zhuanlan.zhihu.com

【rocksdb源码分析】写优化之JoinBatchGroup

27-34 minutes

N次阅读rocksdb和leveldb源码后，我对它们的简答粗暴概括理解如下：

> 跟leveldb学习LSM-Tree及C++（C98）实践
> 跟rocksdb学习存储引擎的实现

rocksdb在leveldb的基础上做了大量（非常大）的优化，更适合在生产环境使用，我们自己的开源项目（pika、zeppelin）等都使用rocksdb作为存储引擎。

所以，我打算积累一个【rocksdb源码分析】系列，详细整理一下rocksdb的实现原理及相比较于leveldb，它在细节处做的优化。

本篇就先介绍一下rocksdb在写入时的一个优化点实现。

注：源码分析主要基于rocksdb v5.0.1及v5.4.5两个版本，后者相较于前者在代码结构及实现上都有较大改动，不过核心一样，后续内容也主要以这两个版本为主

这个函数主要做什么呢，leveldb和rocksdb都支持多线程，不过对于Write是单写者的实现，这就需要使用类似队列的东西将上层多线程的多个Writer进行排列，每次只允许队列头的Writer写db，队头Writer写完后再唤醒队列其他的Writer。leveldb在这里有一个优化，就是队头Writer在写db时，并不是只将自己的WriteBatch写完就拉倒，而是在写之前先将自己和它之后正在等待的其他Writer的WriteBatch一起打包成一个更大的WriteBatch，然后一起再写，这样，当它写完db唤醒其他Writer后，有一部分Writer会发现自己的活已经被做完了，直接返回。这样的实现可以提高写入速度，算是一个不小的优化。那么问题来了，rocksdb基于这之上还能做哪些优化呢？

2. 优化点

rocksdb在Writer之间Wait的地方做了优化，先看下leveldb这块是怎么做的：

```
Status DBImpl::Write(const WriteOptions& options,
WriteBatch* my_batch) {
  ......

  MutexLock l(&mutex_);
  writers_.push_back(&w);
  while (!w.done && &w != writers_.front()) {
```

```
    w.cv.Wait()
  }
  if (w.done) {
    return w.status;
  }
}
```

很简单，就是pthread_cond_wait。这样做有什么问题吗？呃...貌似有

> For reference, on my 4.0 SELinux test server with support for syscall auditing enabled, the minimum latency between FUTEX_WAKE to returning from FUTEX_WAIT is 2.7 usec, and the average is more like 10 usec. That can be a big drag on RockDB's single-writer design.

从FUTEX_WAIT到FUTEX_WAKE平均需要10us的时间，这对于单写着的引擎来说，代价的确不小，因外除过真正写引擎的时间，还有很大一部分时间用在了pthread_cond_wait及pthread_cond_signal上。

2. 如何优化

条件锁因为Context Switches而代价高昂，rocksdb通过一系列优化来尽量少用条件锁的使用并且尽可能的减少Context Switches。它将leveldb简单一条

pthread_cond_wait拆成3步来做:

- **Loop**
- **Short-Wait**: Loop + std::this_thread::yield()
- **Long-Wait**: std::condition_variable::wait()

下面来依次分析

1. Loop

这里主要就是通过循环忙等待一段有限的时间，大约1us，绝大多数的情况下，这1us的忙等足以让state条件满足（Leader Writer的WriteBatch执行完），而忙等待是占着CPU，不会发生Context Switches，这就减小了额外开销；

```
// On a modern Xeon each loop takes about 7
nanoseconds (most of which
// is the effect of the pause instruction), so
200 iterations is a bit
// more than a microsecond.  This is long enough
that waits longer than
// this can amortize the cost of accessing the
clock and yielding.
for (uint32_t tries = 0; tries < 200; ++tries) {
  state =
w->state.load(std::memory_order_acquire);
```

```
  if ((state & goal_mask) != 0) {
    return state;
  }
  port::AsmVolatilePause();
}
```

实现非常简单，循环200次（大约1us），每次循环判断条件是否满足，满足则返回。有个地方值得注意：

```
port::AsmVolatilePause();
```

这个是做什么用的呢？跟一下，它的实现是这样：

pause指令，查了一下文档，如下：

> *Improves the performance of spin-wait loops. When executing a “spin-wait loop,” a Pentium 4 or Intel Xeon processor suffers a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.*
> *An additional function of the PAUSE instruction is to reduce the power consumed by a Pentium 4 processor*

> *while executing a spin loop. The Pentium 4 processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.*
> *This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a pre-defined delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).*

可以看出，pause指令主要就是提升spin-wait-loop的性能，当执行spin-wait的时候处理器会在退出循环的时候检测到memory order violation而进行流水线重排，造成性能损失，pause指定则是告诉cpu，当前正处在spin-wait中，绝大多数情况下，处理器根据这个提示来避免violation，提高性能。

结合rocksdb，发现上面的200次循环中每次都会load state变量，检查是否符合条件，当Leader Writer的

WriteBatch执行完，修改了这个state变量时，会产生store指令，由于处理器是乱序执行的，当有了store指令后，需要重排流水线确保在store之后的load指令在执行store之后再执行，而重排会带来25倍左右的性能损失。pause指令其实就是延迟40左右个clock，这样可以尽可能减少流水线上的load指令，减少重排代价。

另外pause指令还可以减少处理器能耗，不过这不是我们关心的。

2. Short-Wait

如果能够准确预测未来，那么rocksdb其实只需要Loop和Long-Wait两种策略即可，预测到等待时间很短就用Loop，等待时间很长则只能用Long-Wait。但没有预言家，不可能每次提前准确知道该用那个，所以rocksdb才有了Short-Wait策略，这个一个灵活测策略，先来看实现：

```
while ((iter_begin - spin_begin) <=

std::chrono::microseconds(max_yield_usec_)) {
  std::this_thread::yield();

  state =
w->state.load(std::memory_order_acquire);
  if ((state & goal_mask) != 0) {
```

```
    // success
    would_spin_again = true;
    break;
  }

  auto now = std::chrono::steady_clock::now();
  if (now == iter_begin ||
      now - iter_begin >=
std::chrono::microseconds(slow_yield_usec_)) {
    // conservatively count it as a slow yield if
our clock isn't
    // accurate enough to measure the yield
duration
    ++slow_yield_count;
    if (slow_yield_count >=
kMaxSlowYieldsWhileSpinning) {
      // Not just one ivcsw, but several.
Immediately update ctx
      // and fall back to blocking
      update_ctx = true;
      break;
    }
  }
  iter_begin = now;
}
```

和Loop一样，还是循环判断state条件是否满足，满足则跳出循环，不满足则std::this_thread::yield()来主动让出时间片，这里的循环不是无止境的，最多持续max_yield_usec_us（需要用enable_write_thread_adaptive_yield=true来打开，打开后默认是100us）。这么做是可取的，因为yield并不一定会发生Context Switches，如果线程数小于CPU的core数, 也就是每个core上只有一个线程的时候，是不会发生Context Switches，花费差不多不到1us。不同于Loop每次固定循环200次，Short-Wait循环的上限是100us，这100us使用CPU的高占用(involuntary context switches)来换取rocksdb可能的高吞吐，如果很不幸每次100us后state还没有满足条件而进去最后的Long-Wait，那么这100us做了很多无谓的Context Switches，消耗了CPU。有没有什么办法来动态判断在Short-Wait中是否需要break出循环直接进行Long-Wait呢？rocksdb是通过yield的持续时长来做的调整，如果yield前后间隔大于3us，并且累计3次，则认为yield已经慢到足够可以通过直接Long-Wait来等待而不用进行无谓的yield。

另外，进不进行Short-Wait其实也是有条件的，如下：

```
if (max_yield_usec_ > 0) {
  update_ctx =
Random::GetTLSInstance()->OneIn(256);
```

```
  if (update_ctx ||
ctx->value.load(std::memory_order_relaxed) >= 0)
{
    ......
  }
  ......
}
```

首先max_yield_usec_大于0，其次update_ctx等于true（1/256的概率）或者ctx->value大于0；ctx->value就是这个动态的开关，如果在Short-Wait中成功等到state条件满足，则增加value，如果Short-Wait没有成功等到条件满足而最终还是靠Long-Wait来等待，则减少这个value，然后通过它是否大于0来决定下次是否需要进行Short-Wait，可以看到，如果Short-Wait大量命中，则value一定会远大于0，每次都进行Short-Wait。value的更新策略如下：

```
if (update_ctx) {
  auto v =
ctx->value.load(std::memory_order_relaxed);
  // fixed point exponential decay with decay
constant 1/1024, with +1
  // and -1 scaled to avoid overflow for int32_t
  v = v + (v / 1024) + (would_spin_again ? 1 :
-1) * 16384;
```

```
  ctx->value.store(v, std::memory_order_relaxed);
}
```

3. Long-Wait

很不幸，前两步的尝试都没有等到条件满足，只能通过代价最高的std::condition_variable::wait()来做的，这里就和leveldb的逻辑一样了，不过就在这里rocksdb还是做了优化：

```
uint8_t WriteThread::BlockingAwaitState(Writer* w, uint8_t goal_mask) {
  // We're going to block.  Lazily create the mutex.  We guarantee
  // propagation of this construction to the waker via the
  // STATE_LOCKED_WAITING state.  The waker won't try to touch the mutex
  // or the condvar unless they CAS away the STATE_LOCKED_WAITING that
  // we install below.
  w->CreateMutex();
  ......
}
```

直到需要进行std::condition_variable::wait()的时候，才创建Writer的std::condition_variable变量。还是可以看出

rocksdb对于single-writer的写入流程做了尽可能极致的优化来最大程度上提高性能。

总结

1. leveldb的一条pthread_cond_wait被rocksdb扩展出这么多的步骤及策略，目的还是为了尽可能的优化性能。的确，基础组件或者服务写的好不好直接决定着上层应用的性能，对于可能存在的瓶颈一定要吃透直到最优。
2. rocksdb默认打开Short-Wait的（enable_write_thread_adaptive_yield = true）max_yield_usec_默认是100us，可以通过配置项write_thread_slow_yield_usec来调整，增大它就是靠消耗更多CPU（Short-Wait持续时间越长）来提高rocksdb的吞吐。另外slow_yield_usec_默认是3us，可以通过配置项write_thread_slow_yield_usec来调整，增大它则会导致slow_yield_usec_门槛变高，减少Short-Wait半路break出去直接进行Long-Wait的概率，同样是用CPU来换吞吐。