

# Database Management Systems

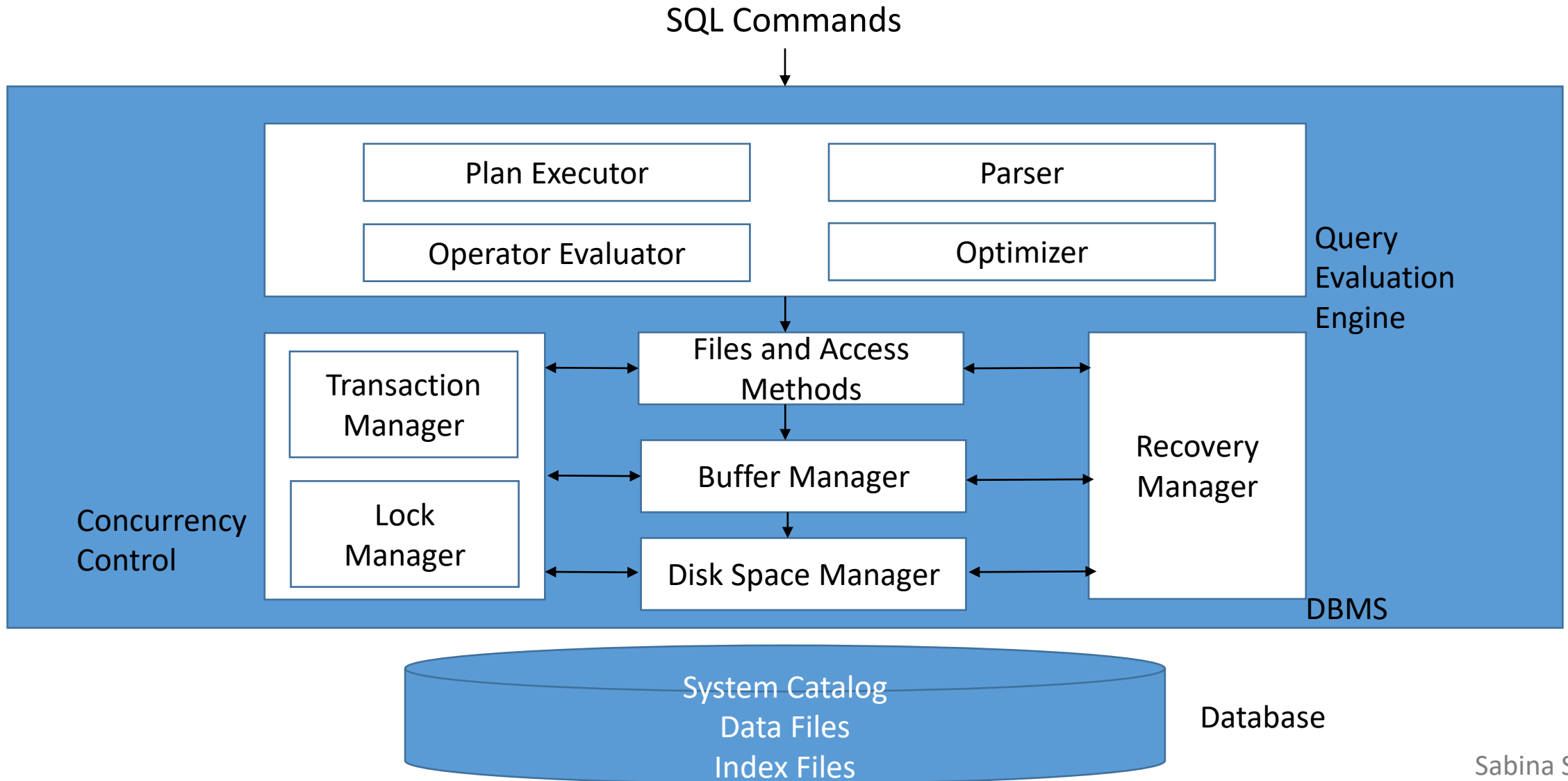
Lecture 1

Introduction - Transactions

# DBMSs

- Lecture2 + Seminar1 + Laboratory1
- grading
  - written exam (W) - 50%
  - practical test (P) - 25%
  - labs (L) - 25%
- to attend the exam, a student must have at least 6 laboratory attendances and at least 5 seminar attendances, according to the Computer Science Department's decision:
  - <https://www.cs.ubbcluj.ro/wp-content/uploads/Hotarare-CDI-29.04.2020.pdf>
- <https://www.cs.ubbcluj.ro/~sabina>
- sabina@cs.ubbcluj.ro

# DBMS Architecture



# Concurrency in a DBMS

- scenarios

1. airline reservation system

- travel agent 1 (TA1) processes customer 1's (C1) request for a seat on flight 10
  - system retrieves from the DB tuple *<flight: 10, available seats: 1>* and displays it to TA1
  - C1 takes time to decide whether or not to make the reservation
- travel agent 2 (TA2) processes customer 2's (C2) request for a seat on flight 10
  - system displays the same tuple to TA2: *<flight: 10, available seats: 1>*
  - C2 immediately reserves the seat:
    - system updates the tuple retrieved by TA2: *<flight: 10, available seats: 0>*

# Concurrency in a DBMS

- scenarios

1. airline reservation system

- system updates the database, i.e., replaces flight 10's tuple in the DB with the new tuple *<flight: 10, available seats: 0>*
  - C1 also decides to reserve the seat
    - however, TA1 is still working on the previously retrieved tuple, not knowing another user has changed the data
    - system updates the tuple retrieved by TA1: *<flight: 10, available seats: 0>*
    - system updates the database, i.e., replaces flight 10's tuple in the DB with the new tuple *<flight: 10, available seats: 0>*
- => the last remaining seat on flight 10 has been sold 2 times!

# Concurrency in a DBMS

- scenarios

## 2. bank system

- 1000 accounts:  $A_1, A_2, \dots, A_{1000}$
- program P1 is computing the total balance:  $\sum_{i=1}^{1000} \text{Balance}(A_i)$
- program P2 transfers 500 lei from  $A_1$  to  $A_{1000}$
- if P2's transfer occurs after P1 has “seen”  $A_1$ , but before it has “seen”  $A_{1000}$   
=> the total balance will be incorrect:  $\sum_{i=1}^{1000} \text{Balance}(A_i) + 500$
- objective: prevent such anomalies without disallowing concurrent access!

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?

- consider again scenario 1

- regard the activity of  $TA_i$  as invoking a copy of the program below:

```
S ← seatsAvailable(flight 10)
```

```
if (S > 0) then
```

```
    if (customer decides to purchase ticket)
```

```
        S--
```

```
        seatsAvailable(flight 10) ← S
```

```
    endif
```

```
else
```

```
    tell customer there are no more seats on flight 10
```

```
endif
```

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
    - if the scheduling policy doesn't allow interleaved execution, TA2's copy of the previous program can begin execution only after TA1's invocation of the program has completed, i.e., after C1 has made the decision and the DB has been updated
- => if C1 decided to buy the ticket, TA2 sees tuple *<flight: 10, available seats: 0>* => C2 will be correctly informed that flight 10 is sold out



# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
- disallowing concurrent execution can lead to unacceptable delays
  - e.g., while C1 decides whether to make the reservation, travel agent 3 (TA3) receives a request from customer 3 (C3) for a seat on flight 11
  - since C1 and C3 are interested in different flights, their requests can be concurrently processed
  - if concurrent access is not allowed, C3's request can be processed only after C1 has made the decision and TA1's invocation of the program has finished

# Concurrency in a DBMS

- what happens when concurrent access is forbidden?
- obs. when programs are executed serially (i.e., one after the other), results are correct
- however, the DBMS interleaves the operations (reads / writes of database objects) of different user programs, as concurrent execution is fundamental for the performance of the application
- the disk is frequently accessed (and access times are relatively slow) => overlap I/O and CPU activity
- if concurrent execution is allowed, one needs to tackle the effects of interleaving user programs

# Transactions

- database partitioned into items (e.g., *tuple*, *attribute*)
- transaction
  - any *one execution* of a program in a DBMS
  - sequence of one or more operations on a database: *read* / *write* / *commit* / *abort*
  - final operation:
    - commit or abort

# Transactions

- $I$  – an item in a DB
- fundamental operations in a transaction  $T$  ( $T$  can be omitted when clear from context):
  - $\text{read}(T, I)$
  - $\text{write}(T, I)$
  - $\text{commit}(T)$ 
    - successful completion:  $T$ 's changes are to become permanent
  - $\text{abort}(T)$ 
    - $T$  is *rolled back*: its changes are undone

# Transaction Properties – ACID

- properties a DBMS must ensure for a transaction in order to maintain data in the presence of concurrent execution / system failures
- atomicity
  - a transaction is atomic
  - either all the operations in the transaction are executed, or none are (*all-or-nothing*)
- consistency
  - a transaction must preserve the consistency of the database after execution, i.e., a transaction is a correct program
- isolation
  - a transaction is protected from the effects of concurrently scheduling other transactions

# Transaction Properties – ACID

- durability
  - the effects of a successfully completed transaction should persist even if a system crash occurs before all the changes have been written to disk

# Transaction Properties – Atomicity

- a transaction can commit / abort:
  - *commit* – after finalizing all its operations (successful completion)
  - *abort* – after executing some of its operations; the DBMS can itself abort a transaction
- user's perspective:
  - either all operations in a transaction are executed, or none are
  - the transaction is treated as an indivisible unit of execution
- the DBMS *logs* all transactions' writes, so it can *undo* them if necessary (i.e., undo the actions of incomplete transactions)
- *crash recovery*
  - guarantees atomicity in the presence of system crashes

# Transaction Properties – Atomicity

- e.g., consider the following transaction transferring money from AccountA to AccountB:

AccountA  $\leftarrow$  AccountA - 100

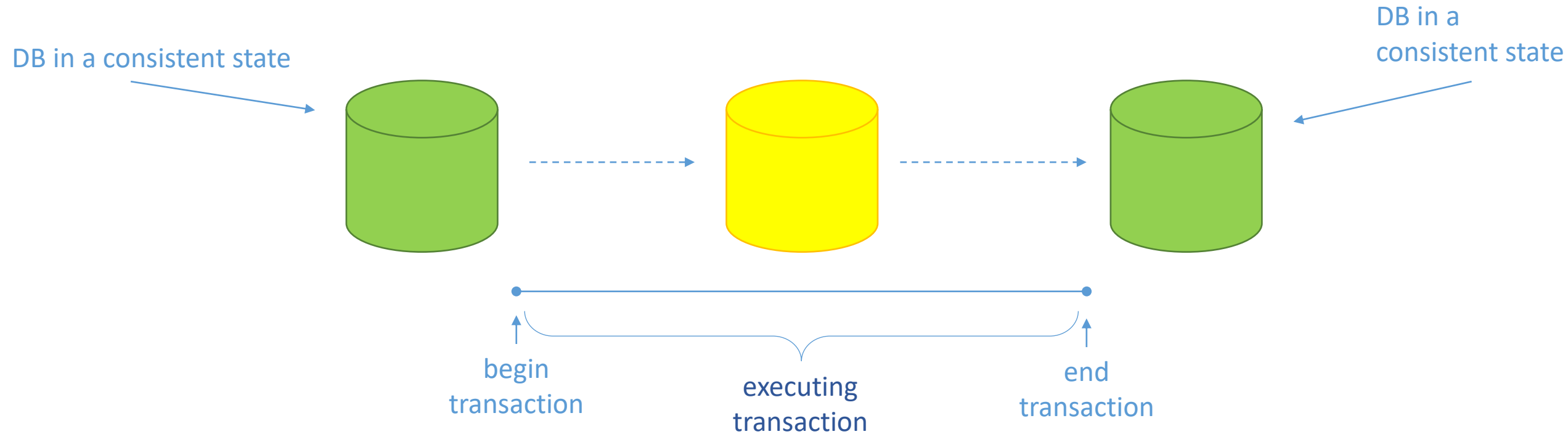
AccountB  $\leftarrow$  AccountB + 100

- if the transaction fails in the middle, 100 lei must be restored to AccountA, i.e., partial effects are undone



# Transaction Properties – Consistency

- in the absence of other transactions, a transaction that executes on a consistent database state leaves the database in a consistent state



# Transaction Properties – Consistency

- transactions do not violate the integrity constraints (ICs) specified on the database (e.g., restrictions declared via CREATE TABLE statements) and enforced by the DBMS
- transactions are correct programs
- e.g.: consider again the money transfer transaction:

AccountA  $\leftarrow$  AccountA - 100

AccountB  $\leftarrow$  AccountB + 100

- assume the transaction starts execution at time  $t_i$  and completes at  $t_j$
- consistency constraint:  $\text{Balance}(\text{AccountA}) + \text{Balance}(\text{AccountB}) \text{ at } t_i = \text{Balance}(\text{AccountA}) + \text{Balance}(\text{AccountB}) \text{ at } t_j$
- it's the responsibility of the user to write a correct transaction that meets the above constraint

# Transaction Properties – Consistency

- the DBMS does not understand the semantics of the data, of user actions (e.g., how is the interest computed, the fact that AccountB must be credited with the amount of money debited from AccountA, etc.)

# Transaction Properties – Isolation

- transactions are protected from the effects of concurrently scheduling other transactions
- users concurrently submit transactions to the DBMS, but can think of their transactions as if they were executing in isolation, independently of other users' transactions
- i.e., users must not deal with arbitrary effects produced by other concurrently running transactions; a transaction is seen as if it were in single-user mode

# Transaction Properties – Durability

- the system must guarantee the effects of a successfully completed transaction won't be lost, even if failures subsequently occur
- i.e., once the DBMS informs the user a transaction has completed successfully, its effects should persist, even if a system crash occurs before all changes have been saved on the disk
  - the *Write-Ahead Log* property
    - changes are written to the log (on the disk) before being reflected in the database
- the log ensures atomicity, durability

# Interleaved Executions - Example

- consider the two transactions below:

T1: BEGIN             $A \leftarrow A - 100$              $B \leftarrow B + 100$             END

T2: BEGIN             $A \leftarrow 1.05 * A$              $B \leftarrow 1.05 * B$             END

- T1 is transferring 100 lei from account A to account B
- T2 is adding 5% interest to both accounts
- T1 and T2 are submitted together to the DBMS
- serial execution: T1 can execute before or after T2 on a database DB, resulting in one of the following database states:
  - State( (T1T2), DB) or
  - State( (T2T1), DB)
- interleaved execution: the net effect of executing T1 and T2 *must* be identical to State( (T1 T2), DB) or State( (T2 T1), DB)

# Interleaved Executions - Example

- consider the following interleaving of operations (schedule):

T1	T2
$A \leftarrow A - 100$	
	$A \leftarrow 1.05 * A$
$B \leftarrow B + 100$	
	$B \leftarrow 1.05 * B$



# Interleaved Executions - Example

- but what about the following schedule:





# Interleaved Executions - Example

- the DBMS's view of the second schedule:

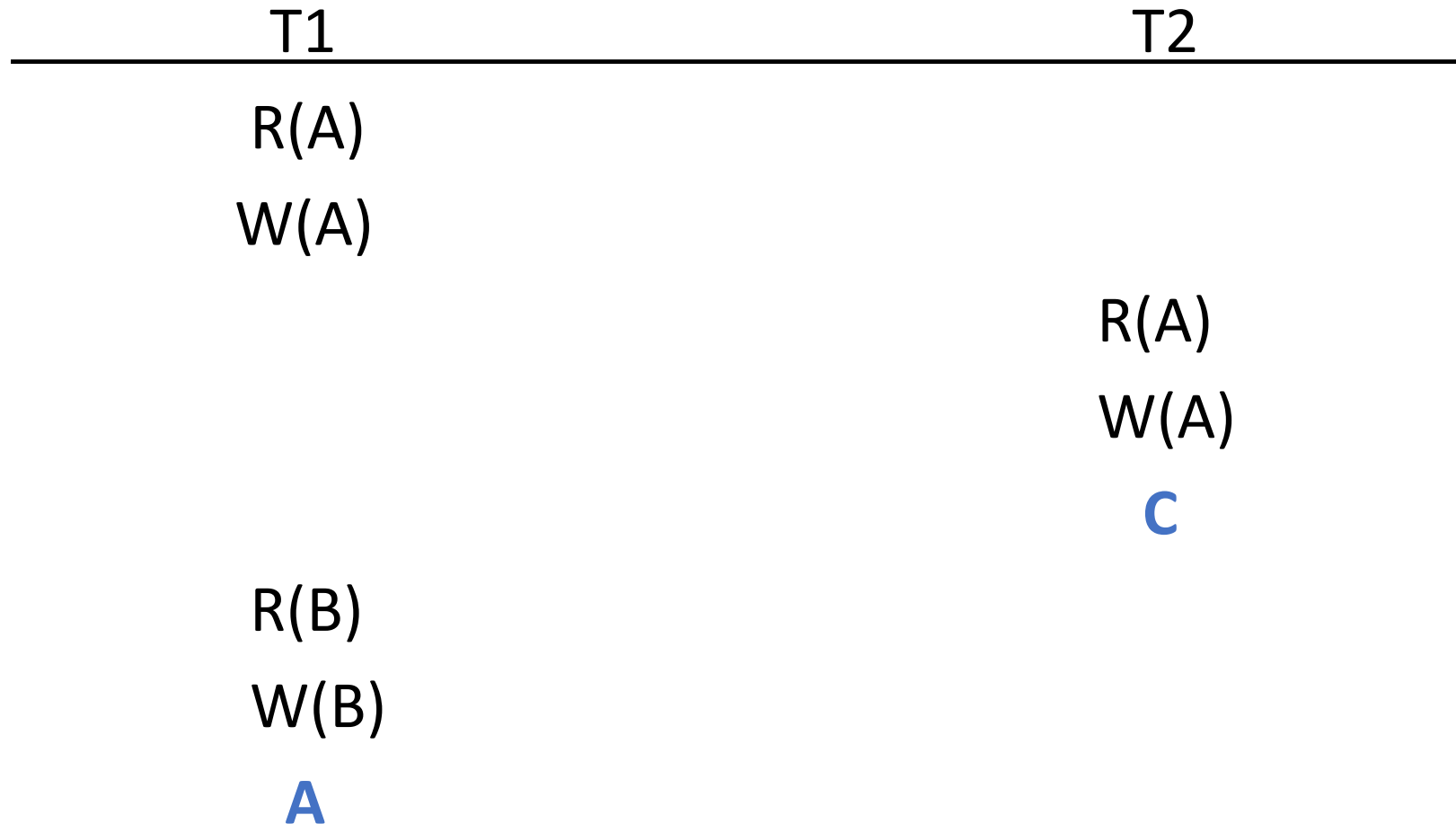
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

# Interleaved Executions - Anomalies

- two transactions are only reading a data object => no conflict, order of execution not important
- two transactions are reading and / or writing completely separate data objects => no conflict, order of execution not important
- two transactions are operating on the same data object, and at least of them performs a write operation => order of execution is important
  - WR conflict
    - T2 is reading a data object previously written by T1
  - RW conflict
    - T2 is writing a data object previously read by T1
  - WW conflict
    - T2 is writing a data object previously written by T1

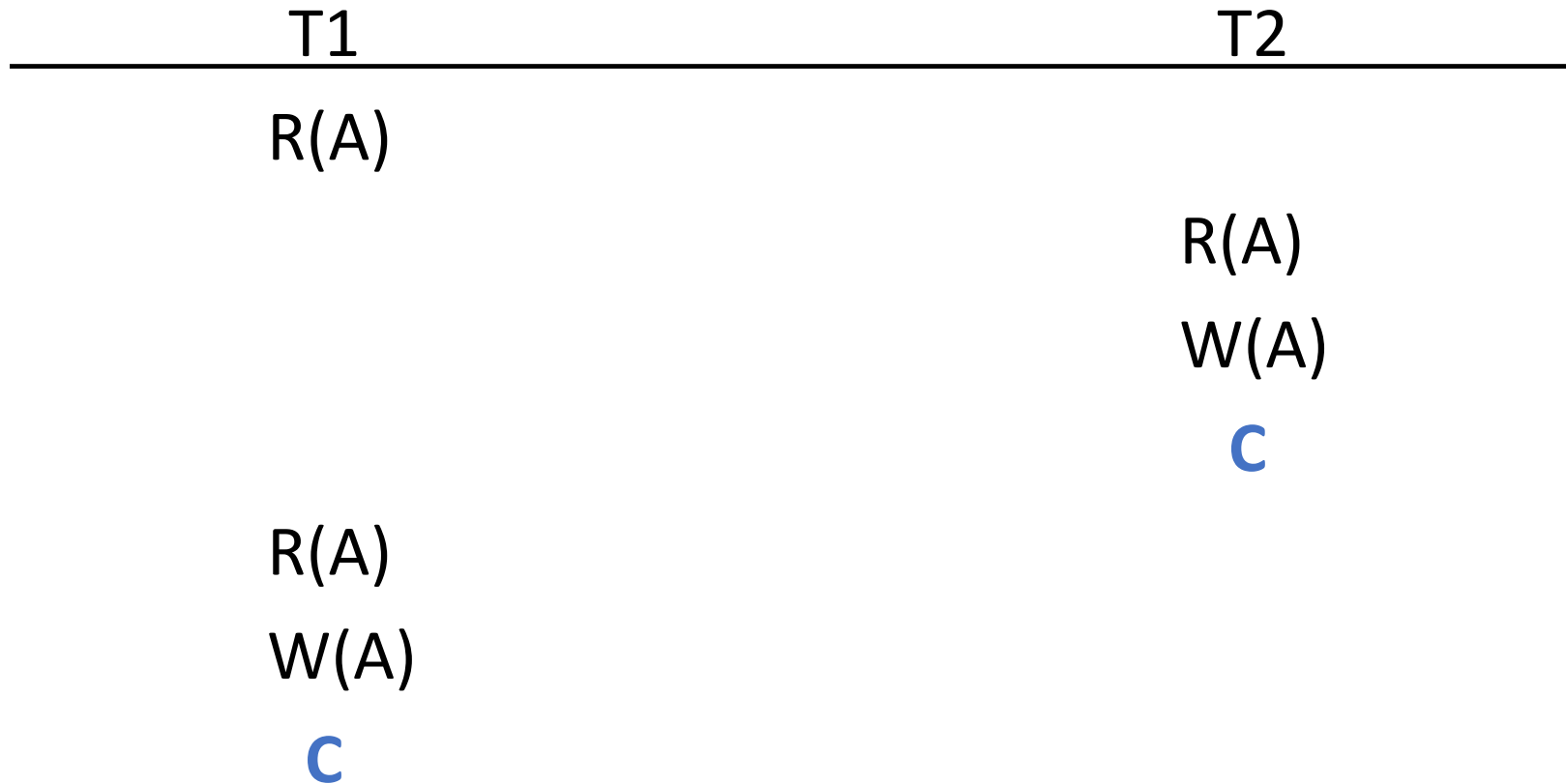
# Interleaved Executions - Anomalies

- reading uncommitted data (WR conflicts, *dirty reads*)



# Interleaved Executions - Anomalies

- unrepeatable reads (RW conflicts)



# Interleaved Executions - Anomalies

- overwriting uncommitted data (WW conflicts):



# Scheduling Transactions

- *schedule*

- a list of operations (Read / Write / Commit / Abort) of a set of transactions with the property that the order of the operations in each individual transaction is preserved

# Scheduling Transactions

T1

read(V)  
read(sum)

read(V)  
sum := sum + V  
write(sum)  
commit

T2

read(V)  
V := V + 50  
write(V)  
commit

Schedule

read1(V)  
read1(sum)  
read2(V)

write2(V)  
commit2  
read1(V)

write1(sum)  
commit1

# Serial and Non-Serial Schedules

- serial schedule

- a schedule in which the actions of different transactions are not interleaved

T1

```
read(V)
read(sum)
read(V)
sum := sum + V
write(sum)
commit
```

T2

```
read(V)
V := V + 50
write(V)
commit
```



# Serial and Non-Serial Schedules

- *non-serial schedule*

- a schedule in which the actions of different transactions are interleaved

read1(V)

read1(sum)

read2(V)

write2(V)

commit2

read1(V)

write1(sum)

commit1

# Serializability

- $C$  – set of transactions
- $Sch(C)$  – the set of schedules for  $C$
- serializable schedule
  - a schedule  $S \in Sch(C)$  is serializable if the effect of  $S$  on a consistent database instance is identical to the effect of some serial schedule  $S_0 \in Sch(C)$

# Serializability

- serializability - correctness criterion for an interleaved execution schedule
  - consider the serial execution schedule  $(T_1, T_2, \dots, T_n)$ ,  $T_i \in C$
  - assume the database instance is in a correct state prior to executing  $T_1$
  - every transaction must preserve the consistency of the database
    - $\Rightarrow$  the database is in a correct state after  $T_n$  completes execution
    - $\Rightarrow$  if a serializable schedule is executed on a correct database instance, it produces a correct database instance (since it is equivalent to some serial schedule)
- ensuring serializability prevents inconsistencies generated by concurrent transactions that interfere with one another
- objective - finding interleaved schedules enabling transactions to execute concurrently without interfering with each other (such schedules result in a correct database state)

# 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 Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3<sup>rd</sup> Edition,  
<http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html>
- [Si11] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (6th Edition), McGraw-Hill, 2011
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, <http://codex.cs.yale.edu/avi/db-book/>
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems,  
<http://infolab.stanford.edu/~ullman/fcdb.html>