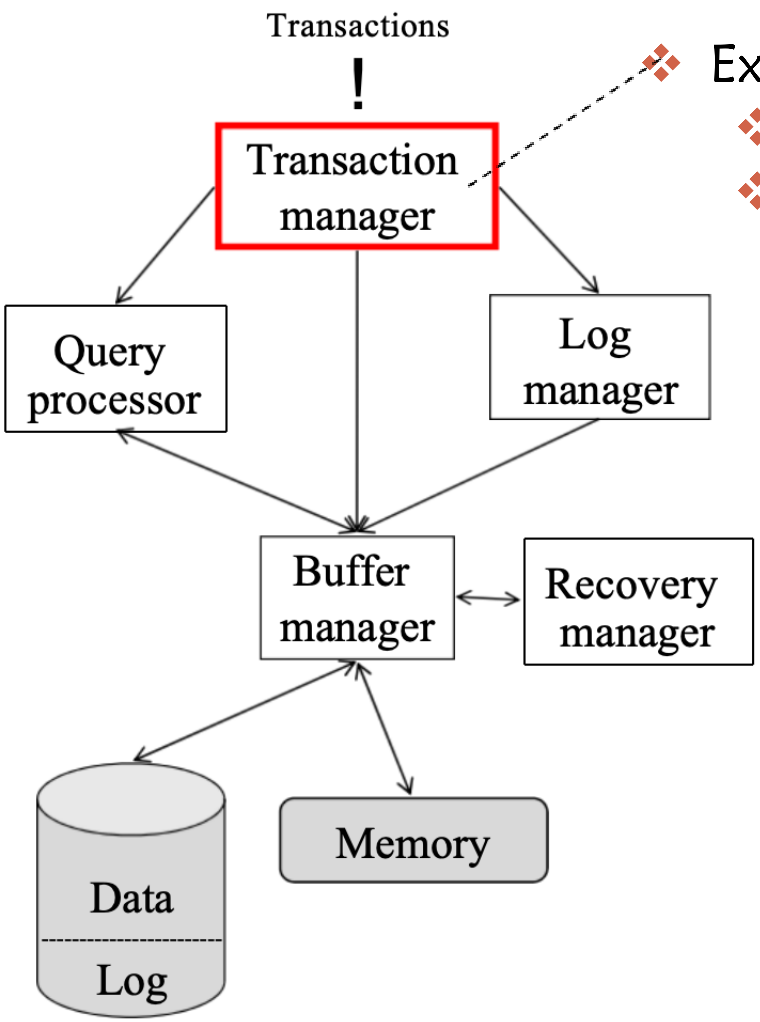
Concurrency Control

南方科技大学
唐 博
tangb3@sustech.edu.cn

Scheme



- Execute a concurrency control scheme
  - Control the interaction among transactions
  - Produce **serializable** schedule

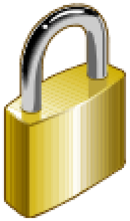
If No Concurrency Control

Problems

- Lost update
  - A transaction's update is overwritten by another transaction
- Dirty read
  - A transaction reads data which was produced by uncommitted transaction
- Non-repeatable read
  - A transaction reads the same data item twice (or more), but obtains different values each time
- Phantom read
  - A transaction runs the same query twice (or more), but obtains different results each time
    - E.g., caused by INSERT, DELETE from other transactions

Lecture Objectives

- Two-Phase Locking
- Problems of Locking



- Timestamp Ordering
- Optimistic Concurrency Control

Industry

FYI: What are used in the industry?

- Microsoft SQL Server supports
  - Locking
  - Optimistic concurrency control
    - https://technet.microsoft.com/en-us/library/ms189132(v=sql.105).aspx

- Oracle Database supports
  - Locking
    - https://docs.oracle.com/cd/B19306_01/server.102/b14220/consist.htm#i5702

Idea of Locking

- Idea: mutual exclusion
  - Avoid transactions with conflicting operations to execute concurrently
    - Do you still remember "*conflicting operations*"?
- Typically done by locking
  - Shared-lock (lock-S): a transaction must hold a lock-S on an item x before using Read(x)
  - Exclusive-lock (lock-X): a transaction must hold a lock-X on an item x before using Write(x)

Lock Conflict

- Two locks conflict if
  - they are on the *same item*,
  - issued by *different transactions*, and
  - at least one of them is an exclusive lock

- Compatibility matrix

	Lock-S	Lock-X
Lock-S	yes	no
Lock-X	no	no

Problem

Locking alone cannot avoid non-serializable schedule!

Anon-serializable schedule

T_1	T_2
R(x)	
	W(x)
	W(y)
R(y)	

with locking operations →

T_1	T_2
Lock-S(x)	
R(x)	
Unlock(x)	
	Lock-X(x)
	Lock-X(y)
	W(x)
	Unlock(x)
	W(y)
	Unlock(y)
Lock-S(y)	
R(y)	
Unlock(y)	

- We need a **locking protocol!**
  - a set of rules for requesting & releasing locks in transactions
  - to restrict the possible schedules

Basic Two-phase Locking

- The basic 2PL protocol
  - A transaction *T* must hold a lock on an item *x* in the appropriate mode before accessing *x*
  - *T* waits if a conflicting lock on *x* is being held by another transaction
  - Once *T* releases a lock, it cannot obtain any other lock subsequently

- It divides a transaction into two phases:
  - A *growing phase* (obtaining locks)
  - A *shrinking phase* (releasing locks)

Why Two Phases?

Under 2PL, this unlock operation cannot be placed here.

T_1	T_2
Lock-S(x)	
R(x)	
Unlock(x)	
	Lock-X(x)
	Lock-X(y)
	W(x)
	Unlock(x)
	W(y)
	Unlock(y)
Lock-S(y)	
R(y)	
Unlock(y)	

2PL makes sure that the conflicting operations of the two transactions are ordered the same way

Under 2PL, this is the earliest possible time to unlock x

Why Two Phases?

Red operations: have been executed

Black Operations: will consider in future

T_1	T_2
Lock-S(x)	
R(x)	
	Lock-X(x)
	Lock-X(y)
	W(x)
	Unlock(x)
	W(y)
	Unlock(y)
Lock-S(y)	
Unlock(x)	
R(y)	
Unlock(y)	

T_2 lock *x* operation cannot be granted at this point

T_2 can lock x after T_1 unlocks x

Why Two Phases?

T_1	T_2
Lock-S(x)	
R(x)	
Lock-S(y)	
Unlock(x)	
	Lock-X(x)
	Lock-X(y)
	W(x)
	Unlock(x)
	W(y)
	Unlock(y)
R(y)	
Unlock(y)	

T_2 lock *y* operation cannot be granted at this point

T_2 can lock y after T_1 unlocks y

Why Two Phases?

T_1	T_2
Lock-S(x)	
R(x)	
Lock-S(y)	
Unlock(x)	
	Lock-X(x)
R(y)	
Unlock(y)	
	Lock-X(y)
	W(x)
	Unlock(x)
	W(y)
	Unlock(y)

Lock Point

- The transaction's *lock point*
  - The time point when a transaction has obtained its final lock
- 2PL schedule is equivalent to a serial schedule where transactions are ordered by their lock points
- *Question*: Can you find a conflict serializable schedule that cannot be obtained via 2PL?
  - Yes. (leave to you as an exercise)

Recoverability and 2PL

- 2PL allows cascading roll-back
  - ☹ if an transaction is aborted, may need to undo many other transacations

- How to modify the 2PL protocol to avoid this drawback?

- **Strict two-phase locking**
  - A transaction must hold all its exclusive locks till it commits/aborts
  - Guarantees cascadeless schedules

Lock Conversion

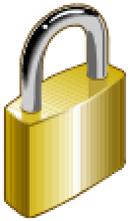
- If a transaction *T* reads an item *x* first then
- later writes *x*, what lock shall *T* acquire?
  - [1] *T* can acquire lock-X on *x* before it reads *x*
  - [2] *T* acquires lock-S on *x* before it reads *x*, then convert lock-S(*x*) to lock-X(*x*)
- The latter case [2] is called *lock conversion*
  - Allow more concurrency
  - The DBMS may not know whether a transaction that reads *x* will or will not write *x* later

2PL with Lock Conversions

- First Phase
  - can acquire a lock-S on item
  - can acquire a lock-X on item
  - can convert a lock-S to a lock-X (upgrade)
- Second Phase
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability

Lecture Objectives

- Two-Phase Locking
- Problems of Locking



- Timestamp Ordering
- Optimistic Concurrency Control

Starvation

- Any locking protocol has to deal with 2 problems: *starvation* and *deadlock*
- Starvation occurs when a lock request will *wait indefinitely*
  - A transaction is waiting for an X-lock on an item, while many other transactions request and are granted S-lock on the same item.
- How to prevent starvation?

Deadlock

- A deadlock occurs when there is a *circular wait* of transactions

T_1	T_2	T_3
Lock-S(x)		
R(x)		
	Lock-X(y)	
	W(y)	
Wait for Lock-S(y)		
		Lock-X(z)
		W(z)
		Wait for Lock-X(x)
	Wait for Lock-S(z)	

Blocked!

Blocked!

Blocked!

- When a deadlock occurs, the DBMS must abort (and later restart) a transaction in the deadlock
  - to release all locks held by the aborted transaction

Deadlock

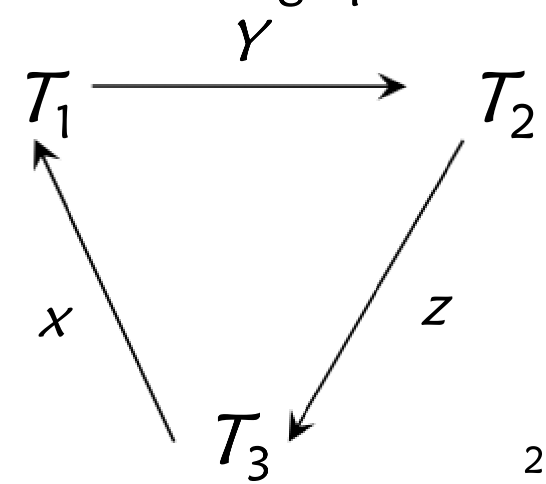
- Three solutions for solving deadlock
  - Maintaining the wait-for-graph
  - A timeout-based scheme
  - Priority-based scheme

Deadlock Detection

- A *wait-for graph*
  - A vertex denotes a transaction
  - A directed edge Ti → Tj means that Ti is waiting for Tj to release a data item
- **Insert an edge Ti → Tj in the graph**
  - When Ti requests a data item being held by Tj
- **Remove an edge Ti → Tj**
  - When Tj releases lock of a data item needed by Ti
- **A cycle in the wait-for graph == deadlock happens**
  - Must invoke a deadlock-detection algorithm periodically to detect cycles

T_1	T_2	T_3
Lock-S(x)		
R(x)		
	Lock-X(y)	
	W(y)	
Lock-S(y)		
		Lock-X(z)
		W(z)
		Lock-X(x)
	Lock-S(z)	

Wait-for graph:



Note: Don't mix up *precedence graph* and *wait-for-graph*!

Deadlock Recovery

- How to select a victim (to abort)?
  - Effort already invested in a transaction
  - Cost of aborting a transaction (e.g., will it cause cascading aborts?)
  - Effort to finish the transaction
  - Age of the transaction
  - Number of rollbacks (to avoid starvation)
- Rollback -- determine how far to roll back transaction
  - More effective to roll back transaction only as far as necessary to break deadlock

Timeout-based Scheme

- *Timeout* : abort a transaction if it has been waiting too long (for a lock)

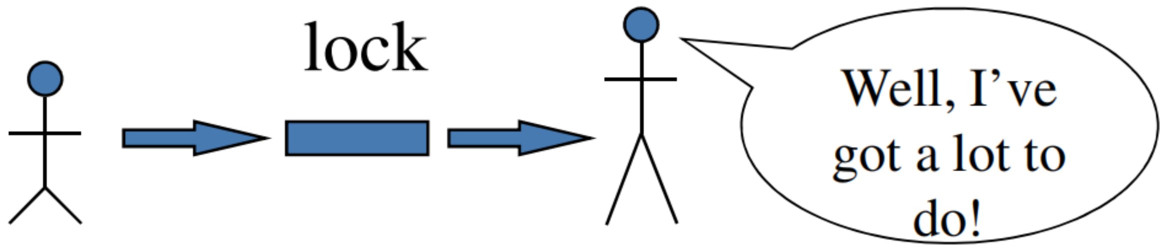
- The timeout period should be:
  - long enough so that most transactions that are aborted are actually deadlocked
  - short enough so that deadlocked transactions don't wait too long for their deadlocks to be broken

Deadlock Avoidance Schemes

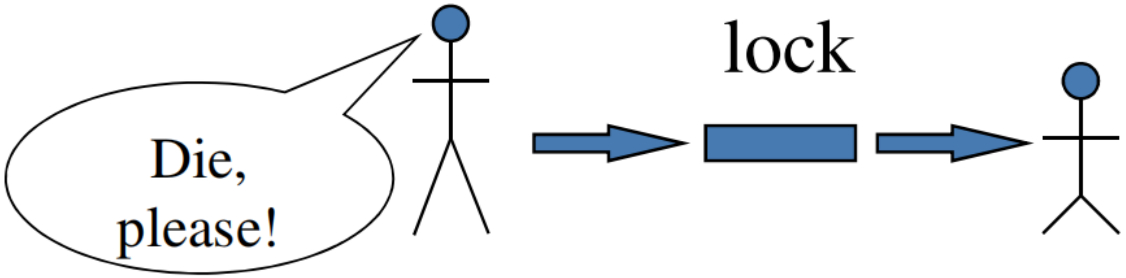
- Deadlock avoidance: prevent a transaction to block any transaction with "higher priority"
  - E.g., assign priorities to transactions based on transaction timestamps (e.g., start time)
- If a transaction Ti requests a lock held by a transaction Tj ……
- *Wait-die* scheme:
  - Ti waits if Ti.timestamp < Tj.timestamp
  - Ti is rolled back otherwise
- *Wound-wait* scheme:
  - Ti waits if Ti.timestamp > Tj.timestamp
  - Tj is rolled back otherwise

Wait-die vs. Wound-wait

- Wait-die: a young transaction may get rolled back repeatedly



- Wound-wait: there may be fewer rollbacks



- With wound-wait, an old transaction never waits.

Cyclic Restart

- A *cyclic restart* occurs when a transaction is repeatedly rolled back without making progress
- Both wait-die and wound-wait prevent cyclic restarts because:
  - The oldest transaction never gets rolled back
  - The oldest transaction eventually gets to execute to completion
  - A transaction will eventually become the oldest transaction in the system
- *Question*: when a transaction is restarted, shall we use its old timestamp, or get a new timestamp?

Summary of These Schemes

- The wait-for-graph (WFG)
  - its maintenance requires too much resource, especially if deadlock occurs rarely
- A timeout-based scheme
  - easy to implement
  - E.g., if an application does not receive any terminal input before the timeout, it may roll back a transaction
- Priority-based schemes
  - may cause many unnecessary rollbacks

Lecture Objectives

- Two-Phase Locking
- Problems of Locking

- Timestamp Ordering
- Optimistic Concurrency Control

Timestamp Ordering

- Each transaction *T* is given a *timestamp* TS(*T*)
  - An old transaction (with small timestamp) has high priority

- Given two transactions *Ti* and *Tj*, if TS(*Ti*) < TS(*Tj*), we want a schedule *S* such that *S* is equivalent to a serial schedule *S'* with *Ti* before *Tj*

- Maintain 2 timestamps for each data item *Q*:
  - W-TS(*Q*): the largest timestamp of any transaction that has executed write(*Q*)
  - R-TS(*Q*): the largest timestamp of any transaction that has executed read(*Q*)

Timestamp Ordering Protocol

- **TOP** Idea: ensure that conflicting operations are executed in timestamp order
  - Note: a restarted transaction will be given a *new timestamp*
- On a transaction T issuing read(Q):
  - TS(T) < W-TS(Q): T issues read(Q) too late ⇒ restart T
  - TS(T) > W-TS(Q): execute read(Q) and update R-TS(Q) to max{R-TS(Q),TS(T)}

time	T_1	T_2
1	*Start*	
2		*Start*
3		Write(Q)
4	Read(Q)	

$TS(T_1)=1$ $W\text{-}TS(Q)=2$

time	T_1	T_2
1		*Start*
2	*Start*	
3		Write(Q)
4	Read(Q)	

$TS(T_1)=2$ $W\text{-}TS(Q)=1$

Timestamp Ordering Protocol

- On a transaction T issuing write(Q):
  - If TS(T) < R-TS(Q) or TS(T) < W-TS(Q)
    T issues write(Q) too late ⇒ restart T
  - Else: execute write(Q),
    update W-TS(Q) to max{W-TS(Q),TS(T)}

time	T_1	T_2
1		*Start*
2	*Start*	
3	Read(Q)	
4		Write(Q)

$TS(T_2)=1$ $R\text{-}TS(Q)=2$

time	T_1	T_2
1		*Start*
2	*Start*	
3	Write(Q)	
4		Write(Q)

$TS(T_2)=1$ $W\text{-}TS(Q)=2$

TOP Properties

- Ensure conflict serializability ☺
  - We skip the proof
- Deadlock free ☺
- May lead to starvation of long transactions ☹
- Schedules can be un-recoverable ☹

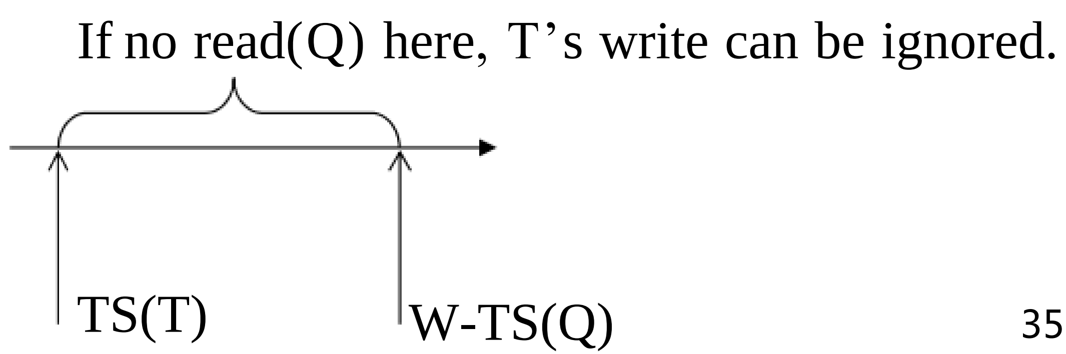
TOP: Managing Timestamps

- Large space requirement ☹
  - need 2 timestamps for each data item
- Be careful when we manipulate timestamps
  For example:
  - Store timestamps on non-volatile storage or not?
  - When a transaction aborts, do we rollback a timestamp's value or not?
  - A read operation causes a write to a data item's timestamp

Thomas Write Rule

- In TOP, when a transaction T issues a write(Q), if TS(T) < W-TS(Q), then restart T
- However, restarting T is really unnecessary
- <u>If no transaction should have read T' s value</u>, then ignore T' s write

How to check this condition?



Thomas Write Rule

- TOP with *Thomas write rule* may produce schedules that are view serializable but not conflict serializable
- Example:

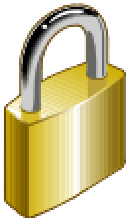
Time	T_1	T_2
1	*Start*	
2		*Start*
3	Read(Q)	
4		Write(Q)
5	Write(Q)	

$TS(T_1)=1$
$TS(T_2)=2$

at this point → $R\text{-}TS(Q)=1$, $W\text{-}TS(Q)=2$

Lecture Objectives

- Two-Phase Locking
- Problems of Locking



- Timestamp Ordering
- Optimistic Concurrency Control

Optimistic Concurrency Control (OCC)

- Both 2PL and TOP have bookkeeping overhead for each read/write operation.
  - Wasteful if conflicts occur rarely in the system

- Also, lock-based protocols resolve conflicts by blocking
  - Could lead to severe *data contention thrashing*:

    adding more transactions to the system

  ->more transactions are waiting for locks

  ->transactions thus spend time waiting rather than working

  ->decreases the system’s throughput

Optimistic Concurrency Control

- Optimistic concurrency control *assumes that conflicts don't happen frequently*
  - Allow transactions to proceed first
  - When a transaction commits, the OCC scheduler verifies that the transaction is not in any conflict

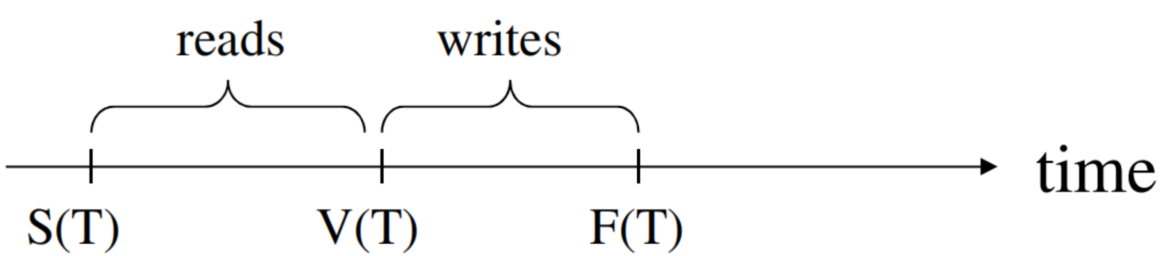
- OCC would be efficient if most of the transactions are **read-only**

The OCC Protocol

- OCC requires each transaction to declare its *ReadSet* and *WriteSet* before execution
- A transaction T is divided into 3 phases.
- *Read phase*:
  - T reads into local memory the data items it reads/writes
  - Writes to these data items are done only to the local memory
- *Validation phase*:
  - When T is ready to commit, the scheduler checks if T conflicts with any other transactions
  - If so, the scheduler rejects T’s commit and restarts T
- *Write phase*:
  - T passes its validation test
  - The values T wrote to the local copies are copied to the database

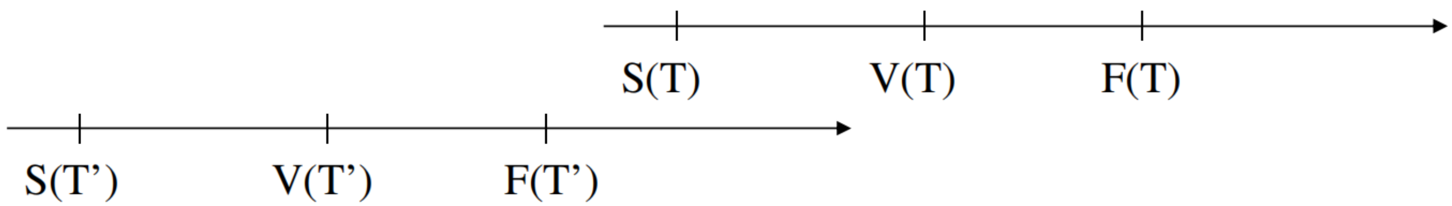
OCC: Validation

- Each transaction T is associated with 3 timestamps:
  - Start(T): start time of T
  - Validation(T): time at which T starts its validation
  - Finish(T): time at which T completes its write phase
- *Goal of the validation test*: order transactions with conflicting operations based on their validation times



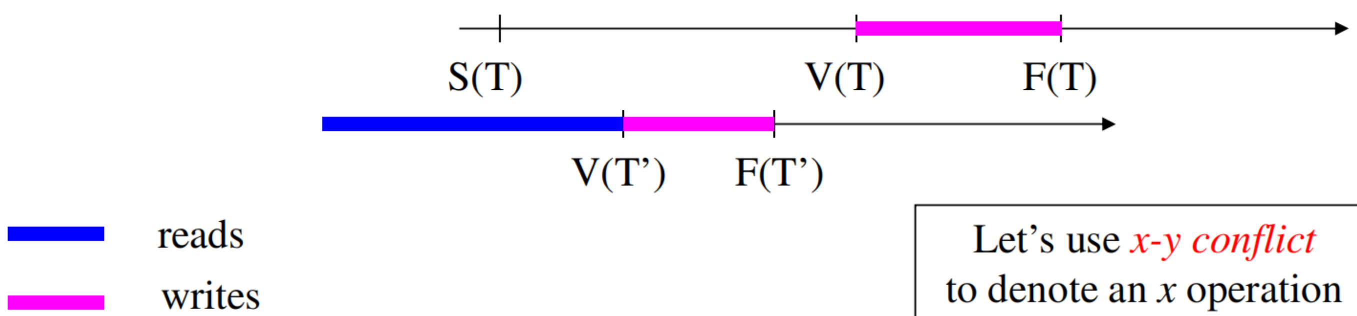
OCC: Validation

- Consider two transactions *T* and *T'* such that validation(*T'*) < validation(*T*).
- OCC would order conflicting operations of *T* and *T'* such that those of *T'* precede those of *T*.
- Consider validating *T*:
  - Case 1: Finish(*T'*) < Start(*T*)
    - All conflicting operations (if exist) are ordered properly



OCC: Validation

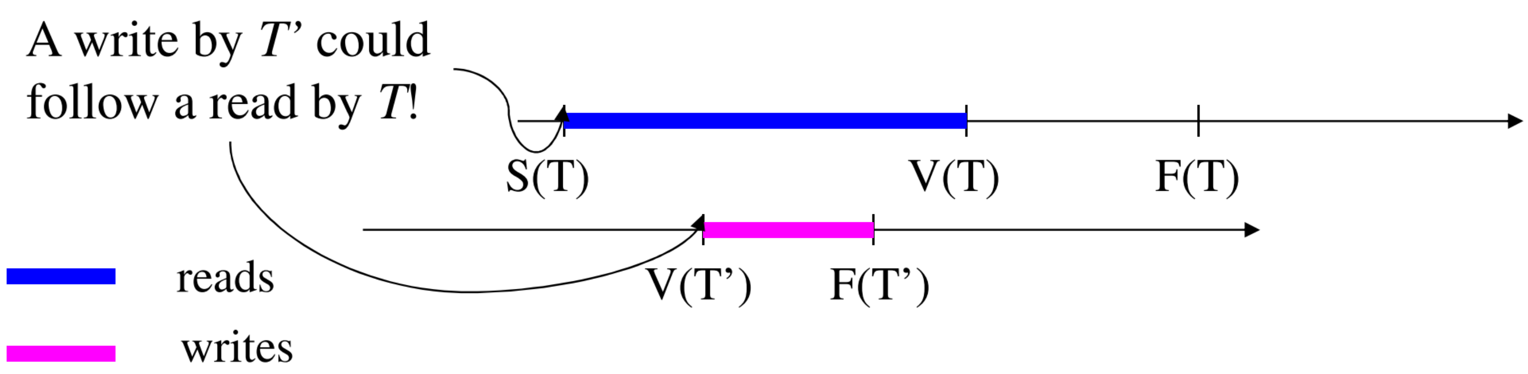
- Case 2: Start(T) < Finish(T') < Validation(T)
  - Since all reads by T' occur before V(T') and all writes by T occur after V(T) and V(T') < V(T), any *read-write conflicts* are ordered properly.
  - Since all writes by T' occur before F(T') and all writes by T occur after V(T) and F(T') < V(T), any *write-write conflicts* are ordered properly



Let's use *x-y conflict* to denote an *x* operation executed by *T'* and a *y* operation executed by *T*

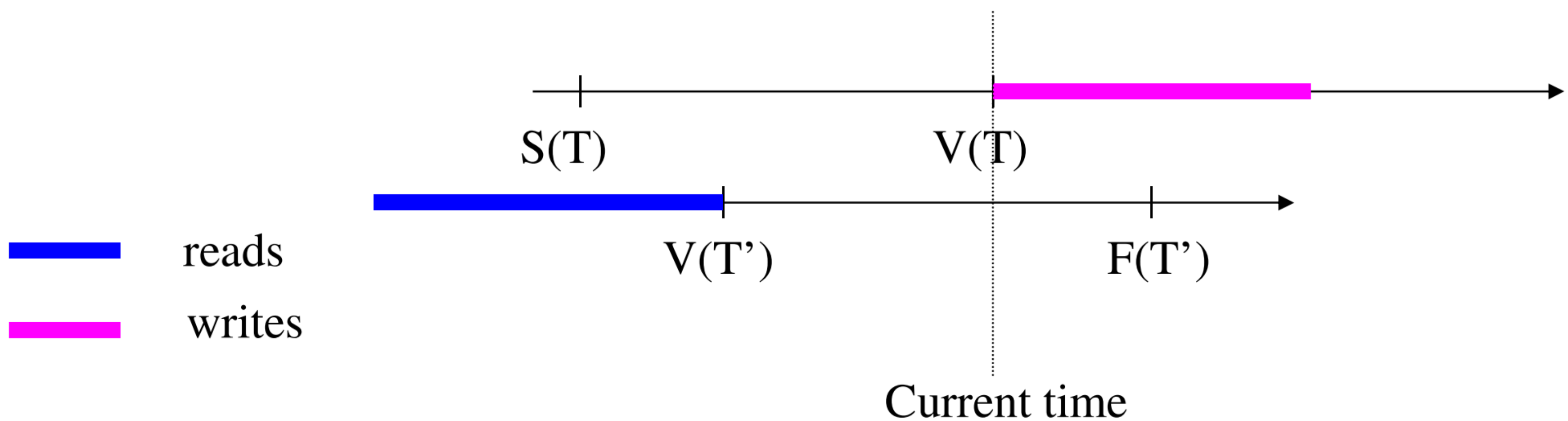
OCC: Validation

- Case 2: Start(T) < Finish(T') < Validation(T)
  - However, *write-read conflicts* may violate the (intended) serialization order.
  - Solution: if ReadSet(T) ∩ WriteSet(T') ≠ φ, abort T.



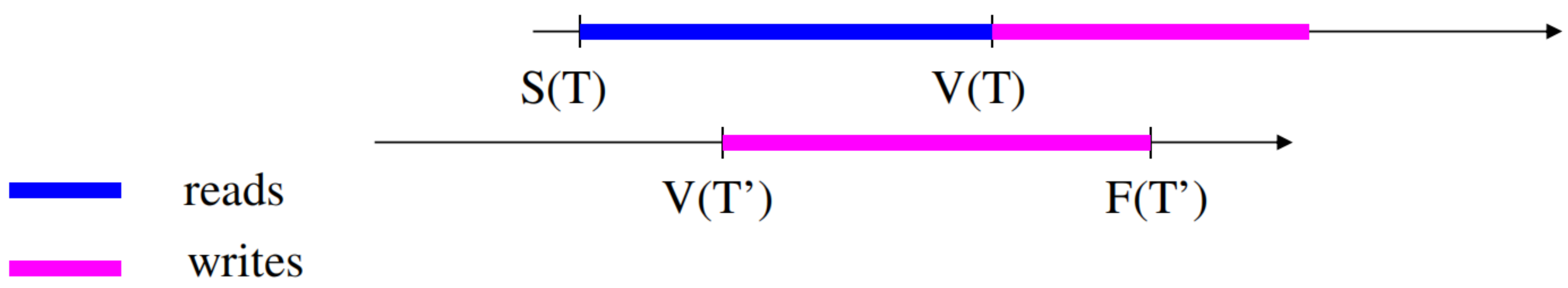
OCC: Validation

- Case 3: Validation(T) < Finish(T')
  - Technically, the system does not know F(T') yet, since it is validating T. The system, however, knows that F(T') > V(T) because T' has not finished its write phase.
  - Since all reads by T' occur before V(T') and all writes by T occur after V(T) and V(T') < V(T), any *read-write conflicts* are ordered properly.



OCC: Validation

- Case 3: Validation(T) < Finish(T')
  - *Write-write conflicts* could violate the (intended) serialization order.
    - Solution: if WriteSet(T) ∩ WriteSet(T') ≠ ϕ, abort T.
  - *Write-read conflicts* could violate the (intended) serialization order.
    - Solution: if ReadSet(T) ∩ WriteSet(T') ≠ ϕ, abort T.



OCC: Validation Rules

- If Finish(T') < Start(T)
  - T passes its validation
- If Finish(T') > Start(T)
  - Write-read conflicts could violate the serialization order
  - Write-write conflicts could violate the serialization order if Finish(T') > Validation(T)
- To validate a transaction T, **check** these two rules for every transaction T' with Validation(T') < Validation(T)
  - Rule 1: if Finish(T') > Start(T) then
    if ReadSet(T) $\cap$ WriteSet(T') $\neq \phi$, abort T
  - Rule 2: if Finish(T') > Validation(T) then
    if WriteSet(T) $\cap$ WriteSet(T') $\neq \phi$, abort T

Conclusions

Two classes of concurrency control protocols:

- Pessimistic (e.g., 2PL and TOP)
- Optimistic (e.g., OCC)
  - performs well for read-intensive transactions
  - but validation and restart are more costly

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

谢谢！

DBGroup @ SUSTech
Dr. Bo Tang (唐博)
tangb3@sustech.edu.cn