Transaction Management

Example: transfer CHF 50 from A to B

1. Read balance of A from DB into Variable *a*: **read**(*A,a*);
2. Subtract 50.- CHF from the balance: *a*:= *a* – 50;
3. Write new balance back into DB: **write**(*A,a*);
4. Read balance of B from DB into Variable *b*: **read**(*B,b*);
5. Add 50,- CHF to balance: *b* := *b* + 50;
6. Write new balance back into DB: **write**(*B, b*);

N.B.: Actually, banks do not do this in one TA! ☹

Properties of Transactions: ACID

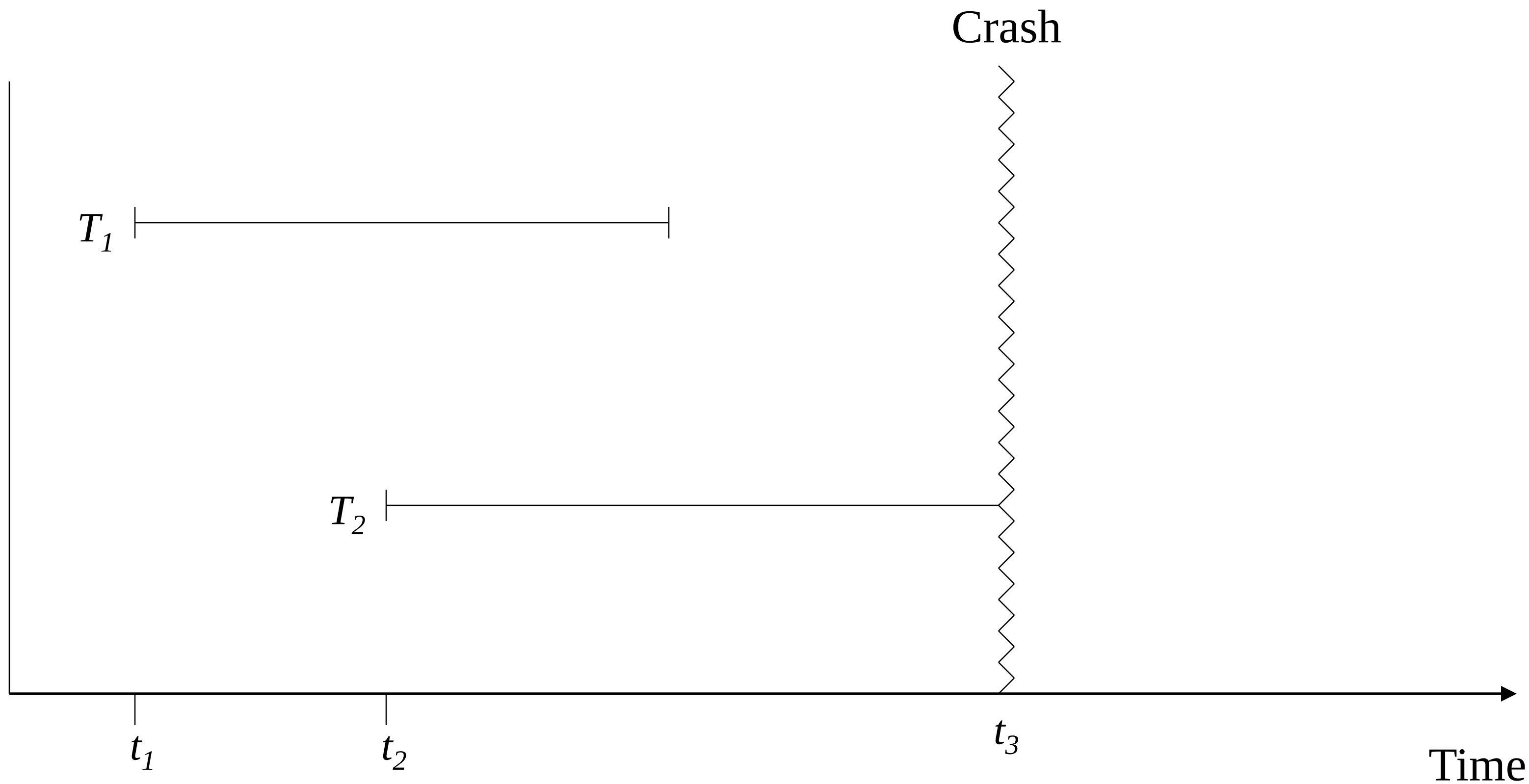
- **A**tomicity
  - All or nothing
  - Undo changes if there is a problem
- **C**onsistency
  - If DB consistent before a TA, DB consistent after TA
  - Check integrity constraints at the end of a TA
- **I**solation
  - TA is executed as if there were no other TA
  - Synchronize operations of concurrent TAs
- **D**urability
  - Updates of a completed TA must never be lost
  - Redo changes if there is a problem

Getting Married(Mr. X, Ms. Y)

1. Mr. X, do you want to marry Miss Y?
2. if (no) then abort()
3. *write(X.spouse, Y)*
4. Miss Y, do you want to marry Mr. X?
5. if (no) then abort()
6. *write(Y.spouse, X)*
7. Does anybody object?
8. if (yes) then abort()
9. *commit()*

N.B.: This is how it really works! ☺

Properties of a Transaction (A & D)



Types of Failures: R1-R4 Recovery

1. Abort of a single TA (application, system)
   - *R1* Recovery: Undo a single TA

1. System crash: lose main memory, keep disk
   - *R2* Recovery: Redo committed TAs
   - *R3* Recovery: Undo active TAs

1. System crash with loss of disks
   - *R4* Recovery: Read backup of DB from tape

Programming with Transactions

- **begin of transaction (BOT)**: Starts a new TA
- **commit**: End a TA (success).
  - Application wants to make all changes durable.

- **abort**: End a TA (failure).
  - Application wants to undo all changes.

- N.B. Many APIs (e.g., JDBC) have an auto-commit option:
  - Every SQL statement run in its own TA.

SQL Example

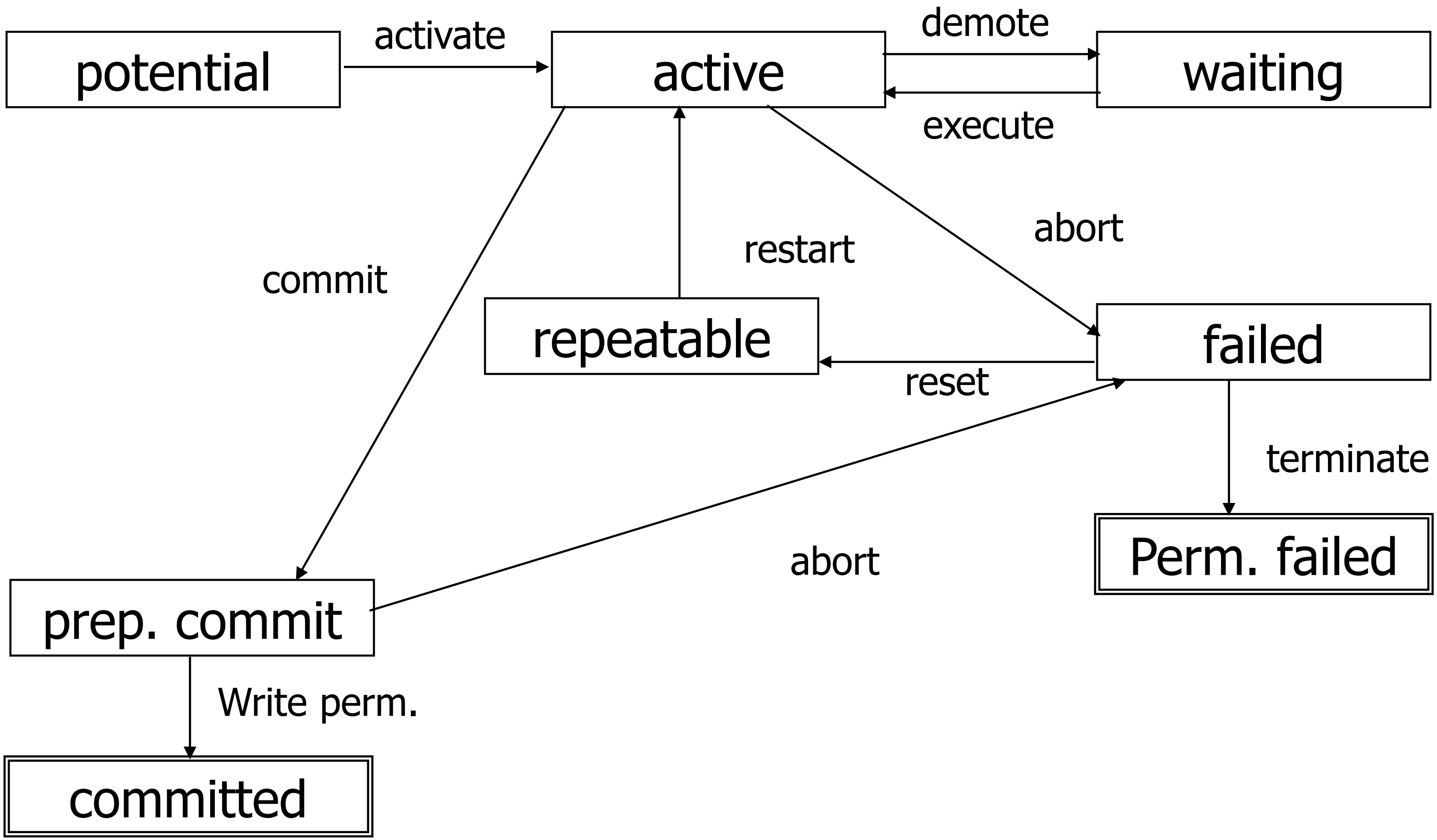
```
insert into Lecture
        values (5275, `Kernphysik`, 3, 2141);
insert into Professor
        values (2141, `Meitner`, `FP`, 205);
commit
```

Advanced TA Features

- **define savepoint:** Establish a recoverable intermediate state
  - Attractive for long-running TAs; protect against crashes
  - Does not imply commit or abort!!!

- **backup transaction**: Reset state of TA/DB to savepoint.
  - Undo all changes after savepoint.
  - Redo all changes before savepoint.
  - Stay within the same TA context.

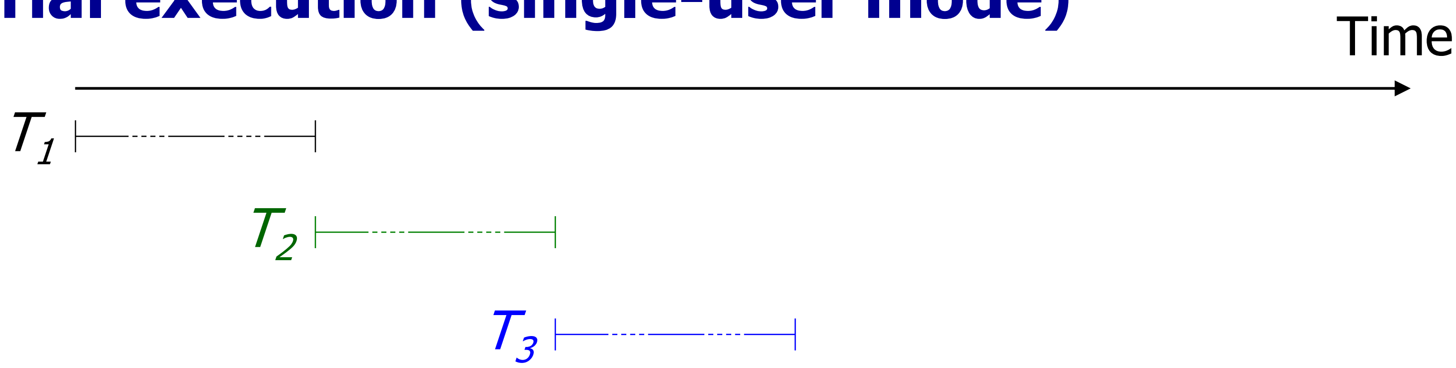
State-transitions of TAs



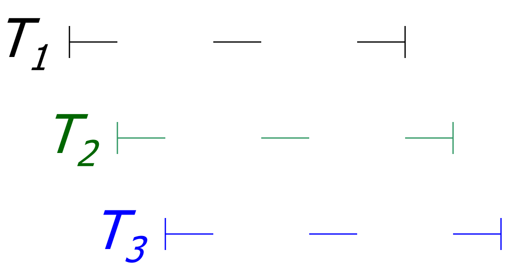
Concurrent Transactions

Alternative ways to execute T_1, T_2 und T_3:

(a) Serial execution (single-user mode)



(b) Concurrent execution (multi-user mode)



Trade-off between *correctness* and *low latency*!

Lost Update

Step	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.		read(A,a_2)
4.		$a_2 := a_2 * 1.03$
5.		write(A,a_2)
6.	write(A,a_1)	
7.	read(B,b_1)	
8.	$b_1 := b_1 + 300$	
9.	write(B,b_1)	

Uncommitted Read

Step	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.	write(A,a_1)	
4.		read(A,a_2)
5.		$a_2 := a_2 * 1.03$
6.		write(A,a_2)
7.		commit
8.	read(B,b_1)	
9.	abort	

Phantom

T_1	T_2
	select sum(balance)
	from Account
insert into Account	
values (*C*,1000,...)	
	select sum(balance)
	from Account

Serializability

- Concurrent history is equivalent to a serial history!
  - (need to define *equivalence* of *histories*)
- The following history is serializable (i.e., correct):

Step	T_1	T_2
1.	**BOT**	
2.	read(*A*)	
3.		**BOT**
4.		read(*C*)
5.	write(*A*)	
6.		write(*C*)
7.	read(*B*)	
8.	write(*B*)	
9.	**commit**	
10.		read(*A*)
11.		write(*A*)
12.		**commit**

Defintion: Transaction

A TA (T_i) is defined as a sequence of operations:

- (BOT implicit – not considered here)
- $r_i(A)$: T_i reads Object *A*
- $w_i(A)$: T_i writes Object *A*
- a_i : T_i aborts
- c_i : T_i commits

A TA defines a total order (<) on all its operations.

Defintion: Transaction (ctd.)

- TA has either an **abort** or a **commit**; never both!
- No operations after an **abort**, if T_i aborts
  - for all operations (except a_i): $p_i(A) <_i a_i$.
- No operations after a **commit**, if T_i commits
  - for all operations (except c_i): $p_i(A) <_i c_i$

Definition of History (H)

- $H = \bigcup_{i=1}^{n} T_i$

- $<_H$ is a partial order that is consistent with $<_i$:

$$<_H \supseteq \bigcup_{\mathrm{i}=1}^{\mathrm{n}} <_{\mathrm{i}}$$

- $<_H$ orders operations which are in conflict

That is, for all $p,q \in H$: $p <_H q$ *OR* $q <_H p$

Def.: Conflicts of Reads and Writes

For i <> j

- $r_i(A)$ and $r_j(B)$: no conflict.
- $r_i(A)$ and $w_j(B)$: conflict iff A = B.
- $w_i(A)$ and $r_j(B)$: conflict iff A = B.
- $w_i(A)$ and $w_j(B)$: conflict iff A = B.

(Within the same TA all opertations are in conflict!)

- Lemma: Two histories are equivalent if they execute all pairs of conflicting operations in the same order.
- Does this lemma define a „notwendig“ or „hinreichend“ crit.?

Def.: Conflicts of Aborts, Commits

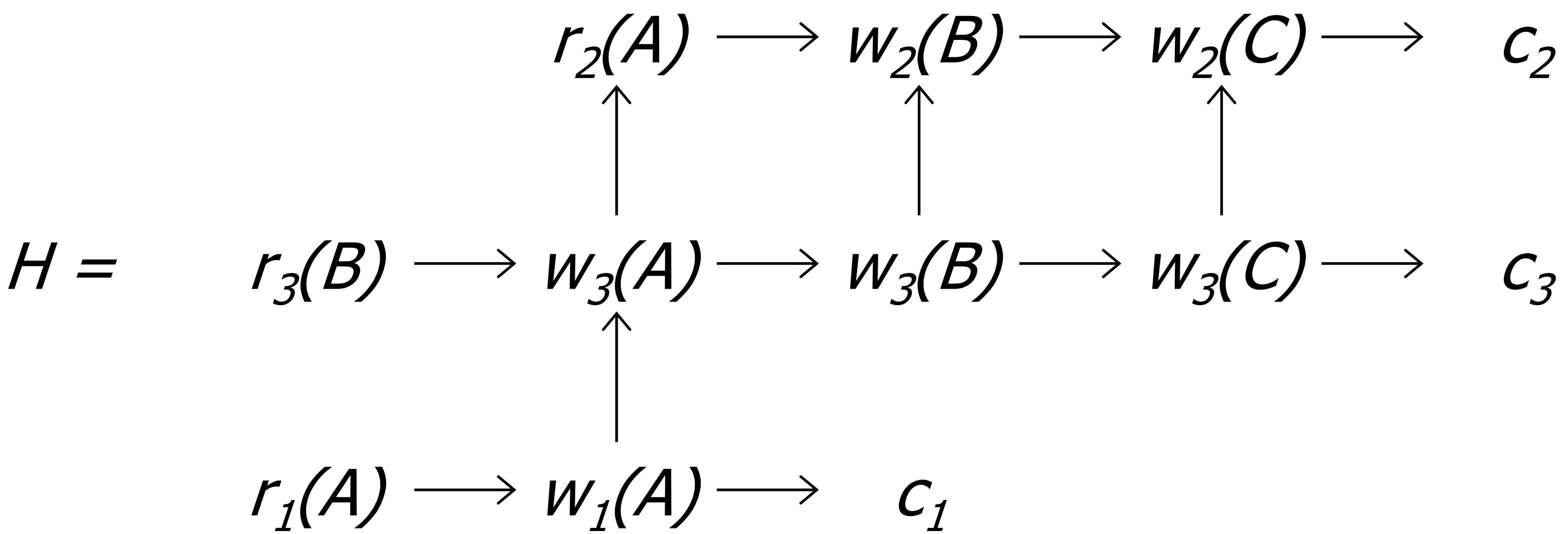
Abort

- $r_i(A)$ and a_j: Conflict if T_j updated Object A.
- $w_i(A)$ and a_j: Conflict if T_j updated Object A.
  - N.B. Reads of T_j are irrelevant.

Commit

- $r_i(A)$ and c_j: no conflict
- $w_i(A)$ und c_j: no conflict

History of three TAs



Definition: Serial History

A Serial History defines a total order on all Transactions:

- TA1 < TA2 iff all operations o1 of TA1, o2 of TA2

  o1 < o2

(N.B. A Serial History defines a total order on all operations.)

Serial Execution: T_1 / T_2

Step	T_1	T_2
1.	**BOT**	
2.	read(*A*)	
3.	write(*A*)	
4.	read(*B*)	
5.	write(*B*)	
6.	**commit**	
7.		**BOT**
8.		read(*C*)
9.		write(*C*)
10.		read(*A*)
11.		write(*A*)
12.		**commit**

Definition: Equivalence of Histories

- Two histories are equivalent
  - All reads (of committed TAs) return the same result.
  - At the end, the state of the DB is the same

- Corner Case:

$R_1(x)\ W_2(x)\ R_1(x)\ A_1\ C_2$

is equivalent to

$R_1(x)\ R_1(x)\ A_1\ W_2(x)\ C_2$

Criterion for Equivalent Histories

- $H \equiv H'$ if all conflict operations are executed in same order. (Exercise: Proof for this Criterion.)

$r_1(A) \rightarrow r_2(C) \rightarrow w_1(A) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

Defintion: Serializable History

A History is Serializable iff it is equivalent to a serial history.

- There are many serial histories. Okay to be equivalent to 1.
- n! complexity to test for serializability with n concurrent TAs
  - How can you do that more efficiently?
  - How do you test whether a DBMS only generates serializable histories?

Non-serializable History

Step	T_1	T_3
1.	**BOT**	
2.	read(*A*)	
3.	write(*A*)	
4.		**BOT**
5.		read(*A*)
6.		write(*A*)
7.		read(*B*)
8.		write(*B*)
9.		**commit**
10.	read(*B*)	
11.	write(*B*)	
12.	**commit**	

Is this history serializable?

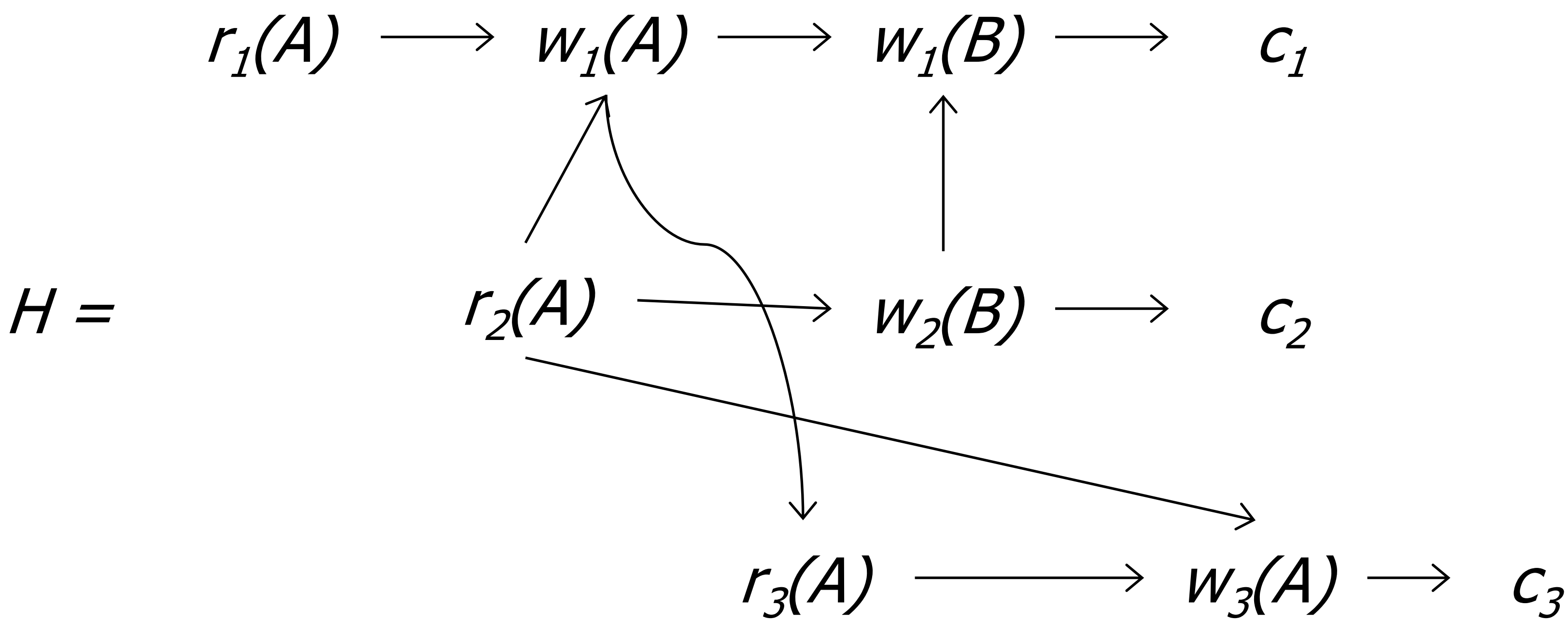
Step	T_1	T_3
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 - 100$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 + 100$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Is this history serializable?

Step	T_1	T_3
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 * 1.03$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 * 1.03$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

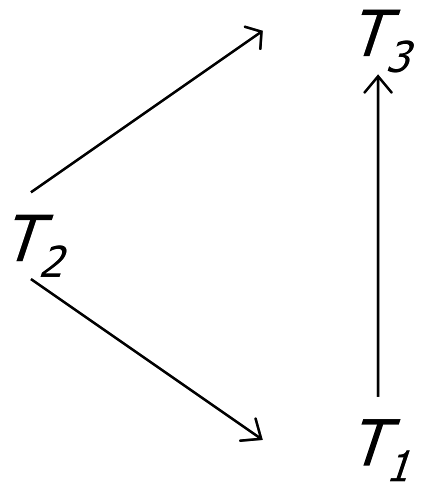
Serializable History

Is the following history serializable? If yes, what is the serial hist.?



Serializability Graph

$SG(H) =$



- $w_1(A) \rightarrow r_3(A)$ in H implies $T_1 \rightarrow T_3$ in SG(H)
- Compact representation of the dependencies in a history.

Serializability Theorem (Proof?)

A history is serializable iff its serializability graph is acyclic.

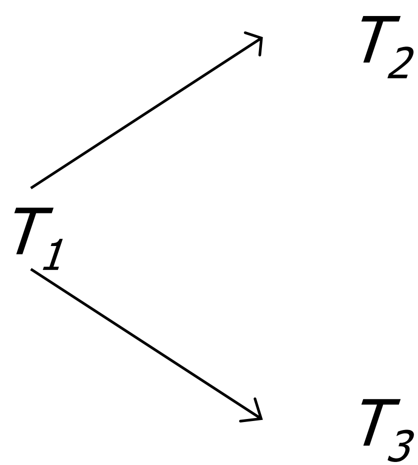
History

H =

$$w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$$

Serializability Graph

SG(H)=



Topological Sorting

$$H_s^1 = T_1 \mid T_2 \mid T_3$$

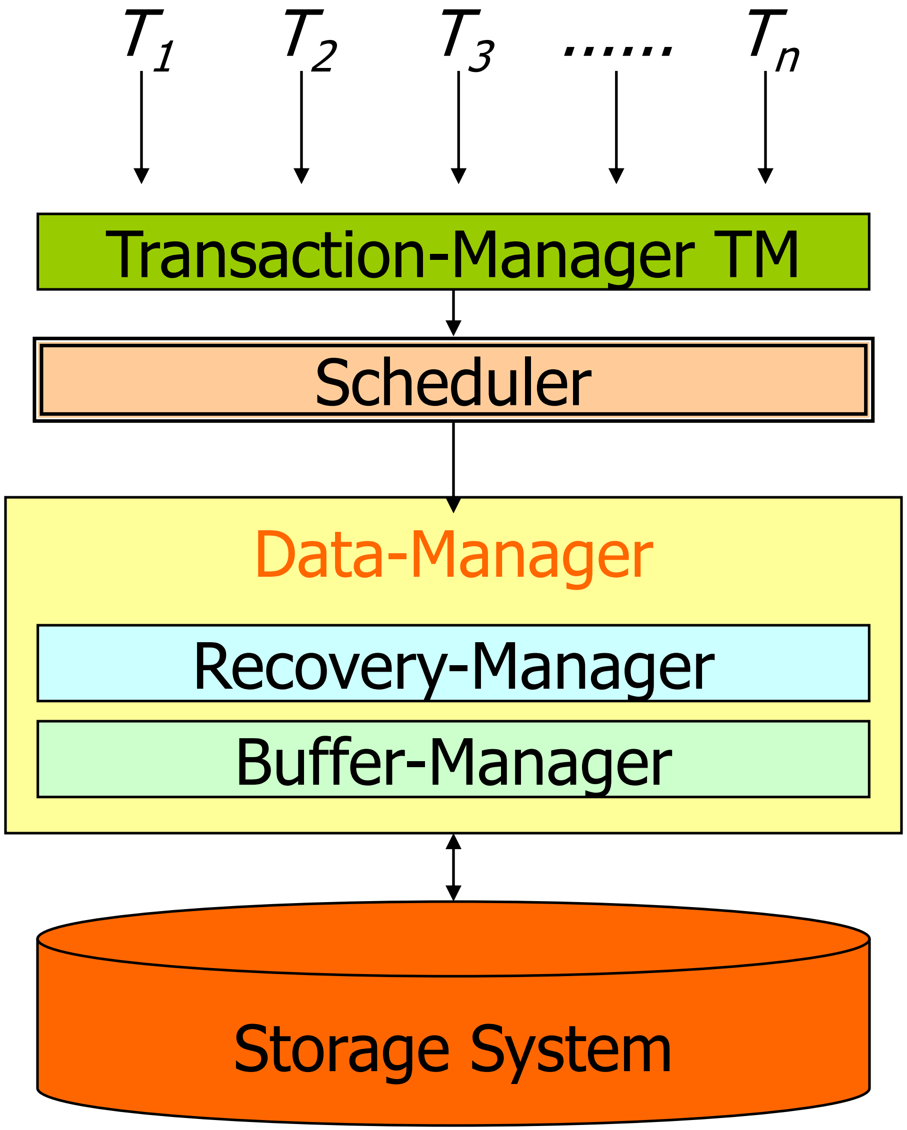
$$H_s^2 = T_1 \mid T_3 \mid T_2$$

$$H \circ H_s^1 \circ H_s^2$$

Time in Databases

- Networks vs. Database Systems (DBMS)
  - Networks bridge space
  - Database systems bridge time
- A DBMS orders operations (and TAs)
  - Databases do NOT define time intervals (seconds, min., ...)
  - But, order determines *visibility* and *recoverability* of updates
- Distinguish between transaction time and app time
  - Bi-temporal: Order for 2010 may be entered in 2009

Database-Scheduler



Pessimistic Synchronization

- Basic Idea: Control Visibility by blocking TAs
- Locking
  - *S* (shared, read lock): needed for read operations
  - *X* (exclusive, write lock): needed for write operations
- Compatibility Matrix
  - decide when to grant lock vs. block TA
  - many sophisticated variants: trade concurrency vs. overhead

		NL	*S*	*X*
request	*S*	✓	✓	-
request	*X*	✓	-	-

Lock Modes: OS vs. DB

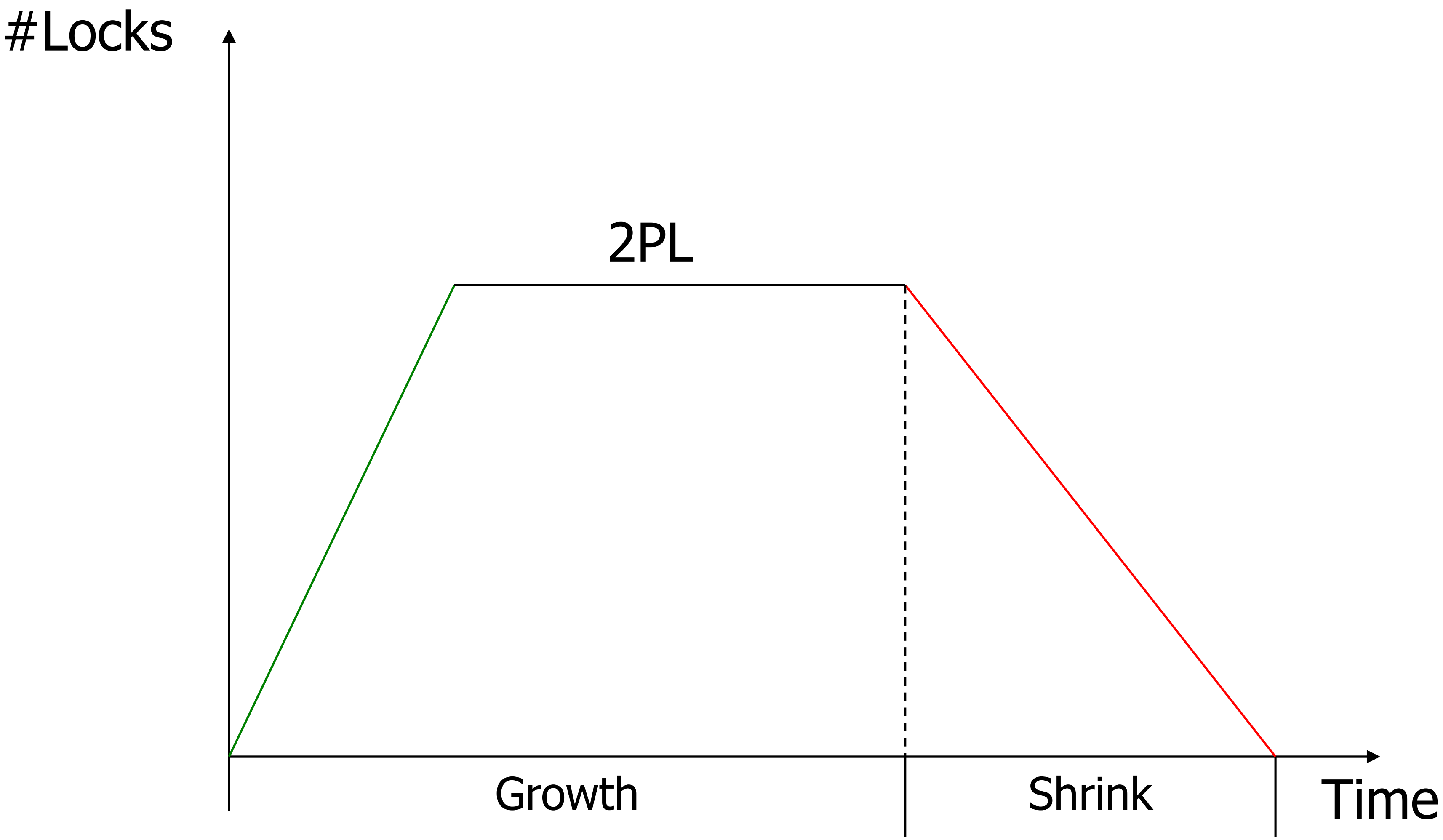
- Why use multiple kinds of locks?
  - increases concurrency: e.g., two concurrent reads
- Why can´t OS play the same tricks?
  - DB knows semantics of operations –> compatability
  - OS must make „worst case" assumptions; ops are black box
  - (similar optimization as buffer management in DBMS)
- Practice
  - Many more lock modes in real systems
  - Many further optimizations and tricks possible
  - (see Information Systems Class for more)

Two-phase Locking Protocol (2PL)

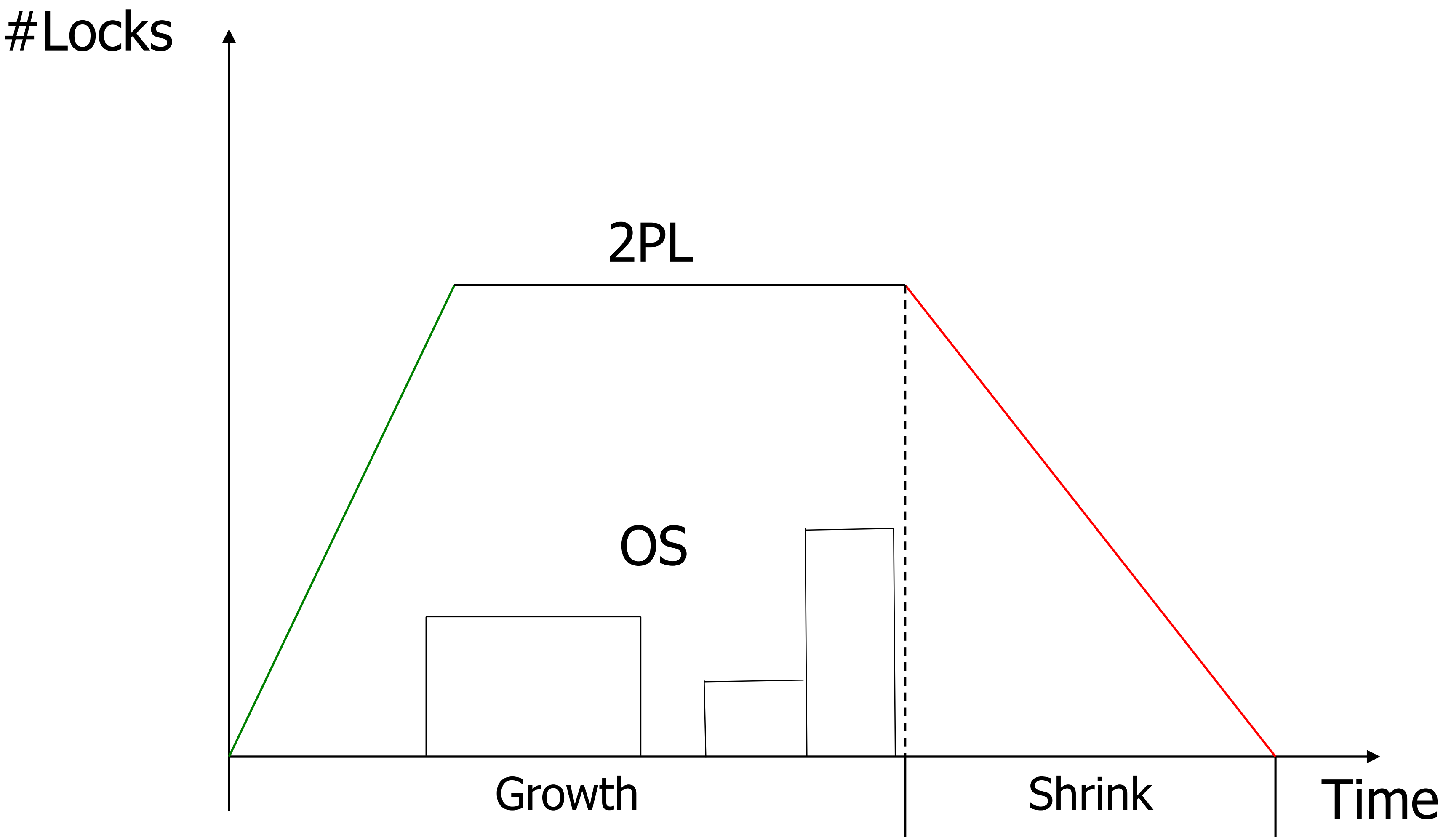
1. Before accessing an object, a TA must acquire lock.
2. A TA acquires a lock only once. Lock upgrades are possible.
3. A TA is blocked if the lock request cannot be granted according to the compatibility matrix.
4. **A TA goes through two phases:**
   - ***Growth*: Acquire locks, but never release a lock.**
   - ***Shrink*: Release locks, but never acquire a lock.**
5. At EOT (commit or abort) all locks must be released.

- N.B.: 2PL also relevant if you have only X locks. Why?

Two-phase Locking



Two-phase Locking



Synchronization of TAs using 2PL

- T_1 modifies Objects *A* and *B* (e.g., money transfer)
- T_2 reads Objects *A* and *B*

Synchronization of TAs using 2PL

Step	T_1	T_2	Comment
1.	**BOT**		
2.	**lockX**(*A*)		
3.	read(*A*)		
4.	write(*A*)		
5.		**BOT**	
6.		**lockS**(*A*)	T_2 is blocked
7.	**lockX**(*B*)		
8.	read(*B*)		
9.	**unlockX**(*A*)		T_2 is reactivated
10.		read(*A*)	
11.		**lockS**(*B*)	T_2 is blocked
12.	write(*B*)		
13.	**unlockX**(*B*)		T_2 is reactivated
14.		read(*B*)	
15.	**commit**		
16.		**unlockS**(*A*)	
17.		**unlockS**(*B*)	
18.		**commit**	

Violation of 2PL: Non-ser. History

Step	T_1	T_3
1.	**lockX(A)**	
2.	write(A)	
3.	**unlockX(A)**	
4.		**lockX(A)**
5.		write(A)
6.		**lockX(B)**
7.		write(B)
8.		**unlockX(A,B)**
9.		**commit**
10.	**lockX(B)**	
11.	write(*B*)	
12.	**commit**	

2PL and Phantoms: How does that work?

T_1	T_2
	select sum(balance)
	from Account
insert into Account	
values (*C*,1000,...)	
	select sum(balance)
	from Account

Need a lock on „*Account*". Typically done with index, but tricky!

Correctness of 2PL (Proof Sketch)

- Let H be a history generated by 2PL
- Assume that H is not serializable
- Serializability Graph of H must have a cycle (Ser. Theorem)
- Wlog, assume that the cycle has length 2 with T1 and T2
  - (proof generalizes to any number of transactions in cycle)
- There must exist operations o1, o1′ in T1 and o2, o2′ in T2:
  - conflict(o1, o2)
  - conflict(o1′, o2′)
  - o1 < o2 in H
  - o2′ < o1′ in H
- Wlog, assume o2 < o2′ in H
  - o1 < o2 < o2′ < o1′
- Contradicts 2PL
  - T1 **releases lock** for o1 **before getting lock** for o1′ (qed)

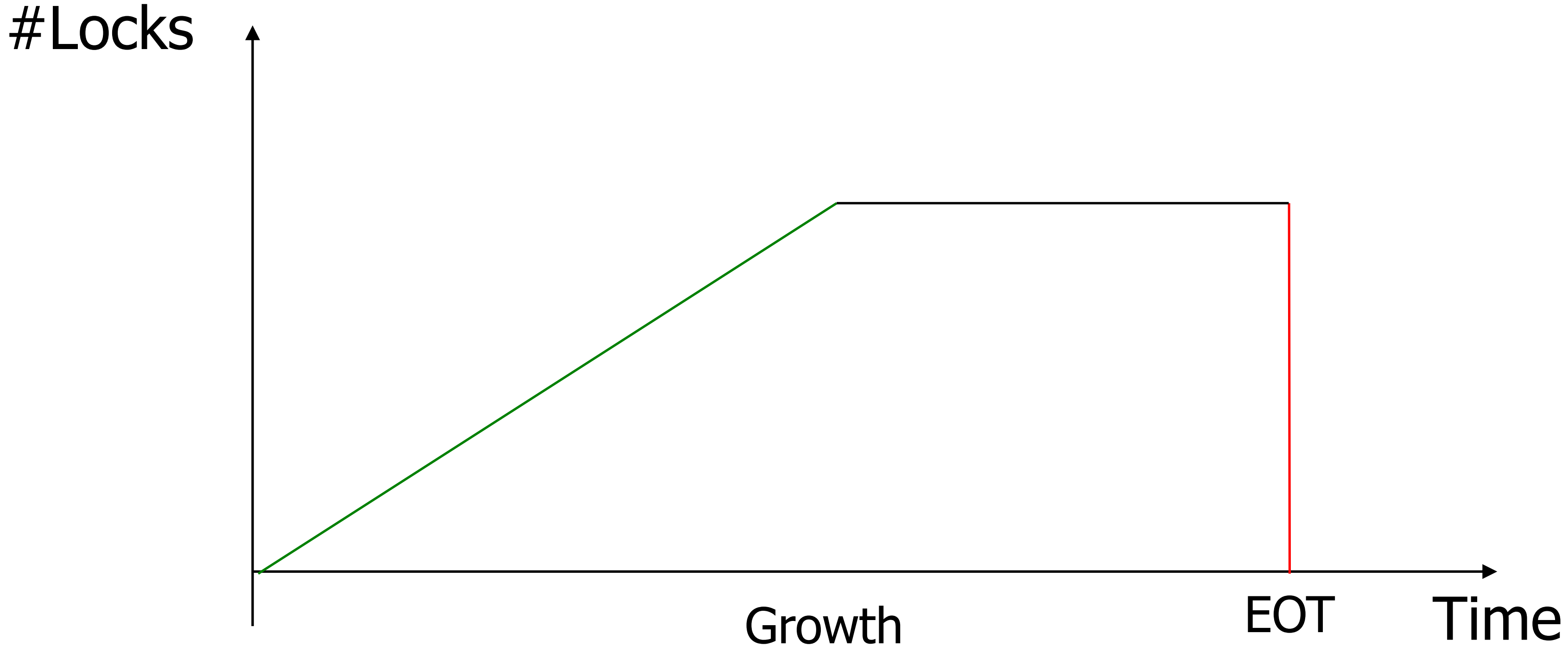
Does 2PL prevent this phenomenon?

Step	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.	write(A,a_1)	
4.		
5.		read(A,a_2)
6.		$a_2 := a_2 * 1.03$
7.		write(A,a_2)
8.	. . .	. . .
9.	abort	

Abort of T1 triggers abort of T2. Possible domino effect.

Strict 2PL

- All locks are kept until EOT (commit or abort)



Discussion: Strict 2PL

- Avoid cascading aborts
  - Deal with uncommitted read problems (Phenomenon 2)
  - Avoids implicit violation of 2PL: implicit lockX for abort

- Important: Avoids rollback of committed TAs
  - Basic 2PL does not implement ACID properly

- Couples visibility with recoverability
  - Recoverabilty at commit: Definition of A and D
  - Visibility at commit: Artifact of strict 2PL
  - Important: differentiate between these two concepts

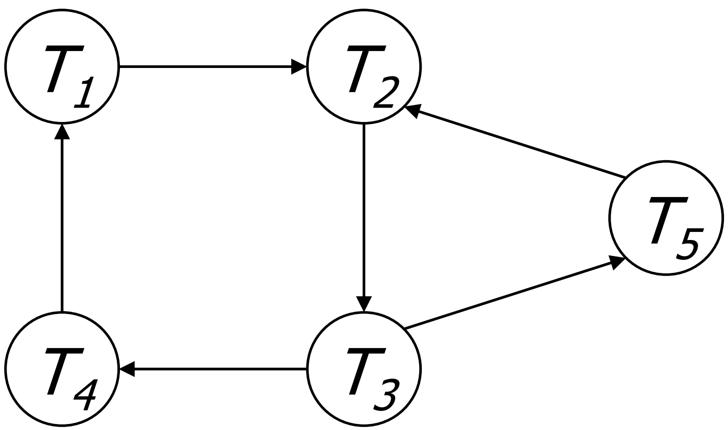
Deadlocks

Step	T_1	T_2	Comment
1.	**BOT**		
2.	**lockX**(*A*)		
3.		**BOT**	
4.		**lockS**(*B*)	
5.		read(*B*)	
6.	read(*A*)		
7.	write(*A*)		
8.	**lockX**(*B*)		T_1 must wait for T_2
9.		**lockS**(*A*)	T_2 must wait for T_1
10.	...	...	$\Rightarrow$ *Deadlock*

Deadlock Detection

Wait-for Graph

- $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$
- $T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$



- Abort T_3 will resolve both cycles
- Alternative: Deadlock detection with timeouts. Pros/cons?

2PL: OS vs. DB

- Both use locking to protect resources
  - OS: printers, critical paths (code)
  - DB: objects (data)
- Difference: individual vs. collection of resources
  - OS: individual resource
  - DB: collection of resources with integrity constraints
- Both assign locks to „sequence of operations"
  - OS: process
  - DB: transaction
- Difference: duration of keeping locks
  - OS: keep lock as long as resource is used
  - DB: keep lock beyond usage – end of transaction
- (Distributed System add another dimension:
  - Maintain and synchronize copies of the same resource)

Snapshot Isolation

- When a TA starts it receives a timestamp, T.
- All reads are carried out *as of* the DB version of T.
  - Need to keep historic versions of all objects!!!
- All writes are carried out in a separate buffer.
  - Writes only become visible after a commit.
- When TA commits, DBMS checks for conflicts
  - Abort TA1 with timestamp T1 if exists TA2 such that
    - TA2 committed after T1 and before TA1
    - TA1 and TA2 updated the same object
- **Basic idea the same as for SVN!**

- Does Snapshot Isolation give you serializability? [Berenson+95]
- What are the advantages/disadv. of Snapshot Isolation?

SI and Lost Update

Step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.		**BOT**
4.		read(A)
5.		write(A)
6.		**commit**
7.	write(A)	
8.	**commit**	

SI and Lost Update (ctd.)

Step	T_1	T_2
1.		**BOT**
2.		read(A)
3.	**BOT**	
4.	read(A)	
5.		write(A)
6.		**commit**
7.	write(A)	
8.	**commit**	

SI and Lost Update (ctd.)

Step	T_1	T_2
1.		**BOT**
2.		read(A)
3.		write(A)
4.	**BOT**	
5.	read(A)	
6.		**commit**
7.	write(A)	
8.	**commit**	

SI reorders R1(A) and W2(A) -> not seriliz. -> abort of T1

SI and Uncommitted Read

Step	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.		**BOT**
5.		read(A)
6.		write(A)
7.	read(B)	...
8.	**abort**	

SI and Phantoms: How does that work?

T_1	T_2
	select sum(balance)
	from Account
insert into Account	
values (*C*,1000,...)	
	select sum(balance)
	from Account

„Sandbox“ also involves „set of accounts“! Works nicely!

Discussion of Snapshot Isolation

- Concurrency and Availability
  - No read or write of a TA is ever blocked
  - (Blocking only happens when a TA commits.)
- Performance, Overhead:
  - Need to keep write-set of a TA only
  - Very efficient way to implement aborts
  - Often keeping all versions of an object useful anyway
  - No deadlocks, but unnecessary rollbacks
  - Implicitly deals with phantoms (complicated with 2PL)
- Correctness (Serializability): Problem „Write Skew“
  - Checking integrity constraint also happens in the snapshot
  - Two concurrent TAs update different objects
  - Each update okay, but combination not okay
  - Example: Both doctors sign out…

Example: One doctor on duty!

Step	T_1	T_2	Comment
1.	**BOT**		(A, duty); (B, duty)
2.	write(A, free)		
3.		**BOT**	
4.		write(B, free)	
5.	check-constraint		Okay: (B, duty)
6.		check-constraint	Okay: (A, duty)
7.	**commit**		
8.		**commit**	
9.			**Constraint violated!!!**

N.B. Example can be solved if check part of DB commit. Impossible to solve at the app level.

Interesting History

Step	T_1	T_2	T_3
1	**BOT**		
2		**BOT**	
3		write(B)	
4		write(C)	
5		**commit**	
6	read(B)		
7			**BOT**
8			read(A)
9			read(C)
10			**commit**
11	write(A)		
12	**commit**		

Interesting History: Discussion

- 2PL
  - accepts history
  - supports serialization: T2 -> T3 -> T1
  - everything okay

- Snapshot Isolation
  - accepts this sequence of operations
  - „logically“ reorders operations: write(B) and read(B)
  - enforces serialization: T1 -> T2 -> T3
  - but NOT equivalent to serial execution of T1; T2; T3
    - reordering creates a cycle in history

Time in 2PL vs. Time in SI

- (Revisited) Definition of History
  - Partial order of all operations
  - Total order of conflict operations
- Histories in 2PL
  - Partial order of all operations
  - Total order of conflict operations
  - No re-ordering of operations
  - Only serializable histories (modulo phantoms, tbd)
- Histories in SI
  - Partial order of all operations
  - Avoids R-W conflicts by reordering R-W operations
    - reads always executed *before* conflicting write operations
  - Abort to deal with W-W conflict operations
  - Allows non-serializable histories

Isolation Levels in SQL92

set transaction

[read only, |read write,]

[**isolation level**

read uncommitted, |

read committed, |

repeatable read, |

serializable,]

[**diagnostic size** ...,]

Isolation Levels in SQL92

- **read uncommitted:**
  - Lowest level of isolation
  - Only allowed for read-only TAs
  - Allows the „uncommitted read" phenomenon

T_1	T_2
	read(A)
	. . .
	write(A)
read(A)	
. . .	
	rollback

  - *Why only for read-only transactions?*

Isolation Level in SQL92

- **read committed:**
  - TAs only read committed versions of objects
  - However, one TA may read different versions
  - Modifiy 2PL: allow short-lived S locks

T_1	T_2
read(A)	
	write(A)
	write(B)
	commit
read(B)	
read(A)	
. . .	

Isolation Level in SQL92

- **repeatable read:**
  - Prevents reading different versions of the same object
  - However, phantoms can happen
- **serializable:**
  - full isolation (no phantoms, etc.)

Money Transfer in the Real World

- TA1: Withdrawal(A, B, M)
  - read(A,a)
  - write(A,a-M)
  - enqueue(valuta, B, M)
  - commit()
- TA2: Valuta to B – periodic batch process
  - dequeue(valuta, B, M)
  - read(B, b)
  - write(B, b+M)
  - commit()
- Reading and writing to Queue is transacted!!!
  - atomicity enforced using 2PC (two-phase commit)

Money Transfer in the Real World

- The world is distributed
  - different banks participate in a „transaction“
  - different services within a bank
- It is difficult (impossible) to implement distrib. ACID
  - queues are a way to decouple entities
- Eventual Atomicity
  - *at some point*, all or nothing (partial visibility in between)
  - failures in TA2 will result in *compensation* of TA1
- Other ACID properties
  - Durability: okay
  - Consistency: okay
  - Isolation: no

- This is good enough for money transfer!!!

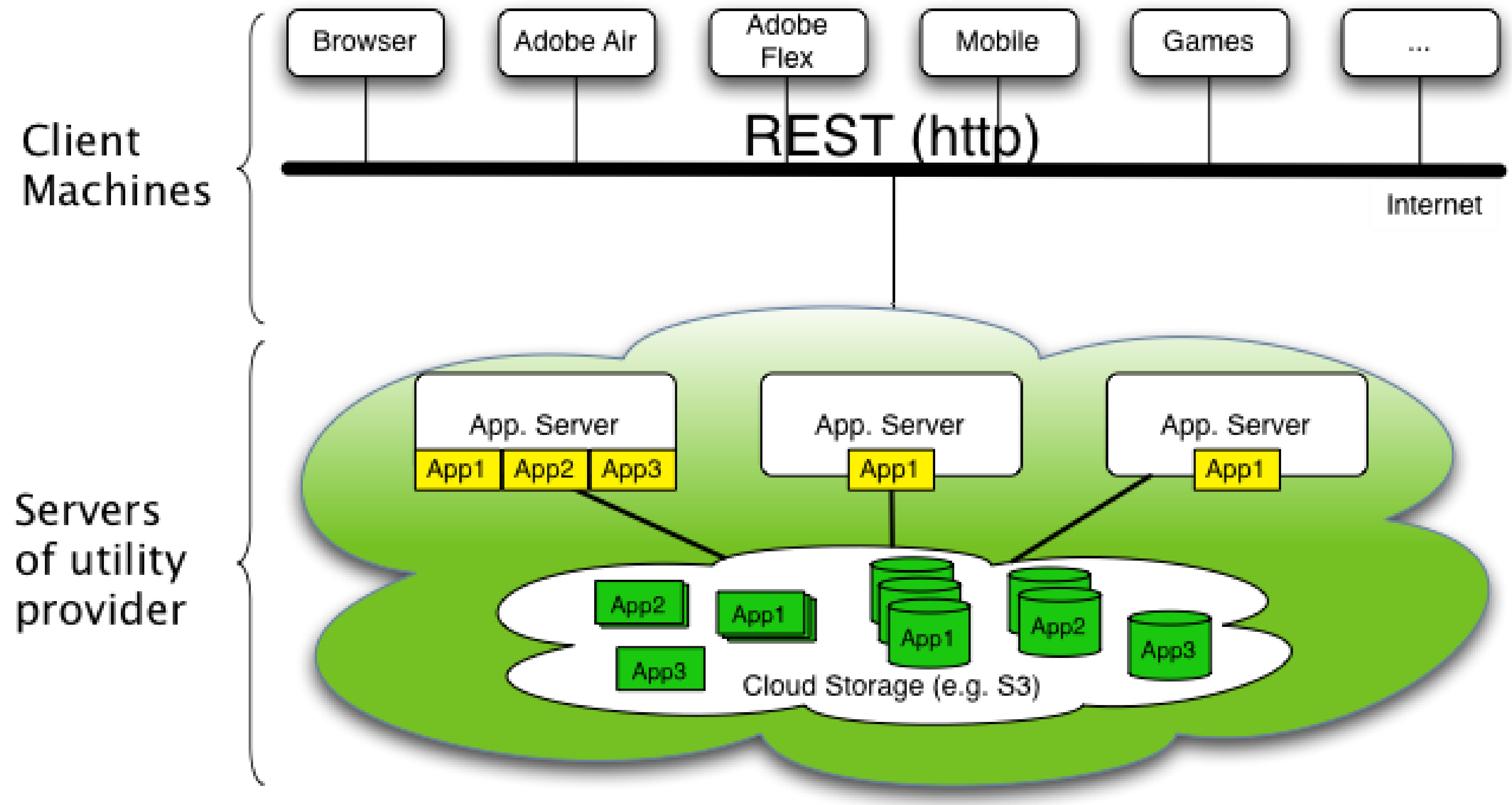
Withdrawal (ATM) in the Real World

- TA1: Reserve money M from Account A
  - read(A.balance, b)
  - read(A.reservation, r)
  - if (b < reservation + M) abort()
  - write(A.reservation, r+M)
  - commit()
- TA2: Periodic process: reservation - withdrawal
  - read(A.reservation, r)
  - if (r = 0) then abort()
  - read(A.balance, b)
  - write(A.balance, b-r)
  - write(A.reservation, 0)
  - commit()
- Why would you do this?

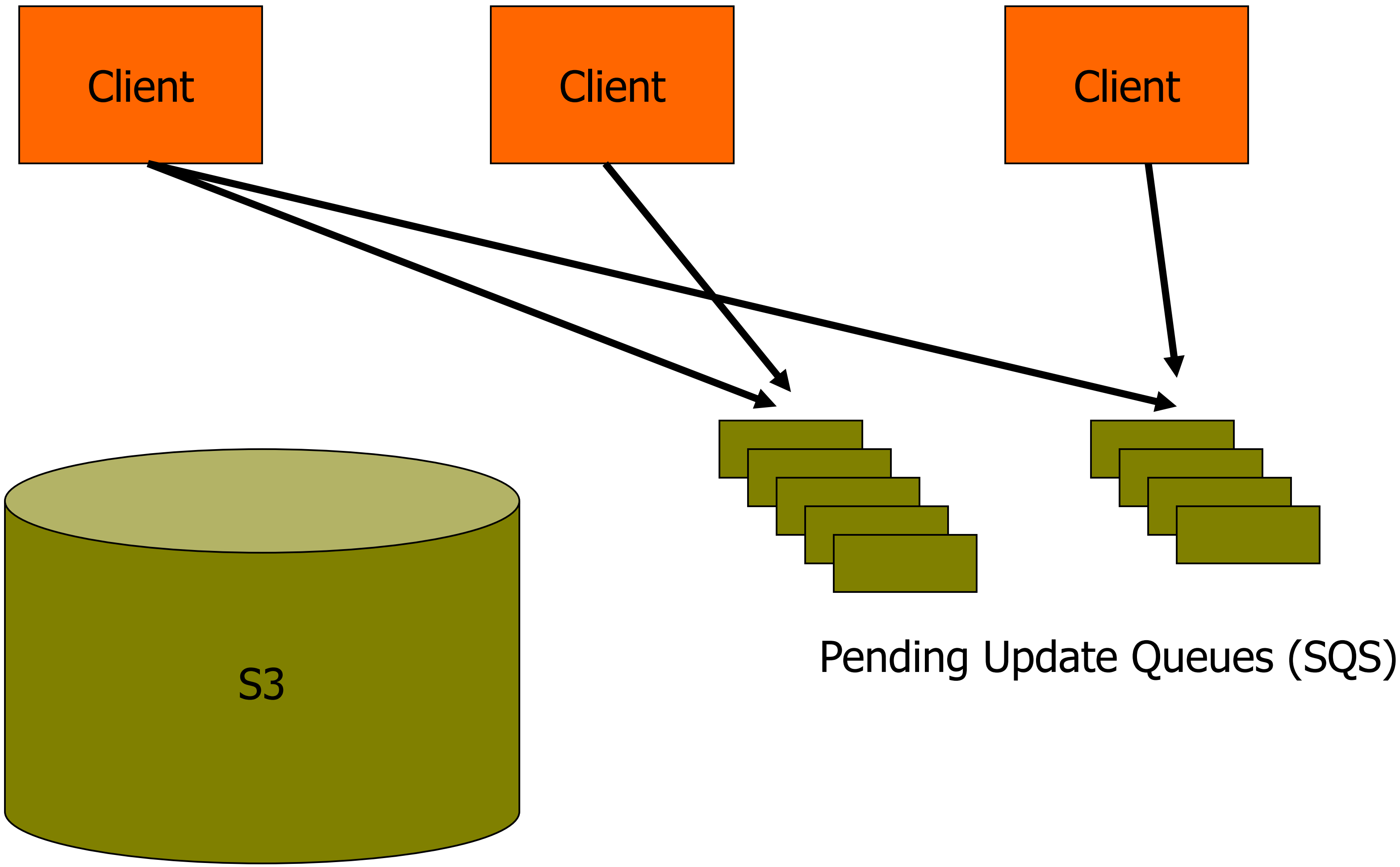
Discussion of Withdrawal

- Similar to a Two-phase commit
  - Get agreement from everybody
  - Implement transaction
- Big advantage
  - decouple participants
  - semantic locking (ESCROW); higher concurrency
    - need not block the $ resource while thinking
- ACID Properties
  - all fulfilled
- Other example: Shopping Cart

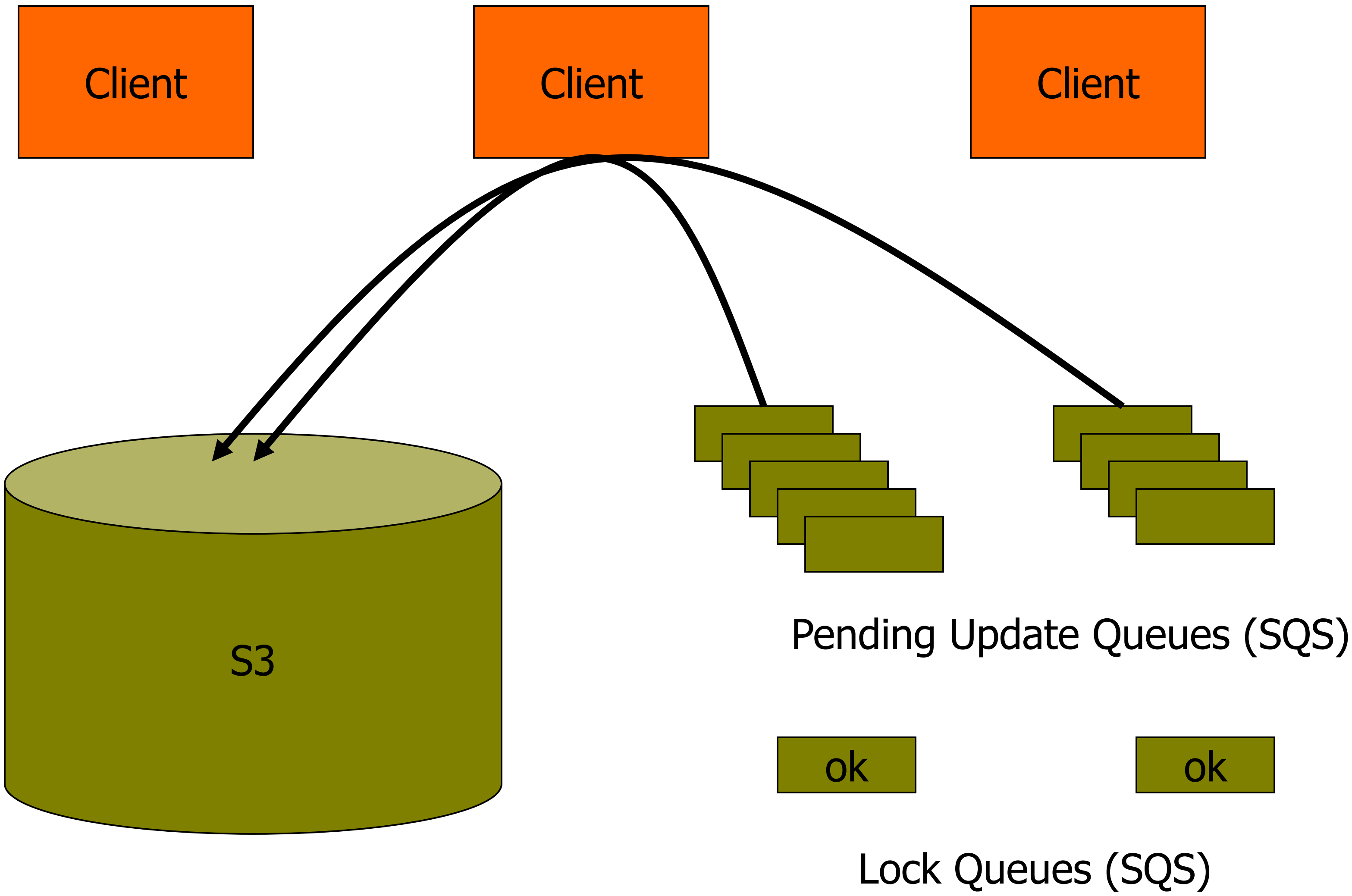
DBs in the Cloud (connected)



Step 1: Clients commit update records to pending update queues



Step 2: Checkpointing propagates updates from SQS to S3



Transaction Management in the Cloud

- Use a distributed key-value store (DHT)
  - Replicate business objects on cheap HW
  - Propagate updates from one copy to the next

- Implement TA Properties on top of that
  - Many different variants
  - Trade consistency for availability
  - Trade consistency for $

- ACID Properties
  - Atomicity: eventual atomicity possible
  - Consistency: okay (compromised by isolation)
  - Isolation: read+write monotonicity
  - Durability: yes