Data Management

Maurizio Lenzerini

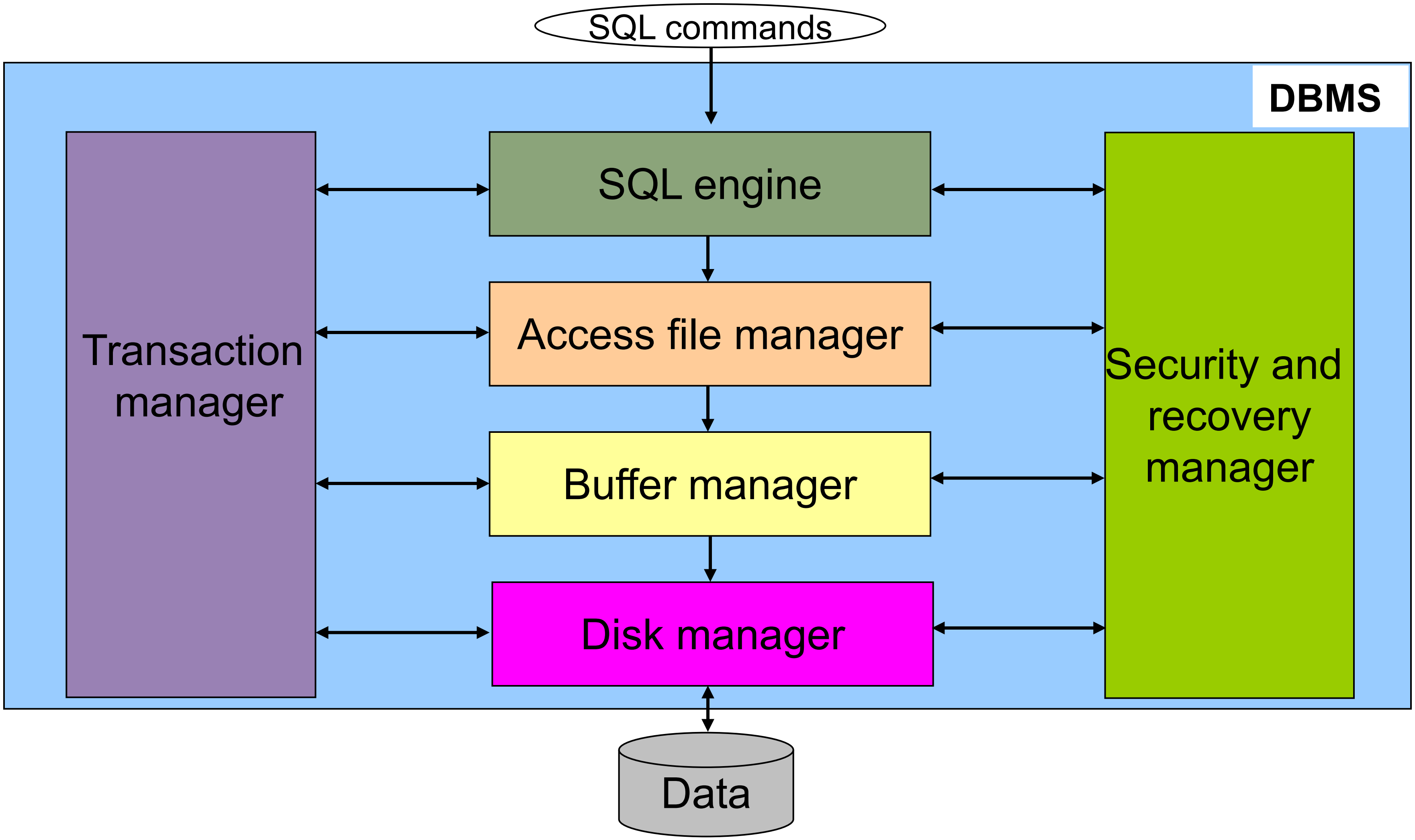
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università di Roma "La Sapienza"

Academic Year 2024/2025

Part 5
Transaction management and concurrency

http://www.dis.uniroma1.it/~lenzerin/index.html/?q=node/53

Architecture of a DBMS



5. Transaction management and concurrency

5.0 The buffer manager
5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

5.0 The buffer manager

The secondary storage

At the physical level, a data base is a set of database files, where each file is constituted by a set of pages, stored in physical blocks.

Using a page requires to bring it in main memory. The size of a block (and therefore of a page) is exactly the size of the portion of storage that can be transferred from secondary storage to main memory, and back from main memory to the secondary storage.

The buffer

The database buffer (also called simply buffer or buffer pool) is a non-persistent main memory space used to cache database pages.

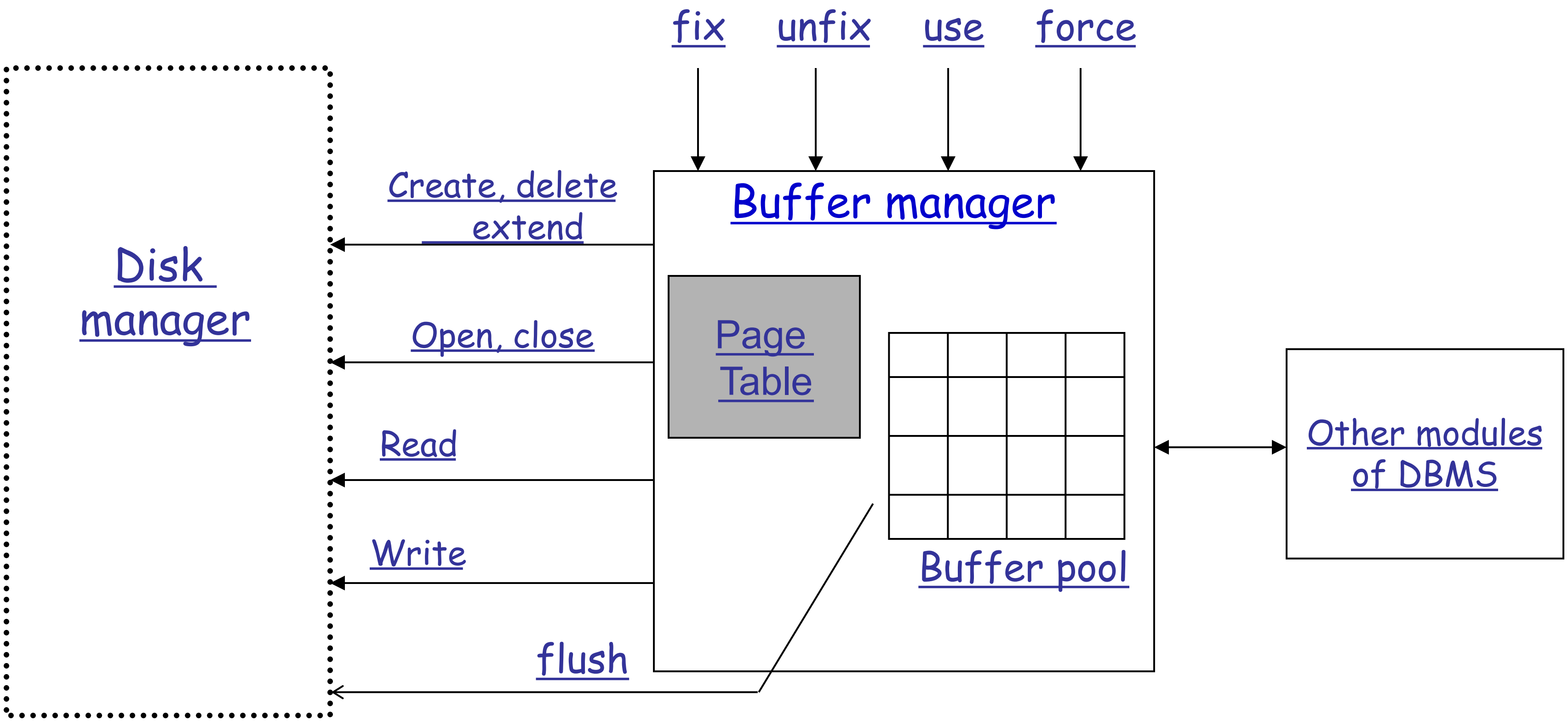
The database processes request pages from the buffer manager, whose responsibility is to minimize the number of secondary memory accesses by keeping needed pages in the buffer. Because typical database workloads are I/O-bound, the effectiveness of buffer management is critical for system performance.

The buffer manager is responsible of the transfer of the pages from the secondary storage to the buffer pool, and back from the buffer pool to the secondary storage.

The buffer pool is

- Shared by all transactions (i.e., all programs using the databases managed by the DBMS)
- Used by the system (e.g., by the recovery manager)

Architecture of buffer manager



The buffer pool

- The buffer pool is organized in “frames”. Each frame has an identifier (a number), corresponds to one main memory page, whose size is that of a block, where, as we said, a block is the transfer unit from/to the secondary storage. The typical size ranges from 2Kb to 64Kb.
- Since the buffer pool is in main memory, the management of their pages is more efficient (the cost of atomic operations is of the order of billionth of seconds) with respect to the cost of the management of secondary storage pages (thousandth of seconds)
- The buffer pool is managed with similar principles as the ones used for cache memory.

The buffer manager

The buffer manager uses the “Page Table” data structure, that associates to each frame in the buffer the last secondary storage page (denoted by its Page Identifier (PID)), if any, loaded in the frame. In some sense, this frame is the “main memory twin” of such secondary storage page.

The buffer manager uses the following primitive operations:

- Fix: load a page
- Unfix: releases a page
- Use: registers the use of a page in the buffer
- Force: synchronous transfer to secondary storage
- Flush: asynchronous transfer to secondary storage

The Fix operation

- An external module (transaction) issues the ”Fix” operation in order to ask the buffer manager to load a specific secondary storage page into the buffer

- For each frame F in the buffer, the buffer manager maintains:
  - As said, the information about which page (if any) it contains (this information is provided by the Page Table)
  - pin-count(F): how many transactions are using the page contained in F (initially, set to 0).
  - dirty(F): a bit whose value indicates whether the content of the frame F has been modified (true) or not (false) from the last load (initially, set to false).

The Fix operation

Suppose that the DBMS module M issues a Fix operation that asks the buffer manager to load page P (denoted by its PID) in the buffer for transaction T.

1. Looking at the Page Table, the buffer manager checks whether there is a frame F in the buffer pool already containing P
2. If yes, then the buffer manager increments pin-count(F)
3. If not, then
    1. using a certain replacement policy, the buffer manager chooses a frame F’ (if possible) that can host page P in the buffer,
    2. If dirty(F’)=true, then the buffer manager writes the content of frame F’ back to the appropriate page in secondary storage in order not to loose its content
    3. The buffer manager reads the content of page P from the secondary storage, loads it in frame F’ and initializes pin-count(F’) to 0 and dirty(F’) to false
4. The buffer manager returns the address of the frame containing P to M, or NULL if such a frame does not exist

Replacement policy

The frame for the replacement (the “victim”) is chosen among those frames F with pin-count(F) = 0.

If no frame F with pin-count(F)=0 exists, then the request is placed in a queue, or the transaction is aborted (and later re-executed).

Otherwise, several policies are possible in order to choose the victim:

– LRU (least recently used): this is done through a queue containing the frames F with pin-count(F)=0
– Clock replacement
– Other strategies

Force or No-force strategy

Force/no-force:

- Force: all active pages of a transaction are written in secondary storage when the transaction commits (i.e., it ends its operations successfully)
- No-force: the active pages of a transaction that has committed are written asynchronously in secondary storage through the flush operation

Generally, the no-force is the one used, because it enables a more efficient buffer management

The other operations

- Unfix:
  – Transaction T certifies that it does not need the content of a specific frame anymore
  – The pin-counter of that frame is decremented

- Use:
  – The transaction modifies the content of a frame
  – The dirty bit of that frame is set to true

- Force: Synchronous (i.e., the transactions waits for the successful completion of the operation) transfer to secondary storage of the page contained in a frame

- Flush: Asynchronous (i.e., executed when the buffer manager is not busy) transfer to secondary storage of the pages used by a transaction

5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability

5.2 View-serializability

5.3 Conflict-serializability

5.4 Concurrency control through locks

5.5 Recoverability of transactions

5.6 Concurrency control through timestamps

5.7 Multiversion concurrency control

5.8 Optimistic concurrency control

5.9 Concurrency control in SQL

Transactions

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

Syntactically, we will often assume that every transaction contains:

- one “begin” instruction
- one “end” instruction
- one among “commit” (confirm what you have done on the database so far, sometime equivalent to “end”) or “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)

Concurrency

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

Taking into account the requirements of real applications, in a DBMS, we want the throughput to be approximately 1000-10.000tps. 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, web commerce…
For example: Amazon is estimated to support hundreds of thousands ACID transaction per second.

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 ↓

Since “Find seat” in the two transactions find the same seat, the result is that we have two reservations for the same seat! ERROR!

Isolation of transactions

One desirable way for the DBMS to deal with this problem is by ensuring the so-called “isolation” property for the transactions

This property for a transaction T essentially means that T is executed like it was the only one in the system, i.e., without concurrent transactions. This means that, even if concurrency exists, it is harmless, because no user will notice that a concurrent execution took place.

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 have their effect
2. Consistency: each transaction execution brings the database to a correct state (as state where no integrity constraint is violated)
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

Desirable properties of transactions

From now on, we will assume that every single transaction enjoys the ACID properties.

PROBLEM

Even if every single transaction enjoys the ACID properties, how can we be sure that the concurrent (i.e., interleaved) execution of a set of transactions behaves correctly?

This is exactly the problem of concurrency control.

Schedules and serial schedules

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

A sequence of actions of transactions {T1,T2,…,Tn} that is a prefix of a schedule on {T1,T2,…,Tn} is called a partial schedule. A schedule is also called total (or complete), to distinguish it from a partial schedule.

A (total) 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.

Examples

T1: a1,a2,a3

T2: b1,b2

T3: c1,c2,c3,c4

S1: a1, a2, b1, c1, c2, b2, c3, a3, c4

S2: a1, a2, c3, c4, b1, c1, c2, b2, a3

S3: b1, a1, b2, a2, a3, c1, c2

S4: c1, c2, c3, c4, a1, a2, a3, b1, b2

S1 is a (total) schedule

S2 is not a schedule (the actions of T3 is S2 are not ordered correctly)

S3 is a partial schedule

S4 is a serial schedule

Serializability

Example of serial schedules:

Given T1 (x=x+x; x= x+3) and T2 (x= x**2; x=x+2), possible serial schedules on them are:

Sequence 1: $\overbrace{\texttt{x=x+x; x= x+3;}}^{T1}\ \overbrace{\texttt{x= x**2; x=x+2}}^{T2}$

Sequence 2: $\underbrace{\texttt{x= x**2; x=x+2;}}_{T2}\ \underbrace{\texttt{x=x+x; x= x+3}}_{T1}$

A serial schedule is obviously "correct" with respect to concurrency, because it does not have interleaving.

What about the correctness of a schedule having interleaving? Intuitively, we would like to say that a schedule S that is not serial is "correct" with respect to concurrency if it is "equivalent" to a serial schedule S' constituted by the same transactions of S.

Serializability

Definition of serializable schedule

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 from the database state D produces the same outcome as the execution of S2 starting from the same database state D.

Notice that, in general, when we talk about the "outcome" we talk about the final state of the process represented by the schedule, which incorporates both the state of the database, and the state of the local store.

Notation

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

- Commands of type begin/commit
- Actions that read an element (attribute, record, table) in the database and store the value in local store, or write a value from the local store to the database
- Actions that process elements in the local store (main memory)

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

action that reads the value of the DB element A and stores such value in the element s of the local store

action that processes the element s of the local store

A serial schedule

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

A serial schedule execution

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		

in the initial state, A=25 and B = 25

A serializable schedule S

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 in the execution of S starting from A=B=25 are the same as the execution of the serial schedule <T1,T2> starting from the same state

A serializable schedule S

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,t)			
t := t+100			
WRITE(B,t)			125
commit			
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

We can indeed show that, no matter what the value a1 of A and the value b1 of B in the initial state, both S and the serial schedule <T1,T2> leave the value (a1+100)*2 in A and the value (b1+100)*2 in B.

So, S and <T1,T2> are equivalent, and therefore S is serializable.

A non-serializable schedule S'

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		

The execution of *S'* starting with A=B=25 leaves the values 250 for *A* and 150 for B.

However, the execution of <T1,T2> leaves the value 125 for *A* and 125 for B, and the execution of <T2,T1> leaves that value 50 for *A* and 50 for B.

This shows that *S'* is not serializable.

A non-serializable schedule S'

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??

A non-serializable schedule S’

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		

The problem is that T2 works on a value of A determined by T1, whereas it works on a value of B which is not determined by T1: this behavior cannot show up neither in the case of the serial execution <T1,T2> nor in the case of the serial execution <T2,T1>

Anomalies

In order to help characterizing possible problematic situations caused by concurrency, people have singled out some common patterns that should be treated carefully, because they represent “incorrect” behaviors of concurrent schedules.

These patterns are called anomalies. We will now analyze the most common anomalies investigated in the literature. Notice, however, that they do not represent all possible problematic situations: there are other issues in concurrent schedules that are not captured by the anomalies that we now discuss.

Anomaly 1: reading temporary data

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(B,x)	
commit	

Note that the interleaved execution is different from any serial execution. As we said, the problem comes from the fact that the value of A is read by T2 after T1 has written on A, whereas the value of B is read by T2 before T1 writes on B.

This is a reading temporary data anomaly, because it shows up when a transaction T reads an element written by another transaction T' that has not finished yet and can therefore interfere with T' in the future

Anomaly 2: update loss

- Let T_1, T_2 be two transactions, each of the same 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 (note that x is a local variable, and therefore each x is local to the transaction it belongs to):

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

Note that the interleaved execution is different from any serial execution. 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 2: update loss

The update loss 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 3: unrepeateable read

T_1 executes two consecutive reads of the same data (assume the initial vale of A is 20):

T_1	T_2
begin	begin
READ(A,x)	
	x := 100
	WRITE(A,x)
	commit
READ(A,x)	
commit	

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

Note that the interleaved execution is different from any serial execution.

Anomaly 4: ghost update

Assume that the following integrity constraint A = B must hold

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

Note that neither T1 nor T2 in isolation 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.

Note that the interleaved execution is different from any serial execution.

Simplyfing the notion of schedule

We have seen that a schedule S is serializable if there exists a serial schedule on the same transactions that is “equivalent” to S, where two schedules S1 and S2 are equivalent if, for each database state D, the execution of S1 starting from the database state D produces the same outcome as the execution of S2 starting from the same database state D.

Warning: is the problem of checking equivalence of two schedules decidable?

Simplyfing the notion of schedule

We have seen that a schedule S is serializable if there exists a serial schedule on the same transactions that is “equivalent” to S, where two schedules S1 and S2 are equivalent if, for each database state D, the execution of S1 starting from the database state D produces the same outcome as the execution of S2 starting from the same database state D.

Warning: is the problem of checking equivalence of two schedules decidable?

If we consider general schedules (i.e., expressed in any programming language), the answer is NO, and therefore we must simplify the notion of schedule: from now on, we generally characterize each transaction T_i (where i is a nonnegative integer identifying the transaction) in terms of its read, write, commit or rollback actions, where each action of transaction Ti is denoted by a letter (read, write, commit o rollback) and the subscript i. In other words, in a schedule we ignore the operations in the local store (main memory).

Example:

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

Scheduler

A schedule represents the sequence of actions of transactions presented to the data manager. The scheduler is the part of the transaction manager that is responsible of managing the schedule, and works as follows:

– It deals with new transactions entered in the system, assigning them an identifier

– It instructs the buffer manager to read and write on the DB according to a particular sequence (in general, a serializable sequence) derived by the input schedule, making sure that concurrency is dealt with correctly by means of a specified policy (the concurrency control method used in the system)

– It is NOT concerned with specific operations on the local store of transactions (as we said before)

– It is NOT concerned with constraints on the order of executions of transactions. The last conditions means that every order by which the transactions present in the input schedule are executed by the system is acceptable to the schedule.

The last condition is very important: it implies that if two transactions are presented concurrently to the system, there is nothing in the application that imposes an order between the two transactions. Indeed, if such an order were relevant (for example, because the execution of T1 should be completed before T2 starts), then the application would have made sure that one of the transactions (e.g., T2) was presented to the system after the completion of the other transaction (e.g., T1).

Example

Application 1

1. check whether product #10 is available (Transaction T1)
2. if so, sell the product and update the db appropriately (Transaction T2)

Application 2

1. add one item of product #10 to the storage and update the db appropriately (Transaction T3)

Application 3

1. list all products that are currently not available (Transaction T4)

The way in which Application 1 has been designed, rules out the possibility that T1 and T2 are executed concurrently.

All other interleaving's among the actions of the various transactions are possible:

- T1 and T3 can be presented simultaneously to the system
- the same for T1 and T4
- the same for T2 and T3
- the same for T2 and T4
- the same for T3 and T4
- the same for T1, T3, T4
- the same for T2, T3, T4

Serializability and equivalence of schedules

As we saw before, the definition of serializability relies on the notion of equivalence between schedules, and we have decided to consider only read, write, commit and rollback) actions of transactions in dealing with serializability.

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 D of equivalence, we will be interested in

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

Two important assumptions

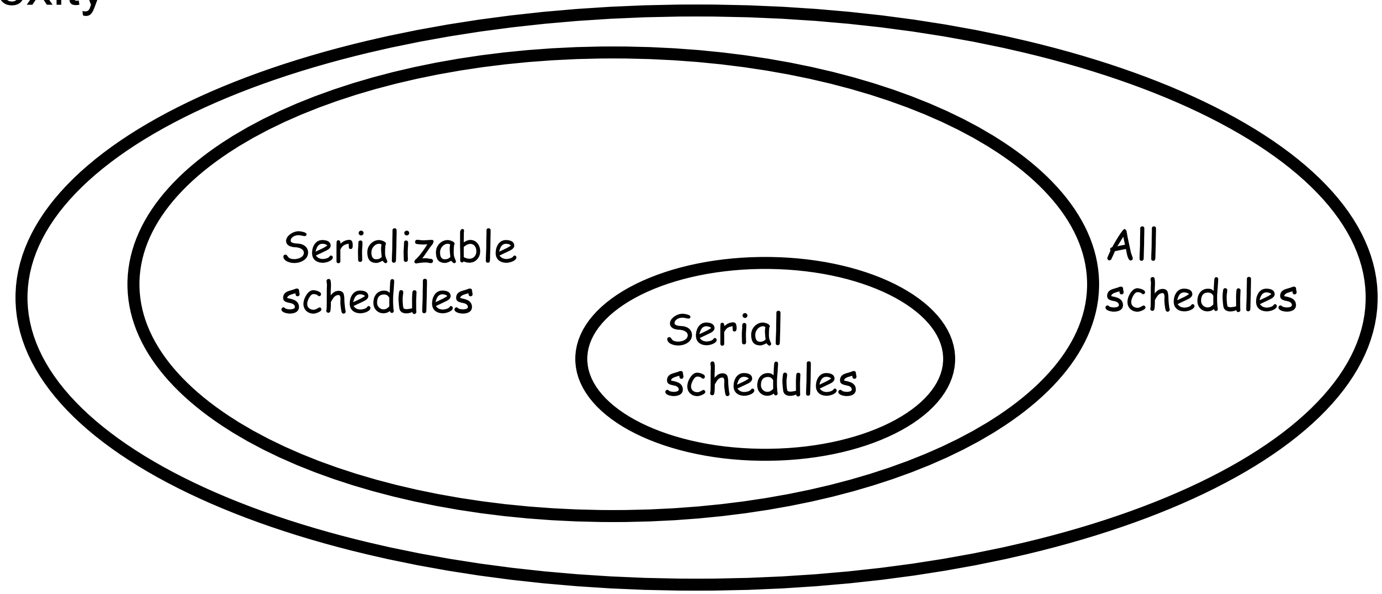
In the next slides, we will generally work under two assumptions:

1. No transaction reads or writes the same element twice and no transaction reads an element that it has written
2. No transaction executes the "rollback" command (i.,e. we assume that all executions of transactions are successful)

Sometimes we will remove the first assumptions. Later, we will remove the second assumption.

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 (i.e., the problem is decidable), and can be done with reasonable computational complexity



We will define several notions of serializability, starting with
- view-serializability
- conflict-serializability

5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability

5.2 View-serializability

5.3 Conflict-serializability

5.4 Concurrency control through locks

5.5 Recoverability of transactions

5.6 Concurrency control through timestamps

5.7 Multiversion concurrency control

5.8 Optimistic concurrency control

5.9 Concurrency control in SQL

View-equivalence and view-serializability

Preliminary definitions:

- In a schedule S, we say that $r_i(x)$ READS-FROM $w_j(x)$ if $w_j(x)$ preceeds $r_i(x)$ in S, and there is no action of type $w_k(x)$ between $w_j(x)$ and $r_i(x)$. The READS-FROM relation associated to S is

$$\text{READS-FROM}_S = \{ \langle r_i(x), w_j(x) \rangle \mid r_i(x) \text{ READS-FROM } w_j(x) \}$$

- In a schedule S, we say that $w_i(x)$ is a FINAL-WRITE if $w_i(x)$ is the last write action on x in S. The FINAL-WRITE set associated to S is

$$\text{FINAL-WRITE}_S = \{ w_i(x) \mid w_i(x) \text{ is the last write action on x in S} \}$$

Definition of view-equivalence: let S1 and S2 be two (total) schedules on the same transactions. Then S1 is view-equivalent to S2 if S1 and S2 have the same READS-FROM relation, and the same FINAL-WRITE set.

Definition of view-serializability: a (total) schedule S on {T1,…,Tn} is view-serializable if there exists a serial schedule S' on {T1,…,Tn} that is view-equivalent to S

View-serializability

Consider the following schedule (for the purpose of this example, we again consider also operations on the local store):

read1(A,t) read2(A,s) s:=100 write2(A,s) t:=100 write1(A,t)

- Is it serializable?
- Is it view-serializable?

View-serializability

Consider the following schedule (for the purpose of this example, we again consider also operations on the local store):

read1(A,t) read2(A,s) s:=100 write2(A,s) t:=100 write1(A,t)

- Is it serializable?
- Is it view-serializable?

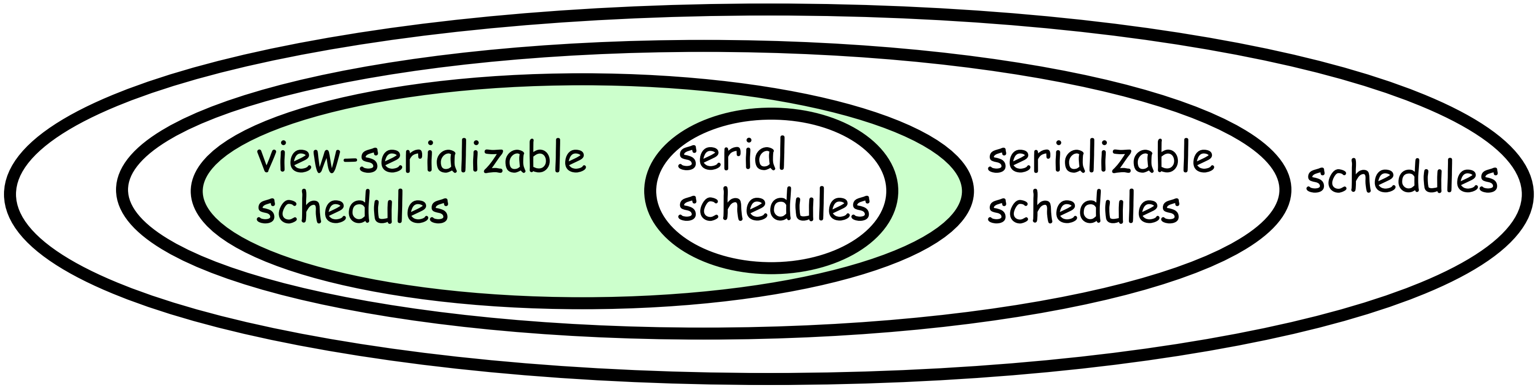
It is easy to see that the above schedule is serializable but is NOT view-serializable.

View-serializability

- There are serializable schedules that are not view-serializable. For example, the schedule we have just seen:

  read1(A,t) read2(A,s) s:=100 write2(A,s) t:=100 write1(A,t)

  is serializable, but not view-serializable
- Note, however, that in order to realize that the above schedule is serializable, we need to take into account the operations performed on the local store
- If we limit our attention to our abstract model of transaction (where only read and write operations count), and we consider as outcome of a schedule only the database state, then view-equivalence and view-serializability are the most general notions



$S_1: \ r_1(x) \ w_2(x) \ w_1(y)$

$S_2: \ r_1(x) \ w_1(y) \ w_2(x)$

READ-FROM $S_1 = \{ \}$

READ-FROM $S_2 = \{ \}$

FINAL-WRITE $S_1 = \{ w_2(x), w_1(y) \}$

FINAL-WRITE $S_2 = \{ w_2(x), w_1(y) \}$

THEY ARE VIEW-EQUIVALENT

$S_1: \ r_1(A) \ r_2(A) \ w_2(A) \ w_1(A)$

$S_2: \ r_1(A) \ w_1(A) \ r_2(A) \ w_2(A)$

READ-FROM $S_1 = \{ \}$

READ-FROM $S_2 = \{ \langle r_2(A), w_1(A) \rangle \}$

THEY ARE NOT VIEW-EQUIVALENT

$S_1: \ r_1(A) \ r_2(A) \ w_2(A) \ w_1(A)$

$S_2: \ \underline{r_1(A) \ w_1(A)} \ \underline{r_2(A) \ w_2(A)}$

$S_3: \ \underline{r_2(A) \ w_2(A)} \ \underline{r_1(A) \ w_1(A)}$

READ-FROM $S_1 = \{ \}$

READ-FROM $S_2 = \{ \langle r_2(A), w_1(A) \rangle \}$

READ-FROM $S_3 = \{ \langle r_1(A), w_2(A) \rangle \}$

S_1 IS NOT VIEW-SERIALIZABLE

Exercise

- Is the following problem decidable?

  Given two schedules on the same transactions, check whether they are view-equivalent

- If the above problem is decidable, answer the following question:

  Given two schedules on the same transactions, checking whether they are view-equivalent can be done in polynomial time?

- Is the following problem decidable?

  Given one schedule, check whether it is view-serializable

- If the above problem is decidable, answer the following question:

  Given one schedule, checking whether it is view-serializable can be done in polynomial time?

Properties of view-equivalence

- Given two schedules, checking whether they are view-equivalent is decidable and can actually be done in polynomial time
- Given one schedule, checking whether it is view-serializable is decidable and is an NP-complete problem
  - It is easy to verify that the problem is in NP. Indeed, the following is a nondeterministic polynomial time algorithm for checking whether S is view-serializable ot not: non deterministically guess a serial schedule S' on the transactions of S, and then check in polynomial time if S' is view-equivalent to S
  - Proving that the problem is NP-hard is much more difficult
- The above result implies that the best-known deterministic algorithm for checking view serializability requires exponential time in the worst case, and this is one reason why view-serializability is not used in practice

Exercise 1a

- Consider the schedules:
  1. w0(x) r2(x) r1(x) w2(x) w2(z)
  2. w1(y) r2(x) w2(x) r1(x) w2(z)
  3. w1(x) r2(x) w2(y) r1(y)
  4. w0(x) r1(x) w1(x) w2(z) w1(z)

  and tell which of them are view-serializable

- Consider the following schedules, verify that they are not view-serializable, and tell which anomalies they suffer from
  1. r1(x) w2(x) r1(x)
  2. r1(x) r2(x) w1(x) w2(x)
  3. w1(x) w2(y) w1(y) w2(x)

Exercise 1a

- Consider the schedules:
    1. w0(x) r2(x) r1(x) w2(x) w2(z)
    2. w1(y) r2(x) w2(x) r1(x) w2(z)
    3. w1(x) r2(x) w2(y) r1(y)
    4. w0(x) r1(x) w1(x) w2(z) w1(z)

  and tell which of them are view-serializable

- Consider the following schedules, verify that they are not view-serializable, and tell which anomalies they suffer from
    1. r1(x) w2(x) r1(x) -- unrepeatable read
    2. r1(x) r2(x) w1(x) w2(x) -- lost update
    3. w1(x) w2(y) w1(y) w2(x) -- ghost update

Exercise 1b

Consider the following schedule

S = r1(x) w3(x) w3(z) w2(x) w2(y) r4(x) w4(z) w1(y)

and tell whether S is view-serializable or not, explaining the answer in detail.

5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability

5.2 View-serializability

5.3 Conflict-serializability

5.4 Concurrency control through locks

5.5 Recoverability of transactions

5.6 Concurrency control through timestamps

5.7 Multiversion concurrency control

5.8 Optimistic concurrency control

5.9 Concurrency control in SQL

Conflicts and the commutativity rule

We now consider transaction actions of a certain predefined types (even more general that read, write, commit), and assume that we know which are the pairs of actions of the various types (where the two actions belong to different transactions) that are conflicting. For example, we may know that the actions of type K1 is in conflict with actions of type K2 with respect to serializability.

So, given a sequence S of actions from a set of transactions, we can build the following set (called the conflict relation of S):

conf(S) = { <p,q> | p,q are conflicting and p preceeds q in S }

Notice that "p preceeds q in S" means that p comes (not necessarily immediately) before q in S.

Based on conf(S), we can define the commutativity rule for a sequence S as follows: if p,q are adiacent actions in S belonging to different transactions, and they are such that <p,q> is not in conf(S), then the sequence p,q can be replaced by the sequence q,p (in other words, p and q can be swapped). We denote S → S' the condition by which the sequence S' can be obtained from S by means of a finite sequence of applications of the commutativity rule based on conf(S).

Example of commutativity rule applications

Suppose all red actions are conflicting with all blue actions and subscripts denote transactions. The following is a set of applications of the commutativity rule:

p1 q1 m2 <u>s2 t1</u> v1 v2 t2

Example of commutativity rule applications

Suppose all red actions are conflicting with all blue actions and subscripts denote transactions. The following is a set of applications of the commutativity rule:

p1 q1 m2 <u>s2 t1</u> v1 v2 t2

p1 q1 <u>m2 t1</u> s2 v1 v2 t2

Example of commutativity rule applications

Suppose all red actions are conflicting with all blue actions and subscripts denote transactions. The following is a set of applications of the commutativity rule:

p1 q1 m2 <u>s2 t1</u> v1 v2 t2

p1 q1 <u>m2 t1</u> s2 v1 v2 t2

p1 q1 t1 m2 <u>s2 v1</u> v2 t2

Example of commutativity rule applications

Suppose all red actions are conflicting with all blue actions and subscripts denote transactions. The following is a set of applications of the commutativity rule:

p1 q1 m2 <u>s2 t1</u> v1 v2 t2

p1 q1 <u>m2 t1</u> s2 v1 v2 t2

p1 q1 t1 m2 <u>s2 v1</u> v2 t2

p1 q1 t1 <u>m2 v1</u> s2 v2 t2

Example of commutativity rule applications

Suppose all red actions are conflicting with all blue actions and subscripts denote transactions. The following is a set of applications of the commutativity rule:

p1 q1 m2 <u>s2 t1</u> v1 v2 t2

p1 q1 <u>m2 t1</u> s2 v1 v2 t2

p1 q1 t1 m2 <u>s2 v1</u> v2 t2

p1 q1 t1 <u>m2 v1</u> s2 v2 t2

p1 q1 t1 v1 m2 s2 v2 t2

If we denote by Si (i=1,2,3,4,5) the various sequences, we have that S1 → S2, S2 → S3, S1 → S3, and so on.

It is immediate to verify that S → S' implies S' → S, i.e., if S' can be obtained from S by means of a finite sequence of applications of the commutativity rule based on conf(S), then S can be obtained from S' by means of a finite sequence of applications of the commutativity rule based on conf(S').

The notion of conflict-equivalence for sequences of actions

Definition of conflict-equivalence for sequences of actions: Two sequences of actions S1 and S2 on the same transactions are conflict-equivalent if S1 → S2, i.e., if S1 can be transformed into S2 through a sequence of applications of the commutativity rule based on conf(S1).

S1:	p1 q1 m2 <u>s2 t1</u> v1 v2 t2
S2:	p1 q1 <u>m2 t1</u> s2 v1 v2 t2
S3:	p1 q1 t1 m2 <u>s2 v1</u> v2 t2
S4:	p1 q1 t1 <u>m2 v1</u> s2 v2 t2
S5:	p1 q1 t1 v1 m2 s2 v2 t2

In the example, S1 is conflict-equivalent to S2, to S3, to S4 and to S5. S2 is conflict equivalent to S3, to S4 and to S5, and so on.

The notion of conflicts in schedules

We now go back to our context, where transactions only contain read, write and commit actions. If we want to use the notion of conflict equivalence in this context, we have to specify precisely when two actions are conflicting.

Definition of conflicting actions in schedules: 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 action. In other words, for a schedule S

conf(S) = { <p,q> | p,q belong to different transactions and p or q is a write action }

This definition reflects the following intuitions:

- Swapping two actions operating on different elements or swapping two consecutive reads in different transactions does not change the effect of the schedule: such two actions are not conflicting
- Swapping two write operations w1(A) w2(A) of different transactions on the same element may result in a different final value for A and, more generally, can change the effects of the schedule: they are conflicting
- Swapping two consecutive operations such as r1(A) w2(A) or w2(A) r1(A) may cause T1 read different values of A (before and after the write of T2, respectively), and again can change the effects of the schedule: they are conflicting.

With the above definition we know what conf(S) is for any schedule S and therefore we know which is the commutativity rule in our context.

Conflict-equivalence on schedules

It is immediate to derive the following definition for schedules:

Definition of conflict-equivalence for schedules: Two schedules S1 and S2 on the same transactions are conflict-equivalent if S1 → S2, i.e., if S1 can be transformed into S2 through a sequence of applications of the commutativity rule, based on conf(S1).

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)

A very important property

We invite the students to prove the following property:

Theorem Two schedules S1 and S2 on the same transactions T1,…,Tn are conflict-equivalent if and only if conf(S1) = conf(S2), i.e., 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

This property is extremely important, because it allows us to check conflict-equivalence in a very direct way (by the way, in polynomial time), without resorting to trying all possible sequences of applications of commutativity rules.

$$S_1: \ \dots \ r_1(x) \ \dots \ w_2(x)$$

$$S_2: \ \dots \ w_2(x) \ \dots \ r_1(x)$$

NOT EQUIVALENT!

Conflict-serializability

We are ready to provide the definition of 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

But how can conflict-serializability be checked?

One possibility would be again to enumerate all possible serial schedules on the set of transactions of S and for each of them checking equivalence with respect to S. However, this would result again in an exponential time algorithm, as in the case of view-serializability.

Conflict-serializability

Can we check conflict-serializability more efficiently?

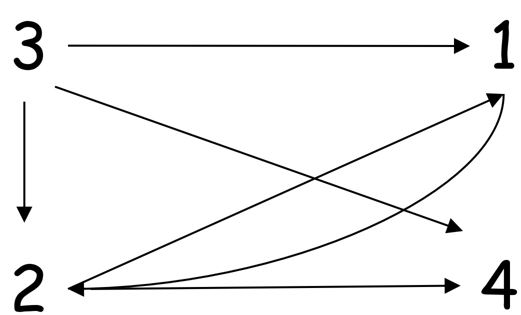
Yes, we can do it by an algorithm based on the so-called precedence graph (also called conflict graph) associated to a schedule.

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.

To prove the theorem:

- we observe that if S is a serial schedule, then the precedence graph P(S) is acyclic (easy to prove)
- we prove a preliminary lemma

Exercise 2: Prove that if S is a serial schedule, then the precedence graph P(S) is acyclic.

Preliminary lemma

Lemma If two schedules S1 and S2 on the same transactions are conflict-equivalent, then P(S1) = P(S2).

Preliminary lemma

Lemma If two schedules S1 and S2 on the same transactions are conflict-equivalent, then P(S1) = P(S2)

Proof Let S1 and S2 be two conflict-equivalent schedules on the same transactions and assume that P(S1) $\neq$ P(S2). We show that this leads to a contradiction. If P(S1) $\neq$ P(S2), then, P(S1) and P(S2) have different edges, i.e., there exists one edge Ti $\rightarrow$ Tj in P(S1) that is not in P(S2). But Ti $\rightarrow$ Tj in P(S1) means that S1 has the form

…pi(A)... qj(A)…

with conflicting pi, qj. In other words, pi(A) $<_{S1}$ qj(A). Since P(S2) has the same nodes as P(S1), S2 contains qj(A) and pi(A), and since P(S2) does not contain the edge Ti $\rightarrow$ Tj, we infer that qj(A) $<_{S2}$ pi(A). But then, S1 and S2 differ in the order of a conflicting pair of actions, i.e., conf(S1) <> conf(S2), and therefore they cannot be transformed one into the other through the swap of two non-conflicting actions. This means that they are not conflict-equivalent, and we get a contradiction. Hence, we conclude that P(S1)=P(S2).

The converse does not hold

If the converse of the previous lemma held, then the conflict-serializability theorem would already be proved. However, the converse does not hold. In fact, we can prove that P(S1)=P(S2) does not imply that S1 and S2 are conflict-equivalent.

Indeed:

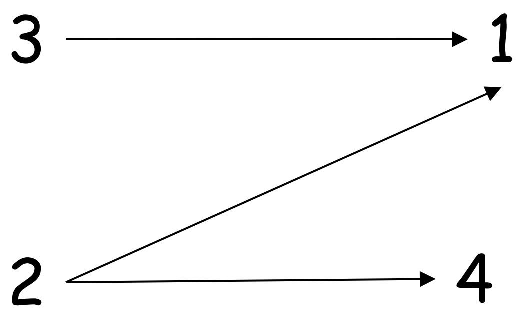
S1 = w1(A) r2(A) w2(B) r1(B)
S2 = r2(A) w1(A) r1(B) w2(B)

have the same precedence graph, but they are not conflict-equivalent, since they contain at least one pair of conflicting actions appearing in different order in the two schedules, i.e., conf(S1) <> conf(S2)).

Topological order of a graph

Definition of topological order: Given a graph G, the topological order of G is a total order S (i.e., a sequence) of the nodes of G such that if the edge Ti → Tj is in the graph G, then Ti appears before Tj in the sequence S.

Example



Possible topological order:
3 2 1 4
3 2 4 1
2 3 4 1
2 3 1 4

The following propositions are easy to prove:

- if the graph G is acyclic, then there exists at least one topological order of G
- if S is a topological order of G, and there exists a path from node n1 to node n2 in G, then n1 is before n2 in S

Exercise 3

Prove the above propositions, i.e.,

1. If the graph G is acyclic, then there exists at least one topological order of G

2. If S is a topological order of G, and there exists a path from node n1 to node n2 in G, then n1 is before n2 in S

Proof of the conflict-serializability theorem

($\Leftarrow$) We have to show that if S is conflict-serializable, then the precedence graph P(S) is acyclic. If S is conflict-serializable, then there exists a serial schedule S' on the same transactions that is conflict-equivalent to S. Since S' is serial, the precedence graph P(S') associated to S' is acyclic (Exercise 2'). But for the preliminary lemma, since S is conflict-equivalent to S', we have that P(S)=P(S'), and therefore P(S) is acyclic.

($\Rightarrow$) Let S be defined on the transactions T1,…,Tn, and suppose that P(S) is acyclic. Then there exists at least one topological order of P(S), i.e., a sequence of its nodes such that if Ti $\rightarrow$ Tj is in P(S), then Ti appears before Tj in the sequence. To such a topological order of P(S), it corresponds the serial schedule S' on T1,…,Tn such that, if Ti $\rightarrow$ Tj is in the graph, then all actions of Ti appear before Tj in S'. It is easy to see that such a schedule S' is conflict-equivalent to S. Indeed, if S' is not conflict-equivalent to S, then there is a pair of conflicting actions a_h e b_k such that ($a_h <_{S'} b_k$) and ($b_k <_S a_h$). But ($b_k <_S a_h$) means that Tk $\rightarrow$ Th is in the graph P(S), and therefore by definition of topological order, Tk appears before Th in S'. However, ($a_h <_{S'} b_k$) means that Th appears before Tk in S', and this contradicts the fact that S' corresponds to a topological order of P(S).

Algorithm for conflict-serializability

The above theorem allows us to derive the following algorithm for checking whether a given schedule S is conflict-serializable:

- build the precedence graph P(S) corresponding to S
- check whether P(S) is acyclic or not
- return true if P(S) is acyclic, false otherwise

It is immediate to verify that the time complexity of the algorithm is polynomial with respect to the size of the schedule S

Exercise 4

Check whether the following schedule is conflict-serializable

w1(x) r2(x) w1(z) r2(z) r3(x) r4(z) w4(z) w2(x)

$$\begin{tikzcd}
1 \arrow[r] \arrow[d] \arrow[dr] & 2 \arrow[d] \\
3 \arrow[ur] & 4
\end{tikzcd}$$

ALSO VIEW-SER

ACYCLIC, SO IT'S CONFLICT-SERIALIZABLE

Exercise 4a

Check whether the following schedule is view-serializable

S = r1(x) w3(x) w3(z) w2(x) w2(y) r4(x) w1(v) r4(v)

Comparison with view-serializability

The main property to understand for comparing conflict-serializability and view-serializability is the following:

Theorem Let S1 and S2 be two schedules on the same transactions. If S1 and S2 are conflict-equivalent, then they are view-equivalent.

On the basis of this theorem, one can easily show the following:

Theorem If S is conflict-serializable, then it is also view-serializable.

Exercise 5

Prove the two theorems above.

1) $\text{FINAL-WRITE}_S \neq \text{FINAL-WRITE}_{S'}$

$\text{FINAL-WRITE}_S = \{ \dots\ w_i(x) \dots \}$

$\text{FINAL-WRITE}_{S'} = \{ \dots\ w_j(x) \dots \}$

$i \neq j$

$S = \dots\ w_j(x) \dots w_i(x) \dots$

$S' = \dots\ w_i(x) \dots w_j(x) \dots$

S AND S' ARE NOT CONFLIT-EQ

2) $\text{READ-FROM}_S \neq \text{READ-FROM}_{S'}$

$\{ \langle r_i(x), w_j(x) \rangle \} \in \text{READ-FROM}_S$

$\{ \langle r_i(x), w_j(x) \rangle \} \notin \text{READ-FROM}_{S'}$

$w_j(x) \dots \dots\dots\ r_i(x)$

NOT WRITE ON x

Exercise 6

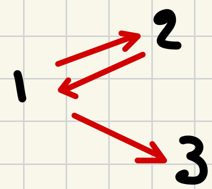
Consider the following schedule

$$w_1(y)\ w_2(y)\ w_2(x)\ w_1(x)\ w_3(x)$$

and

- check whether it is view-serializable or not,
- check whether it is conflict-serializable or not.

$S: w_1(y)\ w_2(y)\ w_2(x)\ w_1(x)\ w_3(x)$



NOT CONFLICT-SERIALIZABLE

IT'S VIEW-SER

$\underbrace{w_2(y)\ w_2(x)}_{T_2}\ \underbrace{w_1(y)\ w_1(x)}_{T_1}\ \underbrace{w_3(x)}_{T_3}$ ✗

$\underbrace{w_1(y)\ w_1(x)}_{T_1}\ \underbrace{w_2(y)\ w_2(x)}_{T_2}\ \underbrace{w_3(x)}_{T_3}$ ✓

Comparison with view-serializability

We have observed that every conflict-serializable schedule is also view-serializable.

It is important to note, however, that the converse does not hold. Indeed, there are schedules that are view-serializable and **not** conflict-serializable.

For example,

r1(x) w2(x) w1(x) w3(x)

is view-serializable, but not conflict-serializable

Order preserving conflict serializability

Let CSR denote the class of conflict serializable schedules.

Definition (Order Preservation)
A schedule S is **order preserving conflict serializable** if it is conflict equivalent to a serial schedule S' and for all t, t' $\in$ tran(S): if t completely precedes t' in S, then the same holds in S'. OCSR denotes the class of all schedules with this property.

Theorem
OCSR $\subset$ CSR.

Example showing that there are schedules in CSR that are not in OCSR:

$S = w_1(x)\ r_2(x)\ c_2\ w_3(y)\ c_3\ w_1(y)\ c_1$ $\rightarrow \in$ CSR

$\rightarrow \notin$ OCSR

Commit-order preserving conflict serializability

Definition (Commit Order Preservation)
A schedule S is **commit order preserving conflict serializable** if for all t_i, $t_j \in$ tran(S): if there are conflicting actions $p \in t_i$, $q \in t_j$ in S such that p precedes q in S, then c_i precedes c_j in S. COCSR denotes the class of schedules with this property.

Theorem
COCSR $\subset$ CSR.

Theorem
A schedule S is in COCSR iff there is a serial schedule S' conflict equivalent to S such that for all t_i, $t_j \in$ tran(S): t_i precedes t_j in S' if and only if c_i precedes c_j in S.

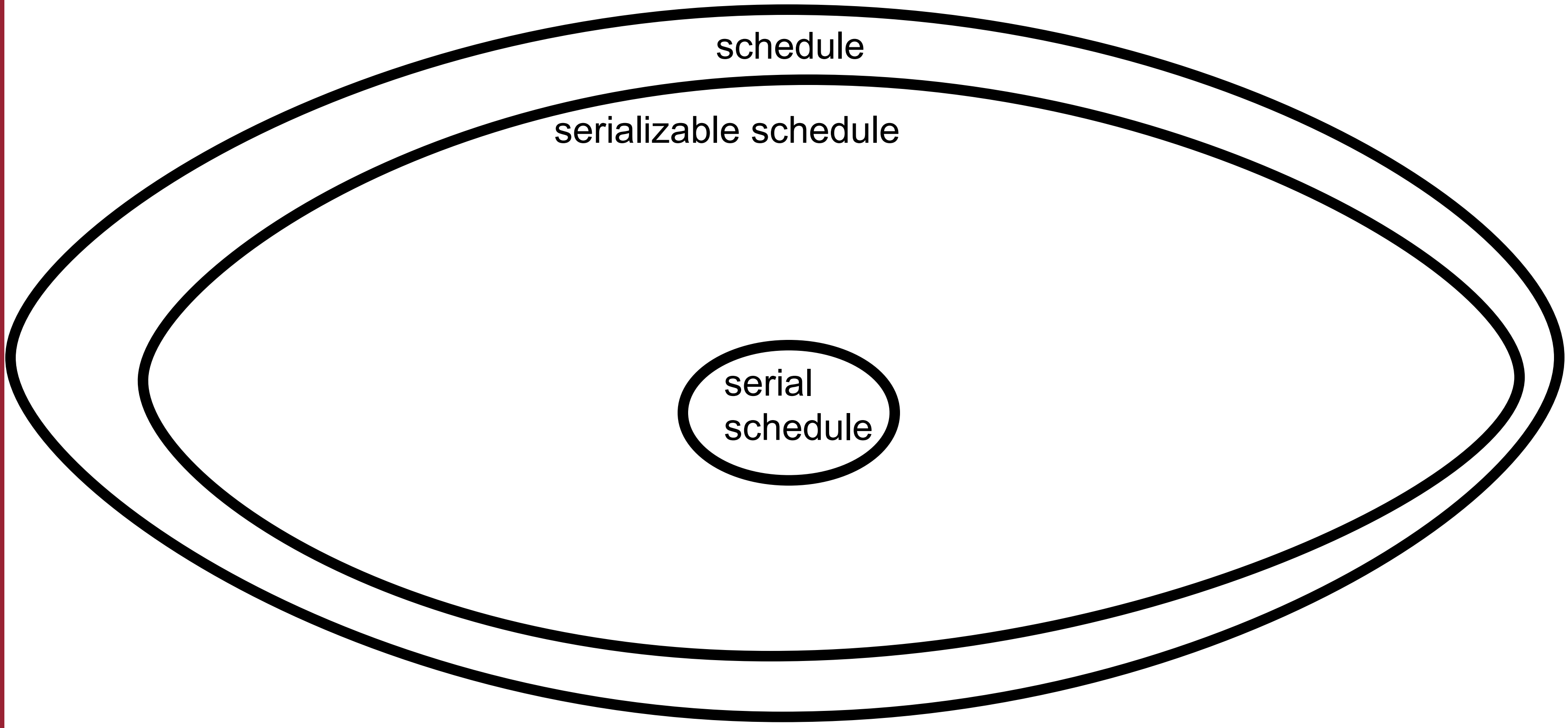
Theorem
COCSR $\subset$ OCSR.

Example showing that there are schedules in OCSR that are not in COCSR:
S = $w_3(y)\ c_3\ w_1(x)\ r_2(x)\ c_2\ w_1(y)\ c_1$ $\rightarrow \in$ OCSR

$\rightarrow \notin$ COCSR

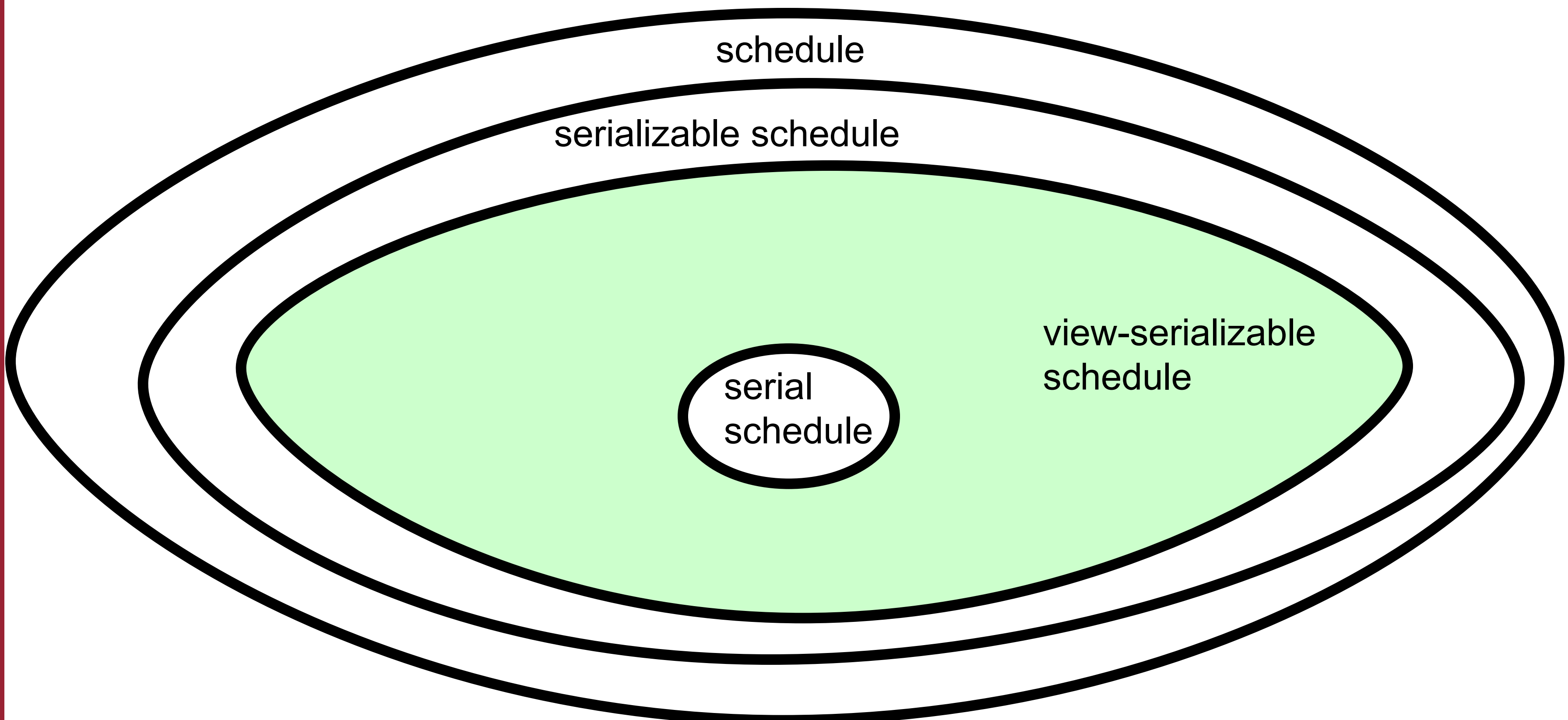
View-serializability and conflict-serializability

The relationship between view-serializability and conflict-serializability (and its variants) can be visualized as follows:



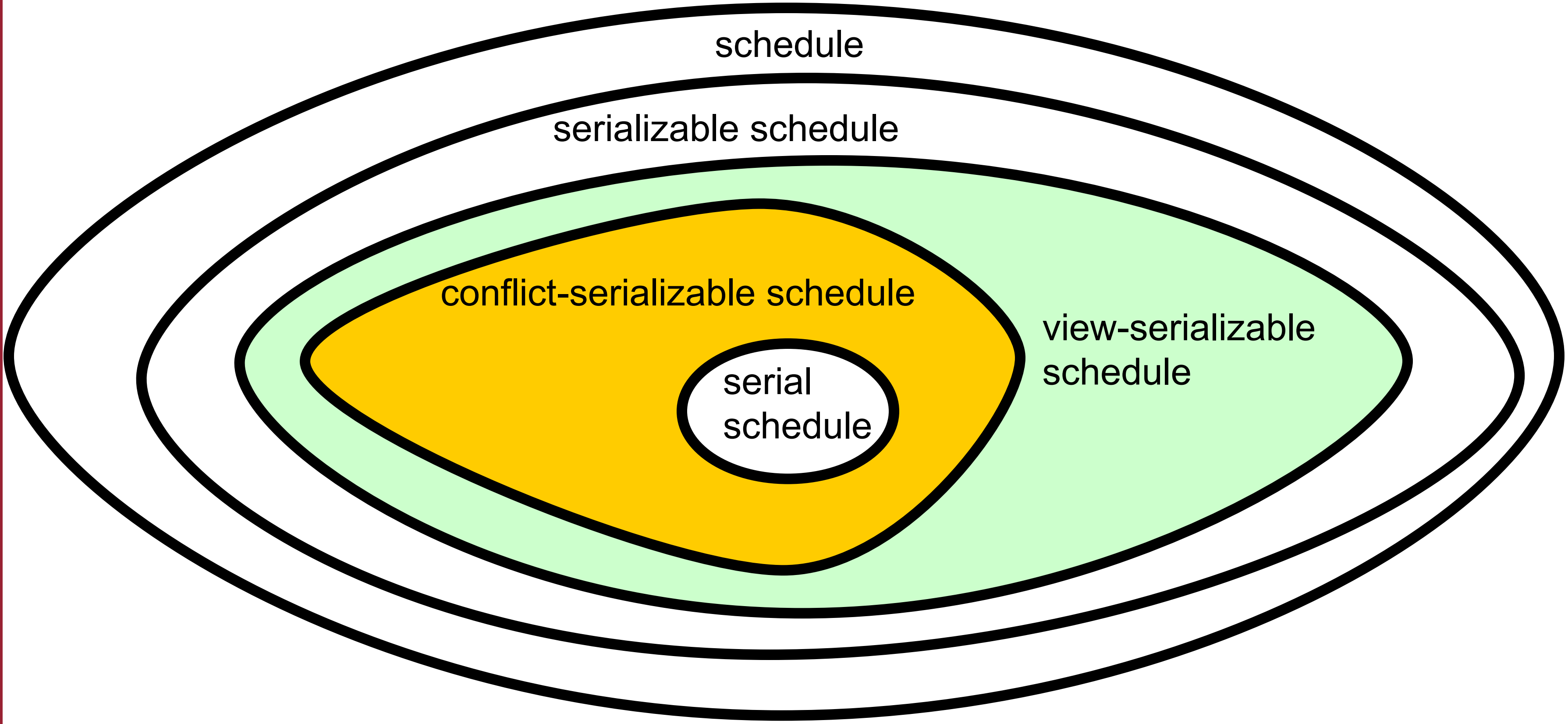
View-serializability and conflict-serializability

The relationship between view-serializability and conflict-serializability (and its variants) can be visualized as follows:



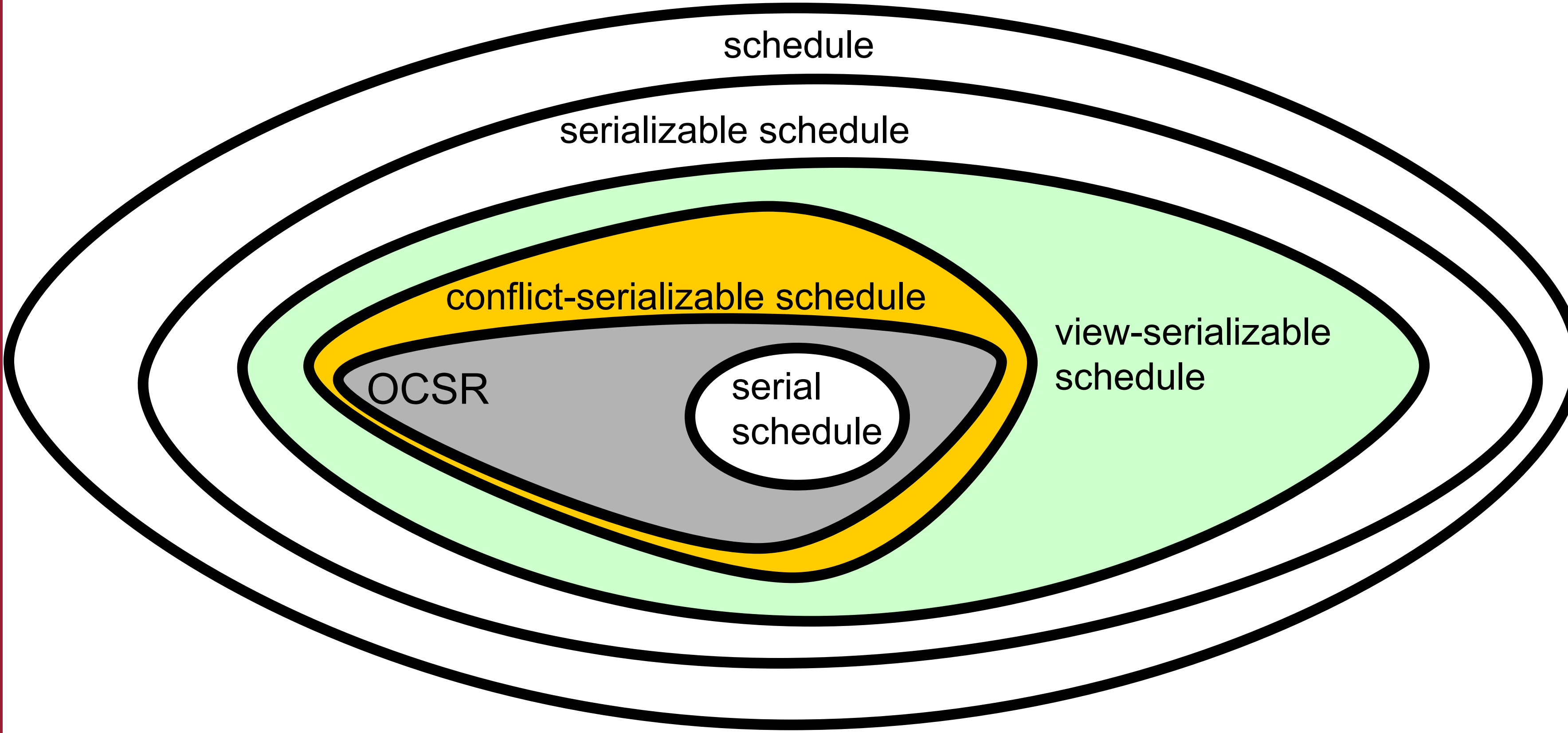
View-serializability and conflict-serializability

The relationship between view-serializability and conflict-serializability (and its variants) can be visualized as follows:



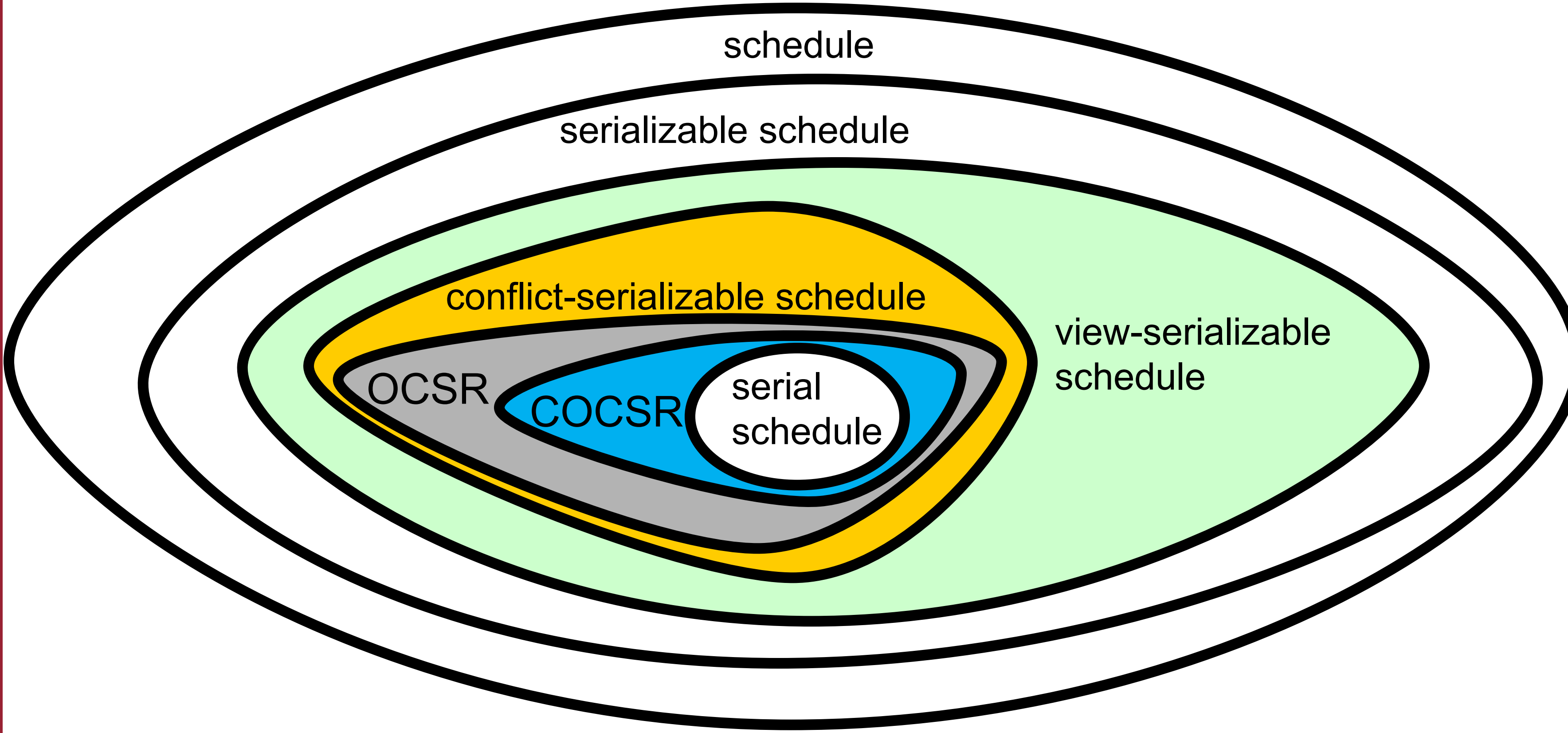
View-serializability and conflict-serializability

The relationship between view-serializability and conflict-serializability (and its variants) can be visualized as follows:



View-serializability and conflict-serializability

The relationship between view-serializability and conflict-serializability (and its variants) can be visualized as follows:



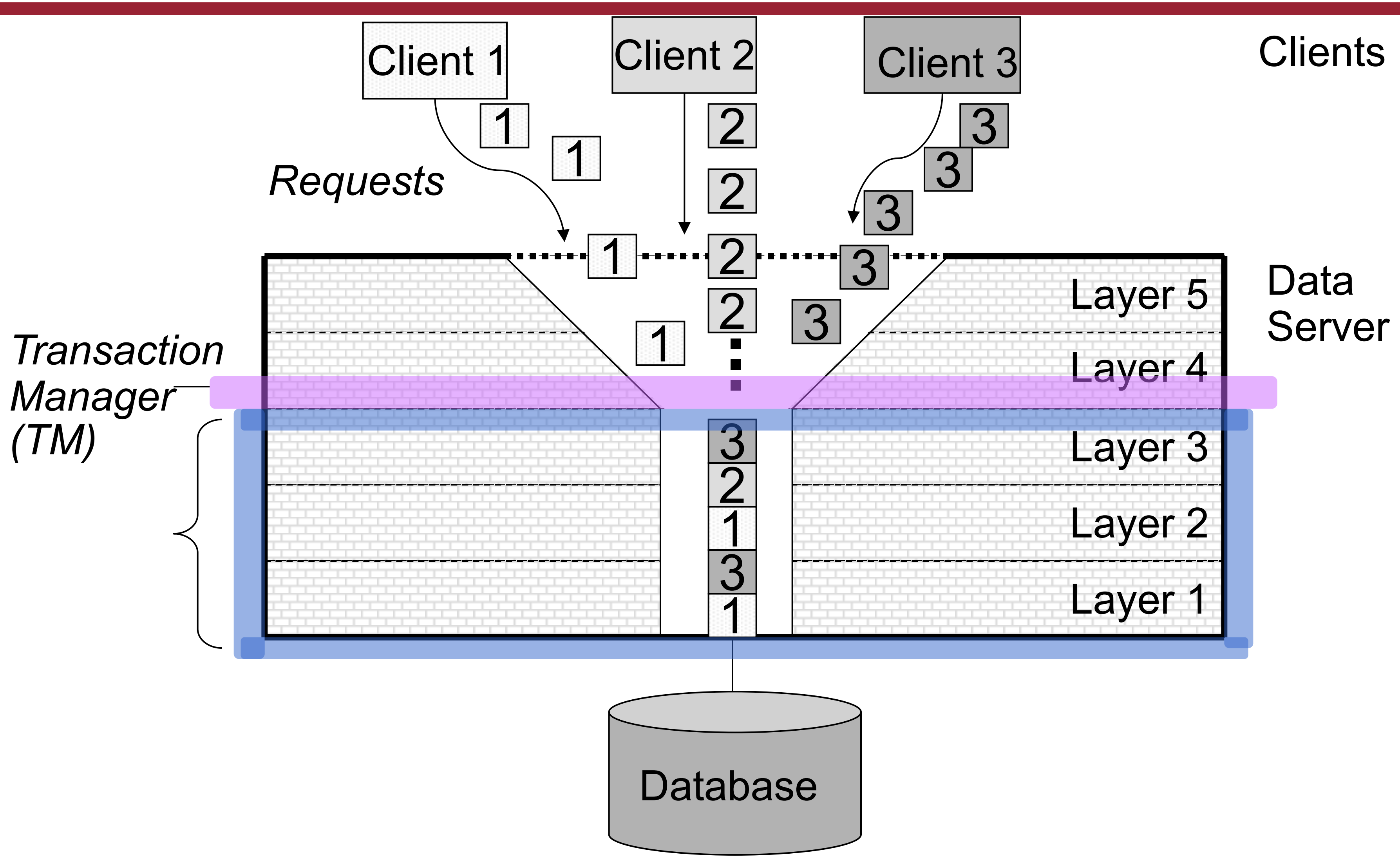
Algorithms for concurrency control

Serializability, view-serializability and conflict-serializability are extremely important in the theory of concurrency, since they represent the basic notions for characterizing the correctness of concurrency control.

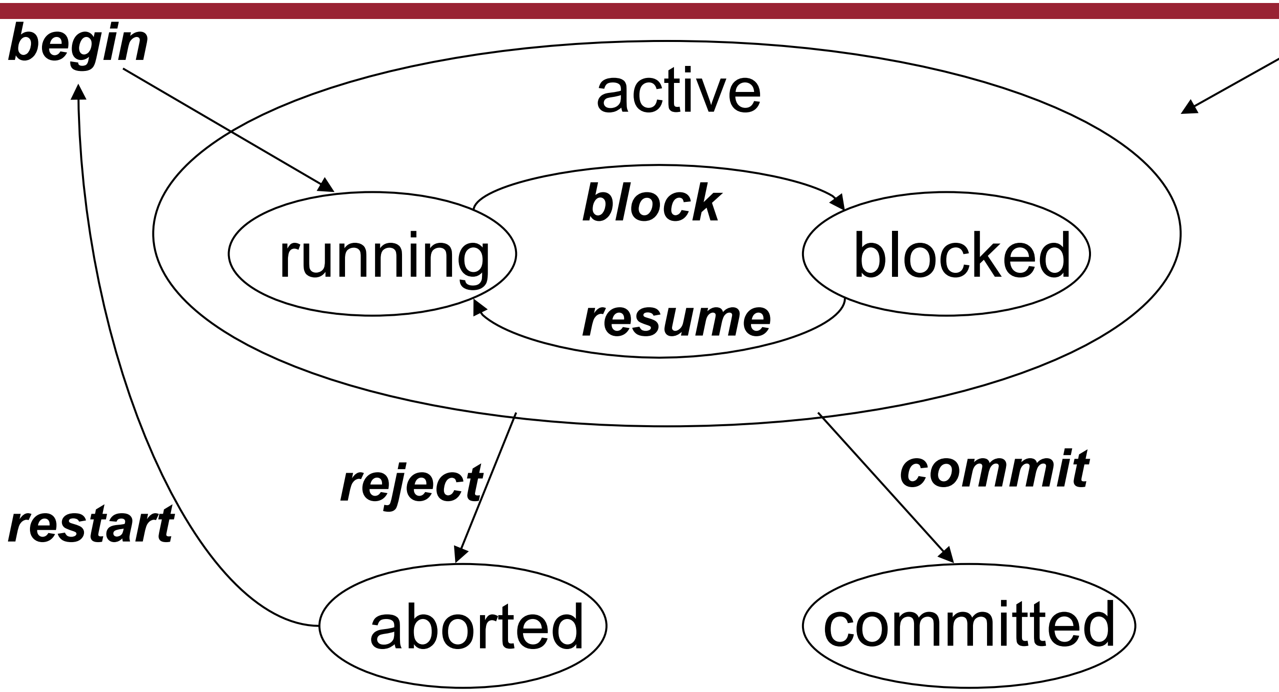
In practice, however, the “transaction and concurrency control manager” must provide an algorithm (also called **protocol**) for concurrency control. Such an algorithm corresponds to the method implememénted in the scheduler, one of the basic modules of the concurrency control manager.

The goal of the scheduler is to analyze the input schedule resulting from the requested concurrent execution of multiple transactions, and to output a corresponding schedule (the sequence with which the actions are really executed), according to a specific protocol. From now on, we concentrate on schedulers that produce conflict-serializable schedules.

Transaction Scheduler



Scheduler Actions and Transaction States

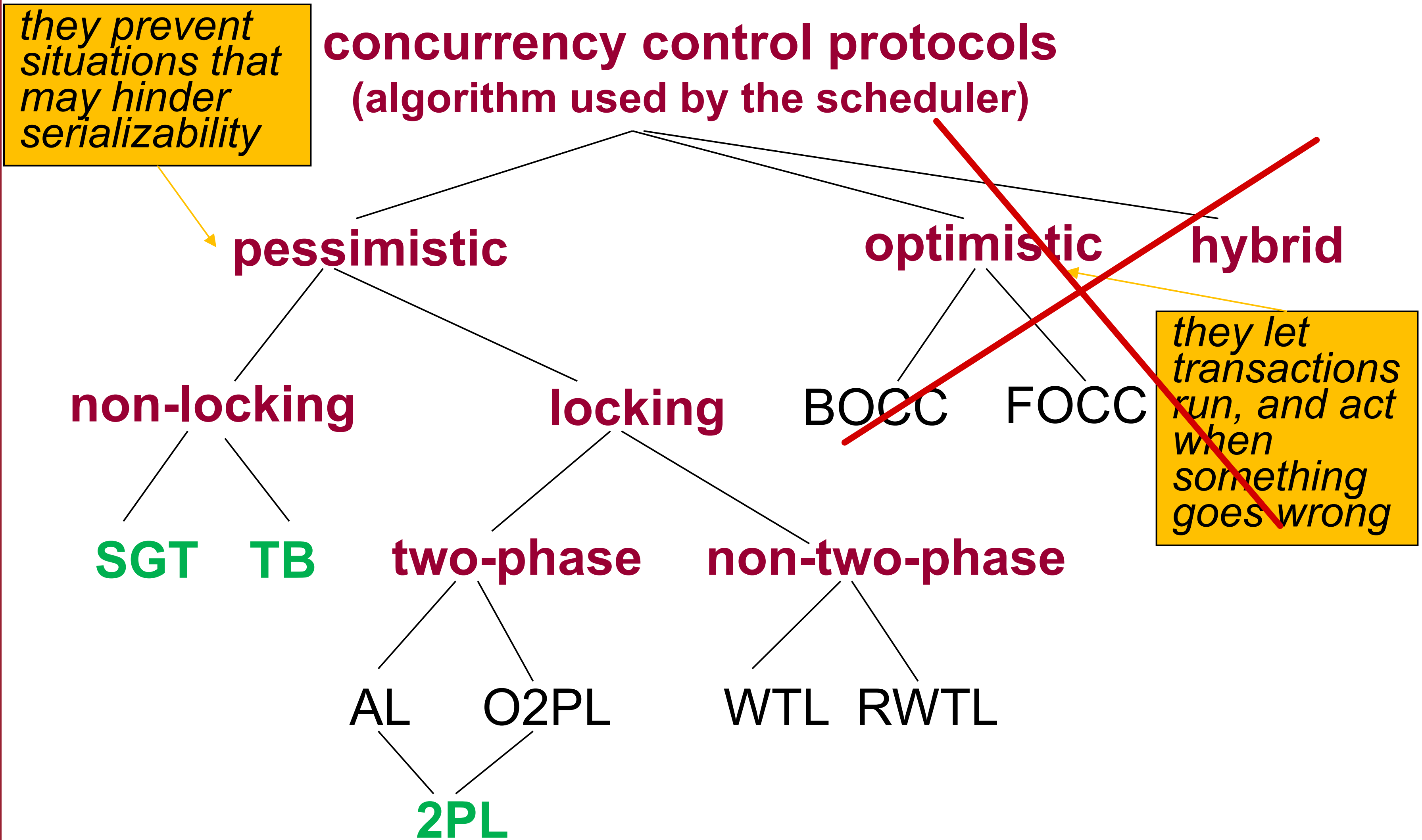


Finite state automaton for one transaction, where states are transaction states and transitions are scheduler actions performed on the transaction

For a scheduler *s*, **Gen(*s*)** denotes the set of all schedules S such that there is an input schedule S' such that S is the output produced by *s* while processing S'. In other words, Gen(s) is the set of schedules that *s* can generate in output.

Definition 4.1 (CSR Safety):
A scheduler *s* is called **CSR safe** if Gen(*s*) $\subseteq$ CSR.

Scheduler classification



Serialization graph testing SGT

Serialization graph testing is a pessimistic protocol used by the scheduler based on conflict-serializability (scheduler called SGT). The scheduler

– receives the sequence S of actions of the active transactions, possibly in an interleaved order
– manages the precedence graph associated to the sequence S
– once a new action is added to S, it updates the precedence graph of the current schedule (that is not necessarily complete), and
  - if a cycle appears in the graph, it aborts (or, kills) the transaction where the action that has introduced the cycle appears (note that killing a transaction is a complex process)
  - otherwise, it accepts the action, and continues

Obvously, Gen(SGT) = CSR. Since maintaining the precedence graph can be very costly (the size of the graph can have thousands of nodes), the notion of conflict-serializability is not used in commercial systems.

However, contrary to view-serializability, conflict-serializability can used in practice, in particular, in some sophisticated applications where concurrency control has to be taken care of by a specialized module, that can impement the SGT.

Exercise 6a

Obviously, SGT should have a strategy for removing nodes (transactions) from the precedence graph without compromising the correctness of the scheduler (otherwise, the precedence graph would grow indefinitely).

1. Prove that the following strategy is incorrect: remove the node corresponding to transaction t (as well as its ingoing and outgoing edges) from the precedence graph when t commits.

2. Define a correct strategy for removing a transaction from the precedence graph.

Exercise 6b

Consider the following schedule S (with both read and write actions, and actions on local stores):

$$r1(A, t),\ t := t - 50,\ w1(A, t),\ r2(B, s),\ s := s - 10,\ w2(B, s),$$
$$r1(B, v),\ v := v + 50,\ w1(B, v),\ r2(A, u),\ u := u + 10,\ w2(A, u)$$

and answer the following questions, with a detailed motivation for each answer:

1. Tell whether S is serializable.
2. Tell whether S is conflict serializable.
3. Tell whether S is view serializable.

Exercise 6c

Consider the three transactions T1, T2, T3 defined as follows:

T1 = r1(A), w1(A)

T2 = r2(A), w2(A)

T3 = r3(A), w3(A)

and answer the following questions, motivating the answer:

1. How many non-serial schedules on T1,T2 exist which are conflict serializable?
2. Is there at least one non-serial schedule on T1, T2, T3 that is view-serializable?

5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

Concurrency control through locks

- We observed that view-serializability and conflict-serializability are not used in commercial systems
- We will now study a method for concurrency control that is used in commercial systems (perhaps, combined with other methods). Such method is based on the use of locks and on so called "locking schedulers"
- In the methods based on locks, a transaction must get a permission in order to operate on an element of the database. The lock is a mechanism to ask and get such a permission
- We will study:
  - the general rules for using locks
  - specific rules (protocols) ensuring serializability
- We will study
  - first, only exclusive locks (for simplicity)
  - then, both exclusive and shared locks

Primitives for exclusive lock

- As we said before, for the moment we will consider exclusive locks only. Later on, we will take into account more general types of locks (shared and exclusive)

- We introduce two new operations (besides "read" and "write") that can appear in lock-extended schedules, i.e., "schedules with locks and unlocks", or simply "schedules with locks". Such operations are used to request and release the exclusive use of an element A in the database:
  - Lock (exclusive): $l_i(A)$
  - Unlock: $u_i(A)$

LOCKING
SCHEDULER BASED ON LOCKING
PROPERTIES OF SUCH SCHEDULER

EXCLUSIVE

- The lock operation $l_i(A)$ means that the exclusive use of element A of the database is asked in order for transaction Ti to operate on A.

- The unlock operation $u_i(A)$ means that the exclusive use of element A is taken off from transaction Ti (so, Ti renounces the use of A)

Well-formed transactions and legal locking schedules

The following two rules must be satisfied in order for a lock-extended schedule to be meaningful:

- Rule 1: Every transaction that appears completely in a schedule is well-formed. A transaction Ti is well-formed if no $l_i(x)$ or $u_i(x)$ is issued more than once, and every read or write action pi(A) 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 with locks is legal. A schedule S with locks is legal if no transaction in it locks an element A when a different transaction has currently the lock on A, i.e., it has granted the lock on A and has not yet unlocked A

  S: ……$l_i(A)$ ……………….... $u_i(A)$ ……

  $\longleftrightarrow$

  no $l_j(A)$

In the following we assume that no transaction issues a lock or unlock command twice.

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,T2,T3 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

Locking scheduler based on exclusive locks

To trace all the locks granted, the locking scheduler manages a data structure, called lock table.

A passive locking scheduler (see later for the notion of active locking scheduler) processes a lock-extended schedule S in input (i.e., a schedule with lock/unlock commands) and produces a lock-extended schedule in output by using the following rules:

- When processing a step $o_i(x)$ of the input lock-extended schedule S (where $o_i \in \{r,w\}$), the passive locking scheduler proceeds as follows:

  if x is locked by T_i

  then o_i (x) proceeds

  else T_i is blocked (and re-executed later on)

Passive locking scheduler (for exclusive locks)

- When processing a step $l_i(x)$ of the input lock-extended schedule S, the passive locking scheduler proceeds as follows:

  If x is locked by a transaction T_j (with $j \neq i$)

  then T_i is blocked (and, possibly, resumed later on)

  else the $l_i(x)$ command is executed and the lock table is updated
- When processing a step $u_i(x)$ of the input lock-extended schedule S, the locking scheduler simply updates the lock table
- The locking scheduler makes sure that, before dimissing T_i, the command $u_i(x)$ is present in the output for each item x such that $l_i(x)$ is present in the output

It follows that the effect of a passive locking scheduler when processing a lock-extended schedule in input is to produce in output a lock-extended schedule (which might be different from the input one, because of blocked and/or resumed transactions) that is legal and is such that all its transactions are well-formed.

Example of locking scheduler behaviour

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

Output:

T1	T2
l1(A); r1(A); w1(A)	
	l2(A) – blocked
l1(B); r1(B); u1(A)	
	l2(A) – resumed
	r2(A); w2(A); u2(A)
w1(B); u1(B)	
	l2(B); r2(B)
	w2(B); u2(B)

The risk of deadlock

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

T1	T2
l1(A); r1(A)	
	l2(B); r2(B)
w1(A)	
	w2(B)
l1(B) – blocked	
	l2(A) – blocked

To ensure that the lock-extended schedule is legal, the passive locking 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).

Locking scheduler based on exclusive locks

In the previous description, we have analyzed the case where the input lock-extended schedule has the complete set of locking commands, i.e., the case where such commands are issued by the transactions and the scheduler is passive, i.e., it does not insert locking commands itself.

In reality, several locking schedules are able to deal with the situation where the lock and unlock commands in the input lock-extended schedule can be issued not only by transactions, but also by the locking scheduler itself.

In this case, the locking scheduler is called active, and its behavior is more complex, because it may decide autonomously to issue lock or unlock commands in order not to block the transactions.

In what follows we illustrate the behaviour of an active locking schedule, and we will assume that all the locking schedulers we will refer to are active.

Examples of active locking scheduler behaviour

The following are examples of how an active locking scheduler can behave when processing an input schedule (possibily extended with locks):

COMMITS

Input : l1(x) r1(x) w2(x) l2(y) w2(y) u2(y) c1 c2

Possible output : l1(x) r1(x) u1(x) l2(x) w2(x) l2(y) w2(y) u2(y) u2(x) c1 c2

Input : r1(x) w2(x) w2(y) c1 c2

Possible output : l1(x) r1(x) u1(x) l2(x) l2(y) w2(x) u2(x) w2(y) u2(y) c1 c2

Input : r1(x) l2(X) w2(x) w2(y) w3(y) w1(x) c1 c2 c3

Possible output : l1(x) r1(x) l3(y) w3(y) u3(y) w1(x) u1(x) c1 l2(x) l2(y) w2(x) w2(y) u2(x) u2(y) c2 c3

The examples show that:

1. The input schedule may contain lock/unlock commands (but not necessarily all of them)
2. Given an input schedule, the output schedule of a locking scheduler is not uniquely determined; in particular, the scheduler may have more than one option for issueing the lock/unlock commands

Locking scheduler based on exclusive locks

If S is a lock-extended schedule, then DT(S) denotes the "data action projection of S", i.e., the projection of S onto the actions of type read, write, commit, abort (r,w,c,a).

For example, for the schedule S:

l1(x) r1(x) w2(x) l2(y) w2(y) u2(y) c1 c2

we have that DT(S) is:

r1(x) w2(x) w2(y) c1 c2

Notice that if a schedule S contains only data actions (actions of type r,w,c,a), then obviously DT(S) = S.

Also, we remind the reader that for a scheduler s, Gen(s) denotes the set of all schedules that s can produce in output (i.e., can generate). In what follows, by abuse of notation, for a locking scheduler s, we often denote by Gen(s) the set Gen(s) = { DT(S) | S is produced in output by s }

Does the locking scheduler ensure 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 have to restrict the behavior of the locking scheduler. One famous method is the "Two-Phase Locking (2PL) protocol":

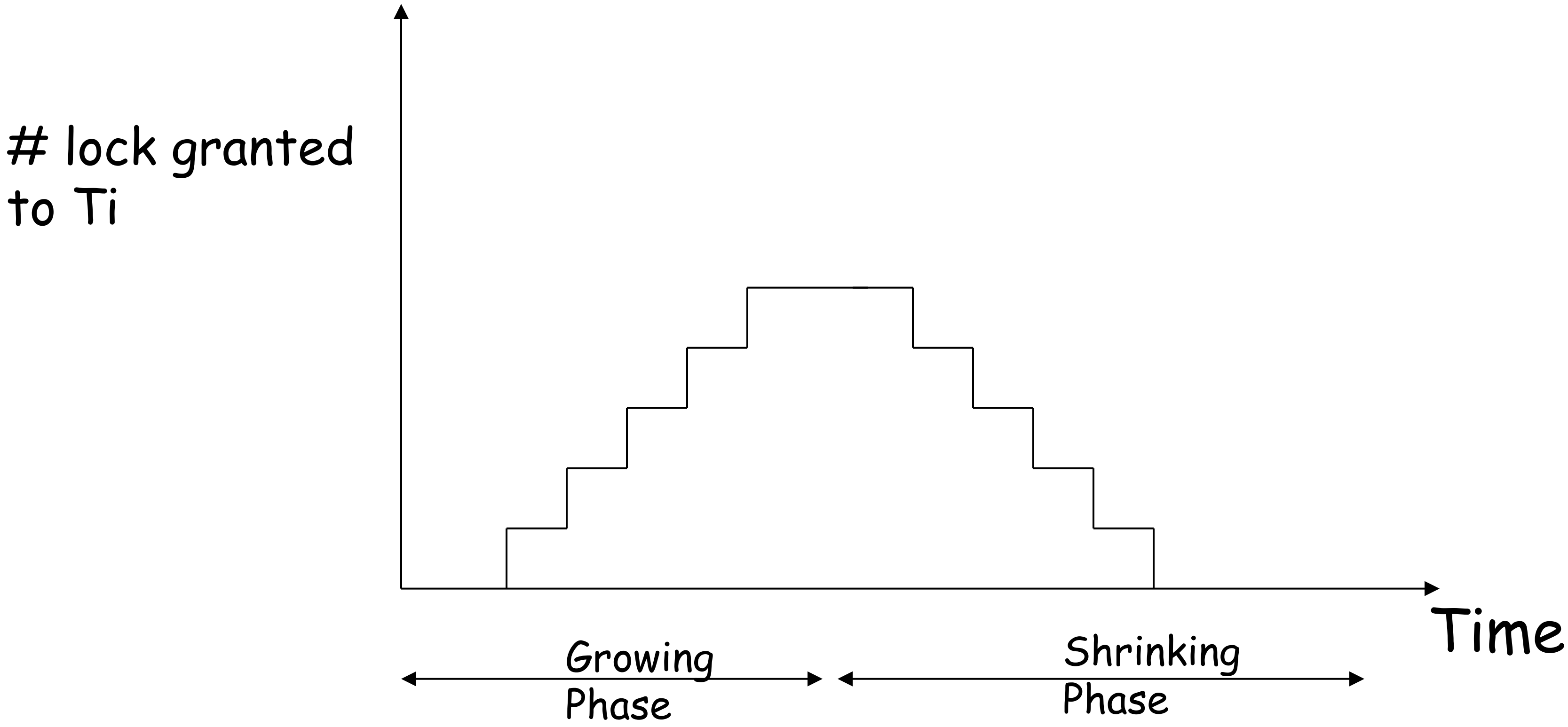
Definition of two-phase locking protocol (with only exclusive locks): A locking scheduler (with exclusive lock) follows the two-phase locking protocol if for every output S generated by the scheduler and every transaction Ti appearing in S, all lock operations of Ti precede all unlock operations of Ti.

In picture, if S is an output of a locking scheduler following the two-phase locking protocol, then S has the form:

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

The two phases of Two-Phase Locking

Locking and unlocking scheme in a transaction following the Two-Phase Locking (2PL) protocol



Example of schedule following the 2PL protocol

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

Example of schedule following 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) - resumed u2(A); r2(B) B:=Bx2; w2(B); u2(B)

The 2PL protocol avoids the ghost update while accepting concurrency

Note that the lock-extended schedule is legal

The risk of deadlock exists even with 2PL

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

NOTE: Deadlock can still occur with schedulers following the 2PL protocol.

Properties of the schedules generated by the 2PL scheduler

By combining the definition of 2PL protocol with what we said about general locking schedulers, we can show that any locking scheduler (with exclusive lock) following the 2PL protocol (called "2PL scheduler with exclusive locks", or simply 2PL for the moment) ensures that the set of schedules generated by it is the set of lock extended schedules enjoying the following properties:

1. is legal
2. all its transactions are well-formed
3. in all its transactions, all lock operations precede all unlock operations

We denote by Gen(2PL) the set of all schedules generated by all the 2PL schedulers.

Properties of the schedules generated by the 2PL scheduler

Since we are talking about active locking schedulers, it is very important to notice that, in order to generate schedules with the above properties, and to try to avoid blocking transactions, a 2PL scheduler can make use of its ability to insert lock/unlock commands, possibly exploiting the knowledge it has on the transactions.

For example, when processing the schedule

r1(X) w2(X) w1(Y) c1 …

the 2PL scheduler, in order not to block T2, may decide to anticipate the lock on Y for transaction T1, so to unlock X while still following the 2PL protocol, and thus not blocking any transaction. In other words, the output could be:

l1(X) r1(X) l1(Y) u1(X) l2(X) w2(X) w1(Y) u1(Y) c1 …

Obviously, the attempt to anticipate lock actions in order not to block transactions is not always possible, as this example shows (where either T2 or T3 must be blocked):

r1(X) w2(X) w3(Y) c3 w1(Y) c1 c2 …

In this case, the decision could be to block T2 and produce the output:

l1(X) r1(X) l3(Y) w3(Y) u3(Y) c3 l1(Y) w1(Y) u1(X) u1(Y) c1 l2(x) w2(X) u2(X) c2 …

The class of 2PL schedules with exclusive locks

We denote by "2PL schedule with exclusive locks" the class of schedules defined as follows:

{ DT(S) | there exists a schedule S' such that S is the output of a 2PL scheduler with exclusive locks when processing S' }

In other words, the class includes exactly those DT(S) for some S generated by any 2PL scheduler with only exclusive locks.

If S is in the class of "2PL schedule with exclusive locks", we say that it is "accepted" by the 2PL scheduler with exclusive locks, or, simply, that it is a 2PL schedule with exclusive locks.

Indeed, if S is in the class of "2PL schedule with exclusive locks", then it is easy to see that if we give S in input to the 2PL scheduler with only exclusive locks, the output is exactly S itself. Therefore, the scheduler does not change the schedule S and this is the reason why we say that S is "accepted" by the 2PL scheduler with exclusive locks.

The class of 2PL schedules with exclusive locks

Note that if S is without lock operations and is accepted by a 2PL scheduler, then S=DT(S1) for some S1 produced in output by a 2PL scheduler R when processing S; in other words, given S in input to R, we get S in output, once we ignore the lock operations in the sequence S1 produced by the scheduler.

Note that, given a schedule S (without lock operations), deciding whether S is in the class of “2PL schedule with exclusive locks” is not trivial: indeed it requires checking whether there exists a schedule with exclusive locks S’ ∈ Gen(2PL) obtained with input S such that DT(S’) = S. In other words, it requires to check whether we can insert exclusive lock and unlock commands into S in such a way that the resulting sequence of actions can be generated by a 2PL schedule with exclusive locks.

Exercises 7a

1. Consider the following schedule S1:

r1(x) w2(z) w2(x) r3(y) w1(z) w3(z) r2(t)

and tell whether it is in the class of "2PL schedule with exclusive locks" ,

2. Consider the following schedule S2:

r1(x) w2(x) w3(y) w3(z) w1(z) w3(x)

and tell whether it is in the class of "2PL schedule with exclusive locks"

1) $r_1(x)\ w_2(z)\ w_2(x)\ w_1(z)\ w_3(z)$

$\downarrow$

$L_1(x)\ r_1(x)\ L_2(z)\ w_2(z)\ U_2(z)\ L_1(z)\ L_2(x)$

2PL and conflict-serializability

We remind the reader that, for the moment, we are only considering exclusive locks.

Theorem If S ∈ Gen(2PL), then DT(S) is conflict-serializable.

Proof

Let S be a schedule in Gen(2PL), i.e., generated by a 2PL scheduler (with only exclusive locks). To show that DT(S) is conflict-serializable, we proceed by induction on the number N of transactions in S.

Base step: If N=1, DT(S) is serial, and therefore is trivially conflict-serializable.

Proof continued

Inductive step: Suppose that S ∈ Gen(2PL) is the output of a 2PL schedule and is defined on transactions T1,…,TN (N>1), and let Ti be the first transaction that executes an unlock operation, say ui(X), in S. We now show that we can move all operations of Ti in front of S, without swapping any pair of conflicting actions. We consider an action wi(Y) in Ti (analogous observation holds if we considered ri(Y) instead of wi(Y)), and we show that it cannot be preceded by any conflicting action in S. Indeed, suppose that there is a conflicting action wj(Y) in S preceding wi(Y) with j different from i:

… wj(Y) … uj(Y) … li(Y) … wi(Y) …

Since Ti is the first transaction that executes an unlock operation ui(X) in S, we either have

… ui(X)… wj(Y) … uj(Y) … li(Y) … wi(Y) …

or

… wj(Y) …ui(X) … uj(Y) … li(Y) … wi(Y) …

In both cases, ui(X) would appear before li(Y) in S, and S ∉ Gen(2PL). Since this is a contradiction, we can then conclude that, by moving all actions of Ti in front of S, we get a schedule S'' such that DT(S'') is conflict-equivalent to DT(S) and S'' has the form

(actions of Ti) (remaining actions of S)

The part denoted by S' = (remaining actions of S) is a legal schedule on (N-1) transactions constituted by well-formed transactions following the 2PL protocol (with exclusive locks) and therefore S' ∈ Gen(2PL). For the inductive hypothesis, DT(S') is conflict-serializable, which means that there is a serial schedule S''' on the (N-1) transactions that is conflict equivalent to DT(S'). Now, consider the serial schedule constituted by Ti followed by S''': such a schedule is obviously conflict equivalent to DT(S), which implies that DT(S) is conflict-serializable.

What does the theorem intuitively say

The theorem says that if S ∈ Gen(2PL), then DT(S) is conflict-equivalent to the serial schedule that orders the transactions of S according to the following rule:

1. Take as first transaction the one that executes the first unlock operation in S
2. Take as second transaction the one that executes the first unlock operation among the remaining (N-1) transactions in S
3. …..

N-1. Take as (N-1)-th transaction the one that executes the first unlock operation among the remaining 2 transactions in S

N. Take the last transaction as the N-th transaction

2PL and conflict-serializability

We have seen that S ∈ Gen(2PL) implies DT(S) conflict-serializable. However, the converse does not hold, as shown here:

Theorem There exists a conflict-serializable schedule S that is not in the class “2PL schedule with exclusive locks”.

Proof It is sufficient to consider the following schedule S:

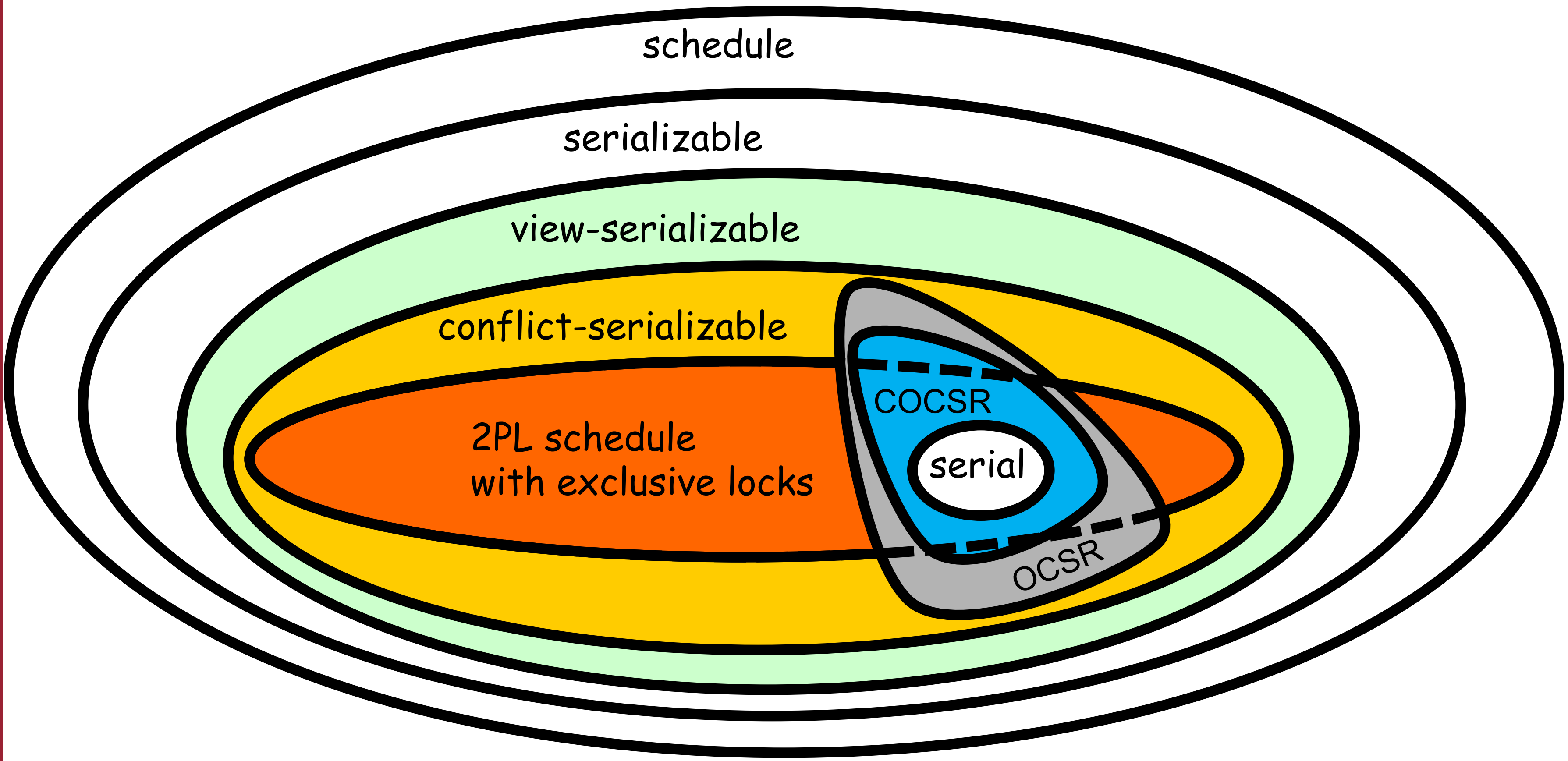
w1(x) r2(x) r3(y) w1(y)

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 the resulting lock extended schedule is in Gen(2PL). Indeed, it suffices to notice that the 2PL scheduler 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 should also 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 no locking scheduler can follow the 2PL protocol while processing S.

2PL and conflict-serializability

Graphically, the relationship between conflict-serializability and 2PL with exclusive locks can be represented as follows:



Adding 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), issued for writing A

sli(A): shared lock (also called read lock), issued for reading A

ui(A): unlock, issued for releasing the lock on A

Well-formed transactions with shared locks

With shared and exclusive locks, Rule 1 for judging the quality of schedules changes as follows.

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) and then to execute xli(A) without unlocking A. 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” or “lock conversion” .

Legal schedule with shared locks

With shared and exclusive locks, Rule 2 for judging the quality of schedules changes as follows.

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

How locks are managed

- The 2PL scheduler now 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 $Tj \neq Ti$ on A

Lock already granted to Ti on A		S	X
	S	yes	no
	X	no	no

Locking scheduler based on shared and exclusive locks

The lock and unlock commands can be issued by:

- the transactions, or
- the locking scheduler in the transaction manager

To trace all the locks granted, the scheduler manages a data structure, called lock table. A locking scheduler processes a lock-extended schedule S in input and produces a lock-extended schedule in output by using appropriate rules, which are generalizations of the rules we have already seen, taking into account the presence of both types of locks, and the corresponding compatibility matrix.

We leave as an exercise to define precisely the behavior of the schedule.

As in the previous case, the effect of a locking scheduler when processing a lock-extended schedule in input is to produce in output a lock-extended schedule (which might be different from the input one) that is legal and is such that all its transactions are well-formed.

Locking scheduler based on shared and exclusive locks

With both shared and exclusive locks,the "Two-Phase Locking (2PL) protocol" becomes:

Definition of two-phase locking protocol (with shared and exclusive locks): A locking scheduler (with both shared and exclusive lock) follows the two-phase locking protocol if for every output S generated by the scheduler and every transaction Ti appearing in S, all lock (xl or sl) operations of Ti precede all unlock 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

Note that the execution of the unlock commands requires also to choose which transaction to allow to proceed (in case there are many of them blocked). This problem was already present in the previous case of only exclusive locks, but it is even more serious now.

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, and the one we assume if not otherwise stated, because it avoids "starvation", i.e., the situation where a request of a transaction is never granted.

The class of 2PL schedules

We denote by "2PL schedule" (or simply 2PL) the class of schedules defined as follows:

{ DT(S) | there exists a schedule S' such that S is the output of a 2PL scheduler with shared and exclusive locks when processing S' }

In other words, the class includes exactly those DT(S) for some S generated by a 2PL scheduler with both shared and exclusive locks. If S is in the class of "2PL schedule", we say that it is "accepted" by the 2PL scheduler, or it is a 2PL schedule with shared and exclusive locks.

As before, if S is without lock operations and is accepted by a 2PL scheduler, then S=DT(S') for some S' produced in output by a 2PL scheduler R when processing S; in other words, given S in input to R, we get S in output, once we ignore lock operations produced by the scheduler.

Note also that, given a schedule S (without lock operations), deciding whether S is in the class of "2PL schedule" is not trivial: indeed, it requires checking whether there exists a schedule S' ∈ Gen(2PL) such that DT(S') = S. In other words, it requires to check whether we can insert shared lock, exclusive lock and unlock commands into S in such a way that the resulting sequence of actions can be generated by a 2PL scheduler.

Exercise 7b

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 7b: 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:

lock anticipation

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

Lock anticipation

In the previous example, we have shown that the schedule S was a 2PL schedule with shared and exclusive locks by showing that a 2PL scheduler can generate S' such that DS(S') = S, in particular using "lock anticipation" (that we already discussed in the context of exclusive locks only)

Obviously, in practice lock anticipation can be used in all the situations in which the scheduler has appropriate knowledge about the "future" actions of the various transaction.

This happens, for example, when the scheduler knows the "code" corresponding to the various transactions.

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 (now, 2PL denotes the 2PL scheduler with both shared and exclusive locks). The following theorem can be proved similarly to the corresponding theorems regarding 2PL schedulers with only exclusive locks.

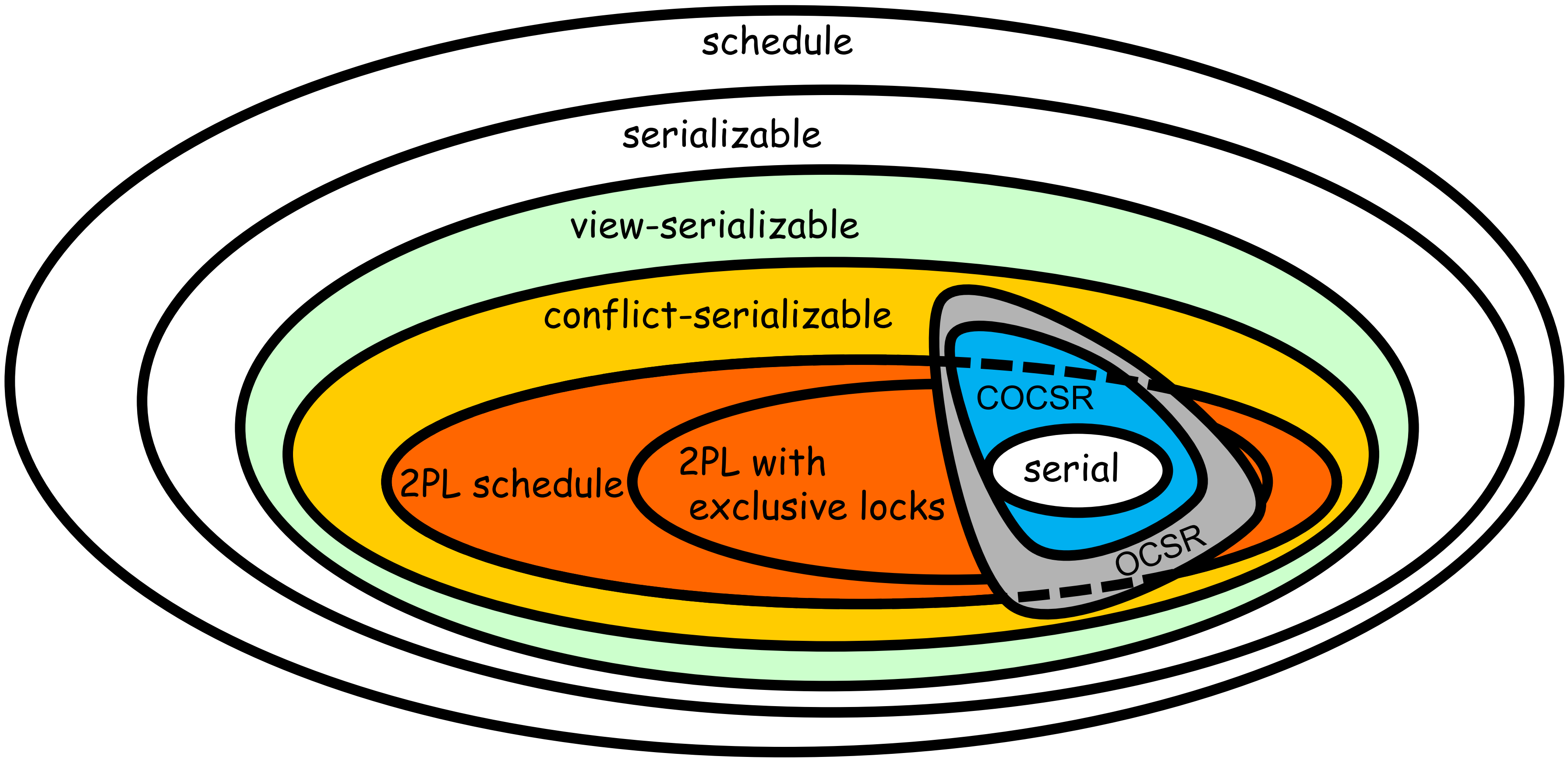
Theorem If S ∈ Gen(2PL), then DT(S) is conflict-serializable.

Theorem There exists a conflict-serializable schedule that is not in the class of 2PL schedule with exclusive and shared locks.

Obviously, with shared locks, the risk of deadlock is still present, like in:

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

2PL and conflict-serializability



Deadlock management

- We recall that the **deadlock** occurs when two transactions T1 and T2 have the use of two elements A and B, and each of them asks for an exclusive lock on the element of the other one, and therefore no one can proceed
- The probability of deadlock grows linearly with the number of transactions and quadratically with the number of lock requests in the transactions

T1	T2
xl1(A); r1(A)	
	xl2(B); r2(B)
A:=A+100;	
	B:=Bx2
w1(A)	
	w2(B)
sl1(B) – blocked!	
	sl2(A) – blocked!

Techniques for deadlock management

1. Timeout

2. Deadlock recognition and solution

3. Deadlock prevention

Timeout

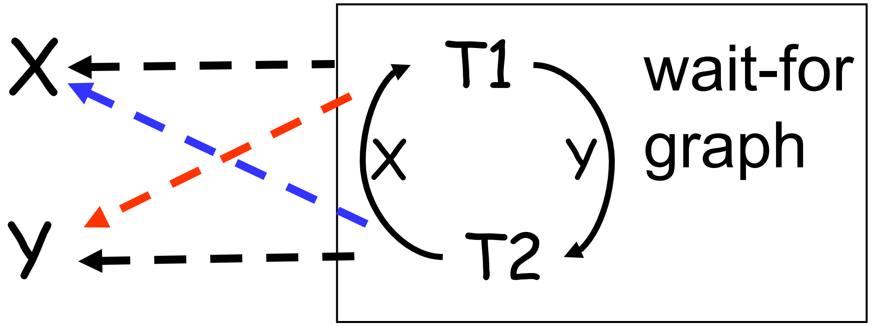
- The system fixes a timeout *t* after which a transaction waiting for a lock is killed

- Advantages
  - very simple

- Disadvantages
  - if *t* is high, the risk is to be late in solving the problem
  - if *t* is low, too many transactions are killed
  - risk of individual block (same transactions killed several times)

Deadlock recognition

- A graph (wait-for graph) is incrementally maintained by the locking scheduler: the nodes are the transactions, and the edge from Ti to Tj means that Ti is waiting for Tj to release a lock
- When a cycle appears in the graph, the deadlock is solved by killing one of the involved transactions, for example the one that made the fewer operations (individual block is a risk)
- Example: $sl_1(X)$ $sl_2(Y)$ $r_1(X)$ $r_2(Y)$ $xl_1(Y)$ $xl_2(X)$



Deadlock prevention: wait-die

To each transaction Ti a priority pr(Ti) is assigned (for example, a number indicating how old is the transaction), in such a way that different transactions have different priorities

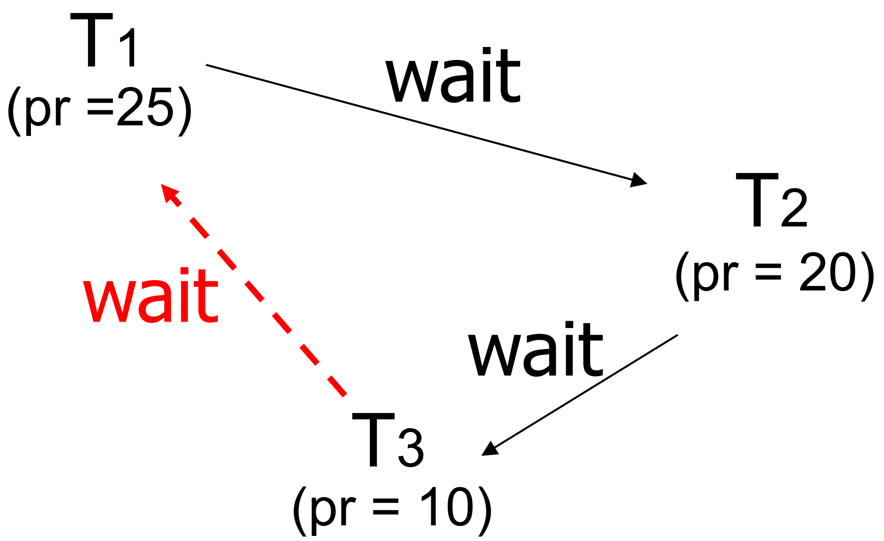
The following rule is applied by the locking scheduler: in case of conflict on a lock, Ti is allowed to wait for Tj only if Ti has greater priority, i.e., if pr(Ti) > pr(Tj), otherwise Ti is killed.

In practice, when a new edge Ti → Tj is added in the wait-for graph:

- if pr(Ti) > pr(Tj): ok
- if pr(Ti) <= pr(Tj): Ti is killed

Example of wait-die

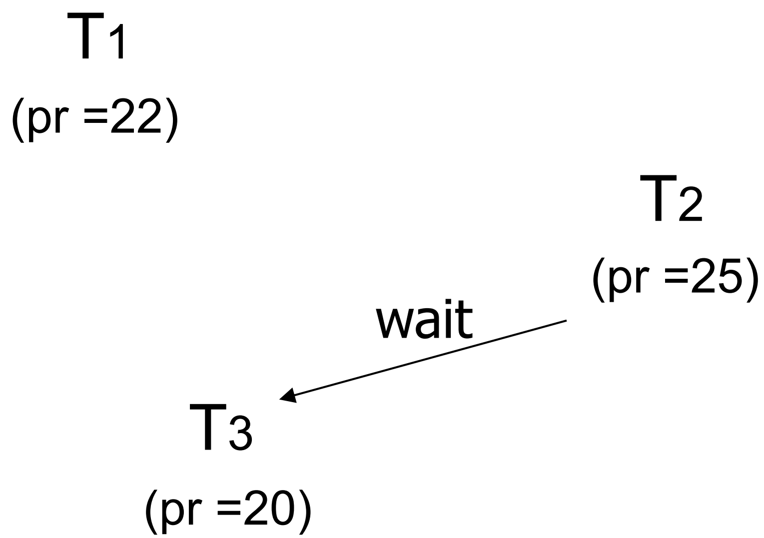
…	
xl1(Y)	T1 uses Y
xl3(X)	T3 uses X
xl2(X)	T2 waits for T3
xl1(X)	T1 waits for T2
xl3(Y)	T3 killed



T3 killed

Example of wait-die

…
xl3(X) T3 uses X
xl2(X) T2 waits for T3
xl1(X) T1 waits for T2 and for T3 ?



Note that pr(T1) > pr(T3), and pr(T1) < pr(T2). If we allow T1 to wait for T3, we have two options when T3 releases the lock on X:
(1) T1 proceeds – in this case T2 will wait for T1, with the risk of starvation;
(2) T2 proceeds, and T1 waits for T2 – this violates the rule that only trasactions with higher priorities wait.
So the right choice is to kill T1.

5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

The rollback problem

So far, we have carried out our study under the assumption that no transaction are rollbacked. Now, we relax this strong assumption, and we study 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 type of 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

We conclude that if a transaction T1 has read a value written from a transaction T2 and T2 rollbacks, T1 should also 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 (in other words, Tj also should rollback)
- If another transaction Th has read from Tj, Th should in turn rollback
- and so on…

This situation, called cascading rollback, should be avoided, since it causes several performance problems.

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 (T2 reads from T1 and commits after the commit of T1):

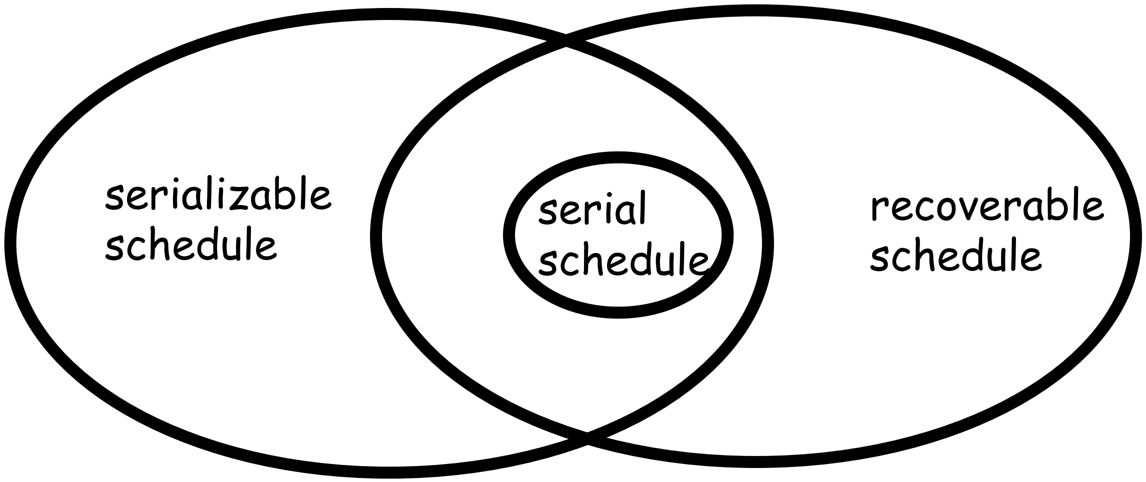
S: w1(A) w1(B) w2(A) r2(B) c1 c2

Example of non-recoverable schedule (T3 reads from T2 and commits before the commit of T2):

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 schedules 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 view-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 (the correct situation can be recovered, but in order to recover it, we may be forced to kill several transactions).

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 read anomaly.

Summing up

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

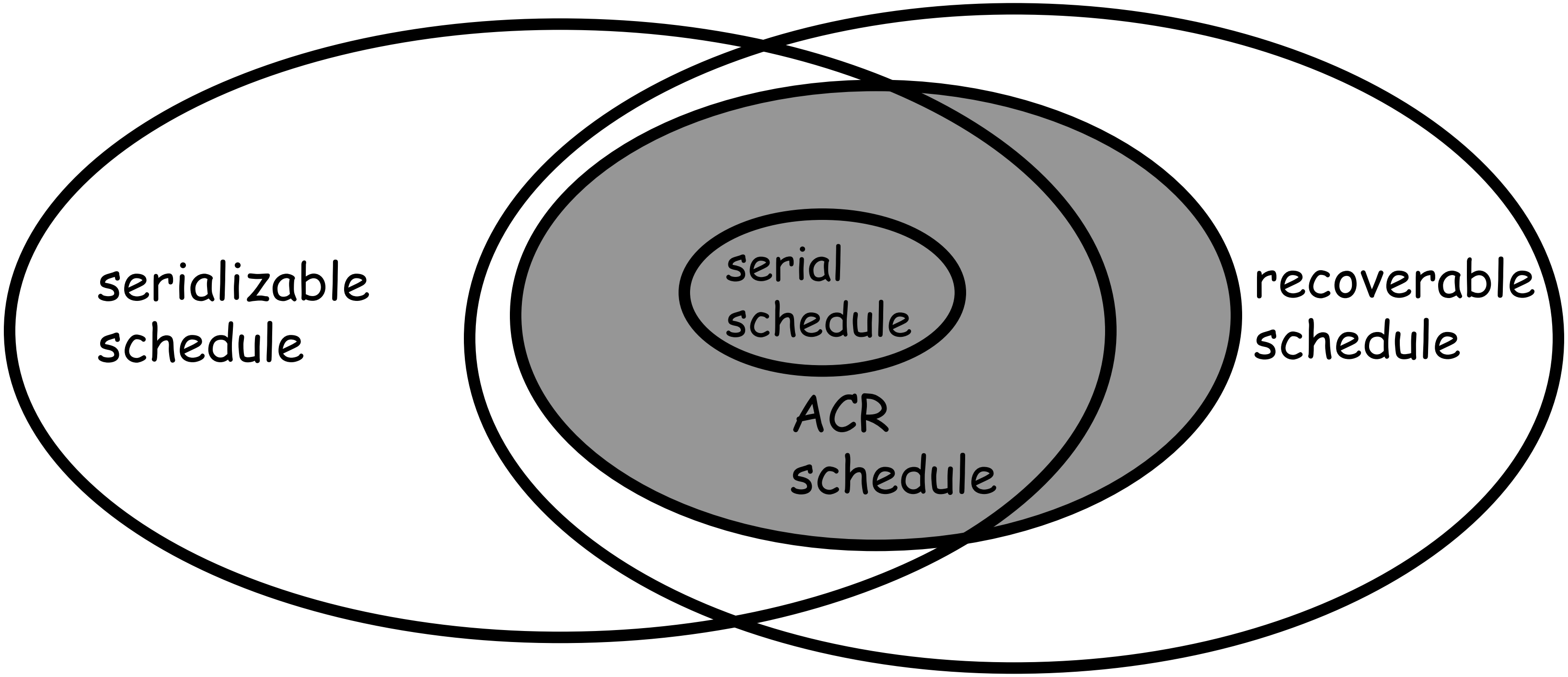
  w1(A) w1(B) w2(A) r2(B) c1 c2

- 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 both serializable and ACR.

Beyond ACR

Typically, when a transaction Ti that executed wi(X) rollbacks (or, aborts), we should restore in X the value that was stored before the action wi(X). For example, in the schedule

.... w1(x) .... w2(x) a2

we should store in X the value written by w1(X).

Now, consider the following schedule:

.... w1(X) .... w2(X) ... w3(X) a2

When T2 aborts, we cannot simply put in X the value that was stored in it before transaction T2: indeed, in this case, we should simply leave the value written by w3(x).

This situation causes a lot of complexity to the scheduler. In order to simplify the management of rollbacks, the notion of “strict schedule” has been introduced.

Strict schedules

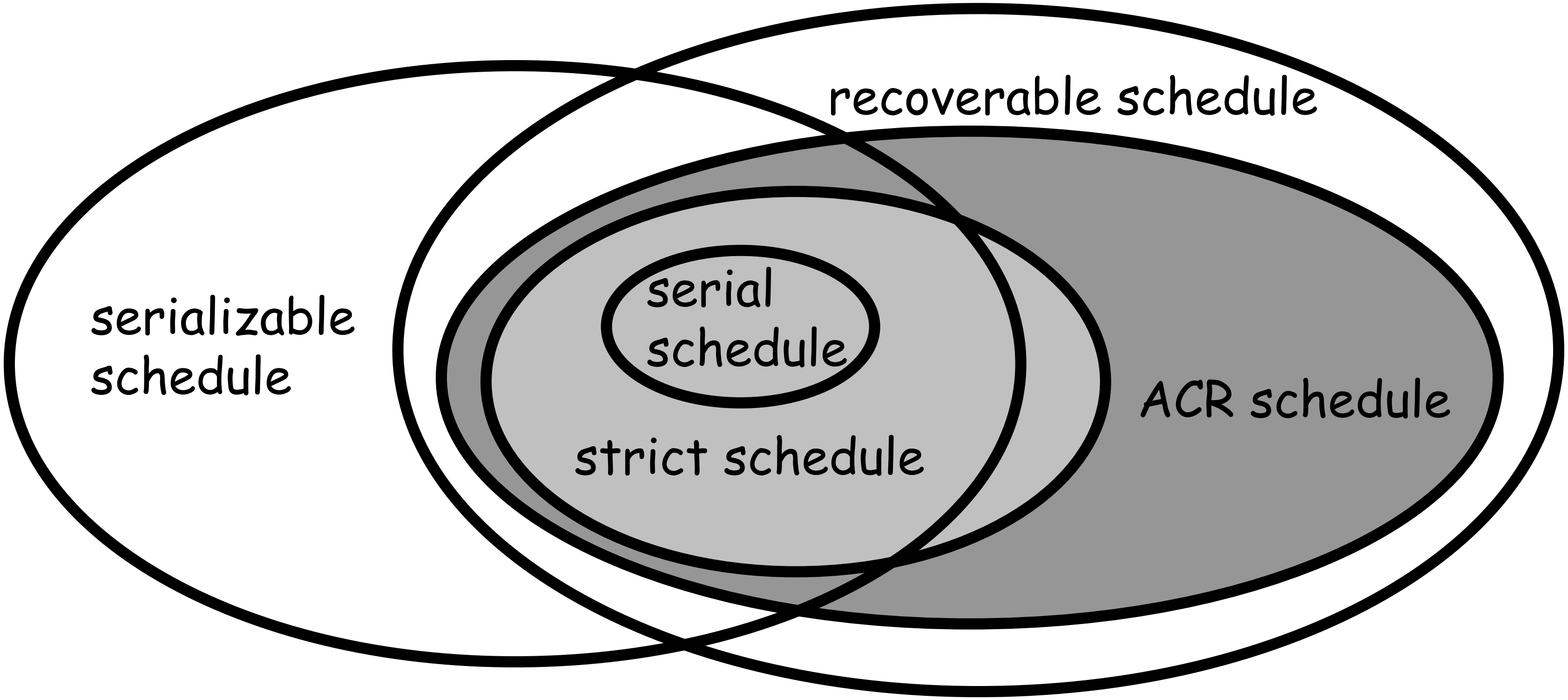
- We say that, in a schedule S, a transaction Ti writes on Tj if there is a wj(A) in S followed by wi(A), and there is no write action on A in S between these two actions

- 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.

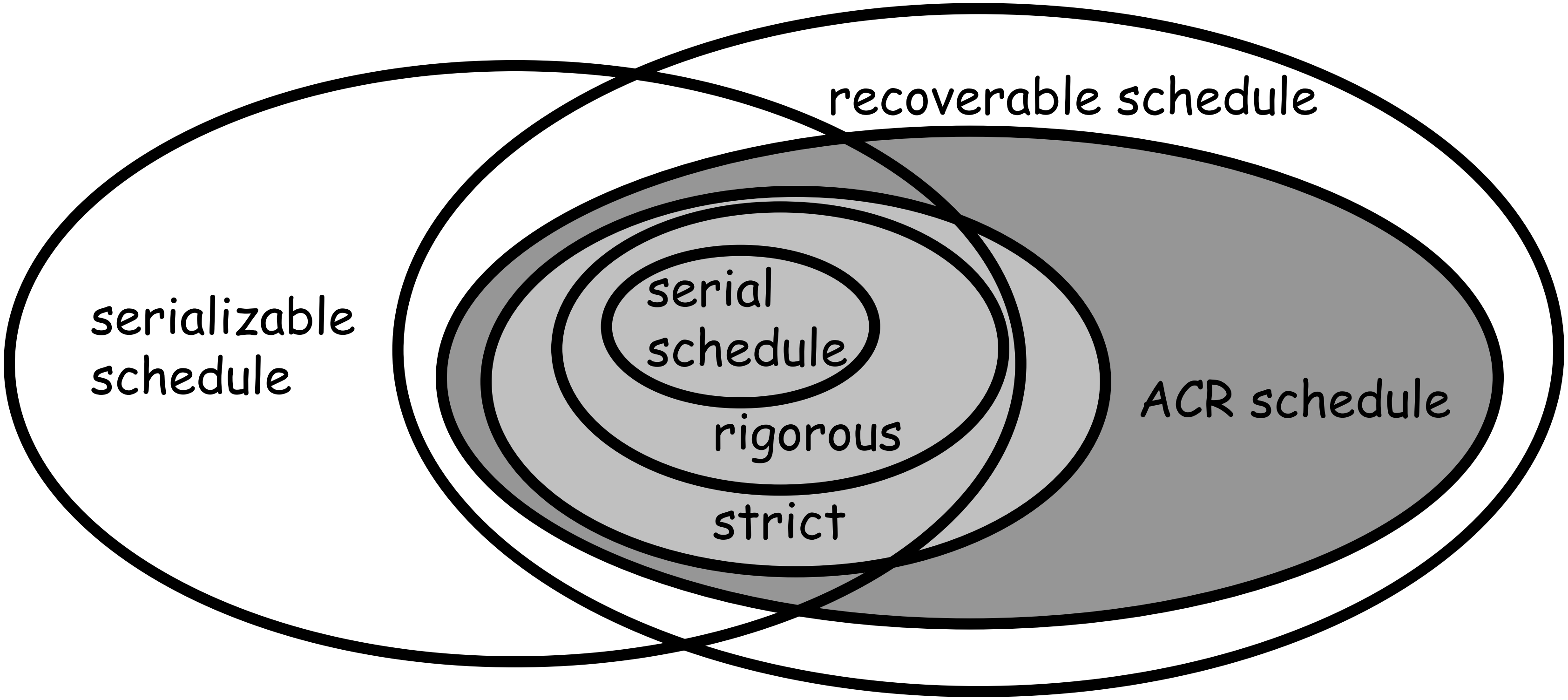
Rigorous schedules

- Although strict schedules have good properties, they do not ensure conflict serializability (prove it!)

- We say that a schedule S is rigorous if for each pair of conflicting actions ai (belonging to transaction Ti) and bj (belonging to transaction Tj) appearing in S, with ai appearing before bj, the commit command ci of Ti appears in S between ai and bj.

- It is easy to verify that every rigorous schedule is strict and ensures commit order conflict serializability

Strict schedules and ACR



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

The classes of recoverable, ACR, strict and rigorous schedules

Let C denote any of the classes recoverable, ACR, strict and rigorous schedules.

We say that a schedule S belongs to class C if we can insert suitable commit commands into S (for transactions for which there is no commit or abort command already present in S) in such a way that the resulting schedule has the property C.

For instance, the schedule S: r1(y) w(y) w1(x) w2(x) is strict because we can insert the commit commands c1 and c2 in this way:

r1(y) w(y) w1(x) c1 w2(x) c2

On the other hand, the schedule S': w1(x) w2(x) r1(y) is not strict.

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)

With the goal of capturing the class in the intersection of strict schedules and 2PL schedules, 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 8)

- Every schedule following the strict 2PL protocol is serializable
  – Obvious, since every 2PL schedule is conflict-serializable!

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.

Strong strict two-phase locking (SS2PL)

A schedule S follows the strong strict 2PL protocol if it follows the 2PL protocol, and all locks of every transaction T are kept by T until either T commits or rollbacks.

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

Properties of strong strict 2PL

- Every schedule following the strong strict 2PL protocol is rigorous

  (see exercise 9)

- Every schedule S following the strong strict 2PL protocol is obviously serializable, and the commit order of S is also a conflict-serializability order. Indeed, it can be shown that every strong 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 9

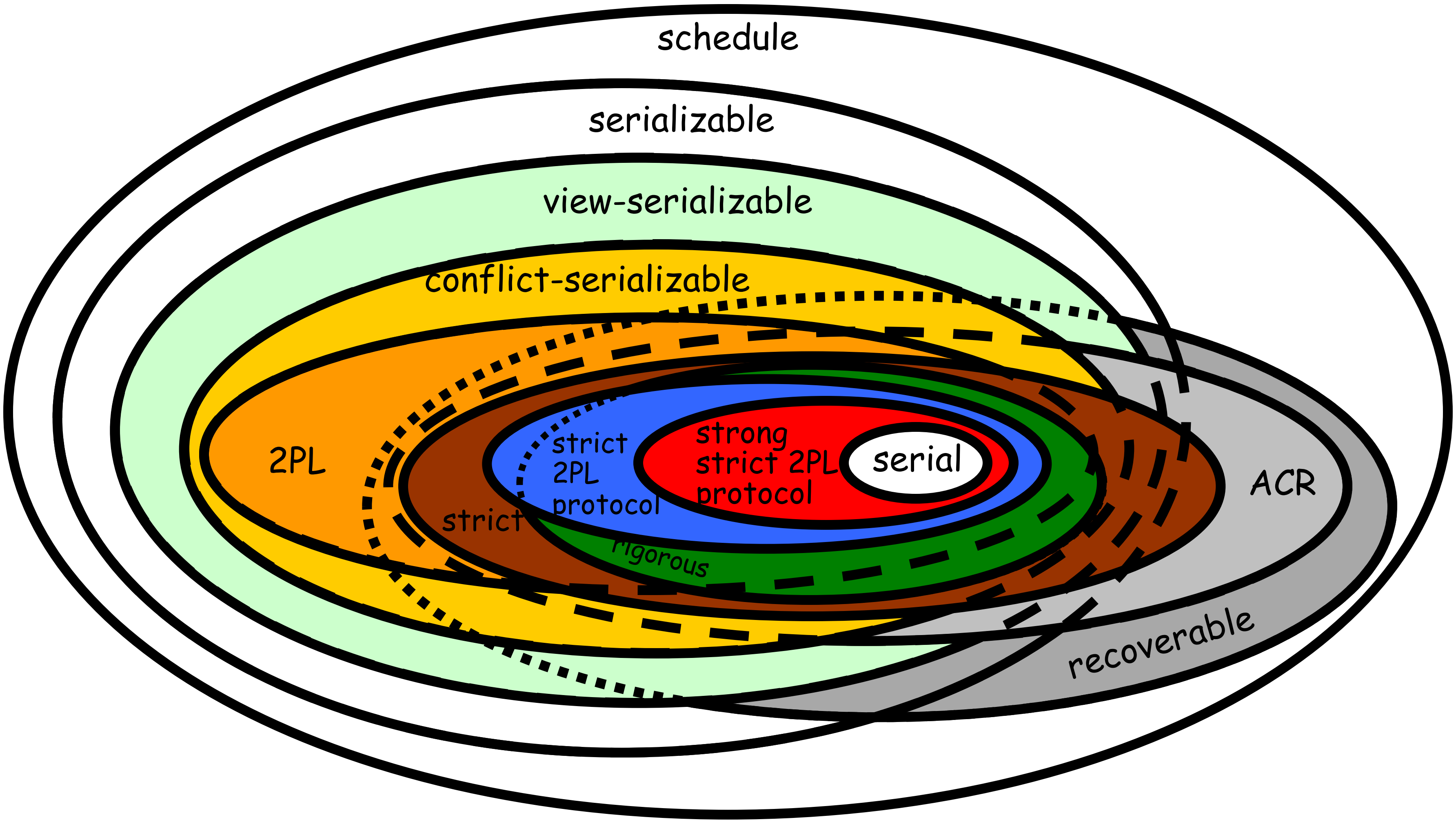
- Prove or disprove the following statement:

  Every schedule following the strong strict 2PL protocol is rigorous.

- Prove or disprove the following statement:

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

The complete picture



5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

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 Tj 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 timestamp

- 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:

for each element X written by Ti,

set cb(X) = true

for each 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:

for each 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)

for each 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

We denote by “TS” the class of schedules defined as follows:

{ S | S is the output of the timestamp-based scheduler when processing S }

In other words, the class includes exactly those schedules S for which the timestamp-based scheduler does not block any action of S.

If S is in the class “TS”, we say that it is "accepted” by the timestamp-based scheduler.

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 processed by the timestamp-based scheduler without using 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 using the Thomas rule, then S may be not conflict-serializable, like for example:

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

  However, if the schedule S is processed by the timestamp-based scheduler using 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

- 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 waiting for locks are put in waiting stage
  - TS: transactions reading too late or writing too late are killed and re-started; the waiting stage is only for transactions waiting for other transaction to commit or rollback

- 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 (when the number of conflicts is low)

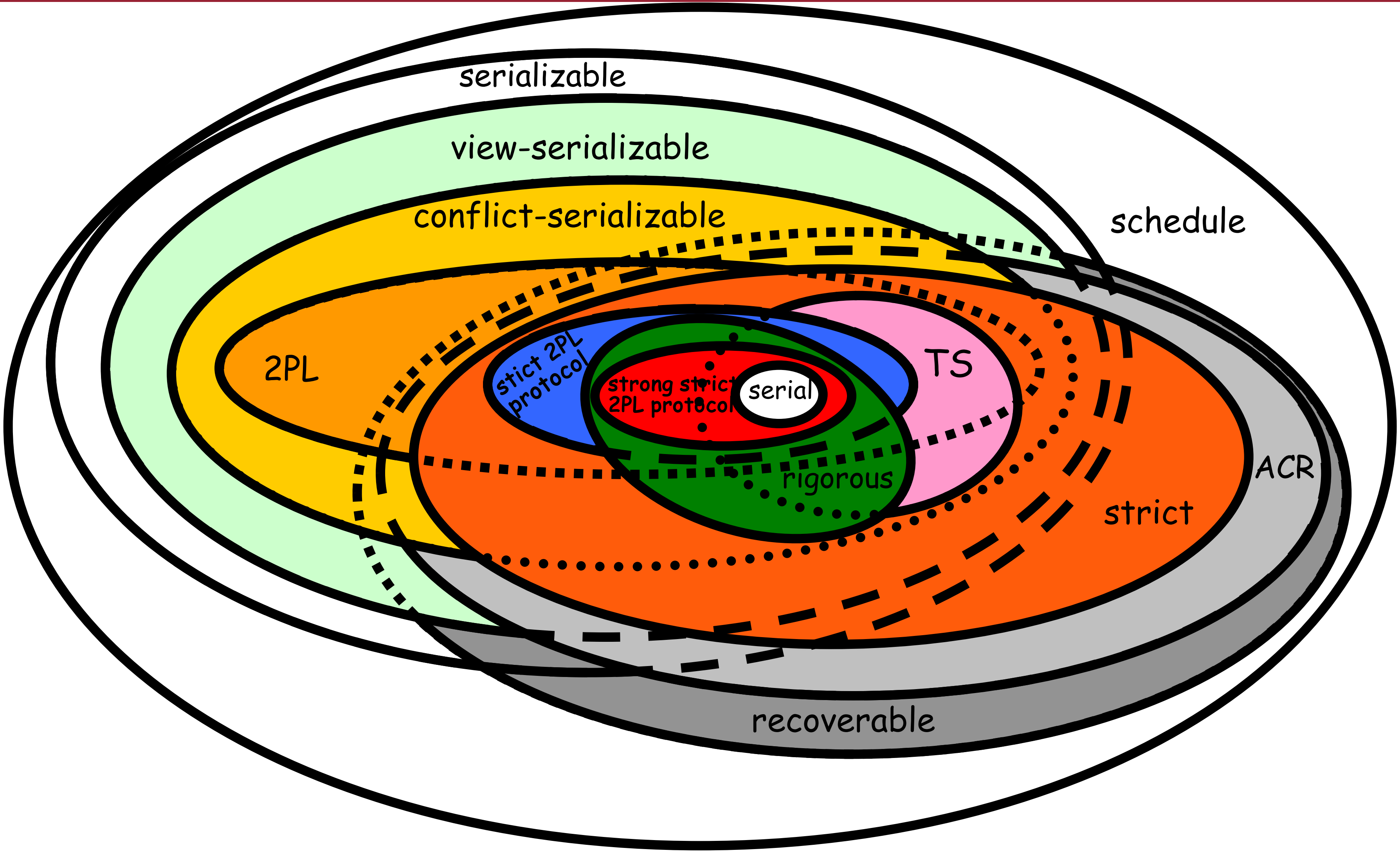
Comparison between timestamps and 2PL

- 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 systems based on timestamps

- In the following picture, 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

The final picture



5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

Multi-version concurrency control

Example 5.1:

S = $w_0(x)\ w_0(y)\ c_0\ r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Not conflict serializable

Multi-version concurrency control

Example 5.1:

$S = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Not conflict serializable

but: S would be tolerable if $r_1(y)$ could read the **old version** y_0 of y (i.e., the version written by T_0), so as to be "coherent" with $r_1(x)$ that read x_0

Multi-version concurrency control

Example 5.1:

$S = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Not conflict serializable

but: S would be tolerable if $r_1(y)$ could read the **old version** y_0 of y (i.e., the version written by T_0), so as to be "coherent" with $r_1(x)$ that read x_0

→ S would then be conflict serializable, since it would be equivalent to the serial schedule S' = $t_0\ t_1\ t_2$

The principle of the approach:

- each "legal" write action creates a new version
- each read action can choose which version it wants/needs to read, while still being "coherent"
- versions are transient (i.e., subject to garbage collection) and transparent to applications

Snapshot isolation

- It is a kind of Multiversion Concurrency Control Mechanism (used in PostgreSQL, for example)
- A transaction executing under snapshot isolation appears to operate on a personal snapshot of the database, taken at the start of the transaction
- When the transaction concludes, it will successfully commit only if the values updated by the transaction have not been changed externally since the snapshot was taken. Conversely, if such a write-write conflict occurs, it will cause the transaction to abort
- Readers never conflict with writers
- Unfortunately, snapshot isolation per se does not guarantee serializability
- We will not study snapshot isolation; rather, we concentrate in the following on another multiversion concurrency control mechanism, i.e., multiversion timestamp-based method.

Multiversion timestamp-based method

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 (i.e., not too late) 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 timestamp of the transaction that wrote that version
- To each version Xh of X, the timestamp rts(Xh)=ts(Ti) is associated, denoting the highest timestamp 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(Tk) < ts(Ti) < ts(Tj) already occurred, i.e.,
  B(Tk) ... B(Ti) ... B(Tj) .... wk(X) ... rj(X) ... wi(X)
  then the write is refused (it is a "write too late" case, because transaction Tj, that has a higher timestamp 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).
- when executing 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). For example, in the following schedule, ri(X) reads Xj
  B(Tk) ... B(Tj) ... B(Ti) .... wj(X) ... wk(X) ... ri(X)
  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 pair 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 of pairs for all versions of X:

  { 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) = [ts(Ti), ts(Ti)].

Multiversion timestamp: example

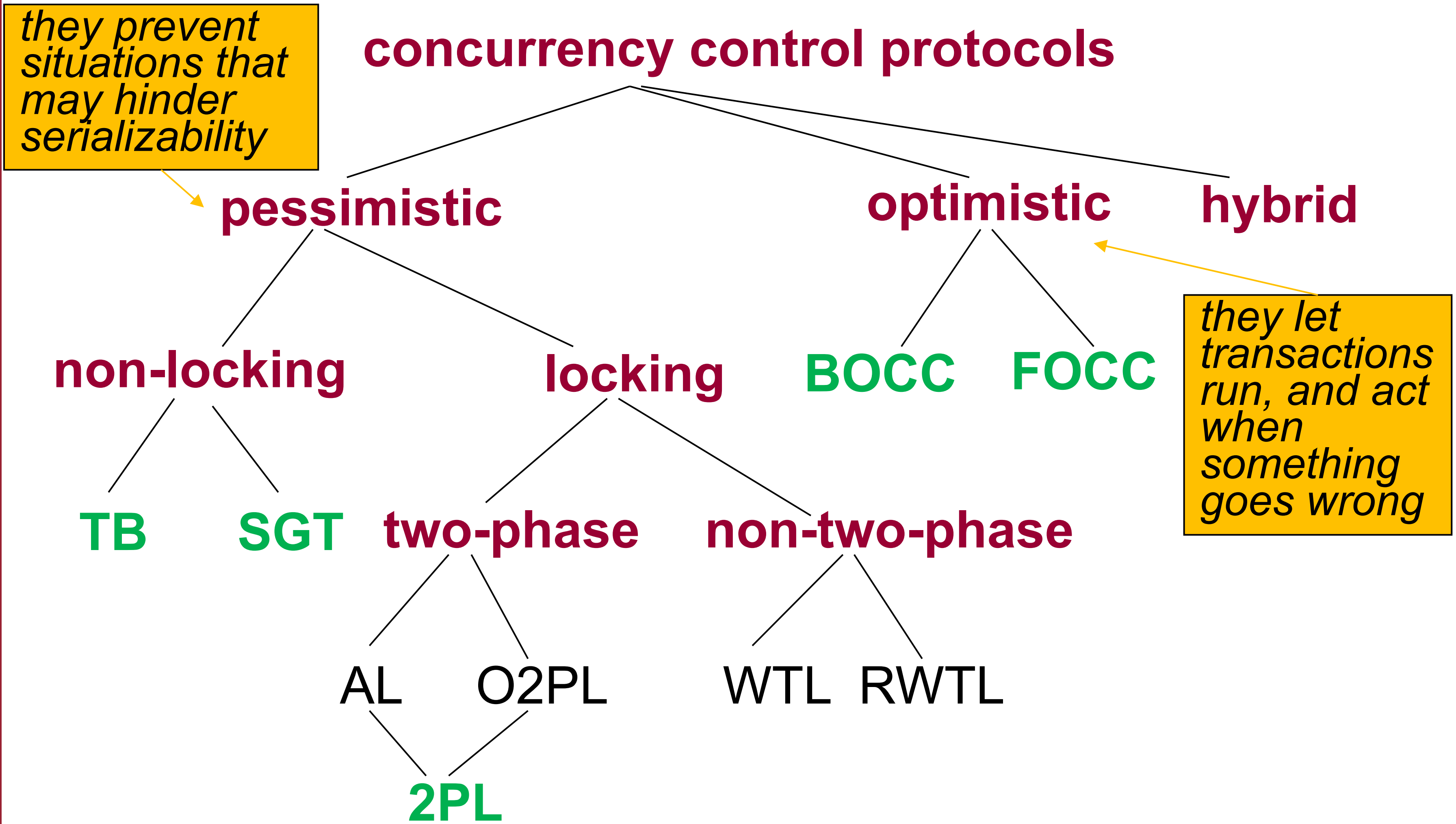
Suppose that the current version of A is A0, with rts(A0)=0.

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

5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

Scheduler classification



Optimistic Protocols

Motivation: used when conflicts are infrequent

Approach:

divide each transaction t into three phases:

- **read phase:**
  - execute transaction with writes into **private workspace**
- **validation phase (certifier):**
  - upon t's commit request
  - test if schedule remains CSR if t commits now
  - based on the read set RS(t) and the write set WS(t) of t
- **write phase:**
  - upon successful validation
    - transfer the workspace contents into the database (**deferred writes**)
  - otherwise abort t (i.e., discard workspace)

Backward-oriented Optimistic CC (BOCC)

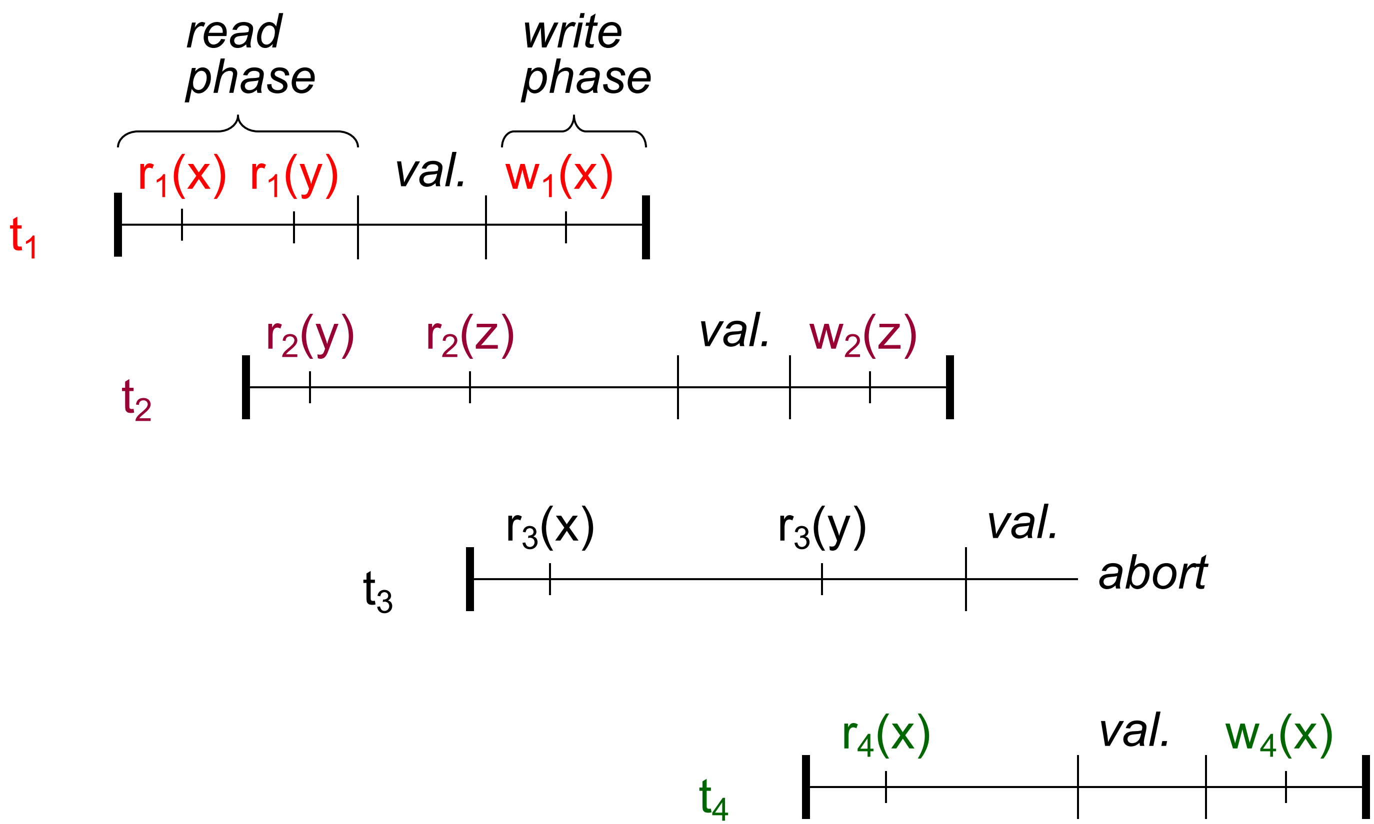
Execute a transaction's validation and write phase together as a **critical section**: while t_j being in the validation and the write phase, no other t_k can enter its validation phase

BOCC validation of t_j:
compare t_j to all previously committed t_i and
accept t_j if for each t_i previously committed one of the following holds:
- t_i has ended before t_j has started, or
- $RS(t_j) \cap WS(t_i) = \emptyset$ and t_i has been validated before t_j

Theorem
Every backward-oriented optimistic schedule is conflict serializable.

Proof:
Assume that G(s) is acyclic. Adding a newly validated transaction can insert only edges into the new node, but no outgoing edges (i.e., the new node is last in the serialization order).

BOCC Example



Forward-oriented Optimistic CC (FOCC)

Execute a transaction's validation and write phase as a **strong critical section**: while t_i being in the validation and write phase, no other t_k can perform any steps.

FOCC validation of t_j: compare t_j to all concurrently active t_i (which must be in their read phase) and accept t_j if for each t_i, $WS(t_j) \cap RS^*(t_i) = \emptyset$ where $RS^*(t_i)$ is the current read set of t_i

Remarks:

- FOCC is much more flexible than BOCC:
  upon unsuccessful validation of t_j, it has three options:
  - abort t_j
  - abort one of the active t_i for which $RS^*(t_i)$ and $WS(t_j)$ intersect
  - wait and retry the validation of t_j later
    (after the commit of the intersecting t_i)
- Read-only transactions do not need to be validated at all.

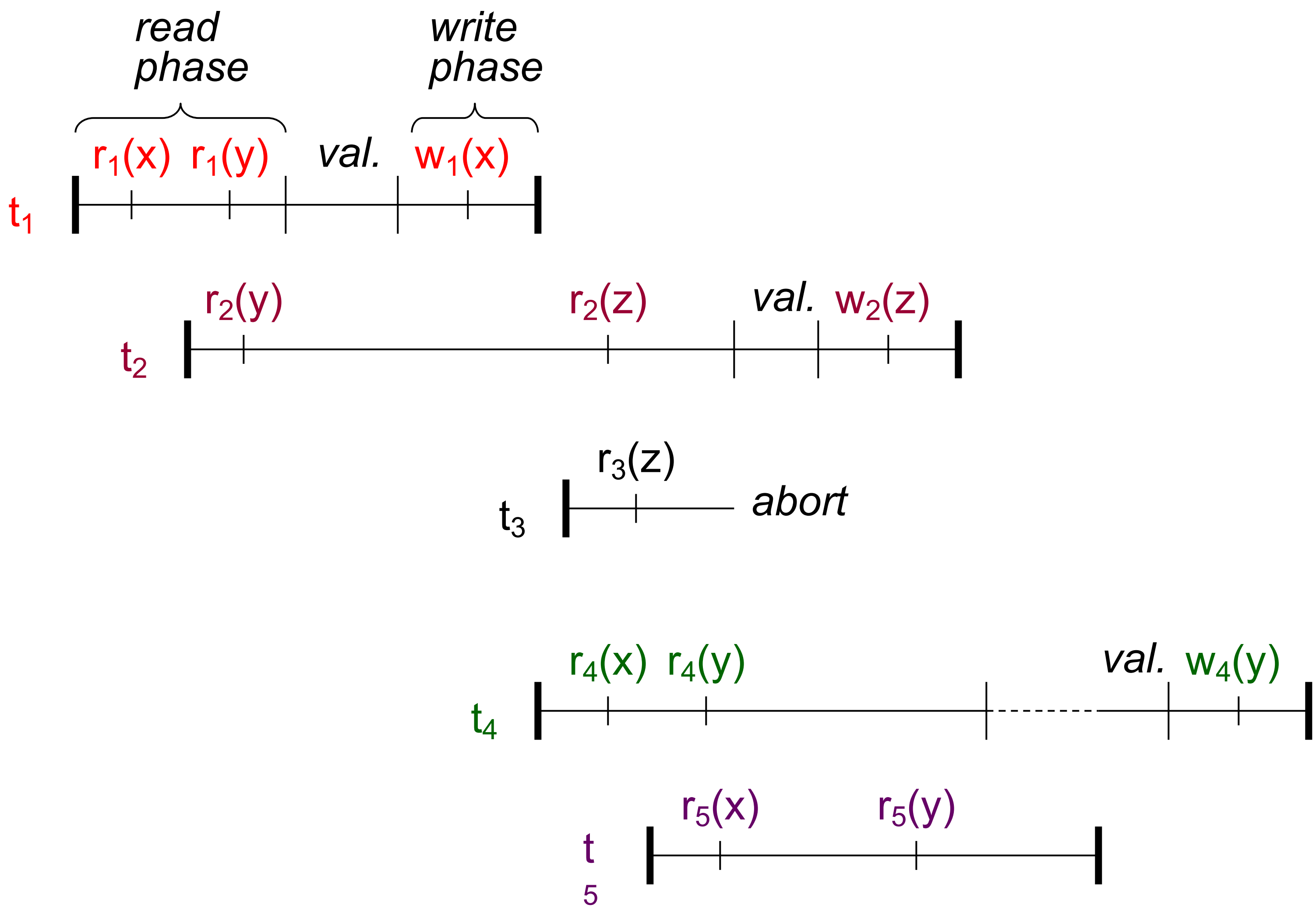
Correctness of FOCC

Theorem
Every forward-oriented optimistic schedule is conflict serializable.

Proof:
Assume that G(s) has been acyclic and that validating t_j would create a cycle. So t_j would have to have an outgoing edge to an already committed t_k. However, for all previously committed t_k the following holds:

- If t_k was committed before t_j started, then no edge (t_j, t_k) is possible.
- If t_j was in its read phase when t_k validated, then WS(t_k) must be disjoint with RS*(t_j) and all later reads of t_j and all writes of t_j must follow t_k (because of the strong critical section);
  so neither a wr nor a ww/rw edge (t_j, t_k) is possible.

FOCC Example



5. Transaction management and concurrency

5.1 Transactions, concurrency, serializability
5.2 View-serializability
5.3 Conflict-serializability
5.4 Concurrency control through locks
5.5 Recoverability of transactions
5.6 Concurrency control through timestamps
5.7 Multiversion concurrency control
5.8 Optimistic concurrency control
5.9 Concurrency control in SQL

Transactions in SQL

- The SQL engine is used by sessions, in which one can define transactions
- If in a session we are not within a transaction, then every SQL command (select, insert, update, etc.) is considered a transaction that ends when the execution of the command terminates
- To define a transaction within a session we use the BEGIN command, and the transaction will end with the COMMIT command (or, equivalently, with END), or ROLLBACK
- The ROLLBACK command undoes all actions of the transaction
- Within a transaction we may also have explicit LOCK/UNLOCK commands, if the DBMS uses locking

The anomalies considered in SQL

- Dirty read

  as we have seen, this anomaly occurs when a transaction reads an element written by a transaction that has not committed or rollbacked yet

- Nonrepeatable read

  as we have seen, this anomaly occurs when a transaction reads the same element twice

- Phantom read

  this is a new kind of anomaly, that occurs when *(i)* a transaction T1 executes a “range” query (like “select * from person where age > 10 and age < 40), *(ii)* another transaction T2 inserts or deletes tuples satisfying the range, and then, *(iii)* T1 executes again the same range query, thus finding different results. It can be avoided by “range” locks.

Concurrency in SQL - standard

Isolation Level	*Dirty Read*	*Nonrepeatable Read*	*Phantom Read*
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

- Every transaction decides its level of isolation (SET TRANSACTION ISALATION LEVEL <level>); we can always know the current isolation level (command SHOW TRANSACTION ISOLATION LEVEL)
- Except for the "Read uncommitted" level, each other level guarantees the absence of a specified set of anomalies (see table above). Serializability is the maximum level of correctness
- The standard does not impose any constraints on the implementation of the concurrency control mechanisms

Possible implementation with locking

- “Read uncommitted”: no action is taken for controlling concurrency.
- “Read committed”: a lock-based concurrency control implementation keeps write locks until the end of the transaction, but read locks are released as soon as the corresponding SELECT operation terminates. Range-locks are not managed. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data.
- “Repeatable read”: a lock-based concurrency control implementation keeps read and write locks until the end of the transaction. However, range-locks are not managed, so phantom reads can occur.
- “Serializable”: a lock-based concurrency control implementation keeps read and write locks to be released at the end of the transaction, as in strong strict 2PL. Also range-locks (locks on all possible elements satisfying the range) must be acquired when a SELECT query uses ranged WHERE clause, to avoid the phantom reads phenomenon.

NOTE: The SQL standard permits a DBMS to run a transaction at an isolation level stronger than that requested (e.g., a "Read committed" transaction may actually be performed at a "Repeatable read" isolation level).

The PostgreSQL DBMS

- The minimum isolation level is “read committed”, which is the default level
- The “repeatable read” isolation level prevents also the “phantom read” anomaly (thus ensuring “repeatable read of ranges), and therefore it strengthens the SQL standard
- The concurrency control strategy of PostgreSQL is a sort of multiversion control combined with (implicit and explicit) locking
- Postgres keeps write locks until the end of the transaction, whereas read locks are released as soon as the SELECT operation is performed
- Reads are never blocked (they read the value written by the last committed transaction), and write actions are performed on a local store (snapshot), and their effects are transferred to the database at commit
- Explicit LOCK/UNLOCK commands are also allowed
- Deadlock management is based on recognition