基于 Bloom Filter 的设计

课 程 报 告

姓　名　黄宇杰

学　号　U201914861

班　号　CS1902 班

日　期　2023.5.29

目　　录

一、实验目的

1. 分析 Bloom Filter 的结构与原理；
2. 基于结构和原理使用 Golang 语言代码实现；
3. 分析 Bloom Filter 的弊端。

二、实验背景

布隆过滤器（Bloom Filter）是由 Howard Bloom 于 1970 年提出的一种极其空间有效的概率型数据结构，用于测试一个元素是否属于集合。它的主要优势在于其空间效率和查询时间都非常高效。

布隆过滤器的原理：

（1） 空间和时间的效率：布隆过滤器使用了一个位数组（bit array）和几个不同的哈希函数。比如，如果我们有一个布隆过滤器用来存储字符串集合，每当一个新的字符串添加到集合中，布隆过滤器将通过哈希函数生成几个不同的索引，然后将对应索引的位设置为 1。

（2） 查询元素：当查询一个元素是否在集合中时，布隆过滤器会根据同样的哈希函数计算出位数组的索引，如果所有的位都是 1，那么元素可能在集合中；如果有任何一位是 0，那么元素肯定不在集合中。

布隆过滤器的特性：

（1） 概率性质：布隆过滤器存在一定的误判率，也就是它可能会对那些并未加入过滤器的元素认为其存在。但是，对于实际存在的元素，布隆过滤器不会出现误报，即不存在“误认为不存在”的情况。

（2） 无法删除元素：一旦一个元素被加入布隆过滤器，就无法从中删除。因为如果尝试删除一个元素（将位从 1 设置为 0），可能会影响到其他元素的存在检测。

三、代码实现

使用语法简单且性能高效的 Go 语言进行简单的代码实现。

数据结构：

```
type Filter struct {
    lock sync.RWMutex
    bits []uint64
    keys []uint64
    m    uint64
    n    uint64
}
```

简易版 hash 函数：

```
func (f *Filter) hash(v hash.Hash64) []uint64 {
    rawHash := v.Sum64()
    n := len(f.keys)
    hashes := make([]uint64, n)

    for i := 0; i < n; i++ {
        hashes[i] = rawHash ^ f.keys[i]
    }

    return hashes
}
```

添加元素：

```
func (f *Filter) Add(v hash.Hash64) {
    f.lock.Lock()
    defer f.lock.Unlock()

    for _, i := range f.hash(v) {
        // f.setBit(i)
        i %= f.m
        f.bits[i>>6] |= 1 << uint(i&0x3f)
    }

    f.n++
}
```

验证元素是否存在：

```
func (f *Filter) Contains(v hash.Hash64) bool {
    f.lock.RLock()
    defer f.lock.RUnlock()
    r := uint64(1)
    for _, i := range f.hash(v) {
        // r |= f.getBit(k)
        i %= f.m

        r &= (f.bits[i>>6] >> uint(i&0x3f)) & 1
    }

    return uint64ToBool(r)
}
```

四、Bloom Filter 的弊端

布隆过滤器的原理：

（1） 具有一定概率的误判率，即假阳性率 fpp。误判率是指可能会将 “不存在的元素误判为存在” ，而这个概率就是 fpp。

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 次查询元素：当查询一个元素是否在集合中时，布隆过滤器会根据同样的哈希函数计算出位数组的索引，如果所有的位都是 1，那么元素可能在集合中；如果有任何一位是 0，那么元素肯定不在集合中。

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。不过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。

参考文献

[1] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese,“Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines,” Proc. ACM SIGCOMM, 2006.
[2] Y. Zhu and H. Jiang, “False Rate Analysis of Bloom Filter Replicas in Distributed Systems,” Proc. Int’l Conf. Parallel Processing (ICPP ’06), pp.255-262, 2006.
[3] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, “Longest Prefix Matching Using Bloom Filters,” Proc. ACM SIGCOMM, pp. 201-212,2003.
[4] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol,” IEEE/ACM Trans. Networking, vol. 8, no. 3, pp. 281-293, June 2000.
[5] 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.
[6] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, “Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems,” Proc. 28th Int’l Conf. Distributed Computing Systems (ICDCS ’08), pp. 403-410, 2008.
[7] D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and Network Application of Dynamic Bloom Filters,” Proc. IEEE INFOCOM, 2006.