基于布尔表达式的索引结构

介绍

对于设置了定向条件的广告，我们可以将它的定向条件写成一个布尔表达式。比如一个广告设置定向投放给超一线城市的男性用户，年龄在20 ~ 40之间，且安装了某个汽车app。这又是个视频广告，所以广告主又加上了投给处于WIFI环境下的用户。假设在我们的系统中，性别属性男的取值为1，超一线城市取值包括Beijing、Shanghai、Guangzhou、Shenzhen，年龄20~40之间可以分到年龄标签4、5、6，WIFI环境取值为1，安装了某个汽车app就是appId=xxx。那么这条广告的定向信息就可以转换为下面这个表达式：

> *GENDER(1) CITY(Beijing, Shanghai, Guangzhou, Shenzhen) AGE(4,5,6) NETWORK(1) INSTALLED_APP (xxx)*

这样的一个布尔表达式，反过来可以代表多个有同样定向条件的广告。当有符合条件的请求到来时，只要索引到这样一个表达式，就可以索引到定向为这种条件的广告。这样，对于广告的高效索引的问题可以转化为对于布尔表达式集合的高效索引。

术语介绍

1. Attribute 一种具体的属性。包含两个信息：属性类别和属性取值。如：*AGE: 5，GENDER: 1*
2. Clause/Predicate 属性取值范围的布尔表达式。如：*AGE ∈ (4,5,6)，GENDER ∈ (1)*
3. Conjunction 由clause/predicate组成的合取范式，表示广告的定向信息。conjunction的size定义为“∈”的个数。如：*PLACEMENT_TYPE∈{157} ∧AGE∈{1,2,3,4,5,6,7}∧NETWORK∈{1} ∧ INSTALLED_APP{80528}*，size=3
4. Assignment 用attribute表示的流量画像。如：
   PLACEMENT_TYPE: 2,PLACEMENT_TYPE: 20,PLACEMENT_TYPE: 28,PLACEMENT_TYPE: 29,AGE: 4,APP_INTEREST: 19-1,APP_INTEREST: 11-0,GENDER: 1,INSTALLED_APP: 30202,INSTALLED_APP: 60592,INSTALLED_APP: 87825,MEDIA_TYPE: 43,NETWORK: 1,TAGS: 6396,TAGS: 8091,TAGS: 8173,TAGS: 5114,TAGS: 8356,TAGS: 6378,TAGS: 8152,TAGS: 7871,TAGS: 8326,TAGS: 6933,TAGS: 6932,TAGS: 6936

算法简介

拿如下conjunction来看，定义conjunction的size为∈语句的个数。

conjunctionId	conjunction	size
1	PlacementType∈(2)	1
2	PlacementType∈(2) ∧ AppInterest∈(19-1)	2
3	PlacementType∈(2) ∧ IpGeo∈(141) ∧ NetworkType∈(1) ∧ AppInterest(19-1)	3
4	PlacementType∈(2) ∧ InstalledApp∈(30202) ∧ NetworkType∈(1)	3
5	PlacementType∈(2) ∧ AppInterest∈(15-0, 19-1) ∧ InstalledApp∈(30202)	3
6	PlacementType∈(2) ∧ AppInterest∈(15-0, 19-1) ∧ NetworkType∈(1)	3

我们构造一些流量请求来看能匹配到哪些conjunction。

Assignment s1 ={PlacementType: 2} 可以选出conjunction 1。

s2 ={PlacementType: 2, AppInterest: 19-1} 可以选出conjunction 1和2。

s3 ={PlacementType: 2, AppInterest: 19-1, IPGeo: 141, NetworkType: 1} 可以选出conjunction 1、2和6。但是不能选出3，因为AppInterest: 19-1对于conjunction 3来说是排除项。

s4 ={PlacementType: 2, AppInterest: 19-1,AppInterest: 15-0} 可以选出conjunction 1和2。不能选出5和6，因为对于5和6中appInterest类别的values，s4的取值只需要满足其中一个就可以。所以s4只满足了conjunction 5、6中的PlacementType和AppInterest 两项要求。而conjunction 5的InstalledApp，以及conjunction 6的NetworkType要求没有被满足。

通过观察得到，assignment要满足conjunction（size k）需要符合以下两个条件：

1. assignment要满足conjunction中的K个∈语句。
2. assignment不能包含语句覆盖的attribute。

一个clause/predicate，如AGE∈(4,5,6)，表示若AGE attribute的取值为4、5、6之一(AGE: 4，AGE: 5，或AGE: 6)，就可以使该predicate为true。为了高效的选出一个流量可以满足的conjunction，我们对conjunction建立attribute → conjunction的倒排索引。然后为了快速检查assignment有没有满足conjunction的可能性，在外层又加了conjunction size为key的索引。

Index
`Map<Integer, Map<Attribute, PostingList>> index;`

转换成倒排索引以后如下所示：

K	Attribute(key)	PostingList(conjunctionId, 包含/不包含)
1	PlacementType: 2	(1, ∈)
2	PlacementType: 2	(2, ∈)
	AppInterest: 19-1	(2, ∈)
3	PlacementType: 2	(3, ∈), (4, ∈), (5, ∈), (6, ∈)
	AppInterest: 19-1	(3,), (5, ∈), (6, ∈)
	AppInterest: 15-0	(5, ∈), (6, ∈)
	InstalledApp: 30202	(4, ∈), (5, ∈)
	NetworkType: 1	(3, ∈), (4, ∈), (6, ∈)
	IPGeo: 141	(3, ∈)

assignment S(包含t个attribute)寻找可以满足的conjunction：

K-index(K <= t)中，S以自己的attributes为key，选出来一些Attribute → PostingList的mapping。这些mapping按照attribute的属性（AGE，APP_INTEREST...）可以合并为n个AttributeCategory → PostingList的mapping。一个conjunction c（包含K个∈语句）需要出现在K个AttributeCategory→ PostingList的mapping中，且在这K个mapping中c都标记为∈。且c不能在任何一个AttributeCategory→ PostingList的mapping中标记为。

那么获取广告候选集的步骤总结如下：

1. 将请求转化为流量画像（Assignment），assignment的大小（attribute数量）为t
2. K为倒排索引中size的最大值，对于k∈(min(K, t), 1, -1)，用Assignment中的attribute为key，取出对应的list来判断是否有可以满足的conjunction。
3. 根据选出的conjunction来获取其关联的adId

其中，算法的核心在于第二步。这一步拆开来看可以分为两个步骤：

1. 以Assignment的attribute为key，取出attribute对应的conjunction list。同时将同类型attribute的conjunction list进行合并。合并完成后保持conjunction id依然从小到大排序。
2. 对每一个conjunction list设置一个current指针，初始化为list中的第一个元素。假设当前处于size=k的index下，conjunction list的合集由postingListHolders表示。按照如下逻辑进行运算：

compareAndFind

```
// kholderconjunctionId
while (postingListHolders.get(k - 1).getCurrentConjunction() != null) {

    Conjunction nextConjunction;
    // conjunctionIdkPostingListHolder
    if (postingListHolders.get(0).getCurrentConjunction() == postingListHolders.get(k - 1).
getCurrentConjunction()) {
        // conjunctionId
        if (postingListHolders.get(0).isCurrentConjunctionExcluded()) {
            // holdercurrentIdrejectId1conjunctionId
            long rejectId = postingListHolders.get(0).getCurrentConjunction().getId();
            nextConjunction = postingListHolders.get(0).nextPossibleConjunction();
            for (PostingListHolder postingListHolder : postingListHolders) {
                if (postingListHolder.getCurrentConjunction() != null
                    && postingListHolder.getCurrentConjunction().getId() == rejectId) {
                    postingListHolder.skipTo(nextConjunction);
                } else {
                    break;
                }
            }
            // holdercurrentId
            Collections.sort(postingListHolders);
            continue;
        } else {
            // conjunctionIdresult
            result.add(postingListHolders.get(k - 1).getCurrentConjunction());
            nextConjunction = postingListHolders.get(k - 1).nextPossibleConjunction();
        }
    } else {
        // kholdercurrentId1holdercurrentIdkholdercurrentIdconjunctionId
        nextConjunction = postingListHolders.get(k - 1).getCurrentConjunction();
    }

    // holdercurrentIdnextIdnextIdid
    for (int i = 0; i < k; i++) {
        postingListHolders.get(i).skipTo(nextConjunction);
    }
    // holdercurrentId
    Collections.sort(postingListHolders);
}
```

文献：

Indexing Boolean Expressions.pdf