解析WiredTiger实现：数据组织

袁荣喜

2016-06-12

在《WiredTiger实现：事务》一文中提到了WiredTiger（简称为WT）是为了适应现代CPU、内存和磁盘的特性而设计的存储引擎，它的特点就是充分利用CPU的速度和内存的容量来弥补磁盘访问速度不足。在介绍WT的组织方式前，先来看看传统数据库引擎的数据组织方式，传统数据库引擎大都是设计一个磁盘和内存完全一样的数据组织方式—page,这个结构是固定的空间大小（innodb的page是16KB）,访问它必须遵守严格的The FIX Rules规则：

 修改一个page需要获得该页的x-latch lock

访问一个page是需要获得该页的s-latch lock或者x-latch lock

持有该page的latch直到修改或者访问该页的操作完成,latch unlock

WT没有像传统的数据库引擎那样设计一套内存和磁盘page完全一致的数据组织方式，而是针对磁盘和CPU、内存三者之间特点设计了一套独特的数据组织方式，这种数据组织结构分为两部分：

In-memory page: 内存中的数据页(page)

Block extent: 基于磁盘文件的偏移量的范围存储

WT内存中的page是一个松散自由的数据结构，而磁盘上的extent只是一个变长的序列化后的数据块，这样做的目的有以下几点：

1.内存中的page松散结构可以不受磁盘存储方式的限制和The FIX Rules规则的影响，可以自由的构建page的无锁多核并发结构，充分发挥CPU多核的能力。

2.可以自由的在内存page和磁盘extent之间实现数据的压缩，提高磁盘的存储效率和减少I/O访问时间。

3.extent的数据进行压缩后比原始数据小很多，在写入文件时，可以充分利用操作系统文件的高速缓冲区(page cache)来提高读写速度。

关于WT的压缩效率见下图

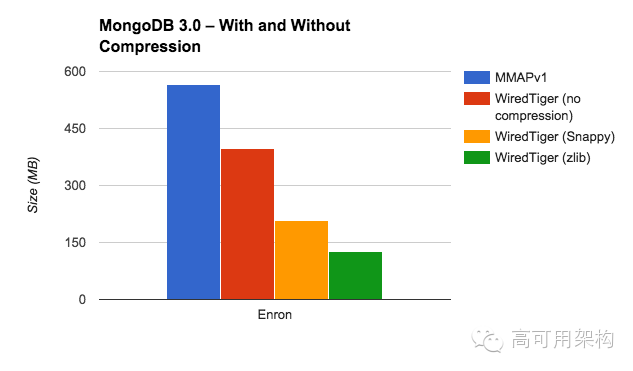


图1

如果需要了解整个WT引擎的原理，首先应该分析和理解它的数据组织方式。在本文中我们将会重点围绕WT的设计目标来分析和梳理WT的数据组织方式和实现。

# WT测试样例

在分析WT数据组织之前，先来看看WT这种数据组织方式产生的效果，我们在一个普通的开发机器上对WT的原生API接口进行测试。测试环境如下：

CPU: i7-4710MQ CPU @ 2.50GHz，8核

内存：4G

硬盘：1TB SATA,5400转

WT数据库配置：

Cache size: 1GB, page max size:64KB, OS page cache:1GB

Key:一个从零开始自增长的整数

Value:一个长度介于100 ~ 200的随机字符串。

测试流程：

测试程序先新建一张表，用8条线程并发向表中插入指定数量（以百万为单位）的K/V对，在插入完成后做一次checkpoint让插入的数据写入磁盘，统计这个表在磁盘上的大小和这个过程的耗时。再用单线程在表中用随机查询1000个不同K/V的查询并统计查询的耗时,通过耗时可以计算出insert和query的TPS。统计的这三个参数分别就是磁盘空间占用、写性能和读性能。我们分别进行不压缩extent的测试和进行ZIP压缩extent的测试。

## 磁盘空间统计对比

图2

从上图可以看出，2亿条记录，没有压缩的磁盘空间30GB，而压缩后的磁盘空间2GB左右。

## 写性能对比

图3

从上图可以看出，写入的数据在1亿条（14GB）压缩(518K TPS)和没压缩(631k TPS)的写性能差距不大，但随着数据量增大，大量的数据在内存和磁盘间swap，选择extent压缩的写性能(790K TPS)要好于没压缩的写性能（351K TPS）。

## 读性能对比

图4

从上图可以得出在表空间文件上的数据大小没有超过WT的OS\_page\_cache限制时，**无压缩的读性能**要好于**有压缩的读性能**，这是因为extent缓冲在操作系统高速缓冲区中，当发生访问时从高速缓冲区直接获取无压缩extent转换成内存中的page，只要一次内存中的结构重建即可，而压缩的extent需要进行解压缩后再重建，所以没有压缩的读性能更好。 如果文件数据大小超出OS\_page\_cache限制后，extent数据要从磁盘上读取，而有压缩的extent占用的空间比较小，从磁盘上读取的I/O访问时间是小于无压缩读取的时间的，这时有压缩的读性能要好于无压缩的读性能。（后面单独用一个篇幅来分析WT的磁盘I/O访问及调度管理）

除了压缩优化了数据的读写，WT内存中的无锁page结构也使得读写操作具有更好的并发性，才使得WT在非常普通机器上有如此好的表现。

# 内存中的数据页

通过 上面的介绍已经对WiredTiger的数据组织方式有了初步的了解，WT数据组织方式就是in-memory page加block-extent。先来对它内存部分的in-memory page（内存数据页，简称为page）来做分析。WT引擎中的page分以下几类：

row internal page: 行存储b-tree的索引页

row leaf page: 行存储b-tree的数据页

column internal page: 列存储b-tree的索引页

column fix leaf page: 列存储b-tree的定长数据页

column var leaf page: 列存储b-tree的变长数据页

因为mongoDB主要是使用行存储，所以在这里主要分析row leaf page这个结构和原理，结构图如下:



图5

上图中主要有以下几个单元：

wt\_page: 内存中的page结构对象，page的访问入口。

wt\_modify: page的修改状态信息，主要包含脏页标示、当前更新事务ID和page的insert lock等。

row\_array: 磁盘上原有row的位置索引数组，主要用于页内检索。

row\_insert\_array: 一个增加kv(insert kv)的跳表对象数组

row\_update\_array:一个在row基础上做更新(update kv)的value mvcc list对象素组。

page\_disk: 从磁盘上读取到的extent数据缓冲区，包含一个page\_header和一个数据存储缓冲区(disk data)

page disk data：存储在磁盘上的page kv行集合数据

罗列了这么多结构，他们之间是通过什么方式来关联的呢？我们通过一个实例来说明，假如一个page存储了一个[0, 100]的key范围，从磁盘上原来存储的行key=2, 10 ,20, 30 , 50, 80, 90,他们的值分别是value = 102, 110, 120, 130, 150, 180, 190。在page读到内存后，分别对key=20的value进行了两次修改，两次修改的值是分别402,502。对key = 20 ,50的value做了一次修改，修改后的value = 122, 155，后有分配insert了新的key = 3,5, 41, 99，value = 203,205,241,299。那么在内存中的page就是如下图组织数据的。



图6

row\_array的长度是根据page从磁盘中读取出来的行数确定的，每个数组单元(wt\_row)存储的是这个kv row在page\_disk\_data缓冲区偏移的位置和编码方式（这个位置和编码方式在WT上定义成一个wt\_cell对象，在后面的K/V cell章节来分析），通过这个信息偏移位置信息就可以访问到这一样在disk\_data缓冲区中的K/V内容。

每一个wt\_row对象在row\_update\_array数组中对应一个mvcc list对象，mvcc\_list与wt\_row是一一对应的，list当中存储对wt\_row修改的值，修改的值包括值更新和值删除,是一个无锁单向链表。

相邻的两wt\_row之间可能不是连续的，他们之间可以插入新的单元，例如row1(key = 2)和row2(key=10)直接可以插入3 和5，这两个row之间需要有一个排序的数据结构（WT用skiplist数据结构）来存储插入的K/V，就需要一个skiplist对象数组page\_insert\_array与row array对应。这里需要说明的是图6当中红色框当中的skiplist8，它是用于存储row1(key=2)范围之前的insert数据，图6中如果有key =1的数据insert,那么这个数据会新增到skiplist8当中。那么图6中row与insert skiplist的对应关系就是：

**row1之前的范围对应insert是skiplist8**

row1和row2之间对应的insert是skiplist1

row2和row3之间对应的insert是skiplist3

.

**row7之后的范围对应的insert是skiplist7**

## wt\_row结构

从上面对应page的整体分析来看在WT的page中，row对象是整个row leaf page的关键结构，row其实就是K/V位置的描述值（kv\_pos），它的定义:

wt\_row{

uint64 kv\_pos; //这个值是在page读入内存时根据KV存储在page\_disk的位置确定的

}

这个kv\_pos的对应的数据有对应的K/V存储位置信息，kv\_pos表示这个K/V的值有三种方式，结构如下图：



图7

上图中的wt\_row对应的空间上都有2个bit的flag，这个flag的值：

CELL\_FLAG: 0x01,表示这个row用cell对象来k/v标示存储位置的,因为key和value的值有可能很长，一个page存储不下，这个时候引入cell只是存储这些超长值对应的overflow page的索引值。

K\_FLAG: 0x02,表示这个row只标示了k/v中key的存储位置，但是value比较大，value是CELL\_FLAG方式标示位置存储的，因为value的CELL是紧跟在key存储的位置后面，找到key就能读取value的cell并找到value。

KV\_FLAG: 0x03,表示这个row同时标示了k/v中key和value的存储位置。

## cell结构

cell是一个值key或者value）信息被序列化后的数据块，cell在磁盘上和在内存中内容是一致的，它是根据值（key或者value）内容、长度、值类型序列化构建的。在内存中cell是存储在page\_disk缓冲区中，在磁盘上是存储在extent body上，在读取cell的时候需要根据cell数据内容进行发序列化得到一个cell\_unpack内存结构对象，让后再根据这个cell\_unpack对象中的内容来读取这个值（key或者value）。以下是它们之间的结构关系：



图8

那么在row对象中的CELL\_FLAG row、K\_FLAG row和KV\_FLAG row对应的值是怎么产生的呢？其实是在page的数据从磁盘读到内存中时，先会对整个page\_disk按照cell为单位转化成cell\_unpack,并根据cell\_unpack中的信息构造row的这三种格式，这样做的目的是**让能生成K\_FLAG /KV\_FLAG的row在每次被访问的时候不需要去做这个过程的转化，加快访问速度。**

## 内存中的修改

### 跳表(skiplist)

### 新增的K/V结构

### 修改的value结构

## overflow page

## page内部的检索与增删改

# Disk extent结构

## extent address结构

## page header

## Block header

## 数据区

# Page的磁盘读写

## 读过程

## 写过程

### Page整体压缩

### 页内值字典引用存储

# 后记