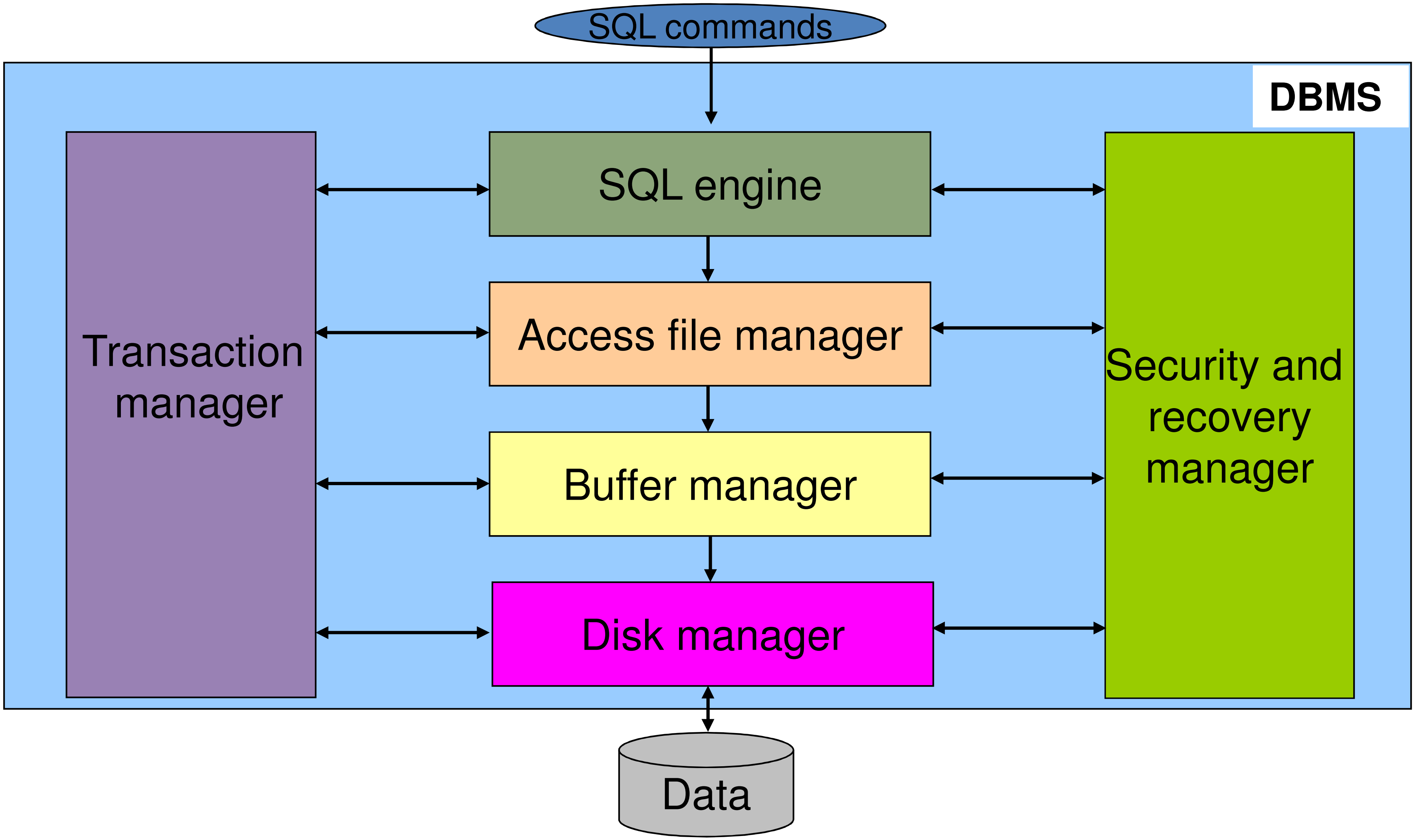
Database Management Systems: transaction management and recovery management

Maurizio Lenzerini, Riccardo Rosati

Dipartimento di Ingegneria informatica automatica e gestionale
Sapienza Università di Roma

2017/2018

Architecture of a DBMS



1 - Transaction management

1.1 Transactions, concurrency, serializability
1.2 Recoverability
1.3 Concurrency control through locks
1.4 Concurrency control through timestamps
1.5 Transaction management in SQL

1 - Transaction management

1.1 Transactions, concurrency, serializability
1.2 Recoverability
1.3 Concurrency control through locks
1.4 Concurrency control through timestamps
1.5 Transaction management in SQL

Transactions

A transaction models the execution of a software procedure constituted by a set of instructions that may ”read from” and “write on” a database, and that form a single logical unit.

Syntactically, we will assume that every transaction contains:

- one “begin” instruction
- one “end” instruction
- one among “commit” (confirm what you have done on the database so far) and “rollback” (undo what you have done on the database so far)

As we will see, each transaction should enjoy a set of properties (called ACID)

Example of "real" transaction

```
begin
  writeln('Inserire importo, conto di partenza, conto di arrivo');
  read (Importo, contoPartenza, contoArrivo);
  EXEC SQL
    select Saldo into :saldoCorrente
    from ContiCorrenti
    where Numero = :contoPartenza
  if saldoCorrente < Importo
  then begin
    writeln('Saldo Insufficiente');
    ABORT;
  end;
  else begin
    EXEC SQL
      UPDATE ContiCorrenti
      set Saldo=:saldoCorrente - :Importo
      where Numero = :contoPartenza;
    writeln('Operazione eseguita con successo');
    COMMIT;
  end;
end;
```

Tabella ContiCorrenti

Numero	Saldo

Effect of a transaction

Let DB be a database

Let T be a transaction on DB

Effect (or result) of T = state of DB after the execution of T

As we shall see, every transaction must enjoy a set of properties (called ACID properties) that deal with the effect of the transaction

Concurrency

The throughput of a system is the number of transactions per second (tps) accepted by the system

In a DBMS, we want the throughput to be approximately 100-1000tps

This means that the system should support a high degree of concurrency among the transactions that are executed

- Example: If each transaction needs 0.1 seconds in the average for its execution, then to get a throughput of 100tps, we must ensure that 10 transactions are executed concurrently in the average

Typical applications: banks, flight reservations, …

Concurrency: example

Suppose that the same program is executed concurrently by two applications aiming at reserving a seat in the same flight

The following temporal evolution is possible:

	Application 1	Application 2
1.	Find seat	
2.		Find seat
3.	Book seat	
4.		Book seat

time ↓

The result is that we have two reservations for the same seat!

Isolation of transactions

The DBMS deals with this problem by ensuring the so-called "isolation" property for the transactions

This property for a transaction essentially means that it is executed like it was the only one in the system, i.e., without concurrent transactions

While isolation is essential, other properties are important as well

Desirable properties of transactions

The desirable properties in transaction management are called the ACID properties. They are:

1. Atomicity: for each transaction execution, either all or none of its actions are executed
2. Consistency: each transaction execution brings the database to a correct state
3. Isolation: each transaction execution is independent of any other concurrent transaction executions
4. Durability: if a transaction execution succeeds, then its effects are registered permanently in the database

Schedules and serial schedules

Given a set of transactions T1,T2,…,Tn, a sequence S of executions of actions of such transactions respecting the order within each transaction (i.e., such that if action a is before action b in Ti, then a is before b also in S) is called schedule on T1,T2,…,Tn, or simply schedule.

A schedule on T1,T2,…,Tn that does not contain all the actions of all transactions T1,T2,…,Tn is called partial

A schedule S is called serial if the actions of each transaction in S come before every action of a different transaction in S, i.e., if in S the actions of different transactions do not interleave.

Serializability

Example of serial schedules:

Given T1 (x=x+x; x= x+2) and T2 (x= x**2; x=x+2), possible serial schedules on them are:

Sequence 1: x=x+x; x= x+2; x= x**2; x=x+2
Sequence 2: x= x**2; x=x+2; x=x+x; x= x+2

Definition of serializable schedule: A schedule S is serializable if the outcome of its execution is the same as the outcome of at least one serial schedule constituted by the same transactions of S, no matter what the initial state of the database is.

Serializability

In other words, a schedule S on T1,T2,…,Tn is serializable if there exists a serial schedule on T1,T2,…,Tn that is "equivalent" to S

But what does "equivalent" mean?

Definition of equivalent schedules: Two schedules S1 and S2 are said to be equivalent if, for each database state D, the execution of S1 starting in the database state D produces the same outcome as the execution of S2 starting in the same database state D

Notation

A successful execution of transaction can be represented as a sequence of

- Comands of type begin/commit
- Actions that read and write an element (attribute, record, table) in the database
- Actions that read and write an element in the local store

T_1	T_2
begin	begin
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)
commit	commit

A serial schedule

T_1	T_2	A	B
		25	25
begin			
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
commit			
	begin		
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

A serializable schedule

T_1	T_2	A	B
		25	25
begin	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
commit			
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

The final values of A and B are the same as the serial schedule T1, T2, no matter what the initial values of A and B.

We can indeed show that, if initially A=B=c (c is a costant), then at the end of the execution of the schedule we have: A=B=2(c+100)

A non-serializable schedule

T_1	T_2	A 25	B 25
begin	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
	commit		
READ(B,t)			
t := t+100			
WRITE(B,t)			150
commit			

Where is the problem??

A non-serializable schedule

T_1	T_2	A	B
		25	25
begin	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150
commit			
	commit		

Where is the problem??

Anomaly 1: reading temporary data (WR anomaly)

T_1	T_2
begin	begin
READ(A,x)	
x := x-1	
WRITE(A,x)	
	READ(A,x)
	x := x*2
	WRITE(A,x)
	READ(B,x)
	x := x*2
	WRITE(B,x)
	commit
READ(B,x)	
x:=x+1	
WRITE(A,x)	
commit	

Note that the interleaved execution is different from any serial execution. The problem comes from the fact that the value of A written by T1 is read by T2 before T1 has completed all its changes.

This is a WR (write-read) anomaly

Anomaly 2a: update loss (RW anomaly)

- Let T_1, T_2 be two transactions, each of the form:

  READ(A, x), x := x + 1, WRITE(A, x)
- The serial execution with initial value A=2 produces A=4, which is the result of two subsequent updates
- Now, consider the following schedule:

T_1	T_2
begin	begin
READ(A,x)	
x := x+1	
	READ(A,x)
	x := x+1
WRITE(A,x)	
commit	
	WRITE(A,x)
	commit

The final result is A=3, and the first update is lost: T2 reads the initial value of A, and writes the final value. In this case, the update executed by T1 is lost!

Anomaly 2a: update loss (RW anomaly)

- This kind of anomaly is called RW anomaly (read-write anomaly), because it shows up when a transaction reads an element, and another transaction writes the same element.

- Indeed, this anomaly comes from the fact that a transaction T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress. The fact that T1 is still is progress means that the risk is that T1 works on A without taking into account the changes that T2 makes on A. Therefore, the update of T1 or T2 are lost.

Anomaly 2b: unrepeateable read (RW anomaly)

T_1 executes two consecutive reads of the same data:

T_1	T_2
begin	begin
READ(A,x)	
	READ(A,x)
	x := x+1
	WRITE(A,x)
	commit
READ(A,x)	
commit	

However, due to the concurrent update of T2, T1 reads two different values.

This is another kind of RW (read-write) anomaly.

Anomaly 3: ghost update (WW anomaly)

Assume the following integrity constraint A = B

T_1	T_2
begin	begin
WRITE(A,1)	
	WRITE(B,2)
WRITE(B,1)	
commit	
	WRITE(A,2)
	commit

Note that T1 and T2 in isolation do not violate the integrity constraints. However, the interleaved execution is different from any serial execution. Transaction T1 will see the update of A to 2 as a surprise, and transaction T2 will see the update of B to 1 as a surprise.

This is a WW (write-write) anomaly

Scheduler

The scheduler is part of the transaction manager, and works as follows:

- It deals with new transactions entered into the system, assigning them an identifier
- It instructs the buffer manager so as to read and write on the DB according to a particular sequence
- It is NOT concerned with specific operations on the local store of transactions, nor with constraints on the order of executions of transactions. The last conditions means that every order by which transactions are entered into the system is acceptable to the schedule.

It follows that we can simply characterize each transaction Ti (where i is a nonnegative integer identifying the transaction) in terms of its actions, where each action of transaction Ti is denoted by a letter (read, write, o commit) and the subscript i

The transactions of the previous examples are written as:

T1: r1(A) r1(B) w1(A) w1(B) c1 T2: r2(A) r2(B) w2(A) w2(B) c2

An example of (complete) schedule on these transactions is:

r1(A) r1(B) w1(A) r2(A) r2(B) w2(A) w1(B) c1 w2(B) c2

T1 reads A

T2 writes A

T1 commit

Serializability and equivalence of schedules

As we saw before, the definition of serializability relies on the notion of equivalence between schedules.

Depending on the level of abstraction used to characterize the effects of transactions, we get different notions of equivalence, which in turn suggest different definitions of serializability.

Given a certain definition of equivalence, we will be interested in

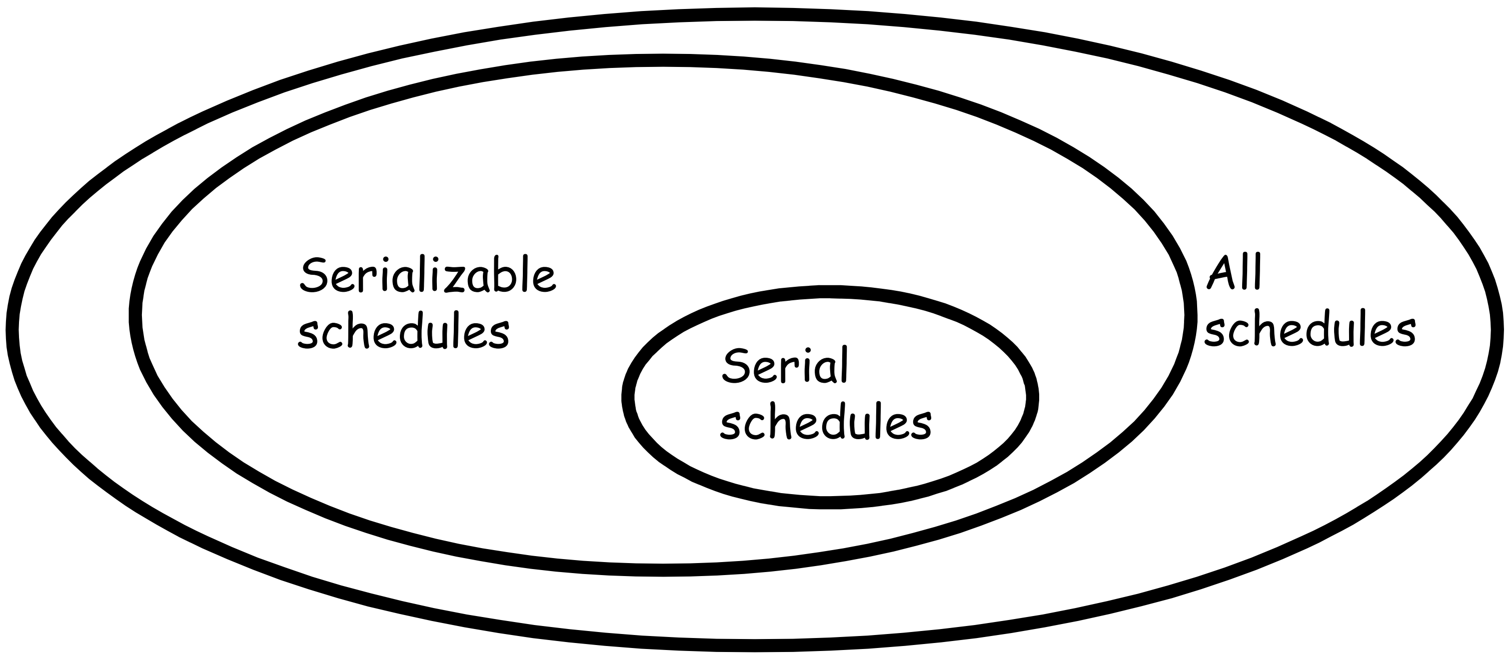
- two types of algorithms:
  - algorithms for checking equivalence: given two schedule, determine if they are equivalent
  - algorithms for checking serializability: given one schedule, check whether it is equivalent to any of the serial schedules on the same transactions
- rules that ensures serializability

Two important assumptions

1. No transaction reads or writes the same element twice
2. No transaction executes the "rollback" command (i.,e. all executions of transactions are successful)

Classes of schedules

Basic idea of our investigation: single out classes of schedules that are serializable, and such that the serializability check can be done with reasonable computational complexity



Conflict-serializability: the notion of conflict

Definition of conflicting actions: Two actions are conflicting in a schedule if they belong to different transactions, they operate on the same element, and at least one of them is a write.

It is easy to see that:

- Two consecutive nonconflicting actions belonging to different transactions can be swapped without changing the effects of the schedule. Indeed,
  - Two consecutive reads of the same elements in different transactions can be swapped
  - One read of X in T1 and a consecutive read of Y in T2 (with Y≠X) can be swapped
- The swap of two consecutive actions of the same transaction can change the effect of the transaction
- Two conflicting consecutive actions cannot be swapped without changing the effects of the schedule, because:
  - Swapping two write operations w1(A) w2(A) on the same elements may result in a different final value for A
  - Swapping two consecutive operations such as r1(A) w2(A) may cause T1 read different values of A (before and after the write of T2, respectively)

Conflict-equivalence

Definition of conflict-equivalence: Two schedules S1 and S2 on the same transactions are conflict-equivalent if S1 can be transformed into S2 through a sequence of swaps of consecutive non-conflicting actions

Example:

S = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)

is conflict-equivalent to:

S’ = r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

because it can be transformed into S’ through the following sequence of swaps:

r1(A) w1(A) r2(A) <u>w2(A) r1(B)</u> w1(B) r2(B) w2(B)

r1(A) w1(A) <u>r2(A) r1(B)</u> w2(A) w1(B) r2(B) w2(B)

r1(A) w1(A) r1(B) r2(A) <u>w2(A) w1(B)</u> r2(B) w2(B)

r1(A) w1(A) r1(B) <u>r2(A) w1(B)</u> w2(A) r2(B) w2(B)

r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

Exercise

Prove the following property:

Two schedules S1 and S2 on the same transactions T1,…,Tn are conflict-equivalent if and only if there are no actions a_i of Ti and b_j of Tj (with Ti and Tj belonging to T1,…Tn) such that

- a_i and b_j are conflicting, and
- the mutual position of the two actions in S1 is different from their mutual position in S2

Conflict-serializability

Definition of conflict-serializability: A schedule S is conflict-serializable if there exists a serial schedule S' that is conflict-equivalent to S

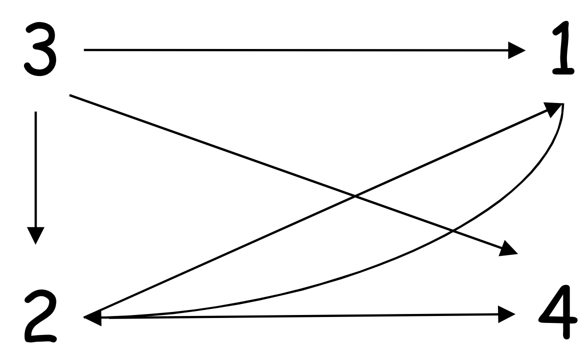
How can conflict-serializability be checked?

We can do it by analyzing the precedence graph associated to a schedule. Given a schedule S on T1,…,Tn, the precedence graph P(S) associated to S is defined as follows:

- the nodes of P(S) are the transactions {T1,…, Tn} of S
- the edges E of P(S) are as follows: the edge Ti → Tj is in E if and only if there exists two actions Pi(A), Qj(A) of different transactions Ti and Tj in S operating on the same object A such that:
  - Pi(A) $<_S$ Qj(A) (i.e., Pi(A) appears before Qj(A) in S)
  - at least one between Pi(A) and Qj(A) is a write operation

Example of precedence graph

S: $w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$



How the precedence graph is used

Theorem (conflict-serializability) A schedule S is conflict-serializable if and only if the precedence graph P(S) associated to S is acyclic.

Exercise: Prove that, if S is a serial schedule, then the precedence graph P(S) is acyclic.

1 - Transaction management

1.1 Transactions, concurrency, serializability
1.2 Recoverability
1.3 Concurrency control through locks
1.4 Concurrency control through timestamps
1.5 Transaction management in SQL

The rollback problem

We now consider the problem of rollback.

The first observation is that, with rollbacks, the notion of serializability that we have considered up to now is not sufficient for achieving the ACID properties.

This fact is testified by the existence of a new anomaly, called “dirty read”.

A new anomaly: dirty read (WR anomaly)

Consider two transactions T1 and T2, both with the commands:

READ(A,x), x:=x+1, WRITE(A,x)

Now consider the following schedule (where T1 executes the rollback):

T_1	T_2
begin	begin
READ(A,x)	
x := x+1	
WRITE(A,x)	
	READ(A,x)
	x := x+1
rollback	
	WRITE(A,x)
	commit

The problem is that T2 reads a value written by T1 before T1 commits or rollbacks. Therefore, T2 reads a “dirty” value, that is shown to be incorrect when the rollback of T1 is executed. The behavior of T2 depends on an incorrect input value.

This is another form of WR (write-read) anomaly.

Commit o rollback?

Recall that, at the end of transaction Ti:

- If Ti has executed the commit operation:
  - the system should ensure that the effects of the transactions are recorded permanently in the database
- If Ti has executed the rollback operation:
  - the system should ensure that the transaction has no effect on the database

Cascading rollback

Note that the rollback of a transaction Ti can trigger the rollback of other transactions, in a cascading mode. In particular:

- If a transaction Tj different from Ti has read from Ti, we should kill Tj (or, Tj should rollback)
- If another transaction Th has read from Tj, Th should in turn rollback
- and so on…

This is called cascading rollback, and the task of the system is to avoid it.

Recoverable schedules

If in a schedule S, a transaction Ti that has read from Tj commits before Tj, the risk is that Tj then rollbacks, so that Ti leaves an effect on the database that depends on an operation (of Tj) that never existed. To capture this concept, we say that Ti is not recoverable.

A schedule S is recoverable if no transaction in S commits before all other transactions it has "read from", commit.

Example of recoverable schedule:

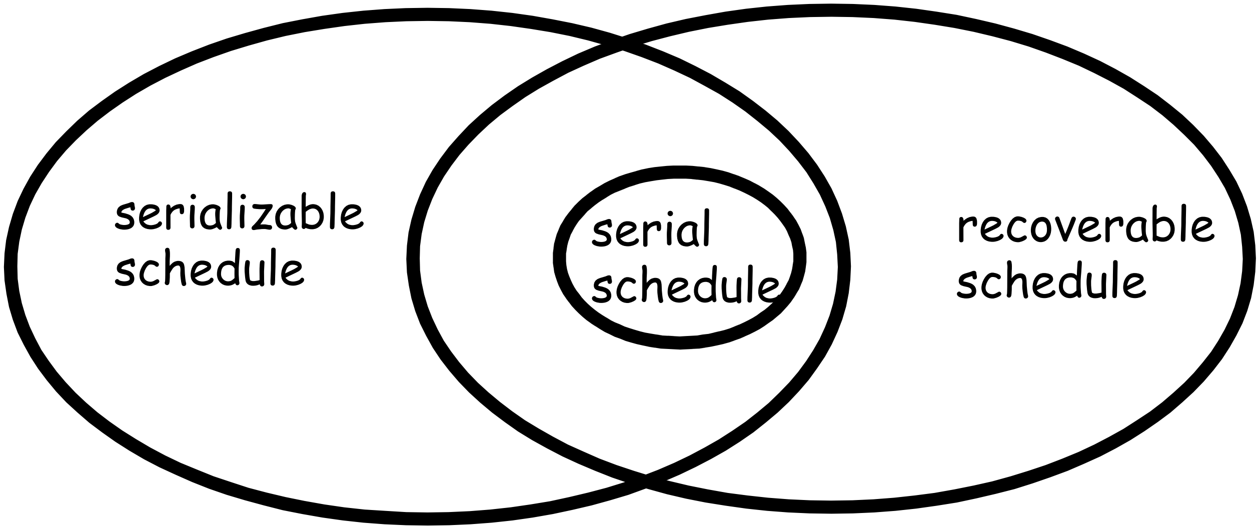
S: w1(A) w1(B) w2(A) r2(B) c1 c2

Example of non-recoverable schedule:

S: w1(A) w1(B) w2(A) r2(B) r3(A) c1 c3 c2

Serializability and recoverability

Serializability and recoverability are two orthogonal concepts: there are recoverable schedules that are non-serializable, and serializable schedule that are not recoverable. Obviously, every serial schedule is recoverable.



For example, the schedule

S1: w2(A) w1(B) w1(A) r2(B) c1 c2

is recoverable, but not serializable (it is not conflict-serializable), whereas the schedule

S2: w1(A) w1(B) w2(A) r2(B) c2 c1

is serializable (in particular, conflict-serializable), but not recoverable

Recoverability and cascading rollback

Recoverable schedules can still suffer from the cascading rollback problem.

For example, in this recoverable schedule

S: w2(A) w1(B) w1(A) r2(B)

if T1 rollbacks, T2 must be killed.

To avoid cascading rollback, we need a stronger condition wrt recoverability: a schedule S avoids cascading rollback (i.e., the schedule is ACR, Avoid Cascading Rollback) if every transaction in S reads values that are written by transactions that have already committed.

For example, this schedule is ACR

S: w2(A) w1(B) w1(A) c1 r2(B) c2

In other words, an ACR schedule blocks the dirty data anomaly.

Summing up

- S is recoverable if no transaction in S commits before the commit of all the transactions it has "read from" Example:

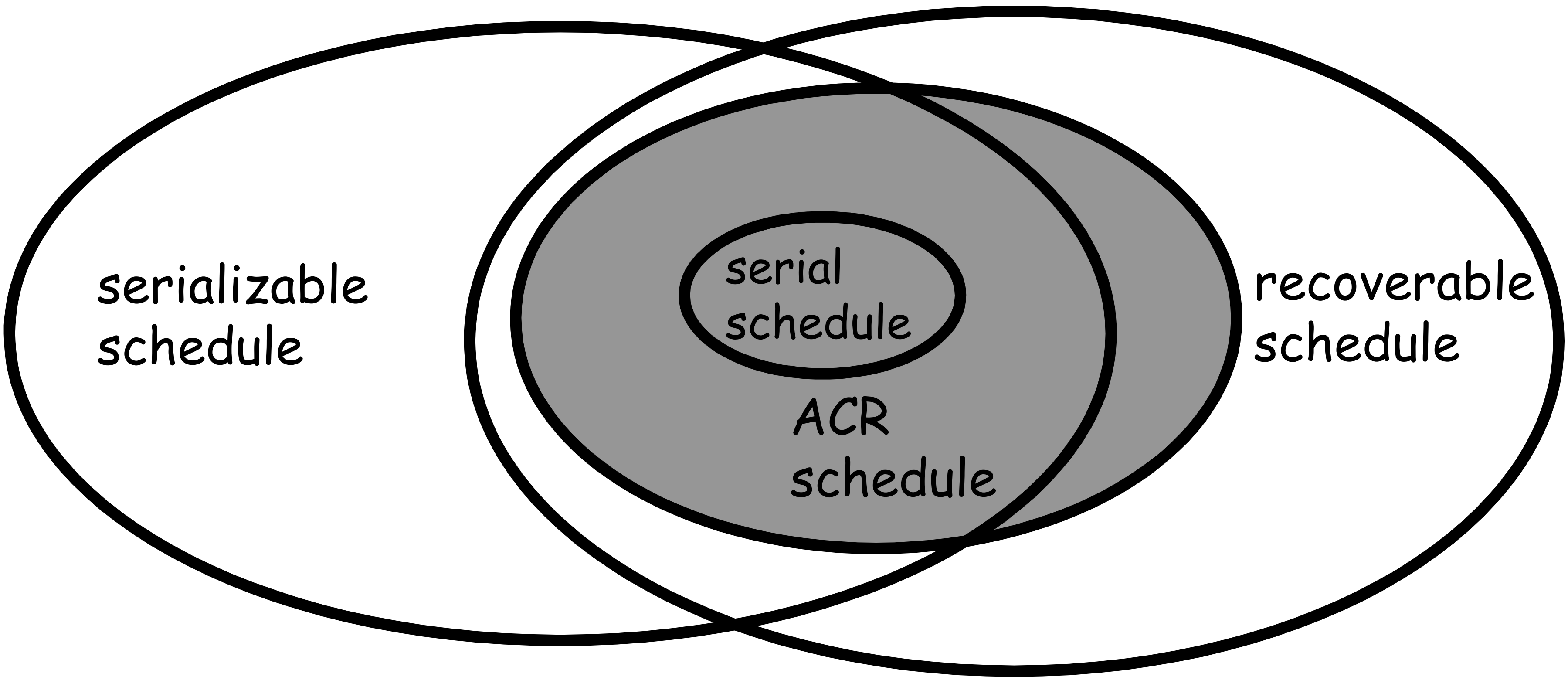
  w1(A) w1(B) w2(A) r2(B) c1 c2

- S is ACR, i.e., avoids cascading rollback, if no transaction "reads from" a transaction that has not committed yet

  Example:

  w1(A) w1(B) w2(A) c1 r2(B) c2

Recoverability and ACR

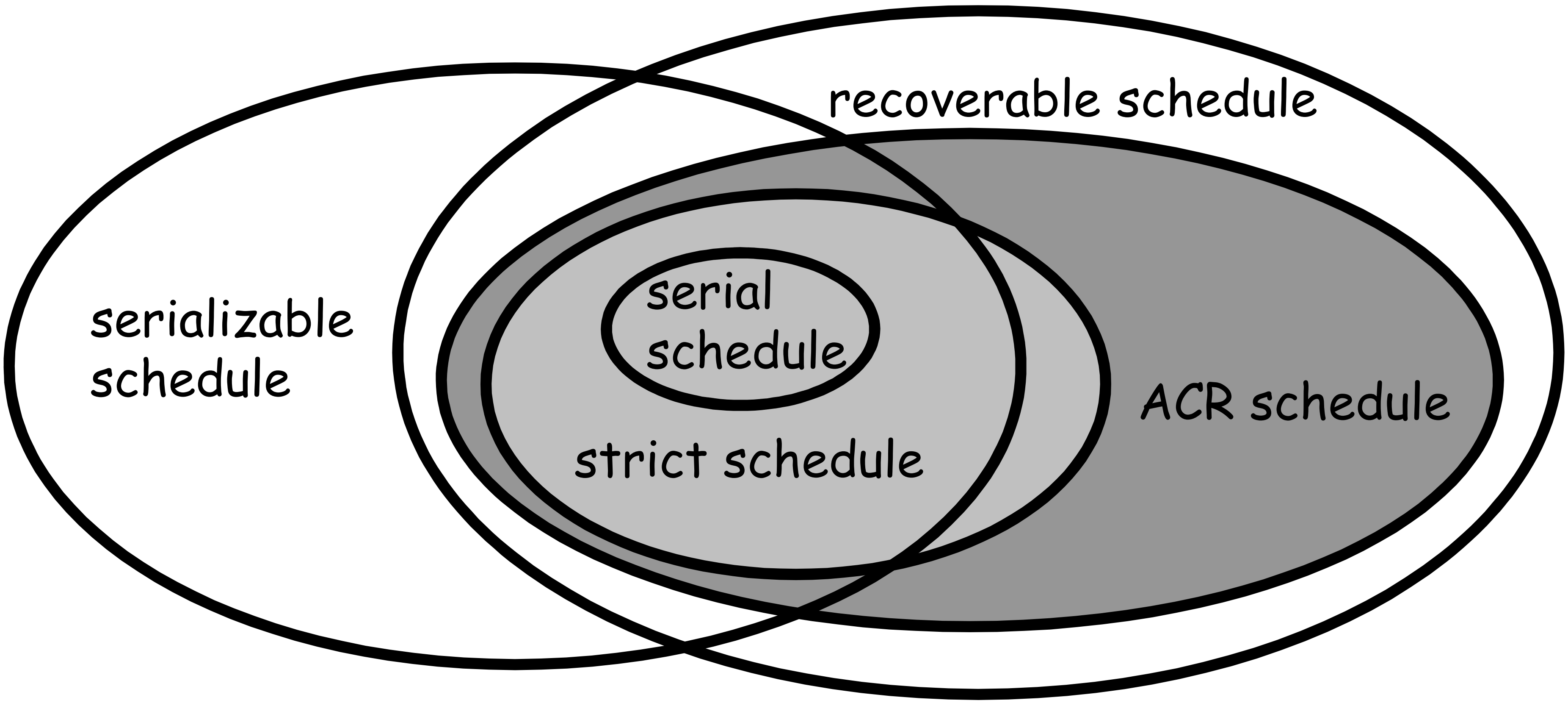


Analogously to recoverable schedules, not all ACR schedules are serializable. Obviously, every ACR schedule is recoverable, and every serial schedule is ACR.

Strict schedules

- We say that, in a schedule S, a transaction Ti writes on Tj if there is a wj(A) in S followed by wi(A), and there is no write action on A in S between these two actions
- We say that a schedule S is strict if every transaction *reads* only values written by transactions that have already committed, and *writes* only on transactions that have already committed
- It is easy to verify that every strict schedule is ACR, and therefore recoverable
- Note that, for a strict schedule, when a transaction Ti rollbacks, it is immediate to determine which are the values that have to be stored back in the database to reflect the rollback of Ti, because no transaction may have written on this values after Ti

Strict schedules and ACR



Obviously, every serial schedule is strict, and every strict schedule is ACR, and therefore recoverable. However, not all ACR schedules are strict.

1 - Transaction management

1.1 Transactions, concurrency, serializability
1.2 Recoverability
1.3 Concurrency control through locks
1.4 Concurrency control through timestamps
1.5 Transaction management in SQL

Concurrency control through locks

- Conflict-serializability is not used in commercial systems
- We will now study a method for concurrency control that is used in commercial systems. Such method is based on the use of lock
- In the methods based on locks, a transaction must ask and get a permission in order to operate on an element of the database. The lock is a mechanism for a transaction to ask and get such a permission

Primitives for exclusive lock

- For the moment, we will consider exclusive locks. Later on, we will take into account more general types of locks

- We introduce two new operations (besides read and write) that can appear in transactions. Such operations are used to request and release the exclusive use of a resource (element A in the database):

  - Lock (exclusive): $l_i(A)$
  - Unlock: $u_i(A)$

- The lock operation $l_i(A)$ means that transaction Ti requests the exclusive use of element A of the database

- The unlock operation $u_i(A)$ means that transaction Ti releases the lock on A, i.e., it renounces the use of A

Well-formed transactions and legal schedules

When using exclusive locks, transactions and schedules should obey two rules:

– Rule 1: Every transaction is well-formed. A transaction Ti is well-formed if every action pi(A) (a read or a write on A) of Ti is contained in a "critical section", i.e., in a sequence of actions delimited by a pair of lock-unlock on A:

Ti: … $l_i(A)$ … $p_i(A)$ … $u_i(A)$ …

– Rule 2: The schedule is legal. A schedule S with locks is legal if no transaction in it locks an element A when a different transaction has granted the lock on A and has not yet unlocked A

S: ……$l_i(A)$ ………………… $u_i(A)$ ……

no $l_j(A)$

Schedule with exclusive locks: examples

S1: l1(A) l1(B) r1(A) w1(B) l2(B) u1(A) u1(B) r2(B) w2(B) u2(B) l3(B) r3(B) u3(B)

T1 well-formed,
S1 not legal

S2: l1(A) r1(A) w1(B) u1(A) u1(B) l2(B) r2(B) w2(B) l3(B) r3(B) u3(B)

T1 ill-formed:
write without lock.
T2 ill-formed: lock
without unlock.

S2 not legal

S3: l1(A) r1(A) u1(A)	l1(B) w1(B) u1(B)	l2(B) r2(B) w2(B) u2(B)	l3(B) r3(B) u3(B)

T1, T2, T3 well-formed,
and S3 legal

Scheduler based on exclusive locks

A scheduler based on exclusive locks behaves as follows:

1. When an action request is issued by a transaction, the scheduler checks whether this request makes the transaction ill-formed, in which case the transaction is aborted by the scheduler.
2. When a lock request on A is issued by transaction Ti, while another transaction Tj has a lock on A, the scheduler does not grant the request (otherwise the schedule would become illegal), and Ti is blocked until Tj releases the lock on A.
3. To trace all the locks granted, the scheduler manages a table of locks, called lock table

In other words, the scheduler ensures that the current schedule is legal and all its transactions are well-formed.

Example of scheduler behaviour

T1	T2
l1(A); r1(A)	
A:=A+100; w1(A);	
	l2(A) – blocked!
l1(B); r1(B); u1(A);	
	l2(A) – re-started!
	r2(A)
	A:=Ax2; w2(A); u2(A)
B:=B+100; w1(B); u1(B)	
	l2(B); r2(B)
	B:=Bx2; w2(B); u2(B)

Is this sufficient for serializability?

T1	T2	A 25	B 25
l1(A); r1(A)			
A:=A+100; w1(A); u1(A)		125	
	l2(A); r2(A)		
	A:=Ax2; w2(A); u2(A)	250	
	l2(B); r2(B)		
	B:=Bx2; w2(B); u2(B)		50
l1(B); r1(B)			
B:=B+100; w1(B); u1(B)			150
		250	150

Ghost update: isolation is not ensured by the use of locks

Two-Phase Locking (with exclusive locks)

We have seen that the two rules for
- well-formed transactions
- legal schedules

are not sufficient for guaranteeing serializability

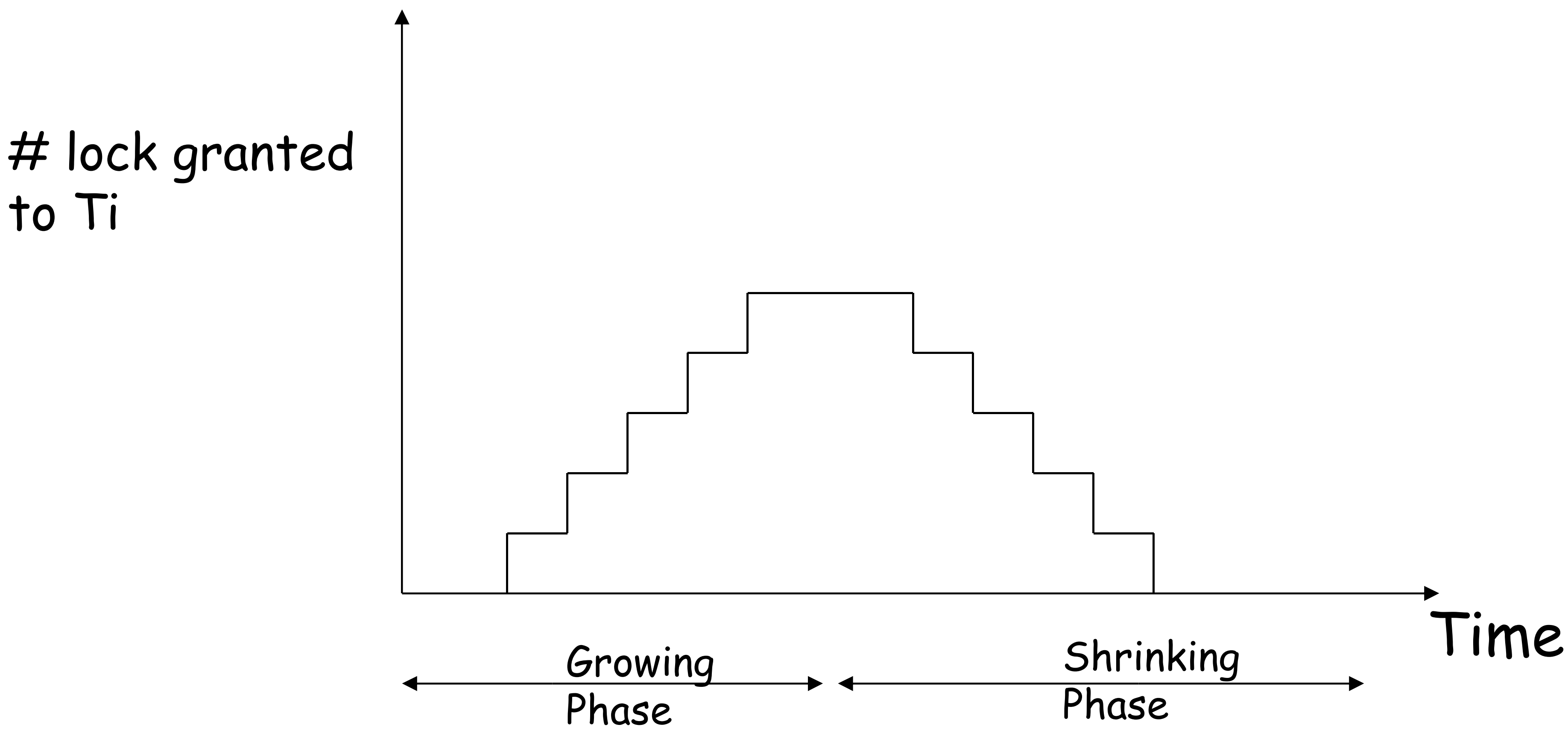
To come up with a correct policy for concurrency control through the use of exclusive locks, we need a further rule (or, protocol), called "Two-Phase Locking (2PL)":

Definition of two-phase locking (with only exclusive locks): A schedule S with exclusive locks follows the two-phase locking protocol if in each transaction Ti appearing in S, all lock operations precede all unlock operations.

$$S: \quad \underbrace{\ldots\ldots\ldots}_{\text{no unlock}} \; l_i(A) \;\ldots\ldots\; u_i(A) \; \underbrace{\ldots\ldots}_{\text{no lock}}$$

The two phases of Two-Phase Locking

Locking and unlocking scheme in a transaction following the 2PL protocol



Example of a 2PL schedule

T1	T2	A	B
		25	25
l1(A); r1(A)			
A:=A+100; w1(A); u1(A)		125	
	l2(A); r2(A)		
	A:=Ax2; w2(A); u2(A)	250	
	l2(B); r2(B)		
	B:=Bx2; w2(B); u2(B)		50
l1(B); r1(B)			
B:=B+100; w1(B); u1(B)			150
		250	150

How the scheduler works in the 2PL protocol

T1	T2
l1(A); r1(A) A:=A+100; w1(A) l1(B)	
u1(A)	
	l2(A); r2(A) A:=Ax2; w2(A); l2(B) – blocked
r1(B) B:=B+100 w1(B); u1(B)	
	l2(B); - re-started u2(A); r2(B) B:=Bx2; w2(B); u2(B)

The 2PL protocol avoids the ghost update while accepting concurrency

Note that the scheduler still checks that the schedule is legal

The risk of deadlock

T1	T2
l1(A); r1(A)	
	l2(B); r2(B)
A:=A+100;	
	B:=Bx2
w1(A)	
	w2(B)
l1(B) – blocked	l2(A) – blocked

S: l1(A) r1(A) l2(B) r2(B) w1(A) w2(B) l1(B) l2(A)

To ensure that the schedule is legal, the scheduler blocks both T1 and T2, and none of the two transactions can proceed. This is a deadlock (we will come back to the methods for deadlock management).

Who issues the lock/unlock commands?

So far, we have assumed that transactions issue the lock/unlock commands. However, this is not necessary.

Indeed, we can design a scheduler in such a way that it inserts the lock/unlock commands while respecting the following conditions:

- Every transaction is well-formed
- The schedule is legal (if at all possible)
- Each transaction, extended with the inserted lock/unclock commands, follows the 2PL protocol

For this reason, even in the presence of locks, we will continue to denote a schedule by means of a sequence of read/write/commit commands. For example, the schedule

l1(A) r1(A) l1(B) u1(A) l2(A) w2(A) r1(B) w1(B) u1(B) l2(B) u2(A) r2(B) w2(B) u2(B)

can be denoted as:

r1(A) w2(A) r1(B) w1(B) r2(B) w2(B)

Scheduler based on exclusive locks and 2PL

We study how a scheduler based on exclusive locks and 2PL behaves during the analysis of the current schedule (obviously, not necessarily complete):

1. If a request by transaction Ti shows that Ti is not well-formed, then Ti is aborted by the scheduler
2. If a lock request by transaction Ti shows that Ti does not follow the 2PL protocol, then Ti is aborted by the scheduler
3. If a lock is requested for A by transaction Ti while A is used by a different transaction Tj, then the scheduler blocks Ti, until Tj releases the lock on A. If the scheduler figures out that a deadlock has occurred (or will occur), then the scheduler adopts a method for deadlock management
4. To trace all the locks granted, the scheduler manages a table of locks, called lock table

Note that (1) and (2) do not occur if the lock/unlock commands are automatically insterted by the scheduler.

Simply put, the above behaviour means that the scheduler ensures that

1. the current schedule is legal
2. all its transactions are well-formed
3. all its transactions follow the 2PL protocol

2PL and conflict-serializability

To compare 2PL and conflict-serializability, we make use of the above observation, and note that every schedule that includes lock/unlock operations can be seen as a “traditional” schedule (by simply ignoring such operations)

Theorem Every legal schedule constituted by well-formed transactions following the 2PL protocol (with exclusive locks) is conflict-serializable.

2PL and conflict-serializability

Theorem There exists a conflict-serializable schedule that does not follow the 2PL protocol (with exclusive locks).

Proof It is sufficient to consider the following schedule S:

r1(x) w1(x) r2(x) w2(x) r3(y) w1(y)

S is obviously conflict-serializable (the serial schedule T3,T1,T2 is conflict-equivalent to S), but it is easy to show that we cannot insert in S the lock/unlock commands in such a way that all transactions are well-formed and follow the 2PL protocol, and the resulting schedule is legal. Indeed, it suffices to notice that we should insert in S the command u1(x) before r2(x), because in order for T2 to read x it must hold the exclusive lock on x, and we should insert in S the command l1(y) after r3(y), because in order for T3 to read y it must hold the exclusive lock on y, and therefore, the command l1(y), which is necessary for executing w3(y), cannot be issued before r3(y). It follows that we cannot insert into S the lock/unlock commands in such a way that the 2PL protocol is respected.

Shared locks

With exclusive locks, a transaction reading A must unlock A before another transaction can read the same element A:

S: ... l1(A) r1(A) u1(A) … l2(A) r2(A) u2(A) …

Actually, this looks too restrictive, because the two read operations do not create any conflict. To remedy this situation, we introduce a new type of lock: the shared lock. We denote by sli(A) the command for the transaction Ti to ask for a shared lock on A.

With the use of shared locks, the above example changes as follows:

S: ... sl1(A) r1(A) sl2(A) r2(A) …. u1(A) u2(A)

The primitive for locks are now as follows:

xli(A): exclusive lock (also called write lock)

sli(A): shared lock (also called read lock)

ui(A): unlock

Well-formed transactions with shared locks

With shared and exclusive locks, the following rule must be respected.

Rule 1: We say that a transaction Ti is well-formed if

- every read ri(A) is preceded either by sli(A) or by xli(A), with no ui(A) in between,
- every wi(A) is preceded by xli(A) with no ui(A) in between,
- every lock (sl or xl) on A by Ti is followed by an unlock on A by Ti.

Note that we allow Ti to first execute sli(A), probably for reading A, and then to execute xli(A), probably for writing A without the unlock of A by means of T. The transition from a shared lock on A by T to an exclusive lock on the same element A by T (without an unlock on A by T) is called "lock upgrade".

Legal schedule with shared locks

With shared and exclusive locks, the following rule must also be respected.

Rule 2: We say that a schedule S is legal if

- an xli(A) is not followed by any xlj(A) or by any slj(A) (with j different from i) without an ui(A) in between
- an sli(A) is not followed by any xlj(A) (with j different from i) without an ui(A) in between

Two-phase locking (with shared locks)

With shared locks, the two-phase locking rule becomes:

Definition of two-phase locking (with exclusive and shared locks): A schedule S (with shared and exclusive locks) follows the 2PL protocol if in every transaction Ti of S, all lock operations (either for exclusive or for shared locks) precede all unlocking operations of Ti.

In other words, no action sli(X) or xli(X) can be preceded by an operation of type ui(Y) in the schedule.

How locks are managed

- The scheduler uses the so-called "compatibility matrix" (see below) for deciding whether a lock request should be granted or not.
- In the matrix, "S" stands for shared lock, "X" stands for exclusive lock, "yes" stands for "requested granted" and "no" stands for "requested not granted"

		New lock requested by Tj ≠ Ti on A	
		S	X
Lock already granted to Ti on A	S	yes	no
	X	no	no

How locks are managed

- The problem for the scheduler of automatically inserting the lock/unlock commands becomes more complex in the presence of shared locks.
- Also, the execution of the unlock commands requires more work. Indeed, when an unlock command on A is issued by Ti, there may be several transactions waiting for a lock (either shared on exclusive) on A, and the scheduler must decide to which transaction to grant the lock. Several methods are possible:
  - First-come-first-served
  - Give priorities to the transactions asking for a shared lock
  - Give priorities to the transactions asking for a lock upgrade

  The first method is the most used one, because it avoids "starvation", i.e., the situation where a request of a transaction is never granted.

Exercise 7

Consider the following schedule S:

r1(A) r2(A) r2(B) w1(A) w2(D) r3(C) r1(C) w3(B) c2 r4(A) c1 c4 c3

and tell whether S is in the class of 2PL schedules with shared and exclusive locks

Exercise 7: solution

The schedule S:

r1(A) r2(A) r2(B) w1(A) w2(D) r3(C) r1(C) w3(B) c2 r4(A) c1 c4 c3

is in the class of 2PL schedules with shared and exclusive locks. This can be shown as follows:

sl1(A) r1(A) sl2(A) r2(A) sl2(B) r2(B) xl2(D) u2(A) xl1(A) w1(A) w2(D)
sl3(C) r3(C) sl1(C) r1(C) u1(C) u1(A) u2(B) u2(D) xl3(B) w3(B) u3(B) u3(C) c2 sl4(A) r4(A) u4(A) c1 c4 c3

Properties of two-phase locking (with shared locks)

The properties of two-phase locking with shared and exclusive locks are similar to the case of exclusive locks only:

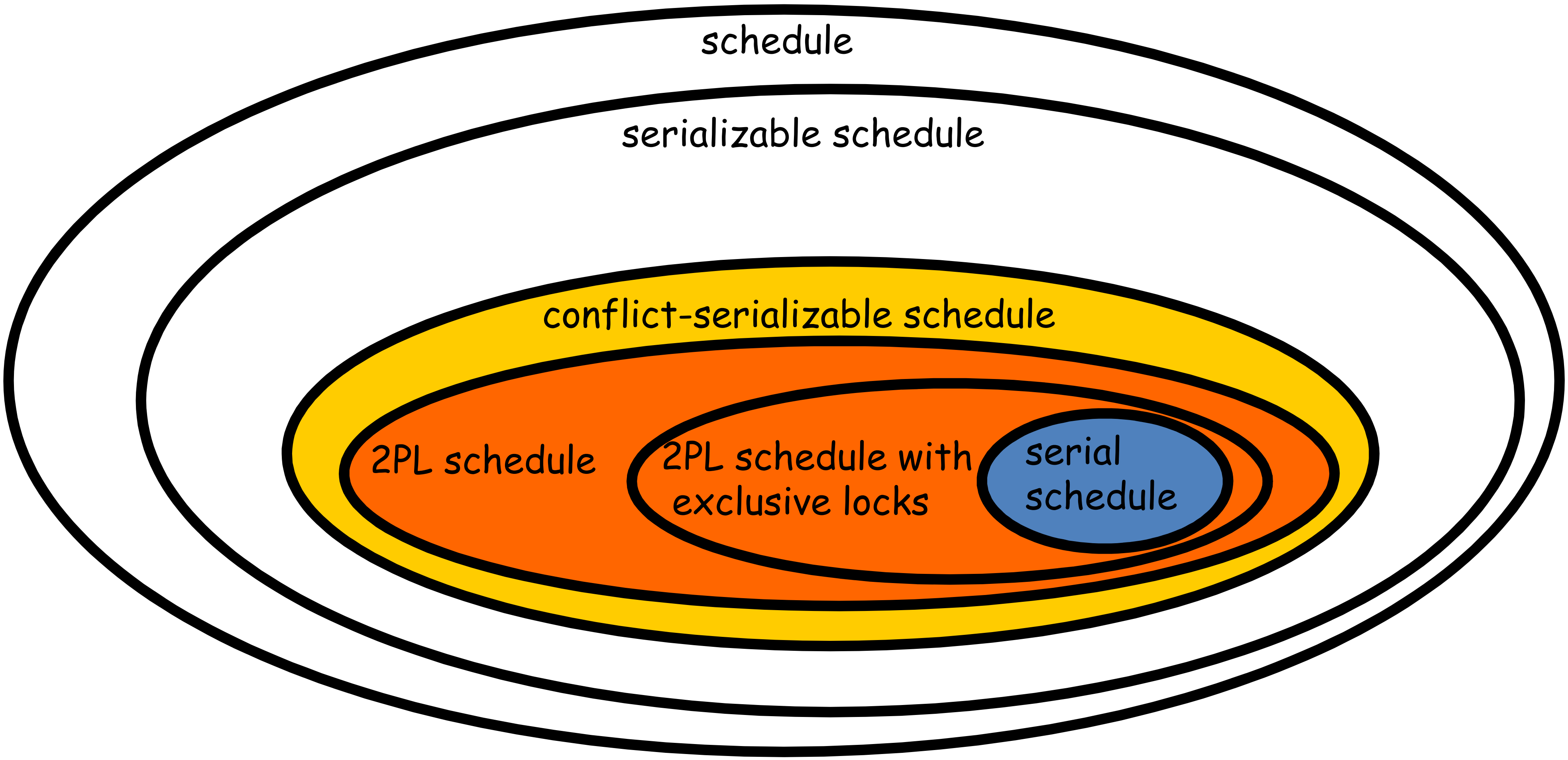
- **Theorem** Every legal schedule with well-formed transactions following the two-phase locking protocol (with exclusive and shared locks) is conflict-serializable.
- **Theorem** There exists a conflict-serializable schedule that does not follow the 2PL protocol (with exclusive and shared locks).

- With shared locks, the risk of deadlock is still present, like in:

sl1(A) sl2(A) xl1(A) xl2(A)

2PL and conflict-serializability

We denote by “2PL schedule” the class of legal schedules with shared and exclusive locks constituted by well-formed transactions following the 2PL protocol.



Recoverability and 2PL

- So far, when discussing about recoverability, ACR, strictness and rigorousness we focused on:
  - read, write
  - rollback
  - commit

- We still have to study the impact of these notions on the locking mechanisms and the 2PL protocol

Strict two-phase locking (strict 2PL)

A schedule S is said to be in the strict 2PL class if

- S is in 2PL, and
- S is strict.

Tj	Ti
wj(A)	
……	
commit	
uj(A)	
	ri(A)

Strict two-phase locking (strict 2PL)

With the goal of capturing the class of strict 2PL the following protocol has been defined: A schedule S follows the strict 2PL protocol if it follows the 2PL protocol, and all exclusive locks of every transaction T are kept by T until either T commits or rollbacks.

Tj	Ti
wj(A)	
rj(B)	
……	
uj(B)	
commit	
uj(A)	
	ri(A)

Properties of strict 2PL

- Every schedule following the strict 2PL protocol is strict:
  (See exercise 7)

- Every schedule following the strict 2PL protocol is serializable:
  it can be shown that every strict 2PL schedule S is conflict-equivalent to the serial schedule S' obtained from S by ignoring the transactions that have rollbacked, and by choosing the order of transactions determined by the order of commit (the first transaction in S' is the first that has committed, the second transaction in S' is the second that has committed, and so on)

Exercise 8

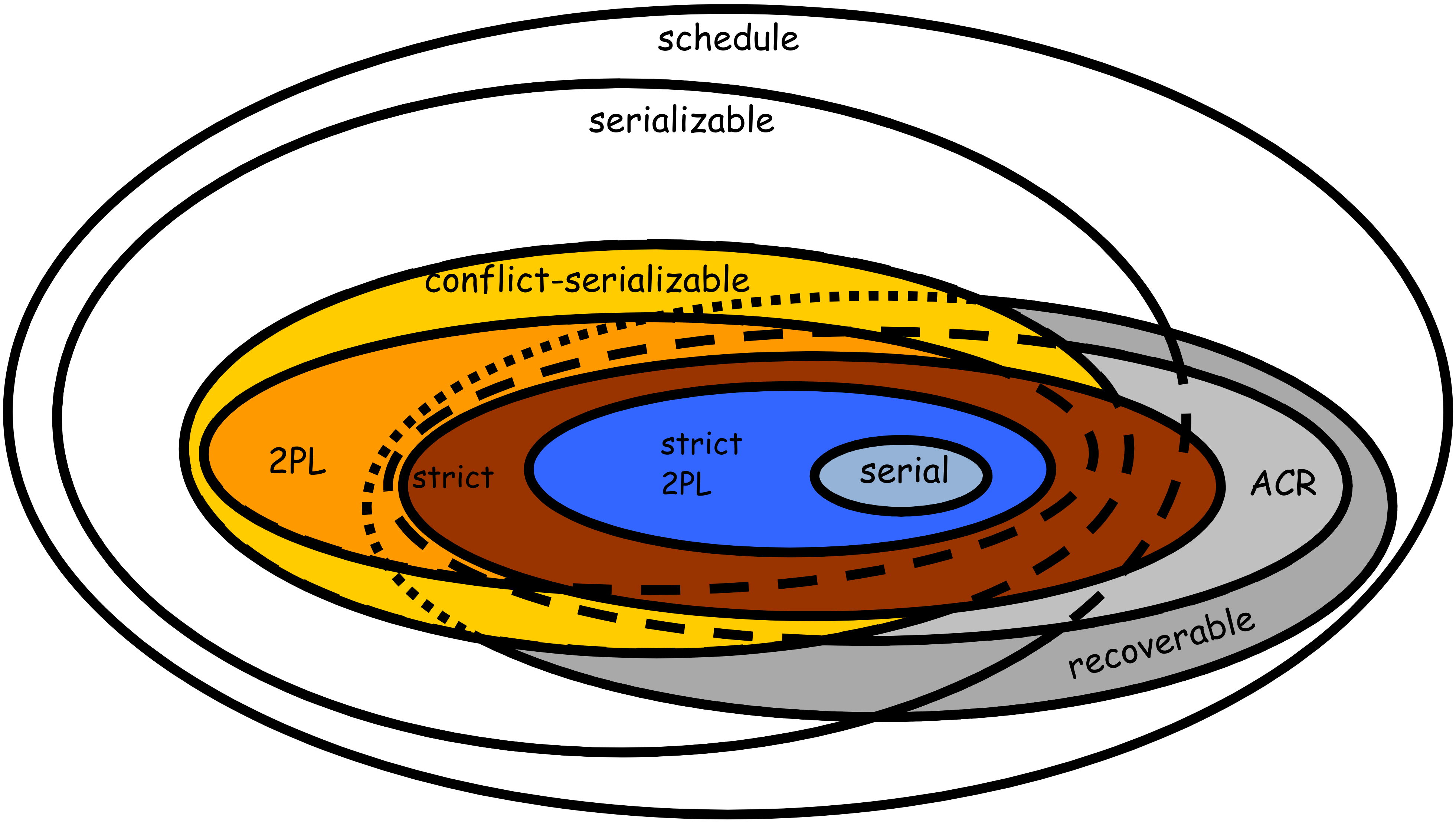
- Prove or disprove the following statement:

  Every schedule following the strict 2PL protocol is strict.

- Prove or disprove the following statement:

  Every schedule that is strict and follows the 2PL protocol also follows the strict 2PL protocol.

The complete picture



1 - Transaction management

Concurrency based on timestamps

- Each transaction T has an associated timestamp ts(T) that is unique among the active transactions, and is such that ts(Tj) < ts(Th) whenever transaction Ti arrives at the scheduler before transaction Th. In what follows, we assume that the timestamp of transaction Ti is simply i: ts(Ti)=i.
- Note that the timestamps actually define a total order on transactions, in the sense that they can be considered ordered according to the order in which they arrive at the scheduler.
- Note also that every schedule respecting the timestamp order is conflict-serializable, because it is conflict-equivalent to the serial schedule corresponding to the timestamp order.
- Obviously, the use of timestamp avoids the use of locks. Note, however, that deadlock may still occur.

The use of timestamps

- Transactions execute without any need of protocols.

- The basic idea is that, at each action execution, the scheduler checks whether the involved timestamps violates the serializability condition according to the order induced by the timestamps.

- In particular, we maintain the following data for each element X:
  - rts(X): the highest timestamp among the active transactions that have read X
  - wts(X): the highest timestamp among the active transactions that have written X (this coincides with the timestamp of the last transaction that wrote X)
  - wts-c(X): the timestamp of the last committed transaction that has written X
  - cb(X): a bit (called commit-bit), that is false if the last transaction that wrote X has not committed yet, and true otherwise.

The rules for timestamps

- Basic idea:
  - the actions of transaction T in a schedule S must be considered as being logically executed in one spot
  - the logical time of an action of T is the timestamp of T, i.e., ts(T)
  - the commit-bit is used to avoid the dirty read anomaly
- The system manages two "temporal axes", corresponding to the "physical" and to the "logical" time. The values rts(X) and wts(X) indicate the timestamp of the transaction that was the last to read and write X according to the logical time.
- An action of transaction T executed at the physical time t is accepted if its ordering according to the physical temporal order is compatible with respect to the logical time ts(T)
- This "compatibility principle" is checked by the scheduler.
- As we said before, we assume that the timestamp of each transaction Ti coincide with the subscript i: ts(Ti)=i. In what follows, t1,…,tn will denote physical times.

Rules – case 1a (read ok)

Case 1.a ⟶

B(T1)	B(T2)	B(T3)	w1(X)	r3(X)	r2(X)
t1	t2	t3	t4	t5	t6

Consider r2(X) with respect to the last write on X, namely w1(X):

- the physical time of r2(X) is t6, that is greater than the physical time of w1 (t4)
- the logical time of r2(X) is ts(T2), that is greater than the logical time of w1(X), which is wts(X) = ts(T1)

We conclude that there is no incompatibility between the physical and the logical time, and therefore we proceed as follows:

1. if cb(X) is true, then
   - generally speaking, after a read on X of T, rts(X) should be set to the maximum between rts(X) and ts(T) – in the example, although, according to the physical time, r2(X) appears after the last read r3(X) on X, it logically precedes r3(X), and therefore, if cb(X) was true, rts(X) would remain equal to ts(T3)
   - r2(X) is executed, and the schedule goes on
2. if cb(X) is false (as in the example), then T2 is put in a state waiting for the commit or the rollback of the transaction T' that was the last to write X (i.e., a state waiting for cb(X) equal true -- indeed, cb(X) is set to true both when T' commits, and when T' rollbacks, because the transactions T'' that was the last to write X before T' obviously committed, otherwise T' would be still blocked)

Rules – case 1b (read too late)

Case 1.b

B(T1)	B(T2)	w2(X)	r1(X)
t_1	t_2	t_3	t_4

Consider r1(X) with respect to the last write on X, namely w2(X):

- the physical time of r1(X) is t4, that is greater than the physical time of w2(X), that is t3
- the logical time of r1(X) is ts(T1), that is less than the logical time of w2(X), i.e., wts(X) = ts(T2)

We conclude that r1(X) and w2(X) are incompatible.

Action r1(X) of T1 cannot be executed, T1 rollbacks, and a new execution of T1 starts, with a new timestamp.

Rules – case 2a (write ok)

Case 2.a

B(T1)	B(T2)	B(T3)	r1(X)	w2(X)	w3(X)
t1	t2	t3	t4	t5	t6

Consider w3(X) with respect to the last read on X (r1(X)) and the last write on X (w2(X)):

- the physical time of w3(X) is greater than that of r1(X) and w2(X)
- the logical time of w3(X) is greater than that of r1(X) and w2(X)

We can conclude that there is no incompatibility. Therefore:

1. if cb(X) is true or no active transaction wrote X, then
   - we set wts(X) to ts(T3)
   - we set cb(X) to false
   - action w3(X) of T3 is executed, and the schedule goes on
2. else T3 is put in a state waiting for the commit or the rollback of the transaction T' that was the last to write X (i.e., a state waiting for cb(X) equal true -- indeed, cb(X) is set to true both when T' commits, and when T' rollbacks, because the transactions T'' that was the last to write X before T' obviously committed, otherwise T' would be still blocked)

Rules – case 2b (Thomas rule)

Case 2.b

B(T1)	B(T2)	B(T3)	r1(X)	w2(X)	……	w1(X)
t1	t2	t3	t4	t5	……	t6

- Consider w1(X) with respect to the last read r1(X) on X: the physical time of w1(X) is greater than the physical time of r1(X), and, since w1(X) and r1(X) belong to the same transaction, there is no incompatibility with respect to the logical time.
- However, on the logical time dimension, w2(X) comes after the write w1(X), and therefore, the execution of w1(X) would correspond to an update loss. Therefore:
  1. If cb(X) is true, we simply ignore w1(X) (i.e., w1(X) is not executed). In this way, the effect is to correctly overwrite the value written by T1 on X with the value written by T2 on X (it is like pretending that w1(X) came before w2(X)
  2. if cb(X) is false, we let T1 waiting either for the commit or for the rollback of the transaction that was the last to write X (i.e., a state waiting for cb(X) equal true -- indeed, cb(X) is set to true both when T' commits, and when T' rollbacks, because the transactions T'' that was the last to write X before T' obviously committed, otherwise T' would be still blocked)

Rules – case 2c (write too late)

Case 2.c				
	B(T1)	B(T2)	r2(X)	w1(X)
	t1	t2	t3	t4

Consider w1(X) with respect to the last read r2(X) on X:

- the physical time of w1(X) is t4, that is greater than the physical time of r2(X), i.e., t3
- the logical time of w1(X) is ts(T1), that is less than the logical time of r2(X), that is rts(X) = ts(T2)

We conclude that w1(X) and r2(X) are incompatible.

Action w1(X) is not executed, T1 is aborted, and is executed again with a new timestamp.

Timestamp-based method: the scheduler

```
Action ri(X):
    if      ts(Ti) >= wts(X)
    then    if cb(X)=true or ts(Ti) = wts(X)                              // (case 1.a)
            then set rts(X) = max(ts(Ti), rts(X)) and execute ri(X)       // (case 1.a.1)
            else   put Ti in "waiting" for the commit or the
                   rollback of the last transaction that wrote X          // (case 1.a.2)
    else    rollback(Ti)                                                  // (case 1.b)


Action wi(X):
    if      ts(Ti) >= rts(X) and ts(Ti) >= wts(X)
    then    if cb(X) = true
            then set wts(X) = ts(Ti), cb(X) = false, and execute wi(X)    // (case 2.a.1)
            else   put Ti in "waiting" for the commit or the
                   rollback of the last transaction that wrote X          // (case 2.a.2)
    else    if ts(Ti) >= rts(X) and ts(Ti) < wts(X)                       // (case 2.b)
            then if cb(X)=true
                   then ignore wi(X)                                      // (case 2.b.1)
                   else   put Ti in "waiting" for the commit or the
                          rollback of the last transaction that wrote X   // (case 2.b.2)
            else   rollback(Ti)                                           // (case 2.c)
```

Timestamp-based method: the scheduler

When Ti executes ci:

<u>for each</u> element X written by Ti,

set cb(X) = true

<u>for each</u> transaction Tj waiting for cb(X)=true or for the rollback of the transaction that was the last to write X, allow Tj to proceed

choose the transaction that proceeds

When Ti executes the rollback bi:

<u>for each</u> element X written by Ti, set wts(X) to be wts-c(X), i.e., the timestamp of the transaction Tj that wrote X before Ti, and set cb(X) to true (indeed, Tj has surely committed)

<u>for each</u> transaction Tj waiting for cb(X)=true or for the rollback of the transaction that was the last to write X allow Tj to proceed

choose the transaction that proceeds

Deadlock with the timestamps

Unfortunately, the method based on timestamps does not avoid the risk of deadlock (although the probability is lower than in the case of locks).

The deadlock is related to the use of the commit-bit. Consider the following example:

w1(B), w2(A), w1(A), r2(B)

When executing w1(A), T1 is put in waiting for the commit or the rollback of T2. When executing r2(B), T2 is put in waiting for the commit or the rollback of T1.

The deadlock problem in the method based on timestamps is handled with the same techniques used in the 2PL method.

The method based on timestamp: example

Action	Effect	New values
r6(A)	ok	rts(A) = 6
r8(A)	ok	rts(A) = 8
r9(A)	ok	rts(A) = 9
w8(A)	no	T8 aborted
w11(A)	ok	wts(A) = 11
r10(A)	no	T10 aborted
c11	ok	cb(A) = true

Timestamps and conflict-serializability

- There are conflict-serializable schedules that are not accepted by the timestamp-based scheduler, such as:

  r1(Y) r2(X) w1(X)

- If the schedule S is accepted by the timestamp-based scheduler and does not use the Thomas rule, then the schedule obtained from S by removing all actions of rollbacked transactions is conflict-serializable

- If the schedule S is accepted by the timestamp-based scheduler and does use the Thomas rule, then S may be non conflict-serializable, like for example:

  r1(A) w2(A) c2 w1(A) c1

  However, if the schedule S is accepted by the timestamp-based scheduler and does use the Thomas rule, then the schedule obtained from S by removing all actions ignored by the Thomas rules and all actions of rollbacked transactions is conflict-serializable

Comparison between timestamps and 2PL

- There are schedules that are accepted by timestamp-based schedulers that are not 2PL, such as

r1(A) w2(A) r3(A) r1(B) w2(B) r1(C) w3(C) r4(C) w4(B) w5(B)

(that is not 2PL because T2 must release the lock on A before asking for the lock on B)

- Obviously, there are schedules that are accepted by the timestamp-based schedulers and are also strict 2PL schedules, such as the serial schedule:

r1(A) w1(A) r2(A) w2(A)

- There are strong strict 2PL schedules that are not accepted by the timestamp-based scheduler, such as:

r1(B) r2(A) w2(A) r1(A) w1(A)

Comparison between timestamps and 2PL

- Waiting stage
  - 2PL: transactions are put in waiting stage
  - TS: transactions are killed and re-started

- Serialization order
  - 2PL: determined by conflicts
  - TS: determined by timestamps

- Need to wait for commit by other transactions
  - 2PL: solved by the strong strict 2PL protocol
  - TS: buffering of write actions (waiting for cb(X) = true)

- Deadlock
  - 2PL: risk of deadlock
  - TS: deadlock is less probable

Comparison between timestamps and 2PL

- Timestamp-based method is superior when transactions are "read-only", or when concurrent transactions rarely write the same elements

- 2PL is superior when the number of conflicts is high because:
  - although locking may delay transactions and may cause deadlock (and therefore rollback),
  - the probability of rollback is higher in the case of the timestamp-based method, and this causes a greater global delay of the system

- In the following picture (page 101), the set indicated by "timestamp" denotes the set of schedules generated by the timestamp-based scheduler, where all actions ignored by the Thomas rule and all actions of rollbacked transactions are removed

Multiversion timestamp

Idea: do not block the read actions! This is done by introducing different versions X1 … Xn of element X, so that every read can be always executed, provided that the "right" version (according to the logical time determined by the timestamp) is chosen

- Every "legal" write wi(X) generates a new version Xi (in our notation, the subscript corresponds to the timestamp of the transaction that generated X)
- To each version Xh of X, the timestamp wts(Xh)=ts(Th) is associated, denoting the ts of the transaction that wrote that version
- To each version Xh of X, the timestamp rts(Xh)=ts(Ti) is associated, denoting the highest ts among those of the transactions that read Xh

The properties of the multiversion timestamp are similar to those of the timestamp method.

New rules for the use of timestamps

The scheduler uses timestamps as follows:

- when executing wi(X): if a read rj(Xk) such that wts(Xk) < ts(Ti) < ts(Tj) already occurred, then the write is refused (it is a “write too late” case, because transaction Tj, that is older than Ti, has already read a version of X that precedes version Xi), otherwise the write is executed on a new version Xi of X, and we set wts(Xi) = ts(Ti).
- ri(X): the read is executed on the version Xj such that wts(Xj) is the highest write timestamp among the versions of X having a write timestamp less than or equal to ts(Ti), i.e.: Xj is such that wts(Xj) <= ts(Ti), and there is no version Xh such that wts(Xj) < wts(Xh) <= ts(Ti). Note that such a version always exists, because it is impossible that all versions of X are greater than ts(Ti). Obviously, rts(Xj) is updated in the usual way.
- For Xj with wts(Xj) such that no active transaction has timestamp less than j, the versions of X that precede Xj are deleted, from the oldest to the newest.
- To ensure recoverability, the commit of Ti is delayed until all commit of the transactions Tj that wrote versions read by Ti are executed.

New rules for the use of timestamps

The scheduler uses suitable data structures:

- For each version Xi the scheduler maintains a range range(Xi) = [wts, rts], where wts is the timestamp of the transaction that wrote Xi, and rts is the highest timestamp among those of the transactions that read Xi (if no one read Xi, then rts=wts).
- We denote with ranges(X) the set:

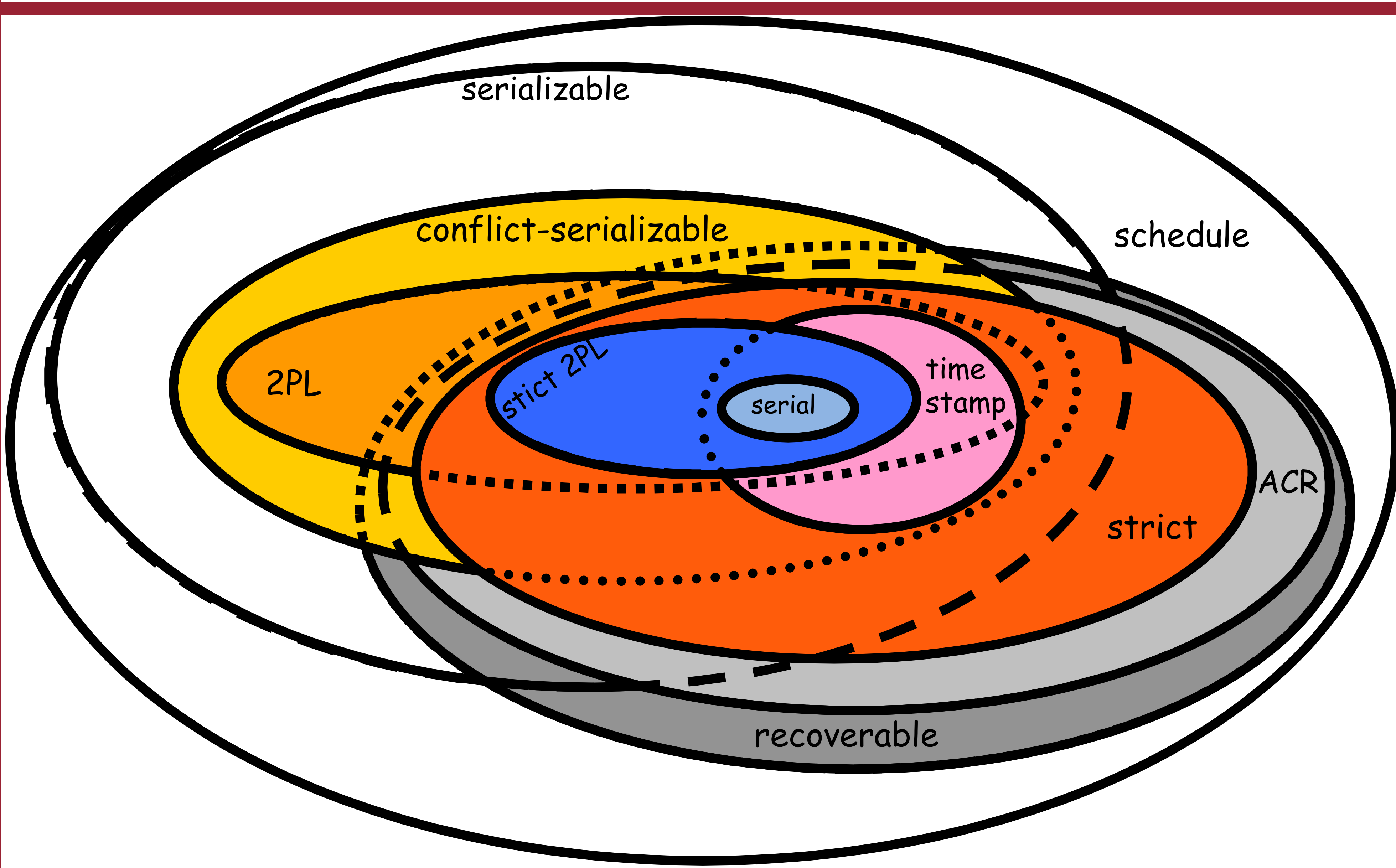
  { range(Xi) | Xi is a version of X }
- When ri(X) is processed, the scheduler uses ranges(X) to find the version Xj such that range(Xj) = [wts, rts] has the highest wts that is less than or equal to the timestamp ts(Ti) of Ti. Moreover, if ts(Ti) > rts, then the rts of range(Xj) is set to ts(Ti).
- When wi(x) is processed, the scheduler uses ranges(X) to find the version Xj such that range(Xj) = [wts, rts] has the highest wts that is less than or equal to the timestamp ts(Ti) of Ti. Moreover, if rts > ts(Ti), then wi(X) is rejected, else wi(Xi) is accepted, and the version Xi with range(Xi) = [wts, rts], with wts = rts = ts(Ti) is created.

Multiversion timestamp: example

Suppose that the current version of A is A0, with rts(A0)=0.

T1(ts=1)	T2(ts=2)	T3(ts=3)	T4(ts=4)	T5(ts=5)	
r1(A)					reads A0, and set rts(A0)=1
w1(A)					writes the new version A1
	r2(A)				reads A1, and set rts(A1)=2
	w2(A)				writes the new version A2
			r4(A)		reads A2, and set rts(A2)=4
				r5(A)	reads A2, and set rts(A2)=5
		w3(A)			rollback T3

The final picture



1 - Transaction management

1.1 Transactions, concurrency, serializability
1.2 Recoverability
1.3 Concurrency control through locks
1.4 Concurrency control through timestamps
1.5 Transaction management in SQL

Transaction management in SQL

- SQL-92 has constructs for defining transactions and concurrency levels
- A single SELECT statement is considered as an atomic execution unit
- SQL does not have an explicit BEGIN TRANSACTION statement
- In SQL, every transaction must have an explicit termination statement (COMMIT or ROLLBACK)

Example

```
EXEC SQL WHENEVER sqlerror GO TO ESCI;
        EXEC SQL SET TRANSACTION
          READ WRITE , DIAGNOSTICS SIZE 8,
          ISOLATION LEVEL SERIALIZABLE;
        EXEC SQL INSERT INTO
          EMPLOYEE (Name, ID, Address)
          VALUES (‘John Doe',1234,‘xyz');
        EXEC SQL UPDATE EMPLOYEE
          SET Address = ‘abc'
          WHERE ID = 1000;
        EXEC SQL COMMIT;
        GOTO FINE;
ESCI:   EXEC SQL ROLLBACK;
FINE:   …
```

Ghost read

- Since SQL considers a whole query as an atomic execution unit, we must consider a further anomaly, the so-called ghost read
- Example:
  - T1 executes query SELECT * FROM R
  - T2 adds a record r to relation R
  - T1 executes the previous query again: the result of the second query contains record r, which was not in the first result
- The ghost read anomaly is a generalized version of the unrepeatable read anomaly, in which the read operation retrieves a set of records instead of a single one

SET TRANSACTION

- The SET TRANSACTION statement allows for defining the following aspects:
- Access mode: READ ONLY or READ WRITE
- Isolation level of the transaction: can assume one of the following values:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE (default value)
- Configuration of the number of error conditions that can be handled (DIAGNOSTIC SIZE n)

Isolation levels

(We assume that the SET TRANSACTION statement is relative to transaction Ti)

- SERIALIZABLE:
  - Transaction Ti only reads from committed transactions
  - No value read or written by transaction Ti can be modified until Ti commits
  - The set of records read by Ti through a query cannot be modified by other transactions until Ti commits (this condition avoids the ghost read anomaly)

Isolation levels

- REPEATABLE READ:
  - Transaction Ti only reads from committed transactions
  - No value read or written by transaction Ti can be modified until Ti commits
  - The set of records read by Ti through a query **can** be modified by other transactions until Ti commits (the ghost read anomaly is thus possible)

Isolation levels

- READ COMMITTED:
  - Transaction Ti only reads from committed transactions
  - No value written by transaction Ti can be modified until Ti commits, while values read by Ti can be modified by other transactions (thus, both the ghost read anomaly and the unrepeatable read anomaly are possible)

Isolation levels

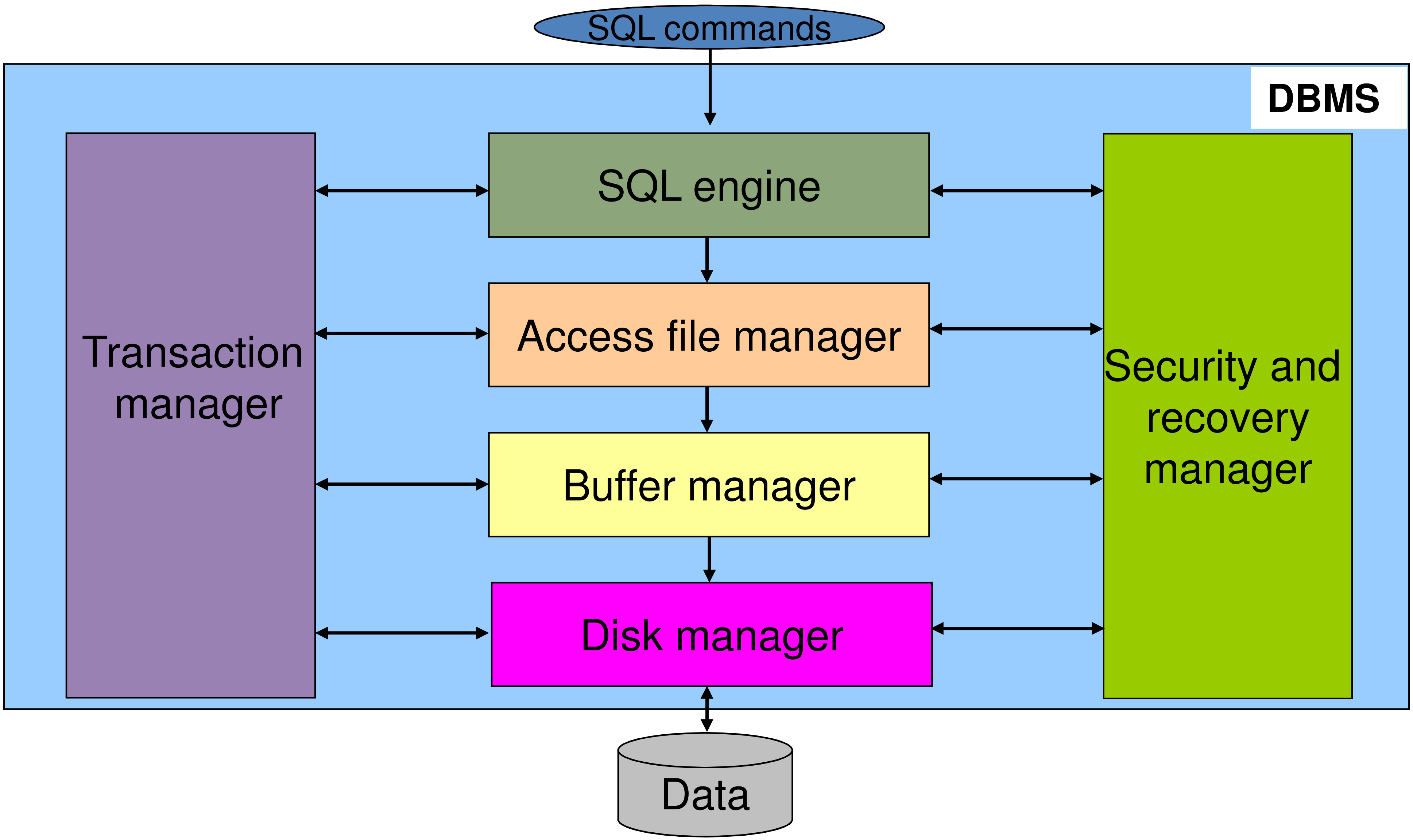
- READ UNCOMMITTED:
  - Transaction Ti can read from any (even uncommitted) transaction (cascading rollback is thus possible)
  - Values both read and written by Ti can be modified by other transactions (thus, besides ghost read and unrepeatable read, also the dirty read anomaly is possible)

Transaction management in commercial systems

- The transaction managers of the main commercial systems (Oracle, DB2, SQL Server, PostgreSQL) use schedulers based on lock and/or (multiversion) timestamp methods
- In such systems, the scheduler usually distinguishes between two classes of transactions:
  - The transactions with read and write are executed under the 2PL protocol
  - The transactions that are “read only” are executed under the method of multiversion timestamp

2. Recovery management

Architecture of a DBMS



The recovery manager

The transaction manager is mainly concerned with isolation and consistency, while the recovery manager is mainly concerned with atomicity and persistency.

It is responsible for:

- Beginning the execution of transactions
- Committing transactions
- Executing the rollback of transactions
- Restore a correct state of the database following a fault condition

It uses a special data structure, called log file

Failure types

- System failures
  - System crash:
    - We loose the buffer content, not the secondary storage content
  - System error or application exception
    - E.g. division by zero
  - Local error conditions of a transaction
  - Concurrency control
    - The scheduler forces the rollback of a transaction
- Storage media failures
  - Disk failures
    - We loose secondary storage content, but not the log file content
  - Catastrophic events”
    - Fire
    - Flooding
    - Etc…

The strategies depend on the failures

- **System failures:**
  - Information loss in the buffer, not in the data
  - Main risk for → **Atomicity**
  - Recovery strategy:
    - Periodically register the system status (**checkpoint**)
    - Analyze back the DB change history
    - Undo and redo some operations
    - Using the **log**

- **Media failure:**
  - Information loss in th data
  - Main risk for → **Durability**
  - Recovery strategy :
    - Load the most recent available DB back-up
    - Reconstruct that state using the log, starting the **dump**

The log file

- The log file (or, simply, the log) records the actions of the various transactions in a stable storage (stable means “failure resistant”)
- Read and write operations on the log are executed as the operations on the database, i.e., through the buffer. Note that writing on the stable storage is generally done through “force”
- The stable storage is an abstraction: stability is achieved through replication
- The physical organization of the log can be based on:
  - Tapes
  - Disk (perhaps coupled with tapes)
  - Two replicated disks

The structure of log

- The log is a sequential file (assumed to be failure-free). The operations on the log are: append a record at the end, scan the file sequentially forward, scan backward
- The log records the actions of the transactions, in chronological order.
- Two types of records in the log:
  - Transaction records (begin, insert, delete, update, commit, abort)
  - System records (checkpoint, dump)
- Please, do not confuse the transaction actions with the actions on the secondary storage. In particular, the actions of the transactions are assumed to be executed on the DB when they are recorded in the log (even if their effects are not registered yet in the secondary storage)

The transaction records

O = element of the DB

AS = After State, value of O after the operation

BS = Before State, value of O before the operation

For each transaction T, the transaction records are stored in the log as follows:

– `begin`:	B(T)
– `insert`:	I(T,O,AS)
– `delete`:	D(T,O,BS)
– `update`:	U(T,O,BS,AS)
– `commit`:	C(T)
– `abort`:	A(T)

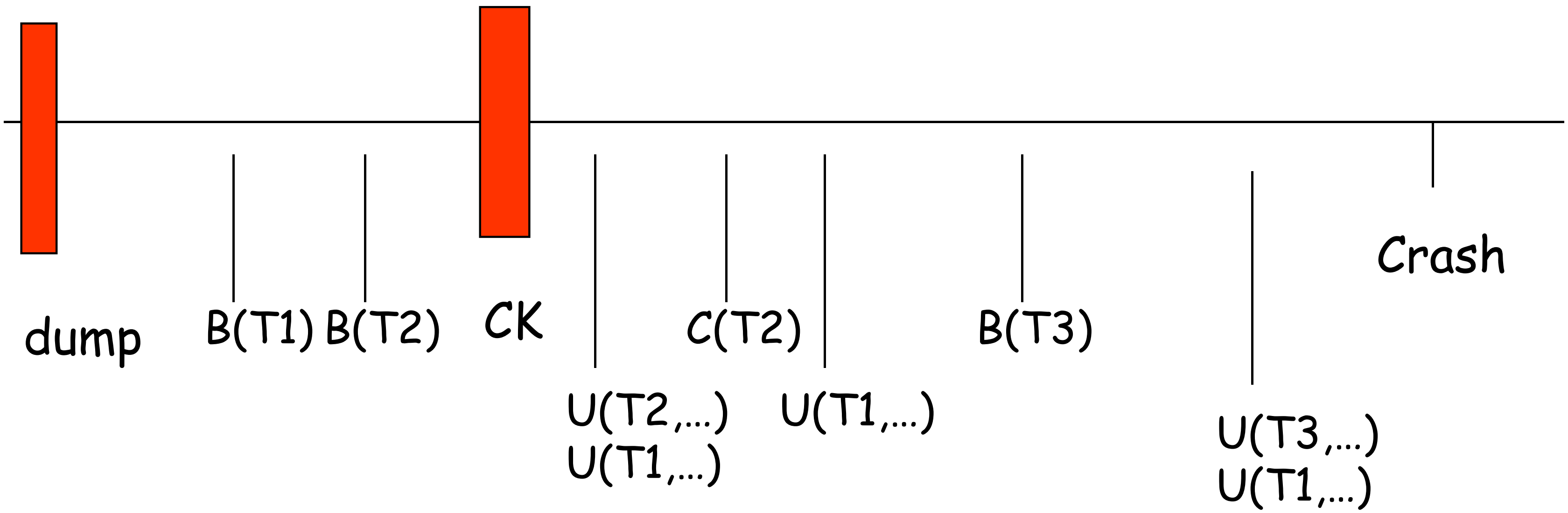
Checkpoint

- The goal of the checkpoint is to register in the log the set of active transactions T1, …, Tn so as to differentiate them from the committed transactions
- The checkpoint (CK) operation executes the following actions:
  - For each committed transaction after the last checkpoint, their buffer pages are copied into the secondary storage (through flush)
  - A record CK(T1,…Tn) is written on the log (through force), where T1,…Tn identify all active transactions that are uncommitted
- It follows that:
  - For each transaction T such that Commit(T) *precedes* CK(T1,…Tn) in the log, we can avoid the "redo" in case of failure
- The checkpoint operation is executed periodically, with fixed frequency

Dump

- The dump is a copy of the entire state of the DB
- The dump operation is executed offline (all transactions are suspended)
- It produces a backup, i.e., the DB is saved in stable storage
- It writes (through force) a dump record in the log

Example: log with checkpoint and dump



The Undo operation

- Restore the state of an element O at the time preceding the execution of an action

- `update`, `delete`*:*
  - assigns the *BS value to* O

- `insert:`
  - delete O

The Redo operation

- Restore the state of an element O at the time *following* the execution of an action
- `insert, update`:
  - assigns the value AS to O
- `delete`:
  - delete O

Atomicity of transactions

- The outcome of a transaction is established when either the Commit(T) record or the Abort(T) record is written in the log
  - The Commit(T) record is written synchronously (force) from the buffer to the log
  - The Abort(T) record is written asynchronously (flush) from the buffer to the log (the recovery manager does not need to know immediately that a transaction is aborted)

- When a failure occurs, for a transaction
  - Uncommitted: since atomicity has to be ensured, in general we may need to undo the actions, especially if there is the possibility that the actions have been executed on the secondary storage → Undo
  - Committed: we need to redo the actions, to ensure durability → Redo

Writing records in the log

The recovery manager follows this rule:

- ***WAL*** **(write-ahead log)**
  - The log records are written from the buffer to the log before the corresponding records are written in the secondary storage
  - This is important for the effectiveness of the Undo operation, because the old value can always be written back to the secondary storage by using the BS value written in the log. In other words, WAL allows to undo write operations executed by uncommitted transactions

Writing records in the log

The recovery manager follows this rule:

- ***Commit-Precedence***
  - The log records are written from the buffer to the log before the commit of the transaction (and therefore before writing the commit record of the transaction in the log)
  - This is important for the effectiveness of the Redo operation, because if a transaction committed before a failure, but its pages have not been written yet in secondary storage, we can use the AS value in the log to write such pages. In other words, the Commit-Precedence rule allows committed transactions whose effects have not been registered yet in the database to be redone.

Writing in secondary storage

For each operation

- Update
- Insert
- Delete

The recovery manager must decide on the strategy for writing in secondary storage

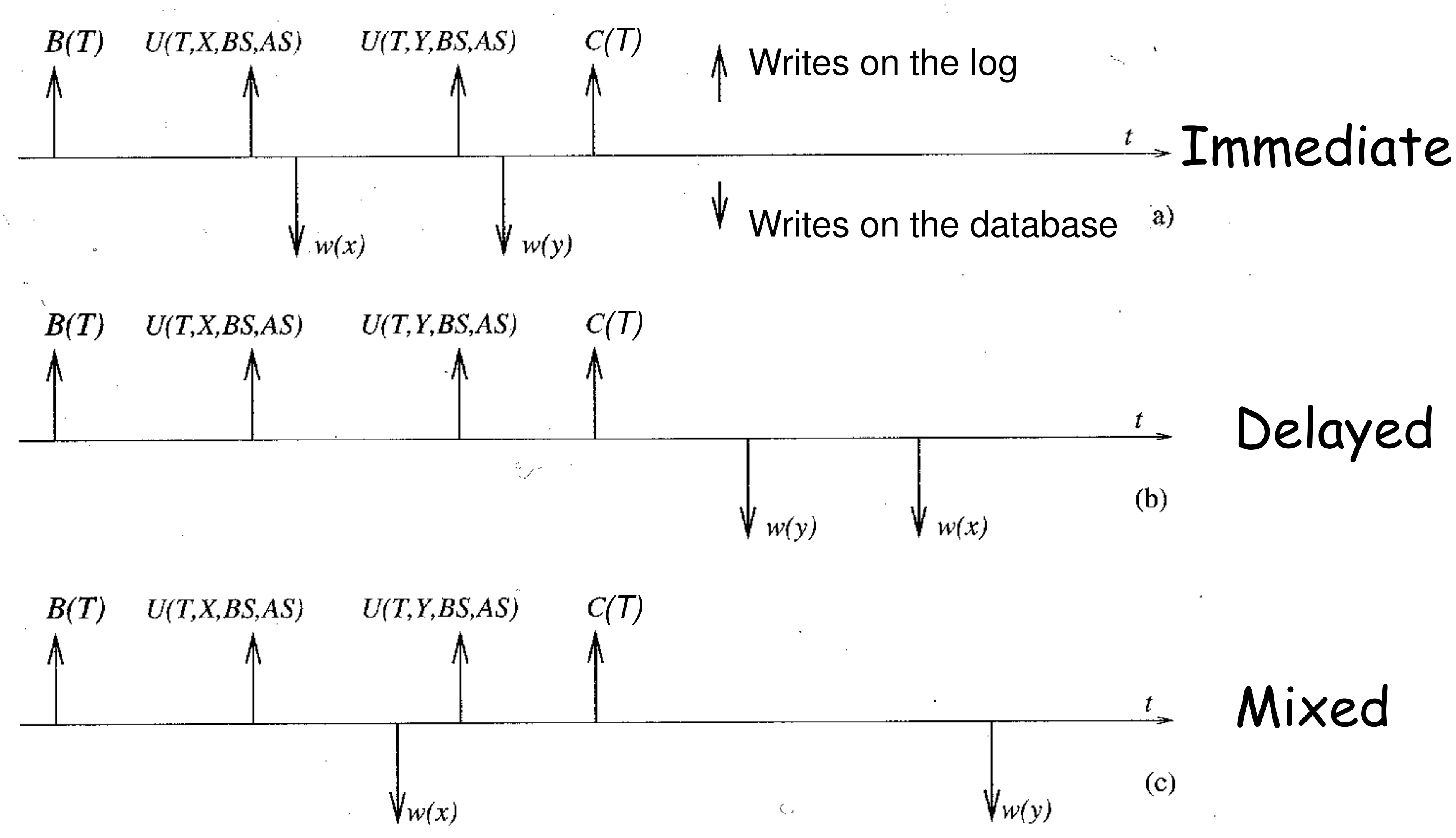
In the following, we concentrate on update, but similar considerations hold for the other operations

Writing in secondary storage

There are three possible methods for writing values into the secondary storage, all coherent with the WAL and the commit-precedence rules

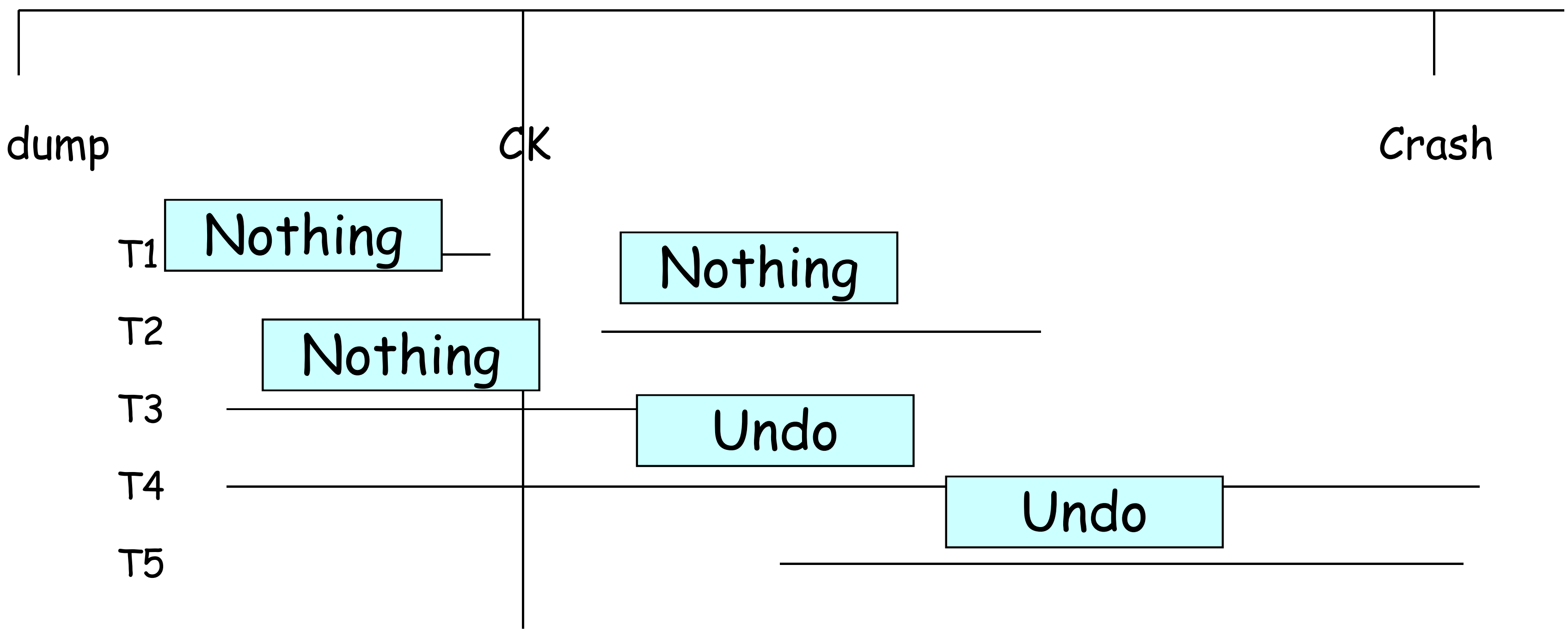
- Immediate effect
  - The update operations are executed immediately on the secondary storage after the corresponding records are written in the log
  - The buffer manager writes (either with force or flush) the effect of an operation by a transaction T on the secondary storage before writing the commit record of T in log
  - It follows that all the pages of the DB modified by a transaction are certainly written in the secondary storage
- Delayed effect
  - The update operations by a transaction T are executed on the secondary storage only after the commit of the transaction, i.e., only after the commit record of T has been written in the log
  - As usual, the log records are written in the log before the corresponding data are written in secondary storage
- Mixed effect
  - For an operation O, both the immediate effect and and the delayed effect are possible, depending on the choice of the buffer manager

Examples


B(T)
U(T,X,BS,AS)
U(T,Y,BS,AS)
C(T)
Writes on the log
t
Immediate
Writes on the database
a)
w(x)
w(y)
B(T)
U(T,X,BS,AS)
U(T,Y,BS,AS)
C(T)
t
Delayed
(b)
w(y)
w(x)
B(T)
U(T,X,BS,AS)
U(T,Y,BS,AS)
C(T)
t
Mixed
(c)
w(x)
w(y)

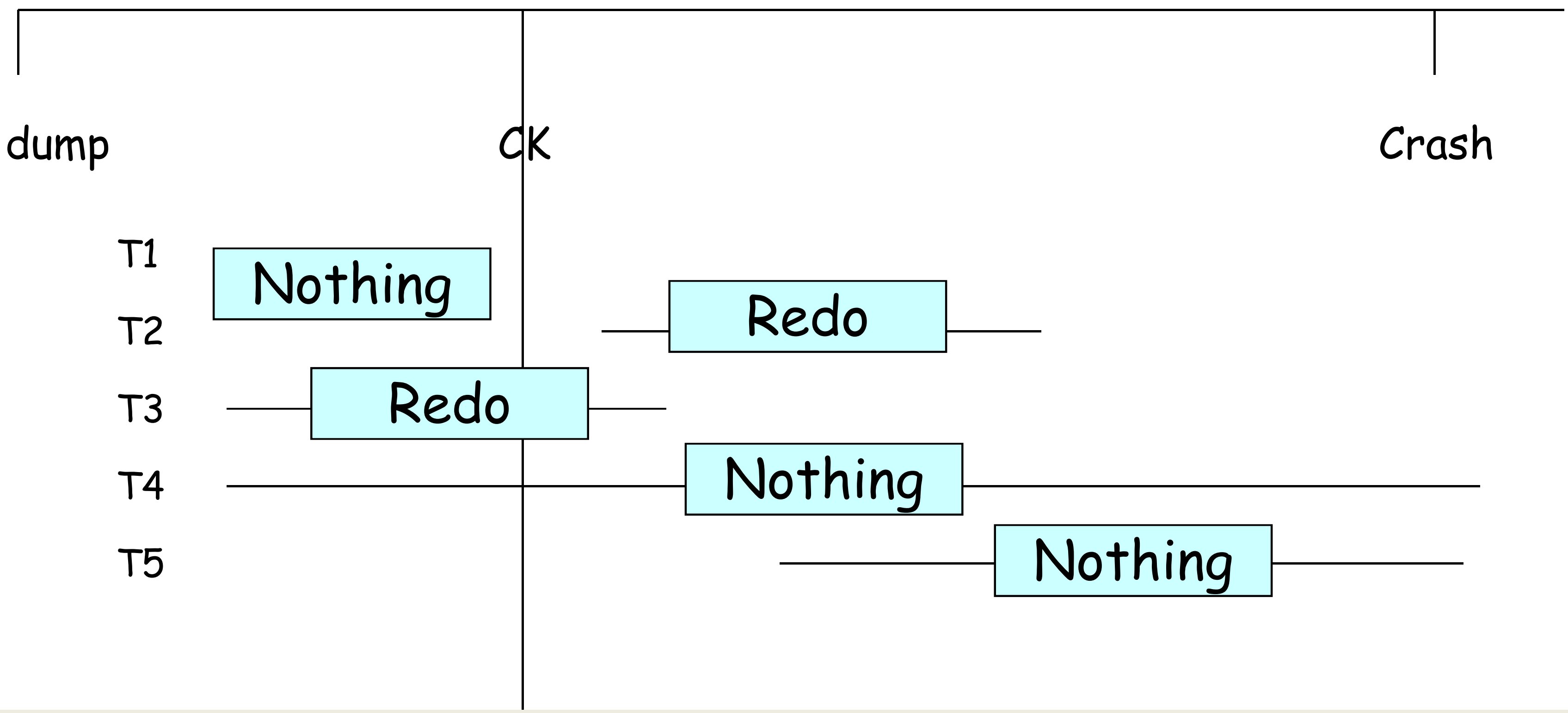
Immediate effect

- The secondary storage may contain AS values from uncomitted transactions
- Undo of transactions that are uncomitted when the failure occurs is needed
- Redo is not needed (if the commit record of T is in the log, all pages of T have been written in secondary storage)



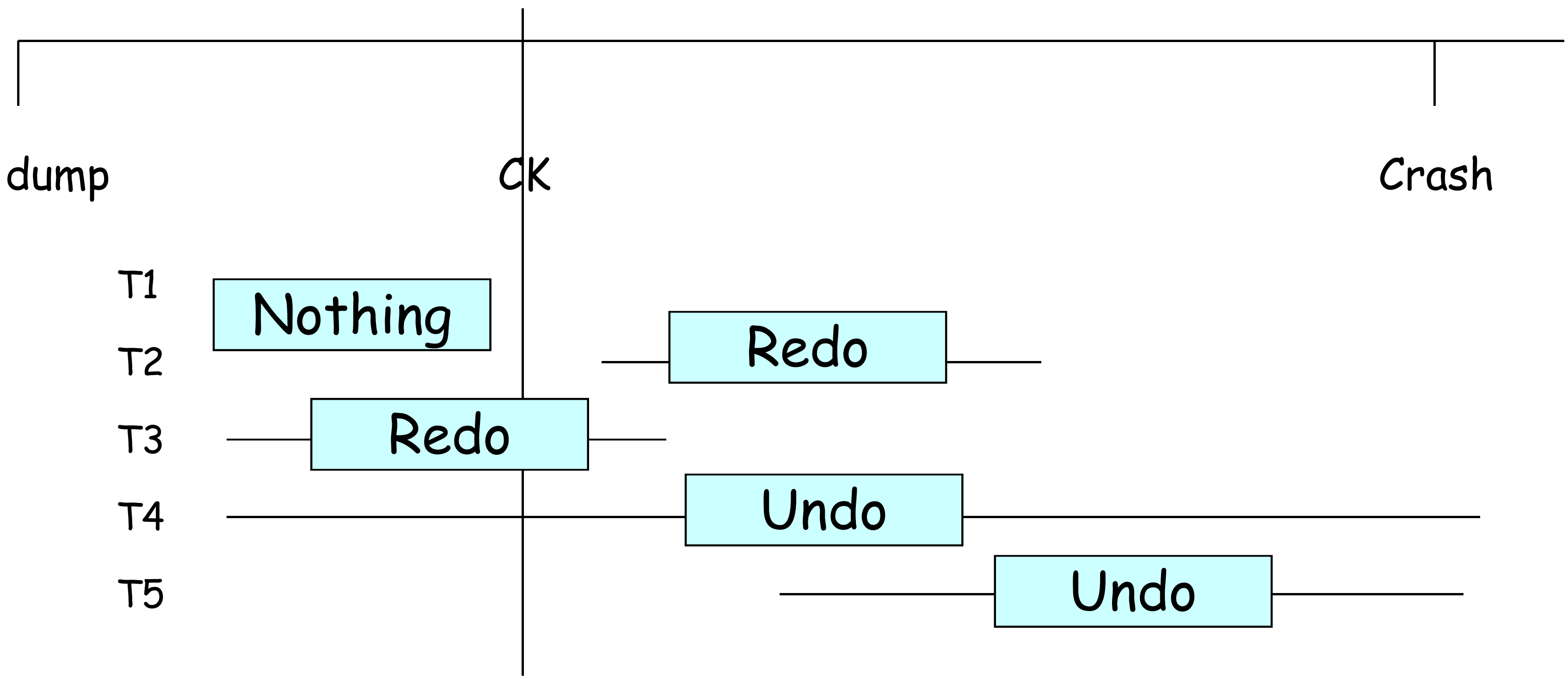
Delayed effect

- The secondary storage does not contain AS values from uncomitted transactions
- Undo is not needed (when we rollback a transaction T, nothing has been done by T on the secondary storage)
- Redo is needed



Mixed effect

- The buffer manager decides its strategy for each of the operation (for this reason, this is the most used method). In particular, this strategy allows to to optimize the execution of the flush operation
- Both Undo and Redo are needed



Two types of recovery

Depending on the type of failure…

- In case of system failure:
  - Warm restart
- In case of disk failure:
  - Cold restart

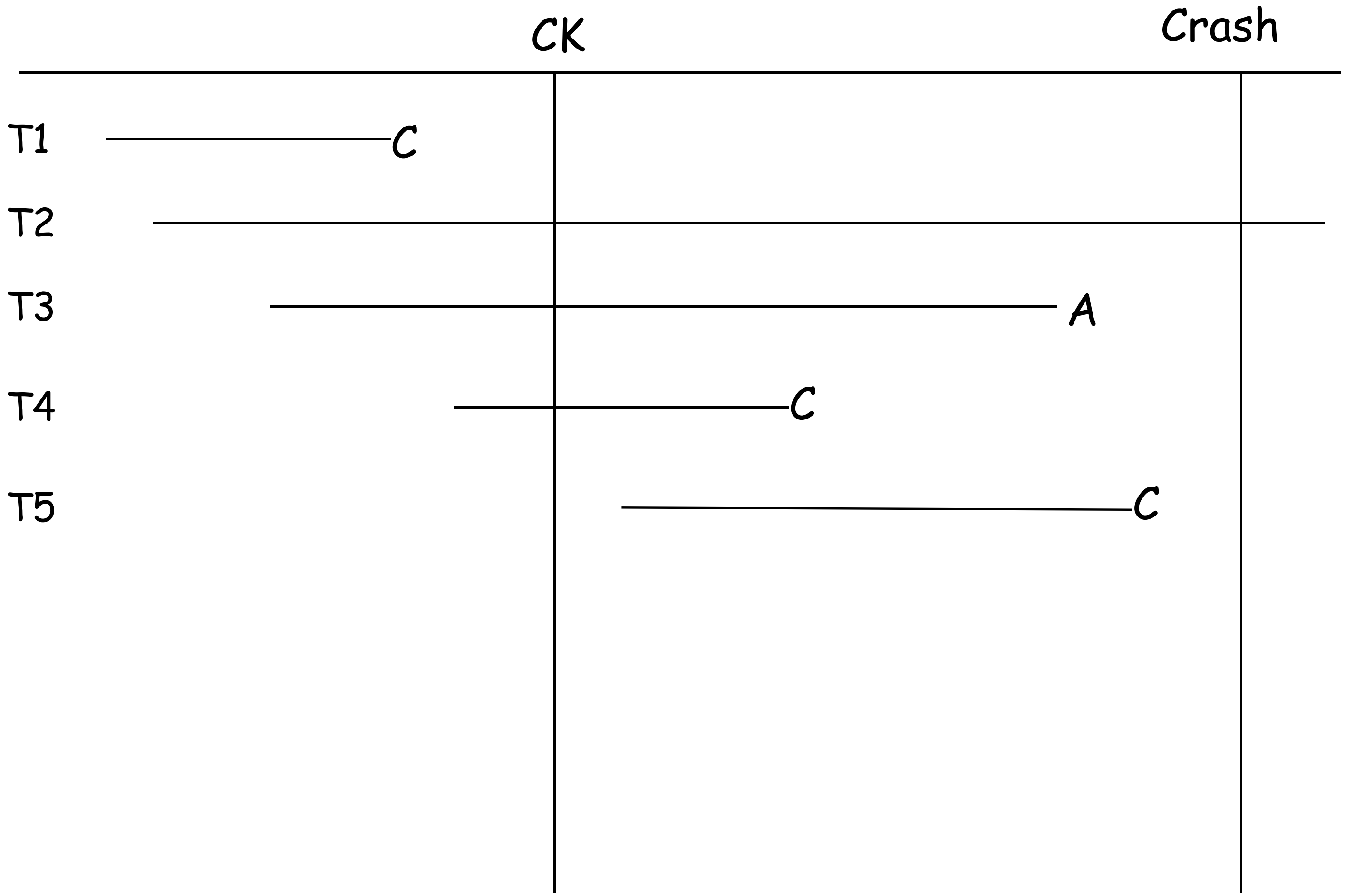
Warm restart

We will assume the mixed effect strategy. The warm restart is constituted by 5 steps:

1. We go backward through the log until the most recent checkpoint record in the log
2. We set s(UNDO) = { active transactions at checkpoint } s(REDO) = { }
3. We go forward through the log adding to s(UNDO) the transactions with the corresponding begin record, and moving those with the commit record to s(REDO)
4. Undo phase: we go backward through the log again, undoing the transactions in s(Undo) until the begin record of the oldest transaction in the set of active transactions at the last checkpoint (note that we may even go before the most recent checkpoint record)
5. Redo phase: we go forward through the log again, redoing the transactions in s(Redo)

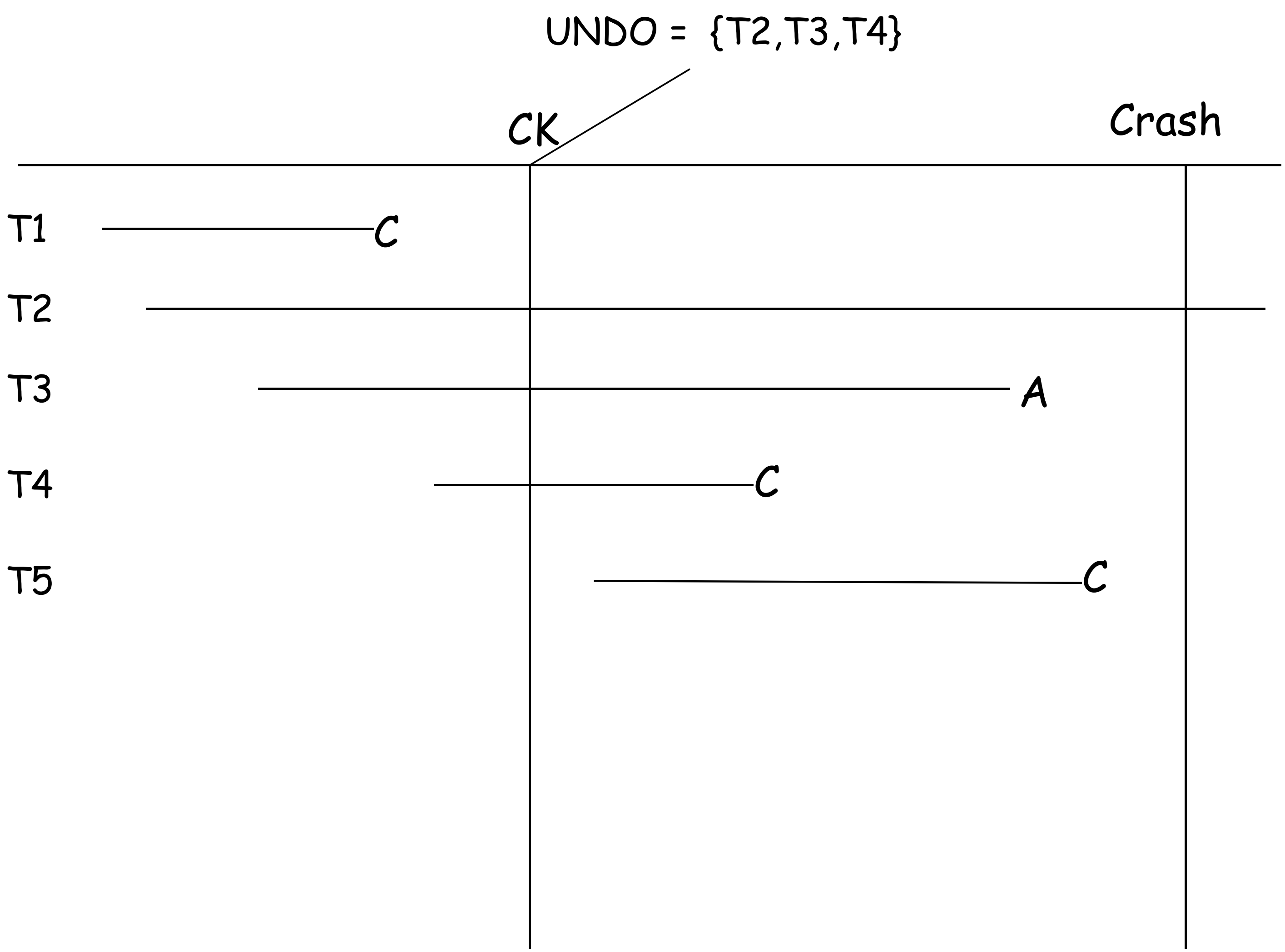
Warm restart: example

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(T2,O6,A8)



Example: the most recent checkpoint

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(T2,O6,A8)



Example: s(UNDO) and s(REDO)

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)

1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Example: the UNDO phase

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)

1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

4. D(O6)

5. O5 =B7

6. O3 = B5

7. O2 =B3

8. O1=B1

Undo phase

Example: the REDO phase

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)

1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Undo phase

4. D(O6)

5. O5 =B7

6. O3 = B5

7. O2 =B3

8. O1=B1

Redo phase

9. O3 = A4

10. O4 = A6

Cold restart

It is constituted by three phases:

1. Search for the most recent dump record in the log, and load the dump into the secondary storage (more precisely, we selectively copy the fragments of the DB that have been damaged by the disk failure)
2. Forward recovery of the dump state:
   1. We re-apply all actions in the log, in the order determined by the log
   2. At this point, we have the database state immediately before the crash
3. We execute the warm restart procedure

Exercise: cold restart

- Consider the following log: DUMP, B(T1), B(T2), B(T3), I(T1,O1,A1), D(T2,O2,B2), B(T4), U(T4,O3,B3,A3), U(T1,O4,B4,A4), C(T2), CK(T1,T3, T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3), CK(T1,T4,T5,T6), B(T7), A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), B(T8), C(T7)

- Suppose that a disk failure occurs. Assume the mixed strategy.

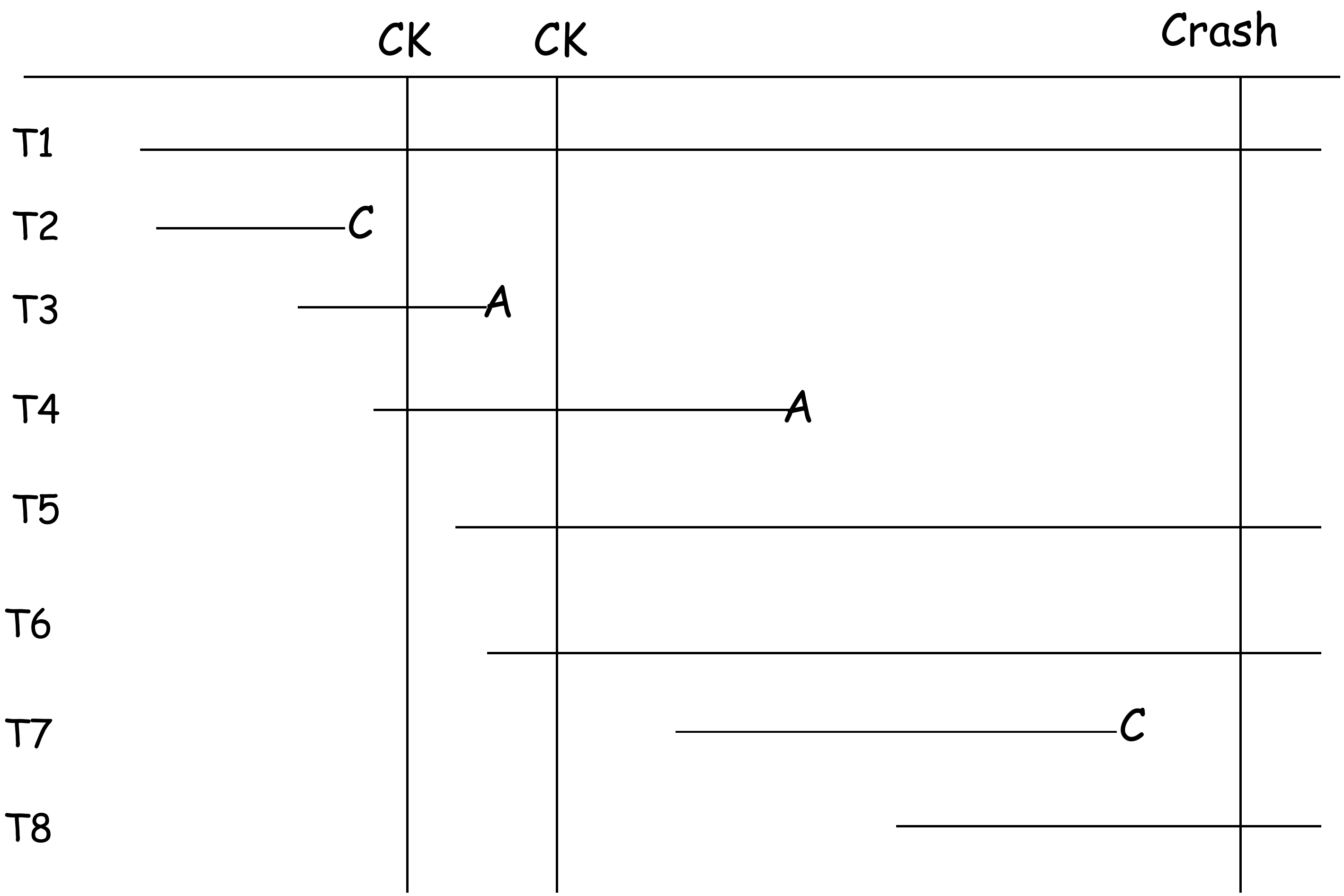
Solution: reconstruct the DB from DUMP

- DUMP, B(T1), B(T2), B(T3), I(T1,O1,A1), D(T2,O2,B2), B(T4), U(T4,O3,B3,A3), U(T1,O4,B4,A4), C(T2), CK(T1,T3, T4), B(T5), B(T6), U(T5,O5,B5,A5), A(T3), CK(T1,T4,T5,T6), B(T7), A(T4), U(T7,O6,B6,A6), U(T6,O3,B7,A7), B(T8), C(T7)

- We go to the most recent dump record in the log (the first record), and load the dump into the secondary storage
- We scan the log forward starting from B(T1), and we execute all actions in the log, until C(T7)
- We execute the warm restart procedure

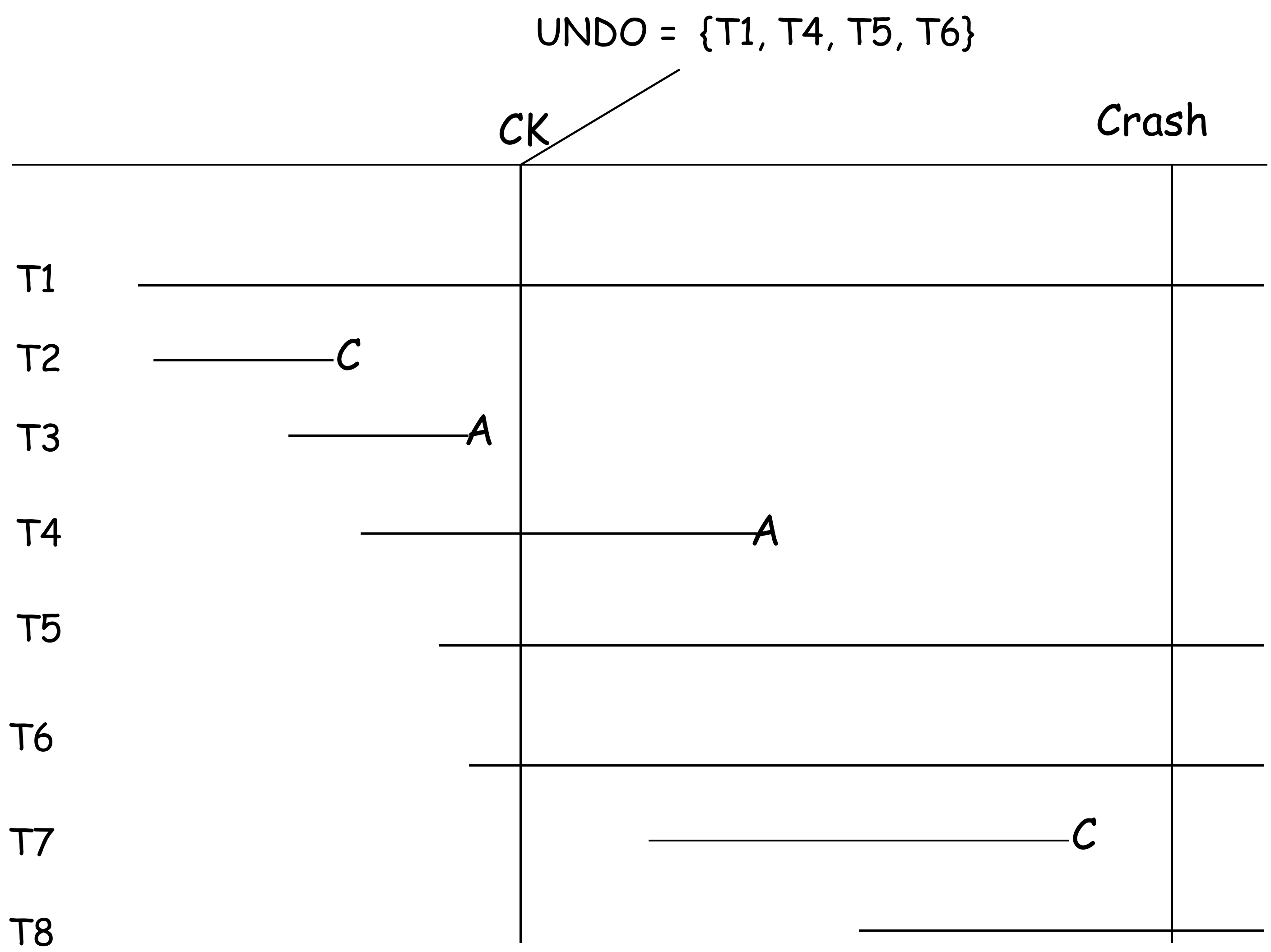
Solution: warm restart

B(T1),
B(T2),
B(T3),
I(T1,O1,A1),
D(T2,O2,B2),
B(T4),
U(T4,O3,B3,A3),
U(T1,O4,B4,A4),
C(T2),
CK(T1,T3, T4),
B(T5),
B(T6),
U(T5,O5,B5,A5),
A(T3),
CK(T1,T4,T5,T6),
B(T7),
A(T4),
U(T7,O6,B6,A6),
U(T6,O3,B7,A7),
B(T8),
C(T7)



Solution: most recent checkpoint

B(T1),
B(T2),
B(T3),
I(T1,O1,A1),
D(T2,O2,B2),
B(T4),
U(T4,O3,B3,A3),
U(T1,O4,B4,A4),
C(T2),
CK(T1,T3, T4),
B(T5),
B(T6),
U(T5,O5,B5,A5),
A(T3),
CK(T1,T4,T5,T6),
B(T7),
A(T4),
U(T7,O6,B6,A6),
U(T6,O3,B7,A7),
B(T8),
C(T7)



Solution: the UNDO and REDO sets

B(T1),
B(T2),
B(T3),
I(T1,O1,A1),
D(T2,O2,B2),
B(T4),
U(T4,O3,B3,A3),
U(T1,O4,B4,A4),
C(T2),
CK(T1,T3, T4),
B(T5),
B(T6),
U(T5,O5,B5,A5),
A(T3),
CK(T1,T4,T5,T6),
B(T7),
A(T4),
U(T7,O6,B6,A6),
U(T6,O3,B7,A7),
B(T8),
C(T7)

0. UNDO = {T1, T4, T5, T6}. REDO = {}

---

1. B(T7) → {T1, T4, T5, T6, T7}. REDO = {}

2. B(T8) → {T1, T4, T5, T6, T7, T8}. REDO = {}

3. C(T7) → {T1, T4, T5, T6, T8}. REDO = {T7}

Solution: the UNDO phase

B(T1),
B(T2),
B(T3),
I(T1,O1,A1),
D(T2,O2,B2),
B(T4),
U(T4,O3,B3,A3),
U(T1,O4,B4,A4),
C(T2),
CK(T1,T3, T4),
B(T5),
B(T6),
U(T5,O5,B5,A5),
A(T3),
CK(T1,T4,T5,T6),
B(T7),
A(T4),
U(T7,O6,B6,A6),
U(T6,O3,B7,A7),
B(T8),
C(T7)

0. UNDO = {T1, T4, T5, T6}. REDO = {}

---

1. B(T7) → {T1, T4, T5, T6, T7}. REDO = {}

2. B(T8) → {T1, T4, T5, T6, T7, T8}. REDO = {}

3. C(T7) → {T1, T4, T5, T6, T8}. REDO = {T7}

---

4. O3 = B7

5. O5 = B5

6. O4 = B4

7. O3 = B3

8. D(O1)

Undo phase

Solution: the REDO phase

B(T1),
B(T2),
B(T3),
I(T1,O1,A1),
D(T2,O2,B2),
B(T4),
U(T4,O3,B3,A3),
U(T1,O4,B4,A4),
C(T2),
CK(T1,T3, T4),
B(T5),
B(T6),
U(T5,O5,B5,A5),
A(T3),
CK(T1,T4,T5,T6),
B(T7),
A(T4),
U(T7,O6,B6,A6),
U(T6,O3,B7,A7),
B(T8),
C(T7)

0. UNDO = {T1, T4, T5, T6}. REDO = {}

---

1. B(T7) → {T1, T4, T5, T6, T7}. REDO = {}
2. B(T8) → {T1, T4, T5, T6, T7, T8}. REDO = {}
3. C(T5) → {T1, T4, T5, T6, T8}. REDO = {T7}

---

4. O3 = B7
5. O5 = B5
6. O4 = B4
7. O3 = B3
8. D(O1)

Undo phase

---

9. O6 = A6

Redo phase