

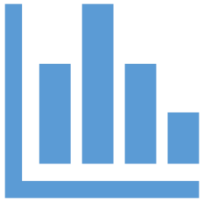


Database transactions

Tomáš Skopal
Michal Kopecký

NDBI025, 24.11.2022

Lecture overview



motivation,
ACID properties

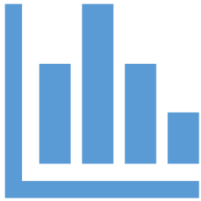


transactions,
schedules



protocols for concurrent
transactions scheduling

Lecture overview



**motivation,
ACID properties**



transactions,
schedules



protocols for concurrent
transactions scheduling

Motivation

- need to execute complex database operations
 - like stored procedures, triggers, etc.
 - multi-user and parallel environment
- database transaction
 - sequence of actions on database objects (+ others like arithmetic, etc.)
- example
 - Let's have a bank database with table **Accounts** and the following transaction to transfer money (pseudocode):

transaction **PaymentOrder**(*amount*, *fromAcc*, *toAcc*)

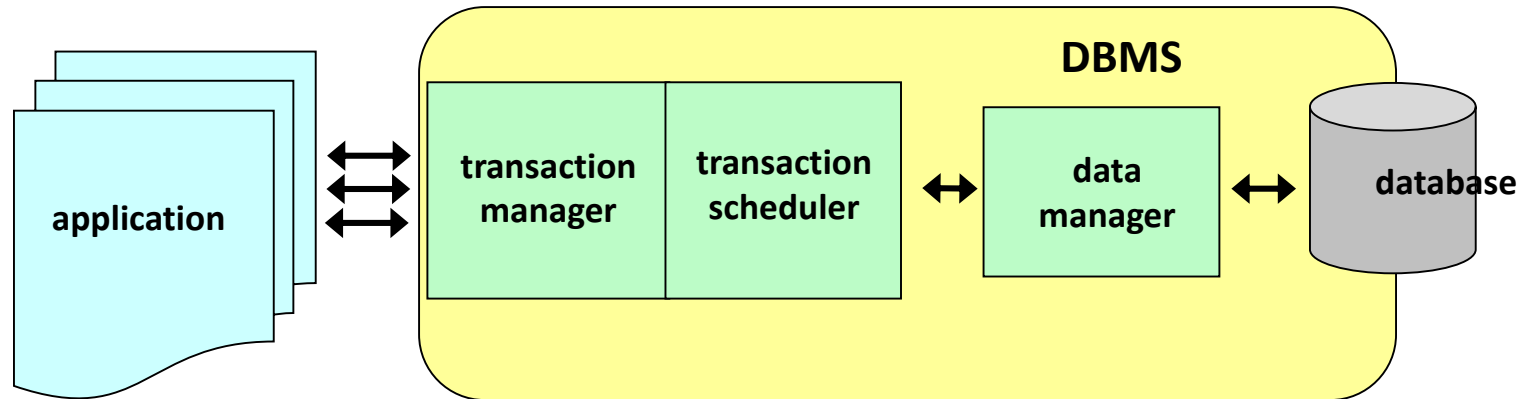
{

1. **SELECT** Balance **INTO** X **FROM** Accounts **WHERE** accNr = *fromAcc*
2. if (X < *amount*) **AbortTransaction**("Not enough money!");
3. **UPDATE** Accounts **SET** Balance = Balance - *amount* **WHERE** AccountNr = *fromAcc*;
4. **UPDATE** Accounts **SET** Balance = Balance + *amount* **WHERE** AccountNr = *toAcc*;
5. **CommitTransaction**;

}

Transaction management in DBMS

- application launches transactions
- transaction manager executes transactions
- scheduler dynamically schedules the parallel transaction execution, producing a **schedule** (history)
- data manager executes partial operation of transactions



Transaction management in DBMS

- transaction termination
 - successful – terminated by **COMMIT** command in the transaction code
 - the performed actions are confirmed
 - unsuccessful – transaction is cancelled
 - **ABORT** (or **ROLLBACK**) command – termination by the transaction code – user could be notified
 - system abort – DBMS aborts the transaction
 - for some integrity constraint violated – user is notified
 - by transaction scheduler (e.g., a deadlock occurs) – user is not notified
 - system failure – HW failure, power loss – transaction must be restarted
- main objectives of transaction management
 - enforcement of **ACID properties**
 - maximal performance (throughput trans per sec)
 - parallel/concurrent execution of transactions

ACID – desired properties of transaction management

- **Atomicity** – partial execution is not allowed (all or nothing)
 - prevents from incorrect transaction termination (or failure)
 - = consistency at the DBMS level
- **Consistency**
 - any transaction will bring the database from one valid state to another
 - = consistency at application level
- **Isolation**
 - transactions executed in parallel do not “see” effects of each other unless committed
 - parallel/concurrent execution is necessary to achieve high throughput and/or fair multitasking
- **Durability**
 - once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors
 - logging necessary (log/journal maintained)

Transaction

- an executed transaction is a sequence of actions

$T = \langle A_T^1, A_T^2, \dots, \text{COMMIT or ABORT} \rangle$

- basic database actions (operations)
 - for now consider a **static database** (no inserts/deletes, just updates), let **A** is some database object (table, row, attribute in row)
 - we omit other actions such as control construct (if, for), etc.
 - **READ(A)** – reads A from database
 - **WRITE(A)** – writes A to database
 - **COMMIT** – confirms actions executed so far as valid, terminates transaction
 - **ABORT** – cancels action executed so far, terminates transaction (with error)
- SQL commands **SELECT, INSERT, UPDATE**, could be viewed as transactions implemented using the basic actions (in SQL **ROLLBACK** is used instead of abort)

Example:

Subtract 5 from A (some attribute), such that $A > 0$.

T = <READ(A), // action 1

if (A ≤ 5) then ABORT

else WRITE(A – 5), // action 2

COMMIT> // action 3

or

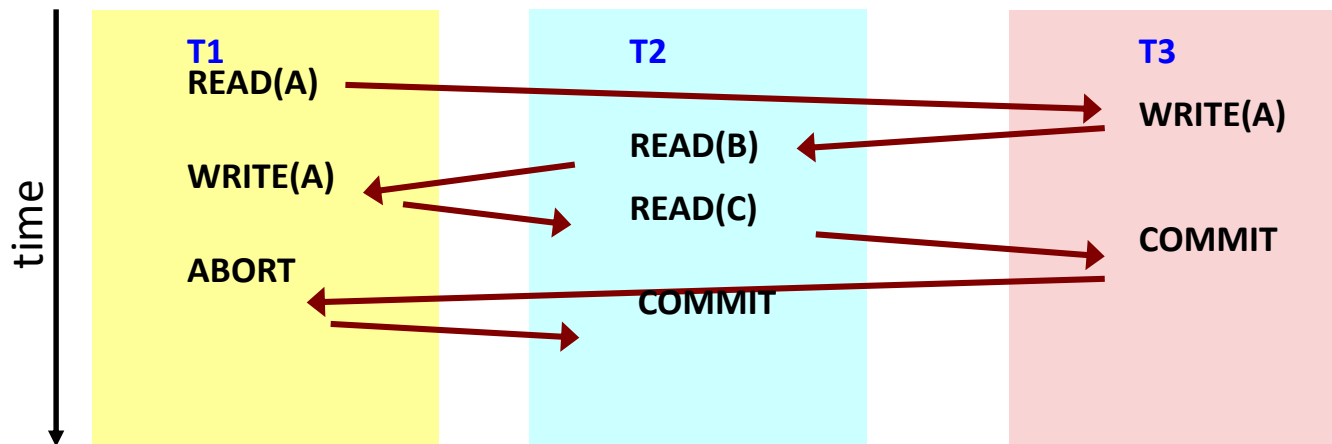
T = <READ(A), // action 1

if (A ≤ 5) then ABORT // action 2

else ... >

Transaction programs vs. schedules

- database program
 - “design-time” (not running) piece of code (that will be executed as a transaction)
 - i.e., nonlinear – branching, loops, jumps
- schedule (history) is a sorted list of actions coming from several (interleaved) transactions
 - „runtime“ history **of already concurrently executed** actions of **several** transactions
 - i.e., linear – sequence of primitive operations, w/o control constructs



Transaction scheduler

- part of the transaction manager that is responsible for scheduling concurrent execution of actions of particular transactions
 - i.e., schedule is just history, once a step of a schedule is created it was executed (so there nothing like offline schedule optimization or correction, it could be just analyzed) scheduler does not see a future when he decides next step – only the order of already scheduled operations, and current operations that can be at the moment requested from clients
- the database state is changing during concurrent execution
 - threat – temporarily inconsistent database state is exposed to other transactions
 - scheduler must provide the „isolation illusion“ and so avoid conflicts
 - example:

T1

READ(A)

A := A + 1

WRITE(A)

COMMIT

T2

READ(B)

READ(A)

COMMIT

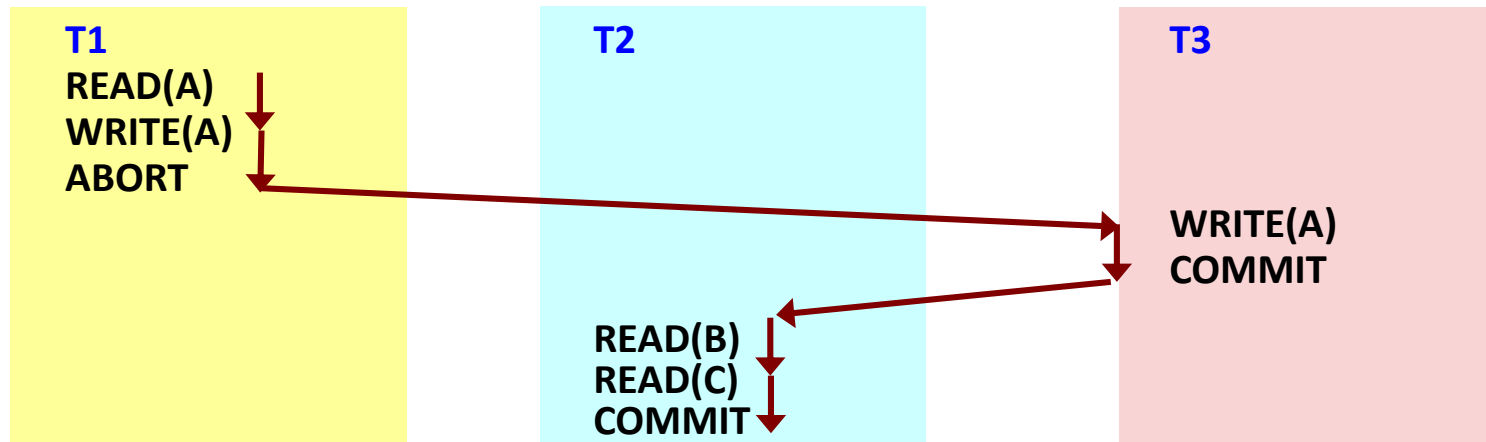
// A = 5

// B = 3

// A = 6 !!!

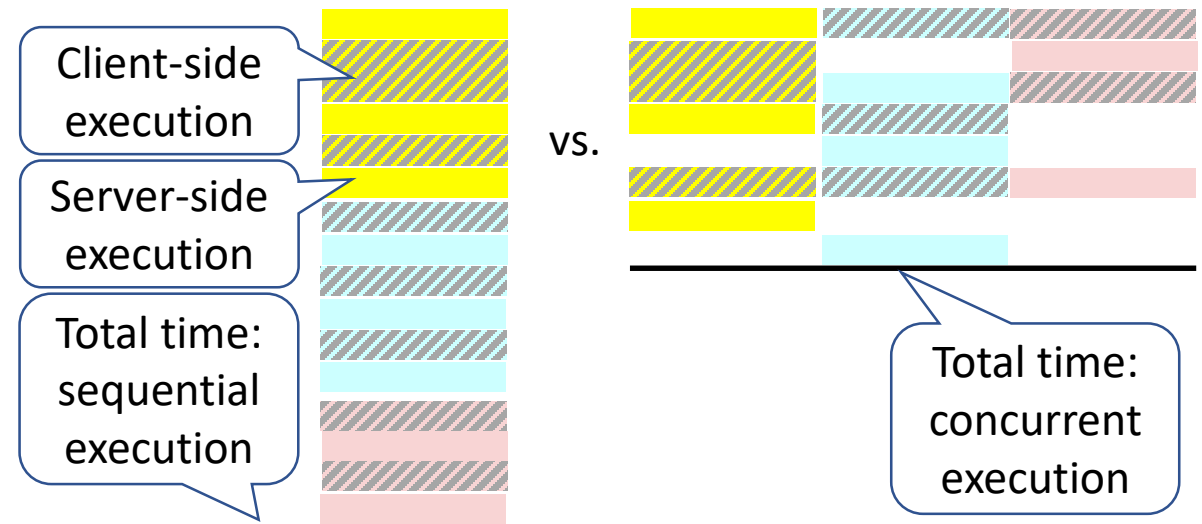
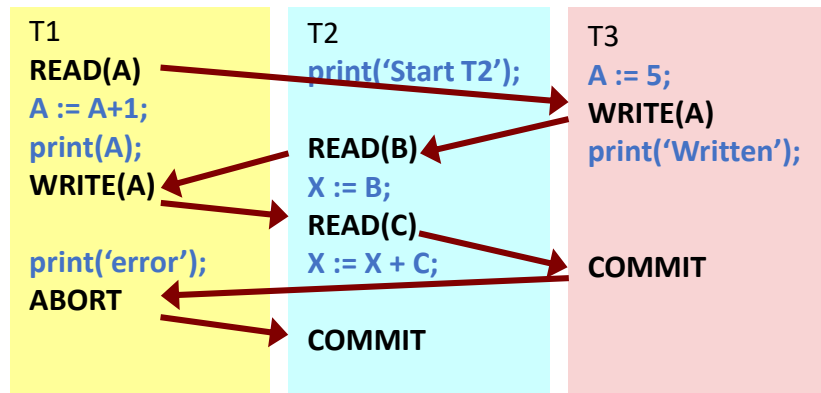
Serial schedules

- specific schedule, where all actions of a transaction are coupled together (no action interleaving)
- given a set S of transactions, we can obtain $|S|!$ serial schedules
 - All of them are correct w.r.t. The final result state of the database can differ for each of them. If the difference in result of swapping two transactions matters, they are not independent and so they should be merged into single transactions – for example, one transaction multiplies all by two, the second adds one to all. $(A*2)+1 \neq (A+1)*2$
- example:

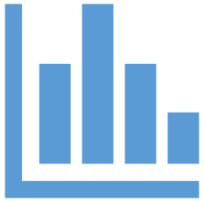


Why to interleave transactions?

- every schedule leads to interleaved **sequential** execution of transactions (there is no parallel execution of database operations)
 - simplified model justified by single storage device
- two reasons why to interleave transactions when the number of steps is the same as serial schedule
 - parallel execution of non-database operations with database operations
 - response proportional to transaction complexity (e.g., OldestEmployee vs. ComputeTaxes)
- example



Lecture overview



motivation,
ACID properties



transactions,
schedules



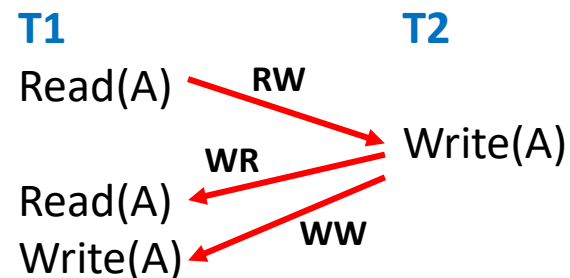
protocols for concurrent
transactions scheduling

Serializability

- a schedule is **serializable** if its execution leads to consistent database state, i.e., if the schedule is **equivalent to any serial schedule**
 - for now we consider only committed transactions and static database
 - note that non-database operations are not considered so that consistency cannot be provided for non-database state (e.g., print on console)
 - does not matter which serial schedule is equivalent (independent transactions)
- strong property
 - secures the Isolation and Consistency in ACID

“Dangers” caused by interleaving

- to achieve serializability (i.e., consistency and isolation), the action interleaving cannot be arbitrary
- there exist 3 types of local dependencies in the schedule, so-called conflict pairs
- four possibilities of reading/writing the same resource in schedule
 - read-read – ok, by just reading the transactions do not affect each other
 - write-read (WR) – T1 writes, then T2 reads – reading uncommitted data
 - read-write (RW) – T1 reads, then T2 writes – unrepeatable reading
 - write-write (WW) – T1 writes, then T2 writes – overwrite of uncommitted data

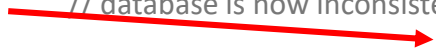


Conflicts (WR)

- reading uncommitted data (write-read conflict)
 - transaction T2 reads A that was earlier updated by transaction T1, but T1 didn't commit so far, i.e., T2 reads potentially inconsistent data
 - so-called **dirty read**

Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)
T2 computes annual interests on accounts (adds 1% per account)

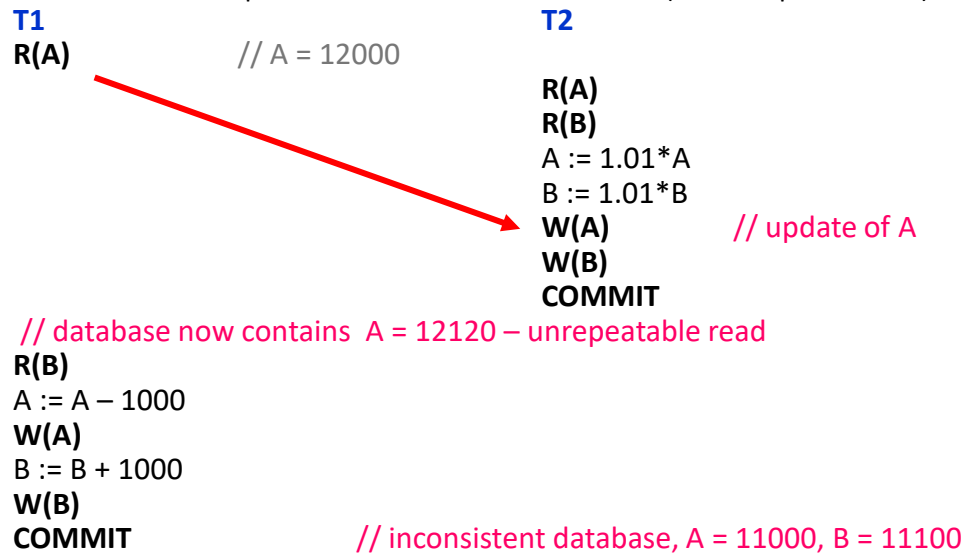
T1	T2
R(A) // A = 12000	
A := A - 1000	
W(A) // database is now inconsistent – account B still contains the old balance	
	R(A) // uncommitted data is read
	R(B)
	A := 1.01 * A
	B := 1.01 * B
	W(A)
	W(B)
	COMMIT
R(B) // B = 10100	
B := B + 1000	
W(B)	
COMMIT	
	// inconsistent database, A = 11110, B = 11100



Conflicts (RW)

- unrepeatable read (read-write conflict)
 - transaction T2 writes A that was read earlier by T1 that didn't finish yet
 - T1 cannot repeat the reading of A (A now contains another value)

Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)
T2 computes annual interests on accounts (adds 1% per account)



Conflicts (WW)

- overwrite of uncommitted data (write-write conflict)
 - transaction T2 overwrites A that was earlier written by T1 that still runs
 - loss of update (original value of A is lost)
 - inconsistency will show at so-called **blind write** (update of unread data)

Example: Set the same price to all DVDs.
(let's have two instances of this transaction, one setting price to 10 USD, second 15 USD)

T1

DVD2 := 10
W(DVD2)

DVD1 := 10
W(DVD1)

COMMIT

T2

DVD1 := 15
W(DVD1)

DVD2 := 15

W(DVD2) // overwrite of uncommitted data

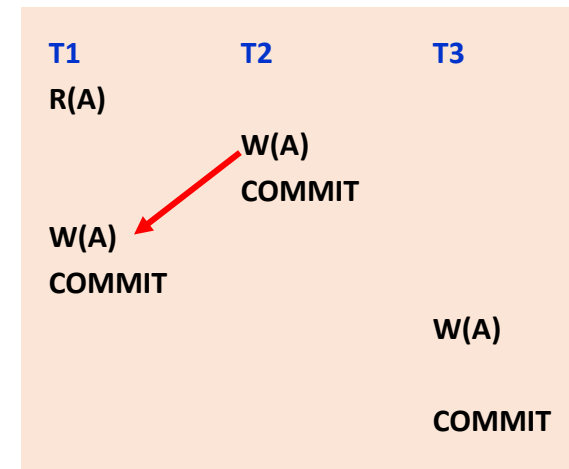
COMMIT

// inconsistent database, DVD1 = 10, DVD2 = 15

Conflict serializability

- two schedules are **conflict-equivalent** if they share the set of conflict pairs
- a schedule is **conflict-serializable** if it is conflict-equivalent to some serial schedule (on the same transactions), i.e., there are no “real” conflicts
- more restrictive than serializability (defined by consistency preservation)
- **conflict serializability** alone does not consider
 - cancelled transactions
 - ABORT/ROLLBACK, so the schedule could be **unrecoverable**
 - dynamic database (inserting and deleting database objects)
 - so-called **phantom** may occur

Example: schedule, that is **serializable**
(serial schedule $\langle T1, T2, T3 \rangle$), but **is not conflict-serializable**
(writes in T1 and T2 are in wrong order)

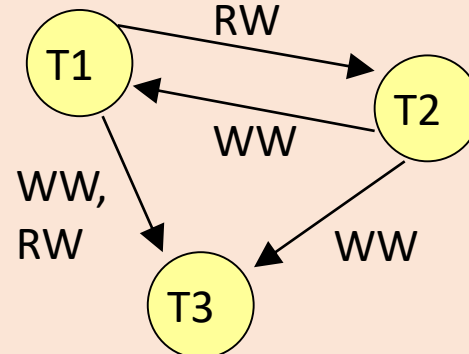


Detection of conflict serializability

- **precedence graph** (also serializability graph) on a schedule
 - nodes T_i are **committed** transactions
 - arcs represent RW, WR, WW conflicts in the schedule
- schedule is conflict-serializable if its precedence graph is **acyclic**

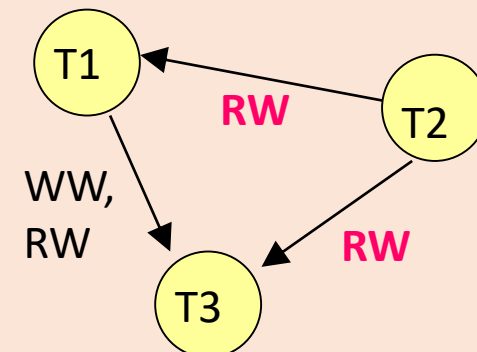
Example: not conflict-serializable

T1 **T2** **T3**
R(A)
W(A)
COMMIT
W(A)
COMMIT
W(A)
COMMIT



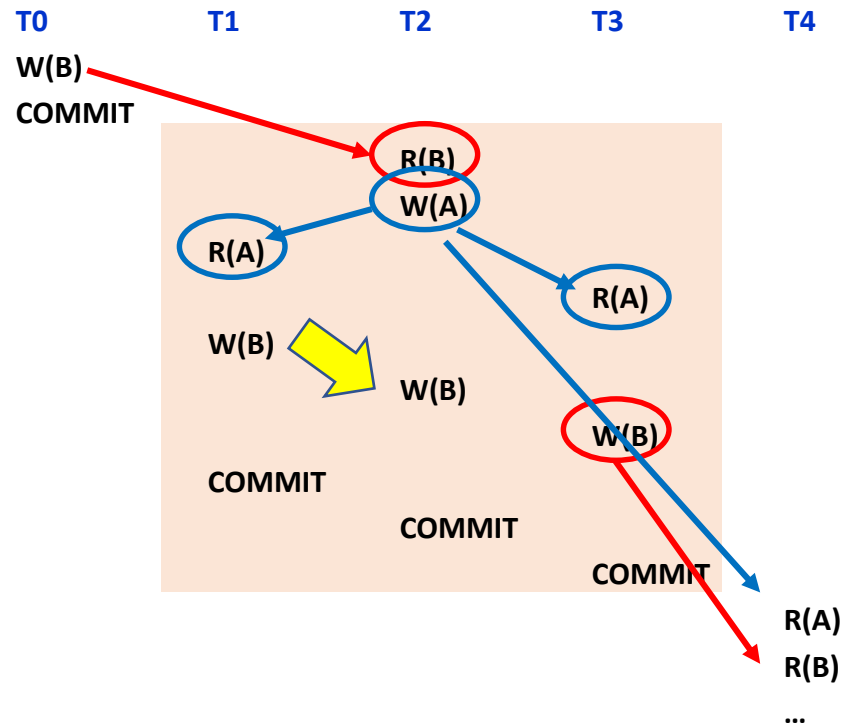
Example: conflict-serializable

T1 **T2** **T3**
R(A)
R(A)
COMMIT
W(A)
COMMIT
W(A)
COMMIT



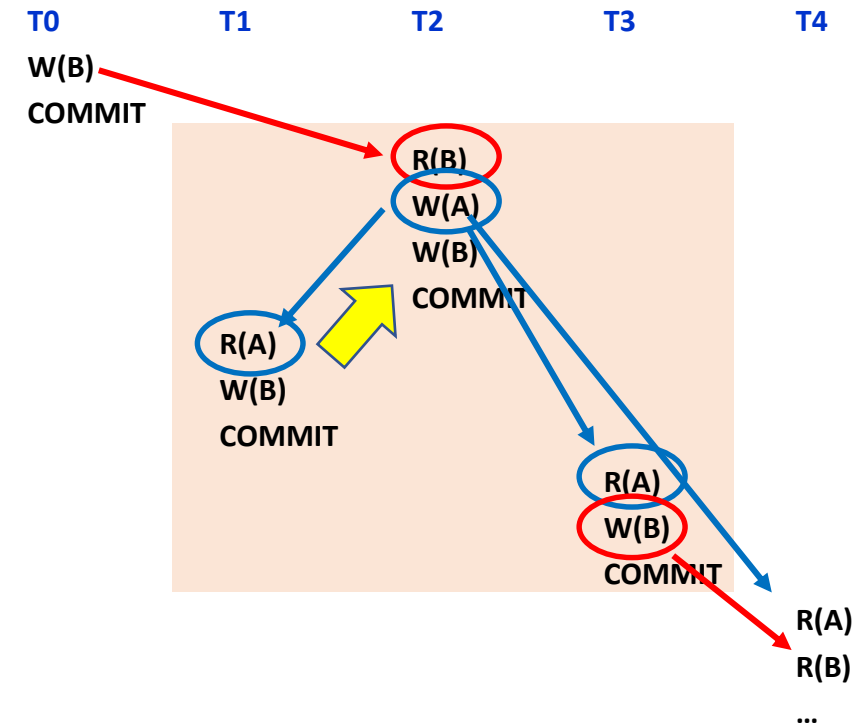
Conflict serializability – Too strict?

Example: schedule, that is **(view-)serializable**
(serial schedule $\langle T1, T2, T3 \rangle$), but **is not conflict-serializable**
(Precedence graph contains cycles)



<http://infolab.stanford.edu/~ullman/dscb/vs-old.pdf>

Equivalent serial schedule

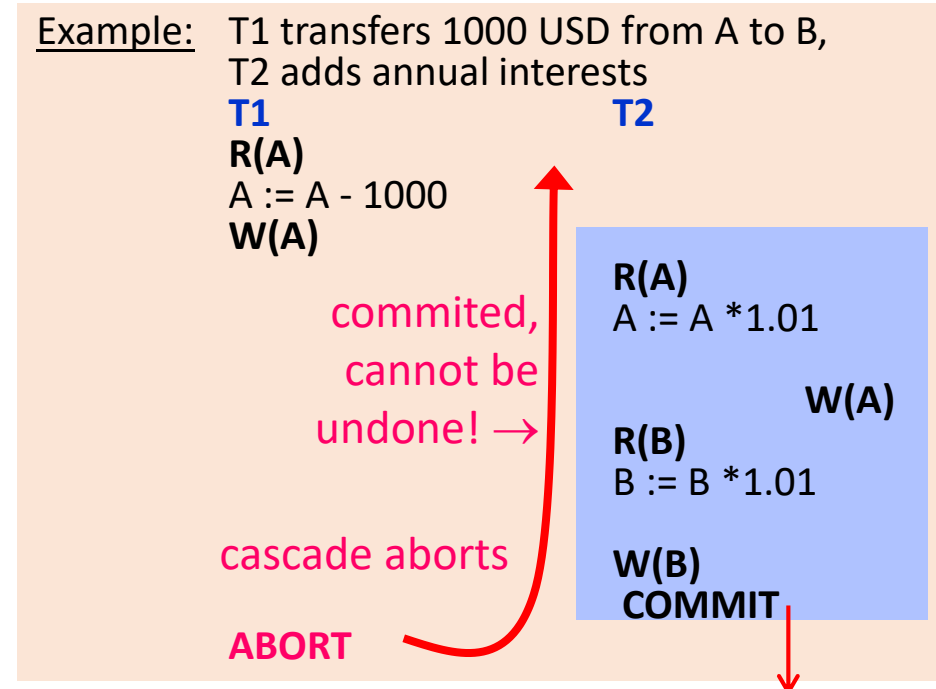


View serializability

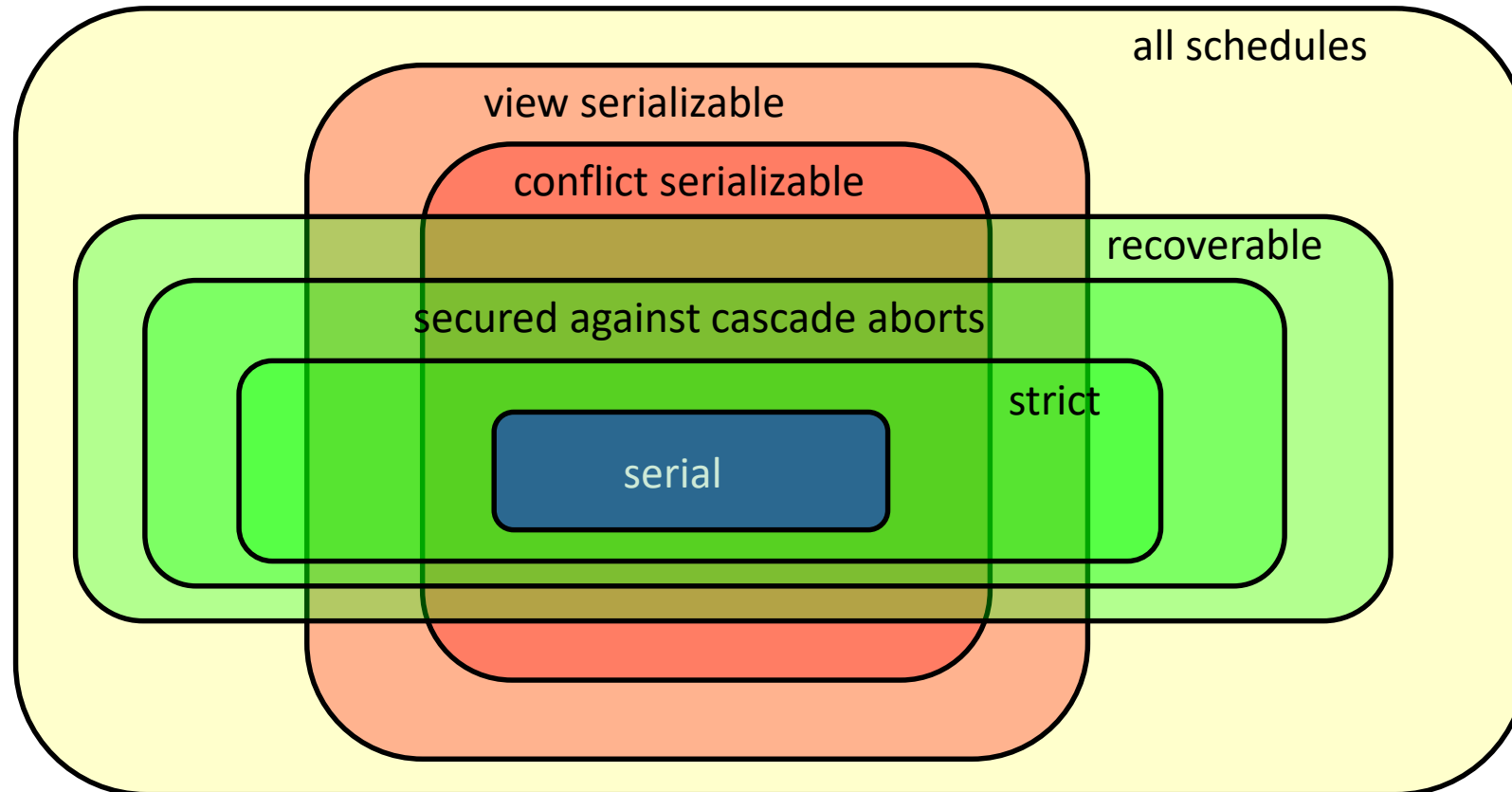
- Two schedules are **view-equivalent** if
 - Transaction T_i reads the initial value of X in one schedule if and only if it reads the initial value in the second schedule
 - Operation O_i in transaction T_i reads the value of X produced by operation O_j in transaction T_j in one schedule if and only if it reads the same value in the second schedule
 - Transaction T_i writes the final value of X in one schedule if and only if it writes the initial value in the second schedule
- A schedule is **view-serializable** if it is view-equivalent to some serial schedule
- Every conflict-serializable schedule is also view-serializable, but not vice-versa.

Unrecoverable schedule

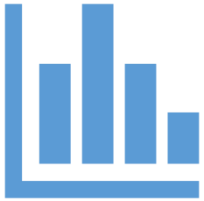
- at this moment we extend the transaction model by ABORT which brings another “danger” – **unrecoverable schedule**
 - one transaction aborts so that undos of every write must be done, however, this cannot be done for already committed transactions that read changes cause by the aborted transaction (Durability property of ACID)
- in **recoverable schedule**
 - a transaction T is committed after all other transactions commit that affected T (i.e., they changed data later read by T)
 - moreover, if reading changed data is allowed only for committed transactions, we avoid **cascade aborts of transactions**
 - in the example T2 would begin after T1’s abort



Classes of schedules



Lecture overview



motivation,
ACID properties



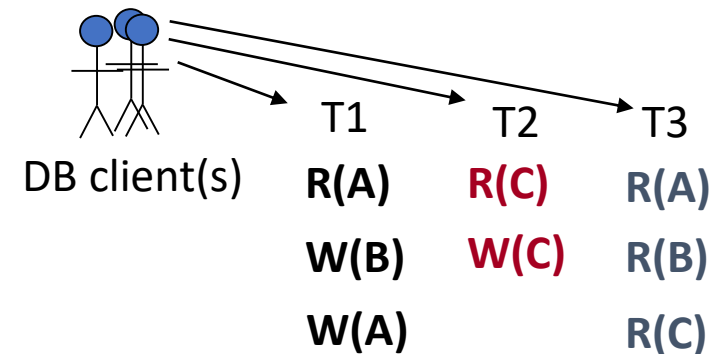
transactions,
schedules



**protocols for concurrent
transactions scheduling**

Protocols for concurrent transaction scheduling

- transaction scheduler works under some protocol that allows to guarantee the ACID properties and maximal throughput
- pessimistic control (highly concurrent workloads)
 - locking protocols
 - time stamps
- optimistic control (not very concurrent workloads)
- why protocol?
 - the scheduler cannot create the entire schedule beforehand
 - scheduling is performed in local time context using **protocol**
 - dynamic transaction execution, branching parts in code



Locking protocols

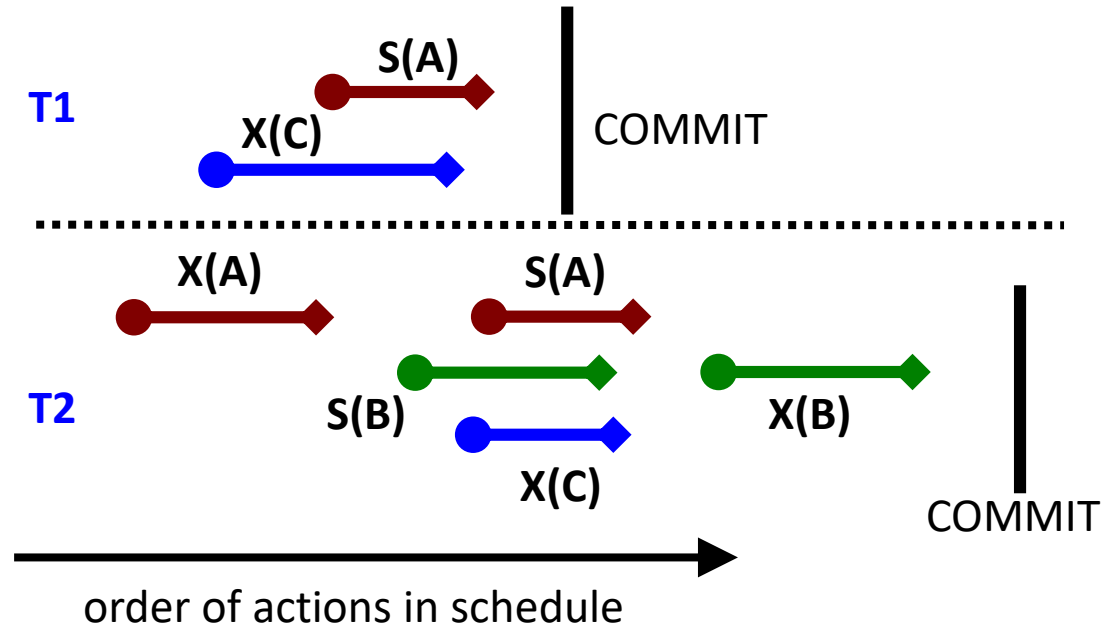
- locking of database entities can be used to control the order of reads and writes and so to secure the conflict serializability
- **exclusive locks**
 - **X(A)**, locks A so that reads and writes of A is allowed only to the lock owner/creator
 - can be granted to just one transaction
- **shared locks**
 - **S(A)**, only reads of A allowed – the lock owner can read and is sure that A cannot change
 - can be granted to (shared by) multiple transactions (if X(A) not granted already)
- unlocking by **U(A)**
- if a lock is required for transaction that is not available (granted to another one), the transaction execution is suspended and wait for releasing the lock
 - in schedule, the lock request is denoted, followed by empty rows of waiting
- the un/locking code is added by the transaction scheduler
 - i.e., operation on locks appear just in the schedules, not in the original transaction code

Well-formed transactions

- Transaction is well-formed, iff
 - Before reading from A the transaction have requested and obtained at least shared lock
 - Before writing to A the transaction have requested and obtained at least exclusive lock
 - At last at the end of transaction all locks are unlocked
- The locks can be requested explicitly, or inserted implicitly by the scheduler

Example: schedule with locking and well-formed transactions

T1	T2
$X(C)$	$X(A)$
$R(C)$	$W(A)$
$W(C)$	$U(A)$
$S(A)$	$S(B)$
$R(A)$	$R(B)$
$U(C)$	$X(C)$
$U(A)$	$S(A)$
COMMIT	$W(C)$
	$R(A)$
	$U(B)$
	$U(C)$
	$U(A)$
	$X(B)$
	$W(B)$
	$U(B)$
	COMMIT



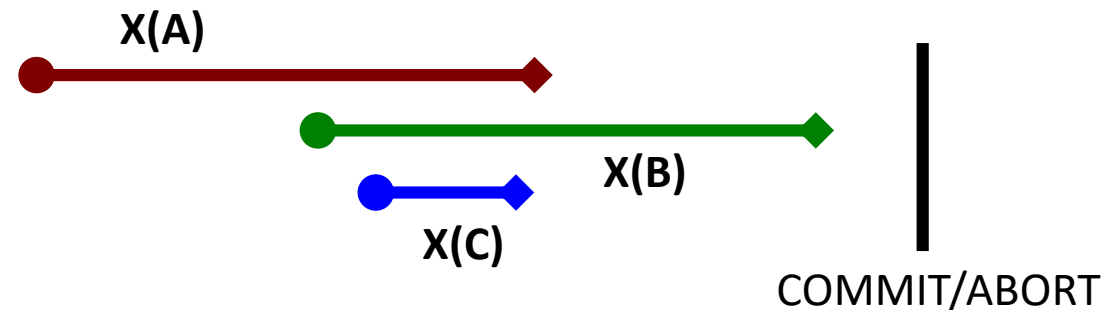
Two-phase locking protocol (2PL)

2PL applies one additional rule for building the schedule (among rules for well-formed transactions):

- 1) transaction **cannot requests a lock**, if it already released one (regardless of the locked entity)

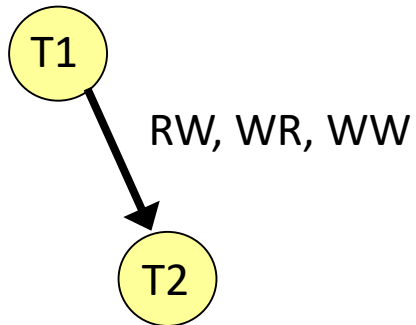
Two obvious phases – locking and unlocking

Example: 2PL adjustment of the second transaction in the previous schedule



Properties of 2PL

- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict-serializable**
- 2PL does **not guarantee recoverable schedules**



Example: 2PL-compliant schedule, but not recoverable, if T1 aborts

T1

X(A)

R(A)

W(A)

U(A)

T2

X(A)

R(A)

A := A * 1.01

W(A)

S(B)

U(A)

R(B)

B := B * 1.01

W(B)

U(B)

COMMIT

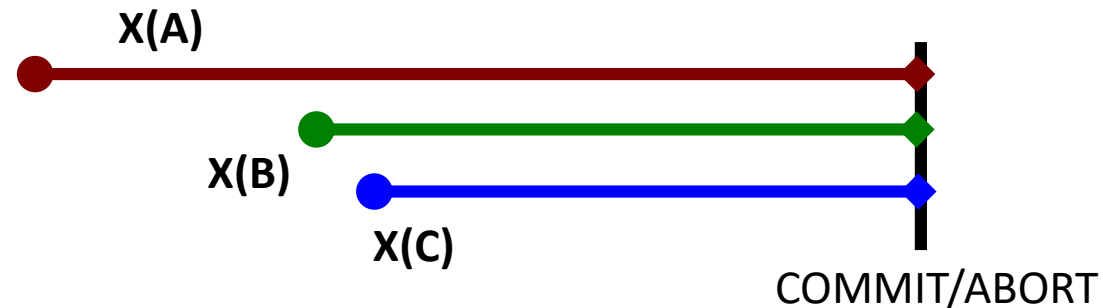
ABORT / COMMIT

Strict 2PL

Strict 2PL makes the second rule of 2PL stronger, so that both rules become:

- 1) if a transaction wants to read (write) an entity A, it must first acquire a shared (exclusive) lock on A
- 2) **all locks are released at the transaction termination**

Example: strict 2PL adjustment of second transaction in the previous example



Insertions of $U(A)$ is not needed (implicit at the time of COMMIT/ABORT).

Properties of strict 2PL

- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict-serializable**
- moreover, strict 2PL ensures
 - schedule **recoverability**
 - avoids **cascade aborts**

Example: schedule built using strict 2PL

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
X(C)	
R(C)	
	R(B)
	W(B)
	COMMIT
W(C)	
ABORT / COMMIT	

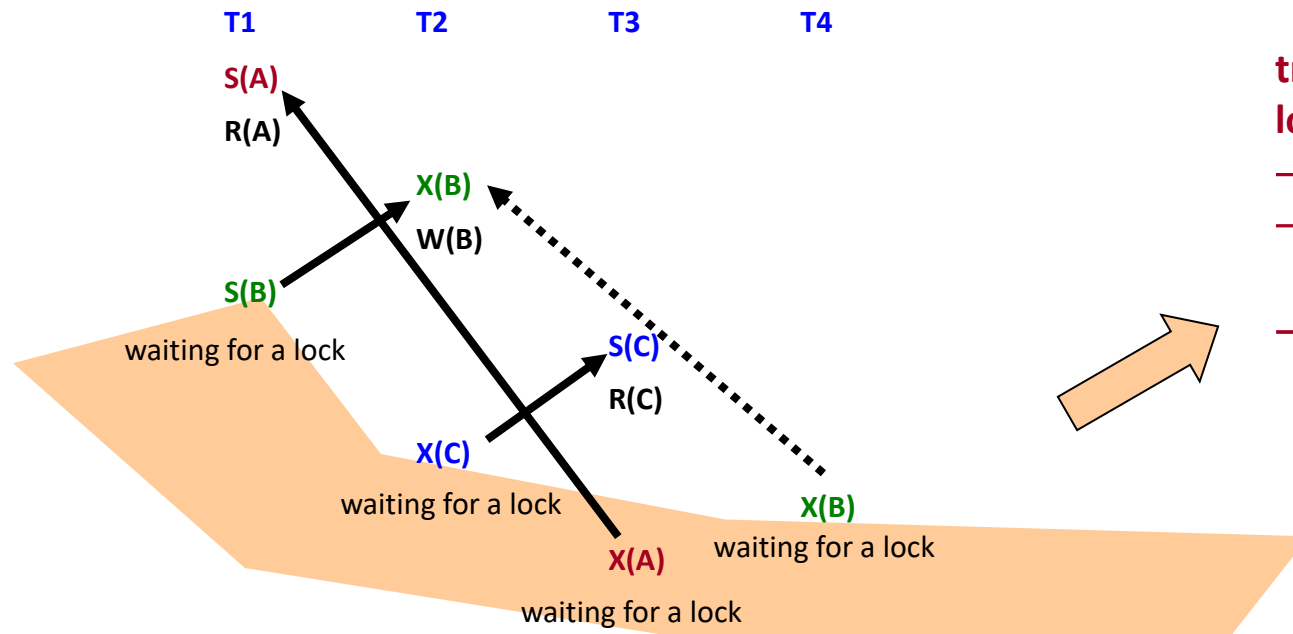
T1

T2

Deadlock

- during transaction execution it may happen that transaction T1 requests a lock that was already granted to T2, but T2 cannot release it because it waits for another lock kept by T1
 - could be generalized to multiple transactions, T1 waits for T2, T2 waits for T3, ..., Tn waits for T1
- strict 2PL cannot prevent from deadlock (not speaking about the weaker protocols)

Example:



transactions T1, T2 and T3 wait for a lock in circle

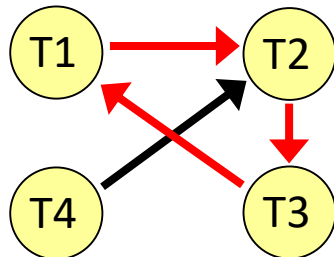
- no one can release a lock
- scheduler cannot schedule nor execute transactions
- **deadlock**

Deadlock detection

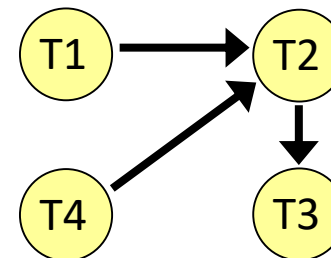
- deadlock can be detected by repeated checking the waits-for graph
- **waits-for graph** is dynamic graph that captures the waiting of transactions for locks
 - nodes are active transactions
 - an arc denotes waiting of transaction for lock kept by another transaction
 - a cycle in the graph = **deadlock**

Example: graph to the previous example

(a) T3 requests X(A)



(b) T3 does not request X(A)



Deadlock resolution and prevention

- deadlocks are usually not very frequent, so the resolution could be simple
 - abort of the waiting transaction and its restart (user will not notice)
 - testing waits-for graph – if deadlock occurs, abort and restart a transaction in the cycle
 - such transaction is aborted, that
 - holds the smallest number of locks
 - performed the least amount of work
 - is far from completion
 - aborted transaction is not aborted again (if another deadlock occurs)
- deadlocks could be prevented
 - prioritizing
 - each transaction has a priority (e.g., time stamp), while if T1 requests a lock kept by T2, the lock manager chooses between two strategies
 - **wait-die** – if T1 has higher priority, it can wait, if not, it is aborted and restarted
 - **wound-wait** – if T1 has higher priority, T2 is aborted, otherwise T1 waits
 - conservative 2PL protocol
 - all locks possibly used must be requested at the beginning
 - hard to guess all relevant lock, not used in practice for high locking overhead

Phantom

- now consider dynamic database, that is, allowing inserts and deletes
- if one transaction works with some *semantic set* of data entities, while another transaction (*logically*) changes this set (inserts or deletes), it could lead to inconsistent database (inserializable schedule)
 - why: T1 locks all entities that at the given moment are relevant (e.g., fulfill some WHERE condition of a SELECT command)
 - during execution of T1 a new transaction T2 could logically extend the set of entities (i.e., at that moment the number of locks defined by WHERE would be larger), so that some entities are locked and some are not
- applied also to strict 2PL

Example – phantom

T1: find the oldest male and female employees

(**SELECT * FROM** Employees ...) + **INSERT INTO** Statistics ...

T2: insert new employee Phill and delete employee Eve (employee replacement)

(**INSERT INTO** Employees ..., **DELETE FROM** Employees ...)

Initial state of the database: {[**Peter**, 52, m], [**John**, 46, m], [**Eve**, 55, f], [**Dana**, 30, f]}

T1

T2

S(Peter)

S(John) // lock men

$M = \max\{R(\text{Peter}), R(\text{John})\}$

Insert(Phill, 72, m)

X(Eve)

X(Dana) // lock women

Delete(Eve)

COMMIT

phantom

new male employee can be inserted, although **all men** should be locked

S(Dana) // lock women

$F = \max\{R(\text{Dana})\}$

Insert(M, F) // result is inserted into table Statistics

COMMIT

Although the schedule is **strict 2PL** compliant, the result [**Peter**, **Dana**] is not correct as it does not follow the serial schedule T1, T2, resulting in [**Peter**, **Eve**], nor T2, T1, resulting in [**Phill**, **Dana**].

Phantom – prevention

- if there exist indexes (e.g., B⁺-trees) on the entities defined by the „lock condition“, it is possible to “watch for phantom” at the index level – **index locking**
 - external attempt for the set modification is identified by the index locks updated
 - as an index usually maintains just one attribute, its applicability is limited
- if there do not exist indexes, everything relevant must be locked
 - e.g., entire table or even multiple tables must be locked
- generalization of index locking is **predicate locking**, when the locks are requested for the logical sets, not particular data instances
 - however, this is hard to implement and so not used much in practice

Isolation levels

- the more strict locking protocol, the worse performance of concurrent transaction management
 - for different reasons different protocols can be chosen, in order to achieve maximal performance for sufficient isolation of transactions
- SQL-92 defines isolation levels

Level	Protocol	WR	RW	Phantom
READ UNCOMMITTED (read only transactions)	No	maybe	maybe	maybe
READ COMMITTED	S2PL for X() + 2PL for S()	No	maybe	maybe
REPEATABLE READ	S2PL	No	No	maybe
SERIALIZABLE	S2PL + phantom prevention	No	No	No

Optimistic (not locking) protocols

- if concurrently executed transactions are not often in conflict (not competing for resources), the locking overhead is unnecessarily large
- 3-phase optimistic protocol
 - **Read**: transaction reads data from database but writes into its private local data space
 - **Validation**: if the transaction wants to commit, it forwards the private data space to the transaction manager (i.e., request on database update)
 - the transaction manager decides if the update is in conflict with another transaction
 - if there is a conflict, the transaction is aborted and restarted
 - if not, the last phase takes place:
 - **Write**: the private data space is copied into the database

Timestamps

Every transaction receives a unique timestamp
 $TS(T)$

- The system's clock
 - A unique counter, incremented by the scheduler
- The timestamp order defines the serialization order of the transaction

Timestamps

Associate to each element X :

- $R_T(X)$ = the highest timestamp of any transaction that read X
- $W_T(X)$ = the highest timestamp of any transaction that wrote X

These are associated to each page X in the buffer pool

Timestamps - Main Idea

For any two conflicting actions check that their order is correct:

In each of these cases

- $w_U(X)$ before $r_T(X)$
- $r_U(X)$ before $w_T(X)$
- $w_U(X)$ before $w_T(X)$

Check that $TS(U) < TS(T)$

Timestamp-based Scheduling

When a transaction T requests $R(X)$ or $W(X)$,
the scheduler examines $RT(X)$, $WT(X)$

- Grant request, if timestamps are OK
- Rollback T (and restart it with later timestamp)

END OF THE EIGHT LECTURE