软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):867−890 [doi: 10.13328/j.cnki.jos.006454]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

内存数据库并发控制算法的实验研究*

赵泓尧[1,2], 赵展浩[1,2], 杨皖晴[1,2], 卢 卫[1,2], 李海翔[3], 杜小勇[1,2]

[1](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)

[2](中国人民大学 信息学院, 北京 100872)

[3](腾讯科技(北京)有限公司 计费平台部, 北京 100193)

通信作者: 杜小勇, E-mail: duyong@ruc.edu.cn; 卢卫, E-mail: lu-wei@ruc.edu.cn

摘 要: 并发控制算法是数据库系统保证事务执行正确且高效的重要手段, 一直是数据库工业界和学术界研究的核心问题之一. 将并发控制算法的基本思想归纳为“先定序后检验”, 并基于该思想对现有各类并发控制算法进行了重新描述和分类总结. 基于在开源内存型分布式事务测试床 3TS 上的实际对比实验, 系统性地探究了各类算法的优缺点和适用场景, 为面向内存数据库的并发控制算法的后续研究提供参考.

关键词: 数据库系统; 事务处理; 并发控制算法; 3TS; 内存数据库

中图法分类号: TP311

中文引用格式: 赵泓尧, 赵展浩, 杨皖晴, 卢卫, 李海翔, 杜小勇. 内存数据库并发控制算法的实验研究. 软件学报, 2022, 33(3): 867−890. http://www.jos.org.cn/1000-9825/6454.htm

英文引用格式: Zhao HY, Zhao ZH, Yang WQ, Lu W, Li HX, Du XY. Experimental Study on Concurrency Control Algorithms in In-Memory Databases. Ruan Jian Xue Bao/Journal of Software, 2022, 33(3): 867−890 (in Chinese). http://www.jos.org.cn/1000-9825/6454.htm

Experimental Study on Concurrency Control Algorithms in In-Memory Databases

ZHAO Hong-Yao[1,2], ZHAO Zhan-Hao[1,2], YANG Wan-Qing[1,2], LU Wei[1,2], LI Hai-Xiang[3], DU Xiao-Yong[1,2]

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

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

[3](Billing Platform Department, Tencent Technology (Beijing) Co. Ltd, Beijing 100193, China)

Abstract: The concurrency control algorithm is a key component to guarantee the correctness and efficiency of executing transactions. Thus far, substantial effort has been devoted to proposing new concurrency controls algorithms in both database industry and academia. This study abstracts a common paradigm of the state-of-the-art and summarizes the core idea of concurrency control algorithms as “ordering-and-verifying”. Then, the existing concurrency control algorithms are re-presented following the ordering-and-verifying paradigm. Based on extensive experiments under an open-source memory-based distributed transaction testbed called 3TS, it is systematically demonstrated the advantages and disadvantages of the mainstream state-of-the-art concurrency control algorithms. Finally, the preferable application scenario is summarized for each algorithm and some valuable references are provided for the follow-up research of concurrency control algorithms used in in-memory databases.

Key words: database system; transaction processing; concurrency control algorithm; 3TS; in-memory database

事务是用户定义的一组数据库操作组成的序列, 是数据库管理系统中的最小执行单元[1,2]. 事务概念的提

* 基金项目: 国家重点研发计划(2020YFB2104100); 国家自然科学基金(61972403, 61732014); 中央高校基本科研业务费专项资金(20XNLG22); 中国人民大学-腾讯联合实验室联合项目基金

本文由“数据库系统新型技术”专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.

收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

出, 推动了数据库从实验室中的原型系统转变为现代信息系统不可或缺的基础设施. ACID 理论[1,2]定义了数据库系统事务处理的基本要求, 即数据库中的事务需要具备 4 大特性: 原子性(A)、一致性(C)、隔离性(I)和持久性(D). 原子性要求事务的操作要么全做, 要么全不做; 一致性要求事务的执行必须使数据库从一个一致性状态变迁到另一个一致性状态, 不允许未成功提交事务的临时数据被访问; 隔离性要求事务的执行不能影响其他事务也不能被其他事务影响; 持久性要求数据库持久化存储已提交事务写入的数据. 数据库系统为了保证 ACID 特性, 引入故障恢复子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.

并发控制算法是数据库并发控制子系统的核心, 其目标是对事务进行合理调度, 从而保证事务执行的正确性. 并发控制算法的效率直接影响着事务处理的性能, 因而设计并实现正确且高效的并发控制算法, 一直是数据库系统研究的核心课题之一. 当前, 设计并发控制算法做到正确且高效仍面临以下两大挑战.

- 首先, 并发控制算法在保证可串行化隔离级别的前提下, 保证事务处理的高效性是一个挑战. 可串行化是保证事务正确调度的黄金法则[1,2], 其要求事务的并发执行结果与这些事务的某个串行执行结果相同. 为了保证可串行化, 并发控制算法需要对事务之间的并发关系进行追踪和维护, 并且基于这些信息判断事务是否可以提交. 由于可串行化较严格的排序要求, 其势必会导致事务的并发度受限, 事务的回滚率上升. 因此, 可串行化对并发控制算法的逻辑设计提出了较高的要求.
- 其次, 随着内存数据库的出现和普及, 并发控制算法的处理逻辑需要针对基于内存的数据库架构进行调整. 相比于传统的磁盘数据库, 内存数据库的差异主要在于数据项、索引以及并发控制算法所需的元信息等数据的存储上.
  - 首先, 在磁盘数据库中, 数据项和索引需要存储在磁盘中来防止数据丢失, 而这就使得并发控制算法的设计需要将访问数据项、索引的磁盘 I/O 开销纳入考虑. 因此, 在磁盘数据库中, 降低回滚率并提高事务并发度, 可以降低磁盘 I/O 对性能的影响. 内存数据库由于其高速数据存取的优点, 降低了回滚率和并发度对算法性能的影响. CPU 的计算处理性能反而成为内存数据库的主要瓶颈, 因此, 并发控制算法本身的处理开销是影响算法性能的一个关键. 以锁机制为例, 磁盘数据库系统往往会使用细粒度的锁来减少锁争用, 而在内存数据库中就可以使用粗粒度的锁来减少维护锁的计算开销.
  - 其次, 磁盘数据库中需要额外在内存中设计存储结构来维护元信息, 这些结构不仅实现复杂, 在获取元信息时也会带来一定的开销. 内存数据库则规避了这一问题, 元信息直接存储在数据项的头部简化了实现, 同时减少了从复杂结构中获取元信息的开销.

需要说明的是, 从研究并发控制算法的角度看, 采用内存数据库可以更直接地评估并发控制算法的效率, 因为不需要考虑内外存 I/O 的影响, 而这常常是影响磁盘数据库性能的主要因素. 因此, 并发控制算法的不同特性导致了其对不同系统架构的适用性不同. 而如何利用内存数据库具备的高速数据读取的优点, 设计并优化并发控制算法, 是当前的另一大挑战. 例如, 基于时间戳的并发控制算法由于需要有全局递增的时间戳, 在分布式内存型数据库中会存在时钟分配的性能瓶颈, 从而导致事务执行效率较低.

近年来, 面向内存数据库的并发控制算法研究期望解决这两大挑战. Bamboo[3]在两阶段封锁(two-phase locking, 2PL)算法[4,5]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; MaaT[6]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[7]的逻辑; Sundial[8]引入了 Cache 机制, 在内存型分布式数据库上表现优秀. 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. 因而, 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. Bernstein 在 1981 年[5]就对当时最先进的并发控制算法进行综述和总结. Agrawal 等人[9]、Carey 等人[10]、Huang 等人[11]的工作则在介绍算法的基础上, 主要通过实验分析了各类算法的性能. 最近的工作 Deneva[12]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, Huang 等人[13]和 Tanabe 等人[14]则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.

Huang 等人[13]分析了 7 种因素对 3 种 OCC 类算法 Silo[15]、Tictoc[16]、Cicada[17]性能的影响情况, 并针对高冲突下 OCC 类算法性能较差的情况给出了两种优化方法. Tanabe 等人[14]则是通过实验分析了 7 种并发控制算法和 7 种算法通用的优化方法, 给出了不同算法的优化建议.

本文在参考以上工作的基础上, 将并发控制算法的基本思想归纳为先定序后检验. 即并发控制算法可以统一理解为先对事务进行定序——规定算法要求的事务执行顺序; 之后检测事务实际的执行是否满足这个定序要求, 如不符合, 则事务需要回滚. 基于这一思想, 本文对现有的并发控制算法逻辑进行了梳理, 在开源的内存型分布式事务测试床 3TS[18]上实现了主流的并发控制算法, 并在相同的分布式场景下进行了实验测试.在算法的选择上, 我们选择了两阶段封锁、乐观并发控制、时间戳排序(timestamp ordering, T/O)[5,19]、快照隔离(snapshot isolation, SI)[20]等比较传统的算法以及 MaaT、Calvin[21]等近年来提出的部分算法进行总结分析. 由于篇幅限制, 本文没有描述一些对已有并发控制算法的优化研究[16,22−32], 比如针对 2PL 的优化[27]、确定性方法的优化[22]、实时数据库下特定并发控制算法的优化[31]、OCC 算法的优化[23−26,29,32]、混合并发控制算法的优化[30]. 值得一提的是, 这些优化算法都适用本文提出的“先定序后检验”框架体系.

我们将分布式内存数据库中各类并发控制算法的优缺点和适用场景归纳如下: (1) 2PL、T/O、MVTO[33]、MV2PL[33]以及 Silo 算法适合低冲突率下的应用场景; (2) Calvin 确定性算法受冲突率以及写事务比例影响较小, 适合于高冲突和分布式事务较多的场景; (3) 写快照隔离(write snapshot isolation, WSI)[34]算法低中高冲突率场景下都拥有较为不错的表现; (4) Sundial 算法回滚率较低, 适合中高冲突场景.

本文第 1 节首先概述现有的并发控制算法, 并总结了并发控制算法“先定序后检验”的核心思想. 第 2 节–第 5 节分别对基于 2PL 算法、OCC 类算法、T/O 算法以及多版本并发控制(multi-version concurrency control, MVCC)算法[35]进行了分析和总结. 第 6 节则讨论最新的确定性并发控制算法. 第 7 节通过分布式场景下的实验来分析各种并发控制算法的优缺点. 第 8 节总结全文, 分析了不同场景下的适用算法, 并对未来值得关注的研究方向进行探讨. 为方便叙述, 表 1 给出了各类并发控制算法的全称及简称.

表 1 并发控制算法简称

并发控制算法	简称
两阶段封锁	2PL
乐观并发控制	OCC
时间戳排序	T/O
快照隔离	SI
多版本并发控制	MVCC
可串行化快照隔离(serializable snapshot isolation, SSI)[36,37]	SSI
写快照隔离	WSI

1 数据库并发控制算法的基本思想

并发控制算法用以对事务进行合理调度, 保证事务执行的正确性, 规避数据异常. 数据异常会破坏数据库的一致性和隔离性[1,2]. 如图 1 所示, 以丢失更新异常[20]为例, 账户 X 初始有存款 100 元, 存在两个事务 T_1 和 T_2, 事务 T_1 往账户 X 里存 100 元, 事务 T_2 并发地往账户 X 存 50 元, 如果两个事务都提交成功, 账户 X 的余额应为 250 元. 如图 1 左半部分所示, 如果缺乏对事务 T_1 和 T_2 的合理调度, 两个事务执行完后, X 余额为 200 元, 事务 T_2 所存入的钱被丢失, 因而出现了丢失更新异常. 使用并发控制算法可以规避数据异常, 以 2PL 算法为例, 如图 1 右半部分所示, 事务 T_1 在修改 X 数据项时, 首先对数据项 X 加锁, 事务 T_2 修改数据项 X 时, 会发现无法对数据项 X 加锁, 因此事务 T_2 无法提交, 从而规避数据异常发生.

评估一种算法是否可以避免所有数据异常的黄金法则是可串行化, 即判断并发事务的执行结果是否与某一串行执行这些事务的结果等价: 若等价, 则认为该算法调度后的并发事务序列是可以避免所有数据异常的.

由于并发控制算法种类繁多, 我们很难深入清晰地理解每一种算法的逻辑, 给这方面的科研工作带来了挑战. 但我们观察到, 所有的算法的逻辑存在共性. 例如, 2PL 规定将事务加锁顺序作为可串行化调度中事务

之间的先后顺序, 通过锁进行检验; T/O 规定以事务开始时间作为可串行化调度中事务的先后顺序, 通过检验事务之间的依赖顺序来判断事务是否可串行化. 因此, 本文创新性地将并发控制算法的核心思想归纳为“先定序后检验”.

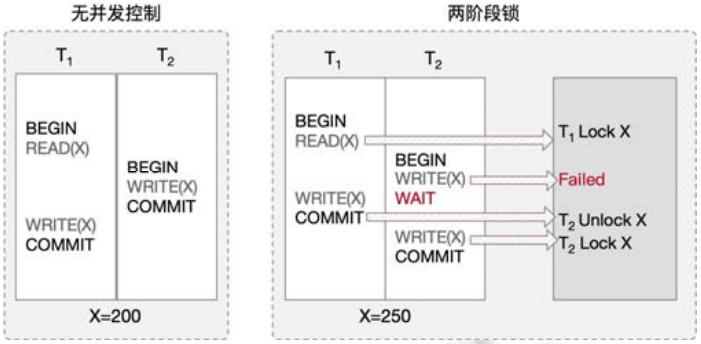


图 1 丢失更新数据异常示例

1.1 定 序

定序是指为并发事务确定一个等价可串行化调度中事务的先后顺序. 根据前期对算法的调研总结, 可以形成表 2.

表 2 主流算法的定序方式

算法经典分类	并发控制算法	定序方式
基于锁的算法	2PL 算法	根据获取锁的先后顺序确定顺序
基于锁的算法	MV2PL	根据获取锁的先后顺序确定顺序
乐观算法	OCC 算法	根据事务进入验证阶段的时间确定顺序
基于时间戳的算法	T/O 算法	根据事务开始时间戳确定顺序
基于时间戳的算法	MVTO	根据事务开始时间戳确定顺序
基于时间戳的算法	SSI	根据事务的提交时间戳确定顺序
基于时间戳的算法	WSI	根据事务的提交时间戳确定顺序
乐观算法	MaaT	根据事务执行时形成的数据依赖动态确定顺序
乐观算法	Sundial	根据事务执行时形成的数据依赖动态确定顺序
乐观算法	Silo	根据事务进入验证阶段的时间确定顺序
乐观算法	Cicada	根据事务开始时间戳确定顺序
确定性的算法	Calvin	根据事务开始时间戳确定顺序

目前, 主流的并发控制算法定序方式可以归纳为两种: (1) 静态定序; (2) 动态定序.

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. 使用该方法的典型算法有 T/O、OCC 等算法. 以丢失修改异常举例, 根据 T/O, 事务 T_1 先于事务 T_2 开始, 因此 T/O 算法会人为规定 T_1 排在 T_2 之前. 然而, 静态定序的方法过于严格, 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. 而事务实际执行过程中, 形成的事务先后顺序很有可能符合另一种串行序列, 而不是静态定序确定的顺序, 从而导致事务不必要的回滚. 因此, 最近的一些算法, 如 MaaT、Sundial 等算法, 均采用了动态定序的方式. 动态定序是指: 根据执行过程中的操作结果, 动态地调整事务在最终串行执行序列中的先后关系. 动态调整顺序的依据就是数据库中的 3 种依赖: 读写依赖、写读依赖和写写依赖[2,38].

- 读写依赖: 如 $R_1(X_0)W_2(X_1)$ (事务 T_n 读取数据项 X 的第 k 号版本标记为 $R_n(X_k)$, 事务 T_n 写数据项 X 并插入版本号为 k 的新版本标记为 $W_n(X_k)$), 即事务 T_1 读取了数据项的版本 X_0, 事务 T_2 写了一个新版本 X_1, 那么说事务 T_2 读写依赖于事务 T_1, $T_1 \xrightarrow{rw} T_2$.
- 写读依赖: 如 $W_1(X_1)R_2(X_1)$, 即事务 T_1 写了一个数据项的新版本 X_1, 事务 T_2 读取到了这个新版本 X_1, 那么说事务 T_2 写读依赖于事务 T_1, $T_1 \xrightarrow{wr} T_2$.
- 写写依赖: 如 $W_1(X_1)W_2(X_2)$, 即事务 T_1 写了一个数据项的新版本 X_1, 事务 T_2 之后写了这个数据项的

新版本 X_2, 那么说事务 T_2 写写依赖于事务 T_1, $T_1 \xrightarrow{ww} T_2$.

根据 3 种依赖的定义可以看出, 在读写依赖和写读依赖中, 事务拥有明显的先后顺序. 读写依赖的例子 $R_1(X_0)W_2(X_1)$中, 事务 T_1 必然需要排在事务 T_2 之前, 否则, 事务 T_1 应该读取到事务 T_2 写的新版本. 写读依赖的例子 $W_1(X_1)R_2(X_1)$, 事务 T_1 必然需要排在事务 T_2 之前, 否则事务 T_2 不应该读取到事务 T_1 修改的版本. 因此, 在并发控制中, 事务可以根据读写依赖和写读依赖来动态地确定事务的先后顺序.

动态定序的方式必然需要增加监控事务之间依赖的开销, 但是更灵活的事务顺序会带来较低的回滚率. 静态定序的方式虽然无须增加监控事务依赖的开销, 但是会引入不必要的回滚. 因此, 并发控制算法如何从动态定序和静态定序选择, 需要一定的权衡.

1.2 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. 检验的核心问题包括: (1) 如何检验; (2) 何时检验. 我们通过总结现有主流的并发控制算法, 归纳出各类算法解决这两个核心问题的 3 种方式.

1.2.1 如何检验

针对如何进行检验的问题, 目前主流算法的方法可以归纳为 3 种(见表 3): (1) 检验冲突; (2) 检验依赖; (3) 两者结合.

表 3 主流并发控制算法的检验方式

并发控制算法	检验方式
2PL 算法	检验冲突
MV2PL	检验冲突
OCC 算法	检验冲突
T/O 算法	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

其中, 检验冲突的方法比较简单, 只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. 若出现冲突操作, 则通过等待或者回滚来解决, 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. 这里的冲突操作包括: (1) 读操作和写操作冲突; (2) 写操作之间冲突. 目前, 主流的 2PL 以及 OCC 等算法均采用了检验冲突的方式. 2PL 算法在读写操作时, 通过加锁来检验是否存在读写、写写冲突. OCC 算法则是在事务提交之前检验是否存在读写冲突, 即检查全局是否有其他事务修改了自己的读集 *Rset*. 以图 1 的丢失修改异常为例, OCC 算法中, 事务 T_1 在验证阶段发现事务 T_2 修改了数据项 X(T_2 的写集 *Wset* 中存在 X), 而 T_1 之前读取过数据项 X(T_1 的读集 *Rset* 中存在 X), 存在读写冲突. 因此, 回滚 T_1 以避免丢失修改异常(如图 2 所示).

检验依赖则通过检验事务执行操作产生的数据依赖关系和事先规定的顺序是否相同, 来判断是否需要回滚. 采用该方法的算法主要有 T/O, MaaT 算法等. T/O 算法以事务开始时间决定顺序, 针对图 1 的例子, 图 3 中先开始的事务 T_1 在写数据项 X 时, 通过数据项上的 *rts* 结构发现数据项 X 已经被后开始的事务 T_2 修改, 此时 $T_2 \xrightarrow{ww} T_1$, 与 T/O 中事务开始顺序相反, 因此 T_1 回滚. 此例中 T1 的开始时间戳为 5, T2 的开始时间戳为 7, 之后其他涉及到时间戳的算法例子中也采用这个假设.

冲突和依赖这两种概念, 本质上都帮助了并发控制算法做到可串行化调度, 但是这两种概念存在一定的区别: 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先后顺序; 其次, 两种方法的检验思路不同, 检验冲突是通过规避和解决并发事务的冲突操作来做到可串行化, 检查依赖则是

根据 3 种依赖体现的事务先后关系来整理出一个可串行化序列, 若两个事务之间存在完全相反的依赖顺序, 则两者之中必须回滚一个来达到可串行化. 从冲突和依赖的关系上来看, 依赖可以理解为在冲突的基础上增加对事务顺序的处理, 更为精细化地对事务进行并发控制. 并发事务之间的依赖也可以简单地被当作冲突来处理. 从检验冲突和检验依赖的优缺点来看, 检验冲突的检验逻辑比较简单, 实现方便, 但是由于条件过于严格而会发现较多的假冲突; 检验依赖的方法检验精度较高, 可以减少很多不必要的假回滚, 但是检验难度较高, 需要维护较多的数据结构来协助检查.

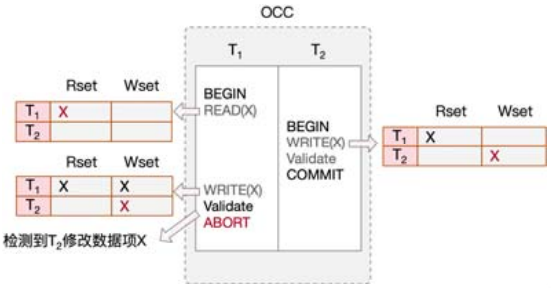


图 2 OCC 算法检验例子

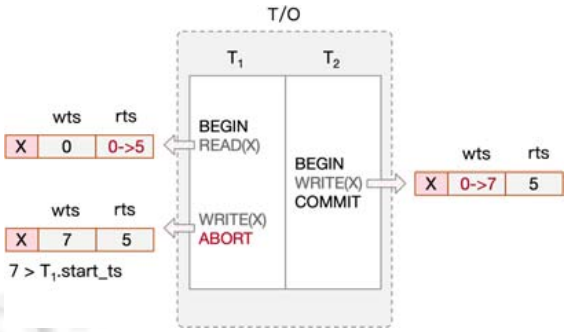


图 3 T/O 算法检验例子

因此, 最近的一些并发控制算法(如 Sundial)考虑使用检验冲突和检验依赖相结合的方式进行检验. 其手段为:

- 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. 预写使得事务只有提交时才会更新数据项, 因此写读依赖仅仅会发生在当前事务与已提交的写事务之间, 因此无须处理.
- 对于读写依赖, 利用数据项上维护的时间戳信息来检查读写依赖是否存在.

1.2.2 何时检验

确定如何检验后, 不同算法选择的事务检验阶段也存在不同. 对于除 Calvin 以外的算法, 我们可以将事务的执行流程拆分为 3 个阶段.

(1) 读写操作阶段(read-write phase), 事务在这个阶段执行读写操作.

(2) 验证阶段(validation phase), 在事务提交之前, 检验事务是否可以提交. 除 OCC 和 SI 算法外, 其他算法省略该阶段.

(3) 结束阶段(finish phase), 执行事务的提交或者回滚.

对于在事务执行中何时检验冲突这一问题, 一般有 3 种选择: (1) 读写操作阶段检验; (2) 验证阶段检验; (3) 两者结合. T/O 算法在读写操作阶段进行检验, 可以尽早检验出不可串行化的事务并将其回滚. OCC 算法则在验证阶段单独对冲突或者依赖进行检验, 提高了事务在读写时的并发度. 为了结合两种方式的优点, 最近的一些并发控制算法选择了两者结合的方式. 比如, Sundial 选择了在写操作时检验写写冲突, 在验证阶段检验读写依赖.

1.3 定序方式与检验方式可能的组合

通过对定序方式和检验方式的自由组合, 我们可以得到一个坐标系图, 将现有的主流并发控制算法标记在图中可以得到如图 4 的结果.

- 第一象限是静态定序与检验依赖的组合, 这种方法的主要思路是: 首先预先设定一个静态的顺序, 然后检验实际出现的事务依赖关系是否满足该顺序. 这种组合既避免了动态定序所需要的额外开销, 又一定程度上通过检验依赖减少了假回滚的数量. 主流算法中 T/O 算法以及 WSI 算法就属于这种组合方式.

- 第二象限是静态定序与检验冲突的组合, 这种组合只需要维护较少的数据结构, 但是会造成最多的假冲突. 使用这一组合的算法最多, 包括 2PL 算法、OCC 类算法和 Calvin 算法.
- 第三象限中, 动态定序与检验冲突的组合是不可能的, 因为动态定序的基础是事务之间的事务依赖, 而检验冲突无法为动态定序提供顺序上的帮助.
- 也正因为动态定序的基础是事务依赖, 第四象限中动态定序方式与检验依赖方式是非常自然的结合, 从中可以推导出最直接的一种算法就是根据事务之间的依赖构建出事务依赖图, 并在其中检查是否存在环路. 然而构建事务依赖图的方式过于耗时, 因此大部分的算法都选择了静态定序的方式来避免这样的开销. 不过, 最近的工作 MaaT 通过计算时间戳的方式, 一定程度上减少了构建事务依赖的开销.

图 4 主流算法定序/检验方式组合

在四大象限之外, 还有诸如 SSI、Sundial 算法选择了冲突和依赖混合检验的方式, 综合了检验依赖和检验冲突两者的优势.

文中算法逻辑的伪代码见 https://github.com/yqekzb123/yqekzb123.github.io/blob/master/CCAppendix.doc.

2 两阶段封锁算法

2PL 是目前应用最广泛的并发控制算法, 选择了静态定序+检验冲突的组合. 2PL 算法的主要思路是: 根据冲突加锁的先后排定事务顺序, 通过加锁操作检查事务之间的冲突. 事务在读操作时添加读锁(*RL*)、写操作时添加写锁(*WL*), 在提交或者回滚时释放所有锁. 其中, 读锁和写锁之间、写锁和写锁之间相互冲突.

当加锁出现冲突时, 如果事务选择等待锁, 最终可能产生死锁. 如图 5 所示, 事务 T_1 读取数据项 *X*, 并添加读锁 *RL*(*X*). 事务 T_2 读取数据项 *Y*, 并添加读锁 *RL*(*Y*). 之后, T_1 尝试修改 *Y*, 发现 *Y* 已被 T_2 添加读锁, 需要等待 T_2 结束. T_2 尝试修改 *X*, 但 *X* 已被 T_1 添加读锁, 需要等待 T_1 结束. 出现了 T_1 等待 T_2、T_2 等待 T_1 的局面.

为了预防或解除死锁, 2PL 提供了多种方式: (1) NO-WAIT[5]; (2) WAIT-DIE[39]; (3) WOUND-WAIT[39]; (4) 死锁检验[5]. NO-WAIT 机制是指当加锁发生冲突时, 立刻回滚当前事务. 在图 5 的例子中, 事务 T_1 向数据项 *Y* 加写锁失败后会立刻回滚, 释放之前获取的读锁 *RL*(*X*).

WAIT-DIE 机制是指在检测到冲突时, 根据事务之间的优先级选择处理方式, 即等待(WAIT)或回滚(DIE). 事务在开始时获取一个开始时间戳 *T.start_ts*, 将其作为判断加锁优先级的依据. 在检测到冲突时, 通过比较加锁事务 T_i 和锁拥有者 *O* 的 *start_ts* 的大小来确定处理方式: 若 $T_i.start\_ts<O.start\_ts$, 则 T_i 事务等待; 否则, T_i 事务回滚. 在以上的死锁例子中, 事务 T_1 在尝试修改数据项 *Y* 时, 由于 T_1 的开始时间戳小于 T_2 的开始时间戳, T_1 需等待 T_2 提交; 事务 T_2 在尝试写数据项 *X* 时, 由于 T_2 的开始时间戳大于 T_1 的开始时间戳, T_2 需回滚, 不会出现 T_2 等待 T_1 的局面, 避免了死锁的出现(如图 6 所示).

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.

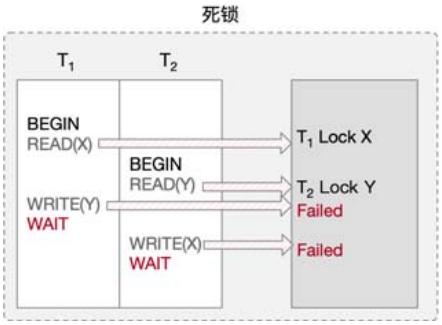


图 5 死锁

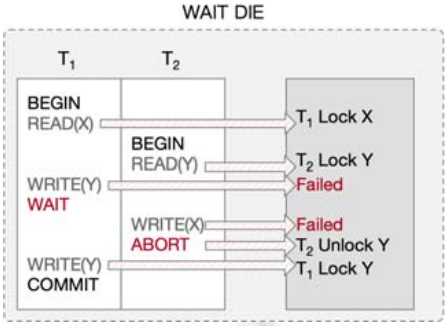


图 6 WAIT-DIE 机制

2.1 讨 论

对比 2PL 算法中的 3 种死锁处理方式, No Wait 方法的优点是实现简单, 缺点是长事务和其他事务产生冲突的可能性较大, 有可能被饿死. Wait Die 和 Wound Wait 通过增加比较优先级的方式避免了事务饿死的情况, 但锁算法下的冲突率依然较高.

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. 第 7 节的测试结果也表明: 以 Wait Die 为例的 2PL 算法整体上性能较优, 但是在高冲突率下性能下降比较严重. 因此, 2PL 类算法不适用于冲突率较高的场景.

3 乐观并发控制算法

OCC 最早在 1981 年[7]提出, 相比于 2PL 的悲观并发控制, OCC 仅在提交之前进行验证, 检查当前事务是否与并发事务存在读写/写写冲突. 然而传统的 OCC 算法性能较差, 因此后续提出了许多优化算法, 如 Silo、Sundial、MaaT、Cicada 算法.

3.1 传统乐观并发控制算法

传统 OCC 位于图 4 中的第二象限, 因此受到了静态定序+检验冲突这一组合高回滚率的影响. 传统 OCC 规定: 以事务进入验证阶段的时间排序, 并在验证阶段检测并发事务之间是否存在冲突. 为了检测并发事务之间的冲突, 在 OCC 中, 所有事务都需要维护 *start_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并发. 此外, 还需要维护 *Rset*(读集)和 *Wset* (写集): *Rset* 代表当前事务读取过的数据项 X 的集合, 其中每个元素 *Rset*[X]包含读到的数据 *data*; *Wset* 代表当前事务准备修改的数据项 X, 其中每个元素 *Wset*[X]包含准备写入的新数据 *data*.

如图 2 中, 事务 T_1 首先读取数据项 X, 然后将数据项 X 存入自己的读集. 之后, T_2 事务写数据项 X, 此时 T_2 仅将新数据存入写集, 并不实际修改数据项 X. T_2 验证操作时发现其并发事务不存在写集, 因此 T_2 正常提交并将写集中的数据写入数据库. 之后, T_1 在验证操作时, 通过遍历其他事务的写集, 发现与 T_2 存在读写冲突, 于是 T_1 回滚, 从而避免丢失修改异常.

但是该算法存在两个缺点: 首先, 只读事务也需要进行验证; 其次, 需要在系统中额外维护大量事务的写集并对其进行检查, 带来了较大的内存开销和验证开销. 因此, 针对原始 OCC 的缺点, Härder[40]重新归纳该算法, 将其命名为为 BOCC (backward validation), 同时提出了 FOCC (forward validation)算法. 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. 相较于 BOCC, FOCC 的只读事务无须验证, 整体上减少了验证的开销. 针对如图 7 中丢失修改的异常, FOCC 算法中, T_2 验证时通过遍历活跃事务的读集, 发现与事务 T_1 存在读写冲突, 因此自己回滚, 从而保证可串行化.

以上两种算法 BOCC 和 FOCC 的验证和结束阶段都必须在临界区中执行, 首先, 临界区会降低事务验证的并发度; 此外, 也导致这两种方法难以应用到分布式系统中. 为此, 原论文提出了并发验证的 OCC 方法[7],

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段以增加并发度. 事务 *T* 验证前先在第 1 个临界区内将自己的写集存入 *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*. 之后, 事务 *T* 通过与读集的对比, 验证两个结构内的写集. 验证通过后, 进入第 2 个临界区, 将自己从 *Active* 中清除并写入 *History*.

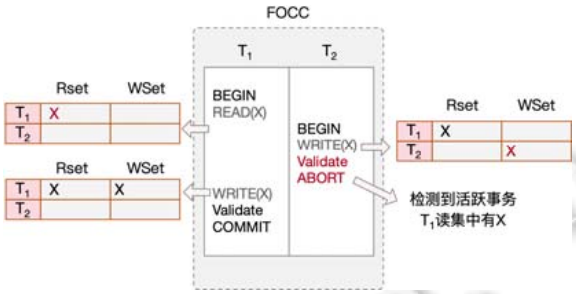


图 7 FOCC 检验例子

增加了并发优化的 OCC 算法仍然存在以下缺点: 首先, 静态定序+检验冲突这一组合导致回滚率较高; 第二, OCC 算法需要消耗大量内存维护读集写集; 第三, OCC 算法验证阶段遍历读集或写集的开销很大; 第四, 临界区的存在进一步降低了算法性能. 因此, 传统的 OCC 类算法在实际的工业场景中并不多见.

3.2 Silo

由于传统 OCC 存在诸多缺陷, 2013 年提出了 OCC 的优化算法 Silo. Silo 算法位于图 4 中的第二象限, 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. 为此, Silo 需要在数据项上维护数据项修改时间戳 *wts* 用于判断读写冲突, 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 8 中, 事务 T_2 在提交时修改数据项的 *wts*, T_1 验证时发现数据项 *X* 的 *wts* 与读操作时不同, 从而回滚.

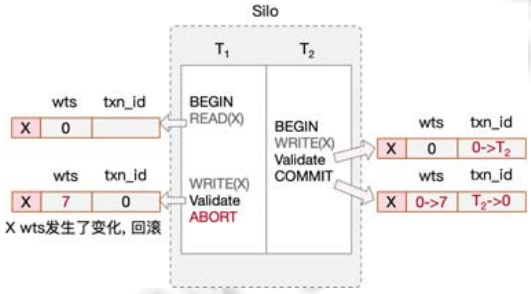


图 8 Silo 检验例子

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.

3.3 MaaT

MaaT 是 OCC 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. 该算法位于图 4 中第四象限, 采用了动态定序+检验依赖的组合, 主要思想是: 根据事务之间的事务依赖确定事务可串行化调度中的先后顺序, 并在验证阶段, 通过调整事务时间戳范围体现先后顺序.

因此在 MaaT 算法中, 为了检测事务的读写、写读依赖, 每个数据项 *X* 需要维护: *readers*, 代表曾读取该数据项但未提交的事务; *writers*, 代表准备写该数据项但未提交的事务; *wts*, 代表最后一次写当前数据项的已提交事务时间戳; *rts*, 代表最后一次读取当前数据项的已提交事务时间戳. 每个事务 *T* 除了维护 *Rset*(读集)和 *Wset*(写集)外, 还需要维护: 时间戳范围[*lower*,*upper*)(初始化为[0,+∞)), 用于体现事务的先后顺序; *before*, 代表应在当前事务之前提交的未提交事务队列; *after*, 代表应在当前事务之后提交的未提交事务队列; *other_writes*, 代表和当前事务修改相同数据项的未提交事务队列; *max_rts*, 代表访问过的数据项中最大的 *rts*; *max_wts*, 代表访问过的数据项中最大的 *wts*.

MaaT 在解决丢失修改异常时, 首先在读写操作中, 通过维护和检查数据项的 *readers*, *writers* 和事务的 *before*, *after* 结构来发现事务之间的依赖关系. 如图 9 中, 事务 T_1 在读取数据项 *X* 后, 将自己计入数据项 *X* 的 *readers*. 之后, T_2 在写数据项 *X* 时, 通过 *readers* 发现 $T_1 \rightarrow T_2$ 的依赖关系. 因此在验证阶段, 会调整 T_1 的时间戳范围为[0,1)、T_2 范围为[1,+*Inf*)来体现两者的先后关系. T_1 进入验证阶段后, 因为 *lower* 大于数据项 *X* 的 *wts*, 所以需要将时间戳范围调整为[2,1), 该时间戳不合法, 因此事务 T_1 回滚.

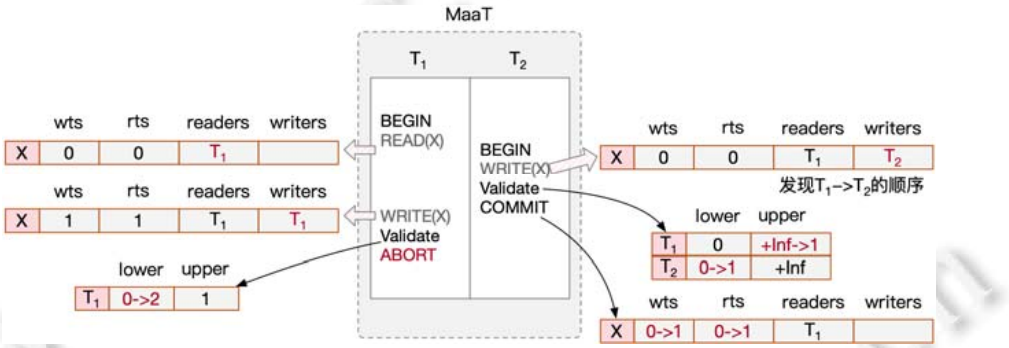


图 9 MaaT 检验例子

MaaT 通过在数据项中维护事务的访问痕迹, 记录了所有并发事务之间的先后关系, 从而动态地确定事务之间的顺序, 降低了事务的假回滚率, 但这些结构的维护以及时间戳范围的调整会带来较大的开销.

3.4 Sundial

Sundial 算法选择了动态定序+混合检验的组合, 采用动态计算提交时间戳的方式动态确定顺序, 通过结合 OCC 和 2PL 的方式检验事务是否满足先后顺序, 并通过在数据项上维护租约(数据项可被访问的事务时间戳范围)来协助判断事务之间的先后关系. 对于读写/写写冲突, Sundial 使用 OCC 验证读写和写读依赖, 通过加锁解决写写冲突. Sundial 算法中, 每个数据项需要额外维护两种元信息: *txn_id* (锁, 用于处理写写冲突)和租约(*logic-lease*, 代表数据项可以被读取的时间戳区间). 租约包括 *wts* 和 *rts* 两部分: *wts* 代表数据项最后被写入的逻辑时间, *rts* 代表当前数据项可以被读取的最晚时间. 当事务的时间戳满足 $wts \leqslant T.commit\_ts \leqslant rts$ 时, 事务可以读取该数据项. 每个事务需要维护 *commit_ts* (事务 *T* 动态计算的提交时间戳)以及 *Rset* (读集)和 *Wset* (写集), 读集中需要暂存数据项被读取时刻的租约信息.

接下来, 我们以图 10 为例介绍 Sundial 算法的主要逻辑. 假设图中数据项 *X* 的租约中 *wts* 和 *rts* 初始都为 2, 数据项 *Y* 租约中的 *wts* 和 *rts* 都为 1. 首先, 事务 T_1 在读取数据项 *X* 时调整自己的提交时间戳 *commit_ts* 为 2 (大于等于 *X.wts* 以保证写读依赖), 并将此时数据项 *X* 的元信息存入读集. 为了便于区分, 我们在读集中暂

存的元信息名称前添加 *tmp* 前缀(如 *tmp_rts*). 之后, T_2 尝试修改 *X*, 调整提交时间戳为 3 (大于 *X.rts* 以保证已提交读事务与自身的读写依赖), 并通过修改 *X* 的 *txn_id* 来加锁防止写写冲突. 接下来, T_2 继续读取数据项 *Y*, 此时已满足 T_2 的提交时间戳大于等于 *Y.wts*, 直接将数据项 *Y* 的元信息存入 T_2 的读集.

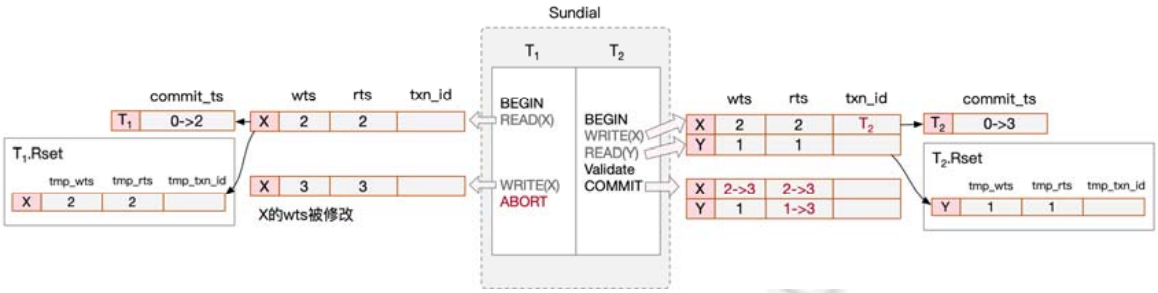


图 10 Sundial 检验例子

T_2 进入验证阶段, 通过验证读集来处理并发事务之间的读写依赖. 由于此时 T_2 的提交时间戳为 3, 大于读集中暂存的 *Y.tmp_rts*, 因此 T_2 必须调整数据项 *Y* 的 *rts* 来保证 T_2.*commit_ts*<*Y.rts*. 此时, 由于数据项 *Y* 未被其他写事务修改, 即不存在并发事务和 T_2 存在读写依赖, T_2 可以顺利修改 *Y* 的 *rts*. 最后, T_2 进入提交阶段, 修改数据项 *X* 的 *wts* 和 *rts* 为 3. 当事务 T_1 修改数据 *X* 时, 发现 *X* 的 *wts* 与读集中暂存的 *tmp_wts* 不同, 从而回滚.

Sundial 的优势在于通过动态定序实现了更低的回滚率, 在高冲突下表现更优. 相比于 MaaT, Sundial 在事务和数据项上维护的元信息更少, 有更小的内存开销.

3.5 讨 论

本节介绍了 OCC 算法及其变体算法. 其中, 传统 OCC 算法缺陷较大, 在此不再赘述. Silo 算法的优点在于检验和维护数据项元信息开销较小, 但也因此产生了回滚率较高的缺陷, 仍需消耗内存空间来存储读集和写集. 结合第 7.2 节, Silo 算法更适用于冲突率较低的场景. MaaT 算法的优势在于动态定序在一定程度上减少了事务的回滚, 但是数据项和事务中维护的大量元信息带来了较大的维护开销, 并且验证阶段中, 频繁修改事务时间戳范围的操作也很大程度上影响了性能. Sundial 算法则兼顾了动态定序的低回滚和混合检验的高效率, 在中高冲突率下性能依然较好. 在第 7.2 节的测试中, 中高冲突率下, Sundial 的性能最高可达 Silo 的 5 倍. 不过, Sundial 维护的元信息仍然较多, 因此低冲突率场景下性能依然比 Silo 算法差, 如第 7.5 节的 TPCC NewOrder 事务场景. 综上所述, 我们推荐在低冲突率下使用 Silo 算法, 在中高冲突率下选择 Sundial 算法.

从整体上看, OCC 算法多种多样, 既有适合中高冲突的 Sundial 算法, 也有适合低冲突场景的 Silo 算法. 因此, 自适应地根据冲突率调整算法检验逻辑, 可能是未来的一个研究方向.

4 时间戳排序算法

T/O 是基于时间戳的并发控制算法, 选择了静态定序+检验依赖的组合方式. 其主要思路是: 在事务开始时, 为其分配开始时间戳, 并按照开始时间戳对事务进行排序. 在执行读写操作时, 检测当前事务的实际执行顺序是否违背预先规定的顺序: 如果与预定顺序不符, 当前事务需要回滚或者陷入等待. 为了实现这一功能, T/O 算法需要在事务结构中维护事务的 *start_ts*, 即事务的开始时间戳. 数据项上需要维护如下字段: *data*, 代表数据项的值; *wts*, 代表写入当前数据项的事务时间戳; *rts*, 代表最后一次读取当前数据项的事务时间戳.

事务通过维护数据项的 *wts* 和 *rts* 来记录事务对数据项的修改, 从而检查事务之间的依赖关系. 如图 3 中, 假设事务 T_1 和 T_2 的时间戳分别为 5 和 7. 事务 T_1 在读取数据项 *X* 时更新 *rts*, T_2 写 *X* 时发现存在时间戳为 5 的事务曾读取过 *X*, 但 T_2 时间戳更大, 可以继续执行, 并修改 *wts* 为 7. 当 T_1 同样打算修改 *X* 时, 将通过 *wts* 检查到 $T_2 \xrightarrow{ww} T_1$ 的写写依赖关系, 并因该依赖关系与时间戳顺序 $T_1 \rightarrow T_2$ 相反而回滚.

基础 T/O 算法在回滚时会遇到问题: 若事务 T_1 修改了数据项 *X*, 之后, T_2 在 T_1 的基础上再次修改 *X*, 此时

若 T_2 提交而 T_1 回滚, 将无法确定 X 应当回滚到的版本. 因此在 T/O 基础上引入两阶段提交, 增加预写操作[5]来解决这一问题. 即事务写操作时不将新数据写入数据库, 而是将新数据暂存, 待事务提交时再将新数据写入. 预写操作解决了无法处理回滚的问题, 但会影响读取操作的正常执行: 假如一个时间戳更大的读事务在读取数据时发现一个时间戳更小的写事务尚未提交, 为了满足时间戳顺序, 读事务必须等待.

为此, 数据项上需要增加两个字段: *min_pts*, 代表当前数据项上所有预写操作的最小时间戳, 用于判断读操作是否需要等待; *min_rts*, 代表当前数据项上所有预读操作的最小时间戳, 用于判断写操作是否要等待. 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. 如图 11 所示, 事务 T_2 和 T_1 在进行写操作时都会将自己计入 *pre_reqs*, 在提交时根据时间戳先后顺序先后修改 *wts*, 并将新数据写入数据库(T_1 先提交, T_2 后提交).

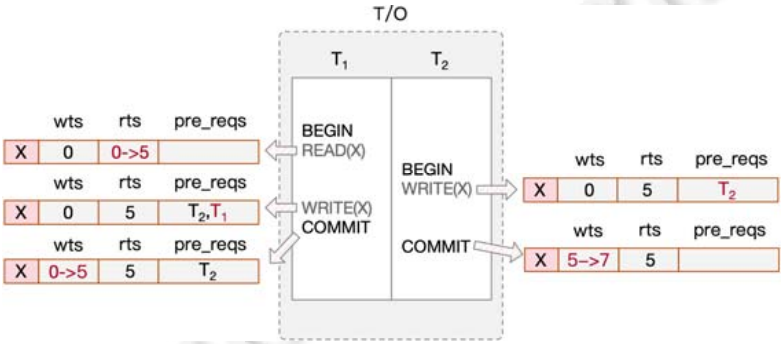


图 11 引入预写的 T/O 算法检验例子

- 讨论

本节提到了基础 T/O 和增加两阶段提交的 T/O 两种算法. 其中, 基础 T/O 算法优势在于简单、容易理解, 但是存在无法处理回滚的缺陷. 增加两阶段提交的 T/O 算法优势在于解决了回滚处理的问题, 同时支持分布式场景, 但是需要额外维护事务等待队列和事务写集, 消耗了更大的内存空间, 带来了更多的维护处理开销.

整体上, T/O 算法利用静态定序+检验依赖的方式平衡了事务回滚率和元信息维护的开销, 可以在低冲突率或高冲突率下都保证较好的性能. 但是 T/O 算法由于采用了静态定序, 在高冲突率下依然会有较大的性能损耗.

5 多版本并发控制算法

之前介绍的并发控制机制在处理读写冲突时, 必然要延迟或回滚一个事务来保证可串行化. 比如, 一个读操作要读取的数据已经被一个写操作修改, 这些的并发控制机制都会选择将读操作回滚. 但是, 如果将读操作要读取的旧值维护起来, 该操作就可以通过读取旧值来正常执行. 这一方法被称为多版本并发控制(MVCC), 数据项上维护的每个值被称作版本. MVCC 是目前数据库中最普遍使用的并发控制算法, 最早在 1978 年由 David Patrick Reed[35]提出. 由于每个数据项上维护多个物理版本的特性, MVCC 可以将事务对数据项的操作转化为对数据项版本的操作, 从而提高并发度, 提供读写互不阻塞的能力. 但是 MVCC 本身没有处理事务冲突的机制, 所以一般需要结合 T/O、2PL、OCC 来做到可串行化.

5.1 时间戳排序机制+多版本并发控制

在 MVCC 中, 将时间戳作为读取数据项某个版本的依据是一个最常用的技术. 因此, 将 MVCC 与 T/O 算法结合的 MVTO 是最直接的手段. 该方法选择了静态定序+检验依赖的方式, 其整体思路与 T/O 相同: 通过事务的开始时间戳确定事务顺序, 在读写操作时, 检测实际执行产生的 3 种依赖是否违背时间戳规定的顺序. 在 MVTO 算法中, 每个事务在其开始时都被分配一个开始时间戳 *start_ts*. 对于数据项的每个版本, 需要维护以

下信息来帮助检查事务之间的依赖关系: *data*, 代表数据项的值; *wts*, 代表写入当前版本的事务时间戳, 也代表版本的提交时间戳, 当事务的 *start_ts* 大于 *wts* 时, 该版本对事务可见; *rts*, 代表成功读过该版本的所有事务的时间戳的最大值, 当事务的 *start_ts* 大于所有版本的 *rts* 时, 该事务可对当前数据项进行修改; *txn_id*, 代表当前正在写入该版本的事务号, 可以理解为写锁, 用于避免写写冲突.

如图 12 所示, 在利用 MVTO 解决丢失修改异常时, 事务 T_2 会根据 MVCC 算法插入新版本 X_1, 并更新 X_0 版本的 *rts*. 当事务 T_1 也要写入新版本时, 由于新版本 X_1 的时间戳大于 T_1 的开始时间戳, T_1 只能回滚.

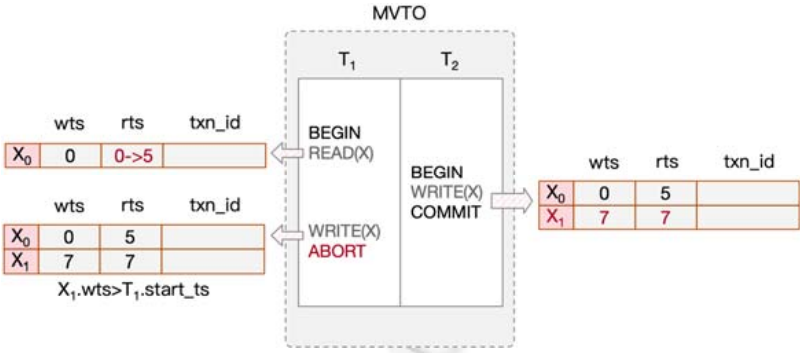


图 12 MVTO 算法检验例子

T/O 算法结合 MVCC 后, 读写操作互相不冲突, 带来了更高的并发度, 减少了不必要的事务回滚.

5.2 两阶段封锁+多版本并发控制

MVCC 也可以应用在 2PL 中. 该方法叫做 MV2PL, 通过加锁的先后顺序为事务定序, 在读写操作阶段, 借助锁机制处理事务之间的冲突. 与原始 2PL 的区别在于: 多版本下的读写锁等信息被维护在数据项的每一个版本上, 包括 *txn_id*(写锁), *read_cnt*(读锁数量)以及 *wts*(当前版本写入时间戳). 事务在进行读写操作之前需要获取一个开始时间戳 *start_ts*. 之后, 事务开始执行读写操作. 针对丢失修改异常, 2PL+MVCC 通过对最新版本加锁避免并发事务的写入, 如图 13, T_1 在 X_0 上的读锁, 避免了 T_2 对 X 的并发修改.

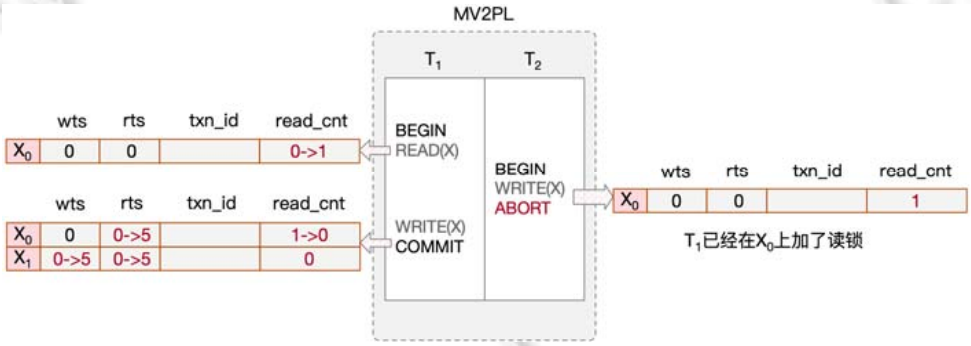


图 13 两阶段封锁+多版本算法检验例子

2PL 算法结合 MVCC 后, 读写操作互相不冲突, 带来了更高的并发度, 减少了不必要的回滚.

目前, MV2PL 已经被广泛应用到了 Postgres[41]、Oracle[42]等数据库系统中.

5.3 乐观并发控制+多版本并发控制

Cicada 是 OCC 结合 MVCC 的并发控制算法, 可以视为 Silo 算法结合 MVCC,选择了静态定序+检验冲突的方式. Cicada 中, 事务按照开始时间戳确定顺序, 在验证阶段检查是否存在 3 种冲突.

Cicada 的版本链中除数据本身以外, 每个版本还需要维护如图 14 中的 3 种元信息: (1) *status*, 代表当前版本的状态(PENDING 代表当前版本尚未提交, COMMITTED 代表当前版本已经提交, ABORTED 代表当前版本已被回滚); (2) *wts*, 代表当前版本的提交时间戳; (3) *rts*, 代表读取过当前版本的已提交事务的最大时间戳.

图 14 Cicada 版本链结构

Cicada 执行事务的主要流程如图 15 所示, 首先, 读操作从版本链中读取满足时间戳条件的版本, 若当前版本为 PENDING 状态, 则等待该版本提交后再读. 写操作时首先判断能否写入数据, 若数据项的 *rts* 和 *wts* 均不满足时间戳读取的条件, 则事务需要回滚. Cicada 的验证阶段分为 3 步.

- 首先, 在无 PENDING 状态版本的情况下插入新版本, 并将其状态设置为 PENDING.
- 然后更新读集的 *rts*, 并检查读集数据是否被修改.
- 最后, 再次检查写操作是否满足条件: 若验证通过, 则将 PENDING 版本状态改为 COMMIMTTED; 否则改为 ABORTED.

图 15 Cicada 算法流程

在 Cicada 的检验逻辑下, 对于丢失修改异常, 图 16 中 T_2 验证提交时更新 X_0 版本的 *rts*, 并插入新版本 X_1. T_1 在验证阶段发现 T_2 写入了新版本 X_1, 且 $X_1.wts$ 大于 T_1 的开始时间戳, 因此 T_1 无法写入必须回滚, 避免丢失修改异常.

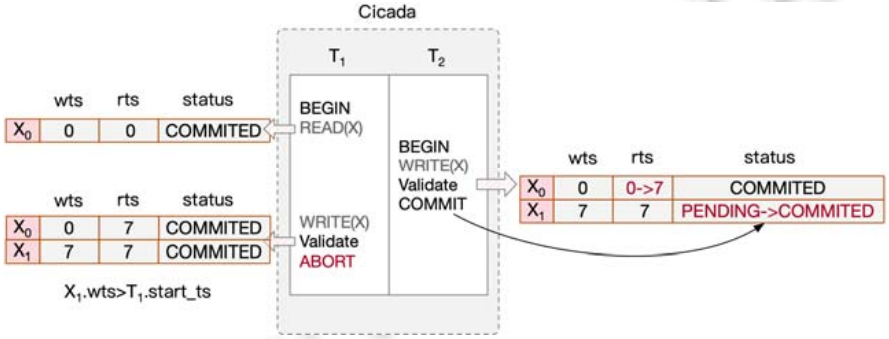


图 16 Cicada 算法检验例子

Cicada 由于采用了 MVCC 机制, 读写互不冲突, 与纯粹的静态定序+检验冲突方式相比有更高的并发度、更低的回滚率. 相比于 Silo, 其性能理论上更有优势.

5.4 快照隔离类算法

快照隔离(SI)[20]是一种基于 MVCC 的并发控制算法, 是静态定序+检验依赖的组合方式, 其中, 读写混合事务根据提交时间戳、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. 对于写写冲突, SI 规定: 数据项不能被两个事务同时修改, 并遵循"先提交者获胜策略". 即先提交的事务成功, 后提交的事务回滚. SI 在 MVCC 方法的基础上, 利用快照读(读取事务开始时符合一致性状态的数据)处理写读依赖. 因此, SI 具有写读互不阻塞的优势. 但是由于没有对读写依赖进行限定, SI 算法并不能达到可串行化. 目前, 主流的两种达到可串行化的方法有 SSI 和 WSI.

5.4.1 SSI

SSI 通过理论证明: 只要在 SI 基础上避免出现 T_i 读写依赖于 T_j、T_j 读写依赖于 T_k 的情况, 就能达到可串行化. 因此, SSI 算法的核心就是在 SI 的基础上检测连续的读写依赖. 我们将这样的思路归纳为静态定序+混合检验. SSI 在每个数据项上维护两个结构: (1) *txn_id*, 用于基础 SI 算法处理写写冲突; (2) *SIRead-Lock*, 记录读取过该数据项的所有事务, 用于协助事务判断是否存在读写依赖. 每个版本需要记录提交时间戳 *wts* 和写入该版本的事务号 *creator*. 每个事务 *T* 上需要维护开始时间戳 *start_ts* 和提交时间戳 *commit_ts*. 此外, 为了检测读写依赖, 还需要额外维护 *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.

对于丢失修改异常, 如图 17, 事务 T_1 在读取数据项时需要添加 *SIRead-Lock*, 之后与 X_0 版本的创建者检查是否存在两个连续的读写依赖. 之后, 事务 T_2 写数据项 *X*, 在 *X* 上加写锁, 并检查是否存在两个读写依赖. 这里, T_2 根据 T_1 的 *SIRead-Lock* 检验出 $T_1 \rightarrow T_2$ 的读写依赖, 但是不存在第 2 个读写依赖(即 $T_n \rightarrow T_1$ 或 $T_2 \rightarrow T_m$), 因此 T_2 可以正常提交. 但 T_1 准备写数据项 *X* 时, 会因为发现 X_1 版本的 *wts* 大于自己的开始时间戳而回滚.

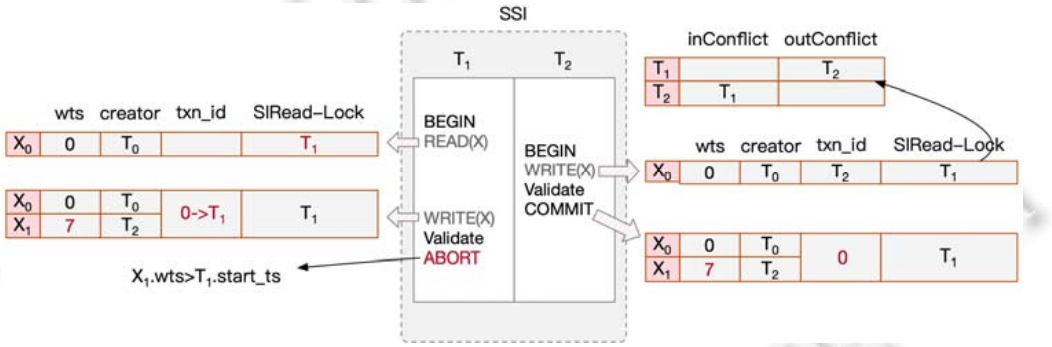


图 17 SSI 算法检验例子

SSI 算法在快照隔离的基础上做到了可串行化, 与 2PL 和 OCC 算法相比, 并发度更高, 因为在 2PL 和 OCC 算法中不允许出现读写依赖. 但是为了寻找读写依赖, SSI 需要维护事务的 inConflict 和 outConflict 结构, 增加了维护开销. 目前, Postgres 系统中[41]已经将 SSI 应用到了数据库中, 并对 SSI 读写依赖的处理进行了进一步优化.

5.4.2 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.

在 WSI 中,每个事务需要维护开始时间戳 *start_ts* 和提交时间戳 *commit_ts*; 每个数据项需要维护最大提交时间戳 *lastCommit*. 在 3 种依赖关系的检测中, 事务在验证阶段检验读集数据项的 *lastCommit* 是否发生改变来检查读写依赖, 如果发生改变, 则说明在读操作后有写事务写入了新版本, 存在读写依赖, 此时需要回滚当前事务. 针对写读依赖, SI 的快照读只会读到当前事务之前的已提交版本, 保证了写事务排在读事务之前的写读依赖. 最后, WSI 没有检测写写依赖的机制, 因为写写依赖对事务的先后顺序不作要求.

WSI 算法处理丢失修改异常, T_1 发现读集数据项 *X* 的 *lastCommit* 发生变化, 进而得知自己需要回滚(如图 18 所示).

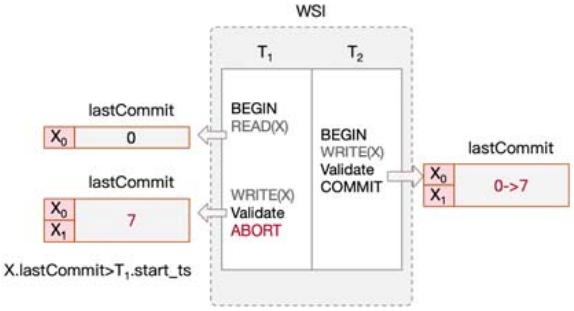


图 18 WSI 算法检验例子

此外, 只读事务不会修改数据和元信息, 所以 WSI 中只读事务无须验证. 相比于 SSI 算法, WSI 不允许并发事务出现读写依赖的限制虽然会导致更高的回滚率, 但是节省了检验写写冲突和维护 inConflict 等结构的开销. 结合第 7.2 节–第 7.5 节的实验可知, WSI 算法的性能总体表现最优.

5.5 讨 论

在本节介绍的基于 MVCC 的算法中, T/O、OCC、2PL 等算法在结合 MVCC 后虽然带来维护版本链的开销,但是增加了并发度, 降低了回滚率. 第 7.1 节 MVTO 算法的测试结果也证实了这一点. 因此在实际系统的算法选择中, 更推荐使用基于 MVCC 的算法.

MVCC 类算法的优缺点各有不同: MVTO 存在与 T/O 算法相似的优缺点, 并优化了 T/O 算法在高冲突率下的表现; MV2PL 优化了高冲突率下的性能, 但是受 2PL 高冲突率的限制, 其性能理论上低于 MVTO; Cicada 的优点是高冲突率下回滚率低, 缺点是版本链元信息的维护开销会影响低冲突率下的性能; 快照隔离类算法中, SSI 算法利用混合检验降低了回滚率, 但读写依赖的检验方式比较粗糙, 且带来了较大的维护检查开销; WSI 算法只检查读写依赖, 因此只需维护很少的元信息. 此外, 基于检验依赖的方式回滚率较低. 因此, 如第 7 节实验结果所示, WSI 算法的性能总体表现最优. 综上所述, 目前与 MVCC 结合的算法都选择了静态定序的方式, 动态定序的并发控制算法与 MVCC 的结合较少, 可以作为未来的研究方向.

6 确定性并发控制算法——Calvin

确定性的并发控制算法 Calvin 默认应用在如存储过程等预先知道事务的全部 SQL 语句的场景中, 其主要思想是: 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. Calvin 中额外包含两个模块: 定序器(sequencer)和调度器(scheduler). 其中, 定序器用于拦截事务并且为这些事务确定顺序, 即事务进入定序器的顺序; 调度器负责按照事务顺序为事务加锁, 保证事务的执行严格按照确定顺序.

Calvin 算法中, 一个事务 T 的执行流程为: 事务 T 被定序器截获并放入一个 batch 中, 根据事务的截获顺序排序. 在经过一个固定周期后, 定序器将事务 T 所在的 batch 发送给对应的参与者节点. 参与者节点的调度器接收 batch, 根据 batch 中事务的顺序加锁. 以图 19 为例, $Server_1$ 的 batch 中存在事务 T_1 和 T_2, 且要求 T_1 排在 T_2 之前. 其中, T_1 写数据项 A 且读数据项 B, T_2 读数据项 A 且写数据项 C. 调度器首先为 T_1 的读写操作加锁(图 19 左侧), 再为 T_2 的读写操作加锁, 此时, 数据项 A 已有 T_1 的写锁, 因此将 T_2 存入数据项 A 的 WaitList 等待队列(图 19 右侧). 该操作保证了实际执行时, T_2 必须等待 T_1 结束才可以读取到数据项 A. 加锁完成后, 执行器执行 batch 中的事务, 首先分析事务的读写集, 然后执行本地读. 在本地读执行完毕后, 同步各个节点上子事务的读写集, 以解决诸如本地写操作需要远程读取结果的问题. 在同步完读写集后, 执行本地的写操作, 完成事务的执行.

- 讨论

Calvin 算法的优势在于事务严格按照顺序执行, 不存在回滚事务; 其次, 事务分解而成的子事务并发执行, 提高了事务执行的效率. 缺点在于 Calvin 算法存在调度器这一单点瓶颈, 是否存在优化方法还有待研究;

其次, Calvin 的定序方法未能发挥预先得知事务读写集的优势, 将动态定序应用到其中可以给出更好的定序方案.

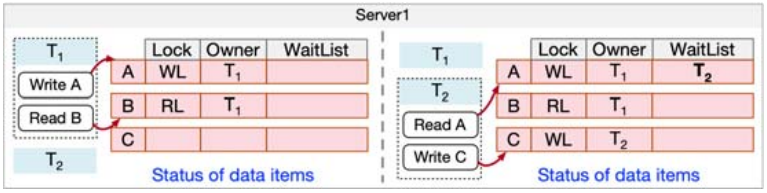


图 19 Calvin 检验例子

7 并发控制算法实验评价

根据并发控制算法的分类, 本文选取了具有代表性的 6 种算法: Wait Die、Silo、Sundial、MVTO、WSI、Calvin 进行评估, 研究它们在分布式内存数据库场景下的适用性. 为了保证实验的公平性, 我们将这些算法统一在开源内存型分布式事务测试床 3TS (代码地址: https://github.com/Tencent/3TS)下进行了实现, 相关的代码已经开源. 实验环境由 4 个虚拟节点组成, 每个节点都包含 4 核/8 线程, 并且拥有 32 GB 的内存, 各个节点之前的网络时延 RTT 大约在 0.3 ms 左右. 每一轮测试会进行 60 s, 并且在测试之前存在 60 s 的预热时间. 评估算法优劣的主要因素为: (1) 吞吐量, 代表平均每秒成功提交事务数; (2) 回滚率, 代表回滚的事务占总执行事务的百分比.

7.1 工作负载

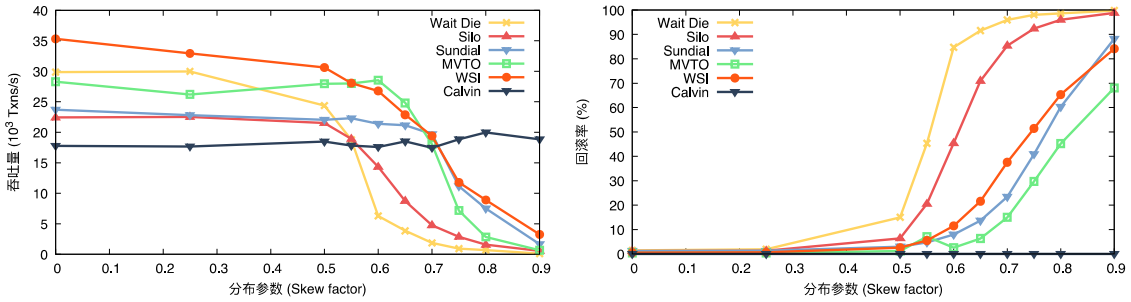
性能测试使用 YCSB 和 TPCC 两种负载.

- YCSB (Yahoo! cloud serving benchmark)[43]是雅虎公司开源的数据库服务器端压力测试工具, 它提供了可调试参数如写操作比例、冲突率等. 事务访问服从 Zipf 分布, 即少量数据获得大量访问的长尾分布. 分布参数(又称 skew factor)在 0–1 之间, 越接近 1, 数据访问冲突越大, 用于测试系统在不同冲突率下的性能表现. 默认我们使用 workload-a 来进行性能测试(写操作占比 50%, 分布参数 0.5).
- TPCC[44]则是一种流行的 OLTP 基准, 模拟了一个仓库订单处理应用程序, 其中包含 9 张表, 每个仓库包含 100 MB 的数据大小. 默认情况下, 我们为每个节点设置 32 个仓库. TPCC 包含 5 种事务: NewOrder、Payment、Delivery、Stock-level 和 Order-status, 其中只有 NewOrder 和 Payment 是分布式事务, 占据了 TPCC 中事务总数的 88%. 在接下来的测试中, 我们主要使用 NewOrder 和 Payment 来测试并控制算法在 TPCC 下的可扩展性.

7.2 冲突率

首先测试算法在不同冲突场景下的吞吐量和回滚率的变化, 这是影响 OLTP 系统性能的关键因素. 通过调整 YCSB 负载的分布参数从 0.0 到 0.9, 来观察不同算法在不同冲突率下的性能变化.

从图 20 中可以看出, 在低冲突场景下, 即分布参数小于 0.5 时, WSI 算法性能最优, MVTO、Wait Die 次之;在中冲突场景下, 即分布参数在 0.5–0.7 之间时, MVTO 算法性能最优, WSI、Sundial 次之, 除 Calvin 外, 所有算法都出现了明显的下降; 在高冲突场景下, 即分布参数在 0.75 以上时, Calvin 算法最优, 此时除 Calvin 算法外, 性能都表现很差. 而 Calvin 的性能可以在任何分布参数下保持稳定, 其原因有 3 个: 首先, Calvin 的所有事务都会由调度器统一加锁, 因此不会加锁失败而回滚, 从图 20(b)中也可以看出, Calvin 算法的回滚率为 0; 其次, 当事务获取到全部的锁后, 就可以立刻执行全部操作并释放锁, 这意味着锁持有时间比较短, 降低了对其他冲突事务的影响; 最后, Calvin 各个节点的子事务之间几乎不需要通信, 减少了事务调度和网络通信带来的开销. 但是, 虽然 Calvin 具有以上优势, 由于调度器的单点瓶颈, Calvin 在冲突率较低的情况下性能并不能达到最优.



(a) 吞吐量 (b) 回滚率

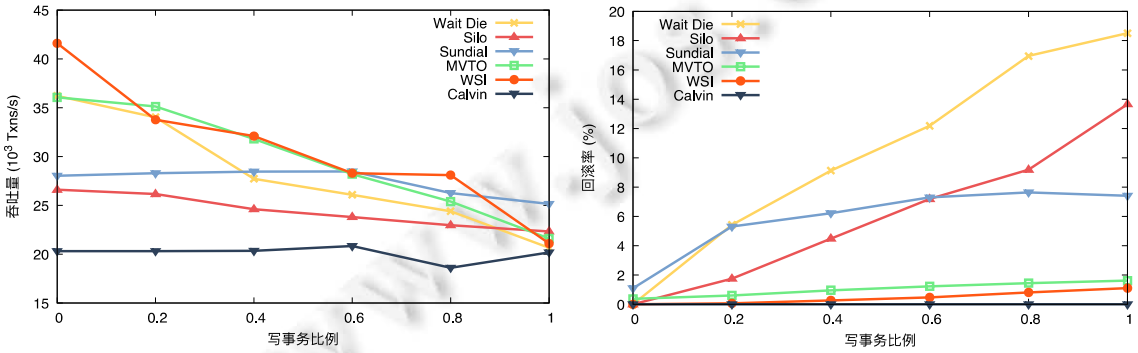
图 20 YCSB 复杂冲突率测试

Wait Die 算法在分布参数小于 0.5 时性能比较优秀, 但是分布参数大于 0.5 时, 其性能因为回滚率增大而下降(图 20(b)). MVTO 算法在分布参数较低的情况下性能比 Wait Die 差, 这是因为冲突率小的场景下无法发挥 MVTO 读写不冲突的优势, 还会因为更长的版本链产生增大访问开销的负面影响. 当分布参数在 0.6 时, MVTO 性能达到最优; 但是随着冲突率继续提升, MVTO 也因为回滚率增大而性能快速下降. WSI 算法相比于除 Calvin 外的其他算法都较优, 这是因为 WSI 基于 SI, 能够发挥多版本的优势, 再加上 WSI 的验证阶段开销很少, 因此在各个冲突场景下也都表现良好. Sundial 算法由于动态定序, 在高冲突下的性能表现与 WSI 相当; 但是在低冲突率下受制于数据项上的元信息维护, 性能较差. 此外, Silo 算法在低冲突场景下性能较差的表现与传统集中式环境下的观察不同, 这是因为分布式场景下 2PC 的影响. 由于 Silo 算法需要验证读集, 因此只读事务也需要进行 2PC, 所以表现不佳.

7.3 写操作比例

接下来测试事务中写操作比例对不同算法性能的影响. 通过调整 YCSB 负载的写操作比例从 0.0 到 1.0 来进行测试. 此时, YCSB 负载的分布参数设置为 0.5.

从图 21 中可以看出, 除 Calvin 和 Sundial 以外的算法, 吞吐量都随着写事务比例的升高而下降. 下降比较明显的是 WSI、Wait Die 和 MVTO 算法. 其中, WSI 算法的性能随着写事务比例从 0 到 0.2 下降了 18.8%, WSI 算法只读事务无须验证, 因此在只读场景下性能较优. 写比例升高后, 只读事务比例下降, 从而导致吞吐量下降. No Wait 算法则是因为事务回滚率随着写比例升高而明显上升(图 21(b)), 从而导致性能下降. MVTO 算法随着写事务比例增加, 版本链长度变长从而降低了读写效率. Calvin 算法虽然受写比例影响较少, 但是其性能由于调度器单点瓶颈并不占据优势. Sundial 算法拥有动态定序的优势, 因此受写比例影响也比较小, 但是在写比例小于 0.6 时, 动态定序带来的开销也导致其性能较差.



(a) 吞吐量 (b) 回滚率

图 21 YCSB 写事务比例变化测试

7.4 事务涉及节点数

本节讨论分布式事务的性能, 设置了 16 个虚拟节点构成的分布环境. 通过调整 YCSB 负载中事务最大涉及节点数量从 1 到 16, 来分析不同算法的性能表现.

从图 22 中可以发现, 当涉及节点数从 1 增加到 2 时, 所有算法都呈现出明显下降的趋势; 从 2 节点到 4 节点, 各种算法仍然有 24%–58%的性能下降. 这是因为随着节点数的增加, 事务需要消耗更多时间用于远程消息的发送和接收, 并且增加了更多节点之间的调度开销. 而增加的网络通信时间和调度时间增大了事务的延时, 导致了事务冲突率的上升, 进一步导致性能的下降. Calvin 在本轮测试中受到的影响比其他算法小, 在跨节点数为 4, 8, 16 时, 性能比其他算法都更优. 这是因为 Calvin 各个节点上子事务可以并发执行, 而且子事务之间的网络开销和调度开销仅有 1 次, 因此在跨节点数较多的情况下, Calvin 效果较好.

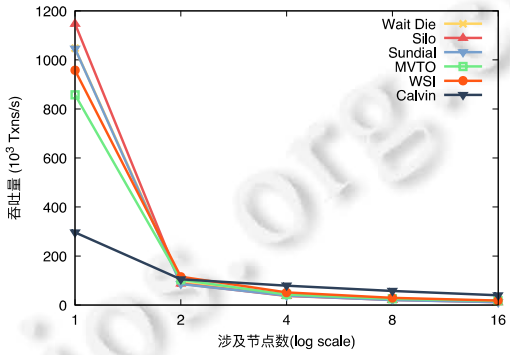


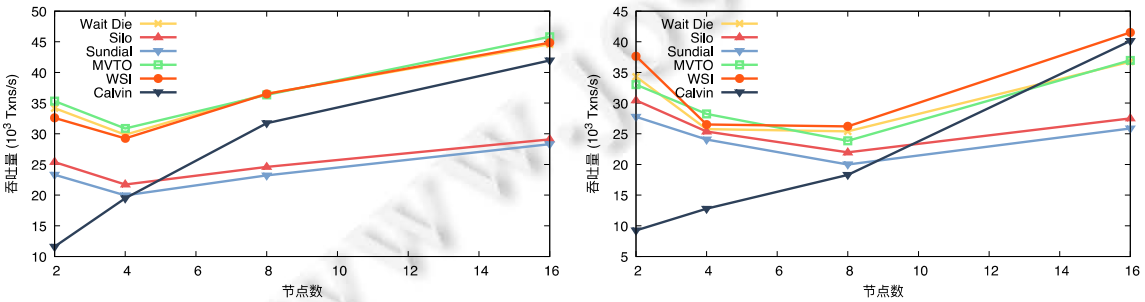
图 22 YCSB 涉及节点数测试

7.5 可扩展性

本小节评估不同算法对于节点数的可扩展性, 固定事务的最大涉及节点数, 调整分布环境中的总节点数从 2 到 16 来分析不同算法的性能表现. 我们设计了 4 种场景, 分别是 YCSB 负载下只读事务、YCSB 负载下中冲突事务(分布比例为 0.5)、TPCC 负载的 NewOrder 事务以及 TPCC 负载的 Payment 事务.

Payment 场景下, 所有事务都需要修改 warehouse 表中的支付信息, 而一个节点上仅有 32 条 warehouse 信息, 因此对 warehouse 的修改是 Payment 场景的瓶颈. NewOrder 场景是创建一个新订单, 并购入 5–15 项产品, 其中, 产品的信息可能存储在远程节点上. 该事务的瓶颈在于并发的更新 district 表中的订单序号, 不过一个节点中有 320 条 district, 因此 NewOrder 的冲突率比 Payment 要小.

图 23 表现了 YCSB 两种场景下的可扩展性情况. 除 Calvin、Sundial 外, 所有算法从 2 节点到 4 节点都存在一定的下降, 之后的性能基本可以达到线性可扩展. 这是因为 YCSB 负载中一个事务包含 10 个读写操作, 而随着节点数的增多, 事务可能会访问更多的节点从而导致性能的下降.

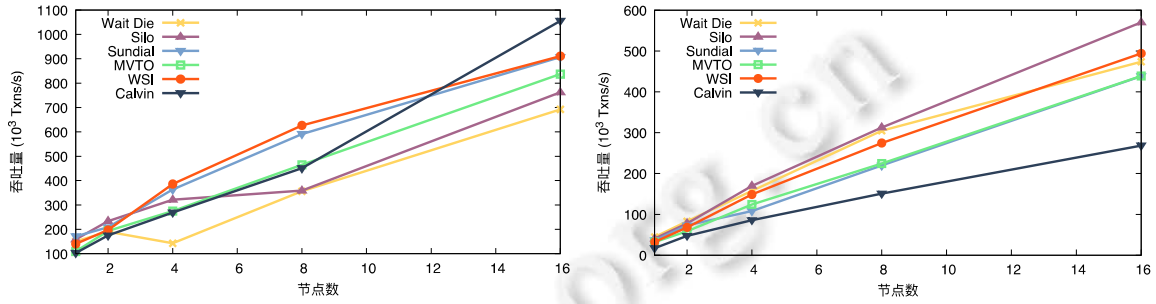


(a) 只读事务 (b) 中冲突场景

图 23 YCSB 可扩展性测试

在 YCSB 的只读可扩展性测试(图 23(a))中, Wait Die、WSI、MVTO 这 3 种算法的性能较优; 而在中冲突率场景下(图 23(b)), 随着事务冲突率的提高, 3 种算法虽然仍然有一定的优势, 但是优势已经不是特别明显. 这里值得关注的是, Calvin 算法在中冲突率下, 随着节点数的增多, 性能提升十分明显.

图 24 中表现了 TPCC 两种事务的可扩展性情况. 在 TPCC 的 Payment 负载中(图 24(a)), Calvin、WSI、Sundial 这 3 种算法的可扩展性要更好. 这是因为 Payment 负载中的冲突率较大, 而 Calvin、WSI、Sundial 这 3 种算法在高冲突率场景下的性能都比较好(如第 7.2 节所示). 在 TPCC 的 NewOrder 负载下(图 24(b)), Silo、Wait Die 等方法的可扩展性要更优. 这是因为这些算法更适用于低冲突率场景.



(a) Payment 事务 (b) NewOrder 事务

图 24 TPCC 可扩展性测试

7.6 实验结果总结

根据第 7.2 节–第 7.5 节的实验结果, 表 4 给出了各种算法的适用场景. 若算法的性能比起其他算法性能更差, 则表示为(∇); 若优于算法性能的平均值, 则表示为(Δ); 若算法是该场景下的最优选择, 则表示为(ΔΔ).

以 Wait Die 为例的 2PL 算法拥有方法简单、元信息维护开销小的优势, 因此在低冲突场景(分布参数小于 0.5)下性能表现较好; 而在中高冲突场景下, 由于回滚率高的缺陷, 性能表现较差. 以 MVTO 为例的时间戳排序算法则兼顾了事务回滚率和元信息的维护开销, 因此在低中冲突负载下性能表现较好. WSI 算法由于验证效率高而且回滚率低, 因此在所有负载场景中总体表现最优. Sundial 算法存在元信息维护的开销, 因此在低冲突场景下性能较差; 不过动态定序方式带来的低回滚率, 导致 Sundial 在中高场景下表现较优. Silo 算法虽然有着元信息开销较小的优势, 但是低冲突场景下其效率仍然无法与 Wait Die、MVTO、WSI 等算法相比;而中高冲突场景下, 其较高的回滚率导致性能表现不佳. Calvin 算法由于 0 回滚、不受负载影响的优势,在高冲突场景和分布式场景下表现最优; 在低中冲突率下, 单点瓶颈的存在使得其性能仍然很差. 值得强调的是, 表 4 中验证了 Calvin 类确定性并发控制算法不受冲突率、读写事务比例等因素的影响, 性能在这些场景下十分稳定. 此外, 在跨节点数较高的情况下, 性能比起其他算法来说都占据明显的优势.

表 4 适用场景总结

并发控制算法	定序和检验方法	低冲突场景	中冲突场景	高冲突场景	分布式场景
Wait Die	静态定序+检验冲突	Δ	∇	∇	∇
MVTO	静态定序+检验依赖	Δ	ΔΔ	∇	∇
WSI	静态定序+检验依赖	ΔΔ	Δ	Δ	Δ
Sundial	动态定序+混合检验	∇	Δ	Δ	∇
Silo	静态定序+检验冲突	∇	∇	∇	Δ
Calvin	静态定序+检验冲突	∇	∇	ΔΔ	ΔΔ

此外, 静态定序方法+检验依赖的方法如 WSI, 是实验中性能最好的结合方式. 这是因为静态定序方法既避免了动态定序额外引入的定序开销, 也通过检验依赖降低了回滚率. 此外, 检验依赖和依赖冲突混合检验的方法拥有较低的回滚率, 更加适用于高冲突场景.

8 总结与展望

本文创新性地将现有并发控制算法的基本思想归纳为: 先定序后检验. 根据这一思想, 重新描述了现有主流的并发控制算法, 并利用统一的分布式事务测试床 3TS 对它们进行了实验探究, 得出以下结论: (1) 2PL、MVTO、WSI 算法由于元信息维护开销较少, 更适用于低冲突率的场景; (2) WSI、MVTO、Sundial 拥有回滚率小的优势, 适合于中冲突率的场景; (3) Calvin 算法的回滚率为 0, 且在高冲突下性能表现稳定, 因此适用于冲突率较高的场景; (4) Calvin、WSI 在分布式场景下性能表现较优, 因此在分布式数据库中可以尝试主要采用这些算法; (5) WSI 算法兼顾了回滚率低和验证开销小的优势, 在各种场景下总体表现最优, 适合于冲突率存在变化的场景; (6) MVTO 算法和 Sundial 算法维护的元信息开销较大, 因此实际系统在应用这两种算法时, 需要优化元信息的设计来减少元信息维护对性能的影响. 表 5 总结了各类算法的优缺点.

表 5 各类算法优缺点

并发控制算法	定序和检验方法	优点	缺点
Wait Die	静态定序+检验冲突	1. 方法简单 2. 元信息维护开销少	1. 回滚率较高 2. 高冲突下, 吞吐量下降明显
MVTO	静态定序+检验依赖	1. 回滚率较低	1. 元信息维护开销较大 2. 高冲突下, 吞吐量下降明显
WSI	静态定序+检验依赖	1. 元信息维护开销少 2. 回滚率低	1. 高冲突下, 吞吐量下降明显
Sundial	动态定序+混合检验	1. 回滚率低	1. 元信息维护开销较大 2. 高冲突下, 吞吐量下降明显
Silo	静态定序+检验冲突	1. 元信息维护开销较小	1. 回滚率较高 2. 高冲突下, 吞吐量下降明显
Calvin	静态定序+检验冲突	1. 无回滚事务 2. 不受负载影响, 性能稳定	1. 需要提前知道事务的所有操作 2. 低冲突下, 吞吐量不高

基于实验观察, 本文对并发控制算法进行了展望: 1) 动态定序的回滚率较低, 但是表示并发现事务之间的依赖关系消耗很大, 如何设计一套合适的并发控制算法有待研究; 2) Calvin 算法中, 可以利用事先得知的事务读写集构建事务依赖, 从而在定序时给出更加合理的定序方案; 3) 不同定序和检验组合适合场景不同, 提出一种自适应的方法来综合不同定序方式和检验方式的优势, 也有待进一步研究.

进一步地, 当前数据管理系统的演进, 也为并发控制算法提供了一些新的研究视角: 首先是随着存储数据量的不断增大, 分布式数据库逐渐成为主流, 因此设计适合的高性能分布式并发控制算法依然是未来主流的研究方向; 其次, 随着新硬件的不断发展, 涌现了诸如 Infiniband、RDMA 的高速网络技术, 如何使用这些新型硬件加速并发控制算法的执行, 也存在很大的研究空间; 此外, 既支持 OLAP(在线分析处理)也支持 OLTP(在线事务处理)的 HTAP 系统也是未来数据库发展的一个主要方向. 然而, OLAP 和 OLTP 的业务逻辑差异很大给系统中的并发控制带来了挑战: OLAP 以分析型业务为主, 事务普遍很长且读操作较多; OLTP 以事务处理为主, 事务较短但写操作较多. 不恰当的并发控制算法设计, 有可能导致 OLAP 长事务阻塞 OLTP 事务的执行造成性能下降. 因此在 HTAP 场景下, 设计合适的并发控制算法也是一个颇具意义的研究方向. 更进一步, 在云计算环境下, 云原生数据库要动态适应不同类型的负载, 这就要求并发控制子系统能动态选择最合适的并发控制算法.

致谢 我们向参与本文技术部分讨论的李婧瑶同学, 以及对本文提出宝贵建议的匿名评审老师一并表示衷心的感谢.

References:

[1] Wang S, Sa SX. Introduction to Database System. 5th ed., Beijing: Higher Education Press, 2014 (in Chinese).

[2] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. San Francisco: Morgan Kaufmann Publishers Inc., 1992.

[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] Gray J, Lorie RA, Putzolu GR, Traiger IL. Granularity of locks and degrees of consistency in a shared data base. In: Readings in Database Systems. 3rd ed., 1976. 365−394.

[5] Bernstein P, Goodman N. Concurrency control in distributed database systems. ACM Computing Surveys, 1981, 13(2): 185−221. [doi: 10.1145/356842.356846]

[6] 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.

[7] Kung HT, Robinson JT. On optimistic methods for concurrency control. ACM Trans. on Database Systems, 1981, 6(2): 213−226. [doi: 10.1145/319566.319567]

[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] Agrawal R, Carey MJ, Livny M. Concurrency control performance modeling: Alternatives and implications. ACM Trans. on Database Systems, 1987, 12(4): 609−654. [doi: 10.1145/32204.32220]

[10] Carey MJ, Livny M. Distributed concurrency control performance: A study of algorithms, distribution, and replication. In: Proc. of the 14th Int'l Conf. on Very Large Data Bases (VLDB '88). San Francisco: Morgan Kaufmann Publishers Inc., 13−25.

[11] Huang JD, Stankovic JA, Ramamritham K, Towsley DF. Experimental evaluation of real-time optimistic concurrency control schemes. In: Proc. of the 17th Int'l Conf. on Very Large Data Bases (VLDB '91). San Francisco: Morgan Kaufmann Publishers Inc., 1991. 35−46.

[12] 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]

[13] 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.

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

[15] Tu S, Zheng WT, Kohler E, Liskov B, Madden S. Speedy transactions in multicore in-memory databases. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP 2013). New York: Association for Computing Machinery, 2013. 18−32. [doi: 10.1145/2517349.2522713]

[16] Yu XY, Pavlo A, Sanchez D, Devadas S. TicToc: Time traveling optimistic concurrency control. In: Proc. of the 2016 Int'l Conf. on Management of Data (SIGMOD 2016). New York: Association for Computing Machinery, 2016. 1629−1642. [doi: 10.1145/2882903.2882935]

[17] Lim H, Kaminsky M, Andersen DG. Cicada: Dependably fast multi-core in-memory transactions. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data (SIGMOD 2017). New York: Association for Computing Machinery, 2017. 21−35. [doi: 10.1145/3035918.3064015]

[18] 3TS. 2020. https://github.com/Tencent/3TS

[19] Bernstein PA, Goodman N. Timestamp-based algorithms for concurrency control in distributed database systems. In: Proc. of the 6th Int'l Conf. on Very Large Data Bases—Vol.6 (VLDB '80). VLDB Endowment, 1980. 285−300.

[20] Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P. A critique of ANSI SQL isolation levels. In: Proc. of the '95 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD '95). New York: Association for Computing Machinery, 1995. 1−10. [doi: 10.1145/223784.223785]

[21] Thomson A, Diamond T, Weng SC, Ren K, Shao P, Abadi DJ. Calvin: Fast distributed transactions for partitioned database systems. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 2012). New York: Association for Computing Machinery, 2012. 1−12.

[22] Cowling J, Liskov B. Granola: Low-overhead distributed transaction coordination. In: Proc. of the 2012 USENIX Conf. on Annual Technical Conf. (USENIX ATC 2012). USENIX Association, 2012. 223−235.

[23] Escriva R, Wong B, Sirer EG. Warp: Lightweight multi-key transactions for key-value stores. arXiv: abs/1509.07815, 2015.

[24] Dragojević A, Narayanan D, Nightingale EB, Renzelmann M, Shamis A, Badam A, Castro M. No compromises: Distributed transactions with consistency, availability, and performance. In: Proc. of the 25th Symp. on Operating Systems Principles (SOSP 2015). New York: Association for Computing Machinery, 2015. 54−70.

[25] Mu S, Cui Y, Zhang Y, Lloyd W, Li JY. Extracting more concurrency from distributed transactions. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2014). USENIX Association, 2014. 479−494.

[26] Shute J, Vingralek R, Samwel B, Handy B, Whipkey C, Rollins E, Oancea M, Littlefield K, Menestrina D, Ellner S, Cieslewicz J, Rae I, Stancescu T, Apte H. F1: A distributed SQL database that scales. Proc. of the VLDB Endowment, 2013, 6(11): 1068−1079. [doi: 10.14778/2536222.2536232]

[27] Wei XD, Shi JX, Chen YZ, Chen R, Chen HB. Fast in-memory transaction processing using RDMA and HTM. In: Proc. of the 25th Symp. on Operating Systems Principles (SOSP 2015). New York: Association for Computing Machinery, 2015, 87−104. [doi: 10.1145/2815400.2815419]

[28] Zhang Y, Power R, Zhou SY, Sovran Y, Aguilera MK, Li JY. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In: Proc. of the 24th ACM Symp. on Operating Systems Principles (SOSP 2013). New York: Association for Computing Machinery, 2013. 276−291.

[29] Zhang I, Sharma NK, Szekeres A, Krishnamurthy A, Ports DRK. Building consistent transactions with inconsistent replication. ACM Trans. on Computer Systems, 2018, 35(4): Article No.12.

[30] 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]

[31] Han QL, Hao ZX. Real-time concurrency control protocol based on accessing temporal data. Journal of Software, 2007, 18(6): 1468−1476 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/1468.htm [doi: 10.1360/jos181468]

[32] Ding BL, Kot L, Demers A, Gehrke J. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In: Proc. of the 6th ACM Symp. on Cloud Computing (SoCC 2015). New York: Association for Computing Machinery, 2015. 262−275. [doi: 10.1145/2806777.2806837]

[33] Wu YJ, Arulraj J, Lin JX, Xian R, Pavlo A. An empirical evaluation of in-memory multi-version concurrency control. Proc. of the VLDB Endowment, 2017, 10(7): 781−792. [doi: 10.14778/3067421.3067427]

[34] Yabandeh M, Ferro DG. A critique of snapshot isolation. In: Proc. of the 7th ACM European Conf. on Computer Systems (EuroSys 2012). New York: Association for Computing Machinery, 2012. 155−168.

[35] Reed DP. Naming and synchronization in a decentralized computer system [Ph.D. Thesis]. Cambridge: Massachusetts Institute of Technology, 1978.

[36] Fekete A, Liarokapis D, O'Neil E, O'Neil P, Shasha D. Making snapshot isolation serializable. ACM Trans. on Database Systems, 2005, 30(2): 492−528. [doi: 10.1145/1071610.1071615]

[37] Cahill MJ, Röhm U, Fekete AD. Serializable isolation for snapshot databases. ACM Trans. on Database Systems, 2009, 34(4): Article No.20. [doi: 10.1145/1620585.1620587]

[38] Adya A, Liskov B. Weak consistency: A generalized theory and optimistic implementations for distributed transactions [Ph.D. Thesis]. Massachusetts Institute of Technology, 1999.

[39] Rosenkrantz DJ, Stearns RE, Lewis PM. System level concurrency control for distributed database systems. ACM Trans. on Database Systems, 1978, 3(2): 178−198. [doi: 10.1145/320251.320260]

[40] Härder T. Observations on optimistic concurrency control schemes. Information Systems, 1984, 9(2): 111−120.

[41] Postgres. 2021. https://www.postgresql.org/

[42] Oracle. 2021. https://www.oracle.com/index.html

[43] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symp. on Cloud Computing (SoCC 2010). New York: Association for Computing Machinery, 2010.143−154. [doi: 10.1145/1807128.1807152]

[44] TPC-C. 2021. http://www.tpc.org/tpcc/

附中文参考文献:

[1] 王珊, 萨师煊. 数据库系统概论. 第 5 版, 北京: 高等教育出版社, 2014.

[31] 韩启龙, 郝忠孝. 基于数据时态特性的实时事务并发控制. 软件学报, 2007, 18(6): 1468−1476. http://www.jos.org.cn/1000-9825/18/1468.htm [doi: 10.1360/jos181468]

赵泓尧(1997−), 男, 博士生, CCF 学生会员, 主要研究领域为分布式数据库系统, 事务处理.

赵展浩(1995−), 男, 博士生, CCF 学生会员, 主要研究领域为分布式数据库系统, 事务处理.

杨皖晴(1999−), 女, 硕士生, CCF 学生会员, 主要研究领域为分布式数据库系统, 事务处理.

卢卫(1981−), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为数据库基础理论, 大数据系统研制, 时空背景下的查询处理, 云数据库系统和应用.

李海翔(1974−), 男, 硕士, 腾讯首席架构师, CCF 专业会员, 主要研究领域为分布式计算, 云数据库, 事务处理, 查询优化.

杜小勇(1963−), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为智能信息检索, 高性能数据库, 非结构化数据管理.