14 高效的布隆过滤器 - RedisBloomDupeFilter

更新时间：2019-06-14 14:35:30

> 学习从来无捷径，循序渐进登高峰。
>
> —— 高永祚

在高速的RedisDupeFilter去重过滤器的基础上，如果采用布隆算法进行优化在面对亿级以上爬取量的目标网站就显得更加游刃有余。

布隆过滤器（Bloom Filter）是由Burton Howard Bloom于1970年提出的，它是一种space efficient的概率型数据结构，用于判断一个元素是否在集合中。在垃圾邮件过滤的黑白名单方法、爬虫（Crawler）的网址判重模块等场景中经常被用到。哈希表也能用于判断元素是否在集合中，但是布隆过滤器只需要哈希表的1/8或1/4的空间复杂度就能完成同样的任务。布隆过滤器可以插入元素，但不可以删除已有元素。元素越多，false positive rate（误报率）越大，但是false negative（漏报）是不可能的。

在爬虫中使用布隆过滤器可以实现高效去重。布隆过滤器可以用于快速检索一个元素是否在集合中。布隆过滤器实际上是一个很长的二进制向量和一系列随机映射函数（Hash函数）。而一般判断一个元素是否在集合中的做法是：用需要判断的元素和集合中的元素进行比较，大部分数据结 构如链表、树，都是这么实现的。

优点

相比于其他数据结构，布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数O(k)。另外，散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合中非常有优势。
布隆过滤器可以表示全集，其他任何数据结构都不能；k和 m相同，使用同一组散列函数的两个布隆过滤器的交并来源请求运算可以使用位操作进行。

确定性

当使用相同大小和数量的哈希函数时，某个元素通过布隆过滤器得到的是正反馈还是负反馈的结果是确定的。对于某个元素x，如果它现在可能存在，那五分钟之后、一小时之后、一天之后、甚至一周之后的状态都是可能存在的。当我得知这一特性时有一点惊讶。因为布隆过滤器是概率性的，其结果显然应该存在某种随机因素，难道不是吗？确实不是。它的概率性体现在我们无法判断究竟哪些元素的状态是可能存在的。

换句话说，过滤器一旦做出可能存在的结论后，结论就不会发生变化。

布隆过滤器的容量

布隆过滤器需要事先知道要插入元素的个数。如果并不知道或者很难估计元素的个数，情况就不太妙。也可以随机指定一个很大的容量，但这样会浪费许多存储空间，存储空间是我们试图优化的首要任务，也是选择使用布隆过滤器的原因之一。一种解决方案是创建一个能够动态适应数据量的布隆过滤器，但是在某些应用场景下这个方案无效。有一种可扩展布隆过滤器，它能够调整容量来适应不同数量的元素。它能弥补一部分短板。

空间效率

如果想在集合中存储一系列的元素，有很多种不同的做法。可以把数据存储在HashMap，随后在HashMap中检索元素是否存在，HashMap插入和查询的效率都非常高。但是，由于HashMap直接存储内容，所以空间利用率并不高。

如果希望提高空间利用率，我们可以在元素插入集合之前做一次哈希变换。还有其他方法吗？我们可以用位数组来存储元素的哈希值。我们也允许在位数组中存在哈希冲突。这正是布隆过滤器的工作原理，它们就是基于允许哈希冲突的位数组，可能会造成一些误报。在布隆过滤器的设计阶段就允许哈希冲突的存在，否则空间使用就不够紧凑了。

缺点

但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但如果元素数量太少，则使用散列表。
另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加1，这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器中。这一点单凭过滤器是无法保证的。另外计数器回绕也会造成问题。在降低误算率方面有不少方法，出现了很多布隆过滤器的变种。

误报

布隆过滤器能够“拍着胸脯”说某个元素“肯定不存在”，但是对于一些元素它们会说“可能存在”。针对不同的应用场景，这有可能会是一个巨大的缺陷，或是无关紧要的问题。如果在检索元素是否存在时不介意引入误报情况，那么就应当考虑用布隆过滤器。

另外，如果随意地减小了误报比率，哈希函数的数量就要相应地增加，插入和查询时的延时也会相应地增加。本节的另一个要点是，如果哈希函数是相互独立的，并且输入元素在空间中均匀地分布，那么理论上真实误报率就不会超过理论值。否则，由于哈希函数的相关性和更频繁的哈希冲突，布隆过滤器的真实误报比例会高于理论值。

布隆过滤器的构造和检索

在使用布隆过滤器时，我们不仅要接受少量的误报率，还要接受速度方面的额外时间开销。相比于HashMap，对元素做哈希映射和构建布隆过滤器时必然存在一些额外的时间开销。

无法返回元素本身

布隆过滤器并不会保存插入元素的内容，只能检索某个元素是否存在。因为存在哈希函数和哈希冲突，我们无法得到完整的元素列表。这是它相对于其他数据结构的最显著优势，空间的使用率也造成了这块短板。

删除某个元素

想从布隆过滤器中删除某个元素可不是一件容易的事情，我们无法撤回某次插入操作，因为不同项目的哈希结果可以被索引在同一位置。如果想撤消插入，只能记录每个索引位置被置位的次数，或是重新创建一次。两种方法都有额外的开销。基于不同的应用场景，若要删除一些元素，我们更倾向于重建布隆过滤器。

布隆过滤器的算法

> 创建一个m位BitSet，先将所有位初始化为0，然后选择k个不同的哈希函数。第i个哈希函数对字符串str哈希的结果记为h（i，str），且h（i，str）的范围是0到m-1 。

第一步：加入字符串

下面是每个字符串处理的过程，首先是将字符串str“记录”到BitSet中：

对于字符串str，分别计算`h（1，str）`、`h（2，str）`…　　…`h（k，str）`。然后将BitSet的第`h（1，str）`、`h（2，str）`……`h（k，str）`位设为1。

第二步：检查字符串是否存在

下面是检查字符串str是否被BitSet记录过的过程：

对于字符串str，分别计算`h（1，str）`、`h（2，str）`……`h（k，str）`。然后检查BitSet的第`h（1，str）`、`h（2，str）`…　　…`h（k，str）`位是否为1，若其中任何一位不为1，则可以判定str一定没有被记录过。若全部位都是1，则“认为”字符串str存在。

若一个字符串对应的Bit不全为1，则可以肯定该字符串一定没有被BloomFilter记录过（这是显然的，因为字符串被记录过，其对应的二进制位肯定全部被设为1了）。

若一个字符串对应的Bit全为1（实际上是不为100%的），则肯定该字符串被BloomFilter记录过（因为有可能该字符串的所有位都刚好被其他字符串所对应）。这种将该字符串划分错的情况，称为“假阳性”（false positive）。

布隆过滤器的参数选择

哈希函数的选择

哈希函数的选择对性能的影响应该是很大的，一个好的哈希函数要能近似等概率地将字符串映射到各个Bit。选择k个不同的哈希函数比较麻烦，一种简单的方法是选择一个哈希函数，然后送入k个不同的参数。

** m、n、k如何取值**

- 可能把不属于这个集合的元素误认为属于这个集合（假阳性，False Positive）。
- 不会把属于这个集合的元素误认为不属于这个集合（假阴性，False Negative）。

哈希函数的个数k取10，位数组大小m设为字符串个数n的20倍时，假阳性发生的概率是0.0000889，即10万次的判断中，会存在9次误判，对于一天1亿次的查询，误判的次数为9000次。

哈希函数的个数k、位数组大小m、加入的字符串数量n的关系可以参考下表。

m/n	k	k=17	k=18	k=19	k=20	k=21	k=22	k=23	k=24
22	15.2	2.67e-05							
23	15.9	1.61e-05							
24	16.6	9.84e-06	1e-05						
25	17.3	6.08e-06	6.11e-06	6.27e-06					
26	18	3.81e-06	3.76e-06	3.8e-06	3.92e-06				
27	18.7	2.41e-06	2.34e-06	2.33e-06	2.37e-06				
28	19.4	1.54e-06	1.47e-06	1.44e-06	1.44e-06	1.48e-06			
29	20.1	9.96e-07	9.35e-07	9.01e-07	8.89e-07	8.96e-07	9.21e-07		
30	20.8	6.5e-07	6e-07	5.69e-07	5.54e-07	5.5e-07	5.58e-07		
31	21.5	4.29e-07	3.89e-07	3.63e-07	3.48e-07	3.41e-07	3.41e-07	3.48e-07	
32	22.2	2.85e-07	2.55e-07	2.34e-07	2.21e-07	2.13e-07	2.1e-07	2.12e-07	2.17e-07

> 上表引用自http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html。

布隆过滤器的Python实现

了解实现原理后，我们可以用“已有的轮子”来将布隆过滤器接入至Scrapy中。pybloomfiltermmap（https://github.com/axiak/pybloomfiltermmap ）是Python世界中比较有名的布隆过滤器，按以下方式在命令行安装：

```
$ pip install pybloomfiltermmap
```

pybloomfiltermmap的使用非常简单，它提供了一个`BloomFilter`类，在实例化时只需要输入布隆过滤器的大小与误判率与缓存文件名即可。先来看一个基本的使用示例，具体代码如下：

```
>>> fruit = pybloomfilter.BloomFilter(100000, 0.1, '/tmp/words.bloom')
>>> fruit.update(('apple', 'pear', 'orange', 'apple'))
>>> len(fruit)
3
>>> 'mike' in fruit
False
>>> 'apple' in fruit
True
```

`update`方法是将需要判断的字符串加入布隆过滤器。

接下来就可以将其集成至Scrapy中，编写一个`BloomDupeFilter`，使Scrapy能支持布隆去重，具体代码如下所示。

```
from pybloomfilter import BloomFilter
from scrapy.utils.request import request_fingerprint

class BloomDupeFilter(object):

    def __init__(self):
        self.bloomfilter = BloomFilter(100000,0.1,'request_seen.bloom')

    def request_seen(self, request):
        fp = request_fingerprint(request)
        if fp in self.bloomfilter:
            return True
        else:
            self.bloomfilter.add(fp)
            return False

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value('dupefilter/filtered', spider=spider)
```

BloomDupeFilter与RFPDupeFilter相比运行速度是有所提升的，但是由于其持久化方式仍然采用文件形式，加入文件时会将所有数据一次性地加载到系统的内存中，一旦文件体积变得越来越大，仍然无法逃脱晒爆内存的窘境。

编写布隆过滤器 RedisBloomDupeFilter

要摆脱超大的持久化文件撑爆内存的问题，最佳的解决办法还是将布隆过滤器持久化数据保存到数据库中。如果完全理解了布隆过滤器的算法与实现思路，则一定会发现Redis可以作为布隆过滤器的数据载体，Redis和布隆过滤器简直就是天生一对！Redis原生就有BitSet类型，非常容易操控。

首先需要实现一个HashMap：

```
class HashMap(object):

    def __init__(self, m, seed):
        self.m = m
        self.seed = seed

    def hash(self, value):
        """
        哈希算法
        :param value: Value
        :return: Hash Value
        """
        ret = 0
        for i in range(len(value)):
            ret += self.seed * ret + ord(value[i])
        return (self.m - 1) & ret
```

然后是基于Redis实现的布隆过滤器：

```
BLOOMFILTER_HASH_NUMBER = 6
BLOOMFILTER_BIT = 30

class BloomFilter(object):

    def __init__(self, server, key='bloomfilters', bit=BLOOMFILTER_BIT, hash_number=BLOOMFILTER_HASH_NUMBER):
        """
        构造 BloomFilter
        :param server: Redis 服务器地址
        :param key: 布隆过滤器在Redis使用的键名
        :param bit: m = 2 ^ bit - 指定内存空间
        :param hash_number: 进行Hash的数量
        """
        # 默认 1 << 30 = 10,7374,1824 = 2^30 = 128MB, 最大过滤的请求指纹数为：2^30/hash_number = 1,7895,6970
        self.m = 1 << bit
        self.seeds = range(hash_number)
        self.server = server
        self.key = key
        self.maps = [HashMap(self.m, seed) for seed in self.seeds]

    def exists(self, value):
        """
        判断数据是否存
        :param value:
        :return:
        """
        if not value:
            return False
        exist = True
        for map in self.maps:
            offset = map.hash(value)
            exist = exist & self.server.getbit(self.key, offset)
        return exist

    def insert(self, value):
        """
        将数据插入到布隆过滤器的键值存储空间内
        :param value:
        :return:
        """
        for f in self.maps:
            offset = f.hash(value)
            self.server.setbit(self.key, offset, 1)
```

最后接入Scrapy的过滤器框架：

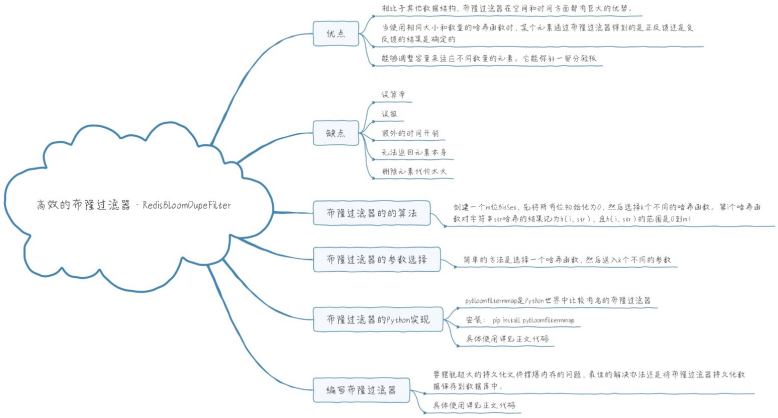
```
from redis import Redis
from redis_bloomfilter import BloomFilter
from scrapy.utils.request import request_fingerprint
import logging

class RedisBloomDupeFilter(BaseDupeFilter):

    def __init__(self, host='localhost', port=6379, db=0, blockNum=1, key='bloomfilter'):
        self.redis = Redis(host=host, port=port, db=db)

        self.bit_size = 1 << 31  # Redis的String类型最大容量为512M，现使用256M
        self.seeds = [5, 7, 11, 13, 31, 37, 61]
        self.key = key
        self.blockNum = blockNum
        self.hashfunc = []
        for seed in self.seeds:
            self.hashfunc.append(SimpleHash(self.bit_size, seed))

        self.logger = logging.getLogger(__name__)

    @classmethod
    def from_settings(cls, settings):
        _port = settings.getint('REDIS_PORT', 6379)
        _host = settings.get('REDIS_HOST', '127.0.0.1')
        _db = settings.get('REDIS_DUP_DB', 0)
        key = settings.get('BLOOMFILTER_REDIS_KEY', 'bloomfilter')
        block_number = settings.getint(
            'BLOOMFILTER_BLOCK_NUMBER', 1)

        return cls(_host, _port, _db, blockNum=block_number, key=key)

    def request_seen(self, request):
        fp = request_fingerprint(request)
        if self.exists(fp):
            return True

        self.insert(fp)
        return False

    def exists(self, str_input):
        if not str_input:
            return False
        m5 = md5()
        m5.update(str(str_input).encode('utf-8'))
        _input = m5.hexdigest()
        ret = True
        name = self.key + str(int(_input[0:2], 16) % self.blockNum)
        for f in self.hashfunc:
            loc = f.hash(_input)
            ret = ret & self.redis.getbit(name, loc)
        return ret

    def insert(self, str_input):
        m5 = md5()
        m5.update(str(str_input).encode('utf-8'))
        _input = m5.hexdigest()
        name = self.key + str(int(_input[0:2], 16) % self.blockNum)
        for f in self.hashfunc:
            loc = f.hash(_input)
            self.redis.setbit(name, loc, 1)

    def log(self, request, spider):
        msg = ("已过滤的重复请求: %(request)s")
        self.logger.debug(msg, {'request': request}, extra={'spider': spider})
        spider.crawler.stats.inc_value(
            'redisbloomfilter/filtered', spider=spider)
```

布隆过滤器配置：

```
REDIS_DUP_DB = 0
BLOOMFILTER_HASH_NUMBER = 6
BLOOMFILTER_BIT = 30
```

小结

基于Redis的Bloomfilter去重，既用上了Bloomfilter的海量去重能力，又用上了Redis的可持久化能力，基于Redis也方便分布式机器的去重。在使用的过程中，要估算好待去重的数据量，根据上面的表，适当地调整seed的数量和blockNum数量（seed越少去重速度越快，但漏失率越大）。



13 去重处理— 高性能爬虫调优技术

15 为网易爬虫配置存储大规模数据存储