Read Committed 丢失更新

↳

你提出的问题很有道理，但确实，**丢失更新** 的关键在于事务 A 提交时，如果事务 B 已经提交了更新，事务 A 提交的更新会覆盖掉事务 B 的更改，导致数据的不一致或“丢失”。
在以上例子中，数据C只能有一个值，不是150，就是200，无论怎么避免始终都会有一个更新丢失，那还有什么用？

你提出的这个问题非常关键，确实在并发事务中，**数据 C 只能有一个最终值**，它的值要么是 **150**（事务 A 提交的更新），要么是 **200**（事务 B 提交的更新）。无论如何处理，**丢失更新问题** 是指在这种并发操作的情况下，由于没有适当的协调机制，最终有一个事务的更新“被丢失”，或者说它被另一个事务的更新覆盖。

丢失更新的根本含义

在这个情境中，"丢失" 是指：一个事务的更新被另一个事务的更新覆盖，因此这个被覆盖的事务的更改**没有对最终数据产生影响**。例如：

- **事务 A** 更新了 C 为 150。
- **事务 B** 更新了 C 为 200，并提交了。
- 然后 **事务 A** 提交，最终 C 的值是 150，而不是 200。事务 B 的修改被事务 A 覆盖，**这就是“丢失更新”**。

你说的“最终值只能是150或200”

你提到的“最终值只能是 150 或 200”确实是正确的。在并发事务中，只有一个最终的值被保留，但是丢失更新问题的关键点在于：

- **哪一个事务的更新应该优先？**
- **如何避免无意识的覆盖？**

丢失更新的作用与解决的意义

虽然丢失更新问题本身并不会让数据库中最终的值不可用，但它反映了并发环境下操作顺序的不确定性，带来的问题是数据的不一致性和冲突。通过采取适当的同步控制措施，目的是：

- **确保数据的完整性和一致性**，即避免无意中丢失事务 A 或事务 B 的有效更新。
- **控制事务的执行顺序**，让开发人员可以更加精确地控制并发事务的交互，确保不会发生冲突。

解决丢失更新的目的

1. **避免无意识的数据覆盖**：
    - 假设在事务 A 和事务 B 都提交时，数据库应该在两者之间做出判断，或者至少让用户或系统开发者知道哪个事务的更新被覆盖。

- 如果没有冲突检测机制，事务 A 可能“无意”地覆盖了事务 B 的有用更改（或反之）。这对某些业务流程可能是不可接受的，尤其是在涉及关键数据或高优先级更新时。

2. **增加事务的协调性**：
   - 通过使用**悲观锁**或**乐观锁**，可以显式地防止或减少并发冲突。它帮助系统判断两个事务是否可能冲突，并在冲突时采取适当措施（例如，回滚一个事务、重试、提示用户等）。
3. **避免潜在的错误**：
   - 如果没有冲突控制，丢失更新问题可能在不经意间发生，导致系统出现意外错误或者不一致的结果。尤其是在复杂业务逻辑中，丢失某个事务的更改可能意味着严重的业务错误。

解决方案的应用场景

1. **乐观锁**：可以在事务提交时检查数据是否已被其他事务修改，从而决定是否提交更新。比如，使用版本号，若版本号不同，表明数据已被修改，事务就可以选择回滚或重试。乐观锁的核心在于通过检测版本号来避免不必要的数据覆盖。
2. **悲观锁**：如果事务 A 修改的数据可能被事务 B 修改，可以对数据加锁，事务 B 必须等待事务 A 提交后才能修改数据，这样可以避免冲突和丢失更新。

总结

你提到的“最终只有一个值”没有错，确实如此。但**丢失更新问题**的核心在于，两个并发事务同时修改相同的数据时，没有机制来合理协调哪个事务的更改应该保留，导致某个事务的有效修改“丢失”。解决这个问题的关键是通过适当的锁机制或事务隔离级别来确保事务的执行不会相互冲突，从而提高数据一致性和系统的可靠性。