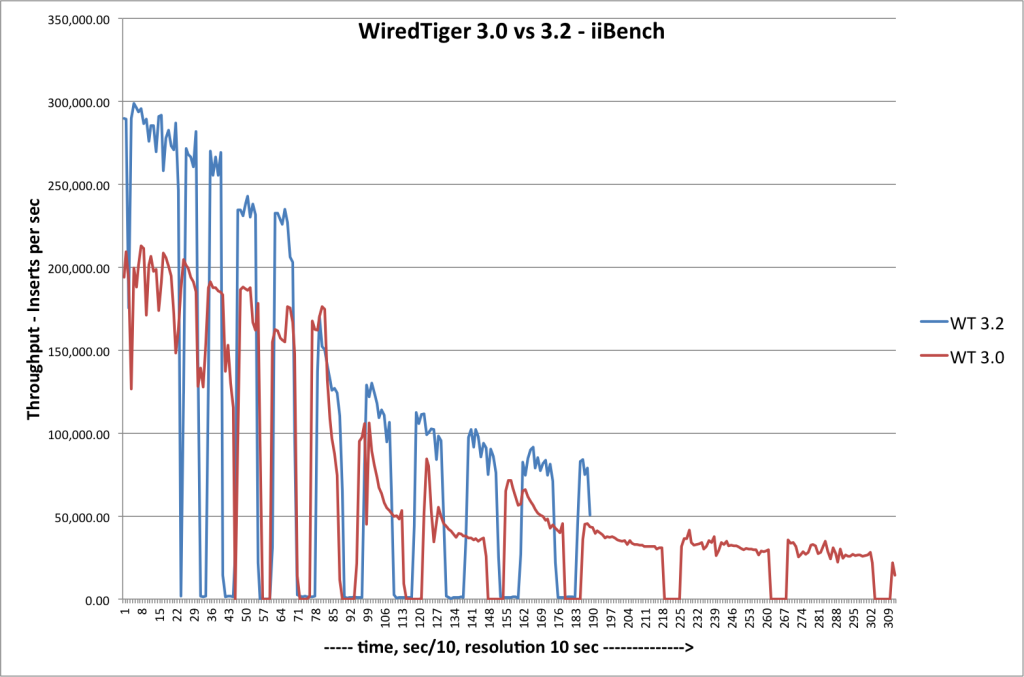
WiredTiger实现：一个LRU cache深坑引发的分析

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10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:

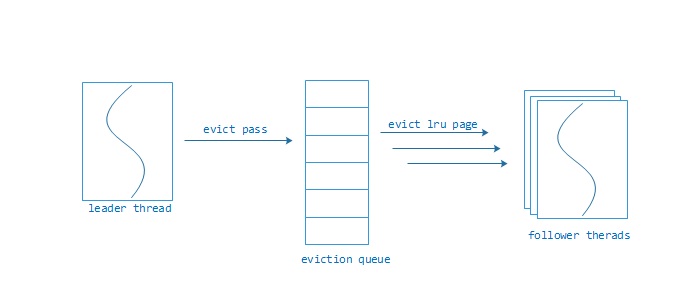


图2

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机制来统计的，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。

## LRU eviction策略

## evict page过程

# evict cache与checkpoint之间的事

## btree的checkpoint过程

## write hung

## 内存和磁盘I/O的权衡

# 后记