## Concurrency Control
## Ensuring Isolation

## Concurrency control

### Concurrency

To increase throughput and response time, a DBMS will execute multiple transactions at the same time.

**Concurrency control ensures that transactions have the same effect as if they were executed in isolation**

## Concurrency control

### Problem: WR conflict

| $T_1$ | $T_2$ |
|---|---|
| `READ(A,s)` | |
| `s -= 100` | |
| `WRITE(A,s)` | |
| | `READ(A,t)` |
| | `t *= 1.06` |
| | `WRITE(A,t)` |
| | `READ(B,t)` |
| | `t *= 1.06` |
| | `WRITE(B,t)` |
| `READ(B,s)` | |
| `s += 100` | |
| `WRITE(B,s)` | |

## Concurrency control

### Problem: WW conflict

| $T_1$ | $T_2$ |
|---|---|
| `s = 100` | |
| `WRITE(A,s)` | |
| | `t = 200` |
| | `WRITE(A,t)` |
| | `t = 200` |
| | `WRITE(B,t)` |
| `s = 100` | |
| `WRITE(B,s)` | |

## Concurrency control

### Definitions

- An action is an expression of the form $r(X)$ or $w(X)$
- A transaction is a sequence of actions.

  $r(A), r(B), w(A), w(B)$

  We abstract away from the actual values read or written.
- A schedule is a sequence of actions belonging to multiple transactions. Subscripts indicate to which transaction an action belongs.

  $r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$
- A serial schedule is a schedule in which transactions are not executed concurrently. In a serial schedule the actions hence occur grouped per transaction.

  $r_2(A), w_2(A), r_2(B), w_2(B), r_1(A), w_1(A), r_1(B), w_1(B)$

## Concurrency control

### Serializability

A schedule is called serializable if there exists an equivalent serial schedule.

### Example

The following schedules are equivalent:

$$\begin{aligned}S_1 :=& r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)\\ S_2 :=& r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B)\end{aligned}$$

Hence $S_1$ is serializable.

## Concurrency control

### Conflict-serializability

- Two actions in a schedule are in conflict if:
  1. they belong to the same transaction; or
  2. act upon the same element, and one of them is a write.

  $$r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B)$$

- A schedule is conflict-serializable if we can obtain a serial schedule by (repeatedly) swapping non-conflicting actions.

### Example

We can obtain $S_2$ by swapping only non-conflicting actions from $S_1$:

$$\begin{aligned} S_1 &:= r_1(A), w_1(A), r_2(A), w_2(A), r_1(B), w_1(B), r_2(B), w_2(B) \\ S_2 &:= r_1(A), w_1(A), r_1(B), w_1(B), r_2(A), w_2(A), r_2(B), w_2(B) \end{aligned}$$

Consequently $S_1$ is conflict-serializable.

## Concurrency control

**Clearly, conflict-serializability implies serializability**

**The converse is not true**

$S_1$ is equivalent to $S_2$, but $S_2$ cannot be obtained from $S_1$ by conflict-free swapping:

$$S_1 := w_1(Y), w_2(Y), w_2(X), w_1(X), w_3(X)$$
$$S_2 := w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X)$$

Hence $S_1$ is not conflict-serializable, but it is serializable.

**In practice, a DBMS will only allow conflict-serializable schedules**

## Concurrency control

### A simple algorithm to check conflict-serializability

- Construct the precedence graph
- Check whether this graphs contains cycles. If so, output “no”, otherwise output “yes”

### Example

$$S_1 := r_2(A), r_1(B), w_2(A), r_3(A), w_1(B), w_3(A), r_2(B), w_2(B)$$

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3
\end{tikzcd}$$

$$S_2 := w_1(Y), w_2(Y), w_2(X), w_1(X), w_3(X)$$

$$\begin{tikzcd}
1 \arrow[r] \arrow[rr, bend right] & 2 \arrow[r] \arrow[l, bend right] & 3
\end{tikzcd}$$

## Concurrency control

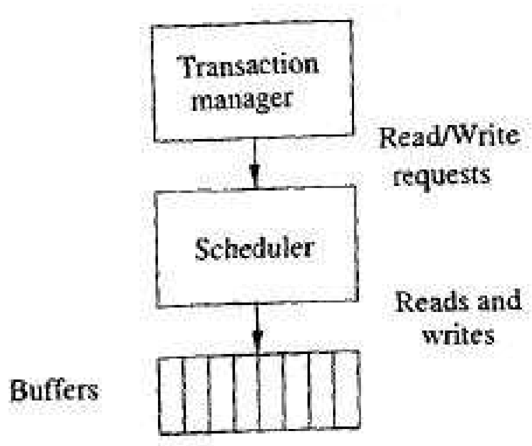
### Why does this work?

- If there exists a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ in the dependency graph then we there are actions from $T_1$ that (1) follow actions from $T_n$ and (2) cannot be moved before the start of $T_n$ by means of conflict-free swapping. Conversely, there are also actions of $T_n$ that follow actions of $T_1$ and that cannot be moved before $T_{n-1}$ by means of conflict-free swapping. As a consequence, we can never obtain a serial schedule by means of conflict-free swapping (in a serial schedule all actions of $T_1$ must occur together).
- If there is no cycle in the dependency graph then we can obtain an equivalent serial schedule by topologically sorting the dependency graph. Illustration on the blackboard.
- See Section 18.2.3 in the book

## Concurrency control

### The scheduler in a DBMS

- It is the taks of the scheduler in a DBMS to create, given a number of transactions, a (conflict-)serializable schedule to be executed.
- New transactions arrive continuously, however, and the scheduler never fully knows the transactions (e.g., because the transactions are large and require a lot of time to run)
- The scheduler hence needs to construct its schedule dynamically, by allowing certain read and write requests; blocking others; and restarting transactions when necessary



## Concurrency control

**Multiple kinds of schedulers:**

- Based on locking
- Based on timestamping
- Based on validation

## Concurrency control

### Lock-based schedulers

- Add actions of the form $l(X)$ and $u(X)$ to schedules.
- Before an item can be read or written, a transaction must have a lock.
- If transaction $i$ requests a lock that is already taken by another transaction $j$, the scheduler will pause the execution of $i$ until $j$ releases the lock. It is in particular impossible for two transaction to possess a lock on the same item at the same time.

## Concurrency control

**Example:**

| $T_1$ | $T_2$ |
|---|---|
| $l_1(A), r_1(A)$ | |
| $w_1(A), l_1(B)$ | |
| $u_1(A)$ | |
| | $l_2(A), r_2(A)$ |
| | $w_2(A)$ |
| | $l_2(B)$ denied |
| $r_1(B), w_1(B)$ | |
| $u_1(B)$ | |
| | $l_2(B), u_2(A)$ |
| | $r_2(B), w_2(B)$ |
| | $u_2(B)$ |

## Concurrency control

**Example:**

$$l_1(A), r_1(A), w_1(A), u_1(A), l_2(A), r_2(A), w_2(A), u_2(A),$$
$$l_2(B), r_2(B), w_2(B), u_2(B), l_1(B), r_1(B), w_1(B), u_1(B)$$

Question: is this conflict-serializable?

## Concurrency control

### Two-phase locking

In order to always obtain a conflict-serializable schedule using locks, we require that in each transaction all lock requests precede all unlock requests.

### Why is this sufficient to guarantee conflict-serializability?

Illustration on the blackboard. See Section 18.3.3 in book.

## Concurrency control

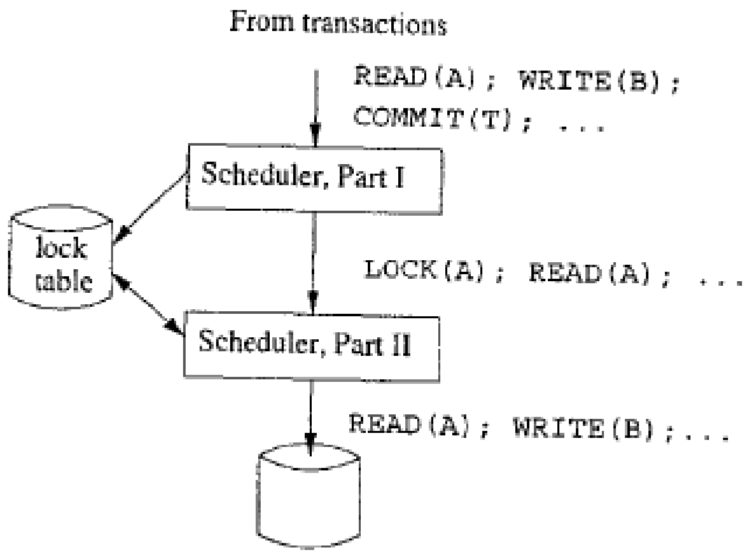
**Observe:**

- It is harmless for multiple transactions to read the same item at the same time.

  → shared and exclusive locks. See Section 18.4 in book.

- In practice transactions will only make read and write requests. They do not make lock and unlock requests. It is the task of the scheduler to add the latter to the schedule

  → see Section 18.5 in book



## Concurrency control

### Schedulers based on timestamping

- Are optimistic schedulers
- Assume that we execute transactions $T_1, T_2$, and $T_3$ where $T_1$ was started first, $T_2$ second, and $T_3$ third. A timestamping scheduler allows arbitrary reorderings of actions from these transactions, but checks at appropriate times if the reordering used are equivalent to the serial schedule $T_1, T_2, T_3$. If not, certain transactions are aborted and restarted.

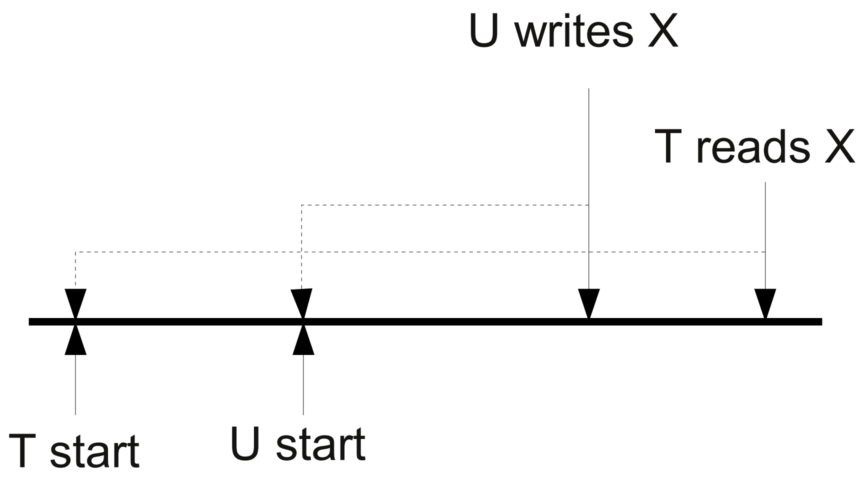
## Concurrency control

### How does it work?

- Every transaction $T$ receives a timestamp $\mathrm{TS}(T)$ upon creation. This can just be a counter that is incremented for each new transaction.
- To each item $X$ we associate two timestamps $\mathrm{RT}(X)$ and $\mathrm{WT}(X)$, and a boolean $\mathrm{C}(X)$.
  - ◦ $\mathrm{RT}(X)$ is the highest timestamp of a transaction that has read $X$
  - ◦ $\mathrm{WT}(X)$ is the highest timestamp of a transaction that has written $X$
  - ◦ $\mathrm{C}(X)$ is true if, and only if, the most recent transaction to write $X$ has already committed.

## Concurrency control

### Unrealizable behavior that we want to avoid (1/4)

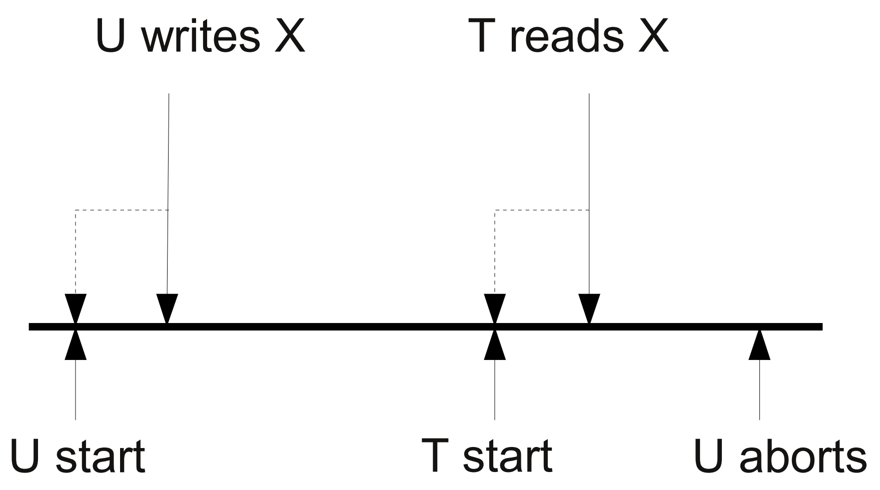


**Hence**

A read request $r_T(X)$ should only be granted if $\mathrm{TS}(T) \geq \mathrm{WT}(X)$.

# Concurrency control

## Unrealizable behavior that we want to avoid (2/4)



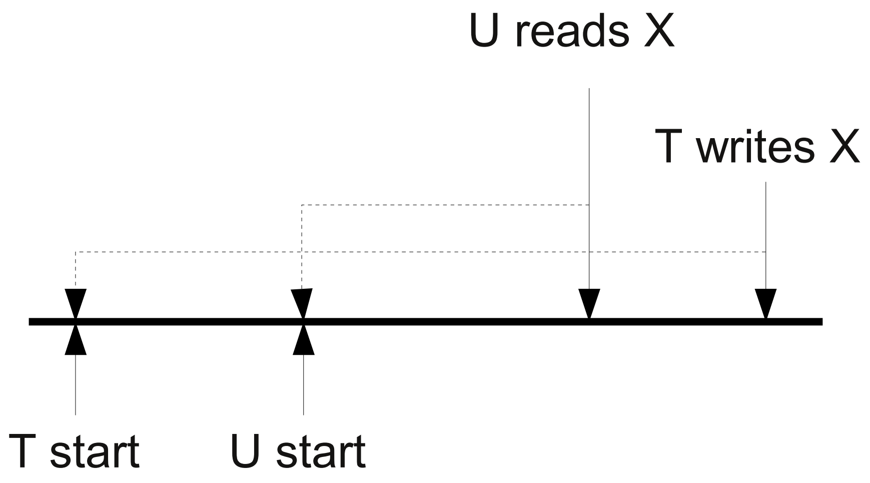
### Hence

Read to $X$ should be delayed until the transaction with timestamp $\mathrm{RT}(X)$ commits (i.e., $\mathrm{C}(X)$ becomes true).

## Concurrency control

### Unrealizable behavior that we want to avoid (3/4)

Suppose $\mathrm{TS}(U) \geq \mathrm{WT}(X)$ at the time when $U$ requests $r_U(X)$.

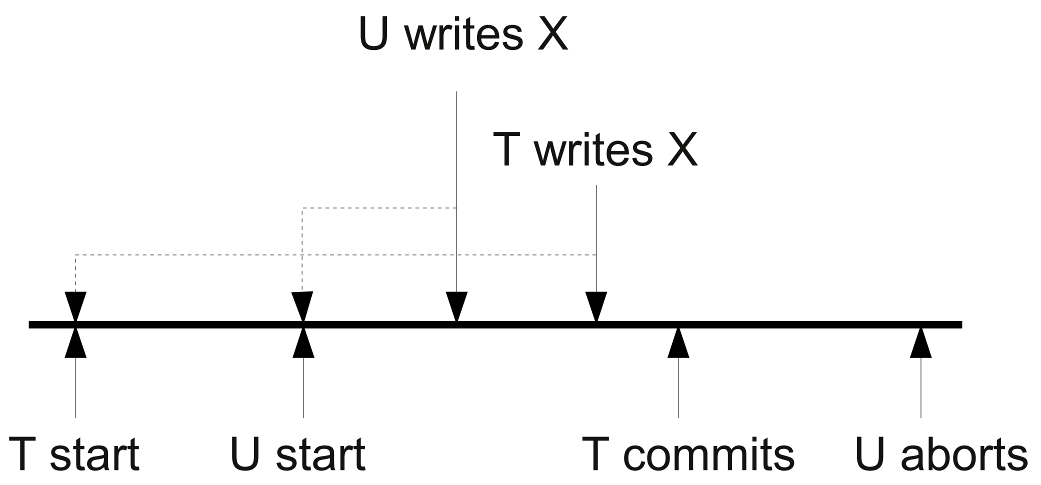


**Hence**

A write request $w_T(X)$ should only be granted if $\mathrm{TS}(T) \geq \mathrm{RT}(X)$

## Concurrency control

### Unrealizable behavior that we want to avoid (4/4)



**Hence**

Request $w_T(X)$ is realizable if $\mathrm{TS}(T) \geq \mathrm{RT}(X)$ and $\mathrm{TS}(T) < \mathrm{WT}(X)$ **BUT**:

- if $\mathrm{C}(X)$ is false then $T$ must be delayed until the transaction with timestamp $\mathrm{WT}(X)$ commits (i.e. $\mathrm{C}(X)$ becomes true)
- if $\mathrm{C}(X)$ is true then the write can be ignored

## Concurrency control

### How does it work: conclusion

- Every transaction receives a timestamp upon creation. This can just be a counter that is incremented for each new transaction.
- To each item $X$ we associate two timestamps $\mathrm{RT}(X)$ and $\mathrm{WT}(X)$, and a boolean $\mathrm{C}(X)$.
- A transaction with timestamp $t$ is allowed to read item $X$ if $t \geq \mathrm{WT}(X)$. If $\mathrm{C}(X)$ is false then the execution is paused until $\mathrm{C}(X)$ becomes true or the transaction that has last written $X$ aborts. If $t < \mathrm{WT}(X)$ then the transaction is aborted and restarted with a larger timestamp.
- A transaction with timestamp $t$ is allowed to write item $X$ if $\mathrm{RT}(X) \leq t$ and $\mathrm{WT}(X) \leq t$. If $t < \mathrm{RT}(X)$ then the transaction is aborted and restarted with a larger timestamp. If $\mathrm{RT}(X) \leq t < \mathrm{WT}(X)$ and $\mathrm{C}(X)$ is true then we keep the current value of $X$. Otherwise the execution is paused until $\mathrm{C}(X)$ becomes true, or until the transaction that last wrote $X$ aborts.

## Concurrency control

### Locking versus timestamping

- Locking is very efficient when we have many transactions that both read and write. In that case, timestamping will need to abort and restart many transactions.
- Timestamping is very efficient when we have many transactions that make only read requests. In that case, many transactions would have to wait for locks when using a lock-based scheduler, while they can immediately proceed with timestamping-based schedulers.

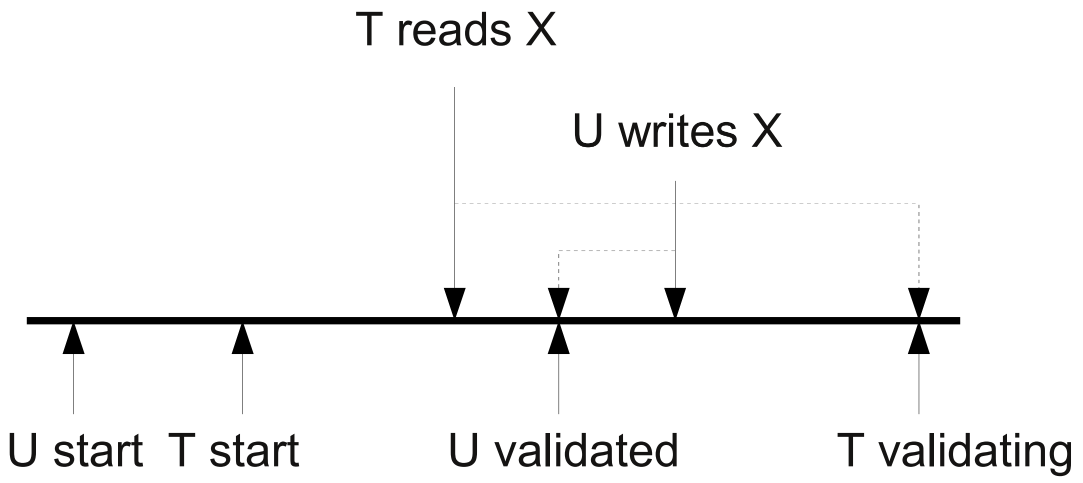
## Concurrency control

### Schedulers based on validation

- Are optimistic
- The scheduler records, for every transaction $T$, the set $\mathrm{RS}(T)$ of items read by $T$, and the set $\mathrm{WS}(T)$ of items written by $T$.
- Transactions are executed in three phases. In the first phase a transaction reads all items in $\mathrm{RS}(T)$. In the second phase, the scheduler validates the transaction based on $\mathrm{RS}(T)$ and $\mathrm{WS}(T)$. If validation fails, the transaction is aborted and restarted. In the third phase the transaction writes all items in $\mathrm{WS}(T)$.
- The goal is again to obtain a schedule that is equivalent with the serial transaction schedule that orders transactions by their starting time.

## Concurrency control

### Unrealizable behavior that we want to avoid (1/2)

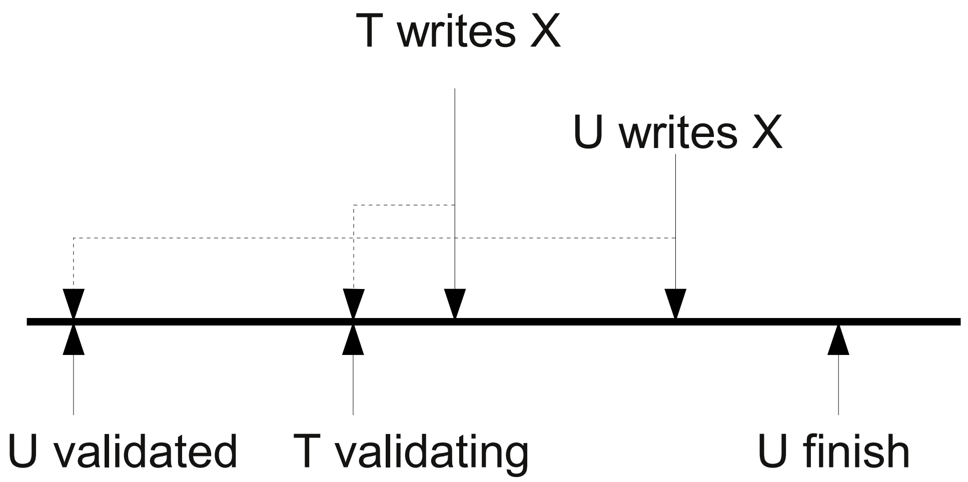


**Hence**

- Record, for every transaction $V$, the time $\mathrm{START}(V)$, $\mathrm{VAL}(V)$, and $\mathrm{FIN}(V)$ at which $V$ starts, validates, and finishes, respectively.
- $T$ can only successfully validate if $\mathrm{RS}(T) \cap \mathrm{WS}(U) = \emptyset$ for any previously validated transaction $U$ that was not yet finished when $T$ started, i.e., $\mathrm{FIN}(U) > \mathrm{START}(T)$.

## Concurrency control

### Unrealizable behavior that we want to avoid (2/2)



**Hence**

$T$ can only successfully validate if $\mathrm{WS}(T) \cap \mathrm{WS}(U) = \emptyset$ for every previously validated $U$ that did not finish before $T$ validated, i.e., $\mathrm{FIN}(U) > \mathrm{VAL}(T)$.

## Concurrency control

### How does the scheduler validate?

A transaction $T$ passes validation if:

1. $\mathrm{RS}(T) \cap \mathrm{WS}(U) = \emptyset$ for every transaction $U$ that has already been validated, but was not finished when $T$ started.
2. $\mathrm{WS}(T) \cap \mathrm{WS}(U) = \emptyset$ for every transaction $U$ that has already been validated, but is currently not yet finished.

If $T$ does not pass validation, it is aborted and restarted.