Scaling Memcache at Facebook

Memcached 是一种众所周知的、简单的内存缓存解决方案。本文描述了如何 Facebook 利用 memcached 作为构建块来构造和扩展一个分布式键值存储支持世界上最大的社交网络。

1.Introduction

一个社交网络（FB）的**基础架构**通常需要以下

1. 允许实时通信（近似，允许一定的延迟），
2. 动态地，从多个来源聚合内容，
3. 能够访问和更新非常流行（热点）共享内容，以及
4. 能够处理大规模用户请求（每秒数百万次 request ）

传统web架构难以满足以上要求，对于计算、网络和I/O需求压力十分巨大。

为什么使用 Memcached

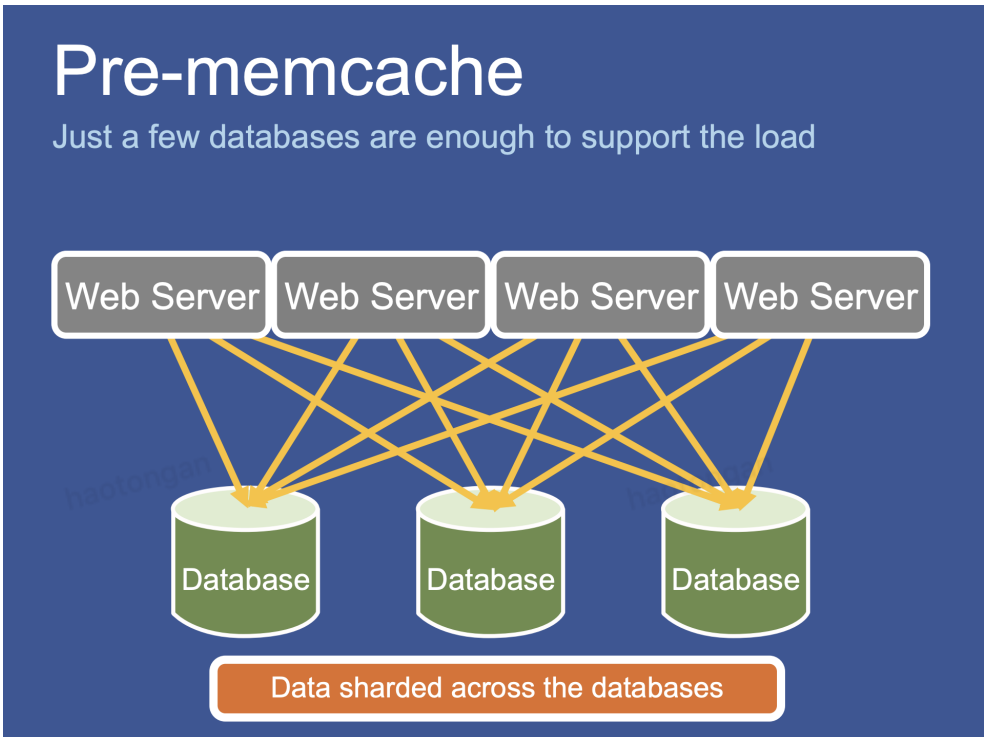
一个页面请求会产生数以百计的数据库请求，而这样的功能只能停止在原型阶段，因为实现起来会太慢，代价也太高。在实际应用里，web 页面通常都会从 memcached 服务器获取数以千计的 key-value 对。

本文包括四个**主要贡献**如下

1. 描述了 Facebook 的基于 memcached 架构的演变。
2. 增强了 memcached，以提高性能并且改进内存效率。
3. 增加机制，提高大规模系统操作的能力。
4. 对生产工作负载赋予了特性。

2 Overview

Pre-memcache



基于以下两个事实

1.人们通常是消费内容而非创造内容（读 >> 写）

2.读操作从各种来源获取数据，例如 MySQL 数据库、HDFS、后端服务等。这就需要灵活的缓存策略真对不同来源，进行数据的缓存。

Query cache

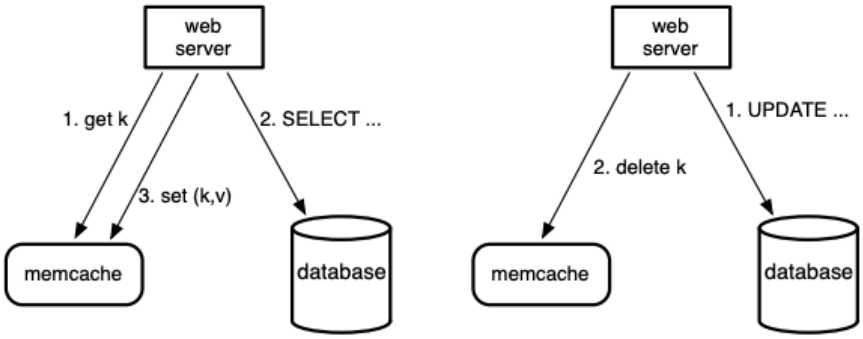


Figure 1: Memcache as a demand-filled look-aside cache. The left half illustrates the read path for a web server on a cache miss. The right half illustrates the write path.

memcache 的 cache policy 主要如下：

- demand-filled look-aside (read)
- write-invalidate (write)

读数据时，web server 先尝试从 memcache 中读数据，若 cache miss 则从 database 中获取数据，并写入到 memcache 中；**写数据时**，先更新 database，然后将 memcache 中相应的数据删除。

Generic cache

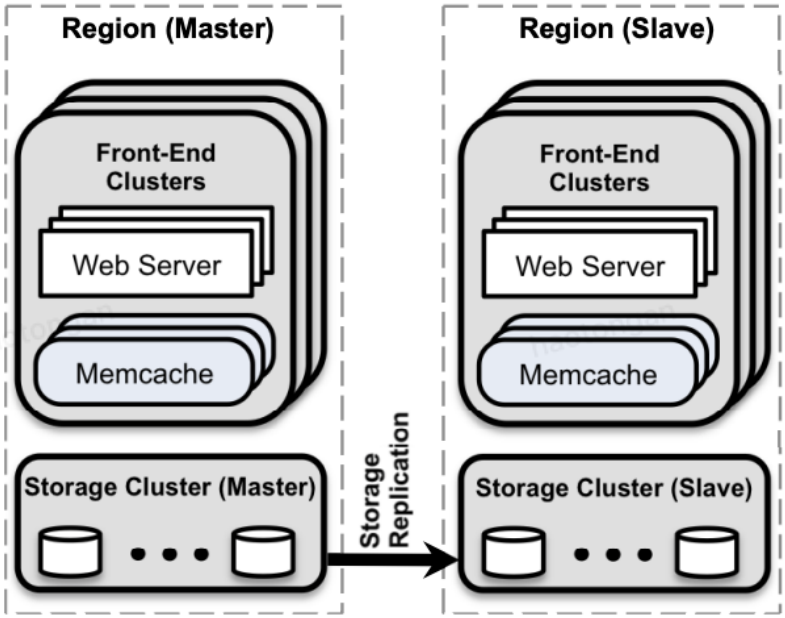


Figure 2: Overall architecture

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。接下来描述如何基于 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。

在三种不同的规模下出现的问题：

Single front-end cluster

- Read heavy workload
- Wide fanout
- Handling failures

Multiple front-end clusters

- Controlling data replication
- Data consistency

Multiple Regions

- Data consistency

在系统的发展中，我们将这两个重大的**设计目标**放在首位：

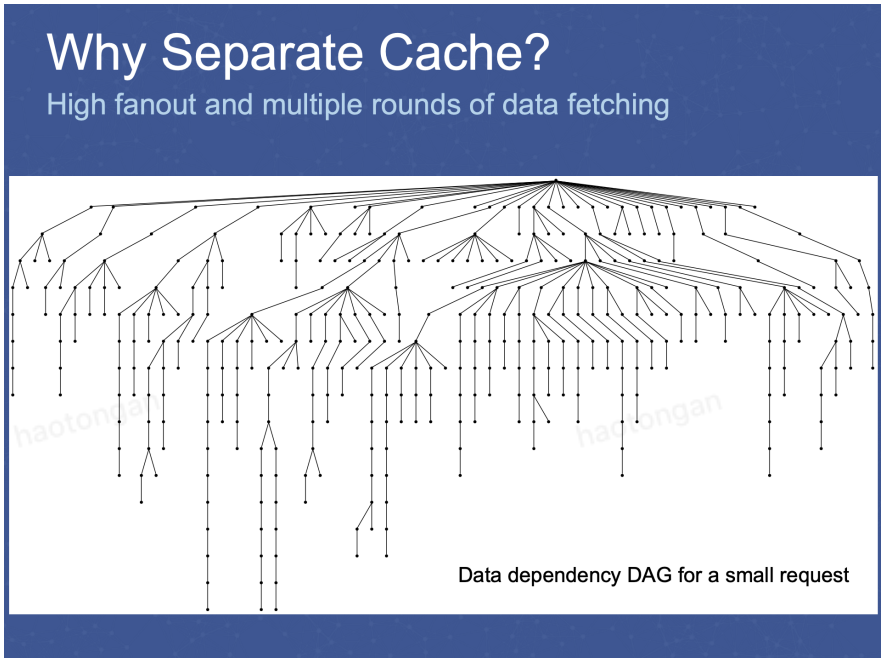
1. 只有已经对用户或运维产生影响的问题，才值得改变。我们极少考虑范围有限的优化。

2. 对陈旧数据的瞬时读取，其概率和响应度类似，都将作为参数来调整。我们会选择暴露略微陈旧的数据以便减少后台存储服务过高的负载。

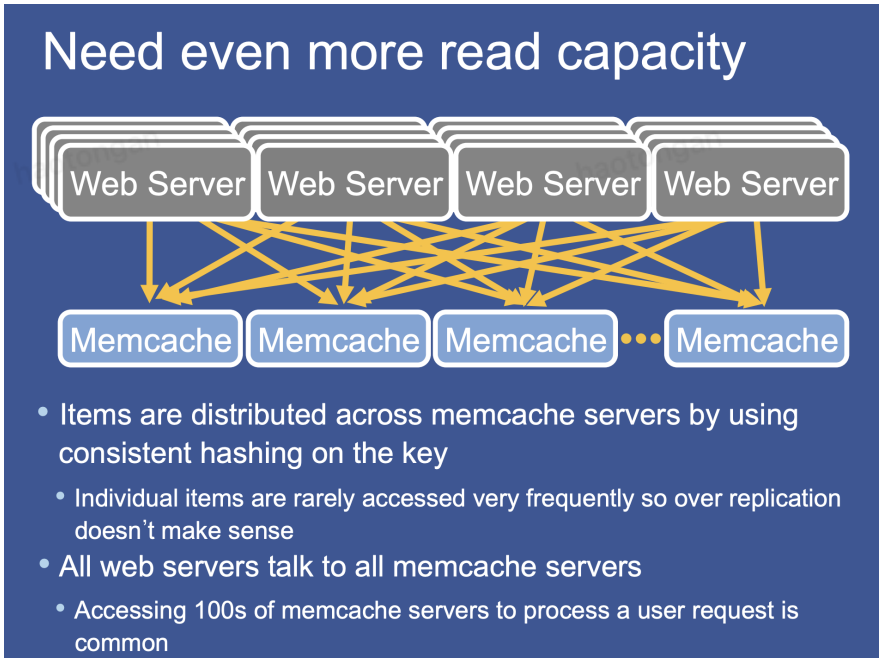
3 In a Cluster: Latency and Load

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 cache miss 时数据库的负载。

3.1 **Reducing Latency**



无论缓存是否命中，Memcache 的响应时间对于用户请求的响应时间来讲都是一个重要因素。单个用户web请求一般包含数百个 Memcache 读请求。如热门页面的加载通常需要从 Memcache 中获取521个不同的资源。



为了减轻数据库和其他服务的负担，我们准备了由数百台 Memcache 服务器组成的集群。资源项通过一致性哈希存于不同的 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。这也产生了 **all-to-all communication** 问题，主要有以下两个影响。

1. **incast congestion**
2. **a single server** to become the **bottleneck** for many web servers.

数据复制通常可以缓解单服务器瓶颈的问题，但是这会导致内存效率使用低下。

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、错误处理以及批处理等。

Parallel requests and batching

应用层面上通过**并行请求**和**批处理**可以在一定程度上减少延迟。通过构造的数据依赖 DAG，最大化并行请求数据资源，同时批量请求（平均24 keys）。

Client-server communication

Memcached 服务器之间不进行通信，主要在客户端上改造，客户端主要提供两个组件：一个是可以嵌入到应用的库，一个称为 Mcrouter 的代理程序。这个代理对外提供 Memcached 服务器的接口，对不同服务器之间的请求/响应进行路由。客户端使用 UDP 和 TCP 协议与 Memcached 服务器通信。

1.对于读请求（get），通过 UDP 来减少延迟。允许 web 服务器中的每个线程绕过 Mcrouter 直接与 Memcached 服务器直接通信，当使用 UDP 通信时，如果出现丢包或者顺序错误的情况，直接返回错误，不尝试解决。实际情况中，在峰值负载条件下，观察 Memcache 客户端约0.25%的请求会被丢弃。其中大约80%是由于延迟或丢包，其余的是失序交付。客户端将这类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，会跳过更新 Memcached 服务器这一环节，以便避免对服务器增添额外的负载（此时网络环境可能负载已经过高）。

2.对于写请求（delete，set），web 服务器不与 Memcached 服务器直接通信，而是通过与 Mcrouter 建立 TCP 连接，由 Mcrouter 来保持与 memcached server 之间的连接。这种方案一定程度上可以减少维持 TCP 连接、处理网络 I/O 所需的 CPU 以及内存资源。

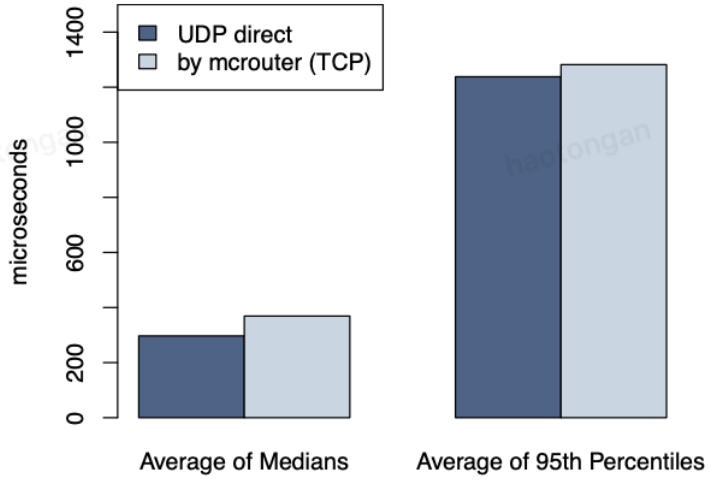


Figure 3: Get latency for UDP, TCP via `mcrouter`

Incast congestion

Memcache 客户端实现流量控制机制来限制 Incast congestion。使用滑动窗口机制来控制未处理请求的数量。当客户端收到一个响应的时候，那么下一个请求就可以发送了。同 TCP 的拥塞控制类似，滑动窗口也是动态改变，请求成功窗口大小就逐渐增加，没有响应就缩小。

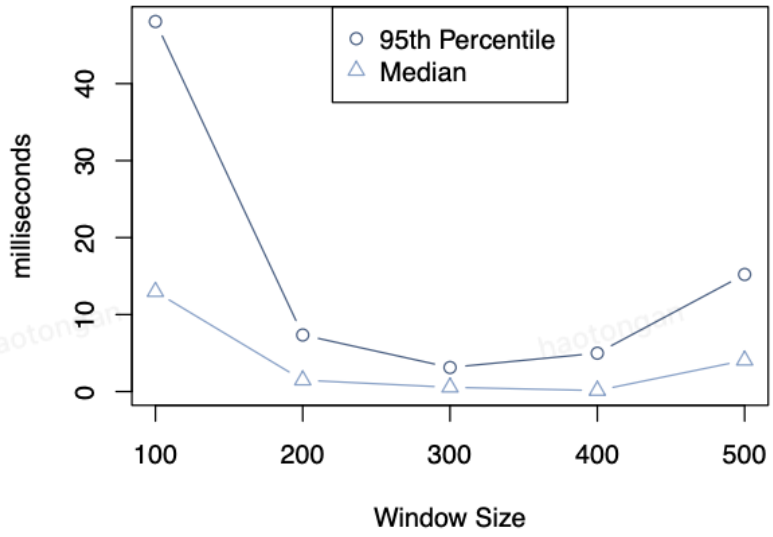


Figure 4: Average time web requests spend waiting to be scheduled

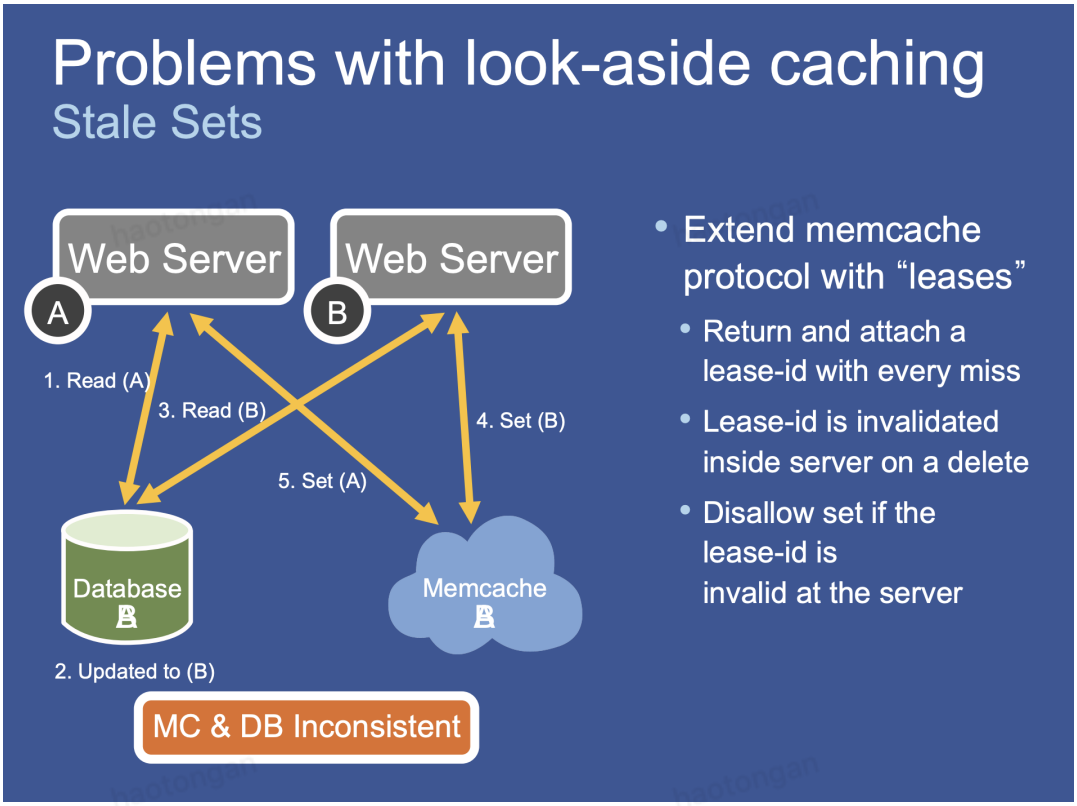
3.2 Reducing Load

使用 Memcache 来减少访问 DB 读数据的频率。当缓存没有命中时，DB 的负载就会升高。下面将描述三种技术，来减少负载。

3.2.1 Leases

引入了一个称为 **leases** 的新机制来解决两个问题：**stale sets** 和 **thundering herds**。

stale sets



look-aside 缓存策略有概率发生数据不一致的情况。

假设两个 web server，s1 和 s2，需要读取同一条数据d，其执行顺序如下：

1. s1 从 memcache 中读取数据 d，发生 cache miss，从数据库读出 d = A；
2. DB 中的数据 d 更新为 B；
3. s2 从 memcache 中读取数据 d，发生 cache miss，从数据库读出 d = B；
4. s2 将 d = B 写入 memcache 中；
5. s1 将 d = A 写入 memcache 中；

leases 机制简单来说，当发生 cache miss 时，Memcached 实例给客户端一个 ***lease***，并将起设置到缓存中。***lease*** 是一个64 bit 的 token，与请求的 key 进行绑定。当客户端写入数据时需要携带这个 ***lease***，Memcached 可以通过这个 ***lease*** 来验证并判断这个数据是否应该被存储。同时如果收到了这个key的删除请求，Memcached 会使之前发放的 ***lease*** 失效。

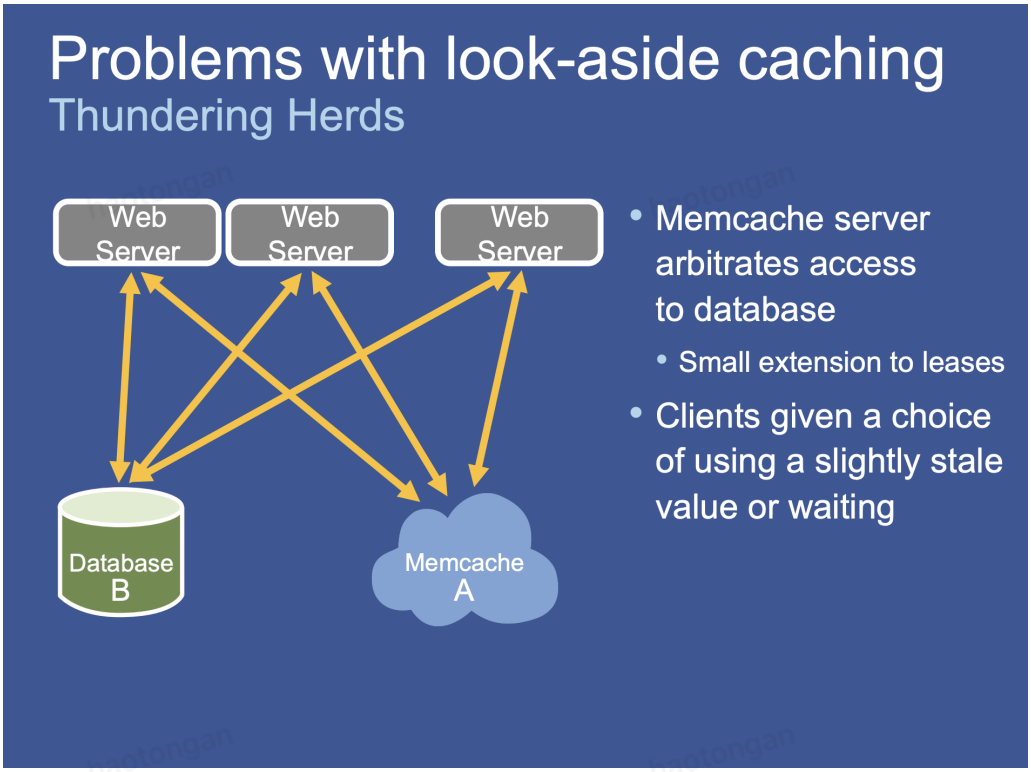
因为存在 **leases** 机制，那么上面读取同一条数据d的情况如下：

1. s1 从 memcache 中读取数据 d，发生 cache miss，得到lease L1，从数据库读出 d = A；
2. DB 中的数据 d 更新为 B，lease L1 失效；
3. s2 从 memcache 中读取数据 d，发生 cache miss，得到lease L2，从数据库读出 d = B；

4. s2 将 d = B 写入 memcache 中，lease L2有效，写入成功；
5. s1 将 d = A 写入 memcache 中，lease L1无效，写入失败；

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。

thundering herds



当某个特定的主键被大量频繁的读写，那么一次 **thundering herds** 就发生了。当某个 hot key 过期失效时，大量的读请求会发生 cahe miss，从而导致数据库负载增高。**leases** 机制可以缓解这个问题。Memcached 服务器可以调节发放 ***lease*** 的频率。默认情况下，同个数据项每10秒只会发放一个 ***lease***，10秒内有请求时，会进行一个特殊响应，客户端可以选择等一会，通常情况下，拥有 ***lease*** 的客户端会在几毫秒内成功的写入数据。当客户端重试之后，数据就能正常返回了。

Stale values

在 **leases** 机制下，通过返回过期的数据，可以进一步减少客户端的等待时间。也就是说，当 cache miss 发生时，一个读请求可以返回一个 ***lease*** ，也可以返回一份标记为过时的数据。应用可以使用过时的数据继续进行处理，而不需要等待从数据库读取的最新数据。

3.2.2 Memcache Pools

使用 Memcache 做为通用的缓存层要求不同的 workloads 共享基础设施。不同应用的 workloads 可能会互相干扰。FB 将 Memcached 服务器分割成独立的池。

- wildcard pool：默认 pool 用来存储大部分数据
- small pool：存储访问频率高但 cache miss 的成本不高的数据
- large pool：存储访问频率低但 cache miss 的成本高的数据

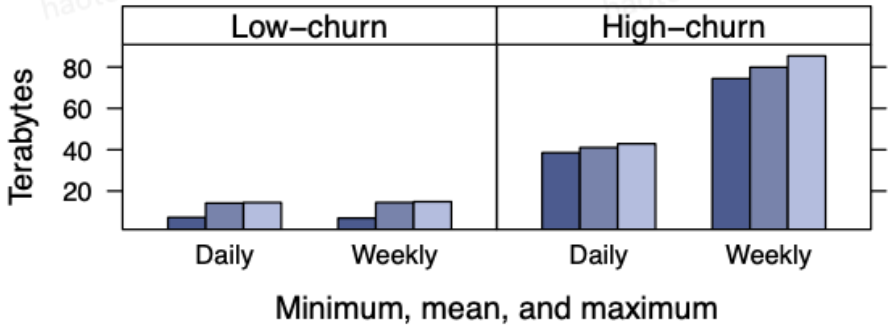


Figure 5: Daily and weekly working set of a high-churn family and a low-churn key family

可以看出，更新频率高的占了大部分，所有数据使用同一个 pool 有可能发生高频率更新的数据将低频率更新的数据替换掉的情况（LRU）。划分不同的 pool 来减少这种冲突影响。

3.2.3 Replication Within Pools

在一些池中，使用复制来改善 Memcached 服务器的延迟和效率。

1. the application routinely fetches many keys simultaneously,
2. the entire data set fits in one or two memcached servers and
3. the request rate is much higher than what a single server can manage.

当以上情况出现时，FB 会选择在池内复制这一类数据。

相比起进一步划分主键空间，FB 更倾向于在实例内进行复制。如果只复制一部分数据，有可能需要通过请求多个实例来获取，这样并不会降低单台实例处理请求的数量。

3.3 Handling Failures

如果无法从 Memcache 中读取数据，这将会导致后端服务负载增加，对于这种情况，FB 考虑一下两种情况。

1. 由于网络或服务器故障，少量的主机无法连接；
2. 集群内相当大比例的服务器停机或故障；

如果整个的集群不得不离线，FB 转移用户的 web 请求到别的集群，这样将会有效地转移 Memcache 所有的负载。如果只是少量主机因为网络原因等失联，依赖于一种自动恢复机制，这不是即时的，通常恢复需要花费几分钟时间，但这段期间也有可能会产生连锁故障。因此 FB 引入了一个机制进一步将后端服务从故障中隔离开来。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%。

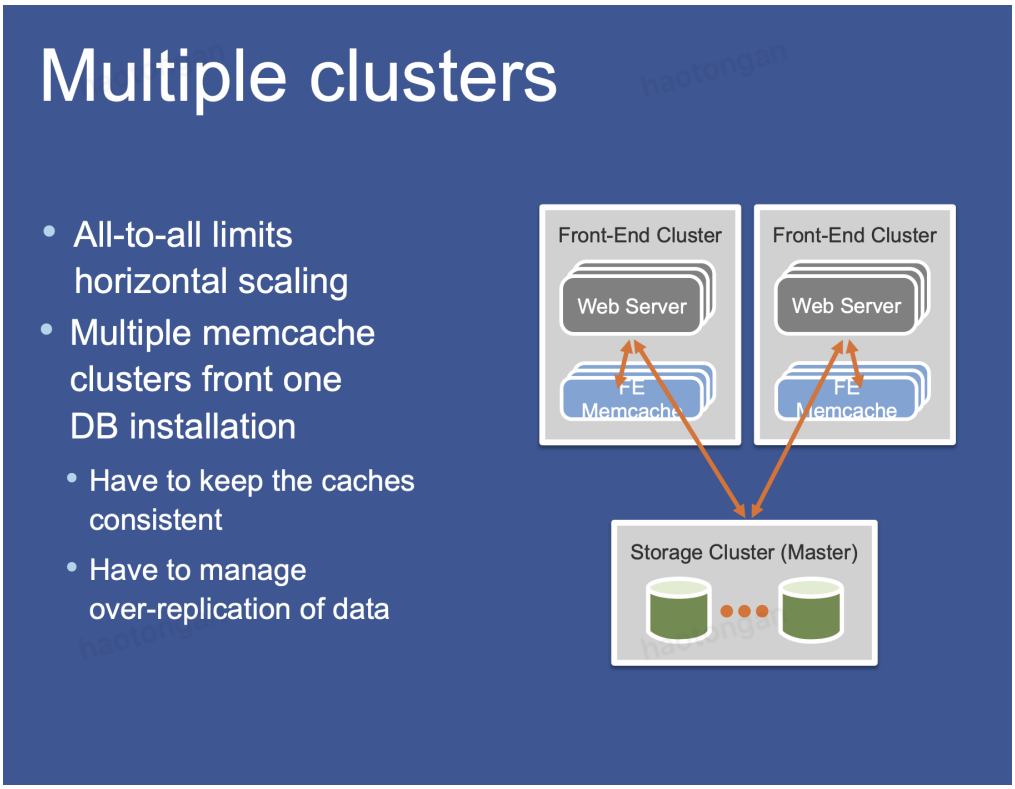
通常来说，每个失败的请求都会导致对后端储存的一次存取，潜在地将会使后端过载。使用 Gutter 存储这些结果，很大部分失败被转移到对 Gutter 池的存取，因此减少了后端存储的负载。在实践中，Gutter 机制的存在，每天减少了约99%失败请求，其中10%-25%的失败转化为缓存命中。如果一台 Memcached 服务器整个发生故障，在4分钟之内，Gutter 池的命中率将会普遍增加到35%，经常会接近50%。因此对于由于故障或者小范围网络事故造成的一些 Memcached 服务器不可达的情况，Gutter 将会保护后端存储免于流量激增。

4 In a Region: Replication

随着需求的增长，增加更多的机器来部署 Web server 和 Memcached server 来扩展集群，但是单纯地扩展系统并不能解决所有问题。

1. 更多的 Web server 的加入来处理增长的用户流量，高请求频率的数据项只会变的更加流行；
2. 随着 Memcached server 的增加，client 和 server 更多的通信会使得 Incast 拥塞变的更严重；

将 Web server 和 Memcached server 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。



4.1 Regional Invalidations

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。

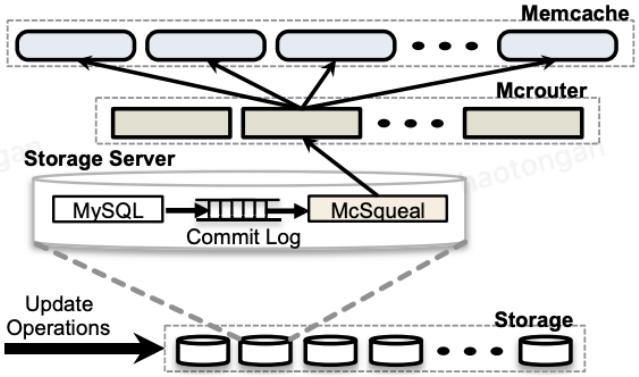
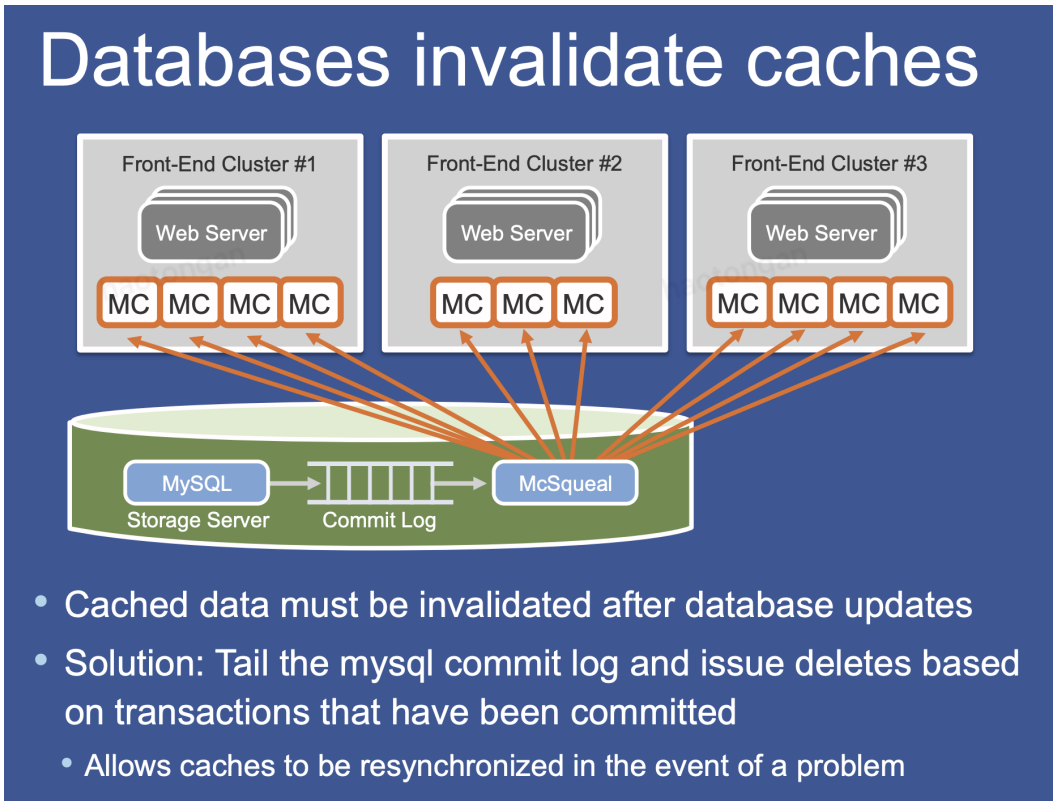


Figure 6: Invalidation pipeline showing keys that need to be deleted via the daemon (`mcsqueal`).

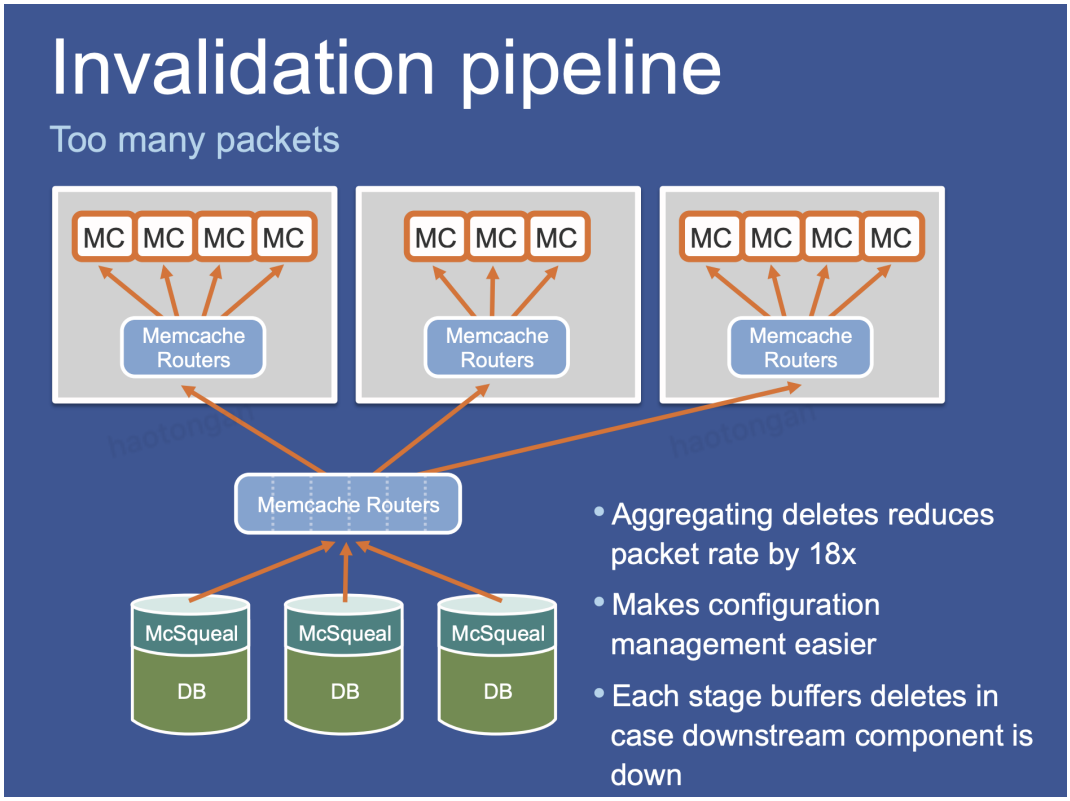
FB 在每一台数据库上部署了 **Invalidation daemons**（mcsqueal）。每个 daemon 检查数据库提交的SQL语句，提取任意的删除命令，并且将删除命令广播到 Region 内所有的 Memcached 集群。

Reducing packet rates

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。大量的数据库中 mcsqueal 和 Memcached server 跨集群通信，会产生网络问题（fanout）。



通过指定一组运行 Mcrouter 实例的专用服务器，Invalidation daemons 批量发送删除操作到专用 Mcrouter 服务器，再由 Mcrouter 就批量操作中分离出单独的删除操作，将删除命令路由到正确的 Memcached 服务器。



Invalidation via web servers

通过 Web server 广播删除命令到所有 Front-end cluster 服务器更简单。但这个方法存在两个问题。

1. 因为 Web server 在批处理删除命令时没有 mcsqueal 有效率，具有更高的成本；

2. 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；

4.2 Regional Pools

如果用户的请求被随机路由到所有可获得的 Front-end cluster 中，那么集群中缓存的数据将会大致一致。过度复制数据会使内存使用效率降低，特别是对很大的、很少存取的数据项。通过减少副本的数量，使多个前端集群共享同一个 Memcached 服务器集合，FB 成此为 region 池。

	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

Table 1: Deciding factors for cluster or regional replication of two item families

类型 A 数据项相比类型 B 来说更大，但是由于存取的频率很高，所以不把 A 放进 Region pool 中，而 B 的存取频率和使用的用户数量都较低，适合放进 Region pool 中。

4.3 Cold Cluster Warmup

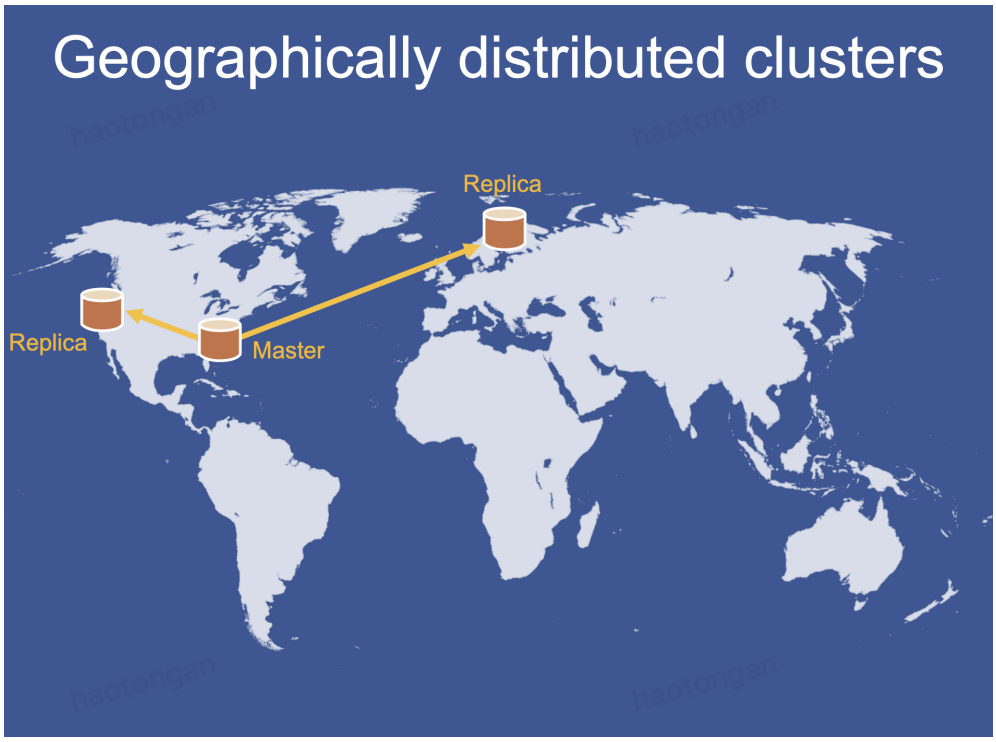
由于存在的集群发生故障或者进行定期的维护，或者增加新的集群上线，此时缓存命中率会很低，大量的 cache miss 会导致后端服务负载增加。一个称作 Cold Cluster Warmup 的系统可以缓和这种情况，这个系统使 “Cold Cluster” （也就是具有空缓存的集群）中的客户端从“Warm Cluster” （也就是具有正常缓存命中率的集群）中检索数据而不是从持久化存储。这利用到了前面提到的跨集群的数据复制，使用这个系统可以使冷集群在几个小时恢复到满负载工作能力而不是几天。

5 Across Regions: Consistency

将数据中心分布到广泛的地理位置具有很多优势。

1. 将web服务器靠近终端用户可以极大地较少延迟；
2. 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；
3. 新的位置可以提供更便宜的电力和其它经济上的支持；

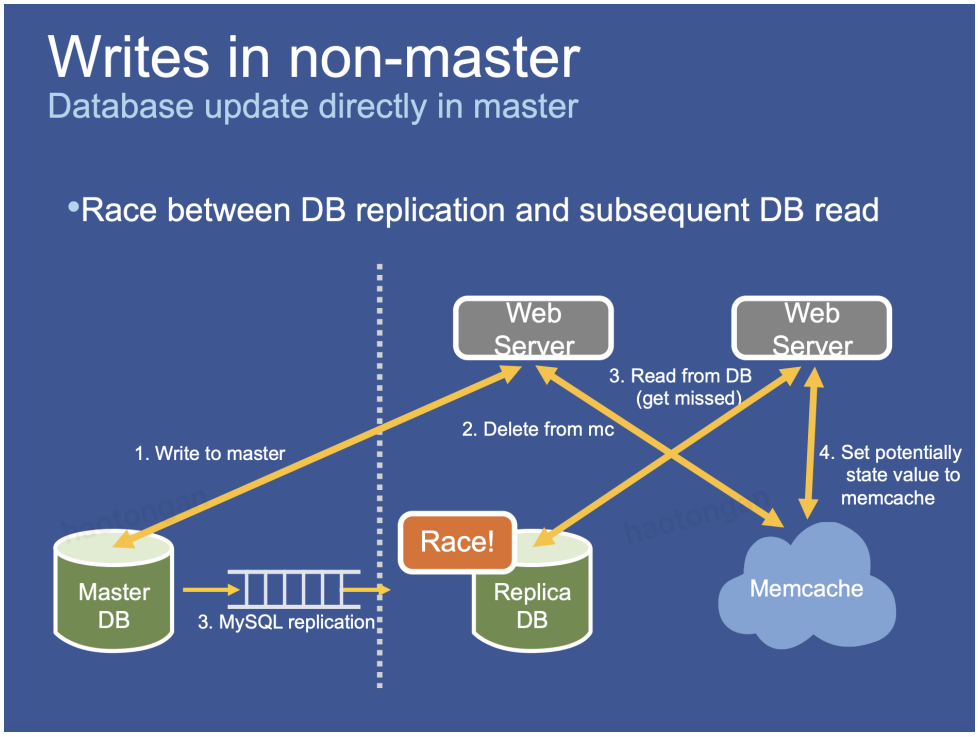
FB 通过部署多个 region 来获得这些优势。 region 包含一个存储集群和多个 front-end 集群。我们指定一个 region 持有主数据库，别的 region 包含只读的副本从库，依赖 MySQL 的复制机制来保持从库与主库的同步。基于这样的设计，Web server 无论访问本地 Memcached 服务器还是本地数据库副本的延迟都很低。



Writes from a master region

考虑一个 Web server 已完成 Master region 数据库修改的并试图使现在过时的数据失效。这种情况可能引发数据的不一致性。在 Master region 内是缓存数据失效是安全的。但是，让 Replica region 中的数据无效可能为时过早，因为主库的数据更改可能尚未同步到从库中。接下来对 Replica 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，FB实现了上面提过的 mcsqueal 机制。

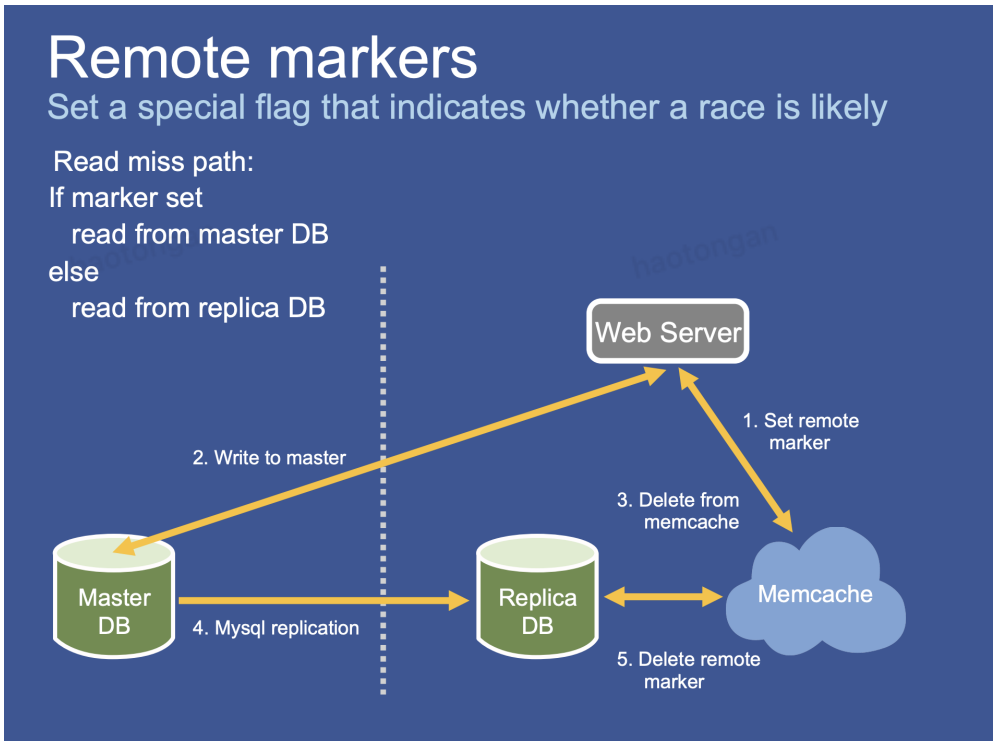
Writes from a non-master region



现在考虑当复制滞后非常大的时候，用户从 Replica 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。

1. Replica region 中的 web server s1 写入数据到 master DB；
2. s1 将本地 memcache 中的数据删除；
3. Replica region 中的 web server s2 从 memcache 中读取数据发生 cache miss，从本地 DB 中获取数据；
4. s1 写入的数据从 master DB 中同步到 replica DB，并通过 mcsqueal 机制将本地 memcache 中的数据删除；
5. web server s2 将其读到的数据写入 memcache 中；

只有当数据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，后续请求将会导致读取过期数据并且缓存。



FB 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 Master region。

当 Replica region 的 Web server 需要写入某数据时：

1. 在本地 memcache 上打上 remote marker，数据 d 标记为 rd；
2. 将 d 写入到 master DB 中；
3. 将 d 从 memcache 中删除 (rd 标记不删除)
4. 等待 Master DB 将数据同步到 Replica region 中的 replica DB 中；
5. Replica region 中的 Replica DB 通过 mcsqueal 删除本地 memcache 中数据 d 的标记 rd

当 Replica region 的 Web server 想要读取数据 d 发生 cache miss 时：

1. 如果 memcache 中数据 d 带了 rd，则从 Master DB 中读取数据；

2. 如果 memcache 中数据 d 没有 rd，则直接从本地的 Replica DB 中读取数据；

Operational considerations

略

6 Single Server Improvements

All-to-all 的通信模式意味着单个服务器可能成为集群的瓶颈。改进单服务器缓存性能。

6.1 Performance Optimizations

对于单线程，使用固定大小的哈希表的 memcached，FB 做出以下几点优化。

1. 允许自动扩展哈希表，以避免查找时间漂移到O(n)；
2. 从单线程变成多线程，通过全局锁来保护数据结构；
3. 为每个线程提供单独的 UDP 端口，提高通信效率；

Get Performance

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，预先填充了32 byte 的缓存数据，然后测量命中的性能。

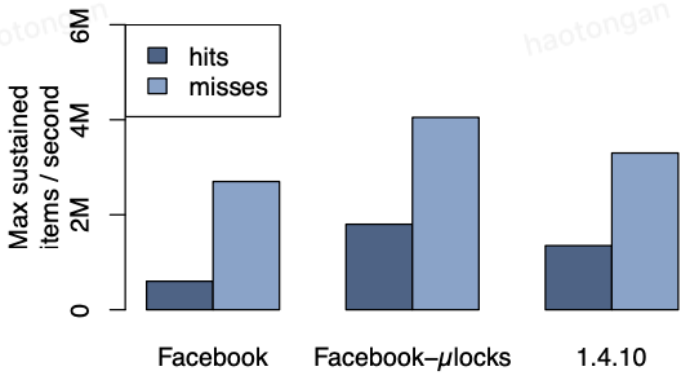


Figure 7: Multiget hit and miss performance comparison by memcached version

更细的锁粒度，意味着更高的性能。由于缓存命中时返回内容需要构建和传输，而 miss 则不用太多的操作，所以，更多的命中也代表着更多的耗时。

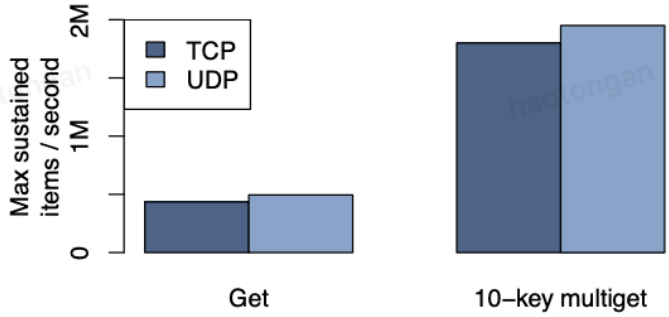


Figure 8: Get hit performance comparison for single gets and 10-key multigets over TCP and UDP

使用UDP代替TCP的性能影响。FB 实验发现 UDP 实现的性能在单主键获取情况下超出 TCP 实现13%，在10-key 获取的情况下超出8%。

6.2 Adaptive Slab Allocator

Memcached 使用 slab 分配器来管理内存。分配器将内存组织到 slab class 中，每个 slab class 都包含预分配的、大小一致的内存块。Memcached 将项存储在尽可能小的 slab class 中，可以容纳元数据、键和值。

FB 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 class 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 class 中的最近最少使用的数据项的使用时间多至少20%，那么就说明这个 slab class 需要更多内存。如果找到了一个这样的 slab class，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 class。

6.3 The Transient Item Cache

因为 memcached 支持过期时间，通常来说，数据项条在它们过期之后仍可以驻留在内存中（延时删除）。 当数据项被请求时或者当它们到达 LRU 的尾端时，memcached 检查其过期时间后才会进行删除操作。 这种模式会使得那些偶尔活跃的短期键值长期占据内存空间，直到它们到达 LRU 的尾部。

FB 引入一种混合模式，对大多数键值使用延时删除，而对过期的短期键值则立即删除。使用链表，按秒索引构建一个缓冲区，每一秒，缓冲区头部中的所有项都会被清除。

6.4 Software Upgrades

FB 修改了 memcached，将其缓存的值和主要数据结构存储在 System V 共享内存区域中，以便数据可以在软件升级期间保持活跃，从而最大限度地减少中断。

7 Memcache Workload

在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 Measurements at the Web Server

Fanout

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。对于流行页面的请求分布，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。

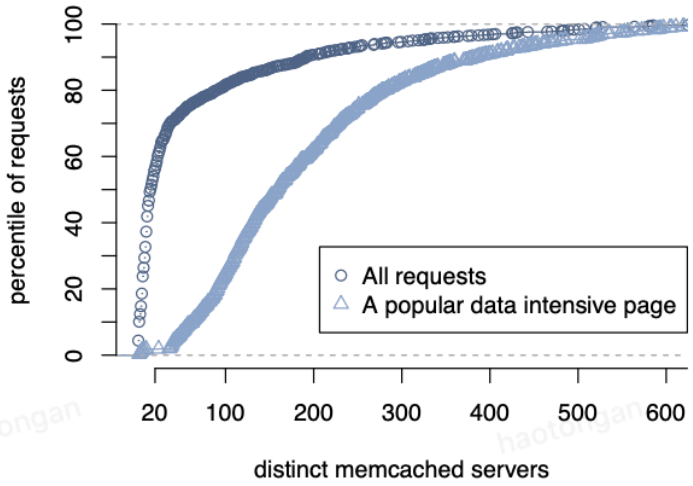


Figure 9: Cumulative distribution of the number of distinct `memcached` servers accessed

Response size

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。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。

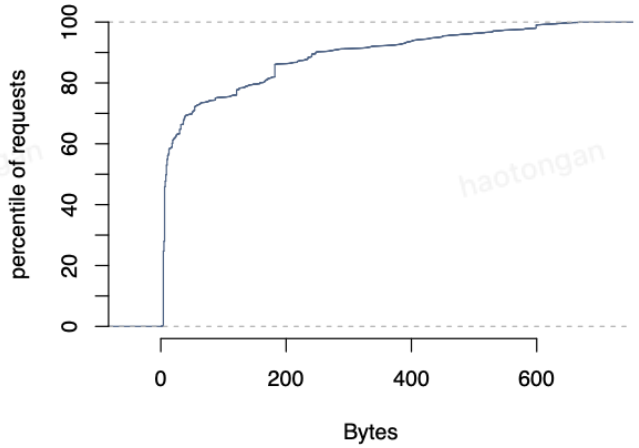


Figure 10: Cumulative distribution of value sizes fetched

Latency

测量从 memcache 请求数据的往返延迟，其中包括路由请求和接收响应的成本、网络传输时间以及反序列化和解压缩的成本。在7天内，请求延迟的中位数为333µs，而第75和第95百分位分别为475µs和1.135ms。空闲web server 的端到端延迟中位数为178µs，而p75和p95分别为219µs和374µs。在第95百分位上延迟的巨大差异是由处理数据量大的响应和等待可运行线程调度引起的。

7.2 Pool Statistics

1. wildcard：默认；
2. app：专门设定给特定应用的；
3. replicated pool：给存取频繁的数据的；
4. regional pool：很少存取的数据的；

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

Table 2: Traffic per server on selected `memcache` pools averaged over 7 days

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

Table 3: Distribution of item sizes for various pools in bytes

面对不同工作负载，划分不同的 pool，使用不同的策略进行管理是很有必要的。

7.3 Invalidation Latency

当删除操作发起后到集群中 key 失效的延迟，采样统计如下：

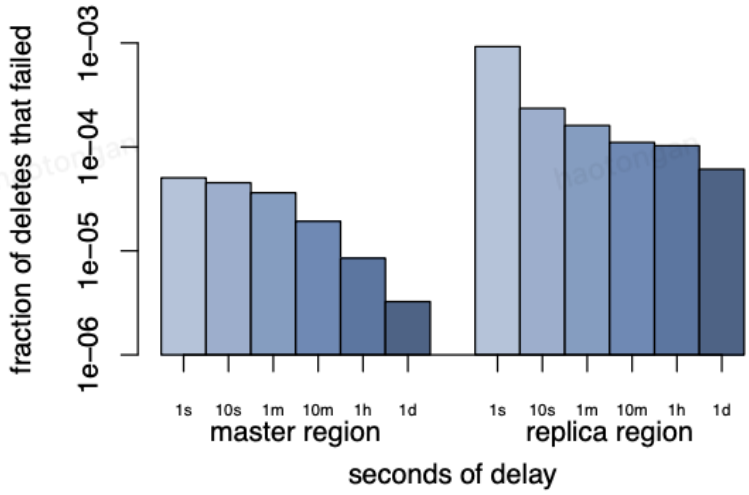


Figure 11: Latency of the Delete Pipeline

Master region 删除的延迟相比 Replica region 删除的延迟要好很多。

8 Related Work & 9 Conclusion

略

Usenix 2013: Scaling Memcache at Facebook, video, slides, paper