W8

Concurrency

数据库中的并发控制确保调度是冲突可串行化或视图可串行化的、可恢复的，并且最好无级联。

理想的并发控制协议在保证串行化的同时，不会拖慢事务执行速度。

锁协议

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

锁协议是所有调度请求和释放锁的规则，限制了可能的调度集合

两种锁：

- **X mode** （exclusive）：允许事务读写资源
  - 其他事务都不能再对这个资源任何类型的锁
- **S mode**（Shared）：允许事务读取资源
  - 多个事务可以同时对同一资源持有 S 锁

锁兼容性矩阵(Lock-compatibility matrix)：

	S	X
S	true	false
X	false	false

If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions are released. The lock is then granted.

锁请求在兼容时才会被批准，否则会等待直至锁释放。

Two-Phase Locking Protocol

2PL 是保证冲突可串行化性的一种协议。

分为两个阶段：

1. Growing phase （获取）
   - 事务可以获取锁但不能释放锁。
2. Shrinking phase （释放）
   - 事务可以释放锁但不能获取锁。

事务在释放锁后无法获取任何锁，因此可以按照顺序串行化。

basic 2PL仍然可能发生死锁和级联回滚

- **Strict 2PL**：
  - 事务必须持有X锁直到commit/abort
  - 确保调度可恢复和cascadeless
- **Rigorous 2PL**：
  - 所有锁必须被持有直到commit/abort
  - 事务可以按照commit的顺序串行
- **Conservative 2PL**：
  - 事务在执行前获取所有需要的锁
  - 避免deadlock 但限制并行

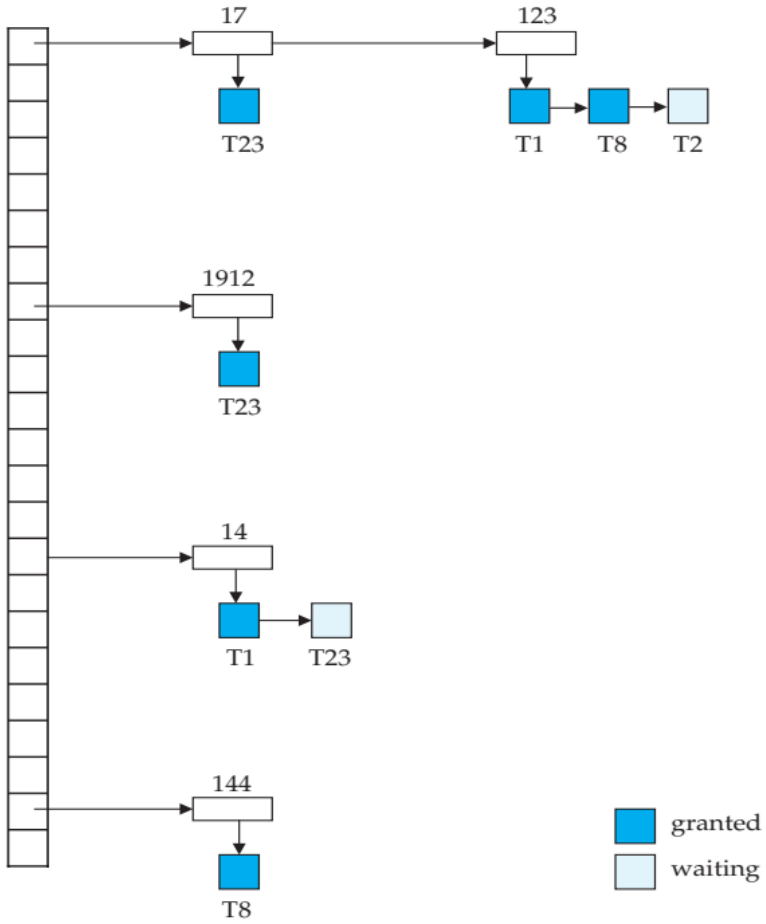
锁转化（Lock Conversions）

两阶段锁协议允许锁的升级和降级以提高并发性。

- **升级**：将 S 锁转换为 X 锁。
- **降级**：将 X 锁转换为 S 锁。

保证了可串行性但依赖手动添加锁指令

lock table



死锁

所有事务都在等待彼此释放其锁时，就会发生死锁。

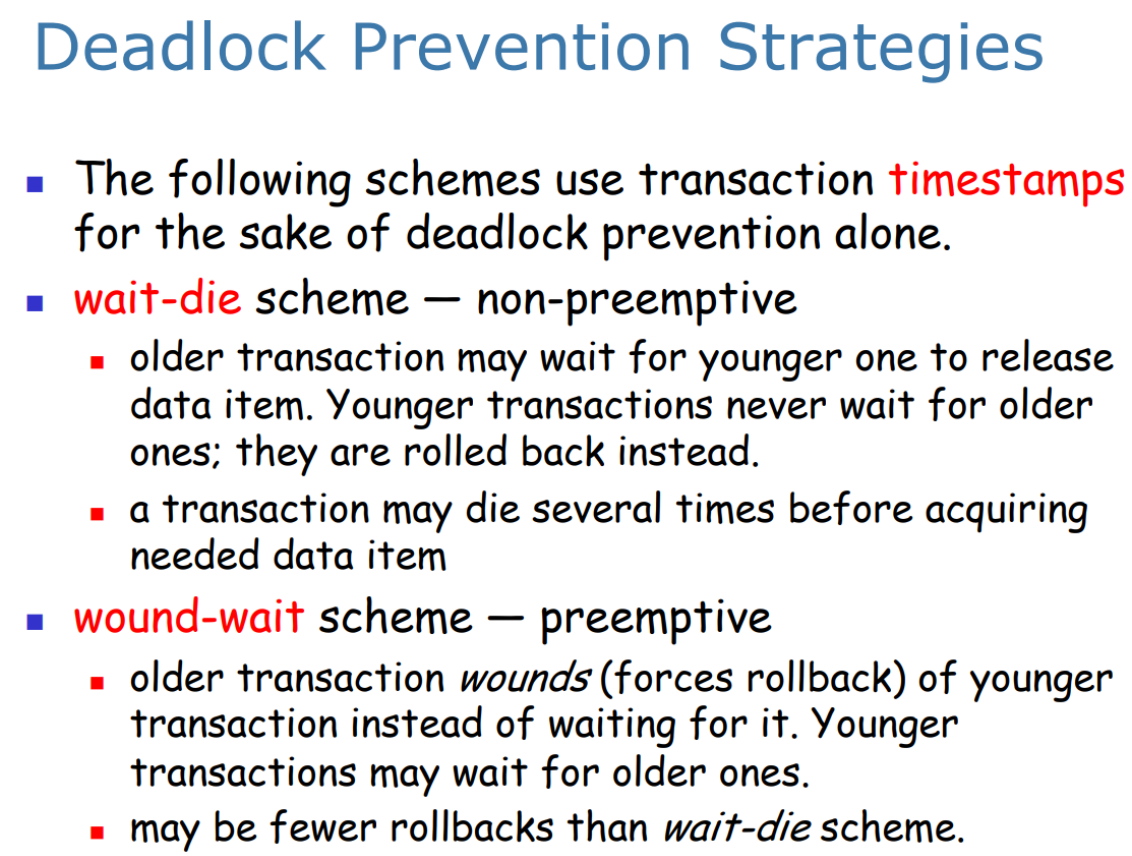
饥饿：比如事务因为反复roll back得不到处理

solution：设置roll back 次数

2PL中两种处理死锁的方法：

- **Deadlock prevention**：避免死锁发生。
- **Deadlock detection**：通过后台检查来检测并解决(break)死锁。

死锁预防



if T1 want a lock while T2 holds a conlicting lock:

Wait-Die Scheme (non-preemptive):

- (for T1) Older wait, younger die (abort).
- 老的会等新的，新来的滚（老的不会饿死）
- 事务可能会滚好几次

Wound-Wait Scheme (preemptive):

- Older transactions force younger ones to rollback. younger one wait.
- 老的让新的滚，新来的等（不会饿死且更少的回滚）

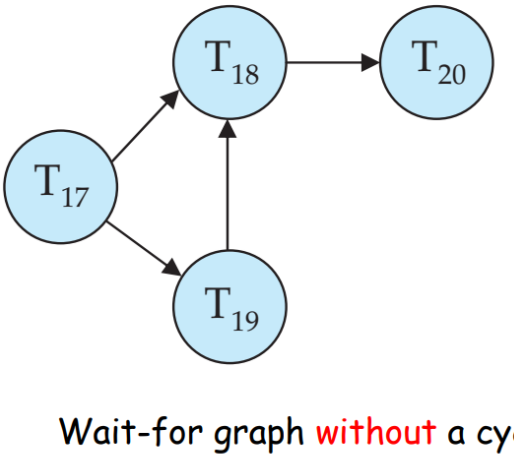
上述两个方法避免了饥饿，但不必要的回滚可能发生

Lock Timeout-Based Schemes:

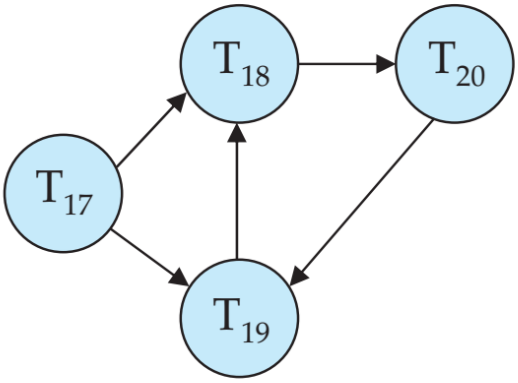
给事务的等待设置上限(timeout)，等够了就滚。

死锁检测

可以用 **wait-for graph** 描述，箭头是等待指向方释放的意思，有cycle则有死锁



Wait-for graph without a cycle



Wait-for graph with a cycle

死锁恢复

当检测到死锁，三种解法：

1. 开销最小的事务作为victim回滚
2. Total rollback：abort transaction and restart it
3. 如果同一个事物总是作为victim会饥饿
    - 加入回滚次数在cost factor来避免

Failure Recovery

事务故障（Transaction Failure）

- logical errors：事务因内部错误无法完成，例如违反约束
- system errors：数据库系统因错误中止活动事务（如死锁）。

系统崩溃（System Crash）

- 定义：由于硬件或软件故障导致系统崩溃（如断电）
- 假设：
    - 假定非易失性存储（如磁盘）不会在系统崩溃中损坏。
    - 数据库系统通过完整性检查防止数据损坏。

磁盘故障（Disk Failure）

- 磁盘硬件损坏可能导致部分或全部数据丢失。
- 驱动程序通过校验和检测故障。

Storage Categories

1. **易失性存储（Volatile Storage）**
    - **特点**：系统崩溃后数据丢失。
    - **示例**：内存、缓存。
2. **非易失性存储（Non-Volatile Storage）**
    - **特点**：可抵御系统崩溃。

- **示例**：磁盘、闪存、电池供电的 RAM。
- 仍然可能fail

3. **稳定存储（Stable Storage）**
   - **特点**：理论上可抵御所有故障。
   - **实现方式**：通过多份数据副本在不同非易失性介质上存储模拟。

Recovery Algorithms

恢复目标

1. 在发生故障后，将数据库恢复到一致状态。
2. 确保数据完整性(integrity)与一致性(consistency)，防止数据丢失。

算法的两部分

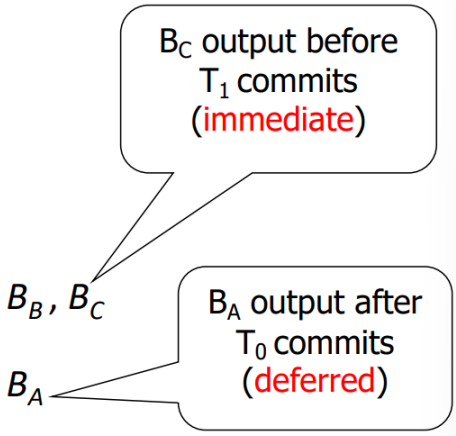
1. 正常事务处理期间：
   - 记录足够的信息以便从故障中恢复。
2. 故障后：
   - 恢复数据库内容，确保atomicity, consistency and durability.。

Log-Based Recovery

1. **定义**：
   - **Log**是一系列记录事务更新活动的序列，存储在**稳定存储（Stable Storage）**中。
   - 日志记录格式：
     - `<Ti start>`：事务开始时记录。
     - `<Ti, X, V1, V2>`：事务更新某数据项前后的值。
     - `<Ti commit>` 或 `<Ti abort>`：事务提交或中止时记录。
2. **日志的两种使用方式**：
   - 延迟更新（Deferred Modification）：
     - 事务提交后才更新数据库。
     - 恢复时简单，但需要存储本地副本。
   - 立即更新（Immediate Modification）：
     - 未提交的更新可以写入磁盘。
     - 恢复时更复杂，但更灵活。

当commit log 输出到稳定存储时一个事务才算提交了

Log	Write	Output
<T_0 start>		
<T_0, A, 1000, 950>	A = 950	
<T_0, B, 2000, 2050>	B = 2050	
<T_0 commit>		
<T_1 start>		
<T_1, C, 700, 600>	C = 600	
<T_1 commit>		



- Note: B_X denotes block containing X.

所有的并发事务共享一个disk buffer and log

假设使用 strict two-phase locking （当一个事务处理这个资源的时候，其他事务都不准处理它直到commit/abort）

Undo and Redo

Undo:

- 通过将旧值 `v1` 写回数据项 `x` 恢复事务的原始状态。从下往上
- additional log: `<Ti, X, V1>` and `<Ti abort>` when complete

Redo:

- 将新值 `v2` 写入数据项 `x`，确保事务的操作反映在数据库中。从上往下
- 没有额外log

恢复场景:

- **undo**:
  - log包含 `<Ti start>`
  - 不包含 commit / abort
- **redo**:
  - log包含`<Ti start>`
  - 包含 commit / abort

that *failures occur immediately after the last statement*.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
		$<T_1$ commit>

Recovery actions in each case above are:

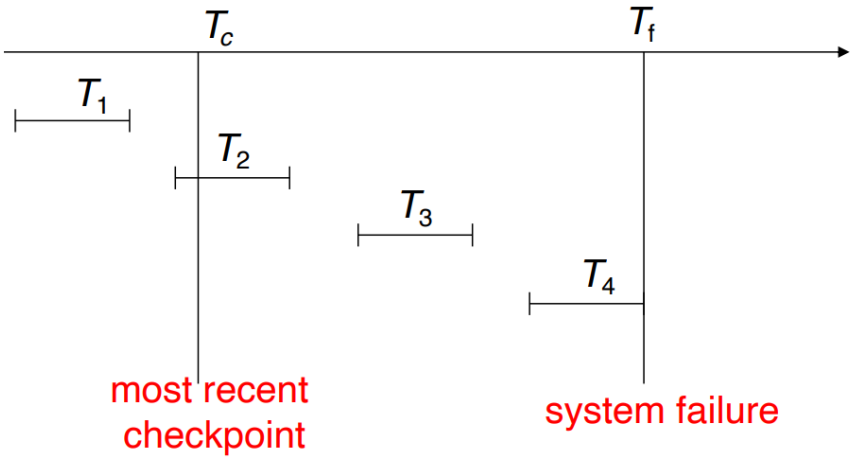
- (a) undo (T_0): B is restored to 2000 and A to 1000, and log records $< T_0$, B, 2000>, $< T_0$, A, 1000>, $< T_0$, **abort**> are written out
- (b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and *C* is restored to 700. Log records $< T_1$, C, 700>, $< T_1$, **abort**> are written out.
- (c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600. No additional log records need to be. Written out.

Note that if transaction T_i was undone earlier and $< T_i$ **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone

- such a redo redoes all the original actions including the steps that restored old values
  - Known as repeating history
  - Seems wasteful, but simplifies recovery algorithm greatly

Checkpoints

1. **定义**:
   - 检查点通过周期性地将日志和缓冲区内容写入稳定存储，简化恢复过程。
2. **步骤**:
   - 将内存中日志写入稳定存储。
   - 将修改的缓冲区块写回磁盘。
   - 写入 `<checkpoint L>`，其中 `L` 为活动事务列表。
   - check时所有其他操作暂停
3. **恢复时作用**:
   - 恢复时仅需处理最近检查点之后的事务，只有在checkpoint后面开始的事务需要redo or undo



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone (*but all instructions in T4 up to the failure point need to be redone*)

Recovery Algorithm

Logging

- 在事务操作期间记录 `<Ti start>` 和 `<Ti, Xj, V1, V2>` 等信息。

Recovery

1. Redo 阶段：
    - 重做从最近检查点后开始的所有事务操作。
2. Undo 阶段：
    - 撤销未完成的事务操作。