

Concurrency Control in Distributed Systems

A thick, horizontal yellow brushstroke with a textured, painterly appearance, spanning most of the width of the slide.

ECE 677

University of Arizona

Agenda



What?

Why?

Main problems

Techniques

Two-phase locking

Time stamping method

Optimistic Concurrency Control

Why concurrency control?



Distributed systems combine both distribution and integration

CC problems arise when a computer program's software, data and interface are distributed over several computers.

Probability of conflict increases

Conflict causes damage to the system consistency and integrity

Why is concurrency control needed?



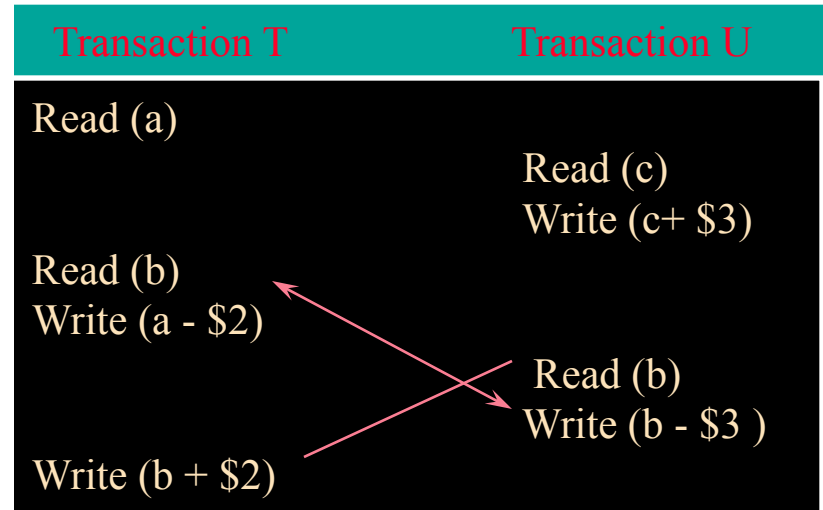
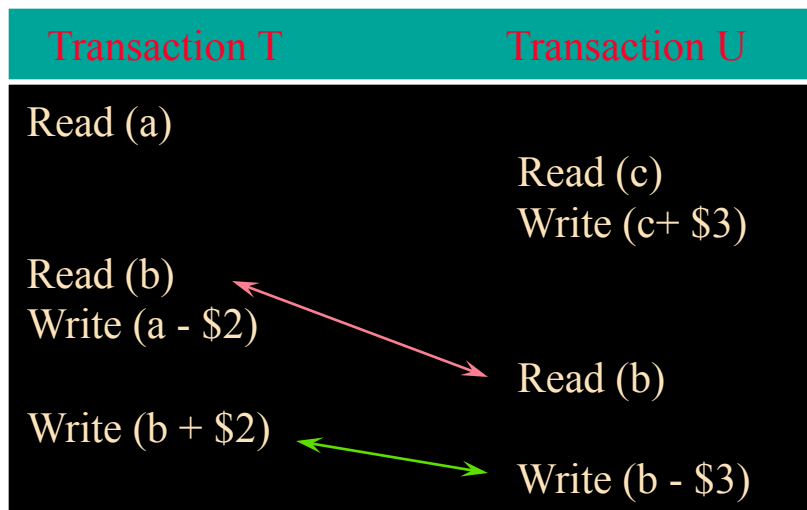
Temporary update problem : When a transaction fails after updating some data items and another transaction accesses them before it can be changed to its original value.

Lost update problem : Two transactions operate on a data item at the same time. Suppose one is reading data before the other transaction updates the value.

The Need for Transactions

Lost Update Problem \Rightarrow needs transactions to be serializable

When transactions execute concurrently without synchronization the exact sequence of reads and writes is not predictable



Possible interleaving of T and U

Serial Equivalence (Serializability)

The following interleaving of transactions has the same effect as serial execution

Transaction T	Transaction U
Read (a)	Read (c)
Read (b)	
Write (a - \$2)	Write (c+ \$3)
Write (b + \$2)	Read (b) Write (b - \$3)

How Do We Achieve Concurrency Control?

Serial Equivalence (Serializability) (cont)



How Do We Achieve Concurrency Control?

- ◆ locking: each data item has a lock which is used to provide mutual exclusive access to that data item
- ◆ optimistic control: perform all operations until the end, commit transaction if no conflict, otherwise abort it
- ◆ timestamps: each transaction and data item are timestamped every time they are accessed timestamps enforce serializable execution

Transactions



Database term.

Execution of program that accesses a database.

In distributed systems,

Concurrency control in the client/server model.

From client's point of view, sequence of operations executed by server in servicing client's request in a single step.

Transaction Properties



ACID:

Atomicity.

Consistency.

Isolation.

Durability.

Transaction Atomicity



“All or nothing”.

Sequence of operations to service client's request are performed in one step, i.e., either all of them are executed or none are.

Issues:

Multiple concurrent clients: “isolation”.

Server failures: “failure atomicity”.

Transaction Features



Recoverability: server should be able to "roll back" to state before transaction execution.

Serializability: transactions have same effect whether executed concurrently or sequentially.

Durability: effects of transactions are permanent.

The Transaction Service

File service that supports atomic transactions on its files

Transaction service operation

OpenTransaction -> *Trans*

start a new transaction and delivers a unique TID *Trans*. This identifier will be used in the other operations in the transaction.

CloseTransaction(Trans) -> (*Commit*, *Abort*)

ends a transaction: a *Commit* returned value indicates that the transaction has committed: an *Abort* returned value indicates that it has aborted.

AbortTransaction (Trans)

aborts the transaction.

TWrite(Trans, File, i, Data) - *REPORTS(BadPosition)*

has the same effect as *Write(File, i, data)* but records the new data in a tentative form pending the completion of the transaction *Trans*.

The Transaction Service (cont)

Transaction service operation (cont)

TRead(Trans, File, i, n) -> Data - REPORTS(BadPosition)

delivers the tentative data resulting from the transaction *Trans* if any has been recorded, otherwise has the same effect as *Read(File, i, n)*.

TCreate(Trans, filetype) -> File

records a tentative *Create* pending the completion of the transaction *Trans*.

TDelete(Trans, File)

records a tentative *Delete* pending the completion of the transaction *Trans*.

TTruncate(Trans, File)

records a tentative *Truncate* pending the completion of the transaction *Trans*.

TLength(Trans, File) -> Length

delivers the tentative new length resulting from the transaction *Trans* if any has been recorded. otherwise has the same effect as *Length*.

Transaction File Service



Transaction module : keeps transaction record including phase	
File module : relate file IDs to particular files	
File access module : reads or writes the file data or attributes (uses file index)	
Concurrency control module	Recovery module : makes private tentative data items

Block service

Stable storage

Block module : access and allocation of disk blocks	Stable storage module : access and allocation of stable blocks
Device module : disk I/O and buffering	

Concurrency Control Techniques



Run transactions one at a time

total elimination of concurrency is neither acceptable nor necessary

Run transactions concurrently if they use different file items and sequentially when they use the same file items

difficult to predict which items to be used by a transaction

two transactions may use the same item for a short period

Alternative methods:

locking

optimistic concurrency control

timestamps

Concurrency Control Techniques



Locking

A lock guarantees exclusive use of a data item to a current transaction. A transaction must claim a read (shared) lock or write (exclusive) lock on a data item prior to data access.

Time stamping

Use time to order concurrent access to shared data items

Optimistic methods

Assumption: the majority of the system operations do not conflict

All processes can concurrently access data, but before any update is saved, a check is made to see if any concurrent access has taken place.

Locks



Order transactions accessing shared data based on order of access to data.

Lock granularity: affects level of concurrency.

1 lock per shared data item.

Read (shared) locks and write locks.

Lock Implementation



Server lock manager

Maintains table of locks for server data items.

Lock and *unlock* operations.

Clients *wait* on a lock for given data until data is released; then client is *signaled*.

Each client's request runs as separate server thread.

Deadlock



Use of locks can lead to deadlock.

Deadlock: each transaction waits for another transaction to release a lock forming a wait cycle.

Deadlock condition: cycle in the wait-for graph.

Deadlock prevention and detection.

Deadlock resolution: lock timeout.

Basic Synchronization Primitives



Locks: a transaction can lock a data item in two modes:

Exclusive Mode: no other transaction can concurrently lock the data item

Shared Mode: other transactions can lock the data item

Timestamps: a unique number assigned to each transaction that is monotonically increasing and unique across the system

Lamport proposed a 2-tuple timestamp:

$ts_i = (C_i, i)$; C_i is a logical clock and i denotes the site

Locks



Locks on data items ensure that only one client at a time may access each data item

Locks and waiting are used to force serial equivalence on clients

a transaction that needs to access a locked item must wait until its lock is released

which signals the condition variable associated with a waiting transaction

Lock can be defined as a record with

binary variable (locked or not)

condition variable (wait or signal)

TID that has set the lock

Locking Schemes



Static Locking

a transaction acquires locks on all the objects it needs before executing any action on the data objects

it unlocks all the locked objects after it has executed all of its actions

Drawbacks of Static Locking



This approach is simple

it limits concurrency

It needs to define a priori all the objects that need to lock;

This might not be known at the beginning of the transaction

Two-Phase Locking Scheme



- ◆ It is a dynamic locking scheme in which a transaction requests a lock on a data object when it needs the data object.
- ◆ This protocol imposes a constraint on lock acquisition and the lock releases actions to guarantee consistency
- ◆ The protocol has two phases: a growing phase and a shrinking phase.
- ◆ Two-phase locking increases concurrency over static locking

Two-Phase Locking Scheme- Cont.



During the first phase (growing phase)
new locks are acquired

During the second phase (shrinking phase)
locks are released

Locks must not be released until the
transaction is either committed or
aborted

Two-phase Locking

Use of locks

- When a client operation accesses an item within a transaction :
 - (a) If the item is not already locked, the server locks it and proceeds to access the data for the client
 - (b) If the item is already locked for another transaction, the client must wait until it is unlocked.
 - (c) If the server has already locked the item in the same transaction, it can proceed to access the item.
- When a transaction is committed or aborted, the server unlocks all items, it locked for the transaction.

To ensure adherence to these rules, users have no access to locking operations; they are performed by the transaction service

Locks (cont)

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
OpenTransaction		OpenTransaction	
TRead(a)	locks a	TRead(c)	lock c
		TWrite(c+\$3)	
TRead(b)	lock b	Tread(b)	wait
TWrite(a-\$2)			
TWrite(b+\$2)			
CloseTransaction	unlock a and b		lock b
		TWrite(b-\$3)	
		CloseTransaction	unlock b and c

Read and Write Locks:

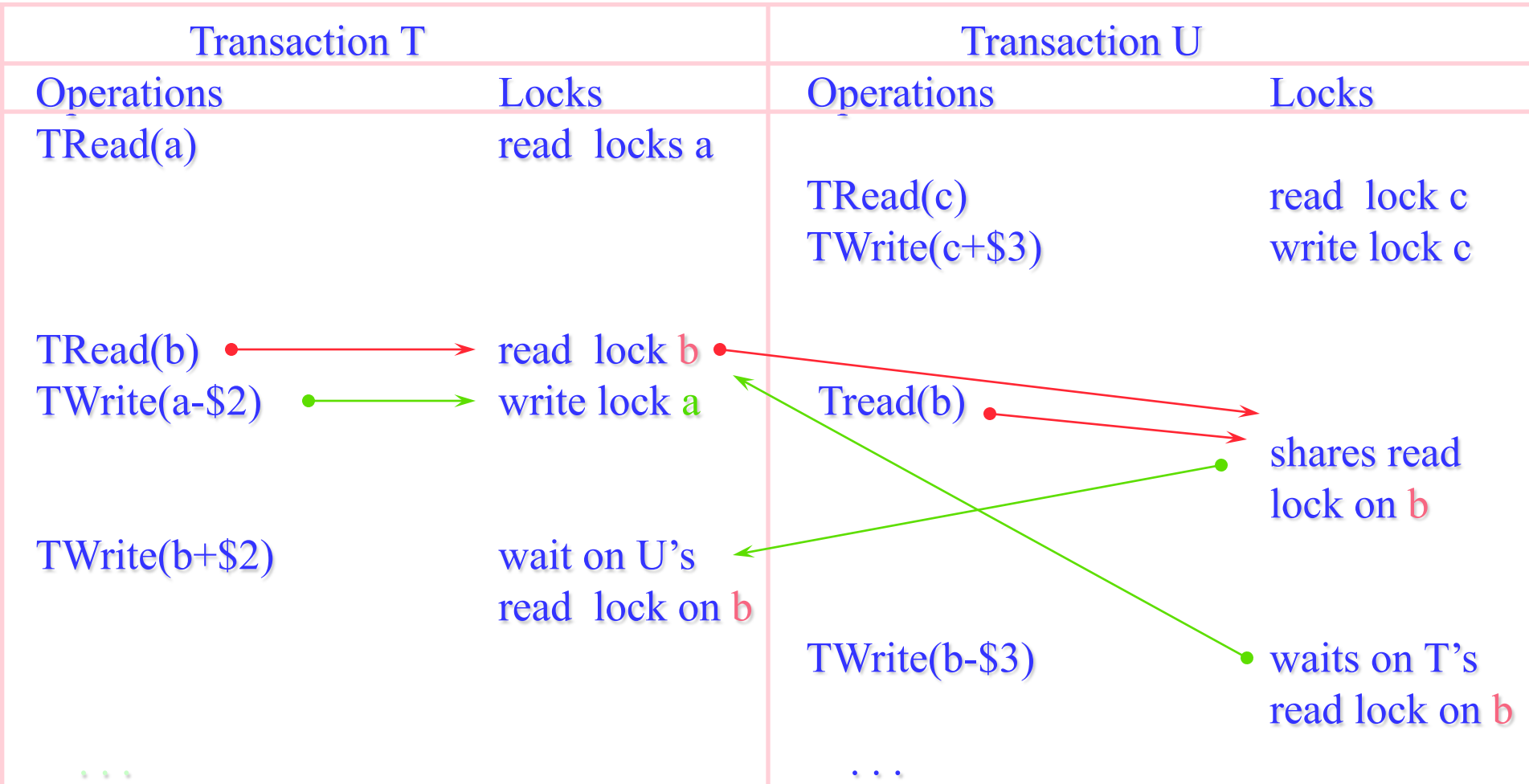
Simple lock for both Tread and Twrite operations reduces concurrency more than necessary

It is Okay to have several concurrent transactions reading an item, or a single T writing an item, but not both (many reader/single writer scheme)

Lock already set	Lock to set	
	Read	Write
None	ok	ok
read	ok	wait
write	wait	wait

If the transaction is writing to an item on which it has previously placed a read lock, the read lock is converted to a write lock

Read and Write Locks: (cont)



Intention-to-write locks

the existence of a read lock prevents any other transaction from writing the locked data item

we could allow a transaction to proceed with its tentative writes until it commits, it might be aborted so no change has occurred to the locked data item

T's operation	U's operation		
	read	I-write	Commit
none	ok	ok	ok
read	ok	ok	wait
I-write	ok	wait	wait
commit	wait	wait	wait

Intention-to-Write Lock

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
TRead(a)	read lock a	TRead(c)	read lock c
		TWrite(c+\$3)	I-write lock c
TRead(b)	read lock b		
TWrite(a-\$2)	I-write lock a	TRead(b)	shares read lock on b
TWrite(b+\$2)	I-write lock b		
...		TWrite(b-\$3)	waits on T's I-write lock b
...		...	
CloseTransaction	wait on U's read lock on b		

Drawbacks of 2P-Lock Scheme



1. Deadlocks

it is prone to deadlocks because a transaction can request a lock while holding locks on other object; a deadlock occurs when a set of transactions are involved in a circular wait

2. Cascaded roll-backs

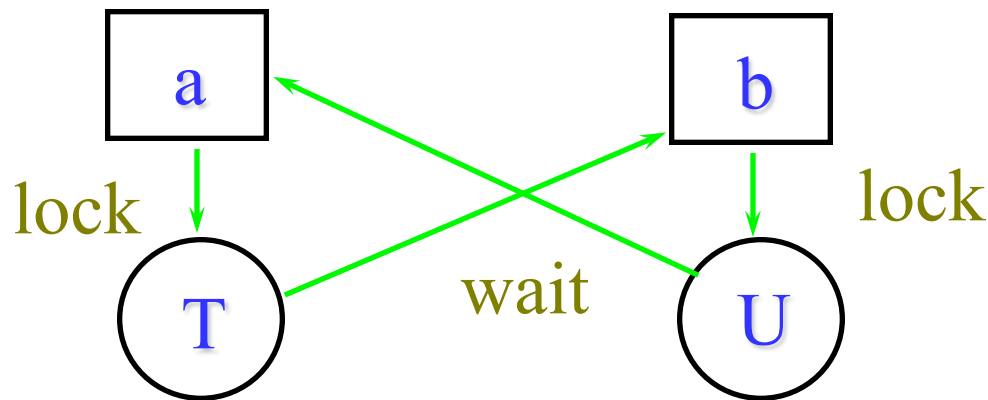
when a transaction is rolled back, all the data objects modified by it are restored to their original states.

This property might lead to roll back another set of transactions and so on.

Deadlock Resolution

Deadlock occurs when each one in a group is waiting for some other member to release a lock

Deadlock detection is based on a wait-for graph containing transactions and locks in the nodes



The wait-for graph

Deadlock Resolution (cont)



Timeouts: each lock is given a period in which it is invulnerable, after that period, it becomes vulnerable and its locks can be released in overloaded system

- timeout increases

- long time T can be penalized

- it is hard to decide on timeout interval

Deadlock Resolution (cont)



If there are timeouts on locks, the rules for converting an I-write lock to a commit lock are :

1. If another process has a vulnerable read lock, the server breaks the vulnerable read lock and converts the I-write lock to a commit lock
2. If the I-write lock is vulnerable and another process has read lock that is not vulnerable, the server aborts the transaction owning the I-write lock
3. If neither the I-write lock nor the read lock of another process is vulnerable, the server waits until one of the two previous cases occurs

Age Group	Percentage
18-24	~10%
25-34	~35%
35-44	~25%
45-54	~20%
55-64	~15%
65-74	~10%
75-84	~5%
85+	~2%

Read, T-write and commit locks in T and U

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
TRead(a)	read locks a	TRead(c)	read lock c
		TWrite(c+\$3)	I-write lock c
TRead(b)	read lock b	TRead(b)	shares read lock on b
TWrite(a-\$2)	I-write lock a		
TWrite(b+\$2)	I-write lock b	TWrite(b-\$3)	waits on T's I-write lock b
...		...	
CloseTransaction	wait on U's read lock on b		
	commit lock b		read lock on b now vulnerable abort U

Timestamp-based Locking

when a transaction is submitted, it is assigned a unique timestamp that defines a total order of transactions and can be used to resolve conflicts among transactions

The use of time stamps prevents deadlocks.

A conflict occurs when

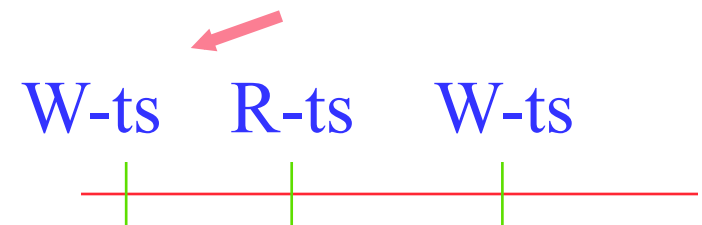
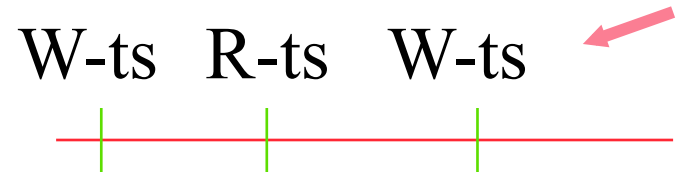
- a transaction makes a read request for a data object, for which another transaction currently has a write access

- or a transaction makes a write request for a data object for which another transaction has a write or read access

Timestamps (Write Rules)

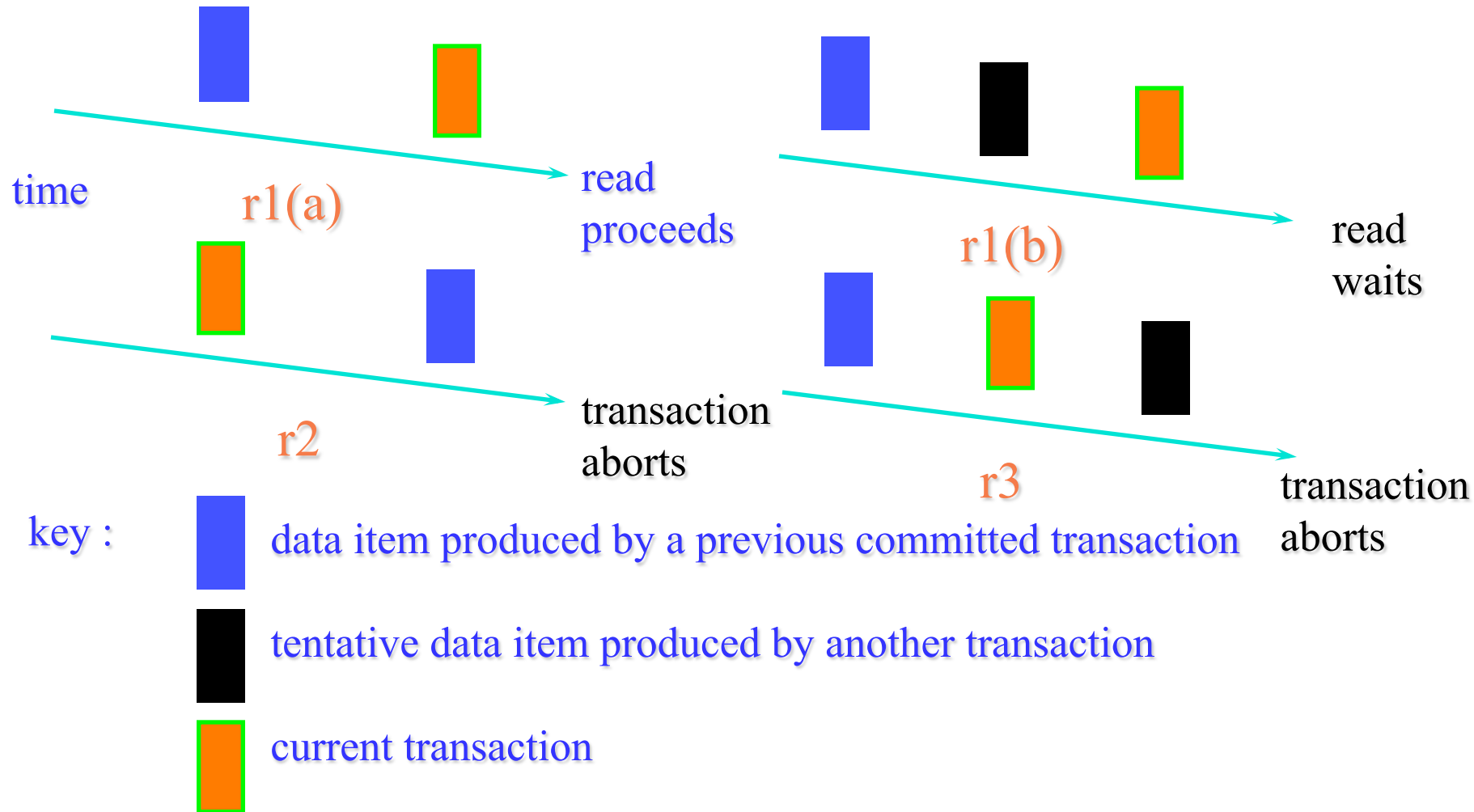
Conflicts occur when

T's	U's	
	read	write
read	ok	no
write	no	no



Write Operation Rules:

1. The timestamp of the current transaction is more recent than the read and (committed) write timestamps of the data item. A tentative write operation is performed
2. The timestamp of the transaction requesting the write is older than the timestamp of the last read or committed write of the data item. This implies that the write is arriving too late - another transaction has accessed the item since the current transaction started - and the current transaction is aborted



Read/Write Operation Rules

$W_{\neg ts} \ R_{\neg ts} \ W_{\neg TS}$ ←

← $W-TS$ $R-ts$ $W-ts$

T	U	Timestamps					
		ta		tb		tc	
		read	write	read	write	read	write
TRead(a)		tr					
	TRead(c)					tr	
	Twrite(c+\$3)						tw
TRead(b)				tr			
TWrite(a-\$2)			tr	Ur			
	TRead(b)			Tw			
Twrite(b+\$2)							
Abort							
	Twrite(b-\$3)				Uw		

Timestamps in transactions T and U

Timestamp Locking-Continue



Conflict Resolution


WAIT: wait until the conflicting transaction either completes or aborts

it is resolved by taking one of the following actions:

Restart: Either the requesting transaction or the one it conflicts with is aborted and started afresh;

restart is achieved using either Die or Wound

Conflict Resolution- Restart Operation



a) **Die**: the requesting transaction aborts and starts a fresh

b) **Wound**: the transaction in conflict with the requesting T is tagged as Wound transaction.

This message is then broadcasted to all sites visited by the wound transaction. If these messages are received before the **wound transaction commits**, then the transaction is **aborted**; otherwise the wound messages are ignored

The requesting transaction proceeds after the wound transaction completes or aborts

Wait-Die Algorithm



It is a nonpreemptive algorithm because a requesting transaction never forces the transaction holding the requested data object to abort

Conflict: Read while older Ts doing write Or

Write while older Ts did Read or Write

If Requesting T1 is in conflict with Transaction T2

If T1 is older, then T1 waits

otherwise T1 (newer) dies and restarts afresh

It prefers older transactions

Wound-Wait Algorithm



it is preemptive algorithm

if T1 is in conflict with T2

If T1 is older, it wounds T2; then
broadcast wound message;

if received before commit T2 at a
site, T2 aborts

otherwise, T1 is newer, it waits

Comparison Between the Algorithms



Waiting Time:

The wait-die makes older transactions wait for younger transactions

The opposite occurs in wound-wait algorithm; older transactions never waits for younger ones and wound those in conflict with older ones

The older a transaction becomes, the less it waits

Number of Restarts:

In wait-die algorithm, a younger transaction might die and restart several times before it completes

In wound-wait algorithm, if the requester is younger, it waits rather than continuously dying and restarting

Timestamp-based Algorithms



Every site maintains a logical clock that is incremented when a transaction is submitted and is updated whenever it receives a message with a higher clock value.

Timestamps are used in two ways:

1. determine the currency and outdateness of a request with respect to the data object is operating on
2. order read-write requests with respect to one another

Basic Timestamp Ordering (BTO) Algorithm

keeps track of the largest timestamp of any read or write processed so far for each data object

$R\text{-ts}(x)$, $W\text{-ts}(x)$

Read Request $\text{read}(x, TS)$:

if $TS < W\text{-ts}(x)$, it is rejected and its T is aborted

otherwise, it is executed and $R\text{-ts}(x)$ is set to $\max \{R\text{-ts}(x), TS\}$

Write Request $\text{write}(x, v, TS)$

if $TS < R\text{-ts}(x)$ or $TS < W\text{-ts}(x)$, then it is rejected.

otherwise it is executed, and $W\text{-ts}(x)$ is set to TS

Multiversion Timestamp Ordering Algorithm

a history of a set of R-ts , and $\langle W\text{-ts, value} \rangle$ pairs (versions) is kept for each data object

a Read request read (x, TS) is executed by reading the version of x with the largest timestamp less than TS and then adding TS to the x's set of R-ts's

a write(x,v, TS) request is executed as follows:

if there exists a R-ts(x) with stamp larger than TS in the interval between TS and the previous write timestamp, the write request is rejected

otherwise, $TS > R\text{-ts}(x)$, it is accepted



Conservative Timestamp Ordering Algorithm



it eliminates aborts and restarts of transactions by executing the requests in strict timestamp order.

A scheduler processes a request when it is sure that there is no other request with a smaller (older) timestamp in the system

The scheduler maintains two queues R and W-queues

Conservative Ordering Algorithm-continue

1. A $\text{read}(x, TS)$ request is executed if every W -queue nonempty and the first write on each W -queue has a timestamp larger than TS , the read is executed; otherwise the read request is buffered in the R -queue
2. A $\text{write}(x, v, TS)$ request is executed if all R -queues and W -queues are nonempty and the first write on each W -queue has a timestamp greater than TS , then the write is executed; otherwise, the write request is buffered in the W -queue
3. When read or write requests are buffered, buffered requests are tested to see if any can be executed.

Drawbacks of CTO Algorithm



termination is not guaranteed;

a scheduler might not receive any request
and thus its queue will be empty

very conservative approach;

all actions (conflicting and non conflicting
actions) are executed in timestamp order

Optimistic Concurrency Control



Disadvantage of locking schemes

lock maintenance

pay high overhead most of time, but in worst case scenario you need it

probability of accessing the same data is $1/n$

lock can result in deadlock

transactions maybe aborted and only then locks are released

=> reduce potential concurrency

Optimistic Concurrency Control 1



Assume that most of the time, probability of conflict is low.

Transactions allowed to proceed in parallel until *close transaction* request from client.

Upon *close transaction*, checks for conflict; if so, some transactions aborted.

Optimistic Concurrency 2



Read phase

Transactions have *tentative* version of data items it accesses.

Tentative versions allow transactions to abort without making their effect permanent.

Validation phase

Executed upon *close transaction*.

Checks serially equivalence.

If validation fails, conflict resolution decides which transaction(s) to abort.

Optimistic Concurrency 3



Write phase

If transaction is validated, all of its tentative versions are made permanent.

Read-only transactions commit immediately.

Write transactions commit only after their tentative versions are recorded in permanent storage.

Optimistic Concurrency Control (cont)



In optimistic scheme, it assumes transactions are seldom have data conflicts so

read requests are performed immediately

write requests are recorded in a tentative form invisible to other transactions

The above two steps are the first phase.

when Tclose is received, the transaction is validated

if successful, the tentative change are made permanent and T commits

else, T is aborted and delivers Abort

Preceding Concurrent Transactions (cont)



The validation test proceeds as follows :

1. It compares T_j start- t_n with T_i finish- t_n for each transaction T_i in the list, passing over all transactions that were completed before T_j started
2. It next considers all the transactions in the list that committed between T_j start- t_n and T_i finish- t_n . The current transaction T_j only passes the test if the writes of T_i did not affect any of the same data items read by T_j .
3. Lastly it considers transactions whose second phase was partially concurrent with the second phase of the current transaction. These transactions should not have updated any items that T_j read or updated

Validation Process



Read and write sets of a T are compared with the write sets of all concurrent transactions that have completed 1st phase before T (preceding Ts)

if common elements exist, the validation fails => T is aborted

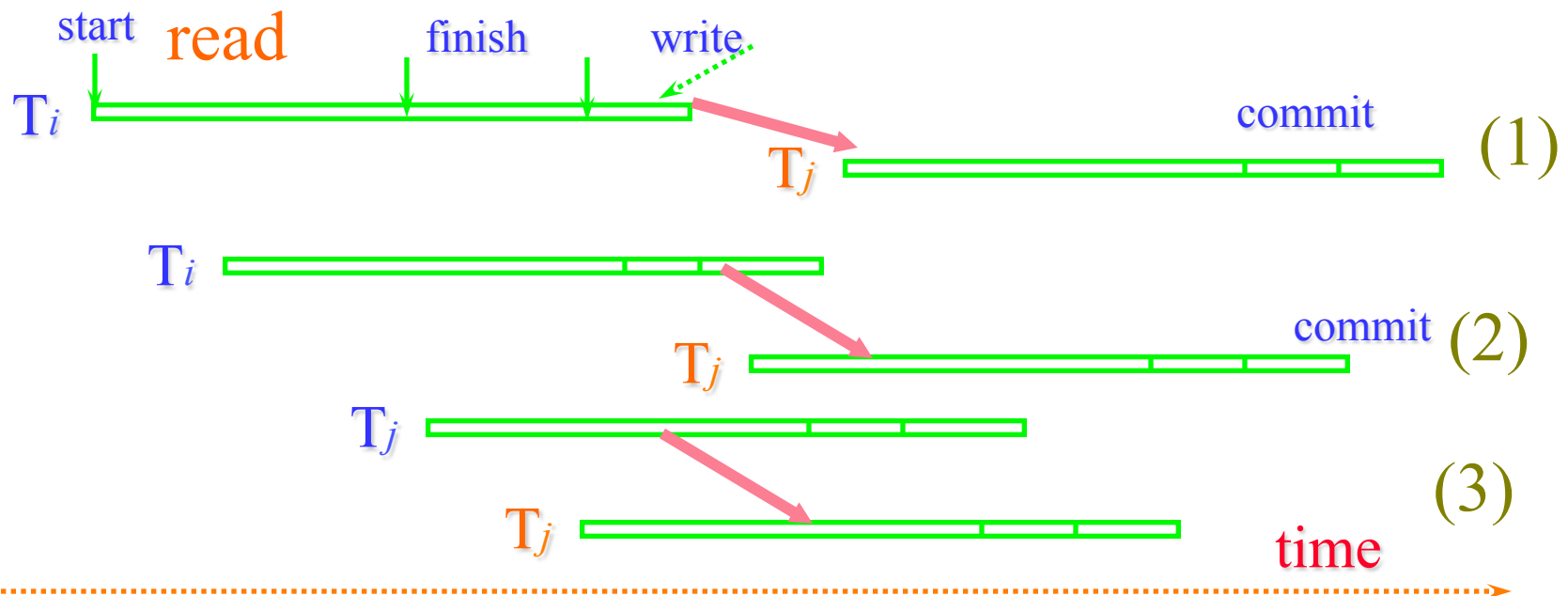
else, validation is successful

Preceding Concurrent Transactions

Three ways transactions may overlap in time

each T is given a transaction number, t_n , at the end of 1st phase
when a validate succeeds, its info is stored in preceding T List
which contains start- t_n and write set

t_{old} is deleted when $T_j \times \text{start-}t_n > T_{old} \times \text{finish-}t_n$



Comparison of Methods for Concurrency Control



time stamping and locking schemes detect conflicts as each data item is accessed

timestamps have additional information that allow some transactions to proceed while they can not in locking method

Optimistic concurrency control allows transaction to proceed and abort them when conflicts occur

=> efficient with low conflicts

locking has been widely used in database systems and file servers