13 | 隔离性：为什么使用乐观协议的分布式数据库越来越少?

2020-09-07 王磊

分布式数据库30讲 进入课程 >

讲述：王磊
时长 18:48 大小 17.23M

你好，我是王磊，你也可以叫我 Ivan。

我们在 11、12 两讲已经深入分析了读写冲突时的控制技术，这项技术的核心是在实现目标隔离级别的基础上，最大程度地提升读写并发能力。但是，读写冲突只是事务冲突的部分情况，更多时候我们要面对的是写写冲突，甚至后者还要更重要些。

你可能经常会听到这类说法，“某某网站的架构非常牛，能抗住海量并发” “某某网站很弱，搞个促销，让大家去抢红包，结果活动刚开始 2 分钟系统就挂了”。其实，要想让系统支持海量并发，很重要的基础就是数据库的并发处理能力，而这里面最重要的就是对写冲突的并发控制。因为并发控制如此重要，所以很多经典教材都会花费大量篇幅来探讨这个问题，进而系统性地介绍并发控制技术。在这个技术体系中，虽然有多种不同的划分方式，但最为大家熟知就是悲观协议和乐观协议两大类。

TiDB 和 CockroachDB 的流行一度让大家对于乐观协议这个概念印象深刻。但是，经过几年的实践，两款产品都将默认的并发控制机制改回了悲观协议。他们为什么做了这个改变呢？我们这一讲，专门来探讨什么是乐观控制协议，以及为什么 TiDB 和 CockroachDB 不再把它作为默认选项。

并发控制技术的分类

首先，无论是学术界还是工业界，都倾向于将并发控制分为是悲观协议和乐观协议两大类。但是，这个界限在哪，其实各有各的解释。

我先给一个朴素版的定义。所谓悲观与乐观，它和我们自然语言的含义大致是一样的，悲观就是对未来的一种负面预测，具体来说，就是认为会出现比较多的事务竞争，不容易获得充足的资源完成事务操作。而乐观，则完全相反，认为不会有太多的事务竞争，所以资源是足够的。这个定义虽然不那么精准，但大体表示了两种机制的倾向。

再进一步，落到实现机制上，有一个广泛被提到的定义版本。乐观协议就是直接提交，遇到冲突就回滚；悲观协议就是在真正提交事务前，先尝试对需要修改的资源上锁，只有在确保事务一定能够执行成功后，才开始提交。

总之，这个版本的核心就是，悲观协议是使用锁的，而乐观协议是不使用锁的。这就非常容易把握了。

但是，这个解释和真正分布式数据库产品的实现还是有些差距的，比如 TiDB 就宣称自己使用了乐观锁。对，你没听错，是乐观锁。怎么有锁还能乐观呢，是不是有点蒙了？

为了让你在学习过程中不背负着这个巨大的问号，我们就先放下对定义的探讨，先来看看 TiDB 乐观锁的实现方式。

乐观锁：TiDB

TiDB 的乐观锁基本上就是 Percolator 模型，这个模型我们在第 9 讲时曾经介绍过，它的运行过程可以分为三个阶段。

1. **选择 Primary Row**

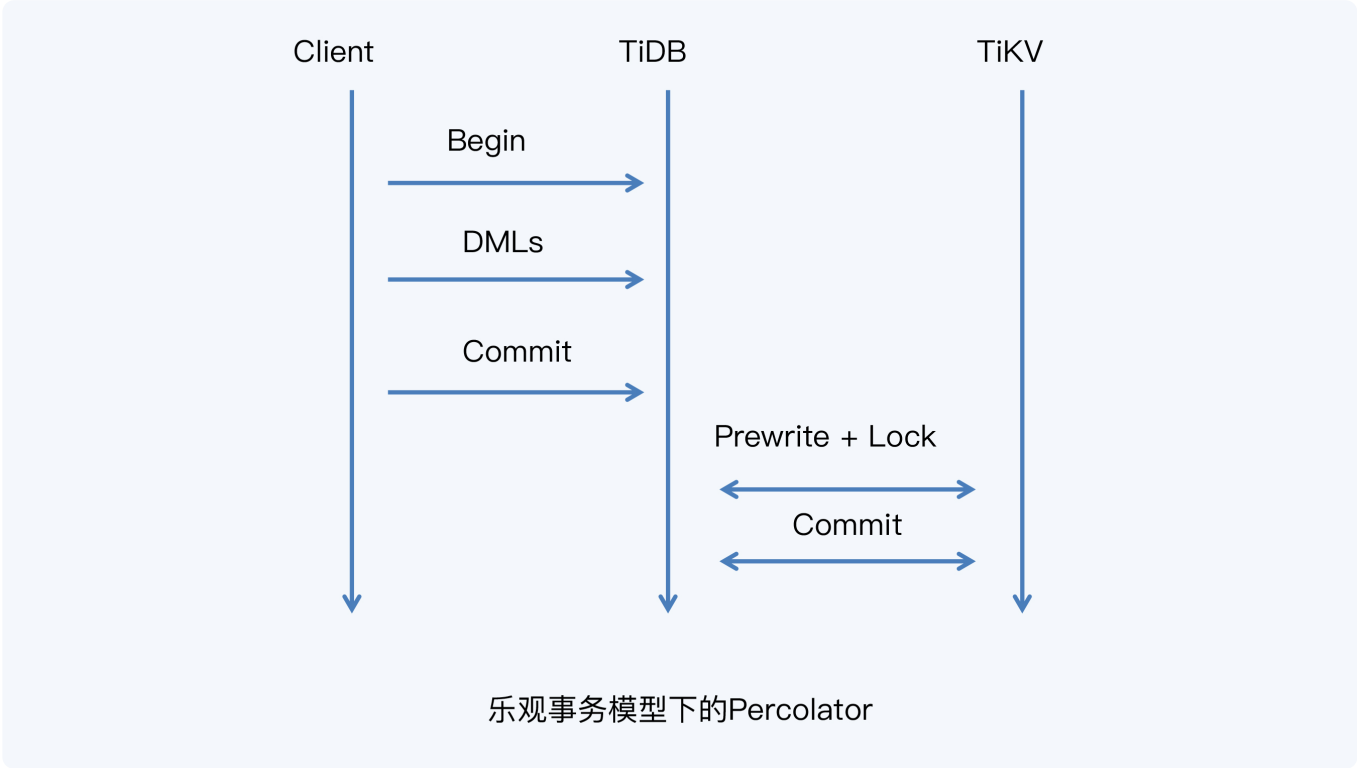
收集所有参与修改的行，从中随机选择一行，作为这个事务的 Primary Row，这一行是拥有锁的，称为 Primary Lock，而且这个锁会负责标记整个事务的完成状态。所有其他修改行也有锁，称为 Secondary Lock，都会保留指向 Primary Row 的指针。

2. **写入阶段**

按照两阶段提交的顺序，执行第一阶段。每个修改行都会执行上锁并执行“prewrite”，prewrite 就是将数据写入私有版本，其他事务不可见。注意这时候，每个修改行都可能碰到锁冲突的情况，如果冲突了，就终止事务，返回给 TiDB，那么整个事务也就终止了。如果所有修改行都顺利上锁，完成 prewrite，第一阶段结束。

3. **提交阶段**

这是两阶段提交的第二阶段，提交 Primary Row，也就是写入新版本的提交记录并清除 Primary Lock，如果顺利完成，那么这个事务整体也就完成了，反之就是失败。而 Secondary Rows 上的锁，则会交给异步线程根据 Primary Lock 的状态去清理。



乐观事务模型下的Percolator

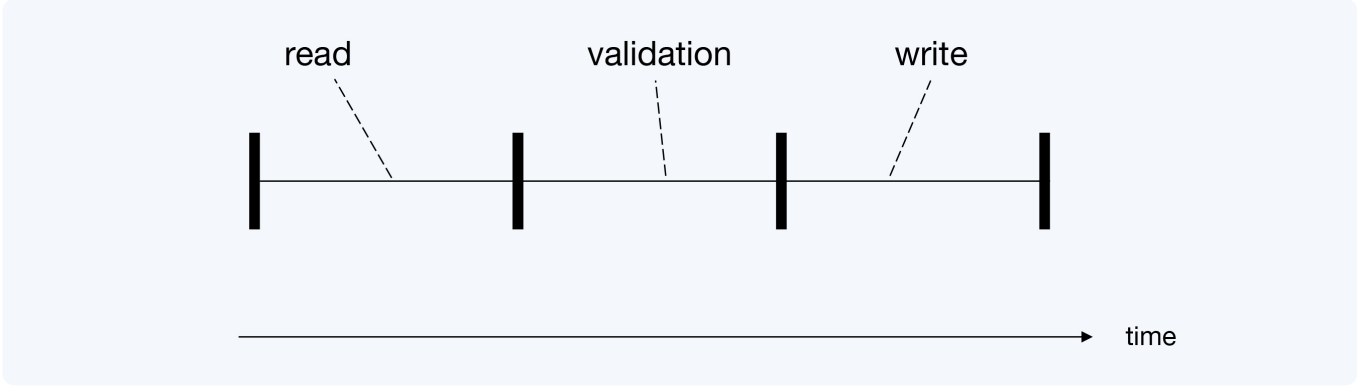
你看这个过程中不仅有锁，而且锁的数量还不少。那么，为什么又说它是乐观协议呢?

想回答这个问题，我们就需要一些理论知识了。

并发控制的三个阶段

在经典理论教材“Principles of Distributed Database Systems”中，作者将乐观协议和悲观协议的操作，都统一成四个阶段，分别是有效性验证（V）、读（R）、计算（C）和写（W）。两者的区别就是这四个阶段的顺序不同：悲观协议的操作顺序是 VRCW，而乐观协议的操作顺序则是 RCVW。因为在比较两种协议时，计算（C）这个阶段没有实质影响，可以忽略掉。那么简化后，悲观协议的顺序是 VRW，而乐观协议的顺序就是 RVW。

RVW 的三阶段划分，也见于研究乐观协议的经典论文“On Optimistic Methods for Concurrency Control”。



关于三个阶段的定义在不同文献中稍有区别，其中“Principles of Distributed Database Systems”对这三个阶段的定义通用性更强，对于 RVW 和 VRW 都是有效的，我们先看下具体内容。

读阶段（Read Pharse），每个事务对数据项的局部拷贝进行更新。

要注意，此时的更新结果对于其他事务是不可见的。这个阶段的命名特别容易让人误解，明明做了写操作，却叫做“读阶段”。我想它大概是讲，那些后面要写入的内容，先要暂时加载到一个仅自己可见的临时空间内。这有点像我们抄录的过程，先读取原文并在脑子里记住，然后誊写出来。

有效性确认阶段（Validation Pharse），验证准备提交的事务。

这个验证就是指检查这些更新是否可以保证数据库的一致性，如果检查通过进入下一个阶段，否则取消事务。再深入一点，这段话有两层意思。首先这里提到的检查与隔离性目标

有直接联系；其次就是检查可以有不同的手段，也就是不同的并发控制技术，比如可以是基于锁的检查，也可以是基于时间戳排序。

写阶段（Write Pharse），将读阶段的更新结果写入到数据库中，接受事务的提交结果。

这个阶段的工作就比较容易理解了，就是完成最终的事务提交操作。

还有一种关于乐观与悲观的表述，也与三阶段的顺序相呼应。乐观，重在事后检测，在事务提交时检查是否满足隔离级别，如果满足则提交，否则回滚并自动重新执行。悲观，重在事前预防，在事务执行时检查是否满足隔离级别，如果满足则继续执行，否则等待或回滚。

我们再回到 TiDB 的乐观锁。虽然对于每一个修改行来说，TiDB 都做了有效性验证，而且顺序是 VRW，可以说是悲观的，但这只是局部的有效性验证；从整体看，TiDB 没有做全局有效性验证，不符合 VRW 顺序，所以还是相对乐观的。

下面这一段，我们稍微延伸一下有关乐观并发控制的知识。

狭义乐观并发控制（OCC）

“Transactional Information Systems : Theory, Algorithms, and the Practice of Concurrency Control and Recovery”给出了一个专用于 RVW 的三阶段定义，也就是说，它是专门描述乐观协议的。其中主要差别在“有效性确认阶段”，是针对可串行化的检查，检查采用基于时间戳的特定算法。

这个定义是一个更加具体的乐观协议，严格符合 RVW 顺序，所以我把它称为狭义上的乐观并发控制（Optimistic Concurrency Control），也称为基于有效性确认的并发控制（Validation-Based Concurrency Control）。很多学术论文中的 OCC，就是指它。而在工业界，真正生产级的分布式数据库还很少使用狭义 OCC 进行并发控制，唯一的例外就是 FoundationDB。与之相对应的，则是 TiDB 这种广义上的乐观并发控制，说它乐观是因为它没有严格遵循 VRW 顺序。

乐观协议的挑战

TiDB 的乐观锁已经讲清楚了，我们再回到这一讲的主线，为啥乐观要改成悲观呢？主要是两方面的挑战，一是事务冲突少是使用乐观协议的前提，但这个前提是否普遍成立，二是现有应用系统使用的单体数据库多是悲观协议，兼容性上的挑战。

事务频繁冲突

首先，事务冲突少这个前提，随着分布式数据库的适用场景越来越广泛，显得不那么有通用性了。比如，金融行业就经常会有一些事务冲突多又要保证严格事务性的业务场景，一个简单的例子就是银行的代发工资。代发工资这个过程，其实就是从企业账户给一大批个人账户转账的过程，是一个批量操作。在这个大的转账事务中可能涉及到成千上万的更新，那么事务持续的时间就会比较长。如果使用乐观协议，在这段时间内，只要有一个人的账户余额发生变化，事务就要回滚，那么这个事务很可能一直都在重试、回滚，永远也执行不完。这个时候，我们就一点也不要乐观了，像传统单体数据库那样，使用最悲观的锁机制，就很容易实现也很高效。

当然，为了避免这种情况的出现，TiDB 的乐观锁约定了事务的长度，默认单个事务包含的 SQL 语句不超过 5000 条。但这种限制其实是一个消极的处理方式，毕竟业务需求是真实存在的，如果数据库不支持，就必须通过应用层编码去解决了。

遗留应用的兼容性需求

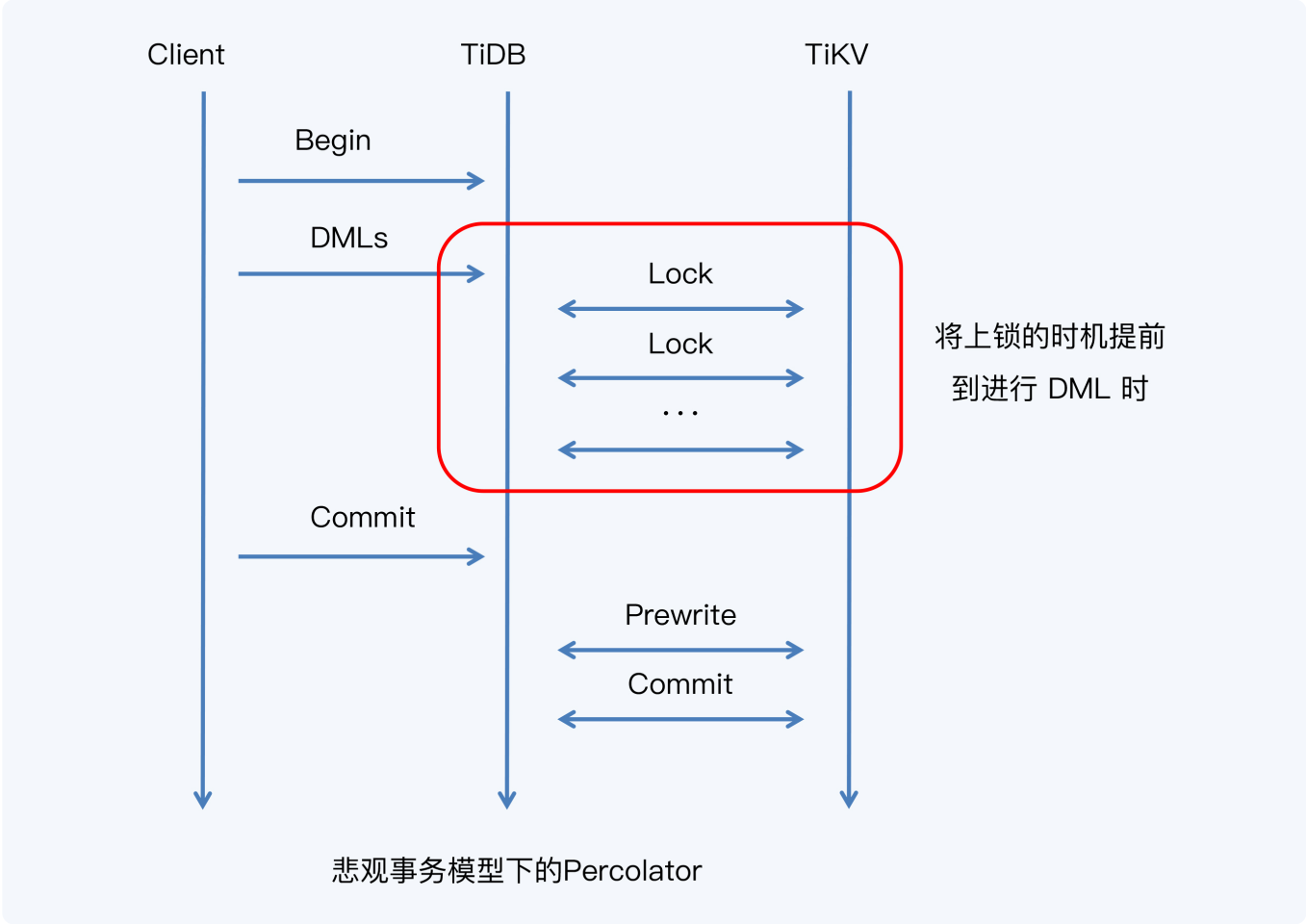
回到悲观协议还有一个重要的原因，那就是保证对遗留应用系统的兼容性。这个很容易理解，因为单体数据库都是悲观协议，甚至多数都是基于锁的悲观协议，所以在 SQL 运行效果上与乐观协议有直接的区别。一个非常典型的例子就是 select for update。这是一个显式的加锁操作，或者说是显式的方式进行有效性确认，广义的乐观协议都不提供严格的 RVW，所以也就无法支持这个操作。

select for update 是不是一个必须的操作呢？其实也不是的，这个语句出现是因为数据库不能支持可串行化隔离，给应用提供了一个控制手段，主导权交给了应用。但是，这就是单体数据库长久以来的规则，已经是生态的一部分，为了降低应用的改造量，新产品还是必须接受。

乐观协议的改变

因为上面这些挑战，TiDB 的并发控制机制也做出了改变，增加了“悲观锁”并作为默认选项。TiDB 悲观锁的理论基础很简单，就是在原有的局部有效性确认前，增加一轮全局有效

性确认。这样就是严格的 VRW，自然就是标准的悲观协议了。具体采用的方式就是增加了悲观锁，这个锁是实际存在的，表现为一个占位符，随着 SQL 的执行即时向存储系统（TiKV）发出，这样事务就可以在第一时间发现是否有其他事务与自己冲突。



另外，悲观锁还触发了一个变化。TiDB 原有的事务模型并不是一个交互事务，它会把所有的写 SQL 都攒在一起，在 commit 阶段一起提交，所以有很大的并行度，锁的时间较短，死锁的概率也就较低。因为增加了悲观锁的加锁动作，变回了一个可交互事务，TiDB 还要增加一个死锁检测机制。

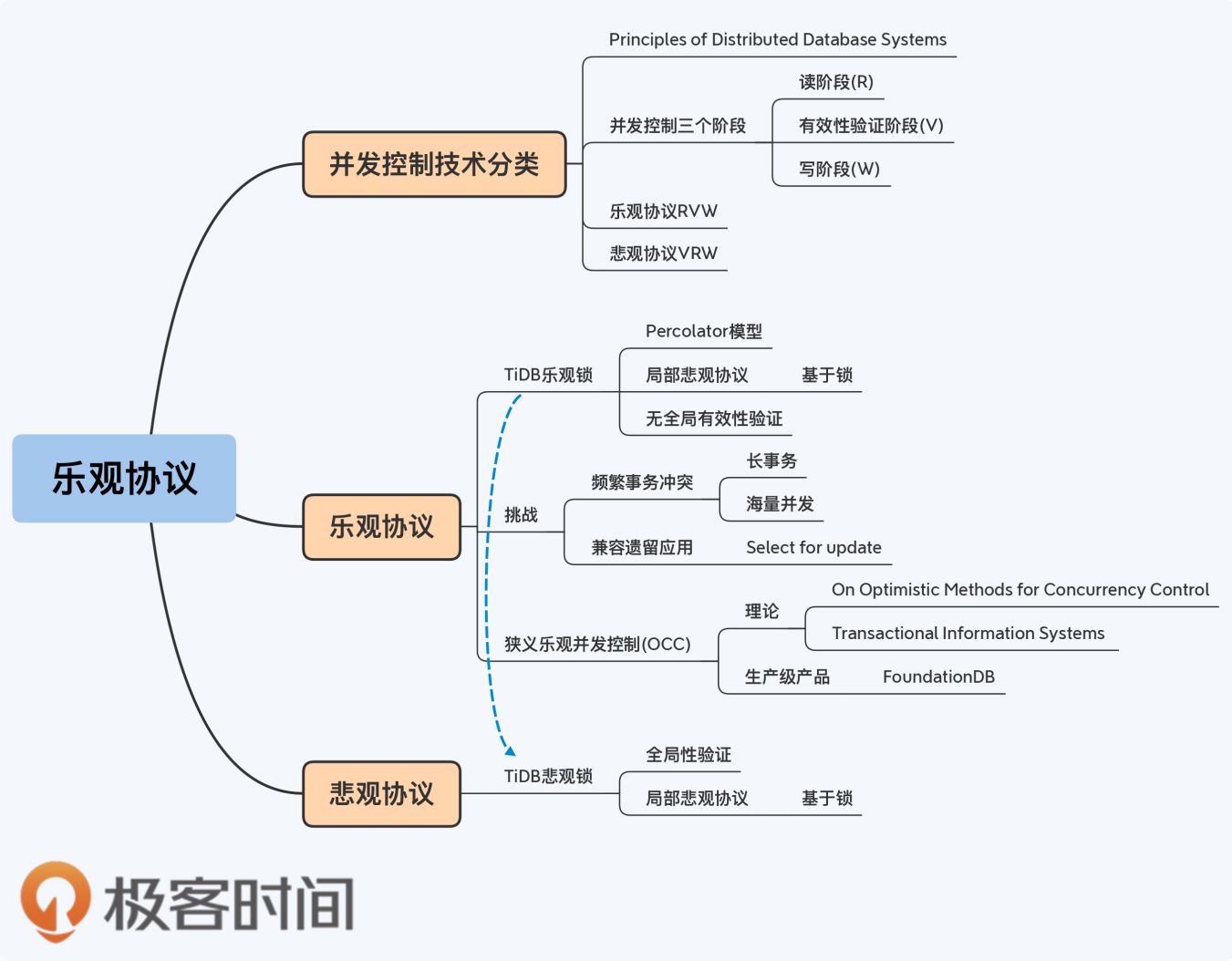
小结

到这里，今天的内容就告一段落了，让我们一起梳理下今天内容。

1. 并发控制分为乐观和悲观两种，它们的语义和自然语言都是对未来正面或负面的预期。乐观协议预期事务冲突很少，所以不会提前做什么，悲观协议认为事务冲突比较多，所以要有所准备。

2. 按照经典理论，并发控制都是由三个阶段组成，分别是有效性确认（V）、读（R）和写（W）。悲观协议的执行顺序是 VRW，乐观协议的执行顺序是 RVW。所以，又可以得到两个乐观协议的定义，狭义上必须满足 RVW 才是乐观并发控制，而且三阶段有更具体的要求，这个就是学术论文上的 OCC。另外广义的定义，只要不是严格 VRW 的并发控制，都是相对乐观的，都是乐观并发控制，TiDB 就是相对乐观。生产级的分布式数据库很少有使用狭义的 OCC 协议，FoundationDB 是一个例外。
3. 乐观协议的挑战来自两个方面。第一点，乐观协议在事务冲突较少时，因为避免了锁的管理开销，能提供更好的性能；但在事务冲突较多时会出现大量的回滚，效率低下。总的来说，乐观协议的通用性并不好。第二点，传统单体数据库几乎都是基于锁的悲观协议，乐观协议在语义层面没有对等的 SQL，例如 select for update。因此，TiDB 和 CockroachDB 都由乐观协议转变为悲观协议，并且是基于锁的悲观协议。
4. TiDB 的悲观协议符合严格的 VRW 顺序，在原有的两阶段提交前，增加了一轮悲观锁占位操作，实现全局有效性确认。但是随着悲观锁的引入，TiDB 转变为一个可交互事务模式，出现死锁的概率大幅提升，对应增加死锁检测功能。

今天的课程中，我们澄清了对乐观协议的常见误解，也解释了为什么 TiDB 要把默认选项从乐观锁改为悲观锁。当然你一定注意到了，除了基于锁的悲观协议外，还有一些其他技术，比如基于时间戳排序（TO）和串行化图检测（SGT）等，我们将在下一讲中继续探讨这个话题。



思考题

课程的最后，我们来看看今天的思考题。

在这一讲开头，我们说 TiDB 和 CockroachDB 都经历了从乐观协议到悲观协议的转变，但在文中只展开了 TiDB 变化前后的设计细节。其实，对于 CockroachDB 你应该也不陌生了，我们在最近几讲中都有提及到它的一些特性。所以，我今天的问题是，请你推测下 CockraochDB 的悲观协议大概会采用什么方式？而这种方式又有什么优势？当然，CockroachDB 的实现方式很容易查到，所以我更关心你的思考的过程。

欢迎你在评论区留言和我一起讨论，我会在答疑篇和你继续讨论这个问题。如果你身边的朋友也对乐观协议或者并发控制技术这个话题感兴趣，你也可以把今天这一讲分享给他，我们一起讨论。

学习资料

Gerhard Weikum and Gottfried Vossen: *Transactional Information Systems : Theory, Algorithms, and the Practice of Concurrency Control and Recovery*

H. T. Kung and John T. Robinson: *On Optimistic Methods for Concurrency Control*

M. Tamer Özsu and Patrick Valduriez: *Principles of Distributed Database Systems*

提建议

上一篇 12 | 隔离性：看不见的读写冲突，要怎么处理?

下一篇 14 | 隔离性：实现悲观协议，除了锁还有别的办法吗?

精选留言 (5)

写留言

扩散性百万咸面包
2020-09-10

老师感觉这里没说清楚：
TiDB在2PC prewrite阶段会给每一行加锁，这个锁是局部性的，怎么理解局部性？意思是说这个加锁只是这个事务的单方面行为，而没有通过数据库的锁管理器等组件吗？因为我看到TiDB是会观察所有加锁请求是否成功的，感觉这好像已经算是做全局性的有效验证了？要怎样加锁才是全局性的有效验证呢？

展开

> 作者回复: 局部性是说，只要局部加锁成功，就开始局部的写入了。TiDB的乐观锁也不是在所有记录加锁成功以后，才写入新的数据版本。如果是，那还会因为事务冲突多而频繁回滚吗？再想想:)
>
> 怎么加锁才是全局有效性验证？TiDB后来增加的悲观锁就是例子。这个加锁动作是统一的，只要有一条记录加锁不成功，就不会启动任何写入动作。

1

feihui
2020-09-12

老师，关于你在留言中说到的锁，是不是可以理解mysql inodb 引擎的写锁也是乐观锁？毕竟也只是在访问的时候才会加锁

展开

真名不叫黄金
2020-09-09

老师，你好，有一个问题想请教一下：
Spanner 在文档中称自己的事务是基于悲观锁的，但是其实它也是将 Writes 缓存住的，在 commit 时一次性获取锁、完成两阶段提交，它在执行 DML 时并没有检测冲突，Spanner 在提交时采用伤停等待策略，实践中确实可能会带来冲突时很多事务的大量 abort 和 retry。...

展开

扩散性百万咸面包
2020-09-07

一个非常典型的例子就是 select for update。这是一个显式的加锁操作，或者说是显式的方式进行有效性确认，广义的乐观协议都不提供严格的 RVW，所以也就无法支持这个操作。
这句话没懂，广义的乐观协议不是支持加锁吗（比如TiDB对每一行修改前加锁）？还是说像TiDB这样的，它的加锁不是显式的，只能在percolator 2PC中加锁，所以不支持这种...
展开

春风
2020-09-07

老师，TiDB全局性验证是否是增加了一轮共识算法的时间

作者回复: 是的，因为TiDB的悲观锁是要写一个实际的占位符，并且必须成功后，才能开始原有的乐观锁处理过程，所以就会增加一轮共识算法时间。