Database Management Systems

Lecture 1

Introduction - Transactions

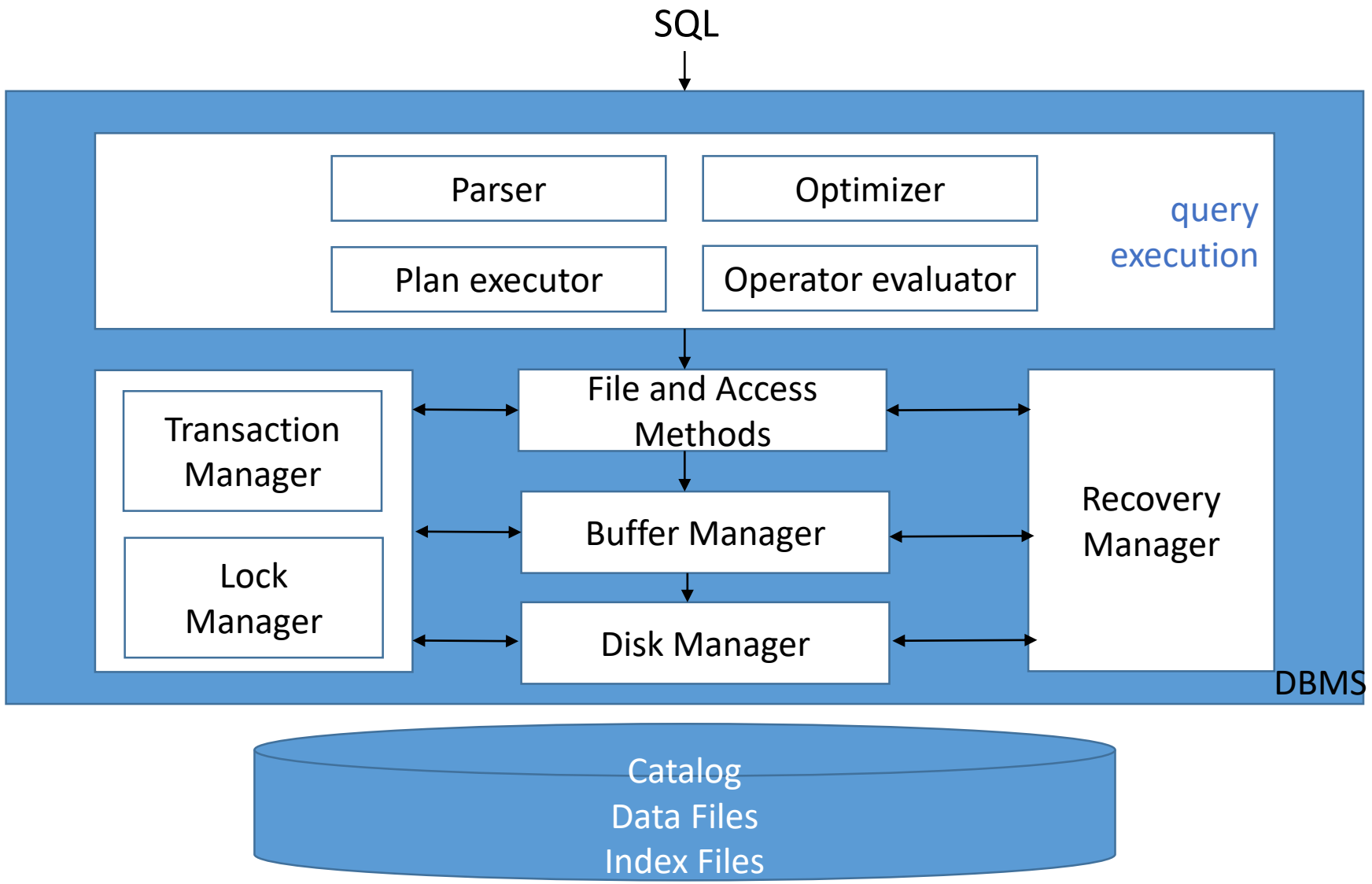
DBMSs

- Lecture2 + Seminar1 + Laboratory1
- grading
  - written exam (W) - 50%
  - practical exam (P) - 25%
  - labs (L) - 25%
  - W, P >= 5
- 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:
  - http://www.cs.ubbcluj.ro/wp-content/uploads/Hotarare-CDI-15.03.2017.pdf
- http://www.cs.ubbcluj.ro/~sabina
- sabina@cs.ubbcluj.ro

DBMSs

- lab
  - DBMS: SQL Server
  - application development: .NET / C#
  - data access: ADO.NET

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>*
- 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, ..., A_{1000}
- program P1 is computing the total balance: $\sum_{i=1}^{1000} Balance(A_i)$
- 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: $\sum_{i=1}^{1000} Balance(A_i)$ + 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 transactions

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*
  - *commit* occurs if and only if the transaction is not aborted
  - 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):
  - read(T, I)
  - write(T, I)
  - commit(T)
    - successful completion: T's changes are to become permanent
  - abort(T)
    - T is *rollbacked*: 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
  - committed changes are persisted to the database
  - the effects of a committed 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' actions, so it can *undo* them if necessary (i.e., undo the actions of incomplete transactions)
- *crash recovery*
  - the process of guaranteeing atomicity in the presence of system crashes

Transaction Properties – Atomicity

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

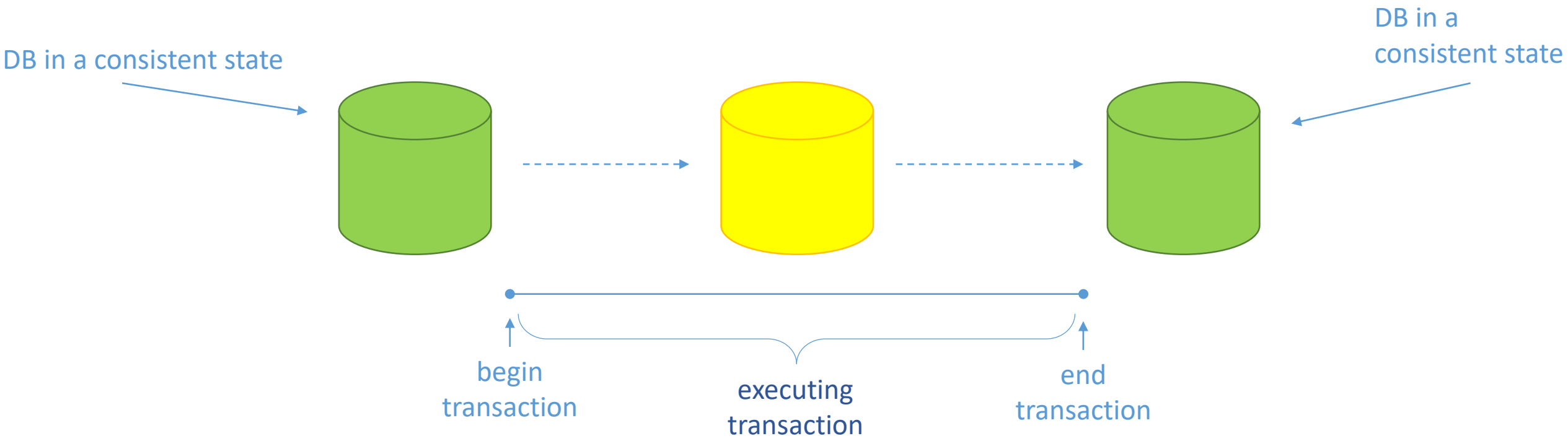
  AccountA ← AccountA - 100

  AccountB ← AccountB + 100

- if the transaction fails in the middle, 100 lei must be put back into 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
- however, users can have consistency criteria that go beyond the expressible ICs
- e.g.: consider again the money transfer transaction:

  AccountA ← AccountA - 100

  AccountB ← AccountB + 100

  - assume the transaction starts execution at time t_i and completes at t_j
  - consistency constraint: Total(AccountA, AccountB, t_i) = Total(AccountA, AccountB, 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

- once a transaction commits, the system must guarantee the effects of its operations won't be lost, even if failures subsequently occur
- i.e., once the DBMS informs the user a transaction has committed, 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
    - ensures atomicity, durability

Interleaved Executions - Example

- consider the two transactions below:

T1: BEGIN	A ← A-100	B ← B+100	END
T2: BEGIN	A ← 1.05*A	B ← 1.05*B	END

- T1 is transferring 100 lei from account A to account B
- T2 is adding a 5% interest to both accounts
- T1 and T2 are submitted together to the DBMS
- 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)
- 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 ← A - 100	
	A ← 1.05 * A
B ← B + 100	
	B ← 1.05 * B

✓

Interleaved Executions - Example

- but what about the following schedule:

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

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
- one transaction is writing a data object and another one is either reading or writing the same object => 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*)

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

Interleaved Executions - Anomalies

- unrepeatable reads (RW conflicts)

T1	T2
R(A)	
	R(A)
	W(A)
	C
R(A)	
W(A)	
C	

Interleaved Executions - Anomalies

- overwriting uncommitted data (WW conflicts):

T1	T2
W(A)	
	W(A)
	W(B)
	C
W(B)	
C	

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

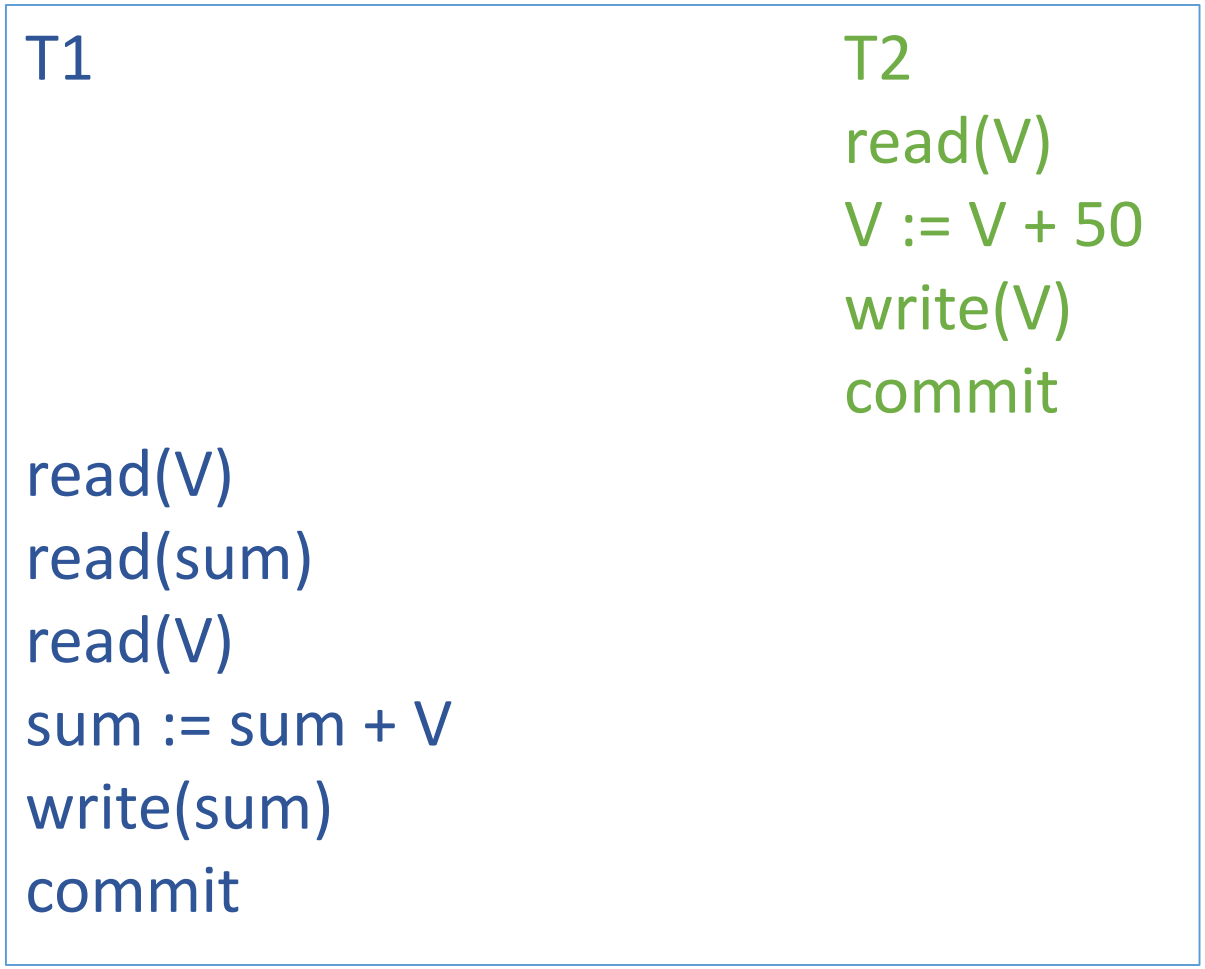
Scheduling Transactions

T1	T2
read(V)	
read(sum)	
	read(V)
	V := V + 50
	write(V)
	commit
read(V)	
sum := sum + V	
write(sum)	
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



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, …, 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

    => the database is in a correct state after T_n completes execution

    => 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

Serializability

- objective
  - finding interleaved schedules enabling transactions to execute concurrently without interfering with each other (such schedules result in a correct database state)

- the order of *read* and *write* operations is important
- actions that cannot be swapped in a schedule:
  - actions belonging to the same transaction
  - actions in different transactions operating on the same object if at least one of them is a *write*

References

- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Le99] LEVENE, M., LOIZOU, G., A Guided Tour of Relational Databases and Beyond, Springer, 1999
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, 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