1. **背景**

使用guava CACHE缓存命中率逐渐下降。 Why?

1. **guava原理**

参见[《cache-1-guava》](cache-1-guava.docx)

1. **常见缓存算法与实践**
2. **LRU**

* 优点
  + 结构简单
  + 占用内存少
  + 善于处理稀疏请求（Sparse burst）
* 缺点
  + 由于没有统计概念，可能会将一些高频数据错误抛弃

1. **LFU**

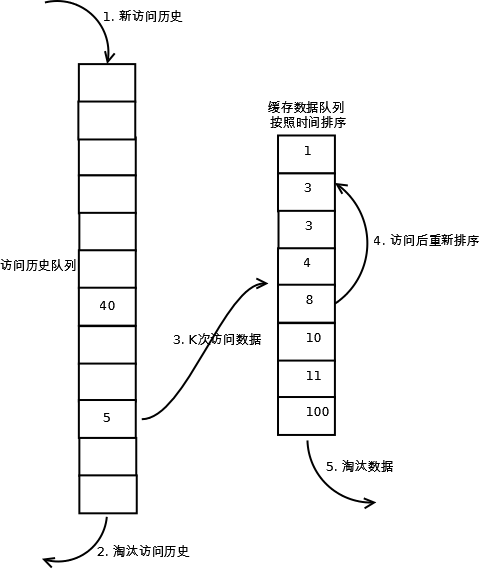
* 优点
  + 可以统计高频数据
* 缺点
  + 占用内存多
  + 高频数据难以过期或移除
  + 无法处理稀疏请求（Sparse burst），因为可能始终无法累计到一定数量存入缓存中

1. **LRU-2和LRU-k**

LRU-K中的K代表最近使用的次数，因此LRU可以认为是LRU-1。LRU-K的主要目的是为了解决LRU算法“缓存污染”的问题，其核心思想是将“最近使用过1次”的判断标准扩展为“最近使用过K次”。

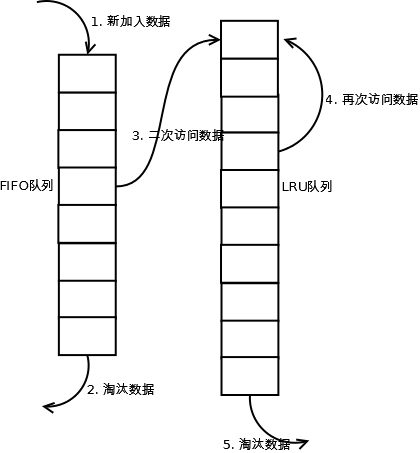
相比LRU，LRU-K需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到K次的时候，才将数据放入缓存。当需要淘汰数据时，LRU-K会淘汰第K次访问时间距当前时间最大的数据。

历史队列并不是缓存本身



通常我们使用LRU-2，因为更多的计数需要更多的内存来存储。同时使得LRU-2历史队列成为访问一次的真正缓存队列，这就是2-queue，

当数据第一次访问时，2Q算法将数据缓存在FIFO队列里面，当数据第二次被访问时，则将数据从FIFO队列移到LRU队列里面，两个队列各自按照自己的方法淘汰数据。详细实现如下：



1. **Redis中LRU近似算法**
2. **使用场景**

redis过期键有惰性、定期的删除策略，也有maxmemory设置，具体参见[2]。其中maxmemory设置之后，client在执行耗费内存命令redisCommand（flags会标识是否为耗费内存操作）会首先判断是否达到maxmemory，如果达到则需要清理过期键或释放内存。maxmemory释放内存的方式主要有三种[3]：

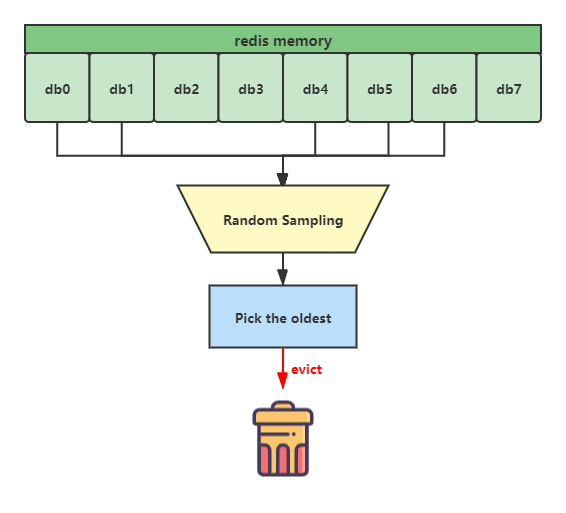
* lru：volatile代表设置了过期时间，allKeys则是所有key
* random：volatile代表设置了过期时间，allKeys则是所有key
* ttl：即将过期

1. **优化原因**

这里的lru其实是近似lru，redis并没有在内存中按照双端链表的形式构建lru列表，而只有原有redisDb中的数据字典redisDb.dict和过期时间字典redisDb.expire，这样只是为了减少内存开销。

1. **优化措施**

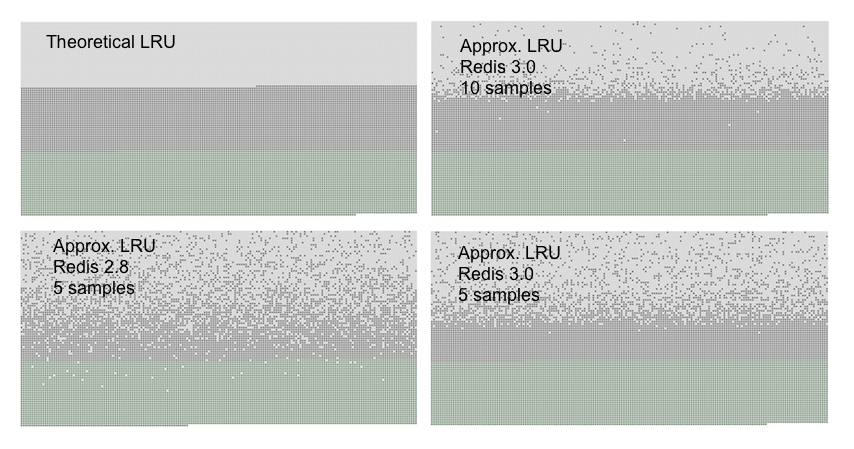
根据设定的maxmemory\_samples，我们从dict数据字典中随机取出maxmemory\_samples个抽样key，然后我们找到这些key中那个最老没有调用的执行删除（samples中的lru）。maxmemory\_samples越大越接近理论上的lru。



1. **测试结果**

Redis 不采用真正的 LRU 实现的原因是为了节约内存使用。虽然不是真正的 LRU 实现，但是它们在应用上几乎是等价的。下图是 Redis 的近似 LRU 实现和理论 LRU 实现的对比：

测试开始首先在 Redis 中导入一定数目的 key，然后从第一个 key 依次访问到最后一个key，因此根据 LRU 算法第一个被访问的 key 应该最新被置换，之后再增加 50% 数目的 key，导致 50% 的老的 key 被替换出去。



在上图中你可以看到三种类型的点，组成三种不同的区域：

* 淡灰色的是被置换出去的key
* 灰色的是没有被置换出去的key
* 绿色的是新增加的key

理论 LRU 实现就像我们期待的那样，最旧的 50% 数目的 key 被置换出去，Redis 的 LRU 将一定比例的旧 key 置换出去。可以看到在样本数为 5 的情况下，Redis3.0 要比 Redis2.8 做的好很多，Redis2.8 中有很多应该被置换出去的数据没有置换出去。在样本数为10的情况下，Redis3.0 很接近真正的 LRU 实现。

LRU 是一个预测未来我们会访问哪些数据的模型，如果我们访问数据的形式接近我们预想——幂律，那么近似 LRU 算法实现将能处理的很好。在模拟测试中我们可以发现，在幂律访问模式下，理论 LRU 和 Redis 近似 LRU 的差距很小或者就不存在差距。

如果你将 maxmemory-samples 设置为 10，那么 Redis 将会增加额外的 CPU 开销以保证接近真正的 LRU 性能，可以通过检查命中率来查看有什么不同。

1. **Mysql innodb**

参见[2]《mysql之innodb-9-lru优化》

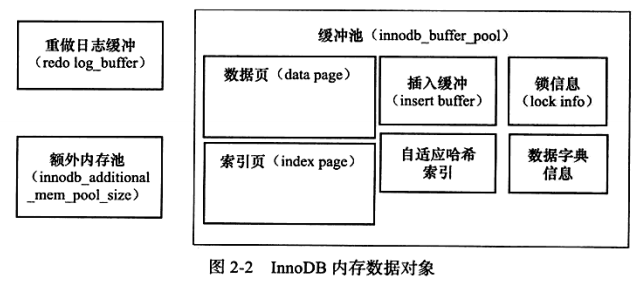
1. **使用场景**

Innodb是基于磁盘存储的，在CPU中，使用缓冲池几乎来提高数据库整体性能。

缓存池简单来说就是一块内存区域，在数据库中读取页的操作，首先将该页放置于缓冲池中，下次再读取相同的页时，就可以通过缓冲池读取。

对于数据页中的修改操作，会先修改缓冲池中的页为脏页，再以一定频率刷新到磁盘上，其刷新频率由checkpoint机制决定（参见3））。这样可以减少随机读写，提高顺序读写。

具体来看缓冲池缓冲的数据页有：索引页、数据页、undo页、insert buffer、自适应哈希索引和Innodb的锁信息、数据字典信息、double write buffer(2MB)等。



缓冲池的更新一般使用LRU列表来管理缓冲页，一般需要保证100个空页，否则触发LRU机制。如果LRU列表中的页被修改，被称为脏页，flush列表中的页就是脏页。LRU列表用于管理缓冲池页的可用性，flush列表用于将脏页刷会磁盘，二者互不影响，脏页既存在于LRU列表，也存在于FLUSH列表。

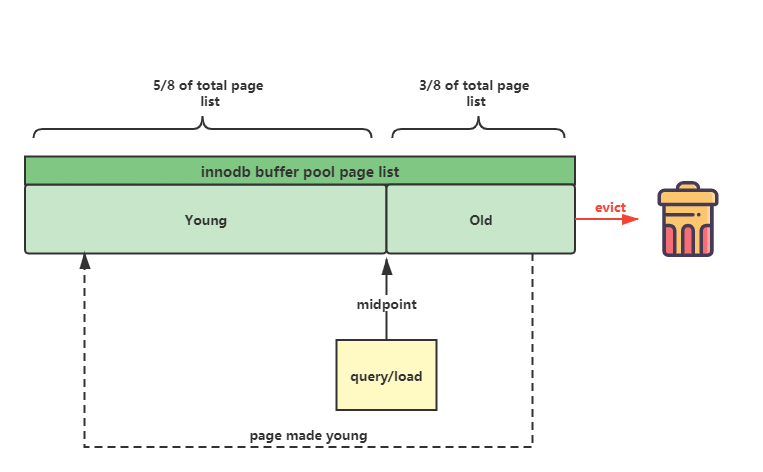
我们可以通过show engine innodb status中的modified db pages可以看出FLUSH列表中的脏页大小。

缓冲池中的页大小默认16 kb。操作系统的页大小可能是2kb或4kb。

1. **优化原因**

数据库中经常有根据索引或数据的扫描操作，这类操作通常需要访问表中许多页，而这些操作的数据通常又只在这次查询使用，不是活跃的热点数据，如果直接把读到的页放到lru首部，显然会将真正的热点缓存页刷出去。

1. **优化措施——分代**



innodb将LRU进行分代，midpoint位于距离tail 3/8，以midpoint为界，向上到head处为热点数据，表示为new，线下到tail处为old代。

新数据插入到lru双向链表的midpoint处，而不是队列head，同时引入innodb\_old\_blocks\_time，用于表示新页读取到midpoint处多久才被放置到lru首部。当页从old部分加入到new部分，称为page made young。如果因为innodb\_old\_blocks\_time设置导致没有从old部分移动到new部分称之为page not made young。

1. **Linux 页缓存——LRU-2**

linux中lru是一对双向链表：active和inactive，这两个链表是进行页面回收的关键数据结构，经常使用且活跃的位于active链表，不活跃的可以被回收的位于inactive链表。那些最近最少使用的页面会被逐个放到 inactive 链表的尾部。进行页面回收的时候，Linux 操作系统会从 inactive 链表的尾部开始进行回收。页面会在这两个链表上来回移动，至于是否获取的判断标识如下：

* PG\_active：标识该页为活跃页；
* PG\_referenced：标识该页最近是否被访问，每次被访问都会被置位，如果一段时间后页面没有被访问，系统会清除页面的PG\_referenced。

Linux 必须同时使用这两个标志符来判断页面的活跃程度，判断标准如下：

1. **进入inactive**

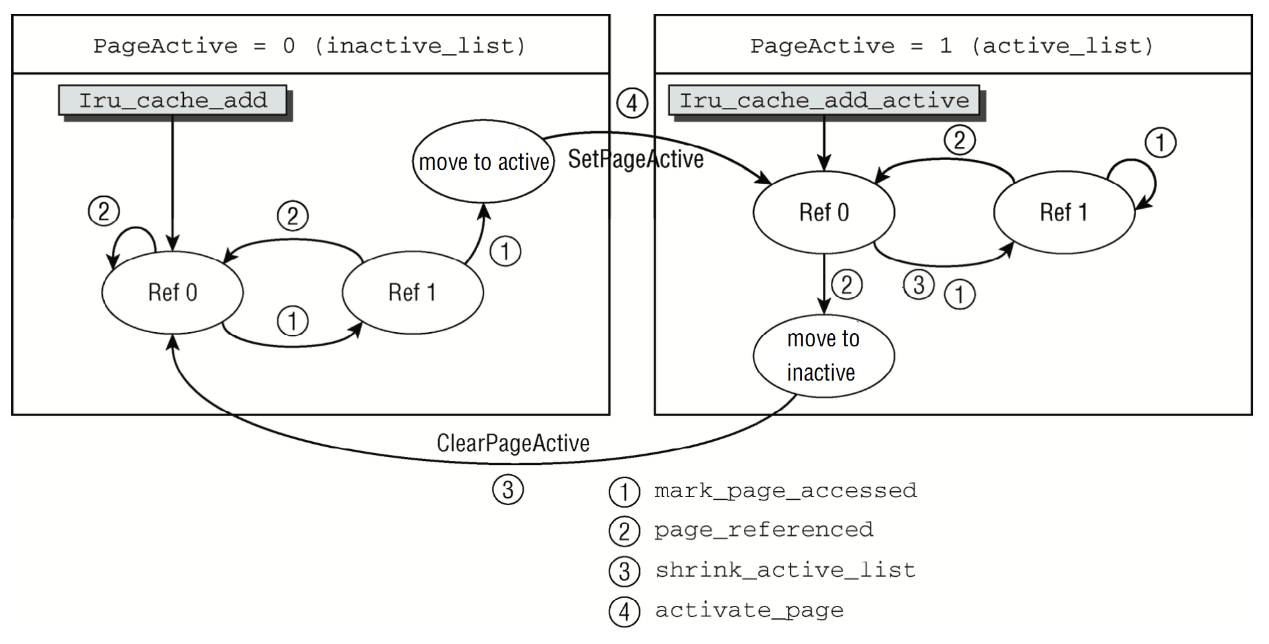
页面访问时，如果发现当前页位于inactive链表，且PG\_referenced标识未置位，则PG\_referenced标识置位，还存放在inactive。

1. **inactive -> active**

页面访问时，如果发现当前页位于inactive链表，且PG\_referenced标识已置位，则认为该页经常访问，将其置位PG\_active，移动到active链表并清除PG\_referenced标识；

1. **active -> inactive**

在active链表上，其PG\_active被置位，如果给定时间PG\_referenced未被置位，则清除PG\_active标识，将其移动到inactive中。



1. **Solaris ZFS——ARC**
2. **介绍**

Solaris ZFS 中实现的ARC(Adjustable Replacement Cache)读缓存淘汰算法真是很有意义的一块软件代码。它是基于IBM的Megiddo和Modha提出的ARC（Adaptive Replacement Cache）淘汰算法演化而来的。但是ZFS的开发者们对IBM 的ARC算法做了一些扩展，以更适用于ZFS的应用场景。ZFS ARC的最早实现展现在FAST 2003的会议上，并在杂志《;Login:》的一篇文章中<https://www.usenix.org/publications/login/2003-08/index.html>被详细描述。

在一些文件系统缓存中实现的标准的LRU淘汰算法是有一些缺点的。例如，它们对扫描读模式是没有抵抗性的。但你一次顺序读取大量的数据块时，这些数据块就会填满整个缓存空间，即使它们只是被读一次。当缓存空间满了之后，你如果想向缓存放入新的数据，那些最近最少被使用的页面将会被淘汰出去。在这种大量顺序读的情况下，我们的缓存将会只包含这些新读的数据，而不是那些真正被经常使用的数据。在这些顺序读出的数据仅仅只被使用一次的情况下，从缓存的角度来看，它将被这些无用的数据填满。

另外一个挑战是：一个缓存可以根据时间进行优化（缓存那些最近使用的页面），也可以根据频率进行优化（缓存那些最频繁使用的页面）。但是这两种方法都不能适应所有的workload。而一个好的缓存设计是能自动根据workload来调整它的优化策略。

1. **本质**

* Adaptive Replacement Cache本身就是2-queue或者LRU-2的实现
* 只不过增加了ghost list来记录之前evict的历史情况，从而在ghost list范围内动态调整LRU list或LFU list大小，以平衡recency-biased或frequency-biased的workload。
* ARC的动态调整简单有效
* ARC没有真正意义上实现frequency-biased的缓存

1. **ARC的内部工作原理**

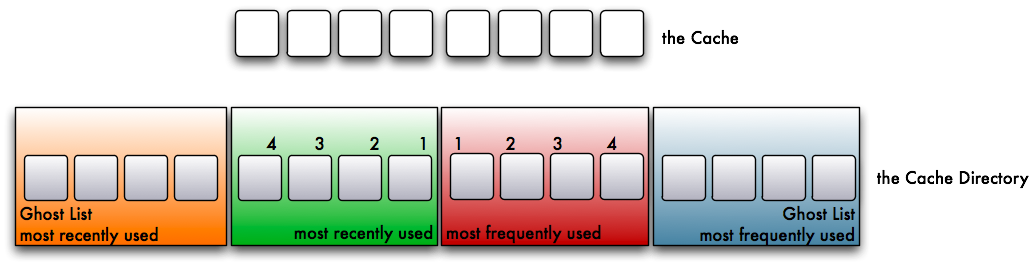
在ARC原始的实现（IBM的实现）和ZFS中的扩展实现都解决了这些挑战，或者说现存问题。我将描述由Megiddo和Modha提出的Adaptive Replacement Cache的一些基本概念，ZFS的实现版本作为这个实现机制的一个扩展来介绍。这两种实现（原始的Adaptive Replacement Cache和ZFS Adjustable Replacement Cache）共享一些基本的操作原理，所以我认为这种简化是一种用来解释ZFS ARC切实可行的途径。

首先，假设我们的缓存中有一个固定的页面数量。简单起见，假设我们有一个8个页面大小的缓存。为了是ARC可以工作，在缓存中，它需要一个2倍大小的管理表。这个管理表分成4个链表。头两个链表是显而易见的：

1. 最近最多使用的页面链表 （LRU list）
2. 最近最频繁使用的页面链表（LFU list）

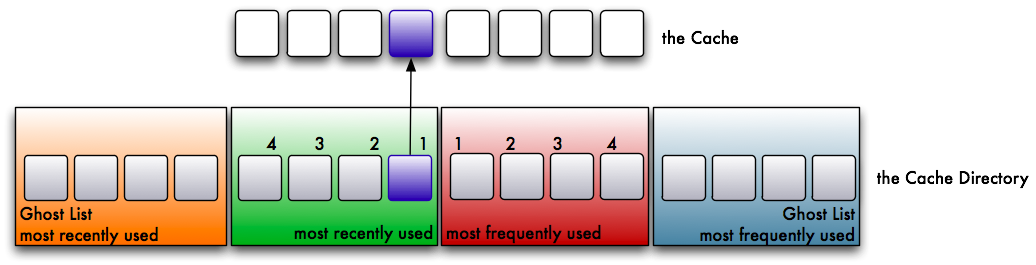
另外两个链表在它们的角色上有些奇怪。它们被称作ghost链表。那些最近被淘汰出去的页面信息被存储在这两个链表中：

1. 存储那些最近从最近最多使用链表中淘汰的页面信息 （Ghost list for LRU）
2. 存储那些最近从最近最频繁使用链表中淘汰的页面信息（Ghost list for LFU）

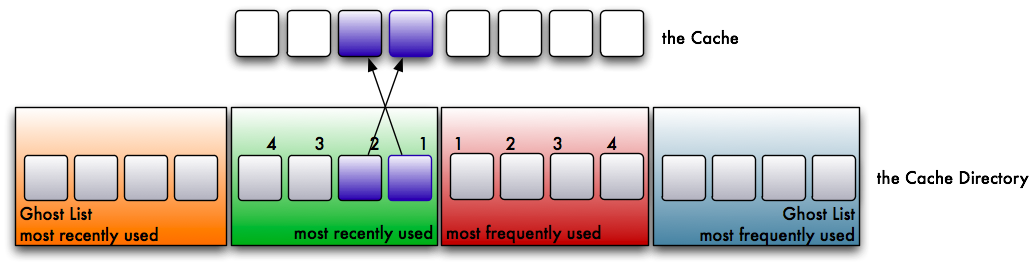


这两个ghost链表不储存数据（仅仅储存页面信息，比如offset，dev-id），但是在它们之中的命中对ARC缓存工作的行为具有重要的影响，我将在后面介绍。那么在缓存中都发生了什么呢？

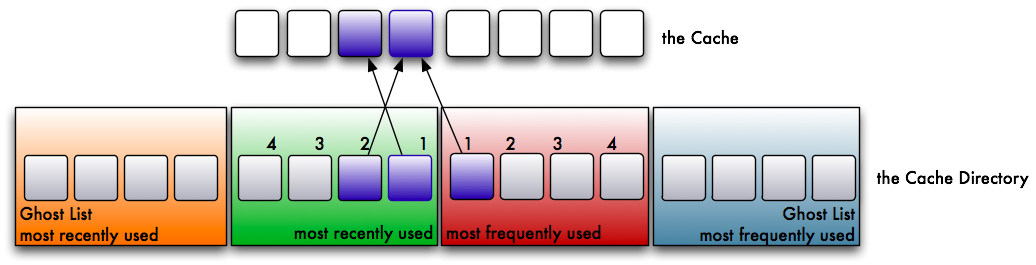
假设我们从磁盘上读取一个页面，并把它放入cache中。这个页面会放入LRU 链表中。



接下来我们读取另外一个不同的页面。它也会被放入缓存。显然，他也会被放入LRU 链表的最近最多使用的位置（位置1）：

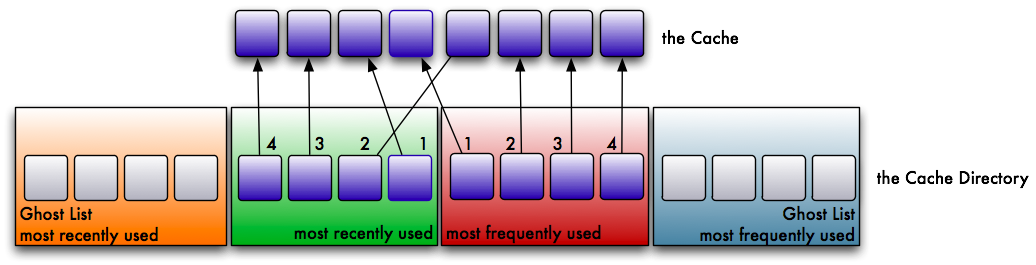


好，现在我们再读一次第一个页面。我们可以看到，这个页面在缓存中将会被移到LFU链表中。所有进入LRU链表中的页面都必须至少被访问两次。无论什么时候，一个已经在LFU链表中的页面被再次访问，它都会被放到LFU链表的开始位置（most  frequently used）。这么做，那些真正被频繁访问的页面将永远呆在缓存中，不经常访问的页面会向链表尾部移动，最终被淘汰出去。

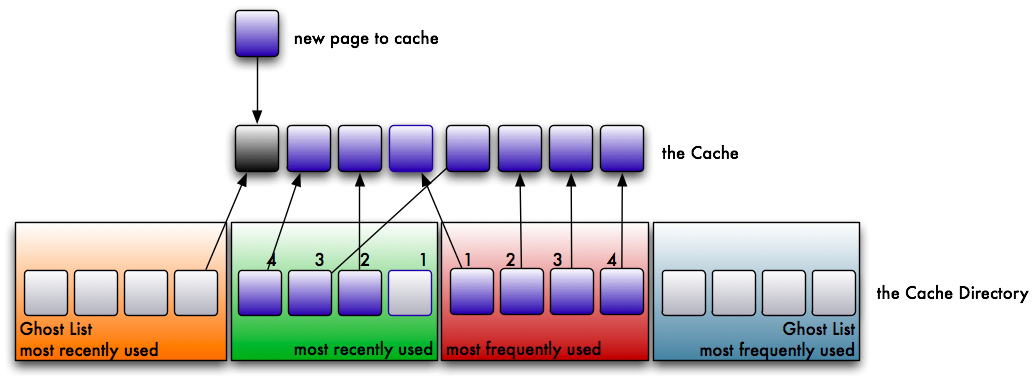


随着时间的推移，这两个链表不断的被填充，缓存也相应的被填充。这时，缓存已经满了，而你读进了一个没有被缓存的页面。所以，我们必须从缓存中淘汰一个页面，为这个新的数据页提供位置。这个数据页可能刚刚才被从缓存中淘汰出去，也就是说它不被缓存中任何的非ghost链表引用着。

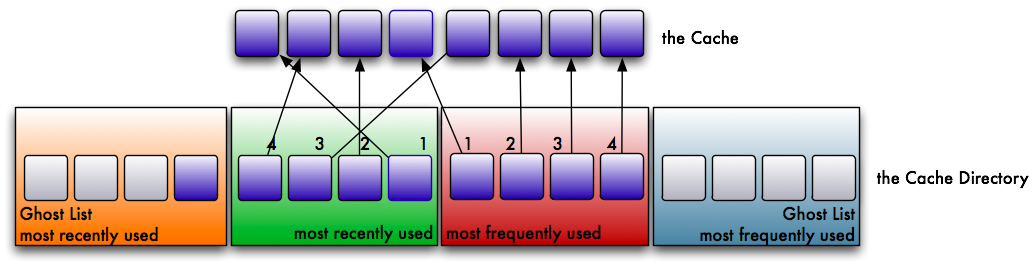
假设LRU链表已经满了：



这时在LRU链表中，最近最少使用的页面将会被淘汰出去。这个页面的信息会被放进LRU ghost链表中。

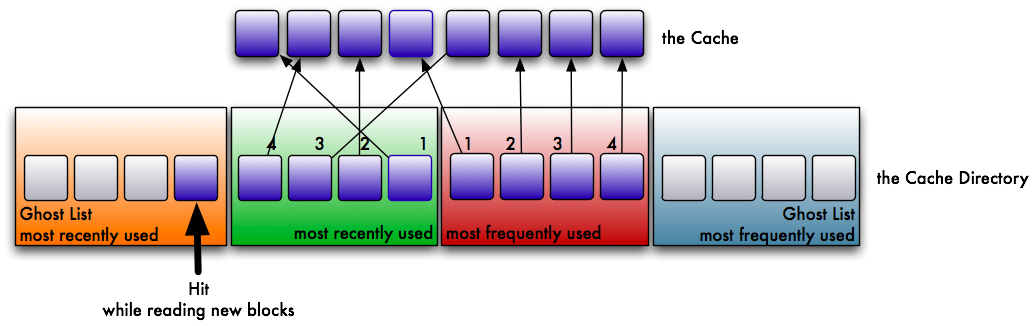


现在这个被淘汰的页面不再被缓存引用，所以我们可以把这个数据页的数据释放掉。新的数据页将会被缓存表引用。

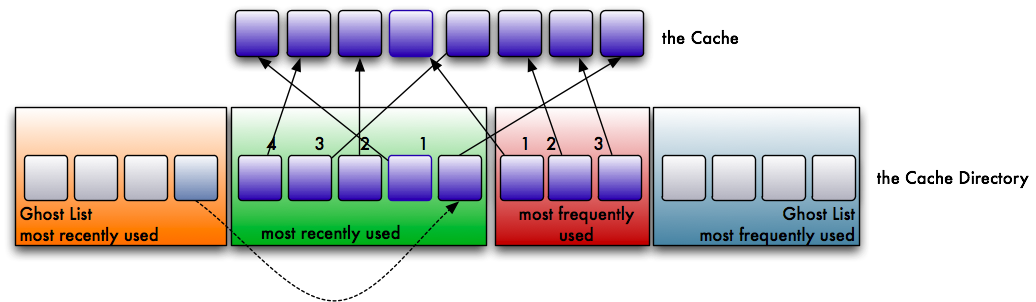


随着更多的页面被淘汰，这个在LRU ghost中的页面信息也会向ghost链表尾部移动。在随后的一个时间点，这个被淘汰页面的信息也会到达链表尾部，LRU链表的下一次的淘汰过程发生之后，这个页面信息也会从LRU ghost链表中移除，那是就再也没有任何对它的引用了。

好的，如果这个页面在被从LRU ghost链表中移除之前，被再一次访问了，将会发生什么？这样的一个读将会引起一次幽灵（phantom）命中。由于这个页面的数据已经从缓存中移除了，所以系统还是必须从后端存储媒介中再读一次，但是由于这个幽灵命中，系统知道，这是一个刚刚淘汰的页面，而不是第一次读取或者说很久之前读取的一个页面。ARC用这个信息来调整它自己，以适应当前的I/O模式（workload）。



很显然，这个迹象说明我们的LRU缓存太小了。在这种情况下，LRU链表的长度将会被增加一。显然，LFU链表的长度将会被减一。



但是同样的机制存在于LFU这边。如果一次命中发生在LFU ghost 链表中，它会减少LRU链表的长度（减一），以此在LFU 链表中加一个可用空间。

利用这种行为，ARC使它自己自适应于工作负载。如果工作负载趋向于访问最近访问过的文件，将会有更多的命中发生在LRU Ghost链表中，也就是说这样会增加LRU的缓存空间。反过来一样，如果工作负载趋向于访问最近频繁访问的文件，更多的命中将会发生在LFU Ghost链表中，这样LFU的缓存空间将会增大。

进一步，这种行为开启了一个灵活的特性：假设你为处理log文件而读取了大量的文件。你只需要每个文件一次。一个LRU 缓存将会把所有的数据缓存住，这样也就把经常访问的数据也淘汰出去了。但是由于你仅仅访问这些文件一次，它们不会为你带来任何价值一旦它们填满了缓存。

一个ARC缓存的行为是不同的。显然这样的工作负载仅仅会很快填满LRU链表空间，而这些页面很快就会被淘汰出去。但是由于每个这样的页面仅仅被访问一次，它们基本不太可能在为最近访问的文件而设计的ghost链表中命中。这样，LRU的缓存空间不会因为这些仅读一次的页面而增加。

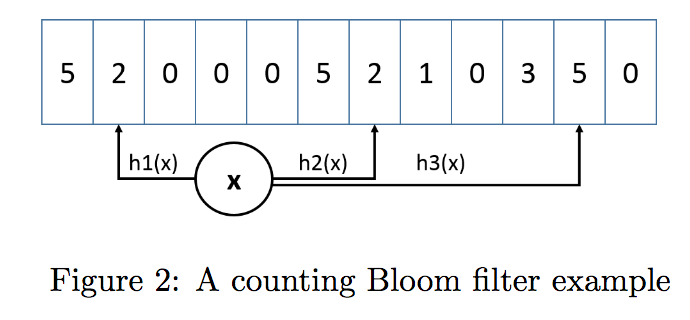
假如你把这些log文件与一个大的数据块联系在一起（为了简单起见，我们假设这个数据块没有自己的缓存机制）。数据文件中的数据页应该会被频繁的访问。被LFU ghost链表引用的正在被访问的页面就很有可能大大的高于LRU ghost链表。这样，经常被访问的数据库页面的缓存空间就会增加。最终，我们的缓存机制就会向缓存数据块页面优化，而不是用log文件来污染我们的缓存空间。

1. **总结**

无论是对于各种LRU的优化，如innodb分代缓存，linux的LRU-2，以及通过ghost list来动态调整2-queue的ARC，都是在LRU的基础上进行优化调整，但因为LFU的三大缺点没有真正实现frequency-biased缓存。

1. **基于LRU和LFU相结合的W-TinyLFU**
2. **解决LFU占用内存：BloomFilter**

我们不希望记录LFU中每个key对应的count，那么可以参考布隆过滤器，利用k次hash，将对应的hash槽加1，那么对于key的多个hash槽的count，取最小值作为lfu近似count。这样可以大大减少空间复杂度。这种算法称为CountMin Sketch。



1. **解决LFU数据过期：reset**

TinyLFU使用reset机制来保证sketch中的数据尽可能最新。每增加一个新的元素到approximation sketches，会增加一个计数值size，一旦计数值size达到了一个预设的采样尺寸（W）：

* 就会将频率采样（CBF）维护的所有计数值除以2（可以使用高效的寄存器位移来实现）
* 同时，size=size/2

论文也花了较大的篇幅通过数学归纳法来证明了这种Reset机制的正确性，且评估了其存在的截断错误（3会被reset为1，而非1.5），并且得出了以下结论：

* reset在constant distribution频率分布下完全正确，且可以应对流量频率的变化（数学归纳法证明，感兴趣的可以参考原文3.3.1）。
* 采样数W越大，截断错误的带来的影响越小。

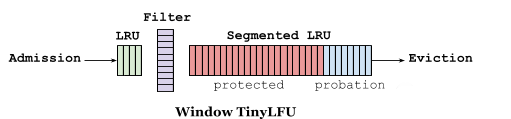
1. **解决稀疏流量：window cache**

有了tinyLFU，我们就可以使用其作为Lru过期淘汰策略。同时为了保证不会有突发稀疏流量，这部分流量可能无法累计足够的count而进入tinyLFU缓存，从而导致缓存失败。所以在tinyLFU之前我们加入了一个window cache（lru，占用整个w-tinyLFU内存1%）用于处理LRU适合的burst场景

1. **Cache分代**

window cache（lru，占用整个w-tinyLFU内存1%）用于处理LRU适合的burst场景，tinyLFU作为filter来执行淘汰策略，主缓存则是一个分段缓存：segmented cache（lru，占用整个w-tinyLFU内存99%）。这个分段缓存分为两个区域，和linux中的active和inactive链表类似。

* probation A1，占用整个分段缓存20%的存储空间，主要用于**执行淘汰**。
  + 如果这里的元素被访问，则将其升级至protected段，如果protected满了则而将protected段最老元素会降级到probation段。
  + 如果从window cache有淘汰元素进入，则需要将probation中的最老元素与其进行count比较来判断到底淘汰谁。
* protected A2，占用整个分段缓存80%的存储空间，主要存储有价值的缓存数据。



1. **tinyLFU的内存优化**
2. **counter**

对tinyLFU，其reset采样大小为w，那么显然counter就不会无限扩大，其大小最大为w，对应数据大小为log(w) bit。设缓存大小为c，对于新的需要进入cache的元素，其计数器大小本身不需要达到log(w)，因为我们只需要比较在缓存中需要丢弃元素的计数（频率大于等于1/c才能保持在缓存里，而reset又会降低一半的频率），与新元素计数即可，而新元素的计数器只需要保证最大为w/c即可。例如w=100, c= 10，那么新元素的计数器只需要限制在10以内即可。

1. **doorKeeper**

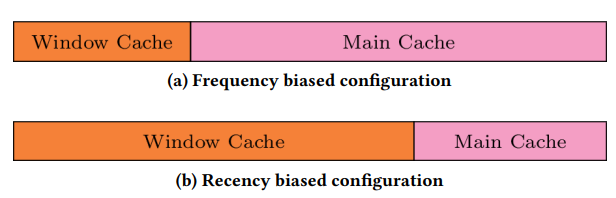
通常会有一些大量只出现一次的对象进入tinyLFU计数，显然这些对象利用count min sketch计算也会增加counter的长度（counter的数据类型可以动态增加）。那么我们可以在进入tinyLFU计算前，加入一个名为doorKeeper的标准布隆过滤器。只有在doorKeeper的元素，才会真正计算器tinyLFU，否则只停留在doorKeeper中，即count为1。

reset会清空doorKeeper。

1. **Hill Climber**

W-TinyLFU并不是真正的银弹，Ben提到，希望caffeine能够在大部分场景下提升20%~30%的命中率，同时在最差场景下损失5%的命中率也是可以接受的。如果使用固定的W-TinyLFU，我们就无法根据workload动态调整我们的缓存。

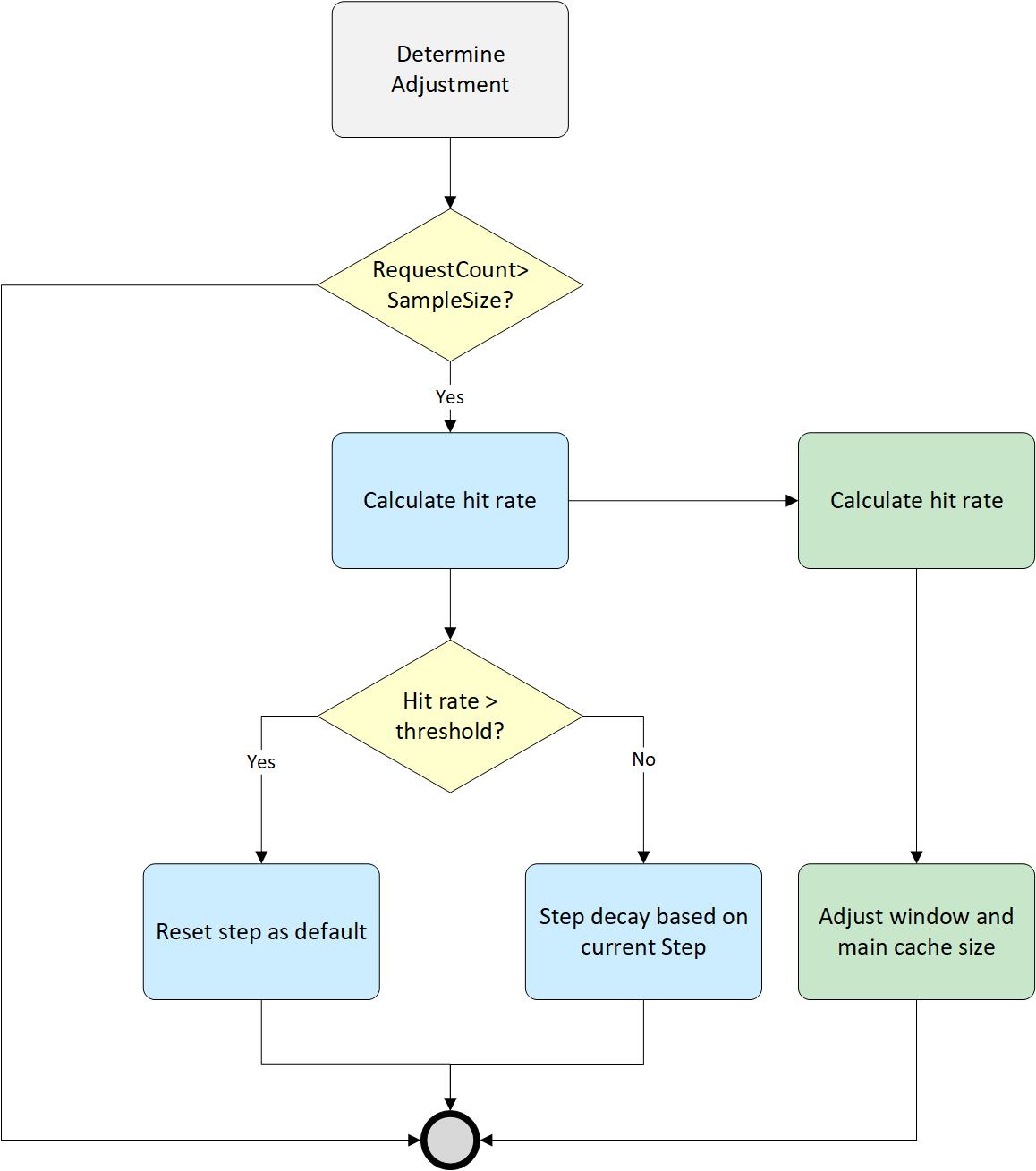
自适应缓存未来发展的方向。在caffeine的[issue](https://github.com/ben-manes/caffeine/issues/106) *https://github.com/ben-manes/caffeine/issues/106*中有人提到caffeine的命中率不如LRU。Ben回答到，我们需要搞清楚数据源(workload)究竟是recency-biased，还是frequency-biased，如果是更接近recency-biased，则需要进行recency补偿，让我们的缓存模型更接近LRU，反之亦然。



caffeine补偿的方法是利用Hill Climber[5]调整window cache大小来实现的。具体的实现非常简单：

* 根据统计的hit rate动态调整window cache(LRU属性)大小，从而完成对workload的recency-biased或frequency-biased的补偿
* 这种补偿并不是频繁的，而是在访问次数达到frequencySketch定义的sampleSize才执行一次。
* 对缓存的补偿主要作用于window大小，同时想要教快的完成对缓存的调整，所以设置刚开始的移动step为缓存总大小的6.25%。
* 如果hit rate变化且没有超过HILL\_CLIMBER\_RESTART\_THRESHOLD门限5%，可以认定我们调整的方向是对的，那么我们会降低step的绝对值，即将其变成原来的98%，并根据hit rate的正负号来设定新step的符号，从而来调整window cache。
* 如果hit rate变化超过门限5%，说明调整带来了非常大的变化，此时需要将Step重置为较大间隔，以快速完成对hit rate大变化的项目，那么则重新将Step变成缓存总大小的6.25%。

具体流程如下：



1. **Adaptive 优化的选择**

[5][6]中提到，我们对w-tinyLFU的调整可以包括两方面：

1. Window cache
2. countMinSketch
   * sample size： 调小则说明recency-biased，调大说明frequency-biased
   * counter自增大小：默认是1，如果我们设定为2或者更大，则说明recency-biased

caffeine在实践中发现，**window cache的调整要优于countMinSketch**。

此外，除了hill Climber这种动态调整外，Prof. Dr. Roy Friedman还提出了indicator的方法，即通过sketch中的counter数量来直接计算recency-biased的数值，从而一步到位直接修改window cache大小。实践结论如下：

* indicator要比hill climber更快，且命中率有轻微提升;
* indicator要比hill climber更占用空间，实现更复杂，所以轻微的命中提升不足以弥补

综上，caffeine使用了**hill climber**来调整**window cache 来逐步达到对workload的匹配。**

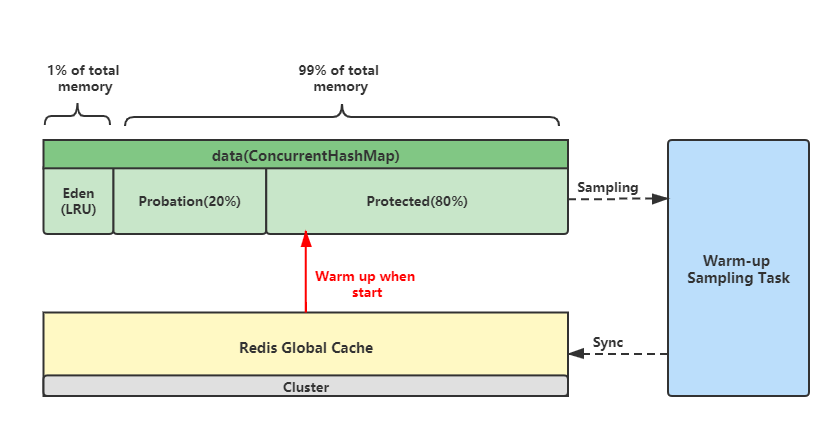
1. **Caffeine缺点**

* Caffeine实现的W-tinyLFU属于frequency缓存，在处理LRU时需要一定的hill climber，显然没有直接使用LRU来的快。
* 在不知道workload究竟是recency-biased还是frequency-biased情况下，我们通过hill climber来调整显然属于slow warm up。

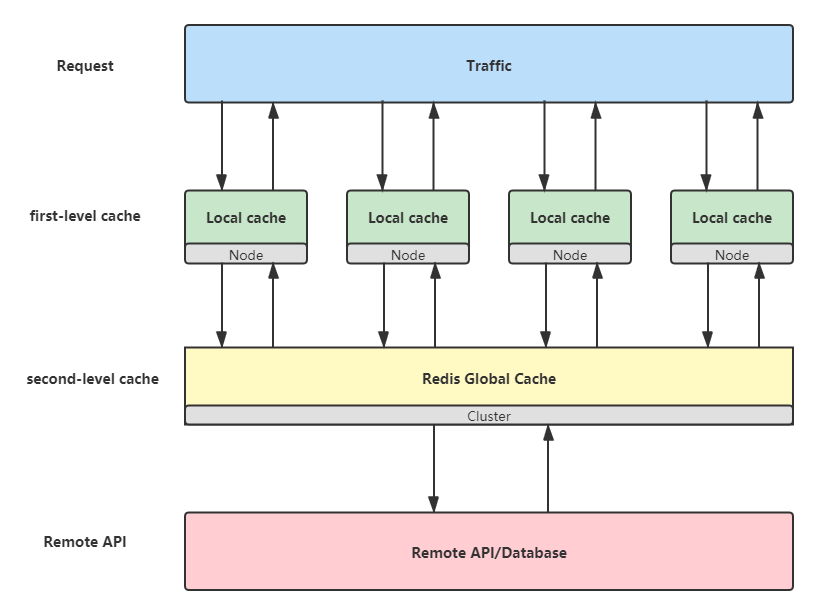
1. **Caffeine的实现**

参见《[cache-2-caffeine](cache-2-caffeine.docx)》

1. **进一步思考**
2. **本地缓存热启动**



1. **全局二级缓存**



**参考**

1. [cache-1-guava](cache-1-guava.docx)
2. [mysql之innodb-9-lru优化](mysql之innodb-9-lru优化.docx)
3. [cache-2-caffeine](cache-2-caffeine.docx)
4. LRU-k和2-queue: <https://www.jianshu.com/p/c4e4d55706ff?ivk_sa=1024320u>
5. ARC <http://blog.chinaunix.net/uid-28466562-id-3837685.html>
6. caffeine-adaptive software cache management
7. w-tinyLFU study video: <https://www.tele-task.de/lecture/video/7396/>
8. LIRS <https://www.cnblogs.com/dyllove98/archive/2013/06/08/3127492.html>
9. LIRS <https://www.jianshu.com/p/036f82975f81>