并发控制

摘要。

1、前言

2、内存数据库的发展。

在计算机的体系中，内存一直是非常重要的，它负责将CPU和外部的设备连接起来，随着计算机技术的不断发展，内存技术日新月异。它能够为CPU数据处理提供直接的数据访问。一般来说，磁盘数据库系统将数据都放在磁盘上进行处理，而这个会造成经常地访问磁盘，造成巨大的I/O代价。内存数据库在内存计算中是非常具有代表性的，内存虽然是易失性的存储介质，但是相对于磁盘来说，内存的计算能力是极好的。内存数据库一般支持IPC、DOMAIN、TCP/IP等多种连接方式，应用可以很方便地进行扩展。内存数据库天然地可以达到非常高的数据存取速度和很好的并发访问能力。

随着半导体存储器变得越来越便宜并且芯片密度增加，在存储器中存储越来越大的数据变得可行，使得内存数据库(MMDB)成为现实。 由于数据可以直接在内存中访问，MMDB与DRDB相比，响应时间和交易吞吐量要好得多。 这对于实时应用程序来说尤其重要，因为在这些应用程序中，交易必须在指定的期限内完成。

在内存数据库中（MMDB）中，它们将数据存储在主物理内存中，并提供非常高速的访问。 传统的磁盘数据库系统（DRDB）针对磁盘存储机制的特定特性进行了优化。 另一方面，内存数据库使用不同的优化来组织和结构化数据，并使其具有可靠性。很明显的是，在磁盘数据库系统中，我们进行数据读写的时候，由于要进行磁头的机械移动，所以响应时间非常的慢。而在内存数据库中，内部的优化算法相对来说比较简单，且执行的CPU指令较少，同时读写数据时发生的是电信号的移动，这个速度是非常快的。通过内存数据库使用实现数据存储来提高吞吐量，并且降低延迟。

由于计算机的内存相对于磁盘来说有着许多不同的特性，这些差异对数据库的设计和性能有着深远的影响，下面对这些特性进行总结。

（1）对内存的访问时间比对磁盘的访问时间减少了几个数量级。

（2）内存通常是易失性的，而磁盘存储是非易失性的。但是也有可能构建非易失性的内存。

（3）磁盘是面向块的存储设备，而主存并不是面向块存储的。

（4）磁盘上的数据分布比内存中的数据分布重要，因为有序的访问磁盘比无序的访问磁盘速度上快很多。然而，在内存中，顺序地访问并不是很重要。

数据库是一个共享的资源，可以满足多个用户并发的访问。在一个事务中，一系列的数据库操作只有全做和全不做两种选择，即它们是绑定在一起的一个整体。当许多用户同时访问时，并行的事务数可以有许多个。这容易对内存数据库的ACID等特性造成破坏，为了能够保持数据库的事务特性在内存数据库中进行并发控制访问是非常重要的。事务型的内存数据库发展的也比较快。以下是一些支持事务处理的典型内存数据库系统。

1、Hekaton。Hekaton是在2012年的SQLServer专业人士的技术会议上，微软发布的一个内存数据库系统。它是针对事务处理的基于行的内存数据管理系统。后来，它被被集成到SQL Server 2014数据库中，成为SQL Server内存OLTP引擎。它由两部分组成：内存优化表和本地编译存储过程。它采用哈希索引，并发的隔离方式也对应的变成了MVCC。但是在分配时间戳的时候，容易产生全局的争论点。

2、VoltDB

它是H-Store的商用版本，其中H-Store是一个高性能的分布式的内存数据库，它是为在线事务处理应用而设计的数据库管理系统，被称为NewSQL。VoltDB将所有的内容存储在内存中，通过避免进入磁盘，以求获得显著的性能提升。它通过并行单线程来保证事务的一致性和高性能。VoltDB数据库有大量分散在多个服务器上的分区组成，VoltDB对每个节点的内存进行管理，在每个节点上创建多个分区，所有分区表中的数据，都分散在各个分区中，所以在读写的时候，就可以实现多个分区并发进行。其中每一个分区对应一个VoltDB服务器进行，在单个进程里使用单线程，所有的事务执行都是顺序的。这种分区机制也会带来问题，当集群需要扩容的时候，需要停止整个集群，然后再进行扩容；当集群启动的时候，VoltDB会重新调整数据分布，在所有数据分布调整完毕之后，才开始提供服务，传统的关系数据库（比如 MySQL）和大多数 NoSQL 系统将其数据存储在磁盘上。

3、Silo

Silo是一个基于多内核的内存数据库，有着高性能和可扩展性。它避免了集中化的 Transaction ID的分配。它的主要贡献是一个基于乐观并发控制的提交协议。由此，它能够支持可串行化，同时对于只读的事务，它避免了对共享内存的写操作。Silo通过它自定义的定期更新的epochs(时间戳)来提供可恢复性。Silo在保证完成基本的数据库功能的同时，避免了不可扩展的瓶颈，也没有更多的额外延迟。通过实验，Silo在TPC-C的测试基准，32内核的计算机下，每秒能够执行700,000个事务。它同时支持存储过程。Silo的数据表被设计成索引树的集合，每一个表包括一个primary tree和多个secondary trees。在表的primary tree上，每一个record被存储在单独分配的内存块上，叶节点的data域存放的则是对应的record的地址。

3、内存数据库中的并发控制。

1、乐观并发控制的历史和算法

乐观并发控制的算法是在1981年提出来。它乐观地假设多个并发的事务在运行的时候是互不影响的，由此在执行阶段不加锁地运行，提高了事务的执行效率。只有在事务提交的时候才会检测，每一个事务会检查在该事物读取了某一个数据后，是否有另外一个事务又修改了该数据。如果确实如此的话，通过将事务回滚来保证一致性。乐观并发控制认为在数据库的操作上，大部分的数据库操作应该是不会有冲突的。乐观并发控制一般经过三个步骤。如下所示：

1.读取阶段：事务将数据读取本地缓存中，事务所有的操作都在本地这个缓存中进行，在此期间，本地的副本并不会被其它事务访问到。

2.验证阶段：验证本事务读取过的数据是否会被其它并行的事务操作而发生更改。

3.写入阶段：如果验证通过，则将数据永久地写入数据库中。

一般来说乐观并发控制对于处理冲突较少的场景是非常有效的，但是如果并发的事务冲突较大，则不宜使用乐观并发控制。

2、Silo中的基于乐观并发控制的提交协议

1、epochs

Silo是基于一个被称为epochs的时间间隔来提供可串行化的恢复，同时进行垃圾移除和提供只读的snapshots。每一个epoch有一个epoch number。

有一个全局的时间戳E对于所有线程都是可见的。在系统中，有一个特定的线程定期地更新E，即通过不断地做自增操作。其它所有要提交事务的线程可以访问这个全局的时间戳。由于这个时间间隔会影响事务的时延，所以E应该频繁地增加。但是，相对于事务的持续时间长度，在这段时间里，epoch基本是不会发生改变的，所以可以被缓存在本地中。在设计中，每一个时间块的时间长度被设计成40ms，当然，如果将时间块长度设置的稍微短些也是符合要求的。

每一个worker w访问全局的epoch E的时候，都会在本地缓存一个局部的epoch number:ew。当worker在计算的时候，这个本地的ew会变得比E小，这个可以用来决定什么时候可以安全地回收垃圾。Silo要求E-ew<=1。如果一个worker没有在有效的时间里完成事务，随着时间的进行，E可能会越来越大，所以为了满足约束，那么特定的线程会延迟更新全局E；如果执行的是较长的事务，则workers定期更新本地的ew，确保在下一个epoch里能够继续执行该事务。

2、TID

Silo的并发控制依赖于它的TID，通过这个TID可以识别事务和record versions。这个在Silo中被当做锁来使用，同时可以检测冲突。每一个record保存了最新近修改过它的事务的TID。

它是一个64-bit的整数，高位部分是上面提到的epoch number，中间部分是一个序列化的数字，用来区分同一个epoch发生的事务；低位三个比特位是状态位，是用来自我区分的，其中低三位分别是lock bit、latest-version bit、absent bit。TID的顺序大体上能够反映事务的序列顺序。Silo和其它许多系统是不一样的，Silo以去中心化的原则来分配TIDS，一个worker只有确认该事务可以提交才去选择该事务的TID，然后TID选择满足如下规则的一个最小的TID。

1、比该事务中正在读或写的任何record中的TID都要大。

2、比该worker最新近选择的TID都要大。

3、在当前 global epoch下，即TID的高位部分数字为当前的E。

其中TID中的低三位被称为状态位。 这三位比特位分别是一个lock位，一个latest-version位，一个absent位。在TID中包含这三个比特位是非常有用的。它可以简化并发控制，例如：Silo可以在一个原子操作里更新reord的版本同时给锁住record。这个lock位可以保护record memory免受并发其它的并发事务的更新；在数据库的术语中它可以被称为一个latch。当记录保存相应主键的最新数据时，这个latest-version位是1。当这个记录被取代以后(例如，当一个快照事务暂时保存一个过时的记录)，这个latest-version就会变为0。最后，由这个absent位标记这个record，这个就意味着这个record是不存在的; 这样的records是在我们的插入和删除实现中创建的。我们使用TID来表示一个存粹的事务的ID，而使用TID word 来表示一个加了状态位的TID。

3、数据分布

Silo中的一个record是32字节，包含如下信息：1、一个上述说明的TIDword。2、一个指向先前版本的指针。如果没有先前的版本，那么该指针为NULL。这个先前的版本主要是是支持快照事务的。3、该record中的数据。在可能的情况下，实际record数据与record header存储在相同的缓存行中,这么做是为了避免访问字段值而进行的额外内存提取。

提交的事务通常会及时修改record数据。这个加快了short writes的性能，这个主要是因为可以减少对于record对象的内存分配开销。但是，读者必须使用版本验证协议来确保他们已经读取了每个record数据的一致版本。

4、提交协议

我们利用乐观并发控制的思想，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回滚。

在执行的阶段的时候，通过假设冲突较少，各个事务可以并发地执行。

当一个worker开始运行一个事务的时候，该worker会维护一个read-set ，在该read-set中保留了所有被读取的record和它们的TID。同时维护一个write-set，该写集里面保存的是已经修改过的record和TID，注意这里的TID也是修改过的TID。提交协议如下所示：

w表示write-set，r表示read-set，v表示new-value,N表示node set。

**// Phase 1**

for w, v in WriteSet {

Lock(w); // use a lock bit in TID

}

**Fence**(); // compiler-only on x86

e = Global\_Epoch; // serialization point

**Fence**(); // compiler-only on x86

**// Phase 2**

for r, t in ReadSet {

Validate(r, t); // abort if fails

}

for node,version in N do{

if node ,version ≠version then abort();

}

tid = Generate\_TID(ReadSet, WriteSet, e);

**// Phase 3**

for w, v in WriteSet {

Write(w, v, tid);

Unlock(w);

}

在第一个阶段，即读取阶段，该worker检查了该事务中write-set中的所有records。通过获取records中的lock位，将每一个record加锁。为了避免死锁，workers以一个任意全局的顺序给records加锁，同时Silo使用指针来定位records。当获取到所有的write lock以后，该worker经过一次单独的内存访问，获取一个全局epoch number的快照。fences被用来确保这个read是到内存中进行读取操作的，而不是进入到一个过期的cache中进行读取。这是由于在编译的时候，有可能会改变代码的执行顺序，fences并不会影响编译的指令，它们只是防止编译器任意地移动代码。这里获取到的全局的E是事务提交时候的一个序列化点。

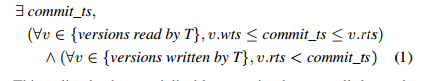
在阶段二, 该worker检查本事务中read-set中的所有records，这里的records可能同时有被进行读取操作和写操作。对于read set里面的record，我们通过record.tid操作得到此时record中的最新tid，将它和当初保存在read-set 中的read-tid进行比较，如果它此刻的TID和执行阶段得到的TID不一样的话，或者被其它并行的事务给加锁，那么本事务将会释放它的锁，并且中止该事务；如果它的TID没有改变，我们将会用阶段一得到的E，利用前面叙述的TID选择规则，得到一个TID。

在最后的阶段，将修改过的记录records写入到数据库中，并且同时使用前面阶段二计算出来的TID来更新它们的TID。在这里当record被写入的时候，每一个锁可以及时地被释放。Silo可以确保当锁被释放的时候，这个新的TID就可以被其它事务访问了，这是因为TID和锁同时在一个TID word中，这样可以保证原子性写入。

2、TicToc算法

像其它基于时间戳的算法一样，TicToc使用时间戳来显示事务的序列顺序。但是，与以前的方法不同的是，它不会使用集中式的分配时间戳，避免了许多冲突。相反的是，事务的时间戳是在提交时的时候，以分布式的方式基于它所访问的元组而计算出来的。这个时间戳管理策略有两个关键的优势。首先，它的分布性避免了时间戳分配中固有的瓶颈，使算法具有高度的可扩展性。其次，在工作负载中，有利于更多的产生更多的并行性，因此，可以减少异常情况，同时提高性能。

为了在元组中编码序列化的信息，在TicToc中每一个数据版本有一个有效的提交时间戳范围，这个范围由写时间戳(wts)和读时间戳(rts)来确定。特别的是，一个特别的数据版本在写时间戳(wts)的时候被创建，并且直到读时间戳(rts)的时候都是有效的。被一个事务读的数据版本的时间戳是有效的，只有该提交时间戳是在wts和rts之间的。而一个被一个事务写的时间戳是有效的，有且仅当这个事务的提交时间戳比先前版本的读时间戳要大。即对于一个要提交的事务，它必须满足下面所示的不等式的要求：

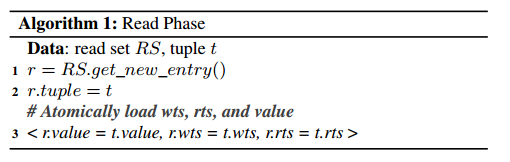


此策略有办法保证可序列化的执行，因为事务内的所有读取和写入发生在相同的时间戳。读取总是返回在该时间戳处有效的数据版本，并且在对相同元组的较旧版本的所有读取之后，才会进行写操作。

TicToc算法也是乐观并发控制的一个改进的算法。在执行阶段，可以不加锁地访问数据库。这个在TicToc中被称为读取阶段。当事务调用提交操作时，协议会通过验证阶段来检查事务是否被允许提交。如果确实允许的话，接着它进入到写入阶段，即意味着事务的更改可以被保存在共享数据库中。

在读取阶段，对于每一个事务，DBMS分开保存一个单独的read-set和write-set。在这个阶段，被访问的元组被存进read-set，而被修改的元组被存在write-set中。同silo一样，此时存在本地的read-set和write-set只能被当前事务访问。在write-set和read-set中的每一个条目以{tuple,data,wts,rts}的形式存放，在这里tuple是一个指针，可以在验证阶段方便的读取到tuple中的最新数据。data是tuple中的数据值，而wts和rts则是当前这个事务访问元组时读取到的读时间戳和写时间戳。对于一个读的条目，TicToc保证时间戳顺序在wts和rts之间，该数据版本是有效的。

算法1展示了在读取阶段，对于一个元组访问的过程。根据请求的类型，来决定指向元组的指针是保存在write-set中还是read-set中，同时，数据值和wts和rts也会被同时记录。假设这个值和时间戳是原子地被记录的，这样可以保证这个值匹配的是对应的时间戳。



在验证阶段，TicToc使用前面阶段存储在对应事务中read-set和write-set中的时间戳来计算出它的提交时间戳(commit\_ts)。然后，根据我们计算出来的这个提交时间戳，这个算法会检查在当前事务中的read-set中的元组是否是有效的。如下面算法2中所描述的那样，验证阶段的第一步，是以主键顺序锁定事务write-set中的所有元组以防止其他事务并行地更新相应的数据行。这个方法在Silo中也有使用，按顺序的加锁，是为了防止发生死锁。

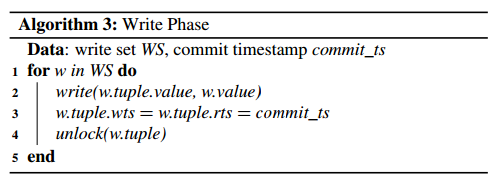
验证阶段的第二步是根据write-set和read-set中每个元组条目中存储的时间戳来计算事务的提交时间戳。就像前面所描述的那样，对于在read-set中的元组，但是不在write-set中的元组，这个提交时间戳不应该比wts小，如果提交时间戳比wts还小，这个元组将会有不同的数据版本。对于这个事务中的write-set中的元组，这个提交时间戳需要保证不会比rts+1小，因为先前的数据版本直到rts为止都是有效的。

在最后一步，该算法会验证在本事务中的read-set中的元组。如果事务的提交时间戳(commit\_ts)小于或者等于rts，即这个commit\_ts在[wts,rts]里面。这意味着被这个事务读取的数据版本是有效的，接着验证阶段便结束了。但是如果read-set中条目中rts比提交时间戳(commit\_ts)小，那么在提交时间戳commit\_ts的时候，并不清楚这个本地保存的值是否是有效的。很有可能同时有其它并行的事务在rts和commit\_ts之间的某一个逻辑时间修改了这个数据。如果可以确定并没有其它的事务修改这个元组，那么rts可以被扩展到commit\_ts，使得该数据版本在提交时间戳commit\_ts的时候是有效的。

特别地，这个本地的写时间戳先被用来和最新的wts进行比较。如果他们是不同的，那么可以认为这个元组已经被其它并行的事务修改了，因此，不可以将tuple中的rts扩展到这个临时的版本。如果本地的写时间戳和最新的wts时间戳是相等的，但是这个元组已经被其它不同的事务给加锁了，即这个元组不在本事务中的write-set中，但是却被加锁了，这时rts是不可以被扩展的。如果这个rts是可以扩展的，或者说这个数据版本在commit\_ts是可以被扩展的，那么这个元组的的rts可以被至少扩展到commit\_ts。

验证read-set中的值是否是有效的这个阶段，整个过程是原子地进行的，为了防止其它事务的干扰。由于在验证阶段开始的时候，在write-set中的元组已经被加锁了，所以数据库管理系统不需要去验证write-set中的元组是否是有效的。在TicToc中，在整个事务进行的阶段，不会有集中化的冲突点。

在写入阶段，如果这个事务访问的元组经过了验证，那么这个事务进入到了写入阶段。就如算法3描述的那样，在这个阶段，该事务中write-set中的tuples被写入到数据库中。同时，对于在该事务write-set中的每一个tuple，数据库管理系统将他们的wts和rts设置成commit\_ts，这表明这是一个新的版本。同时，在验证阶段中给加的锁全部被释放掉，接着其它的事务便可以看到这种变化。



3、silo中的提交协议和Tictoc算法的比较。

两者都避免了集中化的时间戳分配，这都减少了会发生的冲突点。相对于Silo来说，使用Tictoc的算法相对来说还可以减少冲突和提高性能。