文章编号: 1000-5641(2018)05-0107-13

新型 OLTP 系统的技术与实践

贺小龙[1], 马海欣[2], 何毓锟[2], 庞天泽[2], 赵 琼[2]

(1. 华东师范大学 数据科学与工程学院, 上海 200062;
2. 交通银行 数据中心, 上海 200062)

摘要: 自 20 世纪 70 年代以来, 硬件已经得到了巨大的发展, 高性能服务器大多数配备 TB 级的容量、数十个物理核. 然而, 传统的事务型系统仍旧是基于磁盘存储, 运行在物理核数较少的硬件环境上, 无法有效地、充分地利用新硬件的运算能力. 另一方面, 随着互联网的发展, 应用对事务型系统的性能有了更高的要求. 部分应用在极端情况下需要服务百万甚至千万的并发访问, 然而传统的数据库系统并不能支撑这些高并发、高吞吐率的应用. 因此, 在高性能硬件上重新设计与实现事务型数据库系统已成为重要的研究热点. 本文将重点介绍在大内存、多核环境下, 事务型数据库系统在各个方面最新的研究工作, 并结合开源数据库系统 OceanBase, 综合介绍新型 OLTP (on-line analytical processing) 系统的设计.

关键词: 事务处理; 并发控制; 日志与容错; 多核扩展性
中图分类号: TP392 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2018.05.009

Technology and implementation of a new OLTP system

HE Xiao-long[1], MA Hai-xin[2], HE Yu-kun[2],
PANG Tian-ze[2], ZHAO Qiong[2]

(1. *School of Data Science and Engineering, East China Normal University, Shanghai* 200062, *China*;
2. *Data center, Bank of Communications, Shanghai* 200062, *China*)

Abstract: Since the 1970s, there has been considerable progress in hardware development; in particular, high-performance servers are now equipped with TB-level memory capacity and dozens of physical cores. Traditional OLTP systems, however, are still based on disk storage and designed for hardware with a small number of physical cores; hence, these systems are unable to effectively and fully exploit the computing power offered by new hardware. With the development of the Internet, applications commonly have high performance requirements for transactional systems. In extreme cases, some applications service millions of concurrent access requests, which traditional database systems cannot satisfy. Hence, the redesign and implementation of a transactional database system on high performance hardware has become an important research topic. In this study, we

收稿日期: 2018-07-10
基金项目: 国家 863 计划项目 (2015AA015307)
第一作者: 贺小龙, 男, 博士研究生, 研究方向为内存数据库事务处理. E-mail: hxlongecnu@163.com.

focused on recent work on transaction database systems on large memory and multi-core environments. We used OceanBase, an open source database developed by Alibaba, as an example to analyze the design of a new OLTP system.

Keywords: transaction processing; concurrency control; log and recovery; multi-core scalability

0 引 言

传统的联机事务型系统[1]通常包括以下组件: 磁盘上的B树索引[2], 两阶段锁并发控制[3-4], ARIES日志[5-8]以及多线程的支持. 这样的存储和执行架构是针对20世纪70年代的计算机硬件条件优化的. 当时的计算机具有较小的内存, 数据存储主要依赖于磁盘, CPU的指令执行速度和低速设备的访问速度(例如读写磁盘)相差若干个数量级. 当程序访问低速设备时, CPU处于阻塞态, 因此使用多线程技术并发执行多个程序来提高CPU的利用率. 然而硬件技术在摩尔定律的支配下有了巨大的发展. 一方面, 内存的容量不断更新迭代, 其容量的增大和价格的变化使得许多的OLTP数据库以内存为主要存储介质; 另一方面, 处理器拥有的物理核数在不断地增多, 物理核数的增多允许更多的物理线程并行执行, 这同时也对系统的多核扩展性有了更高的要求. 然而, 基于磁盘存储设计的系统架构并不能充分发挥大内存、多物理核的处理性能. 如何在新的硬件条件下, 设计高性能的OLTP系统是当前的一大研究热点. 本文将主要介绍在大内存、多物理核环境下的OLTP系统设计的技术发展和国内外的主要研究方向与研究进展. 此外, 我们将以OceanBase[9]为例, 综合介绍现有技术在工业界的应用情况.

相对于磁盘存储, 我们需要考虑到内存的易失性以及并发控制技术的设计目标已发生根本性改变, 由此带来了数据的恢复、数据持久化以及并发控制问题. 在内存数据库中, 我们处理事务时不再将磁盘I/O(Input/Output)的阻塞作为第一影响要素, 当前的并发控制技术更加侧重于降低并发控制技术的维护代价, 并且提升其多核扩展性. 在提高多核扩展性同时, 系统的并发度在提高, 会出现临界区冲突的增大、并发控制开销增大等问题. 为了充分利用这些新硬件的特性, 我们需要在并发控制上做出有针对性的大量优化以及保证数据可靠性和持久化. 本文将首先介绍当前主流的内存事务系统, 然后重点介绍并发控制相关的最新技术, 在第三章中主要介绍容错与恢复相关的技术, 在第四章中主要介绍内存型事务系统如何进行数据持久化, 最后总结全文.

1 主流系统及其特点

当前主要的内存型数据库系统有: VoltDB(http://www.voltdb.com), Hekaton[10], MemSQL(https://github.com/memsql), OceanBase等. 这些系统设计服务于那些并发数、吞吐率要求特别高的应用, 如VoltDB能够大幅降低服务器资源开销, 单节点每秒数据处理远远高于其他数据库管理系统. 下面主要介绍这几种数据库.

VoltDB是多所高校联合开发的原型数据库H-Store的商业版本. 它是无共享的多核分布式内存数据库系统, 其产生源自Harizopoulos等[11]针对传统数据库的一次分析. 分析结果显示, 传统架构的系统约有30%的时间花费在了缓冲区管理, 30%的时间花费在了Lock和Latch的维护, 30%的时间花费在了日志上. 对此, VoltDB通过使用内存作为主要的存储介质去除了缓冲管理的开销. 它通过将数据进行分区, 每个分区上的事务请求由单一

线程顺序调度, 以此避免了使用Lock和Latch进行并发控制; 最后通过逻辑日志Command Logging[12]去除WAL日志.

Hekaton是微软研发的单机内存事务引擎. 它主要具有以下特点: ①优化了内存索引; ②核心共享结构, 例如内存池、内存索引以及事务表采用了无Latch和Lock的设计方式, 并且使用了乐观的多版本并发控制技术, 从而消除了锁和锁表; ③编译事务逻辑, 所有的SQL操作, 以及存储过程逻辑都被编译为机器指令, 大大减少了解析执行逻辑和识别数据类型等的指令开销. Hekaton不同于VoltDB, 它采用了无分区和共享存储的策略. 这主要是因为Hekaton服务的应用数据和负载是不可切分的.

OceanBase是阿里巴巴公司研发的分布式可扩展的关系型数据库. 它使用两层LSM-Tree来管理增量和基线数据, 其单节点的内存事务引擎和可扩展的存储节点分别对应LSM-Tree的内存端和磁盘端. CBASE是交通银行基于开源的OceanBase 0.4.2版本开发的分布式数据库, 与Oceanbase0.4版本相比, CBASE采用了Paxos协议实现主备同步日志, 完成了多点更新的内存事务引擎以及针对特定业务场景的查询优化器的设计.

MemSQL是由Eric Frenkiel(前Facebook员工)和Nikita Shamgunov(前微软SQL Server高级工程师)设计的一款基于内存的分布式关系数据库. 它通过将数据存储在内存中, 并将SQL语句预编译为C++而获得极高的执行效率.

为了服务于这些对性能要求特别高的应用, 系统不仅需要有高性能的硬件支持, 同时需要对传统的系统架构和算法实现进行大量的优化. 在下文中, 将重点介绍当前最新的研究工作是如何针对数据库的并发控制、日志容错与恢复等模块进行优化的.

2　并发控制技术

传统的数据库, 例如MySQL(https://github.com/go-sql-driver/mysql), Oracle(https://www.oracle.com)和PostgreSQL(https://www.postgresql.com)均采用了两阶段锁来管理读写事务的并发控制, 采用多版本并发控制技术管理只读事务. 两阶段锁保证了读写事务的隔离性, 多版本使得只读事务可以无锁执行, 提高了并发度. 由于传统的并发控制技术都是基于磁盘存储进行设计的, 以及物理核数的增加, 当前的并发控制技术更加侧重于降低并发控制技术的维护代价, 并且提升其多核扩展性. 本章首先分析传统并发控制技术的特点以及当前并发控制技术发展的方向, 然后详细地介绍当前并发控制方面的研究成果及其主要特点.

2.1　并发控制技术发展现状分析

传统的并发控制技术是为基于磁盘存储的系统设计的, 而这类系统最主要的性能瓶颈是磁盘I/O阻塞造成的. 由于磁盘I/O的影响, 系统的行为具有以下特点: 执行事务会因为磁盘I/O频繁阻塞, 单个事务的执行延迟较长, 系统整体的CPU使用率降低. 为此, 传统的并发控制技术尝试同时并发控制更多的事务, 以提升系统的吞吐率并且充分利用CPU. 这就要求设计复杂的并发控制策略, 同时由于并发事务数量较多, 需要设计良好的死锁检测机制.

对于内存数据库而言, 磁盘I/O阻塞不会再是系统的主要性能瓶颈. 因此, 并发控制技术的设计目标已经有了根本性的改变, 系统并不需要并发大量的事务以保证充分利用CPU. 同时, 复杂的并发控制策略具有高昂的维护代价, 并且存在多核扩展性差的缺点. 这些因素限制了系统性能的提升. 因此, 当前的并发控制技术更加侧重于降低并发控制技术的维护代价, 并且提升其多核扩展性.

2.2 传统的并发控制实现

传统的并发控制实现主要有两阶段锁、基于时间戳的并发控制以及乐观并发控制. 下面将着重介绍两阶段锁的主要实现. 两阶段锁要求事务在执行过程中分为加锁阶段和解锁阶段, 并且两个阶段不相交. 数据库系统设计了集中式的锁管理器, 对外提供加锁与解锁的接口. 锁管理器使用哈希表管理锁请求, 如图1所示.

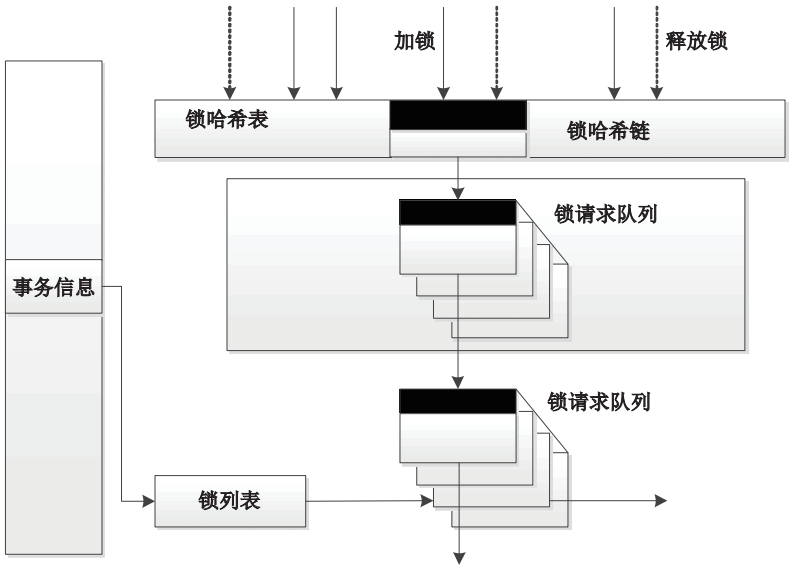

图1 锁管理器

Fig. 1 Lock manager

一次加锁操作需要经过以下基本流程:

(1) 计算数据的哈希值. 查找锁表上对应的桶.

(2) 访问桶, 进入哈希桶的临界区.

(3) 在哈希链上查找数据锁, 如果不存在则新增一个锁节点; 获得数据锁的请求队列.

(4) 进入数据锁请求队列的临界区.

(5) 离开哈希桶临界区.

(6) 将加锁请求添加到队列尾部.

(7) 判断当前加锁请求是否与队列中其他请求冲突, 并为当前请求设置相应的状态.

(8) 离开数据锁请求队列临界区.

事务加锁成功后继续执行, 失败后进入等待, 直到锁释放后唤醒; 在事务提交后, 释放所有的数据锁, 并唤醒等待锁的事务. 由于发生取锁冲突后, 事务会保留当前持有的锁, 并进入等待状态. 当两个事务均需要对方的锁, 而进入等待状态时, 会陷入死锁状态. 为此, 需要设计死锁检测机制. 两阶段锁保证了不同事务之间的隔离执行. 它要求每个事务在执行过程都需要获取相应的读写锁. 事实上, 一个应用中通常包含了一定比例的只读事务. 当采用多版本并发控制策略时, 只读事务会访问固定版本的数据, 而读写事务会访问最新版本的数据. 通过版本隔离, 只读事务在执行过程中不需要加锁, 故不会阻塞读写事务的执行. 这可以大大提高事务的并发度. 因此, 诸如MySQL, Oracle和PostgreSQL等都采用了多版本并发控制技术来隔离只读事务, 采用了两阶段锁来隔离读写事务.

在这一小节中, 重点介绍了传统数据库中常用的并发控制协议及其实现方式. 可以看到, 两阶段锁协议以及多版本并发控制确保了系统能同时并发大量的事务. 这样的设计使用了集中式的共享数据结构. 一方面, 复杂的设计耗费了大量的CPU时间; 另一方面, 集中式的结构在多核环境下会出现剧烈的资源冲突限制系统的多核扩展性. Harizopoulos等[11]分析指出, Lock&Latch在单核环境下消耗了33%的CPU时间. Johnson和Tu等[13-14]分析指出, 多核环境下访问锁表造成的资源冲突浪费了大量的CPU时间. 随着硬件的不断发展, 物理核数增多, 内存容量变大, 这样的设计已然不适用. 下面, 将介绍近些年来, 对多核、大内存下的事务引擎优化工作, 并介绍OceanBase的设计思路.

2.3 最新工作

在多核、大内存的环境下, 当前对并发控制技术的优化访问主要围绕以下几点: ①减少共享结构的使用, 降低对共享结构访问产生的冲突, 提高系统的多核可扩展性; ②优化并发控制协议, 降低并发控制协议的开销.

2.3.1 减少共享结构的冲突访问

访问共享结构需要使用临界区保护, 随着物理核数的增加, 系统对临界区的并行访问产生的Latch Contention也随之增加. 两阶段锁的实现依赖于全局共享的锁管理器, 访问锁表需要使用临界区. 为了避免随着物理核数增加而导致临界区冲突的增大, 主要有以下几方面的优化.

优化集中式锁表. Gray等[1]针对MySQL分析发现: 访问锁管理器产生的Latch Contention在高并发线程数下会成为瓶颈. 多线程访问锁管理器时, 获得Latch造成的等待耗费了大量的CPU时间. 物理核数的增多加剧了对锁管理器的冲突访问. 针对这样的问题, 文中设计了Lock-Free的锁管理器降低多线程访问锁表时产生的冲突时间, 其主要特点是: ①避免使用互斥量, 自旋锁保护一段代码的多线程访问; ②使用CAS和Memory Barrier维护共享数据的原子修改与操作顺序. 此外, 文中设计了成组释放锁的策略, 减少线程修改锁管理器的次数. 该方案通过优化锁表的访问方式降低了并发控制的维护开销, 并提升了多核扩展性.

减少集中式锁表的访问. Johnson等[13]研究发现, 在多数应用中对集中式锁表的大量访问旨在获得一些热点页面的共享锁. 减少锁的请求数量可以有效降低对锁表的冲突访问. 为此, 作者提出了一种锁继承的方式. 在一个线程中, 当上一个事务提交后, 它可以不把锁返回给锁管理器, 并且传递给下一个事务. 如果下一个事务能够重用这些锁, 那么就成功减少了锁的释放与申请代价, 降低了对锁管理器的争夺. 该方案通过锁在多个事务间继承锁, 减少了对集中式锁表的访问, 从而降低了并发控制的维护开销.

构建线程局部的锁表. Tu等[14]发现访问集中式锁表产生的资源冲突成为了多核环境下数据库系统的瓶颈. 这项工作可通过对数据进行分区, 每个分区上数据操作由单一的线程操作. 因此, 集中式的锁表由多个线程局部的锁表代替, 其代价是需要一个事务会在多个线程上切换执行. 系统需要根据分区策略设计相应的事务切分算法, 将一个完整的事务切分为若干个子任务, 并分发到不同的线程上执行.

简化锁与锁表. 集中式的两阶段锁管理不仅存在多核扩展性的问题, 同时耗费了大量的CPU时间. 因此, 当前的许多系统都摒弃了集中式的锁管理器. Tu和Blanas等[14-15]提出了一种轻粒度的锁, 具有较小的执行开销. 首先, 文中提出将锁信息作为数据的隐藏列存储在数据头中. 其次, 使用定长字段表示锁信息, 而不是使用请求队列. 当加锁或者解锁是修改锁字段. 该方案具有以下优势: ①避免使用和访问全局共享的锁管理器, 具有更好的多核扩

展性; ②简化了锁的表示和加锁解锁的过程, 从而减少了使用锁的CPU开销; ③数据与锁联合存储可以提高操作在加锁和访问数据时的缓存命中率. 然而, 对锁的简化也引入一些重要问题. 由于不再维护每个锁的请求链表, 当一个事务释放锁后, 它无法唤醒下一个等待事务. Tu 和 Blanas 等[14-15]采用了不同的方案解决阻塞事务唤醒的问题, 本文将在并发控制协议的优化中介绍这些工作.

优化版本号的分配. 在多版本并发控制中, 需要维护事务修改的时序关系, 这通常是通过维护一个集中式的版本号来实现的. 任意事务在提交修改前, 需要获得一个版本号. 这个版本号反应了所有事务修改发生的先后次序关系, 由此确定了整个数据库版本变化的顺序. Tu 等[14]在对事务引擎进行了大量优化的情况下, 版本号的集中式分配成为了多核扩展的瓶颈. 针对这个问题, 作者提出成组分配事务号的方案. 在系统执行过程中, 以 25 ms 作为一个阶段, 该阶段内提交的事务共享一个集中分配的全局版本号. 阶段内部的事务由执行线程分配一个本地的版本号. 全局版本号确定了不同阶段事务执行的时序. 局部版本号无法确定所有在同一阶段提交的事务的先后时序, 它仅能确定修改同一个数据项的事务或者由相同工作线程调度的事务的时序. 该方案的优点在于减少对全局版本号的同步写冲突, 而缺点则是弱化了事务执行的时序关系, 只能提供阶段结束后的全局状态的一致读, 对于一个阶段内部, 无法区分事务的先后顺序, 无法对阶段内部的全局状态提供全局一致读.

以上综述了当前最新工作对并发控制协议实现的优化, 在表 1 中总结了这些方面在提升多核扩展性以及降低维护开销方面的效果.

表 1 不同并发控制协议优化性能对比

Tab. 1 Comparison of optimization performance of different concurrency control protocols

优化手段	提升多核扩展性	降低维护开销
优化集中式锁表	√	
减少集中式锁表访问		√
构建线程局部的锁表		√
简化锁与锁表	√	√
优化版本号的分配	√	

2.3.2 优化并发控制

并发控制协议隔离多个事务对数据的冲突访问. 传统的并发控制协议主要有两阶段锁[16]、乐观并发控制协议[17], 基于时间戳的并发控制协议[14]以及多版本并发控制[16,18]. 在新硬件条件下, Agrawal 等[3]提出了确定性执行的并发控制协议, Ren 等[19]对传统的乐观并发控制与悲观并发控制进行了优化. Yu 和 Pandis 等[20-21]使用了新的冲突解决策略与死锁检测技术.

确定性执行策略. 确定性事务执行策略是当前的研究热点, 许多原型系统都采用了这样的策略[13,22]. Harizopoulos 等[11]针对传统数据库的模块开销统计发现, 并发控制协议的维护占据了 21% 的 CPU 时间. 为此, H-Store 和 VoltDB 等系统彻底摒弃了使用并发控制策略, 而将数据进行分区, 每个分区上由单线程顺序执行事务请求. 在每个线程上, 由于事务请求时按严格的顺序执行, 因此不需要进行并发控制. 另外, 由于数据绑定到线程, 故在访问数据时不需要使用 Latch. 并且可以通过调整数据布局, 以避免 NUMA 环境下访问到远程内存. 然而, 该策略下, 访问多分块的事务需要采用代价高昂的分布式提交协议提交[23]. 多分块数据的访问会出现负载倾斜[24]. 此外, 该策略限制了事务的类型. 事务必须采用存储过程描述, 在执行过程中无法与外部交互. 当存储过程中包含较多计算时, 这也会加重数据库服务器的计算负担.

乐观并发控制协议. 乐观并发控制协议[17]采用验证的方式来解决事务之间的读写冲突.

它避免了大量的加锁与解锁及死锁检测等开销. 然而, 当事务冲突较高时, 大量事务会因为验证失效而回滚. 在传统的磁盘数据库中, 事务的延迟通常较高, 导致了更高的事务冲突率. 在内存环境下, Tu 和 Blanas 等[14-15]采用了乐观的并发控制协议, 读操作在提交时验证版本的有效性, 写操作在执行过程中写入最新版本并获得数据锁, 若其他事务尝试写入将导致写写冲突并回滚. 在该协议下, 读写操作都不会造成阻塞, 因此会具有较低的执行延迟. 乐观协议具有较小的维护代价, 然而当事务延迟较高, 读写冲突较高时, 大量事务的回滚会导致 CPU 资源的浪费.

悲观并发控制协议. Blanas 等[15]在内存数据库中设计了悲观的并发控制协议, 使用了轻粒度的锁[19,24]替代了集中式的锁表, 读写操作都会获得数据锁. 发生写写冲突时, 事务将被强制回滚. 发生读写冲突时, 为了避免加锁导致的事务阻塞, 协议允许写操作强行修改已被读锁保护的数据, 并建立事务之间的提交依赖关系, 限制写事务在读事务后提交, 保证正确的串行顺序. 相应地, 协议也允许读操作强行读取未提交的写入数据. 该机制下, 需要建立事务之间的提交依赖关系.

冲突解决策略与死锁检测. 传统的数据库采用了冲突-等待-唤醒的策略解决加锁冲突, 在发生读写冲突或者写写冲突时, 事务会等待锁的释放. 在持锁事务提交释放锁时, 会唤醒等待事务. 为了避免事务之间循环等待数据锁而发生死锁, 需要设计死锁检测机制. Ren 等[19]工作中没有采用冲突-等待-唤醒的策略. 一方面, 这是因为数据锁的实现被简化了, 只维护锁的状态信息, 不维护请求链; 另一方面, 等待策略会造成上下文的切换. 一次上下文切换将需要执行上千次指令, 这将造成 CPU 时间的浪费并增大事务的执行延迟. Ren 等[19] 都采用 NO-WAIT 的策略, 回滚当前加锁失败的事务, 加入任务队列中等待重新调度. NO-WAIT 策略认为在内存环境下重试事务的代价将远小于维护事务等待的代价. 此外, 由于任何事务都不会持锁等待, 所以不会存在死锁, 不需要进行死锁检测. 然而, 当冲突率较高时, 频繁的事务回滚会导致大量的 CPU 资源的浪费. Tu 等[14]对写写冲突采用了 NO-WAIT 的策略. 当发生读写冲突时, 通过允许读取被写锁保护的数据 (修改被读锁保护的数据), 避免了事务阻塞以及上下文切换, 同时建立冲突事务之间的提交依赖, 确保持有锁的事务提交后本次加锁失败的事务才会提交, 这保证了正确的串行化顺序. 事务之间的提交依赖可能导致死锁的发生. 为此, 该工作使用 Tarjan 算法探测事务间的等待图, 判断是否存在强连通分支以确定是否发生了死锁.

2.4　OceanBase 的设计

OceanBase 的事务管理是在全内存环境下进行的, 采用的并发控制技术为两阶段锁与多版本并发控制技术.

为了实现多版本并发控制, 系统采用集中式的版本号管理方式, 维护一个全局递增的版本号, 以及当前发布的最新版本号. 读写事务在预提交时, 向全局版本号申请一个新的版本号, 并将版本号写入更新数据中. 在完成日志写磁盘与同步备机后, 更新当前发布的最新版本号. 只读事务在执行前获得当前发布的最新版本号, 并读取该版本的数据. 多版本并发控制保证了读操作部分需要申请读锁, 增大了系统的并发性.

为了实现两阶段锁, 系统在数据的行头存储数据锁[19]. 由于 OceanBase 实现的隔离级别为 Read Committed, 在事务执行过程中, 写操作在修改数据前会获得行锁, 而读操作根据最新发布版本号读取最新已提交数据. 进入预提交阶段, 在完成版本控制操作并将数据写入内存表后, 释放所有的数据锁, 然后开始写日志.

在事务执行过程中, 当发生加锁冲突时, 加锁冲突失败的 SQL 请求将会回滚重试本次 SQL 请求, 即采用了 NO-WAIT[20]策略解决加锁冲突. 此外, 系统采用超时机制避免一个事务长时间占据锁, 防止死锁的产生.

OceanBase 采用的并发控制技术具有以下优点:

(1) 多版本与两阶段锁混合的并发控制策略有效地隔离了读操作与写操作之间的访问冲突, 提高系统的并发处理能力.

(2) 简化的锁实现降低了维护两阶段锁协议的开销. 此外, 这种实现策略避免了集中式的管理, 有利于提高系统的多核扩展能力.

(3) 无阻塞的冲突解决方式. 由于同时内存数据库在事务执行过程中不会发生 I/O 阻塞, 系统仅需要与物理核数相当的执行线程, 一方面避免了启动过多的线程, 另一面可以有效地减少上下文切换的次数.

以上并发控制策略同样具有一些缺点, 例如在处理 Begin ... Commit 类的长事务时, 若事务之间的访问冲突较高, 由于事务持有的数据锁不会在短时间内释放, 冲突的 SQL 请求将在系统内部不断的重试, 这将导致 CPU 计算资源被占据浪费, 影响系统最终的吞吐率. 因此, OceanBase 的并发控制策略更适用于事务较短、事务之间的冲突较少的情形.

3 日志容错与恢复

当前的大多数数据库系统依赖于写前日志保证数据的容错与恢复[7]. 它主要在数据写入磁盘前写入 Undo 日志或者 Redo 日志. 其中 Undo 日志主要消除写入磁盘中的未提交数据, Redo 日志恢复未写入磁盘的提交数据. 写前日志允许事务在没有把所有数据修改写入磁盘前提交. 这种技术需要一个全局的日志管理器. 为了避免频繁的磁盘写入操作, 数据库通常还实现了组提交策略[25]. 这种策略将综合多个事务的提交数据, 成组写入磁盘. 在内存数据库中, 数据是在内存中存储管理的, 为了实现容错, 需要周期性的将数据写入磁盘, 并且记录事务修改的日志. 我们将首先介绍传统数据库系统的写日志流程, 然后介绍在新硬件环境下对日志维护的优化.

3.1 传统的日志实现

以 PostgreSQL 为例, 介绍它的写日志流程 (见图 2).

(1) 首先执行线程在本地准备好日志.

(2) 执行线程获得日志缓冲区的排它锁, 并向日志缓冲区写入日志. 释放排它锁.

(3) 执行线程休眠一段时间, 等待足够多的事务进行成组提交.

(4) 执行线程唤醒, 查看当前写入磁盘的最大日志号, 判断当前事务的日志是否已写入磁盘. 若写入, 则完成提交.

(5) 若当前事务日志尚未写入磁盘, 申请获得全局写前日志锁, 将当前缓冲区的日志写入磁盘, 释放锁,完成提交.

(6) 完成提交后, 执行线程释放本次事务申请的资源, 释放数据锁, 响应客户端.

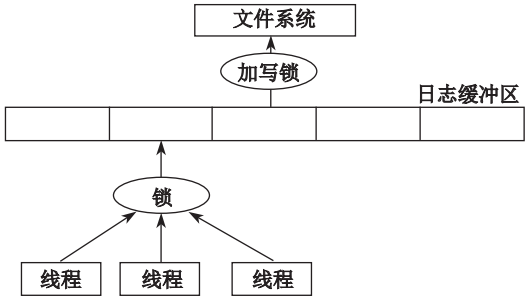

图 2 日志模块示意图

Fig. 2 Diagram of log module

然而, 以上写前日志设计会对事务产生 4 种类型的性能影响[26].

磁盘 I/O 造成的延迟. 在系统响应客户端前, 系统必须保证数据达到非易失性的存储介质. 然而, 磁盘写入会造成毫秒级的延迟. 且当磁盘写入由大量较短的日志时, 写入性能将进一步下降. 针对这个问题 Alpern等[27]采用了高性能的固态存储介质. Helland 等[25]提出的成组提交可以批量地向磁盘写入日志, 避免大量短日志的写入.

增长持锁时间. 在传统的写前日志下, 事务首先需要将所有的日志写入磁盘, 然后释放数据锁. 这增长了事务的持锁时间, 事务之间的冲突将增多, 从而会降低系统的事务并发.

额外的上下文切换. 由于写日志造成的I/O 阻塞将会导致事务的执行线程进入等待状态. 在日志写入磁盘后被唤醒执行. 一次上下文切换需要上千的指令操作, 这将造成 CPU 时间的浪费. 另一方面, 物理核数的增多使得系统需要调度更多的线程, 相应地, 上下文切换数量也会增加, 导致系统进行线程调度的压力骤增.

日志缓冲区访问冲突. 日志缓冲区由轮询锁保护, 确保同一个时刻只有一个线程在写入日志. 随着物理核数的升高, 并行线程对轮询锁的争夺将造成大量的 CPU 时间开销.

为了降低日志维护产生的延迟与冲突, 当前主要有以下工作.

成组提交与异步提交. 传统数据库普遍采用成组提交与异步提交来解释写日志减少磁盘延迟导致的锁冲突, 减少线程上下文的切换. 成组提交[25]累计多个事务的日志同时写入磁盘, 提高了单次磁盘的写入量, 能更加有效地利用磁盘写入带宽. 异步提交[27]允许日志尚未持久化前提交事务. 由于事务提交不需要等待日志写入磁盘, 因此可以更早地释放锁, 减少冲突. 并且执行线程不会因为磁盘I/O 而阻塞, 从而避免了上下文的切换. 然而, 当发生系统故障时, 异步提交可能由于尚未将日志写入磁盘而导致丢失部分已提交的数据. 由于异步提交带来了较大的性能优化, 多数商业数据库与开源数据库都实现了该方案.

日志持久化前释放锁. 事务提交时需要等待日志持久化后释放锁. 这是为了确保后续事务操作的数据必然是已提交的数据. 然而, 磁盘I/O 通常具有毫秒级的延迟. 这导致了加锁时间的延长, 事务之间的加锁冲突将会加剧, 最终影响系统的吞吐率. Dewitt 等[6]观察到, 当满足一定的约束提交时, 事务的锁可以在日志持久化前释放. 从事务释放锁到完成日志持久化的阶段称之为预提交阶段[28]. 若后续事务读取了预提交事务的数据, 则需要同时等待预提交事务完成持久化后再响应客户端. 文献 [28] 形式化地论述了该技术需要满足的条件. 当系统处理的请求由大量较短, 同时冲突率较高的事务构成时, 在持久化完成前释放锁将具有重要的意义. Wang 等[12]实验发现, 在处理 TPC-B 应用时, 若事务冲突率较高, 提前释放锁将带来数十倍的性能提升.

流水线式提交. 完成一次事务请求主要经过了事务执行, 事务提交和响应客户端 3 个过程. 由于事务提交过程会因为磁盘I/O 而导致线程阻塞, 触发上下文切换. 随着计算机物理核数的不断增大, 由此导致的系统调用将会占据一定比例的 CPU 资源. 现有的数据库系统通过异步提交的方式来避免写日志产生的线程阻塞, 代价是可能会丢失提交的数据. Johnson 等[26]提出了流水式的任务调度方式: 当事务进入提交阶段需要写日志时, 执行线程将当前事务压入队列, 转而执行其他的事务; 由唯一的提交线程将多个事务的日志成组写入磁盘, 并通知执行线程已经完成持久化的事务; 执行线程将事务执行结果响应客户端. 该方案避免了执行线程因为写日志的I/O 阻塞而触发上下文切换. 然而, 任务在多个线程间的切换调度将增加任务的整体延迟. Johnson 等[26]通过结合 Early Lock Release 技术, 避免延迟增加导致的加锁冲突的增多.

可扩展的日志缓冲区. 集中式的日志缓冲区存在扩展性的问题. 物理核数增多引起的并行度增大导致访问日志缓冲区的冲突也不断增大. 为了缓解对集中式日志缓冲区的访问冲突, Johnson 等[26]采用了两种方式: ①首先减少了访问日志缓冲区的临界区长度, 将向缓冲区拷贝数据的操作移出临界区; ②采用 Elimination based backoff 技术并行合并多个事务的日志请求, 减少访问临界区的次数.

基于 NVM 的分布式日志. Wang 等[12]采用非易失性内存保存日志, 由于在系统崩溃后日志仍旧可以在内存中保存, 因此事务提交时不需要立即将日志写到磁盘, 仅需当日志缓冲区满时写入磁盘. 作者基于此提出了 passive group commit 策略. 此外, 集中式的日志存在扩展性的瓶颈. 因此, 作者提出了基于 NVM 的分布式日志. 由于在传统架构上, 分布式日志在事务提交时会产生大量的短日志写磁盘请求, 这将导致较高的写入代价. 而采用 NVM 后, 日志仅需在每个日志缓冲区满后写入磁盘, 因此可以有效地避免分布式日志高昂的写入代价.

逻辑日志. Malviya 等[29]研究了针对 VoltDB 和 H-Store 的轻量级的数据恢复机制 Command Logging, 其基本思想是在日志中仅仅保存事务本身, 而不保存事务所涉及的元组更改前和更改后的状态. 这是因为这类系统中多个事务是串行执行的, 事务间不会交叉调度. 因此, 记录事务的执行顺序即可恢复出完整的操作历史. 实验结果表明这种恢复机制的开销更小, 适合采用确定性执行策略的内存数据库系统.

以上综述了当前最新工作对日志提交协议实现的优化, 在表 2 中总结了这些技术对系统产生的影响.

表 2 日志优化策略影响

Tab. 2 Impact of log optimization strategy

	降低磁盘I/O对性能的影响	避免提交过程中占据锁	减少提交导致的上下文切换	减少对日志缓冲区的冲突访问
成组提交与异步交	√	√	√	
日志持久化前释放锁		√		
流水线式提交			√	
可扩展的日志缓冲区				√
基于NVM的分布式日志				√
逻辑日志	√	√	√	

3.2 OceanBase 的日志模块设计

OceanBase 采用的是 ARIES 日志, 在系统运行中记录了所有数据修改操作的 Redo 日志. 在日志模块中, OceanBase 采用了与上述研究工作不同的设计, 并达到了高效的事务提交效率.

在现有的工作中, 事务提交时通过事务的执行线程共同访问日志管理器, 向日志缓冲区拷贝日志, 周期性的触发写磁盘来完成的. 对日志管理器的排他性访问以及磁盘 I/O 操作都会导致线程的阻塞. 文献 [12] 的优化方案虽然避免了磁盘 I/O 操作的阻塞, 但仅仅是缓解了对日志管理器冲突访问产生的阻塞. OceanBase 采用彻底的方案避免了提交造成的阻塞.

OceanBase 设计了单独的提交线程负责所有事务的提交, 彻底地流水线化了事务的提交过程, 流程见图 3. 事务提交过程如下:

(1) 事务在本地线程完成预提交工作. 申请版本号, 将版本号与数据写入内存表, 并采用 Early Lock Release 的优化提前释放锁.

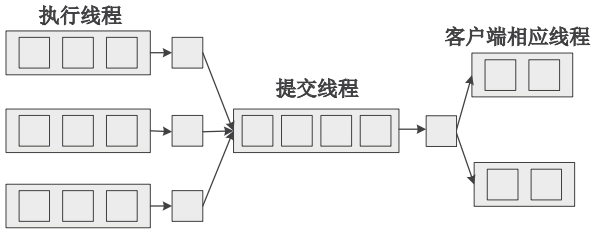

图 3 OceanBase 事务执行图

Fig. 3 Transaction execution diagram of OceanBase

(2) 事务按照版本号的顺序进入提交线程队列, 进入提交阶段. 事务的执行线程将继续调度其他的请求. 版本号的顺序确保了 Early Lock Release 的正确性.

(3) 提交线程依次将每个事务的日志拷贝到日志缓冲区, 并周期性的触发日志写磁盘与同步备机操作.

(4) 提交线程周期性地判断当前写入磁盘的最大日志号, 确定已经完成提交的事务, 并更新系统的发布版本号, 并将事务加入响应客户端的线程池中.

(5) 响应客户端的线程负责释放事务的内存资源, 并响应客户端.

这样的设计一方面避免了事务执行线程的阻塞, 另外一方面减少了系统内部对共享数据的冲突访问. 然而, 该方案的劣势也在于单线程的设计. 当提交任务过多时, 提交线程的处理时间将被耗尽, 提交任务将在队列中等待. 在一台主频 2.00 GHz 的服务器上, OceanBase 在处理每秒约 14 万个 的提交请求时, 提交线程将达到处理性能的极限.

4　数据持久化

虽然内存技术在近些年得到了长足的发展, 然而相比于磁盘, 内存的容量依然有限, 在很多场景下可能会出现数据大小大于内存容量的情形. 此外内存的价格通常也远高于磁盘的价格, 全内存的存储方式并不经济. 从应用的角度, 大量的应用负载是倾斜的, 少量的"热"数据会被频繁地访问, 大量的"冷"数据极少访问, "冷"数据由于修改频率较低, 并不需要始终存储在内存中. 最后, 为了保证系统能够快速地恢复, 同样需要周期性地将数据库快照持久化到磁盘中. 基于以上这些因素, 内存数据库系统仍然需要支持磁盘存储, 使数据存储在磁盘中. 在这一节中, 将主要介绍现有的内存数据库系统对磁盘的支持.

4.1　内存数据库的数据持久化

微软的 Stoica 等[30]研究指出大量的 OLTP 应用负载访问数据频率是倾斜的, 将大量的极少被访问的"冷"数据存储在磁盘将有利于减少内存的使用, 使得数据库系统更加经济. 作者提出了一种针对数据访问日志的离线分析算法, 采用 Exponential Smoothing 近似估计每条数据的访问频率, 将高频访问的数据存储在内存中. 该算法被应用在 Hekaton 系统中[31].

在 H-Store 系统中, Debrabant 等[32]提出了 Anti-Caching 策略, 同样是把一些"冷"数据放在磁盘中保存, 和传统的缓存策略相反, H-Store 系统中内存作为主存, 磁盘作为辅助存储. 当事务需要访问磁盘上的数据时候, 延迟执行此事务而在后台进行读磁盘操作, 当所需数据都读取到内存后再执行此事务.

文献 [33] 针对确定性数据库系统提出了 Prefetch 策略来支持磁盘存储, 其思想和 Anti-Caching 相同, 通过延迟事务的执行, 保证所需要的数据都已经读入内存中.

以上工作在不同的系统上主要解决以下几个问题: 如何区分"冷"、"热"数据以确定需要使用磁盘存储的数据, 数据是如何从内存转储到磁盘, 以及如何将数据从磁盘读取到内存. 在下一节中, 将重点介绍 OceanBase 使用的数据持久化策略.

4.2　OceanBase 的数据持久化设计

OceanBase 的数据持久化主要是通过周期性的每日合并将内存事务引擎中的增量修改数据与分布式文件系统上的基线数据合并存储.

OceanBase 区分"冷"、"热"数据的策略相对简单. 在每次数据合并时, 内存事务引擎中所有的数据都作为"冷"数据与分布式文件系统上的数据合并存储. 在下一次合并发生前, 所有被修改的数据将作为"热"数据再次维护在内存事务引擎中.

OceanBase 将数据从内存事务引擎保存到存储节点中主要通过每日合并完成. 随着数据写入的不断增多, 内存事务引擎中的数据会不断增加, 当数据大小到达一定阈值或者系统达到预设冻结时间的时候, OceanBase 会触发每日合并过程, 冻结当前内存引擎中的数据, 将其与分布式

文件系统中存储的基线数据合并. 每日合并完成后, 内存事务引擎中冻结的数据即可释放, 其占用的内存也被回收.

OceanBase 读取持久化的数据主要是在查询或者修改操作执行前, 根据需要访问的数据主键从存储节点上读取需要的基线数据.

以上策略保证了 OceanBase 能够收缩事务引擎使用的内存容量, 更为经济地使用磁盘存储, 并加速系统的灾难恢复过程.

5 结 论

本文主要介绍了在大内存、多核环境下事务型数据库系统相关的最新研究工作. 大量的工作表明, 数据库系统设计需要降低系统内部维护的开销, 将更多的计算时间用于实际有效的数据处理工作. 需要实现系统的多核扩展性, 以适应当前 CPU 物理核数不断增多的趋势. 同时, 系统可以引入对负载特点假设来改善内部设计, 通过设计专用的事务系统来获得更高的性能.

[参 考 文 献]

[1] GRAY J, REUTER A. Transaction Processing: Concepts and Techniques [M]. San Francisco: Margan Kaufmann, 2007.

[2] BAYER R, MCCREIGHT E M. Organization and maintenance of large ordered indexes [J]. Acta Informatica, 1972, 1(3): 173-189.

[3] AGRAWAL D, BERNSTEIN A J, GUPTA P, et al. Distributed optimistic concurrency control with reduced rollback [J]. Distributed Computing, 1987, 2(1): 45-59.

[4] BERNSTEIN P A, HADZILACOS V, GOODMAN N. Concurrency Control and Recovery in Database Systems [M]. MA: Addison-Wesley Publishing, 1987.

[5] FREEDMAN C S, ISMERT E, LARSON P, et al. Compilation in the Microsoft SQL Server Hekaton Engine [J]. IEEE Data(base) Engineering Bulletin, 2014: 22-30.

[6] DEWITT D J, KATZ R H, OLKEN F, et al. Implementation techniques for main memory database systems [J]. Acm Sigmod Record, 1984, 14(2): 1-8.

[7] 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, 1992, 17(1): 94-162.

[8] COBURN J, BUNKER T, SCHWARZ M, et al. From ARIES to MARS: Transaction support for next-generation, solid-state drives [C]// Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 197-212.

[9] 阳振坤. OceanBase 关系数据库架构 [J]. 华东师范大学学报 (自然科学版), 2014(5): 141-148.

[10] DIACONU C, FREEDMAN C, ISMERT E, et al. Hekaton:SQL server's memory-optimized OLTP engine [C]// ACM SIGMOD International Conference on Management of Data. ACM, 2013: 1243-1254.

[11] HARIZOPOULOS S, ABADI D J, MADDEN S, et al. OLTP through the looking glass, and what we found there [C]// ACM SIGMOD International Conference on Management of Data. ACM, 2008: 981-992.

[12] WANG T, JOHNSON R. Scalable logging through emerging non-volatile memory [J]. Proceedings of the Vldb Endowment, 2014, 7(10): 865-876.

[13] JOHNSON R, PANDIS I, AILAMAKI A. Improving OLTP scalability using speculative lock inheritance [J]. Proceedings of the Vldb Endowment, 2009, 2(1): 479-489.

[14] TU S, ZHENG W, KOHLER E, et al. Speedy transactions in multicore in-memory databases [C]// Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 18-32.

[15] BLANAS S, DIACONU C, FREEDMAN C, et al. High-performance concurrency control mechanisms for main-memory databases [J]. Proceedings of the Vldb Endowment, 2011, 5(4): 298-309.

[16] THOMASIAN A. Two-phase locking performance and its thrashing behavior [J]. ACM Transactions on Database Systems, 1993, 18(4): 579-625.

[17] KUNG H T. On optimistic methods for concurrency control [J]. Acm Transactions on Database Systems, 1981, 6(2): 213-226.

[18] SADOGHI M, CANIM M, BHATTACHARJEE B, et al. Reducing database locking contention through multi-version concurrency [J]. Proceedings of the Vldb Endowment, 2014, 7(13): 1331-1342.

[19] REN K, THOMSON A, ABADI D J. Lightweight locking for main memory database systems [C]// International Conference on Very Large Data Bases. VLDB Endowment, 2012: 145-156.

[20] YU X. An evaluation of concurrency control with one thousand cores [D]. Boston: Massachusetts Institute of Technology, 2015.

[21] PANDIS I, JOHNSON R, HARDAVELLAS N, et al. Data-oriented transaction execution [J]. Proceedings of the Vldb Endowment, 2010, 3(1/2): 928-939.

[22] THOMSON A, THOMSON A, ABADI D J. An evaluation of the advantages and disadvantages of deterministic database systems [J]. Proceedings of the Vldb Endowment, 2014, 7(10): 821-832.

[23] PAVLO A, CURINO C, ZDONIK S B, et al. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems [C]//International conference on management of data, 2012: 61-72.

[24] GOTTEMUKKALA V, LEHMAN T J. Locking and latching in a memory-resident database system [C]// Intl Conf on Very Large Databases, 1992: 533–544.

[25] HELLAND P, SAMMER H, LYON J, et al. Group Commit Timers and High Volume Transaction Systems [C]//High performance transaction systems workshop, 1987: 301-329.

[26] JOHNSON R, PANDIS I, STOICA R, et al. Aether: a scalable approach to logging [J]. Proceedings of the Vldb Endowment, 2010, 3(1/2): 681-692.

[27] ALPERN D, ARORA G, BARCLAY C, et al. Oracle Database Advanced Application Developer's Guide, 11g Release 2 (11.2) E17125-06[R/OL]. [2018-07-10]. https://docs.oracle.com/cd/E11882_01/appdev.112/e41502/toc.htm.

[28] SOISALON-SOININEN E, YLÖNEN T. Partial strictness in two-phase locking [C]// International Conference on Database Theory. Springer-Verlag, 1995: 139-147.

[29] MALVIYA N, WEISBERG A, MADDEN S, et al. Rethinking main memory OLTP recovery [C]// International Conference on Data Engineering. IEEE, 2014: 604-615.

[30] STOICA R, LEVANDOSKI J J, LARSON P A. Identifying hot and cold data in main-memory databases [C]//International Conference on Data Engineering. IEEE, 2013: 26-37.

[31] ELDAWY A, LEVANDOSKI J, LARSON P Å. Trekking through Siberia: Managing cold data in a memory-optimized database [J]. Proceedings of the Vldb Endowment, 2014,7(11): 931-942.

[32] DEBRABANT J, PAVLO A, TU S, et al. Anti-caching: A new approach to database management system architecture [J]. Proceedings of the Vldb Endowment, 2013, 6(14): 1942-1953.

[33] THOMSON A, DIAMOND T, WENG S C, et al. Calvin: Fast distributed transactions for partitioned database systems [C]//ACM SIGMOD International Conference on Management of Data. ACM, 2012: 1-12.

(责任编辑: 李万会)

(上接第 90 页)

[14] ESWARAN K P, GRAY J N, LORIE R A, et al. The notions of consistency and predicate locks in a database system [J]. Readings in Artificial Intelligence & Databases, 1989, 19(11): 523-532.

[15] Sysbench. [EB/OL]. [2018-07-02]. https://dev.mysql.com/downloads/benchmarks.html.

(责任编辑: 林　磊)