**课程设计1-个性化图书兴趣分析与推荐系统**

**任务一：数据预处理**

**T1\_Books\_Simplified:**

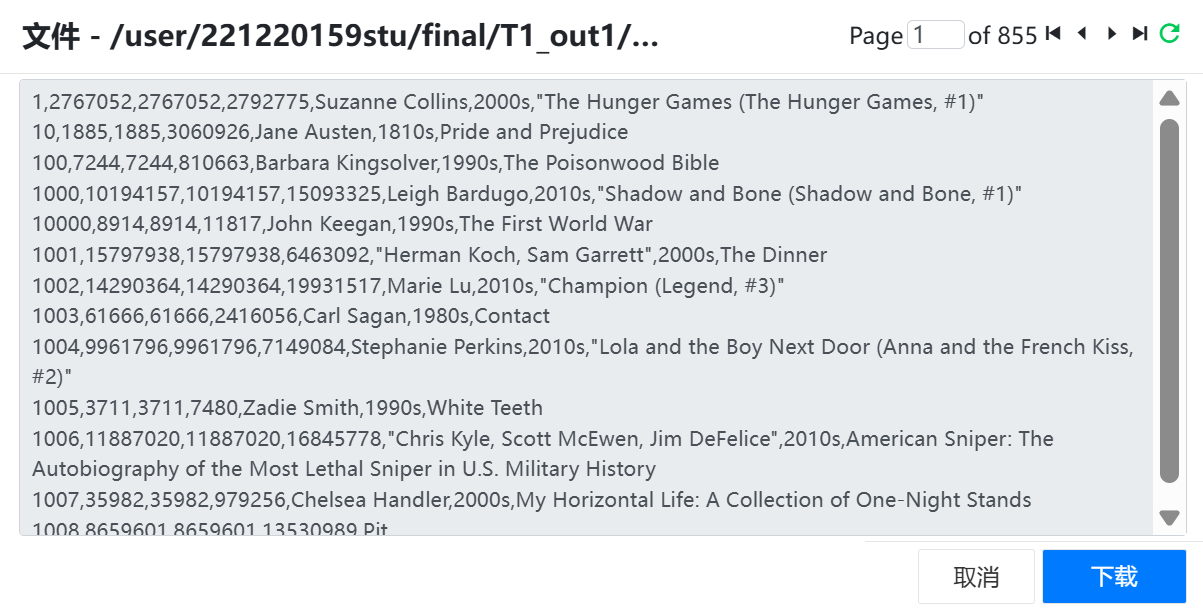
实现了日期空值替换为1900s，所有日期转换为xxx0s格式，并输出按指定列进行提取后的Text。

**输入**：books.csv

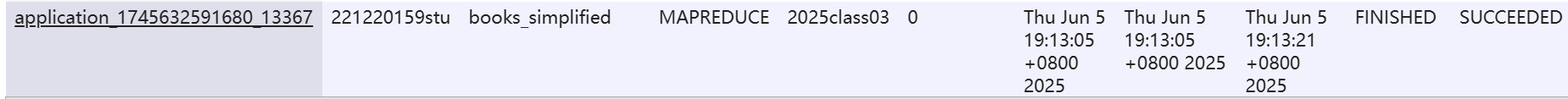
**输出**：books\_simplified，格式为(book\_id,goodreads\_book\_id,best\_book\_id,work\_id,authors,original\_publication\_decade,title)

**mapper**输出键为Text(最后结果)，值为NullWritable，没有**reducer**

**运行结果**：



**平台运行记录**：



**T1\_Book\_Tags\_Flattened：**

计算图书被打的所有标签及次数

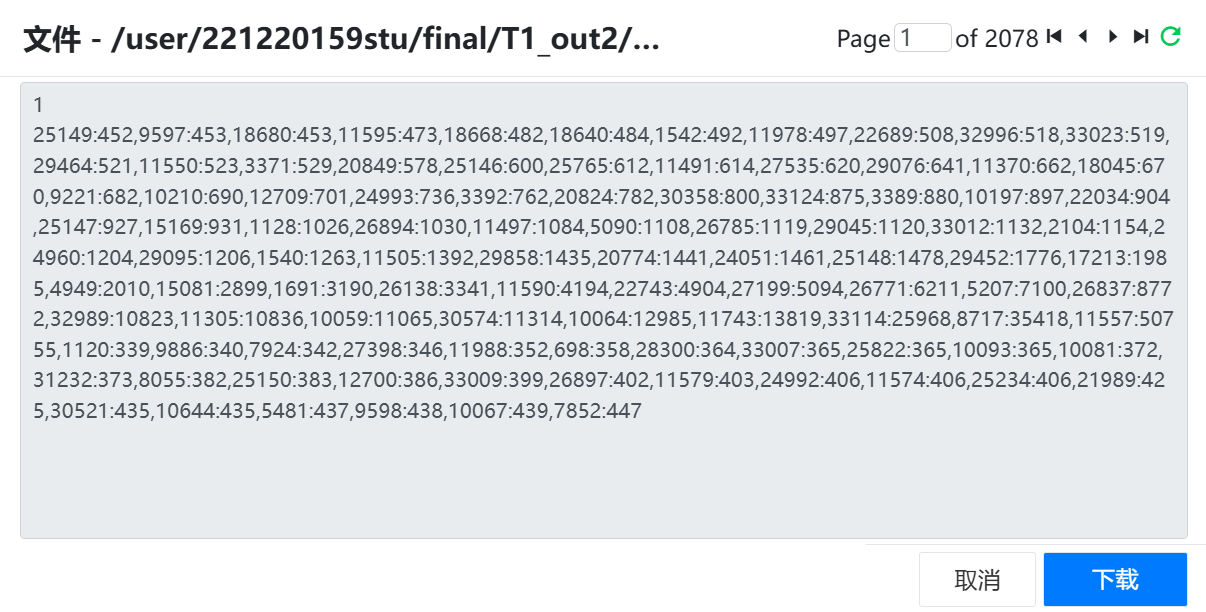
**输入**：book\_tags.csv、books\_simplified(cache)

**输出**：book\_tags\_flattened，格式为(book\_id \t {tag\_id1}:{num1},{tag\_id2}:{num2}…)，例如"1  2:3,4:5,6:1"

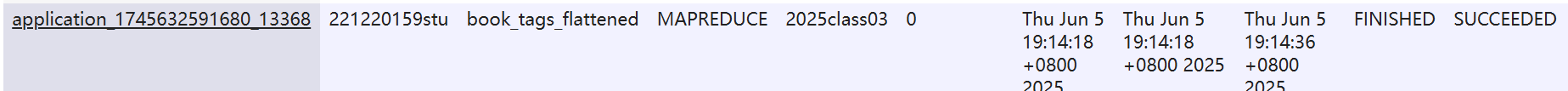
在**map**方法中把读入的book\_tags.csv的文件中的good\_reads\_book\_id替换为其对应的所有book\_id，mapper输出的键为book\_id,值为tag\_id:tag\_cnt形式的字符串。同一个键(book\_id)的数据被发送到同一个**reducer**拼接成最终输出的字符串。

**优化**：为了提高效率设置了**combiner**，combiner可以直接使用reducer的类，将同一个键对应的tag\_id:tag\_cnt形式的字符串拼接起来减少数据传输过程的压力（shuffle）。

**运行结果**：



**平台运行记录**：



**任务二：图书相似度计算**

**PartA**

**T2\_User\_Cnt：**

得到每个用户评过分的所有书，并统计所有的用户数量（相似度计算中会用到）

**输入**：ratings.csv

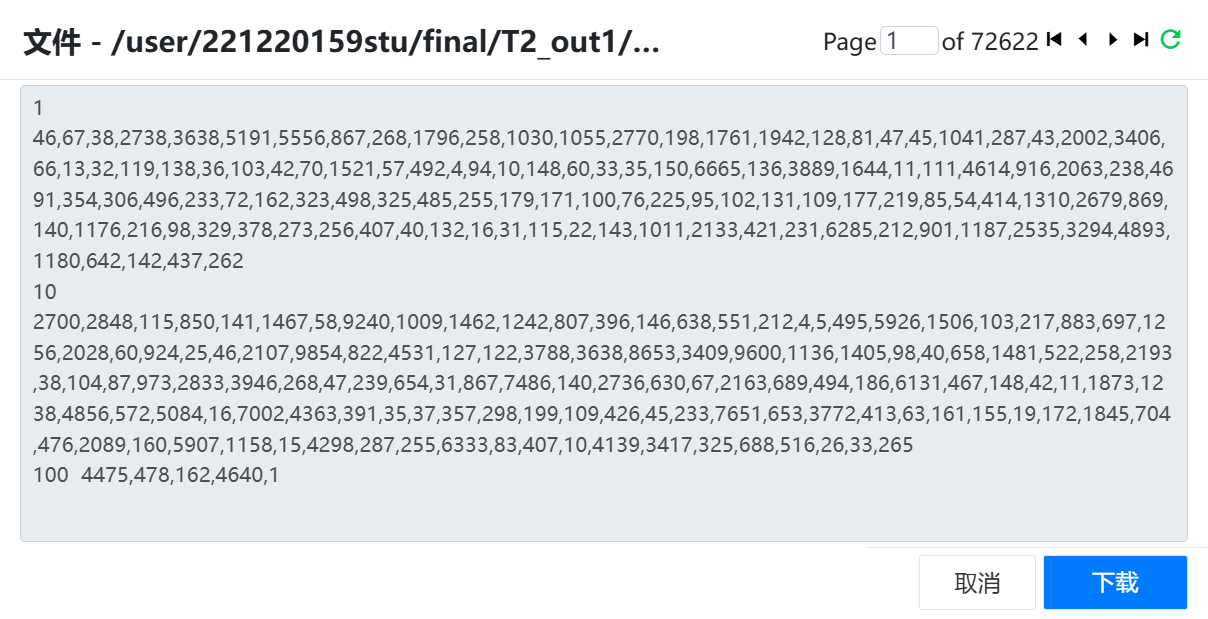
**输出**：每个用户评价过的书的列表、用户数量

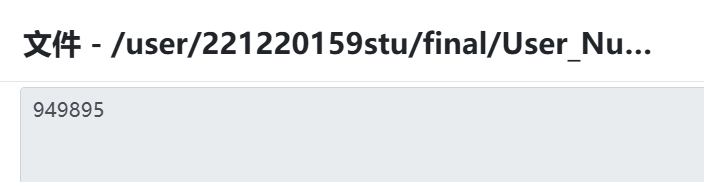
**Mapper**输出键为Text代表user\_id，值为Text代表该用户评价的book\_id。

**Reducer**的输出键为Text代表user\_id，值为Text代表该用户评价过的所有book\_id通过逗号连接起来的字符串，形如book\_id0,book\_id1…

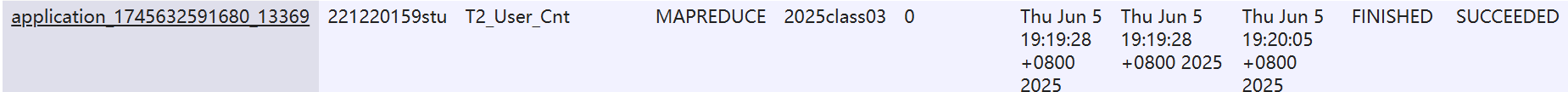
并且还有一个计数器public enum User\_Counter{USER\_NUM},统计所有的用户数目,最后把这个计数写入第二个输出目录(949895个用户)。

**运行结果：**

****

****

**平台运行过程：**



**T2\_Book\_Pair:**

在任一用户的评分历史中共现的图书对，统计该共现图书对在所有用户的各自评分历史中的总共现次数

**输入**：T2\_User\_Cnt的输出（用户id和该用户评价过的所有图书）

**输出**：图书对及出现次数，格式为((book\_idi,book\_idj),总共现次数)，小号在前

**mapper**输入的格式为“user\_id \t book\_id0,book\_id1…”

内部通过双重循环输出所有共现图书对，同时为了避免重复，每个对只输出一次，

同时保证输出的对book\_id\_i,book\_id\_j一定满足i<j。

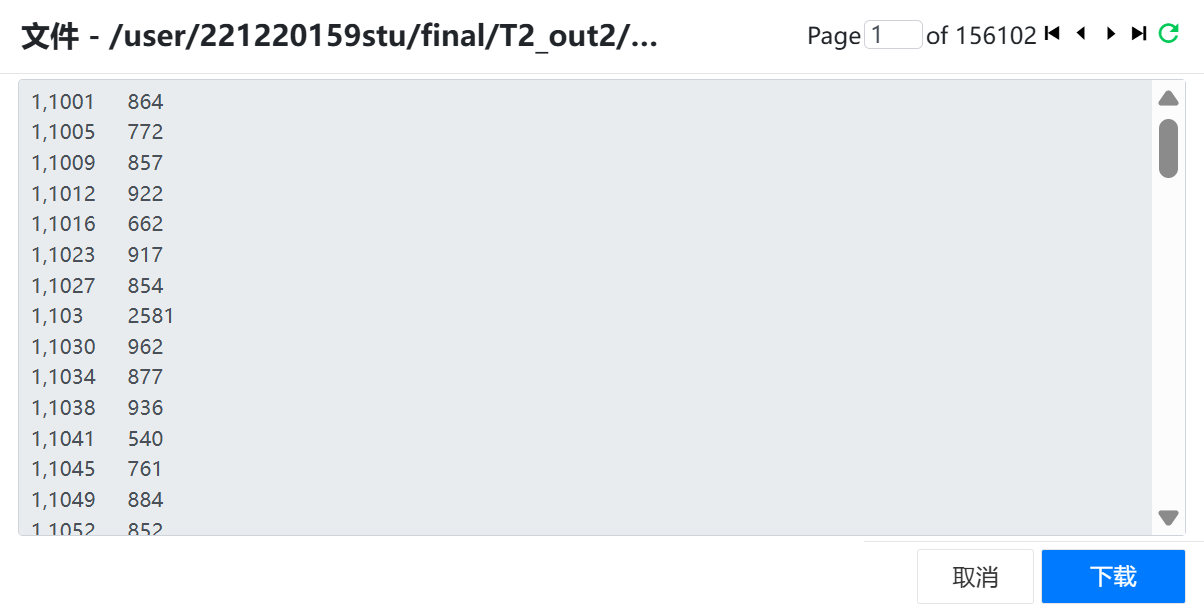
输出为键为Text形式的图书对，值为IntWritable形式的图书对出现次数

**reducer**对每个图书对的共现次数进行累加，与mapper输出格式和含义一致。

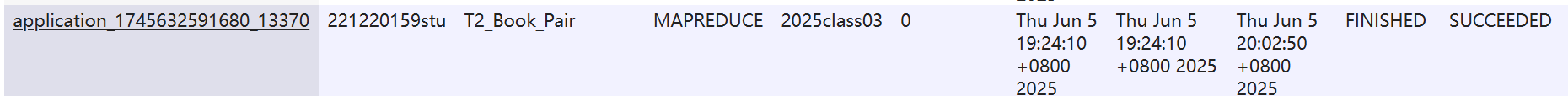
**优化**：为了降低mapper和reducer之间数据传输的代价使用combiner

（逻辑与reducer类完全一致）将本地输出的相同的键的值加在一起。

**运行结果：**

****

**平台运行记录：**



**PartB&C**

**T2\_Similarity**:

计算每个图书对的相似度

**输入**：book\_pair、books\_simplified(cache)、book\_tags\_flattened(cache)

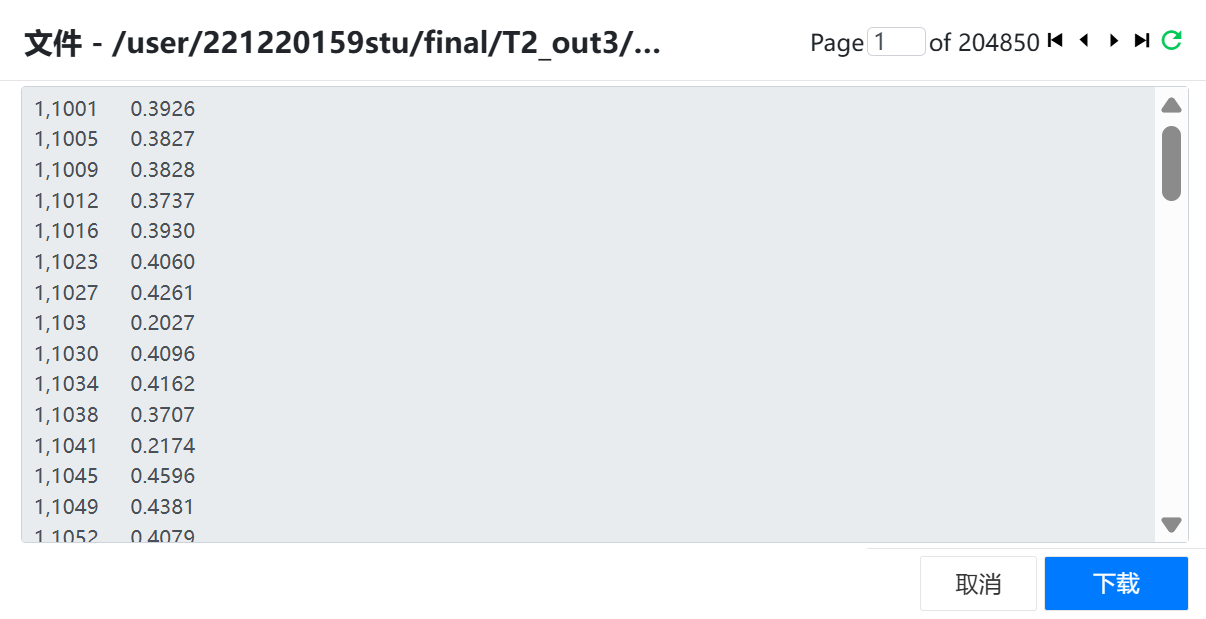
**输出**：(book\_id1,book\_id2), similarity(book\_id1,book\_id2)

观察到tags相似度需要单独计算，而其他的相似度可以在一起计算，我们可以利用**multipleInput**设置两类mapper分别计算tags相似度和其他的相似度之和，最后在reducer之中加在一起除以4即得到最终的相似度。

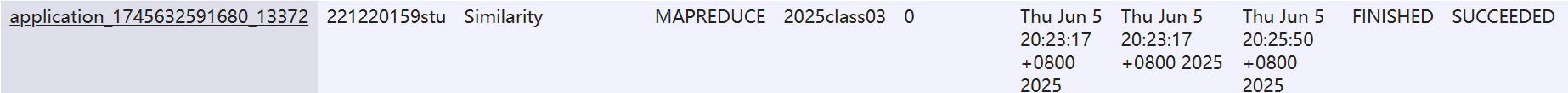
**TSMapper** 用于计算 tags 相似度，在mapper 的 **setup**方法中读取cache中的book\_tags\_flattened，并记录每个 book\_id对应的所有 tag\_id及该tag的数目。**map** 函数输入的键为 book\_pair 及其出现次数形式的字符串，形如book\_id0,book\_id1 cnt。对每一个输入的book\_pair 计算它们的tags相似度：它们重合tag的所有数目/它们所有tag的数量（重复的tag数目都计入，例如book1有10个tag A，10个tag B，book2有5个tag A，25个tag C,它们的相似度就是（10+5）/（10+10+5+25）=30%）。

另一个Mapper(**SMapper**)计算了除了tags相似度之外其他相似度的总和。从cahcedfiles中读取books\_simplified中的信息并在setup方法中记录下来，通过输入的book\_id可以找到相应的信息（作者，出版年份）。共现次数在输入的值中给出，总的用户数在T2\_User\_Cnt中给出，authors相似度计算方法与tags类似，不过一个author在一本书的记录中只可能出现一次，所以author不需要记录数目，使用map记录判断有无这个author即可。对于Years相似度，分析数据集数据可知，10000条数据里只有100条左右数据在1800年之前，同时最小值-1750（某篇史诗，这些数据经过查询都是正确且真实的），最大跨度要3900左右，这会导致按照PPT上公式计算出来的year similarity过于接近，没有较好的区分度，针对这个问题我们固定max\_year\_gap为2010-1800，如果计算出来的year\_similarity小于0就把它记作0。

**运行结果**：



**平台运行记录**：



**任务三：构建用户画像，形成用户聚类**

**任务3.1构建用户画像**

**T3\_User\_Profile**

为每位用户构建用户画像

**输入**：ratings.csv和to\_read.csv，books\_simplified、book\_tags\_flattened

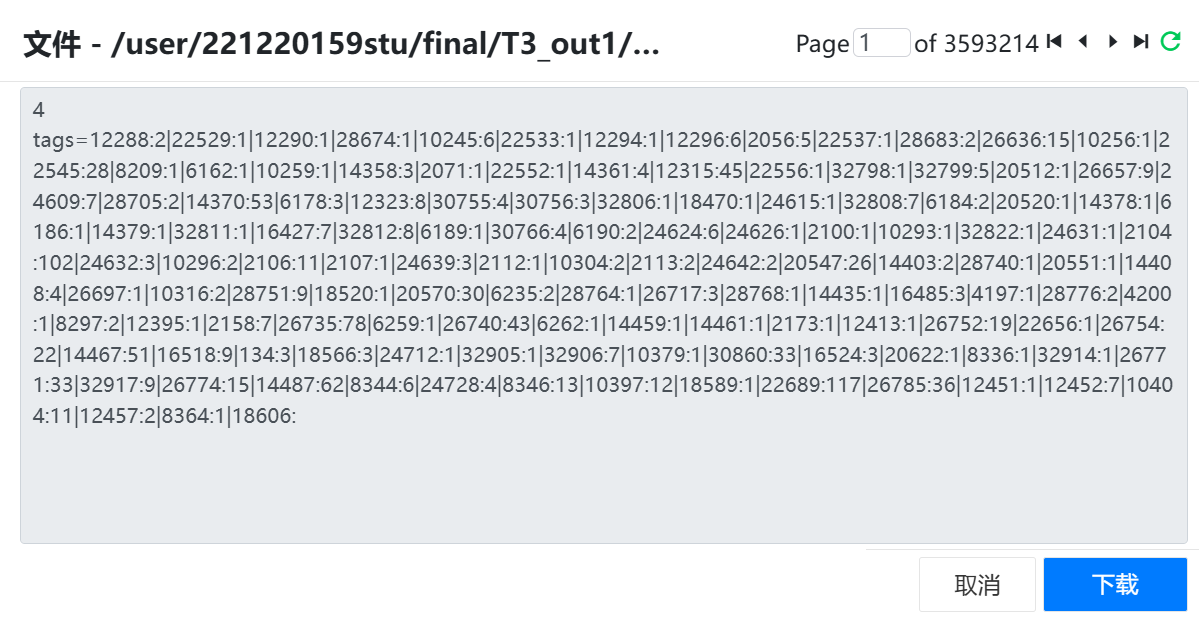
**输出**：user\_id \t tags={tag\_id1}:{t\_num1}|{tag\_id2}:{t\_num2}|…;authors={author1}:{a\_num1}|{author2}:{a\_num2}|…;years={year1}:{y\_num1}|{year2}:{y\_num2}|…

将books\_simplified、book\_tags\_flatted存入cachefile，**mapper**读入ratings.csv和to\_read.csv的数据，输出的键为user\_id，值为book\_id（类型均为IntWritable)，这样在reducer中就能得到一个user\_id相关的所有book\_id。

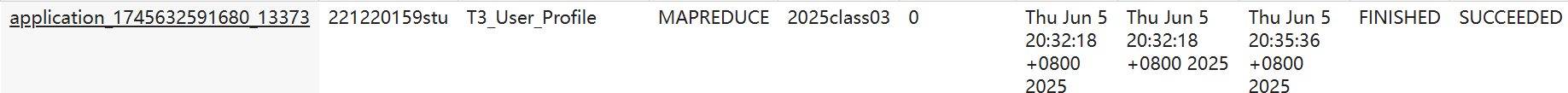
**reducer**中将cachefile中的文件读入并记录，根据book\_id查找该book\_id相关的信息并记录在tag\_map、year\_map、author\_map中，map中的值为出现次数，处理完所有book\_id后生成用户画像的信息字符串并输出。

**优化**：mapper只负责发送user\_id,book\_id对，具体的信息在reducer中存储，需要在reducer中才添加该book\_id相关的信息，这样做，相比于在mapper中就生成该book\_id相关信息的字符串输出并在reducer中重新解析，减少了传输的数据量以及减少了reducer输入的数据量和reducer字符串解析的工作量，提高了效率。

**运行结果：**



**平台运行记录：**



**任务3.2 形成用户聚类**

**T3\_User\_Feature**

为每位用户生成800维向量：512维标签，256维作者，32维年份

**输入**：user\_preference(T3\_User\_Profile的输出） tags.csv

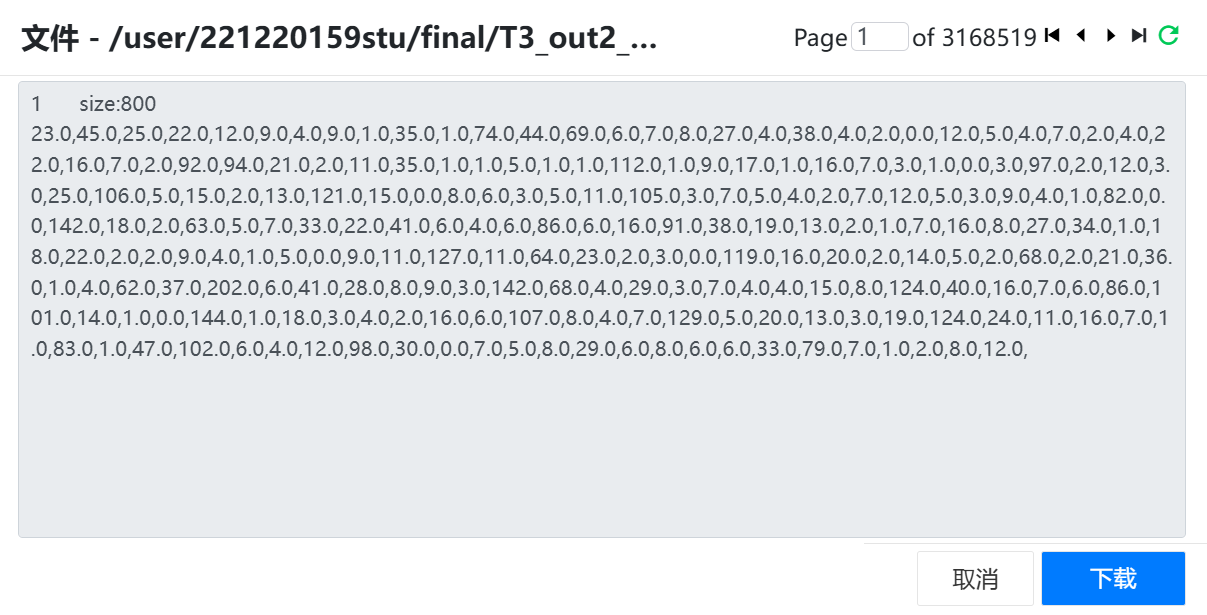
**输出**：序列化（IntWritable，VectorWritable）键值对user\_id user\_vector

在**mapper**的**setup**方法中，建立tag\_id到tag\_name的映射，**map**中对每个user\_id创建800维向量（512维标签，256维作者，32维年份），通过哈希映射将原始信息映射到相应维度，每个维度里面存储该维度信息出现的次数（tag/author/year次数，如果不同的信息映射到同一维则计数叠加）。onehot编码年份时，根据数据分布的特点，小于1500放一个维度，1500到1800放一个维度，之后的数据每10年一个维度。

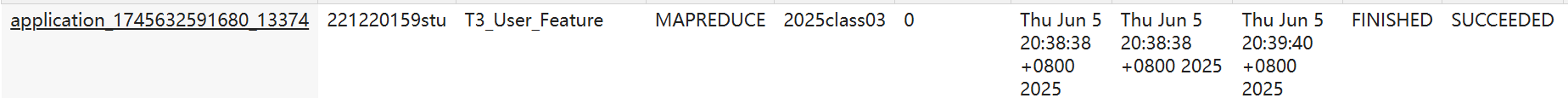
优化：采用**SequenceFile**输出序列化的文件，节省了处理和解析字符串的时间，提高了效率。通过实现VectorWritable类，实现readFields和write方法，并通过job.setOutputFormatClass(SequenceFileOutputFormat.class); job.setInputFormatClass(SequenceFileInputFormat.class);以实现序列化文件的读写。

**VectorWritable**由一个double数组和一个int变量cnt组成，读/写的时候会首先读/写数组当前的大小，然后按顺序读/写数组每一个元素，cnt为计数，便于后面计算向量的均值（计算向量均值的时候，可以直接叠加向量，最后输出当前叠加的结果，以及叠加的向量的数目，这样便于使用combiner进行优化）。为了避免重复申请空间导致内存溢出以及减少申请空间的开销，VecrorWritable对象只有在当前的length与读入的VectorWritable对象不一样时才会申请新的空间，否则会一直使用原来分配的内存空间。

**运行结果：**

****

**平台运行记录：**



**T3\_PCA\_Prepare**

计算所有向量的均值、向量平方的均值

**输入**： T3\_User\_Feature的输出

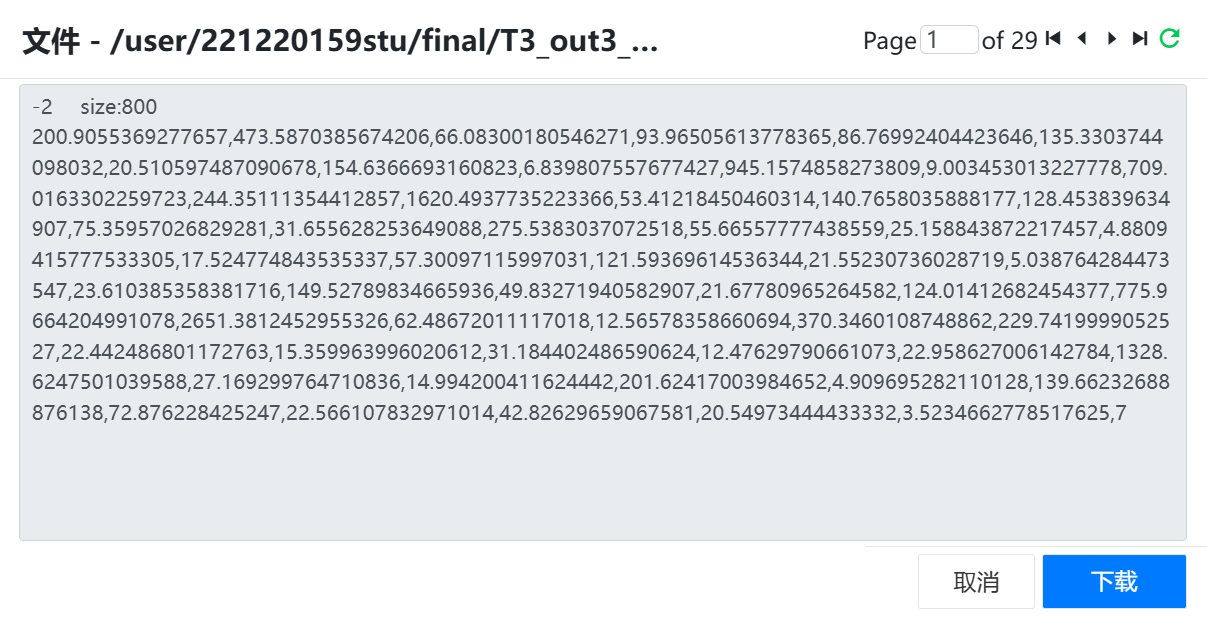
**输出**：<-1, mean\_vector> <-2, mean\_square\_vector>

**mapper**：将分配来的所有向量进行加和，进行平方后也进行加和，在cleanup中通过键-1/-2区分并发送到reducer

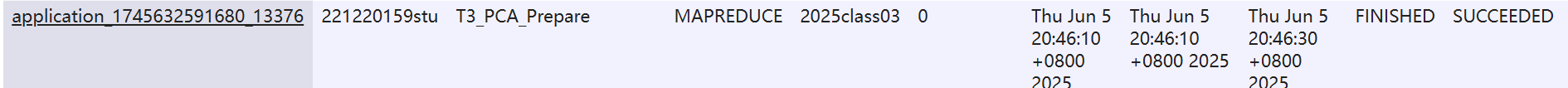
**reducer**：继续进行加和后，分别求评价并输出

优化：vectorwritable里面实现了向量的原地加法，以及向量平方后的原地加法，原地加法可以节省空间，并避免了再次分配空间的开销，同时维护cnt记录当前累加的向量数目，**mapper**里面只进行加法不输出，在所有mapper累加结束后在**cleanup**方法里面输出当前mapper的累加结果，并最终在**reducer**里面累加所有局部结果并计算均值，这样处理效率最高（使用combiner会在缓冲区满后溢写时进行combine的操作，需要再次读入数据，同时combiner不一定每次都会调用，导致最终combiner处理后输出的数据量要比mapper在cleanup输出开销大很多）。

**运行结果：**



**平台运行记录：**



**T3\_PCA\_Standardization:**

将向量减去均值再除以标准差进行标准化处理，标准差为平方的均值减去均值的平方再开方。

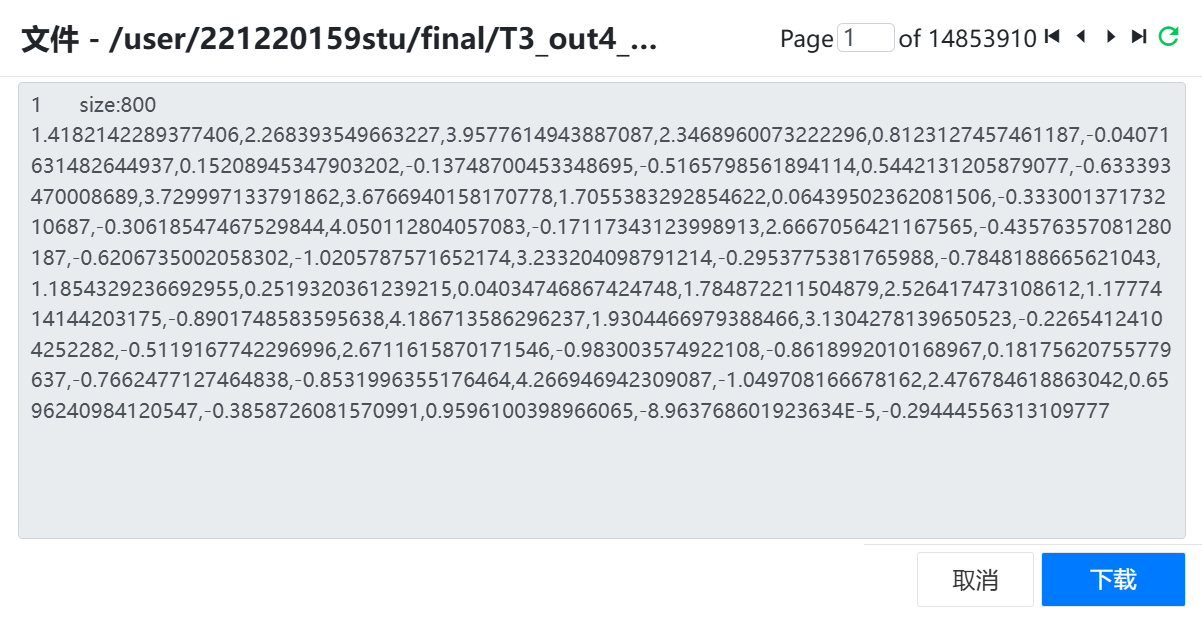
**输入**： T3\_User\_Feature的输出、T3\_PCA\_Prepare的输出

**输出**：标准化后的向量

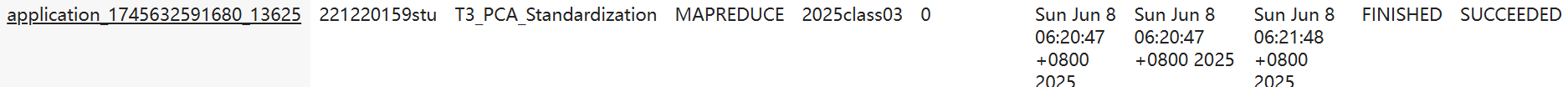
将T3\_PCA\_Prepare的结果加入cachefile并在**mapper**的**setup**方法中读入，调用VectorWritable的squareAndSubInPlace方法计算标准差。

**map**函数中，将向量减去均值再除以标准差，进行标准化处理。没有**reducer**。

**运行结果：**



**平台运行记录：**

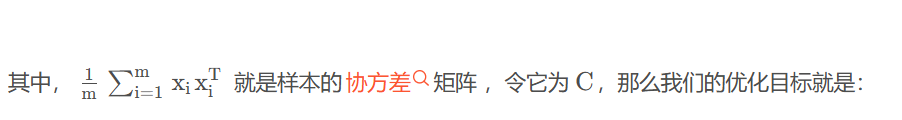


**T3\_PCA\_Covariance**

计算协方差矩阵

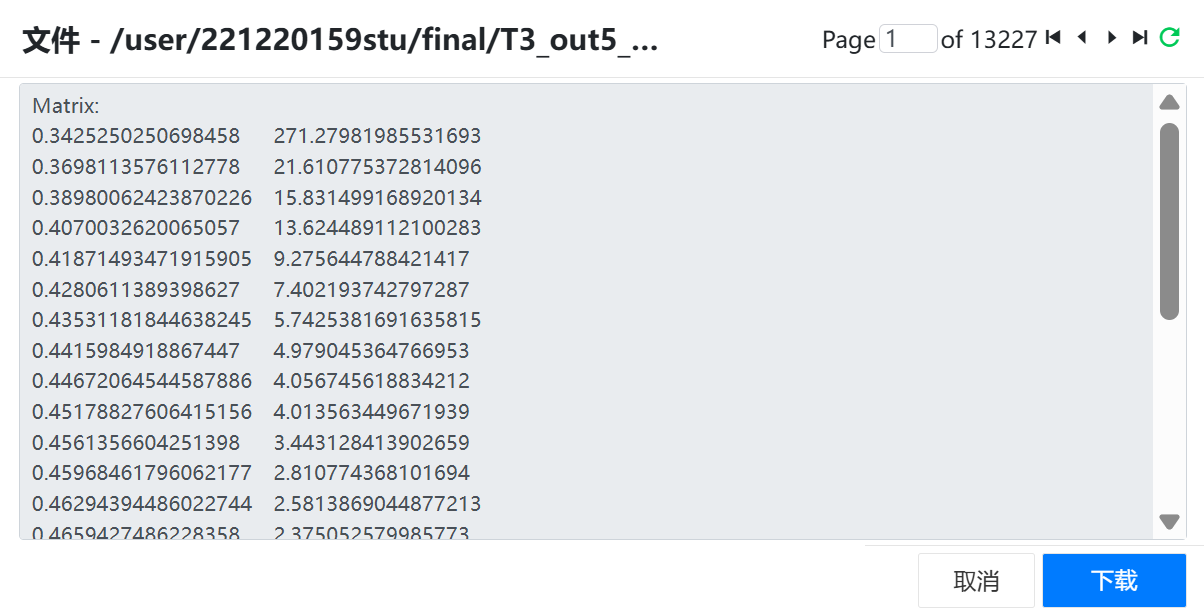
**输入**：T3\_PCA\_Standardization的输出

**输出**：协方差矩阵特征值分解后的前800个特征向量按照特征值从大到小的顺序排成的矩阵

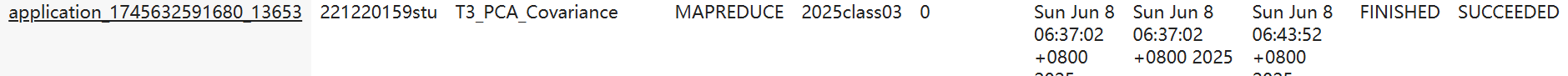


每一个向量乘以它的转置并将这个800\*800维矩阵累加起来，优化方法与计算均值一致，每个**mapper**里面累加所有矩阵，在**cleanup**方法中发送给reducer。最后对它进行特征值分解并且将特征值从大到小排序，再按照排序后特征值的顺序将相应的特征向量放入输出矩阵的一行。还在这个job中输出了排序后的特征值，以及大于等于这行的特征值之和占所有特征值之和的比率判断保留了多少信息。

**运行结果：**



**平台运行记录：**



**T3\_PCA\_Projection**

实现降维，指定参数k为降维的目标维数。

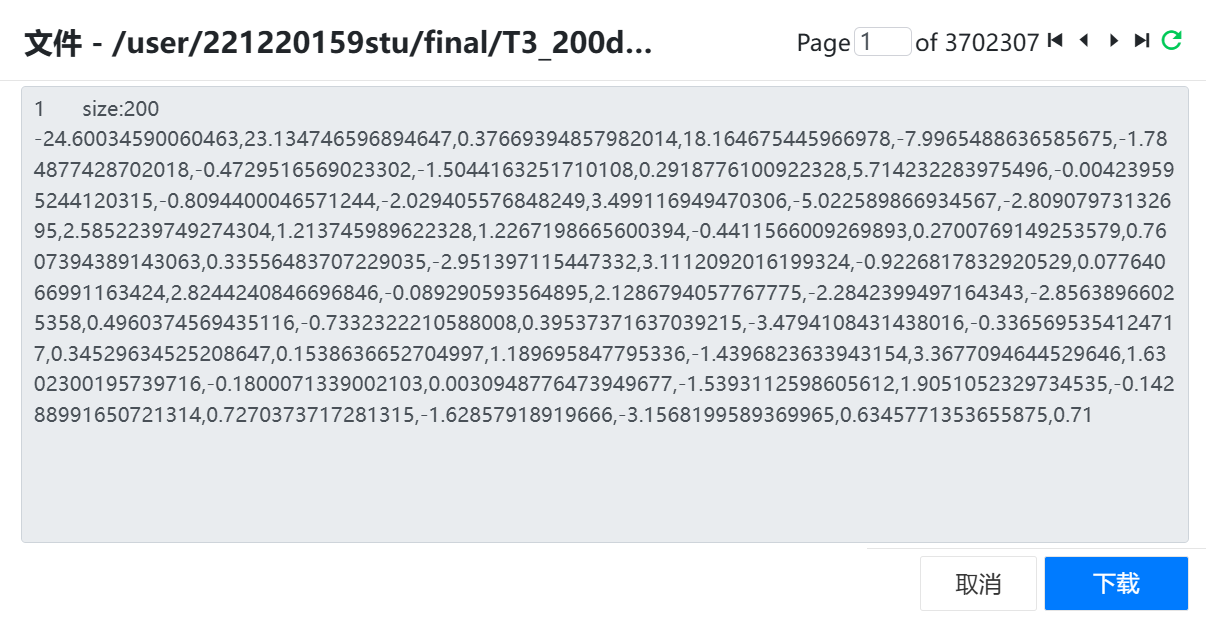
**输入**： T3\_PCA\_Covariance的输出的降维矩阵、T3\_User\_Feature输出的用户向量

**输出**：降维后的向量

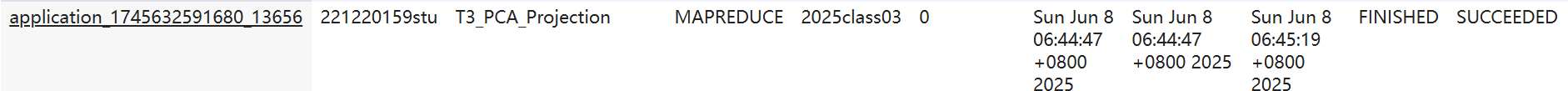
在mapper的setup方法中读入T3\_PCA\_Standardization输出的每一行都是一个特征向量的矩阵的前k行，用这个k\*n矩阵乘以n维向量则得到一个k维向量。

我们选择将800维向量降维200维，这样既可以保留较多的信息，又可以减轻计算的负担。同时实现了MatrixWritable类方便处理序列化矩阵文件的输入输出。

**运行结果：**



**平台运行记录：**



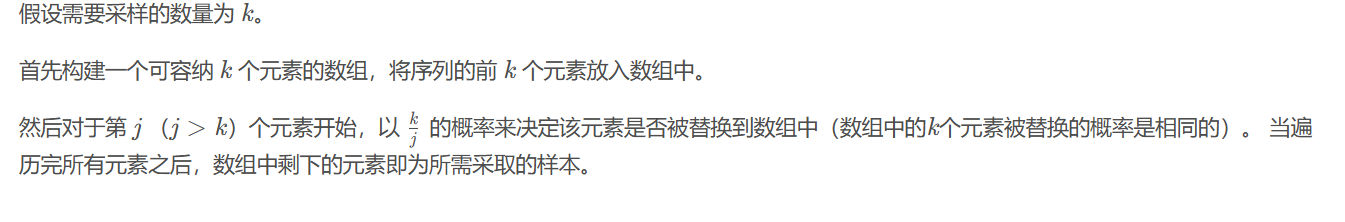
**T3\_Sampling**

随机采样100个向量作为初始中心，指定参数k为采样的数目

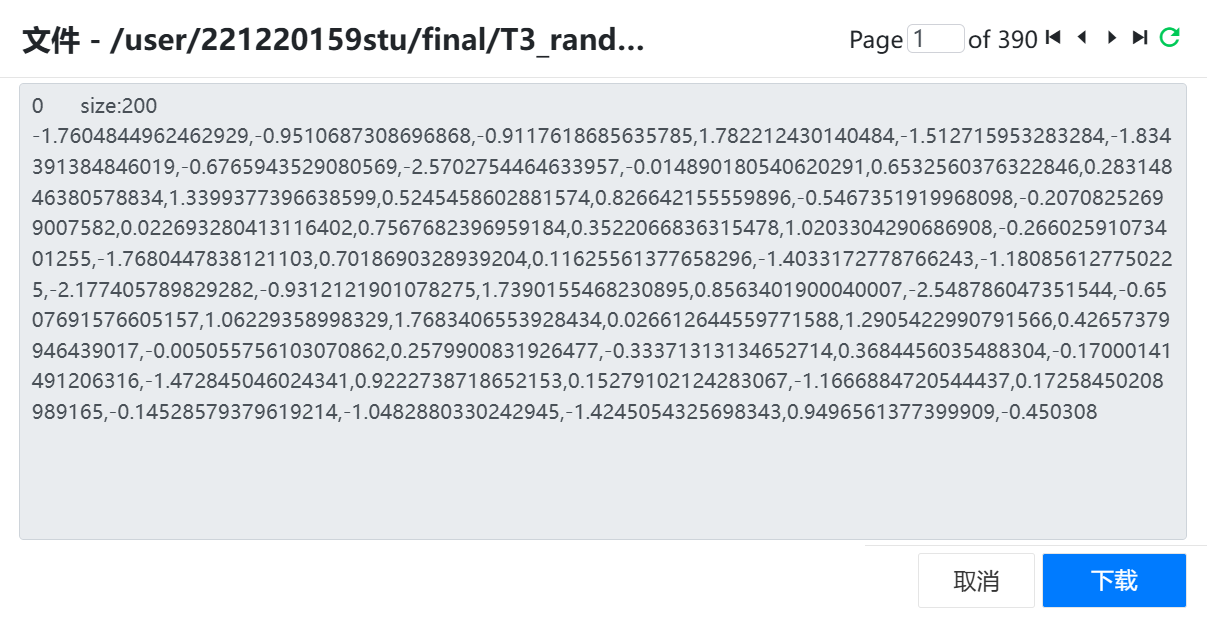
**输入**： T3\_PCA\_Projection输出的降维后的向量

**输出**：随机选择k个向量，此处k指定为100

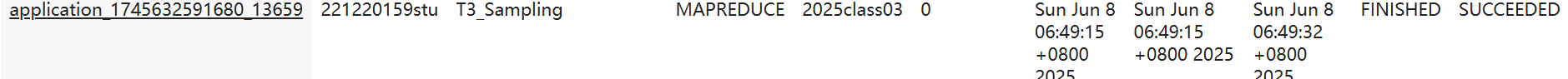
采用蓄水池采样方法选取，保证选中每个向量的概率均匀。每个mapper选取k个样本汇集到reducer中再选取k个样本。对于下标j生成一个小于j的随机数，如果j小于k则替换数组中该位置的向量。



**运行结果：**



**平台运行记录：**



**T3\_KMeans**

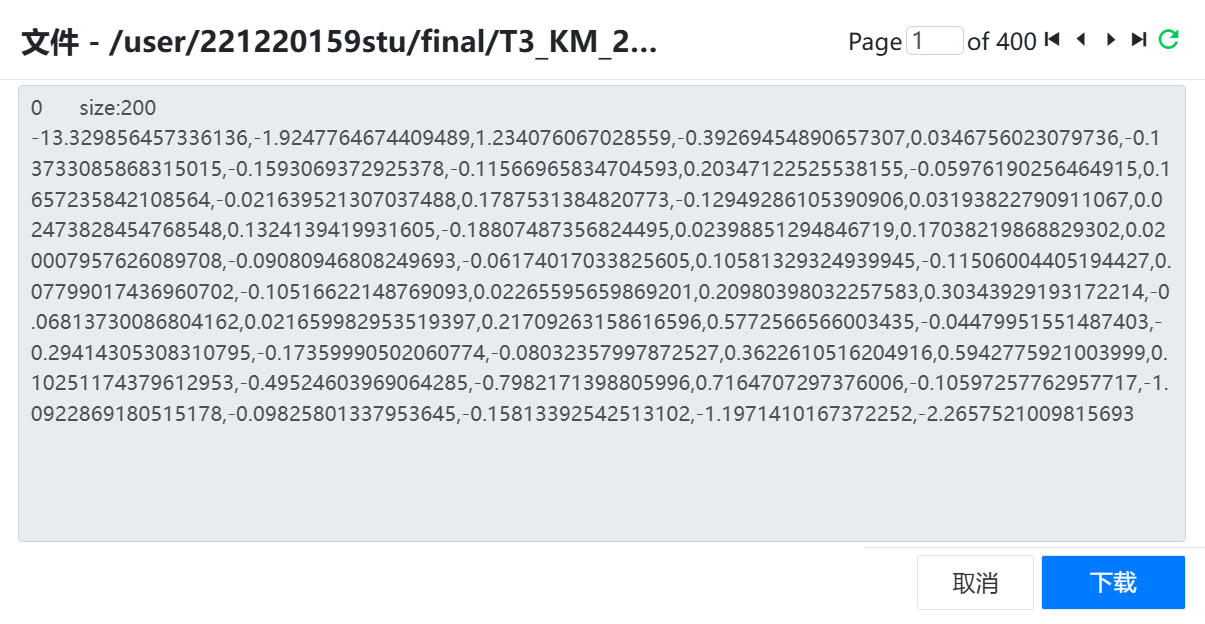
实现kmeans聚类。

**输入**：降维后的200维向量，随机选取的100个初始聚类中心向量。

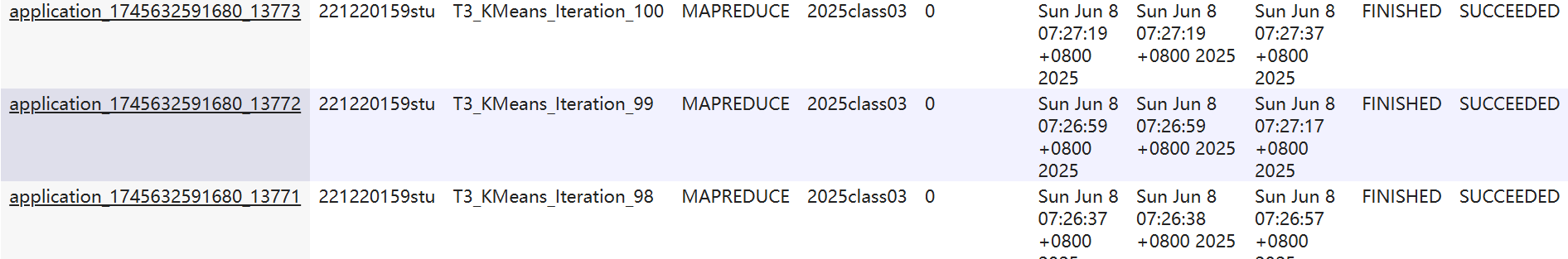
**输出**：100个聚类中心，(IntWritable,VectorWritable) 键值对 cluster\_id cluster\_center

采用一个combiner计算分配到相同cluster\_id的向量的和，并记录向量的数目，减少数据传输的代价。添加迭代终止条件：质心移动的最大距离小于0.1或者到达一百轮。

**运行结果：**



**平台运行记录：**



辅助类：**Read\_Matrix**类用于读取序列化的矩阵数据并将分散的数据合并为一个文本文件呈现。**Read\_Vector**类似，用于读取序列化的向量数据并转换为文本文件。这两个类使用时格式为:

project.jar Read\_Vector final/T3\_random\_200d final/T3\_random\_200d\_Text

**Cal\_Dis**类用于判断kmeans聚类的结果质量，它计算了两个相邻迭代的kmeans中心向量之间的距离，使用格式为：

project.jar Calc\_Dis final/T3\_KM\_200d\_1/iteration-30 final/T3\_KM\_200d\_1/iteration-31 final/temp/30

**T3\_User\_Cluster**

记录每个用户所属的cluster

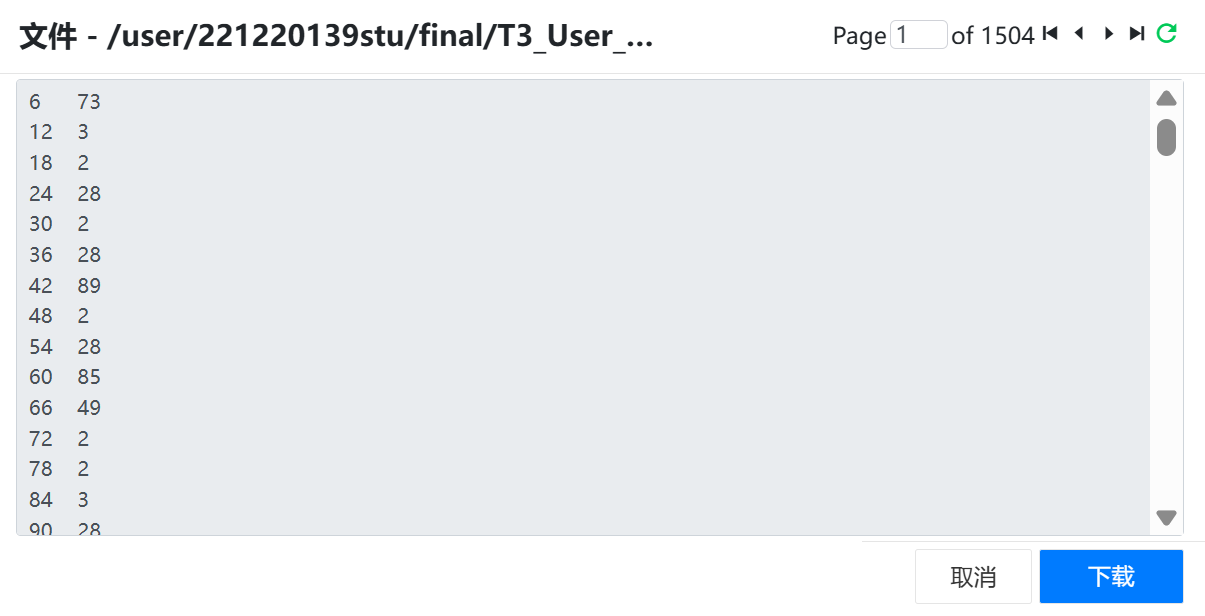
**输入**：降维后的用户向量 、kmeans迭代后的聚类中心数据(cache)、k值

**输出**：user\_id cluster\_id

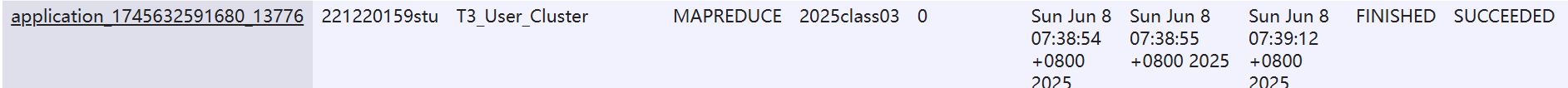
**mapper**的**setup**函数中，存储了每个聚类中心的位置，**map**函数中计算用户向量与每个聚类中心的距离，输出每个用户id以及距离最近的cluster\_id

没有**reducer**

**运行结果：**

****

**平台运行记录：**

****

**T3\_Cluster\_Preference**

根据用户偏好信息、用户聚类信息，生成用户类簇的偏好信息

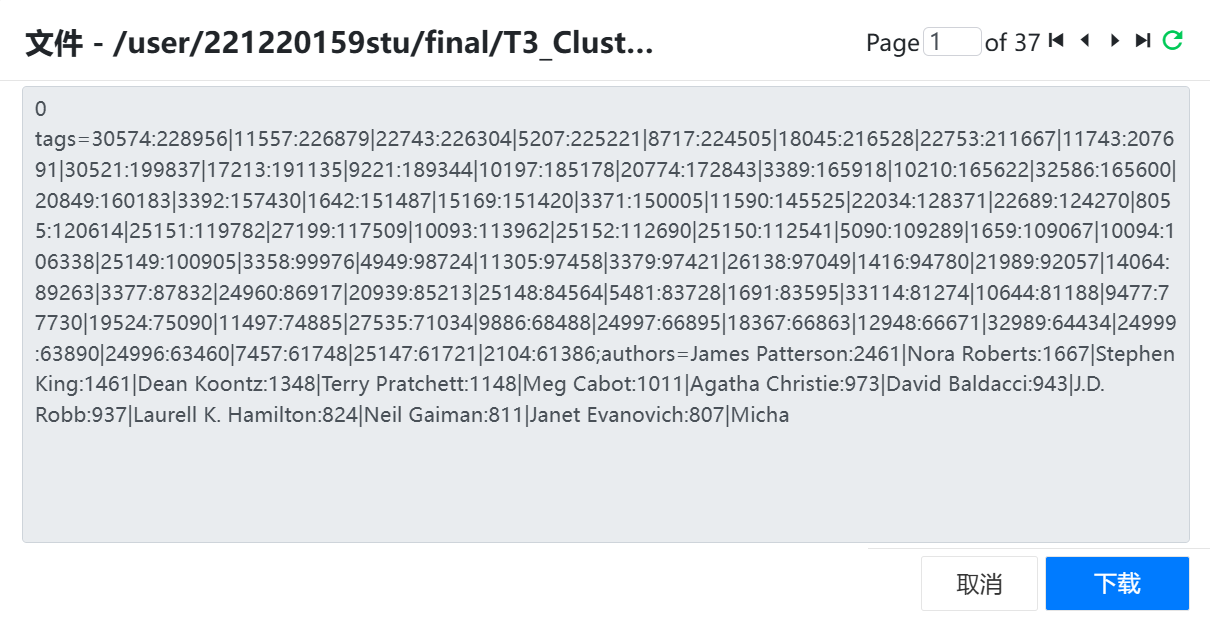
**输入**：user\_preference、 user\_cluster（cache）

**输出**：cluster\_id \t tags={tag\_id1}:{t\_num1}|{tag\_id2}:{t\_num2}|…;authors={author1}:{a\_num1}|{author2}:{a\_num2}|…;years={year1}:{y\_num1}|{year2}:{y\_num2}

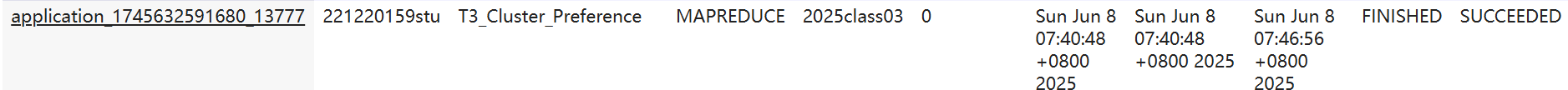
**mapper**的**setup**函数中存储了用户到类簇的对应关系，**map**函数中对于输入的用户偏好信息，将用户id改为所属的类簇id，向reducer中输出类簇的偏好信息，在**reducer**中再对同一个类簇收到的偏好信息进行整合

优化：选取最大的36/18个值时采用优先队列（堆）实现而不是将全量数据排序，降低了时间复杂度。

**运行结果：**

****

**平台运行记录：**

****

**任务四：个性化图书推荐服务**

**T4\_User\_R:**

获取该用户已评分的或已标记为‘to\_read’的图书集合R

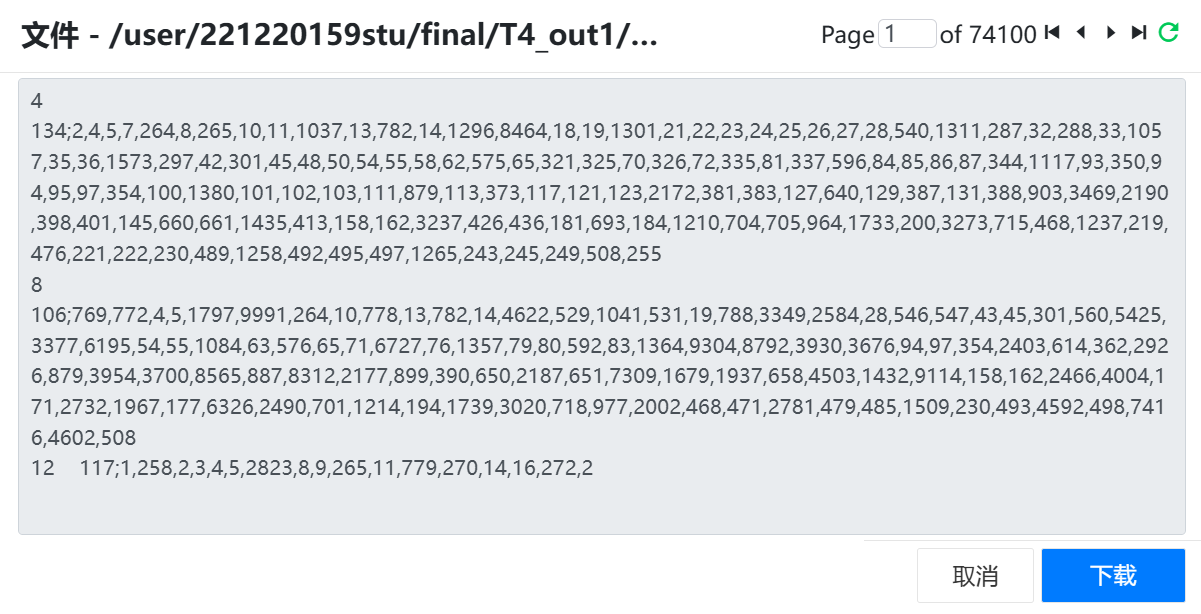
**输入**：ratings.csv、to\_read.csv

**输出**：每个用户已评分或已标记为"to\_read"的图书集合R，格式为"user\_id |R|;book\_id1,book\_id2,..."，例如"1 3;1,2,3" (|R|表示集合R的大小)

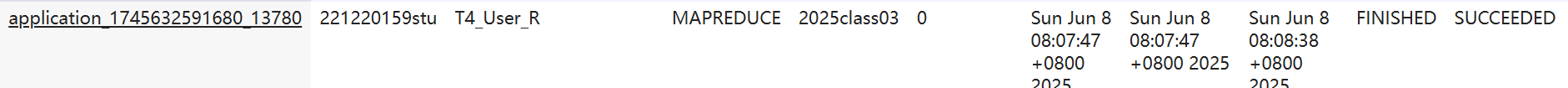
**mapper**中接收两个输入文件ratings.csv和to\_read.csv，输出(user\_id, book\_id)的键值对

**reducer**中对于同一个user\_id收到了其标记过的所有图书，使用set过滤，输出集合的大小以及其中的所有图书（经验证R的最大值为200）

**运行结果：**

****

**平台运行记录：**



**T4\_Cluster\_Recommend**

获取每个类簇相似度最高的220本图书

**输入**：cluster\_preference、books\_simplified、book\_tags\_flattened

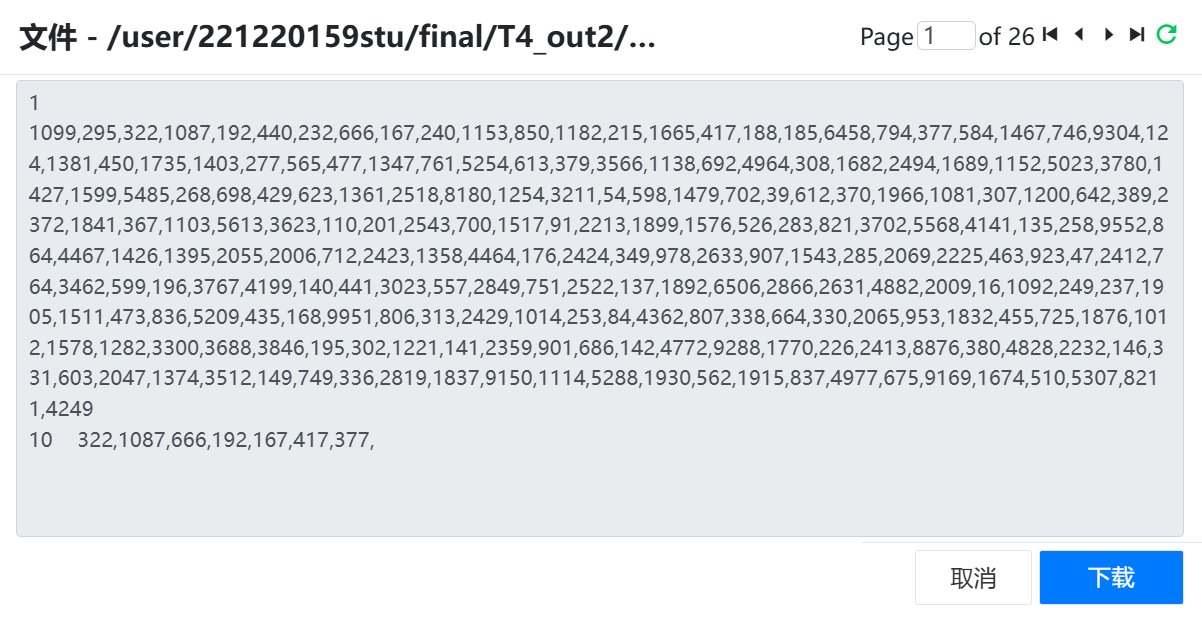
**输出**：与每个类簇相似度最高的220本图书，格式为"0 1,2,3,...,219"

由于基于兴趣匹配推荐需要根据用户所属的类簇的偏好信息推荐图书，就可以先预处理与每个类簇相似度最高的220本图书（R最大为200，220本可以满足10+|R|的要求）

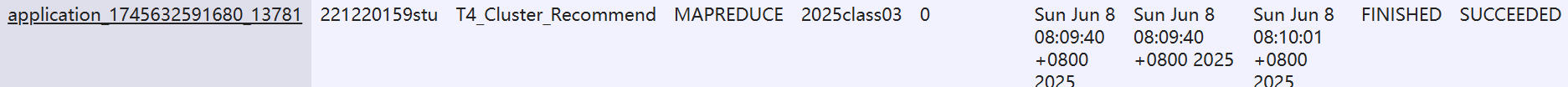
首先在**mapper**的**setup**方法中，把 books\_simplified 和 book\_tags\_flattened 从 cache 中读取出来，建立列表存储每本图书对应的标签及数量、年份、作者；在map函数中，输入一个类簇的偏好信息，将其存储，然后遍历所有图书，计算与该类簇的相似度，输出的键为cluster\_id，值为<bookId, totalSim>

**reducer**中使用一个最小堆取相似度最高的220本图书（添加后超出容量则pop），再经过排序后输出

**运行结果：**

****

**平台运行记录：**



**T4\_W1**

获取每个用户基于兴趣匹配推荐的图书集合W1

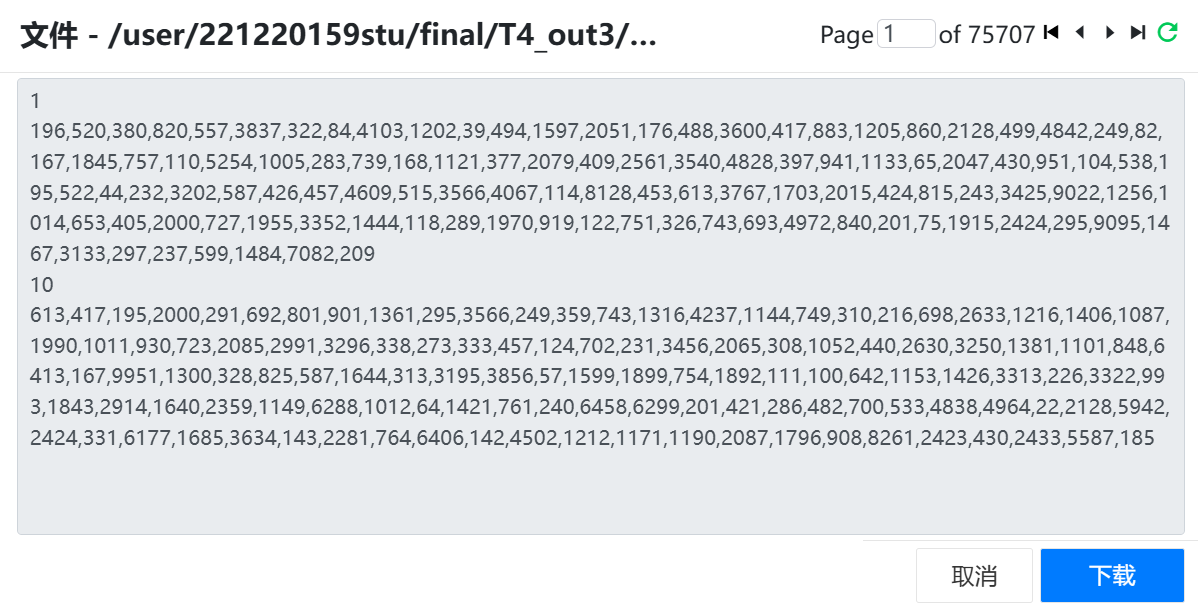
**输入**：user\_R、user\_cluster(cache)、cluster\_recommend(cache)

**输出**：每个用户的推荐图书集合W1，格式为"user\_id book\_id1,book\_id2,..."

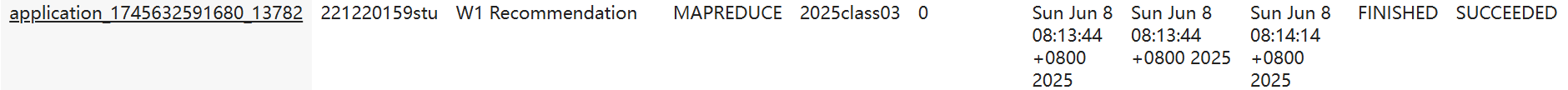
首先在**mapper**的**setup**方法中，从cache中读取 user\_cluster 和 cluster\_recommend，使用map存储用户所属类簇的映射表，以及类簇推荐图书的映射表；在**map**函数中，输入每个用户对应的图书集合R，通过用户id获取所属类簇id，再获取与该类簇偏好信息最相似的220本图书，截取前10+|R|本，排除已经在R里的图书并输出

没有**reducer**

**运行结果：**

****

**平台运行记录：**

****

**T4\_books\_similarity\_process**

获取与每本书相似度最高的 220 本书

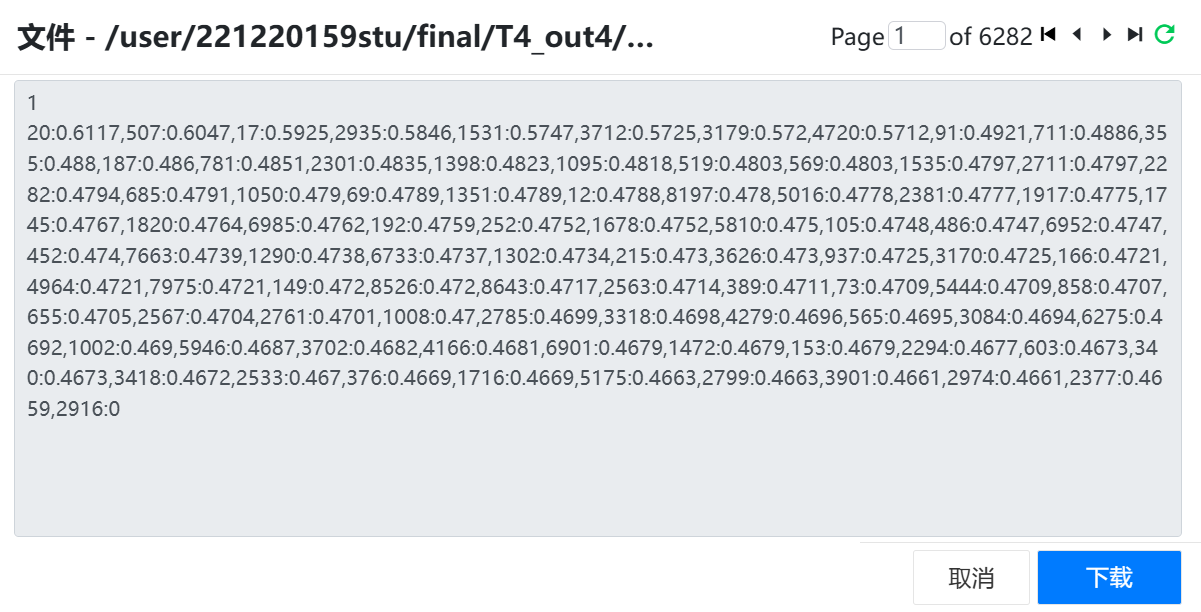
**输入**：books\_similarity

**输出**："book\_id bookX:simX,bookY:simY,...""

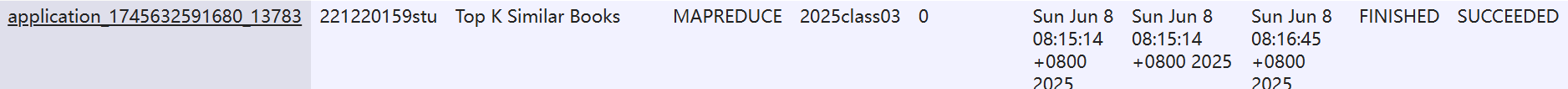
由于books\_similarity中存储的是图书对及相似度，10000本图书最多有10000\*10000个图书对，复杂度非常高，直接用该文件算W2会得到mapper超时的结果，因此新加一个类，从图书对的相似度获取与每本书相似度最高的 220 本书

**mapper**中对于一条输入“book1, book2 similarity”，双向输出“book1 book2:similarity”以及“book2 book1:similarity”，**reducer**中对于同一本书接收到的其它书与它的相似度，同样使用最小堆获取相似度最大的220本图书，并降序输出

**运行结果：**



**平台运行记录：**

****

**T4\_W2**

获取每个用户基于协同过滤推荐的图书集合W2

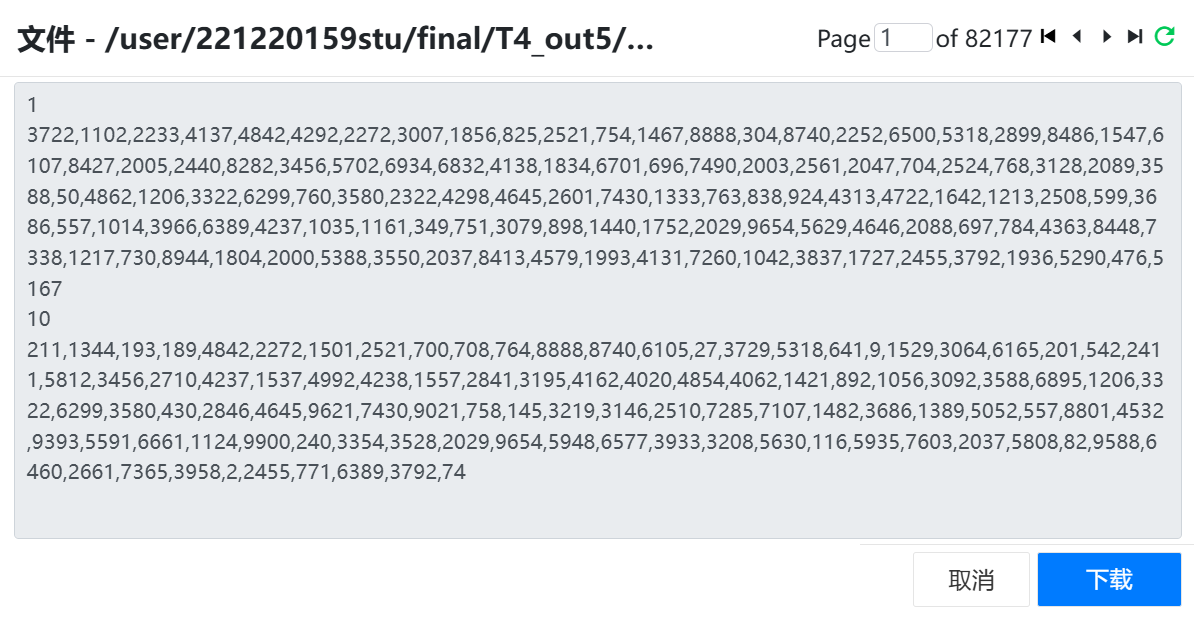
**输入**：user\_R、books\_similarity\_process

**输出**："user\_id book\_id1,book\_id2,..."

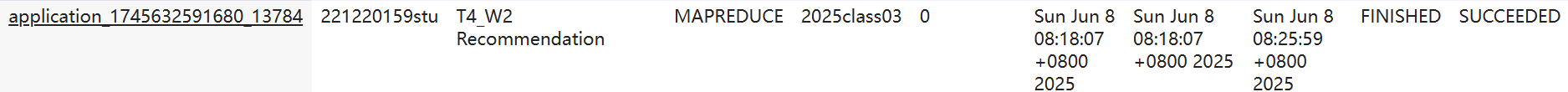
**mapper**中，首先在**setup**阶段从cache中读取刚才得到的books\_similarity\_process，建立每本书与其最相似图书列表的映射，**map**里遍历用户每本已读书籍获取相似书籍，使用最小堆获取相似度最高的10+|R|本图书，再排除R里的图书

没有**reducer**，新建了一个类BookSimilarity便于存储每本书及其相似度

**运行结果：**

****

**平台运行记录：**

****

**T4\_Recommend**

获取每个用户的加权推荐图书

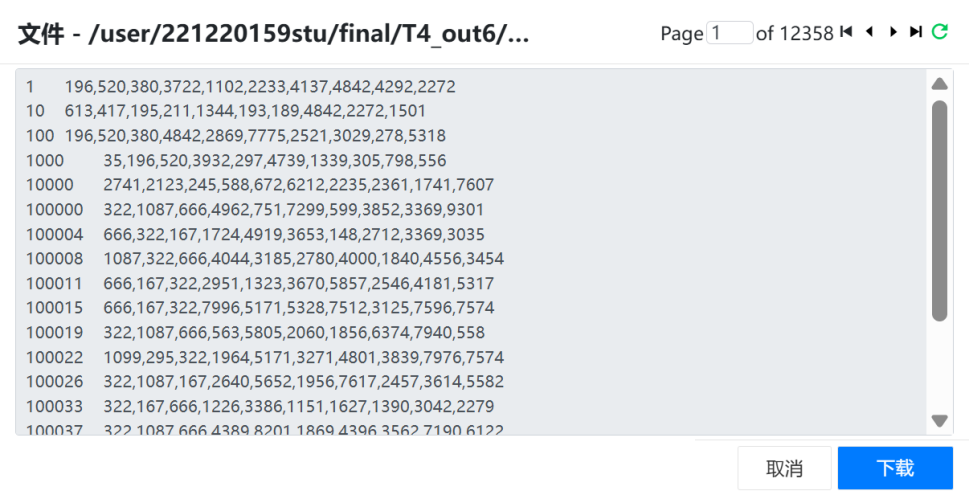
**输入**：推荐集合W1,W2

**输出**：10个文件中存放了依据不同系数获得的加权推荐图书，格式为"user\_id book\_id1,book\_id2,..."

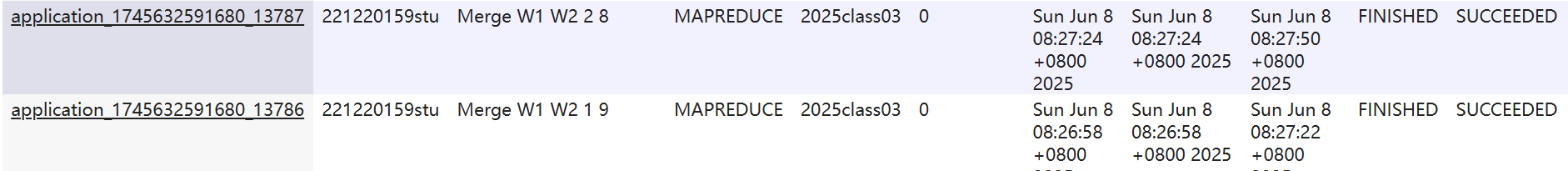
建立两个**mapper**，使用**MultipleInputs**将w1和w2映射到各自的mapper里，在其中将原本的value打上标号来自w1还是w2，**reducer**中遍历所有系数，分别取相应数量的图书输出

**运行结果：**

****

****

**平台运行记录：**

****