



第六章

分布式事务

第六章、分布式事务

01

6.1 事务的基本概念

02

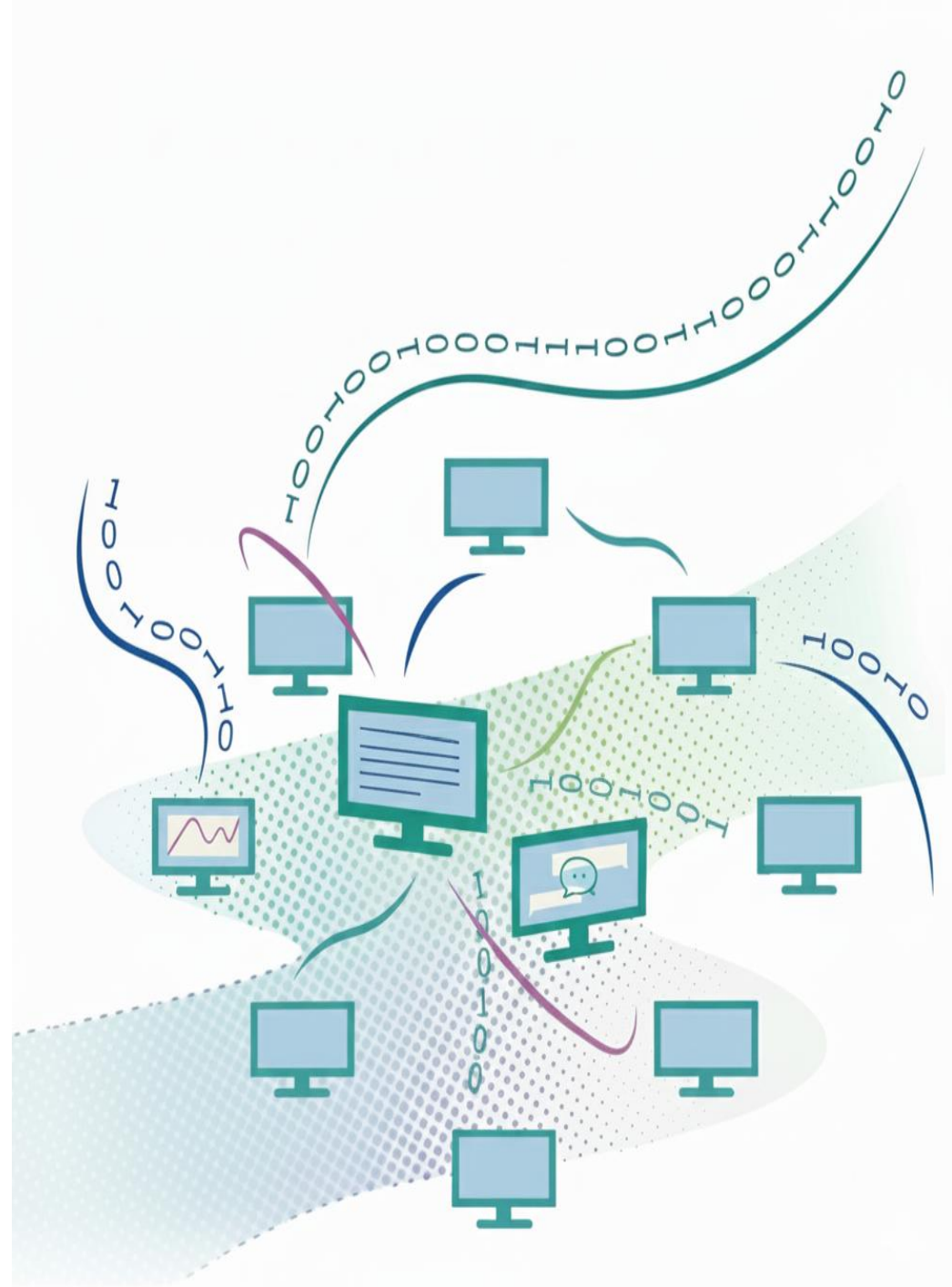
6.2 分布式事务与并发控制

03

6.3 分布式提交

04

6.4 分布式事务处理系统



6.1 事务的基本概念

01

6.1.1 事务的定义

02

6.1.2 事务的基本性质

03

6.1.3 并发事务与隔离级别



6.1.1 事务的定义

1、事务定义

事务是由若干个为完成某一任务而逻辑相关的操作组成的操作序列，是保证数据库正确性的基本逻辑单元。

下面是一个完整的SQL事务示例，包括事务的开始、操作和结束（提交或中止）：

```
SQL ▼
1  -- 开始一个事务
2  BEGIN TRANSACTION;
3  -- 执行一些数据库操作
4  INSERT INTO Employees (Name, Department) VALUES ('Alice', 'HR');
5  UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
6  UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
7  -- 如果所有操作都成功，则提交事务
8  COMMIT;
```

2、事务具体组成

一个事务由开始标识、数据库操作和结束标识三部分组成，对应于SQL中的三种执行命令：begin_transaction、commit、abort

3、事务的两种结束方式

- (1) commit: 提交
- (2) abort: 中止

6.1 事务的基本概念

01

6.1.1 事务的定义

02

6.1.2 事务的基本性质

03

6.1.3 并发事务与隔离级别



6.1.2 事务的基本性质

1、事务的4个特性（简称事务ACID特性）

- (1) 原子性 (Atomicity)
- (2) 一致性 (Consistency)
- (3) 隔离性 (Isolation)
- (4) 持久性 (Durability)

事务的ACID特性用于确保事务在数据库系统中安全正确地执行，从而确保数据的可靠完整。

2、大多数系统都需要保证数据的可靠完整

银行系统、购物系统、预定系统（假如没有确保数据可靠完整的保证，那么在执行数据操作时可能会出现各种问题，如账户资金错误、订单丢失等等。）

3、并非所有数据库系统都需要保证事务的ACID特性

在某些情况下为了追求更高的性能或满足特定需求

6.1.2 事务的基本性质——原子性 (Atomicity)

1、定义

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

2、举例

(1) 正例：一个事务更新100行记录，但是在更新了20行之后崩溃（原因如进程崩溃、网络连接中断、磁盘写满、或者某种完整性约束被违反），那么此时数据库会回滚（撤销）对那20条记录的修改。

(2) 反例：某个数据库事务要从A账户向B账户转账100元，即更新A存款-100，更新B存款+100，但如果更新A的存款-100后，事务崩溃。这将造成B无法收到A的100元转账，而A的100元却凭空消失的严重后果。

因此，数据库的原子性是维护数据库安全正确的重要保证。如果违反事务原子性，即事务的操作只完成了一部分就中止了，这将导致数据的不一致。

6.1.2 事务的基本性质——一致性 (Consistency)

1、定义

一致性是指事务执行的结果必须是使数据库从一个有效（正确）状态变换到另一个有效（正确）状态。

2、举例

在银行账户中，账户余额不可以为负，并且转账前后银行账户总和要保持不变。

例：账户A和账户B转账前均为100，即总和为200，那么在账户A向账户B转账50后，必须保证账户A和账户B的总和仍为200，并且转账不可超过100，这是银行账户系统必须满足的完整性约束，只有满足这些完整性约束，银行账户数据库才是一致的。

事务的一致性是由应用层保证的，也就是依赖于开发者，而其它原子性、隔离性、持久性是由数据库系统保证的。

6.1.2 事务的基本性质——隔离性 (Isolation)

1、定义

隔离性是指多个用户并发访问数据库时，数据库为一个用户开启的事务，不能被其他事务的数据操作所干扰，多个并发事务之间要相互隔离。

2、并发异常

在多用户场景下，多个事务可能并发地对相同的数据进行操作。如果没有隔离性，这些并发事务可能产生一系列问题，如：

脏写 (Dirty Write, 一个事务修改了另一个事务未提交的更新数据)、

脏读 (Dirty Read, 一个事务读取了另一个事务未提交的更新数据)、

不可重复读 (Non-repeatable Read, 一个事务中对同一数据项的多次读取结果不一致)、

幻读 (Phantom Read, 一个事务中同一范围查询的结果集合发生变化)。

6.1.2 事务的基本性质——隔离性 (Isolation)

3、举例

隔离性确保了每个事务都能独立地操作数据，避免了并发事务之间相互干扰。

例如：

两个事务同时读取 $x=1$ ，并将 x 自增1，并写回数据库。在保证事务隔离性的情况下，两个并发事务相互隔离，事务A先自增1，事务B后自增1，最终结果 $x=3$ 。

在不保证隔离性的情况下，两个事务同时读 $x=1$ ，事务A和事务B各自将 x 自增1，分别都得到 $x=2$ ，两个事务各自都写入 $x=2$ ，最终结果为 $x=2$ ，导致错误发生。

6.1.2 事务的基本性质——隔离性 (Isolation)

4、隔离性的多种级别 (隔离级别)

- 可串行化：数据库保证多个并发事务的执行结果和某个顺序下串行执行结果一致。
- 可串行化是事务隔离性的最高级别，即如果一组事务在可串行化隔离级别下执行，那么它们的执行结果与按某顺序一个一个执行的结果相同。
- 因此可串行化隔离级别消除了所有的并发问题，包括脏写、脏读、不可重复读和幻读。
- 为了平衡性能和数据一致性，一些数据库系统提供了不同的隔离级别，包括：
 - 读未提交 (Read Uncommitted) 、
 - 读已提交 (Read Committed) 、
 - 可重复读 (Repeatable Read)
- 这些隔离级别提供了较弱的隔离性，允许事务之间交织执行，具有更高的并发性和执行性能。

6.1.2 事务的基本性质——持久性 (Durability)

1、定义：

持久性是保证数据可靠性的重要保证。即一旦事务成功提交，数据的更改就被永久保存，即使系统发生意外故障或异常，写入的数据也不会丢失。

2、单机与分布式：

- 在单机数据库中，持久性通常意味着数据已被写入非易失性存储设备，如硬盘或固态硬盘 SSD。通常数据还会被预写入日志或类似的文件，以便在磁盘上的数据结构损坏时进行恢复。
- 在多副本的分布式数据库中，持久性意味着数据已成功复制到一些节点。为了提供持久性保证，数据库必须等到这些写入或复制完成后，才能报告给用户事务已经成功提交。

6.1 事务的基本概念

01

6.1.1 事务的定义

02

6.1.2 事务的基本性质

03

6.1.3 并发事务与隔离级别



6.1.3 并发事务与隔离级别

当多个事务并发访问数据库时，并发事务之间可能产生的并发异常包括脏写，脏读，不可重复读，幻读，丢失更新和写偏序等等。ANSI SQL-92标准根据这些并发异常明确了读未提交、读已提交、可重复读和可串行化四种隔离级别。四种隔离级别与并发异常的关系如下表。除此之外，还存在快照隔离等其他隔离级别。

	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

隔离级别与并发异常的关系

1、读未提交 (Read Uncommitted)

读未提交是数据库事务隔离级别中最低的一级，一个事务可以读取另一个事务尚未提交的未经验证的修改。

举例：

在事务1未提交时，事务2就可以读到 $x=2$ 。但是如果事务1回滚，即写入的 $x=2$ 回滚成初始值 $x=1$ 了，也就是说事务2读到了未提交的 $x=2$ ，出现了脏读问题。事务2之后如果基于读到的未提交的 $x=2$ 继续执行后面的操作，比如将 x 自增1得到 $x=2+1=3$ ，基于一个未提交成功的值更新数据库，可能会产生异常问题。

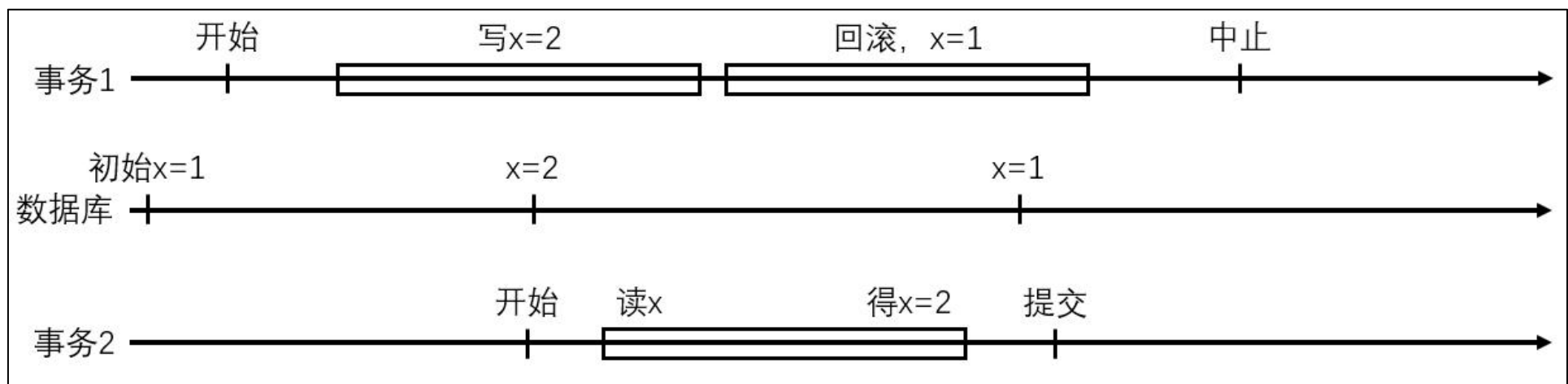


图6.1 脏读

锁 (Locking)

造成脏读的原因是数据被两个事务不加限制地同时访问和修改。为了避免这种情况，最基本的方法是对数据上锁使得同一时间只有一个事务可以写入数据。

“写锁” 和 “读锁”

- (1) “写锁”（排他锁或独占锁）：被施加“写锁”的数据将不可被其他事务写入或读取，即其他事务在写入或读取时申请的“写锁”和“读锁”都将被阻塞。
- (2) “读锁”（共享锁）：被施加“读锁”的数据不可被其他事务更改，但可被其他事务读取，即其他事务在写入前申请“写锁”时被阻塞，但读取前申请“读锁”时被允许。

	写锁	读锁	不上锁
写锁	×	×	√
读锁	×	√	√
不上锁	√	√	√

表6.2 锁的共存关系

锁的粒度

锁的粒度指封锁对象的单位。

- 对于不同的数据库系统，锁的粒度也不同。锁的粒度可以是记录的某个数据（字段），**一行记录**，**一个数据页**，**一个数据表**，甚至**整个数据库**。
- 锁粒度的大小对数据库的并发性和性能开销均有影响，锁粒度越大，维护锁的开销越小，但并发度越小。即并发度与锁粒度成反比。

2、读已提交 (Read Committed)

读已提交在保证数据库在读取数据时，只能读取已提交的数据，即保证没有脏写的同时保证了没有脏读。

实现方式：在修改数据前对数据上“写锁”（直到事务提交后再释放“写锁”），这样被上“写锁”的数据就不会被其他事务读取，保证了没有脏读。

举例：事务1在写 $x=2$ 之前上“写锁”，因为事务读取数据需上“读锁”的缘故，事务2在读 x 时无法上读锁，就无法读 $x=2$ ，防止了脏读。

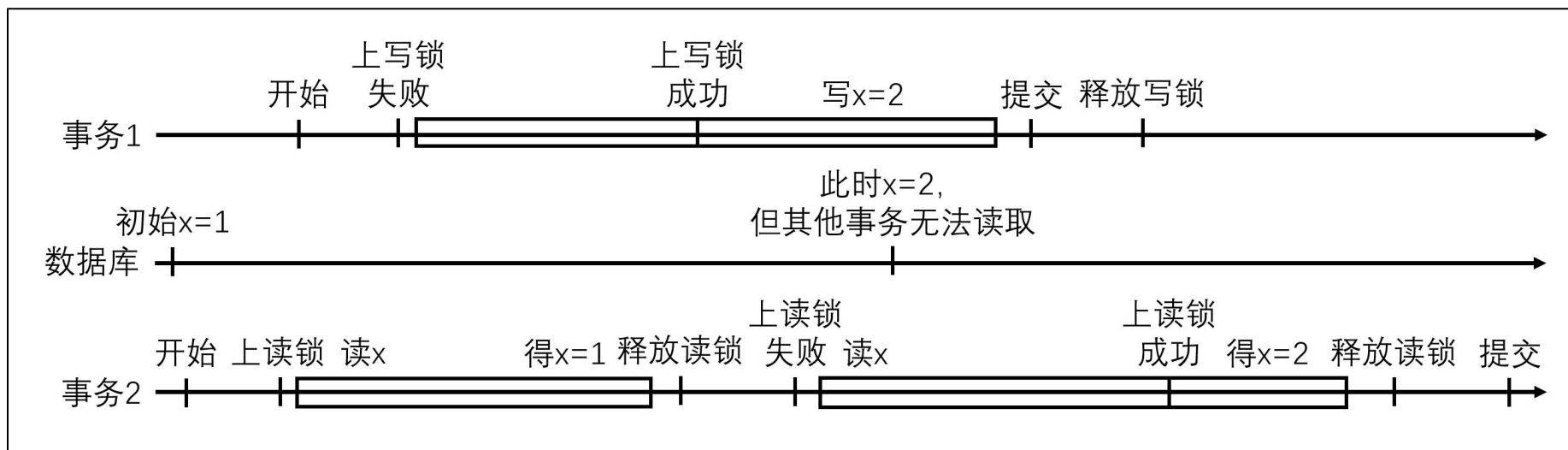


图6.2 读已提交示例

2、读已提交的问题——不可重复读

事务2两次读取x得到了不同的结果，这使得事务在执行期间两次读取的数据是不一致的，这可能导致**不可重复读**异常。

例如，在银行系统中，假设某用户当前账户余额为101元，该用户执行提现事务：首先读取账户余额，若账户余额大于100元，则提现100元，即将账户余额-100元。但在用户读取账户余额并释放“读锁”后，另一个手机话费自动扣款事务并发执行，扣除了50元手机话费，即将账户余额-50元，并成功提交。这时，账户余额已变成 $101-50=51$ 元，余额不足100元。之前的提现事务继续，执行读取余额并将余额-100元的操作，使得账户余额变为-49，违反了银行账户余额必须大于等于0的一致性约束。这样的错误就是由不可重复读导致的。

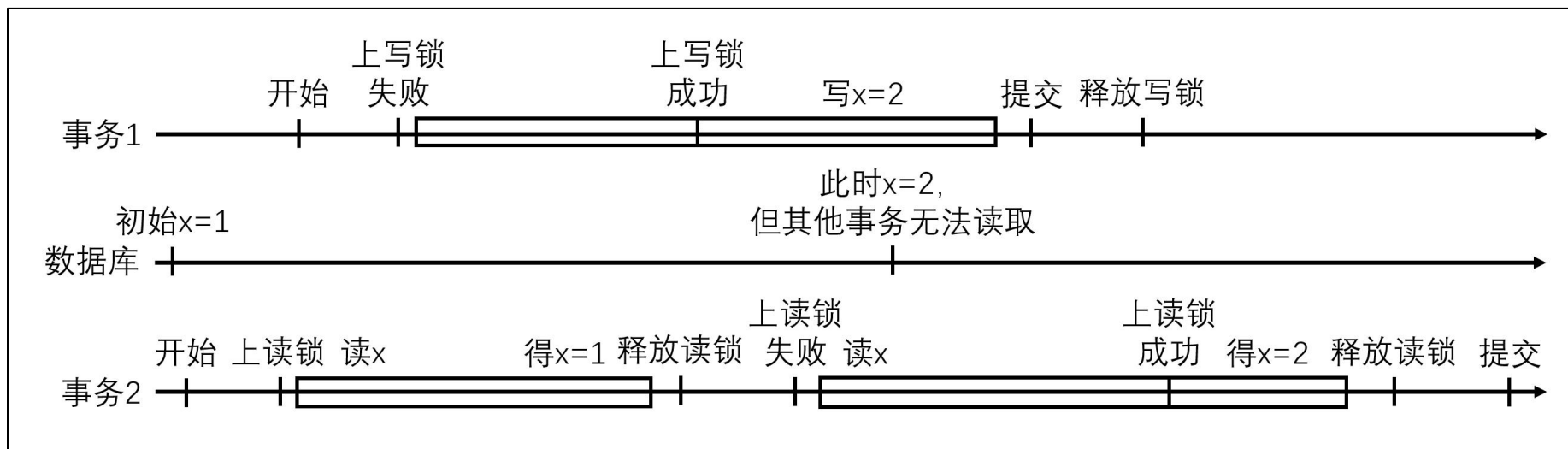


图6.2 读已提交示例

3、可重复读 (Repeated Read)

可重复读是在保证读已提交的基础上，保证一个事务在执行期间读到的数据始终是一致的。

实现方式：为了实现可重复读，事务对数据上的“读锁”必须要等到事务提交/中止才能释放。这样其他事务就不会在该事务两次读取操作之间修改数据。从而保证了可重复读。

举例：在事务2对x上“读锁”后，其他事务无法在事务2提交并释放锁前对x进行修改，因此事务2无论何时再对x进行读取都会得到同样的结果。

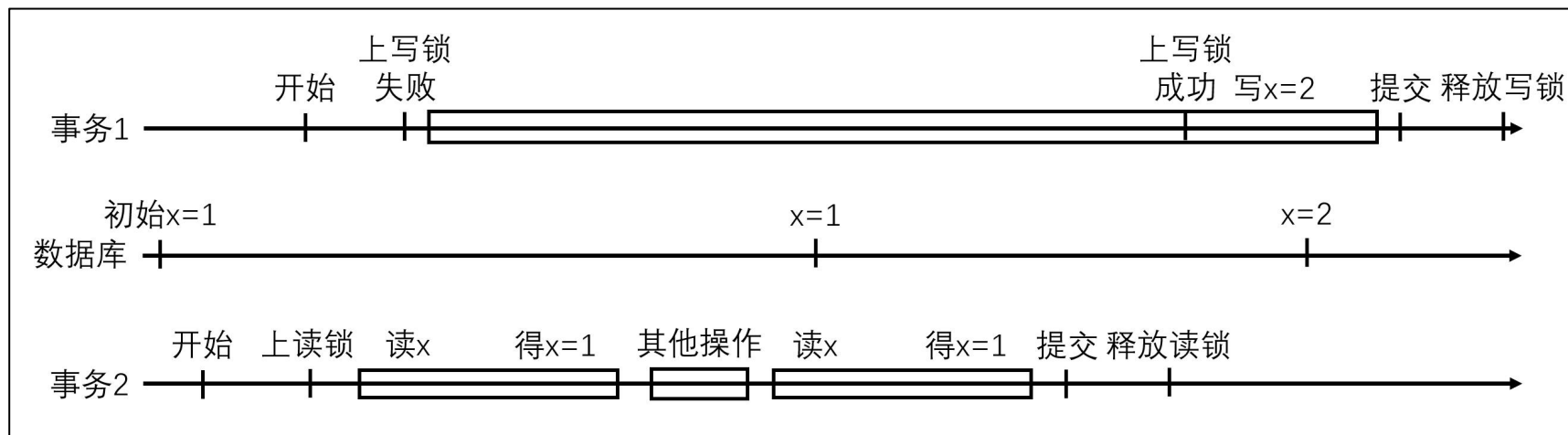


图6.3 可重复读示例

3、可重复读的问题——幻读（Phantom Read）

事务2先进行第一次范围查询，查询 $0 < id < 100$ 的所有行数据，返回三行数据A、B、C，之后事务2会对查询得到的行上“读锁”，保证可重复读隔离级别。这时事务1插入了 $id=4$ 的一行数据，因为原本数据库中并没有这行数据，因此没有事务会对该行数据上锁，故此次插入操作成功提交。之后事务2再查询 $0 < id < 100$ 范围的数据时会发现多出一行，事务2两次读取的结果不一样。这就是**幻读**异常。

解决方法：解决幻读的方法有很多，其中一种办法是加大上锁粒度，使用谓词锁。另一种办法是使用**多版本**来存储数据。

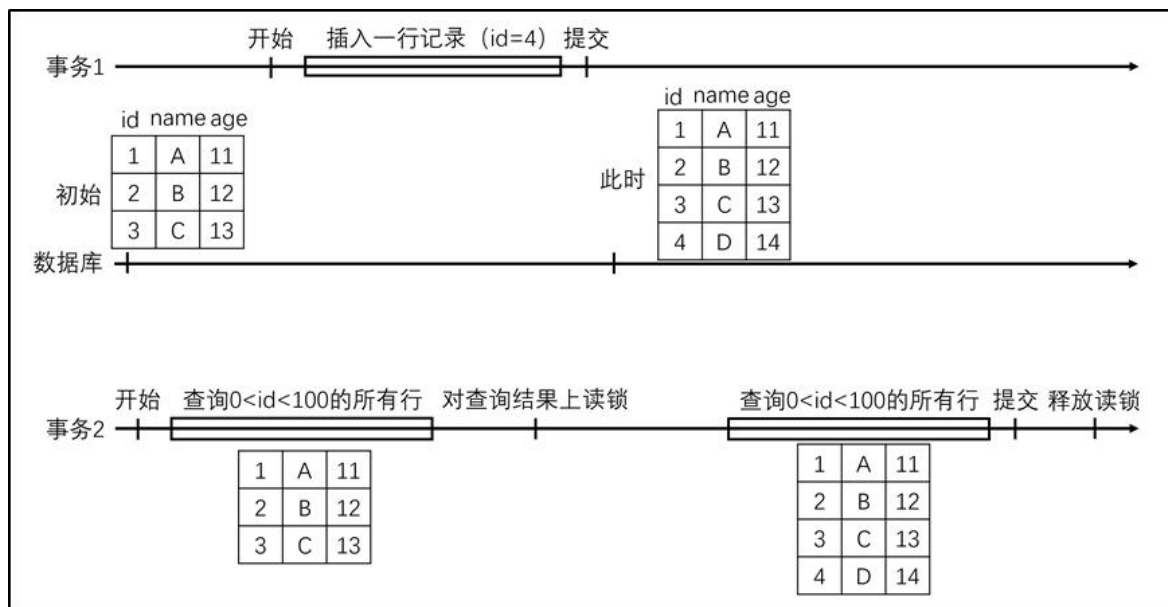
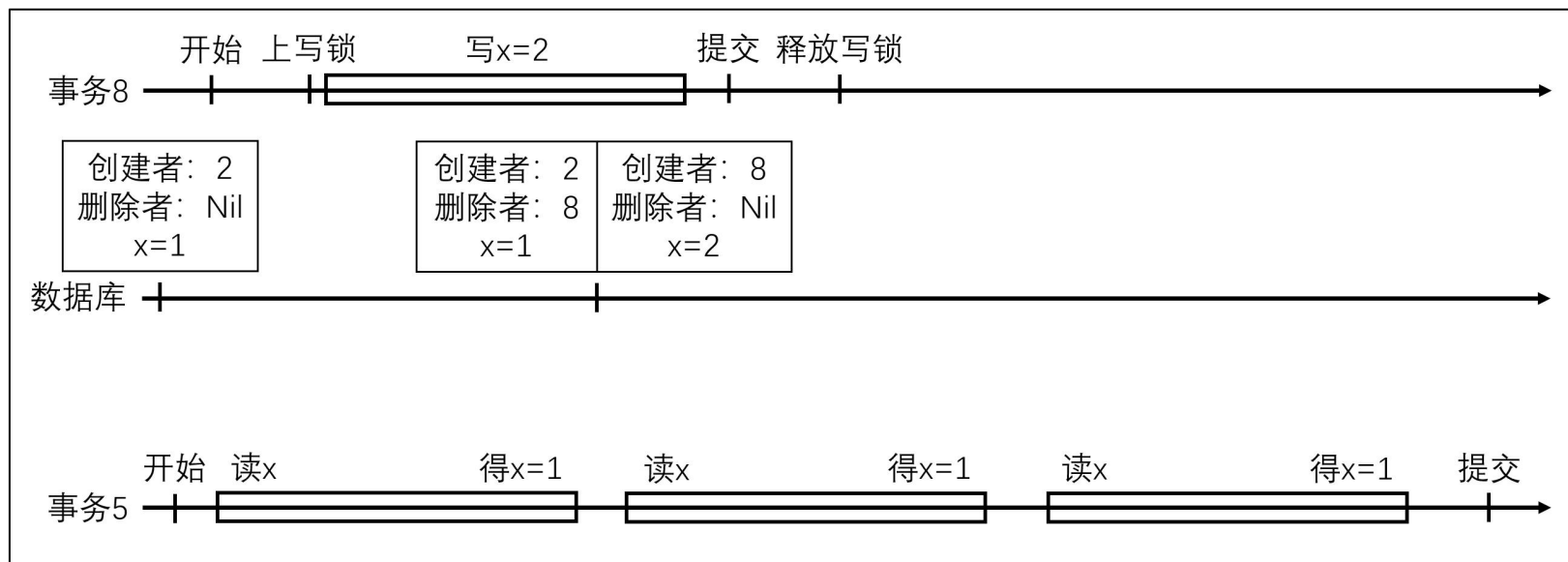


图6.4幻读

4、快照隔离 (Snapshot Isolation)

快照隔离指的是事务开始时会创建一个数据库的快照，并且事务在执行过程中只会基于该快照执行读取操作，不会看到其他事务在该事务开始后对数据的修改。

实现方法（多版本并发控制）：数据记录包含**创建者**和**删除者**字段。创建者字段记录插入该记录的tx_id，如果某事务更新或删除了该记录，那么将删除者记为该tx_id，而并非将其真正被删除。因为数据存在多个版本，读取不需上锁，不会被阻塞，事务根据自己的tx_id读取数据。既保证了**可重复读**，也避免了**幻读**。



4、快照隔离 (Snapshot Isolation)

举例：事务8将 $x=1$ 更新为 $x=2$ ，所以 $x=1$ 版本的数据的删除者是8。并且事务8新生成了 x 的一个数据版本 $x=2$ ，所以 $x=2$ 数据的创建者为8。而 $x=2$ 是最新版本的数据，所以它的删除者是空。这样事务5在读取数据时，无论是否有其他事务更新数据，事务5都会读取创建者为2、删除者为8的那一版本的数据，因为 tx_id 按照时间顺序递增，事务5不应该看到 tx_id 大于5的数据版本。因此，事务5的三次读取结果均为 $x=1$ 。并且事务5的读操作不会阻塞事务8的写操作，事务8的写操作也不会阻塞事务5的读操作。

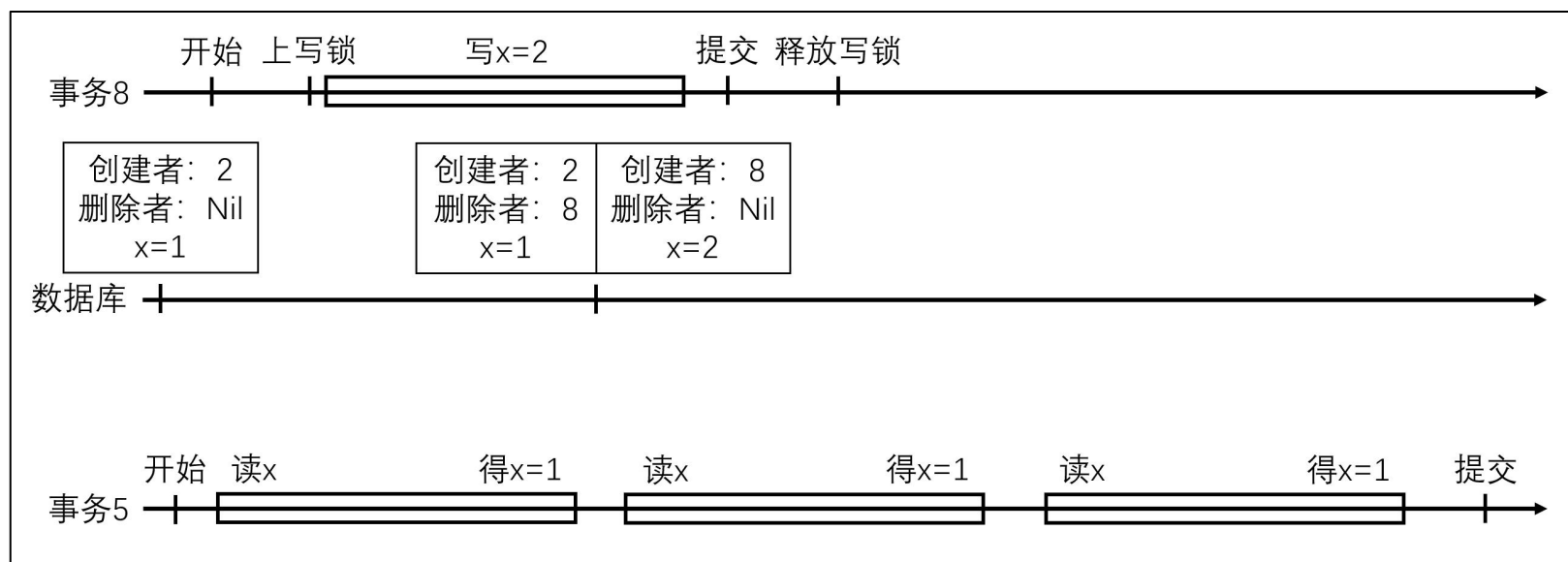


图6.5 快照隔离示例

5、可串行化 (Serializable)

可串行化指的是当多个事务并发执行时，最终结果与某个串行执行的情况相同。

举例：事务1读x写y，而事务2读y写x，那么最终事务1、事务2的读取结果要与两事务按某一顺序串行执行的结果相同，即一旦事务1读取了事务2修改的数据，则事务2就不能读取到事务1修改的数据，反之亦然。

在所有隔离级别中，可串行化是所有隔离级别中最强的，它在单机环境下不存在任何并发异常，因此ACID特性中的隔离性一般指的就是可串行化。

在分布式场景下，隔离级别与多副本一致性级别混合到一起会使事务并发变得更加复杂，在学术界有若干工作对此进行深入研究，本课程不再对此做深入介绍。

第六章、分布式事务

01

6.1 事务的基本概念

02

6.2 分布式事务与并发控制

03

6.3 分布式提交

04

6.4 分布式事务处理系统



6.2 分布式事务与并发控制

01

6.2.1 悲观并发控制

02

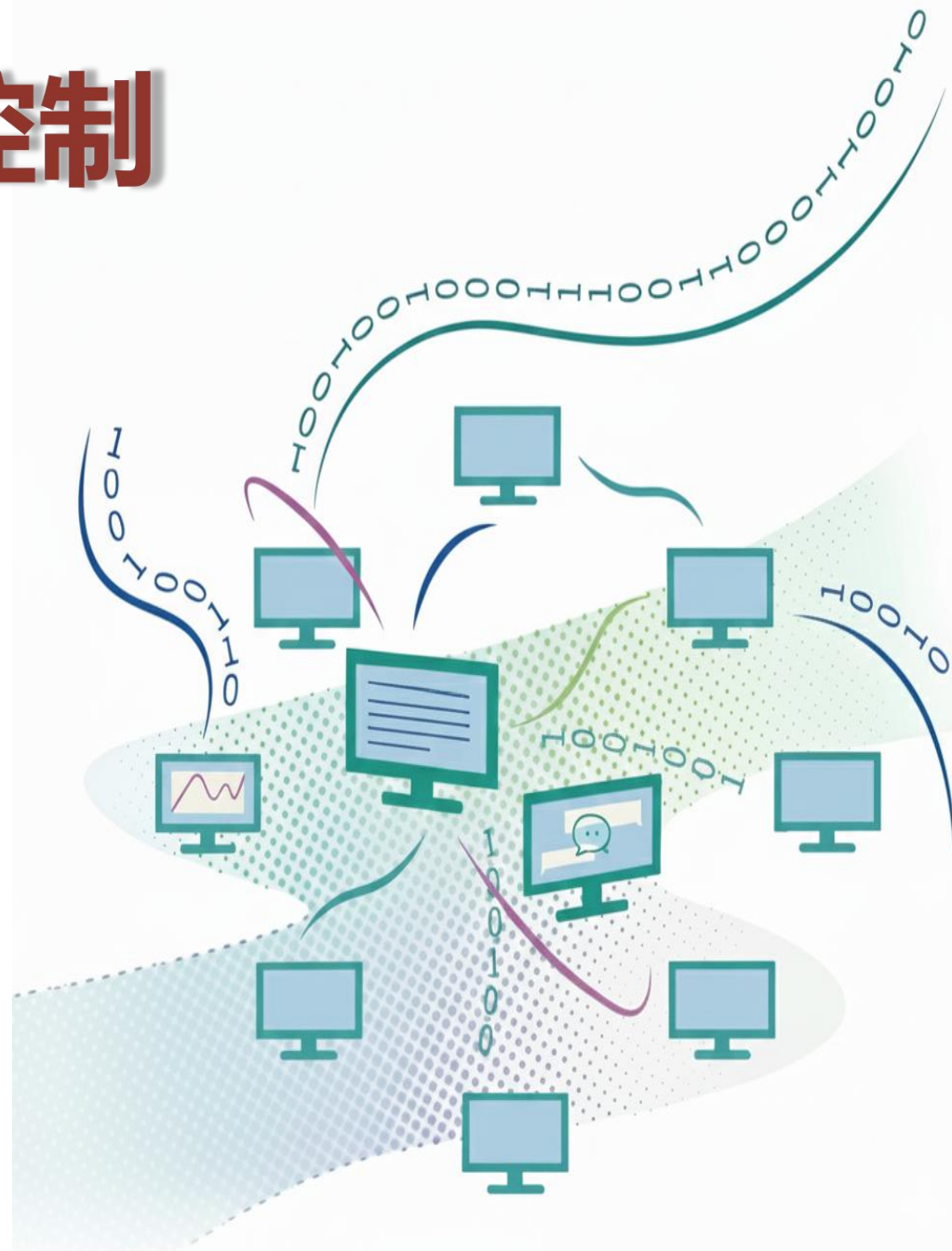
6.2.2 基于时间戳的并发控制

03

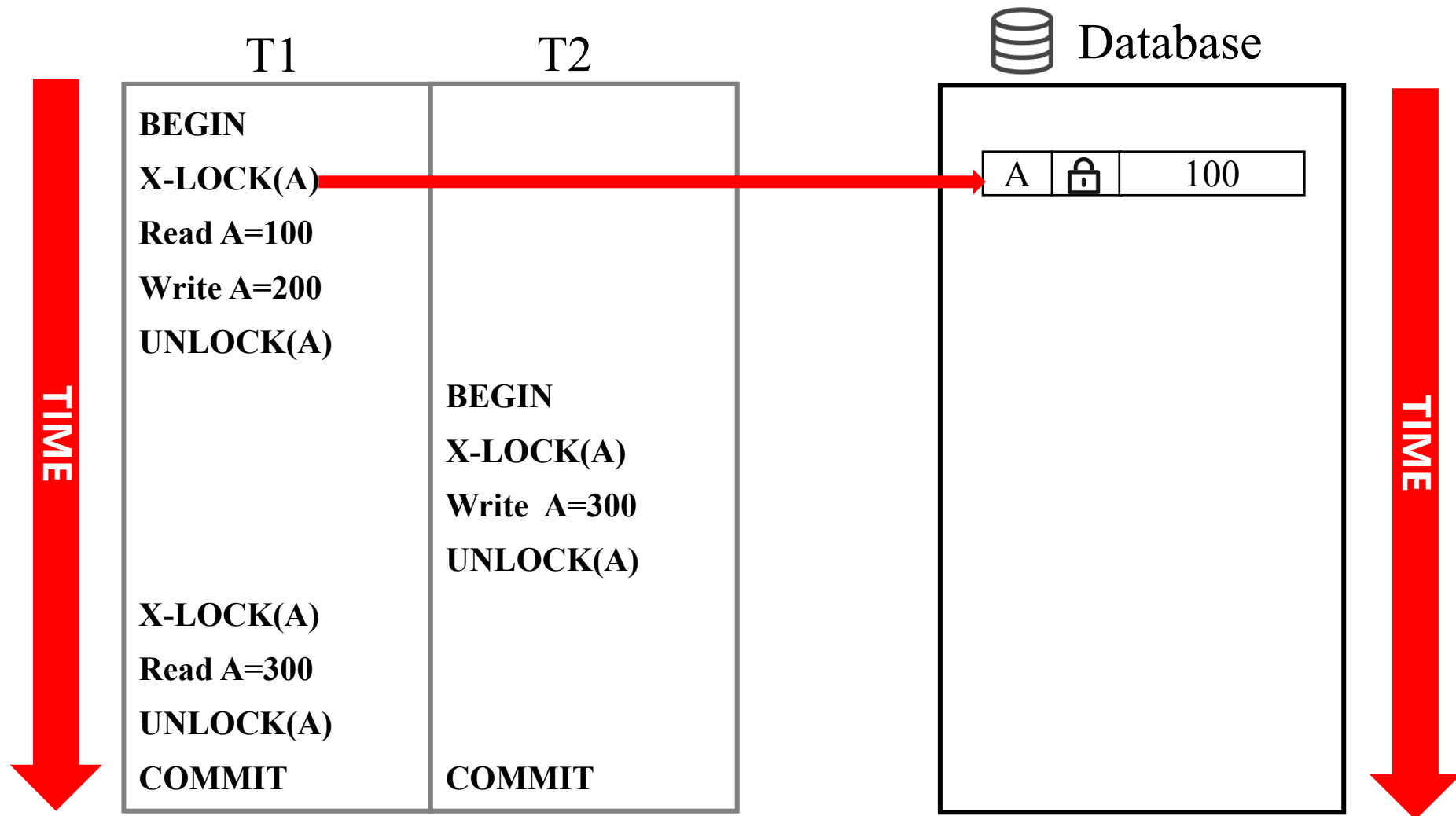
6.2.3 乐观并发控制

04

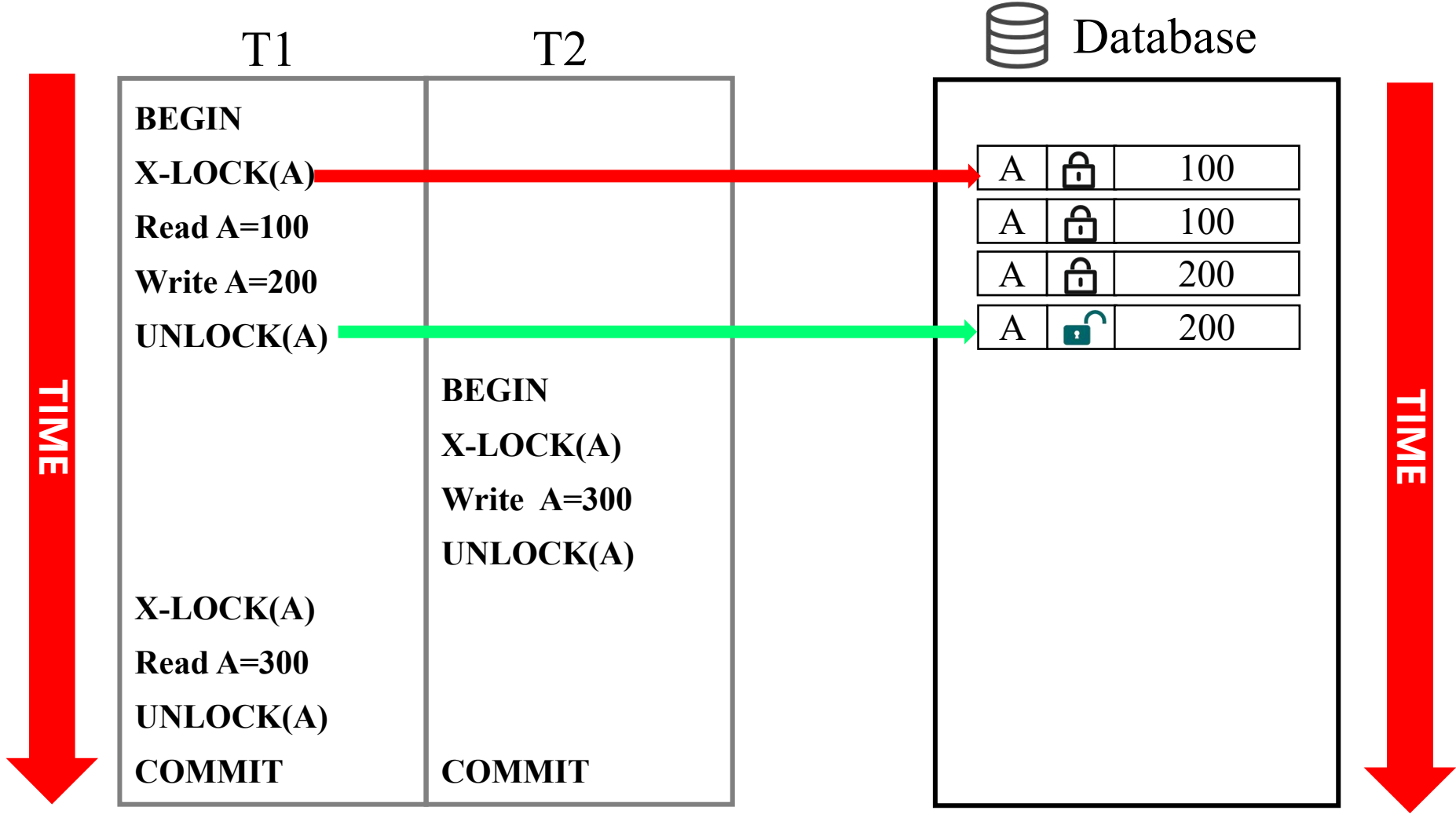
6.2.4 多版本并发控制



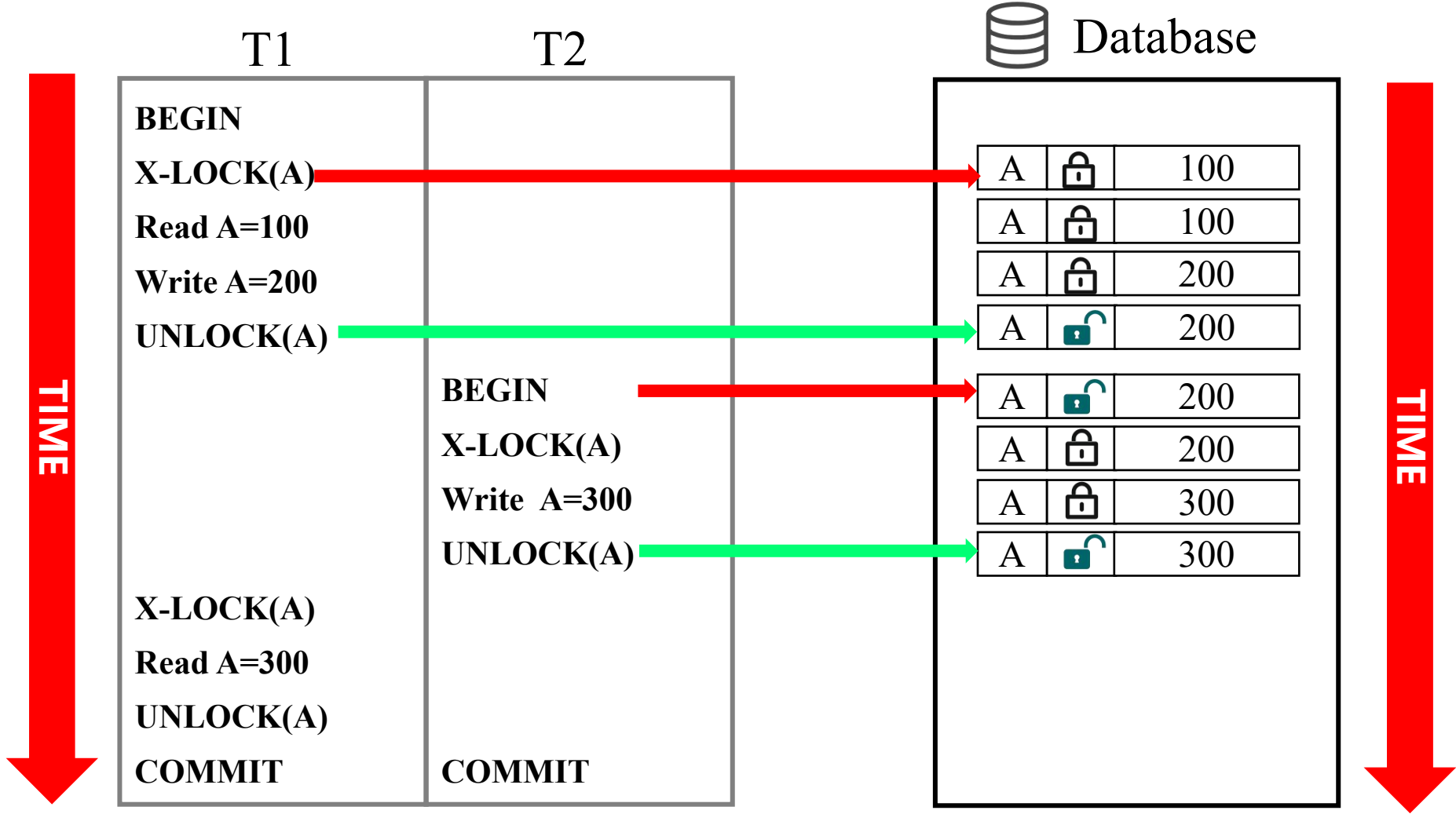
封锁协议的问题：丢失修改



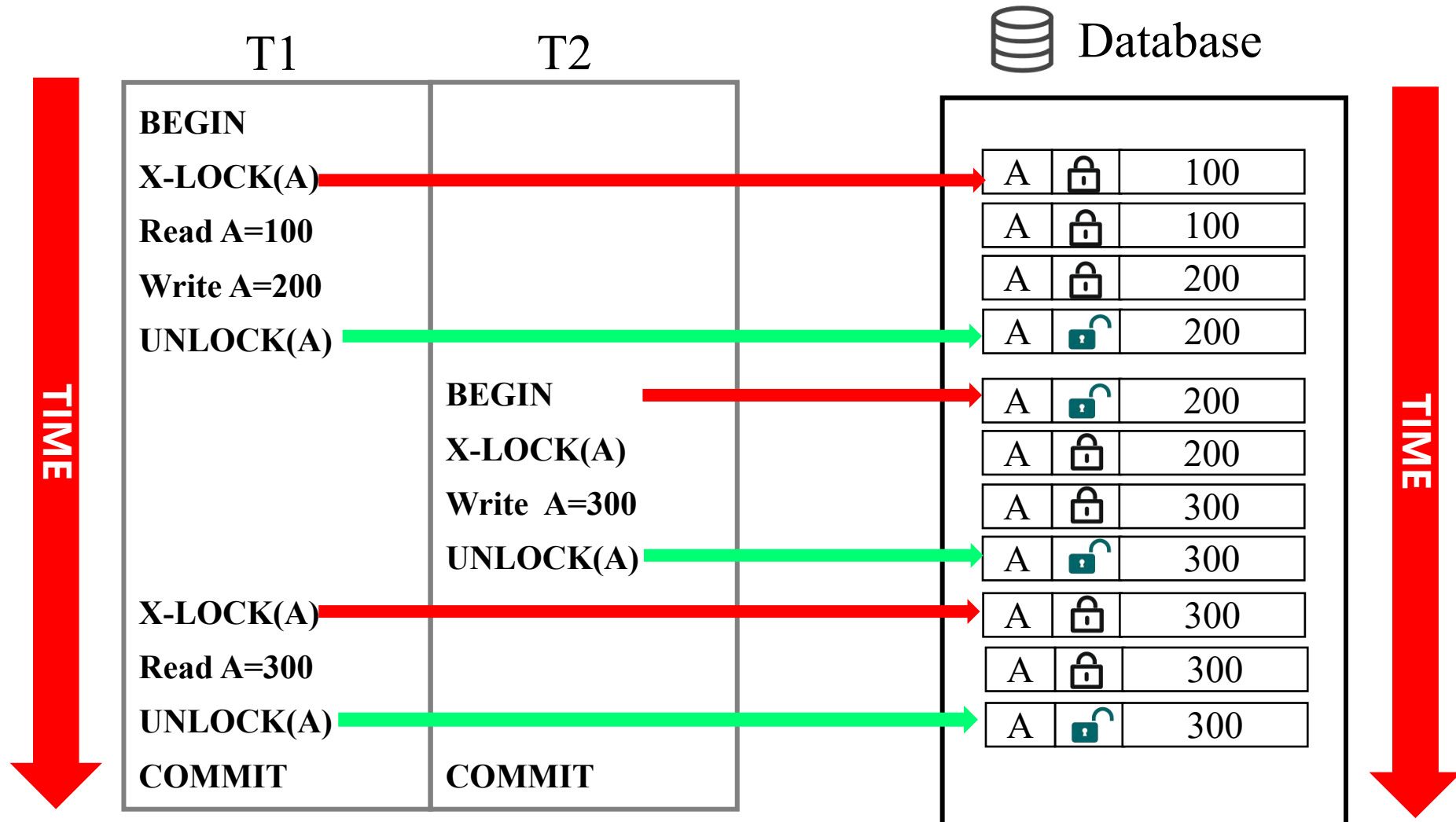
封锁协议的问题：丢失修改



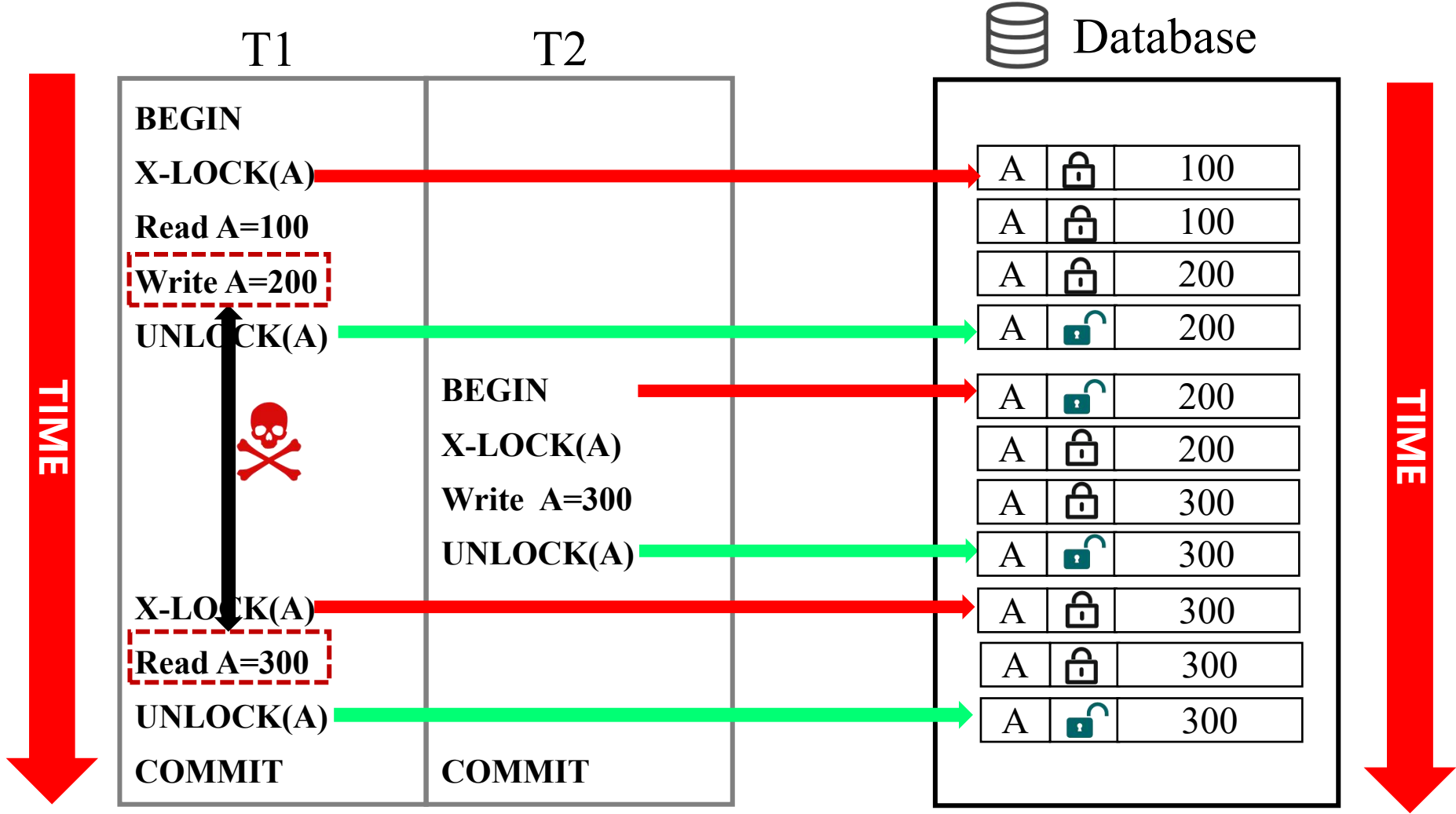
封锁协议的问题：丢失修改



封锁协议的问题：丢失修改



封锁协议的问题：丢失修改



6.2.1 悲观并发控制

基本的两阶段封锁协议 (Two Phase Locking, 2PL)

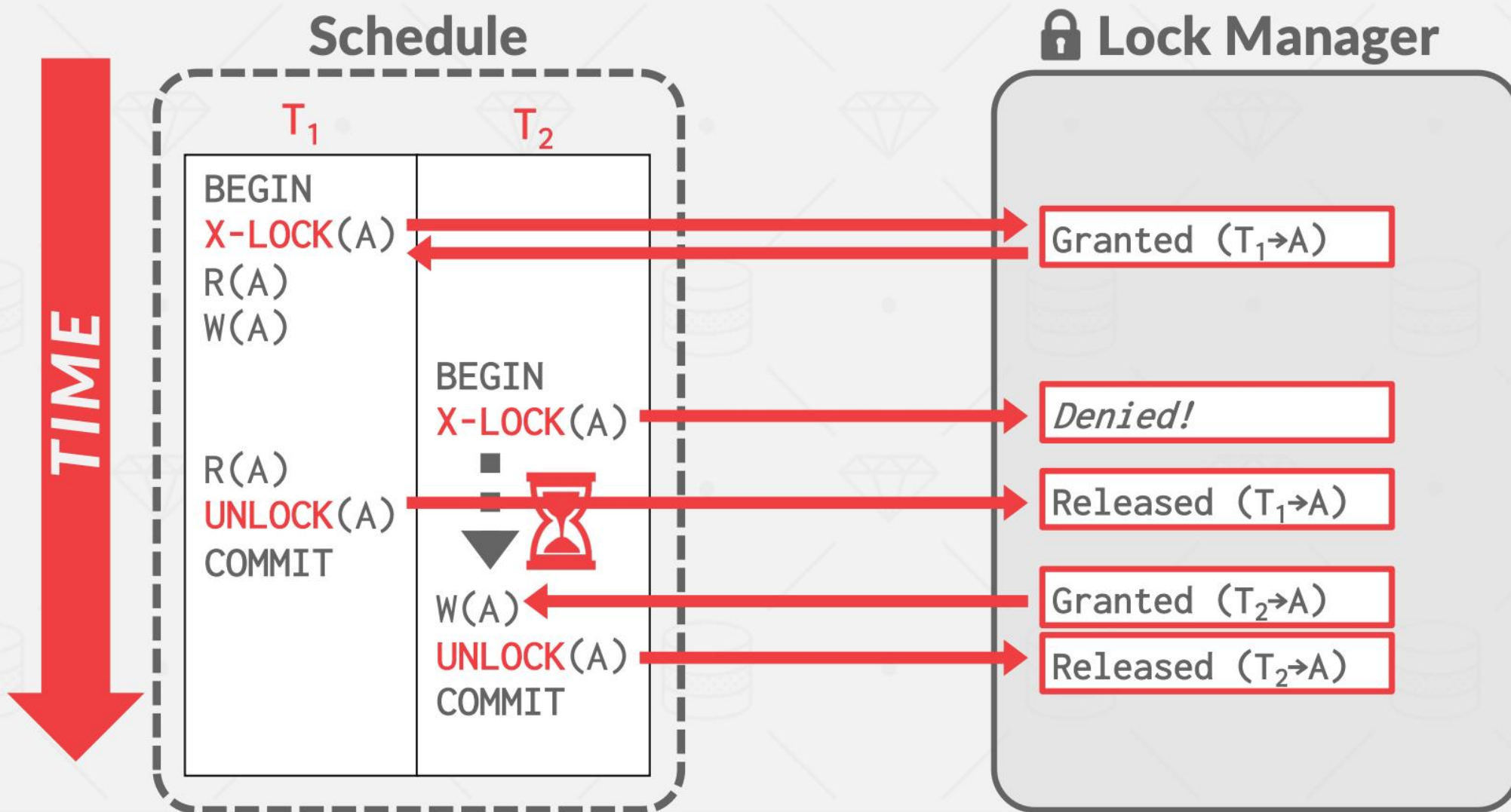
两阶段封锁将事务分为两个阶段，上锁阶段和解锁阶段。具体规则描述如下：

上锁阶段（也称为扩展阶段）：在这个阶段，事务可以申请获得操作数据项上的任何类型的锁，但是不能释放锁。若出现并发冲突则等待，获得锁后执行事务操作，并在进行下一个数据操作前继续申请锁。

解锁阶段（也称为收缩阶段）：在这个阶段，事务可以释放该事务操作数据项上的任何类型的锁，但是不能再申请任何锁。

下页图对照上面图进行修改，说明2pl怎么避免丢失修改问题

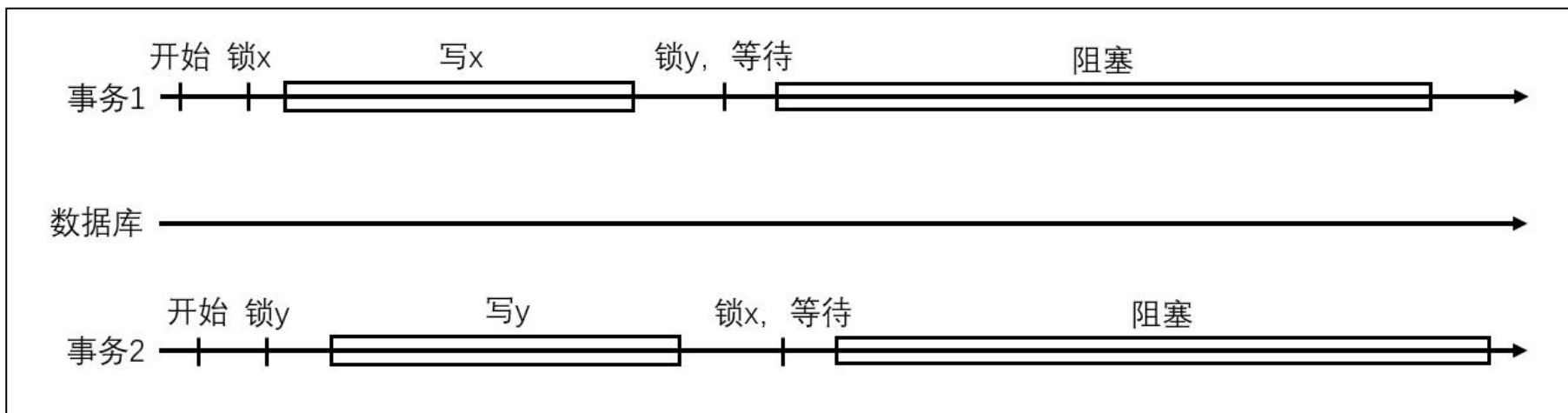
EXECUTING WITH 2PL



死锁

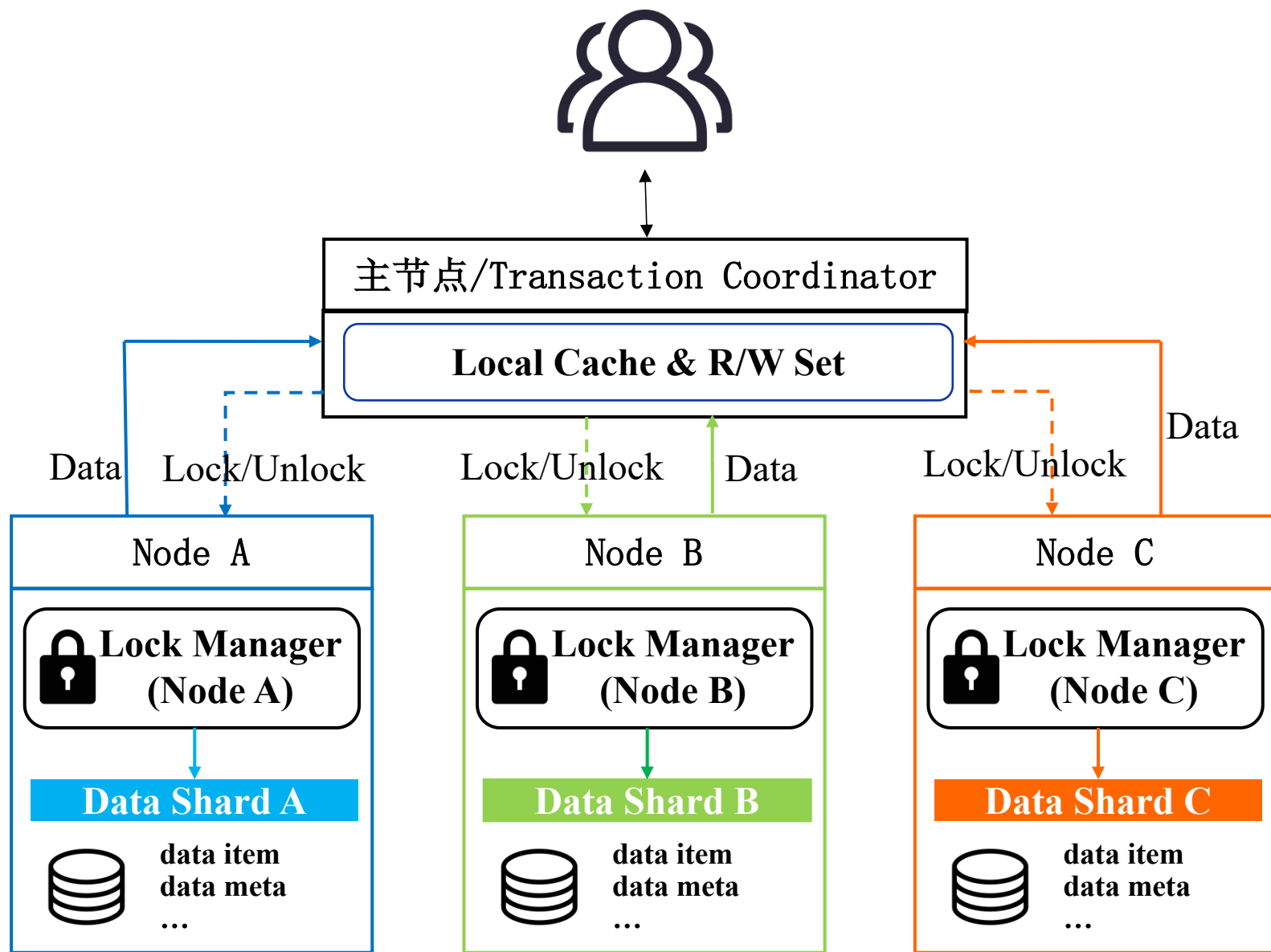
事务并发执行过程中，每个正在执行的事务都会持有其操作数据的锁，在其释放持有的锁之前，可能需要申请其他事务已上锁的数据，那么该事务将被阻塞。若所有并发执行的事务均处于被阻塞状态，并等待其他事务释放锁而获得所需资源，则此时事务执行进入死锁状态。

举例：事务1对x上锁后请求对y上锁，而事务2对y上锁后请求对x上锁，这时两个事务都被对方阻塞并等待对方释放锁，这种情况称为死锁。下面介绍两种解决死锁的方法，死锁检测和死锁避免。



分布式悲观并发控制

- 数据被分片放置在多个节点上，每个节点负责管理一个数据分片。一个事务可能涉及读写多个数据分片上的数据，即跨节点事务。
- 主节点从各数据分片上读取所需数据到本地缓存，执行事务生成读写集。
- 在封锁阶段，在读取数据的同时可以让负责该数据项的节点对其上读锁，将写入的数据发送到负责该数据的节点上，先上写锁和完成远程写入。
- 在解锁阶段，客户端向该事务涉及的相关节点发送释放锁的指令。



6.2 分布式事务与并发控制

01

6.2.1 悲观并发控制

02

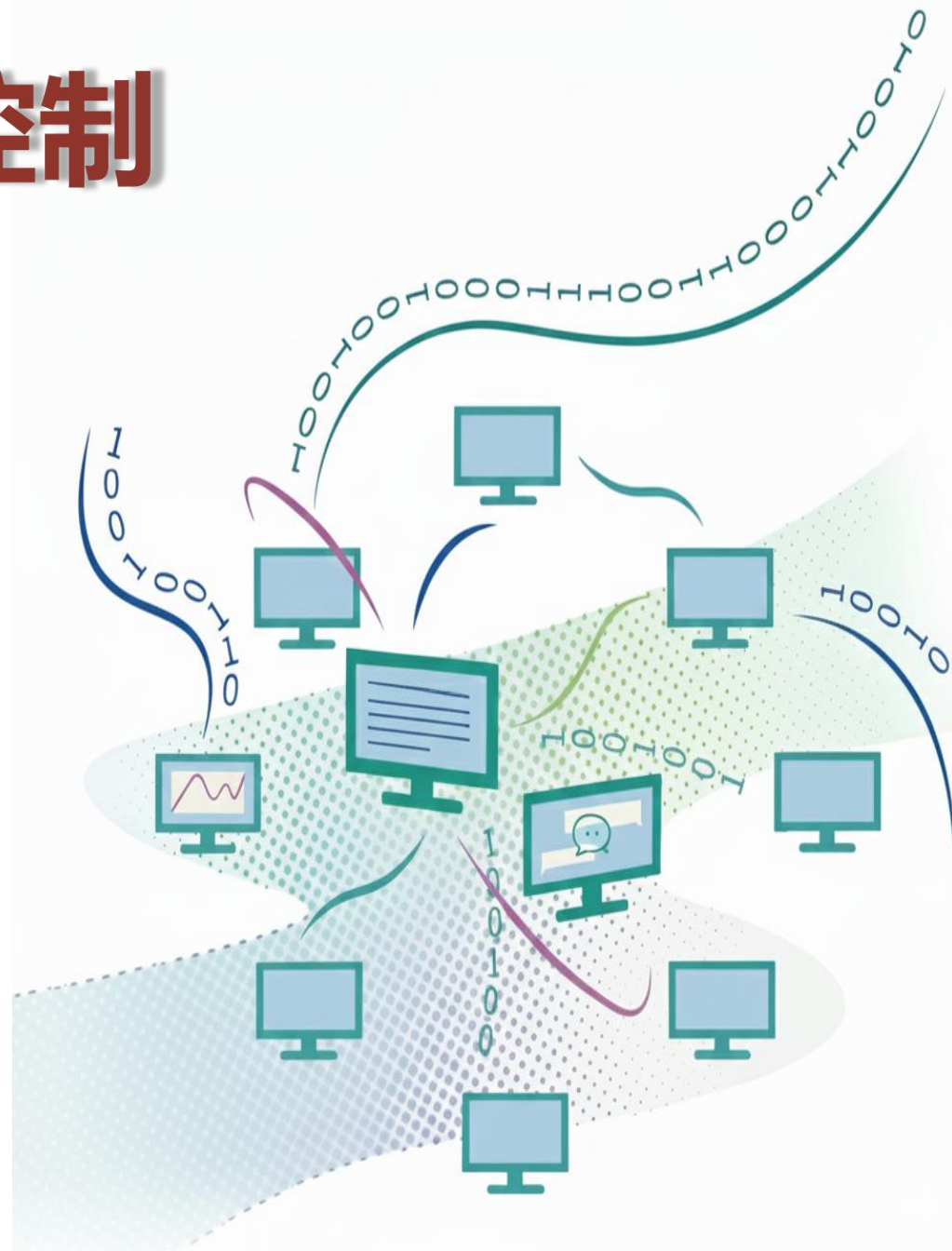
6.2.2 基于时间戳的并发控制

03

6.2.3 乐观并发控制

04

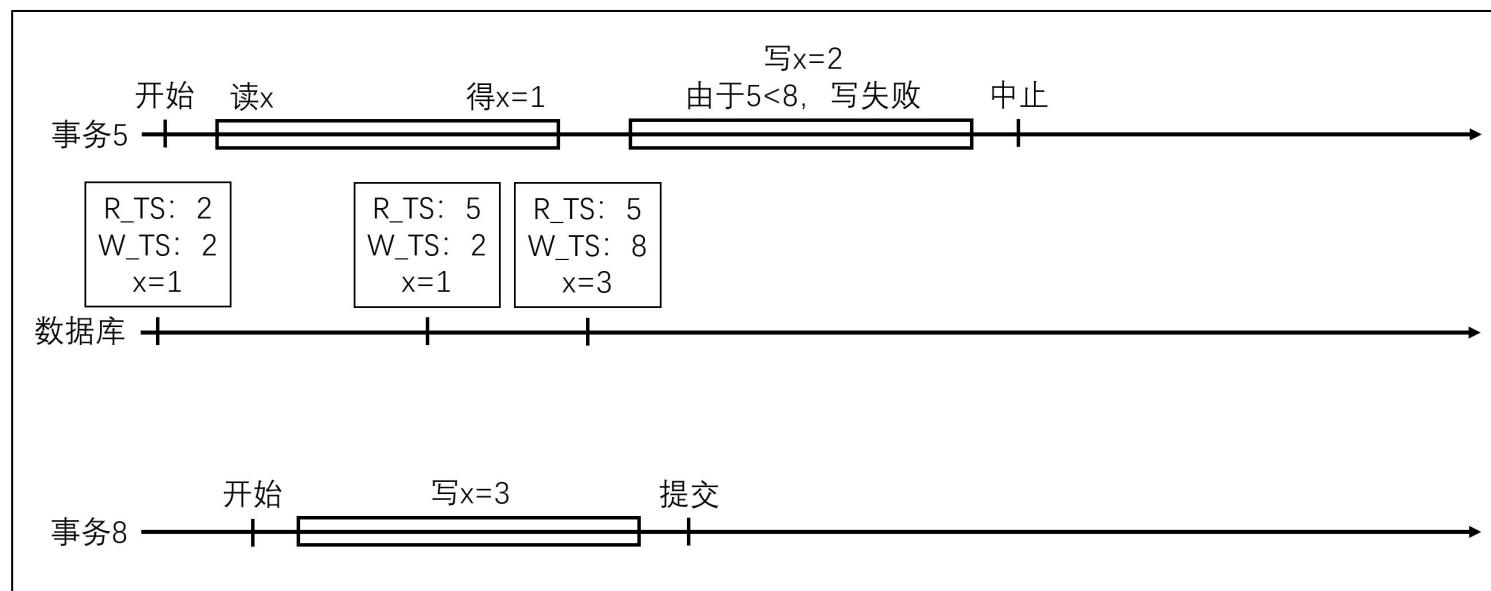
6.2.4 多版本并发控制



基本时间戳排序协议 (Timestamp Ordering (TO) Protocol)

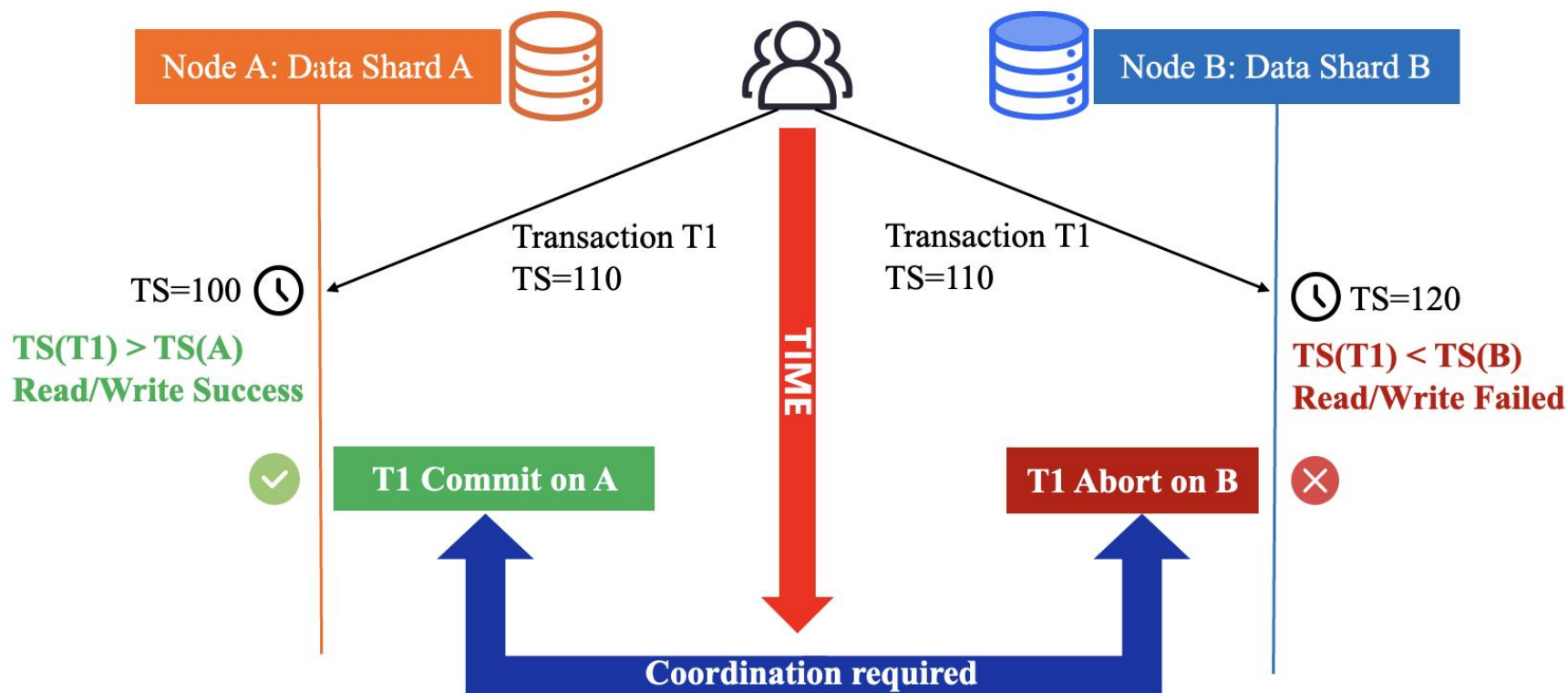
数据项上带有两个时间戳，读时间戳 (R_TS) 和写时间戳 (W_TS)。

举例：初始时，数据项 $x=1$ ，其读时间戳 $R_TS=2$ ，写时间戳 $W_TS=2$ 。而后事务5先开始执行，即 $TS(T5)=5$ ，首先进行读操作，判断得 $TS(T5) > R_TS(x)$ ，成功读取 $x=1$ ，并将 x 的 R_TS 修改为 $TS(T5)=5$ 。而后事务8开始执行，即 $TS(T8)=8$ ，并进行写操作，判断得 $TS(T8) > R_TS(x)$ 且 $TS(T8) > W_TS(x)$ ，成功写 $x=3$ ，并将 x 的 R_TS 和 W_TS 均修改为 $TS(T8)=8$ ，并成功提交。这时，事务5进行写操作，判断得 $TS(T5) < W_TS(x)$ ，故中止事务5。



分布式时间戳并发控制

在分布式场景下，多个节点维护多个数据分片，一个分布式事务可能涉及对多个数据分片上的读写操作，即跨节点事务。比如，在数据分片A上因为时间戳排序靠前提交成功，在数据分片B上因为时间戳排序靠后而被中止，这样就涉及两个节点的协调处理，这部分内容涉及分布式提交技术，将在6.3节详细阐述。



6.2 分布式事务与并发控制

01

6.2.1 悲观并发控制

02

6.2.2 基于时间戳的并发控制

03

6.2.3 乐观并发控制

04

6.2.4 多版本并发控制

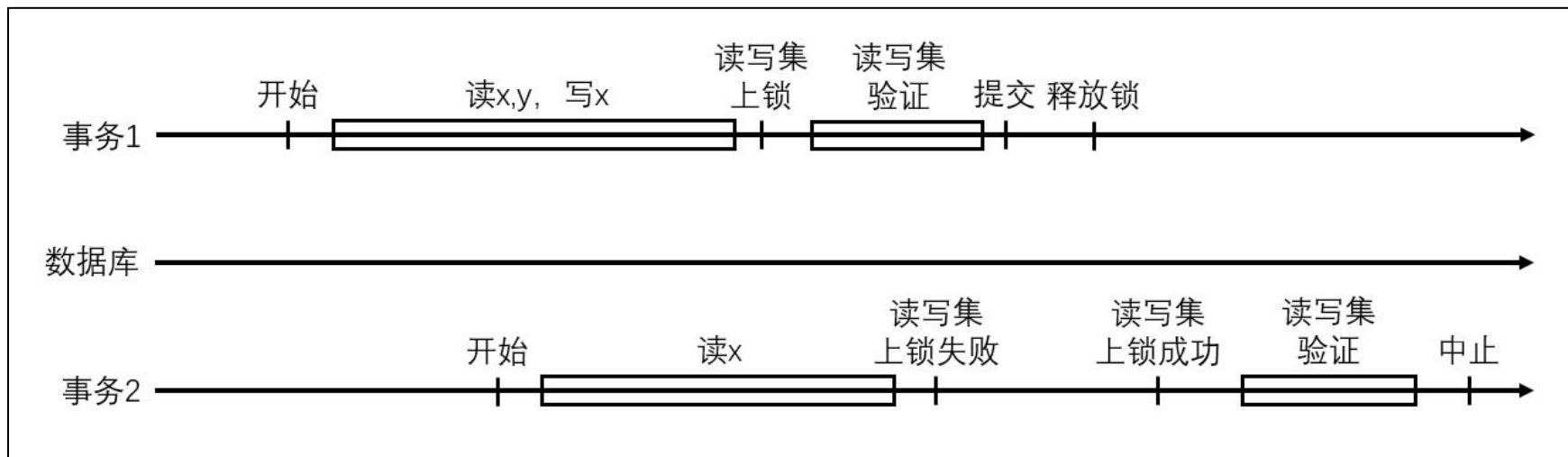


乐观并发控制算法 (Optimistic Concurrency Control, OCC)

事务执行乐观并发控制过程包括三个阶段：执行阶段、验证阶段、提交阶段。

举例：事务1首先乐观地执行读写操作并成功提交。而事务2在乐观读取数据x后进行验证时发现其读集（数据x）被事务1修改，则验证失败并中止。

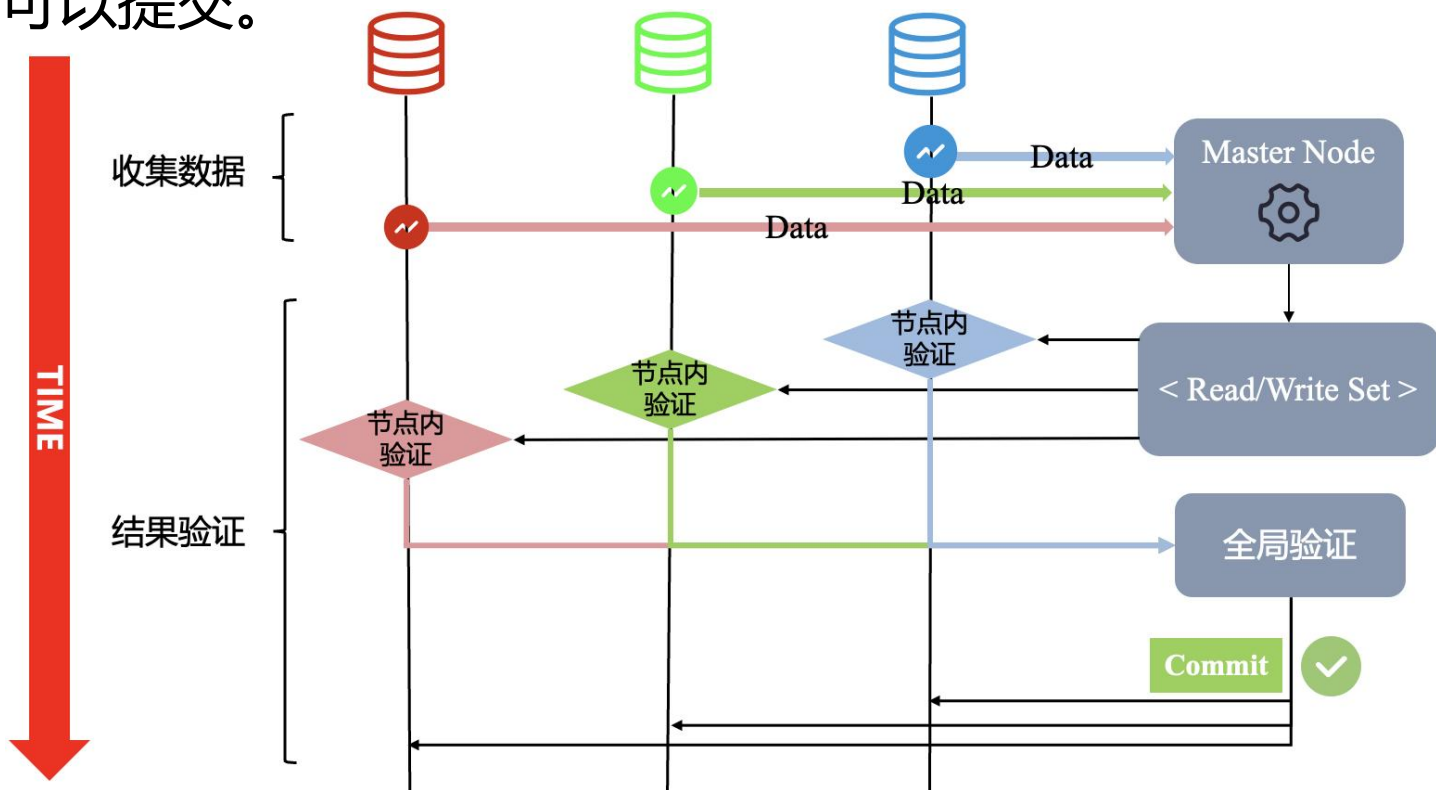
乐观并发控制在事务执行阶段不上锁，先允许事务穿插进行没有预防措施，从而增加了并发度，允许多个事务同时访问同一数据。然后只在验证阶段进行冲突检测，判断是否又事务穿插进行的情况，再中止可能会影响可串行化隔离的事务。



分布式乐观并发控制

在分布式场景下，多个数据分片分布在多个节点上。客户端在验证阶段会将读写集发送给负责该数据的相关节点，各个节点将收到的读集和数据的最新版本进行对比，验证是否一致，若不一致说明该数据被其他事务修改过，因此会返回给客户端需要中止该事务的请求；若一致则说明可以提交。

- 客户端接收所有节点的验证结果进行全局验证，决定最终是否可以提交（所有节点都通过验证）
- 全局验证可确保如果两个冲突的事务在所有节点上有相同的验证结果，或者同时提交成功或者同时被中止，来保证事务原子性



6.2 分布式事务与并发控制

01

6.2.1 悲观并发控制

02

6.2.2 基于时间戳的并发控制

03

6.2.3 乐观并发控制

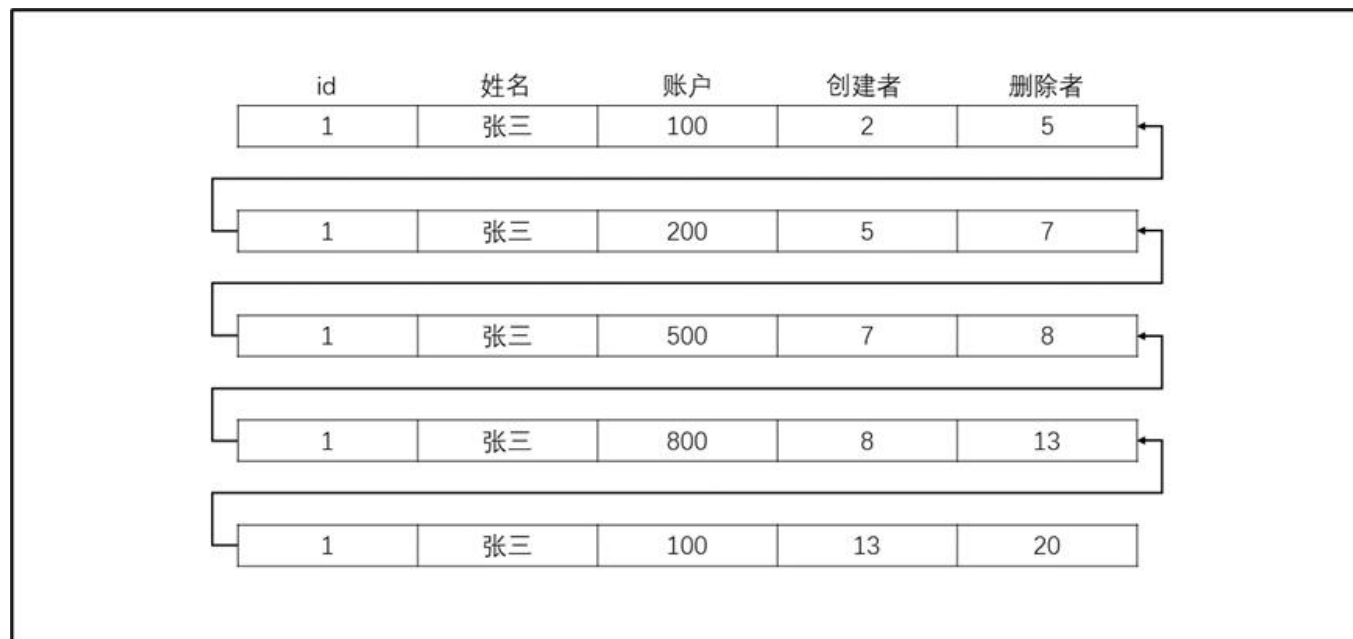
04

6.2.4 多版本并发控制



6.2.4 多版本并发控制 (Multi-Version Concurrency Control, MVCC)

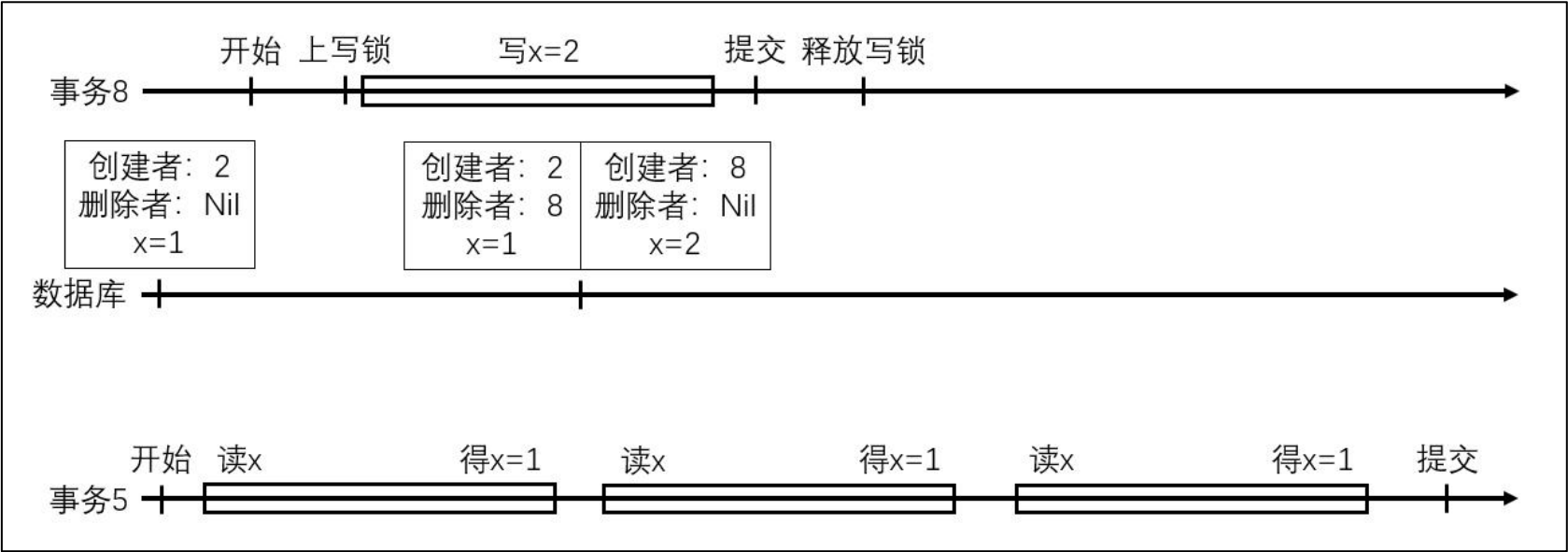
在MVCC中，一个数据项可能会存在多个版本，以多版本形式存储的数据除了键值字段，还包含**创建者字段**和**删除者字段**。如下图。这些版本是由事务在执行过程中，当事务对数据库进行插入或更新操作时，事务会拷贝旧版本，并将数据更新后形成新版本，然后向前链接指向旧版本，这样就形成了一个数据的多版本链。同一个数据的多个版本形成的一个版本链，这个版本链是以最新数据为头的链表。



多版本悲观并发控制 (MV2PL)

在MV2PL中，事务进行写操作之前，必须对要修改的数据上“写锁”。并且遵循两阶段封锁规定，即事务开始释放锁后就不能再对数据上锁。

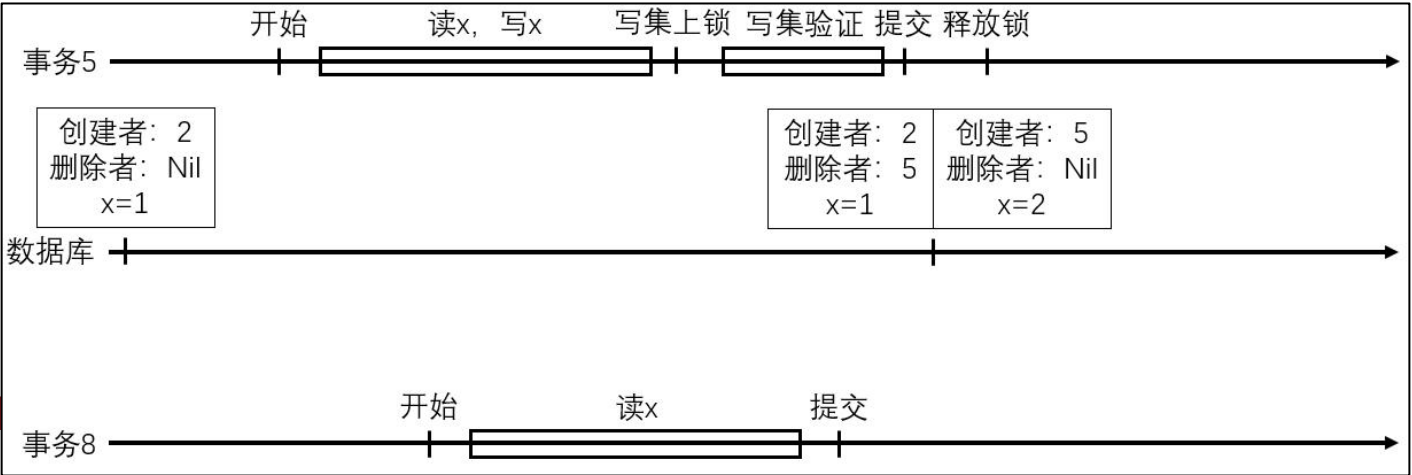
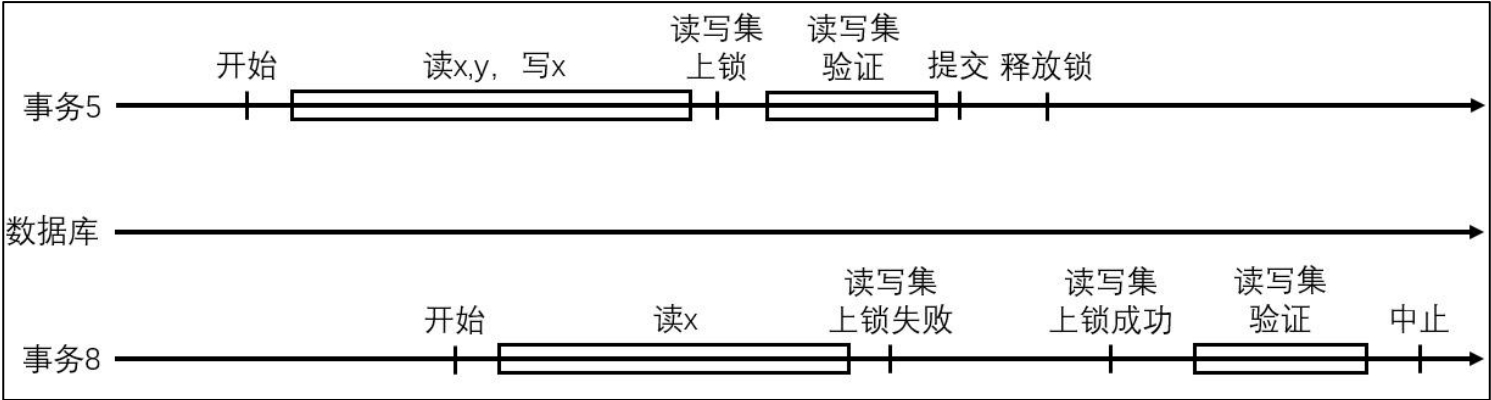
事务8在写操作之前需对数据上锁，并在提交后释放。事务5进行读操作时不需要上锁，这样即使该数据被其他事务持有“写锁”，读操作也不会被阻塞。



多版本悲观并发控制 (MV2PL)

多版本乐观并发控制 (MVOCC)

当乐观并发控制与多版本相结合，就可以将读写集验证弱化为写集验证，从而使得只读事务可以无需验证。若采用乐观并发控制，则事务8将因为读集验证失败中止，而采用多版本乐观并发控制 (MVOCC) 的方法，则可以提交成功。乐观并发控制下，读写冲突和写写冲突均会导致事务中止。而多版本乐观并发控制下，只有写写冲突会导致事务中止，而读写冲突则不会。



第六章、分布式事务

01

6.1 事务的基本概念

02

6.2 分布式事务与并发控制

03

6.3 分布式提交

04

6.4 分布式事务处理系统



6.3 分布式提交

01

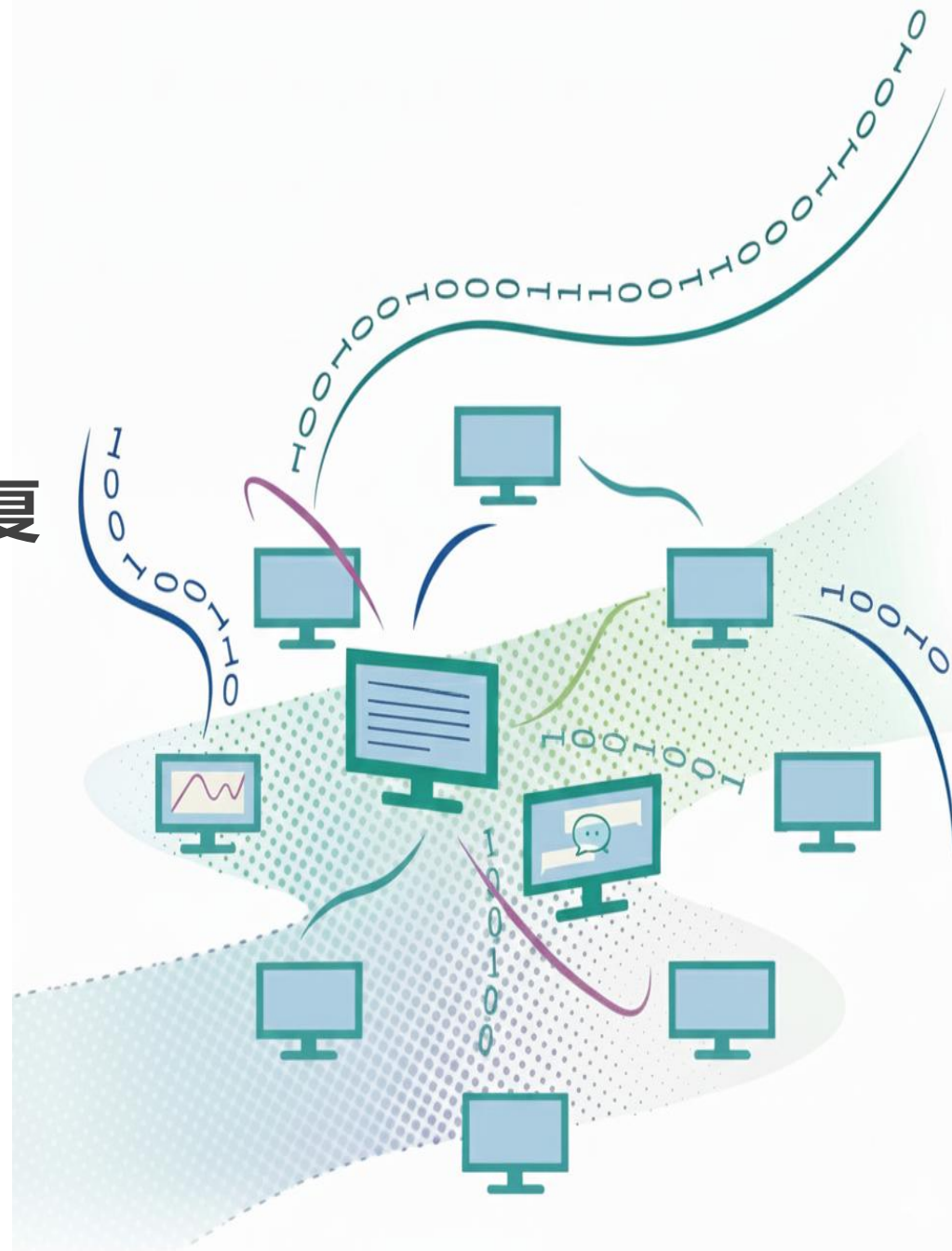
6.3.1 两阶段提交协议

02

6.3.2 两阶段提交协议的故障恢复

03

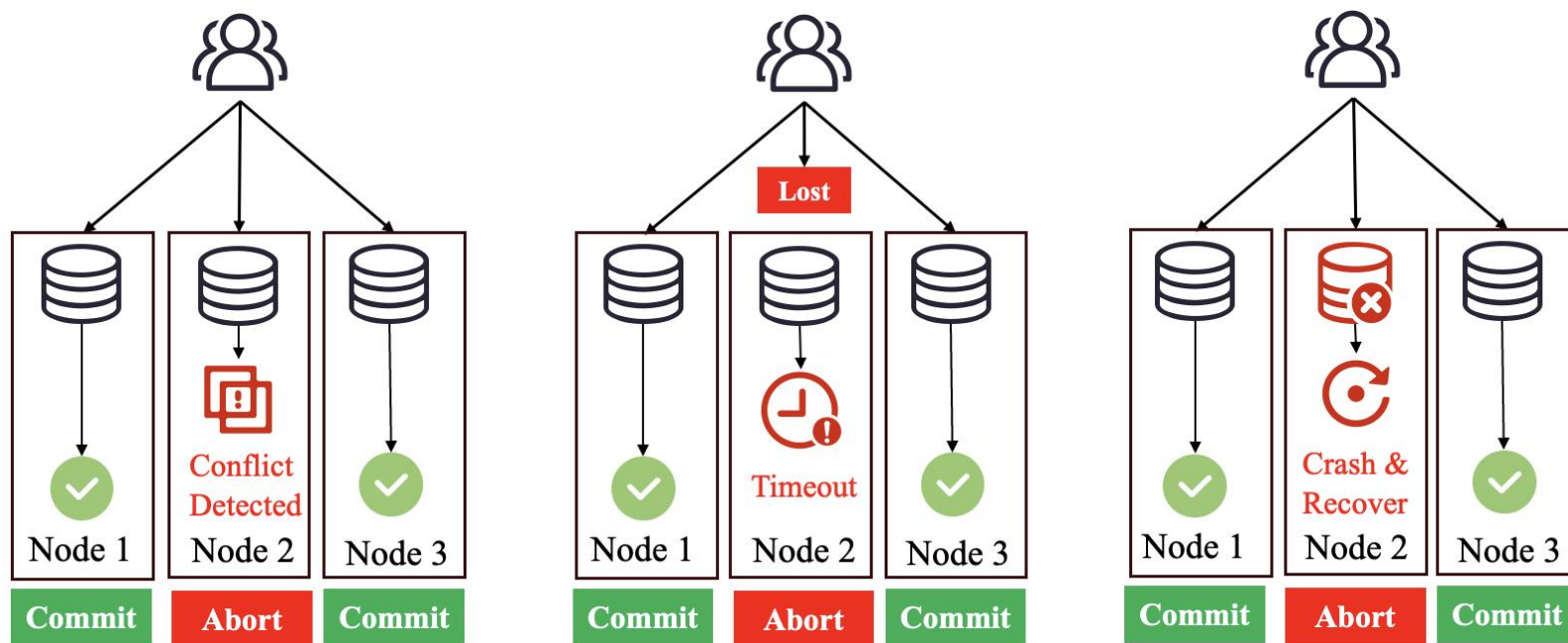
6.3.3 两阶段提交与副本共识



分布式事务的原子提交问题

用户向所有节点发出提交命令，各节点收到命令后各自独立地开始提交子事务，并分别在提交或中止后返回用户。以下情况导致违反原子性，而提交的子事务是无法撤回的：

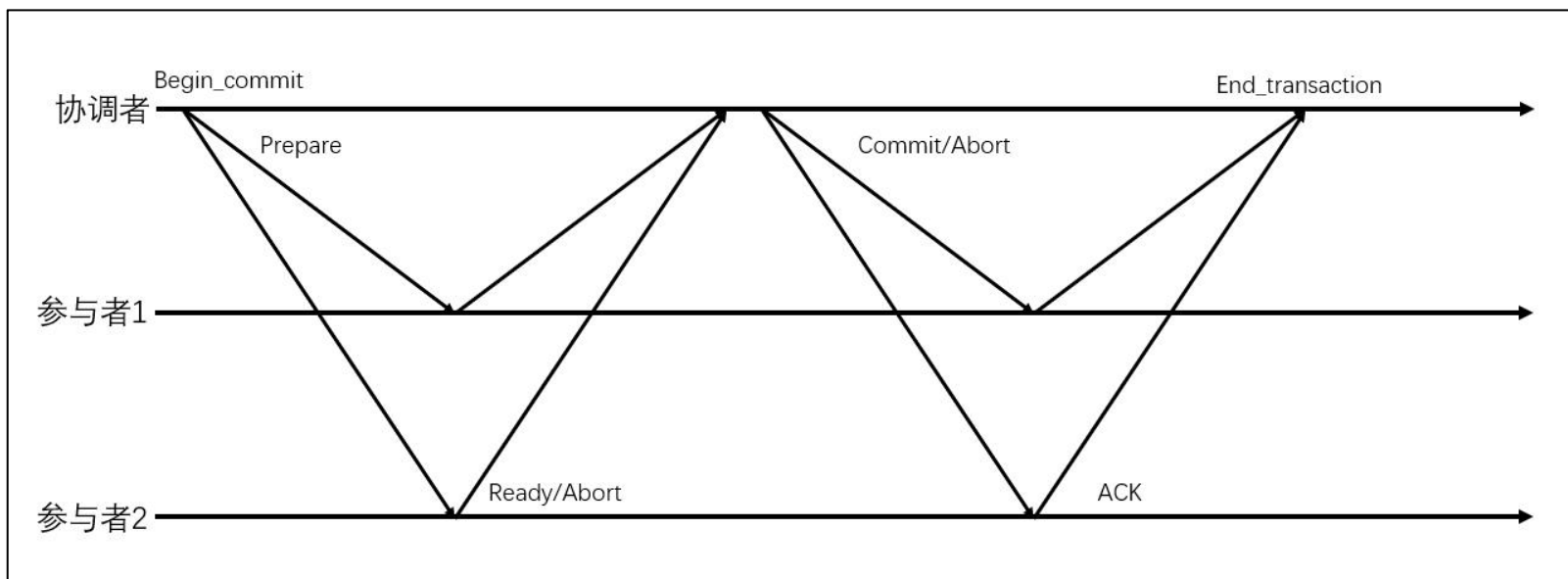
- (1) 某些节点检测到冲突或违反约束导致中止，而其他节点提交成功。
- (2) 某些提交命令在网络中丢失，最终某些节点因等待超时而中止，而其他节点提交成功。
- (3) 某些节点在提交过程中崩溃，恢复后事务回滚，而其他节点提交成功。



两阶段提交流程

第一阶段：准备阶段 (Prepare Phase)

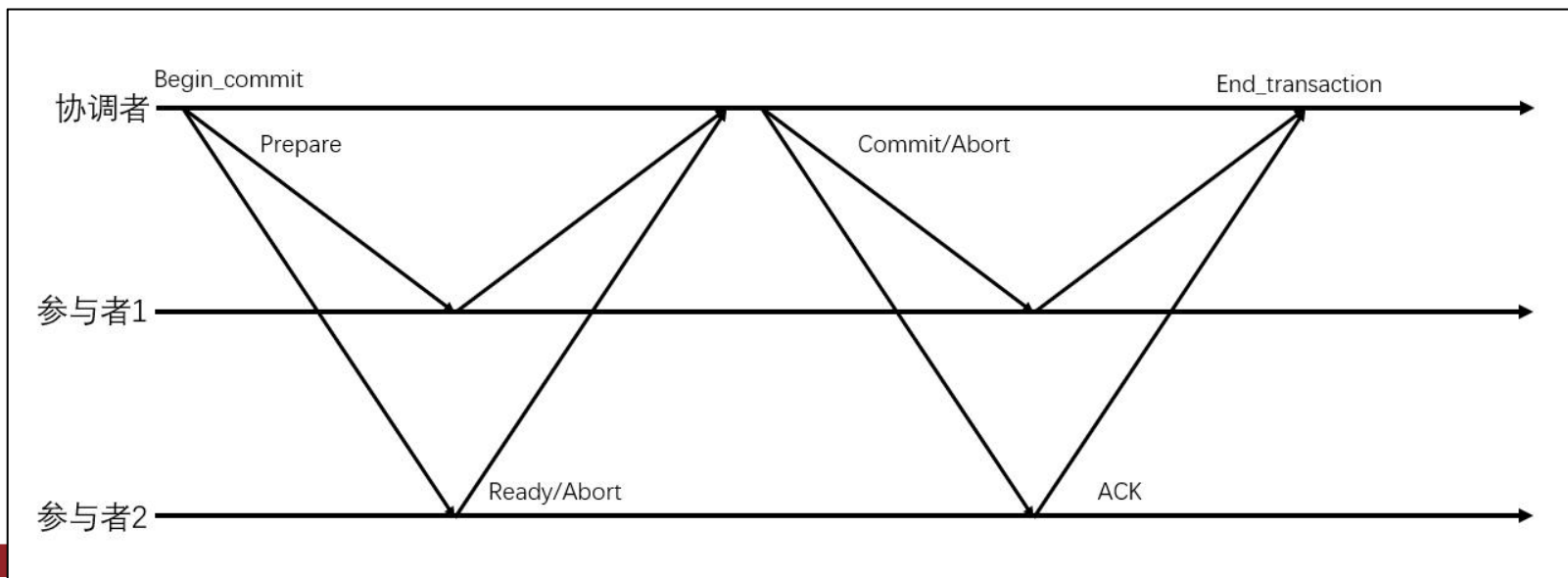
首先协调者在日志中写入 “Begin_commit” 记录，向参与者发送 “Prepare” 请求，等待参与者应答。参与者收到 “Prepare” 信息后，判断其子事务是否准备好提交，如果准备提交则在日志中写入 “Ready” ， 否则写入 “Abort” ， 并将对应应答反馈给协调者。



两阶段提交流程

第二阶段：提交阶段 (Commit Phase)

协调者收集所有参与者的应答，采用“一票否决制”。若在所有参与者反馈中不存在“Abort”应答，则该事务可以提交，并在日志中写入“Commit”，否则事务将中止，并在日志中写入“Abort”。协调者根据提交或者中止的决策结果向所有参与者发送“Commit”或者“Abort”命令，等待参与者返回确认信息。参与者接收到命令后，执行提交/中止子事务，并发送确认信息 (Ack) 给协调者。最后协调者收到所有确认信息后，在日志中写入“End_transaction”，全局事务执行完毕。



两阶段提交解决的问题

前文提到的三种违反原子提交的情况，它们在两阶段提交协议下都得到了解决。

(1) 当某些节点检测到冲突或违反约束导致中止时，它将向协调者返回 “Abort” 应答，那么协调者将向所有参与者发出 “Abort” 命令从而回滚所有子事务。

(2) 当协调者或参与者在两阶段提交过程中崩溃或发出的信息在网络中丢失，协调者一直等待收到所有参与者的应答，两阶段提交协议进入不到提交阶段，并且子事务都在等待协调者第二阶段的指令，任何一个子事务都无法提交，协调者等待过长触发超时机制。

(3) 当某些节点在提交过程中崩溃，恢复后事务回滚，而其他节点提交成功，可以通过日志回放提交或者回滚指令，保证所有子事务全部提交或者全部回滚。

6.3 分布式提交

01

6.3.1 两阶段提交协议

02

6.3.2 两阶段提交协议的故障恢复

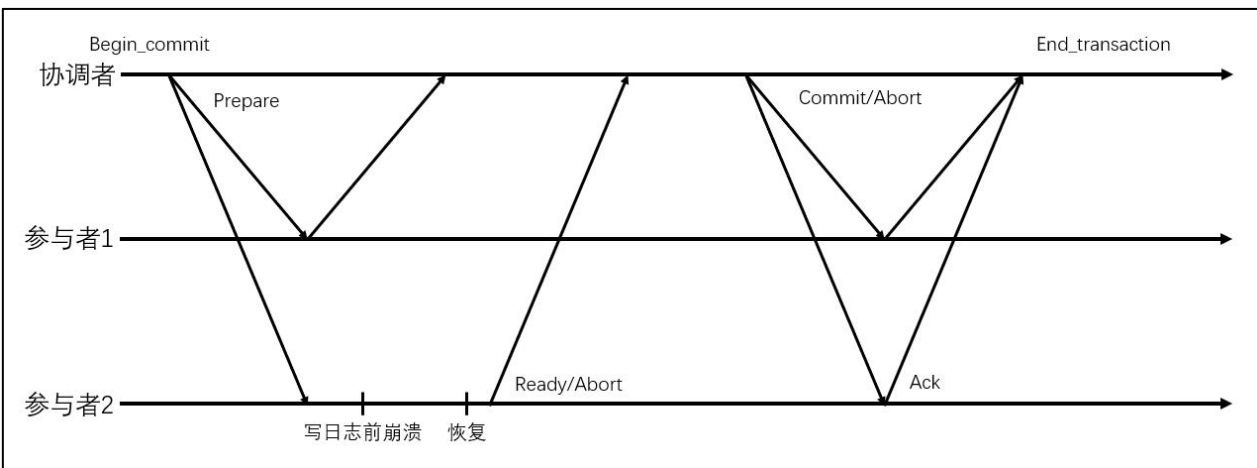
03

6.3.3 两阶段提交与副本共识

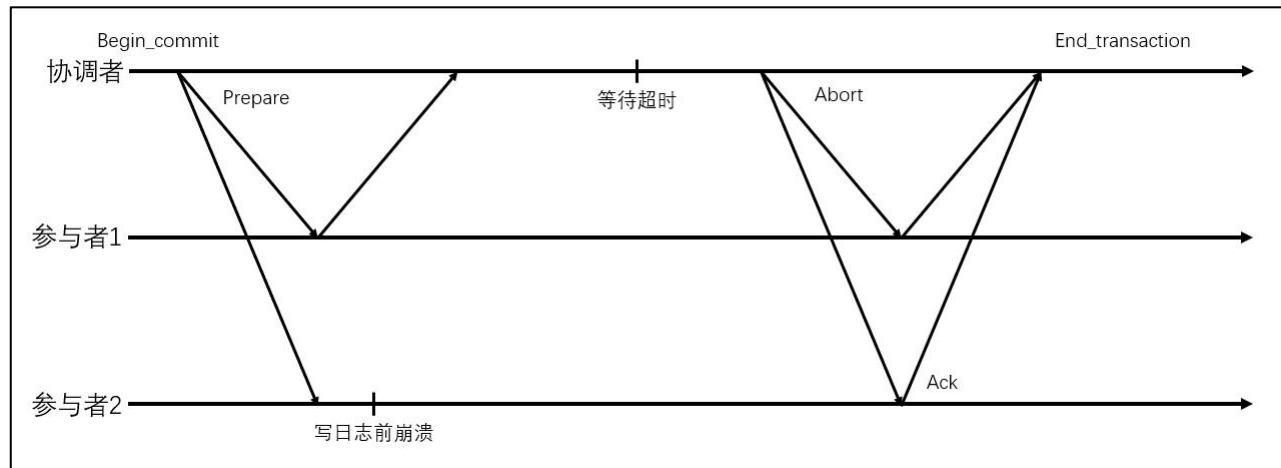


1、准备阶段参与者崩溃

(1) 参与者在向协调者发送 “Ready” / “Abort” 之前，需首先写日志，若参与者在写日志前崩溃，协调者将无法收到参与者的应答。若参与者在协调者等待超时前恢复，则该参与者进行重试并重新向协调者发送 “Ready” / “Abort” 应答，如下左图所示。若在协调者等待超时后，参与者仍未恢复，协调者将默认为收到 “Abort” 应答，并向所有参与者发送 “Abort” 命令，如下右图所示。



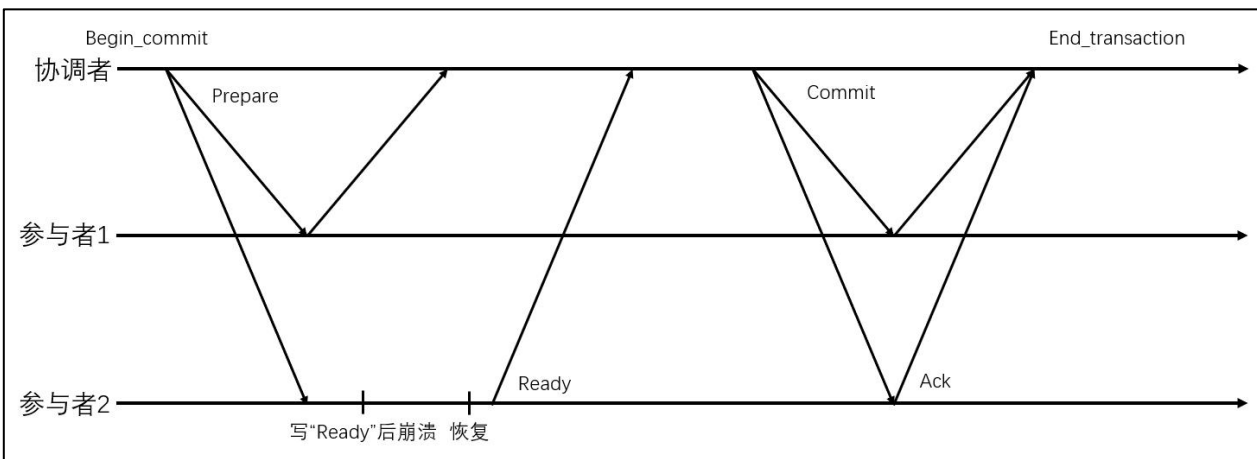
准备阶段参与者写日志前崩溃（超时前恢复）



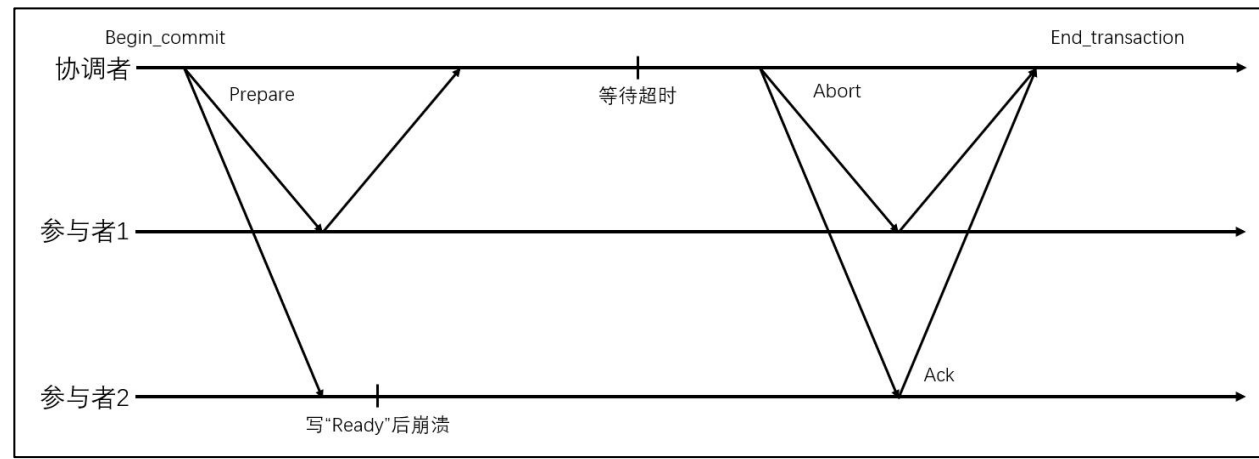
准备阶段参与者写日志前崩溃（触发超时机制）

1、准备阶段参与者崩溃

(2) 若参与者做出“Ready”决定，并成功写日志，但在发送“Ready”前崩溃或者发送给协调者的“Ready”丢包。这时协调者同样无法收到参与者的应答，因此处于等待状态。若参与者在协调者等待超时前恢复，协调者会向所有参与者发出询问命令，参与者根据日志记录做出“Ready”应答，而后恢复正常流程，如下左图所示。若该参与者在协调者等待超时后仍未恢复，则协调者默认收到“Abort”应答，并向所有参与者发送“Abort”命令，如下右图所示。



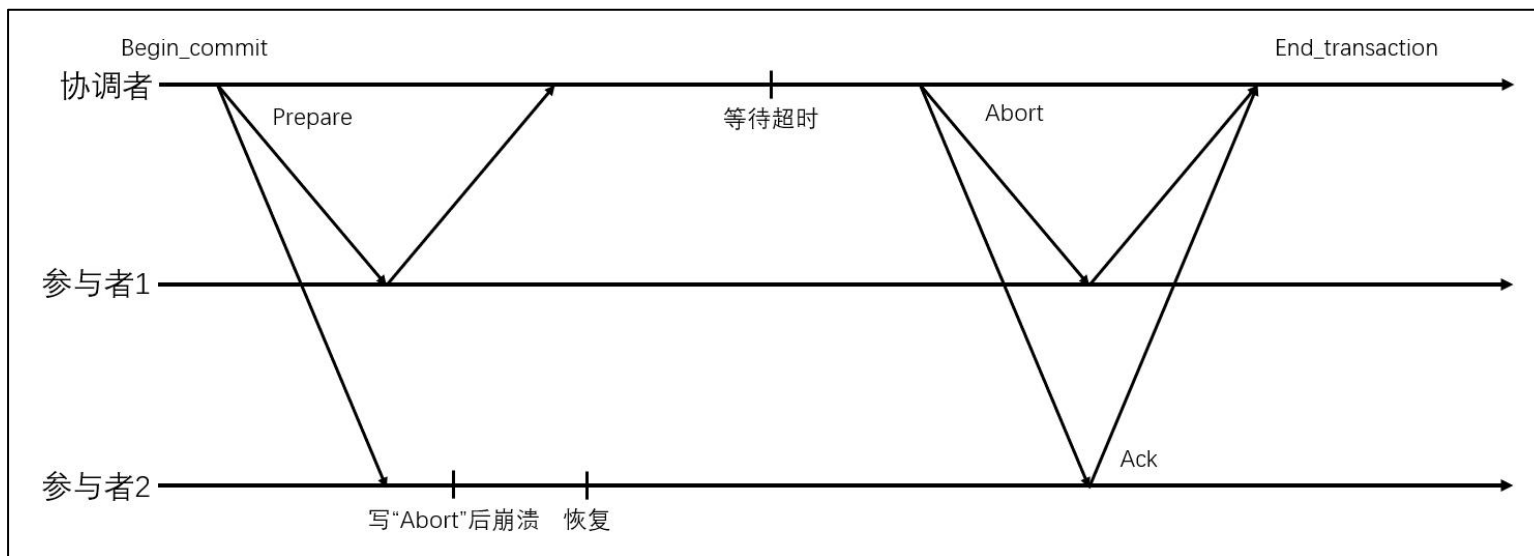
准备阶段参与者发送“Ready”应答前崩溃
(超时前恢复)



准备阶段参与者发送“Ready”应答前崩溃
(触发超时机制)

1、准备阶段参与者崩溃

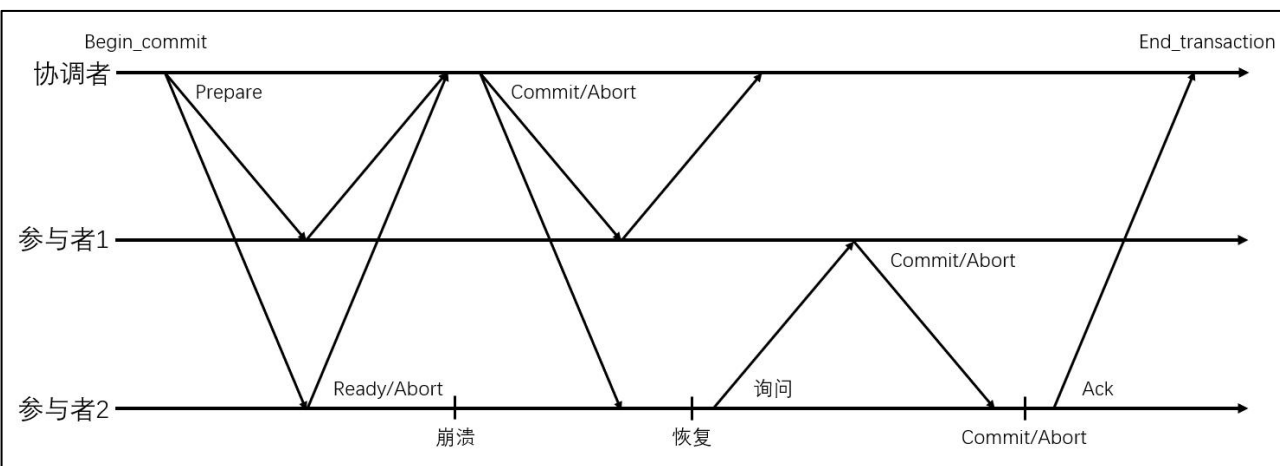
(3) 若参与者做出“Abort”决定，并成功写日志，但在发送前崩溃或者发送给协调者的“Abort”丢包。则该参与者恢复后可以继续发送“Abort”信息告知协调者，也可以不做任何处理，直到协调者等待超时，会自动向所有参与者发送Abort”命令。



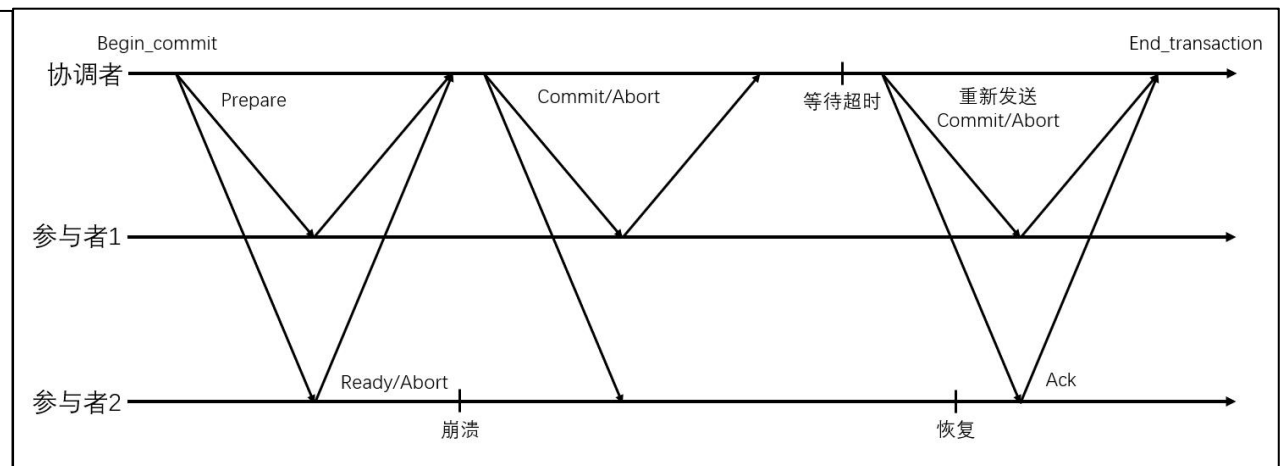
准备阶段参与者发送“Abort”应答前崩溃

2、提交阶段参与者崩溃

(1) 若参与者在准备阶段发送“Ready” / “Abort” 应答后并在提交阶段发送“Ack” 应答前崩溃，则协调者将收到该参与者的应答，并做出“Ready” / “Abort” 决定，进入提交阶段，并向所有参与者发送“Commit” / “Abort” 命令。因为有参与者崩溃，所以协调者无法收到所有“Ack” 应答，因此处于等待状态。若参与者在协调者等待超时前恢复，则参与者通过访问其他参与者做出决定（如果所有参与者均提交或准备提交，则该参与者提交，否则中止）。若参与者在协调者等待超时后仍未恢复，则协调者将重新发送“Commit” / “Abort” 命令，等待所有参与者反馈。



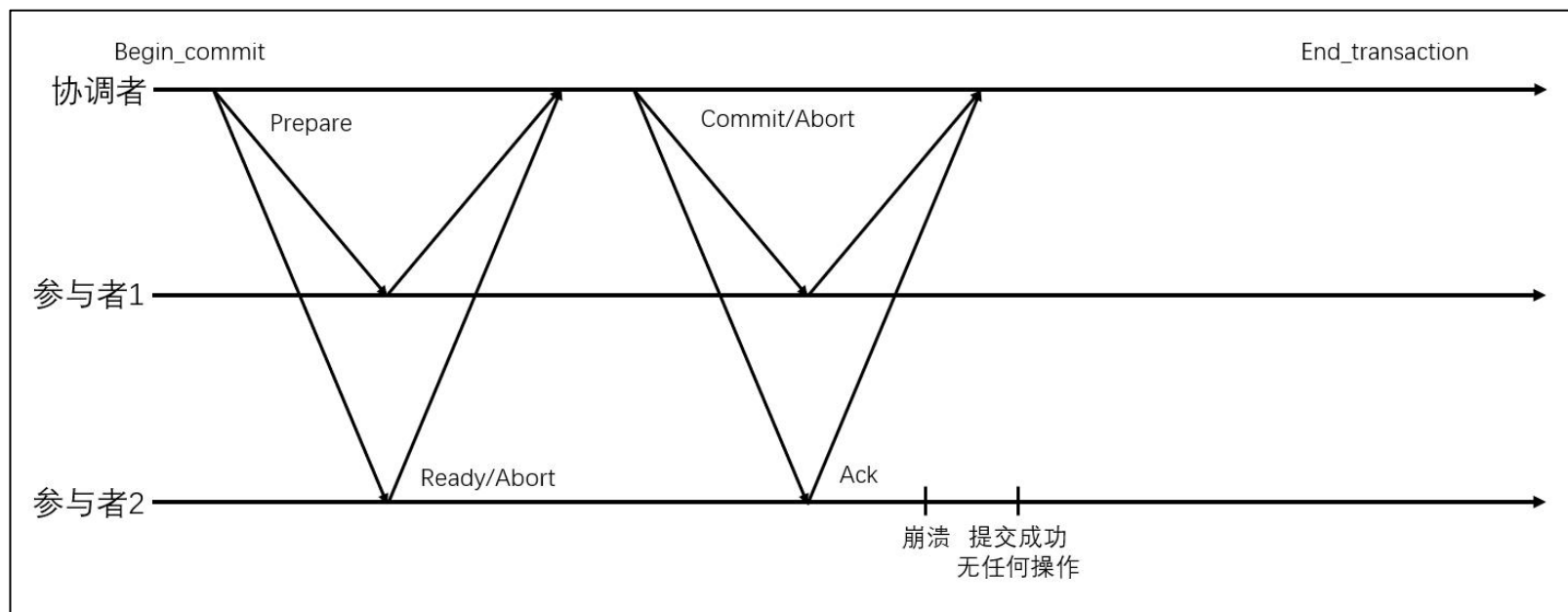
提交阶段参与者发送“Ack” 应答前崩溃
(超时前恢复)



提交阶段参与者发送“Ack” 应答前崩溃
(触发超时机制)

2、提交阶段参与者崩溃

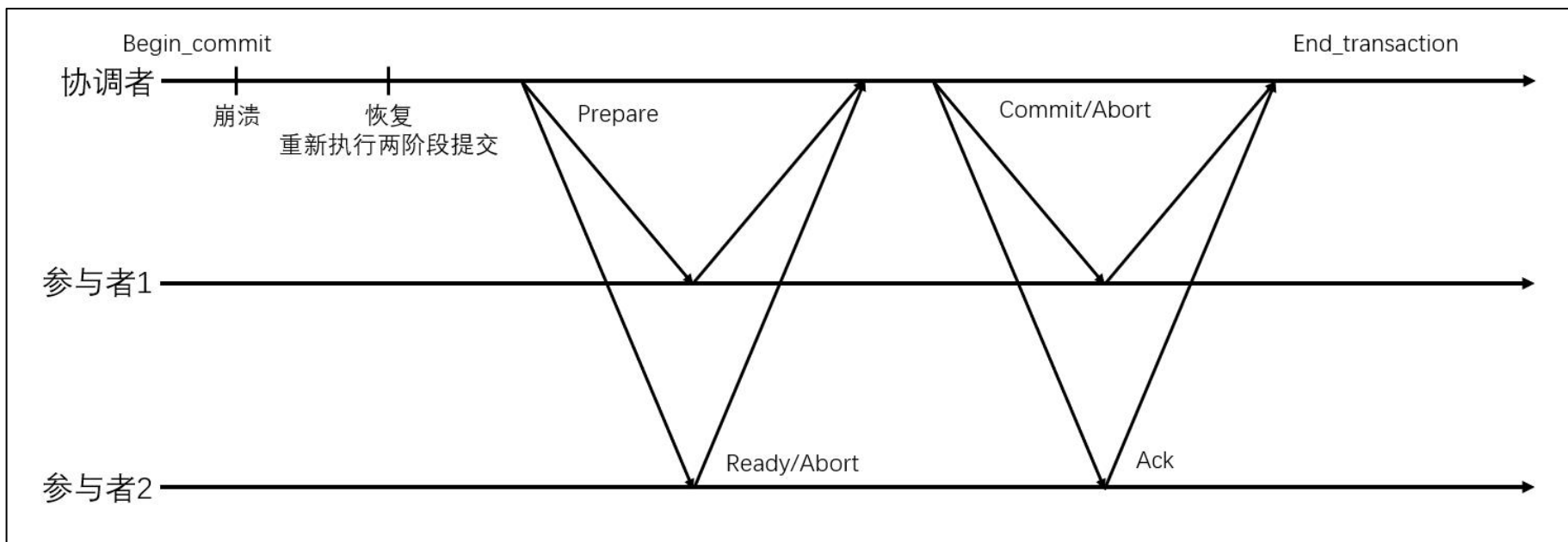
(2) 若参与者在发送 “Ack” 应答后崩溃。协调者和参与者均正常结束事务，无需任何操作。



提交阶段参与者发送 “Ack” 应答后崩溃

3、准备阶段协调者崩溃

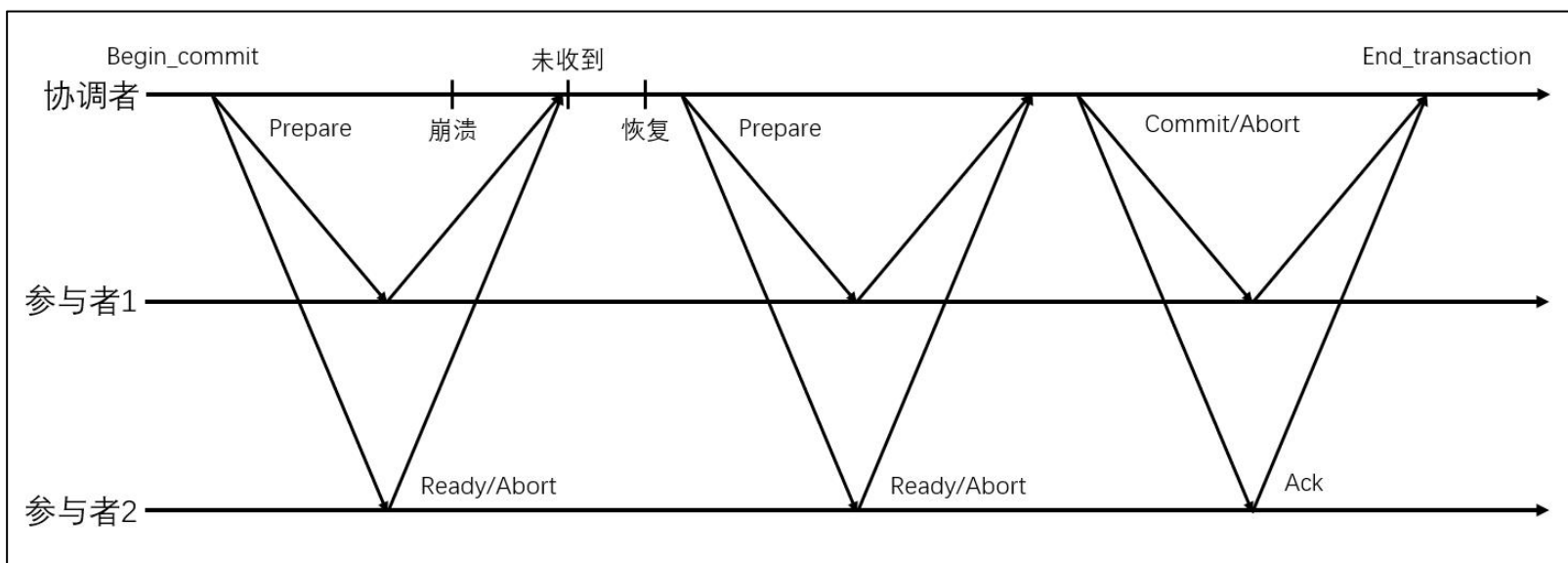
(1) 协调者在发送“Prepare”命令前崩溃。协调者恢复后，重新执行两阶段提交协调，并重新发送“Prepare”命令。



准备阶段协调者发送“Prepare”命令前崩溃

3、准备阶段协调者崩溃

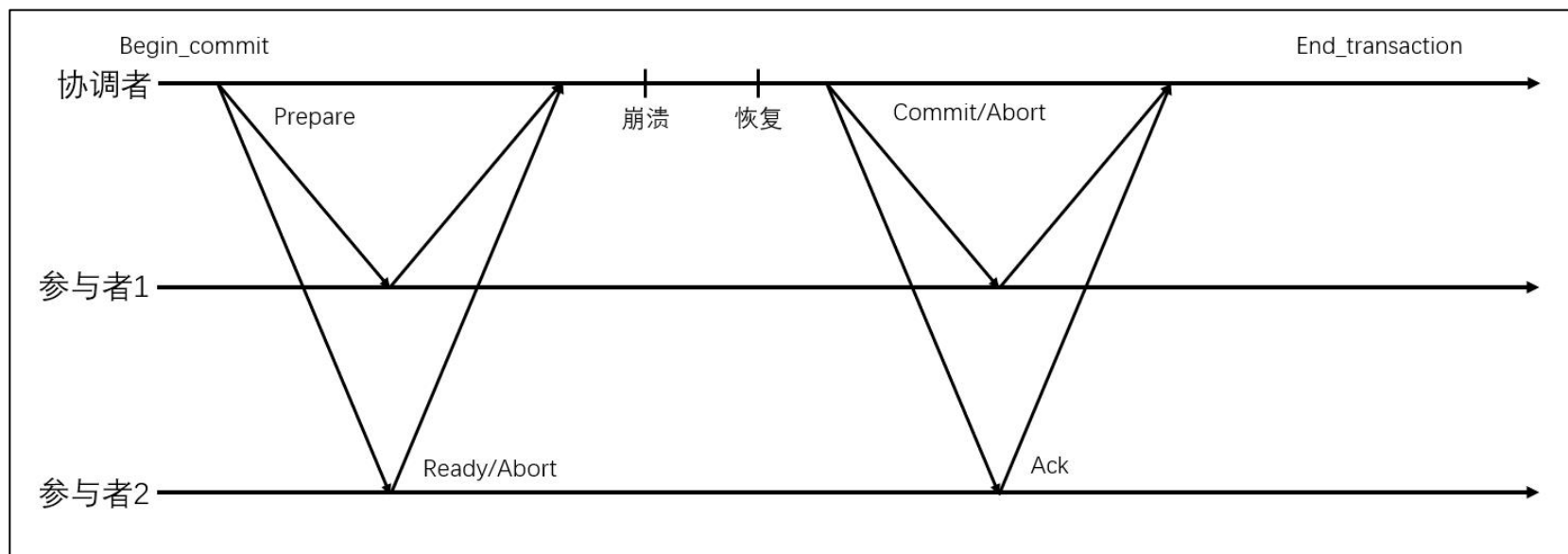
(2) 协调者在发送“Prepare”命令后崩溃。参与者在收到命令后将发送“Ready” / “Abort” 应答，之后参与者将一直处于等待状态。直到协调者恢复，重新向所有参与者发送“Prepare”命令。



准备阶段协调者发送“Prepare”命令后崩溃

4、提交阶段的协调者崩溃

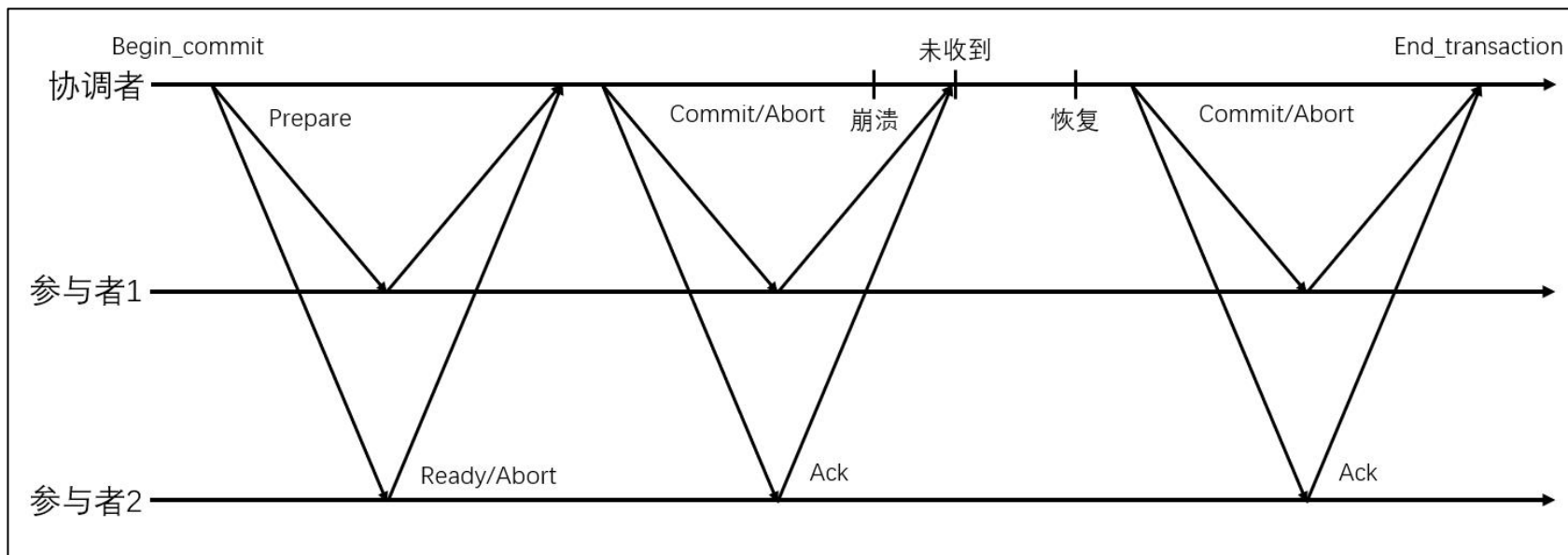
(1) 协调者在发送 “Commit” / “Abort” 命令前崩溃。参与者将一直处于等待状态，直到协调者恢复，向所有参与者重发 “Commit” / “Abort” 命令。



提交阶段协调者发送 “Commit” / “Abort” 命令前崩溃

4、提交阶段的协调者崩溃

(2) 协调者在发送 “Commit” / “Abort” 命令后崩溃。协调者将无法收到 “Ack” 应答，只能等待协调者恢复后重发 “Commit” / “Abort” 命令。



提交阶段协调者发送 “Commit” / “Abort” 命令后崩溃

6.3 分布式提交

01

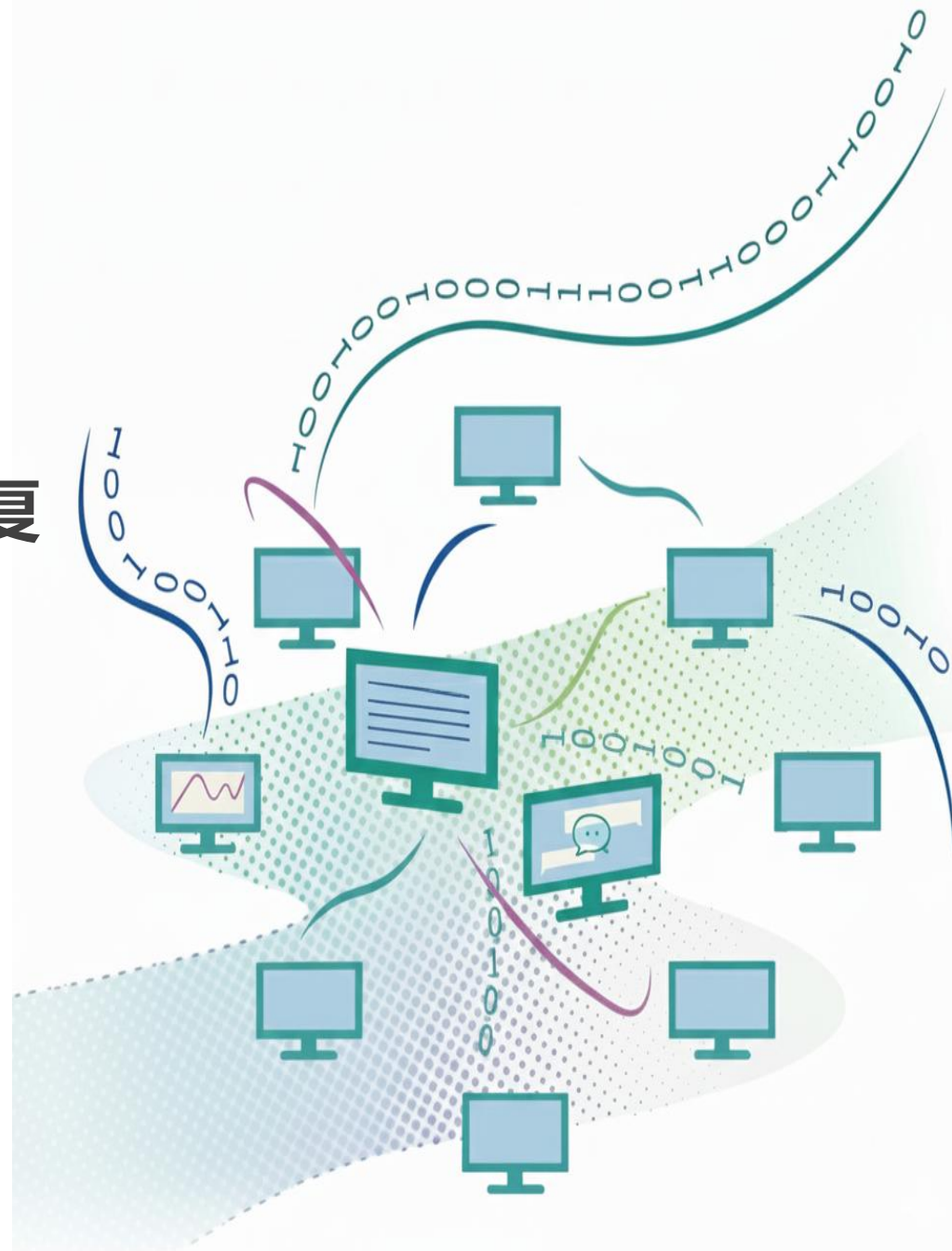
6.3.1 两阶段提交协议

02

6.3.2 两阶段提交协议的故障恢复

03

6.3.3 两阶段提交与副本共识



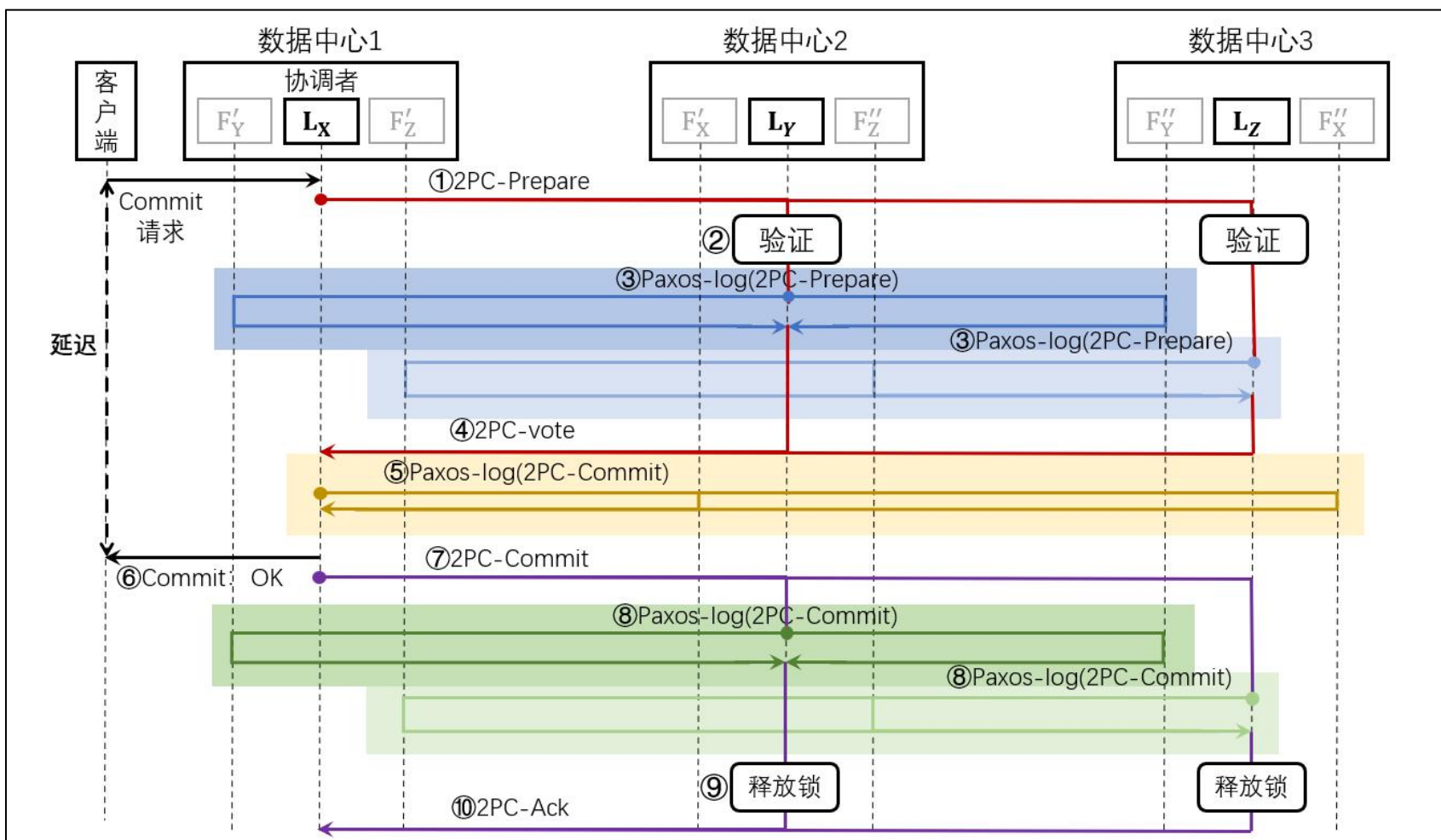
6.3.4 两阶段提交与副本共识

为了实现分布式数据库的高可用性，分布式数据库往往为数据设置多个副本，这些副本存储于不同的节点。

数据库为数据设置多个副本之后，在管理分布式事务时就不仅需要考虑原子性、持久性等ACID特性，还需要考虑副本一致性的问题。

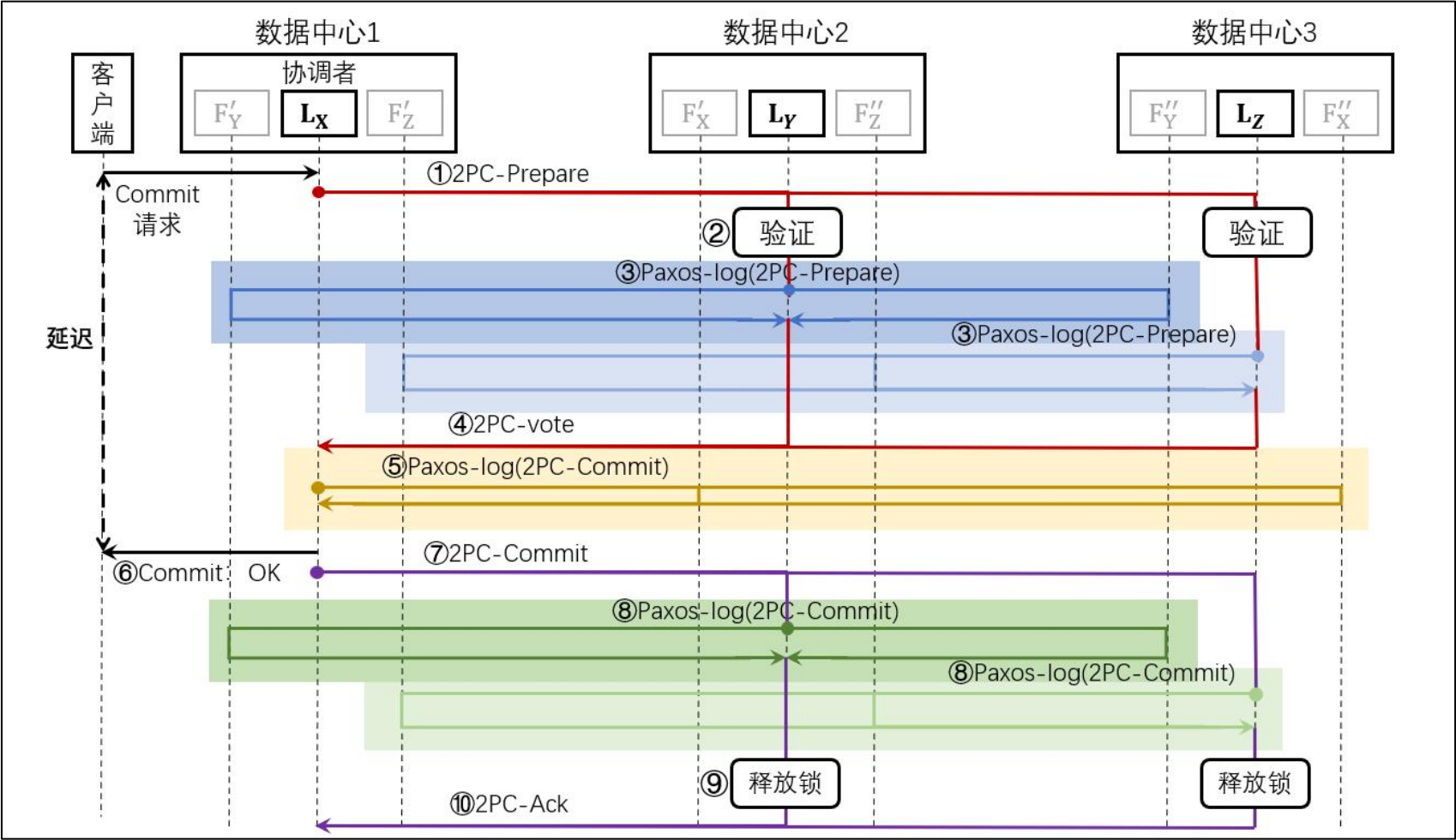
本小节将介绍两阶段提交和副本共识的混合设计，在分布式提交过程中利用Paxos共识算法来同时保证事务原子性和副本一致性。

6.3.4 两阶段提交与副本共识



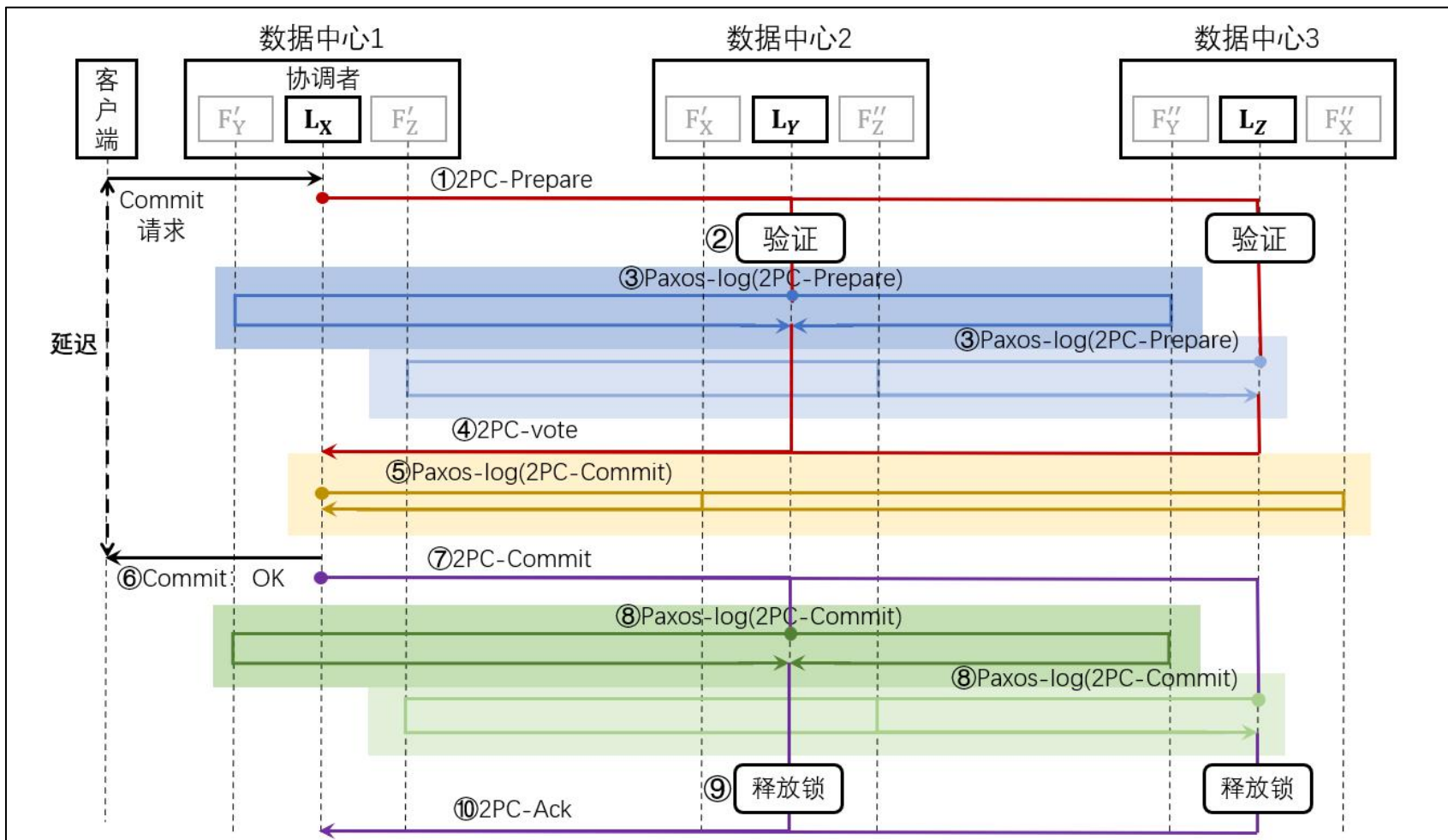
以典型的3个数据中心为例，且每个数据中心拥有3个数据分片X、Y、Z（即每个数据中心都拥有全量数据），每个分片都有3个副本位于不同数据中心。一个分布式事务涉及X、Y、Z三个数据分片，三个分片的主副本分别位于三个数据中心，其中操作数据分片LX的进程为两阶段提交的协调者Leader，操作数据分片LY和LZ的进程为参与者Leader。

6.3.4 两阶段提交与副本共识



首先①，数据中心1的协调者LX向数据中心2和数据中心3的参与者LY和LZ发送两阶段提交的“Prepare”信息，其中包含事务元数据和读写集。②，参与者Leader收到“Prepare”信息后，通过并发控制信息和读写集开始乐观并发控制的验证阶段，这时需要对写集上锁，若验证成功，则在本地记录“Prepare”日志。③，与传统两阶段提交不同，为了保证副本一致性，需要将“Prepare”日志通过Paxos共识进行同步复制给参与者Follower。④，参与者Leader向协调者Leader返回两阶段提交的“Prepare”请求的应答。

6.3.4 两阶段提交与副本共识



⑤, 协调者Leader收到所有参与者Leader的应答后, 做出“Commit/Abort”的决定, 记录此决定后并通过Paxos同步复制给协调者Follower。⑥, 就可以将“Commit/Abort”的信息返回用户。

下面的过程将异步进行。⑦, 协调者Leader向参与者发送提交阶段的“Commit/Abort”命令。⑧, 收到命令的参与者Leader将响应信息记录日志并通过Paxos同步给参与者Follower, 并丢弃写集。⑨, 参与者释放所有锁。⑩, 参与者Leader向协调者Leader回复“Ack”信息。

在没有故障的情况下, 加入Paxos的两阶段提交过程的平均延迟受往返延迟 (Round-Trip Time, RTT) 和节点处理时延的影响, 而尾延迟主要受节点处理时延的影响。因此, 当工作负载的并发争用较低时, 分布式数据库的吞吐量主要受网络带宽的影响。当工作负载的并发争用较高时, 吞吐量主要受锁持有时间的影响。

第六章、事务

01

6.1 事务的基本概念

02

6.2 分布式事务与并发控制

03

6.3 分布式提交

04

6.4 分布式事务处理系统



6.4 分布式事务处理系统

01

6.4.1 跨地理区域事务处理Spanner

02

6.4.2 确定性事务处理Calvin

03

6.4.3 无协调事务处理Berkeley Anna

04

6.4.4 多主乐观事务处理GeoGauss

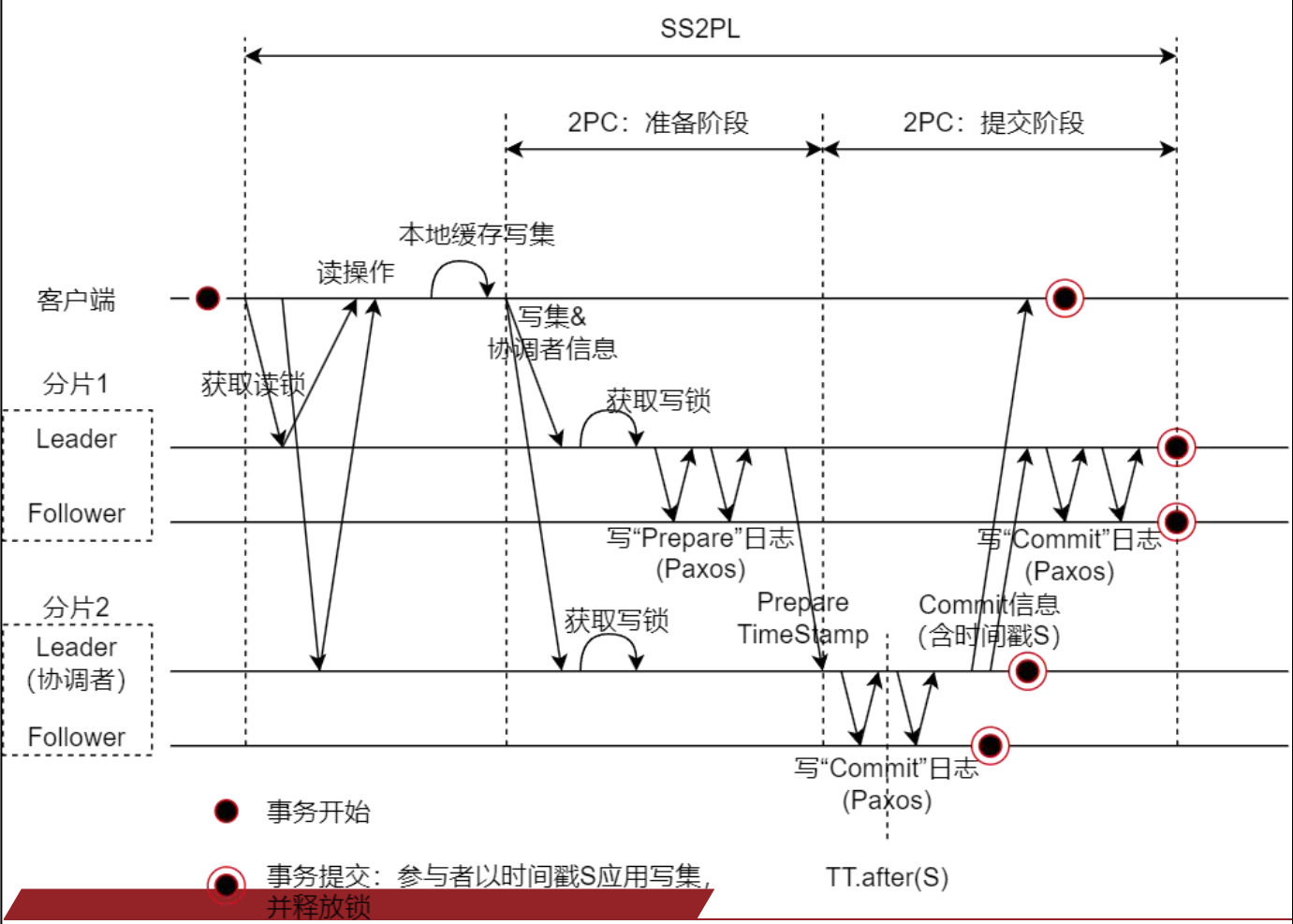


6.4.1 跨地理区域事务处理Spanner

- Spanner是由谷歌开发的一款全球分布式、可扩展、高可用的关系型数据库系统。
- Spanner数据库既具有大数据数据库系统的可扩展性，也具有分布式数据库事务处理能力，将数据复制到全球范围的多个数据中心，并保证数据的强一致性。
- Spanner使用TrueTime机制保证事务并发控制的正确性，并配合Paxos实现数据库副本的外部一致性（External Consistency），即同时保证最强的一致性级别线性一致性（Linearizability）和最强的隔离级别可串行化（Serializability），要求两个事务完全按照时间先后顺序在数据库中发生，也叫做严格可串行化（Strict Serializability）。此外，Spanner还支持无锁只读事务和对历史数据的非阻塞读。读操作和快照读都不包含锁机制、不会阻塞后面到达的写操作，并且无论是读操作还是快照读，都可以在足够新的副本上执行。
- 从复制架构上看，Spanner采用了主从复制架构，每个节点上存放有100-1000个分片（Tablet），每个分片有一个主副本（Leader）和多个从副本（Follower），被存放在跨地理区域的多个节点上，对某一分片的写操作必须由该分片的领导者节点完成，用Paxos共识协议维护主从副本的一致状态。

Spanner跨分片事务处理流程

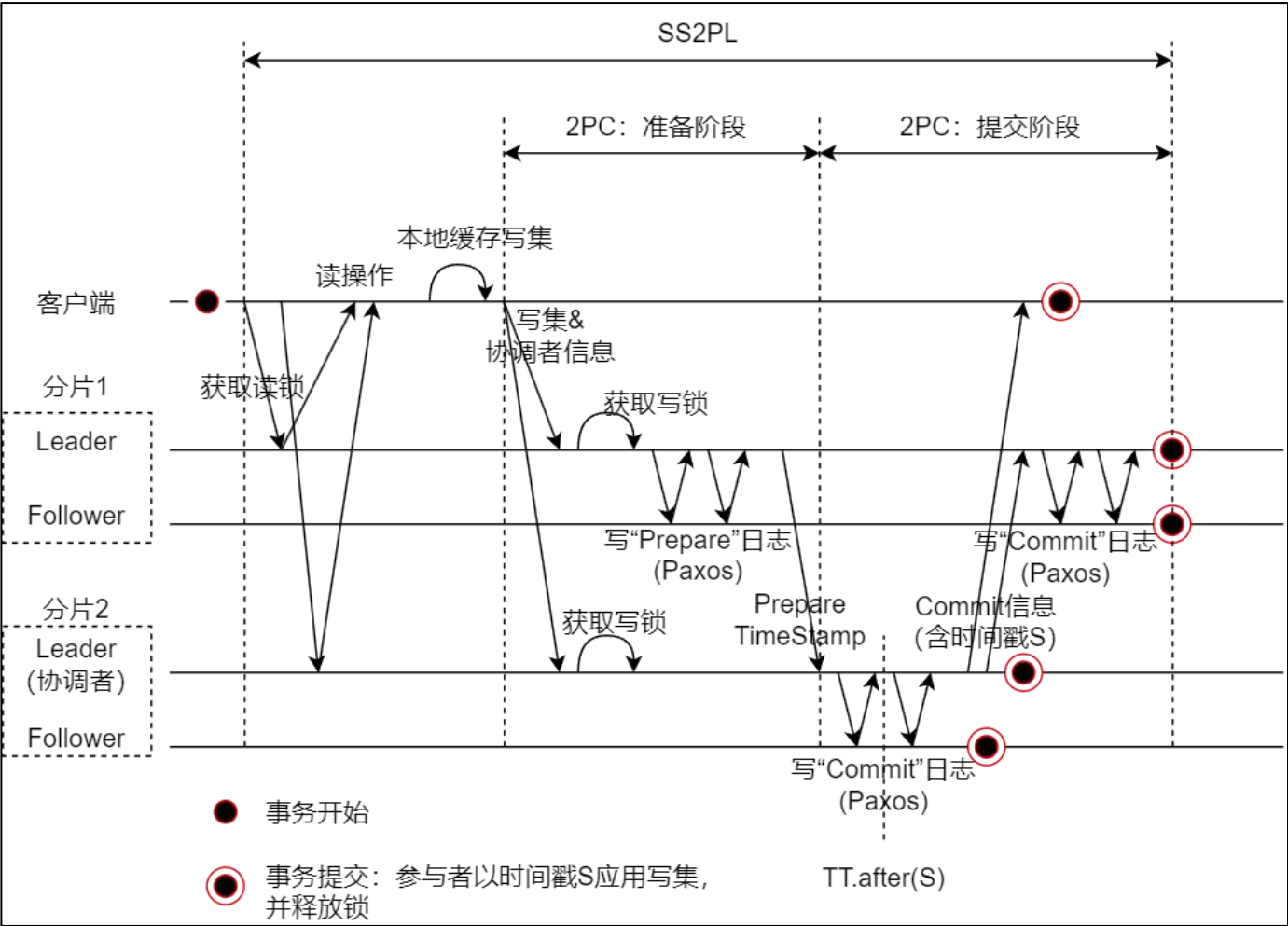
Spanner对于读写事务（read-write）和只读事务（read-only）的处理方式不同。对于一个跨分片的读写事务，Spanner采用SS2PL+2PC来保证事务的隔离性和原子提交，并需要有一个协调者对该事务负责。



在事务开始时，客户端会对将要读取数据的分片加读锁并读取数据（SS2PL的封锁阶段），事务内的所有写操作（写集）将被缓存在客户端本地。当事务执行完毕，客户端开始发起**2PC两阶段提交**。

在2PC的**准备阶段**，客户端将向所有涉及的分片的Leader节点发送写集，以及当前事务的协调者信息。每个**参与者Leader**需要先在本地对将要写的数据加写锁（SS2PL的封锁阶段），并通过Paxos把“Prepare”记录写入日志，然后将一个根据本地时钟获取的最新的、比之前签发给其他事务都要大的时间戳Prepare Timestamp发给协调者，用来帮协调者确定一个全局最大的时间戳。

Spanner跨分片事务处理流程



协调者的过程稍有不同：需要先给将要写的数据加写锁，但跳过准备阶段，先收集所有参与本事务的Leader发来的时间戳Prepare Timestamp，从这些时间戳以及本地当前时间的最迟估计中选取一个最大的时间戳作为本事务的提交时间戳S，并通过Paxos写入一个“Commit”日志记录。

再根据TrueTime API得知S时间戳确定过去后（调用TT.after(S)），该事务可以安全提交，注意等待提交的时间常与Paxos通信时间重叠。最后，协调者发送提交时间戳给客户端和所有参与者leader，每个参与者Leader接收到协调者发来的提交时间戳之后，将事务结果写入日志并释放锁（SS2PL的解锁阶段），同时也使用Paxos同步给Follower副本节点，并统一使用此时间戳作为该事务的提交时间。

Spanner的事务处理使得发生在提交时间之后的事务都可以读取到当前事务的数据更新，保证了外部一致性。

6.4 分布式事务处理系统

01

6.4.1 跨地理区域事务处理Spanner

02

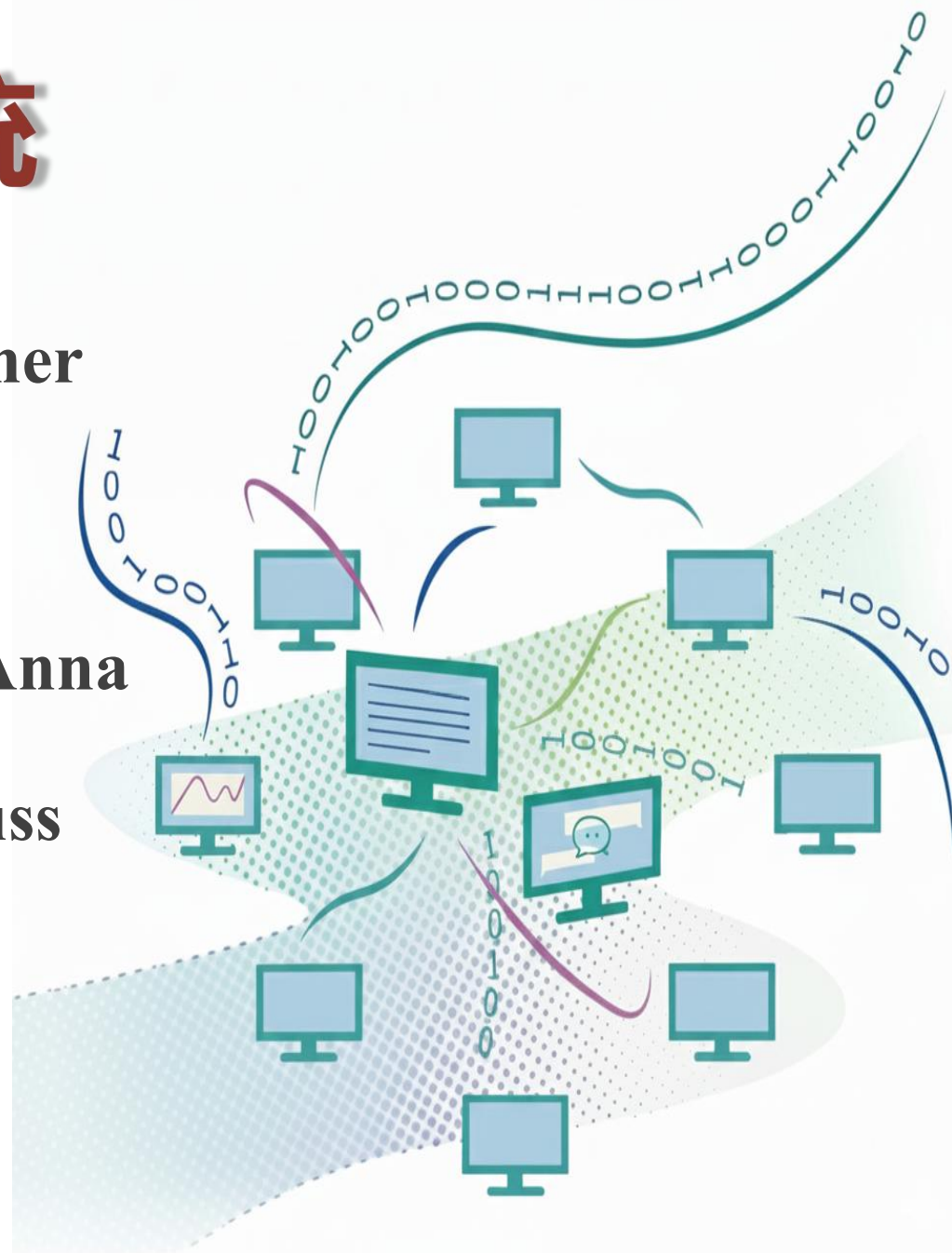
6.4.2 确定性事务处理Calvin

03

6.4.3 无协调事务处理Berkeley Anna

04

6.4.4 多主乐观事务处理GeoGauss



6.4.2 确定性事务处理Calvin

- Calvin是由耶鲁大学2012年提出确定性事务处理数据库，准确地说它是一个**可扩展的事务层**，采用确定性事务处理技术，它可以连接任何实现了插入、读取、更新、删除的存储系统，从而形成完整的数据库系统。例如，存储系统可以是多节点键值存储系统，Calvin可以组织每个节点上的存储的数据，并协调事务执行过程中节点间的网络通信。
- Calvin没有采用2PC，避免2PC中协调者和参与者之间的多次协调开销，而是在事务执行之前进行**预处理**（确定多个事务的先后执行顺序），使所有事务在所有节点按同一种确定性的顺序执行，因此Calvin被称作确定性数据库。
- 这一全局顺序可以被认为是一个日志记录，该日志将被复制到所有副本，所有副本并行地执行此日志。这样，即使一个副本故障，其他副本也能执行并提交成功。同时还避免了分布式事务过程中节点之间的相互协调，减少了**网络通信**开销。
- 此外，一般情况下在2PC过程中数据必须处于上锁状态，直到事务提交才能释放，所以Calvin还间接地减少了事务持有锁的时间，降低了事务间冲突的概率。

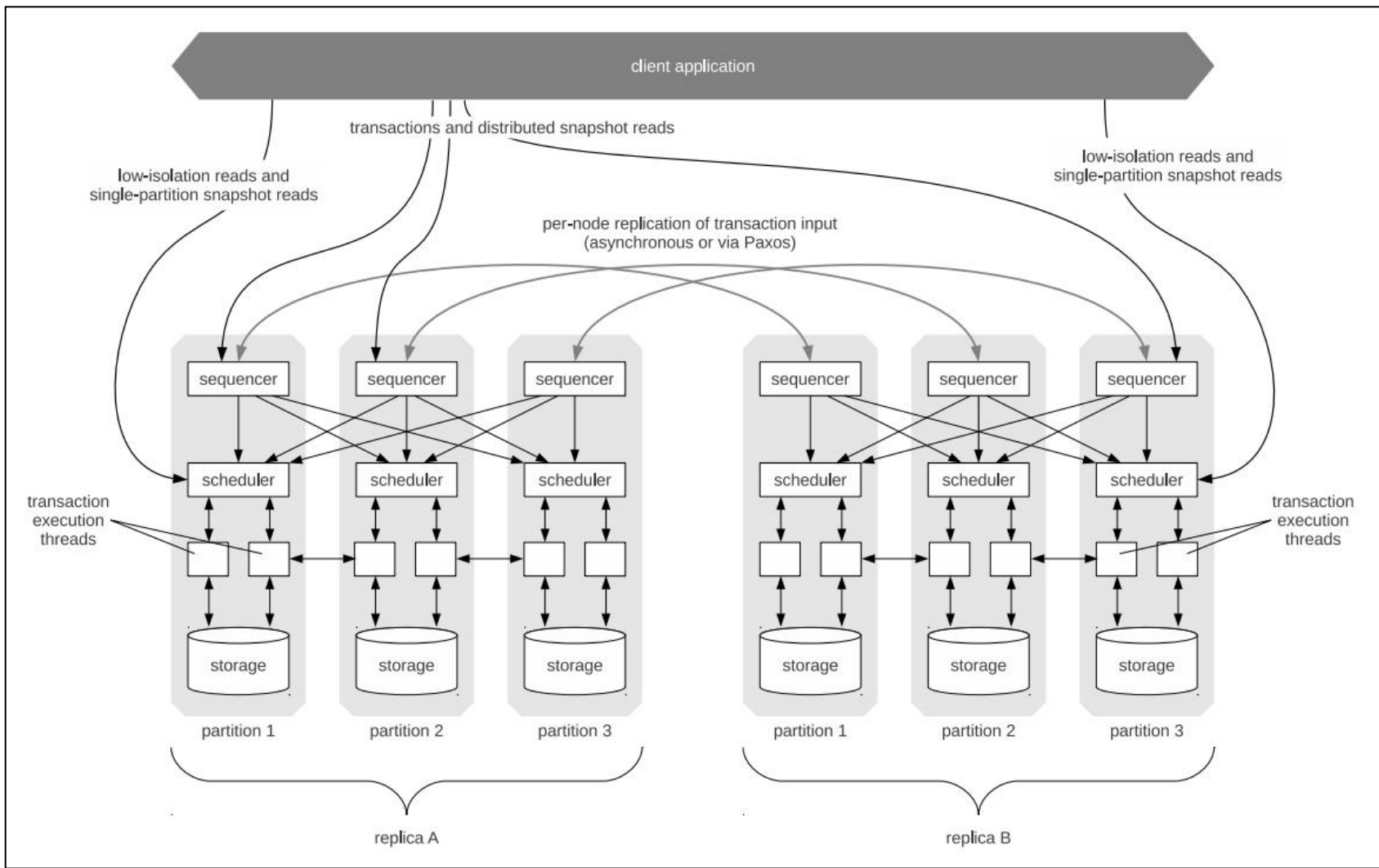
Calvin的系统架构和处理流程

系统分成三个独立的处理层：

1) 排序(Sequencing)层：多个排序器(Sequencer)以10ms为一个周期(epoch)并行接收客户端应用发来的事务，将一个周期接收的事务排序，并将排序结果记录成一个日志文件。

2) 调度(Scheduling)层：存储层的数据是分片的，每个数据分片都有一个对应的Scheduler用来排序涉及该分片数据的事务，每个Scheduler收到全部Sequencer发送过来的排序日志后，根据确定性的规则对这些事务排序，整理形成此epoch的全局顺序。

3) 存储 (Storage) 层：存储层负责物理数据的布局，使用CRUD接口 (Create/Read/Update/Delete) 访问数据，任何支持类似接口的存储引擎都可以连接Calvin，因此如RocksDB [10]这类单机存储可以通过连接Calvin实现对外支持分布式事务。



Calvin优势与弊端

Calvin优势：

Calvin提供了一种比较新颖的事务处理方案，通过事务预处理提前对事务并发顺序达成共识，减少事务执行过程中临时处理顺序冲突带来的开销，在保证吞吐量的同时提高了可扩展性。

Calvin弊端：

同时也存在一些弊端，由于需要收集一个批次的事务进行确定性排序、执行后才能将事务执行结果返回给用户，这使得确定性数据库缺乏用户与数据库之间的可交互性。而当下主流数据库业务往往需要用户与数据库进行多轮次交互，这是确定数据库实际应用到企业业务处理上的一大难题。

6.2 分布式事务与并发控制

01

6.4.1 跨地理区域事务处理Spanner

02

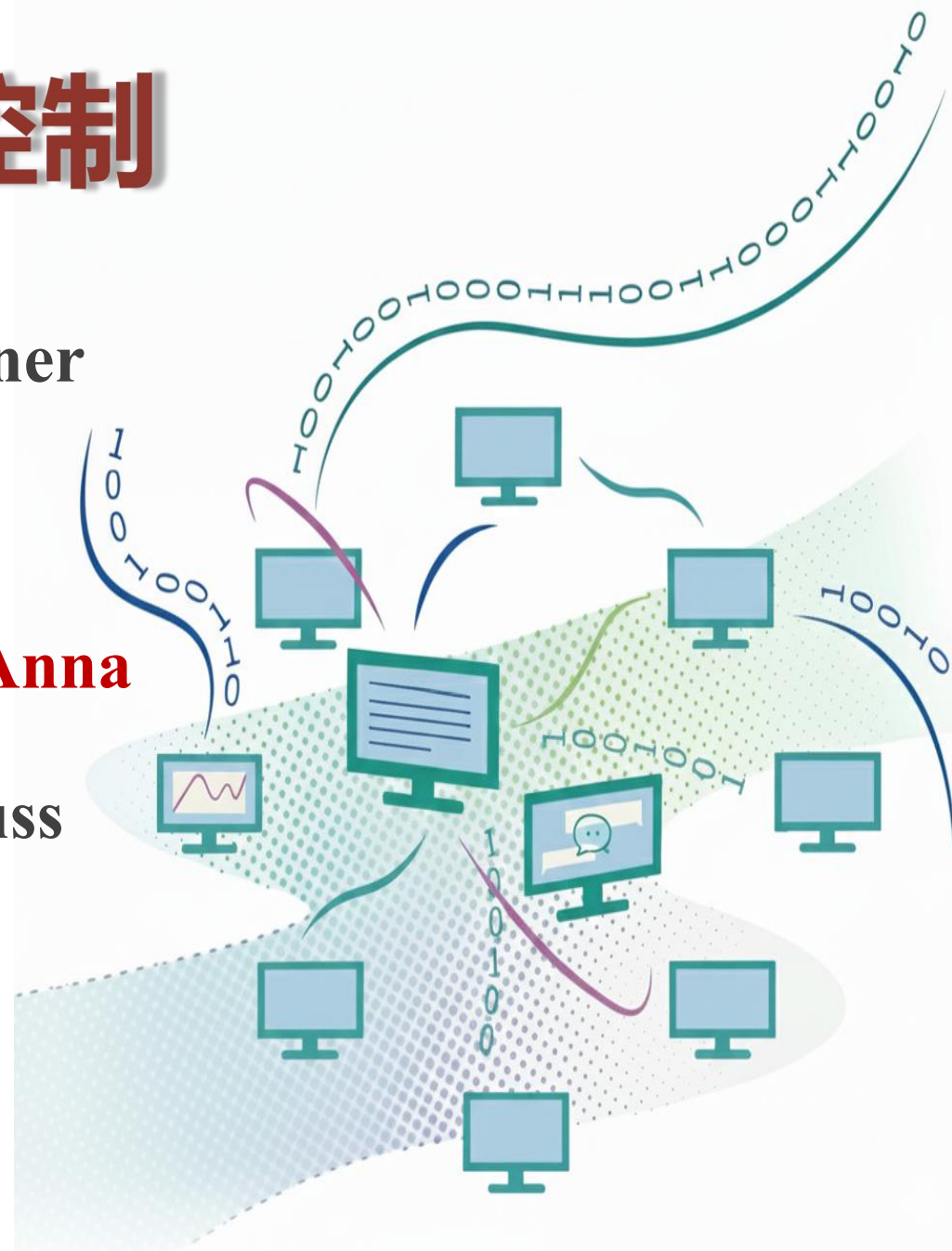
6.4.2 确定性事务处理Calvin

03

6.4.3 无协调事务处理Berkeley Anna

04

6.4.4 多主乐观事务处理GeoGauss



6.4.3 无协调事务处理Berkeley Anna

Anna是伯克利大学研究的一个分布式键值对存储数据库。它用格结构（**Lattice**）组合来存储和异步合并状态，以实现一系列的无协调的一致性级别，同时具备顶尖的读写性能和任意规模的可扩展性。

Anna基于ACI特性设计了一种特殊的数据结构**Lattice**，它由一个状态集合 S 、一个二进制运算符 $\sqcup$ 、和一个“最小”值 $\perp$ 组成。其中运算符 $\sqcup$ 被称为“least upper bound”（合并），且满足ACI特性：

- 1) 交换性 (Commutativity) : $\sqcup(a, b) = \sqcup(b, a) \quad \forall a, b \in S$
- 2) 结合性 (Associativity:) : $\sqcup(\sqcup(a, b), c) = \sqcup(a, \sqcup(b, c)) \quad \forall a, b, c \in S$
- 3) 幂等性 (Idempotence) : $\sqcup(a, a) = a \quad \forall a, b \in S$

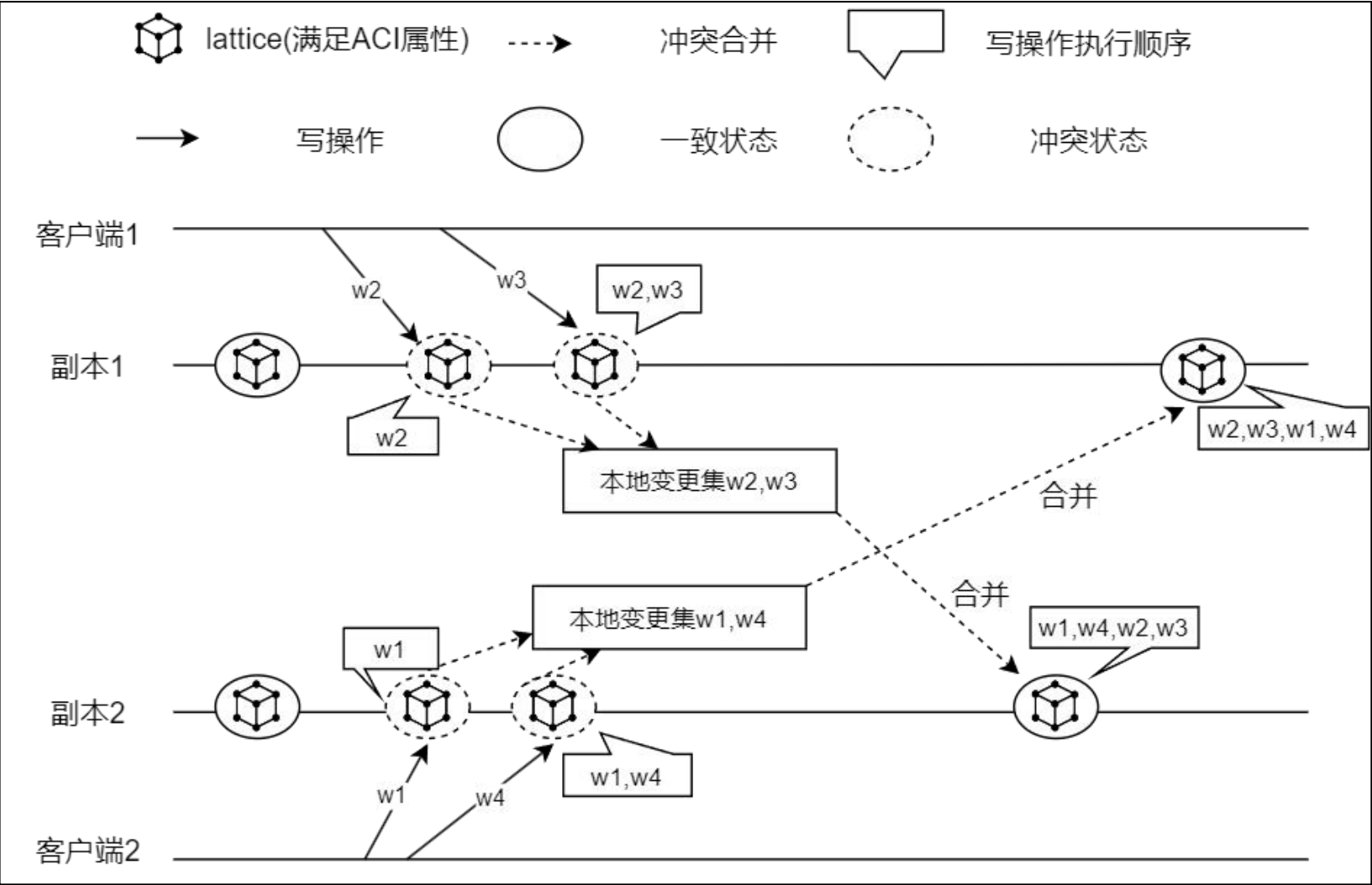
例如， $\max$ 运算就是满足ACI特性的合并运算。 ($\max(a, b) = \max(b, a)$ 、 $\max(\max(a, b), c) = \max(a, \max(b, c))$ 、 $\max(a, a) = a$)

$\sqcup$ 运算可以推导出集合 S 中任意两元素的偏序。（对于 S 中的任意两元素 a, b ，如果 $\sqcup(a, b) = b$ ，则称 b 的次序比 a 高，记为 $a < b$ 。）

$\perp$ 定义了集合 S 中的最小值 ($\forall a \in S, \sqcup(a, \perp) = a$) 。

存储在**Lattice**的数据在消息延迟、重新排序和重传情况下依旧保持一致性。

Anna 事务处理流程图



Anna在进行事务处理时，每个副本反复检查来自客户端代理的传入请求，并将结果添加到本地的变更集，也就是说该变更集存放的是一段时间内更新的键值对。

这段时间结束时，每个副本将其变更集中的键更新进行合并，并将其合并结果传播给负责这些键的副本，并清空变更集。它还检查从其他副本传来的消息，并将这些消息中键更新合并到其本地状态。

而Lattice对合并更新的顺序不敏感，因此即使各个副本以不同的顺序接收更新，它们也可以保证副本之间的一致性，满足最终一致性。

如图，副本2收到来自客户端2的写操作请求w1、w4，副本1收到来自客户端1的写操作请求w2、w3，此时它们处于冲突状态。一段时间后副本1和副本2把各自本地的变更集发给对方，对方分别利用Lattice的ACI属性进行冲突合并后，它们最终达到一致状态。

Anna优势

Anna优势:

Anna的开发目标之一是减轻开发负担，即Anna希望程序员编写程序的合并函数时，无需关注它的ACI特性。换句话说，需要在更标准化、模块化的Lattice中构造特定于应用的合并函数，而将维护和更新元数据的工作都交给Lattice，从而使Lattice具有更简洁的ACI特性。

因果一致性:

在多版本存储中，因果一致性需要保证同一对象的不同版本之间因果关系。因此，为了保证因果一致性，某些情况就不能调用数据的合并函数。例如，事务1更新了数据 $x=1$ ，而后数据 x 被事务2读取，并更新为 $x=2$ ，从而覆盖了事务1的更新（而非调用数据的合并函数），因为这两个事务的更新是具有因果关系的。但是，若事务2未读取数据 x ，而是直接更新 $x=2$ ，那么它们之间就不存在因果关系，这时，此并发冲突就可以通过调用合并函数来解决。

6.4 分布式事务处理系统

01

6.4.1 跨地理区域事务处理Spanner

02

6.4.2 确定性事务处理Calvin

03

6.4.3 无协调事务处理Berkeley Anna

04

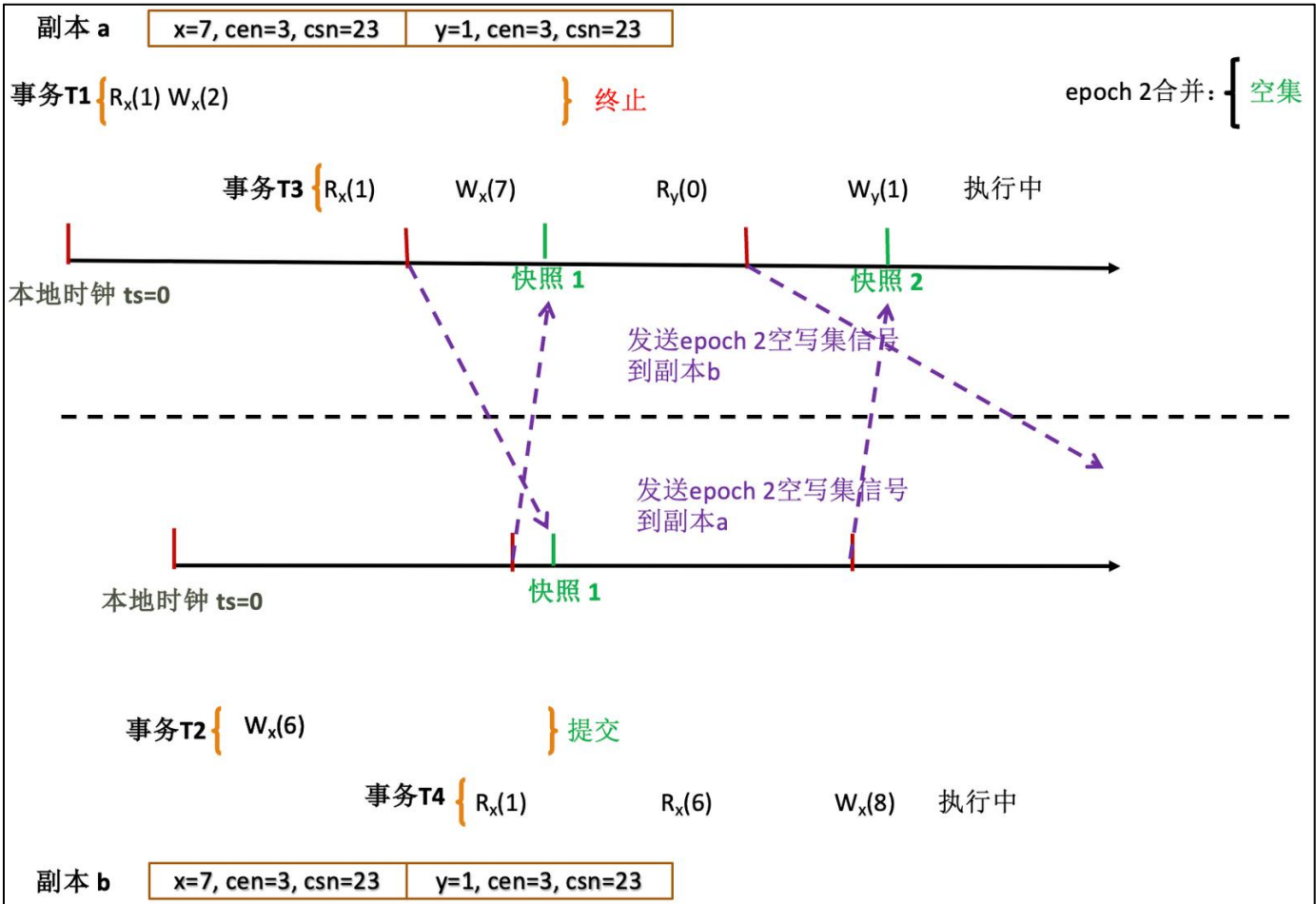
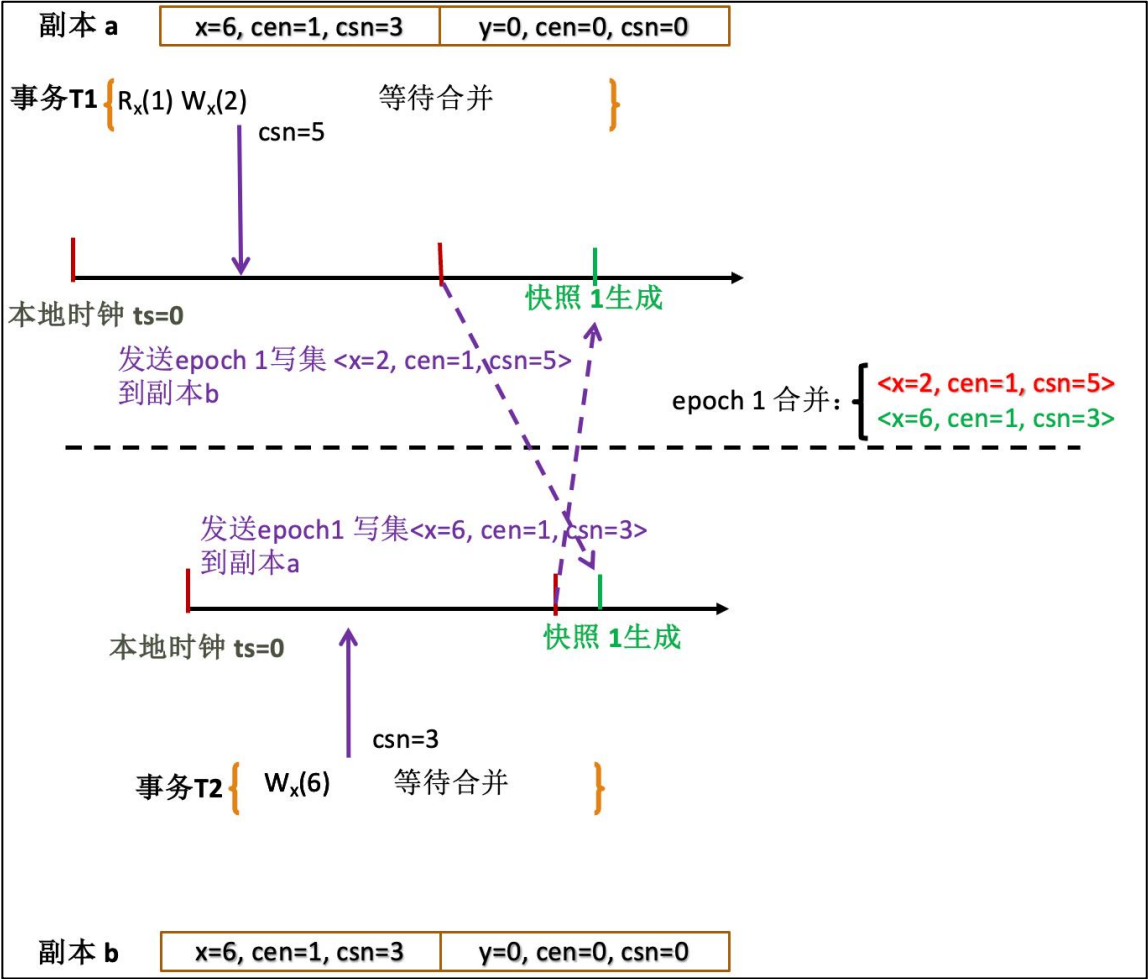
6.4.4 多主乐观事务处理GeoGauss



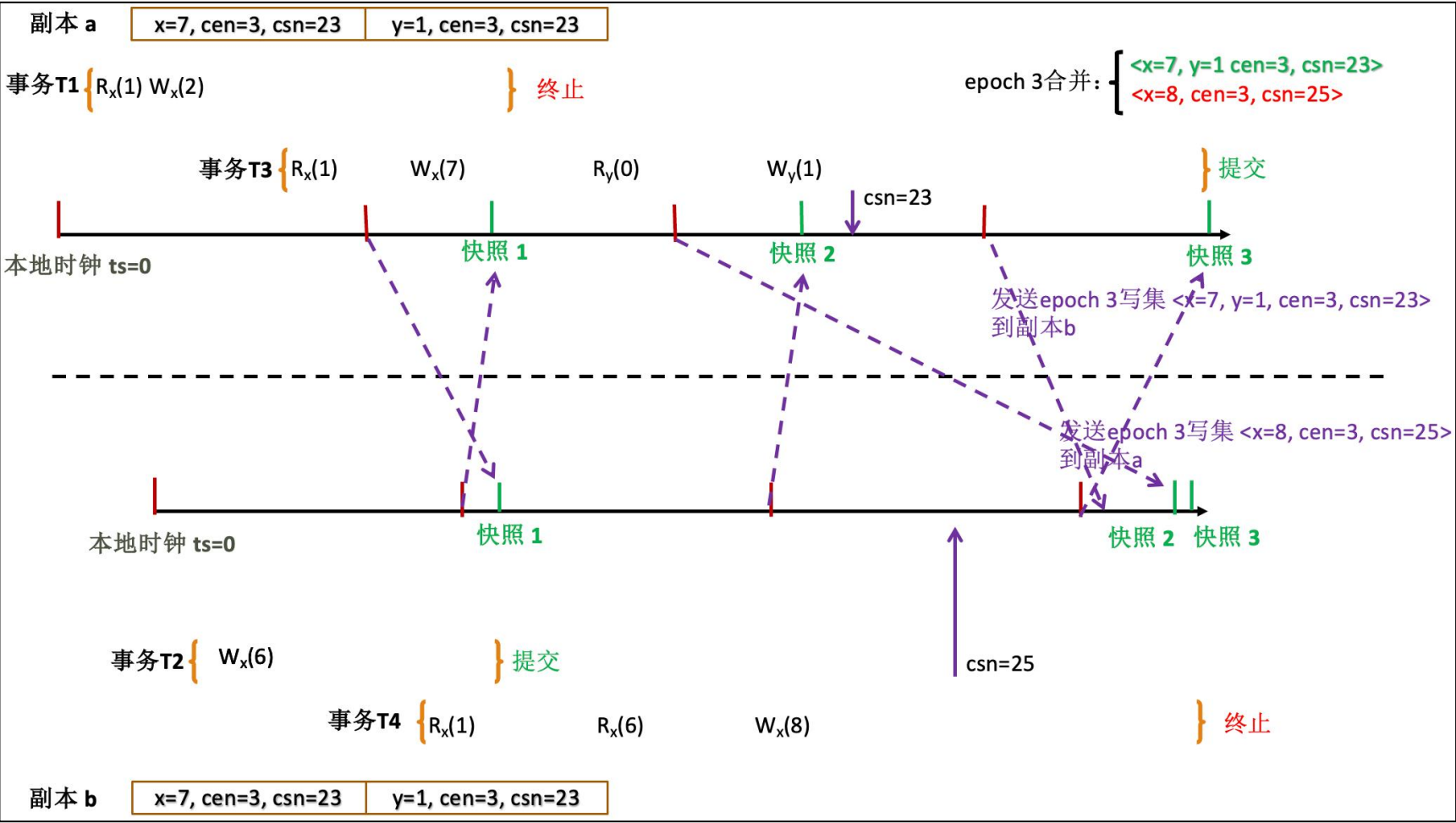
6.4.4 多主乐观事务处理GeoGauss

- GeoGauss参考Anna的CRDT方法，在无需协调的情况下，解决多主架构下的事务并发冲突。通过这种方式，能够避免多主在跨区域场景下的协调通信开销。为了解决引入CRDT而无法实现副本强一致性的问题。GeoGauss基于Epoch的协调通信，每个副本收集一段时间内的事务写集合并，在epoch（例如10ms）结束时，每个副本将当前时间段内收集到的写集广播给其他副本。副本在收集到其他所有副本当前epoch的写集后，利用CRDT特性，对事务写集进行合并，解决事务并发冲突。在副本完成一个epoch的写集合并后，会基于上一个快照，将完成合并的写集更新到快照上，生成新的快照，实现事务的快照隔离级别。
- GeoGauss是一种基于Epoch的多主乐观事务并发控制算法，使用CRDT的ACI属性，避免了事务并发冲突处理中的协调开销。
- 事务处理以Epoch为粒度，在多个副本中同步各个Epoch的写集，在轻量级的协调开销下，解决了使用CRDT技术只能保证副本最终一致性的问题，实现了多副本的顺序一致性

GeoGauss并发更新处理流程



GeoGauss并发更新处理流程



GeoGauss优势:
通过采用基于epoch的多主乐观事务并发控制算法, GeoGauss在轻量的协调开销下, 实现了副本的顺序一致性, 并避免事务冲突时的协调处理, 极大的降低了副本之间的协调开销。

本章小结

- 6.1节：介绍了分布式事务的概念和事务的ACID特性，了解了事务的ACID特性是数据库完整可靠的重要保证，并通过事务之间的多种并发异常，介绍了五种隔离级别，包括读未提交、读已提交、可重复读、快照隔离和可串行化
- 6.2节：介绍了四种并发控制算法以及它们的分布式实现，包括悲观并发控制、基于时间戳的并发控制、乐观并发控制和多版本并发控制
- 6.3节：介绍了分布式两阶段提交协议，包括两阶段提交协议流程、两阶段提交的故障恢复和优化、以及它与多副本共识的混合设计
- 6.4节：拓展介绍了四种前沿的事务处理方法，包括Spanner跨地理区域事务处理、Calvin确定性事务处理、Anna无协调事务处理、和GeoGauss多主乐观事务处理

第六章 完