Review 3

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 & 004 only**

Announcements

- Final exam: 9AM August 11th @ PAC 2,4

Why we need transactions

- A database is a shared resource accessed by many users and processes concurrently.
  - Both queries and modifications

- Not managing this concurrent access to a shared resource will cause problems
  - Problems due to concurrency
  - Problems due to failures

Case For Isolation During Concurrent Access

- Clients want concurrency, because databases are designed to be used my multiple clients, and DBMSs can exploit parallelism
- Clients also want: to access the db ***in isolation***, i.e., run a set of queries and statement as if no others are running concurrently.
- All or nothing guarantee: Run the set of statements only if the DBMS can guarantee that they were *all running atomically as if in isolation*.
- Any guarantee on subsets of statements is not useful.

Case For Atomicity To Handle Failures

- All or nothing guarantee: Run the set of statements only if the DBMS can guarantee that they *will all succeed and be persistent or all will fail and no update they make will be persistent.*

Transactions solve Concurrency & Failure Problems

- Transactions : a set of queries/updates that are treated as an atomic unit
- Transactions (appear to) run in isolation during concurrent access (different levels of isolation exist; see later in lecture).
- Transactions are atomic, ie., either all queries/statement will run and persist any modifications to the DBMS, or none will.
- From users' perspective: By wrapping a set of queries/updates in one transaction, users obtain concurrency and resilience guarantees
- Note: internally DBMSs use 2 completely different algorithms/protocols to provide these functionalities for transactions
  - E.g.: locking for concurrency; logging for resilience (lecture 19)

ACID Properties

- Transactions provide 4 main properties known as *ACID properties*:

  A: Atomicity

  C: Consistency

  I: Isolation

  D: Durability

ACID: Atomicity

- Provides all-or-nothing guarantee

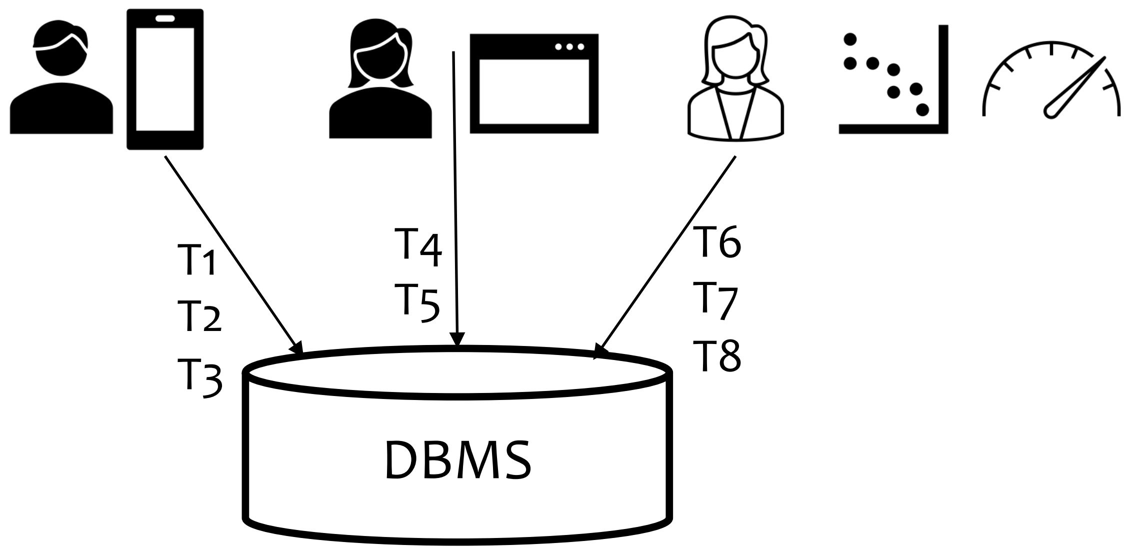
- Partial effects of a transaction must be undone when
  - User explicitly aborts the transaction using ROLLBACK
  - The DBMS crashes before a transaction commits

- Partial effects of a modification statement must be undone when any constraint is violated
  - Some systems roll back only this statement and let the transaction continue; others roll back the whole transaction

How is atomicity achieved?

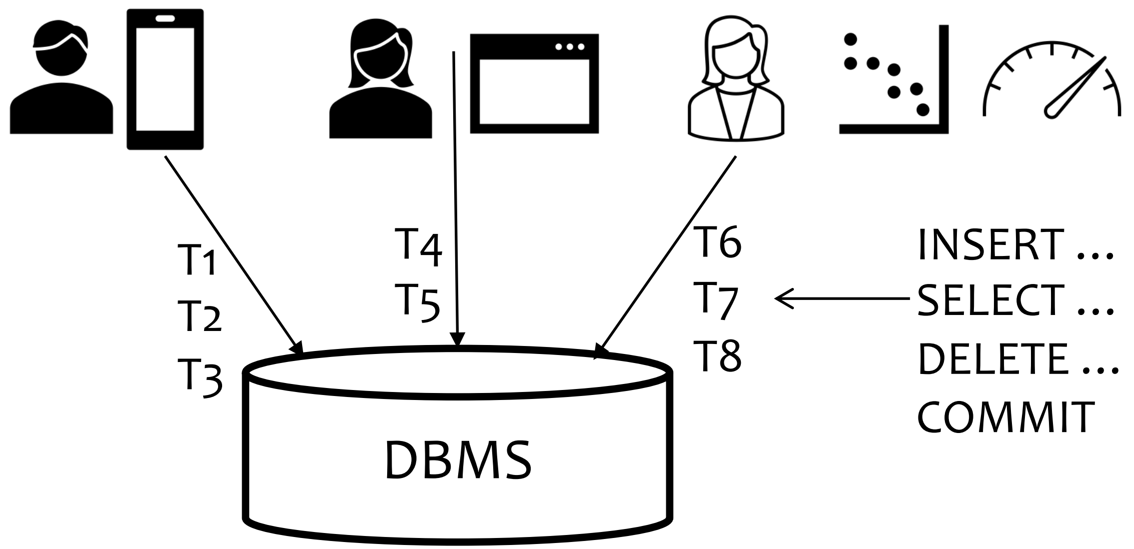
Logging (to support undo) –lecture 19

ACID: Consistency



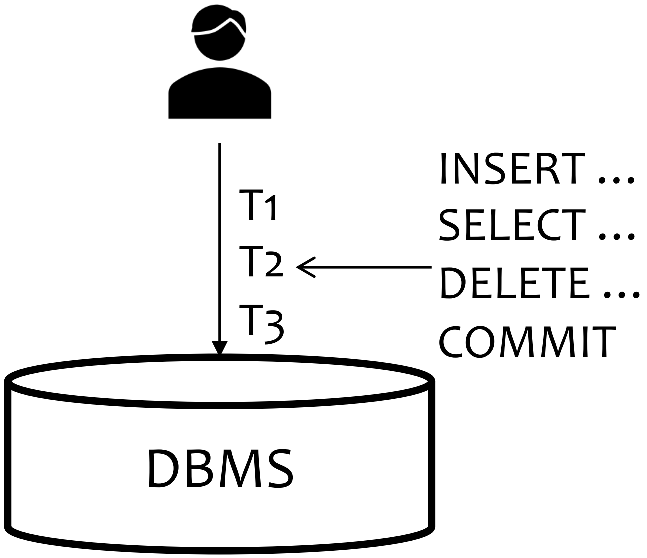
- Guaranteed by constraints and triggers declared in the database and/or transactions themselves
  - E.g., Order amount > 0
- Whenever inconsistency arises,
  - abort the statement or transaction, or
  - fix the inconsistency within the transaction

ACID: Isolation (focus of this lecture)



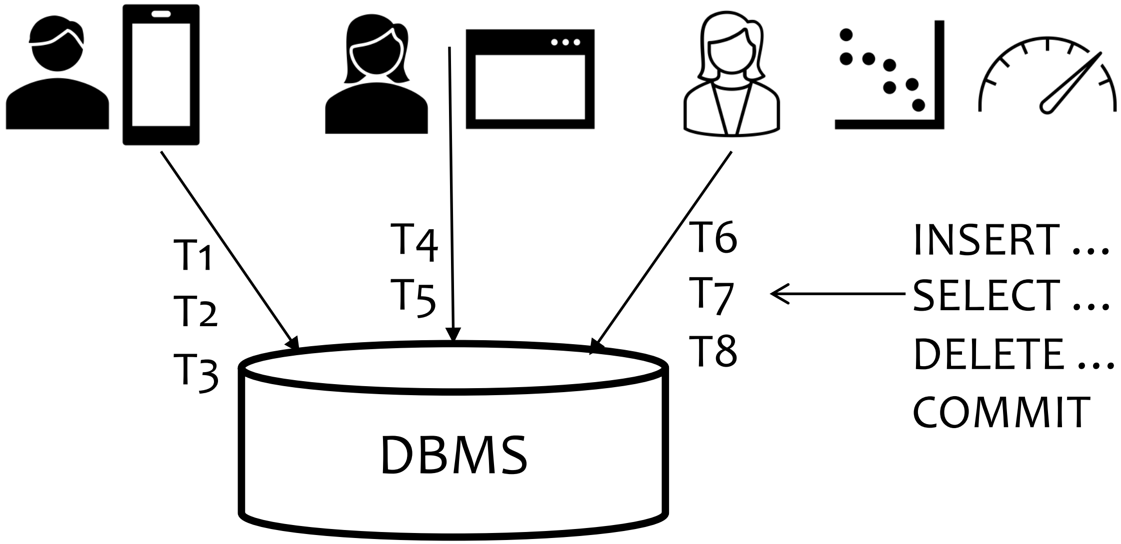
- Serializability: A set of transactions **T** might run concurrently and interleave but final outcome is equivalent to ***some serial order*** of executing the transactions in **T**.
- But DBMSs also provide lower isolation guarantees (later).
- Question to ponder: How can a DBMS guarantee serializability?
- Locking or "verifying modifications at commit time" (next lecture)

ACID: Durability



- Durability: Handles guarantees for *crashes after commit*
  - Guarantee: all modifications will persist
- Question to ponder: How can a DBMS guarantee durability?
- Logging (Lecture 19)

Problems With Serializability



- Serializability: A set of transactions **T** might run concurrently and interleave but final outcome is equivalent to ***some serial order*** of executing the transactions in **T**.
- Best consistency guarantee!
- Guaranteeing at the system-level has performance overheads.
- Q: Can users get weaker guarantees but at higher performance?

Weaker Isolation Levels

Stronger Consistency

Higher Overheads

Less Concurrency

Isolation Levels in SQL Standard
Read Uncommitted
Read Committed
Repeatable Read
Serializable

Weaker Consistency

Lower Overheads

More Concurrency

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
SELECT * FROM Order;
...
COMMIT TRANSACTION
```

How to handle two concurrent transactions with different isolation levels? → CS 448

READ UNCOMMITTED

- Can read *dirty data: an* item written by an uncommitted txn

Txn 1:
UPDATE Order
SET price = price + 5
WHERE oid = o1 || oid = o2

Txn 2: (READ UNCOMMITTED)
SELECT sum(price) FROM Order
WHERE oid = o1 || oid=o2

time	Txn 1	Txn 2
	r:(o1,$20)	
	w:(o1,$25)	
		r:(o1, $25)
		r:(o2, $40)
		commit
	r:(o2,$40)	
	w:(o2,$45)	
	commit	

If Serializable would either read:

(i) o1=20 & o2=40; Sum=60; or

(ii) o1=25 & o2=45; Sum=70

- This can happen and no errors would be given.
- If approx. results OK, e.g., computing statistics, e.g., avg price, one can optimize perf. over consistency and pick read uncommitted

Note on Dirty Reads of The Same Transaction

- There is no such thing as dirty read of the same txn!
- Every (uncommitted) txn will read values it has written.
- That is not considered "dirty" even if it comes from uncommitted txn.

Suppose there is only 1 transaction running

```
BEGIN TRANSACTION
UPDATE Order
SET price = price + 5
WHERE oid = o1

SELECT price FROM Order
WHERE oid = o1;

COMMIT
```

Suppose sets 20->25

Will read 25 (not considered a dirty read)

READ COMMITTED

- No dirty reads but *reads of the same item may not be repeatable*.

Txn 1:
UPDATE Order
SET price = price + 5
WHERE oid = o1 || oid = o2

Txn 2: (READ COMMITTED)
SELECT sum(price) FROM Order
WHERE oid = o1 || oid=o2

SELECT sum(price) FROM Order
WHERE oid = o1 || oid=o2

Txn 1	Txn 2
r:(o1,$20)	
w:(o1,$25)	
	r:(o1, $20)
	r:(o2, $40)
r:(o2,$40)	
w:(o2,$45)	
commit	
	r:(o1, $25)
	r:(o2, $45)
	commit

time

- This behavior is allowed.
- Still not serializable: serializable execution would give 60 or 70 twice.

REPEATABLE READ

- No repeatable reads but *phantom reads may appear*

Txn 1:
UPDATE Order SET price = price+5
WHERE oid = o1

INSERT INTO Order VALUES (o3, 10)

Txn 2: (REPEATABLE READ)
SELECT sum(price) FROM Order

SELECT sum(price) FROM Order

Txn 1	Txn 2
r:(o1,$20)	
w:(o1,$25)	
	r:(o1, $20)
	r:(o2, $40)
w:(o3,$10)	
commit	
	r:(o1, $20)
	r:(o2, $40)
	r:(o3, $10) ← phantom read
	commit

time ↓

- Suppose only o1 and o2 exist
- Still not serializable: serializable would give 60 or 75 twice.
- Provided as a by-product of locking protocols in DBMSs

SERIALIZABLE

- All the three anomalies should be avoided:
  Dirty reads
  Unrepeatable reads
  Phantoms

- For any two txns T1 and T2:
  - Serial executions of T1 and T2 definitely prevent the three anomalies:
    T1 followed by T2 or T2 followed by T1

- Can we run T1 and T2 concurrently and achieve the same serial effect?

Summary of Isolation Levels

Isolation level/read anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

Example: Lowest Isolation Level To Set? (1)

- -- T1:
  INSERT INTO Order
  VALUES (03,10)
  COMMIT;

Isolation level	Possible anomalies for T1
READ UNCOMMITTED	Dirty reads
READ COMMITTED	Unrepeatable Reads
REPEATABLE READ	Phantoms
SERIALIZABLE	None

- Consider other possible concurrent transactions
  - Does not do any reads
  - No read concern
  - Lowest isolation level: read uncommitted

Example: Lowest Isolation Level To Set? (2)

- -- T1:
  UPDATE Order
  SET price = 25
  WHERE oid = 01;
  COMMIT;

Isolation level	Possible anomalies for T1
READ UNCOMMITTED	Dirty reads
READ COMMITTED	Unrepeatable Reads
REPEATABLE READ	Phantoms
SERIALIZABLE	None

- Consider other possible concurrent transactions
  - Does not read same item twice: reads Order only once
  - Only concern: transaction T2 might be updating oid=01 => may lead to dirty reads
  - Lowest isolation level: read committed

Example: Lowest Isolation Level To Set? (3)

- -- T1:
  ```
  SELECT sum(price)
  FROM Order;
  COMMIT;
  ```

Isolation level	Possible anomalies for T1
READ UNCOMMITTED	Dirty reads
READ COMMITTED	Unrepeatable Reads
REPEATABLE READ	Phantoms
SERIALIZABLE	None

- Consider other possible concurrent transactions
  - Does not read same item twice: reads User only once
  - Only concern: transaction T2 might be updating Order => may lead to dirty reads
  - Lowest isolation level: read committed

Example: Lowest Isolation Level To Set? (4)

- -- T1:
  SELECT AVG(price)
  FROM Order;

  SELECT MAX(price)
  FROM Order;
  COMMIT;

Isolation level	Possible anomalies for T1
READ UNCOMMITTED	Dirty reads
READ COMMITTED	Unrepeatable Reads
REPEATABLE READ	Phantoms
SERIALIZABLE	None

- Consider other possible concurrent transactions
  - Now reads same tuples twice
  - Concerns: transaction T2 might be inserting/updating/deleting a row to Order, i.e., reads many not be repeatable and phantoms might appear
  - Lowest isolation level: serializable

Execution histories (or schedules)

- An execution history over a set of transactions $T_1 \dots T_n$ is an interleaving of the operations of $T_1 \dots T_n$ in which the operation ordering imposed by each transaction is preserved.

- Two important assumptions:
  - Transactions interact with each other only via reads and writes of objects
  - A database is *a fixed set* of *independent* objects

- Example: $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]c_1r_2[x]r_2[y]c_2$
  - $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$ [next slide expands this example]
  - $H_d = r_2[x]r_2[y]c_2\ w_1[x]w_1[y]c_1$

Examples for valid execution history

- $T_1 = \{w_1[x], w_1[y], c_1\}, T_2 = \{ r_2[x], r_2[y], c_2 \}$

T_1	T_2
w1(x)	
	r2(x)
w1(y)	
	r2(y)
c1	
	c2

H_a

T_1	T_2
w1(x)	
w1(y)	
c1	
	r2(x)
	r2(y)
	c2

H_b

T_1	T_2
w1(x)	
	r2(x)
	r2(y)
w1(y)	
c1	
	c2

H_c

T_1	T_2
	r2(x)
	r2(y)
	c2
w1(x)	
w1(y)	
c1	

H_d

Check equivalence

- Two operations conflict if:
  1. they belong to **different transactions**,
  2. they operate on the **same object**, and
  3. at least one of the operations is a **write**

2 types of conflicts: (1) Read-Write (or write-read) and (2) Write-Write

- Two histories are (conflict) equivalent if
  1. they are over the same set of transactions, and
  2. the ordering of each pair of conflicting operations is the same in each history

Example

- Consider
  - $H_a = w_1[x]r_2[x]w_1[y]r_2[y]c_1c_2$
  - $H_b = w_1[x]w_1[y]r_2[x]r_2[y]c_1c_2$

Step 1: check if they are over the same set of transactions

- $T_1 = \{w_1[x], w_1[y]\}, T_2 = \{ r_2[x], r_2[y] \}$

Step 2: check if all the conflicting pairs have the same order

Conflicting pairs	H_a	H_b
$w_1[x], r_2[x]$	$<$	$<$
$w_1[y], r_2[y]$	$<$	$<$

Serializable

- Does H_c have an equivalent serial execution?
  - $H_c = w_1[x]r_2[x]r_2[y]w_1[y]c_1c_2$
- Only 2 serial execution to check:
  - H_b: T_1 followed by T_2: $w_1[x]w_1[y]c_1r_2[x]r_2[y]c_2$
    - $r_2[y]$ reads different value as in H_c
  - H_d: T_2 followed by T_1: $r_2[x]r_2[y]c_2w_1[x]w_1[y]c_1$
    - $r_2[x]$ reads different value as in H_c

Conflicting pairs	H_b	H_c	H_d
$w_1[x], r_2[x]$	<	<	>
$w_1[y], r_2[y]$	<	>	>

- Do we need to check all the serial executions?

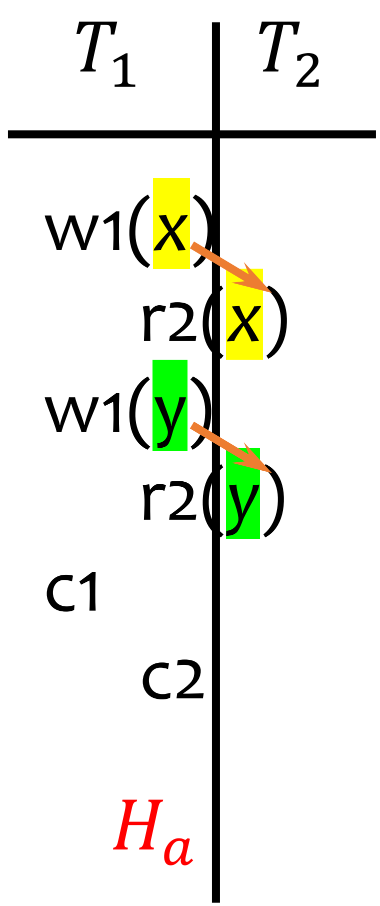
How to test for serializability?

- Serialization graph $SG_H(V, E)$ for history H:
  - $V = \{T | T \text{ is a committed transaction in } H\}$
  - $E = \{T_i \rightarrow T_j \text{ if } o_i \in T_i \text{ and } o_j \in T_j$ **conflict** and $o_i < o_j\}$

- A history is serializable iff its serialization graph is acyclic.

Example

- Example: $H_a = w_1[x] r_2[x] w_1[y] r_2[y]\ \mathrm{c}_1 \mathrm{c}_2$

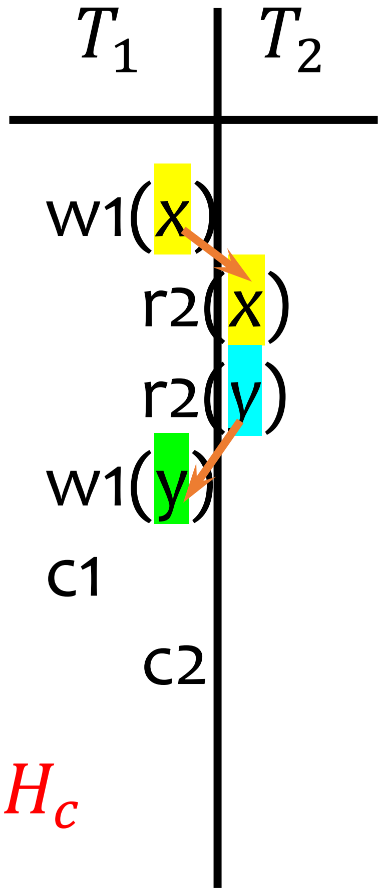


$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$
$w_1[y]$ and $r_2[y]$ conflict, and $w_1[y] < r_2[y]$

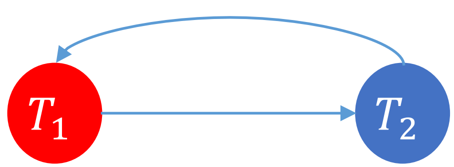
Serialization graph: no cycles → serializable

Example

- Example: $H_c = w_1[x] r_2[x] r_2[y] w_1[y] c_1 c_2$



$w_1[x]$ and $r_2[x]$ conflict, and $w_1[x] < r_2[x]$;
$w_1[y]$ and $r_2[y]$ conflict, and $r_2[y] < w_1[y]$



Not serializable

Locking

- Rules
  - If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
  - If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
  - Allow one exclusive lock, or multiple shared locks

Mode of the lock requested

Mode of lock(s) currently held by other transactions

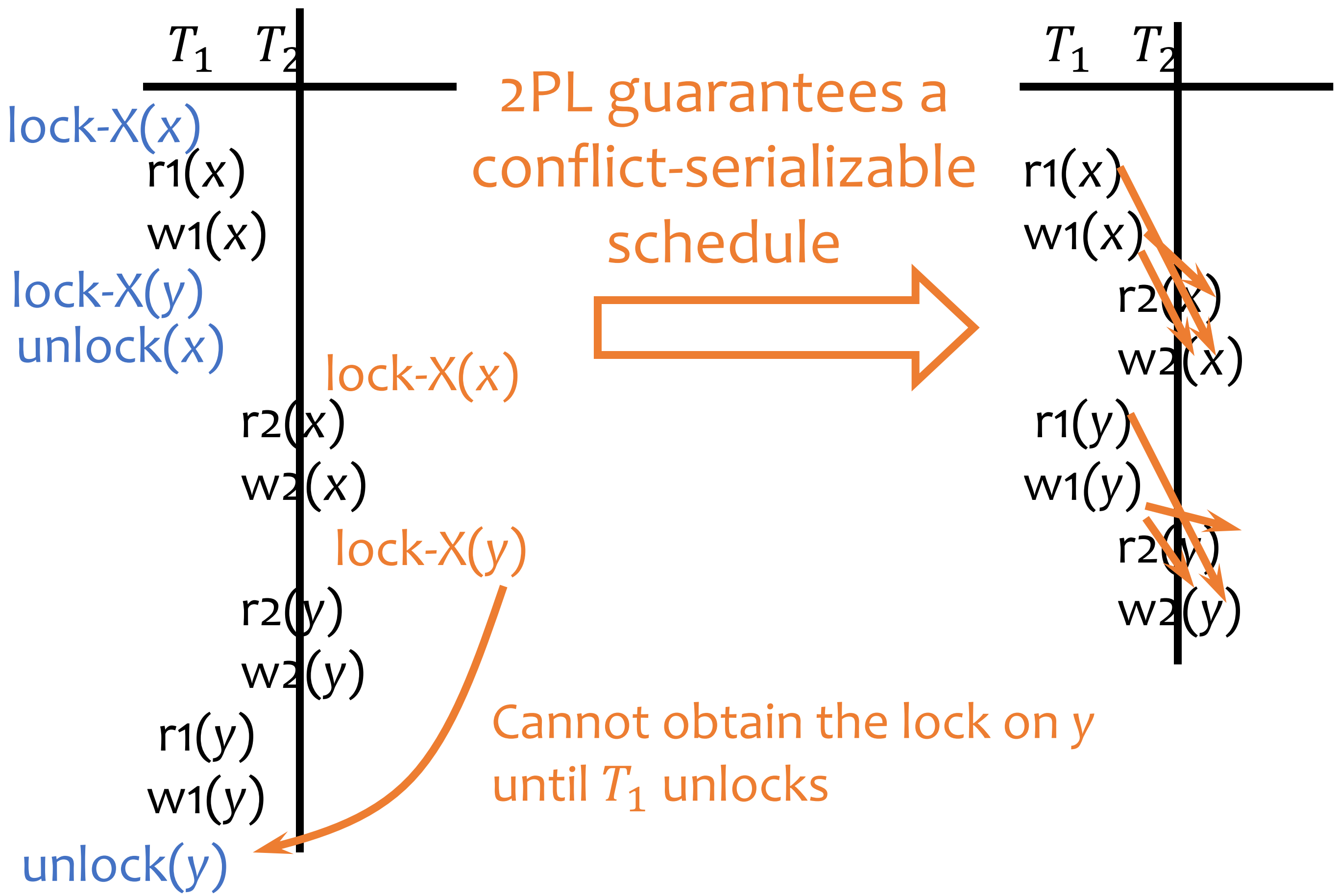
	S	X
S	Yes	No
X	No	No

Grant the lock?

Compatibility matrix

Two-phase locking (2PL)

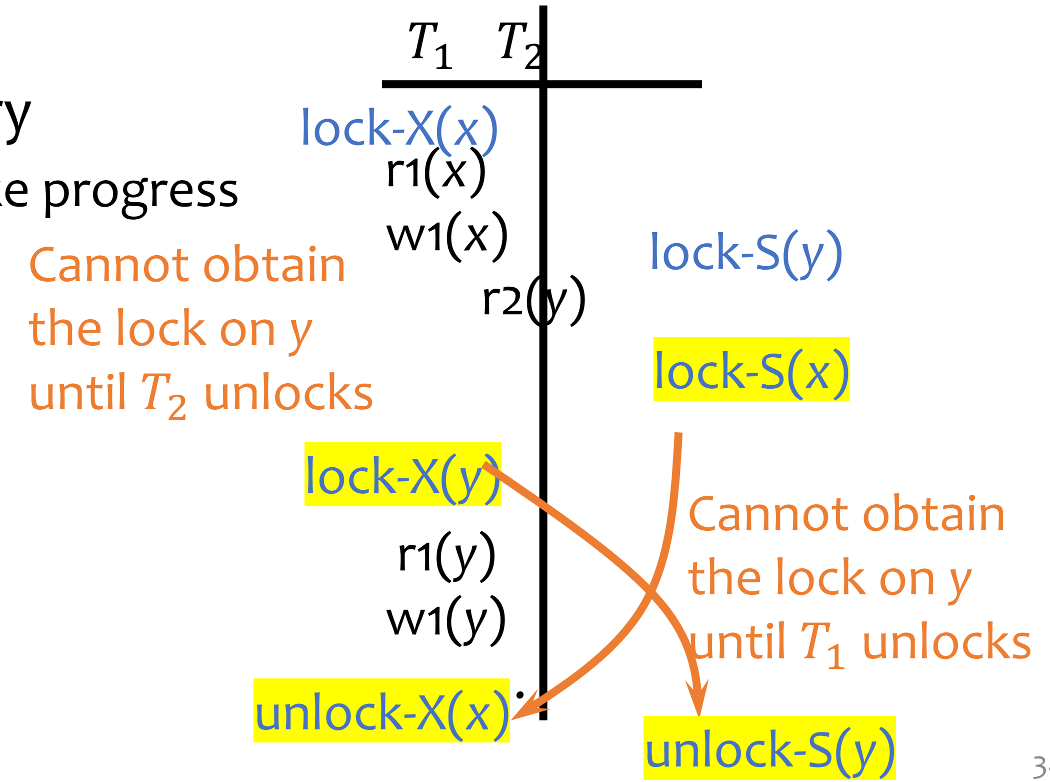
- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks



Deadlocks

- A transaction is deadlocked if it is blocked and will remain blocked until there is an intervention.
- Locking-based concurrency control algorithms may cause deadlocks requiring abort of one of the transactions

- Consider the partial history
  - Neither T_1 nor T_2 can make progress



Strict 2PL

- Only release X-locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts
- Used in many commercial DBMS
  - Avoids cascading aborts
  - But deadlocks are still possible!
- Conservative 2PL: acquire all locks at the beginning of a txn
  - Avoids deadlocks but often not practical

Logging

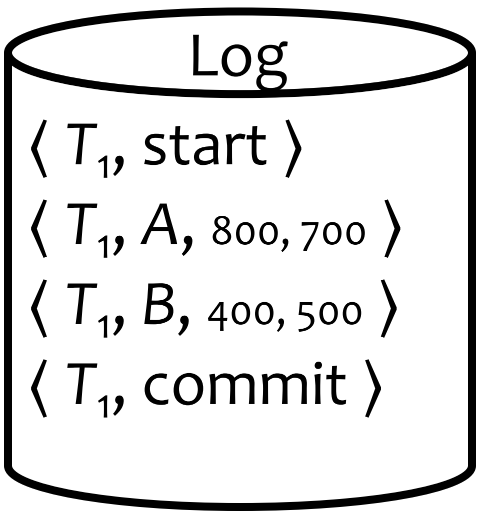
- ACID
  - Atomicity: TX's are either completely done or not done at all
  - Consistency: TX's should leave the database in a consistent state
  - Isolation: TX's must behave as if they are executed in isolation
  - Durability: Effects of committed TX's are resilient against failures
- SQL transactions

```
-- Begins implicitly
SELECT ...;
UPDATE ...;
ROLLBACK | COMMIT;
```

Log format

- When a transaction T_i starts
  - ⟨ T_i, start ⟩

- Record values before and after each modification:
  - ⟨ T_i, *X*, *old_value_of_X*, *new_value_of_X* ⟩
  - T_i is transaction id
  - *X* identifies the data item



- A transaction T_i is committed when its commit log record is written to disk
  - ⟨ T_i, commit ⟩

Log example - redo

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 T2 T3

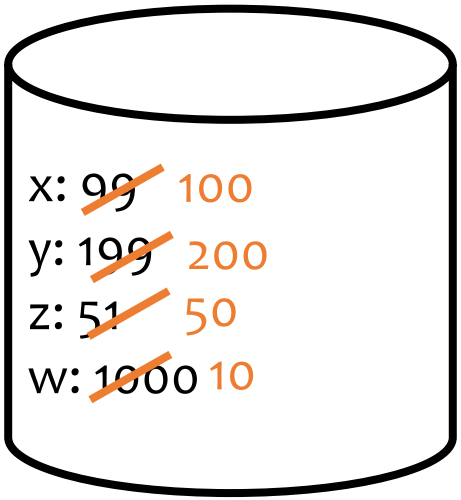
Start of log →

End of log →

	Log
redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
	T_2, commit
	T_4, start
	T_3, z, 51
	T_3, abort
	T_4, y, 200, 50

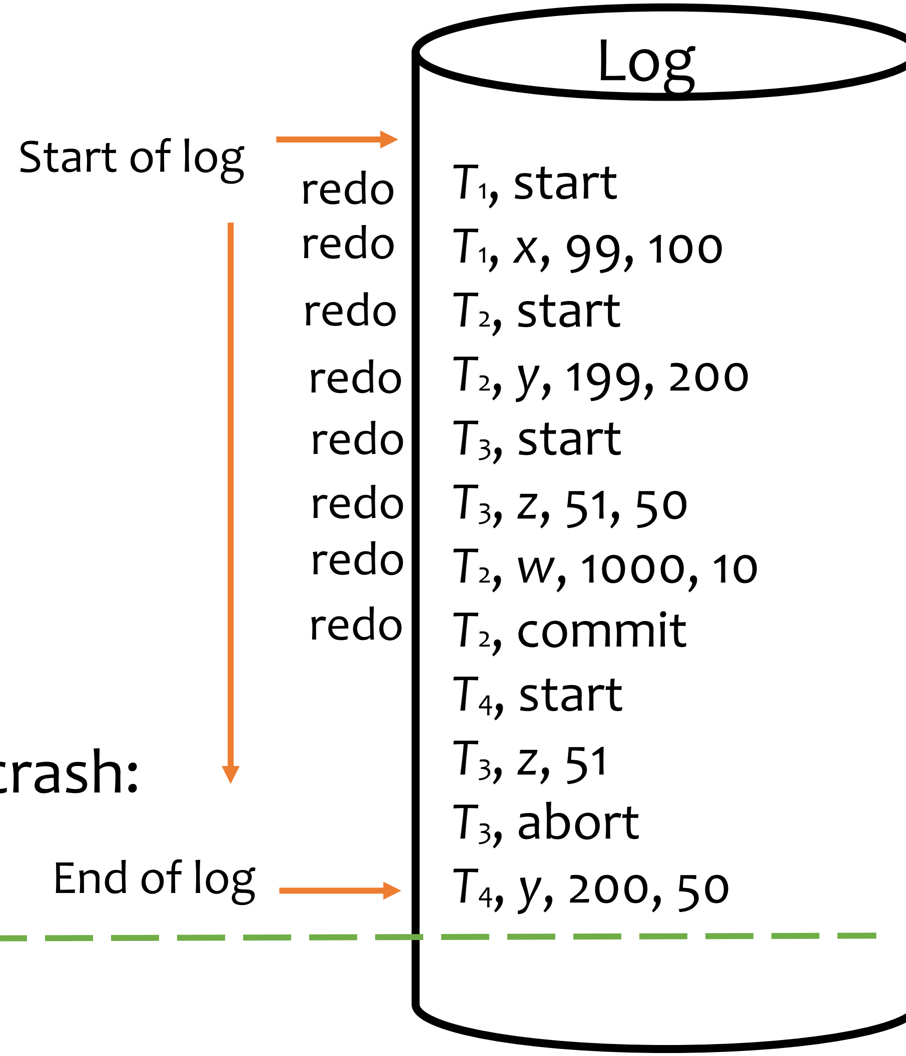
Log example

- Redo phase:



List of active transactions at crash:

T1 ~~T2~~ T3



Log example

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ 200
z: ~~51~~ 50
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ T3 T4

Log

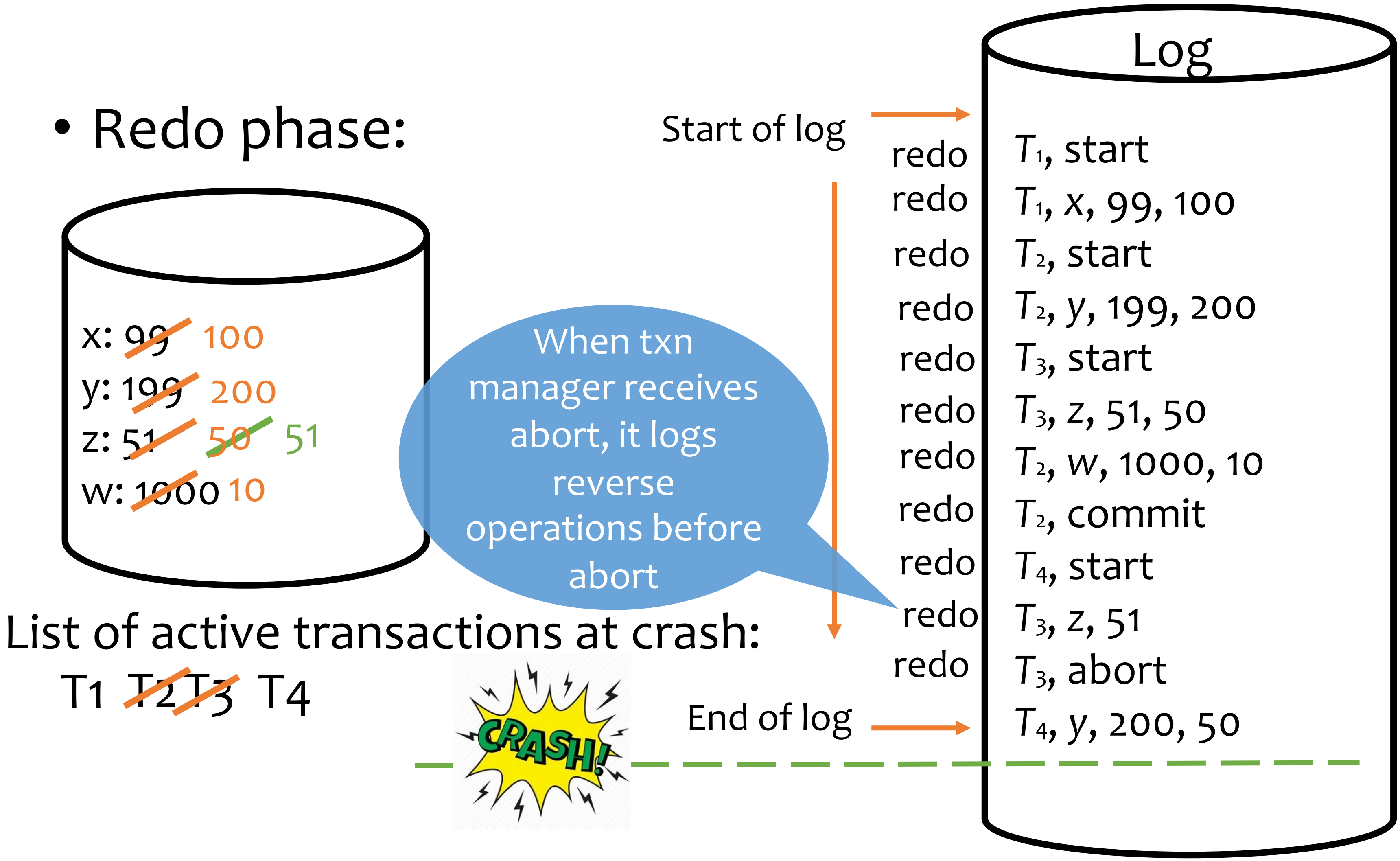
Start of log →

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
	T_3, z, 51
	T_3, abort
	T_4, y, 200, 50

End of log →

Log example

- Redo phase:



Log example

- Redo phase:

x: ~~99~~ 100
y: ~~199~~ ~~200~~ 50
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ ~~T3~~ T4

Log

Start of log →

redo	T_1, start
redo	T_1, x, 99, 100
redo	T_2, start
redo	T_2, y, 199, 200
redo	T_3, start
redo	T_3, z, 51, 50
redo	T_2, w, 1000, 10
redo	T_2, commit
redo	T_4, start
redo	T_3, z, 51
redo	T_3, abort
redo	T_4, y, 200, 50

End of log

Log example - Undo

- Undo phase: T1, T4

x: ~~99~~ ~~100~~ 99
y: ~~199~~ ~~200~~ ~~50~~ 200
z: ~~51~~ ~~50~~ 51
w: ~~1000~~ 10

List of active transactions at crash:
T1 ~~T2~~ ~~T3~~ T4

Start of log →

End of log → undo

Log

* undo	T_1, start
	T_1, x, 99, 100
	T_2, start
	T_2, y, 199, 200
	T_3, start
	T_3, z, 51, 50
	T_2, w, 1000, 10
	T_2, commit
*	T_4, start
	T_3, z, 51
	T_3, abort
undo	T_4, y, 200, 50
	T_4, y, 200
	T_4, abort
	T_1, x, 99
	T_1, abort

Undo/redo logging

- U: used to track the set of active transactions at crash

- Redo phase: scan forward to end of the log
  - For a log record ⟨ *T*, start ⟩, add *T* to *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *new*)
  - For a log record ⟨ *T*, commit | abort ⟩, remove *T* from *U*
    - *If abort, undo changes of T* i.e., add ⟨ *T*, *X*, *old* ⟩ *before logging abort*

  ☞ Basically repeats history!

- Undo phase: scan log backward
  - Undo the effects of transactions in *U*
  - That is, for each log record ⟨ *T*, *X*, *old*, *new* ⟩ where *T* is in *U*, issue write(*X*, *old*), and log this operation too, i.e., add ⟨ *T*, *X*, *old* ⟩
  - Log ⟨ *T*, abort ⟩ when all effects of *T* have been undone

The end!