内存数据库并发控制算法的研究

李程鹏[1)]

[1)](华中科技大学电子信息与通信学院 湖北 武汉 430074)

摘 要 随着互联网技术的发展，数据规模近年来呈指数级别增长，这对现有数据库管理系统提出了更高的要求。以往的磁盘数据库受限于磁盘 I/O 难以满足性能和可扩展性要求，与此同时内存数据库变得越来越流行。内存数据库从硬件层面避免了磁盘 I/O 的限制，但也使数据库瓶颈转移到多核 CPU 争用。并发控制算法一直是保证数据库系统正确和高效执行的保证，是解决多核 CPU 争用核心设计之一。本文对现有主流的并发控制算法进行调研和简要介绍，并根据相关文献中的思想建立了一个“定序+检验”的设计框架进行归纳总结，为后续研究提供参考思路。

关键词 内存数据库 并发控制算法 事务 扩展性 串行性

A study of concurrency control algorithms for in-memory databases

Chengpeng Li[1)]

[1)](School of Electronic Information and Communication, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

Abstract With the development of Internet technology, data size has been growing exponentially in recent years, which puts higher requirements on existing database management systems. Previous disk databases are limited by disk I/O difficult to meet the performance and scalability requirements, while in-memory databases are becoming more and more popular. In-memory databases avoid the disk I/O limitations at the hardware level, but also shift the database bottleneck to multi-core CPU contention. Concurrency control algorithms have always been a guarantee for correct and efficient execution of database systems, and are one of the core designs for solving multi-core CPU contention. This paper investigates and briefly introduces the existing mainstream concurrency control algorithms, and establishes a design framework of "sequencing + checking" based on the ideas in the related literature to summarize and provide reference ideas for the subsequent research.

Key words In-memory databases Concurrency control algorithms Transactions Scalability serialization

1 引言

随着互联网技术的飞速发展，数字资源的规模正呈指数级别爆炸式增长，全球已经进入大数据时代。根据国际数据公司 IDC 的预测，2025 年数据规模将从 2018 年的 33ZB 增长到 175ZB，平均年增长率达到 61%，并且约 30%的全球数据需要进行实时处理。这给数据底层存储和处理带来了巨大的挑战，对数据库管理系统的性能和可扩展性均提出了更高的要求。

传统基于磁盘的数据库的性能受限于磁盘 I/O 访问延迟，性能难以满足海量数据的实时访问需求。近年内存价格的下降和容量的上升，内存数据库这一构想得以广泛实现，变得越来越流行。内存数据库系统突破了传统系统中磁盘 I/O 的性能瓶颈，采用 RAM 而不是硬盘驱动器（HDD）或固态驱动器（SSD）来存储数据，使得访问数据延迟大大减小，为高效的数据处理和分析带来了新的机遇。

内存数据库相比于磁盘数据库主要有两大优点：（1）内存芯片与 CPU 的接口性能更强，访问速度能够达到 SSD 的近 30 倍；（2）内存数据库一般可以采用非结构化或半结构化的存储格式，更加直接、更易扩展。但同时内存数据库也面临着新的挑战，传统数据库管理系统架构以最小化磁盘 I/O 算

法提升性能，难以直接迁移到内存数据库上，需要设计新的架构用以进一步提升性能和扩展性。

内存数据库已经将性能瓶颈从存储转移到多核 CPU 争用，现有提高内存数据库可扩展性的研究可以集中于在三个瓶颈上：范围索引结构、低竞争和高竞争（有依赖关系的事务）工作负载的序列化事务隔离。事务隔离性的保证离不开对并发控制算法的研究，并发控制算法的目的是对事务进行合理调度，从而保证事务执行的正确性，其直接影响着事务处理的性能。

本文聚焦于并发控制算法，对现有主流的并发控制算法进行调研梳理，引出了算法设计的基本思想，总结归纳出各算法的优缺点和适用场景。第二节首先阐述了并发处理算法的相关背景知识以及内存数据库的相关研究现状。第三节引出了并发控制算法的基本思想，将主流并发控制算法放在同一框架进行描述与讨论。第四节紧接着具体介绍相关并发控制算法，本文主要介绍了两阶段封锁、乐观并发控制、时间戳排序、快照隔离等主流算法。第五节总结了全文，并对该领域的未来值得关注的研究方向提出了展望。

2 背景

内存数据库现在变得越来越流行，表 1 总结了目前主流商用内存数据库系统的特点。

表 1 主流的内存数据库系统

内存数据库名称	产商	内存数据库特点
ExtremeDB	McObject	ExtremDB 为实时嵌入式系统设计；所有数据在内存中，不基 于文件系统；支持事务和部分 SQL；通过数据库定义语言为 应用系统提供各自的 API，具有工业应用强度
Redis	开源	Reids 是一个高性能的键值对数据库系统；支持多种数据结构；简单稳定；支持数据持久化和备份；
SQLite	开源	SQLite 为嵌入系统设计；提供了多种语言接口；支持 SQL 语句和事务处理；资源占用低；支持跨平台操作；利用内存和磁盘间的同步操作持久化数据
FastDB	开源	FastDB 是高效的内存数据库系统；支持事务、备份和系统崩溃后的自动恢复；支持类似 SQL 语言并提供了 C++接口

2.1 事务及ACID要求

事务是用户定义的一系列可执行数据库操作的集合，是数据库管理系统中最小执行单元。事务概念的提出推动了数据库的应用，使其成为了现代信息系统不可或缺的基础设施。事务的 ACID（原子性、一致性、隔离性、持久性）四大属性，定义了数据库系统事务处理的基本要求，成为了数据库设计时必须遵循的准则。下面进行简要阐述。

（1）原子性（Atomicity）：一个事务必须被视为一个不可分割的最小工作单元，即整个事务中所有操作要么全部成功，要么全部失败，对于一个事务来说不可能只执行其中的一部分操作；

（2）一致性（Consistency）：数据库总是从一个一致性状态转换到另一个一致性状态，不允许未提交事务的临时数据被访问；

（3）隔离性（Isolation）：要求事务的执行不能影响到其他事务也不能被其他事务影响，即在提交之前，事务之间是相互不可见的；

（4）持久性（Duration）：要求数据库持久化存储已提交事务写入的数据。一旦事务提交所有修改都会被永久保存到数据库。

支持并发访问是现代数据库的基本需求，为了保证事务的 ACID 特性，会引入故障恢复系统来保证原子性和持久性，并引入并发控制子系统来保证一致性和隔离性。

2.2 事务隔离级别

为了增大吞吐量并减小响应时间，通常会允许多个事务并发执行，但在并发访问数据库，多事务常常会产生多类问题，主要有脏读、丢失修改、可重复读、幻读。

（1）脏读（Dirty Read）：当一个事务正在访问数据并对数据进行了修改，而这个修改还没有提交到数据库中，这时另外一个事务也访问了这个数据并使用了该数据。因为这个数据是还没有提交的数据，另一个事务读取到的数据为“脏数据”，根据“脏数据”做出的操作可能是错误的。

（2）不可重复读（Unrepeatableread）：一个事务内多次读同一数据。在该事务处理的同时，另一事务也访问该数据。那么，在第一个事务中的两次

读数据之间，第二个事务的修改会导致两次读取的数据不同。

（3）幻读（Phantom read）：与不可重复读类似，一个事务查询读了几行数据，接着另一个并发事务插入了一些数据，导致第一个事务在后续的查询中发现多了原本不存在的记录。

（4）丢失修改（Lost to modify）:在一个事务读取并修改一个数据时，另一个事务也访问修改了该数据导致第一个事务的修改被第二个事务覆盖，也即丢失修改。另一类情况是第一个事务在时候进行回滚会导致第二个事务的修改被覆盖丢失。

根据以上可能问题，SQL 标准定义了以下四个隔离级别：

（1）读未提交（Read-uncommitted）：最低的隔离级别，允许读取尚未提交的数据变更；

（2）读已提交（Read-committed）：允许读取并发事务已经提交的数据，能有效防止脏读；

（3）可重复读（Repeatable-read）：对同一字段的多次读取结果都是一致的，除非数据是被本身修改的，能有效阻止脏读以及不可重复读；

（4）可串行化（Serializable）：最高的隔离级别完全服从 ACID 特性，所有事务依次逐个执行，事务之间完全不可能产生干扰。

数据库管理系统的并发控制算法追求在保证可串行化的隔离级别的前提下，保证事务处理的高效性。

2.3 现有研究

随着数据量的规模指数级增加，现代关系型数据库的可扩展需求面临着挑战。内存数据库的可扩展性瓶颈主要集中于范围索引结构以及不同工作负载下的事务隔离上。

经过几十年的研究，范围索引在并发访问下的可扩展性仍然难以捉摸，这主要是由于这些数据结构的分层性质。在支持多线程并发访问的同时需要自动同步执行这些修改。文献[2]提出了一个基于异步并发控制的可扩展性多核内存事务数据库 ScaleDB。通过将事务执行与范围索引更新解耦，ScaleDB 可以专注于提高事务执行与索引更新的可扩展性，而不需要在性能上进行不必要的权衡。ScaleDB 通过避免共享数据结构上不必要的争用为 ACID 事务提供了可扩展的序列化隔离，并具有高吞吐量、低提交延迟和低中止率。

事务隔离的核心便是并发控制算法，当前并发控制算法设计的设计面临着两大挑战。一是如何在保证可串行化隔离这一事务正确调度黄金法则的基础上，让事务处理更加高效，这对并发控制算法的逻辑设计提出了较高的要求；二是现有并发控制算法均以最小化磁盘 I/O 为目标，而内存数据库颠覆了这一要求，并发控制算法的架构设计要求更加贴近内存数据库的现状和相关问题。

近年来，研究者们对并发控制算法的研究都期望着能够解决上述两大挑战。文献[3]在两阶段封锁(two-phase locking， 2PL)算法的基础上减少了锁等待时间，优化了 2PL 的算法逻辑，提升了事务并发度，从而提升性能;文献[7]通过引入动态时间戳调整，降低事务回滚率，优化了乐观并发控制(Optimisitic Concurrency Control, OCC)算法的逻辑；文献[8]引入了 Cache 机制，在内存型分布式数据库上表现优秀。文献[6]提出了一种混合并发控制算法的优化。而随着越来越多并发控制算法的提出，由于算法描述方式各异，导致并发控制算法理解和实现成本较高。因此，在内存数据库设计与实现时，针对特定场景对并发控制算法进行选择显得尤为困难。从 20 世纪 80 年代起，许多综述研究对并发控制算法进行了分析和归类。最近的工作文献[9]中 Deneva 也测试分析了 6 种并发控制算法在分布式场景下的表现情况。2020 年，文献[4]和文献[5]在单机场景下实验分析了算法的性能，同时研究了影响算法性能的主要因素和优化方法。

3 并发控制算法的基本思想

并发控制算法是一种事务合理调度的方法，用于实现数据库系统的可串行化隔离，保证事务执行的正确性，规避并行执行事务引起的数据异常。我们在第二小节中已经列举了事务并发处理过程中可能出现的几类数据异常，并介绍了 SQL 标准所定义的四种隔离级别。

并发控制算法设计的黄金法则便是得以实现可串行化事务隔离，最简单的实现方法便是单线程的串行处理事务，但由于吞吐量和处理时延的需求，现代数据库需要并发响应，绝大多数时间需要处理多核多线程的事务要求。评估一个并发控制算法是否能够满足可串行化的要求便是判断该算法是否能够与某一串行执行事务的结果等价，如果等价，则认为按照该算法调度事务是可以避免所有数据异常的，即具有可串行性。

信息社会的到来使得数据库也发展十分迅速，

现有并发控制算法种类繁多，实现的机制也大不相同，这给新型算法的创新设计带来了很大的挑战。文献[1]总结了主流的并发控制算法的逻辑共性，构建了一个便于归纳和发散的并发控制算法的核心思想：“定序+检验”。

3.1 定序

定序是指为并发事务确定一个等价可串行化调度的先后顺序。根据对各算法的调研总结，可以将主流算法分为两种定序模型：静态定序和动态定序。主流算法的定序方式如表 2 所示。

表 2 主流算法定序方式

算法经典分类	算法名称	定序方式
基于锁的算法	2PL	获取锁先后顺序
基于锁的算法	MV2PL	获取锁先后顺序
乐观算法	OCC	事务进入验证阶段的时间顺序
基于时间戳的算法	T/O	事务开始时间戳
基于时间戳的算法	MVTO	事务开始时间戳
基于时间戳的算法	SSI	事务提交时间戳
基于时间戳的算法	WSI	事务提交时间戳
乐观算法	MaaT	动态定序
乐观算法	Sundial	动态定序
乐观算法	Silo	事务进入验证阶段的时间顺序
乐观算法	Cicada	事务开始时间戳
确定性的算法	Calvin	事务开始时间戳

静态定序，即按照事务进入某个处理阶段的先后顺序来确定顺序。使用静态定序的典型算法有 T/O、OCC 等。例如 T/O 算法，事务 T1 先于事务 T2 开始，所以 T/O 算法会规定 T1 排在 T2 前面。静态定序较为简单，却过于严格，其要求事务的执行顺序必须与预先确定的某一种串行序列等同，实际上在事务的执行过程中，可能能会满足另一种串行序列。此时，也不会出现数据异常但按照静态定序会出现不必要的回滚，从而影响数据库整体性能。因此近年的提出的算法如 MaaT、Sundial 等，均采用了动态定序的方式。

动态定序指的是：根据执行过程中的操作结果，动态地调整事务最终贴近的串行化执行序列的顺序关系。其调整的依据便是数据中的 3 种依赖：读写依赖、写读依赖和写写依赖。

（1）读写依赖：如$R_1(X_0)W_2(X_1)$，事务 T1 读取了数据的版本 X0，事务 T2 写了一个新版本 X1，那么可以称事务 T2 读写依赖于事务 T1，记为$T_1 \xrightarrow{rw} T_2$

（2）写读依赖：如$W_1(X_1)R_2(X_1)$，事务 T1 写了一个数据新版本X_1，事务 T2 读取了数据的新版本X_1，那么可以称事务 T2 写读依赖于事务 T1，记为$T_1 \xrightarrow{wr} T_2$

（3）写写依赖：如$W_1(X_1)W_2(X_2)$，即事务 T1 写了一个数据的新版本 X1，事务 T2 之后又写了一个新版本 X2，那么可以称事务 T2 写写依赖于事务 T1，记为$T_1 \xrightarrow{ww} T_2$

注：上述事务T_n读取数据 X 的第 k 号版本记为$R_n(X_k)$，事务T_n写数据 X 的新版本，版本号为 k，记为$W_n(X_k)$

上述三种依赖具有严格的顺序要求，在实际并发控制中可以根据依赖关系动态的确定事务的先后顺序。但动态定序必然需要增加资源开销用于监控事务之间的依赖关系，以降低不必要的回滚率。具体实际不同的算法中采用动态还是静态需要在资源开销与低回滚率之间进行权衡。

3.2 检验

检验指检查事务的实际操作是否满足定序步骤中规定的事务先后顺序。其核心问题有如何检验和何时检验。下面将根据对相关算法的调研，分别从这两方面进行说明。

3.2.1 如何检验

关于如何检验的问题，文献[1]归纳总结出了 3 种方法：检验冲突、检验依赖、两者结合。主流算法的检验方式如表 3 所示。

表 3 主流算法检验方式

算法名称	检验方式
2PL	检验冲突
MV2PL	检验冲突
OCC	检验冲突
T/O	检验依赖
MVTO	检验依赖
SSI	两者结合
WSI	检验依赖
MaaT	检验依赖
Sundial	两者结合
Silo	检验冲突
Cicada	检验冲突
Calvin	检验冲突

检验冲突的方法较为简单，其关注事务执行过程中是否存在并发事务的冲突操作与自己访问了

相同的数据项。如果出现冲突操作，只能通过等待或回滚来解决，并不关心实际操作过程中自己与并发事务形成的先后关系。常见检查的冲突有读写冲突与写写冲突。经典的算法 2PL 和 OCC 均是采用检测冲突的方式，2PL 通过对数据加锁实现，数据一旦被一个事务加上写锁，则不能再被其他事务获得读锁或写锁，只能等待事务释放锁。OCC 在验证阶段会进行检验，如果其他事务的写集与自己的读集合冲突，则认为有冲突验证失败选择回滚。

检验依赖通过检验事务执行操作产生的数据依赖关系和事务规定的顺序是否相同，来判断是否需要回滚。主流算法中采用检验依赖的有 T/0 算法、MaaT 算法等。T/O 算法以事务开始时间决定顺序，如图 1 所示，事务 T1 在写数据项 X 时，发现数据 X 的 wts 已经被后开始的 T2 事务修改，此时$T_2 \xrightarrow{ww} T_1$，与 T/O 中事务开始顺序相反，所以 T1 进行回滚。

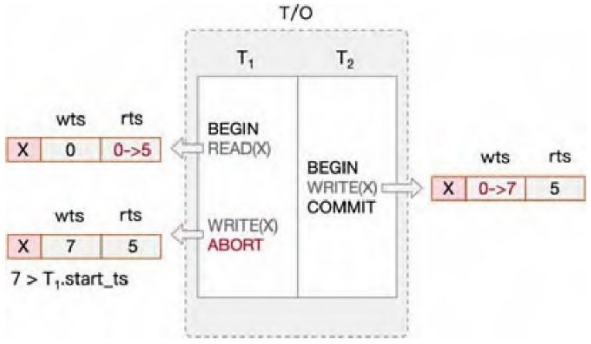

图 1 T/O 检验例子

冲突与依赖之间存在相互区别。冲突只关注并发事务是否存在对相同数据项的冲突操作、依赖则关注两个事务之间的相互顺序；两者的检验思路也不同，检验冲突是通过规避和解决冲突来实现可串行化，检查依赖是根据事务的先后关系推断出的依赖关系整理出一个可串行化序列。检验冲突逻辑较为简单，实现较为方便，但条件过于严苛会导致假冲突的出现，检验依赖的方法精度较高，可以减少假回滚但难度较高，占用资源较大。

因此，近年来的一些并发控制算法如 Sundial 考虑采用检测冲突和检测依赖相结合的方式进行检验。

- 对于写写冲突，利用加锁来解决
- 对于写读依赖，利用预写操作来解决，只有提交时才会更新数据项，写读依赖指挥发生在当前事务与已提交事务之间，无须处理
- 对于读写依赖，利用数据中时间戳信息来检查是否存在

3.2.2 何时检验

参考 OCC 算法，一般可以分为 3 个阶段：

（1）读写操作阶段，事务在这个阶段执行读写操作

（2）验证阶段，检查事务是否可以提交

（3）结束阶段执行事务的提交或回滚

检验一般集中于读写操作阶段和验证阶段，验证阶段最具代表的便是 OCC 算法。

4 并发控制算法

4.1 两阶段封锁算法（2PL）

4.1.1 算法介绍

2PL 在基于磁盘的数据库上应用广泛，是一种悲观的并发控制算法，其主要目的是减少回滚率以减少磁盘 I/O 的次数保证性能。该算法采用的是静态定序+检验冲突的方法组合。

锁是 2PL 实现事务调度可串行性的基本工具，在 DBMS 中锁通常分为两种，locks 和 latches，二者的区别如图 2 所示。我们主要关注事务级别的锁，即 locks。Locks 有两种基本类型，共享锁（S-Lock）与互斥锁（X-Lock），又称读锁与写锁。两个事务只有获取共享锁不会产生冲突，读锁与写锁、写锁之间相互冲突。

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

图 2 lock 和 latch 区别对比

DBMS 中存在一个专门的模块 lockmanager 用于管理系统中的 locks，事务需要加锁时均需要向其发起请求。磁盘数据库的缓冲层会维护一个 lock table 用于存储锁的分配信息，内存数据库由于缓冲层的缺失，锁信息一般会与记录绑定存储。

2PL 分为两个阶段，增长阶段（growing）和缩减阶段（shrinking）。增长阶段，事务可以按需获取某条数据的锁，lock manger 同意或拒绝；缩减阶段，事务只能释放之前获取的锁，不能获得新锁。当每个事务在结束前，其写过的数据不能被其它事务读取或者重写，即一直保持数据的锁直到事务结束。这种算法是 2PL 的增强版 Rigorous 2PL，可以避免级联中止，但并发程度会更差。

2PL 无法避免的一个问题就是死锁，如图 3 所示，事务 T1 获取数据 X 的读锁，同时事务 T2 获

取数据 Y 读锁。事务 T1 下一步需要对 X 进行写，等待事务 T2 释放 Y 的锁；事务 T2 下一步需要对 X 进行写，等待事务 T1 释放 X 的锁。两个事务之间互相等待对方释放自己想要的锁。

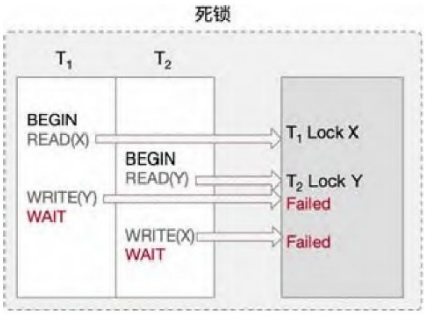

图 3 死锁

4.1.2 死锁机制

死锁通常有两类解决方式，事后检测（detection）与事前阻止（prevention）。

死锁检测是一种事后行为。为了检测死锁，DBMS 会维护一张 waits-for graph，来跟踪每个事务正在等待（释放锁）的其它事务，然后系统会定期地检查 waits-for graph，看其中是否有成环，如果成环了就要决定如何打破这个环。检测到死锁之后会选择一个“受害者”事务进行回滚，这里又有 WAIT-DIE、WOUND-WAIT 等方式。WAIT-DIE 机制根据不同事务的开始时间戳作为加锁优先级的依据，例如图 4 中，T1 事务时间戳早于 T2 事务，会让 T2 事务进行回滚便于 T1 事务继续处理。WOUND-WAIT 机制与 WAIT-DIE 机制相似。在检测到冲突时，会根据事务之间的优先级选择处理方式，即等待(WAIT)或抢占锁(WOUND)。若当前事务的优先级较高，则当前事务抢占锁并回滚之前的锁拥有者，否则，当前事务需要等待。

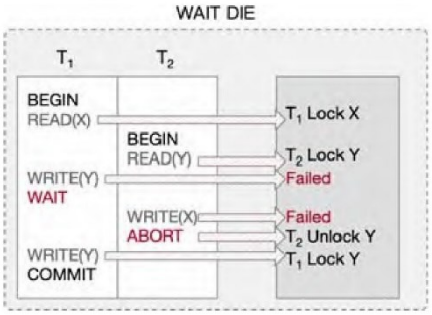

图 4 WAIT-DIE 机制

死锁阻止是一种事前行为，采用这种方案的 DBMS 无需维护 waits-for graph，也不需要实现 detection 算法，而是在事务尝试获取其它事务持有的锁时直接决定是否需要将其中一个事务中止。通常会按照事务的年龄来赋予优先级，事务越先开始，优先级越高。Old waits for young 和 Young waits for old 两种算法均能保证死锁的阻止。

从整体上来看, 2PL 存在方法简单、节省内存空间、元信息维护开销小这 3 个优势，但是也存在静态定序+检验冲突导致的高回滚率问题。因此，2PL 类算法不适用于冲突率较高的场景.。

4.2 乐观并发控制算法（OCC）

OCC 最早于 1981 年提出，该算法认为大多数情况下不需要竞争资源，只需要在提交时检查是否存在冲突，存在冲突就回滚，没有就提交。其只会在提交之前进行验证，检测事务之间是否存在读写/写写冲突。该算法提出时思维较为新颖，但性能较差。因此后续提出了许多优化的算法，本小节中也会逐个介绍。

4.2.1 传统乐观并发控制算法

传统 OCC 是第三节中静态定序+检验冲突这一组合，回滚率较高。

乐观并发算法规定：访问对象时不做检查操作、只在事务提交阶段检测冲突、若存在冲突则放弃一些事务。OCC 中将事务处理划分为 3 个阶段：工作阶段、验证阶段、更新阶段。

（1）工作阶段：每个事务拥有所修改对象的临时版本，当事务放弃时不会产生副作用，临时值（写操作）对其他事务不可见；每个事务维护访问对象的两个集合，读集合和写集合；

（2）验证阶段：发起结束事务的请求，判断是否存在与其他事务存在冲突，若不存在冲突则验证成功允许提交，验证失败则放弃当前事务或者冲突的事务

（3）更新阶段：只读事务通过验证立即提交，写事务在对象的临时版本记录到持久存储器后提交

OCC 在验证阶段通过检测事务之间的读写以及写写冲突来确保事务执行对于其他重叠事务而言时串行等价的。当该事务启动时，其他未提交的事务便被记录为重叠事务。为了检测冲突，所有事务均需维护 strat_ts（事务的开始时间戳）和 end_ts（事务进入验证时的时间戳），用于判断当前事务和另一个事务是否存在并发。

如图 5 所示，T1 事务在工作阶段，首先读取数据 X 到自己临时读集中。之后 T2 事务将新的数据 X 写入自己的写集，并不修改实际数据 X。T2 事务提交验证时发现其他并发事务并不存在写集，所以

T2 事务正常通过验证完成提交。T1 事务提交验证时，遍历重叠事务时发现与 T2 存在读写冲突，从而使 T1 回滚以避免冲突导致的数据异常。

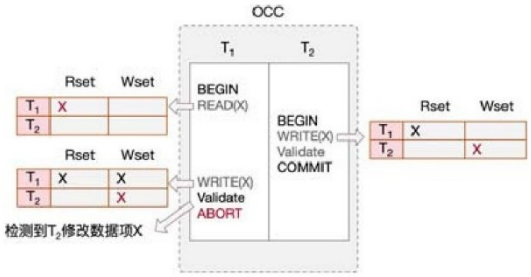

图 5 OCC 算法检验例子

上述算法存在两个缺点：只读事务也需要进行验证并且需要占用系统大量资源开销用以维护大量事务的写集并进行检查验证。因此出现了两种改进算法，BOOC（backward validation）以及 FOOC（forward validation）。其验证阶段只检查当前事务的写集是否与活跃事务（正在读写阶段的事务）的读集 ActiveRset 存在交集。相比于 BOCC，FOCC 的只读事务无需验证，整体上减少了验证的开销。

以上两种算法 BOCC 和 FOCC 的验证和结束阶段都在临界区中执行，这会极大的降低事务验证的并发度，并且也难以应用到分布式系统中。并发验证的 OCC 方法将所有事务的写集划分为 History（已提交事务的写集）和 Active（验证阶段的事务写集），并将验证阶段拆分为几个临界区以增加并发度。事务 T 验证先在第一临界区将自己的写集存入 Active，保证可以被后进入验证的事务发现，并拷贝一份现有 History 和 Active。验证通过之后进入第二临界区，将自己从 Active 中清楚并写入 History。

4.2.2 Silo

由于传统乐观并发控制算法存在较多缺陷和问题，后续研究者便对其提出了相关改进，Silo 便是其中一种改进算法。该算法的主要思路为：事务根据进入验证阶段的时间确定顺序，在验证阶段通过检测自己的读集是否被其他事务修改进行检验。因此，Silo 在数据上维护数据项的修改时间戳 wts 用于判断写冲突，并额外维护 txn_id 写锁用于避免写写冲突。在事务中则需要维护读集 Rset 和写集 Wset,记录读写过的数据项。

Silo 算法的读写操作与传统 OCC 算法相同，但需要额外记录读取时数据项的元信息，便于验证阶段的对比。在验证阶段，Silo 算法支持多个事务并发进行验证。为了解决并发验证存在的写写冲突问题，验证阶段，第 1 步需要对写集中的数据加锁。为了防止加锁过程中发生死锁，加锁前需要对写集的数据进行排序。第 2 步，检查读集中数据是否被其他事务加锁或 wts 信息是否改变，以此来判断是否存在读写冲突。在验证通过后，解锁并写入新数据。如图 6 所示，事务 T2 在提交时修改数据项的 wts，T1 验证时发现数据项 X 的 wts 与读操作时不同，从而回滚。

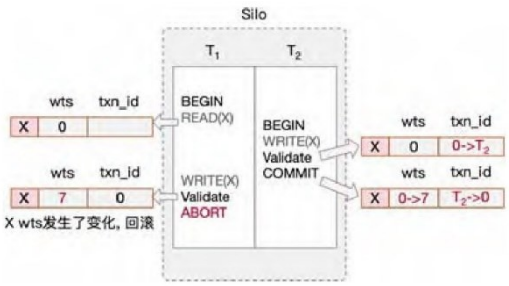

图 6 Silo 例子

Silo 通过数据项来检验冲突，大大减少了验证阶段所需的操作;其次，Silo 算法下多个事务并发验证，提高了事务执行的并发度。因此，相比于传统的 OCC，Silo 在性能上有很大提升。然而，Silo 中每个事务仍需维护读写集，在实际系统中，对数据库的内存提出了较大的挑战。

4.2.3 MaaT

MaaT 是传统 OCC 算法的另一种优化。采用了动态时间戳范围调整的方式来降低事务回滚率。该算法采用了动态定序+检验依赖的组合。

MaaT 采用了动态时间戳范围调整的方式来降低事务回滚率。其主要思想是通过事务间的读写操作之间形成的关系，确定事务的先后顺序，从而确定可串行化要求的等价串行序列中事务的先后顺序。例如，Ti 事务读 x 之后，Tj 事务需要更新 x，则在等价串行序列中，Ti 需要排在 Tj 的前面。

MaaT 需要在每个数据项上额外维护元数据，包括：(1)记录读了该数据项但仍未提交的事务 ID，称为读事务列表 readers；(2) 记录要写该数据项但仍未提交的事务 ID，称为写事务列表 writers；(3) 读过该数据项的事务中最大的提交时间戳，记为 Rts；(4)写过该数据项的事务中最大的提交时间戳，记为 wts。每个事务会有一个时间戳范围[lower,upper)，并初始化为[0,+)。事务中的各个操作的流程主要包括：

1.读操作

a)将数据项的写事务列表存入事务的uncommitted_writes；

b)更新当前事务的 greatest_write_timestamp=Max{greatest_write_timestamp,wts}；

c)将当前事务 ID 写入所读数据项的读事务列表；

d)读对应数据项，并将读到的数据存入读集。

2.写操作

a)将数据项的写事务列表存入事务的uncommitted_writes_y。

b)将数据项的读事务列表存入事务的uncommitted_reads。

c) 更新当前事务的greatest_write_timestamp=Max{greatest_write_timestamp,wts}，greatest_read_timestamp=Max{greatest_red_timestamp,rts}；

d)将当前事务 ID 写入要写数据项的写事务列表；

e)将要写的数据项新值存入写集；

3.验证阶段（事务协调者根据所有参与者返回的 lower 和 upper 取交集确定 lower 和 upper，如下操作均在参与者上执行）：

a)更新 lower=Max{greatest_write_timestamp+1,lower}；

b)保证 uncommitted_writes（未提交写事务列表）中事务的 lower 大于当前事务的 upper；

如果 uncommitted_writes 中的事务已经验证通过，修改当前事务的 upper；否则将 uncommitted_writes 中的事务放进当前事务的 after 队列（队列中的事务需要在当前事务之后提交）；

c)更新 lower=Max{greatest_read_timestamp+1,lower}；

d)保证 uncommitted_reads（未提交读事务列表）中事务的 upper 小于当前事务的 lower；

如果 uncommitted_reads 中的事务已经验证通过，修改当前事务的 lower；否则将列表中的事务放进当前事务的 before 队列（队列中的事务需要在当前事务之前提交）；

e)调整 uncommitted_writes_y（存在写写冲突的未提交事务列表）和当前事务的先后关系；

如果 uncommitted_writes_y 中的事务已经验证通过，修改当前事务的 lower 大于列表中已验证通过事务的 upper；否则将列表中的事务事务放进当前事务 after 队列；

f)检查 lower<upper 是否成立，不成立则回滚当前事务；

g)协调调整当前事务的 lower 和 before 队列中事务的 upper，保证当前事务的 lower 大于 before 队列事务的 upper；

h)协调调整当前事务的 upper 和 after 队列中事务的 lower，保证当前事务的 upper 小于 after 队列中事务的 lower；

4.写入阶段（首先在协调者上确定提交时间戳（commit_ts）为最终时间戳区间的 lower，然后在参与者上执行如下操作）

a)对于读集中的每个元素，将当前事务从对应数据项的读事务列表中清除，并进行如下操作：

保证写事务列表中事务的 lower 大于当前事务的 commit_ts；更新 Rts=Max{commit_ts,Rts}；

b)对于写集中的每个元素，将当前事务从对应数据项的写事务列表中清除，并进行如下操作：

保证写事务列表中事务的 upper 小于当前事务的 commit_ts。保证读事务列表中事务的 upper 小于当前事务的 commit_ts。更新 Wts=Max{commit_ts,Wts}。

4.2.4 Sundial

Sundial 时动态定序+混合检验的组合，其通过动态计算提交时间戳以减少回滚率。同时在数据项上维护租约（即数据项的可以被访问到的逻辑时间范围），便于在发生冲突时快速确定事务的先后顺序。此外 Sundial 在乐观并发控制的基础上，结合了悲观并发控制的思路，读写/写读冲突用 OCC、写写冲突用 2PL 锁的方式来减少分布式事务协调调度的开销。

Sundial 在数据项上维护租约(wts,rts)，分别代表了数据项最后被写入的时间和数据项可以被读到的最晚时间。在事务上维护 commit_ts，代表事务的提交时间戳。在读写集中额外维护 orig.rts 和 orig.wts，代表访问数据项当时的 rts 和 wts。我们对 Sundial 的主要操作的执行流程介绍如下：

1.读操作

a)首先从读写集中读取所需要的数据项，如果读写集中不存在所需数据项则需要访问数据存储，找到对应数据项并读取，并记录此时数据项的 wts 和 rts，记为 orig.wts 和 orig.rts；更新当前事务的 commit_ts=max{orig.wts,commit_ts}；

b)如果读写集中存在所需数据，直接返回对应

数据；

2. 写操作

a) 首先从写集中找到所要修改的数据项，如果写集中不存在所需数据项：对元组加锁，若加锁失败，存入等待队列 waiting_set；否则，直接返回数据项，以及对应的 wts 和 rts，记为 orig.wts 和 orig.rts；

b)如果读写集中存在当前待更新的数据项对应元素，则在写集中对其进行更新；

c)更新当前事务的 commit_ts=max{orig.rts,commit_ts}；

3.验证阶段

a)首先计算出提交时间戳 commit_ts，主要通过如下两步（该步骤为 3TS 中实现新增，由于读写操作在参与者上进行，协调者在进入验证前需要汇总所有参与者的信息得到 commit_ts）：遍历写集，更新 commit_ts 大于等于写集中所有元素的 orig.rts；遍历读集，更新 commit_ts 大于等于读集的 orig.wts；

b)验证读集中的每一个元素：如果提交时间戳 commit_ts 小于 rts，跳过当前元素；尝试更新元组租约：（1）如果 orig.wts！=wts，即当时读取的 wts 和元组现在的 wts 不同，当前事务需要回滚；（2）如果当前元组被加了锁，当前事务回滚；（3）否则更新元组的 rts=Max{rts,commit_ts}；

4. 写入阶段

a)提交操作，对写集中元素对应的数据项更新并解锁；

b)回滚操作，对写集中元素对应的数据项解锁。

4.3 时间戳排序算法

T/O 是基于时间戳的并发控制算法，该算法是静态定序和检测依赖的组合。

该协议的主要特点：（1）每个事务都会被赋予一个 Timestamp；（2）每条数据都会记录最近读取该数据的事务 id，以及最近写入的事务 id；（3）每个事务在读写数据时需要根据事务的 timestamp 和数据的读写 timestamp 进行冲突检测；（4）DBMS 会把事务读取过的数据拷贝到一个 private 的空间，来实现可重复读.

根据上述特点，可以了解到 T/O 要求每条记录额外存储以下两个信息：read timestamp——表示读过该数据中最大的事务 Timestamp；write timestamp——表示最近更新该数据的事务 Timestamp。

事务通过维护记录中的 wts 与 rts 来检测依赖，其具体协议如下：

对事务 T 发出的 read(Q)请求：

1.若 TS(T)< WTS(Q)，说明 T 要读的值被覆盖，read 操作被拒绝，事务 T 回滚。

2.否则，执行 read 操作，同时设置 RTS(Q) = Max(TS(T), RTS(Q))

对事务 T 发出的 write(Q) 请求：

1.若 TS(T)<RTS(Q)，说明有更新的事务已经读取了 Q，T 产生的 Q 值不应该被写入，否则更新事务写入的 Q 值就是过期的了。因此，系统拒绝 write 请求，事务 T 回滚。

2.否则，若 TS(T)< WTS(Q)，说明 T 写入的值是过时的，因此系统拒绝 write，事务 T 回滚。

3．否则，执行 write 操作，同时更新 WTS(Q)=TS(T)。

若事务 T 被回滚，系统赋予其新的时间戳并重新执行 T。

上述协议面临这一个问题：当 T1 事务修改数据 X 后，T2 事务再次修改了 X 并已经提交，此时 T1 事务需要进行回滚，无法确定数据 X 的回滚版本。解决该问题的方法是加入预写机制即事务写操作时不将新数据写入数据库而是暂存等到事务提交时再写入。该机制解决了无法处理回滚，但会影响读取操作的正常执行，此时后开始处理的事务需要等待前一事务处理完成。

预写操作需要在数据上增加两个字段：min_pts 表示当前数据上所有预写操作的最小时间戳，用于判断是否等待；min_rts 代表数据上所有预读操作的最小时间戳，判断写操作是否要进行等待。此外，还需要额外维护 3 个队列来记录等待的事务:read_reqs，代表当前数据项上等待的读操作；pre_reqs，代表当前数据项上的预写操作;write_reqs，代表当前数据项上等待的提交操作。

4.4 多版本并发控制算法

之前所介绍的相关算法在处理读写冲突时，必然要延迟或回滚一个事务来保证可串行化。读写冲突在所有冲突中占比很大，均选择回滚会对性能产生很大影响，但如果将读操作要读取的旧值维护起来，读写冲突就可以通过读取旧值来正常执行。这种维护数据的多个版本的方法被称为多版本并发控制（MVCC）。MVCC 是现代数据库中最普遍实现的并发控制算法，早在 1978 年并被提出。维护数据的多个版本使得事务对数据的操作变为对数据版本的操作极大的提高并发度，提供读写并行的能力。但 MVCC 需要配合其他算法实现可串行化，其

与 T/O、2PL、OCC 的结合的算法分别有 MVTO、MV2PL、Cicada，这里不做相关介绍。下面介绍基于 MVCC 的快照隔离算法。

4.4.1 快照隔离类算法（SI）

根据第三节相关阐述，快照隔离算法（SI）是静态定序加检测依赖的组合方式。该算法将事务分为两类，分别是只读事务和读写混合事务。其中读写混合事务根据提交时间戳而只读事务根据开始时间戳确定顺序。该算法通过检验写写冲突和写读依赖来进行并发控制。

快照隔离在每个事务开始时为事务提供一个数据库的“快照”，快照里所有数据都是数据库中当前已经提交的数据。每个事务在与其他并发事务完全隔离的情况下对快照进行读写。因为快照中的数据都是已提交的，所以只读事务不会和任何活跃的读写事务冲突，更不会和其他只读事务冲突，所以只读事务永远不会被阻塞或被中止。并发执行且需要更新数据库的事务之间可能会有潜在冲突。所以在快照隔离中，所有事务的更新操作都写在各自的私有空间里，事务在准备好提交后需要先先向并发管理器申请验证，通过验证的事务才允许将其更改提交到数据库，没通过验证的事务会被中止。而对于写写冲突，SI 为避免丢失更新，有两种验证方案，分别是“先提交者获胜策略”和“先更新者获胜策略”。SI 虽然能够通过以上策略避免一些冲突，但无法对读写依赖进行限定，并不能达到可串行化，需要其他机制不足，现在主流有两种方法 SSI 和 WSI。

1.SSI

该方法的核心是在 SI 的基础上检测连续的读写依赖，可以归纳为静态定序+混合检验的组合。该方法会在数据上维护两个结构：

i).txn_id，用于基础 SI 算法处理写写冲突

ii).SIRead-Lock，记录读取过该数据的所有事务，用于协助事务判断是否存在读写依赖。

每个版本需要记录数据提交时间戳 wts 和写入该版本的事务号 creator。此外为了检验读写依赖，还需维护 inConflict 和 outConflict，分别代表读写依赖于 T 的事务和被 T 读写依赖的事务。

SSI 的并发度相比于 2PL 和 OCC 更高，但增加了维护事务的 inConflict 和 outConflict 结构的开销。

2.WSI

WSI 将对写写冲突的检测转化为对读写依赖的检测，并通过处理读写依赖来达到可串行化。

WSI 中数据需要维护最大提交时间戳 lastCommit。事务会在验证阶段检验 lastCommit 是否改变来检查读写依赖，改变了说明之后写入了新版本需要进行回滚。SI 基本的快照机制会保证写读依赖，写写依赖对事务先后顺序不作要求无需检测。

4.5 确定性并发控制算法——Calvin

Calvin 默认应用在预先知道事务的全部 SQL 语句的场景中，其主要思路为：预先为事务确定顺序，之后强制按照确定的顺序执行事务，以避免分布式协调的开销。该算法主要靠两个模块完成工作：定序器（sequencer）和调度器（scheduler）。

定序器负责拦截事务并将事务放入全局事务输入序列中。同时它还负责复制。调度器负责使用确定性锁定方案来协调事务执行，以保证与定序器指定的串行顺序等效，同时允许事务由事务执行线程池并发执行。

如图 7 所示，定序器规定的 batch 中存在 T1 和 T2 事务，且 T1 排在 T2 之前。T1 事务需要写数据 A 并读数据 B，T2 读数据 A 写数据 C。调度器首先为 T1 事务的读写操作加锁，再为 T2 的读写操作加锁。但此时，数据 A 已经有了 T1 的写锁，T2 事务只能排入数据 A 的等待队列中，知道 T1 事务结束后才能读取到数据 A。加锁完成后可以依序执行 batch 中的事务，根据事务的读写集执行本地读，之后同步各个节点上子事务的读写集，最后执行本地写操作，完成事务执行。

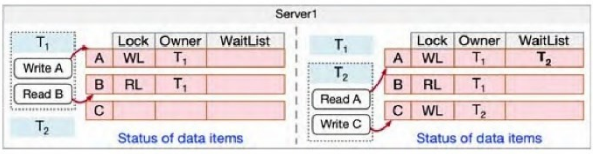

图 7 Calvin 检验例子

5 总结与展望

本文调研了现有的主流并发控制算法，并根据相关文献的思想将算法划分进入先定序再检验的基本框架，为后续并发控制算法的相关创新提供了思路。此外，本文还详细的介绍了各个算法的具体协议规则，并借此分析了算法的优缺点。

如今内存数据库虽然相比于磁盘数据库有较大的性能特征，但也存在相当大的空缺等待填充，传统为基于磁盘的数据库设计的架构已经不能适

应内存数据库的需求。内存数据库的优化技术可以从下述角度进行思考：

1)去掉传统的缓冲区机制：传统的缓冲区机制在内存数据库中并不适用，锁和数据不需要再分两个地方存储，但仍然需要并发控制，需要采用与传统基于锁的悲观并发控制不同的并发控制策略。

2)尽量减少运行时开销：磁盘 I/O 不再是瓶颈，新的瓶颈在于计算性能和功能调用等方面，需要提高运行时性能。

3)采用编译执行方式：传统数据库多采用火山模型执行引擎，每一个 Operator 都被实现为一个迭代器，提供三个接口：Initial、Get-Next、Closed，从上往下依次调用。这种执行引擎的调用开销在基于磁盘的数据库管理系统中不占主要比重（磁盘 I/O 是最主要瓶颈），但在内存数据库里可能会构成瓶颈。假设要读取 100 万条记录，就需要调用 100 万次，性能会变得难以忍受，这就是内存数据库中大量采用编译执行方式的原因。直接调用编译后的机器代码，不再需要运行时的解释和指针调用，性能会有效提升。

4)可扩展的高性能索引构建：虽然内存数据库不从磁盘读数据，但日志依然要写进磁盘，需要考虑日志写速度跟不上的问题。可以减少写日志的内容，例如把 undo 信息去掉，只写 redo 信息；只写数据但不写索引更新。如果数据库系统崩溃，从磁盘上加载数据后，可以采用并发的方式重新建立索引。只要基础表在，索引就可以重建，在内存中重建索引的速度也比较快。

关于并发控制算法的设计，如今分布式系统逐渐成为主流，因此设计合适的高性能分布式并发控制算法可能是新兴研究方向。此外，新型高速网络技术 Infiniband 和 RDMA 发展迅速，如何将其应用到并发加速中存在很大的研究空间。HATP 系统同时支持 OLAP 和 OLTP，但两者差异性很大，在设计过程中如何同时保证两者的性能也是一个挑战！

致 谢 感谢三位老师的指导！本课程受益良多！

参 考 文 献

[1] 赵泓尧,赵展浩,杨皖晴等.内存数据库并发控制算法的实验研究[J].软件学报,2022,33(03):867-890.DOI:10.13328/j.cnki.jos.006454.

[2] Mehdi S A, Hwang D, Peter S, et al. {ScaleDB}: A Scalable, Asynchronous {In-Memory} Database[C]//17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 2023: 361-376.

[3] Guo ZH, Wu K, Yan C, Yu XY. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2021. 658−670. [doi: 10.1145/3448016.3457294]

[4] Huang YH, Qian W, Kohler E, Liskov B, Shrira L. Opportunities for optimism in contended main-memory multicore transactions.Proc. of the VLDB Endowment, 2020, 13(5): 629−642

[5] Takayuki T, Takashi H, Hideyuki K, Osamu T. An analysis of concurrency control protocols for in-memory databases with CC Bench. Proc. of the VLDB Endowment, 2020, 13: 3531−3544. [doi: 10.14778/3424573.3424575]

[6] Zhao ZH. Efficiently supporting adaptive multi-level serializability models in distributed database systems. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2021. 2908−2910. [doi: 10.1145/3448016.3450579]

[7] Mahmoud HA, Arora V, Nawab F, Agrawal D, Abbadi A. MaaT: Effective and scalable coordination of distributed transactions in the cloud. Proc. of the VLDB Endowment, 2014, 7(5): 329−340

[8] Yu XY, Xia Y, Pavlo A, Sanchez D, Rudolph L, Devadas S. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. Proc. of the VLDB Endowment, 2018, 11(10): 1289−1302. [doi: 10.14778/3231751.3231763]

[9] Harding R, Van Aken D, Pavlo A, Stonebraker M. An evaluation of distributed concurrency control. Proc. of the VLDB Endowment, 2017, 10(5): 553−564. [doi: 10.14778/3055540.3055548]

附录：汇报记录

问题一：这个架构在防止系统崩溃，确保数据持久性问题上是怎么进行处理的？

内存数据库的存储介质是内存，内存通常是易失性介质，虽然现在也在研究非易失类的内存但离实际应用还较远。所以现在内存数据库主要用于数据缓冲层，即内存数据库会依附于一个磁盘数据库进行备份。现有内存数据库主要用预写式日志的机制进行回滚和恢复。

（1）所有更新都要对应日志，日志没有落盘之前，数据的修改不允许落盘

（2）每条日志都有一个 LSN(log sequence number)号，单调递增，向磁盘连续写

问题二：这篇论文中的异步是那两个异步？两者出现并发，范围查询在更新前后不一致是怎么处理的？

这篇论文通过将事务执行与范围索引进行更新解耦，专注于提高事务执行与索引更新可扩展性。延迟范围索引更新可以批处理避免范围索引争用，提高数据库的可扩展性。该数据库设计了 indexlet 和 phantomlet 用于存储和指示异步更新之前的写操作。如果范围索引过程中发现出现新增的，即幻象指示器发生变化，根据并发控制协议，意味着事务出现冲突需要回滚，重新进行事务操作，保证事务处理的可串行性。