COMP9311:
Database Systems

Term 3 2022

Week 9 Transactions and Concurrency

By Xiaoyang Wang, CSE UNSW

Textbook: Chapters 20, 21 and 22

Disclaimer: the course materials are sourced from

- previous offerings of COMP9311 and COMP3311
- Prof. Werner Nutt on Introduction to Database Systems (http://www.inf.unibz.it/~nutt/Teaching/IDBs1011/)

Transactions, Concurrency

DBMSs provide access to valuable information resources in an environment that is:

- *shared* - concurrent access by multiple users
- *unstable* - potential for hardware/software failure

DBMS aims to shield the users from this environment … Each user should see the system as:

- unshared - their work is not inadvertantly affected by others
- stable - the data survives in the face of system failures

Goal of DBMS: “data integrity is always maintained”.

Transactions, Concurrency

DBMS implements the following three concepts to ensure data integrity …

(1) Transaction processing

techniques for describing "logical units of work" in applications in terms of underlying DBMS operations

(2) Concurrency control

techniques for ensuring that multiple concurrent transactions do not interfere with each other

(3) Recovery mechanisms

techniques to restore information to a consistent state, even after major hardware shutdowns/failures

What is … Transaction Processing

For DBMS to handle potential failures correctly and efficiently:

Each database user must express her database work requirements (e.g., your SQL statements) as **a set of transactions.**

A database *transaction* is a "logical *unit* of work" in a DB application.

A transaction typically comprises multiple DBMS operations.

E.g. select ... update ... insert ... select ... insert ...

Examples:

- booking a concert ticket
- transferring funds between bank accounts
- updating stock levels via point-of-sale terminal
- enrolling in a course or class

Example Transaction in code

Transfer funds between two accounts in same bank. Rough implementation in PLpgSQL:

```
create or replace function
    transfer(src int, dest int, amount float) returns void
as $$
declare
    oldBalance float;
    newBalance float;
begin
    -- error checking
    select * from Accounts where id=src;
    if (not found) then
        raise exception 'Invalid Withdrawal Account';
    end if;
    select * from Accounts where id=dest;
    if (not found) then
        raise exception 'Invalid Deposit Account';
    end if;
...
```

```
...
    -- action
(A) select balance into oldBalance
    from Accounts where id=src;
    if (oldBalance < amount) then
        raise exception 'Insufficient funds';
    end if;
    newBalance := oldBalance - amount;
(B) update Accounts
    set    balance := newBalance
    where  id = src;
    -- partial completion of transaction
(C) update Accounts
    set    balance := balance + amount
    where  id = dest;
    commit; -- redundant; function = transaction
end;
$$ language plpgsql;
```

Example Transaction

Now using this function and the transaction that contains A, B and C action points …

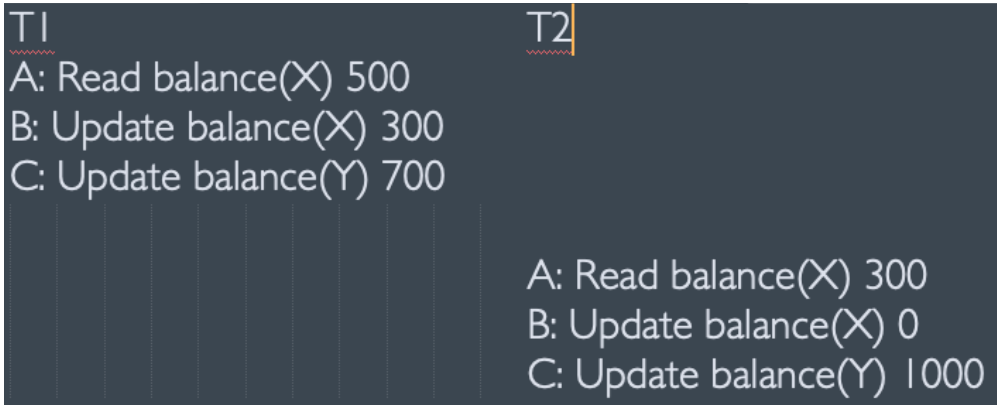
Consider two simultaneous transfers between accounts …

we have two transactions … starting balances of X = 500, Y = 500

- T1 transfers $200 from account X to account Y
- T2 transfers $300 from account X to account Y

If the sequence of events is like:

```
T1:  ...  A  B  C  ...
T2:             ...  A  B  C  ...
```



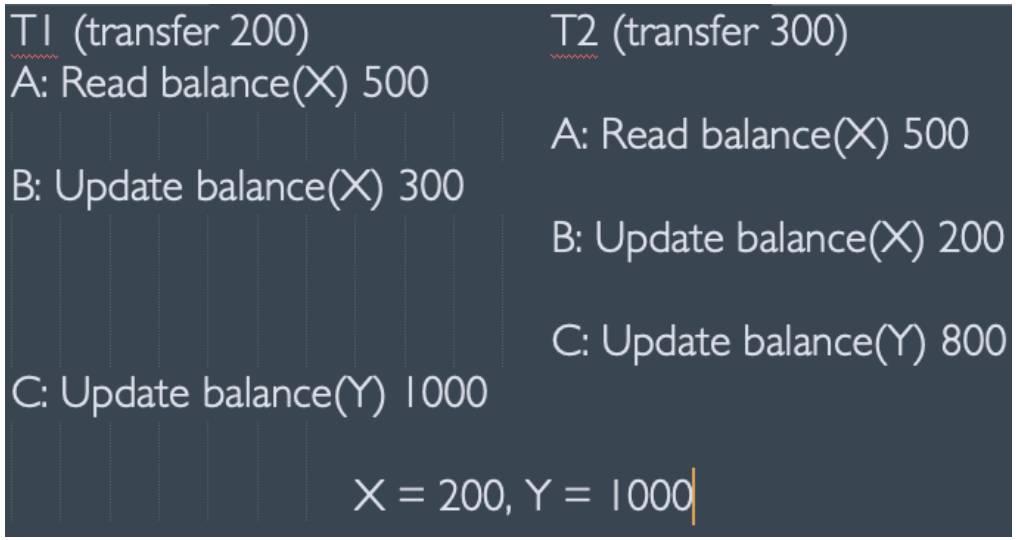
everything works correctly, i.e.

- overall, account X is reduced by $500
- overall, account Y is increased by $500

Example Transaction

What if the sequence of events is like?

```
T1:   ...   A    B          C   ...
T2:     ...   A    B    C   ...
```



In terms of database operations,

this is what happens:

- T1 gets balance from X (oldBal)
- T2 gets same balance from X (oldBal)
- T1 decrements balance in X (oldBal - 200)
- T2 decrements balance in X (oldBal - 300)
- T2 increments balance in Y (bal + 200) `// new balance for Y is bal + 200 = 700`
- T1 increments balance in Y (bal + 300) `// new balance for Y is bal + 300 = 1000`

Final balance of Y is ok; final balance of X is wrong (inter-leaving operations may cause it)

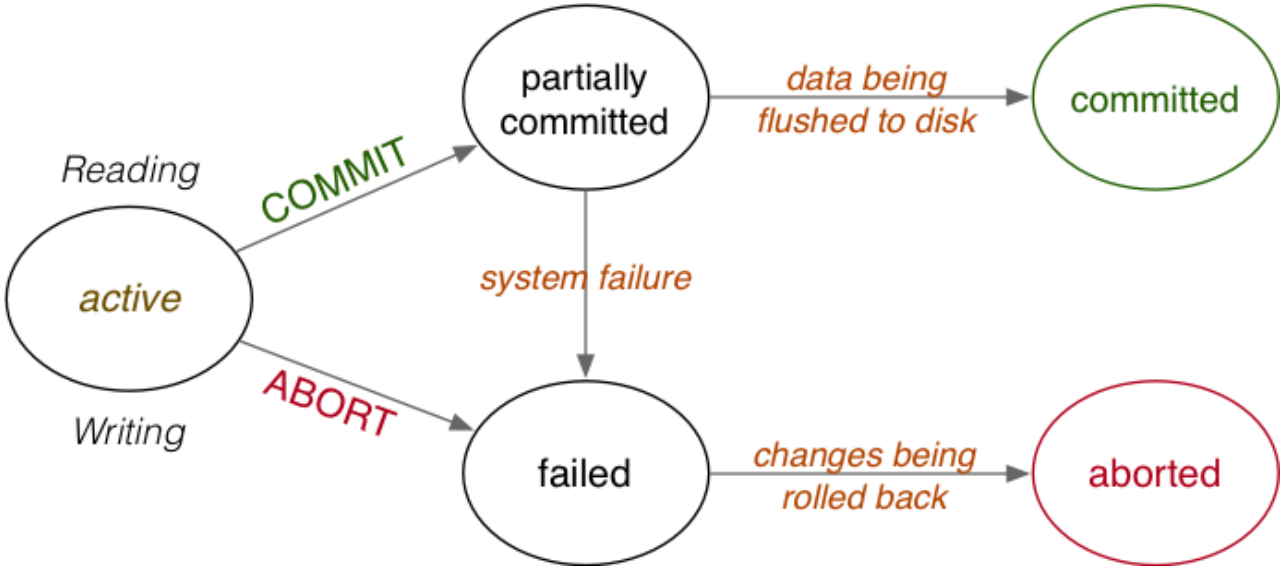
Transaction management is to prevent transactions being executed in this fashion, while still being able to run as many transactions as possible at any given time.

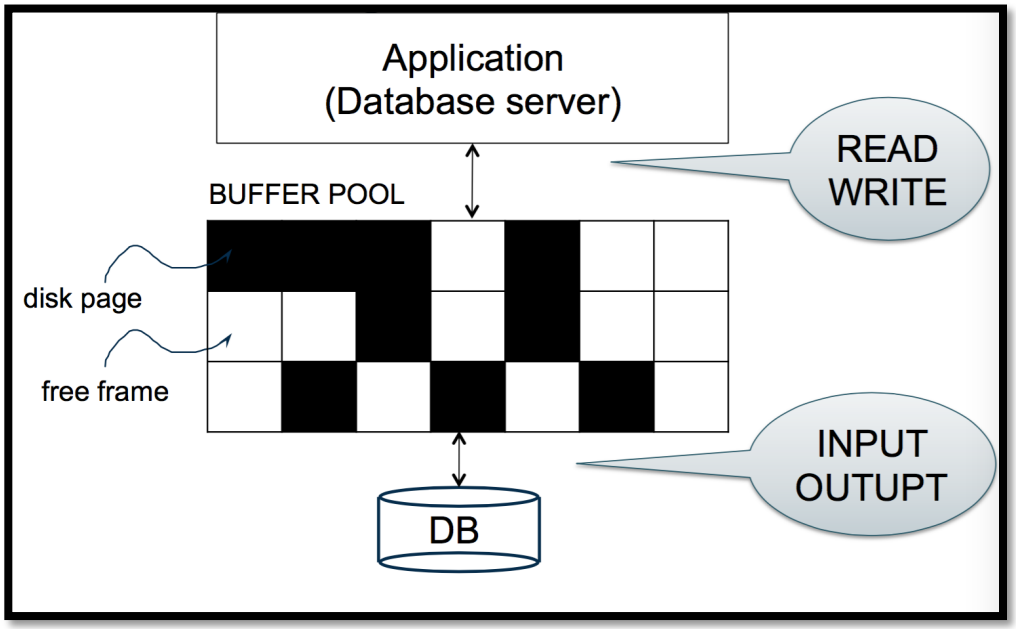
Transaction Concepts

Let's define some basic concepts for transaction processing.

A **transaction** must always terminate, either:

- successfully (COMMIT), with all changes *correctly* preserved
- unsuccessfully (ABORT), with database *unchanged*





Transaction Concepts – Read/Write

READ - compute the data block that contains the item to be read

Either

- find a buffer containing the block, or
- read from disk into a buffer

Copy (read) the value from the buffer.

SELECT produces READ operations on the database.

WRITE - compute the disk block containing the item to be written

Either

- find a buffer containing the block, or
- read from disk into a buffer

Copy (write) the *new value* into the buffer

At some point (maybe later), write the buffer back to disk.

INSERT, UPDATE, DELETE produce WRITE/ READ operations.

Transaction Concepts

To describe transaction effects, we consider:

- READ - transfer data from disk to memory
- WRITE - transfer data from memory to disk
- ABORT - terminate transaction, unsuccessfully
- COMMIT - terminate transaction, successfully

The READ, WRITE, ABORT, COMMIT operations:

- occur in the context of some transaction *T*
- involve manipulation of data items *X, Y, ...* (READ and WRITE)

The operations are typically denoted as:

$R_T(X)$	read item *X* in transaction *T*
$W_T(X)$	write item *X* in transaction *T*
A_T	abort transaction *T*
C_T	commit transaction *T*

Transaction Concepts

Execution of the above funds transfer example can be described as

```
T: READ(S);  READ(D);  -- S = source tuple, D = dest tuple
   READ(S);  S.bal := S.bal-amount;  WRITE(S)
   READ(D);  D.bal := D.bal+amount;  WRITE(D)
   COMMIT;
```

or simply as:

$$R_T(S) \quad R_T(D) \quad R_T(S) \quad W_T(S) \quad R_T(D) \quad W_T(D) \quad C_T$$

Let's say there were two transactions (i.e., running the funds transfer function 2 times)

schedule: $R_1(S) \quad R_1(D) \quad R_1(S) \quad W_1(S) \quad R_1(D) \quad W_1(D) \quad C_1 \; R_2(S) \quad R_2(D) \quad R_2(S) \quad W_2(S) \quad R_2(D) \quad W_2(D) \quad C_2$

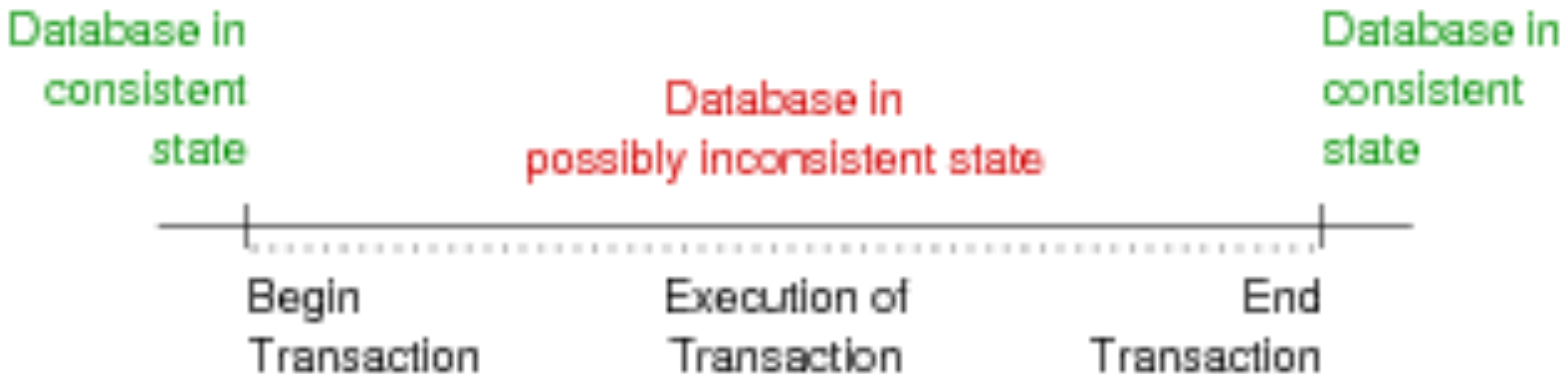
This is known as a ***schedule*** for the transactions.

Note: this is not a "plan", this is a trace of the execution of transactions (i.e., log/history). It is a very important distinction to remember.

Transaction Consistency

Transactions typically have intermediate states that are inconsistent.

However, states *before* and *after* transaction must be consistent.



ACID Properties

Data integrity is assured if transactions satisfy the following:

Atomicity

- Either all operations of transaction are reflected in database or none are.

Consistency

- Execution of a transaction in isolation preserves data consistency.

Isolation

- Each transaction is "unaware" of other transactions executing concurrently in the system.

Durability

- After a transaction completes successfully, its changes persist even after subsequent system failure.

ACID Properties

Atomicity is handled by the *commit* and *rollback* mechanisms.

- **commit** saves all changes and ends the transaction
- **rollback** *undoes* changes already made by the transaction

Durability is handled by implementing *stable storage*, via

- redundancy, to deal with hardware failures
- logging/checkpoint mechanisms, to recover state

Here, we consider primarily **C**onsistency and **I**solation.

Transaction Anomalies

If *concurrent* transactions access *shared* (i.e., the same) data objects, various anomalies can arise.

We give examples using the following two transactions:

```
T1: read(X)
    X := X + N
    write(X)
    read(Y)
    Y := Y - N
    write(Y)
```

```
T2: read(X)
    X := X + M
    write(X)
```

and initial DB state X=100, Y=50, N=5, M=8.

T1 and T2 are sharing X.

Serial Schedules

A schedule is a trace of transaction operations. A serial schedule is a schedule that doesn't overlap with other transaction operations (i.e., isolated with each other)

If T1 and T2 transactions are executed in a ***serial*** schedule, like:

```
T1: R(X) W(X) R(Y) W(Y)
T2:                    R(X) W(X)

or

T1:           R(X) W(X) R(Y) W(Y)
T2: R(X) W(X)
```

the database is left in a consistent state. So this is ideal schedule. We want to run transactions as if they are isolated from each other … i.e., as serial schedules.

The basic idea behind serial schedules:

- the database starts in a consistent state
- the first transaction completes, leaving the DB consistent
- the next transaction completes, leaving the DB consistent

As would occur e.g. in a single-user database system.

Serial Schedules

initial DB state X=100, Y=50, N=5, M=8

For the first schedule in our example (executed in serial):

Database		T1			T2	
X	Y		X	Y		X
100	50		?	?		?
		read(X)	100			
		X:=X+N	105			
105		write(X)				
		read(Y)		50		
		Y:=Y-N		45		
	45	write(Y)				
					read(X)	105
					X:=X+M	113
113					write(X)	
113	45					

Serial Schedules

initial DB state X=100, Y=50, N=5, M=8

For the second schedule in our example (executed in serial):

Database		T1			T2	
X	Y		X	Y		X
100	50		?	?		?
					read(X)	100
					X:=X+M	108
108					write(X)	
		read(X)	108			
		X:=X+N	113			
113		write(X)				
		read(Y)		50		
		Y:=Y-N		45		
	45	write(Y)				
113	45					

Serial Schedules

Note that serial execution doesn't mean that each transaction got the same results, regardless of the order.

Consider the following two transactions:

```
T1: select sum(salary)
    from Employee where dept='Sales'

T2: insert into Employee
    values (....,'Sales',...)
```

- If we execute T1 then T2, we get a smaller salary total than if we executed T2 then T1.
- In both cases, however, the salary total is *consistent* with the state of the database *at the time* the query is executed:

Again … we repeat: the basic idea behind serial schedules:

- the database **starts** in a consistent state
- the first transaction completes, **leaving** the DB **consistent**
- the next transaction completes, **leaving** the DB **consistent**

Concurrent Schedules

A serial execution of transactions leaves DB consistent.

but not great in terms of processing a large amount of transactions because it meant all transactions are executed in sequence… so DBMS run transactions “concurrently (non-serial)”.

But … if transactions execute in a manner that they produces a *concurrent* (non-serial) schedule, the potential exists for conflict among their effects. In the worst case, the effect of executing the transactions ...

- is to leave the database in an inconsistent state
- even though each transaction, by itself, *is* consistent

So why don't we observe such problems in real DBMSs with high # of transactions? ...

- *concurrency control* mechanisms handle them (see later).

Valid Concurrent Transactions

Not all concurrent executions cause problems.

For example, the schedules (interleaving, not serial)

```
T1: R(X) W(X)             R(Y) W(Y)
T2:           R(X) W(X)
```

or

```
T1: R(X) W(X)       R(Y)       W(Y)
T2:           R(X)       W(X)
```

or ...

still leave the database in a consistent state.

Lost Update Problem

Consider the following schedule where the transactions execute in parallel:

```
T1: R(X)       W(X) R(Y)       W(Y)
T2:      R(X)             W(X)
```

In this scenario:

- T2 reads data (X) that T1 is currently operating on
- then makes changes to X and overwrites T1's result

This is called a *Write-Read (WR) Conflict* or *dirty read*.

The result: T1's update to X is lost.

Lost Update Problem

Consider the states in the WR Conflict schedule: (n=5, m=8)

Database		T1			T2	
X	Y		X	Y		X
100	50		?	?		?
		read(X)	100			
		X:=X+N	105			
					read(X)	100
					X:=X+M	108
105		write(X)				
		read(Y)		50		
108					write(X)	
		Y:=Y-N		45		
	45	write(Y)				
108	45					

Temporary Update Problem

Consider the following schedule where one transaction fails:

```
T1: R(X) W(X) A
T2:              R(X) W(X)
```

Transaction T1 aborts after writing X.

The abort *will* undo the changes to X, but where the undo occurs can affect the results.

Consider three places where undo might occur:

```
T1: R(X) W(X) A [1]      [2]      [3]
T2:                R(X)     W(X)
```

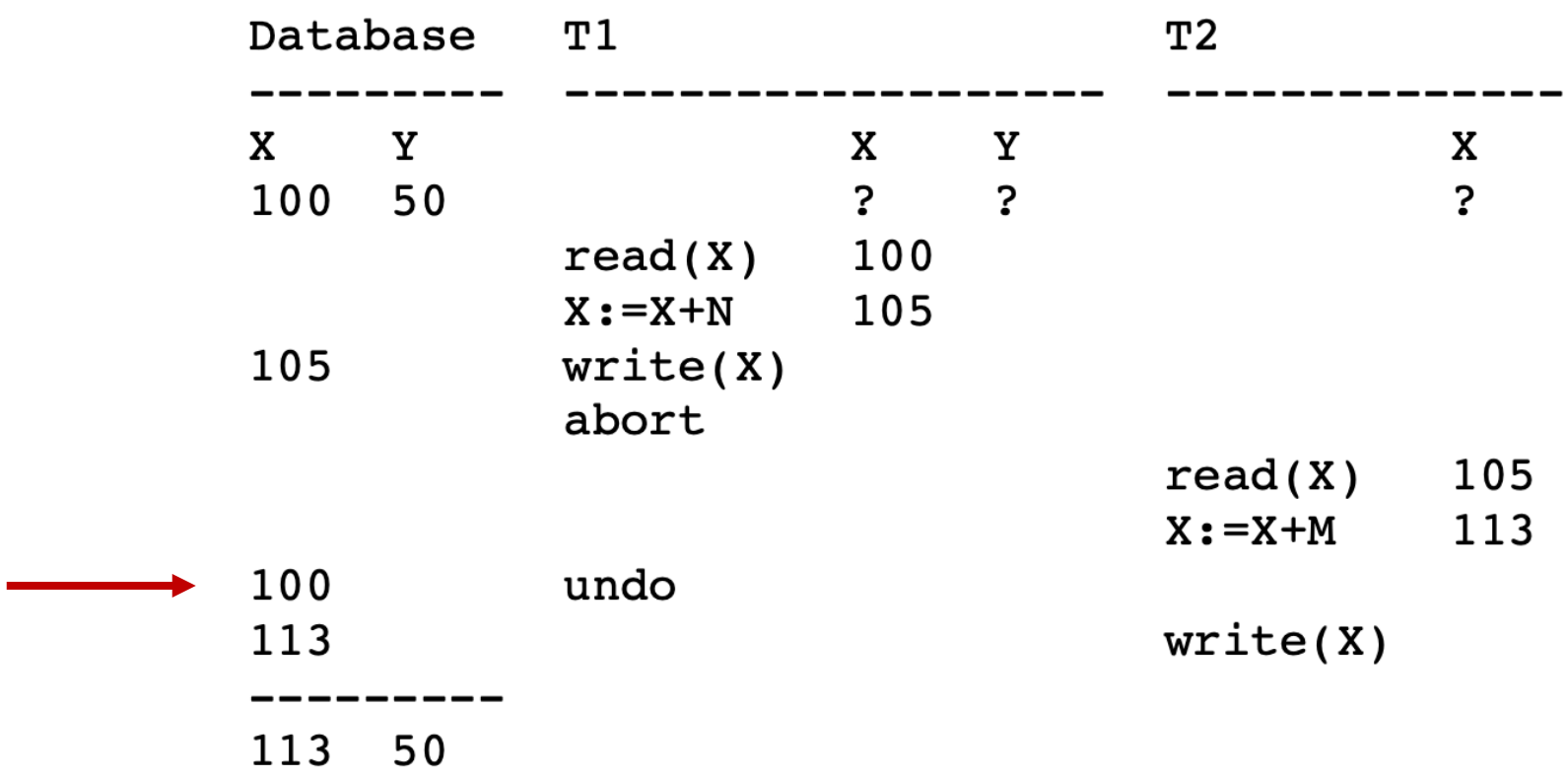
Temporary Update – Case 1

This scenario is ok. T1's effects have been eliminated.

Database		T1			T2	
X	Y		X	Y		X
100	50		?	?		?
		read(X)	100			
		X:=X+N	105			
105		write(X)				
		abort				
100		undo				
					read(X)	100
					X:=X+M	108
108					write(X)	
108	50					

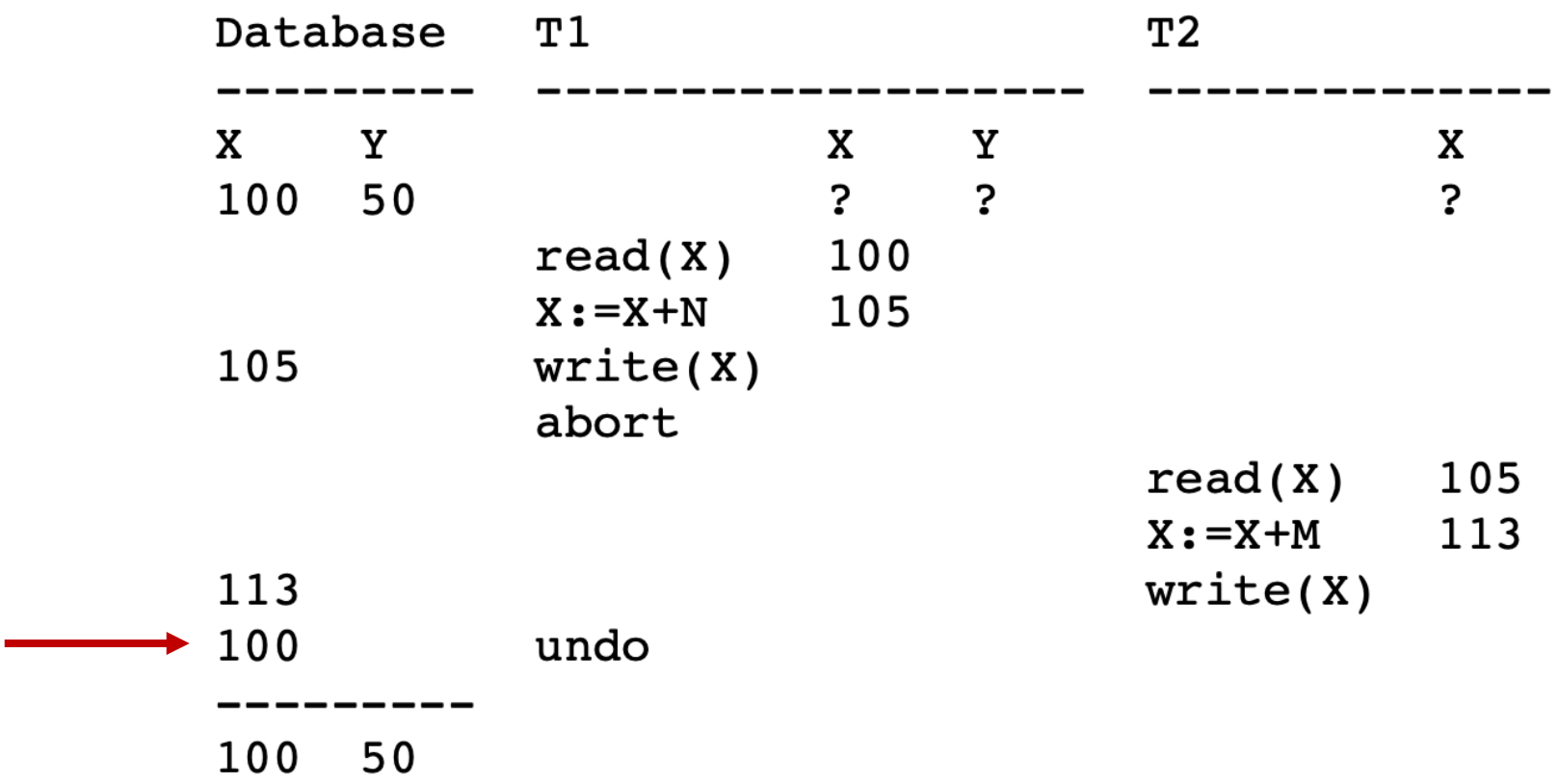
Temporary Update – Case 2

In this scenario, some of T1's effects have been retained.



Temporary Update – Case 3

In this scenario, T2's effects have been lost, even after commit.



Database		T1			T2	
X	Y		X	Y		X
100	50		?	?		?
		read(X)	100			
		X:=X+N	105			
105		write(X)				
		abort				
					read(X)	105
					X:=X+M	113
113					write(X)	
100		undo				
100	50					

Serializability

For ACID, the DBMS must run transactions in a way that it produces a “serializable” schedule

Serializable

- *The effect of executing n concurrent transactions is the same as the effect of executing them serially in some order*.
  - (i.e., may not be in serial, but the effect is as if they are in serial)
- DBMS uses a concurrency control technique to run transactions concurrently so they do not have to run the high number of transactions sequentially
- For assessing the correctness of concurrency control methods, we can perform a test to see whether it produces serializable schedules (→ Serializability Test) …
  - DBMS should devise a concurrency control method that is guaranteed to produce only serializable schedules.
  - The use of the serialisability concept is relevant in “theoretically” proving/testing that the schedules are serializable

Serializability

If a concurrent schedule for transactions $T_1 .. T_n$ **acts like a serial schedule** for $T_1 .. T_n$, then consistency is guaranteed.

To determine **this** requires a notion of *schedule equivalence (i.e., "equality" test)*.

Note: we are not attempting to determine equivalence of entire computations, simply of the interleaved sequences of read/write operations.

A *serializable schedule* is a concurrent schedule that produces a final state that is the same as that produced by *some serial schedule*.

There are two primary testing formulations of serializability:

- *conflict serializibility* (read/write operations occur in the "right" order)
- *view serializibility* (read operations *see* the correct version of data)

Conflict Serializability

Consider two transactions T_1 and T_2 acting on data item X.

Considering only read/write operations, the possibilities are:

T_1 first	T_2 first	Equiv?
$R_1(X)\ R_2(X)$	$R_2(X)\ R_1(X)$	yes
$R_1(X)\ W_2(X)$	$W_2(X)\ R_1(X)$	no
$W_1(X)\ R_2(X)$	$R_2(X)\ W_1(X)$	no
$W_1(X)\ W_2(X)$	$W_2(X)\ W_1(X)$	no

If T_1 and T_2 act on different data items result is equivalent regardless of order.

Conflict Serializability

Two transactions have a potential *conflict* if

- they perform operations on the same data item
- at least one of the operations is a write operation

- In the above cases, the order of operations affects the result.
- Conversely, if two operations in a schedule don't conflict, we can swap their order without affecting the overall result.
- This gives a basis for determining equivalence of schedules.

If we can transform a schedule

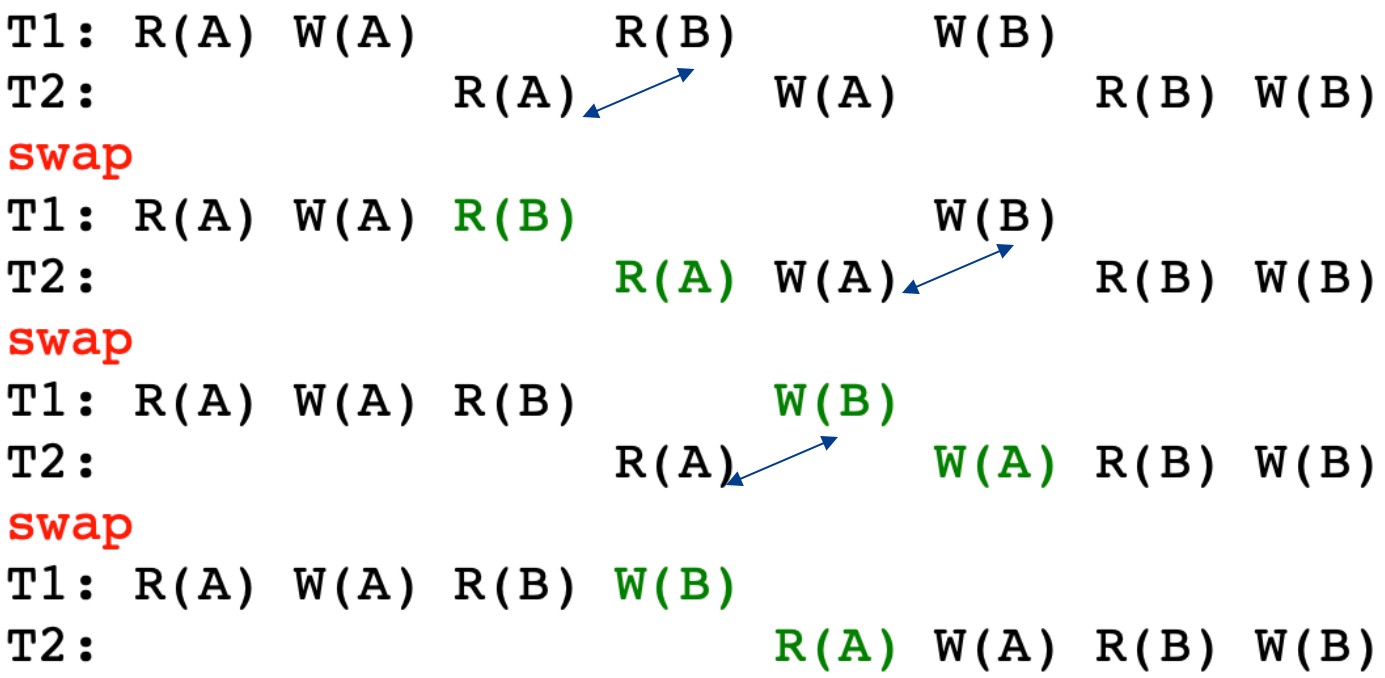
- by swapping the orders of non-conflicting operations
- such that the result is a serial schedule

then we say that the schedule is *conflict serializible*.

If a concurrent schedule is equivalent to some (any) serial schedule, then we have a consistency guarantee.

Conflict Serializability

Example: transform a concurrent schedule to serial schedule



View Serializability

View Serializability is

- an alternative formulation of serializability
- that is less conservative than conflict serializability (CS)
  (i.e., some schedules that are view serializable are not conflict serializable)

As with CS, it is based on a notion of schedule equivalence

- a schedule is "safe" if *view equivalent* to a serial schedule
- i.e., compare a schedule with a serial schedule → check “view equivalent”, if yes, “view serialisiable”

The idea: if all the read operations in two schedules ...

- always read the result of the same write operations
- then the schedules must produce the same result

View Serializability

Two schedules *S* and *S'* on T_1 .. T_n are *view equivalent* iff

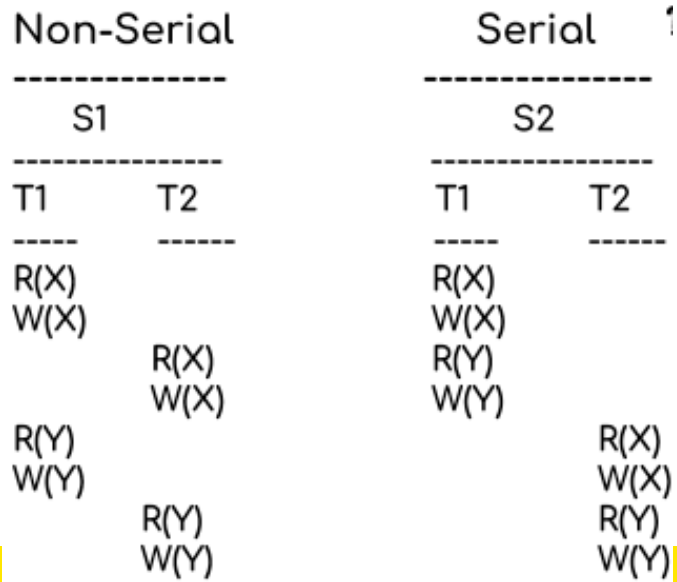
for each shared data item *X*

- (1) initial read: if T_j reads the initial value of *X* in *S*, then it also reads the initial value of *X* in *S'*
- (2) update read: if T_j reads *X* in *S* and *X* was produced by T_k, then T_j must also read the value of *X* produced by T_k in *S'*
- (3) final write: if T_j performs the final write of *X* in *S*, then it must also perform the final write of *X* in *S'*

To check serializibilty of *S*,

find a serial schedule

that is view equivalent to *S*



Testing Serializability

In designing concurrency control schemes (i.e., lock management system), we need a way of checking whether they produce "safe" schedules (i.e., serializable schedules).

This is typically achieved by a demonstration that the scheme generates only serializable schedules, and we need a serializability test for this.

There is a simple and efficient test for conflict serializability; there is a more complex test for view serializablity.

Both tests are based on notions of

- building a graph to represent transaction interactions
- testing properties of this graph (checking for cycles)

Testing Serializability - *precedence graph*

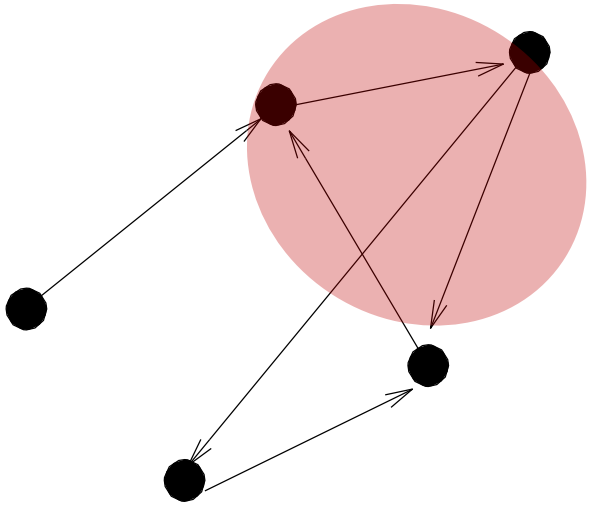
A *precedence graph G = (V,E)* for a schedule *S* consists of

- a vertex in *V* for each transaction from T_1 .. T_n
- an edge in *E* for each pair T_j and T_k, such that
  - there is a pair of conflicting operations between T_j & T_k
  - the T_j operation occurs before the T_k operation

Note: the edge is directed from $T_j \rightarrow T_k$

If an edge $T_j \rightarrow T_k$ exists in the precedence graph

- then T_j must appear before T_k in any serial schedule



Cyclic Graph

Implication: if the precedence graph has cycles, then *S* can't be serialized.

Thus, the serializability test is reduced to cycle-detection

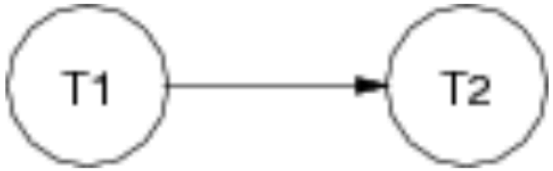
(and there are cycle-detection algorithms available in many algorithms textbooks)

Serializability Test Examples

Serializable schedule (with conflicting operations shown in red):

```
T1: R(A) W(A)        R(B)        W(B)
T2:           R(A)        W(A)        R(B) W(B)
```

Precedence graph for this schedule:



No cycles ⇒ serializable (as we already knew)

Serializability Test Examples

Consider this schedule:

```
T1: R(A)                    W(A) R(B) W(B)
T2:      R(A) W(A) R(B)                    W(B)
```

Precedence graph for this schedule:

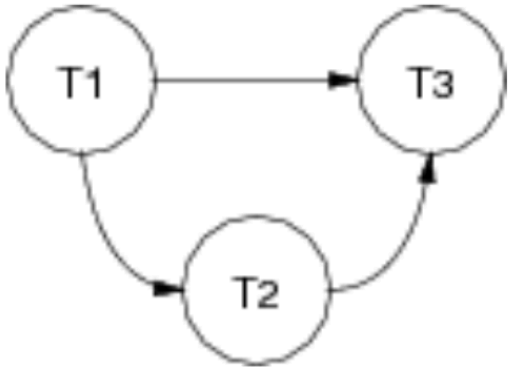
Has a cycle ⇒ not serializable

Serializability Test Examples

Consider this 3-transaction schedule:

```
T1: R(A)R(C)W(A)     W(C)
T2:              R(B)     R(A)     W(B)          W(A)
T3:                                   R(C)     R(B)W(C)     W(B)
```

Precedence graph for this schedule:



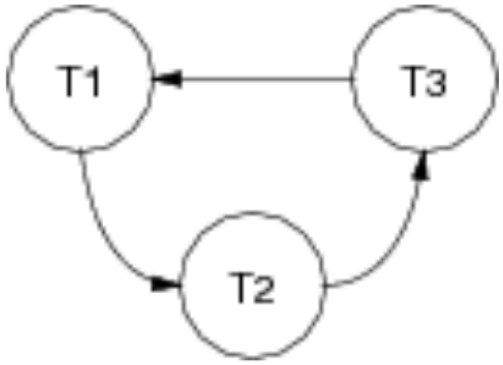
No cycles ⇒ serializable

Serializability Test Examples

Consider this 3-transaction schedule:

```
T1: R(A)                    W(A)          R(C)     W(C)
T2:      R(B)     W(B)                R(A)     W(A)
T3:           R(C)     W(C)     R(B)                    W(B)
```

Precedence graph for this schedule:



Has a cycle ⇒ not serializable

Concurrency Control

- Having serializability tests is useful theoretically, but they do not provide a practical tool for organising schedules.
  - *the # of n* transactions to be considered at any given time is high in most database management systems
  - the cost of testing for serializability via graphs may not be acceptable
  - Besides … the schedules are traces (i.e., log/history) of executed transactions, not applicable for organising *future* transactions

- *What is required are methods (= a set of rules) that can be applied to each transaction individually which guarantee that any combination of transactions is serializable*

- *This method is called concurrency control and the method aims to produce serializable schedules from concurrent transactions.*

Concurrency Control Methods

Approaches that DBMS use to produce serializable schedules:

- **lock-based**: synchronise transaction execution via locks on some portion of the database.
- **multiversion-based**: allow multiple "*consistent*" versions of the data to exist, and allocate each transaction exclusively to access one version.
- **timestamp-based**: organise transaction execution in advance by assigning timestamps to operations.
- **validation-based (optimistic concurrency control)**: exploit typical execution-sequence properties of transactions to determine safety dynamically.

Concurrency Control Methods

Locking Mechanism

- The idea of locking some data item X is to:
  - give a transaction exclusive use of the data item X,
  - do not restrict the access of other data items.

- This prevents one transaction from changing a data item currently being used in another transaction.

We will discuss a simple locking scheme which locks individual items, using read and write locks

Lock-based Concurrency Control

Synchronise access to shared data items via following rules:

- before reading *X*, get shared (=read) lock on *X*
- before writing *X*, get exclusive (=write) lock on *X*
- an attempt to get a shared lock on *X* is blocked if another transaction already has exclusive lock on *X*
- an attempt to get an exclusive lock on *X* is blocked if another transaction has any kind of lock on *X*

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

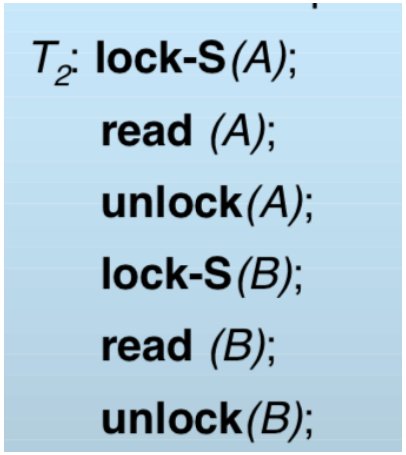
	Shared (read)	Exclusive (write)
Shared	Yes	No
Exclusive	No	No

In this scheme,

- Several read locks can be issued on the same data item at the same time.
- A read lock and a write lock cannot be issued on the same data item at the same time, neither two write locks

Lock-based Concurrency Control

Example of a transaction performing locking:



Display Sum (A and B)

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.
- Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.
- These rules alone do not guarantee serializability.

Two Phase Locking (2PL)

To guarantee serializability, transactions must also obey the two-phase locking protocol:

- **Growing Phase**: all locks for a transaction must be obtained before any locks are released
    - transaction may obtain locks
    - transaction may not release locks
- **Shrinking Phase**: gradually release all locks (once a lock is released no new locks may be requested).
    - transaction may release locks
    - transaction may not obtain locks

The protocol assures conflict serializability. It can be proved that the transactions can be serialized in the order of their lock points → the point where a transaction acquired its final lock.

Two Phase Locking (2PL) Example

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

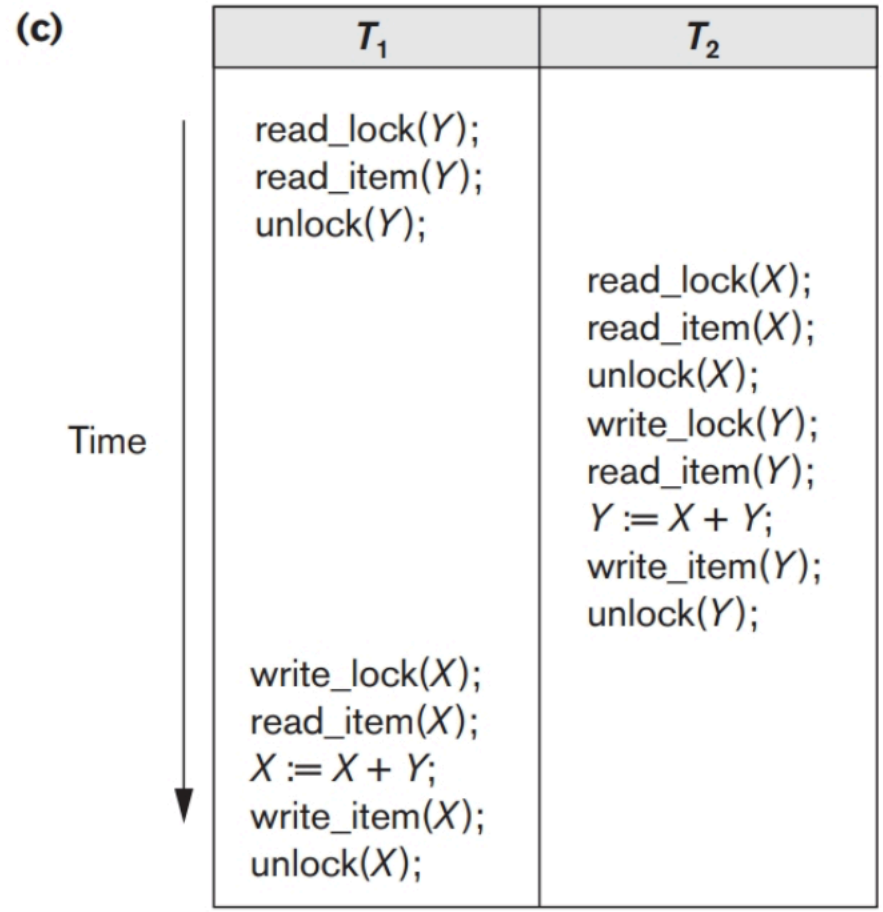
(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(a) Two transactions T1 and T2 (do not obey two-phase locking)
(b) Results of possible serial schedules of T1 and T2.

Two Phase Locking (2PL) Example



Initial values: X = 20, Y = 30

Result of schedule *S*:
X=50, *Y*=50
(nonserializable)

(c) A nonserializable schedule S that uses locks.

Two Phase Locking (2PL) Example

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Transactions T1′ and T2′, which are the same as T1 and T2 previously but follow the two-phase locking protocol.

Problems with Locking

Appropriate locking can guarantee correctness.

However, it also introduces potential undesirable effects:

- **Deadlock**

  No transactions can proceed; each waiting on lock held by another.

- **Starvation**

  One transaction is permanently "frozen out" of access to data.

- **Reduced performance**

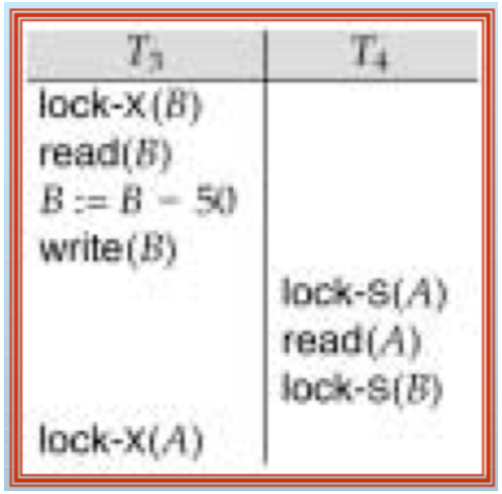
  Locking introduces delays while waiting for locks to be released.

Deadlock

Deadlock occurs when two transactions are waiting for a lock on an item held by the other.

Example:

```
T1                T2
--------------    ---------------
write_lock(X)
read(X)
                  write_lock(Y)
                  read(Y)
write_lock(Y)
waiting for Y     write_lock(X)
waiting for Y     waiting for X
```



T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Neither transaction can proceed … Such a situation is called a deadlock.

Deadlock Check

Create the wait-for graph for currently active transactions:

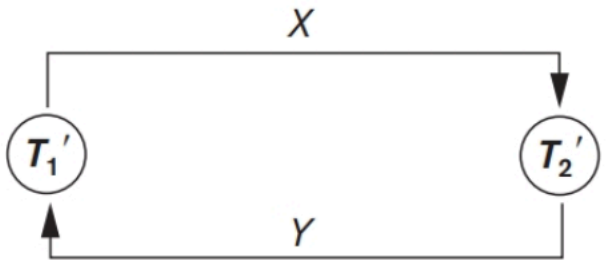
- create a vertex for each transaction; and
- create a directed edge from Ti to Tj, if Ti is waiting for an item locked by Tj.
- If the graph has a cycle, then a deadlock has occurred.

(a)

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Time

(b)



deadlock detection

periodically check for deadlocks, abort and rollback some transactions (restart them later).

Deadlock

```
T1              T2
--------------  ---------------
write_lock(X)
read(X)
                write_lock(Y)
                read(Y)
write_lock(Y)
waiting for Y   write_lock(X)
waiting for Y   waiting for X
```

Handling deadlock involves forcing a transaction to "back off".

- select a process to "back off"
  - choose on basis of how far transaction has progressed, # locks held, ...
- roll back the selected process
- prevent starvation
  - need methods to ensure that same transaction isn't always chosen

Deadlock prevention

Assign priorities based on timestamps (early start has higher priority).

Assume T1 wants a lock that T2 holds.

Two policies (strategies) are possible:

- "Wait-die": If T1 has higher priority, T1 waits for T2; otherwise T1 aborts
- "Wound-wait": If T1 has higher priority, T2 aborts; otherwise T1 waits

If a transaction re-starts, make sure it has its original timestamp

Looking at Performance

Locking typically reduces concurrency ⇒ reduces throughput.

Granularity levels: field, row (tuple), table, whole database

Granularity of locking can impact performance:

+ lock a small item ⇒ more of database accessible

+ lock a small item ⇒ quick update ⇒ quick lock release

- lock small items ⇒ more locks ⇒ more lock management

Multiple lock-granularities give best scope for optimising performance.

Concurrency Control in SQL

Transactions in SQL are specified by

BEGIN ... start a transaction

COMMIT ... successfully complete a transaction

** **ROLLBACK** ... undo changes made by transaction + abort

In PostgreSQL, other actions that cause rollback:

- **raise exception** during execution of a function
- returning null from a **before** trigger

Concurrency Control in SQL

Explicit control of concurrent access is available, e.g.

Table-level locking: **LOCK TABLE**

- various kinds of shared/exclusive locks are available
  - » **access share** allows others to read, and some writes
  - » **exclusive** allows others to read, but not to write
  - » **access exclusive** blocks all other access to table
- Row-level locking: **SELECT FOR UPDATE**, **UPDATE**, **DELETE**
  - allows others to read, but blocks write on selected rows

Some SQL commands automatically acquire locks

- » e.g. **ALTER TABLE** acquires an **access exclusive** lock

All locks are released at end of transaction (no explicit unlock)

PostgreSQL – Transactions, Concurrency

For more details on PostgreSQL's handling of these:

Chapter 12: Concurrency Control

SQL commands: BEGIN, COMMIT, ROLLBACK, LOCK, etc.

(for your reference only)