# 高级计算机编程实战

## 字符串检索之 Bloomfilter

**熊永平@计算机学院**

# 实验任务回顾

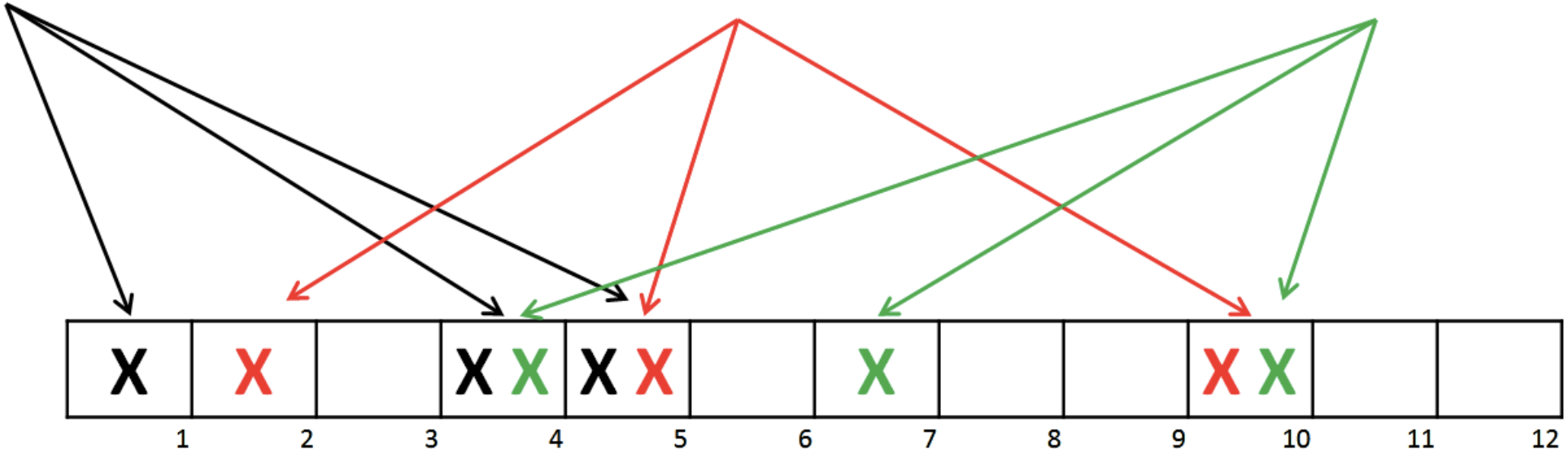
- 当前任务
  - 120万个中文字符串中查找字符串
  - 利用hashtable存储所有字符串在内存
- 任务延伸
  - 1k*127万个字符串？
  - 需要内存1G*10（假定平均3个汉字加一个指针）=10G

# HASH思想的延伸

h1(“oracle”) = 1
h2(“oracle”) = 4
h3(“oracle”) = 5

h1(“database”) = 2
h2(“database”) = 5
h3(“database”) = 10

h1(“filter”) = 4
h2(“filter”) = 7
h3(“filter”) = 10

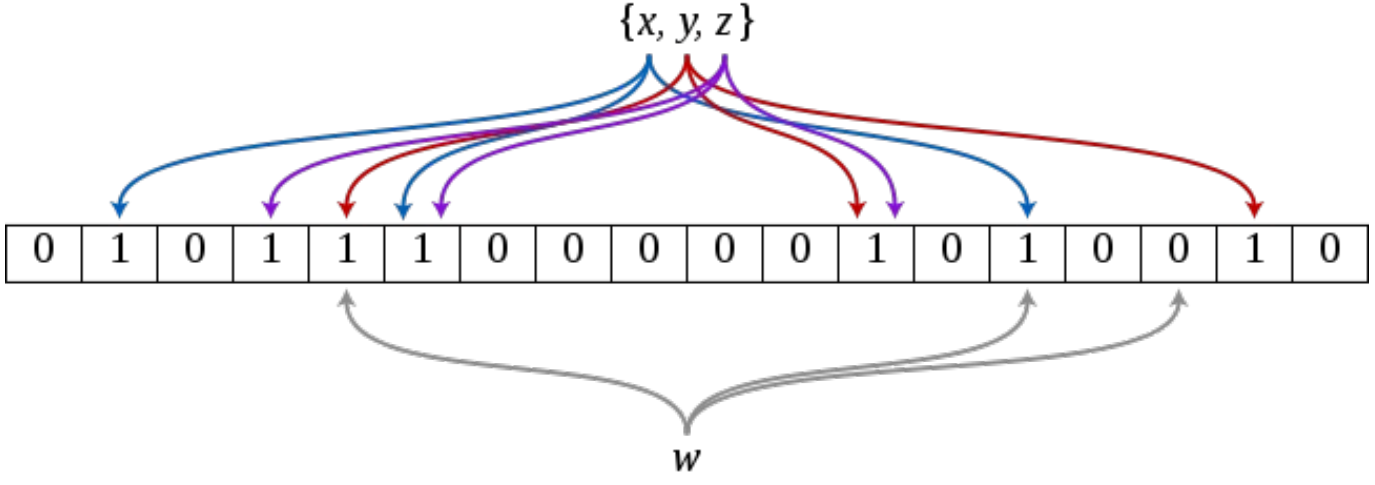


# Bloom Filter（布隆过滤器）

- 背景
  - 1970年Burton Bloom论文《Space/time trade-offs in hash coding with errors》
- 概念
  - **一个很长的二进制向量和一系列随机hash函数**
  - 字典集合查找
  - 准确率换空间思想延伸
- 优点
  - 空间效率和查询时间都远超过一般的算法
- 缺点
  - 有一定的误识别率和删除困难

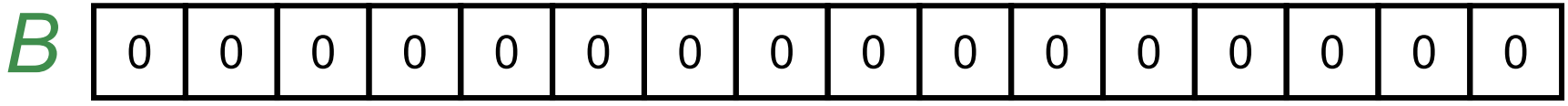
# Bloom Filter构建

- 定义
  - 将n个元素集合S={$x_1$, $x_2$,…,$x_n$}
  - 一个包含m位的二进制位数组存储
  - K个**相互独立**的哈希函数映射到{1,…,m}的范围
  - S集合中的每个元素用k个hash函数映射到，{1,…,m}范围内，将相应的位置为1

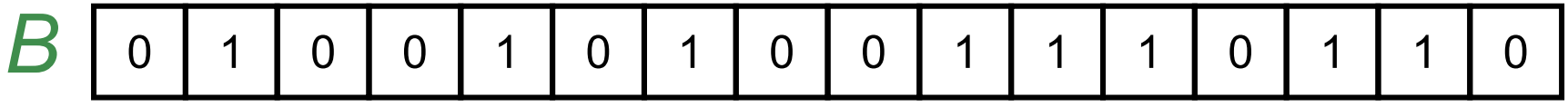


# Bloom Filter原理

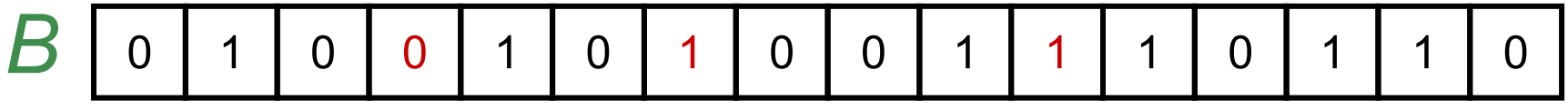
初始化时$m$ 位数组置零



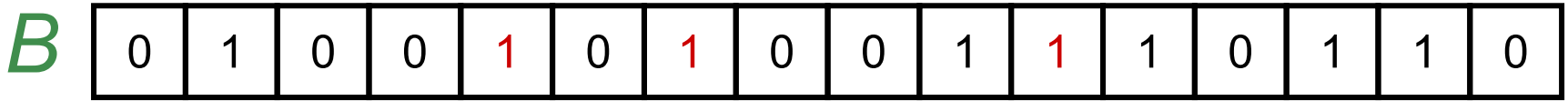

对集合中的每个元素$x_j$ 分别进行$k$ 次hash，If $H_i(x_j) = a$, set $B[a] = 1$.




要检测$y$是否在集合$S$，测试所有$k$ 个$B$ $[H_i(y)]$是否都为1.




可能出现false positive：即所有$k$个值都是1，但$y$不在集合$S$中




$n$ items　　$m = cn$ bits　　$k$ hash functions

# 错误率估计

- 初始位向量为0
- 插入一个元素后，被k个哈希函数（完全独立）映射到位向量后，某一位还是为0的概率是：$\left(1-\frac{1}{m}\right)^k$
- 集合$S=\{x_1, x_2,\ldots,x_n\}$的所有元素都插入到位向量后，某一位还是为0的概率就是

$$p' = \left(1-\frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \qquad \lim_{x\to\infty}\left(1-\frac{1}{x}\right)^{-x} = e$$

- 令ρ为位向量中0的比例，则ρ的数学期望E(ρ)= p'。令$p = e^{-kn/m}$ 在ρ已知时要求的错误率（false positive rate）为：$f = \left(1-e^{-kn/m}\right)^k = (1-p)^k.$

(1-ρ)为位数组中1的比例，$(1-\rho)^k$就表示k次哈希都刚好选中1的区域

M. Mitzenmacher已经证明位向量中0的比例非常集中地分布在它的数学期望值的附近

# 最优的哈希函数个数k

- 问题
  - Bloom Filter要用多个哈希函数将集合映射到位向量中，应该选择几个哈希函数才能使元素查询时的错误率降到最低？
  - 对 $f = \left(1 - e^{-kn/m}\right)^k = (1-p)^k.$
  - 令$g = k \ln(1 - e^{-kn/m})$，让g取到最小，f自然也取到最小
  - 将g写成

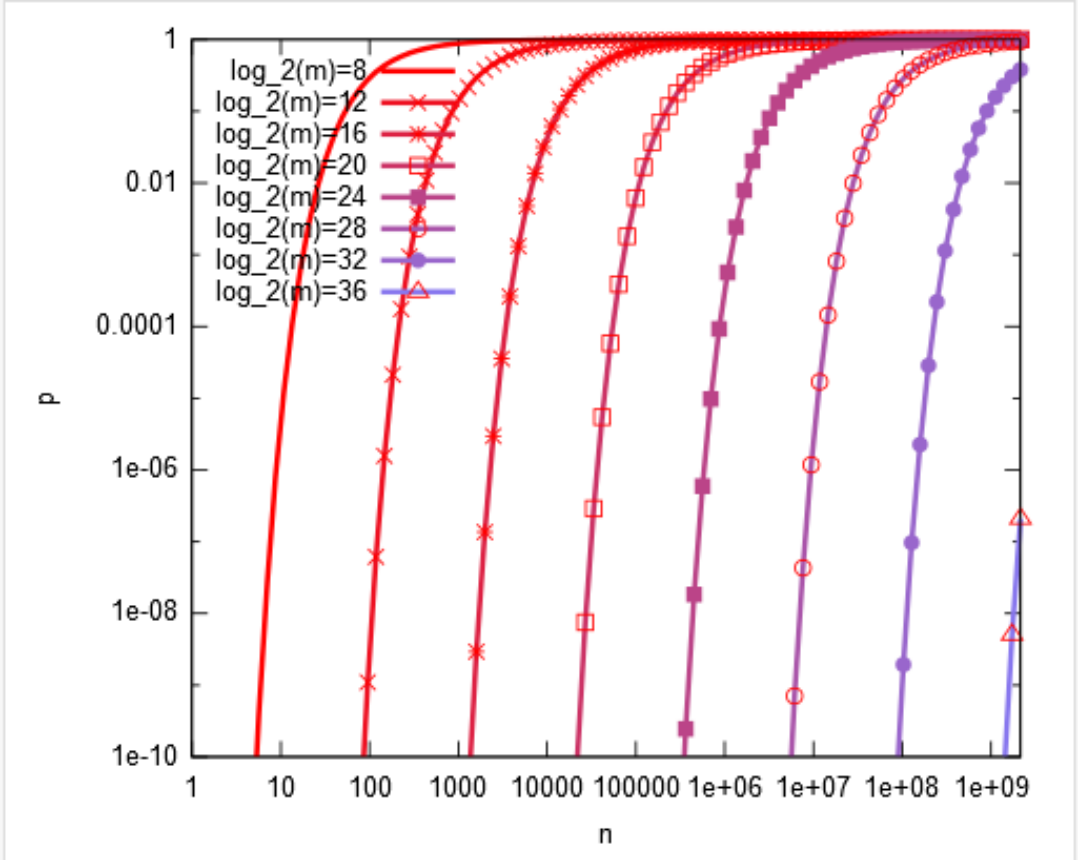
$$g = -\frac{m}{n}\ln(p)\ln(1-p),$$

- 结果
  - 很容易看出当$p = 1/2$，也就是$k = \ln 2 \cdot (m/n)$时，g取得最小值
  - 在这种情况下，最小错误率f等于$(1/2)^k \approx (0.6185)^{m/n}$
  - 另外，注意到p是位数组中某一位仍是0的概率，所以$p = 1/2$对应着位数组中0和1各一半
  - 换句话说，要想保持错误率低，最好让位数组有一半还空着

# 特定错误率下需要的存储空间m

- 推导结果公式：(设允许的false rate为*p*)

$$m = n * \log_2 e * \log_2(1/p)$$
$$\approx n * 1.44 * \log_2(1/p)$$



# 参数取值举例

目标：降低错误率

通过设计布隆过滤器的4个参数来实现

f: 期望的错误率 ---- 0.001
n: 待存储的字符串个数 ----1500w
m: 需开辟的存储空间位数
k：哈希函数的个数

$$m = n * 1.44 * \log_2(1 / f')$$

$$k = 0.693 * m / n$$

m = 215260940 bit = 25M
K = 10

# 测试与参数选择

- 进行3组实验，每组取5个N
  - 取FP1 =0.01%， N=[50W, 100W, 300W, 500W,1000W]
  - 取FP 2=0.001%， N=[50W, 100W, 300W, 500W,1000W]
  - 取FP3 =0.00001%， N=[50W, 100W, 300W, 500W,1000W]

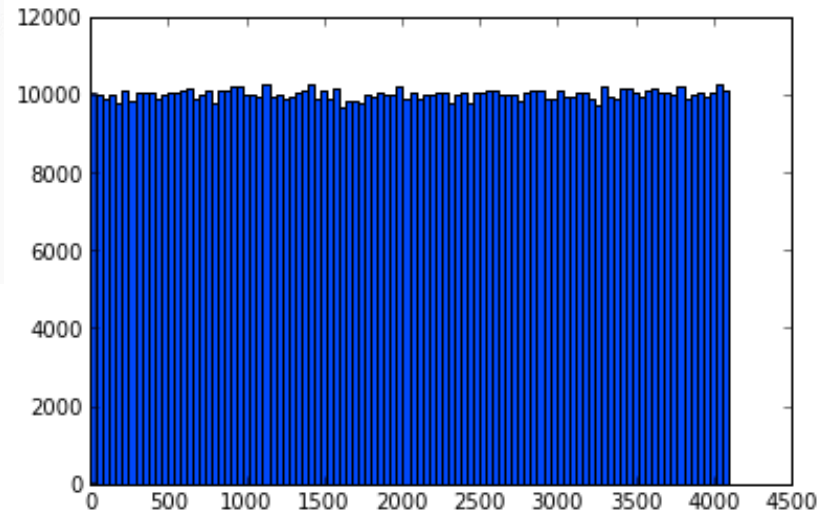
| N | (Vector size)m | | | 内存 | | | (Hash num) k | | | X | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FP1 | FP2 | FP3 | FP1 | FP2 | FP3 | FP1 | FP2 | FP3 | FP1 | FP2 | FP3 |
| 50W | 958W | 1198 W | 1677W | 1M | 1M | 1M | 13 | 17 | 23 | 37091 | 3691 | 38 |
| 100W | 1917 W | 2396 W | 3355W | 2M | 2M | 3M | 13 | 17 | 23 | 36958 | 3770 | 47 |
| 300W | 5751 W | 7188 W | 10064 W | 6M | 8M | 11M | 13 | 17 | 23 | 36585 | 3689 | 38 |
| 500W | 9585 W | 11981 W | 16773 W | 11 M | 14M | 19M | 13 | 17 | 23 | 36569 | 3701 | 45 |
| 1000W | 19170 W | 23962W | 33547 W | 22 M | 28M | 39M | 13 | 17 | 23 | 36533 | 3552 | 41 |

# Hash算法不够

- **HASH算法**
  - The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields.
  - 常用的HASH算法
    - unsigned int RSHash  (char* str, unsigned int len);
    - unsigned int JSHash  (char* str, unsigned int len);
    - unsigned int PJWHash (char* str, unsigned int len);
    - unsigned int ELFHash (char* str, unsigned int len);
    - unsigned int BKDRHash(char* str, unsigned int len);
    - unsigned int SDBMHash(char* str, unsigned int len);
    - unsigned int DJBHash (char* str, unsigned int len);
    - unsigned int DEKHash (char* str, unsigned int len);
    - unsigned int BPHash  (char* str, unsigned int len);
    - unsigned int FNVHash (char* str, unsigned int len);
    - unsigned int APHash  (char* str, unsigned int len);

# murmurhash2/3哈希算法之母

```
uint32_t MurmurHash2 ( const void * key, int len, uint32_t seed )
{
  // 'm' and 'r' are mixing constants generated offline.
  // They're not really 'magic', they just happen to work well.

  const uint32_t m = 0x5bd1e995;
  const int r = 24;

  // Initialize the hash to a 'random' value

  uint32_t h = seed ^ len;

  // Mix 4 bytes at a time into the hash

  const unsigned char * data = (const unsigned char *)key;

  while(len >= 4)
  {
    uint32_t k = *(uint32_t*)data;

    k *= m;
    k ^= k >> r;
    k *= m;

    h *= m;
    h ^= k;

    data += 4;
    len -= 4;
  }

  // Handle the last few bytes of the input array

  switch(len)
  {
  case 3: h ^= data[2] << 16;
  case 2: h ^= data[1] << 8;
  case 1: h ^= data[0];
      h *= m;
  };

  // Do a few final mixes of the hash to ensure the last few
  // bytes are well incorporated.
```



# 程序要求

- **程序名和输入参数**
  - bf_search fp dict.txt string.txt
  - fp：允许错误率，例如：0.0000001 百万分之一
  - dict.txt：127万个字符串
  - 待匹配的1.7万个字符串：string.txt
- **程序实现**
  - 根据错误率和字符串数量动态计算最优向量bit数量
  - 选择合适的哈希函数个数
- **实验结果result.txt**
  - 所有查找到的串，一行一个
    - Keyword1
    - Keyword2

# 报告要求

- **实验报告**
  - 主要数据结构和流程
  - 实验过程
  - 遇到的问题
  - 结果指标：cpu 内存 准确率等
  - 结论和总结


THE END