# 海量数据处理

七月算法 邹博

2015年11月14日

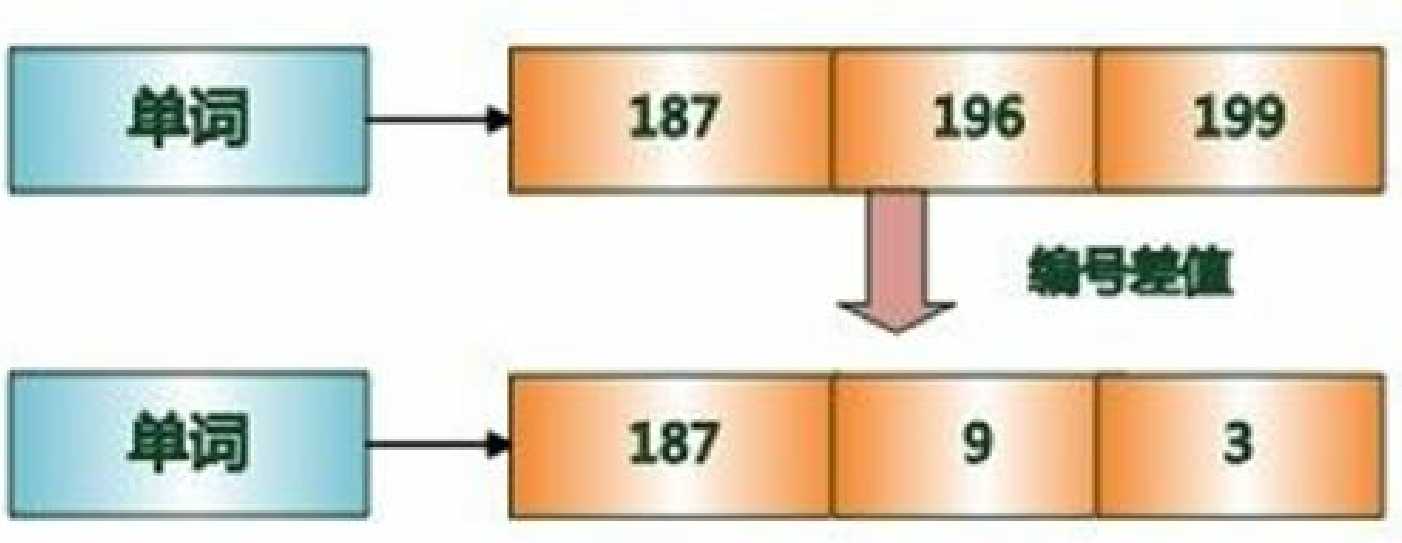
# 倒排索引

- 倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件称为倒排索引文件，简称倒排文件(inverted file)。

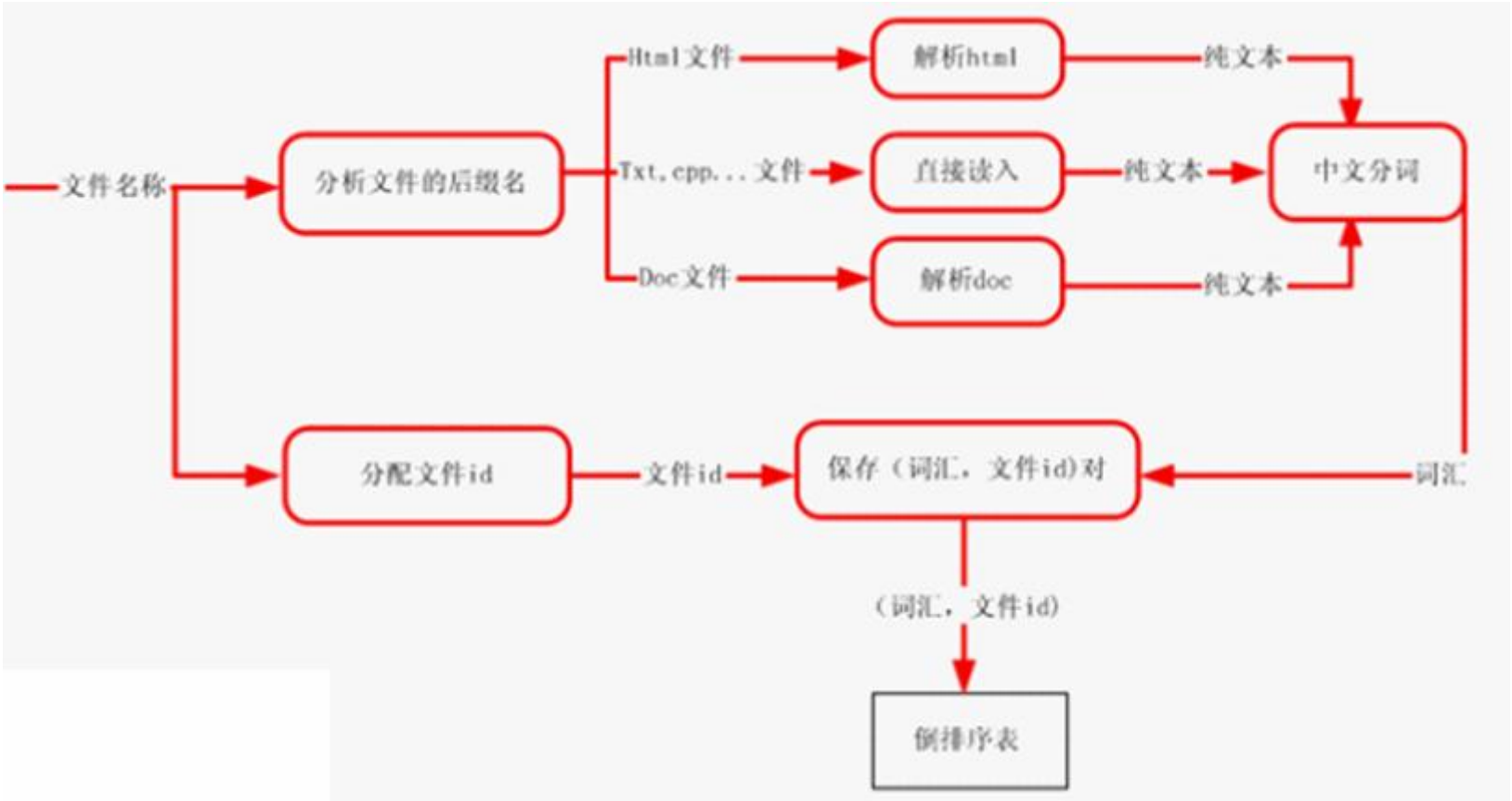
# 倒排列表

- 倒排列表记录了某个单词位于哪些文档中。一般在文档集合里会有很多文档包含某个单词，每个文档会记录文档编号(DocID)，单词在这个文档中出现的次数(TF)及单词在文档中哪些位置出现过等信息，这样与一个文档相关的信息被称做倒排索引项(Posting)，包含这个单词的一系列倒排索引项形成了列表结构，这就是某个单词对应的倒排列表。

# 倒排列表



# 倒排索引


文件名称
分析文件的后缀名
Html文件
Txt,cpp...文件
Doc文件
解析html
直接读入
解析doc
纯文本
纯文本
纯文本
中文分词
分配文件id
文件id
保存（词汇，文件id)对
词汇
（词汇，文件id)
倒排序表

# 更新策略

- 完全重建策略：当新增文档到达一定数量，将新增文档和原先的老文档整合，然后利用静态索引创建方法对所有文档重建索引，新索引建立完成后老索引会被遗弃。此法代价高，但是主流商业搜索引擎一般是采用此方式来维护索引的更新。
- 再合并策略：当新增文档进入系统，解析文档，之后更新内存中维护的临时索引，文档中出现的每个单词，在其倒排表列表末尾追加倒排表列表项；一旦临时索引将指定内存消耗光，即进行一次索引合并，这里需要倒排文件里的倒排列表存放顺序已经按照索引单词字典顺序由低到高排序，这样直接顺序扫描合并即可。其缺点是：因为要生成新的倒排索引文件，所以对老索引中的很多单词，尽管其在倒排列表并未发生任何变化，也需要将其从老索引中取出来并写入新索引中，这样对磁盘消耗是没必要的。
- 原地更新策略：试图改进再合并策略，在原地合并倒排表，这需要提前分配一定的空间给未来插入，如果提前分配的空间不够了需要迁移。实际显示，其索引更新的效率比再合并策略要低。
- 混合策略：出发点是能够结合不同索引更新策略的长处，将不同索引更新策略混合，以形成更高效的方法。

# simHash算法

- 问题的起源：设计比较两篇文章相似度的算法。
- simHash算法分为5个步骤：
  - 分词
  - Hash
  - 加权
  - 合并
  - 降维

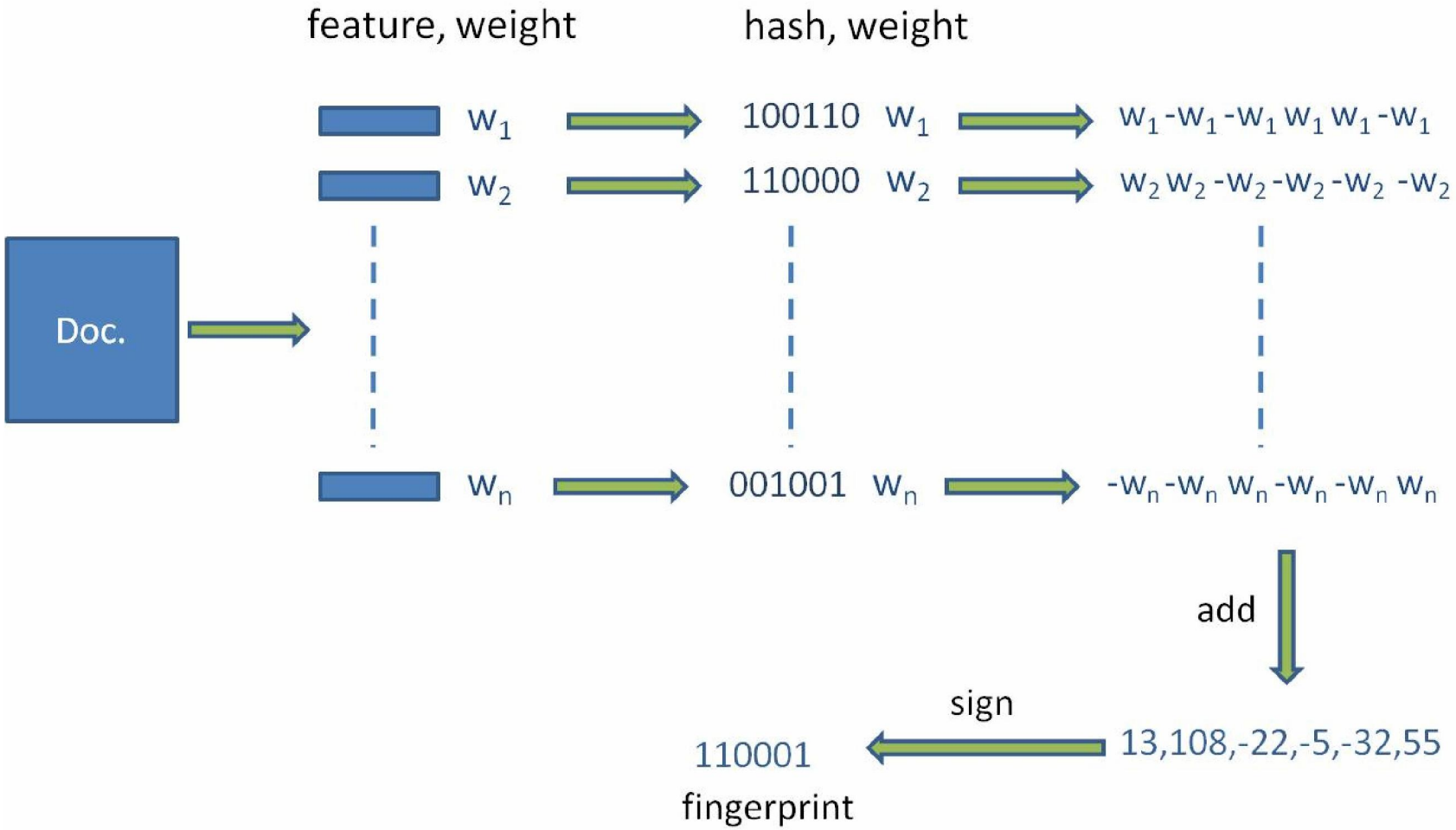
# simHash的具体算法

- 分词
  - 对待考察文档进行分词，把得到的分词称为特征，然后为每一个特征设置N等级别的权重。如给定一段语句：“CSDN博客结构之法算法之道的作者July”，分词后为：“CSDN 博客 结构 之 法 算法 之 道 的 作者 July”，然后为每个特征向量赋予权值：CSDN(4)博客(5)结构(3)之(1)法(2)算法(3)之(1)道(2)的(1)作者(5)July(5)，权重代表了这个特征在整条语句中的重要程度。
- hash
  - 通过hash函数计算各个特征的hash值，hash值为二进制数组成的n位签名。Hash(CSDN)=100101，Hash(博客)=101011。
- 加权
  - W = Hash * weight。W(CSDN)=100101*4=4-4-44-44，W(博客)=101011*5=5-55-555。
- 合并
  - 将上述各个特征的加权结果累加，变成一个序列串。如：“4+5,-4+-5,-4+5,4+-5,-4+5,4+5”，得到“9,-9,1,-1,1”。
- 降维
  - 对于n位签名的累加结果，如果大于0则置1，否则置0，从而得到该语句的simhash值，最后我们便可以根据不同语句simhash的海明距离来判断它们的相似度。例如把上面计算出来的“9,-9,1,-1,1,9”降维，得到“101011”，从而形成它们的simhash签名。

# 分词的权值计算

- 词频-逆文档频率
- TF-IDF(term frequency–inverse document frequency)是一种用于资讯检索与资讯探勘的常用加权技术。TF-IDF是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。
- 字词的重要性随着它在文件中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降：Weight = TF*IDF 。
- 如果某个分词在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。
  - 事实上，该技术在自然语言处理用途广泛，可以配合其它方法一起使用，如余弦距离(反比于相似度)、LDA主题模型等，用于聚类、标签传递算法等后续分析中。

# Simhash


feature, weight
hash, weight
Doc.
w1
w2
wn
100110 w1
110000 w2
001001 wn
w1 -w1 -w1 w1 w1 -w1
w2 w2 -w2 -w2 -w2 -w2
-wn -wn wn -wn -wn wn
add
13,108,-22,-5,-32,55
sign
110001
fingerprint

# simHash的应用

- 每篇文档得到simHash签名值后，接着计算两个签名的海明距离即可。根据经验值，对64位的SimHash值，海明距离在3以内的可认为相似度比较高。
- 海明距离的求法：两个二进制数异或值中1的个数
  - 即：两个二进制数位数不同的个数。

# 对simHash的分块处理

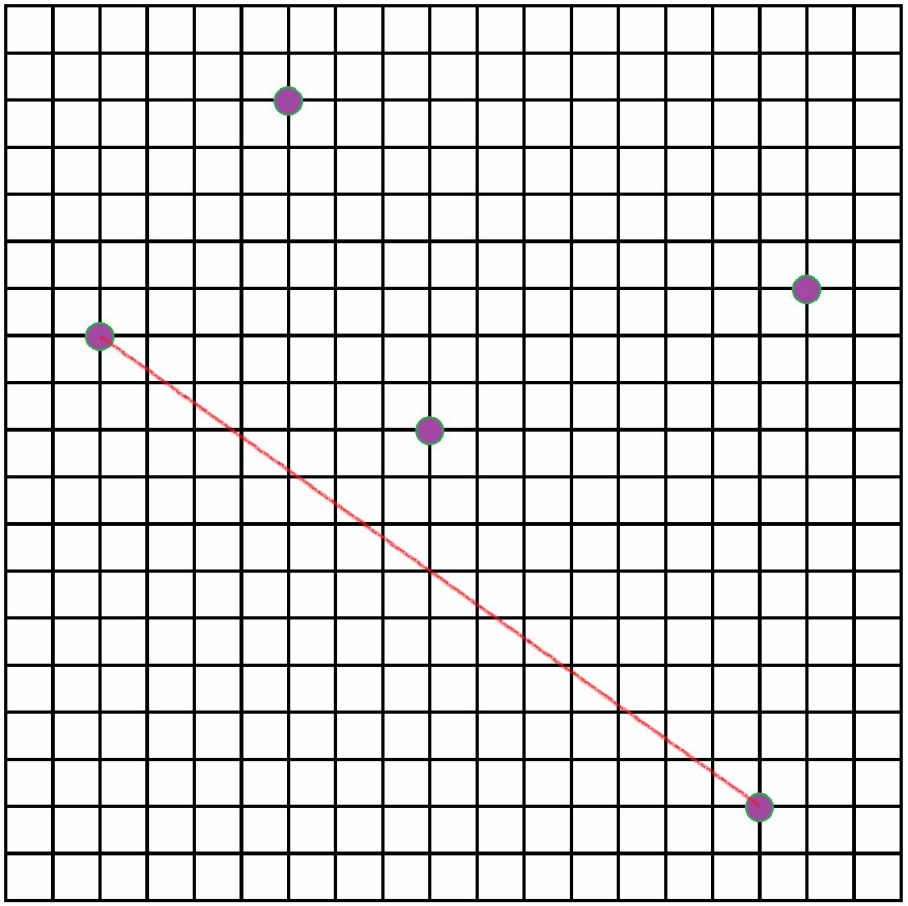
- 如何将其扩展到海量数据呢？譬如如何在海量的样本库中查询与其海明距离在3以内的记录呢？
- 一种方案是查找待查询文本的64位simhash code的所有3位以内变化的组合
- 大约43744个。

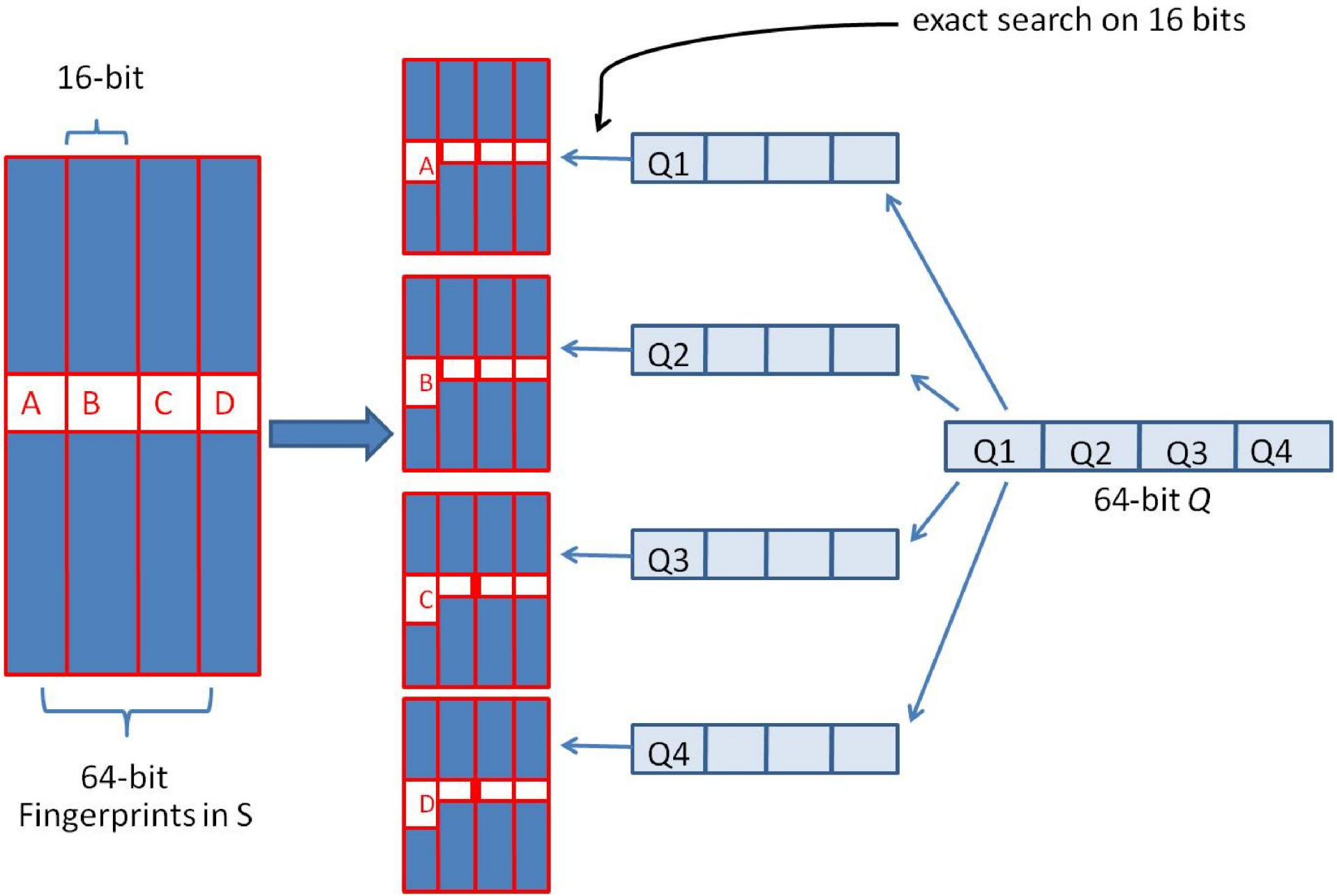
# 倒排索引的应用：对simHash的分块处理

- 把64位的二进制simHash签名均分成4块，每块16位。根据**抽屉原理**，如果两个签名的海明距离在3以内，它们必有一块完全相同。
- 然后把分成的4块中的每一个块分别作为前16位来进行查找，建立**倒排索引**。

# 附：抽屉原理的应用

- 定义：二维坐标系oXY中，坐标(x,y)都是整数的点叫做“格点”。
- 试证明：任取平面上5个格点，它们的连接线段的中点至少有1个是格点。




exact search on 16 bits
16-bit
A
B
C
D
64-bit
Fingerprints in S
A
B
C
D
Q1
Q2
Q3
Q4
Q1
Q2
Q3
Q4
64-bit Q

# 对simHash进一步的思考

- 完全丢掉了位置信息和语义信息
  - 考虑使用WordNet影响Hash值？
  - 考虑使用主题模型、标签传递等非确定性机器学习方法分析语义。

## 进一步的思考

- 允许交换，算一次变换：如meter/metre
  - 能否写出递推关系式？
  - 还能设计出O(n^2)的算法吗？
- 如果计算字符串的语义距离，怎么考虑？
  - WordNet是由Princeton 大学的心理学家，语言学家和计算机工程师联合设计的一种基于认知语言学的英语词典。它不是光把单词以字母顺序排列，而且按照单词的意义组成一个“单词的网络”。

# 倒排索引在实践中的另外一个应用

- 跳跃链表、跳跃表、跳表；
- GIS中的POI(Point of Interest)查询
  - 部分匹配：七月算法在线学院，简称七月算法
  - 跳跃匹配：中国科学院、中科院

# POI信息点搜索总框架

```
void CFileObject::Search(LPCTSTR lpszContent, int nIndex)
{
    if(!lpszContent || !lpszContent[0])
        return;
    CreateSearchTree(nIndex);

    SearchFuzzy(lpszContent, nIndex);
}
```

# 建立查找树

```
bool CFileObject::CreateSearchTree(int nIndex)
{
    int nSymbolType = VerdictType();
    vector<CDataType*>* pAF = GetAF(nSymbolType);
    if(!pAF)
        return false;
    int size = (int)pAF->size();
    if((nIndex < 0) || (nIndex >= size))
        return false;

    if(IsSearchIndexValid(nIndex))
        return true;

    CSearchIndex* pSI = GetSearchIndex(nIndex);
    DWORD dwStart = GetTickCount();
    switch(m_iSymbolType)
    {
        case FO_SYMBOL:
        {
            CreateST(pSI, nIndex, m_vecAcnode);
            break;
        }
        case FO_ROUTE:
        case FO_THREAD:
        {
            CreateST(pSI, nIndex, m_vecRoute);
            break;
        }
        case FO_REGION:
        {
            CreateST(pSI, nIndex, m_vecTopology);
            break;
        }
    }
    return true;
}
```

# 建立查找树

```
bool CFileObject::CreateST(CSearchIndex* pSI, int nIndex, vector<CStamp*>& vecSymbol)
{
    vector<CStamp*>::iterator itEnd = vecSymbol.end();
    CStamp* pSymbol = NULL;
    CSimpleData* pData = NULL;
    LPCTSTR lpszString;
    for(vector<CStamp*>::iterator it = vecSymbol.begin(); it != itEnd; it++)
    {
        pSymbol = *it;
        if(pSymbol && !pSymbol->IsDelete())
        {
            pData = pSymbol->GetAttribute(nIndex);
            if(pData)
            {
                lpszString = pData->GetString();
                pSymbol->SetData(0);
                pSI->AddSymbol(lpszString, pSymbol);
            }
        }
    }

    pSI->SetValid(true);
    return true;
}
```

# 处理Hash冲突

```
bool CSearchIndex::AddSymbol(int nIndex, CStamp* pSymbol)
{
    ASSERT(nIndex >= 0);
    ASSERT(nIndex < SI_COUNT);
    if((nIndex < 0) || (nIndex >= SI_COUNT))
        return false;
    if(!m_dtSI[nIndex])
    {
        m_dtSI[nIndex] = new CBalanceTree<CStamp*>;
    }
    bool bInsert = m_dtSI[nIndex]->Insert(pSymbol);
    return bInsert;
}
```

# Hash查找

```
bool CSearchIndex::Search(int nIndex, CBalanceTree<CStamp*>& avlTree)
{
    ASSERT(nIndex >= 0);
    ASSERT(nIndex < SI_COUNT);
    if((nIndex < 0) || (nIndex >= SI_COUNT))
        return false;
    CBalanceTree<CStamp*>* pTreeSymbol = m_dtSI[nIndex];
    if(!pTreeSymbol)
        return false;
    pTreeSymbol->InOrder(SearchSetData, &avlTree);
    return true;
}
```

# 该复合结构可用性分析

- 假定POI总数为100万，每个POI平均字数为10个，那么，问题总规模为1000万；
- 假定常用汉字为1万个，那么，Hash之后，1万个汉字对应的槽slot平均含有1000个POI信息；
- log1000=9.9658：即，将1000万次搜索，降到10次搜索。
  - 注：以上只是定性考虑，非准确分析

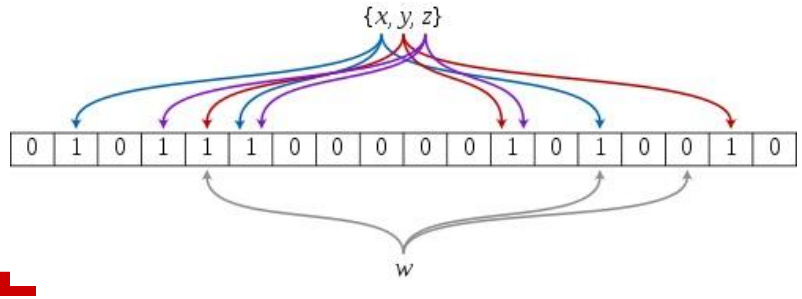
# Bloom Filter

- 布隆过滤器(Bloom Filter)是由Burton Howard Bloom于1970年提出的，它是一种空间高效(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、爬虫(Crawler)的网址判重等问题中经常被用到。
- 哈希表也能用于判断元素是否在集合中，但是Bloom Filter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问题。Bloom Filter可以插入元素，但不可以删除已有元素。集合中的元素越多，误报率(false positive rate)越大，但是不会漏报(false negative)。

# Bloom Filter

- 如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比对来判定是否在集合内：链表、树等数据结构都是这种思路。但是随着集合中元素数目的增加，我们需要的存储空间越来越大，检索速度也越来越慢(O(n),O(logn))。
- 可以利用Bitmap：只要检查相应点是不是1就知道可以集合中有没有某个数。这就是Bloom Filter的基本思想。

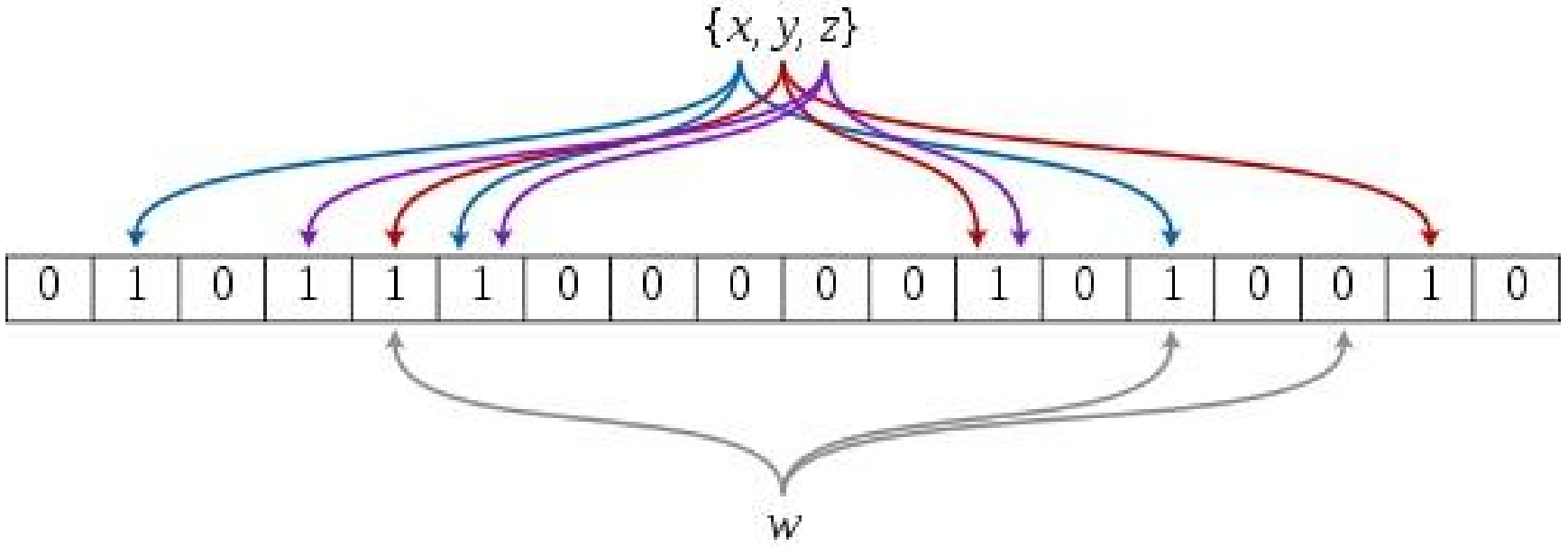
# Bloom Filter算法描述




- 一个空的Bloom Filter是一个有m位的位向量B，每一个bit位都初始化为0。同时，定义k个不同的Hash函数，每个Hash函数都将元素映射到m个不同位置中的一个。
  - 记：n为元素数，m为位向量B的长度(位：槽slot)，k为Hash函数的个数。
- 增加元素x
  - 计算k个Hash(x)的值$(h_1,h_2...h_k)$，将位向量B的相应槽$B[h_1,h_2...h_k]$都设置为1；
- 查询元素x
  - 即判断x是否在集合中，计算k个Hash(x)的值$(h_1,h_2...h_k)$。若$B[h_1,h_2...h_k]$全为1，则x在集合中；若其中任一位不为1，则x不在集合中；
- 删除元素x
  - 不允许删除！因为删除会把相应的k个槽置为0，而其中很有可能有其他元素对应的位。

# Bloom Filter 插入查找数据

- 插入x,y,z
- 判断w是否在该数据集中



# BloomFilter的特点

- 不存在漏报：某个元素在某个集合中，肯定能报出来；
- 可能存在误报：某个元素不在某个集合中，可能也被认为存在：false positive；
- 确定某个元素是否在某个集合中的代价和总的元素数目无关
  - 查询时间复杂度：O(1)

# Bloom Filter参数的确定

- 单个元素某次没有被置位为1的概率为：$1-\frac{1}{m}$
- k个Hash函数中没有一个对其置位的概率为：$\left(1-\frac{1}{m}\right)^k$
- 如果插入n个元素，仍未将其置位的概率为：$\left(1-\frac{1}{m}\right)^{kn}$
- 因此，此位被置位的概率为：$1-\left(1-\frac{1}{m}\right)^{kn}$

# Bloom Filter参数的确定

- 查询中，若某个待查元素对应的k位都被置位，则算法会判定该元素在集合中。因此，该元素被误判的概率(上限)为：

$$q(k)=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}$$

- 考虑到：

$$\left(1-\frac{1}{m}\right)^{kn}=\left(1+\frac{1}{-m}\right)^{-m\cdot\frac{-kn}{m}}=\left(\left(1+\frac{1}{-m}\right)^{-m}\right)^{-\frac{kn}{m}}\approx e^{-\frac{kn}{m}}$$

- 从而：

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

# Bloom Filter参数的确定

- P(k)为幂指函数，取对数后求导：

$$P(k)=\left(1-e^{-\frac{kn}{m}}\right)^k \xrightarrow{令b=e^{\frac{n}{m}}} \left(1-b^{-k}\right)^k$$

$$\Rightarrow \ln P(k)=k\ln\left(1-b^{-k}\right)$$

$$\xrightarrow{取关于k的导数} \frac{1}{P(k)}P'(k)=\ln\left(1-b^{-k}\right)+k\frac{b^{-k}\ln b}{1-b^{-k}}$$

$$\ln\left(1-b^{-k}\right)+k\frac{b^{-k}\ln b}{\left(1-b^{-k}\right)}=0$$

$$\Rightarrow \left(1-b^{-k}\right)\ln\left(1-b^{-k}\right)=b^{-k}\ln b^{-k}$$

$$\Rightarrow 1-b^{-k}=b^{-k} \Rightarrow b^{-k}=\frac{1}{2} \Rightarrow e^{-\frac{kn}{m}}=\frac{1}{2}$$

$$\Rightarrow k=\ln 2\cdot\frac{m}{n}\approx 0.693\cdot\frac{m}{n}$$

$$P(k)=\left(1-\frac{1}{2}\right)^k=2^{-k}=2^{-\ln 2\frac{m}{n}}\approx 0.6185^{\frac{m}{n}}$$

# 参数m、k的确定

- m的计算公式：
  - 由 $$P(k)=\left(1-\frac{1}{2}\right)^{k}=2^{-k}=2^{-\ln 2\frac{m}{n}}\approx 0.6185^{\frac{m}{n}}$$
  - 得 $$P=2^{-\ln 2\frac{m}{n}}\Rightarrow \ln P=\left(\ln 2\cdot\frac{m}{n}\right)\ln 2\Rightarrow m=\frac{\ln P^{-1}}{(\ln 2)^{2}}\cdot n$$
- 此外，k的计算公式：$$k=\ln 2\cdot\frac{m}{n}=\frac{\ln P^{-1}}{\ln 2}$$
- 至此，任意先验给定可接受的错误率，即可确定参数空间m和Hash函数个数k。

# Bloom Filter参数的讨论

- 1.442695041
- 若接收误差率为$10^{-6}$时，需要位的数目为29*n。

$$\begin{cases} m = \dfrac{\ln P^{-1}}{(\ln 2)^2} \cdot n \\ k = \ln 2 \cdot \dfrac{m}{n} = \dfrac{\ln P^{-1}}{\ln 2} \end{cases}$$

| p | m/n | k |
|---|---|---|
| 0.5 | 1.442695041 | 1 |
| 2^-2 | 2.885390082 | 2 |
| 2^-3 | 4.328085123 | 3 |
| 2^-4 | 5.770780164 | 4 |
| 2^-5 | 7.213475204 | 5 |
| 2^-6 | 8.656170245 | 6 |
| 2^-7 | 10.09886529 | 7 |
| 2^-8 | 11.54156033 | 8 |
| 2^-9 | 12.98425537 | 9 |
| 2^-10 | 14.42695041 | 10 |
| 2^-20 | 28.85390082 | 11 |
| 2^-30 | 43.28085123 | 12 |
| 2^-40 | 57.70780164 | 13 |
| 2^-50 | 72.13475204 | 14 |

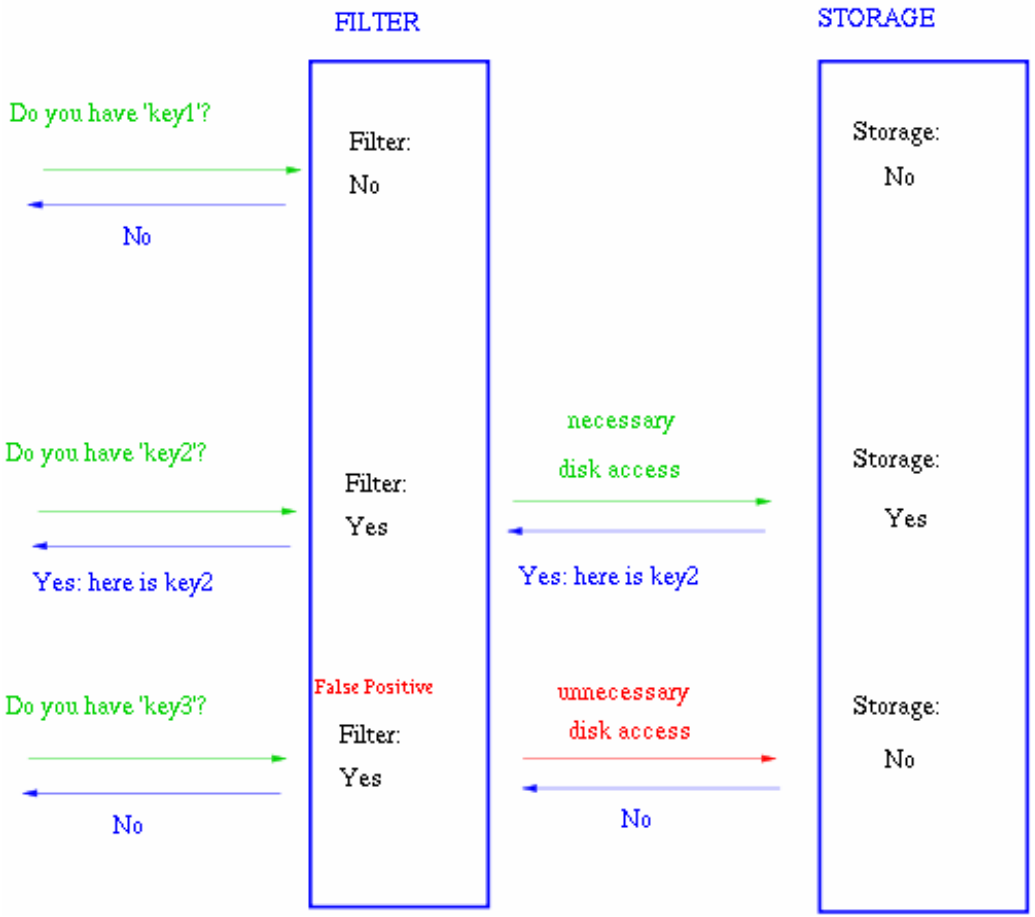
# Bloom Filter的特点

- 优点：相比于其它的数据结构，Bloom Filter在空间和时间方面都有巨大的优势。Bloom Filter存储空间是线性的，插入/查询时间都是常数。另外，Hash函数相互之间没有关系，方便由硬件并行实现。Bloom Filter不存储元素本身，在某些对保密要求非常严格的场合有优势。
- 很容易想到把位向量变成整数数组，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。然而要保证安全的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BloomFilter里面。这一点单凭这个过滤器是无法保证的。另外计数器下溢出也会造成问题(槽的值已经是0了，仍然执行删除操作)。

# BloomFilter用例

- Google著名的分布式数据库Bigtable使用了布隆过滤器来查找不存在的行或列，以减少磁盘查找的IO次数；
- Squid网页代理缓存服务器在cachedigests中使用了BloomFilter；
- Venti文档存储系统采用BloomFilter来检测先前存储的数据；
- SPIN模型检测器使用BloomFilter在大规模验证问题时跟踪可达状态空间；
- Google Chrome浏览器使用BloomFilter加速安全浏览服务；
- 在很多Key-Value系统中也使用BloomFilter来加快查询过程，如Hbase，Accumulo，Leveldb。
  - 一般而言，Value保存在磁盘中，访问磁盘需要花费大量时间，然而使用BloomFilter可以快速判断某个Key是否存在，因此可以避免很多不必要的磁盘IO操作；另外，引入布隆过滤器会带来一定的内存消耗。

# Bloom Filter + Storage结构



# 排序的目的

- 排序本身：得到有序的序列
- 方便查找
  - 长度为N的有序数组，查找某元素的时间复杂度是多少？
  - 长度为N的有序链表，查找某元素的时间复杂度是多少？
    - 单链表、双向链表
    - 如何解决该问题？

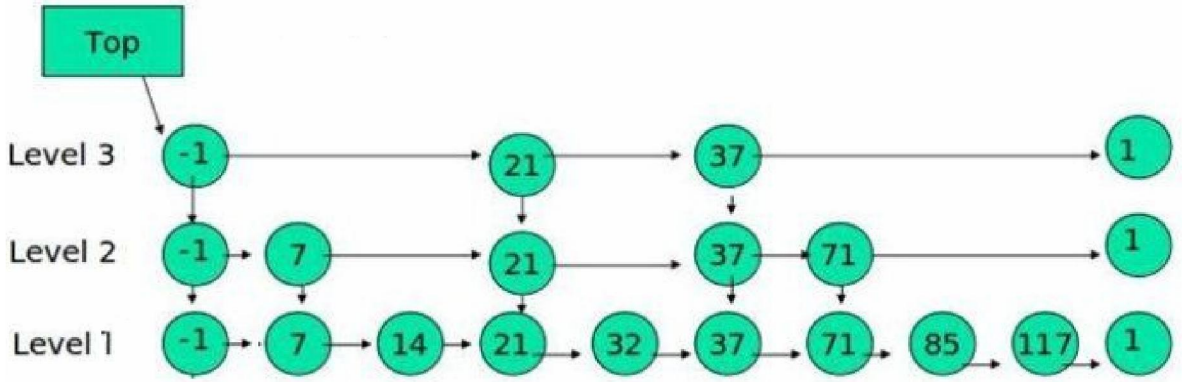
# 跳跃链表(Skip List)

- Treaps/RB-Tree/BTree
- 跳跃链表是一种随机化数据结构，基于并联的链表，其效率与RBTree相当。具有简单、高效、动态(Simple、Effective、Dynamic)的特点。
- 跳跃链表对有序的链表附加辅助结构，在链表中的查找可以快速的跳过部分结点(因此得名)。
  - 查找、增加、删除的期望时间都是O(logN)
    - with high probability(W.H.P. $\approx 1-1/(n^{\alpha})$)

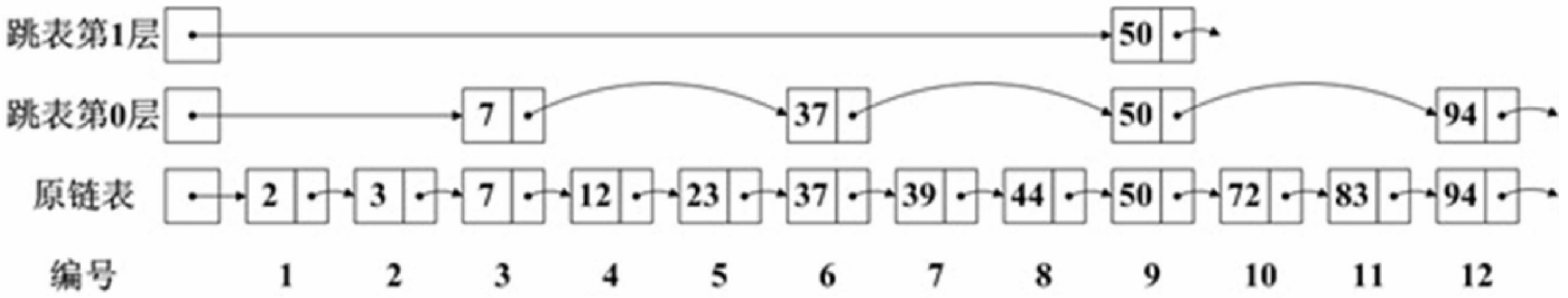
# 跳跃链表(Skip List)

- 跳跃列表在并行计算中非常有用，数据插入可以在跳表的不同部分并行进行，而不用全局的数据结构重新平衡。
- 跳跃列表是按层建造的。底层是一个普通的有序链表。每个更高层都充当下面列表的“快速跑道”，这里在层i中的元素按某个固定的概率p出现在层i+1中。平均起来，每个元素都在1/(1-p)个列表中出现。
  - 思考：为什么是1/(1-p)?

# 跳跃表示例

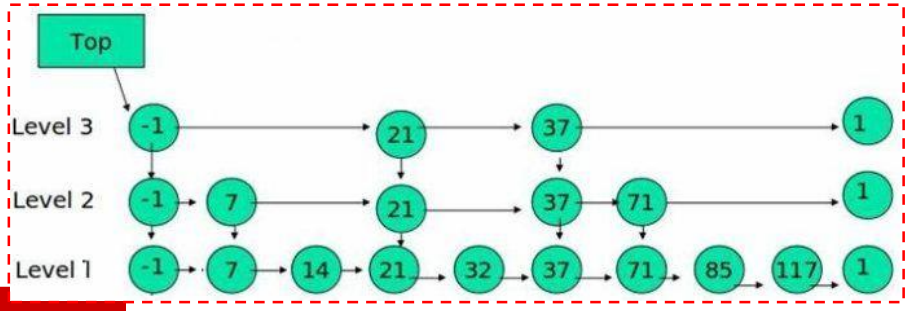



跳跃表：跳跃间隔(Skip Interval) 为 3，层次(Level)共2层




注：各个文献中对于“层”、“间隔”的定义略有差别

# 双层跳表时间计算

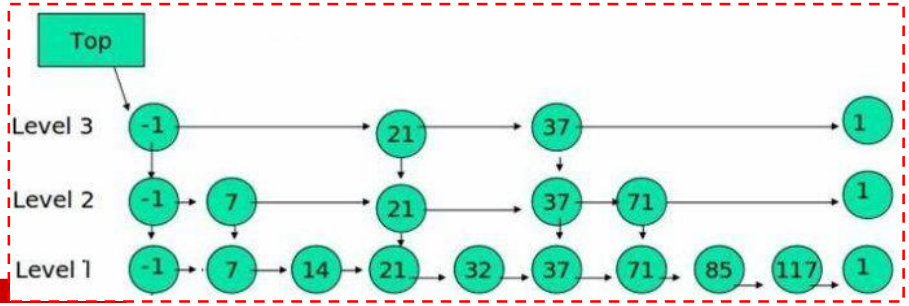



- 粗略估算查找时间：T=L1 + L2/L1(L1是稀疏层，L2是稠密层)
  - 在L2上均匀取值，构成L1；则L2/L1是L1上相邻两个元素在L2上的平均长度
  - L1：在稀疏层的最差查找次数
  - L1/L2：在稀疏层没有找到元素，跳转到稠密层需要找的次数
- 若基本链表的长度为n，即|L2|=n，|L1|为多少，T最小呢？
  - T(x)=x+n/x，对x求导，得到x=sqrt(n)
  - min(T(x))=2sqrt(n)

# 时间复杂度分析

- 粗略估算查找时间：T=L1 + L2/L1 + L3/L2(L1是稀疏层，L2是稠密层，L3是基本层)
  - 在L3上均匀取值，构成L2；在L2上均匀取值，构成L1；则L2/L1是L1上相邻两个元素在L2上的平均长度， L3/L2是L2上相邻两个元素在L3上的平均长度
- 若基本链表的长度为n，即|L3|=n，|L1|、|L2|为多少，T最小呢？
  - T(x,y)=x+y/x+n/y，对x,y求偏导，得到x= $\sqrt[3]{n}$
  - min(T(x,y))= $3*\sqrt[3]{n}$

# 跳表最优时间分析




- 建立k层的辅助链表，可以得到最小时间 $T(n)=k*\sqrt[k]{n}$
- 问题：在n已知的前提下，k取多大最好呢？
- 显然，当k=logN时，T(n)=logN * n^(-logN)
- 问：n^(-logN)等于几？
- 一个很容易在实践中使用的结论是：
  - 当基本链表的数目为N时，共建立k=logN个辅助链表，每个上层链表的长度取下层链表的一半，则能够达到logN的时间复杂度！
- 理想跳表：ideal skip list

# 插入元素

- 随着底层链表的插入，某一段上的数据将不满足理想跳表的要求，需要做些调整。
  - 将底层链表这一段上元素的中位数在拷贝到上层链表中；
  - 重新计算上层链表，使得上层链表仍然是底层链表的1/2；
  - 如果上述操作过程中，上层链表不满足要求，继续上上层链表的操作。
- 新的数据应该在上层甚至上上层链表中吗？因为要找一半的数据放在上层链表(为什么是一半？)，因此：抛硬币！

# 插入元素后的跳表维护

- 考察待需要提升的某段结点。
- 若抛硬币得到的随机数p>0.5，则提升到上层，继续抛硬币，直到p>0.5；
  - 或者到了顶层仍然p>0.5，建立一个新的顶层

# 删除元素

- 在某层链表上找到了该元素，则删除；如果该层链表不是底层链表，跳转到下一层，继续本操作。

# 进一步说明的问题

- 若多次查找，并且相邻查找的元素有相关性(相差不大)，可使用记忆化查找进一步加快查找速度。
- 对关于k的函数$T(n)=k*\sqrt[k]{n}$求导，可计算得到k=lnN处函数取最小值，最小值是elnN。
  - 对于N=10000，k取lnN和logN，两个最小值分布是：25.0363和26.5754。
- 强调：编程方便，尤其方便将增删改查操作扩展成并行算法。
- 跳表用单链表可以实现吗？用双向链表呢？

# 进行10^6次随机操作后的统计结果

| P | 平均操作时间 | 平均列高 | 总结点数 | 每次查找跳跃次数（平均值） | 每次插入跳跃次数（平均值） | 每次删除跳跃次数（平均值） |
|---|---|---|---|---|---|---|
| 2/3 | 0.0024690 ms | 3.004 | 91233 | 39.878 | 41.604 | 41.566 |
| 1/2 | 0.0020180 ms | 1.995 | 60683 | 27.807 | 29.947 | 29.072 |
| 1/e | 0.0019870 ms | 1.584 | 47570 | 27.332 | 28.238 | 28.452 |
| 1/4 | 0.0021720 ms | 1.330 | 40478 | 28.726 | 29.472 | 29.664 |
| 1/8 | 0.0026880 ms | 1.144 | 34420 | 35.147 | 35.821 | 36.007 |

进行$10^6$次随机操作后的统计结果

# Code：Structure

```
typedef struct tagSSkipNode
{
    int value;
    tagSSkipNode* pNext;
    tagSSkipNode* pNextLayer;

    tagSSkipNode(int v) : value(v), pNext(NULL), pNextLayer(NULL) {}
}SSkipNode;
```

```
class CSkipList
{
private:
    SSkipNode* m_pHead;
    int m_nSize;

public:
    CSkipList()
    {
        m_pHead = new SSkipNode(0);
        m_nSize = 0;
    }

    SSkipNode* Find(int value) const;
    bool Insert(int value);
    bool Delete(int value);
    SSkipNode* FindIndex(int n);
    int GetSize() const {return m_nSize;}
    bool IsEmpty() const {return (m_nSize <= 0);}
    void Print() const;
private:
    void PrintLayer(const SSkipNode* pNode) const;
    bool IsSuccess() const
    {
        return rand() > RAND_MAX * 0.36787944117;
    }
};
```

# Code：Find

```
SSkipNode* CSkipList::Find(int value) const
{
    if(!m_pHead)
        return NULL;
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    while(true)
    {
        while(cur && (cur->value < value))  //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value)
            return cur;
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;
    }
    return NULL;
}
```

# Code：Insert

```
bool CSkipList::Insert(int value)
{
    if(!m_pHead->pNext)
    {
        m_pHead->pNext = new SSkipNode(value);
        m_nSize = 1;
        return true;
    }
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    stack<SSkipNode*> path;
    while(true)
    {
        while(cur && (cur->value < value))  //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value)  //已经存在
            return false;
        path.push(pre); //记录插入点
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;
    }
    //插入到pre的后面，cur的前面
    SSkipNode* now = new SSkipNode(value);
    now->pNext = cur;
    pre->pNext = now;
    //随机上升
    SSkipNode* nowInLayer;
    SSkipNode* pLayerHead;
    while(!path.empty())
    {
        if(!IsSuccess())
            break;
        path.pop();
        //得到层的插入位置
        if(path.empty())
            pre = m_pHead;
        else
            pre = path.top();

        //生成结点
        nowInLayer = new SSkipNode(value);
        nowInLayer->pNextLayer = now;
        if(path.empty())    //说明顶层后仍然成功，则新建层
        {
            pLayerHead = new SSkipNode(0);  //生成层的新头指针
            pLayerHead->pNext = m_pHead->pNext;
            pLayerHead->pNextLayer = m_pHead->pNextLayer;
            m_pHead->pNextLayer = pLayerHead;   //退化到下一层
            m_pHead->pNext = nowInLayer;
        }
        else
        {
            nowInLayer->pNext = pre->pNext;
            pre->pNext = nowInLayer;
        }

        //为下次上升做准备
        now = nowInLayer;
    }
    m_nSize++;
    return true;
}
```

# Insert part1

```
bool CSkipList::Insert(int value)
{
    if(!m_pHead->pNext)
    {
        m_pHead->pNext = new SSkipNode(value);
        m_nSize = 1;
        return true;
    }
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    stack<SSkipNode*> path;
    while(true)
    {
        while(cur && (cur->value < value))  //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value)  //已经存在
            return false;
        path.push(pre); //记录插入点
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;
    }
    //插入到pre的后面，cur的前面
    SSkipNode* now = new SSkipNode(value);
    now->pNext = cur;
    pre->pNext = now;
```

# Insert part2

```
//随机上升
SSkipNode* nowInLayer;
SSkipNode* pLayerHead;
while(!path.empty())
{
    if(!IsSuccess())
        break;
    path.pop();
    //得到层的插入位置
    if(path.empty())
        pre = m_pHead;
    else
        pre = path.top();

    //生成结点
    nowInLayer = new SSkipNode(value);
    nowInLayer->pNextLayer = now;
    if(path.empty())    //说明顶层后仍然成功，则新建层
    {
        pLayerHead = new SSkipNode(0);  //生成层的新头指针
        pLayerHead->pNext = m_pHead->pNext;
        pLayerHead->pNextLayer = m_pHead->pNextLayer;
        m_pHead->pNextLayer = pLayerHead;   //退化到下一层
        m_pHead->pNext = nowInLayer;
    }
    else
    {
        nowInLayer->pNext = pre->pNext;
        pre->pNext = nowInLayer;
    }

    //为下次上升做准备
    now = nowInLayer;
}
m_nSize++;
return true;
}
```

# Code：Delete

```
bool CSkipList::Delete(int value)
{
    if(!m_pHead)
        return false;
    SSkipNode* pre = m_pHead;
    SSkipNode* cur = m_pHead->pNext;
    SSkipNode* pHeadPre = NULL;
    SSkipNode* pHead = m_pHead;
    bool bDelete = false;
    while(true)
    {
        while(cur && (cur->value < value))  //当前值小，则遍历下一个
        {
            pre = cur;
            cur = cur->pNext;
        }
        if(cur && cur->value == value)
        {
            bDelete = true;
            pre->pNext = cur->pNext;
            delete cur;
            if(!pHead->pNext)   //该层没有元素，则删除该层
            {
                if(pHead == m_pHead)    //顶层
                {
                    SSkipNode* pNL = m_pHead->pNextLayer;
                    m_pHead->pNextLayer = pNL ? pNL->pNextLayer : NULL;
                    m_pHead->pNext = pNL ? pNL->pNext : NULL;
                    delete pNL;
                }
                else
                {
                    pHeadPre->pNextLayer = pHead->pNextLayer;
                    delete pHead;
                    pHead = pHeadPre;
                }
                pre = pHead;
                cur = pre->pNext;
                continue;   //删除该层后，pre/cur已经向下移动了一层
            }
        }
        if(!pre->pNextLayer)
            break;
        pre = pre->pNextLayer;
        cur = pre->pNext;

        pHeadPre = pHead;
        pHead = pHead->pNextLayer;
    }
    m_nSize--;
    return bDelete;
}
```

# Delete part

```
if(!m_pHead)
    return false;
SSkipNode* pre = m_pHead;
SSkipNode* cur = m_pHead->pNext;
SSkipNode* pHeadPre = NULL;
SSkipNode* pHead = m_pHead;
bool bDelete = false;
```

```
while(true)
{
    while(cur && (cur->value < value))  //当前值小，则遍历下一个
    {
        pre = cur;
        cur = cur->pNext;
    }
    if(cur && cur->value == value)
    {
        bDelete = true;
        pre->pNext = cur->pNext;
        delete cur;
        if(!pHead->pNext)   //该层没有元素，则删除该层
        {
            if(pHead == m_pHead)    //顶层
            {
                SSkipNode* pNL = m_pHead->pNextLayer;
                m_pHead->pNextLayer = pNL ? pNL->pNextLayer : NULL;
                m_pHead->pNext = pNL ? pNL->pNext : NULL;
                delete pNL;
            }
            else
            {
                pHeadPre->pNextLayer = pHead->pNextLayer;
                delete pHead;
                pHead = pHeadPre;
            }
            pre = pHead;
            cur = pre->pNext;
            continue;   //删除该层后，pre/cur已经向下移动了一层
        }
    }
    if(!pre->pNextLayer)
        break;
    pre = pre->pNextLayer;
    cur = pre->pNext;

    pHeadPre = pHead;
    pHead = pHead->pNextLayer;
}
```

```
m_nSize--;
return bDelete;
```

# Code：Test

```
int _tmain(int argc, _TCHAR* argv[])
{
    CSkipList sl;
    int i;
    for(i = 0; i < 100; i++)     //随机插入数据
        sl.Insert(rand() % 100);
    sl.Print();
    while(!sl.IsEmpty())     //随机删除数据
    {
        SSkipNode* p = sl.FindIndex(rand()%sl.GetSize());
        if(p)
        {
            int num = p->value;
            if(sl.Delete(num))
                cout << "Delete " << num << endl;
            else
                cout << "No Delete " << num << endl;
            sl.Print();
            cout<< "================\n";
        }
    }
    return 0;
}
```

# Test

→ 3↓ 44↓ 69↓ 82↓

→ 3↓ 5↓ 44↓ 69↓ 82↓

→ 3↓ 5↓ 44↓ 47↓ 58↓ 69↓ 82↓

→ 0↓ 3↓ 5↓ 11↓ 44↓ 47↓ 58↓ 69↓ 82↓ 91↓

→ 0 3 5 11 24 27 41 44 47 53 58 61 67 69 82 91 95

==========

→ 3↓ 44↓ 69↓ 82↓

→ 3↓ 5↓ 44↓ 69↓ 82↓

→ 3↓ 5↓ 44↓ 47↓ 69↓ 82↓

→ 0↓ 3↓ 5↓ 11↓ 44↓ 47↓ 69↓ 82↓ 91↓

→ 0 3 5 11 24 27 41 44 47 53 61 67 69 82 91 95

# MD5

- MD5(Message Digest Algorithm)，消息摘要算法，为计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护。
  - 文件号RFC 1321(R.Rivest,MIT Laboratory for Computer Science and RSA Data Security Inc. April 1992)

# MD5的框架理解

- 对于长度为512bit的信息，可以通过处理，得到长度为128bit的摘要。
- 初始化摘要：
  0x0123456789ABCDEFFEDCBA9876543210
  - A=0x01234567　　B=0x89ABCDEF
  - C=0xFEDCBA98　　D=0x76543210
- 现在的工作，是要用长度为512位的信息，变换初始摘要。

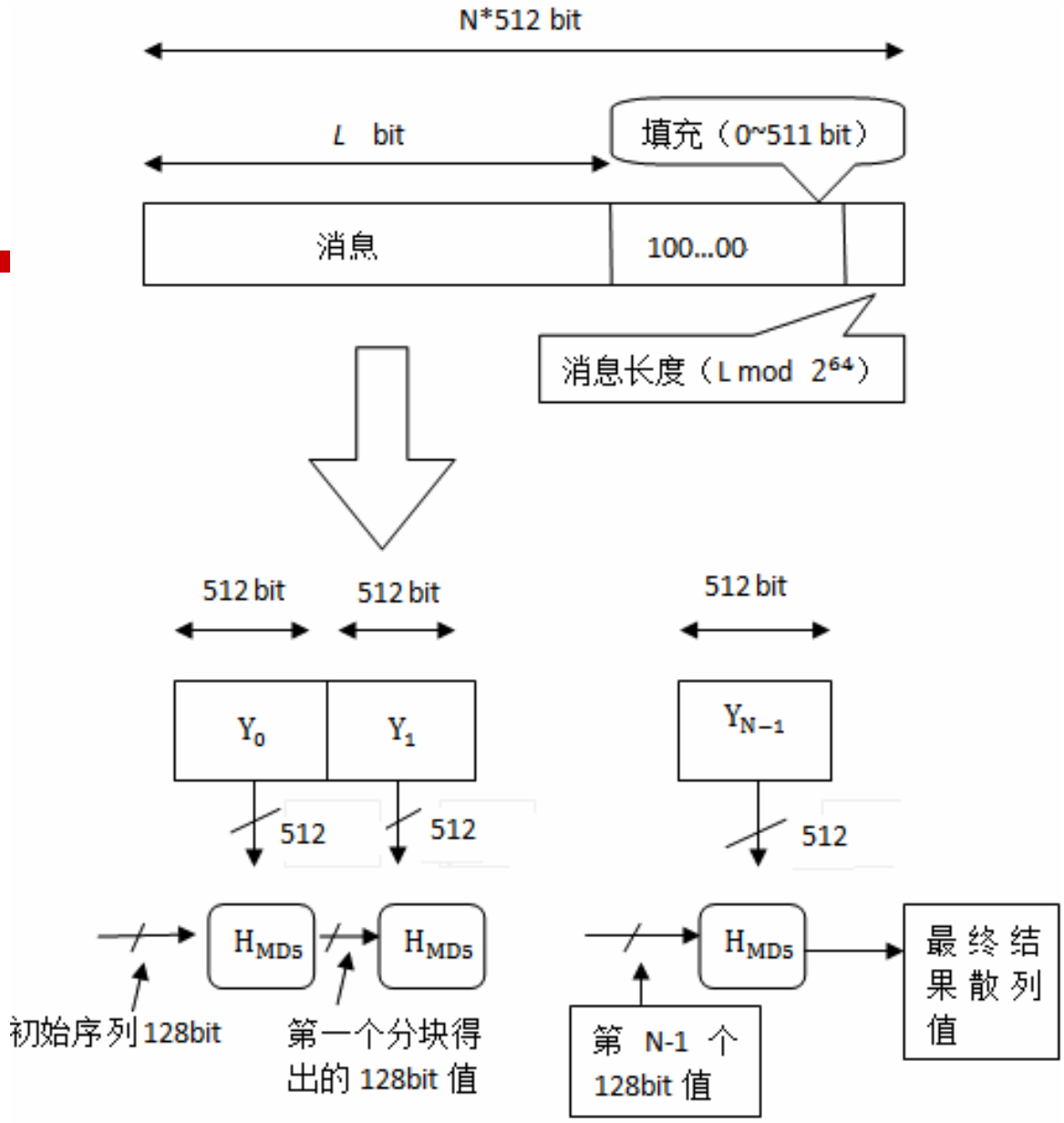
# MD5的总体理解

- 定义变量a,b,c,d,分别记录A,B,C,D;
- 将512bit的信息按照32bit一组，分成16组；分别记为$M_j$ $(0 \leqslant j \leqslant 15)$;
- 取某正数s、$t_k$，定义函数：
  $FF(a,b,c,d,M_j,s,t_k)=(a+F(b,c,d)+M_j+t_k)<<s$
- 利用$M_j$分别进行信息提取，将结果保存到a
  - 其中，$F(X,Y,Z) =(X \& Y) | (\sim X \& Z)$

# MD5的总体理解

- 经过以上16次变换，a,b,c,d带有了$M_j$的信息
- 事实上经过四轮这样的变换(4轮*16次=64次)
- 经过64次变换后，将a,b,c,d累加给A,B,C,D
- 此时，完成了512bit信息的提取；进行下一个512bit信息的相同操作

# MD5框架


N*512 bit
L bit
填充（0~511 bit）
消息
100...00
消息长度（L mod 2^64）
512 bit
512 bit
512 bit
Y0
Y1
YN−1
512
512
512
HMD5
HMD5
HMD5
初始序列128bit
第一个分块得出的128bit值
第 N-1 个128bit值
最终结果散列值

# MD5细致算法

- 在算法中，首先需要对信息进行填充，使其长度对512求余的结果等于448。
- 填充的方法如下，在信息的后面填充一个1和若干个0，直到满足上面的条件。然后，在这个结果后面附加一个以64位二进制表示的原始信息长度Length。
- 经过这两步的处理，数据总长度为N*512+448+64=(N+1)*512，即长度恰好是512的整数倍。

# MD5的细致算法

- 定义四个非线性函数：
  - F(X, Y, Z)=(X&Y)|((~X)&Z)　　H(X, Y, Z)=X^Y^Z
  - G(X, Y, Z)=(X&Z)|(Y&(~Z))　　I(X, Y, Z)=Y^(X|(~Z))
- $M_j$表示数据的第j个子分组(0≤j≤15，循环表示)，常数$t_k$是$2^{32}$*abs(sin(k))的整数部分，1≤k≤64，单位是弧度。
- $FF(a,b,c,d,M_j,s,t_k)=(a+F(b,c,d)+M_j+t_k)<<s$
- $GG(a,b,c,d,M_j,s,t_k)=(b+G(b,c,d)+M_j+t_k)<<s$
- $HH(a,b,c,d,M_j,s,t_k)=(b+H(b,c,d)+M_j+t_k)<<s$
- $II(a,b,c,d,M_j,s,t_k)=(b+I(b,c,d)+M_j+t_k)<<s$

# 64次操作

第一轮
FF(a, b, c, d, M0, 7, 0xd76aa478)
FF(d, a, b, c, M1, 12, 0xe8c7b756)
FF(c, d, a, b, M2, 17, 0x242070db)
FF(b, c, d, a, M3, 22, 0xc1bdceee)
FF(a, b, c, d, M4, 7, 0xf57c0faf)
FF(d, a, b, c, M5, 12, 0x4787c62a)
FF(c, d, a, b, M6, 17, 0xa8304613)
FF(b, c, d, a, M7, 22, 0xfd469501)
FF(a, b, c, d, M8, 7, 0x698098d8)
FF(d, a, b, c, M9, 12, 0x8b44f7af)
FF(c, d, a, b, M10, 17, 0xffff5bb1)
FF(b, c, d, a, M11, 22, 0x895cd7be)
FF(a, b, c, d, M12, 7, 0x6b901122)
FF(d, a, b, c, M13, 12, 0xfd987193)
FF(c, d, a, b, M14, 17, 0xa679438e)
FF(b, c, d, a, M15, 22, 0x49b40821)

第二轮
GG(a, b, c, d, M1, 5, 0xf61e2562)
GG(d, a, b, c, M6, 9, 0xc040b340)
GG(c, d, a, b, M11, 14, 0x265e5a51)
GG(b, c, d, a, M0, 20, 0xe9b6c7aa)
GG(a, b, c, d, M5, 5, 0xd62f105d)
GG(d, a, b, c, M10, 9, 0x02441453)
GG(c, d, a, b, M15, 14, 0xd8a1e681)
GG(b, c, d, a, M4, 20, 0xe7d3fbc8)
GG(a, b, c, d, M9, 5, 0x21e1cde6)
GG(d, a, b, c, M14, 9, 0xc33707d6)
GG(c, d, a, b, M3, 14, 0xf4d50d87)
GG(b, c, d, a, M8, 20, 0x455a14ed)
GG(a, b, c, d, M13, 5, 0xa9e3e905)
GG(d, a, b, c, M2, 9, 0xfcefa3f8)
GG(c, d, a, b, M7, 14, 0x676f02d9)
GG(b, c, d, a, M12, 20, 0x8d2a4c8a)

第三轮
HH(a, b, c, d, M5, 4, 0xfffa3942)
HH(d, a, b, c, M8, 11, 0x8771f681)
HH(c, d, a, b, M11, 16, 0x6d9d6122)
HH(b, c, d, a, M14, 23, 0xfde5380c)
HH(a, b, c, d, M1, 4, 0xa4beea44)
HH(d, a, b, c, M4, 11, 0x4bdecfa9)
HH(c, d, a, b, M7, 16, 0xf6bb4b60)
HH(b, c, d, a, M10, 23, 0xbebfbc70)
HH(a, b, c, d, M13, 4, 0x289b7ec6)
HH(d, a, b, c, M0, 11, 0xeaa127fa)
HH(c, d, a, b, M3, 16, 0xd4ef3085)
HH(b, c, d, a, M6, 23, 0x04881d05)
HH(a, b, c, d, M9, 4, 0xd9d4d039)
HH(d, a, b, c, M12, 11, 0xe6db99e5)
HH(c, d, a, b, M15, 16, 0x1fa27cf8)
HH(b, c, d, a, M2, 23, 0xc4ac5665)

第四轮
II(a, b, c, d, M0, 6, 0xf4292244)
II(d, a, b, c, M7, 10, 0x432aff97)
II(c, d, a, b, M14, 15, 0xab9423a7)
II(b, c, d, a, M5, 21, 0xfc93a039)
II(a, b, c, d, M12, 6, 0x655b59c3)
II(d, a, b, c, M3, 10, 0x8f0ccc92)
II(c, d, a, b, M10, 15, 0xffeff47d)
II(b, c, d, a, M1, 21, 0x85845dd1)
II(a, b, c, d, M8, 6, 0x6fa87e4f)
II(d, a, b, c, M15, 10, 0xfe2ce6e0)
II(c, d, a, b, M6, 15, 0xa3014314)
II(b, c, d, a, M13, 21, 0x4e0811a1)
II(a, b, c, d, M4, 6, 0xf7537e82)
II(d, a, b, c, M11, 10, 0xbd3af235)
II(c, d, a, b, M2, 15, 0x2ad7d2bb)
II(b, c, d, a, M9, 21, 0xeb86d391)

# 海量数据系统设计小结

- 相信理论，重视实践
  - O(NlogN)优于$O(N^2)$
  - $O(N^{2.81})$ VS $O(N^3)$
  - 内存命中、并行开销
- 融会贯通，适度改造
  - POI结构/R树变体

# 思考


ask.julyedu.com/question/130
七月算法 首页 算法 机器学习 视频 问答 题库APP 搜索问题、主题或人
问答社区 / 全部问题 / 面试 / 新浪微博长URL的压缩（存储、查找相关）
面试题
新浪微博长URL的压缩（存储、查找相关）
取消关注 14
【题目】
新浪微博发布内容要求字符不超过140，但是用户如果在发布内容中有很长的url时，会认为是很多字符。所以新浪上发布内容包含一个URL时，时把他压缩成一个TinyURL(缩小)。比如：
(因为无法发站外链接，所以去掉了链接中的http://头)
输入：zhidao.baidu.com/search?ct=17&pn=0&tn=ikaslist&rn=10&word=helloworld&ie=utf-8&fr=wwwt
实际显示：//asdfa.cn/ak78ss
前面asdfa.cn是对应域名zhidao.baidu.com，后面长长的字符串被压缩成ak78ss。
【问题】
现在让你来设计TinyURL的实现，以下问题要怎么设计：
(1)：域名后面的编码如何实现?
(2)：对于已经映射过的一个URL，怎么查找已存在的TinyUrl?
(3)：有10亿个url，一个服务上存不下，需要多台服务器，怎么设计实现
(4)：让你来设计这样一个服务，最大的问题是什么?
【探讨】
暂时的思路(根据网友的回答汇总)，欢迎探讨：
(1)：两种思路，一种是把真实网址存数据库，然后取自增id，做个哈希或者进制转换之类的，生成短网址，用的时候查一下就行了；另一种是直接真实网址哈希然后截取特定位数，6位的话 （26+26+10）^6 种组合，应该够用了，实在不行再做一次碰撞检测
(2)：直接key-value存储查询
(3)：二次哈希， 根据ak78ss这样的值映射到不同机器上，hash或者字母序层次下去
(4)：查询速度？响应时间？还有过期的url在浪费存储空间？

# 讨论

邹博 - 学而时习之

赞同来自: 刚背曾 、升

@Guangzhan @wsk

突然觉得，如果避免冲突，用BloomFilter是否就够了？

如果按照理论最优的2倍的槽数目，存储10亿条URL，只需要10亿*2 / 8 bit = 250M就够了。

2 0 2015-02-03

xiaoxiong

出了hash之外，我 猜一下吧， 是不是 用计算机之内的0-1压缩编码实现的？比如00011111，可以写成3051，代表3个0 ， 5个1，然后按周某种内定的次序进行排序，再按照某种译码方法进行译码呢？

0 0 2015-02-03

邹博 - 学而时习之

先提两个问题：

1、TinyURL中的“asdfa.cn” ，是否只有经过sina的域名服务器，才能解析？其他服务器应该是办不到的吧？

2、将所有URL存储的办法，理论上应该是可行的。但总觉得太“工业界”了。

如果问题1中，只有sina的服务器才能解析tinyURL，那么，它应该是做了些Hash、MD5或者其他映射。

# 讨论

Guangzhan - 让坚持成为一种习惯，让放弃成为一张奢侈

（网上找到的资料）：

① 将长网址用md5算法生成32位签名串，分为4段，，每段8个字符；

② 对这4段循环处理，取每段的8个字符，将他看成16进制字符串与0x3fffffff(30位1)的位与操作，超过30位的忽略处理；

③ 将每段得到的这30位又分成6段，每5位的数字作为字母表的索引取得特定字符，依次进行获得6位字符串；

④ 这样一个md5字符串可以获得4个6位串，取里面的任意一个就可作为这个长url的短url地址。

很简单的理论，我们并不一定说得到的URL是唯一的，但是我们能够取出4组URL，这样几乎不会出现太大的重复。

0 1 2015-02-03

wsk - Idiots. Idiots never change.

1000,000,000个URL,大概是TB级别的存储数据,

我的理解是

要写出映射函数f,使得f(URL)唯一,直接维护数个递变ID应该是最简单的做法(其他方法生成的不一定唯一,算法还挺复杂)

如果我们把字母(区分大小写)和数字,都作为有效的字符(62个),最短用5位就可以包容9亿个URL,

考虑到并发情况,我们不希望用锁来解决一致性问题,可以放宽短地址到6-7位,这样可以预先为几台分配器预留地址(000000~0ZZZZZZ,100000~1ZZZZZ,...),查询起来也方便

# 我们在这里

- 七月算法 http://www.julyedu.com/
  - 视频/课程/社区
- 七月题库APP：Android/iOS
  - http://www.julyapp.com/
- 微博
  - @研究者July
  - @七月题库
  - @邹博_机器学习
- 微信公众号
  - julyedu

# 感谢大家

# 恳请大家批评指正！