# Facebook Haystack

<https://www.cnblogs.com/yuki-lau/archive/2013/03/29/2988594.html>

<http://www.importnew.com/3292.html> 差不多就是Haystack的论文以及与TFS的比较

Facebook每张照片平均80KB，用户每周新增照片10亿，总大小60TB，平均每秒新增照片数为3500次，也就是每秒约3500次写操作，读操作峰值可达每秒百万次。用户上传超过65 billion的图片，对每个上传的照片，Facebook生成和存储4种不同大小的图片，这就产生了超过260 billion张图片，超过20 PB的数据（考虑副本情况，而且一张照片被存储为4种不同大小的）。

Facebook最初采用的是NAS(Network Attached Storage)和NFS(Network File System)存储图片，这种设计因为元数据查询而导致了过多的磁盘操作。所以要竭尽全力的减少每个图片的元数据，让Haystack能**在内存中执行所有的元数据查询**。这个突破让系统腾出了更多的性能来读取真实的数据，增加了整体的吞吐量。

传统基于POSIX的文件系统的缺点主要是目录和每个文件的元数据。对于图片应用，很多元数据（比如文件权限），是无用的而且浪费了很多存储容量。而且更大的性能消耗在于文件的元数据必须从磁盘读到内存来定位文件。文件规模较小时这些花费无关紧要，然而面对几百billion的图片和PB级别的数据，访问元数据就是吞吐量瓶颈所在。

通常情况下，我们读取单个照片就需要好几个磁盘操作：一个（有时候更多）转换文件名为inode number，另一个从磁盘上读取inode，最后一个读取文件本身。简单来说，为了查询元数据使用磁盘I/O是限制吞吐量的重要因素。在实际生产环境中，我们必须依赖内容分发网络来支撑主要的读取流量，即使如此，文件元数据的大小和I/O同样对整体系统有很大影响。

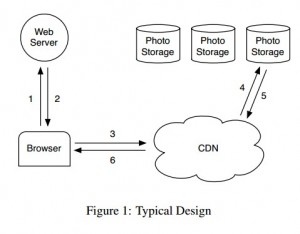
Haystack要实现高吞吐量和低延迟，满足海量用户的查询请求。通常超过处理容量上限的请求，要么被忽略（对用户体验是不可接受的），要么被CDN处理（成本昂贵而且可能遭遇一个性价比转折点）。想要用户体验好，图片查询必须快速。Haystack希望每个读操作至多需要一个磁盘操作，基于此才能达到高吞吐量和低延迟。为了实现这个目标，我们竭尽全力的减少每个图片的必需元数据，然后将所有的元数据保存在内存中。

## 存储场景

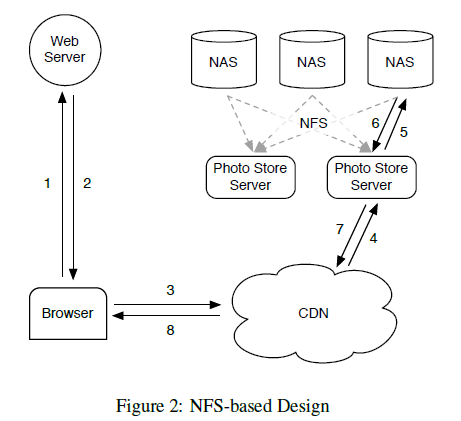
Haystack的使用场景是：

1. write once, read often, never modified and rearly deleted;
2. 海量小文件。Facebook存储了海量的图片文件；用户上传一张图片，Facebook会将其生产4种尺寸的图片，每种存储3份，因此写入量是用户上传量的12倍（准确的说，由于尺寸的缩小，写入量不应该是12倍，二是写入次数是12倍）。

## 原有设计



这是一个概览图，描述了web服务器、CDN和存储系统如何交互协作来实现一个热门站点的图片服务。访问一个页面时，浏览器发送请求到web服务器，对于请求中包含的图片，web服务器会为其构造URL，引导浏览器到对应的位置下载图片。如果是热门的图片，这个URL通常会指向一个CDN，如果CDN缓存了此图片，那么它立刻就会把数据返回给浏览器；如果CDN没有缓存图片，CDN检查URL，URL中包含了足够的信息以供CDN从本站点的存储系统中检索图片。拿到图片后CDN更新它的缓存并将图片发送给浏览器。



上图是基于NFS的图片存储系统设计。对于一个热门的社交网络站点，只有CDN不足以为图片服务提供一个实用的解决方案。对于热门图片，CDN确实可以很高效，但是像Facebook这样的社交网站，会产生大量对不热门图片的请求，称为long tail，long tail的请求也会占据很大流量，并且这些请求还需要访问更下游的图片存储主机，因为这些图片的请求在CDN基本都会命中失败。缓存所有的图片到CDN倒是可以解决问题，但是代价太大了。

基于NFS的设计中，图片文件存储在一组商用NAS设备上，NAS设备的卷被mount到Photo Store Server的NFS上。Photo Store Server解析URL得出卷和完整的文件路径，在NFS上读取数据，然后返回结果给CDN。

最初NFS卷的每个目录下存储几千个文件，导致读取文件时产生了过多的磁盘操作，即使只是读取单个图片，由于NAS设备管理目录元数据的机制，放置几千个文件在一个目录是极其低效的，因为目录的blockmap太大不能被设备有效的缓存。因此检索单个图片都可能需要超过10个磁盘操作。在减少到每个目录下几百个图片后，系统仍然大概需要3个磁盘操作来获取一个图片：一个读取目录元数据到内存、第二个装载inode到内存、最后读取文件内容。

为了减少磁盘操作，我们让图片存储服务器明确的缓存NAS设备返回的文件“句柄”。第一次读取一个文件时，图片存储服务器正常打开一个文件，将文件名与文件“句柄”的映射缓存到memcache中。同时，我们在os内核中添加了一个通过句柄打开文件的接口，当查询被缓存的文件时，图片存储服务器直接用此接口和“句柄”参数打开文件。遗憾的是，文件“句柄”缓存改进不大，因为越冷门的图片越难被缓存到（没有解决long tail问题）。值得讨论的是可以将所有文件“句柄”缓存到memcache，不过这也需要NAS设备能缓存所有的inode信息，这么做是非常昂贵的。总结一下，我们从NAS方案吸取的主要教训是，仅针对缓存——不管是NAS设备缓存还是额外的像memcache缓存——对减少磁盘操作的改进是有限的。存储系统终究是要处理long tail请求（不热门图片）。

**一个文件对应至少一个inode（如果考虑不同尺寸和副本），一个inode已经占了几百byte了，所以在海量图片的情况下，真的需要非常多的内存才可以。（所以要把多个图片存储在一个文件中，这样可以减少很多的内存空间）**

**原有设计的问题**

在POSIX标准的文件系统中：

1. 每个目录、文件都含有元数据，并包含权限等在facebook图片存储场景中无用的信息，占用了不必要的存储空间；
2. 查找一个文件时，需要先将磁盘上的元数据load到内存中，才能找到目标文件，磁盘访问多，效率低；

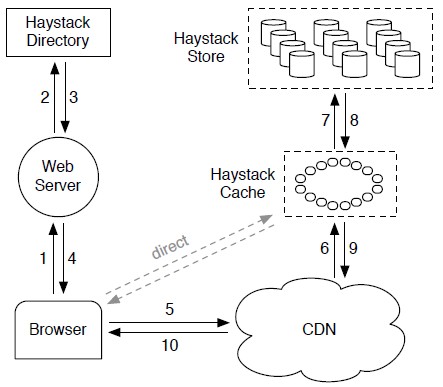
元数据访问成为瓶颈。

Facebook想要使用CDN来支撑热门图片的查询，结合Haystack解决long tail问题。Web站点在查询静态内容遇到I/O瓶颈时，传统的解决方案是使用CDN，它可为下游的存储系统挡住绝大部分的查询请求。但是在Facebook中，为了使底层存储系统不受I/O的摆布，CDN需要缓存难以想象的海量静态内容。

即使有CDN，磁盘操作依然是系统的瓶颈，long tail的请求必然要导致磁盘操作的实现，Haystack会尽量减少除了访问真实图片数据之外其他的操作，并且有效的减少了文件系统元数据的空间，并在内存中保存所有元数据。

## Haystack结构

Haystack的目标：Avoid disk operations when accessing metadata.为了实现这一目标，Haystack系统主要包括三部分：目录、存储以及缓存。



**Haystack Directory**

存放逻辑卷轴和物理卷轴的对应关系，以及照片id到逻辑卷轴之间的映射关系。Haystack缓存主要用于解决对CDN提供商过于依赖的问题，提供最近增加的照片的缓存服务。

1. 逻辑卷和物理卷的对应关系，web服务器上传图片和构建图片URL是都需要这个映射关系；
2. Directory在分配写请求到逻辑卷、分配读请求到物理卷时需要保证负载均衡；
3. Directory要决定一个图片请求应该被发至CDN还是cache，这个功能可以动态调整是否要依赖CDN；
4. Directory指明哪些逻辑卷是只读的，只读意味着存储容量达到上限或者处于运维的原因等，为了运维方便，以机器的粒度来标记卷的只读。

**我们考虑一个网页请求带有图片的情况。网页中包含图片的原始URL，在Directory中利用应用元数据和原始URL构造出“引导URL”以供下游使用。目前只知道Directory中包含了逻辑卷和物理卷之间的映射关系，只有这些关系能够得到最终的URL吗？通常来说原始URL应该只会包含图片的ID，那么Directory如何得知Logical volumn的信息？Directory是不是在upload阶段将photo id与逻辑卷的映射也保存了？如果是，那么这个映射的数量将是非常大的，论文中却没有提起。（Directory中应当保存了很多与photo id相关的元数据，比如所在的逻辑卷、cookie等）**

**Haystack Cache**

cache类似一个内部的CND，它为store提供热门图片的cache，并在上游的CDN节点出现故障时提供庇护，接受客户端的直接请求以重新读取内容。

cache会从CDN或者直接从用户浏览器收到图片查询请求。cache的实现可以理解为一个分布式的Hash table，使用photo id作为key来定位缓存的数据。如果cache未命中，cache根据URL从指定的机器上获取图片，视情况回复给CDN或者用户浏览器。

关于cache中缓存数据的问题值得注意，如果CDN发生了read miss，在cache又发生的read miss，则需要访问store获取数据。按照一般的理解，这个数据应该被缓存在cache中，但是Haystack中，只有符合下面两个条件中的一个时才会缓存图片：

1. The request comes directly from a user and not the CND（一个请求在CDN没有命中一定是非热门图片，那么cache也无需命中，即使图片接下来可能变得活跃，那么也只需要在CDN中可以命中就可以了）;
2. The photo is fetched from a write-enabled store machine.（表明这个图片是最近写入的，因为较早写入的图片所在的物理机很可能已经写满，进入read-only状态）

这两点判断是Facebook对图片分享应用的用户分析得到的：照片在上传后不久就会被访问得最多，对于我们的工作负载，文件系统在读取或写入时通常性能更好，而不是同时读和写。

**Haystack Store**

store机器仅需要逻辑卷ID和offset就能非常快的访问一个图片。**Haystack设计的目的：不需要磁盘操作就可以检索文件名、偏移量、文件大小等元数据。** store机器会将其下所有物理卷的文件描述符（open打开的文件句柄）缓存在内存中，同时photo id到文件系统元数据（文件、偏移量、大小）的映射是检索图片的重要条件，也会缓存在内存中。

store是物理存储节点，以物理卷轴（physical volume）的形式组织存储空间，每个物理卷轴一般都很大，比如100G，这样10TB的数据也只需要100个物理卷轴。每个物理卷轴对应一个物理文件，因此，每个存储节点上的物理文件元数据都很小。多个物理存储节点上的物理卷轴组成一个逻辑卷轴（logical volume），用于备份，Haystack将一个图片存储到逻辑卷轴时，图片被写入到逻辑卷轴对应的所有物理卷，这样实现备份。

Haystack每个逻辑卷轴大小100GB，减少了元信息，但是增加了迁移的时间。

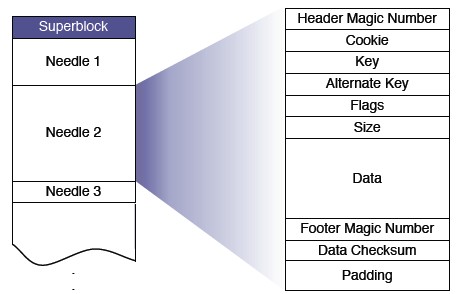
当用户访问一个页面，web服务器使用Directory为每一个图片来构建URL（Directory中有足够的应用元数据来构造URL）。URL包含几块信息，每一块内容可以对应到从浏览器访问CDN（或者cahce）直至最终在一台Store机器上检索到图片的各个步骤。一个典型的URL如下：

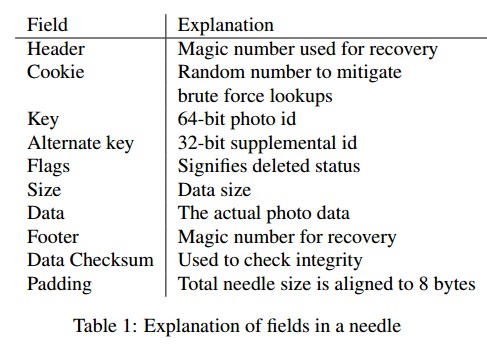
http://CND/cache/Machine id/<Logical volume, Photo>/

CDN指明了从哪个CDN查询图片，到CDN之后再使用<Logical volume, Photo>即可查找缓存的图片。如果CDN没有命中，它从URL中删除CDN相关的信息，然后访问cache。cache其实也是CDN的功能，不过是Haystack系统内部的CDN，所以在cache中查找的过程和CDN类似。如果cache中也没有命中，则从URL中删除cache相关的信息，请求接着被发送给指定Machine id的机器。

## 图片元数据压缩

将多个图片作为一个文件进行存储，这样就可以避免每个图片存储一个文件带来的大量文件元数据的问题。并将该文件称为一个superblock，superblock中的每个图片就称为Needle，一个Needle包含该图片的元数据，和其实际的数据内容，如下图所示：



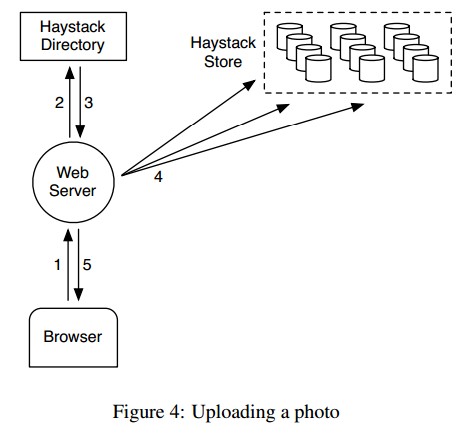


为了快速检索needle，store机器需要为每个卷维护一个内存中的key-value映射。映射的key就是key（photo id）和alternate key的组合，映射的value就是flag、size和卷offset。如果store机器崩溃、重启，它可以分析卷文件来重建这映射关系。

新图片写入的方式与log类似，采用追加的方式。如果一个图片被删除了，则只需对Flags标志位进行置位，并在compaction阶段进行真正的删除。

haystack将每个小文件（needle）在大文件（superblock）中的分布映射保存在内存中，这就减少了查找文件所需的磁盘访问次数。而且这些元信息是根据应用需求而定制的，每个图片需要10字节的元信息（4个尺寸的图片共40字节），而 XFS 需要536字节。

## 写入图片



用户上传一个图片到Haystack时，首先通过浏览器发送数据到web server，web server随后向Directory咨询得到一个可写逻辑卷及其对应的多台Store机器，并且为图片分配一个唯一的ID，随后直接访问这些Store机器，向其提供逻辑卷id、key、alternate key、cookie和真实数据。

每个Store机器为图片创建一个新needle，append到相应的物理卷文件，更新内存中映射。过程很简单，但是append-only策略不能很好的支持修改性的操作，比如旋转（图片顺时针旋转90度之类的）。Haystack并不允许覆盖needle，所以图片的修改只能通过添加一个新needle，其拥有相同的key和alternate key。如果新needle被写入到与老needle不同的逻辑卷，则只需要Directory更新它的应用元数据，未来的请求都路由到新逻辑卷，不会获取老版本的数据。如果新needle写入到相同的逻辑卷，Store机器也只是将其append到相同的物理卷中。Haystack利用一个十分简单的手段来区分重复的needle，那就是判断它们的offset（新版本的needle肯定是offset最高的那个），在构造或更新内存中映射时如果遇到相同的needle，则用offset高的覆盖低的。

## 读取图片

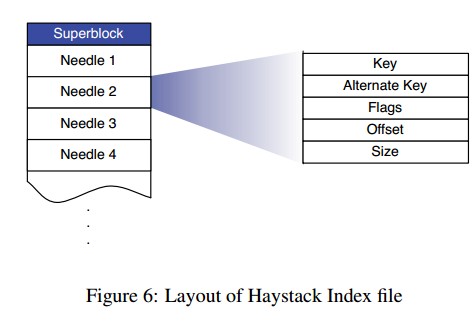
cache向store请求一个图片时，它提供逻辑卷id、key、alternate key和cookie。cookie是一个数字，存在于URL中。新图片上传时Directory为其随机分配一个cookie值，cookie值会作为应用元数据之一存储在Directory。cookie相当于是图片的“私人密码”，可保证所有发往cache或者CDN的请求都是经过Directory批准的。（Cache和Store都持有图片的cookie，若用户自己在浏览器中伪造、猜测URL或发起攻击，则会因为cookie不匹配而失败，从而保证Cache、Store能放心处理合法的图片请求）

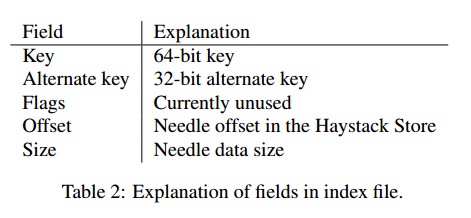
当store机器接收到cache机器发来的图片请求查询时，会利用内存中的映射快速查找相关的元数据。如果图片没有被删除，store则在卷文件中seek到相应的offset，从磁盘上读取整个needle，然后检查cookie和数据完整性，若全部合法则将图片数据返回到cache机器。

## 索引文件

在前面的图片元数据压缩部分提到，store机器故障重启时，它可以分析卷文件来重建卷中元数据的映射关系。起始Haystack使用了索引文件来帮助重启初始化操作。尽管理论上机器可以通过读取所有的物理卷来重新构建它在内存中的映射，但是考虑磁盘大量的数据时，读磁盘还是非常耗时的。索引文件允许store机器快速构建内存中的映射，尽量减少重启时间。

store机器为每个卷维护一个索引文件，索引文件可以想象成内存中映射的一个“存档”，索引文件的布局和卷文件类似，一个超级块包含了一系列索引记录，每个记录对应到各个needle。**索引文件中的记录与卷文件中对应的needle必须保证相同的存储顺序**。





使用索引帮助重启稍微增加了系统复杂度，因为索引文件都是异步更新的，这意味着当前索引文件中的“存档”可能不是最新的。当我们写入一个新图片时，store机器同步append一个needle到卷文件末尾，并异步append一个记录到索引文件。当我们删除图片时，store机器在对应needle上同步设置flag，而不会更新索引文件。这些设计决策是为了让写和删除操作更快返回，避免附加的同步磁盘写。但是也导致了两方面的影响：一个needle可能没有对应的索引记录、索引记录中无法得知图片已删除。

我们将对应不到任何索引记录的needle称为“孤儿”。在重启时，store机器顺序的检查每个孤儿，重新创建匹配的索引记录，append到索引文件。我们能快速的识别孤儿是因为索引文件中最后的记录能对应到卷文件中最后的非孤儿needle。处理完孤儿问题，store机器则开始使用索引文件初始化它的内存中映射。

由于索引记录中无法得知图片已删除，store机器可能去检索一个实际上已经被删除的图片。为了解决这个问题，可以在store机器读取整个needle后检查其flag，若标记为已删除，则更新内存中映射的flag，并回复Cache此对象未找到。

## 故障恢复

为了主动找到有问题的Store机器，我们维护了一个后台任务，称之为pitchfork，它周期性的检查每个Store机器的健康度。pitchfork远程的测试到每台Store机器的连接，检查其每个卷文件的可用性，并尝试读取数据。如果pitchfork确定某台Store机器没通过这些健康检查，它会自动标记此台机器涉及的所有逻辑卷为只读。我们的工程师将在线下人工的检查根本故障原因。

一旦确诊，我们就能快速的解决问题。不过在少数情况下，需要执行一个更加严厉的bulk同步操作，此操作需要使用复制品中的卷文件重置某个Store机器的所有数据。Bulk同步发生的几率很小（每个月几次），而且过程比较简单，只是执行很慢。主要的瓶颈在于bulk同步的数据量经常会远远超过单台Store机器NIC速度，导致好几个小时才能恢复。我们正积极解决这个问题。

## 优化

1. 压缩

压缩操作是直接在线执行的，它能回收已删除的、重复的needle所占据的空间。Store机器压缩卷文件的方式是，逐个复制needle到一个新的卷文件，并跳过任何重复项、已删除项。在压缩时如果接收到删除操作，两个卷文件都需处理。一旦复制过程执行到卷文件末尾，所有对此卷的修改操作将被阻塞，新卷文件和新内存中映射将对前任执行原子替换，随后恢复正常工作。

1. 节省内存

上面描述过，Store机器会在内存中映射中维护一个flag，但是目前它只会用来标记一个needle是否已删除，有点浪费。所以我们通过设置偏移量为0来表示图片已删除，物理上消除了这个flag。另外，映射Value中不包含cookie，当needle从磁盘读出之后Store才会进行cookie检查。通过这两个技术减少了20%的内存占用。

当前，Haystack平均为每个图片使用10byte的内存。每个上传的图片对应4张副本，它们共用同一个key（占64bits），alternate keys不同（占32bits），size不同（占16bits），目前占用(64+(32+16)\*4)/8=32个bytes。另外，对于每个副本，Haystack在用hash table等结构时需要消耗额外的2个bytes，最终总量为一张图片的4份副本共占用40bytes。作为对比，一个xfs\_inode\_t结构在Linux中需占用536bytes。

1. 批量上传

磁盘在执行大型的、连续的写时性能要优于大量小型的随机写，所以我们尽量将相关写操作捆绑批量执行。幸运的是，很多用户都会上传整个相册到Facebook，而不是频繁上传单个图片。因此只需做一些巧妙的安排就可以捆绑批量upload，实现大型、连续的写操作。