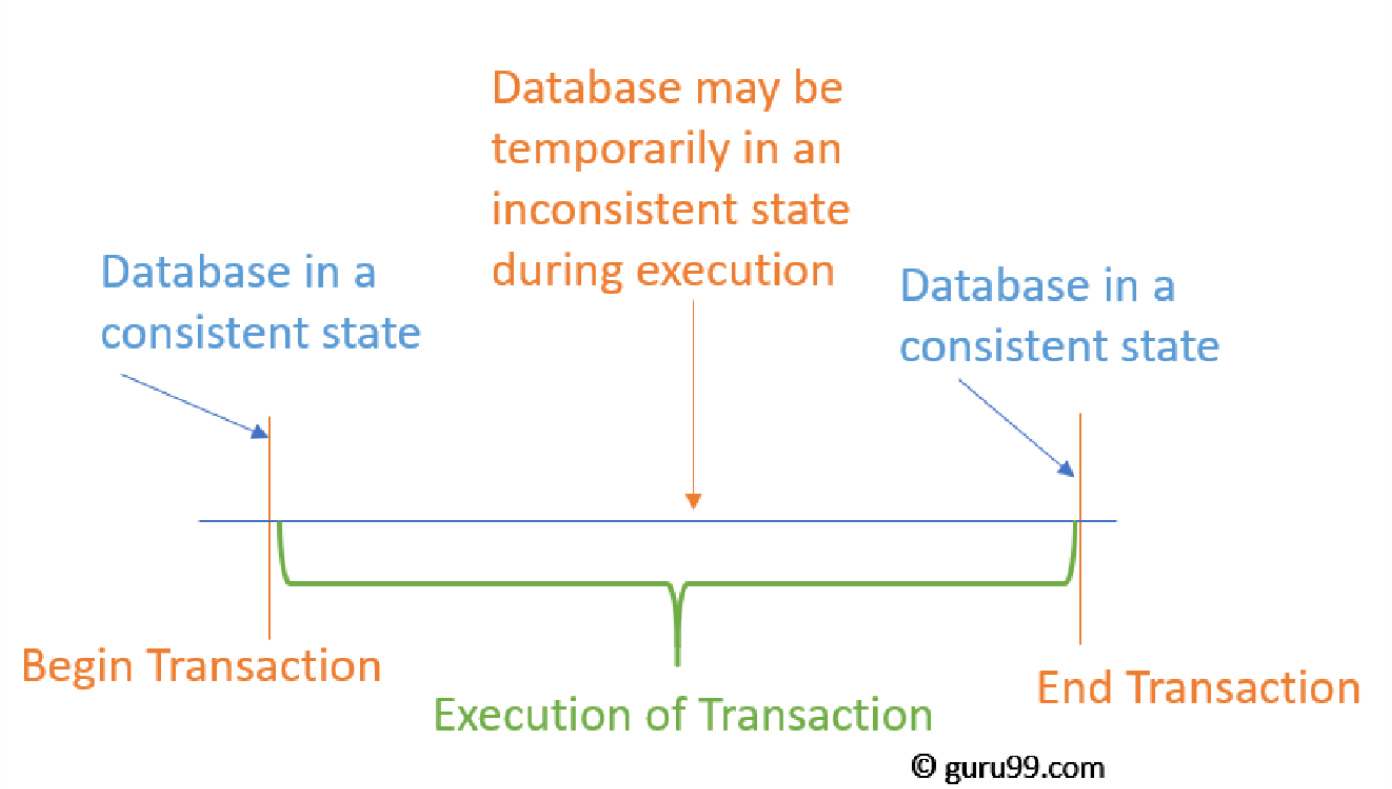
什么是「事务」？



数据库事务通常包含了一序列的对数据库的读/写操作，是一系列操作的集合。包含有以下两个目的：

1. 为数据库操作序列提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。
2. 当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

当事务被提交给了数据库管理系统（DBMS），则DBMS需要确保该事务中的所有操作都成功完成且其结果被永久保存在数据库中，如果事务中有的操作没有成功完成，则事务中的所有操作都需要回滚，回到事务执行前的状态；同时，该事务对数据库或者其他事务的执行无影响，所有的事务都好像在独立的运行。

为什么要有「事务」？

事务是关系型数据库引入的概念，一个现实的业务操作往往关联多处数据，需要在数据库层面提供一种底层的逻辑保障，确保对数据的操作符合一定的特性，简化业务逻辑。
事务transaction解决以下问题：

- 逻辑操作的原子性(**Atomicity**)：

  一系列的逻辑操作要么成功的反应在数据库中，要么完全不反映在数据库中（意味着如果有任何一处失败，整个操作需要回滚）。

- 数据约束的一致性(**Consistency**)：

数据库的表之间存在各种约束条件，比如主键约束、外键约束等，在没有其他事务并发执行的情况下，对数据库的修改需要符合数据库的一致性约束。

- 并发执行的隔离性(**Isolation**)：

  数据库需要能够并发执行多个事务，对于每个事务来说，其他事务是不可见的；整理看起来，事务并发执行的结果跟按顺序一个一个串行地执行结果一样。

- 执行结果的持久性(**Durability**)：

  一旦对数据库操作结果返回成功，就代表这个改变是永久的，不管数据库系统出现什么故障，比如断电、系统故障、磁盘故障等等（意味着数据库需要在保证性能的情况下具备恢复能力，即，每个成功的操作均需要存储在非易失性存储器上）

「事务」跟「SQL」什么关系?

事务是数据库逻辑逻辑的基本单元，由高级数据操纵语言编写；SQL全称为「结构化查询语言(Structured Query Language, SQL)就是数据库高级数据操纵语言的一种。在SQL语言的语法中，除了大家经常用到的`select`、`insert`、`delete`、`update`以外，也有赋值和其他逻辑操作。一个Transaction包含一条或者多条SQL语句，也可以不包含SQL语句。

什么时候一个事务算成功执行?

由于有「原子性」的要求，每个事务都有一个状态机，包含：

- 活动状态(active)
- 部分提交(partially committed)
- 失败(failed)
- 中止(aborted)
- 提交(commit)

只有当事务的commit日志记录到硬盘之后，日志才会进入提交状态，此时事务算是成功执行。
对于一些存在「可见外部写(observable exernal writes)」的事务，一般都是在提交状态之后操作。一个复杂的例子是ATM机：系统状态正常，但是吐钱的时候发生故障。

「原子性」如何实现?

- 原理：影子拷贝(shadow copy)，在数据库副本上执行事务操作，如果成功，则更新数据库指针指向新的copy。
- 前提：数据库指针存储在磁盘上，更新磁盘上数据库指针的操作要确保原子性，要么写入新的指针，要么原指针不被擦除。这个由磁盘提供原子性支持，磁盘系统确保了对块或者磁盘扇区更新的原子性。
- 其他案例：文本编辑器在编辑的时候，也通过shadow copy的方案来确保编辑会话的事务性。

「持久性」如何实现?

持久性的需求：事务成功返回，则记录必须有效保存在非易失存储器（比如硬盘）上。

困难

- 每次事务都直接更新保存在硬盘上的数据库数据文件，不仅性能不行，也无法做到错误恢复；
- 如果在内存中加buffer，将一批事务批处理更新到数据库数据文件，则无法做到持久（系统随时有断电风险）

解决方案

每次事务的修改保存在硬盘上的「更新日志记录(update log record)」，而不是直接修改数据库「数据文件」。

日志文件和数据文件相比，虽然都是保存在硬盘上，但是有两个优点：

1. 效率更高：日志是「顺序」的，又叫顺序日志，记录都是append在上面，不需要磁头寻址；而数据文件是「随机」的，每次都需要寻址。几乎所有的磁盘针对数据库日志文件更新效率都很高。
2. 确保事务的原子性：一个事务对应的日志包括start、commit作为开头和提交，只有日志完整的事务，才会被批量更新到数据文件。

此外，日志文件的更新可以通过在内存中加buffer（日志记录缓冲）实现批处理，进一步提高效率，毕竟将数据输出到硬盘上开销还是比较高。在这种方案下，只要日志没有真正写入硬盘，事务就不进入提交状态。在高并发的情况下，让事务阻塞几纳秒，但是极大提高了日志写的效率，系统整体效率还是提升的。

日志文件中还可以加入「检查点(checkpoint)」,让数据库系统在恢复的时候可以少做一些工作，不需要从头开始redo日志记录，而是从最近的检查点开始。

「隔离性」如何实现?

隔离性是针对事务并发的场景，需要达到的目标：

- 事务尽可能并行执行，但是执行效果跟事务串行执行一样。
- 提高系统的吞吐量(throughput)和处理器、磁盘的利用率(utilization)
- 同时减少事务的平均响应时间(average response time)

调度(schedule)：一个事务组的操作的**绝对执行顺序**就是一个调度。

冲突(conflict)：当两个事务 T_i、T_j 对同一数据项 x 进行操作，其中至少有一个是 `write` 操作时，事务 T_i 和 T_j 是冲突的。有冲突存在，决定了产生冲突的两个事务的冲突操作先后顺序不能改变。

冲突可串行化：在不改变一个调度中同一数据项产生操作冲突的操作的先后顺序的情况下，可以将调度变成一个串行调度，则表示这个调度是冲突可串行化的。

困难

1. 调度可串行化：对于多个事务组成的事务组，每个事务包含多个操作（`read`、`write` 以及 数据修改操作），如何产生一个冲突可串行化的调度，同时尽可能的提高执行效率（并行执行）
2. 调度可恢复&无级联：对于一个调度，如果某一个事务 T_1 失败了，确保整个调度是可以恢复的(recoverable)，也就是：既可以 `redo` 执行失败的事务，而不影响其他事务；也可以 `rollback` 回滚失败的事务，而不会导致其他事务级联回滚(cascading rollable)

思路

1. 为了保证调度的可串行性，首先要明确什么情况下会出现冲突，其次要确定处理冲突的方式。

- 定义冲突的过程中，我们可以对事务操作进行规范，降低复杂度。比如两阶段加锁、有效性检查等方法，都约定了事务操作的规范。
- 解决冲突的方式，要么延迟冲突的操作（比如加锁），要么终止发出冲突操作的事务（比如时间戳协议、有效性检查）。

1. 为了保证调度的可恢复性和无级联性，需要确保「当T2事务读取了T1事务所写的数据项，那么T1事务必须T2读取该数据项之前提交」。

可串行化判定

冲突可串行化判定原理：事务 T_i 对数据项 x 的 `read`、`write`操作和事务 T_j 对数据项 x 的 `read`、`write` 操作决定了他们的串行顺序。以下两种情况均表示 T_i 依赖 T_j（$T_i \rightarrow T_j$）

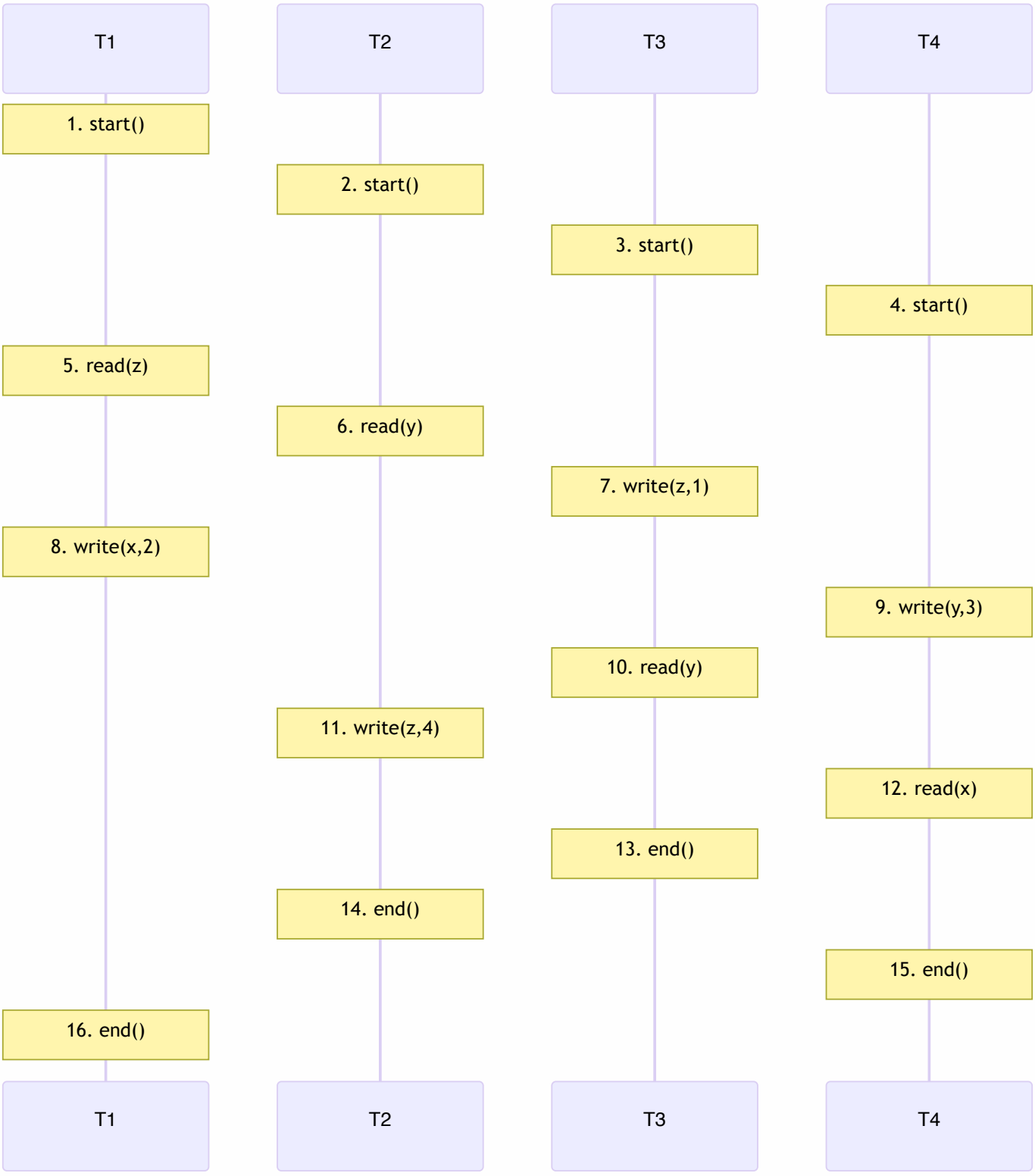
- T_i 事务 `read` 操作前，如果 T_j 有 `write` 操作
- T_i 事务 `write` 操作前，如果 T_j 有 `write` 或 `read` 操作
  （两个事务的 `read` 操作无法确定依赖关系，任何对同一数据项 的`read`、`write` 操作可以确定一组依赖关系）

	Read	**Write**	**Increase**	**Decrease**
Read	OK	依赖	依赖	依赖
Write	依赖	依赖	依赖	依赖
Increase	依赖	依赖	OK	OK
Decrease	依赖	依赖	OK	OK

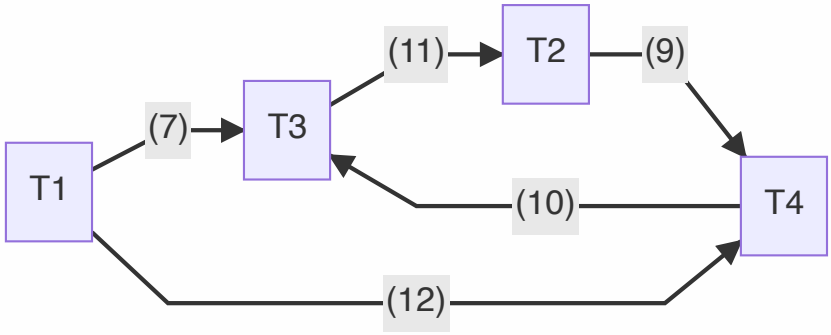
> 例如：两个事务 T_1、T_2 对同一数据项 x 进行操作。在 T_1 的操作 `write(x,1)` 之后 T_2 中有操作 `read(x,1)`。我们可以得到一组依赖关系：$T_1 \rightarrow T_2$

根据依赖关系可以画出调度的优先图**(precedence graph)**，如果该图是无环图，可以通过拓扑排序获得等价的串行(serialisabilite)调度；否则表示这个调度是非串行的。

> 例：


T1
T2
T3
T4
1. start()
2. start()
3. start()
4. start()
5. read(z)
6. read(y)
7. write(z,1)
8. write(x,2)
9. write(y,3)
10. read(y)
11. write(z,4)
12. read(x)
13. end()
14. end()
15. end()
16. end()
T1
T2
T3
T4

> 调度优先图为：



> 由图可见，该图内存在环，所以不符合可串行性。

可串行性解决方案

方案1：加锁（悲观）

两阶段封锁协议(two-phase locking protocol, 2PL)：将事务加锁、解锁请求分成两个阶段，所有**加锁请求必须在第一阶段**（增长阶段growing phase）申请，**所有解锁请求必须在第二阶段**（缩减阶段shrinking phase）申请，**加锁和解锁操作不能交叉执行（同一个事务内）**。

两段锁协议规定所有的事务应遵守的规则：

> ① 在对任何数据进行读、写操作之前，首先要申请并获得对该数据的锁。
>
> ② 在释放一个锁之后，事务不再申请和获得其它任何锁。

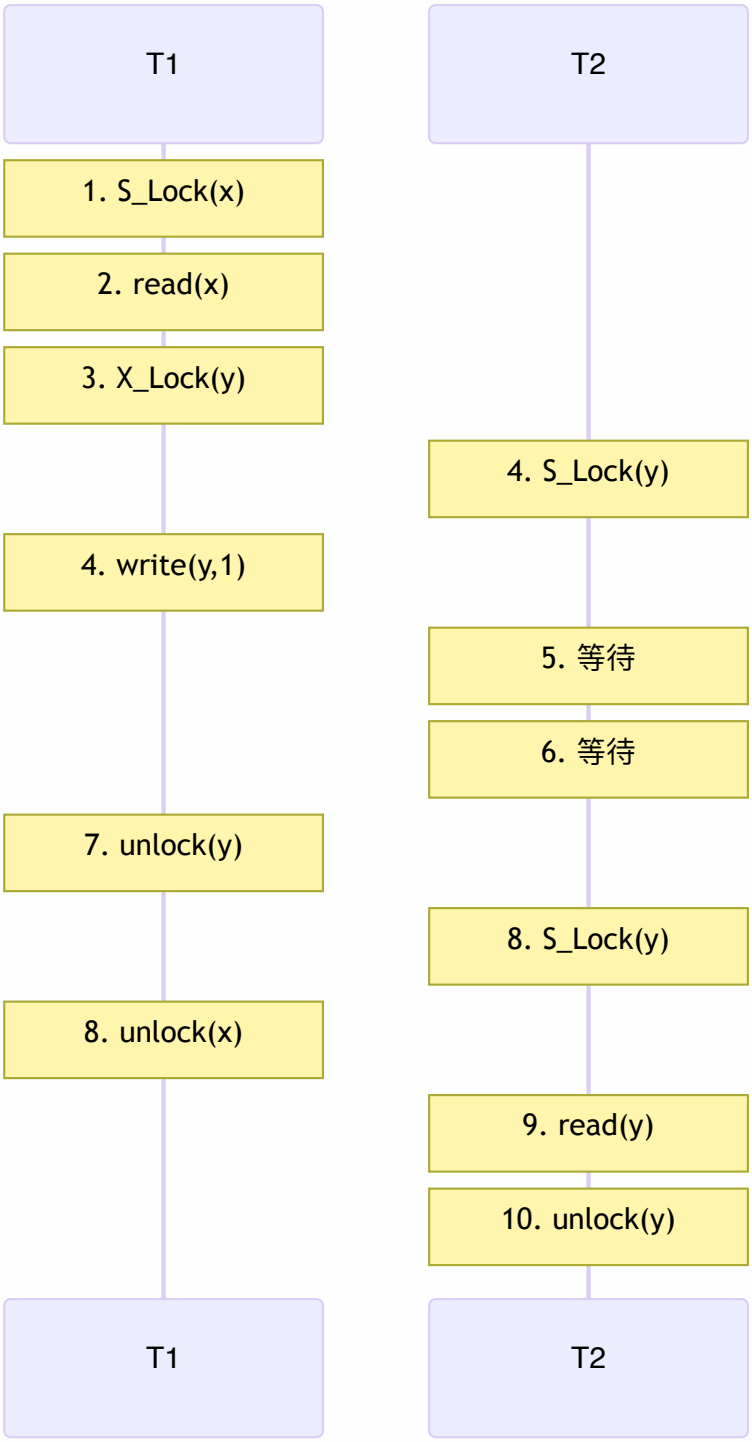
1. 第一阶段（增长阶段Growing Phase）：

   其实也就是该阶段可以进入加锁操作，在对任何数据进行读操作之前要申请获得S锁；在进行写操作之前要申请并获得X锁，加锁不成功，则事务进入等待状态，直到加锁成功才继续执行。
   就是加锁后就不能解锁了。

2. 第二阶段（缩减阶段Shrinking Phase）：

   第二阶段是释放封锁，事务可以释放任何数据项上的任何类型的锁，但不能申请。

   当事务释放一个封锁后，事务进入封锁阶段，在该阶段只能进行解锁而不能再进行加锁操作。



2PL协议解决「串行化」问题的思路是通过把可能冲突的数据操作集中在申请锁的阶段，由锁的不相容性延迟有冲突的事务，从而确定冲突事务的执行次序。这种加锁方式也避免了事务解锁了一部分锁之后又去申请新的锁，导致脏读的问题。但是，2PL协议无法解决「死锁(dead lock)」问题和级联回滚(cascading rollback) 问题。

严格(strict)两阶段封锁协议：规定持有排他锁的事务必须在事务 `commit` 之后才能释放锁。避免了级联回滚。
强(rigorous)两阶段封锁协议：规定持有任何锁的事务必须在事务 `commit` 之后才能释放锁。比strict模式更严格，避免了死锁。

大部分数据库要么采用strict 2PL，要么rigorous 2PL。为了提高并行性能，允许在增长阶段将共享锁S升级(upgrade)为排他锁X，同时在缩减阶段将排他锁X降级(downgrade)为共享锁S，这样确保在write的时间点才排他，操作完之后就不排他，减少事务等待时间。

重点注意：使用2PL，虽然可以确保冲突可串行化，但是不代表它的执行效果跟串行一样。因为加锁状态下的可串行化是理论上的可串行化，是根据每个事物的封锁点(lock point，也就是最后加锁的位置)排序得到的串行化顺序，而调度实际执行的时候主要依靠「锁」来保证对冲突操作的执行顺序，这个是并行执行的，因此会出现不同的事务都优先拿到对方需要的数据集的锁的情况（死锁）。

数据多粒度(granularity)封锁：理论情况下，加锁针对的数据项Q是没有粒度的。实际中，数据项在保存的时候，经常多个数据项聚成一个数据单元。这种情况下，如果要加锁一个范围比较大的数据时，需要挨个数据项加锁，比较费时；如果需要加锁一个范围比较小的数据时，又会锁定无关的数据项，导致整体并发性减弱。为数据建立*多粒度*机制就是解决方案。该方案根据数据在数据库中的存储特点，从「数据库」到「区域的文件节点」再到「记录」，形成一颗提现粒度层次的树。并引入「共享型意向锁IS」、「排他型意向锁IX」、「共享排他型意向锁SIX」，实现多粒度封锁。

多版本两阶段封锁(Multiversion two-phase locking protocol)：2PL的并行性能有待提升，尤其是在大部分事务「只读」，小部分事务是「更新」操作的情况下，只读的事务会被延迟到更新事务完成，这种情况可以通过结合多版本的特性进行提升。
多版本的2PL协议，将事务分为「只读事务」和「更新事务」，针对每一个数据项 x 维护一个全局的版本号ts-counter（计数器型时间戳），事务 T_i 执行 `read(x)` 操作将返回时间戳小于 $TS(T_i)$ 的最大时间戳版本的内容；`write(x)` 操作会增加全局版本号以及写入 x 的新版本。这种情况下，只读事务永远不需要等待锁，更新事务则执行强两阶段封锁协议，确保他们可以按照提交次序串行化。

方案2：时间戳规则（乐观）

时间戳排序协议(timestamp-ordering protocol)：不同于加锁的方式，通过锁的相容性决定冲突事务执行顺序，时间戳排序机制**通过事务的时间戳**来决定事务串行化的次序。

1. 为每一个数据项 x 关联 `read`、`write` 操作最近一次成功执行的时间戳 $R_{ead}T_{imestamp}(x)$　$W_{rite}T_{imestamp}(x)$。
2. 当事务 T_i 对数据项 x 进行 `write` 操作的时候，如果在事务 T_i 开始之后，x 被其他事务 `read` 或者 `write` 过，则当前事务 T_i 进行回滚。

> 基本时间戳排序协议的工作原理如下：
>
> - `TS(Ti)`表示事务 T_i 的时间戳。
> - `RT(x)`表示数据项 x 的读时间戳。
> - `WT(x)`表示数据项 x 的写时间戳。
>
> 1. 每当事务 T_i 发出 `read(x)` 操作时，请检查以下条件：
>
> - 如果 $WT(x) > TS(T_i)$ 则**拒绝该操作**；
> - 如果 $WT(x) \leq TS(T_i)$ 则**执行操作**；
> - 更新所有数据项的时间戳。
>
> 2. 每当事务 T_i 发出 `write(x)` 操作时，请检查以下条件（**Thomas 写规则**）：
>
> - 如果 $TS(T_i) < RT(x)$，则表明 T_i 准备写的值还没来得及写入，x 就提前被读取了，所以 T_i 的 `write(x)` 操作被拒绝，并且事务 T_i 被回滚
> - 如果 $TS(T_i) < WT(x)$，表明 T_i 写的值已过期，比它更新的值已经写到 x 上，所以 T_i 的 `write(x)`操作被拒绝；
> - 剩下的情况，`write(x)` 操作被允许；
> - 更新所有数据项的时间戳。

时间戳排序协议**不会有死锁**，因为没有事务处于「等待」状态，事务发现数据项被后来者动过之后就回滚了。但是，这种规则有很强的「抢断」性质，容易导致**长事务持续被短事务抢断，长事务反复重启，可能导致饿死**。

Thomas写规则：对时间戳排序协议的性能优化，当事务 T_i 对数据项 x 进行`write`操作的时候，如果在事务 T_i 开始之后，x 被其他事务`write` 过，则当前事务 T_i 的 `write` 操作**忽略**，而不是进行回滚。通过这种方式减少不必要的回滚。

多版本时间戳排序机制(multiversion timestamp-ordering scheme)：类似多版本两阶段封锁协议，让每个 `write(x)` 操作创建 x 的新版本，而 `read(x)` 操作则会被分配一个合适的 x 版本进行读取，提高「读」的效率。

方案3：Validation 有效性规则（乐观）

有效性检查协议：假设每个事物Ti的生命周期分为2个阶段（只读事务）或者3个阶段（更新事务）。读阶段只进行读、计算操作，之后进行$validation$ 检查，通过的话就进行写操作。每个阶段都关联一个时间戳：$Start(T_i)$，$Validation(T_i)$，$Finish(T_i)$。有效性测试的原理跟时间戳规则一样，

> *如果事务 T_i 在写数据项 x（有效性验证）的时候发现数据项已经被写过了，则事务 T_i 回滚。*

相比时间戳排序，有效性检查协议把 $validation$ 的时间作为事务的时间戳，而不是使用事务开始的时间，这样可以在冲突度低的情况下有更快响应。

有效性规则是一种**乐观并发控制(optimistic concurrency control)**

非可串行性解决方案：增强并发性能的「弱一致性级别」

为什么要有非可串行化解决方案

在某些应用中，串行化方案可能会影响并发性能，但是应用并不需要精确的信息，因此可以牺牲串行性而提高并发性。

二级一致性

弱一致性要解决的问题

1. 脏读(Dirty read)：当前事务读取的数据项已经过时，原因是其他事务已经修改了该数据，但是并未最后提交（也可能最终回滚）。总之，当前事务读到的数据是不可靠的，是「脏数据」。
2. 不重复读(Unrepeatable read)：同一个事务连续两次读取同一数据项 x，间隙之间其他事务修改了数据项 x，因此本事务先后两次读到的数据结果会不一致。
3. 幻读(Phantom Read)：当前事务进行某个范围数据的读或者写的操作的同时，另一个事物插入了符合当前事务读写约束的新数据，当前事务再更新时，惊奇地发现了这些新数据，貌似之前读到的数据是幻觉一样。

弱一致性的级别：由弱到强一次解决上述3个问题

- 未提交读(Read uncommited)：读操作不加锁，可能读到已经过时的数据，存在「脏读」的问题。
- 已提交读(Read commited)：只允许读取已提交记录，但是不要求可重复读。事务连续两次读取同一数据之间会释放锁，因此两次读取结果不保证一样。解决了「脏读」，但是不解决「不重复读」。
- 可重复读(Repeatable read)：只允许读取已提交记录，且同一个事务两次读取同一数据项 x 得到的结果是一致的。事务在提交之前不会释放锁。解决了「脏读」、「不重复读」问题，但是解决不了「幻读(Phantom Read)」。
- 串行化(Serializable)：执行结果跟串行执行一样。解决了「幻读」问题。

隔离级别	脏读	不可重复读	幻读
未提交读(Read uncommited)	N	N	N
已提交读(Read commited)	Y	N	N
可重复读(Repeatable read)	Y	Y	N
串行化(Serializable)	Y	Y	Y

高效的数据库系统为什么对操作系统是有要求?

数据库系统进行数据操作的逻辑单元--事务，包含两个访问数据的核心操作：

- `read(x)`：从数据库中把数据项 x 传送到执行 `read` 操作的事务的局部缓冲区
- `write(x)`：从执行`write(x)` 操作的事务的局部缓冲区把数据项 x 传回数据库

困难

数据库系统为了实现「持久性」，必须遵守先写日志（write-ahead logging，WAL）规则，这样从数据库文件中读出的存放在内存中的数据块就不能由操作系统自由写回，而应该由数据库系统根据日志的完成情况，强制输出缓冲块。

解决方案

1. 数据库系统保留部分内存作为缓冲区并进行管理，而不是让操作系统来管理。

- 优点：由数据库管理，可以完全按照符合数据库规则的方式使用，效率高。
- 缺点：这部分保留内存无法被数据库以外的应用使用，限制了内存使用的灵活性。

1. 数据库在操作系统提供的虚拟内存中实现缓冲区。

- 缺点：大部分操作系统都会完全控制虚拟内存，导致当数据库系需要输出某个数据块时（保存在虚拟内存中），

需要更多次的操作。

死锁(Dead lock)如何处理？

死锁就是循环「等待(wait)」，只要事务需要等待，就有可能存在死锁。处理死锁问题的思路：

1. 死锁预防(deadlock prevention)：对加锁请求进行排序或者要求同时获得所有锁来确保不会发生循环等待。
2. 死锁恢复(deadlock recovery)：在检测出死锁之后，根据一定的规则选择某些事务进行回滚，打破循环等待的状态。

参考

- Database System Concepts（数据库系统概念）
- 『浅入深出』MySQL 中事务的实现
- MySQL的InnoDB的幻读问题