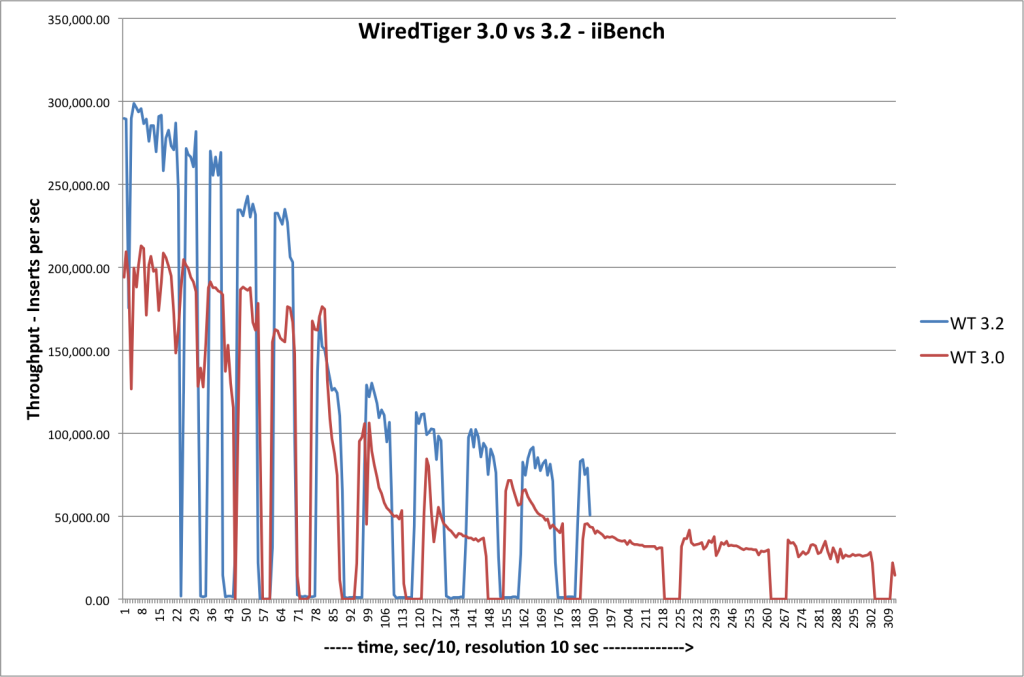
WiredTiger实现：一个LRU cache深坑引发的分析

袁荣喜

从mongoDB 3.0版本引入WiredTiger存储引擎(以下称为WT)以来，一直有同学反应在高速写入数据时WT引擎会间歇性写挂起，有时候写延迟达到了几十秒，这确实是个严重的问题。引起这类问题的关键在于WT的LRU cache的设计模型，WT在设计LRU cache时采用**分段扫描标记**和**hazard pointer**的淘汰机制，在WT内部称这种机制叫eviction cache，其设计目标是充分利用现代计算机超大内存容量来提高事务读写并发。在高速不间断写时内存操作是非常快的，但是内存中的数据最终必须写入到磁盘上，当将页数据（page）由内存中写入磁盘是需要写入时间，必定会和应用程序的高速不间断写产生竞争，这在任何数据库存储引擎都是无法避免的，只是由于WT利用大内存和写无锁的特性，让这种不平衡现象更加显著。下图是一位网名叫**chszs**同学对mongoDB 3.0和3.2版本测试高速写遇到的hung现象.



**图1**

从上图可以看出，数据周期性出现了hung现象，笔者在单独对WT进行高并发顺序写时遇到的情况和上图基本一致，有时候挂起长达20秒。针对此问题我结合WT源码和调试测试进行了分析，基本得出的结论如下：

1. WT引擎的eviction cache实现时没有考虑lru cache的分级淘汰，只是通过扫描btree来标记，这使得它和一些独占式btree操作(例如：checkpoint)容易发生竞争。
2. WT btree的checkpoint机制设计存在bug，在大量并发写事务发生时，checkpoint需要很长时间才能完成，造成刷入磁盘的数据很大，写盘时间很长。容易引起cache 满而挂起所有的读写操作。
3. WT引擎的redo log文件超过1GB大小后就会另外新建一个新的redo log文件来继续存储新的日志，在操作系统层面上新建一个文件的是需要多次I/O操作，一旦和checkpoint数据刷盘操作同时发生，所有的写也就挂起了。

要彻底弄清楚这几个问题，就需要对从WT引擎的eviction cache原理来剖析，通过分析原理找到解决此类问题的办法。先来看eviction cache是怎么实现的，为什么要这么实现。

# eviction cahce原理

eviction cache是一个LRU cache，即页面置换算法缓冲区，LRU cache最早出现的地方是操作系统中关于虚拟内存和物理内存数据页的置换实现，后被数据库存储引擎引入解决内存和磁盘不对等的问题。所以LRU cache主要是解决内存与数据大小不对称的问题，让最近将要使用的数据缓存在cache中，把最迟使用的数据淘汰出内存，这是LRU置换算法的基本原则。但程序代码是无法预测未来的行为，只能根据过去数据页的情况来确定，一般我们认为过去经常使用的数据比不常用的数据未来被访问的概率更高，很多LRU cache大部分是基于这个规则来设计。

WT的eviction cache也不例外的遵循了这个LRU原理，不过WT的eviction cache对数据页采用的是分段局部扫描和淘汰，而不是对内存中所有的数据页做全局管理。基本思路是一个线程阶段性的去扫描各个btree，并把btree可以进行淘汰的数据页添加到一个lru queue中，当queue填满了后记录下这个过程当前的btree对象和btree的位置（这个位置是为了作为下次阶段性扫描位置）,然后对queue中的数据页按照访问热度排序，最后各个淘汰线程按照淘汰优先级淘汰queue中的数据页，整个过程是周期性重复。WT的这个evict过程涉及到多个eviction thread和hazard pointer技术。

**WT的evict过程都是以page为单位做淘汰，而不是以K/V。这一点和memcache、redis等常用的缓存LRU不太一样，因为在磁盘上数据的最小描述单位是page block，而不是记录。**

## eviction线程模型

从上面的介绍可以知道WT引擎的对page的evict过程是个多线程协同操作过程，WT在设计的时候采用一种叫做leader-follower的线程模型，模型示意图如下:

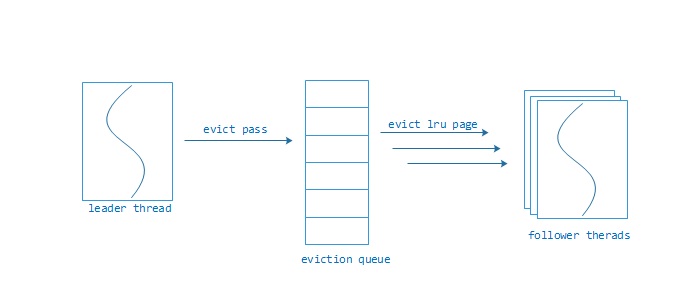


图2

Leader thread负责周期性扫描所有内存中的btree索引树，将符合evict条件的page索引信息填充到eviction queue，当填充queue满时，暂停扫描并记录下最后扫描的btree对象和btree上的位置，然后对queue中的page按照事务的操作次数和访问次做一次淘汰评分，再按照评分从大小做排序。也就是说最评分越小的page越容易淘汰。下个扫描阶段的起始位置就是上个扫描阶段的结束位置，这样能保证在若干个阶段后所有内存中的page都被扫描过一次，这是为了公平性。这里必须要说明的是一次扫描很可能只是扫描内存一部分btree对象，而不是全部，所以我对这个过程称为阶段性扫描（evict pass），它不是对整个内存中的page做评分排序。**这个阶段性扫描的间隔时间是100毫秒**，而触发这个evict pass的条件就是WT cache管理的内存超出了设置的阈值，这个在后面的**eviction cache管理的内存**小节中详细介绍。

在evict pass后，如果evction queue中有待淘汰的page存在就会触发一个操作系统信号来激活follower thread来进行evict page工作。虽然evict pass的间隔时间通常是100毫秒，这里有个问题就是当WT cache的内存触及上限并且有大量写事务发生时，读写事务线程在事务开始时会唤醒leader thread和follower thread,这就会产生大量的操作系统上下文切换，系统性能急剧下降。好在WT-2.8版本修复了这个问题，leader follower通过抢锁来成为leader,通过多线程信号合并和周期性唤醒来follower,而且leader thread也承担evict page的工作，可以避免大部分的线程唤醒和上下文切换。**是不是有点像Nginx的网络模型？**

## hazard pointer

hazard pointer是一个无锁并发技术，其应用场景是单个线程写和多个线程读的场景，大致的原理是这样的，每读的线程设计一个与之对应的无锁数组用于标记这个线程引用的hazard pointer对象。读线程的步骤如下：

1. 读线程在访问某个hazard pointer对象时，先将在自己的标记数组中标记访问的对象。
2. 读线程在访问完毕某个hazard pointer对象时，将其对应的标记从标记数组中删除。

写线程的步骤大致是这样的，写线程如果需要对某个hazard pointer对象写时，先判断所有读线程是否标记了这个对象，如果标记了，放弃写。如果未标记，进行写。

关于hazard pointer理论可以访问https://www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf

Hazard pointer是这样应用在WT中呢？我们这样来看待这个事情，把内存page的读写看做hazard pointer的读操作，把page从内存淘汰到磁盘上的过程看做hazard pointer的写操作，这样瞬间就能明白为什么WT在页的操作上可以不The FIX Rules规则，而是采用无锁并发的页操作。**要达到这种访问方式有个条件就是内存中page本身的结构要支持lock free访问**，这个在《剖析WiredTiger数据页无锁及压缩》一文中介绍过了。从上面的描述可以看出**evict page的过程中首先要做一次hazard pointer写操作检查**，**而后才能进行page的reconcile和数据落盘。**

hazard pointer并发技术的应用是整个WT存储引擎的关键，它关系到btree结构、internal page的构造、事务线程模型、事务并发等实现。Hazard pointer使得WT不依赖The Fix Rules规则，也让WT的btree结构更加灵活多变。

Hazard pointer是比较新的无锁编程模式，可以应用在很多地方，笔者我曾在一个高并发媒体服务器上用到这个技术，以后有机会把里面的技术细节分享出来。

## eviction cache管理的内存

eviction cache其实就是内存管理和page淘汰系统，目标就是为了使得管辖的内存不超过物理内存的上限，而触发淘汰evict page动作的基础依据就是内存上限。eviction cache管理的内存就是内存中page的内存空间，page的内存分为几部分：

1. 从磁盘上读取到已经刷盘的数据，在page中称作disk buffer。如果WT没有开启压缩且使用的MMAP方式读写磁盘，这个disk buffer的数据大小是不计入在WT eviction cache管理范围之内的。如果是开启压缩，会将从MMAP读取到的page数据解压到一个WT 分配的内存中，这个新分配的内存是计在WT eviction cache中的。
2. Page在内存中新增的修改事务数据内存空间，计入在eviction cache中。
3. Page基本的数据结构所有的内存空间，计入在eviction cache中。

*PS:关于page结构和内存相关的细节请查看《剖析WiredTiger数据页无锁及压缩》。*

WT在统计page的内存总量是通过一个footprint机制来统计两项数据，一项是总的内存使用量mem\_size，一项是增删改造成的脏页数据总量dirty\_mem\_size。同时方式很简单，就是每次对页进行载入、增删改、分裂和销毁时对上面两项数据做原子增加或者减少计数，这样可以精确计算到当前系统中WT引擎内存占用量。假设引擎外部配置最大内存空间为cache\_size,内存上限触发evict的比例为80%，内存脏页上限触发evict的比例为75%.那么系统触发evict pass操作的条件为：

mem\_size > cache\_size \* 80%

或者

dirty\_mem\_size > cache\_size \* 75%

满足这个条件leader线程就会进行evict pass阶段性扫描并填充eivction queue,最后驱使follower线程进行evict page操作。

## evict pass策略

前面介绍过evict pass是一个阶段性扫描的过程，整个过程分为扫描阶段、评分排序阶段和evict调度阶段。扫描阶段是通过扫描内存中btree，检查btree在内存中的page对象是否可以进行淘汰。扫描步骤如下：

1. 根据上次evict pass最后扫描的btree和它对应扫描的位置最为本次evict pass的起始位置,**如果当前扫描的btree被其他事务线程设成独占访问方式，跳过当前btree扫描下个btree对象**。
2. 进行btree遍历扫描，如果page满足淘汰条件，将page的索引添加到evict queue中，淘汰条件为：

* 如果page是数据页，必须page当前最新的修改事务必须早以evict pass事务。
* 如果page是btree内部索引页，必须page当前最新的修改事务必须早以evict pass事务且当前处于evict queue中的索引页对象不多于10个。
* 当前btree不处于正建立checkpoint状态

1. 如果本次evict pass当前的btree有超过100个page在evict queue中或者**btree处于正在建立checkpoint时**，结束这个btree的扫描，切换到下一个btree继续扫描。
2. 如果evict queue填充满时或者本次扫描遍历了所有btree，结束本次evict pass。

PS:在开始evict pass时，evict queue可能存在有上次扫描且未淘汰出内存的page,那么这次evict pass一定会让queue填满（大概400个page）。

评分排序阶段是在evict pass后进行的，当queue中有page时，会根据每个page当前的访问次数、page类型和淘汰失败次数等计算一个evict scope，然后按照评分从小打到进行快排，排序完成后，会根据queue中最大分数和最小分数计算一个淘汰边界evict\_throld，queue中所有大于evict\_throld的page不列为淘汰对象。

**WT为了让btree索引页尽量保存在内存中，在评分的时候索引页的分值会加上1000000分，让btree索引免受淘汰。**

evict pass最后会做个判断，如果有follower线程存在，用系统信号唤醒follower进行evict page。如果系统中没有follower，leader线程进行eivct page操作。这个模型在WT-2.81版本已经修改成抢占模式。

## evict page过程

evict page其实就是将evict queue中的page数据先写入到磁盘文件中，然后将内存中的page对象销毁回收。整个evict page也分为三个阶段：从evict queue中获取page对象、hazard pointer判断和page的reconcile过程，整个过程的步骤如下：

1. 从evict queue头开始获取page，如果发现page的索引对象不为空，对page进行LOCKED原子性标记防止其他读事务线程引用并将page的索引从queue中删除。
2. 对淘汰的page进行hazard pointer，如果有其他线程对page标记hazard pointer, page不能被evict，将page的评分加100.
3. 如果没有其他线程对page标记hazard pointer,对page进行reconcile并销毁page内存中的对象。

evict page的过程大部分是由follower thread来执行，这个在上面的线程模型一节中已经描述过。**但在一个读写事务开始之前，会先检查WT cache是否有足够的内存空间进行事务执行，如果WT cache的内存容量触及上限阈值，事务执行线程会尝试去执行evict page工作，如果evict page失败，会进行线程堵塞等待，等待WT cache有执行读写事务的内存空间（是不是读写挂起了？）。**这种状况一般出现在正在建立checkpoint的时候，那么checkpoint是怎么引起这个现象的呢？下面来分析缘由。

# eviction cache与checkpoint之间的事

众所周知，建立checkpoint的过程是将内存中所有的脏页（dirty page）同步刷入磁盘上并将redo log的重演位置设置到最后修改提交事务的redo log位置，相对于WT引擎来说，就是将eviction cache中的所有脏页数据刷入磁盘但并不将内存中的page淘汰出内存。这个过程其实和正常的evict过程是冲突的，而且checkpoint过程中需要更多的内存完成这项工作，这使得在一个高并发写的数据库中有可能出现挂起的状况发生。为了更好的理解整个问题的细节，我们先来看看WT checkpoint的原理和过程。

## btree的checkpoint

WT引擎中的btree建立checkpoint过程还是比较复杂的，过程步骤比较多，而且每个步骤会涉及到索引、日志、事务和磁盘文件等。我以WT-2.7(mongoDB 3.2)版本为例子，checkpoint大致的步骤如下图：



图3

在上图中，其中绿色的部分是在开始checkpoint事务之前，将所有的btree的脏页写入文件OS cache，如果在高速写的情况下，写的速度接近也reconcile的速度，那么这个过程将会持续很长时间，也就是说OS cache中会存在大量未落盘的数据。而且在WT中btree采用的copy on write(写时复制)和extent技术，这意味OS cache中的文件数据大部分是在磁盘上是连续存储的，那么在绿色框最后一个步骤会进行同步刷盘，**这个时候如果OS cache的数据量很大就会造成这个操作长时间占用磁盘I/O**。这个过程是会把所有提交的事务修改都进行reconcile落盘操作，

在上图的紫色是真正开始checkpoint事务的步骤，这里需要解释的是**由于前面绿色步骤刷盘时间会比较长，在这个时间范围里会有新的写事务发生，也就意味着会新的脏页，checkpint必须把最新提交的事务修改落盘而且还要防止btree的分裂，这个时候就会获得btree的独占排他式访问，这时 eviction cache不能对这个btree上的页进行evict操作**（在这种情况下是不是容易造成WT cache满而挂起读写事务？）。

PS:WT-2.8版本之后对checkpoint改动非常大，主要是针对上面两点做了拆分，防止读写事务挂起发生，但大体过程是差不多的。

## 写挂起

通过前面的分析大概知道写挂起的原因了，主要引起挂起的现象主要是因为写内存的速度远远高于写磁盘的速度。先来看一份内存和磁盘读写的速度的数据吧。顺序读写的对比：



图4

从上图可以看出，SATA盘的顺序读写1MB数据大概需要8ms, SSD相对快一点，大概只需2ms.但内存的读写远远大于磁盘的速度。SATA的随机读取算一次I/O时间，大概在8ms 到10ms，SSD的随机读写时间比较快，大概0.1ms。

我们来分析checkpoint时挂起读写事务的几种情况，假设系统在高速写某一张表（每秒以100MB/S的速度写入），每1分钟做一次checkpoint。那么1分钟后开始进行图3中绿色的步骤，这个时候可能会这一分钟写入的脏数据压缩先写入到OS cache中，这个时候可能OS Cache可能存有近2GB的数据。这2GB的sync刷到磁盘上的时间至少需要10 ~ 20秒，而且磁盘I/O是被这个同步刷盘的任务占用了。这个时候有可能发生几件事情：

1. 外部的写事务还在继续，事务提交时需要写redo log文件，这个时候磁盘I/O被占用了，写事务挂起等待。
2. 外部的读写事务还在继续，redo log文件满了，需要新建一个新的redo log文件，但是新建文件需要多次随机I/O操作，磁盘I/O暂时无法调度来创建文件，所有写事务挂起。
3. 外部读写事务线程还在继续，因为WT cache触发上限阈值需要evict page。Evict page时也会调用reconcile将page写入OS cache,但这个文件的OS cache正在进行sync，evict page只能等sync完成才能写入OS cache，evict page线程挂起，其他读写事务在开始时会判断是否有足够的内存进行事务执行，如果没有足够内存，所有读写事务挂起。

这三种情况是因为阶段性I/O被耗光而造成读写事务挂起的。

在图3紫色步骤中，checkpoint事务开始后会先获得btree的独占排他访问方式，这意味这个btree对象上的page不能进行evict,如果这个btree索引正在进行高速写入，有可能让checkpoint过程中数据页的reconcile时间很长，**从而耗光WT cache造成读写事务挂起现象**，这个现象极为在测试中极为少见（碰见过两次）。 要解决这几个问题只要解决内存和磁盘I/O不对等的问题就可以。

## 内存和磁盘I/O的权衡

引起写挂起问题的原因多种多样，但归根结底是因为内存和磁盘速度不对称的问题。因为WT的设计原则就是让数据尽量利用现代计算机的超大内存，可是内存中的脏数据在checkpoint时需要同步写入磁盘造成瞬间I/O很高，这个是矛盾。要解决这些问题个人认为有以下几个途径：

1. 将MongoDB的WT版本升级到2.8，2.8版本对evict queue模型做了分级，尽量避免evict page过程中堵塞问题,2.8的checkpoint机制不在是分为预前刷盘和checkpoint刷盘，而是采用逐个对btree直接做checkpoint刷盘，缓解了OS cache缓冲太多的文件脏数据问题。
2. 试试direct I/O或许会有不同的效果，WT是支持direct I/O模式。笔者试过direct I/O模式，让WT cache彻底接管所有的内存管理，写事务的并发会比MMAP模式少10%，单没有出现过超过1秒的写延迟问题。
3. 尝试将WT cache设小点，大概设置成整个内存的1/4左右。这种做法是可以缓解OS cache中瞬间缓存太多文件脏数据的问题，但会引起WT cache频繁evict page和频繁leader-follower线程线程上下文切换。而且这种机制也依赖于OS page cache的刷盘周期，周期太长效果不明显。
4. 用多个磁盘来存储，redo log文件放在一个单独的机械磁盘上，数据放在单独一个磁盘上，避免redo log与checkpoint刷盘发生竞争。
5. 有条件的话，换成将磁盘换成SSD吧。这一点比较难，mongoDB现在也大量使用在OLAP和大数据存储，而高速写的场景都发生这些场景，成本是个问题。如果是OLTP建议用SSD。

这些方法只能缓解读写事务挂起的问题，不能说彻底解决这个问题，WT引擎发展很快，开发团队正对WT eviction cache和checkpoint正在做优化，这个问题慢慢变得不再是问题，尤其是WT-2.8版本，大量的模型和代码优化都是集中在这个问题上。

# 后记

WT的eviction cache可能有很多不完善的地方，也确实给我们在使用的过程造成了一些困挠，应该用中立的角度去看待它。可以说它的读写并发速度是其他数据库引擎不能比的，正是由于它很快，才会有写挂起的问题，因为磁盘的速度就那么快。以上的分析和建议性的解决方案或许对碰到类似问题的同学有用。

WT团队的开发速度也很快，对BUG的响应速度也很快，这类问题是他们正在重点解决的问题。在国内也有很多mongoDB这方面相关的专家，他们在解决此类问题有非常丰富的经验，也可以请求他们来帮忙解决这类问题。

在本文涉及到的问题分析过程中得到了**阿里云张友东的帮助，在此表示感谢**。