东隅

对象存储系统 Haystack

2020-02-18 | computer science | 13.9k

Haystack 是 Facebook 设计的对象存储系统 (Object Storage System), 本文翻译自 Finding a Needle in Haystack: Facebook's Photo Storage, 原文开放下载.

译序: 在翻译本文的过程中, 首先第一步就遇到标题需要如何翻译的困难. Finding a needle in haystack 是一句英文俗语, 直接含义是"在干草堆中找一根针", 与中文成语"大海捞针"意思相近, 也符合在海量数据中检索某张图片的应用场景, 但将其翻译为"大海捞针"则体现不出系统名为 haystack 的用意, 另外 Haystack Store 中图片都是按照 needle (针) 结构组织的, 同时 Haystack 的状态自检系统叫做 Pitchfork (干草叉), 给翻译带来较大的困难. 最终经过考虑, 由于 Haystack 与 Pitchfork 都属于系统名称, 故保留其英文表达不翻译, 而 needle 借用 pointer 概念将其译为指针 (不是很准确, 因为指针原本不会存储实际数据, 但能理解其含义即可), 而标题则保留原文的英文表述.

摘要: 本文介绍了 Haystack, 一个针对 Facebook 图片应用特别设计的对象存储系统. Facebook 存储超过 2600 亿张图片, 总数据量超过 20PB. 用户每周上传新的图片超过 10 亿张 (约 60TB), 而 Facebook 服务器在峰值时每秒钟需要分发超过 100 万张图片. 相较于此前基于网络附属存储 (Network Attached Storage, NAS) 的网络文件系统 (Network File System, NFS) 实现方式, Haystack 提供了一种开销更低, 性能更好的解决方案. 我们重点观察到传统的设计方案会因为元数据查询而带来过多的磁盘操作, 我们适当地降低了每张图片的元数据量使得 Haystack 存储设备可以在内存中执行所有的元数据查询操作. 这种设计选择能够节省部分磁盘操作性能以供读取实际数据, 增加了系统的整体吞吐量.

1 简介

分享图片是 Facebook 最热门的功能之一. 截至目前, 用户已经上传了超过 650 亿张图片, 使得 Facebook 成为世界上最大的图片分享网站. 对每张上传的图片, Facebook 会生成 4 张尺寸不同的图片并存储, 对应于超过 2600 亿张图片和 20PB 的数据量. 用户每周上传的新图片超过 10 亿张 (约 60TB), 而 Facebook 服务器在峰值时每秒钟需要分发超过 100 万张图片. 鉴于未来预期这些数字还会增长, 故图片存储问题给 Facebook 的基础设施带来了显著的挑战.

本文介绍了 Haystack 的设计和实现, Haystack 是 Facebook 的图片存储系统, 在过去 24 个月一直运行在生产环境中. Haystack 是我们为在 Facebook 上分享图片而设计的对象存储系统 [7, 10, 12, 13, 25, 26], 在此场景中, 数据仅有一次写入, 频繁访问, 从不修改, 很少删除. 由于传统的文件系统在我们的应用场景下表现不佳, 所以我们设计了自己的存储系统.

根据我们的经验, 传统的基于 POSIX [21] 的文件系统的不足在于目录和每个文件的元数据. 对于图片应用而言, 包括文件权限在内的绝大多数的元数据都是无用进而浪费存储空间的. 除此以外, 更加显著的开销在于操作系统必须将文件的元数据从磁盘读入内存才能定位文件. 虽然这些开销在小数据量情况下不显著, 但对于数十亿的图片和 PB 级别的数据, 元数据访问则会成为系统吞吐量的瓶颈, 我们发现这就是使用基于 NAS 的 NFS 的关键问题. 读取单张图片需要多次磁盘操作: 一次 (往往是多次) 操作从文件名得到索引节点 (index node, inode) 编号, 一次从磁盘读取索引节点, 一次从磁盘读取文件本身. 简而言之, 为获取元数据而执行磁盘读写操作是限制吞吐量的关键因素. 实际上, 由于需要使用例如 Akamai [2] 的内容分发网络 (Content Delivery Network, CDN) 来承担主要的读流量, 上述问题会给我们带来额外开销.

由于传统方法存在缺陷, 我们设计了 Haystack 来达成以下四个主要目标:

高吞吐量与低延迟. 我们的图片存储系统必须能够承受用户请求, 超过系统处理能力的请求或者被直接忽略 (就用户体验而言无法接受), 或者被交由 CDN 处理 (开销大并且可能会达到收益转折点). 除此以外, 为了具备良好的用户体验, 系统应该提供更加快速的图片服务. Haystack 对每个读请求执行至多一次磁盘操作, 从而实现高吞吐量和低延迟的目标, 我们通过显著地减少查找图片需要的元数据量, 进而将所有元数据保存在内存中来达成此目标.

容错. 对于大规模存储系统, 故障每天都会发生. 用户期待他们的图片随时保持高可用性, 所以既便发生了无法避免的服务器崩溃与硬盘驱动故障, 也不能让用户接收到错误信息. 整个数据中心断点或者跨国数据链被切断也是有可能发生的. Haystack 会对每张图片进行复制并存储在物理空间上不同的地点. 如果某台机器失去响应, 我们就引入另一台机器取而代之, 必要时复制数据进行冗余容灾.

高成本效率. 相较于此前基于 NFS 的方式 Haystack 性能更好而且成本更低. 我们从以下两个维度来量化 Haystack 成本效率的提高: 每 TB 可用存储的开销与每 TB 可用存储的读取速率. 相比于 NAS 设备, Haystack 每 TB 可用存储[1] 的开销要减少约 28%, 在数据量相同的情况下 Haystack 每秒钟能够处理约 4 倍于 NAS 设备的读请求.

简单. 在实际的生产环境中, 一个设计的实现与维护必须要足够的简单直接. 由于 Haystack 是一个新系统, 缺乏若干年产品级的测试, 我们尤其注意保持其足够简单. 简洁性使得我们能够在几个月而不是几年内搭建并部署一个可用的系统.

本文介绍了我们设计 Haystack 的经验, 涵盖了一个每日为数十亿图片提供服务的生产级系统从概念到实现的所有内容. 我们的三点主要贡献如下:

- Haystack, 一个为亿数量级图片提供存储与检索服务的对象存储系统.
- 搭建与扩展低成本, 高可靠与高可用的图片存储系统的经验教训.
- 对 Facebook 图片分享应用发出的网络请求的分析与特征描述.

论文剩余部分的组织方式如下: 第 2 节给出了背景介绍与此前采用的架构面临的挑战, 第 3 节描述了 Haystack 的设计与实现, 第 4 节对图片读写负载进行了特征描述, 论证 Haystack 实现了设计目标, 第 5 节与相关工作进行了比较, 第 6 节总结了整篇文章.

2 背景与此前采用的设计

本节中我们会介绍在 Haystack 之前采用的架构并强调主要的经验教训, 篇幅所限我们对旧设计的讨论省略了产品级部署的若干细节.

2.1 背景

首先我们先简要说明一下热门站点上网络服务器, CDN 和存储系统互相交互以提供图片服务的典型设计. 图 1 展示了从用户访问一个带图片的网页开始到其从磁盘特定位置下载图片为止的各个步骤. 当访问网页时, 用户的浏览器首先对网络服务器发送一个负责生成供浏览器渲染界面的标记 (markup) 的 HTTP 请求. 对于每张图片, 网络服务器会构建一个将浏览器导向图片数据下载地址的 URL. 对热门网站而言, 此类 URL 一般都指向某个 CDN, 如果 CDN 包含有图片缓存, 就会直接响应并返回图片数据, 否则 CDN 会检查提供 URL, 从中提取出获取图片数据需要的信息, 并从网站的存储系统中获取此图片, 最后 CDN 再更新自身的缓存数据并且将图片发送给用户的浏览器.

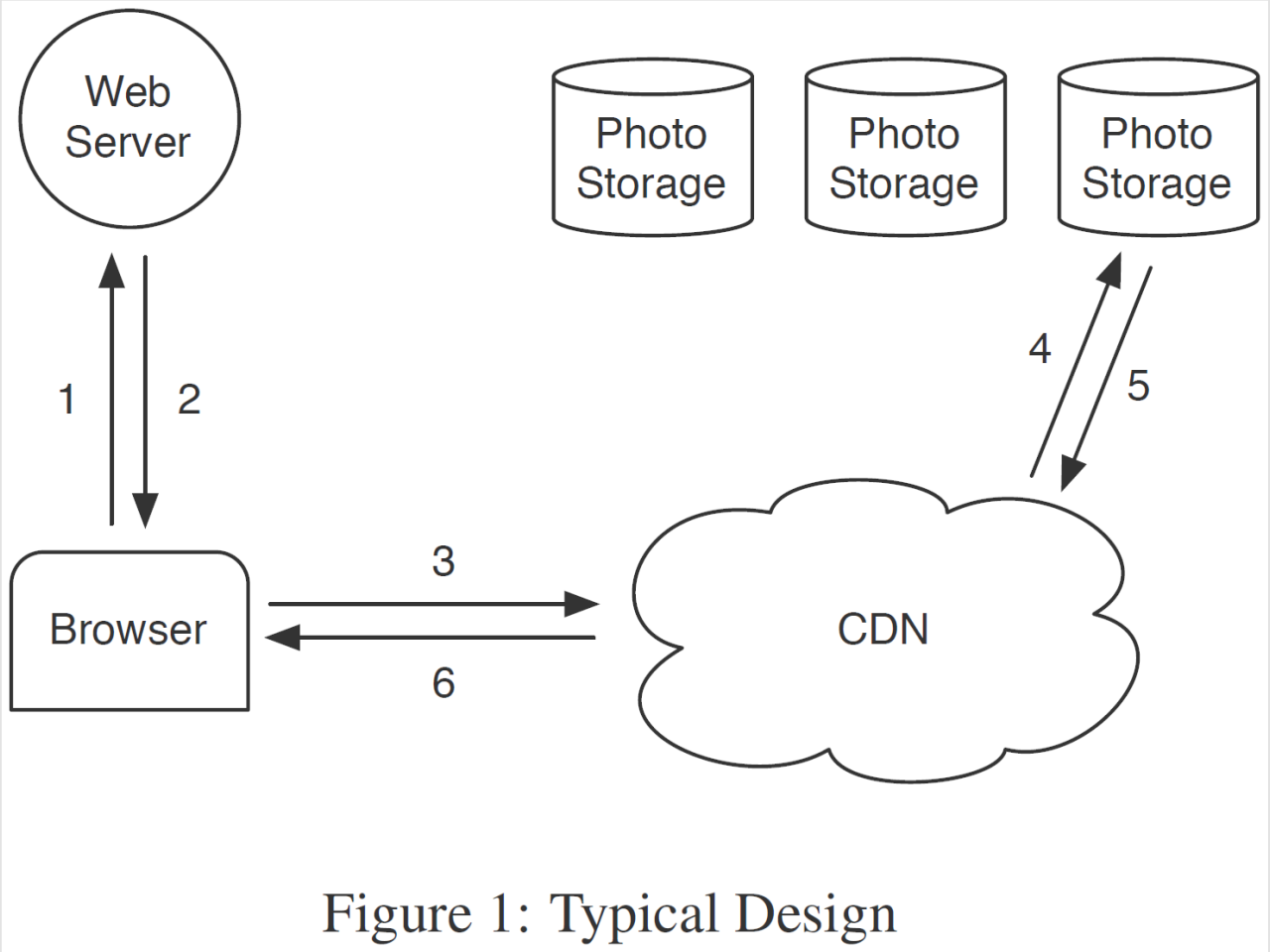

Figure 1: Typical Design

2.2 基于 NFS 的设计

我们 最初实现的图片存储系统是基于 NFS 的, 此小节余下的部分将会对此设计进行更详细的描述. 我们获得的最大教训是, CDN 本身并不能为给社交网站提供图片服务提供行之有效的解决方案. CDN 确实可以高效地分发那些最热门的图片, 例如用户资料照片与最近新上传的图片, 但是像 Facebook 这样的社交网络站点也会产生大量对不那么热门的 (通常是较旧的) 内容的请求, 这种现象我们称之为长尾效应 (long tail). 长尾效应中尾部的请求占用了很大一部分网络流量, 几乎所有此类请求最终都会访问到网站的图片存储主机, 因为它们很难命中 CDN 缓存. 虽然对所有图片都进行缓存可以很方便地解决数据访问地长尾问题, 但是此方法需要的巨大缓存空间使得成本效率较低.

我们此前基于 NFS 的设计将每张图片在商用 NAS 设备上存成一个文件, 然后一组被称为图片存储服务器 (Photo Store Server) 的设备将这些 NAS 设备导出的卷 (volume) 挂载到 NFS 上. 图 2 是对这个架构的描述, 并且展示了图片存储服务器处理对图片的 HTTP 请求的具体流程. 图片存储服务器可以从图片的 URL 中提取出卷信息与图片文件的完整路径, 通过 NFS 读取图片数据, 最终将结果返回给 CDN.

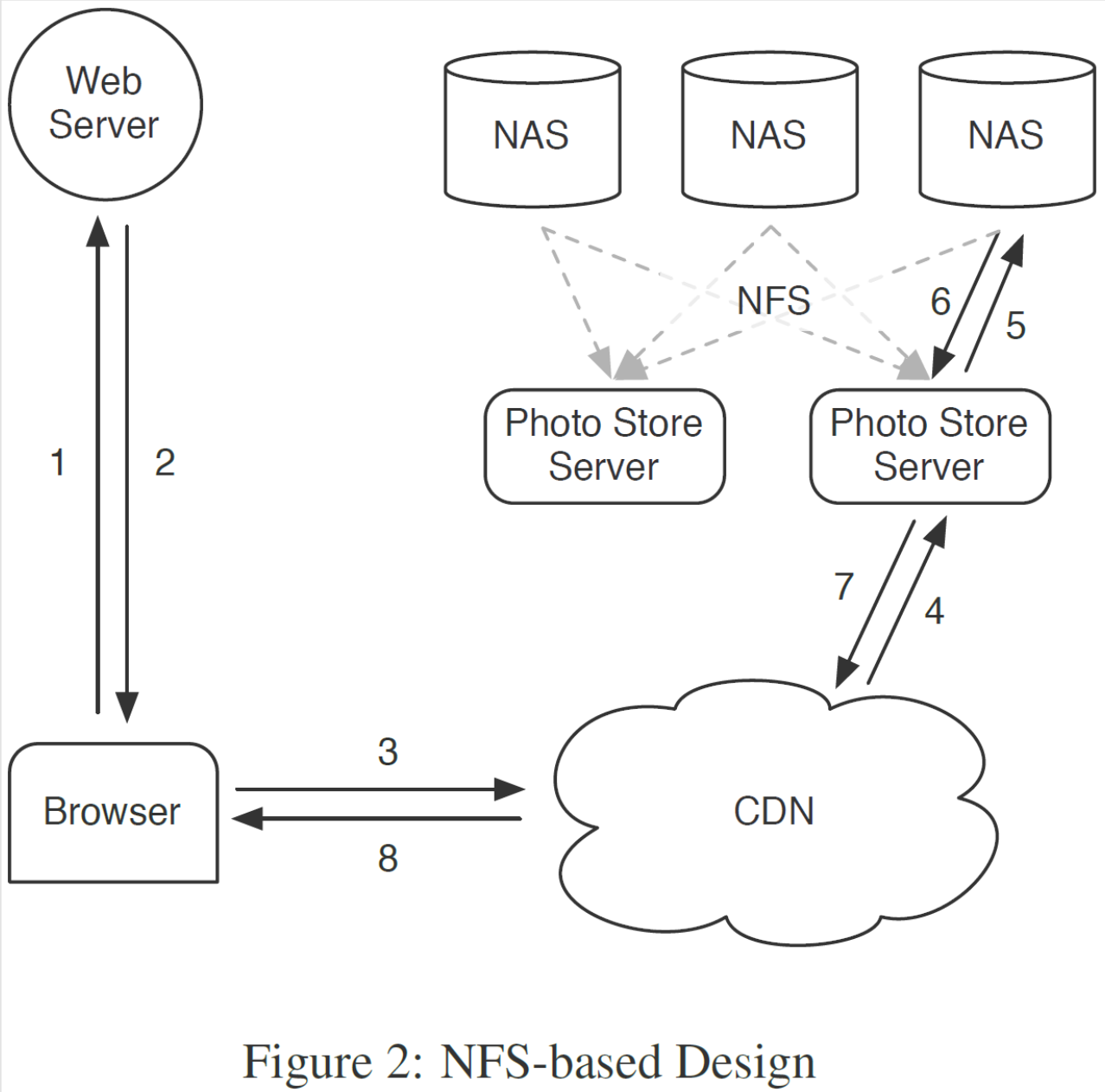

Figure 2: NFS-based Design

最初我们在一个 NFS 挂载卷的每个目录下存储数千个文件, 这导致即便只读取单个文件也会带来大量的磁盘操作. 由于 NAS 设备管理目录元数据的方式, 将数千个文件置于同一个文件夹下是非常低效的, 因为这会导致目录的块表过大, 设备无法对其进行有效的缓存, 结果获取单个文件通常需要超过 10 次磁盘操作. 即便将文件数量减少到每个目录下数百个, 获取单个文件仍然通常需要3次磁盘操作: 一次将目录元数据读入内存, 一次将索引节点读入内存, 最后一次读取文件内容.

为了更进一步减少磁盘操作次数, 我们允许图片存储服务器可以显式地缓存 NAS 设备返回的文件句柄. 当第一次读到某个文件时, 图片存储服务器在打开文件的同时也会将文件名到文件句柄的映射表缓存在 MemCache [18] 中. 当发现请求文件的句柄已经被缓存时, 图片存储服务器可以直接通操作系统内核的定制化系统调用 open_by_filehandle 直接打开文件. 可惜的是, 对文件句柄的缓存仅仅带来了微小的性能提升, 因为较不热门的图片更难被缓存. 有人可能会认为将所有文件的句柄都缓存在 MemCache 中会是一个可行方案, 但此方法仅仅关注到问题的一部分, 因为这要求 NAS 设备将所有的索引节点置于内存中, 对传统文件系统而言开销是巨大的. 我们从 NAS 方法学到的主要教训是, 仅仅把注意力集中在缓存问题

上, 无论是 NAS 设备的系统缓存还是例如 MemCache 的外部缓存, 对减少磁盘操作的影响都非常有限, 存储系统最终总会处理不热门图片的尾部请求, 这些图片不在 CDN 中, 进而很难命中缓存.

2.3 讨论

我们很难对何时需要搭建自制的存储系统这个问题给出明确的准则, 但我们相信我们设计 Haystack 的理由能够给社区一些启发.

面对基于 NFS 设计遇到的性能瓶颈, 我们研究了构建一个类似于 GFS [9] 的系统是否有用. 因为我们绝大多数的用户数据都是存储在 MySQL 数据库中的, 所以文件存储的应用场景多为存储开发工作, 日志数据与图片. NAS 设备已经为前两者提供了性价比很高的存储方案, 面对图片的长尾请求问题, MySQL, NAS 设备与 Hadoop 都不太合适.

有人会认为我们面临的困境可以归结为现存的存储系统缺少正确的内存磁盘比, 然而所谓正确的比率是不存在的, 系统仅仅需要足以将所有文件系统元数据缓存一次的内存容量. 在我们基于 NAS 的方法中, 每张图片对应于一个文件, 每个文件需要至少一个索引节点, 而每个索引节点要占用数百个字节, 这种场景下要有足够的内存是非常高成本的. 为了取得更好的性价比, 我们决定设计自制的存储系统来减少每张图片需要的文件系统元数据, 这样拥有足够的内存就比购入更多的 NAS 设备的成本要低得多.

3 设计与实现

Facebook 使用 CDN 来提供热门图片的服务, 同时用 Haystack 来高效地响应非热门图片的尾部请求. 当一个网站在分发静态内容过程中遇到 I/O 时, 传统的方式是使用 CDN. CDN 可以承担大部分网络请求, 这样存储系统就只用处理剩余的尾部请求. 在 Facebook, 如果要使得传统 (同时也是廉价的) 存储方式不被 I/O 性能限制, CDN 就需要缓存极大量的静态内容.

意识到不远的未来 CDN 就可能不能完全解决我们的问题, 我们设计了 Haystack 来应对基于 NFS 的方法中面临的关键瓶颈: 磁盘操作. 我们承认对非热门图片的请求会带来磁盘操作, 但希望能够将其限制在需要读取实际图片数据的磁盘操作上. Haystack 通过显著地减少文件系统元数据占用的内存来实现这个目标, 进而使得将所有元数据保存在内存中成为可能.

考虑到将每张图片都存储为单个文件会带来超过可缓存量地元数据, Haystack 采取了一种直接的解决方法: 它将多张图片存储为单个文件并且维护大型文件. 我们将证明这种直接的方式是非常有效的, 另外其能够快速实现, 快速部署的简洁性也是一大优点. 加下来我们会讨论这个核心技术要点与其周边的架构组件是如何实现高可靠性, 高可用性的存储系统的. 在后续对 Haystack 的描述中, 我们要区分两种不同的元数据, 应用元数据描述的是用来构造浏览器获取图片的 URL 需要的信息, 文件系统元数据则指代操作系统获取存储在磁盘上的图片需要的信息.

3.1 概述

Haystack 架构由 3 个核心组件构成: Haystack Store, Haystack Directory 与 HayStack Cache, 简洁起见后文我们提到这些组件时就省略”Haystack”前缀. Store 是对图片的持久化存储系统的封装, 并且是管理文件系统元数据的唯一组件. 我们将 Store 的存储空间组织为卷, 例如将单个服务器的 10TB 存储空间划分为 100 个物理卷, 每卷包含 100GB 的存储空间. 我们更进一步地将不同节点上的物理卷分组为逻辑卷, 当 Haystack 对一个逻辑卷执行写入操作时, 图片会写入到所有对应的物理卷上, 这种数据冗余操作可以帮助减轻硬盘驱动故障, 磁盘控制故障等造成的数据丢失问题的不良影响. Directory 维护逻辑层到物理层的映射表, 同时还包含其他应用元数据内容, 例如图片位于哪个逻辑卷, 以及哪个逻辑卷有剩余空间等等. Cache 起到内部 CDN 的功能, 它可以为 Store 承担对热门图片的请求, 并且上游的 CDN 节点失效, 需要重新获取内容时为系统内部提供隔离环境.

图 3 展示了 Store, Directory, Cache 在用户浏览器, 网站服务器, CDN 和存储系统间的典型交互过程. 在 Haystack 架构中, 浏览器可以被定向到 CDN 或 Cache 处, 注意因为 Cache 本质上也是 CDN, 此处我们用”CDN”来指代外部的 CDN 而 Cache 则指代内部用来缓存图片的组件, 在系统内部维护一个缓存结构可以减少对外部 CDN 的依赖.

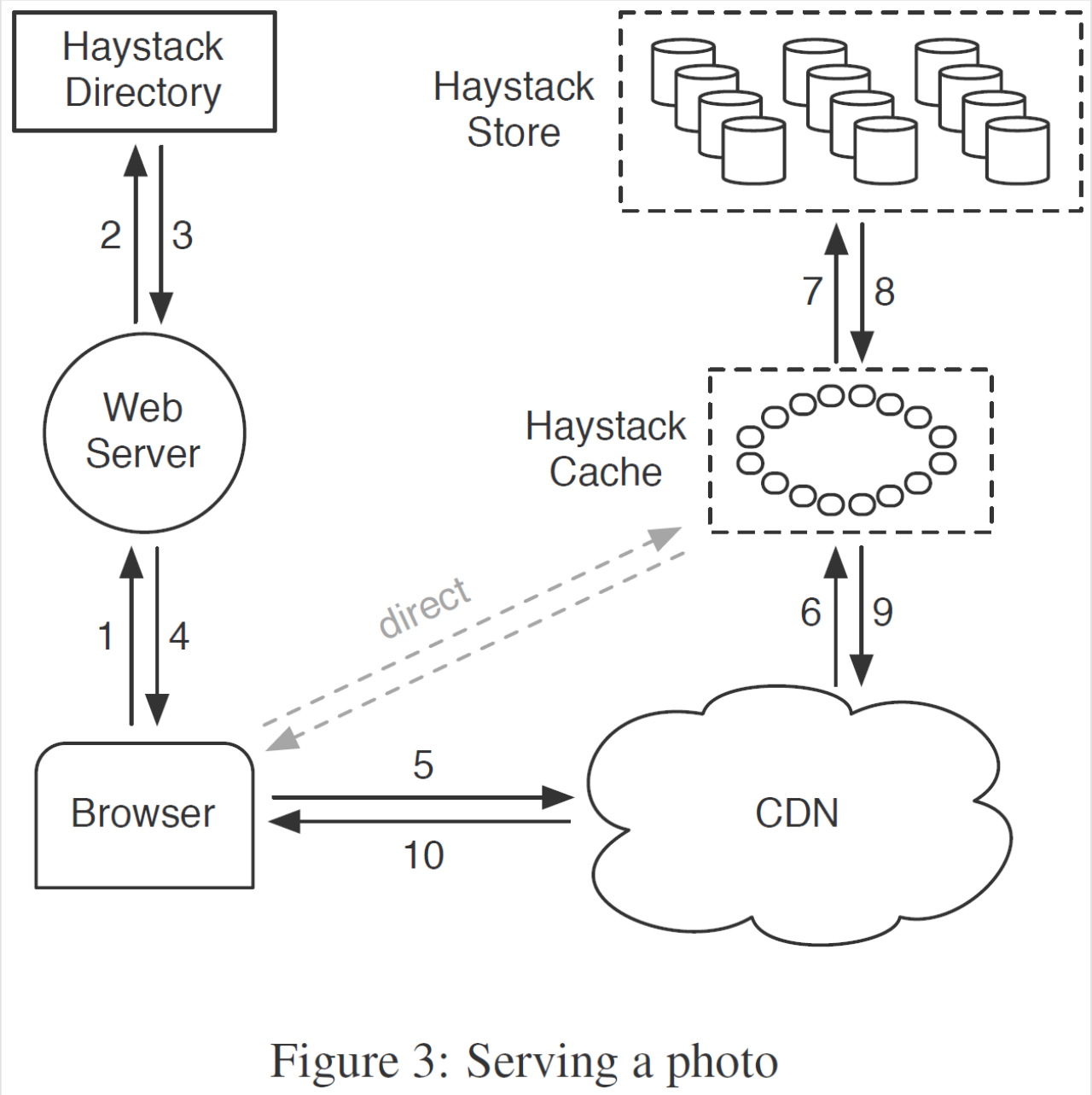

Figure 3: Serving a photo

当用户访问网页时, Directory 组件会为每张图片构建一个 URL, 每个 URL 中都包含了若干段内容, 各段信息对应于用户浏览器访问 CDN (或 Cache) 并最终从 Store 中获取图片数据的一系列步骤. 一个将浏览器定向至 CDN 的典型 URL 如下:

http://<CDN>/<Cache>/<Machine id>/<Logical volume,Photo>

URL 的首段指明了要向哪个 CDN 请求图片, CDN 会使用 URL 的尾段, 也就是逻辑卷与图片编号, 来在内部查找图片. 如果 CDN 没能找到图片, 就将首段的 CDN 地址从 URL 中切除并将剩余部分发送给 Cache, Cache 会做出与 CDN 行为类似的查找. 如果没有找到, 再将 Cache 地址从 URL 中切除, 剩余部分发送给特定的 Store 进行查找. 发送给 Cache 的图片请求和发送给 CDN 的非常类似, 唯一区别在于其接受的 URL 没有指定 CDN 部分的信息.

图 4 描述了 Haystack 中的图片上传路径, 当用户上传图片时, 首先会将数据发送给网络服务器, 然后服务器向 Directory 请求一个可以写入的逻辑卷, 最后网络服务器给图片分配一个唯一编号并将其上传至所给逻辑卷映射到的所有物理卷处.

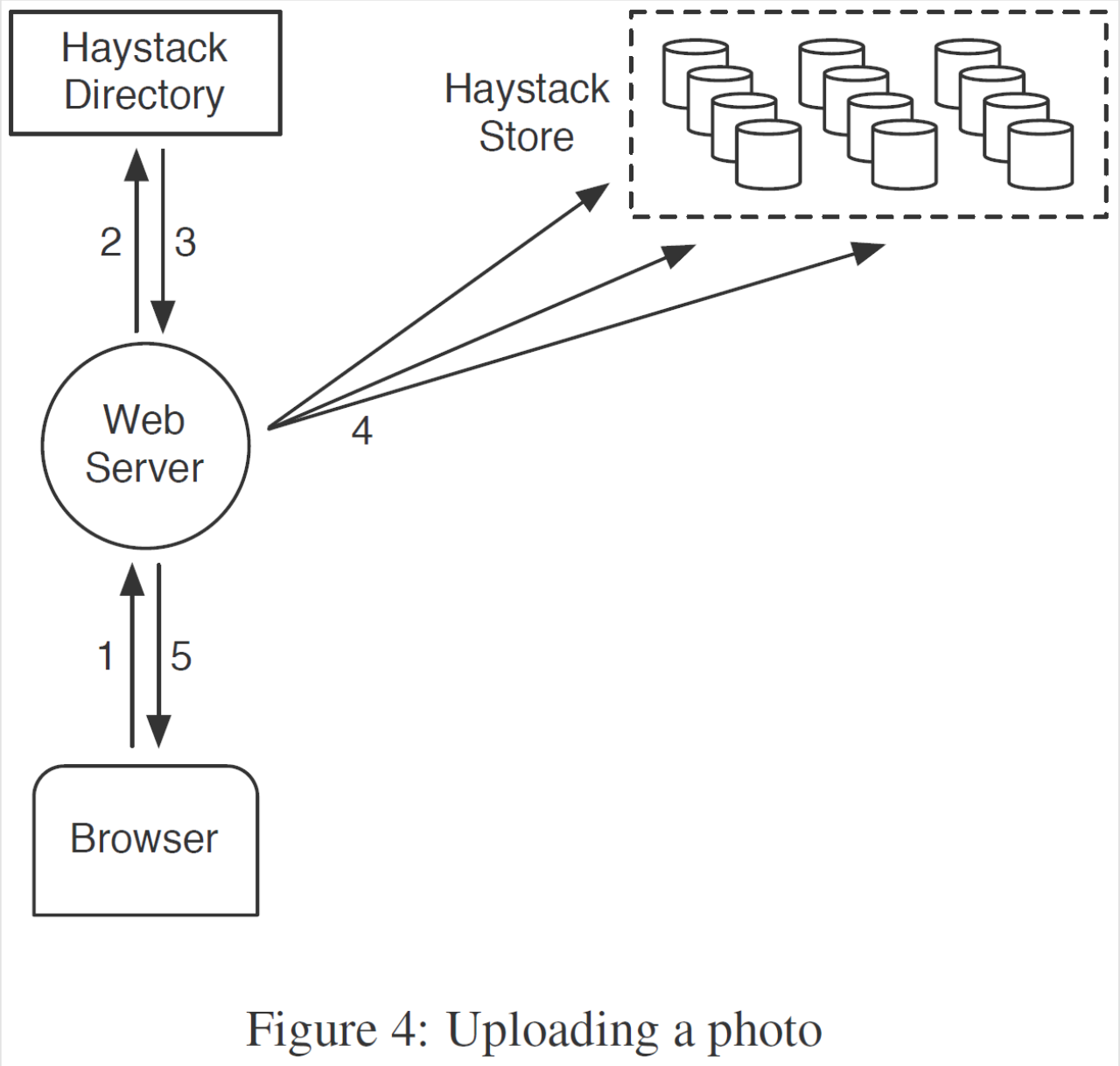

Figure 4: Uploading a photo

3.2 Haystack Directory

Directory 有 4 个主要功能. 第一点, 它提供了从逻辑卷到物理卷的映射表, 访问网页的过程中, 网络服务器会在上传图片以及构建图片 URL 时利用此映射表. 第二点, Directory 负责写请求在各逻辑卷间的负载均衡与读请求在各物理卷间的负载均衡. 第三点, Directory 可以决定一个图片请求应该由CDN还是Cache来处理. 第四点, Directory 可以判定那些逻辑卷是只读的, 或是由于操作上的原因, 或是此卷已经达到了最大存储容量, 出于方便操作的目的, 我们可以在节点粒度上将这些卷标记为只读的.

当通过添加新节点来增加 Store 的容量时, 这些新增加的节点是可读的, 只有可读的节点才会接受上传请求. 这些节点上的可用空间会随着时间而逐渐减小, 当一台节点的可用空间用尽时, 我们就会将其标记为只读的. 在下小节中, 我们将会讨论这种区分会怎样给 Cache 和 Store 带来微妙的影响.

Directory 是设计相对比较直接的组件, 它通过 PHP 接口将自身信息存储在一个利用 MemCache 降低延迟的多副本数据库中. 当一台 Store 节点发生数据丢失时, 我们会将映射表中对应条目删除, 并且当新 Store 节点上线后用新映射条目替换之.

3.3 Haystack Cache

Cache 接收来自 CDN 与用户浏览器的 HTTP 图片请求. 我们将 Cache 的结构组织为分布式哈希表并用图片编号作为键来定位缓存数据. 当 Cache 不能用缓存直接响应请求时, Cache 会用 URL 从 Store 节点中获取图片, 然后恰当地对 CDN 或用户浏览器发出响应.

现在我们要强调 Cache 的一个重要的行为特点. 只有满足 2 个条件时 Cache 才会对图片进缓存: (a) 图片请求直接来自用户浏览器, 而非来自 CDN. (b) 图片从可写的 Store 节点中获取. 根据我们基于 NFS 的设计经验, 位于 CDN 下游的缓存作用很小, 因为没有命中 CDN 缓存的请求是几乎不可能命中内部缓存的, 这是我们设置第一个条件的原因. 设置第二个条件的原因不是很直接, 我们用 Cache 来承担本应该访问可写节点的读请求, 这样做出于 2 个有趣的系统性质: 图片在刚上传时被访问得最为频繁, 以及系统在只处理读请求或只处理写请求时表现较好, 但同时处理读写请求时效果不良 (见4.1节). 所以如果不经过 Cache, 绝大部分读请求都会被可写 Store 节点收到. 根据这个特点, 我们计划实现一个将新上传图片主动载入到 Cache 中的优化, 因为我们预期这些图片会很快被频繁访问.

3.4 Haystack Store

Store 节点的接口被特意设计得较为简单, 读请求通过给定的编号向特定的逻辑卷获取图片, 最终从某个物理的 Store 节点上读取数据. 找到图片则将其返回, 否则节点会返回一个错误.

每个 Store 节点管理若干物理卷, 每个卷会存储数百万的图片, 具体来说, 可以将物理卷简单地认为是存储在”/hay/haystack_<logical volume id>”的超大文件 (100GB). Store 节点只需要对应的逻辑卷的编号与图片所在的文件偏移量就可以快速地获取到图片, 这就是 Haystack 设计的核心要点: 不通过磁盘操作来获取一张图片的文件名, 偏移量与文件大小. 每个 Store 节点会维护其管理的每个物理卷的文件描述符, 同时还有内存中图片编号到获取图片所必需的文件系统元数据 (文件名, 偏移量与文件字节数) 的映射表.

现在我们来描述物理卷的结构以及如何得到卷在内存中的映射表. Store 节点将物理卷表示为大文件, 文件中包含一个超级区块 (superblock) 和一系列指针 (needle), 每个指针代表存储在 Haystack 中的一张图片. 图 5 展示了卷文件和指针的格式, 表 1 描述了指针包含的字段.

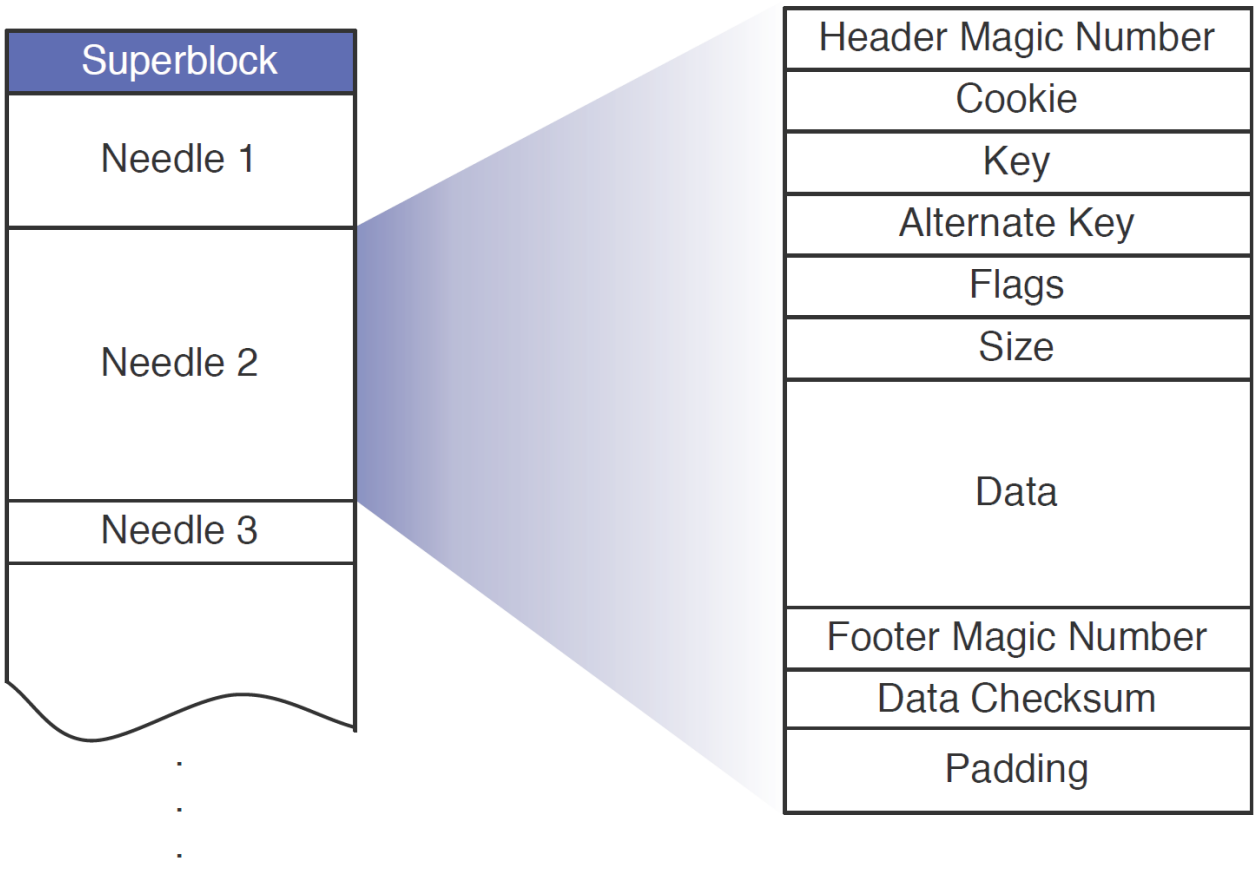

Figure 5: Layout of Haystack Store file

Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

Table 1: Explanation of fields in a needle

为了快速获取指针, Store 节点对每个卷都维护了一个内存数据结构, 此数据结构将键与副键[2]构成的二元对映射到相应的指针的标志位, 字节数与卷偏移量上. 如果发生系统崩溃, Store 节点可以在处理请求之前直接通过卷文件快速地重建映射表. 下面我们将描述 Store 节点日和在响应读, 写和删除请求 (这些是 Store 节点支持的所有操作) 时维护其卷结构与内存映射表.

3.4.1 图片读取

在请求图片时, Cache 节点会提供逻辑卷编号, 键, 副键和 cookie 给 Store 节点. 所谓 cookie 是对应图片嵌入到 URL 中的一串数字, 在图片上传时会随意生成 cookie 并且存储在 Directory 中, 其作用是消除猜测有效图片 URL 方式的攻击.

当 Store 节点接收到来自 Cache 节点的图片请求时, Store 节点首先会在内存映射表中查找相关的元数据, 如果标志位显示图片没有被删除, Store 节点就会在卷文件中找到正确的偏移位置, 并从磁盘读取整个指针 (指针的大小可以事先计算好), 然后检查 cookie 与数据完整性, 如果检查通过就向 Cache 节点返回数据.

3.4.2 图片写入

在向 Haystack 上传图片时, 网络服务器会向 Store 节点提供逻辑卷编号, 键, 副键, cookie 以及图片数据. 各个节点同时向物理卷文件中追加图片指针并且在必要时更新内存映射表. 虽然操作简单, 但仅追加的写入限制使得例如图片旋转这类编辑图片的操作变得更加复杂. 由于 Haystack 禁止覆写指针, 图片只能通过添加一个具有相同键与副键的指针来实现更新. 如果新的指针被写入到与原指针相同的逻辑卷上, 那么 Store 节点就会将新指针追加到对应的相同的物理卷上, Haystack 通过偏移量来区分重复指针, 物理卷中最新版本的指针有着最高的偏移量.

3.4.3 图片删除

删除图片的操作非常直接, 存储节点会同时在内存映射表与卷文件中设置删除标志位. 要获取已删除图片的请求首先会检查内存映射表中的删除标志位, 如果标志位已设置则返回错误. 注意被置为删除的指针占用的空间只是暂时不可用的, 稍后我们会讨论如何通过卷压缩来回收被删除的指针空间.

3.4.4 索引文件

Store 节点在重启动过程中采用了一个重要的优化——索引文件. 虽然理论上机器可以读取所有的物理卷来重新构建其内存映射表, 但这样做是非常耗费时间的, 因为需要读取大量 (TB 级别) 的数据. Store 节点可以通过索引文件快速构建内存映射表, 缩短启动时间.

Store 节点对每个卷都维护一个索引文件, 索引文件是对为在磁盘上快速定位指针的内存数据结构的检验点 (checkpoint). 索引文件的结构和卷文件类似, 包含一个超级区块和每个指针对应的索引条目, 这些条目必须和卷文件中对应指针的出现顺序相同. 图 6 展示了索引文件的结构, 表 2 说明了每个条目中应有的字段.

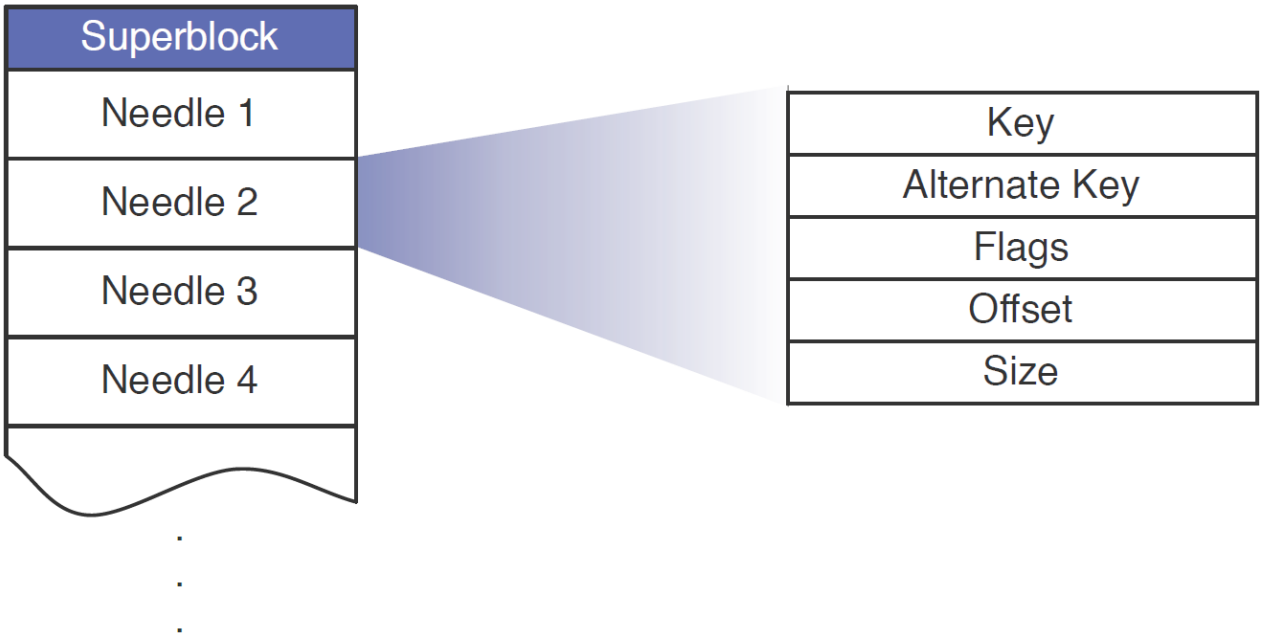

Figure 6: Layout of Haystack Index file

Field	Explanation
Key	64-bit key
Alternate key	32-bit alternate key
Flags	Currently unused
Offset	Needle offset in the Haystack Store
Size	Needle data size

Table 2: Explanation of fields in index file.

通过索引文件进行重启比仅仅读取条目并初始化内存映射表略微复杂一些, 原因是索引文件的更新是异步的, 这意味着索引文件可能代表过期的检验点. 当写入新的图片时, Store 节点先同步地在卷文件末尾追加一个指针, 另外还异步地在索引文件中增加一条记录. 当删除照片时, Store 节点会同步地设置图片指针的删除标志位, 但不更新索引文件. 这些设计决定使得写入和删除操作能够更快的返回结果, 因为省去了额外的磁盘同步写操作. 但这些操作也带来了两个我们必须关注的副作用: 有的指针可能在索引文件中没有对应的条目, 并且索引文件条目不能反映被删除的图片.

我们将没有对应索引文件条目的指针称为孤指针 (orphan). 在重启过程中, Store 节点会顺序地检查所有孤指针, 并且对每个孤指针创建其对应的索引条目, 最后将其追加在索引文件中. 注意我们可以做到快速地检测孤指针, 因为索引文件中最后一条记录必定对应于卷文件中最后一个非孤指针. 要完成重启过程, Store 节点现在只需要用索引文件来构建其内存映射表即可.

由于索引条目不能反映被删除地图片, Store 节点可能会尝试取回已经被实际删除的图片. 为了解决这个问题, 在 Store 节点读取整个指针后会特别检查指针中包含的删除标志位. 如果发现指针被标记为删除, 则 Store 节点会相应地更新自身的内存映射表并通知 Cache 节点图片没有找到.

3.4.5 文件系统

虽说作为对象存储系统, Haystack 在通常的类 UNIX 文件系统上均可工作, 但存在某些文件系统更加适合 Haystack. 特别地, Store 节点应该采用大文件随机寻址操作不会占用过多内存的文件系统. 目前 Store 节点使用的是 XFS [24], 一种基于块存储的文件系统. 使用 XFS 对于 Haystack 的好处有 2 点. 首先, XFS 中连续大文件的块表小到可以载入主存中. 其次, XFS 实现了高效的文件预分配, 可以缓解块碎片化, 防止块表过大.

使用 XFS 可以免除获取图片过程中读文件系统元数据造成的磁盘操作, 但这并不能保证 Haystack 每次读取图片时总能只执行一次磁盘操作, 当图片数据的存储位置跨区块或磁盘边界时会产生一些边界情况. Haystack 会预分配 1GB 的块空间并设置磁盘阵列条带 (stripe) 大小位 256KB, 这样在实际应用时此类边界情况就很少发生.

3.5 故障恢复

和许多其他运行在商用硬件上的大型系统 [5, 4, 9] 类似, Haystack 需要容忍各种故障: 硬盘驱动故障, 磁盘阵列控制故障, 主板故障等等. 我们使用了两种直接的方法来实现系统容灾, 一种用于检测, 另一种用于修复.

为了主动检测到故障的 Store 节点, 我们维护一个后台任务, 称为 Pitchfork, 周期性地检查各个 Store 节点的健康状况. Pitchfork 可以远程测试与 Store 节点地连接状态, 检查各个卷文件的可用性并尝试从 Store 节点读取数据. 如果 Pitchfork 发现上述健康检测在某个节点上持续地失败, 它就会将该节点上所有的逻辑卷标记为只读的, 我们再在线下人工检查失败的实际原因.

诊断出原因后, 一般就可以快速地修复故障. 但偶尔某些情况下需要执行更繁重的称为全同步 (bulk sync) 的操作, 将 Store 节点中的数据重置为来自副本的卷文件数据. 需要全同步的情况很少出现 (每个月仅有几次), 全同步操作简单但也非常慢, 主要的瓶颈在于需要全同步的数据量往往比 Store 节点网卡 (Network Interface Card, NIC) 处理速度大几个数量级, 造成平均需要几个小时来恢复, 目前我们在积极地探索解决方案.

3.6 优化

现在我们讨论一下对Haystack的成功至关重要的几个优化点.

3.6.1 压缩

压缩指回收已删除指针与重复指针 (具有相同的键与副键的指针) 占用的空间的在线操作. Store 节点会通过将指针拷贝到新的文件中来压缩卷文件, 当然拷贝过程中会跳过已删除与重复的条目. 如果在压缩过程中接收到删除操作, 则原文件和拷贝文件都要执行删除. 压缩过程进行到卷文件结尾后, 系统会阻塞住其他的更改操作, 并且原子性地替换卷文件与内存数据结构.

我们用压缩来释放已删除图片占用的空间, 用户对图片删除与图片获取的行为模式是类似的: 新图片更容易被删除, 一年内会有 25% 的图片被删除.

3.6.2 节省内存

如前所述, Store 节点维护了带标志位的内存数据结构, 但现在的系统中标志位只用于标记指针被删除. 实际可以不使用标志位而将偏移量设为零以表示被删除的图片. 其次, Store 节点不需要将 cookie 值保存在内存中, 只需从磁盘中读取指针后, 将其包含的 cookie 与所给的 cookie 值进行比较即可. 通过以上 2 种方法, Store 节点可以降低 20% 的内存占用.

目前, Haystack 每张图片会使用平均 10 字节内存, 前述我们对每张上传的图片进行缩放生成4张不同尺寸的图片. 这些图片有相同的键 (64 比特), 不同的副键 (32 比特) 和不同的数据量 (16 比特). 除了前面共 32 字节之外, Haystack 还会平均每张图片消耗 2 字节用来维护哈希表, 总计 4 张图片消耗 40 字节内存空间. 相比之下, Linux 操作系统的 xfs_inode_t 结构占 536 字节.

3.6.3 批上传

相较于小规模的随机写, 磁盘通常更擅长执行大规模的顺序写, 所以我们会尽可能地批量上传图片, 幸运的是, 很多用户会将整个相册而非单张图片上传至 Facebook, 使得我们可以对整个相册做批量上传. 在第 4 节我们将会量化展示聚合写操作带来的性能提升.

4 评估

我们将评估内容分为 4 个部分, 第一部分我们会对 Facebook 收到的图片请求进行特征描述, 第二部分与第三部分我们将分别论证 Directory 与 Cache 的有效性, 第四部分我们会通过模拟的与生产环境中的负载来分析 Store 的表现.

4.1 图片请求的特征描述

图片是被用户在 Facebook 上分享得最频繁的几种内容之一. 用户每天会上传数百万张图片, 而最近上传的图片比旧图片更热门. 图 7 展示了图片上传后经过的时间与图片热门程度的关系, 为了理解图中的曲线形状, 需要先探讨什么因素推动用户向 Facebook 发送图片请求.

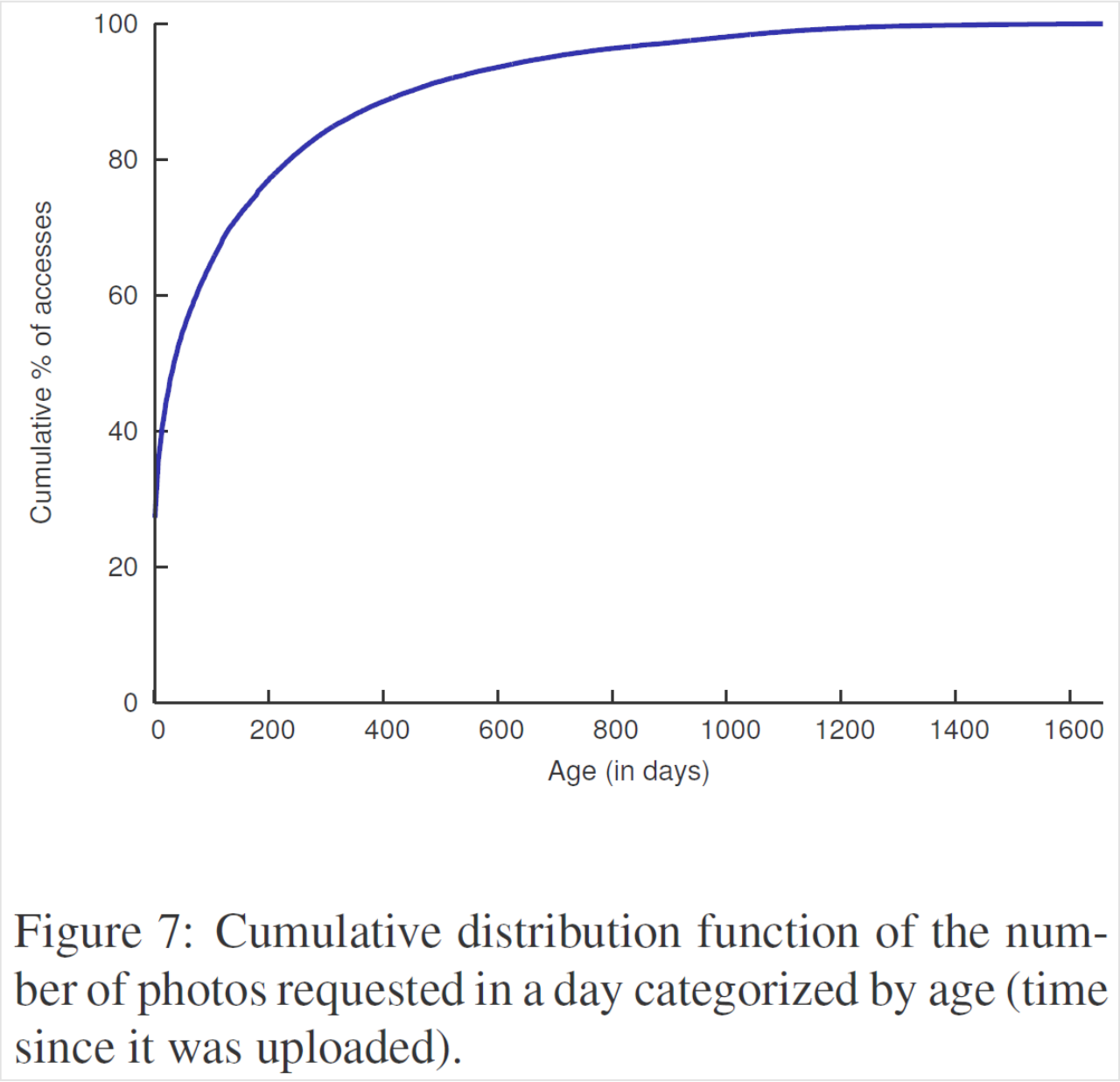

Figure 7: Cumulative distribution function of the number of photos requested in a day categorized by age (time since it was uploaded).

4.1.1 图片请求的成因

98% 的 Facebook 图片请求均由 2 个主要原因引起: 消息推送和相册. 消息推送给用户展示他们的好友最近分享的内容, 相册让用户可以浏览其好友的图片, 用户通过相册可以看最近上传的图片或浏览所有单独的相册.

图 7 中显示刚上传几天的图片对应的斜率很大. 消息推送的大部分流量都用于最近上传的图片, 而约 2 天后消息推送的流量会显著减小, 因为默认的消息推送界面不会显示旧图片. 上图中有 2 个需要强调的关键顶, 首先图片热度的迅速减小说明同时在 CDN 和 Cache 节点对热门内容进行缓存会非常有效, 其次曲线表现出长尾现象说明有相当一部分请求不能直接被缓存处理.

4.1.2 流量

表 3 展示了 Facebook 图片服务的流量. 被写入 Haystack 的图片数量是上传图片数量的 12 倍, 因为我们的应用会将每张图片缩放为 4 种不同的尺寸并将每个尺寸的图片存储在 3 个不同的位置. 表格显示 Haystack 约 10% 的图片请求是通过 CDN 来响应的, 需要注意用户浏览的图片绝大部分是小尺寸图, 这

带来的迅速的性能下降体现我们缩减元数据开销的必要性. 除此以外, 由于大尺寸图片多显示在相册用并且可以通过预加载来隐藏延迟, 但小尺寸图片多用于消息推送, 所以加载小尺寸图片对网络延迟是非常敏感的.

Operations	**Daily Counts**
Photos Uploaded	~120 Million
Haystack Photos Written	~1.44 Billion
Photos Viewed	80-100 Billion
[*Thumbnails*]	10.2 %
[*Small*]	84.4 %
[*Medium*]	0.2 %
[*Large*]	5.2 %
Haystack Photos Read	10 Billion

Table 3: Volume of daily photo traffic.

4.2 Haystack Directory

Haystack Directory 负责了对 Store 节点读写请求的负载均衡. 图 8 体现 Directory 直接的哈希策略对均衡分配读写请求如预期般有效. 图像显示了 9 台部署于生产环境中的 Store 节点在相同时刻接执行的写操作数量, 每个节点都存储了不同的图片. 由图可见 9 条区间几乎不可区分, 说明 Directory 对写操作的负载均衡实现得很好, Store 节点的读操作也有同样均衡的表现.

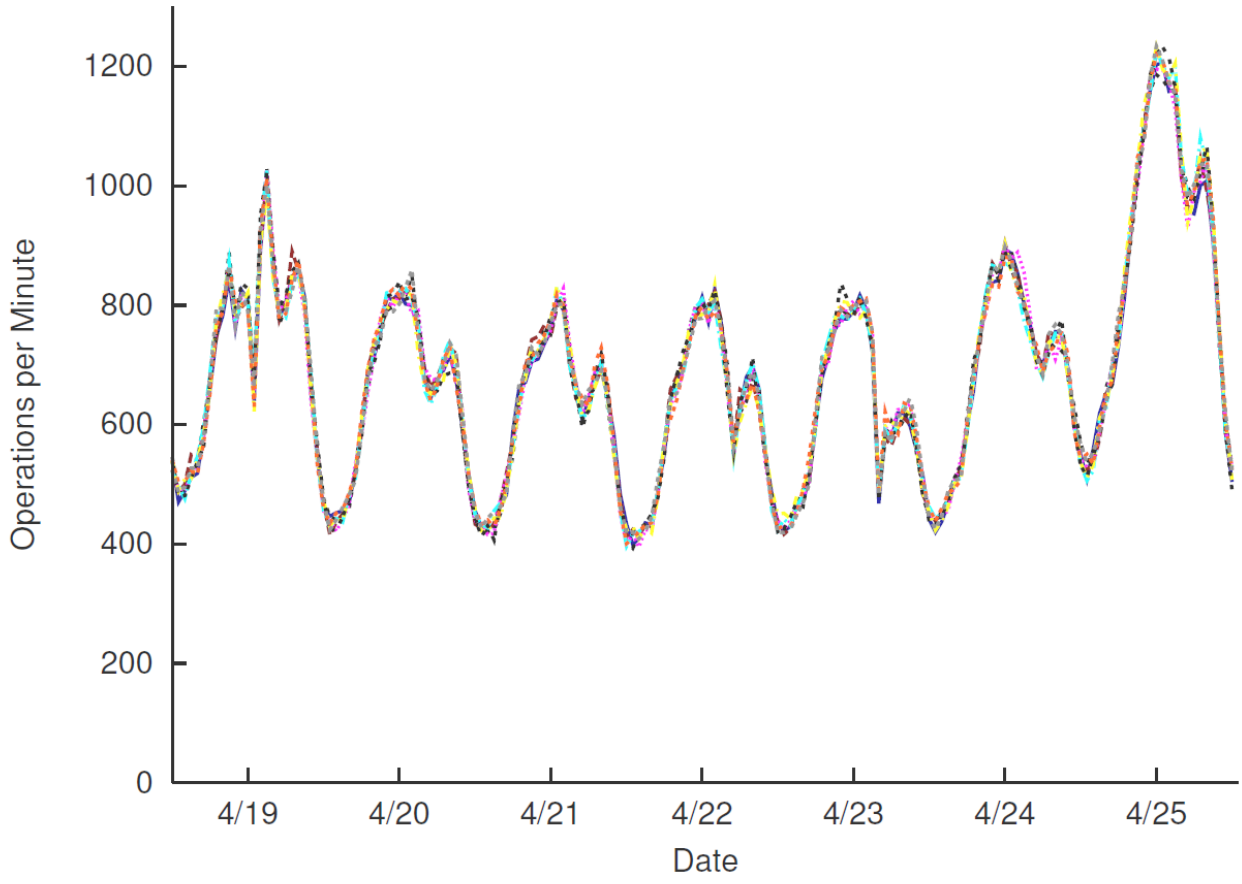

Figure 8: Volume of multi-write operations sent to 9 different write-enabled Haystack Store machines. The graph has 9 different lines that closely overlap each other.

4.3 Haystack Cache

图 9 显示了 Haystack Cache 的缓存命中率, 前述 Cache 只会缓存存储在可写节点上的图片, 此类图像是相对较新上传的, 这解释了近 80% 的高缓存命中率. 由于可写节点上的资源同时也会接收到大量的读请求, Cache 可以有效地降低节点接受读操作的压力.

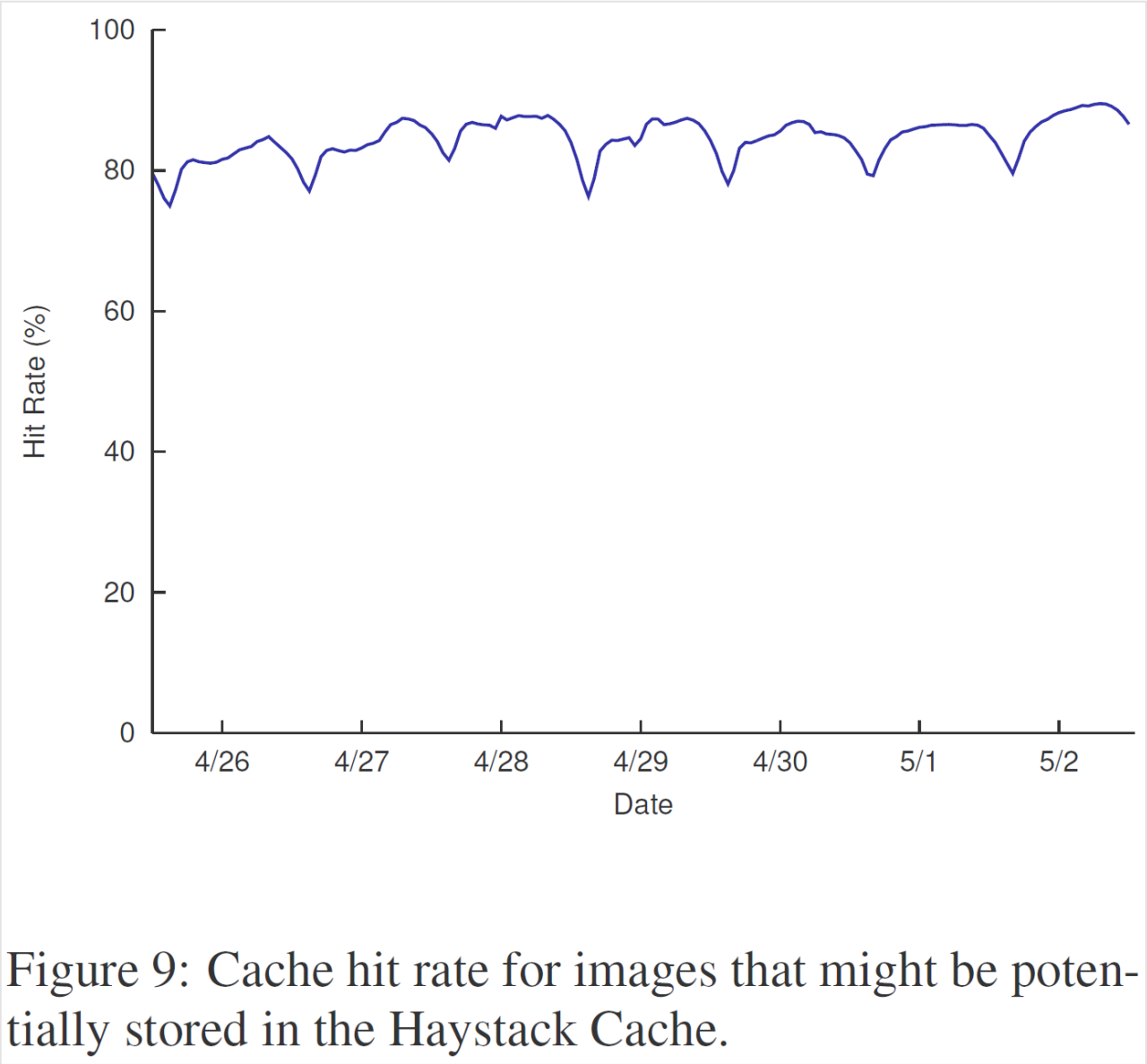

Figure 9: Cache hit rate for images that might be potentially stored in the Haystack Cache.

4.4 Haystack Store

前述 Haystack 的设计目标针对的是图片的长尾请求, 并在接近随机读的场景下保持高吞吐量与低延迟, 下面我们给出 Store 节点在模拟与生产环境负载下的表现.

4.4.1 实验设计

我们将 Store 节点部署在商用存储设备上, 典型的硬件配置为 1 个 2U (3.5 英寸) 机箱, 2 个超线程 4 核 Intel Xeon CPU, 48GB 内存, 1 个带 256–512MB NVRAM 和 12 个 1TB SATA 驱动的硬件 RAID 控制器.

每个机箱能够提供约 9TB 存储容量, 配置为通过硬件 RAID 控制器管理的 RAID-6 分区, RAID-6 能够在保持存储开销较低的前提下做出适当的数据冗余以及良好的读性能. 控制器的 NVRAM 写回缓存弥补了 RAID-6 写性能的不足. 根据经验在 Store 节点上做图片缓存效率较低, 所以我们将 NVRAM 完全留作写入用途. 另外我们出于在系统崩溃或掉电时能保持数据一致性的目的禁用了磁盘缓存.

4.4.2 测试表现

我们使用了 2 个基准测试来衡量 Store 节点的表现: Randomio [22] 和 Haystress. Randomio 是一个开源的多线程磁盘 I/O 程序, 我们用其来衡量存储设备的容量. 程序会采用直接 I/O 的方式发起 64KB 的读操作, 进行扇区对齐的请求, 最终报告可持续的最大吞吐量. 我们使用 Randomio 作为测试读吞吐量的基准, 用于和其他测试结果比较.

Haystress 是我们使用多种模拟负载来评估 Store 节点的一个定制化多线程程序, 它会通过 HTTP 请求联络 Store 节点 (正如 Cache 节点一样), 并测出 Store 节点可以维持的最大读写吞吐量. Haystress 会对大量测试图片发出随机读请求, 目的是减小节点缓存的影响, 换言之几乎所有的读请求都需要一次磁盘操作. 在本文中, 我使用 7 种 Haystress 负载来评测 Store 节点.

表 4 展示了 Store 节点在我们的测试标准下能够位置的读写吞吐量与对应的延迟, 负载 A 是对有 201 个卷的 Store 节点进行 64KB 图片的随机读操作, 结果显示 Haystack 可以发挥出设备 85% 的吞吐量, 并且延迟仅仅增加了 17%.

		Reads			Writes		
Benchmark	**[Config # Operations]**	**Throughput (in images/s)**	**Latency (in ms)**		**Throughput (in images/s)**	**Latency (in ms)**	
			Avg.	**Std. dev.**		**Avg.**	**Std. dev.**
Random IO	[Only Reads]	902.3	33.2	26.8	–	–	–
Haystress	[**A** # Only Reads]	770.6	38.9	30.2	–	–	–
Haystress	[**B** # Only Reads]	877.8	34.2	28.1	–	–	–
Haystress	[**C** # Only Multi-Writes]	–	–	–	6099.4	4.9	16.0
Haystress	[**D** # Only Multi-Writes]	–	–	–	7899.7	15.2	15.3
Haystress	[**E** # Only Multi-Writes]	–	–	–	10843.8	43.9	16.3
Haystress	[**F** # Reads & Multi-Writes]	718.1	41.6	31.6	232.0	11.9	6.3
Haystress	[**G** # Reads & Multi-Writes]	692.8	42.8	33.7	440.0	11.9	6.9

Table 4: Throughput and latency of read and multi-write operations on synthetic workloads. Config **B** uses a mix of 8KB and 64KB images. Remaining configs use 64KB images.

我们将 Store 节点增加的开销归结为 4 个因素: (a) Store 节点不能直接访问磁盘, 而是工作在已有的文件系统上; (b) 由于需要读取整个指针, 磁盘读操作实际超过 64KB; (c) 存储的图片数据可能没有与 RAID-6 设备条带对齐, 所以一小部分图片要从超过一块磁盘上读取; (d) Haystack 服务器给 CPU 带来额外开销 (索引访问, 校验和检查等等).

在负载 B 中, 我们仍然测试只读负载, 但是变更了 70% 的读请求, 令其读取更小尺寸的图片 (8KB 而非 64KB). 实际环境中, 我们发现大部分的请求并非获取大型图片 (因为会显示在相册中), 而多为缩略图与档案图片.

负载 C, D 和 E 展示了 Store 节点的写吞吐量. 前述 Haystack 可以将写操作批量执行, 负载 C, D 和 E 分别将 1 个, 4 个和 16 个写操作批量作为单个写操作. 表格显示分摊至每张图片上的固定开销在批大小为 4 与 16 时分别降低了 30% 与 78%, 每张图片的延迟也如预期地减少.

最后, 我们检测了同时存在读写操作时的表现. 负载 F 包含了 98% 的读操作和 2% 的写操作, 负载 G 则包含 96% 的读操作和 4% 的写操作, 每个写操作批量写入 16 张图片, 这些读写操作的比例反映的是生

产环境中的实际比例. 表格显示即便在写操作存在的情况下, Store 节点仍能保持较高的读吞吐量.

4.4.3 生产负载

此节测试了 Store 在生产环境的机器上的表现. 如第 3 节所述 Store 节点分为 2 种: 可写与只读, 可写节点能处理读请求与写请求, 只读节点只能处理读请求. 由于这 2 类节点的流量特征明显不同, 我们会对每类节点都分别进行分析, 节点的硬件配置均相同.

在秒粒度上观测, Store 节点会产生很多读写操作数量的尖峰, 为了在出现尖峰时仍能保证较低的延迟, 我们保守地分配了较多的可写节点, 使得其平均负载较低.

图 10 展示了可写与只读 Store 节点上不同类型操作的频率, 可以注意到星期日与星期一会出现图片上传的峰值, 在一周的其他时间则会缓慢地下降, 直至星期四到星期六趋于平缓, 整体的流量每天会增长 0.2% 到 0.5%.

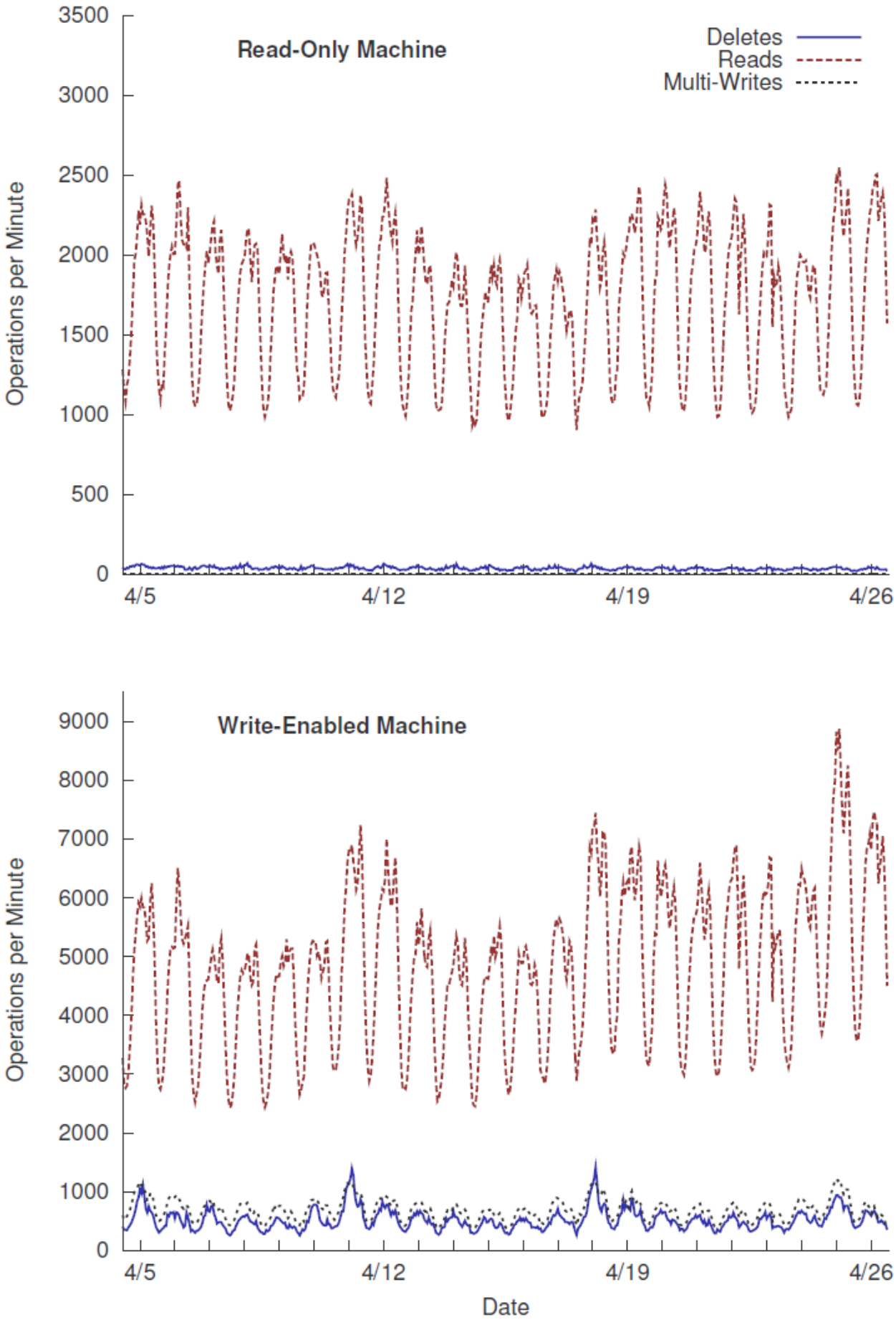

Figure 10: Rate of different operations on two Haystack Store machines: One read-only and the other write-

enabled.

如第 3 节所述, 为了均摊写操作地固定开销, 生产环境中的 Store 节点总是批量写入的. 查找一组图片是非常直接的, 因为 Haystack 会对图片存储 4 个尺寸不同的版本, 另一个常见的用户行为是将多张照片一次性上传到图片相册中. 基于以上 2 点, 在可写节点上单次批量写操作写入的图片数量平均为 9.27 张.

在 4.1.2 节中解释说明了新上传的图片的读取与删除操作频率较高, 并随着时间降低. 这种现象也显示在图 10 中, 体现为可写节点接收到远多于制度节点的请求 (即便很多读操作已经被 Cache 处理).

还有另外值得注意的趋势: 随着更多的数据被写入可写节点, 图片的数量增加导致接收到的读请求也会增多.

图 11 展示了与图 10 相同的 2 个节点在相同时间段内读与批量写操作的延迟.

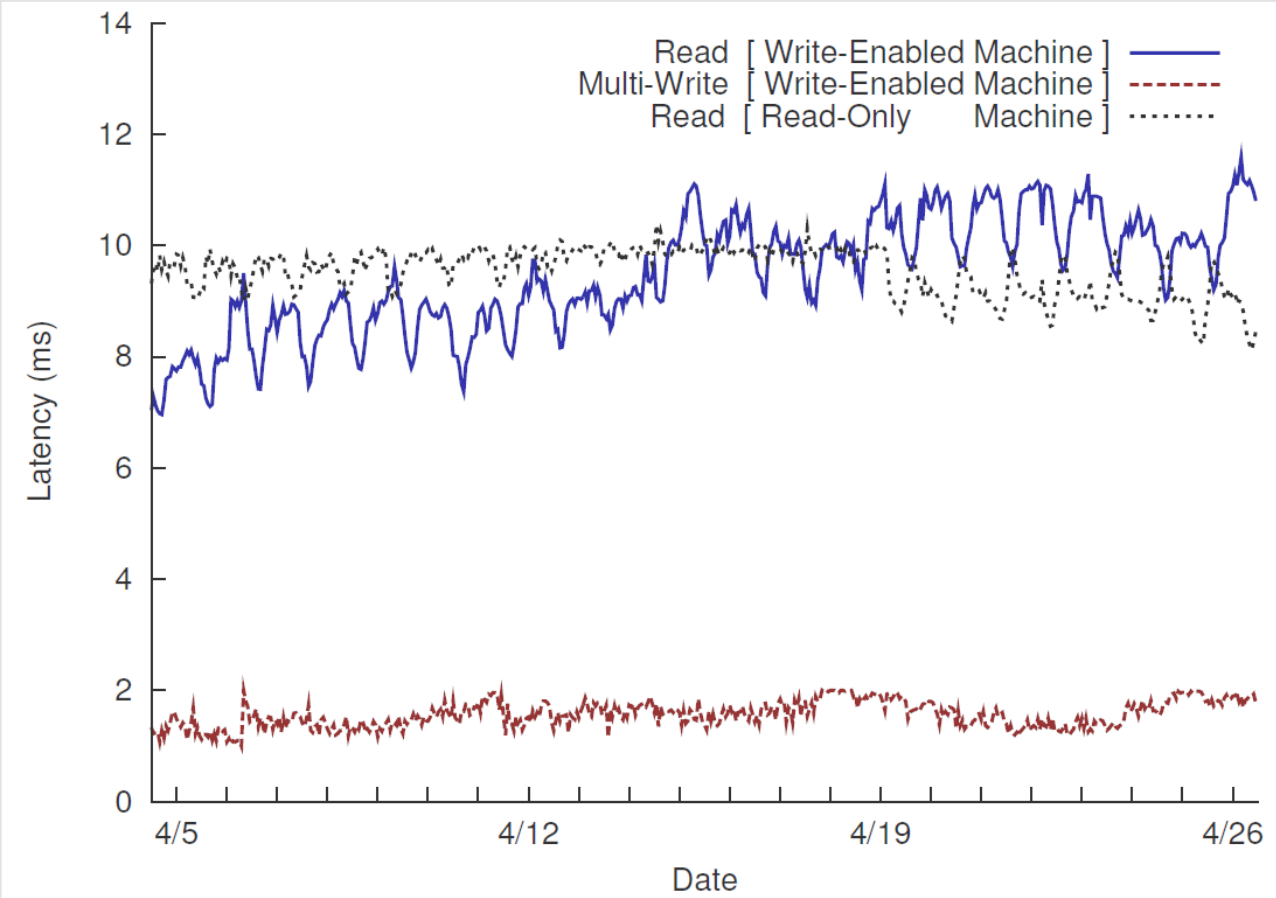

Figure 11: Average latency of Read and Multi-write operations on the two Haystack Store machines in Figure 10 over the same 3 week period.

批量写操作的延迟很低(在 1 毫秒到 2 毫秒之间), 并且在流量大幅震荡时仍然保持稳定, Haystack 节点有 NVRAM 驱动的 RAID 控制器可以实现写缓存. 如第 3 节所述, NVRAM 可以使得我们异步地写入指针, 最后在批量写完成后用一个文件同步 (fsync) 指令刷新卷文件, 所以批量写对应的曲线平坦而且稳定.

只读节点的读延迟也非常稳定, 即便流量变化幅度很大 (最高在 3 周内增长为 3 倍). 对于可写节点而言, 读操作主要被 3 个关键因素影响. 首先, 随着存储在节点上的图片数量的增长, 发往此节点的读流量也会增加 (与图 10 中每周流量相比). 其次, 可写节点的图片会被 Cache 节点缓存, 而只读节点则不会[3], 这意味着对只读节点做缓冲区缓存更有效. 最后, 最近上传的图片往往会被立刻读取, 因为 Facebook 会重点显示近期内容, 这种对可写节点的读请求总会命中缓冲区缓存, 提高其命中率. 图像中曲线的形状由上述 3 个因素共同决定.

Store 节点上的 CPU 负载很低, 92%–96% 的时间 CPU 处于空闲状态.

5 相关工作

就我们所知, Haystack 的目标是为处理大型社交网络中出现的长尾请求提出新的设计要点.

文件系统 Haystack 采用 Rosenblum 与 Ousterhout 设计的日志结构文件系统 [23], 此结构考虑到大部分的写请求可以被缓存处理, 所以主要的优化目标是读吞吐量. 虽然部分测试 [3] 和模拟 [6] 显示日志结构系统在本地文件系统中没有完全发挥出潜力, 其核心思想和 Haystack 关系非常密切. 图片以追加方式写入 Store 节点的卷文件中, 而 Cache 节点则为可写节点承担了主要的读请求, 防止最近上传数据的读流量其性能. 关键的不同在于 (a) Haystack Store 节点将数据以一种在自身转为只读节点后能够快速分发数据的存储, 并且 (b) 旧数据的访问频率随着时间降低.

一些工作 [8, 19, 28] 提出了更高效地管理小文件与元数据的所感方法, 主流方向是更智能地将相关文件与元数据文组. Haystack 与其不同在于将元数据载入内存中, 并且用户经常批量上传图片.

对象存储 Weil 等人 [26, 27] 提出了能够可扩展地管理元数据的 PB 级对象存储 Ceph 系统. Ceph 更进一步地将逻辑单元到物理单元的映射解耦, 实现方法是引入引入生成函数取代显式映射. 在 Haystack 中实现这个功能需要后续工作. Hendricks 等人 [13] 观察到传统的元数据预获取算法在对象存储系统中效果不佳, 原因是文件系统中能够通过目录直接表示文件之间的关联性, 而对象存储中通过编号来表示相关缺乏语义上的分组. 他们的解决方式是将对象间的关系表示加入到对象编号中, 这个想法是 Haystac k中不需要的, 因为 Facebook 已经显式地将这些语义关系存储成了社交关系网络. 在 Leung 等人的 Spyglass [15] 中提出了一种在大型存储系统中利用元数据进行快速, 可扩展搜索的设计. Manber 和 Wu 也在 GLIMPSE [17] 中提出了一种搜索整个文件系统的方法. Patil [20] 等人在 GIGA+ 中使用了一种复杂的算法来管理每个目录下数十亿个文件对应的元数据. 我们采用的方法比许多现存的方法都更加简单, 因为 Haystack 不需要提供搜索关键字与 UNIX 文件系统的相关信息.

分布式文件系统 Haystack 中逻辑卷的概念与 Lee 和 Thekkath [14] 在 Petal 中提出的虚拟磁盘概念非常类似. Boxwood 项目 [16] 尝试使用高层数据结构作为存储系统的基础. 虽然 B 树这类抽象需要更加复杂的算法, 其对 Haystack 特意设计的轻量级接口和语义是没太大影响的. 同样地, Sinfonia [1] 的微事务和 PNUTS [5] 的数据库功能提供了超过 Haystack 需要的功能点与数据保证. Chemawat 等人 [9] 为主要包含追加写操作和大规模顺序读操作的负载设计了谷歌文件系统. BigTable [4] 提供了结构化数据存储系统, 并且为很多谷歌的应用提供了类似数据库的功能. 这么多功能点在图片存储中是否有用暂不明确.

6 结论

本文介绍了 Haystack, 为 Facebook 图片应用设计的对象存储系统, 我们用 Haystack 来应对在大型社交网络中分享图片时出现的长尾问题. 本文的关键点在于在访问元数据时避免磁盘操作. Haystack 以比使用 NAS 设备的传统方法更小的开销与更高的吞吐量为图片存储提供了一个既容错又简单的解决方案. 除此以外, Haystack 可扩展性更强, 这对用户每周会上传上亿图片的系统是非常必要的.

参考文献

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pages 159—174, New York, NY, USA, 2007. ACM.

[2] Akamai. http://www.akamai.com/.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In Proc. 13th SOSP, pages 198—212, 1991.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2):1—26, 2008.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow., 1(2):1277—1288, 2008.

[6] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In Proceedings of the First Symposium on Operating Systems Design and Implementation, pages 267—280, Nov 1994.

[7] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object storage: the future building block for storage systems. In LGDI '05: Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology, pages 119—123, Washington, DC, USA, 2005. IEEE Computer Society.

[8] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference, pages 1—1, Berkeley, CA, USA, 1997. USENIX Association.

[9] S. Ghemawat, H. Gobioff, and S.–T. Leung. The google file system. In Proc. 19th SOSP, pages 29—43. ACM Press, 2003.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost–effective, high–bandwidth storage architecture. SIGOPS Oper. Syst. Rev., 32(5):92—103, 1998.

[11] The hadoop project. http://hadoop.apache.org/.

[12] S. He and D. Feng. Design of an object–based storage device based on i/o processor. SIGOPS Oper. Syst. Rev., 42(6):30—35, 2008.

[13] J. Hendricks, R. R. Sambasivan, S. Sinnamohideen, and G. R. Ganger. Improving small file performance in object–based storage. Technical Report 06–104, Parallel Data Laboratory, Carnegie Mellon University, 2006.

[14] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In ASPLOS–VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems, pages 84—92, New York, NY, USA, 1996. ACM.

[15] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: fast, scalable metadata search for large–scale storage systems. In FAST '09: Proccedings of the 7th conference on File and storage technologies, pages 153—166, Berkeley, CA, USA, 2009. USENIX Association.

[16] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, pages 8—8, Berkeley, CA, USA, 2004. USENIX Association.

[17] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. In WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, pages 4—4, Berkeley, CA, USA, 1994. USENIX Association.

[18] memcache. http://memcached.org/.

[19] S. J. Mullender and A. S. Tanenbaum. Immediate files. Softw. Pract. Exper., 14(4):365—368, 1984.

[20] S. V. Patil, G. A. Gibson, S. Lang, and M. Polte. Giga+: scalable directories for shared file systems. In PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage, pages 26—29, New York, NY, USA, 2007. ACM.

[21] Posix. http://standards.ieee.org/regauth/posix/.

[22] Randomio. http://members.optusnet.com.au/clausen/ideas/randomio/index.html.

[23] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log–structured file system. ACM Trans. Comput. Syst., 10(1):26—52, 1992.

[24] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual

Technical Conference, pages 1—1, Berkeley, CA, USA, 1996. USENIX Association.

[25] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. Obfs: A file system for object-based storage devices. In In Proceedings of the 21st IEEE / 12TH NASA Goddard Conference on Mass Storage Systems and Technologies, pages 283—300, 2004.

[26] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation, pages 307—320, Berkeley, CA, USA, 2006. USENIX Association.

[27] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, page 4, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Z. Zhang and K. Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. SIGOPS Oper. Syst. Rev., 41(3):175—187, 2007.

注1: “可用”包括磁盘阵列 (Redundant Arrays of Independent Disks, RAID), 副本与文件系统占用的空间.

注2: 由于历史遗留问题, 图片编号相当于键而图片类型相当于副键, 在上传过程中, 网络服务器将图片缩放为四种不同的大小并将其存在不同的指针处, 但它们的键是相同的. 这些图片的区别在于副键不同, 分别为 “n”, “a”, “s” 或 “t”.

注3: 这两种情况下, 由 CDN 而来的流量都会被 CDN 缓存, 而不会被 Cache 节点缓存.

distributed system

‹ 全序关系广播与共识算法