大数据分析技术 课程项目报告 - 搜索记录频繁模式挖掘

作者：CLDXiang

GitHub 仓库地址：github.com/CLDXiang/Mining-Frequent-Pattern-from-Search-History

一、项目概述

在这个课程项目中我的选题是搜索记录频繁模式挖掘，试图从搜狗实验室用户查询日志数据中找出搜索记录中有较高支持度关键词的频繁二项集。在实现层面上，我搭建了一个由五台服务器组成的微型 Hadoop 集群，并且用 Python 实现了 Parallel FP-Growth 算法中的三个 MapReduce 过程，并成功在集群上进行了计算。在后面的部分，我通过调整 Hadoop 的部分设置与算法中的一些参数，对不同情况下集群的计算效率进行了粗略对比。

二、项目目标、数据集及环境概述

2.1 项目目标

简单来说，目的就是构建一个搜索记录的频繁模式模型，这个模型可以做到：

输入一个关键词，输出经常和这个关键词一起被搜索的其他关键词。

在我的实现中，仅输出频繁二项集，因为更多项的频繁项集都可以通过频繁二项集递归合并生成。

2.2 数据集

数据集采用老师提供的**搜狗实验室用户查询日志数据集**(https://www.sogou.com/labs/resource/q.php)。

> 搜索引擎查询日志库设计为包括约1个月(2006年8月)Sogou搜索引擎部分网页查询需求及用户点击情况的网页查询日志数据集合。为进行中文搜索引擎用户行为分析的研究者提供基准研究语料。

数据集中共有 31 个文本文件（分别记录一天的用户查询日志），每一个的大小在 40MB ~ 80MB 量级，分别存有 50 万 ~ 80 万条记录。

对于每个文件，每一行代表一条记录，包含访问时间、用户 ID 、查询串、该URL在返回结果中的排名、用户点击的顺序号、用户点击的 URL 等字段（这是数据集说明中的描述，但是事实上最大的数据集中并没有访问时间这一字段）。

其中对于我来说需要用到的字段只有两个：(1) 用户 ID 和 (2) 查询串。

2.3 实验环境

本地：

- macOS Mojave 10.14.6 (18G103)

服务器：

- 五台均为：centos-release-7-2.1511.el7.centos.2.10.x86_64

集群组件及其他软件版本：

- JDK: jdk-8u231-linux-x64
- Zookeeper: zookeeper-3.4.14
- Hadoop: hadoop-2.8.5
- HBase: hbase-2.1.0
- Python: Python-3.7.3

三、数据集预处理及集群搭建

在实现解决问题的具体算法之前，要对数据集进行清洗并转换为适合读取的格式，并且搭建好 Hadoop 计算集群。

3.1 数据集预处理

数据预处理所需计算量并不大，所以全部在本地处理即可，无需使用集群。

3.1.1 数据清洗

搜狗实验室提供的数据集并不能直接拿来用，其主要存在两个问题：

1. **文件编码不是 `UTF-8`** ：经过我的测试，存有数据的文本文件主要的编码方式是 `GB18030` ，而非现在常用的 `UTF-8` ，这应该是因为这个数据集时间比较早（2006年）。另外还有少量的数据无法通过 `UTF-8` 、 `GB18030` 、 `GB2312` 、 `GBK` 、 `Big5` 中任一编码方式解码。
2. **数据格式不规范** : 其中很多数据格式并不规范，比如用户 ID 中会出现一些不应出现的符号，或是关键词没有由中括号包裹等。

针对第一个问题（文件编码不是 `UTF-8` ），我先尝试将文本由 `GB18030` 转为 `UTF-8` 编码，对于无法识别的记录就直接丢弃。代码见 `preprocess/gb2utf8.py` ，丢弃的记录数量见下：

日期	无法识别的记录数
20060815	1
20060821	1
20060831	3
其他	0

针对第二个问题（数据格式不规范），我主要采用正则匹配的方式筛选格式正确的记录，对于格式无法匹配的记录直接丢弃。这部分的代码见 `preprocess/format_file_v2.py` 。

具体来说，**正则匹配的规则**为： ^([A-Za-z0-9*]+)\s([.+])\s(\d+)\s(\d+)\s([,:+~^|;!$*{}"><'`@A-Za-z0-9-[]()\./%=?&]+)$

单匹配上了还不够，进一步**筛选的要求**包括：

- 匹配上的字段数刚好 5 条；
- ~~用户 ID 仅包括字母数字和下划线；~~（我在之后去掉了这条要求，因为我意识到这可能是由编码转换导致的，而且并不会对后续的数据挖掘造成不利影响）
- 查询串字段由中括号对 `[` 和 `]` 包裹；
- 结果排名和用户点击顺序号为整数；
- 域名中至少有一个 `.` ；

- 查询串不为“陋俗”；（经过我分析确认，这是搜狗对数据集中一些敏感关键词的屏蔽处理，对我们的数据挖掘没有帮助）

此外，为了提高后续分词的准确性，我还去掉了查询串中的所有 `\t` ，并将其中的所有全角字母和数字转换为半角。处理结果如下：

日期	有效记录条数	丢弃记录条数
20060801	1030571	6
20060802	773302	4
20060803	809359	0
20060804	847974	3
20060805	685021	5
20060806	681373	12
20060807	822619	8
20060808	816961	4
20060809	797805	24
20060810	807687	12
20060811	769746	10
20060812	655575	11
20060813	667486	16
20060814	804779	8
20060815	808842	6
20060816	785230	5
20060817	768676	3
20060818	742202	5
20060819	583118	2
20060820	591585	2
20060821	739685	5

20060822	654100	1
20060823	668822	2
20060824	630498	3
20060826	533565	4
20060827	513701	2
20060828	638480	5
20060829	643983	6
20060830	590970	4
20060831	562519	7
总和	21426234	185

3.1.2 关键词提取

用户的查询串仅能视作一个单个的字符串，而我们的目的是找出关键词之间的频繁模式，所以需要先从用户的查询串中提取出关键词。

我决定把**同一用户同一天的查询合并为同一条查询串**，这主要基于两点考虑：

- 同一用户可能反复输入相同或类似的查询串，当作多次查询考虑会提高他搜索的关键词在全局的权重，这并不符合逻辑；
- 同一用户在短时间内输入的不同查询串之间往往会存在关联性，这种关联性与我们想要挖掘的模式是正相关的，但由于数据集中时间的最小尺度为天，所以我干脆直接把同一天内同一用户的搜索都认为是相关的，我认为这是合理的。

在合并了同一用户同一天内的查询后，接下来的任务就是从合并的查询串中提取关键词，我考虑了两种策略：

1. 直接以查询串中的空白符（空格、 `\t` 等）和加号 `+` 为分隔符切分查询串。
2. 利用第三方库对查询串进行精确的关键词提取。

对比了效果后我选择了后者，因为前者会产生大量的较长的关键词，这些关键词即便在全局的出现频率也往往特别低，对我们的挖掘并无益处。

我利用了在自然语言处理领域中预处理常用到的 jieba 中文分词库进行了关键词提取，具体用到的方法是 `jieba.analyse.extract_tags(sentence, topK=20, withWeight=False, allowPOS=())` ，即基于 TF-IDF 算法的关键词抽取，由于分词并不是这一课题的核心，所以我直接采用了 jieba 自带的语料库。经过测试提取效果非常不错：

```
[关于对非法加油站的检查计划] -> 加油站 非法 检查 关于 计划
[全新飞利浦+826+手机评价] -> 826 飞利浦 全新 评价 手机
[8月去昆明旅游该穿什麽衣服 昆明游 昆明西双版纳游 美丽的西双版纳] -> 昆明 旅游 衣服 西双版纳 美丽
...
```

关键词的去重上述函数会自动完成，对于仅有一个关键词的查询串直接丢弃，因为它对于后续频繁模式挖掘不起作用。这一部分的代码见 `preprocess/to_db_jieba.py` 。

经过上述处理后，数据集文件就可以直接作为后续算法中第一个 Mapper 的输入了，其每一行都是一条事务，其中每个项目由空格分隔，如 `中科院 物理 研究所` 。

最终共得到 4812030 条有效事务。

3.2 Hadoop 集群搭建

集群搭建主要参考助教的文章《从零搭建Hadoop+Zookeeper+HBase完全分布式集群》，尽管实际上我没有用到 Zookeeper 和 HBase，还是按照这篇文章将它们部署了上去。

搭建过程基本与这篇文章一致，主要的不同在于为了避免后续繁琐的权限问题，我的集群文件基本都放在用户目录下（即 `~/cluster/` 而非 `/usr/local/cluster` ）。

这一部分花费了我挺多时间，也遇上了各种挫折和问题，具体会在这篇报告的第七部分统一总结。

最终搭建起了以一个 Master 和四个 Slave 组成的计算集群。

四、算法原理及实现

这部分会说明 Parallel FP-Growth 算法的原理以及具体实现。

4.1 Parallel FP-Growth 算法的原理

4.1.1 频繁模式挖掘问题的定义

在说明算法前，我们得先把问题的定义阐明：

记 $I = \{a_1, a_2, \ldots, a_m\}$ 为一个**项目集**，记**事务集** $DB = \{T_1, T_2, \ldots, T_n\}$， 其中 T_i 称为**事务**，是项目集的一个子集。

举个例子：

对于

```
a, b, c
c, d
a
a, d
```

这整个块是 DB， 其中每一行为一个事务，项目集 I = `{a, b, c, d}`

对于一个**模式** A （A 为项目集 I 子集）有一个**支持度** $supp(A)$，为 DB 中包含 A 的事务的数目。

当满足 $supp(A) \geq \xi$ 时称 A 为一个**频繁模式**，其中 ξ 是一个预定义的**最小支持度阈值**。

频繁模式挖掘问题：对于给定的 DB 和 ξ，要找出所有频繁模式的集合。

4.1.2 FP-Growth 算法

Parallel FP-Growth 算法可以说就是 **FP-Growth 算法的并行化实现**，所以要先说明一下一般的 FP-Growth 算法。

这个算法老师在课上讲过，其基本思想就是将所有输入的事务压缩到一棵 FP-Tree 中，再从下往上挖掘出频繁模式。

FP-Growth 算法分为两个独立的部分，第一部分为**建树**，第二部分为**从树中输出频繁模式**。下面将分别说明。

4.1.2.1 建立 FP-Tree

为了后续建树，我们要先得到一个项头表，并且将输入的数据集中的每一条事务进行排序，这需要我们扫描两次 DB：

1. 第一次扫描用于统计所有项目的全局支持度（出现次数）得到项头表。
2. 第而次扫描用于对输入的每条事务进行排序与过滤。主要目的是尽量缩小 FP-Tree 的规模。

具体来说，

第一次扫描：

先统计所有项目的全局支持度，其实就是一个经典的 MapReduce 问题：词频统计。

第二次扫描：

对 DB 中每一条事务都进行处理，将其内部的项目按照支持度从大到小排序，并将其中支持度小于某一阈值的项目全部删去。

举例：

假设词频统计得到的项头表为

项目	A	C	E	G	B	D	F
支持度	8	8	8	5	2	2	2

那么对于事务 `A B C E F O` ，其在处理后就会变成 `A C E B F` 。

这一步的具体作用会体现在建树的过程中，在所有项目都按照支持度排序后，任何有相同前缀的事务都会有**共享的祖先路径**，而不会另外再新增一条从根节点开始的路径，这就达到了横向压缩树的目的；而过滤掉支持度较小的项目其实就是对树的**剪枝**，会将树中距离根较远的且对我们目的帮助不大的节点删去，在纵向压缩了树的大小。

得到了项头表和排序后的事务集后，就可以根据每一条输入的事务自顶向下地建立 FP-Tree 了。

具体来说，对于每一条输入的事务，逐个扫描它的项目，并且在树中映射出一条对应的路径即可，每一个结点代表一个项目，结点还要维护一个支持度计数，每当它被扫描到一次就将计数加一。就实现层面来说，对于每一次扫描到某一项目，如果其上一个项目扫描到的结点的子结点中已经有该项目对应的结点，就将其计数加一，否则就新增一个计数为 `1` 的该项目对应的结点。

由于算法思路老师在上课时详细讲过，我在代码中也会有实现，此处不再举例赘述。

4.1.2.2 通过 FP-Tree 挖掘频繁模式

老师上课时也详细讲过，此处仅粗略举例说明。比如想要找到某一项目 `A` 的所有频繁二项集，就先通过项头表找出所有的 `A` 结点，然后自下而上回溯它们的计数，即统计它们的每个祖先项目的所有 `A` 后代的支持度总和，比如 `B` 的所有 `A` 后代的支持度总和为 `4` ，而我们定义的最小支持度阈值为 `3` ，那么 `B，A` 就是 `A` 的频繁二项集中的一对频繁模式。至于频繁多项集，均可以通过频繁二项集递归合并得到。

4.1.3 Parallel FP-Growth 算法基本思想

事实上， FP-Growth 算法从原理上已经能够解决我们的问题了。但是对于比较大的数据集，一次性把所有数据压进一棵树不论是对于算法的空间还是时间效率来说都是不利的。

更具体地，按照 Parallel FP-Growth 算法（后简称 PFP 算法）论文中的说法，**朴素的 FP-Growth 算法有以下的提升空间**：

1. 存储空间。对于规模较大的 DB ，相应地需要建立一棵超大的 FP-Tree ， 这样的 FP-Tree 往往无法存在主存中。所以需要将 DB 切分为一些小的 DB 分别进行处理。
2. 分布式计算。所有的步骤都应该可以并行化，特别是从树中挖掘频繁模式这一步骤。
3. 通信代价高。以往的并行 FP-Growth 算法会将连续的事务分为一组，这样产生的 FP-Trees 往往会相互依赖，并且在并行计算时需要进行同步。
4. 支持度阈值。支持度阈值的选取对于 FP-Growth 算法的影响会很大，因为支持度阈值越高，得到的频繁模式就越少，但相应的也能减少计算量和空间需求。然而对于某些挖掘任务，为了结果我们不得不将支持度阈值设置得很低，这会导致无法预期的计算时间。

PFP 算法正是用来解决这些问题的，它的核心思想就是**通过三个 MapReduce 过程，将 FP-Growth 算法分为多个计算相互独立的 FP-Growth ，分别进行频繁模式挖掘后再将结果合并**。

4.2 Parallel FP-Growth 算法的步骤

PFP 算法共有五步，其中三步可以用 MapReduce 架构实现：

1. **切片**。将 DB 分为一些连续的部分，然后将他们存储在多台机器上。每一个部分称为一个切片。
2. **并行计数**。用一个 MapReduce 对 DB 中所有的项目进行计数（这一步的目的和朴素的 FP-Growth 相同）。每一个 mapper 输入一个切片。这一步可以顺便获得项目词汇表，这对于大的 DB 往往是无法事先取得的。得到的结果存到 F-list 里。
3. **项目分组**。将 F-list 中所有 $|I|$ 个项目分到 Q 个组内。这些组的列表称为 G-list ， 其中每个组被分配一个 group-id(gid)。 因为 F-list 和 G-list 都比较小且时间复杂度为 $O(|I|)$ ，所以这一步只需要单机本地计算几秒钟即可完成。
4. **并行 FP-Growth**。这是 PFP 的核心步骤，用一个 MapReduce 过程完成。对于 mapper ，它的工作是生成“依赖于分组的事务集”：输入一个 DB 切片，对于其中的每条事务，输出其中项目对应的所有 gid 及该 gid 对应的最长前缀模式串（这里可能不好理解，可以看后文例子）。对于 reducer ，对于每个依赖于分组的切片进行 FP-Growth 算法：将同一分组的所有事务交给一个 reducer 来做，每一个分组建立一个 FP-Tree 并分别进行挖掘。
5. **汇总**：将上一步得到的结果进行汇总。即把同一项目的所有频繁模式合并起来。

这里对上面一些步骤进行一点细节上的补充说明：

- F-list 和朴素的 FP-Growth 的项头表类似，就是一个项目频率表，每一行格式为 `项目\t支持度` 。
- 并行 FP-Growth 中 mapper 的输入输出举例：

  可以通过 G-list 得到每一个项目对应的 gid，比如：

项目	gid
北京	1
广州	1
重庆	1
上海	2
深圳	2
昆明	2
拉萨	3

对于一个输入进来的事务： `{上海 广州 重庆 北京}` 。从最后一项开始扫描， `北京` 对应 gid 为 `1` ，故输出 `1\t上海 广州 重庆 北京` ，扫描到 `重庆` 和 `广州` 时不输出，因为它们的 gid 也是 `1` ，已经被输出过了。而到了 `上海` ，其 gid 为 `2` ，就输出 `2\t上海` 。

所以上述输入最后对应两个输出：

```
1\t上海 广州 重庆 北京
2\t上海
```

再举一个例子，对于输入 `{北京 广州 重庆 上海 深圳 昆明 拉萨}` ，输出将是：

```
3\t北京 广州 重庆 上海 深圳 昆明 拉萨
2\t北京 广州 重庆 上海 深圳 昆明
1\t北京 广州 重庆
```

- 并行 FP-Growth 中 reducer 的输出形式将会是 `模式串\t模式串的支持度` ，比如在这一次 reduce 中统计到 `上海 北京` 出现了 `100` 次，就输出 `上海 北京\t100` 。
- 最后一步汇总其实就是将上一步 reduce 的结果按项目全部加起来。比如在两个 reduce 中分别有：

```
[reduce1] 上海 北京\t100
[reduce2] 广州 上海\t50
```

那么汇总的输出结果将会是：

```
上海\t[上海 北京] [广州 上海]
北京\t[上海 北京]
广州\t[广州 上海]
```

4.3 Parallel FP-Growth 算法的实现

在实现过程中，为了能够更好地利用 Hadoop 提供的框架，我在一些实现层面的处理上并没有严格按照 PFP 算法的伪代码来实现，但是基本思路是完全一致的。

另外值得注意的是， Hadoop 中的 Map 和 Reduce 之间会自动根据键进行 Shuffle 和 Sort ，我在实现的过程中充分利用到了这一特性。

算法部分采用 Python 实现，利用 Hadoop Streaming 嵌入到 Hadoop 的 MapReduce 中执行。

4.3.1 切片

这一部分其实并不需要我显式进行操作，因为本来数据文件已经被分为了 31 份，不少于集群中 mapper 的数量，在上传到 HDFS 中后会自动进行分布式存储，其实就已经起到了切片的作用。

当然了，在 mapper 的数量多于输入文件个数的时候，可以将输入文件中的数据再次切分直到不少于 mapper 的个数，并分别上传到 HDFS 中存储，这虽然可能会浪费一些空间（若单个文件大小小于 HDFS 中块的大小），但可以提高计算效率。

4.3.2 并行计数

这是第一个 MapReduce 过程，代码见 `mapper1.py` 和 `reducer1.py` 。过程与经典的词频统计的 MapReduce 实现几乎完全一致。

Mapper:

每行输入： DB 中的一个事务，如 `A B C A`

对于事务中的每一个项目 `X` ，输出键值对 `X : 1`

如对上例，输出：

```
A   1
B   1
C   1
A   1
```

Reducer:

在 Hadoop MapReduce 中，有相同键的行会在 Shuffle 阶段被放到一起，这为 Reduce 的实现提供了便利。

这一步的思想就是将键相同的项的值求和。

维护一个键记录 `current_word` 和一个计数器 `current_count` ，分别记录 Reduce 中上一项的键及其目前的计数和。

举例来说，对于一个输入： `A N` ，若上一个输入键也为 `A` ，则将计数累加： `current_count += N` ；若与上一个输入键不同，说明上一个键已经 Reduce 完毕，输出键值对 `current_word : current_count` ，然后令 `current_word = A, current_count = N` 。当然了，此处的 `N` 应该都为 `1` 。

注意在最后还要再额外输出一次 `current_word : current_count` 。

举一个稍微全局一点的例子，比如对于 Map 并 Shuffle 后的输入：

```
A   1
A   1
B   1
B   1
B   1
C   1
```

输出结果将是

```
A   2
B   3
C   1
```

4.3.3 项目分组

考虑后续对于 G-list 的实际使用，其实是将 G-list 作为一个从项目到 gid 的映射表使用的，所以比起实际建立一个包含所有组的 group list，不如直接建立一个从项目到 gid 的映射表，在此处也称为 G-list。

此处涉及超参数分组数 Q，在实现中其实需要的是每一组的项目数 $|I| = \text{项目总数} \div Q$，所以比起多出一步不必要的计算，不如直接设置超参数 $|I|$，即每一组所含项目数。

这一步计算量其实相对比较小，可以直接在单机环境下进行。具体实现也比较简单，还可以顺便对上一步得到的 F-list 进行排序：

1. 根据值对上一步得到的 F-list 进行排序；
2. 排序后，按照每组 $|I|$ 个项目依次对每个项目分配一个 gid，从 `1` 开始，最终得到 G-list。

这部分的代码见 `sort_kv.py`。

得到的 G-list 格式如下（以 $|I| = 3$ 为例）：

```
项目1   1
项目2   1
项目3   1
项目4   2
项目5   2
项目6   2
项目7   3
...
```

即项目到 gid 的映射表。为了方便在后续程序中使用，将其直接转换为 JSON 格式，以便直接以 Python dict 类型在程序中使用。转换后的 JSON/dict 对象格式如下：

```
G_list = {'项目1':'1', '项目2': '1', '项目3': '1', '项目4':'2', '项目5': '2', '项目
6': '2', '项目7':'3', ...}
```

如此一来，在之后的程序中只需要用 `G_list[项目名]` 就能得到项目对应的 gid 了。

4.3.4 并行 FP-Growth

这是第二个，也是最关键的 MapReduce 过程，代码见 `mapper2.py` 和 `reducer2.py`。其作用就是分别对每一个组建立 FP-Tree 并进行频繁模式挖掘。

Mapper:

如前所述，扫描 DB ，对于每一条输入的事务，输出其中项对应的所有 gid 在该事务中的最长前缀串。例子在 4.2 节中已经举过了。

对于每一条输入会有多条输出，每一条输出的键值对为： `gid`：`该 gid 对应的最长前缀串（即最后一项的 gid 为该 gid 的最长前缀）` 。

在实现层面上，需要用到 G-list 来从项目映射到 gid ，我直接将上一步得到的 dict 对象静态插入到了代码里，以减少文件 IO 的开销。（在 G-list 大小超出内存容量时可能需要考虑从文件读取）

Reducer:

Hadoop 会把 Mapper 的所有输出按照相同 gid 排到一起。

Reducer 所做的事就是把所有相同 gid 的事务建立一棵 FP-Tree。在每一个 gid 的 FP-Tree 建立完毕后，找出其中支持度前 K 个（ K 为超参数）的频繁模式进行输出。

输出键值对为 `模式串`：`模式串的支持度` ，每一个 gid / 每一棵 FP-Tree 会有 K 条输出。

实现层面基本类似朴素的 FP-Growth 算法，具体请参看源代码中的实现。

4.3.5 汇总

这是第三个 MapReduce 过程，用于输出最终结果。

Mappper:

对于每一条上一步输出的 `模式串`：`模式串的支持度` ，遍历该模式串中的每个项目 `A` ，输出键值对 `A`：`模式串\t模式串的支持度` 。

也就是将 `A` 提了出来作为键。所以对于每一条输入的模式串， Mapper 的输出条数为该模式串中的项目数。

举个例子，对于输入的 `B A C`：`55` ，输出将会是：

```
A   B A C   55
B   B A C   55
C   B A C   55
```

Reducer:

Mapper 输出的键为项目， Reducer 要做的就是把相同键（相同项目）的所有值合并到一起按照支持度由大到小输出。

举例说明比较直观：

比如对于 Shuffle 过后的输入：

```
A   B A C   55
A   B A     100
A   A D     72
B   B A C   55
B   B D     11
C   C G     12
C   B A C   55
```

输出将会是：

```
A   [B A] [A D] [B A C]
B   [B A C] [B D]
C   [B A C] [C G]
```

即 A 的频繁模式有 `B A` ， `A D` ， `B A C` 三种，且支持度从大到小。（在我的实现中输出的频繁模式都是频繁二项集）

五、运行测试

5.1 如何在集群上运行

5.1.1 准备工作

首先将所有预处理完的数据文件和源代码上传到 Master 节点：

```
# 在预处理完的数据文件所在目录下
scp *.txt hadoop@master:~/pj/data/
# 在源代码文件所在目录下
scp *.py hadoop@master:~/pj/src/
```

然后在 Master 上，将上传的数据文件再存储到 HDFS 中：

```
hadoop fs -mkdir /pj
hadoop fs -mkdir /pj/data
hadoop fs -put ~/pj/data/*.txt /pj/data/
```

对于每个 Python 脚本文件，在文件头部加入 `#!/usr/bin/python3` 使其可以直接执行，然后用 `chmod +x *.py` 给予其执行权限。

5.1.2 第一个 MapReduce: 并行计数

接下来就可以通过 Hadoop Streaming 进行 MapReduce 了：

```
yarn jar /home/hadoop/cluster/hadoop/share/hadoop/tools/lib/hadoop-streaming-
2.8.5.jar -file ~/pj/src/mapper1.py -file ~/pj/src/reducer1.py -mapper
~/pj/src/mapper1.py -reducer ~/pj/src/reducer1.py -input /pj/data/*.txt -output
/pj/res1/
```

Hadoop Streaming 允许任何可执行脚本作为 Mapper 和 Reducer ， 会用到的命令参数说明如下：

- `file` : 将本地文件临时放入计算节点中，使其可以被 MapReduce 使用。
- `mapper` : 指定作为 Mapper 的脚本。
- `reducer` : 指定作为 Reducer 的脚本。
- `input` : 输入文件。
- `output` : 输出目录。
- `-D mapreduce.job.reduces=N` : 指定作为 Reducer 的节点的个数为 `N` 。

上面命令中 `hadoop-streaming-2.8.5.jar` 这个 JAR 包即为 Hadoop Streaming。

5.1.3 项目分组

可以通过 `hadoop fs -get /pj/res1/part-* ~/pj/res1/` 将得到的结果从 HDFS 拉回到本地，然后使用 `python sort_kv.py` 在本地进行项目分组。将得到的 G-list 的 JSON 对象静态拷贝到 `mapper2.py` 中的 `G-list` 变量中。

5.1.4 第二个 MapReduce: 并行 FP-Growth

同上，运行 MapReduce:

```
yarn jar /home/hadoop/cluster/hadoop/share/hadoop/tools/lib/hadoop-streaming-
2.8.5.jar -D mapreduce.job.reduces=3 -file ~/pj/src/mapper2.py -file
~/pj/src/reducer2.py -mapper ~/pj/src/mapper2.py -reducer ~/pj/src/reducer2.py -
input /pj/data/*.txt -output /pj/res2/
```

这次指定了 Reducer 的个数为 `3` ，因为这一步中 Reduce 的计算量远大于 Mapper 。

5.1.5 第三个 MapReduce: 汇总

同上，运行 MapReduce:

```
yarn jar /home/hadoop/cluster/hadoop/share/hadoop/tools/lib/hadoop-streaming-
2.8.5.jar -D mapreduce.job.reduces=3 -file ~/pj/src/mapper3.py -file
~/pj/src/reducer3.py -mapper ~/pj/src/mapper3.py -reducer ~/pj/src/reducer3.py -
input /pj/res2/part-* -output /pj/res3/
```

结果将会存储在 HDFS 中的 `/pj/res3/` 目录下，如有需要可以将其拉回本地。

5.2 测试过程及结果

我在一开始试图使用整个数据集进行测试，但是发现一直跑不动第二个 MapReduce，所以一开始的测试全都换成了比较小的数据集。但是经过后面的性能比较后，我找出了局部最优的参数设置，经过这样设置后就能够跑动整个数据集了。

输出日志以 MR3 的日志为例：

```
mapper3.py -reducer ~/pj/src/reducer3.py -input /pj/res2_final -output
/pj/res3_final/
19/12/25 21:28:18 WARN streaming.StreamJob: -file option is deprecated, please use
generic option -files instead.
packageJobJar: [/home/hadoop/pj/src/mapper3.py, /home/hadoop/pj/src/reducer3.py,
/tmp/hadoop-unjar6082271143288484985/] [] /tmp/streamjob6308769785279930146.jar
tmpDir=null
```

```
19/12/25 21:28:20 INFO client.RMProxy: Connecting to ResourceManager at
master/10.141.212.243:8032
19/12/25 21:28:20 INFO client.RMProxy: Connecting to ResourceManager at
master/10.141.212.243:8032
19/12/25 21:28:21 INFO mapred.FileInputFormat: Total input files to process : 30
19/12/25 21:28:21 INFO mapreduce.JobSubmitter: number of splits:30
19/12/25 21:28:21 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_1577267356940_0031
19/12/25 21:28:21 INFO impl.YarnClientImpl: Submitted application
application_1577267356940_0031
19/12/25 21:28:21 INFO mapreduce.Job: The url to track the job:
http://master:8088/proxy/application_1577267356940_0031/
19/12/25 21:28:21 INFO mapreduce.Job: Running job: job_1577267356940_0031
19/12/25 21:28:39 INFO mapreduce.Job: Job job_1577267356940_0031 running in uber
mode : false
19/12/25 21:28:39 INFO mapreduce.Job:  map 0% reduce 0%
19/12/25 21:28:49 INFO mapreduce.Job:  map 27% reduce 0%
19/12/25 21:28:51 INFO mapreduce.Job:  map 53% reduce 0%
19/12/25 21:29:01 INFO mapreduce.Job:  map 57% reduce 0%
19/12/25 21:29:03 INFO mapreduce.Job:  map 73% reduce 0%
19/12/25 21:29:07 INFO mapreduce.Job:  map 73% reduce 1%
19/12/25 21:29:08 INFO mapreduce.Job:  map 73% reduce 2%
19/12/25 21:29:09 INFO mapreduce.Job:  map 73% reduce 3%
19/12/25 21:29:11 INFO mapreduce.Job:  map 73% reduce 7%
19/12/25 21:29:12 INFO mapreduce.Job:  map 73% reduce 9%
19/12/25 21:29:13 INFO mapreduce.Job:  map 73% reduce 11%
19/12/25 21:29:14 INFO mapreduce.Job:  map 73% reduce 13%
19/12/25 21:29:20 INFO mapreduce.Job:  map 73% reduce 14%
19/12/25 21:29:25 INFO mapreduce.Job:  map 73% reduce 16%
19/12/25 21:29:26 INFO mapreduce.Job:  map 73% reduce 17%
19/12/25 21:29:28 INFO mapreduce.Job:  map 73% reduce 18%
19/12/25 21:33:38 INFO mapreduce.Job:  map 77% reduce 18%
19/12/25 21:33:40 INFO mapreduce.Job:  map 80% reduce 18%
19/12/25 21:33:41 INFO mapreduce.Job:  map 83% reduce 18%
19/12/25 21:33:42 INFO mapreduce.Job:  map 83% reduce 19%
19/12/25 21:33:45 INFO mapreduce.Job:  map 83% reduce 20%
19/12/25 21:33:48 INFO mapreduce.Job:  map 93% reduce 20%
19/12/25 21:33:49 INFO mapreduce.Job:  map 100% reduce 23%
19/12/25 21:33:50 INFO mapreduce.Job:  map 100% reduce 38%
19/12/25 21:33:51 INFO mapreduce.Job:  map 100% reduce 63%
19/12/25 21:33:52 INFO mapreduce.Job:  map 100% reduce 73%
19/12/25 21:33:54 INFO mapreduce.Job:  map 100% reduce 77%
19/12/25 21:34:05 INFO mapreduce.Job:  map 100% reduce 80%
19/12/25 21:34:07 INFO mapreduce.Job:  map 100% reduce 83%
19/12/25 21:34:08 INFO mapreduce.Job:  map 100% reduce 90%
19/12/25 21:34:09 INFO mapreduce.Job:  map 100% reduce 100%
19/12/25 21:34:11 INFO mapreduce.Job: Job job_1577267356940_0031 completed
successfully
19/12/25 21:34:11 INFO mapreduce.Job: Counters: 52
    File System Counters
        FILE: Number of bytes read=5400868
        FILE: Number of bytes written=20504046
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
```

```
        HDFS: Number of bytes read=1733302
        HDFS: Number of bytes written=2294296
        HDFS: Number of read operations=180
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=60
    Job Counters
        Killed map tasks=3
        Killed reduce tasks=1
        Launched map tasks=30
        Launched reduce tasks=31
        Data-local map tasks=27
        Rack-local map tasks=3
        Total time spent by all maps in occupied slots (ms)=2651041
        Total time spent by all reduces in occupied slots (ms)=6602255
        Total time spent by all map tasks (ms)=2651041
        Total time spent by all reduce tasks (ms)=6602255
        Total vcore-milliseconds taken by all map tasks=2651041
        Total vcore-milliseconds taken by all reduce tasks=6602255
        Total megabyte-milliseconds taken by all map tasks=2714665984
        Total megabyte-milliseconds taken by all reduce tasks=6760709120
    Map-Reduce Framework
        Map input records=112840
        Map output records=225680
        Map output bytes=4949328
        Map output materialized bytes=5406088
        Input split bytes=2850
        Combine input records=0
        Combine output records=0
        Reduce input groups=47048
        Reduce shuffle bytes=5406088
        Reduce input records=225680
        Reduce output records=47048
        Spilled Records=451360
        Shuffled Maps =900
        Failed Shuffles=0
        Merged Map outputs=900
        GC time elapsed (ms)=165511
        CPU time spent (ms)=422790
        Physical memory (bytes) snapshot=13888937984
        Virtual memory (bytes) snapshot=128068333568
        Total committed heap usage (bytes)=10220994560
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=1730452
    File Output Format Counters
        Bytes Written=2294296
19/12/25 21:34:11 INFO streaming.StreamJob: Output directory: /pj/res3_final/
```

最终的部分输出结果：

```
化学 [有限公司 化学] [财务总监 化学] [化学 开发] [化学 甜菜糖] [油脂 化学] [皂值 化学]
[Sap 化学] [OHV 化学] [化学 Hazen] [知识 化学]
大学 [大学 大学化学] [大学 www] [大学 site] [信息工程 大学] [大学 易三仓] [大学 学院] [科
技 大学] [学院 大学] [西安 大学] [大学 罗马大学]
数据库 [每段 数据库] [数据库 号码] [手机号码 数据库] [106781 数据库] [数据库 永久] [数据
库 下载] [gopher 数据库] [BOOTP 数据库] [什么 数据库] [telnet 数据库]
手机 [手机 天目通] [手机 各款] [三星 手机] [LG 手机] [手机 花长] [手机 触屏] [手机 货到付
款] [CECT 手机] [lg 手机] [电影 手机]
经济 [经济 核定表] [公司 经济] [有限公司 经济] [杭州 经济] [适用房 经济] [发展 经济] [实
业 经济] [经济 奇英传] [经济 女帝] [电子 经济]
名著 [读后感 名著]
名言 [萨福 名言] [名言 克瑞翁] [名言 阿那] [警句 名言] [名言 鲁迅]
发展 [北京 发展] [科技 发展] [有限公司 发展] [发展 开发] [中心 发展] [发展 途歌] [发展 时
代] [发展 网络科技] [城市 发展] [有限责任 发展]
南唐 [李煜 南唐] [后主 南唐]
坦克 [坦克 履带式] [主战 坦克] [坦克 虎式] [坦克 艾布拉姆斯] [坦克 M1A2] [坦克 梅卡瓦]
飞机 [飞机 超轻型] [航班 飞机] [飞机 企业] [维修 飞机] [飞机 山东] [飞机 STAEC0] [飞机 怎
么] [飞机 起来] [飞机 曲率] [飞机 专用]
国家标准 [国家标准 GB] [标准 国家标准] [国家标准 GB252] [国家标准 GB209] [gb 国家标准]
[国家标准 下载]
```

可以看出是有较好效果的，如果能有更大的数据集，结果应该会更加精确。另外预处理也还可以进一步优化，比如删除一些无意义的字母序列，或是将繁体转简体。

六、在不同参数、设置下的性能比较

在这个课程项目中，我不打算用客观的评估指标来评价算法效果，因为算法事实上是固定的，对于相同的数据集输入，主观上感受到的准确度差异并不会太大，所以我将更多的关注放在了**性能比较**上。

就我的实现而言，算法中可以调节的超参数共有三个值：

1. 在项目分组这一步的分组大小 `I` ；
2. 在第二个 MapReduce 过程（并行 FP-Growth ） Reduce 时筛选出最高支持度模式串的数量 `K1` (Top K)；
3. 在第三个 MapReduce 过程（汇总） Reduce 时输出的最高支持度模式串的数量 `K2` (Top K)。

其中 `I` 会影响后续建立的 FP-Tree 的数量和大小， `I` 越大，后续建立的 FP-Tree 就越少，而每一棵树会更大。 `K1` 会影响第三个 MapReduce 需要处理的数据规模。 而 `K2` 仅仅会影响输出的频繁模式条数。

从直觉上来说， `I` 的选取应该会对性能有较大影响，但是具体如何影响还需要观察； `K1` 越小性能应该越好（然而准确度应该会下降），但是由于第三个 MapReduce 计算量本来就不大，对于性能的影响应该不会太大； `K2` 不会影响到计算量，仅仅会影响 IO 开销，所以性能影响应该也不会太大。

除了算法的超参数外， Hadoop 的 MapReduce 中也有一些可以调整的设置，最直接的一个就是对于 Reducers 数目的指定。我配置的集群中共有 4 个可用的 8 核计算节点，也就是说共有 32 个计算单元，在 Hadoop 中可以分配出最多 32 个容器。按照 Hadoop 文档中的描述，推荐的 Reducers 个数为 0.95或$1.75 \times sum($节点数 * 节点的容器数$)$

下面我将对上面提到的参数和设置分别进行运行测试，对他们的性能进行比较分析。

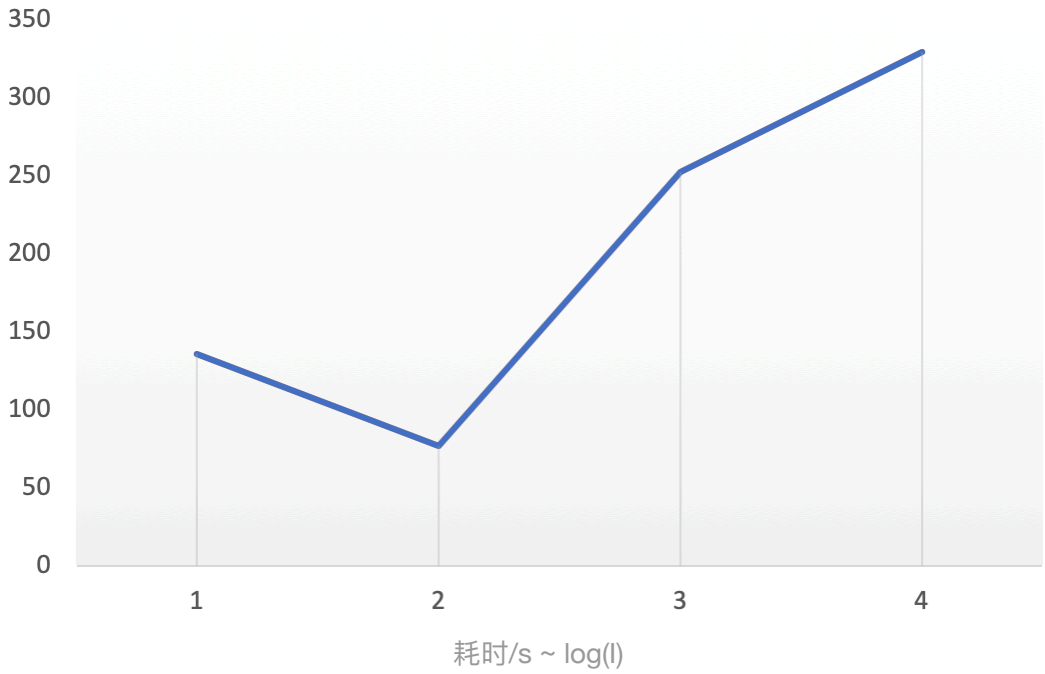
由于仅对性能进行测试，数据集仅选用 `20060801` 一天的数据。在对参数进行测试时，Reducers 数目设为 56 。

6.1 分组大小 `I` 对于性能的影响

对于选取的这一天的数据集，项目共有 `71792` 个。我选取了 $|I| \in \{10, 100, 1000, 10000\}$ 四个值进行测试，主要观察其对第二个 MapReduce 过程性能的影响。

在不同的 `I` 下第二个 MapReduce 的耗时分别为：

I	开始时间	完成时间	耗时/s
10	19/12/25 04:51:12	19/12/25 04:53:28	136
100	19/12/25 04:53:41	19/12/25 04:54:57	76
1000	19/12/25 06:15:12	19/12/25 06:19:25	253
10000	19/12/25 04:55:11	19/12/25 05:00:40	329



可见 `I` **过小或过大都不好**，这是因为 `I` 过小时会有太多的树，造成额外的调度开销与计算资源的浪费；而 `I` 过大时，每一棵树的规模会太大，导致每一个节点的计算效率下降。并且可以看出**存在那么一个最合适的** `I` ，这个 `I` 应该是由数据量、计算复杂性和集群中可分配计算资源三者共同决定的（其实按照直觉我有点怀疑这个值会在总项数的平方根附近）。

6.2 并行 FP-Growth 时 Top K 的 `K` 对于性能的影响

选取 $K \in \{1, 2, 5, 10, 20, 50, 100\}$ 进行测试，主要观察其对第三个 MapReduce 过程性能的影响，顺便证明仅影响 IO 时对于整体性能影响不大。

在不同的 `K` 下第二个 MapReduce 的耗时分别为：

K	开始时间	完成时间	耗时/s
1	19/12/25 06:35:52	19/12/25 06:37:19	87
2	19/12/25 06:37:37	19/12/25 06:39:36	119
5	19/12/25 06:39:46	19/12/25 06:40:57	71
10	19/12/25 06:41:13	19/12/25 06:43:12	119
20	19/12/25 06:43:23	19/12/25 06:44:31	68
50	19/12/25 06:44:42	19/12/25 06:45:50	68
100	19/12/25 06:46:01	19/12/25 06:47:16	75

和理论相符， `K` 的大小在 IO 时对整体性能的影响并不大（其中两个较大的值可能是由于有节点挂掉了）。

在不同的 `K` 下第三个 MapReduce 的耗时分别为：

K	开始时间	完成时间	耗时/s
1	19/12/25 07:26:28	19/12/25 07:33:31	423
2	19/12/25 07:33:45	19/12/25 07:35:33	108
5	19/12/25 07:35:55	19/12/25 07:37:10	75
10	19/12/25 07:37:21	19/12/25 07:38:38	77
20	19/12/25 07:38:54	19/12/25 07:40:10	76
50	19/12/25 07:40:26	19/12/25 07:41:37	71
100	19/12/25 07:41:49	19/12/25 07:43:08	79

说实话，这个结果是出乎我的意料的，照理来说 `K` 越大的话 MR2 的输出应该会越多，所以 MR3 的计算耗时也应该相应增多，但是测试结果却与预想中截然不同。我怀疑是不是集群中出现了什么意外导致了前两组测试的异常，但是因为时间有限我没能重新进行测试。

6.3 汇总输出时 Top K 的 `K` 对于性能的影响

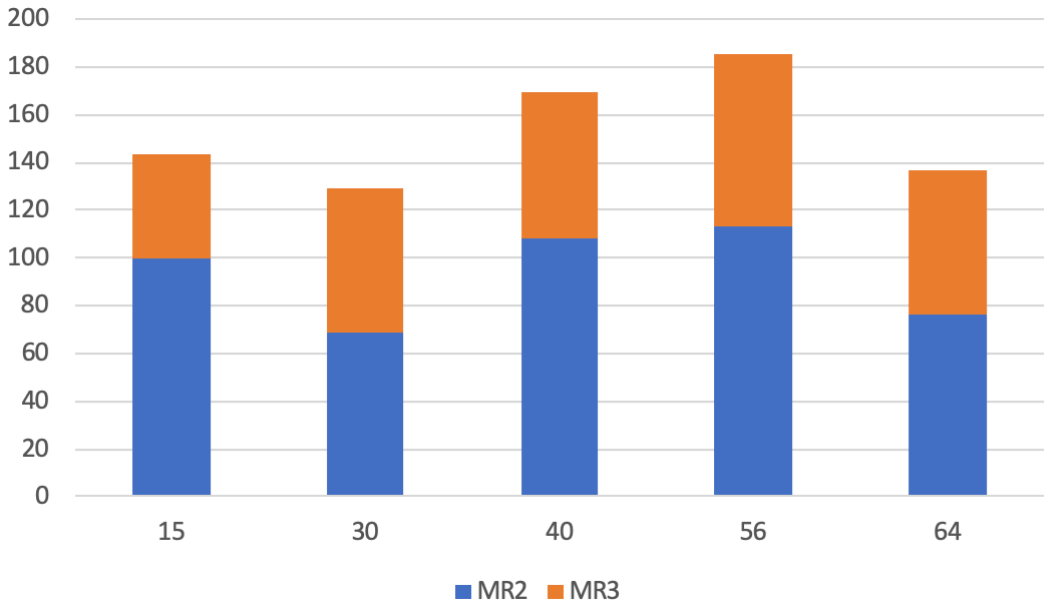
这一步仅仅影响最后输出时的 IO ，并不会影响事实上的计算量，这点在上一节已经得到验证，所以我认为没有必要进行测试对比了。

6.4 集群中 Reducers 数量对于性能的影响

Hadoop 官方推荐的 Reducers 数量为 0.95或$1.75 \times sum($节点数 * 节点的容器数$)$，在这里即 30 或 56，所以我分别取小于、介于、大于它们的三个值（15, 40, 64）加上它们进行测试。

保持其他所有参数不变（ `I = 100, K1 = 10, K2 = 10` ），仅改变 Reducers 的个数，耗时(s)对比如下（仅对 MR2 和 MR3 进行测试）：

MapReduce过程 / Reducers个数	15	30	40	56	64
MR2	100	69	108	113	76
MR3	43	60	61	72	61



不同 Reducers 数量的 MapReduce 过程耗时

其中 MR3 的输入文件均使用 56 个 Reducers 时 MR2 的输出文件。

一看结果，嗯？ 30 的确是耗时最少的没错，但是为啥 56 反而是耗时最多的？我再仔细读了一下文档中对于这两个值的描述：

> With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing. Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.

原来采用 56 时并不是性能最好，而是**性能最稳定**，而 30 才是**性能最好**的选择，这与我的测试结果是相符的。

在了解到这点后，我又仔细观察了下日志，果然发现在 Reducers 为 30 个时出现了失败的 Task:

```
19/12/25 09:19:44 INFO mapreduce.Job:  map 45% reduce 2%
19/12/25 09:19:45 INFO mapreduce.Job: Task Id :
attempt_1577267356940_0030_r_000009_0, Status : FAILED
19/12/25 09:19:51 INFO mapreduce.Job: Task Id :
attempt_1577267356940_0030_r_000004_0, Status : FAILED
19/12/25 09:19:51 INFO mapreduce.Job: Task Id :
attempt_1577267356940_0030_r_000011_0, Status : FAILED
19/12/25 09:19:53 INFO mapreduce.Job:  map 46% reduce 2%
```

七、挫折与解决方式

7.1 集群配置

集群配置过程中遇到了不少无法启动各个组件之类的错误，通过查看错误报告 + 上网搜索 + 询问助教最终都得到了解决。遇到的一些具体问题及解决方式如下：

1. Zookeeper 和 Hadoop 的配置文件中路径名无法被搜索到：

   这是因为我在配置文件中使用了用户目录 `~` 符号，而配置文件中要求使用绝对路径。

2. HBase 启动报错：`[main] regionserver.HRegionServer: Failed construction RegionServer` ：

   这个报错让我发现了一系列问题，总结下来主要分为两类：

   - 从节点的环境变量与主节点不一致：这是我配置时的疏忽，所以一定要保证集群环境配置中主从节点的一致！
   - HBase 无法找到一些库文件：将 `$HBASE_HOME/lib/client-facing-thirdparty` 和 `.../hadoop/share/hadoop/tools/lib/` 中的库文件拷贝到 `$HBASE_HOME/lib/` 下即可。

3. HBase 启动后部分子节点很快就挂掉：

   这个过程没有任何报错，日志中也没有显式地指出问题，但是我发现子节点和主节点日志里的时间戳是对不上的，用 `date` 命令查看本机时间，发现挂掉的从节点的时间果然与主节点对不上！所以要进行时间同步。

   然而服务器并没有办法直接访问外界的时间同步服务器，所以最后我仅仅让所有从节点的时间与主节点进行内网同步，具体做法参考了这篇博文：https://blog.csdn.net/a18838964650/article/details/86084790

4. 重启 Hadoop 时会报错，无法启动 DataNode：`org.apache.hadoop.hdfs.server.datanode.DataNode: Block pool ID needed, but service not yet registered with NN` ：

   最后发现可能是文件系统日志与实际情况不同步造成的，将 `.../hadoop/hdfs/data/current` 备份或删除即可。

7.2 运行测试

1. 内存溢出，包括虚拟内存和物理内存：

   以虚拟内存为例，会报错：

```
19/12/23 04:19:39 INFO mapreduce.Job: Task Id :
attempt_1576823962294_0009_r_000000_2, Status : FAILED
Container [pid=32142,containerID=container_1576823962294_0009_01_000037] is
running beyond virtual memory limits. Current usage: 304.9 MB of 1 GB physical
memory used; 2.1 GB of 2.1 GB virtual memory used. Killing container.
```

在确认服务器的实际内存完全够用后，我决定暂时解除 yarn 的内存限制，在 `yarn-site.xml` 配置文件中加入两个字段：

```
<property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>
<property>
    <name>yarn.nodemanager.pmem-check-enabled </name>
    <value>false</value>
</property>
```

2. Python 脚本放到 Hadoop 中无法运行：

   主要确保四点：

   - 代码正确。最好在本地通过测试；
   - 通过 `chmod +x` 给予了执行权限；
   - 文件头部有 `#!/usr/bin/python3` ，使得 Python 脚本可以直接运行；
   - 文件头部有 `# -*- coding:utf-8 -*-` ，以确保能够正确识别代码文件。

3. Reduce 进度卡在 67% 或 89%：

   经过在网上多方查找问题原因，我了解到 Hadoop 的 Reduce 进度其实分为三个阶段： Shuffle、Sort 和真正 Reduce 。所以卡在 67% 说明是真正在 Reduce 的时候出现的问题，这种卡住却不报错往往有两种原因：(1) 输入过大或数据不平衡，以至于计算缓慢 (2) 死循环。

   由于我在本地已经测试过了我的 Reduce 程序，并没有死循环的情况，所以我认为原因应该是前者。在把数据集规模缩减到三天而非三十一天后，进度条就能正常增长了。

   89% 是在有三个 Reducer 时出现的，其实原因和上面类似，只不过是仅卡住了一个节点。

八、总结与体会

事实上比起写算法与跑测试，我在这次课题中更多的时间恐怕都用在了对于集群计算的探索上，包括集群的部署与如何使用它跑程序。

对于集群部署，我发现其实对于 Slave 节点的配置流程基本都是完全相同的，但是一个一个去复制配置仍然非常令人疲惫，而且稍有不慎漏掉某一个步骤就会导致该节点无法正常工作。对于这个问题，我相信在真正的集群部署工作中应当是有自动化脚本可以完成对于集群节点增删改的操作的（甚至如果给我足够时间我应该都有能力写出来）。

至于在集群上跑程序这方面，我感觉我遇到的很多问题的根本原因是没有用 Java 而是 Python 编写程序。因为 Hadoop 是用 Java 实现的，它原生支持的 MapReduce 程序本该由 Java 编写，但是我对于 Java 并不熟悉，所以决定采用我比较熟悉的 Python 写程序，借由 Hadoop Streaming 的支持来运行。这种做法带来的最大问题就是无法在集群上直接看到由程序导致的报错信息！对于程序员来说，DEBUG 本来就是一件劳心劳力的事，更何况没有报错信息的 BUG …… 所以每次发生错误，我都不得不在本地模拟 MapReduce 的过程手动跑一遍程序，还得自己实现 Shuffle 和 Sort 的过程。所幸通过这种笨办法也算是把所有遇到的问题都解决了。但如果再让我写一个集群计算的程序，我宁可先去把 Java 入门了也不会再选择用其他语言来写了。

然而在这次课程项目中，比起课题与算法本身，我的感受其实更深刻地集中在部署分布式集群来进行大数据计算这一方面上。

在完成课程项目的过程中，我自己动手部署了一个小型的分布式计算系统，并且根据算法论文一边探索一边试错地写出了能够在集群上进行运算的程序。这样大的数据量在我自己的电脑上几乎完全跑不动，但是在集群上利用 MapReduce 却能够在可接受的时间内得到想要的结果，让我深切体会到了集群计算对于大数据处理的重要性。

在亲手实现这一切之后，我对于 MapReduce 这一框架有了远比理论知识更深刻的理解，比如体会到了 Shuffle 在其中的作用等。我还有了基本的集群搭建经验，对于集群的工作原理也有了粗浅的了解，如果未来有要对这一领域的知识进一步了解的需求，我也能够大致知道往哪个方向学习了。

不知道是不是巴德尔-迈因霍夫现象（指当了解到一个新知识，比如新单词、新概念等，不久之后就会有机会再一次遇到它）的缘故，我原本仅仅把这门课当做科普性质的知识拓展，但是这学期却频频在关注实习岗位与前沿公司的产品架构时接触到这门课中介绍过的一些东西，比如 Hadoop 和 Spark 等，我才渐渐意识到这门课中学到的东西很可能会在我未来的工作中派上用场。人们常说学习的最好方式就是结合理论与实践，这一次项目中的实践就算是为我的这趟大数据入门之旅划上了完美的句点。

九、附录

9.1 文件结构说明

```
./
|
+-- data/
|    |
|    +---- raw/ # 原始数据集文件目录
|    |       |
|    |       +-- SogouQ/ # 搜狗数据集原始文件目录
|    |
|    +---- temp/ # 临时文件目录，用于存放转码后的数据文件
|    |
|    +---- clean/ # 存放清洗后的数据文件
|    |
|    +---- BD_jieba/ # 存放关键词提取的结果，这也是第一个 MapReduce 的输入文件目录
|    |
|    +---- result/ # 存放分组结果，即 F-list 和 G-list
|
+-- log/ # 预处理输出日志目录
|
+-- env/ # 存放集群配置中用到的安装包
|
+-- doc/ # 草稿、文档等
|
+-- result/ # 最终输出结果
|
+-- src/ # 源代码目录
     |
     +-- preprocess/ # 预处理代码目录
     |        |
     |        ... # 见下一节
     |
     +-- demo/ # 关键词挖掘效果测试文件目录
```

```
|     |
|     ... # 见下一节
|
... # 见下一节
```

9.2 源代码文件说明

预处理部分 - `src/preprocess/`

- `utils.py`：包含工具函数，主要用于输出日志；
- `gb2utf8.py`：将原始数据由 `GB18030` 转码为 `UTF-8`，并删去无法识别的记录；
- `format_file_v2.py`：清洗数据集；
- `to_db_jieba.py`：从查询词中提取关键词；

算法部分 - `src/`

- `mapper1.py`，`reducer1.py`：并行计数的 MapReduce 实现；
- `sort_kv.py`：项目分组的实现。输入 MR1 的输出结果，得到 F-list 和 G-list；
- `mapper2.py`，`reducer2.py`：并行 FP-Growth 算法的 MapReduce 实现；
- `mapper3.py`，`reducer3.py`：汇总的 MapReduce 实现。

Demo部分 - `src/demo/`

- `combine_parts.py`：将 MR3 输出的所有 `part-*` 文件合并到一个文件中；
- `find_pair.py`：效果 Demo。输入关键词，将输出其所有挖掘到的关联模式。

9.3 成员分工

该项目由我独立完成。

9.4 参考文献

[1] 搜狗数据集： Yiqun Liu, Junwei Miao, Min Zhang, Shaoping Ma, Liyun Ru. How Do Users Describe Their Information Need: Query Recommendation based on Snippet Click Model. Expert Systems With Applications. 38(11): 13847-13856, 2011. [链接]

[2] PFP算法论文： Li, Haoyuan, et al. "Pfp: parallel fp-growth for query recommendation." Proceedings of the 2008 ACM conference on Recommender systems. ACM, 2008. [链接]

[3] 集群搭建： 从零搭建Hadoop+Zookeeper+HBase完全分布式集群 - 复旦猿 [链接]