Data Management

Maurizio Lenzerini

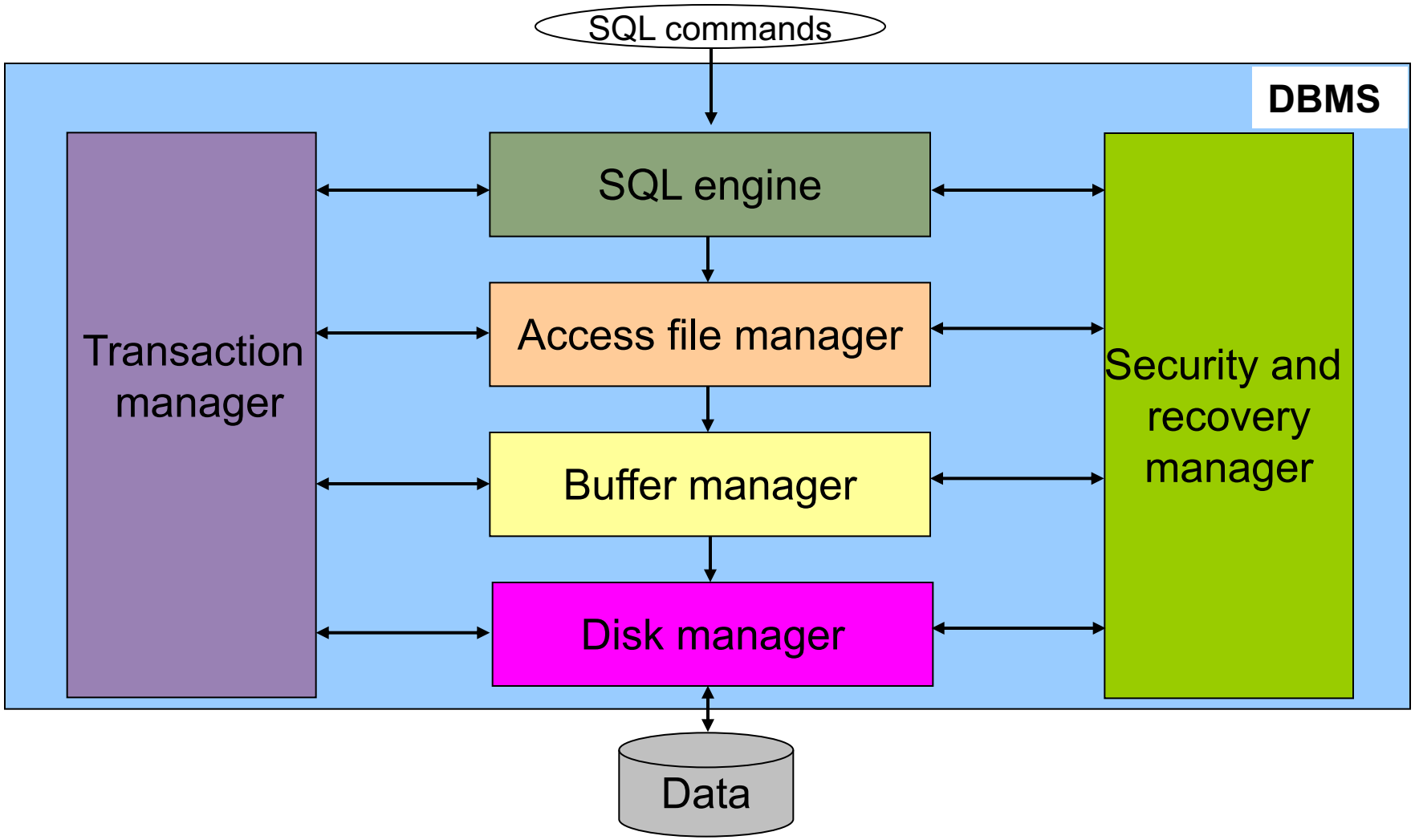
Dipartimento di Informatica e Sistemistica "Antonio Ruberti"
Università di Roma "La Sapienza"

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.

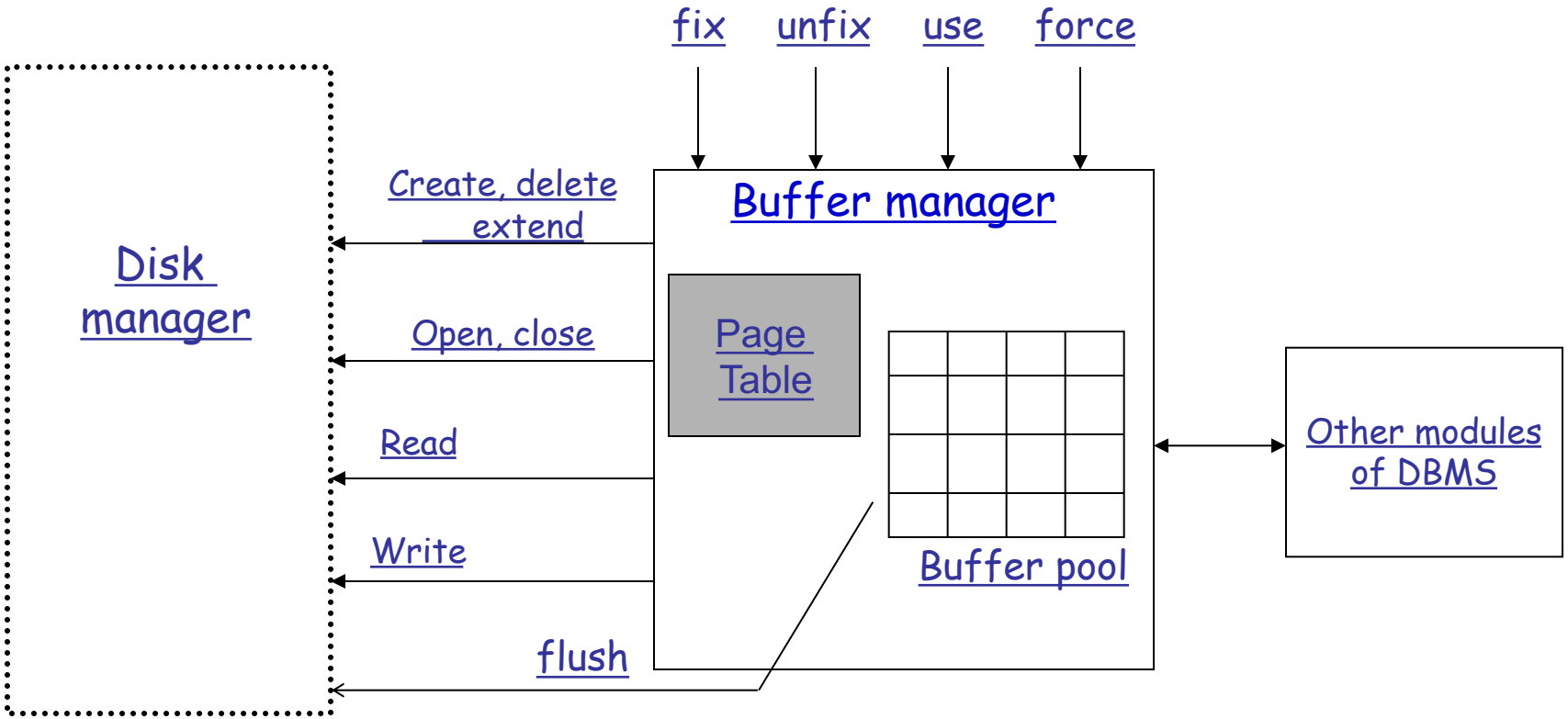
The database processes request pages from the buffer manager, whose responsibility is to minimize the number of secondary memory accesses by keeping needed pages in the buffer. Because typical database workloads are I/O-bound, the effectiveness of buffer management is critical for system performance.

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

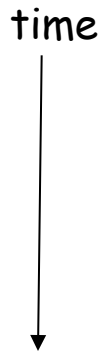
Typical applications: banks, flight reservations, web commerce… For example: Amazon is estimated to support hundreds of thousands ACID transaction per second.

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



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)

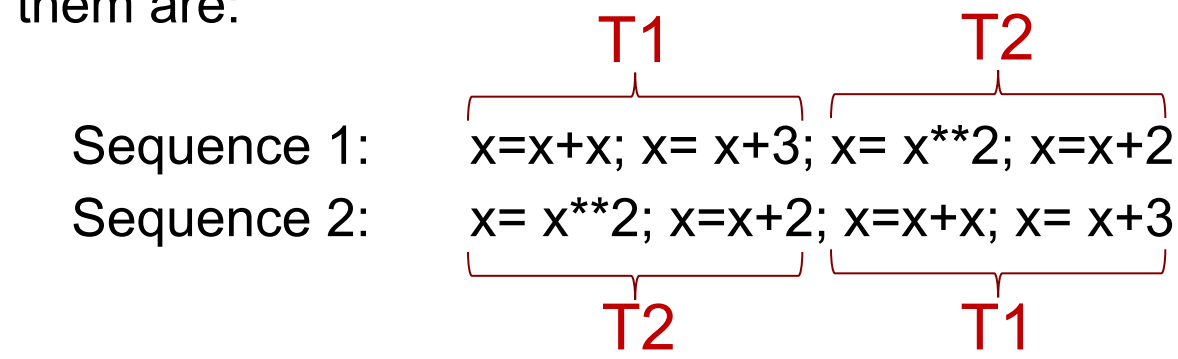
S3 is a partial schedule

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:



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	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		

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		

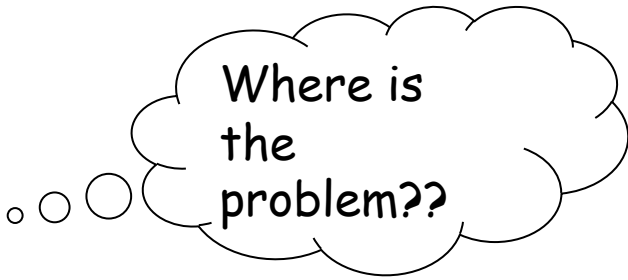
The execution of S' starting with A=B=25 leaves the values 250 for A and 150 for B.

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		



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

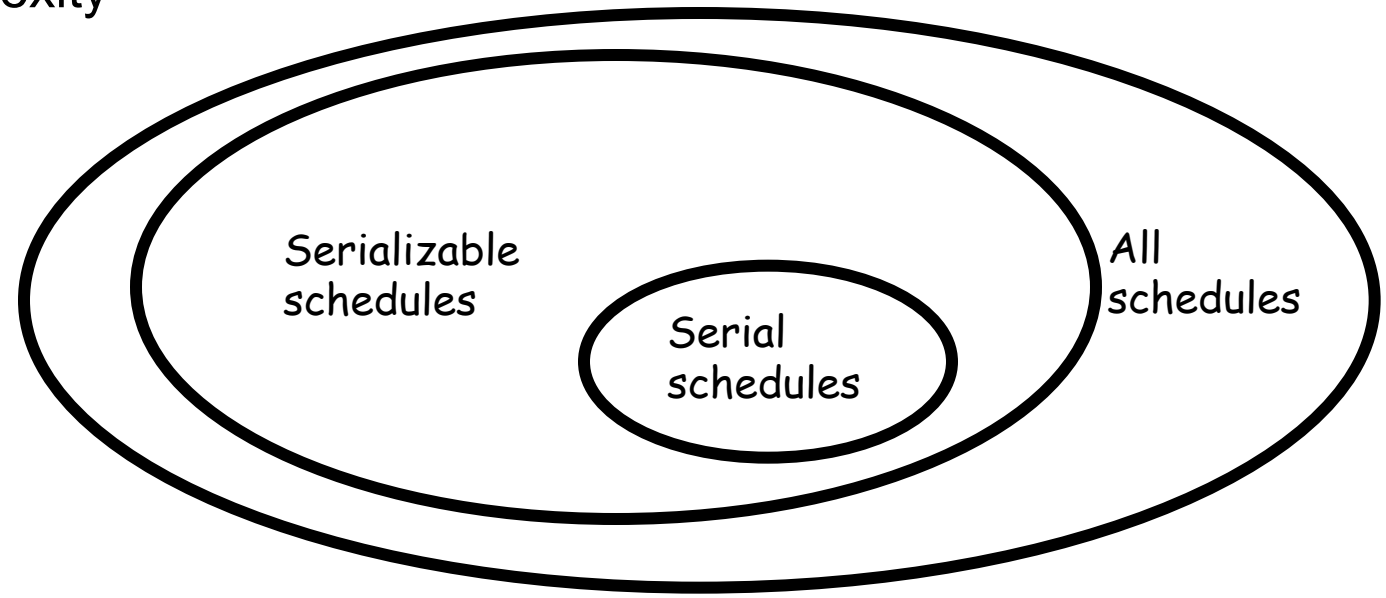
In the next slides, we will generally work under two assumptions:

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?

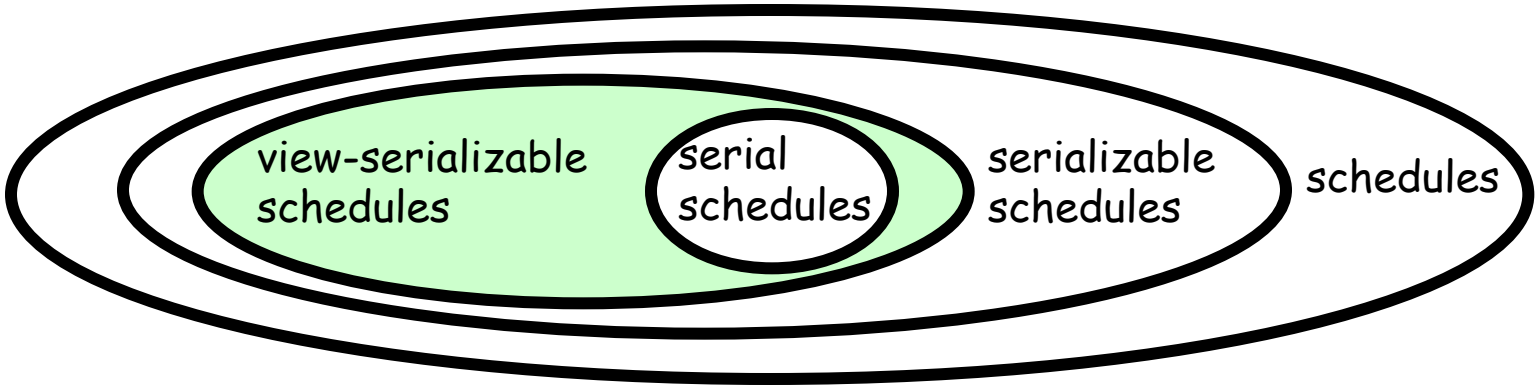
It is easy to see that the above schedule is serializable but is NOT view-serializable.

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



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:

$$p_1\ q_1\ m_2\ \underline{s_2\ t_1}\ v_1\ v_2\ t_2$$

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 s2 t1 v1 v2 t2

p1 q1 m2 t1 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 s2 t1 v1 v2 t2

p1 q1 m2 t1 s2 v1 v2 t2

p1 q1 t1 m2 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 s2 t1 v1 v2 t2

p1 q1 m2 t1 s2 v1 v2 t2

p1 q1 t1 m2 s2 v1 v2 t2

p1 q1 t1 m2 v1 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 s2 t1 v1 v2 t2

p1 q1 m2 t1 s2 v1 v2 t2

p1 q1 t1 m2 s2 v1 v2 t2

p1 q1 t1 m2 v1 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 s2 t1 v1 v2 t2

S2: p1 q1 m2 t1 s2 v1 v2 t2

S3: p1 q1 t1 m2 s2 v1 v2 t2

S4: p1 q1 t1 m2 v1 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) w2(A) r1(B) w1(B) r2(B) w2(B)

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

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

r1(A) w1(A) r1(B) r2(A) w1(B) 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.

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?

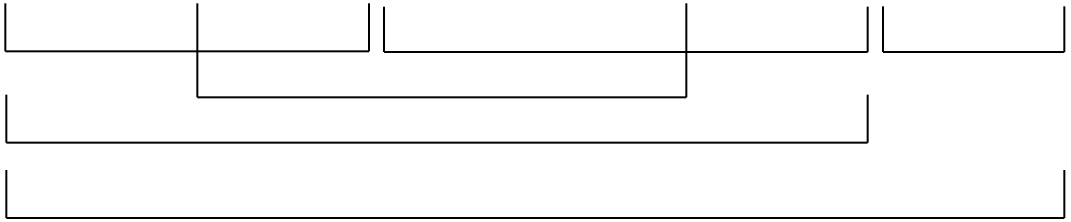
Yes, we can do it by an algorithm based on the so-called precedence graph (also called conflict graph) associated to a schedule.

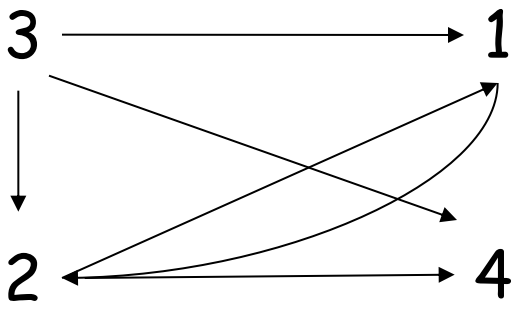
Given a schedule S on {T1,…,Tn}, the precedence graph P(S) associated to S is defined as follows:

- 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:

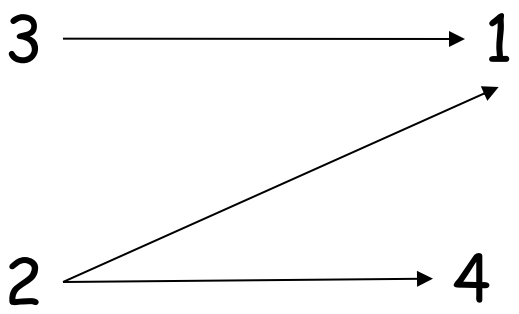
S1 = w1(A) r2(A) w2(B) r1(B)
S2 = r2(A) w1(A) r1(B) w2(B)

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 → 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 → 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 → 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)

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.

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.

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.

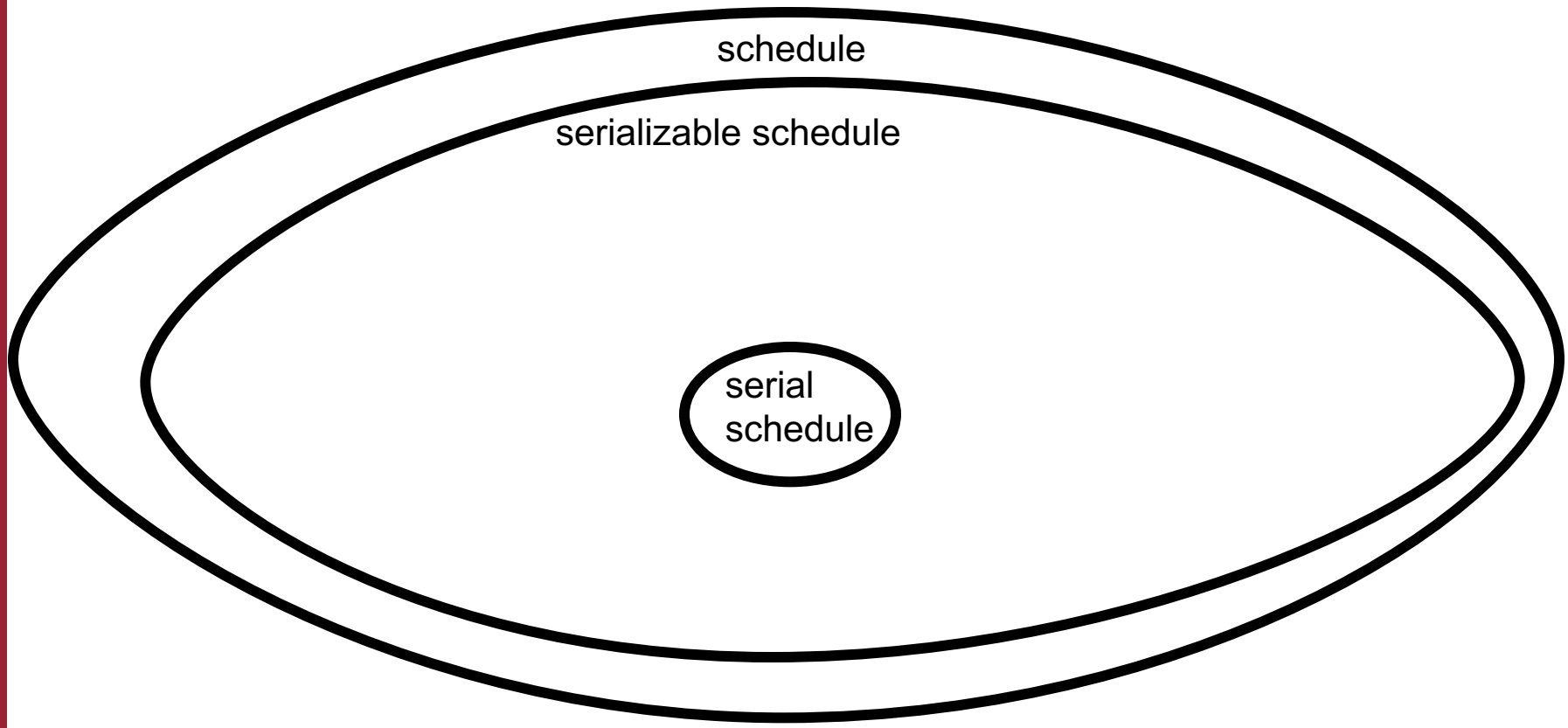
Theorem
A schedule S is in COCSR iff there is a serial schedule S' conflict equivalent to S such that for all t_i, $t_j \in$ tran(S): t_i precedes t_j in S' if and only if c_i precedes c_j in S.

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

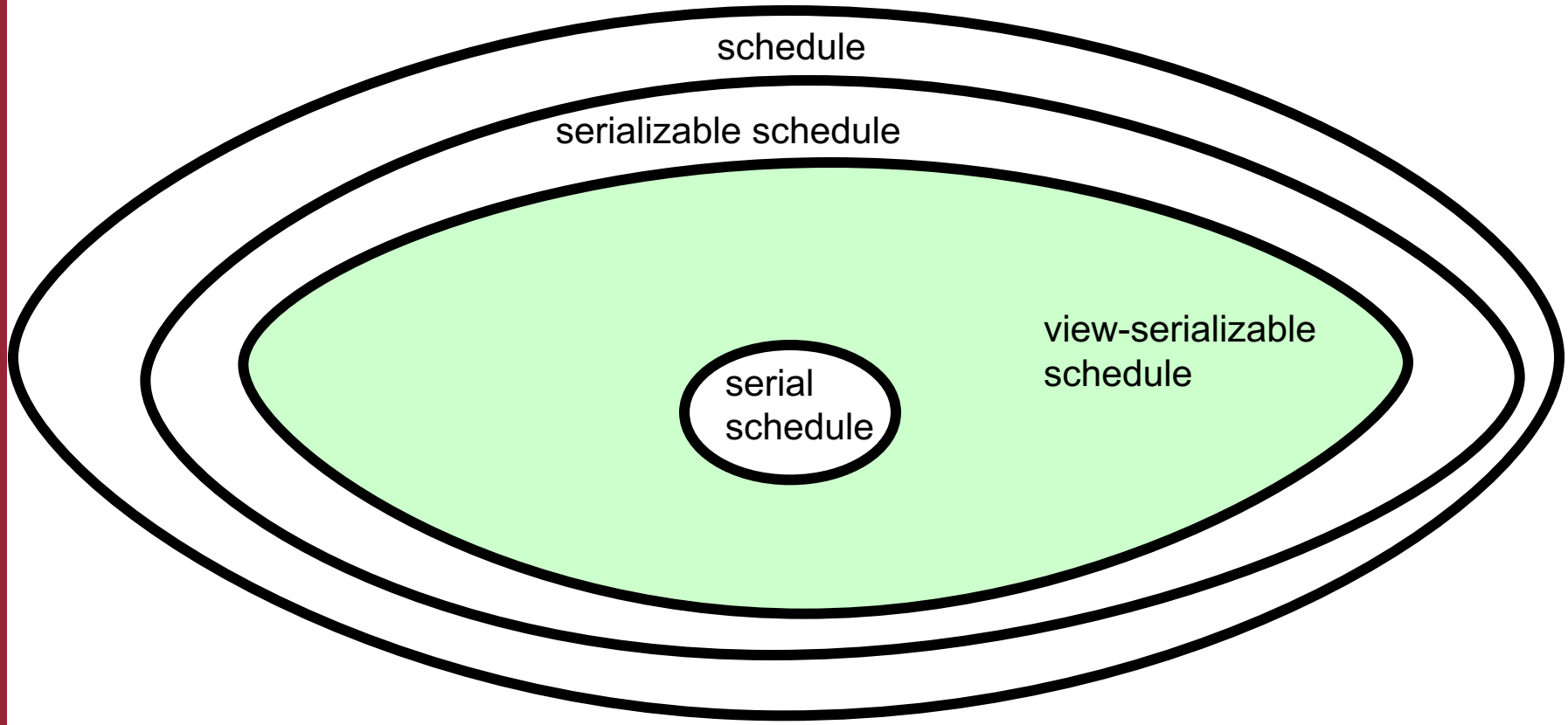
View-serializability and conflict-serializability

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



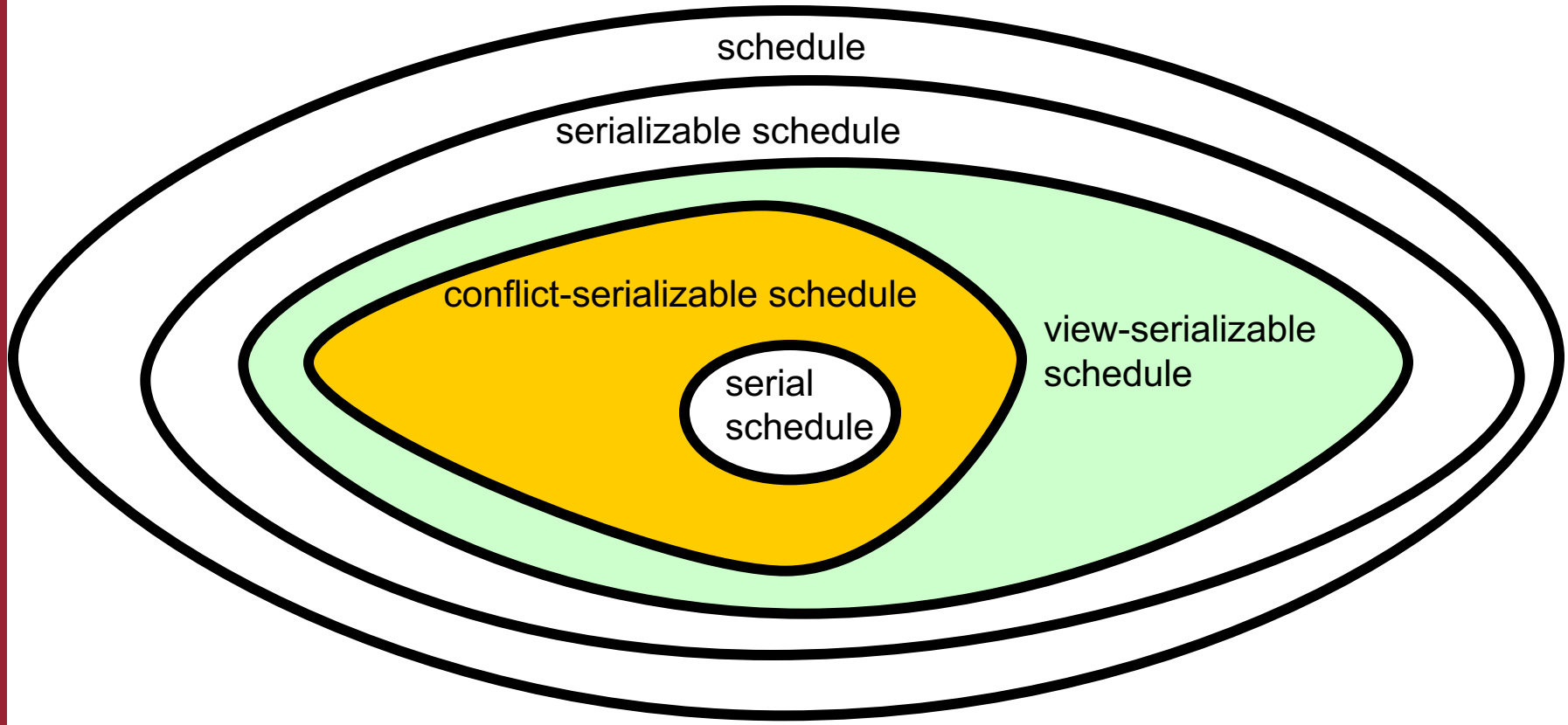
View-serializability and conflict-serializability

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



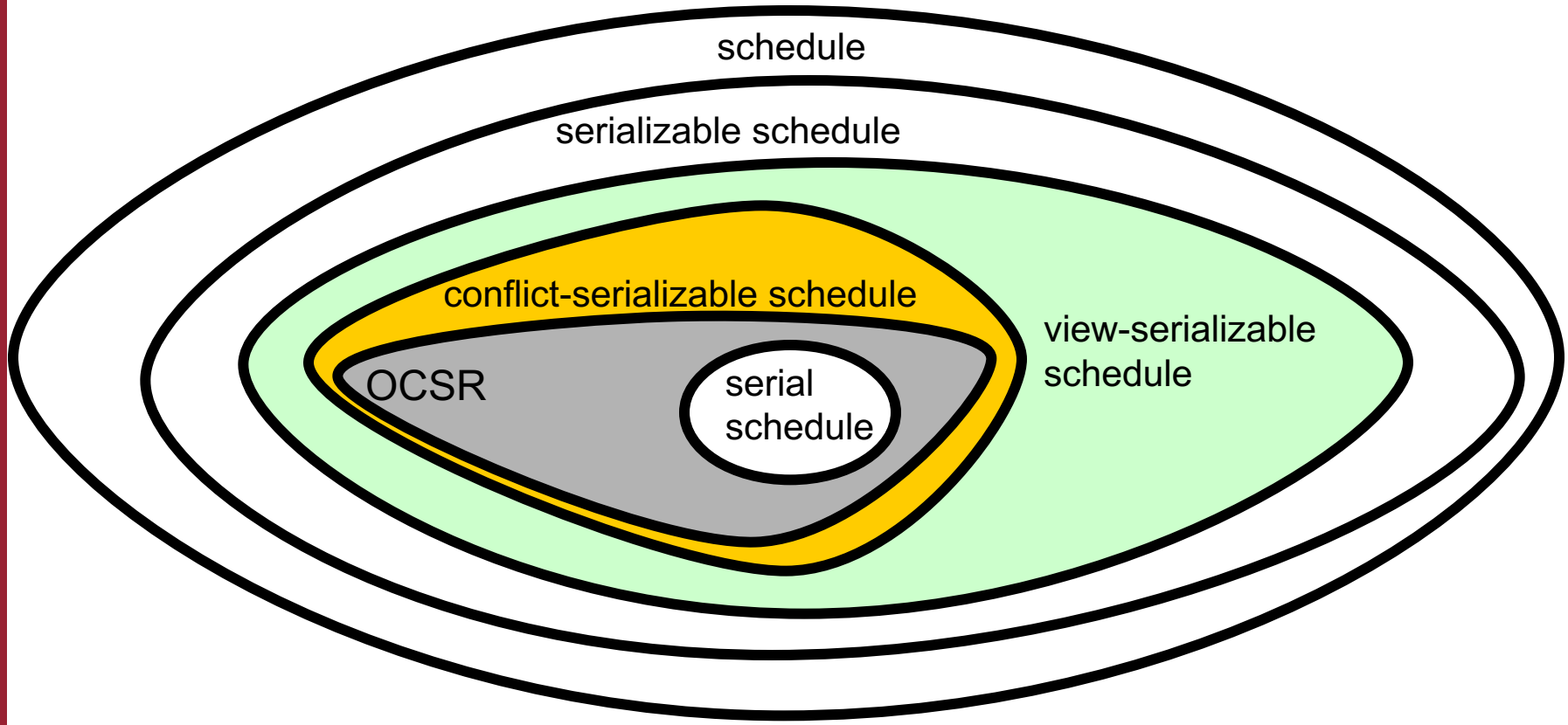
View-serializability and conflict-serializability

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



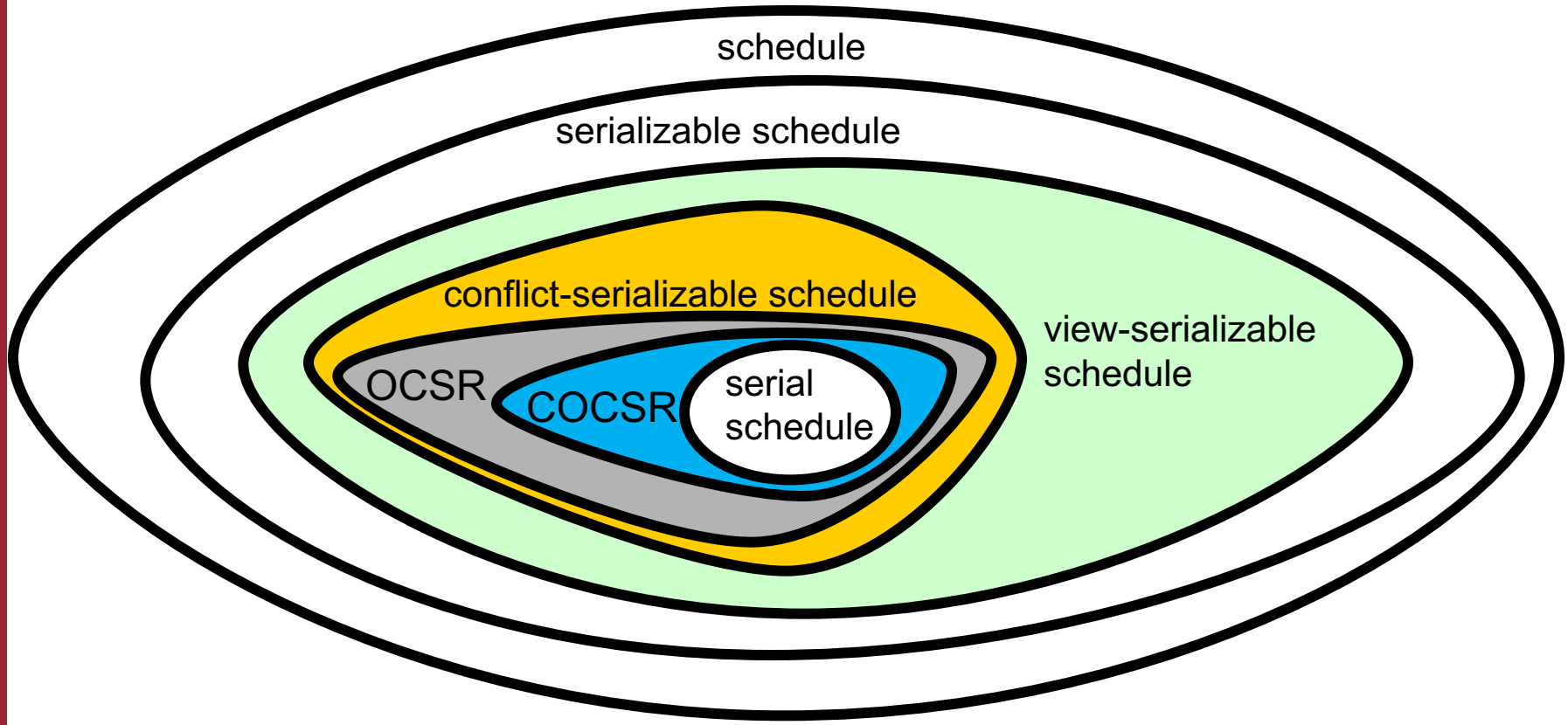
View-serializability and conflict-serializability

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:



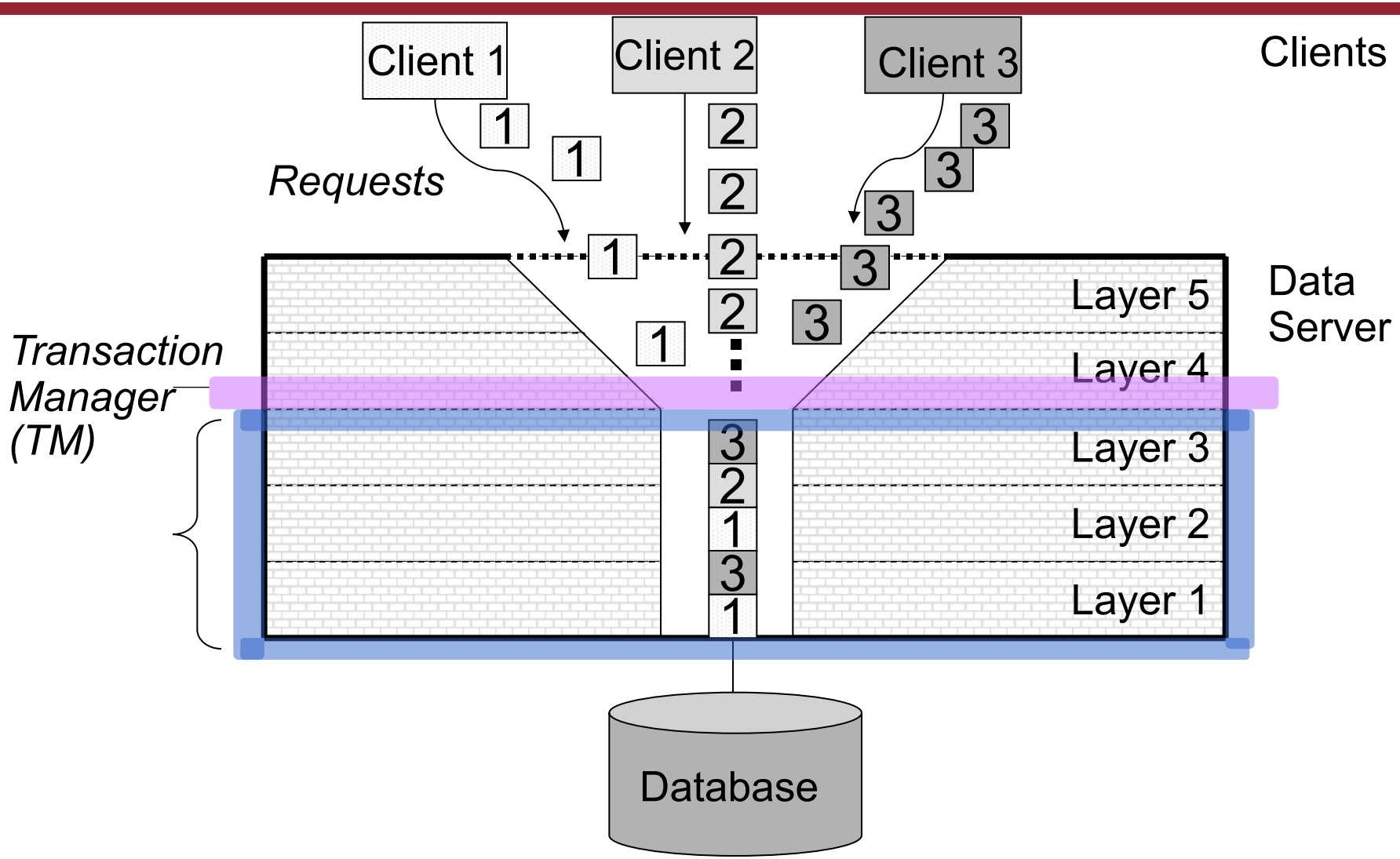
Algorithms for concurrency control

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.

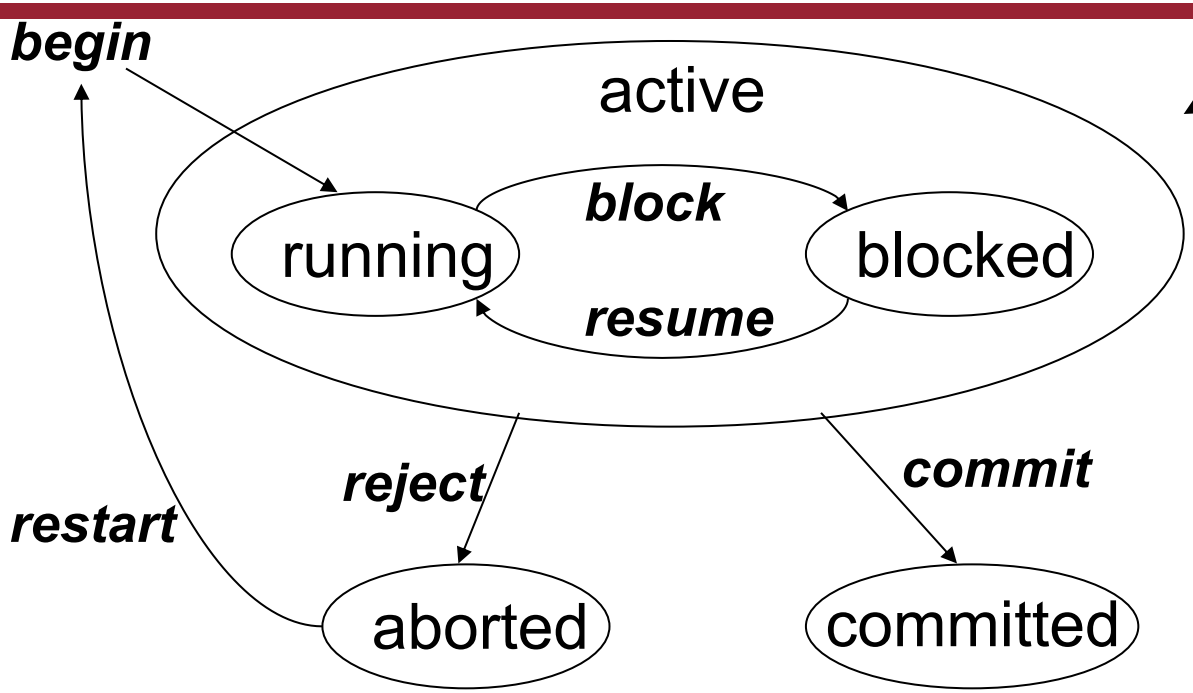
In practice, however, the “transaction and concurrency control manager” must provide an algorithm (also called **protocol**) for concurrency control. Such an algorithm corresponds to the method implememented in the scheduler, one of the basic modules of the concurrency control manager.

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

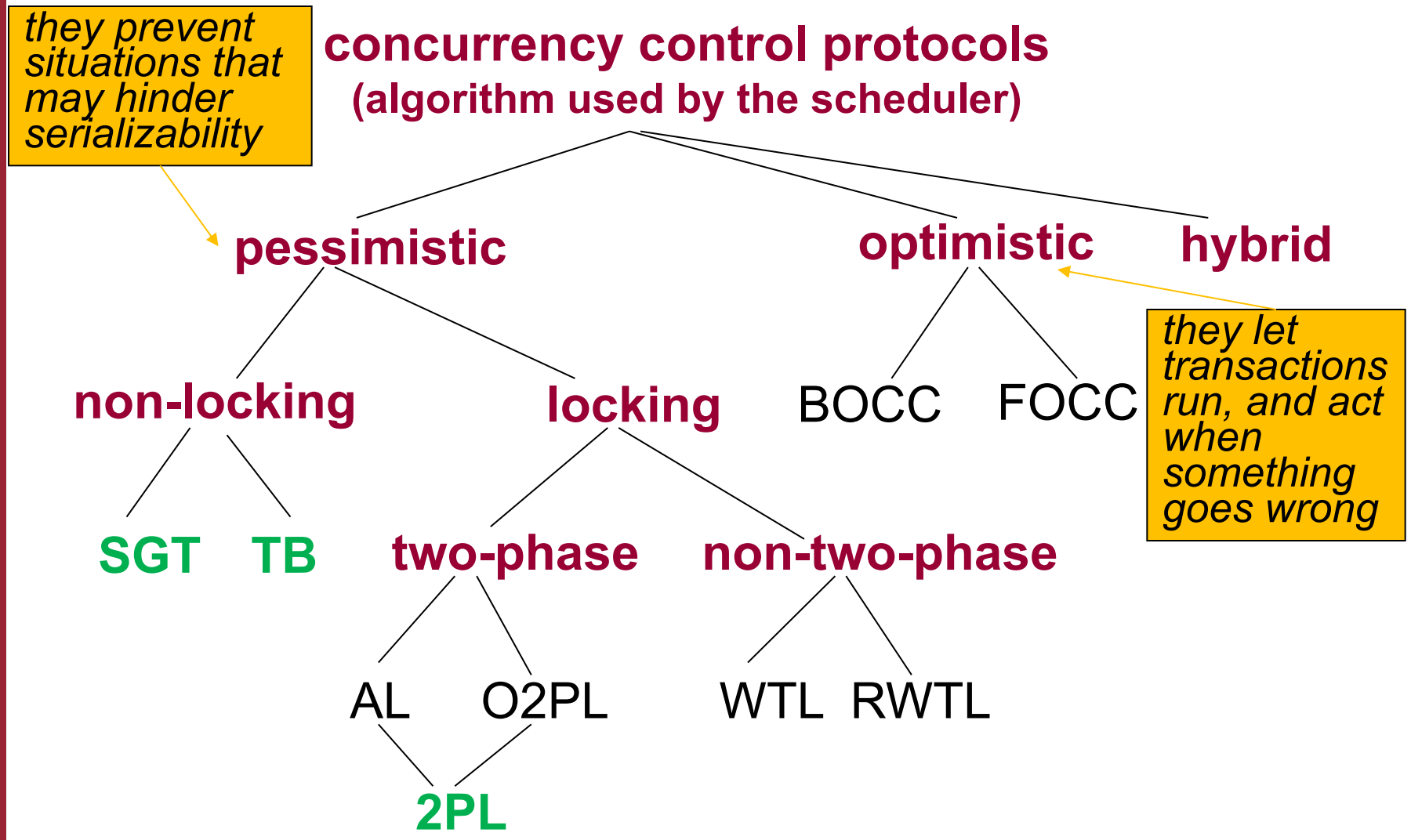


Finite state automaton for one transaction, where states are transaction states and transitions are scheduler actions performed on the transaction

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

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?