你这个问题问得太好了，直接命中了数据库并发控制设计的核心权衡！ 你感到奇怪是非常正常的，因为这正是“可重复读”隔离级别所做出的一个人为的、有代价的保证。

这个问题的答案是：对于读者的这种“保留”（即看到旧版本数据），会一直持续到读者（即该事务）自己结束（提交或回滚）为止。

核心原理：事务的“视界”

数据库的“可重复读”隔离级别为每个事务创建了一个一致的、静态的数据库快照，作为这个事务的“视界”。无论外界（其他事务）如何风云变幻，在这个事务看来，数据库就像被按下了暂停键，永远停留在它开始的那个时间点（或它第一次读取数据的时候）。

所以，保留到什么时候？保留到该事务结束。

这是如何实现的？

主流数据库（如 PostgreSQL、MySQL with InnoDB）通过 多版本并发控制 (MVCC) 技术来实现这一点。

1. 数据行是多版本的：每次更新（UPDATE）一行数据时，数据库并不是直接覆盖旧数据，而是插入一条新的版本记录，并标记其版本号（或时间戳）。旧版本数据并不会被立即删除。
2. 事务有开始时间：每个事务在开始时都会被分配一个唯一的、递增的事务ID（或时间戳）。
3. 读操作访问旧版本：当一个处于“可重复读”隔离级别的事务执行读取操作时，数据库引擎会遵循一个简单的规则：

   > 只查找那些版本号“小于或等于”当前事务ID，并且是已提交的最新版本的数据行。

- 时刻 T1：事务A（ID=100）开始，读取了 `age = 30`。数据库返回这个版本给它。
- 时刻 T2：事务B（ID=101）更新了这行数据，提交了 `age = 31`。数据库创建了一个新版本，该版本的事务ID范围标记为“101至无穷大”。
- 时刻 T3：事务A再次读取。数据库引擎发现有一个更新的版本（age=31），但该版本的事务ID（101）大于事务A的ID（100）。根据规则，这个新版本对事务A是“不可见的”。引擎会继续回溯，找到上一个已提交的、事务ID<=100的版本，也就是 `age=30`，并将其返回给事务A。

为什么要这样做？—— 设计哲学与代价

这种“保留”看似反直觉，但它是为了满足一种重要的业务需求：确保在一个事务单元内，所有基于数据的决策和计算都基于一个一致的状态。

典型应用场景：

1. 对账业务：你在事务中需要计算总和。如果第一次计算总和是1000，第二次因为别人修改了数据而变成900，你就无法判断这100的差额是因为你的计算逻辑问题，

还是外部修改导致的。
2. 一致性报告：生成一份报告，报告中的多个数据项必须反映同一时刻的状态。如果中间数据变了，报告就会自相矛盾。
3. 基于查询结果的后续操作：你先查询“有多少订单金额超过1000？”，得到数量N，然后想为这N个订单执行某个操作。如果在此期间有新订单插入或旧订单金额被修改，你的后续操作对象就错了。

代价：

这种保证不是免费的，它带来了显著的代价：

- 存储开销：数据库需要为同一行数据保留多个版本，直到所有可能用到它们的老事务都结束。这会导致存储空间增长。
- 性能开销：维护版本链和判断可见性需要额外的CPU计算。
- 清理开销：需要有一个后台任务（如 PostgreSQL 的 `VACUUM`）来定期清理那些已经不被任何事务需要的旧数据版本，否则数据库会无限膨胀。

总结

你的困惑	数据库的解答
“写者真实修改了数据，读者的保留要持续到什么时候？”	持续到读者事务自己结束。
“这似乎不反映现实？”	这是一种人为的、强一致性保证，牺牲了“时效性”来换取“可重复性”。
“如何实现的？”	通过MVCC：数据多版本 + 事务快照隔离。读者事务永远只能看到它开始之前已提交的数据版本。

所以，这并不是一个技术上的缺陷，而是一个设计上的选择。数据库提供了不同等级的隔离级别，就是让你根据业务场景在性能、并发性和一致性之间做出权衡。

- 如果你需要最高的并发和最新的数据，可以选择读已提交，接受不可重复读。
- 如果你需要保证事务内部的一致性，就选择可重复读，接受相应的性能和存储代价。

你的“奇怪”感觉，正是理解了这种权衡之后的表现！