Distributed Systems

13 Distributed Transactions

Virtual Lecture
Part of other Lectures

June 29 2009

Gerd Liefländer
System Architecture Group

Schedule of Today

Transactions in Local systems

Characteristic of Transactions

Serializability

Two Phase locking Protocol

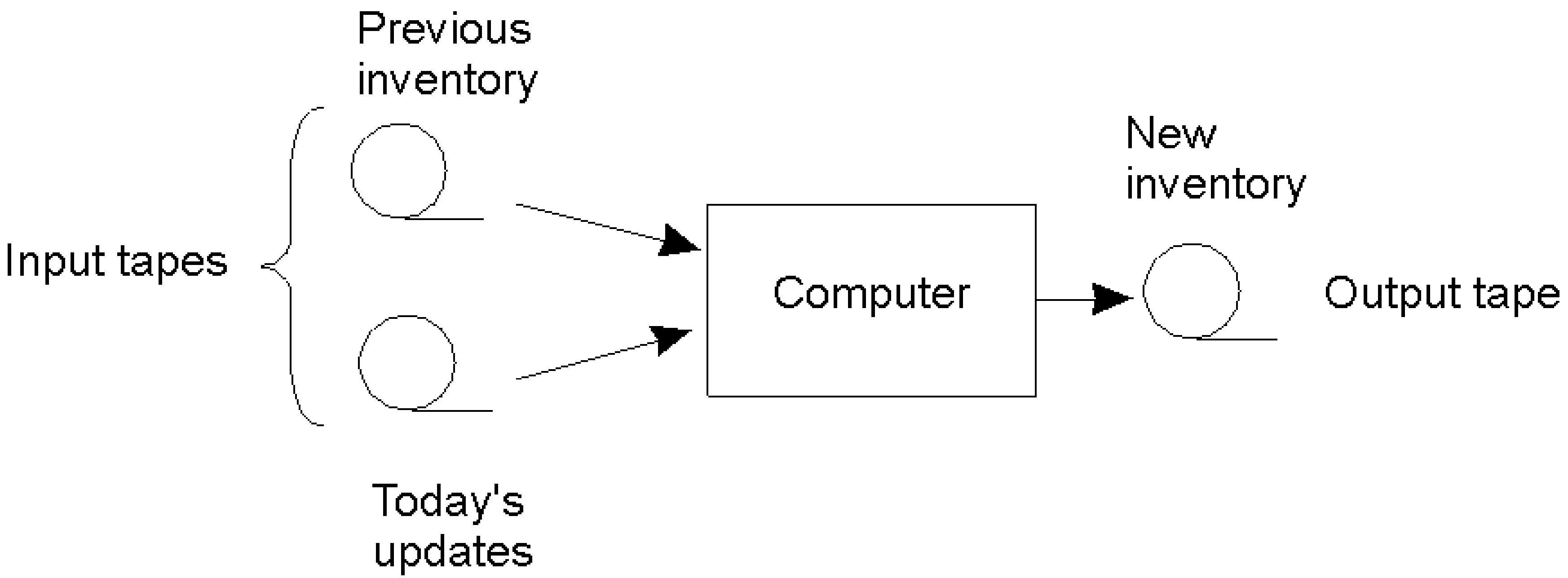
Distributed Transactions

How to support distributed applications on Data Bases

The above topics will not be examined in this course

The Transaction Model (1)

- Updating a master tape is fault tolerant.



The Transaction Model (2)

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

- Examples of primitives for transactions.

The Transaction Model (3)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

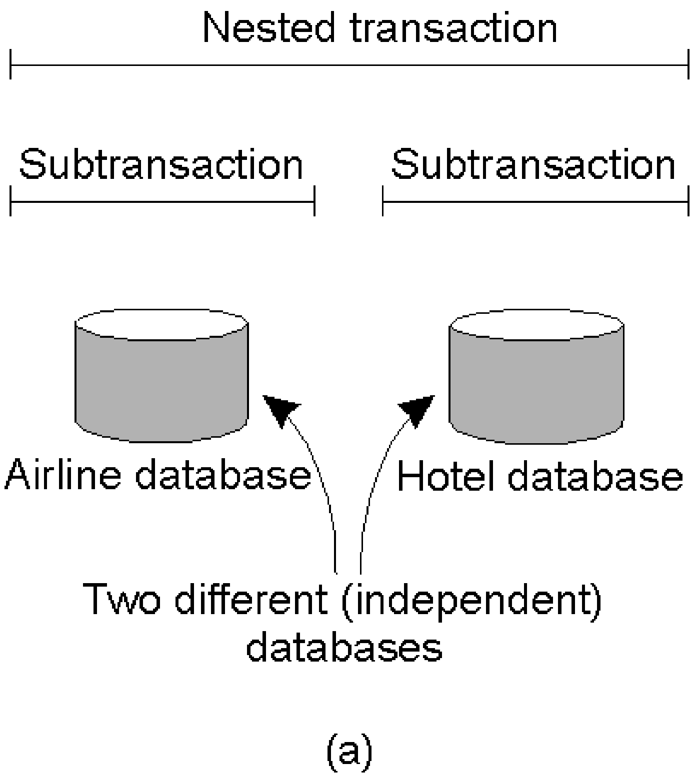
```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

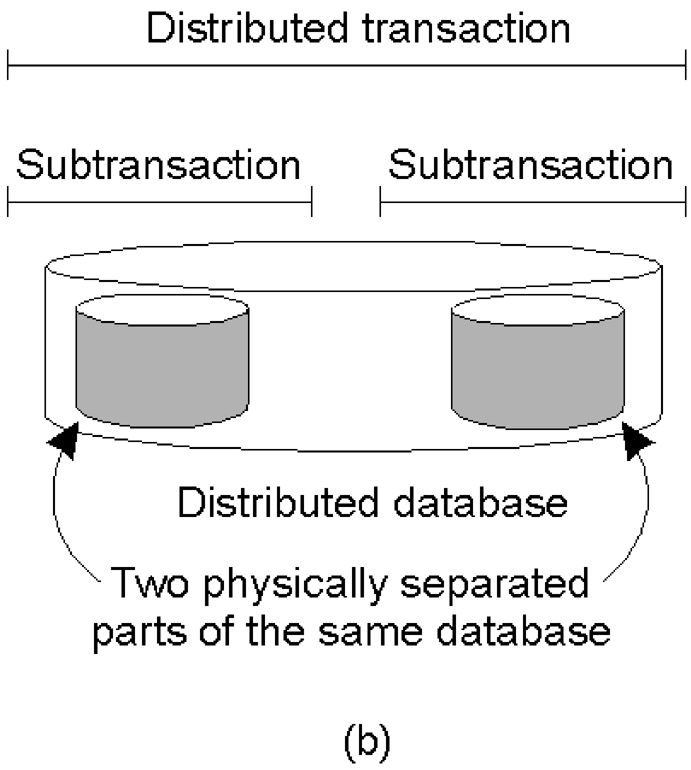
(b)

a) Transaction to reserve three flights commits

b) Transaction aborts when third flight is unavailable

Distributed Transactions

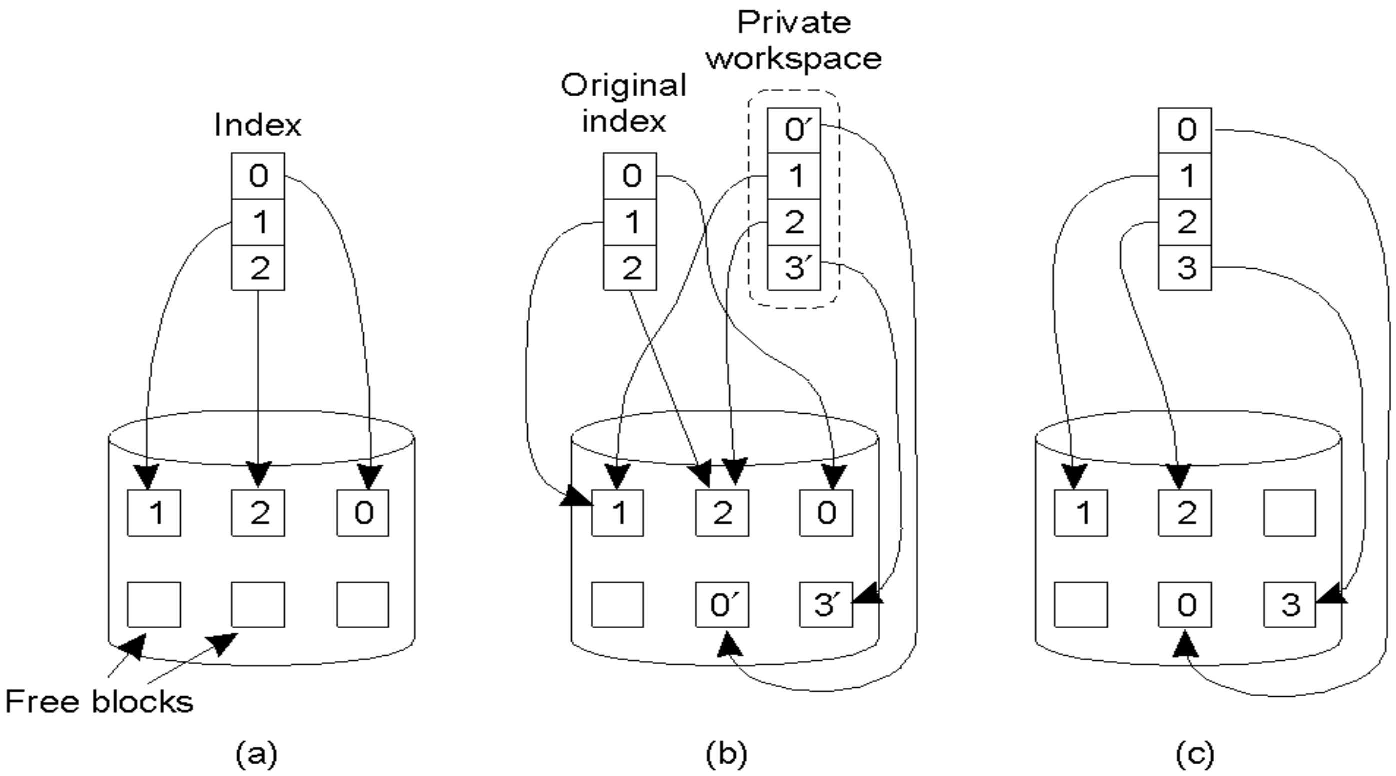




a) A nested transaction

b) A distributed transaction

Private Workspace



a) The file index and disk blocks for a three-block file

b) Situation after a transaction has modified block 0 and appended block 3

c) After committing

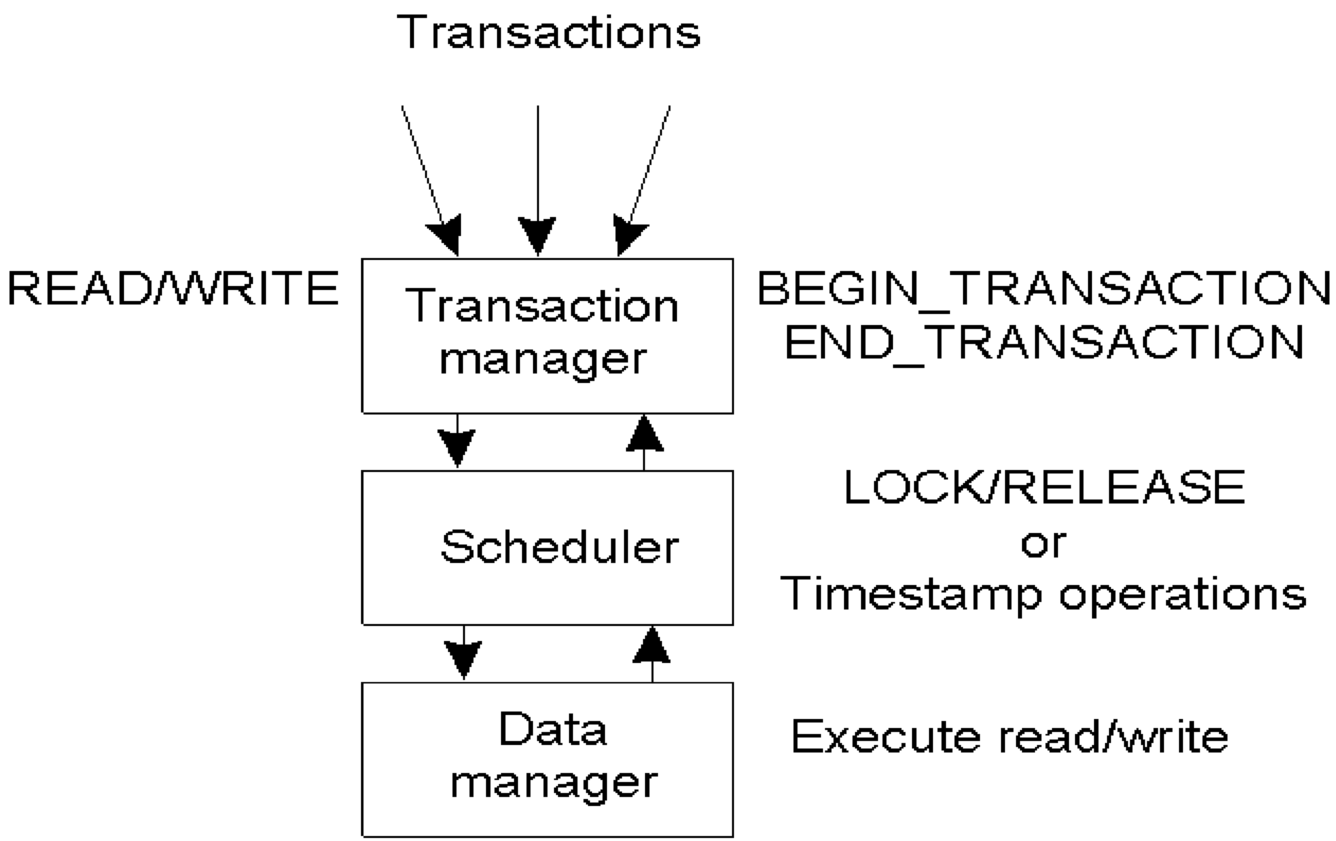
Writeahead Log

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

a) A transaction

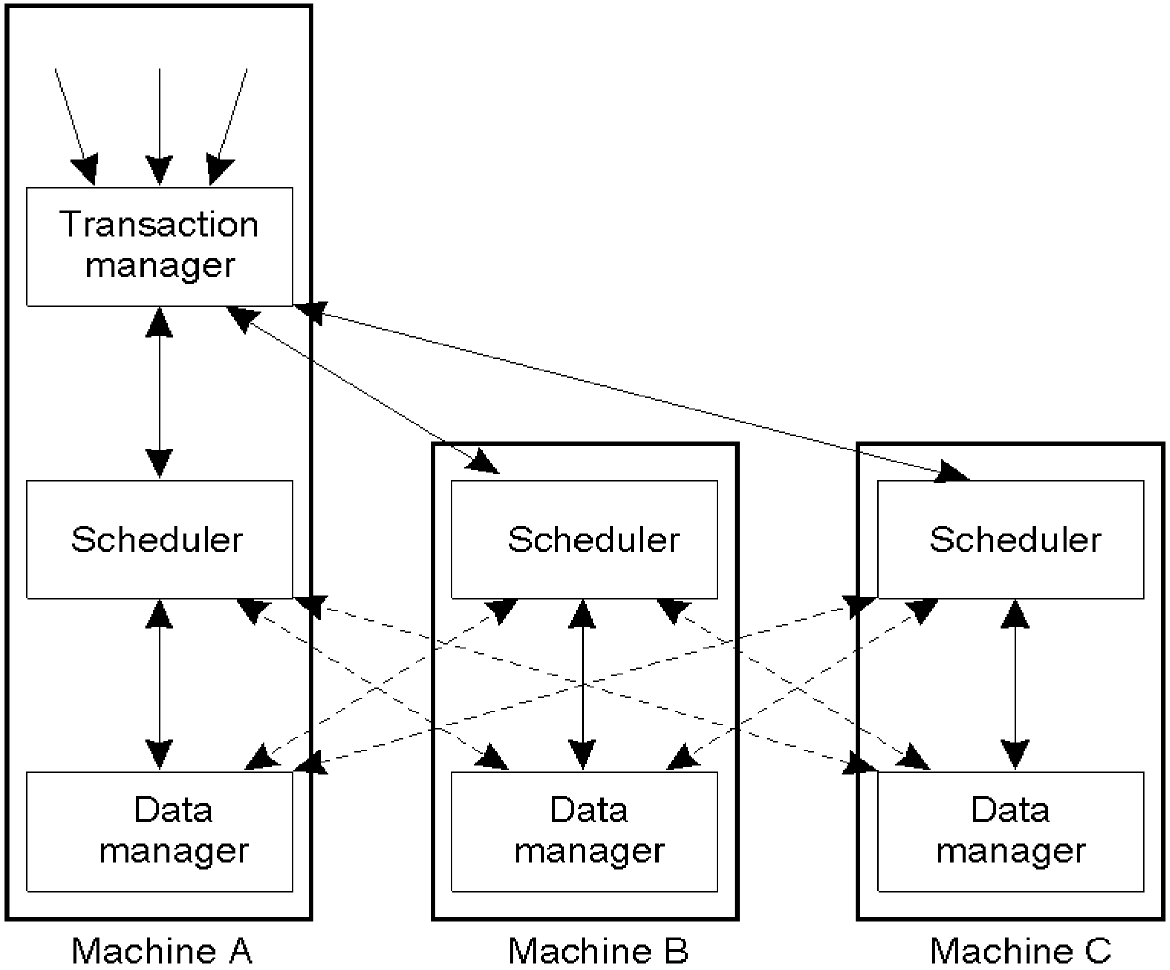
b) – d) The log before each statement is executed

Concurrency Control (1)



- General organization of managers for handling transactions.

Concurrency Control (2)



- General organization of managers for handling distributed transactions.

Serializability

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

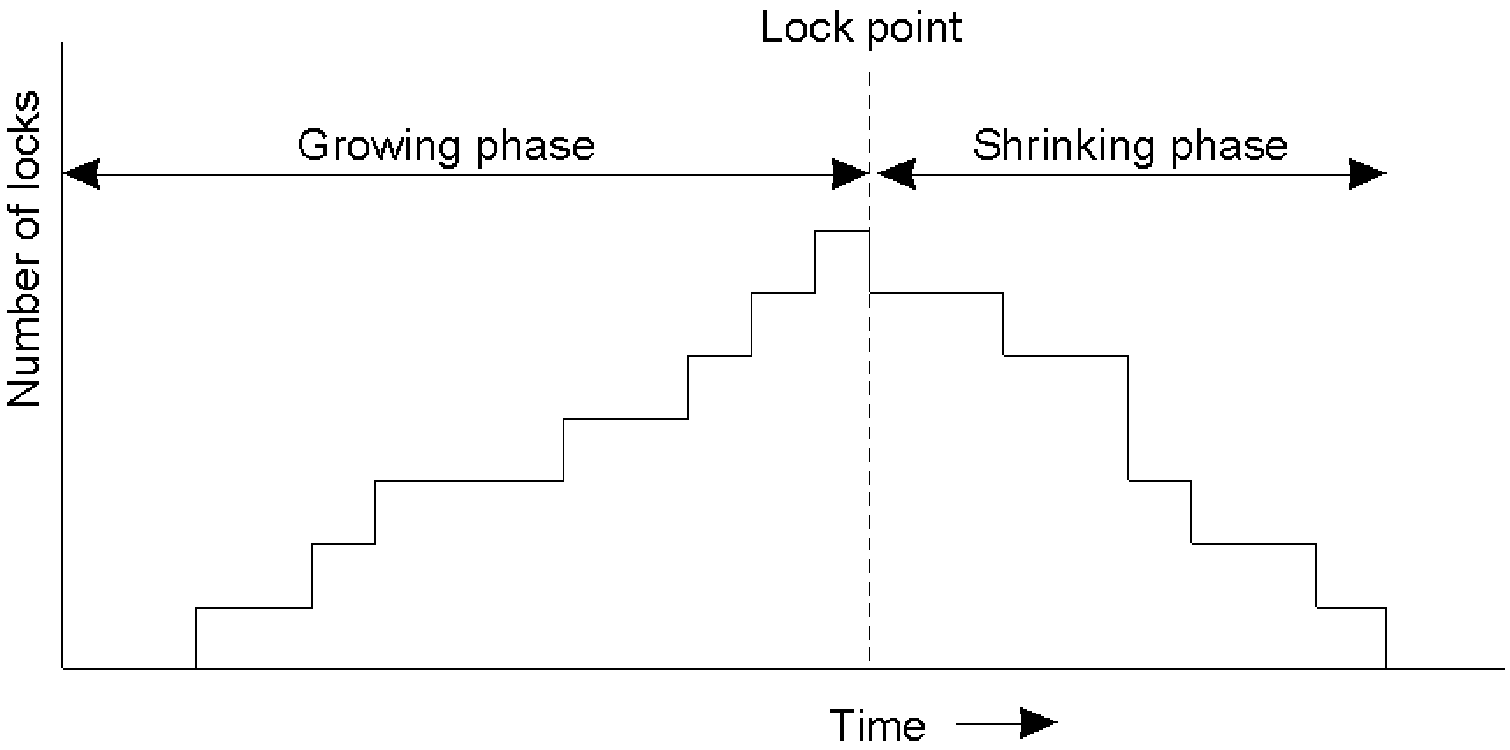
Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	*Illegal*

(d)

a) – c) Three transactions T_1, T_2, and T_3

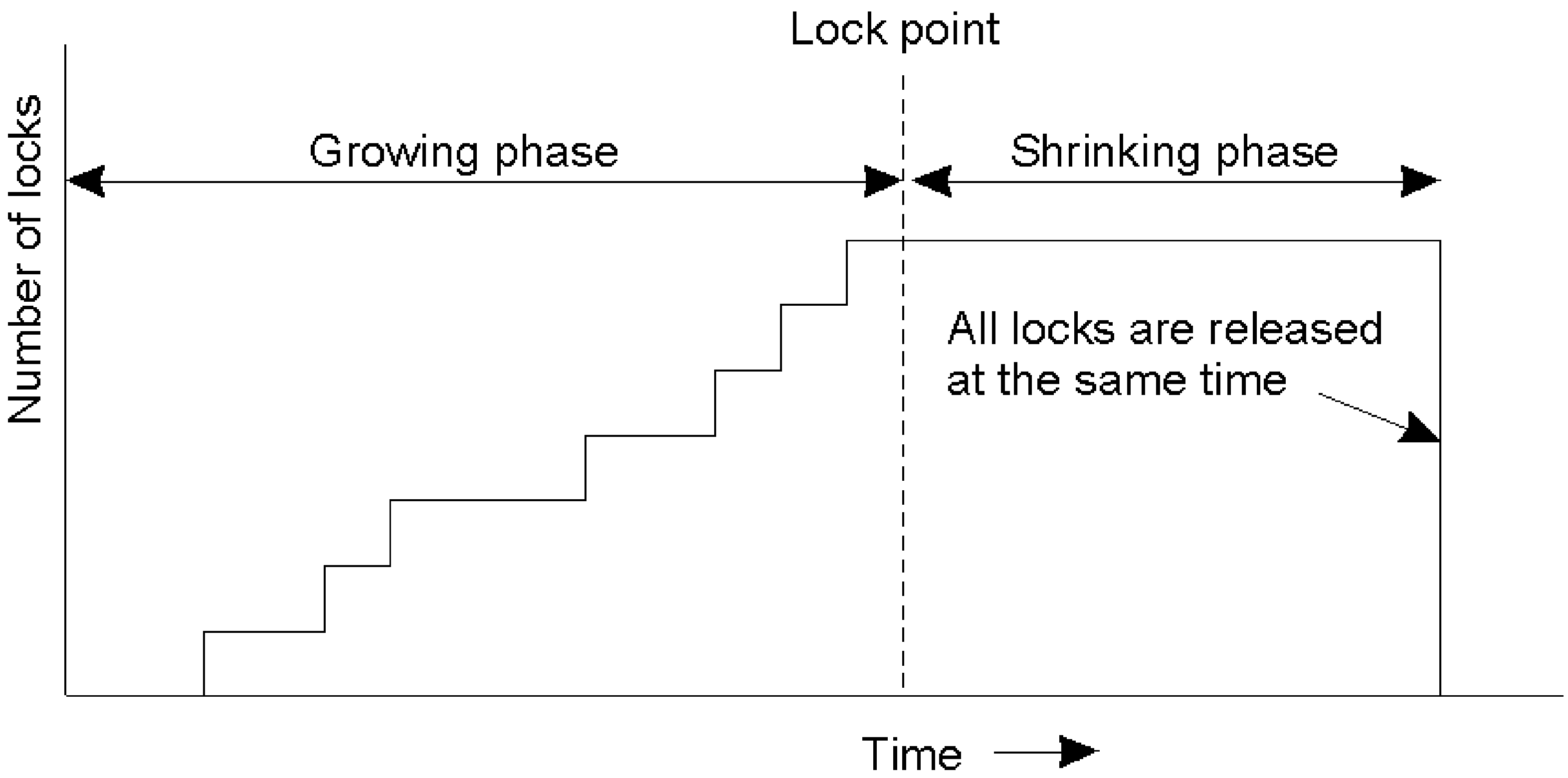
d) Possible schedules

Two-Phase Locking (1)



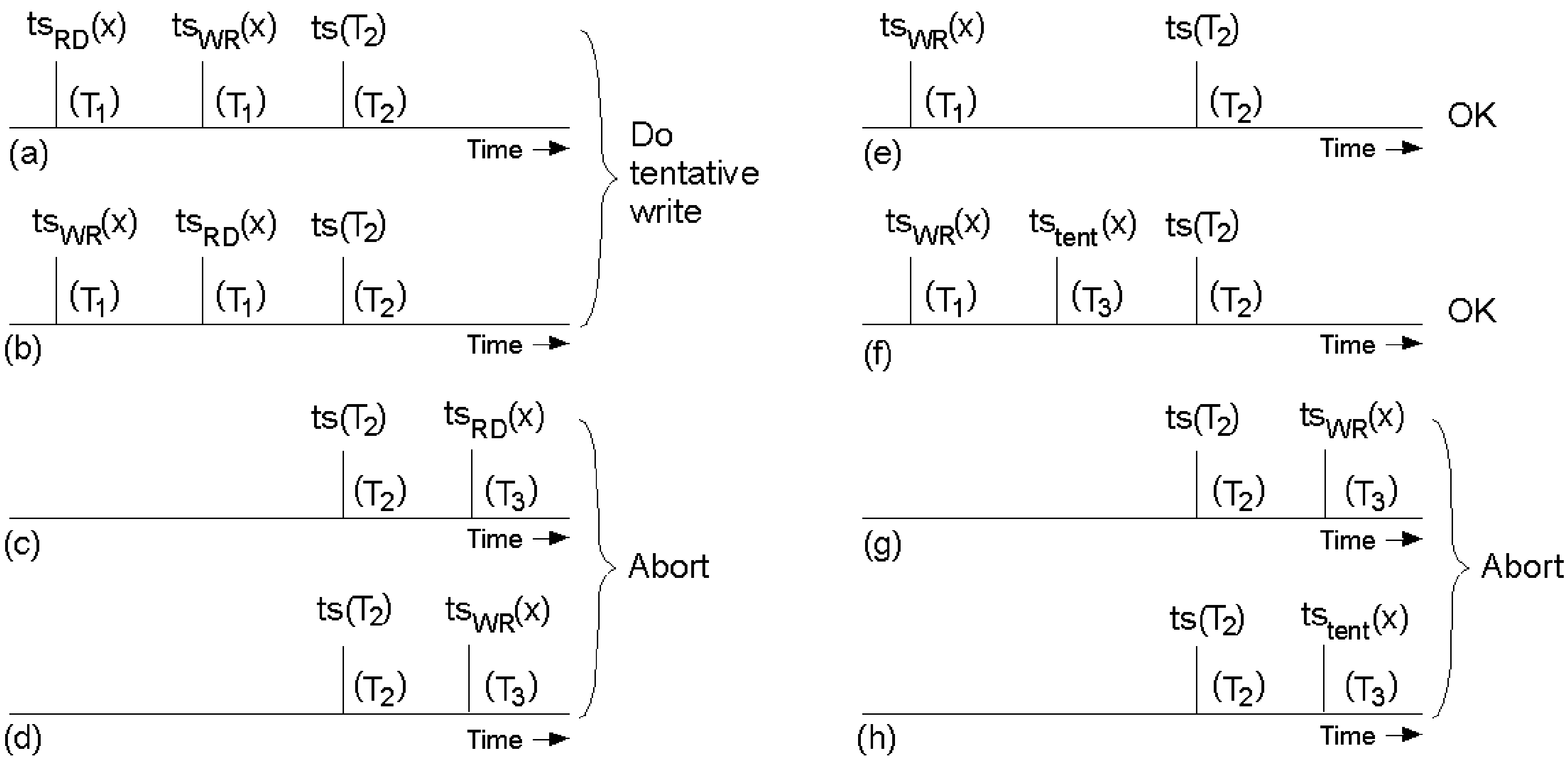
- Two-phase locking.

Two-Phase Locking (2)



- Strict two-phase locking.

Pessimistic Timestamp Ordering



- Concurrency control using timestamps.

Transactions

Notion: A *transaction* is a sequence of operations performing a single “logically composite” function on a shared data base.

Remark:

Transaction derives from traditional business deals:

- You can negotiate changes until you sign on the bottom line
- Then you are stuck
- And your peer is also stuck

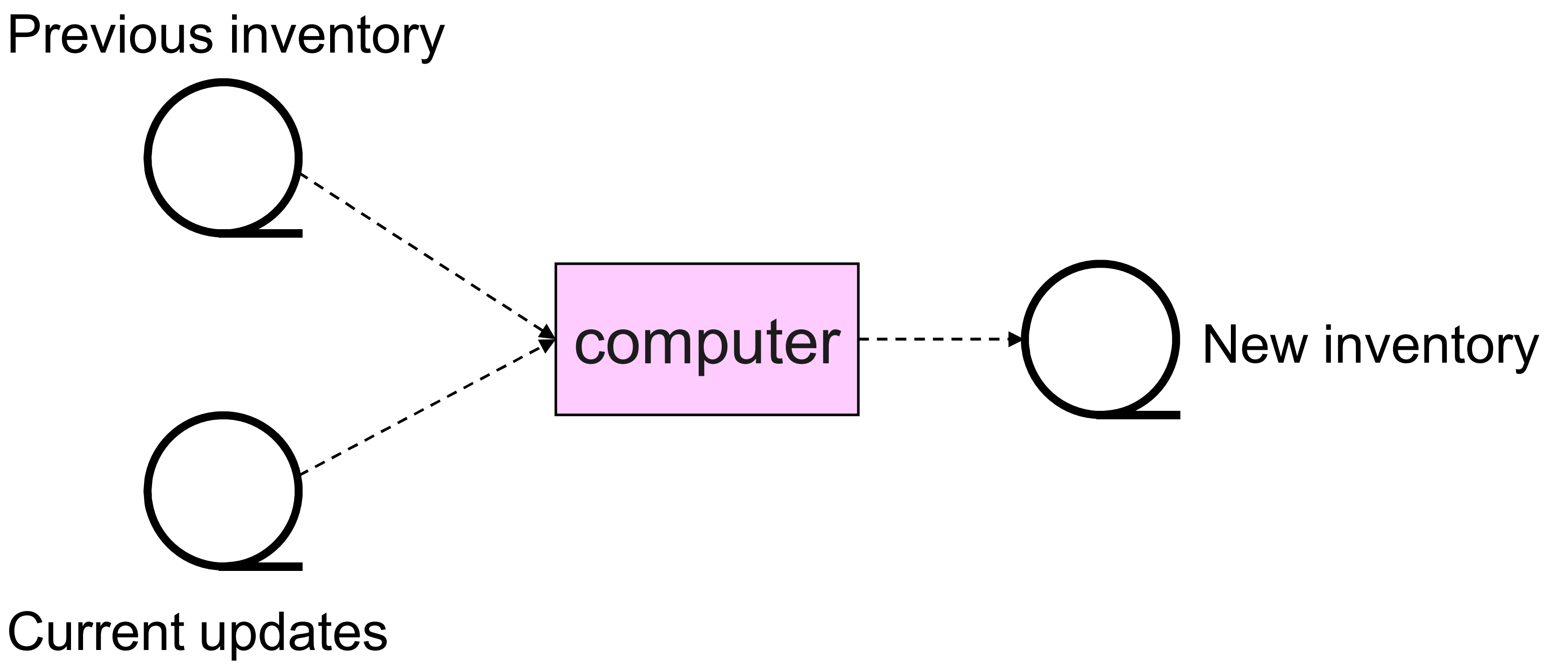
Some Examples

- Reserve a seat for a flight from Frankfurt to JFK in NY
- Transfer money from your account to mine
- Withdraw money from an automatic teller machine
- Buy a book from amazon.com
- Apply a change to a name server

Control of Concurrency

- Critical sections
  - Basic mechanism to enhance data consistency
  - Application programmer has to place locks or semaphores for himself
- Higher Concept
  - Automatically enforcing consistency
  - Support for multiple critical sections
  - Consistency even with failures and crashes

Early „Transaction Model“



Updating a master tape is fault tolerant.
If a failure occurs, just rewind the tapes and restart.

Example (1)

```
{Transfers money from ACC1 to ACC2}
T1: Transfer(ACC1, ACC2, Amount)
      {Part1: Take money from ACC1}
      balance1 := Read(ACC1)
      balance1 := balance1 - Amount
      Write(ACC1, balance1) {debit ACC1}
      {Part2: Put money to ACC2}
      balance2 := Read(ACC2)
      balance2 := balance2 + Amount
      Write(ACC2,balance1)
```

```
{Sum up the balance of both accounts ACC1 and ACC2}
T2: SumUp(ACC1, ACC2, sum)
      sum1 := Read(ACC1)
      sum2 := Read(ACC2)
      sum := sum1 + sum2
```

Problems due to concurrent transactions?

Example (1')

```
{Transfers Amount money from ACC1 to ACC2}
T1: Transfer(ACC1, ACC2, Amount)
      {Part1: Take money from ACC1}
      balance1 := Read(ACC1)
      balance1 := balance1 - Amount
      Write(ACC1, balance1) {debit ACC1}
        {Sumup the balance of both ACC1 and ACC2}
        T2: SumUp(Acc1, ACC2, sum)
              sum1 := Read(ACC1)
              sum2 := Read(ACC2)
              sum := sum1 + sum2
 {Part2 of T1: Put money to ACC2}
       balance2 := Read(ACC2)
       balance2 := balance2 + Amount
       Write(ACC2,balance1)
```

Example (2) “Lost Update Problem”

```
{Transfers Amount money from ACC1 to ACC2}
T1: Transfer(ACC1,ACC2,Amount)
       {Part1: Take money from ACC1}
       balance1 := Read(ACC1)
       balance1 := balance1 - Amount
       Write(ACC1, balance1)
       {Part2: Put money to ACC2}
       balance2 := Read(ACC2) {old value of ACC2}
         {Transfers Amount money from ACC3 to ACC2}
         T2: Transfer(ACC3,ACC2,Amount)
                {Part1: Take money from ACC3}
                balance3 := Read(ACC3)
                balance3 := balance3 - Amount
                Write(ACC1,balance1)
                {Part2: Put money to ACC2}
                balance2 := Read(ACC1)
                balance2 := balance2 + Amount
                Write(ACC2,balance1)
balance2 := balance2 + Amount {ommit the red transaction}
Write(ACC2, balance2)
```

Example (3) (System Failure)

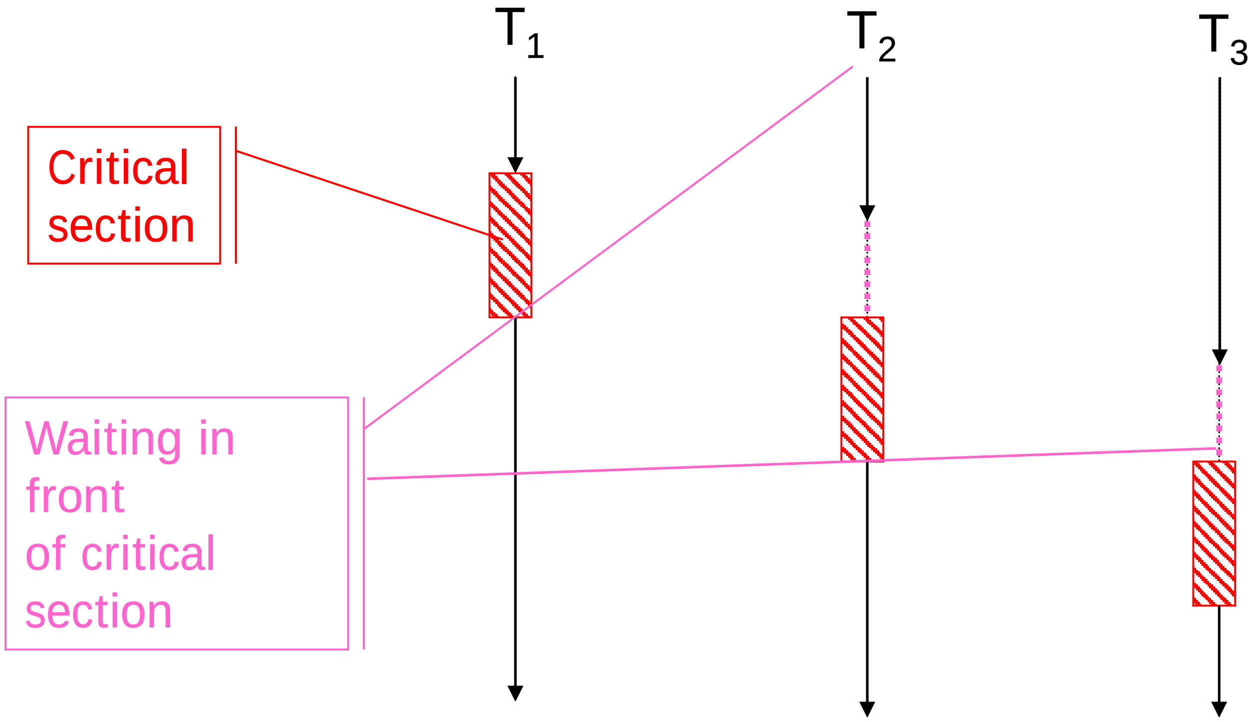
```
{Transfers Amount money from ACC1 to ACC2}
T1: transfer(ACC1,ACC2,Amount)
      {Part1: Take money from ACC1}
      balance1 := READ(ACC1)
      balance1 := balance1 - Amount
      WRITE(ACC1, balance1)
```

Systemcrash

```
      balance2 := READ(ACC1)
      balance2 := balance2 + Amount
      WRITE(ACC2, balance2)
```

Results in an inconsistent state of data-base.

Motivation



Remark: A hard programmer's job in complex systems with o > 1 shared objects and n >> 1 concurrent threads.

Motivation

Example: Book a flight to a conference at San Diego. However, you have to stop for two days in NY where you give a talk.

As a client of a travel office you want to know -after some "minutes"- if you'll get an seat to JFK, a car from JFK to the workshop hall, a nearby hotel, a flight to San Diego etc.

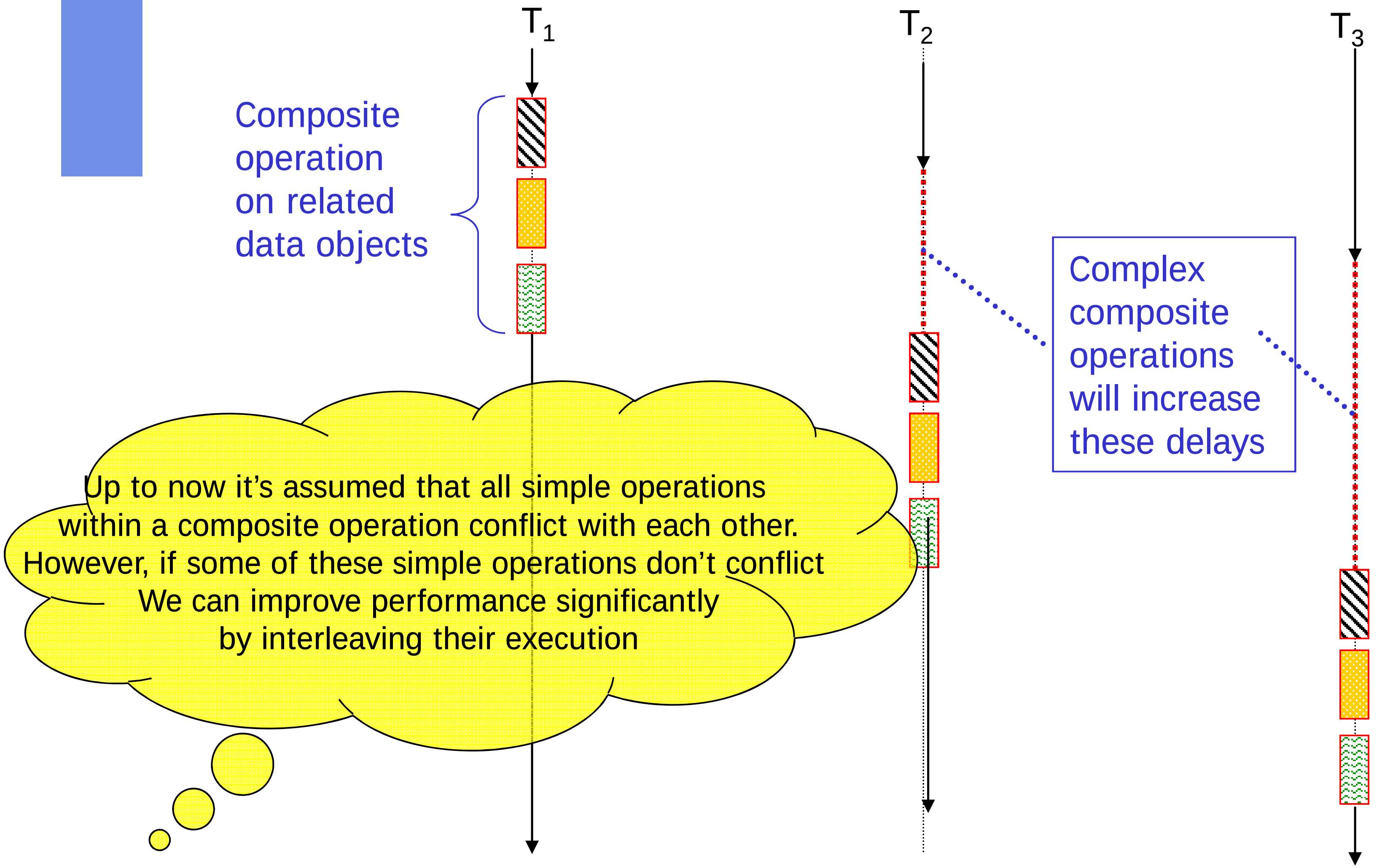
And if they give you an ***OK***, you want to be sure that no other client -at some other place round the globe- has booked the same seat in one of your planes or even worse has booked the same room in the hotel!

$\Rightarrow$ Need for a new higher level concept with the "*all or nothing property*":

Either your request can be *fulfilled* or your fruitless attempt *does not affect* the underlying database at all.

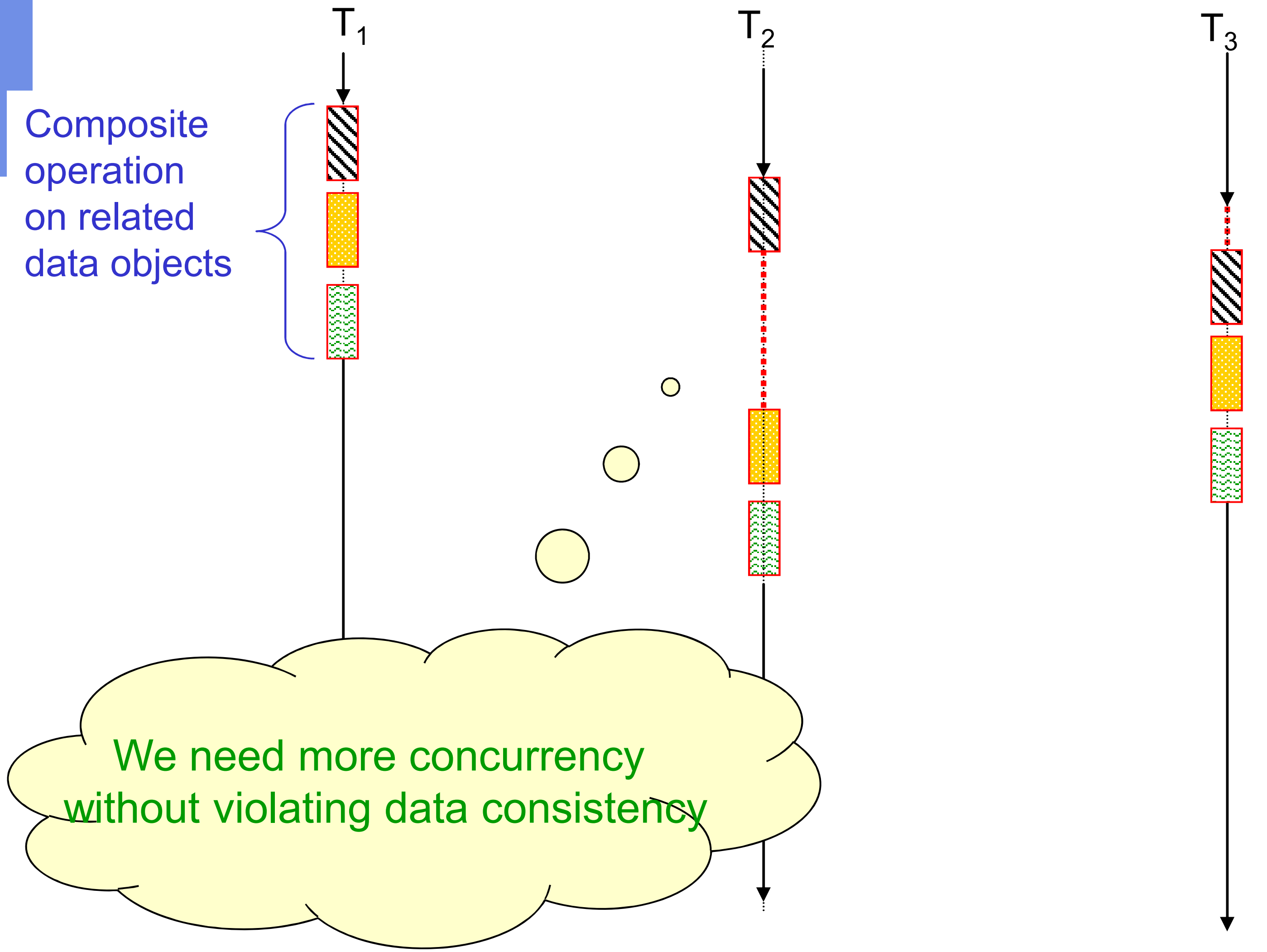
Remark: Let's first have a look at the controlling problem!

Approach 1: Strictly Serial Scheduling



Remark: Works, but with poor performance!

Approach 2: Interleaved Scheduling



Motivation

Requirements for a higher concept:

- It should automatically enforce data consistency
- It should support different critical sections
- It should preserve data consistency even if system errors occur

Transactions

Remark:

The need for transactions is typical for data bases like:

- ticket reservation system
- banking, taxation and assurance systems etc.

normally implemented on hosts within a distributed system.

Assumption (for ease of understanding):

- The data base is located completely in main memory!
- Thus we focus on concurrency aspects and postpone all problems concerning volatile and non volatile memory.

Operations of a Transaction

- An operation within a transaction is any action that *reads* from or *writes* to the shared data base
- A transaction begins with an explicit operation:
  - **Start** (mostly implicitly)
- A transaction is terminated with
  - either a **COMMIT** (if successful)
  - or an **ABORT** (otherwise, with *implicit rollback*)

Terminating Transactions

- COMMIT requests that the effects of the transaction be “signed off on”
- ABORT demands that the effects of the transaction be eliminated, thus an aborted transaction is like a transaction that has never occurred
- COMMIT is *mandatory*, ABORT is *optional*.

Example for Commit and Abort

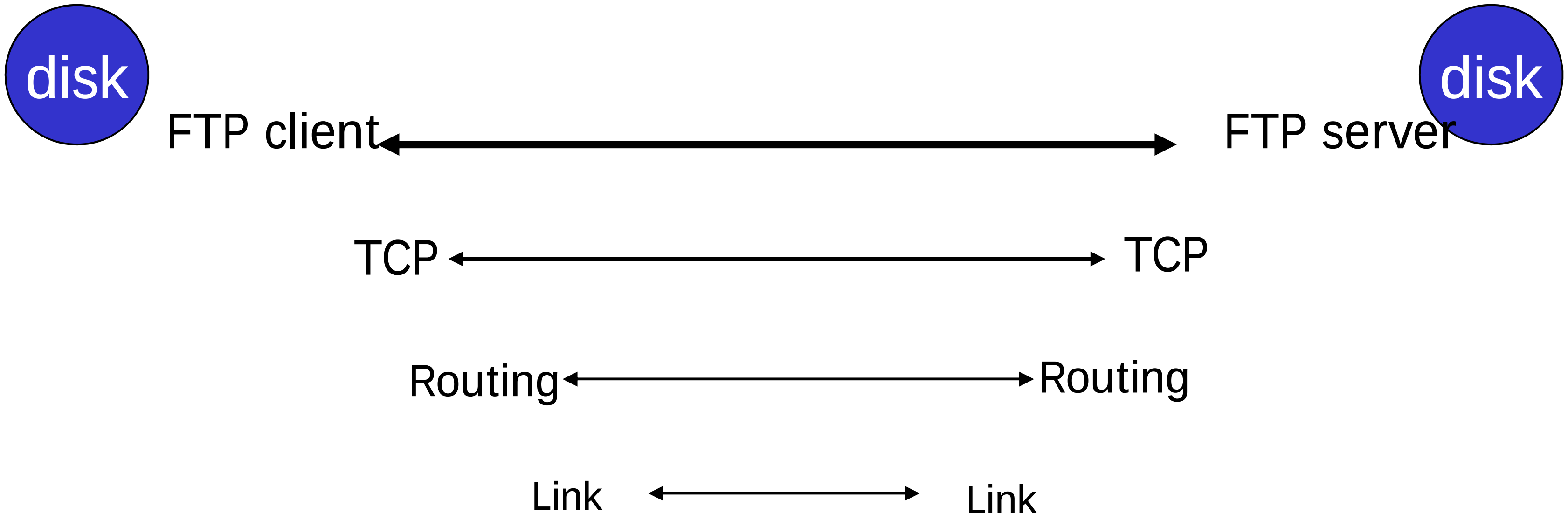
```
{transfer(ACC1, ACC2, Amount)}
begin transaction T account_transfer;
if check_balance(ACC1, Amount)= true
{ACC1 > Amount?}
then begin  debit(ACC1,Amount);
            credit(ACC2,Amount);
            Commit;
     end
else begin  print(“not enough money on ACC1”);
            Abort;
     end
fi
end transaction T;
```

Importance of Transactions

- Basis of all computer-based funds management (1998)
  - about 50 000 000 000 $ per year
- Because COMMIT/ABORT are visible at the application level they provide a good foundation for customized distributed-processing control.
- Operations on the shared data base are “independent” of the implementation of this shared data base.

Transaction: “End to End” Mechanism

Reliability of some logical function can only be expressed at the level of that logical function



Where do the reliability checks go?

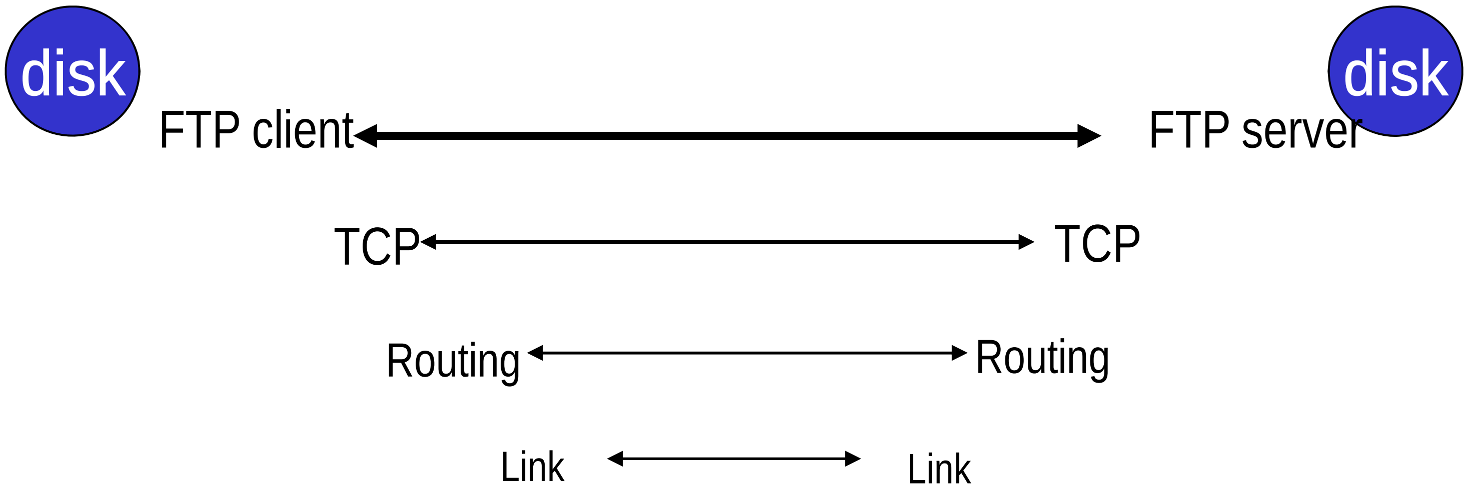
Meaning of an “End to End” Mechanism

- Intermediate checks can not improve correctness
- The only reason to check in lower levels is to improve performance
  - common case failures need not be dealt with at higher levels
  - significantly simplifies programming of distributed applications

Simple FTP Example

```
Open File Locally
Begin Transaction
       Create Remote File
       Send Local Contents to Remote
       Close File Remotely
End Transaction
Close File Locally
```

This section is critical.
It either happens completely
or it doesn't.
Entirely and permanently



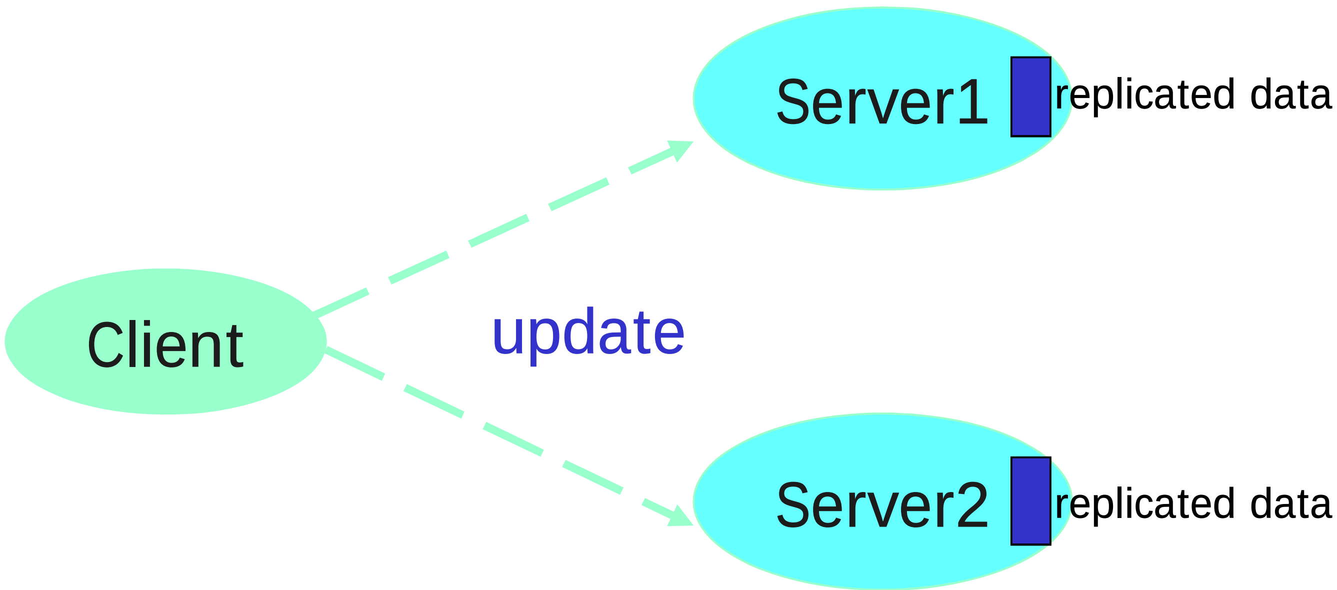
Transactions for distributed applicat.

Assume a <u>replicated name</u> server:

We want to be able to apply changes to the name server and know that changes either took place on both replicas, or at none of them.

```
BeginTransaction
      nameserver.write("espen-pc", xxx.yy.z.007)
      nameserver.write("gerd-tp",xxx.yy.z.815)
EndTransaction(committed)
if(committed)
then io.put("transaction complete\n")
else io.put("update transaction failed\")
```

Implementation



With transactions the above updates will take place on both servers,
or they will take place on none of them,
otherwise the idea of a replicated server must fail.

Characteristics of Transactions

- Clients can crash before/during/after a transaction
- Servers may crash before/during/after a transaction
- Messages can get lost
- Client requests can come in any order to any server
- Servers can reboot
- Performance is an important goal, otherwise useless

Requirements: ACID-Principle

Atomicity = All or nothing concept

Consistency = state(t_i) $\rightarrow$ state(t_{i+1})

Isolation = Intermediate results of a transactions are hidden to other transactions

Durability = Result of a committed transaction is persistent

Properties of Transactions

- Consistency:
  - transactions retain internal consistency of the data base
  - many kinds of integrity
    - unique keys
    - referential integrity
    - fixed relationships hold
  - responsibility belongs to the programmer, not to the data base
  - if each transaction is consistent, then all sequences of transactions are as well

Properties of Transactions

- Isolation:
  - result of running multiple concurrent transactions is the same as running them in complete isolation
  - often called “serializability”

Properties of Transactions

- Durability:
  - once committed, “un-undoable”
  - can be “compensated” for though

How to achieve ACID-Principle?

- ATOMICITY → Undo
- CONSISTENCY → Be a good programmer
- ISOLATION → Serializability and *locking*
- Durability → Stable storage, *logging*

Consistency Predicate

```
BeginTransaction
  old_balance :=
      bank.read(“gerd”);
  new_balance :=
  old_balance + $1M;
  bank.write(“espen”,
             new_balance);
EndTransaction(&commit)
```

```
BeginTransaction
  if bank.read(“espen”) -
  bank.read(“gerd”) != $1M
  then CallTheSEC();
EndTransaction(&commit)
```

The above consistency predicate is quite simple:

“Espen always has 1 M$ more than Gerd”.

As long as one transaction’s operations happens entirely before(after) the other, consistency is still valid.

However, interleaving of these operations might violate consistency!

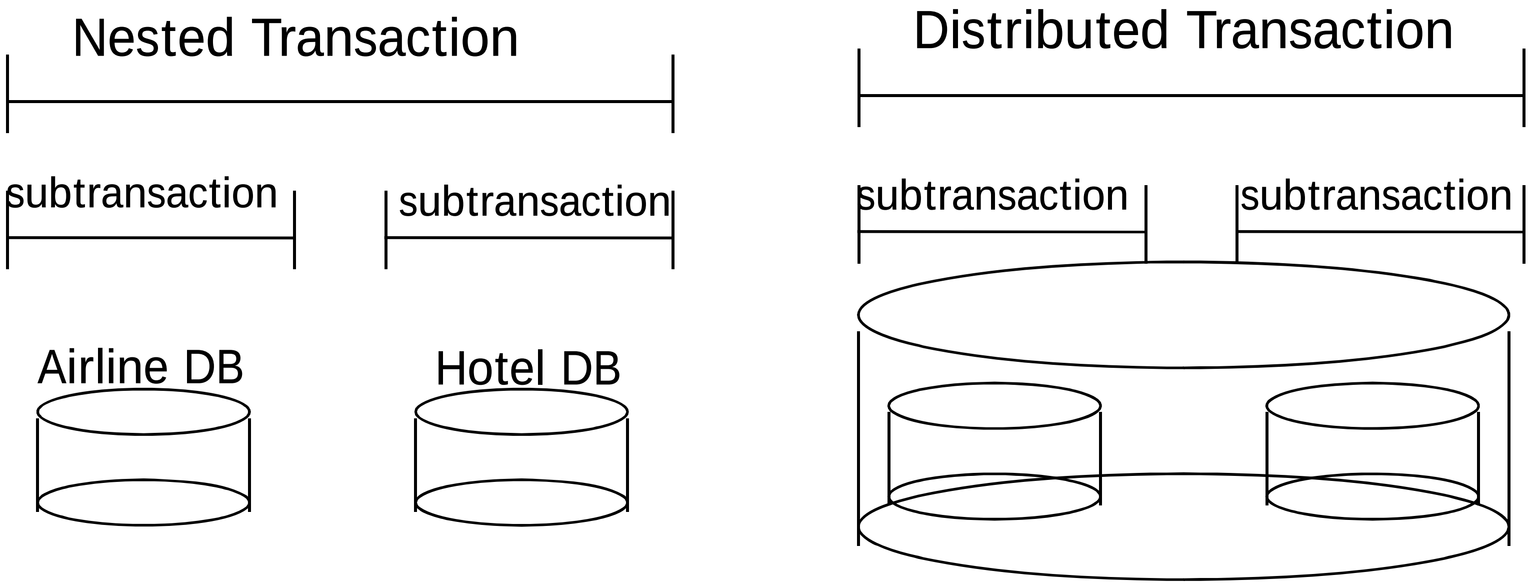
SEC = Security and Exchange Commission

[1]Espens’s favorite bank

Transaction Types

- Flat Transactions
  - No commit of partly successful transaction operations is possible
- Nested Transactions
  - Sub- and sub-sub-transactions
  - ACID-Principle valid only for top-level transaction
- Distributed Transactions

Nested/Distributed Transaction



2 different (independent) data bases

2 physically separated parts of the same data base

Serializability

- Remember sequential consistency:
  - Execution order is consistent with some sequential interleaving of loads and stores in program order
- Serializability is the same, but at transaction level
- Transactions appear to execute in serial (one at a time) order
  - preserves any consistency property of the data base
  - system does not need to know consistency properties
    - untyped transactions
  - can introduce new transactions without worrying about interactions

Serializability: A Powerful Abstraction

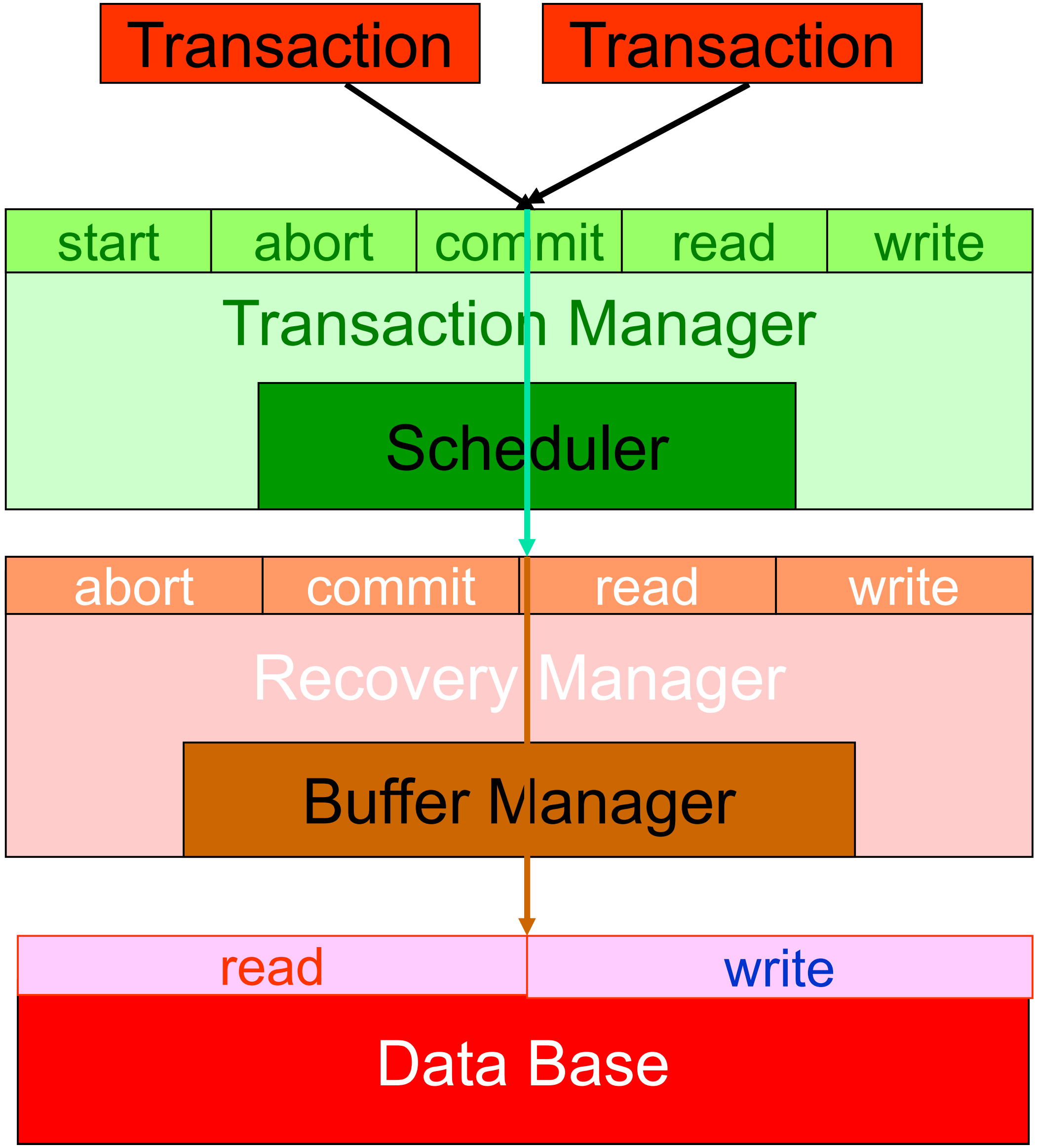
- Important abstraction for constructing parallel programs
- Designing a transaction can be done in “splendid isolation”
  - an important software-engineering argument
- You can compose transactions
  - a modularity argument
- You can verify using pre- and post-conditions
  - thanks to isolation and consistency

Main Modules of Transaction Systems

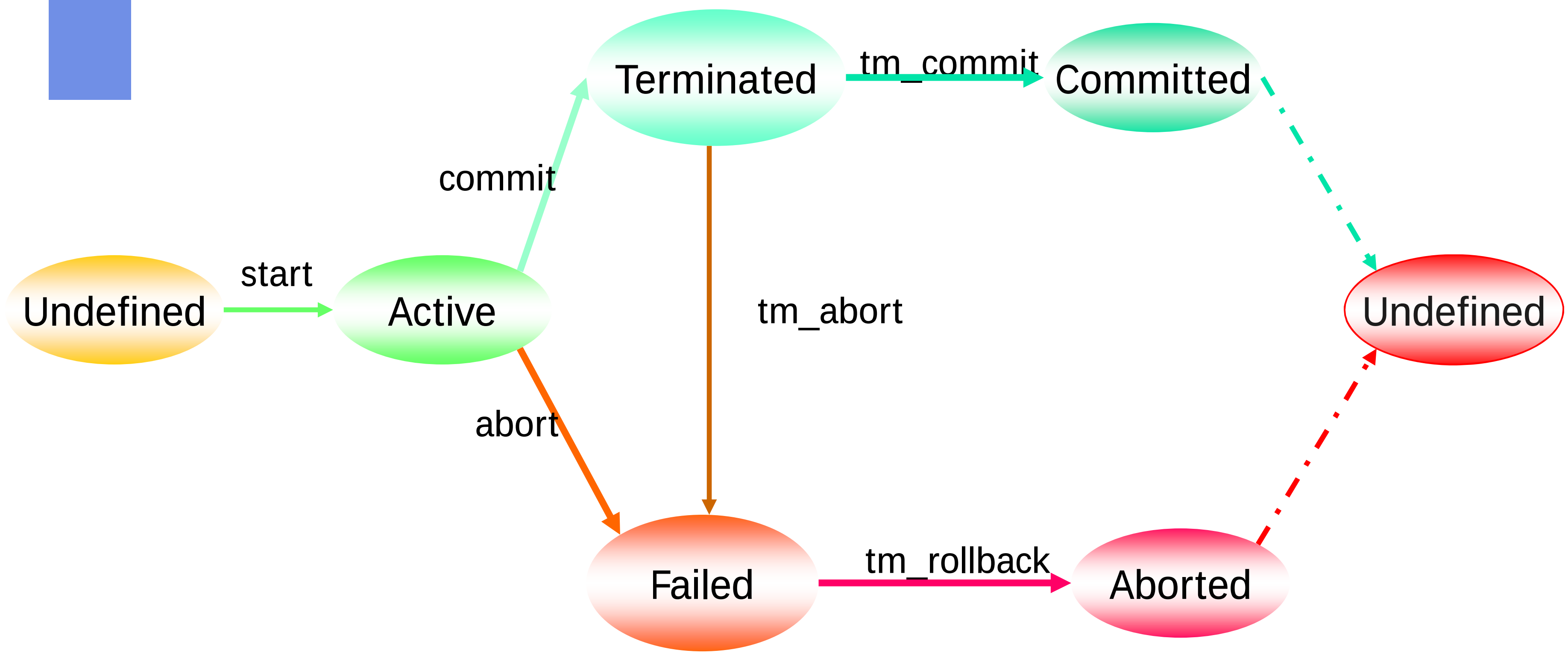
We have to offer mechanisms supporting

- Concurrency Control of Transactions
- Recovery of faulty Transactions

<u>Remark:</u> Both concurrency and recovery may be implemented as isolated modules


Transaction
Transaction
start
abort
commit
read
write
Transaction Manager
Scheduler
abort
commit
read
write
Recovery Manager
Buffer Manager
read
write
Data Base

Transaction States



Transaction

<u>Definition:</u> Transaction T_i is a sequence of reads r_i and/or writes w_i, opened via start s_i and closed either via commit c_i or abort a_i

<u>Remarks:</u> A single transaction T is per se serial, i.e. all its operations are totally ordered.

Result of a single transaction is *unambiguous.*

However, concurrent transactions may lead to *conflicts*, as we have seen in the previous examples.

Conflicting Operations

Definition: $p(x) \in T_i$ and $q(y) \in T_j$ are ***conflicting*** operations $p \leftrightarrow q$, if

(1) they access the same data item, i.e. $\boldsymbol{x = y}$ and

(2) they belong to different transactions, i.e. $\boldsymbol{i \neq j}$ and

(3) at least one of them is a write,
i.e. $\boldsymbol{p} = \mathbf{w}$ or $\boldsymbol{q} = \mathbf{w}$

Objective: Find a schedule, i.e. find a way of controlling ***c*** concurrent transactions T_i, such that the result is equivalent to one of the ***c!*** serial schedules.

Commutative Operations

Definition: Two operations p() and q() are *commutative*, if in any state, executing them in any order $p \angle_S q$ or $q \angle_S p$, you

- get the same result
- leave the system in the same state

Examples of commutative operations:

- operations on distinct data items
- reads from the same data item

Conclusion:

Commutative operations can be executed in any order

Schedule

Definition: A schedule S of concurrent $\{T_1, T_2, \ldots T_n\}$ is a partial ordered set of all transaction operations (i.e. reads $r_{i,k}$ and/or writes $w_{j,k}$) with following properties:

1. Suppose $p, q \in T_i$ and $p \angle_i q$, i.e. p precedes q in T_i, then: $\mathbf{p} \angle_{\mathbf{S}} \mathbf{q}$
   (sequence of operation within a transaction is preserved)

2. $\forall$ $p, q \in S$: $p \leftrightarrow q \Rightarrow$
   either $p \angle_{\mathbf{S}} q$ or $q \angle_{\mathbf{S}} p$,
   (for all conflicting operations we have to find a sequence)

Inconsistent Schedule

Not valid !!

$T_1 \longrightarrow \mathbf{w(x)} \longrightarrow \mathbf{r(x)} \longrightarrow c_1$

$T2 \longrightarrow \mathbf{w(y)} \longrightarrow \mathbf{r(y)} \longrightarrow c_2$

$S': \longrightarrow \mathbf{w(y)} \longrightarrow \mathbf{r(y)} \longrightarrow c_2 \longrightarrow \mathbf{r(x)} \longrightarrow \mathbf{w(x)} \longrightarrow c_1$

S‘ violates condition 1

Execution of a Schedule

T1	T2	T3
read(x)		
write(x)		
		read(x)
		write(x)
	read(x)	
		write(y)
commit		
	write(y)	
	abort	
		commit

T_2: $r_2(x) \rightarrow w_2(y) \rightarrow a_2$

T_3: $r_3(x) \rightarrow w_3(x) \rightarrow w_3(y) \rightarrow c_3$

T_1: $r_1(x) \rightarrow w_1(x) \rightarrow c_1$

Remark: The execution of as schedule determines the schedule but not vice versa!

Recoverability

T_1	T_2
write(x)	
	read(x)
	commit
abort	

time

<u>Remark:</u> The above schedule is not recoverable, because T_2 already has been committed, even though it had read data x which is not valid any longer.

<u>Definition:</u> A transaction T *reads from another transaction* T', if T reads data items that have been written by T' before.

Recoverability

Definition: A schedule S is *recoverable* (S $\in$ RC) if each transaction T in S is committed only if all transactions T', from which T had read before, are either committed or aborted.

In the last case you have to undo transaction T.

Dirty Read

T_1	T_2	T_3
write(x)		
	read(x)	
	write(y)	
		read(y)
abort		
	commit	
		commit

time ↓

Remark: This schedule is recoverable, however transactions T_2 and T_3 have to be rolled back.

<u>Definition</u>: A schedule S avoids cascading aborts ($S \in ACA$), if no transaction ever reads not-yet-committed data.

Dirty Write (1)

Initially x = 1

time ↓	T_1	T_2
	write(x,2)	
		write(x,3)
	commit	
		abort

Remark: You have to ensure that the committed value x=2 will be reinstalled, you need a *before-image* (rolling back transaction T_2).

Dirty Write (2)

Initially x = 1

time ↓

T_1	T_2
write(x,2)	
	write(x,3)
abort	
	commit

Remark: You do not have to reinstall x=1 in this case, because before T_1 is aborted the data object x already got a new consistent value x=3.

Dirty Write (3)

Initially x = 1

time ↓

T_1	T_2
write(x,2)	
	write(x,2)
abort	
	abort

Remark: In this case you have to reinstall x=1, because both transactions abort and both have to be rolled back.

Result: Overwriting uncommitted data may lead to recovery problems !

Definition: Schedule S is *strict* ($S \in ST$) if no transaction reads and overwrites non-committed data

First Idea: Serial Schedule

Run all transactions completely serially =>

- transactions ordered according to some serial order
- the simplest order is the order in which the transactions arrive to the system
- drawback: “no performance” at all

Potential implementation:
Use one central lock for all transactions,
each transaction is a critical section.

Serial Schedule

Definition: A schedule S is serial if the following holds for each pair of transactions <T, T’> $\in$ S: either all transactions operations of T precede all transaction operations of T’ or vice versa.

Conclusion: Suppose each transaction T to be scheduled is *correct*, i.e. after its execution the data base is consistent, then each *serial schedule S* is correct.

Properties: Serial Schedule

Problem:

Suppose there are 2 transactions T and T'.

Are the results of both serial schedules identical?

Not at all!

Simple example: 3 accounts: acc1, acc2, acc3

⇒

T: sumup(acc1, acc2, sum) and

T': transfer (acc1, acc3, 1 000 €)

How to get a Correct Schedule?

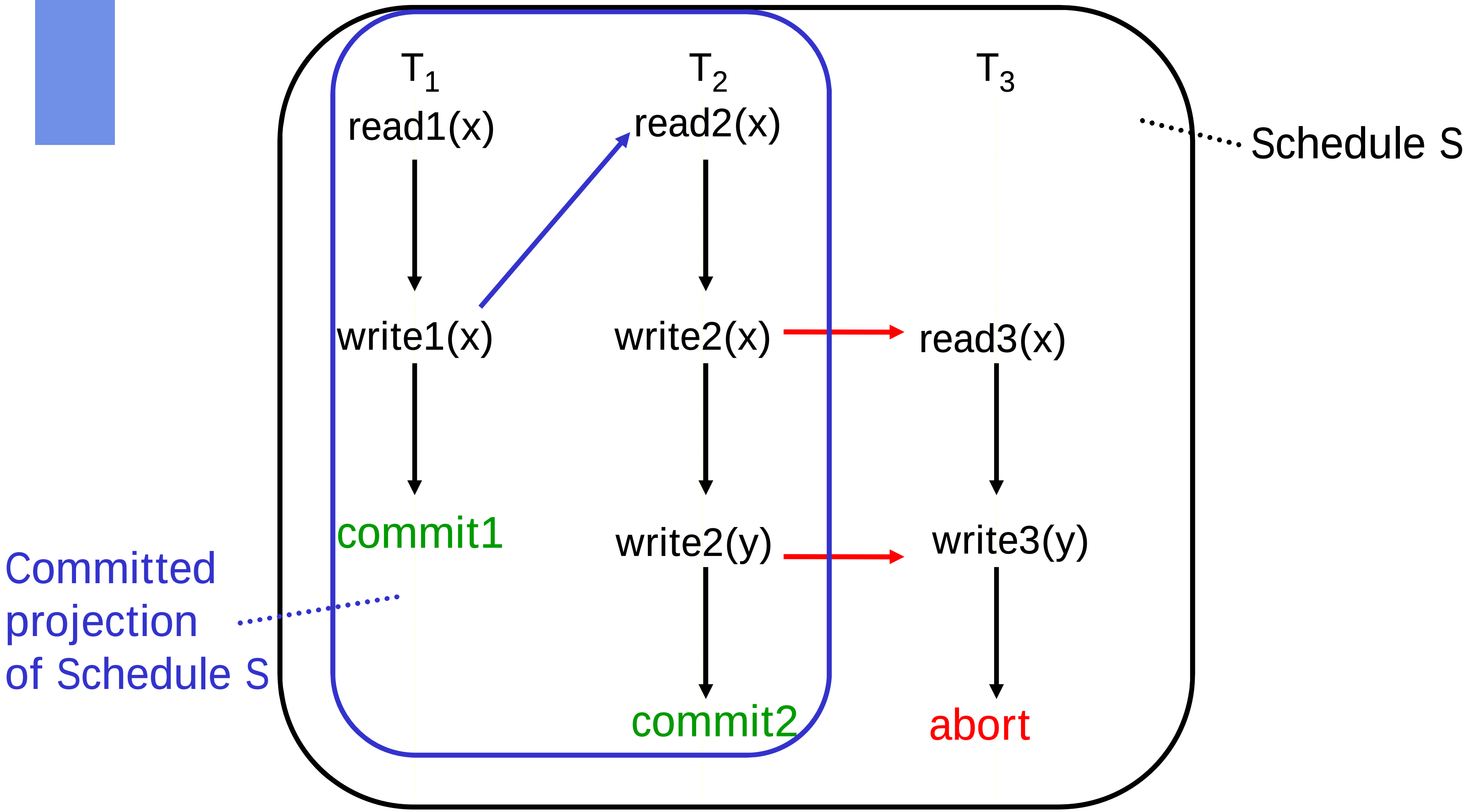
Construct a schedule being equivalent to a serial schedule

Definition: 2 Schedules S and S' are *equivalent*, if their *output results* are identical and if the data base contains the same data.

Definition: *A schedule S is serializable* (S $\in$ SR), *if its committed projection C(S) is equivalent to a serial schedule..*

C(S) $\subseteq$ S: skip all transaction operations belonging to not committed transactions in S

Commited Projection



Assumptions: Serializable Transactions

- Date base consistency requirements are application specific
  - e.g. Espens's additional 1 M $
- Semantics of per-item operations are understood by system
  - Read bank account
  - Write bank account
- Each operation onto a data item of the data base is atomic
  - Reading a bank account is a single operation that either
  - occurs or it does not. It can not partially occur.

How to find an easy way
to construct serializable schedules?

Towards Serializable Schedules

In order to decide during the execution of transactions, whether they contribute to a serializable schedule, we need a definition for equivalent schedules:

Definition: Two schedules S and S' are said to be *conflict-equivalent*, if they contain the same transaction operations and if they resolve their conflicts in the same way, i.e. the following holds:
$\forall$ conflicting transaction operations, i.e. $p \leftrightarrow q$:

$$\mathbf{p} \angle_{S} \mathbf{q} \quad \Leftrightarrow \quad \mathbf{p} \angle_{S'} \mathbf{q}$$

$\Rightarrow$ 2 conflict-equivalent schedules produce the same result.

Remark: 2 schedules S and S' may produce same results, even if they are not conflict-equivalent.

Towards Serializable Schedules

<u>Definition:</u> A schedule S is said to be *conflict-serializable* (i.e. S $\in$CSR), if its committed projection C(S) is conflict-equivalent to a serial schedule.

$\Rightarrow$ Serializability is independent of strictness or recoverability, i.e. in general a serializable schedule can be incorrect!

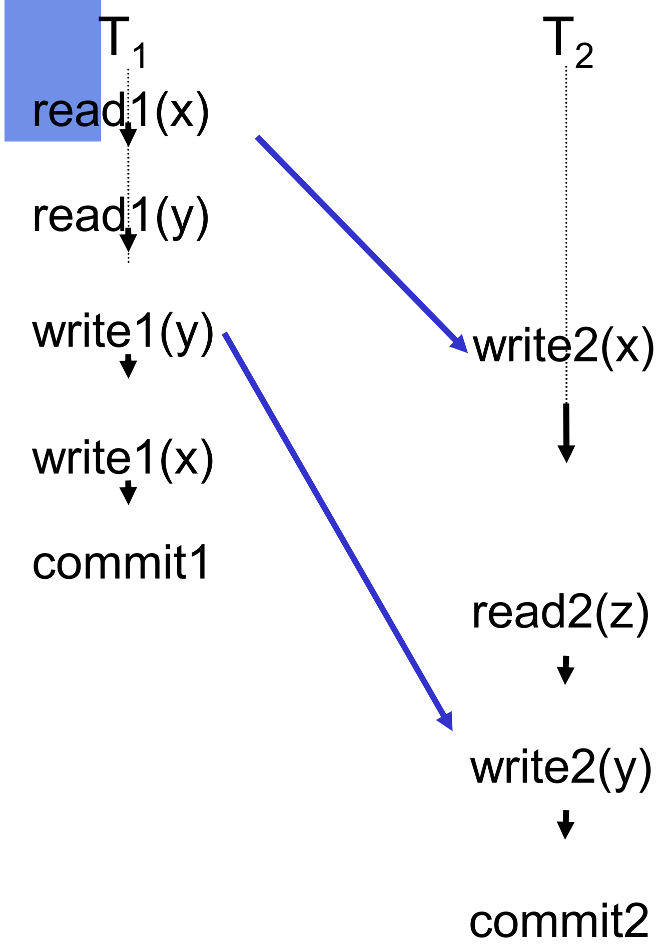
<u>Definition:</u> A schedule S is *correct*, if it is serializable <u>and</u> recoverable

Examples of Schedules

T_1	T_2
read1(x)	
↓	
read1(y)	
↓	
write1(y)	write2(x)
↓	↓
write1(x)	
↓	
commit1	
	read2(z)
	↓
	write2(y)
	↓
	commit2

S_0 is not yet determined, due to write conflicts on x and y

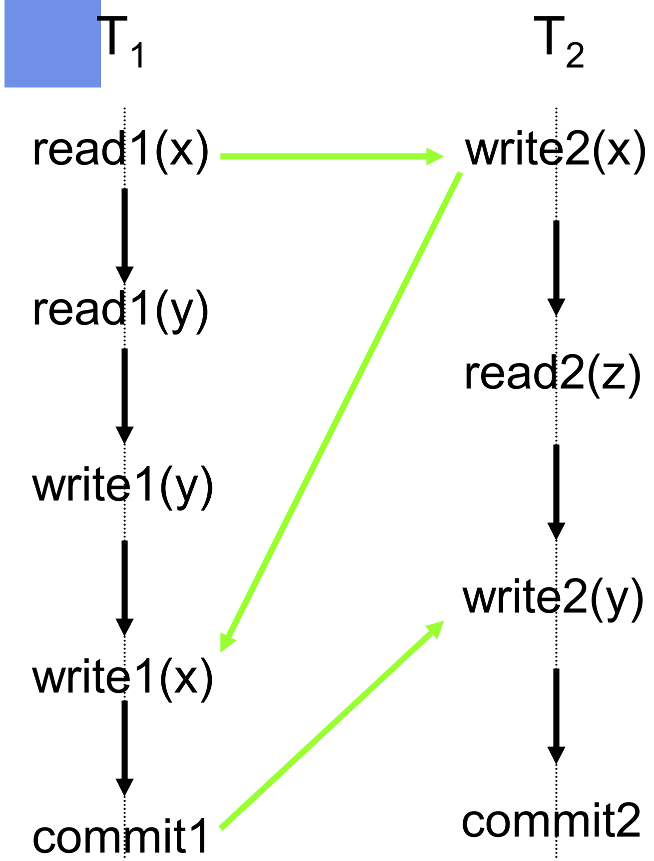
Examples of Schedules



S_0 is not yet determined,
due to write conflicts on x and y

S_1 not yet determined (write conflicts on x)

Examples of Schedules

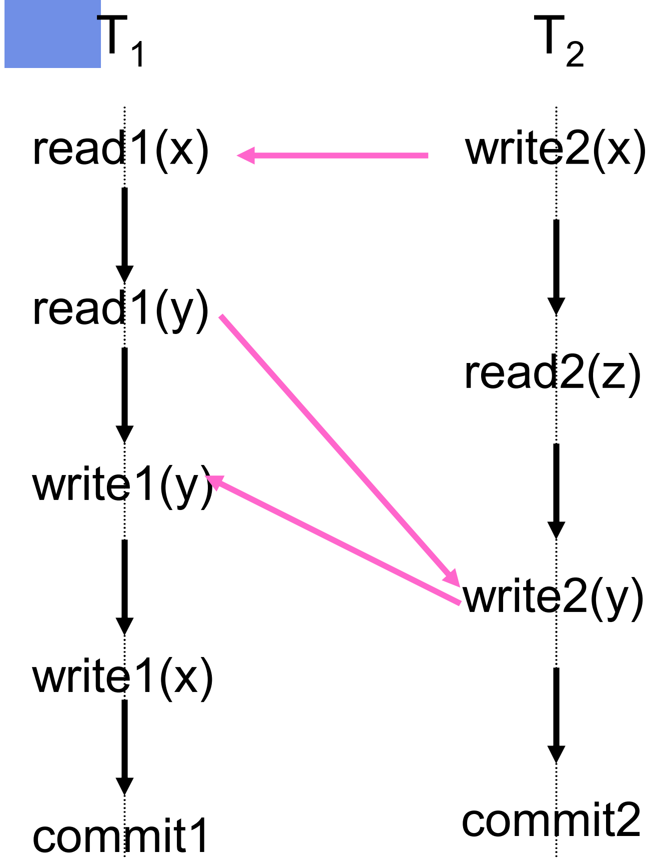


S_0 is not yet determined,
due to write conflicts on x and y

S_1 not yet determined (write conflict on x)

S_2 completely determined, not serializable
(the order of writes on x and y is reverse)

Examples of Schedules



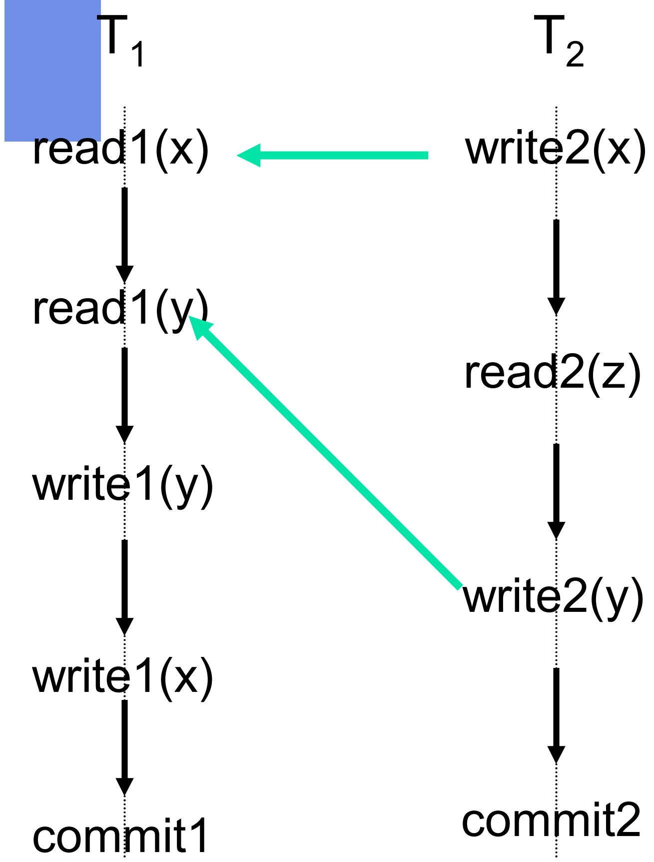
S_0 is not yet determined,
due to write conflicts on x and y

S_1 not yet determined (write conflict on x)

S_2 completely determined, not serializable
(the order of writes on x and y is reverse)

S_3 completely determined, not serializable

Examples of Schedules



S_0 is not yet determined,
due to write conflicts on x and y

S_1 not yet determined (write conflict on x)

S_2 completely determined, not serializable
(the order of writes on x and y is reverse)

S_3 completely determined, not serializable

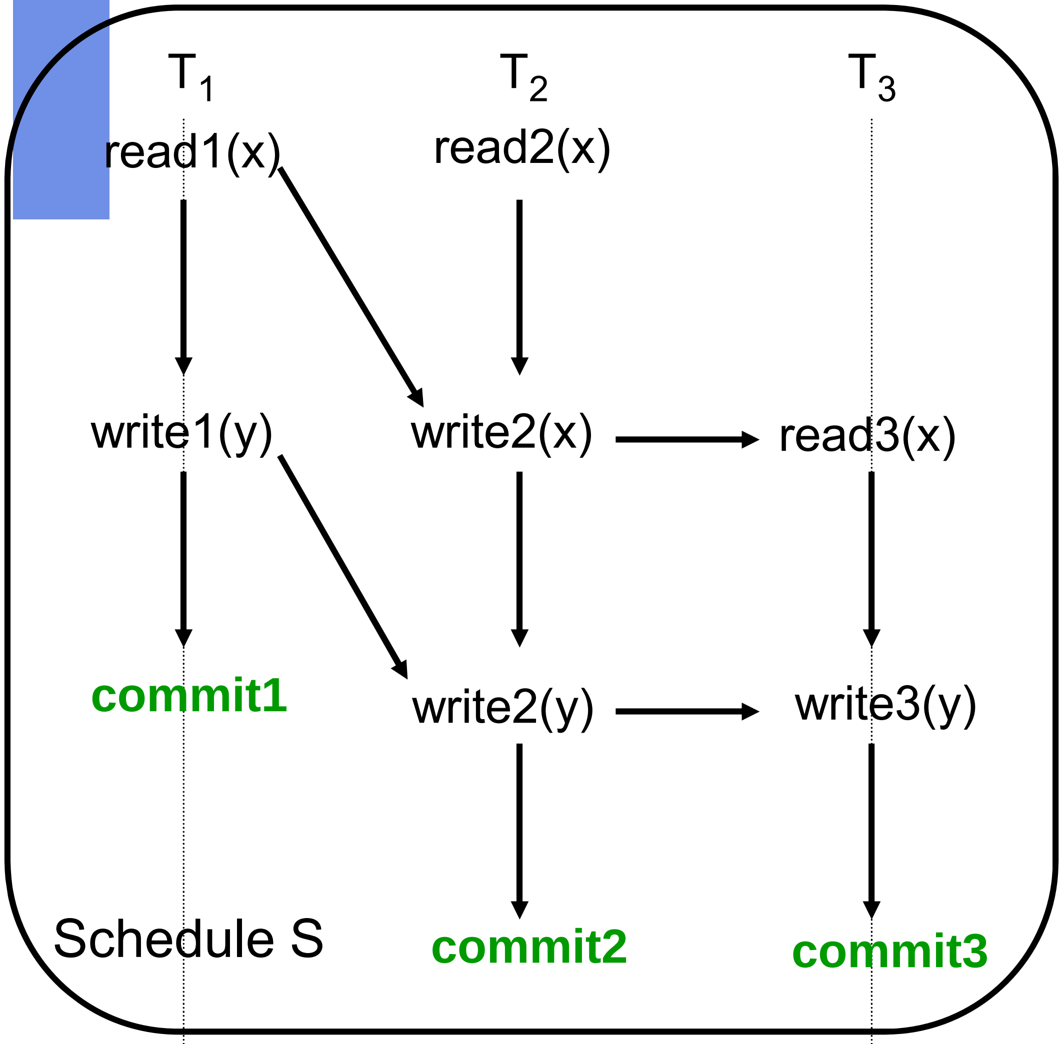
S_4 completely determined <u>and</u> serializable,
equivalent to the serial schedule
S': $T_2 \angle_{S'} T_1$

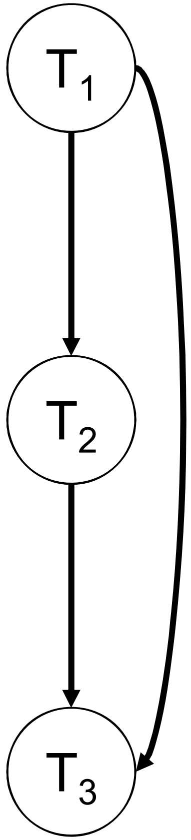
Serialization Graph

Definition: The serialization graph of a schedule S is a digraph (directed graph), whose nodes are all committed transactions.

There is an edge from T_i to T_j iff there is a pair of conflicting transaction operations, i.e. $p_i \in T_i$ and $q_j \in T_j$ and $p_i \angle_S q_i$

Example of a Serialization Graph





Serialization Graph SG(S)

Serializability Theorem

A shedule S is conflict-serializable if its serialization graph SG(S) is ***acyclic***!

Conclusion: Running concurrent transactions we have to guarantee, that executing their transaction operations concurrently does not imply that the corresponding SG(S) becomes acyclic.

Remark: Serializablity is *necessary*, but not sufficient for a correct schedule of concurrent transactions

Achieving Serializability

Problem:

A scheduler of concurrent transactions is responsible for achieving a schedule with some desired properties, i.e. serializability, recoverability etc.

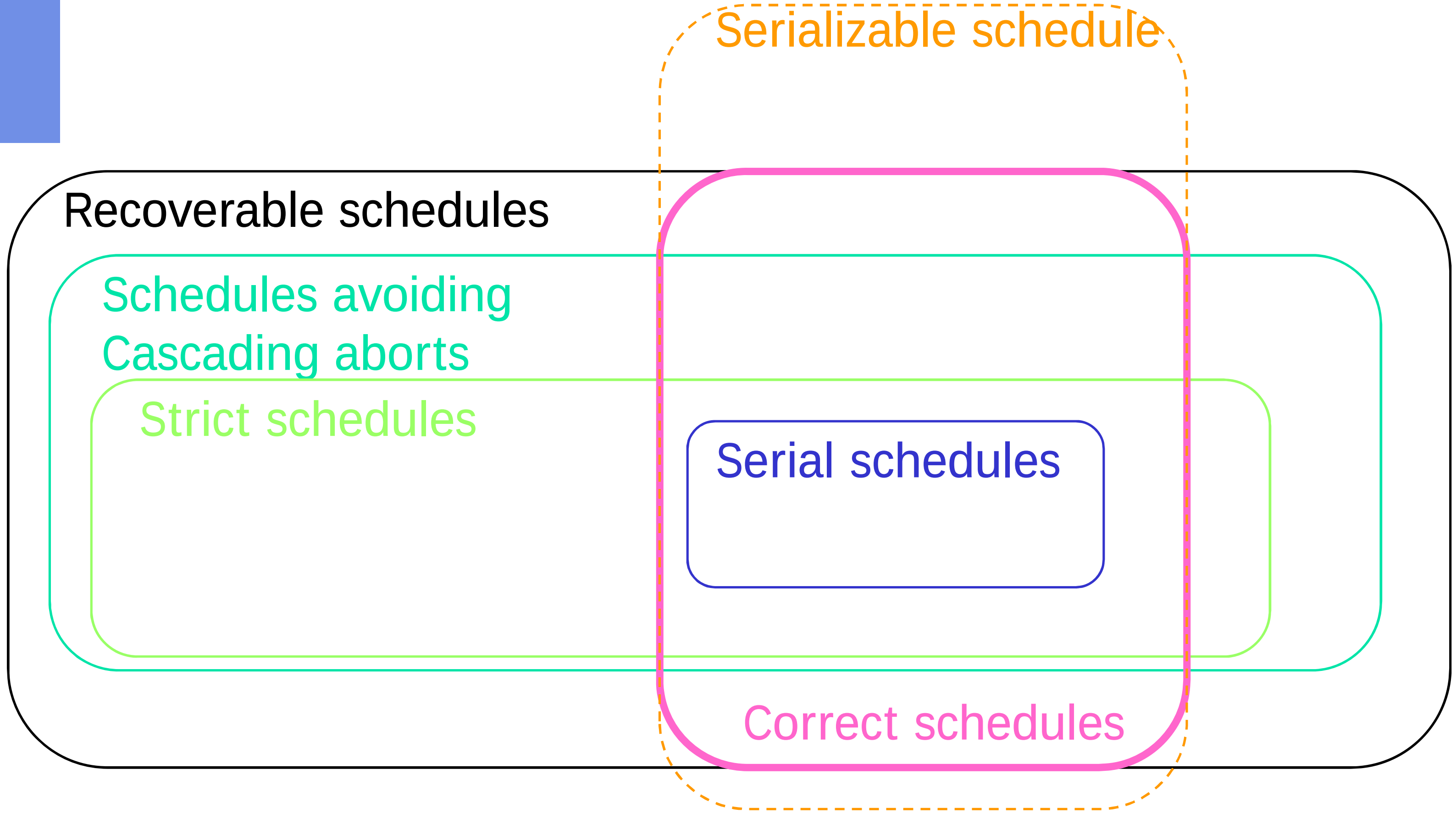
The scheduler can not alter the transaction operations of these concurrent transactions, but it can:

(1) Execute the transaction operation immediately

(2) Postpone its execution (changing the ordering)

(3) Reject its execution, thus aborting its transaction

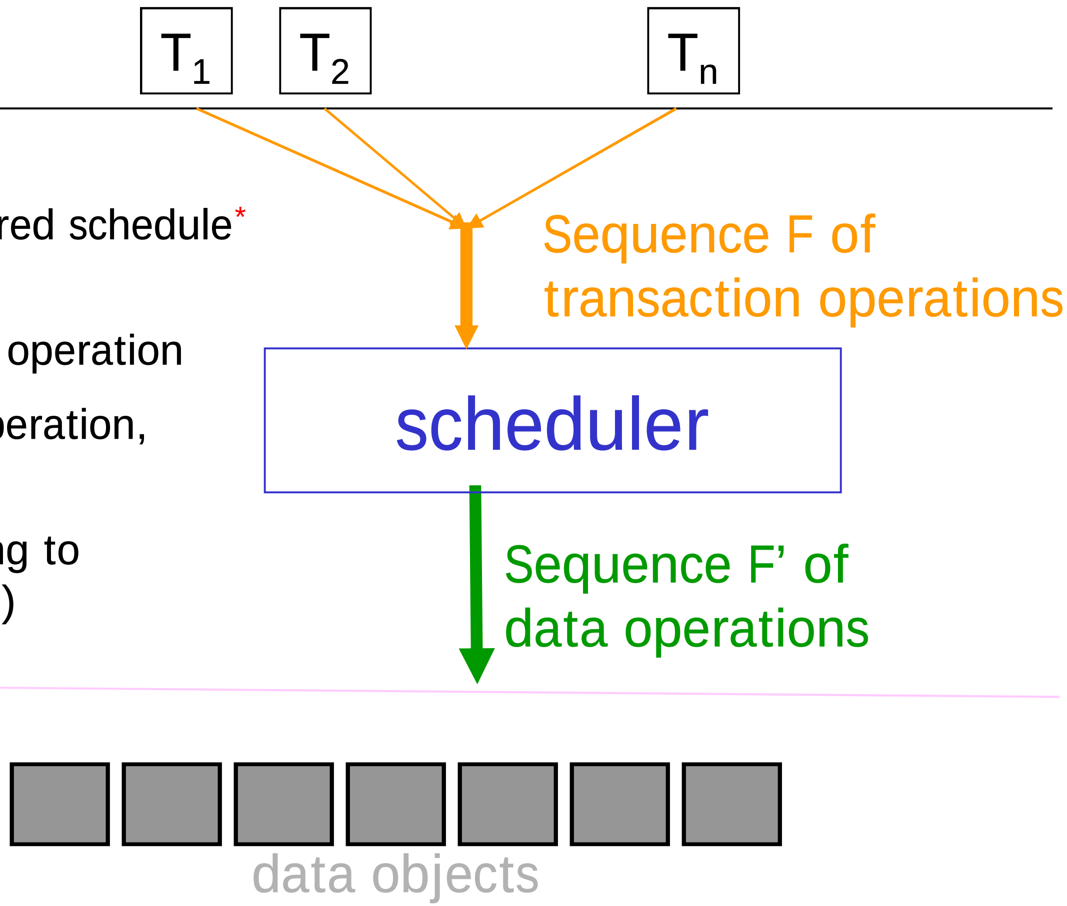
Summary of Schedules



Scheduler

Scheduler has to produce a desired schedule* (e.g. a serializable one):

1. Immediate execution of an operation
2. Delay of execution of an operation, to reorder the sequence
3. Refuse an operation (leading to an abort of the transaction)



*without affecting the operations

Reordering of Commuting Operations

The order of 2 commutative consecutive operations of different transactions within a schedule S can be changed without affecting the result of the reordered schedule S'. This reordering is called a *legal swap*.

Remark:

If schedule S can be transformed into a serial S' via some legal swaps, S is conflict serializable.

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
read(y)	
write(y)	
	read(y)
	write(y)

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
read(y)	
write(y)	
	read(y)
	write(y)

legal swap →

T_1	T_2
read(x)	
write(x)	
	read(x)
read(y)	
	write(x)
write(y)	
	read(y)
	write(y)

Are there further legal reordering possibilities leading to a serial schedule?

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
read(y)	
write(y)	
	read(y)
	write(y)

legal swap →

T_1	T_2
read(x)	
write(x)	
	read(x)
read(y)	
	write(x)
write(y)	
	read(y)
	write(y)

legal swap →

T_1	T_2
read(x)	
write(x)	
read(y)	
	read(x)
	write(x)
write(y)	
	read(y)
	write(y)

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
read(y)	
write(y)	
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
	read(x)
read(y)	
	write(x)
write(y)	
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
read(y)	
	read(x)
	write(x)
write(y)	
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
read(y)	
	read(x)
write(y)	
	write(x)
	read(y)
	write(y)

T_1	T_2
read(x)	
write(x)	
	read(x)
	write(x)
read(y)	
write(y)	
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
	read(x)
read(y)	
	write(x)
write(y)	
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
read(y)	
	read(x)
	write(x)
write(y)	
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
read(y)	
	read(x)
write(y)	
	write(x)
	read(y)
	write(y)

legal swap

T_1	T_2
read(x)	
write(x)	
read(y)	
write(y)	
	read(x)
	write(x)
	read(y)
	write(y)

Result: Serial schedule

Implementing Serializability

- Implementing serializability efficiently is to recognize conflicting versus commutative operations.

- Two main approaches:
  - Conservative, pessimistic protocol via locking mechanisms (similar to read/write locks)
  - Optimistic protocol via timestamps sometimes has to abort a transaction if a conflict is discovered
(see J. Bacon: Concurrent System, Chapt. 18)

We'll focus on pessimistic protocols

Conservative Approach

We need

- *Lock Types* for the “Data Items”

  (similar to those for the Reader/Writer Problem)

  and a

- *Locking Protocol*

  (establishing the serializability)

Two Lock Types

- *ReadLock* (shared lock)
  - ReadLocks may increase concurrency level
- *WriteLock* (exclusive lock)
  - WriteLocks may decrease concurrency level

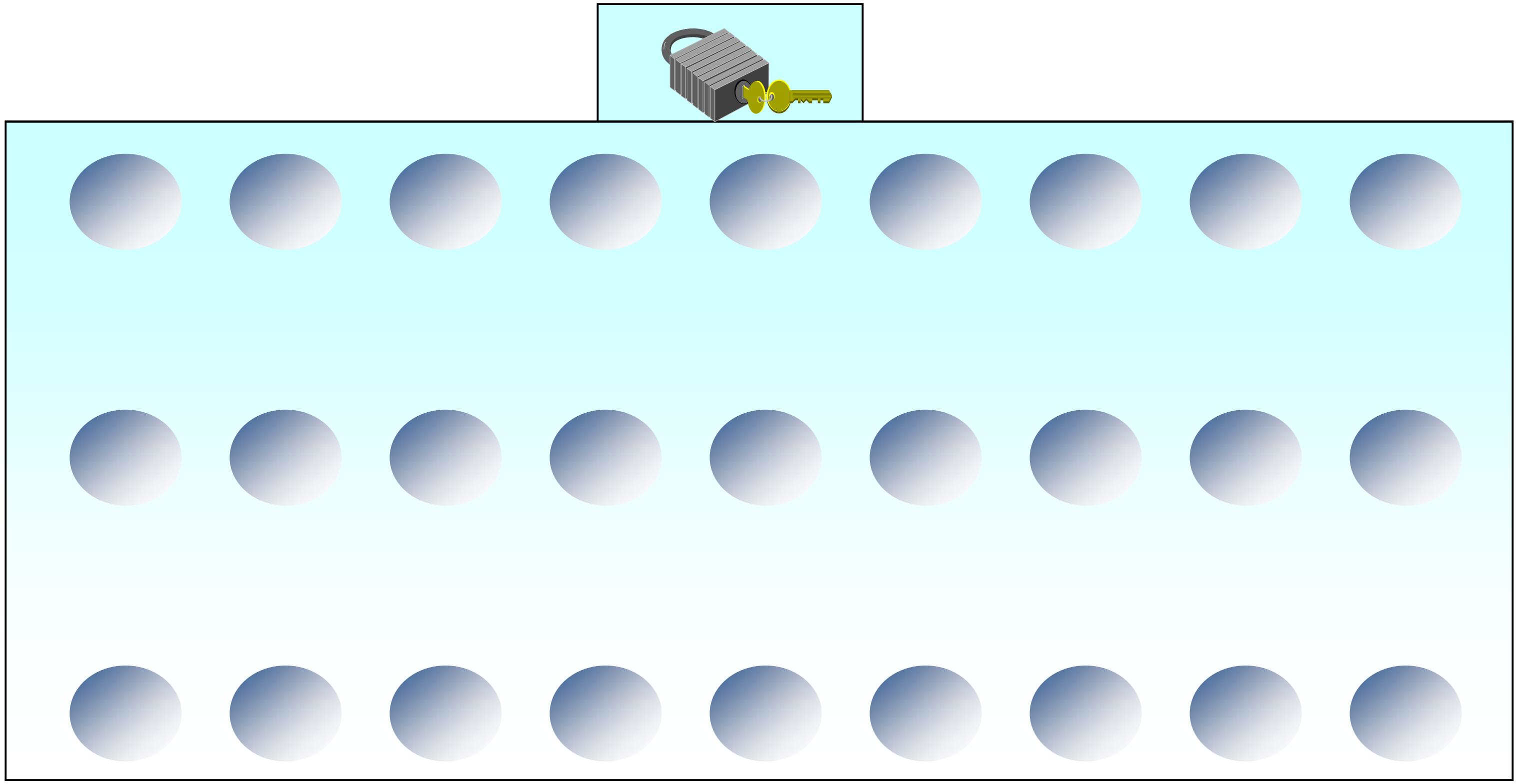
Discuss the semantics of both lock types!

Concurrency of the Two Lock Types

- ReadLock (shared lock)
- WriteLock (exclusive or conflicting lock)

Concurrently held Locks	ReadLock	WriteLock
ReadLock	yes	no
WriteLock	no	no

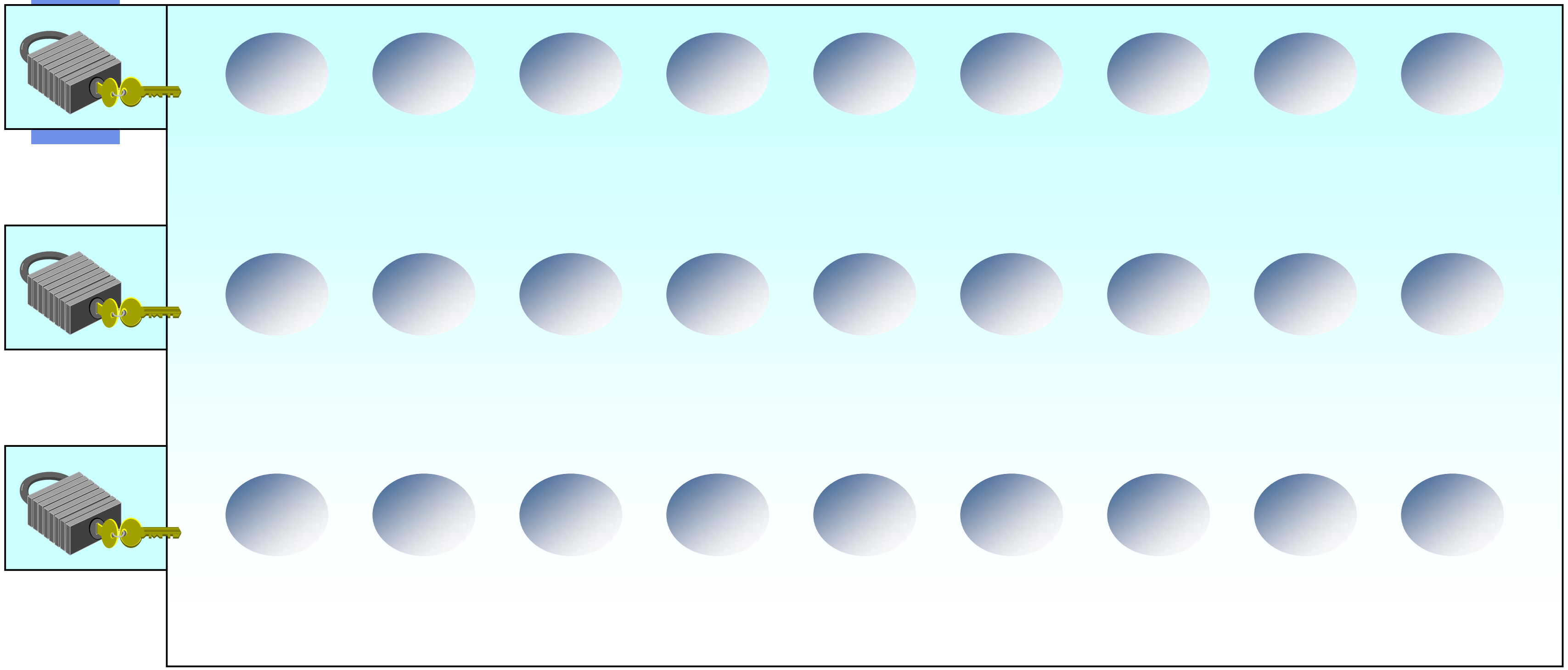
Locking Granularity in a Data Base



Pro: No deadlocks within the data base

Con: No concurrency at all

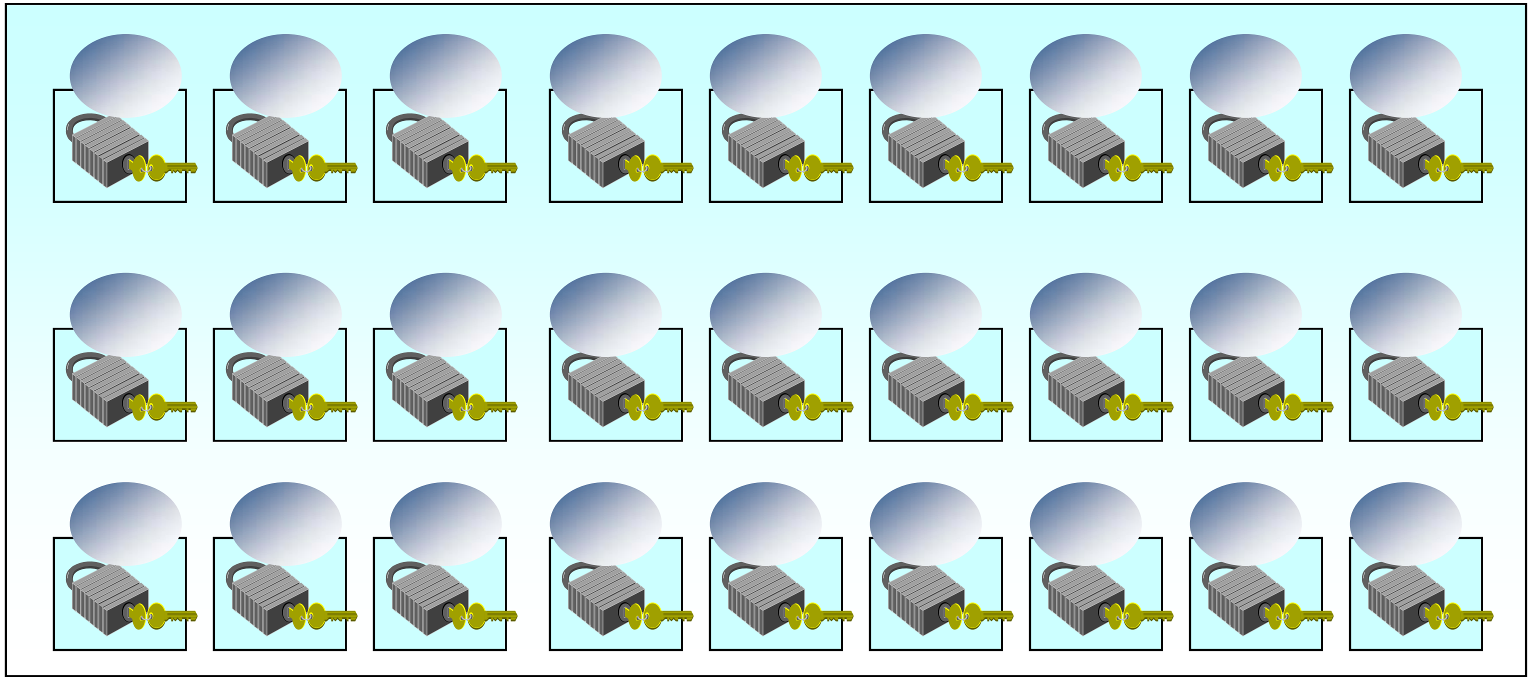
Locking Granularity in Data Base



Pro: Enhanced concurrency

Con: Enhanced danger of deadlocks, improved locking overhead

Locking Granularity in Data Base



Pro: Optimal concurrency

Con: Enhanced danger of deadlocks, maximal locking overhead

Two Phase Locking Protocol

Scheduler has to obey the following rules:

(1) Acquire a ReadLock before reading the data item

(2) Acquire a WriteLock before writing to the data item

(3) Conflicting locks block the invoking transaction

- RW, WR, WW

(4) Can not acquire another lock after releasing a previous one

Two Phase Locking Protocol

Result:

Guarantees that any two transactions which influence one another (RW, WR, WW) are serialized

- the conflict inducing transaction will be blocked
- releasing the lock will unblock the blocked transaction some time later

Two-Phase-Locking Protocol

1. Each transaction has to lock a data item with the appropriate lock type before accessing this data item
2. Wait until consistent locking is possible
3. After unlocking the first lock no further locking is allowed
4. Unlock all locks at the end of the transaction!!!

Requirement 3 determines the name of this protocol.
In the first phase, transactions can acquire their locks.
In the second phase, they only release their locks.

What's the basic idea behind requirement 3?

T1	T2
read_lock(x)	
read(x)	
read_unlock(x)	
	write_lock(x)
	write(x)
	write_lock(y)
	write(y)
	write_unlock(x)
	write_unlock(y)
	commit
write_lock(y)	
write(y)	
write_unlock(y)	
commit	

Remark: Due to the read_unlock(x) within T1 ∃ a so called cyclic dependency between T1 and T2, i.e. read1(x) < write2(x) and write2(y) < write1(y), leading to a non serializable schedule.

Drawback of Two-Phase-Locking Protocol

The normal two-phase-locking protocol enables serializable schedules, however, the schedule does not have to be recoverable.

Example:

T1	T2
write_lock(x)	
write(x)	
write_unlock(x)	
	read_lock(x)
	read(x)
	read_unlock(x)
	commit
abort	

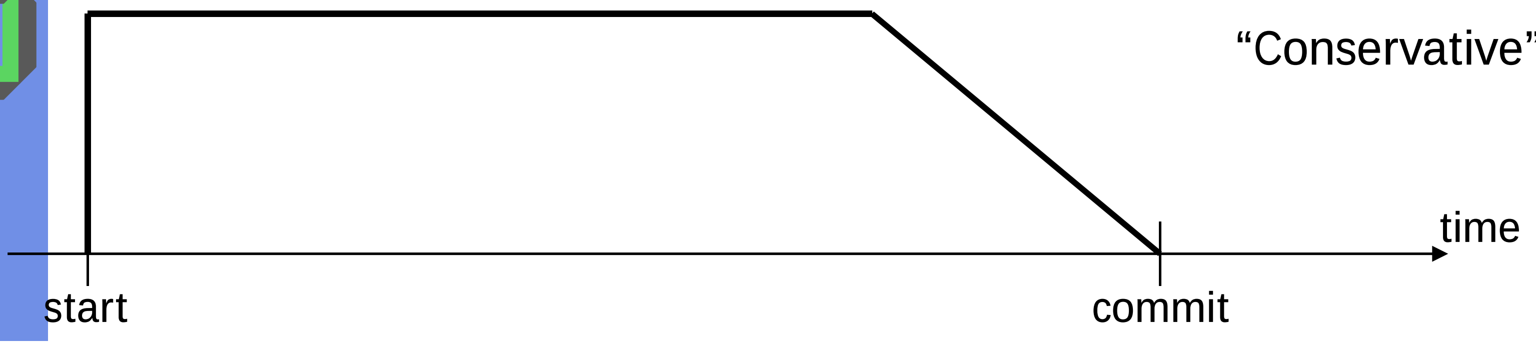
⇒ Introduce the following additional requirement:

5. All acquired locks are held until the end of the transaction.

Strict 2-Phase Locking Protocol

1. Each transaction has to lock a data item with the appropriate lock type before accessing this data item
2. Wait until consistent locking is possible
3. After unlocking the first lock no further locking is allowed
4. Unlock all locks at the end of the transaction!!!
5. All acquired locks are held until the end of the transaction.

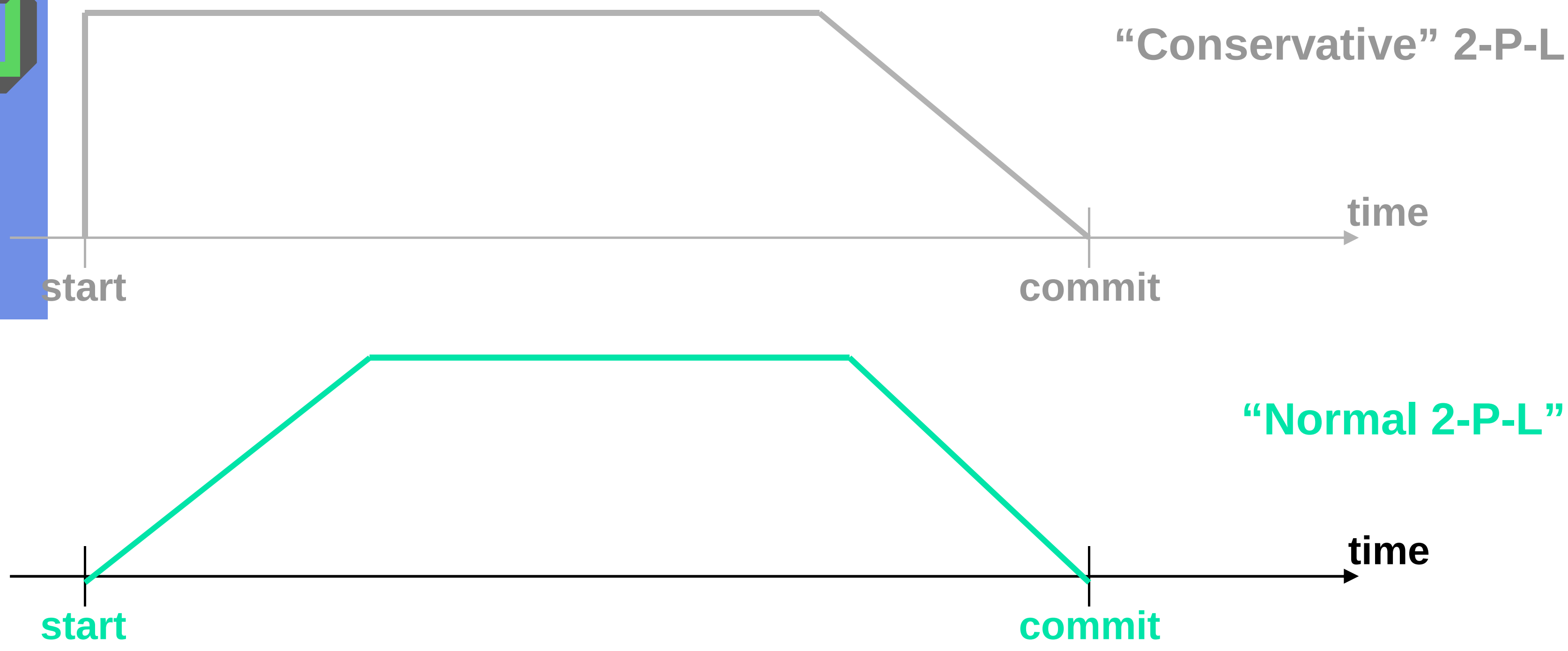
Result: A strict two-phase-locking protocol produces strict *schedules being recoverable and* avoiding cascading aborts



All locks needed within the transaction have to be set at start time!

Pro: No Deadlock

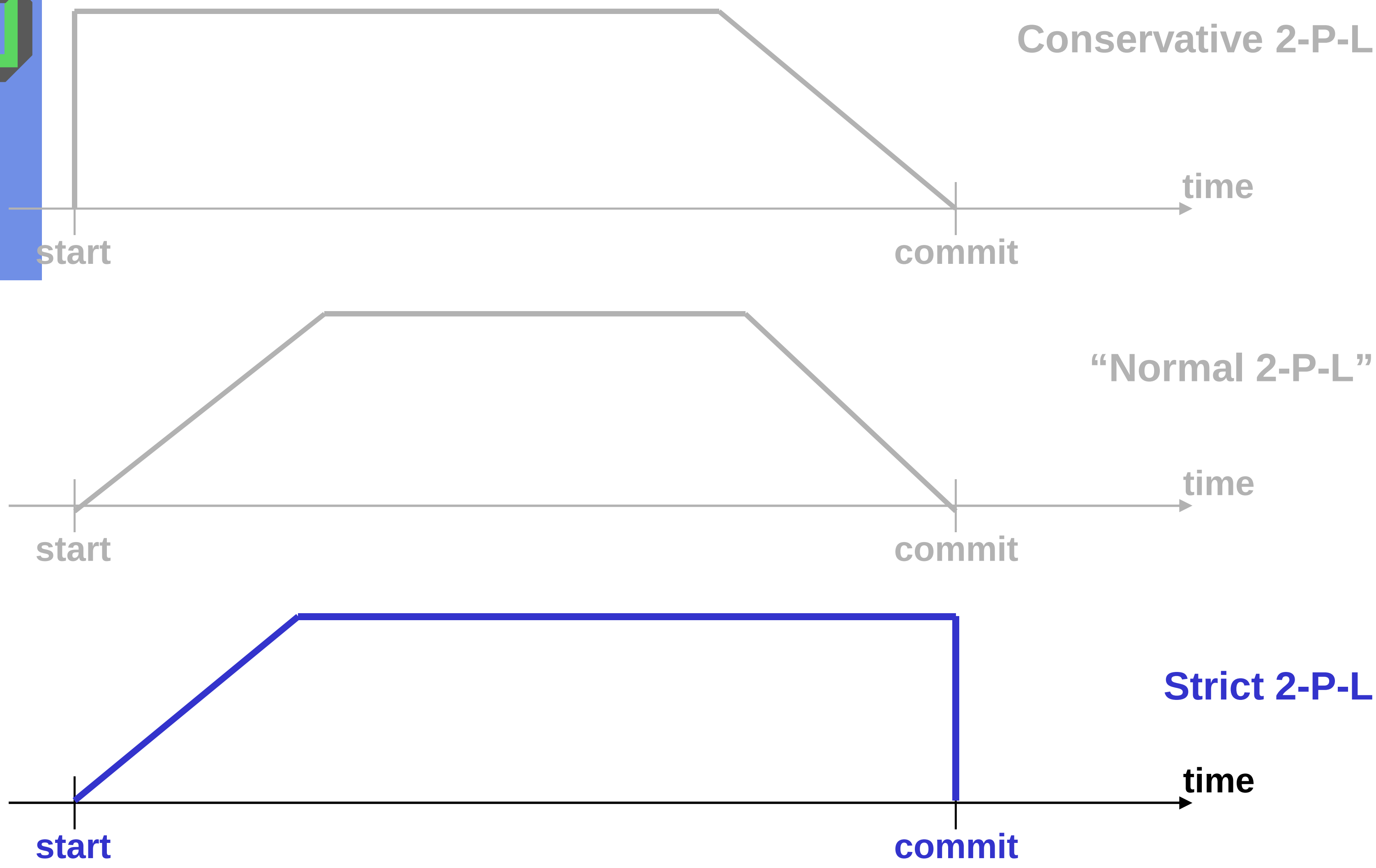
Con: All Locking must be known in advance, reduced Concurrency



Pro: Maximal Concurrency
Serializability

Con: Potential Deadlocks,
Not Recoverable

Transactions



Pro: Serializability + Recovery

Con: Reduced Concurrency
Potential Deadlocks

Locks and Deadlocks

Locks often increase the possibility of deadlocks

- T1 waits for T2 waits for T1
  - T1: Read1(x) Write1(y) Commit
  - T2: Write2(y) Write2(x) Commit
  - Schedule: ReadLock1(x) Read1(x) WriteLock2(y) Write2(y)
    WriteLock1(y) WriteLock2(x)

- Can also happen during "lock conversion"
  - T1: Read1(x) Write1(x) Commit
  - T2: Read2(x) Write2(x) Commit
  - Schedule: ReadLock1(x) Read1(x) ReadLock2(x)
    WriteLock1(x) WriteLock2(x)

Remark: Deadlock detection can be restricted to all blocking events

Problems with Abort due to a Deadlock

Increases load on the system:

- occurs at exactly the wrong time
- we already have contention, that’s why we possibly got a deadlock
- we have to retry the whole transaction

May have cascading aborts if we played it fast and loose with ReadLocks

- T1 sees T2’s actions sees T3’s actions ...
- if no abort, all is OK
- if T3 aborts, T2 aborts, T1 aborts

Improving Locking Performance

- Aborting long running transactions can really hurt
  - for example, reconciling a bank data base
- Weaker locking protocols can help
  - may not ensure serializability, but can be “close enough”
  - e.g. how much money does the bank have
- Degree 3 is “fully serializable”
  - repeatable reads
- Degree 2 serializability (cursor stability)
  - release a read lock immediately after use
  - can only see results of committed transactions
- Degree 1 serializability allows even reads on uncommitted data
  - “dirty reads”
  - no locks at all
  - the airlines??
- Updated transactions remain serializable

Short Overview on Recovery

We have to discuss the potential of failures on to

- Atomicity
- Durability

Types of Failures

- Transactions failures
- System crash
- Memory failures

Transaction Failures

Causes:

- Internal inconsistency or
- Decision of the transaction management system due to
  - Deadlock
  - External inconsistency
  - Scheduler

Actions:

- Undo transaction completely

System Crashes

Causes:

- OS failures
- Power failure
- Failure in the transaction management system

Actions:

- Recover the last committed state
- Redo committed but lost modifications
- Cancel all modifications of uncommitted transactions

Memory Failures

Causes:

- Bugs in Device Drivers
- Hardware faults: controller, bus, etc.
- Mechanical demolition (head crash)
- Losses in magnetism of disk surfaces

Actions:

- Copies of all data at different storage locations
- If database is not up to date, redo the effects of all meanwhile committed transactions

Principal Recovery Mechanisms

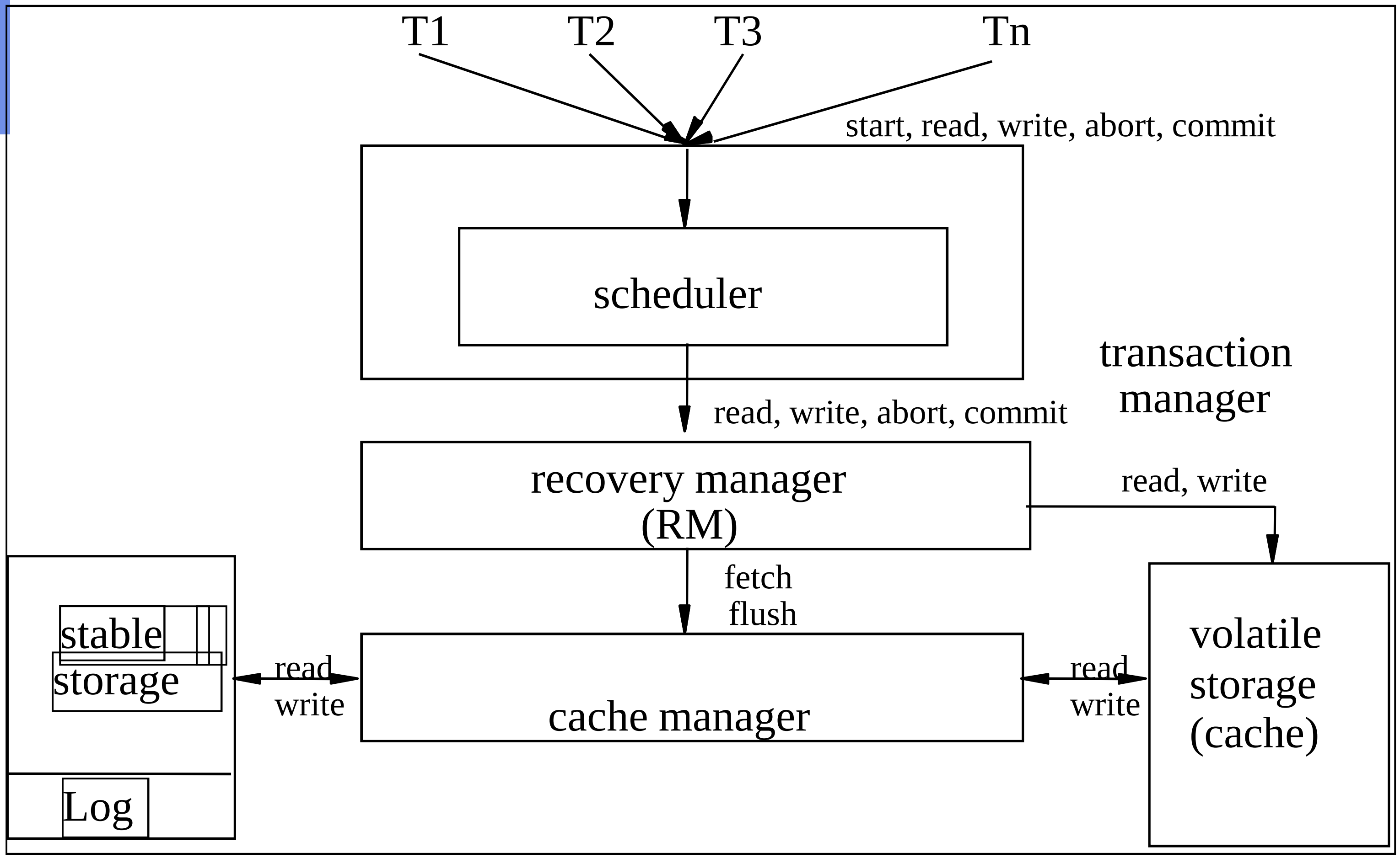
Logging

- Any state of a data object results in some sequence of operations
- If you log this sequence you can reproduce any intermediate state starting with some confirmed state
- Too much overhead that's why you note only periodical checkpoints
- The only 2 Operations needed:
  - Undo and
  - Redo

Shadowing

- If you modify some data you do not overwrite the old value of the data object, but you produce a new version
- If the transaction will be committed the new version is the only valid version

Interaction of different Components



The Log (write-ahead logging)

A log should represent the execution sequence of the transactions:

It contains entries of the following notion:

<Ti, x, v>, whereby
- Ti = TID of the transaction
- x = data object
- v = value (old, new)

for each write on the data base.

A log-entry is transferred to the stable storage immediately <u>before</u> the write on the database.

Other special log records exist to maintain significant events during transaction processing (e.g. start, commit, abort etc.)

The above log-entries are totally ordered and represent exactly the execution sequence of the transactions.

The Log (write-ahead logging)

The recovery algorithm uses the following two operations

- **undo(Ti)**, which restores the values of all data updated by Ti to the old values
- **redo(Ti)**, which sets the values of all data updated by Ti to the new values

Remark:

Both recovery operations must be idempotent,
i.e. multiple executions of an operation have the same result
as does only one execution

Simple Recovery Algorithm

- If transaction T aborts, we can restore the previous state of the data base by simply executing undo(T).
- If there is a system failure, we have to restore the state of all updated data by consulting the log to determine, which transaction need to redone and which transaction need to be undone.

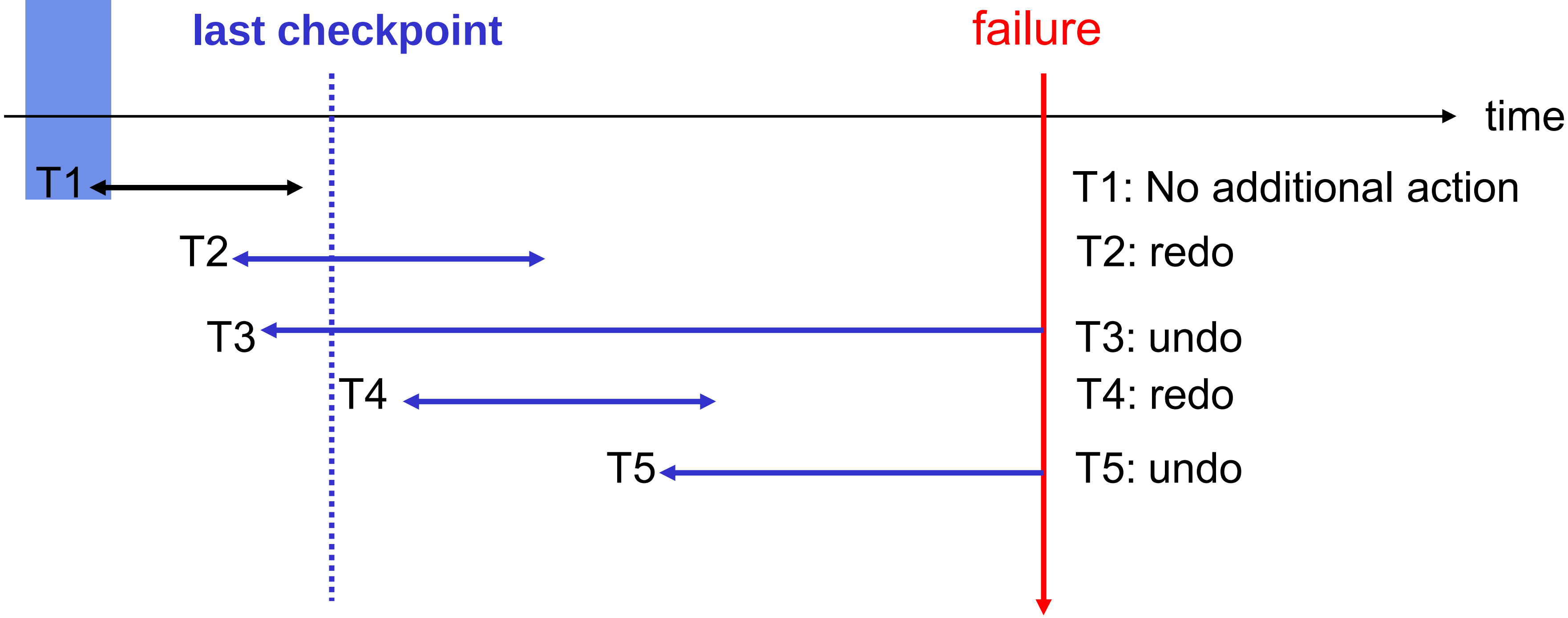
The following holds:

- Transaction T must be undone if the log contains the entry <T, start>, but does not contain the entry <T, commit>.
- Transaction T must be redone if the log contains both records <T, start> and <T, commit>

Analysis of Simple Recovery Algorithm

- When a system failure occurs in principle we have to search the entire log, which may be a bit time consuming.
- Furthermore some of the transaction have to be redone even though their updated results are already on disk.
- To avoid this type of overhead, we introduce periodical checkpoints with the following actions:
    1. Output all log records onto stable storage
    2. Output all modified data onto stable storage
    3. Output a log record <checkpoint> onto stable storage.

Transactions



Managing the Log

The recovery manager is based on the three lists:

- active list (all current transactions)
- commit list (all committed transactions)
- abort list (all aborted transactions)

The recovery manager deletes an entry (Ti, x, v)

- if the transaction Ti has been aborted
- if the transaction Ti has been committed <u>and</u>
- if some other transaction has overwritten x

  with a new value v'.

Cache Management

Due to performance reasons current data are kept in a buffer in main memory, i.e. in a volatile memory portion.

The buffer is organized as a set of cells each containing a complete disk block. Each cell has an dirty bit indicating whether the content in both storage media is identical.

The buffer maintains a buffer directory containing all current cells and cell numbers.

data item	cell number
x	2
y	1
:	:

buffer directory

cell number	dirty bit	content
1	0	'34589.56'
2	1	"New York"
:	:	:

buffer

Operations of Cache Manager

The cache manager supports the following four operations:

- Flush(c) If dirty-bit = set, cell c is written to disk
- Fetch(x) Select a cell c and copy x from disk to c, dirty bit(c) =0, and update buffer directory. If there is no free cell, select a cell c‘ and flush(c‘).
- Pin(c) Prevents flushing of cell c
- Unpin(c) Enables flushing of cell c

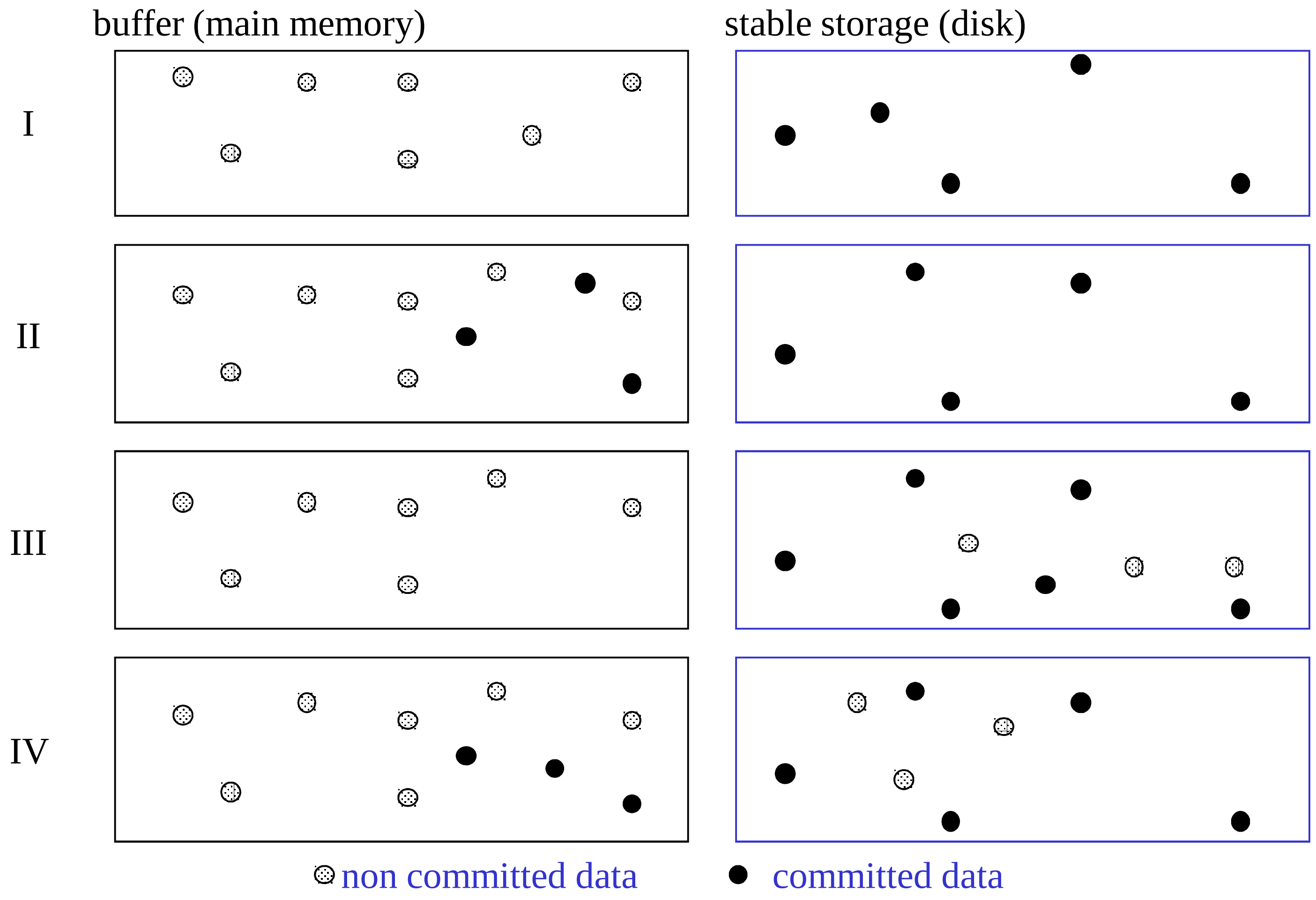
Recovery

- How to manage recovery depends on the way the resource and the cache manager handle data in the buffer.
- <u>Transfer of modifications:</u>
  - During a commit all modifications of a transaction are written to disk
  - After a commit of a transaction there are still some modified data in
  - the buffer not yet having been saved to disk
- <u>Replacement policy:</u>
  - All modifications of a transaction are kept in the buffer until a commit
  - The cache manager may even write uncommitted data to disk
- To keep overhead of recovery low it would be preferable if uncommitted data are kept only within the buffer and if committed data are only on disk.

  However, this results in an increased overhead.

  That‘s why in practice one chosses some compromise.

Situations in Volatile/Stable Storage



Overview on Recovery Mechanisms

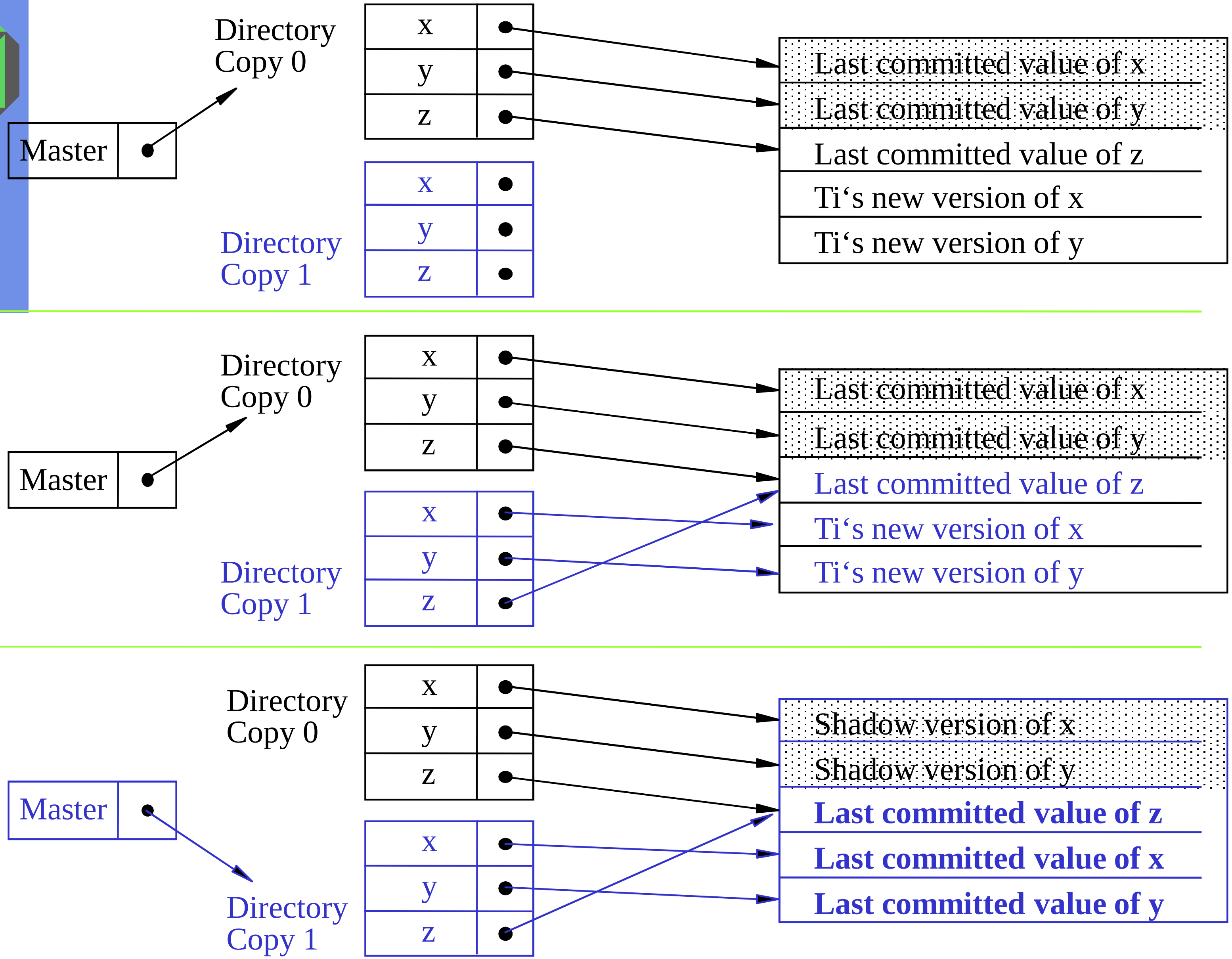
situation	types of data in the buffer	types of data on the disk	needed operations	needed data
I	only non committed	only committed	-	-
II	committed + non committed	only committed	redo	after images
III	only non committed	committed + non committed	undo	before images
IV	committed + non committed	committed + non committed	redo + undo	before + after images

Shadowing

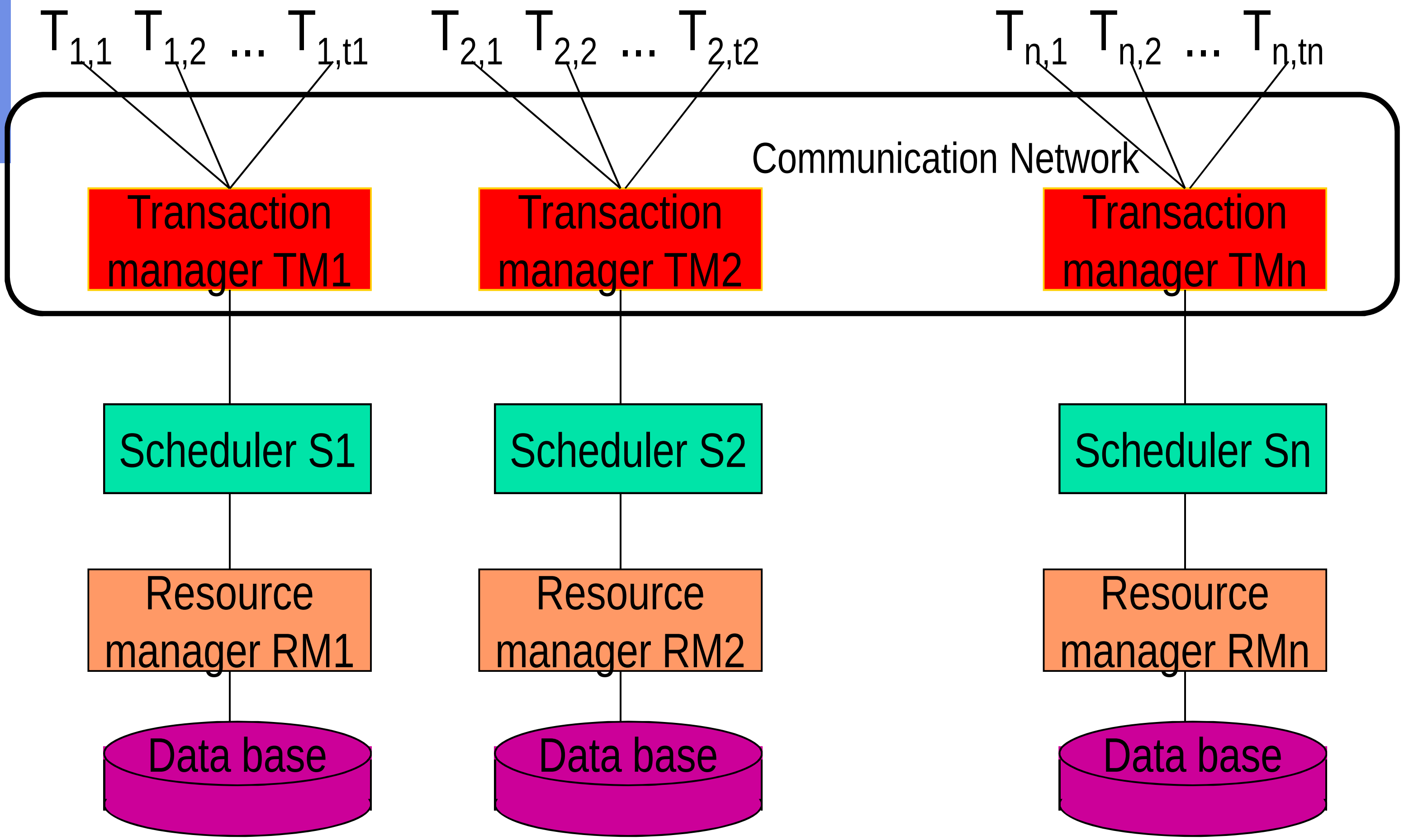
- Shadowing is an alternative to recovery via logging.
- The algorithm:
  - Modifications within a transaction don't overwrite the old value, they are producing a new version of the modified data item.
  - Each transaction maintains two directories with references to all its data items
  - One -the so called current- directory points to commited values, only.
  - The other directory points to the modified data items (the shadow versions)
  - With an abort of a transaction all shadow data are deleted.
  - With a commit the shadow directory takes the role of the current directory.

Remark: We have to guarantee that commit is an atomic operation (inclusive the writing to the disk).

Transactions



System of Transactions



Assumptions

- Homogeneous system,
  each node has a local transaction manager TM
- Each node manages its own data (no replicas)
- Each transaction send its operations to its local transaction manager TM
- If the data is not local, local TM sends request to remote TM
- On a commit and on an abort the TM has to notify all nodes, being affected by the transaction

Potential Failures

Node Failures:

- If node crashes, assume that the node stops immediately, i.e. it does not perform any operations anymore
- The content of volatile memory is lost and the node has to restart again
- A node is either active (i.e. working correctly) or inactive (i.e. does not respond anymore)

Potential Failures

Network Failures:

- Broken connection
- Faulty communication software
- Crashed intermediate node (bridge, gateway etc.)
- Lost of a message
- Altered message
- Partitioning of the network

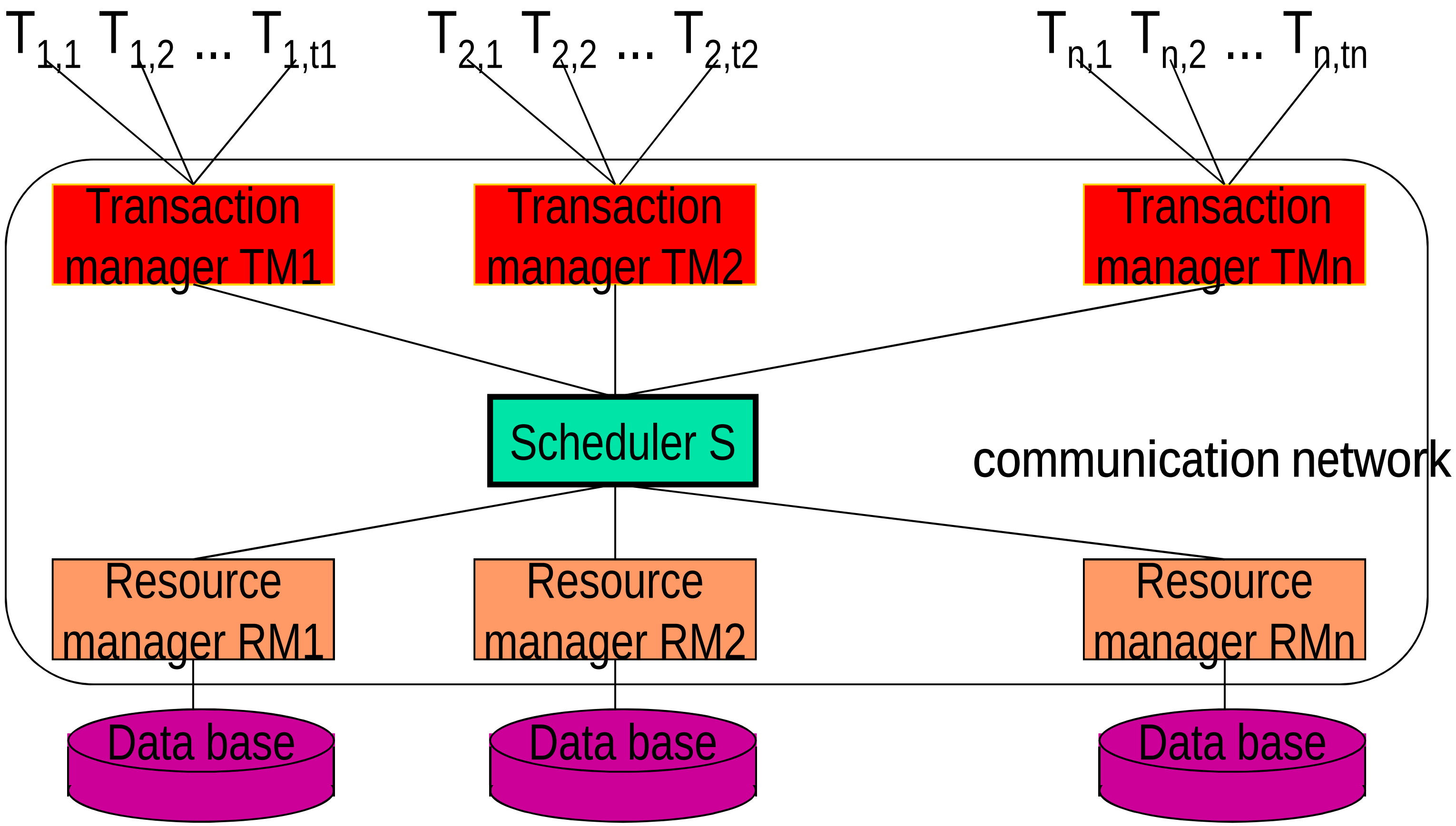
Managing Failures

- Many failures are handled on lower layers of the communication software
- However, a few of them have to be handled on layer 7 within the transaction manager
- The origin of failures on other nodes cannot be detected
- We have to rely on time outs, i.e. we only can conclude that there might be a failure

Coordination of Distributed Transactions

- Central Scheduler,
  i.e. one node is the only scheduler,
  responsible for granting or releasing locks
- Primary 2 phase locking each data item is
  assigned a primary copy
- Decentralized coordination

Centralized Scheduler



Centralized Scheduler

Analysis:

- We can use 2 phase locking protocol, S has a global view on all locks within the DS
- Single point of failure
- Scheduler may become a bottleneck (bad for scalability)
- Nodes are no longer really autonomous
- Even pure local transaction have to be sent to the central scheduler

This is the most common point of all drawbacks

This is the most inconvenient point of all drawbacks