尚硅谷大数据技术之Flink电商实时数仓

DWS层业务实现

(作者：尚硅谷大数据研发部)

版本：V 2.0

# DWS层与DWM层的设计

## 设计思路

我们在之前通过分流等手段，把数据分拆成了独立的Kafka Topic。那么接下来如何处理数据，就要思考一下我们到底要通过实时计算出哪些指标项。

因为实时计算与离线不同，实时计算的开发和运维成本都是非常高的，要结合实际情况考虑是否有必要象离线数仓一样，建一个大而全的中间层。

如果没有必要大而全，这时候就需要大体规划一下要实时计算出的指标需求了。把这些指标以主题宽表的形式输出就是我们的DWS层。

## 需求梳理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 统计主题 | 需求指标 | 输出方式 | 计算来源 | 来源层级 |
| 访客 | pv | 可视化大屏 | page\_log直接可求 | dwd |
| uv | 可视化大屏 | 需要用page\_log过滤去重 | dwm |
| 跳出率 | 可视化大屏 | 需要通过page\_log行为判断 | dwm |
| 进入页面数 | 可视化大屏 | 需要识别开始访问标识 | dwd |
| 连续访问时长 | 可视化大屏 | page\_log直接可求 | dwd |
| 商品 | 点击 | 多维分析 | page\_log直接可求 | dwd |
| 曝光 | 多维分析 | page\_log直接可求 | dwd |
| 收藏 | 多维分析 | 收藏表 | dwd |
| 加入购物车 | 多维分析 | 购物车表 | dwd |
| 下单 | 可视化大屏 | 订单宽表 | dwm |
| 支付 | 多维分析 | 支付宽表 | dwm |
| 退款 | 多维分析 | 退款表 | dwd |
| 评论 | 多维分析 | 评论表 | dwd |
| 地区 | pv | 多维分析 | page\_log直接可求 | dwd |
| uv | 多维分析 | 需要用page\_log过滤去重 | dwm |
| 下单 | 可视化大屏 | 订单宽表 | dwd |
| 关键词 | 搜索关键词 | 可视化大屏 | 页面访问日志 直接可求 | dwd |
| 点击商品关键词 | 可视化大屏 | 商品主题下单再次聚合 | dws |
| 下单商品关键词 | 可视化大屏 | 商品主题下单再次聚合 | dws |

当然实际需求还会有更多，这里主要以为可视化大屏为目的进行实时计算的处理。

## DWS层的定位是什么

* 轻度聚合，因为DWS层要应对很多实时查询，如果是完全的明细那么查询的压力是非常大的。
* 将更多的实时数据以主题的方式组合起来便于管理，同时也能减少维度查询的次数。

# DWS层-访客主题宽表的计算

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 统计主题 | 需求指标 | 输出方式 | 计算来源 | 来源层级 |
| 访客 | pv | 可视化大屏 | page\_log直接可求 | dwd |
| uv | 可视化大屏 | 需要用page\_log过滤去重 | dwm |
| 跳出率 | 可视化大屏 | 需要通过page\_log行为判断 | dwm |
| 进入页面数 | 可视化大屏 | 需要识别开始访问标识 | dwd |
| 连续访问时长 | 可视化大屏 | page\_log直接可求 | dwd |

设计一张DWS层的表其实就两件事：维度和度量(事实数据)

* 度量包括PV、UV、跳出次数、进入页面数(session\_count)、连续访问时长
* 维度包括在分析中比较重要的几个字段：渠道、地区、版本、新老用户进行聚合

## 需求分析与思路

* 接收各个明细数据，变为数据流
* 把数据流合并在一起，成为一个相同格式对象的数据流
* 对合并的流进行聚合，聚合的时间窗口决定了数据的时效性
* 把聚合结果写在数据库中

## 功能实现

### 封装VisitorStatsApp,读取Kafka各个流数据

package com.atguigu.gmall.realtime.app.dws;

import com.atguigu.gmall.realtime.utils.MyKafkaUtil;

import org.apache.flink.streaming.api.datastream.DataStreamSource;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

/\*\*

\* Desc: 访客主题宽表计算

\* <p>

\* ?要不要把多个明细的同样的维度统计在一起?

\* 因为单位时间内mid的操作数据非常有限不能明显的压缩数据量（如果是数据量够大，或者单位时间够长可以）

\* 所以用常用统计的四个维度进行聚合 渠道、新老用户、app版本、省市区域

\* 度量值包括 启动、日活（当日首次启动）、访问页面数、新增用户数、跳出数、平均页面停留时长、总访问时长

\* 聚合窗口： 10秒

\* <p>

\* 各个数据在维度聚合前不具备关联性，所以先进行维度聚合

\* 进行关联 这是一个fulljoin

\* 可以考虑使用flinksql 完成

\*/

public class VisitorStatsApp {

public static void main(String[] args) throws Exception {

//TODO 0.基本环境准备

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

//设置并行度

env.setParallelism(4);

/\*

//检查点CK相关设置

env.enableCheckpointing(5000, CheckpointingMode.AT\_LEAST\_ONCE);

env.getCheckpointConfig().setCheckpointTimeout(60000);

StateBackend fsStateBackend = new FsStateBackend(

"hdfs://hadoop102:8020/gmall/flink/checkpoint/VisitorStatsApp");

env.setStateBackend(fsStateBackend);

System.setProperty("HADOOP\_USER\_NAME","atguigu");

\*/

String groupId = "visitor\_stats\_app";

//TODO 1.从Kafka的pv、uv、跳转明细主题中获取数据

String pageViewSourceTopic = "dwd\_page\_log";

String uniqueVisitSourceTopic = "dwm\_unique\_visit";

String userJumpDetailSourceTopic = "dwm\_user\_jump\_detail";

FlinkKafkaConsumer<String> pageViewSource = MyKafkaUtil.getKafkaSource(pageViewSourceTopic, groupId);

FlinkKafkaConsumer<String> uniqueVisitSource = MyKafkaUtil.getKafkaSource(uniqueVisitSourceTopic, groupId);

FlinkKafkaConsumer<String> userJumpSource = MyKafkaUtil.getKafkaSource(userJumpDetailSourceTopic, groupId);

DataStreamSource<String> pageViewDStream = env.addSource(pageViewSource);

DataStreamSource<String> uniqueVisitDStream = env.addSource(uniqueVisitSource);

DataStreamSource<String> userJumpDStream = env.addSource(userJumpSource);

pageViewDStream.print("pv-------->");

uniqueVisitDStream.print("uv=====>");

userJumpDStream.print("uj--------->");

env.execute();

}

}

### 合并数据流

把数据流合并在一起，成为一个相同格式对象的数据流

合并数据流的核心算子是union。但是union算子，要求所有的数据流结构必须一致。所以union前要调整数据结构。

#### 封装主题宽表实体类VisitorStats

package com.atguigu.gmall.realtime.bean;

/\*\*

\* Desc: 访客统计实体类 包括各个维度和度量

\*/

@Data

@AllArgsConstructor

public class VisitorStats {

//统计开始时间

private String stt;

//统计结束时间

private String edt;

//维度：版本

private String vc;

//维度：渠道

private String ch;

//维度：地区

private String ar;

//维度：新老用户标识

private String is\_new;

//度量：独立访客数

private Long uv\_ct=0L;

//度量：页面访问数

private Long pv\_ct=0L;

//度量： 进入次数

private Long sv\_ct=0L;

//度量： 跳出次数

private Long uj\_ct=0L;

//度量： 持续访问时间

private Long dur\_sum=0L;

//统计时间

private Long ts;

}

#### 对读取的各个数据流进行结构的转换

//TODO 2.对读取的流进行结构转换

//2.1 转换pv流

SingleOutputStreamOperator<VisitorStats> pageViewStatsDstream = pageViewDStream.map(

json -> {

// System.out.println("pv:"+json);

JSONObject jsonObj = JSON.parseObject(json);

return new VisitorStats("", "",

jsonObj.getJSONObject("common").getString("vc"),

jsonObj.getJSONObject("common").getString("ch"),

jsonObj.getJSONObject("common").getString("ar"),

jsonObj.getJSONObject("common").getString("is\_new"),

0L, 1L, 0L, 0L, jsonObj.getJSONObject("page").getLong("during\_time"), jsonObj.getLong("ts"));

});

//2.2转换uv流

SingleOutputStreamOperator<VisitorStats> uniqueVisitStatsDstream = uniqueVisitDStream.map(

json -> {

JSONObject jsonObj = JSON.parseObject(json);

return new VisitorStats("", "",

jsonObj.getJSONObject("common").getString("vc"),

jsonObj.getJSONObject("common").getString("ch"),

jsonObj.getJSONObject("common").getString("ar"),

jsonObj.getJSONObject("common").getString("is\_new"),

1L, 0L, 0L, 0L, 0L, jsonObj.getLong("ts"));

});

//2.3 转换sv流

SingleOutputStreamOperator<VisitorStats> sessionVisitDstream = pageViewDStream.process(

new ProcessFunction<String, VisitorStats>() {

@Override

public void processElement(String json, Context ctx, Collector<VisitorStats> out) throws Exception {

JSONObject jsonObj = JSON.parseObject(json);

String lastPageId = jsonObj.getJSONObject("page").getString("last\_page\_id");

if (lastPageId == null || lastPageId.length() == 0) {

// System.out.println("sc:"+json);

VisitorStats visitorStats = new VisitorStats("", "",

jsonObj.getJSONObject("common").getString("vc"),

jsonObj.getJSONObject("common").getString("ch"),

jsonObj.getJSONObject("common").getString("ar"),

jsonObj.getJSONObject("common").getString("is\_new"),

0L, 0L, 1L, 0L, 0L, jsonObj.getLong("ts"));

out.collect(visitorStats);

}

}

});

//2.4 转换跳转流

SingleOutputStreamOperator<VisitorStats> userJumpStatDstream = userJumpDStream.map(json -> {

JSONObject jsonObj = JSON.parseObject(json);

return new VisitorStats("", "",

jsonObj.getJSONObject("common").getString("vc"),

jsonObj.getJSONObject("common").getString("ch"),

jsonObj.getJSONObject("common").getString("ar"),

jsonObj.getJSONObject("common").getString("is\_new"),

0L, 0L, 0L, 1L, 0L, jsonObj.getLong("ts"));

});

#### 四条流合并起来

//TODO 3.将四条流合并起来

DataStream<VisitorStats> unionDetailDstream = uniqueVisitStatsDstream.union(

pageViewStatsDstream,

sessionVisitDstream,

userJumpStatDstream

);

### 根据维度进行聚合

#### 设置时间标记及水位线

因为涉及开窗聚合，所以要设定事件时间及水位线

//TODO 4.设置水位线

SingleOutputStreamOperator<VisitorStats> visitorStatsWithWatermarkDstream =

unionDetailDstream.assignTimestampsAndWatermarks(

WatermarkStrategy.<VisitorStats>forBoundedOutOfOrderness(Duration.ofSeconds(1)).

withTimestampAssigner( (visitorStats,ts) ->visitorStats.getTs() )

) ;

visitorStatsWithWatermarkDstream.print("after union:::");

#### 分组

分组选取四个维度作为key , 使用Tuple4组合

//TODO 5.分组 选取四个维度作为key , 使用Tuple4组合

KeyedStream<VisitorStats, Tuple4<String, String, String, String>> visitorStatsTuple4KeyedStream =

visitorStatsWithWatermarkDstream

.keyBy(new KeySelector<VisitorStats, Tuple4<String, String, String, String>>() {

@Override

public Tuple4<String, String, String, String> getKey(VisitorStats visitorStats) throws Exception {

return new Tuple4<>(visitorStats.getVc()

, visitorStats.getCh(),

visitorStats.getAr(),

visitorStats.getIs\_new());

}

}

);

#### 开窗

//TODO 6.开窗

WindowedStream<VisitorStats, Tuple4<String, String, String, String>, TimeWindow> windowStream =

visitorStatsTuple4KeyedStream.window(TumblingEventTimeWindows.of(Time.seconds(10)));

#### 窗口内聚合及补充时间字段

//TODO 7.Reduce聚合统计

SingleOutputStreamOperator<VisitorStats> visitorStatsDstream =

windowStream.reduce(new ReduceFunction<VisitorStats>() {

@Override

public VisitorStats reduce(VisitorStats stats1, VisitorStats stats2) throws Exception {

//把度量数据两两相加

stats1.setPv\_ct(stats1.getPv\_ct() + stats2.getPv\_ct());

stats1.setUv\_ct(stats1.getUv\_ct() + stats2.getUv\_ct());

stats1.setUj\_ct(stats1.getUj\_ct() + stats2.getUj\_ct());

stats1.setSv\_ct(stats1.getSv\_ct() + stats2.getSv\_ct());

stats1.setDur\_sum(stats1.getDur\_sum() + stats2.getDur\_sum());

return stats1;

}

}, new ProcessWindowFunction<VisitorStats, VisitorStats, Tuple4<String, String, String, String>, TimeWindow>() {

@Override

public void process(Tuple4<String, String, String, String> tuple4, Context context,

Iterable<VisitorStats> visitorStatsIn,

Collector<VisitorStats> visitorStatsOut) throws Exception {

//补时间字段

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

for (VisitorStats visitorStats : visitorStatsIn) {

String startDate = simpleDateFormat.format(new Date(context.window().getStart()));

String endDate = simpleDateFormat.format(new Date(context.window().getEnd()));

visitorStats.setStt(startDate);

visitorStats.setEdt(endDate);

visitorStatsOut.collect(visitorStats);

}

}

});

visitorStatsDstream.print("reduce:");

### 写入OLAP数据库

为何要写入ClickHouse数据库，ClickHouse数据库作为专门解决大量数据统计分析的数据库，在保证了海量数据存储的能力，同时又兼顾了响应速度。而且还支持标准SQL，即灵活又易上手。

ClickHouse数据库的详细安装及入门，请参考《05\_尚硅谷大数据之FLINK版实时项目\_ClickHouse\_V1.1》。

#### ClickHouse数据表准备

create table visitor\_stats\_2021 (

stt DateTime,

edt DateTime,

vc String,

ch String,

ar String,

is\_new String,

uv\_ct UInt64,

pv\_ct UInt64,

sv\_ct UInt64,

uj\_ct UInt64,

dur\_sum UInt64,

ts UInt64

) engine =ReplacingMergeTree(ts)

partition by toYYYYMMDD(stt)

order by (stt,edt,is\_new,vc,ch,ar);

之所以选用ReplacingMergeTree引擎主要是靠它来保证数据表的幂等性。

* paritition by 把日期变为数字类型（如：20201126），用于分区。所以尽量保证查询条件尽量包含stt字段。
* order by 后面字段数据在同一分区下，出现重复会被去重，重复数据保留ts最大的数据。

#### 加入ClickHouse依赖包

<dependency>

<groupId>ru.yandex.clickhouse</groupId>

<artifactId>clickhouse-jdbc</artifactId>

<version>0.2.4</version>

<exclusions>

<exclusion>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

</exclusion>

<exclusion>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-connector-jdbc\_${scala.version}</artifactId>

<version>${flink.version}</version>

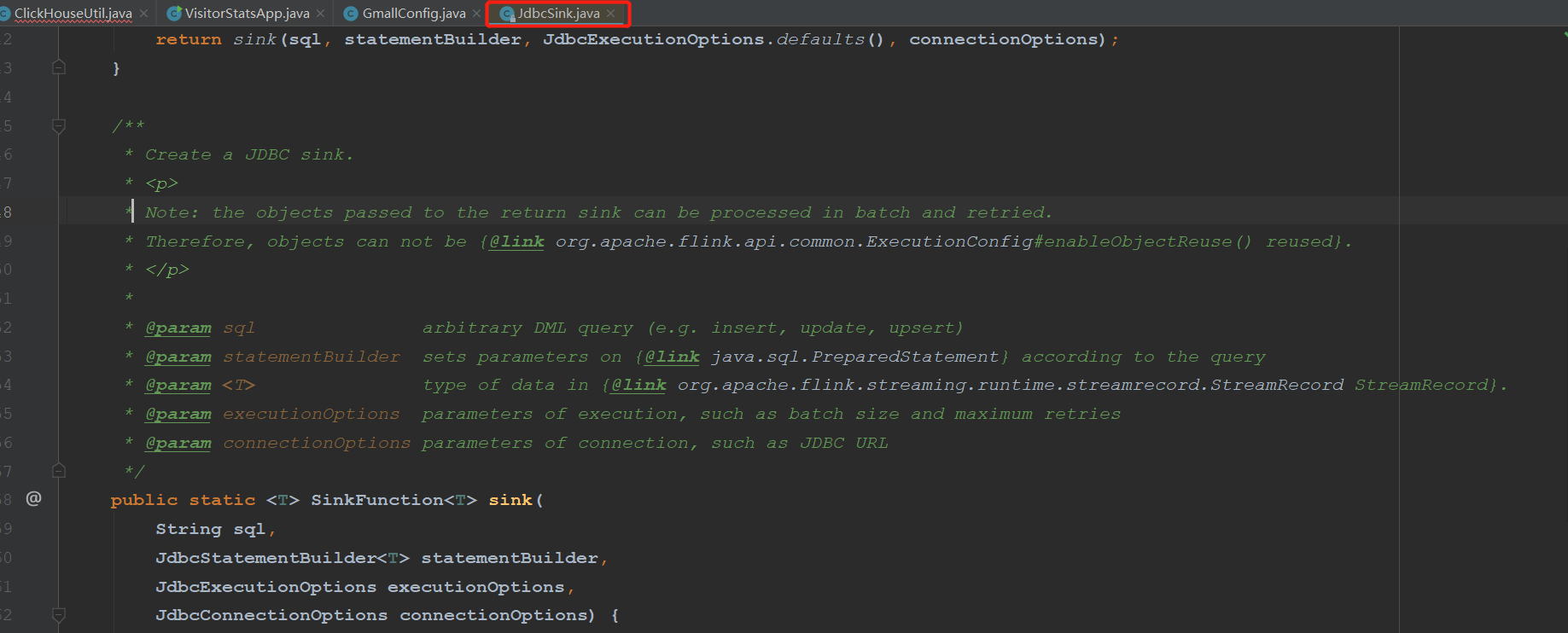
</dependency>

其中flink-connector-jdbc 是官方通用的jdbcSink包。只要引入对应的jdbc驱动，flink可以用它应对各种支持jdbc的数据库，比如phoenix也可以用它。但是这个jdbc-sink只支持数据流对应一张数据表。如果是一流对多表，就必须通过自定义的方式实现了，比如之前的维度数据。

虽然这种jdbc-sink只能一流对一表，但是由于内部使用了预编译器，所以可以实现批量提交以优化写入速度。

#### 增加ClickhouseUtil

##### JdbcSink.<T>sink( )的四个参数说明



* 参数1： 传入Sql，格式如：insert into xxx values(?,?,?,?)
* 参数2: 可以用lambda表达实现(jdbcPreparedStatement, t) -> t为数据对象，要装配到语句预编译器的参数中。
* 参数3：设定一些执行参数，比如重试次数，批次大小。
* 参数4：设定连接参数，比如地址，端口，驱动名。

##### ClickhouseUtil中获取JdbcSink函数的实现

package com.atguigu.gmall.realtime.utils;

import com.atguigu.gmall.realtime.bean.TransientSink;

import com.atguigu.gmall.realtime.common.GmallConfig;

import org.apache.flink.connector.jdbc.JdbcConnectionOptions;

import org.apache.flink.connector.jdbc.JdbcExecutionOptions;

import org.apache.flink.connector.jdbc.JdbcSink;

import org.apache.flink.streaming.api.functions.sink.SinkFunction;

import java.lang.reflect.Field;

/\*\*

\* Desc: 操作ClickHouse的工具类

\*/

public class ClickHouseUtil {

//获取针对ClickHouse的JdbcSink

public static <T> SinkFunction getJdbcSink(String sql) {

SinkFunction<T> sink = JdbcSink.<T>sink(

sql,

(jdbcPreparedStatement, t) -> {

Field[] fields = t.getClass().getDeclaredFields();

int skipOffset = 0; //

for (int i = 0; i < fields.length; i++) {

Field field = fields[i];

//通过反射获得字段上的注解

TransientSink transientSink =

field.getAnnotation(TransientSink.class);

if (transientSink != null) {

// 如果存在该注解

System.out.println("跳过字段：" + field.getName());

skipOffset++;

continue;

}

field.setAccessible(true);

try {

Object o = field.get(t);

//i代表流对象字段的下标，

// 公式：写入表字段位置下标 = 对象流对象字段下标 + 1 - 跳过字段的偏移量

// 一旦跳过一个字段 那么写入字段下标就会和原本字段下标存在偏差

jdbcPreparedStatement.setObject(i + 1 - skipOffset, o);

} catch (Exception e) {

e.printStackTrace();

}

}

},

new JdbcExecutionOptions.Builder().withBatchSize(2).build(),

new JdbcConnectionOptions.JdbcConnectionOptionsBuilder()

.withUrl(GmallConfig.CLICKHOUSE\_URL)

.withDriverName("ru.yandex.clickhouse.ClickHouseDriver")

.build());

return sink;

}

}

##### 创建TransientSink注解，该注解标记不需要保存的字段

package com.atguigu.gmall.realtime.bean;

import java.lang.annotation.Retention;

import java.lang.annotation.Target;

import static java.lang.annotation.ElementType.FIELD;

import static java.lang.annotation.RetentionPolicy.RUNTIME;

/\*\*

\* Desc: 向ClickHouse写入数据的时候，如果有字段数据不需要传输，可以用该注解标记

\*/

@Target(FIELD)

@Retention(RUNTIME)

public @interface TransientSink {

}

由于之前的ClickhouseUtil工具类的写入机制就是把该实体类的所有字段按次序一次写入数据表。但是实体类有时会用到一些临时字段，计算中有用但是并不需要最终保存在临时表中。我们可以把这些字段做一些标识，然后再写入的时候判断标识来过滤掉这些字段。

为字段打标识通常的办法就是给字段加个注解，这里我们就增加一个自定义注解@TransientSink来标识该字段不需要保存到数据表中。

##### 在GmallConfig中配置ClickHouse的连接地址

package com.atguigu.gmall.realtime.common;

/\*\*

\* Desc: 项目常用配置

\*/

public class GmallConfig {

public static final String HBASE\_SCHEMA="GMALL2021\_REALTIME";

public static final String PHOENIX\_SERVER="jdbc:phoenix:hadoop102,hadoop103,hadoop104:2181";

public static final String CLICKHOUSE\_URL="jdbc:clickhouse://hadoop102:8123/default";

}

#### 为主程序增加写入ClickHouse的Sink

//TODO 8.向ClickHouse中写入数据

visitorStatsDstream.addSink(

ClickHouseUtil.getJdbcSink("insert into visitor\_stats\_2021 values(?,?,?,?,?,?,?,?,?,?,?,?)"));

### 整体测试

* 启动ZK、Kafka、logger.sh、ClickHouse、【HDFS】
* 运行BaseLogApp
* 运行UniqueVisitApp
* 运行UserJumpDetailApp
* 运行VisitorStatsApp
* 运行rt\_applog目录下的jar包
* 查看控制台输出
* 查看ClickHouse中visitor\_stats\_2021表数据

# DWS层-商品主题宽表的计算

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 统计主题 | 需求指标 | 输出方式 | 计算来源 | 来源层级 |
| 商品 | 点击 | 多维分析 | page\_log直接可求 | dwd |
| 曝光 | 多维分析 | page\_log直接可求 | dwd |
| 收藏 | 多维分析 | 收藏表 | dwd |
| 加入购物车 | 多维分析 | 购物车表 | dwd |
| 下单 | 可视化大屏 | 订单宽表 | dwm |
| 支付 | 多维分析 | 支付宽表 | dwm |
| 退款 | 多维分析 | 退款表 | dwd |
| 评价 | 多维分析 | 评价表 | dwd |

与访客的dws层的宽表类似，也是把多个事实表的明细数据汇总起来组合成宽表。

## 需求分析与思路

* 从Kafka主题中获得数据流
* 把Json字符串数据流转换为统一数据对象的数据流
* 把统一的数据结构流合并为一个流
* 设定事件时间与水位线
* 分组、开窗、聚合
* 关联维度补充数据
* 写入ClickHouse

## 功能实现

### 封装商品统计实体类ProductStats

package com.atguigu.gmall.realtime.bean;

/\*\*

\* Desc: 商品统计实体类

\* @Builder注解

\* 可以使用构造者方式创建对象，给属性赋值

\* @Builder.Default

\* 在使用构造者方式给属性赋值的时候，属性的初始值会丢失

\* 该注解的作用就是修复这个问题

\* 例如：我们在属性上赋值了初始值为0L，如果不加这个注解，通过构造者创建的对象属性值会变为null

\*/

import lombok.Builder;

import lombok.Data;

import java.math.BigDecimal;

import java.util.HashSet;

import java.util.Set;

@Data

@Builder

public class ProductStats {

String stt;//窗口起始时间

String edt; //窗口结束时间

Long sku\_id; //sku编号

String sku\_name;//sku名称

BigDecimal sku\_price; //sku单价

Long spu\_id; //spu编号

String spu\_name;//spu名称

Long tm\_id; //品牌编号

String tm\_name;//品牌名称

Long category3\_id;//品类编号

String category3\_name;//品类名称

@Builder.Default

Long display\_ct = 0L; //曝光数

@Builder.Default

Long click\_ct = 0L; //点击数

@Builder.Default

Long favor\_ct = 0L; //收藏数

@Builder.Default

Long cart\_ct = 0L; //添加购物车数

@Builder.Default

Long order\_sku\_num = 0L; //下单商品个数

@Builder.Default //下单商品金额

BigDecimal order\_amount = BigDecimal.ZERO;

@Builder.Default

Long order\_ct = 0L; //订单数

@Builder.Default //支付金额

BigDecimal payment\_amount = BigDecimal.ZERO;

@Builder.Default

Long paid\_order\_ct = 0L; //支付订单数

@Builder.Default

Long refund\_order\_ct = 0L; //退款订单数

@Builder.Default

BigDecimal refund\_amount = BigDecimal.ZERO;

@Builder.Default

Long comment\_ct = 0L;//评论订单数

@Builder.Default

Long good\_comment\_ct = 0L; //好评订单数

@Builder.Default

@TransientSink

Set orderIdSet = new HashSet(); //用于统计订单数

@Builder.Default

@TransientSink

Set paidOrderIdSet = new HashSet(); //用于统计支付订单数

@Builder.Default

@TransientSink

Set refundOrderIdSet = new HashSet();//用于退款支付订单数

Long ts; //统计时间戳

}

### 创建ProductStatsApp，从Kafka主题中获得数据流

package com.atguigu.gmall.realtime.app.dws;

import com.atguigu.gmall.realtime.utils.MyKafkaUtil;

import org.apache.flink.streaming.api.datastream.DataStreamSource;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

/\*\*

\* Desc: 形成以商品为准的统计 曝光 点击 购物车 下单 支付 退单 评论数 宽表

\*/

public class ProductStatsApp {

public static void main(String[] args) throws Exception {

//TODO 0.基本环境准备

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

//设置并行度

env.setParallelism(4);

/\*

//检查点CK相关设置

env.enableCheckpointing(5000, CheckpointingMode.AT\_LEAST\_ONCE);

env.getCheckpointConfig().setCheckpointTimeout(60000);

StateBackend fsStateBackend = new FsStateBackend(

"hdfs://hadoop102:8020/gmall/flink/checkpoint/ProductStatsApp");

env.setStateBackend(fsStateBackend);

System.setProperty("HADOOP\_USER\_NAME","atguigu");

\*/

//TODO 1.从Kafka中获取数据流

String groupId = "product\_stats\_app";

String pageViewSourceTopic = "dwd\_page\_log";

String orderWideSourceTopic = "dwm\_order\_wide";

String paymentWideSourceTopic = "dwm\_payment\_wide";

String cartInfoSourceTopic = "dwd\_cart\_info";

String favorInfoSourceTopic = "dwd\_favor\_info";

String refundInfoSourceTopic = "dwd\_order\_refund\_info";

String commentInfoSourceTopic = "dwd\_comment\_info";

FlinkKafkaConsumer<String> pageViewSource = MyKafkaUtil.getKafkaSource(pageViewSourceTopic,groupId);

FlinkKafkaConsumer<String> orderWideSource = MyKafkaUtil.getKafkaSource(orderWideSourceTopic,groupId);

FlinkKafkaConsumer<String> paymentWideSource = MyKafkaUtil.getKafkaSource(paymentWideSourceTopic,groupId);

FlinkKafkaConsumer<String> favorInfoSourceSouce = MyKafkaUtil.getKafkaSource(favorInfoSourceTopic,groupId);

FlinkKafkaConsumer<String> cartInfoSource = MyKafkaUtil.getKafkaSource(cartInfoSourceTopic,groupId);

FlinkKafkaConsumer<String> refundInfoSource = MyKafkaUtil.getKafkaSource(refundInfoSourceTopic,groupId);

FlinkKafkaConsumer<String> commentInfoSource = MyKafkaUtil.getKafkaSource(commentInfoSourceTopic,groupId);

DataStreamSource<String> pageViewDStream = env.addSource(pageViewSource);

DataStreamSource<String> favorInfoDStream = env.addSource(favorInfoSourceSouce);

DataStreamSource<String> orderWideDStream= env.addSource(orderWideSource);

DataStreamSource<String> paymentWideDStream= env.addSource(paymentWideSource);

DataStreamSource<String> cartInfoDStream= env.addSource(cartInfoSource);

DataStreamSource<String> refundInfoDStream= env.addSource(refundInfoSource);

DataStreamSource<String> commentInfoDStream= env.addSource(commentInfoSource);

env.execute();

}

}

### 把JSON字符串数据流转换为统一数据对象的数据流

//TODO 2.对获取的流数据进行结构的转换

//2.1转换曝光及页面流数据

SingleOutputStreamOperator<ProductStats> pageAndDispStatsDstream = pageViewDStream.process(

new ProcessFunction<String, ProductStats>() {

@Override

public void processElement(String json, Context ctx, Collector<ProductStats> out) throws Exception {

JSONObject jsonObj = JSON.parseObject(json);

JSONObject pageJsonObj = jsonObj.getJSONObject("page");

String pageId = pageJsonObj.getString("page\_id");

if (pageId == null) {

System.out.println(jsonObj);

}

Long ts = jsonObj.getLong("ts");

if (pageId.equals("good\_detail")) {

Long skuId = pageJsonObj.getLong("item");

ProductStats productStats = ProductStats.builder().sku\_id(skuId).

click\_ct(1L).ts(ts).build();

out.collect(productStats);

}

JSONArray displays = jsonObj.getJSONArray("display");

if (displays != null && displays.size() > 0) {

for (int i = 0; i < displays.size(); i++) {

JSONObject display = displays.getJSONObject(i);

if (display.getString("item\_type").equals("sku\_id")) {

Long skuId = display.getLong("item");

ProductStats productStats = ProductStats.builder()

.sku\_id(skuId).display\_ct(1L).ts(ts).build();

out.collect(productStats);

}

}

}

}

});

//2.2转换下单流数据

SingleOutputStreamOperator<ProductStats> orderWideStatsDstream = orderWideDStream.map(

json -> {

OrderWide orderWide = JSON.parseObject(json, OrderWide.class);

System.out.println("orderWide:===" + orderWide);

String create\_time = orderWide.getCreate\_time();

Long ts = DateTimeUtil.toTs(create\_time);

return ProductStats.builder().sku\_id(orderWide.getSku\_id())

.orderIdSet(new HashSet(Collections.singleton(orderWide.getOrder\_id())))

.order\_sku\_num(orderWide.getSku\_num())

.order\_amount(orderWide.getSplit\_total\_amount()).ts(ts).build();

});

//2.3转换收藏流数据

SingleOutputStreamOperator<ProductStats> favorStatsDstream = favorInfoDStream.map(

json -> {

JSONObject favorInfo = JSON.parseObject(json);

Long ts = DateTimeUtil.toTs(favorInfo.getString("create\_time"));

return ProductStats.builder().sku\_id(favorInfo.getLong("sku\_id"))

.favor\_ct(1L).ts(ts).build();

});

//2.4转换购物车流数据

SingleOutputStreamOperator<ProductStats> cartStatsDstream = cartInfoDStream.map(

json -> {

JSONObject cartInfo = JSON.parseObject(json);

Long ts = DateTimeUtil.toTs(cartInfo.getString("create\_time"));

return ProductStats.builder().sku\_id(cartInfo.getLong("sku\_id"))

.cart\_ct(1L).ts(ts).build();

});

//2.5转换支付流数据

SingleOutputStreamOperator<ProductStats> paymentStatsDstream = paymentWideDStream.map(

json -> {

PaymentWide paymentWide = JSON.parseObject(json, PaymentWide.class);

Long ts = DateTimeUtil.toTs(paymentWide.getPayment\_create\_time());

return ProductStats.builder().sku\_id(paymentWide.getSku\_id())

.payment\_amount(paymentWide.getSplit\_total\_amount())

.paidOrderIdSet(new HashSet(Collections.singleton(paymentWide.getOrder\_id())))

.ts(ts).build();

});

//2.6转换退款流数据

SingleOutputStreamOperator<ProductStats> refundStatsDstream = refundInfoDStream.map(

json -> {

JSONObject refundJsonObj = JSON.parseObject(json);

Long ts = DateTimeUtil.toTs(refundJsonObj.getString("create\_time"));

ProductStats productStats = ProductStats.builder()

.sku\_id(refundJsonObj.getLong("sku\_id"))

.refund\_amount(refundJsonObj.getBigDecimal("refund\_amount"))

.refundOrderIdSet(

new HashSet(Collections.singleton(refundJsonObj.getLong("order\_id"))))

.ts(ts).build();

return productStats;

});

//2.7转换评价流数据

SingleOutputStreamOperator<ProductStats> commonInfoStatsDstream = commentInfoDStream.map(

json -> {

JSONObject commonJsonObj = JSON.parseObject(json);

Long ts = DateTimeUtil.toTs(commonJsonObj.getString("create\_time"));

Long goodCt = GmallConstant.APPRAISE\_GOOD.equals(commonJsonObj.getString("appraise")) ? 1L : 0L;

ProductStats productStats = ProductStats.builder()

.sku\_id(commonJsonObj.getLong("sku\_id"))

.comment\_ct(1L).good\_comment\_ct(goodCt).ts(ts).build();

return productStats;

});

### 创建电商业务常量类GmallConstant

package com.atguigu.gmall.realtime.common;

/\*\*

\* Desc: 电商业务常量

\*/

public class GmallConstant {

//10 单据状态

public static final String ORDER\_STATUS\_UNPAID="1001"; //未支付

public static final String ORDER\_STATUS\_PAID="1002"; //已支付

public static final String ORDER\_STATUS\_CANCEL="1003";//已取消

public static final String ORDER\_STATUS\_FINISH="1004";//已完成

public static final String ORDER\_STATUS\_REFUND="1005";//退款中

public static final String ORDER\_STATUS\_REFUND\_DONE="1006";//退款完成

//11 支付状态

public static final String PAYMENT\_TYPE\_ALIPAY="1101";//支付宝

public static final String PAYMENT\_TYPE\_WECHAT="1102";//微信

public static final String PAYMENT\_TYPE\_UNION="1103";//银联

//12 评价

public static final String APPRAISE\_GOOD="1201";// 好评

public static final String APPRAISE\_SOSO="1202";// 中评

public static final String APPRAISE\_BAD="1203";// 差评

public static final String APPRAISE\_AUTO="1204";// 自动

//13 退货原因

public static final String REFUND\_REASON\_BAD\_GOODS="1301";// 质量问题

public static final String REFUND\_REASON\_WRONG\_DESC="1302";// 商品描述与实际描述不一致

public static final String REFUND\_REASON\_SALE\_OUT="1303";// 缺货

public static final String REFUND\_REASON\_SIZE\_ISSUE="1304";// 号码不合适

public static final String REFUND\_REASON\_MISTAKE="1305";// 拍错

public static final String REFUND\_REASON\_NO\_REASON="1306";// 不想买了

public static final String REFUND\_REASON\_OTHER="1307";// 其他

//14 购物券状态

public static final String COUPON\_STATUS\_UNUSED="1401";// 未使用

public static final String COUPON\_STATUS\_USING="1402";// 使用中

public static final String COUPON\_STATUS\_USED="1403";// 已使用

//15退款类型

public static final String REFUND\_TYPE\_ONLY\_MONEY="1501";// 仅退款

public static final String REFUND\_TYPE\_WITH\_GOODS="1502";// 退货退款

//24来源类型

public static final String SOURCE\_TYPE\_QUREY="2401";// 用户查询

public static final String SOURCE\_TYPE\_PROMOTION="2402";// 商品推广

public static final String SOURCE\_TYPE\_AUTO\_RECOMMEND="2403";// 智能推荐

public static final String SOURCE\_TYPE\_ACTIVITY="2404";// 促销活动

//购物券范围

public static final String COUPON\_RANGE\_TYPE\_CATEGORY3="3301";//

public static final String COUPON\_RANGE\_TYPE\_TRADEMARK="3302";//

public static final String COUPON\_RANGE\_TYPE\_SPU="3303";//

//购物券类型

public static final String COUPON\_TYPE\_MJ="3201";//满减

public static final String COUPON\_TYPE\_DZ="3202";//满量打折

public static final String COUPON\_TYPE\_DJ="3203";//代金券

public static final String ACTIVITY\_RULE\_TYPE\_MJ="3101";

public static final String ACTIVITY\_RULE\_TYPE\_DZ="3102";

public static final String ACTIVITY\_RULE\_TYPE\_ZK="3103";

public static final String KEYWORD\_SEARCH="SEARCH";

public static final String KEYWORD\_CLICK="CLICK";

public static final String KEYWORD\_CART="CART";

public static final String KEYWORD\_ORDER="ORDER";

}

### 把统一的数据结构流合并为一个流

//TODO 3.把统一的数据结构流合并为一个流

DataStream<ProductStats> productStatDetailDStream = pageAndDispStatsDstream.union(

orderWideStatsDstream, cartStatsDstream,

paymentStatsDstream, refundStatsDstream,favorStatsDstream,

commonInfoStatsDstream);

productStatDetailDStream.print("after union:");

### 设定事件时间与水位线

//TODO 4.设定事件时间与水位线

SingleOutputStreamOperator<ProductStats> productStatsWithTsStream =

productStatDetailDStream.assignTimestampsAndWatermarks(

WatermarkStrategy.<ProductStats>forMonotonousTimestamps().withTimestampAssigner(

(productStats, recordTimestamp) -> {

return productStats.getTs();

})

);

### 分组、开窗、聚合

//TODO 5.分组、开窗、聚合

SingleOutputStreamOperator<ProductStats> productStatsDstream = productStatsWithTsStream

//5.1 按照商品id进行分组

.keyBy(

new KeySelector<ProductStats, Long>() {

@Override

public Long getKey(ProductStats productStats) throws Exception {

return productStats.getSku\_id();

}

})

//5.2 开窗 窗口长度为10s

.window(TumblingEventTimeWindows.of(Time.seconds(10)))

//5.3 对窗口中的数据进行聚合

.reduce(new ReduceFunction<ProductStats>() {

@Override

public ProductStats reduce(ProductStats stats1, ProductStats stats2) throws Exception {

stats1.setDisplay\_ct(stats1.getDisplay\_ct() + stats2.getDisplay\_ct());

stats1.setClick\_ct(stats1.getClick\_ct() + stats2.getClick\_ct());

stats1.setCart\_ct(stats1.getCart\_ct() + stats2.getCart\_ct());

stats1.setFavor\_ct(stats1.getFavor\_ct() + stats2.getFavor\_ct());

stats1.setOrder\_amount(stats1.getOrder\_amount().add(stats2.getOrder\_amount()));

stats1.getOrderIdSet().addAll(stats2.getOrderIdSet());

stats1.setOrder\_ct(stats1.getOrderIdSet().size() + 0L);

stats1.setOrder\_sku\_num(stats1.getOrder\_sku\_num() + stats2.getOrder\_sku\_num());

stats1.setPayment\_amount(stats1.getPayment\_amount().add(stats2.getPayment\_amount()));

stats1.getRefundOrderIdSet().addAll(stats2.getRefundOrderIdSet());

stats1.setRefund\_order\_ct(stats1.getRefundOrderIdSet().size() + 0L);

stats1.setRefund\_amount(stats1.getRefund\_amount().add(stats2.getRefund\_amount()));

stats1.getPaidOrderIdSet().addAll(stats2.getPaidOrderIdSet());

stats1.setPaid\_order\_ct(stats1.getPaidOrderIdSet().size() + 0L);

stats1.setComment\_ct(stats1.getComment\_ct() + stats2.getComment\_ct());

stats1.setGood\_comment\_ct(stats1.getGood\_comment\_ct() + stats2.getGood\_comment\_ct());

return stats1;

}

},

new WindowFunction<ProductStats, ProductStats, Long, TimeWindow>() {

@Override

public void apply(Long aLong, TimeWindow window,

Iterable<ProductStats> productStatsIterable,

Collector<ProductStats> out) throws Exception {

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

for (ProductStats productStats : productStatsIterable) {

productStats.setStt(simpleDateFormat.format(window.getStart()));

productStats.setEdt(simpleDateFormat.format(window.getEnd()));

productStats.setTs(new Date().getTime());

out.collect(productStats);

}

}

});

//productStatsDstream.print("productStatsDstream::");

### 补充商品维度信息

因为除了下单操作之外，其它操作，只获取到了商品的id，其它维度信息是没有的

### 写入ClickHouse

//TODO 6.补充商品维度信息

//6.1 补充SKU维度

SingleOutputStreamOperator<ProductStats> productStatsWithSkuDstream =

AsyncDataStream.unorderedWait(productStatsDstream,

new DimAsyncFunction<ProductStats>("DIM\_SKU\_INFO") {

@Override

public void join(ProductStats productStats, JSONObject jsonObject) throws Exception {

productStats.setSku\_name(jsonObject.getString("SKU\_NAME"));

productStats.setSku\_price(jsonObject.getBigDecimal("PRICE"));

productStats.setCategory3\_id(jsonObject.getLong("CATEGORY3\_ID"));

productStats.setSpu\_id(jsonObject.getLong("SPU\_ID"));

productStats.setTm\_id(jsonObject.getLong("TM\_ID"));

}

@Override

public String getKey(ProductStats productStats) {

return String.valueOf(productStats.getSku\_id());

}

}, 60, TimeUnit.SECONDS);

//6.2 补充SPU维度

SingleOutputStreamOperator<ProductStats> productStatsWithSpuDstream =

AsyncDataStream.unorderedWait(productStatsWithSkuDstream,

new DimAsyncFunction<ProductStats>("DIM\_SPU\_INFO") {

@Override

public void join(ProductStats productStats, JSONObject jsonObject) throws Exception {

productStats.setSpu\_name(jsonObject.getString("SPU\_NAME"));

}

@Override

public String getKey(ProductStats productStats) {

return String.valueOf(productStats.getSpu\_id());

}

}, 60, TimeUnit.SECONDS);

//6.3 补充品类维度

SingleOutputStreamOperator<ProductStats> productStatsWithCategory3Dstream =

AsyncDataStream.unorderedWait(productStatsWithSpuDstream,

new DimAsyncFunction<ProductStats>("DIM\_BASE\_CATEGORY3") {

@Override

public void join(ProductStats productStats, JSONObject jsonObject) throws Exception {

productStats.setCategory3\_name(jsonObject.getString("NAME"));

}

@Override

public String getKey(ProductStats productStats) {

return String.valueOf(productStats.getCategory3\_id());

}

}, 60, TimeUnit.SECONDS);

//6.4 补充品牌维度

SingleOutputStreamOperator<ProductStats> productStatsWithTmDstream =

AsyncDataStream.unorderedWait(productStatsWithCategory3Dstream,

new DimAsyncFunction<ProductStats>("DIM\_BASE\_TRADEMARK") {

@Override

public void join(ProductStats productStats, JSONObject jsonObject) throws Exception {

productStats.setTm\_name(jsonObject.getString("TM\_NAME"));

}

@Override

public String getKey(ProductStats productStats) {

return String.valueOf(productStats.getTm\_id());

}

}, 60, TimeUnit.SECONDS);

productStatsWithTmDstream.print("to save");

#### 在ClickHouse中创建商品主题宽表

create table product\_stats\_2021 (

stt DateTime,

edt DateTime,

sku\_id UInt64,

sku\_name String,

sku\_price Decimal64(2),

spu\_id UInt64,

spu\_name String ,

tm\_id UInt64,

tm\_name String,

category3\_id UInt64,

category3\_name String ,

display\_ct UInt64,

click\_ct UInt64,

favor\_ct UInt64,

cart\_ct UInt64,

order\_sku\_num UInt64,

order\_amount Decimal64(2),

order\_ct UInt64 ,

payment\_amount Decimal64(2),

paid\_order\_ct UInt64,

refund\_order\_ct UInt64,

refund\_amount Decimal64(2),

comment\_ct UInt64,

good\_comment\_ct UInt64 ,

ts UInt64

)engine =ReplacingMergeTree(ts)

partition by toYYYYMMDD(stt)

order by (stt,edt,sku\_id );

#### 为主程序增加写入ClickHouse的Sink

//TODO 7.写入到ClickHouse

productStatsWithTmDstream.addSink(

ClickHouseUtil.<ProductStats>getJdbcSink(

"insert into product\_stats\_2021 values(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)"));

### 整体测试

* 启动ZK、Kafka、logger.sh、ClickHouse、Redis、HDFS、HBase、FlinkCDC
* 运行BaseLogApp
* 运行BaseDBApp
* 运行OrderWideApp
* 运行PaymentWideApp
* 运行ProductsStatsApp
* 运行rt\_applog目录下的jar包
* 运行rt\_dblog目录下的jar包
* 查看控制台输出
* 查看ClickHouse中products\_stats\_2021表数据

注意：一定要匹配两个数据生成模拟器的日期，否则窗口无法匹配上

# DWS层-地区主题表(FlinkSQL)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 统计主题 | 需求指标 | 输出方式 | 计算来源 | 来源层级 |
| 地区 | pv | 多维分析 | page\_log直接可求 | dwd |
| uv | 多维分析 | 需要用page\_log过滤去重 | dwm |
| 下单（单数，金额） | 可视化大屏 | 订单宽表 | dwm |

地区主题主要是反映各个地区的销售情况。从业务逻辑上地区主题比起商品更加简单，业务逻辑也没有什么特别的就是做一次轻度聚合然后保存，所以在这里我们体验一下使用FlinkSQL，来完成该业务。

## 需求分析与思路

* 定义Table流环境
* 把数据源定义为动态表
* 通过SQL查询出结果表
* 把结果表转换为数据流
* 把数据流写入目标数据库

如果是Flink官方支持的数据库，也可以直接把目标数据表定义为动态表，用insert into 写入。由于ClickHouse目前官方没有支持的jdbc连接器（目前支持Mysql、 PostgreSQL、Derby）。也可以制作自定义sink，实现官方不支持的连接器。但是比较繁琐。

## 功能实现

### 在pom.xml文件中添加FlinkSQL相关依赖

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-table-api-java-bridge\_${scala.version}</artifactId>

<version>${flink.version}</version>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-table-planner-blink\_${scala.version}</artifactId>

<version>${flink.version}</version>

</dependency>

### 创建ProvinceStatsSqlApp,定义Table流环境

package com.atguigu.gmall.realtime.app.dws;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.table.api.EnvironmentSettings;

import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/\*\*

\* Desc: FlinkSQL实现地区主题宽表计算

\*/

public class ProvinceStatsSqlApp {

public static void main(String[] args) throws Exception {

//TODO 0.基本环境准备

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

//设置并行度

env.setParallelism(4);

/\*

//CK相关设置

env.enableCheckpointing(5000, CheckpointingMode.AT\_LEAST\_ONCE);

env.getCheckpointConfig().setCheckpointTimeout(60000);

StateBackend fsStateBackend = new FsStateBackend(

"hdfs://hadoop102:8020/gmall/flink/checkpoint/ProvinceStatsSqlApp");

env.setStateBackend(fsStateBackend);

System.setProperty("HADOOP\_USER\_NAME","atguigu");

\*/

//TODO 1.定义Table流环境

EnvironmentSettings settings = EnvironmentSettings

.newInstance()

.inStreamingMode()

.build();

StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings);

env.execute();

}

}

### 把数据源定义为动态表

其中WATERMARK FOR rowtime AS rowtime是把某个虚拟字段设定为EVENT\_TIME

//TODO 2.把数据源定义为动态表

String groupId = "province\_stats";

String orderWideTopic = "dwm\_order\_wide";

tableEnv.executeSql("CREATE TABLE ORDER\_WIDE (province\_id BIGINT, " +

"province\_name STRING,province\_area\_code STRING" +

",province\_iso\_code STRING,province\_3166\_2\_code STRING,order\_id STRING, " +

"total\_amount DOUBLE,create\_time STRING,rowtime AS TO\_TIMESTAMP(create\_time) ," +

"WATERMARK FOR rowtime AS rowtime)" +

" WITH (" + MyKafkaUtil.getKafkaDDL(orderWideTopic, groupId) + ")");

### MyKafkaUtil增加一个DDL的方法

//拼接Kafka相关属性到DDL

public static String getKafkaDDL(String topic,String groupId){

String ddl="'connector' = 'kafka', " +

" 'topic' = '"+topic+"'," +

" 'properties.bootstrap.servers' = '"+ kafkaServer +"', " +

" 'properties.group.id' = '"+groupId+ "', " +

" 'format' = 'json', " +

" 'scan.startup.mode' = 'latest-offset' ";

return ddl;

}

### 聚合计算

//TODO 3.聚合计算

Table provinceStateTable = tableEnv.sqlQuery("select " +

"DATE\_FORMAT(TUMBLE\_START(rowtime, INTERVAL '10' SECOND ),'yyyy-MM-dd HH:mm:ss') stt, " +

"DATE\_FORMAT(TUMBLE\_END(rowtime, INTERVAL '10' SECOND ),'yyyy-MM-dd HH:mm:ss') edt , " +

" province\_id,province\_name,province\_area\_code," +

"province\_iso\_code,province\_3166\_2\_code," +

"COUNT( DISTINCT order\_id) order\_count, sum(split\_total\_amount) order\_amount," +

"UNIX\_TIMESTAMP()\*1000 ts "+

" from ORDER\_WIDE group by TUMBLE(rowtime, INTERVAL '10' SECOND )," +

" province\_id,province\_name,province\_area\_code,province\_iso\_code,province\_3166\_2\_code ");

### 转为数据流

//TODO 4.转换为数据流

DataStream<ProvinceStats> provinceStatsDataStream =

tableEnv.toAppendStream(provinceStateTable, ProvinceStats.class);

### 定义地区统计宽表实体类ProvinceStats

package com.atguigu.gmall.realtime.bean;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import java.math.BigDecimal;

import java.util.Date;

/\*\*

\* Desc:地区统计宽表实体类

\*/

@AllArgsConstructor

@NoArgsConstructor

@Data

public class ProvinceStats {

private String stt;

private String edt;

private Long province\_id;

private String province\_name;

private String province\_area\_code;

private String province\_iso\_code;

private String province\_3166\_2\_code;

private BigDecimal order\_amount;

private Long order\_count;

private Long ts;

public ProvinceStats(OrderWide orderWide){

province\_id = orderWide.getProvince\_id();

order\_amount = orderWide.getSplit\_total\_amount();

province\_name=orderWide.getProvince\_name();

province\_area\_code=orderWide.getProvince\_area\_code();

province\_iso\_code=orderWide.getProvince\_iso\_code();

province\_3166\_2\_code=orderWide.getProvince\_3166\_2\_code();

order\_count = 1L;

ts=new Date().getTime();

}

}

### 在ClickHouse中创建地区主题宽表

create table province\_stats\_201109 (

stt DateTime,

edt DateTime,

province\_id UInt64,

province\_name String,

area\_code String,

iso\_code String,

iso\_3166\_2 String,

order\_amount Decimal64(2),

order\_count UInt64,

ts UInt64

)engine =ReplacingMergeTree(ts)

partition by toYYYYMMDD(stt)

order by (stt,edt,province\_id);

### 写入ClickHouse

//TODO 5.写入到lickHouse

provinceStatsDataStream.addSink(ClickHouseUtil.

<ProvinceStats>getJdbcSink("insert into province\_stats\_2021 values(?,?,?,?,?,?,?,?,?,?)"));

### 整体测试

* 启动ZK、Kafka、ClickHouse、Redis、HDFS、HBase、FlinkCDC
* 运行BaseDBApp
* 运行OrderWideApp
* 运行ProvinceStatsSqlApp
* 运行rt\_dblog目录下的jar包
* 查看控制台输出
* 查看ClickHouse中products\_stats\_2021表数据

注意：因为是事件时间，所以第一次运行rt\_dblog的时候，不会触发watermark，第二次再运行rt\_dblog的jar的时候，才会触发第一次运行的watermark，并且两次运行rt\_dblog时间间隔需要在开窗时间以上。

# DWS层-关键词主题表(FlinkSQL)

## 需求分析与思路



关键词主题这个主要是为了大屏展示中的字符云的展示效果，用于感性的让大屏观看者感知目前的用户都更关心的那些商品和关键词。

关键词的展示也是一种维度聚合的结果，根据聚合的大小来决定关键词的大小。

关键词的第一重要来源的就是用户在搜索栏的搜索，另外就是从以商品为主题的统计中获取关键词。

### 关于分词

因为无论是从用户的搜索栏中，还是从商品名称中文字都是可能是比较长的，且由多个关键词组成，如下图。

所以我们需要根据把长文本分割成一个一个的词，这种分词技术，在搜索引擎中可能会用到。对于中文分词，现在的搜索引擎基本上都是使用的第三方分词器，咱们在计算数据中也可以，使用和搜索引擎中一致的分词器，IK。

## 搜索关键词功能实现

### IK分词器的使用

#### 在pom.xml中加入依赖

<dependency>

<groupId>com.janeluo</groupId>

<artifactId>ikanalyzer</artifactId>

<version>2012\_u6</version>

</dependency>

#### 封装分词工具类并进行测试

package com.atguigu.gmall.realtime.utils;

import org.wltea.analyzer.core.IKSegmenter;

import org.wltea.analyzer.core.Lexeme;

import java.io.IOException;

import java.io.StringReader;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Desc: IK分词器工具类

\*/

public class KeywordUtil {

//使用IK分词器对字符串进行分词

public static List<String> analyze(String text) {

StringReader sr = new StringReader(text);

IKSegmenter ik = new IKSegmenter(sr, true);

Lexeme lex = null;

List<String> keywordList = new ArrayList();

while (true) {

try {

if ((lex = ik.next()) != null) {

String lexemeText = lex.getLexemeText();

keywordList.add(lexemeText);

} else {

break;

}

} catch (IOException e) {

e.printStackTrace();

}

}

return keywordList;

}

public static void main(String[] args) {

String text = "Apple iPhoneXSMax (A2104) 256GB 深空灰色 移动联通电信4G手机 双卡双待";

System.out.println(KeywordUtil.analyze(text));

}

}

### 自定义函数

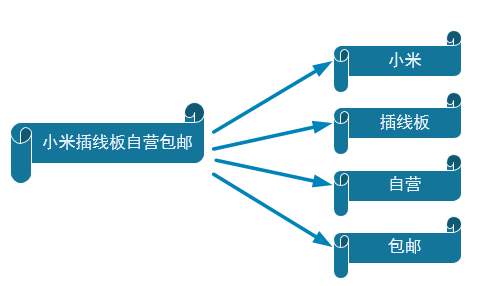
有了分词器，那么另外一个要考虑的问题就是如何把分词器的使用揉进FlinkSQL中。

因为SQL的语法和相关的函数都是Flink内定的，想要使用外部工具，就必须结合自定义函数。

#### 自定义函数分类

* Scalar Function(相当于 Spark的 UDF),
* Table Function(相当于 Spark 的 UDTF),
* Aggregation Functions (相当于 Spark的UDAF)

考虑到一个词条包括多个词语所以分词是指上是一种一对多的拆分，一拆多的情况，我们应该选择Table Function。



#### 封装KeywordUDTF函数

参考<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/functions/udfs.html>

@FunctionHint 主要是为了标识输出数据的类型

row.setField(0,keyword)中的0表示返回值下标为0的值

package com.atguigu.gmall.realtime.app.udf;

import com.atguigu.gmall.realtime.utils.KeywordUtil;

import org.apache.flink.table.annotation.DataTypeHint;

import org.apache.flink.table.annotation.FunctionHint;

import org.apache.flink.table.functions.TableFunction;

import org.apache.flink.types.Row;

import java.util.List;

/\*\*

\* Desc: 自定义UDTF函数实现分词功能

\*/

@FunctionHint(output = @DataTypeHint("ROW<s STRING>"))

public class KeywordUDTF extends TableFunction<Row> {

public void eval(String value) {

List<String> keywordList = KeywordUtil.analyze(value);

for (String keyword : keywordList) {

Row row = new Row(1);

row.setField(0,keyword);

collect(row);

}

}

}

### 创建KeywordStatsApp，定义流环境

package com.atguigu.gmall.realtime.app.dws;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.table.api.EnvironmentSettings;

import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/\*\*

\* Desc: 搜索关键字计算

\*/

public class KeywordStatsApp {

public static void main(String[] args) {

//TODO 0.基本环境准备

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

//设置并行度

env.setParallelism(4);

/\*

//CK相关设置

env.enableCheckpointing(5000, CheckpointingMode.AT\_LEAST\_ONCE);

env.getCheckpointConfig().setCheckpointTimeout(60000);

StateBackend fsStateBackend = new FsStateBackend(

"hdfs://hadoop102:8020/gmall/flink/checkpoint/ProvinceStatsSqlApp");

env.setStateBackend(fsStateBackend);

System.setProperty("HADOOP\_USER\_NAME","atguigu");

\*/

//TODO 1.定义Table流环境

EnvironmentSettings settings = EnvironmentSettings

.newInstance()

.inStreamingMode()

.build();

StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings);

env.execute();

}

}

### 声明动态表和自定义函数

注意json格式的要定义为Map对象

//TODO 2.注册自定义函数

tableEnv.createTemporarySystemFunction("ik\_analyze", KeywordUDTF.class);

//TODO 3.将数据源定义为动态表

String groupId = "keyword\_stats\_app";

String pageViewSourceTopic ="dwd\_page\_log";

tableEnv.executeSql("CREATE TABLE page\_view " +

"(common MAP<STRING,STRING>, " +

"page MAP<STRING,STRING>,ts BIGINT, " +

"rowtime AS TO\_TIMESTAMP(FROM\_UNIXTIME(ts/1000, 'yyyy-MM-dd HH:mm:ss')) ," +

"WATERMARK FOR rowtime AS rowtime - INTERVAL '2' SECOND) " +

"WITH ("+ MyKafkaUtil.getKafkaDDL(pageViewSourceTopic,groupId)+")");

### 过滤数据

//TODO 4.过滤数据

Table fullwordView = tableEnv.sqlQuery("select page['item'] fullword ," +

"rowtime from page\_view " +

"where page['page\_id']='good\_list' " +

"and page['item'] IS NOT NULL ");

### 利用UDTF进行拆分

//TODO 5.利用udtf将数据拆分

Table keywordView = tableEnv.sqlQuery("select keyword,rowtime from " + fullwordView + " ," +

" LATERAL TABLE(ik\_analyze(fullword)) as T(keyword)");

### 聚合

//TODO 6.根据各个关键词出现次数进行ct

Table keywordStatsSearch = tableEnv.sqlQuery("select keyword,count(\*) ct, '"

+ GmallConstant.KEYWORD\_SEARCH + "' source ," +

"DATE\_FORMAT(TUMBLE\_START(rowtime, INTERVAL '10' SECOND),'yyyy-MM-dd HH:mm:ss') stt," +

"DATE\_FORMAT(TUMBLE\_END(rowtime, INTERVAL '10' SECOND),'yyyy-MM-dd HH:mm:ss') edt," +

"UNIX\_TIMESTAMP()\*1000 ts from "+keywordView

+ " GROUP BY TUMBLE(rowtime, INTERVAL '10' SECOND ),keyword");

### 转换为流并写入ClickHouse

#### 在ClickHouse中创建关键词统计表

create table keyword\_stats\_200821 (

stt DateTime,

edt DateTime,

keyword String ,

source String ,

ct UInt64 ,

ts UInt64

)engine =ReplacingMergeTree( ts)

partition by toYYYYMMDD(stt)

order by ( stt,edt,keyword,source );

#### 封装KeywordStats实体类

package com.atguigu.gmall.realtime.bean;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

/\*\*

\* Desc: 关键词统计实体类

\*/

@Data

@AllArgsConstructor

@NoArgsConstructor

public class KeywordStats {

private String keyword;

private Long ct;

private String source;

private String stt;

private String edt;

private Long ts;

}

#### 在主程序中转换流并写入ClickHouse

//TODO 7.转换为数据流

DataStream<KeywordStats> keywordStatsSearchDataStream =

tableEnv.<KeywordStats>toAppendStream(keywordStatsSearch, KeywordStats.class);

keywordStatsSearchDataStream.print();

//TODO 8.写入到ClickHouse

keywordStatsSearchDataStream.addSink(

ClickHouseUtil.<KeywordStats>getJdbcSink(

"insert into keyword\_stats(keyword,ct,source,stt,edt,ts) " +

" values(?,?,?,?,?,?)")

### 整体测试

* 启动ZK、Kafka、logger.sh、ClickHouse
* 运行BaseLogApp
* 运行KeywordStatsApp
* 运行rt\_applog目录下的jar包
* 查看控制台输出
* 查看ClickHouse中keyword\_stats\_2021表数据

# 总结

* DWS层主要是基于DWD和DWM层的数据进行轻度聚合统计
* 掌握利用union操作实现多流的合并
* 掌握窗口聚合操作
* 掌握对clickhouse数据库的写入操作
* 掌握用FlinkSQL实现业务
* 掌握分词器的使用
* 掌握在FlinkSQL中自定义函数的使用