大数据存储与管理

基于 Bloom Filter 的设计

布隆过滤器（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。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.

布隆过滤器的主要作用就是 **"以更低的空间效率检索一个元素是否存在其中"**

数据结构

关键因子

- bloom filter calculate
- `n` 是过滤器预期支持的元素个数
- `m` 是过滤器位数组的大小，即该过滤器总共占用多少 bit 的空间
- `c` 是每个元素平均占用的空间
- `p` 是假阳性概率，即 fpp (false positive probability)
- `k` 哈希函数的个数

计算公式

- $m = -nlog(p)\frac{1}{log(2)^2}$
- $c = \frac{m}{n}$
- $p = (1 - e^{\frac{-kn}{m}})^k = (1 - e^{\frac{-k}{\frac{m}{n}}})^k = (1 - e^{\frac{-k}{c}})^k$
- $k = \frac{m}{n} * ln2 = 0.7\frac{m}{n} = 0.7c$

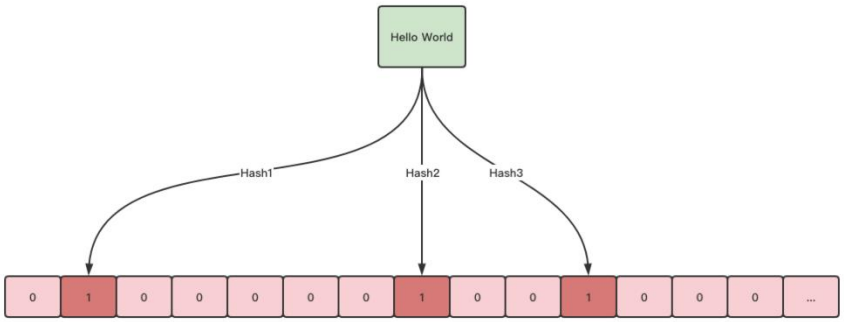
```
n = ceil(m / (-k / log(1 - exp(log(p) / k))))
p = pow(1 - exp(-k / (m / n)), k)
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));
k = round((m / n) * log(2));
```

通常创建一个布隆过滤器，会需要用户提供想支持的 n 和预期的 p，然后通过公式得到其余最佳的参数

结构设计

布隆过滤器的数据结构其实也挺简单，如果之前有了解过位图的话，那么就更简单了。整个布隆过滤器的底层实现就是基于 **“位数组 + 哈希函数”** 实现的

假设某个布隆过滤器的位数组大小为 1000 bits，即 1000 个位。由 3 个 hash functions 组成



当我们往该布隆过滤器传入一个元素 (Hello World) ，那么该元素就会经过 3 个 hash functions 计算，得到 3 个 hash 值，最后哈希值经过模运算映射到位数组的具体第 x 位上，并标记为 1

当我们要判断某个元素是否在布隆过滤器时，同理，也是将该元素传入，经过哈希函数计算，并得到映射位数组的索引值，判断这个几个索引的位是否为 1， 如果都为 1 则代表该元素已经存在了，但只要有一个位置不为 1，我们就认为该元素并不存在

代码实现

结构体

布隆过滤器结构主要可以由三部分构成，**关键因子 (n,m,p,k…)**，**哈希函数**，**位数组**，所以假设我们要实现一个布隆过滤器，首先就要将以上三部分内容构建好，这里写一份伪代码，基本可以简单的体现出布隆过滤器的结构设计

```
public abstract class BloomFilter {

    // 字节数组

    byte[] bytes;

    // 关键因子

    int n;

    double p;

    int m;

    int k;

    // 哈希函数

    List<Hash> hashes;

    // 插入元素

    public void put(byte[] bs);

    // 是否存在

    public boolean contains(byte[] bs);

}
```

因子计算

- 通过 (n,p) 计算最佳的 m

```
public static int optimalNumOfBits(int n, double p) {

        if (p == 0) {

            p = Double.MIN_VALUE;

        }
```

```
        return (int) (-n * Math.log(p) / (Math.log(2) * Math.log(2)));

    }
```

- 通过 (n,m) 计算最佳的 k

```
        public static int optimalNumOfHashFunctions(int n, int m)
{

            return Math.max(1, (int) Math.round((double) m / n *
Math.log(2)));
    }
```

- 通过 (n,m,k) 计算最佳的 p

```
    public static double optimalFpp(int n, int m, int k) {
        // keep 5 digits after the decimal point
            int point = 100000;
            return (double) Math.round((point * Math.pow(1 -
Math.exp((double) (-k * n) / m), k))) / point;
        }
```

- 通过 (n,m) 计算 c

```
        public static double bitsOfElement(int n, int m) {
            return (double) m / n;
        }
```

- 通常我们可以通过用户传递的 n 和 p，构造出一个预期的布隆过滤器

哈希函数

哈希函数可以采用各种混合的哈希算法去搭配，我这件为了简单实现，建议采用 murmur3 hash，通过加不同的 seed 去实现不同的哈希效果

```
    // 成员变量 k 个哈希函数
        private List<Hash> hashes;

        // 获得 k 个索引值
        public int[] hashes(byte[] bs) {
            int[] indexs = new int[hashes.size()];
```

```
            for (int i = 0; i < hashes.size(); i++) {
                //
                int index = (int) ((hashes.get(i).hashToLong(bs) & Long.MAX_VALUE) %
this.m);
                indexs[i] = index;
            }
            return indexs;
        }

        // 哈希接口
        public interface Hash {
            long hashToLong(byte[] bs);
    }
```

插入元素

```
        // 插入元素&是否存在
        public synchronized void put(byte[] bs) {
            int[] indexs = hashes(bs);
            for (int index : indexs) {
                // 所在的具体字节
                byte bits = (byte) (index / B);
                // 所在的具体位
                byte offset = (byte) (1 << (B_MASK - (index % B)));
                bytes[index / B] = (byte) (bits | offset);
            }
    }
```

B = 8，代表一个字节 8 位; B_MASK 是掩码, B_MASK = B - 1，为了做取模运算

首先得到 index，通过 index / B 得到 index 所在那个 byte

其次通过 index % B 得到所在该 byte 的具体位索引，比如等于 2, (00100000，一个 byte 8 位，1 所在的位置就是 2)

通过 index % B 知道了具体的位索引后，为了将该位置变成 1，那么我们就需要得到位加法的偏移量，即得到 00100000 去做加法

怎么得到那？即把 00000001 左移 B_MASK - (index % B) = 7 - 2 = 5 位，左移 5 位后，得到 00100000

假设所在字节为 10000001，我们要将索引为 2 的位置变为 1， 则两个 byte 做或运算，即位加法即可 10000001 | 00100000 = 10100001

最后覆盖 index 所在的位即可

是否存在

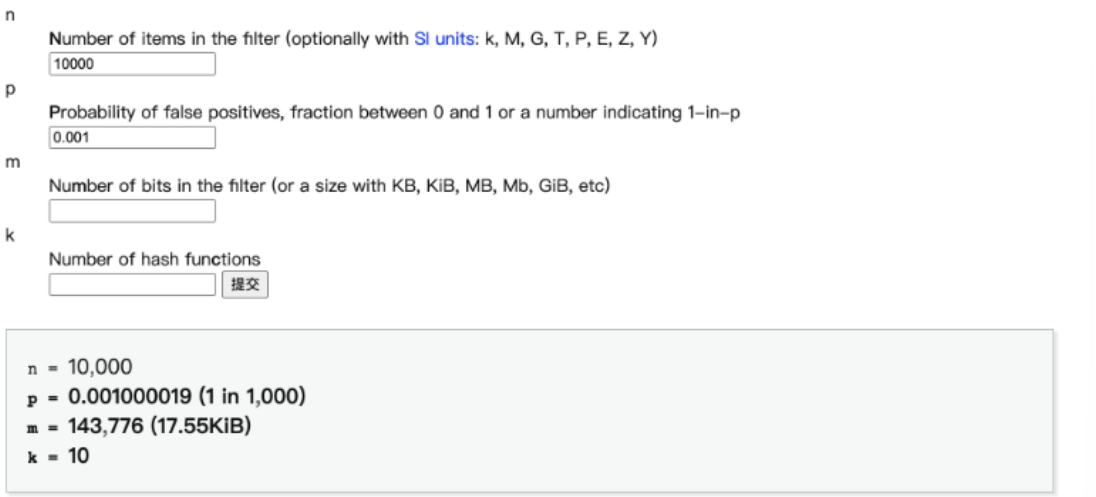
```
        // 是否存在
        public synchronized boolean contains(byte[] bs) {
            int[] indexs = hashes(bs);
```

```
        for (int index : indexs) {
            byte bits = (byte) (index / B);
            byte offset = (byte) (1 << (B_MASK - (index % B)));
            if ((bits & offset) == 0) {
                return false;
            }
        }
        return true;
    }
```

布隆过滤器可以节省多少空间开销?

我们都知道布隆过滤器可以大大节省空间，那么它到底可以节省多少的空间呢？**这要如何去计算呢？**在我们学习了关键因子的计算后，这简直就是一件小事情

我们知道 c = m/n ，c 代表每个元素在布隆过滤器中平均所占用的空间，即多少的 bits。我们假设构建一个满足 p = 0.001, n = 10000 条件的 Filter



经过计算可以知道 m = 143776, k = 10，那么平均每个元素所占用的空间就是 c = m/n = 14.38 bits。通常主键的类型会是是 32/64 整型或是字符串

假设我们要判断的元素是 64 位的整型，一个 64 位的数值占 64 bits，所以每个元素我们就压缩了近 (64-14.38/64) = 63% 的空间

假设我们要判断的元素是 32 长度的字符串，而每个字符串就占用 32 * 8 = 256 bits，那么每个元素我们就将近压缩了 94% 的空间

这是单个元素空间压缩的计算方式，如果觉得不够直观，那我们可以按照总量来计算， 假设 n = 100000000 (1 亿)， p = 0.001，保持 c = 14.38 ，那么 m 就会是 1437758757 bits (171.39MB)。假设依然是 32 长度的字符串，那么要存储 1 亿个这样的字符，我们就需要 256 * 100000000 = 25,600,000,000 = 2.98 GB 的空间大小，而如果我们采用布隆过滤器，则只需要 171 MB

应用场景

1.推荐去重过滤

比如对一个短视频推荐系统而言，在视频 feed 流场景，需求是向用户推荐了指定内容后，一段时间内不再重复出现已推荐过的内容，那么布隆过滤器就可以大派用场。

2.缓存穿透过滤

比如在互联网大并发场景下，为防止流量大面积穿透压垮 RDS，很多系统都会结合使用缓存去抗流量。所以我们也可以考虑使用布隆过滤器，针对识别出的外部非法请求，或是空结果请求，通通记录在过滤器中，并在下次访问中，提前由过滤器先行过滤

3.黑名单系统

比如某些系统需要记录百万，千万甚至更多的黑名单字符，使用传统的哈希表实现，需要使用到较大的存储空间，那么我们也可以采用布隆过滤器去提供相同的能力，并有效降低空间存储成本

布隆过滤器的弊端

我们都知道布隆过滤器相比传统“检索元素是否存在”的容器而言，具有节省空间存储成本的优势，在大数据领域则尤其明显。那么它就只有长处，毫无缺陷吗？

并不是的，布隆过滤器还有具有两个非常重要的缺陷

1.具有一定概率的误判率，即假阳性率 fpp

不具备删除元素的能力

具有一定的误判率是什么意思呢？意思就是可能会将“不存在的元素误判为存在”，而这个概率就是 fpp，如何规避和解决？

首先必然存在误判，如无法接受误判，则布隆过滤器不适合使用。但可以根据业务需求降低概率，或是选择我们可接受的误判概率；

误判率 (假阳性率) 大小会受到哈希函数的个数，位数组的大小，以及预期元素个数等因素的影响

通常根据业务需求，选择可接受的 fpp 构建布隆过滤器，比如 fpp = 0.001，每 1000 次误判 1 次

2.不具备删除元素的能力：根据布隆过滤器的设计，一个元素会被多个哈希函数计算，并得到多个不同的位置；因为是基于哈希实现，那么必然存在哈希冲突，即多个元素的某些哈希函数的哈希值是可能一致，即多个元素会共用一些位置。如果我们要删除某个元素，那么就需要将其所占有的位置重置为 0，但

这可能会影响到其它共有该位置的元素的判断，所以综上布隆过滤器是不支持删除的。如果要删除，要怎么解决呢？

布隆过滤器无法解决，但是可以基于布隆过滤器进行扩展，衍生出可删除的过滤器，比如 CBF (counting bloom filter) 、Cuckoo Filter (布谷鸟过滤器)等，如果要删除，则布隆过滤器的基本构型是不满足需求的，可以采用其他类似的数据结构

Bloom Filter 代码

```
public abstract class BloomFilter {

    // A byte has 8 bits
    private static final int B = 8;

    // B mask
    private static final int B_MASK = B - 1;

    // 字节数组
    private byte[] bytes;

    // 关键因子
    private int n;
    private int p;
    private int m;
    private int k;

    List<Hash> hashes;

    public BloomFilter(int n, int p) {
        this.n = n;
        this.p = p;
        this.m = optimalNumOfBits(n, p);
        this.k = optimalNumOfHashFunctions(this.n, this.m);
        this.bytes = new byte[this.m];
```

```
    }

    // 插入元素&是否存在
    public synchronized void put(byte[] bs) {
        int[] indexs = hashes(bs);
        for (int index : indexs) {
            // 所在的具体字节
            byte bits = (byte) (index / B);
            // 所在的具体位
            byte offset = (byte) (1 << (B_MASK - (index % B)));
            bytes[index / B] = (byte) (bits | offset);
        }
    }

    public synchronized boolean contains(byte[] bs) {
        int[] indexs = hashes(bs);
        for (int index : indexs) {
            byte bits = (byte) (index / B);
            byte offset = (byte) (1 << (B_MASK - (index % B)));
            if ((bits & offset) == 0) {
                return false;
            }
        }
        return true;
    }

    public int[] hashes(byte[] bs) {
        int[] indexs = new int[hashes.size()];
        for (int i = 0; i < hashes.size(); i++) {
            int index = (int) ((hashes.get(i).hashToLong(bs) &
Long.MAX_VALUE) % this.m);
            indexs[i] = index;
```

```
        }
        return indexs;
    }

    private static int optimalNumOfHashFunctions(int n, int m) {
        return Math.max(1, (int) Math.round((double) m / n * Math.log(2)));
    }


    private static int optimalNumOfBits(int n, double p) {
        if (p == 0) {
            p = Double.MIN_VALUE;
        }
        return (int) (-n * Math.log(p) / (Math.log(2) * Math.log(2)));
    }

    public interface Hash {
        long hashToLong(byte[] bs);
    }

}
```