基于All-Pairs&Cluster Join的相似连接

许逸培 18307130103
夏海淞 18307130090
黄尹璇 18307130340
黄韵澄 18307130341

任务背景

给出一个点集，要求计算出点集里“相似度”（如特征向量距离）达到一定阈值的点对。

该问题的解决方法可以应用于检测广告链接的虚假点击、数据清洗、文档分类等诸多不同领域，具有较高的应用价值。

在本次课程项目中，小组成员设计了基于All-Pairs和Cluster Join思想的MapReduce算法，在LinkedGeoData不同规模的数据集下进行了测试，并针对发现的问题提出相应的优化措施。

算法描述

针对相似连接问题，All-Pairs和Cluster Join均将点集D划分为互不相交的子集S_i，在不同机器上同时运行寻找相似点对的朴素算法。然而，相似点对中的点除了均位于同一子集的情况，还存在位于不同子集的情况。因此问题的核心在于：寻找和子集内某些点相似的子集外的点。下文将该类点称为“外点”。

All-Pairs

辨别外点

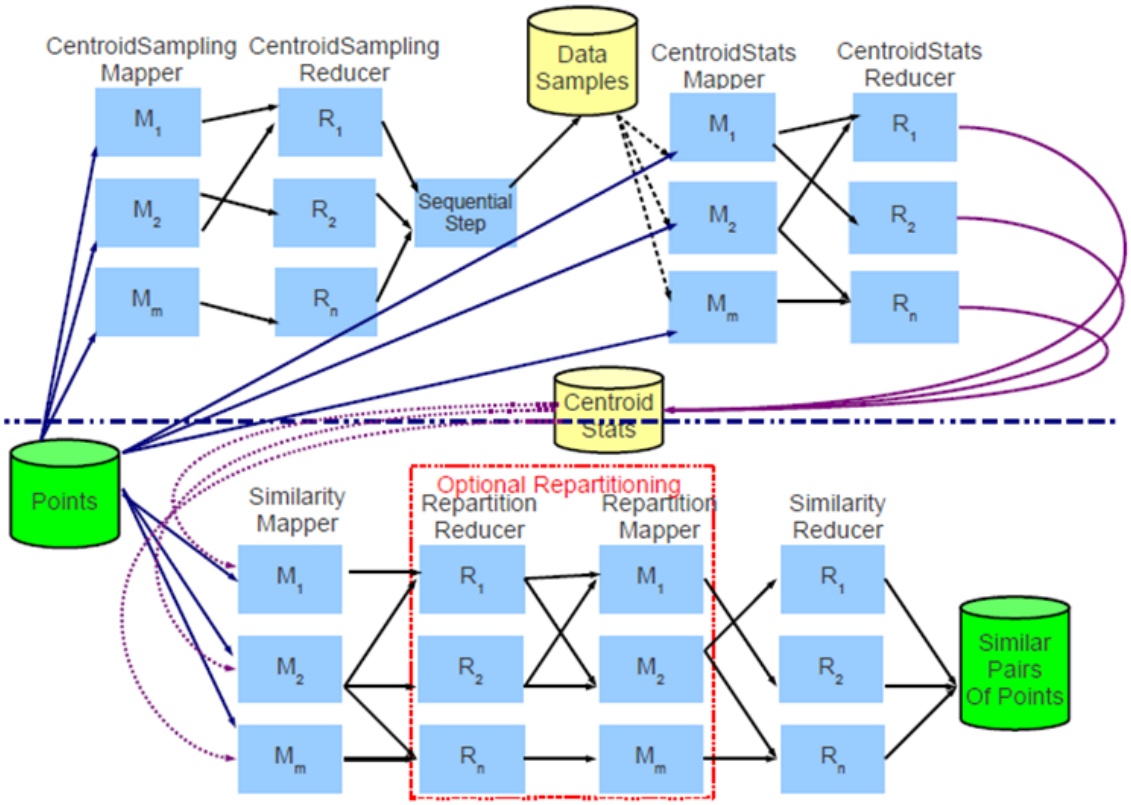
All-Pairs寻找外点的原理类似最短路径算法中的松弛操作。

设距离阈值为m，子集半径为r，则子集的外点到子集中心点距离必定小于$r+m$。因此只需对符合上述条件的外点做进一步验证，大大减少了枚举范围。该方法需要在计算出点集划分的同时计算出划分子集的半径。

重复统计问题

在设计算法时，注意到如果两个相似点位于不同子集，运行时该相似点对会被重复统计。因此增加限制条件：当且仅当外点所在子集ID小于该子集ID时才被统计。解决了重复统计的问题。

算法架构



设计的算法主要分为四部分：

1. 采样Sample
2. 分区Cluster
3. 重分区Repartition
4. 相似连接Similarity

采样部分

采样部分的Map Task将点集作为输入数据，key-value形式为`(null,Point)`；

输出数据为采样后的点集（作为子集的中心点），key-value形式为`(Point,null)`。

采样部分的Reduce Task将输入数据原样输出。

分区部分

分区部分的Map Task将点集和中心点集作为输入数据，key-value形式为`(null,Point)`；

在`map`函数中，程序枚举中心点，求出距该点最近的中心点，将其作为输出数据。key-value形式为`(Point,Distance)`。

分区部分的Reduce Task求出输入数据`(Point,List<Distance>)`中`List<Distance>`的最大值，将其设为子集半径，输出`(Point,null)`。

重分区部分

因为All-Pairs在对点集进行划分后，每个子集中依然需要朴素地寻找相似点对，因此Reduce Task的负载取决于划分后子集的规模。子集大小不均衡会导致任务负载不均，降低算法性能。

因此算法采用了重分区的方法，在子集划分结束后检查子集规模是否过大，如果满足条件则再次随机采样，重复上述步骤，直到子集规模趋于均衡为止。经过小数据试验后，算法将子集规模和采样率的积作为重分区阈值标准。

相似连接部分

相似连接部分采用了朴素的枚举方法。Map Task将子集和点集作为输入数据，key-value形式为`(Point ID,Point)`；

在`map`函数中，程序枚举子集中的中心点，将符合筛选条件的内点和外点输出。key-value形式为`(Point,Point)`。

相似连接部分的Reduce Task在子集中枚举所有点对，求出距离小于给定阈值的相似点对。

Cluster Join

Cluster Join寻找外点的方法为：

设点p的中心点为c，另一中心点为c'。则p称为c'所在子集的外点的条件为：

$$dist(p,c)+2m \geq dist(p,c')$$

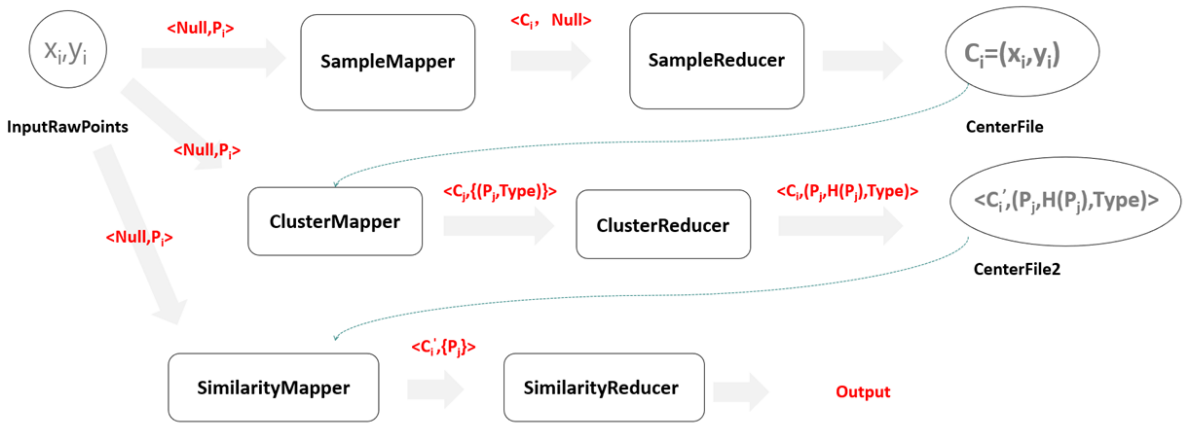
该条件直觉上较All-Pairs设定的筛选条件更为严格。

与All-Pairs方法类似，Cluster Join在实际运行时同样需要采用规定外点ID大小的方法去除重复统计的内外相似点对。

均衡分区

Cluster Join通过自定义的哈希函数将子集中的点尽可能均匀地映射成k份。具体做法是将点(x,y)映射至$(hash(x),hash(y))$，然后将映射后的点均匀分成若干块，数量与哈希函数的设置相关。因此可以通过检查原始子集的规模动态地决定哈希函数，从而控制划分子集的规模，效率较All-Pairs的重分区方法更为高效。

算法架构



设计算法主要分为三部分：

1. 采样Sample
2. 分区Cluster
3. 相似连接Similarity

其中采样部分和相似连接部分类似。

因为Cluster Join采用了基于哈希算法的分区策略，因此该算法相较All-Pairs算法去掉了重分区的步骤。

代码解读

All-Pairs

计算分区信息

mapper：

```
DoubleWritable mnd = new DoubleWritable(1e10);
PointWritable mnp = new PointWritable();
Iterator<PointWritable> cs = centers.iterator();
while (cs.hasNext()) {
    PointWritable c = cs.next();
    double dis = Math.sqrt((x - c.getx()) * (x - c.getx()) + (y - c.gety())
* (y - c.gety()));
    if (dis < mnd.get()) {
        mnd.set(dis);
        mnp = (PointWritable) c.clone();
    }
}
context.write(mnp, mnd);
```

把每个点和每个分区中心点求一遍距离，找到最近的中心点，把该点（距离）划分给那个分区。

reducer：

```
PointWritable c = new PointWritable(key.getid(), key.getx(), key.gety(), 0,
0);
for (DoubleWritable d : value) {
    c.setr(Math.max(c.getr(), d.get()));
}
context.write(c, NullWritable.get());
```

求出每个分区离中心最远的点距。

均衡重分区

mapper：

```
for (int i = 0; i < centers.size(); i += 1) {
    PointWritable c = centers.get(i);
    double dis = dist(c, pt);
    if (i == mnp) {
        PointWritable p = new PointWritable(key.get(), x, y, 0, 0);
        context.write(c, p);
    } else if (mnp != i & c.getr() + m + eps > dist(c, pt)){
        if (((i + mnp) % 2 != 0) ^ (mnp < i)) {
            PointWritable p = new PointWritable(key.get(), x, y, 0, 1);
            context.write(c, p);
        }
    }
}
```

根据每个中心点的信息，找出每个分区的内外点。

reducer：

```
if ((double)(points.size()) * samplerate > 1) {
    for (int i = 0; i < points.size(); i += 1) {
        double p = Math.random();
        PointWritable ct = (PointWritable) (points.get(i)).clone();
        if (p < samplerate) centers.add(ct);
    }
}
/*
    根据重采样得到的中心点求每个分区的半径
*/
for (int i = 0; i < points.size(); i += 1) {
    /*
        找到点 i 对应的分区 mnp
    */
    PointWritable inner = (PointWritable) p.clone(); inner.settype(0);
    PointWritable outer = (PointWritable) p.clone(); outer.settype(1);
    context.write(centers.get(mnp), inner);
    for (int j = 0; j < centers.size(); j += 1)
        if ((mnp =/= j) & (centers.get(j).getr() + m + eps >
dist(centers.get(j), p))){
            if ((((j + mnp) % 2 =/= 0) ^ (mnp < j)))
                context.write(centers.get(j), outer);
        }
}
```

如果每个分区和采样率的乘积>1，说明该分区较大，就根据采样率进一步细分分区。

注意，如果点 i 是总分区的外点，对于新划出的小分区来说仍是外点；否则，根据判断标注判断点 i 是否是小分区外点。

Cluster Join

划分分区的部分与 *All-Pairs* 相似，不再予以说明。

分区均衡

reducer：

```
double pr = Double.valueOf(context.getConfiguration().get("samplerate"));
int m = (int) (pr * points.size()) + 1;
PointWritable ct = (PointWritable)center.clone();
ct.settype(m);
for (int i = 0; i < points.size(); i += 1) {
    int id = Hash(points.get(i), i, m);
    PointWritable pt = (PointWritable) points.get(i).clone();
    pt.setr(id);
    context.write(ct, pt);
}
```

哈希的范围为 $s = \text{分区大小} \times \text{采样率} + 1$，把该分区的点映射到 $[0, s)$ 中，即将点对 (i, j) 映射到 $(hash(i), hash(j))$ 的单元格中。

寻找相似连接

mapper：

```
for (int i = p.getr(); i < c.gettype(); i += 1) {
    PointWritable cid = (PointWritable) c.clone();
    cid.setid(p.getr());
    cid.settype(i);
    context.write(cid, p);
}
for (int i = 0; i < p.getr(); i += 1) {
    PointWritable cid = (PointWritable) c.clone();
    cid.setid(i);
    cid.settype(p.getr());
    if (i ≠ p.getr()) context.write(cid, p);
}
```

根据哈希的结果，将点映射到会被包含的单元格 $(hash(i), hash(j))$ 中。

reducer：

```
for (PointWritable v : value) {
    PointWritable pt = (PointWritable) v.clone();
    if (pt.getr() ≠ key.gettype())
        right.add(pt);
    else if (pt.getr() ≠ key.getid())
        left.add(pt);
    else
        mid.add(pt);
}
for (int i = 0; i < left.size(); i += 1) {
    if (left.get(i).gettype() ≠ 0) break;
    PointWritable inner = left.get(i);
    for (int j = right.size() - 1; j ⩾ 0; j -= 1) {
        PointWritable outer = right.get(j);
    }
}
for (int i = 0; i < right.size(); i += 1) {
    if (right.get(i).gettype() ≠ 0) break;
    PointWritable inner = right.get(i);
    for (int j = left.size() - 1; j ⩾ 0; j -= 1) {
        PointWritable outer = left.get(j);
        if (outer.gettype() ≠ 1) break;
    }
}

for (int i = 0; i < mid.size(); i += 1) {
    if (mid.get(i).gettype() ≠ 0) break;
    PointWritable inner = mid.get(i);
    for (int j = i + 1; j < mid.size(); j += 1) {
        PointWritable outer = mid.get(j);
```

```
31      }
32  }
```

找出一个单元格的内点和外点，寻找相似连接。

实验结果

All-Pairs

采样率	small	middle	large
0.001	00:07:32	00:09:28	/
0.0005	00:07:23	00:08:59	/
0.0001	00:07:29	00:09:13	/
0.0005+调参	00:06:45	00:08:33	00:29:11

最后一组测试中，我们在采样率为0.0005的条件下调整了`map`函数的输入规模限制，增加了`Reduce Task`的数量，提高了资源利用率，取得了一定效果。

Cluster Join

采样率	small	middle	large
0.001	00:04:25	00:05:15	/
0.0005	00:04:20	00:05:10	/
0.0001	00:04:20	00:05:09	00:15:27
0.00005	00:04:22	00:05:10	00:15:28
0.0001+调参	00:04:05	00:05:08	00:10:19

最后一组测试中，我们在采样率为0.0001的条件下调整了`map`函数的输入规模限制，增加了`Reduce Task`的数量，提高了资源利用率，取得了一定效果。

方法比较

Cluster Join方法相较All-Pairs方法效率明显更高，验证了之前设计算法时的猜想。

Cluster Join方法性能更好的可能原因是：

- Cluster Join的外点筛选条件不仅与内点所在子集的中心点相关，还与外点所在子集的中心点相关，利用了更多的信息；
- Cluster Join的分区均衡方法依赖高效的哈希算法，避免了All-Pairs算法执行重分区造成的性能损失。

问题分析与改进

参数调整

为了实现较好的性能，我们把采样率分别设置为了0.0005和0.0001，在上述测试中表现较好。

在一开始，发现虽然程序运行很慢，但负载一直上不去，CPU 资源占用很少。之后发现 mapper 和 reducer 数量太少，无法发挥出集群的性能。

all-pairs

中心点采样： $input.maxsize = 1048576$

分区划分：$input.maxsize = 262144$

重分区：$input.maxsize = 524288,\ reduce\_number = 16$

相似连接查找：$input.maxsize = 268435456,\ reduce\_number = 20$

clusterjoin

中心点采样： $input.maxsize = 1048576$

分区划分和均衡：$input.maxsize = 393216,\ reduce\_number = 16$

相似连接查找：$input.maxsize = 268435456,\ reduce\_number = 30$

问题分析

实验结果尽管表明Cluster Join相较All-Pairs具有明显优势，然而两种方法在小规模数据集上运行时间均远高于单机枚举的时间。该情况随着数据规模的增加有所改善。

分析后认为该情况的原因主要有：

- 小规模数据集下点集离散程度较高，无法在现有点中找到较好的点作为子集中心；
- 算法部分较多，Reduce Task生成的输出数据多次写入HDFS，IO开销较大。

改进措施

1. 将两种算法中分区部分子集中心由原本的子集中心点修改为子集质心，提高了分区后筛选外点的准确性；
2. 采用压缩算法减少Reduce Task生成的中间文件大小，降低了IO开销。

项目总结与分工

项目总结

在本次课程项目中，我们了解了相似连接问题及其应用价值，学习All-Pairs和Cluster Join方法后设计了基于MapReduce的算法，在不同规模的数据集上进行测试。在发现测试问题后，尝试分析背后的原因，提出相应的改进措施，取得了一定的效果。

项目分工

许逸培：项目代码实现

夏海淞：项目报告撰写

黄尹璇：项目PPT制作

黄韵澄：算法评估实验