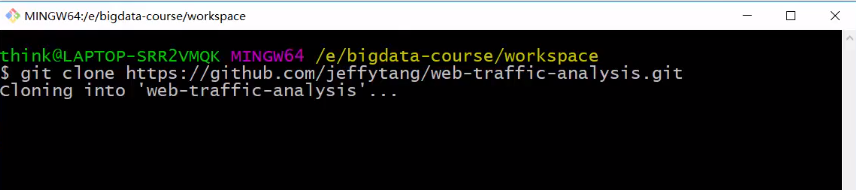
## 1.项目总体架构



## 2.在github上搭建项目结构目录

登录github，创建名为[web-traffic-analysis](https://github.com/Michealthree/web-traffic-analysis)的repository

将<https://github.com/Michealthree/web-traffic-analysis.git>拉倒本地



然后在项目目录下面执行 git bash shell

echo "# web-traffic-analysis" >> README.md

git add README.md

git commit -m "first commit"

git push -u origin master

## 3. 原始预解析逻辑的实现

写代码前先git创建一个分支 feature/preparselog

在backend创建weblog-preparser子项目

新建java实体类，和预解析类，导入junit4，编写单元测试类

## 4.原始与解析后入Hive库

start-dfs.sh

nohup hive --service metastore > ~/bigdata/apache-hive-2.3.6-bin/logs/metastore.log 2>&1 &

创建数据库

create database rawdata;

本地运行需要把core-site.xml hive-site.xml拷贝到resources目录

本地运行要把mysql驱动包导入

/\*\*

\*4.原始与解析后入Hive库

\*/

object PreparseETL {

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

val spark = SparkSession.builder()

.appName("PreparseETL")

.enableHiveSupport()

.getOrCreate()

val rawdataInputPath = spark.conf.get("spark.traffic.analysis.rawdata.input",

"hdfs://master:9999/user/hadoop-zsz/traffic-analysis/rawlog/20180615")

val numberPartitions = spark.conf.get("spark.traffic.analysis.rawdata.numberPartitions", "2").toInt

val preParsedLogRDD :RDD[PreParsedLog] = spark.sparkContext.textFile(rawdataInputPath).flatMap(line =>

Option(WebLogPreParser.parse(line)))

val preParsedLogDS = spark.createDataset(preParsedLogRDD)(Encoders.bean(classOf[PreParsedLog]))

preParsedLogDS.coalesce(numberPartitions)

.write

.mode(SaveMode.Append)

.partitionBy("year", "month", "day")

.saveAsTable("rawdata.web")

spark.stop()

}

## 5.github简单说明

每有一个新功能都要拉一个新的分支，开发测试完成后pull Request，然后可以提交review，没问题就可以Merge到master

## 6.URL的解析

/\*\*

\* 对一个已经编码的字符串进行解码

\* 有些字符串可能已经经过两次编码(比如url的参数等)，所以我们需要二次解码才能真正解码成功，例如：

\* https%3A%2F%2Fwww.underarmour.cn%2F%3Futm\_source%3Dbaidu%26utm\_term%3D%25E6%25A0%2587%25E9%25A2%2598%26utm\_medium%3DBrandZonePC%26utm\_channel%3DSEM

\* 第一次解码后为：

\* https://www.underarmour.cn/?utm\_source=baidu&utm\_term=%E6%A0%87%E9%A2%98&utm\_medium=BrandZonePC&utm\_channel=SEM

\* 第二次解码后为：

\* https://www.underarmour.cn/?utm\_source=baidu&utm\_term=标题&utm\_medium=BrandZonePC&utm\_channel=SEM

\* @param encodedStr 编码后的字符串

\* @return 完全解码后的字符串

\*/

详见代码

## 7.ColumnReader的实现

/\*\*

\* 按照一定的规则解析query\_string，将所有的kv值放到内存中

\* 便于根据key获取相对应的value

\*/

public class ColumnReader {

private Map<String, String> keyvalues = new HashMap<String, String>();

public ColumnReader(String line) {

String[] temps = line.split("&");

for (String kvStr : temps) {

String[] kv = kvStr.split("=");

if (kv.length == 2) {

keyvalues.put(kv[0], kv[1]);

}

}

}

public String getStringValue(String key) {

return UrlParseUtils.decode(keyvalues.getOrDefault(key, "-"));

}

}

## 8.日志解析结构设计

见代码dataobject包和objectbuilder包

包括了

### MouseClick日志的解析

### IPLocaltion parser

### Heartbeat和Event日志的解析

### pageViewde日志的解析

### StringMatcher的实现

1="google,(\\w+\\.)\*google\\.(\\w+|com\\.\\w+),q"

2="google,www\\.googleadservices\\.com/pagead/aclk.\*,null"

3="google,www\\.google\\.com.\*, null"

4="baidu,www.baidu\\.com/link.\*,wd"

通过上面正则来判断是否

### 搜索引擎和关键词的解析

### 来源类型和来源渠道的解析

### MongoDB配置目标页面

在mongoDB创建一个web-analysis数据库

再创建一个Collection名为TargetPage

Targetpage样例：

{

"Name": "test",

"IsDisabled": false,

"Description": "",

"Id": "57f8df278a4bf51e2d834be8",

"ProfileId": 524,

"MatchPattern": "/haier\_regUser3",

"MatchType": "CONTAINS",

"MatchWithoutQueryString": false,

"CreatedDate": "2016-10-02T22:12:46.674+08:00",

"CreatedBy": "Wd3Migrator",

"ModifiedDate": "2016-10-02T22:12:46.674+08:00",

"ModifiedBy": "Wd3Migrator"

}

插入mongo

### 新建metadata模块，用于读取MongoDB

### TargetPage的解析

### TargetPageDataObject的解析

## 9. LogParser对外提供的服务（串联以上的解析逻辑）

LogParser类和LogParserSetting类

## 10. WebLog-Parser需要计算的几个重要字段

Weblog-parser项目中的dataobject里面的类中增加

重要字段的计算方法，如时间解析转换等和pv相关的判断方法等

提前转换好，在ETL的时候就可以直接用了

## 11. WebLog-Parser总结

Weblog-perser的作用就是把preParserLog解析成各种所需的BaseDataObject

# ETL项目

Backend新建子项目spark-sessionization-etl

## 11.avro的Schema的定义

在项目中定义所需要的avro文件

例：

{

"namespace": "com.zsz.web",

"type": "record",

"name": "Heartbeat",

"fields":

[

{"name": "tracker\_version", "type": "string", "default": "-"},

{"name": "profile\_id", "type": "int", "default": 0},

{"name": "user\_id", "type": "string", "default": "-"},

{"name": "server\_session\_id", "type": "long", "default": 0},

{"name": "server\_time", "type": "string", "default": "-"},

{"name": "loading\_duration", "type": "int", "default": 0},

{"name": "page\_view\_id", "type": "string", "default": "-"}

]

}

利用avro maven插件compile以下就能自动生成对应实体类

## 12. Spark离线分析的流程

object WebETL {

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

val conf = new SparkConf

if (args.isEmpty) {

conf.setMaster("local")

}

conf.set("spark.serializer", classOf[KryoSerializer].getName)

conf.set("spark.kryo.registrator", classOf[WebRegistrator].getName)

//预处理输出的基本路径

val wdPreparsedLogBaseDir = conf.getOption("spark.web.etl.inputBaseDir")

.getOrElse("hdfs://master:9999/user/hive/warehouse/rawdata.db/web")

//主ETL输出路径

val outputBaseDir = conf.getOption("spark.web.etl.outputBaseDir")

.getOrElse("hdfs://master:9999/user/hadoop-zsz/traffic-analysis/web")

//日志的时间

val dateStr = conf.getOption("spark.web.etl.startDate").getOrElse("20180616")

//分区数

val numPartitions = conf.getInt("spark.web.etl.numberPartitions", 5)

conf.setAppName(s"WebETL-${dateStr}")

val spark = SparkSession.builder().config(conf).getOrCreate()

//预处理输出的具体路径

val preParsedLogPath =

s"$wdPreparsedLogBaseDir/year=${dateStr.substring(0, 4)}/month=${dateStr.substring(0, 6)}/day=${dateStr}"

/\*\*

\* transform后数据为：

\* PreParsedLog(profileId1, pv, queryString, serverTime....)

\* PreParsedLog(profileId1, mc, queryString, serverTime....)

\* PreParsedLog(profileId2, pv, queryString, serverTime....)

\* ..........

\* PreParsedLog(profileId3, ev, queryString, serverTime....)

\* PreParsedLog(profileIdn, hb, queryString, serverTime....)

\*/

//读取预处理后的日志

val parsedLogRDD = spark.read.parquet(preParsedLogPath)

//将DataFrame转成Dataset[PreParsedLog]再转成RDD[PreparsedLog]

.map(transform(\_))(Encoders.bean(classOf[PreParsedLog])).rdd

//将RDD[PreParsedLog]转成RDD[(CombinedId, BaseDataObject)]

.flatMap(p => WebLogParser.parse(p))

/\*\*

\* WebLogParser.parse转换后数据为：

\* (CombinedId(profileId1,user1), BaseDataObject(profileId1,user1,pv,client\_ip.....))

\* (CombinedId(profileId1,user1), BaseDataObject(profileId1,user1,mc,client\_ip.....))

\* (CombinedId(profileId2,user2), BaseDataObject(profileId2,user2,pv,client\_ip.....))

\* ............

\* (CombinedId(profileId3,user3), BaseDataObject(profileId3,user3,ev,client\_ip.....))

\* (CombinedId(profileIdn,usern), BaseDataObject(profileIdn,usern,pv,client\_ip.....))

\*/

//将parsedLogRDD按照key进行分组

parsedLogRDD.groupByKey(new HashPartitioner(numPartitions)).mapPartitionsWithIndex((index, iterator) => {

//groupByKey后的每一个分区的数据为：

/\*\*

\* 转换后数据为：

\* (CombinedId(profileId1,user1), List(BaseDataObject(profileId1,user1,pv,client\_ip.....),

\* BaseDataObject(profileId1,user1,mc,client\_ip.....)))

\* (CombinedId(profileId2,user2), List(BaseDataObject(profileId2,user2,pv,client\_ip.....),

\* BaseDataObject(profileId2,user2,mc,client\_ip.....),

\* BaseDataObject(profileId2,user2,ev,client\_ip.....)

\* BaseDataObject(profileId2,user2,hb,client\_ip.....)))

\* ............

\* (CombinedId(profileId3,user3), List(BaseDataObject(profileId3,user3,ev,client\_ip.....)))

\* (CombinedId(profileIdn,usern), List(BaseDataObject(profileIdn,usern,pv,client\_ip.....),

\* BaseDataObject(profileIdn,usern,mc,client\_ip.....),

\* BaseDataObject(profileIdn,usern,pv,client\_ip.....)))

\*/

//处理每一个分区的数据

val partitionProcessor = new PartitionProcessor(index, iterator, outputBaseDir, dateStr)

partitionProcessor.run()

//不需要返回，所以返回空

Iterator[Unit]()

//触发action操作

}).foreach( (\_: Unit) => {})

spark.stop()

//给表web-user创建snapshot，以便于数据的重跑

val snapshotAdmin = new HBaseSnapshotAdmin(HbaseConnectionFactory.getHbaseConn)

val targetUserTable = System.getProperty("web.etl.hbase.UserTableName", "web-user")

snapshotAdmin.takeSnapshot(s"${targetUserTable}\_snapshot-${dateStr}", targetUserTable)

}

private def transform(row: Row): PreParsedLog = {

val p = new PreParsedLog

p.setClientIp(row.getAs[String]("clientIp"))

p.setCommand(row.getAs[String]("command").toString)

p.setMethod(row.getAs[String]("method"))

p.setProfileId(row.getAs[Int]("profileId"))

p.setQueryString(row.getAs[String]("queryString"))

p.setServerIp(row.getAs[String]("serverIp"))

p.setServerPort(row.getAs[Int]("serverPort"))

p.setServerTime(row.getAs[String]("serverTime"))

p.setUriStem(row.getAs[String]("uriStem"))

p.setUserAgent(row.getAs[String]("userAgent"))

p

}

}

### Web-ETL数据流

1、从HDFS中读取PreparedLog的parquet数据，数据结构如下：

Row(profileId1, pv, queryString, serverTime....)

Row(profileId1, mc, queryString, serverTime....)

Row(profileId2, pv, queryString, serverTime....)

..........

Row(profileId3, ev, queryString, serverTime....)

Row(profileIdn, hb, queryString, serverTime....)

2、将DataFrame转成RDD[PreParsedLog]

PreParsedLog(profileId1, pv, queryString, serverTime....)

PreParsedLog(profileId1, mc, queryString, serverTime....)

PreParsedLog(profileId2, pv, queryString, serverTime....)

..........

PreParsedLog(profileId3, ev, queryString, serverTime....)

PreParsedLog(profileIdn, hb, queryString, serverTime....)

3、将RDD[PreParsedLog]转成RDD[(CombinedId, BaseDataObject)]

(CombinedId(profileId1,user1), BaseDataObject(profileId1,user1,pv,client\_ip.....))

(CombinedId(profileId1,user1), BaseDataObject(profileId1,user1,mc,client\_ip.....))

(CombinedId(profileId2,user2), BaseDataObject(profileId2,user2,pv,client\_ip.....))

............

(CombinedId(profileId3,user3), BaseDataObject(profileId3,user3,ev,client\_ip.....))

(CombinedId(profileIdn,usern), BaseDataObject(profileIdn,usern,pv,client\_ip.....))

4、将parsedLogRDD按照key进行分组(相同的profileId+userId产生的日志DataObjects在同一个分区中)，

且重新分区，对于每一个分区的数据格式如下：

(CombinedId(profileId1,user1), List(BaseDataObject(profileId1,user1,pv,client\_ip.....),

BaseDataObject(profileId1,user1,mc,client\_ip.....)))

(CombinedId(profileId2,user2), List(BaseDataObject(profileId2,user2,pv,client\_ip.....),

BaseDataObject(profileId2,user2,mc,client\_ip.....),

BaseDataObject(profileId2,user2,ev,client\_ip.....)

BaseDataObject(profileId2,user2,hb,client\_ip.....)))

............

(CombinedId(profileId3,user3), List(BaseDataObject(profileId3,user3,ev,client\_ip.....)))

(CombinedId(profileIdn,usern), List(BaseDataObject(profileIdn,usern,pv,client\_ip.....),

BaseDataObject(profileIdn,usern,mc,client\_ip.....),

BaseDataObject(profileIdn,usern,pv,client\_ip.....)))

## 13. PartitionProcessor的编写

*每一个分区数据的处理者(分区级别)*

### PartitionProcessor数据流

1、PartitionProcessor的输入数据格式：

(CombinedId(profileId1,user1), List(BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....)))

(CombinedId(profileId2,user2), List(BaseDataObject(profileId2,user2,2018-06-15 12:20:21,pv,client\_ip.....),

BaseDataObject(profileId2,user2,2018-06-15 12:20:26,mc,client\_ip.....),

BaseDataObject(profileId2,user2,2018-06-15 12:20:27,ev,client\_ip.....)

BaseDataObject(profileId2,user2,2018-06-15 12:20:34,hb,client\_ip.....)))

............

(CombinedId(profileId3,user3), List(BaseDataObject(profileId3,user3,2018-06-15 17:20:21,ev,client\_ip.....)))

(CombinedId(profileIdn,usern), List(BaseDataObject(profileIdn,usern,2018-06-15 10:10:21,pv,client\_ip.....),

BaseDataObject(profileIdn,usern,2018-06-15 10:10:27,mc,client\_ip.....),

BaseDataObject(profileIdn,usern,2018-06-15 10:10:38,pv,client\_ip.....)))

2、循环处理每一个user的DataObjects(以CombinedId(profileId1,user1)为例)

2.1、对一个user中的所有的DataObject按照时间进行升序排序

(CombinedId(profileId1,user1), List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....)))

2.2、会话的切割，按照30分钟最基本的规则，切成了3个会话

SessionData1 -> (List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....))

SessionData2 -> (List(BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....)))

SessionData3 -> (List(BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....)))

2.3、对当前user产生的会话进行聚合计算，生成最终的实体

DataRecords1 -> (Session(profileId1,user1,2018-06-15 13:20:30,serverSessionId1,...Session相关字段...),

List(PageView(profileId1,user1,2018-06-15 13:20:30,...PageView相关字段...)),

List(MouseClick(profileId1,user1,2018-06-15 13:21:30,...MouseClick相关字段...),

MouseClick(profileId1,user1,2018-06-15 13:21:45,...MouseClick相关字段...)),

List(Heartbeat(profileId1,user1,2018-06-15 13:22:30,...Heartbeat相关字段...)))

DataRecords2 -> (Session(profileId1,user1,2018-06-15 19:22:31,serverSessionId2,...Session相关字段...),

List(PageView(profileId1,user1,2018-06-15 19:22:31,...PageView相关字段...)),

List(MouseClick(profileId1,user1,2018-06-15 19:22:33,...MouseClick相关字段...)),

List(Heartbeat(profileId1,user1,2018-06-15 19:22:39,...Heartbeat相关字段...)))

DataRecords3 -> (Session(profileId1,user1,2018-06-15 22:20:21,serverSessionId3,...Session相关字段...),

List(PageView(profileId1,user1,2018-06-15 22:20:21,...PageView相关字段...),

PageView(profileId1,user1,2018-06-15 22:20:27,...PageView相关字段...)),

List(MouseClick(profileId1,user1,2018-06-15 22:20:22,...MouseClick相关字段...),

MouseClick(profileId1,user1,2018-06-15 22:20:23,...MouseClick相关字段...)),

List(Heartbeat(profileId1,user1,2018-06-15 22:20:25,...Heartbeat相关字段...),

Heartbeat(profileId1,user1,2018-06-15 22:20:30,...Heartbeat相关字段...)),

List(Conversion(profileId1,user1,2018-06-15 22:20:28,...Conversion相关字段...)))

2.4、将产生的记录写到HDFS中

## 14. 会话切割逻辑实现

具体逻辑见代码

使用了foldLeft

### SessionGenerator数据流

会话的切割是user级别的，针对每一个user所产生的所有的dataObjects

1、会话切割的输入: 排序好的user1在网站profileId1上产生的dataObjects

(CombinedId(profileId1,user1), List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:29:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:30:10,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:31:12,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:45:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:46:30,hb,client\_ip.....)))

2、按照最基本的规则，两个相邻dataObjects的时间超过30分钟则切割会话

第一个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:29:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:30:10,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:31:12,hb,client\_ip.....))

第二个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....))

第三个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:45:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:46:30,hb,client\_ip.....))

3、按照重要入口pv再次切割会话：

第一个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....))

第二个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 13:29:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:30:10,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:31:12,hb,client\_ip.....))

第三个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....))

第四个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....))

第五个会话：

List(BaseDataObject(profileId1,user1,2018-06-15 22:45:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:46:30,hb,client\_ip.....))

4、转换成SessionData

第一个会话：

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....)))

第二个会话：

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 13:29:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:30:10,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:31:12,hb,client\_ip.....)))

第三个会话：

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....)))

第四个会话：

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....)))

第五个会话：

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 22:45:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:46:30,hb,client\_ip.....)))

## 15.会话聚合计算逻辑

将用户产生的会话数据聚合计算转换成 相应实体类

### UserSessionDataAggregator数据流

对当每一个user产生的会话进行聚合计算，生成最终的实体，以user1在网站profileId1上产生的所有的SessionData为例

1、聚合计算的输入数据：

List(SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 13:20:30,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:30,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:21:45,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:22:30,hb,client\_ip.....))),

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 13:29:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:30:10,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 13:31:12,hb,client\_ip.....))),

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 19:22:31,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:33,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 19:22:39,hb,client\_ip.....))),

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....))),

SessionData(List(BaseDataObject(profileId1,user1,2018-06-15 22:45:31,重要pv,client\_ip.....),

BaseDataObject(profileId1,user1,2018-06-15 22:46:30,hb,client\_ip.....))))

2、给一个会话中的所以dataObject进行归类(会话级别), ClassifiedSessionData中的数据格式为，以第四个会话为例

每一个会话对应一个ClassifiedSessionData

pvData = List(PvDataObject(profileId1,user1,2018-06-15 22:20:21,pv,client\_ip.....),

PvDataObject(profileId1,user1,2018-06-15 22:20:27,pv,client\_ip.....))

hbDataMap = List(HeartbeatDataObject(profileId1,user1,2018-06-15 22:20:25,hb,client\_ip.....),

HeartbeatDataObject(profileId1,user1,2018-06-15 22:20:30,hb,client\_ip.....))

targetPageData = List()

mcData = List(McDataObject(profileId1,user1,2018-06-15 22:20:22,mc,client\_ip.....),

McDataObject(profileId1,user1,2018-06-15 22:20:23,mc,client\_ip.....))

eventData = List(EventDataObject(profileId1,user1,2018-06-15 22:20:28,ev,client\_ip.....))

3、分别对所有的ClassifiedSessionData进行遍历，聚合计算出最终需要的实体，实体的数据结构为，还是以第四个会话为例：

DataRecords： (Session(profileId1,user1,2018-06-15 22:20:21,serverSessionId3,...Session相关字段...),

List(PageView(profileId1,user1,2018-06-15 22:20:21,...PageView相关字段...),

PageView(profileId1,user1,2018-06-15 22:20:27,...PageView相关字段...)),

List(MouseClick(profileId1,user1,2018-06-15 22:20:22,...MouseClick相关字段...),

MouseClick(profileId1,user1,2018-06-15 22:20:23,...MouseClick相关字段...)),

List(Heartbeat(profileId1,user1,2018-06-15 22:20:25,...Heartbeat相关字段...),

Heartbeat(profileId1,user1,2018-06-15 22:20:30,...Heartbeat相关字段...)),

List(Conversion(profileId1,user1,2018-06-15 22:20:28,...Conversion相关字段...)))

4、每一个实体中的每一个字段的详细计算

### 具体代码

/\*\*

\* 每一个user产生的所有的会话的聚合计算(user级别)

\*

\* @param id user唯一标识

\* @param serverSessionIdStart 起始会话id

\*/

class UserSessionDataAggregator(

id: CombinedId,

serverSessionIdStart: Long,

lastPersistedUserVisitInfo: Option[UserVisitInfo])

extends ConversionBuilder

with AvroRecordBuilder {

import com.zsz.spark.web.RichCollection.\_

/\*\*

\* 会话聚合计算

\* 1、生成最终的Session对象

\* 2、生产最终的Conversion对象

\* 3、生成最终的PageView对象

\* 4、生成最终的Heartbeat对象

\* 5、生成最终的MouseClick对象

\*

\* @param sessions 一个user产生的所有的会话

\* @return 这个user产生的所有的聚合后DataRecords

\*/

def aggregate(sessions: Seq[SessionData]): (UserVisitInfo, Seq[DataRecords]) = {

val classifiedSessionData = sessions.zipWithIndex map { case (sessionData, index) =>

//给每一个会话中的所以dataObject进行归类

new ClassifiedSessionData(index, sessionData)

}

val initUserVisitInfo = lastPersistedUserVisitInfo getOrElse UserVisitInfo(id, UserVisitInfo.INIT\_LAST\_VISIT\_TIME, 0)

//对每一个会话进行聚合计算，将最终的实体计算出来

val (userVisitInfo, finalRecordsBuilder) =

classifiedSessionData.foldLeft(initUserVisitInfo, Vector.newBuilder[DataRecords]) {

case ((currentUserVisitInfo, recordsBuilder), data) => {

//以下所有都是一个会话级别的计算

//计算会话id

val sessionId = serverSessionIdStart + data.sessionIndex

//计算这个会话最终的会话实体

val session = produceSession(sessionId, data, currentUserVisitInfo)

//计算这个会话中的所有的PageView实体

val pageViews = producePageViews(data, session)

//计算这个会话中的所有的Heartbeat实体

val heartbeats = produceHeartBeats(data.hbDataMap, session)

//计算这个会话中的所有的MouseClick实体

val mouseClicks = produceMouseClicks(data.mcData, session, data.pvData)

//计算这个会话中的转化

val conversions = produceConversions(sessionId, data.allActiveTargetInfo, data.eventData)

//将会话的信息派生到转化实体中

val conversionsWithSession = conversions.map(buildConversion(\_, session))

//更新用户访问信息

val updatedVisitInfo = currentUserVisitInfo.copy(

lastVisitIndex = currentUserVisitInfo.lastVisitIndex + 1,

lastVisitTime = data.sessionStartTime)

//这个会话中的所有的实体对象

recordsBuilder += DataRecords(session, pageViews, mouseClicks, conversionsWithSession, heartbeats)

(updatedVisitInfo, recordsBuilder)

}

}

(userVisitInfo, finalRecordsBuilder.result())

}

private def produceConversions(sessionId: Long,

targetInfoData: Seq[(TargetPageDataObject, TargetPageInfo)],

eventData: Seq[EventDataObject]): Seq[Conversion] = {

buildConversions(sessionId, targetInfoData, eventData)

}

/\*\*

\* 计算Session

\* @param sessionId 会话id

\* @param data 会话中所有的DataObject

\* @return Session

\*/

private def produceSession(sessionId: Long,

data: ClassifiedSessionData, userVisitInfo: UserVisitInfo): Session = {

val sessionBuilder = Session.newBuilder()

sessionBuilder.setServerSessionId(sessionId)

//计算会话停留时长

sessionBuilder.setSessionDuration(data.fetchSessionDuration)

//计算是否是新的访客

val isNewVisitor = if (userVisitInfo.lastVisitTime == UserVisitInfo.INIT\_LAST\_VISIT\_TIME) true else false

//计算这个访客自从上次访问到这次访问中间隔了多少天

val daysSinceLastVisit =

if (isNewVisitor) -1

else {

import com.github.nscala\_time.time.Imports.\_

val lastDateMidNight = new DateTime(userVisitInfo.lastVisitTime).toDateMidnight

val currentDateMidNight = new DateTime(data.sessionStartTime).toDateMidnight

(lastDateMidNight to currentDateMidNight).toDuration.days.toInt

}

sessionBuilder.setIsNewVisitor(isNewVisitor)

sessionBuilder.setDaysSinceLastVisit(daysSinceLastVisit)

sessionBuilder.setUserVisitNumber(userVisitInfo.lastVisitIndex + 1) //访客访问的次数

//会话特定的页面维度

val (landingPageViewInfo, secondPageViewInfo, exitPagePageViewInfo) =

getPageViewInfos(data.pvData, data.selectedPVOpt)

sessionBuilder.setLandingPageUrl(landingPageViewInfo.url)

sessionBuilder.setLandingPageOriginalUrl(landingPageViewInfo.originalUrl)

sessionBuilder.setLandingPageHostname(landingPageViewInfo.hostName)

sessionBuilder.setLandingPageTitle(landingPageViewInfo.title)

sessionBuilder.setSecondPageUrl(secondPageViewInfo.url)

sessionBuilder.setSecondPageOriginalUrl(secondPageViewInfo.originalUrl)

sessionBuilder.setSecondPageHostname(secondPageViewInfo.hostName)

sessionBuilder.setSecondPageTitle(secondPageViewInfo.title)

sessionBuilder.setExitPageUrl(exitPagePageViewInfo.url)

sessionBuilder.setExitPageOriginalUrl(exitPagePageViewInfo.originalUrl)

sessionBuilder.setExitPageHostname(exitPagePageViewInfo.hostName)

sessionBuilder.setExitPageTitle(exitPagePageViewInfo.title)

//会话实体统计维度

sessionBuilder.setPvCount(data.pvData.length)

sessionBuilder.setPvDistinctCount(data.pvData.distinctBy(\_.getSiteResourceInfo.getUrl).length)

sessionBuilder.setIsBounced(data.pvData.size == 1)

sessionBuilder.setMouseClickCount(data.mcData.length)

sessionBuilder.setTargetCount(data.allActiveTargetInfo.length)

sessionBuilder.setEventCount(data.eventData.length)

sessionBuilder.setConversionCount(data.eventData.length + data.allActiveTargetInfo.length)

sessionBuilder.setTargetDistinctCount(data.allActiveTargetInfo.distinctBy { case (\_, info) => info.getKey }.length)

sessionBuilder.setEventDistinctCount(data.eventData.distinctBy(e => (e.getEventCategory, e.getEventLabel, e.getEventAction)).length)

buildSession(data.pvData, data.selectedPVOpt, data.selectedFirstDataObject, sessionBuilder)

}

/\*\*

\* 计算PageView的字段

\* @param data 当前会话中的所有的dataObject

\* @param session 当前会话

\* @return PageView

\*/

private def producePageViews(data: ClassifiedSessionData, session: Session): Seq[PageView] = {

val pvData = data.pvData

val (\_, pageViewsBuilder, \_) = pvData.zipWithIndex.foldLeft(1, new VectorBuilder[PageView], None: Option[(PvDataObject, Int)]) {

//pageViewDepth表示页面访问深度，pageViewBuilder表示计算后的PageView的列表

//previous表示前一个pv及其在所有pv中的index

//currentPv表示当前pv， currentIndex当前pv在所有pv中的index

case ((pageViewDepth, pageViewBuilder, previous), (currentPv, currentIndex)) =>

val recordBuilder = PageView.newBuilder()

//获取当前pv对应的hb

val currentPvHbOpt = data.hbDataMap.get(currentPv.getPvId)

//计算页面加载时长

val loading = currentPvHbOpt match {

case Some(hb) => hb.getLoadingDuration

case None => 0

}

recordBuilder.setLoadingDuration(loading)

recordBuilder.setAccessOrder(currentIndex + 1) //页面的访问顺序

recordBuilder.setPageDuration(currentPv.getDuration) //页面的停留时长，已经在计算会话停留时长的时候计算过了

recordBuilder.setIsExitPage(currentIndex >= pvData.length - 1) //判断是否是退出页

//计算页面访问深度

val nextDepth = previous map { case (previousPv, \_) =>

if (currentPv.getSiteResourceInfo.getUrl.equals(previousPv.getSiteResourceInfo.getUrl)) {

recordBuilder.setIsRefresh(true)

pageViewDepth

} else {

pageViewDepth + 1

}

} getOrElse (pageViewDepth)

recordBuilder.setPageViewDepth(nextDepth)

//计算PageView的其他的字段

val filledPageView = buildPageView(currentPv, recordBuilder, session)

pageViewBuilder += filledPageView

(nextDepth, pageViewBuilder, Some(currentPv, currentIndex))

}

pageViewsBuilder.result()

}

private def produceHeartBeats(hbDataMap: Map[String, HeartbeatDataObject], session: Session): Seq[Heartbeat] = {

hbDataMap.values.map(buildHeartbeat(\_, session))(scala.collection.breakOut)

}

private def produceMouseClicks(mcData: Seq[McDataObject],

session: Session, pvData: Seq[PvDataObject]): Seq[MouseClick] = {

mcData.map(buildMouseClick(\_, session))(scala.collection.breakOut)

}

private def getPageViewInfos(pvArray: Seq[PvDataObject], mandatoryPvOpt: Option[PvDataObject]

): (PageViewInfo, PageViewInfo, PageViewInfo) = pvArray match {

case Seq(onlyOnePage) => //如果只有一个pv的话，则这个pv是着陆页也是退出页

val sitResourceInfo = onlyOnePage.getSiteResourceInfo

val pageInfo = PageViewInfo(sitResourceInfo.getUrl, sitResourceInfo.getOriginalUrl,

sitResourceInfo.getDomain, sitResourceInfo.getPageTitle)

(pageInfo, PageViewInfo.Default, pageInfo)

case Seq(firstPv, secondPv, \_\*) => { //含有2个pv或者以上的情况

//着陆页pv先取是重要入口的pv，如果没有重要入口pv的话就取首个pv

val firstPvSiteResourceInfo = mandatoryPvOpt.getOrElse(firstPv).getSiteResourceInfo

val secondPvSiteResourceInfo = if (mandatoryPvOpt.nonEmpty && firstPv.isDifferentFrom(mandatoryPvOpt.orNull)) firstPv.getSiteResourceInfo else secondPv.getSiteResourceInfo

val lastPv = pvArray.last

val lastPvSiteResourceInfo = if (lastPv.isDifferentFrom(mandatoryPvOpt.orNull)) lastPv.getSiteResourceInfo else pvArray(pvArray.length - 2).getSiteResourceInfo

(PageViewInfo(firstPvSiteResourceInfo.getUrl, firstPvSiteResourceInfo.getOriginalUrl, firstPvSiteResourceInfo.getDomain, firstPvSiteResourceInfo.getPageTitle),

PageViewInfo(secondPvSiteResourceInfo.getUrl, secondPvSiteResourceInfo.getOriginalUrl, secondPvSiteResourceInfo.getDomain, secondPvSiteResourceInfo.getPageTitle),

PageViewInfo(lastPvSiteResourceInfo.getUrl, lastPvSiteResourceInfo.getOriginalUrl, lastPvSiteResourceInfo.getDomain, lastPvSiteResourceInfo.getPageTitle))

}

case \_ => (PageViewInfo.Default, PageViewInfo.Default, PageViewInfo.Default)

}

}

//会话特定的页面维度

private case class PageViewInfo(url: String, originalUrl: String, hostName: String, title: String)

private object PageViewInfo {

val Default = PageViewInfo("-", "-", "-", "-")

}

## 16.Session字段的解析

见代码

UserSessionDataAggregator中的produceSession方法

## 17. PageView字段的解析计算

UserSessionDataAggregator中的producePageViews方法

## 18.其他实体字段的解析计算

//计算这个会话中的所有的Heartbeat实体

val heartbeats = produceHeartBeats(data.hbDataMap, session)

//计算这个会话中的所有的MouseClick实体

val mouseClicks = produceMouseClicks(data.mcData, session, data.pvData)

//计算这个会话中的转化

val conversions = produceConversions(sessionId, data.allActiveTargetInfo, data.eventData)

## 19.将实体以avro的格式写到HDFS中

/\*\*

\* ETL输出组件(以Avro的文件格式输出到HDFS)

\*/

trait AvroOutputComponen

## 20.scala的单元测试和集成测试

<dependency>

<groupId>org.scalatest</groupId>

<artifactId>scalatest\_2.11</artifactId>

<version>3.0.5</version>

<scope>test</scope>

</dependency>

集成测试直接在本地运行local模式的spark应用

然后查看HDFS下是否生成结果文件

验证结果等应用跑完，执行etl\_temp.sql，然后在hive中统计数据

## 21．数据导入到最终的模型数据库中

流程：跑完的数据在HDFS，然后建立hive临时表指定对应路径，最后使用脚本将临时表（avro）的数据导入实体表（parquet）中

最终的数据是要以parquet格式分区存储

执行etl\_init.sql

创建对应数据库和 实体表

执行web\_etl\_daily.sh ， ./web\_etl\_daily.sh 20180615

需要启动yarn

将spark-session-etl输出的数据按天导入到HIVE

## 22.user访问信息维度计算

需要将访客信息存储到数据库中，数据量较大且要支持随机读写，选择Hbase来存储

相关代码在com.zsz.spark.web.external包中

HbaseConnectionFactory获取Hbase连接的工厂类

建表

create 'web-user', 'f', {SPLITS => ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], COMPRESSION => 'SNAPPY'}

//测试环境切分两个就好

create 'web-user', 'f', {SPLITS => ['0', '1']}

UserSessionDataAggregator中也有相关业务逻辑

代码重新运行，启动hbase，提前建好hbase表，行运web-ETL

运行正常会在HDFS生成数据文件，运行脚本./web\_etl\_daily.sh 20180615将ETL好的数据导入Hive

## 23.数据重跑逻辑

假设要跑20180616这天的数据

### 正常逻辑

#### 预处理

1.准备原始日志 weblog-20180616.txt put 到HDFS中

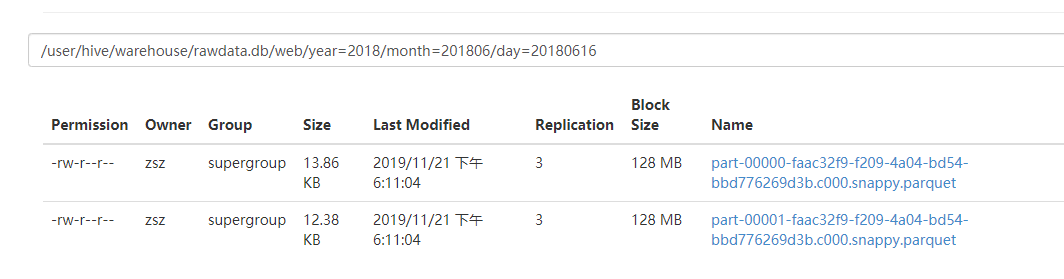
hadoop fs -mkdir hdfs://master:9999/user/hadoop-zsz/traffic-analysis/rawlog/20180616

hadoop fs -put weblog-20180615.txt hdfs://master:9999/user/hadoop-zsz/traffic-analysis

/rawlog/20180616

2.运行spark-preparse-etl工程的PreparseETL

输入参数配置为hdfs://master:9999/user/hadoop-zsz/traffic-analysis/rawlog/20180616

生成以上文件

select count(\*) from rawdata.web where day=20180616 ;

hive查看预处理后的数据

#### 业务ETL（会话切割）

1.运行 spark-sessionization-etl的web-ETL

2. scan 'web-user' 查看用户状态，此时Hbase中的用户状态应该会发生改变

3. 然后执行脚本 web\_etl\_daily.sh

假设20180616跑一半出bug了，紧急修复后不能接着跑，因为Hbase里面的信息以及改变了，这时候需要利用hbase的snapshot

### 重跑逻辑

#### 创建snapshot的类

object HBaseSnapshotAdmin {

private val logger = LoggerFactory.getLogger(classOf[HBaseSnapshotAdmin])

}

class HBaseSnapshotAdmin(conn: Connection) {

import HBaseSnapshotAdmin.\_

def takeSnapshot(snapshotName: String, tableName: String): Unit = {

for (admin <- managed(conn.getAdmin)) {

val table = TableName.valueOf(HbaseConnectionFactory.hbaseTableNamespace, tableName)

admin.disableTable(table)

// check if snapshot with the name already exists

val snapshots = admin.listSnapshots(snapshotName)

if (!snapshots.isEmpty) {

logger.warn(s"Snapshot ${snapshotName} already exists, deleting it and take the latest snapshot again.")

admin.deleteSnapshot(snapshotName)

}

admin.snapshot(snapshotName, table)

admin.enableTable(table)

}

logger.info(s"Successfully took a snapshot ${snapshotName} for table ${tableName}.")

}

}

#### 创建snapshot

在main方法最后

//给表web-user创建snapshot，以便于数据的重跑

val snapshotAdmin = new HBaseSnapshotAdmin(HbaseConnectionFactory.getHbaseConn)

val targetUserTable = System.getProperty("web.etl.hbase.UserTableName", "web-user")

snapshotAdmin.takeSnapshot(s"${targetUserTable}\_snapshot-${dateStr}", targetUserTable)

#### 清空hbase web-user数据 ,重跑第一天数据20180615

truncate ‘web-user’

list\_snapshot

会有一个web-user\_snapshot-20180615 的快照

将快照克隆出来

clone \_snapshot ‘web-user\_snapshot-20180615’, ‘web-user -20180615’

scan ‘web-user -20180615’

假设20180616跑数据时出了问题

disable ‘web-user’

restore\_snapshot ‘web-user\_snapshot-20180615’

将20180615的快照恢复到’web-user’里面

enable ‘web-user’

然后可以开始重跑

最后执行脚本./web\_etl\_daily.sh 20180616 导入20180616的数据

此时HDFS应该有了20180616的数据

/user/hive/warehouse/web.db/session/year=2018/month=201806/day=20180616

在hive执行select day,count(\*) from session group by day;统计数据

## 24.数据倾斜问题解决（过滤方式）

两阶段聚合的方法在这里不适用，因为一个正常用户短时间产生大批量的访问不太现实

有可能出现数据倾斜的地方 都是某个key下数据量过大

一般在groupByKey算子出现

//将parsedLogRDD按照key进行分组

parsedLogRDD.groupByKey

这里在parsedLogRDD生成之前的WebLogparser.parse方法解决

写一个trait，作为不正常用户的探测器

AbnormalUserDetector

WebLogparser继承AbnormalUserDetector

可以在parse方法中将异常流量的用户过滤掉

## 25. 在服务器上跑网站流量离线分析的jobs

### 1.上传weblog-20180617.txt到HDFS

hadoop fs -mkdir hdfs://master:9999/user/hadoop-zsz/traffic-analysis/rawlog/20180617

hadoop fs -put weblog-20180617.txt hdfs://master:9999/user/hadoop-zsz/traffic-analysis/rawlog/20180617

### 2. pom引入打包插件

<build>

<plugins>

<plugin>

<groupId>net.alchim31.maven</groupId>

<artifactId>scala-maven-plugin</artifactId>

<version>3.2.0</version>

<executions>

<execution>

<id>compile-scala</id>

<phase>compile</phase>

<goals>

<goal>add-source</goal>

<goal>compile</goal>

</goals>

</execution>

<execution>

<id>test-compile-scala</id>

<phase>test-compile</phase>

<goals>

<goal>add-source</goal>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

<configuration>

<scalaVersion>${scala.version}</scalaVersion>

</configuration>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<testExcludes>

<testExclude>/src/test/\*\*</testExclude>

</testExcludes>

<encoding>utf-8</encoding>

</configuration>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

<executions>

<execution>

<id>make-assembly</id> <!-- this is used for inheritance merges -->

<phase>package</phase> <!-- 指定在打包节点执行jar包合并操作 -->

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

### 3.直接在父项目backend打包

spark-preparse-etl-1.0-SNAPSHOT-jar-with-dependencies.jar上传到

traffic-analysis/jars下

### 4.先跑预解析任务(原始解析后入Hive)

export HADOOP\_CONF\_DIR=/home/hadoop-zsz/bigdata/hadoop-2.7.7/etc/hadoop/

执行

spark-submit --master yarn \

--class com.zsz.preparser.PreparseETL \

--driver-memory 512M \

--executor-memory 512M \

--num-executors 2 \

--executor-cores 1 \

--conf spark.traffic.analysis.rawdata.input=hdfs://master:9999/user/hadoop-zsz/traffic-analysis/rawlog/20180617 \

/home/hadoop-zsz/traffic-analysis/jars/spark-preparse-etl-1.0-SNAPSHOT-jar-with-dependencies.jar prod

//后面的prod是随便加了一个参数，因为代码设置了参数为空的话就在本地跑

#### Check结果

任务成功后

HDFS中/user/hive/warehouse/rawdata.db/web/year=2018/month=201806/day=20180617

目录会出现相关parquet文件

Hive中会有20180617的数据生成在rawdata.web表中

### 5. 跑网站流量离线分析Spark任务

这里是用standalone方式提交，需要启动spark

MongoDB需要在配置文件里面绑定ip后重启才可以远程访问

spark-submit --master spark://master:7077 \

--class com.zsz.spark.web.WebETL \

--driver-memory 512M \

--executor-memory 1G \

--total-executor-cores 2 \

--executor-cores 1 \

--conf spark.web.etl.inputBaseDir=hdfs://master:9999/user/hive/warehouse/rawdata.db/web \

--conf spark.web.etl.outputBaseDir=hdfs://master:9999/user/hadoop-zsz/traffic-analysis/web \

--conf spark.web.etl.startDate=20180617 \

--conf spark.driver.extraJavaOptions="-Dweb.metadata.mongodbAddr=192.168.101.4 -Dweb.etl.hbase.zk.quorums=master" \

--conf spark.executor.extraJavaOptions="-Dweb.metadata.mongodbAddr=192.168.101.4 -Dweb.etl.hbase.zk.quorums=master -Dcom.sun.management.jmxremote.port=1119 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false" \

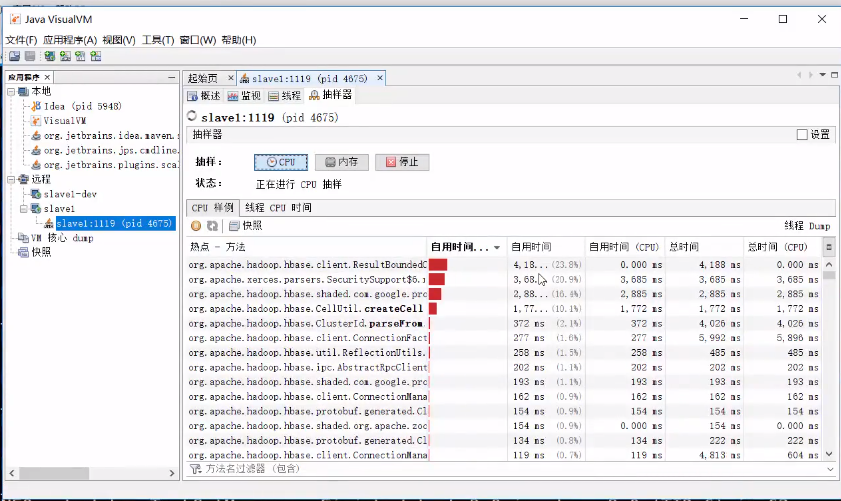
/home/hadoop-zsz/traffice-analysis/jars/spark-sessionization-etl-1.0-SNAPSHOT-jar-with-dependencies.jar prod

//MongoDB的配置driver端和executor都传递

// Dcom.sun.management.jmxremote.port=1119 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false这几个参数是通过非安全模式打开监控

可以用1119端口来监控程序运行情况

利用JDK提供的jvisualvm来监控



可以利用这个工具进行调优

#### Check结果

HDFS下/user/hadoop-zsz/traffic-analysis/web/session/year=2018/month=201806/day=20180617

目录会出现相关文件

#### 执行脚本导入数据

./web\_etl\_daily.sh 20180617

将输出导入Hive

Check结果,可以通过SQL来查看hive中web数据库各个维度表的数据

如select day,count(\*) from web.session group by day;

## 26. SQL分析网站流量的情况

--指标：浏览量(pv)

select sum(pv\_count) as pv\_count from session where profile\_id=702 and day=20180615;

--指标：访客数

select count(distinct(user\_id)) as user\_count from session where profile\_id=702 and day=20180615;

--指标：独立Ip数

select count(distinct(client\_ip)) as client\_ip\_count from session where profile\_id=702 and day=20180615;

--指标：跳出率

select round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate from session where profile\_id=702 and day=20180615;

--指标：平均停留时间

select round(sum(session\_duration) / count(\*),2) as avg\_duration from session where profile\_id=702 and day=20180615;

--基本指标

select

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

count(distinct(client\_ip)) as client\_ip\_count,

round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate,

round(sum(session\_duration) / count(\*),2) as avg\_duration

from session

where profile\_id=702 and day=20180615;

--从时间小时的维度剖析网站流量的基本指标

select

hour\_of\_day,

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

count(distinct(client\_ip)) as client\_ip\_count,

round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate,

round(sum(session\_duration) / count(\*),2) as avg\_duration

from session

where profile\_id=702 and day=20180615

group by hour\_of\_day order by hour\_of\_day asc;

--从时间天的维度剖析网站流量的基本指标

select

day,

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

count(distinct(client\_ip)) as client\_ip\_count,

round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate,

round(sum(session\_duration) / count(\*),2) as avg\_duration

from session

where profile\_id=702 and day between 20180615 and 20180623

group by day order by day asc;

--从来源类型的维度剖析网站流量的基本指标

select

source\_type,

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

count(distinct(client\_ip)) as client\_ip\_count,

round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate,

round(sum(session\_duration) / count(\*),2) as avg\_duration

from session

where profile\_id=702 and day between 20180615 and 20180623

group by source\_type;

--访问分析：

with url\_contribution as (

select page\_view\_referrer\_url, count(\*) as cnt

from pageview

where profile\_id=702 and day between 20180615 and 20180623

group by page\_view\_referrer\_url)

select

page\_url,

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

sum(if(is\_exit\_page, 1, 0)) as exit\_page\_count,

sum(nvl(cnt, 0)) as contribution\_count,

round(sum(page\_duration) / count(\*),2) as avg\_duration

from pageview as pv left join url\_contribution as uc on pv.page\_url = uc.page\_view\_referrer\_url

where profile\_id=702 and day between 20180615 and 20180623

group by page\_url;

--转化分析

select

day,

count(distinct(server\_session\_id)) as visits,

count(distinct(user\_id)) as user\_counts,

count(\*) as conversion\_counts,

round(count(\*) / count(distinct(server\_session\_id)), 2) as conversion\_rate,

sum(conversion\_value) as conversion\_values

from conversion

group by day;

--按浏览器的维度剖析

select

browser\_brief,

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

count(distinct(client\_ip)) as client\_ip\_count,

round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate,

round(sum(session\_duration) / count(\*),2) as avg\_duration

from session

where profile\_id=702 and day between 20180615 and 20180623

group by browser\_brief;

select

country,

sum(pv\_count) as pv\_count,

count(distinct(user\_id)) as user\_count,

count(distinct(client\_ip)) as client\_ip\_count,

round(sum(if(is\_bounced, 1, 0)) / count(\*),2) as bounced\_rate,

round(sum(session\_duration) / count(\*),2) as avg\_duration

from session

where profile\_id=702 and day between 20180615 and 20180623

group by country;

## 27.真实的数据量和配置

### 每天需要处理的数据量

1条日志 -> 1kb

1.2亿pv -> 114G

114G \* 4 = 500G

### 真实环境的配置

（2小时内完成）

spark-submit --master yarn \

--class com.twq.spark.web.WebETL \

--driver-memory 10G \

--executor-memory 24G \

--num-executors 20 \

--executor-cores 4 \

--conf spark.web.etl.inputBaseDir=hdfs://master:9999/user/hive/warehouse/rawdata.db/web \

--conf spark.web.etl.outputBaseDir=hdfs://master:9999/user/hadoop-zsz/traffic-analysis/web \

--conf spark.web.etl.startDate=20180617 \

--conf spark.driver.extraJavaOptions="-Dweb.metadata.mongodbAddr=192.168.1.102 -Dweb.etl.hbase.zk.quorums=master" \

--conf spark.executor.extraJavaOptions="-Dweb.metadata.mongodbAddr=192.168.1.102 -Dweb.etl.hbase.zk.quorums=master -Dcom.sun.management.jmxremote.port=1119 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false" \

/home/hadoop-zsz/traffice-analysis/jars/spark-sessionization-etl-1.0-SNAPSHOT-jar-with-dependencies.jar prod

### 数据存储在Hive中的数据量

rawdata.web ==> 300G

web.pageview ==> 150G

web.session ==> 120G

web.mouseclick ==> 190G

web.heatbeat ==> 160G

web.conversion ==> 90G

1天1T的数据

### HBase中的数据量

每天2000万的user, 每一个user的信息大小为0.2KB

总大小约为：4GB

每天增量的user为500万左右的user，大小大概为：1GB左右

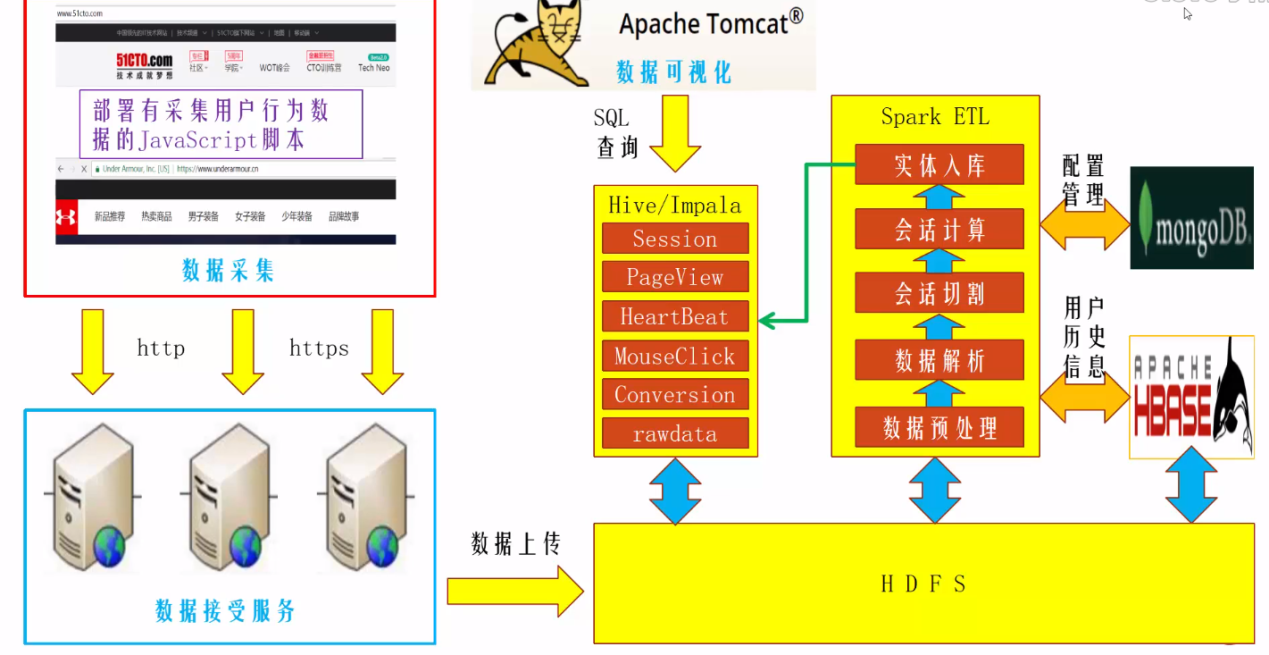
### 集群的规模

Hadoop集群：52个节点

HBase集群：23个节点

## 28．网站流量分析团队组合

### 项目流程



团队组合：

数据采集，1个负责人带一个2个前端开发

数据接收服务（API团队），4个人

数据可视化（API团队），4个

### 大数据后台

6开发+2测试+2产品