实时计算模块设计与实现

实时计算与离线计算应用于推荐系统上最大的不同在于实时计算推荐结果应该反映最近一段时间用户近期的偏好，而离线计算推荐结果则是根据用户从第一次评分起的所有评分记录来计算用户总体的偏好。

用户对物品的偏好随着时间的推移总是会改变的。比如一个用户 u 在某时刻对电影 p 给予了极高的评分，那么在近期一段时候，u 极有可能很喜欢与电影 p 类似的其他电影；而如果用户 u 在某时刻对电影 q 给予了极低的评分，那么在近期一段时候，u 极有可能不喜欢与电影 q 类似的其他电影。所以对于实时推荐，当用户对一个电影进行了评价后，用户会希望推荐结果基于最近这几次评分进行一定的更新，使得推荐结果匹配用户近期的偏好，满足用户近期的口味。如果实时推荐继续采用离线推荐中的 ALS 算法，由于算法运行时间巨大，不具有实时得到新的推荐结果的能力；并且由于算法本身的使用的是评分表，用户本次评分后只更新了总评分表中的一项，使得算法运行后的推荐结果与用户本次评分之前的推荐结果基本没有多少差别，从而给用户一种推荐结果一直没变化的感觉，很影响用户体验。

另外，在实时推荐中由于时间性能上要满足实时或者准实时的要求，所以算法的计算量不能太大，避免复杂、过多的计算造成用户体验的下降[38]。鉴于此，推荐精度往往不会很高。实时推荐系统更关心推荐结果的动态变化能力，只要更新推荐结果的理由合理即可，至于推荐的精度要求则可以适当放宽。

所以本文的实时推荐的算法应该具有以下一些特点：

（1）用户本次评分后、或最近几个评分后系统可以明显的更新推荐结果；

（2）计算量不大，满足响应时间上的实时或者准实时要求；

总之，实时推荐算法对推荐精度的要求不是很高，更注重的是算法响应时间实时性能力、更新推荐结果的合理性。

1 实时推荐算法设计

本文设计的实时算法的大体思想是：

当用户 u 对电影 p 进行了评分，将触发一次对 u 的推荐结果的更新。由于用户 u 对电影 p 评分，对于用户 u 来说，他与 p 最相似的电影们之间的推荐强度将发生变化，所以选取与电影 p 最相似的 K 个电影作为候选电影。

每个候选电影按照“推荐优先级”这一权重作为衡量这个电影被推荐给用户 u 的优先级。

这些电影将根据用户u最近的若干评分计算出各自对用户u的推荐优先级，然后与上次对用户u的实时推荐结果的进行基于推荐优先级的合并、替换得到更新后的推荐结果。

具体来说：

首先，获取用户u按时间顺序最近的K个评分，记为RK；获取电影p的最相似的K个电影集合，记为S；

然后，对于每个电影$q \in S$，计算其推荐优先级E_{uq}，计算公式见公式(1)：

$$E_{uq} = \frac{\sum sim(q,r) \times R_r}{sim\_sum} + \lg \max\{incount,1\} - \lg \max\{recount,1\} \quad (1)$$

其中：

R_r表示用户u对电影r的评分；

sim(q,r)表示电影q与电影r的相似度，设定最小相似度为0.6，当电影q和电影r相似度低于0.6的阈值，则视为两者不相关并忽略；

sim_sum表示q与RK中电影相似度大于最小阈值的个数；

incount表示RK中与电影q相似的、且本身评分较高（>=3）的电影个数；

recount表示RK中与电影q相似的、且本身评分较低（<3）的电影个数；

公式的意义如下：

首先对于每个候选电影q，从u最近的K个评分中，找出与q相似度较高（>=0.6）的u已评分电影们，对于这些电影们中的每个电影r，将r与q的相似度乘以用户u对r的评分，将这些乘积计算平均数，即$\frac{\sum sim(q,r) \times R_r}{sim_sum}$，作为用户u对电影q的评分预测；。

然后，将u最近的K个评分中与电影q相似的、且本身评分较高（>=3）的电影个数记为incount，计算$\lg \max\{incount,1\}$作为电影q的“增强因子”，意义在于电影q与u的最近K个评分中的n个高评分(>=3)电影相似，则电影q的优先级被增加$\lg \max\{incount,1\}$。如果电影q与u的最近K个评分中相似的高评分电影越多，也就是说n越大，则电影q更应该被推荐，所以推荐优先级被增强的幅度较大；如果电影q与u的最近K个评分中相似的高评分电影越少，也就是n越小，则推荐优先级被增强的幅度较小；

而后，将u最近的K个评分中与电影q相似的、且本身评分较低（<3）的电影个数记为recount，计算$\lg \max\{recount,1\}$作为电影q的“削弱因子”，意义在于电影q与u的最近K个评分中的n个低评分(<3)电影相似，则电影q的优先

级被削减 $\lg\max\{incount,1\}$。如果电影 q 与 u 的最近 K 个评分中相似的低评分电影越多，也就是说 n 越大，则电影 q 更不应该被推荐，所以推荐优先级被减弱的幅度较大；如果电影 q 与 u 的最近 K 个评分中相似的低评分电影越少，也就是 n 越小，则推荐优先级被减弱的幅度较小；

最后，将增强因子增加到上述的预测评分中，并减去削弱因子，得到最终的 q 电影对于 u 的推荐优先级。

在计算完每个候选电影 q 的 E_{uq} 后，将生成一组<电影 q 的 ID, q 的推荐优先级>的列表 updatedList：

$$\text{updatedList} = \bigcup_{q \in S} \{qID, E_{uq}\} \tag{2}$$

而在本次为用户 u 实时推荐之前的上一次实时推荐结果 Rec 也是一组<电影 m,m 的推荐优先级>的列表，其大小也为 K：

$$\text{Rec} = \bigcup_{m \in \text{Rec}} \{mID, E_{um}\},\ \ len(\text{Rec}) = K \tag{3}$$

接下来，将 updated_S 与本次为 u 实时推荐之前的上一次实时推荐结果 Rec 进行基于合并、替换形成新的推荐结果 NewRec：

$$\text{New Rec} = topK(i \in Rec \cup updatedList, cmp = E_{ui}) \tag{4}$$

其中，i 表示 updated_S 与 Rec 的电影集合中的每个电影，topK 是一个函数，表示从 $Rec \cup updated\_S$ 中选择出最大的 K 个电影，$cmp = E_{ui}$ 表示 topK 函数将推荐优先级 E_{ui} 值最大的 K 个电影选出来。最终，NewRec 即为经过用户 u 对电影 p 评分后触发的实时推荐得到的最新推荐结果。

总之，实时推荐算法流程流程基本如下：

（1）用户 u 对电影 p 进行了评分，触发了实时推荐的一次计算；

（2）选出电影 p 最相似的 K 个电影作为集合 S；

（3）获取用户 u 最近时间内的 K 条评分，包含本次评分，作为集合 RK；

（4）利用公式(4-1)，计算电影的推荐优先级，产生<qID,>集合 updated_S；

将 updated_S 与上次对用户 u 的推荐结果 Rec 利用公式(4-4)进行合并，产生新的推荐结果 NewRec；作为最终输出。

2 电影间相似度的计算

由于算法需要多次计算电影间相似度，而为了计算电影间相似度需要为电影提取特征，但是本文的数据集并不包含电影的具体信息，比如电影类别、导演、演员等，而仅仅有电影的名称，电影本身的信息量太少，所以无法直接提取特征。

不过这并不代表电影间相似度无法计算，这里可以应用离线计算的 ALS 算法中的一部分内容。回忆离线计算的 ALS 算法，算法最终会为用户、电影分别生成最终的特征矩阵，分别是表示用户特征矩阵的$U(\mathrm{m}\times k)$矩阵，每个用户由 k 个特征描述；表示物品特征矩阵的$V(n\times k)$矩阵，每个物品也由 k 个特征描述。

我们发现，$V(\mathrm{n}\times k)$表示物品特征矩阵，每一行是一个 k 维向量，虽然我们并不知道每一个维度的特征意义是什么，但是 k 个维度的数学向量表示了该行对应电影的特征。

所以，每个电影用$V(\mathrm{n}\times k)$每一行的$<t_1,t_2,t_3,....,t_k>$向量表示其特征，于是任意两个电影 p：特征向量为$V_{\mathrm{p}}=<t_{\mathrm{p1}},t_{p2},t_{p3},....,t_{pk}>$；电影 q：特征向量为$V_{\mathrm{q}}=<t_{q1},t_{q2},t_{q3},....,t_{qk}>$之间的相似度$\mathrm{sim(p,q)}$可以使用$V_{\mathrm{p}}$和$V_{\mathrm{q}}$的余弦值来表示，见公式(4-5)：

$$Sim(p,q)=\frac{\sum_{i=0}^{k}(t_{pi}\times t_{qi})}{\sqrt{\sum_{\mathrm{i=0}}^{k}t_{pi}^2}\times\sqrt{\sum_{\mathrm{i=0}}^{k}t_{qi}^2}} \tag{5}$$

数据集中任意两个电影间相似度都可以由公式(4-5)计算得到。由于此公式涉及到了向量乘法、向量取长度等比较复杂的计算，所以相对来说相似度计算是比较费时的。

更关键的是，由于算法需要多次计算电影间相似度，且算法本身是实时算法也就是不断重复运行的，于是采用此算法的实时系统将不断执行大量的相似度计算，造成了计算效率的下降，可能影响到系统的实时响应能力。

幸运的是，任意两个电影间的相似度是固定值，不会随着时间而改变。所以我们可以利用“用空间换取时间”的思想，事先计算出每两个电影间相似度并保存在特定的数据结构里。如此一来，算法需要电影间相似度时，直接查询这个数据结构即可，只要这个数据结构的查询时间很优秀，整个算法即可节省大量计算时间。

本文将各个电影间相似度提前计算并存储到命名为 simHash 的二层哈希表（即哈希表的 Value 也是一个哈希表）中，simHash 数据结构见图 2：

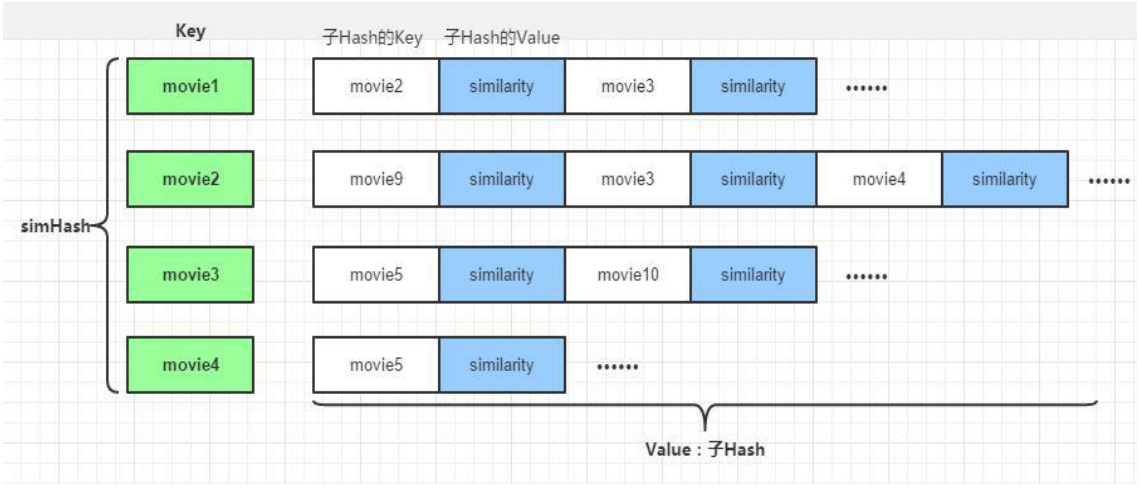

图 2 保存相似度的变量 simHash 数据结构

如图所示，这个数据结构和图的邻接表表示法很相似。simHash 的 key 为电影 Id，value 为其他电影与本电影相似度的哈希表。

这样的数据结构使得相似度查询非常高效（时间复杂度为 $O(1)$），比如要在 simHash 变量上查询 movie1 与 movie3 的相似度，其过程如下：

（1）先在 simHash 查询并获取 key=movie1 的 value，也就是获取 movie1 的子哈希 subHashForMovie1；

（2）在 movie1 的子哈希 subHashForMovie1 中查询 key=movie3 的 value，这里的 value 就是 movie1 和 movie3 之间的相似度。

这样一来，实时计算过程中每当需要计算某两个电影间相似度的时候，只需要去 simHash 二级哈希表中查询即可，查询的时间复杂度为 $O(1)$。当然，出色的时间复杂度是建立在大量的空间复杂度上的，由于本文涉及的电影个数为 17770 个，所以形成的 simHash 将占用较大的内存空间，经过编程实现得知 simHash 占用了多达 556MB 内存空间，不过由于本文的实验环境为真实的分布式平台且任一台服务器的内存承受这个内存空间的消耗都是毫无压力的，所以空间复杂度是可以忍受的。

3 Spark Streaming 实时推荐算法实现

由于算法使用到了用户的最近 K 次评分，本文事先将用户的所有评分记录到 MongoDB 中，以便于算法查询，具体细节将在下一章 MongoDB 存储部分详细介绍。这里只需知道在 MongoDB 中创建了数据库 DB=RecommendingSystem，并在这个 DB 中创建了集合 collection=ratingsCollection 来存储历史评分数据。

根据前文的描述，已经将所有评分记录保存到 MongoDB 数据库中的 DB[RecommendingSystem]数据库下的集合 collection[ratingsCollection]中；并且

已经将电影间相似度通过 Spark broadcast 机制发送到了各个计算节点。本文接下来详细描述了实时推荐算法的具体实现。

实时推荐算法输入为一个评分<userId, movieId, rate, timestamp>，而执行的核心内容包括：获取 userId 最近 K 次评分、获取 movieId 最相似 K 个电影、计算候选电影的推荐优先级、更新对 userId 的实时推荐结果。

（1）获取 userId 最近 K 次评分

对于一个 userId，查询最近的 K 次评分方法如下：

```
use RecommendingSystem;
db.ratingsCollection.find({“userId”:userId}).sort({“timestamp”:-1}).limit(K)
```

上述查询语句的意思是：使用数据库 RecommendingSystem，并在集合 ratingsCollection 中查询 userId 列为 userId 的评分条目并按照 timestamp 从大到小排序，并选出前 K 个条目，于是用户 userId 的最近 K 次评分被获取。

上述 MongoDB 查询的输出是元素内容为<movieId,userId 给 movieId 的评分值 rate>、数据类型为 Array[(Int, Double)]的数组，命名为 recentRatings；

（2）获取 movieId 最相似 K 个电影

由于电影间相似度都保存在 simHash 中，所以每个电影 movieId 的最相似的 K 个电影很容易获取：从 movieId 在 simHash 对应的子哈希表中获取相似度前 K 大的那些电影。输出是数据类型为 Array[Int]的数组，表示与 movieId 最相似的电影集合，并命名为 candidateMovies 以作为候选电影集合。

（3）计算候选电影的推荐优先级

根据推荐优先级计算公式(4-1)，对于候选电影集合 candidateMovies 和 userId 的最近 K 个评分 recentRatings，算法内容如下：

算法 4-1：计算候选电影的推荐优先级

输入：候选电影集合candidateMovies，userId的最近K次评分recentRatings

输出：每个候选电影的推荐优先级

```
allSimilars:Array[(Int, Double)]() //声明allSimilars数组
incount:HashMap[Int,Int]() //声明incount哈希表
recount:HashMap[Int,Int]() //声明recount哈希表

对candidateMovies中每个电影p, 对recentRatings中每个评分 <q, userId, rate>:
    sim = 用p和q在simHash中查询相似度的值
    if sim >= 0.6:
        allSimilars += (p, rate×sim)
        if rate >= 3.0: incount[p] += 1
        else: recount[p] += 1
    tmpSimilars = allSimilars.groupBy{case (p, v) => p}
    updatedRecommends = tmpSimilars.map{ case (p, array) =>
```

```
    (p, array.sum/array.length+log(incounter[p])-log(recounter[p])}
return updatedRecommends
```

其中，allSimilars 是用于存储中间结果的数组；

incount 为哈希表，key=每个候选电影 q，value=RK 中与电影 q 相似的、且本身评分较高（>=3）的电影个数，value 初始值=0；

recount 为哈希表，key=每个候选电影 q，value=RK 中与电影 q 相似的、且本身评分较低（<3）的电影个数，value 初始值=0；

算法首先对候选电影集合 candidateMovies，userId 的最近 K 次评分 recentRatings 进行了遍历，对于每个候选电影 p、每个评分<q,rate>：

a.用电影 p 的 ID、q 的 ID 查询 simHash，获取电影 p、q 的相似度 sim；

b.如果 sim >= 0.6，说明 p、q 电影相关，则将临时元组(p,rate×sim)存入数组 allSimilars 中；

c.如果 rate >= 3.0，说明 q 电影得到了 userId 较高的评分，则 incount[p]+=1；否则，即 rate < 3.0，说明 q 电影得到了 userId 较低的评分，则：recount[p]+=1；

d.遍历完成后，allSimilars 数组的每个元素是元组(p,v)，从上面得知 p 表示候选电影，v 表示公式(4-1)中的 $sim(q,r)\times R_r$；

e.接下来，将 allSimilars 按照每个元组元素的 p 来将 p 值相同的元组的 v 值合并成数组 array；不同的 p 将拥有不同的 array，<p,array>将组成一个新数组 tmpSimilars：

```
val tmpSimilars = allSimilars.groupBy{case (p, v) => p}
```

这个操作会将 allSimilars 变换成新数组，示例如图 4-3：

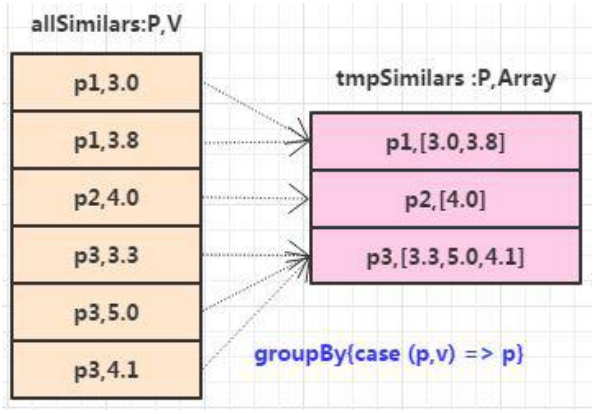

图 4-3 groupBy 过程

f.对于新数组 tmpSimilars 的每一项<p,array>而言，对 array 计算：array.sum

/ array.length 即表示其电影 p 对应公式(4-1)的 $\frac{\sum_{r\in RK} sim(\mathrm{q},r)\times R_r}{sim\_sum}$ 部分；而 log(incounter[p])即表示其电影 p 对应公式(4-1)的 $\lg\max\{incount,1\}$，且 log(recounter[p])即表示其电影 p 对应公式(4-1)的 $\lg\max\{recount,1\}$。所以，接下来从 tmpSimilars 计算<p, E_{up}>数组：

```
val updatedRecommends = tmpSimilars.map{
  case (p, array) =>
 (p, array.sum/array.length+log(incounter[p])-log(recounter[p])}
```

updatedRecommends 即为<p, E_{up}>数组，它作为算法最终输出。

（4）更新对 userId 的实时推荐结果

当计算出候选电影的推荐优先级的数组 updatedRecommends<movieId, E>后，这个数组将被发送到 Web 后台服务器，与后台服务器上 userId 的上次实时推荐结果 recentRecommends<movieId, E>进行合并、替换并选出优先级 E 前 K 大的电影作为本次新的实时推荐。具体而言：

a.合并：将 updatedRecommends 与 recentRecommends 并集合成为一个新的<movieId, E>数组；

b.替换（去重）：当 updatedRecommends 与 recentRecommends 有重复的电影 movieId 时，recentRecommends 中 movieId 的推荐优先级由于是上次实时推荐的结果，于是将作废，被替换成代表了更新后的 updatedRecommends 的 movieId 的推荐优先级；

c.选取 TopK：在合并、替换后的<movieId, E>数组上，根据每个 movie 的推荐优先级，选择出前 K 大的电影，作为本次实时推荐的最终结果。

实时计算模块的流程如图 4-4：

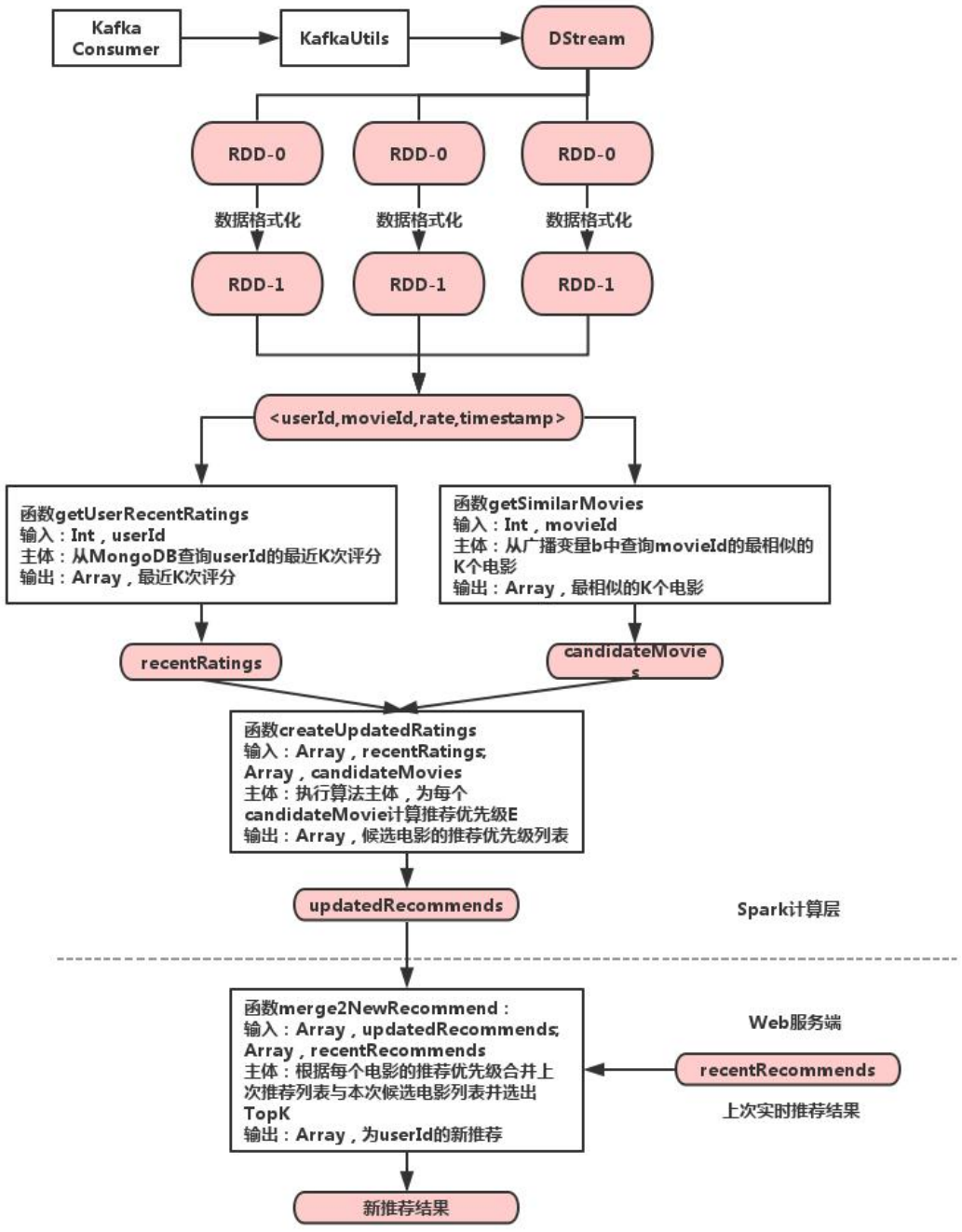

图 4-4 实时推荐算法流程图

（1）首先 Spark 实时计算程序利用 KafkaUtils 工具获取 Kafka 集群得到的消息，生成 DStream；

（2）对于 DStream 中每个 RDD：RDD-0、对于每个 RDD-0 中的每个数据条目，进行简单的数据格式化产生每条记录为<userId,movieId,rate,timestamp>的新 RDD：RDD-1；

（3）对于 RDD-1 中每条<userId,movieId,rate,timestamp>记录：

a.根据 userId，从 MongoDB 中获取 userId 最近的 K 次评分记录 recentRatings；

b.根据 movieId，从广播变量中获取与 movieId 最相似 K 个电影集 candidateMovies；

c.对于每个电影 q∈candidateMovies，利用公式(4-1)计算出其推荐优先级 E_{uq}，产生<qId, E_{uq}>的列表 updatedRecommends；

d.将 updatedRecommends 发回到 Web 后台；

（4）Web 后台将收到的每条 updatedRecommends 与其对应的 userId 上次实时推荐结果进行合并、替换，根据 E_{uq} 值选出前 K 个作为最新推荐结果。