3TS 腾讯事务处理技术验证系统（上）

作者：李海翔

Tencent Transaction Processing Testbed System（简称 3TS），是腾讯公司 TDSQL 团队与中国人民大学数据工程与知识工程教育部重点实验室，联合研制的面向**数据库事务处理的验证系统**。该系统旨在通过设计和构建事务（包括分布式事务）处理统一框架，并通过框架提供的访问接口，方便使用者快速构建新的并发控制算法；通过验证系统提供的测试床，可以方便用户根据应用场景的需要，对目前主流的并发控制算法在相同的测试环境下进行公平的性能比较，选择一种最佳的并发控制算法。目前，验证系统已集成 13 种主流的并发控制算法，提供了 TPC-C、Sysbench、YCSB 等常见基准测试。3TS 还进一步提供了一致性级别的测试基准，针对现阶段分布式数据库系统的井喷式发展而造成的系统选择难问题，提供一致性级别判别与性能测试比较。

3TS 系统旨在深度探索数据库事务处理相关理论与实现技术，其核心理念是：**开放、深度、进化**。开放，秉承开源之心，共享知识、共享技术；深度，践行系统化钻研之精神，对于事务处理技术的本质问题进行研究，不破楼兰终不还；进化，路漫漫其修远兮，吾将上下而求索，不断前行，不断推进。

1、3TS 整体架构

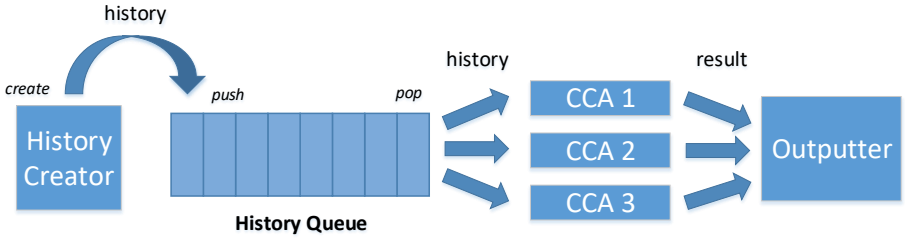
作为一个事务处理技术相关的框架，3TS 致力于探索的本质问题主要包括：

1. 世界上有多少种数据异常？数据异常的体系化研究方法如何建立？
2. 为什么并发访问控制算法会有很多种？各种并发访问控制算法之间，有没有的本质的关联关系？
3. 单机事务型数据库分布式化后，哪些方面（可用性？可靠性？安全性？一致性？扩展性？功能？性能？架构？……）会受到影响？
4. 有哪些新技术，会影响着且如何影响着分布式事务型数据库系统？
5. 分布式事务型数据库系统的评价、评测体系应该如何建立？

在代码层面，如上的每一类研究问题，都有对应的子系统。如首期开源的包括 3TS-DA 子系统混合 3TS-Deneva 子系统。

2、3TS-DA，数据异常子系统

3TS-DA 数据异常子系统，位于 src/3ts/da 路径下，其项目结构如下图所示：



1. History Creator：负责生成 history，输出给算法执行验证
2. CCA Group：CCA 即并发访问控制算法（Concurrent access control algorithm），可以对传入的 history 进行异常检测，并返回异常检测结果。
3. Outputter：负责将各个 CCA 对当前 history 的检测结果输出到文件中

3TS-DA 子系统当前功能

1. **测试数据生成**：支持三种 history 的生成方式，遍历生成、随机生成、从文本读取
2. **算法添加**：提供统一的算法接口，能够较便捷地添加新的并发算法，同时框架本身内置有多种算法，包括：可**串行化、冲突可串行化、SSI、WSI、BOCC、FOCC 等**
3. **测试指标**：框架提供多种测试指标，包括：算法回滚率、真回滚率、假回滚率、执行时间等
4. **异常扩展**：框架实现了数据异常扩展算法，能够扩展生成无限多的数据异常 history，供算法进行测试

3、3TS-Deneva，并发算法框架

Devena[1]是 MIT 开源的一个分布式内存数据库并发访问控制算法的评估框架，原始代码位于 https://github.com/mitdbg/deneva。它可以在受控环境下研究并发控制协议的性能特性。该框架提供了 **Maat、OCC、TO、Locking（No Wait、 Wait Die）、Calvin** 等主流算法。3TS-Deneva 是腾讯在 Deneva 基础上对原有系统的改进，包括多个层面。其中在算法层面，增加了更多的并发访问控制算法，包括：**可串行化、冲突可串行化、SSI、WSI、BOCC、FOCC、Sundial、Silo** 等。

3.1 基础架构

Devena 使用了自定义的引擎以及其它一些设置，可以在这个平台上部署和实现不同的并发控制协议，以进行一个尽量公平的评估。该系统的架构，如图 1 所示，主要包括两大模块：

1. 客户端实例，作为事务的发起者。客户端中的每个线程负责发起事务请求，并将发起的事务请求放到消息队列中，按序发送给服务端具体执行。客户端与服务端实例是全连接的拓扑结构，一般部署在不同的机器上；
2. 服务端实例，具体执行事务中的各类操作。不同服务端实例中存有数据，数据使用一致性哈希进行索引，从而形成服务端 IP 与所存数据的全局分区映射表，通过控制分区映射在测试期间不会修改的方法，保证该映射表是所有节点都可以准确获取的。客户端与服务端实例、服务端与服务端实例间的通信使用 TCP/IP 协议。每个服务端实例内部可被细分为四个模块：

a) 输入消息队列，暂存由客户端/其他服务端发来的消息；

a) 执行引擎，分配多个工作线程来对消息队列中的消息进行解析并实际执行，采用一

个核绑定一个线程的资源调度方式；

b) 并发控制模块，工作线程在执行事务操作的过程中，要维护特定并发控制协议要求的信息，并执行协议规定的流程，从而保证所指定的并发控制协议生效；

数据存储模块，负责管理本实例中的数据，将数据都放在内存中。

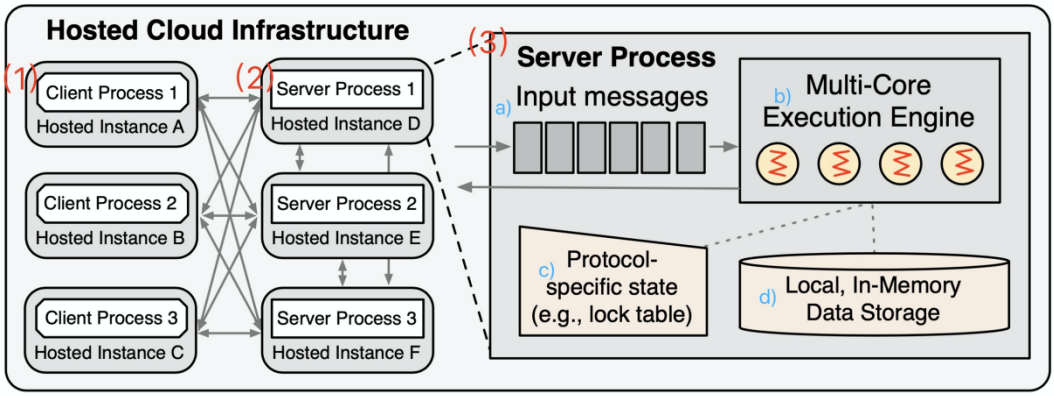


图 1　Deneva 系统架构图

在 3TS 中，对 Deneva 进行改进。改进后的代码位于 contrib/deneva 路径下，其内部项目结构如图 2 所示：

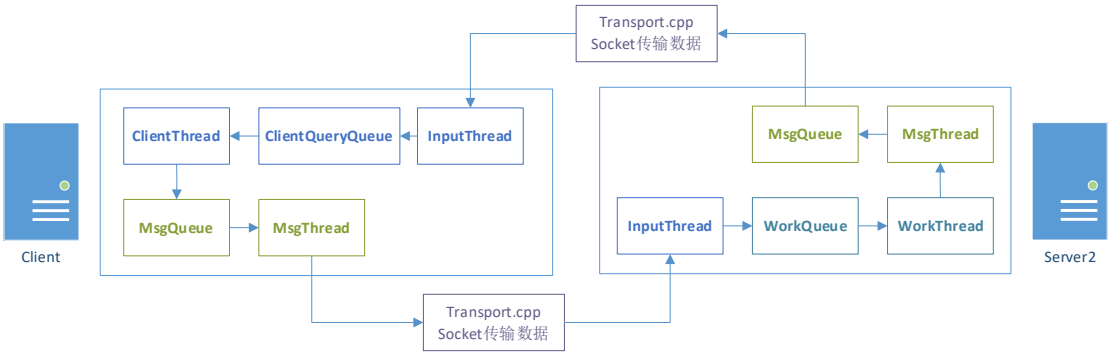


图 2 Deneva 实现框架图

- Deneva 整体上分为 Server 和 Client 两种节点。
- Client 用于生成并向 server 发送要执行的负载 query。Server 用于协调执行 Client 发来的负载 query
- Client 和 Server 共有模块：
  - MsgQueue：消息队列，存储了要发送的 msg。
  - MsgThread：消息发送线程，循环从 MsgQueue 中取出 msg 发送出去。
  - InputThread：消息接收线程，接收 Server/Client 发来的信息。
- Client 上专有模块
  - ClientQueryQueue：客户端的 query 队列，存储了测试开始前生成 query 列表。
  - ClientThread:客户端线程,循环从 ClientQueryQueue 取出 query,通过 MsgThread 和 MsgQueue 发送给 Server。
- Server 上专有模块
  - WorkQueue：服务端的待处理 msg 队列， InputThread 接收到 msg 后，会将其放入 WorkQueue。

- WorkThread：服务端的执行线程，从 WorkQueue 取 msg 出来执行处理，在执行完毕后，会生成返回 msg，同样通过 MsgThread 和 MsgQueue 发送出去。

3.2 事务执行流程

在 Deneva 中，如图 3 所示，一个事务的执行流程为：

（1） 客户端首先发起一个事务请求（由多个操作组成），并将事务放到 ClientQueryQueue 中。ClientThread 会从队列中取出事务请求存入消息队列 MsgQueue。之后，消息发送线程会从 MsgQueue 中取出某一事务的操作集合，封装为一个 Request，发送到某一服务端（通过第一个操作所访问的数据确定），作为这个事务的协调节点；

（2） Request 到达服务器后，服务器先对请求进行解析，并把这一事务的所有操作作为一个元素，放到工作队列（WorkQueue）中。工作队列中放置的有来自客户端的新事务和已经开始执行的事务的远程操作，后者在队列中比前者优先级更高。工作线程池中的线程轮询工作队列并处理事务中的操作。当某一工作线程处理当前事务的操作时，其首先对事务进行初始化，然后按顺序执行读写操作，并最终提交或者回滚；

a) 执行事务的过程中有两种情况会导致事务进入等待，一是等待某一资源上的排他锁释放；二是需要访问远程服务端中的数据。当远程访问其他服务端中的数据需要的等待时，远程服务端将返回 WAIT 指令给当前协调节点，协调节点会暂存当前事务的等待状态并调度当前工作线程执行其他事务的操作，从而避免了工作线程的阻塞。当某一等待事务可以继续执行时，基于工作队列的优先级调度策略，它将由第一个可用的工作线程继续执行；

b) 并发控制协议所要求的额外操作会嵌入在事务执行过程中，包括读写操作、验证操作、提交/回滚操作等。

（3） 当协调节点完成某一事务的操作后，他会将事务执行结果放入消息队列，然后由消息发送线程通知客户端当前事务的执行结果。

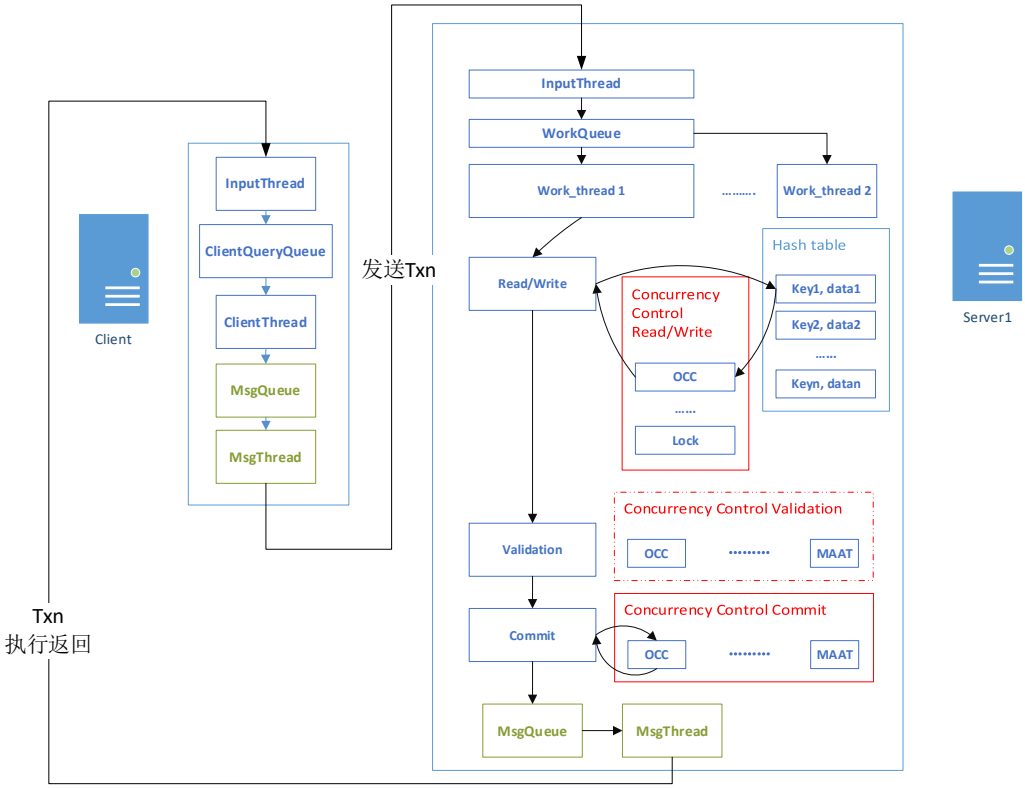


图 3　Deneva 事务执行流程图

4、分布式事务概述

文献[15]对分布式事务定义为：

> A distributed transaction is a database transaction in which two or more network hosts are involved. Usually, hosts provide transactional resources, while the transaction manager is responsible for creating and managing a global transaction that encompasses all operations against such resources. Distributed transactions, as any other transactions, must have all four ACID (atomicity, consistency, isolation, durability) properties, where atomicity guarantees all-or-nothing outcomes for the unit of work (operations bundle).

分布式事务以分布式系统为物理基础，实现了事务处理的语义要求，即也要在分布式系统上满足 ACID 特性。所以分布式数据库的分布式事务处理，同样要遵循单机数据库系统下的事务相关理论，确保每个事务符合 ACID 的要求，采用分布式的并发访问控制技术来处理分布式系统下的数据异常现象，实现分布式的事务 ACID 特性。

分布式事务处理机制的基本技术，以单机数据库系统中的事务处理技术为基础，但也有些不同，诸如如何处理分布式数据异常、如何做到分布式架构下的可串行化，如何做到跨节点的原子提交，如何做好存在网络分区或有较高延时下事务的响应等。

3TS 框架，是一个分布式环境，这些内容，都将在 3TS 中得到实现和验证。

5、3TS 提供的并发访问控制算法

3TS 中目前集成了十三种并发访问控制算法，主要包括：

（1）两阶段封锁协议（2PL：No-Wait，Wait-Die）
（2）时间戳排序协议（T/O）
（3）多版本并发控制协议（MVCC）
（4）乐观并发控制协议（OCC、FOCC、BOCC）
（5）优化的乐观并发控制协议（MaaT、Sundial、Silo）
（6）确定性并发控制协议（Calvin）
（7）基于快照隔离的并发控制协议（SSI、WSI）

下面依次对这些并发控制协议进行简要介绍：

5.1 两阶段封锁协议（2PL）

两阶段封锁协议（Two-phase Locking, 2PL）是目前最广泛应用的并发控制协议。2PL 依靠读写操作发生时获取共享锁或排他锁的方式，来对事务间的冲突操作进行同步。根据 Bernstein 和 Goodman 的描述[2]，2PL 对锁的获取有如下两条规则：1）同一个数据项上不能同时存在两个互相冲突的锁；2）一个事务在释放了任意一个锁之后，不能再获取锁。第二条规则规定了事务中的锁操作分为了两个阶段：生长阶段（growing phase）和衰减阶段（shrinking phase）。在生长阶段中，事务将为其需要访问的所有记录获取锁。读取操作获取共享锁，写入操作获取互斥锁。共享锁是不冲突的，互斥锁与共享锁或其他互斥锁冲突。一旦某一事务释放了其中任意一个锁，事务便进入 2PL 的第二阶段，称为衰减阶段。进入此阶段后，事务就不允许获取新的锁。

3TS 实现了直到事务提交或终止后才释放锁的严格 2PL（Strict 2PL）协议。根据避免死锁方法的不同，3TS 中实现的 2PL 包括两种：2PL（No-Wait）和 2PL（Wait-Die），其实现遵循了[2,3]中对这两种协议的描述：

5.1.1 2PL（No-Wait）

该协议规定，当事务尝试加锁时发现锁冲突，则回滚当前正在请求锁的事务。被回滚的事务已持有的锁都将被释放，从而允许其他事务获得锁。**No_Wait 机制可以避免死锁**，因为事务间的等待不会成环。但是，并非每次出现锁冲突都会导致死锁，因此回滚率可能较高。

5.1.2 2PL（Wait-Die）

Rosenkrantz[3]提出 2PL（Wait-Die），用事务开始时间戳作为优先级，来保证锁的等待关系与时间戳顺序一致。只要事务时间戳比当前拥有锁的任何事务小（旧），当前事务需要等待；否则，当前事务需要回滚。对于任意两个冲突事务 Ti 和 Tj，该协议使用时间戳优先级来决定是否让 Ti 等待 Tj，**如果 Ti 优先级更低（时间戳更小）那么 Ti 需要等待，否则 Ti 回滚。因而锁等待图里不会构成环，该方法可以避免死锁的发生**。该算法是 TO 和 Locking 技术的融合。

但是，利用 2PL 的原理实现的分布式事务处理机制，要么避免死锁（回滚率大），要么需要解决死锁的问题（资源死锁和通信死锁）。在分布式系统中解决死锁的代价会很大（单机系统上解决死锁的代价已经很大，基于多进程或多线程架构的现代数据库系统，解决死锁操作可能导致系统几乎停止服务。如 MySQL5.6、5.7 版本中对同一个数据项并发更新，死锁检测操作机会导致系统几乎停止服务）。

不仅是死锁检测会消耗巨大资源，锁机制本身带来的弊端一直为人诟病。文献[5]认为封锁机制的弊端如下（对这些弊端的清晰认识，促使文献[5]的作者设计了 OCC，Optimistic Concurrency Control，乐观并发访问控制）：

1. **封锁机制开销大：**为保证可串行性，加锁协议对于不改变数据库完整性约束的只读事务，需要加读锁互斥并发写操作，以防止别人修改；对于可能造成死锁的加锁协议还需要忍受死锁预防/死锁检测等机制带来的开销。
2. **封锁机制复杂：**为了避免死锁，需要定制各种复杂的加锁协议（如什么时候加锁、什么时候才能释放锁，怎么保证严格性等）。
3. **降低系统的并发吞吐量：**
   a) 在等待 I/O 操作的一个事务持锁，将大幅降低系统整体的并发吞吐量。
   b) 事务回滚完成前，加锁事务回滚时必须持有锁，直到事务回滚结束，这也将降低了系统整体的并发吞吐量。

另外，使用锁的机制进行互斥操作，对于操作系统而言，会引发耗时的内核态操作而使得锁机制的效率低下。这意味着基于操作系统的锁机制的带有事务处理语义的 2PL 技术更加不可采用（但也有一些技术在不断改进基于封锁协议的并发访问控制算法）。

5.2 时间戳排序协议（T/O）

时间戳排序协议（Timestamp Ordering, T/O）在事务开始时，为事务分配时间戳，并按照时间戳的顺序对事务进行排序[2]。当某一操作的执行会违背事务之间已经规定的顺序时，当前操作所在的事务会被回滚或进入等待状态。

3TS 中的 T/O 算法实现遵循[2]中第 4.1 节的描述，可详细参考该文献获得更多内容。如下图，给出了一个基本的 T/O 算法的实现。

Each transaction (T_i) is an ordered list of actions (A_{ix}). Before the transaction performs its first action (A_{i1}), it is marked with the current timestamp, or any other strictly totally ordered sequence: $TS(T_i) = NOW()$. Every transaction is also given an initially empty set of transactions upon which it depends, $DEP(T_i) = []$, and an initially empty set of old objects which it updated, $OLD(T_i) = []$.

Each object (O_j) in the database is given two timestamp fields which are not used other than for concurrency control: $RTS(O_j)$ is the time at which the value of object was last used by a transaction, $WTS(O_j)$ is the time at which the value of the object was last updated by a transaction.

For all T_i:

For each action A_{ix}:

If A_{ix} wishes to use the value of O_j:

If $WTS(O_j) > TS(T_i)$ then **abort** (a more recent thread has overwritten the value),

Otherwise update the set of dependencies $DEP(T_i).\text{add}(WTS(O_j))$ and set $RTS(O_j) = \max(RTS(O_j), TS(T_i))$;

If A_{ix} wishes to update the value of O_j:

If $RTS(O_j) > TS(T_i)$ then **abort** (a more recent thread is already relying on the old value),

If $WTS(O_j) > TS(T_i)$ then **skip** (the Thomas Write Rule),

Otherwise store the previous values, $OLD(T_i).\text{add}(O_j, WTS(O_j))$, set $WTS(O_j) = TS(T_i)$, and update the value of O_j.

While there is a transaction in $DEP(T_i)$ that has not ended: **wait**

If there is a transaction in $DEP(T_i)$ that aborted then **abort**

Otherwise: **commit**.

To **abort**:

For each $(\text{old}O_j, \text{old}WTS(O_j))$ in $OLD(T_i)$

If $WTS(O_j)$ equals $TS(T_i)$ then restore $O_j = \text{old}O_j$ and $WTS(O_j) = \text{old}WTS(O_j)$

图 4 T/O 算法

5.3 多版本并发控制协议（MVCC）

多版本并发访问控制（Multi-version Concurrency Control，MVCC）是目前的数据库管理系统普遍采用的并发访问控制技术，其首先在 1978 年由 David Patrick Reed [4]提出，其主要思路是将一个逻辑数据项扩充为多个物理版本，将事务对数据项的操作转换为对版本的操作，从而提高事务处理的并发度，并可以提供读写互不阻塞的能力。

Multi version concurrent access control，多版本并发访问控制技术，简称 MVCC 技术。

1970 年 MVCC 技术被提出，1978 年的[4]《Naming and synchronization m a decentralized computer system》进一步描述，之后在 1981 年文献[16]详细描述了 MVCC 技术，但是其描述的 MVCC 技术是基于时间戳的 MVCC。

Multiversion T/O

For rw synchronization the basic T/O scheduler can be improved using **multiversion data items** [REED78]. For each data item x there is a set of R-ts's and a set of (W-ts, value) pairs, called versions. The Rts's of x record the timestamps of all executed dm-read(x) operations, and the versions record the timestamps and values of all executed dm-write(x) operations. (In practice one cannot store R-ts's and versions forever; techniques for deleting old versions and timestamps are described in Sections 4.5 and 5.2.2.)

之后，MVCC 技术被大量使用并衍生出多种版本。

2008 年，文献[13]发表，提出“Serializable Snapshot Isolation”技术实现基于 MVCC 的可串行化隔离级别。这使得 PostgreSQL V9.1 使用该技术实现了可串行化隔离级别。

2012 年，文献[14]发表，提出“Write-snapshot Isolation”技术通过验证读写冲突实

现基于 MVCC 的可串行化隔离级别，相较于依靠检测写写冲突的方式提高并发度（某种写写冲突是可串行化的）。该文作者基于 HBase 做了系统实现。

2012 年，文献[17]在 PostgreSQL 实现了 SSI 技术。该文献不仅讲述了序列化快照的理论基础、PostgreSQL 对于 SSI 技术的实现方式，还提出了为支持只读事务而实现的“Safe Snapshots”、“Deferable Transaction”，因为避免读-写冲突造成事务回滚的影响而对被回滚的事务采取“safe retry”策略，以及两阶段提交对选取回滚读-写冲突的事务的影响等重要话题。

文献[19]较为系统地讨论了 MVCC 技术涉及地四个方面，分别是：并发访问控制协议、多版本存储、旧版本垃圾回收、索引管理，另外讨论了 MVCC 的多种变体的原理，实现多种变体（MV2PL、MVOCC、MVTO 等）后在 OLTP workload 上测试评估各个变体的效果。[18]则对于 MVCC 的旧版本垃圾回收进行了详细讨论。

3TS 中，主要基于[2]中第 4.3 节的描述，结合 T/O 算法实现了 MVCC。因此，事务间仍然使用开始时间戳进行排序，与传统 T/O 算法不同的是，MVCC 利用多版本的特性，可以减少 T/O 中的操作等待开销。MVCC 中的操作执行机制如下（用 ts 代表当前事务的时间戳）：

1. 读操作
   a) 如果 ts 大于 prereq 中所有事务的时间戳，且在 writehis（当前数据项的版本链）中存在一个版本，其 wts 大于 ts 且小于 pts，则当前版本可以返回，且将当前事务的时间戳存入 readhis。如果 writehis 不存在对应的时间戳，则将当前事务的时间戳存入 readreq。主要原因为：
      i. 如果在预写事务和读操作之间存在一个已经提交的写，那么代表当前读操作读到的数据是已经提交写事务写入的，满足时间戳排序，可以读到该版本；
      ii. 读操作的时间戳比当前未完成的写事务的时间戳大，应该读到新数据，所以要等待；
   b) 否则，当前读操作通过时间戳读到最新可见版本，并将当前事务的时间戳存入 readreq；
2. 写操作
   a) 如果 ts 小于 readhis 中所有事务的时间戳，且在 writehis 中存在一个时间戳在 rts 和 ts 之间，可以正常预写数据。如果 writehis 中不存在符合条件的版本，那么当前事务回滚；
   b) 将当前写操作暂存在 prereq_mvcc 中；
3. 提交操作
   a) 将当前事务时间戳和写入的新版本插入 writehis；
   b) 从 prereq 中把当前事务的写操作删除；
   c) 继续执行 readreq 中满足时间戳顺序的读事务；

更多并发访问控制算法，待续...

致谢

特别感谢腾讯 CynosDB（TDSQL）团队与中国人民大学数据工程与知识工程教育部重点实验室对本项工作的支持和帮助，感谢赵展浩，刘畅，赵泓尧等同学为本文做出贡献。

Reference

[1] Rachael Harding, Dana Van Aken, Andrew Pavlo, Michael Stonebraker: An Evaluation of Distributed Concurrency Control. Proc. VLDB Endow. 10(5): 553-564 (2017)
[2] Philip A. Bernstein, Nathan Goodman: Concurrency Control in Distributed Database Systems. ACM Comput. Surv. 13(2): 185-221 (1981)
[3] Daniel J. Rosenkrantz, Richard Edwin Stearns, Philip M. Lewis II: System Level Concurrency Control for Distributed Database Systems. ACM Trans. Database Syst. 3(2): 178-198 (1978)
[4] D. P. Reed. Naming and synchronization in a decentralized computer system. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.
[5] H. T. Kung, John T. Robinson: On Optimistic Methods for Concurrency Control. ACM Trans. Database Syst. 6(2): 213-226 (1981)
[6] Theo Härder: Observations on optimistic concurrency control schemes. Inf. Syst. 9(2): 111-120 (1984)
[7] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi: MaaT: Effective and scalable coordination of distributed transactions in the cloud. Proc. VLDB Endow. 7(5): 329-340 (2014)
[8] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, Srinivas Devadas:
Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. Proc. VLDB Endow. 11(10): 1289-1302 (2018)
[9] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, Samuel Madden: Speedy transactions in multicore in-memory databases. SOSP 2013: 18-32
[10] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, Daniel J. Abadi: Calvin: fast distributed transactions for partitioned database systems. SIGMOD Conference 2012: 1-12
[11] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10
[12] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, Dennis E. Shasha: Making snapshot isolation serializable. ACM Trans. Database Syst. 30(2): 492-528 (2005)
[13] Michael J. Cahill, Uwe Röhm, Alan D. Fekete: Serializable isolation for snapshot databases. SIGMOD Conference 2008: 729-738
[14] Maysam Yabandeh, Daniel Gómez Ferro: A critique of snapshot isolation. EuroSys 2012: 155-168
[15] https://en.wikipedia.org/wiki/Distributed_transaction
[16] P. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
[17] D. R. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," PVLDB, vol. 5, no. 12, pp. 1850-1861, 2012.
[18] J. Böttcher, et al., Scalable Garbage Collection for In-Memory MVCC Systems, in *VLDB*, 2019

[19] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, Andrew Pavlo:An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. Proc. VLDB Endow. 10(7): 781-792 (2017)

[20] D. Lomet and M. F. Mokbel, “Locking key ranges with unbundled transaction services,” VLDB, pp. 265–276, 2009.

[21] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, Aoying Zhou: Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. PVLDB 12(5): 584-596 (2019)

[22] X. Yu, A. avlo, D. Sanchez, and S. Devadas, “Tictoc: Time traveling optimistic concurrency control,” in Proceedings of SIGMOD, vol. 8, 2016, pp. 209–220.

[23] Rudolf Bayer, Klaus Elhardt, Johannes Heigert, Angelika Reiser:Dynamic Timestamp Allocation for Transactions in Database Systems. DDB 1982: 9-20