使用 Bloom Filters

原作者：Maciej Ceglowski
April 08, 2004

译者：兰花仙子
March 20, 2005

任何 perl 使用者都熟悉 hash 查询，一个存在测试的语句可以这样写：

```
foreach my $e ( @things ) { $lookup{$e}++ }

sub check {
    my ( $key ) = @_;
    print "Found $key!" if exists( $lookup{ $key } );
}
```

虽然 hash 查询很有用，但对非常大的列表，或 keys 自身非常大时，这种查询可能变得不实用。当查询 hash 增长得太大，通常的做法是将它移到数据库或文件中，只在本地缓存里保存最常用的关键字，这样能改善性能。

许多人不知道有一种优雅的算法，用以代替 hash 查询。它是一种古老的算法，叫做 Bloom filter。 Bloom filter 允许你在有限的内存里（你想在这块内存里存放关键字的完整列表），执行成员测试，这样就能避开使用磁盘或数据库进行查询的性能瓶颈。也许你会认为，空间的节省是有代价的：存在着可大可小的假命中率风险，并且一旦你增加 key 到 filter 后，就不能删除它。然而在许多情形下，这些局限是可接受的，Bloom filter 能编制有用工具。（仙子注：例如代理服务器软件 Squid 就使用了 Bloom filter 算法。）

例如，假如你运行了一个高流量的在线音乐存储站点，并且如果你已知歌曲存在，

就可以通过仅获取歌曲信息的方法，来最大程度的减少数据库压力。你可以在启动时构建一个 Bloom filter，在试图执行昂贵的数据库查询前，可以用它执行快速的成员存在测试。

```
use Bloom::Filter;

my $filter = Bloom::Filter->new( error_rate => 0.01, capacity => $SONG_COUNT );
open my $fh, "enormous_list_of_titles.txt" or die "Failed to open: $!";

while (<$fh>) {
    chomp;
    $filter->add( $_ );
}

sub lookup_song {
    my ( $title ) = @_;
    return unless $filter->check( $title );
    return expensive_db_query( $title ) or undef;
}
```

在该示例里，该测试给出假命中的几率是 1%，在假命中率情况下程序会执行昂贵的数据库索取操作，并最终返回空结果。尽管如此，你已避开了 99%的昂贵查询时间，仅使用了用于 hash 查询的一小片内存。更进一步，1%假命中率的 filter，每个 key 的存储空间在 2 字节以下。这比你执行完整的 hash 查询所需的内存少得多。

Bloom filters 在 Burton Bloom 之后命名，Burton Bloom 1970 年首先在文档里描述了它们，文档名 Space/time trade-offs in hash coding with allowable errors.在那些内存稀少的日子里，Bloom filters 因其简洁而倍受重视。事实上，最早的应用之一是拼写检查程序。然而，由于有少数非常明显的特性，该算法特别适合社会软件应用。

因为 Bloom filters 使用单向 hash 来存储数据，因此不可能在不做穷举搜索的情况下，重建 filter 里的 keys 列表。甚至这点看起来并非象很有用，既然来自穷举搜索的假命中会覆盖掉真正的 keys 列表。所以 Bloom filters 能在不向全世界广播完整列表的情况下，共享关于已有资料的信息。因为这个理由，它们在 peer-to-peer 应用中特别有用，在这个应用中大小和隐私是重要的约束。

1 Bloom filters 如何工作

Bloom filter 由 2 部分组成：1 套 k hash 函数，1 个给定长度的位向量。选择位向量的长度，和 hash 函数的数量，依赖于我们想增加多少 keys 到设置中，以及我们能容忍的多高的假命中率。

Bloom filter 中所有的 hash 函数被配置过，其范围匹配位向量的长度。例如，假如向量是 200 位长，hash 函数返回的值就在 1 到 200 之间。在 filter 里使用高质量的 hash 函数相当重要，它保证输出等分在所有可能值上——hash 函数里的“热点”会增加假命中率。（仙子注：所谓“热点”是指结果过分频繁的分布在某些值上。）

要将某个 key 输入 bloom filer 中，我们在每个 k hash 函数里遍历它，并将结果作为在位向量里的 offsets，并打开我们在该 offsets 上找到的任何位。假如该位已经设置，我们继续保留其打开。还没有在 Bloom filter 里关闭位的机制。

在本示例里，让我们看看某个 Bloom filter，它有 3 个 hash 函数，并且位向量的长度是 14。我们用空格和星号来表示位向量，以便于观察。你也许想到，空的 Bloom filter 以所有的位关闭为开始，如图 1 所示。

图 1：空的 Bloom filter

现在我们将字符 apples 增加到 filter 中去。为了做到这点，我们以 apples 为参数来运行每个 hash 函数，并采集输出：

hash1("apples") = 3
hash2("apples") = 12
hash3("apples") = 11

然后我们打开在向量里相应位置的位——在这里就是位 3，11，和 12，如图 2 所示。

图 2：激活了 3 位的 Bloom filter

为了增加另 1 个 key，例如 plums，我们重复 hash 运算过程：

hash1("plums") = 11
hash2("plums") = 1
hash3("plums") = 8

再次打开向量里相应的位，如图 3 里的高亮度显示。

图 3：增加了第 2 个 key 的 Bloom filter

注意位置 11 的位已被打开——在前面的步骤里，当我们增加 apples 时已设置了它。位 11 现在有双重义务，存储 apples 和 plums 两者的信息。当增加更多的 keys 时，它也会存储其他 keys 的信息。这种交迭让 Bloom filters 如此紧凑——任何位同时编码多个 keys。这种交迭也意味着你永不能从 filter 里取出 key，因为你不能保证你所关闭的位没有携载其他 keys 的信息。假如我们试图执行反运算过程来从 filter 里删除 apples，就会不经意的关闭编码 plums 的 1 个位。从 Bloom filter 里剥离 key 的唯一方法是重建 filter，剔除无用 key。

检查是否某个 key 已经存在于 filter 的过程，非常类似于增加新 key。我们在所有的 hash 函数里遍历 key，然后检查是否在那些 offsets 上的位都是打开的。假如任何一位关闭，我们知道该 key 肯定不存在于 filter 中。假如所有位都打开，我们知道该 key 可能存在。

我说“可能”是因为存在一种情况，该 key 是个假命中。例如，假如我们用字符 mango 来测试 filter，看看会发生什么情况。我们运行 mango 遍历 hash 函数：

hash1("mango") = 8
hash2("mango") = 3
hash3("mango") = 12

然后检查在那些 offsets 上的位，如图 4 所示。

图 4：Bloom filter 的假命中

所有在位置 3，8，和 12 的位都是打开的，故 filter 会报告 mango 是有效 key。

当然，mango 并非有效 key——我们构建的 filter 仅包含 apples 和 plums。事实是 mango 的 offsets 非常巧合的指向了已激活的位。这就找到了 1 个假命中——某

个 key 看起来位于 filter 中，但实际不是。

正如你想的一样，假命中率依赖于位向量的长度和存储在 filter 里的 keys 的数量。位向量越宽阔，我们检查的所有 k 位被打开的可能性越小，除非该 key 确实存在于 filter 中。在 hash 函数的数量和假命中率之间的关系更敏感。假如使用的 hash 函数太少，在 keys 之间的差别就很少；但假如使用 hash 函数太多，filter 会过于密集，增加了冲突的可能性。可以使用如下公式来计算任何 filter 的假命中率：

c = (1 - e(-kn/m))k

这里 c 是假命中率，k 是 hash 函数的数量，n 是 filter 里 keys 的数量，m 是 filter 的位长。

当使用 Bloom filters 时，我们先要有个意识，期待假命中率多大；也应该有个粗糙的想法，关于多少 keys 要增加到 filter 里。我们需要一些方法来验证需要多大的位向量，以保证假命中率不会超出我们的限制。下列方程式会从错误率和 keys 数量求出向量长度：

m = -kn / (ln(1 - c ^ 1/k))

请注意另 1 个自由变量：k，hash 函数的数量。可以用微积分来得出 k 的最小值，但有个偷懒的方法来做它：

```
sub calculate_shortest_filter_length {
    my ( $num_keys, $error_rate ) = @_;
    my $lowest_m;
    my $best_k = 1;

    foreach my $k ( 1..100 ) {
        my $m = (-1 * $k * $num_keys) /
            ( log( 1 - ($error_rate ** (1/$k))));

        if ( !defined $lowest_m or ($m < $lowest_m) ) {
            $lowest_m = $m;
            $best_k   = $k;
        }
    }
    return ( $lowest_m, $best_k );
}
```

为了给你直观的感觉，关于错误率和 keys 数量如何影响 Bloom filters 的存储 size，表 1 列出了一些在不同的容量/错误率组合下的向量 size。

ErrorRate	Keys	RequiredSize	Bytes/Key
1%	1K	1.87 K	1.9
0.1%	1K	2.80 K	2.9
0.01%	1K	3.74 K	3.7
0.01%	10K	37.4 K	3.7
0.01%	100K	374 K	3.7
0.01%	1M	3.74 M	3.7
0.001%	1M	4.68 M	4.7
0.0001%	1M	5.61 M	5.7

2 在 Perl 里构建 Bloom Filter

为了构建 1 个工作 Bloom filter，我们需要 1 套良好的 hash 函数。这些容易解决——在 CPAN 上有几个优秀的 hash 算法可用。对我们的目的来说，较好的选择是 Digest::SHA1，它是强度加密的 hash，用 C 实现速度很快。通过对不同值的输出列表进行排序，我们能使用该模块来创建任意数量的 hash 函数。如下是构建唯一 hash 函数列表的子函数：

```
use Digest::SHA1 qw/sha1/;

sub make_hashing_functions {
      my ( $count ) = @_;
      my @functions;

      for my $salt (1..$count ) {
            push @functions, sub { sha1( $salt, $_[0] ) };
      }

      return @functions;
}
```

为了能够使用这些 hash 函数，我们必须找到 1 个方法来控制其范围。Digest::SHA1 返回令人为难的过长 160 位 hash 输出，这仅在向量长度为 2 的 160 次方时有用，而这种情况实在罕见。我们结合使用位 chopping 和 division 来将输出削减到可用大小。

如下子函数取某个 key，运行它遍历 hash 函数列表，并返回 1 个长度（$FILTER_LENGTH）的位掩码：

```
sub make_bitmask {
    my ( $key ) = @_;
    my $mask       = pack( "b*", '0' x $FILTER_LENGTH);

    foreach my $hash_function ( @functions ){

        my $hash             = $hash_function->($key);
        my $chopped       = unpack("N", $hash );
        my $bit_offset = $result % $FILTER_LENGTH;

        vec( $mask, $bit_offset, 1 ) = 1;
    }
    return $mask;
}
```

让我们逐行分析上述代码：

```
my $mask = pack( "b*", '0' x $FILTER_LENGTH);
```

我们以使用 perl 的 pack 操作来创建零位向量开始，它是$FILTER_LENGTH 长。pack 取 2 个参数，1 个模型和 1 个值。b 模型告诉 pack 将值解释为 bits，*指“重复任意多需要的次数”，跟正则表达式类似。perl 实际上会补充位向量的长度为 8 的倍数，但我们将忽视这些多余位。

有 1 个空的位向量在手中，我们准备开始运行 key 遍历 hash 函数：

```
my $hash = $hash_function->($key);
my $chopped = unpack("N", $hash );
```

我们保存首个 32 位输出，并丢弃剩下的。这点可让我们不必要求 BigInt 支持。第 2 行做实际的位 chopping。模型里的 N 告诉 unpack 以网络字节顺序来解包 32 位整数。因为未在模型里提供任何量词，unpack 仅解包 1 个整数，然后终止。

假如你对位 chopping 过度狂热，你可以将 hash 分割成 5 个 32 位的片断，并对它们一起执行 OR 运算，将所有信息保存在原始 hash 里：

```
my $chopped = pack( "N", 0 );
```

```
my @pieces    =    map { pack( "N", $_ ) } unpack("N*", $hash );
$chopped      = $_ ^ $chopped foreach @pieces;
```

但这样作可能杀伤力过度。

现在我们有了来自 hash 函数的 32 位整数输出的列表，下一步必须做的是，裁减它们的大小，以使其位于（1..$FILTER_LENGTH）范围内。

```
my $bit_offset = $chopped % $FILTER_LENGTH;
```

现在我们已转换 key 为位 offsets 列表，这正是我们所求的。

剩下唯一要做的事情是，使用 vec 来设置位，vec 取 3 个参数：向量自身，开始位置，要设置的位数量。我们能象赋值给变量一样来分配值给 vec：

```
vec( $mask, $bit_offset, 1 ) = 1;
```

在设置了所有位后，我们以 1 个位掩码来结束，位掩码和 Bloom filter 长度一样。我们可以使用这个掩码来增加 key 到 filter 中：

```
sub add {
    my ( $key, $filter ) = @_;

    my $mask = make_bitmask( $key );
    $filter    = $filter | $mask;
}
```

或者我们使用它来检查是否 key 已存在：

```
sub check {
    my ( $key, $filter ) = @_;
    my $mask    = make_bitmask( $key );
    my $found = ( ( $filter & $mask ) eq $mask );
    return $found;
}
```

注意这些是位逻辑运算符 OR(|)和 AND(&)，而并非通用的逻辑 OR(||)和 AND(&&)运算符。将这两者混在一起，会导致数小时的有趣调试。第 1 个示例将掩码和位向量进行 OR 运算，打开任何未设置的位。第 2 个示例将掩码和 filter 里相应的位置进行比较——假如掩码里所有的打开位也在 filter 里打开，我们知道已找到一个匹配。

一旦你克服了使用vec,pack和位逻辑运算符的难度，Bloom filters实际非常简单。http://www.perl.com/2004/04/08/examples/Filter.pm 这里给出了Bloom::Filter模块的完整信息。

3 分布式社会网络中的 Bloom Filters

当前的社会网络机制的弊端之一是，它们要求参与者泄露其联系列表给中央服务器，或公布它到公共 Internet，这 2 种情况下都牺牲了大量的用户隐私。通过交换 Bloom filters 而不是暴露联系列表，用户能参与社会网络实践，而不用通知全世界他们的朋友是谁。编码了某人联系信息的 Bloom filter 能用来检查它是否包含了给定的用户名或 email 地址，但不能强迫要求它展示用于构建它的完整 keys 列表。甚至有可能将假命中率（虽然它听起来不像好特性），转换为有用工具。

假如我非常关注这些人，他们通过对 Bloom filter 运行字典攻击，来试图对社会网络进行反工程。我可以构建 filter，它具备较高的假命中率（例如 50%），然后发送 filter 的多个拷贝给朋友，并变换用于构建每个 filter 的 hash 函数。我的朋友收集到的 filters 越多，他们见到的假命中率越低。例如，在 5 个 filters 情况下，假命中率是 0.5 的 5 次方，或 3%——通过发送更多 filters，还能进一步减少假命中率。

假如这些 filters 中的任何一个被中途截取，它会展示全部 50%的假命中率。所以我能隔离隐私风险，并且一定程度上能控制其他人能多清楚的了解我的网络。我的朋友能较高程度的确认是否某个人位于联系列表里，但那些仅截取了 1 个或 2 个 filters 的人，几乎不会获取到什么。如下是个 perl 函数，它对 1 组嘈杂的 filters 检查某个 key：

```
use Bloom::Filter;

sub check_noisy_filters {
    my ( $key, @filters ) = @_;
    foreach my $filter ( @filters ) {
        return 0 unless $filter->check( $key );
    }
    return 1;
}
```

假如你和你的朋友同意使用相同的 filter 长度和 hash 函数设置，你也能使用位掩

码对比来估计在你们的社会网络之间的交迭程度。在 2 个 Bloom filters 里的共享位数量会给出 1 个可用的距离度量。

```
sub shared_on_bits {
    my ( $filter_1, $filter_2 ) = @_;
    return unpack( "%32b*",    $filter_1 & $filter_2 )
}
```

另外，你能使用 OR 运算，结合 2 个有相同长度和 hash 函数的 Bloom filters 来创建 1 个复合 filter。例如，假如你参与某个小型邮件列表，并希望基于组里每个人的地址本来创建白名单，你可以为每个参与者独立的创建 1 个 Bloom filter，然后将 filters 一起进行 OR 运算，将结果输入 Voltron-like 主列表。组里成员不会了解到其他成员的联系信息，并且 filter 仍能展示正确的行为。

肯定还有其他针对社会网络和分布式应用的 Bloom filter 妙用。如下参考列出一些有用资源。

4 参考

- **Bloom filters -- the math**. A good place to start for an overview of the math behind Bloom filters.
- **Some Motley Bloom Tricks**. Handy filter tricks and theory page.
- **Bloom filter Survey**. A handy survey article on Bloom filter network applications.
- **LOAF**. Our own system for incorporating social networks onto email using Bloom filters.
- **Compressed Bloom filters**. If you are passing filters around a network, you will want to optimize them for minimum size; this paper gives a good overview of compressed Bloom filters.
- **`Bloom16`.** A CPAN module implementing a counting Bloom filter.
- **`Text::Bloom`**. CPAN module for using Bloom filters with text collections.
- **Privacy-Enhanced Searches Using Encryted Bloom filters**. This paper discusses how to use encryption and Bloom filters to set up a query system that prevents the search engine from knowing the query you are running.
- **Bloom filters as Summaries**. Some performance data on actually using Bloom filters as cache summaries.
- **Using Bloom filters for Authenticated Yes/No Answers in the DNS**. Internet draft for using Bloom filters to implement Secure DNS