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：

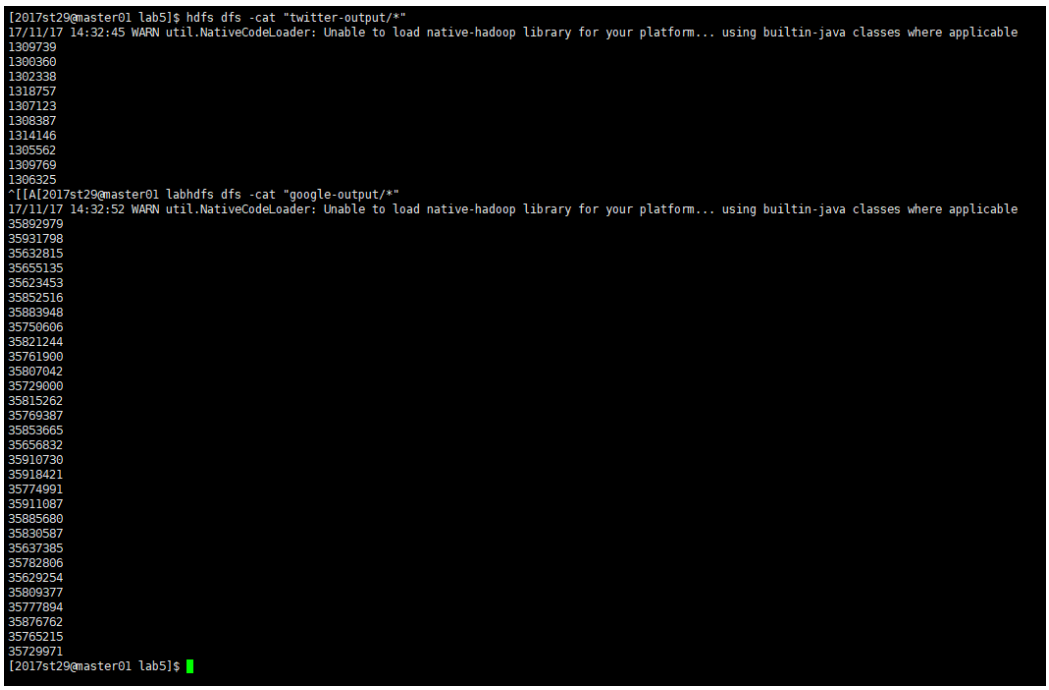

图 1 必做部分输出结果

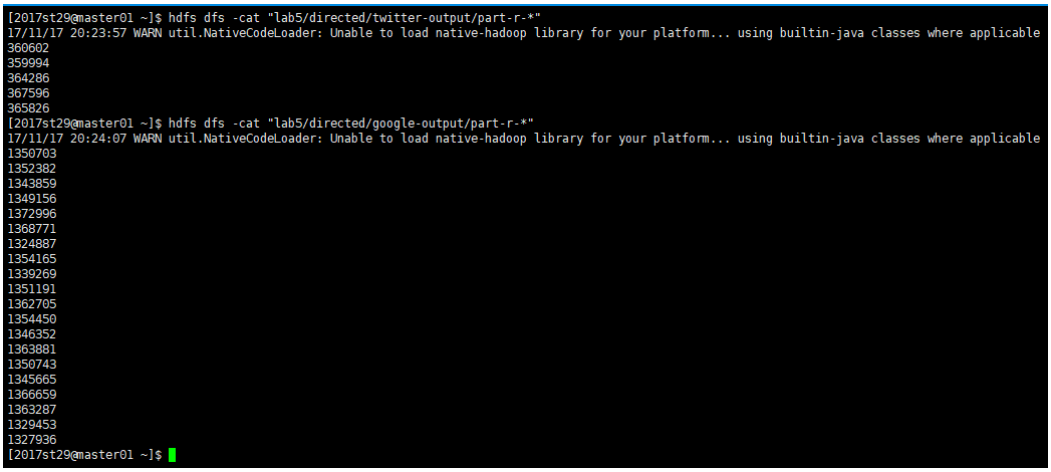

图 2 选做部分输出结果

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

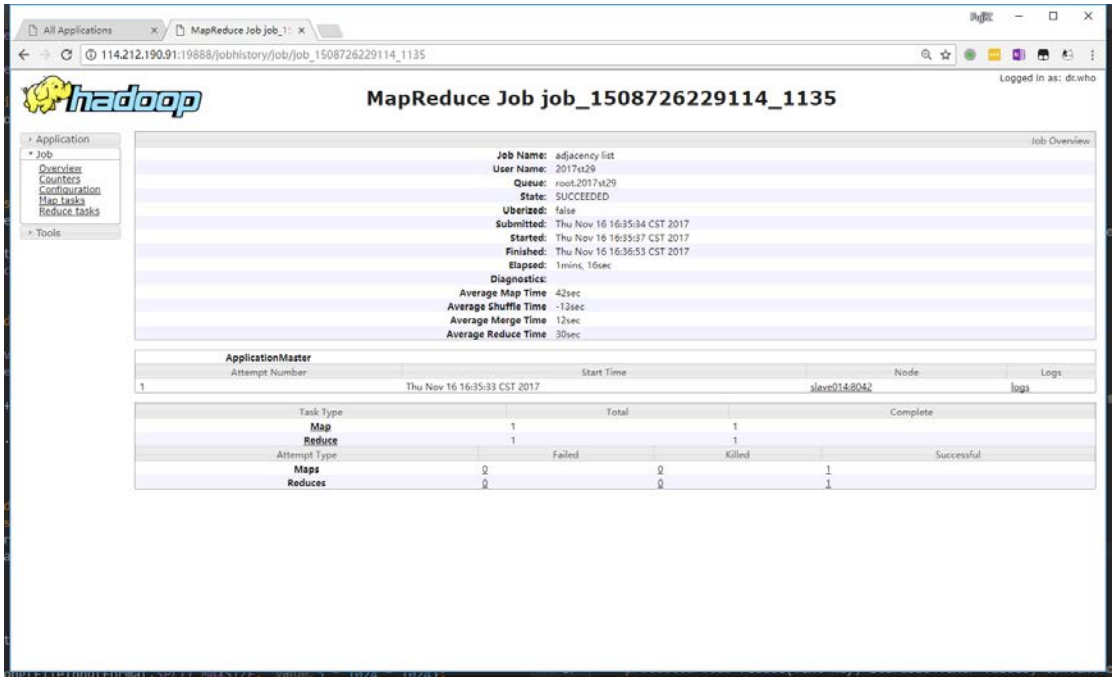

图 3 Twitter 邻接表 Job

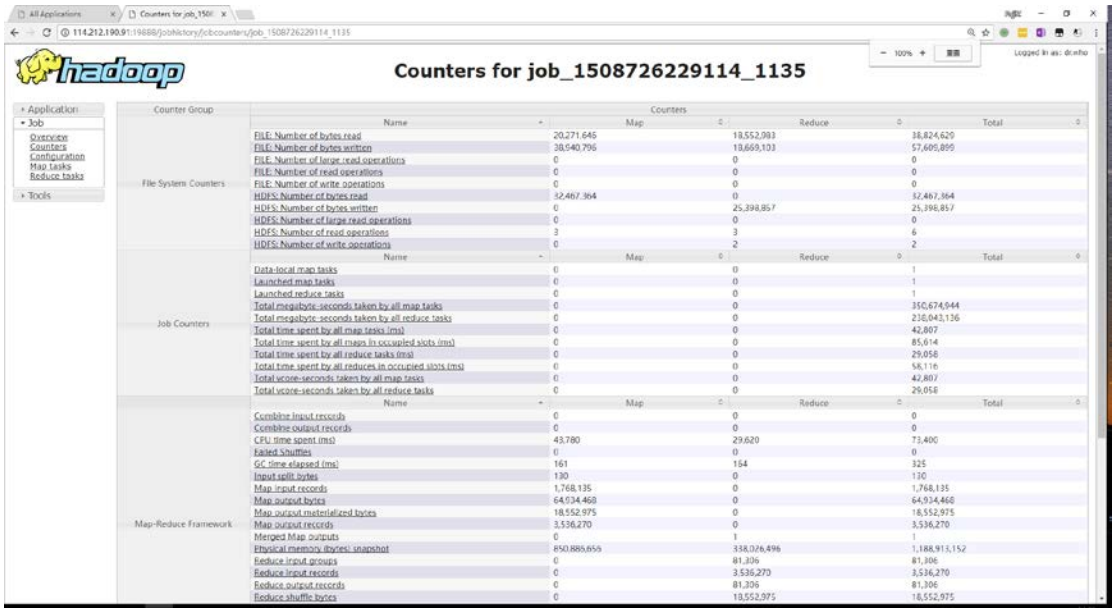

图 4 Twitter 邻接表 Counters

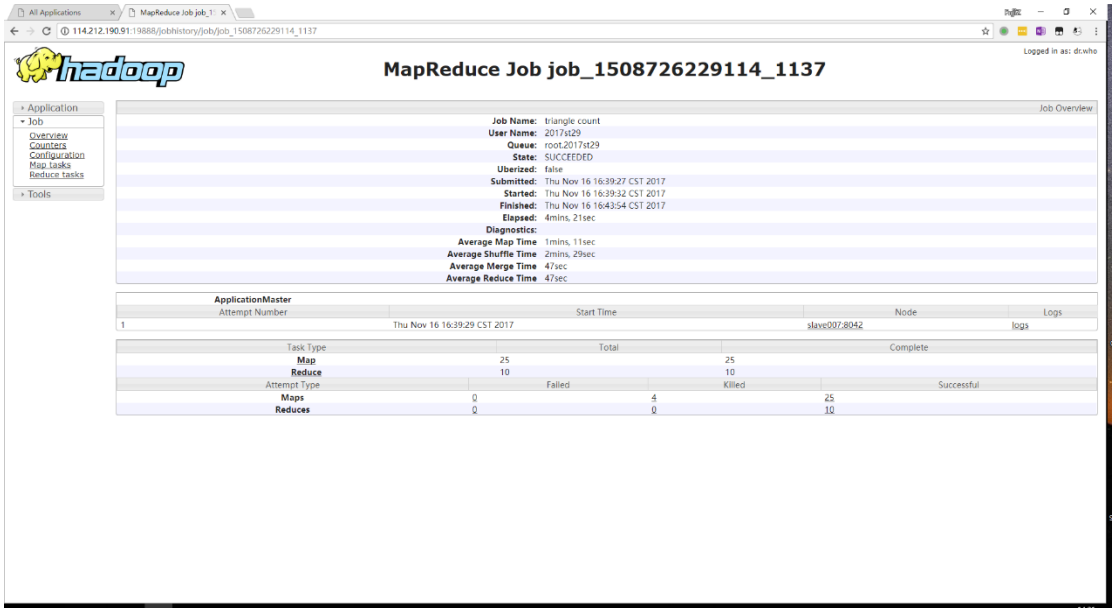

图 5 Twitter 三角形计数 Job

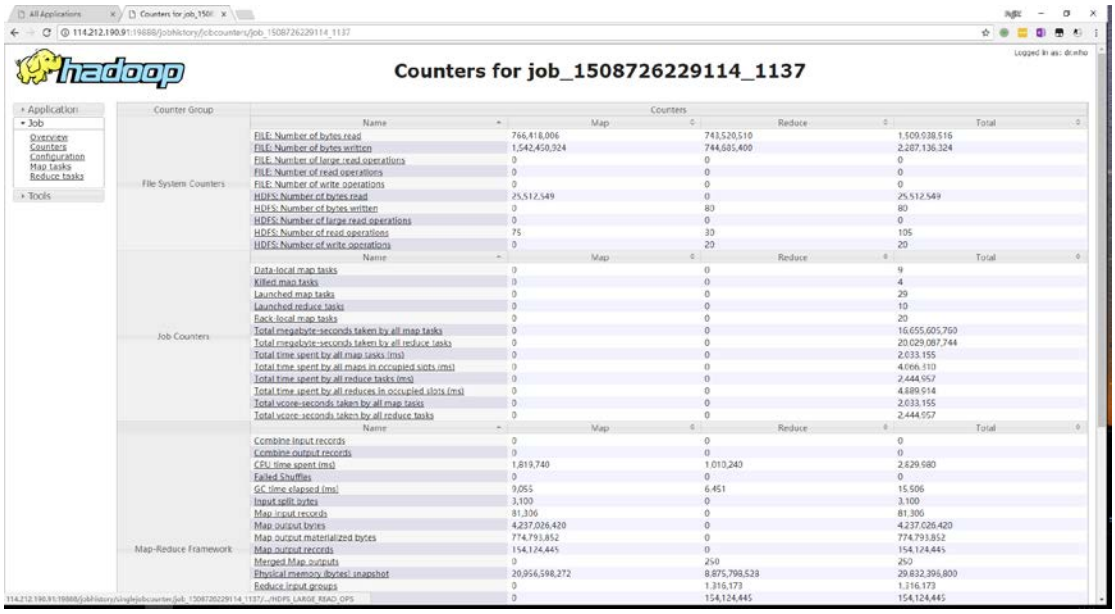

图 6 Twitter 三角形计数 Counters

Google

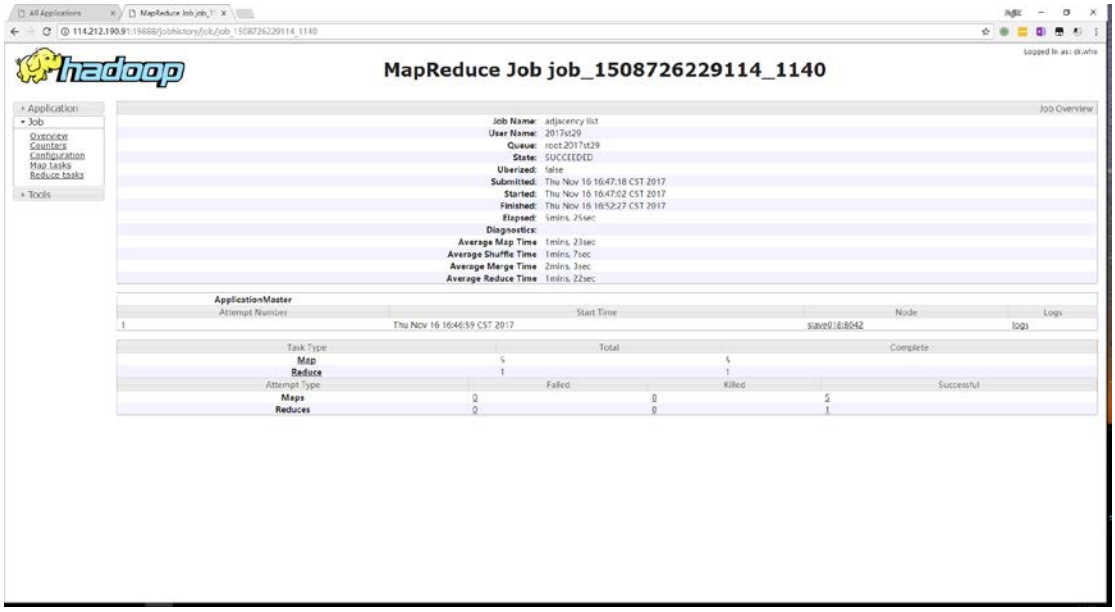

图 7 Google 邻接表 Job

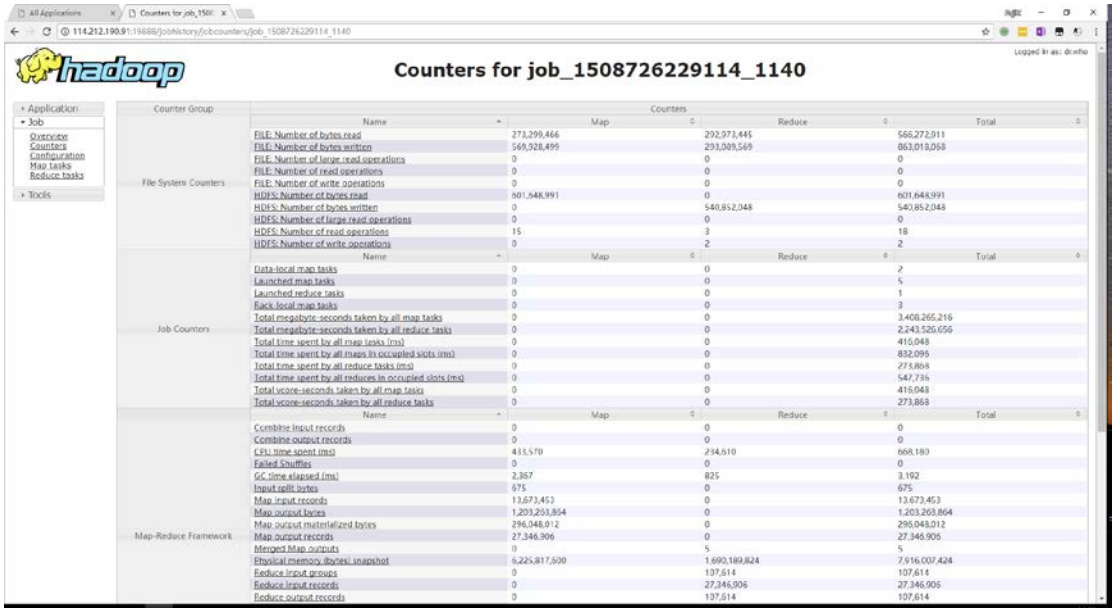

图 8 Google 邻接表 Counters

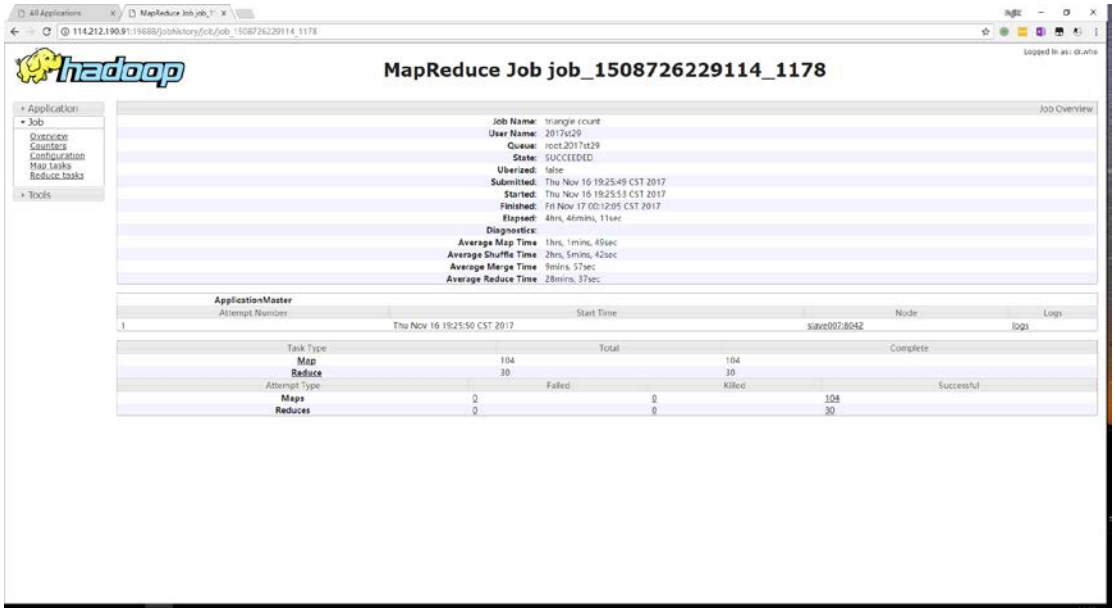

图 9 Google 三角形计数 Job

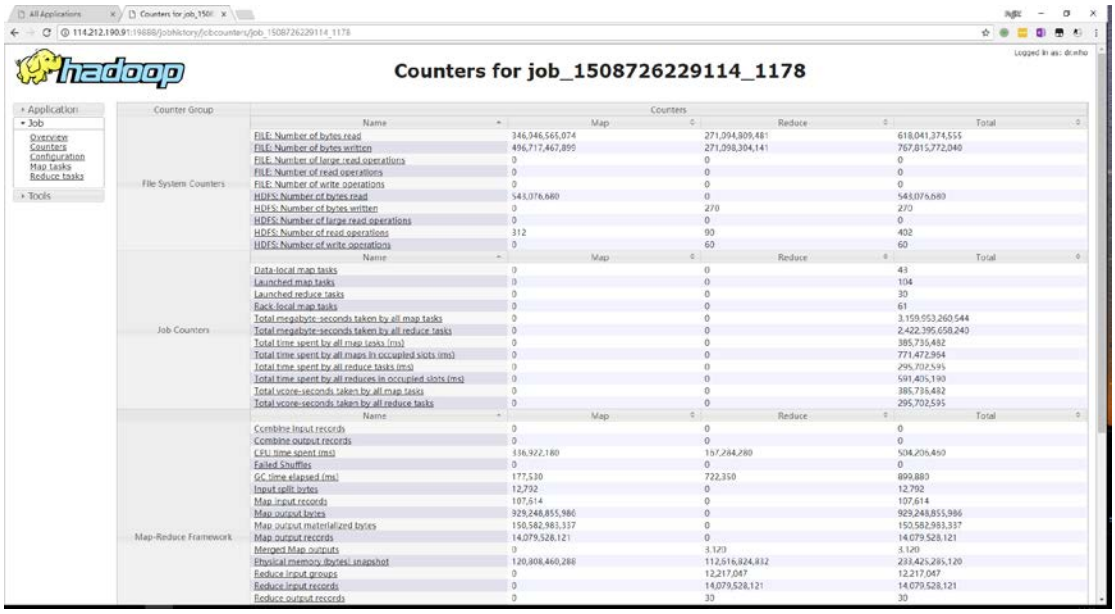

图 10 Google 三角形计数 Counters

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$O(n^2)$。

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。