文章编号:1000-5641(2014)05-0149-15

OceanBase 内存事务引擎

李 凯, 韩富晟
(阿里巴巴集团,北京 100020)

摘要: OceanBase 是一个分布式可扩展的关系数据库,采用基线静态数据与动态增量数据分离存储的架构设计. 其内存事务引擎提供了动态数据的存储、写入和查询服务,用户写入的数据被存储在内存中称为 Memtable 的数据结构中. Memtable 及其周边的事务管理结构共同组成了内存数据库引擎,来实现事务的 ACID 特性. 在事务引擎中,通过多版本的并发控制技术实现读写相互不阻塞,实现只读事务满足"快照隔离"级别;通过经典的行锁方式实现多个写之间的并发控制,实现最高满足"已提交读"的事务隔离级别.

关键词: 关系数据库; 分布式系统; 事务; 互联网
中图分类号: TP333.1 文献标识码: A DOI:10.3969/j.issn.1000-5641.2014.05.013

Memory transaction engine of OceanBase

LI Kai, HAN Fu-sheng
(*Alibaba Group*, *Beijing* 100020, *China*)

Abstract: OceanBase is a distributed scalable relational database. Its storage architecture is designed by separating baseline static data and increment dynamic data, whose memory transaction engine, namely Memtable, provide dynamic data storage, write, and query, clients wrote data to the in-memory data structure. Memtable and some transaction management structures together form the in-memory database engine, which can achieve the transaction ACID properties. By-based multi-version concurrency control techniques to prevent reading and writing from blocking each other to achieve read-only transactions to meet the"snapshot isolation"level; Provide multi-write concurrency control by using classic row-lock technology to meet the"read committed" transaction isolation level.

Key words: relational database system; distributed system; transaction; internet

0 引 言

在互联网高速发展的今天,互联网公司不仅有大量非结构化的数据,例如网站访问日志等;也有很多结构化的数据,例如 Google 的搜索广告、淘宝/天猫的商品搜索广告的计费. 淘宝/天猫、Amazon 和 eBay 等的网上和移动购物,支付宝的网上和移动支付,等等,都是商务

收稿日期:2014-06
第一作者:李凯,男,阿里巴巴集团技术专家,研究方向为分布式系统与数据库. E-mail:yubai.lk@alipay.com.
通信作者:韩富晟,男,阿里巴巴集团技术专家,研究方向为分布式系统与数据库. E-mail:yanran.hfs@alipay.com.

交易和金融交易. 对这些数据的处理依赖于严格的数据库事务语义,即原子性(Atomicy)、一致性(Consistency)、隔离性(Isolation)和持久性(Durability)的保证. 同时也对数据库的存储容量和事务处理能力提出了越来越高的要求.

使用商业数据库、配置高端服务器和存储设备,在一段时间内确实能够解决互联网公司对数据库的需求,但是随着业务的发展,对存储容量和事务处理能力的需求越来越高,购买商业解决方案的成本变得难以接受,并且也同样遇到软硬件处理能力的瓶颈. 一些公司开始转向使用开源数据库软件如 MySQL,搭配廉价服务器,使用分库分表的方式对业务数据进行拆分存储,而拆分后的数据库扩展困难,运维成本高,不支持跨域分库分表的事务,且基于传统数据设计的 MySQL 并不能充分发挥当前大容量内存加 SSD 的主流服务器配置优势. 一些新兴的内存数据库如 MemSQL,能够充分利用大容量内存的优势,但是由于数据必须全部存放在内存中,而数据容量有限成本偏高,基于单机数据的设计,使得扩展能力也有限. Google 的 MegaStore 基于 BigTable/GFS 开发,有良好的可扩展性和较低的成本,但是对数据库事务支持有限,无法应用在对事务要求严格的电子商务领域. 阿里巴巴集团自主研发的数据库 OceanBase,适应电子商务领域对数据库事务的严格要求,充分发挥大容量内存和 SSD 发展的优势,以较低的成本提供海量存储和高性能事务,良好的可扩展性和容错能力很大程度上降低了运维成本.

OceanBase 是一个分布式的关系数据库,采用基线数据与动态数据分离存储的架构设计,其中基线数据被划分为多个有序的分块,称为 Tablet,每个 Tablet 存储若干个副本随机地分布在机群的多台机器上;动态数据就是一段时间内的数据更新记录,每次查询需要将基线数据与动态数据合并后产生最终结果返回给客户端. 整个机群主要由 4 个模块组成:Chunkserver 提供基线数据的存储和查询服务,定期将本地保存的基线数据与动态数据合并,生成新版本的基线数据;Updateserver 提供动态数据的存储、查询和写入服务,每隔一段时间将当前动态数据"冻结",并分配一个"冻结版本号",用于 Chunkserver 合并新版本的基线数据;Rootserver 提供 Tablet 在多台 Chunkserver 上位置信息的存储和查询服务,负载平衡、迁移、复制等的调度管理,以及 Schema 信息的管理;Mergeserver 实现 SQL 接口,处理客户端的查询和写入请求,将 SQL 请求转化为 Updateserver 和 Chunkserver 认识的执行计划,同时负责从多台机器接收查询结果后的合并和处理.

如上所述,Updateserver 提供了动态数据的存储、写入和查询服务,任何数据写入都将最终被发送到 Updateserver 执行,并存储在内存中称为 Memtable 的数据结构,上面提到的冻结则意味着当前 Memtable 停止数据写入,构造新的 Memtable 来接收后续写入的数据. Memtable 及其周边的事务管理结构共同组成了 Updateserver 的内存事务引擎,实现满足 ACID 特性的数据库事务.

内存事务引擎提供交互式事务接口,使用多版本的并发控制技术实现读写相互不阻塞,提供快照读事务;经典的两阶段行锁方式实现写的并发控制[1]. 使用 logging ahead 方法的记录 redolog 来持久化事务[2],并且通过同步实时的备机来保证服务的持续可用. 通过批处理、预解锁、并发提交等技术,来优化事务提交性能. 本文第 1 节概述内存事务引擎,第 2 节介绍并发事务设计,第 3 节介绍事务引擎的持久化与恢复,第 4 节介绍性能优化技术,最后对整个 OceanBase 内存事务引擎做出总结,展望未来的发展方向.

1　内存事务引擎概述

1.1　总体架构

如图 1 所示，内存事务引擎主要由 TableMgr、SessionMgr、LockMgr 和 TransExecutor 组成，其中 TableMgr 管理了多个版本的冻结 Memtable 和一个活跃 Memtable. 数据总是被写入活跃 Memtable，而达到冻结的触发条件时，活跃 Memtable 被转为冻结，然后构造新的活跃 Memtable 来接收数据写入. SessionMgr 为每个活跃事务维护一个描述符到事务上下文结构 SessionCtx 的映射. LockMgr 为事务执行过程中提供行锁加锁和解锁实现，在每个读写事务的上下文中维护两阶段锁. TransExecutor 是事务完整流程的执行器，在它维护的线程池中调用上述 TableMgr、SessionMgr 和 LockMgr 的接口执行事务预处理和提交.

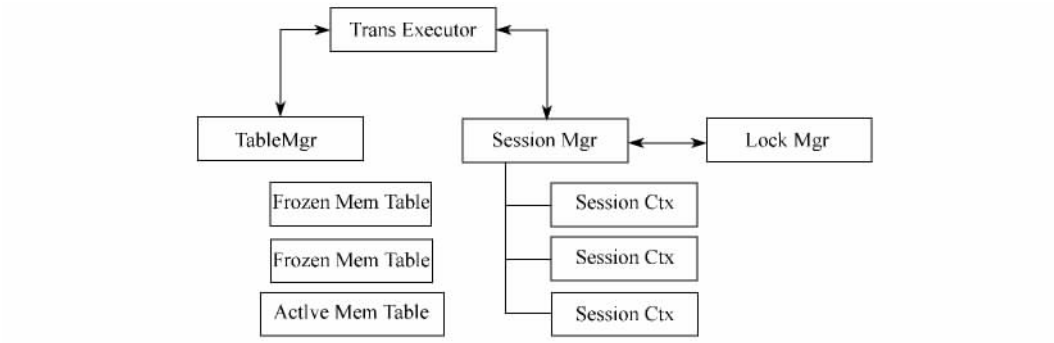

图 1　内存引擎结构

Fig. 1　Memory engine structure

● 事务执行流程

需要注意的是，本文所说的事务，既包括 begin...commit 这种形式的事务，同时也包括单条 autocommit 的语句. 如图 2 所示，事务在执行分为 3 个阶段，分别是预提交、提交和发布. 在预提交阶段，由多线程对并发的事务执行预处理，包括加行锁和进行逻辑判断（如能否执行 insert/update/delete 语句，以及上述语句中 where 条件的判断和数据读取），然后将待更新数据写入事务上下文的临时空间. 在确定事务要提交（收到 commit 或单条 autocommit）的情况下，由多线程执行提交操作：即申请事务版本号、提交对数据的修改和释放行锁. 提交操作完成后，由单个线程执行发布操作：即同步 redolog 和原子性地发布当前事务. 发布阶段完成后，事务对数据的修改才可以被后续开始的其他事务读取到.

● Memtable 内存结构

Memtable 是内存事务引擎用于存储数据和处理查询的核心结构，数据以行为单位在 Memtable 中存储，每行数据所使用的内存从 Memtable 内部的内存池分配，通过基于行主键的索引提供查询功能，包括 btree 范围索引和 hash 单行索引. 其中 Memtable 与事务管理器（SessionMgr）配合实现多版本并发控制，而与锁管理器（LockMgr）配合实现两阶段锁并发控制，这 2 种并发控制将在后面的章节中详细说明.

1.2　内存存储格式

在经典的数据库实现中，每行数据以稠密格式存储在数据块中，对行内容的修改将在数据块上就地修改，同时保存 undo 信息用于处理事务回滚和一致性读. 而在 OceanBase 内存

事务引擎中,为了更好地管理和使用内存,存储的则是对数据修改的操作记录,每次读取时需要将内存事务引擎中的数据修改记录应用到基线数据上[3]. 因此,每行数据以稀疏格式存储,仅仅保存被修改的列的值(或修改操作如"+1").

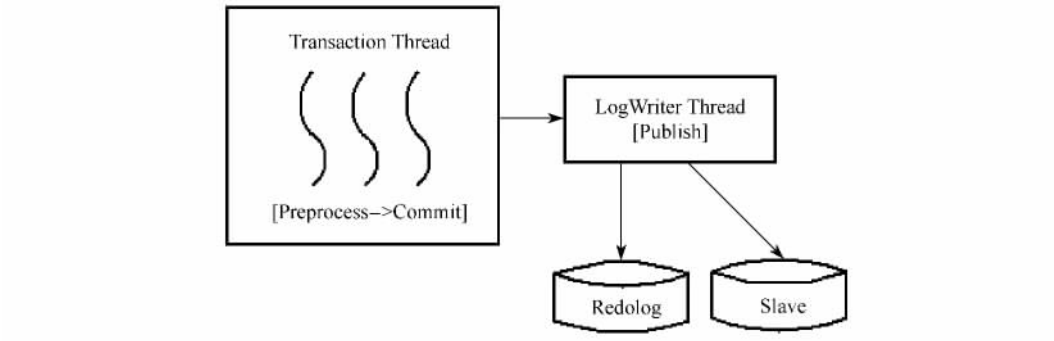

图 2 事务执行流程

Fig. 2 Transaction execution process

如图 3 所示,每次事务对行数据的更新记录都保存在一个变长的内存块中,多次事务保存的内存块按时间顺序串成链表,读取到这一行的时候,从最早的数据块开始遍历,将对同一列的更新记录合并,然后应用到基线数据上并生成最终数据. 随着对同一行多次执行更新,上述链表会变得越来越长,读取时合并计算的代价也会越来越大,因此在链表超过配置的长度后,多个块将被合并为一个块(称为 RowCompaction),减少读取时遍历链表合并的代价.

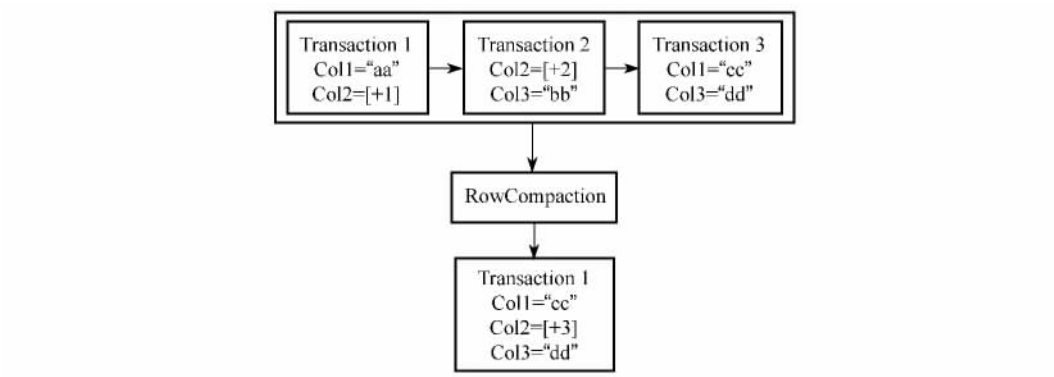

图 3 多次修改的合并

Fig. 3 RowCompaction

因为活跃表被"冻结"后,将会与基线数据进行合并,合并完成之后的读写操作将不再需要这个冻结表,其占用的内存也可以被整体释放掉,因此 Memtable 采用了最为简单的内存管理策略,活跃表在处理数据更新时申请的内存,只申请不释放,而是在冻结表被释放的时候一次性释放掉. 如上述执行 RowCompaction 后,3 个块合并成 1 个块,原先的 3 个块不会立即被释放,而是等待冻结后集中释放.

由于采用了上述简单的内存管理方式,在活跃表整个生存周期内,它管理的内存只申请不释放. 因此,为了节省内存的使用,在上述变长内存块中,多列数据以压缩格式存储,每次

读取的时候需要将数据解压缩后再进行处理. 数据压缩算法简单但行之有效：对于整数类型(整数和时间类型)根据其数值大小分别存储为 1/2/4/8 byte，对于字符串类型如果长度小于 8 byte 则在内存块中就地存储，否则使用 8 byte 保存指向外部存储空间的指针.

1.3　并发索引结构

内存引擎提供针对行主键的单行和范围查询，为了实现最优的查询性能，实现了 B+tree 和 Hash 的双主键索引设计，范围查询通过 B+tree 来服务，单行的随机查询则通过 hash 来服务. 由于我们使用了 SessionMgr 和 LockMgr 实现了事务的并发控制，使用 Memtable 管理内存，因此对于底层索引结构的需求则弱化为可支持并发读写的 KeyValue 结构. 下面将对两种索引结构进行逐一说明.

● B+tree

对于并发 B+tree，可以使用精细控制的小粒度锁来处理并发写入，使用 copy on write 技术来实现读写互不阻塞[4]. 而与经典的 B+tree 实现不同，虽然数据都存储在叶子节点，但是为了简化并发处理的逻辑，不同于传统的 B+树，因此并没有将叶子节点串联起来. 如下图所示，插入 Key-Value 到 B+tree 的过程为以下 3 个步骤：

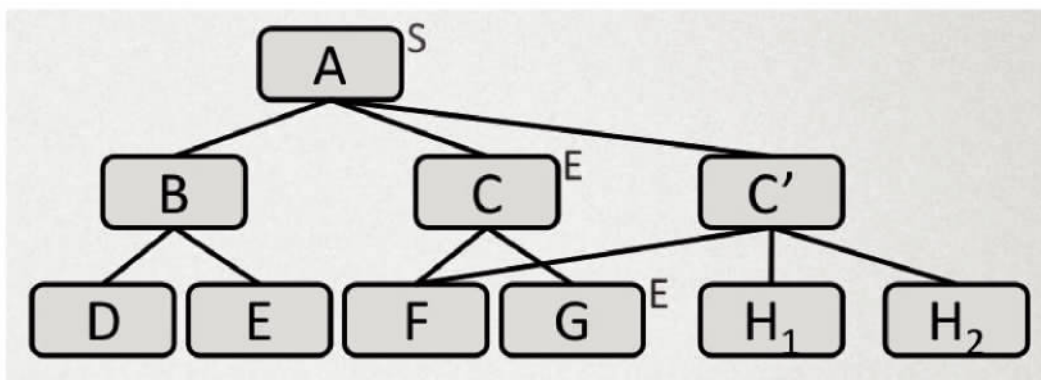

图 4　B+tree 结构

Fig. 4　B+tree structure

(1) 从根节点开始遍历查询 KV 要插入的位置，并在遍历过程中对路径上的节点加共享锁；

(2) 将叶子节点的共享锁升级为互斥锁，拷贝当前叶子节点，将 KV 插入拷贝出的叶子节点，如果节点不分裂，则原子的修改其父节点中指向它的指针即可；

(3) 如果节点需要分裂，则将其父节点的共享锁升级为互斥锁，拷贝父节点，然后将分裂后的索引插入拷贝出的父节点，如果父节点还需要分裂则递归执行此操作；

需要注意的是，由于共享锁的获取方向是自顶向下，而互斥锁则是随着分裂自底向上，为了避免出现死锁，获取共享锁的方式为 try_lock，在加锁失败的情况下，将路径上已经加成功的锁释放掉，然后重新遍历.

● Copy on write 内存回收的处理

在修改 B+tree 节点的过程中，除了原子替换子节点指针外，对节点的修改都需要先拷贝一次，在拷贝出的副本上进行修改，而在这个过程中原节点可能还会被读取到，因此，原节点不能立即删除，而是需要暂时保存直到确认不会再被读取时才删除.

这需要引入多版本的概念,为每次对 B+tree 修改操作维护一个 VersionNode 对象,当前修改操作过程中拷贝的多个原节点指针都保存在这个 VersionNode 中;维护一个全局版本号 V,每次修改操作开始前将 V 加 1,并将加 1 后的值保存在 VersionNode 中,修改操作结束后将 VersionNode 保存在线程本地,等待后续的内存回收. 每次读写操作开始前,将当前全局版本号保存在线程本地,读取操作执行完成后将线程本地保存的版本号置为无效.

当线程本地缓存的 VersionNode 关联的内存过多时,则执行内存回收逻辑,遍历所有线程局部的有效版本号,取得最小值 MinV. 表示版本号小于 MinV 的 VersionNode 引用的 B+tree 节点都不会再被引用,可以回收[5].

遍历的多线程问题,因为遍历多个线程局部版本号的操作并非原子,可能出现遍历得到的 MinV 大于实际 MinV 的情况(如线程拿到全局版本号后,到保存到线程本地的过程中,遍历操作就执行完了),因此每次执行读写操作,线程将全局版本号保存在本地前,先将全局版本号保存到临时变量中,然后将这个临时变量保存到线程本地之后,对其进行二次检查,如果全局版本号已经变化,则重新执行上述操作;对于回收逻辑,则需要在遍历线程局部版本号之前,先单独保存当前全局版本号到临时变量中,取 MinV 与这个临时变量的最小值作为最终的 MinV.

按操作时序举例说明如下:

1. 当前全局版本号为 10;
2. 只读操作 $T1$
   (a) 将全局版本号 10 保存在线程本地
   (b) 检查全局版本号 10 是否变化,如果变化则重新执行(a)
3. 读写操作 $T2$:
   (a) 将全局版本号 10 保存在线程本地
   (b) 检查全局版本号 10 是否变化,如果变化则重新执行(a)
   (c) 将全局版本号加 1 得到 11
   (d) 构造对象 VersionNode,将 11 保存在 VersionNode 中,$T1$ 执行过程中拷贝的源节点指针都保存在这个 VersionNode 中
   (e) $T2$ 结束,将线程本地保存的版本号改为 INT64_MAX
   (f) 将 VersionNode 串在全局链表中等待回收逻辑处理
4. 回收操作
   (a) 保存当前全局版本号 11
   (b) 遍历所有线程中保存的版本号取得最小值为 10
   (c) 取得 min(11,10)为 10 作为内存回收的最大版本号
   (d) 遍历全局 VersionNode 链表,回收版本小于 10 的 VersionNode
5. 只读操作 $T1$
   (a) $T1$ 结束,将线程本地保存的版本号改为 INT64_MAX
6. 回收操作
   (a) 保存当前全局版本号 11
   (b) 遍历所有线程中保存的版本号取得最小值为 INT64_MAX
   (c) 取得 min(11,INT64_MAX)为 11 作为内存回收的最大版本号

(d) 遍历全局 VersionNode 链表，回收版本小于 11 的 VersionNode

- Hash

并发 Hash 的实现比较简单，使用一个称为 BitLock 的结构，为每个哈希桶维护一个 bit 的锁标记，在查询或修改这个桶的时候对这个 bit 原子置为 1，表示互斥占用. 为了实现简单，哈希桶的数组使用了一整段连续内存而非 2 维数组形式，因此在大内存机器这个数组可能长达 10G 以上，为了处理初始化的时候对整个数组 memset 过慢的问题，引入延迟初始化的设计，即每 64K 大小的连续桶作为一个初始化单位，使用一个 byte 来作为初始化标记和并发锁标记，当查询或插入操作涉及到某个 64K 块的时候，再对这块内存调用 memset，调用的时候要使用上面提到的 bit 标记避免多线程同事操作的问题.

2　并发事务设计

OceanBase 使用 mvcc 技术实现只读事务与读写事务相互不阻塞，每次对行列数据的修改并非在数据的存储位置上就地修改，而是保存了一份以事务版本号标记的修改记录，在不考虑每日合并机制的情况下，可以认为在内存表中保存了每一次事务的修改历史，并且可以读取每一行的任意历史快照，但是实际上由于内存的限制，不可能一直保存数据的修改历史. 因此，在每次每日合并时，截至每日合并之前的所有修改历史将被合并. 如图 5 所示，原本保存的四个历史版本，在经过每日合并之后，snapshot[1-3]被合并为 snapshot3 当时的快照，后续只能读到 snapshot3 和 4 这两个历史快照的数据，而 snapshot3 之前的 snapshot1 和 2 已经被丢弃.

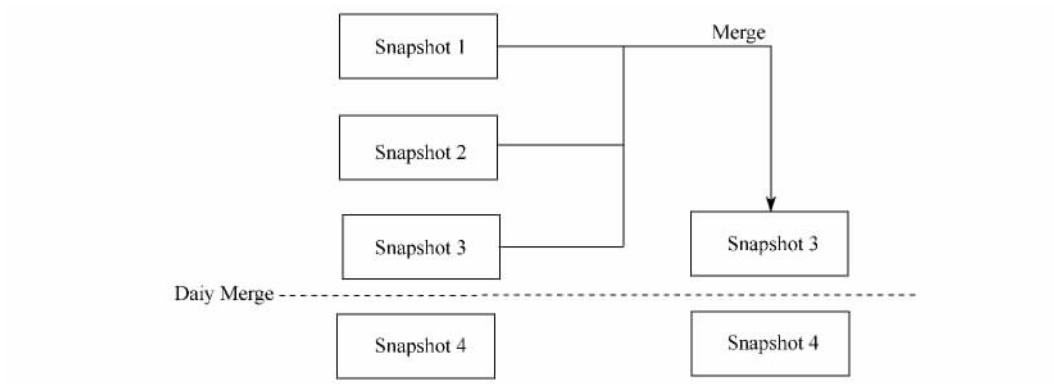

图 5　多版本与每日合并

Fig. 5　Multiple version and daily merge

2.1　事务的隔离级别

OceanBase 提供 read commited 和 snapshot read 两种隔离级别的事务，其中 snapshot read 利用上一节提到的多版本技术，提供能够持续较长时间(一个冻结周期内)的只读事务，这里不再赘述. 而 read committed 事务的行为与 Oracle 一致，即保证无论是否在事务(begin … commit)中，每条语句都能够读到这条语句开始之前，其他事务已经提交的修改. 同时还保证了被 Oracle 称为 Statement-Level Read Consistency 的隔离行为[6]，即语句运行在当前语句开始之前的快照基础上，如果语句产生的执行计划涉及到多次与内存事务引擎交互(如 select 子查询，索引查询，根据索引修改等)，那么在语句执行过程中其他事务提交的修改，在

当前语句执行过程中都保证读不到到这些修改.

● Transaction Set 的保证[7]

上面提到的语句级别的 read consistency,需要处理快照冲突的情况,即一条语句 S 涉及多次与内存事务引擎交互,在这个过程中有其他并发的事务修改并提交了行 R,之后语句 S 要修改行 R,因为每条语句要运行在当前语句开始之前的快照基础上,同时还要避免 lost update,因此当遇到这种情况时,语句 S 发现在自己运行过程中行 R 已经被其他事务修改,那么内存事务引擎就将语句 S 已经做的修改回滚,然后在一个最新的快照基础上重试执行语句 S,重试若干次都失败后,说明对行 T 的修改的并发度比较大,对客户端返回执行失败.

2.2 多版本并发控制

(1) 事务版本号的分配

如上一段所述,在内存中为每次事务修改都保存了一份历史记录,这个记录由一个全局唯一的事务版本号标记,而 mvcc 提供的快照读取特性需要保证能够读到任意一个有效历史时刻上的数据. 比如,行 A 保存了版本号为 1、3、5 的事务已经提交的修改,行 B 保存了版本号为 2、4、6 的事务已经提交的修改,要读取快照点为 5 的历史版本,则要读取行 A 上事务版本号为 5 的版本,和行 B 上事务版本号为 4 的版本. 这里可以看出,事务版本号要满足递增特性,以保证读取到正确的历史版本.

某些数据库在事务开始的时候即分配了事务版本号,在执行事务过程中,写入的数据都以这个版本号进行标记,这种情况下基本可以保证事务版本号的递增特性,但是需要处理较早开始的事务分配了一个较小的版本号却较晚提交的情况. 比如,起始状态,行 $A=1$,行 $B=1$. 开启事务 $T1$,版本号为 1,修改了行 $A=2$,并标记事务版本号为 1;之后开启事务 $T2$,版本号为 2,修改了行 $B=3$,并标记事务版本号为 2;这时提交事务 $T2$,记录当前已提交的事务版本号为 2;之后读取 $T2$ 提交之后的快照,即截至版本号为 2 的快照,如果直接读取事务版本号小于 2 的数据,则行 A 上版本号为 1 的修改(即 $A=2$)也会被读取出来,不满足事务隔离性,因此进行快照读取的时候需要对当前尚未结束事务的修改进行过滤,如示例中的版本号为 1 事务的修改 $A=2$,将不能被读取出来,而只能读取到 $A=1$. MySQL 使用了类似的实现方式,这种方式虽然事务版本号的维护比较简单,但是快照读取逻辑比较复杂,并且由于事务版本号顺序没有严格反映事务的提交顺序,为备机保证与主机一致的事务性回放增加了处理难度[8].

因此内存事务引擎没有采用在事务开启时分配事务版本号的方式,而是在事务提交时按照严格的事务提交顺序分配递增的事务版本号. 这意味需要在事务提交时将分配到的事务版本号写回到本次事务修改的所有数据上去,对比 Oracle 的实现,考虑到向回滚块(undo block)写回事务版本号,可能需要将回滚块从磁盘加载回内存,并且当事务修改的行过多时,也会使得写回操作耗时过长,因此 Oracle 并没有在提交事务的时候立即将事务版本号写回所有的回滚块,而是维护了一个全局事务槽[9],在事务修改数据的过程中,将事务槽的地址保存在数据块中. 在事务提交时,将事务版本号保存在事务槽中,然后采用延迟的方式将事务版本号写回数据块.

对于 OceanBase 内存事务引擎来说,所有数据都在内存中存储,不存在写回和加载等数据块的管理成本;并且面向互联网应用,暂时不提供对长事务的支持(目前,OceanBase 的设

计只是针对单条事务的 redolog 不能超过 2M). 因此，并没有按照 Oracle 延迟写回事务版本号的方式，而是在提交时遍历所有被修改的数据，在内存中将事务版本号写回.

(2) 多版本数据的存储

如图 6 所示，在 Memtable 中，一次事务对一行数据的修改被保存为一个链表节点(称为数据块)，并在这个节点上标记事务版本号；而每行的行头结构 RowValue 中保存了三个重要的指针，分别是 undo list head，data list head 和 data list tail.

data list head 和 data list tail 指向最近一次执行 RowCompaction 之后的数据块链表，如图 6 所示，事务版本号为 1-5 的数据块经过三次合并(1-2 合并为 2，2-4 合并为 4，4-5 合并为 5)后保存在 TransID 为 5 的数据块中，后续写入的 TransID 为 6 和 7 的数据块串联在链表尾部.

undo list head 指向的链表中，每个节点保存了每次执行 RowCompaction 之前的数据块链表，如 TransID 为 1 和 2 的数据块，被合并时构造了一个 undo list node 指向合并前的数据块链表(即 1→2)，然后将 data list head 和 data list tail 指向了合并后的数据块链表；同理 TransID 为 2，3，4 的数据块，被合并时构造了一个 undo list node 指向合并前的块链表(2→3→4).

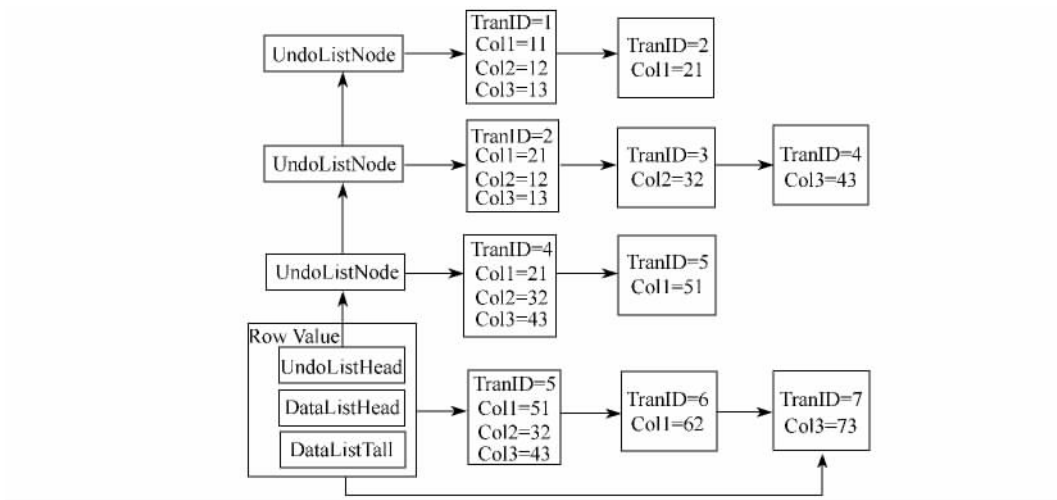

图 6　多版本数据的内存格式

Fig. 6　Memory layout of multiple version

执行快照读时，如果事务需要读取 TransID 为 5 或之后的快照，则从 data list head 开始遍历；而如果事务要读取 5 之前的快照，因为已经做过了 RowCompaction，所以需要遍历 UndoList，比如要读取的快照点为 4，则遍历到第 1 个 UndoListNode 即可找到 TransID 为 4 的数据块，再比如要读取的快照点为 1，则需要继续遍历 UndoList，直到通过第 3 个 UndoListNode 找到 TransID 为 1 的数据块.

(3) 事务提交与回滚

为了实现 read committed 级别的事务，内存引擎需要实现满足 ACID 特性的事务，主要包括如下几个关键特性：事务原子性提交，事务级别回滚，语句级别回滚，读取事务内未提交数据. 为每个事务维护一个被称为事务上下文的结构，用于保存事务执行过程中的未提交数

据、锁信息等. OceanBase 为每条语句产生一个执行计划,这个执行计划结构中包括了必要的基线数据、操作类型(insert/update/delete/replace/select for update)、待写入的数据、判断条件等. 内存引擎执行这个计划的过程主要包括:在当前活跃表中查询必要的动态数据的行头(即上一节提到的 RowValue 结构)并加锁,如果行不存在,则写入一个空的行头,用于加行锁;将一行待写入的多列数据以紧缩格式保存在上一节提到的数据块链表节点(称为 TransInfoNode)中. 如图 7 所示,每行的行头指针与对应的 TransInfoNode 被保存在一个称为 UCInfoNode 结构中(即 uncommited info node),事务内多行的 UCInfoNode 串联在一起保存在事务上下文中. 事务提交时,遍历事务上下文中每行的 UCInfoNode 找到 TransInfoNode,将事务版本号回填.

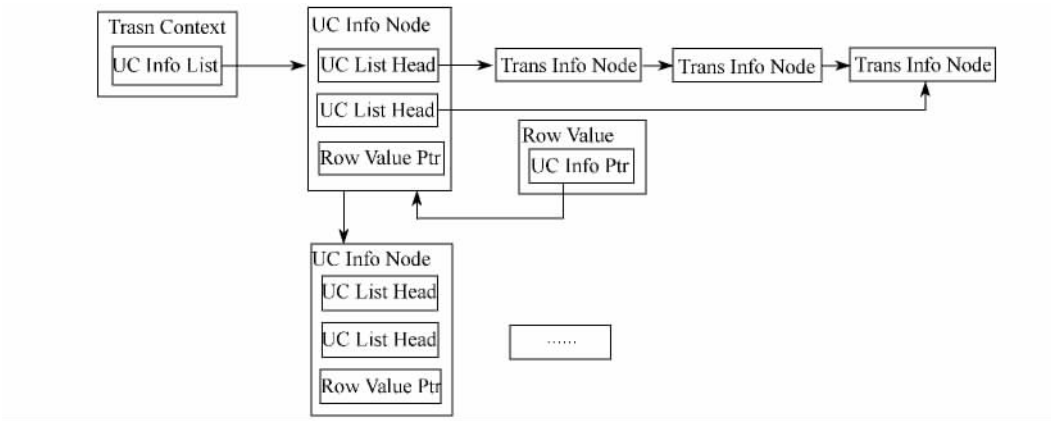

图 7 事务上下文中缓存的未提交数据

Fig. 7 Uncommited data in transaction context

● 事务提交,全局维护了一个当前已提交的最大事务版本号(published_trans_id),事务开启时获取当前的 published_trans_id 作为当前快照点. 在事务提交时,先获取事务版本号,完成提交操作后,将本事务版本你好原子性的更新到 published_trans_id. 而这里需要注意的是多个事务之间更新 published_trans_id 的顺序要保证与获取事务版本号的顺序一致.

● 事务回滚,将事务上下文结构连同保存在它里面的 UCInfoNode 一起释放即完成回滚.

● 语句级别的回滚,事务内单条语句的执行失败需要保证事务状态回滚到语句开始之前,因此在事务上下文中需要在语句开始时记录 UCInfoNode 链表的状态,在语句回滚时将 UCInfoNode 链表的状态回滚到语句开始之前.

● 读取事务内的未提交数据,事务内修改的数据可能会被后面执行的语句读取到,因此在事务执行过程中将上面提到的 TransInfoNode 提前接到每行 data list tail 的后面,但是不修改 data list tail 指针,而事务内的读取逻辑遍历 data list 时,并不按照事务开始时的快照点读取,而是遍历完整的链表,因为并未修改 data list tail 指针,因此事务回滚时也不需要额外的工作.

● 事务内多次修改同一行,单条语句对一行的修改可以保存为一个 TransInfoNode,而事务内多条语句对一行的修改则对应了多个 TransInfoNode,而为了处理上述"读事务内未提交数据"的需求,需要保证一行内多个未提交的 TransInfoNode 能够串联到一起,因此在

行头结构中保存指向这一行 UCInfoNode 的指针，而在 UCInfoNode 中保存多次修改的 TransInfoNode 串联成的链表.

2.3 两阶段锁并发控制与冲突调度

内存事务引擎使用经典的两阶段锁方式来处理并发的更新事务（比如 insert，update，delete 等）和 select for update 的事务，即在参与并发的所有事务中加锁与解锁是相互不重叠的两个阶段. 事务中的 DML 语句在执行过程中，涉及到被读取或修改的行都将被加上互斥行锁，语句执行结束后并不释放锁，而是要在事务结束时才能释放锁. 考虑到范围锁或谓词锁实现较复杂，以及互联网应用的特点，OceanBase 目前只提供 read committed 级别的事务，因此我们只实现了行级别的互斥锁. 每行的行头使用 4 byte 保存锁标志，其中最高位标记锁状态，其余的 31 位用于保存持有当前行锁的事务的描述符（即锁的拥有者）.

- 锁冲突调度

与传统的数据库实现不同，OceanBase 内存事务引擎执行事务操作都是在内存中进行的，没有 I/O 操作，所以启动的线程数量一般不会超过 CPU 核的数量. 线程资源宝贵，因此不能出现线程阻塞，来等待某些事件发生或条件满足的情况. 事务执行过程中可能会遇到加行锁冲突，为了避免线程陷入锁等待，将尝试加锁失败的请求回滚，并放回任务队列等待重新执行，线程则继续处理后面的请求. 加锁失败时，可以获取到持有当前行锁的事务描述符，进而可以得到正在处理这个事务的线程 ID. 对于 autocommit 类型的事务，将加锁失败的任务放回这个线程自己的任务队列排队，待当前持有锁的事务执行完成后，从队列中取出重试任务，从而避免在锁未释放的情况下其他线程的重试开销. 而对于 begin…commit 类型的事务，尽管事务持有锁的时间由客户端决定，但是将任务放回队列的设计，从而保证了系统在繁忙时能够正常处理其他没有锁冲突的请求，同时有机会重试加锁失败的请求.

3 事务持久化、恢复

3.1 批处理与提交

事务提交时，需要通过日志系统将事务的操作日志记录在硬盘上，在保证事务的操作已经持久化后，事务才算真正地完成. 操作日志的记录顺序需要与事务完成的顺序一致，如果每完成一个事务就记录一次操作日志，而记录的日志需要保证刷到硬盘上，显然，这种记录日志的方式在性能方面是无法接受的. 所以，每次都会将批量连续的一批事务日志，作为一次批处理单元，从而一次将这多个事务的操作日志刷到硬盘上. 常见的数据库是通过一个统一的日志缓冲区，加上一个定期工作的刷日志线程来完成这一功能. OceanBase 没有采用这种方案，而是采用了一种独特的解决方案，下面将详细描述这一方法.

内存事务引擎中有一系列负责执行事务的线程 Trans Executor. 在显示（用户手动提交）和隐式（系统自动提交）两种情况下，事务线程都会进行写日志的操作，其中自动提交类型的事务，在语句执行结束的时候会进行写日志操作；而用户显式执行 Commit 操作的事务，会在 Commit 执行完成后进行写日志操作. 在 OceanBase 系统中，内存事务引擎有一个日志缓冲区，见图 8 所示的 Log Buffer. 日志缓冲区由若干个缓冲区块（Buffer Block）组成，每个缓冲区块是 2MB 大小. 目前 OceanBase 系统不支持单个事务超过 2MB 的情况，所以一个事务的所有操作日志是可以在一个缓冲区块中存储的.

事务线程执行写日志操作有 3 个步骤：首先，要在日志缓冲区占位；然后，将事务线程准

备好的事务操作日志数据写到日志缓冲区中;最后,由缓冲区块最后一个完成写日志操作的事务线程负责将缓冲区块发起写硬盘和同步备机的操作.每个事务线程在缓冲区中填充完日志后,会判断当前的负载情况,如果负载较低,则不管缓冲区是否已满,而立即将缓冲区提交,以降低事务延迟.

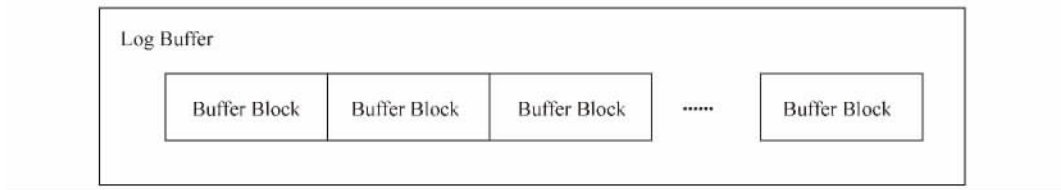

图 8 日志缓冲区

Fig. 8 Log buffer

在日志缓冲区中的占位操作,通过一个 128 位组合起来的数字来确定:

```
struct
{
  int64_t block_id;
  int32_t offset;
  int32_t id;
}
```

block_id 表示操作日志所在的缓冲区块的编号,block_id 严格递增,缓冲区块被循环使用.offset 表示一次事务的操作日志在缓冲区块内部的偏移量,id 表示操作日志在缓冲区块内部的序号.使用这样的数据结构,事务线程在日志缓冲区中的占位操作就可以用一次操作系统的 CAS 指令完成.在 CAS 操作之前,先判断当前正在使用的缓冲区块还够不够存储要写入的日志,如果够,则直接修改 offset 和 id;如果不够,那么使用下一个缓冲区块,block_id 递增 1,offset 重置为 0,id 重置为 1.然后用修改完的 3 元组原子替换之前的 3 元组,如果替换成功,则占位成功,如果原子替换没成功,则重复之前的流程.

内存事务引擎在执行事务时,会记录事务中操作的数据,记录的信息是以内存数据结构暂存在事务上下文中.当事务线程完成占位后,则将事务上下文中的事务信息以序列化的格式写入缓冲区块.

每个缓冲区块会记录块内一共有多少个事务的日志,当事务线程完成写日志后,会在缓冲区块中计数,缓冲区块中最后一个完成的事务,需要负责将这个缓冲区块持久化.持久化的工作就是将操作日志写到硬盘并且同步备机.这个操作不会占用事务线程很多时间,事务线程仅进行发起工作,将缓冲区块交给写盘的线程,并且发起异步的网络请求即可.后续确认日志数据是否写完,是在写盘和网络操作的回调函数中处理.

3.2 并发事务回放

数据库的主机以并行方式执行多个事务,数据库的备机通过回放操作日志来追赶主机的数据状态,经常会面临性能问题.为了保证事务完成的顺序性,回放很难做成并行的方式.OceanBase 使用新的模式结合多版本并发控制实现了备机的并发日志回放,保证即使主机在以最大并发工作时,备机也可以追赶上主机.

如图 9 所示，并发回放模块由一个读日志线程和若干个回放线程组成，读日志线程从操作日志中读取主机同步到备机的操作日志，并且解析成单独的事务，然后将每个事务打包成回放任务，交给回放线程. 所有的回放线程以消费者工作模式，只要拿到回放任务就进行回放操作，将操作日志中记录的信息写到内存表中.

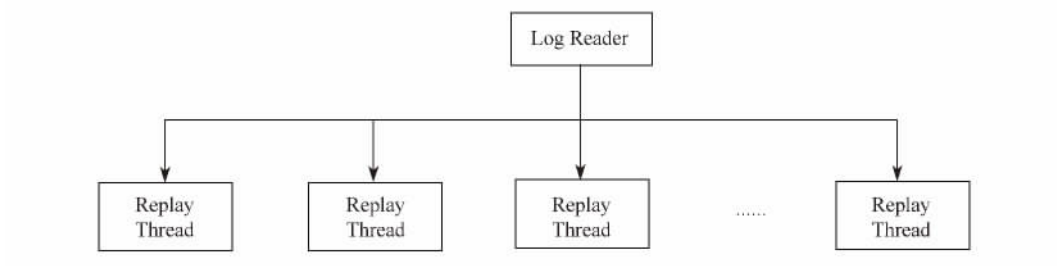

图 9　日志回放线程

Fig. 9　Log replay thread

回放线程操作 Memtable 时，需要给所要回放的行加写锁，但是与主机进行事务操作不同的是，回放线程不加两阶段锁. 只在操作到具体的行的时候，在操作之前加上锁，操作完成后，就解开了行锁. 这种方式让两个可能操作同一行的事务可以并行回放.

内存事务引擎使用多版本并发控制，事务回放的顺序可能不同于主机操作事务的顺序，但是操作过程中，行内插入数据时，保证链表内的数据是按照事务的版本有序的，即保证了备机和主机是同样的 Memtable.

4　性能优化

本章介绍 OceanBase 内存事务引擎互联网实际应用环境下几个通用的性能优化方案，包括针对并发热点行事务的优化和并发读写锁和引用计数的优化. 事务两阶段锁，在事务执行过程中加锁，事务回滚或提交 redolog 成功后才释放锁，对于多个并发事务修改不同行的情况不存在冲突问题，这些并发执行的事务提交的 redolog 有机会组成一个 group，一次 I/O 就写到磁盘. 但是对于并发修改同一行的事务，比如电子商务平台的秒杀活动，几十万人同时在线对同一件商品下单，对于服务器来说就是大量并发事务同时执行判断并扣减商品库存.

在两阶段锁的设计中，因为事务在提交 redolog 成功后才释放行锁，在行锁上冲突的其他事务才能继续执行，因此对于同一行的并发事务，只能够与 I/O 串行化执行，对热点行事务的提交能力退化成磁盘的 IOPS 能力. 因此对于热点行的并发事务，我们对内存事务引擎进行了两点优化.

● 提前释放锁

对于 autocommit 事务，在事务预处理阶段执行成功，在日志缓冲区占位后，就可以立即释放事务执行过程加的行锁，使后续对相同行的修改事务可以获取到锁开始预处理. 这样使得修改相同行的多个并发事务虽然预处理阶段仍然是串行化的，但是耗时最多的写 redolog 阶段有机会组成一个组一次提交，增加引擎的整体吞吐能力[10].

尽管释放了行锁，因为事务还处于未提交的状态，所以没有提交 redolog 成功前，是不

会给客户端应答成功的,后续对读取相同行的只读事务也不会读取到未提交的数据;而后续对相同行的修改事务则需要读到未提交数据,以保证在前一个事务修改的基础之上进行操作,比如每次判断并扣减库存要在上一次扣减结果的基础上进行判断和扣减.

对于 redolog 提交失败的处理,当前事务要被回滚. 对于只读事务,因为不能读取到未提交的数据,因此不受影响;而对于修改了相同行的事务,也要一起回滚,因为提交 redolog 是顺序执行的,当前事务提交 redolog 失败,意味着它后续的事务尚未提交,所以回滚是安全的;而对于 select…for update 语句,对目标行加锁成功后还需要等待提前释放锁的事务提交之后才能继续读取,否则可能出现读到的未提交数据因为提交 redolog 失败而被回滚的风险.

● 热点行请求调度

尽管提前释放行锁解决了并发事务提交 redolog 的冲突,但是事务预处理操作仍然需要在加锁成功后才能执行,在并发压力大的情况下,处理锁冲突调度的代价仍然比较大. 因此考虑到事务预处理既然要通过加锁达到串行化处理的目标,那么可以直接将修改相同行的事务请求调度到相同的线程排队处理,避免一次锁冲突调度的开销.

4.1 并发读写锁与引用计数

在 OceanBase 内存事务引擎中,经常会使用到一些全局对象,比如 Schema 管理器、Memtable 管理器等、Schema 变更或活跃表冻结等管理变更操作时会修改这些对象,而处理一般的事务请求时则不会修改这些对象,因此使用读写锁或引用计数保护这些全局对象,处理事务请求时加读锁,执行管理变更操作时加写锁. OceanBase 使用 CAS 原子操作实现读写锁和引用计数,在 CPU 核心数量不多时性能较好,但是随着 CPU 核的数量的增加原子操作对全局共享变量的读写代价变得明显.

因此针对全局对象读多写少的特性,OceanBase 实现了并发读写锁和引用计数,通过减小读锁代价,增加写锁代价来优化性能. 即读锁的加锁的解锁,引用计数的增加和减少都只操作线程局部变量;要获取写锁时,则需要遍历所有线程,对每个线程局部的读写锁都加上写锁;引用计数同理,要尝试释放对象时,遍历所有线程局部的引用计数,确认都没有引用的情况下才释放对象,否则在每个线程局部打上标记,等减引用计数时再尝试释放对象.

5 总 结

OceanBase 内存事务引擎是在动态数据静态数据分离存储架构下的产物,其在内存格式、并发控制、持久化和性能优化等方面,参考了经典数据库理论中成熟的方法. 并在此基础上充分发挥 OceanBase 架构特点,并结合当今电子商务的应用特性,为 OceanBase 高性能事务处理奠定了基础. 未来,OceanBase 将持续优化内存事务引擎性能,实现分布式 Updateserver,在保证事务 ACID 特性的基础上,增强可扩展性和容错能力.

[参 考 文 献]

[1] BERENSON H, BERNSTEIN P, GRAY J, et al. A critique of ANSI SQL isolation levels[C]//ACM SIGMOD Record. ACM, 1995, 24(2): 1-10.

[2] MOHAN C, HADERLE D, LINDSAY B, et al. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging[J]. ACM Transactions on Database Systems (TODS), 1992, 17(1): 94-162.

[3] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.
[4] RODEH O. B-trees, shadowing, and clones[J]. ACM Transactions on Storage (TOS), 2008, 3(4): 2.
[5] MICHAEL M M. Hazard pointers: Safe memory reclamation for lock-free objects[J]. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(6): 491-504.
[6] Oracle. Data Concurrency and Consistency[EB/OL]. [2014-07-31]. http://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm#i17841.
[7] Oracle. Transaction Set Consistency [EB/OL]. [2014-07-31]. http://docs.oracle.com/cd/B28359_01/server.111/b28318/consist.htm#CNCPT1322.
[8] **何登成**. InnoDB **事务/锁/多版本 实现分析**[EB/OL]. [2014-07-31]. http://hedengcheng.com/?p=286.
[9] LEWIS J. Transactions and Consistency[M]//Oracle Core. New York: Apress, 2011: 25-58.
[10] JOHNSON R, PANDIS I, STOICA R, et al. Aether: a scalable approach to logging[J]. Proceedings of the VLDB Endowment, 2010, 3(1): 681-692.

(责任编辑　赵　伟)

(**上接第** 148 **页**)

[参　考　文　献]

[1] **天猫微博**[EB/OL]. http://weibo.com/1768198384/AiigJrzYT?mod=weibotime.
[2] **支付宝微博**[EB/OL]. http://weibo.com/1627897870/AiiuiseVH?mod=weibotime.
[3] OceanBase **开源**[EB/OL]. http://alibaba.github.io/oceanbase/.
[4] **天猫微博**. http://weibo.com/1768198384/Aie2CyONt?mod=weibotime#_rnd1404271771131.
[5] **阿里巴巴招股书**[EB/OL]. 2014-06-17. http://www.sec.gov/Archives/edgar/data/1577552/000119312514236860/d709111df1a.htm.
[6] Angry Birds Racks Up 8 Million Downloads in One Day[EB/OL]. http://www.forbes.com/sites/davidthier/2013/01/04/angry-birds-racks-up-8-million-downloads-in-one-day/.
[7] LAMPORT, L. The part-time parliament[J]. ACM TOCS, 1998, 16(2): 133-169.
[8] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[J]. OSDI, 2006: 205-218.
[9] GHEMAWAT S, GOBIOFF H, LEUNG, et al. The Google file system[R]. ACM SOSP, 2003: 29-43.
[10] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally-distributed database[C]. OSDI, 2012: 251-264.

(责任编辑　李　艺)