并发控制实现方法的比较研究*

萧美阳, 叶晓俊

(清华大学 软件学院 信息系统与工程研究所, 北京 100084)

摘　要: 并发控制是 DBMS 的关键技术,直接影响系统的正确性和性能。通过 PostgreSQL 和 InnoDB 并发控制在底层实现和上层方法两个方面的细致比较,发现 InnoDB 在并发控制方面有较好的性能,值得 PostgreSQL 借鉴。最后基于 PostgreSQL 系统对底层的实现进行改进,并提供了详细的实验数据。

关键词: PostgreSQL; InnoDB; 并发控制; 多版本; 封锁

中图法分类号: TP311.5　　**文献标识码**: A　　**文章编号**: 1001-3695(2006)06-0019-04

Comparative Study on Concurrency Control

XIAO Mei-yang, YE Xiao-jun

(Institute of Information System & Engineering, School of Software, Tsinghua University, Beijing 100084, *China)*

Abstract: Concurrency control is one of key technologies in DBMS, and directly impacts the correctness and performance. This paper compares the two databases PostgreSQL and InnoDB from bottom to top, and discovers that the InnoDB has more advantages in concurrency control than PostgreSQL. Last, the author does some improvement based on PostgreSQL, and then supplies detailed experiment data.

Key words: PostgreSQL; InnoDB; Concurrency Control; Multi-Version; Locking

1　引言

数据库系统是面向多用户的系统,事务并发可以提高系统的吞吐量,所以并发控制在保证系统正确性的同时,机制的好坏也直接影响系统的性能,是 DBMS 的关键技术。

并发控制技术的实现通常依赖于底层的并发控制机制,一般我们称其为同步机制[4]。操作系统提供了多种同步对象,如事件 Event、互斥锁 Mutex 和条件变量 Cond、信号量 Semaphore、读写锁 RWLock、自旋锁 Spinlock 等。不同的数据库系统采用了不同的同步对象,如 PostgreSQL 采用信号量,InnoDB 采用事件或是互斥锁。性能上这些同步对象都存在较大的差别,并适用于不同的场合。

目前提到的并发技术方法多为封锁与时间戳[1]。大多数的商业数据库(DB2, SQL Server)采用了 2PL(两阶段封锁)协议,本文所说到的 InnoDB 也采用了此种方式[6]。该封锁协议保证了并发事务执行的可串行化,在任何读写操作之前都将访问的对象加上共享/排他锁,其结果是读写阻塞。

PostgreSQL 则采用了一种更为有效的方式很好地解决了读写阻塞的问题[5]。在任何操作的读阶段都不加锁,而是读取一个符合自己要求的元组版本,写操作在写阶段则产生一个新的版本,将旧版本维护起来,对于不同的 DBMS 维护旧版本的方式不同。这种并发控制的方式我们通常称其为多版本并发控制(MVCC),是基于时间戳并发控制的一个变种。该方法很大程度上提高了并发度,但是也带来维护多个版本的存储开销。InnoDB 也利用了多版本的方式来协助封锁方式完成并发工作。

本文对 PostgreSQL 和 InnoDB 的同步机制、并发控制方式及其性能进行了比较,并在 PostgreSQL 基础上对同步机制进行了修改,给出了前后的性能对比。

收稿日期: 2005-05-15; 修返日期: 2005-06-22

基金项目: 国家“973”计划资助项目(2004CB719400);国家自然科学基金资助项目(60473077);国家“863”计划资助项目(2003AA413230)

2　PostgreSQL 和 InnoDB 的同步机制

PostgreSQL 和 InnoDB 采用完全不同的两种同步对象来实现底层的同步。

2.1　PostgreSQL 同步实现方法

PostgreSQL 采用传统的信号量,通过对操作系统函数的统一封装实现进程之间的同步。PostgreSQL 封装了两种标准的信号量,分别是 Posix 信号量(包括有名和无名两种类型)和 System V 信号量。当然 PostgreSQL 还采用了信号、消息等通信手段实现进程同步。这里主要讲信号量的同步,因为它是上层 DBMS 并发控制的基础。

PostgreSQL 提供自旋锁 Spinlock 和轻量锁 LWLock 实现共享内存中数据的并发控制。自旋锁主要依赖于计算机硬件对 TAS(Test-And-Set,硬件指令,同 SWAP 一样可以实现临界区的互斥)指令的支持。如果计算机硬件恰好不支持,而操作系统又不支持互斥锁等同步对象,则 PostgreSQL 基于上述的信号量模拟实现自旋锁。轻量锁依赖自旋锁保护变量的方式实现了多个读一个写,比自旋锁更为有效,所以它的使用也是 PostgreSQL 中最为频繁的。

信号量除了模拟自旋锁,主要的作用是实现 PostgreSQL 内各服务进程的阻塞唤醒。PostgreSQL 中每一个进程对应一个进程结构体 PGPROC,该结构体中有一个重要的域——信号量。服务进程运行过程中,遇到加锁不成功,需要进入锁等待,此时进程就将自己阻塞在自身进程结构的信号量上,等待阻塞的进程释放拥有的锁唤醒自己。

2.2 InnoDB 同步实现方法

InnoDB 提供更为完善的接口,对操作系统层的互斥锁、事件等同步对象进行了完美的封装。事件是 Windows 操作系统提供的同步对象,所以面向 UNIX 等操作系统,InnoDB 用互斥锁和条件变量模拟 Windows 的事件,以此来实现查询线程之间的同步。

InnoDB 认为操作系统自身提供的互斥锁是慢速的,于是同 PostgreSQL 一样借助硬件的 TAS 指令实现了互斥自旋锁 Mutex 和读写自旋锁 RWLock。读写锁的实现依赖互斥锁,如果硬件不支持该原子指令,则继续采用操作系统的互斥锁。

InnoDB 的自旋锁并不是真正意义上的自旋锁,因为它将最终进入等待状态,这一点与 PostgreSQL 中的忙等方式是不一样的。InnoDB 采用主等待队列的方式实现查询线程因等待互斥锁或是读写锁进入阻塞。主等待队列的大小根据系统支持的最大线程数在数据库初始化时设定。主等待队列有很多的槽,每一个槽中都有一个事件对象用于线程的停靠。采用主等待队列的方式可以使互斥锁变得更快,而且这种方式比一个互斥锁对应一个事件来得更为有效。查询线程运行过程中,遇到加锁冲突需要进入锁等待,此时线程就到主等待队列中找一个没有被使用的槽,将自己挂在该槽的事件对象上,等待别人的唤醒。由于队列的大小是大于等于最大线程数的,所以不会发生找不到槽的现象。

2.3 PostgreSQL 和 InnoDB 的同步机制比较

PostgreSQL 和 InnoDB 都分别提供了两种类型的内部锁(通常更专业的称谓是闩锁或销(Latch))用于内存中共享数据的保护,如表 1 所示。

表 1 两种类型的内部锁

类型	类型①	类型②
PostgreSQL	自旋锁	轻量锁
InnoDB	旋转互斥锁	旋转读写锁

表 1 中,①型锁以独占的方式访问共享数据;②型锁以多个读一个写的方式访问共享数据。PostgreSQL 的①型锁采用自己忙等的方式,等待时间从 10ms 到 1s 不等,等完继续尝试,直到获得需要的锁为止。InnoDB 的①型锁和②型锁也忙等一段时间,但并不是永远地忙等直到获得该锁,而是忙等一定的时间之后进入阻塞,等待别人的唤醒。PostgreSQL 的②型锁则是有冲突就阻塞。

在忙等的方式上,PostgreSQL 和 InnoDB 也采用了不同的方法。PostgreSQL 中,进程先进行连续 100 次的 TAS,然后调用操作系统的 Select 函数进入阻塞状态,醒来之后再进行 100 次的 TAS 操作,如果仍然无法获得,继续进入 Select 阻塞,这样重复直到获得该锁。TAS 操作有硬件的支持,是快速的原子操作,但是如果不停地进行 TAS 操作,势必引起总线(寄存器到存储器的地址数据传送)繁忙。所以在 InnoDB 中,采用了一个聪明的做法:定义了一个内存值,TAS 操作只做一次,然后就不停地检测这个内存值,让出总线。如果检测到的内存值变成了期望值(0),那么就做一次 TAS 检测一下,判断是不是真的可以获得该锁,可以就成功返回;否则,检查自己的时间片是否还有,还有就继续检测内存值,没有了就准备进入等待状态。但是在线程悬挂之前,再连续进行四次 TAS 操作,做最后的尝试,实在无法获得才进入真正的阻塞,等待别人唤醒。InnoDB 采用这样的方式减少了 CPU 的争用,一定程度上提高了效率,这也是后面实验中修改测试的一个方面。

PostgreSQL 和 InnoDB 的阻塞唤醒依赖对象是不同的,在上面的叙述中也可以清楚地看到,PostgreSQL 采用信号量,InnoDB 采用事件(UNIX 下用互斥锁和条件对象模拟)。应该说互斥锁是为上锁而优化的,条件变量是为等待而优化的。信号量既可用于上锁也可用于等待,所以相对也更为复杂,需要的代价也更高。

在《UNIX 网络编程》一书中给出的线程同步性能测试中,互斥锁的性能是最好的,其次是读写锁,然后才是信号量。所以在后面的实验中修改测试的另一方面就是该同步对象。

3 PostgreSQL 和 InnoDB 的并发控制

3.1 PostgreSQL 并发控制实现方法

PostgreSQL 主要依赖多版本来保证读一致性。锁主要用于无法接受多版本的时候,如两个事务同时更新一个元组时,采用事务锁实现并发控制。

PostgreSQL 将查询操作分为两个阶段:读阶段和写阶段,对只读操作就没有写阶段。在读阶段都采用多版本的方式读取对自己可见的数据,在写阶段用事务锁并发控制。PostgreSQL 做到了读写永不堵塞,只有写阻塞(这样来说是不满足串行化定理的,所以 PostgreSQL 的可串行化隔离级别并不是真正严格意义上的可串行化,需要程序员自己来保证)。

PostgreSQL 的每一次更新操作都在原数据文件中产生一个新的版本,所以在存储空间上造成了很大的浪费,需要定时的人工做 Vacuum 操作来清理旧版本,而 Vacuum 操作是很耗时的。但是这样做的好处是使得恢复操作极其简单,而且也做到了真正意义上的多版本。PostgreSQL 采用快照的概念来表示一个查询可以看到的版本,这与后面 InnoDB 采用读视图的概念在理论上是一致的。

PostgreSQL 分别针对不同的查询语句,提供了八种重量锁,主要的加锁对象有事务、关系、页,但是它们并不是都满足两阶段封锁协议。PostgreSQL 定义了标准的四级隔离级别,但是在内部却只是实现了两级,这与它使用多版本是有直接关系的,因为多版本只能提供两种快照,一种语句级的,实现了读提交隔离级别;一种事务级的,实现了可串行化。但实际上,PostgreSQL 的可串行化级别并不是严格的可串行化,只能算作可重复读。严格的可串行化需要应用程序员自己来保证,如使用 Lock Table, Select…For Update 命令。

重量锁加锁的过程容易引起死锁问题。PostgreSQL 采用乐观的死锁处理方法,即对等待队列进行拓扑排序,预先消除软死锁,防止进一步的硬死锁产生。系统定时启动死锁检测程

序来发现处理死锁,但是现在 PostgreSQL 有一个最为严重的错误,它没有锁超时的检测,这样一个事务为获得一个锁可能等待很长时间,而拥有该锁的事务恰好发生某些状况一直不提交。

3.2　InnoDB 并发控制实现方法

InnoDB 采用多版本和两阶段封锁协议相结合的方式实现上层的并发控制。InnoDB 不是只注重多版本也不是只注重封锁,而是采用了一种折中的策略,将查询分为两种,即无格式(Plain)的查询(也就是普通查询,理解为只读查询,包括单一的只读查询和子查询中的只读查询)和有格式(Unplain)的查询(包括 Select For Update、插入、删除、更新及其他操作)。针对前者,InnoDB 采用多版本读的方式实现读一致性;对于后者,InnoDB 采用两阶段封锁(强两阶段)来保证读一致性。

多版本的实现,InnoDB 将旧信息存储在回滚段,利用回滚段中的信息来恢复旧版本,这与 PostgreSQL 是不同的。回滚段可以重复利用,所以解决了占用大量存储空间的缺点。同样也不需要费时的 Vacuum 操作,小型的 Purge, Truncate 操作就清除了回滚段、数据文件中的旧数据。InnoDB 采用读视图这个概念来表示一个查询可以读到的版本,机制同 PostgreSQL 一致。

InnoDB 实现了标准的四个隔离级别。在最高级别可串行化中,InnoDB 强行将所有的非封锁一致读转换为锁读,严格地抵制了各类不一致现象的发生。

InnoDB 对死锁的处理相比 PostgreSQL 简单,在加锁过程中产生冲突时进行死锁检测。检测到死锁,选择一个代价较小的事务回滚;没有死锁,进入锁等待,并启动锁等待超时检测线程,一旦超时,就给用户发送消息,希望用户主动重启该事务。

3.3　PostgreSQL 和 InnoDB 的并发控制比较

Oracle 作为现在的主流数据库,也在此方面与其进行了比较。

在操作的处理上,PostgreSQL 对操作分阶段,读写操作分为读、写两个阶段,只读操作只有读阶段。Oracle 是先分类,分为只读操作和读写操作,然后对读写操作再分阶段,分为读、写两个阶段。InnoDB 仅仅分类,分为无格式查询和有格式查询。

在存储上,PostgreSQL 性能最差,将所有的旧版本保留在原数据文件中,造成存储空间的严重浪费,Oracle 和 InnoDB 就改变了它的做法,将旧信息提取到回滚段中。但是 InnoDB 保留了 PostgreSQL 的机制,通过事务 ID 号加活动事务链表判断一个元组的可见性,而 Oracle 则采用了一个新的机制,对每一个元组增加一个系统序列号来表示版本的新旧,这样减少了很多的判断,存在回滚段中的信息也进一步减少,节约了空间。如何改变当前 PostgreSQL 存储空间大量浪费的现象是今后研究的目标之一。

多版本和封锁都可以保证读一致性,对于读事务较多的系统来说,多版本的方式是最理想的,因为多版本方式读写不阻塞,但是对写事务尤其是对同一元组的竞争较多的系统来说,封锁方式应该更为理想一些。

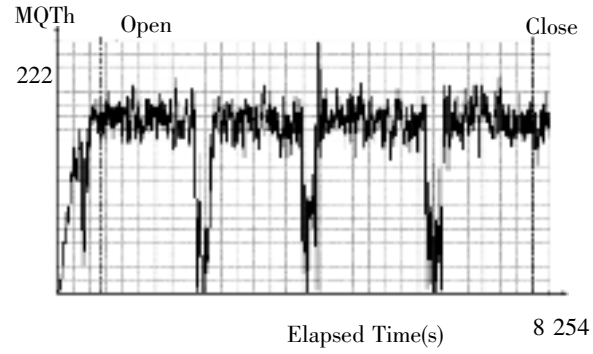
对并发控制方面的研究在国内外都有很多的专著,但是谁也不能很明确地表示哪一种就是最好的,因为他们都有各自适应的角度。所以在选择采用何种并发控制方法时,需要看产品面向的应用领域,以此选择最合适最有效的方法。现在有更多的专家提出 Lock-Free(从不阻塞)的方式,但是现在还没有发现应用到数据库的实现上。PostgreSQL 的开发组成员也提到是不是可以应用这类方式,但讨论的结果是无法应用。但是这方面的研究必将热火朝天地继续下去。

4　实验与分析

基于上面的比较分析,我们对 PostgreSQL 的同步机制借鉴 InnoDB 的方式从两个方面进行了修改,并进行 TPCC 测试,可以从中看出不同方式的性能对比。在实验的过程中,还了解到 PostgreSQL 全球开发组在近几年的 MAIL 讨论中,也提出过对同步机制的修改,尤其是 Spinlock,它是全球开发组 TODO. list 中的一项内容。实验的环境如下:

硬件: 内存 1. 0GB, CPU SY 2. 0GHz; 操作系统: RedHat Linux 9. 0, 内核为 Linux2. 4. 2; 数据量: TPCC 库 12 个; 数据库版本: PostgreSQL 7. 4. 3; 测试工具: 多机版 TPCC 测试软件; 测试时间: 7 200s。

图 1 给出原始的 PostgreSQL 性能。我们主要考查吞吐量、每分钟 NewOrder 事务个数统计图和每分钟 NewOrder 事务响应时间。从图中可以看出 PostgreSQL 的检查点时间长而且效率很低。



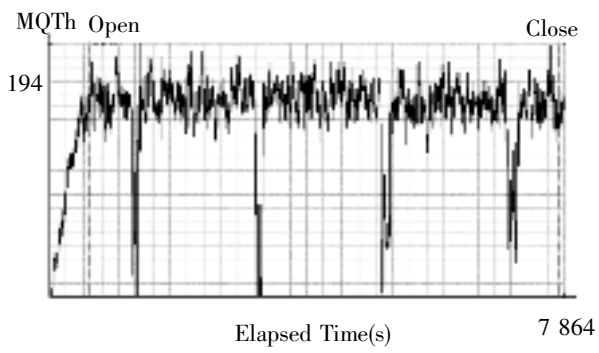
吞吐量	90%	平均	最大
142. 57	0. 24	2. 1	195. 13

图 1　PostgreSQL 原始性能数据

4.1　实验一

实验一的方式是修改忙等方式,借鉴 InnoDB 的妙处,摈弃频繁的硬件操作,替换为读取内存值的方式,以 20ms 为时间片长度,循环检测。获得失败,采用 Select( ) 函数阻塞一定的时间,时间从 10ms 到 1s 增长。这里还可以考虑用 UNIX 支持的 Sched_yield( ) 函数来出让 CPU 的控制权。

修改后每分钟 NewOrder 事务个数统计图及数据如图 2 所示。



时间	吞吐量	90%	平均	最大
修改前	142. 57	0. 24	2. 1	195. 13
修改后	145. 7	1. 1	1. 69	109. 23

图 2　实验一实验数据

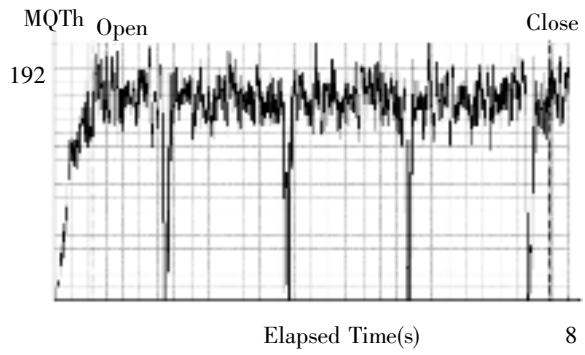
从图形的比较上发现检查点的时间有比较大的缩短。从数据上比较,主要比较吞吐量和新订单 NewOrder 的响应时间:我们可以看到,吞吐量有一定的提高,但不是很明显,90% 的响应时间有比较大的下降,但是平均响应时间有比较大的提升。

4.2 实验二

实验二的方式采用互斥锁和条件变量来替换原来的信号量,以此实现进程的阻塞唤醒。互斥锁和条件变量都是操作系统提供的机制,只要对操作系统提供的函数做好比较完整的封装就可实现进程的阻塞唤醒。

需要注意的一点是,互斥锁和条件变量必须都在共享内存中初始化才可以实现进程相互之间的唤醒,因为进程是跑在私有内存中的,互相之间需要通过共享内存通信。

修改后每分钟 NewOrder 事务个数统计图及数据如图 3 所示。从图形的比较上我们也发现检查点的时间有明显的缩短。从数据上看结果与上面较为一致,吞吐量有一定的提高,但不是很明显,90% 的响应时间有比较大的下降,但是平均响应时间有比较大的提升。

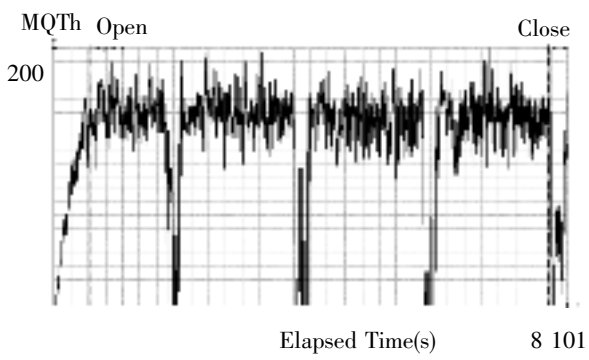


时间	吞吐量	90%	平均	最大
修改前	142.57	0.24	2.1	195.13
修改后	146.03	0.94	1.58	111.57

图 3 实验二实验数据

4.3 实验三

两个修改结合起来一起测试,每分钟 NewOrder 事务个数统计图及数据如图 4 所示。从图形的比较上我们也发现检查点的时间有一定的缩短,但是没有独立测试时明显。而从数据上看,结果大大出乎预想,吞吐量、90% 的响应时间、平均响应时间都有很大程度的下降。



时间	吞吐量	90%	平均	最大
修改前	142.57	0.24	2.1	195.13
修改后	138.32	3.29	2.76	203.08

图 4 实验三实验数据

4.4 实验总结分析

从三个实验数据上看,效果最明显的是检查点的时间缩短,其他的效果都不是很明显,尤其是实验三的数据与预想相差很大,分析原因可能有以下几个方面:

(1) 进程阻塞唤醒的时间加快,进程可以更关注自己的工作,检查点进程可以专心地做刷写磁盘的工作,而不需要费很大的时间去阻塞唤醒他人。这是检查点时间减少的原因。

(2) PostgreSQL 采用进程的方式执行查询,需要不停地切换进程,进程切换的开销远远大于信号量或是互斥锁上锁释放锁的时间,这是效果不明显的原因。换言之,如果 PostgreSQL 可以采用线程来执行查询,那么这样的修改应该可以在很大程度上提高效率。

(3) 采用互斥锁和条件变量实现进程的阻塞唤醒,其中条件变量也是一个内存值,而自旋锁采用读内存值的方式,这样一定程度上两个之间存在争用,因此导致第三个实验的数据反而下降。

5 总结

本文从实现和方法两个角度剖析了 PostgreSQL 和 InnoDB 的并发控制技术。

(1) 在实现上,主要剖析了锁实现的基础——同步机制:PostgreSQL 采用系统信号量实现进程的阻塞唤醒,并实现自旋锁 Spinlock 和轻量锁 LWLock 两种类型的内部锁满足共享内存中对数据的互斥访问;InnoDB 采用系统事件(UNIX 系统通过系统互斥锁和系统条件变量模拟实现)实现线程的阻塞唤醒,并实现互斥自旋锁 Mutex 和读写自旋锁 RWLock 两种类型的内部锁满足需要互斥访问的内存数据之间的并发。

(2) 在方法上,PostgreSQL 依靠多版本(MVCC),辅以重量锁实现关系、记录等逻辑上数据的互斥访问;InnoDB 折中利用多版本和重量锁实现。

最后,笔者通过对同步机制的修改实验对比了它们的优劣性,虽然没有强烈的数据改变,但是反映了同步对象各自适用的不同环境。实验总结的原因理论基础、实验基础并不是很充分,加上目前 PostgreSQL 没有全面的类似 Oracle 的动态性能视图来表示运行的状态,很难分析明确的原因,所以接下来的工作之一是在 PostgreSQL 中实现动态性能视图,用它来分析更为细致的问题。而进一步的研究目标就是如何提高 PostgreSQL 的并发性能。

参考文献:

[1] ector Garcia-Molina, Jeffrey D Ullman, Jennifer Widom. Database System Implementation[M]. China Machine Press, 2002. 467-539.

[2] Alexander Thomasian. Concurrency Control: Methods, Performance, and Analysis[J]. ACM Computing Surveys, 1998, 30(1): 70-119.

[3] Jim Gray, Andereas Reuter. Transaction Processing: Concepts and Techniques[M]. China Machine Press, 2004. 269-350.

[4] 汤子瀛,等. 计算机操作系统[M]. 西安:西安电子科技大学出版社, 1996. 61-71.

[5] PostgreSQL 7.4 文档[EB/OL]. http://www.pgsqldb.org/twiki/bin/view/PgSQL/PgDocList.

[6] InnoDB Reference Manual[EB/OL]. http://www.innodb.com/ibman.html.

作者简介:

萧美阳(1981-),女,硕士研究生,研究方向为数据库事务处理;叶晓俊,副教授,研究方向为企业信息系统与数据库。