CS 348 Lecture 17

Transactions I

Semih Salihoğlu

Nov 16, 2021

Announcements

- A4 due on Nov 19th midnight
- A5 will be released on Nov 19th

CS 348 Diagram

User/Administrator Perspective

Primary Database Management System Features

- Data Model: Relational Model
- High Level Query Language: Relational Algebra & SQL
- Integrity Constraints
- Indexes/Views
- Transactions

Relational Database Design

- E/R Models
- Normal Forms

How To Program A DBMS (0.5-1 lecture)

- Embedded vs Dynamic SQL
- Frameworks

DBMS Architect/Implementer Perspective

- Physical Record Design
- Query Planning and Optimization
- Indexes
- Transactions

Other (Last 1/2 Lectures)

- Graph DBMSs
- MapReduce: Distributed Data Processing Systems

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)

Motivation For Transactions

- Transactions: one or more dbms operations that appear as a unit

```
Select …;
Insert …;
Update Customer
Set GoldMember = True
WHERE CID IN (SELECT … );
Delete …;
```

Each can succeed or fail independently (e.g, if the insert succeeds and inserts 10 tuples, but update fails, those 10 records will be persisted in the db)

Motivation For Transactions

- Transactions: one or more dbms operations that appear as a unit

```
Begin Trx;
Select …;
Insert …;
Update Customer
Set GoldMember = True
WHERE CID IN (SELECT … );
Delete …;
Commit;
```

All of the operations is 1 unit of work.
So they fail or succeed together.
E.g., if update fails now, it is as if none of the operations in the trx executed, so the 10 tuples will not be persistent.

- As such, transactions are the solution to:

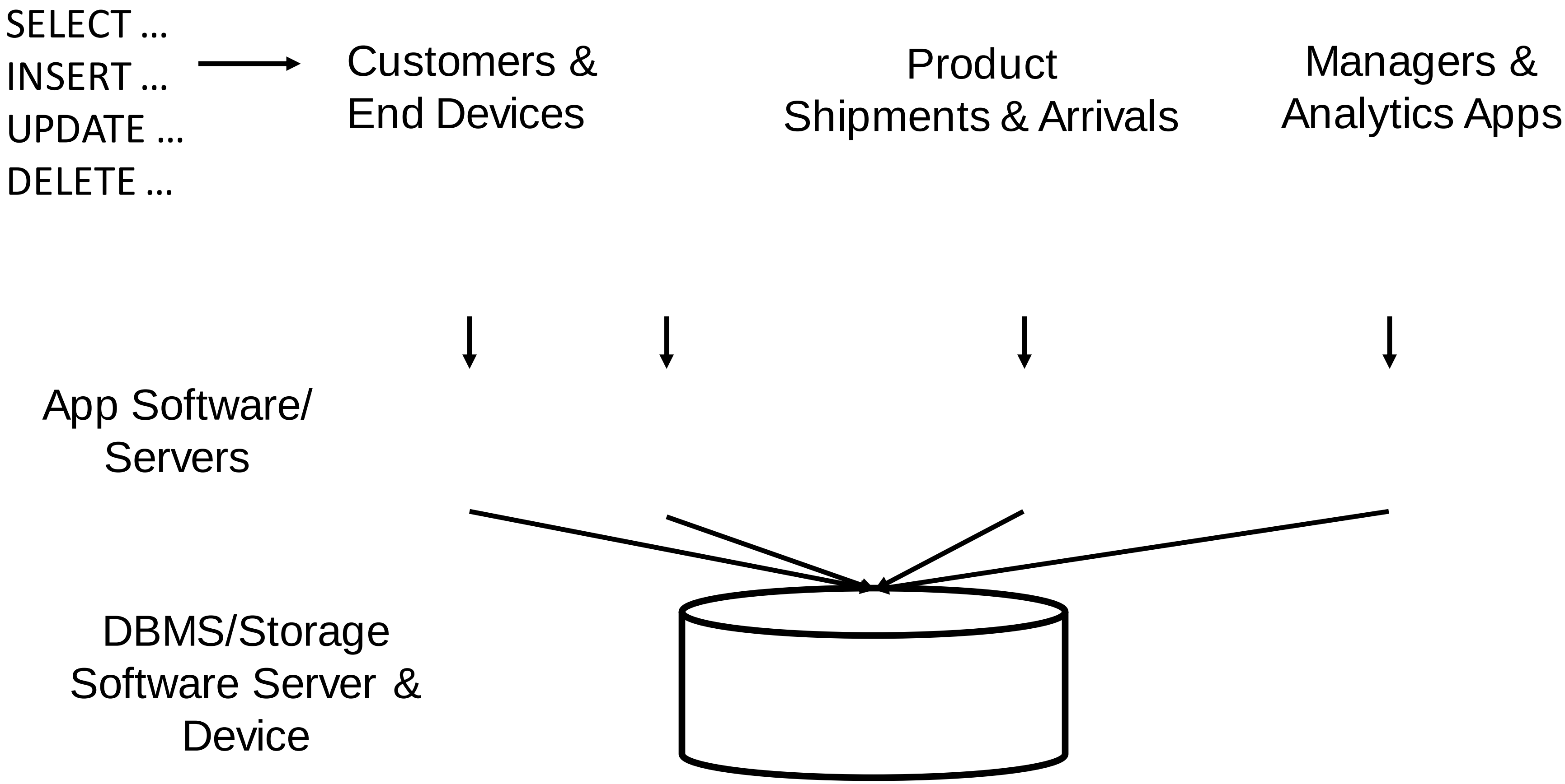
  1. Resilience to system failures

- Independently, they are also the solution to:

  2. Safe concurrent access to the DBMS (Isolation)

[redacted]current Access to the DBMS

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



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
…	…	…	…

- Query processor will read; modify; write same attrib./row/page twice
- Attribute-level inconsistency. In absence of safe concurrency:

time						
r:(…,$20)		r:(…,$20)			r:(…, $20)	
	r:(…, $20)	w:(…,$25)		r:(…,$20)		
w:(…,$25)			r:(…, $25)		w:(…,$30)	…
	w:(…,$30)		w:(…,$35)	w:(…,$25)		
Possibility 1		Possibility 2		Possibility 3		…

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**
o1	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: Client 1's statements is *not meant* to change the total # orders in the enterprise (across Orders and 2021_Orders).
- 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.

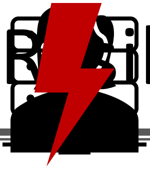
Resilience to System Failures (Slides From Lecture 1)

- 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

Resilience to System 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

X

✓

Product	NumInStock
...	...
BookA	0
BookB	6

Case For Atomicity For Resilience To Failures

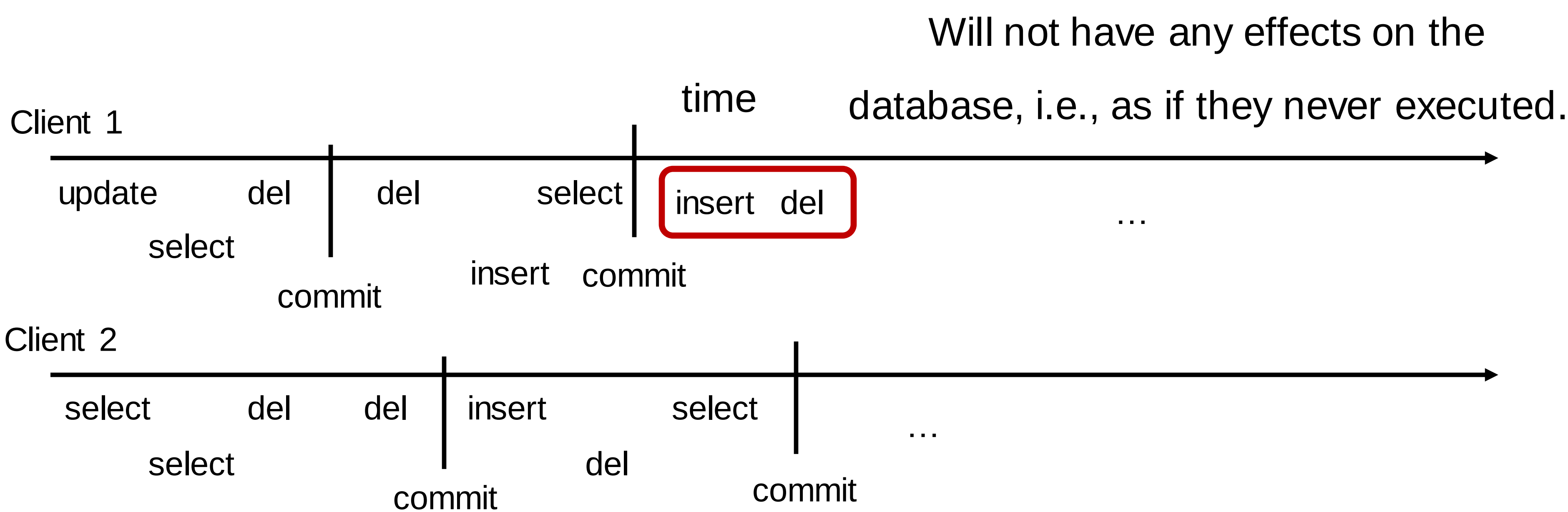
- All or nothing guarantee: Run the set of statements only if the DBMS can guarantee that they *will all succeed and be persistent or all will fail and no update they make will be persistent.*

Transactions: Solution to Concurrency & Resilience

- Transactions are the mechanism for both problems: 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 submits a statement or issues a “Begin Transaction” command & ends with the “commit” keyword.
- Autocommit: treats each statement as a separate transaction



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*:

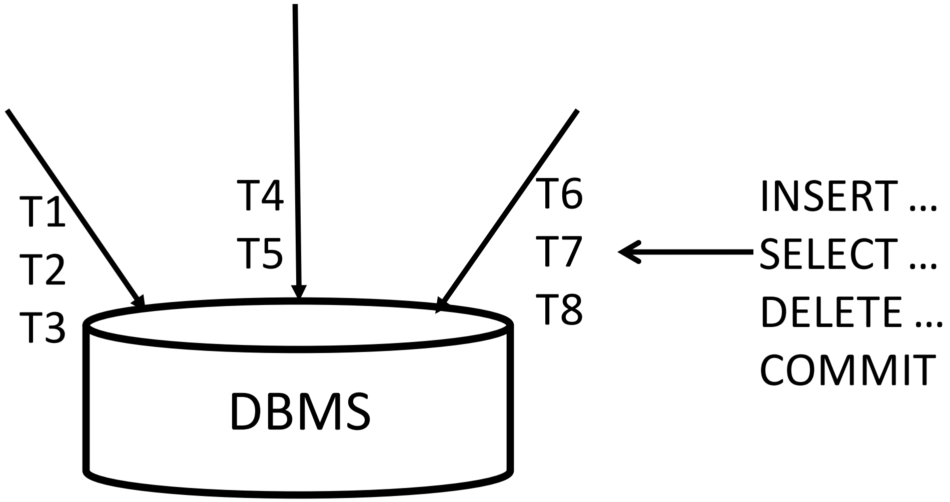
  A: Atomicity

  C: Consistency

  I: Isolation

  D: Durability

[redacted]D: Isolation



- 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)

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

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

Order			
o1	bust1	bookA	$20

- Attribute-level inconsistency In absence of safe concurrency

time

r:(…,$20)		r:(…,$20)			r:(…, $20)	
	r:(…, $20)	w:(…,$25)		r:(…,$20)		
w:(…,$25)			r:(…, $25)		w:(…,$30)	…
	w:(…,$30)		w:(…,$35)	w:(…,$25)		
Possibility 1		Possibility 2		Possibility 3		
						…

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)

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

```
Trx 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)

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

```
Trx 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)

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

DELETE FROM Orders WHERE year = 2021
```

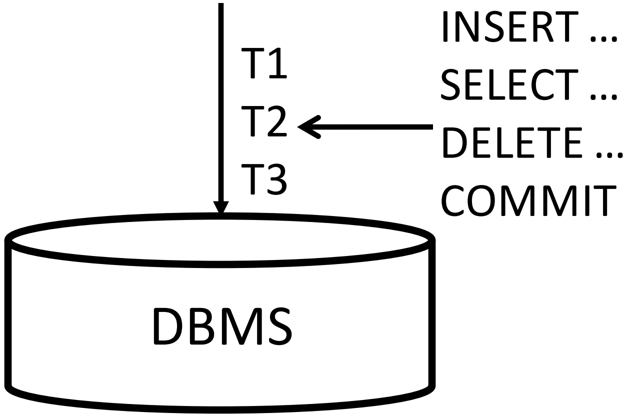
```
Trx 2:
SELECT Count(*) FROM Orders
SELECT Count(*) FROM 2021_Orders
```

- 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

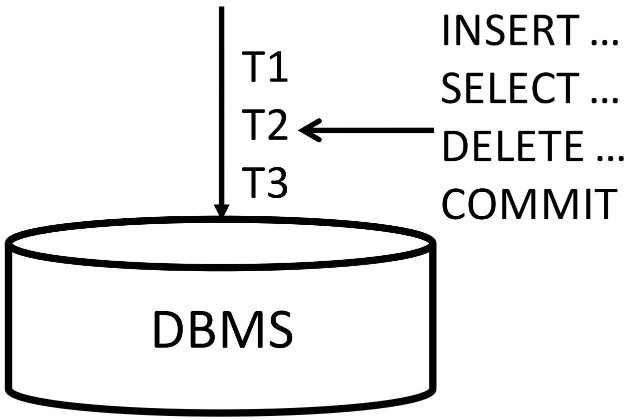
  If T2; T1 => All 2021 records counted once in Order

ACID: Durability



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

ACID: Atomicity



- Atomicity: Handles guarantees for *crashes before commit*
  - Guarantee: none of the modifications will persist
- Question to ponder: How can a DBMS guarantee atomicity?
- Also through logging (Lecture 19).
- Partial changes are undone/rolled back upon system coming back.

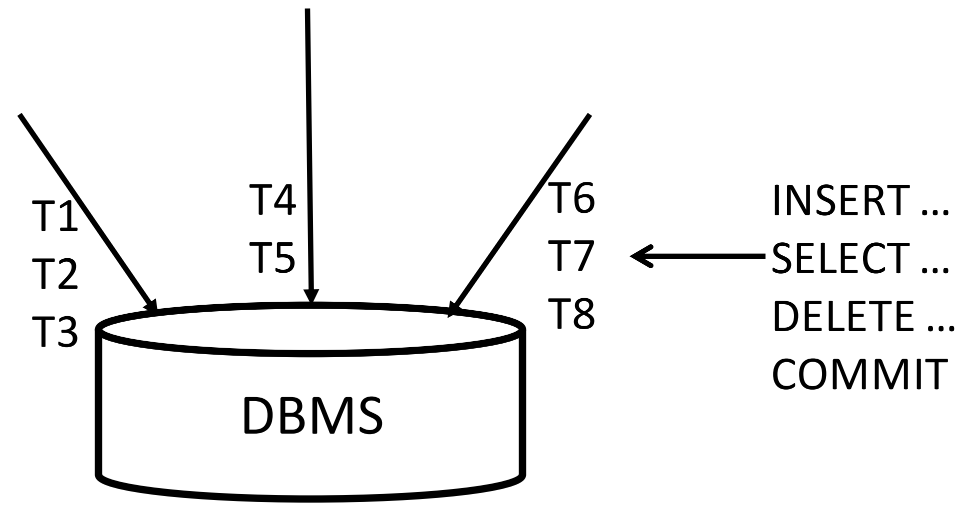
Rolling Back Transactions

- Mechanism to undo any effects of modifications between:
  - Transaction begin and crash (and of course before commit)
- Importantly: Can also be manually triggered by applications

```
BEGIN TRANSACTION;
// Display some information and get input from user
SELECT …
// "Temporarily" execute user's preferred action
SELECT …; INSERT …; UPDATE …;
// Ask user for confirmation after showing the result of action
If ans = "OK" THEN COMMIT ELSE ROLLBACK
```

- Extremely useful. Very difficult to implement such "preview results"->confirm->proceed-or-cancel logic directly in the application.
- Warning: Long begin-and-commit periods decreases the chance of successful commit and can lock portions of the db to other trxs.

[redacted]D: Consistency



- Consistency: *If* application is written in a way that:

  Each transaction if ran in isolation keeps integrity constraints intact

  Then when transactions are ran concurrently, all integrity constraints must remain intact after they complete.

Serializability guarantees consistency but only if app is written correctly.

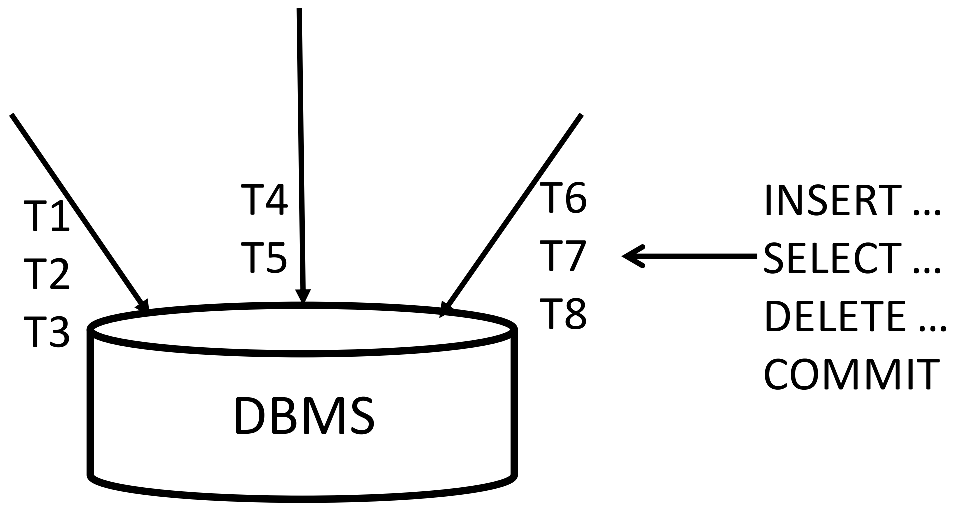
Ex Serial order: | T1 | T7 | T8 | T4 …

holds holds holds …

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

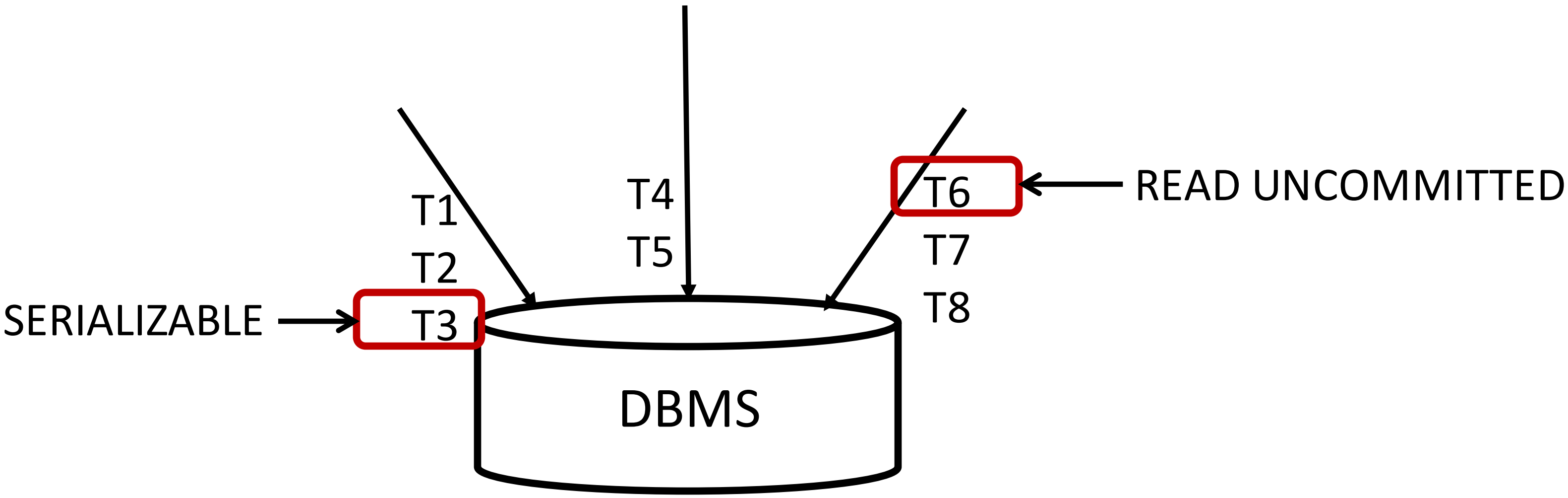
Lower Overheads

More Concurrency

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

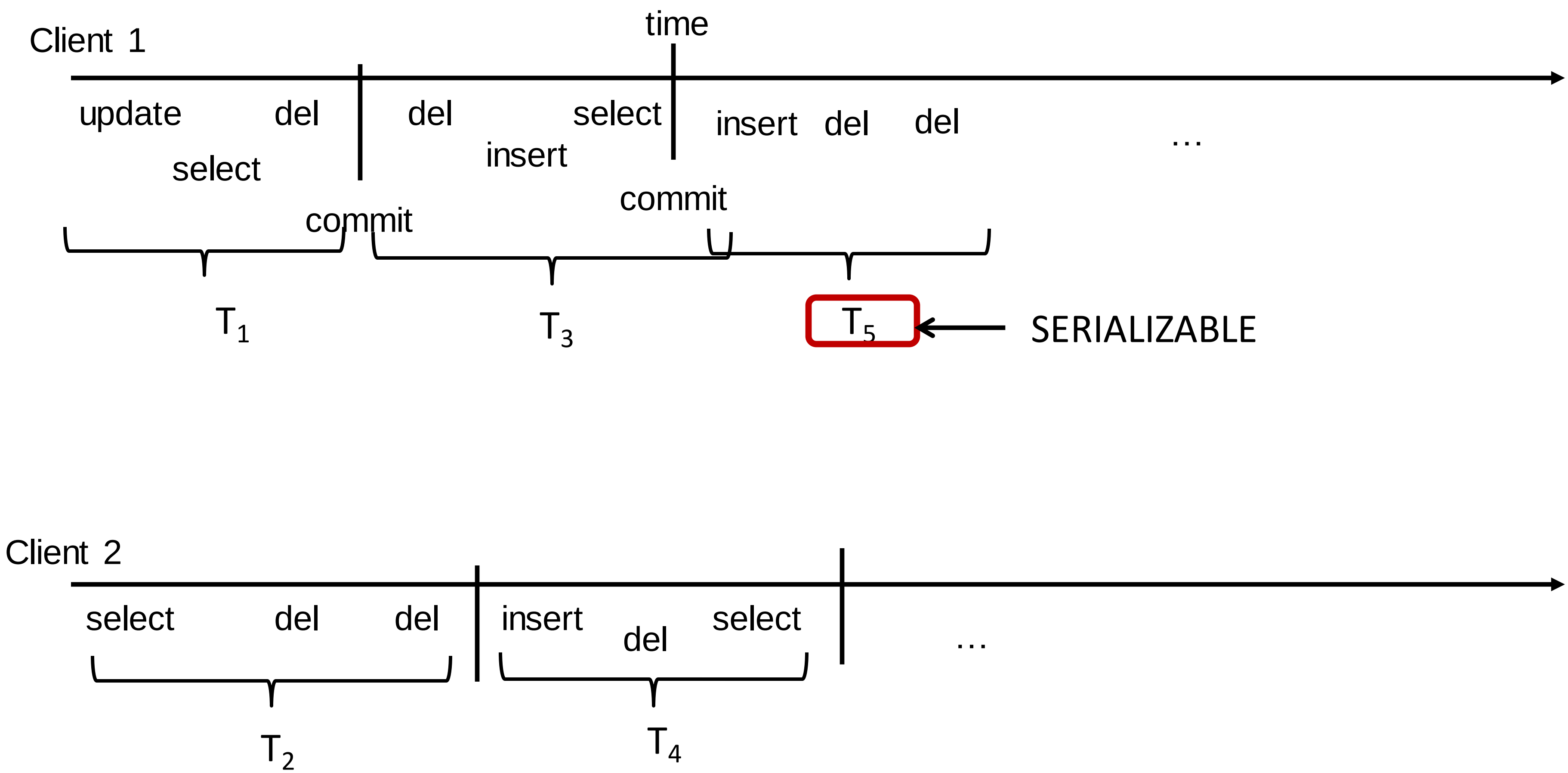
[redacted]rtant Note On Per-transaction Isolation Levels

- Isolations levels are *per transaction!*



- Q: What? We just defined serializability per a set of transactions **T**.
- What does it mean for T_j to be serializable/read uncommitted etc?
- A1: T_j gets a specific *guarantee for properties of its read operations!*
- A2: For serializability specifically: Unless all trxs set serializability, state of the db at time t_i is not necessarily equivalent to some serial order of trxs committed until t_i.

Example



Client 1
time
update
select
del
commit
del
insert
select
commit
insert
del
del
...
T_1
T_3
T_5
SERIALIZABLE
Client 2
select
del
del
insert
del
select
...
T_2
T_4

Example Continued

- If T_j is set to serializable the guarantee is the following:
- Some set of trxs **T**, e.g., {T1, T3, T2}, will be committed before T_j, and left the db in a state, let's call $D_{<j}$
- $D_{<j}$ is not necessarily a state after some serial exec. of **T**
- Let D_j be the state T_j leaves the db in after execution.

Guarantee: D_j = state T_j would leave $D_{<j}$ in if it were the only transaction running on $D_{<j}$.

- Equivalent to previous dfn of serializability if all trxs are serializable: i.e., final db state is the output of some serial order of trxs.

READ UNCOMMITTED

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

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

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

time	Trx 1	Trx 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 trx!
- Every (uncommitted) trx will read values it has written.
- That is not considered “dirty” even if it comes from uncommitted trx.

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*.

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

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

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

Trx 1	Trx 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*

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

INSERT INTO Order VALUES (o3, 10)
```

```
Trx 2: (REPEATABLE READ)
SELECT sum(price) FROM Order

SELECT sum(price) FROM Order
```

Trx 1	Trx 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

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

- No dirty reads; every read is repeatable; no scan of any relation can be phantom
- Recall the guarantee for a trx T_j that is set to serializable:

Guarantee: D_j = state T_j would leave $D_{<j}$ in if it were the only transaction running on $D_{<j}$.

- Note running T_j without concurrency cannot introduce phantoms.

Summary of Isolation Levels

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

Summary of Isolation Levels

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

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			
SERIALIZABLE			

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			

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 (o3,10)
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 = o1;
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=o1 => 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 (if the app knows no updates can happen, then repeatable read is OK too).