Transactions 1

CS348 Spring 2023

Instructor: Sujaya Maiyya

Sections: **002 & 004 only**

Announcements

- Milestone 2
  - Due today!

- Assignment 3
  - Due Thursday, July 20th

Outline For Today

1. Motivation For Transactions
2. ACID Properties
3. Different Levels of Isolation Beyond Serializability

User's Perspective

Serializability:

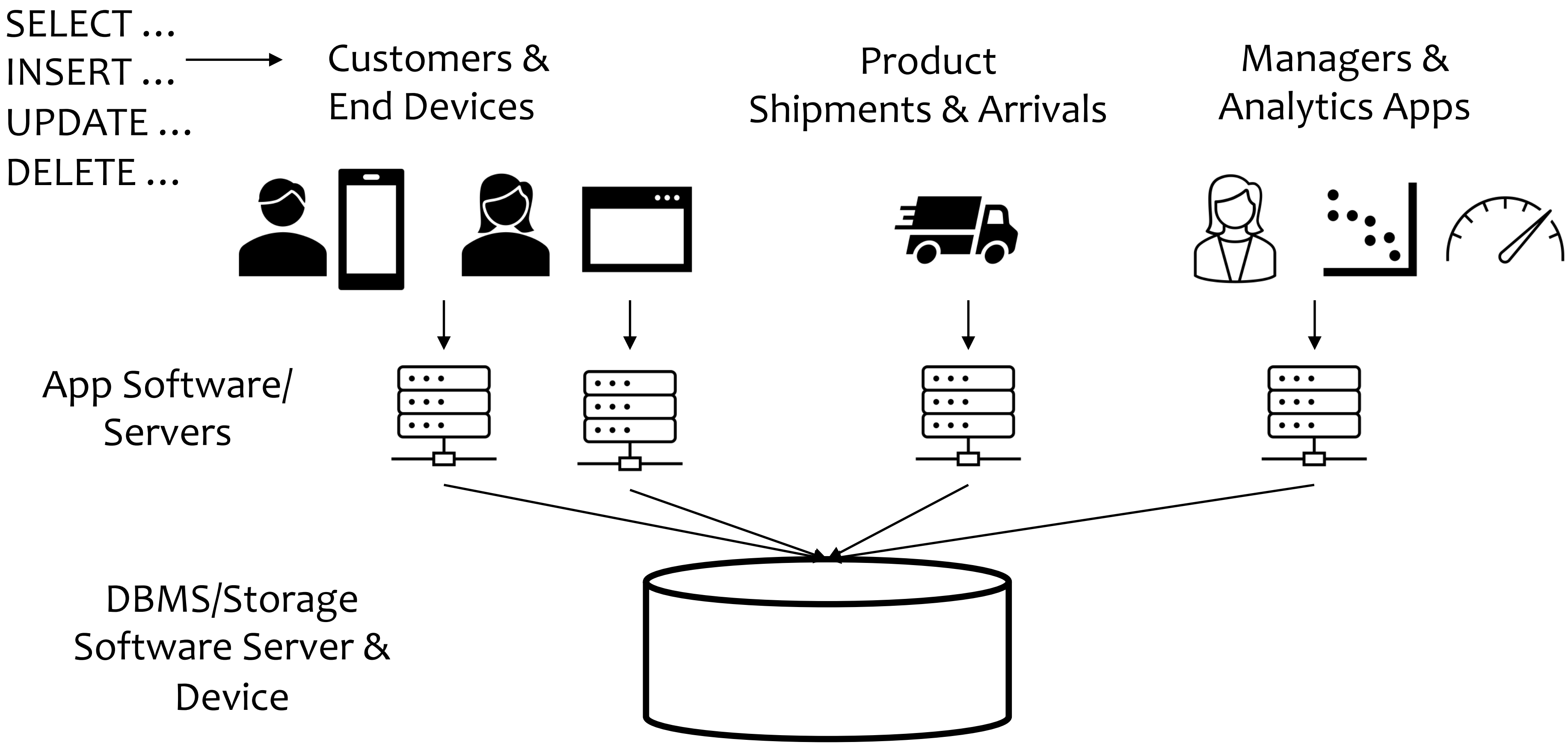
- Execution Histories
- Conflict Equivalence
- Checking For Conflict Equivalence

System's Perspective
(and more next 2
lectures)

Thanks to Prof. Semih Salihoglu for the slides

Recall example for Lecture 1

- Ex Application: Order & Inventory Management in E-commerce



SELECT …
INSERT …
UPDATE …
DELETE …

Customers & End Devices

Product Shipments & Arrivals

Managers & Analytics Apps

App Software/ Servers

DBMS/Storage Software Server & Device

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

Example Problems With Concurrency (1)

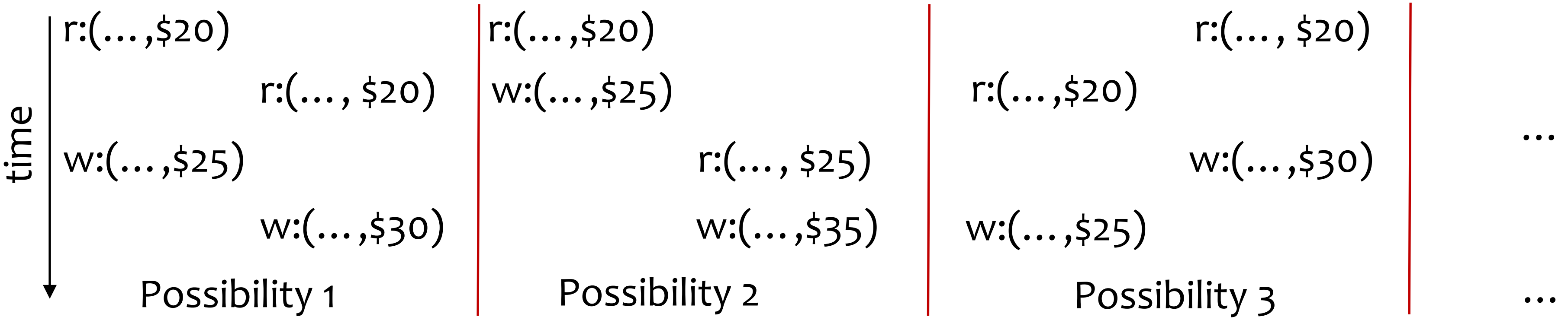
- Read-only queries are simple to execute concurrently.
- Ex: Two clients concurrently update the same relation in DBMS

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

```
UPDATE Order
SET price = price + 10
WHERE oid = o1
```

Order			
o1	bust1	bookA	$20
…	…	…	…

- Possible attribute-level inconsistency in absence of safe concurrency:



Example Problems With Concurrency (2)

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

```
UPDATE Order
SET pID = WatchA
WHERE oid = o1
```

Order			
o1	cust1	BookA	$20
…	…	…	…

- Possible Tuple-level inconsistency

o1	cust1	BookA	$25

o1	cust1	WatchA	$20

o1	cust1	WatchA	$25

Example Problems With Concurrency (3)

Update Statement 1:

```
UPDATE Customer
SET membership = Gold
WHERE cid IN (Select cid FROM Orders
              WHERE price >= 20)
```

Update Statement 2:

```
UPDATE Order
SET price = price*0.9
WHERE pid = BookA
```

Customer		
cid	name	membership
cust1	Alice	Silver
…	…	…

Order			
oid	**cid**	**pid**	**price**
01	cust1	BookA	$20
…	…	…	…

- Possible Relation-level inconsistency
- Statement 1’s update on Customer depends on Order table, which is concurrently being updated.
- Data in Customer can be corrupted if the executions overlaps.

Example Problems With Concurrency (4)

```
Client 1
INSERT INTO 2021_Orders
SELECT * FROM Orders WHERE year = 2021

DELETE FROM Orders WHERE year = 2021
```

```
CLIENT 2:
SELECT Count(*) FROM Orders
SELECT Count(*) FROM 2021_Orders
```

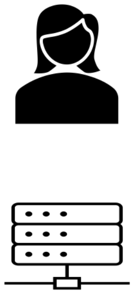
- Possible Database-level inconsistency
- Expectation: Total # orders in the enterprise (across Orders and 2021_Orders) remains unchanged.
- But Client 2 can see an inconsistent number of order counts across both databases depending on how much of the data from Orders has been moved to 2021_Orders and also deleted.

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.

Problems due to failures (Slides From Lecture 1)

- What if your disk fails in the middle of an order?
- What if your server software fails due to a bug?
- What if there is a power outage in the machine storing files?
- Suppose Alice orders both BookA and BookB



w (A, 0)

Product	NumInStock
…	…
BookA	1
BookB	7

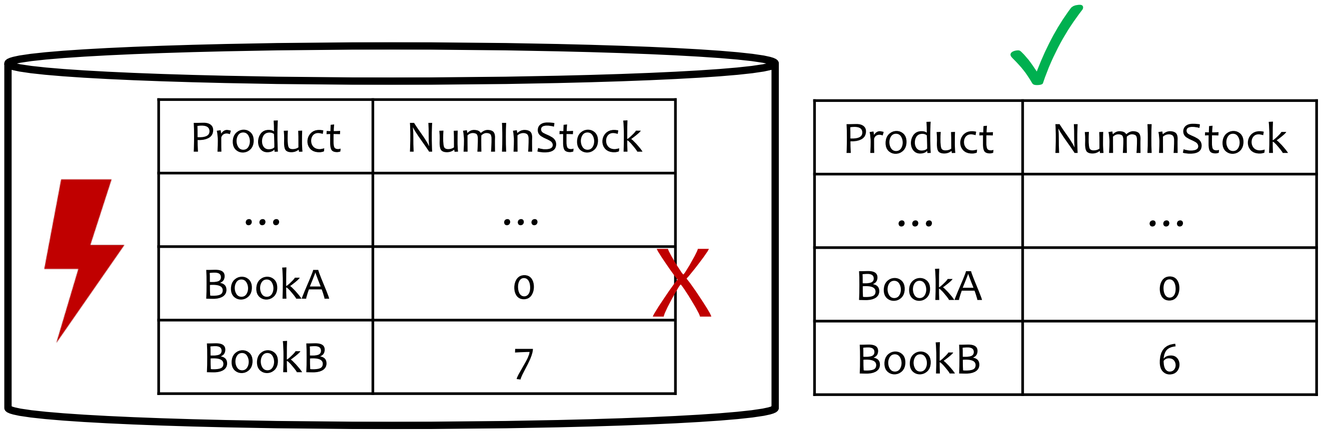
Problems due to failures (Slides From Lecture 1)

- What if your disk fails in the middle of an order?
- What if your server software fails due to a bug?
- What if there is a power outage in the machine storing files?
- Suppose Alice orders both BookA and BookB

Before (B, 6) is written, there is a crash!
Inconsistent data state!

PR: What happens when the system is back up?
How to recover from inconsistent state?

w (A, 0)



Product	NumInStock
…	…
BookA	0
BookB	7

Product	NumInStock
…	…
BookA	0
BookB	6

Case For Atomicity To Handle Failures

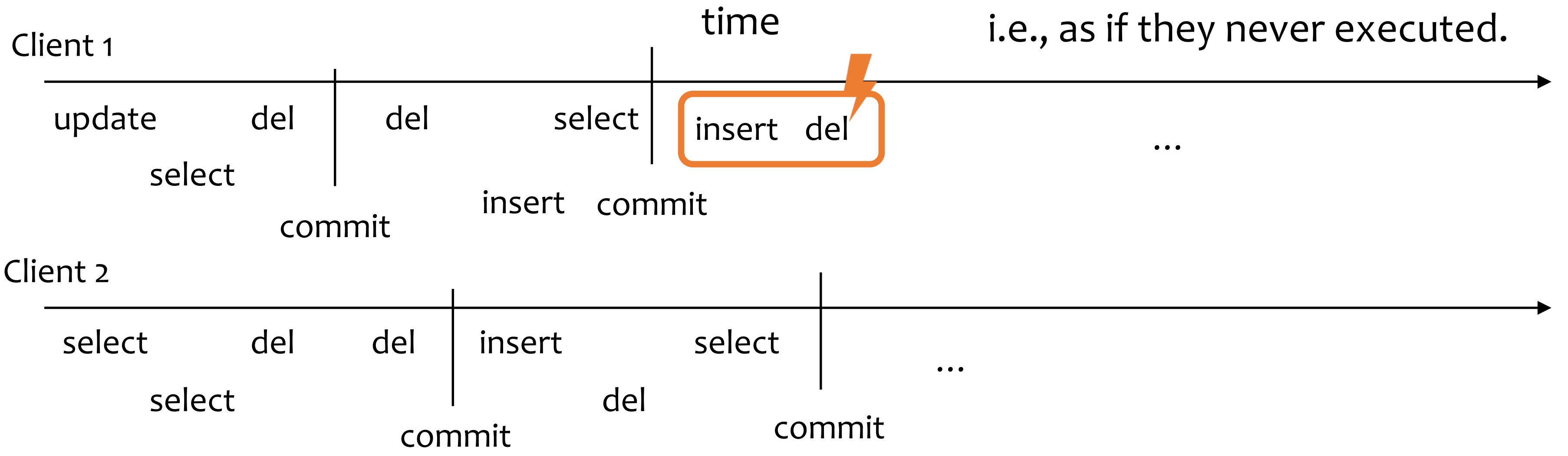
- All or nothing guarantee: Run the set of statements only if the DBMS can guarantee that they *will all succeed and be persistent or all will fail and no update they make will be persistent.*

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)

Transactions in SQL

- In SQL Standard, transactions begin when a client issues a "Begin Transaction" command & ends with the "commit" or "rollback" keyword.
- Autocommit: treats each statement as a separate transaction



Will not have any effects on the database, i.e., as if they never executed.

If client statement and operations really run concurrently and overlap: What guarantees can a DBMS really give with transactions?

Outline For Today

1. Motivation For Transactions
2. ACID Properties
3. Different Levels of Isolation Beyond Serializability

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

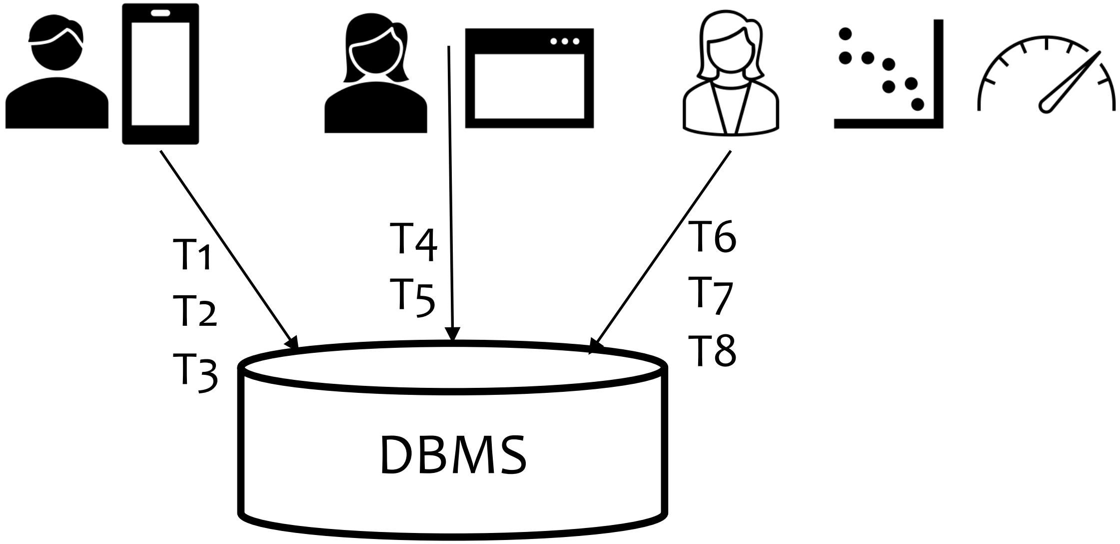
- Partial effects of a transaction must be undone when
  - User explicitly aborts the transaction using ROLLBACK
  - The DBMS crashes before a transaction commits

- 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

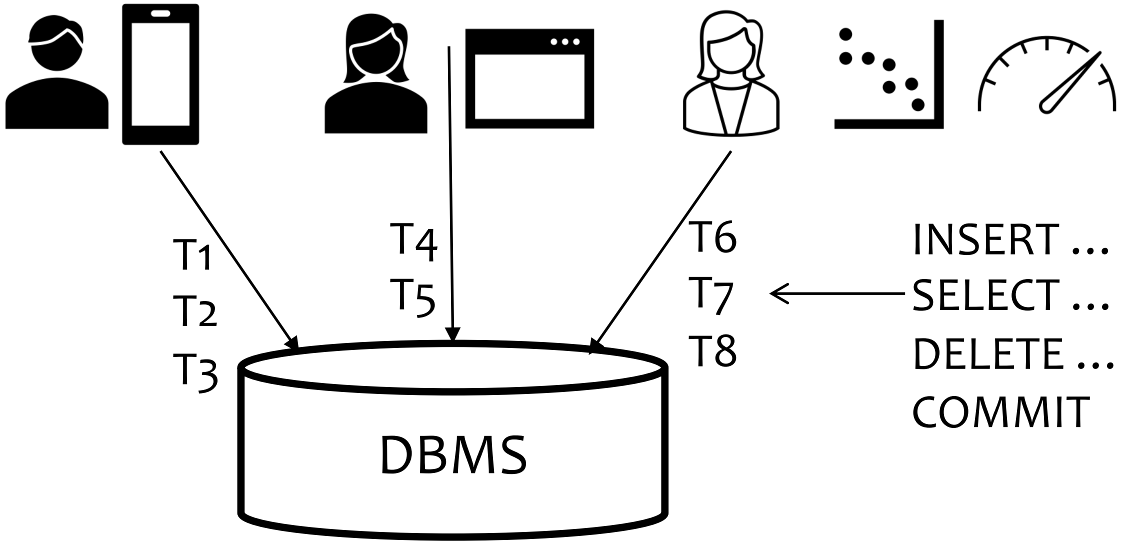
ACID: Consistency



- 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)

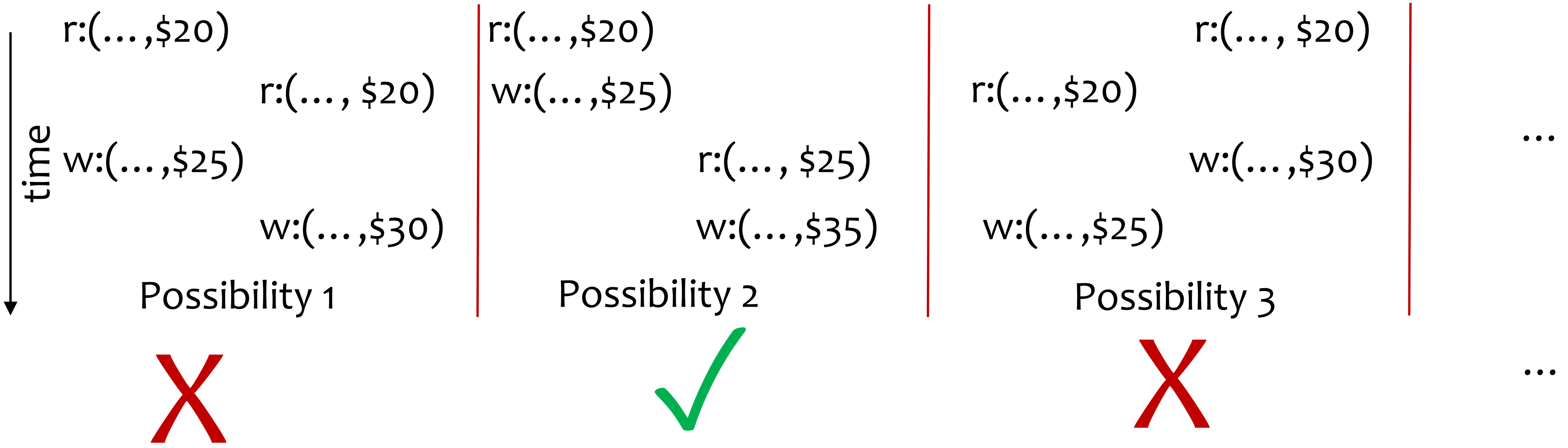
Recall Example Problems With Concurrency (1)

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

Txn 2:
UPDATE Order
SET price = price + 10
WHERE oid = o1

Order			
o1	bust1	bookA	$20

- ➢ Attribute-level inconsistency In absence of safe concurrency



r:(…,$20)
r:(…, $20)
w:(…,$25)
w:(…,$30)

Possibility 1

X

r:(…,$20)
w:(…,$25)
r:(…, $25)
w:(…,$35)

Possibility 2

✓

r:(…, $20)
r:(…,$20)
w:(…,$30)
w:(…,$25)

Possibility 3

X

…

…

Two possibilities now: T1; T2 (e.g possibility 2)
or T2; T1 (not shown in figure but also leading to $35)

Recall Example Problems With Concurrency (2)

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

Txn 2:
UPDATE Order
SET pID = WatchA
WHERE oid = o1

Order			
o1	cust1	BookA	$20

- Possible Tuple-level inconsistency

o1	cust1	BookA	$25

✗

o1	cust1	WatchA	$20

✗

o1	cust1	WatchA	$25

✓

Two possibilities again: T1; T2 or T2; T1 (both leading to possibility 3)

Recall Example Problems With Concurrency (3)

Txn 1:
Update Statement 1:
UPDATE Customer
SET membership = Gold
WHERE cid IN (Select cid FROM Orders
WHERE price >= 20)

Txn 2:
Update Statement 2:
UPDATE Order
SET price = price*0.9
WHERE pid = BookA

- Possible Relation-level inconsistency

Customer		
cid	name	membership
cust1	Alice	Silver
…	…	…

Order			
oid	**cid**	**pid**	**price**
o1	cust1	BookA	$20
…	…	…	…

Two possibilities again: T1; T2 or T2; T1
Interestingly order now matters unlike Examples 1 & 2 previously.
E.g., suppose Alice has only 1 order:
If order is T1; T2: she becomes a Gold member
If it is T2; T1: she remains a Silver member.

Recall Example Problems With Concurrency (4)

```
Txn 1:
INSERT INTO 2021_Orders
SELECT * FROM Orders WHERE year = 2021

DELETE FROM Orders WHERE year = 2021
```

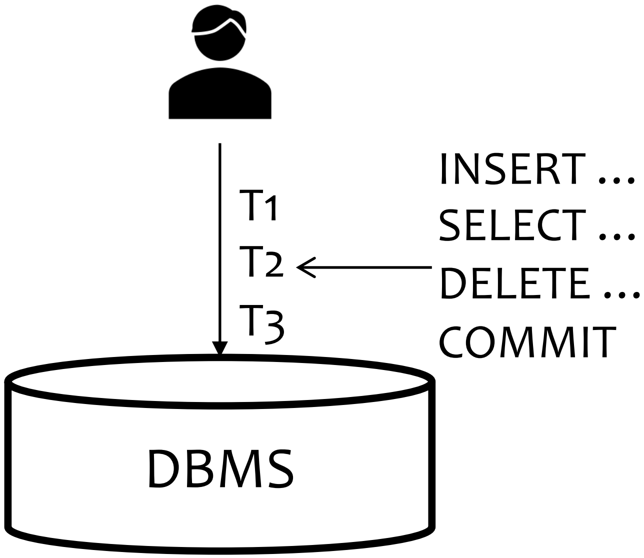
```
Txn 2:
SELECT Count(*) FROM Orders
SELECT Count(*) FROM 2021_Orders
```

- Possible Database-level inconsistency
- 2 count queries are now guaranteed to see a consistent state of the database records (though there are 2 possible “consistent” outputs)

  If T1; T2 => All 2021 records counted once in 2021_Orders

  If T2; T1 => All 2021 records counted once in Order

ACID: Durability



T1
T2
T3

INSERT …
SELECT …
DELETE …
COMMIT

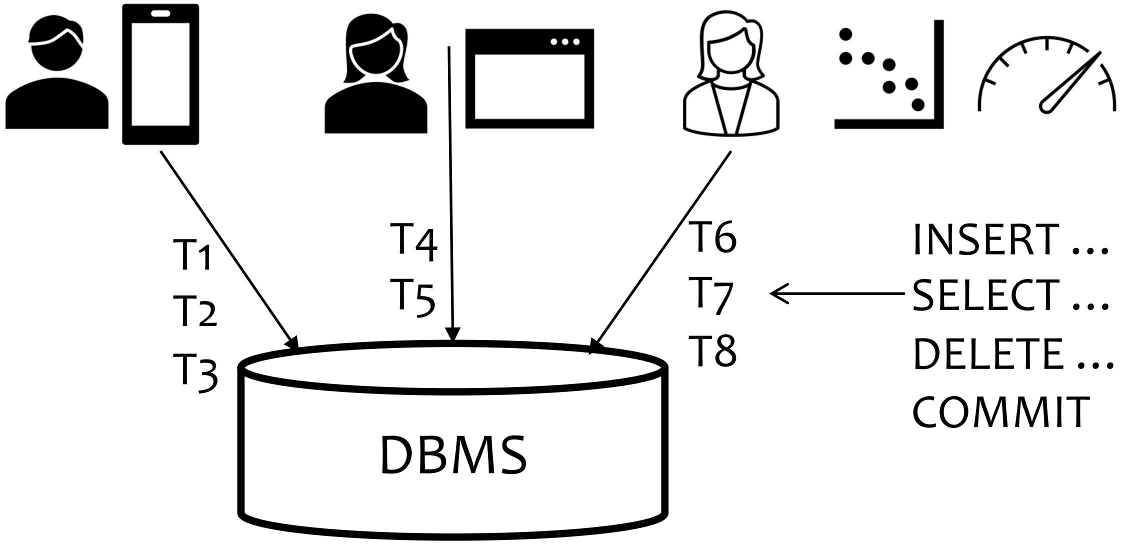
DBMS

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

Outline For Today

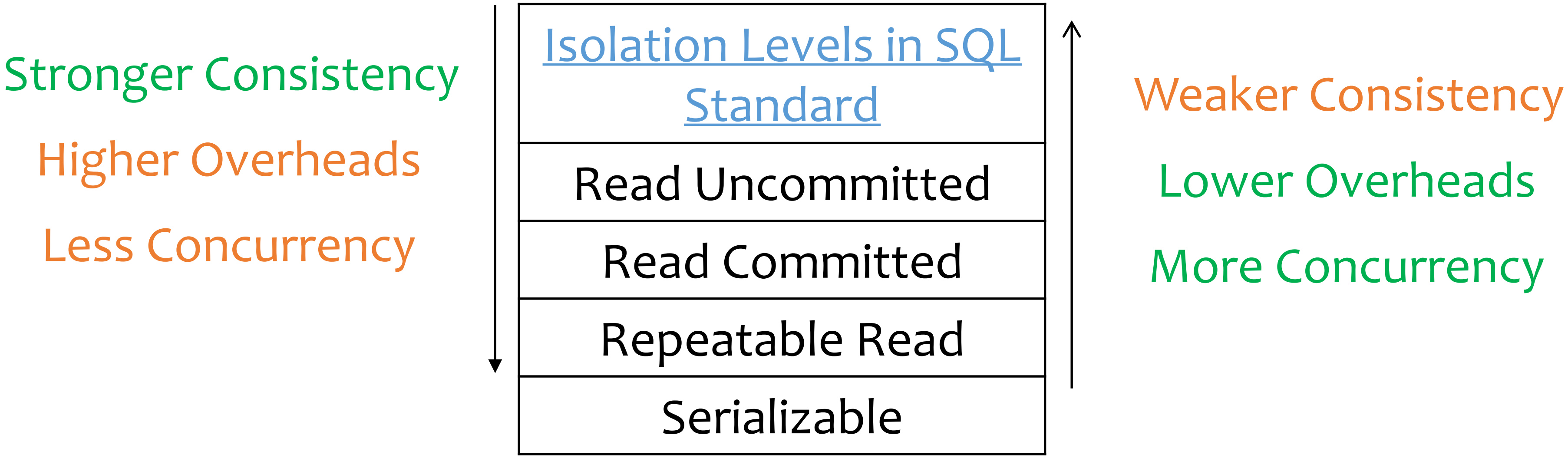
1. Motivation For Transactions
2. ACID Properties
3. Different Levels of Isolation Beyond Serializability

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 = 01

SELECT price FROM Order
WHERE oid = 01;

COMMIT
```

Suppose sets 20->25

Will read 25 (not considered a dirty read)

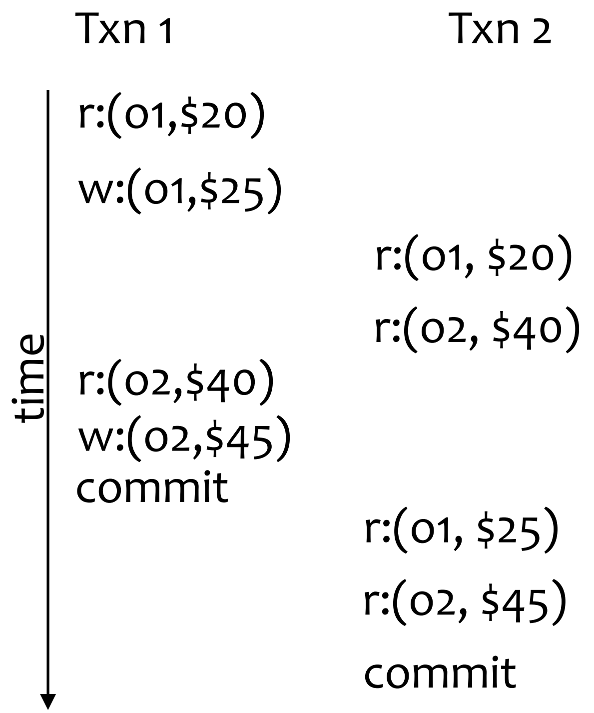
READ COMMITTED

- 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

- 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
```

time

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

- 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

Summary

1. Motivation For Transactions
2. ACID Properties
3. Different Levels of Isolation Beyond Serializability

User's Perspective

Serializability:

- Execution Histories
- Conflict Equivalence
- Checking For Conflict Equivalence

System's Perspective
(and more next 2
lectures)