2019 级

《大数据数据存储与管理》课程

课 程 报 告

姓　名　杨超淇

学　号　U201914974

班　号　计算机 1903 班

日　期　2022.04.18

目　　录

一、报告选题和目的

1. 课程报告选题为基于 Bloom Filter 设计的大数据存储查询系统。
2. 了解 bloom filter 的应用背景，掌握其基本原理。
3. 尝试基于 bloom filter 的存储结构开发大数据存储系统。
4. 通过实验分析 bloom filter 的优势以及缺陷。

二、选题分析

2.1 选题背景

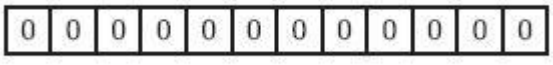
Bloom Filter 是 1970 年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数，可以用于检索一个元素是否在一个集合中。

Bloom Filter 是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter 的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

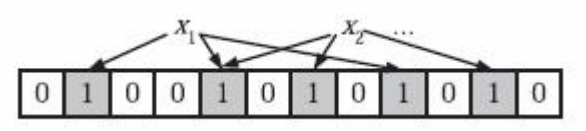
如需要判断一个元素是不是在一个集合中，我们通常做法是把所有元素保存下来，然后通过比较知道它是不是在集合内，链表、树都是基于这种思路，当集合内元素个数的变大，我们需要的空间和时间都线性变大，检索速度也越来越慢。Bloom filter 采用的是 hash 函数的方法，将一个元素映射到一个 m 长度的阵列上的一个点，当这个点是 1 时，那么这个元素在集合内，反之则不在集合内。这个方法的缺点就是当检测的元素很多的时候可能有冲突，解决方法就是使用 k 个 hash 函数对应 k 个点，如果所有点都是 1 的话，那么元素在集合内，如果有 0 的话，元素则不在集合内。

2.2 原理分析

Bloom Filter 使用一串位数组存储数据，并只支持对数据进行查询，无法读取。初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0。



为了表达 S={x1, x2,…,xn}这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的 hash 函数，它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素 x，第 i 个 hash 函数映射的位置 hi(x)就会被置为 1（1≤i≤k）。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个 hash 函数选中同一个位置（从左边数第五位）。



在判断 y 是否属于这个集合时，我们对 y 应用 k 次 hash 函数，如果所有 hi(y)的位置都是 1（1≤i≤k），那么我们就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。下图中 y1 就不是集合中的元素。y2 或者属于这个集合，或者刚好是一个 false positive。

错误率估计：

前面已经提到，Bloom Filter 在判断一个元素是否属于它表示的集合时会有一定的错误率，下面就来估计错误率的大小。在估计之前为了简化模型，我们假设 kn<m 且各个 hash 函数是完全随机的。当集合 S={x1, x2,⋯,xn}的所有元素都被 k 个 hash 函数映射到 m 位的位数组中时，这个位数组中某一位还是 0 的概率是：

$$p = (1 - \frac{1}{m})^{kn}$$

其中 1/m 表示任意一个 hash 函数选中这一位的概率，(1-1/m)表示 hash 一次没有选中这一位的概率。要把 S 完全映射到位数组中，需要做 kn 次 hash。某一位还是 0 意味着 kn 次 hash 都没有选中它，因此这个概率就是（1-1/m）的 kn 次方。为了简化运算，可以使用计算 e 时常用的近似：

$$\lim_{x\to\infty}(1 - \frac{1}{x})^{-x} = e$$

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

令 ρ 为位数组中 0 的比例，则 ρ 的数学期望 E(ρ)=p。在 ρ 已知的情况下，错误率（即不属于集合的数据的 hash 选中位均为 1 的概率）为：

$$(1-\rho)^k \approx (1-p)^k$$

(1-ρ)为位数组中 1 的比例，(1-ρ)k 就表示 k 次 hash 都刚好选中 1 的区域，即错误率。p 只是 ρ 的数学期望，在实际中 ρ 的值有可能偏离它的数学期望值。位数组中 0 的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。将 p 代入上式中，得：

$$(1-p)^k \approx (1-e^{-kn/m})^k$$

上式即可用于估算 Bloom Filter 的错误率。

三、实验环境

操作系统：Windows 10 家庭中文版

处理器：Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz

内存：16GB

开发平台：IntelliJ IDEA 2020.3.2 x64

开发工具：java jdk-15.0.2

四、实验设计

本次实验目的为开发基于 bloom filter 的数据存储、查询系统，实验程序使用 JAVA 语言实现。实验程序分为 3 个类：

bloomfilter 类实现对字符串数据的存储和查询功能；

simplehash 类用于根据随机种子生成不同的 hash 函数对象；

benchmark 类用于对 bloomfilter 与 JAVA 自带的 hashset 进行性能测试和对比。

4.1 hash 函数设计

由于用于实现 bloom filter 的 hash 函数数量 k 不确定，且每个 hash 函数生成的 hash 码要尽可能不同，因此使用 simplehash 类统一生成 hash 函数对象。同时在构造 simplehash 对象时，传入一个随机数（0~100）seed 用于区分不同的 hash 函数。

simplehash 类中统一实现的 hash 函数如下图所示。其中 m 为构造对象时，传入的位数组长度参数。hash 函数对传入的字符串数据逐一获取字符的 Unicode 编码，并与随机种子 seed 进行循环运算，得到 hash 编码。由于 hash 编码的值可能为负数或超出位数组的范围，最后需要求绝对值并取余，将 hash 码约束在 0~m 的范围内。

```
public int hash(String data){
    int code = 0;
    for(int i = 0; i < data.length(); i++){
        code = code * this.seed + (int)data.charAt(i);
    }
    return Math.abs(code) % m;
}
```

4.2 bloom filter 设计

bloom filter 类为基于 Bloom Filter 的技术设计的存储、查询系统，其具体实现的数据结构、构造函数、数据存储方法、数据查询方法如下图所示。

数据结构：由一个长度为 m 的位数组（通过实例化 BitSet 类得到）和 k 个 hash 函数（通过实例化 simplehash 类得到）组成。为了实现 bloom filter 的通用性，位

数组的长度 m 和 hash 函数数量 k 不能够为常量，而是在构造 bloom filter 对象时作为参数传入。由于每个 hash 函数的种子不同，且随机种子应该由调用 bloom filter 类的使用者决定，所以参数 k 实际上是由随机数组成的数组 seeds。

构造函数：在构造 bloom filter 对象时，根据传入的 m 和 seeds，修改对应的数据成员。然后，根据位数组长度 m 构建 BitSet 对象，使用成员 bits 引用；根据随机数组 seeds 和 hash 码范围 m 构建 k 个不同的 simplehash 对象组成数组，使用成员 hashs 引用。

add 方法：该方法实现 bloom filter 的存储功能。获取需要存储的字符串对象后，在 k 次循环中，使用 hashs 数组内不同的 simplehash 对象的 hash 方法解析字符串，得到 hash 码作为索引值，使用 Bitset 对象的 set 方法将数据成员 bits 中的对应位置为 true。

contains 方法：该方法实现 bloom filter 的查询功能。获取需要查询的字符串对象后，在 k 次循环中，使用 hashs 数组内不同的 simplehash 对象的 hash 方法解析字符串，得到 hash 码作为索引值，使用 Bitset 对象的 get 方法获取数据成员 bits 中对应位置的值，若该值不为 true，则查询失败，返回 false。当循环结束时，则查询成功，返回 true。

```
public class bloomfilter {
    private int m;
    private int[] seeds;
    private simplehash[] hashs;
    private BitSet bits;

    bloomfilter(int m,int[] seeds){
        this.m = m;
        this.bits = new BitSet(m);
        this.seeds = seeds;
        this.hashs = new simplehash[this.seeds.length];
        for(int i = 0; i < this.seeds.length; i++){
            this.hashs[i] = new simplehash(this.m,this.seeds[i]);
        }
    }

    public void add(String data){
        for(simplehash h : this.hashs){
            this.bits.set(h.hash(data),true);
        }
    }

    public boolean contains(String data){
        for(simplehash h : this.hashs){
            if(!this.bits.get(h.hash(data))) return false;
        }
        return true;
    }
}
```

五、性能测试

5.1 数据集与内存占用

实现 simplehash 类以及 bloom filter 类的设计后，数据存储查询系统基本上设计完成。在 benchmark 类中设置程序入口 main，对该数据存储系统进行性能测试，并将其与 java 自带的 hashset 类进行性能对比。

测试使用数据集为计算机学院 2019 级新生名单（共 329 个学生名字），以及用于测试错误率的校交班学生名单（共 60 个学生名字），两个数据集交集为空集（除去重名），分别保存在 student0.txt 和 student1.txt。

benchmark 测试程序中使用 BufferReader 对象 br0、br1 读取数据集文件，获得字符串对象。构造 bloom filter 对象时，设置 k 为 8，通过 Random.nextInt(100)获取 100 以内的随机数作为 simplehash 的随机种子。

内存占用：使用对应的 add 方法将 br0 读取的数据存入 bloom filter 和 hashset。bloom filter 在 k=8、n=329 的情况下，要求错误率达到 0.01，需要的位数组长度在 3200 左右，即占用 400 字节的内存空间。hashset 类则存储完整的字符串数据，java 字符集使用 unicode 编码，则存储 927 个字符至少需要占用 1854 字节的内存。

5.2 错误率分析

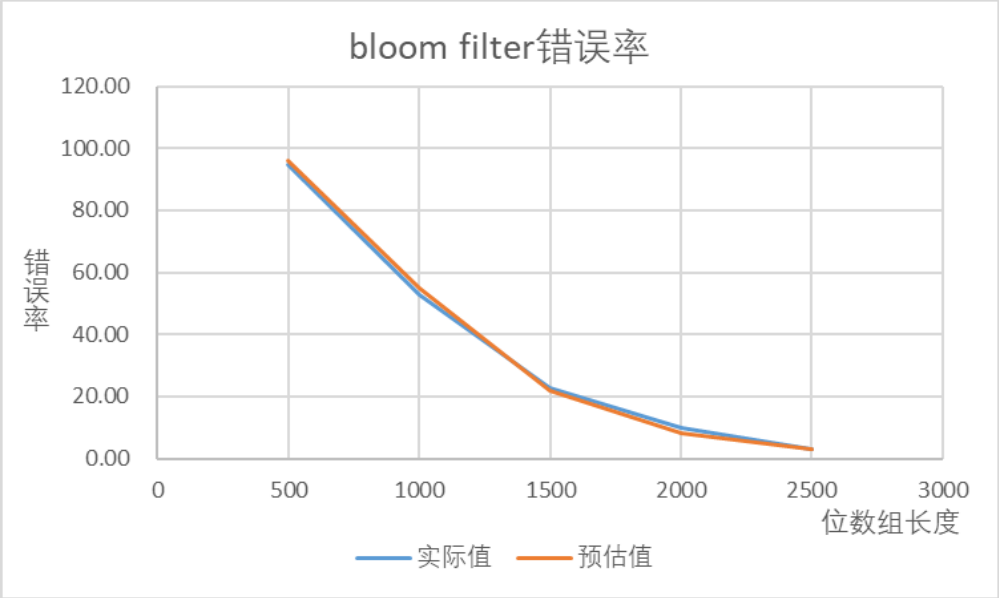
使用 BufferReader 对象 br1 读取 student2 数据集，该数据集与已存储的 student1 没有交集（可能有重名），可以用来测试 bloom filter 的错误率。

hashset 的错误率为 0，可以用来去除重名。在读取数据的循环中，通过 contains 方法判断出数据不在 hashset 内时，total++，再判断数据在 bloom filter 中时，wrong++ 得到 bloom filter 错误率为 wrong/total。

为了数据的准确性，以及分析 bloom filter 的错误率与位数组长度的关系，将 m 在 500~2500 之间变化，并测试 10 次取平均值。

测试次数	bloom filter错误率（位数组长度）				
	500	1000	1500	2000	2500
1	98.27	50.00	20.68	10.34	0.00
2	98.27	58.62	25.86	12.06	3.44
3	82.75	36.20	24.13	15.51	3.44
4	93.10	60.34	13.79	8.62	0.00
5	91.37	67.24	18.96	1.72	1.72
6	100.00	46.55	29.31	6.89	5.17
7	98.27	63.79	22.41	13.79	3.44
8	94.82	53.44	20.68	13.79	5.17
9	91.37	48.27	27.58	8.62	3.44
10	98.27	44.82	25.86	6.89	3.44
平均值	94.65	52.93	22.93	9.82	2.93
预估值	95.93	55.03	21.89	8.22	3.23

根据上述表格画出 bloom filter 错误率随位数组长度变化的折线图。由折线图可以看出，性能测试的错误率变化基本与预测值吻合。



5.3 查询延迟对比

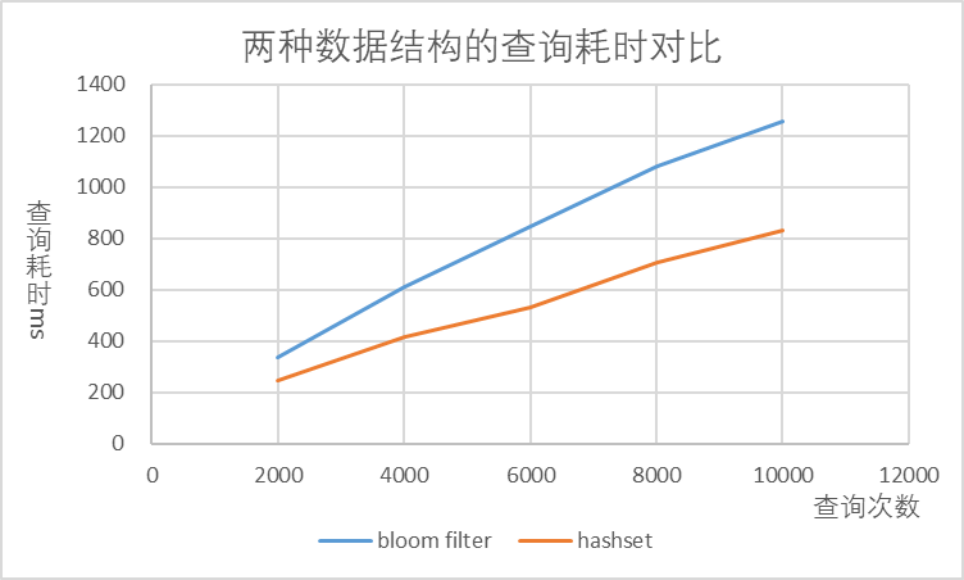
使用 student0 数据集进行数据查询，由于 student0 数据集已经存储在 bloom filter 中，因此查询耗时为最大值，即需要经过所有 hash 函数的检验。同时也对 hashset 进行数据查询，比较二者的查询性能。

计算查询耗时为在查询开始与结束时，使用 System.currentTimeMillis()方法获取系统当前时间，记为 starttime 和 endtime，通过两者相减得到数据查询循环消耗的时间（毫秒级）。

由于单次查询的时间极小，且为了求出耗时变化趋势，查询次数在 2000~10000 之间变化，每次查询整个数据集，重复测试 10 次求平均值。

测试次数	bloom filter查询耗时（查询次数）					hashset查询耗时（查询次数）				
	2000	4000	6000	8000	10000	2000	4000	6000	8000	10000
1	297	588	796	994	1187	234	459	521	644	832
2	328	639	781	1024	1197	250	375	549	657	795
3	312	641	885	988	1275	265	421	534	640	834
4	328	609	869	1073	1260	245	422	528	712	821
5	344	669	992	1604	1192	234	432	561	1091	870
6	313	547	791	970	1306	234	390	544	671	826
7	375	578	857	1035	1270	266	406	524	674	801
8	328	609	856	1035	1410	259	406	508	676	904
9	370	619	781	1063	1239	234	406	535	662	788
10	344	594	837	1049	1213	234	453	507	657	851
平均值	334	609	845	1084	1255	246	417	531	708	832

下图为根据上述数据画出的 bloom filter 和 hashset 的查询耗时随查询次数的变化，以及两者查询耗时的对比。可以看出，由于 bloom filter 没有进行优化，且需要经过多个 hash 函数计算才能得到查询结果，其查询耗时性能不如 Java 自带的 hashset 存储结构。



六、实验总结

通过性能测试与对比，分析可得 Bloom Filter 与通常的 hash 表相比，主要优点在于其所占用的内存空间非常小，代价是由于数据并非真的被存储在 Bloom Filter 内，存在一定的错误率，也无法读取数据。在查询性能方面，随着数据的增多，通常的存储结构会越来越复杂，导致查询耗时增加，而 Bloom Filter 本身结构简单，在大数据情况下查询效率更高。

总的来说，Bloom Filter 有着广阔的应用前景，例如网页 URL 的去重，垃圾邮件的判别，集合重复元素的判别，查询加速，数据库防止查询击穿等，借助 Bloom Filter 可以大幅度节省空间资源、提高效率。

参考文献

• F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyo nd Bloom Filters: From Approximate Membership Checks to Approximate State Machines,” Proc. ACM SIGCOMM, 2006.

• Y. Zhu and H. Jiang, “False Rate Analysis of Bloom Filter Replicas in Distri buted Systems,” Proc. Int’l Conf. Parallel Processing (ICPP ’06), pp. 255-262, 20 06.

• S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, “Longest Prefix Matchi ng Using Bloom Filters,” Proc. ACM SIGCOMM, pp. 201-212, 2003.

• L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary Cache: A Scalable Wi de-Area Web Cache Sharing Protocol,” IEEE/ACM Trans. Networking, vol. 8, no. 3, pp. 281-293, June 2000.

• B. Xiao and Y. Hua, “Using Parallel Bloom Filters for Multi- Attribute Repre sentation on Network Services,” IEEE Trans. Parallel and Distributed Systems, v ol. 21, no. 1, pp. 20-32, Jan. 2010.

• Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Scalable and Adaptive Met adata Management in Ultra Large-scale File Systems,” Proc. 28th Int’l Conf. Dis tributed Computing Systems (ICDCS ’08), pp. 403-410, 2008.

• D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and Network Application of Dynamic Bloom Filters,” Proc. IEEE INFOCOM, 2006.