COMP207
Database Development

Lecture 7

Transaction Management:
A Few More Locks and then Dealing with
Transaction Aborts & System Failures

Canceled lecture

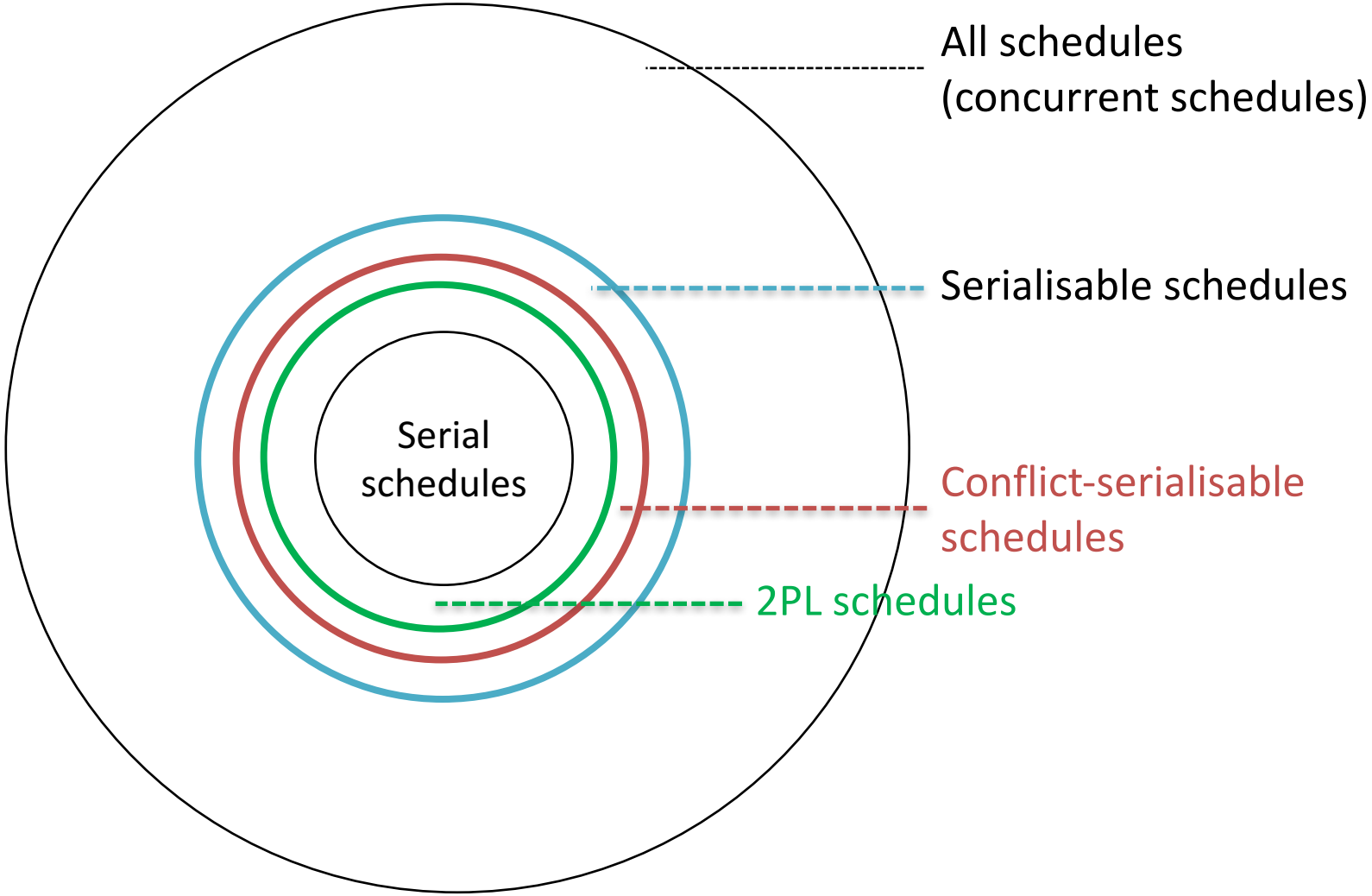
- Lecture tomorrow at 17-18 (5pm-6pm) is **canceled**
  - Prof Paul Spirakis gives an inaugural lecture in ELT 17:30-20

- The lecture tomorrow at 13-14 (1pm-2pm) is **NOT** canceled

Review

- Previous lecture:
  - How to enforce conflict serialisability
  - Common approach: use locks
  - First approximation: simple locking
  - Refinement: two-phase locking (2PL)
  - Varies kinds of locks

Schedules Review



Basic Schedulers

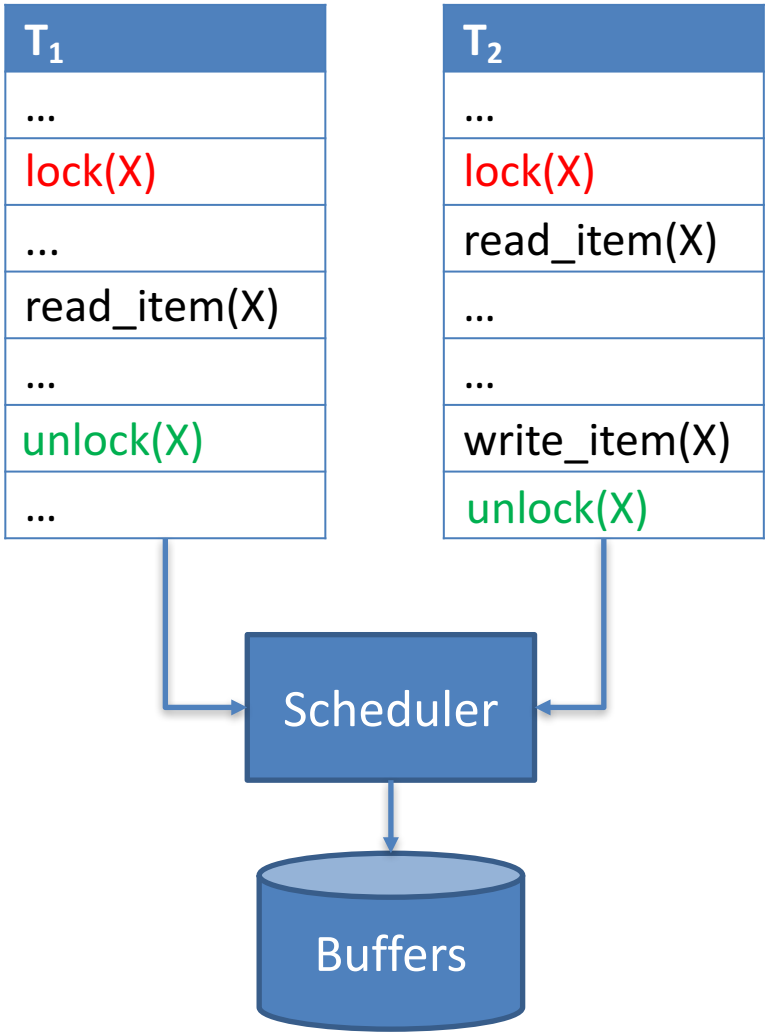
- Serial schedules
  - Ones where transactions are executed one by one
- Concurrent schedules
  - Ones with arbitrary interleaving of transactions (but each transactions operations appear in the order from the transaction)

“Good” schedulers

- Serialisable schedules
  - Ones that behave like some serial ***schedule***
- Conflict-serialisable schedules
  - Ones where all conflicts are in the same order as in some serial ***schedule***
  - (A pair of (read or write) operations are in conflict iff they are from different transactions, accesses the same item and at least one of them is a write)
- 2PL schedules
  - Ones for which the involved ***transactions*** are 2PL

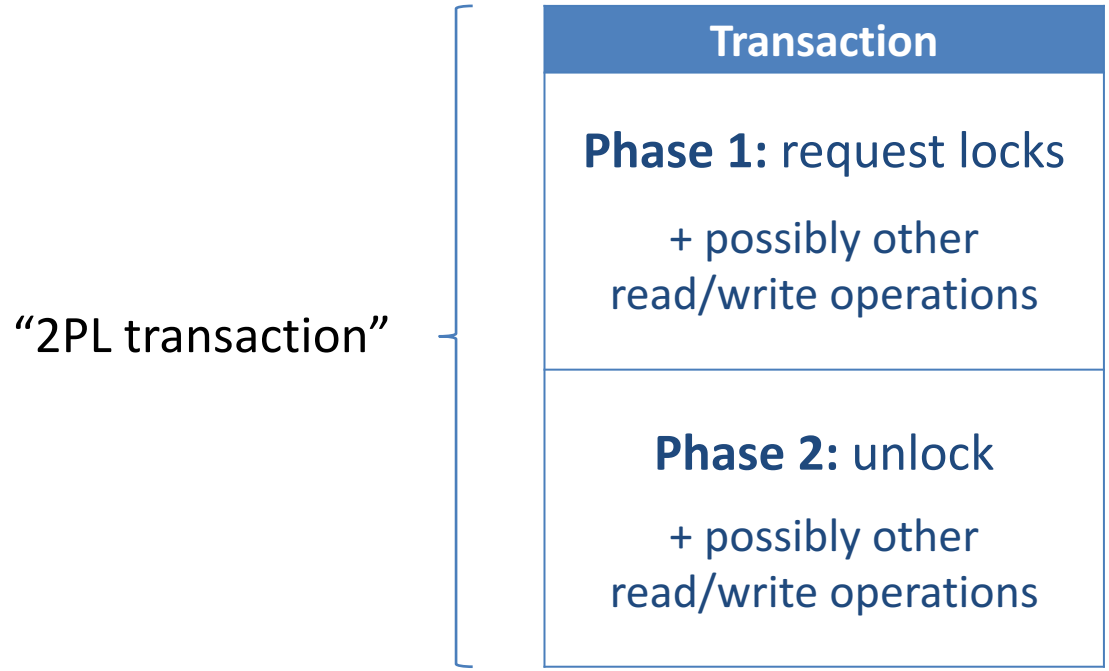
Simple Locking Mechanism

- A transaction has to lock an item before it accesses it.
- Locks are requested from & granted by the scheduler:
  - Each item is locked by at most one transaction at a time.
  - Transactions wait until a lock can be granted.
- Each lock has to be released (unlocked) eventually.
- Problem: (conflict-) serialisability not yet guaranteed



Two-Phase Locking (2PL)

- Modification of the simple locking mechanism that *guarantees conflict-serialisability*
- **Two-phase locking (2PL) condition:**
  In each transaction, all lock operations precede all unlocks.

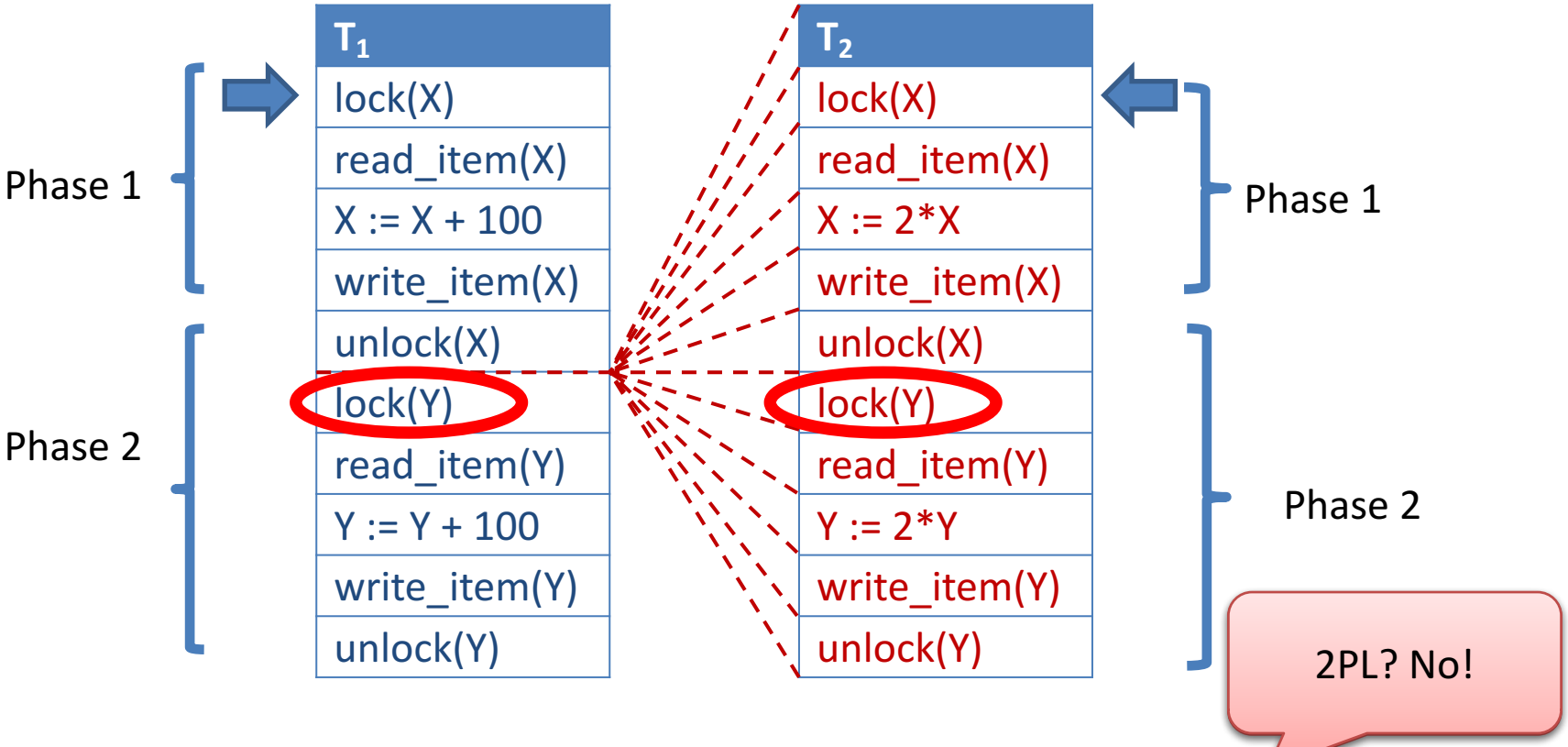


How to test if a transaction is 2PL

- Find first unlock
  - Phase 1 is up to, but not including, that unlock
  - Phase 2 is from that unlock until the end
- Transaction is 2PL iff phase 2 contains ***no*** lock

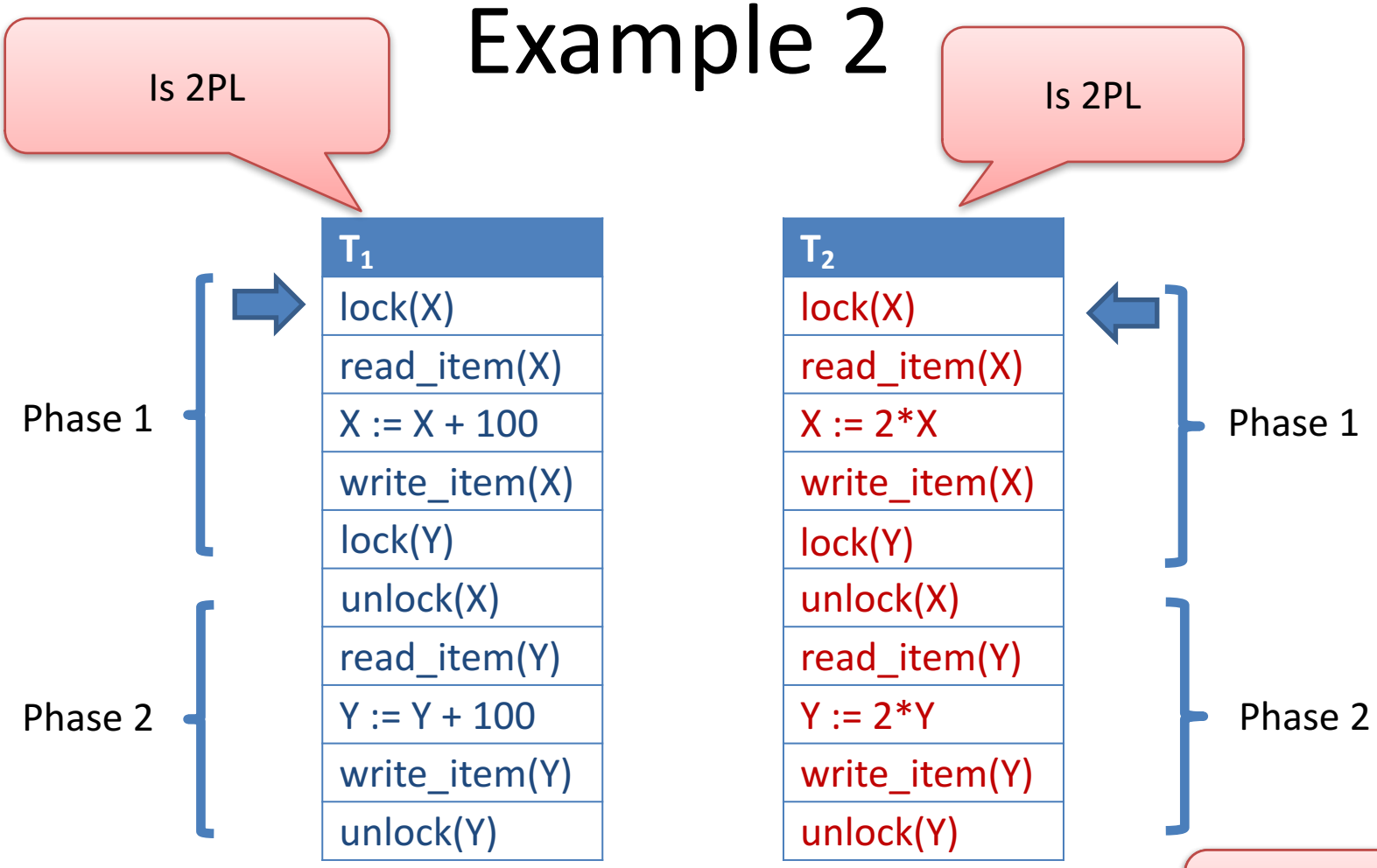
- (A schedule is 2PL iff all its transactions are 2PL)

Example 1



S: $l_1(X); r_1(X); w_1(X); u_1(X); l_2(X); r_2(X); w_2(X); u_2(X);$
$l_2(Y); r_2(Y); w_2(Y); u_2(Y); l_1(Y); r_1(Y); w_1(Y); u_1(Y)$

Example 2



S: $l_1(X);\ r_1(X);\ w_1(X);\ l_1(Y);\ u_1(X);\ l_2(X);\ r_2(X);\ w_2(X);$
$r_1(Y);\ w_1(Y);\ u_1(Y);\ l_2(Y);\ u_2(X);\ r_2(Y);\ w_2(Y);\ u_2(Y)$

2PL? Yes!

Share & Exclusive locks

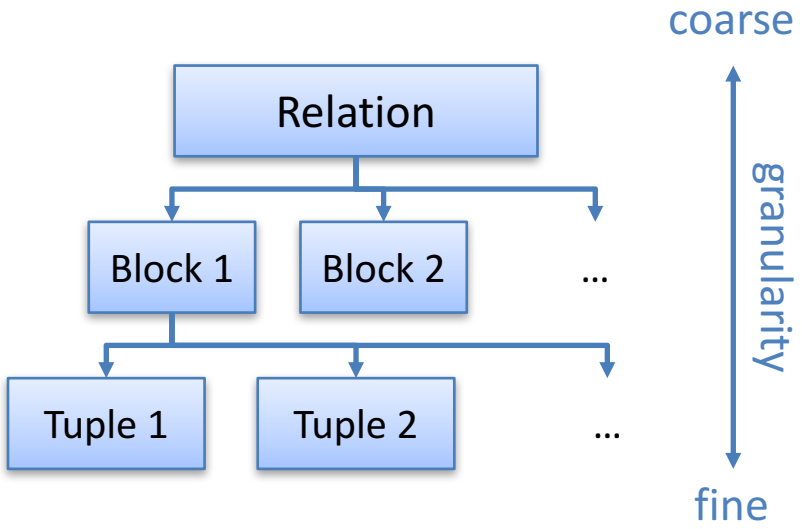
- Exists locks more efficient than basic locks
  - Efficient = allows more concurrency
- First variant has 2 types of locks: shared and exclusive
- Exclusive locks are like basic locks:
  - Exclusive locks can be used for **reads AND writes**
  - Exclusive locks can be granted if none else holds a **lock** on that item
- Share (or read) locks:
  - Share locks can be used for **only reads**
  - Share locks can be granted if none holds an **exclusive lock** on that item
  - (many transactions can have a shared lock for the same item at the same time)
- Can lead to new deadlocks

Update locks

- Second variant eliminates extra deadlocks and has 3 types of locks: shared, exclusive and update locks
- Share and exclusive locks are mostly as before
- Exceptions:
  - Share locks are granted only if none holds an **exclusive or update lock** on the item
  - Exclusive locks are **not** granted if the transaction holds a shared lock on the item
- Update locks:
  - Update locks can be used for **only reads**
  - Update locks can be granted only if none holds an update or exclusive lock on the item

Locks With Multiple Granularity

- DBMS may use locks at different levels of granularity
  - May lock relations
  - May lock disk blocks
  - May lock tuples



- Examples:
  - SELECT name FROM Student WHERE studentID = 123456;
  - SELECT avg(salary) FROM Employee;

Shared lock on tuple suffices

Shared lock on relation might be necessary

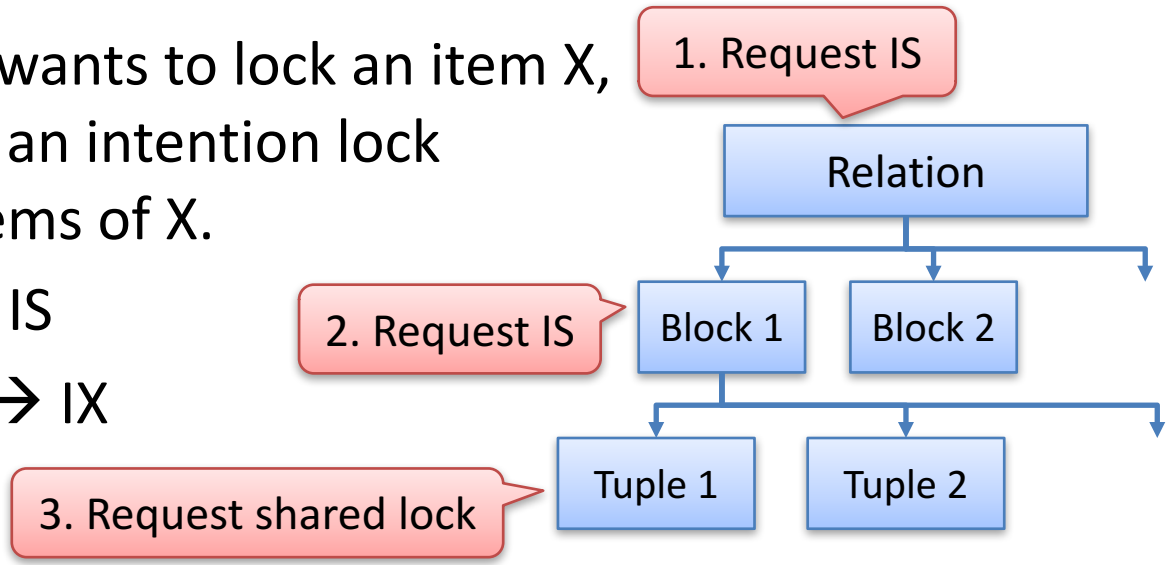
Trade-Offs

- Locking at **too coarse** granularity:
  - Low overhead (don't need to store too much information)
  - Less degree of concurrency: may cause unnecessary delays
- Locking at **too fine** granularity:
  - High overhead: need to keep track of all locked items
  - High degree of concurrency: no unnecessary delays
- Need to prevent issues such as the following to guarantee (conflict-) serialisability:
  - A transactions holds shared lock for a tuple.
  - Another transaction holds exclusive lock for the relation.

Intention Locks

(a.k.a. Warning Locks)

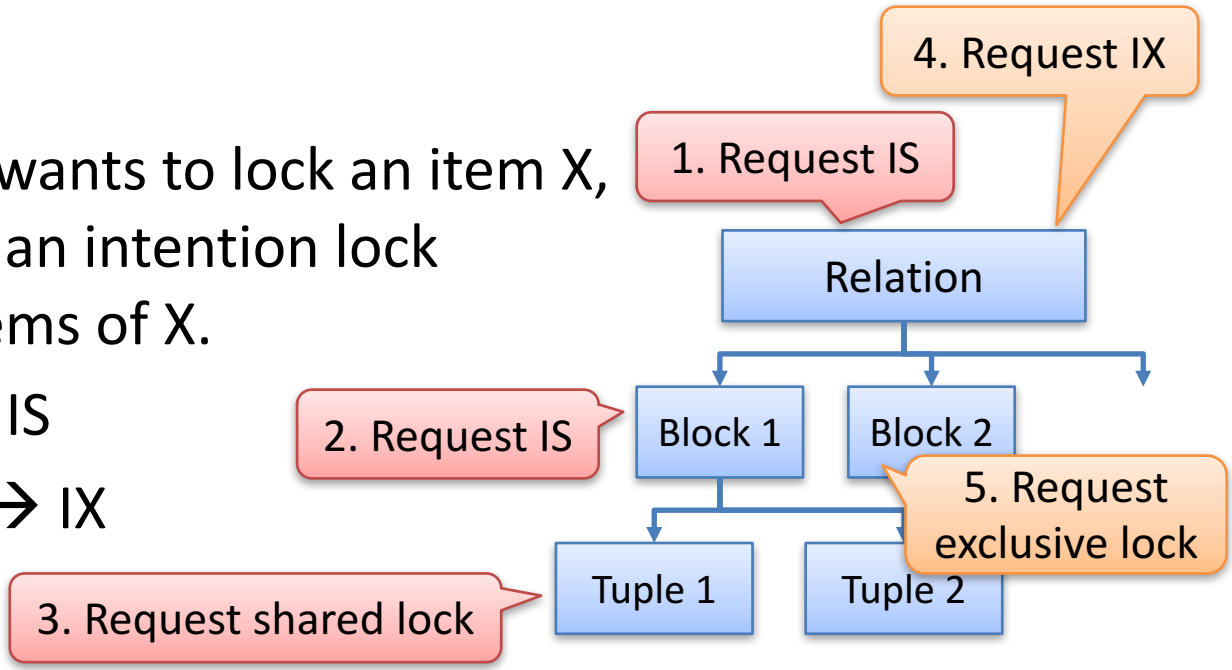
- We use shared and exclusive locks (no update locks)
- New **intention locks**:
  - **IS**: Intention to request a shared lock on a sub-item
  - **IX**: Intention to request an exclusive lock on a sub-item
- Rules:
  - If a transaction wants to lock an item X, it must *first* put an intention lock on the super-items of X.
  - Shared locks → IS
  - Exclusive locks → IX



Intention Locks

(a.k.a. Warning Locks)

- We use shared and exclusive locks (no update locks)
- New **intention locks**:
  - **IS**: Intention to request a shared lock on a sub-item
  - **IX**: Intention to request an exclusive lock on a sub-item
- Rules:
  - If a transaction wants to lock an item X, it must *first* put an intention lock on the super-items of X.
  - Shared locks → IS
  - Exclusive locks → IX



Policy for Granting Locks

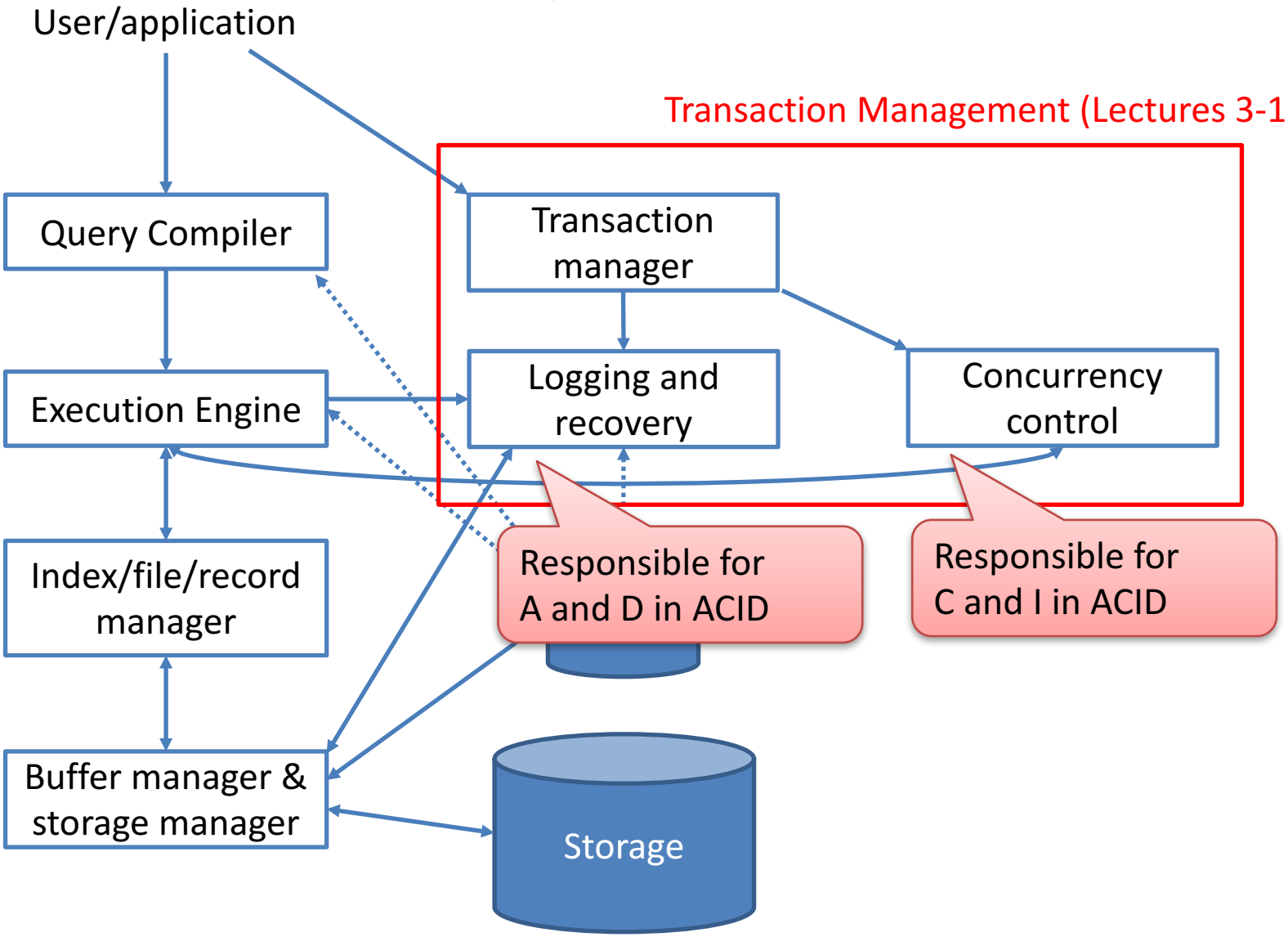
Transaction requests lock of type …

	Shared (S)	Exclusive (X)	IS	IX
Shared (S)	yes	no	yes	no
Exclusive (X)	no	no	no	no
IS	yes	no	yes	yes
IX	no	no	yes	yes

Grant if the only types of locks held by *other* transactions are those with a "yes"

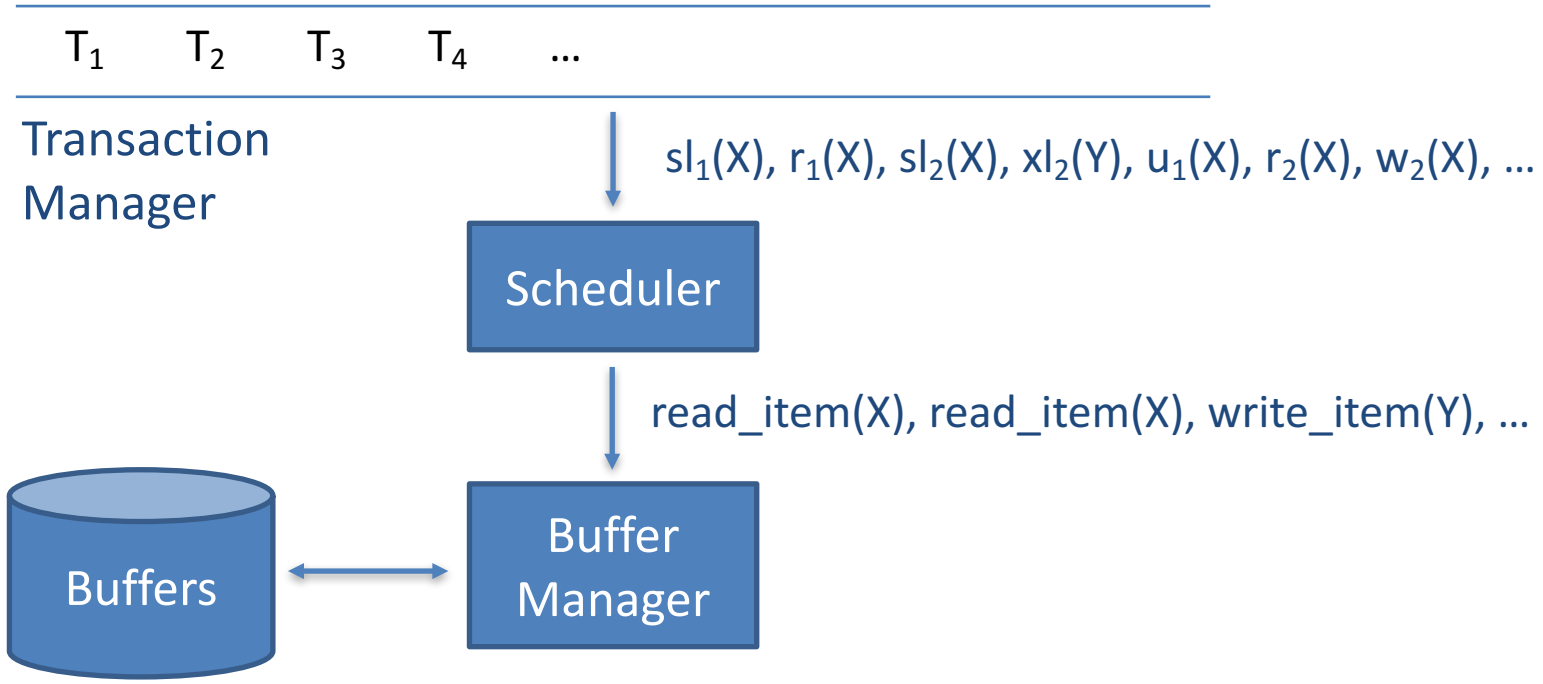
Relational DBMS Components

(Simplified)



Review of Concurrency Control

- So far…. executing transactions concurrently



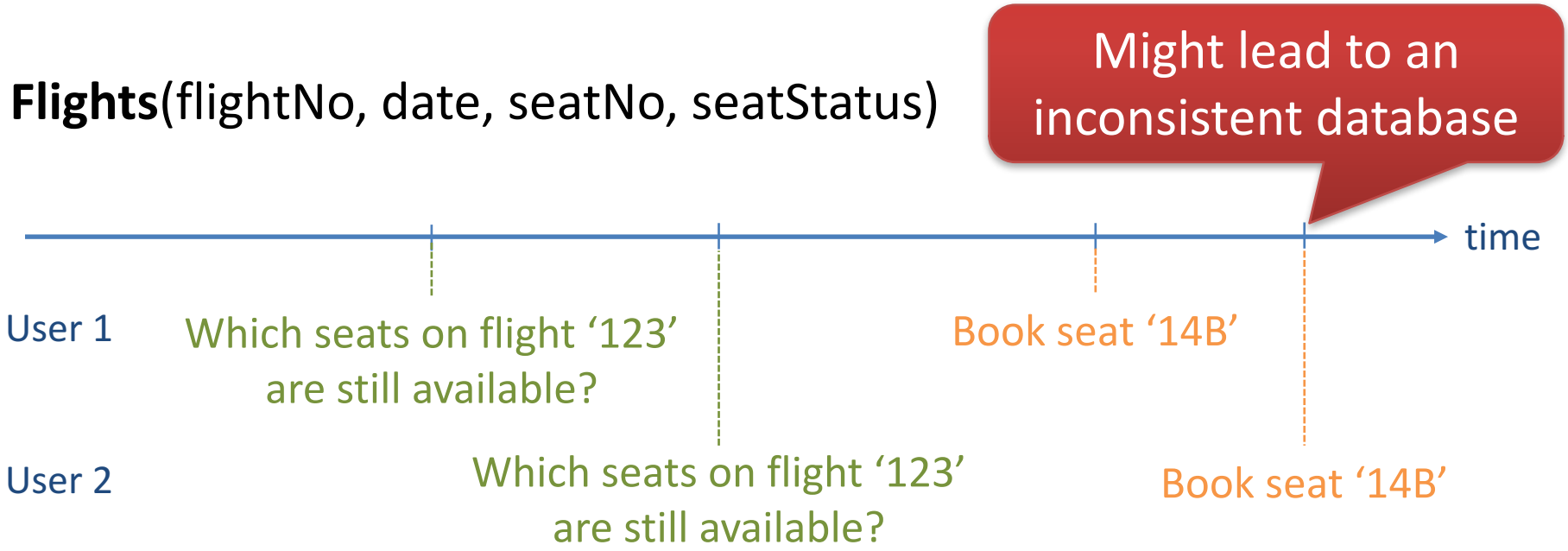
- Problems: transactions may abort & other failures

Why Might a Transaction Abort?

- Errors while executing transactions
  - Violation of integrity constraints, other run-time errors

Problem 1: Concurrency

Flights(flightNo, date, seatNo, seatStatus)




```
SELECT seatNo
FROM   Flights
WHERE  flightNo = 123
   AND date = '2018-10-2
   AND seatStatus = 'available';
```

```
UPDATE Flights
SET    seatStatus = 'occupied'
WHERE  flightNo = 123
   AND date = '2018-10-2'
   AND seatNo = '14B';
```

Why Might a Transaction Abort?

- Errors while executing transactions
  - Violation of integrity constraints, other run-time errors
- Deadlocks
  - E.g., when using two-phase locking
  - Concurrency control requests to abort transactions

Deadlocks

T_1
lock(X)
read_item(X)
X := X + 100
write_item(X)
lock(Y)
unlock(X)
read_item(Y)
Y := Y + 100
write_item(Y)
unlock(Y)

T_2
lock(Y)
read_item(Y)
Y := 2*Y
write_item(Y)
lock(X)
unlock(Y)
read_item(X)
X := 2*X
write_item(X)
unlock(X)

$l_1(X); r_1(X); w_1(X); l_2(Y); r_2(Y); w_2(Y);$ ____?____

T_2's request for lock on X denied

T_1's request for lock on Y denied

Why Might a Transaction Abort?

- Errors while executing transactions
  - Violation of integrity constraints, other run-time errors
- Deadlocks
  - E.g., when using two-phase locking
  - Concurrency control requests to abort transactions
- Explicit request

Later…

```
START TRANSACTION;

INSERT INTO SeatReservations(CustomerID, SeatNo, FlightNo, Date)
VALUES (12345678, '14B', 123, '9/10/2017');

// additional code

ROLLBACK;
```

Requests to undo all the changes

Beyond the DBMS's Control

- Media failures:
  - The medium holding the database becomes partially or completely unreadable
  - Example: changes of bits, head crashes
- Catastrophic events:
  - The medium holding the database is destroyed
  - Examples: explosions, fires, etc.
- System failures
  - Information about the active transaction's state is lost
  - Examples: power failures, software errors

Beyond the DBMS's Control

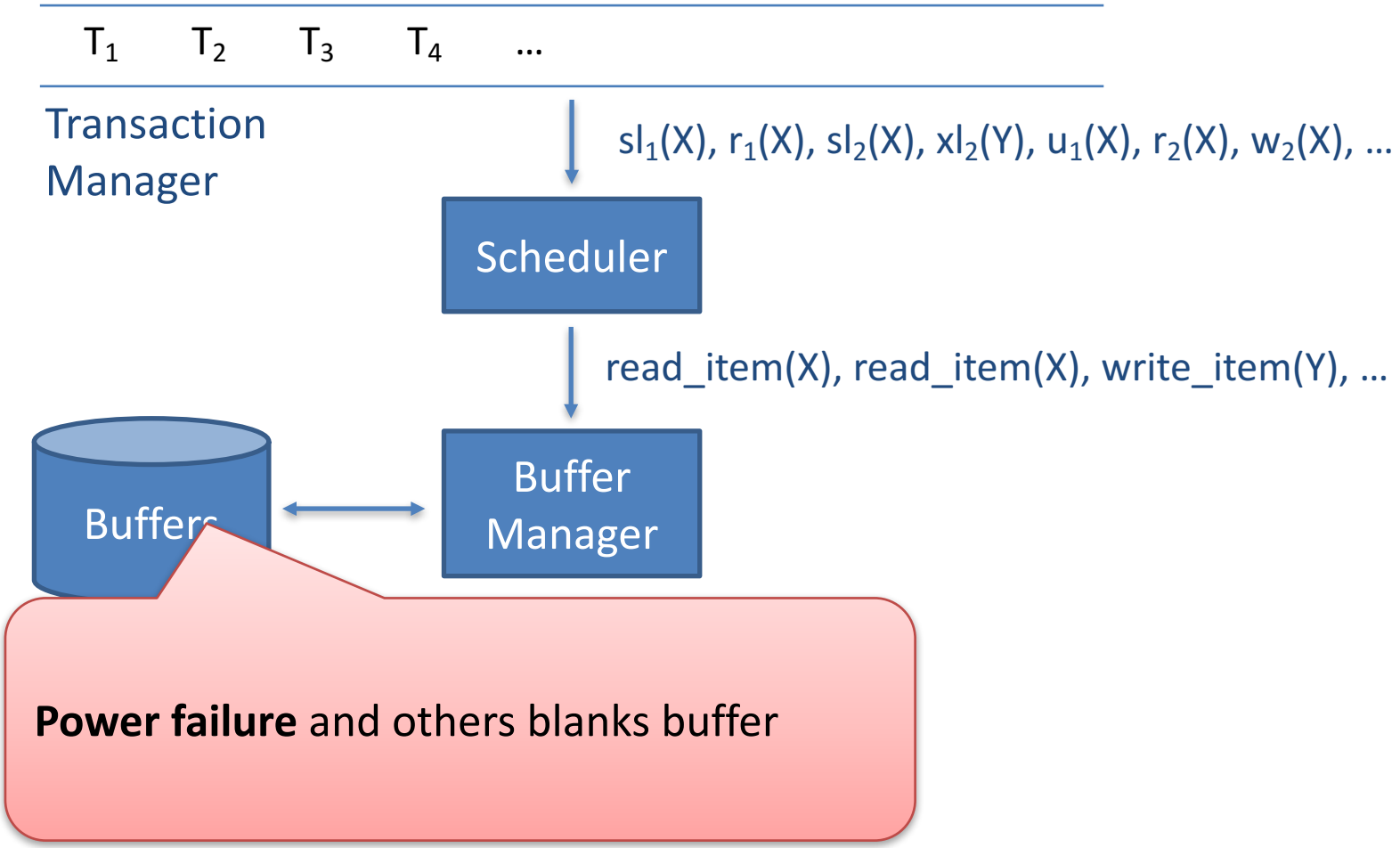
- Media failures:
  - The medium holding th
    completely unreadable
  - Example: changes of bits, head crashes
- Catastrophic events:
  - The medium holding th
  - Examples: explosions, fires, etc.
- System failures
  - Information about the active transaction's state is lost
  - Examples: power failures, software errors

Safeguards:
- Archives: full + incremental
- Controlled redundancy
  - RAID
  - Copies at different locations

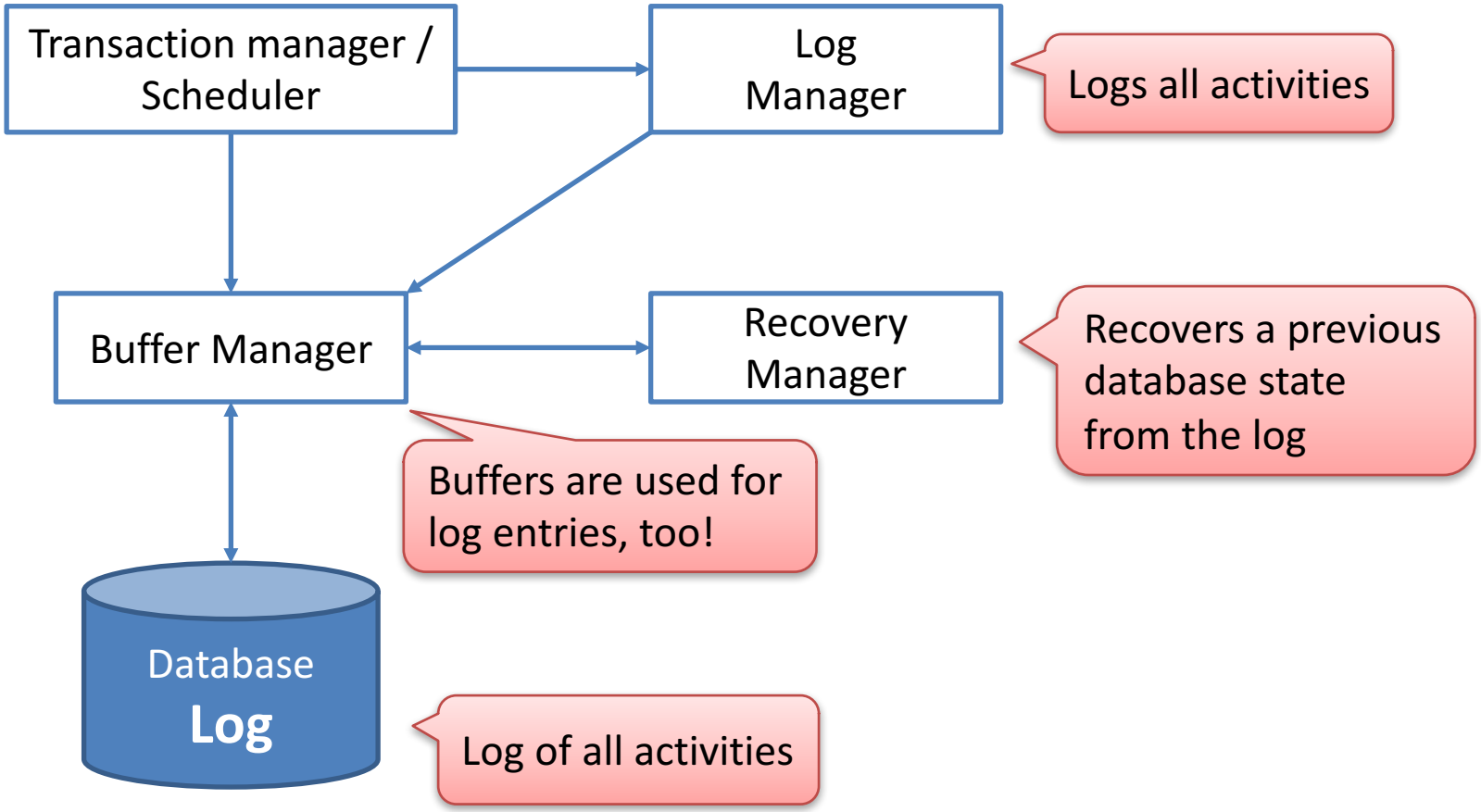
Safeguards:
- Archives at safe and different locations
- Copies at different locations

Review of Concurrency Control

- So far…. executing transactions concurrently



Facilities Relevant For Aborts & System Failures

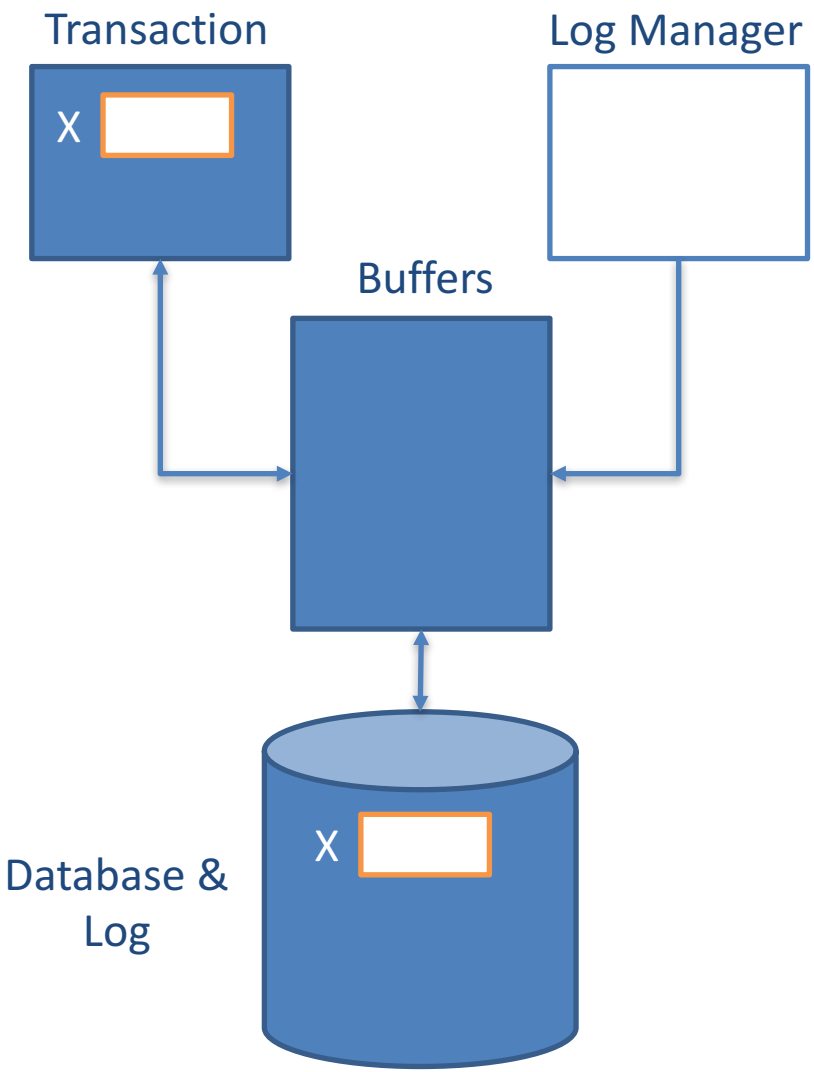


Logging
(The Log and the Log Manager)

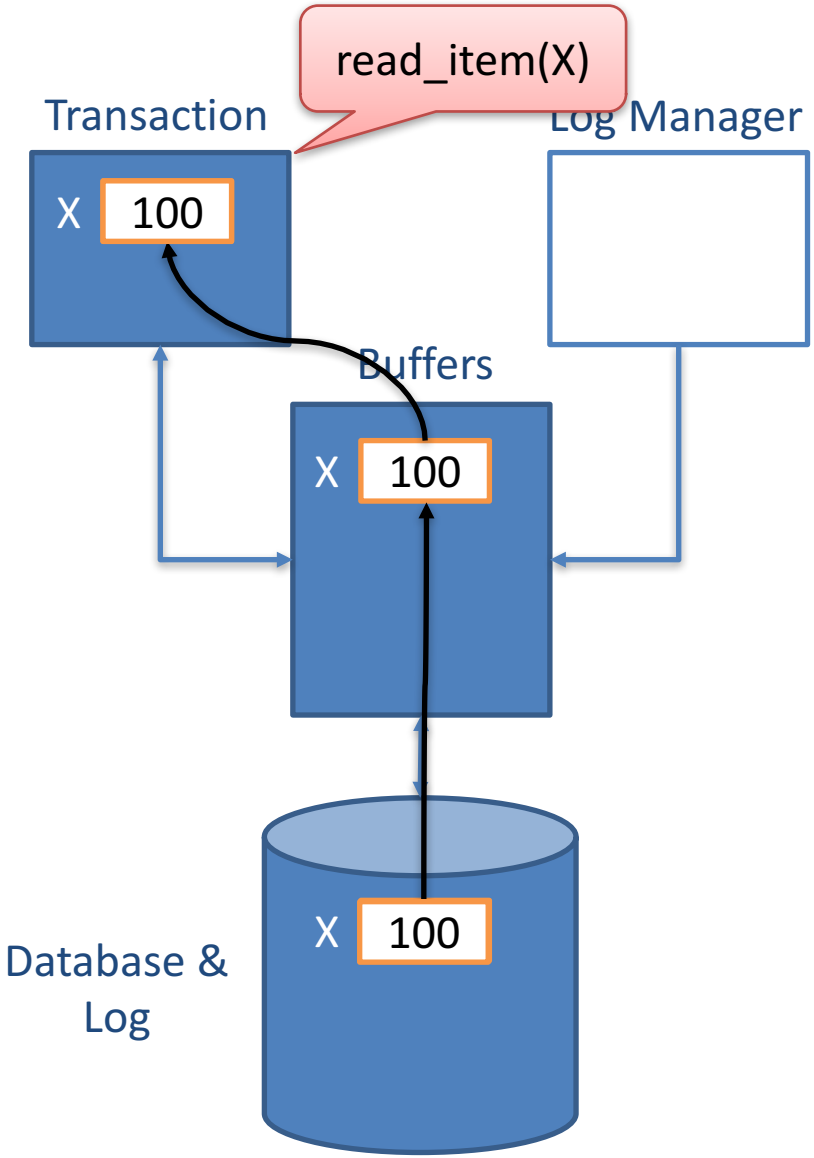
Logging in DBMS

- Idea: write important activities to a log so that a desired database state can be recovered later
- Examples of important activities:
  - Starts of transactions, commits, aborts
  - Modification of database items
- Should work *even in case of system failures*!
- Techniques:
  - Undo logging (for maintaining *Atomicity*)
  - Redo logging (for maintaining *Durability*)
  - Combinations thereof

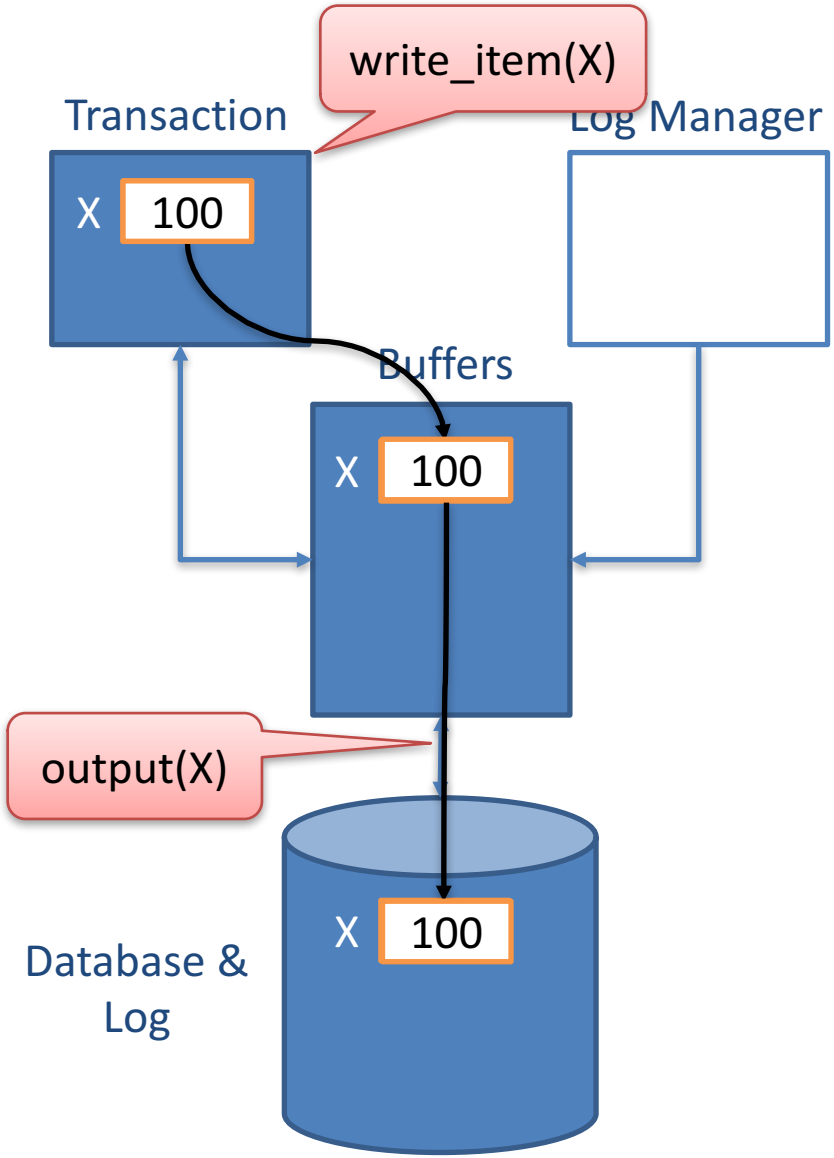
Extension of Transaction Syntax



Extension of Transaction Syntax



Extension of Transaction Syntax

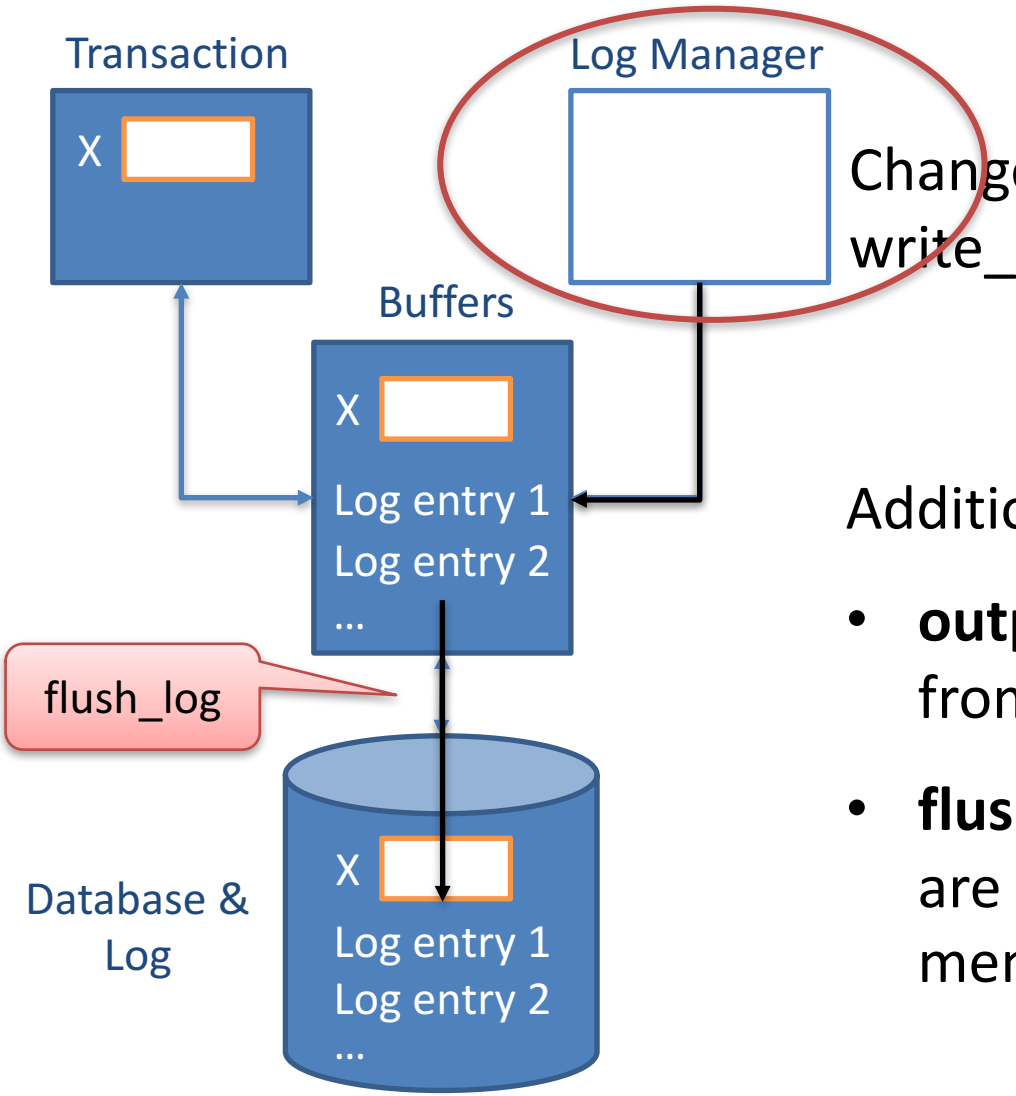


Changes:
write_item(X) writes **only** to buffer

Additional operations:

- **output(X)**: copy database item X from buffer to database

Extension of Transaction Syntax



Changes:
write_item(X) writes **only** to buffer

Additional operations:

- **output(X)**: copy database item X from buffer to database
- **flush_log**: write log entries that are currently residing in main memory (buffer) to log

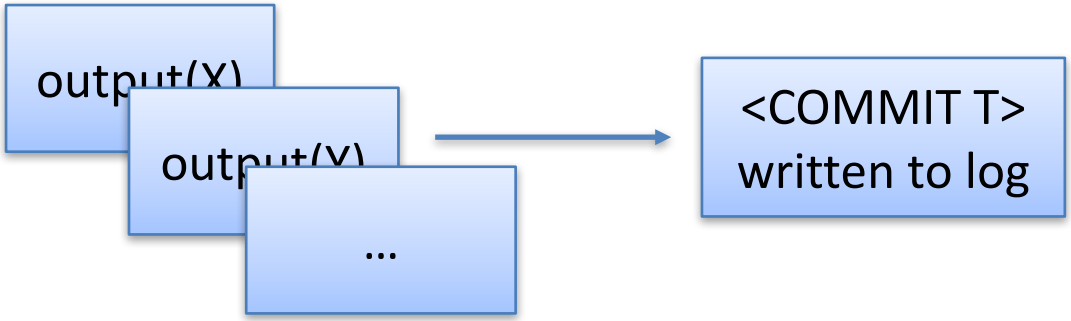
Undo Logging

- Logs activities with the goal of restoring ("undoing") a previous database state.
- Log records (or log entries):
  - **<START T>**: Transaction T has started.
  - **<COMMIT T>**: Transaction T has committed.
  - **<ABORT T>**: Transaction T was aborted.
  - **<T, X, v>**: Transaction T has updated the value of database item X, and the old value of X was v.
    - Response to **write_item(X)**
    - If this entry occurs in the log, then the new value of X might not have been written to the database yet.
- Slightly different records for redo logging (later…)

Undo Logging: Procedure

1. If transaction T updates database item X and the old value was v, then <T, X, v> must be written to the log on disk **before** X is written to disk.

2. If transaction T commits, then <COMMIT T> must be written to disk **as soon as** all database elements changed by T have been written to disk



Example

Transaction
read_item(X)
X := X*2
write_item(X)
read_item(Y)
Y := Y*2
write_item(Y)
flush_log
output(X)
output(Y)
flush_log

Writes all log entries for updates to disk

Writes all updates to disk

Writes the <COMMIT T> record to disk

Example

		Local		Buffer		Database		
Time	**Transaction**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**Log record written**
0						1	10	<START T>
1	read_item(X)	1		1		1	10	
2	X := X*2	2		1		1	10	
3	write_item(X)	2		2		1	10	<T, X, 1>
4	read_item(Y)	2	10	2	10	1	10	
5	Y := Y*2	2	20	2	10	1	10	
6	write_item(Y)	2	20	2	20	1	10	<T, Y, 10>
7	**flush_log**	2	20	2	20	1	10	
8	**output(X)**	2	20	2	20	2	10	
9	**output(Y)**	2	20	2	20	2	20	
10		2	20	2	20	2	20	<COMMIT T>
11	**flush_log**	2	20	2	20	2	20	

What if a system failure occurs? (The Recovery Manager)

Scenario 1

- <COMMIT T> occurs in the log on disk

T has committed successfully (no recovery needed)

Time	Transaction	X	Y	X	Y	X	Y	
0						1	10	<START T>
1	read_item(X)	1		1		1	10	
2	X := X*2	2		1		1	10	
3	write_item(X)	2		2		1	10	<T, X, 1>
4	read_item(Y)	2	10	2	10	1	10	
5	Y := Y*2	2	20	2	10	1	10	
6	write_item(Y)	2	20	2	20	1	10	<T, Y, 10>
7	**flush_log**	2	20	2	20	1	10	
8	**output(X)**	2	20	2	20	2	10	
9	**output(Y)**	2	20	2	20	2	20	
10		2	20	2	20	2	20	<COMMIT T>
11	**flush_log**	2	20	2	20	2	20	

Error

Scenario 2

- <COMMIT T> **does not occur** in the log on disk

Time	Transaction	X	Y	X	Y			
0								
1	read_item(X)	1		1				
2	X := X*2	2		1				
3	write_item(X)	2		2		1	10	<T, X, 1>
4	read_item(Y)	2	10	2	10	1	10	
5	Y := Y*2	2	20	2	10	1	10	
6	write_item(Y)	2	20	2	20	1	10	<T, Y, 10>
7	**flush_log**	2	20	2	20	1	10	
8	**output(X)**	2	20	2	20	2	10	
9	**output(Y)**	2	20	2	20	2	20	
10		2	20	2	20	2	20	<COMMIT T>
11	**flush_log**	2	20	2	20	2	20	

Must undo all updates to database items that were written to disk.

For each log record <T, X, v> on disk, replace X on disk by v.

Error

Scenario 3

- <COMMIT T> **does not occur** in the log on disk

What to do in this case?

Time	Transaction	X	Y	X	Y	X	Y	Lo
0						1	10	<START T>
1	read_item(X)	1		1		1	10	
2	X := X*2	2		1		1	10	
3	write_item(X)	2		2		1	10	<T, X, 1>
4	read_item(Y)	2	10	2	10	1	10	
5	Y := Y*2	2	20	2	10	1	10	
6	write_item(Y)	2	20	2	20	1	10	<T, Y, 10>
7	**flush_log**	2	20	2	20	1	10	
8	**output(X)**	2	20	2	20	2	10	
9	**output(Y)**	2	20	2	20	2	20	
10		2	20	2	20	2	20	<COMMIT T>
11	**flush_log**	2	20	2	20	2	20	

Error

Recovery With Undo Logs
(Simple Variant)

- If an error occurs, the recovery manager restores the last consistent database state
- Traverses the undo log backwards
- If the current entry is…
  - **<COMMIT T>**: remember that T was committed successfully
  - **<ABORT T>**: remember that T was aborted
  - **<T, X, v>**: if T has not finished successfully (no COMMIT, no ABORT), change the value of X on disk to v
- Write **<ABORT T>** for each *uncommitted* transaction T that was not previously aborted & call flush_log

Summary

- Aborted transactions & system failures can be dealt with using careful logging & restoring data using logs
- Undo logging
  - Maintains Atomicity
  - Logs old value for each updated database item
  - Recovery manager: use this information to restore the last consistent database state