多核处理器下事务型数据库性能优化技术综述

朱阅岸[1),2)] 周 烜[1),2)] 张延松[1),2),3)] 周 明[1),2)] 牛 嘉[1),2)] 王 珊[1),2)]

[1)](中国人民大学信息学院 北京 100872)

[2)](数据工程与知识工程教育部重点实验室(中国人民大学) 北京 100872)

[3)](中国人民大学中国调查与数据中心 北京 100872)

摘 要 传统数据库的设计假设磁盘为主要存储设备,其性能取决于基于 I/O 代价模型的优化.然而,当前数据库运行的平台已逐渐转移到由多核处理器、大内存和以闪存为代表的低延迟存储所构成的新型硬件平台上.在大多数情况下,工作数据集能够全部加载到内存或者闪存等高速存储器中.这样,数据库的性能瓶颈由传统的 I/O 转移到 CPU 上.而传统数据库的加锁操作、闩锁竞争、日志管理以及缓冲区管理在设计时均未考虑到多核处理器的使用,因而成为了限制 CPU 利用率的明显瓶颈.改变传统数据库的优化重点以适应硬件的发展对应用而言是十分必要的.该文针对当前新的应用背景,主要围绕数据库系统中锁管理、日志管理、缓冲区管理以及 B 树索引等核心模块在多核平台下已有的优化技术进行详细介绍和归纳总结.同时介绍了中国人民大学在数据库系统的多核处理器优化方面所做的一些工作.

关键词 数据库系统优化;锁;日志;缓冲区管理;B 树

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2015.01865

A Survey of Optimization Methods for Transactional Database in Multi-Core Era

ZHU Yue-An[1),2)] ZHOU Xuan[1),2)] ZHANG Yan-Song[1),2),3)]
ZHOU Ming[1),2)] NIU Jia[1),2)] WANG Shan[1),2)]

[1)](*School of Information, Renmin University of China, Beijing* 100872)

[2)](*Key Laboratory of Data Engineering and Knowledge Engineering of Ministry of Education (Renmin University of China), Beijing* 100872)

[3)](*National Survey Research Center at Renmin University of China, Beijing* 100872)

Abstract The design of traditional DBMS assumes magnetic disk as the storage device. Its optimization techniques are focused on the reduction of I/O cost. However, the database platforms in the future will be dominated by multi-core processors, large main memory and low-latency semiconductor storage, such as SSD. On such platforms, the entire data set can normally fit into main memory or high-speed storage. Thus, the performance bottleneck of query execution has shifted from disk I/O to CPU. The components for locking, latching, logging and buffer management of traditional DBMS were not originally designed for multi-core processors. These components severely prohibit the scalability of DBMS in multi-core architectures. Adaptation of traditional DBMS to new hardware is a common and necessary practice. This paper provides a survey of the recent optimization techniques proposed for DBMS on multi-core platforms. Meanwhile, the efforts by Renmin University of China on DBMS optimization on multi-core platform are introduced.

Keywords database system optimization; lock; log; buffer management; B-tree

收稿日期:2014-12-22;最终修改稿收到日期:2015-04-30.本课题得到国家自然科学基金(61272138,61232007)、中央高校基本科研业务费专项资金(12XNQ072,13XNLF01)和中国人民大学研究生科学研究基金(13XNH216)资助.**朱阅岸**,男,1983 年生,博士研究生,主要研究方向为高性能数据库系统.E-mail:iwillgoon@126.com.**周 烜**(通信作者),男,1979 年生,博士,副教授,主要研究方向为高性能数据库、信息检索.E-mail:zhou.xuan@outlook.com.**张延松**,男,1973 年生,博士,讲师,主要研究方向为内存数据库、OLAP.**周 明**,男,1990 年生,硕士,主要研究方向为内存数据库.**牛 嘉**,女,1989 年生,硕士,主要研究方向为高性能数据库.**王 珊**,女,1944 年生,教授,博士生导师,中国计算机学会(CCF)高级会员,主要研究领域为高性能数据库、知识工程、数据仓库.

1 引 言

摩尔定律揭示了集成电路上晶体管集成度的增长规律,摩尔定律首先表现为 CPU 主频的持续增长,然后又表现为 CPU 核心数量的持续增长. CPU 的频率持续逐年增长,加上其指令执行的并行度也在增加,使得单线程的工作性能不断提高[1,2]. 早在 2002 年,CPU 的频率就已经达到 3GHz. 然而,高频率的 CPU 带来的是更多的能耗,现实应用中不一定会带来高的性价比,因而阻止 CPU 主频的进一步发展. 这个问题使得近年来 CPU 的发展方向产生了转变. 为了进一步优化 CPU 的性能,生产商倾向于在单个 CPU 上增加更多的处理核心. CPU 开始转向使用并行多线程和片上多处理器技术,以替代单处理流的高度复杂 CPU 技术. 生产商不再竞争速度而是转向提高并行度. 按照最近多核的发展趋势,单个 CPU 上的核数基本每两年翻一倍. 大部分研究人员预测,CPU 的核数会继续增长一段时间. 为了更好地利用新 CPU 的并行处理能力,工程师需要修改或重新设计软件系统. 而基础软件,包括操作系统、Web 服务器和数据库等,则会最先遇到复杂的多核架构带来的扩展性问题.

操作系统社区对 Unix 类型操作系统在共享内存多处理器上的扩展性研究已经进行了很长一段时间. 大家公认粗粒度的锁是导致系统扩展性差的主要原因. 现有的研究成果均建议将多核架构看作分布式系统,利用共享内存进行快速消息传递[3],系统设计的主要思想集中在增加数据局部性和均衡处理核心之间的负载[3-4]. 诸多实用的研究成果,如分布式内存管理、可扩展的封锁技术、避免等待的同步技术和针对多处理器调度技术,都已经被工业界采纳,并被植入了新的 Linux 内核[4].

从一开始,数据库存储引擎的研究就是定位在高负载下的高性能和可靠性. 一方面,数据的可靠性和一致性大都通过在共享数据结构上的封锁机制实现;另一方面,在现代多核架构下,性能的提升必须通过利用更高程度的并行性. 如何解决这两者之间的矛盾是一个很大的挑战. 现实中,关系数据库系统大部分是基于优化 I/O 和面向单处理器的陈旧设计思想,并不适合多核处理器的架构.

以事务处理为主的数据库系统多数依赖并发取得高吞吐率. 并发是通过多线程或者多进程实现的. 理想情况下,随着硬件上下文的增加,他们应该具备良好的扩展性,特别是针对以读为主工作负载. 实际上,为了维护一致性和持久性,数据库系统大量使用共享数据结构和同步原语. 在高度并行的环境下,对共享数据进行频繁的原子更新不得不由线程串行地执行[5]. 因此,用于保护共享数据结构的同步原语会导致很大的同步开销. 这些同步原语大量存在于共享内存缓冲区、锁表、索引与日志管理器中. 此外,丰富的硬件上下文使得并行线程竞争硬件资源,例如,高速缓存的争用将降低缓存命中率从而增加内存访问延迟. 另外,传统的封锁方式例如阻塞和忙等待策略在多核环境下效率低下[6]. 增加的并行度还会导致更大异构的工作负载,这给索引数据结构里的读写同步控制原语也造成更大的压力[7]. 总之,所有这些原因导致了运行在多核架构上的数据库系统的性能瓶颈. 近年来,学者们提出许多不同的思想和方法,用于重新构建多核环境下的数据库系统.

本文的目的是对已有的多核数据库优化技术进行总结和讨论. 同时介绍了中国人民大学在数据库多核优化所做的一些努力. 纵观整个数据库系统内核,多核扩展的瓶颈突出表现于锁管理、日志管理器、缓冲区管理和索引(主要是 B-tree)四大部件. 本文依次对这四大部件的多核优化技术做整理归纳.

2 锁管理器的多核优化

几乎所有的数据库都利用某种形式的多粒度封锁机制,允许应用程序在并发度和系统开销之间进行一个折中. 多粒度锁机制将数据库视为嵌套的数据结构. 数据库中包含表,表又由一系列页面组成,页面包含元组. 每个层次的数据结构都有一个锁与之相对应. 例如,访问大量数据的请求可以申请大粒度的锁对表进行封锁,牺牲并发度以减少系统开销. 对于访问少量数据的请求可以只对它们需要访问的数据加锁,以最大化系统并发度. 多粒度锁机制对于系统的扩展性至关重要,因为它在逻辑层面上实现高效的细粒度并发控制. 这种多粒度的锁机制可以有效控制系统的并发度,然而也给系统扩展性带来新的问题. 单节点的数据库系统在处理器数目较少的时候利用分时机制处理请求,对资源的竞争不明显. 随着处理器上核数的不断增加,硬件并发度的增大,集中式的锁管理器遇到了瓶颈,尤其是当多粒度锁强制许多线程不断地更新某些频繁访问的锁的状态时. 为了访问数据库系统的某个元素,所有的事务都必须获得高层次上的意向锁,使得这些意向锁的访问异常频繁. 对锁的访问必须具备原子性,导致临界区资源竞争. 对于具有良好扩展性的数据库程序

而言也会遇到与锁相关的瓶颈，即使它们几乎不会引起逻辑上的冲突.

为了降低锁管理器的资源竞争，要么降低加锁的次数，要么减轻加锁过程中原子操作的排他性，要么进一步分布化并行化锁管理器从而缓解对锁资源的争夺. 以下逐一介绍现有的 3 种锁管理器多核优化策略.

2.1　锁的传递与继承

Rdb/VMS[8] 利用投机锁继承机制减少网络传输代价. 在这个分布式数据库中，系统的任意一个节点可将锁传递到请求该锁的另外一个节点上. 文献[1]简要地描述了“锁传递”的思想：只要没有冲突的锁请求到来，可以将锁缓存在本地，而不用在事务结束的时候将锁返回给主节点. 这种方式避免了将锁返回给主节点的开销. 如果锁可以被后面的事务重用，那么每一次锁传递可以省略一趟网络传输. 对于两节点的系统而言，性能可以提高 60%以上. Johnson 等人[9]将锁传递思想应用在单节点的数据库系统上以解决锁状态上的竞争.

不同于 Rdb/VMS 的高网络传输代价和较少的节点数目，文献[9]在共享高速缓存和内存的多核平台上利用单节点 Shore-MT 数据库系统[10] 实现投机锁继承算法 SLI(Speculative Lock Inheritance，SLI). SLI 将锁管理器上的封锁请求分发到不同的线程，以减少冲突. SLI 思想主要基于以下观察：应用程序几乎总是以相容模式申请某些热锁；否则，若某一时刻系统的大部分事务被阻塞在某个排它锁上，不会为更新锁的状态而造成竞争. 如果大部分事务以相容模式申请热锁，那么事务可以较长时间持有该锁而不会降低系统并发度. 这个锁只需保证少数更新事务可以正确地隔离其他事务. SLI 利用被频繁访问的共享锁无逻辑上的竞争，消除在锁内部临界区的物理竞争. SLI 允许事务结束的时候将它所持有的锁传递给下一个事务. 这种策略避免了对每一个传递的锁申请和释放时都需要调用锁管理器. 在事务提交的时候，事务的代理线程挑选候选传递锁，然后将这些锁存放于线程的局部锁列表，并用局部锁列表初始化下一个事务的锁列表. SLI 机制利用以下 5 条准则选取锁作为候选继承锁：(1) 这个锁是页面上或者是更高层次上的锁；(2) 这个锁被频繁访问；(3) 这个锁以共享模式(S、IS、IX)为某个事务所持有；(4) 没有别的事务等待这个锁；(5) 如果该锁上存在更高层次上的锁，上述条件仍然成立. SLI 通过以下两个途径优化性能：(1) 减少事务的锁申请和释放的次数，从而取得较快的响应时间，多个短事务可以分摊锁申请开销；(2) 申请锁的其他事务面临较少的锁管理器内部临界区的竞争，这种方法适用于短事务，以便分摊锁申请和释放的开销. 另外，投机锁继承也仅限于共享锁和页级以上的锁，并且没有其他事务等待该锁. 文献[11]利用消除闩锁(latch-free)的数据结构取代传统的锁和缓冲区哈希表，降低了事务申请锁和释放锁的开销. 这种方法能够取得与投机锁继承方法接近的性能，但是不会受到事务特性限制.

IBM DB2 数据库系统也提供了一个参数 DB2_KEEPTABLELOCK①，允许事务甚至是会话持有只读模式的表级锁，直到必要才释放锁. 然而，只有当事务不断地释放然后又重新申请相同的锁的情况下，这种策略才会有收益. IBM DB2 文档提示在一个会话的生命周期内一直持有表级锁会导致较差的并发性能，因为其他事务不能执行更新操作. 这个设置在 DB2 默认是禁用的.

2.2　轻量级并发原语

现代 CPU 提供多种并发控制原语，其中在数据库常用的两种为：原子读后写(Atomic Write After Read，AWAR)和写后读(Read After Write，RAW). 在 AWAR 中，线程/进程以原子操作的方式读取一个公共变量然后写回. AWAR 的原子性是通过闩锁(例如旋转锁，spin lock)而获得的. 闩锁的实现依赖机器的原子指令例如 compare-and-swap 或 test-and-set. 在基于探测方式保持高速缓存一致性的现代多处理器体系结构上，AWAR 会导致高速缓存失效，占用系统总线带宽. 这是因为当某一个进程释放闩锁以后，它需要将 *false* 写入锁变量. 这会使得其他竞争者的相应的高速缓存内的锁变量失效. 其他竞争者各承受一次高速缓存未命中，重新读入新值，接着调用 test-and-set 指令获取锁. 第一个成功获得锁的进程又会使得其他竞争者的高速缓存的锁变量副本失效，从而又需要再次读取内存导致总线拥挤. 另外一个可替代的编程方式是 RAW. 具体的方式是进程 P1 写某一公共变量 A，接着该进程读取另外一个不同的公共变量 B. 若在这之间没有别的进程写公共变量 B，RAW 可以保证并发进程看到共享变量的顺序一致性状态. 利用 RAW 编程方式必须确保并发进程只能看到共享变量的可串行化的一致性状态. 图 1(a)展示了可能出现的问题.

① http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp

进程 P 和 Q 根据 RAW 模式访问公共变量 A 和 B. 在没有限制的情形下，如图 1(a)，数据的竞争可能导致进程 P 看见 $B=0$ 和进程 Q 看见 $A=0$(由于高速缓存内容的改变没有及时反映到内存上). 这违背了可串行化原则. 图 1(b)展示了利用内存栅障(memory barrier)协调进程访问内存的顺序以保证程序执行的正确性，实现可串行化，也就是保证进程 P 和 Q 能够看到这些写操作. 执行内存栅障会带来一定的开销，但是每一个进程只需付出一次硬件代价. 相比而言，AWAR 会导致高速缓存块在进程之间频繁切换，造成系统颠簸. Jung 等人[5]认为在多核平台上 RAW 编程模式更具有扩展性. 利用 RAW，他们将锁管理器修改为几乎不用闩锁的方式. 修改后的锁申请和释放的伪代码如表 1 所示. 传统的锁管理器中，事务首先在数项上创建一个锁. 然后将这个锁插入这个数据项上的锁列表(这时候需要对锁表进行封锁). 然后，这个事务遍历该数据项上的锁列表，检测是否存在不相容的锁. 如果存在，则标记这个锁为等待状态，接着进行死锁检测. 利用原子操作可以去除哈希表上的闩锁. 右侧是改写后去除闩锁的代码，它用函数 *atomic_lock_insert*() 将锁以原子操作的方式加入哈希表. 与安全的 RAW 一致，每次写一个共享变量，接着读另外一个不同的共享变量都要有内存栅障的保护. 修改后的代码中增加了一个新的锁状态 *OBSELETE*. 这个新的状态是为了批量回收锁的内存而设计的，表示

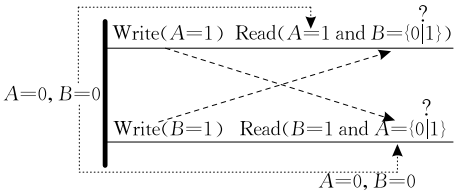
表 1 MySQL 中锁的申请和释放的实现(左边的代表原来的实现，右边的代表减少闩锁的实现. S1，S2，…，S7 表示需要加 RAW 同步的地方)

传统的锁管理器	改进的锁管理器
增长阶段的锁请求	

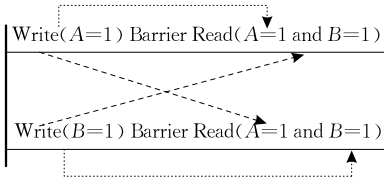
mutex_enter(lock_table→mutex);
n_lock=*lock_creat*( );
n_lock→state=ACTIVE;
lock_insert(*n_lock*);
FOR all locks(*lock*) in the hash_bucket
IF(*lock* is incompatible with *n_lock*) Begin
n_lock→state=WAIT;
IF(*deadlock_check*()==TRUE)
Abort *Tx*;
BREAK;
ELSE
CONTINUE;
END IF
END IF
END FOR
mutex_exit(*lock_table*→mutex);
IF(*n_lock*→state==wait)
mutex_enter(*Tx*→mutex);
Tx→state=WAIT;
Os_cond_wait(*Tx*→mutex)
mutex_exit(*Tx*→mutex)
END IF
	n_lock=*lock_creat*();
	n_lock→state=ACTIVE;
	***atomic_lock_insert*(*n_lock*); //S1**
	FOR all locks(*lock*) in the hash_bucket BEGIN
	IF(*lock* is incompatible with *n_lock*)
	***n_lock*→state=WAIT; //S2**
	atomic_synchronize();
	IF(lock→state==OBSELETE) BEGIN
	***n_lock*→state=ACTIVE; //S3**
	atomic_synchronize();
	CONTINUE;
	END IF
	IF(*new_deadlock_check*()==TRUE) BEGIN
	Abort *Tx*;
	BREAK;
	END IF
	ELSE
	CONTINUE;
	END IF
	END FOR
	IF(*n_lock*→state==WAIT)
	***mutex_enter*(*Tx*→mutex);//S4**
	atomic_synchronize();
	IF(*n_lock* has to wait)
	Tx→state=WAIT;
	Os_cond_wait(*Tx*→mutex)
	ELSE
	***n_lock*→state=ACTIVE; //S5**
	atomic_synchronize();
	END IF
	mutex_exit(*Tx*→mutex)
	END IF

收缩阶段的锁请求	

mutex_enter(lock_table→mutex);
FOR all locks(*lock1*) in *Tx* BEGIN
lock_release(*lock1*);
FOR all locks(*lock2*) following *lock1* BEGIN
IF *lock2* does have to wait BEGIN
lock_grant(*lock2*)
lock2→state=ACTIVE;
END IF
END FOR
END FOR
mutex_exit(lock_table→mutex);
	FOR all locks(*lock1*) in *Tx* BEGIN
	***lock1*→state=OBSELETE; //S6**
	atomic_synchronize( )
	FOR all locks(*lock2*) following *lock1* BEGIN
	mutex_enter(*lock2*→*Tx*→mutex)
	IF(*lock2*→*Tx*→mutex==WAIT &&
	lock2 does not have to wait) BEIGN
	***lock2*→*Tx*→mutex=ACTIVE; //S7**
	atomic_synchronize( );
	Os_cond_signal(*lock2*→*Tx*);
	Mutex_exit(lock2→Tx→mutex);
	END IF
	END FOR
	END FOR



(a) 不可串行化的执行



(b) 带内存栅障的安全RAW

图 1　安全的与不安全的 RAW 比较

这个锁不再有效. 同时,需要设计新的死锁检测算法,以避免竞争. 修改后的锁释放代码也去除了闩锁,根据安全的 RAW 模式增加内存栅障. 注意到修改后的锁管理器只是将锁标记为 *OBSELETE* 而没有释放结构体的内存. 这个阶段不会产生悬挂指针. 实际的锁内存释放以批量的方式异步进行. 重要的是,锁释放代码需要检查在这个锁之后申请的其他锁,唤醒因为申请与当前锁不相容的锁而被阻塞的事务. 文献[12]的核心思想是将锁数据结构的内存分配和释放与事务对锁的申请和释放解耦合. 对锁数据结构的内存分配和释放以批量的方式进行,与事务处理异步.

批量方式进行锁分配和回收的方法的潜在问题是过时的锁存在于锁管理器的哈希链表中,从而使得链表较长.

2.3　分散锁管理器功能

Ren 等人[13]提出的技术彻底更改了传统的集中式锁管理策略. 一是 VLL(Very Lightweight Locking). 该技术对锁管理器作了两点主要改变:不再将锁信息集中存储,而将锁的信息与数据一起存储,例如行级锁,在表的属性中增加一个隐藏属性,来记录每一行记录的锁信息,这样数据和锁信息位于同一个高速缓存块里,可以提高高速缓存命中率;另一点是用信号量(整数)来取代请求列表,并且给事务排序,事务按照到达顺序请求锁,且在事务开始执行前一次性请求全部锁. 另一种技术是 SCA (Selective Contention Analysis),该技术用来解决 VLL 并发性差、CPU 利用率低的问题.

VLL 技术取消了传统锁管理器使用的列表,改而为每一个加锁对象维护一个整数对(Cx,Cs),分别代表请求该对象的排它锁、共享锁的事务数. 每个请求锁的操作,只用简单地将对应的 Cx 或 Cs 加一,与之相对应,每个释放锁的操作,只用简单地将对应的 Cx 或 Cs 减一. 显然,当事务请求某个对象的排它锁时,将该锁对应的 Cx 加一,当且仅当此时 $Cx=1$,$Cs=0$,事务获得该排它锁;当事务请求某个对象的共享锁时,将该锁对应的 Cs 加一,当且仅当此时 $Cx=0$,事务获得该共享锁. 除此之外,在每个 partition,都有一个事务请求序列 TxnQueue. 当新事务到达时,它一次性请求所有锁(即将对应的 Cx 和 Cs 加一)后加入请求队列. 在事务请求锁之后,系统检查它请求的所有锁,如果事务可以顺利获得请求的所有锁,该事务加入请求队列 TxnQueue 时被标记为 Free,且立即执行;反之,事务被标记为阻塞,直到被系统判定为可以解除阻塞后再执行. 事务提交时,释放所有锁(即将对应的 Cx 和 Cs 减一),然后从 TxnQueue 移除. 在传统的锁管理器中,当锁释放时,通过遍历锁的请求列表来决定继承锁的事务,而 VLL 里没有请求锁的事务的信息,它只是简单的根据以下的理由选择队列中排在最靠前的阻塞事务来解除阻塞然后执行:排在队列头部的事务,之前使得它阻塞的事务已经全部提交且从队列中移除了,所以这个事务肯定可以获得所有锁. 这样一来,系统中每个时刻最多只有一个阻塞的事务可以被解除阻塞并执行. 随着事务的执行、提交,其他很多在传统锁管理情况下已经可以被解除阻塞(使它阻塞的事务已经提交了)的事务,也只能等待排在它前面所有的事务都提交才能被解除阻塞后执行,这大大降低了竞争激烈时系统的并发性.

为了解决以上问题,提高并发度,提高系统的 CPU 利用率,文献[13]提出了 SCA 技术. SCA 指在需要的时候(CPU 有空闲时,例如队列满)进行竞争分析,在 TxnQueue 中寻找可以被提前唤醒的事务. 在 TxnQueue 中,排在第 i 个位置的事务,只有可能被前 $i-1$ 个事务阻塞,所以越靠前的事务,被唤醒提前执行的可能性就越大.

SCA 维护两个大小为 100 KB 的数组 Dx 和 Ds,SCA 从 TxnQueue 头开始逐一扫描事务,对于每个事务 T:对于 T 的 ReadSet 中所有 key,将 Ds[hash(key)]置 1,对于 T 的 WriteSet 中的所有 key,将 Dx[hash(key)]置 1. 在扫描事务的同时,如果事务是阻塞的,将 Dx 和 Ds 置位之前,检查 T 的读集合的每一个 key 对应的 Dx 是否为 0,检查 T 的写集合的每一个 key 对应的 Ds 和 Dx 是否都为 0,如果全部满足,则代表 T 与前面的事务所请求的

对象没有锁冲突,即 T 可以被 unblock 并执行,此时 SCA 返回 T. 这样在系统 CPU 空闲时,SCA 算法能找到另外的不在队列头部但可以被解除阻塞并执行的事务.

SCA 方法的缺点是为了提高系统并发度必须维护一个全局的读锁集合与写锁集合. 所有事务的读锁集合与写锁集合都会被映射到这两个集合. 确定在事务队列是否存在可以提前执行的事务需要在 CPU 空闲的时候不断扫描这两个集合. 如果某个事务持有的锁不会与事务队列中位于该事务之前的事务持有的锁冲突(通过扫描全局读锁集合与写集集合)则该事务可以提前执行. 如果是计算密集型的事务类型,SCA 退化为 VLL. 当竞争因子小于 0.01 的时候(竞争因子$=\frac{1}{\text{热元组数目}}$),VLL 占据大约 2% 的系统执行时间,而传统的锁管理器占系统执行时间的比重大约为 21%. 相比之下,VLL 性能提升很大;当竞争因子大于 0.08 的时候,VLL 的性能不如传统的集中式锁管理策略.

3 日志管理的多核优化

日志管理器是数据库系统的一个重要部件. 几乎所有的数据库系统都使用集中式的预先写日志策略避免在系统崩溃的时候带来的数据的损坏和丢失已提交工作,以保证事务的持久性①[14-15]. 由于其集中式的设计和对 I/O 的依赖使得它很容易成为性能瓶颈. 较长的日志刷新时间、由日志带来的锁竞争、日志缓冲区上的竞争都会影响系统的扩展性. 日志操作给系统带来的延迟主要是 4 类[16]:(1) 系统必须保证事务提交之前日志必须写到非易失性存储介质;因为磁盘的访问时间是毫秒级别的,刷新日志通常为事务中执行时间最长的部分;此外,当很多小的 I/O 请求使得记录日志的设备例如 SSD 上达到饱和状态时,记录日志的延迟变为串行;(2) 在刷新日志的过程中,事务一直持有写锁,直到 I/O 结束;在很多工作负载中,这都会造成瓶颈,特别是锁竞争激烈的场景;(3) 刷新日志除了 I/O 延迟以外还有其他开销;在等待 I/O 完成的过程中,事务不能继续执行,代理线程必须被挂起,直到 I/O 完成;与 I/O 延迟不同,上下文切换和调度决策会消耗 CPU 时间,不能重叠执行其他任务;多核硬件环境下同时运行的线程很多,这使得操作系统调度器会负载过重;(4) 除了逻辑上锁的竞争和上下文切换的开销,许多线程同时想要执行日志插入操作;集中式的日志缓冲区有明显的临界区,其上的竞争显然也会影响系统的扩展性.

为了提高日志管理器的扩展性,必须设法提高日志的 I/O 效率,并且减少日志临界区的数量和缩短关键路径的长度.

成组提交技术[17]通过将许多小的刷新日志的请求组合到单个 I/O 操作减缓磁盘的压力,减少 I/O 等待时间. 成组提交技术能够减少磁盘访问次数和增大读取的磁盘块,进而减少磁头转动获得更好的响应时间. 但是成组提交技术不能消除不必要的上下文切换开销,因为过多的事务会阻塞来自日志管理器的挂起通知. 异步提交技术②结合了成组提交技术的优点,将许多刷新日志的请求组合到一起,并且允许事务结束而不用等待刷新日志操作完成. 这个优化将日志刷新操作从关键路径上完全移除,但是牺牲了事务持久性这个特性,也就是说,已提交工作有可能由于系统崩溃而丢失.

Johnson 等人[16]针对多核平台优化了的写日志操作. 他们重新审视了提前锁释放技术(Early Lock Release,ELR)[18],并在 4 个不同延迟的非易失性设备上测试了 ELR 的有效性. ELR 技术是指事务在 commit 记录到达磁盘之前可以释放锁,减少持有锁的时间. 在 ELR 技术下,只有提交事务需要等待 I/O 结束. 不存在其他事务需要等待提交事务的锁而与提交事务一起等待 I/O 结束. 文献[16]的实验结果表明在越慢的设备上,ELR 获得的收益越好. 即使是在像 SSD 上的快速磁盘设备,ELR 也可以获得比较好的收益(短事务的执行时间比 I/O 要短得多). 另外,数据的偏斜程度也对 ELR 造成影响. 如果数据偏斜度小,对锁的竞争不激烈,ELR 减缓锁的竞争效果不大. 如果数据的偏斜度大,锁的竞争很激烈,即使在没有等待日志刷新的情况下,ELR 效果也不好. 按照 80% 的访问都是集中在 20% 的数据上的原则,对应数据的偏斜度为 0.85 左右,这时候锁的竞争程度刚好是 ELR 的优化点. 针对多核平台上过多的调度而导致的系统瓶颈和上下文切换带来的开销,文献[16]提出一种新的刷新日志的方法:流水线方式刷新日志(flush pipelining). 流水线刷

① http://www.ibm.com/developer
http://support.microsoft.com/kb/230785
https://docs.oracle.com/cd/B19306_01/server.102/b14231.pdf

② http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14251/adfns_sqlproc.htm
http://www.postgresql.org/files/documentation/pdf/8.4/postgresql-8.4.2-A4.pdf.

新日志类似异步提交方式不用在等待 I/O 的时候将线程挂起，从而不会有上下文切换的开销. 但是不同于异步提交方式，流水线刷新不将结果返回给客户端，而是转而执行其他事务. 守护进程在完成刷新日志以后，通知代理线程/进程返回继续执行后续事务. 实验表明，流水线刷新和 ELR 组合的效果能够达到异步提交的性能，而且在系统崩溃的时候具有可恢复性.

为减缓多核情况下日志缓冲区上的竞争，文献[16]设计了 3 种新的日志缓冲区管理方法. 传统的写日志缓冲区需要以下 3 个步骤：(1) 首先获取写日志缓冲区上的排它锁. 如果当时正好有其他的服务子线程在写日志缓冲区，则此子线程必须等待直到获得写日志缓冲区上的写锁；(2) 线程将日志记录复制到相应的日志缓冲区；(3) 释放缓冲区上的锁. 由于它的简单性，这种方法具有吸引力. 这个方法的缺点是：即使是缓冲区从来不会重叠的情况下，填充日志缓冲区的操作也是串行化执行. 图 2(B)展示了由于单一的长日志记录会给后续的线程造成比较大的延迟. 日志记录由一个头部加上任意长度的值组成. 日志记录结构体空间的申请是可复合的，也就是两个连续的日志记录的缓冲区申请也可以由一个头部加上任意长度的属性值组成. 文献[16]利用这种空间的可复合性将线程对日志缓冲区的填充按组进行. 每一个组都有一个组织者. 一个组只有组织者才要竞争缓冲区上的锁，且一个组只有最后离开的线程需要等待锁的释放. 组织者在等待互斥变量的时候，后面到来的请求可以“回退”到一个数组将他们的请求组合到一起. 如图 2(C)所示，组内日志缓冲区的填充可以并行执行，但组之间仍然串行执行. 由于日志缓冲的填充不具有串行特征，只要满足以 LSN 的顺序将日志写回即可. 文献[16]修改了原来的日志缓冲区填充算法，线程申请的锁可以在获得缓冲区以后马上释放. 因此将缓冲区填充与锁的持有解耦合. 缓冲区的填充可以按流水线的方式进行：下一个缓冲区的填充可以立刻开始，只要线程获得日志缓冲区空间. 如图 2(D)所示，将锁的持有与缓冲区的填充解耦合可以消除长日志记录对缓冲区填充的影响. 前面讨论的两张方法具有互补性，因此将两种方法组合在一起，把日志缓冲区上的竞争限制在某一个常数下，同时也消除日志记录的大小对刷新日志的影响.

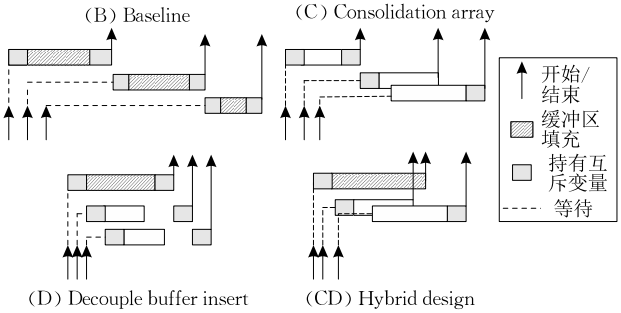


图 2　不同的日志插入方式

3.1　PostgresQL-MC 的日志多核优化

虽然相对传统的日志插入算法，文献[16]提出将锁的持有与日志缓冲区的填充解耦合的方法具有很大优势，但是计算日志记录在日志缓冲区所占长度的操作仍不可避免的需要串行化执行. 在竞争激烈的应用中，这也会给系统的响应时间造成很大延迟. 中国人民大学开发的 PostgreSQL-MC，即针对多核处理器优化的 PostgreSQL，使用了并行化的日志填充方案. PostgreSQL-MC 将日志缓冲区分成若干不相交区域，日志填充分别在不同的区域并行执行. 每个日志记录在插入到某个日志缓冲区之前需要获取一个时间戳. 这个时间戳保证了日志重放顺序与对数据库操作的顺序一致. PostgreSQL-MC 中系统日志检查点需遵循完整性原则. 现在给出完整系统检查点的定义.

定义 1.　多路日志系统 S 具有 n 个日志缓冲区：$L_1, L_2, \cdots, L_n, ck_i^{t_u}$ 对应第 i 路日志缓冲区在 t_u 时刻创建的检查点. 如果存在 $ck_1^{t_u}, ck_2^{t_v}, \cdots, ck_n^{t_w}$，其中，$t_u = t_v = \cdots = t_w$，则称这是一个完整系统检查点.

如图 3 所示，由虚线包围的 3 个检查点是同一逻辑时刻创建，属于一个完整的系统检查点. 系统在某时间点 t 崩溃，而第 2 路日志缓冲的检查点没来得及创建，那么这个系统检查点是不完整的. 故障恢复系统需要反向扫描日志文件寻找最近完整系统检查点 SCK. PostgreSQL-MC 重做(redo)例程从 SCK 上每个日志检查点 ck_i 开始挑选具有最小时间戳的日志记录进行重放. 为了避免日志空洞，进行重放的

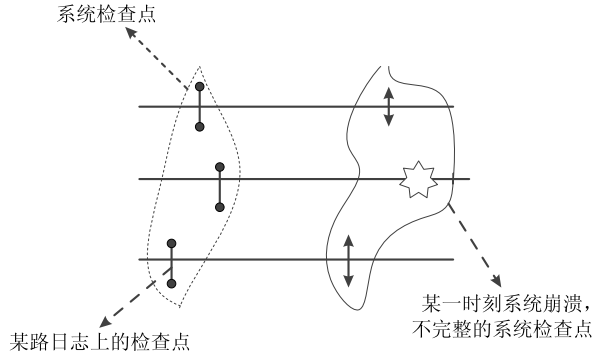


图 3　三路日志检查点创建

连续两个日志记录 $r_i^{t_u}$，$r_j^{t_v}$ 的时间戳需要满足：

$$t_u = t_v + 1.$$

我们将 PostgreSQL-MC 与 PostgreSQL9.2、PostgreSQL9.4 开发版进行比较(PostgreSQL9.4 还没有正式版，之所以选择 PostgreSQL9.4 是因为这个版本的 PostgreSQL 对日志系统进行了优化)．实验所用的硬件平台参数如表 2 所示．实验平台的操作系统为 Linux 3.2.0．利用 TPC-B 以及我们开发的 Micro-benchmark 对系统进行测试．为了消除磁盘 I/O 的瓶颈，数据集全部放入内存文件系统 *tempfs*．Micro-benchmark 由一张表构成，表模式为(*Id*：*int primary key*；*Info*：*text*)．为了增加日志子系统临界区的压力，Micro-benchmark 的测试语句为 100%更新类型．数据库的连接数为 CPU 核数的两倍．

表 2 实验所用硬件平台参数

硬件名称	参数
CPU	Intel Xeon CPU E7-4830
CPU Sockets	4 sockets
Hardware Thread	32 (No Hyper Threading Support)
Clock Speed	2.1GHz
L1 D-Cache	32KB(per core)
L1 I-Cache	32KB(per core)
L2 Cache	256KB(per core)
L3 Cache	24MB(per socket)
Memory	256GB DDR3 1333MHz

PostgreSQL9.2 的日志填充技术如图 2(B)所示．日志记录的插入需要串行化进行．本文测试所用 PostgreSQL9.4 是开发版的．PostgreSQL9.4 的日志填充技术如图 2(D)所示．将日志的填充与获取临界区的锁并行化．PostgreSQL9.4 将日志填充的临界区代码量从 PostgreSQL9.2 的几百行缩减到只有 5 行，即获取临界区互斥变量以后只是简单地计算该日志记录所占的空间，然后即可释放互斥变量．实验结果如图 4、图 5 所示．在一个 CPU 核数的情况下，3 个系统的吞吐率差别不大．在 16 个 CPU 核数的情况下，PostgreSQL9.4 的吞吐率在不同的测试标准上要比 PostgreSQL9.2 的吞吐率分别高 6%(TPC-B)与 30%(Micro-benchmark)；在 32 核的情况下 PostgreSQL9.4 的吞吐率要比 PostgreSQL9.2 的吞吐率分别高 67%(TPC-B)与 40%(Micro-benchmark)．在 TPC-B 的测试下，PostgreSQL-MC 的吞吐率在 16 核与 32 核的情况下比 PostgreSQL9.2 的吞吐率分别高 12%与 100%；在 Micro-benchmark 的测试下，PostgreSQL-MC 的吞吐率在 16 核与 32 核的情况下比 PostgreSQL9.2 的吞吐率分别高 40%与 60%．PostgreSQL-MC 的优化效果比 PostgreSQL9.4 开发版好．

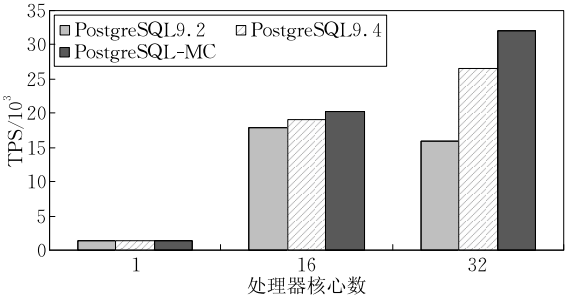


图 4 不同核数下，利用 TPC-B 测试 3 个系统的吞吐率

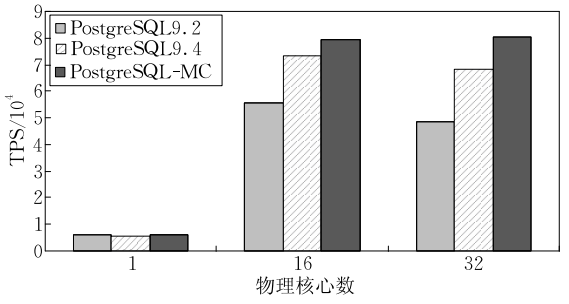


图 5 不同核数下，利用 Micro-benchmark 测试 3 个系统的吞吐率

值得一提的是随着像固态硬盘的新闪存产品开始进入主流的计算机市场，闪存技术被认为是磁盘技术的替代．最近也有相关研究利用闪存技术提高数据库系统性能[19-25]．文献[24]的研究表明固态硬盘可以显著提高事务写日志的性能．Chen[25]则主要关注以下 4 点：(1)突破带宽限制，利用多闪存设备，并行化 I/O 提高刷新日志操作性能；(2)解决由于设备擦除操作而引起的不同的写延迟；(3)高效的系统恢复处理；(4)将闪存设备与磁盘设备结合起来获得更好的日志操作和系统恢复性能．文献[25]的目标是廉价、高效和简单的日志处理方法．利用闪存技术提升 DBMS 日志的性能主要是加快刷新日志的性能，而不是减缓日志模块临界区的竞争．

4 多核下的缓冲区管理

在数据库系统的缓冲区管理器中，存在缓冲块索引(通常由哈希表构成)和 LRU 链等关键数据结构．这些数据结构的访问具有排他性．即便在 I/O 带宽无限大的情况下，多核处理器对这些数据结构的访问也必须是串行的，这造成了严重的扩展性瓶颈．因此，缓冲区管理器是多核扩展优化的重点对象．

已有的关于数据库系统缓冲区研究都集中在如何提高缓冲区命中率[26-27]．Tsuei 等人[28]在对称多处理机上设计实验以探讨数据库大小、缓冲区大小和

处理器数目对数据库性能造成的影响，尤其是对缓冲区命中率和系统吞吐率的影响.他们用 TPC-C 负载研究缓冲区大小与性能的关系，发现比数据库稍小的缓冲区大小就已经足够.同时，他们也给出规则：10%～15%的数据库大小就已经贡献了 80%的缓冲区命中率.实际上 OSDL 的 6.8GB DBT-1 和 5.6GB DBT-2 测试数据集(分别由 TPC-W 和 TPC-C 演变而来)在装备 256 MB 缓冲区的 PostgreSQL 8.2 上可以产生 95%的缓冲区命中率.

为了减少 LRU 链上的竞争，ADABAS[29]将缓冲区池划分成几个物理区域，每个物理区域都有自己的 LRU 链.这种方法会减少缓冲区命中率，尤其是数据偏斜比较严重的时候.同时，这种方法也不适合大规模多线程并发的环境.因为在这种场景下，它按照处理器数目细分缓冲区会造成缓冲区命中率下降，同时竞争的减少却不明显.另外，文献[29]没有讨论缓冲区的划分与缓冲区命中率的变化的关系.

Bp-wrapper[30]在缓冲区管理上应用批量处理技术.这种批量处理技术可以归类为延迟同步技术，即刻返回逻辑操作结果.这种批量处理技术可以分摊申请锁而带来的开销.Bp-Wrapper[30]可以工作在任何置换算法中，并且在缓冲命中的时候消除竞争.但是，根据文献[30]的实验结果，Bp-Wrapper 的优点在 LRU 算法族不能充分发挥，并且提高 CLOCK 算法类的吞吐率上效果不明显，因为 Bp-Wrapper 不能消除缓冲区未命中的时候锁的竞争.如果其中一个并发访问线程遇到缓冲区未命中，需要申请加锁，Bp-wrapper 必须对阻塞操作进行排序.在缓冲区命中的情况下，CLOCK 算法也不需要申请加锁.因此，Bp-wrapper 不能改善当前 PostgreSQL 的缓冲区管理策略.

大部分缓冲区并发控制问题的解决都是依赖个人开发者的经验知识.关于缓冲区的并发控制还没有被密集地讨论的一个原因是大规模的多处理器还未普及，而且目前的主要关注点仍在如何提高缓冲区命中率以减少 I/O.然而，为请求的磁盘页面钉住缓冲区页面的 bufferfix 操作却不受限于 I/O[31].尽管 bufferfix 操作导致页面置换时将脏页写出会发生磁盘 I/O，但是现代 DBMS 通过预先刷新脏页面和选择非脏页面作为置换页来减少此类 I/O.这就意味着如果内存空间足够且有大量的缓冲区池可用，那么就可以最大程度地减少由 bufferfix 操作带来的页面置换次数.在这种情况下 bufferfix 操作就变成受限于 CPU 的任务.因此，缓冲区管理中针对 CPU 上的可扩展性就成为多核处理时代的主要问题.实际上，fix 和 unfix 操作是数据库中调用最频繁的基本操作.高效的 fix 和 unfix 操作尤其重要，因为它们频繁导致临界区上的竞争[31].

数据库每访问一次缓冲区管理器模块，临界区上的操作都需要获得互斥变量，在多处理器环境下有可能出现“互斥变量抖动”现象[32].此外，对锁的大量访问请求有可能导致护卫现象，它出现在当持有锁的线程由于某种中断例如缺页而被挂起的时候.然后，申请锁的其他线程就会排队，不能继续执行.即使锁在后面被释放，清空队列也需要花费一定的时间.PostgreSQL 8.2 和 MySQL 5.0 通过使用细粒度的锁来减缓在缓冲池上的锁竞争问题.它们采用称为“锁分段”的传统方法来提高哈希表上的并发.Yui 等人[31]采用更激进的方法在多核环境下进行同步.他们利用现有的非阻塞哈希表①设计非阻塞的置换算法 Nb-GCLOCK.因此，缓冲区页面的查找和分配都不需要申请锁因而也就避免了旋转锁带来的问题②[33].Nb-GCLOCK 置换算法是 Generalized Clock[34]置换算法的变种.因此它具有 Generalized Clock 置换算法的以下优点：(1)低开销和高并发；(2)由并发访问共享变量而引起竞争的概率很低；这些良好的性质和性能已经被规范地证明和确定[34-35].文献[31]的作者在 Apache 的开源数据系统 Derby 上实现了 Nb-GClock 算法.在实验中，Nb-GClock 的扩展性可以达到 64 核，消除了由 bufferfix 操作时需要加锁而引起的扩展性差等问题.

4.1　PostgreSQL-MC 的缓冲区多核优化探索

为了消除其他模块，例如锁管理器、恢复子系统等对查找数据库缓冲区的瓶颈带来的干扰，我们选取了只读事务对系统进行测试.通过大量的实验分析以及源码的阅读，我们发现缓冲区管理器的扩展瓶颈集中在缓冲区空闲链表(Buffer Free List)以及哈希表管理.PostgreSQL 的数据页访问算法如图 6 所示：每当事务需要访问一个数据页，首先在哈希表中查找.若找到，说明要访问的数据块已经在缓冲池中，直接返回；若没有找到，则需要从缓冲池中寻找一个空闲页面来装入要访问的数据块.系统在缓冲区空闲链表中查找是否存在空闲页面；如果存在，那么直接返回该空闲页面即可；若不存在，则需要采用时钟算法为需要访问的数据块淘汰页面.这个过程

① http://sourceforge.net/projects/high-scale-lib
② https://software.intel.com/sites/default/files/m/d/4/1/d/8/17689_w_spinlock.pdf

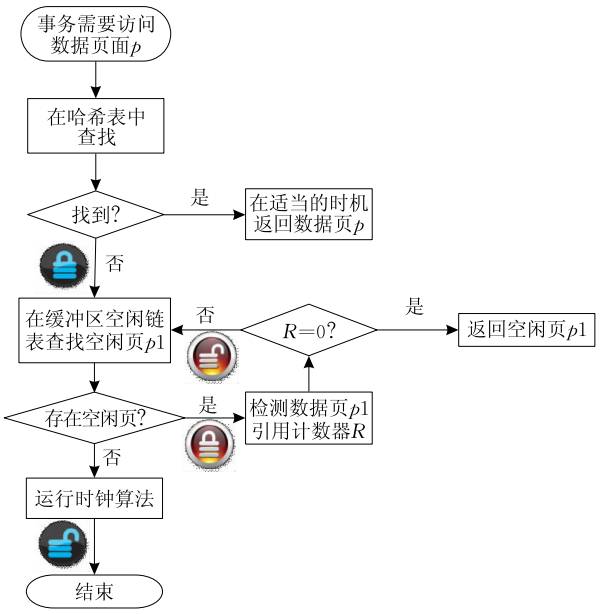


图 6 PostgreSQL 访问数据页流程图

中有两个需要串行化执行的点:(1)缓冲区空闲链表的访问;(2)将新的数据块插入到哈希表过程中哈希表空闲结点的获取.中国人民大学 PostgreSQL-MC 研究团队探索了以下优化:

(1)缓冲区空闲链表的访问.通过去除缓冲区空闲链表上的轻量级锁,减少事务获取空闲页面的等待时间,增大系统并发度.竞争空闲数据页的事务需要获得旋转锁,第一个获得旋转锁的事务会标记空闲数据页的引用计数器,防止其他事务使用该页面.

因此去除缓冲区空闲链表上的轻量级锁不会导致出现一个空闲页被多个事务使用情况.此外,通过检测缓冲区空闲链表是否由于并发而遭受破坏,我们保证每次测试不会受到缓冲区空闲链表的干扰.

(2)时钟置换算法的修改.每个事务随机挑选起始位置开始时钟置换算法,事务之间并行执行各自的时钟算法.

(3)哈希表空闲节点的获取.虽然 PostgreSQL8.2 以后采用称为"锁分段"的方法来提高哈希表上的并发,但是哈希表空闲结点的获取还是只有一个入口,从而成为系统的瓶颈.我们采取与"锁分段"类似的做法,将哈希表的空闲结点的入口分为多个,分别管理.以下的实验依次采取这些优化.实验的硬件配置见 3.1 节.图 7 给出不同核数下系统的吞吐率.数据库的连接数固定在 100,测试时间为 2 min.我们利用数据仓库星型模型测试基准 SSBM(Star Scheme Benchmark)生成测试数据(扩展因子 $SF=1$),只保留事实表 Lineorder 的整型字段.事实表 Lineorder 的大小约为 300 MB,PostgreSQL9.2 的缓冲区为 32 MB.查询语句为在事实表上简单的 SELECT-FROM-WHERE 点查询.PostgreSQL9.2 系统吞吐率的最大值出现在 8 核左右.随着核数的增加数据库系统不能再利用多余的硬件资源.不管在 PostgreSQL 的基础上去除缓冲区空闲链表的轻量级锁以及并行化时钟置换算法还是调节哈希表分段锁的个数,系统的扩展性基本都停留在 8 核左右.随着核数的增加,系统的性能反而有所下降.利用 VTune(Intel 公司一个性能分析软件)观测,此时系统的瓶颈出现在哈希表的插入以及删除上.接着,我们将哈希表空闲结点的获取入口由一个扩展为多个.此时 系统的扩展性可以达到 32 核.从以上实验结果可以得到如下结论:缓冲区的瓶颈出现在缓冲池空闲页面的管理和哈希表上.PostgreSQL-MC 研究团队未来的工作方向是开发面向多核的生产者-消费者算法管理缓冲池以及适合多核多 CPU 平台的哈希表.

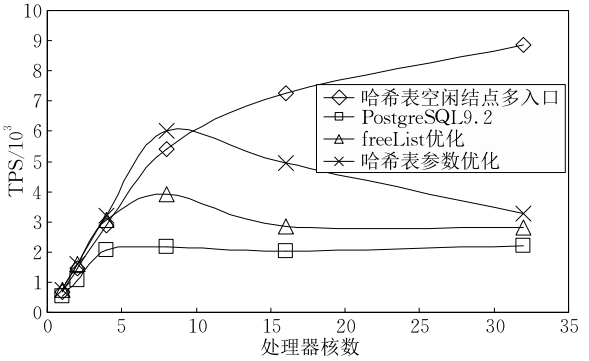


图 7 不同的缓冲区管理策略下,系统的扩展性

不同于 DBMS 的缓冲区研究,OS 中内存管理的研究较早前就关注多核环境下的优化问题[36-42].文献[36-39]利用页面染色的方法划分高速缓存以解决高速缓存争用的问题,从而更好地管理 OS 缓冲区页面的分配.文献[40]认为对内存控制器的竞争对系统的性能造成很大影响,改善内存子系统性能需要设计公平的内存器.文献[41-42]则设计了高速缓存竞争敏感的调度器以最小化资源竞争.

以上关于 DBMS 的缓冲区在多核环境下的研究仅仅局限于 DBMS 缓冲区层面算法上的修改.而现实 DBMS 系统的缓冲区管理大多是在操作系统的缓冲区基础上进行定制.因此,研究多核处理器下 DBMS 的缓冲区的优化需要考虑操作系统的缓冲区修改,例如文献[43].我们认为 DBMS 可以借鉴这些思想结合硬件特性优化其缓冲区的设计,提出更好的缓冲区管理算法.

5　多核下索引并发控制

B 树[44-45]作为数据库管理系统中的默认索引，广泛用于数据库应用. 在事务处理方面作出过杰出贡献的 Jim Gray 曾经说："B 树是迄今为止在数据库和文件系统中最重要的存取路径数据结构". 与其他索引相同，B 树的功能是将搜索关键字映射到其相关信息. 除了提供精确匹配查找，B 树也支持范围查询，并且为基于排序的查询执行算法，例如合并连接算法，省略了显式的排序操作. 然而，B 树上的并发控制在多核扩展上表现为明显的瓶颈. B 树利用两种不同形式锁，也就是闩与锁，分别保护 B 的物理结构和逻辑内容. 闩与锁的区别如表 3 所示. 由于 B 树节点与数据呈一对多的关系，对 B 树节点加锁往往是粗粒度的，会导致并发度降低. 此外，闩锁对应的临界区也是阻碍多核扩展的一个因素.

表 3　闩锁与锁的区别

	锁	闩
隔离内容	用户事务	线程
保护内容	数据库数据	内存数据结构
存在方式	整个事务	临界区
模式	排它锁、共享锁、意向锁等	读、写、更新
死锁	死锁检测 & 消除	避免死锁
死锁处理	终止事务	通过编码原则避免死锁
存放位置	锁管理器的哈希表	被保护的数据结构

5.1　B 树物理结构的保护

Blink[46]树是一种放宽 B 树结构限制的访问方法. 它将结点分裂划分成两个独立的阶段. 每一个结点都可以有一个高位保护值和指向其右邻居的指针. 从根到叶子节点的遍历中，每到达一个结点就必须将搜索值和该结点的高位保护值相比较. 如果高位保护值小于搜索值，必须沿着该结点指向右邻居的指针继续查找. 分裂节点的第 1 步是创建结点的高位保护值和右邻居结点；第 2 步是独立的步骤，将高位保护值放置在父节点. 第 2 个步骤可以作为其后从根结点到叶子节点遍历的附加操作，应该尽早完成. 这个操作也有可能被系统重启或系统崩溃所推迟，但是它们不会造成 Blink 树出现数据丢失和不一致性情况. Blink 树的优点是结点的分配和引进是一个局部操作，只对溢出结点有影响，因此并发度高；缺点是搜索不够高效，并且 Blink 树的一致性验证会变得更复杂. 在值频繁插入的情况下，需要措施保证指向邻居结点的链表不会过长. B 树结点的移除也可以借鉴这个方法以提高并发度[47]. 第 1 步是创建指向邻居结点的指针. 第 2 步在父节点中将指向删除结点的指针擦除. 第 3 步合并结点. Jaluta 等人[48]通对 Blink 做了进一步改进，将分裂操作限制在由合适的父结点指向的结点避免了频繁插入带来的长链表. 文献[49]提出避免闩锁的一种 B 树——Bw-tree. Bw-tree 针对多核和 SSD 硬盘的新硬件平台而设计的. Bw-tree 利用 Blink 树的原子分裂技术，结点分裂由两个独立部分组成. Bw-tree 树中状态的改变都是利用原子交换指令完成，因此可以避免闩锁，而且不会阻塞. 唯一使得 Bw-tree 阻塞的情况就是缺页. 由于 Bw-tree 消除闩锁的特性，因此一些事务不可避免地会遇到竞争而导致更新页面状态失败. 此外，Bw-tree 中页面的概念是虚拟的，任何对页面的更新都首先必须创建一个 Δ 记录，然后附加在虚拟页面上. 过长的 Δ 记录链表会降低搜索的性能，需要额外的开销处理这些链表. Foster B-tree[50]是针对多核平台和类似 SSD 等非易失存储器而提出的另一 B-tree 变种. Foster B-tree 设计者们的目标是：(1) 设计最小化并发控制需求的数据结构；(2) 方便数据节点往新的存储上迁移；(3) 支持连续和复杂的自我检测功能. Foster B-tree 在处理更新操作时采用类似 B-link 树的方法，每次只需对两个索引节点加闩锁，以最大化并发操作. Foster B-tree 与 B-link 树的不同在于为了方便数据的迁移 Foster B-tree 的索引节点最终只有一个指向它的指针，也就是消除了同一层节点之间的指针，只保留指向它的父节点指针. Foster B-tree 采用对称保护码允许在从根到叶子节点的遍历过程中进行有效性验证. 设计上，Foster B-tree 集合了 B-link 树[46]、对称保护码以及写优化 B-tree[51]的特点，最大程度地避免它们的缺点.

5.2　B 树逻辑内容的保护

锁将读事务与写事务隔离开. 为了保证一致性，事务持有读锁直到结束. 为了保证事务终止时能够撤销所有对数据库的修改，事务总是持有写锁直到结束. 此外，可串行化不仅要求对存在的数据加锁，也要求对不存在的数据加锁. 例如对于一个 top-k 查询，第二执行的相同的 top-k 查询必须得到相同的结果，而且任何影响结果集的插入操作都应该被阻止. 对 B 树而言，这通常由关键字范围封锁来实现. Mohan 的 ARIES/KVL[52]直接区分了锁与闩的区别(见表 3)，采用一个关键字-值锁(Key Value Lock，KVL)覆盖一个区间和区间上界的关键. 对非单一索引而言，作用在某个关键字上的意向锁对

具有相同值的所有行都有效.然而将行 ID 包含在锁 ID 上的设计却不用区分单一和非单一索引.Gray[15]描述 B 树上的行级锁就是基于关键字-值锁.Lomet 的关键字范围锁[53]尝试将层次锁与多粒度锁应用在值和半开闭区间,但是需要额外的锁模式,范围插入模式,以获得满意的并发度.Graefe[54]将传统的层次锁更严格地应用在关键字和关键字的区间,并且在插入和删除操作上使用了伪删除记录.这种设计可以减少特例情况,取得更高并发度.最近也有研究将增量锁与 B 树上的关键字范围锁结合起来以取得最大并发度[49].

锁的设计不仅考虑保护数据库一致性与隔离性,还需要考虑锁协议的简单性与最大化系统的并发度.相关的研究者一直在努力设计简单的锁协议,同时最大化系统并发度.然而,锁的维护、申请和释放都会给系统造成不小的开销.随着虚拟技术的发展使得在多核处理器部署无共享系统成为了可能[55].无共享系统从物理上划分数据库,一个极端的例子 H-Store[56].它通过嵌入多个单线程的执行引擎.各个执行引擎之间并行执行,互不干扰,单个执行引擎的事务串行执行从而避免并发控制.

6 研究展望

现在,数据库研究社区已经开始重视多核架构对数据库系统产生的影响[57-58].为了克服当前数据库系统共同的弱点,也就是系统随着处理器个数或核数的增长其性能难以提升,许多学者提出了不同的解决方案.针对上述多核环境下数据库中的锁管理器、缓冲区管理器、日志以及索引遇到的问题总结起来,解决方案可以分成两类:分布式与共享.在分布式架构下,将多核与多处理器当做分布式系统对待;而共享系统下,则需要解决通信和资源竞争的问题.

分布式系统解决方案.对分布式数据库的研究在 20 世纪 80 年代末 90 年代初就已经开展[59-60].早期分布式数据库系统的许多技术可以用于改善集中式数据库系统在多核平台下的扩展性.Salomie 等人[61]开发的 Multimed 系统将多个数据库实例(PostgreSQL 或者 MySQL)部署在多核平台上.运行这些数据库实例的核不相交.主节点负责接收并分发只读查询请求,同时执行更新请求.然后主节点将更新结果按异步方式分发到卫星节点.H-Store[56]将数据在多个节点和核上水平划分,然后事务在每个核上以单线程的方式串行执行.因此,每个数据划分上都不需要同步访问.类似传统分布式数据库,这些系统需要工作负载感知的数据划分策略以减少划分间的通信.数据偏斜或者是数据访问与数据划分的模式不对应都会造成此类系统性能上的问题.

共享系统.一个共享系统的例子是由 Johnson 等人[10]开发的 shore-MT.在分析了现有数据存储系统的瓶颈之后,他们通过优化锁、闩锁和同步机制从而减少竞争得到一个多线程的、可扩展的系统.物理逻辑组合划分是介于分布式系统和共享系统之间的一个解决方案[62-63]:数据仍然是共享的,但是利用一个多根 B-tree 划分数据,避免昂贵的对数据页加闩锁的操作.DORA 系统[62]将各个线程绑定到不相交的数据集,根据事务访问的数据把事务划分成更小的动作序列.其中,每个线程都拥有各自的锁机制来控制对所属数据的访问,减轻集中式锁管理带来的弊端.此外,近年来涌现的一些优秀的内存数据库例如微软的 Hekaton[64]和 SAP 的 HANA[65]也极力优化其查询处理引擎以及主要模块的临界区以便能充分利用多核平台的优势.

7 总 结

在这篇综述中,我们主要讨论新的 CPU 架构下的数据库系统优化技术以及中国人民大学在开源数据库 PostgreSQL 上提出的多核处理器优化技术.现在,多核处理器提供的更高的并行性对数据库系统带来的挑战是明显的.数据库系统必须经过彻底的改进,才能利用不断增长的硬件上下文和可用的资源,从而进一步改善性能.

参 考 文 献

[1] Wentzlaff D, Agarwal A. Factored operating systems (fos): The case for a scalable operating system for multicores. Operating Systems Review, 2009, 43(2): 76-85

[2] Wentzlaff D, Gruenwald III C, Beckmann N, et al. An operating system for multicore and clouds: Mechanisms and implementation//Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis, USA, 2010: 3-14

[3] Baumann A, Peter S, Schüpbach A, et al. Your computer is already a distributed system. Why isn't your OS?//Proceedings of the 12th Workshop on Hot Topics in Operating Systems. Verità, Switzerland, 2009: 12

[4] Boyd-Wickizer S, Clements A T, Mao Y, et al. An analysis of linux scalability to many cores//Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation. Vancouver, Canada, 2010: 1-8

[5] Cieslewicz J, Ross K A. Adaptive aggregation on chip multiprocessors//Proceedings of the 33rd Very Large Data Bases. University of Vienna, Austria, 2007: 339-350

[6] Johnson R, Athanassoulis M, Stoica R, Ailamaki A. A new look at the roles of spinning and blocking//Proceedings of the 5th International Workshop on Data Management on New Hardware. Providence, USA, 2009: 21-26

[7] Unterbrunner P, Giannikis G, Alonso G, et al. Predictable performance for unpredictable workloads. Proceedings of the VLDB Endowment, 2009, 2(1): 706-717

[8] Joshi A M. Adaptive locking strategies in a multi-node data sharing environment//Proceedings of the 17th International Conference on Very Large Data Bases. Barcelona, Spain, 1991: 181-191

[9] Johnson R, Pandis I, Ailamaki A. Improving OLTP scalability using speculative lock inheritance. Proceedings of the VLDB Endowment, 2009, 2(1): 479-489

[10] Johnson R, Pandis I, Hardavellas N, et al. Shore-MT: A scalable storage manager for the multicore era//Proceedings of the 12th International Conference on Extending Database Technology. Saint Petersburg, Russia, 2009: 24-35

[11] Horikawa T. Latch-free data structures for DBMS-Design, implementation, and evaluation//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 409-420

[12] Jung H, Han H, Fekete A D, et al. A scalable lock manager for multicores//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 73-84

[13] Ren K, Thomson A, Abadi D J. Lightweight locking for main memory database systems. Proceedings of the VLDB Endowment, 2012, 6(2): 145-156

[14] Mohan C, Haderle D, Lindsay B, et al. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems, 1992, 17(1): 94-162

[15] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. San Francisco: Morgan Kaufmann, 1993

[16] Johnson R, Pandis I, Stoica R, et al. Scalability of write-ahead logging on multicore and multisocket hardware. Proceedings of the VLDB Endowment, 2012, 21(1): 239-263

[17] Helland P, Sammer H, Lyon J, et al. Group commit timers and high-volume transaction systems//Proceedings of the 2nd International Workshop on High Performance Transaction Systems, Asilomar Conference Center. Pacific Grove, USA, 1987: 301-329

[18] DeWitt D J, Katz R H, Olken F, et al. Implementation techniques for main memory database systems//Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. Boston, USA, 1984: 1-8

[19] Bouganim L, Jónsson B T, Bonnet P. uFLIP: Understanding flash io patterns//Proceedings of the 4th Biennial Conference on Innovative Data Systems Research. Asilomar, USA, 2009

[20] Caulfield A M, Grupp L M, Swanson S. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009: 217-228

[21] Graefe G. The five-minute rule twenty years later, and how flash memory changes the rules//Proceedings of the Workshop on Data Management on New Hardware. Beijing, China, 2007: 6

[22] Koltsidas I, Viglas S. Flashing up the storage layer. Proceedings of VLDB Endowment, 2008, 1(1): 514-525

[23] Lee S W, Moon B. Design of flash-based DBMS: An in-page logging approach//Proceedings of the ACM SIGMOD International Conference on Management of Data. Beijing, China, 2007: 55-66

[24] Lee S W, Moon B, Park C, et al. A case for flash memory ssd in enterprise database applications//Proceedings of the ACM SIGMOD International Conference on Management of Data. Vancouver, Canada, 2008: 1075-1086

[25] Chen S. FlashLogging: Exploiting flash devices for synchronous logging performance//Proceedings of the ACM SIGMOD International Conference on Management of Data. Providence, USA, 2009: 73-86

[26] Nicola V F, Dan A. Dias D M. Analysis of the generalized clock buffer replacement scheme for database transaction processing. Sigmetrics Performance Evaluation Review, 1992, 20(1): 35-46

[27] Johnson T, Shasha D. 2Q: A low overhead high performance buffer management replacement algorithm//Proceedings of the 20th International Conference on Very Large Data Bases. Santiago, Chile, 1994: 439-450

[28] Tsuei T F, Packer A N, Ko K T. Database buffer size investigation for OLTP workloads//Proceedings of the ACM SIGMOD International Conference on Management of Data. Tucson, USA, 1997: 112-122

[29] Schöning H. The ADABAS buffer pool manager//Proceedings of the 24th International Conference on Very Large Data Bases. New York, USA, 1998: 675-679

[30] Ding X, Jiang S, Zhang X. BP-wrapper: A system framework making any replacement algorithms (almost) lock contention free//Proceedings of the 25th International Conference on Data Engineering. Shanghai, China, 2009: 369-380

[31] Yui M, Miyazaki J, Uemura S, Yamana H. Nb-GCLOCK: A non-blocking buffer managementBased on the generalized CLOCK//Proceedings of the 26th International Conference on Data Engineering. Long Beach, USA, 2010: 745-756

[32] Blasgen M, Gray J, Mitoma M, Price T. The convoy phenomenon. Operating Systems Review, 1979, 13(2): 20-25

[33] Zhou J, Cieslewicz J, Ross K A, Shah M. Improving database performance on simultaneous multithreading processors//Proceedings of the 31st International Conference on Very Large Data Bases. Trondheim, Norway, 2005: 49-60

[34] Nicola V F, Dan A, Dias D M. Analysis of the generalized clock buffer replacement scheme for database transaction processing. ACM SIGMETRICS Performance Evaluation Review, 1992, 20(1): 35-46

[35] Smith A J. Sequentiality and prefetching in database systems. ACM Transactions on Database Systems, 1978, 3(3): 223-247

[36] Ding X, Wang K, Zhang X. SRM-Buffer: An OS buffer management technique to prevent last level cache from thrashing in multicores//Proceedings of the 6th European conference on Computer Systems. Salzburg, Austria, 2011: 243-256

[37] Tam D K, Azimi R, Soares L B, Stumm M. Rapidmrc: Approximating *l*2 miss rate curves on commodity systems for online optimizations//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009: 121-132

[38] Zhang X, Dwarkads S, Shen K. Towards practical page coloring-based multicorecache management//Proceedings of the 2009 EuroSys Conference. Nuremberg, Germany, 2009: 89-102

[39] Lin J, Lu Q, Ding X, et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems//Proceedings of the 14th International Conference on High-Performance Computer Architecture, Salt Lake City, USA, 2008: 367-378

[40] Moscibroda T, Mutlu O. Memory performance attacks: Denial of memory servicein multicore systems//Proceedings of the USENIX Security Symposium. Boston, USA, 2007: 1-18

[41] Fedorova A, Seltzer M I, Smith M D. Improving performance isolation on chip multiprocessors via an operating system scheduler//Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. Brasov, Romania, 2007: 25-38

[42] Blagodurov S, Zhuravlev S, Fedorova A. Contention-aware scheduling on multicore systems. ACM Transactions on Computer Systems, 2010, 28(4): 8

[43] Lee R, Ding X, Chen F, Lu Q, Zhang X. MCC-DB: Minimizing cache conflicts in multi-core processors for databases. Proceedings of the VLDB Endowment, 2008, 2(1): 373-384

[44] Bayer B, McCreight E M. Organization and maintenance of large ordered indices. Acta Informatica, 1972, 1(3): 173-189

[45] Bayer R. The universal B-tree for multidimensional indexing: General concepts//Proceedings of the Worldwide Computing and Its Applications. Tsukuba, Japan, 1997: 198-209

[46] Lehman P L, Yao S B. Efficient locking for concurrent operations on B-trees. ACM Transactions on Database Systems, 1981: 6(4): 650-670

[47] Lomet D B. Simple, robust and highly concurrent B-trees with node deletion//Proceedings of the 20th International Conference on Data Engineering. Boston, USA, 2004: 18-28

[48] Jaluta I, Sippu S, Soisalon-Soininen E. Concurrency control and recovery for balanced B-link trees. VLDB Journal, 2005, 14(2): 257-277

[49] Levandoski J J, Lomet D B, Sengupta S. The Bw-tree: A B-tree for new hardware platforms//Proceedings of the 29th IEEE International Conference on Data Engineering. Brisbane, Australia, 2013: 302-313

[50] Graefe G, Kimura H, Kuno H A. Foster B-tree. ACM Transactions on Database Systems, 2012, 37(3): 17-29

[51] Graefe G. Write-optimized B-trees//Proceedings of the 30th International Conference on Very Large Data Bases. Toronto, Canada, 2004: 672-683

[52] Mohan C. ARIES/KVL: A key-value locking method for concurrency control of multi action transactions operating on B-tree indexes//Proceedings of the 16th International Conference on Very Large Data Bases. Brisbane, Australia, 1990: 392-405

[53] Lomet D B. Simple, robust and highly concurrent B-trees with node deletion//Proceedings of the 20th International Conference on Data Engineering. Boston, USA, 2004: 18-28

[54] Graefe G. Hierarchical locking in B-tree indexes//Proceedings of the Datenbanksysteme in Business, Technologie und Web. Aachen, Germany, 2007: 18-42

[55] Bugnion E, et al. Disco: Running commodity operating systems on scalable multiprocessors. ACM Transactions on Computer Systems, 1997, 15(4): 412-447

[56] Stonebraker M, Madden S, Abadi D J, et al. The end of an architectural era (It's time for a complete rewrite)//Proceedings of the 33rd International Conference on Very Large Data Bases. Vienna, Austria, 2007: 1150-1160

[57] Ailamaki A, Dewitt D J, Hill M D, Wood D A. DBMSs on a modern processor: Where does time go?//Proceedings of the 25th International Conference on Very Large Data Bases. Edinburgh, Scottland, 1999: 266-277

[58] Hardavellas N, Pandis I, Johnson R, et al. Database servers on chip multiprocessors: Limitations and opportunities//Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research. Asilomar, USA, 2007: 79-87

[59] Dewitt D J, Ghandeharizadeh S, Schneider D, et al. The Gamma database machine project. IEEE Transactions on Knowledge and Data Engineering, 1990, 2(1): 44-62

[60] Apers P M G, van den Berg C A, Flokstra J, et al. PRISMA/DB: A parallel, main memory relational DBMS. Knowledge and Data Engineering, 1992, 4(6): 541-554

[61] Salomie T, Subasu I E, Giceva J, Alonso G. Database engines on multicores, why parallelize when you can distribute?//Proceedings of the 6th European conference on Computer Systems. Salzburg, Austria, 2011: 17-30

[62] Pandis I, Johnson R, Hardavellas N, Ailamaki A. Data-oriented transaction execution. Proceedings of the VLDB Endowment, 2010, 3(1): 928-939

[63] Pandis I, Tözün P, Johnson R, Ailamaki A. PLP: Page latch-free shared-everything OLTP. Proceedings of the VLDB Endowment, 2011, 4(10): 610-621

[64] Diaconu C, et al. Hekaton: SQL server's memory-optimized OLTP engine//Proceedings of the ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 1243-1254

[65] Sikka V, et al. Efficient transaction processing in SAP HANA database—The end of a column store myth// Proceedings of the ACM SIGMOD International Conference on Management of Data. Scottsdale, USA, 2012: 731-742

ZHU Yue-An, born in 1983, Ph. D. His current research interests include IR and high performance database.

ZHOU Xuan, born in 1979, Ph. D. His current research interests include IR and high performance database

ZHANG Yan-Song, born in 1973, Ph. D. His current research interests include main-memory database, OLAP and high performance database.

ZHOU Ming, born in 1990, M. S. His current research interest focuses on main-memory database.

NIU Jia, born in 1989, M. S. Her current research interest focuses on High-performance database.

WANG Shan, born in 1944, professor, Ph. D. supervisor. Her current research interests include data warehouse, high performance database and knowledge engineering.

Background

The emerge of multi-socket and multi-core hardware brings the new challenges to database system, since the performance is affected by software parallelism greatly. Although traditional database accepts simultaneous requests, a lot of synchronization primitives lie within it, which serializes the execution. The recent research shows that as the increase of the CPU cores, the database performance will drop on the contrary in the face of some workloads. Now, database community pays more and more attention to this problem, and how to make good use of new hardware is main topic of development of database. There are multitudes of research still on the road. Until recently, the focus is on how to optimize the locking, logging, latch and buffer management in database. This work is to identify the main techniques to promote the performance of database in multi-core era. Renmin University has a long history to develop the high performance database and our team also solve the problem of OLAP in multi-core CPU to some extent. Now, we try to grab opportunities to develop new database, fitting in with new hardware platform.

This work is supported by the National Natural Science Foundation (Grant Nos. 61272138, 61232007), the Basic Research funds in Renmin University of China from the Central Government (Grant Nos. 12XNQ072, 13XNLF01), and the Graduate Science Foundation of Renmin University of China (Grant No. 13XNH216)