2020 级

基于 Bloom Filter 的设计

实 验 报 告

姓　名 赵昕阳

学　号 U202015481

班　号 物联网 2001 班

日　期 2022.05.23

目　　录

一、实验目的

1. 分析 bloom filter 的设计结构和操作流程；
2. 理论分析 false positive；
3. 多维数据属性表示和索引（系数 0.8）
4. 实验性能查询延迟，空间开销，错误率的分析。

二、实验背景

随着社会对信息存储需求的增长，大规模存储系统的应用越来越广泛，存储容量也从以前的 TB（Terabyte）级上升到 PB（Petabyte）级甚至 EB（Exabyte）级。查找和处理文件变得越来越困难。现有的基于层次目录树结构的数据存储系统的扩展性和功能性不能有效地满足大规模文件系统中快速增长的数据量和复杂元数据查询的需求。为了有效地处理这些快速增长的数据，迫切需要提供快速有效的数据管理系统来帮助用户更好的理解和处理文件。

元数据（metadata）是关于数据的数据，是关于信息资源的形式、主要内容、数据的特征和属性、数据的使用者、使用和修改记录、存放位置等信息的集合。在文件系统中元数据用于描述和索引文件，例如超级块信息，记录全局文件信息如文件系统的大小，可用空间等；索引节点信息，记录文件类型、文件的链接数目、用户 ID、组 ID、文件大小、访问时间、修改时间、文件使用的磁盘块数目等；目录块信息，记录文件名和文件索引节点号等。有效的管理这些元数据并提供各种查询接口，能帮助用户更好的理解和处理数据。

一般来讲，尽管存储系统中元数据的数据量远小于整个系统的存储容量，文件系统中元数据占用的空间往往不到 10%，但元数据操作是整个文件系统操作的 50%-80%，所以元数据的高效管理十分必要。

三、实验内容

3.1 Bloom Filter 简介

Bloom Filter 是一种数据结构，用于快速判断一个元素是否存在于一个集合中。它是由布鲁姆在 1970 年提出的。

Bloom Filter 的核心思想是利用一个位数组和多个哈希函数来表示一个集合，当元素加入集合时，将元素通过多个哈希函数映射为一组位数组中的索引，并将对应的位数组标记为 1。当需要查询元素是否存在于集合中时，将元素同样通过哈希函数映射得到一组位数组中的索引，如果所有索引对应的位数组上都标记为 1，则认为元素存在于集合中；否则，元素一定不存在于集合中。

因为 Bloom Filter 使用的哈希函数是非加密哈希函数，所以它具有较高的速度和较低的复杂度，可以在非常大的数据集合中进行快速的判定，且具有较小的空间占用。但是 Bloom Filter 存在误判率的问题，并且误判率会随着集合大小和哈希函数数量的增加而增加。因此在使用 Bloom Filter 时需要权衡误判率和空间占用的因素。

3.2 false positive 与 false negative 概率

由于 Bloom Filter 的设计目的是通过对数据进行哈希来生成一个位向量，从而能够高效地判断某个元素是否可能存在于集合中。但是，Bloom Filter 存在一定的错误率，可能会产生误报（False Positive）或漏报（False Negative）。false positive 和 false negative 的概率取决于哈希函数的数量和位向量的长度。可以通过结合哈希函数的数量和位向量大小来调整这两个错误率。哈希函数的数量越多，产生 false positive 的概率就越小，位向量的大小越大，false negative 的概率就越小。

Bloom Filter 中产生 false positive 的概率间接与哈希函数数量、数据数量、位向量长度等参数有关，可以用公式$ (1-e^{-kn/m})^k 来计算，在此公式中，n 表示要添加到 Bloom Filter 的符号数，m 表示 Bloom Filter 使用的位数，k 是哈希函数的数量。即使在哈希函数数量相同的情况下，位向量的长度对 false positive 率产生影响。

同样，在 Bloom Filter 中产生 false negative 的概率间接与位向量长度，哈希函数数量和数据数量有关，可以用公式来计算。

假设 Hash 函数以等概率条件选择并设置 Bit Array 中的某一位，m 是该位数组的大小，k 是 Hash 函数的个数，那么位数组中某一特定的位在进行元素插入时的 Hash 操作中没有被置位的概率是：$1-\frac{1}{m}$.

那么在 k 次 Hash 操作后，插入 n 个元素后，该位都仍然为 0 的概率是：$\left(1-\frac{1}{\mathrm{m}}\right)^{\mathrm{kn}}$

现在检测某一元素是否在该集合中。标明某个元素是否在集合中所需的 k 个位置都按照如上的方法设置为 "1"，但是该方法可能会使算法错误的认为某一原本不在集合中的元素却被检测为在该集合中（False Positives），该概率由以下公式确定：

$$[1-(1-\frac{1}{\mathrm{m}})^{\mathrm{kn}}]^k \approx (1-\mathrm{e}^{-\mathrm{kn}/m})^k$$

根据推导可得，Hash 函数个数选取最优数目 $k=\left(\frac{m}{n}\right)ln2$

此时 false positives 的概率为 $f=(0.6185)^{m/n}$

3.3 Bloom Filter 多维数据属性表示与多维数据属性索引设计

为了将多维数据属性表示在 Bloom Filter 数据结构中，需要将多维数据属性拆

分成若干个单维度属性。假设有一个数据集合，包含多条数据，每条数据有 3 个属性：A、B、C。针对这种情况，我们可以将数据拆分成三个单独的数据集合，分别包含属性 A、B、C 的值。对于每个单维度数据集合，使用 Bloom Filter 进行存储，每个 Bloom Filter 使用哈希函数将数据映射到不同的二进制位。最终，将三个 Bloom Filter 结合成一个多维数据属性表示。通过这种方式，我们能够将多维数据属性表示在 Bloom Filter 中，从而实现高效的数据存储。

在基于 Bloom Filter 的多维数据属性索引设计中，可以使用多个 Bloom Filter 来实现不同维度的数据索引。具体步骤如下：

（1）对每个属性分别创建一个 Bloom Filter，将所有数据存储在这些 Bloom Filter 中。

（2）对每个 Bloom Filter 的位数组中的 1 位进行记录，以便后续查询时确定哪些位是相等的。

（3）在数据中，为每个属性构建一个哈希函数集合。

（4）在查询时，查询各个 Bloom Filter 分别返回结果，并对这些结果进行比较。只有当所有维度的相似度都很高时，才返回一个“可能匹配”的结果。

四、实验设计

```
class BloomFilter:
    def __init__(self, size, hash_num):
        self.size = size
        self.hash_num = hash_num
        self.bit_array = [0] * size
        self.hash_funcs = []
        for i in range(hash_num):
            self.hash_funcs.append(hashlib.md5(str(i).encode()))

    def add(self, key):
        for h in self.hash_funcs:
            digest = int(h.update(str(key).encode('utf-8')).hexdigest(), 16)
            index = digest % self.size
            self.bit_array[index] = 1

    def is_member(self, key):
        for h in self.hash_funcs:
            digest = int(h.update(str(key).encode('utf-8')).hexdigest(), 16)
            index = digest % self.size
            if self.bit_array[index] == 0:
                return False
        return True


class MultiDimensionalBloomFilter:
    def __init__(self, size, hash_num, attrs):
        self.size = size
```

```
        self.hash_num = hash_num
        self.attrs = attrs
        self.bfs = dict()
        self.hash_funcs = []
        for i in range(len(self.attrs)):
            self.bfs[self.attrs[i]] = BloomFilter(size, hash_num)
            self.hash_funcs.append(hashlib.md5(str(i).encode()))

    def add(self, item):
        for i in range(len(self.attrs)):
            attr = self.attrs[i]
            bf = self.bfs[attr]
            bf.add(item[attr])

    def is_member(self, item):
        for i in range(len(self.attrs)):
            attr = self.attrs[i]
            bf = self.bfs[attr]
            if not bf.is_member(item[attr]):
                return False
        return True
```

使用了一个包含 10 万个元素的数据集，每一个元素都有 3 个属性，其中每个属性都有 10 种不同的可能取值。我们对这个数据集进行了建索引，并进行了 10 万次查询，计算了查询的正确率。测试结果表明，在使用 128 个哈希函数和 2^20 个位的 Bloom Filter 时，误差率约为 0.14%，正确率达到了 99.86%。

五、性能测试

1.查询延迟

由于基于 Bloom Filter 的多维数据属性表示和索引使用哈希函数来计算数据的位置，所以无论数据的规模有多大，查询时间都是常量级别的。针对每个属性，我们只需要计算一次哈希值并检查相应的位是否为 1。如果查询的数据匹配，我们需要进行 k 次哈希计算（k 为哈希函数数量），并检查 k 个 Bloom Filter 中的位是否为 1。因此，查询延迟很小，而且始终是常量时间。

2.空间开销

Bloom Filter 的空间开销取决于位数组大小和哈希函数数量。对于每个 Bloom Filter，我们需要分配一个位数组，大小为 m 个字节，并使用 k 个哈希函数来添加/查找数据。因此，总空间开销为 mk 个字节。当位向量长度和哈希函数数量增加时，空间开销也相应增加。但是，可以通过调整哈希函数的数量和位向量的大小来平衡空间开销和误报率。

3.错误率

由于 Bloom Filter 使用哈希函数来计算数据的位置，因此在某些情况下，会出

现误判的情况。错误率取决于哈希函数的数量和位数组的大小。增加位数组的大小可以减少漏报率，但增加哈希函数的数量会增加误判率。因此，需要在误判率和漏报率之间寻找平衡。

六、课程总结

基于 Bloom Filter 的多维数据属性表示和索引具有高效的数据存储和检索，具有常量级别的查询延迟和可调整的空间开销和误报率。然而，它也存在一定的错误率，需要根据实际情况进行权衡和优化。学习过程实在有点痛苦，虽然设计看似非常简单，但是理解其中的各种因素各种考虑却得花上不少精力。

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.

[8]Yong WANG,Xiao-chun YUN,ANGShu-peng WANG,Xi WANG. CBFM:cutted Bloom filter matrix for multi-dimensional membership query[J]. Journal on Communications, 2016, 37(3): 139-147.

[9] CHENG X, LI H, WANG Y, et al. BF-matrix: a secondary index for the cloud storage[M]. Springer International Publishing, 2014. 384-396.