# Database Management Systems

Lecture 2

Transactions. Concurrency Control

Sabina S. CS

Conflict Serializability
- $C$ – set of transactions
- $Sch(C)$ – the set of schedules for $C$
- $Op(C)$ – set of operations of the transactions in $C$
- consider schedule $S \in Sch(C)$
- the *conflict relation* of $S$ is defined as:
  - *conflict(S)* = {(op$_1$, op$_2$) | op$_1$, op$_2$ $\in$ *Op(C),* op$_1$ occurs before op$_2$ in $S$, op$_1$ and op$_2$ are in conflict}

Conflict Serializability
- $C$ – set of transactions
- $Sch(C)$ – the set of schedules for $C$
- two schedules $S_1$ and $S_2 \in Sch(C)$ are _conflict equivalent_, written $S_1 \equiv_c S_2$, if $conflict(S_1) = conflict(S_2)$, i.e.:
    - $S_1$ and $S_2$ contain the same operations of the same transactions and
    - every pair of conflicting operations is ordered in the same manner in $S_1$ and $S_2$

# Conflict Serializability

**S1**

| T1 | T2 |
|---|---|
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |

**S2**

| T1 | T2 |
|---|---|
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |

conflict(S1) = conflict(S2)
=> S1 $\equiv_c$ S2

conf(S1) = {(Read(T1, A), Write(T2, A)), (Write(T1, A), Read(T2, A)), (Write(T1, A), Write(T2, A)), (Read(T1, B), Write(T2, B)), (Write(T1, B), Read(T2, B)), (Write(T1, B), Write(T2, B))}

# Conflict Serializability

## S1

| T1 | T2 |
|---|---|
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |

## S3

| T1 | T2 |
|---|---|
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |

$$\text{conflict}(S1) \neq \text{conflict}(S3)$$
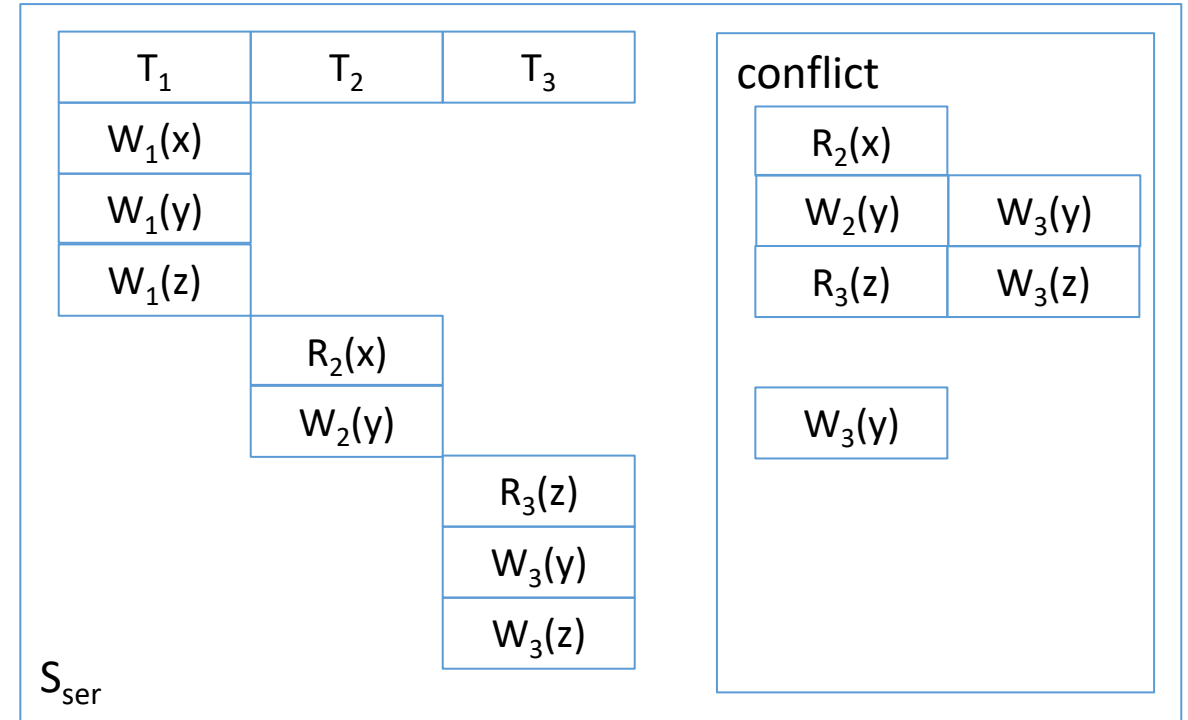$$\Rightarrow S1 \not\equiv_c S3$$

Conflict Serializability

- $C$ – set of transactions
- $Sch(C)$ – the set of schedules for $C$
- let $S$ be a schedule in $Sch(C)$
- schedule $S$ is <u>*conflict serializable*</u> if there exists a serial schedule $S_0 \in Sch(C)$ such that $S \equiv_c S_0$, i.e., $S$ is conflict equivalent to some serial schedule

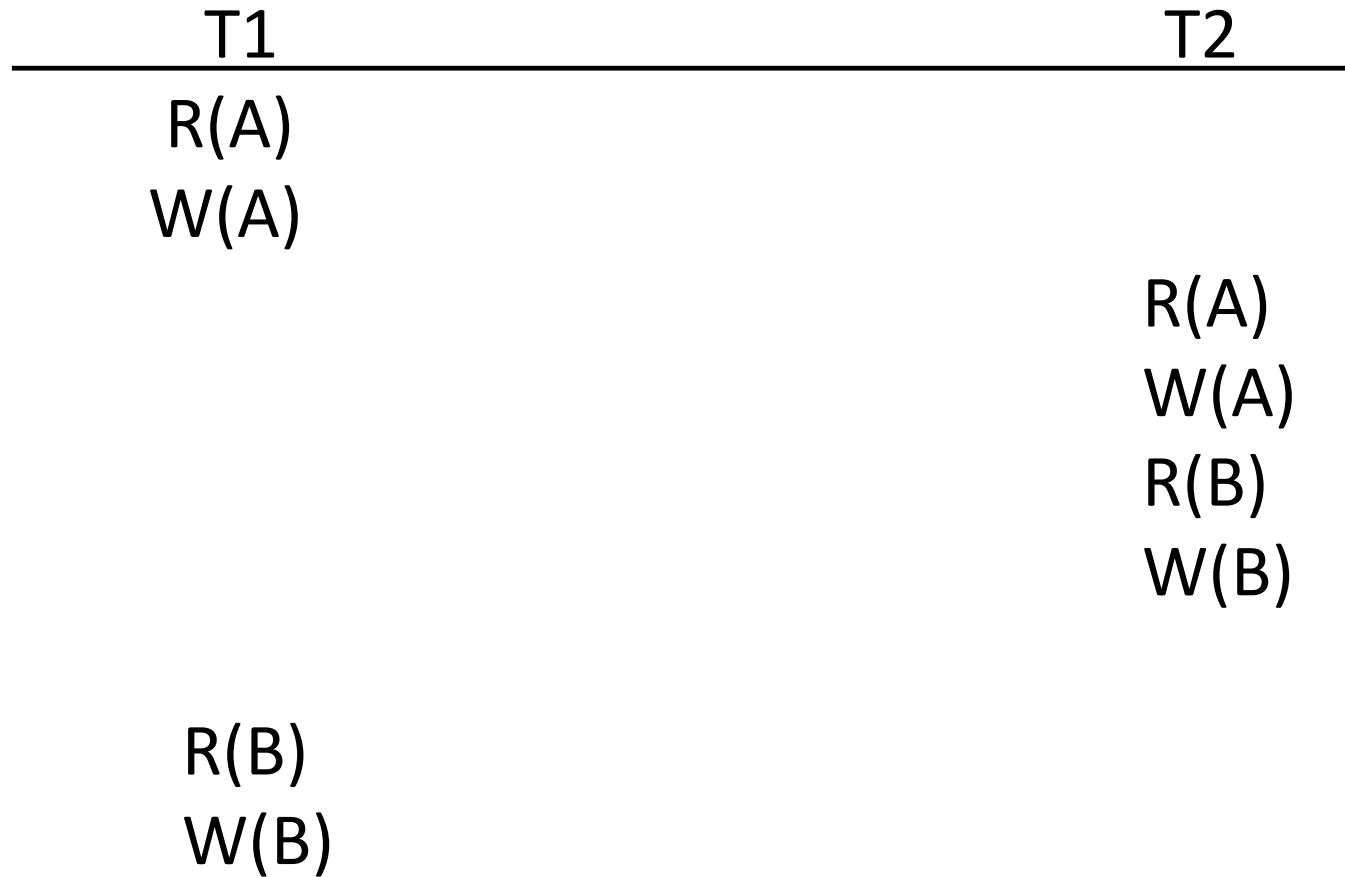# Conflict Serializability
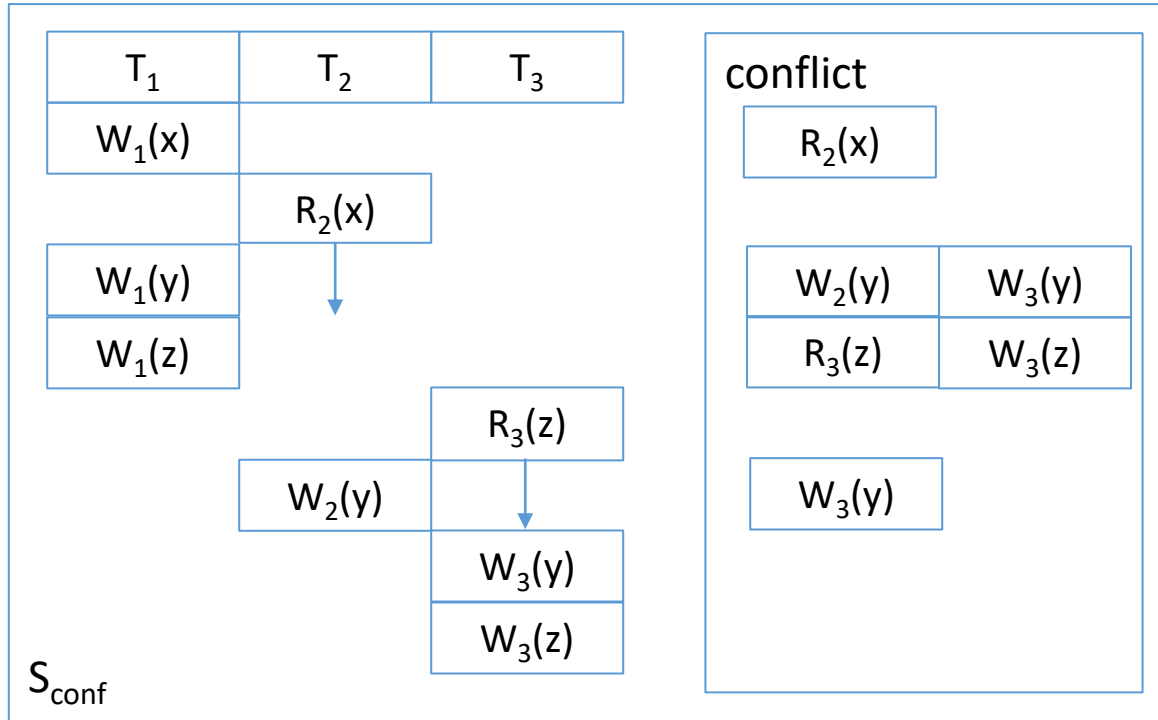


conflict serializable schedule

serial schedule
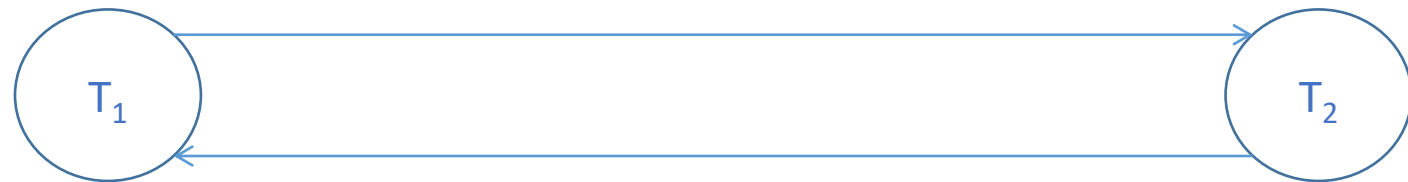
Conflict Serializability - Precedence Graph

- let $S$ be a schedule in *Sch(C)*
- the *precedence graph* (*serializability graph*) of $S$ contains*:*
    - one node for every committed transaction in $S$
    - an arc from $T_i$ to $T_j$ if an action in $T_i$ precedes and conflicts with one of the actions in $T_j$

- Theorem:
    - a schedule $S \in$ *Sch(C)* is conflict serializable if and only if its precedence graph is acyclic

# Conflict Serializability - Precedence Graph

- example - a schedule that is not conflict serializable:

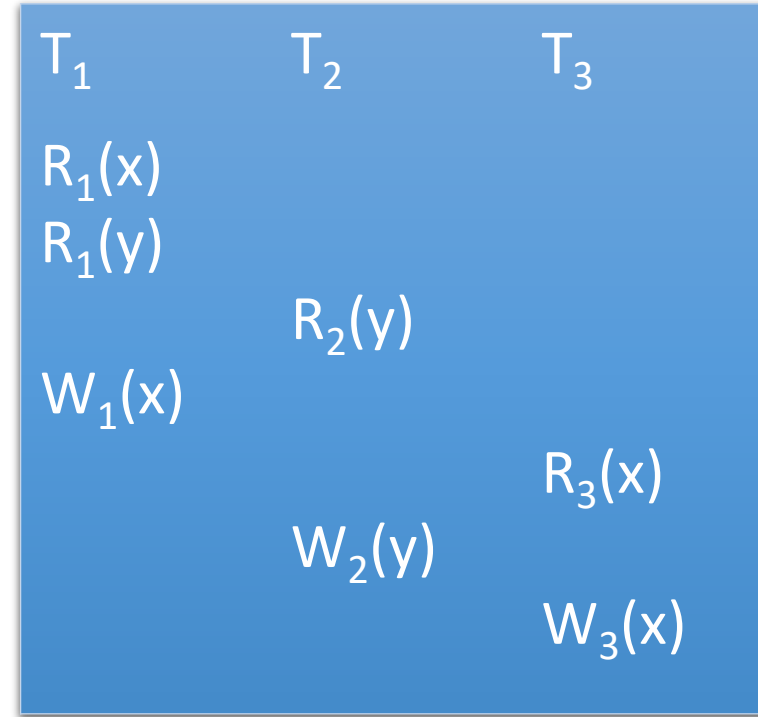| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |
| | R(B) |
| | W(B) |
| R(B) | |
| W(B) | |

- the precedence graph has a cycle:
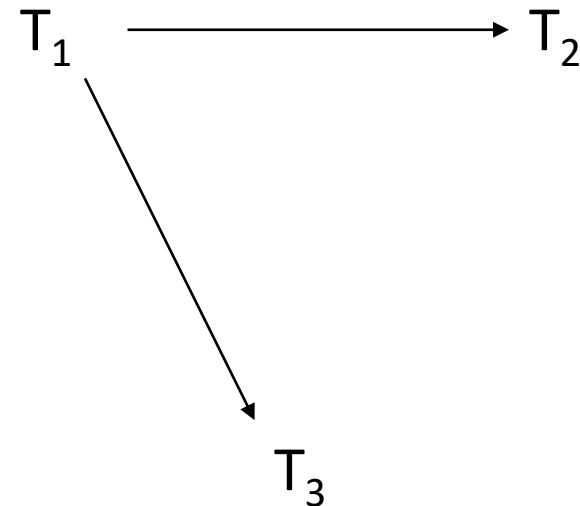
Conflict Serializability - Precedence Graph

• algorithm to test the conflict serializability of a schedule $S \in Sch(C)$

1. create a node labeled $T_i$ in the precedence graph for every committed transaction $T_i$ in the schedule

2. create an arc $(T_i, T_j)$ in the precedence graph if $T_j$ executes a Read(A) after a Write(A) executed by $T_i$

3. create an arc $(T_i, T_j)$ in the precedence graph if $T_j$ executes a Write(A) after a Read(A) executed by $T_i$

4. create an arc $(T_i, T_j)$ in the precedence graph if $T_j$ executes a Write(A) after a Write(A) executed by $T_i$

5. $S$ is conflict serializable if and only if the resulting precedence graph has no cycles

# Conflict Serializability - Precedence Graph

- examples
- let $S_1$ be a schedule over $\{T_1, T_2, T_3\}$:

| $T_1$ | $T_2$ | $T_3$ |
|-------|-------|-------|
| $R_1(x)$ | | |
| $R_1(y)$ | | |
| | $R_2(y)$ | |
| $W_1(x)$ | | |
| | | $R_3(x)$ |
| | $W_2(y)$ | |
| | | $W_3(x)$ |

- the precedence graph for $S_1$:

$T_1 \longrightarrow T_2$

$T_1 \searrow$

$T_3$

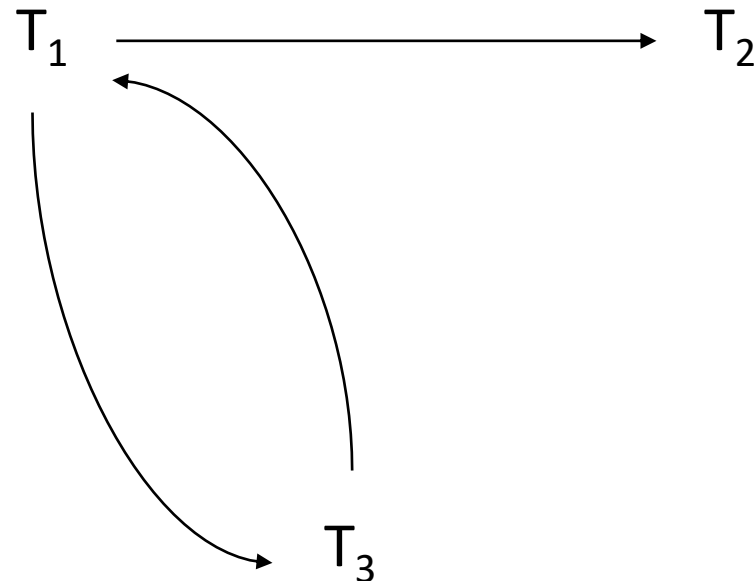# Conflict Serializability - Precedence Graph

- examples
- let $S_2$ be a schedule over $\{T_1, T_2, T_3\}$:

| $T_1$ | $T_2$ | $T_3$ |
|-------|-------|-------|
| $R_1(x)$ | | |
| $R_1(y)$ | | |
| | | $R_3(x)$ |
| | $R_2(y)$ | |
| $W_1(x)$ | | |
| | $W_2(y)$ | |
| | | $W_3(x)$ |

- the precedence graph for $S_2$:

$$T_1 \longrightarrow T_2$$

$T_1 \rightleftharpoons T_3$

# Conflict Serializability

- every conflict serializable schedule is serializable (in the absence of inserts / deletes, when items can only be updated)
- there are serializable schedules that are not conflict serializable
- $S$ is equivalent to the serial execution of transactions $T_1$, $T_2$, $T_3$ (in this order), but it is not conflict equivalent to this serial schedule (the write operations in $T_1$ and $T_2$ are ordered differently)

$S$

| $T_1$ | $T_2$ | $T_3$ |
|-------|-------|-------|
| $R_1(A)$ | | |
| | $W_2(A)$ | |
| | commit | |
| $W_1(A)$ | | |
| commit | | |
| | | $W_3(A)$ |
| | | commit |
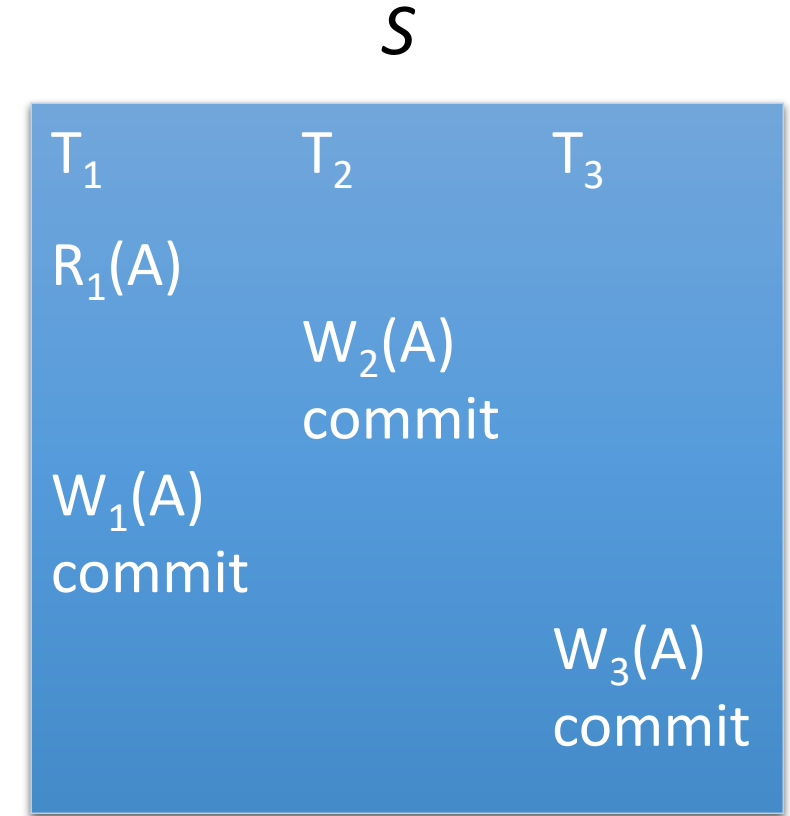
View Serializability

- conflict serializability is a sufficient condition for serializability, but it is not a necessary one
- *view serializability*
  - a more general, sufficient condition for serializability
  - based on *view-equivalence,* a less stringent form of equivalence

View Serializability
- *C* – set of transactions
- *Sch(C)* – the set of schedules for *C*
- let $T_i$, $T_j \in C$, $S_1$, $S_2 \in Sch(C)$; $S_1$ and $S_2$ are *view equivalent,* written $S_1 \equiv_v S_2$, if the following conditions are met:
  - if $T_i$ reads the initial value of V in $S_1$, then $T_i$ also reads the initial value of V in $S_2$;
  - if $T_i$ reads the value of V written by $T_j$ in $S_1$, then $T_i$ also reads the value of V written by $T_j$ in $S_2$;
  - if $T_i$ writes the final value of V in $S_1$, then $T_i$ also writes the final value of V in $S_2$.
- i.e.:

    each transaction performs the same computation in $S_1$ and $S_2$

and

    $S_1$ and $S_2$ produce the same final database state.

# View Serializability

S1

| T1 | T2 |
|---|---|
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |

S2

| T1 | T2 |
|---|---|
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |

$$S1 \equiv_v S2$$

# View Serializability

**S1**

| T1 | T2 |
|---|---|
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |

**S3**

| T1 | T2 |
|---|---|
| | Read(A) |
| | A := A * 0.2 |
| | Write(A) |
| | Read(B) |
| | B := B + 300 |
| | Write(B) |
| Read(A) | |
| A := A − 100 | |
| Write(A) | |
| Read(B) | |
| B := B + 200 | |
| Write(B) | |

$$S1 \not\equiv_v S3$$

# View Serializability

- $C$ – set of transactions
- $Sch(C)$ – the set of schedules for $C$
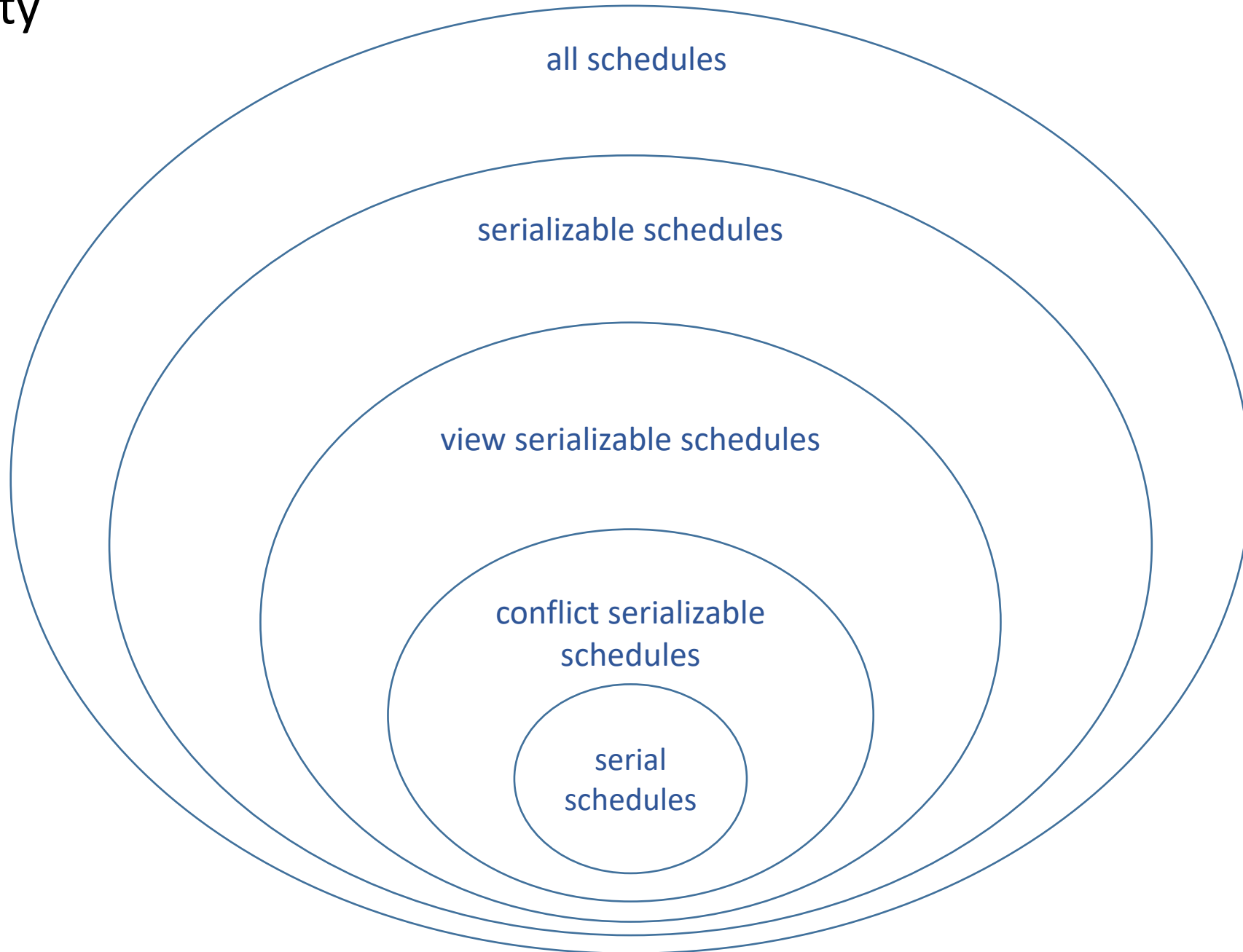- a schedule $S \in Sch(C)$ is _view serializable_ if there exists a serial schedule $S_0 \in Sch(C)$ such that $S \equiv_v S_0$, i.e., $S$ is view equivalent to some serial schedule

| $T_1$ | $T_2$ | $T_3$ |
|-------|-------|-------|
| $R_1(A)$ | | |
| | $W_2(A)$ | |
| | commit | |
| $W_1(A)$ | | |
| commit | | |
| | | $W_3(A)$ |
| | | commit |

| $T_1$ | $T_2$ | $T_3$ |
|-------|-------|-------|
| $R_1(A)$ | | |
| $W_1(A)$ | | |
| commit | | |
| | $W_2(A)$ | |
| | commit | |
| | | $W_3(A)$ |
| | | commit |

# Serializability



all schedules

serializable schedules

view serializable schedules

conflict serializable schedules

serial schedules

Sabina S. CS
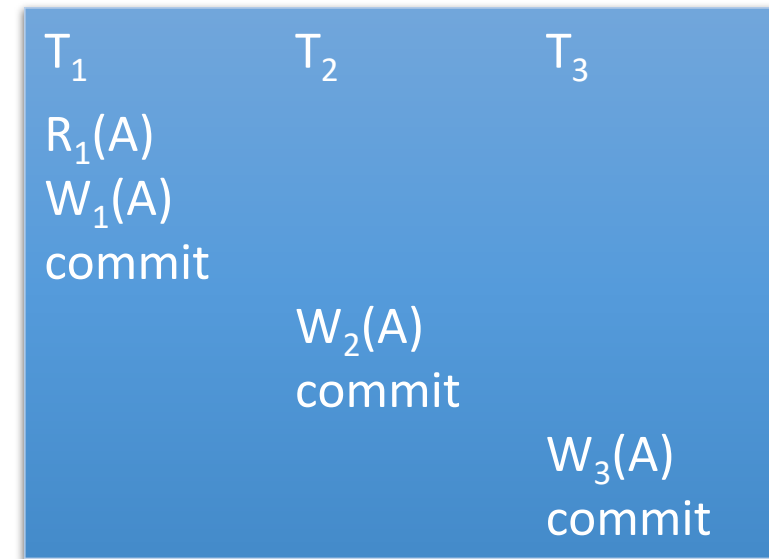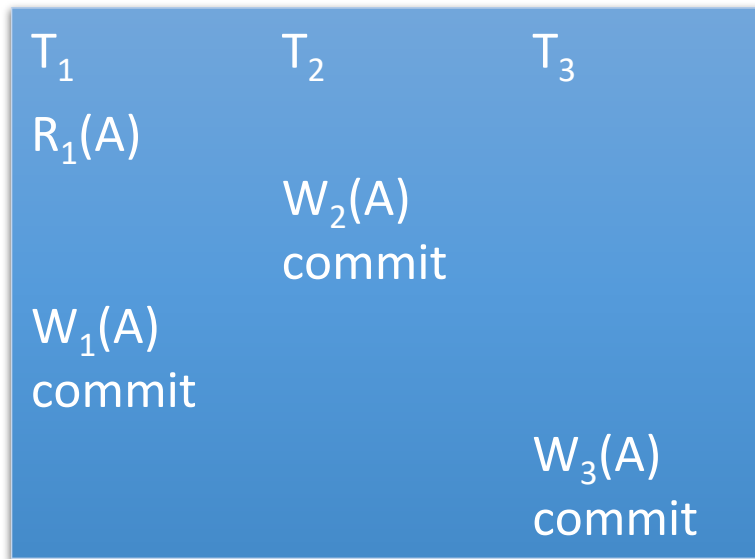
# Recoverable Schedules

- consider schedule $S$ over $\{T_1, T_2\}$

| $T_1$ | $T_2$ |
|---|---|
| Read(x) | |
| x := x + 10 | |
| Write(x) | |
| | Read(x) |
| | x := x * 5 |
| | Write(x) |
| | Read(y) |
| | y := y * 5 |
| | Write(y) |
| | commit |
| abort | |

- $T_2$ operates on a value of x that shouldn't have been in the database, since $T_1$ aborted

# Recoverable Schedules

| $T_1$ | $T_2$ |
|---|---|
| Read(x) | |
| x := x + 10 | |
| Write(x) | |
| | Read(x) |
| | x := x * 5 |
| | Write(x) |
| | Read(y) |
| | y := y * 5 |
| | Write(y) |
| | commit |
| abort | |
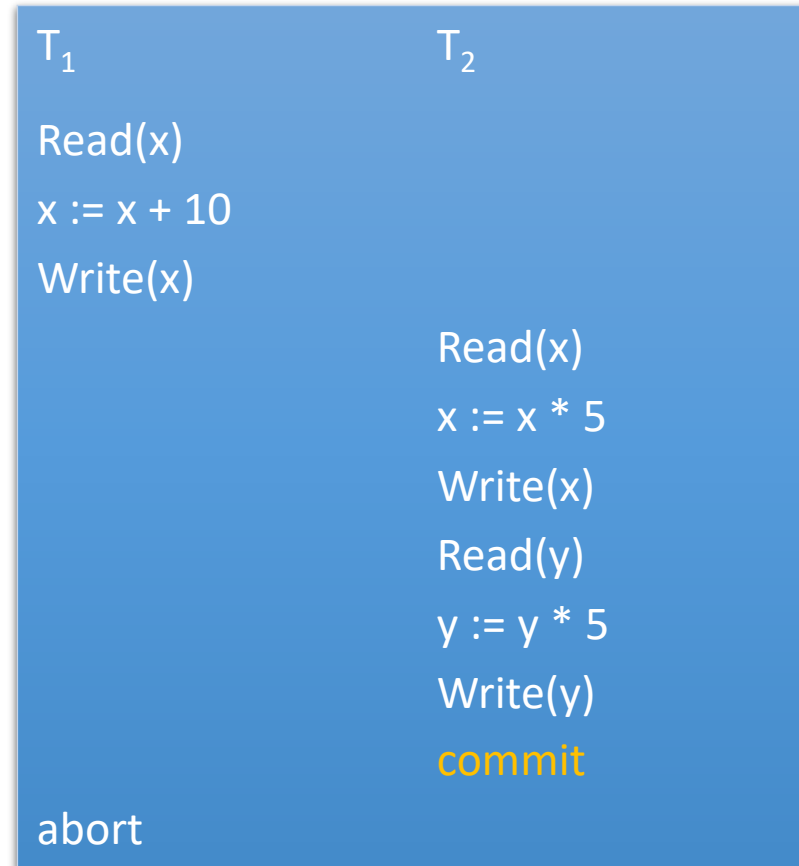
- cannot cascade the abort of $T_1$, since $T_2$ has already committed
- schedule *S* is *unrecoverable*

# Recoverable Schedules

- <u>recoverable schedule</u>
    - a schedule in which a transaction T commits only after all transactions whose changes T read commit

# Avoiding Cascading Aborts

| $T_1$ | $T_2$ |
|-------|-------|
| Write(x) | |
| | Read(x) |
| | Write(y) |
| abort | |
| | => abort |
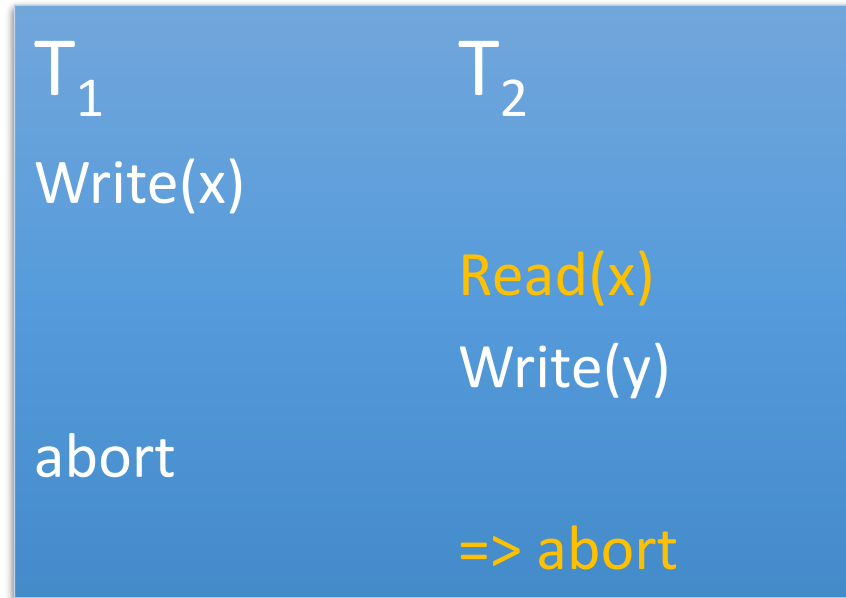
- a schedule in which a transaction T is reading only changes of committed transactions is said to <u>avoid cascading aborts</u>
- avoiding cascading aborts => recoverable schedules

Lock-Based Concurrency Control
- technique used to guarantee serializable, recoverable schedules
- *lock*
  - a tool used by the transaction manager to control concurrent access to data
  - prevents a transaction from accessing a data object while another transaction is accessing the object
- *transaction protocol*
  - a set of rules enforced by the transaction manager and obeyed by all transactions
  - example – simple protocol: before a transaction can read / write an object, it must acquire an appropriate lock on the object
  - locks in conjunction with transaction protocols allow interleaved executions

Lock-Based Concurrency Control
- _SLock_ (_shared_ or _read lock_)
    - if a transaction holds an SLock on an object, it can read the object, but it cannot modify it
- _XLock_ (_exclusive_ or _write lock_)
    - if a transaction holds an XLock on an object, it can both read and write the object
- if a transaction holds an SLock on an object, other transactions can be granted SLocks on the object, but they cannot acquire XLocks on it
- if a transaction holds an XLock on an object, other transactions cannot be granted either SLocks or XLocks on the object

|  | Shared | Exclusive |
| --- | --- | --- |
| Shared | Yes | No |
| Exclusive | No | No |

Lock-Based Concurrency Control
- *lock table*
  - structure used by the lock manager to keep track of granted locks / lock requests
  - entry in the lock table (corresponding to one data object):
    - number of transactions holding a lock on the data object
    - lock type (SLock / XLock)
    - pointer to a queue of lock requests

Lock-Based Concurrency Control
- *transactions table*
    - structure maintained by the DBMS
    - one entry / transaction
    - keeps a list of locks held by every transaction

# References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson, 2009
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html