Linearizability vs Serializability 以及分布式事务

Original Accela推箱子 Accela推箱子 2023-01-10 00:02 Posted on 上海

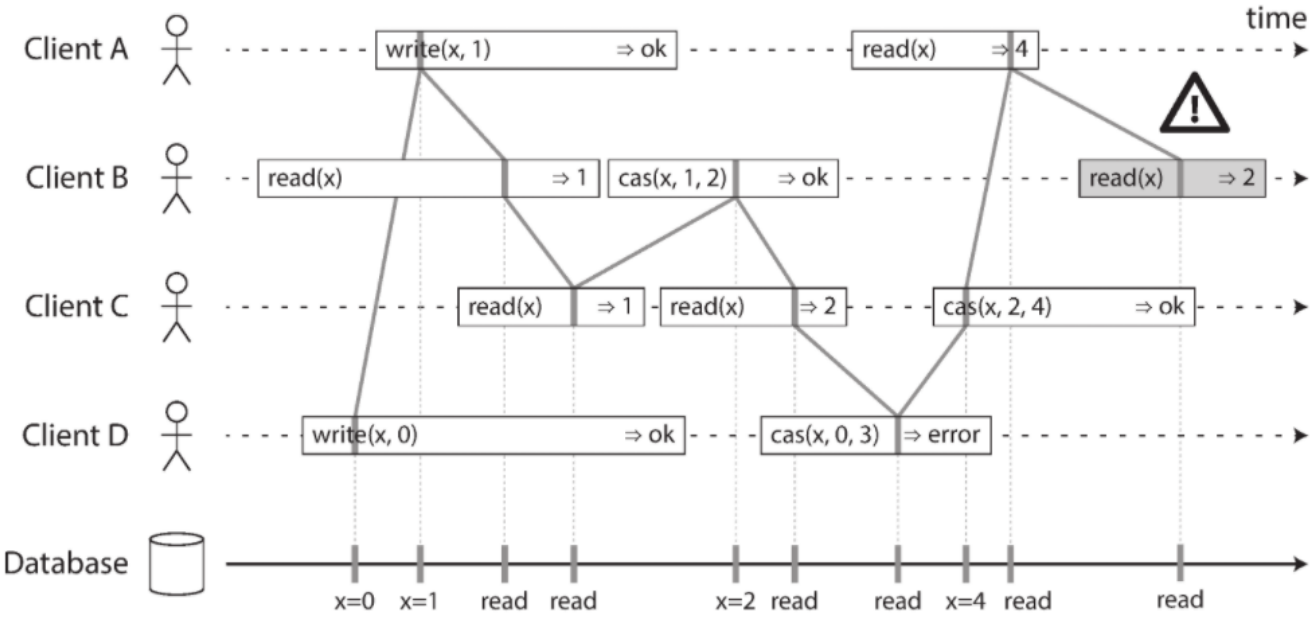
Linearizability vs Serializability 是分布式存储系统的基本概念。虽然容易混淆，但解释它们的文章不多。

前文：分布式系统-分布式事务（P3完）

基本概念

Linearizability

Linearizability（也称作atomic consistency, strong consistency, immediate consistency）描述单个对象（存储单个值）的读取和写入。它不涉及多个对象，不涉及组合多个对象的"事务"。它将每个操作视为原子的，在单个时间点生效，没有时间跨度。



(违反Linearizability的示例，图片来自*Designing Data-Intensive Applications* [1] - 328页)

Linearizability支持两种操作：读，写一个值。如果还支持CAS（compare-and-set），它会更有用（稍后会看到）。

Linearizability保证Recency 。 一旦写入新值，该值立即生效，所有读立即见到新值。读总是见到最新的直。Linearizability中的操作总是全序的（total ordered），没有并发操作。

Serializability

Serializability是数据库事务（Transaction，Tx）的隔离级别。它来自数据库社区，与 Linearizability 起源不同。

Serializability描述多个事务。一个事务通常操作多个对象。

数据库可以并行执行事务（以及并行操作）。Serializability保证结果与按某种串行顺序（serial order）执行事务相同。

Serializability并不能保证上述"某种顺序"符合Recency。该顺序可能与事务执行时间不同。例如，Tx1 比 Tx2 开始得早，但结果表现得好像 Tx2 在 Tx1 之前执行。换言之，为满足 Serializability，有不止一种可能的执行调度。

现实世界的数据库

Serializability与事务时间戳

在数据库中，Serializability通常由事务时间戳来反映。事务虽然是并行执行的，但结果就像它们按时间戳的串行顺序执行一样。

Serializability要求事务符合全序（total order），事务仿佛在同一时间戳（提交时间戳）执行其读写。但是在快照隔离（Snapshot Isolation）中，允许事务拥有不同的读、写时间戳。（在 *Cockroach 论文* [2] - 3.3接，或*这篇文章*[3]中有说明。）

Linearizability与事务

现代数据库通常支持MVCC和快照隔离。读指向特定版本，而不是最新值。然而 Linearizability总是读最新值（Recency）。因而，Linearizability在事务中用处不大。

但"Recency"仍然是有用概念。Spanner 创造了**外部一致性**（External Consistency）概念。它将"Recency"结合到Serializability中（尽管 Spanner 论文直接将其称为"Linearizability"）。

前文提到，Serializability允许不符实际的串行顺序，例如较早的 Tx1 表现得好像比 Tx2 晚执行，这正是外部一致性想要修复的情况，见下表。节点时钟不一致，Tx1 比 Tx2 执行得早，但 Tx1 获得了一个较晚的（更大的）提交时间戳（commit timestamp）。特别是在节

点 C 上，Tx2 写值晚于 Tx1，但 Tx2 设置的时间戳更早（更小）。这是异常(Abnormal)的情形。

	Tx1 on A,B,C	Tx2 on C,D,E
Node A clock	120	
Node B clock	80	
Node C clock	50	60
Node D clock		80
Node E clock		90
Commit timestamp	120	90
Realworld clock	80	90

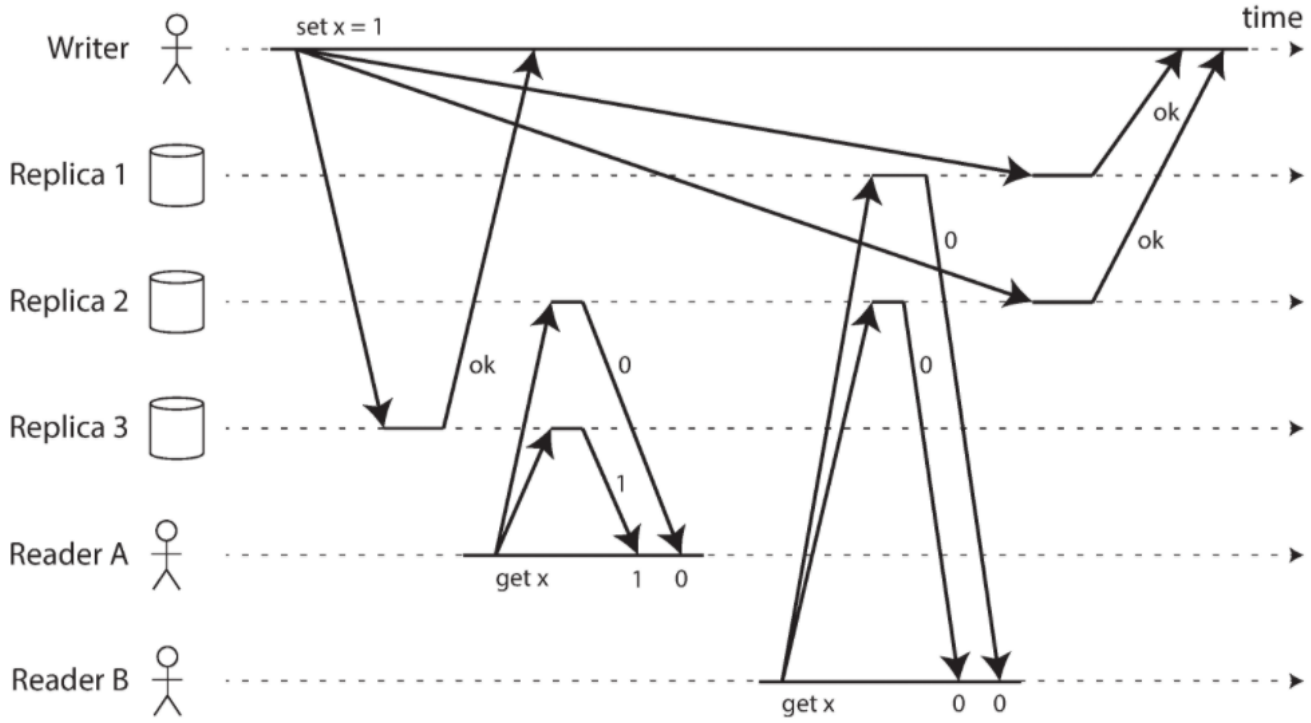
（违反外部一致性的示例）

Spanner 通过 **Commit Wait** 修复上述问题。您可以在*文章1*[4]、*文章2*[5]中看到更多内容。

Linearizability与Quorum读

Dynamo[6]风格的Quorum写/读是实现“*强*[7]”一致性的一种巧妙方法。

但是，见下表（N=3，W=3，R=2）。网络有不同的延迟，客户端 A 仅部分地看到新值 1，但客户端 B 只看到旧值 0。Linearizability被部分地违反了。



（Quorum读违反Linearizability的示例，图片来自*Designing Data-Intensive Applications* [1] - 334 页）

副本节点有不准确的时钟和并发写入时，会变得更加复杂。

全序广播（Total Order Broadcast）和共识（Consensus）

全序广播和共识与Linearizability 密切相关。稍后我们将看到"Linearizability+CAS" <=> 全序广播 <=> 共识，它们是等价的。

全序广播

全序广播（也称为原子广播）是一种跨节点交换消息的协议， 它满足

- **可靠送达**（Reliability delivery）：如果消息被送达到一个节点，那么它最终会被送达到所有节点。如果一个节点收到消息，那么所有节点最终都会收到消息。或者相反，所有节点都没有收到消息。你可以在这里看到 all or nothing 或原子性。消息不需要立即送达。

- **全序送达**（Total ordered delivery）：如果一个节点先收到Msg1，然后再收到Msg2，那么任何其他节点也必须先收到Msg1，再收到Msg2。消息送达是全序的，并且每个节点都以相同的顺序送达。此外，送达顺序是固定的，不允许任何节点在中间插入消息。

- **Uniform Integrity**：一条消息最多被每个节点接收一次，必须发生在广播后。（此属性在*Wiki*[8]中有所提及，但不知为何不在*Designing Data-Intensive Applications* [1] - 348 页中。）

您会发现全序广播就像一个在所有节点之间达成共识的日志。消息按 LSN（Log Sequence Number）排序。*CORFU*[9]是一个构建分布式共享日志的项目。

共识

共识通常被描述为多个节点提议（Propose），节点最终同意相同的提议值，共识算法满足：

- **Uniform agreement**：每个节点最终都同意相同的值。
- **完整性**（Integrity）：如果一个节点决定了值 v，它必须在未来总是决定相同的值。
- **终止**（Termination）：每个节点（如果没有崩溃）最终决定一某个值。

最著名的共识算法是*Viewstamped Replication*[10] (VSR)、*Paxos*[11]、*Raft*[12] 和 *Zab*[13] (ZooKeeper)

全序广播可以看作是多轮共识。在每一轮中，节点提出下一条要发送的消息，然后决定并同意下一条按照全序送达的消息。此外，Paxos 实现通常依赖于日志，日志可以看作是在所有节点之间的全序广播。

虽然 2PC（两阶段提交）是一种流行的算法，但它不满足"终止"属性。一个典型的解决方法是将每个参与者和协调者实现为一个高可用（High Availability）的 Paxos Quorum。见*Spanner*[14]。

等价于Linearizability

从上面可以看出共识和全序广播是等价的，我们可以用一个来实现另一个。例如，Paxos 本质上是广播有序的日志消息。全序广播，在每一轮消息中，可以使用Paxos来同意消息广播顺序和exactly once。

此外，全序广播 和共识都等价于Linearizability。

- **Linearizability可以通过全序广播实现。**想象一下共享日志，写被插入到日志中，以相同的顺序广播到所有节点。读永远不会对历史产生异议。为了确保读取的Recency，我们可以将读也插入到日志中，其中 LSN 是执行读的时间点（这就是在 Paxos 中实现 *Linearizable Quorum Reads* [15]的方法）。或者，为了确保读取的Recency，我们可以在读取之前调用 sync()，以确保节点已更新。或者，我们只使用同步消息复制。Linearizability CAS很容易实现，因为读总是在每个节点上看到相同的值。

- **全序广播可以通过 Linearizability和CAS 来实现。**假设每条消息都有一个递增的 LSN，新消息到达后，节点CAS比较最后消息的LSN+1与新消息的 LSN，这样只能接受下一条消息（LSN+1）。这样，每个节点都以相同的全序接收每条消息。可以重新发送消息，通过验证LSN，落后的节点可以接收下一条消息，并且可以检测并丢弃重复的消息。

- **Linearizability可以通过共识来实现。**这与通过全序广播实现 Linearizability一样。共识可以通过Linearizability来实现，Recency是每个节点都对写入的新值达成共识。

更多的等价

除了上述等价链，Linearizability+CAS <=> 全序广播 <=> 共识，还有更多与它们等价的东西。这些等价性是对分布式系统最深刻和令人惊讶的见解之一。（设计数据密集型应用程序一书，第 374 页。）

分布式锁（Distributed Lock）

- **分布式锁可以通过共识来实现。**锁需要Linearizability：每个节点必须就谁拥有锁达成一致，这就是共识所保证的。此外，分布式锁通常以租约（Lease）的形式实现，当锁的拥有者崩溃后，租约到期。

- **共识可以通过分布式锁来实现。**拥有锁的节点就是Leader，追随者只是同意Leader所说的一切。Leader选择是 etcd 或 ZooKeeper 提供的一项有用的服务，来简化自己实现 Paxos 的复杂性。

唯一性约束（Uniqueness Constraint）

- **唯一性约束可以通过分布式锁来实现**。唯一性约束的典型用例是避免创建重复的用户名。显然，只需写对象上锁。或者，Linearizability CAS也可以用来实现唯一性约束。

- **分布式锁可以通过唯一性约束来实现**。上锁是对锁的唯一所有权，唯一性约束确保所有者是唯一的。

原子事务提交（Atomic Transaction Commit）

- **分布式事务可以通过所有节点之间达成共识的共享日志来实现**。共享日志是全序广播。原子事务提交是指最终每个节点就是否提交或中止分布式事务达成一致，即共识。

- **全序广播可以通过分布式事务来实现**。每轮消息都是一个事务提交。分布式事务是一种更强大的语义，它可以承载任何你需要的东西。原子事务提交可看作全序广播中的接受的消息。

这些等价性构成了实现分布式事务的基础，我们将在下一节中看到更多内容。

进入分布式事务（Distributed Transactions）

现代数据库通常将分区（Partition）分布在多个节点，甚至跨地理区域。跨多个分区进行协调的分布式事务具有挑战性。典型的工业实现来自于 2PC：*Percolator*[16]、*Spanner*[14]、*CockroachDB*[17]、*TiDB*[18]。

通过将每个 2PC 参与者（表分区）作为高可用的 Paxos Quorum 运行，可以解决2PC 的Liveness问题。协调者（Coordinator）可以运行在其中一个参与者Quorum上以实现高可用（*Spanner*[14]），或者协调者故障导致事务中止（*CockroachDB*[17]），或者协调者运行在单独的高可用服务上（*TiDB*[18]）上。

2PC可以更灵活.. 其实.. 让我们思考一系列的问题。

首先，事务操作（读、写）如何强制它们的顺序?

- **方法一，上帝节点（god node）**：上帝节点知道每一笔事务，统一调度每笔事务的每个操作，达到想要的隔离级别（Isolation Level）。分区节点盲目执行预定的操作，不需要协调。上帝节点可以使用 Paxos 进行复制或备份，以实现高可用。如果事务绝对不重

叠，上帝节点也许可以通过分区进行横向扩展。总之，分布式事务问题简化为单节点事务，不需要2PC算法。真实系统包括 *OceanBase V0.1*[19]（或者也包括 V2.0？）和 *Calvin DB*[20]。

- **方法二，去中心化**：没有上帝节点，每笔事务自己安排操作。它们利用时间戳、读/写意图（Intents）或锁。这里我都称之为**锁**，它简化了之后的讨论。因为这里的锁是事务了解其他事务，并确定等待或中止/重试的通用方法。事务实现通常将上锁和写暂存数据合并为一个步骤，即写意图（Write Intent)。2PC用方法二工作。

接下来，事务如何确定冲突？什么被视为冲突？

- **只对写上锁**：只有重叠的写集（Write Set）才被视为冲突。读从不阻塞，也不需要上锁。这通常是快照隔离的实现方式。快照隔离存在*Write Skew*[21]问题，由于没有上锁读集（Read Set），事务不知道其前提已更改。具有Serializable隔离级别的数据库也可以提供"*快照读*[14]（Snapshot Read）"，不需要上锁。

- **对读和写都上锁**：这通常是Serializability隔离级别的实现方式。该机制也称为 2PL（两阶段上锁。请注意，它与 2PC 无关，请参见*此处*[22]）。上锁所有访问，保证安全，但性能不佳。可以使用*Predicate Locks*[23]或*Index-range Locks*[1]使上锁更细粒度。

- **Serializable Snapshot Isolation (SSI)**：这是一种较新的方法，在 *PostgreSQL*[24] 中实现。SSI 跟踪事务中的读/写依赖关系。在事务提交（commit）之前，它会检查是否违反了隔离级别。例如。另一个 Tx 已提交；通过追溯读写依赖关系，我们发现受害事务的读写集发生了变化，那么我们知道受害事务需要中止并重试。事务执行是非阻塞（Non-blocking）的，虽然该算法需要事务获取"SIREAD 锁"，用于跟踪读/写依赖性。（所以看到"上锁"并不一定意味着阻塞，它只是措辞。）

- **Serial Safe Net (SSN)**：*论文*[25]、*幻灯片*[26]、*评论*[27]。SI+SSN是实现Serializability的常用方式。SSN 与 SSI 类似，它拒绝提交可能构成依赖循环的事务。与 SSI 相比，SSN 进行了*改进*[28]，它拒绝更少的误报（false positives）并提供更高的并发性。AWS Redshift [28]通过更早地evaluating certification来减少CPU/内存消耗，从而对 SSN 进行了更多改进。

上面说的锁是泛化的。它可以是文章中常见的锁（例如*MySQL InnoDB*[29]），或读/写意图（例如 *CockroachDB*[17]），或只是一个时间戳（例如 *Percolator*[16]），或一个读/写依赖跟踪器（SSI 算法）。与其说它们是锁，不如说它们是事务之间的"**共享点**"，事务使用这些"共享点"相互交互，并确定其读写操作的顺序。

（进一步..如果锁是由时间戳实现的，代码如何原子地比较时间戳并提交，或者实现阻塞行为？首先，请参阅Latch vs Locks[30]。Latch与锁的区别来自数据库社区，其中锁用于事务，Latch用于数据库内部代码，两者都是控制并发的。在这段代码中，cmtlock.Lock(); compare timestamp; do commit; cmtlock.Unlock()，cmtlock是Latch，而这里的锁是时间戳。也可以说，这是无锁的OCC事务协议（使用Latch）。所以要回答最初的问题，在数据库内部代码中，可以用Latch实现原子操作和阻塞行为。此外，B-tree也是一个使用很多Latch的地方。）

接下来，当一个事务遇到另一个Tx持有的锁时，等待还是不等待？

- **应该等待锁**：这是典型的悲观上锁实现，例如快照隔离（只锁写），或者Serializable隔离（2PL）。

- **不要等待锁，较新的事务重试**: 如果不等待锁，两个竞争事务之一必须中止并重试（或者更巧妙的做法是，"半重试"，例如 CockroachDB 中的"*Read Refreshes*[17]"），否则违反隔离级别。这通常会导致 OCC（Optimistic Concurrency Control，乐观并发控制）上锁实现。以写-写冲突（write-write conflict）时的快照隔离为例，重叠写集上的提交时间戳可以看作是广义的锁。当旧事务提交时，时间戳递进。稍后当较新事务看到时间戳高于它记忆的时间时，它知道写集在中间被更改，较新事务中止并重试。（这篇*论文*[31]说明了 MVOCC。）Spanner *wound-wait*[32]是另一个例子，当较旧的事务请求锁时，"较旧"的事务将"伤害"（中止）持有锁的较新的事务。

- **不要等待锁，较旧的事务重试**：示例是 *Percolator*[16]。可以看到读操作 (Get(c)) 将清除正在进行的写事务持有的锁 (cleanupStaleLock(k, ts))。这些正在进行的写事务是这里的"旧"事务。它们被中止，因为锁被较新的事务的读清除。总是让较新的事务中止较旧的事务可能过于激进。实际上在 *Percolator 论文*[33]中，BackoffAndMaybeCleanup处有更多的*细节*[34]。较新的事务将使用 Chubby 令牌来确定较旧的事务是否仍然存活，然后再选择中止它。

实际数据库可能会结合多种等待/不等待策略。例如，在 Spanner *wound-wait*[32]中，较新的事务选择阻塞等待锁持有者（较旧的事务），而较旧的事务立即中止锁持有者（较新的事务）。例如，*CockroachDB*[17]处理写-写冲突也结合了等待较旧事务，中止较新事务的行为。例如，*TiDB*[18]实现了 Percolator 算法的 Optimistic 和 Pessimistic 版本。

接下来，分布式事务如何判断提交完成呢？我们知道信息分散在许多节点上。

- **提交的单点（Single point of commit）**：Percolator 在事务要写的所有记录中选择一条作为主（primary）记录。尽管所有记录都获取锁，但主记录上的锁充当单点同步：1) 每个辅助（secondary）记录锁都指向主记录。2) 如果主记录被上锁，事务可以在不检查任何辅助记录上的锁的情况下提交。3) 如果主记录被提交，则整个事务被视为已提交，而辅助记录的写意图可以异步生效（由下一个读者执行）。4) 当有人试图中止事务时，它首先清除主记录的锁，这会立即中止整个事务。BigTable 仅提供记录级（即行级）CAS（原子 *ReadModifyWriteRow*[35]）。巧妙的是，以主记录作为单点提交，Percolator 在一个记录级别的CAS中同时完成：检查主记录锁（存储在同一个主记录行中），和提交写意图（在主记录行中添加一个指针指向写意图）。

- **协调者等待所有 2PC 参与者完成复制**：前文提到 *Spanner*、*CockroachDB*、*TiDB* 使用 2PC 分布式事务。每个参与者都在 Paxos Quorum 上进行复制以实现高可用。为了事务完成提交，协调者向每个参与者发送请求并等待Quorum完成复制。**提交的单点**发生于协调者确认，或标记事务为COMMITTED。注意到，当两个竞争 Tx1 和 Tx2 向同一组参与者发送写意图（即上锁）时，如果上锁顺序交错，则 Tx1 和 Tx2 可能会死锁。Spanner使用 *wound-wait* 来避免死锁。*CockroachDB* 使用分布式死锁检测来中止等待循环。

- **每个 2PC 参与者实际完成异步复制**：这是 CockroachDB 中使用的*Parallel Commit*[36]。OceanBase的一篇*文章*[37]在2016年也提到了类似的想法。类似于".. 等待所有2PC参与者 .."，事务协调者向每个参与者发送请求。不同的是，协调者立即向用户客户端确认提交完成（STAGING状态）。现在事务最终可能成功（COMMITTED 状态）或中止，这取决于是否"每个 2PC 参与者实际完成复制"。结果是异步的，**没有提交的单点**。后续事务需要运行评估阶段（称为"事务状态恢复过程"），轮询每个参与者，以确定前一个事务是否真正完成复制（即成功提交）。为了避免代价高昂的评估阶段，协调者也积极评估并将以前的事务移至已提交状态（COMMITTED 状态）。注意到，Parallel Commit 还避免了因协调者故障而导致2PC失败，因为现在提交完成条件已转移给参与者。

最后一个问题是，我们应该如何在分布式数据库中存储事务记录（transaction records）？事务有状态要保存，例如 COMMITTED、PENDING、ABORTED， 以及其他元数据。它们通常称为事务记录。

- **中央事务表**：数据库可以使用中央事务表来存储所有事务记录。这通过将元数据管理与事务处理分开来简化。中央事务表可以通过 Paxos 复制实现高可用，通过分区进行横向扩展。它可以只是另一个普通表，以利用分布式数据库已经提供的分区和复制。

- **没有中央事务表**：分布式数据库想要分布一切。*Percolator* 使用主记录作为提交的单点，它甚至不需要事务记录。*CockroachDB*[38] 将事务记录放在事务中第一个键的同一分区中，所有的写意图都指向这个事务记录。

- **在客户端**：可以在用户客户端跟踪事务状态，它的可用性不高。潜在的假设是，如果客户端崩溃，事务无论如何都会中止，所以也不再需要事务状态。

长话短说，图表中总结了以上所有内容。策略的谱系可以帮助理解和设计分布式事务。有一些有趣的可能性

- 2PC 和 Paxos 都可以用来达成共识。分布式事务目前使用2PC来提交（而Paxos只是为了表分区的高可用）。**是否可以使用 Paxos 来提交事务**？例如，协调者只需要向至少 N/2+1 个参与者发送提交请求，而不是所有参与者，这减少了尾部延迟（tail latency）。例如，如果 A 显示高延迟，对分区A分区的写可以先临时缓存在B。

- 上帝节点大大简化了分布式事务的复杂性。由于跨节点交互较少，它还应该具有性能优势。**上帝节点可以通过分区进行扩容吗**？鉴于我们确信两笔事务永远不会重叠，这应该是可行的。

Components of a distributed trnasaction	Strategies	Examples
How to enforce ordering of reads/writes?	God node masters all transaction reads & writes	Calvin DB (SIGMOD12), OceanBase V0.1 (or also V2.0?)
	Each transaction arrange by themselves (by timestamps, read/write intents, locks)	Percolator, Spanner, CockroachDB, TiDB
How to determine what is a conflict?	Only write-write conflict. Lock write only	Snapshot Isolation
	Both ww-conflict, rw/wr-conflict. Lock both read and write	Serializable Isolation. 2PL.
	Serializable Snapshot Isolation (SSI) detects “dangerous structure”. SIREAD lock to track read/write dependencies	PostgreSQL SSI
Should it wait on Lock (timestamp mismatch, read/write intents, lock)?	Wait on Lock. Pessimistic locking implementation.	Spanner, CockroachDB serializable read-write transaction
	Don’t wait on Lock. Younger transaction abort & retry	OCC. Spanner wound-wait
	Don’t wait on Lock. Older transaction abort & retry	OCC. Percolator clean stale locks.
How to determine commit is finished?	Single point of commit. Percolator primary record	Percolator primary record
	Single point of commit. Coordinator ack back after waited for all participants finish replication. Transaction record marked COMMITTED	Spanner, TiDB
	No single point of commit. Need to examine each participant actually finished replication. Coordinator ack back very earlier.	CockroachDB Parallel Commit
Where to store transaction records	A central transaction table	Didn’t find example
	No central transaction table. Transaction record resides at one partition written by the transaction	Percolator, CockroachDB
	At client-side. (If client crashes, transaction is aborted, transaction record is safe to lose.)	Didn’t find example

Note: The "Lock" here is not the common programming "lock". Lock is the point where concurrent transactions exchange info. A Lock can be waited or not waited. A Lock can be a programming lock, or just a timestamp. Actually, DB literatures use "latch" wording for common programming locks, while "lock" wording for transaction concurrency.

(分布式事务策略谱系)

引用

[1] Designing Data-Intensive Applications: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[2] Cockroach论文: https://dl.acm.org/doi/pdf/10.1145/3318464.3386134

[3] Cockroach Pipelining consensus writes: https://www.cockroachlabs.com/blog/transaction-pipelining/

[4] 解释Spanner的TrueTime: https://zhuanlan.zhihu.com/p/44254954

[5] 理解数据库的外部一致性:
https://www.zhihu.com/question/56073588/answer/519284998

[6] Dynamo paper: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[7] DynamoDB Read consistency:
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

[8] Wiki Atomic broadcast: https://en.wikipedia.org/wiki/Atomic_broadcast

[9] CORFU: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final30.pdf

[10] Viewstamped Replication: http://www.pmg.lcs.mit.edu/papers/vr-revisited.pdf

[11] Paxos: https://en.wikipedia.org/wiki/Paxos_(computer_science)

[12] Raft: https://raft.github.io/

[13] Zab: https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab+vs.+Paxos

[14] Spanner：
https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf

[15] Linearizable Quorum Reads:
http://muratbuffalo.blogspot.com/2019/09/linearizable-quorum-reads-in-paxos.html

[16] Percolator: https://github.com/pingcap/tla-plus/blob/master/Percolator/Percolator.tla

[17] CockroachDB: https://dl.acm.org/doi/pdf/10.1145/3318464.3386134

[18] TiDB: http://www.vldb.org/pvldb/vol13/p3072-huang.pdf

[19] OceanBase V0.1: https://zhuanlan.zhihu.com/p/93721603

[20] Calvin DB: http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf

[21] Write skew: https://arxiv.org/pdf/1208.4179.pdf

[22] 2PL vs 2PC: https://www.quora.com/Is-there-a-difference-between-Two-Phase-Locking-and-Two-Phase-Commit

[23] Predicate locks: http://www.scs.stanford.edu/nyu/02fa/notes/l8.pdf#3

[24] PostgreSQL SSI: https://arxiv.org/pdf/1208.4179.pdf

[25] SSN paper: https://arxiv.org/pdf/1605.04292.pdf

[26] SSN slides: https://event.cwi.nl/damon2015/slides/slides-wang.pptx

[27] SSN review: https://zhuanlan.zhihu.com/p/524580964

[28] AWS Redshift SSI: https://assets.amazon.science/93/e0/a347021a4c6fbbccd5a056580d00/sigmod22-redshift-reinvented.pdf

[29] MySQL InnoDB: https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html

[30] Latch vs Locks: https://stackoverflow.com/questions/3111403/what-is-the-difference-between-a-lock-and-a-latch-in-the-context-of-concurrent-a

[31] MVOCC paper: http://www.vldb.org/pvldb/vol10/p781-Wu.pdf

[32] Spanner wound-wait: https://cloud.google.com/spanner/docs/whitepapers/life-of-reads-and-writes

[33] Percolator paper: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36726.pdf

[34] Percolator BackoffAndMaybeCleanup： https://www.jianshu.com/p/f8dcd85fd675

[35] BigTable ReadModifyWriteRow: https://cloud.google.com/bigtable/docs/writes

[36] CockroachDB Parallel Commit：https://www.cockroachlabs.com/blog/parallel-commits/

[37] OceanBase 2PC工程实践: http://oceanbase.org.cn/?p=195

[38] CockroachDB transaction records: https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer.html#transaction-records

封面图片 Image by rawpixel.com on Freepik: https://www.freepik.com/free-vector/atom-science-education-icon-vector-neon-digital-graphic_16311773.htm#query=atom&position=26&from_view=search&track=sph