**基于bitmap实现用户画像的标签圈人功能**

用户画像系统中有一个很重要的功能点： 基于标签圈人。这里有个很核心的概念，什么是标签？

标签是简化用户表示的一种思维方式。 刻画用户的标签越多，用户画像就越立体。 比如:

90后，码农，宅男 3个标签就刻画了一类人。标签类似于戏曲中的脸谱来表现人物的性格和特征。

标签有哪些类型呢？

枚举类标签: 描述性别，地理位置。这类标签取值通常是可枚举出来的。

时间类标签: 描述业务触达和流失时间信息。 注: 时间类标签可存储成数值。

数值类标签: 比如账户金额，积分数量等。

所以本质上，标签只有两种: 离散枚举和连续数值。

有了标签后，如何在计算机中建模存储呢？

最简单最直观的方式就是设置大宽表，即每个标签一个字段。 通常一个小型的画像系统，有几百个标签足够。所以对于大部分场景宽表足够简单可依赖。

宽表一般存储在Hive中，出于性能考虑，会存储到Impala中。当数据量较大时，Impala也一般无法满足查询的性能需求。这是因为Impala没有索引，每次查询都是扫表。所以，为了能够利用索引提升性能，大宽表一般会从Impala转存到Elasticsearch中。

当一个用户Id附着成百上千个标签，按ES存储方式，会相当耗费存储资源，导入数据到ES也会成为性能瓶颈。 所以变通的方案是将所有的标签存储到ES的一个array字段中。但本质上，还是大宽表的方案。

大宽表的方案最大的问题: 新增标签时间成本太大，所以画像系统基本是T+1的实效性。 如果对响应时间没有苛刻的要求，基于Hadoop生态的ad hoc查询引擎构建宽表，比如Impala或presto是可以使用多张宽表来解决新曾标签T+0生效问题，毕竟大数据系统，存储资源还是很充足的。

可惜的是业务对系统的需求是: 更高，更快，更强，像体育运动一样。

我们用ES存储标签，查询速度快的原因是ES构建了倒排索引。我们构建标签时，标签数据的主体是用户ID, 而在ES的世界，站在倒排索引的角度，标签数据的主体是标签，这完全是两个对立面。

我们使用标签圈人，本质上是集合的交并补运算。 所以，我们可以干脆再往前迈一步: 直接构建标签-用户ID的映射关系，而非原始的用户ID-标签。

这样，整个数据结构就变成类似如下的样式:

男: 张三，李四，王五...

1.

由于一个标签可以圈定上亿的用户，如何存储这样的结构？ RoaringBitmap 。这样存储后，标签圈人就脱离了SQL和ES语法，还原到最本质的集合运算:A and B or (C and D)。

使用标签-用户ID这种数据建模方式，有个很大的问题: 数值类标签的处理。比如用户积分。 通常有一种解决方法就是分段，然而这样做损失了数据精度。变得不灵活了。还有一种解决方法是为每个值建立一个bitmap。 这样做一则耗费空间，二则无法很好处理区间查询的问题。

使用标签-用户ID这种方式， bitmap存储数据关系是标签值等于XXX的用户ID, 提取核心点bitmap存储的是等于关系。 那么bitmap存储大于或者小于关系也是可以的。

对于数值型标签，我们重新定义存储关系: bitmap(2) 表示value值大于2的所有用户ID。 同理， bitmap(5) 表示value值大于5的所有用户ID。这样的话，计算value=(3,1000)之间的用户，使用bitmap(3) andNot bitmap(999)就可以了。很好地解决了区间查询的问题。 依然遗留了一个问题： 需要为每个值准备一个bitmap。

这个问题的解决思路很巧妙: 多个bitmap组合表示一个数值。例如200, 拆分成个位，十位，百位3个部分，每一部分用10个bitmap存储。这样就能够把bitmap的数量控制在有限的数量里面。比如对于int整型，最多需要100个bitmap。

优化是没有止尽的，我们还能走得更远。如果数值采用二进制表示，那么每一位只需要2个bitmap, 一个Int类型最多需要64个bitmap。 采用二进制，存储的规则可以如下设置:

bitmap(0)表示该位为0的用户ID集合。

bitmap(1)表示该位为0或1的用户ID集合。

1.

2.

由于对于二进制的某一位，取值只有0和1两种可能，所以对于二进制，每一位只需要bitmap(0), 所以最多需要32+1=33个bitmap存储。

综上， 我们解决bitmap数量的问题，也解决了区间查询的问题。但是多位二进制组合处理区间查询，又引出了新的问题： 多个bitmap如何组合表示一个区间？

我们把问题再简化一下，多个bitmap如何表示一个小于等于的区间。 比如i<7 如何用bitmap表示？ 再回顾bitmap的存储规则:

bitmap(0)表示该位为0的用户ID集合。

bitmap(1)表示该位为0或1的用户ID集合。

1.

2.

我们按从右到左的顺序给bitmap位取名字，下标从1开始。 例如01，有两位，分别是b2,b1。

这样的话: i<=7 = i<0111， 用bitmap表示就是b4。 再举几个例子:

i<=5 等价于 i <= 0101<二进制>, 用bitmap表示就是 (b4 and b2) or (b4 and b3)

1.

为了理解这个过程，我自己画了如下的横向树形图:

1

1

0

0

1

0

0

来观察这个规律，最后实现的代码如下:

import lombok.Data;

import org.roaringbitmap.RoaringBitmap;

import java.util.\*;

public class RangeBitmapDemo2 {

@Data

private static class QueryCond{

private List<String> base = new ArrayList<>();

private List<List<String>> lowerRange = new ArrayList();

public void addBase(String val){

this.base.add(val);

}

public void addLowerRange(String val){

List<String> list = new ArrayList();

list.addAll(base);

list.add(val);

this.lowerRange.add(list);

}

}

public static void query(int upper, int binaryLength){

String val =String.format("%"+binaryLength+"s", Integer.toBinaryString(upper)).replaceAll(" ","0"); //这里可以补空格

System.out.println("query: upper is "+val);

QueryCond cond = new QueryCond();

char cur = val.charAt(0);

if(cur=='0'){

cond.addBase("b"+binaryLength);

}

for(int i=1;i<val.length();i++){

cur = val.charAt(i);

if(cur == '0'){

int back = i-1;

while(back>=0 && val.charAt(back)=='1'){

cond.addLowerRange("b"+(binaryLength-back));

back -=1;

}

cond.addBase("b"+(binaryLength-i));

}

}

System.out.println("query cond: "+cond);

}

public static void main(String[] args) {

for(int i=0;i<32;i++){

query(i, 6);

}

}

}

打印的结果

query: upper is 000110

RangeBitmapDemo2.QueryCond(base=[b6, b5, b4, b1], lowerRange=[[b6, b5, b4, b2], [b6, b5, b4, b3]])

表示000110的组合关系为(b6 & b5 & b4 & b3 & b2 & b1) or (b6 & b5 & b4 & b2) or (b6 & b5 & b4 & b3) 即整个结果由3个部分组合而成。