Hash 在不同的应用场景

摘要：

散列表（hash）由于其特性而活跃在一些和（1）编码（2）查找（3）聚类沾边的领域，从最基础的程序设计中会应用到这个数据结构以及用以构造的其他数据结构，数据库中用在索引来提高各种操作性能；网页爬虫为提高网页质量和防止环路必要做的重复文档检测，降维等一些应用或功能；文件系统中被用在命名空间，目录查找等。通过对这样一基础数据结构的改造，用以适应迥异的领域和应用，这之间是否有更简单通用的方式来解释其内在机制。恩，好像就是基础数据结构的改造来适应具体的应用，而且改造样式也大体相似。在文章的最后，给出文章涉及到的几种 hash 方法的一个统一视角。通过从更基础简单的角度来看待这个工具，能够让我们更容易的从 hash 的几个改造点尝试看是否能够满足你面临的问题需要。本文同样也对 LSH 和 DHT 做一个简单综述。主要贡献：提供看待 Hash 工具的不同视角，给出综述，并用简单的思维来理解，通用的概念来解释内在机制。

1 介绍

Hash 出现在加密，聚簇，相似性检测，缓存的分布式策略等应用中，本文主要就应用中涉及的：（1）局部敏感 Hash（locality sensitive hash/LSH）（2）一致性 Hash（consistent Hash）两大类 Hash 做简单综述，在这些不同的应用中，用集合的视角来给这些 Hash 工具一个通用的简单的理解方式。Hash 定义如下：

定义：将数据元素的关键字 K 作为自变量，通过一定的函数关系（称为哈希函数），计算出的值，即为该元素的存储地址。表示为：

Addr = H（key）

想有一种更广义的理解，那散列（hash）是一种编码方式（别名），这种编码方式有一个对应函数，因此查询效率高，但这个函数存在不同的关键字映射到同一个值的特点；因此在应用中对应 Hash 的这些特点，个人理解可以分成三大类相关：（1）编码相关：特征提取等压缩相关或者加密相关；（2）查询相关：数据库索引，文件目录，数据定位；（3）聚簇相关：相似性检测，重复数据删除，最近邻，聚簇等。

散列（Hash）	相关技术：稠密索引；稀疏索引；局部敏感 Hash（LSH）；SimHash；I-match 等；布隆过滤器；分布式 Hash 表（DHT）；一致性 Hash（consistent Hash） 应用：相似性搜索，近邻计算，范围搜索，聚类，降维，等用途的 LSH<Locality Sensitive Hashing 位置敏感的哈希>算法

2 局部敏感 Hash

直观感性的看 LSH 参看下图：

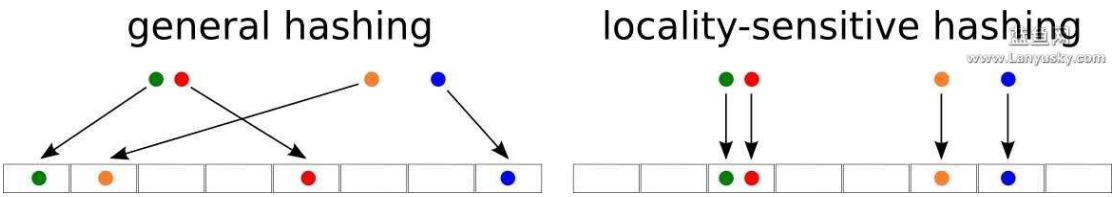

图 1

定义：Similarity function *sim(x,y)*，Family of hash functions *F* with probability distribution such that

$$\mathbf{Pr}_{h\in\mathcal{F}}[h(x)=h(y)]=sim(x,y),$$

由 LSH 的定义可见，LSH 家族是和 sim 函数相对应的。在网页去重，文档相似检测等可查找关键字 Shingling 算法，I-Match 算法，SimHash 算法等。下面给出 SimHash 在进行网页去重的例子，来理解 LSH。但并不是所有的相似度量都存在 LSH。具体证明和相关 Lemma 参考[sim]中第二节。

1.1 SimHash:

给出一个 LSH 的实例，simHash 的步骤如下：

1，将一个 f 维的向量 V 初始化为 0；f 位的二进制数 S 初始化为 0； 2，对每一个特征：用传统的 hash 算法对该特征产生一个 f 位的签名 b。对 i=1 到 f： 如果 b 的第 i 位为 1，则 V 的第 i 个元素加上该特征的权重； 否则，V 的第 i 个元素减去该特征的权重。

3，如果 V 的第 i 个元素大于 0，则 S 的第 i 位为 1，否则为 0；

4，输出 S 作为签名。

再对照图看一下，在这个例子里有引入权重 w，基本思想按照不同的特征来提取出信息，并将信息进行压缩整合还原，再根据还原之后的进行判断相似性。用了 Hash 本身的转码，压缩的属性，也应用了 LSH 的多个 Hash 来提高相似概率的特性，这么做的理论支持参考<>,例子的具体步骤网上有很详尽的介绍，不赘述。

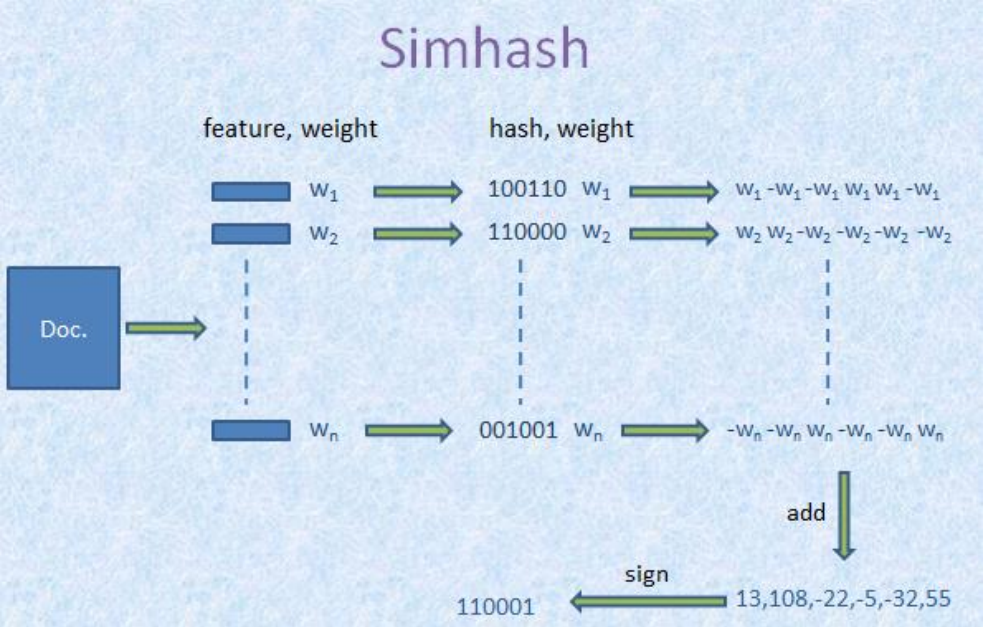

图 2

1.2 Bloom Filter

Bloom filter 是由 Howard Bloom 在 1970 年提出的二进制向量数据结构，它具有很好的空间和时间效率，被用来检测一个元素是不是集合中的一个成员，这种检测只会对在集合内的数据错判，而不会对不是集合内的数据进行错判，这样每个检测请求返回有“在集合内（可能错误）”和“不在集合内（绝对不在集合内）”两种情况，可见 Bloom filter 是牺牲了正确率换取时间和空间。

步骤：

1， 开始时集合内没有元素

2， 来了一个元素 a，进行判断，这里哈希函数有两个，计算出对应的比特位上为 0，即是 a 不在集合内，将 a 添加进去：

3， 判断是不是在集合内，也是同 a 一样的方法，只有对元素哈希后对应位置上都是 1 才认为这个元素在集合内（虽然这样可能会误判）：

4， 随着元素的插入，Bloom filter 中修改的值变多，出现误判的几率也随之变

大，当新来一个元素时，满足其在集合内的条件，即所有对应位都是 1，这样就可能有两种情况，一是这个元素就在集合内，没有发生误判；还有一种情况就是发生误判，出现了哈希碰撞，这个元素本不在集合内。

结合图来看一下，如（图 3）所示，为两个 Hash 函数的 Bloom Filter，在最后一步判断是否在集合内的时候，如果两个 H 对应位置都为 1，则在集合内的可能性就很大了。

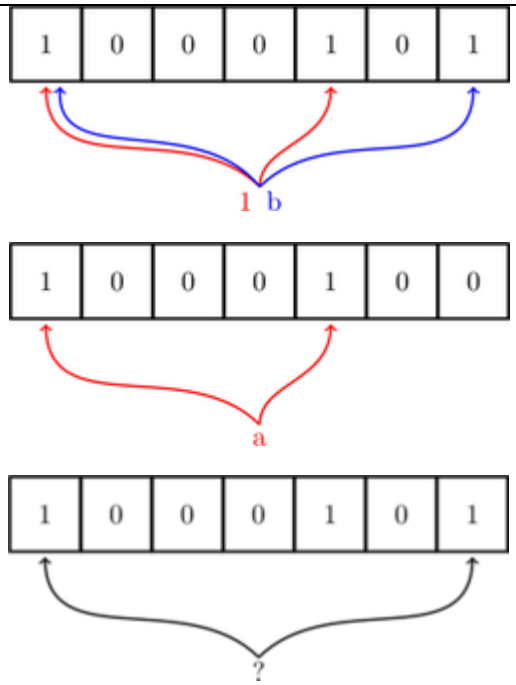

图 3

总结：LSH 的大概思想很简单，对数据进行有损压缩，如果在不同的压缩函数下距离都很近，那么原数据就有很大概率相似。几何意义上来讲，就是在高维超平面空间中，如果两个点在一个向量上的投影是一样的，并不能说明这个点对的距离近。为了让 Hash 更靠谱，就多选了几个随机的向量，如果这个点对能够在不同的向量上投影都相近，则原点对相近的可能性就更大。对应到 SimHash 中的汉明码的话，对应到{0,1}取决于原始点对和选择的随机向量的夹角的 cos 值的正负。BloomFilter 通过一个 Hash 方程来判断有一定的误码，多个来判断能减少误判率。

3 一致性 Hash

在分布式存储中数据在节点上的分布方式。一种简单的方式可以通过对数据 data 进行哈希分布，计算出一个 hash 值，再对节点数取模。但这样的简单方式在分布式系统中存在着一个很大的问题。分布式环境中，节点 Down 掉，或加入新节点扩展原系统是非常正常的。例子-1 给出这样 Hash 的问题：

例子-3.1：

三个节点的时候：

N1:d0,d3,d6,d9 N2:d1,d4,d7,d10 N3:d2,d5,d8	N1:d0,d4,d8 N2:d1,d5,d9 N3:d2,d6,d10

	N4:d3,d7

添加一个节点造成了全部的重组。对应到数据存储中也就是需要整个数据在节点之间的迁移。 Hash 函数的衡量指标之一：单调性(Monotonicity).而一致性 Hash 就是解决这个问题存在的。

概念：Consistent hashing is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots。

例子-3.2：

直接看图，左边为一致性 Hash 的设计，右边为添加节点的情况。途中可以看出，只是将原来的介于添加节点的后继节点上的部分数据给迁移过来，没有整个大规模的数据迁移。

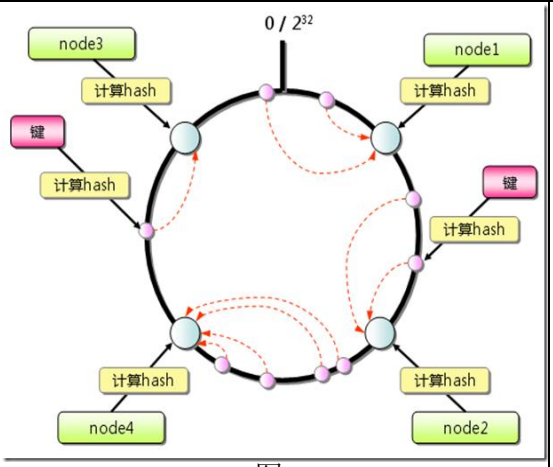

图 4-1

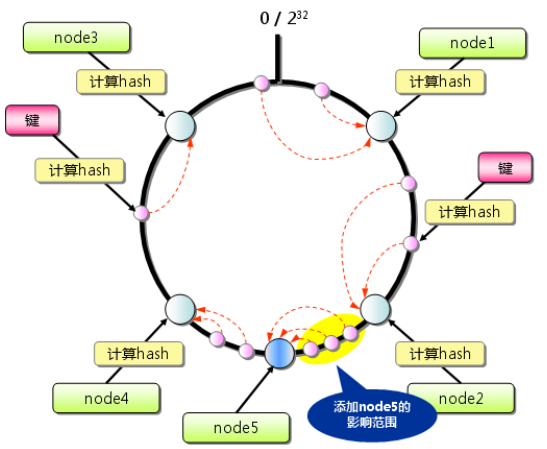

图 4-2

4 对比和展望

在这一小节的第一部分，给出如何用集合的角度来看 Hash。

4.1 集合的角度

集合的角度不是我提的，算法导论里有说，因为觉得这个角度可以很方便的统一来说别的几个变种。

参看图 5-3 是对 Hash 的表述，集合 K 是要映射的 key 的动态集合，之所以说 K 是动态集合，因为在应用中可能会对 K 里的内容有增删改查的操作，但是不管怎么增删改查，这里的 Key 都取自于全集 U，映射后到集合 B（集合 B 就是图 5-3 中的右边椭圆中的那个集合）。给一个 key,知道 B 中的对应就能够进行寻址，或者别的一些操作和应用了。

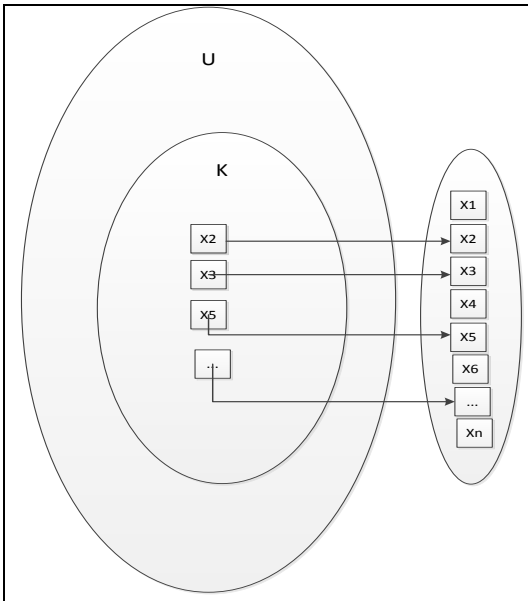

图 5-1：数组(位图)

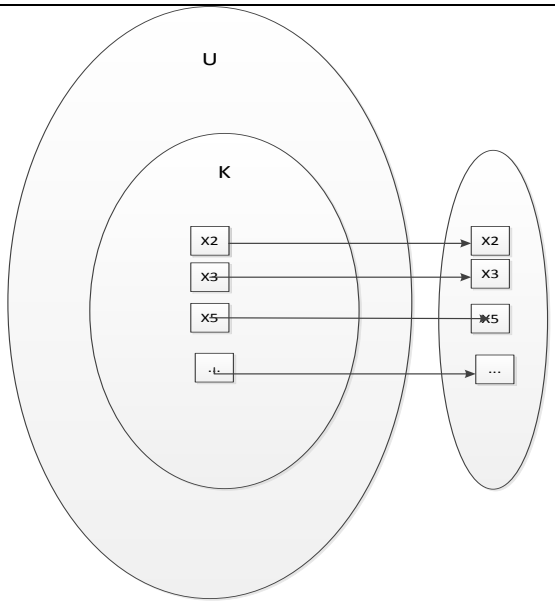

图 5-2：直接寻址表

如图（5-3）中的一般化 Hash，在固定好集合 B 之后，有一个固定的散列函数H，不管集合K怎么动态变化，增加减少，都不影响 B 的大小。散列函数 H 是一个从集合 K 到集合 B 的一个映射，因为 B 的集合不因 K 集合的变化而变化，函数 H 就不可能是一个单射函数。也就存在着多个自变量对应 B 中同一个值的情况。也就使得查询会有冲突的可能（例如 Bloom Filter 存在结婚判断错误可能），反过来看也利用这个特性，被拿来做聚类，相似检测。

如图（5-2）直接寻址表，可以看做是特殊的 Hash，也就是当散列函数 H 为单射函数的情况，为了保证一一映射，集合 B 也就会随着 K 集合的变化而变化。直接寻址表也是一种常见的索引方式。

如图（5-1）数组，这个名称可能不太合适，所以又写了个位图，但位图又暗示了这个信息是 0,1 值，嗯，实际上应该也算是直接寻址表，不过存储上有不同。差不多就是这个意思。图（1）和图（2）的差别在于，图（1）会在 K 集合没有增长这么多的时候，就预先把 B 集合扩充在和全集 U 一样的大小。这种策略也常碰到，位图就是。比起（2）来讲，有优点也有缺点，优点就是，对于（2）的查找操作是常量级的，但是如果是中间的插入，删除等操作就会有性能上的不足，当然应该也可以通过别的一些方式，使用些小技巧不删除，只追加数据（譬如：Dynamo 在存取数据的策略好像也这个思想，只增加数据，不删除数据，减少维护成本），而这里的话，那他的各种增删改查的操作都是常量级的。缺点就显而易见了，就是空间效率不高，要构造和全集 U 对应大小的 B 集合。这就使得使得其保持优秀的代价是这个先天缺陷往往使得在一些情况下的使用是不现实的或者是不高效的，譬如集合U不是有限集合的时候，K相对于U是很稀疏的时候等。 各有好坏，根据应用选择您要的 Hash 工具。

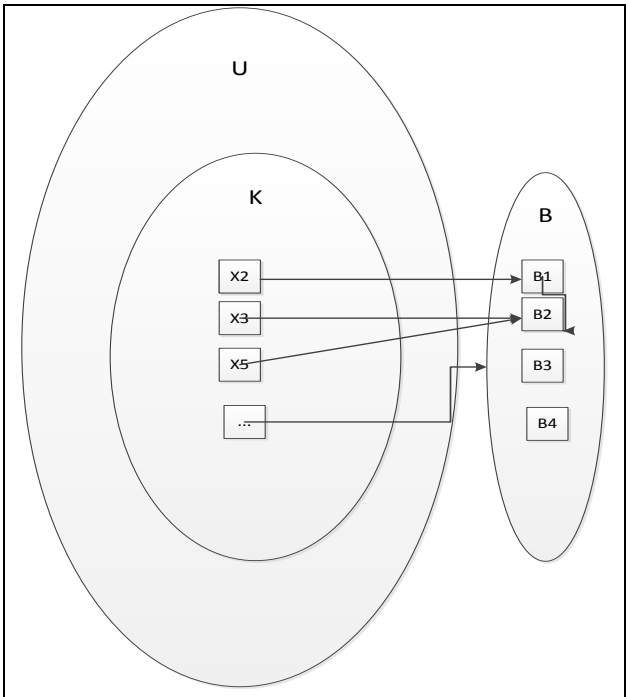

图 5-3：一般化 Hash

4.2 集合的角度：一致性 Hash 和 LSH

有了前面铺垫的知识，觉得图 4 和图 5 的出现会使得 LSH 和一致性 Hash 变得好理解很多。

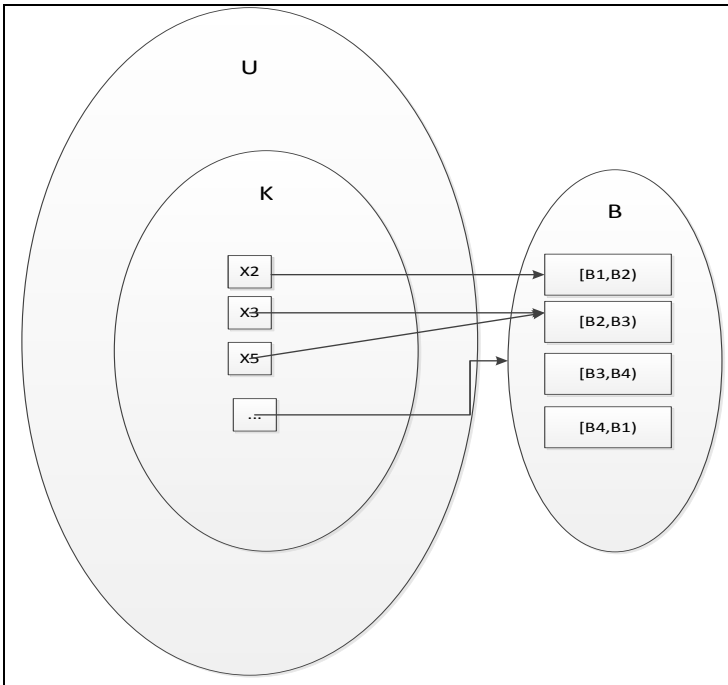

图 5-4：一致性 Hash

图（4），对比图（3）中的一般性 Hash，他们的区别就是，图（3）中的集合 B，其中的每个元素就是一个确定的值，而在图（4）中，将这些离散的值元素替换成了连续的一个区间元素，这里的散列函数 H 可能还是原来的那个，当然也可以多变几个来对应（这个具体没找到应用，但肯定有或者会有~）。从（3）到（4）的变化，也算是个一般化的过程，对于（3）的一般化 Hash，就像是有这样一个法律规则 H，根据规则你如果对应到刑法，死；对应到民法，活。这样就使得，如果法律规则改成了 H'，那一部分人可能从活对应到了死，还有一部分相反，那就会变得很悲催，一念只差 H 的规则很容易将你在刑法还是民法之间切换，但是所受的罪的差别没个缓冲，就会特别的扛不住了。而改进成（4）之后就像是，你们按照 H 规则，都归到了刑法里，这里并不是只有一个惩治方案，而是会有什么一到三年，还是三年以上五年以下等。这样子算不算提高了容错性~换个规则，或是不同的规则映射都不会使得最后的处理太敏感。图（4）可以对应到一致性 Hash 中。另外，区分下动态 Hash，虽然一致性 Hash 解决了动态加入节点删除节点对整体影响不大的问题，但和动态 Hash 中为了适应 K 集合的增减，通过对集合 B 中元素分裂归并来让 B 集合变成动态的，和这里的一致性还是有差的。

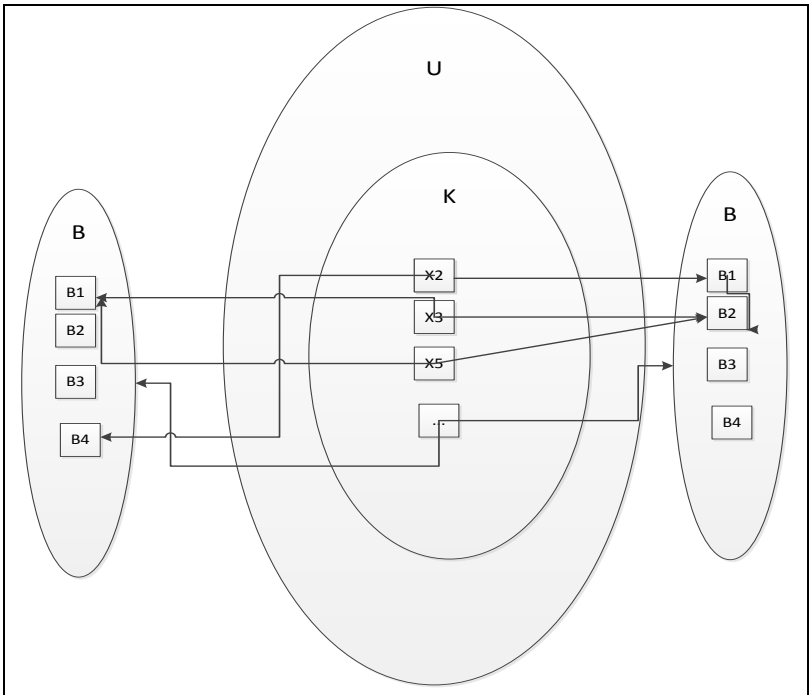

图 5-5：局部敏感性 Hash

图（5），对比图（3）中的一般性 Hash，他们的区别是这里明确有多个 Hash 函数，是一个 Hash 函数组。应用在相似性检测中，还有判断集合元素归属中，多一些 Hash 函数，帮助提高命中率。这个可以类比为在取证的时候，我们要多方取证，才能够帮助在定罪的时候支持你的论点。多种取证方式就像是多个 Hash 函数。

4.3 展望

一方面因为不是对某个问题进行 Survey,我的知识面也比较有限，所以可能覆盖知识面不全，希望后续再看到再继续补充；其次散列表大部分时候可能要用到概率的知识，这里没有从概率给出分析。

参考：

detecting Near-Duplicates for web crawling//simhash 应用在网页去重上

similarity estimation techniques from rounding algorithms //simHash 原理部分

Dynamo: Amazon's Highly Available Key-value Store //一致性 hash 应用

《这就是搜索引擎》//第 10 章，关于 simHash, I-Match 的最容易了解的笔记。

《数据库系统概念》//第 12 章，索引与散列

《Mining of massive datasets》//第三章。对应有中文书籍。觉得和<这就是搜索引擎>有重合部分

http://en.wikipedia.org/wiki/Locality-sensitive_hashing // LSH 的 wiki 上的解释

http://en.wikipedia.org/wiki/Distributed_hash_table // DHT 分布式 hashtable 的

http://en.wikipedia.org/wiki/Bloomfilter //bloomFilter 的 wiki 解释

http://en.wikipedia.org/wiki/Consistent_hash //一致性 hash

ttp://www.cnblogs.com/linecong/archive/2010/08/28/simhash.html //simHash

http://www.jiacheo.org/blog/174 一致性 Hash