByKey实例

object updateStateByKeyWordCount {

def main(args: Array[String]): Unit = {

val conf = new SparkConf().setMaster("local[2]").setAppName("Wordcount") val ssc = new StreamingContext(conf, Seconds(1)) ssc.checkpoint("hdfs://s100:8020/wordcount\_checkpoint")

val lines = ssc.socketTextStream("localhost", 9999) val words = lines.flatMap(\_.split(" "))

val pairs = words.map(word => (word, 1))

val wordCount = pairs.updateStateByKey((values:Seq[Int], state:Option[Int]) =>{ var newValue = state.getOrElse(0)

for(value <- values){ newValue += value

}

Option(newValue)

})

wordCount.print()

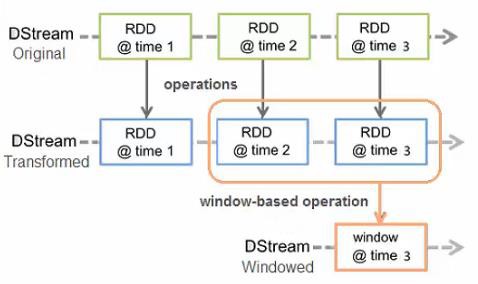
ssc.start() ssc.awaitTermination()

}

}

* + 1. **窗口操作**

Spark Streaming 提供了窗口计算，允许在数据的滑动窗口上应用转换，下图说明了这个滑动窗口：

图 2-2 窗口操作

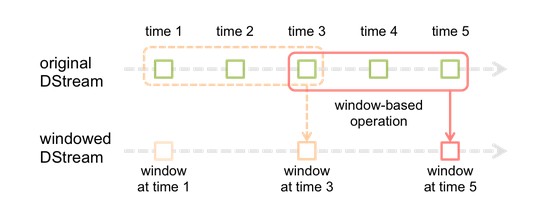


图 2-3 窗口操作

如图所示，每当窗口滑过源 DStream 时， 落在窗口内的源 RDD 被组合并运行， 以产生窗口 DStream 的 RDD。 在这种具体情况下， 操作应用于最近 3 个时间单位的数据，并以 2 个时间单位滑动。 这表明任何窗口操作都需要指定两个参数。

* + - * **窗口长度** - 窗口的持续时间（ 此图中窗口长度为 3）。
      * **滑动间隔** - 执行窗口操作的间隔（ 此图中滑动间隔为 2）。

###### 这两个参数必须是源 DStream 的 batch 间隔的倍数（此图中 batch 间隔为 1）。

batch 间隔为切割 RDD 的间隔，滑动间隔为每隔多长时间来计算一次，窗口长度为每次计算的数据量是多少。

* + 1. **Receiver 与 Direct**

1. Receiver

Receiver 是使用 Kafka 的高层次 Consumer API 来实现的。Receiver 每隔一段

batch 时间去 Kafka 获取那段时间最新的消息数据，Receiver 从 Kafka 获取的数据都是存储在 Spark Executor 的内存中的，然后 Spark Streaming 启动的 job 会去处理那些数据。

对于高阶消费者，谁来消费分区不是由 Spark Streaming 决定的， 也不是 Storm 决定的，有一个高阶消费者 API， 由高阶消费者决定分区向消费者的分配， 即由高阶消费者 API 决定消费者消费哪个分区， 而消费者读取数据后什么时候提交 offset 也不是由它们自己决定的，高阶消费者 API 会根据参数配置隔几秒提交一次。

这会引起一个问题，当 Spark Streaming 中的 Receiver 读取 Kafka 分区数据时， 假设读取了 100 条数据， 高阶消费者 API 会执行 offset 的提交，例如每隔 3 秒，这

100 条数据就是 RDD，假设此 RDD 还没有处理完， 高阶消费者 API 执行了 offset

提交，但是 Spark Streaming 挂掉了，由于 RDD 在内存中，那么 RDD 的数据就丢失了， 如果想重新拿数据， 从哪里去拿不是由 Spark Streaming 说了算的， 是由高阶

API 决定的，由于 offset 已经提交，高阶 API 认为这个数据 Spark Streaming 已经拿过了，再拿要拿 100 条以后的数据， 那么之前丢失的 100 条数据就永远丢失了。

针对这一问题，Spark Streaming 设计了一个规则，即 Spark Streaming 预写日志规则（ Write Ahead Log， WAL） ，每读取一批数据，会写一个 WAL 文件， 在 WAL 文件中，读了多少条就写多少条， WAL 文件存储于 HDFS 上。假设 RDD 中有 100 条数据，那么 WAL 文件中也有 100 条数据， 此时如果 Spark Streaming 挂掉，那么回去读取 HDFS 上的 WAL 文件，把 WAL 文件中的 100 条数据取出再生成 RDD，然后再去消费。由于这一设计需要写 HDFS，会对整体性能造成影响。

假设有 6 个分区，高阶消费者的话会在 Spark 集群的 Worker 上启动 Receiver， 有 6 个分区则会用 6 个线程去读取分区数据，这是在一个 Worker 的一个 Receiver 中有 6 个线程同时读取 6 个分区的数据， 随着数据量越来越大， 数据读取会成为瓶颈，此时可以创建多个 Receiver 分散读取分区数据， 然后每个 Receiver 创建一个

Dstream，再把这些流全部都合并起来，然后进行计算。读取时，一方面把 RDD 放在内存中，一方面写 HDFS 中的 WAL 文件。

根据上面的情景，又要创建多个 Receiver，又要进行合并，又要在内存中存储 RDD， 又要写 HDFS 上的 WAL 文件，高级 API 的缺点还是比较多的。

高阶消费者是由高阶消费者 API 自己提交 offset 到 ZooKeeper 中。

1. Direct

低阶消费者需要自己维护 offset， Spark Streaming 从分区里读一部分数据， 然后将 offset 保存到 CheckpointPath 目录中， 比如 5s 生成一个 Spark Streaming job

（每个 action 操作启动一次 job） ， 每个 job 生成的时候， 会写一次 CheckpointPath下的文件， Checkpoint 中有 job 信息和 offset 信息（当然还有 RDD 依赖关系等其他信息），即保存了未完成的 job 和分区读取的 offset，一旦 Spark Streaming 挂掉后重启，可以通过从 CheckpointPath 中的文件中反序列化来读取 Checkpoint 的数据。假设有 5 个分区，第一次 Spark Streaming 读取 100 条数据，那么每个 partition

都会读取 100 条数据，这 100 条数据对应 offset 是 0~99，这 5 个分区的 100 条数据数据直接对应 RDD 的 5 个分区，针对这一 RDD 会启动一个 job 进行处理，job 启动时会将 job 信息和 offset（ 0~99）写入 CheckpointPath，处理完成前保存 job 和 offset， 一旦处理完成，job 信息会被删除，但是 offset 信息会被保留，通过这次的 offset 确定下一次的读取范围，即 100~199，新的 job 信息会被写入，新的 offset 100~199 覆盖原来的 0~99。如果处理第二批次的时候挂掉了， offset 还在， 就可以重读这块数据。

* 1. **Java**
     1. **对象池**

在学习 MySQL 时，我们接触到了数据库连接池技术，数据库连接池负责分配、管理和释放数据库连接， 它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个； 释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。这项技术能明显提高对数据库操作的性能。

在实际开发时，对象的创建和销毁操作也是非常消耗资源的，因此， 我们考虑使用对象池技术。当我们需要创建对象时， 向对象池申请一个对象， 如果对象池里有空闲的可用节点， 就会把节点返回给用户； 当我们需要销毁对象时，将对象返回给对象池即可。

我们常用的数据库连接池是 C3P0 等数据库连接池， 根据对象池的概念， 我们发现对象池与数据库连接池有很大的相似之处，其实，很多数据库连接池就是借助对象池技术实现的，因此，我们可以通过对象池实现自己的数据库连接池。

代码清单 2-1 MySQL代理

/\*\*

* MySQL 客户端代理对象

\*

* @param jdbcUrl MySQL URL
* @param jdbcUser MySQL 用户
* @param jdbcPassword MySQL 密码
* @param client 默认客户端实现

\*/

case class MySqlProxy(jdbcUrl: String, jdbcUser: String, jdbcPassword: String, client: Option[Connection] = None) {

// 获取客户端连接对象

private val mysqlClient = client getOrElse { DriverManager.getConnection(jdbcUrl, jdbcUser, jdbcPassword)

}

/\*\*

* + 执行增删改 SQL 语句

\*

* + @param sql
  + @param params
  + @return 影响的行数

\*/

def executeUpdate(sql: String, params: Array[Any]): Int = { var rtn = 0

var pstmt: PreparedStatement = null

try {

// 第一步：关闭自动提交

mysqlClient.setAutoCommit(false)

// 第二步：根据传入的 sql 语句创建 prepareStatement pstmt = mysqlClient.prepareStatement(sql)

// 第三步：为 prepareStatement 中的每个参数填写数值

if (params != null && params.length > 0) { for (i <- 0 until params.length) {

pstmt.setObject(i + 1, params(i))

}

}

// 第四步：执行增删改操作

rtn = pstmt.executeUpdate()

// 第五步：手动提交

mysqlClient.commit()

} catch {

case e: Exception => e.printStackTrace

}

rtn

}

/\*\*

* 执行查询 SQL 语句

\*

* @param sql
* @param params

\*/

def executeQuery(sql: String, params: Array[Any], queryCallback: QueryCallback) { var pstmt: PreparedStatement = null

var rs: ResultSet = null

try {

// 第一步：根据传入的 sql 语句创建 prepareStatement pstmt = mysqlClient.prepareStatement(sql)

// 第二步：为 prepareStatement 中的每个参数填写数值

if (params != null && params.length > 0) { for (i <- 0 until params.length) {

pstmt.setObject(i + 1, params(i))

}

}

// 第三步：执行查询操作

rs = pstmt.executeQuery()

// 第四步：处理查询后的结果

queryCallback.process(rs)

} catch {

case e: Exception => e.printStackTrace

}

}

/\*\*

* 批量执行 SQL 语句

\*

* @param sql
* @param paramsList
* @return 每条 SQL 语句影响的行数

\*/

def executeBatch(sql: String, paramsList: Array[Array[Any]]): Array[Int] = { var rtn: Array[Int] = null

var pstmt: PreparedStatement = null try {

// 第一步：关闭自动提交

mysqlClient.setAutoCommit(false)

pstmt = mysqlClient.prepareStatement(sql)

**———————————————**

**尚硅谷大数据项目之电商分析平台**

### ——————————————

// 第二步：为 prepareStatement 中的每个参数填写数值

if (paramsList != null && paramsList.length > 0) { for (params <- paramsList) {

for (i <- 0 until params.length) { pstmt.setObject(i + 1, params(i))

}

pstmt.addBatch()

}

}

// 第三步：执行批量的 SQL 语句

rtn = pstmt.executeBatch()

// 第四步：手动提交

mysqlClient.commit()

} catch {

case e: Exception => e.printStackTrace

}

rtn

}

// 关闭 MySQL 客户端

def shutdown(): Unit = mysqlClient.close()

}

###### 如代码清单 2-1 所示，我们完成了 MySQL 代理类 MySqlProxy 的创建， 每个

MySqlProxy 对象都会完成一次与 MySQL 的连接并提供操作 MySQL 数据库的接口， 那么如果我们将 MySqlProxy 对象创建的工作交给对象池，那么就可以实现重复利用与 MySQL 建立的连接， 这与数据库连接池的功能是一样的。

在本项目中，我们使用了 Apache common-pool2 框架， Apache common-pool2 包提供了一个通用的对象池技术的实现。可以很方便的基于它来实现自己的对象池， 比如 DBCP 和 Jedis 他们的内部对象池的实现就是依赖于 common-pool2。

common-pool2 有四个核心：

1. 工作类： 要通过对象池创建对象的类， 例如 MySqlProxy 类；
2. 工厂类： 生产工作类的工厂， 工厂类是基于 BasePooledObjectFactory 的；
3. 配置类 ：对象池活跃对象个数、最大空闲数等信息都需要配置，基于

GenericObjectPoolConfig；

1. 对象池： 实际的对象池类， 基于 GenericObjectPool， 其对象的创建需要传入工厂类对象和配置类对象。

common-pool2 的对应关系如图 2-1 所示：

**———————————————**

**尚硅谷大数据项目之电商分析平台**

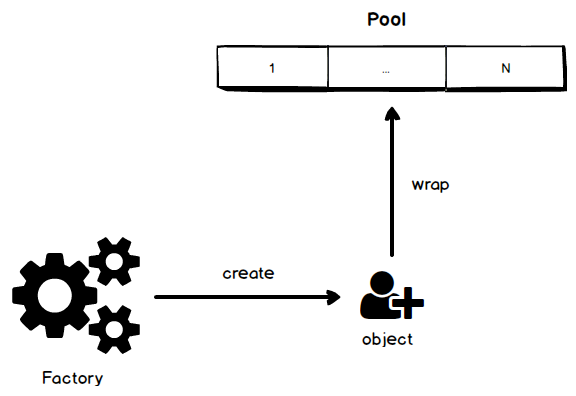
**——————————————**

图 2-1 common-pool2 对应关系

代码清单 2-2 对象池实现

/\*\*

* 创建自定义工厂类，继承 BasePooledObjectFactory 工厂类，负责对象的创建、包装和销毁
* @param jdbcUrl
* @param jdbcUser
* @param jdbcPassword
* @param client

\*/

class PooledMySqlClientFactory(jdbcUrl: String, jdbcUser: String, jdbcPassword: String, client: Option[Connection] = None) extends BasePooledObjectFactory[MySqlProxy] with Serializable {

// 用于池来创建对象

override def create(): MySqlProxy = MySqlProxy(jdbcUrl, jdbcUser, jdbcPassword, client)

// 用于池来包装对象

override def wrap(obj: MySqlProxy): PooledObject[MySqlProxy] = new DefaultPooledObject(obj)

// 用于池来销毁对象

override def destroyObject(p: PooledObject[MySqlProxy]): Unit = { p.getObject.shutdown()

super.destroyObject(p)

}

}

/\*\*

* 创建 MySQL 池工具类

\*/

object CreateMySqlPool {

// 加载 JDBC 驱动，只需要一次

Class.forName("com.mysql.jdbc.Driver")

// 在 org.apache.commons.pool2.impl 中预设了三个可以直接使用的对象池： GenericObjectPool、

GenericKeyedObjectPool 和 SoftReferenceObjectPool

// 创建 genericObjectPool 为 GenericObjectPool

// GenericObjectPool 的特点是可以设置对象池中的对象特征，包括 LIFO 方式、最大空闲数、最小空闲数、是否有效性检查等等

private var genericObjectPool: GenericObjectPool[MySqlProxy] = null

// 伴生对象通过 apply 完成对象的创建

def apply(): GenericObjectPool[MySqlProxy] = {

// 单例模式

if (this.genericObjectPool == null) { this.synchronized {

// 获取 MySQL 配置参数

val jdbcUrl = ConfigurationManager.config.getString(Constants.JDBC\_URL) val jdbcUser = ConfigurationManager.config.getString(Constants.JDBC\_USER)

val jdbcPassword = ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD) val size = ConfigurationManager.config.getInt(Constants.JDBC\_DATASOURCE\_SIZE)

val pooledFactory = new PooledMySqlClientFactory(jdbcUrl, jdbcUser, jdbcPassword) val poolConfig = {

// 创建标准对象池配置类的实例

val c = new GenericObjectPoolConfig

// 设置配置对象参数

// 设置最大对象数

c.setMaxTotal(size)

// 设置最大空闲对象数c.setMaxIdle(size) c

}

// 对象池的创建需要工厂类和配置类

// 返回一个 GenericObjectPool 对象池

this.genericObjectPool poolConfig)

}

}

genericObjectPool

}

}

= new

GenericObjectPool[MySqlProxy](pooledFactory,

GenericObjectPool 的核心方法如下：

borrowObject()：从对象池中取出一个对象。

returnObject()：将使用完成的对象还给对象池。

在 每 次 使 用 CreateMySqlPool 时， 通过 borrowObject() 提 取 对 象 ， 通过

returnObject()归还对象。

**第三章 程序框架解析**

* 1. **模块分析**

**3.1.1 commons 模块**

表 3-1 commons模块

|  |  |
| --- | --- |
| **包名称（package）** | **解析** |
| **conf** | 配置工具类  获取 commerce.properties 文件中的所有配置信息，使用户可以通过对象的方式访问 |

|  |  |
| --- | --- |
|  | commerce.properties 中的所有配置 |
| **constant** | 常量接口  包括项目中所需要使用的所有常量 |
| **model** | Spark SQL 样例类  包括 Spark SQL 中的用户访问动作表、用户信息表、产品表的样例类 |
| **pool** | MySQL 连接池  通过自定义MySQL 连接池实现对MySQL 数据库的操作 |
| **utils** | 工具类  提供了日期时间工具类、数字格式工具类、参数工具类、字符串工具类、校验工具类等工具类 |

* + 1. conf 包

代码清单 3-1 ConfigurationManager类

/\*\*

\* 配置工具类

\*/

object ConfigurationManager {

// 创建用于初始化配置生成器实例的参数对象

private val params = new Parameters()

// FileBasedConfigurationBuilder:产生一个传入的类的实例对象

// FileBasedConfiguration:融合 FileBased 与 Configuration 的接口

// PropertiesConfiguration:从一个或者多个文件读取配置的标准配置加载器

// configure():通过 params 实例初始化配置生成器

// 向 FileBasedConfigurationBuilder()中传入一个标准配置加载器类，生成一个加载器类的实例对象，然后通过 params 参数对其初始化

private val builder = new FileBasedConfigurationBuilder[FileBasedConfiguration](classOf [PropertiesConfiguration])

.configure(params.properties().setFileName("commerce.properties"))

// 通过 getConfiguration 获取配置对象

val config = builder.getConfiguration()

}

* + 1. constant 包

代码清单 3-2 Constants类

/\*\*

\* 常量接口

\*/

object Constants {

/\*\*

\* 项目配置相关的常量

\*/

val JDBC\_DATASOURCE\_SIZE = "jdbc.datasource.size" val JDBC\_URL = "jdbc.url"

val JDBC\_USER = "jdbc.user"

val JDBC\_PASSWORD = "jdbc.password"

val KAFKA\_TOPICS = "kafka.topics"

/\*\*

\* Spark 作业相关的常量

\*/

val SPARK\_APP\_NAME\_SESSION = "UserVisitSessionAnalyzeSpark" val SPARK\_APP\_NAME\_PAGE = "PageOneStepConvertRateSpark"

val FIELD\_SESSION\_ID = "sessionid"

val FIELD\_SEARCH\_KEYWORDS = "searchKeywords"

val FIELD\_CLICK\_CATEGORY\_IDS = "clickCategoryIds" val FIELD\_AGE = "age"

val FIELD\_PROFESSIONAL = "professional" val FIELD\_CITY = "city"

val FIELD\_SEX = "sex"

val FIELD\_VISIT\_LENGTH = "visitLength" val FIELD\_STEP\_LENGTH = "stepLength" val FIELD\_START\_TIME = "startTime"

val FIELD\_CLICK\_COUNT = "clickCount" val FIELD\_ORDER\_COUNT = "orderCount" val FIELD\_PAY\_COUNT = "payCount"

val FIELD\_CATEGORY\_ID = "categoryid" val SESSION\_COUNT = "session\_count"

val TIME\_PERIOD\_1s\_3s = "1s\_3s" val TIME\_PERIOD\_4s\_6s = "4s\_6s" val TIME\_PERIOD\_7s\_9s = "7s\_9s"

val TIME\_PERIOD\_10s\_30s = "10s\_30s" val TIME\_PERIOD\_30s\_60s = "30s\_60s" val TIME\_PERIOD\_1m\_3m = "1m\_3m"

val TIME\_PERIOD\_3m\_10m = "3m\_10m" val TIME\_PERIOD\_10m\_30m = "10m\_30m" val TIME\_PERIOD\_30m = "30m"

val STEP\_PERIOD\_1\_3 = "1\_3" val STEP\_PERIOD\_4\_6 = "4\_6" val STEP\_PERIOD\_7\_9 = "7\_9"

val STEP\_PERIOD\_10\_30 = "10\_30" val STEP\_PERIOD\_30\_60 = "30\_60" val STEP\_PERIOD\_60 = "60"

/\*\*

\* 任务相关的常量

\*/

val TASK\_PARAMS = "task.params.json" val PARAM\_START\_DATE = "startDate" val PARAM\_END\_DATE = "endDate"

val PARAM\_START\_AGE = "startAge" val PARAM\_END\_AGE = "endAge"

val PARAM\_PROFESSIONALS = "professionals" val PARAM\_CITIES = "cities"

val PARAM\_SEX = "sex"

val PARAM\_KEYWORDS = "keywords"

val PARAM\_CATEGORY\_IDS = "categoryIds"

val PARAM\_TARGET\_PAGE\_FLOW = "targetPageFlow"

}

* + 1. model 包

代码清单 3-3 DateModel类

/\*\*

* 用户访问动作表

\*

* @param date 用户点击行为的日期
* @param user\_id 用户的 ID
* @param session\_id Session 的 ID
* @param page\_id 某个页面的 ID
* @param action\_time 点击行为的时间点
* @param search\_keyword 用户搜索的关键词
* @param click\_category\_id 某一个商品品类的 ID
* @param click\_product\_id 某一个商品的 ID
* @param order\_category\_ids 一次订单中所有品类的 ID 集合
* @param order\_product\_ids 一次订单中所有商品的 ID 集合
* @param pay\_category\_ids 一次支付中所有品类的 ID 集合
* @param pay\_product\_ids 一次支付中所有商品的 ID 集合
* @param city\_id 城市 ID

\*/

case class UserVisitAction(date: String,

user\_id: Long, session\_id: String, page\_id: Long, action\_time: String, search\_keyword: String, click\_category\_id: Long, click\_product\_id: Long,

order\_category\_ids: String, order\_product\_ids: String, pay\_category\_ids: String, pay\_product\_ids: String, city\_id: Long

)

/\*\*

* 用户信息表

\*

* @param user\_id 用户的 ID
* @param username 用户的名称
* @param name 用户的名字
* @param age 用户的年龄
* @param professional 用户的职业
* @param city 用户所在的城市
* @param sex 用户的性别

\*/

case class UserInfo(user\_id: Long,

username: String, name: String, age: Int,

professional: String, city: String,

sex: String

)

/\*\*

* 产品表

\*

* @param product\_id 商品的 ID
* @param product\_name 商品的名称
* @param extend\_info 商品额外的信息

\*/

case class ProductInfo(product\_id: Long,

product\_name: String, extend\_info: String

)

##### utils 包

表 3-2 utils 包

|  |  |
| --- | --- |
| **类名称（class）** | **解析** |
| **DateUtils** | 时间工具类  负责时间的格式化、判断时间先后、计算时间差值、获取指定日期等工作 |
| **NumberUtils** | 数字工具类  负责数字的格式化工作 |
| **ParamUtils** | 参数工具类  负责从 JSON 对象中提取参数 |
| **StringUtils** | 字符串工具类  负责字符串是否为空判断、字符串截断与补全、从拼接字符串中提取字段、给拼接字符串中字段设置值等工作 |
| **ValidUtils** | 校验工具类  负责校验数据中的指定字段是否在指定范围范围内、校验数据中的指定字段中是否有值与参数字段相同、校验数据中的指定字段是否与参数字段相同等工作 |

* + 1. **mock 模块**

mock 模块负责产生模拟数据。

表 3-3 mock模块

|  |  |
| --- | --- |
| **Object** | **解析** |
| **MockDataGenerate** | 离线模拟数据生成  负责生成离线模拟数据并写入 Hive 表中，模拟数据包括用户行为信息、用户信息、产品数据信息等 |
| **MockRealtimeDataGenerate** | 实时模拟数据生成  负责生成实时模拟数据并写入 Kafka 中，实时模拟数据为实时广告数据 |

* + 1. **analyse 模块**

analyse 模块是需求的具体实现模块， 我们将会在下一章中进行详细解析。

**第四章 需求解析**

* 1. **需求一：Session 各范围访问步长、访问时长占比统计**
     1. **需求解析**

需求一要统计出符合筛选条件的 session 中，访问时长在 1s~3s、4s~6s、7s~9s、

10s~30s、30s~60s、1m~3m、3m~10m、10m~30m、30m 以上各个范围内的 session 占比；访问步长在 1~3、4~6、7~9、10~30、30~60、60 以上各个范围内的 session 占比，并将结果保存到 MySQL 数据库中。

在计算之前需要根据查询条件筛选 session，查询条件比如搜索过某些关键词的用户、访问时间在某个时间段内的用户、年龄在某个范围内的用户、职业在某个范围内的用户、所在某个城市的用户，发起的 session。找到对应的这些用户的 session， 并进行统计， 之所以需要有筛选主要是可以让使用者， 对感兴趣的和关系的用户群体，进行后续各种复杂业务逻辑的统计和分析，那么拿到的结果数据，就是只是针对特殊用户群体的分析结果；而不是对所有用户进行分析的泛泛的分析结果。比如说，现在某个企业高层，就是想看到用户群体中， 28~35 岁的，老师职业的群体， 对应的一些统计和分析的结果数据，从而辅助高管进行公司战略上的决策制定。

session 访问时长，也就是说一个 session 对应的开始的 action，到结束的 action，之间的时间范围；还有，就是访问步长，指的是，一个 session 执行期间内，依次点击过多少个页面，比如说，一次 session，维持了 1 分钟， 那么访问时长就是 1m， 然后在这 1 分钟内，点击了 10 个页面， 那么 session 的访问步长，就是 10.

比如说，符合第一步筛选出来的 session 的数量大概是有 1000 万个。那么里面， 我们要计算出，访问时长在 1s~3s 内的 session 的数量，并除以符合条件的总 session 数量（ 比如 1000 万），比如是 100 万/1000 万，那么 1s~3s 内的 session 占比就是 10%。依次类推，这里说的统计，就是这个意思。

这个功能可以让人从全局的角度看到，符合某些条件的用户群体，使用我们的产品的一些习惯。比如大多数人，到底是会在产品中停留多长时间， 大多数人，会在一次使用产品的过程中，访问多少个页面。那么对于使用者来说， 有一个全局和清晰的认识。

* + 1. **数据源解析**

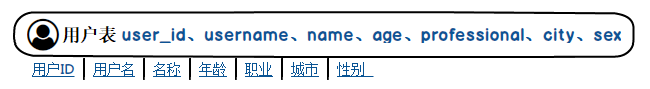


图 4-1 需求一数据源解析

* + 1. **数据结构解析**

##### UserVisitAction

/\*\*

* 用户访问动作表

\*

* @param date 用户点击行为的日期
* @param user\_id 用户的 ID
* @param session\_id Session 的 ID
* @param page\_id 某个页面的 ID
* @param action\_time 点击行为的时间点
* @param search\_keyword 用户搜索的关键词
* @param click\_category\_id 某一个商品品类的 ID
* @param click\_product\_id 某一个商品的 ID
* @param order\_category\_ids 一次订单中所有品类的 ID 集合
* @param order\_product\_ids 一次订单中所有商品的 ID 集合
* @param pay\_category\_ids 一次支付中所有品类的 ID 集合
* @param pay\_product\_ids 一次支付中所有商品的 ID 集合
* @param city\_id 城市 ID

\*/

case class UserVisitAction(date: String,

user\_id: Long, session\_id: String, page\_id: Long, action\_time: String, search\_keyword: String, click\_category\_id: Long, click\_product\_id: Long,

order\_category\_ids: String, order\_product\_ids: String, pay\_category\_ids: String, pay\_product\_ids: String, city\_id: Long

)

##### AggrInfo

SessionID | 搜索关键字 | 点击品类 | 访问时长 | 访问步长|开始时间

Session\_Id | Search\_Keywords | Click\_Category\_Id | Visit\_Length | Step\_Length | Start\_Time|Age|Professional|Sex|City

* + 1. **需求实现流程**

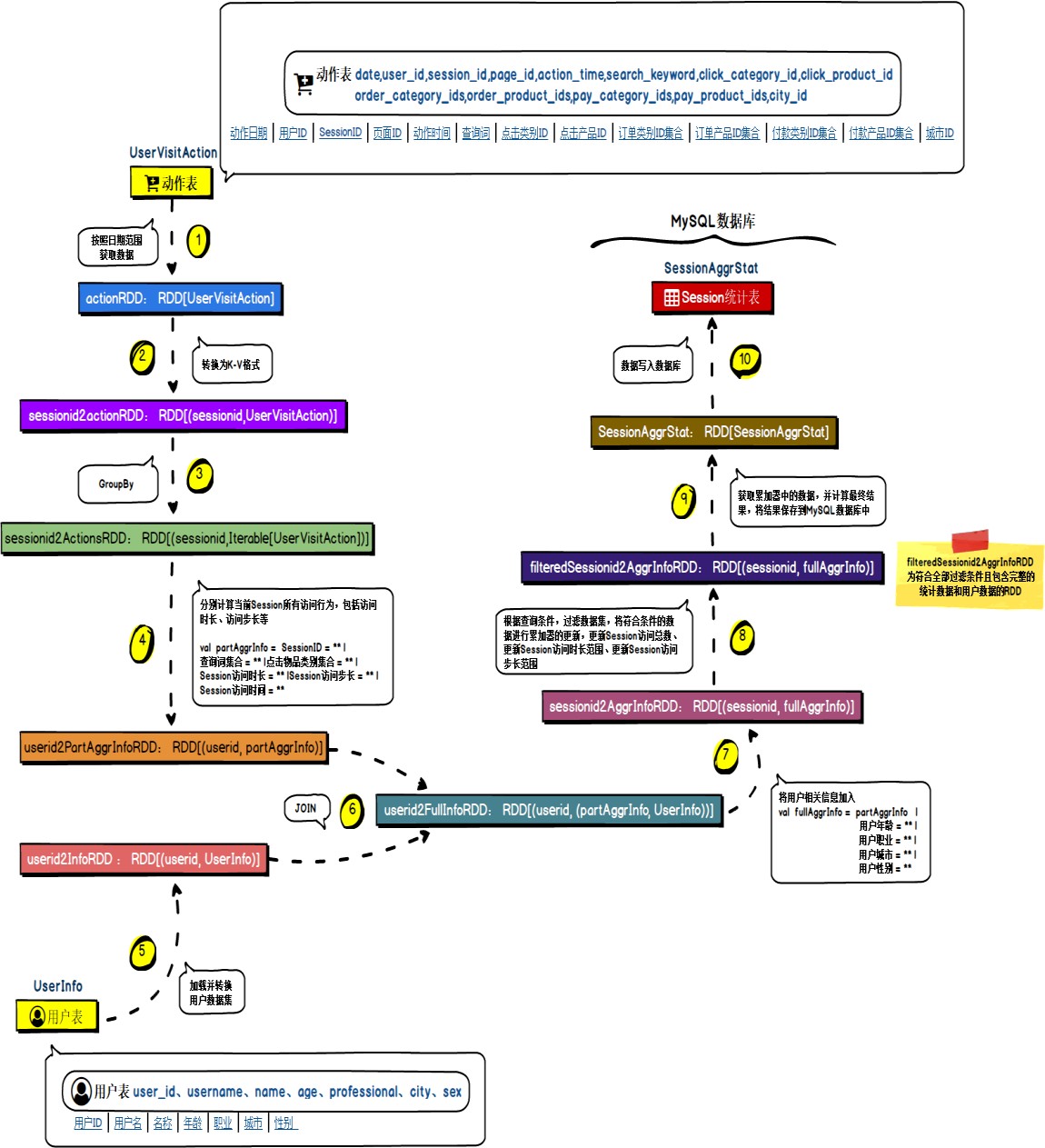


图 4-2 需求一实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `session\_aggr\_stat`

-- ----------------------------

DROP TABLE IF EXISTS `session\_aggr\_stat`; CREATE TABLE `session\_aggr\_stat` (

`taskid` varchar(255) DEFAULT NULL,

`session\_count` int(11) DEFAULT NULL,

`visit\_length\_1s\_3s\_ratio` double DEFAULT NULL,

`visit\_length\_4s\_6s\_ratio` double DEFAULT NULL,

`visit\_length\_7s\_9s\_ratio` double DEFAULT NULL,

`visit\_length\_10s\_30s\_ratio` double DEFAULT NULL,

`visit\_length\_30s\_60s\_ratio` double DEFAULT NULL,

`visit\_length\_1m\_3m\_ratio` double DEFAULT NULL,

`visit\_length\_3m\_10m\_ratio` double DEFAULT NULL,

`visit\_length\_10m\_30m\_ratio` double DEFAULT NULL,

`visit\_length\_30m\_ratio` double DEFAULT NULL,

`step\_length\_1\_3\_ratio` double DEFAULT NULL,

`step\_length\_4\_6\_ratio` double DEFAULT NULL,

`step\_length\_7\_9\_ratio` double DEFAULT NULL,

`step\_length\_10\_30\_ratio` double DEFAULT NULL,

`step\_length\_30\_60\_ratio` double DEFAULT NULL,

`step\_length\_60\_ratio` double DEFAULT NULL, KEY `idx\_task\_id` (`taskid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-1 需求一代码解析

/\*\*

* 根据日期获取对象的用户行为数据
* @param spark
* @param taskParam
* @return

\*/

def getActionRDDByDateRange(spark: SparkSession,

taskParam: JSONObject): RDD[UserVisitAction] = { val startDate = ParamUtils.getParam(taskParam, Constants.PARAM\_START\_DATE) val endDate = ParamUtils.getParam(taskParam, Constants.PARAM\_END\_DATE)

import spark.implicits.\_

spark.sql("select \* from user\_visit\_action where date>='" + startDate + "' and date<='"

+ endDate + "'").as[UserVisitAction].rdd

}

/\*\*

* + 对 Session 数据进行聚合
  + @param spark
  + @param sessinoid2actionRDD
  + @return

\*/

def aggregateBySession(spark: SparkSession, sessinoid2actionRDD: RDD[(String, UserVisitAction)]): RDD[(String, String)] = {

// 对行为数据按 session 粒度进行分组

val sessionid2ActionsRDD = sessinoid2actionRDD.groupByKey()

// 对每一个 session 分组进行聚合，将 session 中所有的搜索词和点击品类都聚合起来，

<userid,partAggrInfo(sessionid,searchKeywords,clickCategoryIds)>

val userid2PartAggrInfoRDD = sessionid2ActionsRDD.map { case (sessionid, userVisitActions) =>

val searchKeywordsBuffer = new StringBuffer("") val clickCategoryIdsBuffer = new StringBuffer("")

var userid = -1L

// session 的起始和结束时间var startTime: Date = null var endTime: Date = null

// session 的访问步长

**———————————————**

**尚硅谷大数据项目之电商分析平台**

var stepLength = 0

**——————————————**

// 遍历 session 所有的访问行为

userVisitActions.foreach { userVisitAction => if (userid == -1L) {

userid = userVisitAction.user\_id

}

val searchKeyword = userVisitAction.search\_keyword

val clickCategoryId = userVisitAction.click\_category\_id

// 实际上这里要对数据说明一下

// 并不是每一行访问行为都有 searchKeyword 何 clickCategoryId 两个字段的

// 其实，只有搜索行为，是有 searchKeyword 字段的

// 只有点击品类的行为，是有 clickCategoryId 字段的

// 所以，任何一行行为数据，都不可能两个字段都有，所以数据是可能出现 null 值的

// 我们决定是否将搜索词或点击品类 id 拼接到字符串中去

// 首先要满足：不能是 null 值

// 其次，之前的字符串中还没有搜索词或者点击品类 id

if (StringUtils.isNotEmpty(searchKeyword)) {

if (!searchKeywordsBuffer.toString.contains(searchKeyword)) { searchKeywordsBuffer.append(searchKeyword + ",")

}

}

if (clickCategoryId != null && clickCategoryId != -1L) {

if (!clickCategoryIdsBuffer.toString.contains(clickCategoryId.toString)) { clickCategoryIdsBuffer.append(clickCategoryId + ",")

}

}

// 计算 session 开始和结束时间

val actionTime = DateUtils.parseTime(userVisitAction.action\_time)

if (startTime == null) { startTime = actionTime

}

if (endTime == null) { endTime = actionTime

}

if (actionTime.before(startTime)) { startTime = actionTime

}

if (actionTime.after(endTime)) { endTime = actionTime

}

// 计算 session 访问步长

stepLength += 1

}

val searchKeywords = StringUtils.trimComma(searchKeywordsBuffer.toString)

val clickCategoryIds = StringUtils.trimComma(clickCategoryIdsBuffer.toString)

// 计算 session 访问时长（秒）

val visitLength = (endTime.getTime() - startTime.getTime()) / 1000

// 聚合数据，使用 key=value|key=value

val partAggrInfo = Constants.FIELD\_SESSION\_ID + "=" + sessionid + "|" +

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

Constants.FIELD\_SEARCH\_KEYWORDS + "=" + searchKeywords + "|" + Constants.FIELD\_CLICK\_CATEGORY\_IDS + "=" + clickCategoryIds + "|" + Constants.FIELD\_VISIT\_LENGTH + "=" + visitLength + "|" + Constants.FIELD\_STEP\_LENGTH + "=" + stepLength + "|" + Constants.FIELD\_START\_TIME + "=" + DateUtils.formatTime(startTime)

(userid, partAggrInfo);

}

// 查询所有用户数据，并映射成<userid,Row>的格式

import spark.implicits.\_

val userid2InfoRDD = spark.sql("select \* from user\_info").as[UserInfo].rdd.map(item => (item.user\_id, item))

// 将 session 粒度聚合数据，与用户信息进行 join

val userid2FullInfoRDD = userid2PartAggrInfoRDD.join(userid2InfoRDD);

// 对 join 起来的数据进行拼接，并且返回<sessionid,fullAggrInfo>格式的数据

val sessionid2FullAggrInfoRDD = userid2FullInfoRDD.map { case (uid, (partAggrInfo, userInfo)) =>

val sessionid = StringUtils.getFieldFromConcatString(partAggrInfo, "\\|", Constants.FIELD\_SESSION\_ID)

val fullAggrInfo = partAggrInfo + "|" + Constants.FIELD\_AGE + "=" + userInfo.age + "|" +

Constants.FIELD\_PROFESSIONAL + "=" + userInfo.professional + "|" + Constants.FIELD\_CITY + "=" + userInfo.city + "|" + Constants.FIELD\_SEX + "=" + userInfo.sex

(sessionid, fullAggrInfo)

}

sessionid2FullAggrInfoRDD

}

/\*\*

* 业务需求一：过滤 session 数据，并进行聚合统计
* @param sessionid2AggrInfoRDD
* @return

\*/

def filterSessionAndAggrStat(sessionid2AggrInfoRDD: RDD[(String, String)],

taskParam: JSONObject,

sessionAggrStatAccumulator: AccumulatorV2[String, mutable.HashMap[String, Int]]): RDD[(String, String)] = {

// 获取查询任务中的配置

val startAge = ParamUtils.getParam(taskParam, Constants.PARAM\_START\_AGE) val endAge = ParamUtils.getParam(taskParam, Constants.PARAM\_END\_AGE)

val professionals = ParamUtils.getParam(taskParam, Constants.PARAM\_PROFESSIONALS) val cities = ParamUtils.getParam(taskParam, Constants.PARAM\_CITIES)

val sex = ParamUtils.getParam(taskParam, Constants.PARAM\_SEX)

val keywords = ParamUtils.getParam(taskParam, Constants.PARAM\_KEYWORDS)

val categoryIds = ParamUtils.getParam(taskParam, Constants.PARAM\_CATEGORY\_IDS)

var \_parameter = (if (startAge != null) Constants.PARAM\_START\_AGE + "=" + startAge + "|" else "") +

(if (endAge != null) Constants.PARAM\_END\_AGE + "=" + endAge + "|" else "") +

(if (professionals != null) Constants.PARAM\_PROFESSIONALS + "=" + professionals + "|" else "") +

(if (cities != null) Constants.PARAM\_CITIES + "=" + cities + "|" else "") +

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

(if (sex != null) Constants.PARAM\_SEX + "=" + sex + "|" else "") +

(if (keywords != null) Constants.PARAM\_KEYWORDS + "=" + keywords + "|" else "") + (if (categoryIds != null) Constants.PARAM\_CATEGORY\_IDS + "=" + categoryIds else "")

if (\_parameter.endsWith("\\|")) {

\_parameter = \_parameter.substring(0, \_parameter.length() - 1)

}

val parameter = \_parameter

// 根据筛选参数进行过滤

val filteredSessionid2AggrInfoRDD = sessionid2AggrInfoRDD.filter { case (sessionid, aggrInfo) =>

// 接着，依次按照筛选条件进行过滤

// 按照年龄范围进行过滤（startAge、endAge）

var success = true

if (!ValidUtils.between(aggrInfo, Constants.FIELD\_AGE, parameter, Constants.PARAM\_START\_AGE, Constants.PARAM\_END\_AGE))

success = false

// 按照职业范围进行过滤（professionals）

// 互联网,IT,软件

// 互联网

if (!ValidUtils.in(aggrInfo, Constants.FIELD\_PROFESSIONAL, parameter, Constants.PARAM\_PROFESSIONALS))

success = false

// 按照城市范围进行过滤（cities）

// 北京,上海,广州,深圳

// 成都

if (!ValidUtils.in(aggrInfo, Constants.FIELD\_CITY, parameter, Constants.PARAM\_CITIES))

success = false

// 按照性别进行过滤

// 男/女

// 男，女

if (!ValidUtils.equal(aggrInfo, Constants.FIELD\_SEX, parameter, Constants.PARAM\_SEX)) success = false

// 按照搜索词进行过滤

// 我们的 session 可能搜索了 火锅,蛋糕,烧烤

// 我们的筛选条件可能是 火锅,串串香,iphone 手机

// 那么，in 这个校验方法，主要判定 session 搜索的词中，有任何一个，与筛选条件中

// 任何一个搜索词相当，即通过

if (!ValidUtils.in(aggrInfo, Constants.FIELD\_SEARCH\_KEYWORDS, parameter, Constants.PARAM\_KEYWORDS))

success = false

// 按照点击品类 id 进行过滤

if (!ValidUtils.in(aggrInfo, Constants.FIELD\_CLICK\_CATEGORY\_IDS, parameter, Constants.PARAM\_CATEGORY\_IDS))

success = false

// 如果符合任务搜索需求

if (success) { sessionAggrStatAccumulator.add(Constants.SESSION\_COUNT);

// 计算访问时长范围

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

def calculateVisitLength(visitLength: Long) { if (visitLength >= 1 && visitLength <= 3) {

sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_1s\_3s);

} else if (visitLength >= 4 && visitLength <= 6) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_4s\_6s);

} else if (visitLength >= 7 && visitLength <= 9) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_7s\_9s);

} else if (visitLength >= 10 && visitLength <= 30) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_10s\_30s);

} else if (visitLength > 30 && visitLength <= 60) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_30s\_60s);

} else if (visitLength > 60 && visitLength <= 180) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_1m\_3m);

} else if (visitLength > 180 && visitLength <= 600) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_3m\_10m);

} else if (visitLength > 600 && visitLength <= 1800) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_10m\_30m);

} else if (visitLength > 1800) { sessionAggrStatAccumulator.add(Constants.TIME\_PERIOD\_30m);

}

}

// 计算访问步长范围

def calculateStepLength(stepLength: Long) { if (stepLength >= 1 && stepLength <= 3) {

sessionAggrStatAccumulator.add(Constants.STEP\_PERIOD\_1\_3);

} else if (stepLength >= 4 && stepLength <= 6) { sessionAggrStatAccumulator.add(Constants.STEP\_PERIOD\_4\_6);

} else if (stepLength >= 7 && stepLength <= 9) { sessionAggrStatAccumulator.add(Constants.STEP\_PERIOD\_7\_9);

} else if (stepLength >= 10 && stepLength <= 30) { sessionAggrStatAccumulator.add(Constants.STEP\_PERIOD\_10\_30);

} else if (stepLength > 30 && stepLength <= 60) { sessionAggrStatAccumulator.add(Constants.STEP\_PERIOD\_30\_60);

} else if (stepLength > 60) { sessionAggrStatAccumulator.add(Constants.STEP\_PERIOD\_60);

}

}

// 计算出 session 的访问时长和访问步长的范围，并进行相应的累加

val visitLength = StringUtils.getFieldFromConcatString(aggrInfo, "\\|",

Constants.FIELD\_VISIT\_LENGTH).toLong

val stepLength = StringUtils.getFieldFromConcatString(aggrInfo, "\\|",

Constants.FIELD\_STEP\_LENGTH).toLong calculateVisitLength(visitLength) calculateStepLength(stepLength)

}

success

}

filteredSessionid2AggrInfoRDD

}

/\*\*

* 获取通过筛选条件的 session 的访问明细数据 RDD

\*

* @param sessionid2aggrInfoRDD
* @param sessionid2actionRDD
* @return

\*/

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

def getSessionid2detailRDD(sessionid2aggrInfoRDD: RDD[(String, String)], sessionid2actionRDD: RDD[(String, UserVisitAction)]): RDD[(String, UserVisitAction)] = {

sessionid2aggrInfoRDD.join(sessionid2actionRDD).map(item => (item.\_1, item.\_2.\_2))

}

/\*\*

* 计算各 session 范围占比，并写入 MySQL
* @param value

\*/

def calculateAndPersistAggrStat(spark: SparkSession, value: mutable.HashMap[String, Int], taskUUID: String) {

// 从 Accumulator 统计串中获取值

val session\_count = value(Constants.SESSION\_COUNT).toDouble

val visit\_length\_1s\_3s = value.getOrElse(Constants.TIME\_PERIOD\_1s\_3s, 0) val visit\_length\_4s\_6s = value.getOrElse(Constants.TIME\_PERIOD\_4s\_6s, 0) val visit\_length\_7s\_9s = value.getOrElse(Constants.TIME\_PERIOD\_7s\_9s, 0)

val visit\_length\_10s\_30s = value.getOrElse(Constants.TIME\_PERIOD\_10s\_30s, 0) val visit\_length\_30s\_60s = value.getOrElse(Constants.TIME\_PERIOD\_30s\_60s, 0) val visit\_length\_1m\_3m = value.getOrElse(Constants.TIME\_PERIOD\_1m\_3m, 0)

val visit\_length\_3m\_10m = value.getOrElse(Constants.TIME\_PERIOD\_3m\_10m, 0) val visit\_length\_10m\_30m = value.getOrElse(Constants.TIME\_PERIOD\_10m\_30m, 0) val visit\_length\_30m = value.getOrElse(Constants.TIME\_PERIOD\_30m, 0)

val step\_length\_1\_3 = value.getOrElse(Constants.STEP\_PERIOD\_1\_3, 0) val step\_length\_4\_6 = value.getOrElse(Constants.STEP\_PERIOD\_4\_6, 0) val step\_length\_7\_9 = value.getOrElse(Constants.STEP\_PERIOD\_7\_9, 0)

val step\_length\_10\_30 = value.getOrElse(Constants.STEP\_PERIOD\_10\_30, 0) val step\_length\_30\_60 = value.getOrElse(Constants.STEP\_PERIOD\_30\_60, 0) val step\_length\_60 = value.getOrElse(Constants.STEP\_PERIOD\_60, 0)

// 计算各个访问时长和访问步长的范围

val visit\_length\_1s\_3s\_ratio = NumberUtils.formatDouble(visit\_length\_1s\_3s / session\_co unt, 2)

val visit\_length\_4s\_6s\_ratio = NumberUtils.formatDouble(visit\_length\_4s\_6s / session\_co unt, 2)

val visit\_length\_7s\_9s\_ratio = NumberUtils.formatDouble(visit\_length\_7s\_9s / session\_co unt, 2)

val visit\_length\_10s\_30s\_ratio = NumberUtils.formatDouble(visit\_length\_10s\_30s / sessio n\_count, 2)

val visit\_length\_30s\_60s\_ratio = NumberUtils.formatDouble(visit\_length\_30s\_60s / sessio n\_count, 2)

val visit\_length\_1m\_3m\_ratio = NumberUtils.formatDouble(visit\_length\_1m\_3m / sessio n\_count, 2)

val visit\_length\_3m\_10m\_ratio = NumberUtils.formatDouble(visit\_length\_3m\_10m / sessio n\_count, 2)

val visit\_length\_10m\_30m\_ratio = NumberUtils.formatDouble(visit\_length\_10m\_30m / sessio n\_count, 2)

val visit\_length\_30m\_ratio = NumberUtils.formatDouble(visit\_length\_30m / session\_count, 2)

val step\_length\_1\_3\_ratio = NumberUtils.formatDouble(step\_length\_1\_3 / session\_count, 2)

val step\_length\_4\_6\_ratio = NumberUtils.formatDouble(step\_length\_4\_6 / session\_count, 2)

val step\_length\_7\_9\_ratio = NumberUtils.formatDouble(step\_length\_7\_9 / session\_count, 2)

val step\_length\_10\_30\_ratio = NumberUtils.formatDouble(step\_length\_10\_30 / session\_cou nt, 2)

val step\_length\_30\_60\_ratio = NumberUtils.formatDouble(step\_length\_30\_60 / session\_cou

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

nt, 2)

val step\_length\_60\_ratio = NumberUtils.formatDouble(step\_length\_60 / session\_count, 2)

// 将统计结果封装为 Domain 对象

val sessionAggrStat = SessionAggrStat(taskUUID,

session\_count.toInt,

visit\_length\_7s\_9s\_ratio,

visit\_length\_1s\_3s\_ratio,

visit\_length\_4s\_6s\_ratio,

visit\_length\_10s\_30s\_ratio, visit\_length\_30s\_60s\_ratio, visit\_length\_1m\_3m\_ratio, visit\_length\_3m\_10m\_ratio, visit\_length\_10m\_30m\_ratio, visit\_length\_30m\_ratio, step\_length\_1\_3\_ratio, step\_length\_4\_6\_ratio, step\_length\_7\_9\_ratio, step\_length\_10\_30\_ratio, step\_length\_30\_60\_ratio, step\_length\_60\_ratio)

import spark.implicits.\_

val sessionAggrStatRDD = spark.sparkContext.makeRDD(Array(sessionAggrStat)) sessionAggrStatRDD.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "session\_aggr\_stat")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

}

代码清单 4-2 SessionAggrStat类

/\*\*

* 聚合统计表

\*

* @param taskid 当前计算批次的 ID
* @param session\_count 所有 Session 的总和
* @param visit\_length\_1s\_3s\_ratio 1-3sSession 访问时长占比
* @param visit\_length\_4s\_6s\_ratio 4-6sSession 访问时长占比
* @param visit\_length\_7s\_9s\_ratio 7-9sSession 访问时长占比
* @param visit\_length\_10s\_30s\_ratio 10-30sSession 访问时长占比
* @param visit\_length\_30s\_60s\_ratio 30-60sSession 访问时长占比
* @param visit\_length\_1m\_3m\_ratio 1-3mSession 访问时长占比
* @param visit\_length\_3m\_10m\_ratio 3-10mSession 访问时长占比
* @param visit\_length\_10m\_30m\_ratio 10-30mSession 访问时长占比
* @param visit\_length\_30m\_ratio 30mSession 访问时长占比
* @param step\_length\_1\_3\_ratio 1-3 步长占比
* @param step\_length\_4\_6\_ratio 4-6 步长占比
* @param step\_length\_7\_9\_ratio 7-9 步长占比
* @param step\_length\_10\_30\_ratio 10-30 步长占比
* @param step\_length\_30\_60\_ratio 30-60 步长占比
* @param step\_length\_60\_ratio 大于 60 步长占比

\*/

case class SessionAggrStat(taskid: String,

session\_count: Long, visit\_length\_1s\_3s\_ratio: Double, visit\_length\_4s\_6s\_ratio: Double, visit\_length\_7s\_9s\_ratio: Double, visit\_length\_10s\_30s\_ratio: Double, visit\_length\_30s\_60s\_ratio: Double, visit\_length\_1m\_3m\_ratio: Double, visit\_length\_3m\_10m\_ratio: Double, visit\_length\_10m\_30m\_ratio: Double, visit\_length\_30m\_ratio: Double, step\_length\_1\_3\_ratio: Double, step\_length\_4\_6\_ratio: Double, step\_length\_7\_9\_ratio: Double,

**———————————————**

**尚硅谷大数据项目之电商分析平台**

### ——————————————

step\_length\_10\_30\_ratio: Double, step\_length\_30\_60\_ratio: Double, step\_length\_60\_ratio: Double

)

**4.2 需求二：Session 随机抽取**

**4.2.1 需求解析**

###### 在符合条件的 session 中，按照时间比例随机抽取 1000 个 session

这个按照时间比例是什么意思呢？随机抽取本身是很简单的，但是按照时间比例，就很复杂了。比如说，这一天总共有 1000 万的 session。那么我现在总共要从这 1000 万 session 中，随机抽取出来 1000 个 session。但是这个随机不是那么简单的。需要做到如下几点要求：首先，如果这一天的 12:00~13:00 的 session 数量是 100 万，那么这个小时的 session 占比就是 1/10，那么这个小时中的 100 万的 session， 我们就要抽取 1/10 \* 1000 = 100 个。然后再从这个小时的 100 万 session 中，随机抽取出 100 个 session。以此类推，其他小时的抽取也是这样做。

这个功能的作用，是说，可以让使用者，能够对于符合条件的 session，按照时间比例均匀的随机采样出 1000 个 session，然后观察每个 session 具体的点击流/行为， 比如先进入了首页、然后点击了食品品类、然后点击了雨润火腿肠商品、然后搜索 了火腿肠罐头的关键词、接着对王中王火腿肠下了订单、最后对订单做了支付。

之所以要做到按时间比例随机采用抽取，就是要做到，观察样本的公平性。

抽取完毕之后，需要将 Session 的相关信息和详细信息保存到 MySQL 数据库中。

**4.2.1 数据源解析**

本需求的数据源来自于需求一中获取的的 Session 聚合数据（ AggrInfo ） 和

Session 用户访问数据（UserVisitAction） 。

* + 1. **数据结构解析**

1. UserVisitAction

/\*\*

* 用户访问动作表

\*

* @param date 用户点击行为的日期
* @param user\_id 用户的 ID
* @param session\_id Session 的 ID
* @param page\_id 某个页面的 ID
* @param action\_time 点击行为的时间点
* @param search\_keyword 用户搜索的关键词
* @param click\_category\_id 某一个商品品类的 ID
* @param click\_product\_id 某一个商品的 ID
* @param order\_category\_ids 一次订单中所有品类的 ID 集合
* @param order\_product\_ids 一次订单中所有商品的 ID 集合
* @param pay\_category\_ids 一次支付中所有品类的 ID 集合
* @param pay\_product\_ids 一次支付中所有商品的 ID 集合
* @param city\_id 城市 ID

\*/

case class UserVisitAction(date: String,

user\_id: Long, session\_id: String, page\_id: Long, action\_time: String, search\_keyword: String, click\_category\_id: Long, click\_product\_id: Long,

order\_category\_ids: String, order\_product\_ids: String, pay\_category\_ids: String, pay\_product\_ids: String, city\_id: Long

)

##### AggrInfo

SessionID | 搜索关键字 | 点击品类 | 访问时长 | 访问步长|开始时间

Session\_Id | Search\_Keywords | Click\_Category\_Id | Visit\_Length | Step\_Length | Start\_Time

* + 1. **需求实现流程**

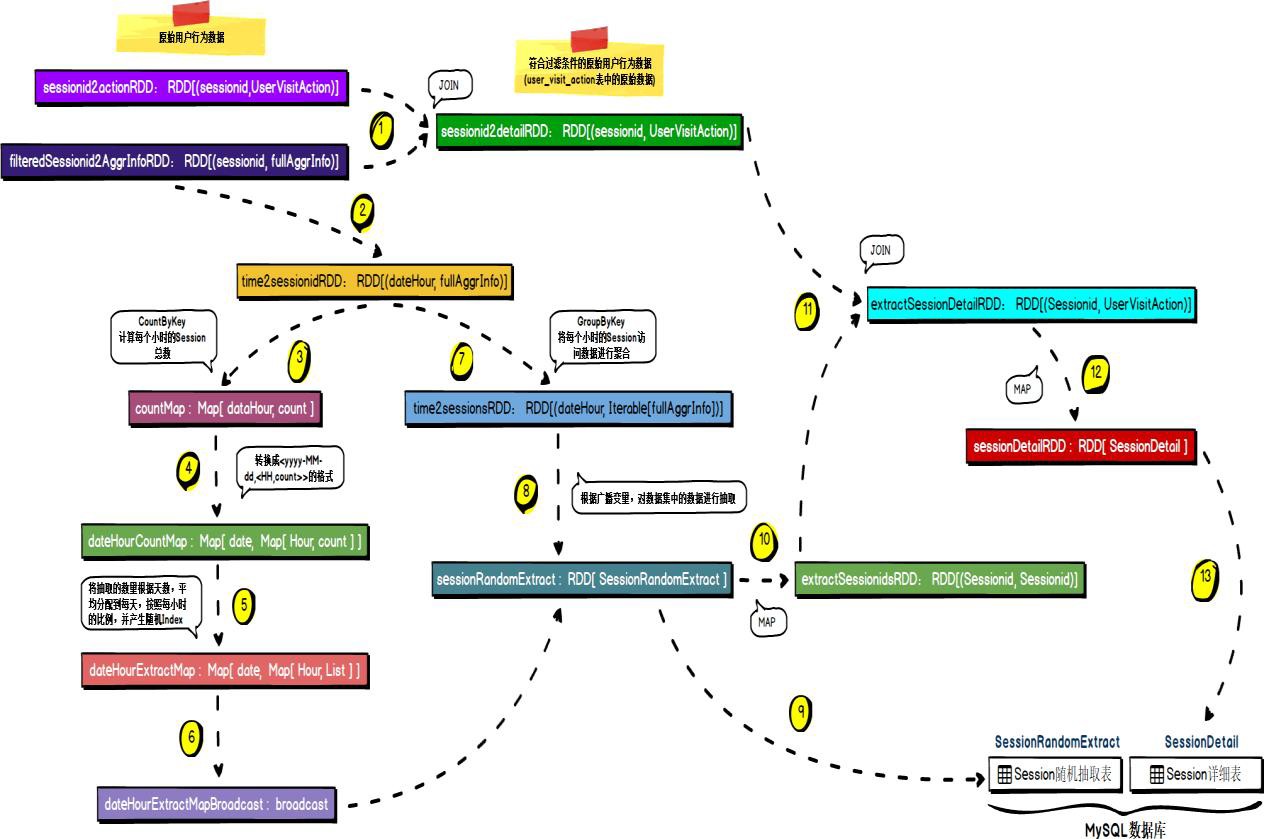


图 4-3 需求二实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `session\_detail`

-- ----------------------------

DROP TABLE IF EXISTS `session\_detail`; CREATE TABLE `session\_detail` (

`taskid` varchar(255) DEFAULT NULL,

`userid` int(11) DEFAULT NULL,

`sessionid` varchar(255) DEFAULT NULL,

`pageid` int(11) DEFAULT NULL,

`actionTime` varchar(255) DEFAULT NULL,

`searchKeyword` varchar(255) DEFAULT NULL,

`clickCategoryId` int(11) DEFAULT NULL,

`clickProductId` int(11) DEFAULT NULL,

`orderCategoryIds` varchar(255) DEFAULT NULL,

`orderProductIds` varchar(255) DEFAULT NULL,

`payCategoryIds` varchar(255) DEFAULT NULL,

`payProductIds` varchar(255) DEFAULT NULL, KEY `idx\_task\_id` (`taskid`),

KEY `idx\_session\_id` (`sessionid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------

-- Table structure for `session\_random\_extract`

-- ----------------------------

DROP TABLE IF EXISTS `session\_random\_extract`; CREATE TABLE `session\_random\_extract` (

`taskid` varchar(255) DEFAULT NULL,

`sessionid` varchar(255) DEFAULT NULL,

`startTime` varchar(50) DEFAULT NULL,

`searchKeywords` varchar(255) DEFAULT NULL,

`clickCategoryIds` varchar(255) DEFAULT NULL, KEY `idx\_task\_id` (`taskid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-3 需求二代码解析

/\*\*

* 业务需求二：随机抽取 session
* @param sessionid2AggrInfoRDD

\*/

def randomExtractSession(spark: SparkSession, taskUUID: String, sessionid2AggrInfoRDD: RDD[(String, String)], sessionid2actionRDD: RDD[(String, UserVisitAction)]) {

// 第一步，计算出每天每小时的 session 数量，获取<yyyy-MM-dd\_HH,aggrInfo>格式的 RDD

val time2sessionidRDD = sessionid2AggrInfoRDD.map { case (sessionid, aggrInfo) => val startTime = StringUtils.getFieldFromConcatString(aggrInfo, "\\|", Constants.

FIELD\_START\_TIME)

// 将 key 改为 yyyy-MM-dd\_HH 的形式（小时粒度）

val dateHour = DateUtils.getDateHour(startTime) (dateHour, aggrInfo)

}

// 得到每天每小时的 session 数量

// countByKey()计算每个不同的 key 有多少个数据

// countMap<yyyy-MM-dd\_HH, count>

val countMap = time2sessionidRDD.countByKey()

// 第二步，使用按时间 比例随机抽 取算法， 计算 出每天每小 时要抽取 session 的索引，将

<yyyy-MM-dd\_HH,count>格式的 map，转换成<yyyy-MM-dd,<HH,count>>的格式

// dateHourCountMap <yyyy-MM-dd,<HH,count>>

val dateHourCountMap = mutable.HashMap[String, mutable.HashMap[String, Long]]() for ((dateHour, count) <- countMap) {

val date = dateHour.split("\_")(0) val hour = dateHour.split("\_")(1)

// 通过模式匹配实现了 if 的功能

dateHourCountMap.get(date) match {

// 对应日期的数据不存在，则新增

case None => dateHourCountMap(date) = new mutable.HashMap[String, Long](); dateHourCountMap(date) += (hour -> count)

// 对应日期的数据存在，则累加

// 如果有值，Some(hourCountMap)将值取到了 hourCountMap 中

case Some(hourCountMap) => hourCountMap += (hour -> count)

}

}

// 按时间比例随机抽取算法，总共要抽取 100 个 session，先按照天数，进行平分

// 获取每一天要抽取的数量

val extractNumberPerDay = 100 / dateHourCountMap.size

// dateHourExtractMap[天，[小时，index 列表]]

val dateHourExtractMap = mutable.HashMap[String, mutable.HashMap[String, mutable.ListBuffer[Int]]]()

val random = new Random()

/\*\*

* 根据每个小时应该抽取的数量，来产生随机值
* 遍历每个小时，填充 Map<date,<hour,(3,5,20,102)>>
* @param hourExtractMap 主要用来存放生成的随机值
* @param hourCountMap 每个小时的 session 总数
* @param sessionCount 当天所有的 seesion 总数

\*/

def hourExtractMapFunc(hourExtractMap: mutable.HashMap[String, mutable.ListBuffer[Int]], hourCountMap: mutable.HashMap[String, Long], sessionCount: Long)

{

for ((hour, count) <- hourCountMap) {

// 计算每个小时的 session 数量，占据当天总 session 数量的比例，直接乘以每天要抽取的数量

// 就可以计算出，当前小时需要抽取的 session 数量

var hourExtractNumber = ((count / sessionCount.toDouble) \* extractNumberPerDay).toInt if (hourExtractNumber > count) {

hourExtractNumber = count.toInt

}

// 仍然通过模式匹配实现有则追加，无则新建

hourExtractMap.get(hour) match {

case None => hourExtractMap(hour) = new mutable.ListBuffer[Int]();

// 根据数量随机生成下标

for (i <- 0 to hourExtractNumber) {

var extractIndex = random.nextInt(count.toInt);

// 一旦随机生成的 index 已经存在，重新获取，直到获取到之前没有的 index while (hourExtractMap(hour).contains(extractIndex)) {

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

extractIndex = random.nextInt(count.toInt);

}

hourExtractMap(hour) += (extractIndex)

}

case Some(extractIndexList) =>

for (i <- 0 to hourExtractNumber) {

var extractIndex = random.nextInt(count.toInt);

// 一旦随机生成的 index 已经存在，重新获取，直到获取到之前没有的 index while (hourExtractMap(hour).contains(extractIndex)) {

extractIndex = random.nextInt(count.toInt);

}

hourExtractMap(hour) += (extractIndex)

}

}

}

}

// session 随机抽取功能

for ((date, hourCountMap) <- dateHourCountMap) {

// 计算出这一天的 session 总数

val sessionCount = hourCountMap.values.sum

// dateHourExtractMap[天，[小时，小时列表]] dateHourExtractMap.get(date) match {

case None => dateHourExtractMap(date) = new mutable.HashMap[String, mutable.ListBuffer[Int]]();

// 更新 index

hourExtractMapFunc(dateHourExtractMap(date), hourCountMap, sessionCount)

case Some(hourExtractMap) => hourExtractMapFunc(hourExtractMap, hourCountMap, sessionCount)

}

}

/\* 至此，index 获取完毕 \*/

//将 Map 进行广播

val dateHourExtractMapBroadcast = spark.sparkContext.broadcast(dateHourExtractMap)

// time2sessionidRDD <yyyy-MM-dd\_HH,aggrInfo>

// 执行 groupByKey 算子，得到<yyyy-MM-dd\_HH,(session aggrInfo)> val time2sessionsRDD = time2sessionidRDD.groupByKey()

// 第三步：遍历每天每小时的 session，然后根据随机索引进行抽取,我们用 flatMap 算子，遍历所有的

<dateHour,(session aggrInfo)>格式的数据

val sessionRandomExtract = time2sessionsRDD.flatMap { case (dateHour, items) => val date = dateHour.split("\_")(0)

val hour = dateHour.split("\_")(1)

// 从广播变量中提取出数据

val dateHourExtractMap = dateHourExtractMapBroadcast.value

// 获取指定天对应的指定小时的 indexList

// 当前小时需要的 index 集合

val extractIndexList = dateHourExtractMap.get(date).get(hour)

// index 是在外部进行维护

var index = 0

val sessionRandomExtractArray = new ArrayBuffer[SessionRandomExtract]()

// 开始遍历所有的 aggrInfo

for (sessionAggrInfo <- items) {

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

// 如果筛选 List 中包含当前的 index，则提取此 sessionAggrInfo 中的数据

if (extractIndexList.contains(index)) {

val sessionid = StringUtils.getFieldFromConcatString(sessionAggrInfo, "\\|", Constants.FIELD\_SESSION\_ID)

val starttime = StringUtils.getFieldFromConcatString(sessionAggrInfo, "\\|", Constants.FIELD\_START\_TIME)

val searchKeywords = StringUtils.getFieldFromConcatString(sessionAggrInfo, "\\|", Constants.FIELD\_SEARCH\_KEYWORDS)

val clickCategoryIds = StringUtils.getFieldFromConcatString(sessionAggrInfo, "\\|", Constants.FIELD\_CLICK\_CATEGORY\_IDS)

sessionRandomExtractArray += SessionRandomExtract(taskUUID, sessionid, starttime, searchKeywords, clickCategoryIds)

}

// index 自增

index += 1

}

sessionRandomExtractArray

}

/\* 将抽取后的数据保存到 MySQL \*/

// 引入隐式转换，准备进行 RDD 向 Dataframe 的转换

import spark.implicits.\_

// 为了方便地将数据保存到 MySQL 数据库，将 RDD 数据转换为 Dataframe sessionRandomExtract.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "session\_random\_extract")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

// 提取抽取出来的数据中的 sessionId

val extractSessionidsRDD = sessionRandomExtract.map(item => (item.sessionid, item.sessionid))

// 第四步：获取抽取出来的 session 的明细数据

// 根据 sessionId 与详细数据进行聚合

val extractSessionDetailRDD = extractSessionidsRDD.join(sessionid2actionRDD)

// 对 extractSessionDetailRDD 中的数据进行聚合，提炼有价值的明细数据

val sessionDetailRDD = extractSessionDetailRDD.map { case (sid, (sessionid, userVisitAction)) =>

SessionDetail(taskUUID, userVisitAction.user\_id, userVisitAction.session\_id, userVisitAction.page\_id, userVisitAction.action\_time,

userVisitAction.search\_keyword,

userVisitAction.click\_category\_id, userVisitAction.click\_product\_id, userVisitAction.order\_category\_ids,

userVisitAction.order\_product\_ids, userVisitAction.pay\_category\_ids, userVisitAction.pay\_product\_ids)

}

// 将明细数据保存到 MySQL 中

sessionDetailRDD.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "session\_detail")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

}

代码清单 4-4 SessionRandomExtract类

/\*\*

* Session 随机抽取表

\*

* @param taskid 当前计算批次的 ID
* @param sessionid 抽取的 Session 的 ID
* @param startTime Session 的开始时间
* @param searchKeywords Session 的查询字段
* @param clickCategoryIds Session 点击的类别 id 集合

\*/

case class SessionRandomExtract(taskid:String,

sessionid:String, startTime:String, searchKeywords:String,

clickCategoryIds:String)

代码清单 4-5 SessionDetail类

/\*\*

* Session 随机抽取详细表

\*

* @param taskid 当前计算批次的 ID
* @param userid 用户的 ID
* @param sessionid Session 的 ID
* @param pageid 某个页面的 ID
* @param actionTime 点击行为的时间点
* @param searchKeyword 用户搜索的关键词
* @param clickCategoryId 某一个商品品类的 ID
* @param clickProductId 某一个商品的 ID
* @param orderCategoryIds 一次订单中所有品类的 ID 集合
* @param orderProductIds 一次订单中所有商品的 ID 集合
* @param payCategoryIds 一次支付中所有品类的 ID 集合
* @param payProductIds 一次支付中所有商品的 ID 集合

\*\*/

case class SessionDetail(taskid:String,

userid:Long, sessionid:String, pageid:Long, actionTime:String, searchKeyword:String, clickCategoryId:Long, clickProductId:Long, orderCategoryIds:String, orderProductIds:String, payCategoryIds:String,

payProductIds:String)

* 1. **需求三：Top10 热门品类**
     1. **需求解析**

在符合条件的 session 中，获取点击、下单和支付数量排名前 10 的品类。

数据中的每个 session 可能都会对一些品类的商品进行点击、下单和支付等等行

为， 那么现在就需要获取这些 session 点击、下单和支付数量排名前 10 的最热门的品类。也就是说，要计算出所有这些 session 对各个品类的点击、下单和支付的次数， 然后按照这三个属性进行排序，获取前 10 个品类。

这个功能，很重要，就可以让我们明白， 就是符合条件的用户， 他最感兴趣的商品是什么种类。这个可以让公司里的人， 清晰地了解到不同层次、不同类型的用户的心理和喜好。

计算完成之后， 将数据保存到 MySQL 数据库中。

* + 1. **数据源解析**

本需求的数据源来 自于需求一中获取 的 Session 用 户 访 问 数 据

（ UserVisitAction）。

* + 1. **数据结构解析**

/\*\*

* 用户访问动作表

\*

* @param date 用户点击行为的日期
* @param user\_id 用户的 ID
* @param session\_id Session 的 ID
* @param page\_id 某个页面的 ID
* @param action\_time 点击行为的时间点
* @param search\_keyword 用户搜索的关键词
* @param click\_category\_id 某一个商品品类的 ID
* @param click\_product\_id 某一个商品的 ID
* @param order\_category\_ids 一次订单中所有品类的 ID 集合
* @param order\_product\_ids 一次订单中所有商品的 ID 集合
* @param pay\_category\_ids 一次支付中所有品类的 ID 集合
* @param pay\_product\_ids 一次支付中所有商品的 ID 集合
* @param city\_id 城市 ID

\*/

case class UserVisitAction(date: String,

user\_id: Long, session\_id: String, page\_id: Long, action\_time: String, search\_keyword: String, click\_category\_id: Long, click\_product\_id: Long,

order\_category\_ids: String, order\_product\_ids: String, pay\_category\_ids: String, pay\_product\_ids: String, city\_id: Long

)

## 需求实现流程

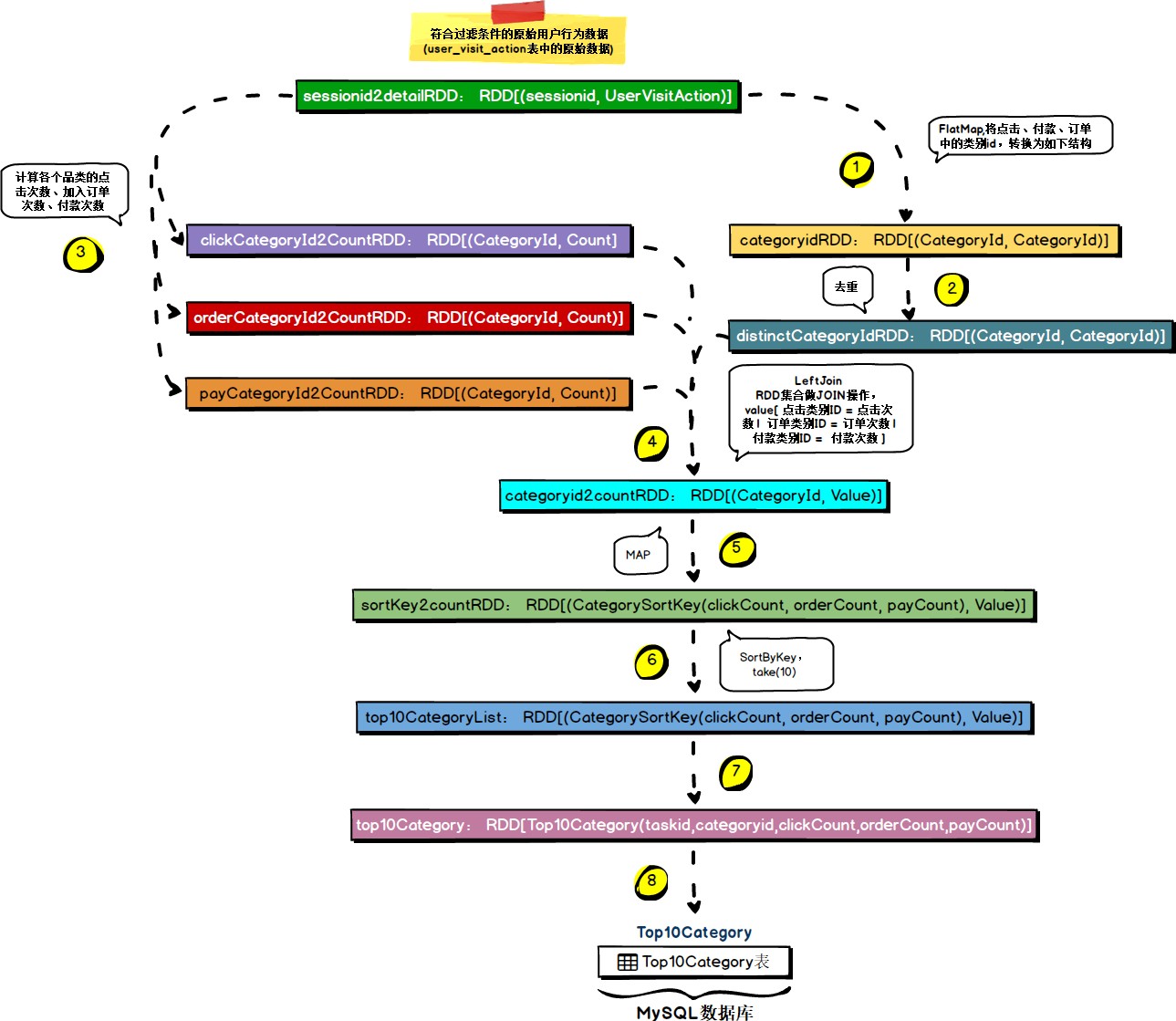


图 4-4 需求三实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `top10\_category`

-- ----------------------------

DROP TABLE IF EXISTS `top10\_category`; CREATE TABLE `top10\_category` (

`taskid` varchar(255) DEFAULT NULL,

`categoryid` int(11) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL,

`orderCount` int(11) DEFAULT NULL,

`payCount` int(11) DEFAULT NULL, KEY `idx\_task\_id` (`taskid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-6 需求三代码解析

/\*\*

* + - * 业务需求三：获取 top10 热门品类
      * @param spark
      * @param taskid
      * @param sessionid2detailRDD
      * @return

\*/

def getTop10Category(spark: SparkSession, taskid: String, sessionid2detailRDD: RDD[(String, UserVisitAction)]): Array[(CategorySortKey, String)] = {

// 第一步：获取每一个 Sessionid 点击过、下单过、支付过的数量

// 获取所有产生过点击、下单、支付中任意行为的商品类别

val categoryidRDD = sessionid2detailRDD.flatMap { case (sessionid, userVisitAction) => val list = ArrayBuffer[(Long, Long)]()

// 一个 session 中点击的商品 ID

if (userVisitAction.click\_category\_id != null) {

list += ((userVisitAction.click\_category\_id, userVisitAction.click\_category\_id))

}

// 一个 session 中下单的商品 ID 集合

if (userVisitAction.order\_category\_ids != null) {

for (orderCategoryId <- userVisitAction.order\_category\_ids.split(",")) list += ((orderCategoryId.toLong, orderCategoryId.toLong))

}

// 一个 session 中支付的商品 ID 集合

if (userVisitAction.pay\_category\_ids != null) {

for (payCategoryId <- userVisitAction.pay\_category\_ids.split(",")) list += ((payCategoryId.toLong, payCategoryId.toLong))

}

list

}

// 对重复的 categoryid 进行去重

// 得到了所有被点击、下单、支付的商品的品类

val distinctCategoryIdRDD = categoryidRDD.distinct

// 第二步：计算各品类的点击、下单和支付的次数

// 计算各个品类的点击次数

val clickCategoryId2CountRDD = getClickCategoryId2CountRDD(sessionid2detailRDD)

// 计算各个品类的下单次数

val orderCategoryId2CountRDD = getOrderCategoryId2CountRDD(sessionid2detailRDD)

// 计算各个品类的支付次数

val payCategoryId2CountRDD = getPayCategoryId2CountRDD(sessionid2detailRDD)

// 第三步：join 各品类与它的点击、下单和支付的次数

// distinctCategoryIdRDD 中是所有产生过点击、下单、支付行为的商品类别

// 通过 distinctCategoryIdRDD 与各个统计数据的 LeftJoin 保证数据的完整性

val categoryid2countRDD = joinCategoryAndData(distinctCategoryIdRDD, clickCategoryId2CountRDD, orderCategoryId2CountRDD, payCategoryId2CountRDD);

// 第四步：自定义二次排序 key

// 第五步：将数据映射成<CategorySortKey,info>格式的 RDD，然后进行二次排序（降序）

// 创建用于二次排序的联合 key —— (CategorySortKey(clickCount, orderCount, payCount), line)

// 按照：点击次数 -> 下单次数 -> 支付次数 这一顺序进行二次排序

val sortKey2countRDD = categoryid2countRDD.map { case (categoryid, line) =>

val clickCount = StringUtils.getFieldFromConcatString(line, "\\|",

Constants.FIELD\_CLICK\_COUNT).toLong

val orderCount = StringUtils.getFieldFromConcatString(line, "\\|",

Constants.FIELD\_ORDER\_COUNT).toLong

val payCount = StringUtils.getFieldFromConcatString(line, "\\|",

Constants.FIELD\_PAY\_COUNT).toLong (CategorySortKey(clickCount, orderCount, payCount), line)

}

// 降序排序

val sortedCategoryCountRDD = sortKey2countRDD.sortByKey(false)

// 第六步：用 take(10)取出 top10 热门品类，并写入 MySQL

val top10CategoryList = sortedCategoryCountRDD.take(10)

val top10Category = top10CategoryList.map { case (categorySortKey, line) =>

val categoryid = StringUtils.getFieldFromConcatString(line, "\\|",

Constants.FIELD\_CATEGORY\_ID).toLong

val clickCount = StringUtils.getFieldFromConcatString(line, "\\|",

Constants.FIELD\_CLICK\_COUNT).toLong

val orderCount = StringUtils.getFieldFromConcatString(line, "\\|",

Constants.FIELD\_ORDER\_COUNT).toLong

val payCount = StringUtils.getFieldFromConcatString(line, "\\|", Constants.FIELD\_PAY\_COUNT).toLong

Top10Category(taskid, categoryid, clickCount, orderCount, payCount)

}

// 将 Map 结构转化为 RDD

val top10CategoryRDD = spark.sparkContext.makeRDD(top10Category)

// 写入 MySQL 之前，将 RDD 转化为 Dataframe import spark.implicits.\_ top10CategoryRDD.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "top10\_category")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

top10CategoryList

}

/\*\*

* + - * 连接品类 RDD 与数据 RDD

\*

* + - * @param categoryidRDD
      * @param clickCategoryId2CountRDD
      * @param orderCategoryId2CountRDD
      * @param payCategoryId2CountRDD
      * @return

\*/

def joinCategoryAndData(categoryidRDD: RDD[(Long, Long)], clickCategoryId2CountRDD: RDD[(Long, Long)], orderCategoryId2CountRDD: RDD[(Long, Long)], payCategoryId2CountRDD: RDD[(Long, Long)]): RDD[(Long, String)] = {

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

// 将所有品类信息与点击次数信息结合【左连接】

val clickJoinRDD = categoryidRDD.leftOuterJoin(clickCategoryId2CountRDD).map { case (categoryid, (cid, optionValue)) =>

val clickCount = if (optionValue.isDefined) optionValue.get else 0L

val value = Constants.FIELD\_CATEGORY\_ID + "=" + categoryid + "|" +

Constants.FIELD\_CLICK\_COUNT + "=" + clickCount (categoryid, value)

}

// 将所有品类信息与订单次数信息结合【左连接】

val orderJoinRDD = clickJoinRDD.leftOuterJoin(orderCategoryId2CountRDD).map { case (categoryid, (ovalue, optionValue)) =>

val orderCount = if (optionValue.isDefined) optionValue.get else 0L

val value = ovalue + "|" + Constants.FIELD\_ORDER\_COUNT + "=" + orderCount (categoryid, value)

}

// 将所有品类信息与付款次数信息结合【左连接】

val payJoinRDD = orderJoinRDD.leftOuterJoin(payCategoryId2CountRDD).map { case (categoryid, (ovalue, optionValue)) =>

val payCount = if (optionValue.isDefined) optionValue.get else 0L val value = ovalue + "|" + Constants.FIELD\_PAY\_COUNT + "=" + payCount (categoryid, value)

}

payJoinRDD

}

/\*\*

* + - * 获取各个品类的支付次数 RDD

\*

* + - * @param sessionid2detailRDD
      * @return

\*/

def getPayCategoryId2CountRDD(sessionid2detailRDD: RDD[(String, UserVisitAction)]): RDD[(Long, Long)] = {

// 过滤支付数据

val payActionRDD = sessionid2detailRDD.filter { case (sessionid, userVisitAction) => userVisitAction.pay\_category\_ids != null }

// 获取每种类别的支付次数

val payCategoryIdRDD = payActionRDD.flatMap { case (sessionid, userVisitAction) => userVisitAction.pay\_category\_ids.split(",").map(item => (item.toLong, 1L)) }

// 计算各个品类的支付次数

payCategoryIdRDD.reduceByKey(\_ + \_)

}

/\*\*

* + - * 获取各品类的下单次数 RDD

\*

* + - * @param sessionid2detailRDD
      * @return

\*/

def getOrderCategoryId2CountRDD(sessionid2detailRDD: RDD[(String, UserVisitAction)]): RDD[(Long, Long)] = {

// 过滤订单数据

val orderActionRDD = sessionid2detailRDD.filter { case (sessionid, userVisitAction) => userVisitAction.order\_category\_ids != null }

// 获取每种类别的下单次数

val orderCategoryIdRDD = orderActionRDD.flatMap { case (sessionid, userVisitAction) => userVisitAction.order\_category\_ids.split(",").map(item => (item.toLong, 1L)) }

// 计算各个品类的下单次数

orderCategoryIdRDD.reduceByKey(\_ + \_)

}

/\*\*

* 获取各个品类的支付次数 RDD

\*

* @param sessionid2detailRDD
* @return

\*/

def getPayCategoryId2CountRDD(sessionid2detailRDD: RDD[(String, UserVisitAction)]): RDD[(Long, Long)] = {

// 过滤支付数据

val payActionRDD = sessionid2detailRDD.filter { case (sessionid, userVisitAction) => userVisitAction.pay\_category\_ids != null }

// 获取每种类别的支付次数

val payCategoryIdRDD = payActionRDD.flatMap { case (sessionid, userVisitAction) => userVisitAction.pay\_category\_ids.split(",").map(item => (item.toLong, 1L)) }

// 计算各个品类的支付次数

payCategoryIdRDD.reduceByKey(\_ + \_)

}

代码清单 4-7 CategorySortKey类

case class CategorySortKey(val clickCount: Long, val orderCount: Long, val payCount: Long) extends Ordered[CategorySortKey] {

/\*\* Result of comparing `this` with operand `that`.

\*

* Implement this method to determine how instances of A will be sorted.

\*

* Returns `x` where:

\*

\*

\*

\*

\*

\*

\*

\*/

- `x < 0` when `this < that`

* `x == 0` when `this == that`
* `x > 0` when `this > that`

override def compare(that: CategorySortKey): Int = { if (this.clickCount - that.clickCount != 0) {

return (this.clickCount - that.clickCount).toInt

} else if (this.orderCount - that.orderCount != 0) { return (this.orderCount - that.orderCount).toInt

} else if (this.payCount - that.payCount != 0) { return (this.payCount - that.payCount).toInt

} 0

}

}

代码清单 4-8 Top10Category类

/\*\*

* 品类 Top10 表
* @param taskid
* @param categoryid
* @param clickCount
* @param orderCount
* @param payCount

\*/

case class Top10Category(taskid:String,

categoryid:Long, clickCount:Long, orderCount:Long,

payCount:Long)

* 1. **需求四：Top10 热门品类 Top10 活跃 Session 统计**
     1. **需求解析**

对于排名前 10 的品类，分别获取其点击次数排名前 10 的 session。

这个就是说， 对于 top10 的品类， 每一个都要获取对它点击次数排名前 10 的

session。

这个功能，可以让我们看到，对某个用户群体最感兴趣的品类， 各个品类最感兴趣最典型的用户的 session 的行为。

计算完成之后，将数据保存到 MySQL 数据库中。

* + 1. **数据源解析**

本需求的数据源来 自于需求一中获取 的 Session 用 户 访 问 数 据

（ UserVisitAction）。

* + 1. **数据结构解析**

/\*\*

* 用户访问动作表

\*

* @param date 用户点击行为的日期
* @param user\_id 用户的 ID
* @param session\_id Session 的 ID
* @param page\_id 某个页面的 ID
* @param action\_time 点击行为的时间点
* @param search\_keyword 用户搜索的关键词
* @param click\_category\_id 某一个商品品类的 ID
* @param click\_product\_id 某一个商品的 ID
* @param order\_category\_ids 一次订单中所有品类的 ID 集合
* @param order\_product\_ids 一次订单中所有商品的 ID 集合
* @param pay\_category\_ids 一次支付中所有品类的 ID 集合
* @param pay\_product\_ids 一次支付中所有商品的 ID 集合
* @param city\_id 城市 ID

\*/

case class UserVisitAction(date: String,

user\_id: Long, session\_id: String, page\_id: Long, action\_time: String, search\_keyword: String, click\_category\_id: Long, click\_product\_id: Long,

order\_category\_ids: String, order\_product\_ids: String, pay\_category\_ids: String, pay\_product\_ids: String, city\_id: Long

)

## 需求实现流程

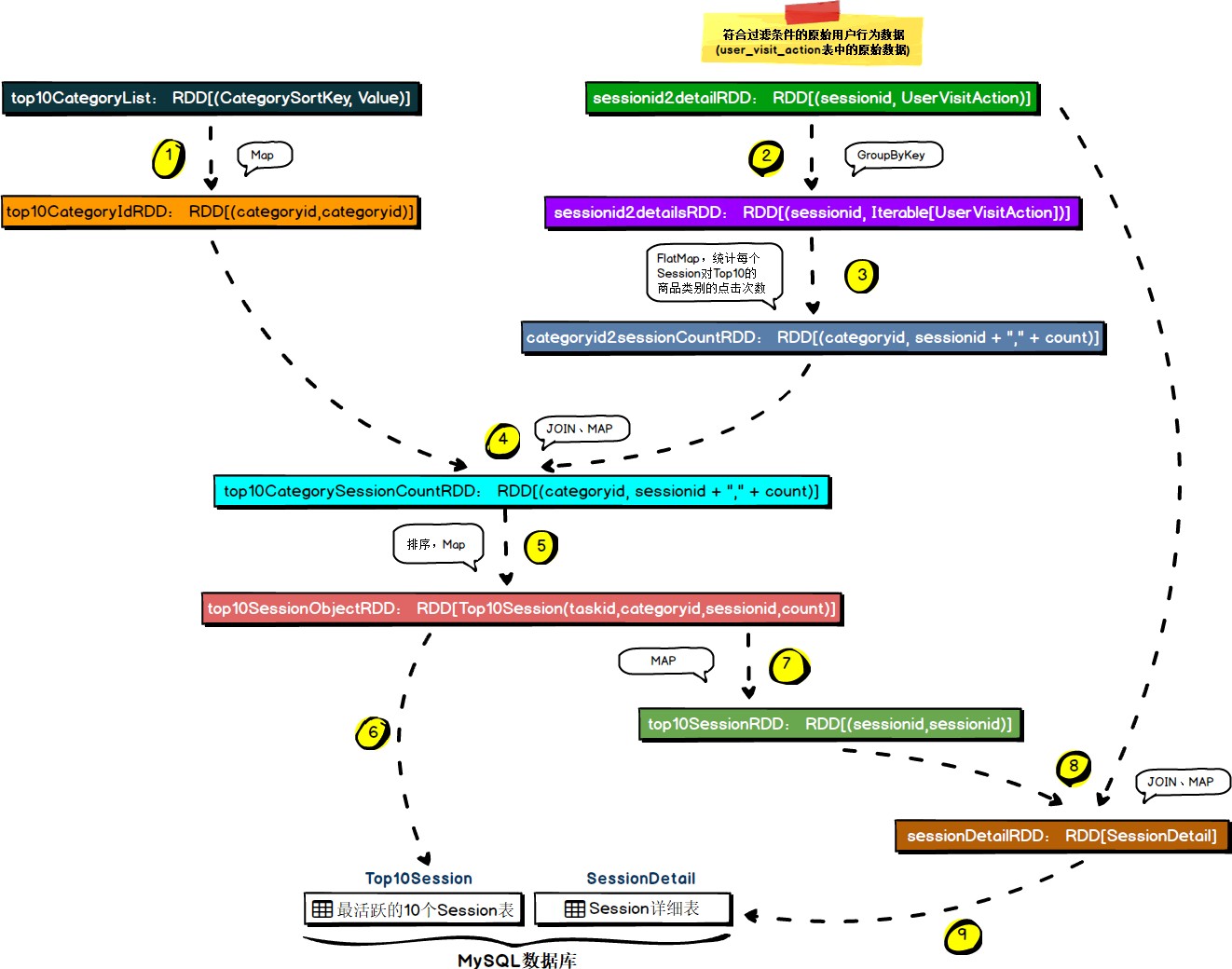


图 4-5 需求四实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `top10\_session`

-- ----------------------------

DROP TABLE IF EXISTS `top10\_session`; CREATE TABLE `top10\_session` (

`taskid` varchar(255) DEFAULT NULL,

`categoryid` int(11) DEFAULT NULL,

`sessionid` varchar(255) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL, KEY `idx\_task\_id` (`taskid`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

/\*\*

代码清单 4-9 需求四代码解析

* + - * 业务功能四：获取 top10 热门品类的活跃 session

\*

* + - * @param taskid

\*/

def getTop10Session(spark: SparkSession, taskid: String, top10CategoryList: Array[(CategorySortKey, String)], sessionid2ActionRDD: RDD[(String, UserVisitAction)]) {

// 第一步：将 top10 热门品类的 id，生成一份 RDD

// 获得所有需要求的 category 集合

val top10CategoryIdRDD = spark.sparkContext.makeRDD(top10CategoryList.map { case (categorySortKey, line) =>

val categoryid = StringUtils.getFieldFromConcatString(line, "\\|", Constants.FIELD\_ CATEGORY\_ID).toLong;

(categoryid, categoryid)

})

// 第二步：计算 top10 品类被各 session 点击的次数

// sessionid2ActionRDD 是符合过滤(职业、年龄等)条件的完整数据

// sessionid2detailRDD ( sessionId, userAction )

val sessionid2ActionsRDD = sessionid2ActionRDD.groupByKey()

// 获取每个品类被每一个 Session 点击的次数

val categoryid2sessionCountRDD = sessionid2ActionsRDD.flatMap { case (sessionid, userVisitActions) =>

val categoryCountMap = new mutable.HashMap[Long, Long]()

// userVisitActions 中聚合了一个 session 的所有用户行为数据

// 遍历 userVisitActions 是提取 session 中的每一个用户行为，并对每一个用户行为中的点击事件进行计

数

for (userVisitAction <- userVisitActions) {

// 如果 categoryCountMap 中尚不存在此点击品类，则新增品类

if (!categoryCountMap.contains(userVisitAction.click\_category\_id)) categoryCountMap.put(userVisitAction.click\_category\_id, 0)

// 如果 categoryCountMap 中已经存在此点击品类，则进行累加

if (userVisitAction.click\_category\_id != null && userVisitAction.click\_category\_id !=

-1L) {

categoryCountMap.update(userVisitAction.click\_category\_id, categoryCountMap(userVisitAction.click\_category\_id) + 1)

}

}

// 对 categoryCountMap 中的数据进行格式转化

for ((categoryid, count) <- categoryCountMap) yield (categoryid, sessionid + "," + count)

}

// 通过 top10 热门品类 top10CategoryIdRDD 与完整品类点击统计 categoryid2sessionCountRDD 进行

join，仅获取热门品类的数据信息

// 获取到 to10 热门品类，被各个 session 点击的次数【将数据集缩小】

val top10CategorySessionCountRDD = top10CategoryIdRDD.join(categoryid2sessionCountRDD).map { case (cid, (ccid, value)) => (cid, value) }

// 第三步：分组取 TopN 算法实现，获取每个品类的 top10 活跃用户

// 先按照品类分组

val top10CategorySessionCountsRDD = top10CategorySessionCountRDD.groupByKey()

// 将每一个品类的所有点击排序，取前十个，并转换为对象

val top10SessionObjectRDD = top10CategorySessionCountsRDD.flatMap { case (categoryid, clicks) =>

// 先排序，然后取前 10

val top10Sessions = clicks.toList.sortWith(\_.split(",")(1) > \_.split(",")(1)).take(10)

// 重新整理数据

top10Sessions.map { case line => val sessionid = line.split(",")(0)

val count = line.split(",")(1).toLong Top10Session(taskid, categoryid, sessionid, count)

}

}

// 将结果以追加方式写入到 MySQL 中import spark.implicits.\_ top10SessionObjectRDD.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "top10\_session")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

val top10SessionRDD = top10SessionObjectRDD.map(item => (item.sessionid, item.sessionid))

// 第四步：获取 top10 活跃 session 的明细数据

val sessionDetailRDD = top10SessionRDD.join(sessionid2ActionRDD).map { case (sid, (sessionid, userVisitAction)) =>

SessionDetail(taskid, userVisitAction.user\_id, userVisitAction.session\_id, userVisitAction.page\_id, userVisitAction.action\_time,

userVisitAction.search\_keyword,

userVisitAction.click\_category\_id, userVisitAction.click\_product\_id, userVisitAction.order\_category\_ids,

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

userVisitAction.order\_product\_ids, userVisitAction.pay\_product\_ids)

}

userVisitAction.pay\_category\_ids,

// 将活跃 Session 的明细数据，写入到 MySQL

sessionDetailRDD.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "session\_detail")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

}

代码清单 4-10 Top10Session类

/\*\*

* Top10 Session
* @param taskid
* @param categoryid
* @param sessionid
* @param clickCount

\*/

case class Top10Session(taskid:String,

categoryid:Long, sessionid:String,

clickCount:Long)

* 1. **需求五：页面转化率统计**
     1. **需求解析**

计算页面单跳转化率， 什么是页面单跳转换率， 比如一个用户在一次 Session 过程中访问的页面路径 3,5,7,9,10,21，那么页面 3 跳到页面 5 叫一次单跳，7-9 也叫一次单跳，那么单跳转化率就是要统计页面点击的概率，比如： 计算 3-5 的单跳转化率，先获取符合条件的 Session 对于页面 3 的访问次数（PV）为 A，然后获取符合条件的 Session 中访问了页面 3 又紧接着访问了页面 5 的次数为 B，那么 B/A 就是 3-5 的页面单跳转化率，我们记为 C；那么页面 5-7 的转化率怎么求呢？先需要求出符合条件的 Session 中访问页面 5 又紧接着访问了页面 7 的次数为 D，那么 D/B 即为 5-7 的单跳转化率。

产品经理，可以根据这个指标，去尝试分析， 整个网站，产品， 各个页面的表现怎么样，是不是需要去优化产品的布局；吸引用户最终可以进入最后的支付页面。

数据分析师，可以此数据做更深一步的计算和分析。

企业管理层， 可以看到整个公司的网站， 各个页面的之间的跳转的表现如何， 可以适当调整公司的经营战略或策略。

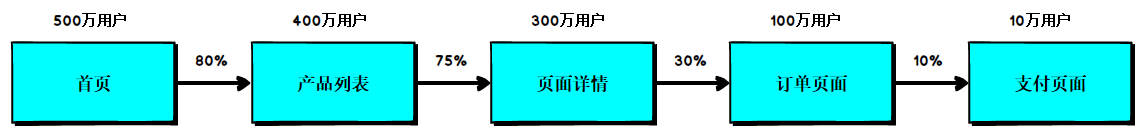


图 4-6 页面转化率分析

在该模块中，需要根据查询对象中设置的 Session 过滤条件，先将对应得 Session 过滤出来，然后根据查询对象中设置的页面路径， 计算页面单跳转化率， 比如查询的页面路径为：3、5、7、8，那么就要计算 3-5、5-7、7-8 的页面单跳转化率。需要注意的一点是， 页面的访问时有先后的。

* + 1. **数据源解析**



图 4-7 需求五数据源

* + 1. **数据结构解析**

/\*\*

* 用户访问动作表

\*

* @param date 用户点击行为的日期
* @param user\_id 用户的 ID
* @param session\_id Session 的 ID
* @param page\_id 某个页面的 ID
* @param action\_time 点击行为的时间点
* @param search\_keyword 用户搜索的关键词
* @param click\_category\_id 某一个商品品类的 ID
* @param click\_product\_id 某一个商品的 ID
* @param order\_category\_ids 一次订单中所有品类的 ID 集合
* @param order\_product\_ids 一次订单中所有商品的 ID 集合
* @param pay\_category\_ids 一次支付中所有品类的 ID 集合
* @param pay\_product\_ids 一次支付中所有商品的 ID 集合
* @param city\_id 城市 ID

\*/

case class UserVisitAction(date: String,

user\_id: Long, session\_id: String, page\_id: Long, action\_time: String, search\_keyword: String, click\_category\_id: Long, click\_product\_id: Long,

order\_category\_ids: String, order\_product\_ids: String, pay\_category\_ids: String, pay\_product\_ids: String, city\_id: Long

)

## 需求实现流程

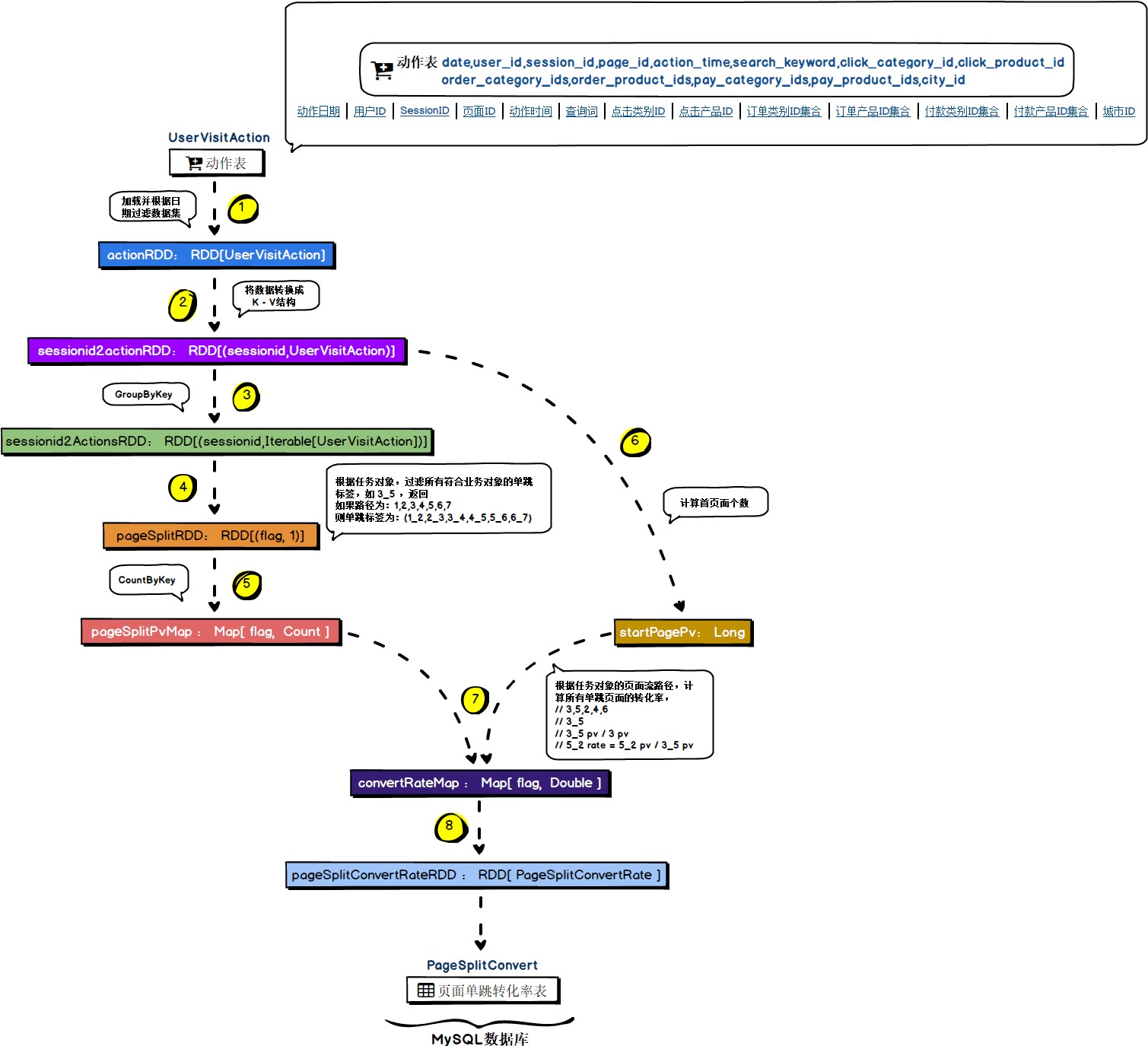


图 4-8 需求五实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `page\_split\_convert\_rate`

-- ----------------------------

DROP TABLE IF EXISTS `page\_split\_convert\_rate`; CREATE TABLE `page\_split\_convert\_rate` (

`taskid` varchar(255) DEFAULT NULL,

`convertRate` varchar(255) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-11 需求五代码解析

/\*\*

* 页面单跳转化率模块 spark 作业

\*

* 页面转化率的求解思路是通过 UserAction 表获取一个 session 的所有 UserAction，根据时间顺序排序后获取全部 PageId
* 然后将PageId 组合成PageFlow，即1,2,3,4,5 的形式（按照时间顺序排列），之后，组合为1\_2, 2\_3, 3\_4, ...

的形式

* 然后筛选出出现在 targetFlow 中的所有 A\_B

\*

* 对每个 A\_B 进行数量统计，然后统计 startPage 的 PV，之后根据 targetFlow 的 A\_B 顺序，计算每一层的转化率

\*

\*/

object PageOneStepConvertRate {

def main(args: Array[String]): Unit = {

// 获取统计任务参数【为了方便，直接从配置文件中获取，企业中会从一个调度平台获取】

val jsonStr = ConfigurationManager.config.getString(Constants.TASK\_PARAMS) val taskParam = JSONObject.fromObject(jsonStr)

// 任务的执行 ID，用户唯一标示运行后的结果，用在 MySQL 数据库中

val taskUUID = UUID.randomUUID().toString

// 构建 Spark 上下文

val sparkConf = new SparkConf().setAppName("SessionAnalyzer").setMaster("local[\*]")

// 创建 Spark 客户端

val

spark

=

SparkSession.builder().config(sparkConf).enableHiveSupport().getOrCreate() val sc = spark.sparkContext

// 查询指定日期范围内的用户访问行为数据

val actionRDD = this.getActionRDDByDateRange(spark, taskParam)

// 将用户行为信息转换为 K-V 结构

val sessionid2actionRDD = actionRDD.map(item => (item.session\_id, item))

// 将数据进行内存缓存

sessionid2actionRDD.persist(StorageLevel.MEMORY\_ONLY)

// 对<sessionid,访问行为> RDD，做一次 groupByKey 操作，生成页面切片

val sessionid2actionsRDD = sessionid2actionRDD.groupByKey()

// 最核心的一步，每个 session 的单跳页面切片的生成，以及页面流的匹配，算法

val pageSplitRDD = generateAndMatchPageSplit(sc, sessionid2actionsRDD, taskParam)

// 返回：(1\_2, 1)，(3\_4, 1), ..., (100\_101, 1)

// 统计每个跳转切片的总个数

// pageSplitPvMap：(1\_2, 102320), (3\_4, 90021), ..., (100\_101, 45789)

val pageSplitPvMap = pageSplitRDD.countByKey

// 使用者指定的页面流是 3,2,5,8,6

// 咱们现在拿到的这个 pageSplitPvMap，3->2，2->5，5->8，8->6

// 首先计算首页 PV 的数量

val startPagePv = getStartPagePv(taskParam, sessionid2actionsRDD)

// 计算目标页面流的各个页面切片的转化率

val convertRateMap = computePageSplitConvertRate(taskParam, pageSplitPvMap, startPagePv)

// 持久化页面切片转化率

persistConvertRate(spark, taskUUID, convertRateMap)

spark.close()

}

/\*\*

* 持久化转化率
* @param convertRateMap

\*/

def persistConvertRate(spark:SparkSession, taskid:String, convertRateMap:collection.Map[String, Double]) {

val convertRate = convertRateMap.map(item => item.\_1 + "=" + item.\_2).mkString("|")

val pageSplitConvertRateRDD = spark.sparkContext.makeRDD(Array(PageSplitConvertRate(taskid,convertRate)))

import spark.implicits.\_ pageSplitConvertRateRDD.toDF().write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "page\_split\_convert\_rate")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

}

/\*\*

* 计算页面切片转化率
* @param pageSplitPvMap 页面切片 pv
* @param startPagePv 起始页面 pv
* @return

\*/

def computePageSplitConvertRate(taskParam:JSONObject, pageSplitPvMap:collection.Map[String, Long], startPagePv:Long):collection.Map[String, Double] = {

val convertRateMap = new mutable.HashMap[String, Double]()

//1,2,3,4,5,6,7

val targetPageFlow = ParamUtils.getParam(taskParam, Constants.PARAM\_TARGET\_PAGE\_FLOW) val targetPages = targetPageFlow.split(",").toList

//(1\_2,2\_3,3\_4,4\_5,5\_6,6\_7)

val targetPagePairs = targetPages.slice(0, targetPages.length-1).zip(targetPages.tail).map(item => item.\_1 + "\_" + item.\_2)

// lastPageSplitPv：存储最新一次的页面 PV 数量

var lastPageSplitPv = startPagePv.toDouble

// 3,5,2,4,6

// 3\_5

**———————————————**

**尚硅谷大数据项目之电商分析平台**

// 3\_5 pv / 3 pv

// 5\_2 rate = 5\_2 pv / 3\_5 pv

**——————————————**

// 通过 for 循环，获取目标页面流中的各个页面切片（pv）

for(targetPage <- targetPagePairs){

// 先获取 pageSplitPvMap 中记录的当前 targetPage 的数量

val targetPageSplitPv = pageSplitPvMap.get(targetPage).get.toDouble println((targetPageSplitPv, lastPageSplitPv))

// 用当前 targetPage 的数量除以上一次 lastPageSplit 的数量，得到转化率

val convertRate = NumberUtils.formatDouble(targetPageSplitPv / lastPageSplitPv, 2)

// 对 targetPage 和转化率进行存储

convertRateMap.put(targetPage, convertRate)

// 将本次的 targetPage 作为下一次的 lastPageSplitPv lastPageSplitPv = targetPageSplitPv

}

convertRateMap

}

/\*\*

* 获取页面流中初始页面的 pv
* @param taskParam
* @param sessionid2actionsRDD
* @return

\*/

def getStartPagePv(taskParam:JSONObject, sessionid2actionsRDD:RDD[(String, Iterable[UserVisitAction])]) :Long = {

// 获取配置文件中的 targetPageFlow

val targetPageFlow = ParamUtils.getParam(taskParam, Constants.PARAM\_TARGET\_PAGE\_FLOW)

// 获取起始页面 ID

val startPageId = targetPageFlow.split(",")(0).toLong

// sessionid2actionsRDD 是聚合后的用户行为数据

// userVisitAction 中记录的是在一个页面中的用户行为数据

val startPageRDD = sessionid2actionsRDD.flatMap{ case (sessionid, userVisitActions) =>

// 过滤出所有 PageId 为 startPageId 的用户行为数据

userVisitActions.filter(\_.page\_id == startPageId).map(\_.page\_id)

}

// 对 PageId 等于 startPageId 的用户行为数据进行技术

startPageRDD.count()

}

/\*\*

* 页面切片生成与匹配算法
* 注意，一开始我们只有 UserAciton 信息，通过将 UserAction 按照时间进行排序，然后提取 PageId，再进行连接，可以得到 PageFlow
* @param sc
* @param sessionid2actionsRDD
* @param taskParam
* @return

\*/

def generateAndMatchPageSplit(sc:SparkContext, sessionid2actionsRDD:RDD[(String, Iterable[UserVisitAction])], taskParam:JSONObject ):RDD[(String, Int)] = {

/\* 对目标 PageFlow 进行解析 \*/

//1,2,3,4,5,6,7

val targetPageFlow = ParamUtils.getParam(taskParam, Constants.PARAM\_TARGET\_PAGE\_FLOW)

//将字符串转换成为了 List[String]

val targetPages = targetPageFlow.split(",").toList

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

//targetPages.slice(0, targetPages.length-1) ：[1,2,3,4,5,6]

//targetPages.tail ：[2,3,4,5,6,7]

//targetPages.slice(0, targetPages.length-1).zip(targetPages.tail) ：

(1,2)(2,3)(3,4)(4,5)(5,6)(6,7)

//map(item => item.\_1 + "\_" + item.\_2)：(1\_2,2\_3,3\_4,4\_5,5\_6,6\_7)

val targetPagePairs = targetPages.slice(0, targetPages.length-1).zip(targetPages.tail).map(item => item.\_1 + "\_" + item.\_2)

//将结果转换为广播变量

//targetPagePairs 类型为 List[String]

val targetPageFlowBroadcast = sc.broadcast(targetPagePairs)

/\* 对所有 PageFlow 进行解析 \*/

// 对全部数据进行处理

sessionid2actionsRDD.flatMap{ case (sessionid, userVisitActions) =>

// 获取使用者指定的页面流

// 使用者指定的页面流，1,2,3,4,5,6,7

// 1->2 的转化率是多少？2->3 的转化率是多少？

// 这里，我们拿到的 session 的访问行为，默认情况下是乱序的

// 比如说，正常情况下，我们希望拿到的数据，是按照时间顺序排序的

// 但是问题是，默认是不排序的

// 所以，我们第一件事情，对 session 的访问行为数据按照时间进行排序

// 举例，反例

// 比如，3->5->4->10->7

// 3->4->5->7->10

// userVisitActions 是 Iterable[UserAction] ， toList.sortWith 将 Iterable 中的所有

UserAction 按照时间进行排序

// 按照时间排序

val sortedUVAs = userVisitActions.toList.sortWith((uva1, uva2) => DateUtils.parseTime(uva1.action\_time).getTime() < DateUtils.parseTime(uva2.action\_time).getTime())

// 提取所有 UserAction 中的 PageId 信息

val soredPages = sortedUVAs.map(item => if(item.page\_id != null) item.page\_id)

//【注意】页面的 PageFlow 是将 session 的所有 UserAction 按照时间顺序排序后提取 PageId,再将

PageId 进行连接得到的

// 按照已经排好的顺序对 PageId 信息进行整合，生成所有页面切片：(1\_2,2\_3,3\_4,4\_5,5\_6,6\_7) val sessionPagePairs = soredPages.slice(0,

soredPages.length-1).zip(soredPages.tail).map(item => item.\_1 + "\_" + item.\_2)

/\* 由此，得到了当前 session 的 PageFlow \*/

// 只要是当前 session 的 PageFlow 有一个切片与 targetPageFlow 中任一切片重合，那么就保留下来

// 目标：(1\_2,2\_3,3\_4,4\_5,5\_6,6\_7) 当前：(1\_2,2\_5,5\_6,6\_7,7\_8)

// 最后保留：(1\_2,5\_6,6\_7)

// 输 出 ：(1\_2, 1) (5\_6, 1) (6\_7, 1) sessionPagePairs.filter(targetPageFlowBroadcast.value.contains(\_)).map((\_,1))

}

}

def getActionRDDByDateRange(spark:SparkSession, taskParam:JSONObject): RDD[UserVisitAction] = {

val startDate = ParamUtils.getParam(taskParam, Constants.PARAM\_START\_DATE) val endDate = ParamUtils.getParam(taskParam, Constants.PARAM\_END\_DATE) import spark.implicits.\_

spark.sql("select \* from user\_visit\_action where date>='" + startDate + "' and date<='"

+ endDate + "'")

.as[UserVisitAction].rdd

}

}

## 需求六：各区域 Top3 商品统计

**4.6.1 需求解析**

###### 根据用户指定的日期查询条件范围，统计各个区域下的最热门【点击】的 top3

商品，区域信息、各个城市的信息在项目中用固定值进行配置，因为不怎么变动。

表 1-10 区域等级表

|  |  |
| --- | --- |
| 区域等级 | 地区名称 |
| A 级 | 华北、华东 |
| B 级 | 华南、华中 |
| C 级 | 西北、西南 |
| D 级 | 东北、其他 |

* + 1. 查询 task，获取日期范围，通过 Spark SQL，查询 user\_visit\_action 表中的指定日期范围内的数据 ，过滤出，商品点击 行为， click\_product\_id is not null ； click\_product\_id != 'NULL'； click\_product\_id != 'null'； city\_id， click\_product\_id。
    2. 使用 Spark SQL 从 MySQL 中查询出来城市信息（ city\_id、city\_name、area）， 用户访问行为数据要跟城市信息进行 join， city\_id、city\_name、area、product\_id， RDD，转换成 DataFrame，注册成一个临时表。
    3. Spark SQL 内置函数（ case when），对 area 打标记（华东大区，A 级，华中大区， B 级， 东北大区， C 级， 西北大区， D 级）， area\_level。
    4. 计算出来每个区域下每个商品的点击次数， group by area, product\_id；保留每个区域的城市名称列表； 自定义 UDAF， group\_concat\_distinct()函数，聚合出来一个 city\_names 字段， area、product\_id、city\_names、click\_count。
    5. join 商品明细表，hive（ product\_id、product\_name、extend\_info），extend\_info

是 json 类型，自定义 UDF，get\_json\_object()函数，取出其中的 product\_status 字段，

if()函数（ Spark SQL 内置函数） ， 判断， 0 自营， 1 第三方；（ area、product\_id、

city\_names、click\_count、product\_name、product\_status） 。

* + 1. 开窗函数，根据 area 来聚合，获取每个 area 下，click\_count 排名前 3 的 product信息； area 、 area\_level 、 product\_id 、 city\_names 、 click\_count 、 product\_name 、product\_status
    2. 结果写入 MySQL 表中。
    3. **数据源解析**



图 4-9 需求六数据源解析

* + 1. **数据结构解析**

Array((0L, "北京", "华北"), (1L, "上海", "华东"), (2L, "南京", "华东"), (3L, "广州", "华南"),

(4L, "三亚", "华南"), (5L, "武汉", "华中"), (6L, "长沙", "华中"), (7L, "西安", "西北"), (8L,

"成都", "西南"), (9L, "哈尔滨", "东北"))

## 需求实现流程

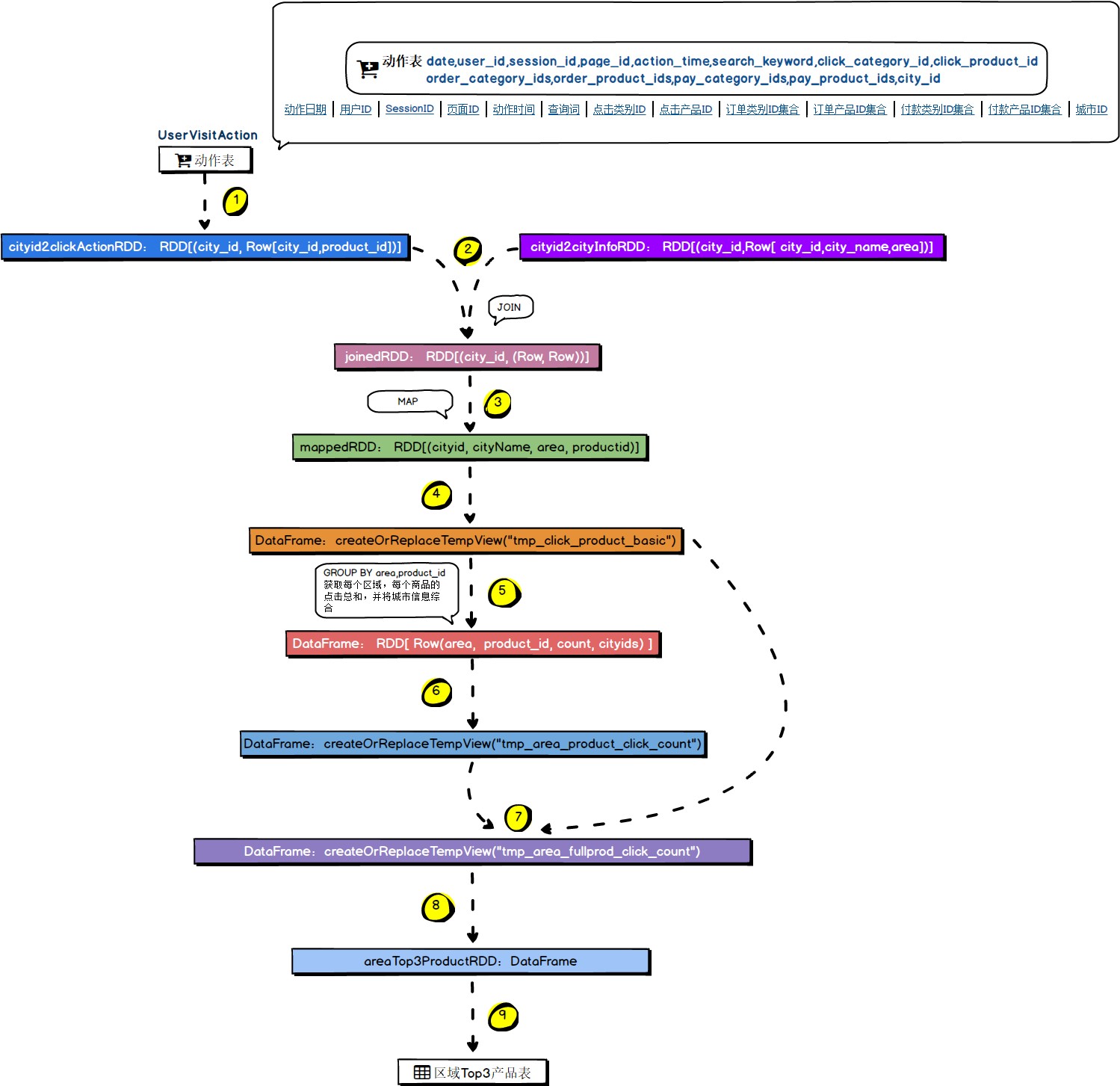


图 4-10 需求六实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `area\_top3\_product`

-- ----------------------------

DROP TABLE IF EXISTS `area\_top3\_product`; CREATE TABLE `area\_top3\_product` (

`taskid` varchar(255) DEFAULT NULL,

`area` varchar(255) DEFAULT NULL,

`areaLevel` varchar(255) DEFAULT NULL,

`productid` int(11) DEFAULT NULL,

`cityInfos` varchar(255) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL,

`productName` varchar(255) DEFAULT NULL,

`productStatus` varchar(255) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-12 需求六代码解析

/\*\*

\* 区域 Top3 商品统计

\*/

object AreaTop3ProductApp {

def main(args: Array[String]): Unit = {

// 获取统计任务参数【为了方便，直接从配置文件中获取，企业中会从一个调度平台获取】

val jsonStr = ConfigurationManager.config.getString(Constants.TASK\_PARAMS) val taskParam = JSONObject.fromObject(jsonStr)

// 任务的执行 ID，用户唯一标示运行后的结果，用在 MySQL 数据库中

val taskUUID = UUID.randomUUID().toString

// 构建 Spark 上下文

val sparkConf = new SparkConf().setAppName("SessionAnalyzer").setMaster("local[\*]")

// 创建 Spark 客户端

val

spark

=

SparkSession.builder().config(sparkConf).enableHiveSupport().getOrCreate() val sc = spark.sparkContext

// 注册自定义函数

spark.udf.register("concat\_long\_string", (v1: Long, v2: String, split: String) => v1.toString + split + v2)

spark.udf.register("get\_json\_object", (json: String, field: String) => { val jsonObject = JSONObject.fromObject(json); jsonObject.getString(field)

})

spark.udf.register("group\_concat\_distinct", new GroupConcatDistinctUDAF())

// 获取任务日期参数

val startDate = ParamUtils.getParam(taskParam, Constants.PARAM\_START\_DATE) val endDate = ParamUtils.getParam(taskParam, Constants.PARAM\_END\_DATE)

// 查询用户指定日期范围内的点击行为数据（city\_id，在哪个城市发生的点击行为）

val cityid2clickActionRDD = getcityid2ClickActionRDDByDate(spark, startDate, endDate)

// 查询城市信息

// 使用(city\_id , 城市信息)

val cityid2cityInfoRDD = getcityid2CityInfoRDD(spark)

// 生成点击商品基础信息临时表

// 将点击行为 cityid2clickActionRDD 和城市信息 cityid2cityInfoRDD 进行 Join 关联

// tmp\_click\_product\_basic

generateTempClickProductBasicTable(spark, cityid2clickActionRDD, cityid2cityInfoRDD)

// 生成各区域各商品点击次数的临时表

// 对 tmp\_click\_product\_basic 表中的数据进行 count 聚合统计，得到点击次数

**———————————————**

**尚硅谷大数据项目之电商分析平台**

// tmp\_area\_product\_click\_count

**——————————————**

generateTempAreaPrdocutClickCountTable(spark)

// 生成包含完整商品信息的各区域各商品点击次数的临时表

// 关联 tmp\_area\_product\_click\_count 表与 product\_info 表，在 tmp\_area\_product\_click\_count

基础上引入商品的详细信息

generateTempAreaFullProductClickCountTable(spark)

// 需求一：使用开窗函数获取各个区域内点击次数排名前 3 的热门商品

val areaTop3ProductRDD = getAreaTop3ProductRDD(taskUUID, spark)

// 将数据转换为 DF，并保存到 MySQL 数据库

import spark.implicits.\_

val areaTop3ProductDF = areaTop3ProductRDD.rdd.map(row =>

AreaTop3Product(taskUUID, row.getAs[String]("area"),

row.getAs[String]("area\_level"), row.getAs[Long]("product\_id"),

row.getAs[String]("city\_infos"), row.getAs[Long]("click\_count"), row.getAs[String]("product\_name"), row.getAs[String]("product\_status"))

).toDS

areaTop3ProductDF.write

.format("jdbc")

.option("url", ConfigurationManager.config.getString(Constants.JDBC\_URL))

.option("dbtable", "area\_top3\_product")

.option("user", ConfigurationManager.config.getString(Constants.JDBC\_USER))

.option("password", ConfigurationManager.config.getString(Constants.JDBC\_PASSWORD))

.mode(SaveMode.Append)

.save()

spark.close()

}

/\*\*

* 需求一：获取各区域 top3 热门商品
* 使用开窗函数先进行一个子查询,按照 area 进行分组，给每个分组内的数据，按照点击次数降序排序，打上一个组内的行号
* 接着在外层查询中，过滤出各个组内的行号排名前 3 的数据

\*

* @return

\*/

def getAreaTop3ProductRDD(taskid: String, spark: SparkSession): DataFrame = {

// 华北、华东、华南、华中、西北、西南、东北

// A 级：华北、华东

// B 级：华南、华中

// C 级：西北、西南

// D 级：东北

// case when

// 根据多个条件，不同的条件对应不同的值

// case when then ... when then ... else ... end

val sql = "SELECT " + "area," +

"CASE " +

"WHEN area='China North' OR area='China East' THEN 'A Level' " + "WHEN area='China South' OR area='China Middle' THEN 'B Level' " + "WHEN area='West North' OR area='West South' THEN 'C Level' " + "ELSE 'D Level' " +

**———————————————**

**尚硅谷大数据项目之电商分析平台**

"END area\_level," + "product\_id," + "city\_infos," + "click\_count," + "product\_name," + "product\_status " +

"FROM (" + "SELECT " +

"area," + "product\_id," + "click\_count," + "city\_infos," + "product\_name," + "product\_status," +

### ——————————————

"row\_number() OVER (PARTITION BY area ORDER BY click\_count DESC) rank " + "FROM tmp\_area\_fullprod\_click\_count " +

") t " + "WHERE rank<=3"

spark.sql(sql)

}

/\*\*

* 生成区域商品点击次数临时表（包含了商品的完整信息）
* @param spark

\*/

def generateTempAreaFullProductClickCountTable(spark: SparkSession) {

// 将之前得到的各区域各商品点击次数表，product\_id

// 去关联商品信息表，product\_id，product\_name 和 product\_status

// product\_status 要特殊处理，0，1，分别代表了自营和第三方的商品，放在了一个 json 串里面

// get\_json\_object()函数，可以从 json 串中获取指定的字段的值

// if()函数，判断，如果 product\_status 是 0，那么就是自营商品；如果是 1，那么就是第三方商品

// area, product\_id, click\_count, city\_infos, product\_name, product\_status

// 你拿到到了某个区域 top3 热门的商品，那么其实这个商品是自营的，还是第三方的，其实是很重要的一件事

// 技术点：内置 if 函数的使用

val sql = "SELECT " + "tapcc.area," + "tapcc.product\_id," + "tapcc.click\_count," + "tapcc.city\_infos," + "pi.product\_name," +

"if(get\_json\_object(pi.extend\_info,'product\_status')='0','Self','Third Party') product\_status " +

"FROM tmp\_area\_product\_click\_count tapcc " +

"JOIN product\_info pi ON tapcc.product\_id=pi.product\_id "

val df = spark.sql(sql)

df.createOrReplaceTempView("tmp\_area\_fullprod\_click\_count")

}

/\*\*

* 生成各区域各商品点击次数临时表
* @param spark

\*/

def generateTempAreaPrdocutClickCountTable(spark: SparkSession) {

**———————————————**

**尚硅谷大数据项目之电商分析平台**

**——————————————**

// 按照 area 和 product\_id 两个字段进行分组

// 计算出各区域各商品的点击次数

// 可以获取到每个 area 下的每个 product\_id 的城市信息拼接起来的串

val sql = "SELECT " + "area," + "product\_id," +

"count(\*) click\_count, " + "group\_concat\_distinct(concat\_long\_string(city\_id,city\_name,':')) city\_infos " +

"FROM tmp\_click\_product\_basic " + "GROUP BY area,product\_id "

val df = spark.sql(sql)

// 各区域各商品的点击次数（以及额外的城市列表）,再次将查询出来的数据注册为一个临时表

df.createOrReplaceTempView("tmp\_area\_product\_click\_count")

}

/\*\*

* 生成点击商品基础信息临时表
* @param cityid2clickActionRDD
* @param cityid2cityInfoRDD

\*/

def generateTempClickProductBasicTable(spark: SparkSession, cityid2clickActionRDD: RDD[(Long, Row)], cityid2cityInfoRDD: RDD[(Long, Row)]) {

// 执行 join 操作，进行点击行为数据和城市数据的关联

val joinedRDD = cityid2clickActionRDD.join(cityid2cityInfoRDD)

// 将上面的 JavaPairRDD，转换成一个 JavaRDD<Row>（才能将 RDD 转换为 DataFrame） val mappedRDD = joinedRDD.map { case (cityid, (action, cityinfo)) =>

val productid = action.getLong(1) val cityName = cityinfo.getString(1) val area = cityinfo.getString(2) (cityid, cityName, area, productid)

}

// 1 北京

// 2 上海

// 1 北京

// group by area,product\_id

// 1:北京,2:上海

// 两个函数

// UDF：concat2()，将两个字段拼接起来，用指定的分隔符

// UDAF：group\_concat\_distinct()，将一个分组中的多个字段值，用逗号拼接起来，同时进行去重import spark.implicits.\_

val df = mappedRDD.toDF("city\_id", "city\_name", "area", "product\_id")

// 为 df 创建临时表

df.createOrReplaceTempView("tmp\_click\_product\_basic")

}

/\*\*

* 使用 Spark SQL 从 MySQL 中查询城市信息
* @return

\*/

def getcityid2CityInfoRDD(spark: SparkSession): RDD[(Long, Row)] = {

val cityInfo = Array((0L, "北京", "华北"), (1L, "上海", "华东"), (2L, "南京", "华东"), (3L,

"广州", "华南"), (4L, "三亚", "华南"), (5L, "武汉", "华中"), (6L, "长沙", "华中"), (7L, "西安", "西北"), (8L, "成都", "西南"), (9L, "哈尔滨", "东北"))

import spark.implicits.\_

val cityInfoDF = spark.sparkContext.makeRDD(cityInfo).toDF("city\_id", "city\_name", "area")

cityInfoDF.rdd.map(item => (item.getAs[Long]("city\_id"), item))

}

/\*\*

* 查询指定日期范围内的点击行为数据

\*

* @param startDate 起始日期
* @param endDate 截止日期
* @return 点击行为数据

\*/

def getcityid2ClickActionRDDByDate(spark: SparkSession, startDate: String, endDate: String): RDD[(Long, Row)] = {

// 从 user\_visit\_action 中，查询用户访问行为数据

// 第一个限定：click\_product\_id，限定为不为空的访问行为，那么就代表着点击行为

// 第二个限定：在用户指定的日期范围内的数据

val sql = "SELECT " +

"city\_id," + "click\_product\_id " +

"FROM user\_visit\_action " +

"WHERE click\_product\_id IS NOT NULL and click\_product\_id != -1L " + "AND date>='" + startDate + "' " +

"AND date<='" + endDate + "'" val clickActionDF = spark.sql(sql)

//(cityid, row)

clickActionDF.rdd.map(item => (item.getAs[Long]("city\_id"), item))

}

}

代码清单 4-13 AreaTop3Product类

/\*\*

* @param taskid
* @param area
* @param areaLevel
* @param productid
* @param cityInfos
* @param clickCount
* @param productName
* @param productStatus

\*/

case class AreaTop3Product(taskid:String,

area:String, areaLevel:String, productid:Long, cityInfos:String, clickCount:Long, productName:String,

productStatus:String)

* 1. **需求七：广告黑名单实时统计**
     1. **需求解析**

实现实时的动态黑名单机制：将每天对某个广告点击超过 100 次的用户拉黑。

* + 1. **数据源解析**

1. Kafka 数据

timestamp province city userid adid

## 数据结构解析

((0L, "北京", "华北"), (1L, "上海", "华东"), (2L, "南京", "华东"), (3L, "广州", "华南"), (4L,

"三亚", "华南"), (5L, "武汉", "华中"), (6L, "长沙", "华中"), (7L, "西安", "西北"), (8L, "成都", "西南"), (9L, "哈尔滨", "东北"))

## 需求实现流程

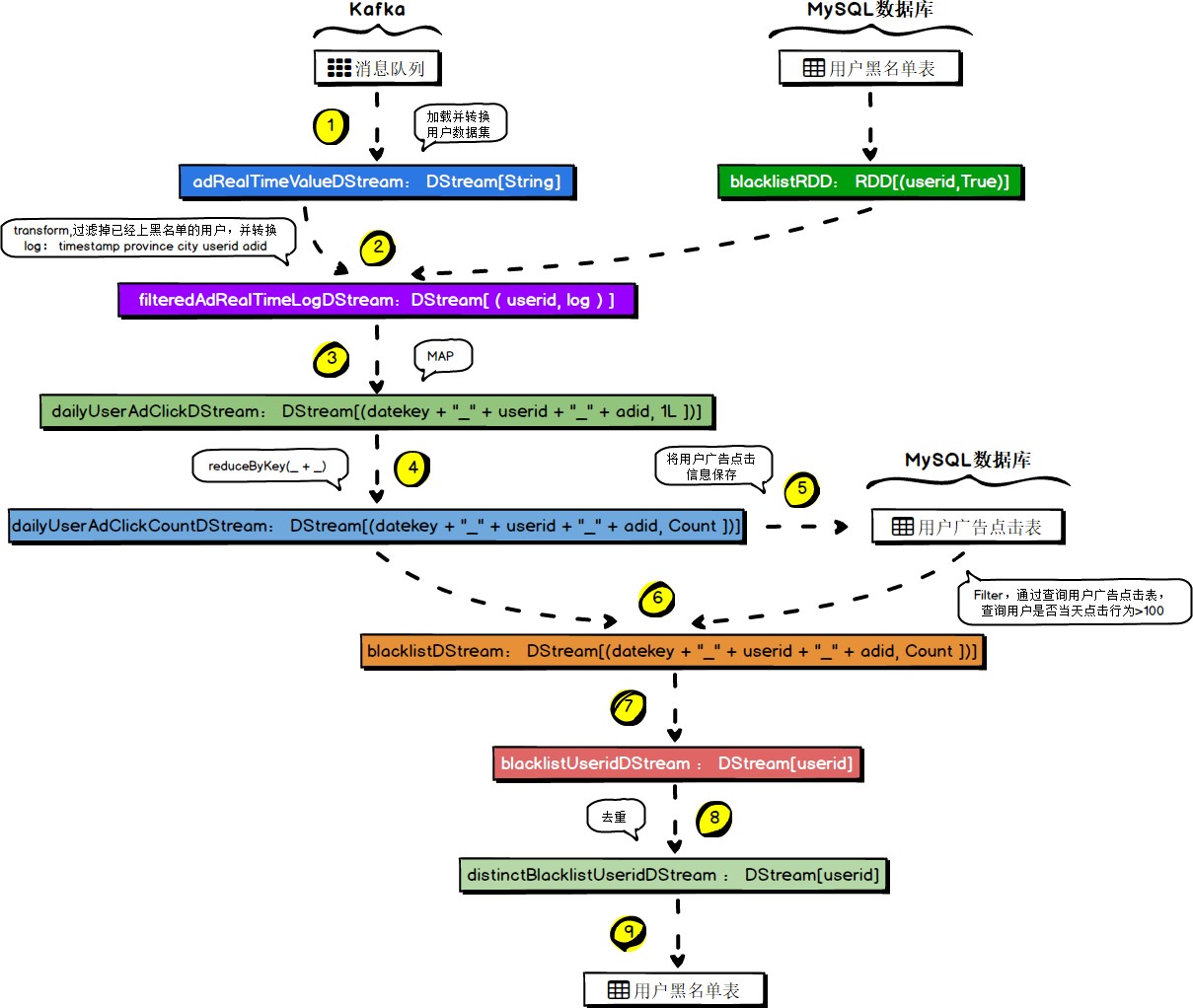


图 4-11 需求七实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `ad\_blacklist`

-- ----------------------------

DROP TABLE IF EXISTS `ad\_blacklist`; CREATE TABLE `ad\_blacklist` (

`userid` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------

-- Table structure for `ad\_user\_click\_count`

-- ----------------------------

DROP TABLE IF EXISTS `ad\_user\_click\_count`; CREATE TABLE `ad\_user\_click\_count` (

`date` varchar(30) DEFAULT NULL,

`userid` int(11) DEFAULT NULL,

`adid` int(11) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-14 需求七代码解析

/\*\*

* 业务功能一：生成动态黑名单
* @param filteredAdRealTimeLogDStream

\*/

def generateDynamicBlacklist(filteredAdRealTimeLogDStream: DStream[(Long, String)]) {

// 计算出每 5 个秒内的数据中，每天每个用户每个广告的点击量

// 通过对原始实时日志的处理

// 将日志的格式处理成<yyyyMMdd\_userid\_adid, 1L>格式

val dailyUserAdClickDStream = filteredAdRealTimeLogDStream.map{ case (userid,log) =>

// 从 tuple 中获取到每一条原始的实时日志

val logSplited = log.split(" ")

// 提取出日期（yyyyMMdd）、userid、adid val timestamp = logSplited(0)

val date = new Date(timestamp.toLong)

val datekey = DateUtils.formatDateKey(date)

val userid = logSplited(3).toLong val adid = logSplited(4)

// 拼接 key

val key = datekey + "\_" + userid + "\_" + adid (key, 1L)

}

// 针对处理后的日志格式，执行 reduceByKey 算子即可，（每个 batch 中）每天每个用户对每个广告的点击量

val dailyUserAdClickCountDStream = dailyUserAdClickDStream.reduceByKey(\_ + \_)

// 源源不断的，每个 5s 的 batch 中，当天每个用户对每支广告的点击次数

// <yyyyMMdd\_userid\_adid, clickCount> dailyUserAdClickCountDStream.foreachRDD{ rdd =>

rdd.foreachPartition{ items =>

// 对每个分区的数据就去获取一次连接对象

// 每次都是从连接池中获取，而不是每次都创建

// 写数据库操作，性能已经提到最高了

val adUserClickCounts = ArrayBuffer[AdUserClickCount]() for(item <- items){

val keySplited = item.\_1.split("\_")

val date = DateUtils.formatDate(DateUtils.parseDateKey(keySplited(0)))

// yyyy-MM-dd

val userid = keySplited(1).toLong val adid = keySplited(2).toLong val clickCount = item.\_2

//批量插入

adUserClickCounts += AdUserClickCount(date, userid,adid,clickCount)

}

AdUserClickCountDAO.updateBatch(adUserClickCounts.toArray)

}

}

// 现在我们在 mysql 里面，已经有了累计的每天各用户对各广告的点击量

// 遍历每个 batch 中的所有记录，对每条记录都要去查询一下，这一天这个用户对这个广告的累计点击量是多少

// 从 mysql 中查询

// 查询出来的结果，如果是 100，如果你发现某个用户某天对某个广告的点击量已经大于等于 100 了

// 那么就判定这个用户就是黑名单用户，就写入 mysql 的表中，持久化

val blacklistDStream = dailyUserAdClickCountDStream.filter{ case (key, count) => val keySplited = key.split("\_")

// yyyyMMdd -> yyyy-MM-dd

val date = DateUtils.formatDate(DateUtils.parseDateKey(keySplited(0))) val userid = keySplited(1).toLong

val adid = keySplited(2).toLong

// 从 mysql 中查询指定日期指定用户对指定广告的点击量

val clickCount = AdUserClickCountDAO.findClickCountByMultiKey(date, userid, adid)

// 判断，如果点击量大于等于 100，ok，那么不好意思，你就是黑名单用户

// 那么就拉入黑名单，返回 true if(clickCount >= 100) {

true

}else{

// 反之，如果点击量小于 100 的，那么就暂时不要管它了

false

}

}

// blacklistDStream

// 里面的每个 batch，其实就是都是过滤出来的已经在某天对某个广告点击量超过 100 的用户

// 遍历这个 dstream 中的每个 rdd，然后将黑名单用户增加到 mysql 中

// 这里一旦增加以后，在整个这段程序的前面，会加上根据黑名单动态过滤用户的逻辑

// 我们可以认为，一旦用户被拉入黑名单之后，以后就不会再出现在这里了

// 所以直接插入 mysql 即可

// 我们在插入前要进行去重

// yyyyMMdd\_userid\_adid

// 20151220\_10001\_10002 100

// 20151220\_10001\_10003 100

// 10001 这个 userid 就重复了

// 实际上，是要通过对 dstream 执行操作，对其中的 rdd 中的 userid 进行全局的去重, 返回 Userid

val blacklistUseridDStream = blacklistDStream.map(item => item.\_1.split("\_")(1).toLong)

val distinctBlacklistUseridDStream = blacklistUseridDStream.transform( uidStream => uidStream.distinct() )

// 到这一步为止，distinctBlacklistUseridDStream

// 每一个 rdd，只包含了 userid，而且还进行了全局的去重，保证每一次过滤出来的黑名单用户都没有重复的distinctBlacklistUseridDStream.foreachRDD{ rdd =>

rdd.foreachPartition{ items =>

val adBlacklists = ArrayBuffer[AdBlacklist]()

for(item <- items)

adBlacklists += AdBlacklist(item)

AdBlacklistDAO.insertBatch(adBlacklists.toArray)

}

}

}

代码清单 4-15 AdBlacklist类

/\*\*

* 广告黑名单
* @author wuyufei

\*

\*/

case class AdBlacklist(userid:Long)

代码清单 4-16 AdUserClickCount类

/\*\*

* 用户广告点击量
* @author wuyufei

\*

\*/

case class AdUserClickCount(date:String,

userid:Long, adid:Long,

clickCount:Long)

* 1. **需求八：广告点击量实时统计**
     1. **需求解析**

每天各省各城市各广告的点击流量实时统计。

* + 1. **数据源解析**

1. Kafka 数据

timestamp province city userid adid

## 数据结构解析

((0L, "北京", "华北"), (1L, "上海", "华东"), (2L, "南京", "华东"), (3L, "广州", "华南"), (4L,

"三亚", "华南"), (5L, "武汉", "华中"), (6L, "长沙", "华中"), (7L, "西安", "西北"), (8L, "成都", "西南"), (9L, "哈尔滨", "东北"))

## 需求实现流程

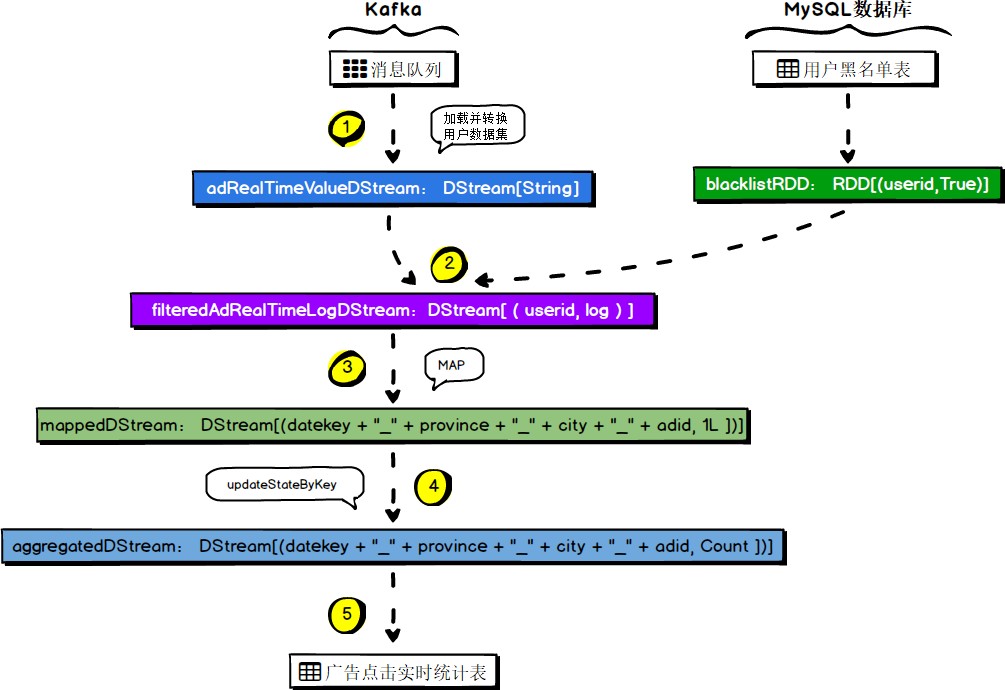


图 4-12 需求八实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `ad\_stat`

-- ----------------------------

DROP TABLE IF EXISTS `ad\_stat`; CREATE TABLE `ad\_stat` (

`date` varchar(30) DEFAULT NULL,

`province` varchar(100) DEFAULT NULL,

`city` varchar(100) DEFAULT NULL,

`adid` int(11) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-17 需求八代码解析

/\*\*

* 业务功能二：计算广告点击流量实时统计
* @param filteredAdRealTimeLogDStream
* @return

\*/

def calculateRealTimeStat(filteredAdRealTimeLogDStream:DStream[(Long, String)]):DStream[(String, Long)] = {

// 计算每天各省各城市各广告的点击量

// 设计出来几个维度：日期、省份、城市、广告

// 2015-12-01，当天，可以看到当天所有的实时数据（动态改变），比如江苏省南京市

// 广告可以进行选择（广告主、广告名称、广告类型来筛选一个出来）

// 拿着 date、province、city、adid，去 mysql 中查询最新的数据

// 等等，基于这几个维度，以及这份动态改变的数据，是可以实现比较灵活的广告点击流量查看的功能的

// date province city userid adid

// date\_province\_city\_adid，作为 key；1 作为 value

// 通过 spark，直接统计出来全局的点击次数，在 spark 集群中保留一份；在 mysql 中，也保留一份

// 我们要对原始数据进行 map，映射成<date\_province\_city\_adid,1>格式

// 然后呢，对上述格式的数据，执行 updateStateByKey 算子

// spark streaming 特有的一种算子，在 spark 集群内存中，维护一份 key 的全局状态

val mappedDStream = filteredAdRealTimeLogDStream.map{ case (userid, log) => val logSplited = log.split(" ")

val timestamp = logSplited(0)

val date = new Date(timestamp.toLong)

val datekey = DateUtils.formatDateKey(date)

val province = logSplited(1) val city = logSplited(2)

val adid = logSplited(4).toLong

val key = datekey + "\_" + province + "\_" + city + "\_" + adid

(key, 1L)

}

// 在这个 dstream 中，就相当于，有每个 batch rdd 累加的各个 key（各天各省份各城市各广告的点击次数）

// 每次计算出最新的值，就在 aggregatedDStream 中的每个 batch rdd 中反应出来

val aggregatedDStream = mappedDStream.updateStateByKey[Long]{ (values:Seq[Long], old:Option[Long]) =>

// 举例来说

// 对于每个 key，都会调用一次这个方法

// 比如 key 是<20151201\_Jiangsu\_Nanjing\_10001,1>，就会来调用一次这个方法 7

// 10 个

// values，(1,1,1,1,1,1,1,1,1,1)

// 首先根据 optional 判断，之前这个 key，是否有对应的状态var clickCount = 0L

// 如果说，之前是存在这个状态的，那么就以之前的状态作为起点，进行值的累加

if(old.isDefined) { clickCount = old.get

}

// values，代表了，batch rdd 中，每个 key 对应的所有的值

for(value <- values) { clickCount += value

}

Some(clickCount)

}

// 将计算出来的最新结果，同步一份到 mysql 中，以便于 j2ee 系统使用

aggregatedDStream.foreachRDD{ rdd => rdd.foreachPartition{ items =>

//批量保存到数据库

val adStats = ArrayBuffer[AdStat]()

for(item <- items){

val keySplited = item.\_1.split("\_") val date = keySplited(0)

val province = keySplited(1) val city = keySplited(2)

val adid = keySplited(3).toLong

val clickCount = item.\_2

adStats += AdStat(date,province,city,adid,clickCount)

}

AdStatDAO.updateBatch(adStats.toArray)

}

}

aggregatedDStream

}

代码清单 4-18 AdStat类

/\*\*

* 广告实时统计
* @author wuyufei

\*

\*/

case class AdStat(date:String,

province:String, city:String, adid:Long,

clickCount:Long)

* 1. **需求九：各省热门广告实时统计**
     1. **需求解析**

统计每天各省 top3 热门广告。

* + 1. **数据源解析**

数据来源于需求八 updateStateByKey 得到的 DStream。

* + 1. **数据结构解析**

Dstream[( dateKey\_province\_city\_adid , count)]

## 需求实现流程

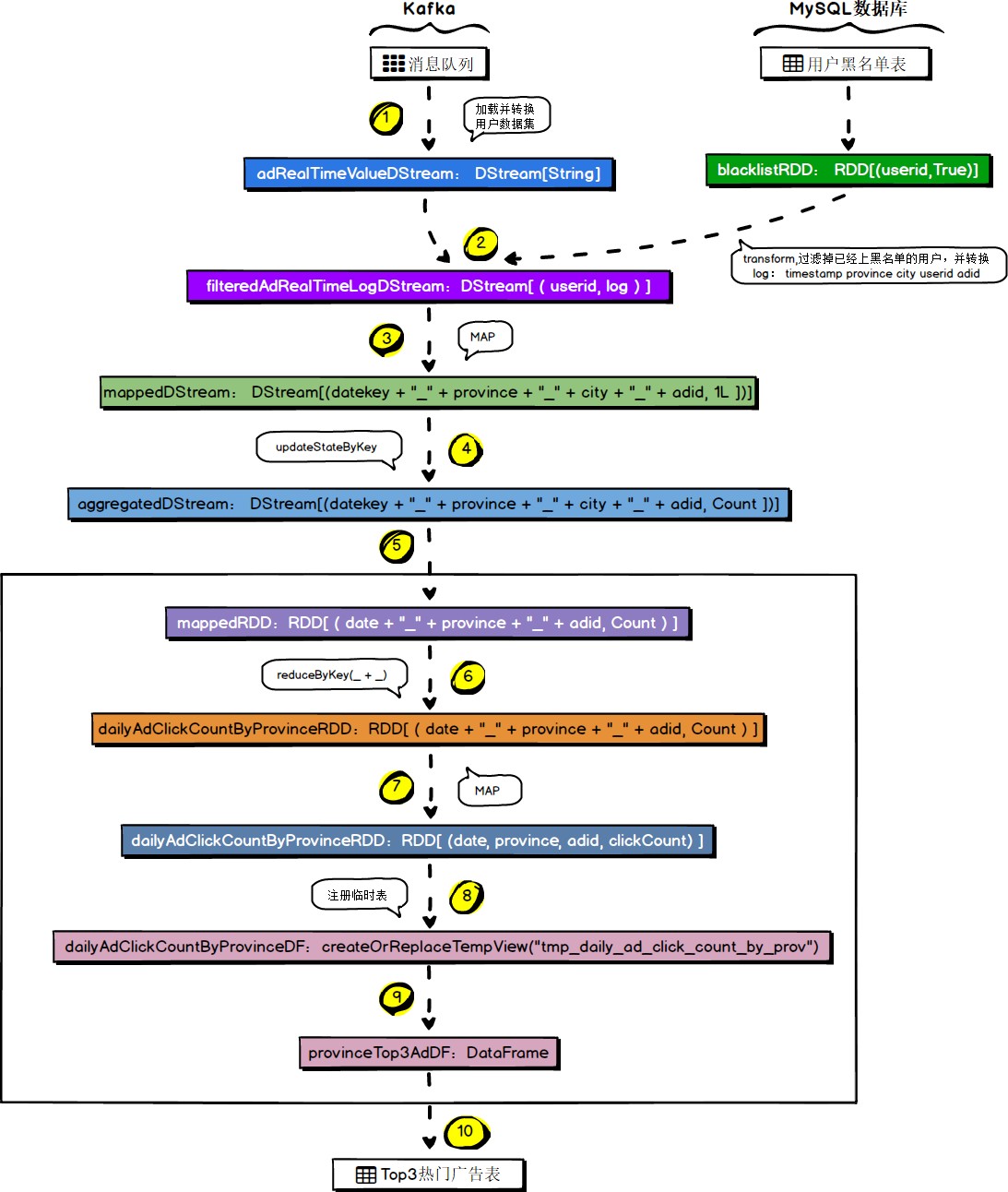


图 4-13 需求九实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `ad\_province\_top3`

-- ----------------------------

DROP TABLE IF EXISTS `ad\_province\_top3`; CREATE TABLE `ad\_province\_top3` (

`date` varchar(30) DEFAULT NULL,

`province` varchar(100) DEFAULT NULL,

`adid` int(11) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-19 需求九代码解析

/\*\*

* 业务功能三：计算每天各省份的 top3 热门广告
* @param adRealTimeStatDStream

\*/

def calculateProvinceTop3Ad(spark:SparkSession, adRealTimeStatDStream:DStream[(String, Long)]) {

// 每一个 batch rdd，都代表了最新的全量的每天各省份各城市各广告的点击量val rowsDStream = adRealTimeStatDStream.transform{ rdd =>

// <yyyyMMdd\_province\_city\_adid, clickCount>

// <yyyyMMdd\_province\_adid, clickCount>

// 计算出每天各省份各广告的点击量

val mappedRDD = rdd.map{ case (keyString, count) =>

val keySplited = keyString.split("\_") val date = keySplited(0)

val province = keySplited(1) val adid = keySplited(3).toLong val clickCount = count

val key = date + "\_" + province + "\_" + adid (key, clickCount)

}

val dailyAdClickCountByProvinceRDD = mappedRDD.reduceByKey( \_ + \_ )

// 将 dailyAdClickCountByProvinceRDD 转换为 DataFrame

// 注册为一张临时表

// 使用 Spark SQL，通过开窗函数，获取到各省份的 top3 热门广告

val rowsRDD = dailyAdClickCountByProvinceRDD.map{ case (keyString, count) =>

val keySplited = keyString.split("\_") val datekey = keySplited(0)

val province = keySplited(1) val adid = keySplited(2).toLong val clickCount = count

val date = DateUtils.formatDate(DateUtils.parseDateKey(datekey)) (date, province, adid, clickCount)

}

import spark.implicits.\_

val dailyAdClickCountByProvinceDF = rowsRDD.toDF("date","province","ad\_id", "click\_count")

// 将 dailyAdClickCountByProvinceDF，注册成一张临时表

dailyAdClickCountByProvinceDF.createOrReplaceTempView("tmp\_daily\_ad\_click\_count\_by\_pro v")

// 使用 Spark SQL 执行 SQL 语句，配合开窗函数，统计出各身份 top3 热门的广告

val provinceTop3AdDF = spark.sql( "SELECT "

+ "date,"

+ "province,"

+ "ad\_id,"

+ "click\_count "

+ "FROM ( "

+ "SELECT "

+ "date,"

+ "province,"

+ "ad\_id,"

+ "click\_count,"

+ "ROW\_NUMBER() OVER(PARTITION BY province ORDER BY click\_count DESC) rank "

+ "FROM tmp\_daily\_ad\_click\_count\_by\_prov "

+ ") t "

+ "WHERE rank>=3"

)

provinceTop3AdDF.rdd

}

// 每次都是刷新出来各个省份最热门的 top3 广告，将其中的数据批量更新到 MySQL 中

rowsDStream.foreachRDD{ rdd => rdd.foreachPartition{ items =>

// 插入数据库

val adProvinceTop3s = ArrayBuffer[AdProvinceTop3]()

for (item <- items){

val date = item.getString(0)

val province = item.getString(1) val adid = item.getLong(2)

val clickCount = item.getLong(3)

adProvinceTop3s += AdProvinceTop3(date,province,adid,clickCount)

}

AdProvinceTop3DAO.updateBatch(adProvinceTop3s.toArray)

}

}

}

代码清单 4-20 AdProvinceTop3 类

/\*\*

\* 各省 top3 热门广告

\* @author wuyufei

\*

\*/

case class AdProvinceTop3(date:String,

province:String, adid:Long,

clickCount:Long)

## 需求十：最近一小时广告点击量实时统计

* + 1. **需求解析**

###### 统计各广告最近 1 小时内的点击量趋势：各广告最近 1 小时内各分钟的点击量。

* + 1. **数据源解析**
       1. Kafka 数据

timestamp province city userid adid

## 数据结构解析

((0L, "北京", "华北"), (1L, "上海", "华东"), (2L, "南京", "华东"), (3L, "广州", "华南"), (4L,

"三亚", "华南"), (5L, "武汉", "华中"), (6L, "长沙", "华中"), (7L, "西安", "西北"), (8L, "成都", "西南"), (9L, "哈尔滨", "东北"))

## 需求实现流程

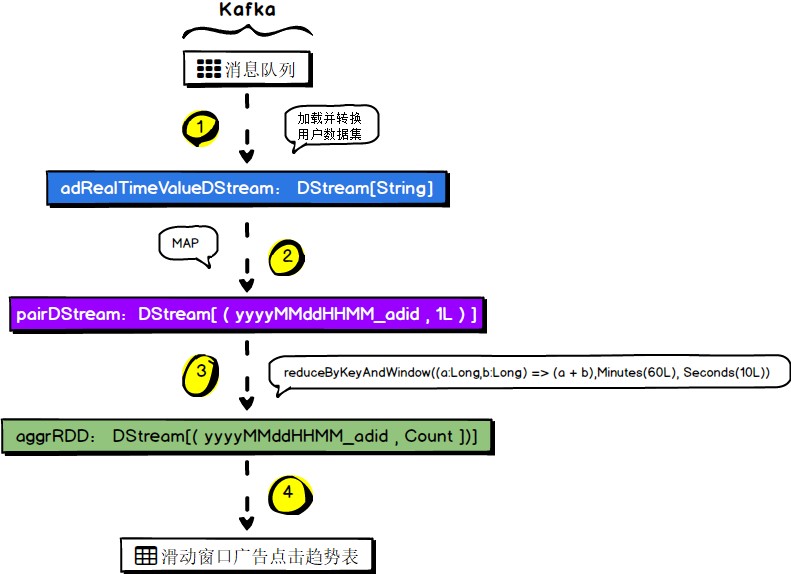


图 4-14 需求十实现流程

* + 1. **MySQL 存储结构解析**

-- ----------------------------

-- Table structure for `ad\_click\_trend`

-- ----------------------------

DROP TABLE IF EXISTS `ad\_click\_trend`; CREATE TABLE `ad\_click\_trend` (

`date` varchar(30) DEFAULT NULL,

`hour` varchar(30) DEFAULT NULL,

`minute` varchar(30) DEFAULT NULL,

`adid` int(11) DEFAULT NULL,

`clickCount` int(11) DEFAULT NULL

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

## 代码解析

代码清单 4-21 需求十代码解析

/\*\*

* 业务功能四：计算最近 1 小时滑动窗口内的广告点击趋势
* @param adRealTimeValueDStream

\*/

def calculateAdClickCountByWindow(adRealTimeValueDStream:DStream[String]) {

// 映射成<yyyyMMddHHMM\_adid,1L>格式

val pairDStream = adRealTimeValueDStream.map{ case consumerRecord => val logSplited = consumerRecord.split(" ")

val timeMinute = DateUtils.formatTimeMinute(new Date(logSplited(0).toLong)) val adid = logSplited(4).toLong

(timeMinute + "\_" + adid, 1L)

}

// 计算窗口函数，1 小时滑动窗口内的广告点击趋势

val aggrRDD = pairDStream.reduceByKeyAndWindow((a:Long,b:Long) => (a + b),Minutes(60L), Seconds(10L))

// 最近 1 小时内，各分钟的点击量，并保存到数据库

aggrRDD.foreachRDD{ rdd => rdd.foreachPartition{ items =>

//保存到数据库

val adClickTrends = ArrayBuffer[AdClickTrend]() for (item <- items){

val keySplited = item.\_1.split("\_")

// yyyyMMddHHmm

val dateMinute = keySplited(0) val adid = keySplited(1).toLong val clickCount = item.\_2

val date = DateUtils.formatDate(DateUtils.parseDateKey(dateMinute.substring(0,

8)))

val hour = dateMinute.substring(8, 10) val minute = dateMinute.substring(10)

adClickTrends += AdClickTrend(date,hour,minute,adid,clickCount)

}

AdClickTrendDAO.updateBatch(adClickTrends.toArray)

}

}

}

代码清单 4-22 AdClickTrend类

/\*\*

* 广告点击趋势
* @author wuyufei

\*

\*/

case class AdClickTrend(date:String,

hour:String, minute:String, adid:Long,

clickCount:Long)

**第五章 项目总结**

本项目通过 Spark 技术生态栈中的 Spark Core、Spark SQL 和 Spark Streaming 三个技术框架， 实现了对电商平台业务的离线和实时数据统计与分析，完成了包括用户访问 session 分析、页面单跳转化率统计、热门商品离线统计、广告流量实时统计 4 个业务模块的开发工作。

本项目涵盖了 Spark Core、Spark SQL 和 Spark Streaming 三个技术框架中核心

的知识点与技术点，对于同学们真正的理解和掌握 Spark 技术生态栈有着良好的促

进作用。