实验4 图的三角形计数

# 实验设计说明

## 主要设计思路

本次实验分为两个MapReduce Job完成。其中AdjList的输入为有向边的列表，输出为图的邻接矩阵；TriangleCount的输入为图的邻接矩阵，输出为图的三角形个数。

## AdjList

### Mapper

AdjListMapper将输入的Text拆分为节点对<v1, v2>，并发射<v1, v2>和<v2, v1>两个键值对。针对自指的边，Mapper通过判断v1与v2是否相等做了过滤。

|  |
| --- |
| **public** **static** **class** **AdjListMapper**  **extends** Mapper<Object, Text, Text, Text> {  **private** Text[] vertexPair = **new** Text[2];  **protected** void map(Object key, Text value, Context context) {  String[] valStrPair = value.toString().split(" ");  vertexPair[0].set(valStrPair[0]);  vertexPair[1].set(valStrPair[1]);  **if** (!vertexPair[0].equals(vertexPair[1])) {  context.write(vertexPair[0], vertexPair[1]);  context.write(vertexPair[1], vertexPair[0]);  }  }  } |

### UndirectedReducer

UndirectedReducer通过vertexSet对values进行去重，其输出为邻接表中的一行。由于版面限制，代码作了精简，省略了边界条件的判断。

|  |
| --- |
| **public** **static** **class** **AdjListUndirectedReducer**  **extends** Reducer<Text, Text, Text, Text> {  **private** Set<String> vertexSet = **new** HashSet<>();  **protected** void reduce(Text key, Iterable<Text> values, Context context) {  **for** (Text vertex : values)  vertexSet.add(vertex.toString());  StringBuilder builder = **new** StringBuilder();  **for** (String vertex : vertexSet) {  builder.append(vertex);  builder.append(' ');  }  context.write(key, **new** Text(builder.toString()));  vertexSet.clear();  }  } |

### DirectedReducer

选做部分中对有向边到无向边的转换有新的定义，当且仅当图中存在<v1, v2>和<v2, v1>这两条边时才认为v1与v2是相连接的。通过使用DirectedReducer就可以做到这点。

我们利用vertexSet判断当前的value是否在之前出现过，由于Mapper对每条有向边都发射了两组相反的键值对，因此values中出现次数大于1的那些节点才与key在两个方向上都有边。因此只有当set中已经存在该value时，才认为当前的value与key相互连接，并将value写入邻接表。由于图中不存在多重边，因此能够保证不会重复计数。由于版面限制，代码作了精简，省略了边界条件的判断。

|  |
| --- |
| **public** **static** **class** **AdjListDirectedReducer**  **extends** Reducer<Text, Text, Text, Text> {  **private** Set<String> vertexSet = **new** HashSet<>();  **protected** void reduce(Text key, Iterable<Text> values, Context context) {  StringBuilder builder = **new** StringBuilder();  **for** (Text vertex : values) {  **if** (!vertexSet.contains(vertex.toString()))  vertexSet.add(vertex.toString());  **else** {  builder.append(vertex);  builder.append(' ');  }  }  context.write(key, **new** Text(builder.toString()));  vertexSet.clear();  }  } |

## TriangleCount

### Mapper

Mapper的输入为图的邻接表中的一行。输出为<<v1, v2>, v3>形式的键值对。Mapper每次将key与values中的一个节点vertex [i]绑定为vertexPair作为发射的key，并将剩余的所有节点vertex [j]作为发射的value。为了保证<v1, v2>与<v2, v1>能够落到一个reducer上，我们对vertexPair定义了严格偏序，即v1<v2，即要根据key与vertex[i]的大小决定vertexPair的顺序。同时为了减少不必要的中间结果的数量，我们仅发射剩余节点中vertex[j]>v2的节点，即发射的entry满足v1<v2<v3。

|  |
| --- |
| **public** **static** **class** **TriangleCountMapper**  **extends** Mapper<Text, Text, Text, Text> {  Text vertexPair = **new** Text();  **protected** void map(Text key, Text value, Context context) {  String keyVertex = key.toString();  String[] vertexList = value.toString().split(" ");    Arrays.sort(vertexList);  int pivot = ~Arrays.binarySearch(vertexList, keyVertex);  **for** (int i = 0; i < vertexList.length; ++i) {  **if** (i < pivot)  vertexPair.set(keyVertex + "#" + vertexList[i]);  **else**  vertexPair.set(vertexList[i] + "#" + keyVertex);  int k = Math.max(pivot, i + 1);  **for** (int j = vertexList.length - 1; j >= k; --j)  *// emit <<v1,v2>,v3>*  *// v1 < v2 < v3*  context.write(vertexPair, **new** Text(vertexList[j]));  }  }  } |

### Reducer

Reducer需要判断节点对<v1, v2>的邻居节点集合中是否有交集，交集中元素的个数即为以<v1, v2>作为定点的三角形的个数。我们同样使用vertexSet来判断当前的value是否在之前出现过，如果vertexSet中存在当前value，即说明value是交集中的一个元素，令count++。在执行cleanUp时再将统计的count数量写入文件中。

|  |
| --- |
| **public** **static** **class** **TriangleCountReducer**  **extends** Reducer<Text, Text, NullWritable, LongWritable> {  **private** Set<String> vertexSet = **new** HashSet<>();  **private** long count = 0;  **protected** void reduce(Text key, Iterable<Text> values, Context context) {  **for** (Text vertex : values) {  **if** (!vertexSet.contains(vertex.toString()))  vertexSet.add(vertex.toString());  **else**  ++count;  }  vertexSet.clear();  }  **protected** void cleanup(Context context) {  context.write(NullWritable.get(), **new** LongWritable(count));  **super**.cleanup(context);  }  } |

# 实验结果

表格 1 实验结果

|  |  |  |
| --- | --- | --- |
| 数据集 | 三角形个数 | Driver程序在集群上的运行时间（秒） |
| Twitter | 13082506 | 337 |
| Google+ | 1073677742 | 17496 |

表格 2 选做1的实验结果

|  |  |  |
| --- | --- | --- |
| 数据集 | 三角形个数 | Driver程序在集群上的运行时间（秒） |
| Twitter | 1818304 | 151 |
| Google+ | 27018510 | 1017 |

# 输出结果

为了加快Reducer的执行速度，我们在job中设置了多个Reducer，因此需要对分片的结果进行累加，才能得到最终结果。在这里我们使用了shell脚本执行该操作，在输出结果目录中执行以下命令：

|  |
| --- |
| cat part-r-\* | python -c"import sys; print(sum(map(int, sys.stdin)))" |

即可将不同Reducer上的结果进行累加。

必做部分的输出结果位于lab5/undirected/twitter-output和lab5/undirected/google-output中；选做部分的输出结果位于lab5/directed/twitter-output和lab5/directed/google-output中：

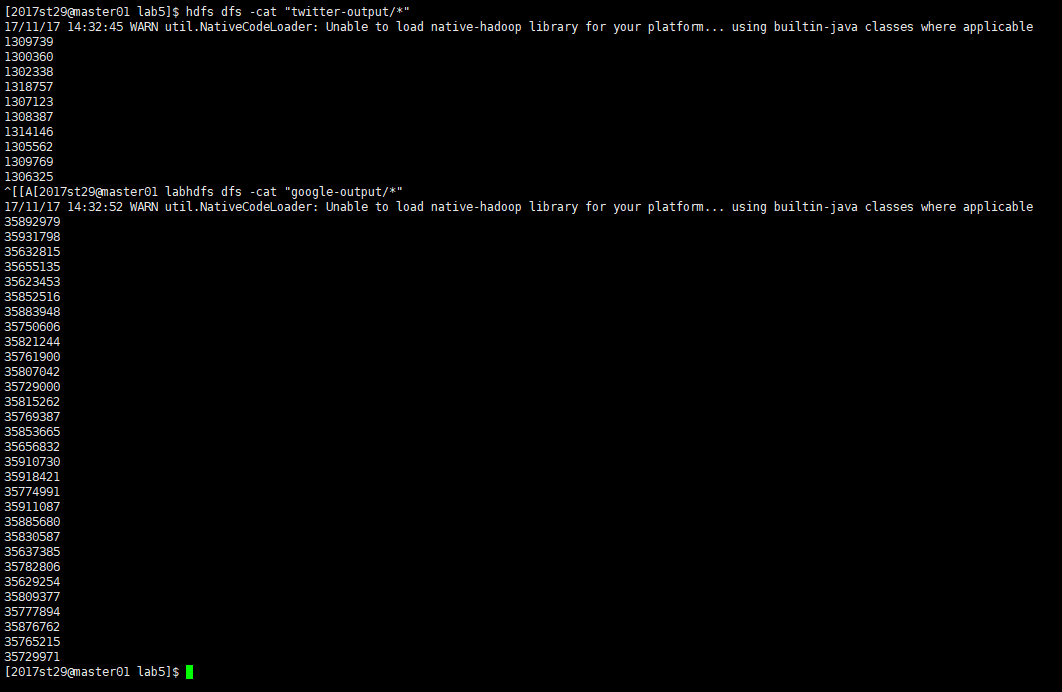


图 1 必做部分输出结果

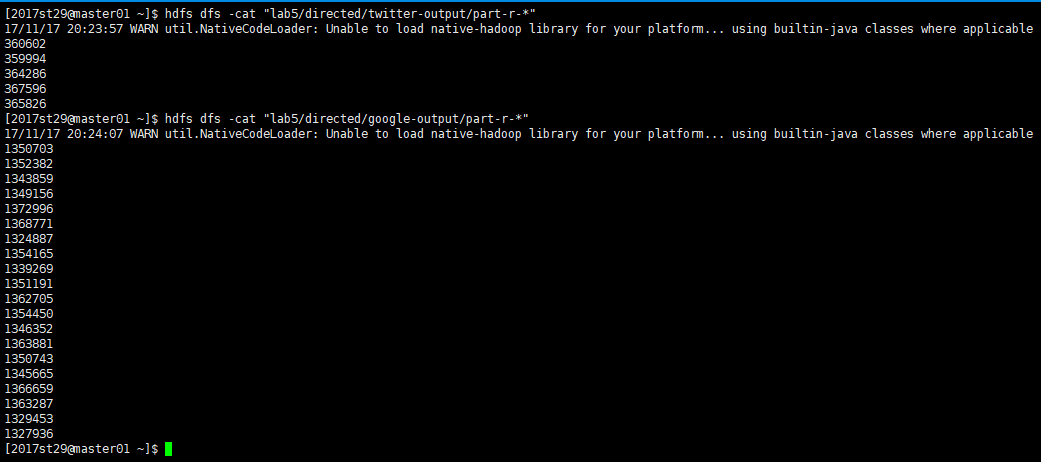


图 2 选做部分输出结果

# 执行报告

必做部分中，Twitter数据集的两个job分别为application\_1508726229114\_1135和application\_1508726229114\_1137；Google数据集的两个job分别为application\_1508726229114\_1140和application\_1508726229114\_1178。

选做部分中，Twitter数据集的两个job分别为application\_1508726229114\_1490和application\_1508726229114\_1493；Google数据集的两个job分别为application\_1508726229114\_1494和application\_1508726229114\_1498。

以下仅粘贴必做部分的截图。

## Twitter

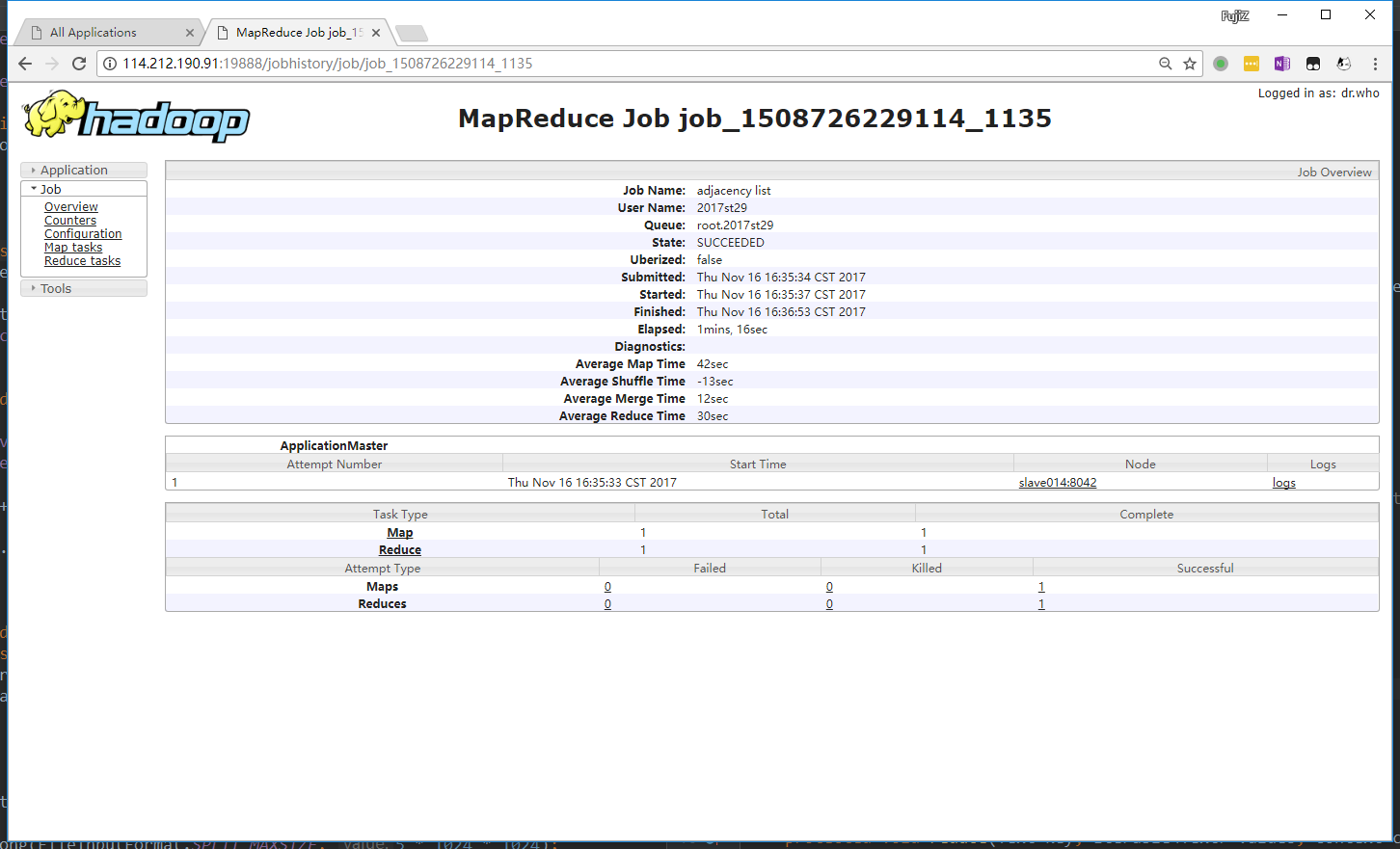


图 3 Twitter邻接表Job



图 4 Twitter邻接表Counters

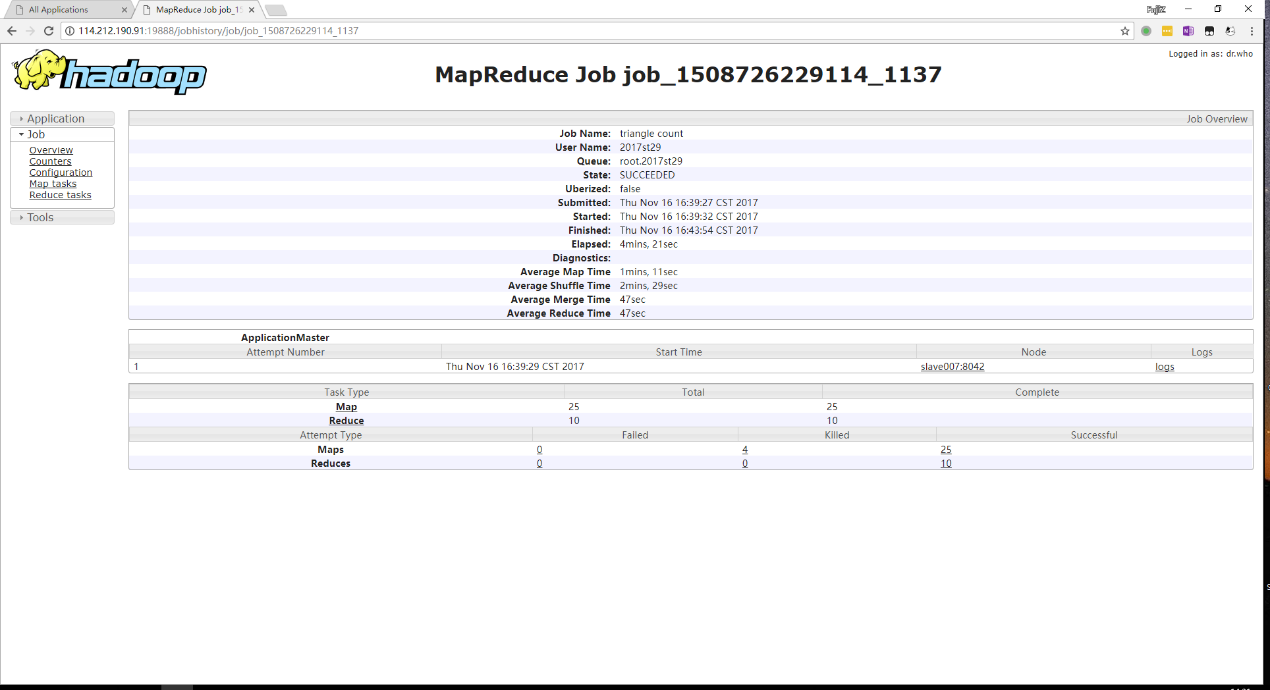


图 5 Twitter三角形计数Job

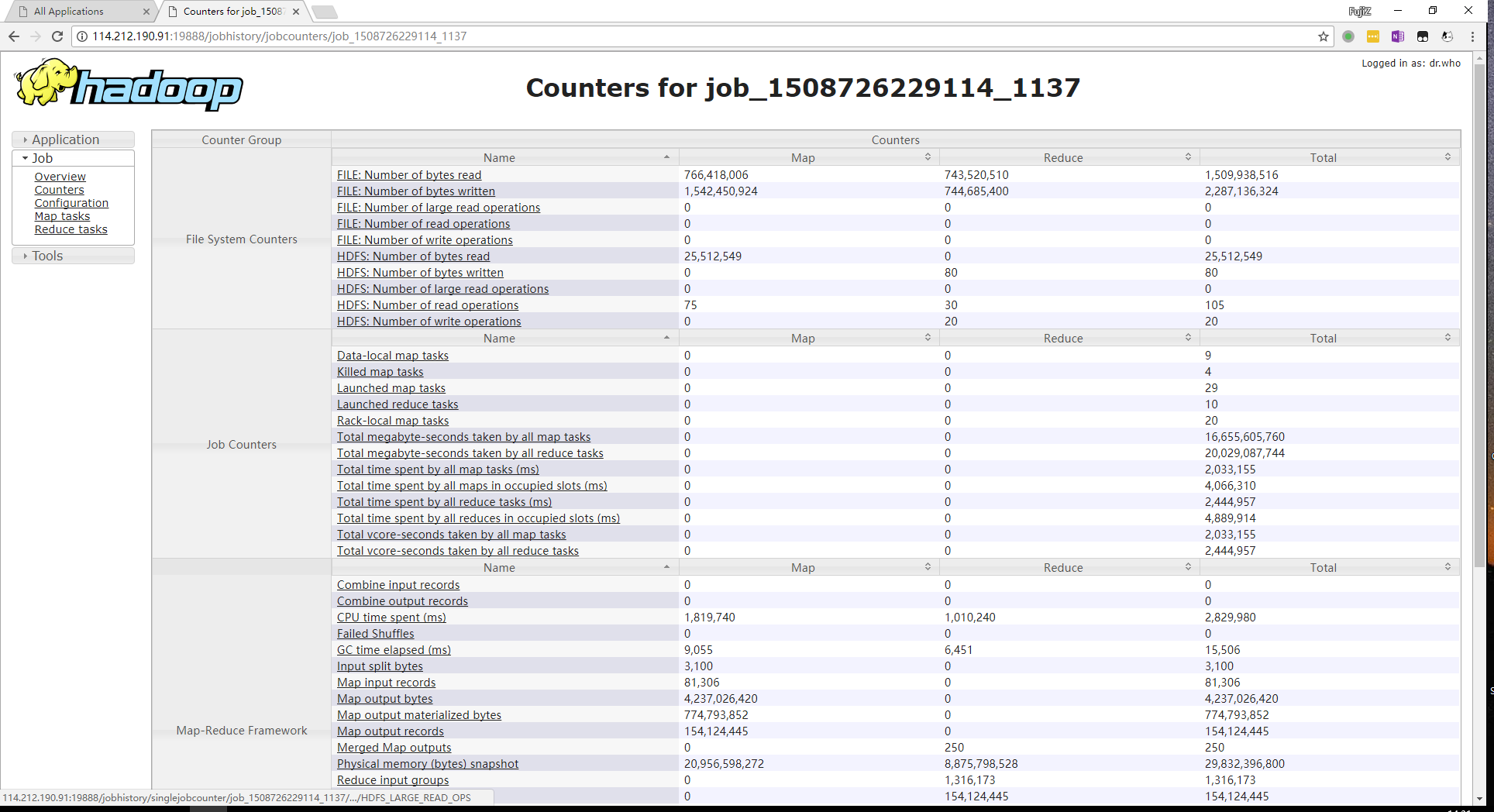


图 6 Twitter三角形计数Counters

## Google

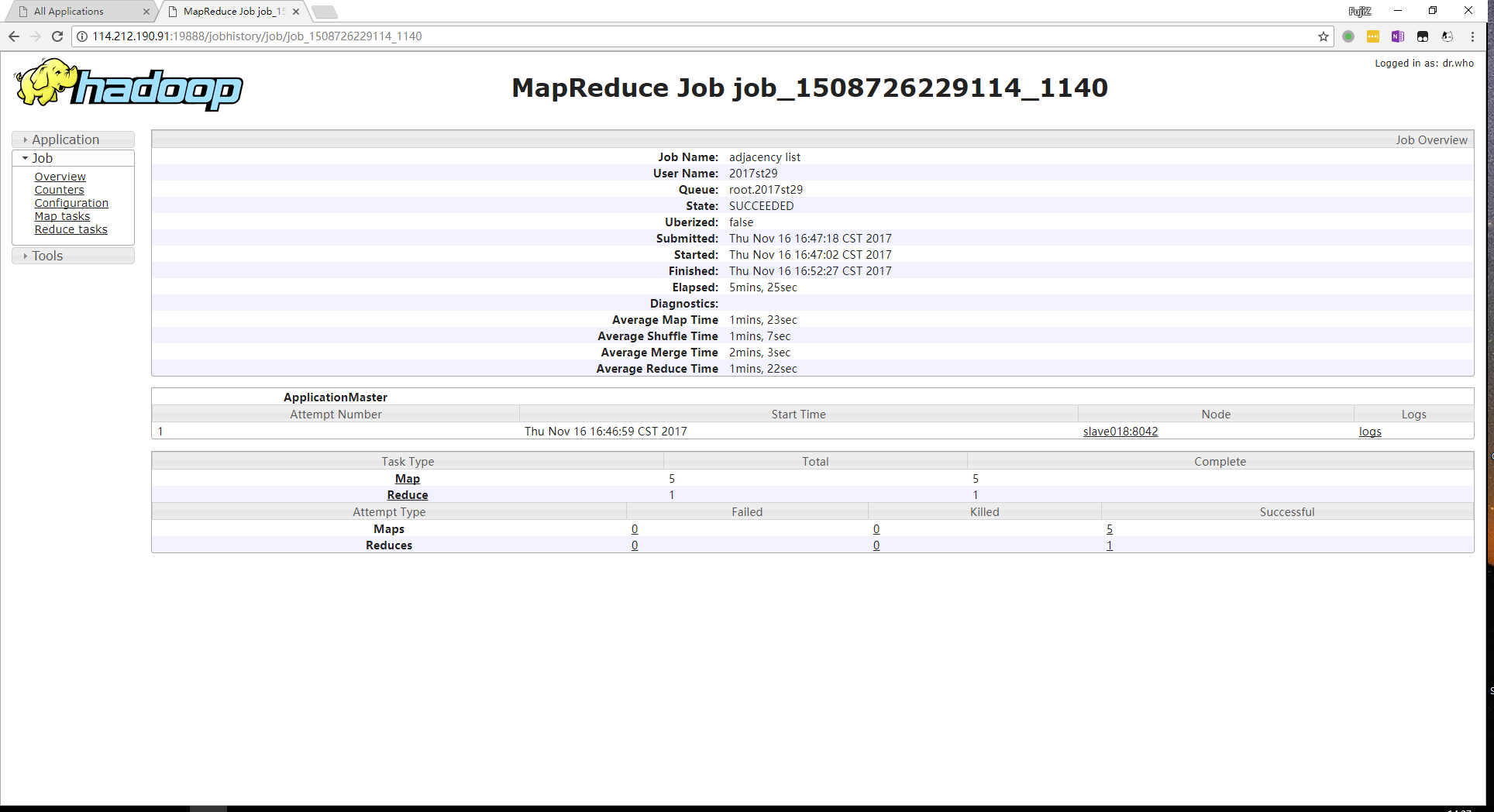


图 7 Google邻接表Job

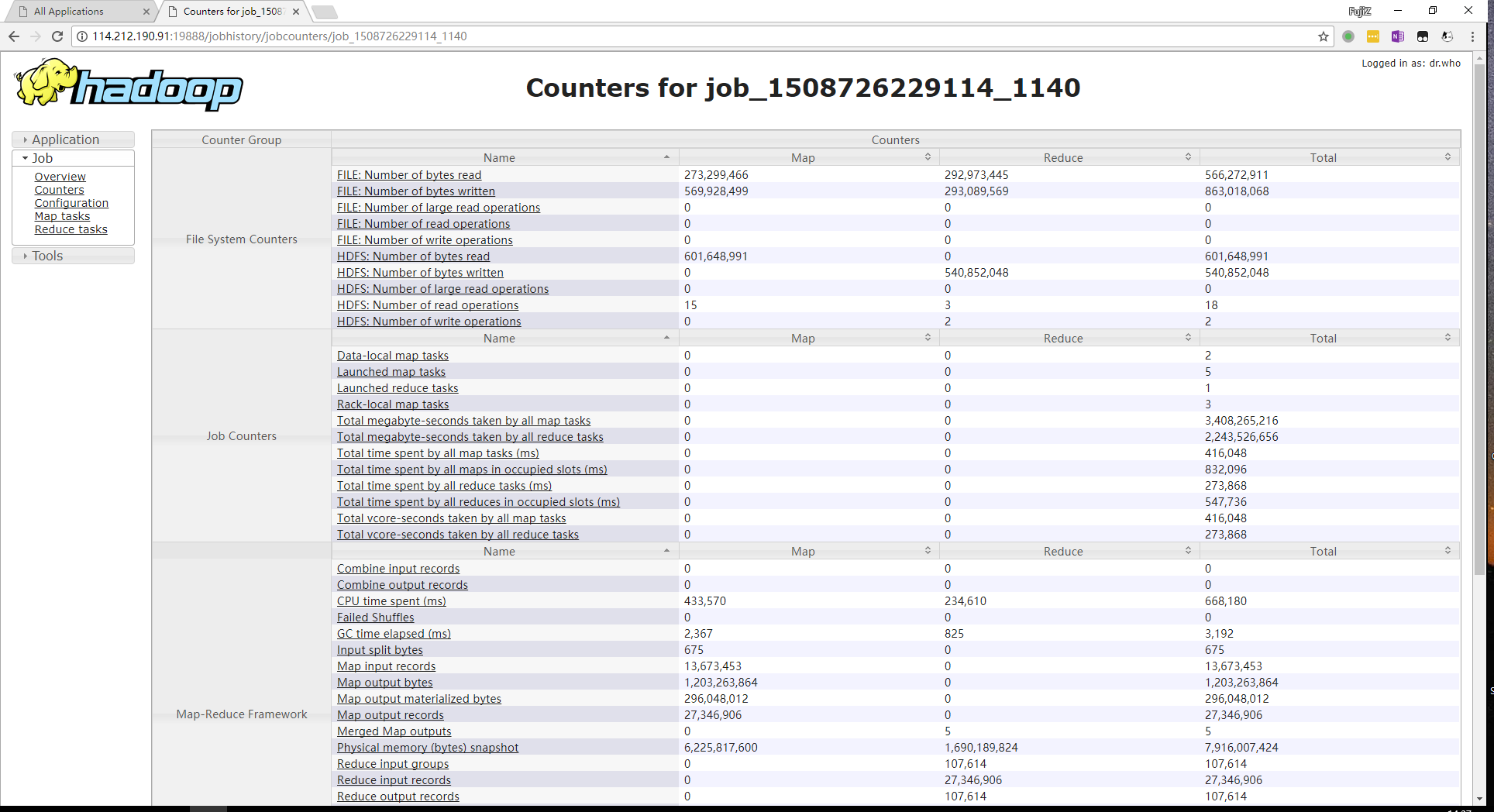


图 8 Google邻接表Counters

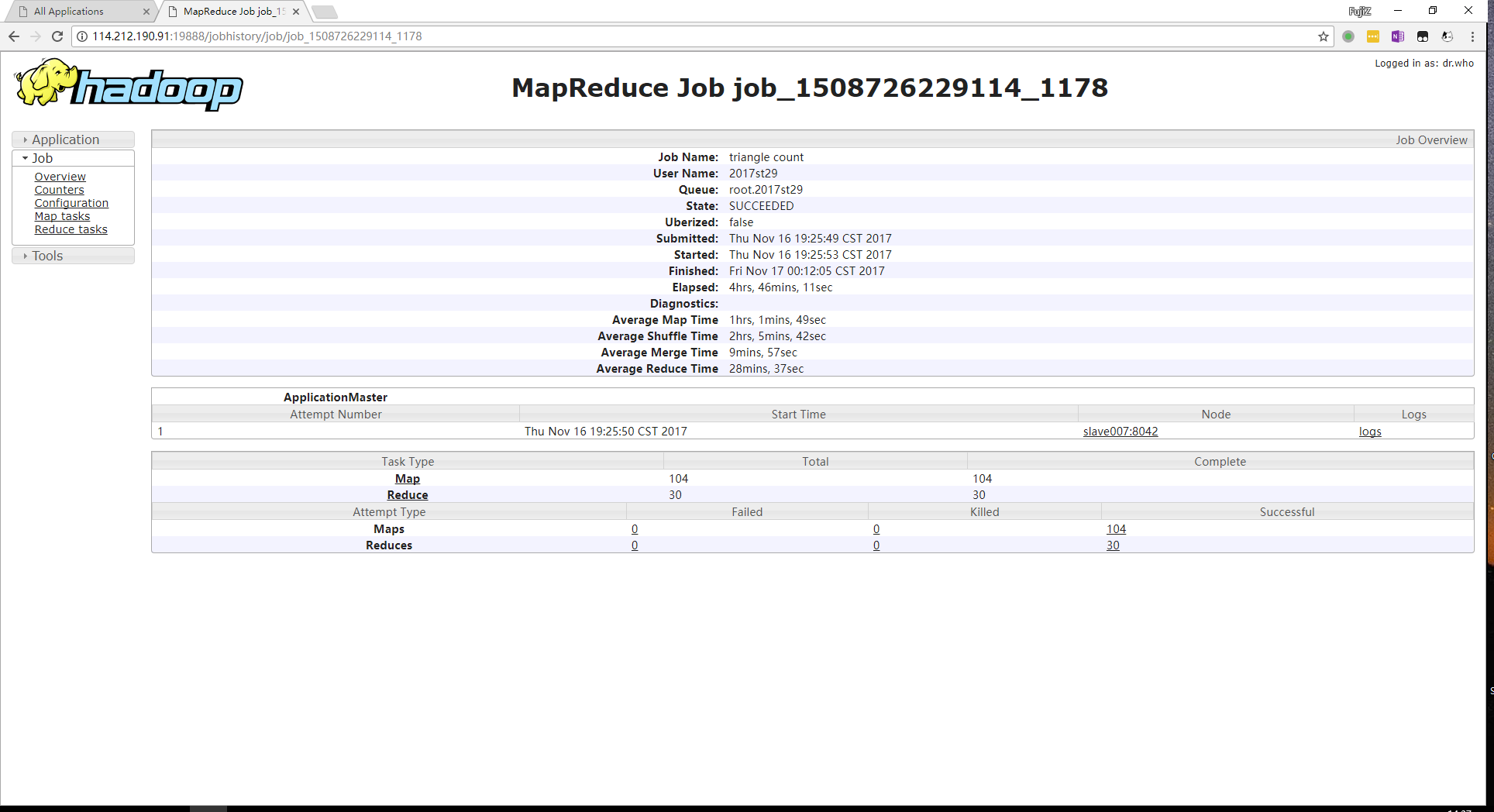


图 9 Google三角形计数Job



图 10 Google三角形计数Counters

# 程序运行性能分析

对于Google的数据集，生成邻接表的Job仅花了5分钟的时间，而计算三角形个数的Job花了4个多小时时间。而在计算三角形个数的Job中，Average Map Time为1小时，Average Reduce Time为28分钟，Average Shuffle Time为2小时，即该Job中绝大部分时间用在了Shuffle过程中。这可能是由于Mapper生成的中间结果过多导致的：对于邻接表中的任意一行，假设存在n个邻居节点，则最后发射的entry数量级为。

# 不足和改进

在测试Twitter数据集时，我们使用IntWritable来表示节点id，但是在Google的数据集中，节点id过长，以至于LongWritable也无法存下。因此我们便退而求其次，使用Text来表示节点id，上述的测试均是在这种表示方式下完成的。如果出于减少I/O开销的目的考虑的话，可以通过自定义数据类型来表示节点id，这大约能够使I/O开销减少到原来的1/2到1/3。但与此同时由于需要对String进行parse，所以Map的耗时可能会增加。

在TriangleCountMapper中，为了使发射的entry满足偏序约束，我们首先对values列表进行排序。而排序的过程其实可以在AdjList中通过组合键的方式完成，这样可以减少TriangleCount Job中Map阶段消耗的时间。但是考虑到在TriangleCount Job中的时间主要消耗在Shuffle阶段，该措施对性能的提升可能不会很大。

# 运行方式说明

jar包名为mapreduce-lab.jar。执行以下命令：

|  |
| --- |
| hadoop jar mapreduce-lab.jar triangle\_count.TriangleCountDriver input adj output undirected|directed split\_size num\_reducer |

其中input为输入文件，adj为邻接表的输出目录，output为三角形计数的输出目录；第四个参数用于指定有向边到无向边的转换方式（undirected为必做内容，directed为选做内容）；split\_size用于指定三角形计数中文件分片的大小，单位为MB；num\_reducer用于指定三角形计数中Reducer的数量。