Facebook 对 Memcache 伸缩性的增强

概要：Memcached 是一个知名的，简单的，全内存的缓存方案。这篇文章描述了facebook是如何使用memcached来构建和扩展一个分布式的key-value存储来为世界上最大的社交网站服务的。我们的系统每秒要处理几十亿的请求，同时存储了几万亿的数据项，可以给全世界超过10亿的用户提供丰富体验。

1 介绍

近些年SNS网络大行其道，这对网站基础建设提出了巨大的挑战。每天有亿万的用户在使用这些网络服务，巨大的计算、网络和I/O资源的需求使传统的web架构不堪重 负。SNS网站的基础架构需要满足：1、近乎实时的交流；2、即时聚合不同来源的内容；3、访问和更新非常热门的共享内容；4、每秒处理几百万的用户请求。

我们将描述我们是如何改进memcached[14]的开源版本，并且用它作为组件来构建用于世界上最大的社会化网络的分布式key-value存储的。我们会讨论从单集群服务器扩展成地理上分布式的多集群的历程。据我们所知，这个系统是世界上已安装的规模最大的memcached系统，每秒可以处理几十亿的请求，存储数以万亿的数据项。

本文是关于认识分布式key-value存储的灵活性和实用性的系列文章[1, 2, 5, 6, 12, 14, 34, 36]的最后一篇。本文关注于memcached，这是一个全内存哈希表的开源实现，它以较低的开销提供了对共享存储的低迟延访问。有了这些特性我们可以构建数据密集的功能，否则是不可能的。例如，如果一个页面请求会产生数以百计的数据库请求，那么这样的功能只能停止在原型阶段，因为实现起来会太慢，代价也太高。然而，在我们的应用里，web页面通常都会从memcached服务器获取数以千计的key-value对。

我们的目标之一，是展现部署在不同尺度（系统）上的重要主题。虽然在所有尺度上是很重要的品质，如性能，效率，容错性和一致性，我们的经验表明，在特定大小的一些素质要求比别人更多的努力来实现。举例来说，保持数据的一致性，如果复制的内容是小量的，可以更容易在小尺度的网络上实现，相比较大的网络往往只是复制必要的内容。此外，找到一个最佳的通信调度的重要性增加的数量增加服务器和网络工作成为瓶颈。

本文包括四个主要贡献：（1）我们描述了Facebook的基于memcach架构的演化。（2）我们确定memcached的提高性能和增加内存效率的改进。（3）我们简明扼要地讲述提高我们的经营能力我们的系统规模的机制。（4）我们对生产工作负载赋予了特色（译者加：对工作负载进行了分类？）。

2综述

以下特点大大影响了我们的设计。第一，用户阅读的内容比他们创建的要多一个数量级，这种行为（读写的特点）所产生工作负载，显然让缓存可以发挥很大的优势。第二，我们是从多个来源读取数据的，比如MySQL数据库、HDFS设备和后台服务，这种多样性要求一个灵活的缓存策略，能够从各个独立的源中储存数据。

MemCached提供了一组简单的操作（set、get和delete），使它在一个大规模的分布式系统中成为注目的基础组件。开源版本提供了单机内存哈希表，在本文中，我们从这个开源版本开始，讨论我们是怎么使用这个基础组件，使它变得更有效，并用它来建一个可以处理每秒数十亿请求的分布式的键-值储存系统。接下来，我们用“memcached”来指代它的源码或者它运行的二进制实例，用“memcache”来指代由每个实例构成的分布式系统。

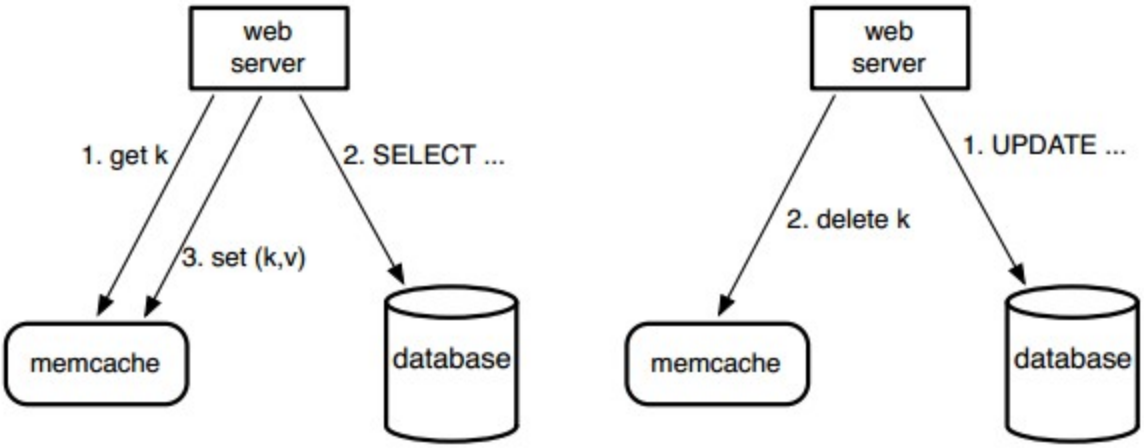

图1：Memcache作为填补需求的旁路缓存系统。左半图说明了WEB服务器读取缓存时命中失败的读取路径，右半图说明其写路径。

查询缓存：我们依赖于memcache来减轻读取数据库的负担。特别的，我们使用memcache作为填补需求的旁路缓存系统，如图1。当一个Web服务器需要数据时，首先通过一个字符串的键在memcache中请求，如果没有找到，它会从数据库或者从后台服务中检索，再使用该键把结果存回memcache中。对于写的请求，Web服务器发送SQL语句到数据库，接着发送删除请求到memcache，使旧的缓存数据失效。因为删除是幂等运算，所以我们使用删除缓存的方式，而不是更新缓存。

在应对MySQL数据库繁重的查询通信的众多方法中，我们选择了memcache，在有限的资源与时间限制下，这是最好的选择。此外，缓存层与持久层分离，让我们可以在工作负载发生变化时快速地调整。

通用缓存：我们同样让memcache成为一个更加通用的键-值储存系统。比如说，工程师们使用memcache保存复杂的机器学习算法的中间结果，这些结果能被很多其它应用程序所使用。它只需要我们付出很少的努力，就可以让新增的服务利用现有的正在使用的基础设施，而无需调整、优化、调配和维护大型的服务器群。

正如memcached没有提供服务器到服务器的协同，它仅仅是运行在单机上的一个内存哈希表。接下来我们描述我们是如何基于memcached构建一个分布式键值储存系统，以胜任在Facebook的工作负载下的操作。

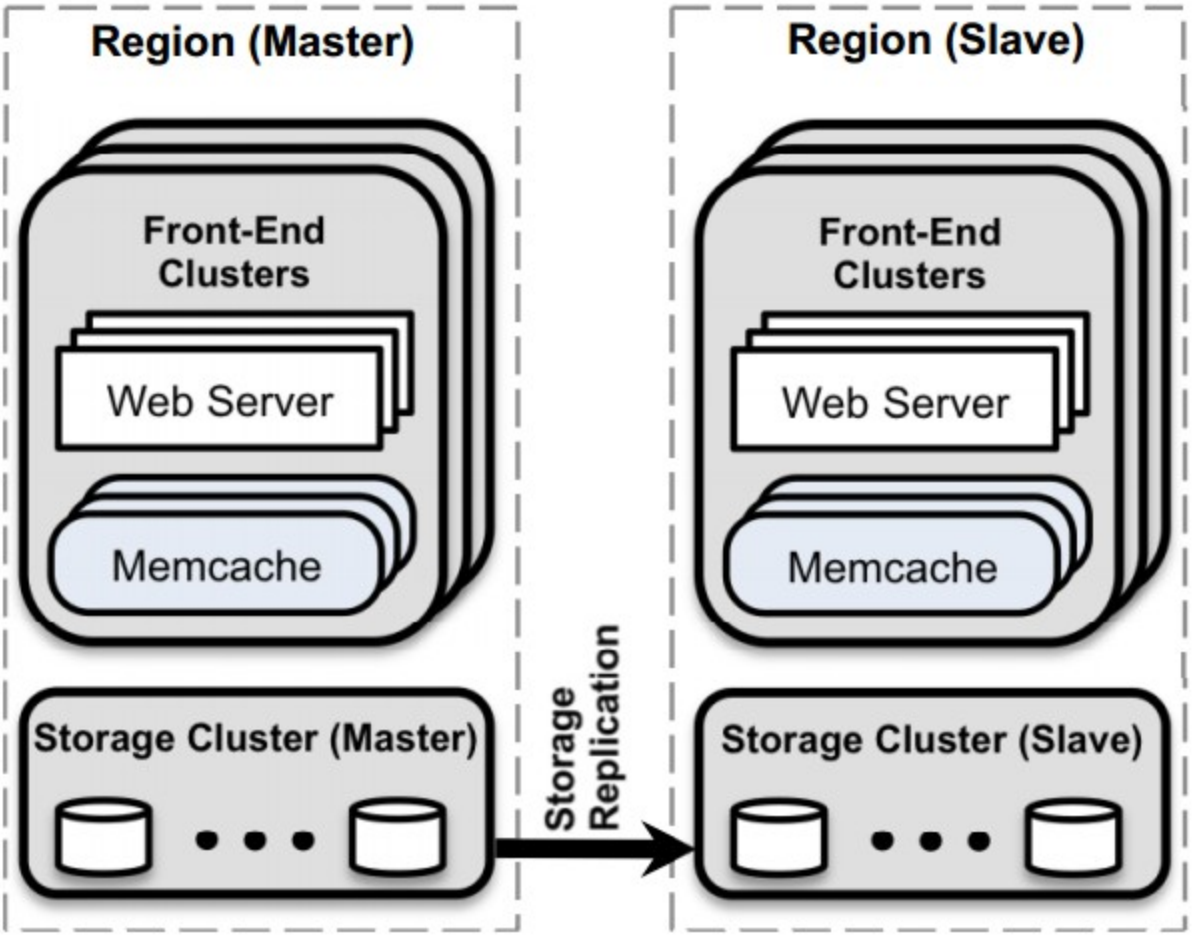

图2：整体架构

论文的结构主要描述了在三种不同的规模下出现的问题。当我们拥有第一个服务器集群时，频繁的读负载和广泛的输出是我们最大的担心。当有必要扩展到多个前端集群时，我们解决了集群间的数据备份问题。最后，我们描述了一种机制，这种机制让我们可以在全世界伸展集群的同时提供平滑的用户体验。不论在什么尺度上，容错性和操作复杂性总是很重要的。我们展示了重要的数据参考，这些数据指引我们做出了最终的设计决定，读者如需获得更多细节性的分析，请参看Atikoglu et al.[8]的工作。提纲挈领的解释参看图2，这是最终的架构，我们将并置集群组织起来，形成一个群体（region），指定一个主群体（master），由主群体提供数据流让非主群体保持数据同步。

在系统的发展中，我们将这两个重大的设计目标放在首位：

1. 只有已经对用户或者我们的运维产生影响的问题，才值得改变。我们极少考虑范围有限的优化。

2. 对陈旧数据的瞬态读取，其概率和响应度类似，都将作为参数来调整。我们会暴露轻度陈旧的数据以便后台存储和高强度负载绝缘。

3 集群之中： 延迟和负载

现在考虑集群中数以千计的服务器所带来的挑战。在这种规模之下，我们着眼于减少获取缓存时的负载，以及缓存不中时数据库的负载。

3.1 减少延迟

不论缓存是否命中，memcache的响应时间都是影响总响应时间的重要因素。单个的网页请求一般包含数百个memcache读请求。如一个较火的页面平均需要从memcache中获取521个不同的资源。

为了减少数据库等的负担，我们准备了缓存集群，每个集群都由数百台memcache服务器组成。资源个体经hash后存于不同的memcache服务器中。因此，web服务器必须请求多台memcache服务器，才能满足用户的请求。

由此导致在很短的时间里每个web服务器都要和所有的memcache服务器沟通。这种所有对所有的连接模式会导致潮涌堵塞（incast congestion）或者某台服务器不幸成为瓶颈。实时备份可以缓解这种状况，但一般又会引起巨大的内存浪费。（译者：为何？）

我们减少延迟的方法主要集中在memcache客户端，每一个web服务器都会运行memcache客户端。这个客户端提供一系列功能，包括：串行化、压缩、请求路由、错误处理以及请求批处理。客户端维护着一个对所以可获得的服务器的映射，对这个映射表的更新需要通过一个辅助的配置系统。

并行请求和批处理：我们构建web应用代码，目的是最小化对于页面请求回应所必要的网络往返数。我们构建了有向无环图（DAG）用来表示数据间的依赖。web服务器使用DAG来最大化可以并发读取的项目数。平均来说，这些批量请求对于每个请求包含24个主键。

客户端-服务器通信：memcached服务器不会直接通信。如果适当，我们将系统的复杂度嵌入无状态的客户端，而不是memcached服务器。这极大地简化了memcached，使我们专注于针对更有限的用例提供高性能。保持客户端的无状态使得我们可以快速迭代开发，同时也简化了部署流程。客户端的逻辑可以提供为两种组件：可以嵌入应用的一个库，或者做为一个名为mcrouter的独立的代理程序。这个代理提供memcached服务器的借口，对不同服务器之间的请求/回复进行路由。

客户端使用UDP和TCP协议与memcached服务器通讯。我们依赖UDP来使请求的延迟和开销缩减。因为UDP是无连接的，web服务器中的每个线程都被允许直接与memcached服务器通信，通过mcrouter，不需要创建与维护连接因而减少了开销。UDP实现了检测出丢失的或失序接收（通过序列号）的包，并在客户端将它们作为异常处理。它没有提供任何试图恢复的机制。在我们的基础架构中，我们发现这个决定很实际。在峰值负载条件下，memcache客户端观察到0.25%的请求会被丢弃。其中大约80%是由于延迟或丢失包，其余的是由于失序的交付。客户端将异常作为缓存不命中处理，但是web服务器在查询出数据以后，会跳过插入条目到memcached，以便避免对可能超载的网络会服务器增添额外的负载。

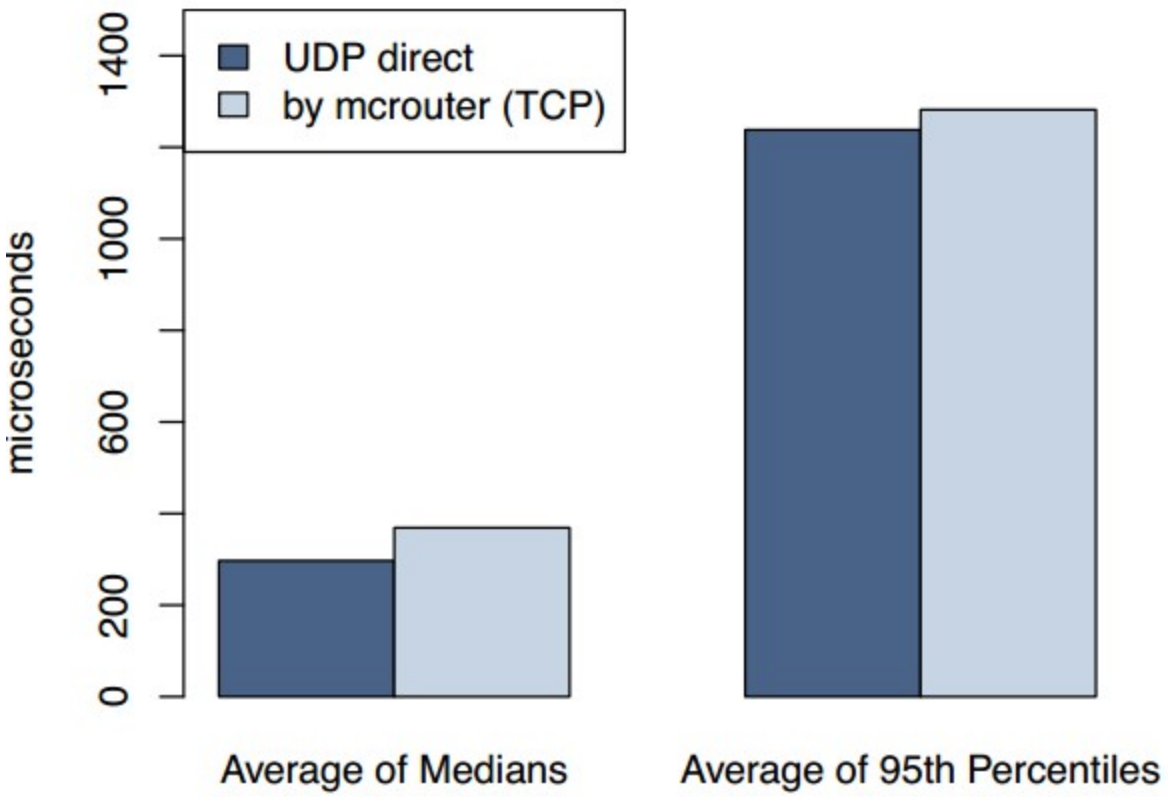

图 3: 经过mcrouter以后 UDP, TCP得到的延迟

为了可靠性，客户端通过同一个web服务器上运行的mcrouter实例，在TCP协议之上运行set与delete操作。对我们需要确认状态变化（更新和删除）的操作，TCP避免了UDP实现中增加重试机制的必要。

Web服务器依赖很高程度的并行性与超量提交来获得高吞吐量。如果不采用由mcrouter合并的某种形式的连接，打开TCP连接需要的大量内存将使得在每个web线程与memcached服务器之间打开连接变得尤其代价昂贵。通过

减少高吞吐量TCP连接对网络，CPU和内存资源的需求，合并这些连接的方式增强了服务器的效率。图3显示了生产环境中web服务器在平均的，中级的，以及百分之95的条件下，在UDP和通过经由TCP的mcrouter机制下获得关键字的延迟。在所有情形，与这些平均值的标准差小于1%。正如数据所示，依赖UDP能有20%的延迟缩减来对请求提供服务。

========== ==========

1 百分之95的页面抓取的是1,740项目。
2 百分之95情形是每个请求有95个关键字。

Incast拥塞：memcache客户端实现流量控制机制限制incast拥塞。当一个客户端请求大量的主键时，如果所有应答同时达到，那么这些应答可以淹没一些组件，例如：机架和集群交换机。因此客户端使用滑动窗口机制[11]来控制未处理请求的数量。当客户端收到一个应答的时候，那么下一个请求就可以发送了。与TCP的拥塞控制类似，滑动窗口的大小随着成功的请求缓慢的增长，当一个请求没有应答的时候就缩小。这个窗口应用于所有的memcache请求，而不关心目的地址；然而TCP窗口仅仅应用于单独的数据流。

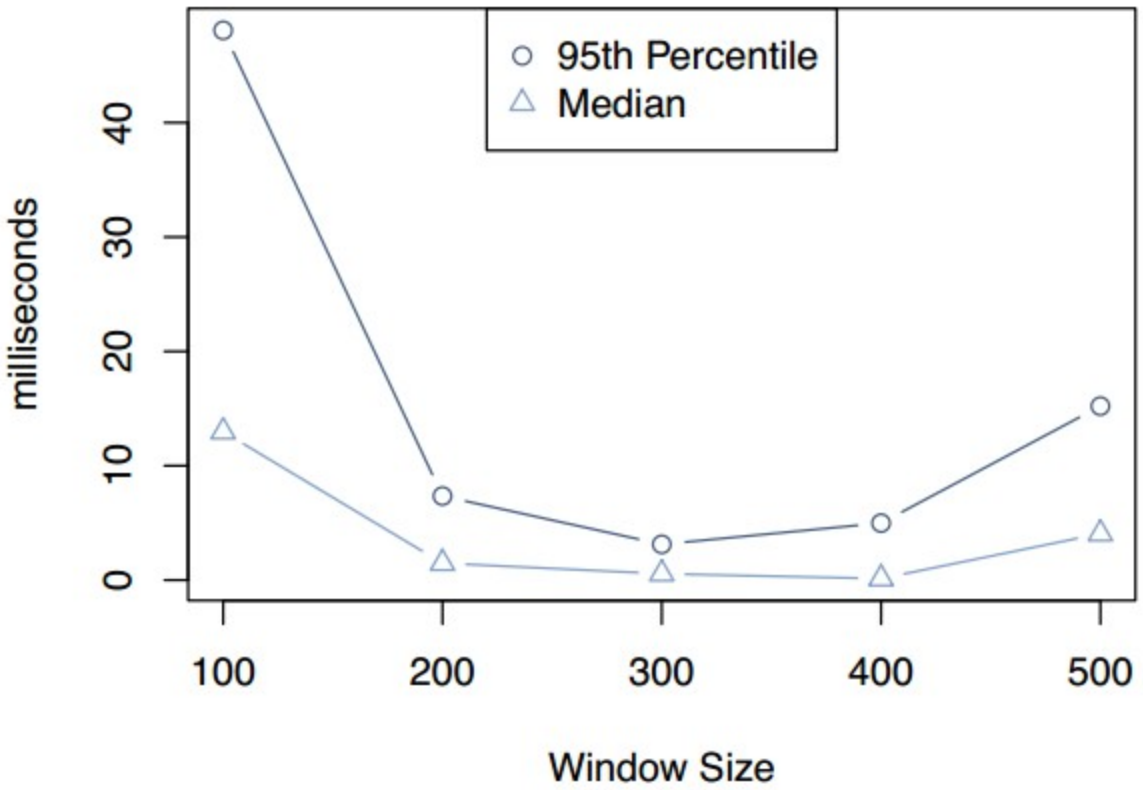

图4：web请求平均等待调度时间 图4展示了窗口大小对web服务器中处于运行态的用户请求等待调度总时间的影响。这些数据从一个前端集群的多台机架采集而来。在每个web服务器，用户请求呈现泊松到达过程。参照Little定律[26]，L=λW，假设输入请求速率是恒定的（在我们的试验中就是这样），在服务器排队的请求数量（L）正比于处理请求的平均时间（W）。web请求的等待调度时间是web请求在系统中数量的一个直接指标。当窗口比较小的时候，应用将不得不串行地分发更多组memcache请求，这将会增加web请求的持续时间。当窗口过大的时候，同时处理的memcache请求的数量将会引发incast拥塞。结果将会是memcache错误，应用退化到从持久化存储中取数据，这样将会导致对web请求的处理更缓慢。在这两个极端之间有一个平衡，处于这个平衡的时候，不必要的延迟将会避免，同时incast拥塞可以被最小化。 **3.2 减少负载**
我们使用memcache来减少用更耗时的方式读数据的频率，比如数据库查询。当期望的数据没有被缓存的时候，web服务器将会退化到使用更耗时方式。下述子章节将会描述三种技术，用来减少负载。

3.2.1 租约（leases）

我们引入了一个称为租约（leases）的新机制来解决两个问题：过时设置（stale sets）和惊群（thundering herds）。当web服务器更新一个在缓存中不是最新版本的值的时候，一次过时设置就发生了。当对memcache的并发更新重新排序的时候，这种情况是会发生的。当某个特定的主键被大量频繁的读写，那么一次惊群就发生

了。因为写操作反复地使最近设置的值失效，那么读操作将会默认地使用更耗时的方式。我们的租约机制解决了这两个问题。

[译者注：此处的leases与Cary G. Gray的leases不一样，不要混淆。]

直观地，当这个客户端发生缓存不命中时，memcached实例给客户端一个租约，将数据设置到缓存中。租约是一个64bit的令牌，与客户端初始请求的主键绑定。当设值到缓存中时，客户端提供这个租约令牌。通过这个租约令牌，memcached可以验证和判断是否这个数据应该被存储，由此仲裁并发写操作。如果因为收到了对这个数据项的删除请求，memcached使这个租约令牌失效，那么验证操作将会失败。租约阻止过时设置的方法类似于load-link/store-conditional操作[20]。

对租约的轻微改动也可以缓和惊群这个问题。每个memcached服务器调节返回令牌的速率。默认情况，我们配置服务器对于每个主键每10秒钟返回一个令牌。当在10秒钟之内有请求，一个特殊的通知将会告诉客户端稍等一下。通常，拥有租约的客户端将会在几个毫秒的时间内成功设置数据。因此，当等待客户端重试的时候，数据经常已经在缓存中了。 为了说明这一点，我们针对容易造成惊群的主键集合收集了一个星期的缓存不命中的记录。如果没有租约机制，所有的缓存不命中都会造成数据库查询率的峰值——17K/s。使用租约机制的时候，数据库查询率的峰值是1.3K/s。因为我们依据峰值负载准备数据库，所有租约机制提供了显著的效率增益。

过期值：当使用租约机制的时候，我们可以最小化某些特定用例下的应用等待时间。我们可以通过鉴别返回稍微过期数据可以接受的情况进一步减少等待时间。当一个主键被删除的时候，对应的值转移到一个保存最近删除项的数据结构中，在被清楚之前将会存活很短的时间。一个get请求可能返回一个租约，或者是一个标记为已过时的数据。应用可以使用过时的数据继续转发处理，而不需要等待从数据库读取的最新数据。经验告诉我们因为缓存数据趋向于单调递增的数据库快照，大部分应用可以在对数据不做改变的情况下使用过时数据。

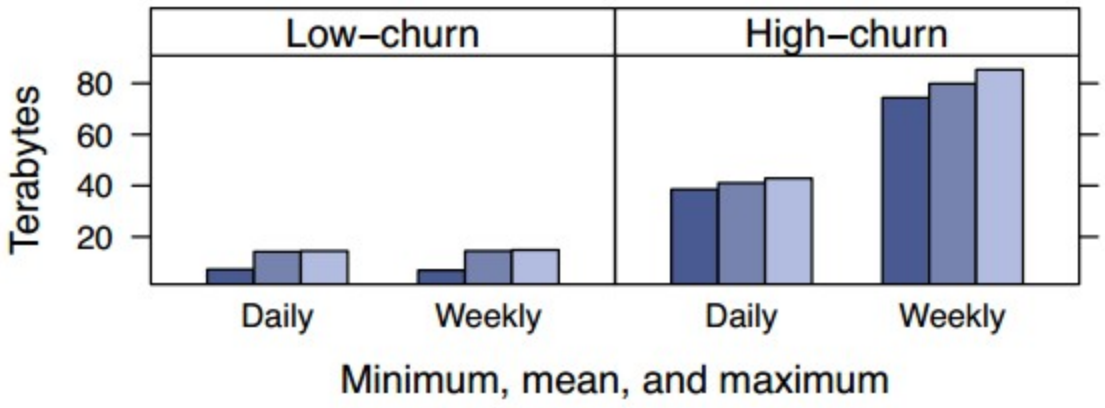

图5：高抖动键集合和低抖动键集合的每日和每周的工作集 **3.2.2 memcache池**

使用memcache做为通用的缓存层要求不同的工作负载分享基础设施，尽管它们具有不过的接入模式、内存占用和服务质量要求。不同应用的工作负载可以产生负干扰，这将会导致命中率下降。

为了适用这些差异，我们将集群的memcached服务器分割成独立的池。我们指定一个池（称作wildcard）为默认池，针对那些放在wildcard中不合适的主键提供另外的池。例如，我们可能为频繁存取但是缓存不命中不耗时的主键分配一个小池。我们也可能为那些不频繁存取但是缓存不命中异常耗时的主键分配一个大池。

图5展示了两个不同的项目集合的工作集，一个低抖动，另一个高抖动。工作集通过对每百万分之一数据项采样所有操作来近似。对于这些数据项，我们收集最小、平均和最大数据项大小。这些数据项大小被加总，然后乘以一百万来近似工作集。每日和每周工作集的不同指出抖动的总数。具有不同抖动特征的数据项以一种不幸的方式相互影响：那些仍然有价值的低抖动主键在那些不再被存取的高抖动主键之前被踢出。将这些不同的主键放在不同的池中将会阻止这种负干扰，同时使我们可以通过设置高抖动池的大小来适用缓存不命中的成本。第7章提供了更深入的分析。

[译者注：工作集定义为在一个特定的时间段内一个进程所需要的内存]

3.2.3 池内的复制（replication）

在某些池内，我们使用复制（replication）来改善延迟和memcached服务器的效率。当（1）应用常规地同时读取很多主键，（2）整个数据集集合可以放到一或两个memcached服务器中，（3）请求率非常高，超出了单台服务器的处理能力的时候，我们选择复制池内的一类主键。

比起进一步划分主键空间，我们更倾向于在实例内进行复制。考虑一个包含100个数据项的memcached服务器，具有对每秒500K请求进行处理的能力。每一个请求查找100个主键。在memcached中每个请求查询100个主键与查询1个主键之间开销的差值是很小的。为了扩展系统来处理1M请求/秒，假如我们增加了第二台服务器，将主键平均分配到两台服务器上。现在客户端需要将每个包含100个主键的请求分割为两个并行的包含50个主键的请求。结果两台服务器都仍然不得不处理每秒1M的请求。然后，如果我们复制所以100个主键到两台服务器，一个包含100个主键的客户端请求可以被发送到任意副本（replica）。这样将每台服务器的负载降到了每秒500K个请求。每一个客户端依据自己的IP地址来选择副本。这种方法需要向所以的副本分发失效消息来维护一致性。 **3.3 故障处理**

无法从memcache中读取数据将会导致后端服务负载激增，这会导致进一步的连锁故障。有两个尺度的故障我们必须解决：（1）由于网络或服务器故障，少量的主机无法接入，（2）影响到集群内相当大比例服务器的广泛停机事件。如果整个的集群不得不离线，我们转移用户的web请求到别的集群，这样将会有效地迁移memcache所有的负载。

对于小范围的停机，我们依赖一个自动化修复系统[3]。这些操作不是即时的，需要花费几分钟。这么长的持续时间足够引发前面提到的连锁故障，因此我们引入了一个机制进一步将后端服务从故障中隔离开来。我们专门准备了少量称作Gutter的机器来接管少量故障服务器的责任。在一个集群中，Gutter的数量大约为memcached服务器的1%。 当memcached客户端对它的get请求收不到回应的时候，这个客户端就假设服务器已经发生故障了，然后向特定的Gutter池再次发送请求。如果第二个请求没有命中，那么客户端将会在查询数据库之后将适当的键-值对插入Gutter机器。在Gutter中的条目会很快过期以避免Gutter失效。Gutter以提供稍微过时的数据为代价来限制后端服务的负载。

注意，这样的设计与客户端在剩下的memcached服务器重新分配主键的方法不同。由于频繁存取的主键分布不均匀，那样的方法会有连锁故障的风险。例如，一个单独的主键占服务器请求的20%。承担这个频繁存取的主键的服务器也会过载。通过将负载分流到闲置的服务器，我们减少了这样的风险。

通常来说，每个失败的请求都会导致对后端储存的一次存取，潜在地将会使后端过载。使用Gutter存储这些结果，很大部分失败被转移到对gutter池的存取，因此减少了后端存储的负载。在实践中，这个系统每天减少99%的客户端可见的失败率，将10%-25%的失败转化为缓存命中。如果一台memcached服务器整个发生故障，在4分钟之内，gutter池的命中率将会普遍增加到35%，经常会接近50%。因此对于由于故障或者小范围网络事故造成的一些memcached服务器不可达的情况，Gutter将会保护后端存储免于流量激增。

4 Region之内：复制（Replication）

随着需求的增长，购买更多的web服务器和memcached服务器来扩展集群是诱惑人的。但是幼稚地扩展系统并不能解决所有问题。随着更多的web服务器加入来处理增长的用户流量，高请求率的数据项只会变的更流行。随着memcached服务器的增加，Incast拥塞也会变的更严重。因此我们将web服务器和memcached服务器分割为多个前端集群。这些集群与包含数据库的存储集群一起统称为region。region架构同样也考虑到更小的故障域和易控制的网络配置。我们用数据的复制来换取更独立的故障域、易控制的网络配置和incast拥塞的减少。

这一章分析了分享同一个存储集群的多个前端集群的影响。特别地，我们说明了允许数据跨集群复制的影响，以及不允许复制潜在的内存效率。

4.1 region内的失效

在region中，存储集群保存数据的权威版本，为了满足用户的需求就需要将数据复制到前端集群。存储集群负责使缓存数据失效来保持前端集群与权威版本的一致性。做为一个优化，当web服务器修改数据后，它也会向所在的集群发送失效命令，提供针对单用户请求的读后写语义，这样可以减少本机缓存的存在时间。

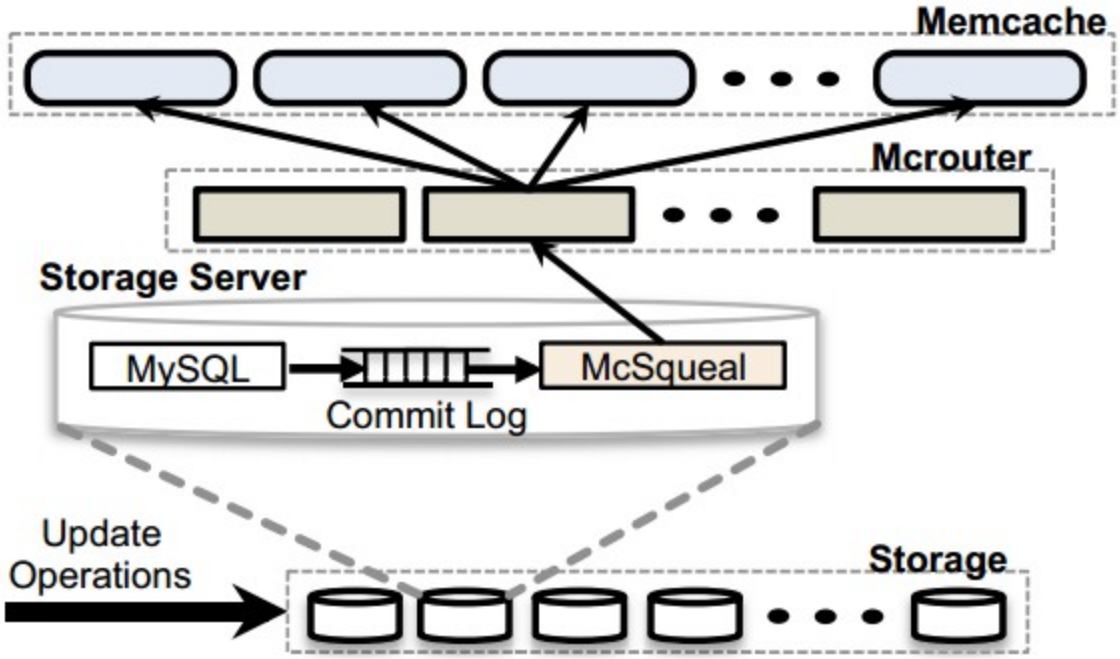

图6：失效流水线 展示那些需要经过守护进程（mcsqueal）删除的主键

修改权威数据的SQL语句被改进为包含事务提交后需要使失效的对应的memcache主键[7]。我们在所有的数据库上部署了失效守护进程（称作mcsqueal）。每个守护进程检查数据库提交的SQL语句，提取任意的删除命令，并且将删除命令广播到region内所有的前端集群。图6展示了这个方法。我们发现大部分发出的失效命令并不会造成删除数据的操作，实际上，所有发出的删除命令只有4%导致实际的缓存数据失效。

减少发包率：如果mcsqueal可以直接联系memcached服务，那么从后端集群到前端集群的发包率将会高的无法接受。有很多数据库和很多memcached服务器跨集群边界通信造成了发包率的问题。失效守护进程批量处理删除操作，使用很少的包把操作发送到每个前段集群运行着mcrouter的指定服务器。然后mcrouter就从每个批量包中分解出单独的删除操作，将失效命令路由到所在前端集群正确的memcached服务器。通过统计每个包中删除命令的中位数可见批处理具有18倍的性能提升。

通过web服务器发送失效命令：通过web服务器广播失效命令到所有前端服务器更简单。很不幸，这个方法存在两个问题。第一个，因为web服务器在批处理无效命令时没有mcsqueal有效率，所以它具有更高的包成本。第二个，当系统性的无效问题出现时，这种方法会无能为力，比如由于配置错误造成的删除命令错误路由。过去，这经常需要动态重启整个memcache基础设施，这样一个缓慢的、破坏性的进程是我们一直想避免的。相反，将失效命令嵌入SQL语句允许mcsqueal简单的重新执行可能已经丢掉的或者错误路由的失效命令，因为数据库提交存储有可靠的日志。

	A (Cluster)	B (Region)
Median number of users	30	1
Gets per second	3.26 M	458 K
Median value size	10.7 kB	4.34 kB

表1： 集群复制或region复制的决定性因素

[译者注：动态重启（rolling restart）是赛车比赛中的一个术语。看看F1比赛就会有个直观的概念，比赛的时候经常会出现安全车领着赛车跑两圈，当安全车离开后出现绿旗，这就是一次rolling start]

4.2 Region池

每个集群依照混合的用户请求独立地缓存数据。如果用户请求被随机的路由到所有可获得的前端集群，那么所有

前端服务器缓存的数据将会大致上一样。这就允许我们离线维护某个集群，而不会导致缓存命中率下降。过度复制数据会使内存没有效率，特别是对很大的、很少存取的数据项。通过使多个前端集群分享同一个memcached服务器集合，我们就可以减少副本的数量。我们称此为region池。

跨集群边界通信会导致更大的延迟。另外，我们的集群间可获得带宽比集群内的少40%。复制用更多的memcached服务器换取更少的集群间带宽，低延迟和更好的容错。对于某一些数据，放弃副本的好处，每个region一个拷贝，从成本上来说更有效率。扩展memcache的一个主要挑战是决定某主键是应该跨前端集群复制，还是每个region一个副本。当region池发生故障时，Gutter也会被使用。 表1总结了我们应用中具有巨大价值的两类项目。我们将B类型的数据移到region池，对于A类型的不做改变。注意，客户端存取B类型数据的频率比A类型数据低一个数量级。B类型数据的低存取率使它成为region池的主要候选者，因为这样的数据不会对集群间带宽造成不利的影响。B类型数据也会占有每个集群wildcard池25%的空间，所以区域化提供了显著的存储效率。然而在A类型的数据项的大小是B类型的两倍，而且存取更频繁，所以从region的角度考虑，不会将它们放在region池中。目前将数据迁移到region池的依据是基于存取率、数据大小和存取用户数的人工的启发式方法。

4.3 冷集群热身

由于存在的集群发生故障或者进行定期的维护，我们增加新的集群上线，此时缓存命中率会很低，这样会削弱隔离后端服务的能力。一个称作冷集群热身（Cold Cluster Warmup）的系统可以缓和这种情况，这个系统使“冷集群”（也就是具有空缓存的前端集群）中的客户端从“热集群”（也就是具有正常缓存命中率的集群）中检索数据而不是从持久化存储。这利用到了前面提到的跨前端集群的数据复制。使用这个系统可以使冷集群在几个小时恢复到满负载工作能力而不是几天。 必须注意避免由于竞争条件引发的不一致。例如，如果冷集群中的一个客户端对数据库做了更新，另外一个客户端在热集群收到失效命令之前检索到过时数据，这个数据项在冷集群中将会不一致。memcached的删除命令支持非零的拖延时间，也就是在指定的拖延时间内拒绝添加操作。默认情况下，冷集群中所有的删除命令都有两秒钟的拖延时间。当在冷集群中发生缓存不命中时，客户端向热集群重新发送请求，然后将结果添加到冷集群中。如果添加失败就表明数据库中有更新的数据，因此客户端将会重新从数据库读数据。删除命令延迟两秒钟以上在理论上来说也是有可能的，但是对于大部分的情况并不会超过两秒钟。冷集群热身运营上的效益远远超过少数缓存不一致所带来的成本。一旦冷集群的命中率趋于稳定，我们就将冷集群热身系统关掉，同时效益也就减少了。

5 跨地区：一致性

将数据中心分布到广泛的地理位置具有很多优势。第一，将web服务器靠近终端用户可以极大地较少延迟。第二，地理位置多元化可以缓解自然灾害和大规模电力故障的影响。第三，新的位置可以提供更便宜的电力和其它经济上的诱因。我们通过部署多个region来获得这些优势。每个region包含一个存储集群和多个前端集群。我们指定一个region持有主数据库，别的region包含只读的副本；我们依赖MySQL的复制机制来保持副本数据库与主数据库的同步。基于这样的设计，web服务器无论访问本地memcached服务器还是本地数据库副本的延迟都很低。当扩展到多region的时候，维护memcache和持久化存储的数据一致性成了主要的技术挑战。这些挑战源于一个问题：副本数据库可能滞后于主数据库。

在一致性和性能平衡的广泛范围上，我们的系统仅仅表示一个点。一致性模型已经演进了很多年来满足站点扩展的需求。在实践上一致性模型是可以构建的，并且不会牺牲高性能的需求。系统管理的大容量数据隐含着任何增加网络和存储需求的细小改动都有重大的成本。大部分提供严格语义的想法都很少走出设计阶段，因为它们实在是过分的昂贵。与专门针对存在的用例而定制的系统不同，memcache和Facebook是一起开发的。这就允许应用工程师和系统工程师可以一起工作来设计一个模型，这个模型对于应用工程师来说是易于理解的、高效的，而且足够的简单来实现可靠的扩展。我们提供了尽力而为的最终一致性，但是强调性能和可用性。因此在实践上这个系统工作的非常好，而且我们找到了一个可以接受的平衡点。 **从主region写：**前面我们提到在我们的系统中是通过存储集群的守护进程来实现数据失效的，这样的设计对多region架构的设计具有重要的影响。特别的，这样的设计可以避免一个竞争情况，也就是在数据从主region复制过来之前失效命令先到达了。考虑一下这种情况，主region中的一个web服务器已经完成对数据库的修改，寻求使现在过时的数据失效。在主region中发送失效命令是安全的。然而，让副本region中的web服务器发送失效命令可能是不成熟的，因为对主数据库的改动可能还没有传播到副本数据库。接下来对副本region的数据查询将会与数据库复制产生竞争，因此增加了将过时数据设置到memcache中的概率。历史上，在扩展到多个region之后，我们实现了mcsqueal。 **从非主region写：**现在考虑当复制滞后非常大的时候，用户从非主region更新数据。如果他最近的改动丢失了，那么下一个请求将会导致混乱。之后当复制流完成之后才允许从副本数据库读取数据并缓存。如果没有这个保障，后续请求将会导致副本中的过时数据被读取并且缓存。

我们使用远程标记（remote marker）机制来最小化读取过时数据的概率。出现标记就表明本地副本数据库中的数据可能是过时的，所以查询应该重定向到主region。当web服务器想要更新主键为k的数据，那么服务器（1）在region中设置远程标记 r_k，（2）向主数据库执行写操作，并且在SQL语句中嵌入应该失效的k和 r_k，（3）在本集群删除k。对于主键k的后续请求，web服务器找不到缓存数据，然后就检查 r_k是否存在，如果存在就将请求定向到主region，否则定向到本地region。在这种情况下，我们用缓存不命中时附加的延迟来换取读取过时数据概率的下降。
我们通过使用region池来实现远程标记。注意，当对于同一个主键并发修改的时候，一个操作可能删除远程标记，而这个标记应该为另外一个正在执行的操作保留，如果出现这种情况，我们的机制就可能返回过时的信息。有一点是需要特别说明的，我们对远程标记memcache的使用以一种微妙的方式违反了缓存结果。做为缓存，删除或移除主键都是安全的；它可能会引起更多的数据库负载，但是不会削弱一致性。相反，远程标记的出现可以帮助区分是否非主数据库拥有过时数据。在实践上，我们发现移除远程标记和并发修改的情况都很少。

运行上的考虑：跨region通信是非常耗时的，因为数据不得不穿过很大的地理距离（比如穿过合众国大陆）。通过分享数据库复制删除流的通信信道，我们在低带宽连接情况下获得了网络效率。

上述4.1章提到的管理删除的系统也部署在了副本region，通过副本数据库广播删除命令到memcached服务器。当下流组件没有反应时，数据库和mcrouter暂存删除命令。任何组件的故障和延迟都会导致读取过时数据概率的增加。一旦下流组件重新可获得了，暂存的删除命令将会重新发送。当发现问题时，代替的方案是让集群下线或者是使前端集群的数据失效。这些方法比起所获得的工作负荷上的好处将会导致更多的混乱。

6 单个服务器的提升

多对多的通信模式隐含着单独的服务器将会成为集群的瓶颈。这章将会讲述性能调优和memcached内存效率的提高，这有利于集群更好的扩展。提升单个服务器缓存的性能是一个活跃的研究领域[9,10,28,25]。

6.1 性能调优

我们开始使用具有固定大小哈希表的单线程memcached。第一批主要的优化是：（1）允许哈希表自动扩展来避免查找时间漂移到O(n)，（2）通过一个全局锁来保护多数据结构使得服务器多线程化，（3）赋予每个线程独立的UDP端口来减少发送副本和稍后传播中断处理开销的争用。前两个优化都贡献给了开源社区。下述章节将会探索还没在开源版本出现的进一步优化。

我们的实验主机拥有一颗2.67GHz（12核、12超线程）的Intel Xeon CPU（X5650），一个Intel 82574L千兆以太网控制器和12GB内存。生产服务器具有更多的内存。更多的细节之前已经公开过[4]。性能测试设备包含15个生成memcache流量的客户端和一台具有24线程的memcached服务器。客户端和服务器放在同一个机架，之间通过千兆以太网连接。这些测试测量两分钟持续负载memcached反应的延迟。

[译者注：X5650好像是6核的]

获取的性能：我们首先研究将原有的多线程单锁的实现替换为细粒度锁的效益。在发送包含10个主键的memcached请求的之前，我们预先填充了拥有32byte值的缓存数据，然后我们测量命中的性能。图7展示了对不同版本的memcached持续的亚毫秒级平均返回时间的最大请求率。第一组柱状图是实现细粒度锁之前的memcached，第二组是我们当前的memcached，最后一组是开源版本1.4.10，这个版本独立实现了一个我们的锁策略的更粗粒度的版本。

使用细粒度的锁使得请求命中的峰值从每秒600K到达了1.8M提升了3倍。不命中的性能也从每秒2.7M提升到了4.5M。因为返回值需要构建和传输，而不命中对于整个多请求仅需要一个表明所有主键不命中的静态回应（END），所以命中的情况更耗时。

我们也研究了使用UDP代替TCP的性能影响。图8展示了对于单主键获取和10个主键获取平均持续延迟小于一毫秒的请求峰值。我们发现UDP实现的性能在单主键获取情况下超出TCP实现13%，在10主键获取的情况下超出8%。

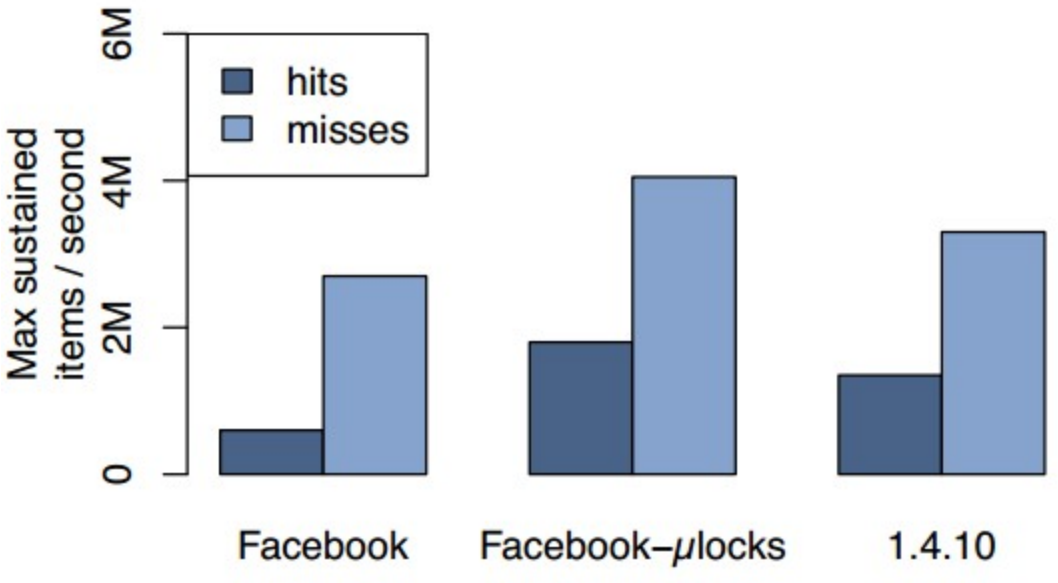

图7：对不同memcached版本多获取命中和不命中的性能比较

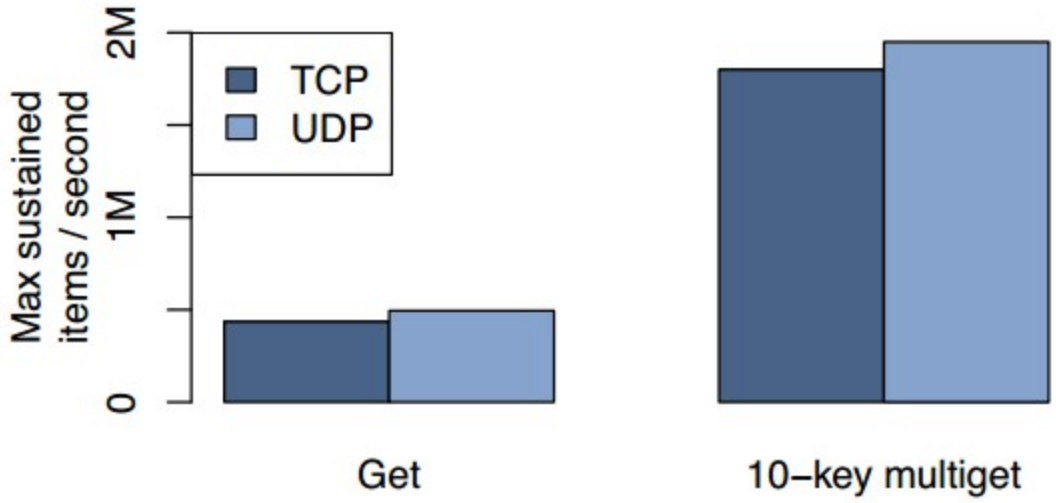

图8：对TCP和UDP单主键请求和10主键请求获取命中的性能比较

因为多主键获取在单个请求比单主键获取多打包了很多数据，所以它们使用了更少的包完成了同样的事情。图8展示了10个主键获取比单主键获取有接近4倍的性能提升。

6.2 适应性的slab分配器

memcached使用一个slab分配器来管理内存。这个分配器将内存组织为slab类，每类包含预分配的均匀大小的内存块。memcached将数据项存储到可以适应数据项元数据、键和值大小的最小可能性的slab类。slab类从64byte开始，以1.07为因子指数性增加到1MB，以4byte对齐[3]。每个slab类维护一个可获得内存块的空闲列表，当它的空闲列表是空的，那么就从1MB slab中请求更多内存。一旦memcached服务器再也不能分配空闲内存，通过移除slab类中最近最少使用（LRU）的数据项来存储新的数据项。当工作负载改变时，原有分配给每个slab类的内存可能不再足够，这样将会导致低命中率。

========== ==========
3 对页取数据的量位于第95百分位的是1740个数据项。这个扩展的因子确保我们同时拥有64byte和128byte，这样更有利于利用硬件缓存线。

我们实现了一个适应性的分配器，这个分配器将会周期性的重新平衡slab分配来适应当前的工作负载。如果slab类正在移除数据项，而且如果下一个将要被移除的数据项比其它slab类中的最近最少使用的数据项的最近使用时间至少近20%，那么就说明这个slab类需要更多内存。如果找到了一个这样的slab类，那么就将存储最近最少使

用数据项的slab释放，然后转移到needy类。注意，开源社区已经独立实现了一个类似的平衡跨slab类移除率的分配器，然而我们的算法关注平衡所有类中最久数据项的时长。平衡时长比调整移除率对整台服务器单个全局最近最少使用（LRU）移除策略提供了更好的近似，而且调整移除率深受接入模式的影响。

6.3 临时条目的缓存

因为memcached支持过期时间，条目在它们过期之后仍可以驻留在内存中。 当条目被请求时或者当它们到达LRU的尾端时，Memcached会通过检查过期时间来延时剔除这些条目。 尽管在一般情况下很有效，但是这种模式允许那些偶尔活跃一下的短期键值占据内存空间，直到它们到达LRU的尾部。

所以我们引入一种混合模式，对多数键值使用延时剔除，而对过期的短期键值则立即剔除。我们根据短期条目的过期时间把它们放入一个由链接表构成的环形缓存区（花费几秒编入索引直到过期） – 我们称之为临时条目缓存区。每一秒钟，该缓存的头部数据格里的所有条目都会被剔除，然后头部向前移动一格。当我们给一个频繁使用的键值集合（它们对应条目的寿命很短）设置一个短超期时间后， 该键值集合使用的memcache缓冲池的比例从6%下降到0.3%，而没有影响到命中率。

6.4 软件升级

升级、bug修复、临时诊断或性能测试都需要频繁的软件变更。一个memcached服务器能够在几小时内达到 90% 的命中率峰值。接下来，可能会耗费12小时来进行memcached服务器升级，这将要求我们谨慎管理数据库负载。我们修改了memcached，使用 System V 共享内存区来存储缓存值和主数据结构，以便在软件升级过程中数据仍能够保持可用，进而最小化损失。

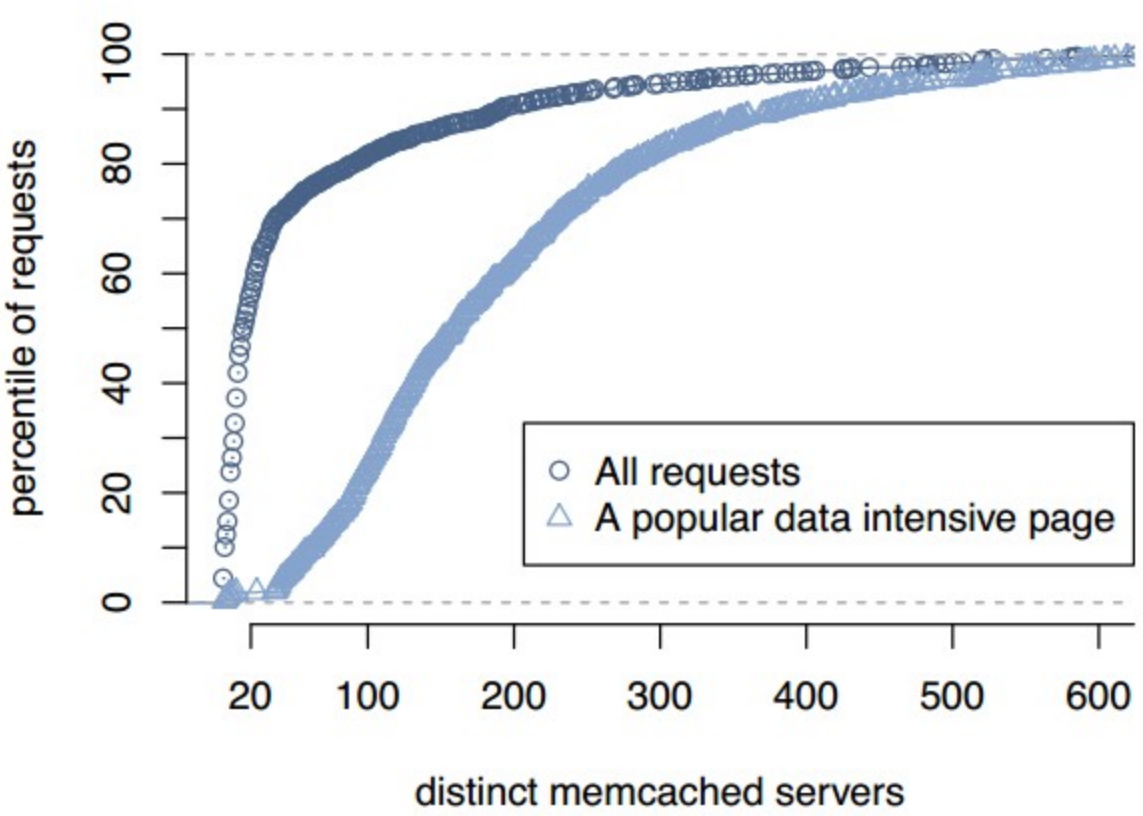

图 9: 不同数量memcached服务器的访问数累积分布图

7 memcache工作负载

现在我们用从生产环境中运行服务器上所获得的数据来描述memcache的负载。

7.1 web服务器上的测量

我们收集了小比例用户请求的所有memcache操作，然后讨论了扇出（fanout）、响应大小和我们工作负载的延迟特征。

扇出：图9展示了当web服务器回应一个页面请求时需要联系的memcahced服务器数量的分布。由图可见，56%的页面请求联系少于20台memcached服务器。按照传输量来说，用户请求倾向于请求小数量的缓存数据。然而这个分布存在一个长尾。这张图也展示了对于流行页面的请求分布，这样的页面可以更好的展示出多对多的通信模式。大部分这样的请求将会接入超过100台独立的服务器；接入几百台memcached服务器也不是少数。

响应大小：图10展示了对memcache请求的响应大小。中位数（135byte）与平均数（954byte）之间的差值隐含着缓存项的大小存在很大差异。另外，在近似200byte和600byte处有三个不同的峰值。大的数据项倾向于存储数据列表，而小的数据项倾向于存储单个内容块。

延迟：我们测量从memcache请求数据的往返延迟，这个延迟包含了请求路由和接收回复的成本、网络传输时间和反序列化和解压缩的成本。通过7天的统计，请求延迟的中位数是333微秒，位于第75百分位的是475微秒，位于第95百分位的是1.135毫秒。空闲web服务器端到端延迟的中位数是178微秒，位于第75百分位的是219微秒，位于第95百分位的是374微秒。在第95百分位上延迟的巨大差异是由处理数据量大的回应和等待可运行线程调度引起的，这些在3.1章已经讨论过了。

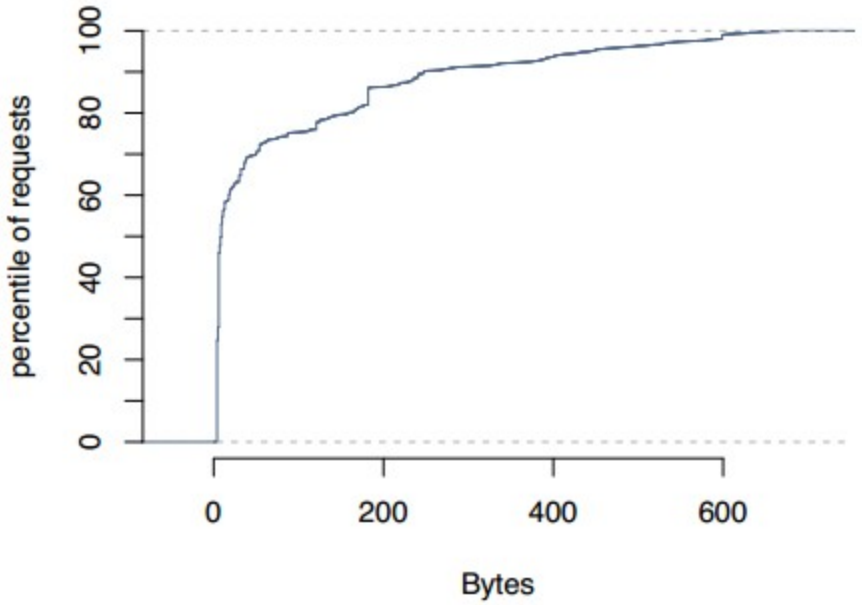

图10：读取数据大小的累积分布

7.2 池的统计

现在我们讨论四个memcache池的测量。这些池分别是wildcard（默认池），app（专门设定给特定应用的池），给存取频繁的数据的replicated pool，给很少存取的数据的regional pool。在每个池中，我们每四分钟收集一次平均的统计，表2展示了一个月的统计周期的最大平均值。这些数据近似于那些池的峰值负载。这张表显示了对于不同的池，get、set和delete操作的频率存在很大差异。表3展示了每个池响应大小的分布。这些不同的特征激发了我们分隔不同工作负载的欲望。

就像在3.2.3章讨论过的那样，我们在池内复制数据，使用批处理的优势来处理高请求率。我们观察到，replicated pool具有最高的get操作率（差不多是第二高的2.7倍），最高的字节比包的比率，尽管该池有最小的数据项大小。这些观察数据与我们设计的期望一致，我们就是想利用复制和批处理来实现更好的性能。在app池，更高的数据抖动自然而然将会导致更高的不命中率。这个池倾向于将数据保存几个小时，然后就会被新的数据踢出。在regional pool中的数据倾向于是较大的而且不频繁被存取的，就像表中的的请求率和数据大小分布展示的那样。

7.3失效延时

我们发现，在确定暴露过期数据的概率上，失效的及时性是一个关键因素。为了监控该生命值，我们从百万次删除操作中取样一次并记录删除命令发出的时间。随后，我们定期地为该样本查询所有前端集群中memcache的内容，如果删除命令将一个字段设定为无效时，该字段仍然缓存，则记录一个错误。

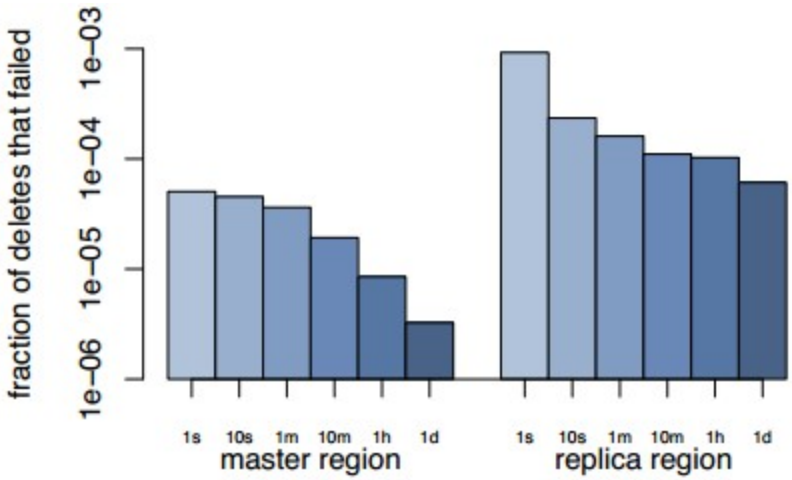

图11:删除的管线的延时

在图11中，我们使用这种监控机制来统计30天的失效延迟。我们将数据分为两组：（1）删除操作由主region中的web服务器发起，并被发送到主region中的一个memcached服务器，（2）删除操作从副本region发起，并且发送到另外一个副本region。由统计数据可以见，当参数操作的发起地和目的地都是在主region的时候，成功率非常的高，一秒钟内就可以达到四个九的可靠性，一小时之后就可以达到五个九。当删除操作的发起地和目的地都不在主region的时候，一秒钟内的可靠性下降到了三个九，十分钟内才达到四个九。按照我们的经验，当几秒钟之后失效操作不成功，最可能的原因是第一次尝试失败，接下来的重试将会解决这个问题。

8 相关工作

一些其他的大型网站已经意识到key-value存储的应用。DeCandia 等[12]构建了高可用的key-value存储系统（Dynamo），已经亚马逊网站应用服务中大量使用。相比较，Dynamo主要着眼于优化高负荷状况下的写操作，而我们系统的负荷主要是大量的读操作。类似的有，LinkedIn 使用Voldemort[5],由Dynamo衍生而出。其他被大量使用的key-value存储方案包括Github，digg和Blizzard使用Redis[6]；Twitter[33]和Zynga使用memcahed。Lakshmanet等[1]开发了Cassandra,一个基于模式的分布式key-value数据库。然而我们更趋向于使用和扩展memcached,主要是由于其简单的设计。

我们的工作是扩展memecached使其在分布式数据架构下工作。Gribble等[19]构建了一个早期版本的key-value存储系统用于Internet扩展服务。Ousterhout等[29]也构建了一个大规模内存key-value存储系统。与这些方案不同，memcache不保证持久性。我们利用其它的系统来解决数据存储的持久性问题。

pool	miss rate	$\frac{get}{s}$	$\frac{set}{s}$	$\frac{delete}{s}$	$\frac{packets}{s}$	outbound bandwidth (MB/s)
wildcard	1.76%	262k	8.26k	21.2k	236k	57.4
app	7.85%	96.5k	11.9k	6.28k	83.0k	31.0
replicated	0.053%	710k	1.75k	3.22k	44.5k	30.1
regional	6.35%	9.1k	0.79k	35.9k	47.2k	10.8

表2：各类型的平均超过7天的memcache程序池流量图

pool	mean	std dev	p5	p25	p50	p75	p95	p99
wildcard	1.11 K	8.28 K	77	102	169	363	3.65 K	18.3 K
app	881	7.70 K	103	247	269	337	1.68K	10.4 K
replicated	66	2	62	68	68	68	68	68
regional	31.8 K	75.4 K	231	824	5.31 K	24.0 K	158 K	381 K

表 3: 各类型程序池关键词大小分布（k）

Ports等[31]提供了一个用于管理事任务数据库查询结果缓存的类库。我们需要的是一个更灵活的缓存措略。我们利用最近和过期读优先措略用来研究高性能系统下缓存一致性和读操作。Ghandeharizadeh和Yap等研究也提出了一个公式，解决基于时间标记而不是确定的版本号过期集合的问题。

虽然软路由易于定制和编程，但是相比较硬路由其效率更低。Dobresuet等[13]利用多处理器、多存储控制器，多队列网络接口和批处理的方式在通用服务器上研究了这些问题。利用这些技术实现微路由来保持进一步的工作。Twitter也独立开发了一个类似微路由的memcache代理[32]。

在Coda[35]中，Satyanarayanan等展示了如何把由于不连贯的操作导致的数据集分歧恢复一致。Glendenninget等[17] 利用杠杆作用Paxos公式[24]和加权因子[16]构建了Scatter,一个线性语义搅动的非贡献哈希表。

TAO[37]是facebook 的另一个严重依赖缓存的系统，主要用户保证大数据量查询时能保持低延迟。TAO和memcache有两方面的重大不同。（1），TAO由一个图形模型实现，在模型中每一个节点由一个固定长度的持久标识符（64位整数）来标识。（2）TAO 有一个编码规范，把它的图形模型映射到持久存储，并且对持久层负责。其他大量的组件，比如我们的客户类库和微路由，都可以在两个系统中通用。

9 总结

在这篇文章里，我们展示了使用基于memcached的技术来满足Facebook不断增长的需求. 文中讨论的很多权衡都不是很基础, 但是却是在优化线上系统性能时真实遇到的，而这个线上系统的规模还在持续部署新产品的过程中不停的扩大. 在建设、维护、扩容我们的系统时，我们学到了一下的经验。(1) 分离的缓存和持久化存储系统使我们可以对他们进行单独的度量. (2) 监视、报错、可选的特性和性能一样重要. (3) 管理有状态的组件要比管理无状态组件复杂的多. 所以将逻辑保存在无状态的客户端里会对特性的反复调用有帮助并且使系统的分裂最小化. (4) 系统要可以逐步的增加或减少新功能，即使这会导致系统的功能集临时的异构. (5) 简洁至关重要.

总结

非常感谢Philippe Ajoux, Nathan Bron-son, Mark Drayton, David Fetterman, Alex Gartrell, Andrii Grynenko, Robert Johnson, Sanjeev Kumar, Anton Likhtarov, Mark Marchukov, Scott Marlette, Ben Maurer, David Meisner, Konrad Michels, Andrew Pope, Jeff Rothschild, Jason Sobel, and Yee Jiun Song ，他们提供的杰出贡献. 我们同时非常感谢那些不知名的评论者，以及我们的指导 Michael Piatek, Tor M. Aamodt, Remzi H. Arpaci-Dusseau, and Tayler Hetherington ，在我们撰写这篇文章的时候他们给了我们很多宝贵的反馈. 最后我们要感谢facebook的工程师同事们，他们给出了很多意见，bug报告，并且支撑memcache成为今天这个样子。

References

[1] Apache Cassandra. http://cassandra.apache.org/.
[2] Couchbase. http://www.couchbase.com/.
[3] Making Facebook Self-Healing. https://www.facebook.com/note.php?note_id=10150275248698920.
[4] Open Compute Project. http://www.opencompute.org.
[5] Project Voldemort. http://project-voldemort.com/.
[6] Redis. http://redis.io/.
[7] Scaling Out. https://www.facebook.com/note.php?note_id=23844338919.

[8] ATIKOGLU, B., XU,Y.,FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. ACM SIGMETRICS Performance Evaluation Review 40, 1 (June 2012), 53–64.
[9] BEREZECKI, M., FRACHTENBERG, E., PALECZNY, M., AND STEELE, K. Power and performance evaluation of memcached on the tilepro64 architecture.Sustainable Computing: Informat-ics and Systems 2, 2 (June 2012), 81 – 90.
[10] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK,M.F.,MORRIS, R., AND ZELDOVICH, N. An analysis of linux scalability to many cores. In Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation(2010), pp. 1–8.
[11] CERF, V. G., ANDKAHN, R. E. A protocol for packet network intercommunication. ACM SIGCOMM Compututer Communi-cation Review 35, 2 (Apr. 2005), 71–82.
[12] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAP-ATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN,S., VOSSHALL, P., ANDVOGELS, W. Dynamo: amazon’ s highly available key-value store.ACM SIGOPS Operating Sys-tems Review 41, 6 (Dec. 2007), 205–220.
[13] FALL, K., IANNACCONE, G., MANESH, M., RATNASAMY, S., ARGYRAKI, K., DOBRESCU, M., ANDEGI, N. Routebricks: enabling general purpose network infrastructure.ACM SIGOPS Operating Systems Review 45, 1 (Feb. 2011), 112–125.
[14] FITZPATRICK, B. Distributed caching with memcached. Linux Journal 2004, 124 (Aug. 2004), 5.
[15] GHANDEHARIZADEH, S., ANDYAP, J. Gumball: a race con-dition prevention technique for cache augmented sql database management systems. In Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks(2012), pp. 1–6.
[16] GIFFORD, D. K. Weighted voting for replicated data. In Pro-ceedings of the 7th ACM Symposium on Operating Systems Prin-ciples(1979), pp. 150–162.
[17] GLENDENNING, L., BESCHASTNIKH, I., KRISHNAMURTHY, A.,ANDANDERSON, T. Scalable consistency in Scatter. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles(2011), pp. 15–28.
[18] GRAY, C., ANDCHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. ACM SIGOPS Operating Systems Review 23, 5 (Nov. 1989), 202–210.
[19] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. InProceedings of the 4th USENIX Sym-posium on Operating Systems Design & Implementation(2000), pp. 319–332.
[20] HEINRICH, J. MIPS R4000 Microprocessor User’ s Manual. MIPS technologies, 1994.
[21] HERLIHY, M. P., ANDWING, J. M. Linearizability: a correct-ness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12, 3 (July 1990), 463–492.
[22] KARGER, D., LEHMAN, E., LEIGHTON,T.,PANIGRAHY, R., LEVINE, M., ANDLEWIN, D. Consistent Hashing and Random trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Proceedings of the 29th annual ACM Symposium on Theory of Computing(1997), pp. 654–663.
[23] KEETON, K., MORREY, III, C. B., SOULES, C. A., AND VEITCH, A. Lazybase: freshness vs. performance in informa-tion management. ACM SIGOPS Operating Systems Review 44,1 (Dec. 2010), 15–19.
[24] LAMPORT, L. The part-time parliament. ACM Transactions on Computer Systems 16, 2 (May 1998), 133–169.
[25] LIM, H., FAN, B., ANDERSEN, D. G., ANDKAMINSKY, M. Silt: a memory-efficient, high-performance key-value store. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles(2011), pp. 1–13.
[26] LITTLE, J., ANDGRAVES, S. Little’ s law. Building Intuition (2008), 81–100.
[27] LLOYD,W.,FREEDMAN, M., KAMINSKY, M., ANDANDER-SEN, D. Don’ t settle for eventual: scalable causal consistency for wide-area storage with COPS. InProceedings of the 23rd ACM Symposium on Operating Systems Principles(2011), pp. 401–416.
[28] METREVELI, Z., ZELDOVICH, N., AND KAASHOEK, M. Cphash: A cache-partitioned hash table. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming(2012), pp. 319–320.
[29] OUSTERHOUT, J., AGRAWAL,P.,ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZI` ERES, D., MI-TRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramcloud. Communications of the ACM 54, 7 (July 2011), 121–130.
[30] PHANISHAYEE, A., KREVAT, E., VASUDEVAN,V.,ANDER-SEN, D. G., GANGER, G. R., GIBSON, G. A., ANDSE-SHAN, S. Measurement and analysis of tcp throughput col-lapse in cluster-based storage

systems. InProceedings of the 6th USENIX Conference on File and Storage Technologies(2008), pp. 12:1–12:14.
[31] PORTS, D. R. K., CLEMENTS,A.T.,ZHANG, I., MADDEN, S.,ANDLISKOV, B. Transactional consistency and automatic management in an application data cache. InProceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation(2010), pp. 1–15.
[32] RAJASHEKHAR, M. Twemproxy: A fast, light-weight proxy for memcached.https://dev.twitter.com/blog/twemproxy.
[33] RAJASHEKHAR, M., ANDYUE, Y. Caching with twem-cache. http://engineering.twitter.com /2012/07/caching-with-twemcache.html.
[34] RATNASAMY, S., FRANCIS,P.,HANDLEY, M., KARP, R., ANDSHENKER, S. A scalable content-addressable network. ACM SIGCOMM Computer Communication Review 31, 4 (Oct.2001), 161–172.
[35] SATYANARAYANAN, M., KISTLER, J., KUMAR,P.,OKASAKI, M., SIEGEL, E., ANDSTEERE, D. Coda: A highly available file system for a distributed workstation environment. IEEE Trans-actions on Computers 39, 4 (Apr. 1990), 447–459.

[36] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review 31, 4 (Oct. 2001), 149–160.
[37] VENKATARAMANI,V.,AMSDEN, Z., BRONSON, N., CABR-ERAIII, G., CHAKKA,P.,DIMOV,P.,DING, H., FERRIS, J., GIARDULLO, A., HOON, J., KULKARNI, S., LAWRENCE, N., MARCHUKOV, M., PETROV, D., ANDPUZAR, L. Tao: how facebook serves the social graph. In Proceedings of the ACM SIGMOD International Conference on Management of Data(2012), pp. 791–792.

本文地址：https://www.oschina.net/translate/scaling-memcache-facebook
原文地址：https://www.usenix.org/conference/nsdi13/scaling-memcache-facebook

---

本文中的所有译文仅用于学习和交流目的，转载请务必注明文章译者、出处、和本文链接
我们的翻译工作遵照 CC 协议，如果我们的工作有侵犯到您的权益，请及时联系我们