**数据处理与分析指导说明书**

目录

[任务1. 业务数据清洗和点击流日志清洗（ETL） 1](#_Toc44278681)

[任务描述 1](#_Toc44278682)

[任务指导 2](#_Toc44278683)

[任务实现 2](#_Toc44278684)

[任务2. 购书转化率分析 8](#_Toc44278685)

[任务描述 8](#_Toc44278686)

[任务指导 8](#_Toc44278687)

[任务实现 8](#_Toc44278688)

# 业务数据清洗和点击流日志清洗（ETL）

## 任务描述

1. 业务数据清洗：去重、空值处理等
2. 日志数据清洗：解析IP、过滤、添加SessionID等
3. 网站分析的指标

## 任务指导

1. 业务数据清洗：去重、空值处理等
2. 日志数据清洗：解析IP、过滤、添加SessionID等
3. 网站分析的指标

## 任务实现

1. 业务数据清洗：去重、空值处理等

* 基本数据清洗

数据清洗最好的方式通常是SQL，前面已经开发了一个执行HiveQL的功能模块，可以利用该模块进行数据清洗的工作。

在这里我们并不需要开发其他的模块，只需利用数据清洗工具将想要执行的HQL配置到配置文件即可。例如下面的配置文件：

<?xml version="1.0" encoding="utf-8" ?>

<root>

<!-- 使用HQL进行数据清洗 -->

<job type="etl\_db">

<hql>数据清洗的HQL</hql>

<hql>……</hql>

<hql>……</hql>

</job>

</root>

* **数据去重**

数据仓库的特点之一是“时变性”，一般情况下，记录都包含某种形式的时间标志用以说明在某一时间的是准确的，通常都会利用记录的时间戳字段来获取记录的时间信息来进行导入，在这个过程中，就可能会产生重复的数据。

例如，在数据库中的Order表中有一条主键为1的记录，它的ModifyTime为1，这时系统会将其导入到时间为1的表分区中。第二天，数据库中的记录被修改了，相应的ModifyTime字段被修改为2（比如第一天下单，第二天修改订单状态为出库），那么该条记录将再次被导入到时间为2的表分区中，此时在Hive中的Order表中存在了两条主键为1的记录，重复数据就是这样产生的，这会干扰分析结果，所以在开始数据分析前必须对数据进行去重。

**例如如下的数据：**

|  |  |
| --- | --- |
| PKID | MODIFYTIME |
| 1 | 2019—03-10 |
| 1 | 2019—03-11 |
| 2 | 2019—03-11 |
| 3 | 2019—03-10 |
| 3 | 2019—03-11 |

**希望去重后的数据：**

|  |  |
| --- | --- |
| PKID | MODIFYTIME |
| 1 | 2019—03-11 |
| 2 | 2019—03-11 |
| 3 | 2019—03-11 |

* + **第一种去重的方法：**

一个简单的方案是，对整个结果集进行分组排序，加上限制条件ROWNUM=1，那么重复的数据就将会被过滤掉。ROWNUM函数在实际的开发中使用频率非常高，但是Hive本身并没有提供这一功能，需要自定义UDF的方式将其实现。

* + **自定义RowNumUDF类，实现ROWNUM功能：**

public class HiveDedupUDF extends UDF {

public static String signature = "-";

public static int order = -1;

public int evaluate(Text text) {

if (text != null) {

// 分组排序的依据，列名，通常为主键

String colName = text.toString();

// 处理第一条数据

if (signature == "-") {

// 记下分组排序的字段:主键,并将rownum设为1

signature = colName;

order = 1;

} else {

// 首先比对是否和上一条的主键相同

if (signature.equals(colName)) {

order++;

} else {

signature = colName;

order = 1;

}

}

}

return order;

}

}

* + **编写完UDF后，导出为一个.jar包，并在Hive中注册：**

hive> add jar /home/hadoop/rownum.jar

Added /home/Hadoop/rownum.jar to class path

Added resource: /home/Hadoop/rownum.jar

hive> CREATE TEMPORARY FUNCTION rownum AS 'com.udf.RowNumUDF'

OK

* + **使用自定义rownum函数，进行数据去重**。需要注意的是，在进入rownum函数之前，必须完成相同PKId的数据按照ModifyTime排序。去重后按照ModifyDate重新插入到相应的分区。

<root>

<!-- 使用HQL进行数据去重 -->

<job type="etl\_data\_dedup">

<job type="etl\_data\_dedup">

<hql>

"insert overrite table orders PARTITION(dt) select \* from (select \* from (select \* from orders where dt=$dt union all select \* from orders where dt<$dt) t1 DISTRIBUTE BY orderid SORT BY orderid, update\_time) t2 WHERE rownum(orderid)=1"

</hql>

</job>

</root>

* + **第二种去重的方法：**

第一种方式可以很好地完成数据去重的任务，但是还需要编写UDF。如果只用HiveQL，也是可以完成数据去重的任务。

<?xml version="1.0" encoding="utf-8" ?>

<root>

<!-- 使用HQL进行数据去重 -->

<job type="etl\_data\_dedup">

<!--公式：结果=（全量数据减去（全量数据与增量数据的交集））再与增量数据取并集 -->

<hql>"insert overwrite table test.dedup\_orders select \* from (select t1.\* from (select \* from test.orders where dt \$dt) t1 left outer join (select \* from test.orders where dt=$dt) t2 on t1.orderid=t2.orderid where t2.orderid is null UNION ALL select \* from test.orders where dt=$dt) t3"

</hql>

</job>

</root>

第二种方式有两个前提条件，默认增量数据是最新的，并且增量数据中没有重复的数据。从执行效率方面，第二种方式会好于第一种方式。

1. **日志数据清洗：解析IP、过滤、添加Session等**

* 一条标准的Apache服务器日志格式为：

|  |  |  |
| --- | --- | --- |
| 1 | 10.178.123.55 | 远端主机 |
| 2 | - | 远端登录名 |
| 3 | - | 远程用户名 |
| 4 | [11/Dec/2018:10:00:32 +0800] | 服务器接收时间 |
| 5 | "GET /\_\_utm.gif HTTP/1.1" | 请求的第一行(请求的路径) |
| 6 | 200 | 最后请求的状态 |
| 7 | 35 | 以CLF格式显示的除HTTP头以外传送的字节数 |
| 8 | "http://easternmiles.ceair.com/flight/index.html" | 上一个访问页面 |
| 9 | "Mozilla/5.0 (Windows NT 6.1;WOW64) AppleWebkit/  537.36 (KHTML,like Gecko) Chrome/31.0.1650.63 Safari/537.36" | 用户浏览器信息 |
| 10 | "BIGipServermu\_122.119.122.14=192575345.20480.000;Webtrends=  120.196.145.58.1386724976245806; " | Cookie信息 |
| 11 | 1482 | 接收的字节数，包括请求头的数据，且不能为零 |
| 12 | 352 | 发送的字节数，包括请求头的数据，且不能为零 |
| 13 | - | %{X-Forwarded-For}i |
| 14 | easternmiles.ceair.com | 访问主机地址（域名） |
| 15 | 749 | 服务器处理本请求所用的时间，以微秒为单位 |

* 我们要获取的信息：

|  |  |
| --- | --- |
| 字段 | 说明 |
| ipaddress | 可以直接从点击流日志中获取 |
| uniqueid | 从点击流日志的Cookie信息中获取 |
| url | 用户访问的链接 |
| sessionid | 由uniqueid和sessionTimes组成生成 |
| sessiontimes | 从点击流日志的Cookie中获取 |
| areaaddress | 通过解析IP地址获取 |
| localaddress | 通过解析IP地址获取 |
| browsertype | 可以直接从点击流日志中获取 |
| operationsys | 可以直接从点击流日志中获取 |
| referurl | 可以直接从点击流日志中获取 |
| receivetime | 可以直接从点击流日志中获取 |
| userid | 从点击流日志的Cookie中获取 |
| csvp | 通过对receiveTime进行排序获取 |

* **在Hive创建一个clickstream\_log表，用来保存上述信息**

CREATE TABLE clickstream\_log(

ipAddress STRING,

uniqueId STRING,

url STRING,

sessionId STRING,

sessionTimes INT,

areaAddress STRING,

localAddress STRING,

browserType STRING,

operationSys STRING,

referUrl STRING,

receiveTime STRING,

userId STRING,

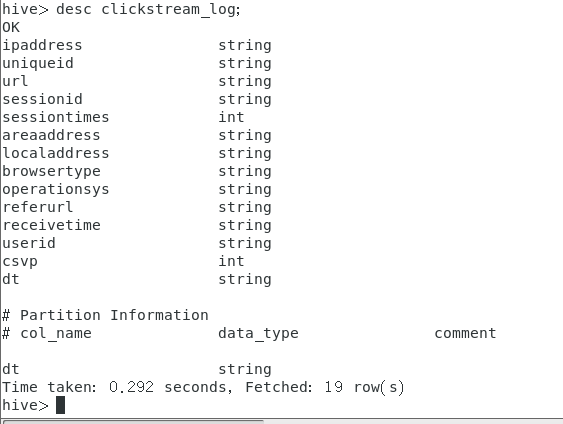
csvp INT

)

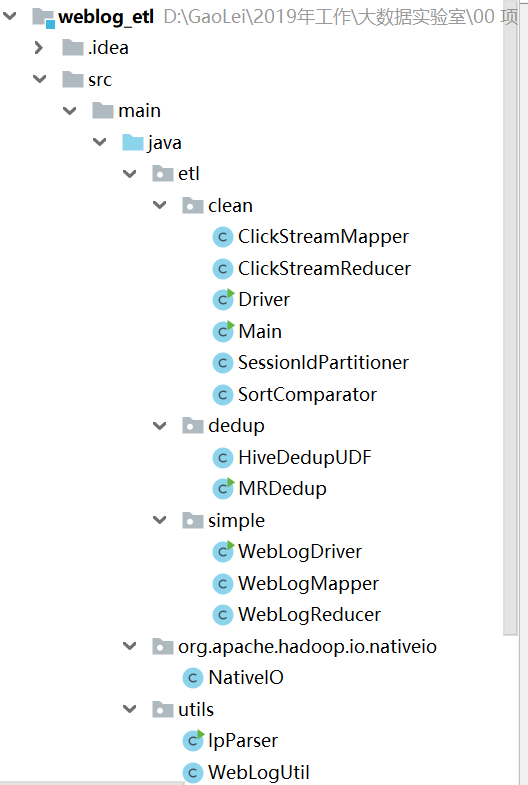
PARTITIONED BY (dt STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

**如下图所示：**



* **编写MapReduce实现获取上述信息的任务（代码参见weblog\_etl示例代码）**



* + 编写IP地址解析器IpParser类
  + 编写Mapper类，获取除了csvp字段外的所有字段
  + 编写Partitioner类，控制Shuffle过程，实现按照SessionId分发的规则
  + 编写Comparator类，控制排序过程，实现按照SessionId排序，当SessionId一致时按照receiveTime排序的规则
  + 编写Reducer类，在经过Shuffle和Sort后，数据已经按照要求排好顺序，这个顺序就是我们需要的csvp，现在只需按照这个顺序对csvp赋值即可
  + 编写Driver类，
  + 编写etl\_clickstream.py文件，通过Python调用jar文件

1. **网站分析的指标（需要学生自行扩展功能，实现对业务数据和日志数据的整体分析，并形成报表）**

当对日志数据和业务数据进行ETL后，我们就有了网站分析的依据，网站分析是一种研究和提升在线体验的方法，可以帮助我们判断搜索引擎营销策略在捕捉潜在受众上是否有效，判断在社交媒体口碑建设上是否物有所值，判断访问的体验是否友好等等。

* 网站分析常见的有以下一些指标：
  + 网站最受欢迎的页面
  + 平均访问时长和访客回报率
  + 带来最多流量的媒介来源或渠道
  + 访客的地理分布和他们的语言设置
  + 网页的“粘性”
  + 网站产生的收入
  + 最有价值的访客来自哪里
  + 最有价值的页面是哪个
  + 访客的转换率
  + 内部搜索的有效性
  + ……等等
* 在下面的任务中，以**购书转化率分析**为例，展示网站常见的一种分析指标。

# 购书转化率分析

## 任务描述

**内 容**：

完成购书转化率分析。购书转化率分析就是要求出从商品页面转到订单页面的点击数，以及从订单页面转到支付完成页面的点击数，并求出每一步的比率，如同一个漏斗，即跳转到支付页面的用户必然是从商品页面跳转过来，在支付完成页面的用户必然是从支付页面跳转来，活动的步骤就像一个漏斗一样，将未跳转下一步的行为筛选出来 。

在活动相邻的两个页面之间是可以经过其他页面的，如在图书展示页面和订单页面之间，用户还可能浏览了其他页面。所以漏斗的统计是在同一个session中完成的，且一个session中可以多次完成漏斗所代表的销售活动，例如，某个用户在一次session中购买了3本书。

**学 时**：

**知识点**：购书转化率计算方式、使用Hive提取字段、使用MapReduce完成漏斗模型的统计

**重点**：使用MapReduce完成漏斗模型的统计

**难点**：数据准备

## 任务指导

完成购书转化率分析功能模块的主要步骤：

* 读取配置文件，获取程序运行的基本参数，如统计的时间跨度、表示漏斗的URL连接的正则表达式，如果参数有误则程序终止运行
* 将所需的数据从数据仓库中提取出来，保存到Hive的临时表中，作为下一步的输入。
* 开发MapReduce程序完成漏斗模型的统计，将结果保存到HDFS作为中间结果
* 使用Hive将HDFS的中间结果进行处理，加载到中间结果表中，经过汇总后得到最后结果

## 任务实现

1. 读取配置文件

* 同其他功能模块一样，此模块也有一个配置文件，在conf目录创建conversion.xml文件

<?xml version="1.0" encoding="utf-8" ?>

<root>

<!-- 漏斗的所有URL正则表达式，具体值根据实际情况而定 -->

<urls>

<!--漏斗的第1个URL的正则表达式-->

<url>url1</url>

<!--漏斗的第2个URL的正则表达式-->

<url>url2</url>

<!--漏斗的第3个URL的正则表达式-->

<url>url3</url>

</urls>

</root>

url的数目根据实际情况决定，如url1表示 漏斗的第一个URL的正则表达式，以此类推。

* 编写buildConversion.py用于解析配置文件

# -\*- coding:UTF-8 -\*-

from com.utils.py\_env import PROJECT\_CONF\_DIR

import xml.etree.ElementTree as ET

def resolve\_conf(type, dt):

# 配置文件的路径

confFile = PROJECT\_CONF\_DIR + "conversion.xml"

# 解析配置文件

xmlTree = ET.parse(confFile)

eles = xmlTree.findall('./urls')

rootEle = eles[0]

# 用来保存漏斗的URL的集合

urls = []

for ele in rootEle.getChildren():

if ele.tag == 'url':

url = ele.text.strip()

if url != None or url != '':

print(url)

urls.append(url)

if len(urls) == 0:

raise Exception('缺少参数，程序终止')

return urls

1. 准备所需的数据

转化率分析的数据从数据仓库中的Clickstream\_log表中提取，并保存到Hive的临时表conversion\_input中。

* 创建表的语句

create table conversion\_input(

url STRING,

uuid STRING,

sessionId STRING,

csvp INT

) PARTITIONED BY(dt STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

sessionid：表示用户的SessionId

csvp：表示用户访问的先后顺序

* 提取所需的数据

def extract\_data(start, end):

hql = "insert into table conversion\_input partition (dt='" + start + "-" + end + "')

select url,uuid,sessionid,csvp from clickstream\_log where where dt>= " + start + " and dt<= " + end

HiveUtil.execute\_shell(hql)

1. 编写MapReduce进行转化率分析

转化率统计的范围是一个session内，所以在map阶段使用sessionid进行分发，利用Hadoop默认的排序功能将同一个session的数据按照csvp进行排序，在reduce阶段按照漏斗的逻辑进行统计。

* 编写Mapper类

Mapper类的主要作用是：按照Session+csvp的规则组合成新的键，并输出。

public class ConversionMapper extends Mapper<LongWritable, Text, Text, Text> {

public static final String SEPARATOR = "@";

//用正则表达式指定字符串是否满足表达式

public static boolean regex(String value, String regex) {

Pattern p = Pattern.compile(regex);

Matcher m = p.matcher(value);

return m.find();

}

@Override

protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

//从Context中获取表示漏斗的url正则表达式

String[] desUrlsRegex = context.getConfiguration().get("urls").split("");

//如果表示漏斗的url为空，则返回

if (desUrlsRegex == null) {

return;

}

String[] logInfos = value.toString().split("\t");

String url = logInfos[0];

//如果该记录未访问目标地址，则丢弃

int flag = 0;

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

if (regex(url, desUrlsRegex[i])) {

break;

} else {

flag += 1;

}

}

if (flag == desUrlsRegex.length) {

return;

}

//获取uuid

String uuid = logInfos[1];

//获取SessionId

String sessionId = logInfos[2];

try {

//获取csvp

int csvp = Integer.parseInt(logInfos[3]);

//将SessionId和csvp组成新的key

String newKey = sessionId + SEPARATOR + csvp;

//剩下的部分作为新的value

String newValue = uuid + SEPARATOR + url;

context.write(new Text(newKey), new Text(newValue));

} catch (Exception e) {

e.printStackTrace();

return;

}

}

}

* 编写Partitioner类

Partitioner类的主要作用是：实现按照SessionId分发的规则。

public class ConversionPartitioner extends Partitioner<Text, Text> {

public static final String SEPARATOR = "@";

public int getPartition(Text key, Text value, int parts) {

String sessionId = "-";

if (key != null) {

sessionId = key.toString().split(SEPARATOR)[0];

}

//将SessionId从0到Integer的最大值进行散列

int reducerNum = (sessionId.hashCode() & Integer.MAX\_VALUE) % parts;

return reducerNum;

}

}

在getPartition函数中，得到SessionId后，将其散列到0—int最大值的范围，并再对parts取模，得到reducer的编号，0为第一个reducer，1为第二个reducer，等等，这样既保证了同一个SessionId的数据进入同一个reducer，又保证了reducer的编号超过reducer的个数。

* 编写SortComparator类

SortComparator类的主要作用是：实现先按照SessionId排序，当SessionId一致时，按照cvsp排序的规则。

public class ConversionComparator extends WritableComparator {

protected ConversionComparator() {

super(Text.class, true);

}

public static final String SEPARATOR = "@";

@Override

public int compare(WritableComparable a, WritableComparable b) {

String[] comp1 = a.toString().split(SEPARATOR);

String[] comp2 = b.toString().split(SEPARATOR);

long result = 1;

if (comp1 != null && comp2 != null) {

//比较SessionId

result = comp1[0].compareTo(comp2[0]);

//如果SessionId相同，则比较csvp

if (result == 0 && comp1.length > 1 && comp2.length > 1) {

long csvp1 = 0;

long csvp2 = 0;

try {

//获取csvp

csvp1 = Long.parseLong(comp1[1]);

csvp2 = Long.parseLong(comp2[1]);

result = csvp1 - csvp2;

if (result == 0) {

//如果csvp相等，则返回0

return 0;

} else {

//如果a的csvp大，则返回1，否则返回-1

return result > 0 ? 1 : -1;

}

} catch (Exception e) {

return 1;

}

}

return result > 0 ? 1 : -1;

}

return 1;

}

}

* 编写Reducer类

Reducer类的主要作用是：按照漏斗模型规则进行统计，由于数据已经经过了自定义的Shuffle和Sort，已经按照SessionId汇集在一起，并且按照csvp的大小排序，根据前面介绍的漏斗模型的定义，reduce函数输出的规则是SessionId+uuid+漏斗的进度。

例：sid03 uid02 3，表示记录属于漏斗01的第3步，用户02在第3个Session中完成。

public class ConversionReducer extends Reducer<Text, Text, NullWritable, Text> {

public static final String SEPARATOR = "@";

//表示前一条记录的SessionId

public static String preSessionId = "not set";

//表示漏斗的步骤，如，1表示漏斗的第一步

public static int process = 0;

//用正则表达式指定字符串是否满足表达式

public static boolean regex(String value, String regex) {

Pattern p = Pattern.compile(regex);

Matcher m = p.matcher(value);

return m.find();

}

@Override

protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

//从Context中获取表示漏斗的url正则表达式

String[] desUrls = context.getConfiguration().get("urls").split(" ");

//获取SessionId

String sessionId = key.toString().split(SEPARATOR)[0];

String value = values.iterator().next().toString();

//获取url

String url = value.split(SEPARATOR)[1];

//获取uuid

String uuid = value.split(SEPARATOR)[0];

//如果是第一次执行reduce函数

if (preSessionId.equals("not set")) {

//记录当前SessionId

preSessionId = sessionId;

//初始化进度

process = 0;

if (regex(url, desUrls[0])) {

process = 1;

String result = sessionId + "\t" + uuid + "\t" + process;

context.write(NullWritable.get(), new Text(result));

} else {

return;

}

} else {

//当presession-session时，说明正在进行漏斗的比较中

if (preSessionId.equals(sessionId)) {

//一个漏斗比较完成

if (process == desUrls.length) {

//开始新的漏斗比较

process = 0;

//当前进度为0时，只需比较第一个漏斗

if (regex(url, desUrls[0])) {

process = 1;

String result = sessionId + "\t" + uuid + "\t" + process;

context.write(NullWritable.get(), new Text(result));

} else {

return;

}

return;

} else {

if (regex(url, desUrls[process])) {

//更新进度

process++;

String result = sessionId + "\t" + uuid + "\t" + process;

context.write(NullWritable.get(), new Text(result));

}

}

} else {//如果是一个新的SessionId

preSessionId = sessionId;

process = 0;

if (regex(url, desUrls[0])) {

process = 1;

String result = sessionId + "\t" + uuid + "\t" + process;

context.write(NullWritable.get(), new Text(result));

} else {

return;

}

}

}

}

}

* 编写Driver类

public class Driver {

public static final String SEPARATOR = "@";

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

Configuration configuration = new Configuration();

if (args.length <= 2) {

System.out.println("参数不完整，参数1：输入路径；参数2：输出路径；参数3：漏斗目标url");

return;

}

String inputPath = args[0];

String outputPath = args[1];

//保存表示漏斗的url的正则表达式

ArrayList<String> hoppers = new ArrayList<String>();

for (int i = 2; i < args.length - 1; i++) {

hoppers.add(args[i]);

}

String urls = "";

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

urls += hoppers.get(i);

if (i != hoppers.size() - 1) {

urls += SEPARATOR;

}

}

//将漏斗URL保存到configuration对象中，供所有Map任务和Reduce任务使用

configuration.set("urls", urls);

Job job = Job.getInstance(configuration, "conversion");

job.setJarByClass(Driver.class);

FileInputFormat.addInputPath(job, new Path(inputPath));

FileOutputFormat.setOutputPath(job, new Path(outputPath));

job.setMapperClass(ConversionMapper.class);

job.setReducerClass(ConversionReducer.class);

//手动设置Reducer个数，需根据集群计算能力综合考虑

job.setNumReduceTasks(4);

job.setOutputFormatClass(TextOutputFormat.class);

job.setPartitionerClass(ConversionPartitioner.class);

job.setSortComparatorClass(ConversionComparator.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

* 通过Python模块调用打包的MapReduce程序

import os

from com.utils.py\_env import HADOOP\_PATH, PROJECT\_LIB\_DIR

def count\_urls(start, end, urls):

# MapReduce作业的输入路径，即conversion\_input表的HDFS地址

inputPath = "/user/warehouse/conversion\_input/dt=" + start + "-" + end

# MapReduce作业的输出路径

outputPath = "/user/warehouse/conversion\_out"

# 删除上一次作业输出目录

os.system(HADOOP\_PATH + "hadoop dfs -rmr " + outputPath)

# 将表示漏斗的正则表达式拼装成一个字符串，作为参数传给MapReduce作业

urlstr = ""

for i in range(len(urls)):

if (i == len(urls) - 1):

urlstr += urls[i]

else:

urlstr += urls[i] + " "

# 拼接成shell命令

shell = HADOOP\_PATH + "hadoop jar " + PROJECT\_LIB\_DIR + "conversion.jar com.conversion.mr.Driver " + inputPath + " " + outputPath + " " + urlstr

# 执行命令

os.system(shell)

1. 汇总最终结果

此步骤的目的是将MapReduce作业输出的统计结果加载到中间结果表，进行汇总得到最终结果，并输出到最终结果表。

* 创建中间结果表：

create table conversion\_middle\_result(

sessionid STRING,

uuid STRING,

process STRING

) PARTITIONED BY(dt STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

* 创建最终结果表：

create table conversion\_result(

process STRING,

counts INT,

countu INT

) PARTITIONED BY(dt STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

* 编写函数，将MapReduce作业的输出结果加载进中间结果表，然后进行汇总得到最终结果表。

import os

from com.utils.HiveUtil import HiveUtil

from com.utils.py\_env import HADOOP\_PATH

def get\_result(start, end, output):

# 最终结果表的分区

dt = start + "-" + end

# 删除作业成功的标志文件和作业的日志文件

shell = HADOOP\_PATH + "hadoop dfs -rm " + output + "/\_SUCCESS"

os.system(shell)

shell = HADOOP\_PATH + "hadoop dfs -rmr " + output + "/\_logs"

os.system(shell)

# 将临时结果加载到中间结果表

hql = "load data inpath '" + output + "' overwrite into table conversion\_middle\_result partition (dt= " + dt + ")"

HiveUtil.execute\_shell(hql)

# 对中间结果进行汇总，并写入到最终结果表

hql = "insert into table conversion\_result partition (dt='" + start + "-" + end + "') select process,count(process),count(distinct(uuid)) from conversion\_middle\_result where dt = " + dt + "group by process"

HiveUtil.execute\_shell(hql)

1. 对上面的流程进行整合

import sys

from com.conversion\_rate\_analysis import buildConversion, buildConversionInput, buildConversionHadoopShell, buildConversionResult

if \_\_name\_\_ == '\_\_main\_\_':

# 统计时间范围的开始时间，通过命令行参数传入

start = sys.argv[1]

# 统计时间范围的结束时间，通过命令行参数传入

end = sys.argv[2]

# 解析配置文件

urls = buildConversion.resolve\_conf()

# 提取所需数据

buildConversionInput.extract\_data(start, end)

# 通过MapReduce进行统计

buildConversionHadoopShell.count\_urls(start, end, urls)

# 对中间结果进行汇总并得到最后结果表

buildConversionResult.get\_result(start, end)