Scalable Commit 功能开发文档

修订历史

版本	修订日期	修订描述	作者	备注
Cedar 0.2	2016-06-15	Scalable Commit 功能开发文档	周欢 胡爽	

1 总体设计

1.1 综述

> Cedar是华东师范大学信息科学与工程研究院院基于OceanBase 0.4.2 研发的可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务并支持数十万的TPS和数百万的QPS。其内存事务引擎使用多版本并发控制实现读写互不阻塞，提供快照读的事务隔离级别；使用经典的两阶段行锁避免写写冲突，提供读已提交的事务隔离级别；使用logging ahead记录redo log来持久化事务，并且通过实时同步备机来保证服务的持续可用。事务执行分为3个阶段，分别是事务处理，事务提交和事务发布，其中事务处理阶段保证了事务的一致性和隔离性，事务提交阶段保证了事务的原子性和持久性。虽然Cedar采用成组提交，预解锁等技术来优化事务提交性能，但是单线程处理事务提交的设计仍然限制了内存事务引擎UpdateServer的提交性能。为了提高UpdateServer的吞吐量，我们使用可扩展的事务提交（Scalable Commit）技术来优化事务提交过程。

1.2 名词解释

主控服务器（RootServer, RS）：Cedar集群的主控节点，负责管理集群中的所有服务器，以及维护tablet信息。

更新服务器（UpdateServer, UPS）：Cedar的内存事务引擎。

基准数据服务器（ChunkServer, CS）：Cedar的分布式存储引擎。

合并服务器（MergeServer, MS）：负责接收并解析客户端的SQL请求，经过词法分析、语法分析、查询优化等一系列操作后发送到CS和UPS，最后合并请求结果。

1.3 功能

为提高提交线程的处理能力，需要优化提交过程中的大量内存拷贝和计算，在保证日志提交顺序的基础上，实现多线程填充日志缓冲区并减少对待发布事务的计算。在集中式内存事务引擎中，Logging是唯一的全局同步点，因此Updateserver事务提交阶段需要严格保证日志的先后顺序，即日志内容在日志缓冲区中的位置。为实现多线程并发填充缓冲区，工作线程需在填充缓冲区之前确定日志顺序和日志内容，主要包括日志号（log_id/LSN）、事务版本号（trans_id）、日志所在文件号（file_id）、日志所在的文件偏移（file_offset）以及memtable校验和。具体实现过程中需要满足以下条件：

1) 并发释放行锁，维护多版本控制信息前决定log_id,trans_id,trans_id是递增的时间戳；

2) 并发填充缓冲区(fill_log)之前，必须给日志分配一个缓冲区位置，称为占位操作；

3) 并发填充缓冲区之前必须决定memtable的checksum；

4) 必须提前给NOP和SWITCH_LOG日志预留log_id和缓冲区位置；

5) 写日志之前必须决定日志所在的file_id和file_offset;

6) 每次写盘的日志不能超过2M。

为充分利用多核扩展性实现并发的关键用原子操作来决定日志顺序。优化后的UpdateServer存在两种线程分别为多事务工作线程（workers）和单事务提交线程（committer），工作线程负责并发处理事务、占位、解行锁、生成日志、填充缓冲区、提交写盘任务，最后响应客户端并回收事务上下文，事务提交线程负责日志写盘、同步备机、判断待发布的事务并唤醒工作线程响应客户端。在Scalable Commit架构中，事务执行过程分为四个阶段：事务处理、事务预提交、事务提交和事务发布。具体的执行流程如下：

1. 事务处理阶段，由多线程对并发的事务执行预处理，包括加行锁，进行逻辑判断（是否能执行insert等操作）；然后将待更新数据写入事务上下文的临时空间；
2. 事务预提交阶段，多线程并发的为待提交事务抢占缓冲区的位置；若抢占成功后则生成日志信息，若不成功则继续抢占；然后采用Early Lock Release技术提前释放行锁，维护多版本并发控制信息；填充缓冲区将预提交事务压入工作线程的私有队列等待提交线程唤醒，再由缓冲区最后一个完成日志填充的工作线程更新trans_committed_id将缓冲区作为一个任务压入提交线程队列中；
3. 事务提交阶段，由单线程处理当前提交线程队列的任务，如刷磁盘同步备机；同时提交线程循环判断同步备机的最大日志号，确定已经完成提交的事务，用已完成提交事务的最大trans_id更新系统发布版本号并清空当前缓冲区，最后唤醒工作线程私有队列中等待的事务；
4. 事务发布阶段，由多线程循环地判断私有队列中事务是否被唤醒，如果被唤醒则多线程并发的更新memtable信息（row_counter_、checksum和last_trans_id）,然后释放事务上下文并响应客户端。

Scalable Commit采用原子操作进行并发占位保证日志连续，实现多线程并发填充缓冲区提高事务提交性能。

1.4 性能指标

同等单机环境配置下，通过client工具测试，原单机性能是10万TPS，Scalable Commit提升到22万。同等三集群环境配置下，原三集群性能5.8万TPS，Scalable Commit提升到6.5万TPS。

2 模块设计

根据功能需求，Scalable Commit的实现，主要分为：

- 占位设计
- 日志缓冲区设计
- 特殊写事务设计
- 提交线程设计

2.1 占位子模块设计

2.1.1 设计概述

占位操作是指工作线程并发抢占缓冲区位置，以在填充缓冲区之前根据位置信息生成日志内容，包括log_id,、trans_id、file_id、file_offset和checksum。并发的关键是用原子操作决定id和偏移，但是上述所有id和偏移都至少是64bit，无法用一个原子操作决定。为了实现并发性，减少抢id和offset的冲突，占位操作主要的设计思路如下：

> - 将待提交的事务分成多个group并保证每个group内的log_id和trans_id都连续；
> - 每个group对应一个2M log_buffer，log_buffer总数可调。每个group有一个递增且连续的group_id；
> - group内的日志是成组提交的，因此在写磁盘时，只需确定group内第一个事务和最后一个事务的file_id、file_offset（对应Cedar中ObLogCursor数据结构）以及group内日志的总长度即可；
> - group间memtable checksum需要保证连续。Group之间采用不可交换算法计算出memtable checksum，group内每条日志采用可交换算法计算出checksum。(注: 当前版本没有实现checksum);
> - group内最后一条日志预留给NOP或者SWITCH_LOG。

2.1.2 设计实现

结构体FLogPos记录占位信息, 函数append()计算缓冲区位置。

```
struct FLogPos
{
  int64_t group_id_;     //group编号
  int32_t rel_id_;       //group内相对id
  int32_t rel_offset_;   //group内相对偏移
  int append(FLogPos& pos, int64_t len...);
}__attribute__ (( __aligned__( 16 ) ));
```

因为ClogPos刚好128bit，所以可以使用CAS保证原子性。具体占位流程如下：初始化全局变量FLogPos next_pos_；工作线程根据自己的日志长度，采用append方法乐观计算出缓冲区位置；并根据group是否切换，用对应当前缓冲区位置原子抢占更新全局next_pos_，若抢占成功，则根据group_id%log_buffer总数定位所属的group，生成日志信息并填充缓冲区；若失败则重新抢占。

2.2 日志缓冲区子模块设计

2.2.1 日志缓冲区结构

日志缓冲区是内存中临时存储日志的结构，它由多个2M大小的buffer组成，每个buffer称为一个group。每个group中记录了日志长度len_、日志个数count_、起始事务号start_timestamp_、起始日志号start_log_id_、group所属文件的起始start_cursor_和终止位置end_cursor以及group状态标识。

> Group结构体说明：
>
> - 每个group包含四种状态，分别为可重用，可设值，可填充，可提交;
> - 一个group的start_timestamp_是由当前group第一条日志的处理线程设置的;
> - 工作线程在填充日志缓冲区之前需要等待当前group为可填充状态；
> - 初始化和主备切换时，更新当前group的start_cursor_;
> - 提交线程刷磁盘之前需要等待当前group为可提交状态；
> - len_和count_是由group最后一条日志的处理线程设置；
> - 工作线程填完日志内容后对当前group的ref_cnt_原子加1，当ref_cnt_ = count_表示当前工作线程是最后一个完成group日志填充的线程，该线程需发起刷磁盘任务；
> - 在Cedar系统中，写日志事务类型分为特殊写和普通事务类型。特殊写事务不响应客户端need_ack_ = false,普通事务响应客户端need_ack_=true，通常特殊写事务和普通事务的日志存于不同的group中；
> - OB系统中，事务同步备机分为异步和同步两种模式。通过Sync_to_slave来区分两种模式并调用相应的刷磁盘函数（flush_log/async_flush_log）。

Group结构体定义如下所示：

```
struct LogGroup
{
  enum { READY = 1};
  volatile bool sync_to_slave_;      //group是否需要同步备机
  volatile bool need_ack_;           //group是否需要响应客户端
  volatile int64_t ref_cnt_;         //group中已完成填充的日志数
  volatile int64_t len_;             //日志长度
  volatile int64_t count_;           //日志总条数
  //ts_seq_=group_id表示group可以被重用
  //ts_seq_=group_id+READY表示group可以被填充
  volatile int64_t ts_seq_ CACHE_ALIGNED;
  volatile int64_t group_id_;        //group编号
  volatile int64_t start_timestamp_;//起始事务号
  volatile int64_t start_log_id_;    //起始日志号
  //last_ts_seq_=group_id+READY
  //表示group可以设置start_timestamp
  volatile int64_t last_ts_seq_ CACHE_ALIGNED;
  //前一条日志用的时间戳 在设置start_timestamp时使用
  volatile int64_t last_timestamp_;
  //log_cursor_seq_=group_id+READY表示group可以被提交
  volatile int64_t log_cursor_seq_ CACHE_ALIGNED;
  ObLogCursor start_cursor_;         //写日志的起始位置
  ObLogCursor end_cursor_;           //写日志的终止位置
  char empty_log_[LOG_FILE_ALIGN_SIZE * 2]; //存储SWITCH_LOG日志
  char nop_log_[LOG_FILE_ALIGN_SIZE * 2];    //存储NOP日志
  char* buf_;                       //存储日志
};
```

2.2.2 缓冲区填写流程

主要步骤如图1:

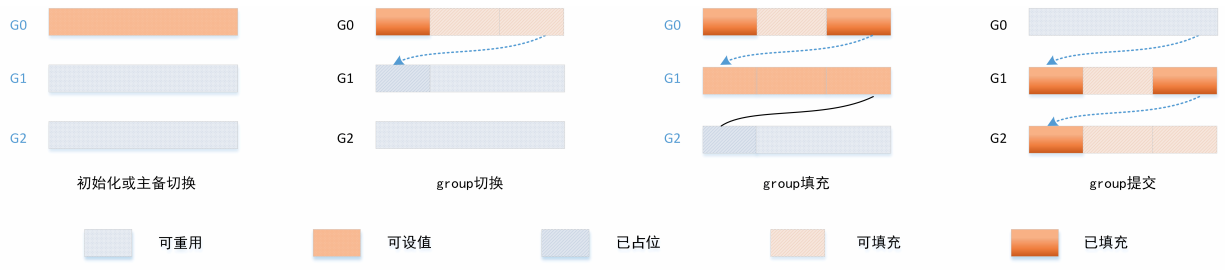

图1 缓冲区操作

1. 初始化或主备切换时，更新全局变量group_id_对应group(G0)的start_log_id_, last_timestamp_和start_log_cursor_，标识当前group G0可重用，可设值，G1、G2可重用；
2. group切换时，填充G0最后一条占位日志的线程标识G0可提交，等G1可重用后，标识G1为可设值。填充G1第一条占位日志线程要等G1可设值后，设置G1的开始时间

戳，标识G1可填充;

3. group填充时,G0的最后一个填充线程更新全局的committed_trans_id_，并将G0压入提交线程队列中;
4. group提交时，提交线程需等当前group为可提交状态，后等刷盘并同步备机成功之后标识G0可重用。

2.3 特殊写事务子模块设计

在Cedar系统中，与日志相关的写事务分为三类分别为普通写事务、系统维护型写事务和每日合并写事务。Cedar0.1中对所有写事务的处理逻辑如图2所示。

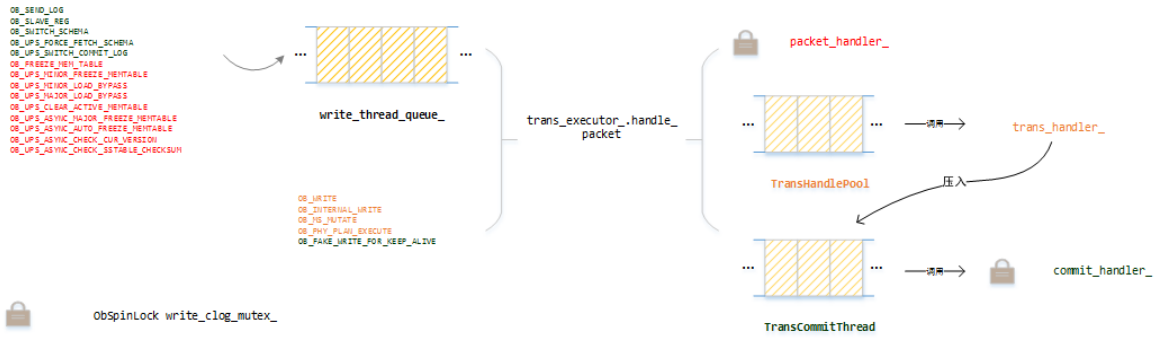

图2 Cedar0.1事务处理逻辑

其中黄色是普通写事务如OB_PHY_PLAN_EXECUTE，绿色是系统维护型写事务如OB_SWITCH_SCHEMA，红色是每日合并写事务如OB_FREEZE_MEM_TABLE。普通写事务由事务执行器处理，后交提交线程进行提交；系统维护型写事务先压入写线程队列，再交提交线程写日志；每日合并写事务先压入写线程队列中，然后再由写线程找到其对应的处理函数。

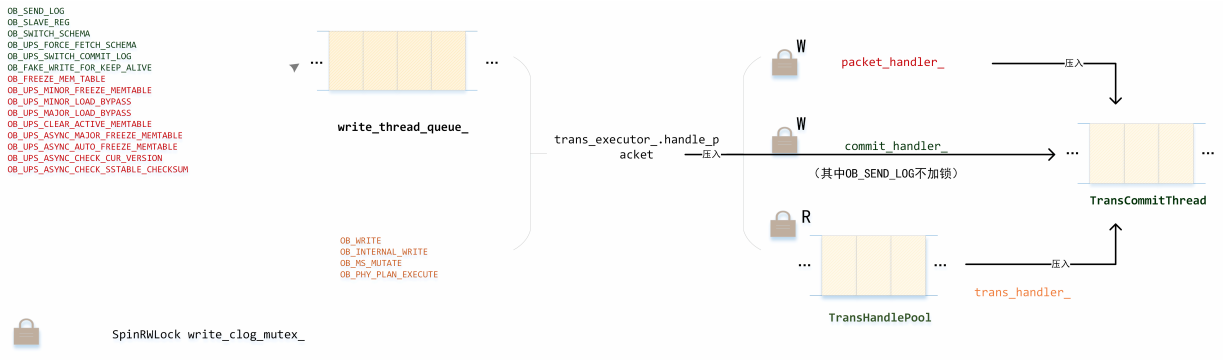

图3 Scalable Commit写事务处理逻辑

在Scalable Commit写事务处理逻辑如图3，事务提交线程只负责刷盘和同步备机以确定日志顺序，填充集中式日志缓冲区是由工作线程完成的。系统维护型写事务由写线程直接处理。其中写线程处理系统维护型写事务和每日合并写事务时仍采用CAS占位操作，在占位前需保证当前group为空即切换group，填充日志缓冲区之后切换group并把当前group压入提交线程队列，再由提交线程刷盘和同步备机。同时采用读写锁spinRWLock互斥写线程和处理线程的占位操作。

系统维护型写事务出现频率不高，其中OB_FAKE_WRITE_FOR_KEEP_ALIVE是每隔50ms才产生一次，因此采用互斥锁不会对系统性能造成太大的影响。另外系统维护型写事务OB_SEND_LOG是主UPS同步备机时产生的任务类型，然后由备UPS处理该事务将主UPS日志缓冲区的内容拷贝到备UPS的内存缓冲区中，因此可不加写锁。

2.4 提交线程子模块设计

2.4.1 总体设计流程

优化的目的在于消除单线程提交性能上的瓶颈。优化后事务提交由多个工作线程和单个提交线程来完成。提交时，多个处理线程根据事务上下文信息并发生成日志，并对日志缓冲区进行占位和填充，再将上下文信息记入线程的私有队列中（以等待publish）；多个事务日志将被填充到一个Group中，提交线程将填入缓冲区的日志刷盘同步备机，并通知一个刷盘成功的日志号，处理线程根据日志号将完成提交的事务publish。

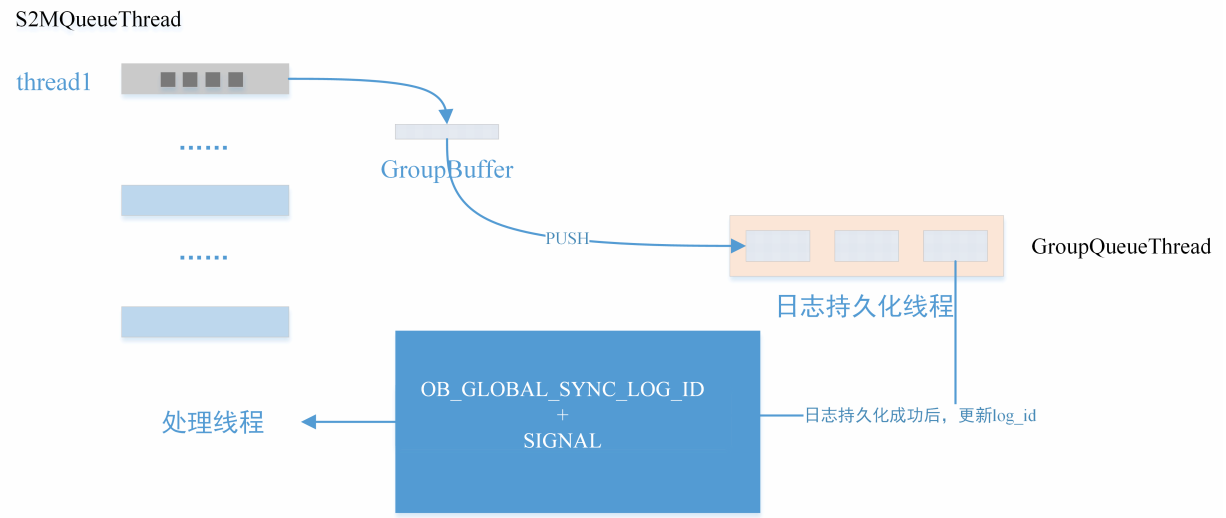

图4 提交线程流程

2.4.2 私有队列

每个工作线程都拥有用于存储事务上下文信息的私有队列ObCommitQueue，结构如下：

```
class ObCommitQueue
{
  public:
   int submit(ICommitTask *task);//将填充完缓冲区的任务压入私有队列
   int next(void * &ptask);
   int64_t get_total() const;
   int pop(void * &ptask);
  private:
   ObFixedQueue<void> task_queue_;
 }
```

1. 工作线程得到一个写任务，如果是特殊写任务，则在处理该任务后生成日志，排他抢占一整个Group，并提交到日志持久化线程；如果是事务处理写任务，则在处理该任务后生成日志与其他线程抢占并填充缓冲区，同时，将事务上下文的信息压入私有队列，等待日志持久化成功后做Publish&Free。
2. 工作线程判定日志持久化的成功进度OB_GLOBAL_SYNC_LOG_ID，将日志号小于此进度的事务上下文从私有队列逐个弹出，并进行publish、释放上下文以及回复客户端等流程。
3. 其中1,2均在工作现场的函数循环中依次执行，如果工作线程此时未得到任何的写任务，则线程会阻塞导致事务未及时提交释放，因此日志持久化线程还会利用回调唤醒signal所有工作线程。

2.4.3 日志持久化

这部分的基本流程概述为：首先是由单线程依次处理待提交的事务的group，进行刷磁盘和同步备机；在刷磁盘和同步备机完成后更新trans_published_id；最后是唤醒已经完成事务提交的处理线程。

日志持久化为单线程，其中队列flush_queue，用以更新事务的published_trans_id。根据group_id依次处理待提交的事务，分为两种情况：系统正常工作时间的普通group和系统空闲时间的特殊group。防止系统空闲时期flush_queue中的group信息没有及时处理，导致处理线程没有及时将事务任务提交返回客户端，在空闲时期会在持久化线程中提交一个包含特殊日志号(-1)的group。如果是特殊的group，则直接唤醒所有处理线程。如果是普通的group，在唤醒所有处理线程之后，判断当前的group的事务是否返回客户端，是则将对应的group的相关信息构成一个数据结构gtask压入flush_queue后，将当前日志缓冲区中的group进行持久化和同步备机操作并提前重用group。同时与备集群通信获取备机同步成功的日志id，与队列flush_queue中的gtask比较，小于成功同步的日志号的gtask被弹出，同时根据其信息将trans_published_id更新;将日志同步进度赋值给全局变量OB_GLOBAL_SYNC_LOG_ID；唤醒所有处理线程。

3 模块接口

缓冲区Group的个数是可调的系统配置项，目前默认设置为30个：

```
DEF_INT(commit_group_size, "30", "[1,100]", "commit_group_size");
```

集群启动时，在updateserver的启动项中添加参数G和修改的Group个数即可，修改的范围是1到100。