OceanBase 的树形两阶段提交框架：优化可扩展性与一致性

Quanqing Xu* Chen Qian* Chuanhui Yang† Fanyu Kong†OceanBase，蚂蚁集团 OceanBase，蚂蚁集团 OceanBase，蚂蚁集团 OceanBase，蚂蚁集团

刘桂香 OceanBase，蚂蚁集团

Fusheng Han OceanBase，蚂蚁集团

翟子翔 OceanBase，蚂蚁集团

arXiv:2603.00866v1 [cs.DB] 1 Mar 2026

摘要

现代分布式数据库面临着在分布式分区间实现事务一致性的挑战。传统两阶段提交（两阶段提交）协议会带来较高的协调开销和延迟，并且在动态分区迁移时需要复杂的恢复机制。本文为 OceanBase 提出了一种新颖的树形两阶段提交框架，通过三项创新利用单机日志流应对这些挑战。首先，我们提出将日志流作为原子参与者，取代分区级协调。将每个日志流视为提交单元后，跨越N个共置分区的事务只需与一个参与者交互，从而将协调开销降低数个数量级（例如，当$N = 100$时降低99%）。其次，我们设计了一种采用以协调者为根的有向无环图拓扑的树形两阶段提交协议，通过递归构建提交树动态处理分区迁移。事务执行期间若分区发生迁移，该协议会将迁移上下文嵌入为叶节点，无需显式更新参与者列表，同时消除循环依赖，并确保在拓扑变化期间实现线性化提交。第三，我们引入准备未知和事务未知状态，以防止参与者丢失上下文时出现一致性违规。这些状态会在重试期间表明存在不确定性，避免“说谎”的参与者导致错误中止，同时避免用户受到歧义的影响。实验评估表明，该框架的性能接近单机事务，同时降低了延迟和带宽消耗，验证了其对现代分布式数据库的有效性。

PVLDB 引用格式：
Quanqing Xu、Chen Qian、Chuanhui Yang、Fanyu Kong、刘桂香、Fusheng Han 和翟子翔。《OceanBase 的树形两阶段提交框架：优化可扩展性与一致性》。PVLDB，20(1)：XXX-XXX，2026。doi:XX.XX/XXX.XX

PVLDB 工件可用性：
源代码、数据和/或其他构件已发布于 https://github.com/oceanbase/oceanbase/tree/develop/src/storage/tx。

*Quanqing Xu 和 Chen Qian 对本研究贡献相同。†Chuanhui Yang 和 Fanyu Kong 为通讯作者。本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议授权。访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看本许可协议副本。对于超出本许可协议范围的任何使用，请发送邮件至 info@vldb.org 获取许可。版权归所有者/作者所有。出版权授权给 VLDB基金会。
Proceedings of the VLDB Endowment，第 20 卷，第 1 期，ISSN 2150-8097。
doi:XX.XX/XXX.XX

1 引言

在大规模分布式数据库时代，确保跨地域分布的数据分区之间的事务一致性仍然是一项根本性挑战。随着现代应用越来越依赖分布式数据库系统来处理海量工作负载，对高效事务协调机制的需求变得至关重要。传统两阶段提交协议 [2, 7, 13, 15] 一直是维护分布式环境中 ACID 属性的基石，为协调多个参与者之间的事务提交提供了可靠的框架。然而，随着数据库系统扩展到支持数千个分区和数百万个并发事务，这些基础协议在满足现代云原生应用的性能和可扩展性要求方面面临着重大挑战。

传统两阶段提交协议面临两个关键局限。首先，可扩展性瓶颈 源于以分区为中心的设计，这种设计将每个分区视为独立的参与者：即使 100 个分区都位于同一台机器上，跨越这 100 个分区的事务仍需经历 100 次准备/提交阶段，导致协调消息和日志同步量呈二次增长，并严重限制高并发下的吞吐量。其次，分区迁移过程中会出现复杂的动态拓扑 问题，而分区迁移对于弹性云中的负载均衡至关重要。现有方法要么需要在迁移后耗费大量成本重新验证参与者列表 [8]，要么会在分区与日志之间引入循环依赖 [1, 12],，当活动事务期间发生分区迁移时，这会损害一致性。

近年来，分布式事务管理取得了长足进展，但根本性局限依然存在。Spanner [6] 和 CockroachDB [16] 等系统仍依赖按分区进行协调，继承了传统方法的可扩展性瓶颈。尽管日志结构化系统展现了以日志为中心的架构的潜力，但它们并未将日志流作为原子单元用于事务协调，因此可扩展性和动态拓扑方面的挑战仍未解决。

在 OceanBase 的单机日志流架构中，同一单元内共置的分区领导者位于同一台机器上，因此这些分区都可以写入同一个共享日志流。这种架构设计为优化两阶段提交提供了独特的机会：通过将日志流而非单个分区用作参与者，我们可以大幅降低协调开销，同时利用日志流固有的结构自然地处理分区迁移。本文提出了一种新颖的基于日志流的两阶段提交框架，通过三项关键创新应对可扩展性和动态拓扑方面的挑战。我们的方法从根本上将原子单元从分区转变为日志流，并引入

一种能够适应分区迁移的动态树形协议，并提供了一种有原则的方法来处理上下文丢失场景。

因此，本文的贡献如下：

- **将日志流作为参与者**：我们将日志流抽象为新的原子单元，而非单个分区，从而将参与者数量降低了数个数量级。现在，涉及 100 个共置分区的事务只需与一个日志流参与者交互，协调开销减少 99%，并实现了单机事务性能。
- **动态树结构两阶段提交**：不同于采用静态结构的传统层次化两阶段提交 [18]，我们提出一种动态树结构协议，其中参与者列表构成一个有向无环图，并随着分区迁移而演变。该结构递归跟踪日志流依赖，自动即时构建树，无需显式更新参与者列表，从而解决“循环事务”问题 [1]。
- **保障一致性的未知状态机制**：当参与者丢失事务上下文时，可能会对先前的决策“撒谎”。我们引入准备未知和事务未知状态：参与者发出不确定信号，重建后的协调者向用户返回事务未知，从而避免强制提交，同时使用户免受模糊状态的影响（详见第 5 节）。

我们的贡献推动了理论理解与实际实现的发展：日志流抽象挑战了分区必须作为原子单元的假设；动态树结构协议能够处理执行期间不断变化的拓扑结构；未知状态机制则确保在上下文丢失时保持一致性。该框架实现了两个设计目标：**(1) 静态事务的高性能**——通过日志流聚合，实现单机事务延迟（1.2ms）；**(2)动态事务的简洁性**——通过自动树构建，以最低复杂度处理分区迁移。在 OceanBase 上进行的严格评估表明，在高负载分区迁移工作负载下，提交吞吐量呈线性扩展，使我们的方案成为云环境中下一代分布式数据库的一项重要进展。

本文结构安排如下。§2介绍 OceanBase 数据库、ACID 属性以及分区迁移期间的日志流拓扑结构背景。§3介绍树形两阶段提交协议，包括事务延迟优化和状态机设计。§4讨论转移过程，以及该过程如何处理事务执行期间的并发分区迁移。§5介绍用于保持一致性的未知状态机制。§6讨论优化和 TLA+ 证明；§7介绍实验评估，以展示该框架的高性能和可扩展性。§8回顾相关工作，§9总结全文。

2 背景

2.1 OceanBase 数据库

OceanBase 是一个高性能分布式关系型数据库系统，旨在满足可扩展性、跨区域容错以及成本效益等现代互联网规模需求应用 [23]。OceanBase 采用无共享架构，强调卓越的可扩展性与高可用性。最新版本 OceanBase 4.0 引入了 Paetica [22],，这是一种创新架构，合并了无共享范式与共享一切范式，从而能够在单机模式与分布式模式之间无缝切换。

OceanBase 的一项关键架构特性是其单机日志流设计：同一单元内共置的分区领导者位于同一台机器上，使这些分区都能写入同一个共享日志流。这种设计为我们的基于日志流的两阶段提交优化奠定了基础，因为它能自然地将多个分区聚合为单个日志流参与者，在保持事务一致性的同时降低协调开销。

2.2 ACID 属性与传统两阶段提交

传统两阶段提交是确保分布式事务原子性的基础协议 [13]。该协议包含两个阶段：(1) 准备阶段，协调器向所有参与者发送准备请求，每个参与者投赞成票（通过写入准备日志）或反对票；(2) 提交/中止阶段，协调器收集所有投票，并根据投票结果向所有参与者发送提交或中止消息。如果所有参与者都投赞成票，事务就会提交；否则，事务就会中止。该协议确保所有参与者要么一起提交，要么一起中止，从而保持事务原子性。

在单机日志流方案中，分区仍是事务读写操作的基本单元，并发控制仍在分区级别进行。为确保 ACID 特性 [19], 原子性的实现要求所有参与并发控制的单元都通过隔离性和持久性检查。因此，如果我们在两阶段提交协议中使用日志流作为参与者，参与者的最小集合必须包含并发控制涉及的分区最终所在的所有日志流。我们将此称为**最小集合要求**。任何包含这一最小集合的参与者列表也都是正确的。关键挑战在于如何在两阶段提交阶段获取所有此类日志流，并完成整个提交过程。

2.3 日志流拓扑

分区迁移是 OceanBase 实现负载均衡和资源优化的重要机制。然而，它也增加了事务协调的复杂性：当分区在事务执行期间迁移时，必须更新参与者列表，将曾承载该事务分区的所有日志流纳入其中。

在一种朴素方法中，协调器仅根据事务开始时执行过读写操作的日志流来构建参与者列表。然而，如果分区迁移发生在事务提交之前，该列表就会违反最小集合要求（第 2.2 节）。如图 1 所示，事务最初涉及日志流 A 和 B，但分区迁移后，日志流 C 也会创建事务上下文，因此必须参与并发控制。朴素方法遗漏了日志流 C，导致参与者列表不正确。

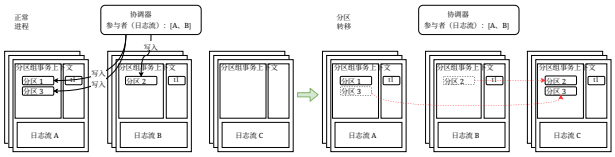


图 1：事务期间参与者列表的变化。

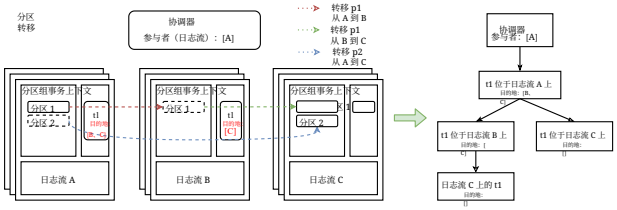


图2：执行过程中的日志流树。

为正确识别所有相关日志流，我们注意到事务上下文会在两种情况下创建：(1) 分区执行读写操作；(2) 分区被转移。读写操作涉及的日志流可以直接获取，而转移涉及的日志流则可以通过转移记录递归获取：当分区从日志流L_s转移到L_d时，源日志流会在其事务上下文中记录目标日志流。递归追踪这些转移记录，即可构建一棵日志流树，其中包含曾托管该事务分区的所有日志流。示例2.1说明了这一构建过程，定理2.2则证明了该树满足最小集合要求。

示例2.1. 如图2左侧所示，事务t1在执行过程中涉及分区1和分区2，二者均位于日志流（LS）A上。随后，分区1先从日志流 A 转移到日志流 B，再从日志流 B 转移到日志流 C；分区2从日志流 A 转移到日志流 C。转移成功使日志流 A 上的事务t1能够记录该转移所产生的目标日志流 B 和日志流 C；同样，日志流 B 上的事务t1记录该转移所产生的目标日志流 C。右侧所示为日志流树，其中协调器为头节点，参与并发控制的分区最终所在的日志流作为叶节点。

定理2.2. 日志流树满足最小集合要求。

证明见附录A.1。

3 树形两阶段提交

3.1 树形结构设计

根据定理2.2，日志流树满足参与者列表的最小集合要求。树形两阶段提交（又称层次化两阶段提交） [18], 将协调器和参与者组织成一棵树，自然契合我们的设计。每个拥有子节点的节点既充当协调器（负责其子节点），也充当参与者（负责其父节点）；其子树中的节点构成其参与者列表。没有子节点的节点仅充当参与者。

然而，传统层次化两阶段提交在事务响应时间方面仍有改进空间。为此，本节介绍 OceanBase 对传统两阶段提交协议所做的优化，并将这些优化与树形结构相结合，从而在保证正确性的同时实现高性能。

3.2 事务延迟优化

传统两阶段提交包含两个阶段：准备和提交/中止。协调器和参与者需要同步写入准备[1] 日志和提交/中止日志。协调器在其提交日志持久化后回复客户端，因此会产生两条 RPC 消息和三次日志同步，这构成了事务延迟。由于两条 RPC 消息不可或缺，我们的优化重点是减少 I/O 操作。

OceanBase 通过消除协调器日志记录来优化两阶段提交：协调器从不写入日志。然而，这会带来两个挑战：（a）在异常情况下，由于没有自己的准备日志，协调器无法确定所有参与者的状态，因此每个参与者都必须在其准备日志中记录所有参与者；（b）由于缺少提交/中止日志，参与者无法通过协调器确认最终事务状态，因此需要一种机制，确保参与者在释放资源前能够独立确定事务状态。

最初，OB 两阶段提交通过增加清理阶段来应对这些挑战：所有参与者在提交/中止后、进入 TOMBSTONE 状态前，同步写入清理日志。协调器收到准备成功/失败消息后便向客户端响应，从而将事务延迟降至两条 RPC消息和一次 I/O操作（参与者的准备日志）。然而，清理日志会增加资源消耗。在后续版本中，清理阶段被移除：协调器在回复客户端后同步写入提交/中止日志，在确保正确性的同时，使事务响应期间保持一次 I/O操作。

3.3 状态机

根据第3节2中的优化，我们介绍 OceanBase 中的树形两阶段提交协议。参与者列表由事务执行日志流或分区迁移的目标日志流构成。迁移目的地记录在源事务上下文的参与者列表中，具体实现详见第4节。为简化设计，我们统一抽象所有树节点，不区分叶子节点和非叶子节点。每个节点维护以下变量：

- 状态：节点自身的状态（同意或反对），初始为同意。
- 状态：两阶段提交状态（运行中、准备、提交、中止、墓碑），初始为运行中。
- 参与者列表：当前状态下子参与者的列表，初始为空。
- 已收集的参与者列表：消息已收集的参与者列表，初始为空。

状态机如图3所示。我们将协议分为两个阶段进行描述：

[1]在某些两阶段提交协议实现中，可以省略协调器的准备日志。

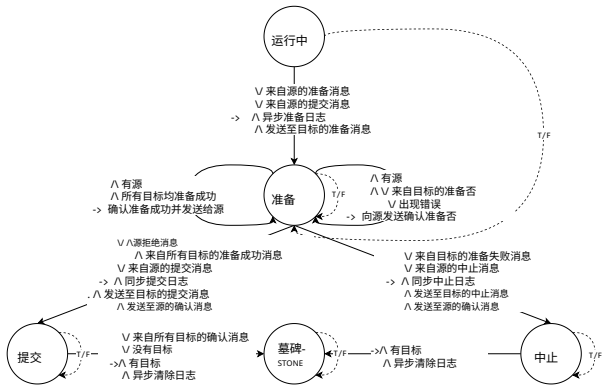


图3：树形两阶段提交状态机。实线表示正常进程；虚线表示异常超时（T）和停机时间（F）。∧ 表示与，∨ 表示或。A → B 表示在当前状态下，输入 A 生成输出 B，并转换到下一状态。

准备阶段。 初始时，节点处于运行状态。收到父节点发送的准备消息或调度器发出的提交请求后，节点会异步写入准备日志，向所有子节点发送准备消息，并重置已收集的参与者列表。准备日志持久化后，节点进入准备状态。收到子节点发送的准备成功/失败消息后，节点会将其添加到已收集的参与者列表 中，并在任一消息为准备否时将状态 设为反对。

提交阶段。 当节点从所有子节点收到 PREPARE OK（即，已收集的参与者列表 等于子节点集合）时，如果存在父节点，则向其发送 PREPARE OK；如果该节点是根节点，则向所有子节点发送 COMMIT 消息。处于准备状态的节点从其父节点收到 COMMIT 消息后，会同步写入提交日志，向所有子节点发送 COMMIT 消息，重置已收集的参与者列表，并向其父节点发送确认，进入提交状态。最后，在从所有子节点收到确认后，异步写入清理日志并进入 TOMBSTONE状态。

中止阶段遵循类似的逻辑。详细的状态机规约见附录 B。

3.4 总结

我们总结树形两阶段提交算法在正常执行下的性能特征（假设提交成功，N 表示每个节点的子参与者平均数，H 表示平均树高）。以下指标针对给定的树；本节不分析扩展行为（即延迟和资源消耗如何随集群规模变化，或在分区迁移期间如何变化），而是在第 7 节通过实证评估。

事务延迟。 从用户发送提交请求到收到响应所需的时间。它需要 $2H$ 次消息往返（协调器/子协调者发送准备请求，参与者/子参与者发送准备响应）和 1 次日志同步（参与者的准备日志）。

行锁释放延迟。 从获取锁到释放锁所需的时间，代表冲突范围。它需要 $3H$ 次消息往返（协调器/子协调者发送准备、提交请求，参与者发送准备响应）和 2 次日志同步（参与者的准备日志和 COMMIT 日志）。

事务资源消耗。 事务消耗的资源总量，以消息和日志计：$5NH$ 次消息传输（协调者发送的 PREPARE、COMMIT 和 RELEASE；参与者返回的 PREPARE 和 COMMIT 响应；关于 RELEASE，见第 6.1.1 节），以及 $2NH$ 条同步复制日志（参与者的 PREPARE 和 COMMIT 日志）。

4 转移过程

4.1 转移原则

在 OceanBase 4.0 中，平板 [5] 和分区是分区单元：平板是最小的物理存储单元，分区则是用户定义的逻辑单元。一个分区可以包含一个或多个平板。最初，分区对称地分布在各台机器上。随着数据发生变化，OceanBase 会使用分区转移 [2] 在服务器之间迁移分区，以实现负载均衡。

第3节介绍了静态树形两阶段提交协议，其中日志流树保持不变。然而，分区转移可能在两阶段提交执行期间的任意时刻发生，从而动态改变日志流树（参与者）。为确保 ACID 属性，我们提出以下转移原则。该原则以日志和消息的形式表述；这两种表述是等价的，因为两阶段提交消息由已记录状态生成，并反映该状态（例如，准备/提交消息对应于持久化在准备/提交日志中的状态）。

定理 4.1（转移原则）。（日志。）转出日志之前写入的事务日志，必须通过转移过程应用到目标日志流。转出日志之后写入的事务日志，必须体现转移的影响（例如，将目标加入参与者列表）。（消息。）转移完成前发送的两阶段提交消息（准备、提交、中止），必须通过转移过程应用到目标日志流。转移完成后发送的消息，必须体现转移的影响。日志表述和消息表述是等价的。

证明见附录 A.2。

4.2 动态分区与转移

为处理两阶段提交执行期间的分区迁移，我们首先考虑一个理想化模型，其中分区迁移和两阶段提交日志同步是原子的。为确保每个阶段的参与者列表完整（依据定理 4.1），我们在当前阶段保留前一阶段的参与者，并在下一阶段保留当前阶段的参与者。事务上下文维护三个参与者变量：

- 参与者：来自 SQL 层的参与者，同时持久化在日志流和事务上下文中。
- 增量部分：来自前一阶段分区迁移的参与者，纳入当前阶段的参与者列表，并同时持久化在日志流和事务上下文中。

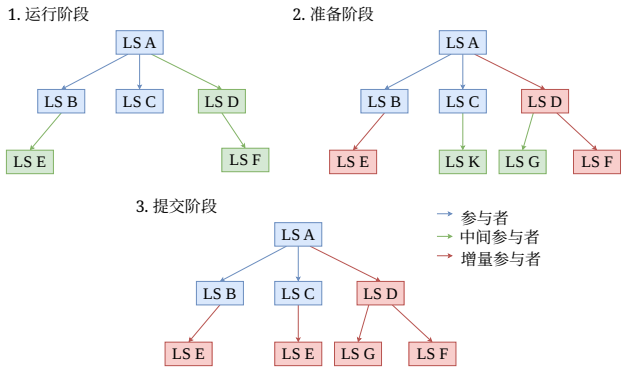


图 4：树形两阶段提交期间的转移。

- 中间部分：来自当前阶段分区迁移的参与者，纳入下一阶段的参与者列表，仅持久化在事务上下文中。

算法 1 展示了在运行中、准备和提交阶段发生转移时的提交过程（假设提交成功）。对于每次转移，$trans_i$、$trans_i.src$和$trans_i.dst$分别表示源端和目标日志流。migrateData 过程将事务数据和状态从源端发送到目标端。commitLogs 过程将中间部分合并到增量部分，并通过 Paxos 同步包含参与者和增量部分的两阶段提交日志。示例 4.2 展示了这一过程。

示例 *4.2*。 如图 4（左上）所示，初始时事务处于运行状态，参与者为参与者A、B 和 C（蓝色）。在运行期间，发生了三次分区迁移：A→D、B→E 和 D→F。因此，D 位于 A 的中间部分中，E 位于 B 的中间部分中，F 位于 D 的中间部分中（绿色）。当事务进入准备阶段（右上）时，中间部分变为增量部分，并在当前阶段加入参与者。准备阶段期间的两次转移将 K 和 G（绿色）添加为中间部分。在提交阶段（底部），日志流树中的所有节点都参与提交。

4.3 转移与日志提交并发

第 4.2 节想化的，在假设是理想化的，在实践中，无法使分区迁移与两阶段提交日志提交保持原子性。例如，当事务收到准备请求并将中间部分合并到增量部分时，并发分区迁移可能会向中间部分 添加一个目标日志流，并在准备日志提交前提交转出日志。这会导致准备日志遗漏迁移目的地，从而违反定理 4.1。

为确保正确性，必须满足以下两项要求：

- 分区迁移必须将转出日志写入前写入的所有事务上下文迁移到目标端。
- 事务必须在增量部分中包含此前所有传输目标日志流。

第一项要求由转移机制处理。对于第二项，以下操作必须是原子的（共享传输锁可避免竞争：

- 分区迁移会将目标日志流广播给所有事务，并提交转出日志。

算法 1并发分区迁移时提交

要求： 事务 T、调度器 s、日志流树 $tree$

/* T 进入运行状态 */
1: s 查找初始平板及相应的日志流 $log_{initial}$。
2: 树 ←日志$_{initial}$
/* 转移 转移$_1$ 在运行中发生 */
3: 转移$_1$.源.中间部分.add(转移$_1$.目标)
4: MIGRATEDATA(转移$_1$.源,转移$_1$.目标)
/* T 进入准备状态 */
5: 提交日志($tree$)
/* 转移 转移$_2$ 在准备状态期间发生 */
6: $trans_2.src$.中间部分.add($trans_2$.目标)
7: MIGRATEDATA($trans_2$.源,$trans_2$.目标)
/* T 进入提交状态 */
8: 提交日志($tree$)
/* 在提交期间发生转移$_3$ */
9: MIGRATEDATA($trans_3$.源,$trans_3$.目标)

10: **过程** MIGRATEDATA(源，目标) 11: 迁移 源 上下文中的数据与事务状态至目标。12: **结束过程**

13: **过程**提交日志(树)14: **对于**所有节点 p ∈ 树 **执行**15: p.增量部分.添加(p.中间部分) 16: PAXOS(p.参与者，p.增量部分) ▷ 通过 Paxos 同步日志 17: **结束循环** 18: **过程结束**

- 参与者将中间部分 合并到增量部分，并提交两阶段提交日志。

算法 2 和 3 展示了详细的分区迁移和日志提交过程。

定理 4.3。 *算法 2 和 3 确保满足上述两个要求，从而满足定理 4.1。*

证明见附录 A.3。

4.4 转移的异常处理

转移必须在故障场景中正确处理参与者列表，以确保正确性。迁移成功但源日志流发生故障时，必须维护正确的参与者列表。转移本身通过事务实现；因此，在异常场景中，所有传输操作都会回滚，从而保持一致性。在提交时，通过对相关状态进行原子替换，以原子方式更新事务状态和分区状态，确保要么完整的迁移结果可见，要么完全不可见。由于转移会将正在执行的事务数据同步到目的地日志流（图5），我们通过两种机制维护参与者列表：

基于迁移的迁移。 事务通过事务日志或持久化事务表进行迁移。在持久化期间，我们更新事务参与者列表，类似于主节点切换。

算法 2分区传输过程

要求： 传输服务 s，转出日志 日志 （包含平板 标签页，目标日志流 目标，时间戳 ts，源日志流源），传输锁 锁 ，事务上下文集 tc

1：s 获取锁
2：**对于**事务$t \in tc$ **执行**
3: **如果**标签页参与了t **则**
4: t 获取锁$_t$
5: 阻止t 写入新日志
6: 将t的事务上下文收集到迁移集中（针对目标日志）
7: t 释放锁$_t$
8: **结束 if**
9: **结束 for**
10: s 在源日志流上提交日志：平板标签页，目标目标，时间戳 ts
11: s 在目标日志流上提交日志：平板 tab，源端 src，事务信息集
12: **对于**事务$t \in tc$ **执行**
13: **如果** 平板 参与了t **与** $ts > t_。ts'$ **那么**
14: t 获取锁$_t$
15: 添加目标到$t_.$中间部分
16: 解除t 写入日志的阻塞
17: t 释放锁$_t$
18: **结束 if**
19: **结束 for**
20: s 释放锁
21：清除s上的信息

算法 3日志提交过程

要求： 传输服务s、传输锁、事务 context集tc

1：s 获取锁
2：**对于**事务$t \in tc$ **执行**
3: **如果**t 未被阻止写入日志，**则**
4: t 获取锁$_t$
5: 合并$t_.$中间部分至$t_.$增量部分 –
6: 提交日志包含$t_。$参与者与$t_。$增量部分 – 通过 Paxos
7: t 释放锁$_t$
8: **结束 if**
9: **结束 for**
10: s 释放锁

双写转移。 我们应用转移日志来更新参与者列表。当事务在日志流上创建上下文时，它会加入参与者列表。通过确保双写完成（在写入转出日志之前，将所有事务日志持久化到相应分区中），我们可以为故障恢复期间仍保留在内存中的事务写入转出日志。

4.5 事务对分区的依赖

基于树的两阶段提交依靠转移来维护正确的参与者列表（包含最小集合的日志流树）。转移过程必须更新源端包含已

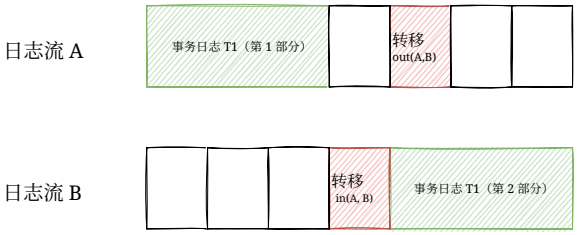


图5：执行期间的日志流树。

转移分区的事务的参与者列表，因此需要事务记录所包含的分区。为尽量减少分区依赖，我们为记录的分区数设置了上限。超过该上限时，转移会将目标添加到事务的参与者列表中。由于任何包含最小集合的参与者列表都是正确的（定理2.2），额外的参与者不会影响正确性。

4.6 循环事务

事务可能在不同日志流之间转移，并回到原始日志流，因此需要处理循环事务 [1, 12]。当参与者从不同发送方收到多条 PREPARE 消息时，不会立即发起处理两阶段提交准备请求。相反，它会等待自身的日志同步完成（所有节点按照状态机写入日志），但无需等待下游日志。它会直接向父节点回复 PREPARE OK（下游日志由父节点所在的同一日志流管理，以缓解线性提交进程中的循环流水线冲突）。修改后的状态机如下：

- 处理两阶段提交准备请求(p)
  - **启用：** 状态为运行中，且收到来自父参与者 p 的两阶段提交准备请求，或收到来自调度器的事务提交请求。
  - **操作：** 如果这是第一条父消息，则异步写入准备日志，向所有参与者发送两阶段提交准备请求（在日志和消息中包含参与者列表及节点状态），并转入准备状态。否则，等待准备日志同步完成，向父节点p 发送准备响应（包含参与者列表及节点状态），并转入准备状态。

活性。 修改后的规则在环形拓扑下仍能保持活性并避免死锁。每个节点只需等待自己的准备日志完成同步，便向其父节点回复 PREPARE OK；它不会等待任何下游节点或兄弟节点。因此，没有节点会等待另一个（直接或通过树形结构）也在等待自己的节点，所以不会发生循环等待。在通常的公平性假设下（日志同步会在有限时间内完成，且消息最终都会送达），每个已收到准备请求的参与者最终都会完成本地准备日志同步，并向其父节点发送 PREPARE OK。因此，进展会从叶节点向根节点推进，根节点可以收集所有投票，并在有限步数内推动事务提交或中止。因此，该协议最终会达到终止状态（全部已提交或全部已中止），从而保持树形两阶段提交的活性保证（参见第6.2.2节）。

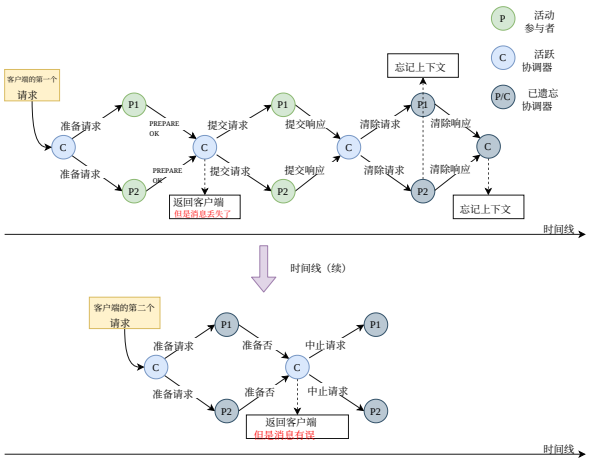


图6：树形两阶段提交中参与者说谎引发的问题。

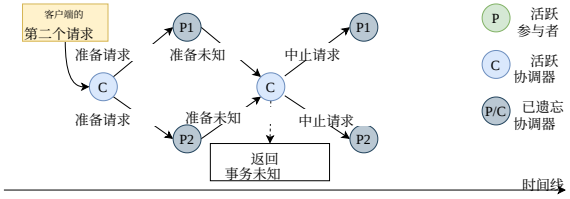


图7：通过“未知”状态解决树形两阶段提交中的参与者说谎问题。

4.7 实现设计目标

本节将说明我们如何实现第1节提出的两个设计目标：无转移事务的高性能，以及有转移事务的简洁性。

高性能。当事务在不包含内嵌参与者列表的日志流上提交时，两阶段提交无需分区感知即可执行。对于高度为 1 的日志流树，在参与者数量相同的情况下，其延迟和资源消耗与优化的两阶段提交相当；与使用分区作为原子单元相比，使用轻量级日志流作为原子单元可带来性能提升。当仅有一个日志流且没有参与者列表时，事务可简化为一阶段提交，从而达到单机性能。

简洁性。基于树的两阶段提交是一种经典的协议变体，只需进行极少的代码改动，即可让中间节点同时充当协调器（下游）和参与者（上游）。在故障恢复过程中，阶段内参与者列表不可变，可确保提交期间的转移处理简单明了；关键逻辑集中于转移管理、事务日志排序和处理，从而保证简洁性与健壮性。

5 “未知”事务状态

本节首先介绍参与者说谎问题，然后提出用于解决该问题的未知状态机制，最后讨论如何消除对未知状态的需求。

5.1 说谎的参与者问题

在第3节介绍的树形两阶段提交协议中，当事务上下文丢失时，树中的参与者（日志流）可能会向其父协调者“说谎”。任何分层两阶段提交结构都可能出现这个问题，包括我们的树形协议。示例5.1详细说明了这种情况。

示例*5.1*。如图6（上）所示：考虑一个树形两阶段提交结构，其中根协调者 C 和两个参与者日志流 P1、P2。协调者 C 收到提交请求后，树形两阶段提交协议正确地将所有参与者（P1、P2）转换到提交状态，事务日志随即被回收（请注意，此处的事务上下文仅指日志的索引）。然而，最终返回给用户的提交响应丢失了。在这种情况下，用户通过重新发送提交请求来重试事务，以重新建立协调者 C 的上下文。树形两阶段提交流程重新启动（如图6下方所示）。由于参与者的上下文已被回收，它们只能返回准备否。这迫使树形两阶段提交协议进入回滚阶段，最终向用户返回中止响应。显然，参与者在对第二个准备请求的响应中撒了谎，导致用户进入错误状态，并违反了一致性（我们将协调者和用户视为特殊类型的参与者）。

5.2 准备未知和事务未知

为解决上述问题，我们在树形两阶段提交状态机中引入了准备未知。当树中的参与者（日志流）收到父协调者发来的准备请求，但不存在事务上下文时（可能是由于正常提交清理或异常回滚），它会向父协调者回复准备未知。这表示该参与者已不再记得自己之前的决定（例如，在前一阶段投票赞成还是反对）。协调器收到准备未知后，会向其子树中的所有参与者发送中止请求，从而终止该事务。

对于用户响应，处理逻辑会更为复杂：如果根协调者上下文是新创建的，并遇到准备未知，系统会返回中止；如果根协调者是重新创建的（例如，在响应丢失后），并检测到准备未知，则会返回事务未知。回顾图6中的场景（如图7所示），当用户重试事务时，日志流树中的参与者因丢失了自身上下文，会响应准备未知，促使根协调者返回事务未知。这避免了树形两阶段提交过程中的不一致，并确保一致性。值得注意的是，第二个协调者被视为与原始协调者不同——它不会继续执行提交，因此不会违反一致性。然而，用户绝不能观察到不一致，因此会使用未知状态来处理此类模糊场景。

5.3 剥离“未知”状态

尽管准备未知和事务未知解决了协调者或参与者向其调用方谎报的问题，但它们显著增加了树形两阶段提交的复杂性。日志流节点既可以作为级联协调者（负责其子节点），也可以作为参与者（隶属于其父节点），因此未知状态机制会增加状态机的复杂性。我们的目标是减少必须向调用方暴露“未知”状态的场景。

为避免复杂的处理逻辑，并在常见情况下剥离未知状态，我们引入了 MaLT [9] 中的 事务数据表 (TDT)。TDT 存储此日志流上每个已达到终止状态的事务的已决状态（已提交或已中止）。当参与者丢失其事务上下文（例如，清理或恢复后）并随后收到查询消息（例如，重复消息或重试的准备请求）时，它会先检查 TDT。如果找到该事务且其决定已确定，参与者便会回复实际决定（例如，根据已提交/已中止的结果回复 PREPARE OK），而不是回复准备未知。随后，协调器可以正常继续，而无需进入未知状态路径，也无需向用户返回事务未知。TDT 中的事务状态会保留一段可配置的时间（例如，默认 30 分钟），足以应对大多数重复或延迟消息（例如，重试和延迟交付）。因此，在事务已提交或已中止，且查询方发送的是重复消息或重放消息这一常见情况下，参与者可以通过 TDT 处理查询，避免暴露“未知”状态，从而简化状态机并减少面向用户处理事务未知的需求。

6 讨论

本节讨论我们的树形两阶段提交框架的若干重要方面。我们首先介绍用于提升事务性能的优化，包括行锁释放延迟降低和 CLEAR 日志最小化。随后，我们使用 TLA+ 对协议的安全性和活性进行形式化验证。最后，我们讨论与现有 OceanBase 模块的集成考量，包括 liboblog 和备份恢复机制。

6.1 优化

6.1.1 释放消息。 在我们的树形两阶段提交实现中，我们观察到，虽然事务延迟保持不变且资源消耗有所降低，但资源释放的延迟增加了一个日志操作的时长。在这些资源中，行锁和日志是两个关键组成部分 [3, 10, 14]：行锁代表事务的竞争足迹，而日志则决定日志回收。虽然日志回收延迟本质上是可容忍的，并且可以通过延迟批处理加以缓解，但行锁竞争是不可容忍的：更高的竞争足迹会直接降低事务吞吐量。

为解决这一问题，我们提出了一项优化，以降低行锁释放延迟。协调器同步写入 COMMIT 日志，可确保在其他参与者发生故障时，参与者能够通过查询协调器的事务状态安全地释放日志资源，从而保证参与者之间不存在残留依赖

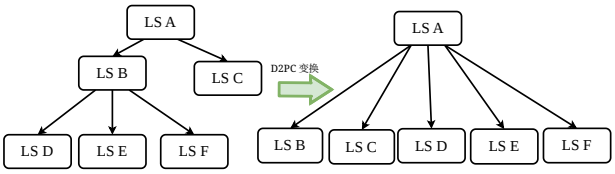


图 8：D2PC 优化。

。不过，释放行锁并不需要执行这一同步日志操作。任何参与者一旦确认事务已成功提交，即可安全地释放行锁。

我们使用两种不同的消息，将行锁释放与日志回收解耦。协调器进入提交点（即收到所有准备确认）后，会并行发送 RELEASE 消息（用于释放行锁）并同步 COMMIT 日志。只有 COMMIT 日志同步成功后，协调器才会发送 COMMIT 消息（触发日志资源清理）。这样既能将日志回收延迟维持在两次日志同步的水平，又能通过以额外一次消息往返取代一次日志同步，降低行锁释放延迟。

以下总结了优化后的树形两阶段提交算法（假设有 N 个参与者且提交成功）：

- **事务延迟**：2 次消息往返（协调器的 PREPARE 请求、参与者的 PREPARE 响应）和 1 次日志同步（参与者的 PREPARE 日志）。
- **行锁释放延迟**：3 次消息往返（协调器的 PREPARE、RELEASE 请求；参与者的 PREPARE 响应）和 1 次日志同步（参与者的 PREPARE 日志）。
- **日志回收延迟**：3 次消息往返（协调器的 PREPARE、COMMIT 请求；参与者的 PREPARE 响应）和 2 次日志同步（协调器的 COMMIT 日志、参与者的 PREPARE 和 COMMIT 日志）。
- **事务资源消耗**：5N 次消息传输（包括协调器的 PREPARE、RELEASE 和 COMMIT 请求，以及参与者的 PREPARE 和 COMMIT 响应）、 2N+1 条同步复制的多数派确认日志（协调器的 COMMIT 日志和参与者的 PREPARE 和 COMMIT 日志），以及 1 条异步复制的多数派确认日志（协调器的 CLEAR 日志）。

6.1.2 通过 *D2PC* 优化 *CLEAR* 日志。我们观察到，只有叶子节点不需要写入 CLEAR 日志，而在我们的优化框架下，这可能会造成效率低下。值得注意的是，由于分区迁移，PREPARE 和 COMMIT 阶段的参与者列表可能不同。为解决两阶段提交过程中频繁的分区迁移导致 CLEAR 日志过多的问题，我们采用 D2PC（动态去中心化两阶段提交）方法 [8]。如图 8 所示，该方法确保只有根日志流（日志流 A，即 LS A）需要写入 CLEAR 日志。通过在不同阶段之间动态调整参与者列表，可以在保持原子性和一致性保证的同时，最大限度地减少冗余日志记录。

6.2 TLA+证明

本节证明所提出的树形两阶段提交协议具备安全性和活性。该协议以 TLA+ 规约（附录 C，模块 2pc tla）进行描述。与之前扁平的 TM/RM 抽象不同，该规约对每个节点采用统一建模：一个节点可以作为协调器与其子节点交互，同时作为参与者与其父节点交互。节点的本地状态记录在资源管理器状态中，父子拓扑则由变量 父节点 和 子节点表示。转移引入的动态成员关系由一个中间子节点集合表示，该集合仅在阶段边界处并入活动子节点集合。所有协议操作（准备、决策、传播、确认、内部中止和转移）均在规约的下一状态关系中定义。

6.2.1 安全性证明。 对于安全性，我们必须确保所有节点始终对相同的结果达成一致，并且一旦某个节点达到提交或中止状态，就不会再更改该决定。

定理 6.1。 根据 *TLA+* 规约，所提出的树形两阶段提交协议的安全性属性直接继承自传统两阶段提交协议的安全性属性。

证明见附录 A.4。

6.2.2 活性证明。 对于活性，我们要求根节点启动协议后，所有节点最终都能在有限次转换内达到终止状态（提交或中止）。

定理 6.2。 根据 *TLA+* 规约，在标准公平性假设下，所提出的树形两阶段提交协议的活性由传统两阶段提交协议的活性推导而来。

证明见附录 A.5。

7 实验评估

本节将对我们在 OceanBase 中采用的树形两阶段提交框架进行实验评估，重点考察其在分区迁移期间的表现。

7.1 实验设置

本小节详细介绍评估所用的实验环境和方法。所有实验均在共享无（SN）部署模式下，基于 OceanBase 4.4.1 版本（社区版）进行。为全面评估所提出的设计，我们采用包括 TPC-C 和 sysbench 在内的标准 OLTP 工作负载。除非另有说明，所有实验均遵循下文所述的统一部署配置、工作负载设置和评估指标。

7.1.1 系统配置。 我们部署了一个最多包含八个 OBServer 节点和一个代理节点的集群，所有 OBServer 节点具有相同的硬件配置。每个 OBServer 节点配备 32 核 Intel Xeon CPU、128 GB 内存和 2 TB SSD。该集群运行于 Ubuntu 24.04，并部署在阿里云的云环境中。每个日志流采用双副本 Paxos 配置进行复制，副本放置在不同的 OBServer 节点上。我们同时评估启用转移与禁用转移两种设置，并在必要时应用定制的事务相关配置。除非另有说明，所有实验均使用相同的系统配置。

7.1.2 工作负载。 我们使用 TPC-C 基准测试来评估分区迁移对高并发下在线事务处理的影响。TPC-C 模拟了真实的多表 OLTP 工作负载，其中分布式事务占比较高，因此非常适合考察分区迁移如何影响事务延迟、吞吐量和性能稳定性。在实验中，我们在稳态执行阶段周期性注入分区迁移，以模拟弹性部署环境中的持续负载均衡。除非另有说明，TPC-C 配置为 3,000 个仓库和 500 个并发客户端连接。每次实验运行一小时，包括初始预热阶段以及随后的稳态测量。在稳态阶段，分区迁移以三分钟的固定间隔触发。

7.1.3 指标。为评估框架的性能和健壮性，我们采用两类主要指标：吞吐量和延迟。对于 TPC-C 基准测试，吞吐量以每分钟事务数（*tpmC*）衡量。事务延迟从客户端侧采集，并报告平均延迟和第99百分位数（P99）延迟。我们特别将 P99 作为尾延迟的关键指标，因为它能有效反映分布式协调开销以及分区迁移期间可能出现的停滞所带来的影响。此外，我们通过监测执行期间的延迟波动来评估系统稳定性。我们不仅报告汇总统计数据，还会分析分区迁移活动期间 P99 峰值的分布，从而量化框架的韧性，以及它在底层日志流拓扑动态变化时维持性能可预测性的能力。

7.2 转移对在线事务的影响

在本小节中，我们评估分区迁移对在线事务处理性能的影响。我们重点关注在现实的高并发工作负载下，迁移对事务延迟、吞吐量和性能稳定性的影响。

7.2.1 实验设置与基线。 我们使用 TPC-C 基准测试评估分区迁移对在线事务性能的影响。为隔离动态拓扑变化的具体影响，我们重点研究框架的两种主要配置：

- **OB（禁用迁移）：** 在整个实验过程中停用分区迁移，以建立峰值性能基线。
- **OB-Transfer（启用迁移）：** 在稳态阶段以固定的三分钟间隔定期触发分区迁移，以模拟负载均衡。

为提供更广泛的性能背景，我们将这些配置与两个外部基线进行比较：MySQL（v8.0），一种独立数据库；以及 TiDB [11]（v8.1），一种使用传统两阶段提交协调的分布式数据库。除非另有说明，否则各项工作负载参数（例如 3,000 个仓库）在所有配置中保持一致。

7.2.2 转移期间的整体性能。 图9展示了分区迁移期间 TPC-C 的延迟—吞吐量特征。结果揭示了我们的框架在静态和动态条件下效率方面的几项关键见解。

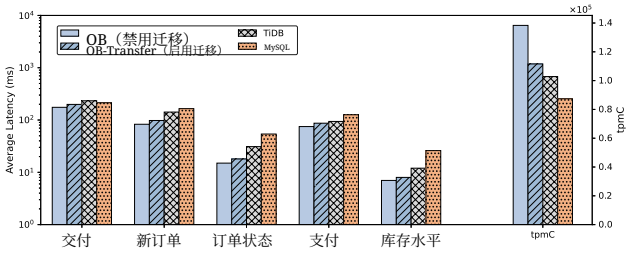


图9：分区转移下 TPC-C 延迟–吞吐量对比。

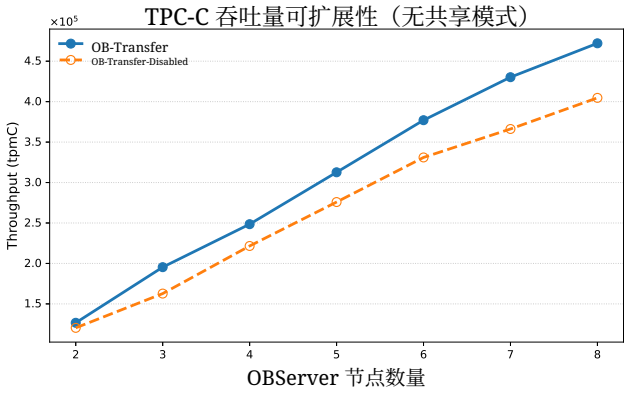


图 10：TPC-C 吞吐量随集群规模变化的可扩展性。

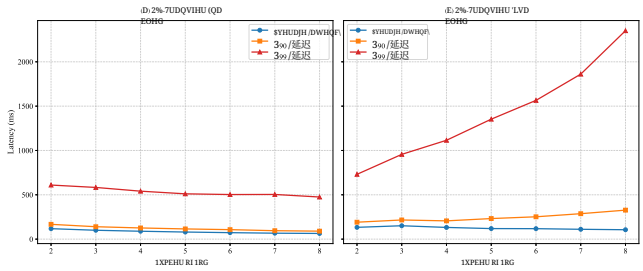


图 11：不同集群规模下的事务延迟分布。

吞吐量与系统韧性。 如图9右侧面板所示，基准 OceanBase（OB）的吞吐量在受评估系统中最高。启用分区迁移（OB-Transfer）后，由于日志流迁移期间需要额外协调，吞吐量有所下降，但降幅适中。不过，系统仍能保持稳定的吞吐量，没有出现骤降，表明转移过程与事务执行集成良好。

这种现象可归因于树形两阶段提交协议：该协议允许事务的参与者集合在执行期间动态变化，而无需中止事务或设置全局同步屏障。通过将事务参与者组织成层次结构，该协议限制了转移引入的协调范围，确保额外开销以可控的方式增长，而不会扩散到所有参与者。

相比之下，在相同的工作负载配置下，TiDB 的吞吐量较低，这反映出跨多个节点执行分布式事务所产生的协调开销更高。在这种高并发场景下，MySQL 的吞吐量显著较低。这主要是因为在 500 个并发客户端连接下，单节点资源达到饱和，CPU 调度、锁管理和日志序列化争用有限的共享资源。随着并发增加，这些争用点逐渐成为主要瓶颈，导致吞吐量迅速饱和，而非逐步扩展。

事务延迟特征与根本原因。 图9左侧面板以对数刻度展示了各类 TPC-C 事务的客户端延迟。OceanBase 在所有事务类别中的延迟均持续低于 TiDB，表明将多个分区整合到单个日志流中的有效性。这种单日志流（SLS）优化减少了参与提交过程的事务参与者数量，从而最大限度地减少了 RPC 调用和跨节点协调开销。

一个重要观察是，OB 与 OB-Transfer 的延迟柱形图非常接近。即使对于新订单等协调密集型事务，

支付，启用转移仅会导致延迟略微增加。这是因为树形提交协议将转移引发的拓扑变化的影响限制在提交路径的一小部分，从而避免协调延迟大范围扩散，并保持较低的端到端延迟。

有趣的是，在所评估的高并发工作负载下，OceanBase 的分布式部署也比独立 MySQL 基线呈现出更有利的延迟表现。通过将客户端连接和事务执行分散到多个 OBServer 节点上，OceanBase 缓解了对集中式资源的竞争。相比之下，MySQL 在单一执行上下文中处理所有事务，其中并发增加会加剧锁竞争和日志瓶颈，导致延迟升高、吞吐量降低。

总体而言，这些结果表明，所提出的框架不仅优化了稳定执行状态下的分布式事务协调，还能有效应对分区迁移带来的额外复杂性，从而在弹性环境中实现可预测且高效的在线事务处理。

7.3 分区迁移下的可扩展性

本小节评估共享无（SN）模式下集群扩展期间分区迁移的可扩展性。主要目标是验证，随着 OBServer 节点数量增加，所提出的迁移感知事务框架能否保持高吞吐量和稳定的延迟。

7.3.1 实验方法。为评估可扩展性，我们改变 OceanBase 集群中的 OBServer 节点数量，同时保持工作负载和系统配置与第

7.1 节中的设置不变。所有实验均在启用分区转移的 SN 模式下进行。我们使用配置为 3,000 个仓库和 500 个并发客户端连接的 TPC-C 基准测试。

我们将集群从 2 个扩展至 8 个 OBServer 节点，并评估各个中间集群规模下的系统性能。对于每种配置，我们测量其达到的每分钟事务数（tpmC）吞吐量，以及客户端侧的事务延迟。此实验设计能够隔离集群扩展对转移行为和事务处理性能的影响。

7.3.2 集群扩展下的吞吐量可扩展性。图 10 展示了 TPC-C 吞吐量随 OBServer 节点数量变化的情况。结果表明，在禁用转移和启用转移的配置中，吞吐量均表现出近线性可扩展性，说明资源利用有效。

值得注意的是，由于分区迁移固有的协调与日志记录开销，启用转移的配置吞吐量略低；但随着集群扩展，两种配置之间的性能差异保持不变。吞吐量曲线呈现平行增长趋势，这表明分区迁移引入的开销有界，不会随集群规模增大而累积。

这种可扩展性直接得益于我们基于日志流的架构和树形两阶段提交协议。通过将共置分区聚合到单个日志流参与者中，该协议限制了事务涉及的参与者数量。此外，分区迁移只会在提交树的局部化分支上引入额外参与者，而不会扩大整体协调范围。因此，集群横向扩展能够提升总处理能力，而不会按比例增加每笔事务的协调成本。

7.3.3 不同集群规模下的延迟稳定性。 图 11 报告了不同集群规模下事务延迟（平均值、P90 和 P99）的分布情况。结果表明，随着 OBServer 节点数量增加，事务延迟仍保持稳定。

至关重要的是，在横向扩展过程中，尾延迟（P99）没有出现明显恶化，证实与转移相关的协调开销仍处于有界范围内。尽管分区迁移需要额外的日志记录和消息交换，但这些成本仅限于事务提交路径中涉及的特定日志流，不会随集群规模增大而累积。这些发现证实，我们的框架有效地将可扩展性与动态分区管理解耦，从而确保在弹性、无共享环境中实现可预测的性能。

7.4 传输效率与开销

在本小节中，我们评估高并发事务型工作负载下分区迁移的执行效率和资源开销。本实验旨在考察分区迁移在与分布式事务并发执行时，能否保持轻量且可预测，而不会成为性能或资源瓶颈。

7.4.1 传输效率实验设置。 我们在启用了分区迁移的 OceanBase 集群上进行实验。

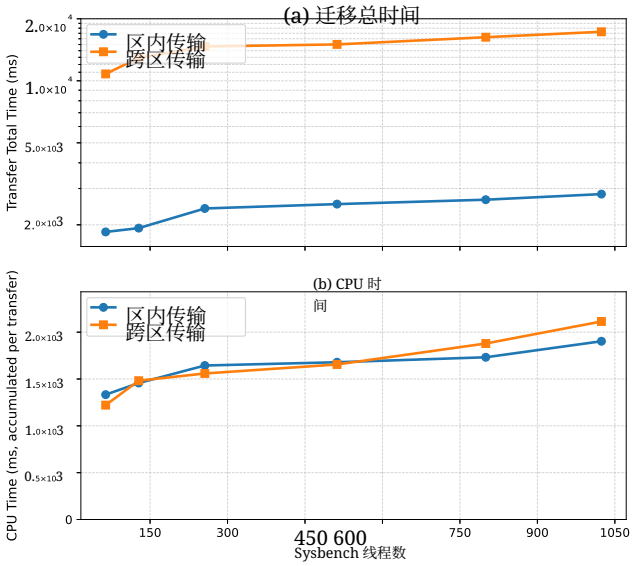


图 12：不同 sysbench 线程数下的分区迁移执行时间与 CPU 开销。

工作负载由 sysbench 生成，包含分布式读写事务。并发客户端线程数从 64 变化到 1024，以模拟不断增大的事务压力。每个 sysbench 事务包含点查询和删除-插入操作，并在事务上下文中执行。

分区迁移以表粒度执行。集群包含 30 张表；每次迁移会随机选择一张表，并将其从当前日志流迁移到另一个随机选定的日志流。该策略可避免偏向特定表或访问模式，并能反映现实的迁移调度。我们评估了两种场景：区内传输，即源端和目标日志流位于同一区域；以及跨区传输，即迁移跨越不同区域。

7.4.2 迁移执行时间。图 12(a) 展示了不同 sysbench 线程数下分区迁移的总执行时间。随着事务压力增加，迁移执行时间缓慢增长，并且在所有测试配置中都保持稳定。即使在高并发下，传输延迟也不会突然增加或出现不稳定。

由于跨区通信和协调需要额外开销，跨区域传输的延迟高于区内传输。不过，随着事务负载增加，这两种场景之间的性能差距始终保持稳定。两条趋势线几乎平行，表明跨区通信带来的开销有限，不会随并发事务强度增加而扩大。综合来看，这些结果表明，分区迁移的执行与前台事务压力基本上相互独立，不会干扰关键路径上的事务处理。

7.4.3 分区迁移的 *CPU* 开销。 图 12(b) 展示了不同 sysbench 线程数下每次转移所消耗的累计 CPU 时间。CPU 时间通过汇总以下操作的执行开销进行测量：

传输相关操作，包括事务筛选、传输日志生成、元数据重构、事务迁移以及物理传输阶段。

结果表明，随着事务压力增加，每次转移的 CPU 消耗会适度上升，但仍处于较窄且可预测的范围内。与传输延迟的情况相同，跨区传输消耗的 CPU 资源多于区内传输，但这种差异在不同负载水平下保持稳定。有界的 CPU 开销表明，即使在高并发工作负载下，分区迁移也不会成为 CPU 瓶颈，这证实了该传输机制较为轻量，可以与密集的分布式事务处理并行运行。

7.5 树形提交的性能

本小节评估提交树的结构如何影响分区迁移下的事务性能。我们的树形两阶段提交协议将日志流组织为层级化参与者，而迁移可能会沿提交树的局部化分支嵌入额外的日志流。如第 3.4 节所述，树形两阶段提交的关键路径开销主要由树高H决定：事务响应延迟需要 $2H$ 次消息往返和一次日志同步，而锁释放延迟需要 $3H$ 次往返和两次日志同步。相比之下，增加扇出主要会增加并行分支，而不会延长协调轮次。

遵循第7.1节的评估方法，我们同时报告吞吐量与客户端延迟指标。我们将尾延迟作为协调停顿与传输诱发的不稳定性的关键指标加以强调，这与我们此前优先关注动态转移事件下P99行为的评估保持一致。

7.5.1 方法：构建宽度/深度受控的提交树。 我们通过操控转移模式来构建受控宽度 与深度的提交树，同时保持工作负载强度和系统资源不变。为增加**树深度**，我们在不同日志流组之间反复转移同一分区，形成一条延长提交路径的传输链（即增大H）。为增加**树宽度**，我们在同一层级将多个分区转移到不同的日志流组，从而在保持深度较浅的同时扩大扇出。随后，我们在启用传输与关闭传输基线条件下测量所达到的吞吐量（每秒事务数）以及事务延迟（均值、P90、P99）。

7.5.2 树宽度的影响。 图13a报告了随着树宽度增加的吞吐量。我们观察到，在启用传输条件下，每秒事务数在很大的宽度范围内保持稳定，仅有轻微波动。这表明增大扇出不会显著影响关键提交路径。该结果与第3.4节的分析一致：宽度主要增加并行分支的数量，而关键路径上的协调轮次数量保持不变。

图13b展示了相应的延迟分布。随着宽度增大，均值与P90延迟仅略有上升，表明平均协调开销仍保持有界。相比之下，P99随宽度增大呈现适度的上升趋势。

这种行为在分层协调中是预期之内的：更大的扇出会提高在并行参与者中遇到落后分支（例如瞬时排队、日志刷新延迟或调度延迟）的概率，这主要体现在尾延迟上。重要的是，P99的上升是渐进的而非骤然的，表明由宽度引起的开销不会放大为全系统不稳定性；相反，它仍局限于受影响的提交分支。

7.5.3 树深度的影响。接下来，我们增加树深度，以直接考察协调路径长度的影响。图14a显示，吞吐量随深度单调下降。与宽度不同，深度会增加协调跳数，并增加关键路径上的同步日志记录量，因此在客户端并发度固定时，事务完成率会降低。值得注意的是，吞吐量下降平缓且可预测，我们没有观察到突然崩溃，这表明即使在人为构造的深层传输链下，传输感知提交协议仍保持稳定。

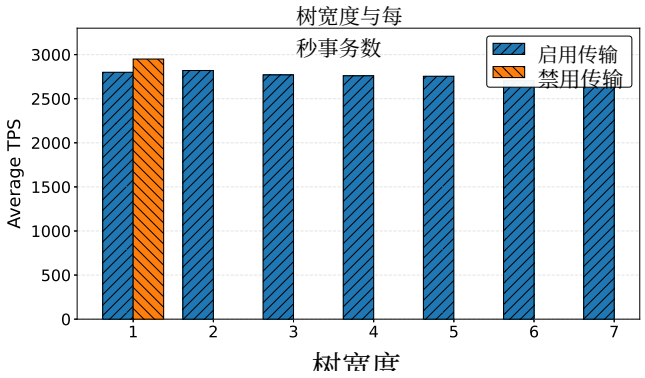
图14b进一步证实，所有报告的百分位延迟都会随深度增加而上升，其中 P99 的增幅比平均值和 P90 更明显。这一趋势与第3.4节的理论模型直接吻合：事务响应延迟需要 $2H$ 次消息往返，而锁释放延迟需要 $3H$ 次往返。随着H 增大，每一层的延迟（例如 RPC 调度、排队和复制日志同步）都会沿关键路径累积，从而放大尾延迟。尽管延迟有所增加，其增幅仍与深度成比例，并未出现失控式放大，这支持了我们先前的论断：传输开销仅限于局部化的提交路径，而不会扩大到全局协调范围。

这些结果验证了我们的树形两阶段提交协议在转移场景下的结构化性能特性。树的深度是增加协调成本和尾延迟的主导因素，而树的宽度对吞吐量和平均延迟的影响有限，仅通过拖尾任务概率对 P99 产生中等程度的影响。对提交树几何结构的可预测敏感性表明，我们面向转移设计的方案引入了有界且局部化的开销，并在分区动态迁移期间保持了稳定的事务性能。

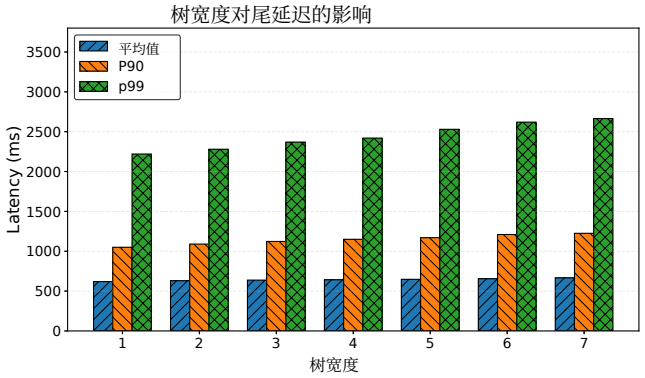
7.6 协调可扩展性：从分区粒度到日志流粒度

本小节评估 OceanBase 3.x 与 4.x 之间在基础协调可扩展性方面的差异。第7.5节研究了树形两阶段提交在转移场景下的结构敏感性，而本文将探讨一个更为基础的架构问题：协调粒度的选择如何影响事务提交的可扩展性。

OceanBase 3.x 按分区粒度组织两阶段提交参与者。每个分区独立参与提交协议，而跨越N 个分区的分布式事务会引入N 个参与者。相比之下，OceanBase 4.x 引入了日志流粒度，将多个分区聚合到一个日志流中，并作为一个协调单元参与两阶段提交。这种设计减少了有效参与者数量，并重新定义了分布式事务的协调边界。

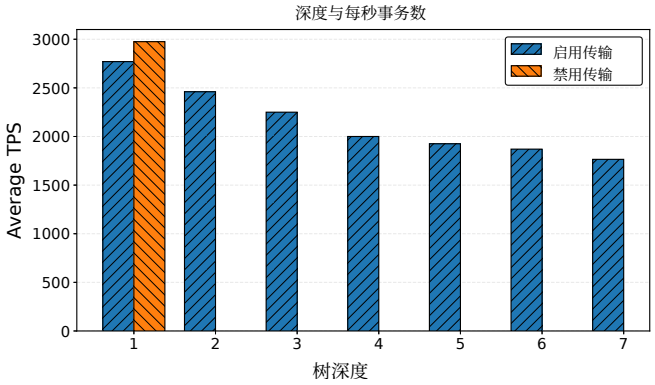


(a) 吞吐量与树宽度的关系。

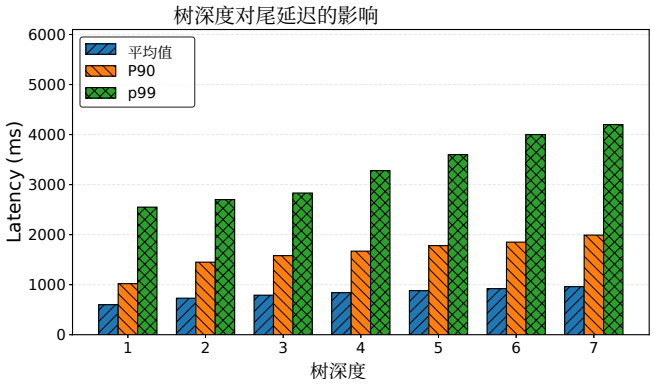


(b) 延迟与树宽度的关系。

图13：转移场景下的树宽度敏感性。



(a) 吞吐量与树深度的关系。



(b) 延迟与树深度的关系。

图14：转移过程中的树深度敏感性。

7.6.1 动机。在大规模分布式系统中，协调可扩展性是一个关键瓶颈。按分区粒度组织时，两阶段提交参与者的数量会随着事务涉及的分区数量线性增长。随着分区数量增加，系统会面临不断加剧的消息扇出、确认聚合开销、调度争用以及复制日志同步压力。即使单条消息的处理效率很高，协调参与者数量本身仍会成为主要成本因素。

日志流粒度从根本上改变了这种扩展行为。通过将分区聚合到日志流中，多个数据分片共享一个协调参与者。因此，两阶段提交参与者的数量随日志流的数量增长，而不是随原始分区数量增长。由于日志流的数量显著少于分区数量，这种聚合将协调复杂度从分区级范围提升到更高层次的结构抽象。

本实验旨在定量验证，4.x 的可扩展性提升源于结构性协调减少，而非渐进式优化。

7.6.2 实验设置。 我们构造了跨越不同数量分区的事务，分区数量从数百个到数万个不等。对于每种配置，我们分别测量 3.x（分区粒度）和 4.x（日志流粒度）下的两阶段提交延迟。为隔离协调开销，我们仅测量提交阶段的延迟，同时确保不同版本的工作负载强度、数据分布和硬件配置完全相同。本实验不涉及任何传输操作，从而确保测得的开销反映的是纯粹的两阶段提交协调行为。

7.6.3 结果与分析。 图15展示了随着分区数量增加而测得的提交延迟。在 3.x 下，提交延迟会随着分区数量增加而迅速增长。增长趋势接近线性，这反映出每增加一个分区，就会在协调阶段引入另一个独立参与者。当 N 变得很大时，累积的协调开销会显著增加消息聚合延迟、日志同步压力和调度争用。在分区数量较高时，3.x 无法在合理时间内完成提交，展现出协调放大和可扩展性崩溃。

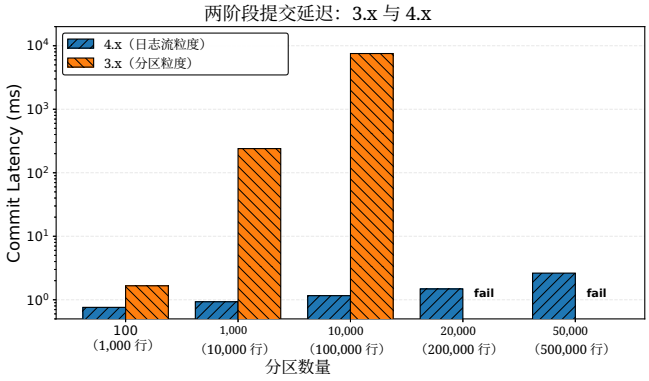


图15：两阶段提交延迟对比。

相比之下，4.x 的延迟增长速度明显更慢。尽管提交延迟仍会随着分区数量增加而上升，但增长斜率显著降低。这一表现证实，日志流聚合有效地将协调参与者数量与原始分区数量解耦。同一日志流中的多个分区在提交树中仅表示为一个参与者，从而避免参与者数量激增，并限制协调轮次的增长。

重要的是，可扩展性的提升是结构性的。4.x并非优化RPC处理或降低日志记录成本，而是缩小了协调范围本身。这一转变将两阶段提交的扩展特性从分区级线性增长改变为日志流级有界增长。因此，即使在分区数量使3.x变得不稳定的情况下，4.x仍能可预测地运行，并保持可接受的提交延迟。

结果表明，从分区粒度到日志流粒度的转变并非仅仅是实现层面的改进，而是根本性的架构演进。通过重新定义迁移与协调的原子单元，4.x避免了协调开销激增，并为感知迁移的分布式事务奠定了可扩展的基础。这一结构性改进也解释了为何4.x中的树形两阶段提交设计在动态分区迁移下仍能保持稳定行为，正如第7.5节所验证的那样。

总体而言，该实验证实日志流聚合显著提升了协调可扩展性。与3.x中的分区级两阶段提交相比，4.x中的日志流级协调在分区数量增加时实现了显著更优的提交延迟扩展，从而验证了所提设计的架构进步。

8 相关工作

分布式事务协调一直是大型数据库系统面临的核心挑战。传统两阶段提交协议 [2, 7, 13,15] 构成了 ACID 保证的基础，但由于依赖逐分区协调，在现代系统中面临可扩展性和延迟方面的问题。近期，分布式数据库通过创新的优化方法和架构创新来解决这些局限。本节回顾了以下三个关键领域的相关工作：全局事务协调、日志结构存储优化，以及带转移处理的动态分区。

8.1 全局事务协调

Spanner [6] 等系统引入了混合逻辑时钟和 TrueTime API，以管理具有强一致性的全局事务，但它们仍依赖逐分区协调，因此继承了传统两阶段提交的可扩展性瓶颈。Calvin [17]转而采用确定性执行和基于日志的调度，以降低协调开销；CockroachDB [16]则利用混合逻辑时钟优化地理分布式环境中的两阶段提交。

然而，这些方法并未明确解决动态分区迁移带来的开销，也未利用日志流聚合来减少参与者数量。我们的框架将日志流视为原子参与者；当多个分区位于同一日志流上并共享该日志流时，这种做法能大幅减少协调消息。

8.2 日志结构化存储与优化

日志结构化设计在提升吞吐量和持久性方面日益受到关注。日志结构化系统利用日志结构化存储构建高吞吐量的事务型键值存储，并强调原子性和隔离性。FASTER [4] 进一步优化了日志结构化存储，以支持低延迟操作，从而展现了以日志为中心的架构的潜力。

MaLT [9] 在 OceanBase 的 LSM 树存储引擎中管理大型事务状态和恢复。虽然这些工作侧重于存储效率和吞吐量优化，但并未将日志流整合为用于事务协调的原子单元——而这正是我们的单日志流两阶段提交框架的核心能力。我们通过将日志流本身而非单个分区用作协调单元，将日志结构化原则扩展到事务协调。

8.3 动态分区与转移

现有的动态分区解决方案，例如 CockroachDB [16], 中的方案，通常会在迁移后重新验证参与者列表，或导致分区与日志之间出现循环依赖 [12, 15]。当分区在事务进行期间迁移时，这些方案需要显式更新参与者列表，并引入复杂的恢复逻辑。相比之下，我们的树形两阶段提交协议通过递归跟踪日志流依赖，自然地支持分区迁移，无需显式更新参与者列表，也简化了恢复逻辑。此设计还通过自动树构建解决了“循环事务”问题 [12], ，即使在在线迁移期间也能确保线性一致的提交推进。当事务在迁移和提交期间因锁而发生竞争时，OceanBase 会依靠分布式死锁检测与解除机制（LCL [20], LCL+ [21]）来避免无限期阻塞。

上述相关工作与我们的树形两阶段提交框架的主要区别在于，我们将日志流 而非分区作为原子参与者和协调单元，并且原生支持动态分区迁移和上下文丢失（通过递归日志流树和未知状态机制），无需在迁移后重新验证参与者列表，也无需进行循环依赖处理。尽管先前的系统在

分布式事务管理方面取得了进展，但它们要么（1）将各个分区视为原子参与者，导致可扩展性瓶颈；要么（2）难以应对迁移过程中的动态拓扑结构。我们的工作通过将日志流抽象为原子单元、减少参与者数量，并引入未知状态机制以在上下文丢失期间保持一致性，弥补了这些不足。这些创新符合现代日志结构化系统的原则 [4]，同时也应对了动态云原生环境中的特定挑战。

9 结论

本文提出了一种面向 OceanBase 的新型树形两阶段提交框架，旨在解决分布式事务协调中的可扩展性和延迟挑战。通过将原子单元从单个分区重新定义为单机日志流，我们的方法将参与者数量减少了几个数量级（例如，100 个分区时减少 99%），从而最大限度地降低协调开销和带宽消耗。树形两阶段提交协议通过以协调者为根的有向无环图拓扑动态处理分区迁移，无需显式更新参与者列表，并消除循环依赖。在在线迁移期间，迁移上下文会作为叶节点递归嵌入提交树中，确保拓扑变化期间的线性一致性进展。此外，未知状态机制（准备未知、事务未知）可在上下文丢失时防止一致性违规，避免因参与者响应具有误导性而导致错误中止，同时使用户免受歧义影响。

在 OceanBase 4.4.1 上进行的实验评估表明，我们的框架性能接近单机事务，同时显著降低了延迟和带宽消耗。该系统在重负载下表现出线性可扩展性，并在并发分区迁移期间保持稳定性能。这项工作将强一致性与高性能相结合，为下一代分布式系统提供了可扩展的蓝图。通过将日志流抽象为原子单元并引入稳健的拓扑处理机制，我们的解决方案为地理分布式架构中的低延迟、高吞吐量事务树立了新标准。OceanBase 中的实际实现验证了其在真实场景中的有效性，尤其适用于要求弹性资源效率和云原生环境下动态分区管理的应用。

参考文献

[1] Ghazi Alkhatib 和 Ronny S Labban。2002。分布式数据库系统中的事务管理：Oracle 两阶段提交案例。*Journal of Information Systems Education* 13，2，95–104。[2] Muhammad Atif。2009。两阶段提交与三阶段提交协议的分析与验证。载于 *2009* 新兴技术国际会议。IEEE，326–331。[3] Claude Barthels、Ingo Müller、Konstantin Taranov、Gustavo Alonso 和 Torsten Hoefler。2019。强一致性并不难实现：数千个核心上的两阶段锁定与两阶段提交。*Proceedingsofthe VLDB Endowment* 12，13（2019），2325–2338。[4] Badrish Chandramouli、Guna Prasaad、Donald Kossmann、Justin Levandoski、James Hunter 和 Mike Barnett。2018。FASTER：支持原位更新的并发键值存储。载于*2018* 年国际数据管理会议论文集。275–290。[5] Fay Chang、Jeffrey Dean、Sanjay Ghemawat、Wilson C. Hsieh、Deborah A. Wallach、Mike Burrows、Tushar Chandra、Andrew Fikes 和 Robert E. Gruber。2008。Bigtable：结构化数据分布式存储系统。*ACM Trans. Com- put. Syst.* 26，2，文章 4（2008 年 6 月），26 页。https://doi.org/10.1145/1365815。1365816[6] James C Corbett、Jeffrey Dean、Michael Epstein、Andrew Fikes、Christopher Frost、Jeffrey John Furman、Sanjay Ghemawat、Andrey Gubarev、Christopher Heiser、Peter Hochschild 等。2013。Spanner：谷歌的全球分布式数据库。《ACM计算机系统汇刊》（TOCS）31卷3期（2013），1–22。[7] Bipin C Desai 和 Boutros S Boutros。1996。两阶段提交协议的性能。《信息与软件技术》38卷9期，581–599。[8] Paul Ezhilchelvan、Amjad Aldweesh 和 Aad van Moorsel。2018。利用区块链实现非阻塞两阶段提交。载于《第1届分布式系统加密货币与区块链研讨会论文集》。36–41。[9] Chenguang Fang、Chen Qian、Qi Yang、Zeyu Wang、Zhenkun Yang、Fanyu Kong、Quanqing Xu、Hui Cao、Fusheng Han 和 Chuanhui Yang。2025。MaLT：OceanBase大型事务管理框架。载于《2025年国际数据管理会议伴随论文集》。378–390。[10] James N Gray。2005。数据库操作系统札记。《操作系统：高级课程》，393–481。[11] Dongxu Huang、Qi Liu、Qiu Cui、Zhuhe Fang、Xiaoyu Ma、Fei Xu、Li Shen、Liu Tang、Yuxing Zhou、Menglong Huang 等。2020。TiDB：基于Raft的HTAP数据库。《VLDB基金会会刊》13卷12期（2020），3072–3084。[12] Heine Kolltveit 和 Svein-Olaf Hvasshovd。2007。循环两阶段提交协议。载于《高级数据库系统国际会议》。施普林格，249–261。[13] Butler Lampson 和 David Lomet。1993。两阶段提交的一种新型推定提交优化。载于《第19届超大型数据库国际会议（VLDB'93）》。630–640。[14] In Kyung Ryu 和 Alexander Thomasian。1990。采用动态锁定的数据库性能分析。《ACM学报》（JACM）37卷3期，491–523。[15] George Samaras、Kathryn Britton、Andrew Citron 和 C Mohan。1995。商业分布式环境中的两阶段提交优化。《分布式与并行数据库》3，325–360。[16] Rebecca Taft、Irfan Sharif、Andrei Matei、Nathan VanBenschoten、Jordan Lewis、Tobias Grieger、Kai Niemi、Andy Woods、Anne Birzin、Raphael Poss 等。2020。CockroachDB：弹性地理分布式SQL数据库。载于《2020年ACM SIGMOD数据管理国际会议论文集》。1493–1509。[17] Alexander Thomson、Thaddeus Diamond、Shu-Chun Weng、Kun Ren、Philip Shao 和 Daniel J Abadi。2012。Calvin：面向分区数据库系统的快速分布式事务。载于《2012年ACM SIGMOD数据管理国际会议论文集》。1–12。[18] Gerhard Weikum 和 Gottfried Vossen。2001。《事务信息系统：并发控制与恢复的理论、算法及实践》。摩根·考夫曼出版社，旧金山，美国加利福尼亚州。[19] Chao Xie、Chunzhi Su、Manos Kapritsos、Yang Wang、Navid Yaghmazadeh、Lorenzo Alvisi 和 Prince Mahajan。2014。Salt：在分布式数据库中结合{ACID}与{BASE}。载于《第11届USENIX操作系统设计与实现研讨会（OSDI 14）》。495–509。[20] Zhenkun Yang、Chen Qian、Xuwang Teng、Fanyu Kong、Fusheng Han 和 Quanqing Xu。2023。LCL：一种用于死锁检测与解决的基于锁链长度的分布式算法。载于《2023年IEEE第39届国际数据工程会议（ICDE）》。IEEE，151–163。[21] Zhenkun Yang、Chen Qian、Xuwang Teng、Fanyu Kong、Fusheng Han、Quanqing Xu 和 Daokun Hu。2025。LCL+：为OceanBase构建的基于锁链长度的分布式死锁检测与解决服务。《ACM计算机系统汇刊》43卷4期（2025），1–33。[22] Zhifeng Yang、Quanqing Xu、Shanyan Gao、Chuanhui Yang、Guoping Wang、Yuzhong Zhao、Fanyu Kong、Hao Liu、Wanhong Wang 和 Jinliang Xiao。2023。OceanBase Paetica：一种支持单机和分布式集群的混合式无共享/全共享数据库。《VLDB会刊》16卷12期（2023），3728–3740。[23] Zhenkun Yang、Chuanhui Yang、Fusheng Han、Mingqiang Zhuang、Bing Yang、Zhifeng Yang、Xiaojun Cheng、Yuzhong Zhao、Wenhui Shi、Huafeng Xi、Huang Yu、Bin Liu、Yi Pan、Boxue Yin、Junquan Chen 和 Quanqing Xu。2022。OceanBase：707百万tpmC分布式关系数据库系统。《VLDB会刊》15卷12期（2022），3385–3397。

A 详细证明

A.1 定理2.2的证明

证明。令P为参与并发控制的分区集合，$L^{final}(p)$表示分区$p \in P$的最终日志流位置。最小集合要求为$\{L^{final}(p) : p \in P\}$。

日志流树通过纳入以下内容构建：(1) 分区 P 执行操作的初始日志流；(2) 所有分区迁移产生的目的地日志流，并递归地沿着转移记录追踪。

完备性：对于任意 $p \in P$，如果 p 从未转移，则 $L^{final}(p)$ 是初始日志流（根据构造纳入）。如果 p 沿着 $L_0 \to \cdots \to L_k = L^{final}(p)$ 转移，则递归构造会纳入所有 L_i，因此也会纳入 $L^{final}(p)$。

极小性：仅当某个日志流承载分区 P 时（初始时或通过转移），才会将其加入，因此树中的每个节点都对应一个有效的分区位置。

叶节点恰为 $\{L^{final}(p) : p \in P\}$。由于最小集合要求接受任何包含此集合的参与者列表，因此该树满足要求。□

A.2 定理 4.1 的证明

证明。我们通过反证法证明转移原则（定理 4.1）。日志表述和消息表述是等价的，因为两阶段提交消息由已记录状态生成并反映该状态；我们以日志为例进行论证，同样的论证也适用于消息。

第 1 部分：转移前日志必须迁移。反设事务日志 L 在转出日志 T_{out} 之前写入，却未迁移到目标日志流 dst。根据最小集合要求（第 2.2 节），参与者列表必须包含所有最终存放参与并发控制的分区的日志流。由于该分区已转移到 dst，dst 便成为其最终位置。如果 L 未迁移，那么：

- 目标日志流 dst 缺少来自 L 的事务上下文，违反了参与者列表的完备性要求。
- dst 处的事务状态不完整，可能导致错误的提交/中止决策。
- 这违反了原子性，因为事务状态被拆分到了源端和目标日志流中。

因此，在T_{out}之前写入的所有日志，都必须通过转移过程迁移到目标 。对于转移完成前发送的两阶段提交消息，也有相同的要求，因为这些消息对应的状态已（或将）持久化到日志中，必须通过转移过程应用到目标端。

第 2 部分：转移之后的日志必须考虑转移的影响。 反设，在T_{out}之后写入的事务日志L'没有考虑转移的影响。这意味着L'未将目标日志流目标 纳入其参与者列表或事务上下文。由于该分区已转移到目标 （如T_{out}所记录），目标 现在是该分区的最终位置。如果L'忽略了这一点：

- L' 中的参与者列表不完整，缺少目标 ，该目标承载着提交时的分区。
- 这违反了最小集合要求，因为参与者列表未包含所有最终日志流的位置。
- 两阶段提交协议可能无法与目标协调，从而导致不同日志流中的提交/中止决策不一致。

因此，在T_{out}之后写入的所有日志，都必须在其参与者列表中包含目标 （通过中间部分 或增量部分），以考虑转移的影响。对于转移完成后发送的两阶段提交消息，也有相同的要求：这些消息必须发往目标 （或将其纳入其中），因为消息是根据已记录状态生成的，而该状态中的参与者列表必须包含目标。

算法 2 和 3 确保了此方法的正确性，它们保证：（1）转移会迁移所有转移前的事务上下文（日志以及消息中反映的状态）；（2）日志提交会在提交前将中间部分 原子地合并增量部分 ，从而确保日志和后续消息都包含完整的参与者列表。□

A.3 定理 4.3 的证明

证明。我们将证明第4.3节所述的两项要求均得到满足，并且这两项要求可以推出定理 4.1。

要求1：分区迁移必须将转出日志写入前写入的所有事务上下文迁移到目标端。

算法 2 获取传输锁（第 1 行），然后收集事务上下文并阻止受影响事务的日志写入（第 2–9 行），接着在源端和目标端提交转出日志（第 10–11 行），然后将目标 添加到受影响事务的中间部分 中，并解除对这些事务的阻塞（第 12–20 行）。传输机制（该算法是其协调部分）经过设计，确保转出日志写入前写入的所有事务上下文，都会在提交该日志的转移之前或作为该转移的一部分迁移到目标端；转出日志的提交（第 10–11 行）是“转移完成”的线性化点。因此，在该提交之前发送的任何事务日志或两阶段提交消息都属于转移前上下文，并通过转移过程应用到目标端。因此，转移过程满足要求1；算法 2 不会在确保转移前上下文已迁移之前提交转出日志。

需求 2：事务必须在增量部分中包含所有先前的传输目标日志流。

我们利用共享传输锁和这两种算法的原子性。考虑任意事务t以及任意分区迁移：该迁移为平板平板 向目标目标 提交一条时间戳为ts的转出日志，其中平板 参与t，且$ts > t_{。}ts'$（因此迁移在t最后一个相关时间戳之后完成）。当t的两阶段提交日志（通过算法 3）提交时，必须有目标 $\in t_{。}$增量部分（因为已提交的日志包含参与者和增量部分，而根据定理 4.1，目标 必须位于参与者列表中）。

情况 A：转移会提交其转出日志（算法 2，第 10–11 行），然后 t 才运行算法 3 中的合并和提交。接着，算法 2（第 12–20 行）遍历tc，并将目标 添加到$t_{。}$中间部分 中（因为标签页 涉及t 以及$\bar{ts} > t_{。}ts'$）。之后t运行算法 3 时，会将中间部分 合并到增量部分 中并提交（第 6–7 行），因此在两阶段提交日志提交时，目标 $\in t_{。}$增量部分 。

情形 B：当 t 进入算法 3 时，转移尚未提交转出日志。随后，该过程获取锁（第 1 行），并针对 t（若未停止）将中间部分合并到增量部分中，并提交两阶段提交日志（第 4–7 行）。之后转移运行算法 2 时，会提交转出日志，然后将目标添加到 t。中间部分中。因此，此次转移是在之后 t 的两阶段提交日志提交后完成的；该日志中的参与者列表无需包含目标（定理 4.1 要求转出之后写入的日志必须考虑该转移）。因此不存在违规。

情形 C：转移在之后 t 于算法 3 中获取锁、但在之前 t 合并并提交时，提交转出日志。由于转移必须获取同一把锁（算法 2，第 1 行），因此会一直等待，直到该过程释放锁（算法 3，第 10 行）。所以，转移不可能在 t 获取锁和释放锁之间提交。因此，唯一可能的交错是情况 A 或 B，需求 2 成立。

借助共享锁，对于每一对（转移，事务），情况 A 和 B 中恰有一种适用：转移要么在 t 合并之前，将目标添加到 t 的中间部分中；要么在 t 的两阶段提交日志提交之后完成。因此，t 的每次两阶段提交日志提交都会在 t 的增量部分中包含每个转移的目标目标，前提是该转移在此次提交之前完成，且涉及 t 中的平板。

定理 4.1 的满足情况。

转移原则要求：（1）将转出日志产生之前生成的日志和两阶段提交消息，通过转移过程应用到目标；（2）转出日志产生之后生成的日志和消息中包含目标（例如，包含在增量部分中）。（1）由要求1保证。（2）由需求 2，以及已提交的两阶段提交日志包含参与者和增量部分这一事实保证；两阶段提交消息则根据该状态生成。因此，定理 4.1 得以满足。□

A.4 定理 6.1 的证明

证明。在传统扁平两阶段提交模型中，众所周知，所有参与者都会保持相同的结果（全部已提交或全部已中止），并且一旦某个参与者进入提交或中止状态，便会保持该状态。这一安全性属性源于两阶段提交规则的结构：仅当所有参与者投出赞成票时才允许提交，而任何反对票都会强制全局中止。

附录 C 中的 TLA+ 规约对相同的决策逻辑进行了建模。节点仅在收集到其所有子节点的赞成票之后，或在收到其父节点的提交决定之后，才可进入提交状态。类似地，仅当观察到反对票或从其父节点收到中止决定时，节点才可进入中止状态。因此，在子树内无法产生相互冲突的局部决定。

树形协议是同一两阶段提交逻辑的分层实现。每个内部节点首先协调其子树以达成决定，然后作为参与者对其父节点行事。这等价于在扁平两阶段提交执行中将每个已决定的子树折叠为单个参与者。因此，树形协议的每一次执行都对应于传统扁平两阶段提交规范的一次有效执行。

由于扁平两阶段提交协议满足一致性不变量（参与者不会被划分为已提交和已中止两种状态），且树形协议遵循相同的状态转换和消息规则，因此树形两阶段提交协议的安全性属性可直接得出。□

A.5 定理 6.2 的证明

证明。传统两阶段提交协议的活性属性指出，在可靠消息传递和可执行动作满足公平性的条件下，每个参与者最终都会到达终止状态，且所有参与者对相同的结果达成一致。

附录 C 中的规约保留了这些假设。消息被建模为可靠传递，并假定可执行的协议动作最终都会执行。当根节点启动协议时，准备请求会沿树传播。每个节点最终都会收集到来自其子节点的投票，之后提交或中止决定便会成为可执行动作。随后，决定向下传播，确认消息则向上传回。由于系统是有限的，且公平性确保没有可执行动作会一直被推迟，因此每个节点最终都会到达终止决定并完成协议。

与安全性论证一样，每次子树执行都可以视为等效扁平两阶段提交执行中的一个参与者。因此，树形协议的每次公平执行都对应于传统两阶段提交规约的一次公平执行，因而可直接得出终止性。□

B 状态机

注：本附录规定了与附录 C 中 TLA+ 规约一致的核心状态机。正文中描述的优化是在此规约基础上应用的。

- 处理两阶段提交准备请求(p)
  - 启用条件：当状态为运行中且收到来自父参与者 p 的 PrepareReq 请求时，或在根节点发出事务提交请求时。
  - 操作：如果 p 存在且尚未记录父节点，则将 p 记录为父节点；将待处理参与者合并到当前参与者列表中；进入准备状态；将所有参与者的投票初始化为未知；向参与者列表中的所有参与者发送 PrepareReq 消息。
- 处理重复准备请求(p)
  - 启用条件：当状态为准备，且收到来自非记录父节点参与者的准备请求时。
  - 操作：回复状态为同意的准备响应，不更改当前状态。
- 处理孤儿准备请求(p)
  - 启用条件：当状态为中止或墓碑，且收到准备请求时。
  - 操作：回复状态为反对的准备响应。
- 处理两阶段提交准备响应(p,$p\ status$)
  - 启用条件：当状态为准备，且从子参与者 p 收到状态为 $p\ status$ 的准备响应消息时。
  - 操作：将参与者 p 的投票相应地更新为同意或反对。
- 处理两阶段提交提交决定()

- 启用：当状态为准备且所有收集到的投票均表示成功时。

- 操作：合并待处理的参与者；进入提交状态；初始化确认跟踪；如果存在父参与者，则向父参与者发送状态为同意的准备响应；否则，向参与者列表中的所有参与者发送提交消息。

- 处理两阶段提交中止决定()
  - 启用条件：当状态为准备，且至少有一票表示失败时。

  - 操作：合并待处理的参与者；进入中止状态；初始化确认跟踪；如果存在父参与者，则向父参与者发送状态为反对的准备响应；否则，向参与者列表中的所有参与者发送中止消息。

- 处理两阶段提交提交请求(*p*)
  - 启用条件：当状态为运行中或准备，且从父参与者 p 收到提交消息时。
  - 操作：合并待处理的参与者；进入提交状态；向所有参与者发送提交消息；向父参与者发送确认。

- 处理两阶段提交中止请求（p）
  - 启用条件：当状态为运行中或准备状态，且收到来自父参与者 p 的中止消息时。
  - 操作：合并待处理的参与者；进入中止状态；向所有参与者发送中止消息；向父参与者发送确认。

- 处理孤立提交请求（p）
  - 启用条件：当状态为提交状态或墓碑状态，且收到提交消息时。- 操作：回复确认，不改变本地状态。

- 处理孤立中止请求（p）
  - 启用条件：当状态为中止或墓碑，且收到中止消息时。
  - 操作：回复确认，但不更改本地状态。

- 处理两阶段提交确认响应（p）
  - 启用条件：当状态为提交或中止，且收到来自子参与者 p 的确认消息时。
  - 操作：标记已收到参与者 p 的确认。

- 遗忘上下文()
  - 启用条件：当状态为提交或中止，且收到所有参与者的确认时。- 操作：进入 TOMBSTONE 状态。

- 添加中间参与者 _(p)
  - 启用条件：当状态为运行中、准备、提交或中止，且通过转移引入了新的参与者 p 时。
  - 操作：将 p 添加到待处理参与者列表；它将在下一次阶段转换时合并。

C FULLTLA+ 规约

注意：本附录给出了扁平两阶段提交模型的完整 TLA+ 规约（模块 NewTwoPhaseCommit），其中包含事务管理器/资源管理器角色和消息类型；它与附录 B 中的状态机一致。

模块 2*pc tla*

扩展自然数、序列、有限集、*TL*

C 常量 *Node*、*Root*、*InitChildren* 变量 资源管理器状态、子节点、中间子节点、消息、投票、确认、父节点 变量 $\triangleq$ 〈资源管理器状态, 子节点, 中间子节点, 消息, 投票, 确认, 父节点〉状态 $\triangleq$ *{“运行中”, “准备”, “提交”, “中止”, “墓碑”}* 准备请求消息 *(*源, 目标*)* $\triangleq$ [*type* $\mapsto$*“PrepareReq”, src* $\mapsto$ *src, dst* $\mapsto$*dst*]准备响应消息*(*源, 目标, 状态*)* $\triangleq$[*type* $\mapsto$*“PrepareResp”, src* $\mapsto$*src, dst* $\mapsto$*dst, status* $\mapsto$ *status*]提交消息 *(*源, 目标*)* $\triangleq$[*type* $\mapsto$*“Commit”, src* $\mapsto$ *src, dst* $\mapsto$*dst*]中止消息 *(*源, 目标*)* $\triangleq$[*type* $\mapsto$*“Abort”, src* $\mapsto$ *src, dst* $\mapsto$*dst*]确认消息 *(*源, 目标*)* $\triangleq$[*type* $\mapsto$*“Ack”, src* $\mapsto$ *src, dst* $\mapsto$ *dst*]是否为根节点 *(n)* $\triangleq$ *n* = *Root* 合并后的子节点 *(n)* $\triangleq$*children* [*n*] $\cup$ *intermediate children* [*n*]应用合并 *(n, mc)* $\triangleq$$\wedge$*children′* = [*children except !* [*n*] = *mc*]$\wedge$*intermediate children′* = [*intermediate children except !* [*n*] = *{}*]记录父节点 *(n, src)* $\triangleq$ [*parent except !* [*n*] =*if* @ =*“none” then src else* @]

所有投票均为肯定 *(n)* ≜∀ *c* ∈*children* [*n*]*: votes* [*n*][*c*] = *“ok”* 存在否定投票*(n)* ≜∃ *c* ∈ *children* [*n*]*: votes* [*n*][*c*] = *“no”* 全部已确认 *(n)* ≜∀ *c* ∈*children* [*n*]*: acks* [*n*][*c*] = *true* 初始化 ≜∧*rmState* = [*n* ∈ *Node* ↦ *“ RUNNING”*]∧*children* = *InitChildren*∧ *intermediate children* =[*n* ∈ *Node* ↦*{}*]∧ *msgs* = *{}*∧*votes* = [*n* ∈ *Node* ↦[*c* ∈ *children*[*n*] ↦ *“unknown”*]]∧*acks* = [*n* ∈ *Node* ↦[*c* ∈ *children*[*n*] ↦ *false*]]∧*parent* = [*n* ∈ *Node* ↦ *“none”*]根节点开始提交 ≜∧ *rmState*[*Root*] = *“RUNNING”*∧ *let mc* ≜ *MergedChildren (Root)in*∧*rmState′* =[*r mState except !* [*Root*] = *“PREPARE”*]∧ *ApplyMerge (Root, mc)*∧ *votes′* = [*votes except !* [*Root*] =[*c* ∈ *mc* ↦*“unknown”*]]∧ *msgs′* = *msgs* ∪*{MsgPrepareReq (Root, c) : c* ∈ *mc}*∧*unchanged* ⟨*acks, parent*⟩ 处理两阶段提交准备请求 *(n)* ≜∧ *rmState*[*n*] = *“RUNNING”* ∧ ∃*m* ∈*msgs :* ∧ *m.type* = *“PrepareReq”* ∧ *m.dst* = *n*∧ *let mc* ≜ *MergedChildren (n)in*∧ *parent*[*parent except !* [*n*] = *m.src*]∧

rmState′ =[*rmState except !* [*n*] = *“PREPARE”*] ∧ *ApplyMerge (n, mc)*∧ *votes′* = [*votes except !* [*n*] =[*c* ∈ *mc* ↦*“unknown”*]]∧*msgs′* = *msgs* ∪ *{MsgPrepareReq (n, c) : c* ∈ *mc}*∧*unchanged* ⟨ *acks*⟩ 处理两阶段提交重复准备请求 *(n)* ≜∧ *rmState*[*n*] = *“PREPARE”*∧ ∃*m* ∈ *msgs :* ∧ *m.type* = *“PrepareReq”* ∧ *m.dst* = *n*∧ *m.src* ≠ *parent*[*n*]∧ *msgs′* = *msgs* ∪ *{MsgPrepareResp (n, m.src, “ok”)}*

∧保持不变⟨资源管理器状态, 子节点, 中间子节点, 投票, 确认, 父节点⟩ 处理孤立两阶段提交准备请求 *(n)* ≜∧ 资源管理器状态[*n*] ∈ *{*“中止”, “墓碑”*}*∧ ∃*m* ∈ 消息 *:* ∧ *m.type* = “准备请求” ∧ *m.dst* = *n*∧ 父节点′ = 记录父节点 *(n, m.src)*∧ 消息′ = 消息 ∪ *{*中止消息 *(n, m.src, “*反对*”)}*∧保持不变⟨资源管理器状态, 子节点, 中间子节点, 投票, 确认⟩ 处理两阶段提交准备响应 *(n)* ≜∧ 资源管理器状态[*n*] = “准备”∧ ∃*m* ∈ 消息 *:* ∧ *m.type* = “ 准备响应” ∧ *m.dst* = *n*∧ ∈ [ ] *m.src* 子节点 *n* 投票 [*n*]∧[*m src*] = “未知”∧投票′ = [投票 *except !* [*n*][*m.src*] = *m.status*]∧保持不变⟨资源管理器状态, 子节点, 中间子节点, 消息, 确认, 父节点 ⟩ 处理两阶段提交已决定提交 *(n)* ≜∧ 资源管理器状态[*n*] = “准备”∧ 所有投票均为肯定 *(n)*∧ 如果 是否为根节点 *(n) then let mc* ≜ 合并后的子节点 *(n)in*∧资源管理器状态′ =[资源管理器状态 *except !* [*n*] = “提交”]∧ 应用合并 *(n, mc)*∧ 确认′ = [确认 *except !* [*n*] =[*c* ∈ *mc* ↦*false*]]∧消息′ = 消息 ∪*{*提交消息 *(n, c) : c* ∈ *mc}*∧保持不变⟨ 投票, 父节点⟩ *else* ∧消息′ = 消息 ∪*{*准备响应消息*(n, parent* [*n*]*, “*同意*”)}*∧保持不变⟨资源管理器状态, 子节点, 中间子节点, 投票, 确认, 父节点⟩ 处理两阶段提交中止决定 *(n)* ≜∧ ∨ 资源管理器状态[*n*] = “准备”∨ *(*是否为根节点 *(n)* ∧ 资源管理器状态[*n*] = “运行中*”)*∧ 存在否定投票 *(n)*∧ 如果 是否为根节点 *(n) then let mc* ≜ 合并后的子节点 *(n)in*∧资源管理器状态′ =[资源管理器状态 *except !* [*n*] = “中止”]∧ 应用合并 *(n, mc)*∧ 确认′ = [确认 *except !* [*n*] =

$[c \in mc \mapsto false]$∧消息′ = 消息 ∪{中止消息 $(n, c) : c \in mc$}∧保持不变〈投票, 父节点〉 *else* ∧ 消息′ = 消息 ∪{准备响应消息$(n, parent\ [n], \text{“反对”})$}∧保持不变〈资源管理器状态, 子节点, 中间子节点, 投票, 确认, 父节点〉 处理两阶段提交提交请求 $(n) \triangleq$∧ ¬是否为根节点 (n)∧ 资源管理器状态 $[n] \in \{\text{“运行中”}, \text{“准备”}\}$∧ $\exists m \in$ 消息 $: \wedge m.type = \text{“提交”} \wedge m.dst = n \wedge$ *let* $mc \triangleq$ 合并后的子节点 $(n) in \wedge$ 父节点′ = 记录父节点 $(n, m.src)$∧资源管理器状态′ =[资源管理器状态 *except !* $[n] = \text{“提交”}]$∧ 应用合并 (n, mc)∧ 确认′ = [确认 *except !* $[n] =[c \in mc \mapsto false]]$∧消息′ = 消息 ∪{提交消息 $(n, c) : c \in mc\} \cup \{$确认消息 $(n, m.src)\}$∧保持不变〈投票〉 处理两阶段提交中止请求 $(n) \triangleq$∧ ¬ 是否为根节点 (n)∧ 资源管理器状态 $[n] \in \{\text{“运行中”}, \text{“准备”}\}$∧ $\exists m \in$ 消息 $: \wedge m.type = \text{“中止”} \wedge m.dst = n \wedge$ *let* $mc \triangleq$ 合并后的子节点 $(n) in \wedge$ 父节点′ = 记录父节点 $(n, m.src)$∧资源管理器状态′ =[资源管理器状态 *except !* $[n] = \text{“中止”}]$∧ 应用合并 (n, mc)∧ 确认′ = [确认 *except !* $[n] =[c \in mc \mapsto false]]$∧消息′ = 消息 ∪{中止消息 $(n, c) : c \in mc\} \cup \{$确认消息 $(n, m.src)\}$∧保持不变〈投票〉 处理孤立两阶段提交提交请求 $(n) \triangleq$∧ ¬是否为根节点 (n)∧ 资源管理器状态 $[n] \in \{\text{“提交”}, \text{“墓碑”}\}$∧ $\exists m \in$ 消息 $: \wedge m.type = \text{“提交”} \wedge m.dst = n \wedge$ 消息′ = 消息 ∪{确认消息 $(n, m.src)\}$∧保持不变〈资源管理器状态, 子节点, 中间子节点, 投票, 确认, 父节点〉

处理孤立两阶段提交中止请求 $(n) \triangleq$∧ ¬是否为根节点 (n)∧ 资源管理器状态 $[n] \in \{\text{“中止”}, \text{“墓碑”}\}$∧ $\exists m \in$ 消息 $: \wedge m.type = \text{“中止”} \wedge m.dst = n \wedge$ 消息′ = 消息 ∪{中止消息 $(n, m.src)\}$∧保持不变〈资源管理器状态, 子节点, 中间子节点, 投票, 确认, 父节点〉 内部中止 $(n) \triangleq \wedge$ 资源管理器状态 $[n] = \text{“运行中”} \wedge$ 令 $mc \triangleq$ 合并后的子节点 $(n)\ parentNotify \triangleq$如果 $parent[n] \neq \text{“none”}$ 则 $\{$准备响应消息$(n, parent\ [n], \text{“反对”})\}$ 否则 $\{\}\ in$∧资源管理器状态′ =[资源管理器状态 *except !* $[n] = \text{“中止”}] \wedge$ 应用合并 (n, mc)∧ 确认′ = [确认 *except !* $[n] =[c \in mc \mapsto false]]$∧消息′ = 消息 ∪{中止消息 $(n, c) : c \in mc\} \cup parentNotify$∧保持不变〈投票, 父节点〉 处理两阶段提交确认响应 $(n) \triangleq \wedge$ 资源管理器状态 $[n] \in \{\text{“提交”}, \text{“中止”}\} \wedge \exists m \in$ 消息 $: \wedge m.type = \text{“确认”} \wedge m.dst = n \wedge m.src \in$ 子节点 n 确认 $[n][m.src] = false \wedge$确认′ =[确认 *except !* $[n][m.src] = true] \wedge$ 保持不变〈资源管理器状态, 子节点, 中间子节点, 消息, 投票, 父节点〉 忘记上下文 $(n) \triangleq \wedge$ 资源管理器状态 $[n] \in \{\text{“提交”}, \text{“中止”}\} \wedge$ 全部已确认 (n)∧资源管理器状态′ =[资源管理器状态 *except !* $[n] = \text{“墓碑”}] \wedge$保持不变〈子节点, 中间子节点, 消息, 投票, 确认, 父节点〉添加中间参与者 $(n, newChild) \triangleq \wedge$ 资源管理器状态 $[n] \in \{\text{“运行中”}, \text{“准备”}, \text{“提交”}, \text{“中止”}\} \wedge newChild \in$ 节点 $\setminus \{n\} \wedge newChild \notin$ 子节点 $[n] \wedge newChild \notin$ 中间子节点 $[n]$

∧中间子节点′ =中间子节点 *except !*[*n*] = @ ∪ *{newChild}*]∧保持不变⟨资源管理器状态, 子节点, 消息, 投票, 确认, 父节点⟩下一步 $\triangleq$∨ 根节点开始提交∨ ∃*n* ∈ 节点 *:* 处理两阶段提交准备请求 *(n)*∨ ∃*n* ∈ 节点 *:* 处理两阶段提交重复准备请求 *(n)*∨ ∃*n* ∈ 节点 *:* 处理孤立两阶段提交准备请求 *(n)*∨ ∃*n* ∈ 节点 *:* 处理两阶段提交准备响应 *(n)*∨ ∃*n* ∈ 节点 *:* 处理两阶段提交已决定提交 *(n)*∨ ∃*n* ∈ 节点 *:* 处理两阶段提交中止决定 *(n)*∨ ∃*n* ∈ 节点 *:* 处理两阶段提交提交请求 *(n)*

∨ ∃*n* ∈ 节点：处理2*pcAbortRequest* (*n*)∨ ∃*n* ∈ 节点：处理孤儿2*pcCommitRequest* (*n*)∨ ∃*n* ∈ 节点：处理孤儿2*pcAbortRequest* (*n*)∨ ∃*n* ∈ 节点：内部中止 (*n*)∨ ∃*n* ∈ 节点：处理2*pcAckResponse* (*n*)∨ ∃*n* ∈ 节点：忘记上下文 (*n*)∨ ∃*n, newChild* ∈ 节点：添加中间参与者(*n, newChild*)

规约 $\triangleq$ 初始化 ∧ □[*Next*]$_{Vars}$

一致性 $\triangleq$∀*n*1, *n*2 ∈ 节点 : ¬(资源管理器状态[*n*1] = “提交”∧ 资源管理器状态[*n*2] = “中止”)