1. Bloom-Filter 算法简介

Bloom-Filter，即布隆过滤器，1970 年由 Bloom 中提出。它可以用于检索一个元素是否在一个集合中。

Bloom Filter（BF）是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。它是一个判断元素是否存在集合的快速的概率算法。Bloom Filter 有可能会出现错误判断，但不会漏掉判断。也就是 Bloom Filter 判断元素不再集合，那肯定不在。如果判断元素存在集合中，有一定的概率判断错误。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 比其他常见的算法（如 hash，折半查找）极大节省了空间。

它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

Bloom Filter 的详细介绍：Bloom Filter

2．Bloom-Filter 的基本思想

Bloom-Filter 算法的核心思想就是利用多个不同的 Hash 函数来解决“冲突”。

计算某元素 x 是否在一个集合中，首先能想到的方法就是将所有的已知元素保存起来构成一个集合 R，然后用元素 x 跟这些 R 中的元素一一比较来判断是否存在于集合 R 中；我们可以采用链表等数据结构来实现。但是，随着集合 R 中元素的增加，其占用的内存将越来越大。试想，如果有几千万个不同网页需要下载，所需的内存将足以占用掉整个进程的内存地址空间。即使用 MD5，UUID 这些方法将 URL 转成固定的短小的字符串，内存占用也是相当巨大的。

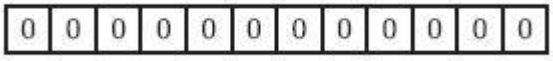
于是，我们会想到用 Hash table 的数据结构，运用一个足够好的 Hash 函数将一个 URL 映射到二进制位数组（位图数组）中的某一位。如果该位已经被置为 1，那么表示该 URL 已经存在。

Hash 存在一个冲突（碰撞）的问题，用同一个 Hash 得到的两个 URL 的值有可能相同。为了减少冲突，我们可以多引入几个 Hash，如果通过其中的一个 Hash 值我们得出某元素不在集合中，那么该元素肯定不在集合中。只有在所有的 Hash 函数告诉我们该元素在集合中时，才能确定该元素存在于集合中。这便是 Bloom-Filter 的基本思想。

原理要点：一是位数组， 而是 k 个独立 hash 函数。

1）位数组：

假设 Bloom Filter 使用一个 m 比特的数组来保存信息，初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0，即 BF 整个数组的元素都设置为 0。

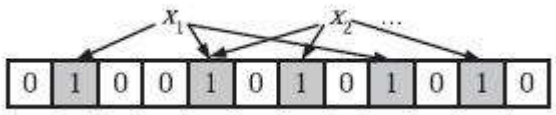


2）添加元素，k 个独立 hash 函数

为了表达 S={x_1，x_2,…,x_n}这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数（Hash Functio n），它们分别将集合中的每个元素映射到{1,…,m}的范围中。

当我们往 Bloom Filter 中增加任意一个元素 x 时候，我们使用 k 个哈希函数得到 k 个哈希值，然后将数组中对应的比特位设置为 1。即第 i 个哈希函数映射的位置 $hash_i(x)$就会被置为 1（1≤i≤k）。

注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1“处）。



3）判断元素是否存在集合

在判断 y 是否属于这个集合时，我们只需要对 y 使用 k 个哈希函数得到 k 个哈希值，如果所有 $hash_i(y)$的位置都是 1（1≤i≤k），即 k 个位置都被设置为 1 了，那么我们就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。下图中 y_1 就不是集合中的元素（因为 y1 有一处指向了“0”位）。y_2 或者属于这个集合，或者刚好是一个 false positive。

显然这 个判断并不保证查找的结果是 100%正确的。

Bloom Filter 的缺点：

1）Bloom Filter 无法从 Bloom Filter 集合中删除一个元素。因为该元素对应的位会牵动到其他的元素。所以一个简单的改进就是 counting Bloom filter，用一个 counter 数组代替位数组，就可以支持删除了。 此外，Bloom Filter 的 hash 函数选择会影响算法的效果。

2）还有一个比较重要的问题，如何根据输入元素个数 n，确定位数组 m 的大小及 hash 函数个数，即 hash 函数选择会影响算法的效果。当 hash 函数个数 k=(ln2)*(m/n)时错误率最小。在错误率不大于 E 的情况 下，m 至少要等于 n*lg(1/E) 才能表示任意 n 个元素的集合。但 m 还应该更大些，因为还要保证 bit 数组里至少一半为 0，则 m 应该>=nlg(1/E)*lge ，大概就是 nlg(1/E)1.44 倍(lg 表示以 2 为底的对数)。

举个例子我们假设错误率为 0.01，则此时 m 应大概是 n 的 13 倍。这样 k 大概是 8 个。

注意:

这里 m 与 n 的单位不同，m 是 bit 为单位，而 n 则是以元素个数为单位(准确的说是不同元素的个数)。通常单个元素的长度都是有很多 bit 的。所以使用 bloom filter 内存上通常都是节省的。

一般 BF 可以与一些 key-value 的数据库一起使用，来加快查询。由于 BF 所用的空间非常小，所有 BF 可以常驻内存。这样子的话，对于大部分不存在的元素，我们只需要访问内存中的 BF 就可以判断出来了，只有一小部分，我们需要访问在硬盘上的 key-value 数据库。从而大大地提高了效率。

一个 Bloom Filter 有以下参数：

m	bit 数组的宽度（bit 数）
n	加入其中的 key 的数量
k	使用的 hash 函数的个数
f	False Positive 的比率

Bloom Filter 的 f 满足下列公式:

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}.$$

在给定 m 和 n 时，能够使 f 最小化的 k 值为:

$$\frac{m}{n}\ln 2 \approx \frac{9m}{13n} \approx 0.7\frac{m}{n},$$

此时给出的 f 为：

$$\left(\frac{1}{2}\right)^k \approx 0.6185^{m/n}.$$

根据以上公式，对于任意给定的 f，我们有：

n = m ln(0.6185) / ln(f)　[1]

同时，我们需要 k 个 hash 来达成这个目标：

k = - ln(f) / ln(2)　　[2]

由于 k 必须取整数，我们在 Bloom Filter 的程序实现中，还应该使用上面的公式来求得实际的 f：

$f = (1 - e^{-kn/m})^k$　　[3]

以上 3 个公式是程序实现 Bloom Filter 的关键公式。

测试：

1. Add extension bloom model
   create extension bloom;

2. Create test table and insert data into test table
   create table tb_bloom_t as
   select trunc(random()*1000000)::int as i1,
   trunc(random()*1000000)::int as i2,
   trunc(random()*1000000)::int as i3,
   trunc(random()*1000000)::int as i4,
   trunc(random()*1000000)::int as i5,
   trunc(random()*1000000)::int as i6,
   trunc(random()*1000000)::int as i7,
   trunc(random()*1000000)::int as i8,
   trunc(random()*1000000)::int as i9,
   trunc(random()*1000000)::int as i10
   from generate_series(1,10000000);

   insert into tb_bloom_c select * from tb_bloom_t;
   insert into tb_bloom select * from tb_bloom_t;

3. Create different index for the two test table
   create index bloomidx on tb_bloom using bloom (i1,i2,i3,i4,i5,i6,i7,i8,i9,i10);
   create index btreeidx_i38 on tb_bloom_c (i3,i8);

   create index btreeidx on tb_bloom_c (i1,i2,i3,i4,i5,i6,i7,i8,i9,i10);

   create index btreeidx_i1 on tb_bloom_c (i1);
   create index btreeidx_i2 on tb_bloom_c (i2);
   create index btreeidx_i3 on tb_bloom_c (i3);
   create index btreeidx_i4 on tb_bloom_c (i4);
   create index btreeidx_i5 on tb_bloom_c (i5);
   create index btreeidx_i6 on tb_bloom_c (i6);
   create index btreeidx_i7 on tb_bloom_c (i7);
   create index btreeidx_i8 on tb_bloom_c (i8);
   create index btreeidx_i9 on tb_bloom_c (i9);
   create index btreeidx_i10 on tb_bloom_c (i10);

4. After creating involved indexes, comparing the size of these indexes and bulk insert lots of data into the table.

--	tb_bloom	tb_bloom_c(i3,i8 独立索引)	tb_bloom_c(全部独立索引)
insert 10000000(1st)	45.2s	1:14 min	7:07 min
insert 10000000(2nd)	44.4s	1:09 min	
insert 10000000(3rd)	44.8s	1:08 min	

postgres@[local]:5432 test# \di+

List of relations

Schema	Name	Type	Owner	Table	Size	Description
public	bloomidx	index	postgres	tb_bloom	153 MB	
public	btreeidx_i3	index	postgres	tb_bloom_c	279 MB	
public	btreeidx_i8	index	postgres	tb_bloom_c	280 MB	

select pg_size_pretty(pg_relation_size('bloomidx'));

```
explain (analyze,verbose,buffers,costs,timing)
select * from tb_bloom where i3=306047 and i8=571233
```

ut pane

ata Output | Explain | Messages | History

	QUERY PLAN text
1	Bitmap Heap Scan on public.tb_bloom (cost=178436.80..178440.82 rows=1 width=40) (actual time=106.882..106.882 rows=0 loops=1)
2	Output: i1, i2, i3, i4, i5, i6, i7, i8, i9, i10
3	Recheck Cond: ((tb_bloom.i3 = 306047) AND (tb_bloom.i8 = 571233))
4	Rows Removed by Index Recheck: 19106
5	Heap Blocks: exact=17079
6	Buffers: shared hit=36687
7	-> Bitmap Index Scan on bloomidx (cost=0.00..178436.80 rows=1 width=0) (actual time=83.302..83.302 rows=19106 loops=1)
8	Index Cond: ((tb_bloom.i3 = 306047) AND (tb_bloom.i8 = 571233))
9	Buffers: shared hit=19608
10	Planning time: 0.075 ms
11	Execution time: 106.911 ms

normal index case 1:

select pg_size_pretty(pg_relation_size('btreeidx_i3'));

select pg_size_pretty(pg_relation_size('btreeidx_i8'));

```
explain (analyze,verbose,buffers,costs,timing)
select * from tb_bloom_c where i3=306047 and i8=571233
```

ut pane

ata Output | Explain | Messages | History

	QUERY PLAN text
1	Bitmap Heap Scan on public.tb_bloom_c (cost=9.28..13.29 rows=1 width=40) (actual time=0.035..0.035 rows=0 loops=1)
2	Output: i1, i2, i3, i4, i5, i6, i7, i8, i9, i10
3	Recheck Cond: ((tb_bloom_c.i8 = 571233) AND (tb_bloom_c.i3 = 306047))
4	Buffers: shared hit=7
5	-> BitmapAnd (cost=9.28..9.28 rows=1 width=0) (actual time=0.034..0.034 rows=0 loops=1)
6	Buffers: shared hit=7
7	-> Bitmap Index Scan on btreeidx_i8 (cost=0.00..4.51 rows=10 width=0) (actual time=0.019..0.019 rows=8 loops=1)
8	Index Cond: (tb_bloom_c.i8 = 571233)
9	Buffers: shared hit=3
10	-> Bitmap Index Scan on btreeidx_i3 (cost=0.00..4.52 rows=11 width=0) (actual time=0.011..0.011 rows=11 loops=1)
11	Index Cond: (tb_bloom_c.i3 = 306047)
12	Buffers: shared hit=4
13	Planning time: 0.086 ms
14	Execution time: 0.056 ms

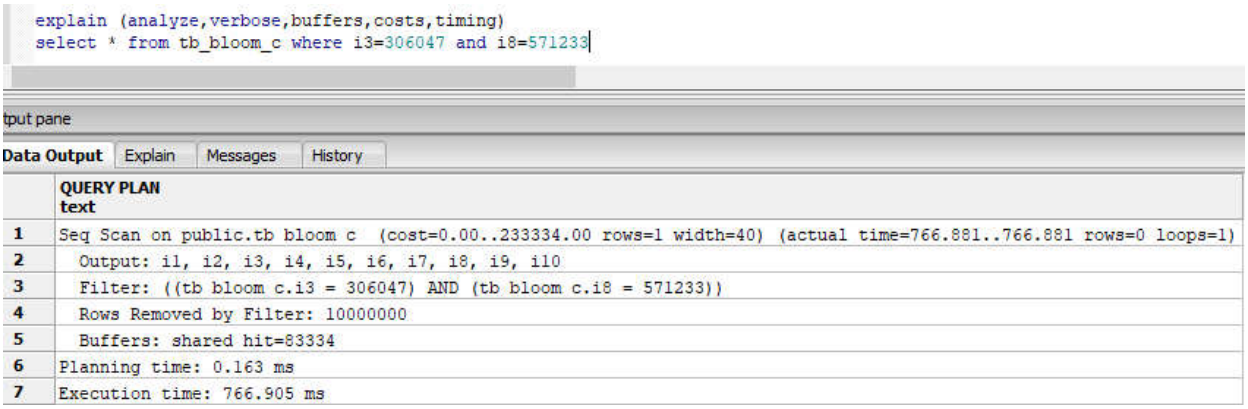
normal index case 2:

select pg_size_pretty(pg_relation_size('btreeidx'));

postgres@[local]:5432 test# \di+

List of relations

```
+--------+----------------------+-------+----------+------------+---------+-------------+
| Schema |         Name         | Type  |  Owner   |   Table    |  Size   | Description |
+--------+----------------------+-------+----------+------------+---------+-------------+
| public | bloomidx             | index | postgres | tb_bloom   | 153 MB  |             |
| public | btreeidx             | index | postgres | tb_bloom_c | 563 MB  |             |
```



```
explain (analyze,verbose,buffers,costs,timing)
select * from tb_bloom_c where i3=306047 and i8=571233
```

tput pane

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Seq Scan on public.tb_bloom_c (cost=0.00..233334.00 rows=1 width=40) (actual time=766.881..766.881 rows=0 loops=1)
2	Output: i1, i2, i3, i4, i5, i6, i7, i8, i9, i10
3	Filter: ((tb_bloom_c.i3 = 306047) AND (tb_bloom_c.i8 = 571233))
4	Rows Removed by Filter: 10000000
5	Buffers: shared hit=83334
6	Planning time: 0.163 ms
7	Execution time: 766.905 ms

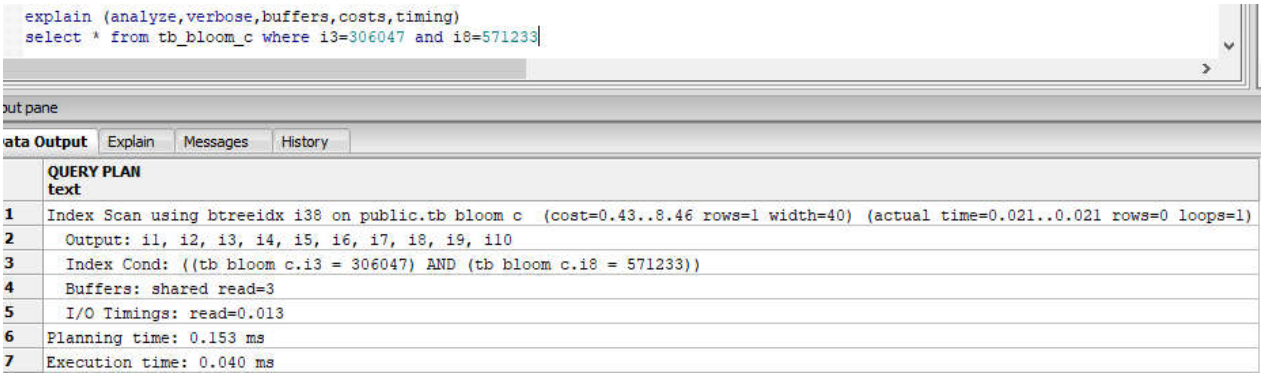
normal index case 3:

select pg_size_pretty(pg_relation_size('btreeidx_i38'));

postgres@[local]:5432 test# \di+

List of relations

```
+--------+----------------------+-------+----------+------------+---------+-------------+
| Schema |         Name         | Type  |  Owner   |   Table    |  Size   | Description |
+--------+----------------------+-------+----------+------------+---------+-------------+
| public | bloomidx             | index | postgres | tb_bloom   | 153 MB  |             |
| public | btreeidx_i38         | index | postgres | tb_bloom_c | 214 MB  |             |
```



```
explain (analyze,verbose,buffers,costs,timing)
select * from tb_bloom_c where i3=306047 and i8=571233
```

ut pane

ata Output | Explain | Messages | History

	QUERY PLAN text
1	Index Scan using btreeidx_i38 on public.tb_bloom_c (cost=0.43..8.46 rows=1 width=40) (actual time=0.021..0.021 rows=0 loops=1)
2	Output: i1, i2, i3, i4, i5, i6, i7, i8, i9, i10
3	Index Cond: ((tb_bloom_c.i3 = 306047) AND (tb_bloom_c.i8 = 571233))
4	Buffers: shared read=3
5	I/O Timings: read=0.013
6	Planning time: 0.153 ms
7	Execution time: 0.040 ms

5. Summary

Bloom index is a more efficient index in some occasions. As for multi columns index and the column combination is random, bloom index is better. Cause of the lower size and more efficient execution, it reduces the cost of maintenance.