2019 级

《物联网数据存储与管理》课程

课 程 报 告

姓　名 张佳辉

学　号 U201916341

班　号 计算机校交 1902 班

日　期 2022.04.14

目　　录

一、目标概述

1. 初步了解 Bloom Filter，明白其结构和原理。
2. 基于理论，分析 Bloom Filter 的 false positive 的概率。
3. 基于 Bloom Filter 的操作流程，实现一个能保证和实现所提出的设计目标的建议 Bloom Filter。
4. 基于传统 Bloom Filter，实现多维度属性数据和索引的 Bloom Filter。

二、初入 Bloom Filter

2.1 布隆过滤器简介

布隆过滤器（英语：Bloom Filter）是 1970 年由布隆提出的。它实际上是由一个很长的 bit 数组和一系列哈希函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。其结构示意图如图 1。

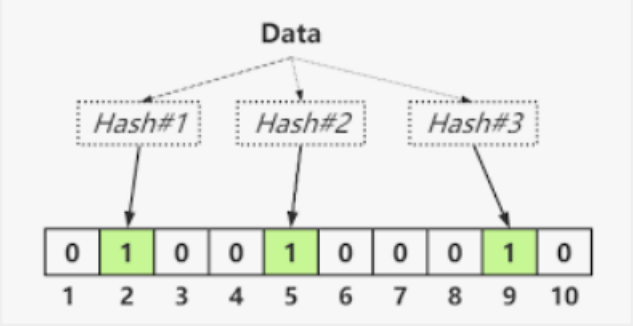

图 1 布隆过滤器示意图

2.2 布隆过滤器原理

布隆过滤器的原理是，当一个元素被加入集合时，通过 K 个哈希函数将这个元素映射成 K 个不同的索引，然后将 bit 数组中对应的 K 个不同索引的值置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了：如果这些点有任何一个 0，则被检元素一定不在；如果都是 1，则被检元素很可能在。这就是布隆过滤器的基本思想。

2.3 布隆过滤器的假阳性

在原理中，提到我们通过看对应 bit 位的值是否位 1 就能大约知道集合中是否存在所查询的元素，那么为什么是大约知道，而不是准确知道呢？

其实这个误差主要是由于哈希冲突所导致的，简单来说就是不同的元素所产生的哈希值相同。

假设一个元素 A 已经在集合中，其哈希之后对应的 bit 位是 1，3，6，而这时我想要查询的元素 B（不在集合中），其哈希之后对应的 bit 位也是 1，3，6，在这种情况下，布隆过滤器判断元素 B 在这个集合中，因此会产生假阳性（false positive）。

三、Bloom Filter 理论分析

3.1 False positive 分析

假设布隆过滤器由 m 位的比特数组以及 n 个元素组成，分析如何选定哈希函数的个数 k，能够使得布隆过滤器的假阳性概率最低。（假设对于哈希函数而言，映射到 0 至 m-1 的概率是相等的。

当在进行一次元素插入集合中的时候，对于任一一个 bit 位而言，被置 1 的概率是

$$\frac{1}{m}$$

因此没有被置为 1 的概率是

$$1-\frac{1}{m}$$

那么在进行 k 次不同 hash 之后，依旧没有被置为 1 的概率是

$$\left(1-\frac{1}{m}\right)^{k}$$

插入 n 个元素之后，对于任一一个 bit 位，为 1 的概率为

$$1-\left(1-\frac{1}{m}\right)^{kn}$$

因此对于一个不存在与集合中的元素，被误判断为在集合中的概率为

$$p=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}\approx\left(1-e^{-\frac{kn}{m}}\right)^{k}$$

使得错误率最小，对于给定的 m 和 n，当 $k=\frac{m}{n}*ln2$ 时候取值最小。

3.2 Bit 数组大小分析

在给定 n（集合中元素的个数）和错误率（最优函数个数 k 的的错误率）的情况下，位数组 M 的大小计算，在最优 k 的情况下

$$p=\left(1-e^{-\frac{kn}{m}}\right)^{k}$$

经过简单的代入和转换可以得到

$$m=-\frac{(n*lnp)}{(ln2)\text{^}2}$$

四、Bloom Filter 操作流程

传统的布隆过滤器的主要操作包括了增加元素，查询元素，以及清空过滤器。而主要组成部分包括 k 个哈希函数，以及一个 m 位的 bit 数组。

当需要对存储进行插入操作的时候，先对布隆过滤器进行插入操作，然后在对存储进行插入操作，流程如图 2 所示。

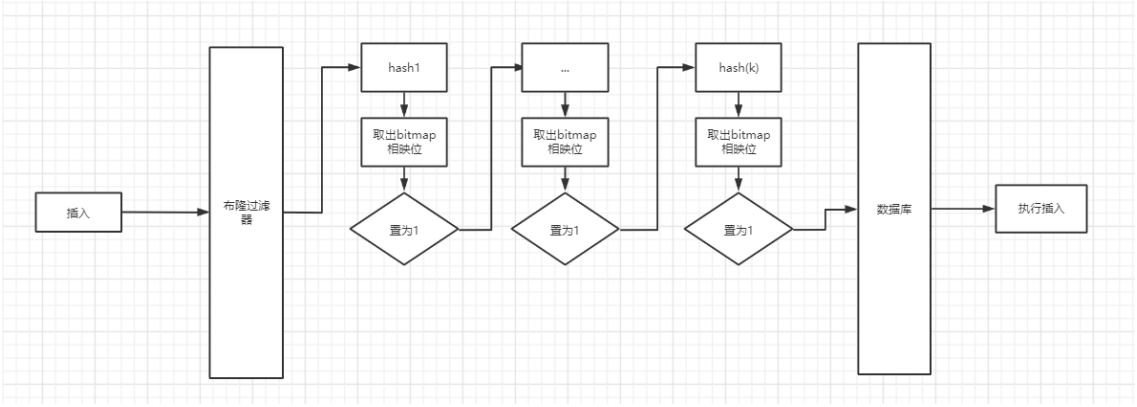

图 2 插入操作流程

当需要对存储进行某个元素的查询的时候，先在布隆过滤器中进行查询操作，如果过滤器判断不存在，直接返回不存在，如果判断存在，则再到存储里面取查询，流程图如图 3 所示。

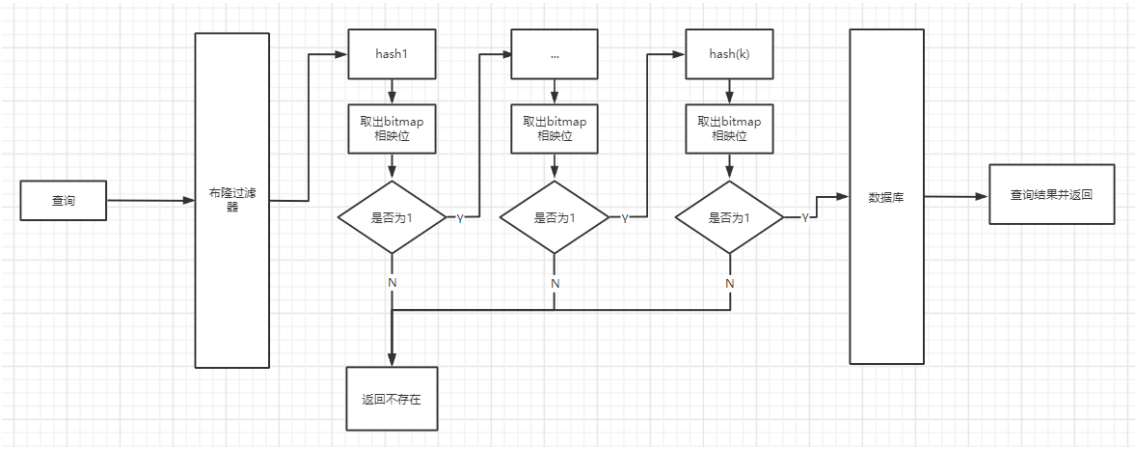

图 3 查询操作流程

五、Bloom Filter 简易实现

5.1 结构设计

设计核心思想就是对于一个 Bloom Filter 类，我们根据给定的容纳的最大元素个数以及最大的错误率，运用第四节中的两个公式来计算出 bit 数组的大小以及 hash 函数的个数，来使得 false positive 最小。

而 Bloom Filter 类中有的核心方法有：加入元素，查询元素，清空过滤器这三个方法。

因此可以根据上述思想来构造一个建议的 Bloom Filter，代码详见附件。

5.2 测试结果

这里我随机生成了 20w 个不重复的字符串作为测试用例，其中 15w 用作存入数据库的数据，剩余 5w 作为查询数据，值得注意的是，在存入数据的时候，bloom filter 会先判断数据是否在数据库中，如果再，则不存入，这里也会有一个 false positive 的存在；而对于查询数据，本来这 5w 的数据就不再数据库中，因此只要知道过滤器判断多少个元素在存储中，即可计算出 false positive 的值。而另外一点，false positive 的值也同样取决于 hash 函数的选择以及 hash 种子的选择。以下是几组测试数据。

图 4 简易 bloom filter 测试结果

六、基于 Bloom Filter 的多维数据属性表示和索引

6.1 SBF 与 PBF

传统布隆过滤器 SBF（Standard Bloom Filter）能够高效地判断一个只具有一维属性的元素在不在一个集合中。常见的 SBF 的结构有两种，一种是多个 hash 函数共享一个 bit 数组，称作 flat form；另一种是多个 hash 函数分别对应一个 bit 数组，而几个 bit 数组的总位数其实和前者是相同的，称作 segment form。结构示意图如图 5。但是传统布隆过滤器存在的主要问题是它并不支持多维度属性的元素的判断

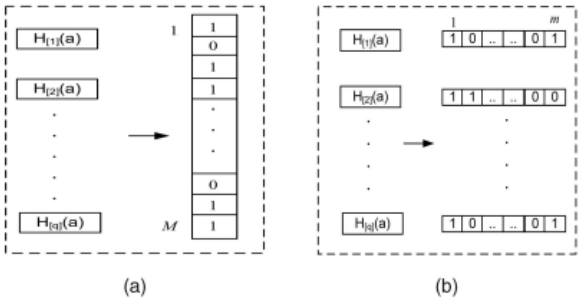

图 5 SBF 两种结构示意图

因此需要一种改进结构，能让布隆过滤器支持多维度属性元素的判断，而这个结构被称为并行布隆过滤器 PBF（Parallel Bloom Filter）。

6.2 PBF 的结构

PBF 解决的主要问题就是多维度属性的元素的查询，而其核心思想就是针对不同的属性，建立多个布隆过滤器。

假设目标元素 a 具有 p 个属性，那么 PBF 可以由 p 个 SBF 来构成，每个 SBF 代表一个属性。而每个 SBF 由 q 个 hash 函数以及 bit 数组组成其结构示意图如图 6。

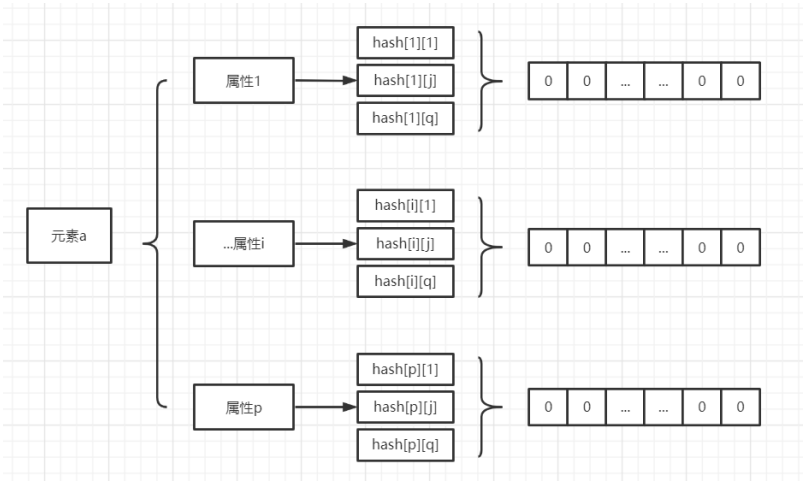

图 6 PBF 结构示意图

6.3 PBF 操作流程

对于 PBF 的插入和查询流程，其实和 SBF 大同小异，用计算机语言来表示如下所示：

插入：

Insert_Item (Input: Item a)

```
for (i = 1; i ≤ p; i++) do
    for (j = 1; j ≤ q; j++) do
        k = H[i][j](a_i)
        C[i][j][k] =1
    end for
end for
```

查询：

Membership_Query_Item (Input: Item a)

```
for (i = 1; i ≤ p; i++) do
    for (j = 1; j ≤ q; j++) do
        k = H[i][j](a_i)
        if C[i][j][k] == 0 then
            Return False
        end if
    end for
end for
Return True
```

6.4 PBF 的 false positive 分析

PBF 和 SBF 一样，也有着一定的可能产生误判。

举个例子，假如我们存储的对象是桌子，每张桌子有两个属性，尺寸和颜色。

现在存储中有两张桌子，一张桌子的尺寸是 large，颜色是 red；另外一张的尺寸是 small，颜色是 green。这时候，我们通过 PBF 查询一张尺寸是 large，颜色是 green 的桌子，PBF 会认为这张桌子在存储中，因为 PBF 中每个属性对应的 SBF 返回的都是 true，但事实是存储中并没有这么一张桌子。这时候就产生了 false positive。而对于具体产生 false positive 的概率，主要有两部分组成。

一部分是由于 SBF 自身的 false positive，概率等于每个 SBF 的 false positive 概率相乘。假设每个 SBF 的 FP 相同，那么 PBF 的$FP = \left(1 - e^{-\frac{n}{m}}\right)^{pq}$

另一部分是由于像上述 case 一样，多维度属性的交叉查询而产生的 false positive。

6.5 PBF 的结构改进 PBF-HT

为了避免上述第二种情况所产生的 false positive，需要对 PBF 原本的结构进行一个优化。

这里引入一个新的 hash table 用于对属性进行校验。因此改进后的布隆过滤器由一个 HT（hash table）和一个 PBF 组成，因此这种新结构命名为 PBF-HT。

引入这个 HT 的核心就是解决 PBF 在写入过程中，每个属性都相互独立而丢失了相互的联系（dependency），而这个 HT 的作用就是取确认所查询到的每个属性是否属于存储中的同一个元素。而在 PBF-HT 进行多维度属性元素查询的时候，相比较原始 PBF，额外多了一个条件，就是 HT 的校验通过。

HT 的主要思想是：对于元素中的每个属性 i，都会有一个校验值 v[i]，那么对于这个元素来说，它的校验值等于每个属性的校验值之和。当我们查询一个元素的时候，即使 PBF 返回了 true，HT 还要根据这个元素的校验值，去查询 HT 中存不存在这么一个校验值所对应的元素。这样通过两步校验的方式来降低了 false positive 的可能性。

因此 PBF-HT 的结构图如下图所示。

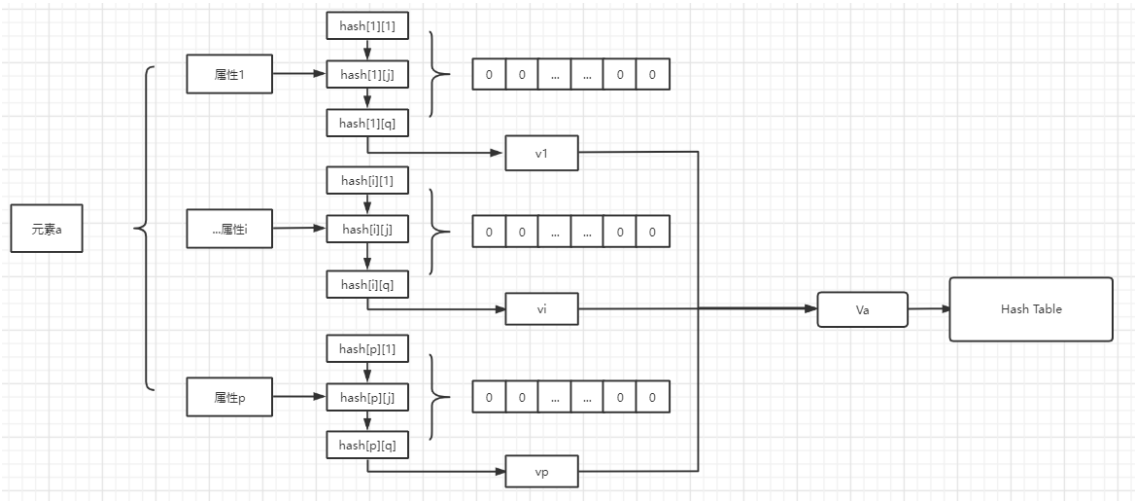

图 7 PBF-HT 结构示意图

6.6 HT 的设计

明白了 HT 的主要功能，这里就涉及到如何将一个属性映射成一个比较合适的校验值 v。即 $v_i = F(H_{[i][j]}(a_i))$ 中，F 这个方法的定义。这里简单介绍两种较为常见的方法。

第一种是直接利用每个属性中 q 个 hash 函数所得出的 hash 值之和。这种方法的缺点在于容易产生校验和相同从而导致产生校验冲突。对于属性 i 而言，有

$$v_i = \sum_{j=1}^{q} H_{ij}(a_i)$$

第二种是基于第一种的改进，对于不同的 hash 函数，我们赋予不同的权重，从而减少校验冲突。对于属性 i 而言，有

$$v_i = \sum_{j=1}^{q} \frac{H_{ij}(a_i)}{2^j}。$$

6.7 PBF-HT 的 False Positive 分析

由于 PBF-HT 采用两步校验的方式，因此 False Positive 的概率为：

$$f_{PBF-HT} = f_{PBF} * f_{HT}$$

$$f_{PBF} = (f_{SBF})^p \approx \left(1 - e^{-\frac{n}{m}}\right)^{pq}$$

经查阅资料，有

$$f_{HT} \leq 2\Phi\left(\frac{n}{2\sigma'}\right) - 1 = 2\Phi\left(\frac{\sqrt{3}2^q n}{p(m-1)(2^q-1)}\right) - 1,$$

where $\Phi(*)$ stands for the standard Normal Distribution. □

由此可以得出 PBF-HT 的 False Positive 概率。

6.8 实现简易的 PBF-HT

PBF-HT 的简易版的实现并没有想象中的那么困难，实际上只需要维护一个 SBF 数组和一个 HashMap 即可。

在插入元素的时候，需要对每个属性进行一次校验值的计算，在每个属性插入完成后，将这个元素的校验值放入到 HashMap 中，并设置值为 true。

在查询元素的时候，需要对每个属性进行一次查询，如果一个属性的 SBF 的 contain 返回 false，那就表明这个元素不存在，直接返回 false 即可，在对属性进行查询的同时，还要进行每个属性的校验值计算，在所有的 SBF 的 contain 都返回 true 之后，计算元素的校验值，并在 HashMap 中查询是否存在该值，如果存在，则返回 true，不存在返回 false。

该类的方法和逻辑如下图所示。

```
4   class PBF_HT {
5     SBFArr; // the SBF of each property
6     HT; // hash table
7     p;  // the number of properties
8     n;  // the max capacity of each SBF
9     error;  // the error rate
10    constructor(p, n, error){
11      this.p = p;
12      this.n = n;
13      this.error = error;
14      this.initSBFArr();
15      this.initHT();
16    }
17
18  >   initSBFArr(){···
23    }
24
25  >   initHT(){···
27    }
28
29  >   add(el){···
37    }
38
39  >   contain(el) {···
53    }
54
55  >   getHT(){···
57    }
58  }
```

图 8 PBF-HT 代码

在编写完了代码，进行一个简单的测试用来检验 PBF-HT 的效果如何。测试用例如下：

```
60    const PBF_HT_INSTANCE = new PBF_HT(2, 10, 0.01);
61
62    PBF_HT_INSTANCE.add(['large','red'])
63    PBF_HT_INSTANCE.add(['small','green'])
64    console.log(PBF_HT_INSTANCE.contain(['large','red']));
65    console.log(PBF_HT_INSTANCE.contain(['small','green']));
66    console.log(PBF_HT_INSTANCE.contain(['large','green']));
67    console.log(PBF_HT_INSTANCE.contain(['small','red']));
68    console.log(PBF_HT_INSTANCE.getHT());
```

图 9 测试用例

结果如下图所示：

```
[Running] node "d:\garfield\study\junior_spring\big_data_management\course_code\PBF-HT.js"
true
true
false
false
Map(2) { 681 => true, 628 => true }
```

图 10 测试结果

七、课程感悟

这次课程报告对我有很深刻的影响，不仅让我阅读了很多国外论文，让我不断贫瘠的英文水平不断恢复；更重要的是，让我对大数据存储和处理有了更深刻的了解。如今我们所处的时代，一天的信息量就可以超过过去几百年甚至几千年信息量的总和，如果有效并且高效地管理和存储这些数据是当今面临的重大难题。通过这次课程总结报告，我对 Bloom Filter 有了深入的理解，但我认为在课程内学到的东西永远都是浅尝辄止，从个人的发展的角度来看，如果有意愿从事于大数据相关行业，不应该限制于课内所学，更亲身实践，自己手撸代码，只有徒手造轮子，才能快速地提升自己的水平，并且真正落实学以致用。

参考文献

[1] https://hardcore.feishu.cn/docs/doccntUpTrWmCkbfK1cITbpy5qc#
[2] https://blog.csdn.net/liebert/article/details/79737042
[3]https://zh.wikipedia.org/wiki/%E5%B8%83%E9%9A%86%E8%BF%87%E6%BB%A4%E5%99%A8
[4] B. Xiao and Y. Hua, “Using Parallel Bloom Filters for Multi- Attribute Representation on Network Services,” IEEE Trans. Parallel and Distributed Systems, vol. 21, no. 1, pp. 20-32, Jan. 2010.