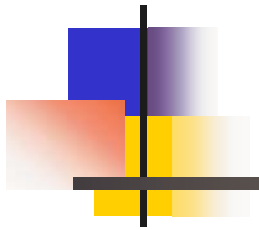
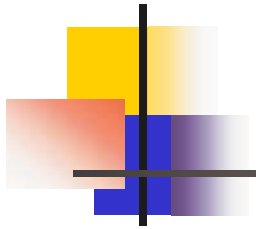
物联网数据存储及管理



物联网元数据管理机制

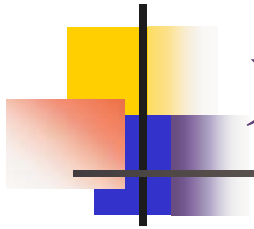
华宇

https://csyhua.github.io/



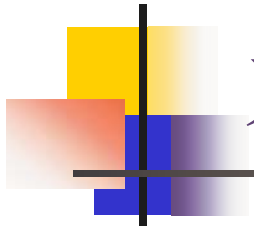
目录

- 元数据管理概述
- 对象存储系统元数据
- 元数据管理方式



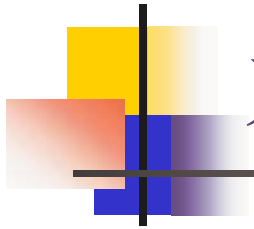
元数据管理概述

- 在存储系统中，元数据是用于描述文件系统组织结构的数据
- 在访问文件数据之前必须先访问其元数据以获取要访问数据的描述信息和空间组织信息
- 然后才能根据这些信息访问到数据。



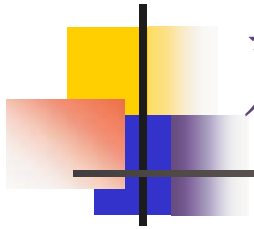
元数据管理概述

- “元数据”最本质的定义为：用于描述数据的数据(data about data)，它广泛应用于许多应用与研究领域
- 在存储系统中，元数据用来描述文件的大小、存储位置、访问权限等。
- 相对于文件数据本身，元数据体积相对小，但是对其的访问却比文件本身数据更多，特别是分布式文件系统中，文件系统中大约50%—80%操作是对元数据进行的



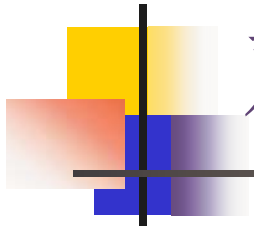
元数据管理概述

- 元数据的访问性能将在很大程度上决定系统的整体性能。
- 因此，元数据的管理在很多应用环境中成了影响系统整体性能的关键，成为了存储领域内的研究热点



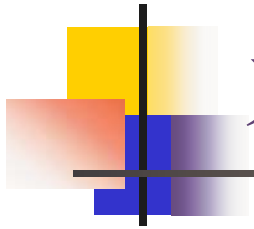
对象存储系统元数据

- 在OBS系统中，元数据分为物理视图（约占元数据总量90%）和逻辑视图（约占元数据总量10%）。
- 物理视图负责将对象的逻辑块映射到存储物理介质，以支持对象在块存储设备上的工作，存放在OSD上；
- 逻辑视图用于描述整个系统的命名空间、目录层次结构和访问控制策略，将文件映射为存储对象，存放在MDS上。



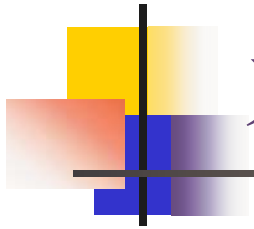
对象存储系统元数据

- 存储在OSD中的元数据负责OSD上局部的数据管理
- 而MDS上维护管理的元数据则负责整个存储系统中全局的数据布局、负载调度与访问控制
- 是OBS元数据管理的重点



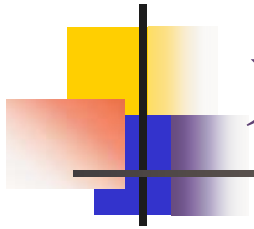
元数据表

- 用来描述文件与对象之间的对应关系。
- 一个较大的文件可以被划分为多个对象，对个较小的文件也可以聚合成一个对象存放，
- 通过表中的flag字段表示



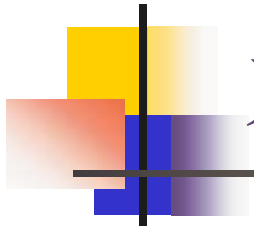
元数据表

- flag为0表示一个文件对应多个对象，索引号字段表示各对象在文件中的物理逻辑顺序，偏移值字段表示该对象在对应文件中的偏移首地址；
- flag为1则表示多个文件存在一个对象中，索引号字段表示各文件在对象中的物理逻辑顺序，偏移值字段表示各文件在中该对象的偏移首地址。



元数据表

- OSD号是各存储设备在OBS存储系统中的全局唯一标识
- 对象号则是根据一定的命名规则形成的对象标识符
- 通常是一个全局唯一的128位无符号数



元数据服务器的基本功能

- 维护全局树形目录结构
- 提供给客户统一的目录视图，
- 完成客户端提交的文件请求到对象的映射，
- 并给出所属对象的存储位置

元数据服务器的基本功能

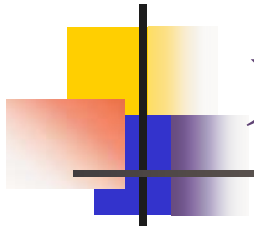
- 元数据的本地存储管理：将MDS上的元数据以一定的组织方法存放在数据库或文件系统中，能够有效的对元数据进行访问与管理；
- 安全策略：对客户的文件访问请求进行安全检测，对客户提交的操作认证与授权，客户凭MDS发出的授权证书才能够正确访问OSD；

元数据服务器的基本功能

- 对象分布管理：将文件根据大小分割或聚合成对象，并按照一定的分布策略将对象分散到不同OSD上，并能够动态的调整对象分布方式，使各OSD访问负载均衡；
- Cache的一致性维护：在OBS系统中，客户端、MDS、OSD上面都有Cache，如何维持各设备上Cache的一致性，也是MDS的重要责任

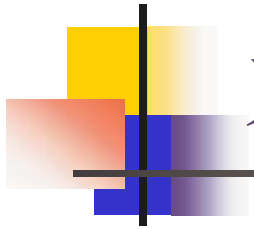
元数据服务器的基本功能

- 在MDS集群模式下，整个存储系统的逻辑视图元数据按照一定的分配策略分布到集群各个节点上，
- 各MDS节点并行工作，提供了强大的聚合处理能力，使MDS机群能够有效面对大量客户的并发访问。
- 分配策略要能够支持MDS集群的动态调整，当集群结构发生变化或全局逻辑视图改变时，能够动态调动元数据，从而显著增强了MDS集群的可扩展性



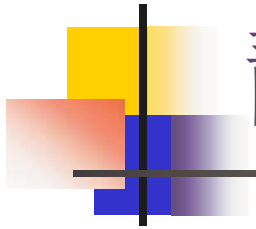
元数据管理方式

- MDS集群环境下，元数据管理的核心问题就是元数据在集群上的分配策略。
- 合理的分配策略可以使元数据在各MDS上均匀分布，
- 来自客户端的访问负载也可以均衡的分布在各个MDS上，
- 在整个MDS集群中尽量分散热点元数据，



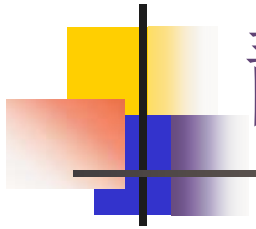
元数据管理方式

- MDS集群处理能力能够随节点数量的增加而扩展。
- 而设计不合理的分配策略可能造成MDS集群负载不均衡，降低了元数据服务效率，严重影响了系统的整体性能与可扩展性。
- 总体来说，元数据的分配方式分为静态方式与动态方式，以及结合新颖数据结构在前两种方式上的改进



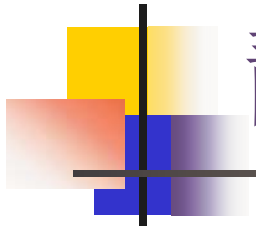
静态目录子树分割

- 按照传统的文件组织方式，将全局目录树划分为若干子树，
- 每个MDS节点上面维护一个或多个子树，
- 各节点共同组成全局的逻辑视图。
- 这种元数据分配策略管理简单，被一些早期经典分布式文件系统采用



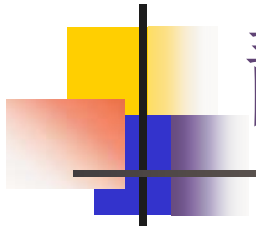
静态目录子树分割

- 策略静态目录子树分割策略最大程度的保留了目录的层次逻辑结构，
- 便于利用文件访问的空间局部性，通过在客户端上设置Cache可以减少与MDS的通信次数；
- 用户的文件访问请求只需涉及一台MDS，各个MDS彼此独立，无须相互通信，简化了MDS集群的设计。



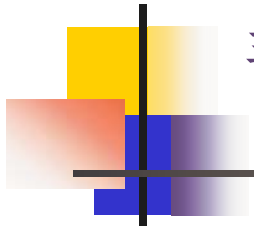
静态目录子树分割

- 但是，这种策略也存在明显的缺陷：若某个目录子树下面的自目录或文件成为访问热点，则存放该目录子树的MDS节点将负载过重，成为MDS集群中的瓶颈。
- 另外，当加入或删除MDS节点时，需要人工的调整目录子树的分配，大大限制了MDS集群的可扩展性



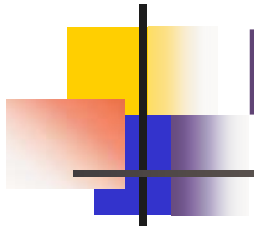
静态哈希

- 静态哈希方法通过将文件标识符、目录/文件名或其他文件特征值进行哈希运算，映射分配到不同的MDS节点上
- 静态哈希策略打破了目录层次结构，将全局元数据均匀的分配到MDS集群各节点上，不会因为存在热点目录导致MDS负载不均衡；
- 客户的文件访问请求，只要经过简单的哈希运算便可以定位到目标MDS



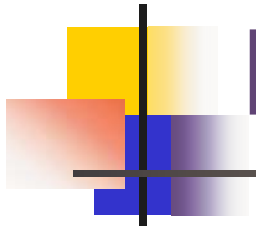
静态哈希

- 但是同样也存在着缺陷：由于放弃了系统目录结构的空间局部性，在对子目录进行遍历操作，要涉及MDS中大量节点的工作，
- 对子目录进行重命名操作将导致大量元数据迁移；客户端上的Cache也因为不断刷新而降低了命中率。
- 集群中加入或删除MDS节点时，必然需要更改哈希函数，将导致MDS集群中元数据的重新分配，大量元数据迁移，
- 因此其可扩展性能很有限



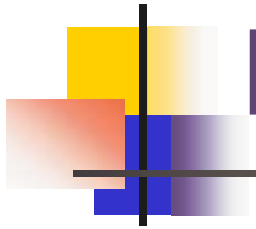
LH（Lazy Hybrid）

- 设计结合了静态目录子树分割与静态哈希策略的优点。
- LH策略采用文件的全路径名进行哈希运算，这样相同路径下的所有文件被映射存放到同一MDS上，这些文件的元数据进一步通过哈希分散开来，并使用一个元数据查找表（Metadata Look-up Table，MLT）来记录分布信息。



LH（Lazy Hybrid）

- 这样的设计设计方式既保留了一定的目录结构，又充分分散了热点目录下的元数据。
- 采用了一种访问控制链表描述文件的安全权限，将目录权限的检查合结合文件的元数据中，
- 不必遍历文件路径的所有目录层次才能确定该文件的访问权限。



LH（Lazy Hybrid）

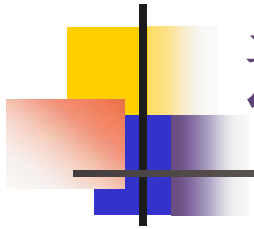
- 将元数据的更新操作暂存延迟到该元数据再次访问的时候进行，分散MDS的更新负担。
- 缺点与静态哈希策略类似，文件系统中目录被重命名时，
- 需要将改目录下的元数据重新分配，导致大量元数据迁移，延迟的更新方式也加重了这一状况

动态目录子树分割

- 将全局目录划分为若干目录子树，分布存储在MDS集群各节点上，
- 当系统中出现热点子目录或目录子树过于庞大时，则采用哈希的方法将该子目录下的所有元数据分配到集群其他MDS节点上。
- 这样的设计可以最大程度保留目录结构，以便充分利用文件的空间局部性，

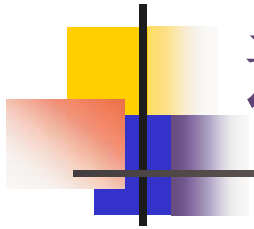
动态目录子树分割

- 当有目录成为访问热点导致有相应MDS负载过大时，
- 动态的进行元数据迁移，调整集群负载。
- 但是在文件访问负载不断变化的情况下，如何能使系统根据负载情况自适应调整元数据分布，是一个实现难题



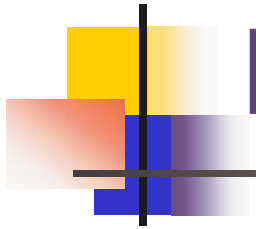
动态哈希

- 提供了相对负载均衡、弹性更新策略和全生命周期管理三种策略，
- 通过MDS之间的检测，将高负载MDS中的部分元数据向低负载MDS迁移



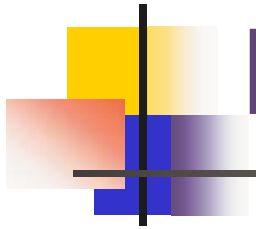
动态哈希

- 当需添加或移出MDS时:
- 采用弹性更新策略进行负载调度
- 全生命周期管理利用缓存发现热点并制作副本
- 分担元数据访问负载



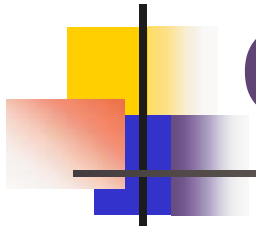
HBA

- 设计思想是在每个MDS上建立两级布隆过滤器阵列（Bloom-filter arrays），
- 第一级是LRU Bloom-filter，用来记录本地MDS访问最频繁的文件，体积小但查找快；
- 第二级Bloom-filter用来记录本地MDS存储的所有文件，体积大但查找慢。
- 同时，每台MDS还保存了集群中其他MDS节点的两级Bloom-filter，构成了两级记录全局文件分布信息的Bloom-filter阵列。



HBA

- 在客户端提交访问请求后，首先在第一级Bloom-filter阵列中查找，命中则直接将请求重定向到对应MDS上，
- 否则查找第二级Bloom-filter阵列，命中则直接将请求重定向到对应MDS上，否则进一步将客户请求在MDS集群中广播。
- HBA策略不关心元数据的具体分配方式，而是通过将客户请求进行两级Bloom-filter阵列查找快速定向到存储该请求文件元数据的MDS上，提高了访问速度，
- 但不命中带在的请求广播会增加系统额外负担



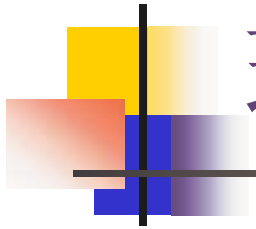
G-HBA

- 中引入了分组的概念，将MDS集群根据物理位置等因素分成若干小组，
- 每个小组维持着一个全局的两级Bloom-filter阵列，分布的存储在各MDS节点上。
- 客户请求到来时，MDS首先查询本地的部分Bloom-filter阵列，命中则直接将请求重定向，否则首先在组内组播查询，若再不命中则在组外广播。
- 这样的设计显著减少了HBA系统中客户请求广播带来的网络负载

现有元数据管理方式对比

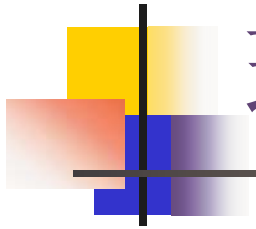
- 静态目录子树分割与静态哈希是最基本的元数据分配策略，
- 管理简单而易于实施，但在目录结构或名称发生变化时会导致大量元数据迁移，
- MDS集群节点的加入与删除也会导致元数据的重新分配，因此可扩展性较差。

现有元数据管理方式对比

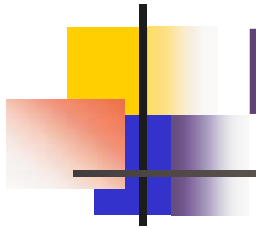


- HBA与G-HBA则为元数据管理发展了一种新的设计思路，
- 既在原有元数据分配策略的基础上建立一个逻辑层，充分利用客户访问的时间局部性，
- 结合Bloom-filter结构的高速查询，使客户请求能够快速定位，
- 显著的分担了MDS的工作负载，提高了MDS集群的整体性能

现有元数据管理方式对比



- LH策略、动态目录子树分割与动态哈希在前两种基本策略基础上进行了改进，
- 通过使用延迟更新策略、元数据查找表、负载动态调整等方法，
- 使元数据管理获得更有效的管理，
- 但是也不可避免的继承了静态目录子树分割与静态哈希的缺点。

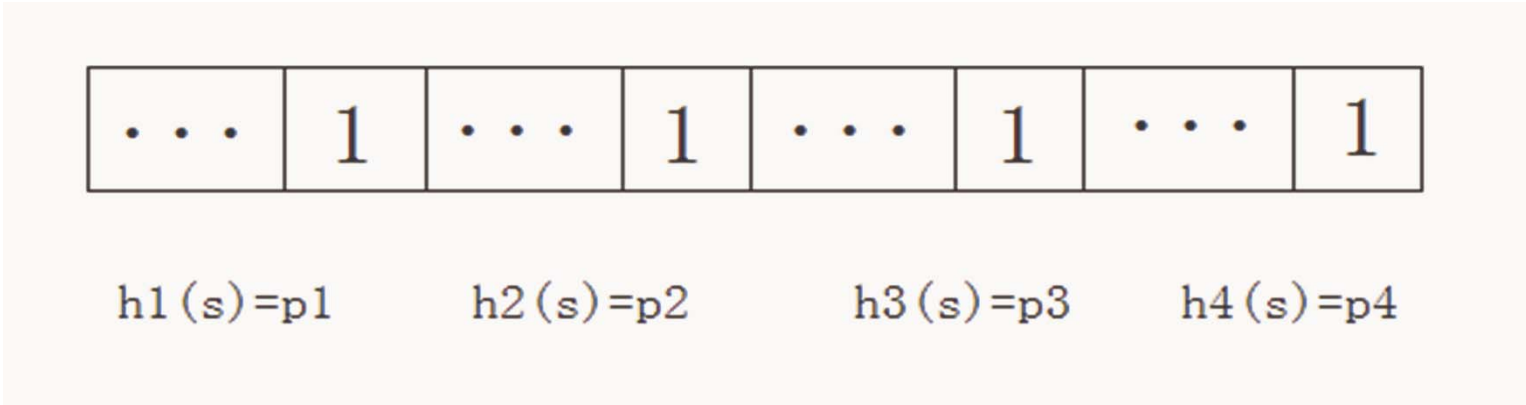


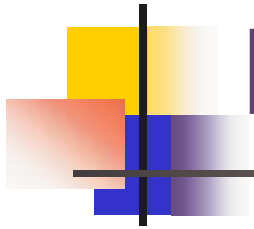
Bloom filter

- Bloom Filter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。Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。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通过极少的错误换取了存储空间的极大节省。

Bloom filter

- 由一个很长的二进制向量数组和一系列随机映射函数组成，
- 它只需要哈希表 1/8 到 1/4 的大小就能解决同样规模的集合的查询问题





Bloom filter

- Bloom filter的本质是哈希计算，不同之处在于Bloom filter对同一数据使用多个哈希函数进行多次哈希，将结果保存在同一个向量数组中，
- 所以Bloom filter在达到相同的功能的情况下比原始的哈希结构更节约存储空间。
- Bloom filter算法的一个缺点在于查询一个元素是否在集合S上可能存在失误定位(False Positive)

Bloom filter

- 集合表示和元素查询
- 下面我们具体来看Bloom Filter是如何用位数组表示集合的。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。

理论分析

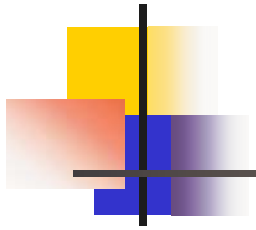
$$\left(1-\frac{1}{m}\right)^{kn} \approx \mathrm{e}^{-kn/m}.$$

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

$$g = k\ln(1-\mathrm{e}^{-kn/m})$$

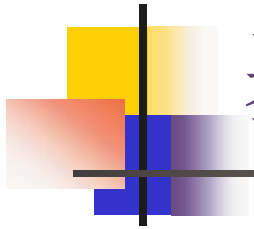
$$\frac{dg}{dk} = \ln\left(1-\mathrm{e}^{-\frac{kn}{m}}\right) + \frac{kn}{m}\frac{\mathrm{e}^{-\frac{kn}{m}}}{1-\mathrm{e}^{-\frac{kn}{m}}}.$$

$$f \text{ is } (1/2)^{k} = (0.6185)^{m/n}.$$



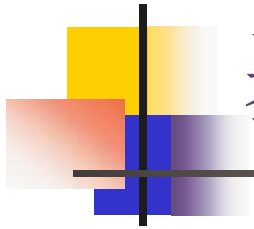
Bloom filter

- 为了表达S={x1, x2,…,xn}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 Function），它们分别将集合中的每个元素映射到{1,…,m}的范围中。对任意一个元素x，第i个哈希函数映射的位置hi(x)就会被置为1（1≤i≤k）。
- 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。



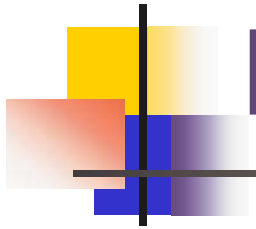
基于Bloom filter的数据索引

- 表示和查找信息是大多数计算机应用程序的核心，这两个过程是密切相关的。
- 比如，采用顺序表表示信息，则可以使用折半查找；
- 采用哈希表组织信息，则可以使用哈希查找。
- 表示就是以计算机能处理的方式，把信息组织成为可计算的实体，查找就是确定一个具有特定值的元素是不是一个特定集合的成员



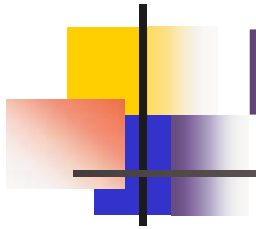
基于Bloom filter的数据索引

- Bloom filter对数据集合采用一个位串表示并能有效支持集合元素的哈希查找操作。
- 由于其表示算法的随机特性，存在某些不属于数据集合的元素被判断为是数据集合的元素的可能性，其大小记为误警率。
- 只要这种可能性足够的小以至于应用能容忍这种误差，由于其哈希查找的常数时间和少量的存储空间开销，使得它具有非常的实用价值



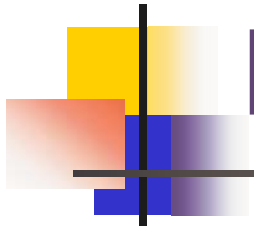
Bloom filter的应用

- 字典存储
- 由于许多单词应用一些简单的规则通过合适的连词符号得到，而大概10%的单词需要通过表格查找。
- 为避免存储那些应用简单规则通过连词符号得到的单词，Bloom提出使用Bloom filter保存那些需要查询的单词



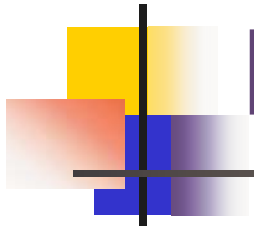
Bloom filter的应用

- 数据库
- Bloom filters很早被应用于数据库中
- 一种应用是加快semi-join操作



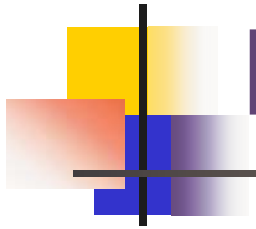
Bloom filter的应用

- 分布式缓存
- Bloom filter在分布式的协议中的应用也有很大潜力,已经用于Web Cache共享。
- 在代理缓存系统中,为了减少信息传输,可以用Bloom filter表示cache目录,在共享的服务器之间不是传输URL列表对应的真正的cache目录,而只是定期地传送Bloom filter。
- 用Bloom算法表示和定位代理缓存系统的任何一个网站的网页,和用URL来表示和定位网页相比,节省了大量的存储空间,并提高检索速度



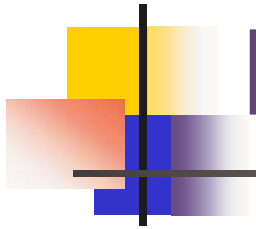
Bloom filter的应用

- P2P/Overlay网络应用
- 在一个几百结点的中等规模的比较稳定的peer-to-peer系统中，在资源定位方面Bloom filter比分布式哈希表更有优势。
- 保存P2P系统中所有其它结点的资源列表是不可行的，但保存所有其它结点的Bloom filter是可行的。
- 例如，Bloom filter可以使用8或16bits来表示一个资源来代替64bits。误判率会对不拥有请求资源的结点产生额外的请求



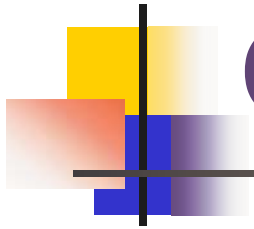
Bloom filter的演变形式

- 由于Bloom filter在表示集合方面的空间高效性，
- 并且还支持成员关系的查询，所以Bloom filter得到广泛的应用，对Bloom filter的研究也从未间断。
- 新的Bloom filter结构以应用在不同的环境中



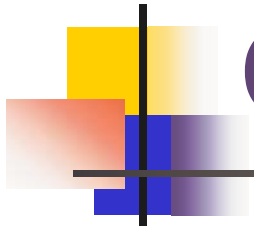
Bloom filter的演变形式

- Counting Bloom Filter，
- Compressed Bloom Filter，
- Hierarchical Bloom Filter，
- Space-code Bloom Filter，
- Spectral Bloom Filter
- Parallel Bloom filters



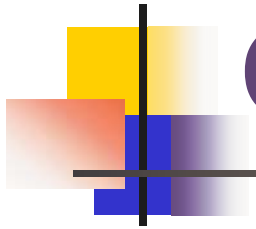
Counting Bloom Filters

- 对于元素经常变动的集合，如频繁的进行元素的插入和删除操作。
- 向Bloom filter中插入元素是很容易的，只要将该元素所有哈希到的位置的比特置1就行了。
- 但是用相反的方法去表示元素的删除是行不通的。
- 因为如果将要删除的元素哈希到的所有位置的比特置0，就有可能将集合中其他某个元素哈希到的位置的比特置成了0。



Counting Bloom Filters

- 为避免这个问题，引入了Counting Bloom Filters。
- 在Counting Bloom Filters中，每一项不再是1个比特位，而是一个小计数器。
- 当增加一个元素时，只要将对应的计数器加1就可以；
- 当删除一个元素时，只要将对应的计数器减1就行了。为避免溢出，计数器要选得足够大

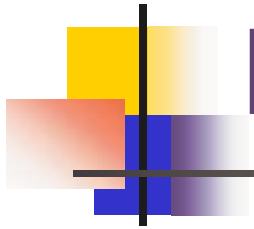


Compressed Bloom Filters

- Compressed Bloom Filters的提出是基于以下问题：假设一台服务器通过网络向其他服务器发送一个Bloom filter，能否从压缩的Bloom filter中获取到信息？
- 这种思想的理论依据如下：通常情况下，是通过限制m的值来优化错误命中率的。

Compressed Bloom Filters

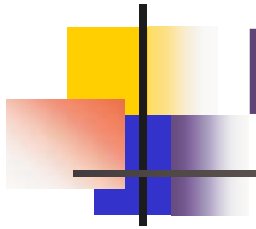
- 假设现在通过限制要传输的比特数(经过压缩后是z)来优化错误命中率，而未经压缩时的m可以更大。
- 事实证明，采用更大、更松散的Bloom filter，只要传输较少的比特数就可以使错误命中率和标准Bloom filter相同



Hierarchical Bloom Filters

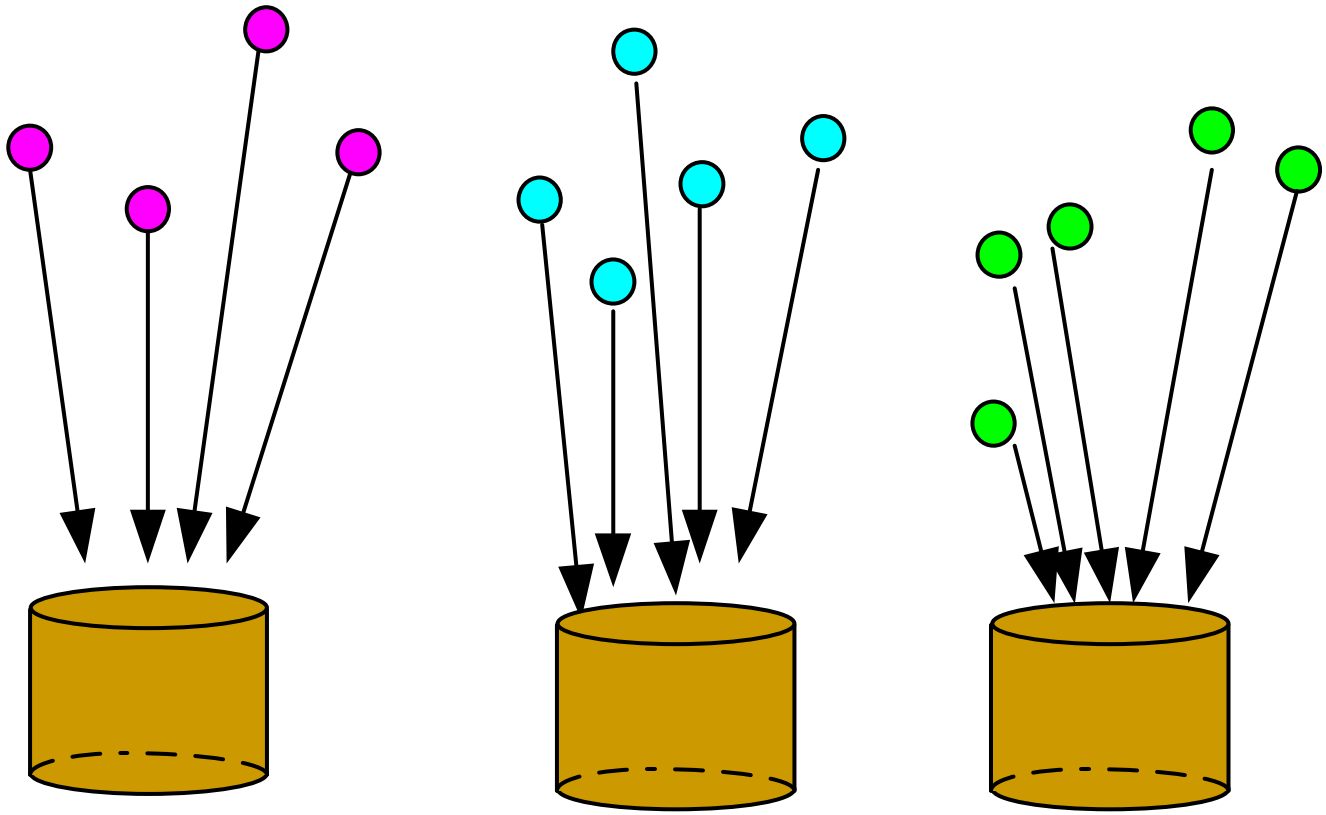
- HBA(Hierarchical Bloom Filter Array)
- 采用两级Bloom Filter结构来表示一个节点上访问量不同的资源，
- 既可以保证查询的准确率，也能节约缓存空间

Hierarchical Bloom Filters



- HBA的实现原理如下：
- 在元数据服务器上维护一张本地最近被访问的资源表(LRU),用第一级Bloom filter(LRU BF)来表示该表中的资源，LRU BF的副本发送到每个元数据服务器的缓存中。
- 当LRU表发生溢出时，LRU替换策略将会在相应LRU BF中触发一个增删操作。
- 当LRU BF的改变超过一个阈值时，该元数据服务器将会向其他所有元数据服务器广播新的LRU BF来更新它在其他服务器的副本

基于LSH的设计和实现



Locality Sensitive Hashing (LSH)

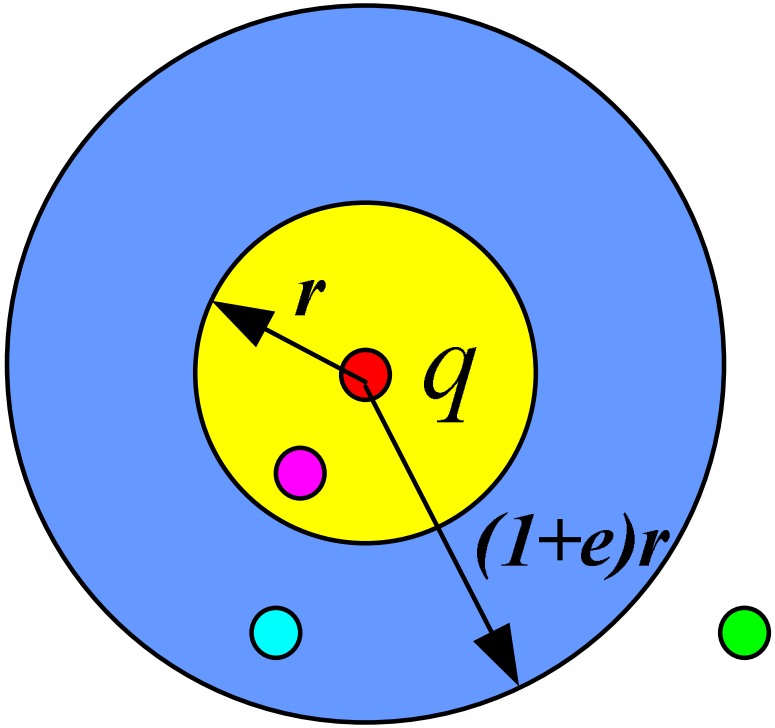
- If $||p,q||_s \leq R$ then $Pr_{\mathbb{H}}[h(p) = h(q)] \geq P_1$,
- If $||p,q||_s > cR$ then $Pr_{\mathbb{H}}[h(p) = h(q)] \leq P_2$.

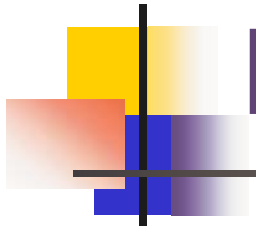
Near neighbor?

yes

not sure

no

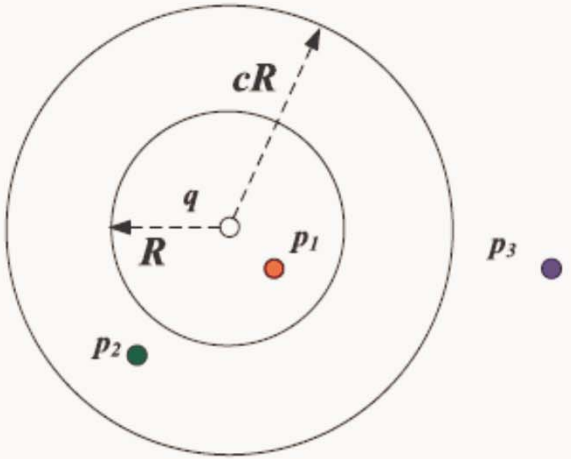




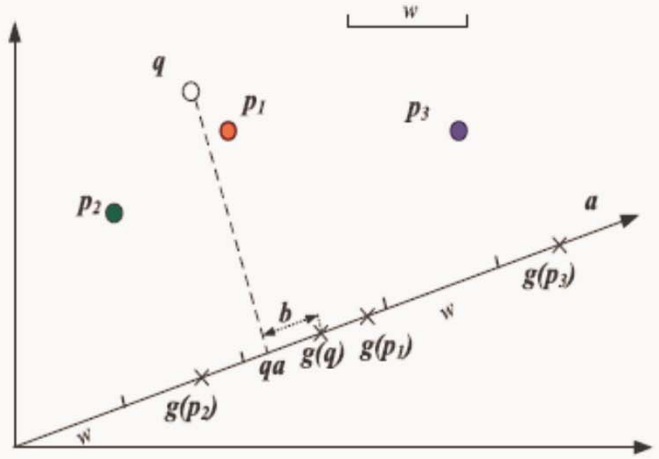
Limitations of Standard LSH

- LSH identifies similar data by allowing them to be placed into the same hash buckets with a high probability
- The similar data have a higher probability of colliding than the data that are far apart
- LSH needs to hash a query point into the buckets of multiple hash tables

基本原理



(a) Illustration of measured distance.



(b) Geometry result of hash functions.

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{\omega} \right\rfloor,$$