南京大學

Mapreduce 课程设计

组　　长　　尹俊宇：161130118

组　　员　　刘扬：171850524

组　　员　　宋昱豪：171850505

题　　目　　日志统计分析

目　录

1 摘要

1.1 背景

电商公司越来越重视接口访问日志的利用，从日志文件里边可以获取到接口的访问性能、访问频率、访问来源，统计有以下的意义：1、能够快速获取接口访问性能是否下降，或者接口访问频率异常。2、结合公司的访问量，可以预估举行促销活动时，需要增加机器的数量。3、接口修改后，是否出现波动等。

1.2 任务需求

1. 任务 1：统计日志中各个状态码（200, 404 500 ）出现总的频次，并且按照小时时间窗输出各个时间段各状态码的统计情况。

2. 任务 2：统计每个 IP 访问总的频次，并且按照小时时间窗，输出各个时间段各个 IP 访问的情况。

3. 任务 3：统计每个接口请求的 URL ，如 /tour/category/query 访问的总的频次，然后对接口根据访问频次降序排序。

4. 任务 4：统计每个接口的平均响应时间，并且以接口为分组，按照小时时间窗，输出各个时间段各个接口平均的响应时间。

5. 任务 5：接口访问频次预测、接口访问频次预测，给 2015-09-08.log 到 2015-09-21.log 共 14 天的日志文件，作为训练数据，设计预测算法来预测下一天 2015-09-22 每个小时窗内每个接口（请求的 URL ）的访问总频次。

1.3 数据结构分析

课题给出了形式如下的数据, 数据含义如图二所示：

```
172.22.49.44 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 124 8
172.22.49.43 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 6304 11
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/hotel-search/nearby-scenic/query HTTP/1.1" GET 200 192 2
172.22.49.44 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 144 6
172.22.49.46 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/ids/query HTTP/1.1" GET 200 4248 3
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 124 2
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 124 2
172.22.49.89 [08/Sep/2015:00:19:16 +0800] "GET /tour/hotel-search/query HTTP/1.1" GET 200 15357 13
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/guide/query HTTP/1.1" GET 200 100 3
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 124 2
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/ids/query HTTP/1.1" GET 200 18936 5
172.22.49.46 [08/Sep/2015:00:19:16 +0800] "GET /tour/guide/query HTTP/1.1" GET 200 100 7
172.22.49.83 [08/Sep/2015:00:19:16 +0800] "GET /tour/hotel-search/query HTTP/1.1" GET 200 18844 75
172.22.49.43 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 12701 63
172.22.49.86 [08/Sep/2015:00:19:16 +0800] "GET /tour/hotel-search/query HTTP/1.1" GET 200 12264 67
172.22.49.44 [08/Sep/2015:00:19:16 +0800] "GET /tour/guide/query HTTP/1.1" GET 200 9360 3
172.22.49.55 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 8712 60
172.22.49.57 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 27840 8
172.22.49.46 [08/Sep/2015:00:19:16 +0800] "GET /tour/product/query HTTP/1.1" GET 200 100 2
172.22.49.46 [08/Sep/2015:00:19:16 +0800] "GET /tour/product/query HTTP/1.1" GET 200 100 1
172.22.49.88 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 124 2
172.22.49.46 [08/Sep/2015:00:19:16 +0800] "GET /tour/product/query HTTP/1.1" GET 200 100 2
172.22.49.44 [08/Sep/2015:00:19:16 +0800] "GET /tour/product/query HTTP/1.1" GET 200 100 1
172.22.49.89 [08/Sep/2015:00:19:16 +0800] "GET /tour/product/query HTTP/1.1" GET 200 100 1
172.22.49.46 [08/Sep/2015:00:19:16 +0800] "GET /tour/category/query HTTP/1.1" GET 200 128 1
172.22.49.46 [08/Sep/2015:00:19:17 +0800] "GET /tour/hotel-search/query HTTP/1.1" GET 200 100 3
172.22.49.46 [08/Sep/2015:00:19:17 +0800] "GET /tour/category/query HTTP/1.1" GET 200 144 6
172.22.49.41 [08/Sep/2015:00:19:17 +0800] "GET /tour/category/weekendproduct/query HTTP/1.1" GET 200 100 6
```

图 1: 数据概览

172.22.49.26	调用方的 IP
[16/Sep/2015:00:22:23 +0800]	调用的时间
GET /tour/category/query HTTP/1.1	HTTP 请求，其中/tour/category/query 是请求的 URL，URL 不同，则视为不同的接口
GET	HTTP METHOD
200	HTTP 状态码，其他经常见到的还有 404，500
156	RESPONSE 返回的字节长度
2	本次请求响应的时间，单位为毫秒

图 2: 数据具体含义

2 基本任务

2.1 任务 1：状态码频次统计

任务分析 任务要求统计各个状态码出现的总的频次，并按照小时时间窗输出各个时间段各种状态码的统计情况。初步分析，需要按照时间窗口来统计状态码出现的频次，并且统计出现的总数即可。由于实验目标的需求，本次任务中只需要关注时间与状态码便可以。本次实验输出结果为单一文件，考虑到输出 24 小时时间窗与三种状态码总数，一共只需要输出 27 行，输出内容较小，要求单文件输出。

Mapper 设计 由于本次任务只需要统计时间与状态码，我们以空格分割数据。用相对位置来取得时间与状态码。由于本次任务只需求对状态码出现次数做统计，所以我们将时间和状态码集成为 key 值，以数字 1 作为 value 值传出。

```
class Task1Mapper extends Mapper<Object, Text, Text, IntWritable> {
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //super.map(key, value, context);
        //<key, value> = <time-code, frequency>
        //FileSplit fileSplit = (FileSplit)context.getInputSplit();
        //String fileName = fileSplit.getPath().getName();
        String[] tokens = value.toString().split(" ");
        //tokens[1]包含时间,只需要h; tokens[7]代表状态码
        //tokens[7]并不代表状态码, 使用相对位置
        String outKey = tokens[1].split(":")[1] + "#" + tokens[tokens.length-3];
        context.write(new Text(outKey), new IntWritable(1));
    }
}
```

图 3: Mapper 设计

Combiner 设计 任务 1 中，我们需要对时间窗内的状态码的数量做一个统计，即求和操作，求和操作是一个满足结合律的操作，因此我们可以使用 combiner 来优化 MapReduce，用本地 reduce 减少数据在网络中传输的量，优化程序的性能。

```
class Task1Combiner extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        //super.reduce(key, values, context);
        Iterator<IntWritable> i = values.iterator();
        int sum = 0;
        while(i.hasNext()) {
            sum += i.next().get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

图 4: Mapper 设计

Partitioner 设计 任务 1 中 Partitioner 的设计方面只需要考虑在 Mapper 处理好数据之后，如何选择不同的 Reducer 来进行处理，从而均匀的将 Mapper 的输出结果

分布在 reducer 上执行。

本次任务中我们需要统计的是时间窗口内的状态码的数量，所以很显然同一个时间窗口内的数据应该放在相同的 reducer 节点中。所以我们按照时间窗口来划分到 24 个 reducer 中。

```java
//按照时间进行划分
class Task1Partitioner extends HashPartitioner<Text, IntWritable> {
    @Override
    public int getPartition(Text key, IntWritable value, int numReduceTasks) {
        String time = key.toString().split("#")[0];
        return super.getPartition(new Text(time), value, numReduceTasks);
    }
}
```

图 5: Mapper 设计

Reducer 设计　Reducer 接受数据并对数据进行处理，首先将键值对分割，得到当前键值对的时间码，从而对各个时间段内三种状态码的出现次数进行统计。

```java
protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
    //super.reduce(key, values, context);
    String[] tokens = key.toString().split("#");
    int curTime = Integer.parseInt(tokens[0]);
    String stateCode = tokens[1];
    if(curTime != time && time != -1) {
        String outKey = String.format("%d:00-%d:00 200:%d 400:%d 404:%d",time, (time+1)%24, sum1, sum2, sum3);
        context.write(new Text(outKey), NullWritable.get());
        sum1 = 0;
        sum2 = 0;
        sum3 = 0;
        time = curTime;
    }
    if(time == -1)
        time = curTime;
    int sum = 0;
    Iterator<IntWritable> i = values.iterator();
    while(i.hasNext()) {
        sum += i.next().get();
    }
    if(stateCode.equals("200")) {
        sum1 += sum;
    }
    else if(stateCode.equals("400")) {
        sum2 += sum;
    }
    else if(stateCode.equals("404")) {
        sum3 += sum;
    }
    else {
        System.out.println("unknown state code "+stateCode);
    }
}
```

图 6: Reducer 设计

任务难点：单文件输出　我们使用 hadoop 文件系统的 javaAPI 来写入文件，因为考虑到任务 1 只需要输出单文件，而我们使用了 24 个 reducer 来分别统计时间窗内状态码的出现次数。但是如果我们通过不同的 reducer 来使用 MultipleOutputs 来写入同一个文件，会发生文件覆盖的现象（未找到解决方案），因此我们先等待 MapReduce 输出 24 个文件后，再对文件内容进行统计输出嗷 1.txt 中。

2.2 任务 2：IP 访问频次统计

任务分析　任务二的主要需求是统计每个 IP 地址在时间窗内访问的次数。所以在任务二中我们依然只需要关注两个数据，分别是时间部分的数据与 IP 地址部分的数据。实验二中输出我们需要关注多文件输出和文件名自定义。

Mapper 设计　本次任务只需要统计时间与 IP 地址，所以在任务二中的 Mapper 部分，我们只需要取出数据中的这两部分发送。在代码中，我们取出数据的 IP 与时间并且过滤掉一些无用 IP，进行拼接后构成一个复合 key 发送。在之后的 reducer 中，我们需要在一个 reducer 上处理同一个 IP 的数据，所以之后我们需要调用一个 Partitioner 来进行处理。

```
class Task2Mapper extends Mapper<Object, Text, Text, IntWritable> {
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //<key, value> = <ip#time, 1>
        String[] tokens = value.toString().split(" ");
        //tokens[0]是ip地址; tokens[1]包含时间,只需要h
        if(!tokens[0].equals("-")) { //可能要过滤掉更多情况
            String outKey = tokens[0] + "#" + tokens[1].split(":")[1];
            context.write(new Text(outKey), new IntWritable(1));
        }
    }
}
```

图 7: Mapper 设计

Combiner 设计　任务 1 中，我们需要对时间窗内的 IP 访问的次数做一个统计，与任务一一样，是一个计数任务，满足结合律，所以我们依然使用 combiner 来对数据进行本地 reduce 以此减少数据在网络中的传输，提高 MapReduce 框架的效率。

```
//本地reduce, 提升性能
class Task2Combiner extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        Iterator<IntWritable> i = values.iterator();
        int sum = 0;
        while(i.hasNext()) {
            sum += i.next().get();
        }
        context.write(key, new IntWritable(sum));
        //System.out.println(key.toString()+" : "+sum);
    }
}
```

图 8: Combiner 设计

Partitioner 设计　任务 2 中 Partitioner 的设计方面只需要考虑在 Mapper 处理好数据之后，如何分区选择不同的 Reducer 来进行处理，从而均匀的将 Mapper 的输出结果分布在 reducer 上执行。

本次任务中我们需要统计的是时间窗口内访问 IP 的次数，所以很显然同一个 IP 的数据应该放在相同的 reducer 节点中。

```java
//按照ip地址进行划分
class Task2Partitioner extends HashPartitioner<Text, IntWritable> {
    @Override
    public int getPartition(Text key, IntWritable value, int numReduceTasks) {
        String time = key.toString().split("#")[0];
        //System.out.println(key.toString());
        return super.getPartition(new Text(time), value, numReduceTasks);
    }
}
```

图 9: Partitioner 设计

Reducer 设计　Reducer 接受数据并对数据进行处理，首先将键值对分割，得到当前键值对的 IP 地址，从而对 IP 在各个时间段内出现的次数进行统计。

```java
protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
    String[] tokens = key.toString().split("#");
    String curIP = tokens[0];
    int time = Integer.parseInt(tokens[1]);
    if(!curIP.equals(ip) && !ip.equals("NO")) {
        //TODO: 写入txt文件
        int sum = 0;
        for(int i=0; i < 24; i++)
            sum += cnt[i];
        String outKey = String.format("%d", sum);
        outputs.write(new Text(outKey), NullWritable.get(), ip+".txt");
        for(int i=0; i<24;i++) {
            outKey = String.format("%d:00-%d:00 %d", i, (i+1)%24, cnt[i]);
            outputs.write(new Text(outKey), NullWritable.get(), ip+".txt");
        }
        ip = curIP;
        for(int i = 0; i < 24; i++)
            cnt[i]=0;
    }
    if(ip.equals("NO"))
        ip = curIP;
    int sum = 0;
    Iterator<IntWritable> i = values.iterator();
    while(i.hasNext()) {
        sum += i.next().get();
    }
    cnt[time] += sum;
}
```

图 10: Reducer 设计

难点：多文件输出与文件名自定义　利用 MultipleOutputs 函数完成多文件输出，但是在默认的多文件输出中会带有-0000 的后缀，为了自定义文件名，我们重写了 TextOutputformat 函数，具体而言，文件名由 getDefaultWorkFile 方法决定，它调用 getUniqueFile 拼接产生 part-r-0000 的字符串；getUniqueFile 的参数 getOutput-Name(context) 默认就是 part，我们改写 getFaultWorkFile，将对 getUniqueFile 的调用替换成 getOutputName(context) 即可输出中包含 part-r-0000 这样的空文件，通过 LazyOutputFormat 解决。

2.3 任务 3：接口访问频次统计

任务分析　统计每个 URL 的访问频次，并按照频次降序排列，解决思路即为用两个 Job 完成任务 3，第一个 Job 用于统计 URL 的访问频次，第二个 Job 使用全排序对第一个 Job 的结果进行排序。

Job 1 Mapper 设计　在 Mapper 方法中处理字符串，以 URL 为 key 值，由于是需要统计访问频次，所以以数值 1 为 value 发送数据。

```
class Task3StatisMapper extends Mapper<Object, Text, Text, IntWritable> {
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //<key, value> = <interface, 1>
        String[] tokens = value.toString().split(" ");
        //tokens[4]是接口
        String outKey = tokens[4];
        if(!outKey.equals("null"))
            context.write(new Text(outKey), new IntWritable(1));
    }
}
```

图 11: Mapper 设计

Job 1 Combiner 设计　Job1 实际上和任务 1,2 一样是一个计数问题，满足结合律，所以我们用 combiner 来优化 MapReduce 框架，在这里就不多加赘述。

```
class Task3StatisCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        Iterator<IntWritable> i = values.iterator();
        int sum = 0;
        while(i.hasNext()) {
            sum += i.next().get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

图 12: Combiner 设计

Job 1 Partitioner 设计　由于此次任务需要的是统计 url 的访问频次，所以我们以 url 来对数据进行划分。

```
//按照接口进行划分
class Task3StatisPartitioner extends HashPartitioner<Text, IntWritable> {
    @Override
    public int getPartition(Text key, IntWritable value, int numReduceTasks) {
        return super.getPartition(key, value, numReduceTasks);
    }
}
```

图 13: Partitioner 设计

Job 1 Reducer 设计　Reducer 接受的 Values 是一个迭代器，合并每个 URL 的访问次数，最后写出。

```java
class Task3StatisReducer extends Reducer<Text, IntWritable, Text, NullWritable> {
    private int sum;
    private String url;
    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        sum = 0;
        url = "NO";
    }
    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        String curUrl = key.toString();
        if(!curUrl.equals(url) && !url.equals("NO")) {
            //TODO: 写入txt文件
            String outKey = String.format("%s %d#0", url, sum);
            context.write(new Text(outKey), NullWritable.get());
            url = curUrl;
            sum = 0;
        }
        if(url.equals("NO"))
            url = curUrl;
        Iterator<IntWritable> i = values.iterator();
        while(i.hasNext()) {
            sum += i.next().get();
        }
    }

    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
        //super.cleanup(context);
        if(url.equals("NO"))
            return;
        //TODO: 写入txt文件
        String outKey = String.format("%s %d#0", url, sum);
        context.write(new Text(outKey), NullWritable.get());
    }
}
```

图 14: Reducer 设计

Job2 排序任务　全排序使用 TotalOrderPartitioner 类，使用 KeyValueTextInputFormat 读取上一步的结果，由于不存在，每一行都被识别为 key，value 为空。使用 KeyValueTextInputFormat 读取上一步的结果，由于不存在□每一行都被识别为 key，value 为空使用 RandomSampler 对输入内容的 key 进行采样，有 n 个 reducer，则采样 n-1 个 key 值，排好序后存放在 partition file 中。全排序根据 partition file 的内容决定每个键值对发给哪个 reducer 。每个 reducer 对接受的键值对按照 key 排序，要求降序排列，此时继承 WritableComparator 类重写 compare 方法实现降序排列

2.4　任务 4：接口平均响应时间统计

任务分析　任务描述任务 4 要求统计每个 URL 的平均响应时间和每个时间窗内的平均响应时间，将上述统计信息输出到以 URL 命名的 txt 文件中。

如果将平均响应时间改为响应时间之和，那么任务 4 将和任务 2 有相同的处理方式。为了获取平均响应时间，我们在统计总响应时间的同时，统计 URL 的访问次数，二者做除法就是平均响应时间。

Mapper 设计　本次任务与前三个任务有区别，非统计型任务，Map 的输出设计中，key 值设计为 url 和时间的一个拼接，value 值设计为响应时间和数值 1 的一个拼接。

Combiner 设计　Combiner 对 value 拆分，分别累加响应时间和访问次数。

```java
class Task4Mapper extends Mapper<Object, Text, Text, Text> {
    @Override
    protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        //<key, value> = <url#hour, response_time#count>
        String[] tokens = value.toString().split(" ");
        //tokens[4]是url; tokens[1]包含时间,只需要h; tokens[9]是响应时间
        if(!tokens.equals("null")) {
            String outKey = tokens[4] + "#" + tokens[1].split(":")[1];
            String outVal = tokens[9] + "#1";
            context.write(new Text(outKey), new Text(outVal));
        }
    }
}
```

图 15: Mapper 设计

```java
class Task4Combiner extends Reducer<Text, Text, Text, Text> {
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        int sumTime = 0;
        int sumCount = 0;
        for (Text t : values) {
            sumTime += Integer.parseInt(t.toString().split("#")[0]);
            sumCount += Integer.parseInt(t.toString().split("#")[1]);
        }
        context.write(key, new Text(sumTime + "#" + sumCount));
    }
}
```

图 16: Combiner 设计

Partitioner 设计　任务 1 中 Partitioner 的设计是按照 url 来进行划分。

```java
class Task4Partitioner extends HashPartitioner<Text, Text> {
    @Override
    public int getPartition(Text key, Text value, int numReduceTasks) {
        String url = key.toString().split("#")[0];
        return super.getPartition(new Text(url), value, numReduceTasks);
    }
}
```

图 17: Partitioner 设计

Reducer 设计　Reducer 接受的键值对按照 key 有序，设置两个长度为 24 的 int 数组用于存放同一个 url 不同时间窗的总响应时间和。

当读取到的 url 发生变化时，认为上一个 url 的信息已经统计完成，进行输出，输出时根据两个数组的内容进行一定的处理即可

2.5　任务 5：接口访问频次预测

基本思路　该任务已知每个 URL 每个时段前 14 天的访问频次数据, 要求预测第 15 天的访问频次数据, 那么同一 URL 同一时段前 14 天的数据必须在同一位置方可进行计算. 而对于不同的 URL 或者同一 URL 的不同时段, 我们认为它们的数据彼此之间没有关联, 则可以并行地进行处理. 得到所有的预测值后, 我们计算 RMSE 值作为预测结果的评价标准. 根据上述思路, 我们将整个大任务分为三个 mapreduce job 进行处理. 首先, 对数据进行预处理, 将同一 URL 同一时段的所有数据集中在一起; 然后, 使用预测算法进行预测得到预测值; 最后, 根据预测值和真实值计算得到

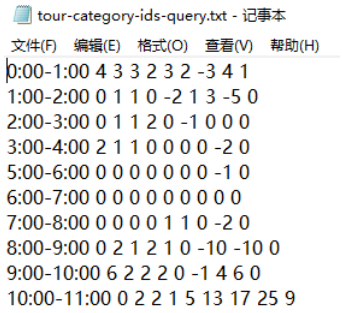
```java
protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    String[] tokens = key.toString().split("#");
    String curUrl = tokens[0];
    int hour = Integer.parseInt(tokens[1]);

    if(!curUrl.equals(url) && !url.equals("@")) {
        int sumTime = 0;
        int sumCount = 0;
        for (int t : resTimeArr) {
            sumTime += t;
        }
        for (int t : countArr) {
            sumCount += t;
        }
        String outKey = String.format("%.2f\n",((double)sumTime) / ((double)sumCount));
        for(int i = 0; i < 24; i++) {
            if (countArr[i] > 0) {// 每个时间窗内可能有未访问的情况
                outKey += String.format("%d:00-%d:00 %.2f\n", i, (i+1)%24, ((double)resTimeArr[i] / (double)countArr[i]));
            }
            else if (countArr[i] == 0) {
                outKey += String.format("%d:00-%d:00 null\n", i, (i+1)%24);
            }
        }
        // 假定出现的接口肯定被访问过一次
        multiOutputs.write(new Text(outKey), NullWritable.get(), url.replaceFirst("/", "").replace("/", "-") + ".txt");
        url = curUrl;
        Arrays.fill(resTimeArr, 0);
        Arrays.fill(countArr, 0);
    }
    // 根据时间窗进行统计
    if (url.equals("@"))
        url = curUrl;

    for (Text t : values) {
        resTimeArr[hour] += Integer.parseInt(t.toString().split("#")[0]);
        countArr[hour] += Integer.parseInt(t.toString().split("#")[1]);
    }
}
```

图 18: Reducer 设计

RMSE 值. 注意到 RMSE 的公式中, 各个小时时间窗的数据是互相独立的, 于是我们可以并行地计算各个时间窗内的 RMSE, 累加之后求平均得到最后的 RMSE. 实验中, 我们分别尝试了四种不同类型的预测算法, 分别是平均值法、改进的平均值法、多项式拟合和 ARIMA 模型, 下面将逐一进行介绍.

数据预处理　读取全部的日志文件, 对每个 URL 每个时段在每一天内的访问频次总数进行统计, 最后对于每个 URL 每个时段, 将属于它的所有的 15 天的数据组织在一起, 按天数从左向右写入同一行, 每行最后一个值即为要预测的真实值, 前 14 个值为前 14 天的数据. 输出文件里每一行的格式按如下方式组织: urlhour frequency1 frequency 2 ... frequency15

预测器　由于预处理过程输出的数据已经满足预测算法所需要的输入形式, 所以这里直接在 map 阶段就调用预测算法得到预测值, 由于有多个预测算法, 这里的预测值有多个结果, 于是 map 发射出的 kv 对满足如下形式: <urlhour, 实际值预测值 1 预测值 2 .....> 考虑到输出结果的可读性, 这里在 partition 阶段按 url 进行划分, 最后 reducer 输出的文件也按 url 进行组织, 每个文件里是该 url 所有时间窗的预测结果, 如下图所示:



朴素预测法　该算法的思想是非常朴素的, 即我们认为同一 URL 同一时段内的访问频次在统计学意义上应该是稳定的, 即在某个固定值范围内振荡, 于是这里简单地对前 14 天的数据求平均, 将平均值作为预测值即可. 然而在现实情况中, 某些时候会出现访问的峰值和低谷, 为了避免这些离群点的影响, 对于上述想法一个简单改进是将前 14 天的数据去除最大最小值再求平均.

多项式拟合　所谓拟合, 就是将平面上一系列的点, 使用一条光滑的曲线连接起来. 对于不同的对象, 有不同的拟合方法: 线性拟合、指数拟合、幂拟合、高斯拟合等. 对于本实验, 人们对 URL 的访问分为常规访问与突发访问. 常规访问如每天在固定时间访问一些网址; 突发访问如双十一活动时淘宝访问激增等. 考虑到我们预测的时间段及 URL 不覆盖节日、电商活动或者突发新闻, 故使用多项式函数进行预测.

首先, 我们将前 14 天的数据组织成 (x,y) 对的形式, 对于第 i 天的数据 $(\mathrm{x}_i, y_i), x_i = i, y_i.n, :$

使用该函数对前 14 天的数据点进行拟合, 即计算 $\mathrm{P}_n(x_1)\ P_n(x_14), y_1\ y_14., , P_n(x_15), .14, , , .$

即 XA=Y, 解得 $\mathrm{A}=(\mathrm{X}^T X)^{(}-1)X^T Y.$

ARIMA 模型　注意到上述的预测方法都没有体现出该任务中数据的特点, 经过分析我们可以知道, 这里的任务是一个典型的短期时间序列分析问题, 于是这里我们使用时间分析中最经典的模型, 差分整合移动平均自回归模型 (ARIMA) 对问题进行建模求解. ARIMA 模型将预测对象随时间推移而形成的数据序列当成一个随机序列, 进而用一定的数学模型来近似表述该序列. 该模型根据原序列是否平稳以及回归中所包含部分的不同分为 AR、MA、ARMA 以及 ARIMA 过程.

ARIMA模型有三个参数:p,d,q。

- p--代表预测模型中采用的时序数据本身的滞后数(lags) ,也叫做AR/Auto-Regressive项
- d--代表时序数据需要进行几阶差分化，才是稳定的，也叫Integrated项。
- q--代表预测模型中采用的预测误差的滞后数(lags)，也叫做MA/Moving Average项

AR:
当前值只是过去值的加权求和。

1、AR（p）(p 阶自回归模型）

$$x_t = \delta + \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + u_t$$

其中 u_t 白噪声序列，δ 是常数（表示序列数据没有 0 均值化）

MA:
过去的白噪音的移动平均。

2、MA（q）（q 阶移动平均模型）

$$x_t = \mu + u_t + \theta_1 u_{t-1} + \theta_2 u_{t-2} + \cdots + \theta_q u_{t-q}$$

$$x_t - \mu = (1 + \theta_1 L + \theta_2 L^2 + \cdots + \theta_q L^q) u_t = \Theta(L) u_t$$

其中 $\{u_t\}$ 是白噪声过程。

ARMA:
AR和MA的综合。

3、ARMA（p，q）（自回归移动平均过程）

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + \delta + u_t + \theta_1 u_{t-1} + \theta_2 u_{t-2} + \cdots + \theta_q u_{t-q}$$

$$\Phi(L) x_t = (1 - \phi_1 L - \phi_2 L^2 - \cdots - \phi_p L^p) x_t$$

$$= \delta + (1 + \theta_1 L + \theta_2 L^2 + \cdots + \theta_q L^q) u_t = \delta + \Theta(L) u_t$$

$$\Phi(L) x_t = \delta + \Theta(L) u_t$$

ARIMA:
和ARMA的区别，就是公式左边的x变成差分算子，保证数据的稳定性。

$$\Phi(L) \Delta^d x_t = \delta + \Theta(L) u_t$$

于是 ARIMA 模型可以理解为

ARIMA的预测模型可以表示为：

> Y的预测值 = 常量c and/or 一个或多个最近时间的Y的加权和 and/or 一个或多个最近时间的预测误差。

假设p，q，d已知，
ARIMA用数学形式表示为：

$$\widehat{y_t} = \mu + \phi_1 * y_{t-1} + \ldots + \phi_p * y_{t-p} + \theta_1 * e_{t-1} + \ldots + \theta_q * e_{t-q}$$

其中，ϕ表示AR的系数，θ表示MA的系数

该模型的建立包含超参数 p,d,q 的确定和模型参数估计两个过程. 通常而言

ARIMA模型运用的基本流程有几下几步：

1. 数据可视化，识别平稳性。
2. 对非平稳的时间序列数据，做差分，得到平稳序列。
3. 建立合适的模型。
   平稳化处理后，若偏自相关函数是截尾的，而自相关函数是拖尾的，则建立AR模型；
   若偏自相关函数是拖尾的，而自相关函数是截尾的，则建立MA模型；
   若偏自相关函数和自相关函数均是拖尾的，则序列适合ARMA模型。
4. 模型的阶数在确定之后，对ARMA模型进行参数估计，比较常用是最小二乘法进行参数估计。
5. 假设检验，判断（诊断）残差序列是否为白噪声序列。
6. 利用已通过检验的模型进行预测。

但由于该应用场景下, 我们无法对所有组数据都进行可视化挑选出最适合的差分阶数, 于是这里采用实践中常用一阶差分. 同时, 超参数 p 和 q 的选择采用遍历法, 遍历从 (1,1) 到 (2,2) 共四组值, 采用其中 AIC 值最小的作为模型的超参数. 在确定模型的超参数后, 这里利用 Yule-Walker 方法对参数进行求解, 同时为了避免在求解过程中进行逆矩阵的计算, 这里采用 Levinson 递推公式求解 Y-W 方程, 得到模型的参数.

RMSE 计算　RMSE 的计算公式为:

$$RMSE = \frac{1}{M}\sum_{i=1}^{M} \sqrt[2]{\frac{\sum_{j=1}^{N}(C1j - C2j)^2}{N}}$$

注意到任务中给出的 RMSE 公式对于各个时间窗来说是独立的, 于是这里可以并行地计算各个时间窗的 RMSE, 累加之后求平均得到最后的 RMSE. 具体而言, 这里使用 24 个 reducer 将各个小时时间窗内的 RMSE 写到 24 个文件中, 最后使用 hdfs 操作文件进行汇总, 得到最后的 RMSE.

3 Spark 重写任务 1-4

使用 Spark 的 JavaAPI 实现任务 1 到任务 4

3.1 任务 1

首先对所有的状态码进行计数。mapToPair 方法执行 map 操作，针对原始数据的每一行产生一个二元组，此二元组可以视为键值对。reduceByKey 方法执行 reduce 操作，将相同 key 值的元组进行合并。两个方法的参数都是匿名函数，规定了如何进行 map 和 reduce 操作。在最终的结果 stateCode$_sum < String, Integer >$

之后按时间窗统计状态码。这一步首先执行一次 mapToPair 方法和 reduceByKey 方法，目的与上一步相同，区别在于此处 key 值是状态码和时间组成的二元组。执行完一次 map、reduce 操作后，将数据重新 map，保证 key 值是时间，再调用 groupByKey、sortByKey 操作，将同一时间窗内的各个状态码的计数构成一个迭代器，并按照时间大小进行排序。

```
//分时间窗统计状态码，按照时间窗划分，再按时间排序
JavaPairRDD<Integer, Iterable<Tuple2<String, Integer>>> stateCode_hourSum = lines.mapToPair(
        line -> {
            String[] tokens = line.split( s: " ");
            int hour = Integer.parseInt(tokens[1].split( s: ":")[1]);
            return new Tuple2<Tuple2<Integer, String>, Integer>(new Tuple2<>(hour, tokens[7]), 1);
        }
).reduceByKey(
        (v1,v2) -> v1+v2
).mapToPair(
        kv -> new Tuple2<Integer, Tuple2<String, Integer>>(kv._1._1, new Tuple2<>(kv._1._2, kv._2))
).groupByKey().sortByKey();
```

经过上面两个步骤，我们已经获取到任务所需的数据，最后只需要把数据写入文件即可。考虑到任务的数据量并不大，最后写入的文件最多只有 27 行，所以此处采用 JavaAPI 对 Hadoop 的文件系统进行操作。直接在输出目录中新建 1.txt 文件，之后遍历数据，将数据写入文件。

3.2 任务 2

此任务的整体思路与任务 1 大致相同。在任务 2 中，我们仍需要获取两种数据，每个 IP 总的访问频次和每个 IP 在时间窗内的访问频次，我们分别统计这两种信息。

```
//检查当前路径是否存在，若存在则需要删除
if(fs.exists(new Path(outputPath)))
    fs.delete(new Path(outputPath), b: true);
FSDataOutputStream outputStream = fs.create(new Path( pathString: outputPath+"/1.txt"));
//写入总的统计信息
int sum1 = 0, sum2 = 0, sum3 = 0;
for(Tuple2<String, Integer> i : out1) {
    String code = i._1;
    if(code.equals("200"))
        sum1 += i._2;
    else if(code.equals("400"))
        sum2 += i._2;
    else
        sum3 += i._2;
}
String str = String.format( s: "200:%d\n400:%d\n404:%d\n",sum1, sum2, sum3);
outputStream.write(str.getBytes());
```

```
//写入按时间划分的统计信息
for(Tuple2<Integer, Iterable<Tuple2<String, Integer>>> i : out2) {
    int hour = i._1;
    sum1 = 0; sum2 = 0; sum3 = 0;
    for(Tuple2<String, Integer> j : i._2) {
        if(j._1.equals("200"))
            sum1 += j._2;
        else if(j._1.equals("400"))
            sum2 += j._2;
        else
            sum3 += j._2;
    }
    String outKey = String.format( s: "%d:00-%d:00 200:%d 400:%d 404:%d\n", ...objects:
    outputStream.write(outKey.getBytes());
}
outputStream.close();
```

```
//统计总的ip频次
JavaPairRDD<String, String> ip_cnt = ip_hourCnt.mapToPair(
        kv -> new Tuple2<String, String>(kv._1, kv._2.split(" ")[1])
).reduceByKey(
        (v1, v2) -> ((Integer)(Integer.parseInt(v1) + Integer.parseInt(v2))).to
);
```

```
//按时间窗统计ip频次
JavaPairRDD<String, String> ip_hourCnt = lines.mapToPair(
        line -> {
            String[] tokens = line.split(" ");
            return new Tuple2<Tuple2<String, Integer>, Integer>(new Tuple2<>(to
            parseInt(tokens[1].split(":")[1])), 1);
            // <<ip, hour>, 1>
        }
).reduceByKey(
        (v1,v2) -> v1+v2
).mapToPair(
        kv -> {
            int hour = kv._1._2;
            return new Tuple2<String, String>(kv._1._1, String.format("%d:00-%d
            24, kv._2.intValue()));
            // <ip, "5:00-6:00 123">
        }
);
```

任务 2 要求自定义文件名和多文件输出，为此，我们首先将上述两种数据进行合并，并按照自定义的顺序进行排序。parallelize 方法构造的 RDD 数据集，每个数据试一个二元组，并不是键值对。排序通过 sortBy 方法实现，排序策略是按照 IP 进行排序，IP 相同根据时间窗排序，若不存在时间窗，则排在相同 IP 的最前面。最后通过 mapToPair 方法构造成键值对，便于最后的输出。

```
JavaPairRDD<String, String> res = jsc.parallelize(
        ip_hourCnt.union(ip_cnt).collect()
).sortBy(
        data -> {
            String[] tokens = data._2.split(":");
            if(tokens.length == 1)
                return data._1;
            else
                return data._1+String.valueOf((char)(Integer.parseInt(tokens[0]
        }, true, 1
).mapToPair( //上述数据不是键值对，而是二元组
        data -> new Tuple2<String, String>(data._1,data._2)
);
```

多文件输出要求我们重写 MultipleTextOutputFormat 类，我们重写 generateFileNameForKeyValue 方法和 generateActualKey 方法。前一个方法可以根据 key 获取待写入的文件，此处则是 ip.txt。第二个方法可以在写入时将 key 值设置为 null，保证 IP 不会被写入文件。

```
public static class RDDMultipleTextOutputFormat extends MultipleTextOutputForma
    public String generateFileNameForKeyValue(String key, String value, String
        return key+".txt";
    }
    public String generateActualKey(String key, String value) {
        return null;
    }
}
```

```
Configuration conf = new Configuration();
try {
    Path outputPath = new Path(output);
    FileSystem hdfs = FileSystem.get(conf);
    //删除已有输出文件
    if (hdfs.exists(outputPath)) {
        hdfs.delete(outputPath, true);
    }
} catch (Exception e) {
    System.out.println("task2 FS error");
}
res.saveAsHadoopFile(output, String.class, String.class, RDDMultipleTextOutputF
```

3.3 任务 3

此任务统计接口的访问频次，并降序排列。在之前使用 mapreduce 框架完成的版本的集群测试中，我们可以知道接口数量并不多，在我们 reduce 之后，完全可以只使用一个 reducer 来完成排序。spark 版本的思路也是如此。通过 mapToPair 和 reduceByKey 方法统计每个接口的访问频次；之后将键值对用 map 方法修改为二元组，调用 sortBy 方法，并指定降序排序；最后使用 mapToPair 方法将数据变成键值对，方便最后的输出。

3.4 任务 4

任务 4 与任务 2 的整体流程相同，区别在于任务 4 获取的数据是平均数，无法直接从原始数据中提取。因此，我们首先统计每个 url 在每个时间窗内的总访问次数和总响应时间。之后的平时响应时间是基于这个数据产生的。

基于上一步产生的数据，我们通过 mapToPair 方法，将 url 作为 key，value 保持不变，之后调用 reduceByKey 方法，统计得到每个 url 总的响应时间和访问次数，二者做除法就是平均响应时间。

同样的，基于第一步产生的数据，调用 mapToPair 方法，对每个键值对修改 value，将其改成每个时间窗内的平均响应时间。

```
public static void task3(JavaRDD<String> lines, String output, JavaSparkContext
    JavaPairRDD<String, Integer> url_cnt = lines.mapToPair(
            line -> new Tuple2<String, Integer>(line.split(" ")[4], 1)
    ).reduceByKey(
            (v1, v2) -> v1 + v2
    ).map( // 键值对不能使用sortBy方法
            kv -> new Tuple2<String, Integer>(kv._1, kv._2)
    ).sortBy(
            tuple -> tuple._2, false, 1
    ).mapToPair(
            tuple -> new Tuple2(tuple._1, tuple._2)
    );
    Configuration conf = new Configuration();
    try {
        Path outputPath = new Path(output);
        FileSystem hdfs = FileSystem.get(conf);
        //删除已有输出文件
        if (hdfs.exists(outputPath)) {
            hdfs.delete(outputPath, true);
        }
    } catch (Exception e) {
        System.out.println("task3 FS error");
    }
    url_cnt.saveAsHadoopFile(output, String.class, Integer.class, TextOutputFor
}
```

```
JavaPairRDD<Tuple2<String, Integer>, Tuple2<Integer, Integer>> url_hour_resTim
        line -> {
            String[] tokens = line.split(" ");
            String url = tokens[4].replaceAll("/", "-").replaceFirst("-", "");
            Integer hour = Integer.parseInt(tokens[1].split(":")[1]);
            Integer time = Integer.parseInt(tokens[9]);
            return new Tuple2<Tuple2<String, Integer>, Tuple2<Integer, Integer
            Integer>(url, hour), new Tuple2<Integer, Integer>(time, 1));
        }
).reduceByKey(
        (v1, v2) -> new Tuple2<Integer, Integer>(v1._1+v2._1, v1._2+v2._2)
);
```

```
JavaPairRDD<String, String> url_resTime = url_hour_resTime.mapToPair(
        kv -> new Tuple2<String, Tuple2<Integer, Integer>>(kv._1._1, kv._2)
).reduceByKey(
        (v1, v2) -> new Tuple2<Integer, Integer>(v1._1+v2._1, v1._2+v2._2)
).mapToPair(
        kv -> {
            int x = kv._2._1, y = kv._2._2;
            String resTime = String.format("%.2f", (double)x / (double)y);
            return new Tuple2<String, String>(kv._1, resTime);
        }
);
```

```
JavaPairRDD<String, String> url_hourResTime = url_hour_resTime.mapToPair(
        kv -> {
            String url = kv._1._1;
            Integer hour = kv._1._2;
            int x = kv._2._1, y = kv._2._2;
            double resTime = (double)x / (double)y;
            String hourResTime = String.format("%d:00-%d:00 %.2f", hour, (hour-
            return new Tuple2<String, String>(url, hourResTime);
        }
);
```

最后，我们将上述两部分数据合并、排序后进行输出。过程不再赘述。

```
JavaPairRDD<String, String> finalRes = jsc.parallelize(
        url_resTime.union(url_hourResTime).collect()
).sortBy(
        kv -> {
            String[] tokens = kv._2.split(":");
            if(tokens.length == 1)
                return kv._1;
            else
                return kv._1+String.valueOf((char)(Integer.parseInt(tokens[0])
        }, true, 1
).mapToPair(
        x -> new Tuple2<String, String>(x._1,x._2)
);

Configuration conf = new Configuration();
try {
    Path outputPath = new Path(output);
    FileSystem hdfs = FileSystem.get(conf);
    //删除已有输出文件
    if (hdfs.exists(outputPath)) {
        hdfs.delete(outputPath, true);
    }
} catch (Exception e) {
    System.out.println("task4 FS error");
}
finalRes.saveAsHadoopFile(output, String.class, String.class, RDDOutputFormat.
```

4　结果展示

5　优化与分析

图 19: 任务 1：状态码频次统计

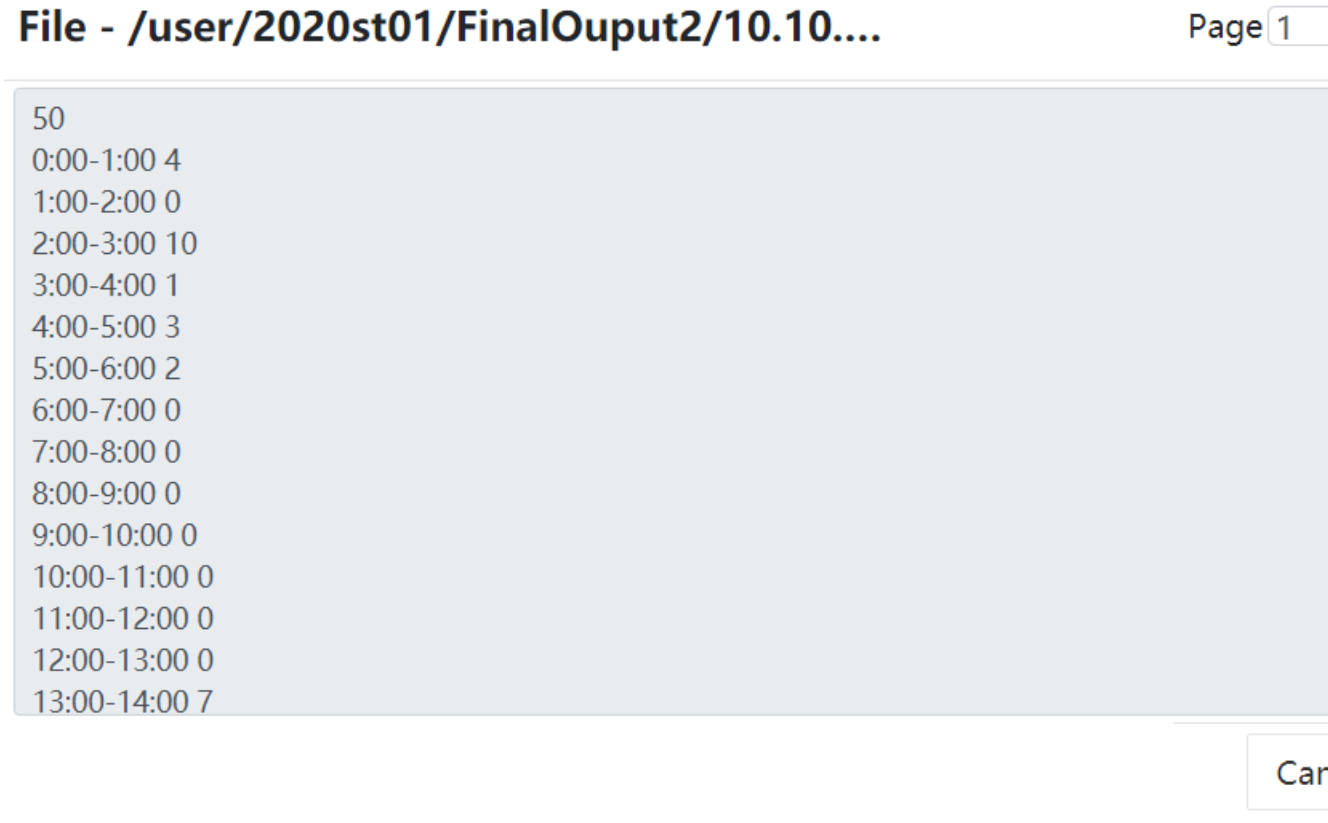


图 20: 任务 2：IP 访问频次统计

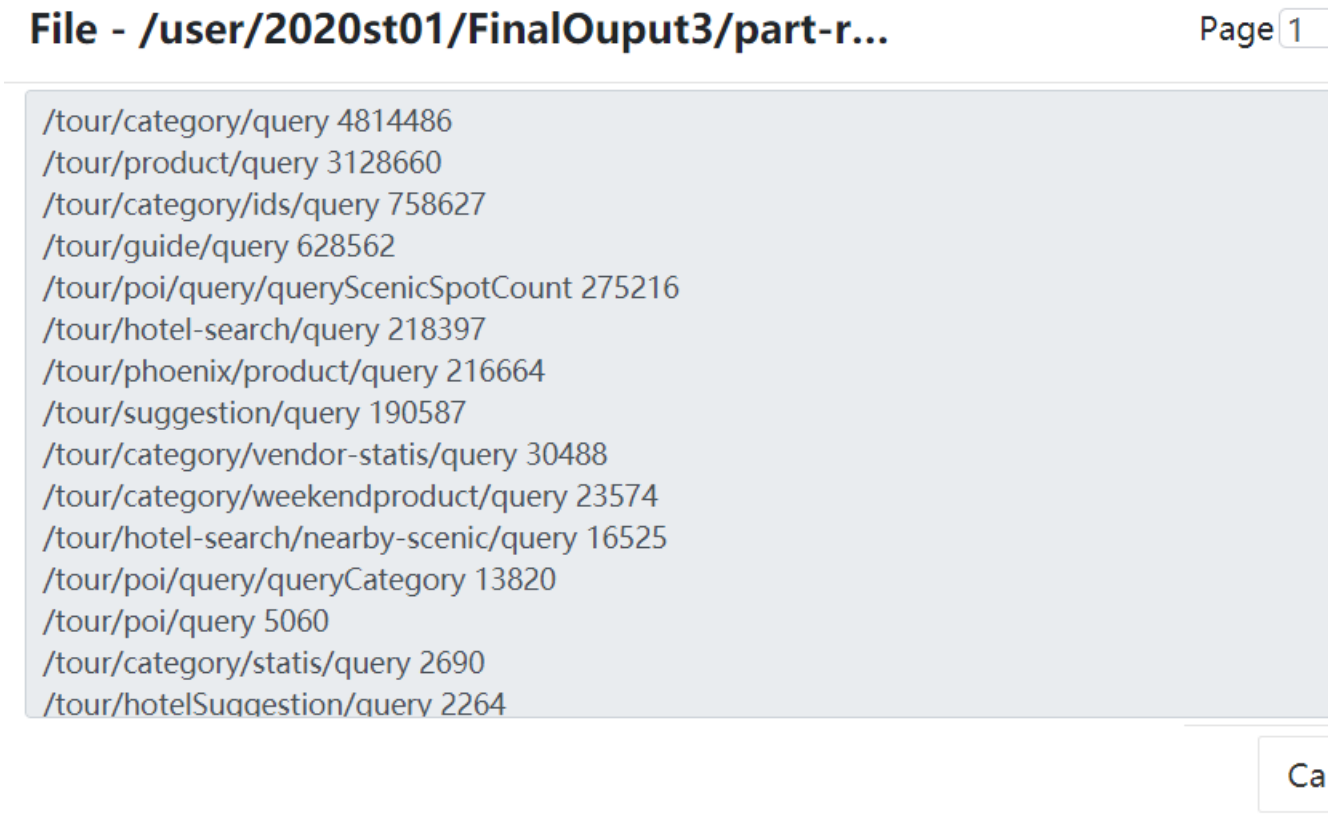


图 21: 任务 3：接口访问频次统计

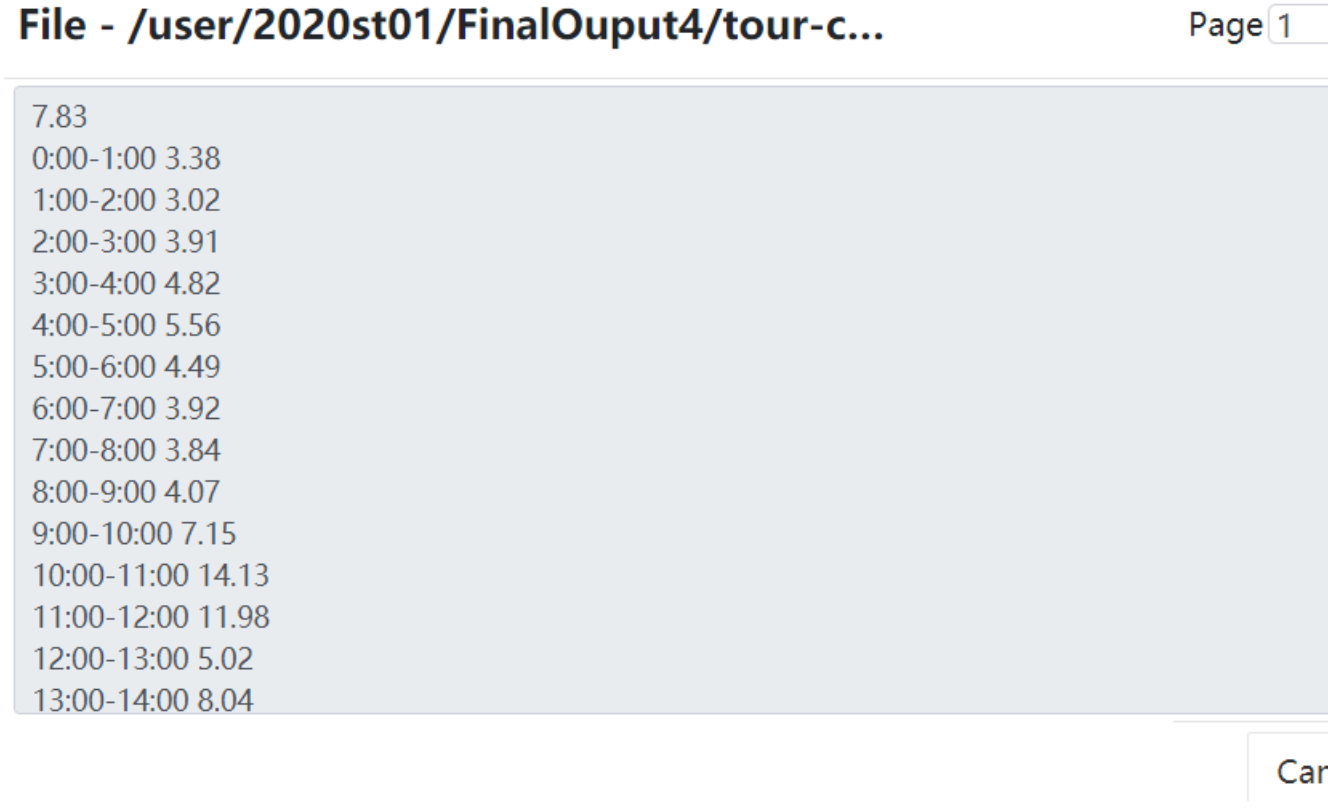


图 22: 任务 4：接口平均响应时间统计

参考文献

[1] 参考文献条目一.

致　谢

致谢内容

A 附录一

A.1 附录 1.1

附录内容