本文主要是关于并发控制和原子性提交，人们将并发控制和原子性提交放在一起，抽象地说就是事务

serializable（顺序性） 的定义指的是，这些结果是有序的，即并发事务执行的结果与某个顺序执行事务的结果一样，如果出现其他的结果，则是不合法的，这能让我们正确执行事务

如果不同事务之间没有使用相同的数据对象，则我们可以并行执行事务。如果你使用的是一个分片系统，我们可以获得真正的并行加速能力，一个事务可能在第一台机器的第一个分片中执行，另一个事务可能并行运行在第二个机器上

另外，如果事务在执行过程中 abort（中止）了，则我们需要能够结束这些事务，并撤销这些事务已经做出的任何修改

并发控制

对于并发控制主要有两种方案：

1. 悲观锁并发控制

   即在事务使用任何数据前，它需要先去获取该数据对应的锁，如果在获取锁前，该锁已经被其他事务持有，那么需要等待

2. 乐观锁并发控制

   不管是否有其他事务和我们同时对数据进行读取，直接对数据进行读写，并将结果写入临时区域，到最后再来检查是否有其他事务对我们所执行的这个事务产生影响，若没有，则执行完毕，如果有，则中止这个事务并重新执行

事实证明，在不同情况下，其中一种方案的速度会比另一种方案来得更快，如果经常发生冲突，则应该用悲观锁并发控制，否则你会遇到一大堆事务中止的情况。如果冲突很少发生，那么使用乐观锁并发控制的速度就会更快，因为这完全避免了锁带来的开销。

下面来看看悲观策略：

这里讲的是两阶段锁，在事务中，两阶段锁的思路是：

1. 在对任何数据进行读取或写入操作前，先去获取该数据对应的锁
2. 直到事务被提交或中止后，事务才能释放掉它所获取的锁

你不允许在事务执行的过程中释放锁，你可能会想，在事务使用完该数据后就释放该锁，而不是事务结束后才释放锁，这样是不对的，以转账为例，在账户 x 加1后，事务1释放了该锁，然后事务2输出 x，然后接着又读到了 y，此时事务1还未将 y 减1，那么就会出现不一致的情况。或者是事务1释放该锁后，中止了，那么就得撤销事务1产生的影响，这时事务2就会读到一个不存在的数

事务并发执行可能也会有死锁的问题，此时数据库会根据一些方案来检测死锁，然后中止其中的一个事务，并撤销这个事务所做的修改

以上就是数据库中两阶段锁的并发控制，目前来说和单机数据库是完全一致的

下面来看看分布式数据库，此时数据分散到多个机器上，即执行一个事务，可能用到许多机器上的值，所以下面讲的是如何构建分布式事务，特别是如何应对故障，比如将一台服务器上的数据加1，另一台服务器上的数据减1，但执行完一台服务器上的操作后，另一台服务器故障了。对于分布式事务，我们要求要么执行事务中的所有操作，要么就不执行该事务中的所有的操作

原子提交协议

原子提交协议的基本特性是，假设有一堆机器，原子提交协议就是帮助这些机器判断能否完成并真正执行该部分任务，或者如果中间发生了故障，参与者就会决定他们不去执行该事务所涉及的所有操作

两阶段提交

假设目前执行的事务所需数据分散在两个服务器中。我们需要一个**事务协调器**，其维护执行这些操作相关的事务代码，比如 put、get 和 add 操作。事务协调器会发送信息给服务器1，说：请对 x 的值加1，发一条消息给服务器2，说：请对 y 的值减1，接着我们需要通过多条消息来确保这两条消息都被执行了，或者都没被执行。每个服务器都会维护一个 lock 表，它们会对状态进行跟踪，同时还有事务 id 的概念，即系统中每条消息都会打上相关事务的唯一 id，事务协调器会在事务开始时去指定这些事务 id。负责执行事务中部分任务的服务器叫做参与者，不同参与者所要操作的相关表会用这个事务 id 进行标记。

两阶段提交简写为 2PC。正如上面事务协调器向两个服务器发送消息那样，其会收到来自参与者的回复

客户端请求执行一个事务，其会等待事务协调器对其进行响应。在我们进行任何事务之前，事务协调器得去确保所有不同的事务参与者都能够执行它们负责的那部分事务，比如事务中出现 put 请求，那么就得确保处理 put 请求的参与者确实具备处理这些 put 请求的能力

一开始，事务协调器会发送 prepare 消息给所有事务参与者，当 A 和 B 收到 prepare 消息，它们会去查看它们的状态，并判断它们实际是否能够完成这个事务中的操作。它们此时可能在通过中断事务来处理死锁，或者崩溃重启，以致于忘记了该事务的有关信息，这样 A 和 B 会回复 No，即它们不能够去执行这个事务。事务协调器回复 Yes 说明能执行。

事务协调器会等待每个事务参与者的回复，如果回复都为 Yes，即说明可以执行，那么可以提交该事务，事务协调器会给每个事务参与者发送一条 commit 消息，然后事务参与者就会对事务协调器回复一条确认信息：ACK

如果在发送 prepare 消息时，有参与者说 No，只要有一个参与者说 No，事务协调器就不会发送 commit 消息，他会对所有参与者发送 abort 消息，并要求回滚该事务。

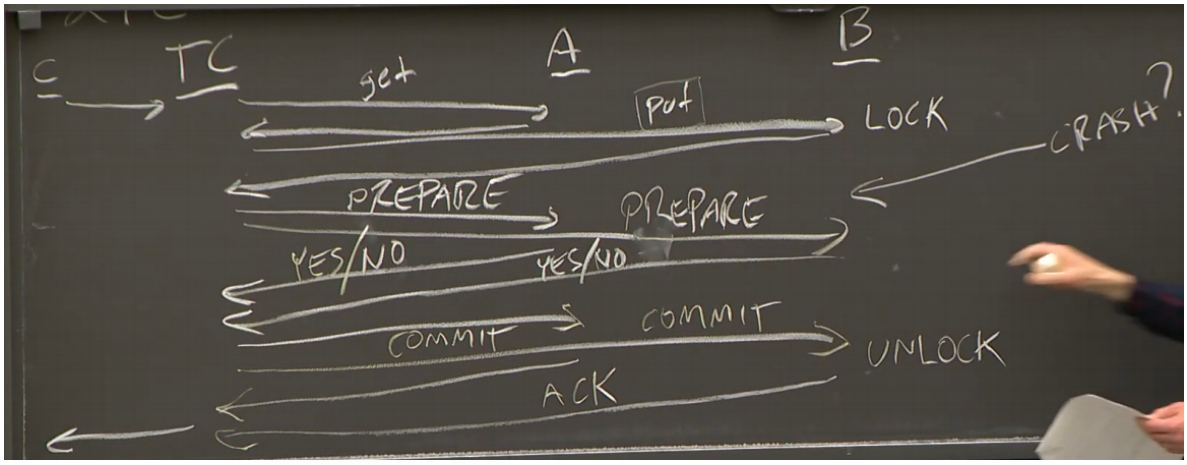
在发送 commit 消息后，会有两件事情，一件事就是，事务协调器会向请求该事务结果的客户端或用户发送该事务的执行结果，如果该事务没有中断，那么它就会被提交，并持久化到磁盘上。另一件事就是，当参与者看到 commit 消息或者 abort 消息时，它们就会释放这些锁

当参与者要去执行其所做的那部分事务时，其会去锁住它所要读取的那部分数据，每个参与者都有一张 lock 表，该表将锁与该事务所操作的数据对象关联起来。参与者会将这些锁信息用这些 lock 表进行保存，收到 commit 或 abort 消息后，就会释放该数据对应的锁，然后其他事务就能使用这些数据

如果所有参与者都遵循这个协议，并且没有故障发生，那么这两个参与者只有在它们收到的都是 commit 消息时，它们才能进行 commit 操作。如果它们无法进行 commit，或者它们中任何一个中止了，那么它们都得中止这个事务，也就是说，要么它们都能提交，要么全都不能

下面来看看出现故障的情况：

假设参与者 B 崩溃了（比如电源故障导致事务执行突然中断，服务器重启完毕后运行事务处理系统中的恢复软件进行恢复）：



1. 假设 B 在将 yes 回复给事务协调器前，他就已经崩溃了，那么它就不会对事务协调器说 Yes，那么事务协调器就不会发送 commit 消息给这些参与者

   如果在发送 yes 前，B 所执行的事务部分就发生了崩溃，从它自身的角度来说，它就不可能再返回一个 yes 了，那么 B 就有权中断该事务，并忘记该事务的相关内容，因为事务协调器不可能向他发送 commit 消息

   如果 B 中所有有关事务的信息此时都放在内存中，还没有发回给事务协调器，如果 B 崩溃重启的话，那么我们就会很容易丢失这些数据，然后如果事务协调器发送一条 prepare 消息给一个参与者，该参与者不清楚事务的信息，那么参与者就会说 No，请中止这个事务
2. B 在发送 Yes 给事务协调器后崩溃了，即收到 commit 消息前崩溃。我们需要 B 恢复之后依然能去完成它所负责的那部分事务。即不能因为崩溃和重启而丢失一个事务的状态。在 B 回复这个 prepare 消息之前，其得将该事务在内存中管理的中间状态持久化到磁盘上。所以，在 B 发送 Yes 来回复事务协调器所发的 prepare 之前，其必须得先将用于提交该事务所需的所有信息写入到磁盘上的日志。恢复时，其会读取日志，它依然准备去完成它所负责的那部分事务，此时它还没有收到 commit 消息，它不知道是否要进行 commit 操作。当 B 最终收到 commit 消息或者 abort 消息时，其会读取它的日志来弄清楚该如何结束它所负责的那部分事务

注意，当 B 收到 commit 消息后，它得将该事务所做的修改保存到它的数据库中，并且也要持久化到磁盘上，所以 B 重启时可能也不需要做什么事情，因为事务已经结束了。所以如果收到两次 commit 消息，因为 B 很容易忘记它已经提交的事务，所以此时简单地回复 ACK 即可

下面来看看事务协调器崩溃的情况

如果有任意参与者已经进行了提交操作，那么我们肯定不能忘记这些参与者已经做过提交操作了。比如 A 已经发送了 Yes 给事务协调器，事务协调器在发送 commit 前崩溃了，那么事务协调器就必须重启并重新发送 commit，以确保两方都知道该事务可以被提交，如果没有发送任何 commit 消息，其实此时事务协调器中止事务也可以。如果有其他参与者询问该事务相关的信息，事务协调器会回复不清楚该事务的任何信息

但如果是发送一条或多条 commit 消息后崩溃了，此时我们不允许事务协调器忘记该事务的相关信息。即事务协调器根据参与者回复的 yes 或 no 做出 commit 或 abort 消息回复时，在发送任何 commit 消息之前，事务协调器首先得将关于该事务的信息写入到它的日志中，并持久化到存储设备中，比如磁盘。如果崩溃重启了，那么这些信息依然还会在磁盘上，不会丢失，只有保存到磁盘上的日志中后，才会开始发送 commit 消息。崩溃恢复时，它的恢复软件就会去查看它的日志并说，目前还处于执行一个事务的期间，根据日志我们选择去提交事务还是中止这个事务，根据实际情况向所有的参与者重新发送 commit 或者 abort

下面来看看出现网络问题的情况

假设事务协调器发送了 prepare 消息给参与者，但是它并没有收到参与者所发生的 yes 或者 no 的回复。事务协调器可能会发出一组新的 prepare 消息，说我没有拿到你们的回复。如果其中有个服务器长时间掉线，可能会使事务一直等待，此时事务协调器可以单方面决定中止这个事务（发送 abort），因为还没有发送 commit 消息，所以这样做是正确的

现在，如果 B 收到 prepare 消息并回复 yes 后，其过了很久还没有收到 commit 消息，此时 B 会等待很长一段时间。但此时 B 没有权利单方面终止事务，其必须一直等下去，这种情况叫做阻塞。

必须一直等下去的原因是，因为 B 已经发送了 Yes 给事务协调器，此时事务协调器可能已经收到了这个 yes，它可能已经给部分参与者发送了 commit 消息。说明此时 A 可能已经收到了 commit 消息，并且提交了该事务，将该事务所做的修改落地，并释放锁，然后将这些修改展示给其他事务，所以 B 必须一直等下去

此时就得人为对事务协调器进行修复，并让他重启，然后读取其保存的日志，并继续发送 commit 消息。

同样的，你也不能单方面去提交事务，因为 A 可能已经投了 No，但 B 还没有收到事务协调器所发送的 abort 消息。所以，超时的时候，你不能中止事务也不能提交事务

实际上，这个阻塞行为其实二阶段提交的一个重要特性。这不是一个好的特性，因为消息丢失时你可能要等很久

之所以我们能够构建出一个能允许A和B同时提交或中止事务的协议的基础部分，其中一个原因是，这个决定是由单体做的，即事务协调器，A 或 B 都不会去决定要不要提交或者中止事务，只有事务协调器才能。由事务协调器去做出最终决定的代价是，在某个时间点我们可能会被阻塞，我们要去等待事务协调器告诉我们是提交事务，还是中止事务。同时事务协调器也要将事务有关信息保存在它的日志中。这样万一崩溃了，也可以避免事务丢失

当事务协调器忘记其日志中有关事务的信息时，其会试着获取参与者所发送的所有确认信息，以确定所有的参与者是都提交了这个事务，还是中止了这个事务。当参与者都发送确认消息给事务协调器后，它也就不需要去知道该事务的相关信息了

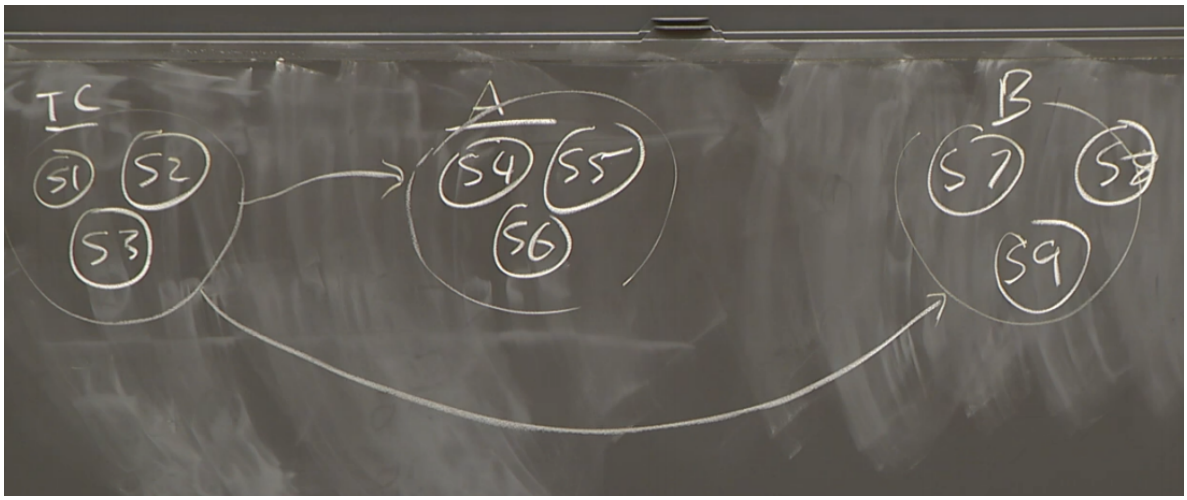
当事务协调器收到确认信息后，它就可以擦除与该事务相关的所有信息，一旦参与者收到了 commit 或 abort 信息，并且它们已经执行完它们所负责的那部分事务，并将它们的更新落地，释放它们所持有的锁。此时，当参与者发送确认消息给事务协调器后，它们也可以完全忘记该事务的有关信息

理论上，事务协调器可能没有收到该确认消息，它可能会重新发送 commit 消息，如果一个参与者收到了一个事务的commit消息，它对该事务并不清楚，因为它已经忘记了与该事务的信息，那么，参与者发送另一个确认消息给事务协调器即可。因为它忘记了与该事务相关的信息

两阶段提交是主要用于数据库或者存储系统上的，它们需要支持可以读取或写入多条记录的事务。很多更专业的存储系统不允许你在多条记录上使用事务。如果这种系统不支持对多条记录进行操作的事务，那么我们就不需要使用两阶段提交。但如果你可以对多条记录使用事务，并且你将数据分片并保存到多台服务器上，如果你想得到具备ACID特性的事务，那么你就需要去支持两阶段提交

两阶段提交速度很慢，因为它得发送好几轮信息，而且还有大量写入磁盘的操作。并且，如果消息丢失或者发生崩溃，参与者可能会拿着锁并等待很长一段时间。

通常来说我们可以利用 raft 来获得高可用性。因为 raft 中所有服务器做的事情都是相同的，而且我们只需要大多数人参与就行，但在两阶段提交中，所有参与者做的事情都是不同的，它们执行自己负责的那部分事务，来让整个事务执行完毕。如果其中一个节点崩溃了，我们就得等到错误修复完毕，才能继续执行，所以我们可以将其与 raft 结合起来，使得系统既具备 raft 进行复制所带来的高可用性，也拥有在使用二阶段提交时让参与者去执行它们所负责的事务的这种能力，即利用 raft 或 paxos 来对每个不同的一方来进行复制，即事务协调器运行一个 raft 集群，每个参与者也运行一个 raft 集群，它们之间会来回交换信息



这样就能获得稳定点的系统