实战 | Kafka + Flink + Redis 的电商大屏实时计算案

来源：LittleMagic

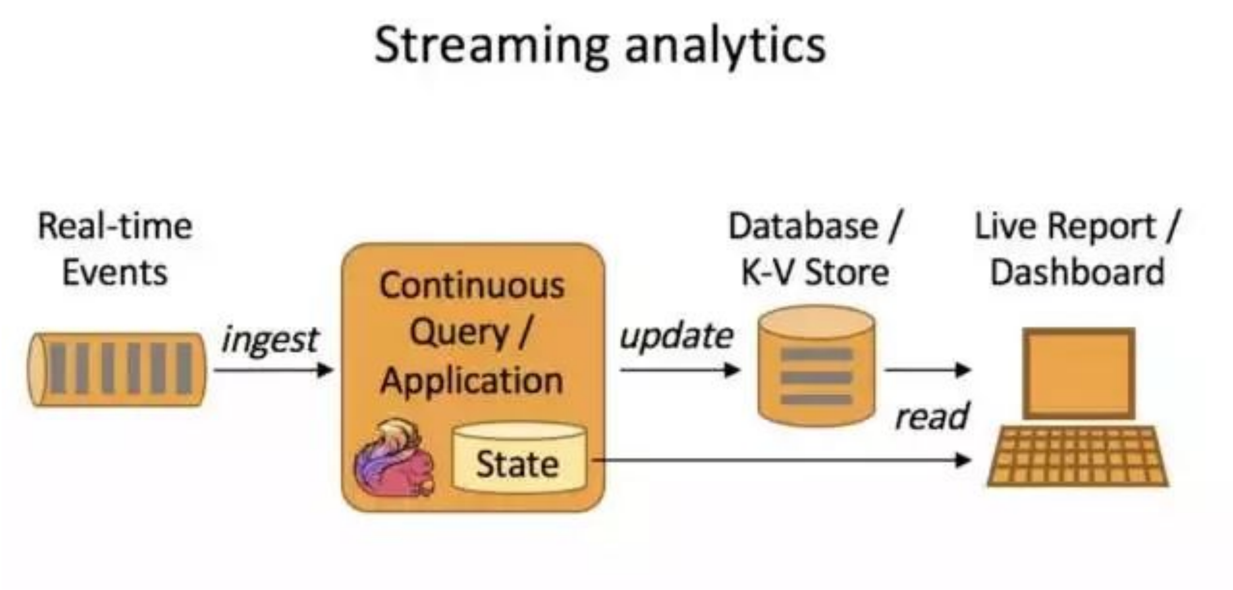
cloud.tencent.com/developer/article/1558372

本篇涉及到主要技术为Kafka + Flink + Redis，其中，Kafka相关的文章师长之前发过不少，对Kafka不太熟悉的可以先了解下：

目录

前言

阿里的双11销量大屏可以说是一道特殊的风景线。实时大屏（real-time dashboard）正在被越来越多的企业采用，用来及时呈现关键的数据指标。并且在实际操作中，肯定也不会仅仅计算一两个维度。由于Flink的“真·流式计算”这一特点，它比Spark Streaming要更适合大屏应用。本文从笔者的实际工作经验抽象出简单的模型，并简要叙述计算流程（当然大部分都是源码）。



数据格式与接入

简化的子订单消息体如下。

```
{
"userId": 234567,
"orderId": 2902306918400,
"subOrderId": 2902306918401,
"siteId": 10219,
"siteName": "site_blabla",
"cityId": 101,
"cityName": "北京市",
"warehouseId": 636,
"merchandiseId": 187699,
"price": 299,
"quantity": 2,
"orderStatus": 1,
"isNewOrder": 0,
"timestamp": 1572963672217
}
```

由于订单可能会包含多种商品，故会被拆分成子订单来表示，每条JSON消息表示一个子订单。现在要按照自然日来统计以下指标，并以1秒的刷新频率呈现在大屏上：

每个站点（站点ID即siteId）的总订单数、子订单数、销量与GMV；

当前销量排名前N的商品（商品ID即merchandiseId）与它们的销量。

由于大屏的最大诉求是实时性，等待迟到数据显然不太现实，因此我们采用处理时间作为时间特征，并以1分钟的频率做checkpointing。

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
env.enableCheckpointing(60 * 1000, CheckpointingMode.EXACTLY_ONCE);
env.getCheckpointConfig().setCheckpointTimeout(30 * 1000);
```

然后订阅Kafka的订单消息作为数据源。

```
Properties consumerProps =
ParameterUtil.getFromResourceFile("kafka.properties");
DataStream sourceStream = env
.addSource(new FlinkKafkaConsumer011<>(
ORDER_EXT_TOPIC_NAME, // topic
new SimpleStringSchema(), // deserializer
consumerProps // consumer properties
))
.setParallelism(PARTITION_COUNT)
.name("source_kafka_" + ORDER_EXT_TOPIC_NAME)
.uid("source_kafka_" + ORDER_EXT_TOPIC_NAME);
```

给带状态的算子设定算子ID（通过调用uid()方法）是个好习惯，能够保证Flink应用从保存点重启时能够正确恢复状态现场。为了尽量稳妥，Flink官方也建议为每个算子都显式地设定ID，参考：**https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/savepoints.html#should-i-assign-ids-to-all-operators-in-my-job**

插一下，有的说为啥不用别的，对比后选择可参考下面这篇：

接下来将JSON数据转化为POJO，JSON框架采用FastJSON。

```
DataStream orderStream = sourceStream
.map(message -> JSON.parseObject(message, SubOrderDetail.class))
.name("map_sub_order_detail").uid("map_sub_order_detail");
```

JSON已经是预先处理好的标准化格式，所以POJO类SubOrderDetail的写法可以通过Lombok极大地简化。如果JSON的字段有不规范的，那么就需要手写Getter和Setter，并用@JSONField注解来指明。

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
public class SubOrderDetail implements Serializable {
private static final long serialVersionUID = 1L;

private long userId;
private long orderId;
private long subOrderId;
private long siteId;
private String siteName;
private long cityId;
private String cityName;
private long warehouseId;
private long merchandiseId;
private long price;
private long quantity;
private int orderStatus;
private int isNewOrder;
```

```
private long timestamp;
}
```

统计站点指标

将子订单流按站点ID分组，开1天的滚动窗口，并同时设定ContinuousProcessingTimeTrigger触发器，以1秒周期触发计算。注意处理时间的时区问题，这是老生常谈了。

```
    WindowedStream siteDayWindowStream = orderStream
.keyBy("siteId")
.window(TumblingProcessingTimeWindows.of(Time.days(1), Time.hours(-8)))
.trigger(ContinuousProcessingTimeTrigger.of(Time.seconds(1)));
```

接下来写个聚合函数。

```
    DataStream siteAggStream = siteDayWindowStream
.aggregate(new OrderAndGmvAggregateFunc())
.name("aggregate_site_order_gmv").uid("aggregate_site_order_gmv");
public static final class OrderAndGmvAggregateFunc
implements AggregateFunction {
private static final long serialVersionUID = 1L;

    @Override
public OrderAccumulator createAccumulator() {
return new OrderAccumulator();
}

    @Override
public OrderAccumulator add(SubOrderDetail record, OrderAccumulator acc)
{
if (acc.getSiteId() == 0) {
acc.setSiteId(record.getSiteId());
acc.setSiteName(record.getSiteName());
```

```
}
acc.addOrderId(record.getOrderId());
acc.addSubOrderSum(1);
acc.addQuantitySum(record.getQuantity());
acc.addGmv(record.getPrice() * record.getQuantity());
return acc;
}

    @Override
public OrderAccumulator getResult(OrderAccumulator acc) {
return acc;
}

    @Override
public OrderAccumulator merge(OrderAccumulator acc1, OrderAccumulator
acc2) {
if (acc1.getSiteId() == 0) {
acc1.setSiteId(acc2.getSiteId());
acc1.setSiteName(acc2.getSiteName());
}
acc1.addOrderIds(acc2.getOrderIds());
acc1.addSubOrderSum(acc2.getSubOrderSum());
acc1.addQuantitySum(acc2.getQuantitySum());
acc1.addGmv(acc2.getGmv());
return acc1;
}
}
```

累加器类OrderAccumulator的实现很简单，看源码就大概知道它的结构了，因此不再多废话。唯一需要注意的是订单ID可能重复，所以需要用名为orderIds的HashSet来保存它。HashSet应付我们目前的数据规模还是没太大问题的，如果是海量数据，就考虑换用HyperLogLog吧。

接下来就该输出到Redis供呈现端查询了。这里有个问题：一秒内有数据变化的站点并不多，而ContinuousProcessingTimeTrigger每次触发都会输出窗口里全部的聚合数据，这样做了很多无用功，并且还会增大Redis的压力。所以，我们在聚合结果后再接一个ProcessFunction，代码如下。

```
    DataStream> siteResultStream = siteAggStream
.keyBy(0)
.process(new OutputOrderGmvProcessFunc(), TypeInformation.of(new
TypeHint>() {}))
.name("process_site_gmv_changed").uid("process_site_gmv_changed");
public static final class OutputOrderGmvProcessFunc
extends KeyedProcessFunction> {
private static final long serialVersionUID = 1L;

    private MapState state;

    @Override
public void open(Configuration parameters) throws Exception {
super.open(parameters);
state = this.getRuntimeContext().getMapState(new MapStateDescriptor<>(
"state_site_order_gmv",
Long.class,
OrderAccumulator.class)
);
}

    @Override
public void processElement(OrderAccumulator value, Context ctx,
Collector> out) throws Exception {
long key = value.getSiteId();
OrderAccumulator cachedValue = state.get(key);
```

```
    if (cachedValue == null || value.getSubOrderSum() !=
cachedValue.getSubOrderSum()) {
JSONObject result = new JSONObject();
result.put("site_id", value.getSiteId());
result.put("site_name", value.getSiteName());
result.put("quantity", value.getQuantitySum());
result.put("orderCount", value.getOrderIds().size());
result.put("subOrderCount", value.getSubOrderSum());
result.put("gmv", value.getGmv());
out.collect(new Tuple2<>(key, result.toJSONString());
state.put(key, value);
}
}

    @Override
public void close() throws Exception {
state.clear();
super.close();
}
}
```

说来也简单，就是用一个MapState状态缓存当前所有站点的聚合数据。由于数据源是以子订单为单位的，因此如果站点ID在MapState中没有缓存，或者缓存的子订单数与当前子订单数不一致，表示结果有更新，这样的数据才允许输出。

最后就可以安心地接上Redis Sink了，结果会被存进一个Hash结构里。

```
    // 看官请自己构造合适的FlinkJedisPoolConfig
FlinkJedisPoolConfig jedisPoolConfig =
ParameterUtil.getFlinkJedisPoolConfig(false, true);
siteResultStream
.addSink(new RedisSink<>(jedisPoolConfig, new GmvRedisMapper()))
.name("sink_redis_site_gmv").uid("sink_redis_site_gmv")
```

```
.setParallelism(1);
public static final class GmvRedisMapper implements RedisMapper> {
private static final long serialVersionUID = 1L;
private static final String HASH_NAME_PREFIX = "RT:DASHBOARD:GMV:";

    @Override
public RedisCommandDescription getCommandDescription() {
return new RedisCommandDescription(RedisCommand.HSET, HASH_NAME_PREFIX);
}

    @Override
public String getKeyFromData(Tuple2 data) {
return String.valueOf(data.f0);
}

    @Override
public String getValueFromData(Tuple2 data) {
return data.f1;
}

    @Override
public Optional getAdditionalKey(Tuple2 data) {
return Optional.of(
HASH_NAME_PREFIX +
new
LocalDateTime(System.currentTimeMillis()).toString(Consts.TIME_DAY_FORMA
T) +
"SITES"
);
}
}
```

商品Top N

我们可以直接复用前面产生的orderStream，玩法与上面的GMV统计大同小异。这里用1秒滚动窗口就可以了。

```
    WindowedStream merchandiseWindowStream = orderStream
.keyBy("merchandiseId")
.window(TumblingProcessingTimeWindows.of(Time.seconds(1)));

    DataStream> merchandiseRankStream = merchandiseWindowStream
.aggregate(new MerchandiseSalesAggregateFunc(), new
MerchandiseSalesWindowFunc())
.name("aggregate_merch_sales").uid("aggregate_merch_sales")
.returns(TypeInformation.of(new TypeHint>() { }));
```

聚合函数与窗口函数的实现更加简单了，最终返回的是商品ID与商品销量的二元组。

```
    public static final class MerchandiseSalesAggregateFunc
implements AggregateFunction {
private static final long serialVersionUID = 1L;

    @Override
public Long createAccumulator() {
return 0L;
}

    @Override
public Long add(SubOrderDetail value, Long acc) {
return acc + value.getQuantity();
}

    @Override
public Long getResult(Long acc) {
return acc;
}
```

```java
    @Override
public Long merge(Long acc1, Long acc2) {
return acc1 + acc2;
}
}


public static final class MerchandiseSalesWindowFunc
implements WindowFunction, Tuple, TimeWindow> {
private static final long serialVersionUID = 1L;

    @Override
public void apply(
Tuple key,
TimeWindow window,
Iterable accs,
Collector> out) throws Exception {
long merchId = ((Tuple1) key).f0;
long acc = accs.iterator().next();
out.collect(new Tuple2<>(merchId, acc));
}
}
```

既然数据最终都要落到Redis，那么我们完全没必要在Flink端做Top N的统计，直接利用Redis的有序集合（zset）就行了，商品ID作为field，销量作为分数值，简单方便。不过flink-redis-connector项目中默认没有提供ZINCRBY命令的实现（必须再吐槽一次），我们可以自己加，步骤参照之前写过的那篇加SETEX的命令的文章，不再赘述。RedisMapper的写法如下。

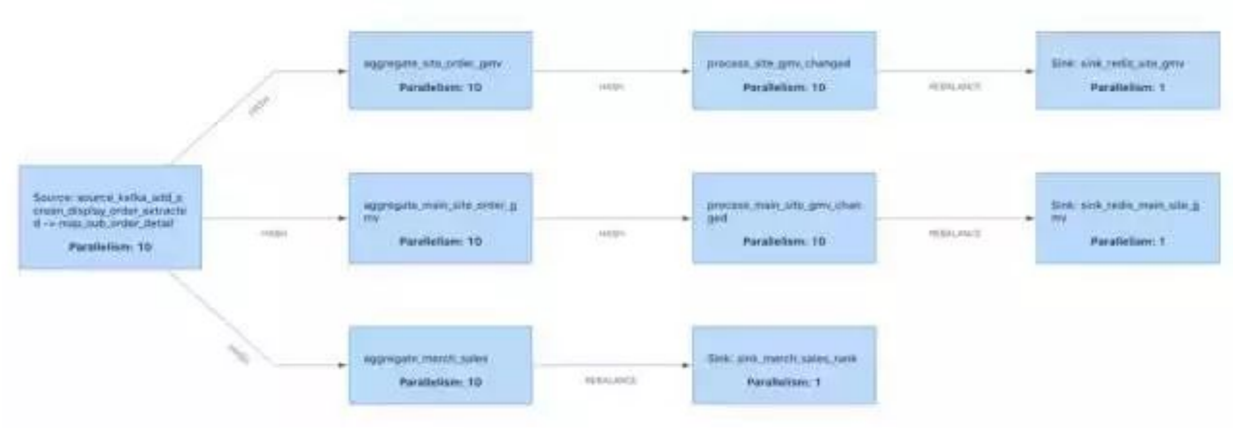
```java
    public static final class RankingRedisMapper implements RedisMapper>
{
private static final long serialVersionUID = 1L;
private static final String ZSET_NAME_PREFIX = "RT:DASHBOARD:RANKING:";
```

```
    @Override
public RedisCommandDescription getCommandDescription() {
return new RedisCommandDescription(RedisCommand.ZINCRBY,
ZSET_NAME_PREFIX);
}

    @Override
public String getKeyFromData(Tuple2 data) {
return String.valueOf(data.f0);
}

    @Override
public String getValueFromData(Tuple2 data) {
return String.valueOf(data.f1);
}

    @Override
public Optional getAdditionalKey(Tuple2 data) {
return Optional.of(
ZSET_NAME_PREFIX +
new
LocalDateTime(System.currentTimeMillis()).toString(Consts.TIME_DAY_FORMA
T) + ":" +
"MERCHANDISE"
);
}
}
```

后端取数时，用ZREVRANGE命令即可取出指定排名的数据了。只要数据规模不是大到难以接受，并且有现成的Redis，这个方案完全可以作为各类Top N需求的通用实现。

The End

大屏的实际呈现需要保密，截图自然是没有的。以下是提交执行时Flink Web UI给出的执行计划（实际有更多的统计任务，不止3个Sink）。通过复用源数据，可以在同一个Flink job内实现更多统计需求。



———— e n d ————