

Transactions Introduction and Concurrency Control

SUN YAT-SEN UNIVERSITY

Review

Database

Tables

Pages

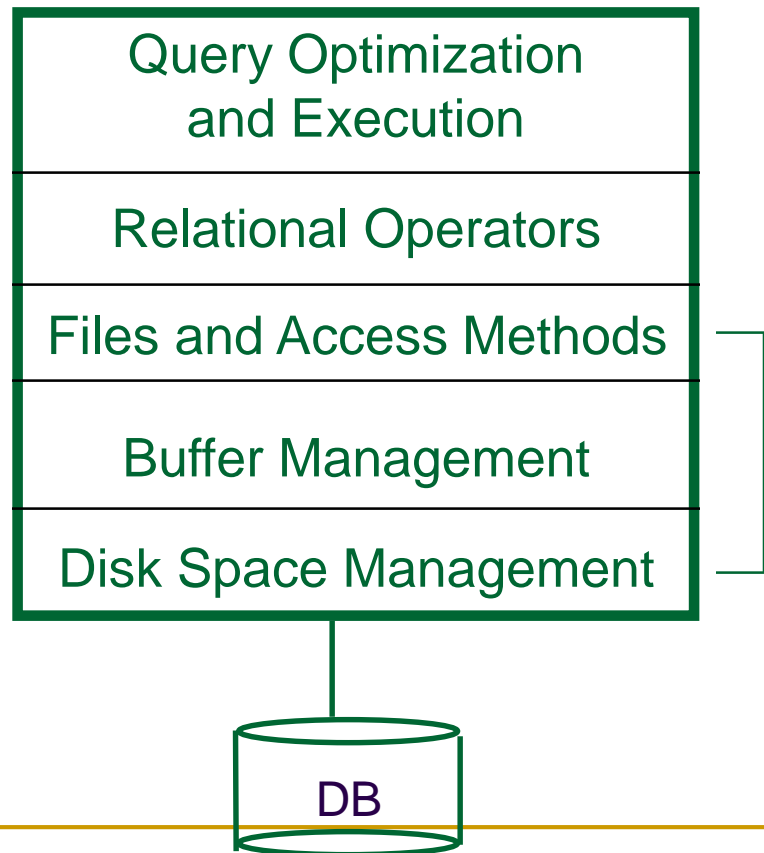
Tuples

- 数据库中的数据对象层次
- 数据库操作
 - 读: `select`
 - 写: `insert, delete, update`
- 完整性约束: 域约束、主键约束、外键约束...

Concurrency Control and Recovery

- Concurrency Control(并发控制)
 - Provide **correct** and **highly available**(高可用) data access in the presence of concurrent access by many users.
- Recovery(崩溃恢复)
 - Ensures database is **fault tolerant**(容错), and not corrupted by software, system or media failure
 - 24x7 access to mission critical data
- A boon to application authors!
 - Existence of CC&R allows applications to be written without explicit concern for concurrency and fault tolerance.

Structure of a DBMS



These layers must consider concurrency control and recovery (Transaction, Lock, Recovery Managers)

Transactions and Concurrent Execution

事务和并发执行

- Transaction (“xact”):
DBMS’s abstract view of a **user program** (or activity)
 - A sequence of **reads** and **writes** of database objects.
 - Batch of work that must **commit** or **abort** as an **atomic unit**
 - **Transaction Manager** controls execution of xacts.
 - User’s **program logic** is invisible to DBMS!
 - Arbitrary computation possible on data fetched from the DB
 - The DBMS only sees data read/written from/to the DB.
 - Challenge: provide atomic xacts to concurrent users!
 - Given only the read/write interface.
-

A Sample Transaction

```
1:  Begin_Transaction
2:  get (K1, K2, CHF) from terminal
3:  Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;
4:  S1 := S1 - CHF;
5:  Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
6:  Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;
7:  S2 := S2 + CHF;
8:  Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;
9:  Insert Into BOOKING (ACCOUNTNR, DATE, AMOUNT, TEXT)
      Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING (ACCOUNTNR, DATE, AMOUNT, TEXT)
      Values (K2, today, CHF, 'Transfer');
12: If S1 < 0 Then Abort_Transaction
11: End_Transaction
```

Concurrency: Why bother? 动机

- The *latency*(延迟) argument
 - **Response time**: the average time taken to complete an xact.
 - A **short xact** could get stuck behind a **long xact**, leading to unpredictable delays in response time.
- The *throughput*(吞吐量) argument
 - **Throughput**: the average number of xacts completed in a given time.
 - Overlapping I/O and CPU activity reduces the amount of time disks and CPU are idle.

ACID properties of Transaction Executions

- **A** tomicity –原子性:
 - ❑ All actions in the Xact happen, or none happen.
 - **C** onsistency-一致性:
 - ❑ If the **DB (Database)** starts consistent, it ends up consistent at end of Xact.
 - **I** solation-隔离性:
 - ❑ Execution of one Xact is isolated from that of other Xacts.
 - **D** urability-持久性:
 - ❑ If an Xact commits, its effects persist.
-

Implications of Atomicity and Durability

- A transaction ends in one of two ways:
 - *commit* (提交) after completing all its actions
 - “commit” is a contract with the caller of the DB
 - *abort* (中止, or be aborted by the DBMS) after executing some actions.
 - Or *system crash* while the xact is in progress; treat as abort.
- *Atomicity* means the effect of aborted xacts must be removed
- *Durability* means the effects of a committed xact must survive failures.
- DBMS ensures the above by *logging* all actions:
 - *Undo* the actions of aborted/failed xacts.
 - *Redo* actions of committed xacts not yet propagated to disk when system crashes.

Transaction Consistency

- Xacts preserve DB *consistency*
 - Given a consistent DB state, produce another consistent DB state
 - DB *consistency* expressed as a set of declarative **Integrity Constraints**
 - CREATE TABLE/ASSERTION statements
 - Xacts that violate ICs are aborted
 - That's all the DBMS can automatically check!
-

Isolation (Concurrency)

- DBMS interleaves actions of many xacts
 - Actions = reads/writes of DB objects
- Users should be able to understand an xact without considering the effect of other concurrently executing xacts.
- Each xact executes as if it were running **by itself**.
 - Concurrent accesses have no effect on an xact's behavior
 - Net effect *must be* identical to executing all xacts for **some serial order**.

Schedule of Executing Transactions

- A schedule(调度) is
 - a list of **actions** (READ, WRITE, ABORT, or COMMIT) from a set of xacts,
 - and the **order** in which two **actions** of an xact T appears in a schedule must be the same as the **order** in which they appear in T .
- A **complete** schedule(完全调度) is
 - A schedule that contains either an **abort** or a **commit** for each xact.

A Schedule Involving Two Transactions:

An Incomplete Schedule

$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Serial Schedule

- **Serial schedule(串行调度)**
 - Each xact runs from start to finish, without any intervening actions from other xacts.
- an example : $T1; T2$.

Serializable Schedule-可串行化调度

- Two schedules are **equivalent** if:
 - They involve the same actions of the same xacts,
 - and they leave the DB in the same final state.
- A **serializable** schedule over a set S of xacts is
 - a schedule whose effect on any consistent database instance is guaranteed to be identical to that of **some complete serial schedule** over the set of **committed** xacts in S .

A Serializable Schedule of Two Transactions

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

The result of this schedule is equivalent to the result of the **serial** schedule:
T1; T2.

Important Points of Serializability

$T1; T2.$

- Executing xacts serially in different orders may produce different results,
 $T2; T1.$
 - but all are presumed to be acceptable;
 - the DBMS makes no guarantees about which of them will be the outcome of an interleaved execution.
- Uncommitted xacts can appear in a serializable schedule S , but their effects are cancelled out by UNDO.

Conflicting Actions-冲突操作

- Need an easier check for equivalence of schedules
 - Use notion of “conflicting” actions
 - Anomalies with interleaved execution are simply caused by conflicting actions.
- Two actions are said **conflict** if:
 - They are by different xacts,
 - they are on the **same object**,
 - and at least one of them is a **write**.
 - Three kinds of conflicts: Write-Read (WR) conflict, Read-Write (RW) and Write-Write (WW) conflicts.

Conflicts: Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads-脏读”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), Commit	

- Unrepeatable Reads (RW Conflicts),不可重复的读:

T1:	R(A),	R(A), W(A), Commit
T2:	R(A), W(A), Commit	

Conflicts (Continued)

- Overwriting Uncommitted Data (WW Conflicts):
丢失更新

T1:	W(A),	W(B), Commit
T2:	W(A), W(B), Commit	

Schedules Involving Aborted Xacts

- Serializability relies on **UNDOing aborted xacts** completely, which **may be impossible** in some situations.

An **Unrecoverable** Schedule:

➤ T2 has already committed, and so can not be undone.

<i>T1</i>	<i>T2</i>
<i>R(A)</i>	
<i>W(A)</i>	
	<i>R(A)</i>
	<i>W(A)</i>
	<i>R(B)</i>
	<i>W(B)</i>
	Commit
Abort	

Reading Uncommitted Data (WR Conflicts, “dirty reads-脏读”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), Commit	

Recoverable Schedule-可恢复调度

- In a recoverable schedule, xacts commit only after (and if) all xacts whose changes they read commit.
 - If xacts read only the changes of committed xacts, not only is the schedule recoverable,
 - but also can avoid **cascading aborts(级联中止)**.

Reading Uncommitted Data (WR Conflicts, “dirty reads-脏读”):

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), Commit	

<i>T1</i>	<i>T2</i>
<i>R(A)</i> <i>W(A)</i>	<i>R(A)</i> <i>W(A)</i> <i>R(B)</i> <i>W(B)</i> Commit
Abort	

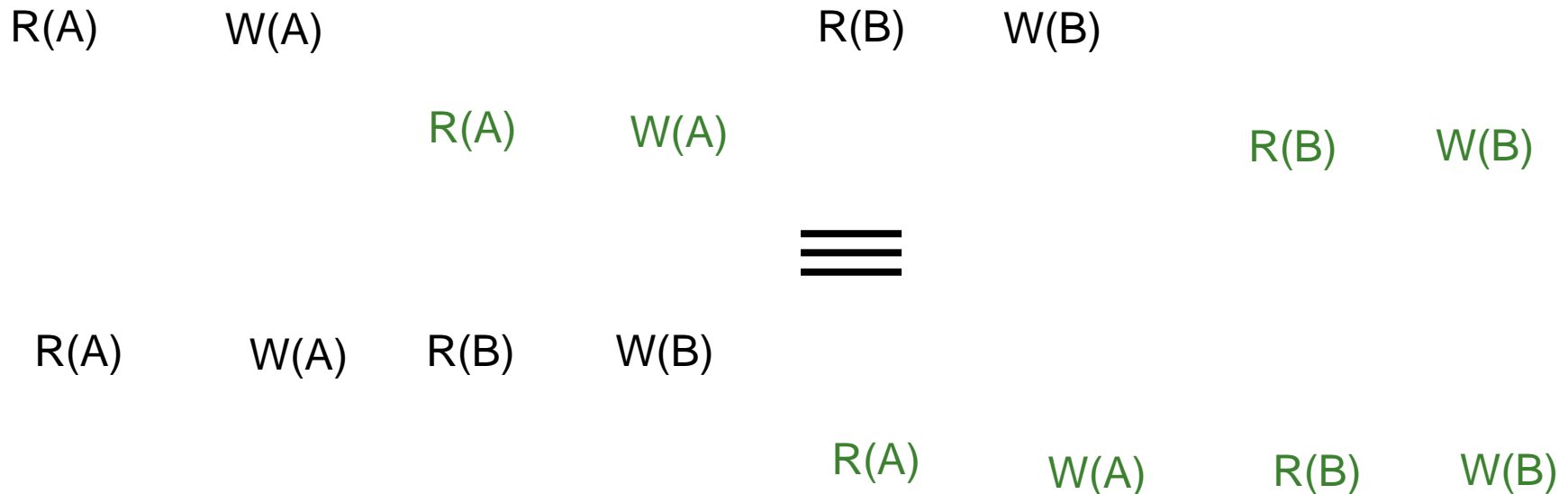
Conflict Serializable Schedules

冲突可串行化调度

- Two schedules are **conflict equivalent**(冲突等价) if:
 - They involve the same set of actions of the same xacts,
 - and they order every pair of conflicting actions of two **committed xacts** in the same way.
- Schedule S is **conflict serializable** if:
 - S is conflict equivalent to some serial schedule.
- Note, some serializable schedules are NOT conflict serializable.
 - A price we pay to achieve efficient enforcement.

Conflict Serializability – Intuition

- A schedule S is conflict serializable if:
 - You can transform S into a serial schedule by swapping **consecutive non-conflicting** operations of different xacts.
- *Example:*



Serializable Schedule That is Not Conflict Serializable

$T1$	$T2$	$T3$
$R(A)$	$W(A)$ Commit	
$W(A)$ Commit		$W(A)$ Commit

➤ This schedule is equivalent to the serial schedule : $T1; T2; T3$.

➤ However it is not conflict equivalent to the serial schedule because the writes of $T1$ and $T2$ are ordered differently.

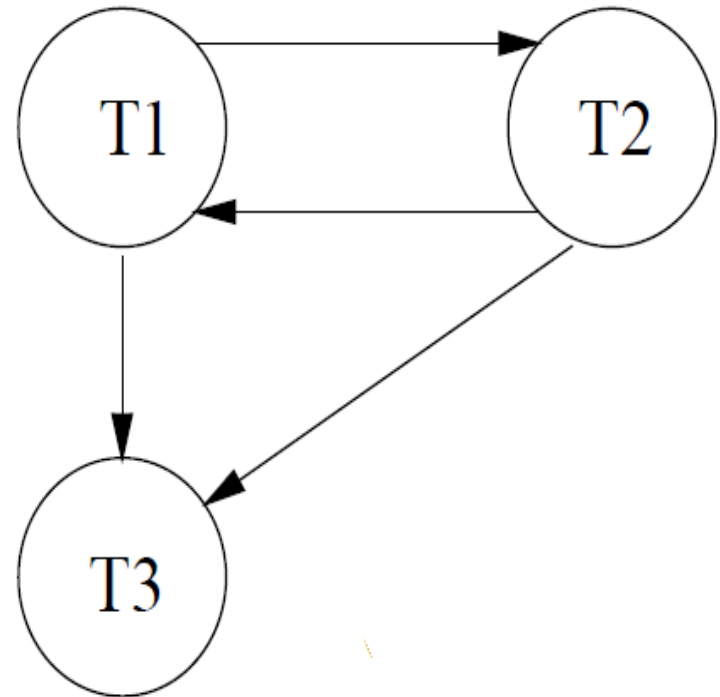
Dependency Graph

依赖图, 优先图

- We use a **dependency graph**, also called a **precedence graph**, to capture all potential conflicts between the xacts in a schedule.
- The **dependency graph** for a schedule S contains:
 - A node for each committed xact
 - An edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions.
- Theorem: Schedule is conflict serializable *if and only if* its dependency graph is **acyclic**. (无环路)

Example of Dependency Graph

$T1$	$T2$	$T3$
$R(A)$	$W(A)$ Commit	
$W(A)$ Commit		$W(A)$ Commit



Two-Phase Locking (2PL)

两阶段加锁

- The most common scheme for enforcing **conflict serializability**.
 - “**Pessimistic**”-悲观的
 - ❑ Sets locks for fear of conflict
 - ❑ The alternative scheme is called **Optimistic** (乐观) **Concurrency Control**.
-

Two-Phase Locking (2PL)

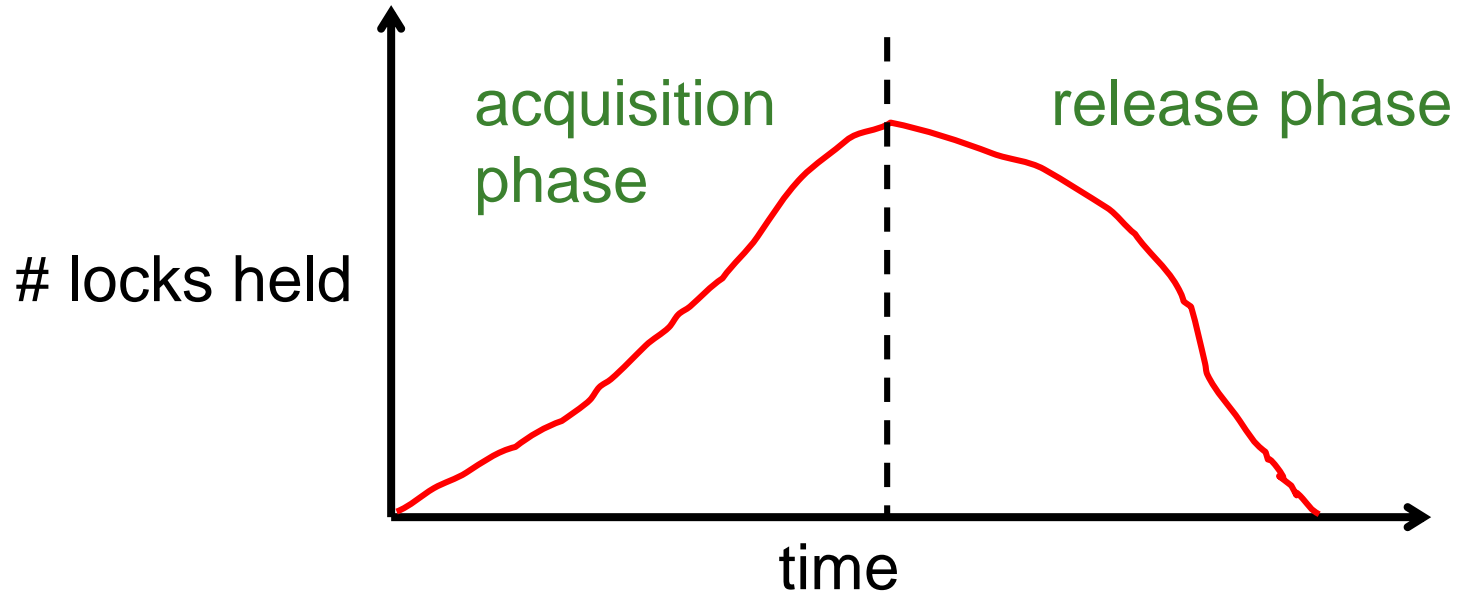
Lock
Compatibility
Matrix

	S	X
S	✓	—
X	—	—

rules:

- An xact must obtain
 - a **S** (*shared*) **lock** before reading, 共享锁
 - and an **X** (*exclusive*) **lock** before writing. 排斥锁
- An xact cannot request additional locks once it releases any lock.

Two-Phase Locking (2PL), cont.



2PL guarantees conflict serializability



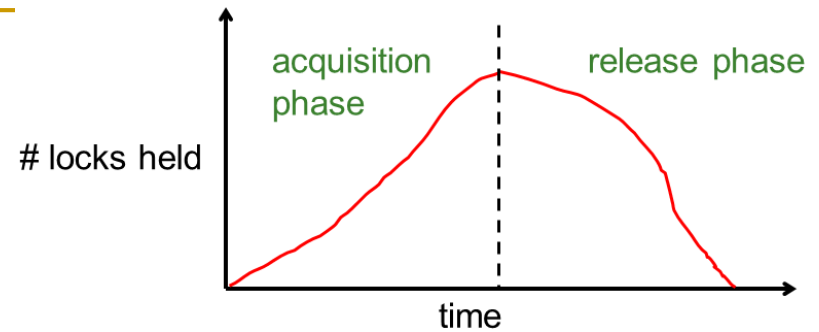
反证?

But, does not prevent **Cascading Aborts**.



Strict 2PL

严格的2PL

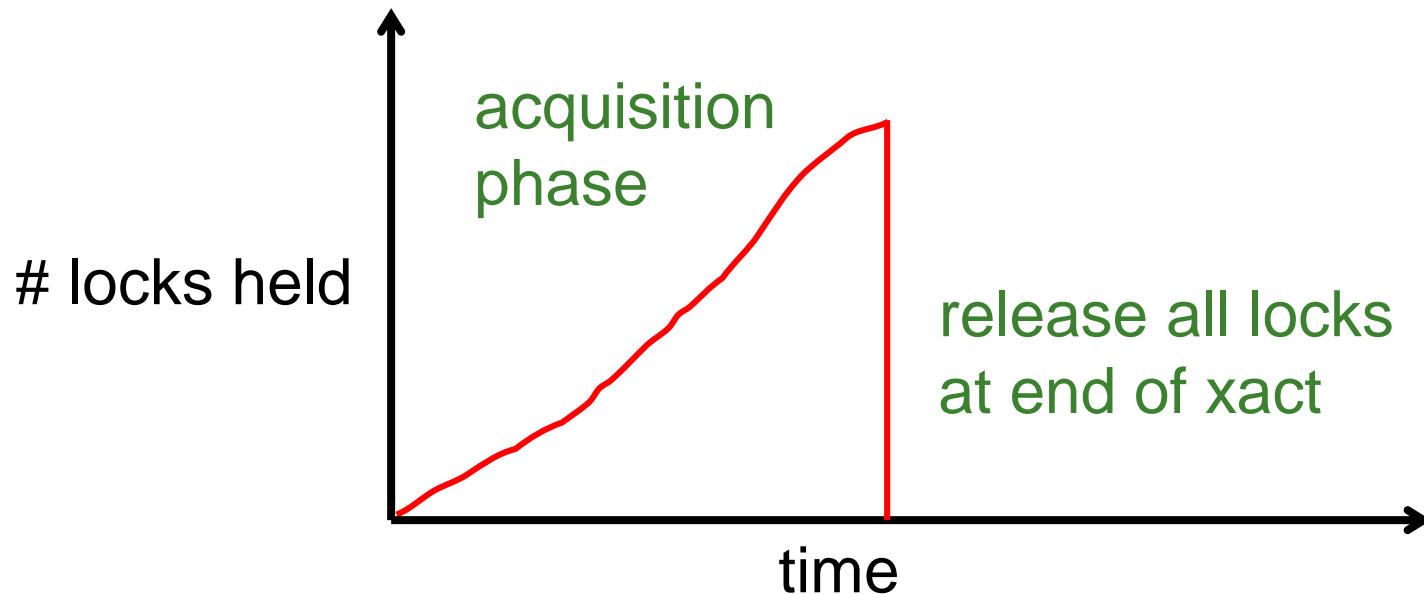


- *Problem:* **Cascading Aborts**
- *Example:* rollback of T1 requires rollback of T2!

T1:	R(A), W(A), R(B), W(B),	Abort
T2:	R(A), W(A)	

- Strict Two-phase Locking (Strict 2PL) protocol:
Same as 2PL, except:
Locks released only when an xact completes
i.e., either:
 - (a) the xact has committed (commit record on disk),
or
 - (b) the xact has aborted and rollback is complete.

Strict 2PL (continued)



T1:	R(A), W(A), R(B), W(B),	Commit/Abort
T2:		R(A), W(A)

xacts read only the changes of committed xacts

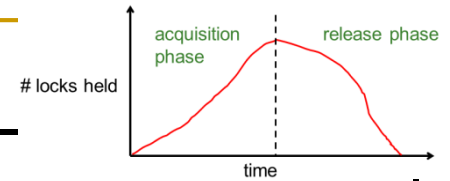
Next ...

- A few examples

Non-2PL, A= 1000, B=2000, Output =?

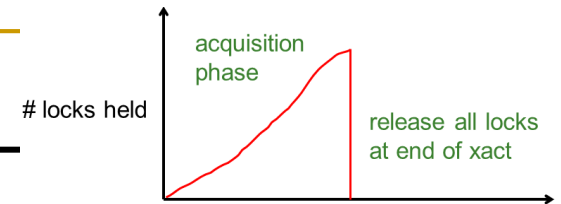
Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	
	Read(A)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B)
	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

2PL, A= 1000, B=2000, Output =?



Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B + 50	
Write(B)	
Unlock(B)	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

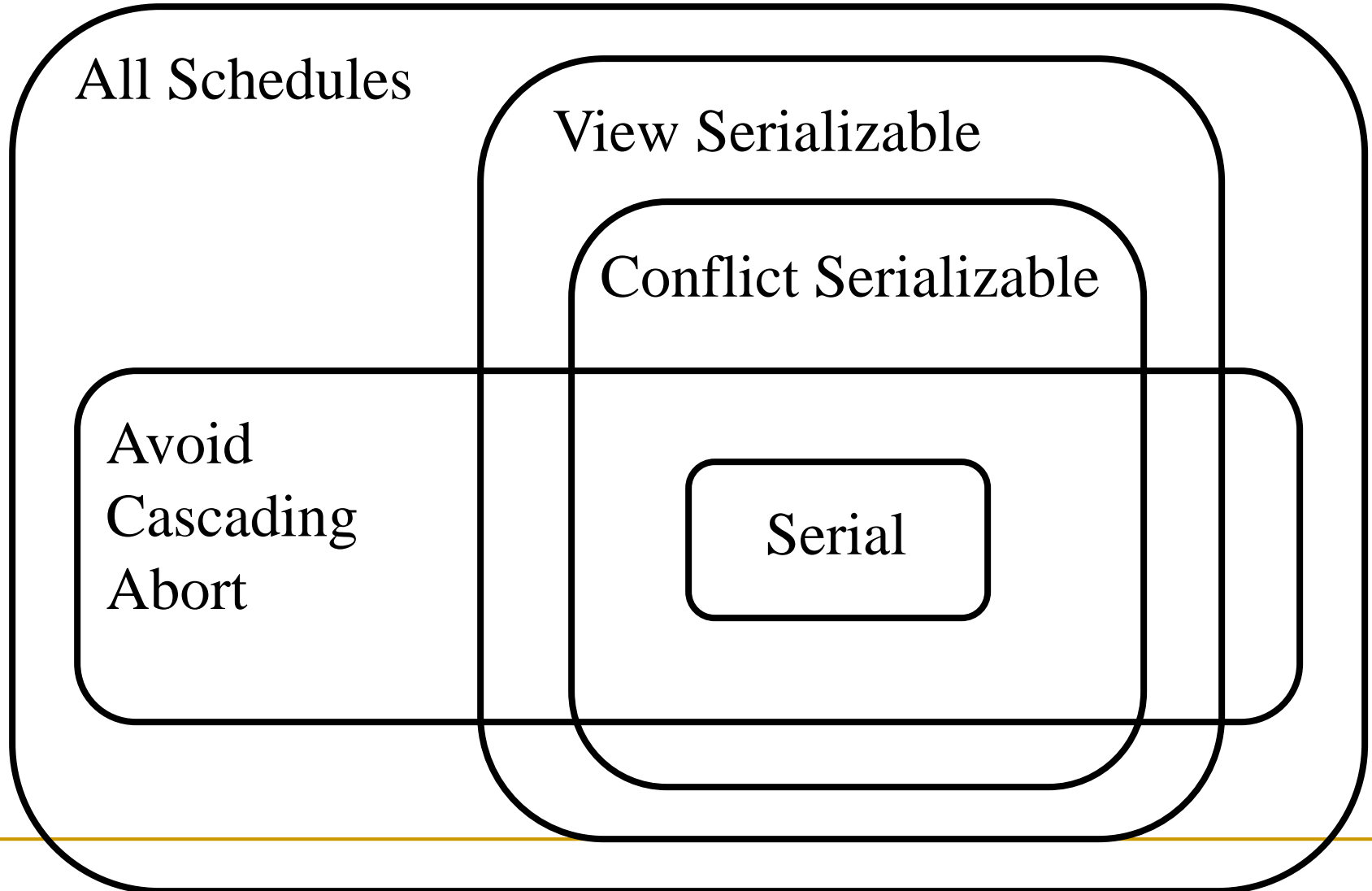
Strict 2PL, A= 1000, B=2000, Output =?



Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

- 编程好处: 只需定义事务即可
- 提高并发度的情况: 无冲突

Venn Diagram for Schedules





Which schedules does Strict 2PL allow?

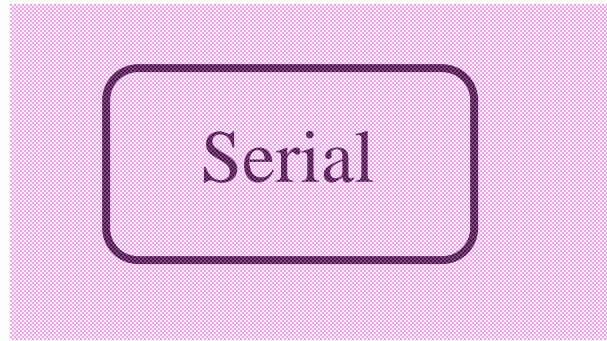
All Schedules

View Serializable

Conflict Serializable

Avoid
Cascading
Abort

Serial



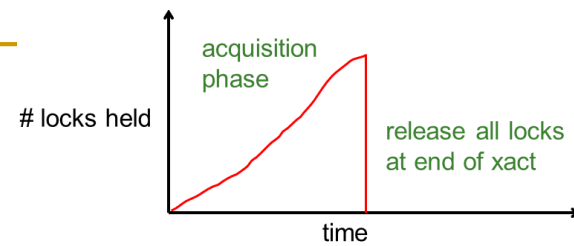
Lock Management 加锁管理

- Lock and unlock requests handled by Lock Manager.
- LM keeps an entry for each currently held lock.
- Entry contains:
 - 某个数据对象标识
 - List of xacts currently holding lock
 - Type of lock held (shared or exclusive)
 - Queue of lock requests

Lock Management (Contd.)

- Entry contains:
 - 某个数据对象标识
 - List of xacts currently holding lock
 - Type of lock held (shared or exclusive)
 - Queue of lock requests
- When lock request arrives:
 - Does any other transaction hold a conflicting lock?
 - If no, grant the lock.
 - If yes, put requestor into wait queue.
- Lock upgrade(锁升级):
 - A transaction with shared lock can request to upgrade to exclusive.

Deadlocks-死锁



- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Ways of dealing with deadlocks:
 - prevention 预防
 - Detection 检测
 - avoidance 避免
- Many systems just punt and use Timeouts
 - What are the dangers with this approach?

Deadlock Prevention 死锁预防

- Common technique in operating systems
- Standard approach: resource ordering 资源排序
 - Screen < Network Card < Printer
- Why is this problematic for Xacts in a DBMS?

Deadlock Detection 死锁检测

- Create and maintain a “waits-for” graph 等待图
- Periodically check for cycles in graph

Deadlock Detection (Continued)

“waits-for” graph 等待图

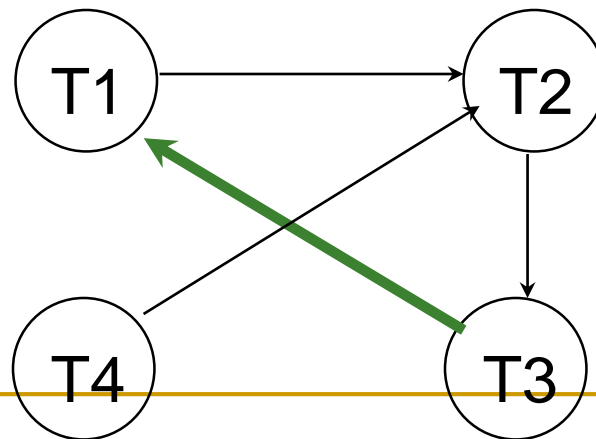
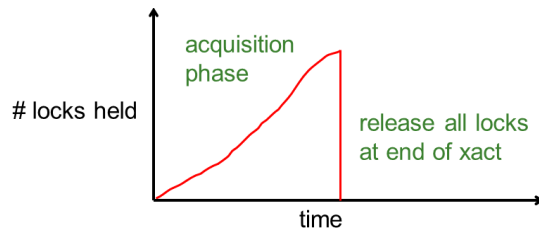
Example:

T1: S(A), S(D), S(B)

T2: X(B), X(C)

T3: S(D), S(C), X(A)

T4: X(B)



be aborted by the DBMS

Deadlock Avoidance 死锁避免

- Assign priorities based on **timestamps**(时间戳).
- Say T_i wants a lock that T_j holds
Two policies are possible:

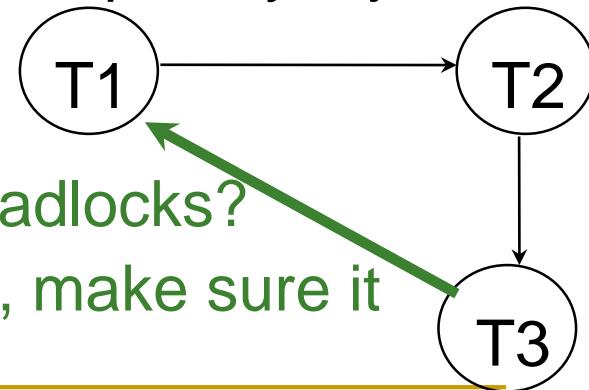
1. **Wait-Die**(等待-死亡): If T_i has higher priority, T_i waits for T_j ; otherwise T_i **aborts**.

高→低

2. **Wound-wait**(伤害-等待): If T_i has higher priority, T_j aborts; otherwise T_i **waits**.

低→高

- Why do these schemes guarantee no deadlocks?
- Important detail: If a transaction re-starts, make sure it gets its original timestamp. -- **Why?**



Locking Granularity(加锁粒度)

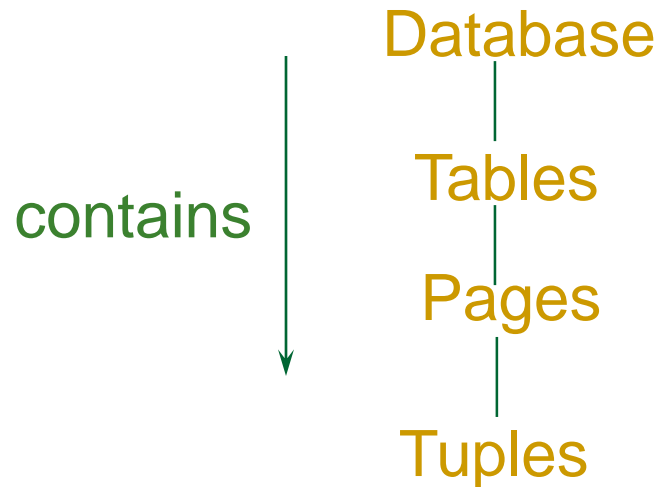
- Entry contains:
 - 某个数据对象标识
 - List of xacts currently holding lock
 - Type of lock held (shared or exclusive)
 - Queue of lock requests

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- *why?* 锁开销和并发度

Multiple-Granularity Locks

多粒度锁

- Shouldn't have to make same decision for all transactions!
- Data “containers” are nested:



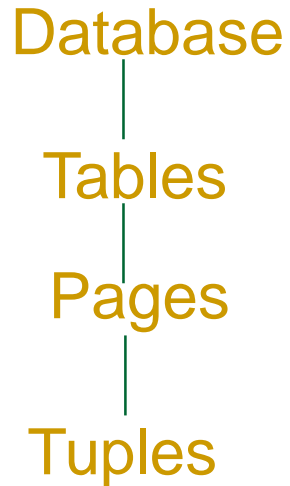
Solution: New Lock Modes, Protocol

- Allow Xacts to lock at each level, but with a special protocol using new “intent” locks:

- ❖ **IS** – Intent to get S lock(s) **意向锁**
at finer granularity. **部分读**
- ❖ **IX** – Intent to get X lock(s)
at finer granularity. **部分写**
- ❖ **SIX mode**: Like S & IX at
the same time. Why useful?

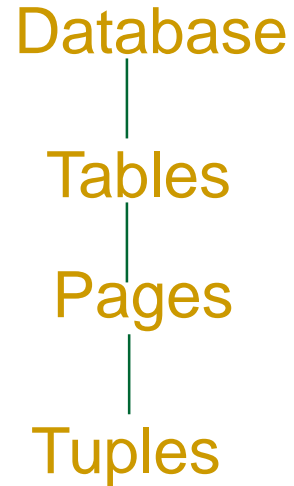
全读 **全写**

- Still need S and X locks, but before locking an item, Xact must have proper intent locks on all its ancestors in the granularity hierarchy.



Multiple Granularity Lock Protocol

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
 - What if Xact holds S on parent? SIX on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.



Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Lock Compatibility Matrix

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-

S 全读
X 全写
IS 部分读
IX 部分写

Database
|
Tables
|
Pages
|
Tuples

- ❖ **IS** – Intent to get S lock(s) at finer granularity.
- ❖ **IX** – Intent to get X lock(s) at finer granularity.
- ❖ **SIX mode**: Like S & IX at the same time.

Examples – 2 level hierarchy

Tables
|
Tuples

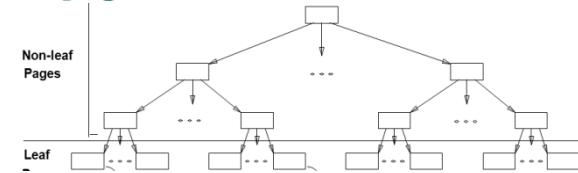
- T1 scans R, and updates a few tuples:
 - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- T2 uses an index to read only part of R:
 - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
 - T3 gets an S lock on R.
 - OR, T3 could behave like T2; can **lock escalation(锁升级)** to decide which.
 - **Lock escalation** dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	✓	✓	✓	✓	
IX	✓	✓	use		
SIX	✓				
S	✓			✓	
X					

Just so you're aware: Optimistic CC

- **Basic idea:** let all transactions run to completion
 - ❑ Make tentative updates on **private copies** of data
 - ❑ At commit time, **check schedules for serializability**
 - ❑ If you can't guarantee it, restart transaction
else "install" updates in DBMS
- Pros & Cons
 - ❑ No waiting or lock overhead in serializable cases
 - ❑ Restarted transactions waste work, slow down others
- OCC a loser to 2PL in traditional DBMSs
 - ❑ Plays a secondary role in some DBMSs
- Generalizations:
 - ❑ Multi-version and Timestamp CC manage the multiple copies in a permanent way

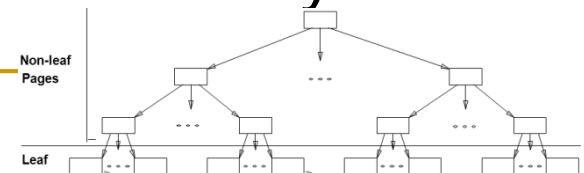
Just So You're Aware: Indexes



- 2PL on B+-tree pages is a rotten idea.
 - Why?
- Instead, do short locks (latches) in a clever way
 - Idea: Upper levels of B+-tree just need to direct traffic correctly. Don't need to be serializably handled!
 - Different tricks to exploit this
- Note: this is pretty complicated!

Just So You're Aware: Phantoms-幻影

- Suppose you query for sailors with rating between 10 and 20, using a B+-tree
 - Tuple-level locks in the Heap File
- I insert a Sailor with rating 12
- You do your query again
 - Yikes! A phantom!
 - Problem: Serializability assumed a static DB!
- What we want: lock the *logical* range 10-20
 - Imagine that lock table!
- What is done: set locks in indexes cleverly



Summary

- Correctness criterion for isolation is “serializability”.
 - In practice, we use “conflict serializability,” which is somewhat more restrictive but easy to enforce.
- Two Phase Locking and Strict 2PL: Locks implement the notions of conflict directly.
 - The lock manager keeps track of the locks issued.
 - **Deadlocks** may arise; can either be prevented or detected.
- Multi-Granularity Locking:
 - Allows flexible tradeoff between lock “scope” in DB, and locking overhead in RAM and CPU
- More to the story
 - Optimistic/Multi-version/Timestamp CC
 - Index “latching”, phantoms

Summary

■ 要求:

- 理解串行调度、可串行化调度、冲突可串行化调度、可恢复调度、避免级联中止、严格调度的概念
 - 有脏读X-R（但后提交） 无脏读X-R 无脏读/写 X-R/X
- 能够根据给定的一个调度，绘制其优先图，从而判断是否冲突可串行化调度？
- 理解2PL、严格2PL协议，能够绘制等待图并判断是否存在死锁？