摘要

多版本并发控制（MVCC）是目前现代数据库管理系统（DBMSs）中最流行的事务管理方案。尽管MVCC是在20世纪70年代末发现的，但它几乎被用于过去十年中发布的所有主要的关系型DBMS中。在处理事务时，维护多个版本的数据可能会增加并行性，而不会牺牲可串行性。但是，在多核和内存环境中扩展MVCC并不容易：当有大量的线程并行运行时，同步开销可能超过多版本的好处。

为了了解MVCC在现代硬件环境下处理事务时的表现，我们对该方案的四个关键设计决策进行了广泛研究：并发控制协议、版本存储、垃圾回收和索引管理。我们在一个内存DBMS中实现了所有这些的最先进的变体，并使用OLTP工作负载对它们进行了评估。我们的分析确定了每个设计选择的基本瓶颈。

1 介绍

计算机体系结构的进步导致了多核内存数据库管理系统的兴起，这些系统采用高效的事务管理机制来最大限度地提高并行性，而不牺牲可序列化性。过去十年开发的数据库管理系统中最流行的方案是多版本并发控制(MVCC)。MVCC的基本思想是，数据库管理系统维护数据库中每个逻辑对象的多个物理版本，以允许对同一对象的操作并行进行。这些对象可以是任何粒度的，但是几乎每个MVCC数据库管理系统都使用元组，因为它提供了并行性和版本跟踪开销之间的良好平衡。多版本允许只读事务访问元组的旧版本，而不阻止读写事务同时生成新版本。与之形成对比的是，在单版本系统中，事务总是在更新元组时用新信息覆盖元组。

最近使用MVCC的数据库管理系统的趋势有意思的是，这个方案并不是新的。第一次提到它出现了的InnoDB引擎(自2001年)。但是尽管这些使用单一版本方案的旧系统有许多同时代的系统(例如IBM DB2，Sybase)，几乎所有新的事务性数据库管理系统都避开了这种方法，而选择了MVCC [37]。这包括商业广告(例如微软Hekaton [16]，SAP HANA [40]，MemSQL [1]，NuoDB [3])和学术(例如HYRISE [21]，HyPer [36])系统。尽管所有这些较新的系统都使用MVCC，但没有一个“标准”实施。有几种设计选择具有不同的权衡和性能行为。到目前为止，还没有一个现代DBMS操作环境下的MVCC综合评价。上一次广泛的研究是在20世纪80年代[13]，但它使用了在单个CPU内核的面向磁盘的数据库管理系统中运行的模拟工作负载。传统的面向磁盘的数据库管理系统的设计选择不适用于运行在具有大量CPU内核的机器上的内存中数据库管理系统。因此，之前的工作并没有反映出最近在无锁[27]和可序列化[20]并发控制以及内存存储[36]和混合工作负载[40]方面的趋势。

在本文中，我们对MVCC数据库管理系统中的关键事务管理设计决策进行了这样的研究:(1)并发控制协议，(2)版本存储，(3)垃圾回收，(4)索引管理。对于这些主题，我们描述了内存数据库管理系统的最新实现，并讨论了它们的权衡。我们还强调了阻止它们扩展以支持更大的线程数和更复杂的工作负载的问题。作为这项研究的一部分，我们实现了Peloton[5]内存MVCC数据库管理系统的所有方法。这为我们提供了一个统一的平台来比较不受其他架构方面阻碍的实现。我们在一台有40个内核的机器上部署了Peloton，并使用两个OLTP基准测试对其进行了评估。我们的分析确定了强调实现的场景，并讨论了减轻它们的方法(如果可能的话)。

2 背景

我们首先概述一下MVCC的高级概念。然后我们讨论数据库管理系统用来跟踪事务和维护版本信息的元数据。

Table 1: MVCC Implementations – A summary of the design decisions made for the commercial and research MVCC DBMSs. The year attribute for each system (except for Oracle) is when it was first released or announced. For Oracle, it is the first year the system included MVCC. With the exception of Oracle, MySQL, and Postgres, all of the systems assume that the primary storage location of the database is in memory.

	Year	Protocol	Version Storage	Garbage Collection	Index Management
Oracle [4]	1984	MV2PL	Delta	Tuple-level (VAC)	Logical Pointers (TupleId)
Postgres [6]	1985	MV2PL/SSI	Append-only (O2N)	Tuple-level (VAC)	Physical Pointers
MySQL-InnoDB [2]	2001	MV2PL	Delta	Tuple-level (VAC)	Logical Pointers (PKey)
HYRISE [21]	2010	MVOCC	Append-only (N2O)	–	Physical Pointers
Hekaton [16]	2011	MVOCC	Append-only (O2N)	Tuple-level (COOP)	Physical Pointers
MemSQL [1]	2012	MVOCC	Append-only (N2O)	Tuple-level (VAC)	Physical Pointers
SAP HANA [28]	2012	MV2PL	Time-travel	Hybrid	Logical Pointers (TupleId)
NuoDB [3]	2013	MV2PL	Append-only (N2O)	Tuple-level (VAC)	Logical Pointers (PKey)
HyPer [36]	2015	MVOCC	Delta	Transaction-level	Logical Pointers (TupleId)

2.1 MVCC 概况

事务管理方案允许终端用户以多程序的方式访问数据库，同时保持他们每个人都在专用系统上单独执行的假象[9]。它确保了DBMS的原子性和隔离性保证。

多版本系统有几个与现代数据库应用程序相关的优点。最重要的是，它可能比单版本系统允许更大的并发性。例如，MVCC数据库管理系统允许一个事务读取一个对象的旧版本，同时另一个事务更新同一个对象。这一点很重要，因为在读写事务继续更新数据库的同时，对数据库执行只读查询。如果数据库管理系统从不删除旧版本，那么系统也可以支持“时间旅行”操作，允许应用程序查询数据库在过去某个时间点存在的一致快照[8]。

上述优势使MVCC成为近年来实施的新数据库管理系统最受欢迎的选择。表1总结了过去三十年来MVCC的实施情况。但是在数据库管理系统中实现多版本有不同的方法，每种方法都会产生额外的计算和存储开销。这些设计决策也高度依赖于彼此。因此，辨别哪些人比其他人好以及为什么是重要的。对于内存中的数据库管理系统尤其如此，因为磁盘不再是主要的瓶颈。

在接下来的章节中，我们将讨论这些设计决策的实现问题和性能权衡。然后，我们在第7节中对它们进行综合评估。在本文中，我们只考虑可序列化的事务执行。虽然日志记录和恢复是数据库管理系统体系结构的另一个重要方面，但我们将它排除在我们的研究之外，因为它与单版本系统没有什么不同，内存中的数据库管理系统日志记录已经在其他地方讨论过了[33，49]。

2.2 数据库管理系统元数据

无论其实现如何，MVCC数据库管理系统都为事务和数据库元组维护公共元数据。

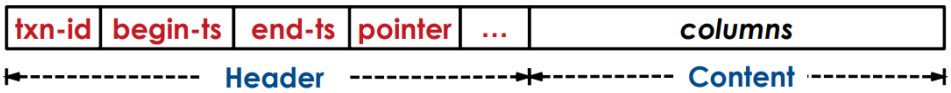


Figure 1: Tuple Format – The basic layout of a physical version of a tuple.

事务:当事务第一次进入系统时，数据库管理系统给事务分配一个唯一的、单调递增的时间戳作为它的标识符。并发控制协议使用这个标识符来标记事务访问的元组版本。一些协议还将它用于事务的序列化顺序。

元组:如图1，每个物理版本在其头中都包含四个元数据字段，DBMS使用这些字段来协调并发事务的执行(下一节讨论的一些并发控制协议包含追加字段)。txn-id字段用作版本的写锁。当元组未被写锁定时，每个元组都将该字段设置为零。大多数DBMSs使用64位txn-id，因此它可以使用单个compareand-swap (CaS)指令自动更新该值。如果标识符为Tid的事务T想要更新元组A，那么DBMS检查A的txn-id字段是否为零。如果是，那么DBMS将使用CaS指令[27，44]将txn-id的值设置为Tid。如果该txn-id字段不为零或不等于其Tid，任何试图更新A的事务都将被中止。接下来的两个元数据字段是表示元组版本生命周期的开始和结束时间戳。两个字段最初都设置为零。当事务删除元组时，数据库管理系统将元组的开始设置为INF。最后一个元数据字段是存储相邻(上一个或下一个)版本(如果有的话)的地址的指针。

3 并发控制协议

每个数据库管理系统都包括一个协调并发事务执行的并发控制协议[11]。该协议确定(1)是否允许事务在运行时访问或修改数据库中的特定元组版本，以及(2)是否允许事务提交其修改。虽然这些协议的基本原理自20世纪80年代以来一直没有改变，但由于没有磁盘操作，它们在多核和主内存环境中的性能特征发生了巨大变化[42]。因此，有一些较新的高性能变体，它们删除了锁/锁存器和集中式数据结构，并针对字节可寻址存储进行了优化。

在本节中，我们描述了MVCC数据库管理系统的四个核心并发控制协议。我们只考虑使用元组级锁定的协议，因为这足以确保可序列化的执行。我们省略了范围查询，因为多版本不会给幻影预防带来任何好处[17]。提供可序列化事务处理的现有方法使用(1)索引中的追加锁存器[35，44]或(2)事务提交时的额外验证步骤[27]。

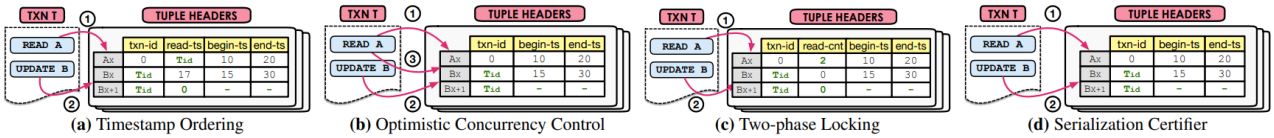


Figure 2: Concurrency Control Protocols – Examples of how the protocols process a transaction that executes a READ followed by an UPDATE.

3.1 时间戳排序 (MVTO)

1979年的MVTO算法被认为是最初的多版本并发控制协议[38，39]。这种方法的关键是使用事务的标识符(Tid)来预先计算它们的序列化顺序。除了第2.2节中描述的字段之外。版本头还包含读取它的最后一个事务的标识符(read-ts)。DBMS中止试图读取或更新其写锁由另一个事务持有的版本的事务。

当事务T调用逻辑元组A上的读操作时，DBMS搜索Tid在开始和结束字段范围之间的物理版本。如图2a所示。如果它的写锁没有被另一个活动事务持有(即txn-id的值为零或等于Tid)，因为MVTO从不允许事务读取未提交的版本。读取Ax时，如果Ax的当前值小于Tid，DBMS将Ax的read-ts字段设置为Tid。否则，事务读取旧版本，而不更新此字段。

对于MVTO，事务总是更新元组的最新版本。如果(1)没有活动事务持有Bx的写锁，并且(2) Tid大于Bx的read-ts字段，事务T创建新版本Bx+1。如果满足这些条件，那么DBMS创建一个新版本Bx+1，并将其txn-id设置为Tid。当T提交时，DBMS将Bx+1的开始和结束字段分别设置为Tid和INF，将Bx的结束字段设置为Tid。

3.2 乐观并发控制 (MVOCC)

下一个协议是1981年提出的基于乐观并发控制(OCC)的的方案[26]。OCC背后的动机是，数据库管理系统假设事务不太可能发生冲突，因此事务在读取或更新元组时不需要获取元组锁。这减少了事务持有锁的时间。最初的OCC协议有所改变，以适应多版本[27]。最重要的是，数据库管理系统不维护事务的私有工作空间，因为元组的版本信息已经阻止了事务读取或更新它们不应该看到的版本。

MVOCC协议将事务分为三个阶段。当事务开始时，它处于读取阶段。这是事务调用数据库上的读取和更新操作的地方。像MVTO一样，要对元组执行读操作，数据库管理系统首先根据开始和结束字段搜索一个可见的版本。如果未获得Ax的写锁，则允许t更新Ax版本。在多版本设置中，如果事务更新版本Bx，然后DBMS创建版本Bx+1，其txn-id设置为Tid。

当一个事务指示数据库管理系统它想要提交时，它就进入了验证阶段。首先，DBMS为事务分配另一个时间戳(Tcommit)，以确定事务的序列化顺序。然后，数据库管理系统确定事务的读取集中的元组是否由已经提交的事务更新。如果事务通过了这些检查，那么它就进入了写阶段，在这个阶段，DBMS安装所有的新版本，并将它们的开始设置为Tcommit，结束设置为INF。

事务只能更新元组的最新版本。但是一个事务不能读取一个新版本，直到创建它的另一个事务提交。读取过时版本的事务只会发现它应该在验证阶段中止。

3.3 两阶段封锁 (MV2PL)

该协议使用两阶段锁定(2PL)方法[11]来保证事务的可串行化。在允许读取或修改逻辑元组之前，每个事务都在逻辑元组的当前版本上获得适当的锁。在基于磁盘的数据库管理系统中，锁与元组分开存储，因此它们永远不会交换到磁盘。这种分离在内存数据库管理系统中是不必要的，因此使用MV2PL，锁嵌入在元组头中。元组的写锁是txn-id字段。对于读锁，DBMS使用read-cnt字段来计算已经读取元组的活动事务的数量。尽管不是必须的，DBMS可以将txn-id和read-cnt打包成连续的64位字，这样DBMS就可以使用一个CaS来同时更新它们。

为了在元组A上执行读取操作，DBMS通过比较事务的Tid和元组的开始字段来搜索可见版本。如果它找到了一个有效的版本，那么如果它的txn-id字段等于零(意味着没有其他事务持有写锁)，DBMS就递增该元组的read-cnt字段。类似地，只有当read-cnt和txn-id都设置为零时，事务才被允许更新版本Bx。当一个事务提交时，DBMS为它分配一个唯一的时间戳(Tcommit)，用于更新由该事务创建的版本的begin-ts字段，然后释放该事务的所有

锁。

2PL协议的关键区别在于它们如何处理死锁。先前的研究表明，无等待策略[9]是最具可扩展性的死锁预防技术[48]。这样，如果数据库管理系统无法获取元组上的锁，它会立即中止事务(而不是等待锁是否被释放)。由于事务从不等待，数据库管理系统不必使用后台线程来检测和打破死锁。

3.4 序列化证明 (Serialization Certifier)

在最后一个协议中，数据库管理系统维护一个序列化图，用于检测和移除由并发事务形成的“危险结构”[12，20，45]。人们可以在较弱的隔离级别上使用基于认证者的方法，这种方法提供了更好的性能，但允许某些异常。

提出的第一个证明者是可串行化快照隔离(SSI)[12]；这种方法通过避免快照隔离的写偏斜异常来保证可串行化。SSI使用事务的标识符来搜索元组的可见版本。只有当元组的txn-id字段设置为零时，事务才能更新版本。为了保证可串行化，DBMS在其内部图中跟踪反依赖边；当一个事务创建了一个新版本的元组，而另一个事务读取了它的先前版本时，就会发生这种情况。数据库管理系统为每个事务维护标志，跟踪入站和出站反依赖边的数量。当数据库管理系统在事务之间检测到两个连续的反依赖边时，它会中止其中一个。

串行安全网(SSN)是一个较新的基于认证者的协议[45]。与仅适用于快照隔离的SSI不同，SSN使用至少与READ COMMITTED一样强的任何隔离级别。它还使用了更精确的异常检测机制，减少了不必要的中止次数。SSN将事务相关性信息编码到元数据字段中，并通过计算低水位线来验证事务T的一致性，低水位线总结了在T之前提交但必须在T之后序列化的“危险”事务[45]。减少错误中止的数量使SSN更适合只读或只读事务的工作负载。

3.5 讨论

这些协议以不同的方式处理冲突，因此对于某些工作负载比其他工作负载更好。MV2PL用每个版本的读取锁记录读取。因此，在元组版本上执行读/写的事务如果试图在该版本上做同样的事情，将导致另一个事务中止。相反，MVTO使用read-ts字段来记录每个版本的读数。在读/写操作期间，MVOCC不会更新元组版本头上的任何字段。这避免了线程之间不必要的协调，读取一个版本的事务不会导致更新同一版本的其他事务中止。但是MVOCC要求数据库管理系统检查事务的读取集，以验证该事务读取操作的正确性。这可能导致长时间运行的只读事务缺乏资源[24]。认证协议减少了中止，因为它们不验证读取，但是它们的反依赖检查方案可能会带来额外的开销。

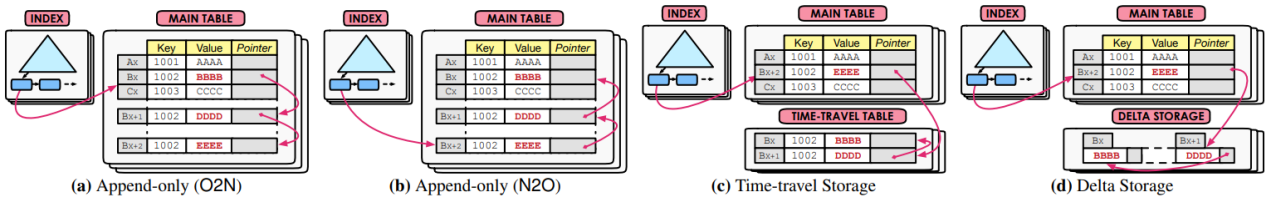
有一些优化上述方案的建议，以提高其对MVCC数据库管理系统的功效[10，27]。一种方法是允许事务推测性地读取由其他事务创建的未提交版本。代价是协议必须跟踪事务的读取依赖性，以保证可序列化的顺序。每个工作线程维护一个它读取其未提交数据的事务数量的依赖计数器。只有当事务的依赖计数器为零时，才允许提交事务，于是数据库管理系统遍历它的依赖列表，并为所有等待它完成的事务递减计数器。类似地，另一种优化机制是允许事务急切地更新未提交事务读取的版本。这种优化还要求数据库管理系统维护一个集中的数据结构来跟踪事务之间的依赖关系。只有当事务依赖的所有事务都已提交时，事务才能提交。

上述两种优化都可以减少某些工作负载不必要的中止次数，但它们也会受到级联中止的影响。此外，我们发现集中式数据结构的维护可能成为主要的性能瓶颈，这阻止了数据库管理系统向几十个内核扩展。

4 版本存储

在MVCC体系下，当事务更新元组时，数据库管理系统总是构建元组的新物理版本。DBMS的版本存储方案指定了系统如何存储这些版本以及每个版本包含什么信息。DBMS使用元组的指针字段来创建一个称为版本链的无锁链表。这个版本链允许数据库管理系统定位事务可见的元组的期望版本。正如我们下面讨论的，链的HEAD不是最新的就是最老的版本。

我们现在更详细地描述这些方案。我们的讨论集中在更新操作的方案权衡上，因为这是数据库管理系统处理版本控制的地方。数据库管理系统向表中插入新的元组，而不必更新其他版本。同样，DBMS通过在当前版本的begin-ts字段中设置一个标志来删除元组。在后续章节中，我们将讨论这些存储方案对DBMS如何执行垃圾回收以及如何维护索引中的指针的影响。



(a) Append-only (O2N) (b) Append-only (N2O) (c) Time-travel Storage (d) Delta Storage

Figure 3: Version Storage – This diagram provides an overview of how the schemes organize versions in different data structures and how their pointers create version chains in an in-memory MVCC DBMS. Note that there are two variants of the append-only scheme that differ on the ordering of the version chains.

4.1 仅追加（Append-only）存储

在第一种方案中，表的所有元组版本都存储在相同的存储空间中。这种方法用于Postgres，以及内存中的数据库管理系统，如Hekaton、NuoDB和MemSQL。为了更新现有的元组，数据库管理系统首先从表中为新的元组版本获取一个空槽。然后，它将当前版本的内容复制到新版本。最后，它将修改应用到新分配的版本槽中的元组。

只追加的方案的关键决定是DBMS如何排列图元的版本链。由于不可能维护一个无锁存器的双链表，版本链只指向一个方向。这种排序方式对数据库管理系统在事务修改元组时更新索引的频率有影响。

Oldest-to-Newest (O2N): 通过这种排序，链的头是元组的现存最老的版本(见图3a)。这个版本可能对任何活动事务都不可见，但是数据库管理系统还没有回收它。O2N的优点是，每当元组被修改时，DBMS不需要更新索引来指向元组的更新版本。但是在查询处理过程中，数据库管理系统可能会遍历一个很长的版本链来找到最新的版本。这是缓慢的，因为指针跟踪，它通过读取不需要的版本污染了CPU缓存。因此，用O2N实现良好的性能在很大程度上取决于系统修剪旧版本的能力。

Newest-to-Oldest (N2O): 另一种方法是将元组的最新版本存储为版本链的HEAD(见图3b)。由于大多数事务访问元组的最新版本，数据库管理系统不必遍历链。然而，缺点是每当元组被修改时，链的HEAD就会改变。然后，数据库管理系统更新表的所有索引(包括主索引和辅助索引)，以指向新版本。就像我们在第二节讨论的那样。6.1中，可以通过间接层来避免这个问题，间接层提供了一个将元组的最新版本映射到物理地址的位置。在这种设置下，索引指向元组的映射条目，而不是它们的物理位置。这对于具有许多辅助索引的表很有效，但是会增加存储开销。

仅追加存储的另一个问题是如何处理非线属性(例如。BLOBs)。考虑一个有两个属性(一个整数，一个BLOB)的表。当事务更新该表中的元组时，在仅追加模式下，数据库管理系统创建BLOB属性的副本(即使事务没有修改它)，然后新版本将指向该副本。这是一种浪费，因为它会创建冗余副本。为了避免这个问题，一个优化是允许同一个元组的多个物理版本指向同一个非内联数据。DBMS维护该数据的引用计数器，以确保只有当值不再被任何版本引用时，才会被删除。

4.2 时间旅行（Time-Travel）存储

下一个存储方案类似于只追加的方法，只是旧版本存储在一个单独的表中。数据库管理系统在主表中维护每个元组的主版本，在单独的时间旅行表中维护同一个元组的多个版本。在一些数据库管理系统中，如SQL Server，主版本是元组的当前版本。其他系统，如SAP HANA，存储元组的最早版本作为主版本，以提供快照隔离[29]。这在垃圾回收期间会产生额外的维护成本，因为数据库管理系统在修剪当前主版本时会将数据从时间旅行表复制回主表。为了简单起见，我们只考虑第一种时间旅行方法，其中主版本总是在主表中。

为了更新元组，数据库管理系统首先获取时间旅行表中的一个槽，然后将主版本复制到这个位置。然后，它修改存储在主表中的主版本。索引不受版本链更新的影响，因为它们总是指向主版本。因此，它避免了每当事务更新元组时维护数据库索引的开销，非常适合访问元组当前版本的查询。

这个方案也存在着与仅追加的方法相同的非内联属性问题。我们上面描述的数据共享优化在这里也适用。

4.3 Delta 存储

在最后一种方案中，DBMS在主表中维护图元的主版本，在单独的delta存储中维护一连串的delta版本。这种存储在MySQL和Oracle中被称为回滚段，在HyPer中也被使用。大多数现有的DBMS将元组的当前版本存储在主表中。为了更新一个现有的元组，DBMS从delta存储中获取一个连续的空间 从delta存储中获取一个连续的空间来创建一个新的delta版本。这个delta版本包含修改过的属性的原始值，而不是整个元组。而不是整个元组。然后DBMS直接执行原地的 更新到主表中的主版本。

该方案非常适合修改元组属性子集的更新操作，因为它减少了内存分配。但是，这种方法会导致读取密集型工作负载的开销更高。要执行访问单个元组的多个属性的读取操作，数据库管理系统必须遍历版本链，以获取该操作访问的每个属性的数据。

4.4 讨论

这些方案具有不同的特征，这些特征会影响它们对于OLTP工作负载的行为。因此，对于任何一种工作负载类型，它们都无法实现最佳性能。仅附加方案更适合执行大型扫描的分析查询，因为版本连续存储在内存中，这可以最大限度地减少CPU缓存未命中，是硬件预取的理想选择。但是访问元组的旧版本的查询会遭受更高的开销，因为数据库管理系统遵循元组的链来找到正确的版本。仅追加方案还向索引结构公开了物理版本，这允许附加的索引管理选项。

所有的存储方案都要求数据库管理系统从集中式数据结构(即表、delta存储)。多个线程将同时访问和更新这个集中式存储，从而导致访问争用。为了避免这个问题，数据库管理系统可以为每个集中式结构(即表、delta存储)并以固定大小增量扩展它们。然后，每个工作线程从单个空间获取内存。这实际上是对数据库进行分区，从而消除了集中的争用点。

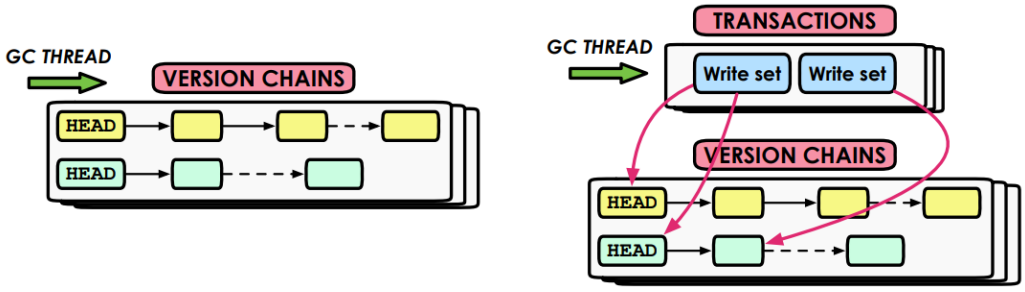
5 垃圾回收

由于MVCC在事务更新元组时会创建新版本，因此除非回收不再需要的版本，否则系统将耗尽空间。这也增加了查询的执行时间，因为数据库管理系统花费更多的时间遍历长版本链。因此，MVCC数据库管理系统的性能高度依赖于其垃圾回收组件以事务安全方式回收内存的能力。

垃圾回收过程分为三个步骤:(1)检测过期版本，(2)取消这些版本与其相关链和索引的链接，(3)回收它们的存储空间。如果版本是无效版本(即由中止的事务创建)或者它对任何活动事务不可见。对于后者，DBMS检查一个版本的结束时间是否小于所有活动事务的Tid。DBMS维护一个集中的数据结构来跟踪这些信息，但这是多核系统中的一个可伸缩性瓶颈[27，48]。

内存中的数据库管理系统可以通过跟踪事务创建的版本的粗粒度的epoch-based内存管理来避免这个问题[44]。总是有一个活动epoch和一个在先epoch的先进先出队列。一段时间后，数据库管理系统将当前活动epoch移动到前一epoch队列，然后创建一个新的活动epoch。这种转换要么由后台线程执行，要么由数据库管理系统的工作线程以协作方式执行。每个epoch包含分配给它的事务数量的计数。数据库管理系统将每个新事务注册到活动epoch，并递增该计数器。当一个事务完成时，数据库管理系统将它从它的纪元(可能不再是当前活动的纪元)中删除，并递减该计数器。如果非活动纪元的计数器达到零，并且所有以前的纪元也不包含活动事务，那么数据库管理系统回收在该纪元中更新的过期版本是安全的。

MVCC有两种垃圾回收实现，它们在数据库管理系统查找过期版本的方式上有所不同。第一种方法是元组级垃圾回收，其中数据库管理系统检查单个元组的可见性。第二个是事务级垃圾回收，它检查由完成的事务创建的任何版本是否可见。需要注意的一点是，并非我们下面讨论的所有垃圾回收方案都与每个版本存储方案兼容。



(a) Tuple-level **(b)** Transaction-level

Figure 4: Garbage Collection – Overview of how to examine the database for expired versions. The tuple-level GC scans the tables' version chains, whereas the transaction-level GC uses transactions' write-sets.

5.1 元组级垃圾回收

通过这种方法，数据库管理系统以两种方式之一检查每个单独元组版本的可见性:

后台清空(VAC): DBMS使用后台线程定期扫描数据库中的过期版本。如表1所示，这是MVCC数据库管理系统中最常见的方法，因为它更容易实现，并且适用于所有版本存储方案。但是这种机制不适用于大型数据库，尤其是具有少量GC线程的数据库。一种更具可扩展性的方法是，事务将无效版本注册到一个无锁的数据结构中[27]。然后垃圾回收线程使用上述epoch-based的方案回收这些过期的版本。另一个优化是DBMS维护一个脏块位图，这样真空线程就不会检查自上次垃圾回收以来没有修改过的块。

协同清理(COOP): 当执行一个事务时，DBMS遍历版本链来定位可见版本。在这个遍历过程中，它识别过期的版本，并将它们记录在一个全局数据结构中。这种方法可以很好地扩展，因为垃圾回收线程不再需要检测过期版本，但它只适用于O2N仅附加存储。一个额外的挑战是，如果事务没有遍历特定元组的版本链，那么系统将永远不会删除它的过期版本。这个问题在Hekaton [16]中被称为"dusty corners"。DBMS通过定期用一个单独的线程（就像在VAC中那样）执行一个完整的GC传递来克服这个问题。

5.2 事务级垃圾回收

在这种垃圾回收机制中，数据库管理系统以事务级粒度回收存储空间。它兼容所有版本存储方案。当事务生成的版本对任何活动的事务都不可见时，数据库管理系统认为事务已过期。在一个epoch结束后，由属于该epoch的事务生成的所有版本都可以被安全地移除。这比元组级垃圾回收方案更简单，因此它可以很好地与事务本地存储优化配合使用(见4.4节)，因为DBMS会一次性回收一个事务的存储空间。然而，这种方法的缺点是，数据库管理系统跟踪每个epoch的读/写事务集，而不是只使用epoch的成员计数器。

5.3 讨论

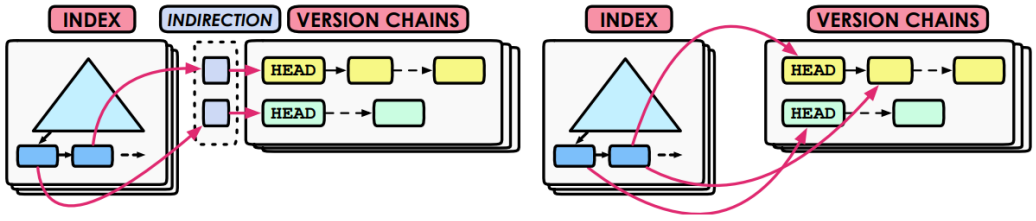
带有VAC的元组级GC是MVCC数据库管理系统中最常见的实现。在这两种方案中，增加专用GC线程的数量可以加速GC过程。在存在长期运行的事务时，数据库管理系统的性能会下降。这是因为在这样一个事务的生命周期中产生的所有版本在它完成之前不能被删除。

6 索引管理

所有MVCC数据库管理系统都将数据库的版本信息与其索引分开。也就是说，索引中某个键的存在意味着该键存在某个版本，但索引条目不包含元组匹配哪个版本的信息。我们将索引条目定义为键/值对，其中键是元组的索引属性，值是指向该元组的指针。数据库管理系统跟踪这个指向元组的版本链的指针，然后扫描该链以定位对事务可见的版本。数据库管理系统永远不会从索引中产生假阴性，但它可能会得到假阳性匹配，因为索引可以指向对特定事务不可见的密钥版本。

主键索引总是指向元组的当前版本。但是DBMS更新主键索引的频率取决于它的版本存储方案在元组更新时是否创建新版本。例如，delta方案中的主键索引总是指向主表中元组的主版本，因此索引不需要更新。对于仅追加，它取决于版本链顺序:N2O要求DBMS在每次创建新版本时更新主键索引。如果一个元组的主键被修改，那么数据库管理系统将它作为一个DELETE后跟一个INSERT应用于索引。

对于辅助索引来说，情况更复杂，因为索引条目的键和指针都可以改变。MVCC数据库管理系统中二级索引的两种管理方案在这些指针的内容上有所不同。第一种方法使用逻辑指针，使用间接映射到物理版本的位置。这与物理指针方法形成对比，物理指针方法的值是元组的精确版本的位置。



(a) Logical Pointers **(b)** Physical Pointers

Figure 5: Index Management – The two ways to map keys to tuples in a MVCC are to use logical pointers with an indirection layer to the version chain HEAD or to use physical pointers that point to an exact version.

6.1 逻辑指针

使用逻辑指针的主要思想是，DBMS使用一个固定的标识符，该标识符在其索引条目中的每个元组都不会改变。然后，如图5a所示，数据库管理系统使用间接层将元组的标识符映射到其版本链的头部。这避免了每当元组被修改时必须更新表的所有索引以指向新的物理位置的问题(即使索引属性没有改变)。每次只需更改映射条目。但是由于索引没有指向确切的版本，数据库管理系统从HEAD开始遍历版本链，寻找可见的版本。这种方法兼容任何版本存储方案。正如我们现在讨论的，这种映射有两种实现选择:

主键(PKey): 标识符与对应元组的主键相同。当数据库管理系统从二级索引中检索一个条目时，它在表的主键索引中执行另一次查找，以定位版本链头。如果辅助索引的属性与主键重叠，那么数据库管理系统就不必在每个条目中存储整个主键。

元组 Id(TupleId): PKey指针的一个缺点是，随着元组主键大小的增加，数据库的存储开销也会增加，因为每个辅助索引都有一个完整的副本。除此之外，由于大多数数据库管理系统为其主键索引使用保序数据结构，执行额外查找的成本取决于条目的数量。另一种方法是使用唯一的64位元组标识符代替主键，并使用独立的无锁哈希表来维护元组版本链头的映射信息。

6.2 物理指针

使用第二种方案，数据库管理系统将版本的物理地址存储在索引条目中。这种方法只适用于仅追加存储，因为数据库管理系统将版本存储在同一个表中，因此所有的索引都可以指向它们。当更新表中的任何元组时，数据库管理系统将新创建的版本插入到所有辅助索引中。通过这种方式，数据库管理系统可以从二级索引中搜索元组，而无需将二级关键字与所有索引版本进行比较。几个MVCC数据库管理系统，包括MemSQL和Hekaton，采用了这个方案。

6.3 讨论

与其他设计决策一样，这些索引管理方案在不同的工作负载上表现不同。逻辑指针方法更适合写密集型工作负载，因为只有当事务修改索引属性时，数据库管理系统才会更新辅助索引。然而，读取可能会更慢，因为数据库管理系统遍历版本链并执行额外的键比较。同样，使用物理指针更适合读取密集型工作负载，因为索引条目指向确切的版本。但是它对于更新操作来说速度较慢，因为这个方案要求数据库管理系统为每个新版本在每个辅助索引中插入一个条目，这使得更新操作速度较慢。

最后一个有趣的点是，除非元组的版本信息嵌入到每个索引中，否则在MVCC数据库管理系统中只进行索引扫描是不可能的。系统必须总是从元组本身检索这些信息，以确定每个元组版本是否对事务可见。NuoDB通过将头元数据与元组数据分开存储，减少了为检查版本而读取的数据量。

7 实验分析

现在我们介绍一下我们对本文讨论的事务管理设计选择的分析。我们尽力地在Peloton DBMS[5]中实现它们中的每一个最先进的版本。Peloton在面向行的、无序的内存堆中存储图元。它的内部数据结构使用libcuckoo[19]哈希表，数据库索引使用Bw-Tree[32]。我们还通过利用无锁编程技术[15]来优化Peloton的性能。我们在SERIALIZABLE隔离级别下以存储过程的形式执行所有事务。我们将Peloton配置为使用基于历时的内存管理（见第5节），历时40毫秒[44]。

我们将Peloton部署在一台4-socket的英特尔至强E7-4820服务器上，该服务器有128GB的DRAM，运行Ubuntu 14.04（64位）。每个socket包含10个1.9GHz的内核和25MB的L3缓存。

我们从并发控制协议的比较开始。然后，我们选择最佳的整体协议，并使用它来评估版本存储、垃圾回收和索引管理方案。对于每次试验，我们执行工作负载60秒，让数据库管理系统预热，并在120秒后测量吞吐量。我们执行每个试验五次，并报告平均执行时间。我们把我们的发现总结在第8节。

7.1 Benchmark

YCSB: 我们修改了YCSB[14]基准，以模拟OLTP应用程序的不同工作负载设置。数据库包含一个包含1000万元组的表，每个元组有一个64位主键和10个64位整数属性。每个操作都是独立的；也就是说，一个操作的输入不依赖于前一个操作的输出。我们使用三种工作负载混合来改变每个事务的读取/更新操作数:(1)只读(100%读取)，(2)读密集型(80%读取，20%更新)，以及(3)更新密集型(20%读取，80%更新)。我们还改变了操作在元组中读取或更新的属性数量。这些操作按照Zipfian分布访问元组，该分布由影响争用量的参数(θ)控制(即偏斜)，其中 θ =1.0是最高偏斜设置。

TPC-C: 该基准是测量OLTP系统性能的当前标准[43]。它用九个表和五种交易类型对一个以仓库为中心的订单处理应用程序进行建模。我们修改了原始的TPC-C工作负载，以包括一个新的表扫描查询，称为库存扫描，它扫描库存表并计算每个仓库中的项目数量。工作负载中的争用量由仓库数量控制。

说明：第七节余下的四小节对应第3到6节介绍的主题（即并发控制协议、版本存储、垃圾回收、索引管理），对每个主题中提到的技术分别进行了测试和对比分析，内容主要是对比实验的描述和大量结果图表，在此不再给出中文翻译。

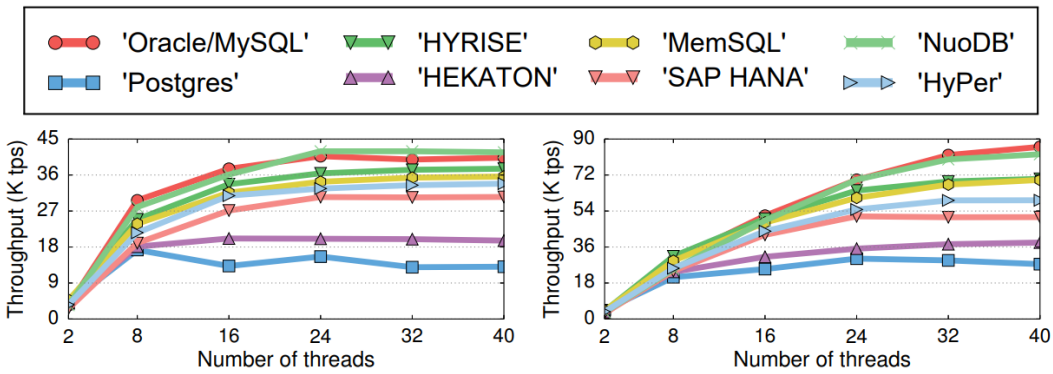
8 讨论

我们在MVCC数据库管理系统中对这些事务管理设计方案的分析和实验产生了四个发现。最重要的是，版本存储方案是在多核环境中扩展内存中MVCC数据库管理系统的最重要组件之一。这违背了数据库研究中的传统智慧，传统智慧主要集中在优化并发控制协议上[48]。我们观察到，只追加和时间旅行方案的性能受到底层内存分配方案效率的影响；积极地划分每个内核的内存空间解决了这个问题。无论内存分配如何，delta存储方案都能够保持相对较高的性能，尤其是当表中存储的属性只有一个子集被修改时。但是该方案存在表扫描性能低的问题，可能不太适合读取量大的分析工作负载。

我们接下来展示了使用适合工作负载的并发控制协议可以提高性能，尤其是在高争用工作负载上。第二节的结果。7.2节的结果表明协议优化会损害这些工作负载的性能。总的来说，我们发现MVTO在各种工作负载下都能很好地工作。我们在表1中列出的系统都没有采用这种协议。

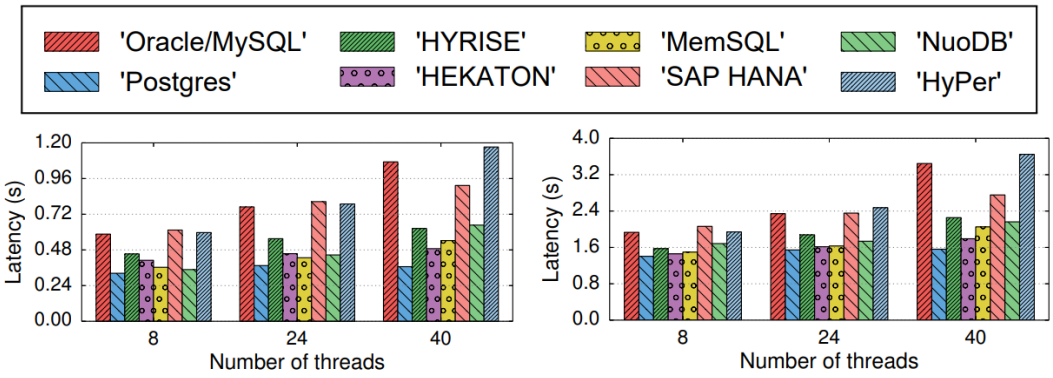
我们还观察到，MVCC数据库管理系统的性能与其气相色谱实现密切相关。特别是，我们发现事务级垃圾回收以最小的内存占用提供了最好的性能。这是因为它以比其他方法更低的同步开销回收过期的元组版本。我们注意到垃圾回收过程会导致系统吞吐量和内存占用量的波动。

最后，我们发现索引管理方案也会影响数据库管理系统的性能，因为已经构建了许多二级索引。7.5节的结果表明，逻辑指针方案总是能够获得更高的吞吐量，尤其是在处理更新密集型工作负载时。这证实了工业界关于这个问题的其他报告[25]。



(a) 10 warehouses **(b)** 40 warehouses

Figure 24: Configuration Comparison (Throughput) – Performance of the MVCC configurations from Table 1 with the TPC-C benchmark.



(a) 10 warehouses **(b)** 40 warehouses

Figure 25: Configuration Comparison (Scan Latency) – Performance of the MVCC configurations from Table 1 with the TPC-C benchmark.

为了验证这些发现，我们对Peloton进行了最后一次实验，在实验中，我们将其配置为使用表1中列出的MVCC配置。我们执行TPC-C工作负载，并使用一个线程重复执行斯托克扫描查询。我们测量数据库管理系统的吞吐量和股票扫描查询的平均延迟。我们承认，在本实验中，我们没有捕捉到真实数据库管理系统中的其他因素(例如数据结构、存储架构、查询编译)，但这仍然是他们能力的一个很好的近似。

如图24所示，使用Oracle/MySQL和NuoDB配置，数据库管理系统在低争用和高争用工作负载上表现最佳。这是因为这些系统的存储方案在多核和内存系统中可以很好地扩展，并且它们的MV2PL协议提供了相对较高的性能，而与工作负载争用无关。HYRISE、MemSQL和HyPer的配置产生相对较低的性能，因为MVOCC协议的使用会带来高开销，因为验证阶段需要读取集遍历。Postgres和Hekaton的配置导致性能最差，主要原因是使用O2N排序的仅追加存储严重限制了系统的可扩展性。实验表明，并发控制协议和版本存储方案都会对吞吐量产生很大影响。

但是图25中的延迟结果表明，使用delta存储时，数据库管理系统的性能最差。这是因为delta存储必须花费更多的时间遍历版本链，以便找到目标元组版本属性。

9 相关工作

第一次提到MVCC是在里德1979年的论文中[38]。之后，研究人员专注于理解MVCC在基于单核磁盘的数据库管理系统中的理论和性能[9，11，13]。我们主要关注近期的研究工作。

并发控制协议：有几个工作提出了优化内存事务处理的新技术[46，47]。拉森等人。[27]比较微软Hekaton DBMS早期版本中的悲观(MV2PL)和乐观(MVOCC)协议[16]。Lomet等人。[31]提出了一种使用时间戳范围来解决事务间冲突的方案。[18]将MVCC的并发控制协议和版本管理与数据库管理系统的事务执行分离开来。考虑到在保证MVCC可串行化方面的挑战，许多数据库管理系统转而支持更弱的隔离级别，称为快照隔离[8]，它不排除写偏斜异常。可序列化快照隔离(SSI)通过消除快照隔离中可能发生的异常来确保可序列化性[12，20]。Kim等人[24]使用SSN在异构工作负载上扩展MVCC。我们在这里的研究范围更广。

版本存储：MVCC数据库管理系统的另一个重要设计选择是版本存储方案。赫尔曼等人。[23]提出一种事务管理的差分结构，以在不影响读取性能的情况下实现高写入吞吐量。Neumann等人。[36]通过事务本地存储优化提高了MVCC数据库管理系统的性能，以降低同步成本。这些方案不同于传统的只追加版本存储方案，后者在主存数据库管理系统中存在较高的内存分配开销。Arulraj等人[7]在运行异构工作负载时，检查物理设计对混合数据库管理系统性能的影响。

垃圾回收：大多数数据库管理系统采用元组级后台清空垃圾回收方案。Lee等人[29]评估现代数据库管理系统中使用的一组不同的垃圾回收方案。他们提出了一种新的混合方案来缩小SAP HANA中的内存占用。Silo的epoch-based内存管理方法允许数据库管理系统扩展到更大的线程数[44]。这种方法仅在一个epoch(以及之前的epoch)不再包含活动事务之后回收版本。

索引管理：最近，提出了新的索引数据结构来支持可扩展的主存数据库管理系统。Lomet等人[32]引入了一种无锁的保序索引，称为Bw-Tree，目前在微软的一些产品中使用。Leis等人。[30]和Mao等人[34]分别提出了ART和Masstree，它们是基于尝试的可伸缩索引结构。这项工作没有考察不同指数结构的绩效，而是侧重于不同的二级指数管理方案如何影响MVCC数据库管理系统的性能。

10 总结

我们评估了in-memory MVCC事务管理设计的各种方案。我们描述了其中每一个的SOTA实现方式，并展示了它们在现有系统中的使用方式。然后，我们在Peloton DBMS中实现了它们，并使用OLTP工作负载对它们进行了评估，以突出其权衡。我们展示了阻碍DBMS支持更大CPU内核数量和更复杂工作负载的问题。

鸣谢：这项工作得到了国家科学基金会（CCF-1438955）和三星奖学金项目的支持。我们也感谢Tianzheng Wang的反馈。