布隆过滤器

Bloom Filter

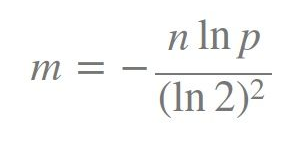
www.bzMe.work

**一、布隆过滤器简介**

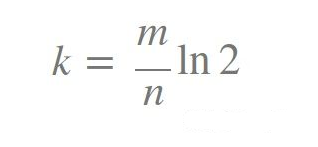
布隆过滤器（Bloom Filter）是1970年由布隆提出的，是为了解决如何在海量数据中判断一个数据是否存在而设计的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都比一般的算法要好的多，缺点是有一定的误识别率和删除困难。

**二、布隆过滤器公式**

理论上说可以用字典来缓存海量数据，并根据Key快速判断某个数据是否存在，但这是面临一个问题，它需要有足够的内存空间来缓存海量数据，例如：假设有100亿整型数据，字典的Key和Value各占用4字节，100亿数据占用的空间是：1000000000\*4\*2=800亿字节（约74G内存），如果是字符串呢？显然要成百上千G的内存才能缓存海量数据，因此用字典在内存中缓存和查找海量数据是不可取的。那么，如果将前面的100亿映射到内存中每个bit位上，这将占用多少内存空间呢？1000000000/8=12亿字节（约1G内存），从74G骤降到1G内存，这显然能解决海量数据快速查询问题了。内存问题解决了，但面临另一个问题，如何将数据正确地映射到每个bit位上而不占用过多的空间呢？布隆过滤器采用了多次Hash的方法，即将数据多次打散映射到bit位上来解决这个问题。并给出了2个公式：

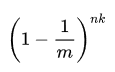


其中m为布隆过滤器长度；n 为预期想插入的元素个数；p 为误报率，介于0到1之间。通过这个公式就能计算出用于容纳元素的最佳空间（以bit为单位的哈希表）。

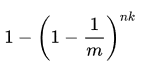


其中k为哈希函数个数，m 为布隆过滤器长度，n 为插入的元素个数。这个公式告诉我们你应该将数据进行k次哈希，才能将数据正确地映射到相应的bit位上，并降低误判率。并给出了误判的概率公式，如下：

插入n个元素后某个bit位依旧为0的概率为：



插入n个元素后某个bit位依旧为1的概率为：



**三、哈希碰撞**

理论上，如果我们选择了一个碰撞率非常低的哈希函数（例如Murmur3哈希函数），有一张非常大的哈希表，使得它总是能根据哈希函数计算出的哈希值找到对应位置，那么就不需要多次哈希，如下图所示：



例如一个ulong型的数据类型，它的取值范围是：0 —— 18,446,744,073,709,551,615，

为其分配18446744073709551615/8/1073741824=2147483647G（约21亿G）空间的位图哈希表(bitmap hashtable)，这样对于一个返回ulong型的哈希函数，其计算出的哈希值就总能从哈希表找到相应的位置。显然这是不现实的，内存空间始终是有限的，因此在有限空间的哈希表中，任何优秀的哈希函数(如Murmur3)生成的哈希值，都必须将其进行缩小，才能命中哈希表中的某个位置，如下图所示：



这就导致一个问题，一旦一个哈希值发生了缩容，必定会发生碰撞，例如100%25=0，200%25=0，300%25=0等等，都命中在哈希表的第0个位置。如下图所示：



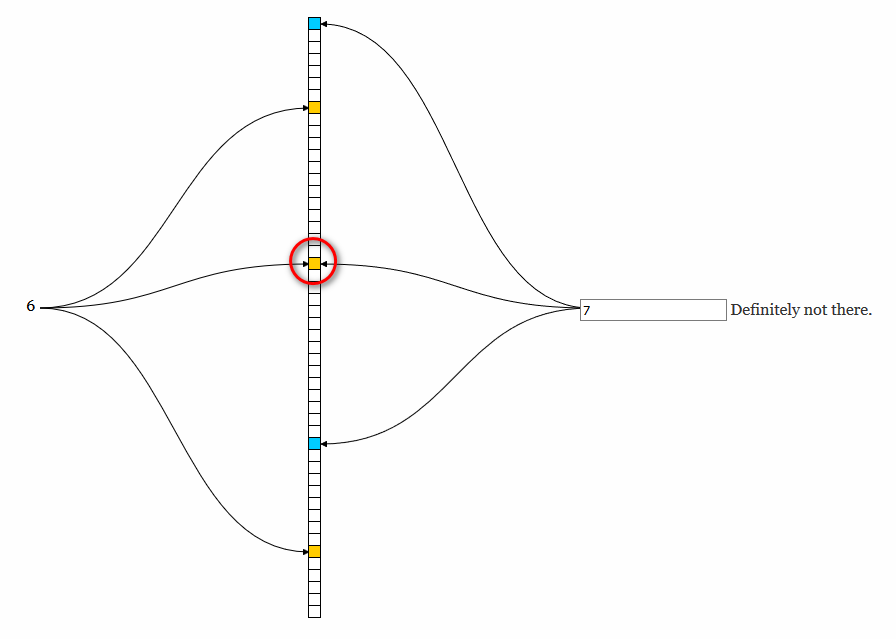
**四、布隆过滤器如何将数据映射到哈希表**

布隆过滤器中，由于哈希表由一个个bit位构成，只能存储1或0，因此为了将数据映射到bit位上，在取得哈希值以后，需要其进行多次随机打散，使之均匀地分布在位图哈希表中，如下图所示：



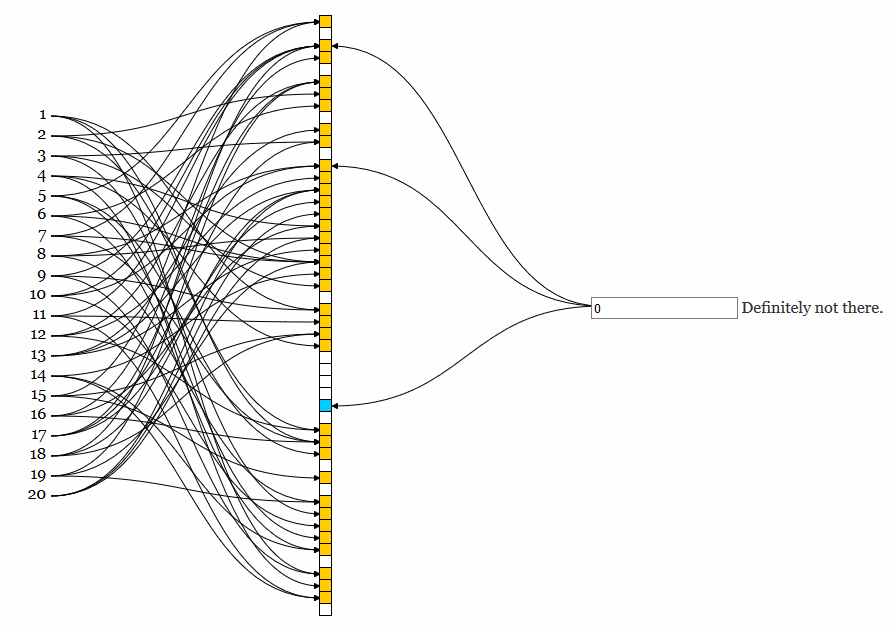
**五、布隆过滤器元素是否存在的判断逻辑**

前面已经说过为了命中哈希表中的某个位置，必须对哈希值进行缩小（c % n），因此对一个哈希值进行多次打散时就有可能发生碰撞，下面是[www.jasondavies.com/bloomfilter/](https://www.jasondavies.com/bloomfilter/) 的在线示例，我们可以看到数值7在进行哈希映射的时候发生了碰撞：



Definitely not there：绝对不存在

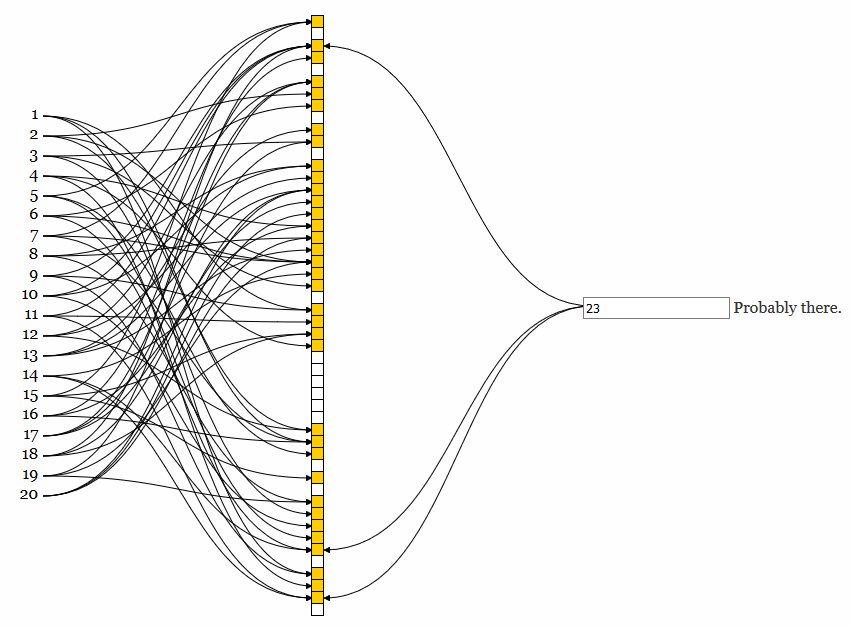
布隆过滤器在判定某个元素是否存在时的逻辑是，在进行多次哈希时，如果存在任一哈希值在哈希表中未被命中（对应bit位上的值为0即称为未命中），那么就判定这个元素一定不存在。如下图所示：



Definitely not there：绝对不存在

虽然元素0在两个位置发生了命中，但有一个(青色方框)没有命中，因此判定元素0一定不存在。

上面已经说过由于哈希值缩容的原因，会产生碰撞，这样会导致实际不存在的某个值，在哈希表中会被命中的情况产生，从而导致误判。如下图所示：



Probably there：可能存在

上图的布隆过滤器中有20个元素，此时我们看到元素23实际是不存在的，但是三次哈希以后都发生了命中，此时布隆过滤器发生了误判，认为元素23是可能存在的。从上图中我们也看到要降低碰撞率，应该保证哈希表足够大。

因此可以得出布隆过滤器的判断逻辑：

1. 如果某个元素不存在，那么这个元素一定不存在。

2. 如果某个元素存在，那么这个元素有可能存在。

**六、布隆过滤器的应用场景**

**1、缓存穿透**

缓存穿透是指缓存和数据库中都没有数据，而客户端不断地发出大量不存在数据请求，导致数据库压力过大而崩溃。此时可以采用布隆过滤器来解决。例如：假设有一个100万条数据的用户名单，将其放入在布隆过滤器中，现在客户端大量请求一个无效的用户ID，当请求到达布隆过滤器，其判断该ID不存在，则直接丢弃请求并返回。如下图所示：



这样无论多大的恶意并发请求，都不能对数据库构成威胁。

但是由于布隆过滤器存在小概率的误判，因此当请求到达布隆过滤器后，判断该ID有可能存在，如果可能存在则进行后续的处理，从缓存中读取ID信息，如果缓存中没有，则从数据库中查找。如下图所示：



在上图中，对于无效ID，只有当布隆过滤器产生误判以后，才会从缓存或数据库中查询数据，这大大地降低数据库的负荷压力。即使客户端采用同一ID让布隆过滤器每次产生误判，但是由于首次查询数据库以后缓存了一条该ID的无效数据，以后也只会从缓存中查询数据，不会给数据库造成压力。

**2、缓存击穿**

缓存击穿是指缓存故障或者由于缓存时间到期，导致缓存中没有而数据库中有数据的时候，

客户端不断地发出查询同一数据的大量请求，导致缓存读不到数据，而同时去数据库取数据，引起数据库压力瞬间过大而崩溃。为了解决这个问题，在增加布隆过滤器的基础上，对于使用频率较高的热点数据设置永不过期以外，在访问数据库提取数据的时候应该进行加锁处理，这样当其它线程试图获取相同数据的时候必须等待当前线程处理完毕，当前线程在获取到数据缓存起来以后，后续的查询就从缓存中读取了。

**3、缓存雪崩**

缓存雪崩是指缓存故障或者由于缓存时间到期，导致缓存中没有而数据库中有数据的时候，

客户端不断地发出查询不同数据的大量请求，导致缓存读不到数据，而同时去数据库取数据，引起数据库压力瞬间过大而崩溃。为了解决这个问题，在增加布隆过滤器的基础上，对于使用频率较高的热点数据设置永不过期以外，应该将其均匀地分布在不同的缓存数据库中（例如Redis集群），并修改过期时间具有一定随机性，避免同一时间大量数据过期，在访问数据库提取数据的时候应该采用任务（.NET）异步读取数据并缓存。

总结，为了防止缓存穿透、击穿、雪崩，应该：

1、增加布隆过滤器。

2、热点数据应永不过期。

3、过期时间应该具有随机性。

4、查询同一数据应该进行加锁。

5、应该提倡采用任务异步编程模式。（任务会把查询放入队列自动进行线程调度进行查询）

最后，布隆过滤器还可以用于邮箱的垃圾邮件过滤、黑名单管理、爬虫URL去重等等。