Transactions and Concurrency Control

CSC 343
Fall 2019

MICHAEL LIUT (MICHAEL.LIUT@UTORONTO.CA)

DEPARTMENT OF MATHEMATICAL AND COMPUTATIONAL SCIENCES

UNIVERSITY OF TORONTO MISSISSAUGA

UNIVERSITY OF TORONTO MISSISSAUGA

UNIVERSITY OF TORONTO MISSISSAUGA

Introduction

- Concurrent execution of user programs is essential for good DBMS performance.
- Since disk accesses is frequent, and relatively slow, it is important to keep the CPU going by working on several user programs concurrently.

UNIVERSITY OF TORONTO MISSISSAUGA

Transactions

- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

Concurrency

UNIVERSITY OF TORONTO MISSISSAUGA

What is Concurrent Process (CP)?

- Multiple users access databases and use computer systems simultaneously.

Example: Airline reservation system

- An airline reservation system is used by hundreds of travel agents and reservation clerks concurrently.
- Banking system: you may be updating your account balances the same time the bank is crediting you interest.

UNIVERSITY OF TORONTO MISSISSAUGA

Concurrency

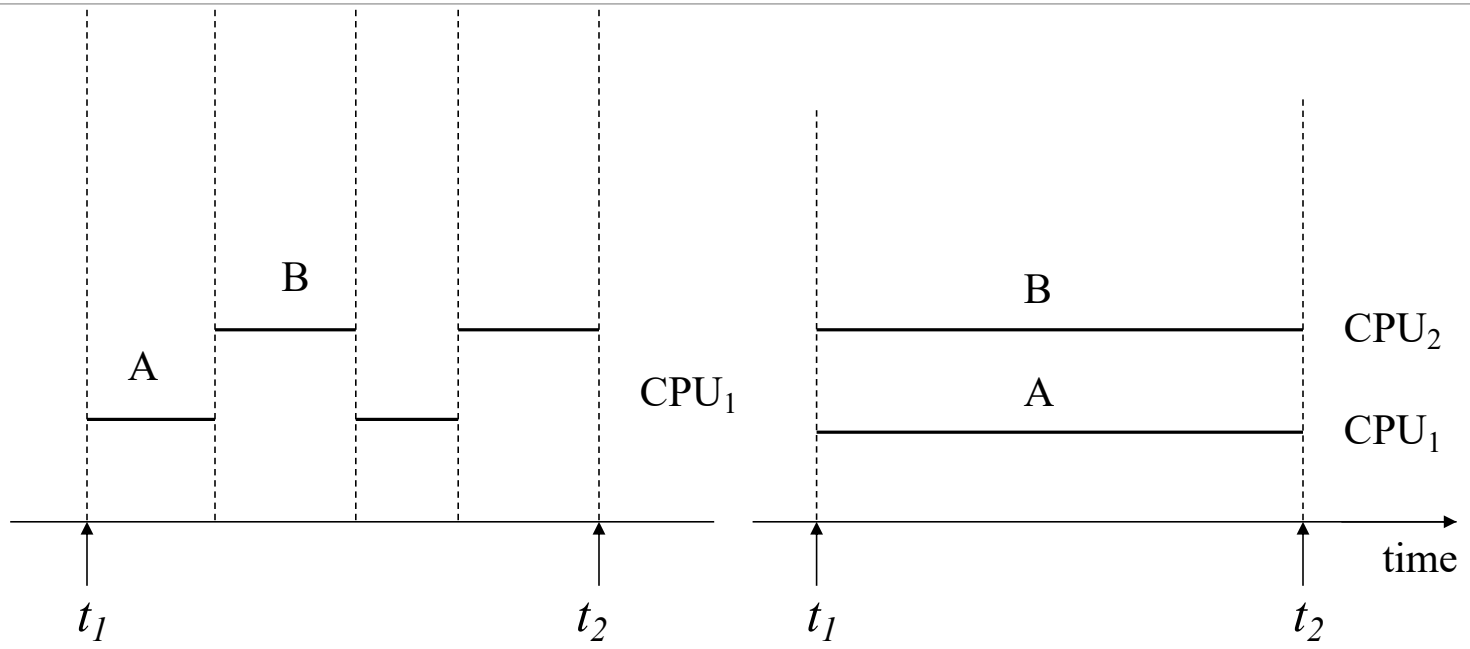
Why Concurrent Process?

- Better transaction throughput and response time
- Better utilization of resource

Concurrency

1. Interleaved Processing
   - Concurrent execution of processes is interleaved in a single CPU.
2. Parallel Processing
   - Processes are concurrently executed in multiple CPUs.

Concurrent Transactions



UNIVERSITY OF TORONTO MISSISSAUGA

Transactions

What is a transaction?

- A sequence of many actions which are considered to be one unit of work.

Basic operations a transaction can include “actions”:

- Reads , writes
- Special actions: commit, abort

UNIVERSITY OF TORONTO MISSISSAUGA

Concurrency

Users submit transactions, and can think of each transaction as executing by itself.

- Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
  - DBMS will enforce some constraints
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

Issues: Effect of *interleaving* transactions, and *crashes*.

Atomicity of Transactions

UNIVERSITY OF TORONTO MISSISSAUGA

A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.

A very important property guaranteed by the DBMS for all transactions is that they are *atomic*. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.

- DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

UNIVERSITY OF TORONTO MISSISSAUGA

Transaction Properties (ACID)

Atomicity

- Transaction is either performed in its entirety or not performed at all, this should be DBMS' responsibility

Consistency

- Transaction must take the database from one consistent state to another.

Isolation

- Transaction should appear as though it is being executed in isolation from other transactions

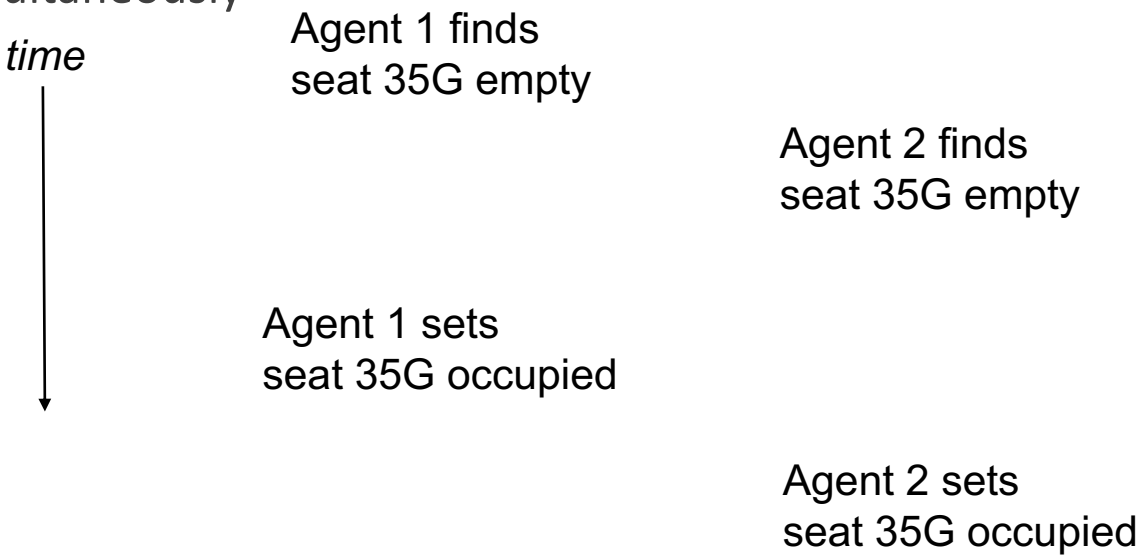
Durability

- Changes applied to the database by a committed transaction must persist, even if the system fails before all changes reflected on disk

Oops, Something's Wrong

Reserving a seat for a flight

In concurrent access to data in DBMS, two users may try to book the same seat simultaneously



UNIVERSITY OF TORONTO MISSISSAUGA

Example

Consider two transactions (*Xacts*):

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,  B=1.06*B  END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example

Consider a possible interleaving (*schedule*):

T1:	A=A+100,		B=B-100
T2:		A=1.06*A,	B=1.06*B

- This is OK. But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- The DBMS's view of the second schedule:

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

What Can Go Wrong?

UNIVERSITY OF TORONTO MISSISSAUGA

Concurrent process may end up violating Isolation property of transaction if not carefully scheduled

Transaction may be aborted before committed undo the uncommitted transactions

- undo transactions that sees the uncommitted change before the crash

Schedule

A schedule S of n transactions T1,T2,…Tn is an ordering of the operations of the transactions subject to the constraint that,

- for each transaction Ti that participates in S, the operations of Ti in S must appear in the same order in which they occur in Ti.

Informally, a schedule is a sequence of interleaved actions from all transactions

Example:

S_a: R1(A),R2(A),W1(A),W2(A), Abort1,Commit2;

T1	T2
Read(A)	
	Read(A)
Write(A)	
	Write(A)
Abort T1	
	Commit T2

Scheduling Transactions

UNIVERSITY OF TORONTO MISSISSAUGA

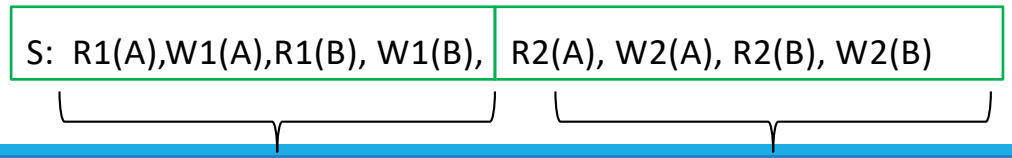
Serial schedule: Schedule that does not interleave the actions of different transactions.

Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.

Serial Schedule

T1	T2
R(A)	
A := A+100	
R(B)	
B := B+100	
	R(A)
	A: = A* 2
	R(B)
	B:= B*2



A Serializable Schedule

T1	T2
R(A)	
A := A+100	
	R(A)
	A: = A* 2
R(B)	
B := B+100	
	R(B)
	B:= B*2

Notice: this is not a serial schedule, i.e., there is interleaving of operations

Net effect is the same as the serial schedule

S: R1(A),W1(A), R2(A), W2(A), R1(B), W1(B), R2(B), W2(B)

A Non-Serializable Schedule

T1	T2
R(A)	
A := A+100	
	R(A)
	A: = A* 2
	R(B)
	B:= B*2
R(B)	
B := B+100	

What is different?

Ordering of operations is not consistent

S: R1(A),W1(A), R2(A), W2(A), R2(B), W2(B), R1(B), W1(B)

Conflict Operations

UNIVERSITY OF TORONTO MISSISSAUGA

Two operations in a schedule are said to be ***conflict*** if they satisfy all three of the following conditions:

(1) they belong to different transactions;
(2) they access the same item A; and
(3) at least one of the operations is a write(A).

Example in Sa: R1(A), R2(A), W1(A), W2(A), A1, C2

- R1(A),W2(A) conflict, so do R2(A),W1(A),
- R1(A), W1(A) do not conflict because they belong to the same transaction,
- R1(A),R2(A) do not conflict because they are both read operations.

Conflicts

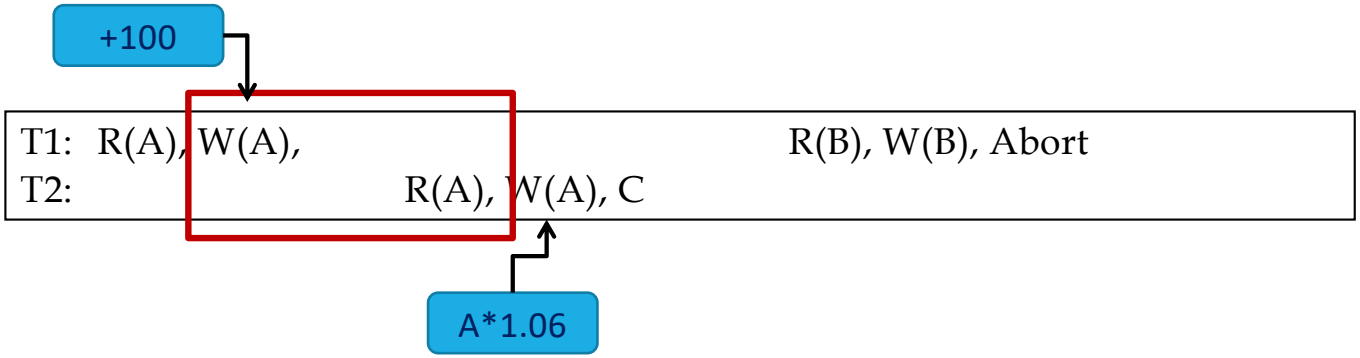
UNIVERSITY OF TORONTO MISSISSAUGA

What can go wrong:

1. Reading uncommitted data (WR), aka “dirty reads”
2. Unrepeatable reads (RW)
3. Lost updates (WW)

Reading Uncommitted Data

WR Conflicts, “dirty reads”

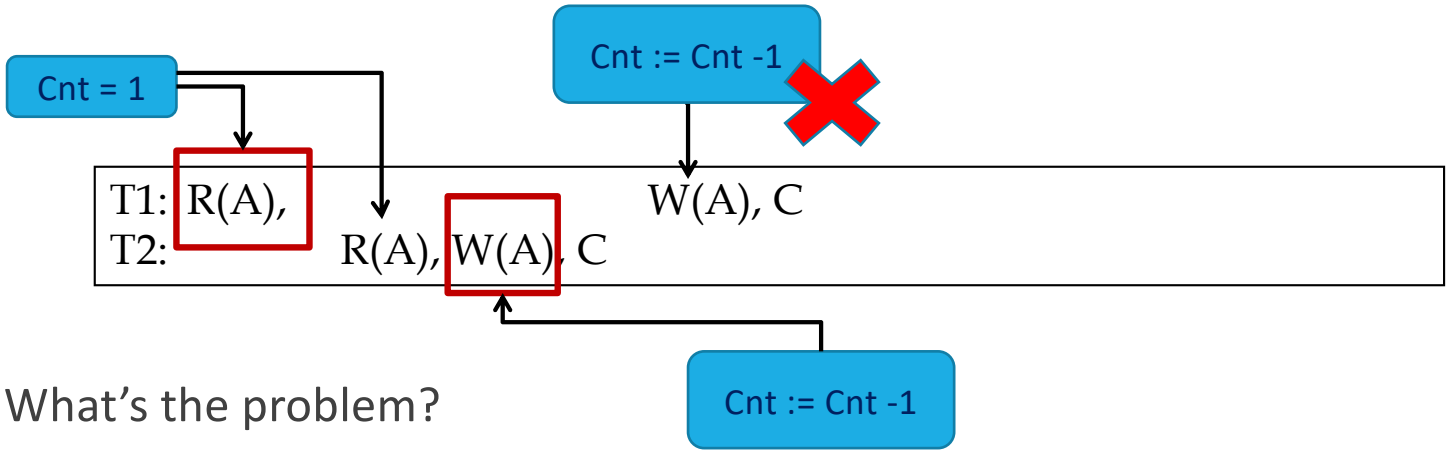


What’s the problem?

T2 has read an updated value A, which is later aborted!

Unrepeatable Reads

RW Conflicts



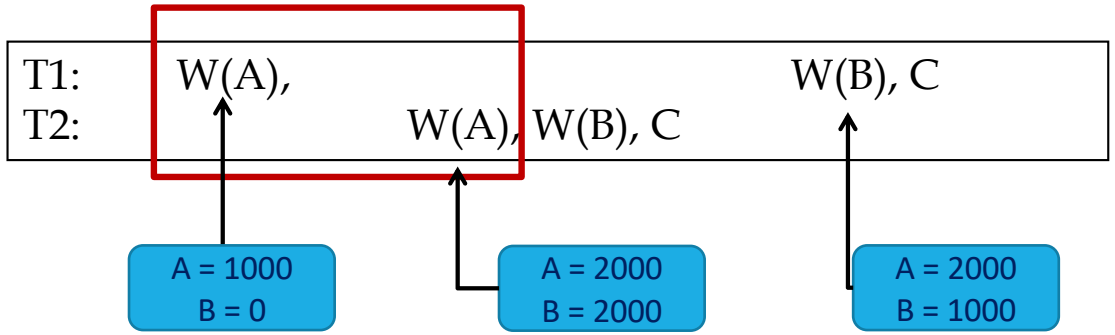
What's the problem?

T2 has changed A that has been read by T1, while T1 is still in progress

Lost Updates

Overwriting uncommitted data

WW Conflicts



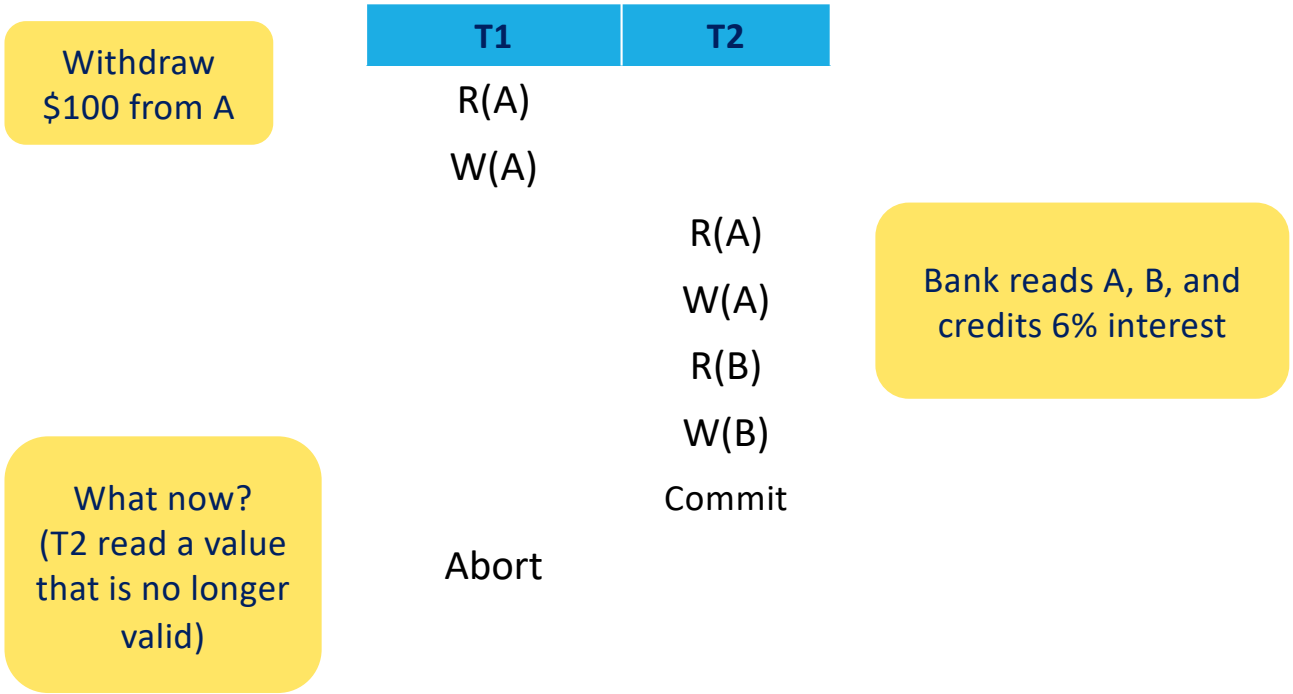
T1: sets salaries to $1000
T2: sets salaries to $2000

Require: A and B salaries to be equal

What's the problem?

T2 overwrites T1's update of A, while T1 is still in progress.

Schedules with Aborts



Options

If T2 did not commit, we abort T1 and *cascade* to T2.

But T2 committed, so we cannot undo

This schedule is *unrecoverable*

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Aborting a Transaction

UNIVERSITY OF TORONTO MISSISSAUGA

If a transaction T_i is aborted, all its actions have to be undone. Not only that, if T_j reads an object last written by T_i, T_j must be aborted as well!

Most systems try to avoid such *cascading aborts*

- If T_i writes an object, T_j can read this only after T_i commits.

Aborting a Transaction

UNIVERSITY OF TORONTO MISSISSAUGA

In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.

This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

- The ***Recovery Manager*** is responsible for ensuring Atomicity and Durability.

Recoverable Schedules and Avoid Cascading Aborts

Recoverable

Aborting T1 requires aborting T2

– But T2 has already committed!

A recoverable schedule is one in which this cannot happen.

– i.e. a Xact commits only after all the Xacts it "depends on" (i.e. it reads from) commit.

No

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	Commit
Abort	

Yes

T1	T2
R(A)	
W(A)	
	R (A)
	W (A)
Commit	
	Commit

Avoid Cascading Abort (ACA)

- Aborting T1 requires aborting T2!
- Aborting a Xact can be done without cascading the abort to other Xacts.
- A Xact only reads data from committed Xacts.
- ACA implies recoverable (but not vice-versa!)

T1	T2
R (A)	
W (A)	
	R (A)
	W (A)
Abort	

T1	T2
R (A)	
W (A)	
Commit	
	R (A)
	W (A)

UNIVERSITY OF TORONTO MISSISSAUGA

Example

W1(X), R2(Y), R1(Y), R2(X), C2, C1

T1, T2: W1(X), R1(Y), R2(Y), R2(X)

- Serializable: Yes, equivalent to T1, T2.
- Recoverable: No. Though it would be, if C1 and C2 are switched.
- ACA: No. Though it would be, if T1 commits before T2 reads X.

Including Aborts in Serialization

UNIVERSITY OF TORONTO MISSISSAUGA

Extend the definition of a serializable schedule to include aborts

Serializable schedule: a schedule that is equivalent to some serial execution of the set of *committed* transactions.

UNIVERSITY OF TORONTO MISSISSAUGA

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

UNIVERSITY OF TORONTO MISSISSAUGA

Precedence Graph Test

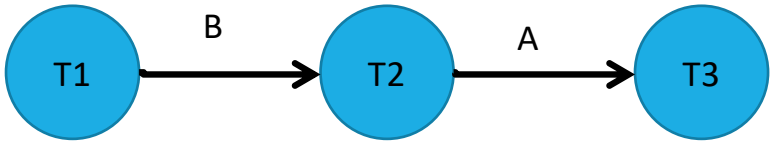
Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions Ti
- Edge from Ti to Tj if Ti comes first, and makes an action that conflicts with one of Tj
- The test: if the graph has no cycles, then it is conflict serializable!

Example 1

R2(A); R1(B); W2(A); R3(A); W1(B); W3(A); R2(B); W2(B)

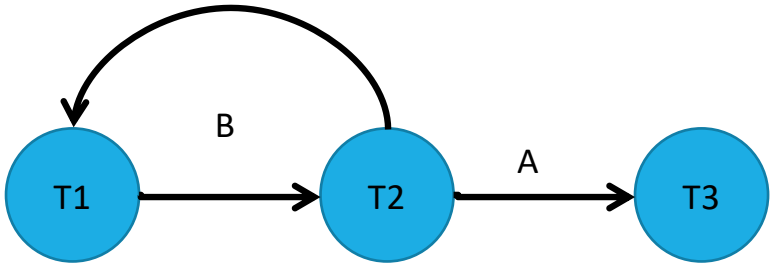


This schedule is conflict serializable!

Example 2

R2(A); R1(B); W2(A); R3(A); W1(B); W3(A); R2(B); W2(B)

R2(A); R1(B); W2(A); R2(B); R3(A); W1(B); W3(A); W2(B)



This schedule is NOT conflict serializable!

The cycle indicates that the output of T1 depends on T2, and vice-versa.

UNIVERSITY OF TORONTO MISSISSAUGA

Strict Schedule

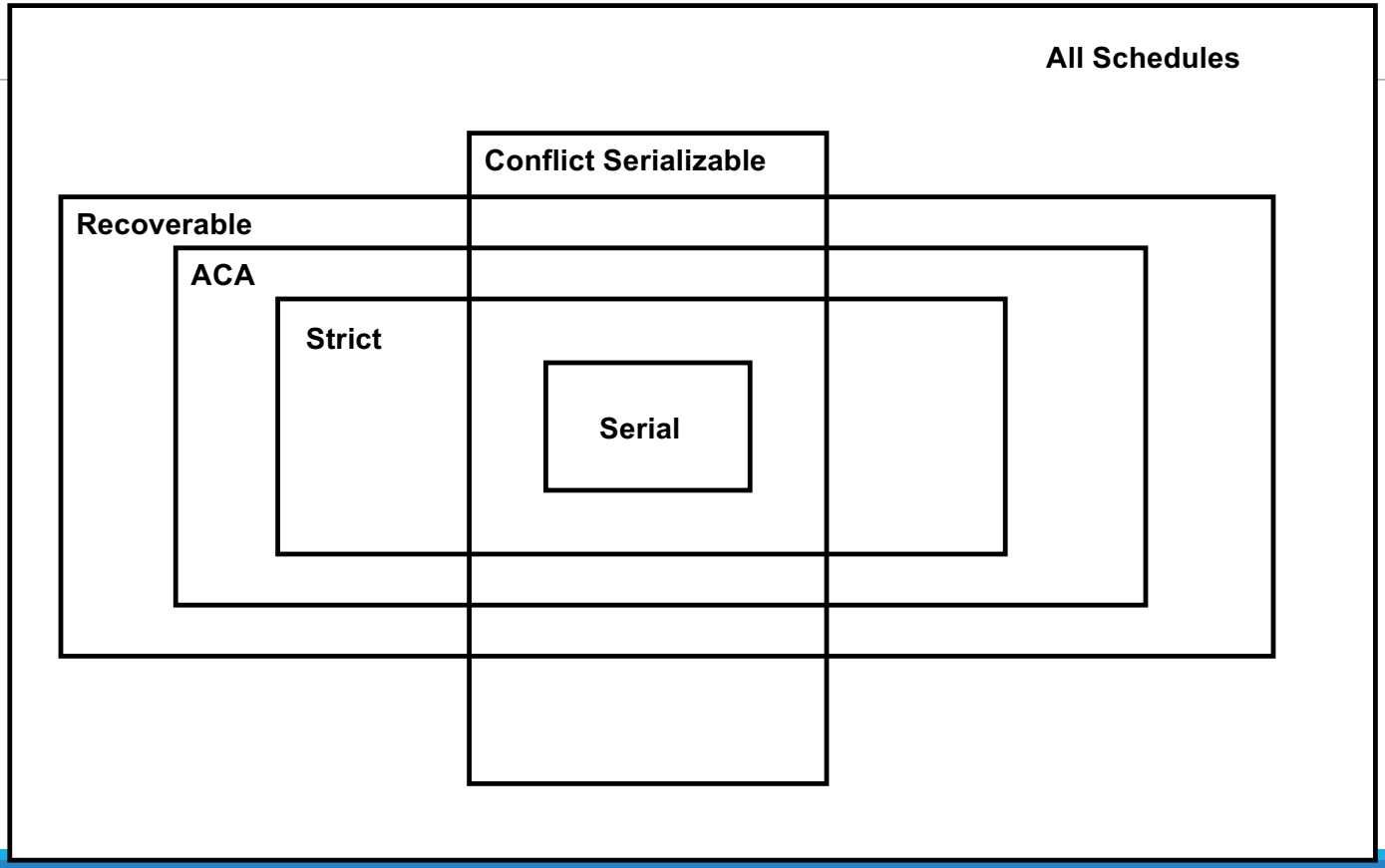
A schedule S is strict if a value written by Ti is not read or overwritten by other Tj until Ti aborts or commits

Example:

W1(A); W1(B), C1; W2(A); R2(B); C2;

Strict schedules are recoverable and avoid cascading aborts.

Venn Diagram for Schedules



Scheduler

UNIVERSITY OF TORONTO MISSISSAUGA

The scheduler is the module that schedules the transaction's actions, ensuring serializability.

How?

- Locks
- Time stamps

Lock Based Concurrency Control

UNIVERSITY OF TORONTO MISSISSAUGA

DBMS aims to allow only recoverable and serializable schedules

Ensure committed transactions are not un-done while aborting transactions

Use a *locking protocol*: a set of rules to be followed by each transaction to ensure serializable schedules

Lock: a mechanism to control concurrent access to a data object

Locking Scheduler

UNIVERSITY OF TORONTO MISSISSAUGA

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

Notation

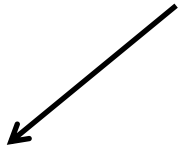
UNIVERSITY OF TORONTO MISSISSAUGA

Li(A) = transaction Ti acquires lock for element A

Ui(A) = transaction Ti releases lock for element A

Example 1

T1	T2
L1(A)	
R1(A), W1(A)	
U1(A), L1(B)	
	L2(A)
	R2(A), W2(A)
	U2(A)
	L2(B) , **DENIED…**
R1(B), W1(B)	
U1(B)	
	GRANTED;
	R2(B), W2(B)
	U2(B)



Scheduler has enforced a conflict serializable schedule

Example 2

T1	T2
L1(A)	
R1(A), W1(A)	
U1(A)	
	L2(A)
	R2(A), W2(A)
	U2(A)
	L2(B)
	R2(B), W2(B)
	U2(B)
L1(B)	
R1(B), W1(B)	
U1(B)	

Scheduler has NOT enforced conflict serializability

For A: T1-T2
For B: T2-T1

Types of Locks

UNIVERSITY OF TORONTO MISSISSAUGA

Shared lock (for reading)

Exclusive lock (for writing, and of course, also for reading)

Notation

- $S_T(A)$: transaction T requests shared lock on object A
- $X_T(A)$: transaction T requests exclusive lock on object A

Lock Modes

UNIVERSITY OF TORONTO MISSISSAUGA

S = Shared lock (for read)

X = Exclusive lock (for write)

Lock compatibility matrix

	None	S	X
None	OK	OK	OK
S	OK	OK	Conflict
X	OK	Conflict	Conflict

Strict Two Phase Locking (Strict 2PL)

UNIVERSITY OF TORONTO MISSISSAUGA

Most widely used locking protocol

Two rules:

1. Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
2. All locks held by a transaction are released when the transaction completes

If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

Strict 2PL allows only schedules whose precedence graph is acyclic (i.e., serializable)

Recoverable and ACA

Strict 2PL Example

UNIVERSITY OF TORONTO MISSISSAUGA

All locks held by T1 are released when T1 completes.

T1	T2
L(A);	
R(A), W(A)	
	L(A); **DENIED…**
L(B);	
R(B), W(B)	
U(A), U(B)	
Commit;	
	…GRANTED
	R(A), W(A)
	L(B);
	R(B), W(B)
	U(A), U(B)
	Commit;

UNIVERSITY OF TORONTO MISSISSAUGA

Implications

The locking protocol only allows safe interleavings of transactions

If T1 and T2 access different data objects, then no conflict and each may proceed

Otherwise, if same object, actions are ordered serially.

- The Xact who gets the lock first must complete before the other can proceed

Two Phase Locking Protocol (2PL)

UNIVERSITY OF TORONTO MISSISSAUGA

Variant of Strict 2PL

Relaxes the 2nd rule of Strict 2PL to allow Xacts to release locks before the end (commit/abort)

Two rules:

- Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
- A transaction cannot request additional locks once it releases any lock.
- If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

2PL Example

UNIVERSITY OF TORONTO MISSISSAUGA

All locks are first acquired, and then released.

T1	T2
X(A), X(B)	
R(A), W(A)	
U(A)	
	X(A)
	R(A), W(A)
	X(B), **DENIED…**
R(B), W(B)	
U(B)	
	..GRANTED
	R(B), W(B)
	U(A), U(B)

2PL Implications

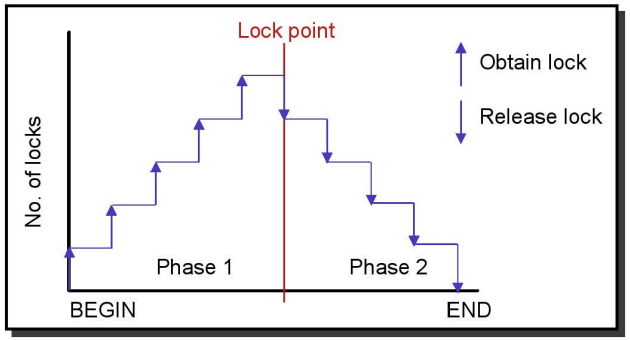
In every transaction, all lock requests must precede all unlock requests.

This ensures conflict serializability

- Why? (Think of order Xacts enter their shrinking phase)
- This induces a sort ordering of the transactions that can be serialized

Each transaction is executed in two phases:

- Growing phase: the transaction obtains locks
- Shrinking phase: the transaction releases locks



Strict 2PL makes 2PL "strict"

UNIVERSITY OF TORONTO MISSISSAUGA

Recall: a strict schedule is one where a value written by T is not read/overwritten until T commits/aborts

- Strict 2PL makes T hold locks until commit/abort
- No other transaction can see or modify the data object

until T is complete

Transaction Support in SQL

UNIVERSITY OF TORONTO MISSISSAUGA

A transaction is initiated implicitly when SQL statement is executed.

- Each DBMS provides either a commit or a rollback (abort) option

Isolation level: controls the extent to which a transaction is exposed to actions of other transactions executing concurrently

Four possible isolation levels

- Increase concurrency → increasing Xact exposure to uncommitted changes from other Xacts

Performance of Locking

UNIVERSITY OF TORONTO MISSISSAUGA

Locking aims to resolve conflicts among transactions by:

- Blocking
- Aborting

} Performance penalty

Blocked Xacts hold locks other Xacts may want

Aborting Xact wastes work done thus far

Deadlock: Xact is blocked indefinitely until one of the Xacts is aborted

Locking Performance

UNIVERSITY OF TORONTO MISSISSAUGA

Locking performance problems are *common!*

The problem is too much blocking.

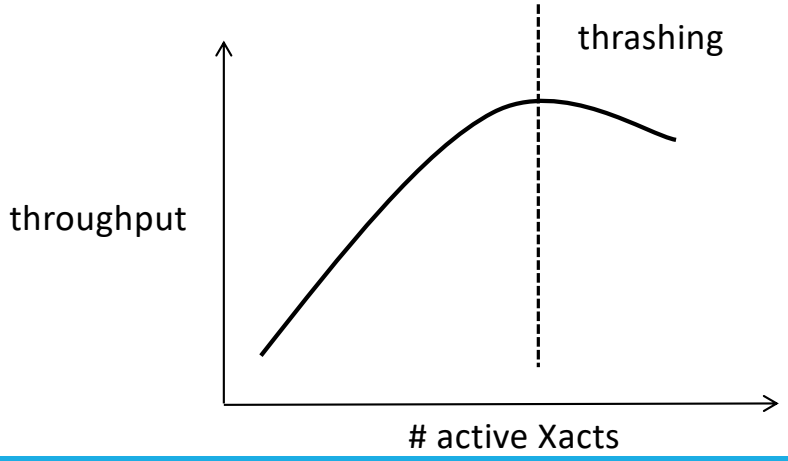
The solution is to reduce the “locking load”

Good heuristic – If more than 30% of transactions are blocked, then reduce the number of concurrent transactions

Performance

Locking overhead is primarily due to delays from blocking; minimizes throughout.

What happens to throughput as you increase the # of Xacts?



UNIVERSITY OF TORONTO MISSISSAUGA

Improving Performance

Lock the smallest sized object

- Reduce likelihood that two Xacts need the same lock

Reduce the time Xacts hold locks

Reduce *hot spots*. A hot spot is an object that is frequently accessed, and causes blocking delays

UNIVERSITY OF TORONTO MISSISSAUGA

Locking Granularity

Granularity - size of data items to lock

- e.g. files, pages, records, fields

Coarse granularity implies

- very few locks, so little locking overhead
- must lock large chunks of data, so high chance of conflict, so concurrency may be low

Fine granularity implies

- many locks, so high locking overhead
- locking conflict occurs only when two transactions try to access the exact same data concurrently

UNIVERSITY OF TORONTO MISSISSAUGA

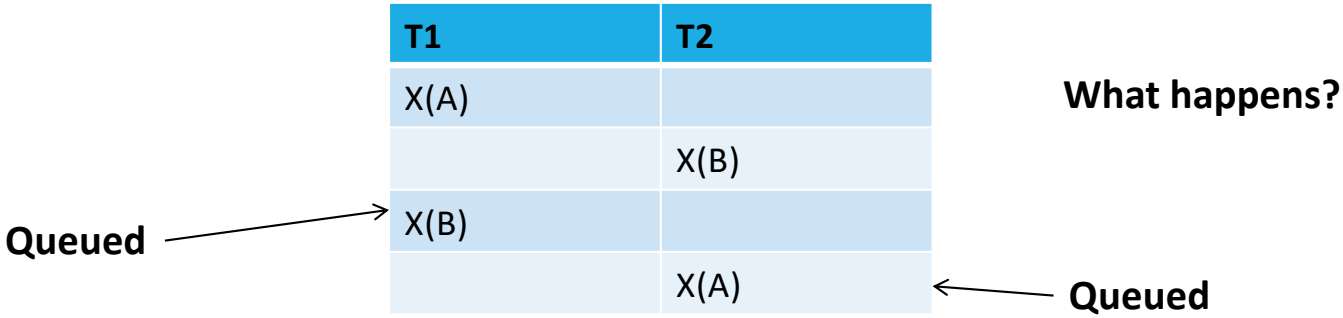
Deadlocks

Deadlock: Cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

- Deadlock detection
- Deadlock prevention

Deadlocks



T1 is waiting for T2 to release its lock
T2 is waiting for T1 to release its lock
→ Such a cycle of transactions is a ***deadlock***

Implications:
- T1 and T2 will make no further progress
- They may hold locks needed by other Xacts
- DBMS tries to prevent or detect (and resolve) deadlocks

UNIVERSITY OF TORONTO MISSISSAUGA

Summary

There are several lock-based concurrency control schemes (Strict 2PL, 2PL).

SQL-92 provides different isolation levels that control the degree of concurrency

The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.

UNIVERSITY OF TORONTO MISSISSAUGA

Lock Conversions

Lock Conversions

UNIVERSITY OF TORONTO MISSISSAUGA

Lock upgrade: Xact that holds a shared lock can be upgraded to hold an exclusive lock

- Provides more concurrency

Lock downgrade: Xact that holds an exclusive lock, downgrades to a shared lock.

- Reduces concurrency (holding locks for writing when not required)
- Reduces deadlocks

2PL with lock conversions

- During growing phase:
  - can acquire an S-lock on item
  - can acquire an X-lock on item
  - can convert an S-lock to an X-lock (upgrade)
  - Special case: allow lock downgrades only if Xact did not modify the data object (read only)

- During shrinking phase:
  - can release an S-lock
  - can release an X-lock
  - can convert an X-lock to an S-lock (downgrade)

Multiple Granularity Locking (MGL)

UNIVERSITY OF TORONTO MISSISSAUGA

We've referred to locking 'data objects'

Sometimes preferable to group several data objects together

- E.g., locking all records in a file
- Define multiple levels of locking *granularity*

Database consists of different data objects:

- a field (attribute)
- a database record
- a page
- a table
- a file
- the entire database

Multiple Granularity Locking (MGL)

UNIVERSITY OF TORONTO MISSISSAUGA

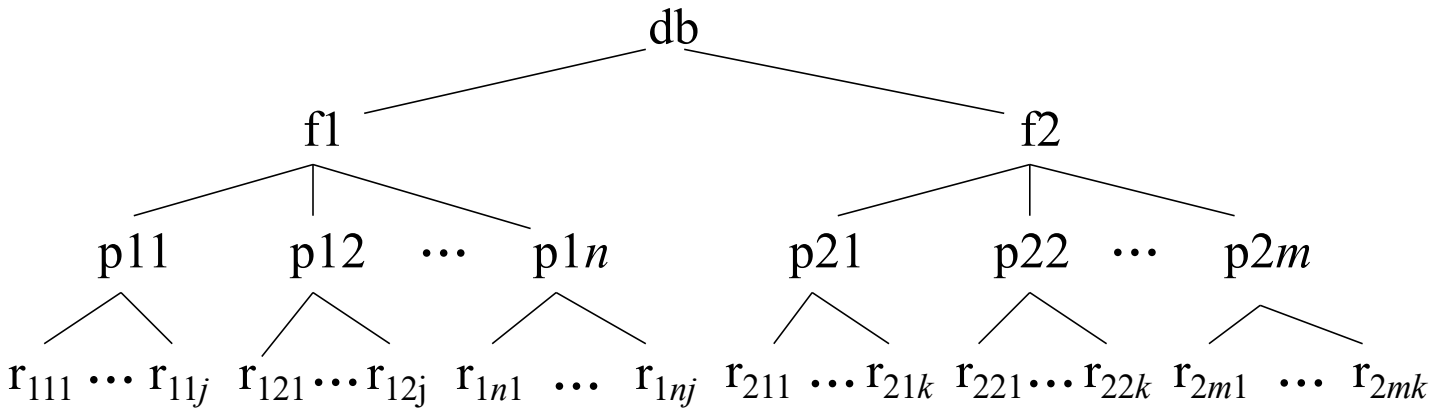
- Coarse grained locking → lower degree of concurrency
- Fine grained locking → higher degree of concurrency
  → Increased locking overhead

- What is the best locking granularity?

It depends on the transactions and the application

Granularity Hierarchy

Most DBMS support multiple levels of locking granularity for different transactions



Problem with S and X Locks

UNIVERSITY OF TORONTO MISSISSAUGA

T1: updates all the records in file f1.
T2: read record r_{1nj}.

Assume that T1 comes before T2:

- T1 locks f1.
- Before T2 is executed, the compatibility of the lock on r_{1nj} with the lock on f1 should be checked.

Assume that T2 comes before T1:

- T2 locks r_{1nj}.
- Before T1 is executed, the compatibility of the lock on f1 with the lock on r_{1nj} should be checked.
- Lock manager must efficiently manage all lock requests across hierarchy
- Each data object (e.g., file or record) has a different id

Solution

UNIVERSITY OF TORONTO MISSISSAUGA

- Exploit the natural hierarchy of data containment
- Before locking fine-grained data, set *intention locks* on coarse grained data that contains it
- e.g. before setting an S-lock on a record, get an intention-shared-lock on the table that contains the record

UNIVERSITY OF TORONTO MISSISSAUGA

Intention Locks

Three types of intention locks:

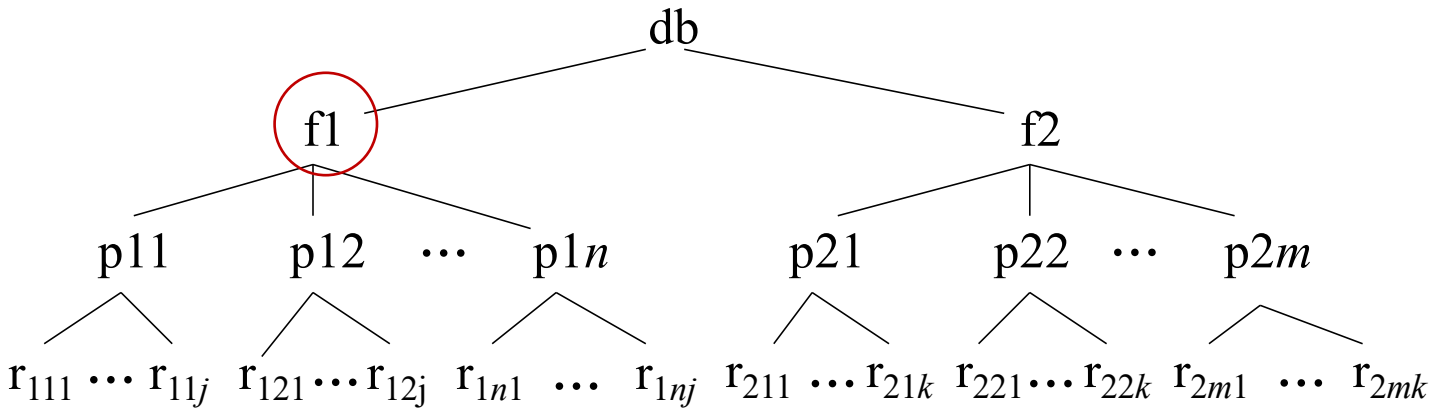
1. Intention-shared (IS) indicates that a shared lock(s) will be requested on some descendant node(s).

2. Intention-exclusive (IX) indicates that an exclusive lock(s) will be requested on some descendant node(s).

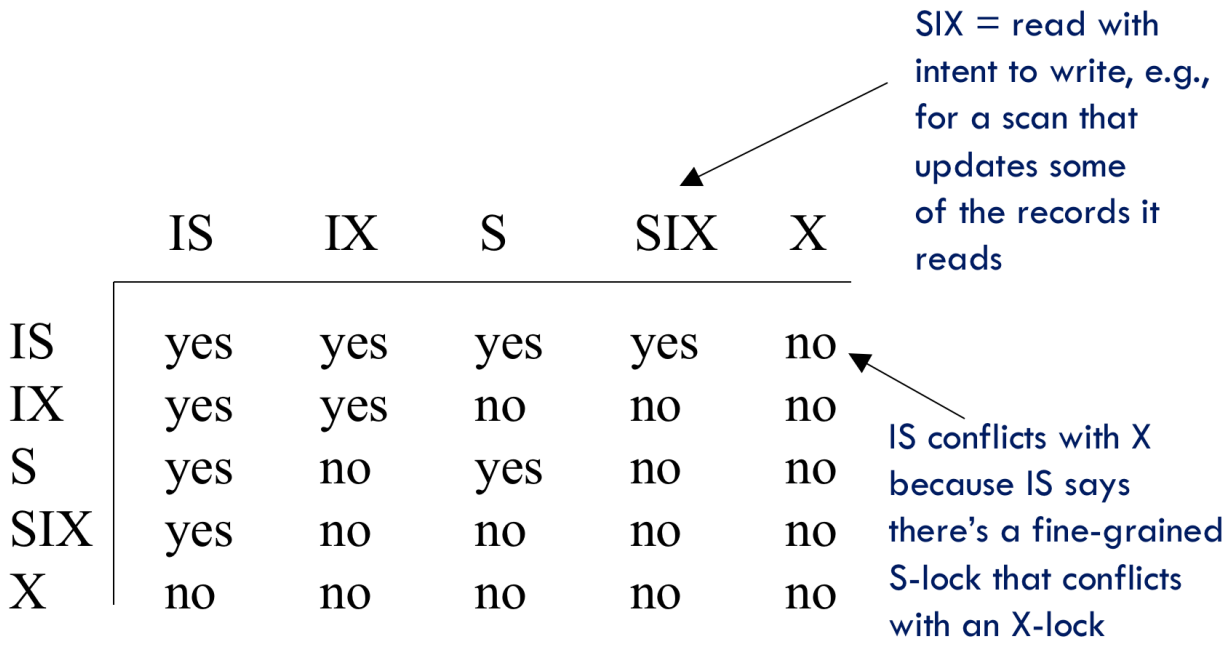
3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).

Back to our example

T1: updates all the records in file f1.
T2: read record r_{1nj}.



Compatibility Matrix



	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Multiple Granularity Lock Protocol

UNIVERSITY OF TORONTO MISSISSAUGA

Each Xact starts from the root of the hierarchy.

To get S or IS lock on a node, must hold IS or IX on parent node.

To get X or IX or SIX on a node, must hold IX or SIX on parent node.

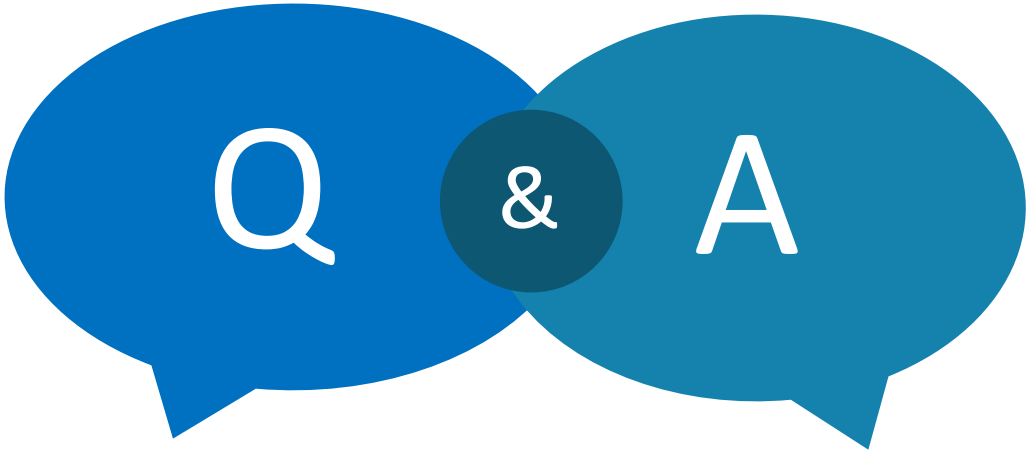
Must release locks in bottom-up (leaf to root) order.

Examples

UNIVERSITY OF TORONTO MISSISSAUGA

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on tuples of R, and occasionally upgrades to X on the tuples.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which specific tuples to get locks on (rather than locking the entire R).

Questions?



THANKS FOR LISTENING

I'LL BE ANSWERING QUESTIONS NOW

Citations, Images and Resources

UNIVERSITY OF TORONTO MISSISSAUGA

Database Management Systems (3rd Ed.), Ramakrishnan & Gehrke

Some content is based off the slides of Dr. Fei Chiang - http://www.cas.mcmaster.ca/~fchiang/

UNIVERSITY OF TORONTO MISSISSAUGA

Supporting Content

Recall Conflicts

Two writes by Ti, Tj to same element

- Wi(X); Wj(X)

Read/write by Ti, Tj to same element

- Wi(X); Rj(X)
- Ri(X); Wj(X)

Conflict Equivalent

UNIVERSITY OF TORONTO MISSISSAUGA

- Outcome of a schedule depends on the order of conflicting operations.
- Can interchange non-conflicting ops without changing effect of the schedule.

- If two schedules S1 and S2 are conflict equivalent then they have the same effect.
  - S1 ←→ S2 by swapping non-conflicting operations.

Conflict Serializability

- Every conflict serializable schedule is serializable.

R1(A); W1(A); R2(A); W2(A); R1(B); W1(B); R2(B); W2(B)

Can we transform into a serial schedule by swapping of adjacent non-conflicting actions?

R1(A); W1(A); R1(B); W1(B); R2(A); W2(A); R2(B); W2(B)

Phantom Problem

UNIVERSITY OF TORONTO MISSISSAUGA

So far we have assumed the database to be a *static* collection of elements (=tuples)

If tuples are inserted/deleted then the *phantom problem* appears

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('gizmo','blue')
SELECT * FROM Product WHERE color='blue'	

Is this schedule serializable ?

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('gizmo','blue')
SELECT * FROM Product WHERE color='blue'	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

This is conflict serializable ! What's wrong ??

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('gizmo','blue')
SELECT * FROM Product WHERE color='blue'	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

Not serializable due to ***phantoms***

UNIVERSITY OF TORONTO MISSISSAUGA

Phantom Problem

A “phantom” is a tuple that is invisible during part of a transaction execution but not all of it.

In our example:

- T1: reads list of products
- T2: inserts a new product
- T1: re-reads: a new product appears!

Phantom Problem

UNIVERSITY OF TORONTO MISSISSAUGA

In a ***static*** database:

- Conflict serializability implies serializability

In a ***dynamic*** database, this may fail due to phantoms

Dealing With Phantoms

UNIVERSITY OF TORONTO MISSISSAUGA

1. Lock the entire table, or
2. Lock the index entry for ‘blue’
   - If index is available

Dealing with phantoms is expensive !

Isolation Levels

SQL-92 provides different isolation levels that control the degree of concurrency

Isolation Level (in MySQL)	Isolation Level (in DB2)	Dirty Read	Unrepeatable Read
Read Uncommitted	Uncommitted Read	Maybe	Maybe
Read Committed (default)	Cursor Stability (default)	No	Maybe
Repeatable Reads	Repeatable Reads	No	No
Serializable	Read Stability	No	No

Phantoms possible

UNIVERSITY OF TORONTO MISSISSAUGA

Degrees of Isolation in SQL

Four levels of isolation

- READ UNCOMMITTED: no read locks
- READ COMMITTED: short duration read locks
- REPEATABLE READ:
  - Long duration read locks on individual items
- SERIALIZABLE:
  - All locks long duration

Trade-off: consistency vs concurrency

Commercial systems give choice of level

UNIVERSITY OF TORONTO MISSISSAUGA

Isolation Levels in SQL

1. "Dirty reads"
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. "Committed reads"
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. "Repeatable reads"
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

UNIVERSITY OF TORONTO MISSISSAUGA

Choosing Isolation Level

Trade-off: efficiency vs correctness

DBMSs give user choice of level

Beware!!
- Default level is often NOT serializable
- Default level differs between DBMSs
- Serializable may not be exactly ACID

Always read the docs!

Deadlock Detection

UNIVERSITY OF TORONTO MISSISSAUGA

Detect deadlocks automatically, and abort a deadlocked transaction (the *victim*).

Preferred approach, because it allows higher resource utilization

Timeout-based deadlock detection - If a transaction is blocked for too long, then abort it.

- Simple and easy to implement
- But aborts unnecessarily (pessimistic) and
- some deadlocks persist for too long

Deadlock Detection

UNIVERSITY OF TORONTO MISSISSAUGA

Create a waits-for graph:

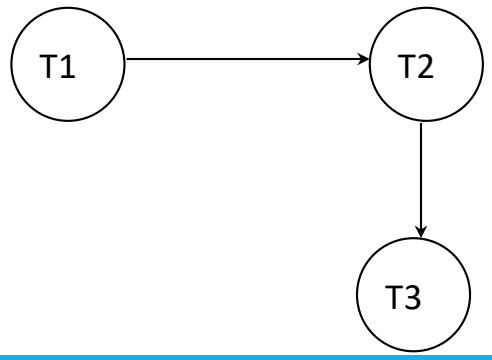
- Nodes are transactions
- There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Lock mgr adds edge when lock request is queued
- Remove edge when lock request granted

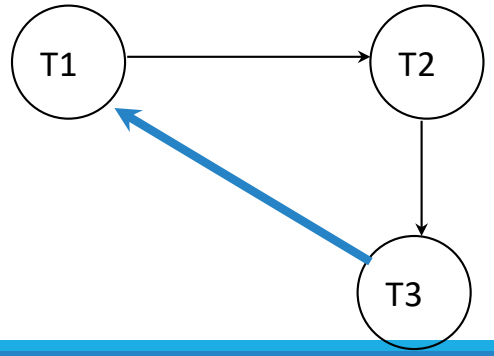
A deadlock exists if there is a cycle in the waits-for graph

Periodically check for cycles

Waits-For Graph: Example

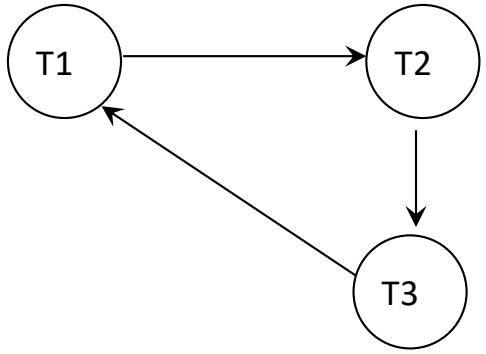
T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)





Example

T1:	X(A), W(A),		S(B)	
T2:		X(B),W(B)		X(C)
T3:			S(C), R(C), X(A)	



UNIVERSITY OF TORONTO MISSISSAUGA

Selecting a Victim

Deadlock is resolved by aborting a Xact in the cycle, and releasing its locks

Different criteria may be used:

a) Xact with the fewest/most locks
b) Xact that has done the least work
c) Xact that is farthest from completion
d) If a Xact is repeatedly restarted, increase its priority to allow it to complete

UNIVERSITY OF TORONTO MISSISSAUGA

Commercial DBMS Approaches

MS SQL Server: Aborts the transaction that is "cheapest" to roll back.

- "Cheapest" is determined by the amount of log generated.
- Allows transactions that you've invested a lot in, to complete.

Oracle: The transaction that detects the deadlock is the victim.

DB2: the deadlock detector arbitrarily selects one deadlocked process as the victim to roll back.

Deadlock Prevention

UNIVERSITY OF TORONTO MISSISSAUGA

When there is a high level of lock contention and an increased likelihood of deadlocks

Prevent deadlocks by giving each Xact a priority

Assign priorities based on timestamps.

- Lower timestamp indicates higher priority
- i.e. oldest transaction has the highest priority
- Higher priority Xacts cannot wait for lower priority Xacts (or vice versa)

Deadlock Prevention

Assume Ti wants a lock that Tj holds. Two policies are possible:

- **Wait-Die:** If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts
- **Wound-wait:** If Ti has higher priority, Tj aborts; otherwise Ti waits

Both schemes will cause aborts even though deadlock may not have occurred.

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

Wait-Die

UNIVERSITY OF TORONTO MISSISSAUGA

Suppose Ti tries to lock an item already locked by Tj.

If Ti is the older transaction then Ti will wait

Otherwise Ti is aborted and re-starts later with the same timestamp.

Lower priority transactions never wait for higher priority transactions.

Example: Wait-Die

UNIVERSITY OF TORONTO MISSISSAUGA

T1	**T2**	
S(A)		T1 starts before T2, so T1 is higher priority
R(A)		
	S(B)	
	R(B)	
X(B)		
T1 is older and so it is allowed to wait.	X(A)	T2 is younger, and is aborted, which results in its locks being released. This allows T1 to proceed.
	Abort	
Resume		

Wound-Wait

UNIVERSITY OF TORONTO MISSISSAUGA

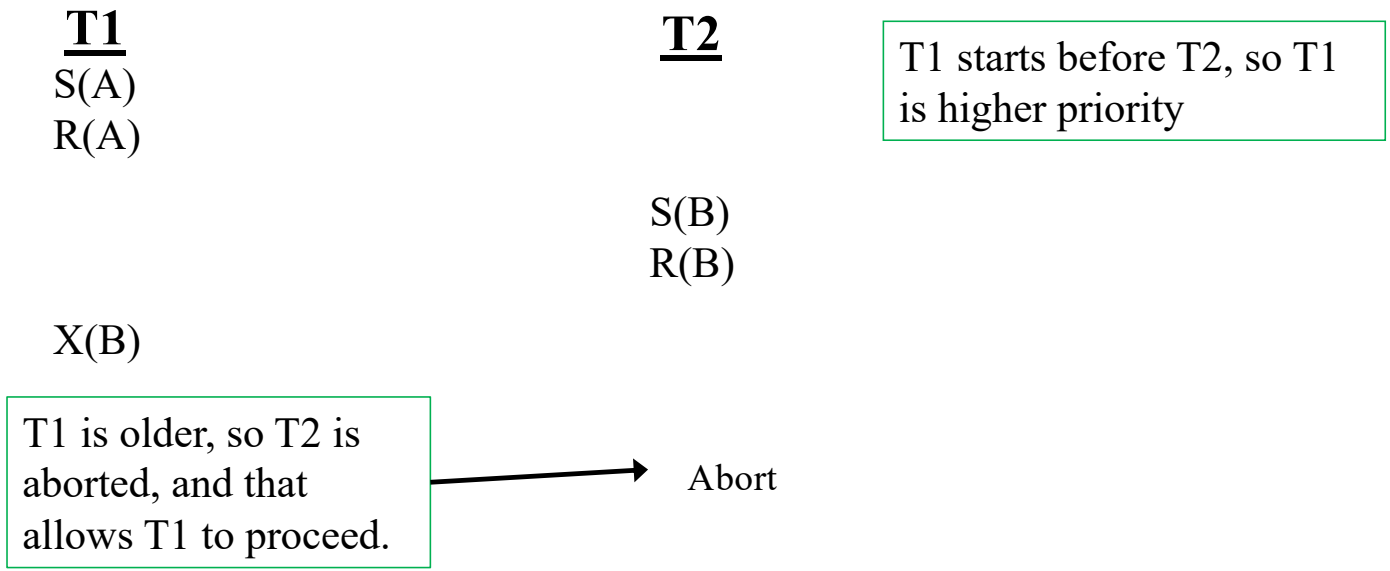
Suppose Ti tries to lock an item locked by Tj.

If Ti is higher priority (older transaction)

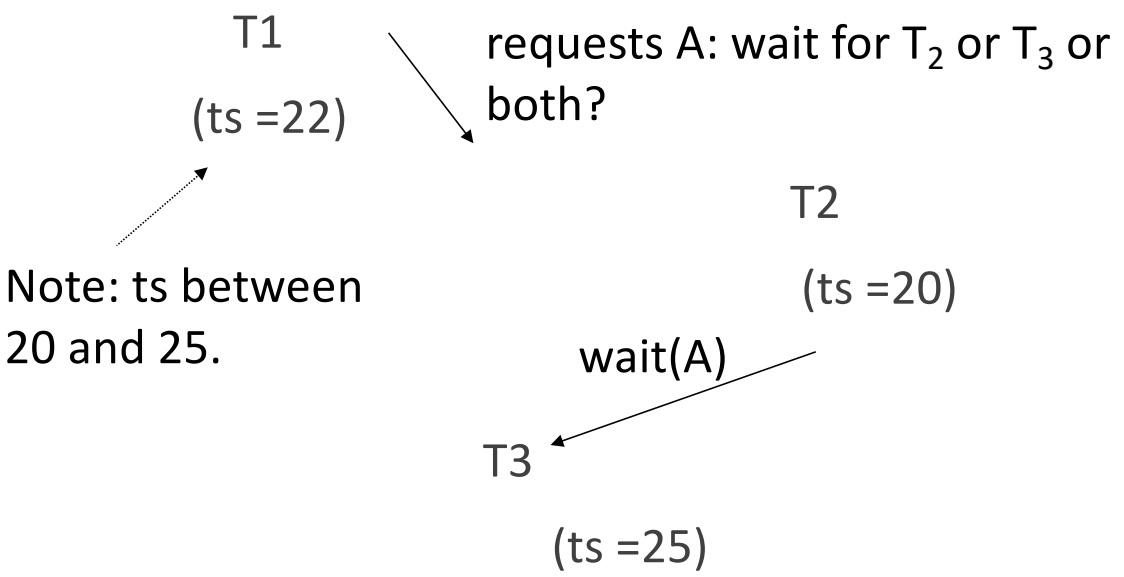
- then Tj is aborted and restarts later with the same timestamp;

Otherwise, Ti waits.

Example: Wound-Wait



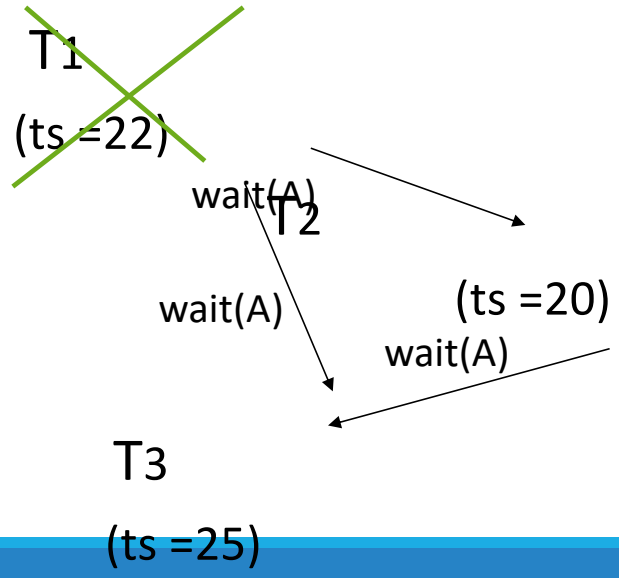
Wait-Die: Let's consider…



Wait-Die: Option 1

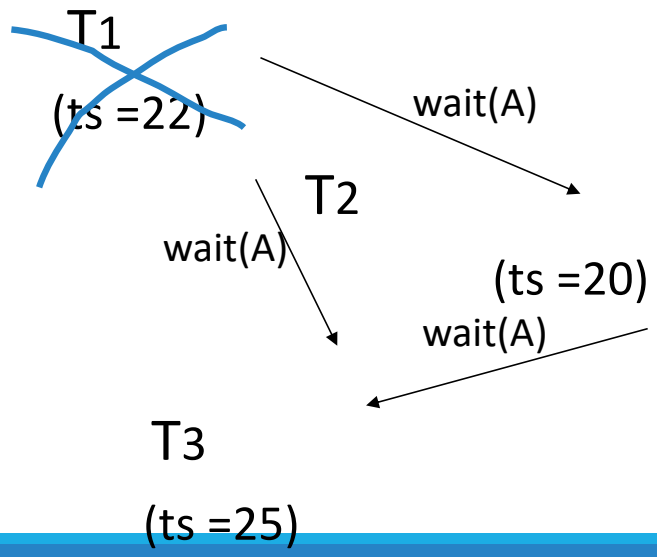
One option: T_1 waits just for T_3, transaction holding lock.

But when T_2 gets lock, T_1 will have to die!



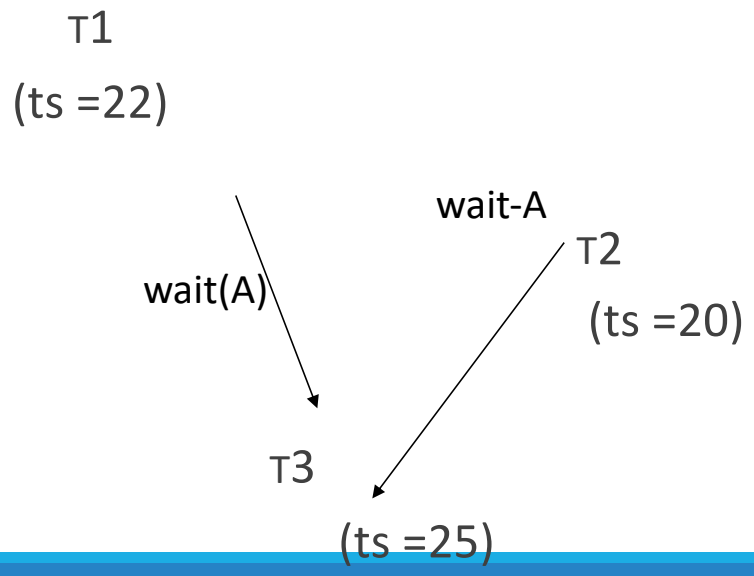
Wait-Die: Option 2

Another option: T_1 waits for both T_2, T_3.
T_1 allowed to wait iff there is at least one younger Xact waiting for A.
But again, when T2 gets lock, T_1 must die!

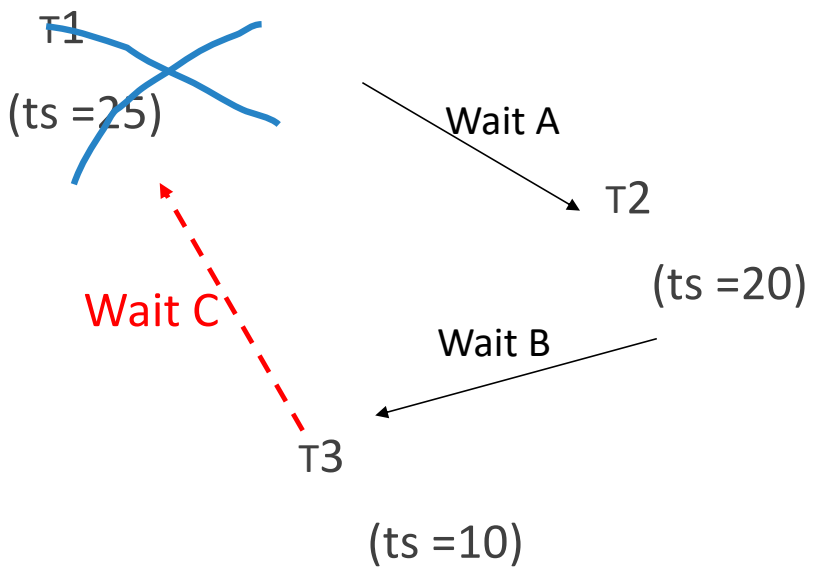


Wait-Die: Option 3

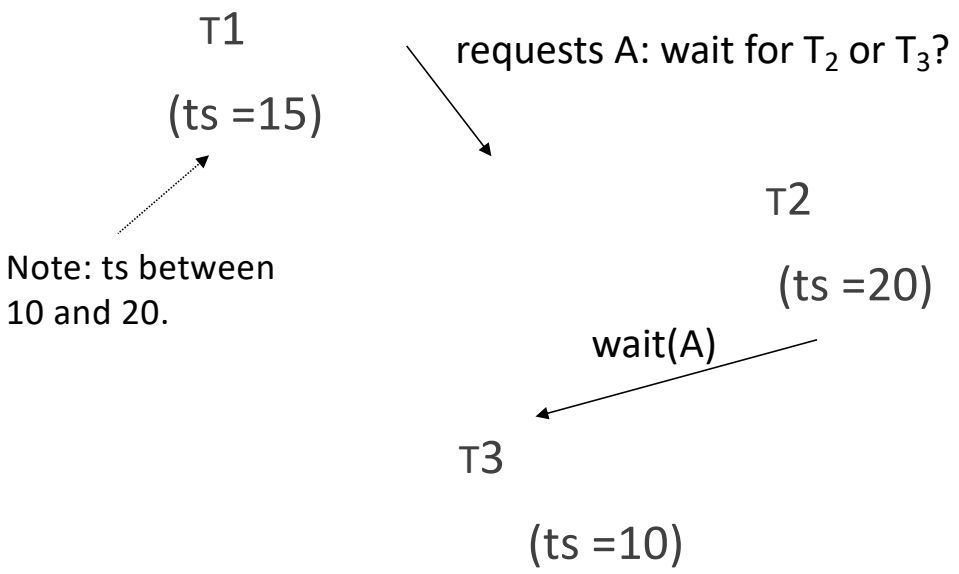
- Yet another option: T_1 preempts T_2 (T_2 is just waiting idle anyway),
- So T_1 only waits for T_3; T_2 then waits for T_1
- And lots of WFG work for Deadlock Manager



Wound-Wait



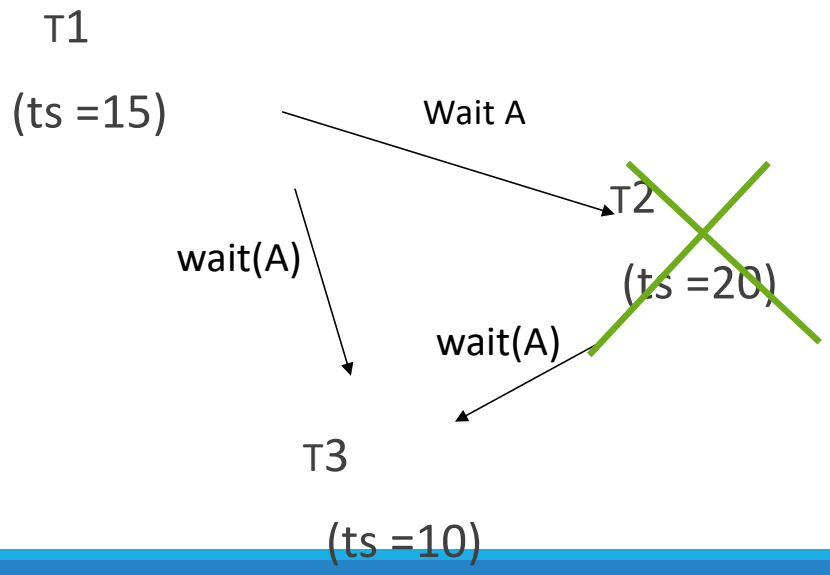
Wound-Wait: Let's Consider …



Wound-Wait: Option 1

One option: T_1 waits just for T_3, transaction holding lock.

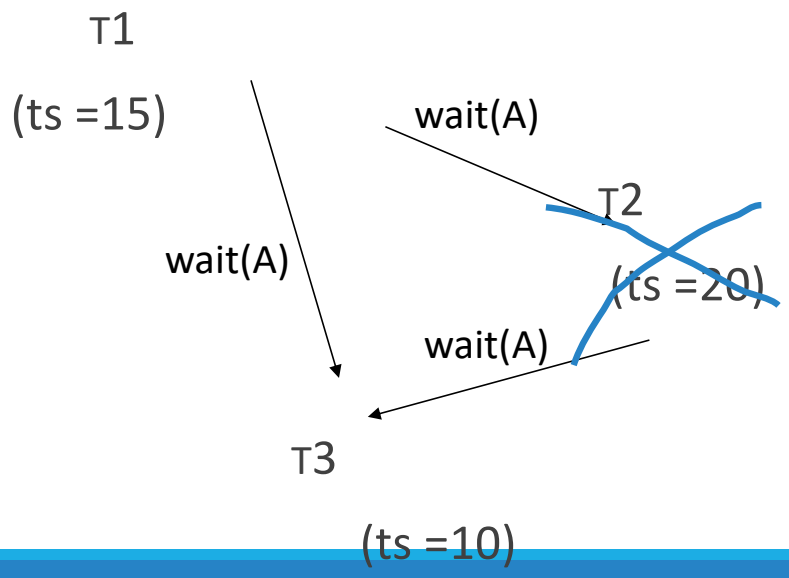
But when T_2 gets lock, T_1 waits for T_2 and wounds T_2.



Wound-Wait: Option 2

Another option:

T_1 waits for both T_2, T_3 $\Longrightarrow$ T_2 wounded right away!

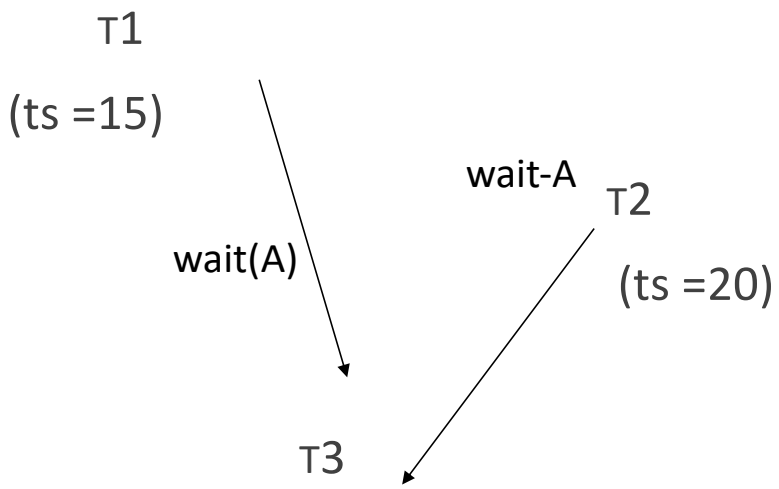


Wound-Wait: Option 3

Yet another option: T_1 preempts T_2, so T_1 only waits for T_3;

T_2 then waits for T_3 and T_1... $\Rightarrow$ T_2 is spared!

Lots of WFG work for Deadlock Mgr (shifting edges)



UNIVERSITY OF TORONTO MISSISSAUGA

Comparing Deadlock Management Schemes

Wait-die and Wound-wait ensure no starvation (unlike detection)

Waits-for graph technique only aborts transactions if there really is a deadlock