

Schedule

- Today: Feb. 21 (TH)
 - ◆ Transactions, Authorization.
 - ◆ Read Sections 8.6-8.7. Project Part 5 due.
- Feb. 26 (T)
 - ◆ Datalog.
 - ◆ Read Sections 10.1-10.2. Assignment 6 due.
- Feb. 28 (TH)
 - ◆ Datalog and SQL Recursion, ODL.
 - ◆ Read Sections 10.3-10.4, 4.1-4.4. Project Part 6 due.
- Mar. 5 (T)
 - ◆ More ODL, OQL.
 - ◆ Read Sections 9.1. Assignment 7 due.

TRANSACTION MANAGEMENT

Airline Reservations	many updates
Statistical Abstract of the US	many queries

Atomicity – all or nothing principle

Serializability – the effect of transactions as if they occurred one at a time

Items – units of data to be controlled
fine-grained – small items
course-grained – large items
(granularity)

Controlling access by locks

Read – sharable with other readers **shared**
Write – not sharable with anyone else **exclusive**

Model – (item, locktype, transaction ID)

Transactions

= units of work that must be:

1. *Atomic* = either all work is done, or none of it.
2. *Consistent* = relationships among values maintained.
3. *Isolated* = appear to have been executed when no other DB operations were being performed.
 - ◆ Often called *serializable* behavior.
4. *Durable* = effects are permanent even if system crashes.

Commit/Abort Decision

Each transaction ends with either:

1. *Commit* = the work of the transaction is installed in the database; previously its changes may be invisible to other transactions.
2. *Abort* = no changes by the transaction appear in the database; it is as if the transaction never occurred.
 - ◆ ROLLBACK is the term used in SQL and the Oracle system.
 - In the ad-hoc query interface (*e.g.*, PostgreSQL psql interface), transactions are single queries or modification statements.
 - ◆ Oracle allows `SET TRANSACTION READ ONLY` to begin a multistatement transaction that doesn't change any data, but needs to see a consistent “snapshot” of the data.
 - In program interfaces, transactions begin whenever the database is accessed, and end when either a `COMMIT` or `ROLLBACK` statement is executed.

Example

`Sells(bar, beer, price)`

- Joe's Bar sells Bud for \$2.50 and Miller for \$3.00.
- Sally is querying the database for the highest and lowest price Joe charges:
 - (1) `SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar' ;`
 - (2) `SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar' ;`
- At the same time, Joe has decided to replace Miller and Bud by Heineken at \$3.50:
 - (3) `DELETE FROM Sells
WHERE bar = 'Joe''s Bar' AND
 (beer = 'Miller' OR beer = 'Bud') ;`
 - (4) `INSERT INTO Sells
VALUES('Joe''s bar', 'Heineken', 3.50) ;`
- If the order of statements is 1, 3, 4, 2, then it appears to Sally that Joe's minimum price is greater than his maximum price.
- Fix the problem by grouping Sally's two statements into one transaction, *e.g.*, with one SQL statement.

Example: Problem With Rollback

- Suppose Joe executes statement 4 (insert Heineken), but then, during the transaction thinks better of it and issues a ROLLBACK statement.
- If Sally is allowed to execute her statement 1 (find max) just before the rollback, she gets the answer \$3.50, even though Joe doesn't sell any beer for \$3.50.
- Fix by making statement 4 a transaction, or part of a transaction, so its effects cannot be seen by Sally unless there is a COMMIT action.

SQL Isolation Levels

Isolation levels determine what a transaction is allowed to see. The declaration, valid for one transaction, is:

```
SET TRANSACTION ISOLATION LEVEL X;
```

where:

- `X = SERIALIZABLE`: this transaction must execute as if at a point in time, where all other transactions occurred either completely before or completely after.
 - ♦ Example: Suppose Sally's statements 1 and 2 are one transaction and Joe's statements 3 and 4 are another transaction. If Sally's transaction runs at isolation level `SERIALIZABLE`, she would see the `Sell`s relation either before or after statements 3 and 4 ran, but not in the middle.

- ***X* = READ COMMITTED**: this transaction can read only committed data.
 - ◆ Example: if transactions are as above, Sally could see the original `Set1s` for statement 1 and the completely changed `Set1s` for statement 2.
- ***X* = REPEATABLE READ**: if a transaction reads data twice, then what it saw the first time, it will see the second time (it may see more the second time).
 - ◆ Moreover, all data read at any time must be committed; *i.e.*, **REPEATABLE READ** is a strictly stronger condition than **READ COMMITTED**.
 - ◆ Example: If 1 is executed before 3, then 2 must see the Bud and Miller tuples when it computes the min, even if it executes after 3. But if 1 executes between 3 and 4, then 2 may see the Heineken tuple.

- $X = \text{READ UNCOMMITTED}$: essentially no constraint, even on reading data written and then removed by a rollback.
 - ◆ Example: 1 and 2 could see Heineken, even if Joe rolled back his transaction.

Independence of Isolation Levels

Isolation levels describe what a transaction T with that isolation level sees.

- They *do not* constrain what other transactions, perhaps at different isolation levels, can see of the work done by T .

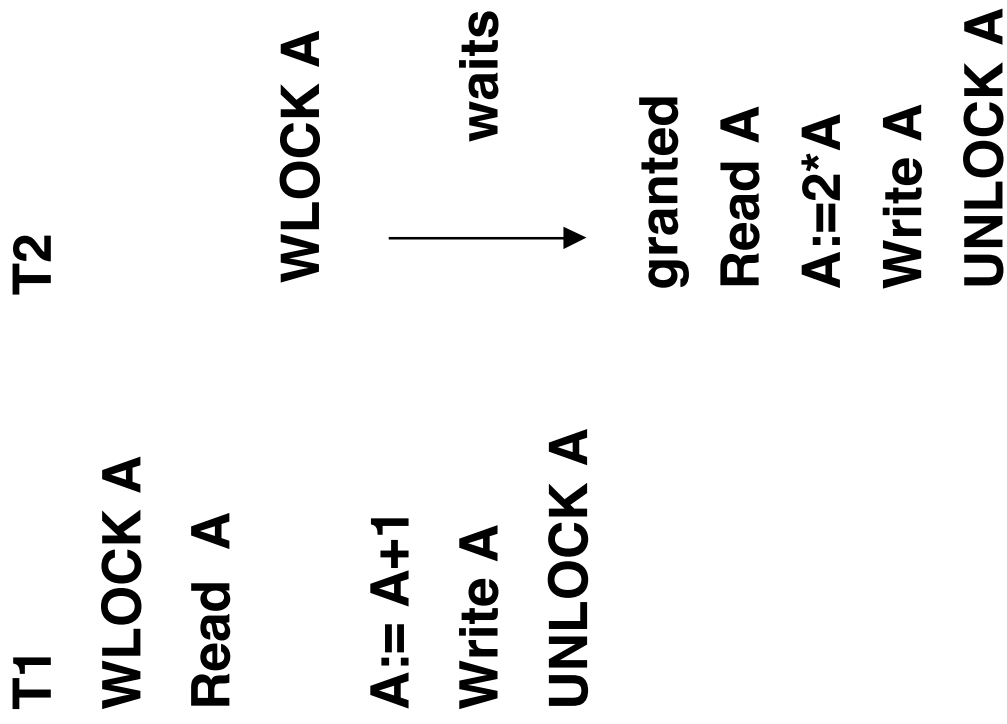
Example

If transaction 3-4 (Joe) runs serializable, but transaction 1-2 (Sally) does not, then Sally might see NULL as the value for both min and max, since it could appear to Sally that her transaction ran between steps 3 and 4.

T1	T2	start with A = 5	
Read A		A on disk	A in T1
	Read A	5	5
A := A + 1		5	5
	A := 2 * A	5	10
	Write A	10	10
Write A		6	10

		THEM		
		NO	R	W
RLOCK A				
WLOCK A	NO	OK	OK	OK
UNLOCK A	US	OK	OK	bad
	W	OK	bad	bad

RLOCK → UNLOCK can enclose a read
WLOCK → UNLOCK can enclose a write or read



	T1	T2
	RLOCK A	
	Read A	
		RLOCK A
		Read A
A:= A+1		A:= 2*A
	WLOCK A	WLOCK A
	wait	↓
		waits
		upgrade lock request
		upgrade lock request

Deadlock!

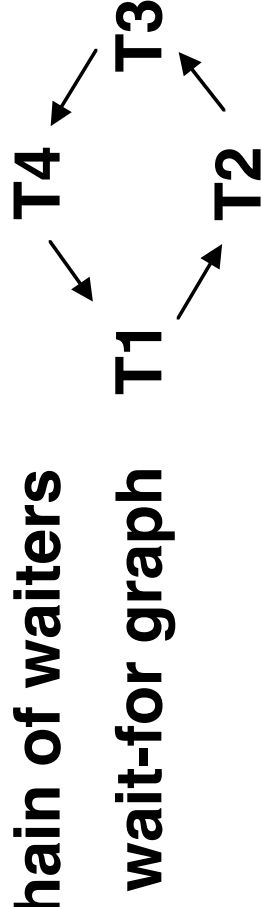
T1	T2
WLOCK A	
WLOCK B	WLOCK B
wait	WLOCK A
UNLOCK A	wait <u>deadlock</u>
UNLOCK B	UNLOCK B
	UNLOCK A

Deadlock

AND

1. Wait and hold some locks while you wait for others

2. Circular chain of waiters



3. No pre-emption

We can avoid deadlock by doing at least ONE of:

1. Get all your locks at once
2. Apply an ordering to acquiring locks
3. Allow preemption (for example, use timeout on waits)

disk 100 200

F

Read (A)

A:=A-50

Write (A)

Read (B)

$$B \stackrel{!}{=} B+50$$

Write (B)

T2

Read (A)

temp := A * 0.1

A := A + temp

Write (A)

Read (B)

B := B - temp

Write (B)

F

T2

disk 100 200

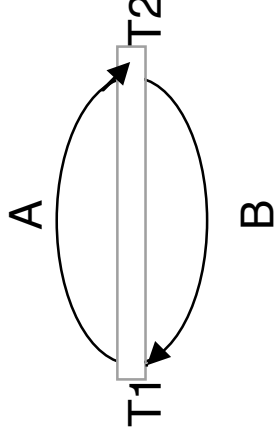
A



Schedule is serializable if effect is the same as a serial schedule

$$T_1 \rightarrow T_2$$


T2-T1



Ignore arithmetic. What is important is the same sequence of operations.

Conflicts

Read-write

Write-read

Write-write

Two schedules S1, S2 are equivalent if

- 1. Set of transactions in S1 and S2 are the same**
- 2. For each data item Q,
if in S1, T_i executes Read (Q)
and the value of Q read by T_i was written by T_j
then the same is true in S2**
- 3. For each data item Q,
if in S1, transaction T_i executes last write (Q),
then same is true in S2**

A schedule is serializable if it is equivalent to a serial schedule.

Non-fatal errors

Not recognizing that a schedule is serializable

Fatal errors

Thinking that a schedule is serializable when it is not

Algorithm: Testing serializability of a schedule

Input: Schedule S for transactions $T_1, \dots, T_k$

Output: Determination of whether S is serializable,
and if so, an equivalent serial schedule.

Method: Create a directed graph G (called a serialization graph)

 Create a node for each transaction and label with transaction ID

 Create an edge for each T_i : UNLOCK A_m followed by T_j : LOCK A_m
 (where lock modes conflict).

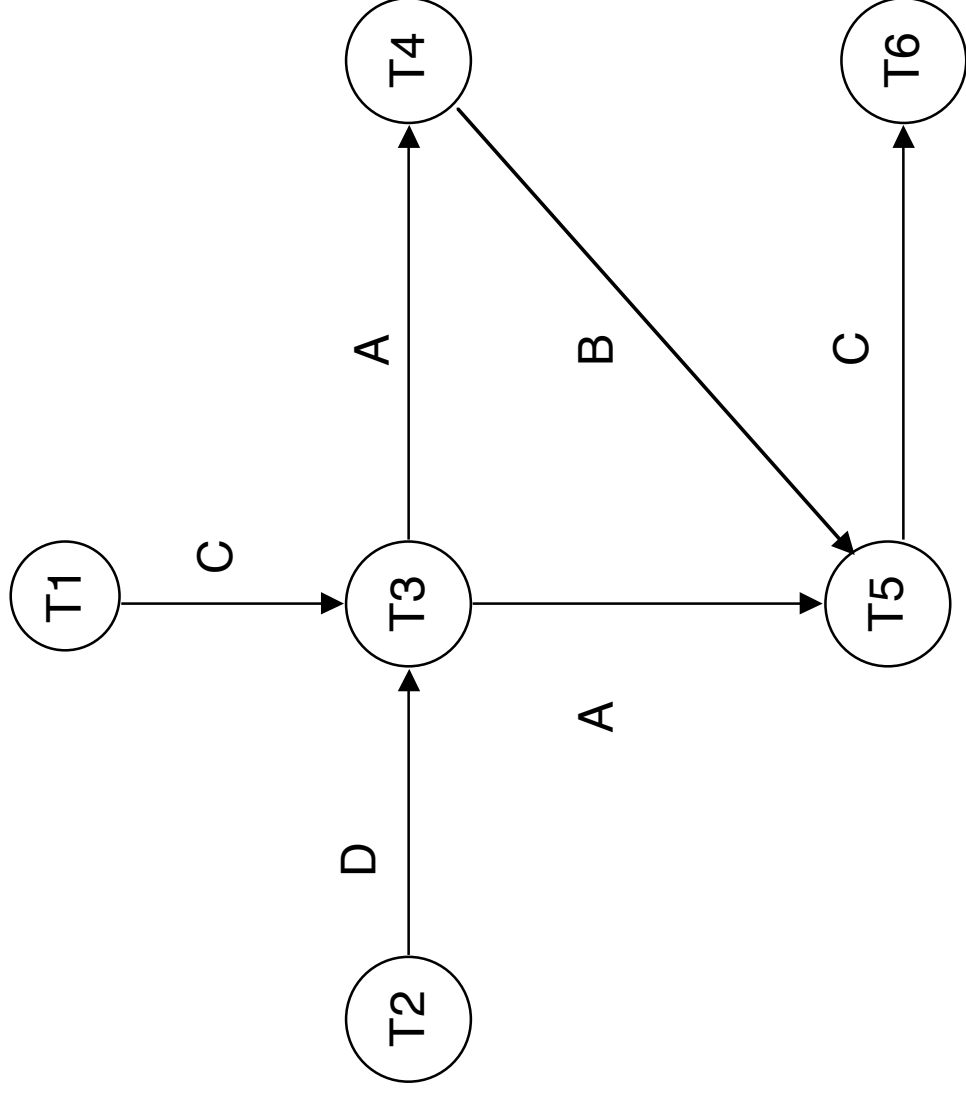
 The edge $T_i \rightarrow T_j$ labeled A_m (the data item)

If there is a cycle then schedule is non-serializable.

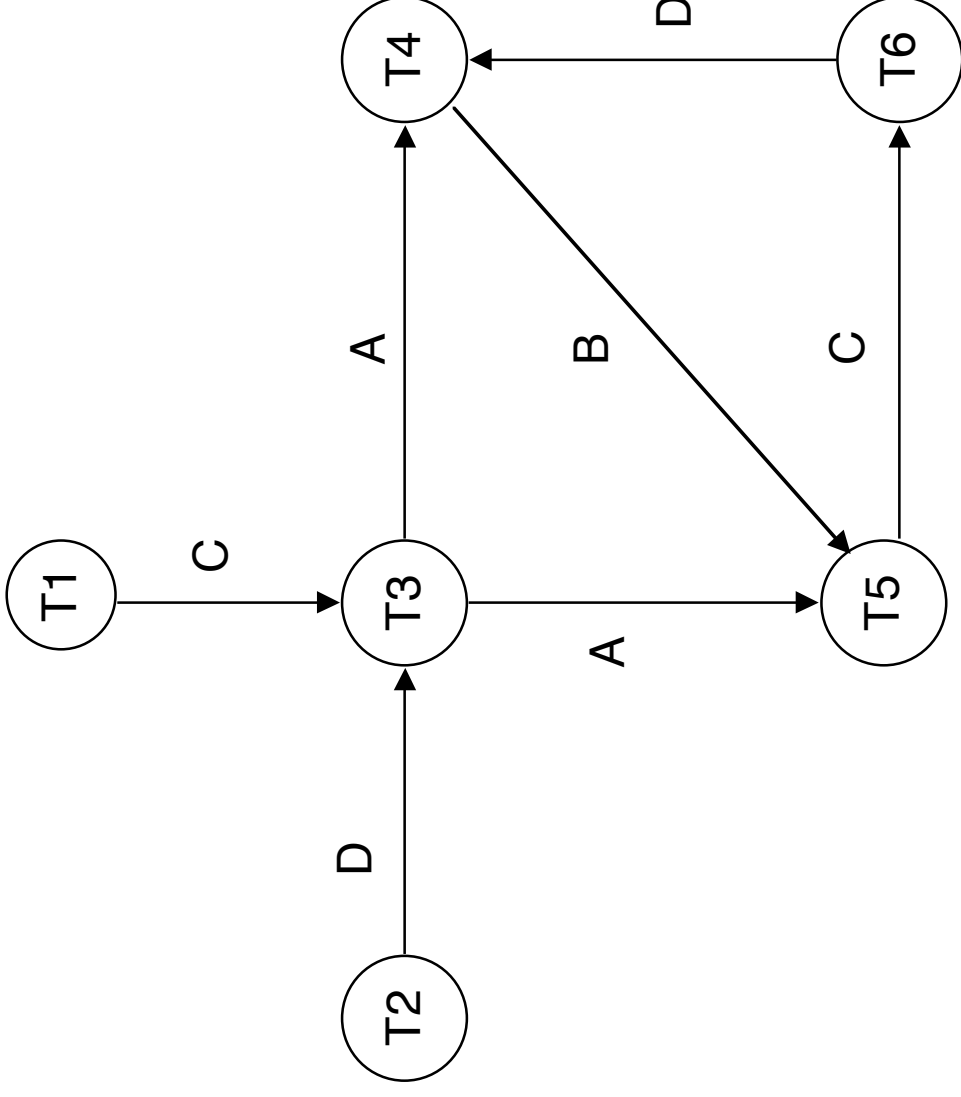
If there is no cycle, then (it is a DAG)
 do a topological sort to get a serial schedule

DAG implies some partial order.

Any total order consistent with the partial order is an equivalent serial schedule.



T1
T2
T3
T4
T5
T6



T1
T2
T3
T4
T5
T6

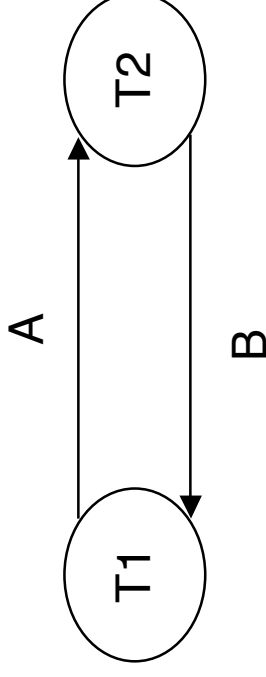
**If no progress is possible,
then there is a cycle**

T1
LOCK A
UNLOCK A

T2

LOCK A
UNLOCK A
LOCK B
UNLOCK B

LOCK B
UNLOCK B



ABORT CAN CAUSE CASCADING ROLLBACK

T1

T2

LOCK A

Read A

change A

Write A

UNLOCK A

LOCK A

Read A

change A

Write A

UNLOCK A

LOCK B

Read B

Discover problem

ABORT

Need to undo the change to A
CASCADED ABORT

How to avoid cascading rollback.

Make decision early

Defer commit of dependent transaction

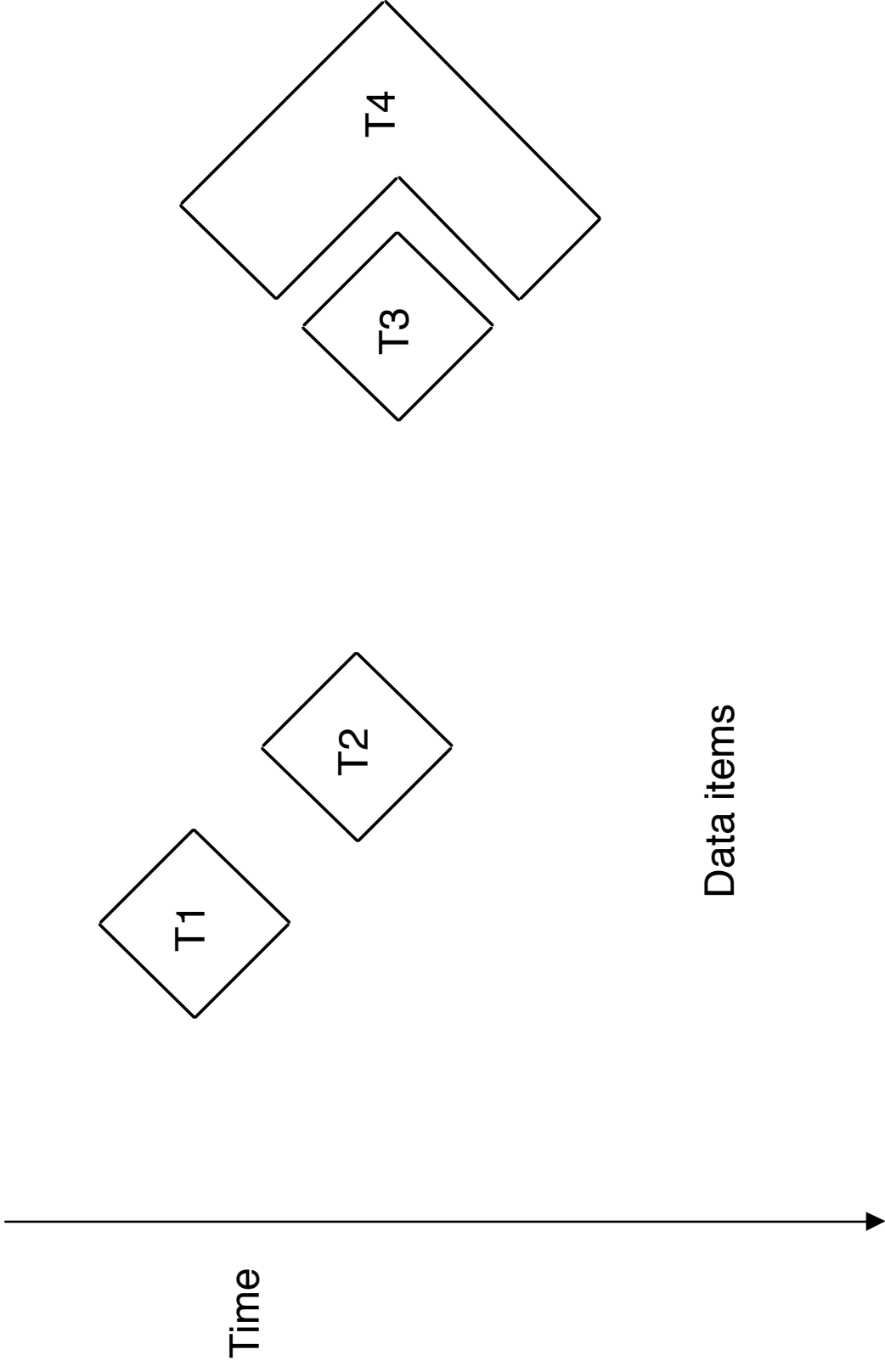
Hold locks until abort no longer possible

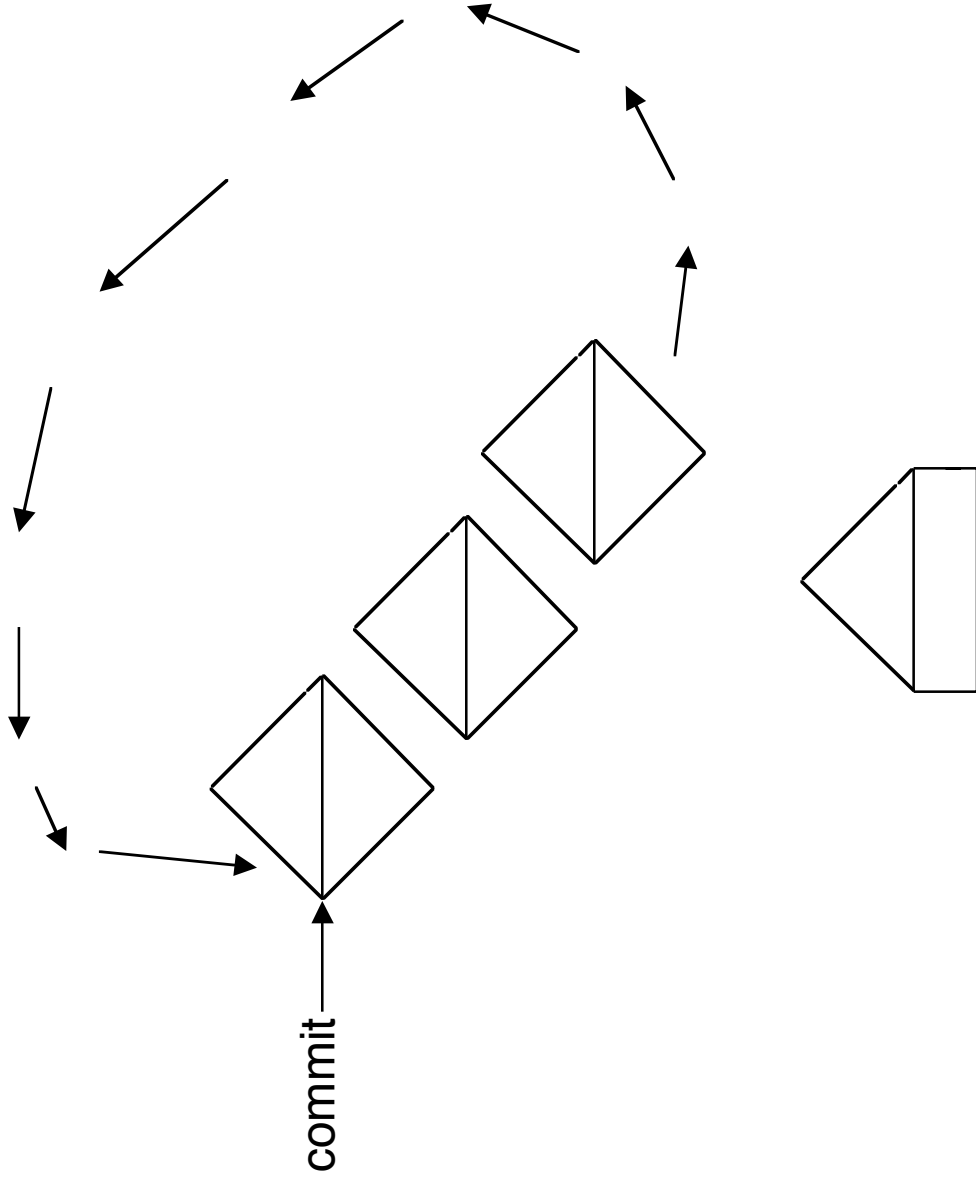
2PL 2-Phase Locking

Phase I: All requesting of locks precedes

Phase II: Any releasing of locks

Theorem: Any schedule for 2-phase locked transaction is serializable



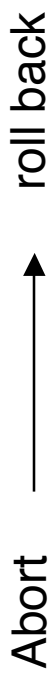


Commit or Abort occurs in between Phase I and Phase II



Effects are
permanent

Effects are
not visible



Read and write locks

Edges:

1. T_i read locks or write locks A_m

T_j is next transaction to write lock A

$i \neq j$ edge $T_i \rightarrow T_j$

R-W, W-W conflict

2. T_i write locks A

then $\forall$ transactions T_k that readlock A after T_i but before another

transaction write locks A

edge $T_i \rightarrow T_k$'s

W-R conflict

Ti WLOCK A

Tk1 RLOCK A
Tk2 RLOCK A

Tj WLOCK A

	N	R	W	I
None	OK	OK	OK	OK
Read	OK	OK	bad	bad
Write	OK	bad	bad	bad
Increment	OK	bad	bad	OK

T1

T2

INCR (A)

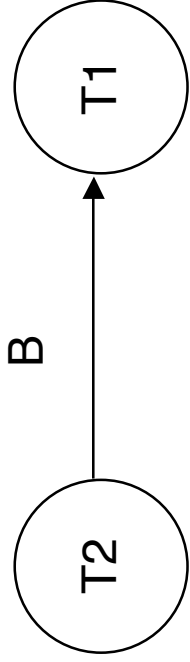
INCR (A)

READ (B)

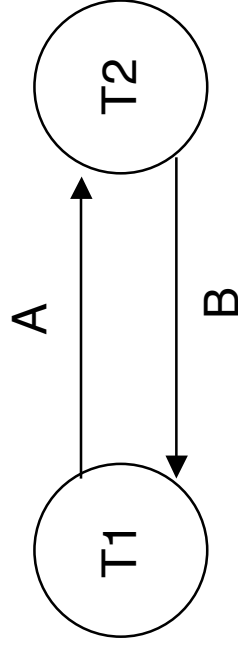
WRITE (B)

READ (B)

WRITE (B)



if increment locks



if no increment locks

Authorization in SQL

- File systems identify certain access privileges on files, *e.g.*, read, write, execute.
- In partial analogy, SQL identifies six access privileges on relations, of which the most important are:
 1. **SELECT** = the right to query the relation.
 2. **INSERT** = the right to insert tuples into the relation – may refer to one attribute, in which case the privilege is to specify only one column of the inserted tuple.
 3. **DELETE** = the right to delete tuples from the relation.
 4. **UPDATE** = the right to update tuples of the relation – may refer to one attribute.

Granting Privileges

- You have all possible privileges to the relations you create.
- You may grant privileges to any user if you have those privileges “with grant option.”
- ◆ You have this option to your own relations.

Example

1. Here, Sally can query `Sells` and can change prices, but cannot pass on this power:

```
GRANT SELECT ON Sells,  
      UPDATE(price) ON Sells  
TO sally;
```
2. Here, Sally can also pass these privileges to whom she chooses:

```
GRANT SELECT ON Sells,  
      UPDATE(price) ON Sells  
TO sally  
WITH GRANT OPTION;
```

Revoking Privileges

- Your privileges can be revoked.
- Syntax is like granting, but REVOKE ... FROM instead of GRANT ... TO.
- Determining whether or not you have a privilege is tricky, involving “grant diagrams” as in text.
However, the basic principles are:
 - a) If you have been given a privilege by several different people, then all of them have to revoke in order for you to lose the privilege.
 - b) Revocation is transitive. if *A* granted *P* to *B*, who then granted *P* to *C*, and then *A* revokes *P* from *B*, it is as if *B* also revoked *P* from *C*.