尚硅谷大数据项目之实时数仓(ads层)

(作者：尚硅谷大数据研发部)

版本：V1.0.0

# ads层数据

## ads层概述

在前面的dws层, 我们是把宽表明细数据写入到OLAP数据库中, 然后分析, 展示.

我们也可以再加一层, 直接使用spark-steaming把**聚合后**的结果, 写入到msyql中, 然后再提供接口展示. 这层就是ads层.

ads层，主要是根据各种报表及可视化来生成统计数据。通常这些报表及可视化都是基于某些维度的汇总统计。

## ads层需求

* 热门品牌统计(课堂完成)
* 热门品类统计(自己完成)
* 热门商品统计 (自己完成)
* 交易用户性别对比(自己完成)
* 交易用户年龄段对比(自己完成)
* 交易额省市分布(自己完成)

## 聚合思路

以热门品牌统计为例, 以商品分摊金额作为热门来统计(以个数也可以, 统一口径)

* 最终的数据应该是这样的:

|  |  |  |
| --- | --- | --- |
| 时间段 | 品牌名 | 品牌(所有商品)分摊总额 |
| 2020-10-01 xxx 至 2020-10-02 xxx | 华为 | 888888.88 |
|  |  |  |

如果是离线数据, 这个很容易统计.

实时数据直接统计出来时间段, 是有难度的, 而且代价极大(spark-Streaming状态)

我们退而其次, 不统计最终结果, 而是每个批次聚合一次, 将来从Mysql查询数据 的时候再做一次聚合. 因为写入到mysql的数据已经是每批次聚合一次了, 数据量已 经不是那么大了, 所以其实可行的

* 我们存入到mysql的数据应该是这样的

|  |  |  |
| --- | --- | --- |
| 聚合时间 | 品牌名 | 品牌(所有商品)分摊总额 |
| 2020-10-01 10:10:10 | 华为 | 888888.88 |
| 2020-10-01 10:10:15 | 华为 | 999999.99 |

* 表的主键问题

主键应该是聚合时间+品牌名

## 数据的精准消费一次

前面我们精准一次消费是利用了 手动保存偏移量+输出系统的幂等 来实现的

如何我们把数据写入到mysql, 幂等这块很难保证. 聚合时间+品牌名 **当碰到数据重复保存的时候, 这个联合主键是不会重复的, 所以无法满足幂等.**

**所以精准一次消费需要使用** 手动保存偏移量+事务 来实现

如果使用事务保证精准一次消费, 则数据库必须支持事务, 这就是为什么选择mysql的原因. 而且要把偏移量的保存和数据的保存在同一数据库中来保证他们处于同一事务中.

* 在drvier还是executor上使用事务?

如果在executor上使用事务, 则executor之间不能保证同时成功或失败, 需要另外考虑分布式事务, 这个一般不使用

我们采取在driver端开启事务, 所有的数据都通过driver写到mysql. 原因有2点:

1. 实现方便, 不容易有bug
2. 数据量量(聚合后的数据), 单独一个driver写入是可以完成的

# OrderWide数据写入到dws层

把OrderWide数据写入到Kafka的dws层.

## 重构DwsOrderWideApp

去掉写入到clickhouse的代码, 添加写入到kafka的代码.

把数据写入到topic dws\_order\_wide中

result.foreachRDD**(**rdd => **{**

// 写入到 kafka 的 dws 层

println**("时间戳.....开始")**

rdd.foreachPartition**(**orderWideIt => **{**

**val** producer: KafkaProducer**[**String, String**]** = MyKafkaUtil.getKafkaProducer**()**

orderWideIt.foreach**(**orderWide => **{**

**implicit val** f = org.json4s.DefaultFormats

producer.send**(new** ProducerRecord**[**String, String**]("dwd\_order\_wide"**, Serialization.write**(**orderWide**)))**

**})**

producer.close**()**

**})**

println**("时间戳.....结束")**

OffsetManager.saveOffsets**(**offsetRanges, groupId, topics**)**

**})**

## 确认kafka是否收到数据

一定要确认kafka收到数据再去做后面的工作!!!

# 消费dws层数据:dws\_order\_wide

## 添加必须的依赖

## 升级OffsetManager

offset要保存到mysql中, 重构OffsetManger, 实现从msyql读取便宜和写入偏移量到mysql.

### 在mysql中创建存储偏移量的表

create database gmall\_result;

use gmall\_result;

CREATE TABLE `ads\_offset` (

`group\_id` varchar(200) NOT NULL,

`topic` varchar(200) NOT NULL,

`partition\_id` int(11) NOT NULL,

`partition\_offset` bigint(20) DEFAULT NULL,

PRIMARY KEY (`group\_id`,`topic`,`partition\_id`)

) ENGINE=InnoDB DEFAULT CHARSET=utf8

### 创建从mysql读取数据的JDBC工具类

**package** com.atguigu.gmall.realtime.util

**import** java.sql.***{***DriverManager, PreparedStatement, ResultSet, ResultSetMetaData***}***

*/\*\**

*\* Author lzc*

*\* Date 2020/8/27 1:57 下午*

*\*/*

**object** JDBCUtil ***{***

*/\*\**

*\* 执行 sql, 并把查询到的结果封装到 C 类中*

*\*/*

**def** query***(***url: String, sql: String, args: List***[***Object***])***: List***[***Map***[***String, Object***]]*** = ***{***

**var** result = *List****[***Map***[***String, Object***]]()***

**val** conn = DriverManager.*getConnection****(***url***)***

**val** ps: PreparedStatement = conn.prepareStatement***(***sql***)***

***(***1 to args.size***)***.foreach***(***i => ***{***

ps.setObject***(***i, args***(***i - 1***))***

***})***

**val** resultSet: ResultSet = ps.executeQuery***()***

**val** meta: ResultSetMetaData = resultSet.getMetaData *// 需要知道有多少列*

**while *(***resultSet.next***()) {***

**var** map: Map***[***String, Object***]*** = *Map****[***String, Object***]()***

**for *(***i <- 1 to meta.getColumnCount***) {*** *// 遍历每一列*

**val** name = meta.getColumnName***(***i***)***.toLowerCase***()***

**val** value = resultSet.getObject***(***i***)***

map += name -> value

***}***

result :+= map

***}***

ps.close***()***

conn.close***()***

result

***}***

***}***

### 创建OffsetManageV2 读写偏移量

**package** com.atguigu.gmall.realtime.util

**import** org.apache.kafka.common.TopicPartition

*/\*\**

*\* Author lzc*

*\* Date 2020/8/26 7:22 下午*

*\*/*

**object** OffsetManagerV2 ***{***

**def** readOffset***(***groupId:String, topic: String***)*** = ***{***

**val** url = **"jdbc:mysql://hadoop102:3306/gmall\_result?characterEncoding=utf-8&useSSL=false&user=root&password=aaaaaa"**

**val** sql =

**"""**

**|select**

**| \***

**|from ads\_offset**

**|where topic=? and group\_id=?**

**|"""**.stripMargin

JDBCUtil

.*query****(***url, sql, *List****(***topic, groupId***))***

.map***(***row => ***{***

**val** partitionId = row***(*"partition\_id"*)***.toString.toInt

**val** partitionOffset = row***(*"partition\_offset"*)***.toString.toLong

***(*new** TopicPartition***(***topic, partitionId***)***, partitionOffset***)***

***})***

.toMap

***}***

*// 写偏移量这块将来要和写数据在一个事务中, 会用专门支持事务的工具完成, 此处略*

***}***

## 升级BaseApp为BaseAppV4

**注意修改调用读取偏移量的方法**

**package** com.atguigu.gmall.realtime

**import** com.atguigu.gmall.realtime.util.***{***MyKafkaUtil, OffsetManager, OffsetManagerV2***}***

**import** org.apache.kafka.clients.consumer.ConsumerRecord

**import** org.apache.kafka.common.TopicPartition

**import** org.apache.spark.SparkConf

**import** org.apache.spark.streaming.dstream.DStream

**import** org.apache.spark.streaming.kafka010.***{***HasOffsetRanges, OffsetRange***}***

**import** org.apache.spark.streaming.***{***Seconds, StreamingContext***}***

**import** scala.collection.mutable.ListBuffer

*/\*\**

*\* Author lzc*

*\* Date 2020/8/26 7:58 下午*

*消费单个topic, offset存储在mysql中*

*\*/*

**abstract class** BaseAppV4 ***{***

**var** appName: String

**var** groupId: String

**var** topic: String

**def** run***(***ssc: StreamingContext, offsetRanges: ListBuffer***[***OffsetRange***]***, sourceStream: DStream***[***ConsumerRecord***[***String, String***]])***

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

**val** conf: SparkConf = **new** SparkConf***()***.setMaster***(*"local[2]"*)***.setAppName***(***appName***)***

**val** ssc: StreamingContext = **new** StreamingContext***(***conf, *Seconds****(***3***))***

**val** fromOffsets: Map***[***TopicPartition, Long***]*** = OffsetManagerV2.*readOffset****(***groupId, topic***)***

**val** offsetRanges: ListBuffer***[***OffsetRange***]*** = ListBuffer.empty***[***OffsetRange***]***

**val** sourceStream: DStream***[***ConsumerRecord***[***String, String***]]*** = MyKafkaUtil

.*getKafkaStream****(***ssc, groupId, topic, fromOffsets***)***

.transform***(***rdd => ***{***

offsetRanges.clear

**val** newOffsetRanges: Array***[***OffsetRange***]*** = rdd.asInstanceOf***[***HasOffsetRanges***]***.offsetRanges

offsetRanges ++= newOffsetRanges

rdd

***})***

run***(***ssc, offsetRanges, sourceStream***)***

ssc.start***()***

ssc.awaitTermination***()***

***}***

***}***

## 创建AdsOrderWideApp

消费dws\_order\_wide数据, 计算每个品牌的交易额

**package** com.atguigu.gmall.realtime.ads

**import** com.atguigu.gmall.realtime.BaseAppV4

**import** com.atguigu.gmall.realtime.bean.OrderWide

**import** org.apache.kafka.clients.consumer.ConsumerRecord

**import** org.apache.spark.streaming.StreamingContext

**import** org.apache.spark.streaming.dstream.DStream

**import** org.apache.spark.streaming.kafka010.OffsetRange

**import** org.json4s.jackson.JsonMethods

**import** scala.collection.mutable.ListBuffer

*/\*\**

*\* Author lzc*

*\* Date 2020/8/31 2:25 下午*

*\*/*

**object** AdsOrderWideApp **extends** BaseAppV4 ***{***

**override var** *appName*: String = **"AdsOrderWideApp"**

**override var** *groupId*: String = **"AdsOrderWideApp"**

**override var** *topic*: String = **"dwd\_order\_wide"**

**override def** run***(***ssc: StreamingContext,

offsetRanges: ListBuffer***[***OffsetRange***]***,

sourceStream: DStream***[***ConsumerRecord***[***String, String***]])***: Unit = ***{***

*// 1. 没 3 秒统计一次 每个品的交易额*

**val** trademarkToAmountStream: DStream***[((***Long, String***)***, Double***)]*** = sourceStream

.map***(***record => ***{***

**implicit val** f = org.json4s.DefaultFormats

**val** orderWide: OrderWide = JsonMethods.parse***(***record.value***())***.extract***[***OrderWide***]***

***((***orderWide.tm\_id, orderWide.tm\_name***)***, orderWide.final\_detail\_amount***)***

***})***

.reduceByKey***(***\_ + \_***)***

***}***

***}***

## 数据写入到mysql

### 在mysql创建数据表

use gmall\_result;

CREATE TABLE `tm\_amount` (

stat\_time datetime,

tm\_id varchar(20),

tm\_name varchar(200),

amount decimal(16,2) ,

PRIMARY KEY (`stat\_time`,`tm\_id`,`tm\_name`)

)ENGINE=InnoDB DEFAULT CHARSET=utf8;

### 写入到mysql

写数据和写offset要考虑他们在一个事务中完成

* **写数据**

根据数据特点绝对不会重复

* **写offset**

第一次写入的时候是插入, 第二次写入应该是更新. 也就是说写入的时候, 如果没有就插入, 有(主键重复)就更新

mysql有2种方式可以实现这种需求

INSERT INTO ads\_offset VALUES('g1', 't1', 1, 400)

ON DUPLICATE KEY UPDATE partition\_offset=400;

REPLACE INTO ads\_offset values('g1', 't1', 1, 400);

### 具体实现代码

// 2. 数据写入到 mysql

// scalalike jdbc 加载配置

DBs.setup**()**

trademarkToAmountStream.foreachRDD**(**rdd => **{**

**val** tmAndAmount: Array**[((**Long, String**)**, Double**)]** = rdd.collect**()**

tmAndAmount.foreach**(**println**)**

println**(**offsetRanges**)**

**if (**tmAndAmount.nonEmpty**) {**

// 所有的操作都会处于同一个事务中

DB.localTx**(implicit** session => **{**

// 插入数据

**val** dt = LocalDateTime.now**()**

**val** stat\_time = **s"${**dt.toLocalDate**}\_${**dt.toLocalTime**}"**

// 使用批次提交: 批次中的参数 每个 Seq 存储一行 sql 的参数

**val** dataBatchParamList: List**[**Seq**[**Any**]]** = tmAndAmount.map **{**

**case ((**tm\_id, tm\_name**)**, amount**)** =>

Seq**(**stat\_time, tm\_id, tm\_name, amount**)**

**}**.toList

**val** insertDataSql =

**"""**

**|insert into tm\_amount values(?, ?, ?, ?)**

**|"""**.stripMargin

SQL**(**insertDataSql**)**.batch**(**dataBatchParamList: \_\***)**.apply**()**

// 插入或更新 offset

**val** offsetBatchParamList = offsetRanges.map**(**offsetRange => **{**

Seq**(**groupId, topic, offsetRange.partition, offsetRange.untilOffset**)**

**})**

**throw new** UnsupportedOperationException

**val** insertOffsetSql =

**"""**

**|replace into ads\_offset values(?, ?, ?, ?)**

**|"""**.stripMargin

SQL**(**insertOffsetSql**)**.batch**(**offsetBatchParamList: \_\***)**.apply**()**

**})**

**}**

**})**

### 在mysql中确认是否收到数据

测试事务是否可以正常工作!

比如在2个插入之间一个抛异常

# 发布接口

# DataV可视化展示