**Apache Beam构建流处理任务**

<https://blog.csdn.net/gzroy/article/details/130453895>

**以一个场景来展示如何用Beam来定义一个流处理的任务**：

假设我们要处理车辆上报的里程数据，车辆会不定期的上报数据，平台收到数据之后，会保存原始数据，同时会每分钟进行一次聚合处理，计算在这一分钟内车辆行驶的距离，然后保存在数据库之中。之后报表就可以根据用户的查询条件，检索在查询时间段内某辆车行驶的里程。

车辆上报的数据会通过平台的Kafka进行转发，数据处理模块会订阅相关的主题，获取数据并处理。车辆上报的数据是一个简单的Json格式，如下：

|  |
| --- |
| {  "telemetry": {  "odometer": {  "odometer": 1234,  }  },  "timestamp": 1682563540419,  "deviceId": "abc123",  } |

基于JAVA来编写一个Beam的流处理Pipeline。首先是用maven建立一个项目

|  |
| --- |
| mvn archetype:generate -DgroupId=com.example -DartifactId=analytics-pipeline -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false |

1. 生成Pipeline

新建一个名为MileageCalculate的类，在里面定义一个options接口，这个接口扩展了StreamingOptions，主要是提供了一些方法来获取和设置Pipeline要运行时的一些参数，例如获取Kafka topic等等

public interface Options extends StreamingOptions {

@Description("Apache Kafka topic to read from.")

@Validation.Required

String getInputTopic();

void setInputTopic(String value);

@Description("BigQuery table to write to, in the form "

+ "'project:dataset.table' or 'dataset.table'.")

@Default.String("beam\_samples.streaming\_beam\_sql")

String getOutputTable();

void setOutputTable(String value);

@Description("Apache Kafka bootstrap servers in the form 'hostname:port'.")

@Default.String("localhost:9092")

String getBootstrapServer();

void setBootstrapServer(String value);

@Description("Define max\_speed for distance abnormal.")

@Default.Integer(100)

Integer getMaxSpeed();

void setMaxSpeed(Integer value);

}

在类的main函数中，我们可以新建一个Pipeline，传入options

Options options = PipelineOptionsFactory.fromArgs(args).withValidation().as(Options.class);

options.setStreaming(true);

Pipeline pipeline = Pipeline.create(options);

2. 读取Kafka消息

建好Pipeline之后，我们第一步是从Kafka中获取数据作为输入。这里用到了Beam提供的KafkaIO类的方法来连接Kafka并订阅消息主题。因为消息的格式是JSON，我们需要新建一个类来映射JSON数据的格式，然后可以用Google的GSON来把消息转换为JAVA类。新建一个名为TelemetryMsg.java文件，内容如下：

import org.apache.beam.sdk.coders.DefaultCoder;

import org.apache.beam.sdk.extensions.avro.coders.AvroCoder;

public class TelemetryMsg {

@DefaultCoder(AvroCoder.class)

public static class UtilizationMsg {

public long timestamp;

public String deviceId;

public Telemetry telemetry;

}

public static class Odometer {

public int usageMode;

public float odometer;

}

public static class Telemetry {

public Odometer odometer;

}

}

因为Kafka消息内容里面已经带了一个时间戳信息，而KafkaIO默认是以收到消息的时间作为时间戳和Watermark的，我们需要设置一个自定义的Timepolicy，读取消息内容的时间戳，为此要新建一个名为CustomFieldTimePolicy的类，代码如下：

import com.examples.TelemetryMsg.UtilizationMsg;

import org.apache.beam.sdk.io.kafka.KafkaRecord;

import org.apache.beam.sdk.io.kafka.TimestampPolicy;

import org.apache.beam.sdk.transforms.windowing.BoundedWindow;

import org.joda.time.Instant;

import java.util.Optional;

import com.google.gson.Gson;

public class CustomFieldTimePolicy extends TimestampPolicy<String, String> {

private static final Gson GSON = new Gson();

protected Instant currentWatermark;

public CustomFieldTimePolicy(Optional<Instant> previousWatermark) {

currentWatermark = previousWatermark.orElse(BoundedWindow.TIMESTAMP\_MIN\_VALUE);

}

@Override

public Instant getTimestampForRecord(PartitionContext ctx, KafkaRecord<String, String> record) {

UtilizationMsg msg = GSON.fromJson(record.getKV().getValue(), UtilizationMsg.class);

currentWatermark = new Instant(msg.timestamp);

return currentWatermark;

}

@Override

public Instant getWatermark(PartitionContext ctx) {

return currentWatermark;

}

}

然后我们就可以为Pipeline添加一个读取Kafka消息的步骤了，代码如下：

PCollection<UtilizationMsg> input =

pipeline

.apply("Read messages from Kafka",

KafkaIO.<String, String>read()

.withBootstrapServers(options.getBootstrapServer())

.withTopic(options.getInputTopic())

.withKeyDeserializer(StringDeserializer.class)

.withValueDeserializer(StringDeserializer.class)

.withTimestampPolicyFactory((tp, previousWaterMark) -> new CustomFieldTimePolicy(previousWaterMark))

.withoutMetadata())

.apply("Get message contents", Values.<String>create())

.apply("Log messages", MapElements.into(TypeDescriptor.of(String.class))

.via(message -> {

LOG.info("Received: {}", message);

return message;

}))

.apply("Parse JSON", MapElements.into(TypeDescriptor.of(UtilizationMsg.class))

.via(message -> GSON.fromJson(message, UtilizationMsg.class)))

.apply("Append event time for PCollection records", WithTimestamps.of((UtilizationMsg msg) -> new Instant(msg.timestamp)));

3. 窗口处理

因为我们需要对一分钟内的数据进行聚合处理，所以需要把流数据按每一分钟划分为一个逻辑组，这里用到了fixedwindow来进行划分。Kafka的消息有可能出现晚到的情况，因此需要设置允许晚到1分钟，如果超出则丢弃。为了能即时处理数据，我们可以设置触发模式，例如从这个时间窗口收到第一条数据开始，经过多长时间我们就可以先进行数据的计算。最后我们还可以对时间窗口的数据进行去重操作。代码如下：

PCollection<UtilizationMsg> input\_window =

input

.apply("Fixed-size windows",

Window.<UtilizationMsg>into(FixedWindows.of(Duration.standardMinutes(1)))

.withAllowedLateness(Duration.standardMinutes(1))

.triggering(

Repeatedly.forever(

AfterWatermark

.pastEndOfWindow()

.withEarlyFirings(

AfterProcessingTime

.pastFirstElementInPane()

.plusDelayOf(Duration.standardMinutes(1)))))

.accumulatingFiredPanes())

.apply("Distinct",

Distinct.<UtilizationMsg>create());

4. 数据保存

每个时间窗口的数据我们可以保存在数据库之中。这里我用Postgres数据库，可以用beam的JdbcIO来进行数据库的连接。我在PG数据库之中新建了一个名为telematics的数据库，里面有一张telemetry\_data的数据表，代码如下：

input\_window

.apply(JdbcIO.<UtilizationMsg>write()

.withDataSourceConfiguration(JdbcIO.DataSourceConfiguration.create("org.postgresql.Driver", "jdbc:postgresql://127.0.0.1:5432/telematics")

.withUsername("postgres")

.withPassword("postgres"))

.withStatement("insert into regular\_data\_utilization Values (?, ?, ?, ?);")

.withPreparedStatementSetter(new JdbcIO.PreparedStatementSetter<UtilizationMsg>() {

public void setParameters(UtilizationMsg element, PreparedStatement query) throws SQLException {

query.setString(1, element.deviceId);

query.setString(2, Instant.ofEpochMilli(element.timestamp).toString());

query.setInt(3, element.telemetry.odometer.usageMode);

query.setFloat(4, element.telemetry.odometer.odometer);

}

}));

5. 数据分组

对于timewindow内的数据，我们可以按照deviceId来进行分组，分组后的数据是Key-Value的形式，Key是deviceId，Value是消息内容，这样可以方便后续对每一个deviceId来计算里程。

PCollection<KV<String, Iterable<UtilizationMsg>>> grouped\_records =

input\_window

.apply("Add DeviceID as Key", ParDo.of(new DoFn<UtilizationMsg, KV<String, UtilizationMsg>>() {

@ProcessElement

public void processElement(@Element UtilizationMsg element, OutputReceiver<KV<String, UtilizationMsg>> out) {

out.output(KV.of(element.deviceId, element));

}

}))

.apply(GroupByKey.<String, UtilizationMsg>create());

6. 计算里程

分组之后，我们就可以计算每个deviceId在每一分钟内行驶的里程了。考虑到里程表的数据可能有异常，我们可以采用multi output的方式来处理，即正常的数据处理后发送到一个PCollection，异常的数据发送到另一个PCollection。为此我们需要先定义两个TupleTag对象，分别对应正常和异常数据。

private static final TupleTag<DistanceObj> normalDistanceTag = new TupleTag<DistanceObj>(){};

private static final TupleTag<String> abnormalDistanceTag = new TupleTag<String>(){};

然后我们对数据进行里程计算，对于正常和异常的数据，会分别发送到不同的Tag的output。如以下代码：

PCollectionTuple distance = grouped\_records

.apply("Calculate distance", ParDo.of(new DoFn<KV<String, Iterable<UtilizationMsg>>, DistanceObj>() {

@ProcessElement

public void processElement(@Element KV<String, Iterable<UtilizationMsg>> element, IntervalWindow window, MultiOutputReceiver out) {

Iterator<UtilizationMsg> iterator = element.getValue().iterator();

List<UtilizationMsg> records = new ArrayList<UtilizationMsg>();

while(iterator.hasNext()) {

records.add(iterator.next());

}

Collections.sort(records, new UtilizationMsgCompare());

Iterator<UtilizationMsg> iter = records.iterator();

int total\_distance = 0;

float pre\_odometer = 0f;

long pre\_timestamp = 0L;

Boolean has\_abnormal\_data = false;

while(iter.hasNext()) {

UtilizationMsg record = (UtilizationMsg) iter.next();

float odometer = record.telemetry.odometer.odometer;

if (pre\_odometer==0) {

pre\_odometer = odometer;

pre\_timestamp = record.timestamp;

continue;

}

if (odometer >= pre\_odometer) {

int distance = (int) (record.telemetry.odometer.odometer - pre\_odometer);

int duration = (int) ((record.timestamp - pre\_timestamp)/1000); //seconds

if(distance <= duration \* max\_speed) {

total\_distance += distance;

} else {

has\_abnormal\_data = true;

}

} else {

has\_abnormal\_data = true;

}

pre\_odometer = odometer;

pre\_timestamp = record.timestamp;

}

DistanceObj d = new DistanceObj(element.getKey(), DateFormat.format(pre\_timestamp), total\_distance);

if (!has\_abnormal\_data) {

out.get(normalDistanceTag).output(d);

}

else {

Instant startWindow = window.start();

Instant endWindow = window.end();

String errorMsg = String.format(

"Abnormal distance found for device: %s, period: %s - %s",

element.getKey(), startWindow.toDateTime().toString(), endWindow.toDateTime().toString());

out.get(abnormalDistanceTag).output(errorMsg);

}

}

})

.withOutputTags(normalDistanceTag, TupleTagList.of(abnormalDistanceTag)));

7. 保存正常里程数据

对正常里程的数据，我们可以保存到数据库当中

PCollection<DistanceObj> normalDistance = distance.get(normalDistanceTag);

normalDistance

.apply(JdbcIO.<DistanceObj>write()

.withDataSourceConfiguration(JdbcIO.DataSourceConfiguration.create("org.postgresql.Driver", "jdbc:postgresql://127.0.0.1:5432/telematics")

.withUsername("postgres")

.withPassword("postgres"))

.withStatement("insert into distance Values (?, ?, ?, ?) ON CONFLICT (deviceId, hour) DO UPDATE SET (distance, process\_time) = (excluded.distance, excluded.process\_time);")

.withPreparedStatementSetter(new JdbcIO.PreparedStatementSetter<DistanceObj>() {

public void setParameters(DistanceObj element, PreparedStatement query) throws SQLException {

Timestamp process\_time = new Timestamp(new Date().getTime());

query.setString(1, element.getDeviceId());

query.setString(2, element.getHour());

query.setInt(3, element.getDistance());

query.setString(4, process\_time.toString());

}

}));

8. 输出异常里程数据到日志

对于异常的数据，这里只做一个简单的处理，输出到错误日志

PCollection<String> abnormalDistance = distance.get(abnormalDistanceTag);

abnormalDistance

.apply("Log abnormal distance", MapElements.into(TypeDescriptor.of(String.class))

.via(message -> {

LOG.error(message);

return message;

}));

9. 运行Pipeline

至此，整个Pipeline搭建完成，我们只要加多一行代码pipeline.run()即可执行。

最后，我们可以通过以下的maven命令来启动，这里是用的DirectRunner来运行，如果要用dataflow, spark或者Flink，需要在POM文件增加相应的依赖，然后在命令中指定Runner即可。

mvn compile exec:java -Dexec.mainClass=com.examples.MileaCalculate -Dexec.args="--inputTopic=TELEMATICS"

10. 测试Pipeline

这里简单的用一个Python脚本来模拟生成一些数据，检验Pipeline是否按设计运行。

from confluent\_kafka import Producer

import json

import time

import random

conf = {'bootstrap.servers': "127.0.0.1:9092",

'client.id': "test123"}

producer = Producer(conf)

topic = "TELEMATICS"

msg\_temp = {

"telemetry": {

"odometer": {

"odometer": 1234,

"usageMode": 0

}

},

"timestamp": 1682563540419,

"deviceId": "abc123"

}

start\_time = int(time.time()\*1000)

start\_odometer = 100

delta\_time = [10, 15, 40, 150, 50]

sleep\_time = [10, 5, 25, 110, 1]

delta\_odometer = [100, 80, 400, 1500, 500]

for i in range(len(delta\_time)):

time.sleep(sleep\_time[i])

timestamp = start\_time + delta\_time[i]\*1000

odometer = start\_odometer + delta\_odometer[i]

msg\_temp['telemetry']['odometer']['odometer'] = odometer

msg\_temp['timestamp'] = timestamp

producer.produce(topic, key="key", value=json.dumps(msg\_temp))

这里模拟了数据从开始时间经过10，15，40，150， 50秒发送。包括了数据异常以及数据晚到这两种异常场景。我们检验日志和数据库的数据，可以看到Pipeline按照设计正常运行。