DBS

Universität Karlsruhe (TH)

Transaktionsverwaltung

Einführung

Dozenten

- Vorlesung im Sommersemester 2007
- Prof. em. Dr. Peter Lockemann

  (Prof. Dr. Klemens Böhm dieses Jahr nicht)

  Institut für Programmstrukturen und Datenorganisation
- Teil des Vertiefungsfaches „Informationssysteme“

  Wird dort geprüft

Voraussetzungen

- Kommunikation und Datenhaltung *oder*
- Datenbankeinsatz *und*
- Grundkenntnisse in Graphentheorie

Organisatorisches (1)

- Webseite zur Vorlesung:
  - http://www.ipd.uka.de/~ipd/institut/tav/
    - Termine
    - Folien
    - Übungsblätter
    - externes Material
    - alles, was sonst noch relevant sein könnte

Organisatorisches (2)

- Kontakte
  - Prof. Lockemann: lockeman@ipd.uni-karlsruhe.de

    Informatikgebäude, Raum 344, Tel. 608-7358
  - Prof. Böhm: boehm@ipd.uni-karlsruhe.de

    Informatikgebäude, Sekretariat 3.OG, Tel. 608-3968
- Sprechstunden
  - Prof. Lockemann: **nach Vereinbarung**
  - Prof. Böhm: **Mi, 10:00 bis 11:00**
- Betreuer: Dipl.-Inform. Guido Sautter

Organisatorisches (3)

- Wo liegt die Bedeutung der Vorlesung?
  - Ein wichtiger Bestandteil des großen Themenbereichs Zuverlässigkeit der Informatiksysteme
  - Behandelt dieses Thema auf der schon eher anwendungsnahen Ebene der Geschäftsprozesse
  - Beschäftigt sich aber trotzdem mit technischen Maßnahmen
- TAV ist ein eher „haariger“ Stoff
    - ziemlich komplex
    - baut stark aufeinander auf
    - wenig „Wiedereinstiegspunkte“
  - deshalb: wichtig, nicht den Faden zu verlieren!!!!
  - gelingt am Ehesten, wenn man den Stoff nicht nur hört, sondern sich aktiv damit beschäftigt.

Organisatorisches (4)

- eigentlich gibt es keine Übung …
- aber: aktive Beschäftigung empfohlen …
- deshalb wird es bei Bedarf eine Übung geben:
  - Übungsblatt ca. 1 Woche vorher online
  - Lösungen können dann in der Übung besprochen werden

Organisatorisches (5)

- Folien werden vor der Vorlesung auf der Webseite verfügbar sein
  - Endfassung ggf. aber erst nach der Vorlesung wg. Einarbeitung von Anmerkungen und Fragen
- Folien sind zum beträchtlichen Teil auf Englisch
  - Vorlesung wird aber trotzdem auf Deutsch gehalten

Organisatorisches (6)

- Umfang 3 SWS
- Termin Vorlesung:
  - Donnerstag + Freitag, 08:00 – 09:30 Uhr
- Termin Übung
  - Bei Bedarf anstelle der Vorlesung
- Genaue Aufteilung der Termine auf Vorlesung und Übung findet sich auf der Vorlesungswebseite
  - Freie Termine werden rechtzeitig bekannt gegeben

Material zur Vorlesung

- Buch zur Vorlesung:
  - G. Weikum, G. Vossen: *Transactional Information Systems*
    - Gibt es in der Informatik-Bibliothek!
- Einzelne Folien entstammen direkt der Foliensammlung zu diesem Buch

Literatur

- Basis:
  - G. Weikum, G. Vossen: *Transactional Information Systems*. Morgan Kaufmann, 2002
- Mit starkem Einfluss:
  - P. A. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987
- Alle gängigen modernen Lehrbücher zu Datenbanksystemen haben ein umfangreiches Kapitel zu Transaktionen.
- Wer tief in die Technik einsteigen will:
  - J. Gray, A. Reuter: *Transaction Processing: Concepts and Technology*. Morgan Kaufmann Publishers, 1993

Inhalt

1. Transaktionen und ihre Eigenschaften
2. Modelle für Transaktionen
3. Isolation: Beispiele
4. Isolation: Korrektheit
5. Isolation: Serialisierbarkeit
6. Isolation: Synchronisation und Scheduling
7. Multiversionen-Synchronisation
8. Recovery
9. Verteilte Transaktionen: Eigenschaften
10. Verteilte Transaktionen: Synchronisation
11. Verteilte Transaktionen: Recovery
12. Lange Transaktionen

Lockemann (bis Ende Juni)

Lockemann (ab Ende Juni)

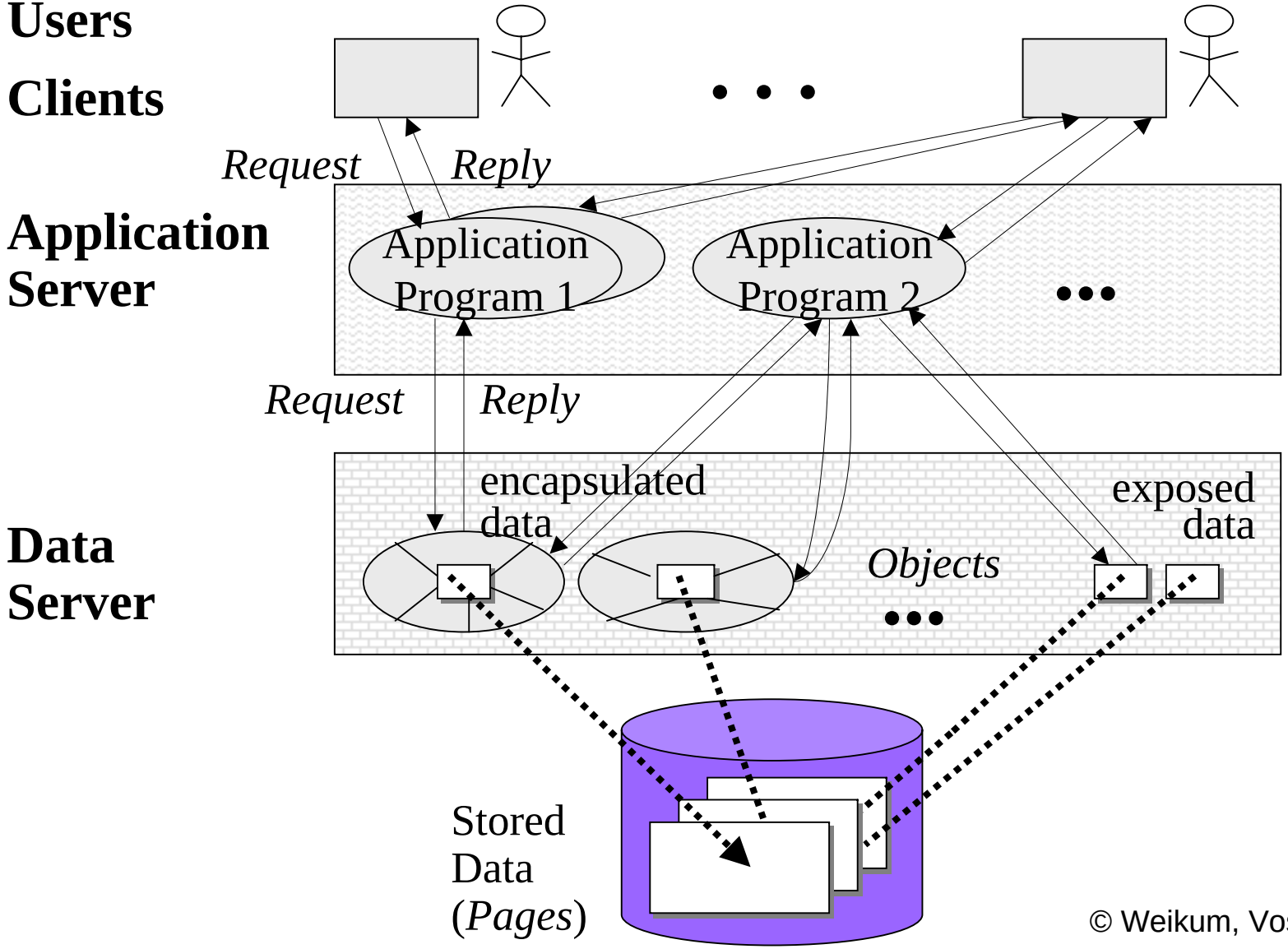
DBS

Universität Karlsruhe (TH)

Chapter 1

Transactions and transactional properties

3-Tier Reference Architecture



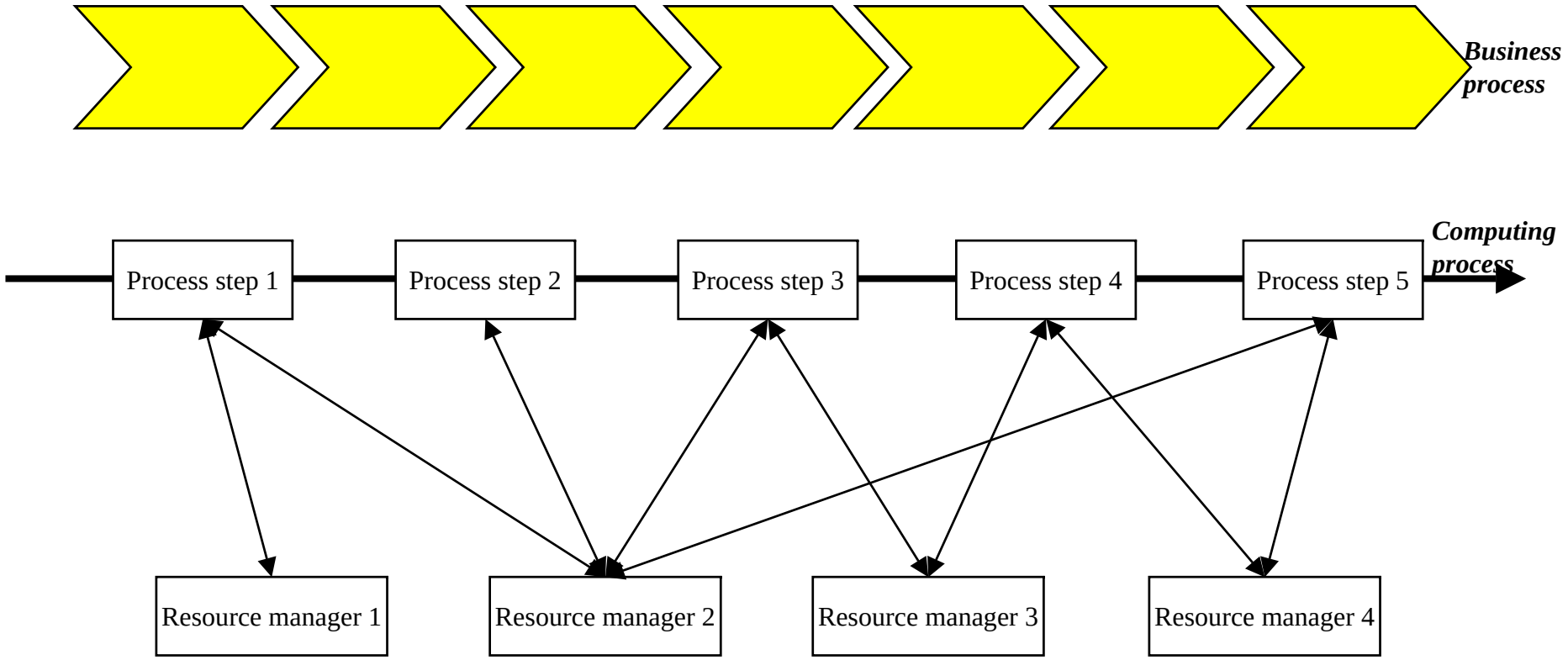
3-Tier Reference Architecture

- **Clients:**
  presentation (GUI, Internet browser)
- **Application server:** business processes
  - application programs (business objects, servlets)
  - request brokering (TP monitor, ORB, Web server)
    based on **middleware** (CORBA, DCOM, EJB, SOAP, etc.)
- **Data server:**
  database / (ADT) object / document / mail / etc. servers

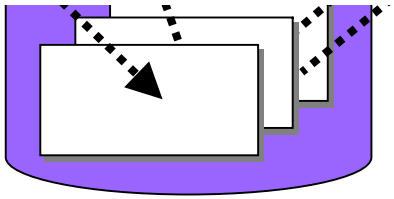
Specialization to 2-Tier Client-Server Architecture:

- Client-server with “fat” clients (app on client + ODBC)
- Client-server with “thin” clients (app on server, e.g., stored proc)

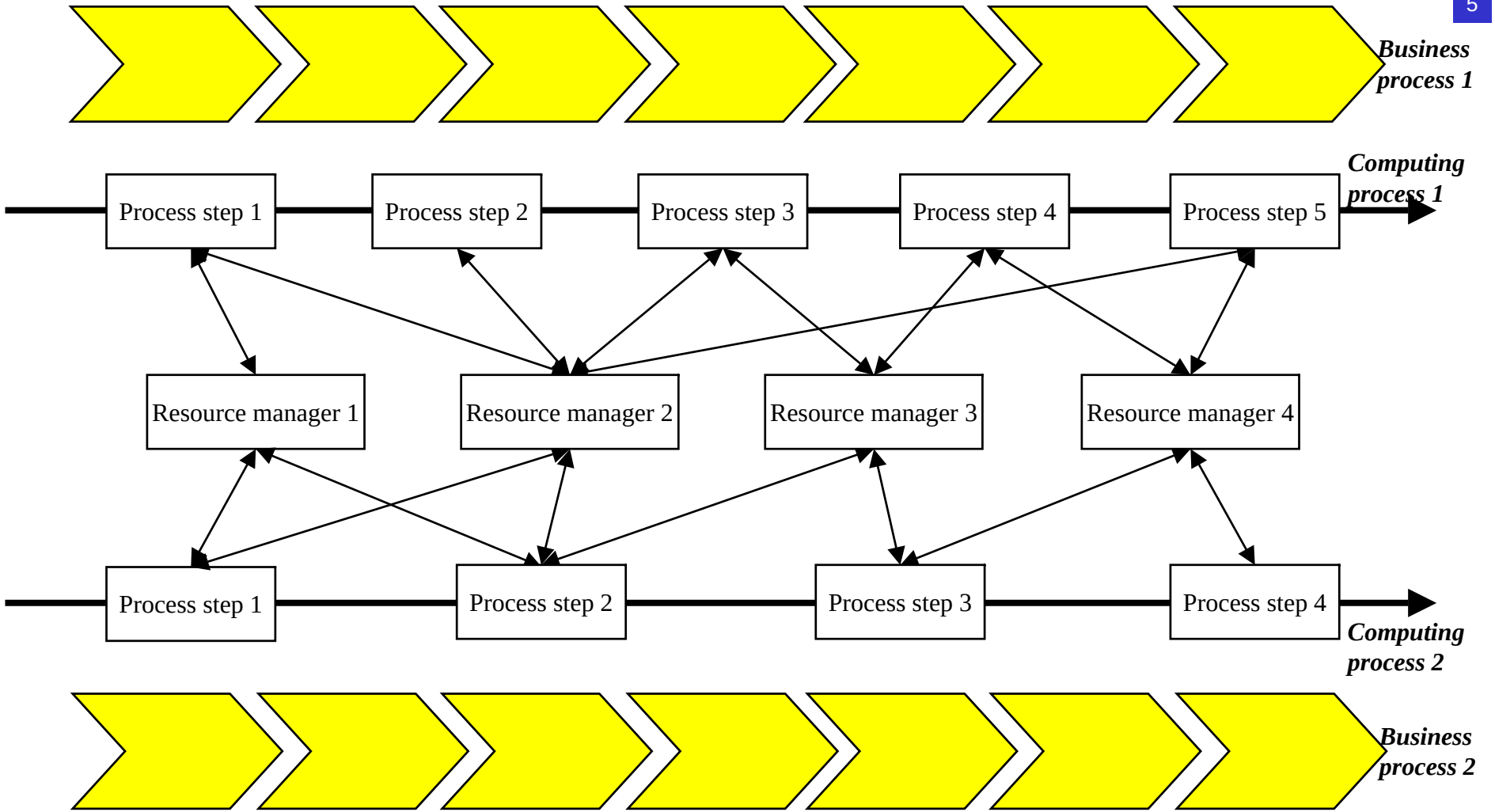
Process abstraction



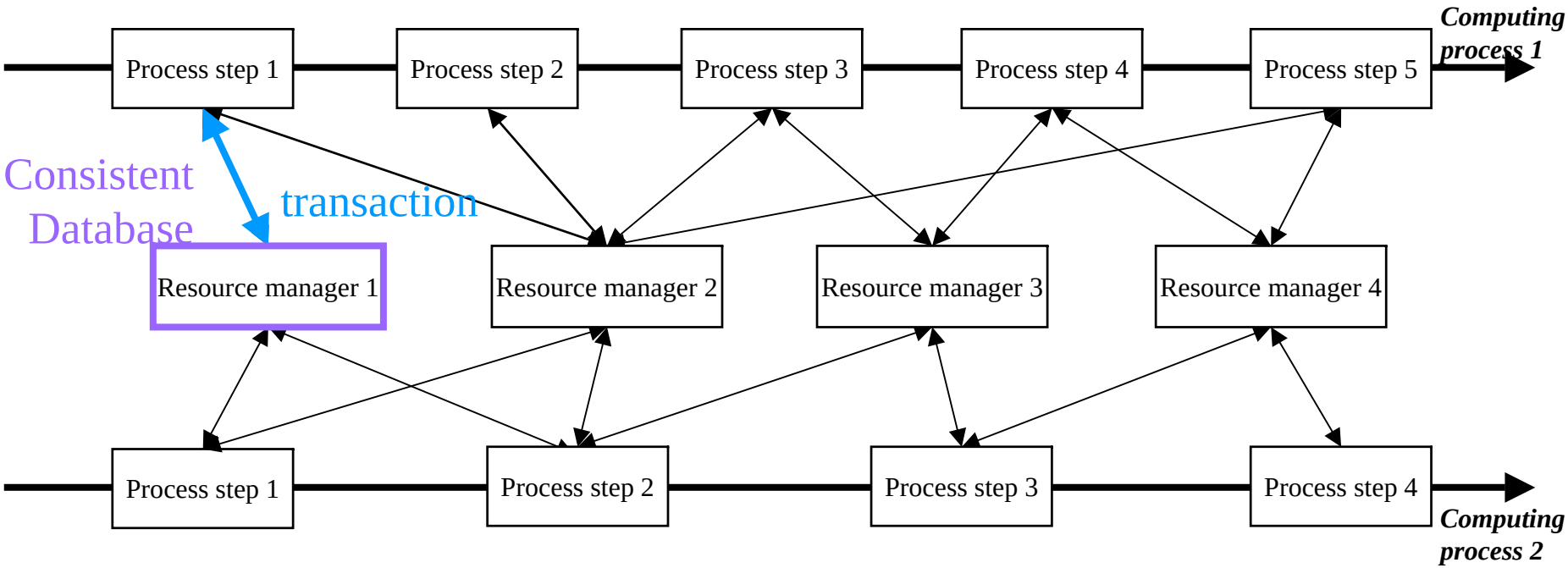
Stored Data (*Pages*)



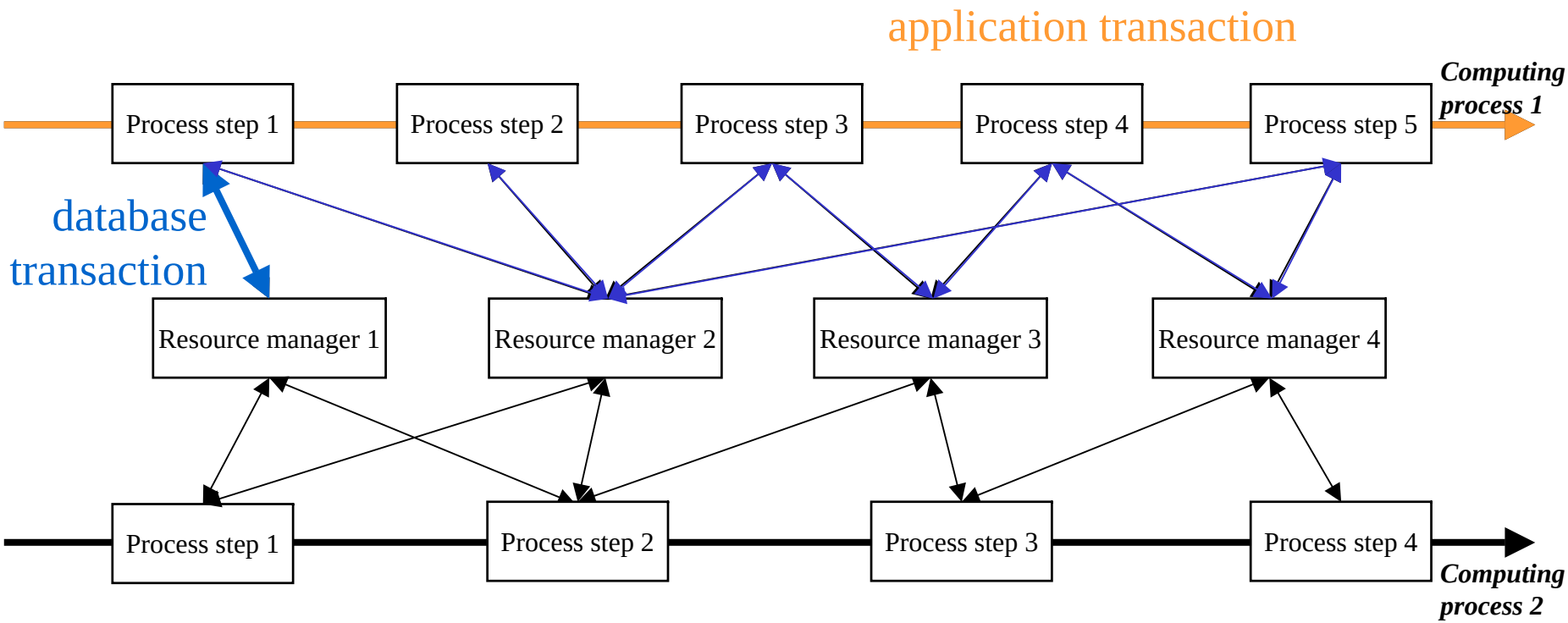
Process abstraction



Transactions (1)



Transactions (2)



ACID properties

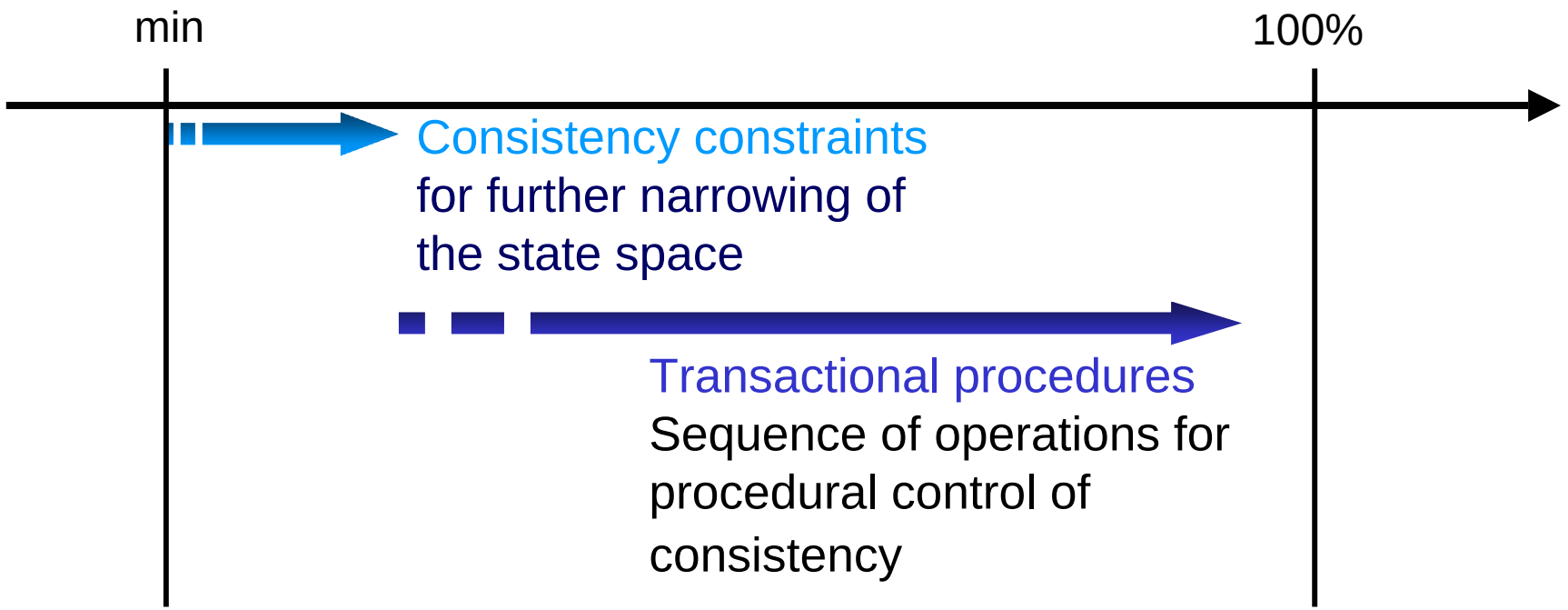
A atomicity

C consistency

I isolation

D durability

Consistency (1)



Legitimate states as a result of executing operators while interpretating the database schema

formal consistency

Ideal: full and up-to-date agreement between database and miniworld

semantic consistency

Consistency (2)

- **Database transaction** (for short: transaction): Execution of a transactional procedure on a single database.
  - A consequence: Consistency is only defined after completing the transaction.
- **Application transaction**: Consistent performance of a business process.
  - A consequence: Coordination of several database transactions needed.

A **transaction** is **consistent** if upon termination it produced a consistent database state.

ACID properties

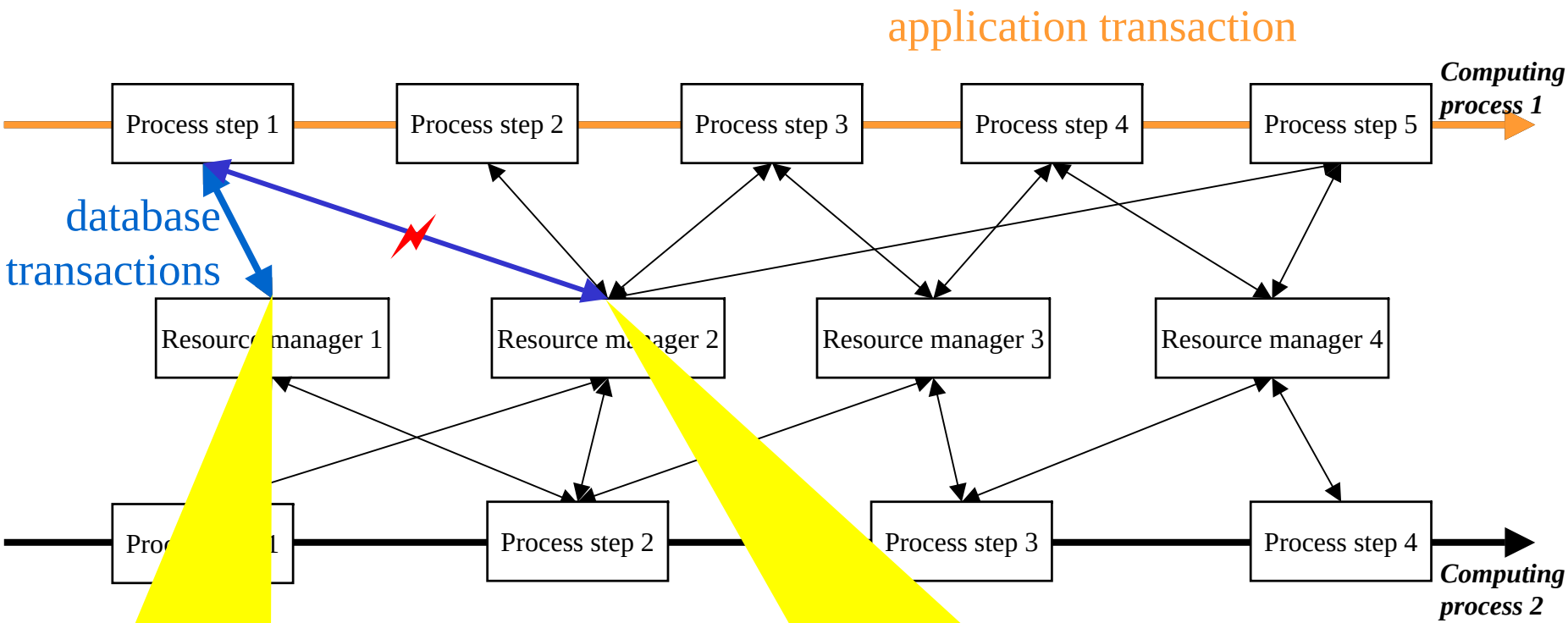
A atomicity

C consistency ✓

I isolation

D durability

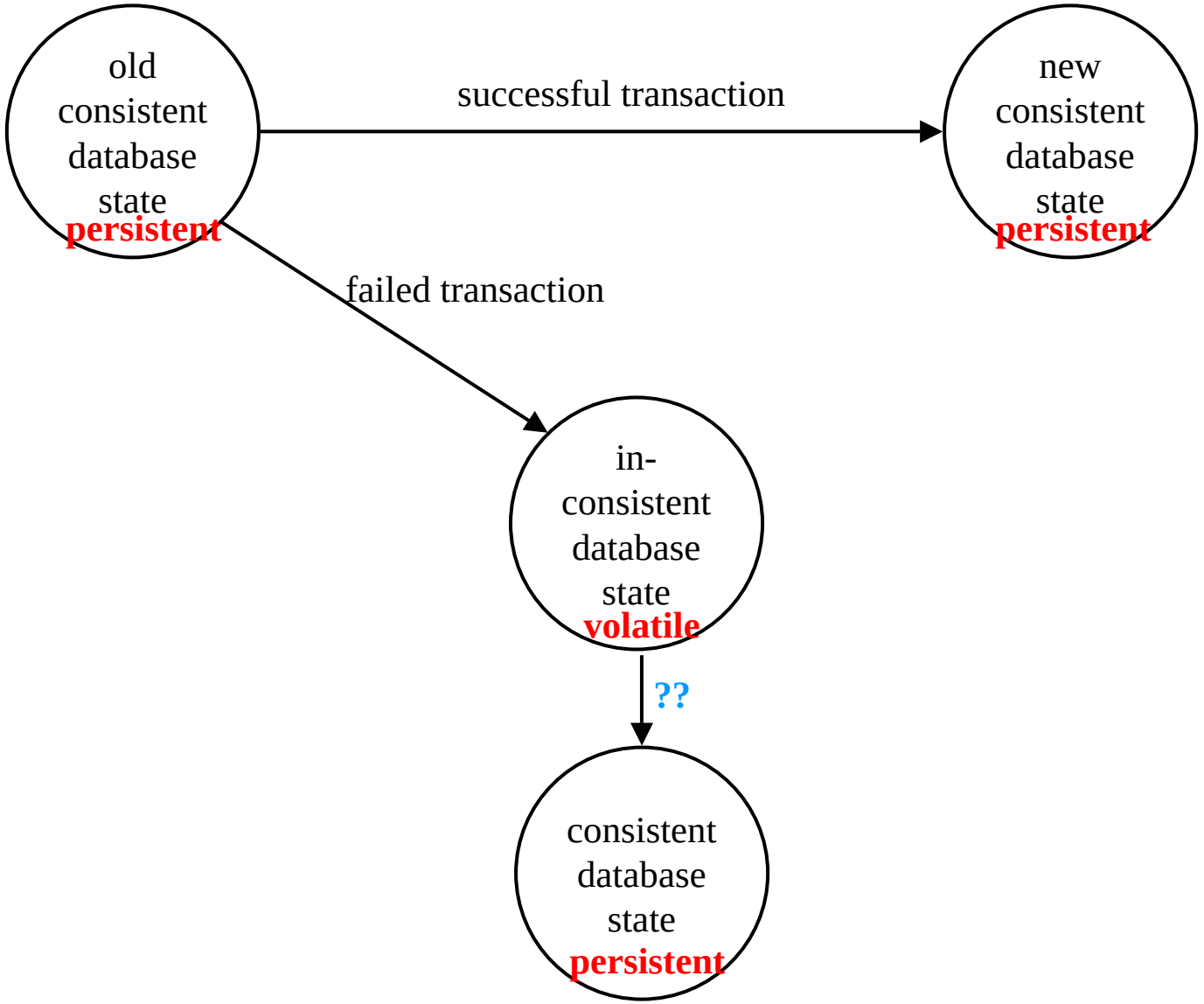
Atomicity (1)



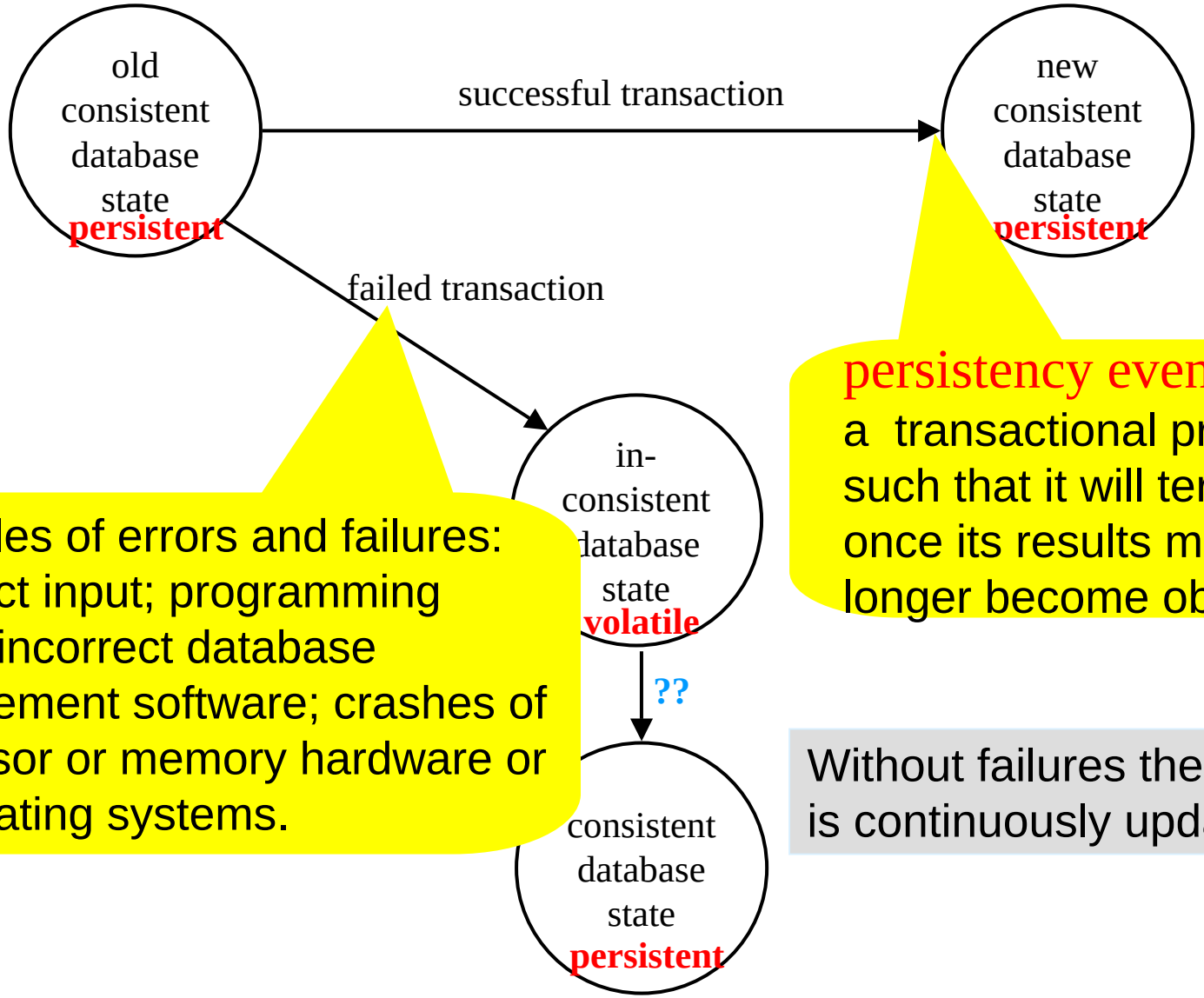
Persistency: The effect of a completed transaction cannot be lost again.

Failure resilience: A transaction reaches a well-defined consistent state even in the presence of disturbances.

Atomicity (2)


old consistent database state
persistent
successful transaction
new consistent database state
persistent
failed transaction
in-consistent database state
volatile
??
consistent database state
persistent

Atomicity (3)

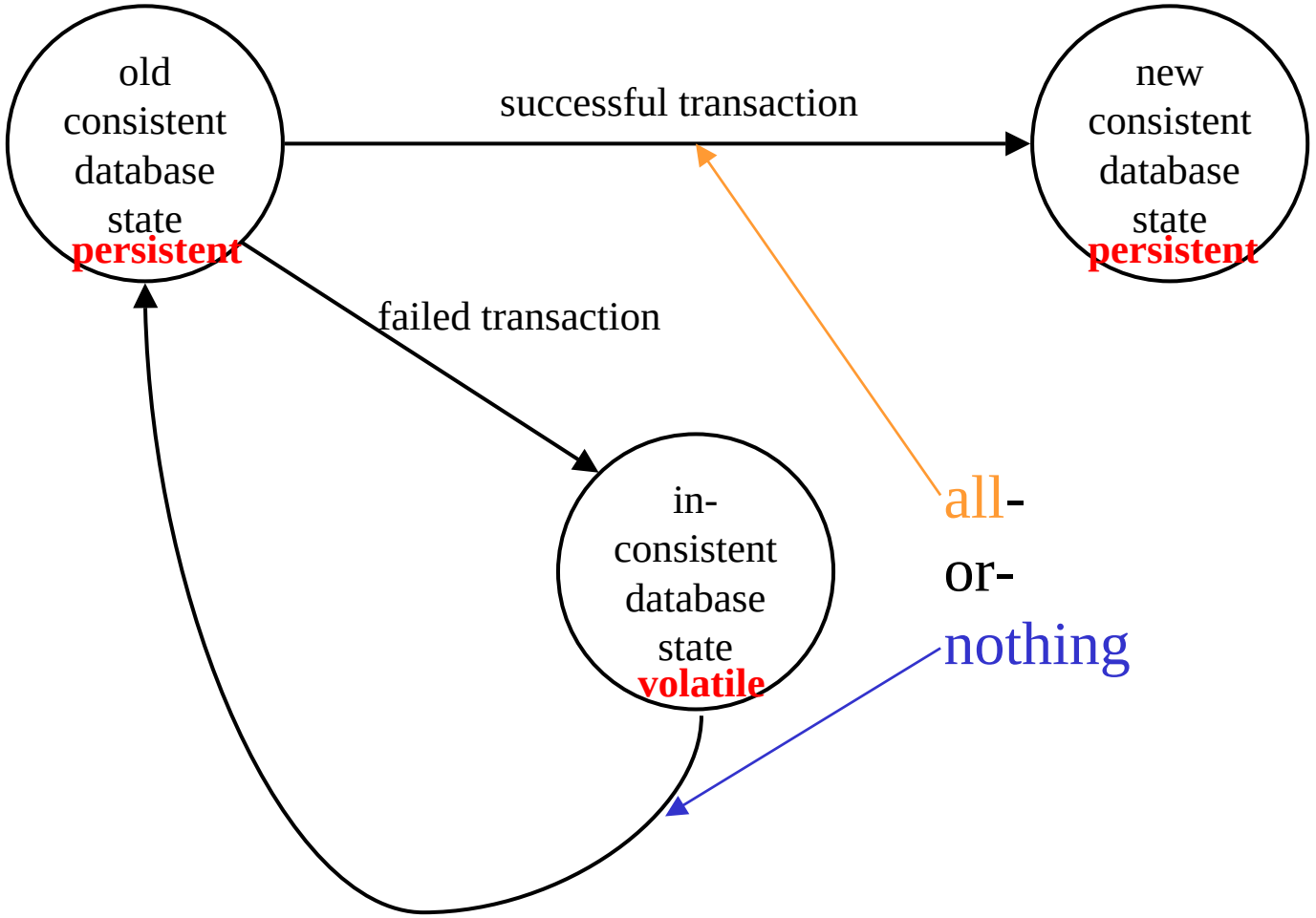


persistency event: design a transactional procedure such that it will terminate once its results must no longer become obsolete.

Examples of errors and failures: Incorrect input; programming errors; incorrect database management software; crashes of processor or memory hardware or of operating systems.

Without failures the database is continuously updated.

Atomicity (4)



A **transaction** is **atomic** if it has the all-or-nothing property.

Standard solution in case of failure.

ACID properties

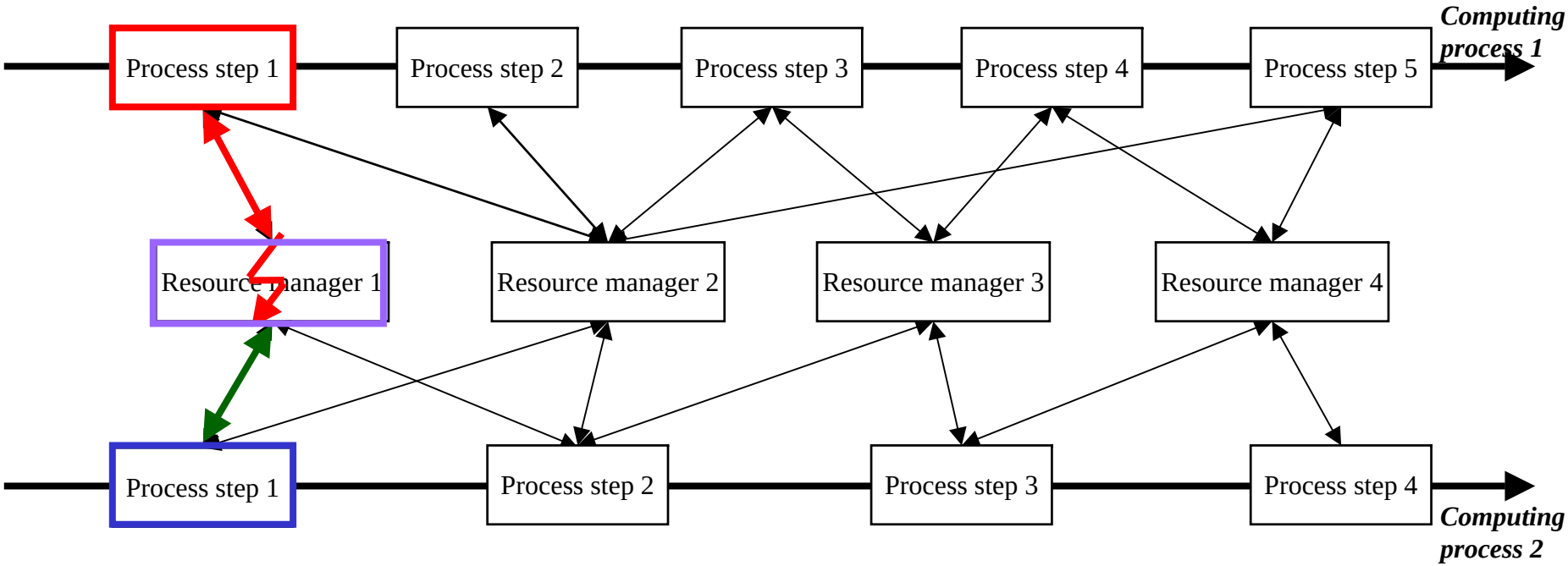
A atomicity ✓

C consistency ✓

I isolation

D durability

Isolation (1)



Threat to consistency: If several client transactions attempt to access the same resource concurrently (compete for the same resource) they may interact in undesirable ways: they are in **conflict**.

Isolation (2)

- **Needed**: Synchronization protocol that avoids conflicts between concurrent and competing client transactions within a database management system (DBMS): **conflict resilience**.
- **Correctness**: Concurrent database transactions are correctly synchronized if each one proceeds as if there was no competition, that is, if the only inconsistencies visible to a client are those caused by its own transaction, and each transaction again reaches a persistent database state.

ACID properties

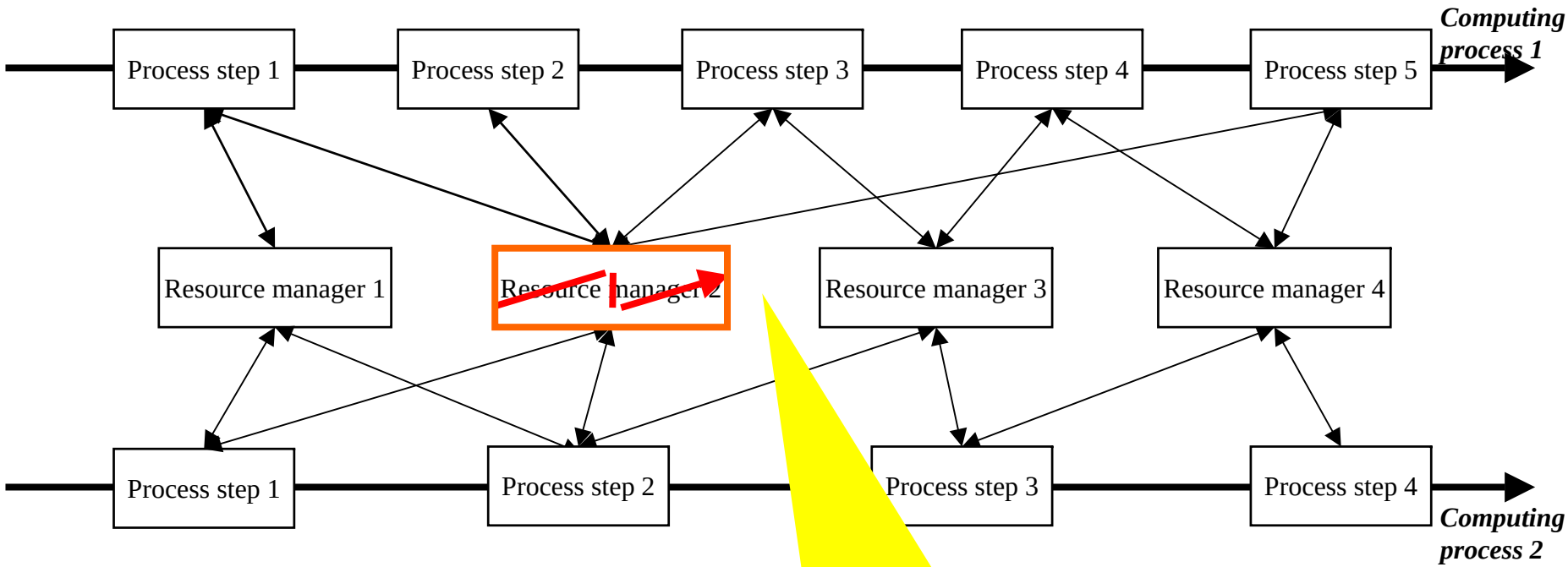
A atomicity ✓

C consistency ✓

I isolation ✓

D durability

Durability (1)



Persistency: Make sure on completion of a transaction that its effect cannot be lost.

Durability: Survival in case of loss non-volatile data: Make sure that the effect of a transaction ***never*** gets lost.

Durability (2)

- **Problem**: Storing data for an unlimited time makes consistency an even harder problem because of the increased probability of disturbances or failures occurring, with a concomitant corruption or complete loss of the data.
  - Sample threats: Failures of peripheral storage, storage media or processors; external events such as fire, water, climate; aging of media with ensuing destruction of the database.
- Durability: Overcome loss by somehow restoring an earlier, still useful database state.
  - Persistency is a prerequisite for durability.

Responsibilities (1)

A atomicity

C consisten

I isolation

D durability

Transaction management (**TM**): control of *a set* of potentially overlapping transactions with guarantees for consistency and robustness *for all of them*, taking into account further factors such as performance.

- **Transaction**: resilient execution of a consistent state transition (single operator or transactional procedure) with persistency of the final state.
  - Design time: **Transactional procedure**: Sequence of primitive operations considered as a unit of consistency and resilience.
  - Runtime: **Transaction** (**TA**): Execution of a transactional procedure, guaranteeing consistency and resilience.

Responsibilities (2)

A atomicity

C consistency

I isolation

D durability

- **Transaction**: resilient execution of a consistent state transition (single operator or transactional procedure) with persistency of the final state.
  - Design time: **Transactional procedure**: Sequence of primitive operations considered as a unit of consistency and resilience.
  - Runtime: **Transaction** (**TA**): Execution of a transactional procedure, guaranteeing consistency and resilience.

Responsibilities (3)

Basic assumption:

The transaction itself is responsible for **local consistency**: If undisturbed a transaction effects a consistent transition, i.e., results in a consistent database state provided it started from a consistent database state.

Responsibilities (4)

A atomicity

C consistency

I isolation

D durability

- **Transaction**: resilient execution of a consistent state transition (single operator or transactional procedure) with persistency of the final state.
  - Design time: **Transactional procedure**: Sequence of primitive operations considered as a unit of consistency and resilience.
  - Runtime: **Transaction** (**TA**): Execution of a transactional procedure, guaranteeing consistency and resilience.

Responsibilities (5)

- **Recovery**: Enforcement of persistency und failure resilience.
- **Atomicity**: The transaction shows an external effect only as a whole. Prior to successful completion there is no observable effect (transiency), after successful completion the effect is generally visible (persistency).
- Responsibilities:
  - Persistency: Transaction has already completed.
  - Failure resilience: Transaction has lost control.
    - Prime responsibility lies with the recovery manager as part of database management.

Responsibilities (6)

A atomicity

C consistency

I isolation

D durability

- **Transaction**: resilient execution of a consistent state transition (single operator or transactional procedure) with persistency of the final state.
  - Design time: **Transactional procedure**: Sequence of primitive operations considered as a unit of consistency and resilience.
  - Runtime: **Transaction** (**TA**): Execution of a transactional procedure, guaranteeing consistency and resilience.

Responsibilities (7)

- Local consistency is not sufficient to guarantee database consistency in a *set* of concurrent and competing transactions. We must enforce **global consistency** by conflict resilience:
  - **Isolation**: Concurrent transactions execute as if each would have its resources all by itself (no „mixing“ of state transitions).
    - Other concurrently executing transactions remain invisible to a given transaction.
- Responsibilities:
  - Conflict resilience: Transaction does not know other transactions.
    - The responsibility rests with the Scheduler as part of database management.

Responsibilities (8)

A atomicity

C consistency

I isolation

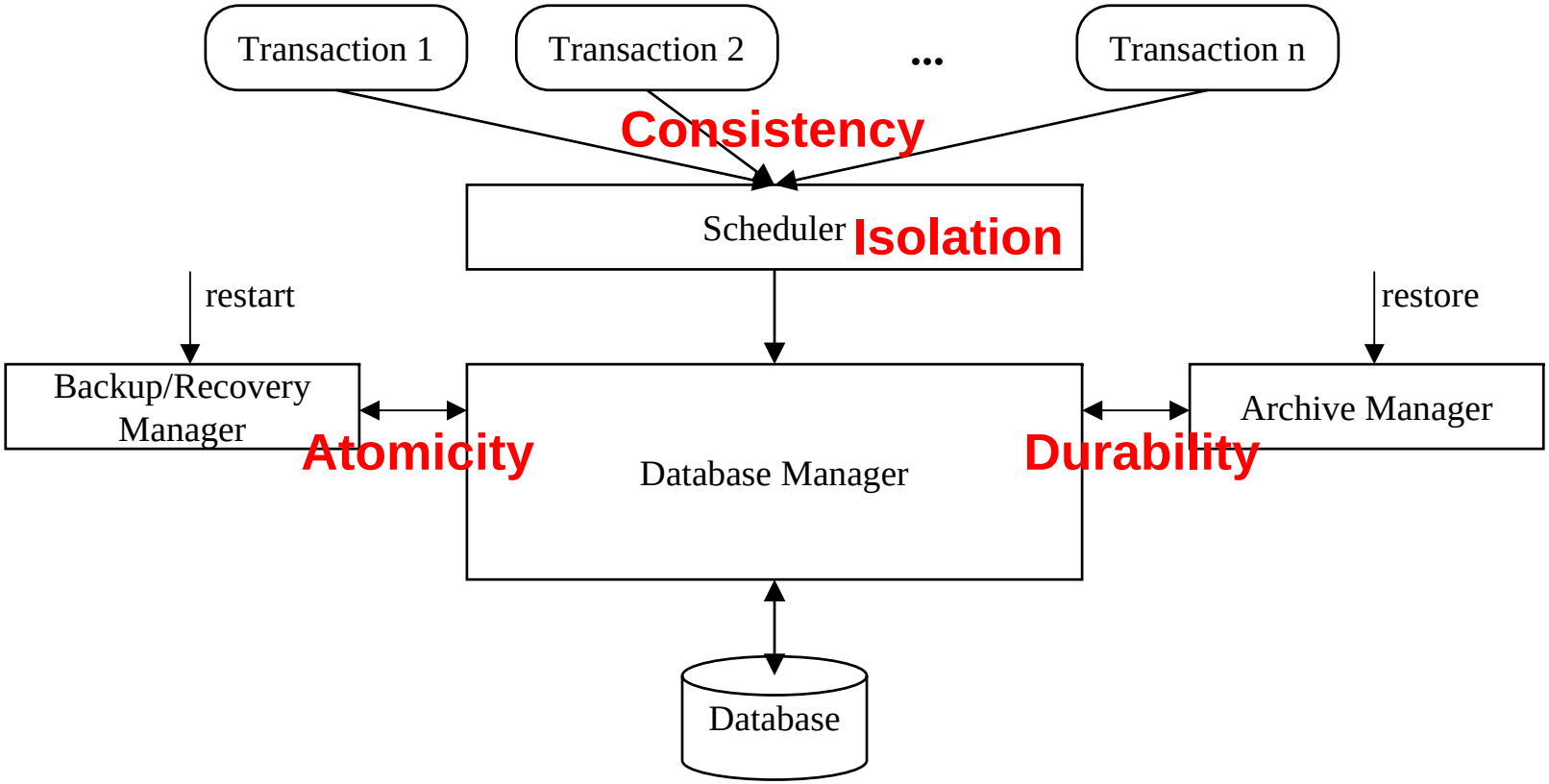
D durability

- **Transaction**: resilient execution of a consistent state transition (single operator or transactional procedure) with persistency of the final state.
  - Design time: **Transactional procedure**: Sequence of primitive operations considered as a unit of consistency and resilience.
  - Runtime: **Transaction** (**TA**): Execution of a transactional procedure, guaranteeing consistency and resilience.

Responsibilities (9)

- **Durability**: Long-duration persistency :
  - The effect of a successfully completed transaction is forever preserved unless it is explicitly renounced by a further transaction.
- Since the transaction does no longer exist after completion and may have occurred a long time ago, the responsibility can only rest with a separate system component (archive management).

System architecture



ACID good for all seasons?

- Useful for standard commercial applications where:
  - transactions are independent and of short duration (e.g., debit/credit transactions),
  - correctness requirements are high.
- Less useful for:
  - cooperating applications (e.g., CAD): Strict isolation makes no sense in cooperative development (e.g., collaborative design of an automobile engine), because intermediate results should be shared by other engineers.
  - long-lasting sessions (e.g., Internet reservations): For long-duration, resource-intensive transactions a complete undo of all work so far according to atomicity seems unacceptable.

DBS

Universität Karlsruhe (TH)

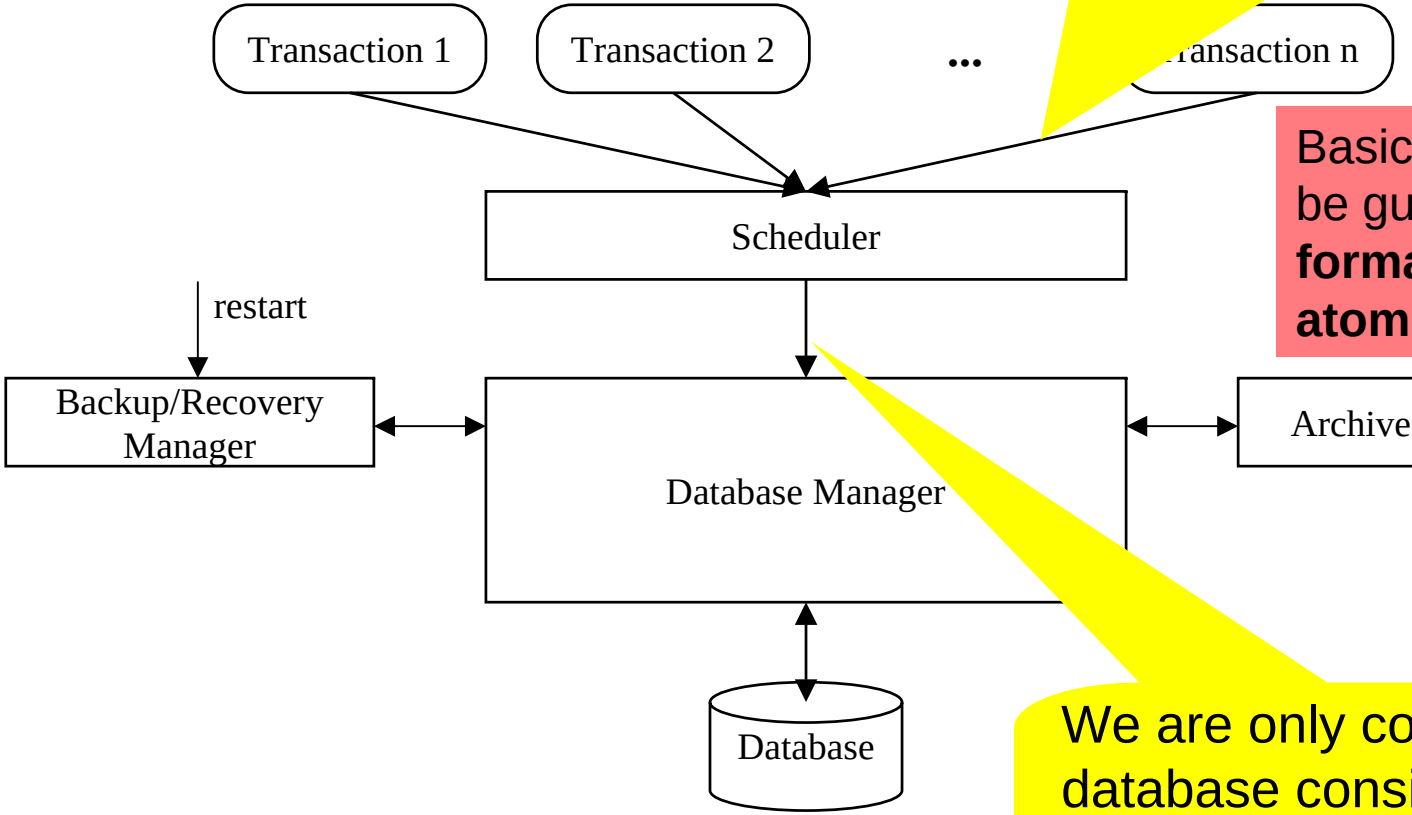
Chapter 2

Model for transactions

Read/write model

Basic **operations** in the model?

We abstract from transaction semantics except for database reads and writes.



Basic operations must be guaranteed to be **formal consistent**, **atomic** and **isolated**.

We are only concerned with database consistency ⇒ we are only interested in database reads and writes.

Read/write model

"Syntax":

Definition 2.1 (Read/Write Model Transaction):
A **transaction** t is a partial order of steps (actions) of the form r(x) or w(x), where x ∈ D (D database) and reads and writes applied to the same object are strictly ordered.
We write t = (op, <)
for transaction t with step set op and partial order <.

Examples: $t_1 = r(s) \rightarrow w(s) \rightarrow r(v) \rightarrow w(v)$

for short: r(s) w(s) r(v) w(v)

$t_2 = r(s) \rightarrow w(s)$
$r(v) \rightarrow w(v)$

Read/write model

"Syntax":

Definition 2.1 (Read/Write Model Transaction):
A **transaction** t is a partial order of steps (actions) of the form r(x) or w(x), where $x \in D$ (D database) and reads and writes applied to the same object are strictly ordered.
We write t = (op, <)
for transaction t with step set op and partial order <.

"Semantics":

Interpretation of j^{th} step, p_j, of t:
If p_j=r(x), then interpretation is assignment $v_j := x$ to local variable v_j
If p_j=w(x) then interpretation is assignment $x := f_j (v_{j1}, ..., v_{jk})$.
with unknown function f_j and $j_1, ..., j_k$ denoting t's prior read steps.

Worst-case assumption!

Read/write model

Examples: $t_1 = r(s) \rightarrow w(s) \rightarrow r(v) \rightarrow w(v)$

for short: r(s) w(s) r(v) w(v)

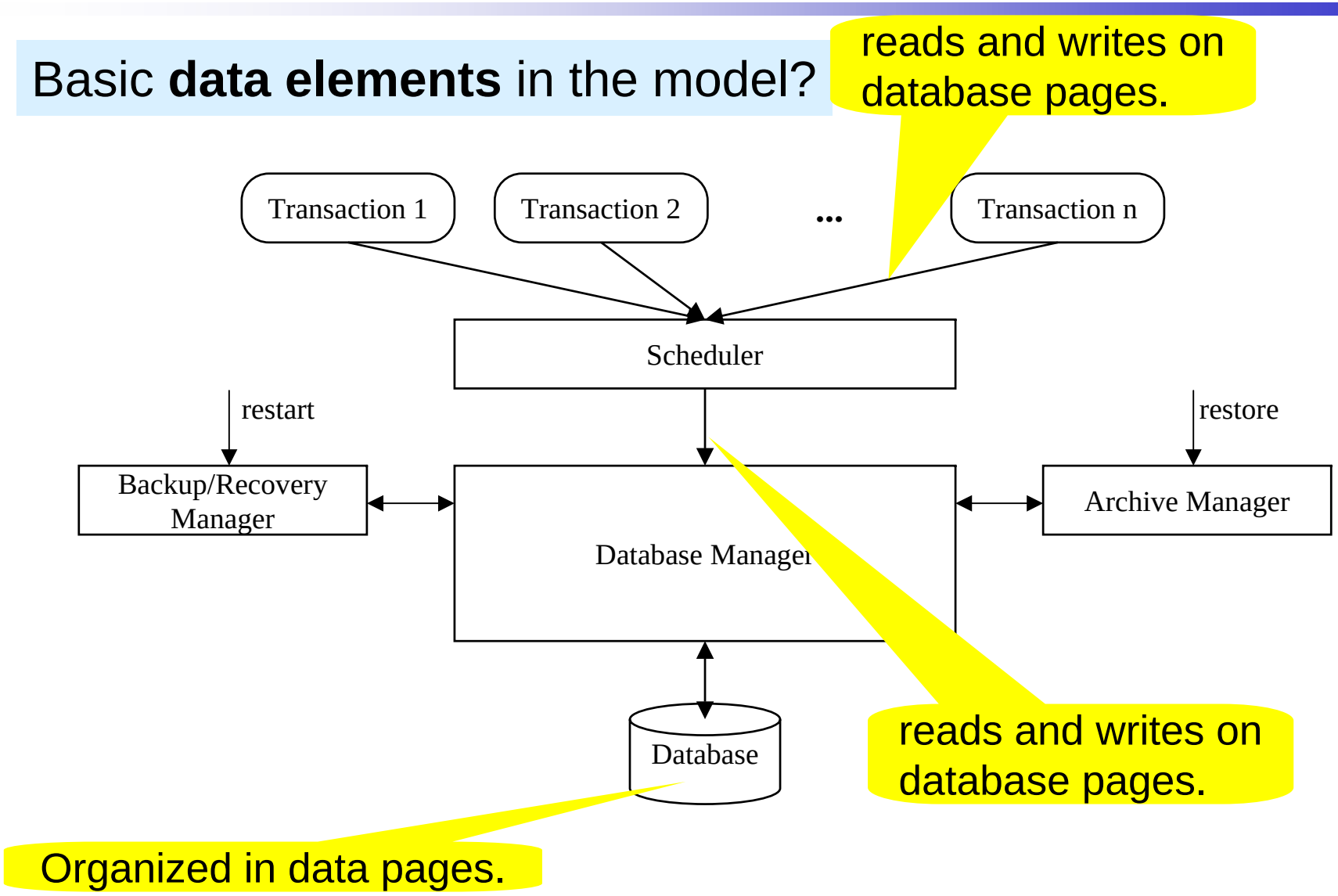
$t_2 = r(s) \rightarrow w(s)$
$r(v) \rightarrow w(v)$

"Semantics":

Interpretation of j^{th} step, p_j, of t:
If p_j=r(x), then interpretation is assignment v_j := x to local variable v_j
If p_j=w(x) then interpretation is assignment x := f_j (v_{j1}, ..., v_{jk}).
with unknown function f_j and j_1, ..., j_k denoting p_j's prior read steps.

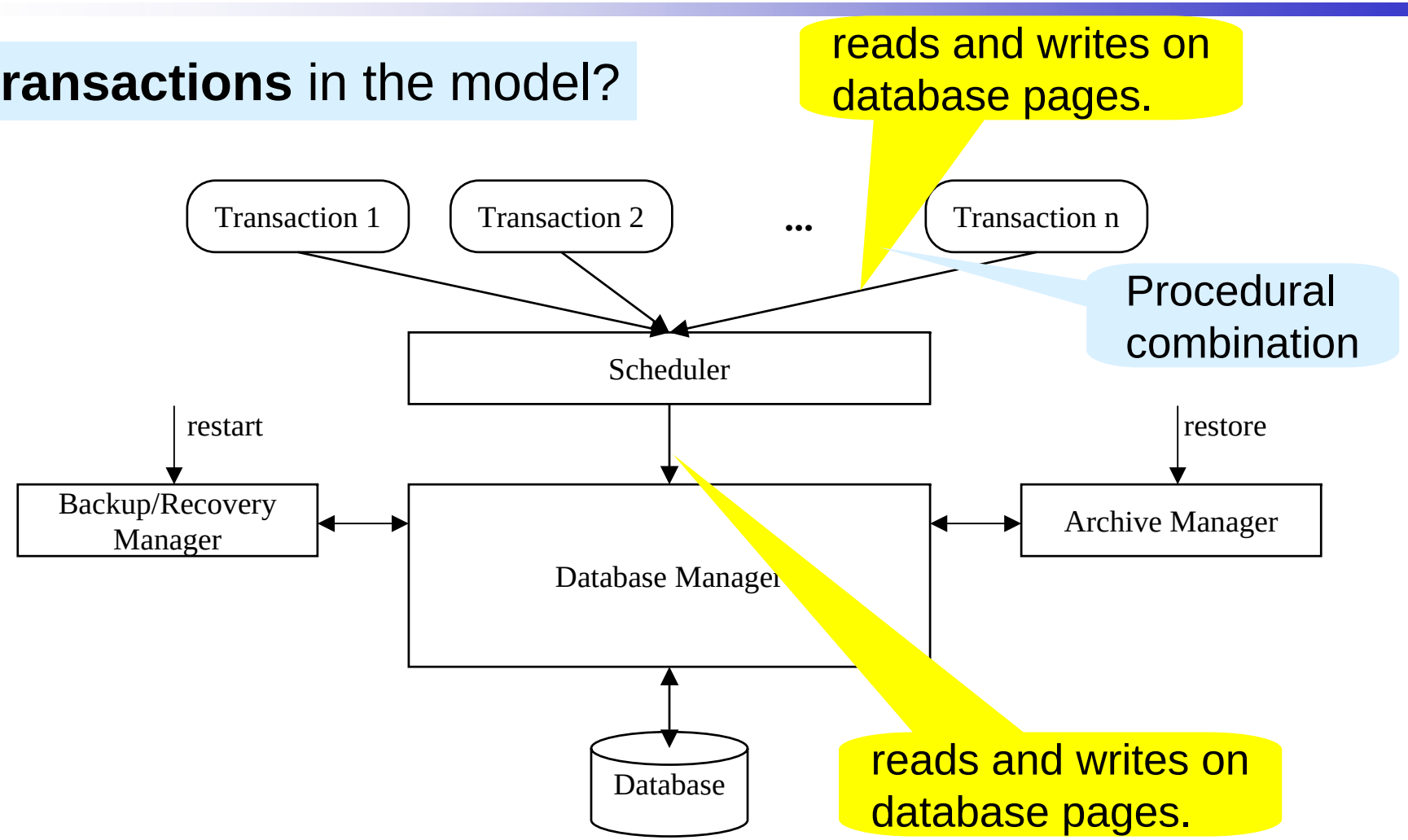
Page model

Basic **data elements** in the model?



Procedural model

Transactions in the model?



Procedural model

"Syntax":

Definition 2.1 (Read/Write Model Transaction):
A **transaction** t is a partial order of steps (actions) of the form r(x) or w(x), where x ∈ D (D database) and reads and writes applied to the same object are strictly ordered.
We write t = (op, <)
for transaction t with step set op and partial order <.

Examples: r(s)→w(s)
r(v)→w(v)

- For some operations their mutual order of execution does not affect the end result.
  - Definition covers the multiprocessor/parallel case.

Procedural model

A transaction brackets a number of operations

- `read` r
- `write` w

abbrev.

by

- `begin transaction` and b
- `end transaction` or c
- `abort` a

abbrev.

Successful completion of the transaction.
Results are made persistent (**commit**).

Unsuccessful termination of the transaction.
Results are discarded.

Transaction model

Definition 2.2 (Read/Write Model Transaction):

A **transaction** t_i is a partial order $(op_i, <_i)$, with op_i all operations in t_i and

1. $op_i \subseteq \{r_i(x), w_i(x) \mid x \text{ is data element}\} \cup \{a_i, c_i\}$
   $(op'_i = op_i \setminus \{a_i, c_i\})$
2. $\forall p \in op_i \; p = c_i \vee p = a_i \succ (\forall q \in op'_i \; q <_i p)$
3. $r_i(x), w_i(x) \in op_i \succ (r_i(x) <_i w_i(x) \vee w_i(x) <_i r_i(x))$
4. $a_i \in op_i \succ c_i \notin op_i$ and $c_i \in op_i \succ a_i \notin op_i$

Each operation occurs at most once

If *a* or *c* in t_i, then as the last operation

Reads and writes on the same element are ordered

If *a* or *c* in t_i, only one of them

Transaction model

Definition 2.2 (Read/Write Model Transaction):

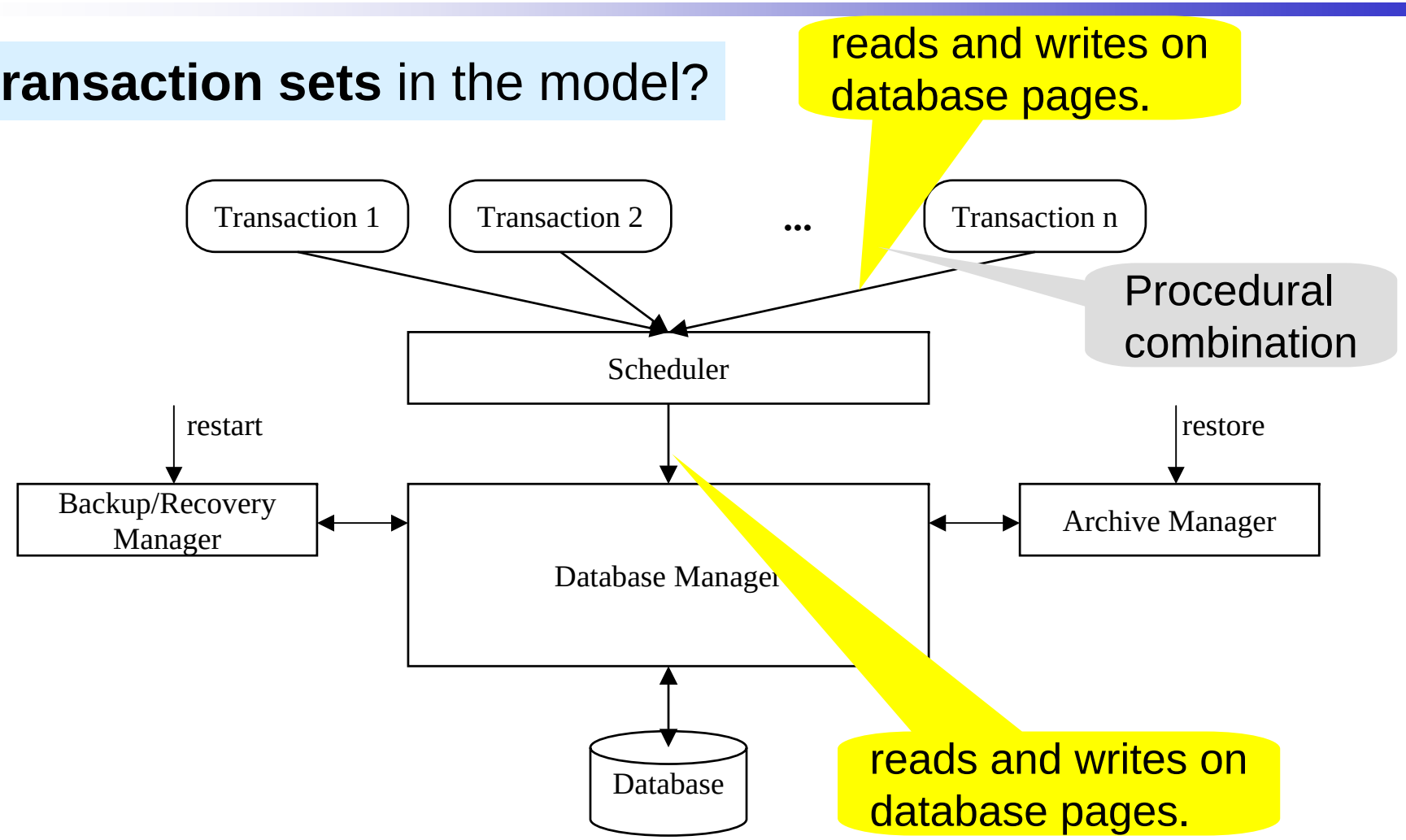
A **transaction** t_i is a partial order $(op_i, <_i)$, with op_i all operations in t_i and

1. $op_i \subseteq \{r_i(x), w_i(x) \mid x \text{ is data element}\} \cup \{a_i, c_i\}$
   $(op'_i = op_i \setminus \{a_i, c_i\})$
2. $\forall p \in op_i \; p = c_i \vee p = a_i \succ (\forall q \in op'_i \; q <_i p)$
3. $r_i(x), w_i(x) \in op'_i \succ (r_i(x) <_i w_i(x) \vee w_i(x) <_i r_i(x))$
4. $a_i \in op_i \succ c_i \notin op_i$ and $c_i \in op_i \succ a_i \notin op_i$

Often excluded due to standard implementation:
No read of an element after it was updated.

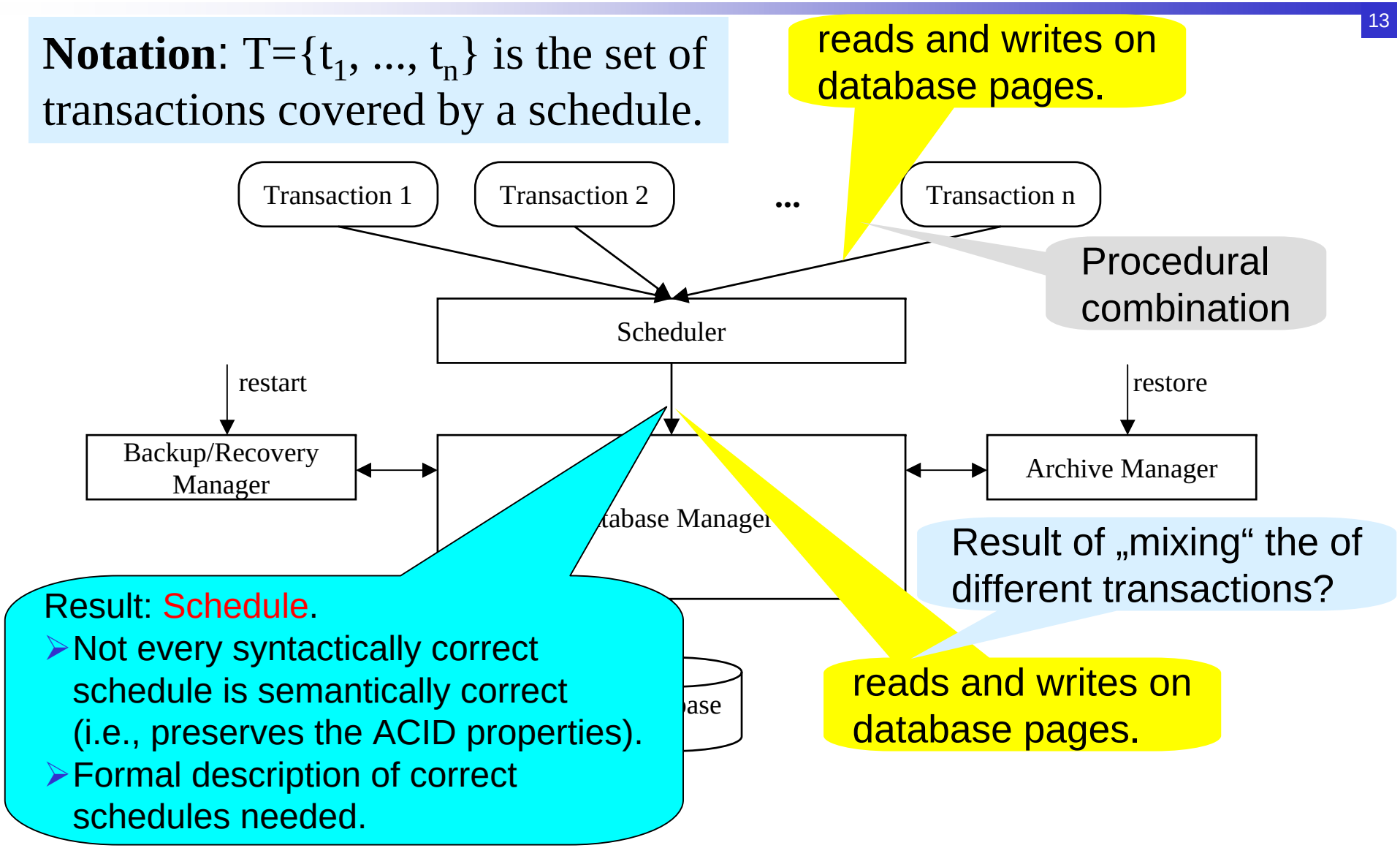
Transaction set model

Transaction sets in the model?



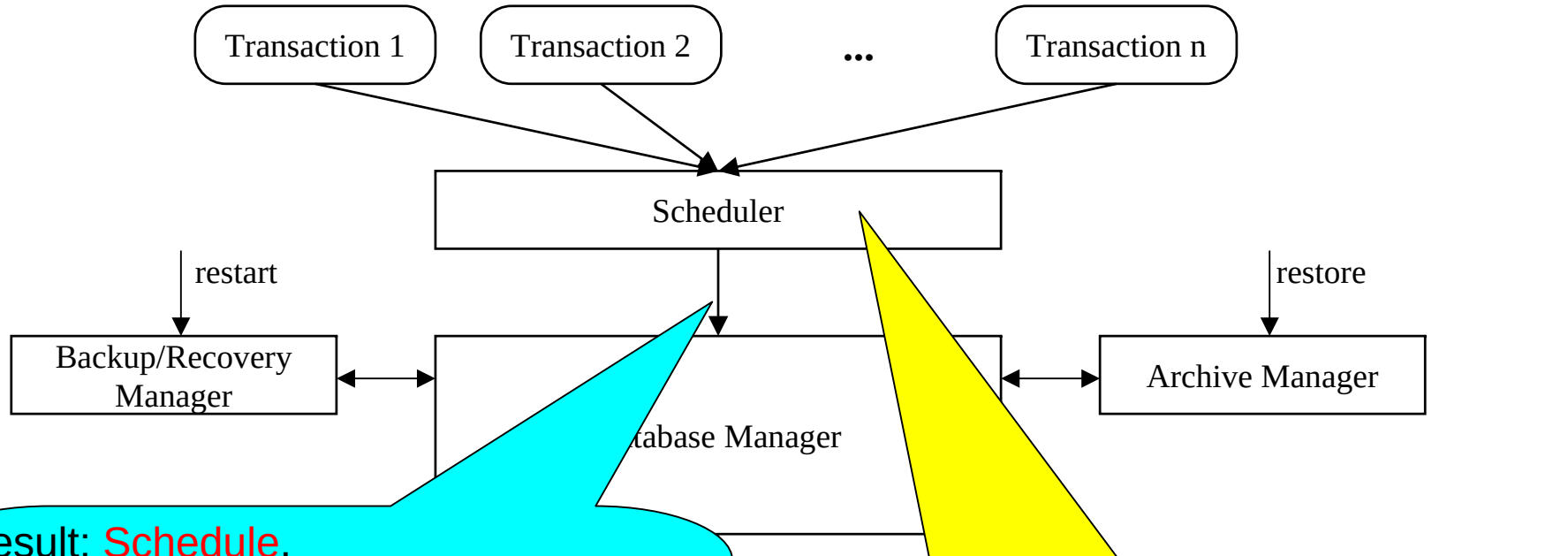
Transaction set model

Notation: $T=\{t_1, ..., t_n\}$ is the set of transactions covered by a schedule.



Transaction set model

Algorithms for the runtime generation of semantically correct schedules?



Result: Schedule.

- Not every syntactically correct schedule is semantically correct (i.e., preserves the ACID properties).
- Formal description of correct schedules needed.

- Application programs send basic operations.
- Algorithm sorts these dynamically into a correct schedule.

DBS

Universität Karlsruhe (TH)

Chapter 3

Isolation: Examples

Example 1

Miniworld

Inventory database of a wine dealer:

White wines	
article	*stock*
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller Thurgau	100

Sales transaction

1. Sales transaction with inventory update (R: Riesling)

step	
1	read(R)
2	update(R)
3	write(R)

Ignored in the R/W model: Local action

Transaction in the R/W model: $t = r(R)\ w(R)$

2. Two sales persons: *Miller* and *Smith*.
3. Database: page ≈ tuple.

Lost Update

time	Miller	Smith	db value	comment
1	read(R)		34	
2		read(R)	34	
3	update(R)		34	Miller sell[illegible]ox
4		update(R)	34	Smith [illegible]ells 2 boxes
5	write(R)		33	34-1
6		write(R)	32	34-2

Mr. Miller's update has been lost (**lost update**).

White wines	
article	*stock*
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller Thurgau	100

White wines	
article	*stock*
Gutedel	12
Riesling	32
Silvaner	2
Weißburgunder	11
Müller Thurgau	100

Schema: $r_1(R)\ r_2(R)\ w_1(R)\ w_2(R)$

Dirty Read (1)

time	Miller	Smith	db value	comment
1	read(R)		34	
2	update(R)		34	Miller sells 1 box
3	write(R)		33	34-1
4		read(R)	33	Smith reads the changed value
5		update(R)	33	Smith sells 2 boxes
6	abort		34	Miller cancels his sale
7		write(R)	31	33-2

Never became valid!

Cancelation ignored.

White wines	
article	*stock*
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller Thurgau	100

White wines	
article	*stock*
Gutedel	12
Riesling	31
Silvaner	2
Weißburgunder	11
Müller Thurgau	100

Schema: $r_1(R)\ w_1(R)\ r_2(R)\ a_1\ w_2(R)$

Dirty Read (2)

- **Dirty read**: Read a value after it was changed by a second transaction but before its validity could be guaranteed.
- Occurs whenever a transaction reads a value changed by a second transaction before that one committed.

Inconsistent Read (1)

Compute the sum total of wines on stock.

Correct a mixup in stock registration.

time	Miller	Smith
1	sum:=0	
2	read(Gutedel)	
3	sum:=12	
4	read(Riesling)	read(Müller-Th.)
5	sum:=12+34	Müller-Th.:= Müller-Th.-10
6	read(Silvaner)	write(Müller-Th.)
7	sum:= 46+2	read(Gutedel)
8	read(Weißb.)	Gutedel:=Gutedel+10
9	sum:= 48+11	write(Gutedel)
10	read(Müller-Th.)	
11	sum:= 59+90	

Inconsistent Read (2)

time	Miller	Smith
1	sum:=0	
2	read(Gutedel)	
3	sum:=12	
4	read(Riesling)	read(Müller-Th.)
5	sum:=12+34	Müller-Th.:= Müller-Th.-10
6	read(Silvaner)	write(Müller-Th.)
7	sum:= 46+2	read(Gutedel)
8	read(Weißb.)	Gutedel:=Gutedel+10
9	sum:= 48+11	write(Gutedel)
10	read(Müller-Th.)	
11	sum:= 59+90	

White wines	*persistent*
article	*stock*
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller Thurgau	100

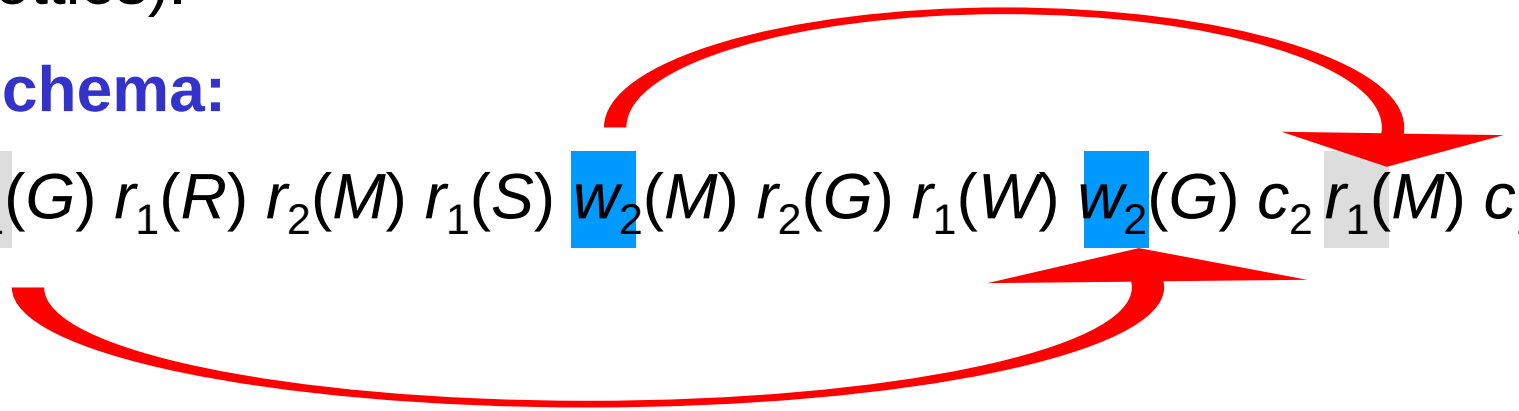
White wines	*transient*
article	*stock*
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller Thurgau	90

White wines	*persistent*
article	*stock*
Gutedel	22
Riesling	34
Silvaner	2
Weißburgunder	11
Müller Thurgau	90

Inconsistent Read (3)

- **inconsistent read**: Read of states that are valid at different times $\Rightarrow$ inconsistent view of the database.
- In the example: resulting count incorrect (missed 10 bottles).
- **Schema:**

$r_1(G)\ r_1(R)\ r_2(M)\ r_1(S)\ w_2(M)\ r_2(G)\ r_1(W)\ w_2(G)\ c_2\ r_1(M)\ c_1$



Threats to the global consistency

- Interaction of reads and writes :
  - Two concurrent read-only transactions have no mutual impact, since in the read/write model the only interactions possible between transactions are via database writes.
  - If a read transaction competes with a write transaction, the write transaction remains unaffected and hence the database consistent. On the other hand, the write transaction may have an effect on the read transaction.
  - Two concurrent write transactions may affect each other, and the database may become inconsistent.

Example 2

Scenario

- Relations:
  - TICKET (ticketNo, name) **T**
  - BOOKING (flightNo, ticketNo, seatCode, date) **B**
- Transactions:
  - t_1: Check the consistency of passenger list und bookings,
  - t_2: Rebook a passenger group,
  - t_3: Cancel a booking.
- Simplification:
  - Relations fit on a single page.

Transaction t_1

- Print number of tickets sold for 12 August 2000 plus the list of the associated passengers:

```
select   count (distinct ticketNo)
from     BOOKING                          read BOOKING
where    date = 12-AUG-00;

print number of tickets;

select   name
from     TICKET                           read TICKET
where    ticketNo in
         (select  ticketNo
          from    BOOKING                 BOOKING already read
          where   date = 12-AUG-00);

print passenger list;

commit;
```

r/w model for t_1 : $r_1(B)\ r_1(T)\ c_1$.

Transaction t_2

- Rebook passengers in row 19 from LH500 on 12 August 2000 to 11 August 2000 and mark ticket number:

```
update  TICKET
set     ticketNo = ticketNo + 100000
where   ticketNo in
            (select  ticketNo
             from    BOOKING
             where   date = 12-AUG-00 and flightNo = "LH500"
             and     (seatCode = "19D" or seatCode = "19E"
                      or seatCode = "19G" ));

update  BOOKING
set     date = 11-AUG-00, ticketNo = ticketNo + 100000
where   date = 12-AUG-00 and flightNo = "LH500"
and     (seatCode = "19D" or seatCode = "19E" or seatCode = "19G");
commit;
```

read TICKET

read BOOKING

write TICKET

BOOKING already read

write BOOKING

Note that the first write depends on the two reads.

r/w model for t_2 : $r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2$.

Transaction t_3

- Cancel the ticket with number 7216087338:

```
delete    from TICKET
where     ticketNo = 7216087338;

delete    from BOOKING
where     ticketNo = 7216087338;

commit;
```

read TICKET

write TICKET
read BOOKING

write BOOKING

r/w model for t_3 : $r_3(T)$ $w_3(T)$ $r_3(B)$ $w_3(B)$ c_3 .

Inconsistent Read (1)


```
select    count (distinct ticketNo)
from      BOOKING                    r1(B)
where     date = 12-AUG-00;

Print number of tickets;
```

```
update    TICKET                                        r2(T)
set       ticketNo = ticketNo + 100000
where     ticketNo in
          (select    ticketNo
          from       BOOKING                  r2(B)
          where      date = 12-AUG-00          and flightNo = "LH500"
          and        (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G" ));
                                                                  w2(T)
update    BOOKING
set       date = 11-AUG-00, ticketNo = ticketNo + 100000
where     date = 12-AUG-00          and flugNr = "LH500"
and       (seatCode = "19D" or seatCode = "19E" or seatCode = "19G");
                                                                        w2(B)
commit;
```

```
select    name
from      TICKET                r1(T)
where     ticketNo in
          (select    ticketNo
           from      BOOKING
           where     date = 12-AUG-00);

print passenger list;

commit;
```

S1: $r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

read-write interaction by interrupting the reader

Inconsistent Read (2)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

Inconsistent Read (3)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

Output t_1 : number of tickets = 3.

Inconsistent Read (4)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads BOOKING and selects.

Inconsistent Read (5)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads TICKET and selects.

Inconsistent Read (6)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 updates and writes TICKET.

Inconsistent Read (7)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 updates and writes BOOKING.

Inconsistent Read (8)

$r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

Output t_1 : empty passenger list.

No problem


```
update    TICKET                                          r2(T)
set       ticketNo = ticketNo + 100000
where     ticketNo in
          (select    ticketNo
          from       BOOKING                    r2(B)
          where      date = 12-AUG-00           and flightNo = "LH500"
          and        (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G" ));
```

S2: $r_2(B)\ r_2(T)\ r_1(B)\ r_1(T)\ c_1\ w_2(T)\ w_2(B)\ c_2$

```
          select     count (distinct ticketNo)
          from       BOOKING                              r1(B)
          where      date = 12-AUG-00;

          print number of tickets;

          select     name
          from       TICKET                    r1(T)
          where      ticketNo in
                     (select     ticketNo
                      from       BOOKING
                      where      date = 12-AUG-00);

          print passenger list;

          commit;
```

read-write interaction by interrupting the writer

No problem, because t_1 reads a database state that was not yet changed. Tough luck for t_1 that it is no longer valid shortly afterwards.

```
update    BOOKING                                         w2(T)
set       date = 11-AUG-00, ticketN
where     date = 12-AUG-00          and flightNo = "LH500"
and       (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G");
                                                                  w2(B)
commit;
```

Dirty Read (1)


```
update    TICKET                                          r2(T)
set       ticketNo = ticketNo + 100000
where     ticketNo in
          (select    ticketNo
          from       BOOKING                  r2(B)
          where      date = 12-AUG-00         and flightNo = "LH500"
          and        (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G" ));
                                                          w2(T)
```

S3: $r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ c_1\ w_2(B)\ c_2$

```
          select     count (distinct ticketNo)
          from       BOOKING                          r1(B)
          where      date = 12-AUG-00;

          print number of tickets;

          select     name
          from       TICKET          r1(T)
          where      ticketNo in
                     (select    ticketNo
                      from      BOOKING
                      where     date = 12-AUG-00);

          print passenger list;

          commit;

update    BOOKING
set       date = 11-AUG-00, ticketNo = ticketNo + 100000
where     date = 12-AUG-00          and flightNo = "LH500"
and       (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G");
                                                          w2(B)
commit;
```

read-write interaction by interrupting the writer

Effect on t_1 as in S1.

Dirty Read (2)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

Dirty Read (3)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads BOOKING and selects.

Dirty Read (4)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads TICKET and selects.

Dirty Read (5)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 updates and writes TICKET.

Dirty Read (6)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

Output t_1 : number of tickets = 3.

Dirty Read (7)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

Output t_1 : empty passenger list.

Dirty Read (8)

$r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ w_2(B)\ c_2\ c_1$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 updates and writes BOOKING.

Dirty Read (9)

- „dirty read“ leaves open whether the transient state is consistent or not.
- Take schedule

  S4: $r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ r_1(B)\ r_1(T)\ c_2\ c_1$.
- $r_1(B)\ r_1(T)$ has a consistent output despite dirty read.
- Still dangerous! Take abort of T_2:

  S4: $r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ r_1(B)\ r_1(T)\ a_2\ c_1$.
- Because t_2 leaves no trace t_1 reads a state that never existed!

Inconsistent + Dirty Read (1)


```
delete      from TICKET                                   r3(T)
where       ticketNo = 7216087338;
                                                          w3(T)
```

```
update      TICKET                          r2(T)
set         ticketNo = ticketNo + 100000
where       ticketNo in
            (select     ticketNo
            from        BOOKING                 r2(B)
            where       date = 12-AUG-00        and flightNo = "LH500"
            and         (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G" ));
                                                                              w2(T)
update      BOOKING
set         date = 11-AUG-00, ticketNo = ticketNo + 100000
where       date = 12-AUG-00        and flightNo = "LH500"
and         (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G");
                                                                              w2(B)
commit;
```

```
delete      from BOOKING                                  r3(B)
where       ticketNo = 7216087338;
                                                          w3(B)
commit;
```

writer-writer interaction by interrupting one of the writers

S5: $r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

Inconsistent + Dirty Read (2)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

Inconsistent + Dirty Read (3)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_3 reads TICKET and selects.

Inconsistent + Dirty Read (4)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_3 writes TICKET.

Inconsistent + Dirty Read (5)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads BOOKING and selects.

Inconsistent + Dirty Read (6)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads TICKET and selects.

Inconsistent + Dirty Read (7)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 writes TICKET.

Inconsistent + Dirty Read (8)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 writes BOOKING .

Inconsistent + Dirty Read (9)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_3 reads BOOKING .

Inconsistent + Dirty Read (10)

$r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_3 writes BOOKING .

Lost Update (1)


```
update    TICKET                                    r2(T)
set       ticketNo = ticketNo + 100000
where     ticketNo in
          (select    ticketNo
          from       BOOKING          r2(B)
          where      date = 12-AUG-00          and flightNo = "LH500"
          and        (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G" ));

                delete     from TICKET                    r3(T)
                where      ticketNo = 7216087338;         w3(T)

                delete     from BOOKING                   r3(B)
                where      ticketNo = 7216087338;         w3(B)

                commit;

                                                          w2(T)
update    BOOKING
set       date = 11-AUG-00, ticketNo = ticketNo + 100000
where     date = 12-AUG-00          and flightNo = "LH500"
and       (seatCode = "19D" or seatCode = "19E„ or seatCode = "19G");
                                                                    w2(B)
commit;
```

S6: $r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

writer-writer interaction by interrupting one of the writers

Lost Update (2)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

Lost Update (3)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads BOOKING and selects.

Lost Update (4)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_2 reads TICKET and selects.

Lost Update (5)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216087338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_3 reads TICKET and selects.

Lost Update (6)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_3 writes TICKET.

Lost Update (7)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216087338	19D	12-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_3 reads BOOKING and selects.

Lost Update (8)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216083970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216080817	Weinand_Mr_C

t_3 writes BOOKING.

Lost Update (9)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216083970	19G	12-AUG-00
LH500	7216080817	19E	12-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 writes TICKET.

Lost Update (10)

$r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$

flightNo	ticketNo	seatCode	date
LH711	7216083495	02B	26-AUG-00
LH3724	7216084316	08A	29-SEP-00
LH3651	7216084316	14F	03-OCT-00
LH408	7216088131	04D	04-SEP-00
LH403	7216088131	05D	08-SEP-00
LH208	7216088131	07C	09-SEP-00
LH2419	7216083969	02E	01-SEP-00
LH4080	7216084728	10K	07-AUG-00
LH4171	7216084728	07A	11-AUG-00
LH191	7216084728	01K	11-AUG-00
LH208	7216084069	05D	01-AUG-00
LH3724	7216088132	07E	14-AUG-00
LH458	7216080815	81K	03-SEP-00
LH710	7216082757	34D	10-SEP-00
LH400	7216084317	05G	21-JUL-00
LH401	7216084317	05D	05-AUG-00
LH500	7216187338	19D	11-AUG-00
LH500	7216183970	19G	11-AUG-00
LH500	7216180817	19E	11-AUG-00
LH778	7216083911	83K	05-AUG-00
LH6390	7216083911	82A	06-AUG-00

ticketNo	name
7216187338	Kuhn_Mrs_E
7216084065	Pulkowski_Mr_S
7216082757	Witte_Mr_R
7216084316	Krakowski_Mrs_P
7216084317	Posselt_Mr_D
7216083495	Gimbel_Mr_M
7216083971	Muelle_Mrs_J
7216183970	Bender_Mr_P
7216080815	Lockemann_Mr_P
7216080816	Simpson_Mr_B
7216180817	Weinand_Mr_C

t_2 writes BOOKING.

Lost Update (11)

- Note: Neither t_2 nor t_3 include „inconsistent read“ or „dirty read“. „lost update“ is an independent phenomenon!

Example 3

OLTP Example: Debit/Credit


```
void main ( ) {
  EXEC SQL BEGIN DECLARE SECTION
    int b /*balance*/, a /*accountid*/, amount;
  EXEC SQL END DECLARE SECTION;
  /* read user input */
  scanf ("%d %d", &a, &amount);
  /* read account balance */
  EXEC SQL Select Balance into :b From Account
    Where Account_Id = :a;
/* add amount (positive for debit, negative for credit) */
  b = b + amount;
/* write account balance back into database */
  EXEC SQL Update Account
    Set Balance = :b Where Account_Id = :a;
  EXEC SQL Commit Work;
}
```

OLTP Example: Concurrent Executions

P1	Time	P2
Select Balance Into :b_1 **From Account** **Where Account_Id = :a**	**1**	
/* b_1=0, a.Balance=100, b_2=0 */		
		Select Balance Into :b_2 **From Account** **Where Account_Id = :a**
	2	
/* b_1=100, a.Balance=100, b_2=100 */		
b1 = b1-50	**3**	
/* b_1=50, a.Balance=100, b_2=100 */		
	4	**b_2 = b_2 +100**
/* b_1=50, a.Balance=100, b_2=200 */		
Update Account **Set Balance = :b_1** **Where Account_Id = :a**	**5**	
/* b_1=50, a.Balance=50, b_2=200 */		
		Update Account **Set Balance = :b_2** **Where Account_Id = :a**
	6	
/* b_1=50, a.Balance=200, b_2=200 */		

DBS

Universität Karlsruhe (TH)

Chapter 4

Isolation: Correctness in the read/write model

Agenda

- Concurrent transactions
- Histories and schedules
- Correct histories and schedules

Concurrent transactions

Transaction model revisited (1)

Definition 4.1 (Read/Write Model Transaction):

A **transaction** t_i is a partial order $(op_i, <_i)$, with op_i all operations in t_i and

1. $op_i \subseteq \{r_i(x), w_i(x) \mid x \text{ is data element}\} \cup \{a_i, c_i\}$
   $(op'_i = op_i \setminus \{a_i, c_i\})$
2. $\forall p \in op_i \; p = c_i \vee p = a_i \succ (\forall q \in op'_i \; q <_i p)$
3. $r_i(x), w_i(x) \in op'_i \succ (r_i(x) <_i w_i(x) \vee w_i(x) <_i r_i(x))$
4. $a_i \in op_i \succ c_i \notin op_i$ and $c_i \in op_i \succ a_i \notin op_i$

Transaction model revisited (2)

Definition 4.2

- **Sequential transaction**: $(op_i, <_i)$ is total order
  - Sufficient for the single-processor case.
- Transaction t_i is
  - **aborted**, if $a_i \in t_i$
  - **committed**, if $c_i \in t_i$
  - **completed**, if aborted or committed.

Two steps to concurrency

Step 1: **Histories and schedules**

- Find a formalism for describing the interleaved and concurrent execution of a set of transactions.

  Required from the formalism:

  - Include the formal characteristics of transactions.
  - Operations that are in conflict can only be executed in sequence.

Step 2: **Serializability**

- Ultimately we wish to know how to arrange the operations that are in conflict such that we obtain a correct ordering of the operations.
  - What definition of correctness?

Histories and schedules

Histories

Definition 4.3 (Histories):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **history** for T is a pair $h=(op(h),<_h)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \leq i \leq n$: $c_i \in op(h) \prec\succ a_i \notin op(h)$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \leq i \leq n$, and all $p \in op'_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all $p, q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item:
$p <_h q$ or $q <_h p$

Histories

Definition 4.3 (Histories):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **history** for T is a pair $h=(op(h),<_h)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \leq i \leq n$: $c_i \in op(h) \prec\succ a_i \notin op(h)$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \leq i \leq n$, and all $p \in op'_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all $p, q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item:
$p <_h q$ or $q <_h p$

all operations from all transactions are included

Histories

Definition 4.3 (Histories):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **history** for T is a pair $h=(op(h),<_h)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \leq i \leq n$: $c_i \in op(h) \prec\succ a_i \notin op(h)$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \leq i \leq n$, and all $p \in op'_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all $p, q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item:
$p <_h q$ or $q <_h p$

each transaction includes **exactly one** completion operation

Histories

Definition 4.3 (Histories):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **history** for T is a pair $h=(op(h),<_h)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \leq i \leq n$: $c_i \in op(h) \prec\succ a_i \notin op(h)$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \leq i \leq n$, and all $p \in op'_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all $p, q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item:

$p <_h q$ or $q <_h p$

the ordering within each transaction is preserved

Histories

Definition 4.3 (Histories):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **history** for T is a pair $h=(op(h),<_h)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \leq i \leq n$: $c_i \in op(h) \prec\succ a_i \notin op(h)$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \leq i \leq n$, and all $p \in op'_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all p, $q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item:

$p <_h q$ or $q <_h p$

the completion operation is the final operation in each transaction

Histories

Definition 4.3 (Histories):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **history** for T is a pair $h=(op(h),<_h)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \leq i \leq n$: $c_i \in op(h) \prec\succ a_i \notin op(h)$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \leq i \leq n$, and all $p \in op'_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all $p, q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item: $p <_h q$ or $q <_h p$

operations in conflict are strictly ordered

Histories

Definition 4.4 (Serial history):
A history h is **serial** if for any two transactions t_i and t_j in h, where $i \neq j$, all operations from t_i are ordered in h before all operations from t_j or vice versa.

Definition 4.5 (Totally ordered history):
A history h is **totally ordered** if in $h=(op(h),<_h)$, $<_h$ is a total order, i.e., all operations from op(h) are ordered in sequence.

Schedules

Definition 4.6 (Schedules):

Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A **schedule** for T is a pair $s=(op(s),<_s)$ s.t.

(a) $op(h) = \cup_{i=1..n} op_i$

(b) for all i, $1 \le i \le n$: ~~$c_i \in op(h) \Leftrightarrow a_i \notin op(h)$~~ $a_i \in op_i \succ c_i \notin op_i \wedge c_i \in op_i \succ a_i \notin op_i$

(c) $\cup_{i=1..n} <_i \subseteq <_h$

(d) for all i, $1 \le i \le n$, and all $p \in op_i$: $p <_h c_i$ or $p <_h a_i$

(e) for all $p, q \in op(h) \setminus \cup_{i=1..n}\{a_i, c_i\})$ s.t. at least one of them is a write and both access the same data item: $p <_h q$ or $q <_h p$

$c_i \in op(h) \succ$

$a_i \in op(h) \succ$

Schedules and Histories

- A schedule is a prefix of a history.
- A history is a projection of a schedule on completed transactions.
- A *committed projection* of a schedule is a projection on committed transactions.

History: example

Example 4.7: Suppose $T = \{t_1, t_3, t_4\}$ with

$t_1 = r_1(x) \rightarrow w_1(x)$

$t_3 = r_3(x) \rightarrow w_3(y) \rightarrow w_3(x)$

$t_4 = r_4(y) \rightarrow w_4(x) \rightarrow w_4(y) \rightarrow w_4(z)$

One history h_1 over T is:

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

$$\uparrow \qquad \uparrow$$

$$h_1 = \quad r_4(y) \rightarrow w_4(x) \rightarrow w_4(y) \rightarrow w_4(z) \rightarrow c_4$$

$$\uparrow$$

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

Schedule: example

Schedule over T as a prefix of h_1 :

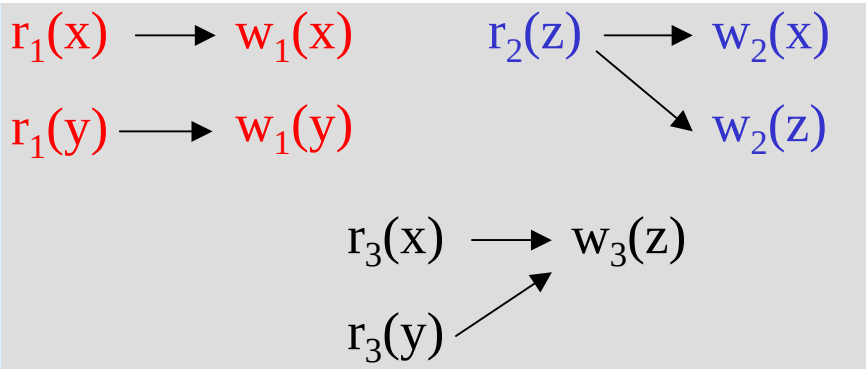
$$r_3(x) \rightarrow w_3(y) \rightarrow w_3(x) \rightarrow c_3$$

$$s_1 = \quad r_4(y) \rightarrow w_4(x) \rightarrow w_4(y) \rightarrow w_4(z) \rightarrow c_4$$

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

History and schedule: example

Example 4.8:



totally ordered history:

h1= $r_1(x)$ $r_2(z)$ $r_3(x)$ $w_2(x)$ $w_1(x)$ $r_3(y)$ $r_1(y)$ $w_1(y)$ $w_2(z)$ $w_3(z)$ c_1 c_2 a_3

totally ordered schedules (history prefixes):

s2 = $r_1(x)$ $r_2(z)$ $r_3(x)$ $w_2(x)$ $w_1(x)$

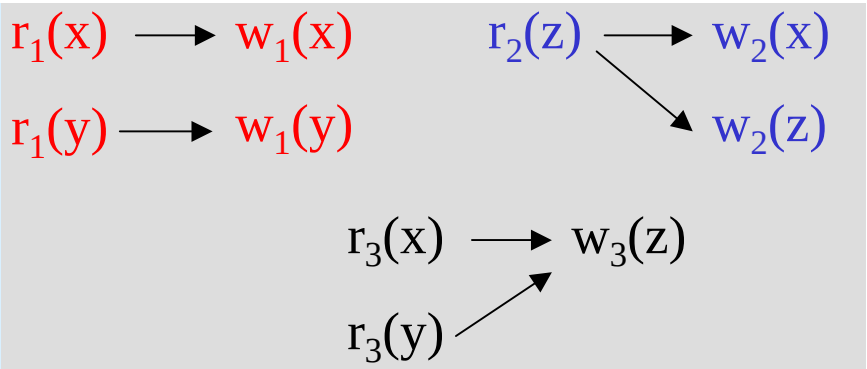
s3 = $r_1(x)$ $r_2(z)$ $r_3(x)$ $w_2(x)$ $w_1(x)$ $r_3(y)$ $r_1(y)$ $w_1(y)$ $w_2(z)$ $w_3(z)$ c_1

serial history:

h4= $r_1(x)$ $w_1(x)$ $r_1(y)$ $w_1(y)$ c_1 $r_3(x)$ $r_3(y)$ $w_3(z)$ a_3 $r_2(z)$ $w_2(x)$ $w_2(z)$ c_2

History and schedule: example

Example 4.8:



totally ordered schedule:

s5= $r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)\ c_1\ r_3(x)\ r_3(y)\ w_3(z)\ a_3\ r_2(z)\ w_2(x)\ w_2(z)$

history as a projected schedule:

h6 = $r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)\ c_1\ r_3(x)\ r_3(y)\ w_3(z)\ a_3$

history as a committed projection:

h7= $r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)\ c_1$

Transaction sets of a schedule (1)

Definition 4. 9 (Transaction sets of a schedule)

- Transactions occurring partially or completely in *s*:
  $trans(s) := \{t_i \mid s \text{ contains operations from } t_i\}$
- Transactions aborted in *s*:
  $abort(s) := \{t_i \mid a_i \in s\}$
- Transactions committed in *s*:
  $commit(s) := \{t_i \mid c_i \in s\}$
- Transactions still active in *s*:
  $active(s) := trans(s) \setminus (commit(s) \cup abort(s))$
- Transactions completed in *s*:
  $complete(s) := commit(s) \cup abort(s)$
- Operations in *s*:
  $op(s) := \{op \mid op \in s\}$

Transaction sets of a schedule (2)

Example 4.10

s1= $r_1(x)$ $r_2(z)$ $r_3(x)$ $w_2(x)$ $w_1(x)$ $r_3(y)$ $r_1(y)$ $w_1(y)$ $w_2(z)$ $w_3(z)$ c_1 c_2 a_3

trans(s1) = $\{t_1, t_2, t_3\}$

commit(s1) = $\{t_1, t_2\}$

abort(s1) =$\{t_3\}$

active(s1) = {}

Transaction sets of histories and schedules

Remarks 4.11:

For each schedule *s*:

- commit(*s*) ∩ abort(*s*) = commit(*s*) ∩ active(*s*)
  = abort(*s*) ∩ active(*s*) = ∅
- trans(*s*) = commit(*s*) ∪ abort(*s*) ∪ active(*s*)

For each history *h*:

- trans(*h*) = commit(*h*) ∪ abort(*h*)
- active(*h*) = ∅

Correct

histories and schedules

Correct schedules (1)

Remember:

- Include the formal characteristics of transactions.
- Operations that are in conflict can only be executed in sequence.

Definitions sufficient?

- Given a set of transactions $t_1,\ldots,t_n$, there are many syntactically correct schedules.
- Which of these guarantee global consistency (semantic correctness)?
  - What do we mean by global consistency?

Correct schedules (2)

Find: Correctness criterion $\sigma : S \rightarrow \{0, 1\}$ for schedules $s \in S$ where

- *correct(S)* := $\{ s \mid \sigma(s) = 1\} \neq \emptyset$,
- *correct(S)* large
- $s \in$ *correct(S)* is efficiently decidable.

To be useful the criterion must admit some schedules.

To be useful the criterion should admit a large number of schedules. (This would give us a choice among alternatives, and we could choose one with a high degree of parallelism.)

Choose the criterion s.t. an algorithm can quickly decide on the correctness of a schedule.

conflict!

And: The criterion should make (semantic) sense!

Correct schedules (3)

correct(*S*) ≠ ∅:

- That‘s easy: In serial schedules transactions are isolated. ⇒ Serial schedules are correct.
- However: Serial execution is extremely inefficient. ⇒ Find schedules that allow for something better than serial execution, but are in some (which?) sense *equivalent* to serial schedules. We call such schedules *serializable*.

i.e., are semantically correct

correct(*S*) large:

- Our hope: There are more serializable schedules than just serial schedules.
- Find a constructive definition of equivalence with a large set of serializable schedules.

correct(*S*) efficiently decidable:

- The definition should make equivalence efficiently decidable.

Correct schedules (4)

Schedules evolve

Histories have a known outcome

$\Rightarrow$

Define correctness on histories!

$\Rightarrow$

Then ensure somehow that schedules result in correct histories!

Final-state equivalence

Definition 4.12

- **Final-state equivalence**: Two histories are final-state equivalent if they have the same operations and result in the same final state for any given initial state.
- **Final-state serializability**: A history $h \in H$ is final-state serializable if there exists a final-state equivalent serial history.
  - Semantics? $\Rightarrow$ Equivalence considers only the final outcome, not the intermediate states and not whether each individual transaction behaves the same in the two histories.
  - correct(H) large: Presumably, because of the weak requirement: Much of the past can safely be ignored.
  - correct(H) efficiently decidable: ?????

Last write in a schedule

Definition 4.13 (Last write)

Let s be a schedule and x a data element. **Last write** of x in s is operation $w_i(x) \in s$ where

- $a_i \notin s$
- $\forall w_j(x) \in s \quad i \neq j \succ w_j(x) <_s w_i(x) \vee a_j \in s$

We write $w_i(x) = FIN_s(x)$

Example 4.14

$s_{12} = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(x)\ c_1\ w_3(x)\ w_3(y)\ c_3\ w_4(x)\ a_4$

Then $w_3(x) = FIN_{s_{12}}(x)$

$w_3(y) = FIN_{s_{12}}(y)$

Final-state equivalence

Definition 4.15 (final-state equivalence)

Let s and s' be schedules. s and s' are **final-state equivalent** ($s \equiv_F s'$) $\prec\!\succ$

- $op(s) = op(s')$
- $\forall x \in D\ FIN_s(x) = FIN_{s'}(x)$ (D database)

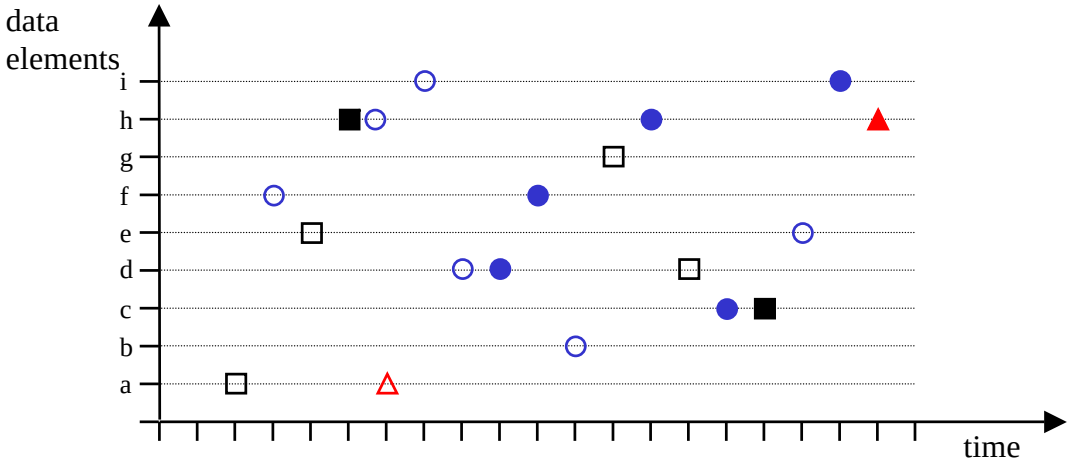
Two schedules – and by extension, two histories – are final-state equivalent iff both include the same operations and result in the same final state.

Conceptual test for serializability

- Existence of a final-state equivalent serial history implies: For each transaction its operations can commute to a single point in time (equivalence time) without changing *FIN*. The order of equivalence times determines the serial order.

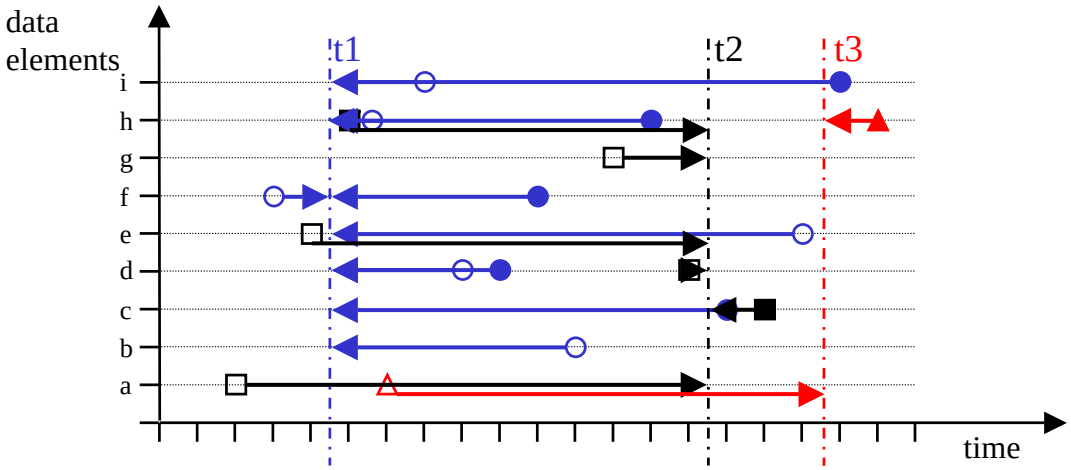
Conceptual test for serializability

- Take history:
  $r_2(a)\ r_1(f)\ r_2(e)\ w_2(h)\ r_1(h)\ r_3(a)\ r_1(i)\ r_1(d)\ w_1(d)\ w_1(f)\ r_1(b)$
  $r_2(g)\ w_1(h)\ r_2(d)\ w_1(c)\ w_2(c)\ r_1(e)\ w_1(i)\ c_1\ w_3(h)\ c_2\ c_3$.



Conceptual test for serializability

- Example:



Legend:

- Read by t1
- Write by t1
- Read by t2
- Write by t2
- Read by t3
- Write by t3

View equivalence

Definition 4.16

- **View equivalence**: Two histories are view equivalent if they have the same operations, if each transaction behaves the same in both histories, i.e., its read and write operations have the same effect, and they result in the same final state for any given initial state.
- **View serializability**: A history $h \in H$ is view serializable if there exists a view equivalent serial history.
  - Natural semantics!
  - correct(H) large: Because of the stronger requirement smaller than for final-state serializability.
  - correct(H) efficiently decidable: ?????

Read and write in schedules (1)

Definition 4.17 (reads from)

Let t_i und t_j be transactions and s a schedule where $t_i, t_j \in s$.
t_j **reads** *x* **from** t_i if

- $\exists x\ w_i(x) <_s r_j(x)$
- $a_i \not<_s r_j(x)$
- $\forall k \neq i,j\ \ w_i(x) <_s w_k(x) <_s r_j(x) \Rightarrow a_k <_s r_j(x)$

A transaction t_j **reads from** t_i if there exists a data element x s.t. t_j reads x from t_i.

Notation: $t_j \rhd_s(x)\ t_i$ and $t_j \rhd_s t_i$, respectively.

Relation (*RF*: „reads from“)

$RF(s) := \{(t_i,x,t_j) \mid t_j \rhd_s(x)\ t_i\}$

Read and write in schedules (2)

Example 4.18

Consider

$s_7 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$

Then:

$t_2 \rhd_{s7}(y)\ t_1$

$t_2 \rhd_{s7}\ t_1$

View equivalence

Definition 4.19 (View equivalence)

Two schedules s und s' are **view equivalent** ($s \equiv_V s'$): $\prec\succ$

- $op(s) = op(s')$
- $RF(s) = RF(s')$
- $\forall x\ FIN_s(x) = FIN_{s'}(x)$

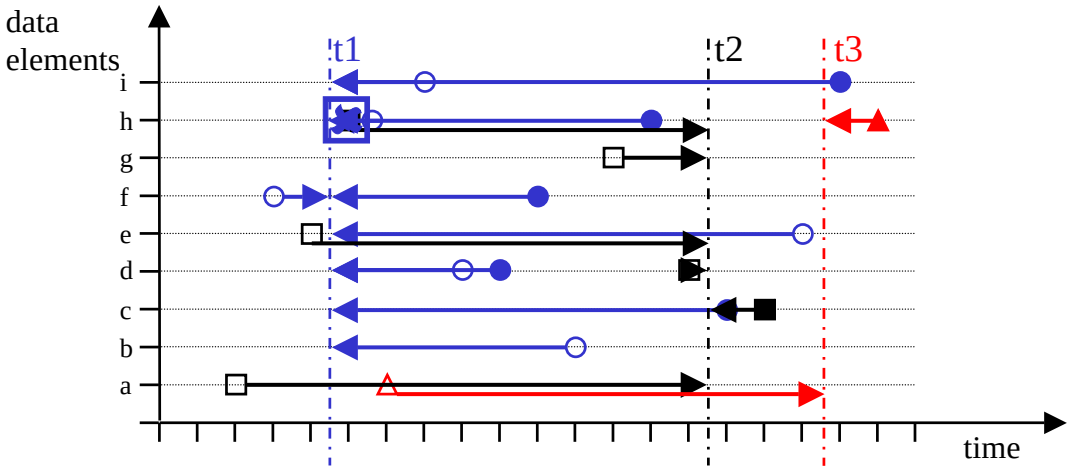
Two schedules – and by extension, two histories – are view equivalent iff both include the same operations, read the same states and result in the same final state.

Conceptual test for serializability

- Existence of a view equivalent serial history implies: For each transaction its operations can commute to a single point in time (equivalence time) without changing *RF* and *FIN*. The order of equivalence times determines the serial order.

Conceptual test for serializability

- Example:



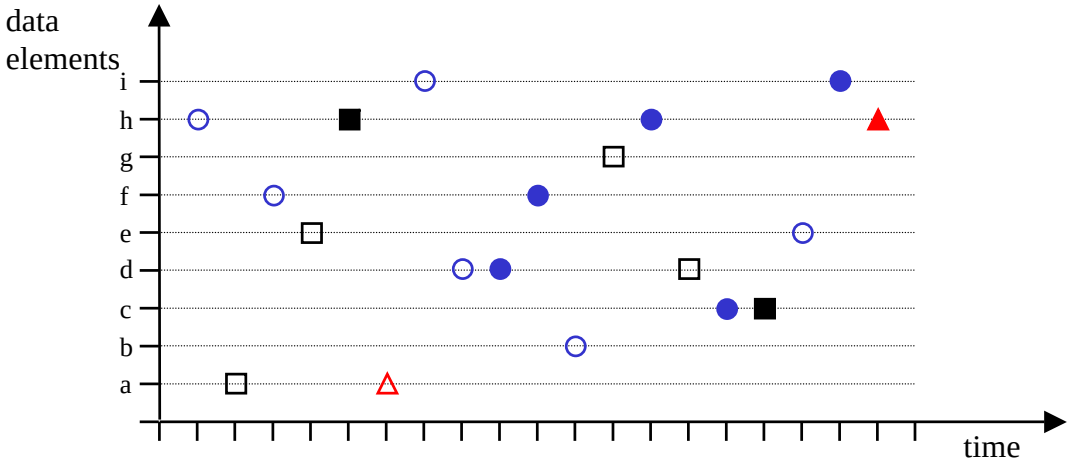
Legend:

- Read by t1
- Write by t1
- Read by t2
- Write by t2
- Read by t3
- Write by t3

Conceptual test for serializability

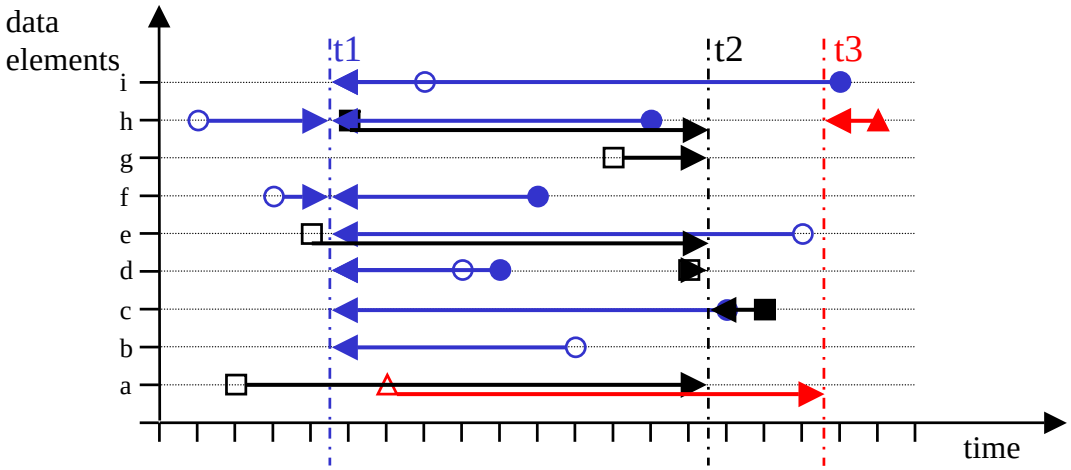
- Take history:

$r_2(a)\ r_1(f)\ r_2(e)\ w_2(h)\ r_1(h)\ r_3(a)\ r_1(i)\ r_1(d)\ w_1(d)\ w_1(f)\ r_1(b)$
$r_2(g)\ w_1(h)\ r_2(d)\ w_1(c)\ w_2(c)\ r_1(e)\ w_1(i)\ c_1\ w_3(h)\ c_2\ c_3.$



Conceptual test for serializability

- Example:



Legend:

- Read by t1
- Write by t1
- Read by t2
- Write by t2
- Read by t3
- Write by t3

Conflicts (1)

Definition 4.20 (in conflict)

Two operations p and q in the same or different transactions are **in conflict** (**do not commute**) if both access the same data element and at least one of them is a write operation.

Remarks

- Notation: $p \not\parallel q$.
- *in conflict* is symmetric.

Conflicts (2)

time	Miller	Smith	db value	comment
1	read(R)		34	
2	update(R)		34	Miller sells 1 box
3	write(R)		33	34-1
4		read(R)	33	Smith reads the changed value
5		update(R)	33	Smith sells 2 boxes
6	abort		34	Miller cancels his sale
7		write(R)	31	33-2

$$w_M(R) \not\| r_S(R)$$

$$w_M(R) \not\| w_S(R)$$

but

$$r_M(R) \| r_S(R)$$

Conflict equivalence

Definition 4.21

- **Conflict equivalence**: Two histories are conflict equivalent if they have the same operations, and operations that are in conflict are ordered the same in both histories.
- **Conflict serializability**: A history $h \in H$ is conflict serializable if there exists a conflict equivalent serial history h': $\exists h'$ serial: $h \equiv_C h'$

Conflict relation

Definition 4.22 (conflict relation):

Let s be a schedule. Then the conflict relation $conf(s)$ is

$$conf(s) = \{ (p_i,q_j) \mid p_i \not\parallel q_j,\ p_i <_s q_j,\ a_i,a_j \notin s \}$$

conf describes which operations of the transactions in a schedule are in conflict.

Conflict equivalence

Definition 4.23 (Conflict equivalence)

Two schedules s und s' are **conflict equivalent** ($s \equiv_C s'$) $\prec\succ$

- $op(s) = op(s')$
- $conf(s) = conf(s')$

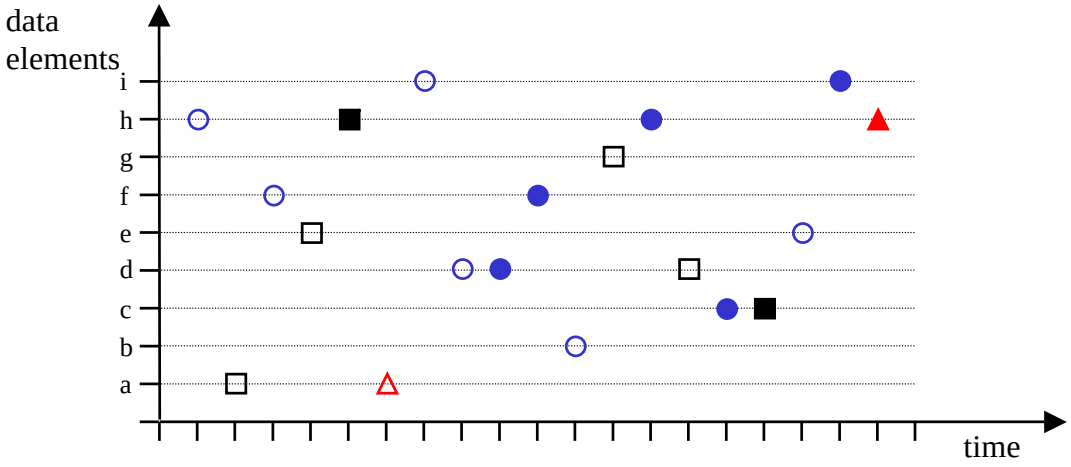
Two schedules – and by extension, two histories – are conflict equivalent iff both include the same operations, and identically order their conflicting operations.

Conceptual test for serializability

- Existence of a conflict equivalent serial history implies: For each transaction its operations can commute to a single point in time (equivalence time) without exchanging operations that are in conflict. The order of equivalence times determines the serial order.

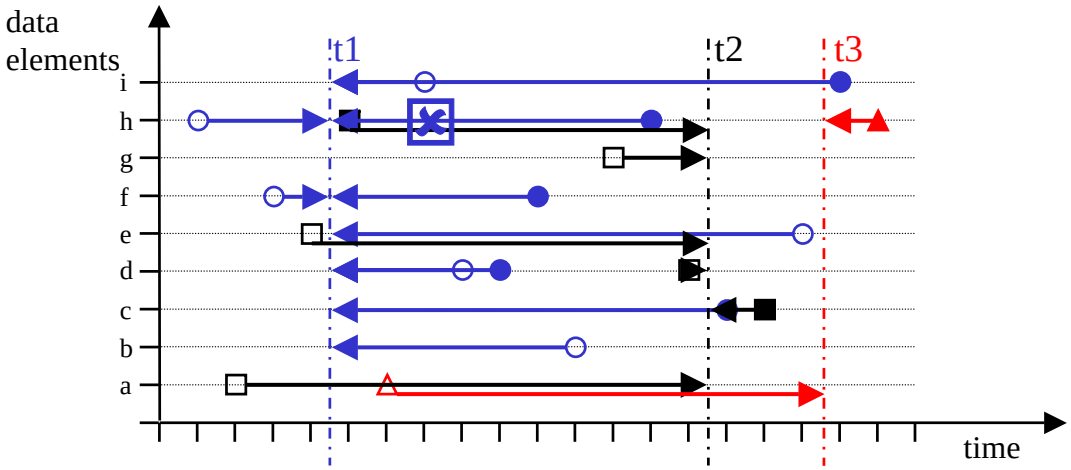
Conceptual test for serializability

- Take history:
  $r_1(h)\ r_2(a)\ r_1(f)\ r_2(e)\ w_2(h)\ r_3(a)\ r_1(i)\ r_1(d)\ w_1(d)\ w_1(f)\ r_1(b)$
  $r_2(g)\ w_1(h)\ r_2(d)\ w_1(c)\ w_2(c)\ r_1(e)\ w_1(i)\ c_1\ w_3(h)\ c_2\ c_3$.



Conceptual test for serializability

- Example:



Legend:

- Read by t1
- Write by t1
- Read by t2
- Write by t2
- Read by t3
- Write by t3

Serializability relationships

Without proof:

$$correct_F(H) \supseteq correct_V(H) \supseteq correct_C(H)$$

where

- Final-state serializability: *F*
- View serializability: *V*
- Conflict serializability: *C*

From histories to schedules

- Serializability is defined on histories: Only for completed transactions we know their outcome.
- Moreover, serializability ignores aborted transactions because *FIN* (and *RF*, *conf*) eliminate them from further consideration.
- Serializability can be extended to schedules if we consider only those transactions that have already committed.
  - **Definition 4.24 (committed projection *CP*(*s*))**

    Let s be a schedule over $T = \{t_1,\ldots,t_n\}$. Then

    $$CP(s) := s|_{t_i \in commit(s)}$$

    $CP(s)$ is a history.
  - A schedule s is X-serializable if there exists a serial history h_s s.t. $\exists h_s$ serial: $CP(s) \equiv_X h_s$

DBS

Universität Karlsruhe (TH)

Chapter 5

Isolation: Serializability

Agenda

- Conflict serializability
- Order preservation
- View serializability
- Final-state serializability
- Commit serializability

Conflict serializability

Conflict equivalence

Remember: We study histories, and we ignore aborted transactions.

Definition 5.1 (Conflict equivalence)

(Def. 4.23 specialized)

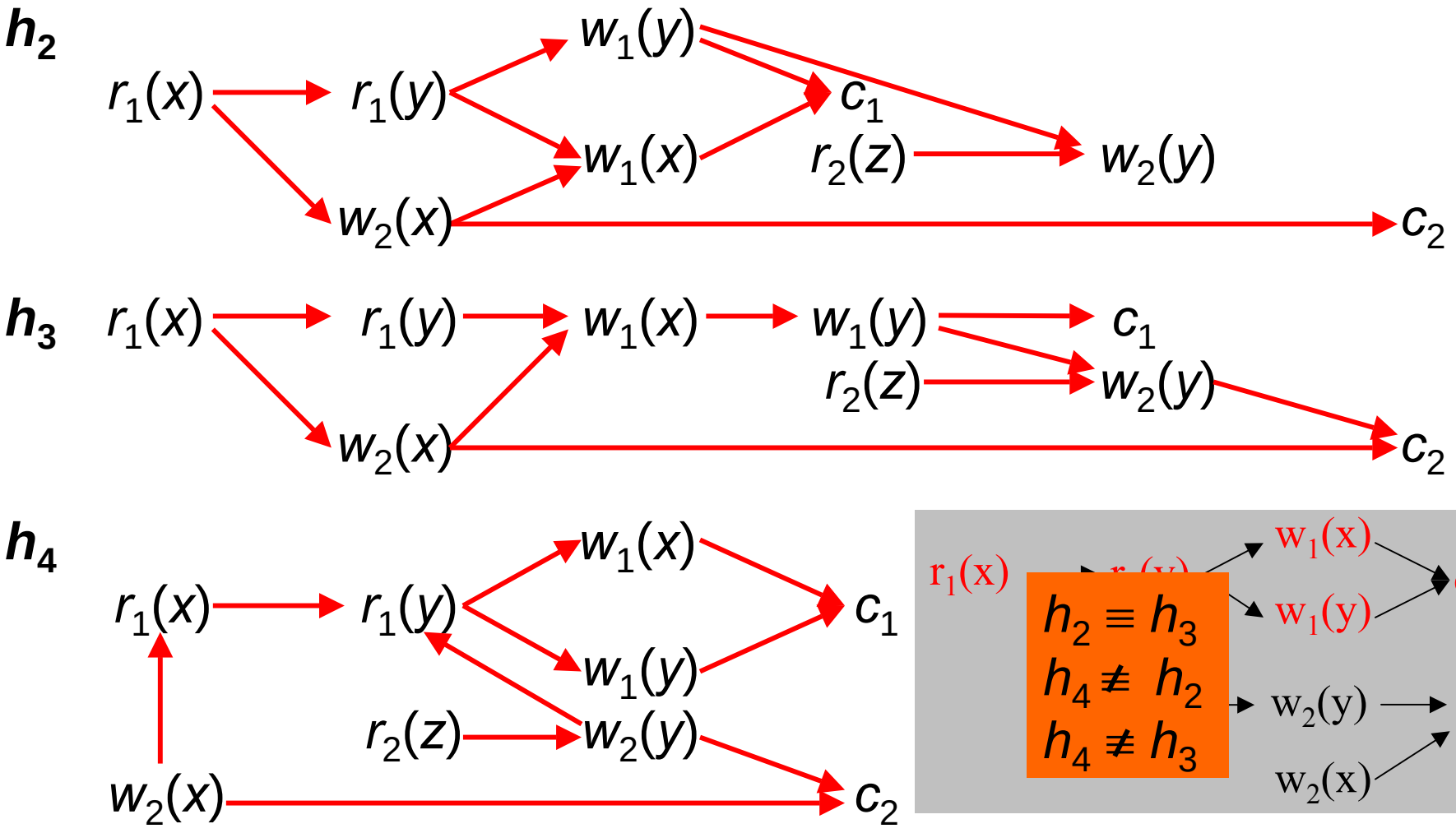
Two histories *h* und *h'* are **conflict equivalent** ($h \equiv_C h'$)

$\Leftrightarrow$

- *op(h) = op(h')*
- *conf(h) = conf(h')*

Conflict equivalence

Example 5.2 Histories over two transactions:



Conflict serializability

Definition 5.3

- **Conflict serializability**: A history h is conflict serializable if there exists a conflict equivalent serial history h_s: $\exists h_s$ serial: $CP(h) \equiv_C h_s$
- A schedule s is conflict serializable if there exists a serial history h_s s.t. $\exists h_s$ serial: $CP(s) \equiv_C h_s$

Corollary 5.4

- A history h is **conflict serializable** if

  $\exists h_s$ serial: $CP(h) \equiv_C h_s \prec\succ conf(h) = conf(h_s)$.

Remember: $conf(s)$ ignores aborted transactions

Definition 5.5

CSR ::= Set of all conflict serializable histories.

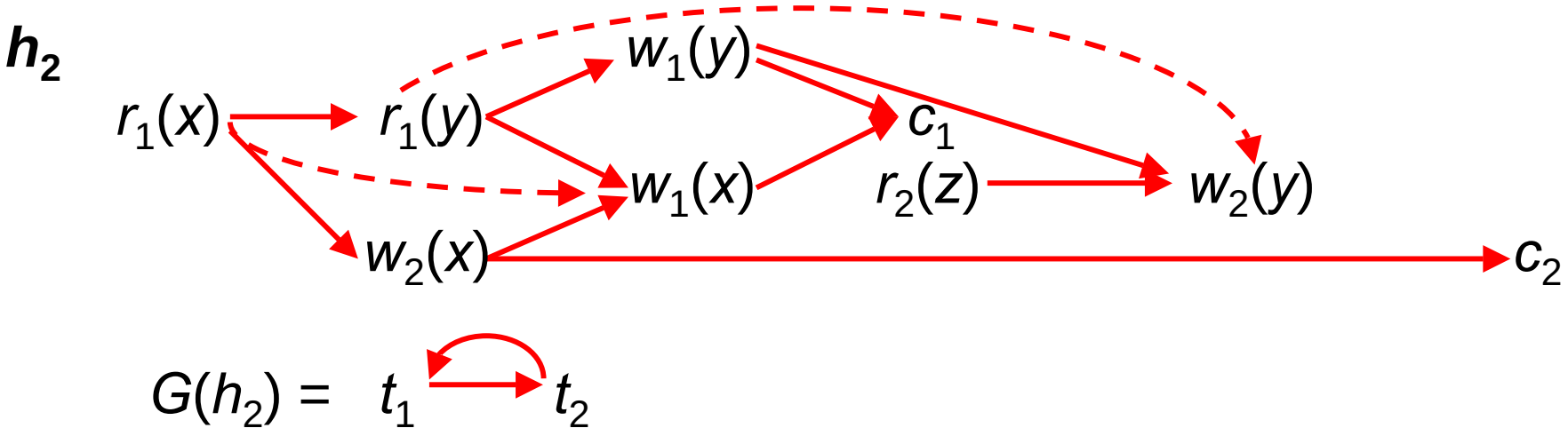
Proving conflict serializability

- **Idea**: Find a good characterization for *conf*(h_s). Then, for *h* serializable the same characterization should hold for *conf*(*h*).
- **Needed**: Efficient proof procedure.
- **Standard approach**: Try a graph.

Conflict graph

Definition 5.6 (Conflict Graph):
Let h be a history. The **conflict graph** G(h) = (V, E) is a directed graph with

vertices V := commit(h) and
edges E := {(t, t‘) | t ≠ t‘
and there are operations p ∈ t, q ∈ t‘ with (p, q) ∈ conf(h)}.



Conflict graph

Definition 5.6 (Conflict Graph):
Let h be a history. The **conflict graph** G(h) = (V, E) is a directed graph with

vertices V := commit(s) and
edges E := {(t, t‘) | t ≠ t‘
and there are operations p ∈ t, q ∈ t‘ with (p, q) ∈ conf(h)}.

- An edge $t \rightarrow t'$ indicates that *at least one* operation in *t* precedes an operation in *t‘* and the two are in conflict.
- In a serial history: All operations in *t* precede those in *t‘*.
  - G(h_s), h_s serial, is acyclic.
  - Failure if G(*h*) contains a cycle.

Correctness proof

From intuition to proof:

Theorem 5.7:

Let h be a history. Then $h \in CSR \prec\succ$ G(h) is acyclic.

Correctness proof

(i) $h \in CSR \prec\succ G(h)$ is acyclic

Let h be a history in CSR. So there is a serial history h‘ with conf(h) = conf(h‘).
Now assume that G(h) has a cycle $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_k \rightarrow t_1$.
This implies that there are pairs $(p_1, q_2), (p_2, q_3), ... , (p_k, q_1)$
with $p_i \in t_i$, $q_i \in t_i$, $p_i <_h q_{(i+1)}$, and p_i in conflict with $q_{(i+1)}$.
Because h‘$\equiv_C$ h, it also implies that $p_i <_{h‘} q_{(i+1)}$.
Because h‘ is serial, we obtain $t_i <_{h‘} t_{(i+1)}$ for i=1, ..., k-1, and $t_k <_{h‘} t_1$.
By transitivity we infer $t_1 <_{h‘} t_2$ and $t_2 <_{h‘} t_1$, which is impossible.
The initial assumption is wrong. So G(h) is acyclic.

(ii) $h \in CSR \prec\succ G(h)$ is acyclic

Let G(h) be acyclic. So it must have at least one source node.
The following topological sort produces a total order < of transactions:

a) start with a source node (i.e., a node without incoming edges),
b) remove this node and all its outgoing edges,
c) iterate a) and b) until all nodes have been added to the sorted list.

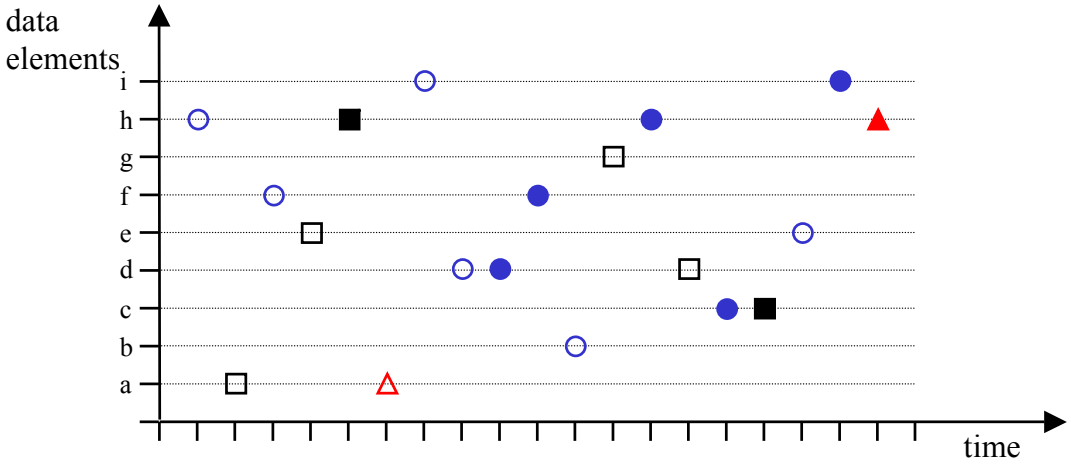
The total transaction ordering order < preserves the edges in G(h);
therefore it yields a serial schedule h‘ for which h‘$\equiv_C$ h.

Efficient correctness proof

- From proof (ii) we conclude that if for a history *h,* G(*h*) is acyclic, *h* is equivalent to every serial history that is a topological order of G(*h)*.
- Since in general there are several such orders, there may be several serial histories that are equivalent to *h*.
- Conflict serializability can be computed in polynomial time – in fact in square time because of the simple test on acyclicity.

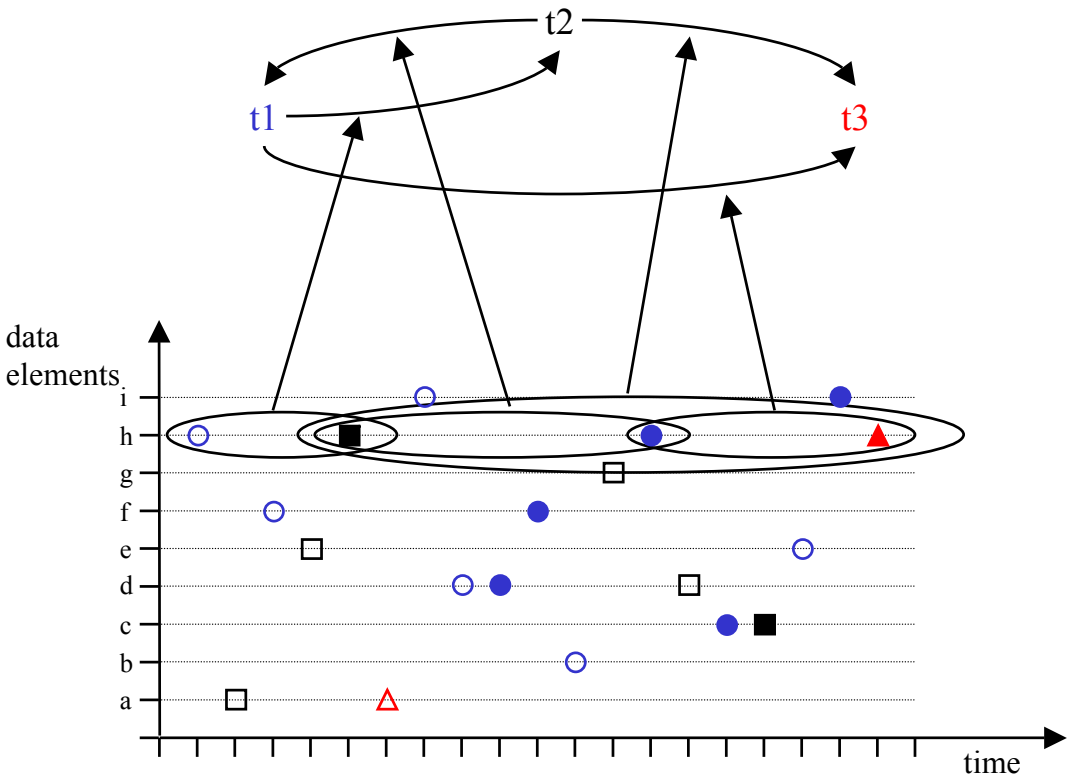
Efficient correctness proof: Illustration

- **Example 5.8**: Take earlier history that was view serializable but not conflict serializable:
  $r_1(h)\ r_2(a)\ r_1(f)\ r_2(e)\ w_2(h)\ r_3(a)\ r_1(i)\ r_1(d)\ w_1(d)\ w_1(f)\ r_1(b)$
  $r_2(g)\ w_1(h)\ r_2(d)\ w_1(c)\ w_2(c)\ r_1(e)\ w_1(i)\ c_1\ w_3(h)\ c_2\ c_3$.



Efficient correctness proof: Illustration

- **Example 5.8 cont'd**: Conflict graph:



Legend:

- Read by t1
- Write by t1
- Read by t2
- Write by t2
- Read by t3
- Write by t3

Efficient correctness proof: Illustration

Example 5.9: Histories in the booking example:

h1: $r_1(B)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_1(T)\ c_1$ ↯

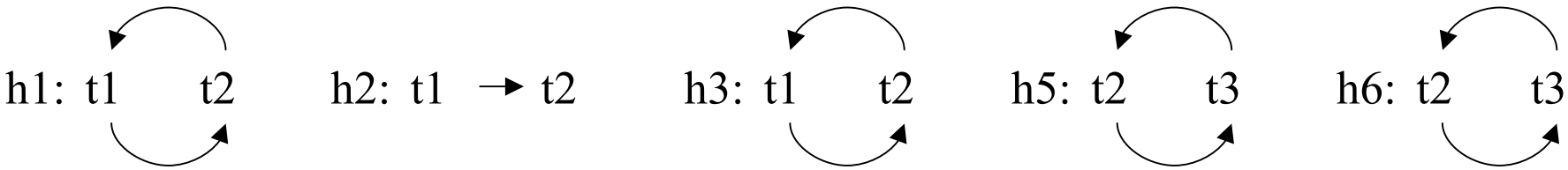
h2: $r_2(B)\ r_2(T)\ r_1(B)\ r_1(T)\ c_1\ w_2(T)\ w_2(B)\ c_2$ √

h3: $r_2(B)\ r_2(T)\ w_2(T)\ r_1(B)\ r_1(T)\ c_1\ w_2(B)\ c_2$ ↯

h4: $r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ r_1(B)\ r_1(T)\ a_2\ c_1$ only t1 to consider!

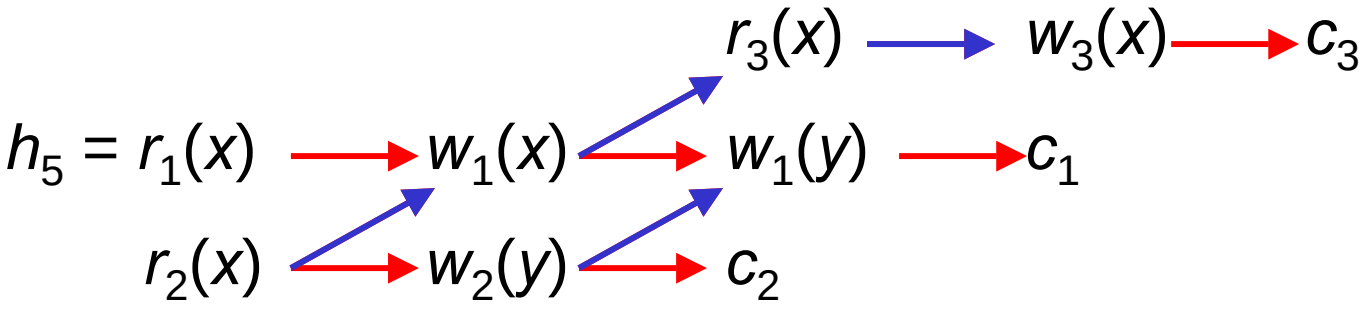
h5: $r_3(T)\ w_3(T)\ r_2(B)\ r_2(T)\ w_2(T)\ w_2(B)\ c_2\ r_3(B)\ w_3(B)\ c_3$ ↯

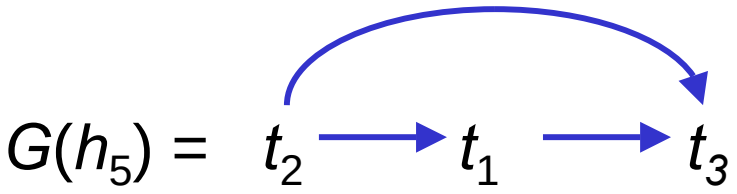
h6: $r_2(B)\ r_2(T)\ r_3(T)\ w_3(T)\ r_3(B)\ w_3(B)\ c_3\ w_2(T)\ w_2(B)\ c_2$ ↯



Efficient correctness proof: Illustration

Example 5.10:

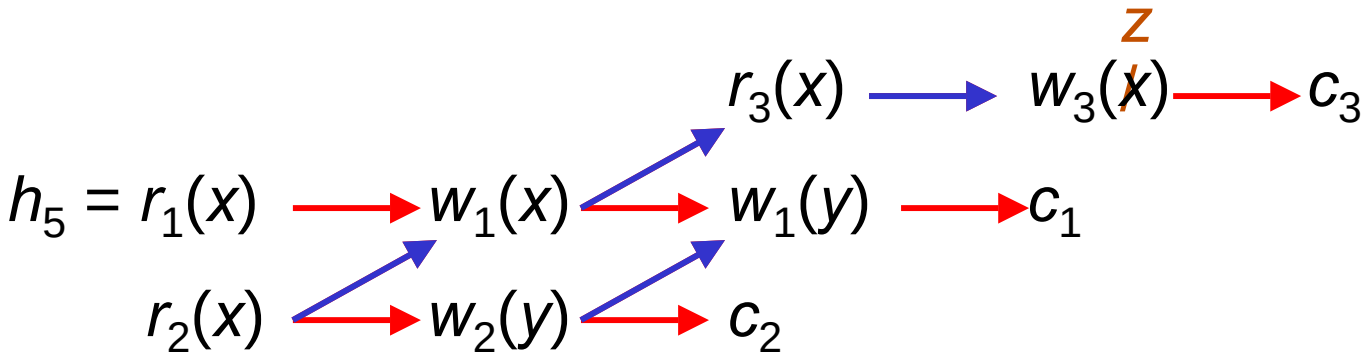


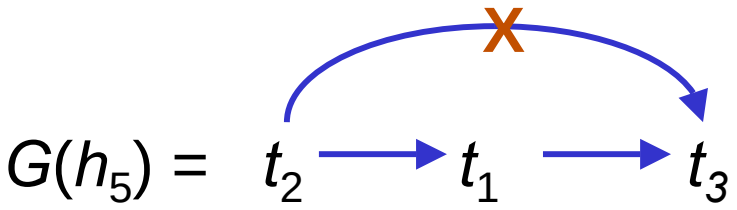


Equivalent serial history: t_2, t_1, t_3

Efficient correctness proof: Illustration

Example 5.10 modified:





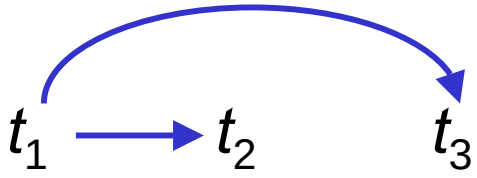
Equivalent serial history: t_2, t_1, t_3

Note: In general $t_i \rightarrow t_j$, $t_j \rightarrow t_k \not\Rightarrow t_i \rightarrow t_k$.

Efficient correctness proof: Illustration

Example 5.11:

$h_6 = w_1(x)\ w_1(y)\ c_1\ r_2(x)\ r_3(y)\ w_2(x)\ c_2\ w_3(y)\ c_3$

$G(h_6) =$ 

Equivalent serial histories:

t_1, t_2, t_3

t_1, t_3, t_2

Serialization by reordering

Remember our examples with serialization by reordering of operations. Is there a relationship between CSR and reordering? Formalize:

Commutativity rules:

C1: $r_i(x)\ r_j(y) \sim r_j(y)\ r_i(x)$ if $i \neq j$

C2: $r_i(x)\ w_j(y) \sim w_j(y)\ r_i(x)$ if $i \neq j$ and $x \neq y$

C3: $w_i(x)\ w_j(y) \sim w_j(y)\ w_i(x)$ if $i \neq j$ and $x \neq y$

Ordering rule:

C4: $o_i(x), p_j(y)$ unordered $\sim o_i(x)\ p_j(y)$
if $x \neq y$ or both o and p are reads

Serialization by reordering

Commutativity rules:

C1: $r_i(x)\ r_j(y) \sim r_j(y)\ r_i(x)$ if $i \neq j$

C2: $r_i(x)\ w_j(y) \sim w_j(y)\ r_i(x)$ if $i \neq j$ and $x \neq y$

C3: $w_i(x)\ w_j(y) \sim w_j(y)\ w_i(x)$ if $i \neq j$ and $x \neq y$

Ordering rule:

C4: $o_i(x), p_j(y)$ unordered $\sim o_i(x)\ p_j(y)$
if $x \neq y$ or both o and p are reads

Example for transformations of schedules:

s $= w_1(x)\ r_2(x)\ w_1(y)\ w_1(z)\ r_3(z)\ w_2(y)\ w_3(y)\ w_3(z)$

~ (C2) $w_1(x)\ w_1(y)\ r_2(x)\ w_1(z)\ w_2(y)\ r_3(z)\ w_3(y)\ w_3(z)$

~ (C2) $w_1(x)\ w_1(y)\ w_1(z)\ r_2(x)\ w_2(y)\ r_3(z)\ w_3(y)\ w_3(z)$

$= t_1\ t_2\ t_3$

$r_2(x)$ and $r_3(z)$ can be delayed

Commutativity-based reducibility

Definition 5.12 (Commutativity Based Equivalence):
Schedules s and s‘ s.t. op(s)=op(s‘) are **commutativity based equivalent**, denoted s ~* s‘, if s can be transformed into s‘ by applying rules C1, C2, C3, C4 finitely many times.

Theorem 5.13:
Let s and s‘ be schedules s.t. op(s)=op(s‘). Then $s \equiv_c s'$ iff s ~* s‘.

Definition 5.14 (Commutativity Based Reducibility):
Schedule s is **commutativity-based reducible** if there is a serial schedule s‘ s.t. s ~* s‘.

Corollary 5.15:
Schedule s is commutativity-based reducible iff $s \in CSR$.

Commutativity-based reducibility

Practical value?

- Scheduler may control the sequence of operations
  - solely based on observing/recognizing conflicts
- Just tell the scheduler which conflicts can occur
  - It does not need to know any further operator details

Order preservation

Order preserving conflict serializability

Example 5.16

Take history

$$h = w_1(x)\ r_2(x)\ c_2\ w_3(y)\ c_3\ w_1(y)\ c_1$$

$$G(h) = \ t_3 \longrightarrow t_1 \longrightarrow t_2$$

Unpleasant because t_2 was committed by the time t_3 starts.

Imagine we keep a chronological record of the transactions and for some reason would have to redo the transactions!

Order preserving conflict serializability

Definition 5.17

History h (or the committed projection $h=CP(s)$ of schedule s) is **order preserving conflict serializable** if h is conflict equivalent to a serial history h_s where

$t,t' \in h$:

if t occurs completely before t' in h

then the same holds in h_s

Definition 5.18

OCSR ::= Set of all order preserving conflict serializable histories.

⇨ In OCSR histories, transactions cannot commute if they do not overlap in time

Order preserving conflict serializability

Corollary 5.19

$$OCSR \subset CSR$$

by Def. 5.17. See example 5.16 for proper inclusion :

$G(h) \notin OCSR$, because $t_3\ t_1\ t_2$ does not preserve the order of the non-overlapping transactions t_2, t_3.

Commit-Order Preservation (1)

Definition 5.20 (Commit-Order Preservation)

History *h* is **commit-ordered** if:

$\forall\ t_i, t_j \in h,\ i \neq j,\ p \in op_i,\ q \in op_j$: $(p,q) \in conf(h) \succ c_i <_h c_j$

Definition 5.21

COCSR ::= Set of all commit-ordered histories.

⇨ The order of two operations that are in conflict agrees with the order of the commit operations of their respective transactions.

Commit-Order Preservation (2)

Theorem 5.22

$$COCSR \subset CSR$$

Proof:

Let $h \in COCSR$ und $(t_i,t_j) \in G(h)$.

(definition $COCSR$) $\succ c_i <_h c_j$.

By induction: for all paths $(t_1,\ldots,t_n) \in G(h)$:

$$c_i <_h c_{i+1} \text{ for } 1 \neq i < n.$$

$COCSR \subset CSR$: Let $h \notin CSR \succ G(h)$ cyclic. Let $(t_1,\ldots,t_n,t_1)$ be a cycle. $\succ c_1 <_h c_1$ (contradiction).

$CSR \not\subset COCSR$: $h = r_1(x)\ w_2(x)\ c_2\ c_1$ is CSR but not $COCSR$.

Commit-Order Preservation (3)

Theorem 5.23

$COCSR \subset OCSR$

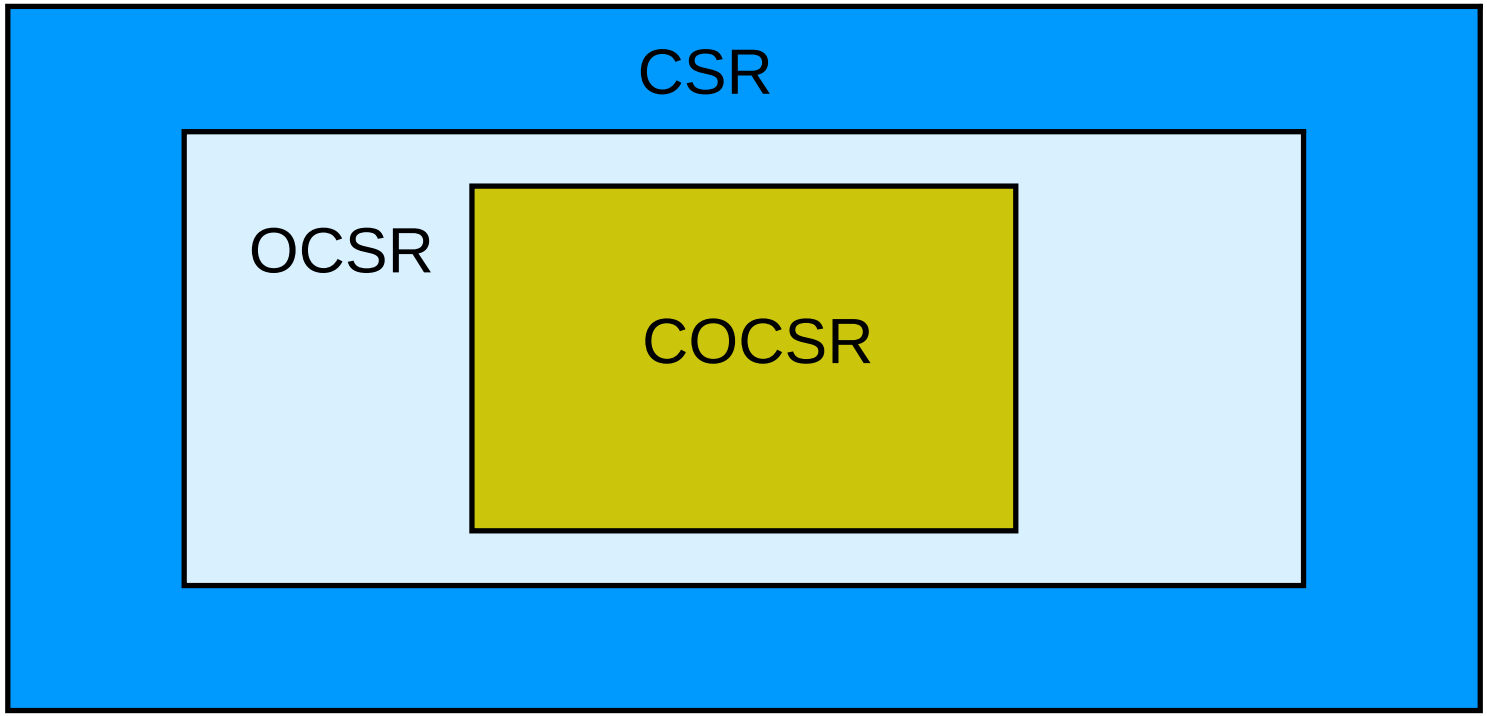
Example 5.24

- $h_1 = w_1(x)\ r_2(x)\ c_2\ w_3(y)\ c_3\ w_1(y)\ c_1$ *CSR, not OCSR, COCSR*
- $h_2 = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ c_2\ w_1(y)\ c_1$ *OCSR, not COCSR*
- $h_3 = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ w_1(y)\ c_1\ c_2$ *COCSR*

$$G(h) = \ t_3 \longrightarrow t_1 \longrightarrow t_2$$

CSR restrictions

Summary



View serializability

View equivalence (1)

Remember:

Definition 5.25

Two histories h and h' are **view equivalent** ($h \equiv_V h'$): $\prec\succ$

- $op(h)=op(h')$
- $RF(h)=RF(h')$
- $\forall x\ FIN_h(x) = FIN_{h'}(x)$

$RF(h) := \{(t_i,x,t_j) \mid t_j \rhd_h(x)\ t_i\}$

where $t_j \rhd_h(x)\ t_i$ ("t_j reads x from t_i"):

- $w_i(x) <_h r_j(x)$, $a_i \not<_h r_j(x)$
- $\forall k \neq i,j\ \ w_i(x) <_h w_k(x) <_h r_j(x) \Rightarrow a_k <_h r_j(x)$

View serializability (1)

Remember:

Definition 5.26

A history h is **view serializable** if there exists a view equivalent serial history h_s: $\exists h_s$ serial: $CP(h) \equiv_V h_s$

Definition 5.27

VSR ::= Set of all view serializable histories.

$RF(h)$ **can be simplified for** $CP(h)$:

$t_j \triangleright_{CP(h)}(x)\ t_i$:

- $w_i(x) <_{CP(h)} r_j(x)$ and $\nexists w_k(x)\ w_i(x) <_{CP(h)} w_k(x) <_{CP(h)} r_j(x)$

Sichtserialisierbarkeit

Satz 5.28 $CSR \subset VSR$

Beweis:

$h \in CSR \succ \exists h_s$ seriell $CP(h) \equiv_C h_s$

Zu zeigen: $CP(h) \equiv_V h_s$.

Zwei Historien h und h' sind sichtäquivalent ($h \equiv_V h'$): $\prec\succ$

- $op(h)=op(h')$
- $RF(h)=RF(h')$
- $\forall x\ FIN_h(x) = FIN_{h'}(x)$

Sichtserialisierbarkeit

Satz 5.28 *CSR*

Beweis:

$h \in CSR \succ \exists h_s$

Zu zeigen: $CP(h) \equiv_V h_s$.

Da in *CSR* für alle i, j $w_i(x) \| w_j(x)$ gilt und beide Historien $CP(h)$ und h_s unverträgliche Operationen in der gleichen Reihenfolge ordnen, sind ihre letzten Schreiboperationen identisch.

$t_i \triangleright_{CP(h)}(x)\ t_j$

$\succ$ $w_j(x) <_{CP(h)} r_i(x)$ und

$\nexists w_k(x)\ w_j(x) <_{CP(h)} w_k(x) <_{CP(h)} r_i(x)$

$\succ_{(CSR)}$ $w_j(x) <_{h_s} r_i(x)$ und

$\nexists w_k(x)\ w_j(x) <_{h_s} w_k(x) <_{h_s} r_i(x)$

$\succ$ $t_i \triangleright_{h_s}(x)\ t_j$

Zwei Historien h und h' sind

- $op(h)=op(h')$ ✔
- $RF(h)=RF(h')$
- $\forall x\ FIN_h(x) = FIN_{h'}(x)$

Echtheit der Teilmengenbeziehung folgt aus früherem Beispiel in Kapitel 4!

View equivalence (2)

How to use solely *RF*:

Definition 5.29 (History completion)

Let h be a history over $T = \{t_1,\ldots,t_n\}$. We complete h to history $\hat{h}$ over $t \cup \{t_0,t_\infty\}$:

- t_0 is fictitious first transaction initializing all database elements $x_1,\ldots,x_k$ that are referenced by h:

  $t_0 = w_0(x_1) \ldots w_0(x_k)\ c_0$,

- t_∞ is fictitious last transaction with operations $r_\infty(x)$ for all data elements x appearing in h:

  $\forall p \in h,\ x \in h \quad p <_h r_\infty(x)$

View serializability (2)

Definition 5.30

Two histories h und h' are **view equivalent** ($h \equiv_V h'$): $\prec\succ$

- $RF(\hat{h})=RF(\hat{h}')$

Definition 5.31

- A history h is **view serializable** if there exists a view equivalent serial history h_s:

  $\exists h_s$ serial: $CP(h) \equiv_v h_s \prec\succ RF(\hat{h})=RF(\hat{h}_s)$

Proving view serializability (1)

- **Idea**: Find a good characterization for $RF(h_s)$. Then, if h is serializable the same characterization should hold for $RF(h)$.
- **Needed**: Efficient proof procedure.
- **Standard approach**: Try a graph.

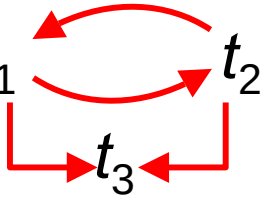
Proving view serializability (2)

But: Conflict graph is insufficient!.

Example 5.32 Take

$$h_{12} = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$$

$G(h_{12}) =$



Not conflict serializable!
But view serializable: $t_1\ t_2\ t_3$ or $t_2\ t_1\ t_3$

Proving view serializability (2)

Remember:

$t_j \triangleright_{CP(h)}(x)\ t_i$:

- $w_i(x) <_{CP(h)} r_j(x)$ and $\nexists w_k(x)\ w_i(x) <_{CP(h)} w_k(x) <_{CP(h)} r_j(x)$

- The reads-from relation must not be broken up by a write operation of some other transaction. All such write operations must appear before the write or after the read.

⇨ We need a type of graph that reflects this condition.

Nachweis der Serialisierbarkeit (3)

Definition 5.33 (Polygraph)

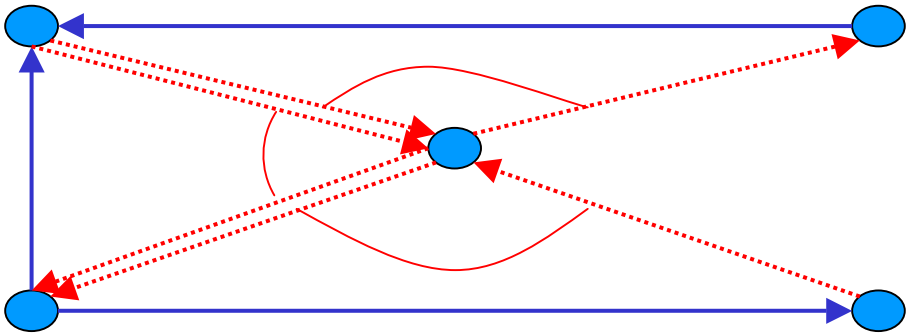
Ein **Polygraph** ist ein Tripel $P = (V,E,C)$ mit

- (V,E) ist ein gerichteter Graph und
- $C \subseteq V \times V \times V$ mit $(u,v,w) \in C \Rightarrow u \neq v \neq w$, $(w,u) \in E$.

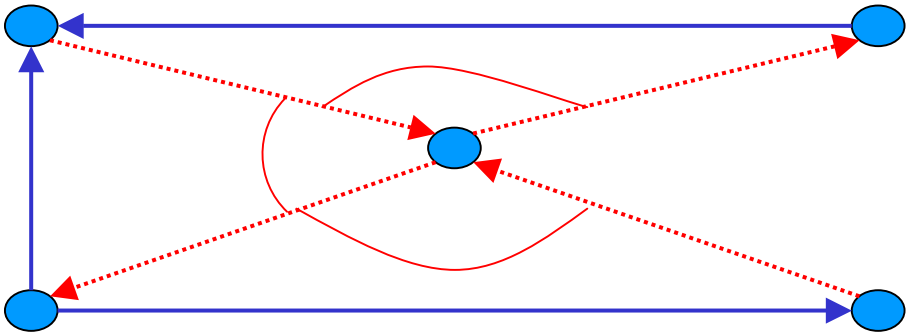
Veranschaulichung

- (V,E) ist ein gerichteter Graph und
- $C \subseteq V \times V \times V$ mit $(u,v,w) \in C \Rightarrow u \neq v \neq w$, $(w,u) \in E$.

Darstellung von (u,v,w) als zwei gerichtete Kanten (Bipfad) $(u,v),(v,w)$, die über Kreissegment verbunden sind.



vereinfacht



Nachweis der Serialisierbarkeit (4)

Definition 5.34 (kompatibel)

Sei $P = (V,E,C)$ ein Polygraph und $G = (V, E')$ ein Graph mit den selben Knoten. G ist **kompatibel** zu P, wenn E' eine minimale Menge von Kanten mit den folgenden Eigenschaften ist:

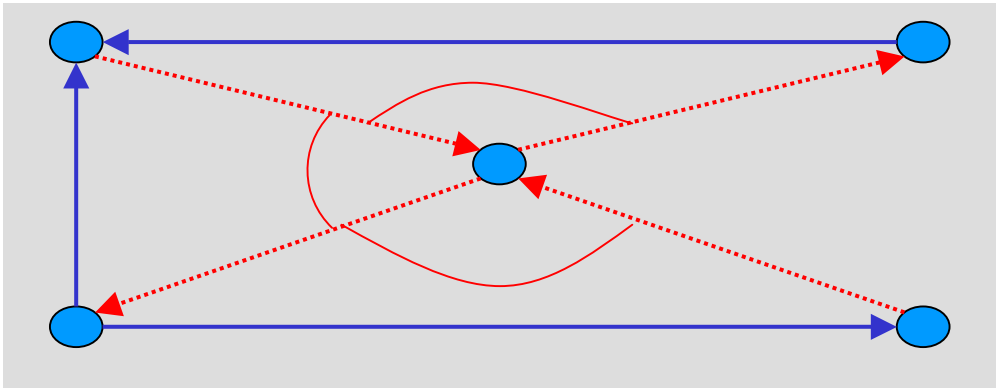
- $E \subseteq E'$
- $\forall\ (u,v,w) \in C\colon (u,v) \in E' \vee (v,w) \in E'$
- $\forall\ (u,v,w) \in C\colon (u,v) \in E' \succ (v,w) \notin E',\ (u,v) \notin E' \succ (v,w) \in E'$

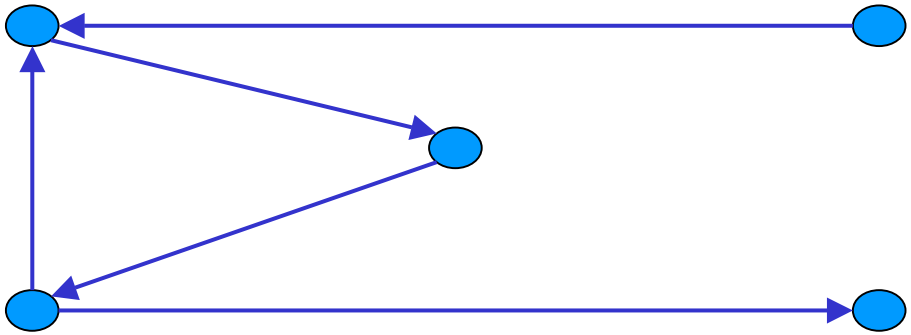
G enthält mindestens alle Kanten aus *E*.

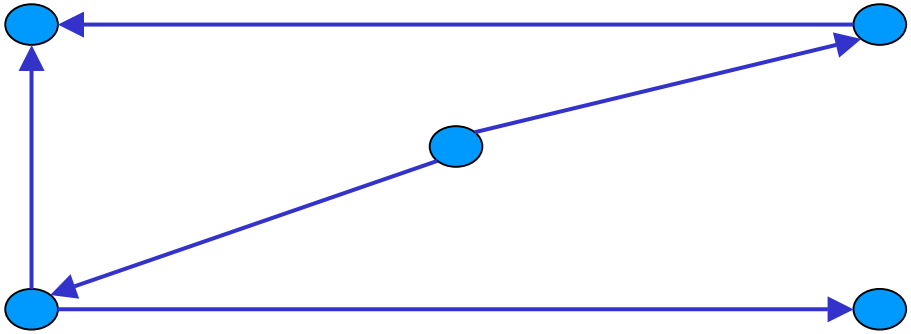
Jeder Bipfad aus *P* ist in *G* durch genau eine Kante repräsentiert

Daher: *C* steht für Choice $\Rightarrow$ Ein Polygraph kann als Familie $\mathbf{D}(V,E,C)$ von gerichteten Graphen (V,E') gedeutet werden.

Beispiel zur Kompatibilität







Nachweis der Serialisierbarkeit (5)

Definition 5.35 ($P(h)$)

Gegeben sei eine Historie h über der Transaktionsmenge $T = \{t_1,\ldots,t_n\}$. Wir definieren $P(h) = (V,E,C)$ mit

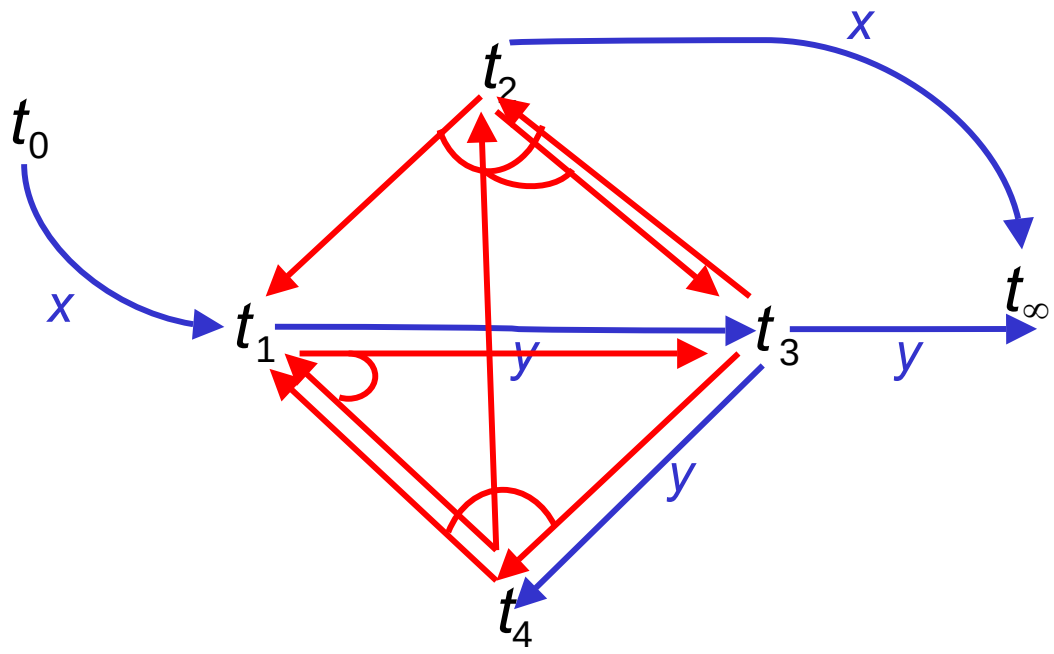
- $V = T \cup \{t_0, t_\infty\}$
- $E = \{ (t_j,t_i) \mid t_i \triangleright_{\hat{h}} (x)\ t_j \}$
- $C = \{ ((t_i,t_k,t_j)) \mid t_i \triangleright_{\hat{h}} (x)\ t_j\ ,\ w_k(x) \in h \}$ mit $i \neq j \neq k$

$w_j(x) < r_i(x)$ und $\exists w_k(x)\ w_j(x) < w_k(x) < r_i(x)$
Eine Kante (t_j,t_i) drückt aus, dass t_j vor t_i stattfinden muss.
Eine Choice (t_i, t_k, t_j) drückt aus, dass t_k entweder nach t_i oder vor t_j stattfinden muss.

Nachweis der Serialisierbarkeit (6)

Beispiel:

$h_1 = w_0(x)\ w_0(y)\ r_1(x)\ w_2(y)\ w_1(y)\ r_3(y)\ w_3(y)\ w_2(x)\ r_4(y)\ r_\infty(x)\ r_\infty(y)$

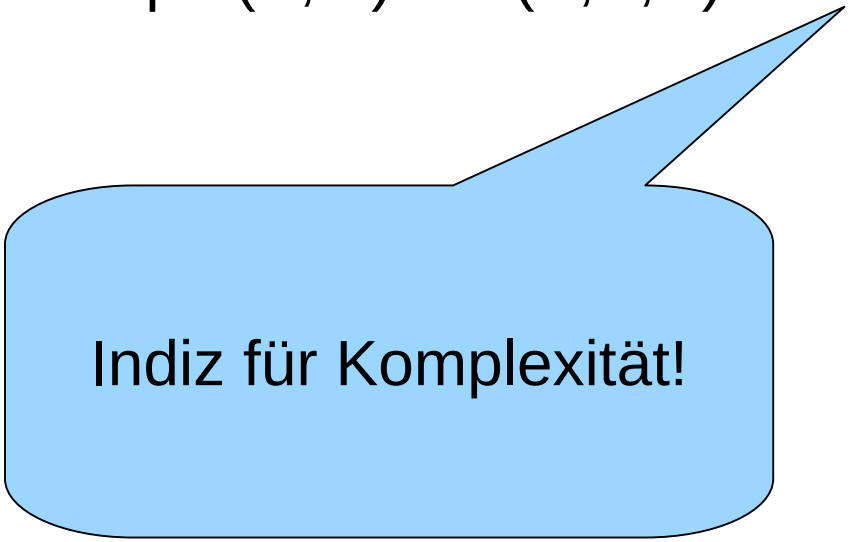


Anmerkung: Für t_0, t_∞ gibt es keine Wahl: t_0 liest nicht, t_∞ schreibt nicht.

Nachweis der Serialisierbarkeit (7)

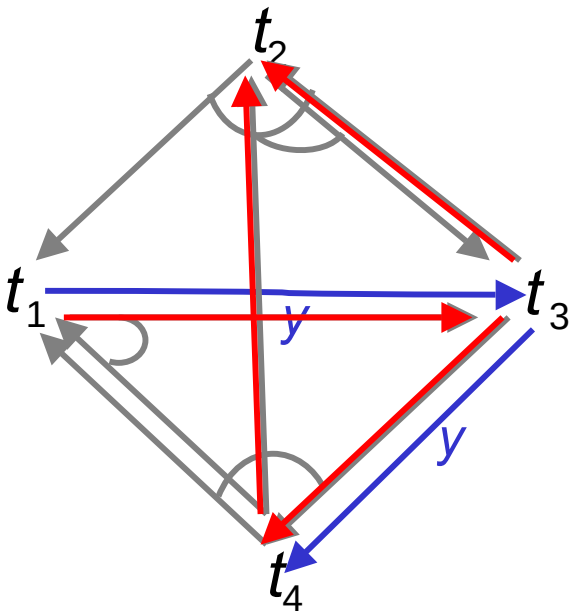
Definition 5.36 (azyklischer Polygraph)

Ein Polygraph $P = (V,E,C)$ heißt **azyklisch**, falls mindestens ein azyklischer gerichteter Graph (V,E') in $\mathbf{D}(V,E,C)$ existiert.

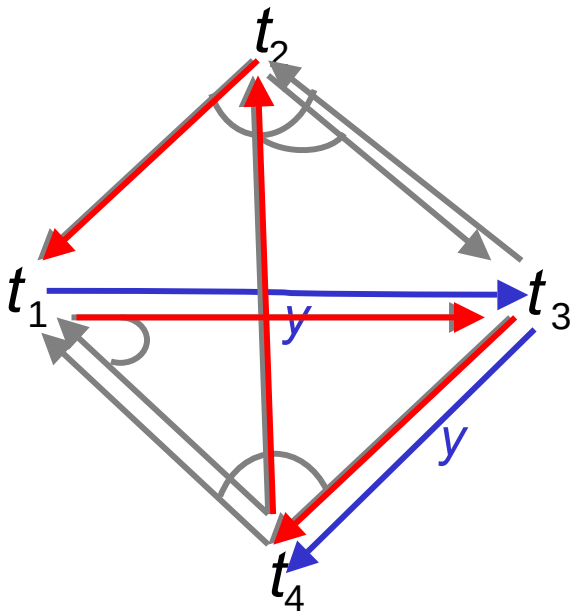


Nachweis der Serialisierbarkeit (8)

Voriges Beispiel: Zwei gerichtete Graphen aus der Familie



zyklenfrei



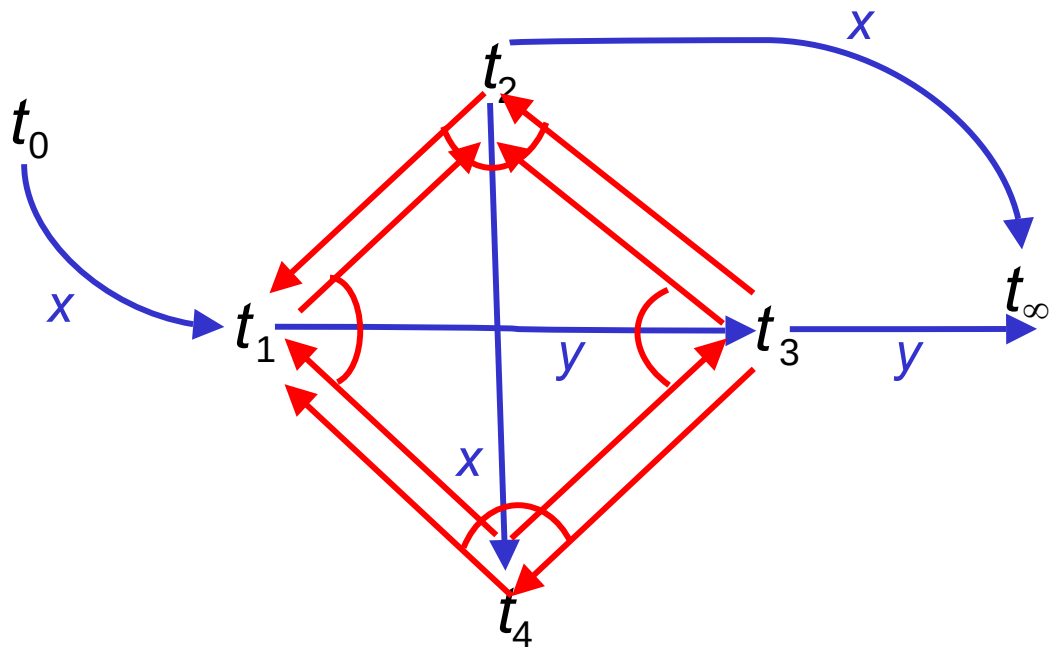
zyklenbehaftet

t_0, t_∞ können nicht an Zyklen partizipieren.

Nachweis der Serialisierbarkeit (9)

Gegenbeispiel:

$$h_1 = w_0(x)\ w_0(y)\ r_1(x)\ w_2(y)\ w_1(y)\ r_3(y)\ w_3(y)\ w_2(x)\ r_4(y)\ r_\infty(x)\ r_\infty(y)$$



Anmerkung: Für t_0, t_∞ gibt es keine Wahl: t_0 liest nicht, t_∞ schreibt nicht.

Nachweis der Serialisierbarkeit (10)

Lemma 5.37

Seien h und h' zwei Historien. Dann gilt:

$$h \equiv_V h' \prec\succ P(h) = P(h').$$

Intuitiver Beweis: Beide setzen gleiche Operationen und gleiche Relation *RF* voraus.

Lemma 5.38

Sei h Historie. Dann: h seriell $\prec\succ$ $P(h)$ azyklisch.

Nachweis der Serialisierbarkeit (11)

Beweis zu 5.38:

- Konstruiere $G = (V,E')$ kompatibel zu P:
  - Knoten $V = T \cup \{t_0, t_\infty\}$ übernehmen
  - Kanten: $(t,t') \in E$ übernehmen $\succ t < t'$ (sonst könnte t' nicht von t lesen)
  - Choices: $(t,t',t'') \in C$ (d.h. Operation p von Transaktion t' schreibt Element, das t von t'' liest:
    - p vor dem Schreiben von t'': t' vor t'': $(t',t'') \in E'$
    - p nach dem Lesen von t: t' nach t: $(t,t') \in E'$
- Somit $E' = \{(t, t') \mid t, t' \in V \wedge t < t'\}$
- (V,E') azyklisch, weil h seriell

Nachweis der Serialisierbarkeit (12)

Satz 5.39

h sichtserialisierbar $\prec\succ$ $P(h)$ azyklisch

(Analogon zum Serialisierbarkeitstheorem für Konfliktserialisierbarkeit.)

Beweis: h sichtserialisierbar $\prec\succ$ $P(h)$ azyklisch

- $\succ$ $\exists h_s$ seriell $h \equiv_V h_s$

  $P(h) = P(h_s)$ nach 5.37

  $P(h_s)$ azyklisch nach 5.38

- $\succ$ $P(h)$ azyklisch

Nachweis der Serialisierbarkeit (13)

Beweis: *h* sichtserialisierbar $\prec\succ$ *P*(*h*) azyklisch

Sei *G* kompatibler Graph. Sei h_s eine topologische Ordnung von *G*.

Zu zeigen: $h_s \equiv_V h$.

Widerspruchsbeweis:

RF sei verschieden,

- d.h. es gibt $r_t(x)$ mit $r_t(x) \rhd_h w_i(x)$, $r_t(x) \rhd_{h_s} w_j(x)$
  - $\square$ $(t, j, i) \in P(h)$
  - $\square$ $(i, t) \in G$ und damit $i<t \in h_s$
  - außerdem: $i<j \in h_s$ und damit $(i, j) \in G$
  - Widerspruch zu (t, j, i)

Entscheidbarkeit der Serialisierbarkeit

- Zur Erinnerung: Das Entscheidungsproblem, ob eine Historie konfliktserialisierbar ist oder nicht, kann in polynomieller Zeit berechnet werden.
- Man konnte bisher nur zeigen, dass das entsprechende Entscheidungsproblem für Sichtserialisierbarkeit NP-vollständig ist.
- Siehe dazu C.H.Papadimitriou: The Serializability of Concurrent Database Updates, JACM 46(4), 631ff, Oct. 79.

Final-State Serializability

(für Interessierte)

Final-state equivalence

Remember:

Definition 5.40

Two histories h and h' are **final-state equivalent** ($h \equiv_F h'$) $\prec\succ$

- $op(h)=op(h')$
- $\forall x \in D\ FIN_h(x) = FIN_{h'}(x)$ (D database)

Compare to:

Two histories h and h' are **view equivalent** ($h \equiv_V h'$): $\prec\succ$

- $op(h)=op(h')$
- $RF(h)=RF(h')$
- $\forall x \in D\ FIN_h(x) = FIN_{h'}(x)$ (D database)

Herbrand-Semantik von Historien (1)

Consequence:

- Unlike before, there is no defined relationship to start from.
- All we can do is take a snapshot and characterize the current state.
- The characterization should take the past actions into account: **semantic characterization** of a history.
- Approach to semantics in mathematical logic: Construct a domain as a set of (syntactic) terms, each for a specific individual (Herbrand domain).

Herbrand-Semantik von Historien (2)

Interpretation of j^{th} step, p_j, of t:

- If $p_j = r(x)$,
  then interpretation is assignment $v_j := x$ to local variable v_j
- If $p_j = w(x)$
  then interpretation is assignment $x := f_j(v_{j1}, ..., v_{jk})$
  with unknown function f_j
  and $j_1, ..., j_k$ denoting t's prior read steps.

Beispiel

$$t = w(u)\ r(x)\ w(v)\ r(y)\ w(w)$$

Die Werte der geschriebenen Elemente sind abhängig von allen vorher gelesenen Elementen. Also:

$$u = f_1()$$
$$v = f_2(x)$$
$$w = f_3(x,y)$$

Herbrand-Semantik von Historien (3)

Definition 5.41 (Herbrand- Semantik)

Sei h eine Historie. Die **Herbrand-Semantik** H_h der Schritte $r_i(x), w_i(x) \in op(h)$ wird rekursiv wie folgt definiert:

- $H_h(r_i(x)) := H_h(w_j(x))$

  wobei $r_i(x) \triangleright_h w_j(x)$ (also $j \neq i$)

- $H_h(w_i(x)) := f_{i,x}(H_h(r_i(y_1)),\ldots,H_h(r_i(y_m)))$, wobei
  - die $r_i(y_j)$ $(1 \leq j \leq m)$ sämtliche Leseschritte von t_i sind, welche in h vor $w_i(x)$ stehen, und
  - $f_{i,x}$ ein uninterpretiertes m-stelliges Funktionssymbol ist.

Wir machen also Gebrauch von der liest von-Beziehung, aber nur zur Berechnung der formalen Semantik!

Anmerkung: Wir benötigen initiales Schreiben. Daher vervollständigen wir die Historie wie bei Sicht-Serialisierbarkeit um Transaktionen t_0, t_∞.

Herbrand-Semantik von Historien (4)

Beispiel 5.42: Für die Historie

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

gilt für H_h:

$$H_h(w_0(x)) = f_{0,x}()$$

$$H_h(w_0(y)) = f_{0,y}()$$

$$H_h(r_1(x)) = f_{0,x}()$$

$$H_h(r_2(y)) = f_{0,y}()$$

$$H_h(w_2(x)) = f_{2,x}(f_{0,y}())$$

$$H_h(w_1(y)) = f_{1,y}(f_{0,x}())$$

Herbrand-Semantik von Historien (5)

Definition 5.43 (Herbrand-Universum)

Das **Herbrand-Universum** *HU* für Transaktionen t_i, $i > 0$, ist die kleinste Menge von Symbolen, für die gilt:

- $f_{0,x}() \in HU$ für jedes $x \in D$ (D Datenbasis), wobei $f_{0,x}$ ein 0-stelliges Funktionssymbol ist.
- $f_{i,x}(v_1,\ldots,v_m) \in HU$ ($f_{i,x}$ ein m-stelliges Funktionssymbol) falls $w_i(x) \in op(t_i)$, mit
  - $m = \left| \; r_i(y) \mid \exists y \in D \;\; r_i(y) <_{t_i} w_i(x) \; \right|$
  - $v_i \in \{ \; r_i(y) \;\; \left| \;\; \exists y \in D \;\; r_i(y) <_{t_i} w_i(x) \; \right\} \subset HU$, $1 \le i \le m$

Herbrand-Semantik von Historien (6)

Definition 5.44 (Semantik einer Historie)

Die **Semantik** einer Historie h ist die Abbildung

$$H(h) : D \rightarrow HU$$

definiert durch

$$H(h)(x) := H_h(FIN_s(x)) \; (x \in D).$$

Die Semantik eines Schedules ist also die Menge derjenigen Werte, die von nicht-abgebrochenen Transaktionen als letzte geschrieben werden.

Herbrand-Semantik von Historien (7)

Beispiel 5.45 Für die Historie Beispiel 5.42

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

gilt für H_h

$$H_h(w_0(x)) = f_{0,x}()$$
$$H_h(w_0(y)) = f_{0,y}()$$
$$H_h(r_1(x)) = f_{0,x}()$$
$$H_h(r_2(y)) = f_{0,y}()$$
$$H_h(w_2(x)) = f_{2,x}(f_{0,y}())$$
$$H_h(w_1(y)) = f_{1,y}(f_{0,x}())$$

und somit für $H(h)$

$$H(h)(x) = H_h(w_2(x)) = f_{2,x}(f_{0,y}())$$
$$H(h)(y) = H_h(w_1(y)) = f_{1,y}(f_{0,x}())$$

Final-State-Äquivalenz (1)

Definition 5.46 (final-state-äquivalent)

Zwei Historien h und h' heißen **final-state-äquivalent** ($h \equiv_F h'$) □□

- $op(h) = op(h')$
- $H(h) = H(h')$

Zwei Schedules sind also final-state-äquivalent gdw. sie beide die gleichen Operationen umfassen und den gleichen Endzustand erzeugen.

Final-State-Äquivalenz (2)

Beispiel 5.47

Gegeben seien die folgenden Schedules:

$h = r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)\ c_1\ c_2\ c_3$

$h' = r_3(z)\ w_3(z)\ c_3\ r_2(y)\ r_2(x)\ w_2(z)\ c_2\ r_1(x)\ w_1(y)\ w_1(x)\ c_1$

Dann gilt $op(h) = op(h')$ und

$$
\begin{aligned}
H(h)(x) &= H_h(w_1(x)) = && f_{1,x}(f_{0,x}()) \\
&= H_{h'}(w_1(x)) = && H(h')(x) \\
H(h)(y) &= H_h(w_1(y)) = && f_{1,y}(f_{0,x}()) \\
&= H_{h'}(w_1(y)) = && H(h')(y) \\
H(h)(z) &= H_h(w_2(z)) = && f_{2,z}(f_{0,x}(), f_{0,y}()) \\
&= H_{h'}(w_2(z)) = && H(h')(z)
\end{aligned}
$$

also insgesamt $h \equiv_F h'$.

Final-State-Äquivalenz (3)

Beispiel 5.48

Gegeben seien die folgenden Schedules:

$$h = r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ c_1\ c_2$$

$$h' = r_1(x)\ w_1(y)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

Dann gilt $op(h) = op(h')$ und

$$H(h)(y) = H_h(w_2(y)) = f_{2,y}(f_{0,y}())$$

$$H(h')(y) = H_{h'}(w_2(y)) = f_{2,y}(H_{h'}(r_2(y)))$$

$$= f_{2,y}(H_{h'}(w_1(y)))$$

$$= f_{2,y}(f_{1,y}(f_{0,x}()))$$

also insgesamt $h \not\equiv_F h'$.

Dieses Beispiel zeigt: Nicht allein das letzte Schreiben ist entscheidend, sondern auch die Vorgeschichte.

Final-State-Äquivalenz (4)

Aufgabe: Erfassen der „einflussreichen“ Vorgeschichte.

Definition 5.49

Eine Operation p ist in h **direkt nützlich** für eine Operation q ($p \mapsto_h q$), falls

- q ist Leseoperation und liest von fremdem Schreiber p
  $q \triangleright_h p$
- q ist Schreiboperation und p geht als eigener Leser voran
  $p = r_i(x)$, $q = w_i(y)$ und $p <_h q$.

$\mapsto^*$ (*nützlich*) bezeichne die reflexive, transitive Hülle von $\mapsto$.

Final-State-Äquivalenz (5)

Definition 5.50

Eine Operation p heißt **lebendig** in $h \prec\succ$

$$\exists q \in t_\infty \quad p \mapsto^* q$$

d.h., ihr Ergebnis trägt zum Endergebnis bei.

Sie heißt **tot** sonst.

Final-State-Äquivalenz (6)

Grundgedanke: Von der Liest-von-Relation interessieren nur die Operationen mit einem Beitrag zum Endergebnis.

Definition 5.51

Die **Lebendig-Liest-von-Relation** (*live-reads-from*) von *h* ist definiert durch

$$LRF(h) := \{(t_i,x,t_j) \mid r_j(x) \text{ lebendig}, r_j(x) \triangleright_h w_i(x)\}$$

Anmerkung: Alle (Lese-)Operationen aus t_∞ gelten als lebendig. (Also (t_i,x,t_∞) immer in *LRF*).

Final-State-Äquivalenz (8)

Beispiel 5.52 Nach vervollständigtem Beispiel 5.48

$$h = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ c_1\ c_2\ r_\infty(x)\ r_\infty(y)\ c_\infty$$

$$h' = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ w_1(y)\ c_1\ r_2(y)\ w_2(y)\ c_2\ r_\infty(x)\ r_\infty(y)\ c_\infty$$

Dann gilt:

$$RF(h) = \{(t_0,x,t_1), (t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$$

$$RF(h') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$$

In h und h': $w_0(x) \mapsto r_1(x) \mapsto w_1(y) \succ w_0(x) \mapsto^* w_1(y)$

$w_0(x) \mapsto r_\infty(x) \succ w_0(x) \mapsto^* r_\infty(x)$

$r_2(y) \mapsto w_2(y) \mapsto r_\infty(y) \succ r_2(y) \mapsto^* r_\infty(y)$

In h: $w_0(y) \mapsto_h r_2(y) \succ w_0(y) \mapsto_h^* r_\infty(y)$

In h': $w_1(y) \mapsto_{h'} r_2(y) \succ$

$w_1(y) \mapsto_{h'}^* r_\infty(y),\ r_1(x) \mapsto_{h'}^* r_\infty(y),\ w_0(x) \mapsto_{h'}^* r_\infty(y)$

Final-State-Äquivalenz (8)

$RF(h) = \{(t_0,x,t_1), (t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

$RF(h') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

In h und h': $w_0(x) \mapsto r_1(x) \mapsto w_1(y) \succ w_0(x) \mapsto^* w_1(y)$

$w_0(x) \mapsto r_\infty(x) \succ w_0(x) \mapsto^* r_\infty(x)$

$r_2(y) \mapsto w_2(y) \mapsto r_\infty(y) \succ r_2(y) \mapsto^* r_\infty(y)$

In h: $w_0(y) \mapsto_h r_2(y) \succ w_0(y) \mapsto_h^* r_\infty(y)$

In h': $w_1(y) \mapsto_{h'} r_2(y) \succ$

$w_1(y) \mapsto_{h'}^* r_\infty(y), r_1(x) \mapsto_{h'}^* r_\infty(y), w_0(x) \mapsto_{h'}^* r_\infty(y)$

LRF wird aus *RF* gewonnen, indem für jedes Tupel (t_j,x,t_i) betrachtet wird, ob $r_j(x)$ lebendig, d.h. nützlich für eine Operation in t_∞, ist. Also:

$LRF(h) = \{(t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

$LRF(h') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

Final-State-Äquivalenz (9)

Die obigen Überlegungen legen den folgenden Satz nahe:

Satz 5.53

Seien h und h' zwei Historien mit $op(h) = op(h')$.

Dann gilt:

$$h \equiv_F h' \prec\succ LRF(h) = LRF(h')$$

Wieder gesucht: Graphbasierte Lösung
$\Rightarrow$ Graph muss auf *LRF* aufbauen.

Final-State-Äquivalenz (10)

Definition 5.54 (Schrittgraph)

Der **Schrittgraph** $D(h) = (V,E)$ einer Historie h ist definiert durch

$V := op(h)$

$E := \{ (p,q) \mid p \mapsto_h q \}$

Der **reduzierte Schrittgraph** $D_1(h)$ entsteht aus $D(h)$ durch Weglassen aller toten Knoten und deren inzidenten Kanten.

Beweis-Skizze für Äquivalenz: Zeige

- $h \equiv_F h' \prec\succ D_1(h) = D_1(h')$
- $D_1(h) = D_1(h') \prec\succ LRF(h) = LRF(h')$

Final-State-Äquivalenz (11)

Beispiel 5.55

Betrachte die Schedules aus Beispiel 5.47 ($op(h) = op(h')$, h' seriell):

$h = w_0(x,y,z)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)\ c_1\ c_2\ c_3$

$h' = w_0(x,y,z)\ c_0\ r_3(z)\ w_3(z)\ c_3\ r_2(y)\ r_2(x)\ w_2(z)\ c_2\ r_1(x)\ w_1(y)\ w_1(x)\ c_1$

Für diese hatten wir anhand gleicher Endzustände bereits gezeigt: $h \equiv_F h'$.

$$LRF(h) =$$

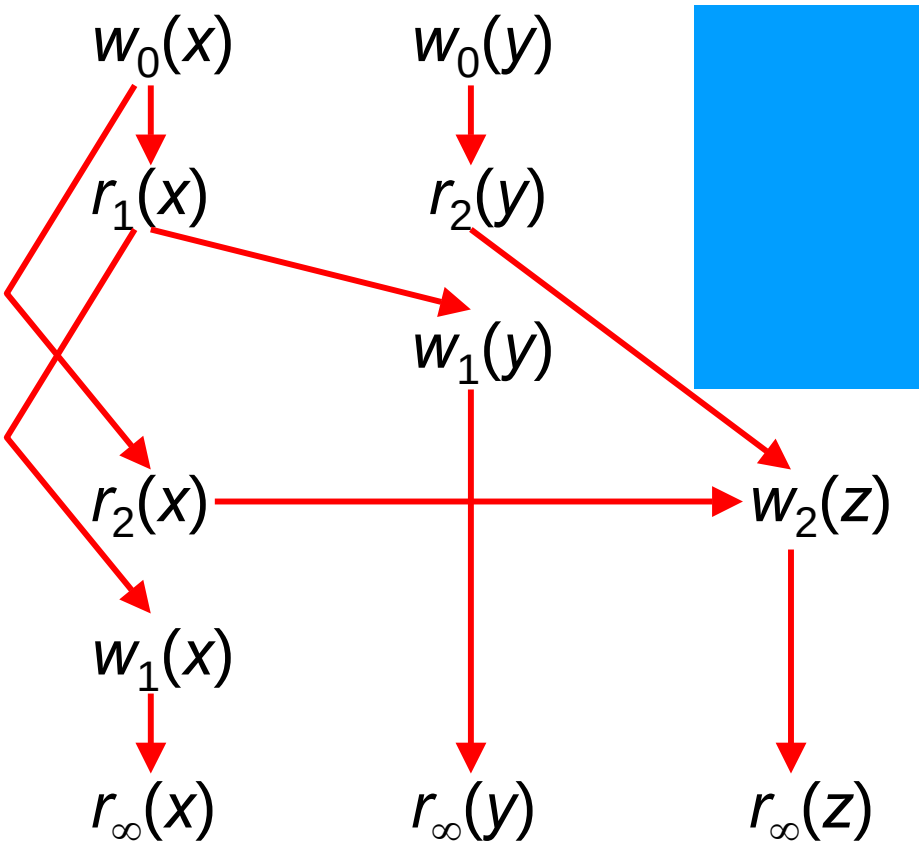
$$\{(t_0,x,t_1),(t_0,x,t_2),(t_0,y,t_2),(t_1,x,t_\infty),(t_1,y,t_\infty),(t_2,z,t_\infty)\}$$

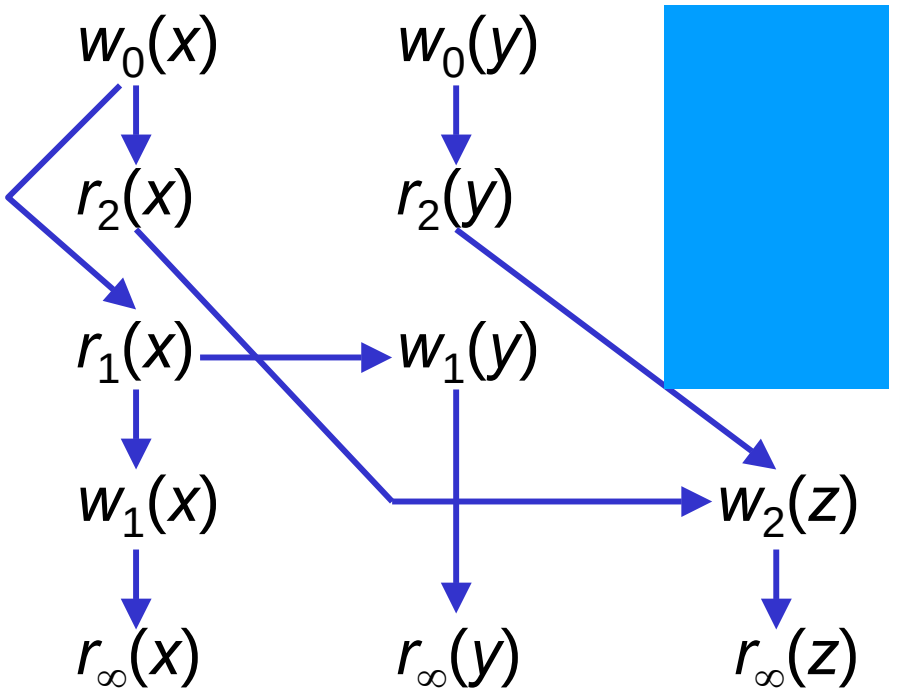
$$= LRF(h')$$

Final-State-Äquivalenz (12)

$h = w_0(x,y,z)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)\ c_1\ c_2\ c_3$

$h' = w_0(x,y,z)\ c_0\ r_3(z)\ w_3(z)\ c_3\ r_2(y)\ r_2(x)\ w_2(z)\ c_2\ r_1(x)\ w_1(y)\ w_1(x)\ c_1$





h und *h'* final-state-äquivalent

Final-State-Äquivalenz (13)

Beispiel 5.56

Betrachte die Historien aus Beispiel 5.52 (*h′* seriell):

$h = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ c_1\ c_2\ r_\infty(x)\ r_\infty(y)\ c_\infty$

$h' = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ w_1(y)\ c_1\ r_2(y)\ w_2(y)\ c_2\ r_\infty(x)\ r_\infty(y)\ c_\infty$

h und *h′* sind nicht final-state-äquivalent:

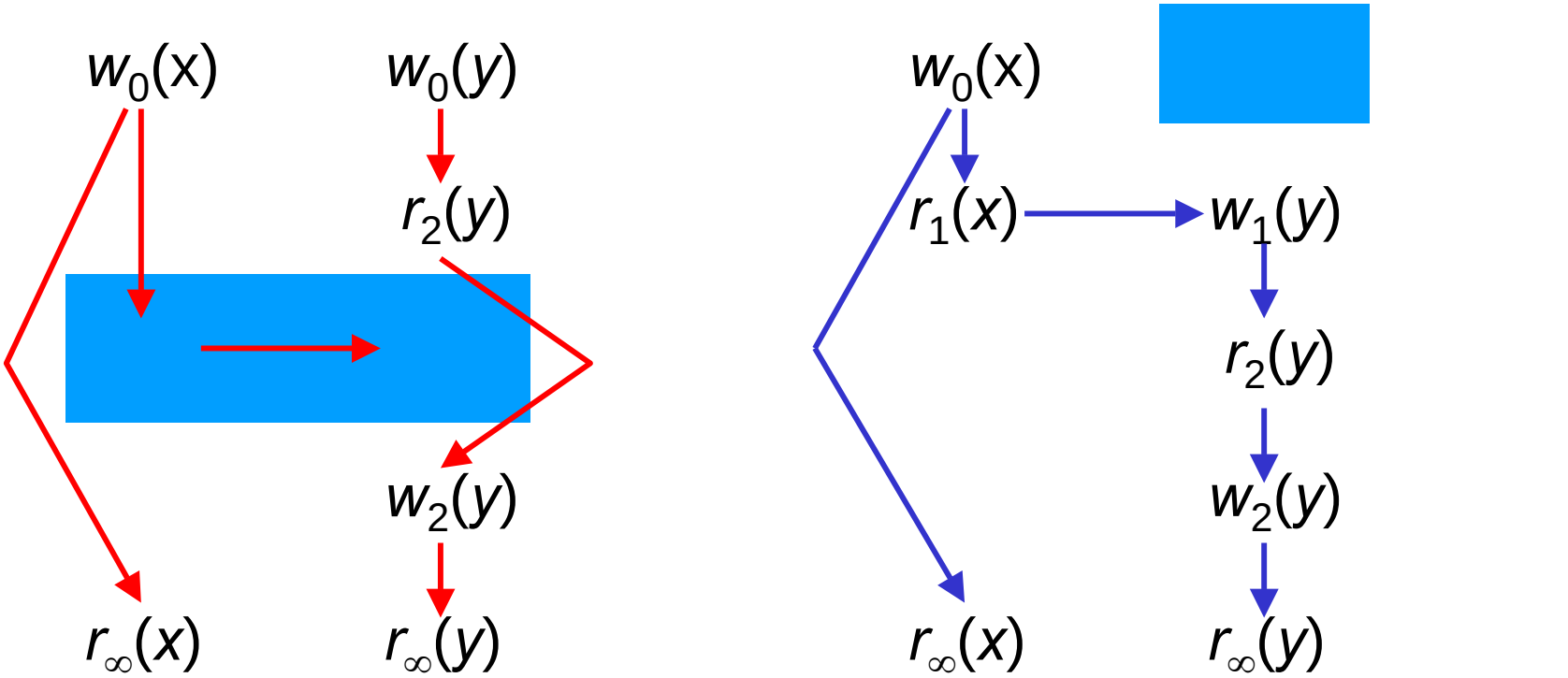
$$LRF(h) = \{(t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$$

$$LRF(h') = \{(t_0,x,t_1), (t_1,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$$

Final-State-Äquivalenz (14)

$h = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ c_1\ c_2\ r_\infty(x)\ r_\infty(y)\ c_\infty$

$h' = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ w_1(y)\ c_1\ r_2(y)\ w_2(y)\ c_2\ r_\infty(x)\ r_\infty(y)\ c_\infty$



h und *h'* nicht final-state-äquivalent

Final-State-Serialisierbarkeit

Korollar 5.57

Die Final-State-Äquivalenz zweier Schedules h und h' kann in polynomieller Zeit entschieden werden.

Definition 5.58 (Final-State-Serialisierbar)

Eine Historie h heißt **final-state-serialisierbar**, falls es eine seriellen Historie h' gibt, mit $h \equiv_F h'$.

FSR = Menge aller final-state-serialisierbaren Historien.

Test auf Final-State-Serialisierbarkeit:

Teste für alle (seriellen) Permutationen h' von h:

$h \equiv_F h'$.

Vermutung: Exponentielle Komplexität.

Vermutung gestützt auf gleichartige Komplexität der bereits eingeschränkten Sichtserialisierbarkeit.

Commit Serializability

Prefix commit closure (1)

- **Remember**: A schedule s is X-serializable if there exists a serial history h_s s.t. $\exists h_s$ serial: $CP(s) \equiv_X h_s$.
- **Problem**: The outcome of active transactions is unknown.
- **Desirable property**: Whatever the outcome (abort or commit), it should not invalidate the correctness of already committed transactions.
- **Formal criterion**: All committed prefixes of correct schedules are correct.

Prefix commit closure (2)

- **Definition 5.59**: **Prefix commit closure** class of schedules:

  $\forall s$: s X-serializable $\succ \forall s' \leq s$: $\exists h_s$ serial: $CP(s') \equiv_X h_s$.

  - **Motivation**: The **complete** past is correct! The schedule can progress without any concern for the past!

- **Definition 5.60**

  A schedule s is **commit serializable (*CM*)** if the prefix commit closure property holds for s.

- Special classes:

  CMVSR = commit view serializable

  CMCSR = commit conflict serializable

Commit Serializability (1)

Theorem 5.61 (without proof)

- Membership in class *CSR* is prefix commit closed:

  $CMCSR = CSR$

- Membership in class *VSR* (set of all view serializable histories) is generally not prefix commit closed.

  $CMVSR \subset VSR$

Commit Serializability (1)

Example 5.62

$h_{12} = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$

$CP(h_{12}) = h_{12}$.

View equivalences (solely writes!)

$h_{12} \equiv_V t_1 t_2 t_3$ and $h_{12} \equiv_V t_2 t_1 t_3$

However, prefix

$h'_{12} = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1$

is equivalent neither to $t_1 t_2$ nor to $t_2 t_1$ (in both cases the final state differs from that for h'_{12}).

$\Rightarrow h'_{12}$ is not in *CMVSR*.

CMVSR is characterized by:

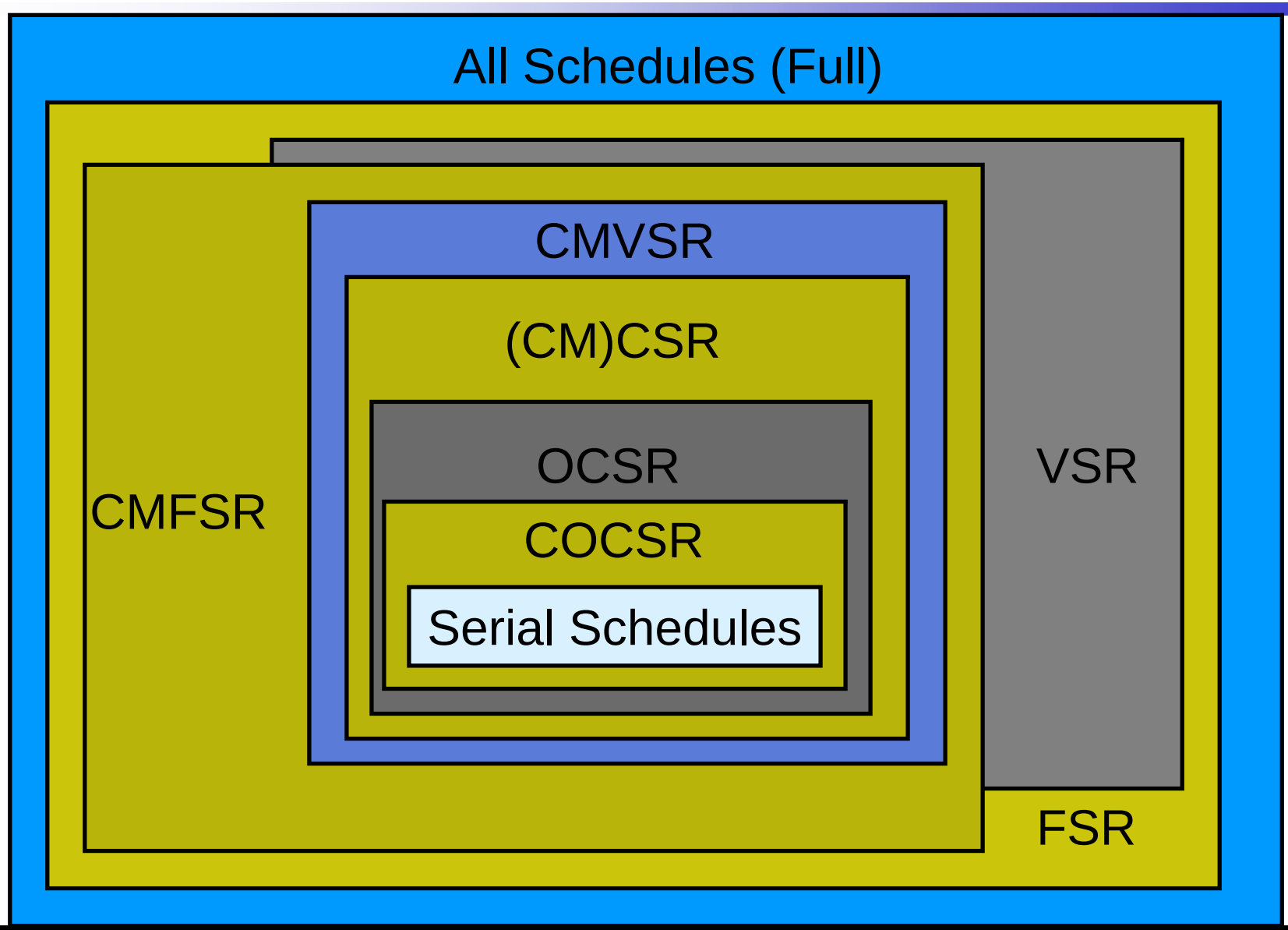
$\forall h' \leq h\ \exists h_s$ serial: $CP(h') \equiv_V h_s$ (not just: $\exists h_s$ serial: $CP(h) \equiv_V h_s$)

Commit Serializability (2)

Theorem 5.63 (without proof)

$CMCSR \subset CMVSR$

Commit Serializability (3)



DBS

Universität Karlsruhe (TH)

Chapter 6

Isolation: Synchronization and Scheduler

Agenda

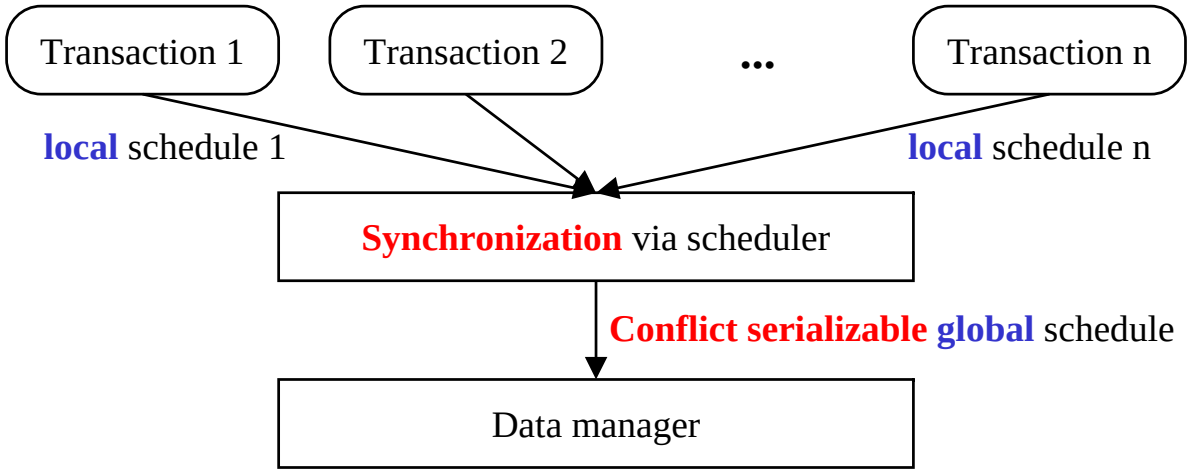
- System architecture
- Basic locking schedulers
- Multiple granularity locking schedulers
- Tree locking schedulers
- Predicate locking
- Non-locking schedulers
- Optimistic schedulers

All CSR!

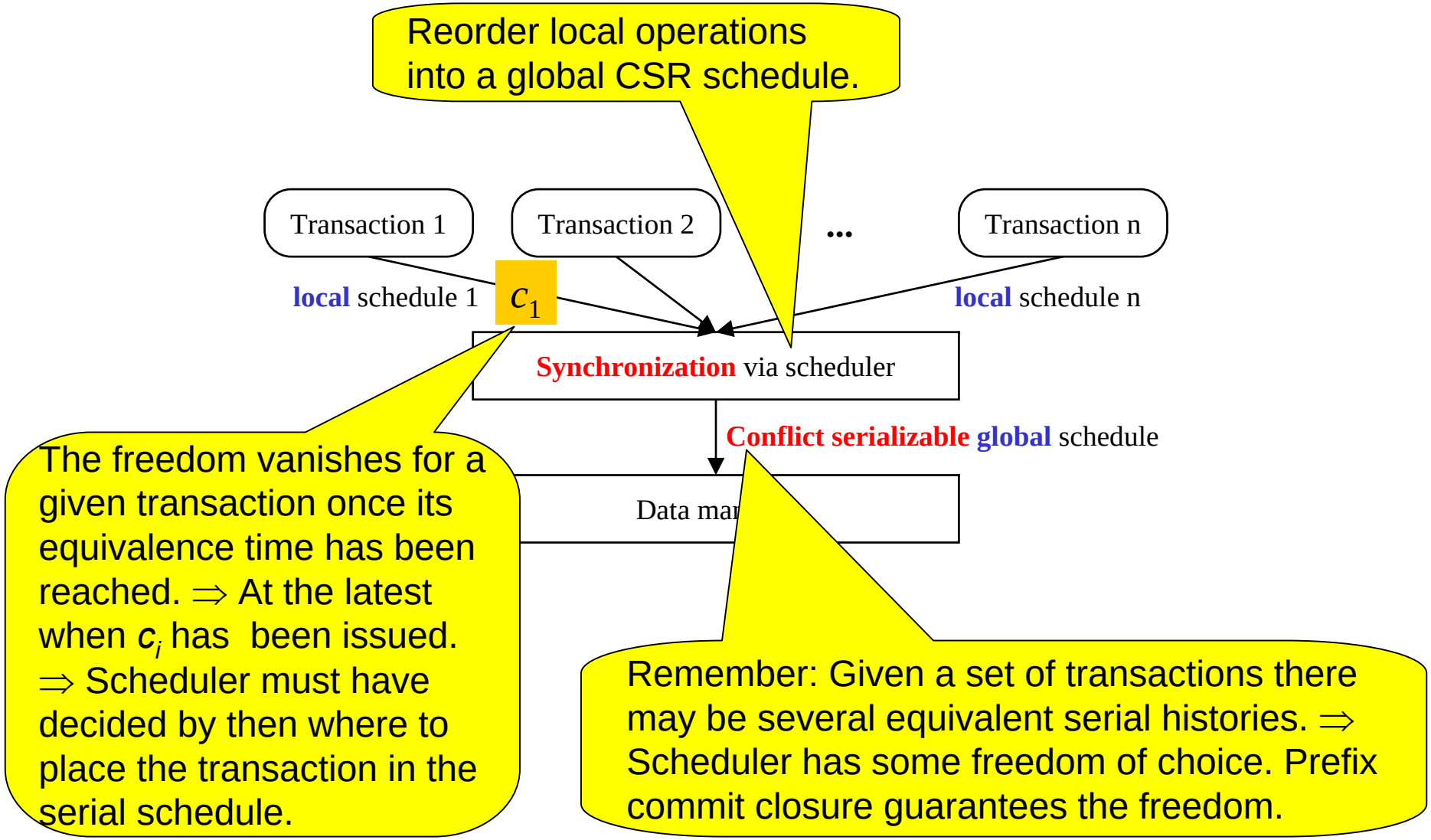
System architecture

Synchronization (1)

Simplified architecture to deal with CSR isolation

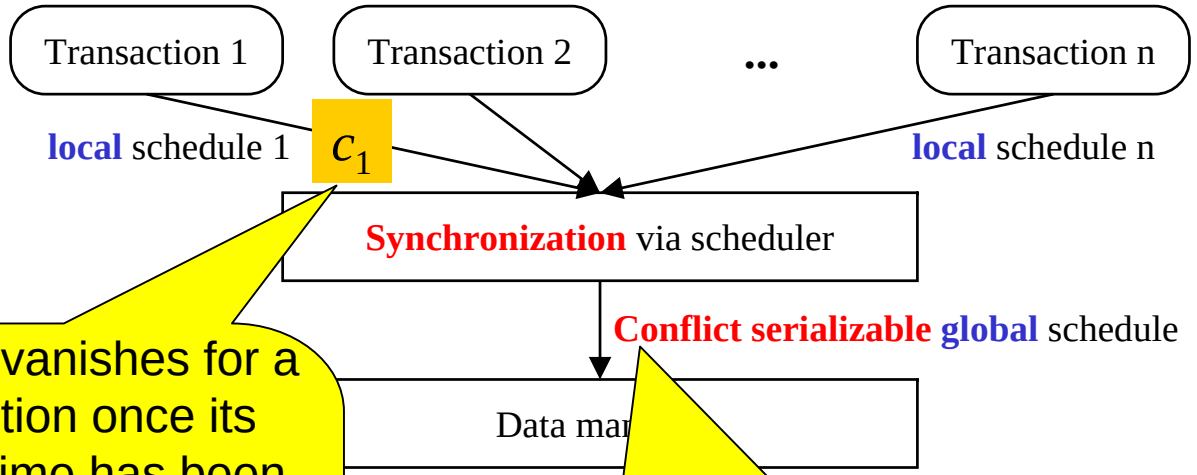


Synchronization (2)



Synchronization (3)

Choice is limited because scheduler has no control over the order of arrival of operations and conflicts. ⇒ Schedule evolves as a mixture of arrival situation and optimizer strategy of the scheduler.



The freedom vanishes for a given transaction once its equivalence time has been reached. ⇒ At the latest when c_i has been issued. ⇒ Scheduler must have decided by then where to place the transaction in the serial schedule.

Remember: Given a set of transactions there may be several equivalent serial histories. ⇒ Scheduler has some freedom of choice. Prefix commit closure guarantees the freedom.

Synchronization strategies

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Aggressive scheduler: Set equivalence time to time at the beginning ⇒ Try immediate execution of an operation ⇒ Little freedom for reordering (fewer equivalent serial schedules can be constructed).

Conservative scheduler: As long as equivalence time is open, execution of an operation can be delayed ⇒ Some freedom for reordering, but slower progress of some transactions.

Pessimistic strategies
"Deal with conflicts as they arise"

Synchronization strategies

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Optimistic scheduler: That just leaves equivalence time to be set to commit time ⇒ No decision until then ⇒ Failure possible ⇒ Limited opportunities for reordering.

Optimistic strategies
"Let's hope there are no conflicts"

CSR Safety

Definition 6.1 (CSR Safety):
For a scheduler S, **Gen(S)** denotes the set of all schedules that S can generate. A scheduler is called **CSR safe** if Gen(S) $\subseteq$ CSR.

Basic locking schedulers

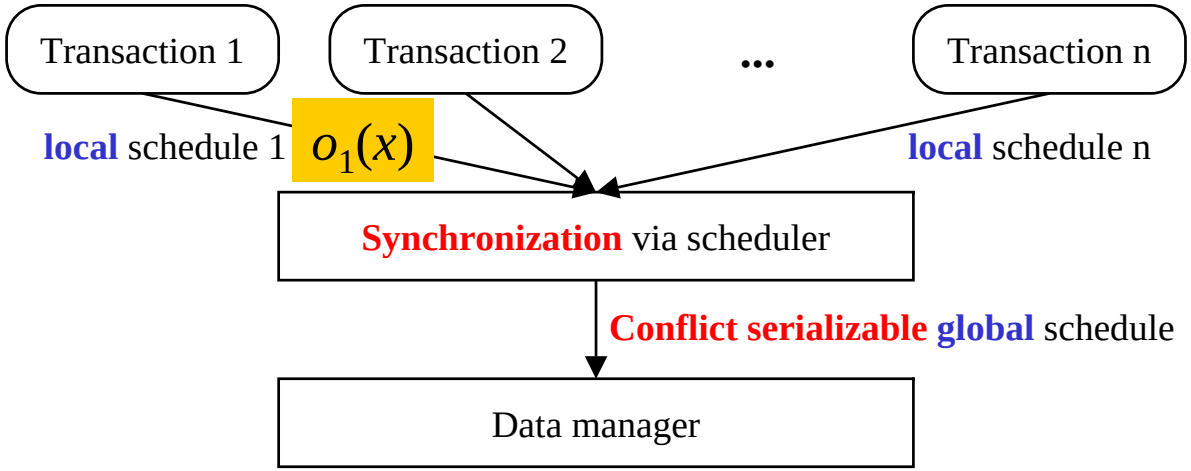
Synchronization strategy pursued

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Conservative scheduler: As long as equivalence time is open, execution of an operation can be delayed ⇒ Some freedom for reordering, but slower progress of some transactions.

Scheduler options

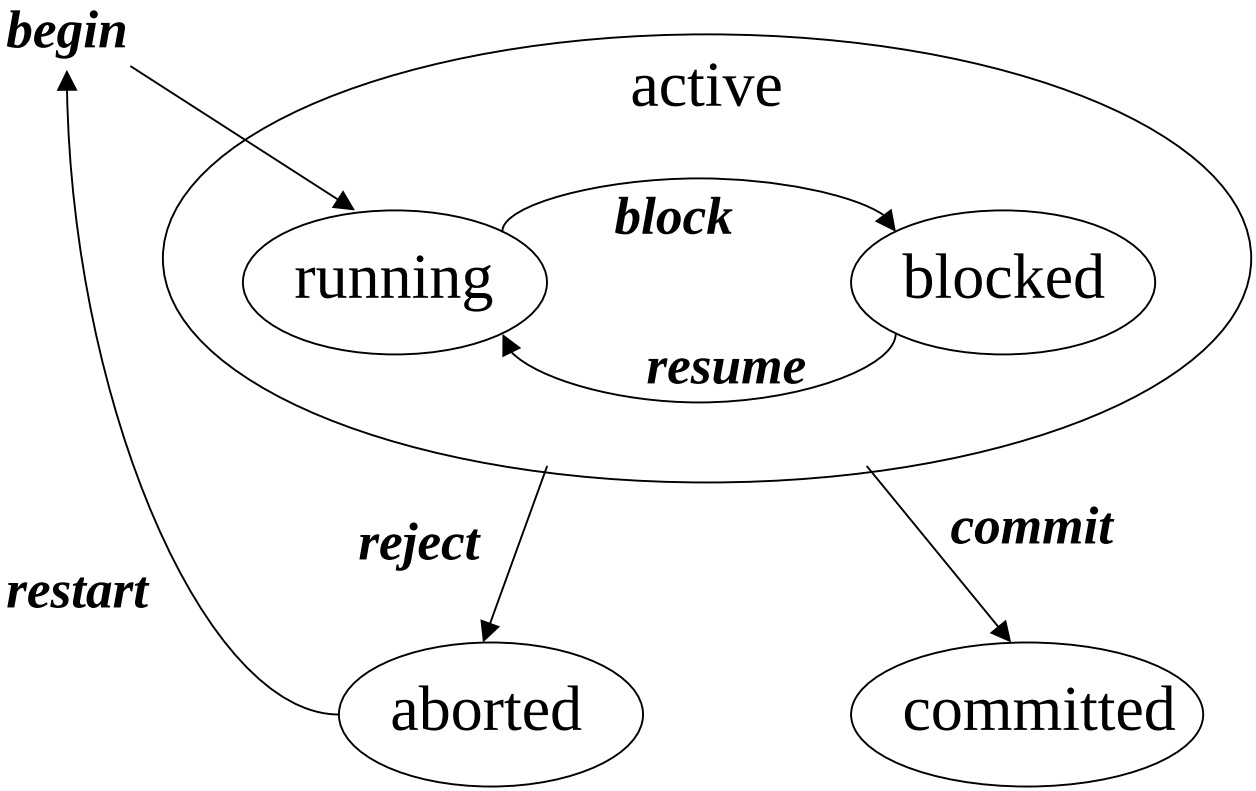


- A scheduler has for each $o_1(x)$ reaching it three options:

no conflict
- immediate execution (passing it on to the data manager),

conflict
- delay (insertion into a queue),
- reject (abort the transaction).

Scheduler Actions and Transaction States



Locks and lock operations

Recognize conflicts by locking:

- Transactions wishing to access a data element solicit a lock for the element.

  Operations:

  - $rl_i(x)$: read lock, *more precisely: lock set by a reader* **share lock**
  - $wl_i(x)$: write lock, *more precisely: lock set by a writer* **exclusive lock**
  - If p is either read or write we write $pl_i(x)$.
  - Further: $pl_i(x)$ denotes both, the lock operation and the resulting lock.

- Once the transaction no longer needs the element it unlocks the element: $pu_i(x)$.

Lock compatibility

- While an element remains locked, other transactions *may* not be able to access it and *must* then wait until the element becomes unlocked.

Definition 6.2

Two locks $pl_i(x)$ and $ql_j(y)$ **are in conflict** ($pl_i(x) \not\parallel ql_j(y)$) $\prec\!\succ$ $p \not\parallel q$ and $i \neq j$ and $x = y$.

Resulting **compatibility matrix**:

lock requestor \ lock holder	–	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	✓	✓	–
$wl_j(x)$	✓	–	–

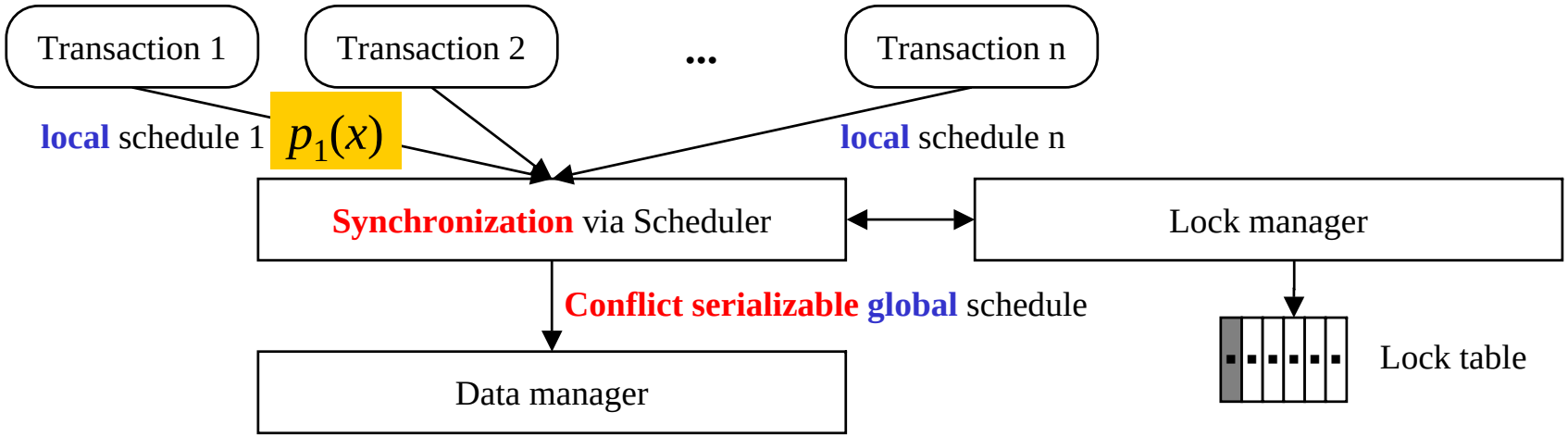
Minimal lock rules

To ensure that conflicting operations are ordered:

- **LR1**: Each data operation $p_i(x)$ must be preceded by $pl_i(x)$ and followed by $pu_i(x)$.
  Each data element must be locked sometime prior to first access, and unlocked sometime after last access.
- **LR2**: For each x and t_i there is at most one $rl_i(x)$ and at most one $wl_i(x)$.

  A transaction can lock the same data element at most once in each mode.
- **LR3**: No $ru_i(x)$ or $wu_i(x)$ is redundant.
  No unnecessary unlocks.
- **LR4**: If x is locked by both t_i and t_j, then these locks are compatible.
  No data element is locked in incompatible modes.

Lock processing



Lock table holds active locks for each data element.

1. Arrival of $p_i(x)$. Check lock table.
2. Check whether there is $ql_j(x)$.
    1. no: Set $pl_i(x)$.
    2. yes: Conflict?
        1. no: Set $pl_i(x)$.
        2. yes: transaction *i* must wait.

Extended Schedules (1)

Example 6.3:

$t_1 : r_1(x)\ w_1(y)\ c_1$

$t_2 : w_2(y)\ w_2(x)\ c_2$

Take schedule:

$r_1(x)\ w_2(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$

Extended schedule satisfying rules LR1 – LR4 :

$s_1 = rl_1(x)\ r_1(x)\ ru_1(x)\ wl_2(y)\ w_2(y)\ wl_2(x)\ w_2(x)\ wu_2(x)\ wu_2(y)$
$c_2\ wl_1(y)\ w_1(y)\ wu_1(y)\ c_1$

Extended Schedules (2)

Definition 6.4:

Let *s* be an extended schedule. Then *DT*(*s*) refers to the projection of *s* that excludes the lock operations.

Example 6.5:

$DT(s_1) = r_1(x)\ w_2(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$

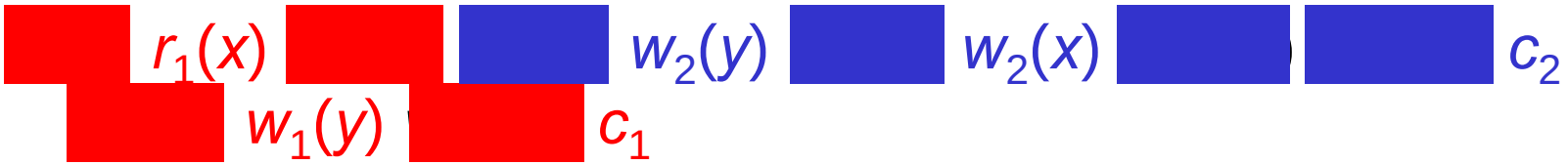
Remark:

If it is clear from the context *s* will refer to both, *s* and *DT*(*s*).

Two-Phase Locking (2PL) (1)

- Rules LR1-LR4 do not suffice to guarantee conflict serializability!

Example 6.6:

$s_1 =$ 

$DT(s_1) = r_1(x)\ w_2(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$

Analysis for s_1:

- s_1 satisfies rules LR1 – LR4.
- $r_1(x) <_{s_1} w_2(x)$ and $w_2(y) <_{s_1} w_1(y) \succ G(CP(DT(s_1)))$ is cyclic $\succ$ s_1 *is* not serializable.

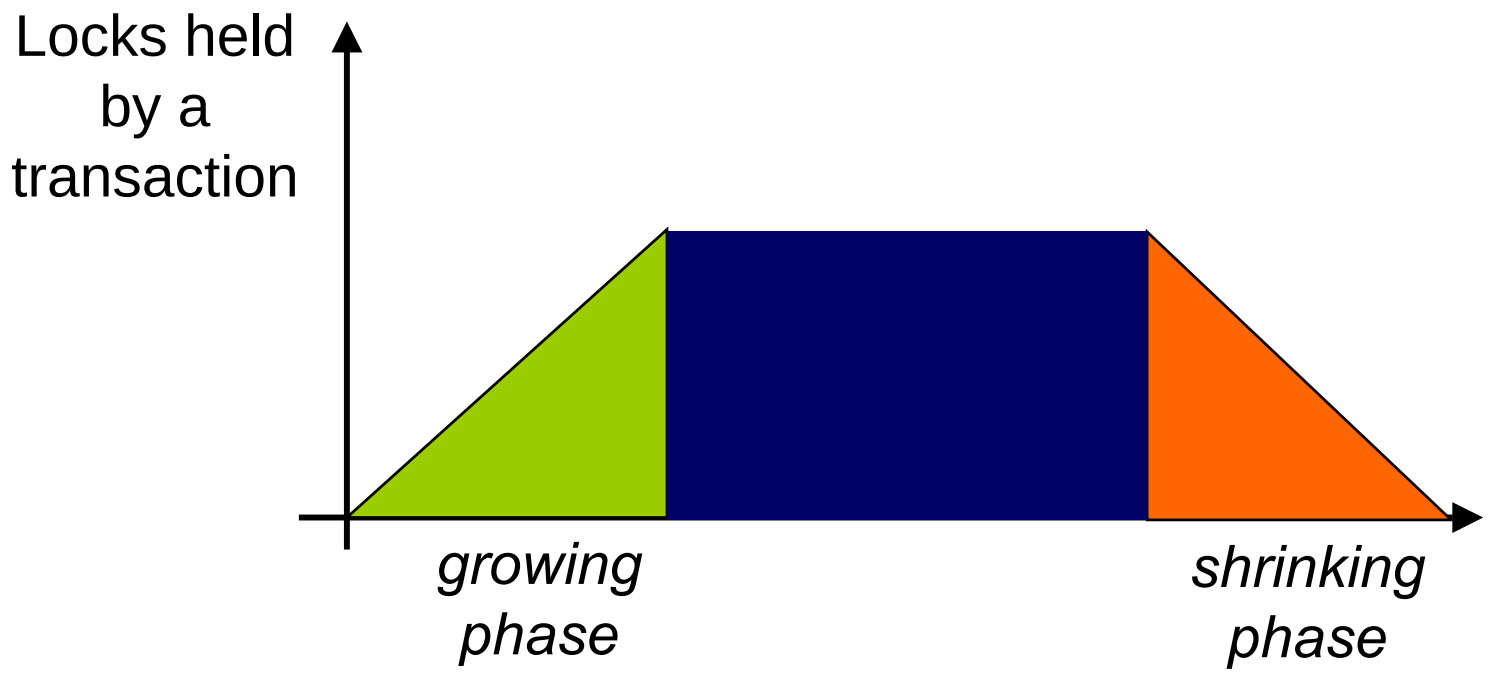
Two-Phase Locking (2PL) (2)

- We add:

Definition 6.6 (2PL):
A locking protocol is **two-phase (2PL)** if for every schedule s and every transaction $t_i \in \text{trans}(s)$ no ql_i step follows the first pu_i step ($p, q \in \{r, w\}$).

A transaction quits setting new locks as soon as it issued the first unlock operation.

Two-Phase Locking (2PL) (3)

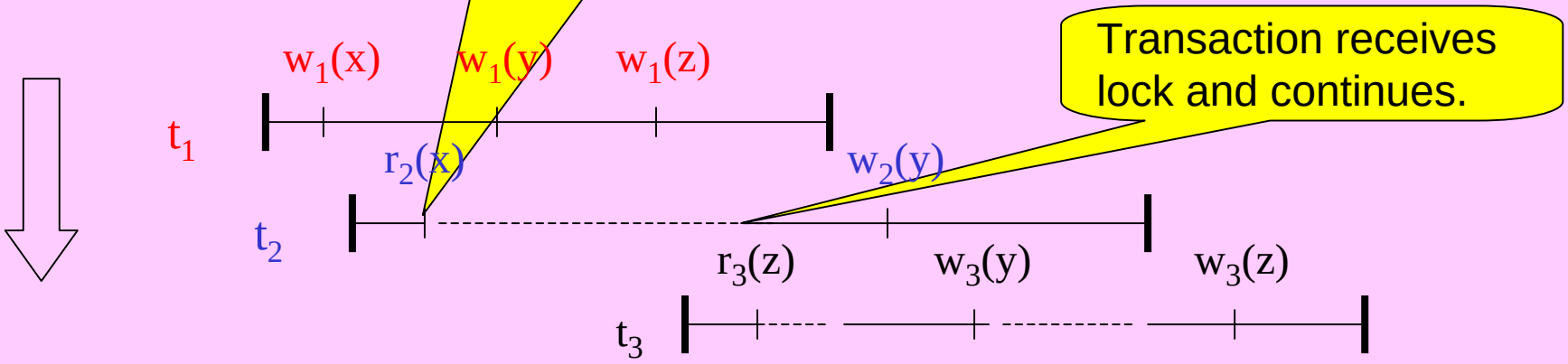


Two-Phase Locking (2PL) (4)

Definition 6.7 (2PL):
A locking protocol is [illegible] schedule s and every transaction $t_i \in$ tra[illegible] step ($p, q \in \{r, w\}$).

Transaction wishes to execute operation. It requests the lock but is blocked until element becomes accessible.

Example 6.8:
arrival order = $w_1(x)\ r_2(x)\ w_1(y)\ w_1(z)\ r_3(z)\ c_1\ w_2(y)\ w_3(y)\ c_2\ w_3(z)\ c_3$



Transaction receives lock and continues.

$wl_1(x)\ w_1(x)\ wl_1(y)\ w_1(y)\ wl_1(z)\ w_1(z)\ wu_1(x)\ rl_2(x)\ r_2(x)\ wu_1(y)\ wu_1(z)\ c_1$
$rl_3(z)\ r_3(z)\ wl_2(y)\ w_2(y)\ wu_2(y)\ ru_2(x)\ c_2$
$wl_3(y)\ w_3(y)\ wl_3(z)\ w_3(z)\ wu_3(z)\ wu_3(y)\ c_3$

Correctness proof: Principle

Goal: Prove $Gen(2PL) \subseteq CSR$.

Proof in two steps:

1. Formalize the properties of 2PL-histories.
2. Show that these properties imply conflict serializability.

Correctness proof: Step 1

Lemma 6.9 (Properties of 2PL schedules):

Let s be generated by a 2PL scheduler. Then the following holds for each transaction:

$t_i \in DT(s)$:

LR1 1. $p_i(x) \in CP(s) \succ pl_i(x), pu_i(x) \in CP(s) \wedge pl_i(x) <_s p_i(x) <_s pu_i(x)$

LR4 2. $p_i(x), q_j(x) \in CP(s), i \neq j: p_i(x) \not\parallel q_j(x) \succ pu_i(x) <_s ql_j(x) \vee qu_j(x) <_s pl_i(x)$

2PL 3. $p_i(x), q_i(y) \in CP(s) \succ pl_i(x) <_s qu_i(y)$

4. LR2, LR3 hold.

LR1: Each data operation $p_i(x)$ must be preceded by $pl_i(x)$ and followed by $pu_i(x)$.
LR2: For each x and t_i there is at most one $rl_i(x)$ and at most one $wl_i(x)$.
LR3: No $ru_i(x)$ or $wu_i(x)$ is redundant.
LR4: If x is locked by both t_i and t_j, then these locks are compatible.

Correctness proof: Step 2 (1)

Lemma 6.10 (conflict graph of 2PL schedules):

Let s be generated by a 2PL scheduler and G := $G(CP(DT(s)))$ be the conflict graph of $CP(DT(s))$. Then:

1. $(t_i, t_j) \in E(G) \succ \exists\, x, p_i(x), q_j(x)\colon\ pu_i(x) <_s ql_j(x)$
2. $(t_1, t_2, \ldots, t_n)$ is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

Correctness proof: Step 2 (2)

1. $(t_i, t_j) \in E(G) \succ \exists\, x, p_i(x), q_j(x)\colon\ pu_i(x) <_s ql_j(x)$
2. $(t_1, t_2, \ldots, t_n)$ is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

Proofs generally are by giving an interpretation to the edges of the relevant graph. (1) does it here:
Assume $(t_i, t_j) \in E(G) \succ \exists\, x, p_i(x), q_j(x)\colon p_i(x) \nparallel q_j(x) \wedge p_i(x) <_s q_j(x)$ (Def. G)
Now, by Lemma 6.9, (1): $pl_i(x), pu_i(x) \in CP(s) \succ pl_i(x) <_s p_i(x) <_s pu_i(x)$
By Lemma 6.9, (2): $i \neq j\colon p_i(x) \nparallel q_j(x) \succ pu_i(x) <_s ql_j(x) \vee qu_j(x) <_s pl_i(x)$
case $qu_j(x) <_s pl_i(x) \succ q_j(x) <_s p_i(x)$ (by 6.9(1), contradiction)
leaves case $pu_i(x) <_s ql_j(x) \succ p_i(x) <_s q_j(x)$ (by 6.9(1))

1. $p_i(x) \in CP(s) \succ pl_i(x), pu_i(x) \in CP(s) \wedge pl_i(x) <_s p_i(x) <_s pu_i(x)$.
2. $p_i(x), q_j(x) \in CP(s), i \neq j\colon p_i(x) \nparallel q_j(x) \succ pu_i(x) <_s ql_j(x) \vee qu_j(x) <_s pl_i(x)$

Correctness proof: Step 2 (2)

1. $(t_i, t_j) \in E(G) \succ \exists\, x, p_i(x), q_j(x)\colon\ pu_i(x) <_s ql_j(x)$
2. $(t_1, t_2, \ldots, t_n)$ is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

By induction on n:

$n = 2$: from Lemma 6.10 (1)

$n > 2$: Let Lemma 6.10 (2) be true for $k = n\text{-}1$:

1. $\exists p_1(x), o_k(z) \in CP(DT(s))\ \ pu_1(x) <_{CP(s)} ol_k(z)$
   With $t_k \rightarrow t_n$ and Lemma 6.10 (1):
2. $\exists o'_k(y), q_n(y) \in CP(DT(s))\ o'u_k(y) <_{CP(s)} ql_n(y)$
3. $ol_k(z) <_{CP(s)} o'u_k(y)$ (Lemma 6.9, (3))

$\succ$ $pu_1(x) <_{CP(s)} ql_n(y)$ (1, 3 and 2 and transitivity of „$<_{CP(s)}$“)

3. $p_i(x), q_i(y) \in CP(s) \succ pl_i(x) <_s qu_i(y)$.

Correctness proof: Step 2 (2)

1. $(t_i, t_j) \in E(G) \succ \exists\, x, p_i(x), q_j(x)\colon\ pu_i(x) <_s ql_j(x)$
2. $(t_1, t_2, \ldots, t_n)$ is path in $G \succ pu_1(x) <_s ql_n(y)$
3. G is acyclic

Proof by contradiction:

Suppose $G(CP(DT(s)))$ contains cycle $t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$ where $n > 1$.

$\exists p_1(x), q_1(y) \in CP(DT(s))\ \ pu_1(x) <_{CP(s)} ql_1(y)$ (with Lemma 6.10 (2))

Contradicts Lemma 6.9 (3).

2PL rule is essential!

3. $p_i(x), q_i(y) \in CP(s) \succ pl_i(x) <_s qu_i(y)$.

Correctness proof: Step 2 (3)

Theorem 6.11 (safe)

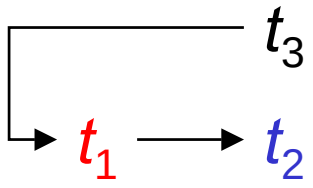
$Gen(2PL) \subset CSR$

Proof:

$\subseteq$ follows directly from Lemma 6.10

Proper subset:

$s = w_1(x)\ r_2(x)\ c_2\ r_3(y)\ c_3\ w_1(y)\ c_1$



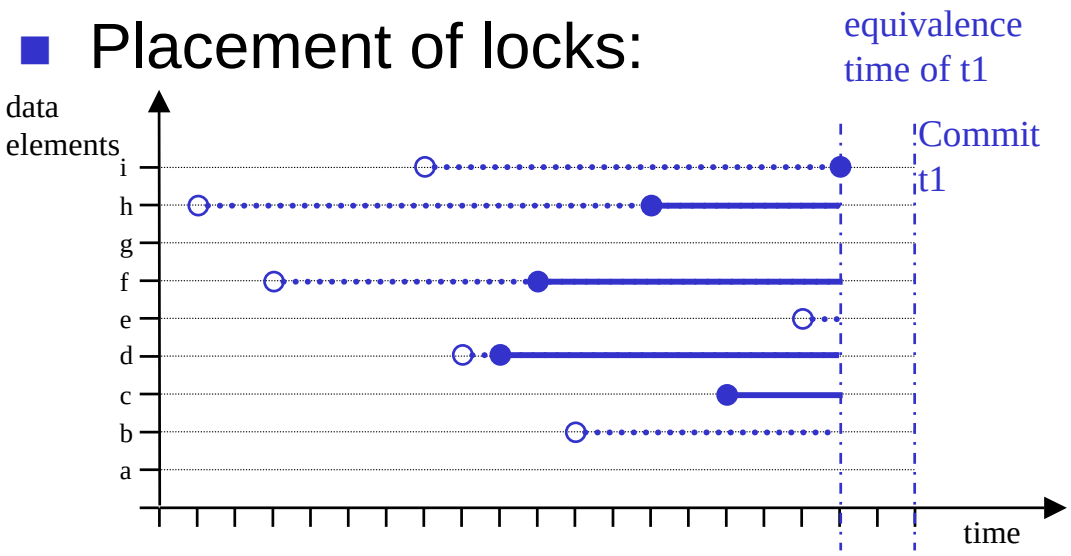
Two-Phase Locking (2PL) (5)

- **Remark**:

  LR2-LR4 permit that a transaction may issue $w_i(x)$ after $r_i(x)$ and, hence, **escalate** a *rl* lock to a *wl* lock. The correctness proof for 2PL is not affected!

2PL: Graphical illustration

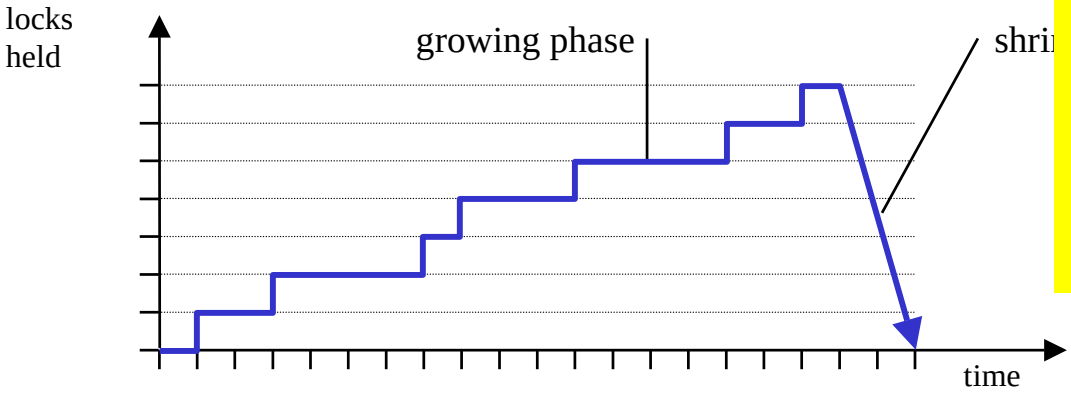
- Placement of locks:



Legend:

- Read by t1
- Write by t1
- Read lock by t1
- Write lock by t1

- Number of locks held:



Interpretation:

2PL schedule is conflict equivalent to a serial schedule where all transactions are ordered wrt their final time of *maximum lock ownership*.

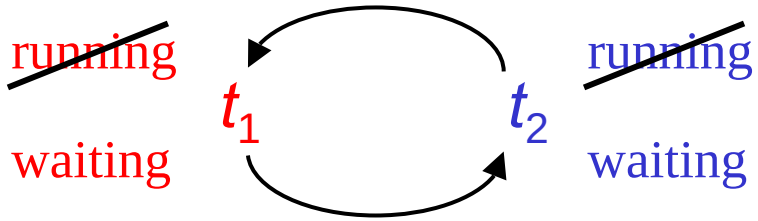
Deadlocks (1)

What happens if a cycle threatens to develop?

Example 6.12

$s_1 =$ [redacted] $wl_2(x)\ w_2(x)\ wu_2(x)\ wu_2(y)\ c_2$

$w_1(y)\ ru_1(x)\ wu_1(y)\ c_1$



Schedule cannot be continued: Both t_1 and t_2 must wait.

Analysis: Attempt to rearrange serial order of transactions.
Fortunately, deadlocks not on every rearrangement!

Deadlocks (2)

2PL does not avoid deadlocks!

Frequent reason for deadlocks: **lock escalation** (also: **lock conversion**) *rl* to *wl*.

Example 6.13

t_1: $rl_1(x)\ r_1(x)\ wl_1(x)\ w_1(x)\ wu_1(x)\ c_1$

t_2: $rl_2(x)\ r_2(x)\ wl_2(x)\ w_2(x)\ wu_2(x)\ c_2$

s_1: $rl_1(x)\ r_1(x)\ rl_2(x)\ r_2(x)\ wl_2(x)\ wl_1(x)$ **X**

Deadlock detection

- Via **timeout**:
  - Each lock request is assigned a maximum wait time.
  - After expiration a deadlock is assumed.
  - Simple implementation, but if wait time is badly chosen erroneous abort or belated detection.
- Via **waits-for graph** $WFG = (N,E)$ where

  $N = active(s)$

  $E = \{(t_i, t_j) \mid t_i \text{ waits for unlock by } t_j\}$.
  - Deadlock if WFG has a cycle.
  - More costly management than for timeouts, but more precise outcome.

Deadlock Resolution

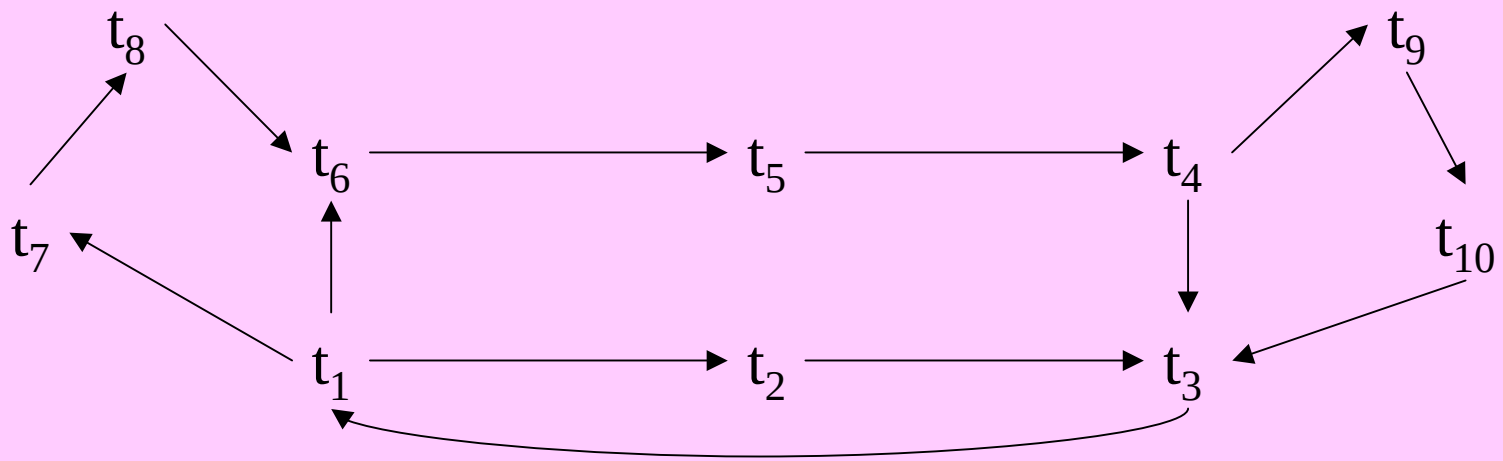
Choose a transaction on a WFG cycle as a **deadlock victim** and abort this transaction, and repeat until no more cycles.

Possible victim selection strategies:

1. Last blocked
2. Random
3. Youngest
4. Minimum locks
5. Minimum work
6. Most cycles
7. Most edges

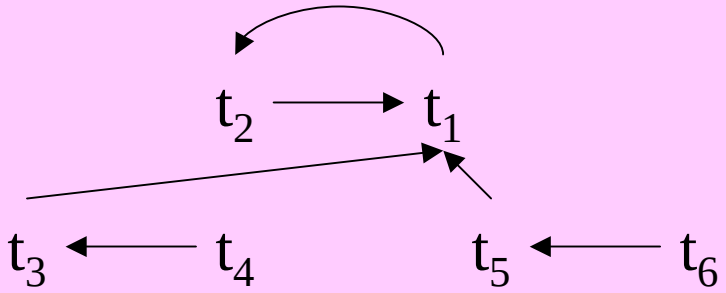
Illustration of Victim Selection Strategies

Example 6.14:



Most-cycles strategy would select t_1 (or t_3) to break all 5 cycles.

Example 6.15:



Most-edges strategy would select t_1 to remove 4 edges.

Deadlock Resolution: Problem

- All aforementioned resolution strategies can result in livelock (or starvation):

  The same transaction is re-selected for abort and restarted (thus again ending up in a cycle and being victimized), etc.

Deadlock Prevention

How about avoiding cycles altogether? ⇒ **Restrict lock waits** to ensure **acyclic WFG** at all times.

Reasonable deadlock prevention strategies:

1. **Wait-die:**
   upon t_i blocked by t_j:
   if t_i started before t_j then wait else abort t_i
2. **Wound-wait:**
   upon t_i blocked by t_j:
   if t_i started before t_j then abort t_j else wait
3. **Immediate restart:**
   upon t_i blocked by t_j: abort t_i
4. **Running priority:**
   upon t_i blocked by t_j:
   if t_j is itself blocked then abort t_j else wait

Transactions can only be blocked by later ones.

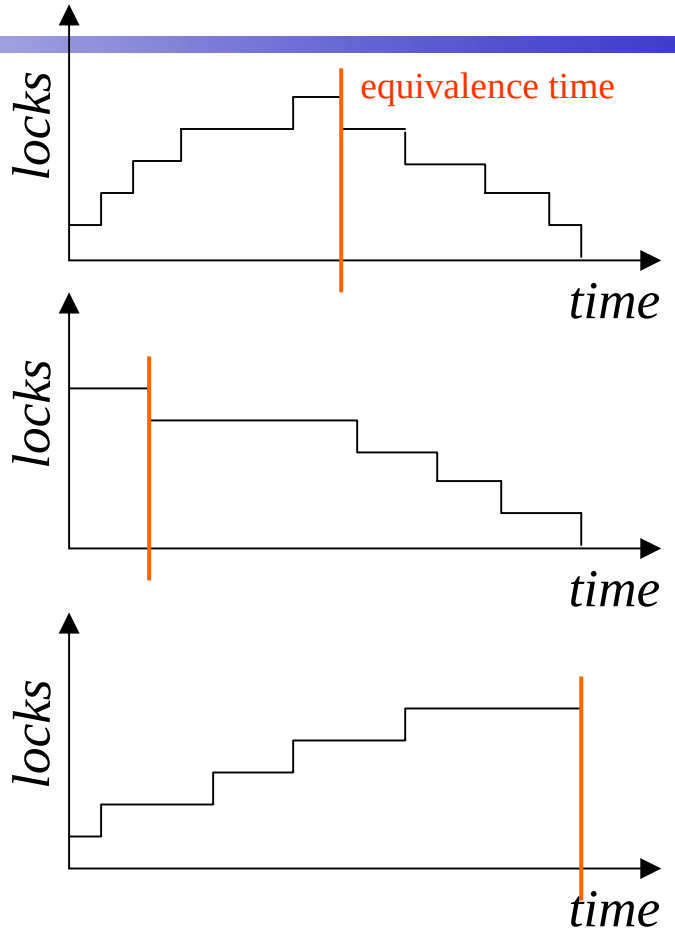
Transactions can only be blocked by earlier ones.

No blocking whatsoever.

Blocked transactions must not impede active ones.

Variants of 2PL

general 2PL



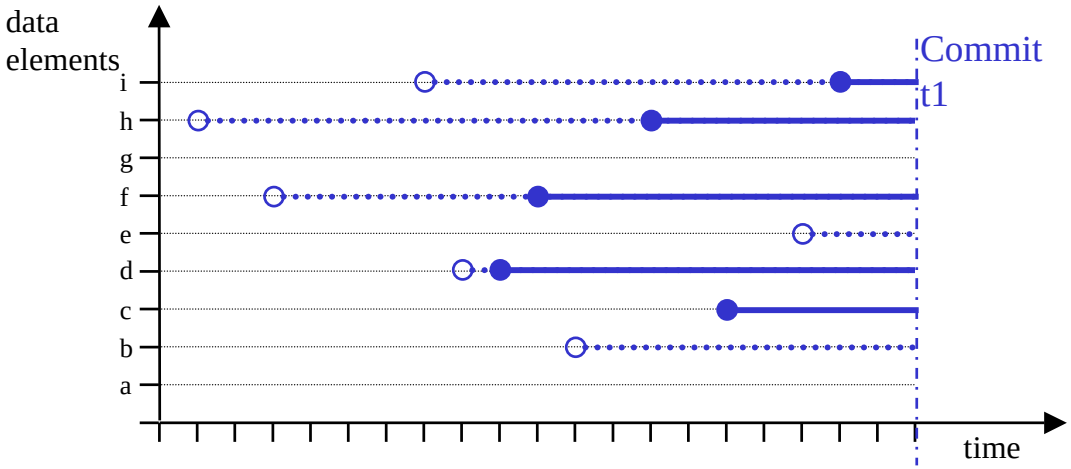
Definition 6.16 (Conservative 2PL):
Under **static** or **conservative 2PL (C2PL)** each transaction acquires all its locks before the first data operation (**preclaiming**).

Definition 6.17 (Strict 2PL):
Under **strict 2PL (S2PL)** each transaction holds all its write locks until the transaction terminates.

Definition 6.18 (Strong 2PL):
Under **strong 2PL (SS2PL)** each transaction holds all its locks (i.e., both r and w) until the transaction terminates.

SS2PL: Graphical illustration

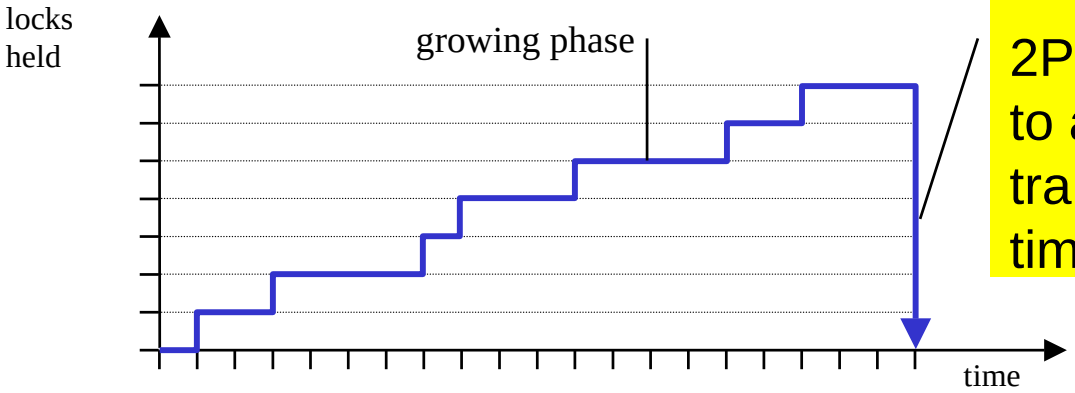
- Placement of locks:



Legend:

- Read by t1
- Write by t1
- Read lock by t1
- Write lock by t1

- Number of locks held:



Interpretation:

2PL schedule is conflict equivalent to a serial schedule where all transactions are ordered wrt their time of *commit*.

Rigorous Schedules (1)

- SS2PL generates rigorous schedules.
  - *s* is **rigorous** if for all transactions t_i, t_j in $CP(s)$ where $i \neq j$: if $o_i(x)$ before $o_j(x)$ in *s* and $o_i(x)$ in conflict with $o_j(x)$ then c_i before $o_j(x)$.

History *h* is **commit-ordered** if:

$\forall\ t_i, t_j \in h,\ i \neq j,\ p \in op_i,\ q \in op_j$: $(p,q) \in conf(h) \succ c_i <_h c_j$

Theorem 6.19:
$Gen(SS2PL) \subset$ COCSR

Theorem 6.20:
$Gen(SS2PL) \subset Gen(S2PL) \subset Gen(2PL)$

Rigorous Schedules (2)

Remark 6.21:

- SS2PL is the protocol offered by most commercial database systems.
  - COCSR plays a central role wrt recovery.
  - Transactions can leave locking/unlocking entirely to the scheduler.

Example

Example 6.22

arrival order = $w_1(x)\ r_2(x)\ w_1(y)\ w_1(z)\ r_3(z)\ c_1\ w_2(y)\ w_3(x)\ c_2\ w_3(z)\ c_3$

2PL $w_1(x)\ w_1(y)\ w_1(z)\ r_2(x)\ r_3(z)\ c_1\ w_2(y)\ w_3(x)\ c_2\ w_3(z)\ c_3$

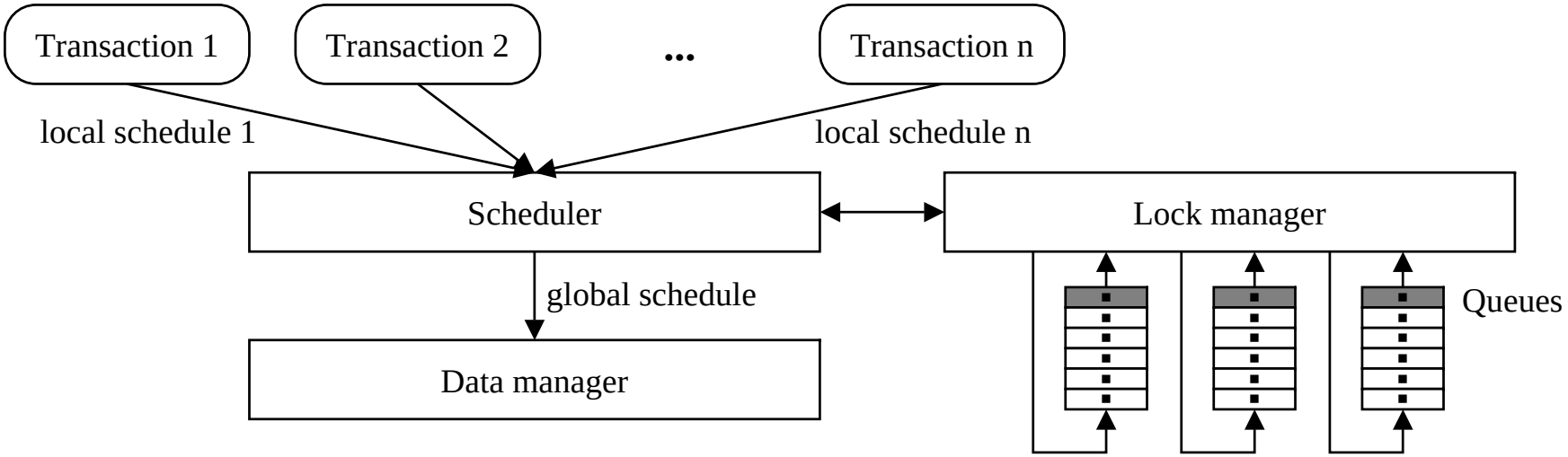
C2PL $w_1(x)\ w_1(y)\ r_2(x)\ w_1(z)\ c_1\ w_2(y)\ r_3(z)\ w_3(x)\ c_2\ w_3(z)\ c_3$

S2PL $w_1(x)\ w_1(y)\ w_1(z)\ c_1\ r_2(x)\ r_3(z)\ w_2(y)\ w_3(x)\ c_2\ w_3(z)\ c_3$

SS2PL $w_1(x)\ w_1(y)\ w_1(z)\ c_1\ r_2(x)\ r_3(z)\ w_2(y)\ c_2\ w_3(x)\ w_3(z)\ c_3$

Implementation of SS2PL (1)

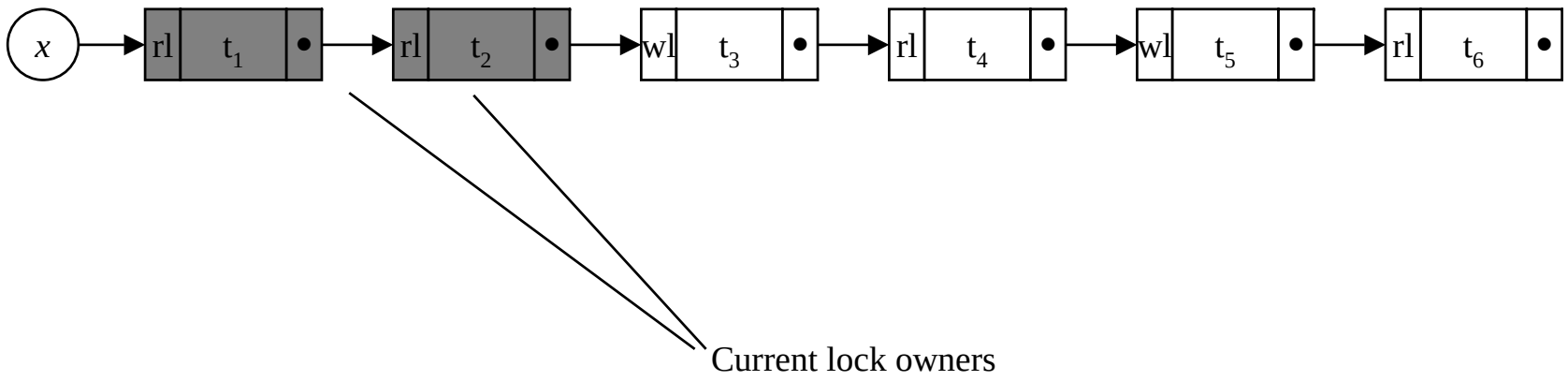
- Refinement of the earlier architecture:



Implementation of SS2PL (2)

Assignment of locks by the lock manager:

- Administer (at least conceptually) a queue for each data element *x*, where the first queue elements refer to the active owners and the remaining elements to the waiting transactions. If there are no locks on *x* the queue is empty.



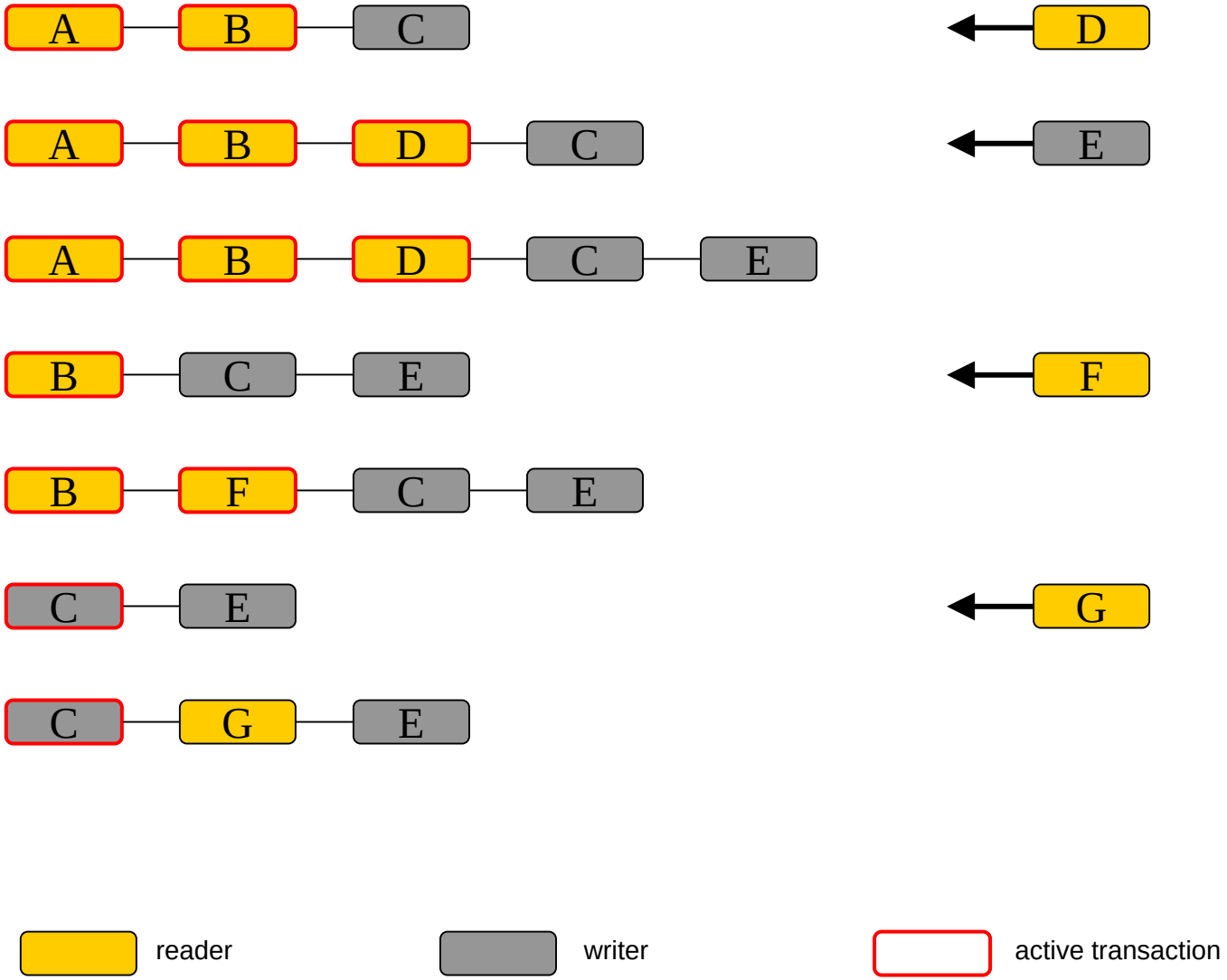
Implementation of SS2PL (3)

- (Optimizing) strategies for satisfying a lock request on *x*:
  - **Fair**: Always insert new element at the end of the queue.
  - **Favoring readers**:
    - A new share lock is inserted in front of all waiting writers. If there is no active writer, no further wait.
    - A new exclusive lock is inserted at the end of the queue.
  - **Favoring writers**:
    - A new exclusive lock is inserted right behind the last writer or if there is none, right behind the last active reader.
    - A new share lock is inserted at the end of the queue.

Implementation of SS2PL (4)

- **Synchronization algorithm favoring readers**:
  - Wait for next arriving operation *op*.
  - If *op* = $r_i(x)$ request read (share) lock. Check with lock manager whether another transaction already owns write (exclusive) lock on *x*. If not grant lock, else wait. Once lock is available execute *op*.
  - If *op* = $w_i(x)$ request write lock. Check with lock manager whether no transaction already owns a lock on *x* or transaction *i* already owns share lock on *x*. If so grant lock, else wait. Once lock is available execute *op*.
  - If *op* = c_i call on the database manager to make persistent all database changes by transaction *i*, then remove all locks held by transaction *i*.
  - If *op* = a_i call on the database manager to roll back all database changes by transaction *i*, then remove all locks held by transaction *i*.

Implementation of SS2PL (5)



Multiple Granularity Locking Schedulers

Phantome (1)

Hr. Müller fragt im Rahmen einer Inventur die Bestände der einzelnen Sorten bei der Relation Weißweine ab und errechnet den Gesamtbestand. Hr. Schmidt erweitert unterdessen das Sortiment um 10 Kisten Grauburgunder.

Weißweine	
Artikel	Bestand
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller-Thurgau	100

Bestände	
Sorte	Bestand
Weißweine	159
Rotweine	200

Phantome (2)

Weißweine	
Artikel	Bestand
Gutedel	12
Riesling	34
Silvaner	2
Weißburgunder	11
Müller-Thurgau	100

t_M

zur Zeit 10: t_S fügt ein: (Grauburgunder, 10)

zur Zeit 9: Position von t_M

zur Zeit 11: t_S ändert: (Weißweine, 169)

zur Zeit 12: Inkonsistente Sicht von t_M : 159≠169

Bestände	
Sorte	Bestand
Weißweine	159
Rotweine	200

s: $rl_M(Gu)$ $r_M(Gu)$ $rl_M(Ri)$ $r_M(Ri)$ $rl_M(Si)$ $r_M(Si)$ $rl_M(Wb)$ $r_M(Wb)$ $wl_S(Gb)$ $w_S(Gb)$ $wl_S(Ww)$ $w_S(Ww)$ $wu_S(Gb)$ $wu_S(Ww)$ c_S $rl_M(Ww)$ $r_M(Ww)$ …..

Phantome (3)

s erfüllt 2PL!

s: $rl_M(Gu)\ r_M(Gu)\ rl_M(Ri)\ r_M(Ri)\ rl_M(Si)\ r_M(Si)\ rl_M(Wb)\ r_M(Wb)\ wl_S(Gb)\ w_S(Gb)\ wl_S(Ww)\ w_S(Ww)\ wu_S(Gb)\ wu_S(Ww)\ c_S\ rl_M(Ww)\ r_M(Ww)$ …..

$DT(s)$: $r_M(Gu)\ r_M(Ri)\ r_M(Si)\ r_M(Wb)\ w_S(Gb)\ w_S(Ww)\ c_S\ r_M(Ww)$ …..

$t_S \longrightarrow t_M$

$DT(s) \in CSR$: Dabei scheint non-repeatable read vorzuliegen! Aber: Alle gelesenen Zustände sind zum gleichen Zeitpunkt c_S gültig.

Phantome (4)

- Worst-case Annahme „in einer Transaktion beeinflussen alle vorangehenden Lesen ein Schreiben“ reicht nicht aus!
- Hier: Auch zwischen Lesen einer Transaktion kann es Zusammenhänge geben.
  - Diese sind Teil der Transaktionssemantik (hier das zwischenzeitliche Aufsummieren) und werden durch das Transaktionsmodell nicht erfasst.
  - Wird zum Problem, wenn der Zusammenhang verschiedene Gültigkeitszeitpunkte überbrückt.
- **Phantom**: „Geisterhaftes“, für eine fremde Transaktion nicht sichtbares Datenelement als Verursacher eines fehlerhaften Zusammenhanges.

Phantome (5)

Analyse aus Sicht 2PL: Ein nicht vorhandenes Element kann nicht mit einer Sperre belegt werden.

Abhilfe: Setzen einer Sperre, die das neue Element automatisch mit abdeckt.

Korngröße der Sperre:

- Im Beispiel ist Korngröße das Tupel. Man wähle als nächstgrößeres Korn die Relation.
- Wir betrachten nachfolgend ein Verfahren, das die Wahl der Körnigkeit systematisiert.

Multi-Korngrößen-Sperren (1)

Multiple Granularity Locking (MGL):

Ausnutzen, dass in einer Datenbasis von der Struktur her verschiedene Granulate (Korngrößen) erkennbar sind.
Beispiele:

Physische Ebene	**Semantische Ebene**
(ganze) Datenbasis	Objektbasis
Segment, Area, DB-Space	Schema
Datei	(Objekttyp-)Extension, Relation
Seite	Objekt, Tupel
Record	Attribut

Multi-Korngrößen-Sperren (2)

Nutzeffekte:

1. Kontrolle der Phantome.
2. Kontrolle des Nebenläufigkeitsgrades und des Verwaltungsaufwandes:
   - Großes Korn $\succ$ wenige Sperren $\succ$ geringe Nebenläufigkeit.
   - Kleines Korn $\succ$ viele Sperren $\succ$ hohe Nebenläufigkeit.

Multi-Korngrößen-Sperren (3)

- **Beispiele**:
  - t_1: Gehaltserhöhung aller Angestellten nach Tarifrunde (Schreiben in alle Angestellten-Objekte)
  - t_2: Berechnen der Gesamtpersonalkosten (Lesen aller Angestellten-Objekte)
  - t_3: Auswertung eines Wettbewerbs, bei dem die besten 3 Vertreter eine Prämie bekommen (Lesen aller Angestellten-Objekte, Schreiben von 3 Objekten)
- All diese längeren Transaktionen haben eine größere Chance durchzukommen, wenn sie nur eine Sperre auf der Menge aller Angestellten setzen.
- Dann aber: Nur serieller Ablauf möglich.

Absichtssperren (1)

Arten von Sperren:

rl: share lock: Sperrt Korn zum Lesen

wl: exclusive lock: Sperrt Korn zum Schreiben.

irl: intention share lock: Zeigt an, dass (ein) Element(e) innerhalb dieses Korns gelesen werden soll(en). Die *share*-Sperren für die Elemente müssen noch einzeln angefordert werden.

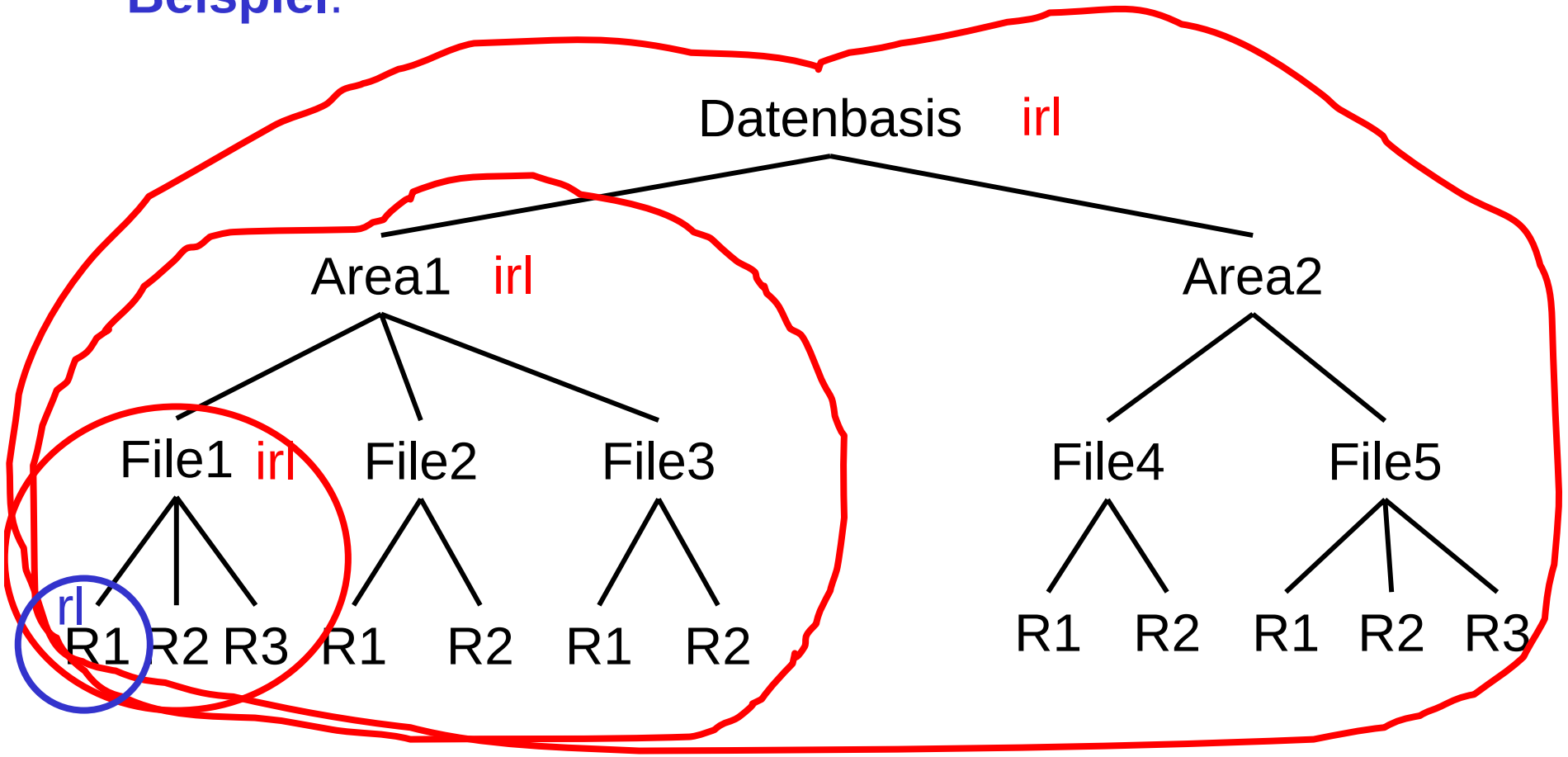
iwl: intention exclusive lock: Analog zum intention share lock.

riwl: share intention exclusive lock: Zeigt an, dass alle Elemente innerhalb dieses Korns gelesen werden und einige geschrieben werden sollen. Die *exclusive*-Sperren zum Schreiben müssen noch explizit angefordert werden.

Absichtssperren (2)

Eine Sperre auf ein Korn X sperrt *implizit* alle unter X befindlichen Körner in gleicher Weise.

Beispiel:



MGL-2-Phasen-Sperrprotokoll (1)

Basis-2PL-Protokoll: **Anpassung**

1. Eintreffen von $p_i(x)$: Falls $p_i(x)$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i(x)$ an t_i vergeben und $p_i(x)$ ausgeführt. Sonst wird $p_i(x)$ verzögert bis die entsprechende Sperre freigegeben ist.
2. Für [illegible] von t_i bearbeitete x kommt genau ein Schritt $rl_i(x)$ bzw. $wl_i(x)$ [illegible] derselben Art werden höchstens einmal

- Falls eine *r*- oder *ir*-Sperre benötigt wird für ein Korn, müssen alle Vorgänger mit *ir* oder *iw* gesperrt sein.
- Falls eine *w*- oder *iw*-Sperre benötigt wird für ein Korn, müssen alle Vorgänger mit *riw* oder *iw* gesperrt sein.
- Falls t_i ein Datenelement lesen (schreiben) will, so benötigt es eine *r*- oder *riw*- (*w*-) Sperre für das Datenelement oder einen Vorgänger.

MGL-2-Phasen-Sperrprotokoll (2)

Basis-2PL-Protokoll: **Anpassung**

1. Eintreffen von $p_i(x)$: Falls $p_i(x)$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i(x)$ an t_i vergeben und $p_i(x)$ ausg
   Sper
2. Für jedes vo x kommt genau ein Schritt $r_i(x)$ bzw. $wl_i(x)$ vor derselben Art werden höchstens einmal ge t).
3. Eine Sperre $pl_i(x)$ wird nicht eher freigegeben bis nicht wenigstens $p_i(x)$ ausgeführt wurde. Alle Sperren werden bzw. sind zu Transaktionsende freigegeben.
4. Nachdem für eine Transaktion mindestens eine Sperre freigegeben wurde, dürfen keine Sperren mehr an diese Transaktion vergeben werden.

Eine Absichts-Sperre kann nur freigegeben werden, falls keine Sperre mehr für Nachfolger gehalten wird.

MGL-2-Phasen-Sperrprotokoll (3)

Basis-2PL-Protokoll: **Anpassung**

1. Eintreffen von $p_i(x)$: Falls $p_i(x)$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i(x)$ an t_i vergeben und $p_i(x)$ ausgeführt. Sonst wird $p_i(x)$ verzögert bis die entsprechende Sperre freigegeben ist.
2. Für jedes von t_i bearbeitete x kommt genau ein Schritt $rl_i(x)$ bzw. $wl_i(x)$ vor (S
   gesetzt).
3. Eine Sperre
   wenigstens
   sind zu Tra
4. Nachdem fü
   freigegeben
   Transaktion

gehalten→ angefordert↓	rl	wl	irl	iwl	riwl
rl	yes	no	yes	no	no
wl	no	no	no	no	no
irl	yes	no	yes	yes	yes
iwl	no	no	yes	yes	no
riwl	no	no	yes	no	no

MGL-2-Phasen-Sperrprotokoll (4)

Definition 6.23

MGL-2PL = Basis-2PL + folgende Ergänzungen:

1. Falls eine *r*- oder *ir*-Sperre benötigt wird für ein Korn, müssen alle Vorgänger *ir* oder *iw* gesperrt sein.
2. Falls eine *w*- oder *iw*-Sperre benötigt wird für ein Korn, müssen alle Vorgänger mit *riw* oder *iw* gesperrt sein.
3. Falls t_i ein Datenelement lesen (schreiben) will, so benötigt es eine *r*- oder *riw*- (*w*-) Sperre für das Datenelement oder einen Vorgänger.
4. Eine Absichts-Sperre kann nur freigegeben werden, falls keine Sperre mehr für Nachfolger gehalten wird.

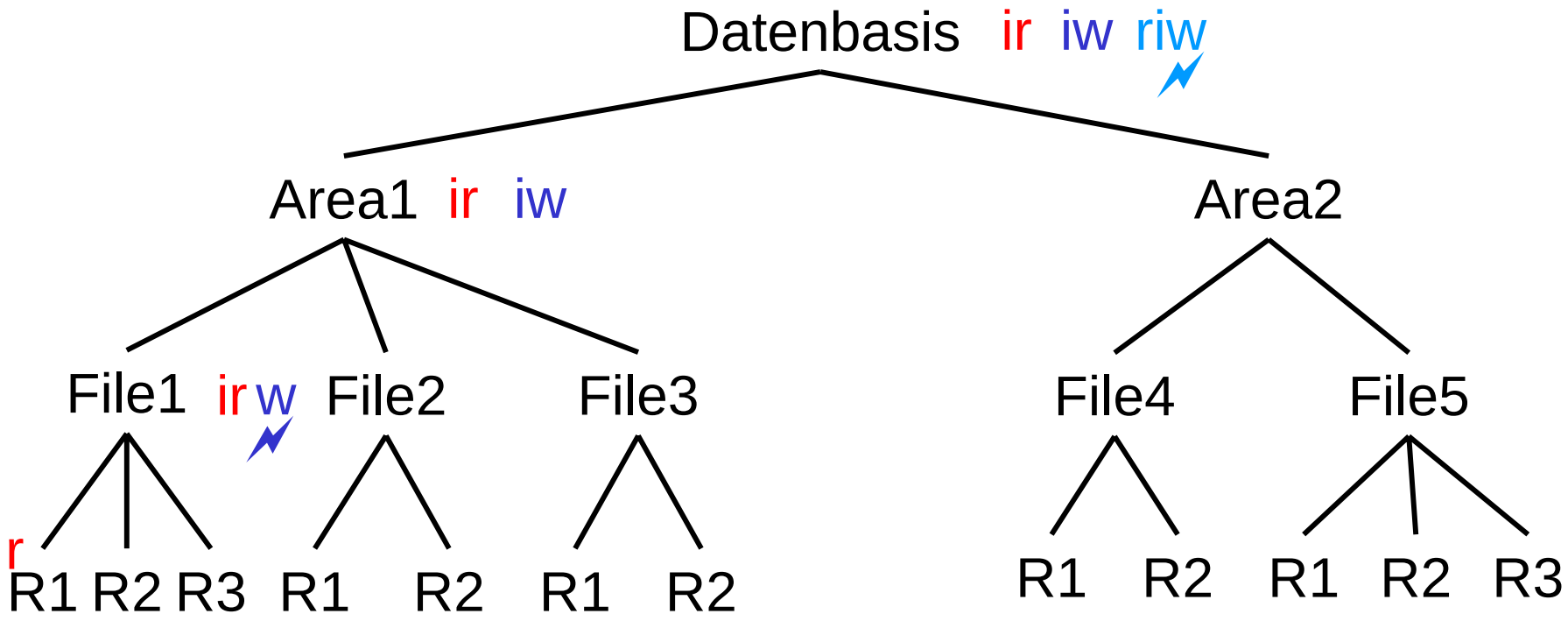
MGL-2-Phasen-Sperrprotokoll (5)

Umsetzung für einzelne Transaktion:

- Eine Transaktion setzt, von der Wurzel her kommend, *ir*-Sperren bzw. *iw*-Sperren so lange, bis sie eine *r*- bzw. *w*-Sperre setzt. Sie setzt *riw*-Sperren so lange, bis sie eine *w*-Sperre setzt.
- Eine Transaktion löst ihre Sperren von den *r*- bzw. *w*-Sperren her kommend sukzessive bis zur Wurzel.

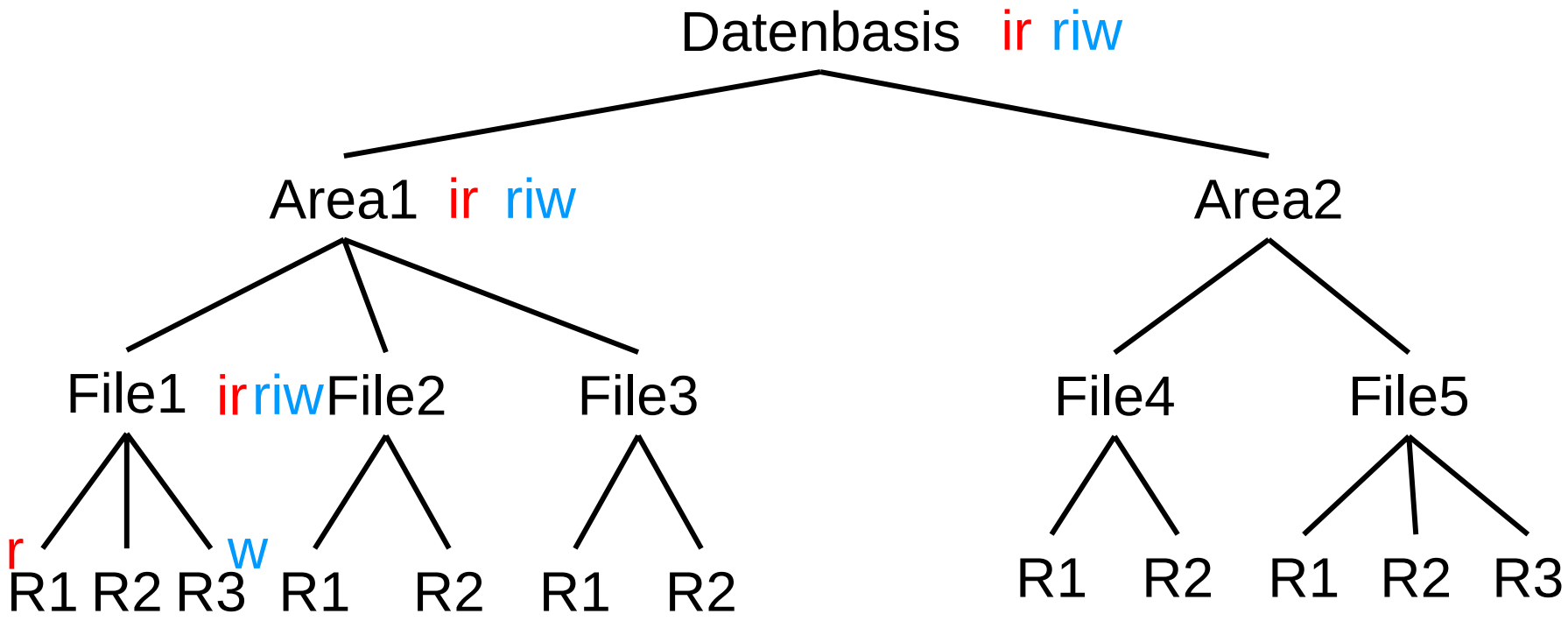
MGL-2-Phasen-Sperrprotokoll (6)

t_1 t_2 t_3



MGL-2-Phasen-Sperrprotokoll (7)

t_1 t_3



Korrektheit (1)

Satz 6.24

Falls alle Transaktionen dem MGL-Protokoll gehorchen, so halten keine zwei Transaktionen zwei in Konflikt stehende Sperren.

Korrektheit (2)

Beweis:

Es genügt, den Satz für Blattknoten zu zeigen, da Beweisführung Vorgänger einbezieht.

Wir nehmen also an, dass t_i und t_j zwei in Konflikt stehende Sperren auf den Blattknoten *x* halten. Es sind 7 Fälle zu unterscheiden:

	t_1	t_2
1	implizite *r* Sperre	explizite *w* Sperre
2	implizite *r* Sperre	implizite *w* Sperre
3	explizite *r* Sperre	explizite *w* Sperre
4	explizite *r* Sperre	implizite *w* Sperre
5	implizite *w* Sperre	explizite *w* Sperre
6	implizite *w* Sperre	implizite *w* Sperre
7	explizite *w* Sperre	explizite *w* Sperre

Korrektheit (3)

Wegen Regel 3 des MGL Protokolls: t_1 hält $rl_1(y)$ für einen Vorgänger y von x.

Wegen Regel 2 des MGL Protokolls: t_2 hält $iwl_2(z)$ für jeden Vorgänger z von x.

Insbesondere hält t_2 also $iwl_2(y)$. Widerspruch.

direkter Widerspruch

ähnlich 1

ähnlich 1

direkter Widerspruch

	t_1	t_2
1	implizite *r* Sperre	explizite *w* Sperre
2	implizite *r* Sperre	implizite *w* Sperre
3	explizite *r* Sperre	explizite *w* Sperre
4	explizite *r* Sperre	implizite *w* Sperre
5	implizite *w* Sperre	explizite *w* Sperre
6	implizite *w* Sperre	implizite *w* Sperre
7	explizite *w* Sperre	explizite *w* Sperre

Korrektheit (4)

Wg. Regel 3 des MGL Protokolls: t_1 hält $rl_1(y)$ für einen Vorgänger y von x.

Wg. Regel 3 des MGL Protokolls: t_2 hält $wl_2(y')$ für einen Vorgänger y' von x.

Wir unterscheiden drei Unterfälle ('>': Baumordnung):

$y=y'$ Widerspruch.

$y>y'$ Widerspruch, da t_2 $iwl_2(y)$ halten muss. (Konflikt zu $rl_1(y)$)

$y'>y$ Widerspruch, da t_1 $irl_1(y')$ halten muss. (Konflikt zu $wl_2(y')$)

	t_1	t_2
1	implizite *r* Sperre	explizite *w* Sperre
2	implizite *r* Sperre	implizite *w* Sperre
3	explizite *r* Sperre	explizite *w* Sperre
4	explizite *r* Sperre	implizite *w* Sperre
5	implizite *w* Sperre	explizite *w* Sperre
6	implizite *w* Sperre	implizite *w* Sperre
7	explizite *w* Sperre	explizite *w* Sperre

ähnlich 2

ung (1)

gehalten→ angefordert↓	rl	wl	irl	iwl	riwl
rl	yes	no	yes	no	no
wl	no	no	no	no	no
irl	yes	no	yes	yes	yes
iwl	no	no	yes	yes	no
riwl	no	no	yes	no	no

. wird für jeden Knoten eine
Eine hinzukommende Sperre
ung, eine wegfallende Sperre
hren (Sperrkonversion).

Sperrenverschärfung mittels Sperrkonversionstabelle: Falls eine Sperre *x* gehalten wird und eine Sperre *y* gewährt wird, ist der Eintrag das Maximum der beiden Sperren:

gehalten→ angefordert↓	r	w	ir	iw	riw
rl	r	w	r	iw	riw
wl	r	w	ir	iw	riw
irl	r	w	ir	iw	riw
iwl	r	w	iw	iw	riw
riwl	r	w	riw	iw	riw

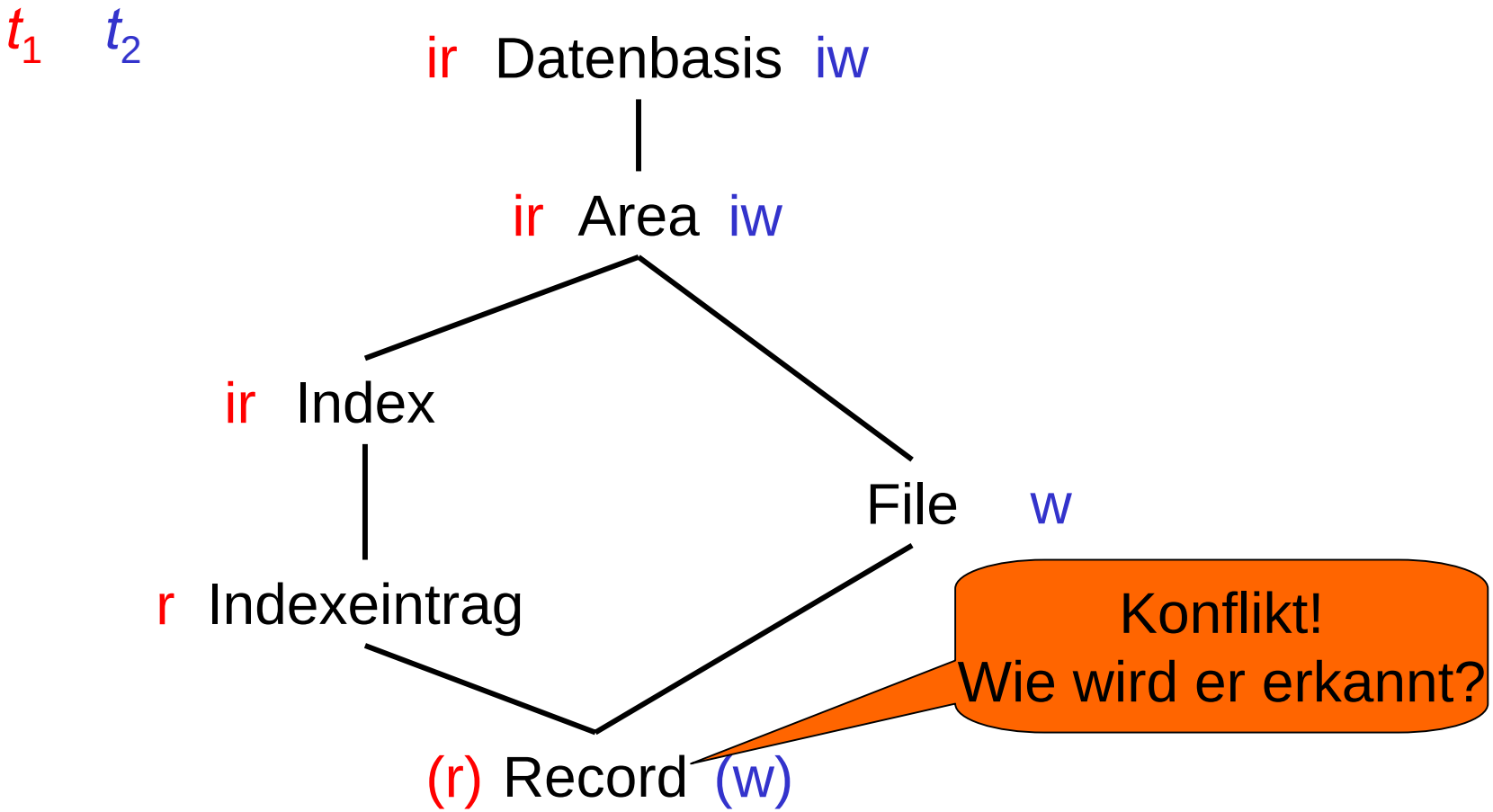
Umsetzung (2)

Wann wird *w*- oder *r*- Sperre auf welchem Korn angefordert?

- Das ist ein Aufwandsproblem. Bei kleiner Korngröße ergeben sich keine Spielräume. Bei großer ist ein gewisses Wissen notwendig darüber, auf wie viele Datenelemente kleinerer Größe zugegriffen wird.
- Dynamische Lösung: Falls eine bestimmte Anzahl von Zugriffen überschritten wird, wird eine entsprechende Sperre für das nächst höhere Korn angefordert.

Körnigkeitsgraphen (1)

Erweiterung von Bäumen auf DAGs.
Beispiel:



Körnigkeitsgraphen (2)

1. Falls eine *r*- oder *ir*-Sperre benötigt wird für ein Granulat, müssen alle Vorgänger *ir* oder *iw* gesperrt sein.
2. Falls eine *w*- oder *iw*-Sperre benötigt wird für ein Granulat, müssen alle Vorgänger mit *riw* oder *iw* gesperrt sein.

3a Falls eine Transaktion t_i ein Datenelement *x* (implizit oder explizit) lesen will, muss sie eine *r*-, *riw*- oder *w*-Sperre für *x* oder einen beliebigen Vorgänger von *x* halten.

3b Falls eine Transaktion t_i ein Datenelement *x* (implizit oder explizit) schreiben will, muss sie auf ***jedem*** Pfad von *x* zur Wurzel für einen beliebigen Vorgänger von *x* eine *w*-Sperre halten.

4. Eine *ix*-Sperre kann nur freigegeben werden, falls keine Sperre mehr für Nachfolger gehalten wird.

Tree Locking Schedulers

Baum-Sperr-Protokolle

- **Zur Erinnerung**: R/W–Modell basiert allein auf äußerer Beobachtung der Transaktionen.
- **Erwartung**: Zusatzkenntnisse über die Transaktionen lassen sich für höhere Nebenläufigkeit und/oder geringere Berechnungskomplexität nutzen.
- **Beispiel**: Transaktionen mit Durchlauf von Bäumen. Weiß man um
  - die Struktur des Baumes,
  - den Einstieg an der Wurzel,

  kann man Aussagen über die Abfolge von Zugriffen machen.
- **Konsequenz**: Keine 2PL-Forderung notwendig.

Einfaches Baumsperrverfahren (1)

- Gegeben Datenbaum mit Wurzel *r*.
- Es gibt nur eine Zugriffsfunktion $a_i(x)$: Transaktion t_i greift auf Datenelement *x*, hier ein Knoten eines Baumes, zu.
  - Daher der Name **Write-only Tree Locking (WTL)**
- Die zugehörige Sperranforderung ist $al_i(x)$. Sperrfreigabe wird durch $au_i(x)$ bezeichnet.
- Falls $i \neq j$, gilt $a_i(x) \not\parallel a_j(x)$ und somit $al_i(x) \not\parallel al_j(x)$.

Einfaches Baumsperrverfahren (2)

Einfaches Baumsperrprotokoll (WTL):

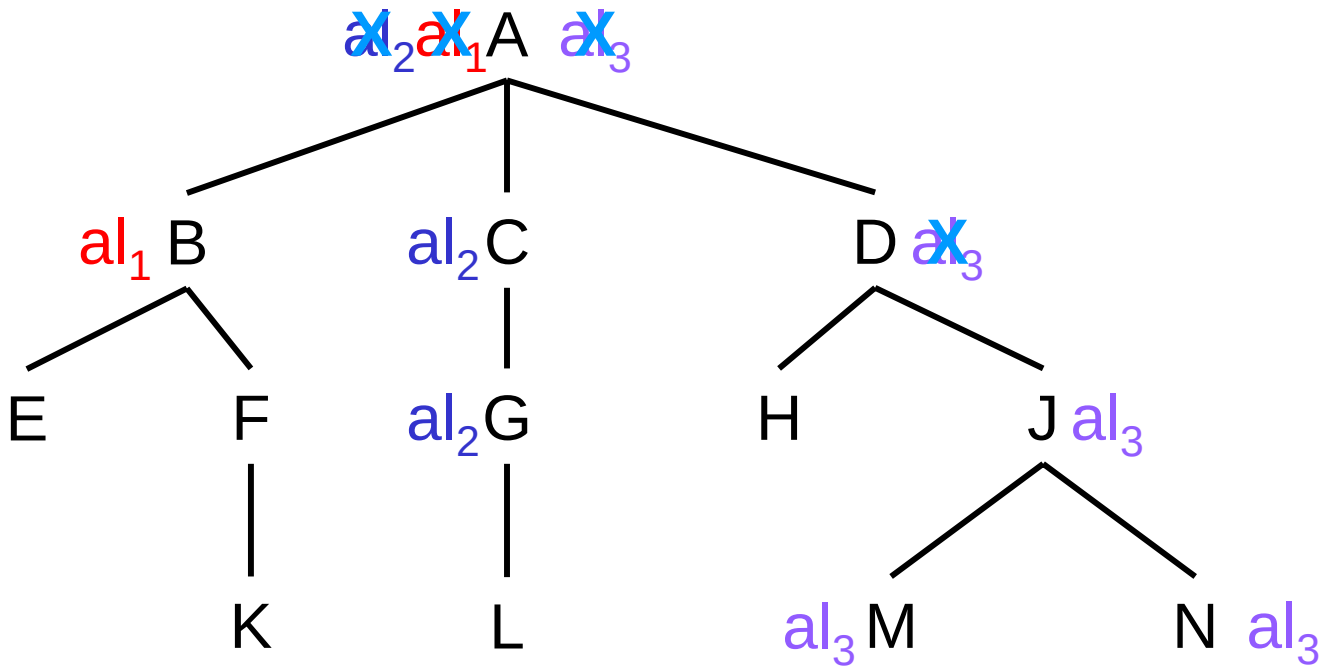
Es müssen die Regeln LR 1 – LR 4 eingehalten werden. Zusätzlich gelten die folgenden Regeln:

WTL1: Für alle Knoten *x* außer der Wurzel gilt: $al_i(x)$ kann nur gesetzt werden, wenn t_i $al_i(y)$ hält *für y Vaterknoten von x*.

WTL2: Nachdem eine Transaktion eine Sperre auf *x* freigegeben hat, darf sie sie nicht neu setzen.

- **Sperrkopplung (lock coupling)**: Der Scheduler kann $al_i(x)$ nur freigeben, falls die Transaktion alle Sperren auf den benötigten Söhnen von *x* bereits hält.
  - Impliziert eine Sperranforderungsrichtung von der Wurzel zu den Blättern.

Einfaches Baumsperrverfahren (3)



Mindestens 1 Sperre pro benutzten Pfad wird garantiert:
Keine Überholung durch andere Transaktion möglich.

Korrektheit

Satz 6.25

$Gen(WTL) \subseteq CSR$

Beweis:

Betrachte $t_i \rightarrow t_j \in G(h)$ für WTL-Historie h.

$\succ \exists a_i(x), a_j(x) \in h \;\; a_i(x) \| a_j(x) \wedge a_i(x) <_h a_j(x)$ (Interpretation Kante!)

$\succ au_i(x) <_h al_j(x)$

$\succ au_i(r) <_h al_j(r)$ (r Wurzel)

WTL1: $al_i(r) <_h al_j(x)$

$\succ au_i(r) <_h al_j(x)$

Annahme: $G(h)$ hat Zyklus der Form

$$t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$$

$\succ au_1(r) <_h al_1(r)$ Widerspruch zu WTL2.

Verklemmungsfreiheit

Satz 6.26

Das WTL-Protokoll vermeidet Verklemmungen.

Beweis:

Falls t_i auf die Wurzelsperre wartet, so kann sie nicht in einer Verklemmung involviert sein, da sie keine Sperren besitzt.

Annahme: t_i wartet auf $au_j(x)$, $x \neq r \succ au_j(r) <_h al_i(r)$

Durch Induktion: Falls der Wartegraph einen Zyklus hat, der t_i beinhaltet, so $au_i(r) <_h al_i(r)$. Widerspruch!

Allgemeines Baumsperrverfahren (1)

- Differenzierung nach Lesen und Schreiben.
- Falls Transaktionen jeweils nur Lesesperren oder nur Schreibsperren anfordern, genügen die normalen Konfliktregeln zwischen diesen Sperren, um Serialisierbarkeit zu gewährleisten.
- Dies ist nicht der Fall, falls eine Transaktion sowohl Lese- als auch Schreibsperren setzt.
  - **Read-Write Tree Locking** (**RWTL**)

Allgemeines Baumsperrverfahren (2)

Problem: t_i locks root before t_j does,
but t_j passes t_i within a "read zone"

Example

$t_1 = t_2 = w(x)\ r(y)\ w(z)$

$h = wl_1(x)\ w_1(x)\ rl_1(y)\ wu_1(x)\ wl_2(x)\ w_2(x)\ rl_2(y)\ wu_2(x)\ r_2(y)\ wl_2(z)$
$ru_2(y)\ w_2(z)\ wu_2(z)\ r_1(y)\ wl_1(z)\ ru_1(y)\ w_1(z)\ wu_1(z)$

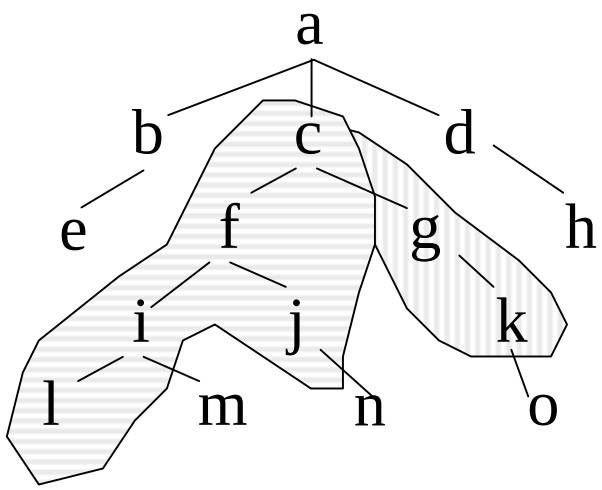
→ appears to follow WTL rules
but ∉ CSR
→ t_1 passes t_2 on element *y* which both only read

Solution: formalize "read zone"
and enforce two-phase property on "read zones"

Locking Rules of RWTL

For transaction t with read set RS(t) and write set WS(t)
let C_1, ..., C_m be the connected components of RS(t).
A **pitfall** of t is a set of the form
$C_i \cup \{x \in WS(t) \mid x \text{ is a child or parent of some } y \in C_i\}$.

Example:
t with RS(t)={f, i, g} and WS(t)={c, l, j, k, o}
has C_1={f, i} and C_2={g}
and pitfalls pf_1={c, f, i, l, j} and pf_2={c, g, k}.



Definition 6.27:
Under the **read-write tree locking protocol (RWTL)** lock requests and releases must obey LR1 - LR4, WTL1, WTL2, and the two-phase property within each pitfall.

Correctness of RWTL

Theorem 6.28:
Gen(RWTL) $\subseteq$ CSR.

TL für beliebigen Einstieg

- TL verlangt nicht zwingend den Einstieg in einen Baum an der Wurzel.
- Jeder beliebige Knoten kann als Einstiegspunkt verwendet werden, allerdings beschränkt sich dann das Sperren auf den so definierten Teilbaum.
- Hieraus kann sich mehr Nebenläufigkeit ergeben.

Freigabe von Sperren

Sperren können eher als bei 2-PL freigegeben werden:

Falls alle benötigten Nachfolger von *x* gesperrt sind, kann die Sperre von *x* aufgegeben werden.

Problem:

Wie kann entschieden werden, wann dies der Fall ist ? („Was wird noch benötigt ?“)

Antworten:

- rein mechanisch: falls alle Nachfolger gesperrt sind
- unter Zusatzkenntnissen: durch Hinweise der Transaktion

Falls nur ein Teil der Nachfolger gesperrt ist, Hinweise aber fehlen, degeneriert RWTL zu S2PL.

Grenzen des Nutzens

Frühe Freigabe von Sperren erhöht die Nebenläufigkeit:

Weniger Sperren werden gehalten, weniger Konflikte, weniger Warten.

Aber:

- Realisierbar nur, wenn der Baum von der Wurzel zu den Blättern durchlaufen wird.
- Zudem liegt die Freigabe des Sperren in der Verantwortung der Transaktion, der Scheduler kennt (zumeist) den Zeitpunkt nicht.
- (Und im Vorgriff:) Die Recovery fordert, dass Sperren bis zum Transaktionsende gehalten werden $\Rightarrow$ kein Gewinn.

Predicate Locking

(für Interessierte)

Prädikatsperren

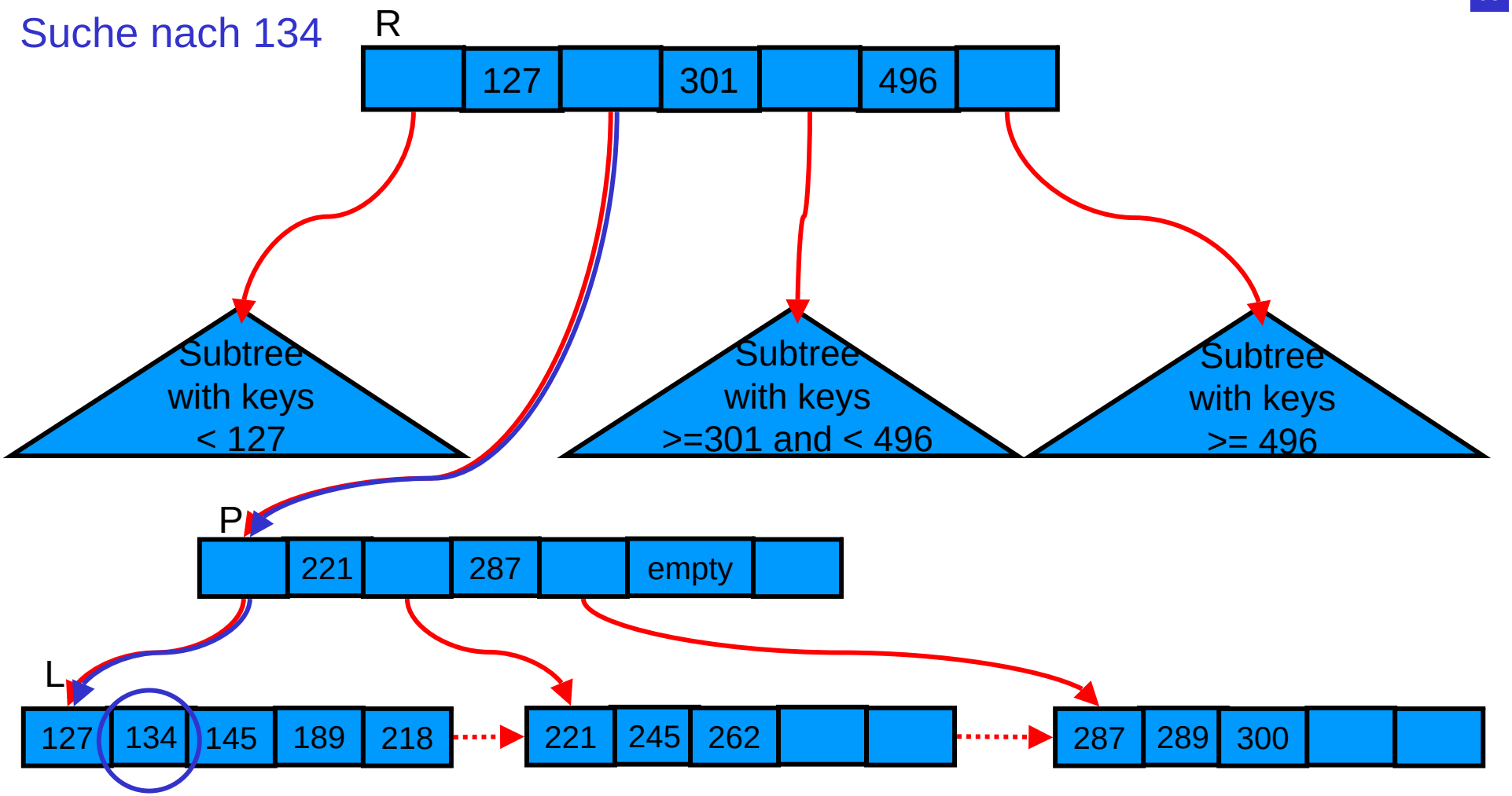
Prädikatsperren

- Prädikatsperren identifizieren die zu sperrenden Datenelemente über eine gemeinsame Eigenschaft (auch: *semantische Sperre*).
  - Beispiel: Multi-Korngrößen-Sperren mit dem einfachen Prädikat „physisch enthalten“.
- Hier: Schlüsselbereichssperren (key range locks), die über die gesamte Transaktionsdauer gehalten werden. Dieser Bereich kann auch nur ein einzelner Schlüsselwert sein, falls es sich um einen Sekundärindex handelt (mehrere Einträge pro Schlüssel).

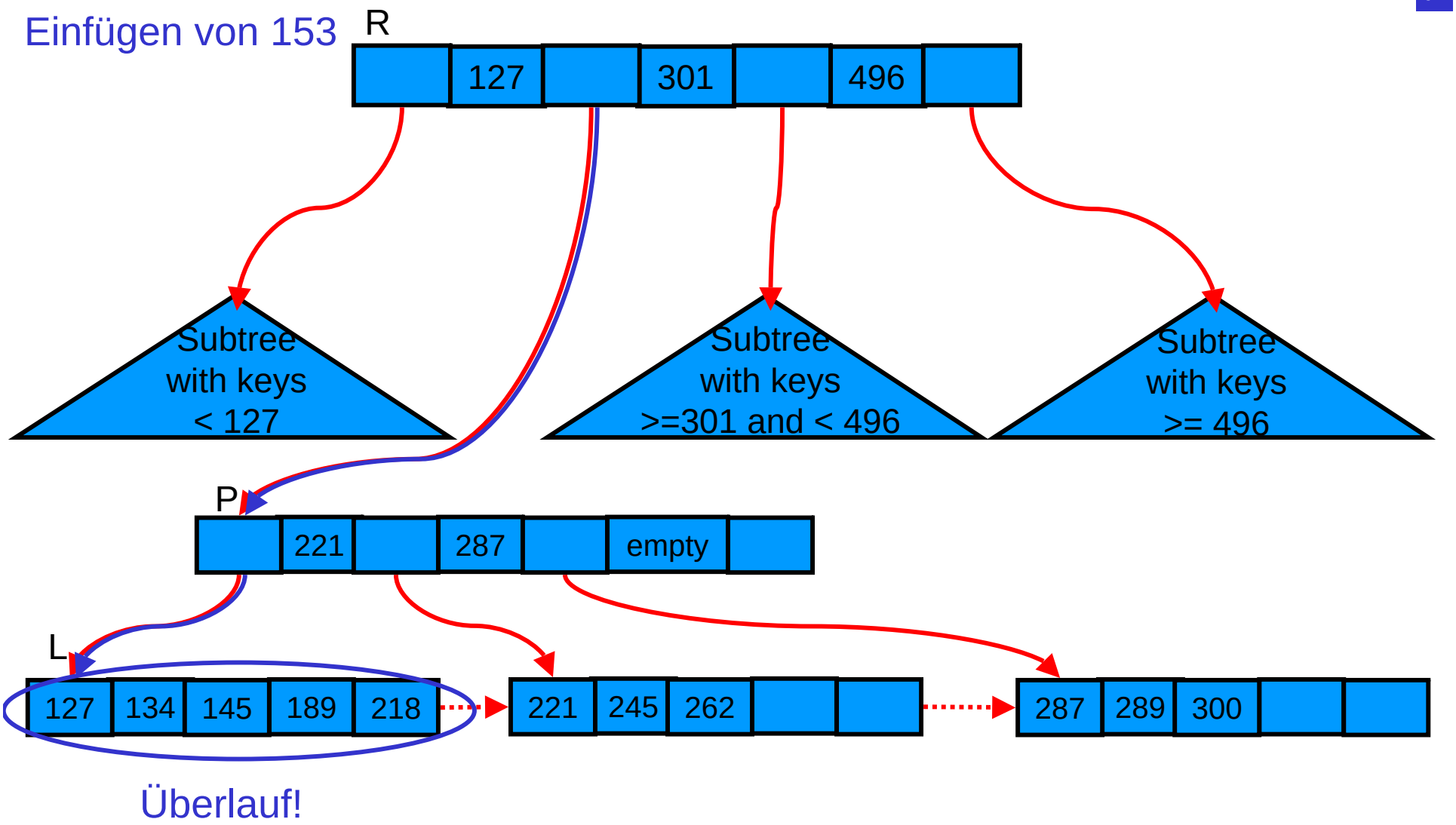
B-Baum-Protokolle (1)

- TL nicht zu übernehmen, da Baum in beiden Richtungen durchlaufen werden kann.
- Notwendigkeit: Spezielle Protokolle wegen hoher Bedeutung des B-Baum (und Varianten) für Indexstrukturen.
- Zusatzkenntnisse: Über B-Baum kennen wir sogar die Algorithmen.
- Vorgehen: Die für uns wichtigen Operationen sind `search`, `insert` und `delete`. Da sich `delete` im wesentlichen auf `insert` zurückführen lässt, betrachten wir nur `search` und `insert`.
  - Basieren auf Schlüsselwerten.

B-Baum-Protokolle (2)



B-Baum-Protokolle (3)



B-Baum-Protokolle (4)

Problem: Eine `insert`-Operation weiß erst bei Erreichen der Blattseite, ob ein Split nötig ist.

Einfache Lösung:

- TL speziell: Zunächst Schreibsperren auf jede zu besuchende Seite zu setzen. Falls diese sich dann bei der anschließenden Inspektion als nicht voll erweist, kann die Sperre beim Vorgänger in eine Lesesperre umgewandelt werden.
  - Dies kann aber zu erheblich weniger Nebenläufigkeit führen.

B-Baum-Protokolle (5)

Problem: Eine `insert`-Operation weiß erst bei Erreichen der Blattseite, ob ein Split nötig ist.

Volles Wissen um und Eingriff in die Algorithmen

- Grundgedanke für das Ändern: Rein lokale Entscheidungen.
- Modifiziere `insert` so, dass es einen Update auf einer Seite durchführen kann, ohne eine Sperre auf dem Vorgängerknoten zu besitzen.
- Also: Entwicklung eines Protokolls, so dass bei `insert` nur *w*-Sperre auf Seite mit Update gehalten wird, nicht aber Sperre auf dem Vorgängerknoten.
- Falls kein Spalten notwendig ist, kein Problem.

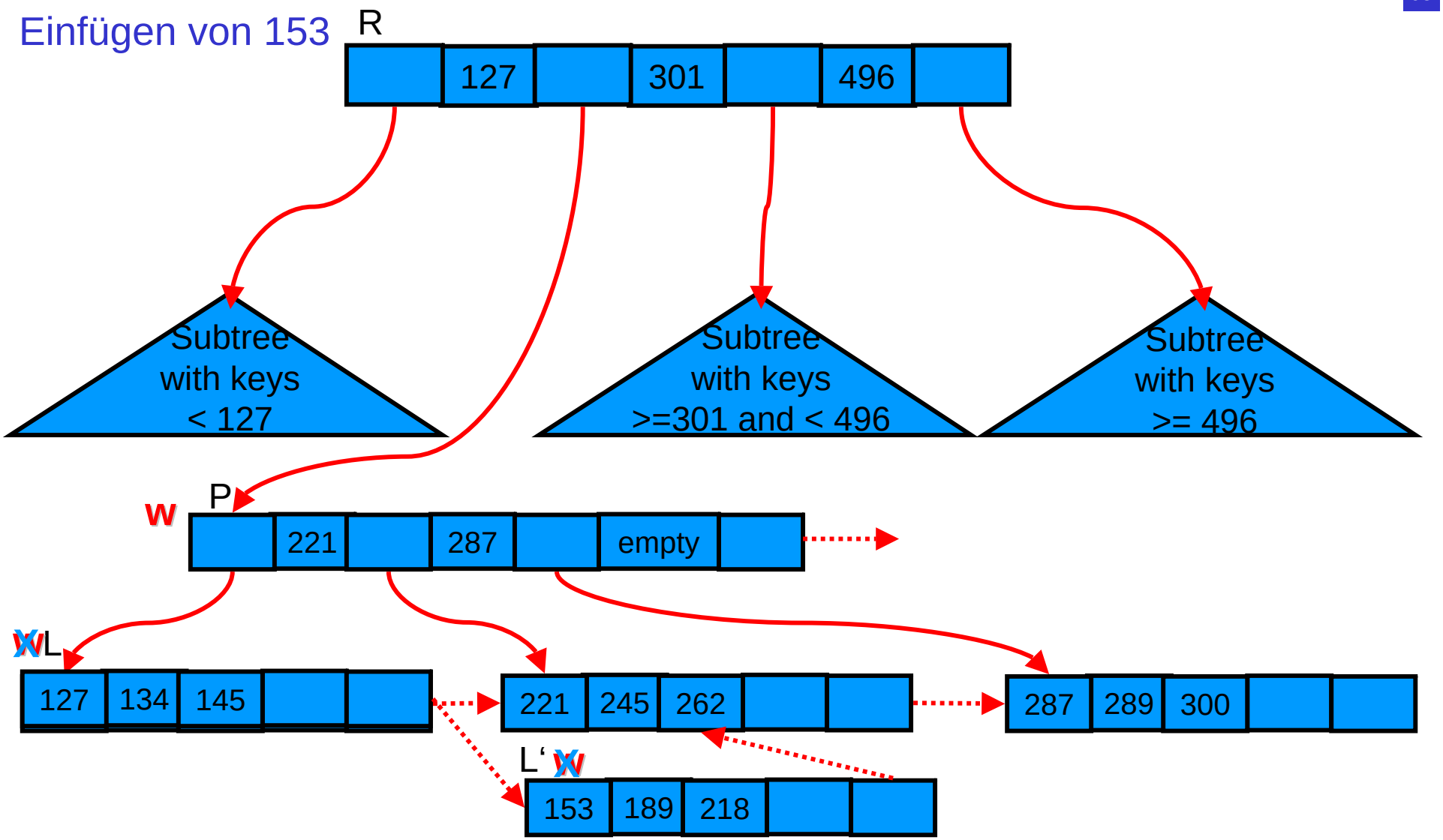
B-Baum-Protokolle (6)

Bei Spalten einer Seite:

- Wie üblich, wird zunächst eine neue Seite *L'* erzeugt, dann die „obere Hälfte“ der Einträge der vollen Seite *L* in die neue Seite eingetragen. Die Geschwisterverweise (Links) von *L* und *L'* werden aktualisiert. Jetzt kann `insert` die Sperren auf *L* und *L'* freigeben. Die Transaktion hält keine weiteren Sperren!
- Im zweiten Schritt wird eine *w*-Sperre auf dem Vorgängerknoten angefordert und der Zeiger von *L'* dort eingetragen.
- Falls der Vorgängerknoten voll ist, erfolgt ein weiteres Spalten.

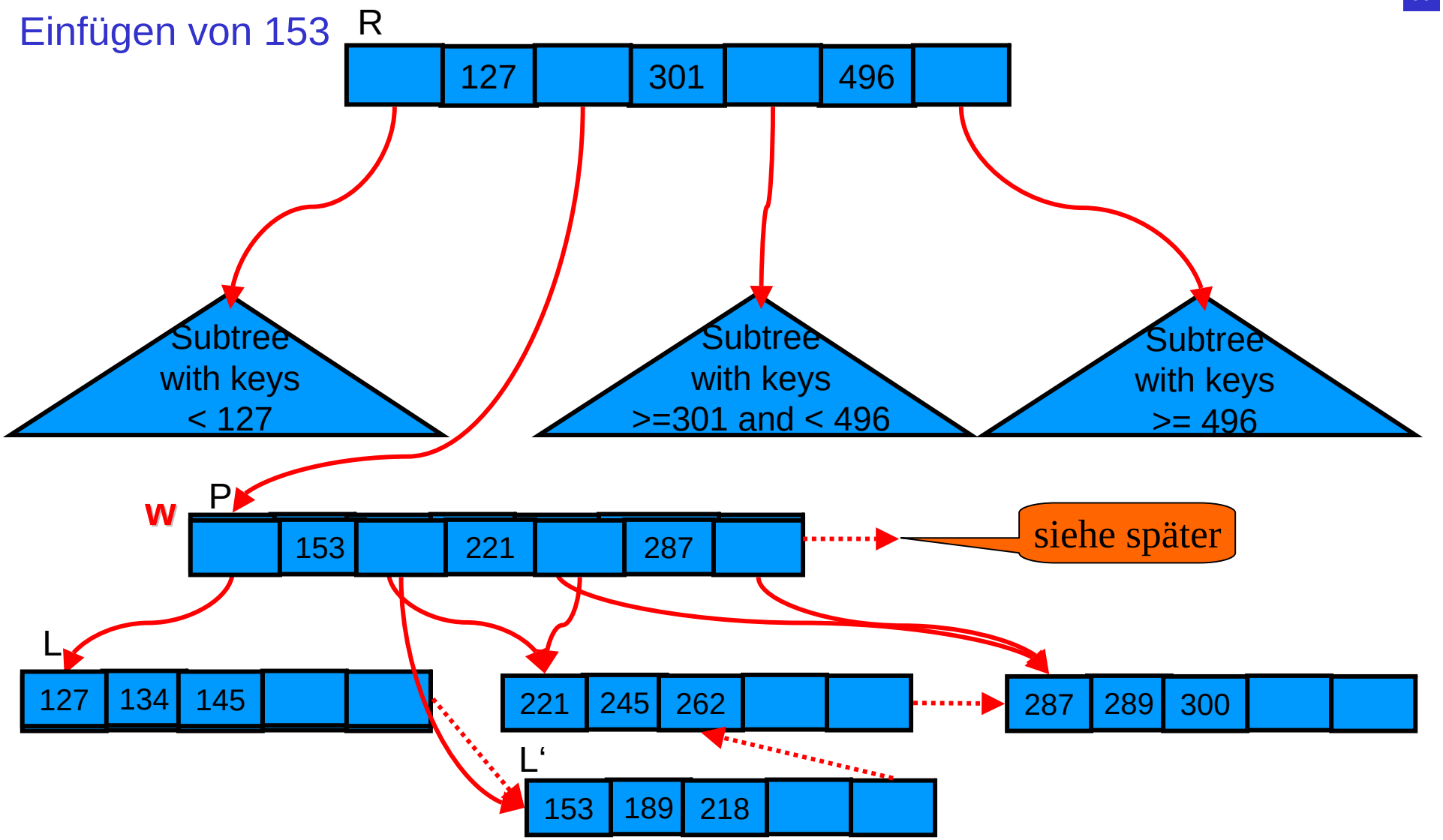
B-Baum-Protokolle (7)

Einfügen von 153



B-Baum-Protokolle (7)

Einfügen von 153



B-Baum-Protokolle (8)

- Folge: Beim Lesen muss die Umgebung auf die lokalen Änderungen untersucht werden (lokales Ändern - globales Lesen).
- Die Umgebung kann aber nicht mehr durch Sperren definiert werden, sondern nur durch Inhalte. Diese müssen über zusätzliche Links beschafft werden.
- Folge: Links nunmehr auf allen Ebenen notwendig (Sperrprotokoll wird in die Baumstruktur eingebaut
- Teil 1: Sperr-Protokoll bei **`search`**: Anfordern einer Lesesperre, direkt nachdem eine Seite gelesen wurde, wird die entsprechende Sperre wieder freigegeben. Danach wird die Sperre der Nachfolgerseite angefordert und diese gelesen.

B-Baum-Protokolle (9)

Teil 2: Systematische Inspektion der Umgebung.

- Es gibt einen Moment, zu dem **`insert`** keine Sperre hält. Dann kann es von einem **`insert`** oder **`search`** überholt werden.
- Da die Seiten von unten nach oben verändert werden, trifft die überholende Transaktion auf eine Änderung, für die es möglicherweise weiter oben noch keinen Hinweis gab, für die also noch kein Nachfolgerzeiger existierte. Dazu muss die Suche so modifiziert werden, dass stets auch die über den Link erreichbare Nachbarseite mit untersucht wird.
- Ebenso gibt es einen Moment, zu dem **`search`** keine Sperre hält. Überholung durch **`search`** ist unproblematisch. Überholung durch **`insert`** wird wie zuvor durch Untersuchung die Nachbarseite behandelt.

B-Baum-Protokolle (10)

Anmerkungen:

- Da beide Operationen, **`insert`** und **`search`**, nur dann eine Sperre anfordern, wenn sie keine halten, kann es zu keiner Verklemmung kommen.
- Das Protokoll funktioniert, da es Zusatzwissen über die Algorithmen für **`insert`**, **`search`** ausnutzt.
- Die Algorithmen für **`insert`**, **`search`** sind nicht mehr unabhängig vom verwendeten Synchronisationsprotokoll.

Hot Spots

Hot Spot: Datenelement, das viele Transaktionen zu ändern wünschen.

→ Systemengpass

→ Gefahr des **Konvoiphänomens**: Bei ungeschickter Schedulingstrategie Ausbildung langer Warteschlangen von Transaktionen, die sich von Hot Spot zu Hot Spot fortpflanzen.

Beispiele für Hot Spots:

- Wurzeln von Bäumen (daher TL-Protokoll!)
- Zähler

Feldzugriffe (1)

Strategie: Mischung aus konservativem und aggressivem Scheduling.

Feldzugriff: Eine Aktion (auf einem Hot Spot-Record) wird in zwei Teile zerlegt,

- ein Prädikat und
- eine Transformation.

Betrachte

Prädikat: Bestand ≥ 10
Transformation: Bestand = Bestand -10;

```
exec sql update hotspot Bestände
         set            Bestand = Bestand - 10
         where          Sorte = Weißweine
         and            Bestand ≥ 10;
```

Feldzugriffe (2)

Protokoll von Feldzugriffen :

1. Sofortiger Test des Prädikates unter kurzer Lesesperre. (Die Sperre auf das ungeänderte Datenelement wird freigegeben, sobald der Test beendet ist.)
2. Falls der Test zu *falsch* evaluiert, wird abgebrochen.
3. Ansonsten wird ein REDO Log Record mit Prädikat und Transformation angelegt.
4. Bei Erreichen von Commit werden 2 Phasen durchgeführt:
   - Phase 1 Alle REDO Log Records der beendenden Transaktion werden bearbeitet, Lesesperren werden angefordert für Feldzugriffe, die keine Transformation beinhalten, Schreibsperren für alle anderen Feldzugriffe. Danach werden alle Prädikate noch einmal evaluiert. Falls mindestens eines zu falsch evaluiert, wird die Transaktion zurückgesetzt. Ansonsten Eintritt in Phase 2 des Commits.
   - Phase 2 Alle Transformationen werden angewendet und die Sperren freigegeben.

Feldzugriffe (3)

Vorteil: Statt einer langdauernden Sperre nur noch Kurzzeitsperren.

Nachteile:

- Die Prädikatevaluierung in Phase 1 des Commits kann (wegen zwischenzeitlicher Änderung durch zweite TA) nachträglich doch noch fehlschlagen, wodurch die Transaktion dann zurückgesetzt werden muss.
- Da eigene Änderungen erst am Ende festgeschrieben werden, werden diese durch die Transaktion bei erneutem Lesen nicht berücksichtigt.

Feldzugriffe (4)

Beispiel:

t1 qoh > 150?
qoh := qoh - 150
commit

t2 qoh > 800?
qoh := qoh - 800
commit

t3 qoh > 100?
qoh := qoh - 100
commit

Feldzugriffe (5)

t1	t2	t3	qoh
qoh>150			1000
	qoh>800		
	commit		
	qoh>800		
	qoh:=qoh-800		200
		qoh>100	
commit			
qoh>150			
qoh:=qoh-150			50
		commit	
		qoh>100 F	

Escrow-Sperren

- **Escrow-Sperre**: Zu beachtendes numerisches Intervall.
- Test ob gegebener Wert *sicher* im Intervall liegt.

t1	t2	t3	Escrow	qoh
			[1000,1000]	1000
qoh>150				
qoh:=qoh-150			[850,1000]	
	qoh>800			
	qoh:=qoh-800		[50,1000]	
	commit		[50,200]	200
		qoh>100 F		
commit			[50,50]	50

t3 könnte warten, bis fest steht, ob die Untergrenze wieder angehoben wird oder nicht, oder abbrechen.

Non-locking schedulers:
Time stamp ordering

Synchronization strategy pursued

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

Aggressive scheduler: Set equivalence time to time at the beginning ⇒ Try immediate execution of an operation ⇒ Little freedom for reordering (fewer equivalent serial schedules can be constructed).

Synchronization by **time stamping**.

Pessimistic strategies
"Watch every step"

Time stamp ordering protocol (1)

Intuition:

- Choose for each transaction as the equivalence time the time of the first operation, i.e., the equivalent serial schedule is practically based on the order in which the transactions were started.
- Conflict equivalence of the real and serial schedules is ensured by checking for each operation whether - without reordering incompatible operations - it could have been executed at the start of the transaction.
- If the check fails abort the transaction.

Time stamp ordering protocol (2)

- **Time stamp ordering (*TO*)** assigns to each transaction t_i a unique time stamp $ts(t_i)$.
  - Each operation of a transaction is assigned the one and same time stamp of the transaction .
- The *TO* protocol follows the rule :

$$\forall p_i(x) \nparallel q_j(x) \in s,\ i \neq j \quad p_i(x) <_s q_j(x) \Leftrightarrow ts(t_i) < ts(t_j)$$

  - *TO* scheduler orders – if possible – operations that are in conflict wrt to their time stamps.

Time stamp ordering protocol (3)

Theorem 6.29

$Gen(TO) \subseteq CSR$

Sketch of proof:

$t_i \rightarrow t_j \in G(h)$

$\succ \exists p_i(x) \not\parallel q_j(x) \in h \;\; p_i(x) <_h q_j(x)$

$\succ ts(t_i) < ts(t_j)$ (interpretation edge!)

Exists cycle $t_1,\ldots,t_n,t_1 \in G(h)$.

By induction: $ts(t_1) < ts(t_1)$. Contradiction!

BTO (1)

- Base TO (BTO) is a straightforward aggressive implementation of *TO*. All operations are directly passed on to the data manager (FCFS ordering).
- Late operations are rejected. Operation $p_i(x)$ *is late* if

  $$\exists\, q_j(x) \in s \;\; p_i(x) \not\parallel q_j(x) \wedge q_j(x) <_s p_i(x) \wedge ts(t_j) > ts(t_i)$$

  (Since $q_j(x)$ has already been executed, $p_i(x)$ must be rejected because there is no way to satisfy *TO*. $\Rightarrow$ t_i must be aborted.
- On restart t_i must be assigned a new and higher time stamp so that one can hope that by now the incompatible operations can correctly be ordered.

BTO (2)

- To determine whether an operation is late, the BTO scheduler holds for each *x* two values:

  max-r-scheduled(*x*)

  - Value of the largest time stamp of earlier read operations on *x* sent to the data manager.

  max-w-scheduled(*x*)

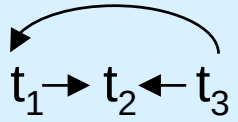
  - Value of the largest time stamp of earlier write operations on *x* sent to the data manager.

- $p_i(x)$ arrives at the scheduler.
  - The scheduler compares $ts(t_i)$ with all *max-q-scheduled*(*x*) where *q* and *p* are in conflict.
  - If $ts(t_i) <$ *max-q-scheduled*(*x*), $p_i(x)$ is rejected.
  - Else $p_i(x)$ is passed on to the data manager, and if $ts(t_i) >$ *max-p-scheduled*(*x*), *max-p-scheduled*(*x*) is set to $ts(t_i)$.

BTO (3)

Example 6.30

However:

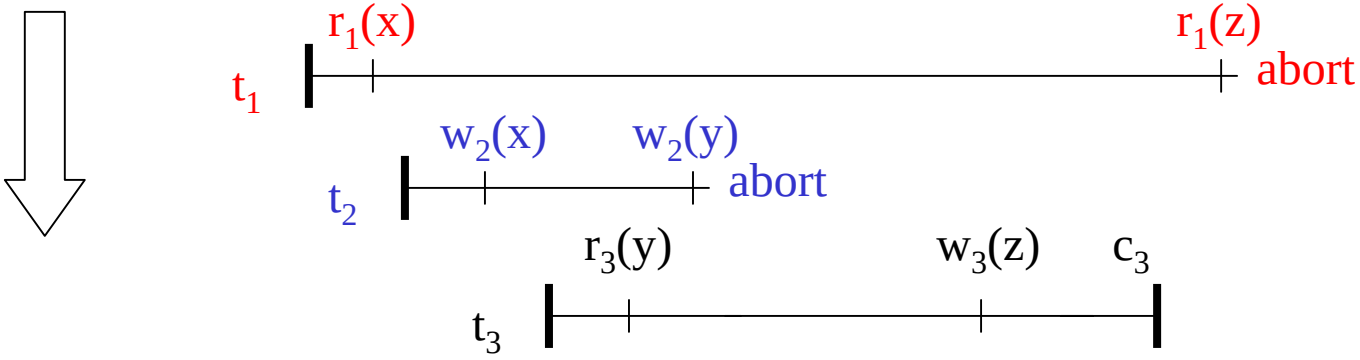


$s = r_1(x)\ w_2(x)\ r_3(y)\ w_2(y)\ c_2\ w_3(z)\ c_3\ r_1(z)\ c_1$

cannot be generated because :

$s = r_1(x)\ w_2(x)\ r_3(y)\ w_2(y)\ c_2\ w_3(z)\ c_3\ r_1(z)\ c_1$

a_2 a_1



$r_1(x)\ w_2(x)\ r_3(y)\ a_2\ w_3(z)\ c_3\ a_1$

BTO (4)

Scheduler und Data manager (DM) are autonomous: Coordination needed

- (Aggressive) scheduler and DM must coordinate their activities to ensure that DM maintains the order of the incompatible operations as submitted by the scheduler.
  - Not needed for 2PL because (conservative) scheduler delays incompatible operations already outside DM.
- Handshake needed: Scheduler counts for each *x* the *r* and *w* operations that were sent to DM but not yet acknowledged as completed:

  r-in-transit(*x*)

  w-in-transit(*x*)
- A queue *queue* (*x*) maintains operations that could be TO-scheduled but still wait for an acknowledgement by DM of earlier operations that are in conflict.

BTO (5)

Example 6.31

op	action	max-r-scheduled	r-in-transit	max-w-scheduled	w-in-transit	queue
		0	0	0	0	()
$r_1(x)$	to DM	1	1			
$w_2(x)$	wait			2		$(w_2(x))$
$r_4(x)$	wait	4				$(w_2(x),r_4(x))$
$r_3(x)$	wait					$(w_2(x),r_4(x),r_3(x))$
$ack(r_1(x))$	$w_2(x)$ to DM		0		1	$(r_4(x),r_3(x))$
$ack(w_2(x))$	$r_4(x),r_3(x)$ to DM		2		0	()

Striktes TO (1)

- Zur Erinnerung strikte Historien: Schreibt eine Transaktion t_i das Datenelement x, so darf eine weitere Transaktion t_j, die erst anschließend auf x zugreift, dies erst nach dem Ende von t_i tun.
- Der strikte TO-Scheduler arbeitet wie der Basis-TO-Scheduler, außer dass *w-in-transit*(x) erst bei a_i oder c_i auf 0 gesetzt wird.
- Hierdurch werden $r_j(x)$ und $w_j(x)$ mit $ts(t_j) > ts(t_i)$ verzögert, bis t_i beendet ist. *w-in-transit*(x) verhält sich wie eine Sperre.
- Trotzdem: Auch jetzt können Verklemmungen nicht auftreten, da t_j nur auf t_i wartet, falls $ts(t_j) > ts(t_i)$.

Striktes TO (2)

Beispiel:

$$h = r_2(x)\ w_3(x)\ c_3\ w_1(y)\ c_1\ r_2(y)\ w_2(z)\ c_2$$

h ist CSR: $G(h)$: $t_1 \rightarrow t_2 \rightarrow t_3$

h ist strikt: $w_1(y) <_h r_2(y)$ und $c_1 <_h r_2(y)$.

Falls $ts(t_1) < ts(t_2) < ts(t_3)$, ist h strikt TO.

SG-basierte Protokolle

Idee: Scheduler basiert auf Konfliktgraphtests.

- Ein SGT-Scheduler baut einen **Serialisierbarkeitsgraph** (SG) des Schedule explizit auf. Operationen ändern den SG. Anpassungen gegenüber Konfliktgraph:
  - Der SG enthält neben abgeschlossenen TAs auch aktive. (Laufende Ermittlung erforderlich!)
  - Der SG enthält nicht jede abgeschlossene TA. (Der größte Teil der Vergangenheit interessiert nicht!)
- Der SGT-Scheduler achtet darauf, dass der SG immer zyklenfrei bleibt.

Basis-SGT (1)

Wenn der SGT-Scheduler eine Operation $p_i(x)$ erhält,

1. falls $t_i \notin$ SG: einfügen
2. für alle $q_j(x) <_s p_i(x)$, $i \neq j$, $q_j(x) \parallel p_i(x)$: SG $\cup \{t_j \rightarrow t_i\}$
3. falls SG jetzt Zyklus enthält: abort t_i. Nach Bestätigung von Daten-Verwalter: SG $\setminus \{t_j \rightarrow t_i\}$
4. sonst: plane $p_i(x)$ ein. Sobald alle in Konflikt stehenden Operationen bestätigt, Weitergabe an Daten-Verwalter.

- Zur Bestimmung von Bedingung 2 zusätzlich für jede t_j
  r-scheduled(t_j) und *w-scheduled*(t_j)
  in denen die Lese- und Schreibmengen (Datenelemente) von t_j gehalten werden (SG zu grobe Abstraktion!).
- Zur Bestimmung von Bedingung 4 wieder *q-in-transit*(x).
- Um Striktheit zu gewährleisten, geht man analog zu striktem TO vor.

Basis-SGT (2)

Wann können TA's aus SG gelöscht werden?

- Nicht direkt nach commit! Betrachte

  $h = r_{k+1}(x)\ w_1(x)\ w_1(y_1)\ c_1\ w_2(x)\ w_2(y_2)\ c_2 \ldots\ w_k(x)\ w_k(y_k)\ c_k$

  SG: $t_{k+1} \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k$

- Jetzt erreiche $w_{k+1}(z)$ den Scheduler.
- Dieser muss testen, ob $z \in \{x, y_1, \ldots, y_k\}$ (falls ja: Zyklus!).
- Hierzu wird jedoch genau die Menge $\{t_1, \ldots, t_k\}$ benötigt , obwohl diese Transaktionen schon abgeschlossen sind.

Basis-SGT (3)

Triviale Feststellung:

Abgeschlossene Transaktionen können aus SG entfernt werden, wenn sie nicht mehr an einem Zyklus teilnehmen können.

Lösung:

- Für Zyklus benötigen Knoten mindestens eine einfallende und eine ausgehende Kante.
- Abgeschlossene Transaktionen ohne einfallende Kanten können aus SG entfernt werden, da nach einem Commit nur ausgehende Kanten entstehen können.

Optimistic schedulers

Synchronization strategy pursued

Decision time for trying to place the transaction in the equivalent serial schedule.

before vs. during vs. at commit of a TA

That just leaves equivalence time to be set to commit time ⇒ This time known only at the end ⇒ Failure possible ⇒ Limited opportunities for reordering.

Bei Rücksetzen muss die gesamte Transaktion unschädlich gemacht werden.
Anwendbarkeit: Bei geringer Konflikthäufigkeit

Optimistic strategies
"Go ahead and then see"

Optimistische Verfahren (1)

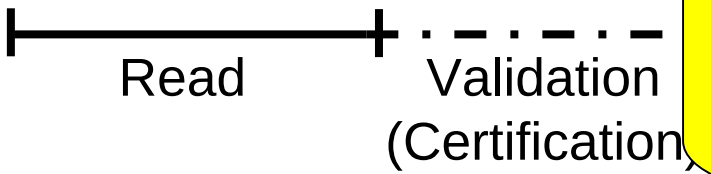
Drei Transaktionsphasen:

1. **Read-Phase:** Die Transaktion wird ausgeführt, aber alle *w*-Aktionen wirken nur auf lokalen Arbeitsbereich. (Keine Verzögerungen anderer Transaktionen!)
2. **Validation-Phase:** Eine commit-willige TA wird validiert, d.h. es wird geprüft, ob sie *CSR* abgelaufen ist.
3. **Write-Phase:** Ist das Ergebnis der Validierung positiv, wird der Inhalt des Arbeitsbereichs in die Datenbasis übertragen.

2. und 3. werden ununterbrechbar ausgeführt („**Valwrite-Phase**“).

Optimistische Verfahren (1)

Drei Transaktionsphasen:



Hört sich harmlos an, ist aber implementierungstechnisch nicht ganz unproblematisch. Realisierung z.B. durch Sperren während dieser Phase.

1. **Read-Phase:** Die Transaktion wird ausgeführt, ab *w*-Aktionen wirken nur auf lokalen Arbeitsbereich. (Ke Verzögerungen anderer Transaktionen!)
2. **Validation-Phase:** Eine commit-willige TA wird valid d.h. es wird geprüft, ob sie *CSR* abgelaufen ist.
3. **Write-Phase:** Ist das Ergebnis der Validierung positiv, ird der Inhalt des Arbeitsbereichs in die Datenbasis übertra gen.

2. und 3. werden *ununterbrechbar* ausgeführt („**Valwrite-Phase**“).

Optimistische Verfahren (2)

Satz 6.32

Sei *G* ein DAG. Wenn ein neuer Knoten ohne ausgehende Kanten zu *G* hinzugefügt wird, so ist auch der resultierende Graph azyklisch.

Wir werden den Satz wie folgt nutzen: Die Validierungsprotokolle werden überprüfen, ob die zu validierende Transaktion einen azyklischen Konfliktgraphen azyklisch belässt $\Rightarrow$ Transaktion darf nur einen Knoten ohne ausgehende Kanten hinzufügen.

Definition 6.33 (Read-Set/Write-Set)

- Das **Read-Set *RS*(*t*)** ist die Menge aller von *t* gelesenen Datenelemente.
- Das **Write-Set *WS*(*t*)** ist die Menge aller von *t* geschriebenen Datenelemente.

Optimistische Verfahren (3)

Zwei Alternativen:

- Rückwärtsvalidierung (backward-oriented optimistic concurrency control, BOCC)
  - zu validierende Transaktion wird gegen die schon abgeschlossenen Transaktionen getestet
- Vorwärtsvalidierung (forward-oriented optimistic concurrency control, FOCC)
  - zu validierende Transaktion wird gegen die gleichzeitig ablaufenden Transaktionen, die sich noch in der Lesephase befinden, getestet.

BOCC (1)

BOCC validation of t_j:

- Compare t_j to all previously committed t_i
- Accept t_j if one of the following holds
  - t_i has ended before t_j has started, or
  - $RS(t_j) \cap WS(t_i) = \emptyset$

Theorem 6.34:

$Gen\ (BOCC) \subset\ CSR$

Proof:

- Assume that $G(CP(s))$ is acyclic.
- The new node is last in the serialization order, and has no outgoing edges.

BOCC (2)

Diese Transaktionen wurden bereits validiert, das heißt, bis zu diesem Zeitpunkt ist *G* azyklisch.

BOCC validation of t_j:

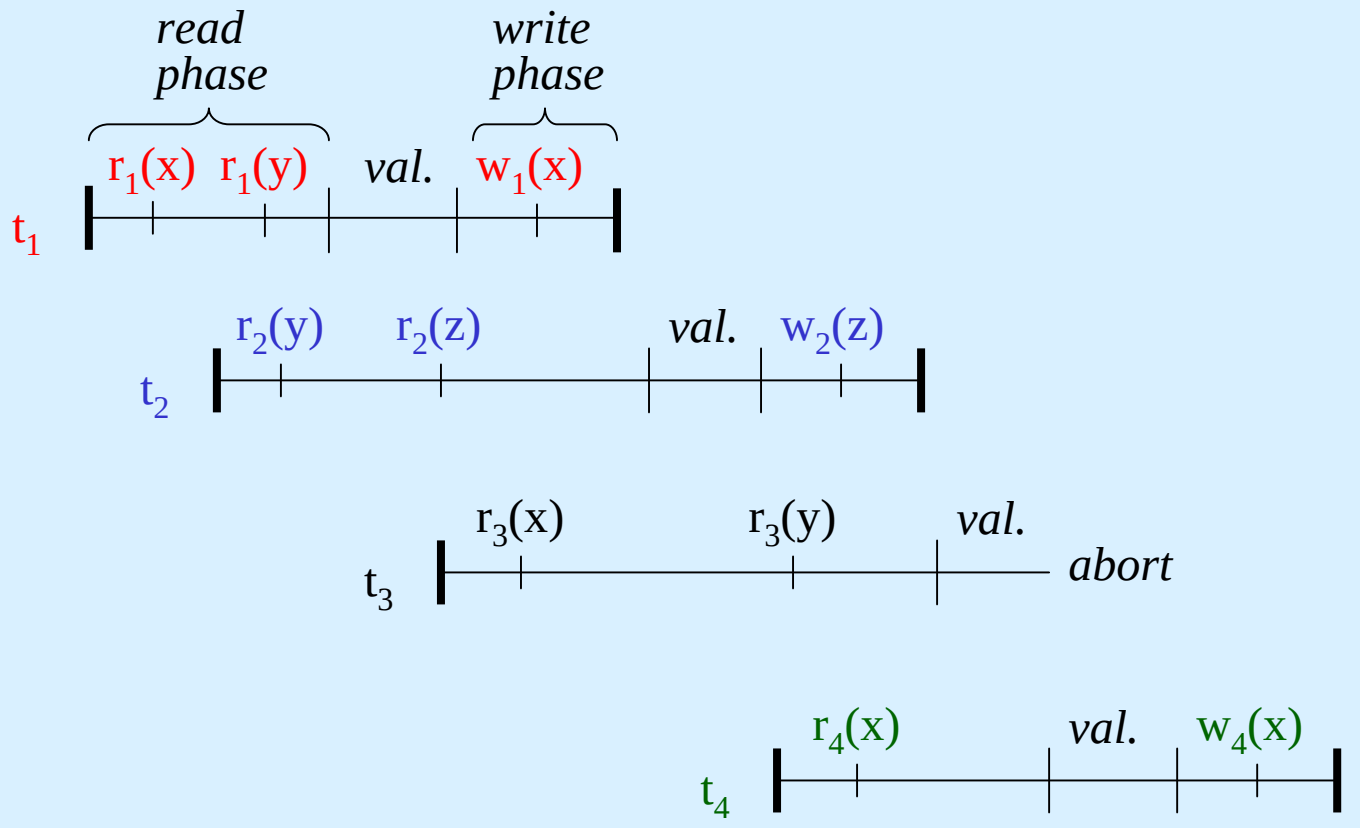
- Compare t_j to all previously committed t_i
- Accept t_j if one of the following holds
  - t_i has ended before t_j has started, or
    $RS(t_j) \cap WS(t_i) = \varnothing$

Falls Konflikt zwischen t_i und t_j existiert, gibt es die Kante (t_i,t_j). Die umgekehrte Kante kann es nicht geben.

t_j hatte keine Gelegenheit von t_i zu lesen. Es kann also wieder keine Kante (t_j,t_i) geben.

BOCC (3)

Example 6.35



FOCC (1)

FOCC validation of t_j:

- Compare t_j to all concurrently active t_i
  - (which must be in their read phase)
- Accept t_j if $WS(t_j) \cap RS^*(t_i) = \emptyset$
  - where $RS^*(t_i)$ is the current read set of t_i

Theorem 6.36:

$Gen\ (FOCC) \subset\ CSR$

Proof:

- Assume that $G(CP(s))$ is acyclic.
- Adding a newly validated transaction can only insert edges going into the new node, but no outgoing edges because for all previously committed t_k the following holds:
  - If t_k committed before t_j started, then no edge (t_j,t_k) is possible.
  - If t_j was in its read phase while t_k validated, then $WS(t_k)$ must be disjoint with $RS^*(t_j)$ and all later reads of t_j and all writes of t_j must follow t_k, so no edge (t_j,t_k) is possible.

FOCC (2)

FOCC validation of t_j:

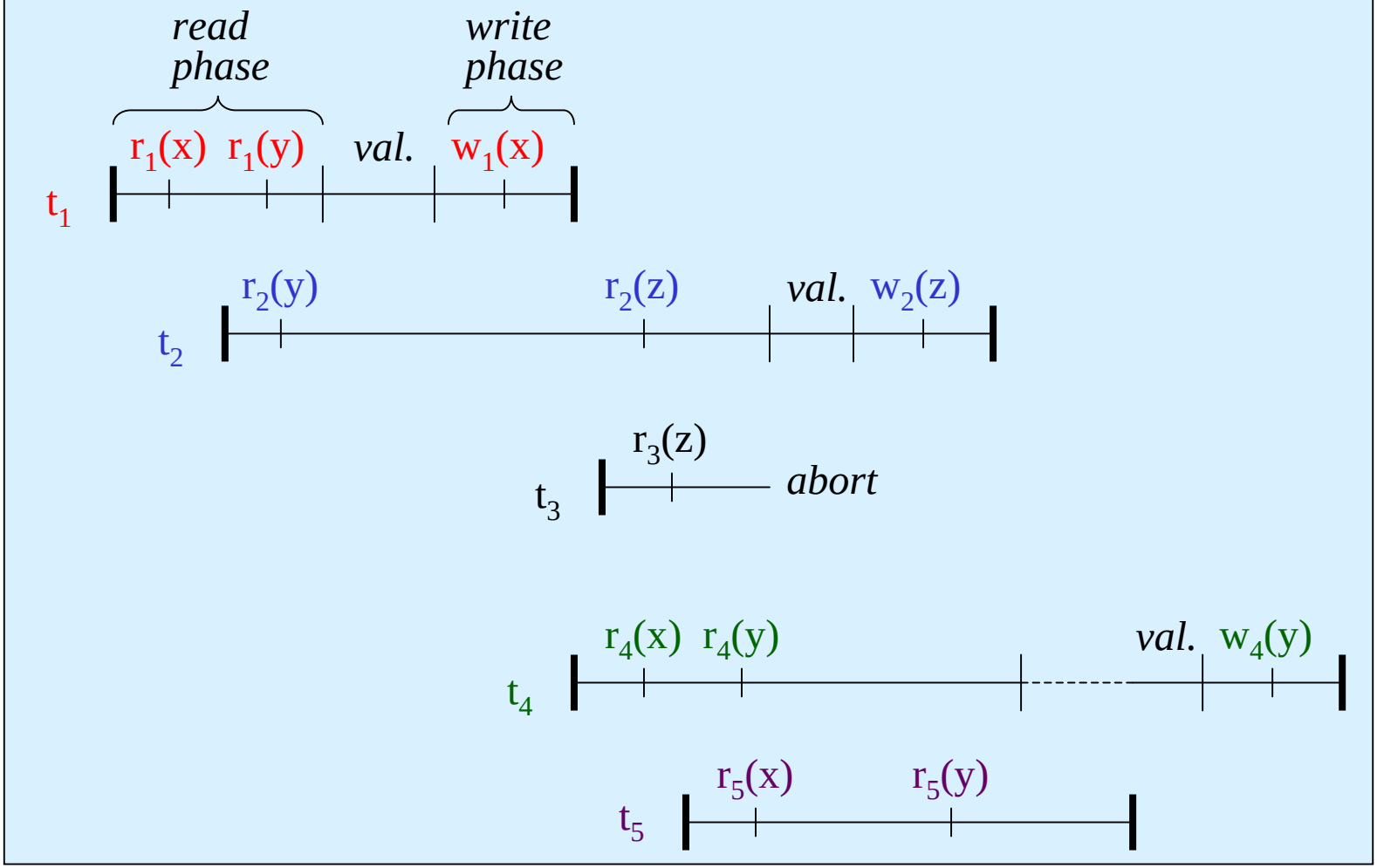
- Compare t_j to all concurrently active t_i
  - (which must be in their read phase)
- Accept t_j if $WS(t_j) \cap RS^*(t_i) = \emptyset$
  - where $RS^*(t_i)$ is the current read set of t_i

Remarks:

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

FOCC (3)

Example 6.37



DBS

Universität Karlsruhe (TH)

Chapter 7

Synchronization: Multiversion Concurrency Control

Multiversion serializability

Motivation (1)

Example 7.1:

$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ c_2\ w_1(z)\ c_1$ $\rightarrow \notin$ CSR

non-repeatable (inconsistent) read

$s = r_1(x)\ w_1(x)\ r_1(y)\ w_1(z)\ c_1\ r_2(x)\ w_2(y)\ c_2$ $\rightarrow \in$ CSR (following 2PL)

no concurrency $\Rightarrow$ inefficient!

$s = r_1(x)\ w_1(x)\ r_2(x)\ r_1(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$ $\rightarrow \in$ CSR

repeatable (consistent) read

Idea for better concurrency:

$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ c_2\ w_1(z)\ c_1$

$s = r_1(x)\ w_1(x)\ r_2(x)\ r_1(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$

Obtain this effect by reading state(y) at beginning of t_1

Motivation (2)

Basic solution:

Each write produces a new version of a data element.

- Extend construction of *CSR* histories by maintaining several „versions“ of *x* and *y*.

Which version to read?

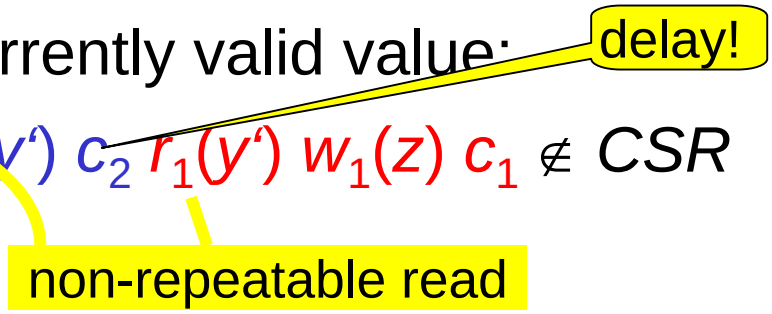
- Example 1: Read the value valid at the beginning of the transaction:

  $s' = r_1(x)\ w_1(x')\ r_2(x)\ w_2(y')\ r_1(y)\ c_2\ w_1(z)\ c_1$

  $G(h') = t_1\ t_2 \Rightarrow h' \in CSR$

- Example 2: Read the currently valid value:

  $s'' = r_1(x)\ w_1(x')\ r_2(x)\ w_2(y')\ c_2\ r_1(y')\ w_1(z)\ c_1 \notin CSR$



Motivation (3)

- Increased concurrency because multiversion schedulers can generate (additional) schedules that are impossible if only a single version is available.
- Multiversion schedulers are part of many commercial database management systems.
- *Two-version* schedules versions even come for free:
  - To achieve rollback under recovery, database must register the value of a data element before it is subjected to change $\Rightarrow$ at the minimum there are always two versions of an element.

Multiversion histories (1)

We start with the **general case** of an unlimited number of versions:

- Each write produces a new version of a data element.
  - The older versions are not overwritten but are kept in the database.
  - Given data element x, then x_i, x_j, … denote versions of x where the index refers to the transaction that wrote the version.
    - Each write of x in a multiversion schedule is of the form $w_i(x_i)$.
- Each read has a free choice of the version it wishes to access.
  - Read is denoted by $r_i(x_j)$.
    - We need initial versions $\Rightarrow$ transaction t_0.

Multiversion histories (2)

Definition 7.2 (Version function):

Let h be a (version free) history with initialization transaction t_0 (write of initial elements).
A **version function** for h is a function f which translates

1. $w_i(x)$ to $w_i(x_i)$,
2. $r_i(x)$ to $r_i(x_j)$ for some j,
3. c_i to c_i and
4. a_i to a_i

We leave open which one
$\Rightarrow$ leaves some latitude for
the ordering of operations!

Multiversion histories (3)

Definition 7.3:

A **multiversion (mv) history** for transactions $T = \{t_1, ..., t_n\}$ is a history $m=(op(m), <_m)$, $<_m$ an order on $op(m)$, with the (additional) properties

(1) $op(m) = \bigcup_{i=1}^{n} f(op(t_i))$ for some version function f,

(2) for all t_i and all $p, q \in op(t_i)$: $p <_{t_i} q \Rightarrow f(p) <_m f(q)$,

(3) $f(r_j(x)) = r_j(x_i) \succ w_i(x_i) <_m r_j(x_i)$

(4) $f(r_j(x)) = r_j(x_i), i \neq j, c_j \in m \succ c_i \in m \wedge c_i <_m c_j$.

A **multiversion (mv) schedule** is a prefix of a [illegible]ersion history.

For each operation of a transaction there is a versioned operation in the mv history.

The version function must maintain the order of operations within a transaction.

Values read by a successful transaction must have been valid.

A bit weaker than before!

Multiversion histories (4)

Remember

$s' = r_1(x)\ w_1(x')\ r_2(x)\ w_2(y')\ r_1(y)\ c_2\ w_1(z)\ c_1$

Rewritten (t_0 ignored):

$m = r_1(x_0)\ w_1(x_1)\ r_2(x_0)\ w_2(y_2)\ r_1(y_0)\ c_2\ w_1(z_1)\ c_1$

where $f(w_i(el)) = w_i(el_i)$

$f(r_1(y)) = r_1(y_0)$

$f(r_2(x)) = r_2(x_0)$

$t_1 \quad t_2$

Take instead:

$m = r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(z_1)\ c_1\ c_2$

where $f(w_i(el)) = w_i(el_i)$

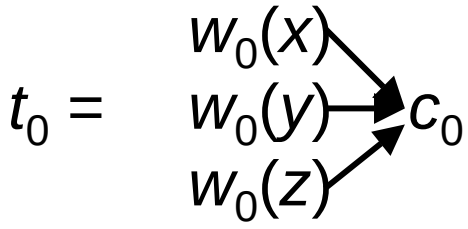
$f(r_1(y)) = r_1(y_0)$

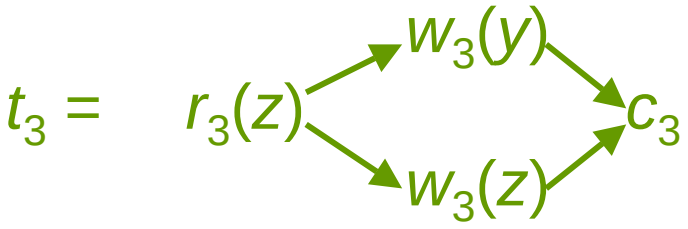
$f(r_2(x)) = r_2(x_1)$

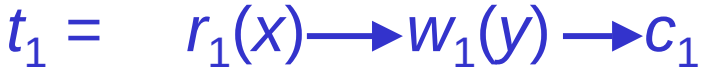
$t_1 \rightarrow t_2$

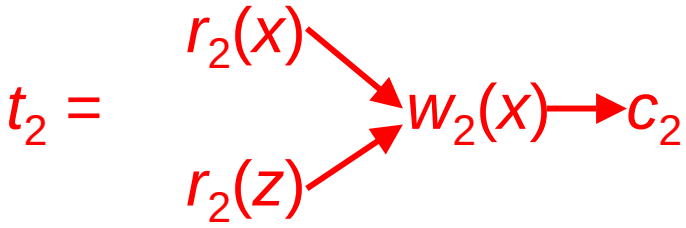
Multiversion histories (5)

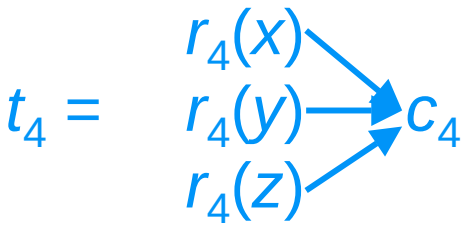
Take transactions:





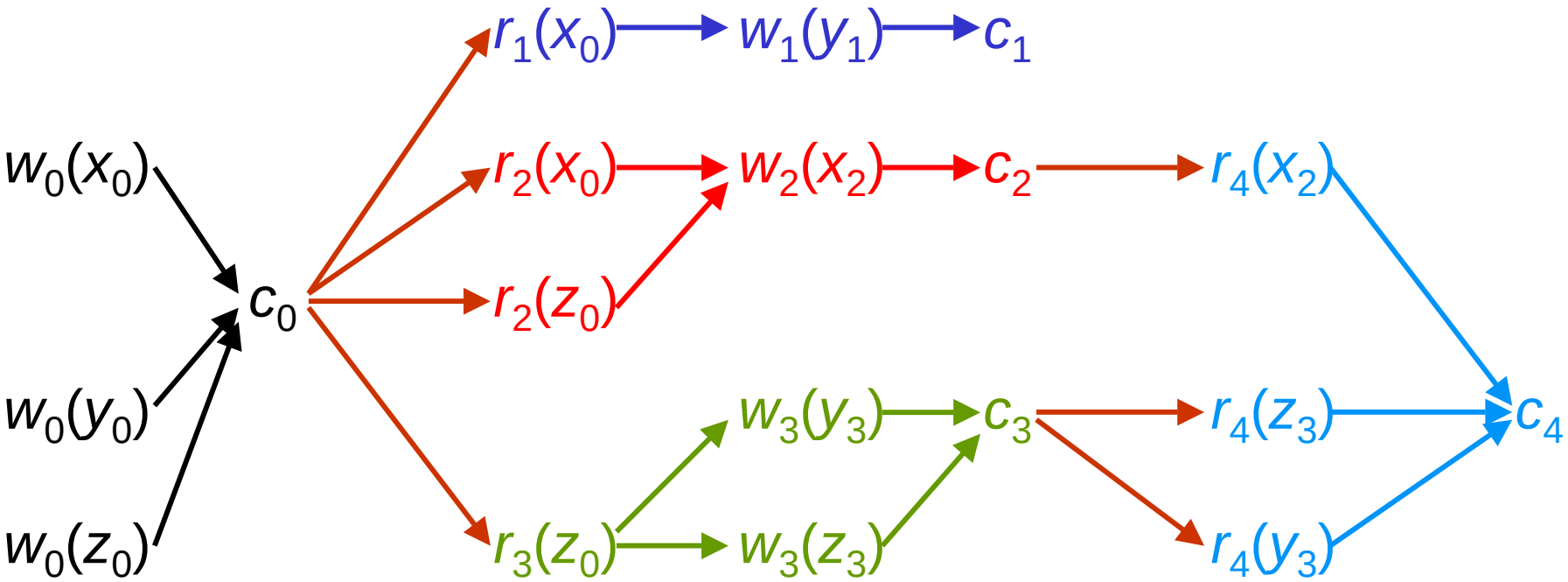






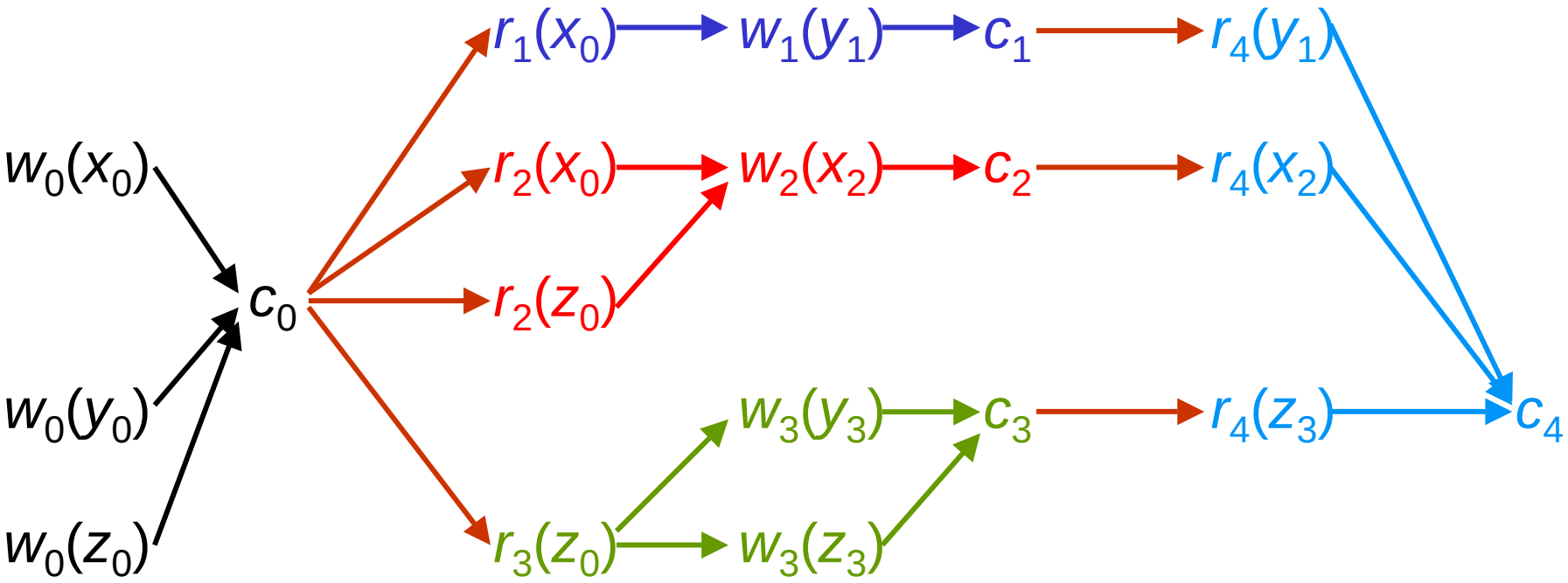
Multiversion histories (6)

Example 7.4: h_6 is a mv history over $\{t_0,\ldots,t_4\}$:



Multiversion histories (7)

Example 7.5: h_7 is a mv history over $\{t_0,\ldots,t_4\}$ with $r_4(y_3)$ changed to $r_4(y_1)$:

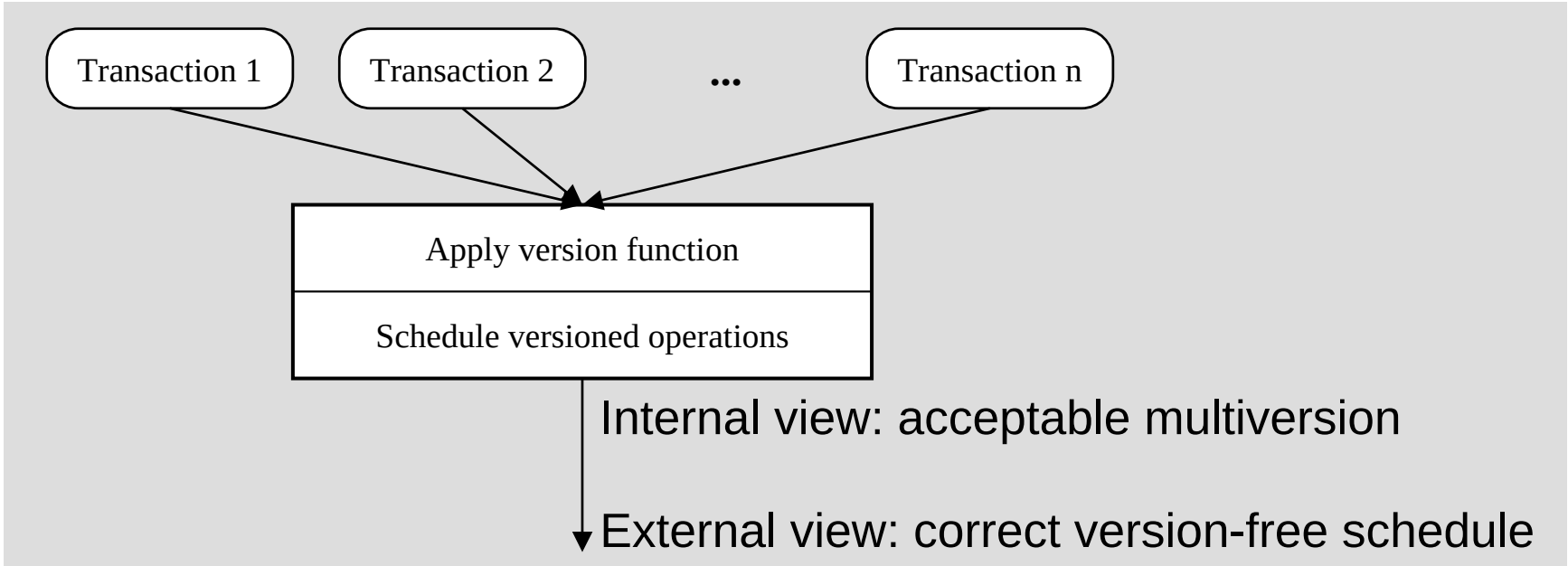


Multiversion view serializability (1)

Which one is “correct”, h_6 or h_7?

What do we mean by “correct”?

- Internal view: Versioning is strictly internal to the database system to allow for better concurrency.
- External view: Versioning must remain entirely transparent to the application ⇒ versioning is externally not visible.



Multiversion view serializability (2)

Intuitive interpretation of a version-free history:

- **Definition 7.6: Monoversion history**

  A multiversion history is called a **monoversion history** if its version function maps each read step to the last preceding mv compliant write step on the same data item.

$h = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ c_2\ w_1(z)\ c_1$

Monoversion: $m = r_1(x_0)\ w_1(x_1)\ r_2(x_0)\ w_2(y_2)\ r_1(y_2)\ c_2\ w_1(z_1)\ c_1$

$h = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Monoversion: $m = r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(z_1)\ c_1\ c_2$

Multiversion view serializability (3)

- Conversely, a monoversion can easily be translated to a version-free history: Just omit the indices.

Monoversion: $m = r_1(x_0)\ w_1(x_1)\ r_2(x_0)\ w_2(y_2)\ r_1(y_2)\ c_2\ w_1(z_1)\ c_1$
Version-free: $h = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ c_2\ w_1(z)\ c_1$

Monoversion: $m = r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(z_1)\ c_1\ c_2$
Version-free: $h = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

Multiversion view serializability (4)

"Acceptable" multiversion history:

Definition 7.7 (Reads-from Relation):

For mv schedule m the reads-from relation of m, $RF(m)$,
$t_j \triangleright_m(x)\ t_i \prec\succ t_j \triangleright_m(x_i)\ t_i$

Definition 7.8 (View Equivalence):

mv histories m and m' with $trans(m)=trans(m')$ are **view equivalent**, $m \approx_v m'$, if $RF(m) = RF(m')$.

If we do not limit the number of versions, t_∞ is unnecessary!

Example 7.9: ($m \approx_v m'$)

$m = w_0(x_0)\ w_0(y_0)\ c_0\ w_1(x_1)\ c_1\ r_2(x_1)\ r_3(x_0)\ w_2(y_2)\ w_3(x_3)\ c_3\ c_2$
$m' = w_0(x_0)\ w_0(y_0)\ c_0\ r_3(x_0)\ w_3(x_3)\ c_3\ w_1(x_1)\ c_1\ r_2(x_1)\ w_2(y_2)\ c_2$

Multiversion view serializability (5)

Example 7.10:

Note: is not a monoversion!

Multiversion:

$m = w_0(x_0)\ w_0(y_0)\ c_0\ w_1(x_1)\ c_1\ r_2(x_1)\ r_3(x_0)\ w_2(y_2)\ w_3(x_3)\ c_3\ c_2$

Equivalent serial multiversion:

$m' = w_0(x_0)\ w_0(y_0)\ c_0\ r_3(x_0)\ w_3(x_3)\ c_3\ w_1(x_1)\ c_1\ r_2(x_1)\ w_2(y_2)\ c_2$

Equivalent serial monoversion:

$m'' = w_0(x_0)\ w_0(y_0)\ c_0\ r_3(x_0)\ w_3(x_3)\ c_3\ w_1(x_1)\ c_1\ r_2(x_1)\ w_2(y_2)\ c_2$

Version-free history:

$h'' = w_0(x)\ w_0(y)\ c_0\ r_3(x)\ w_3(x)\ c_3\ w_1(x)\ c_1\ r_2(x)\ w_2(y)\ c_2$

Multiversion view serializability (6)

Example 7.11:

Is even a monoversion!

Multiversion:

$m = w_0(x_0)\ w_0(y_0)\ c_0\ r_1(x_0)\ r_1(y_0)\ r_2(x_0)\ w_1(x_1)\ w_1(y_1)\ c_1\ r_2(y_1)\ c_2$

Is no longer a monoversion!

Serial multiversion

$m' = w_0(x_0)\ w_0(y_0)\ c_0\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_0)\ r_2(y_1)\ c_2$

But no equivalent serial monoversion can be found (neither/nor):

$m'' = w_0(x_0)\ w_0(y_0)\ c_0\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_1)\ r_2(y_1)\ c_2$

$m''' = w_0(x_0)\ w_0(y_0)\ c_0\ r_2(x_0)\ r_2(y_0)\ c_2\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1$

$\Rightarrow$ no transparent versioning!

Existence of a serial multiversion is not a shortcut!

Multiversion view serializability (7)

Definition 7.12 (Multiversion View Serializability):

m is **multiversion view serializable** if there is a serial monoversion history *m*‘ for the same set of transactions s.t. $m \approx_v m'$.

MVSR is the class of multiversion view serializable histories.

Multiversion view serializability (8)

Theorem 7.13:

$VSR \subset MVSR$

$m \in MVSR$ because $t_1 \rightarrow t_2$

Monoversion: $m = r_1(x_0)\ w_1(x_1)\ r_2(x_0)\ w_2(y_2)\ r_1(y_2)\ c_2\ w_1(z_1)\ c_1$

Version-free: $h = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ c_2\ w_1(z)\ c_1$

Hence $h \in MVSR$ but $h \notin CSR$ and $h \notin VSR$

Theorem 7.14:

The problem of deciding whether a given multiversion history is in *MVSR* is NP-complete.

Multiversion serialization graph (1)

Graph-theoretic proof:

We know the order in which versions were produced.

$\Rightarrow$ Check whether the order imposed on the write operations by the scheduler with the effect of a specific version order defines an equivalent monoversion history.

Definition 7.15 (Version order)

If *x* is a data item, a **version order for** *x* is any nonreflexive and total ordering of all versions of x that are written by operations in mv schedule *m*. A **version order** << **for** *m* is the union of all version orders of data items written by operations in *m*.

Multiversion serialization graph (2)

The only conflicts possible in a multiversion history are *wr*.
$\Rightarrow$ A conventional conflict graph $G(m)$ has an edge from t_i to t_j simply if $r_j(x_i)$ is in m.

Theorem 7.16:

For any two mv schedules m, m', $m \approx_v m' \succ G(m) = G(m')$.

Multiversion serialization graph (2)

Definition 7.17 (MVSG)

For a given mv schedule m and a version order <<, the **multiversion serialization graph MVSG**(m, <<) is the conflict graph $G(m) = (V,E)$ with the following edges added for each $r_k(x_j)$ und $w_i(x_i)$ in $CP(m)$ where $i \neq j \neq k$:

if $x_i << x_j$, then add $t_i \rightarrow t_j$, else $t_k \rightarrow t_i$.

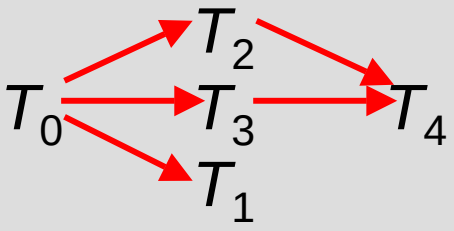
$\Rightarrow t_i < t_j < t_k$
($t_j \rightarrow t_k$ already in $G(m)$)

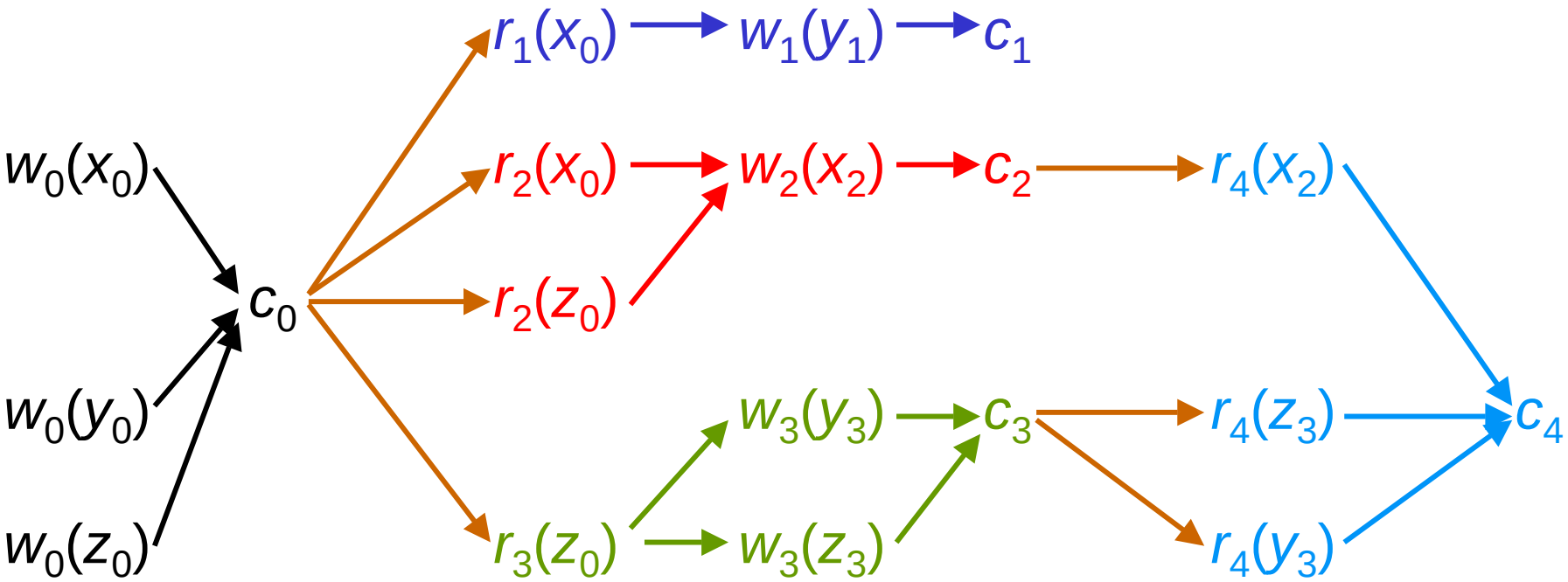
monoversion assumption
$\Rightarrow t_j < t_k < t_i$
($t_j \rightarrow t_k$ already in $G(m)$)

Multiversion serialization graph (3)

$x_0 << x_2$, $y_0 << y_1 << y_3$, $z_0 << z_3$
or
$x_0 << x_2$, $y_0 << y_3 << y_1$, $z_0 << z_3$

$G(m) =$

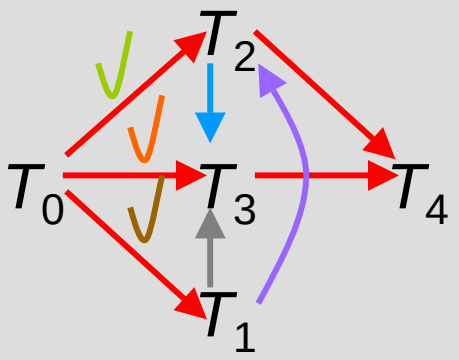


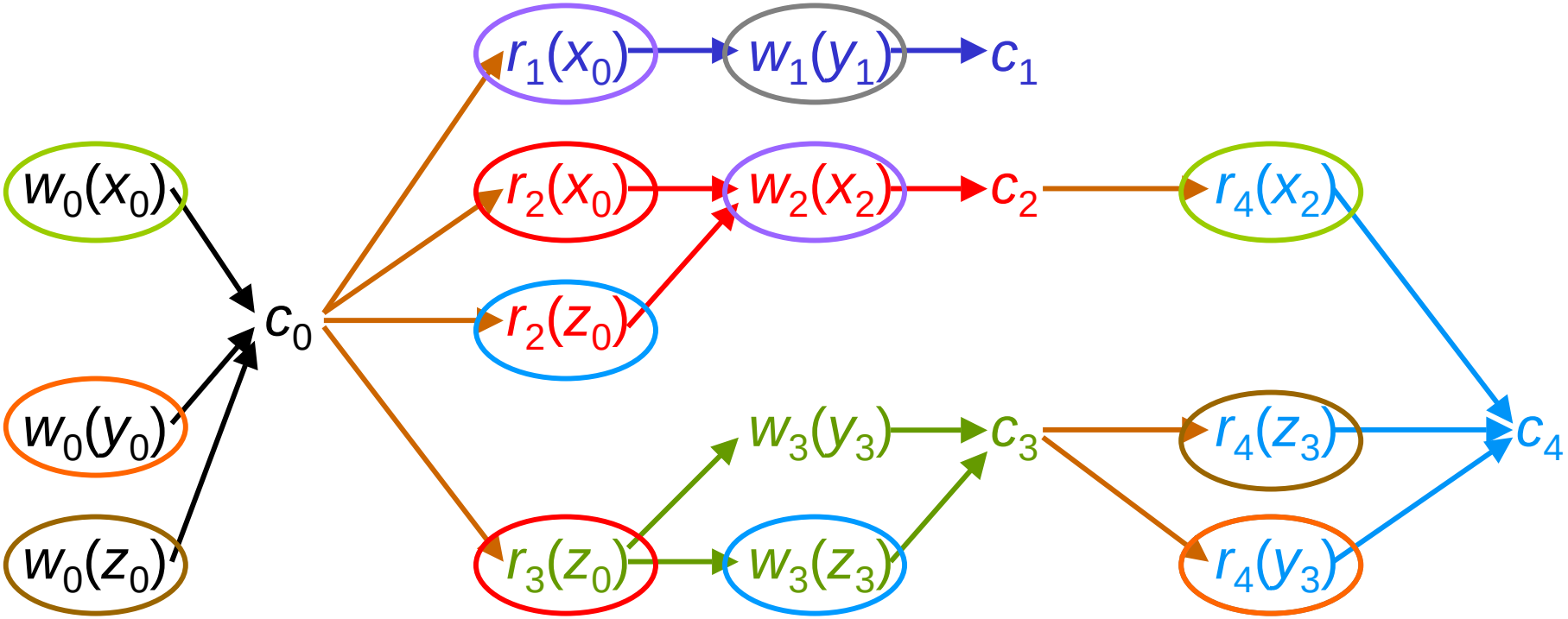


Multiversion serialization graph (4)

$x_0 << x_2$, $y_0 << y_1 << y_3$, $z_0 << z_3$

$G(m) =$





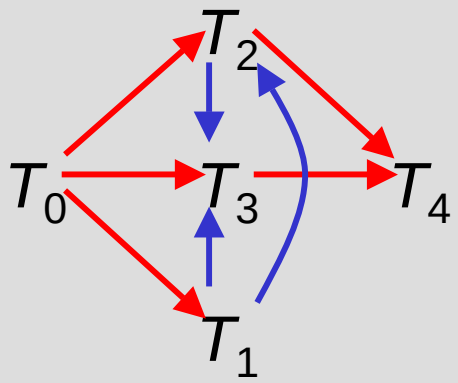
Multiversion serialization graph (5)

Theorem 7.18 (MVSR)

mv schedule *m* is in *MVSR* $\prec\succ$ there exists a version order << s.t. *MVSG*(*m*,<<) is acyclic.

acyclic!

MVSG(*m*,<<) =



Similar to VSR: It cannot necessarily be tested in polynomial time whether a version order of the desired form exists.

Is there some chance to extend *CSR*?

Multiversion conflict serializability (1)

- **Approach**: Which types of conflict are relevant to mv schedules, i.e., for which pairs of operations do we have to watch out for the same order in the original schedule and in the serial schedule?
  - *ww*: not a conflict because a new version is written.
  - *wr*: can be exchanged except where *r* accesses the version written by *w*.
  - *rw*:critical because an exchange opens up to *r* a larger choice among versions.

Definition 7.19 (Multiversion Conflict):
A **multiversion conflict** in m is a pair $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$ $(i \neq j \neq k)$.

Multiversion conflict serializability (2)

- Remember: *CSR* was (also) defined on the basis of reordering, using commutativity-based reducibility.
- We take the same approach (here for the special case of a total order).

Definition 7.20 (Multiversion Reducibility):
A (totally ordered) mv history is **multiversion reducible** if it can be transformed into a serial monoversion history by exchanging the order of adjacent steps other than multiversion conflict pairs.

Definition 7.21 (Multiversion Conflict Serializability):
A mv history *m* is **multiversion conflict serializable** if there is a serial monoversion history for the same set of transactions in which all pairs of operations in multiversion conflict occur in the same order as in *m*.

Multiversion conflict serializability (3)

Graph-theoretic characterization:

Definition 7.22 (Multiversion Conflict Graph):
Let m be a mv history. The **multiversion conflict graph** $G(m) = (V, E)$ is a directed graph with vertices $V = \text{commit}(m)$ and edges $E = \{(t_i,t_k) \mid i \neq j\}$ if there are steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

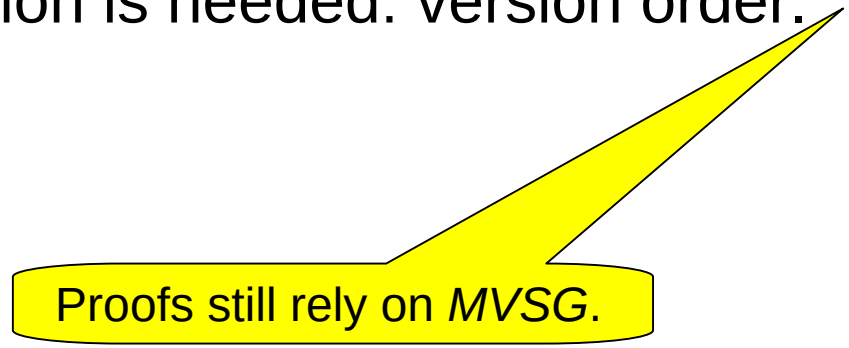
Theorem 7.23:
m is in MCSR $\prec\succ$ m is multiversion reducible $\prec\succ$ m's mv conflict graph is acyclic.

tested in polynomial time.

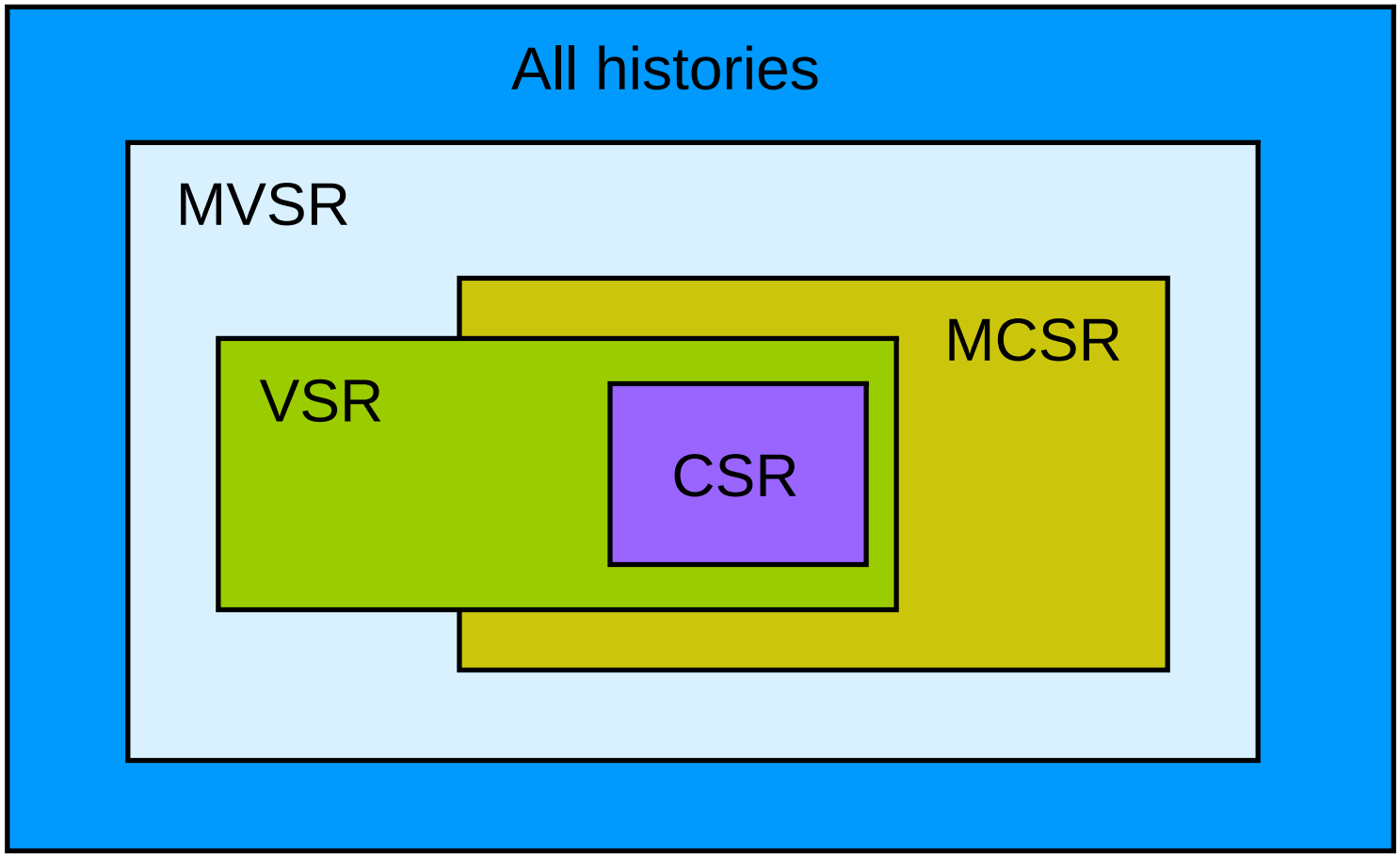
Multiversion conflict serializability (3)

Remark 7.24:

- *MCSR*: class of all multiversion conflict serializable histories.
- $MCSR \subset MVSR$
- *MCSR* has a polynomial membership test.
- The equivalent serial monoversion history cannot simply be derived by topologically sorting the graph. Further information is needed: version order.



Summary



Multiversion schedulers

Multiversion 2PL (MV2PL)

- Protocol uses SS2PL.
- Modifications:
  - a simpler conflict matrix,
  - but version order must be tracked in addition.
- We restrict ourselves to the particularly simple case of 2 versions (2V2PL).

2-version 2PL (1)

Adjustments to 2PL:

- 2PL: A *wl* lock is exclusive. In particular, it prohibits concurrent reads. By using two versions we wish to relax the condition.
- If t_i writes x:
  - a new version x_i is created;
  - a lock must be found for x that inhibits reads on x_i and the creation of further versions for x (2-version 2PL!), and allows other transactions to read the single old x.
- On commit of t_i x_i becomes the current version and the old version of x becomes inaccessible.
- Conclusion: Use 2PL for ww synchronization, and version selection für rw synchronization.

2-version 2PL (2)

Adjustments to the locks:

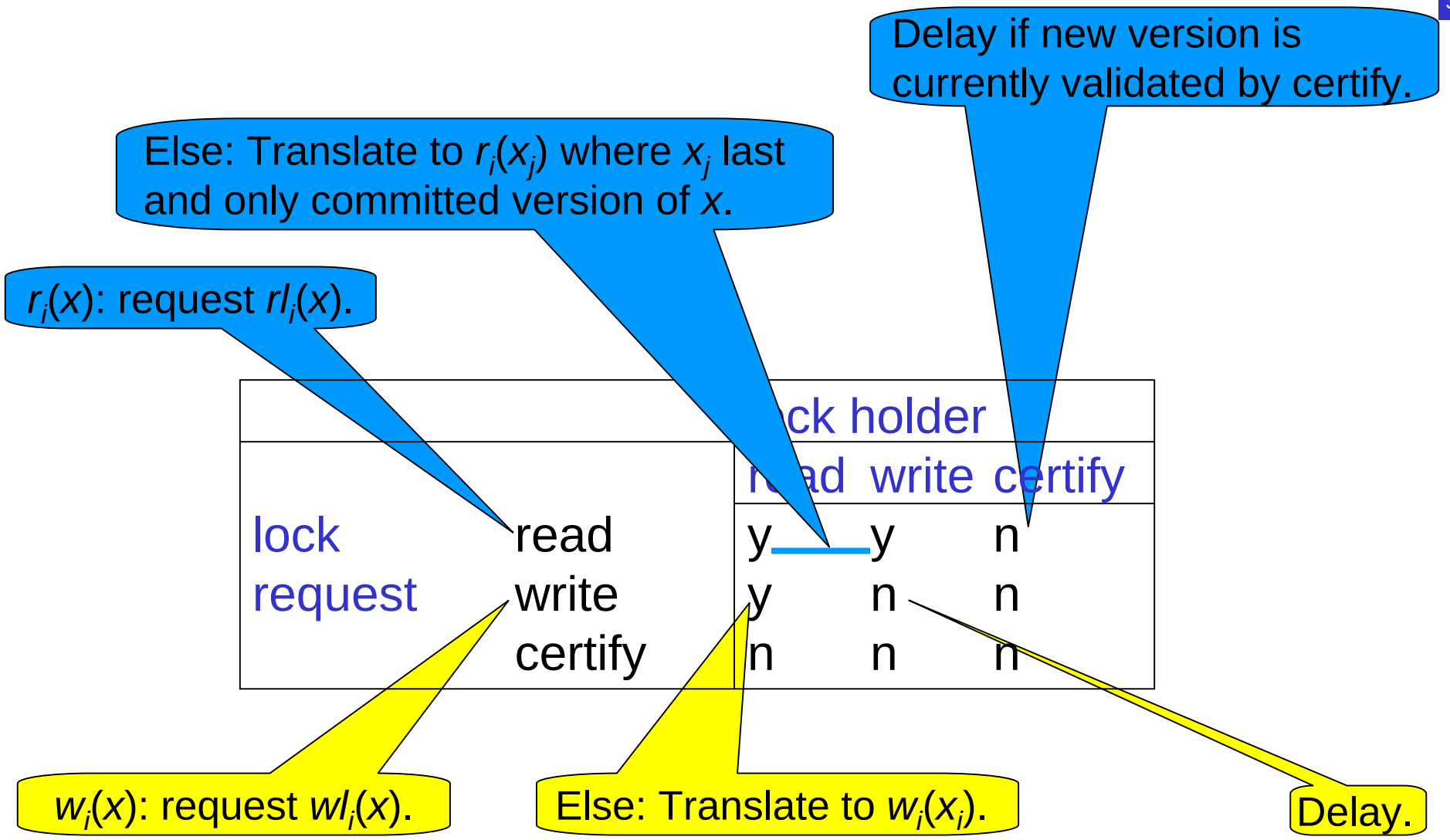
2V2PL uses 3 kinds of locks: read, write and certify (or commit) locks.

Lock control:

- As before for *rl* und *wl*, controlled by the compatibility matrix.
- On commit: all *wl* locks turn into certify locks (*cl*).

		lock holder		
		read	write	certify
lock request	read	y	y	n
	write	y	n	n
	certify	n	n	n

2-version 2PL (3)



2-version 2PL (4)

Only disadvantage: Wait for the end of concurrent readers on commit of a write transaction.

Certify locks behave like write locks in 2PL, however the time over which they are held is much shorter ⇒ advantage over 2PL. Before certify: write + concurrent reads.

lock request	read	write	certify
read	y	y	n
write	y	n	n
certify	n	n	n

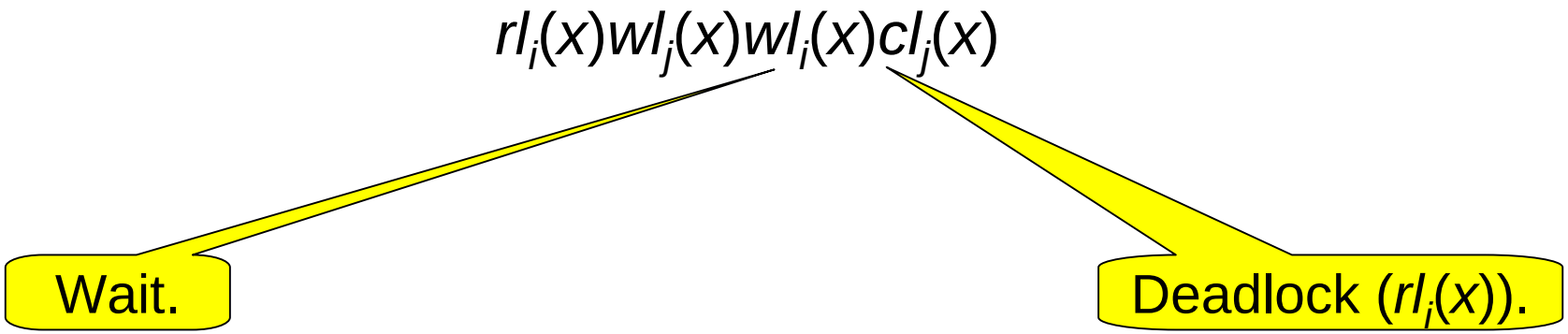
c_i.

Wait until all readers have finished.

Change all write locks to certify locks. (There can be no write locks or certify locks held by other transactions.) After changing all locks and making the new versions persistent: Release locks.

2-version 2PL (5)

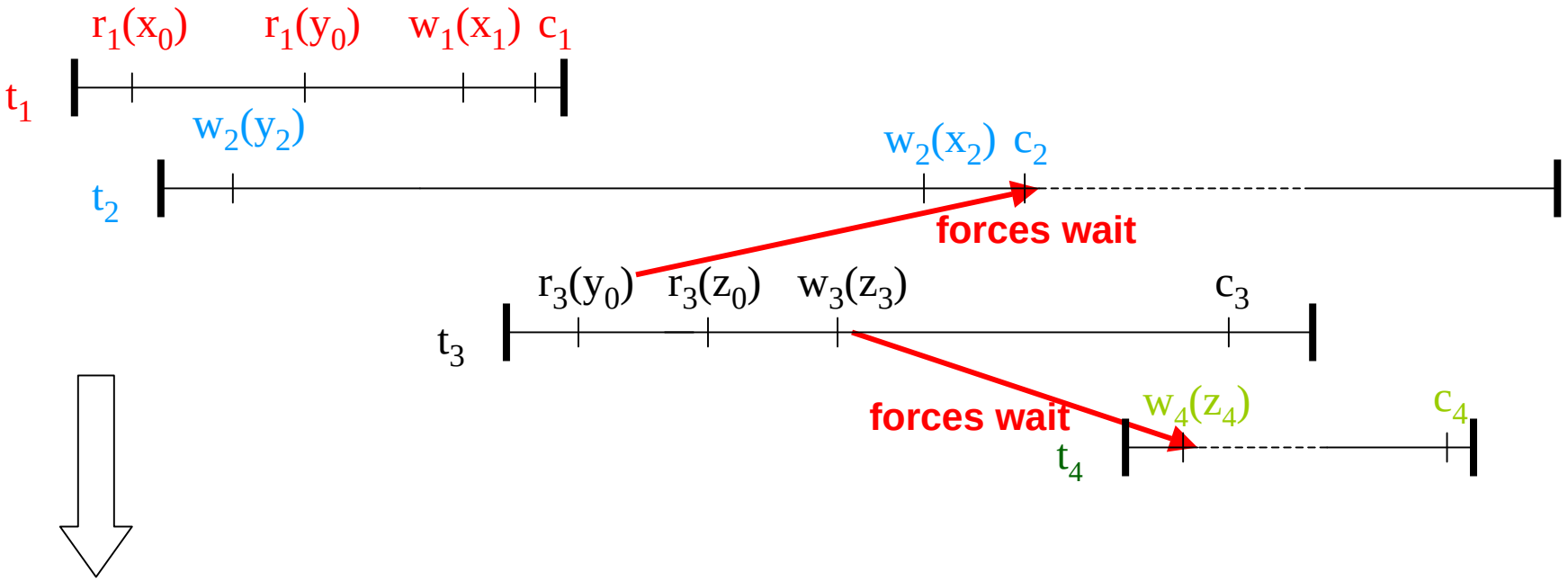
- Lock escalation $rl \rightarrow wl$ is possible and may be a major source for deadlocks.



2V2PL Example

Example 7.25:

arrival order = $r_1(x)\ w_2(y)\ r_1(y)\ w_1(x)\ c_1\ r_3(y)\ r_3(z)\ w_3(z)\ w_2(x)\ c_2\ w_4(z)\ c_4\ c_3$



$rl_1(x)\ r_1(x_0)\ wl_2(y)\ w_2(y_2)\ rl_1(y)\ r_1(y_0)\ wl_1(x)\ w_1(x_1)\ cl_1(x)\ u_1\ c_1$
$rl_3(y)\ r_3(y_0)\ rl_3(z)\ r_3(z_0)\ wl_2(x)\ cl_2(x)\ wl_3(z)\ w_3(z_3)\ cl_3(z)\ u_3\ c_3$
$cl_2(y)\ u_2\ c_2\ wl_4(z)\ w_4(z_4)\ cl_4(z)\ u_4\ c_4$

Correctness of 2V2PL (1)

Theorem 7.26

Gen(2V2PL) $\subset$ *MCSR*

Correctness of 2V2PL (2)

f_i : certify

2V2PL1: $\forall t_i: r_i(x_j), w_i(x_i) <_m f_i \wedge f_i <_m c_i$

2V2PL2: $\forall r_k(x_j) \in CP(m), j \neq k: c_j <_m r_k(x_j)$

Es werden nur freigegebene Versionen gelesen.

2V2PL3: $\forall r_k(x_j), w_i(x_i) \in CP(m), j \neq k: f_i <_m r_k(x_j) \vee r_k(x_j) <_m f_i$

Die Zertifizierungen aller *x* schreibenden Transaktionen müssen mit den *x* lesenden Operationen geordnet sein.

2V2PL4: $\forall r_k(x_j), w_i(x_i) \in CP(m), i \neq j \neq k: f_i <_m r_k(x_j) \succ f_i <_m f_j$

Zusammen mit 2V2PL2: jedes Lesen $r_k(x_j)$ liest die letzte zertifizierte Version von *x*.

2V2PL5: $\forall r_k(x_j), w_i(x_i) \in CP(m), i \neq j, i \neq k: r_k(x_j) <_m f_i \succ f_k <_m f_i$

Eine *x* schreibende t_i kann nicht zertifiziert werden, solange nicht alle Transaktionen, die *x* vorher lasen, zertifiziert wurden.

2V2PL6: $\forall w_i(x_i), w_j(x_j) \in CP(m): f_i <_m f_j \vee f_j <_m f_i$

Zertifizierungen von Transaktionen, die beide ein *x* schreiben, sind wechselseitig atomar.

Correctness of 2V2PL (2)

Beweisskizze:

Definiere eine Versionsordnung << mit $x_i << x_j \prec\succ f_i <_m f_j$.

2VPL6 $\succ$ << ist eine Versionsordnung.

Falls wir beweisen: $t_i \rightarrow t_j \in MVSG(m,<<) \succ f_i <_m f_j$:

Da $f_i <_m f_j$ eine Teilordnung von m und per Definition azyklisch ist, gilt: $MVSG(m,<<)$ ist azyklisch.

$\succ$ Behauptung.

Correctness of 2V2PL (3)

Conflict graph $G(m)$:

Betrachte $t_i \rightarrow t_j \in G(m)$

$\succ \exists x \;\; t_j \triangleright_m(x)\; t_i$

$\succ$ [2V2PL2] $c_i <_m r_j(x_i)$

mit [2V2PL1] $r_j(x_i) <_m f_j$

$\succ$ [2V2PL1,Transitivität] $f_i <_m f_j$

Correctness of 2V2PL (4)

MV (version edges):

Betrachte eine Versionsordnungskante, die durch $w_i(x_i)$ und $r_k(x_j)$ $(i \neq j \neq k)$ induziert wurde. Zwei Fälle:

- $x_i << x_j$: $\succ$ Kante $t_i \rightarrow t_j \succ$ [def <<] $f_i <_m f_j$
- $x_j << x_i$: $\succ$ Kante $t_k \rightarrow t_i$

  $x_j << x_i \succ f_j <_m f_i$

  2V2PL3 $\succ f_i <_m r_k(x_j) \vee r_k(x_j) <_m f_i$

  Erstes Disjunkt: (2V2PL4) $\succ f_i <_m f_j$: Widerspruch!

  Also $r_k(x_j) <_m f_i$

  2V2PL5 $\succ f_k <_m f_i$

Mehrversionen-Zeitstempelordnung (1)

Annahme: Zahl der aufbewahrten Versionen ist unbegrenzt.

- Ein **Mehrversionen-TO-Scheduler** (MVTO) bearbeitet Operationen in Ankunftsreihenfolge.
- Er übersetzt Operationen auf Datenelemente in solche auf Versionen:

  $r_i(x)$ wird in $r_i(x_k)$ übersetzt, wobei x_k die Version mit dem größten Zeitstempel kleiner $ts(t_i)$ ist. $r_i(x_k)$ wird dann zum DV gesendet.

  Für $w_i(x)$ werden zwei Fälle unterschieden:

  1. Falls bereits ein $r_j(x_k)$ mit $ts(t_k) < ts(t_i) < ts(t_j)$ ausgeführt wurde, so wird $w_i(x)$ zurückgewiesen.
  2. Sonst wird $w_i(x)$ in $w_i(x_i)$ übersetzt und zum DV gesendet.
- Wegen Def. 7.3 (4) $f(r_j(x)) = r_j(x_i)$, $i \neq j$, $c_j \in m \succ c_i <_m c_j$ wird c_j solange verzögert, bis c_i für alle t_i mit $t_j \triangleright t_i$ ausgeführt wurde.

Mehrversionen-Zeitstempelordnung (2)

Leser werden nie zurückgewiesen. Schreiber werden nur zurückgewiesen, falls ihnen ein Leser folgt, der eine frühere Version gesehen hat.

$r_i(x)$ wird in $r_i(x_k)$ übersetzt, wobei x_k die Version mit dem größten Zeitstempel kleiner $ts(t_i)$ ist. $r_i(x_k)$ wird dann zum DV gesendet.

Für $w_i(x)$ werden zwei Fälle unterschieden:

1. Falls bereits ein $r_j(x_k)$ mit $ts(t_k) < ts(t_i) < ts(t_j)$ ausgeführt wurde, so wird $w_i(x)$ zurückgewiesen.

2. Sonst wird $w_i(x)$ in $w_i(x_i)$ übersetzt und zum DV gesendet.

1VTO weist Operationen, die zu spät kommen, zurück.

Dabei ist eine Operation $p_i(x)$ zu spät, falls gilt:

$\exists\ q_j(x) \in h \quad p_i(x) \nparallel q_j(x) \in h \wedge q_j(x) <_h p_i(x) \wedge ts(t_j) > ts(t_i)$

Da $q_j(x)$ in einem solchen Fall schon ausgeführt ist, muss $p_i(x)$ zurückgewiesen werden.

Leser und Schreiber, die zu spät kommen, werden zurückgewiesen.

Mehrversionen-Zeitstempelordnung (3)

Unterschiedliches Verhalten MVTO und 1VTO:

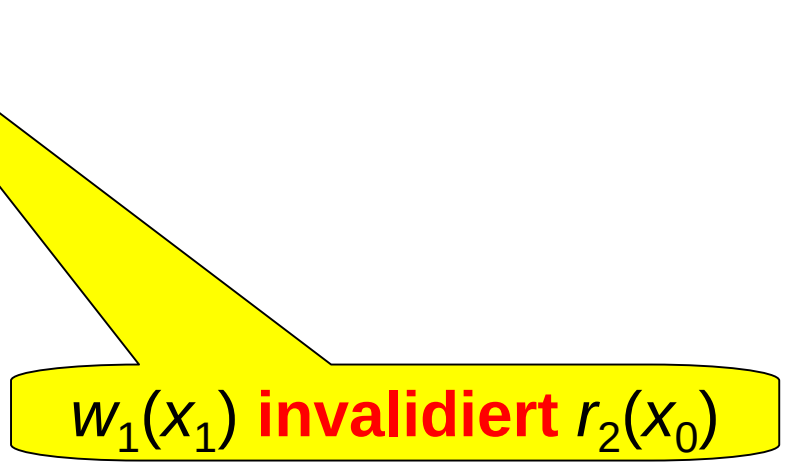
- Nur zulässig unter MVTO:

  $w_0(x_0) < w_2(x_2) < w_1(x_1)$

  $w_1(x_1) < r_2(x_1) < w_0(x_0)$

- Nicht zulässig unter MVTO:

  $w_0(x_0) < r_2(x_0) < w_1(x_1)$



MV-Zeitstempelordnung-Protokolle (1)

MVTO Versionsauswahl:

Zur Auswahl der zu lesenden Versionen und um Invalidierungen zu vermeiden, verwaltet der Scheduler Zeitstempelintervalle. Für jede Version x_i wird ein Zeitstempelintervall $interval(x_i) = [wts, rts]$ geführt, mit

- wts ist der Zeitstempel von x_i und
- rts ist der größte Zeitstempel eines Lesens von x_i. Falls kein solches Lesen existiert, so gilt $wts = rts$.

Sei $intervals(x) = \{interval(x_i) \mid x_i \text{ ist eine Version von } x\}$.

Zur Bearbeitung von $r_i(x)$ untersucht der Scheduler $intervals(x)$, um diejenige Version x_j zu finden, deren Intervall $[wts, rts]$ das größte wts kleiner $ts(t_i)$ hat. Falls $rts < ts(t_i)$, so wird rts auf $ts(t_i)$ gesetzt.

MV-Zeitstempelordnung-Protokolle (2)

MVTO Versionsauswahl:

Zur Auswahl der zu lesenden Versionen und um Invalidierungen zu vermeiden, verwaltet der Scheduler Zeitstempelintervalle. Für jede Version x_i wird ein Zeitstempelintervall $interval(x_i) = [wts, rts]$ geführt, mit

- wts ist der Zeitstempel von x_i und
- rts ist der größte Zeitstempel eines Lesens von x_i. Falls kein solches Lesen existiert, so gilt $wts = rts$.

Sei $intervals(x) = \{interval(x_i) \mid x_i \text{ ist eine Version von } x\}$.

Zur Bearbeitung von $w_i(x)$ untersucht der Scheduler $intervals(x)$, um diejenige Version x_j zu finden, deren Intervall $[wts, rts]$ das größte wts kleiner als $ts(t_i)$ hat. Falls $rts > ts(t_i)$, wird $w_i(x)$ zurückgewiesen, sonst wird $w_i(x_i)$ an den DV gesendet und ein neues Intervall $interval(x_i) = [ts(t_i), ts(t_i)]$ erzeugt.

MVTO-Korrektheit (1)

Eigenschaften einer MVTO-Historie über $\{t_0,\ldots,t_n\}$.

MVTO1 (Eigenschaft von Zeitstempeln): $ts(t_i) = ts(t_j) \prec\succ i = j$

MVTO2 (Übersetzen von Lesen):

$\forall r_k(x_j) \in CP(m)\ \ w_j(x_j) <_{CP(m)} r_k(x_j) \wedge ts(t_j) < ts(t_k)$

MVTO3 (Übersetzen von Schreiben):

$\forall r_k(x_j), w_i(x_i) \in CP(m),\ i \neq j$

(a) $ts(t_i) < ts(t_j)$ XOR

(b) $ts(t_k) < ts(t_i)$ XOR

(c) $i = k \wedge r_k(x_j) <_{CP(m)} w_i(x_i)$

MVTO4 (Vollständigkeit):

$r_j(x_i) \in CP(m),\ i \neq j \quad c_j \in CP(m) \succ c_i <_m c_j.$

MVTO-Korrektheit (2)

Satz 7.27

$Gen(\text{MVTO}) \subset MCSR$

Beweisskizze:

Wir definieren Versionsordnung << wie folgt: $x_i << x_j \prec\!\succ ts(t_i) < ts(t_j)$.

Zeige: $MVSG(CP(m),<<)$ ist azyklisch. Zeige dazu, dass

$$\forall t_i \rightarrow t_j \in MVSG(CP(m),<<): ts(t_i) < ts(t_j).$$

$G(m)$:

$t_i \rightarrow t_j \in G(CP(m) \succ t_j \triangleright_{CP(m)}(x)\ t_i$ für ein x.

MVTO2 $\succ ts(t_i) < ts(t_j)$.

MVTO-Korrektheit (3)

MV:

Seien $r_k(x_j)$, $w_i(x_i) \in CP(m)$, $i \neq j \neq k$. Führt zu einer der folgenden Versionsordnungskanten

1. $x_i << x_j \succ t_i \rightarrow t_j \in MVSG(CP(m),<<)$:
   Nach Def. von <<: $ts(t_i) < ts(t_j)$.
2. $x_j << x_i \succ t_k \rightarrow t_i \in MVSG(CP(m),<<)$:
   Nach Def. von <<: $ts(t_j) < ts(t_i)$.
   MVTO3 $\succ$
   (a) $ts(t_i) < ts(t_j)$ XOR
   (b) $ts(t_k) < ts(t_i)$
   (a) Widerspruch zu Vorauss.: $\succ ts(t_k) < ts(t_j)$.

Da alle Kanten in $MVSG(CP(m),<<)$ in Zeitstempelordnung sind, ist $MVSG(CP(m),<<)$ azyklisch.

MVTO-Speicherbereinigung

Versionen und deren Intervalle können nur in Zeitstempelordnung gelöscht werden. Betrachte:

$$m_{11} = w_0(x_0)\ c_0\ r_2(x_0)\ w_2(x_2)\ c_2\ r_4(x_2)\ w_4(x_4)$$

mit $ts(t_i)=i$. Annahme: x_2 wird gelöscht, aber nicht x_0. $r_3(x)$ erreiche den Scheduler. Der übersetzt es fälschlicherweise in $r_3(x_0)$ statt $r_3(x_2)$.

Korrekte Reihenfolge vermeidet falsches Übersetzen, nicht aber unnötiges Invalidieren.

Annahme: x_0 wird gelöscht. $r_1(x)$ erreiche den Scheduler. Dieser findet keine Version mit $wts < ts(t_1)$. Diese Bedingung zeigt an, dass die zu lesende Version bereits gelöscht wurde. $r_1(x)$ wird zurückgewiesen.

Read-only Multiversion Protocol

Read-only Multiversion Protocol (ROMV):

- For long read transactions that execute concurrently with writers.

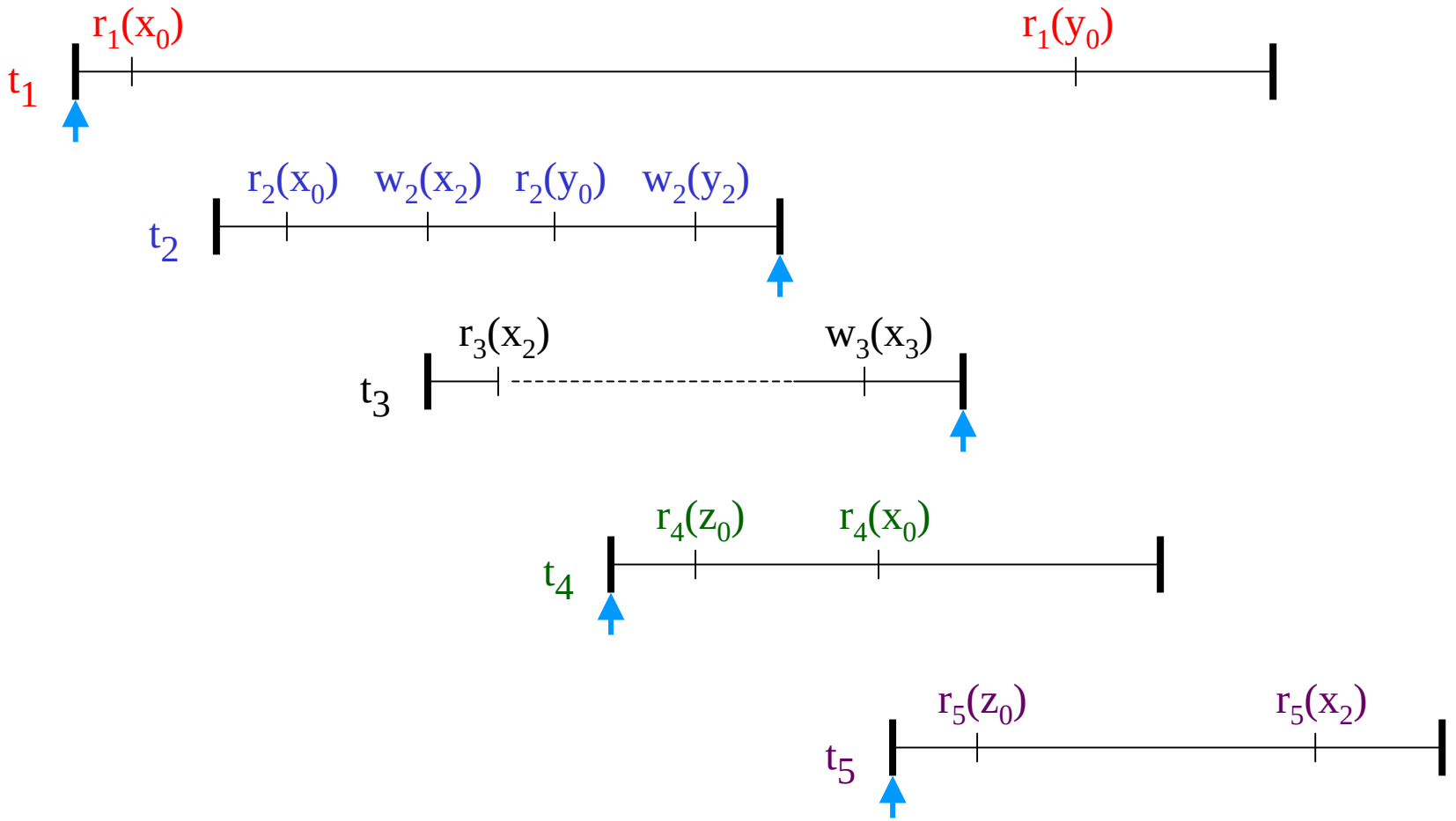
- Each update transaction uses 2PL on both its read and write set but each write creates a new version and timestamps it with the transaction‘s commit time
- Each read-only transaction t_i is timestamped with its begin time
- $r_i(x)$ is mapped to $r_i(x_k)$ where x_k is the version that carries the largest timestamp $\leq ts(t_i)$ (i.e., the most recent committed version as of the begin of t_i)

Gen(ROMV) $\subseteq$ *MVSR*:

Sketch of proof: all edges $t_i \rightarrow t_j \prec\succ c_i < c_j$

Indeed: *Gen*(ROMV) $\subseteq$ *MCSR*

Read-only Multiversion Protocol



DBS

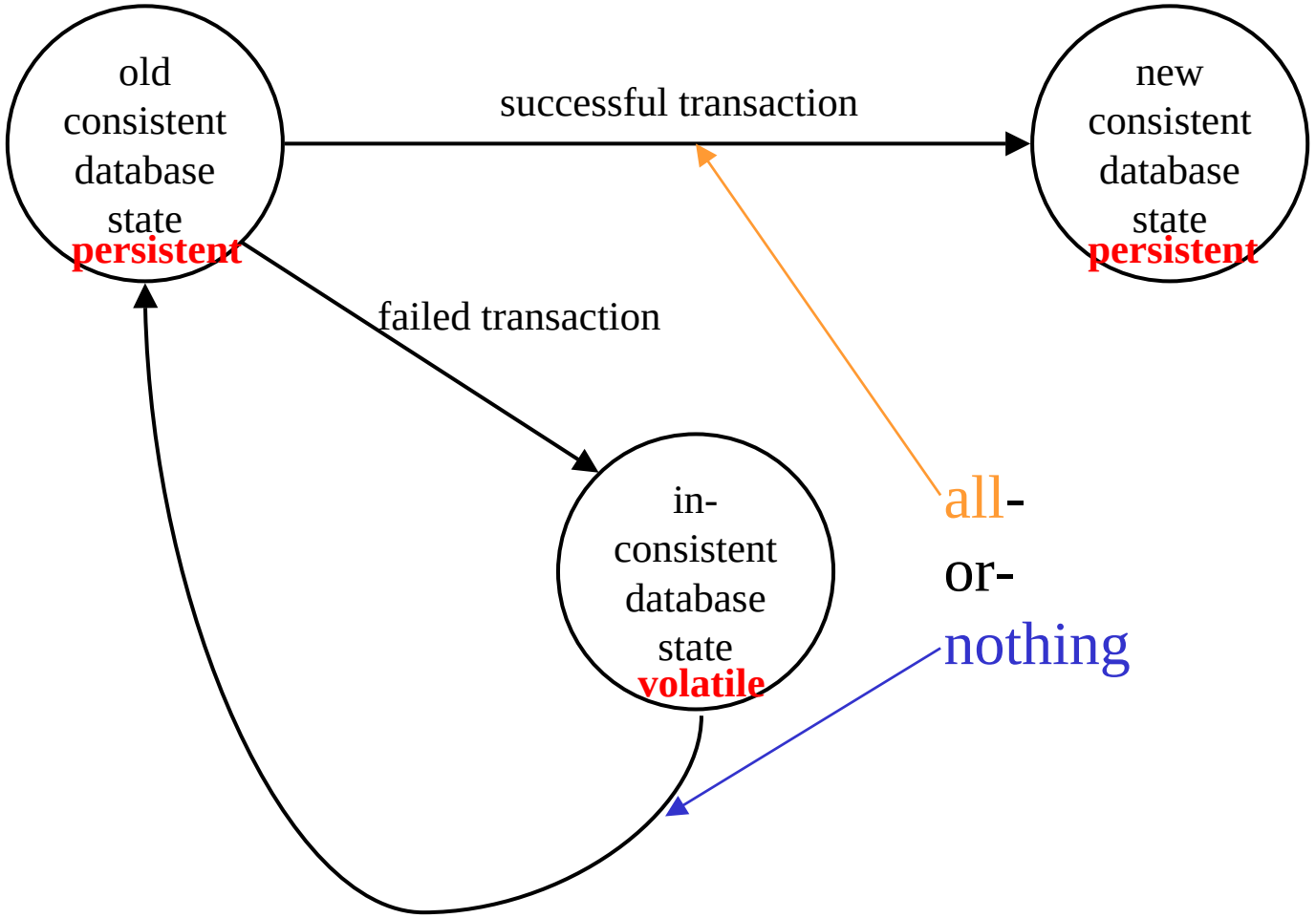
Universität Karlsruhe (TH)

Chapter 8

Transaction Recovery

Atomicity

Remember atomicity



A **transaction** is **atomic** if it has the all-or-nothing property.

Standard solution in case of failure.

All ...

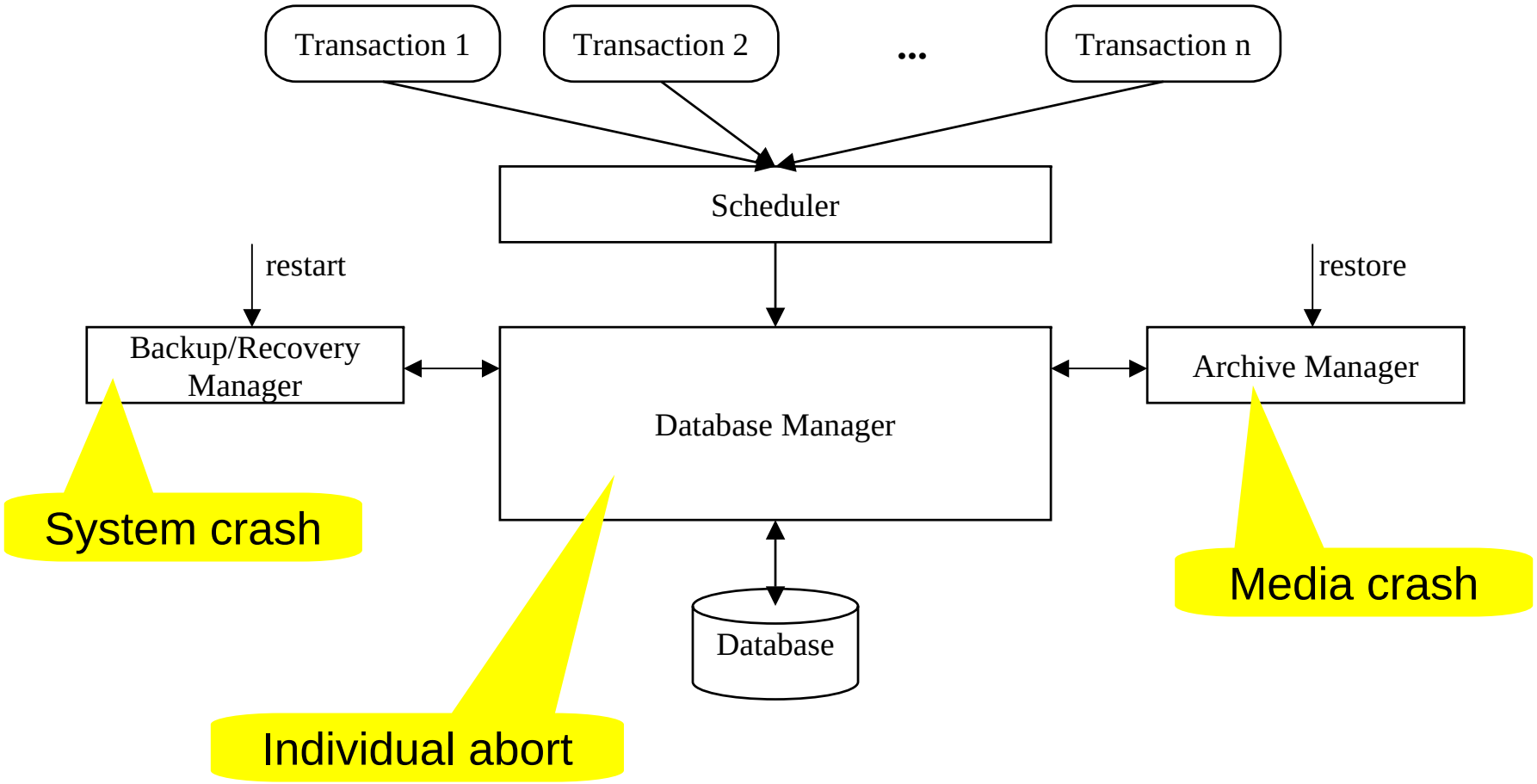
On commit of a transaction:

Make sure that the results of all write operations have safely been stored in non-volatile (stable) store.

… nothing

target	threat	context
Individual executing transaction	Abort by transaction Abort by scheduler Abort by system, e.g., due to input error, programming error, erroneous deletion	System in control
All transactions in the system that have not completed	System crash	System lost control and must regain it, all data in transient store are lost
All transactions from a given time on	Media crash	System in control, all data in stable store are lost

Responsibilities



Correctness

Approach

- **General problem**: Whenever a transaction must be aborted, how to undo its effects in the presence of concurrent transactions s.t.
  - atomicity is enforced, **correctness!**

    must be defined on histories!
  - concurrent transactions remain undisturbed. **convenience!**

    can only be defined on schedules!
- **Requirement**: As with isolation, we should be able to formulate suitable guarantees
  - solely on the basis of the observable effects of transactions,
  - i.e., in terms of permissible schedules of commit, abort, read and write operations.

Correctness

Definition 8.1 (Recoverability):
A history h is **recoverable** if the following holds for all t_i, $t_j \in$ trans(h):
if t_i reads from t_j in h and $c_i \in$ op(h), then $c_j < c_i$.

A schedule is recoverable if its projection on completed transactions is recoverable.

RC denotes the class of all recoverable schedules.

- "history": We ignore transactions that did not yet complete (commit or abort).
  - We take a decision only after they completed.
- "RC": A successful transaction can only have read valid states, i.e. those originating with successful transactions.
  - The same is not required for aborted transactions.

No dirty reads!

Convenience

Definition 8.2 (Avoiding Cascading Aborts):
A schedule s **avoids cascading aborts** if the following holds for all t_i, $t_j \in$ trans(s):
if t_i reads x from t_j in s, then $c_j < r_i(x)$.

ACA denotes the class of all schedules that avoid cascading aborts.

Definition 8.3 (Strictness):
A schedule s is **strict** if the following holds for all t_i, $t_j \in$ trans(s):
for all $p_i(x) \in op(t_i)$, $p=r$ or $p=w$, if $w_j(x) < p_i(x)$ then $a_j < p_i(x)$ or $c_j < p_i(x)$.

ST denotes the class of all strict schedules.

Example

Example 8.4

$t_1 = w_1(x)\ w_1(y)\ w_1(z)\ c_1$

$t_2 = r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$

	CSR	RC	ACA	ST
h_7	+	-		
h_8	+	+	-	
h_9	+	+	+	-
h_{10}	+	+	+	+

Consider schedules (histories):

$h_7 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$

$h_8 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$

$h_9 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$

$h_{10} = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$

RC: $t_i \triangleright_h t_j, c_i \in s \Rightarrow c_j <_h c_i$

ACA: $t_i \triangleright_s (x)\ t_j \Rightarrow c_j <_s r_i(x)$

ST: $w_j(x) <_s p_i(x) \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$

Example

	CSR	RC	ACA	ST
h_7	+	-		
h_8	+	+	-	
h_9	+	+	+	-
h_{10}	+	+	+	+

$h_7 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$

$h_8 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$

$h_9 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$

$h_{10} = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$

Suppose abort of t_1 (a_1 in place of c_1).

h_7: $r_2(y)$ dirty read, possibly incorrect $w_2(y)$:

t_2 may reach incorrect final state, but due to c_2 t_2 cannot be undone.

h_8: $r_2(y)$ dirty read, possibly incorrect $w_2(y)$:

Since no c_2, t_2 can and must be aborted as well.

h_9: $r_2(y)$ clean read, hence $w_2(y)$ correct:

t_2 can continue. Some technical effort to make sure that undo of $w_1(x), w_1(y), w_1(z)$ does not undo effect of $w_2(x)$.

h_{10}:$r_2(y)$ clean read, effect of $w_2(x)$ not endangered.

Strictness and Rigorousness

Definition 8.3 (Strictness):
A schedule s is **strict** if the following holds for all t_i, $t_j \in$ trans(s):
for all $p_i(x) \in$ op(t_i), p=r or p=w,
if $w_j(x) < p_i(x)$ then $a_j < p_i(x)$ or $c_j < p_i(x)$.

ST denotes the class of all strict schedules.

Definition 8.5 (Rigorousness):
A schedule s is **rigorous** if the following holds for all t_i, $t_j \in$ trans(s):
for all $p_i(x) \in$ op(t_i), $p_j(x) \in$ op(t_j), p=r or p=w,
if $p_j(x) < p_i(x)$ then $a_j < p_i(x)$ or $c_j < p_i(x)$.

RG denotes the class of all rigorous schedules.

Wiederanlaufbarkeit und Serialisierbarkeit

Bemerkung 8.6 (Orthogonalität)

- Schnitt zwischen *CSR* und *X* für $X \in \{RC, ACA, ST\}$ ist nicht leer.
- *CSR* ist unvergleichbar (orthogonal) zu *X*, es gilt also keinerlei Teilmengenbeziehung.

$\Rightarrow$ Ein Scheduler muss also Serialisierbarkeit und Wiederanlaufbarkeit (ggf. Vermeidung von kaskadierendem Rücksetzen oder Striktheit) getrennt garantieren.

Relationships Among Schedule Classes

The definitions can be extended to histories. Then:

Theorem 8.6:

$RG \subset ST \subset ACA \subset RC$

Theorem 8.7:

$RG \subset COCSR$

Remember: SS2PL generates rigorous schedules $\Rightarrow$ SS2PL scheduler guarantees both, serializability and recoverability of schedules.

Relationships Among Schedule Classes

Beweis:

$RG \subset ST$: per Def. 8.5

$ST \subset ACA$: Sei $h \in ST$ und $t_i, t_j \in h$ $(i \neq j)$.
$\succ_{(ST)} w_j(x) <_h r_i(x) \succ a_j <_h r_i(x) \vee c_j <_h r_i(x)$
Annahme: $t_i \triangleright_h(x) t_j$ für ein x
$\succ$ nicht $a_j <_h r_i(x) \succ c_j <_h r_i(x)$
$\succ h \in ACA$
Die Echtheit der Teilmengenbeziehung folgt mit h_9.

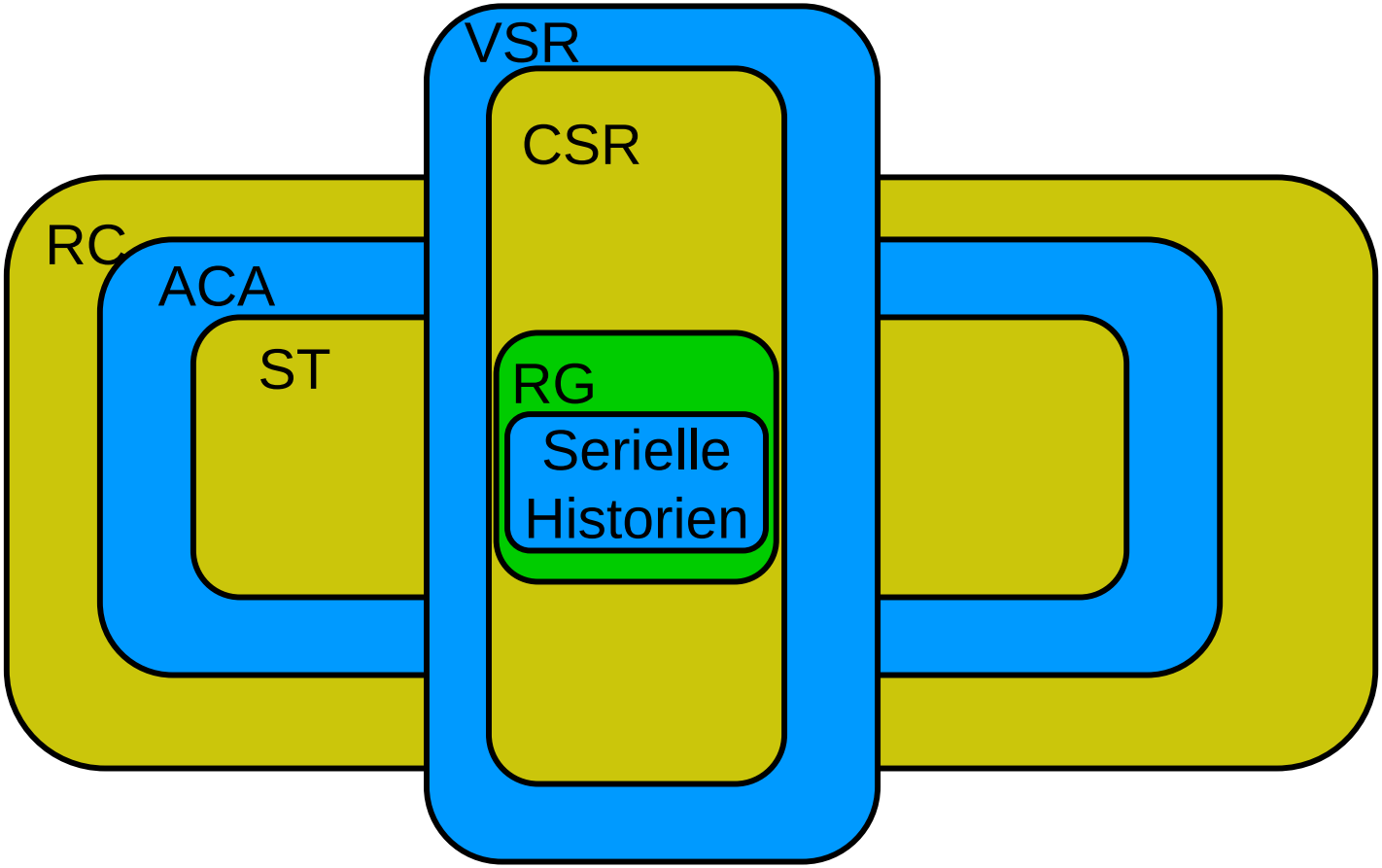
$ACA \subset RC$: Sei $h \in ACA$ und $t_i, t_j \in h$ $(i \neq j)$.
Annahme: $t_i \triangleright_h(x) t_j$ und $c_i \in h$
$\succ_{(ACA)} c_j <_h r_i(x)$
$c_i \in h \succ r_i(x) <_h c_i \succ c_j <_h c_i$
$\succ h \in RC$
Die Echtheit der Teilmengenbeziehung folgt mit h_8.

Wiederanlaufbarkeit und Serialisierbarkeit

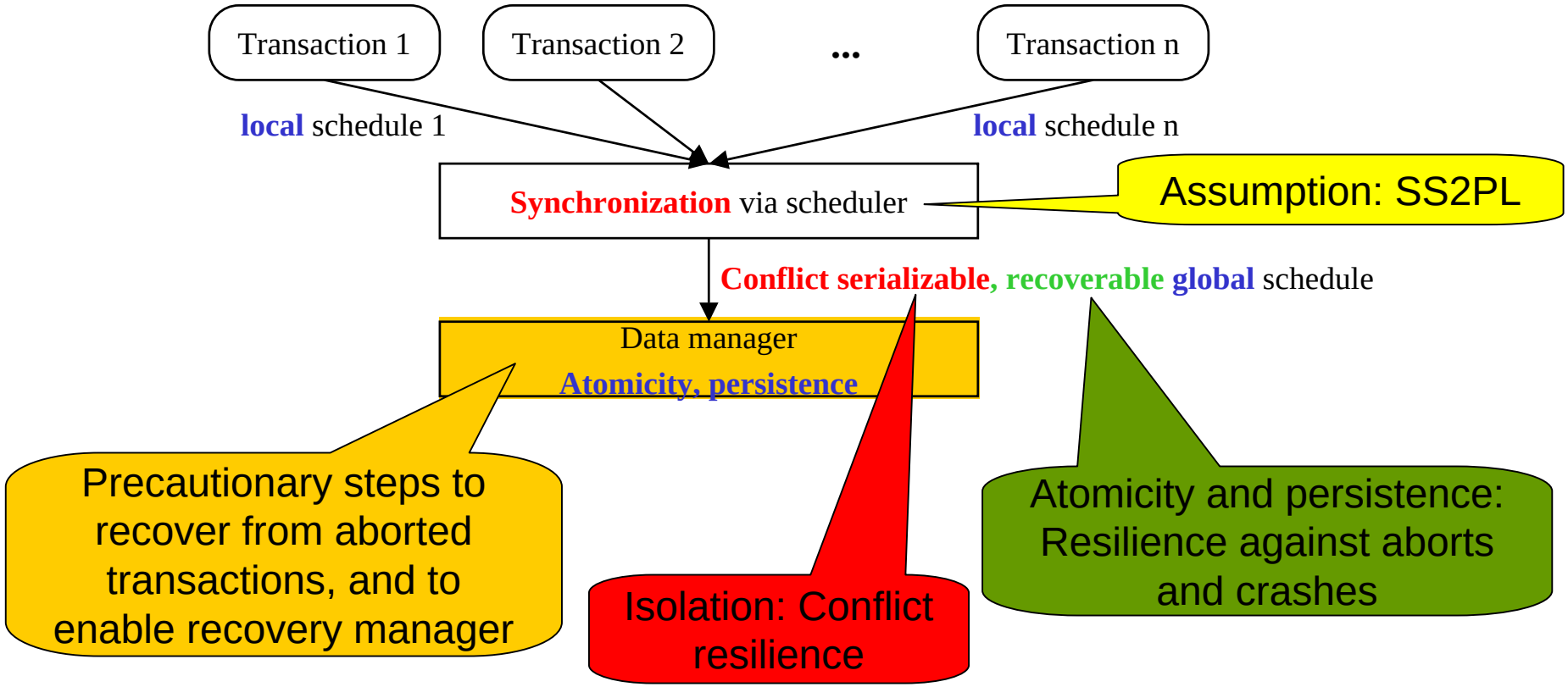
Satz 8.8

CSR, *RC*, *ACA* und *ST* sind *pcc.*

Wiederanlaufbarkeit und Serialisierbarkeit



Summary and next step



System architecture

… nothing

target	threat	context
Individual executing transaction	Abort by transaction Abort by scheduler Abort by system, e.g., due to input error, programming error, erroneous deletion	System in control
All transactions in the system that have not completed	System crash	System lost control and must regain it, all data in transient store are lost
All transactions from a given time on	Media crash	System in control, all data in stable store are lost

Assume: taken care of in the past

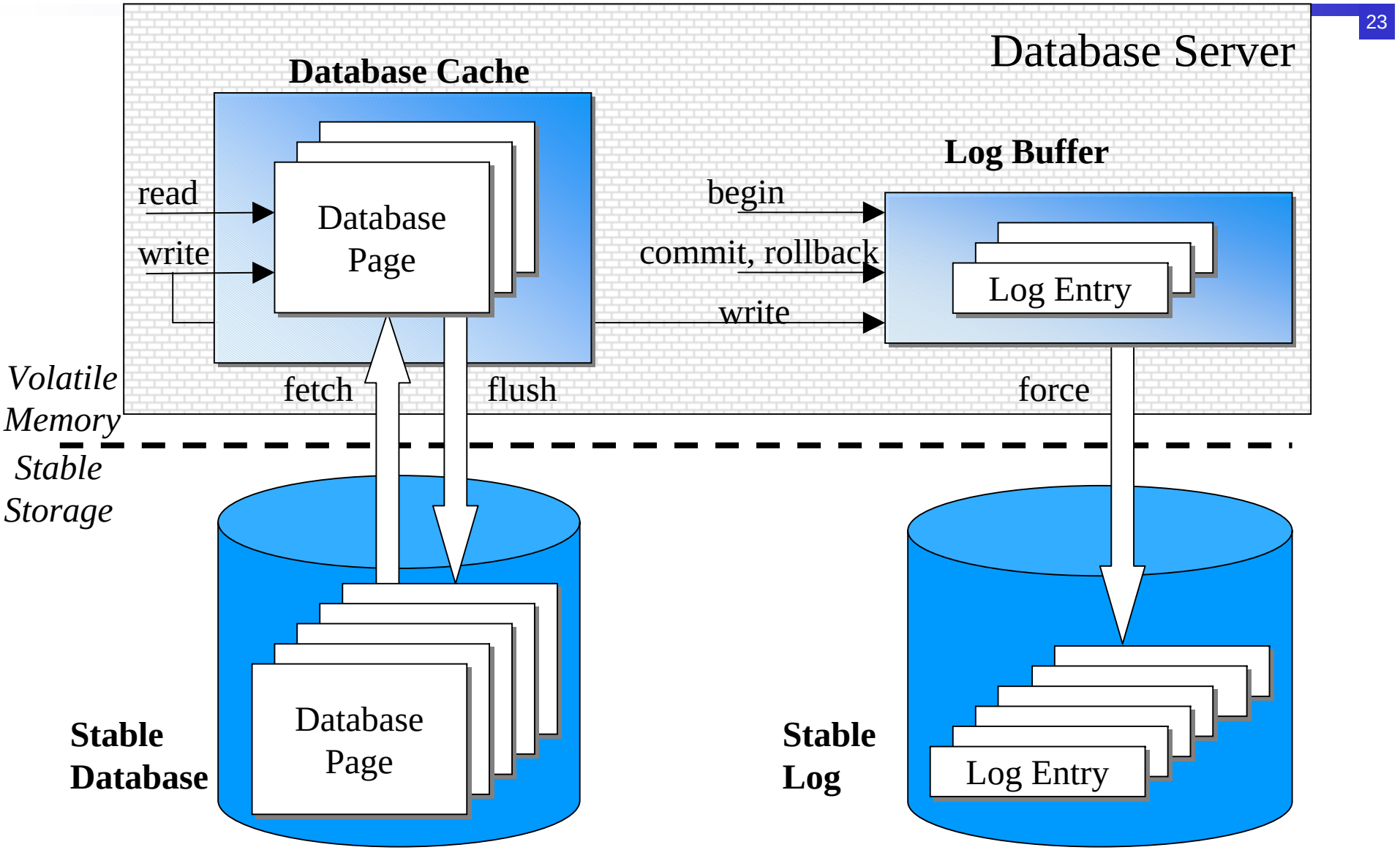
Our focus

What do we mean by “not completed”?

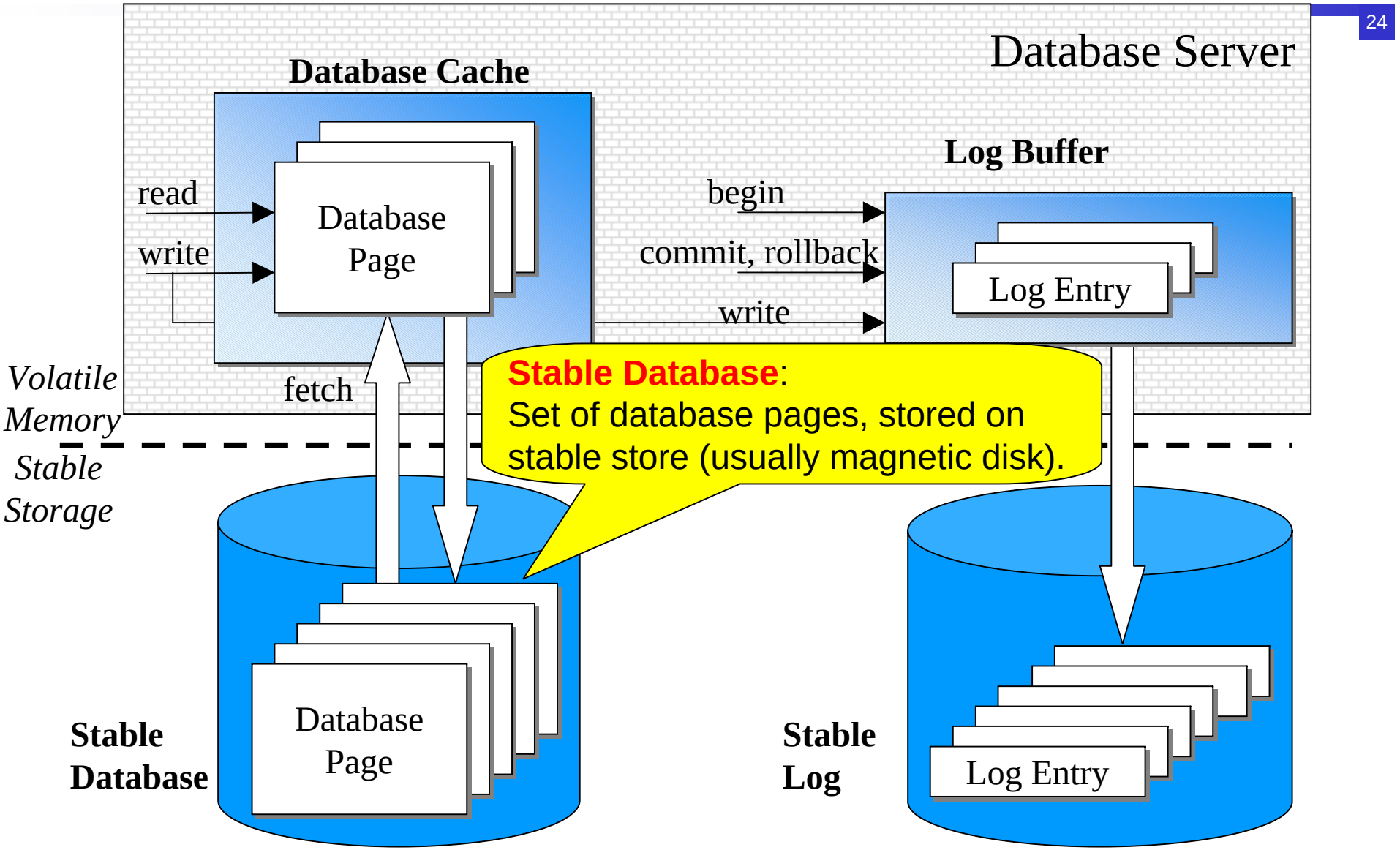
Crash recovery

- Whenever the database system crashes, we must guarantee that after restart the system can be brought into a consistent state, i.e.,
  - all changes due to committed transactions are kept intact
  - all changes due to transactions aborted earlier or active during crash must completely disappear from the database.
- Correctness of restart is critical, otherwise irreparable damage!
  - Restart must remain correct even if the system crashes during recovery.
- Performance must not suffer too much!
  - Recovery must be fast!
  - Little overhead during normal operation!

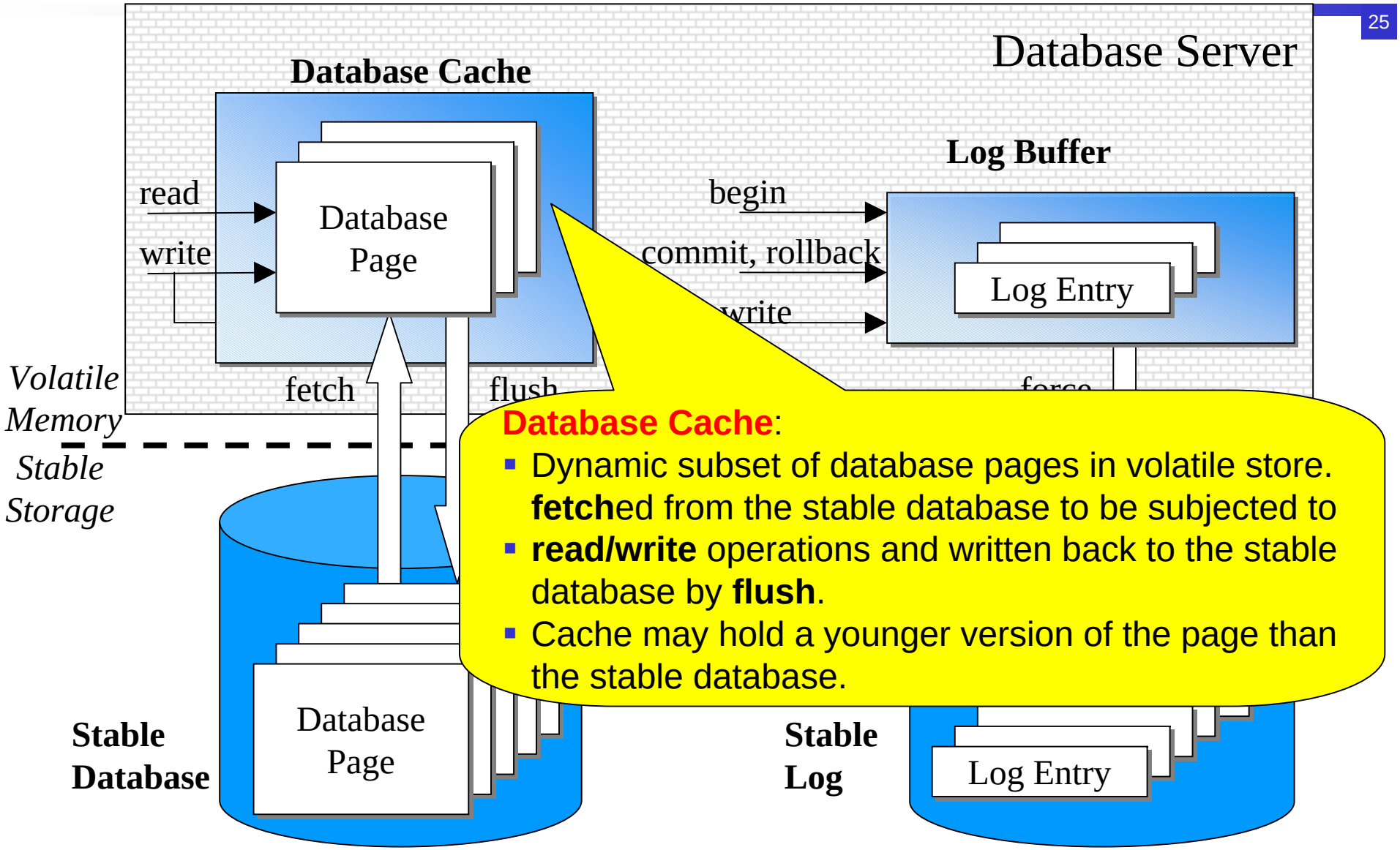
Overview of System Architecture



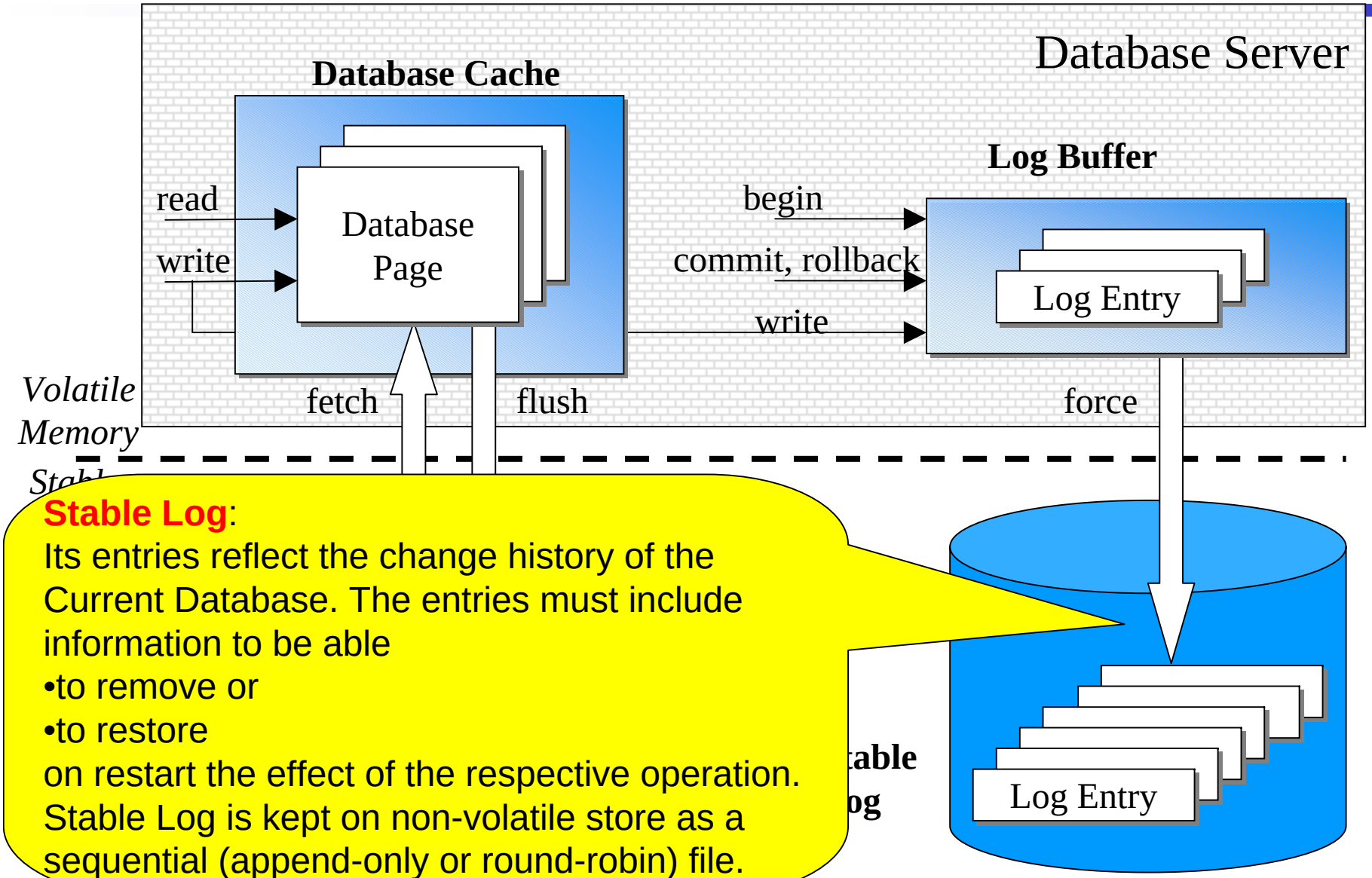
Overview of System Architecture



Overview of System Architecture



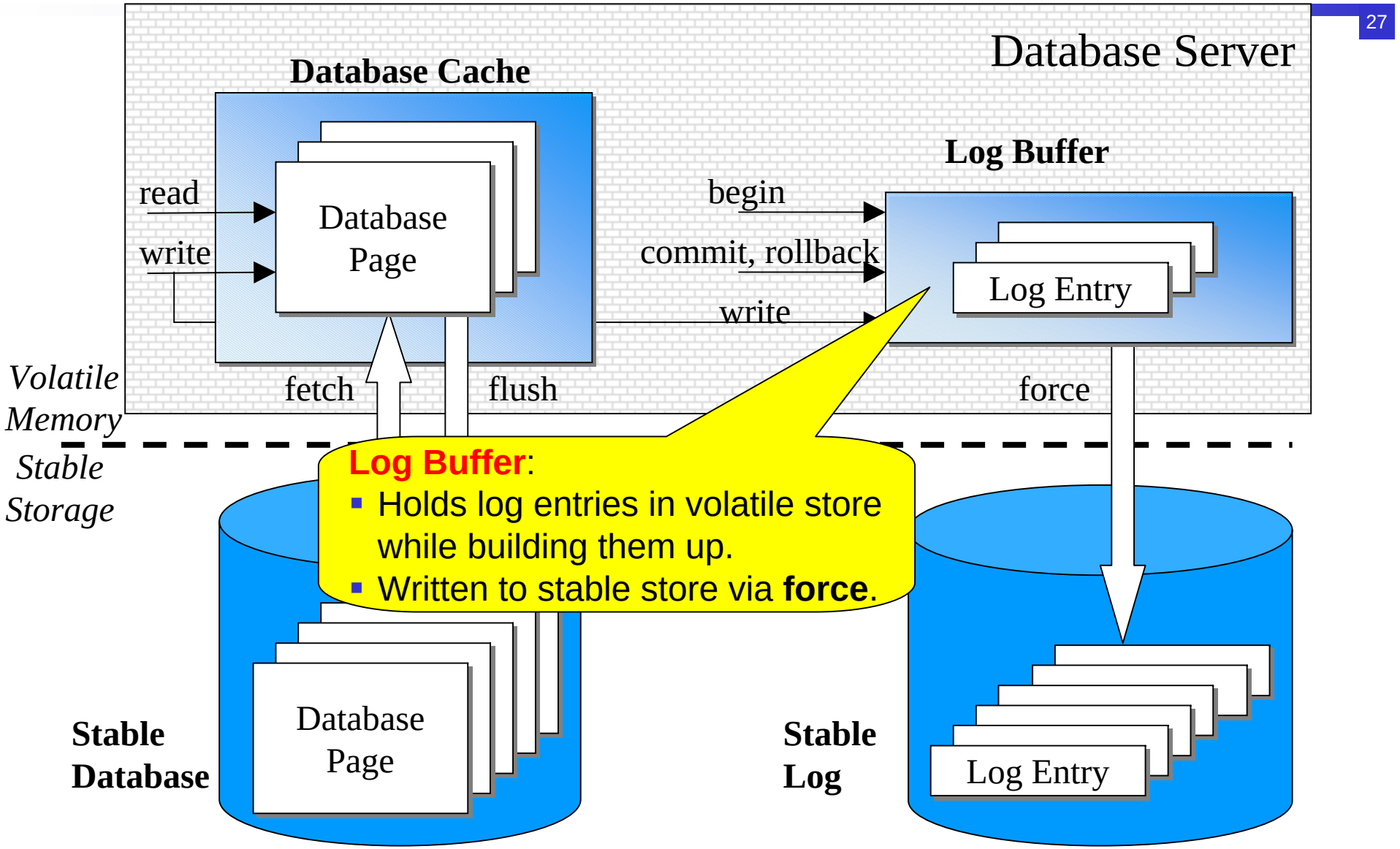
Overview of System Architecture



Stable Log:
Its entries reflect the change history of the Current Database. The entries must include information to be able
- to remove or
- to restore

on restart the effect of the respective operation.
Stable Log is kept on non-volatile store as a sequential (append-only or round-robin) file.

Overview of System Architecture



Model actions during normal operation (1)

Transaction actions:

- *begin (t)*

  Starts transaction *t*.

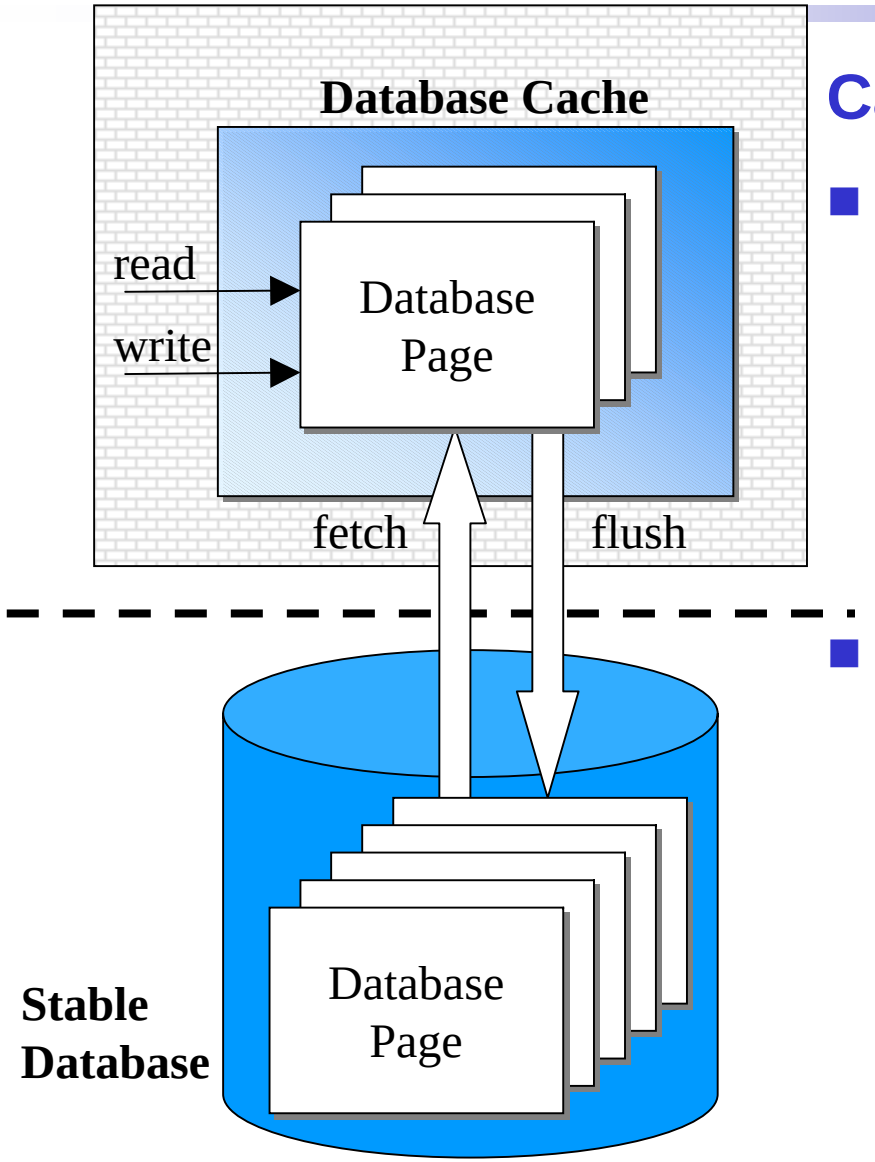
- *commit (t)*

  Successful completion of *t*. **Challenge**: Make sure that all changes by *t* survive all subsequent system crashes.

- *rollback (t)*

  Failure of *t* to come to the desired end. **Challenge**: Make sure that all changes by *t* disappear from the current database.

Model actions during normal operation (2)



Cache model:

- Usually: Transaction receives page cache address, reads and updates page items unobserved.
  - Hence: read ⇒ get me the address, write ⇒ I'm done with the updates.
- Therefore, cache manager must not relocate a page within cache or to the stable database until it is no longer used by the transaction.
  - Pin the page in cache before using it, and unpin it when it is no longer used.

Model actions during normal operation (3)

Data actions:

- *read (pageno, t)*

  Makes sure that page pageno is in the cache and **pin**s it there for transaction t.

- *write (pageno, t)*

  Issued after reading the page and updating (part of) the page. Page is marked as **dirty**.

- *full-write (pageno, t)*

  Issued when a complete page is to be changed without prior read. Page is marked as **dirty**.

For purely technical reasons:

- *unfix (pageno, t)*

  Indicate that page may become unpinned.

- *allocate (pageno, t)*

  Allocates cache space for a full-write and **pin**s the page.

Model actions during normal operation (4)

Caching actions:

- *fetch (pageno)*

  Copies a page that currently is not cached from the stable database to the database cache.

- *flush (pageno)*

  Copies an unpinned page from the database cache to the stable database provided the page is marked **dirty** and the version in cache is younger than the version in the stable database.
  Page remains in cache but is re-marked as **clean**.

For purely technical reasons:

- *pin (pageno)*

  Pins the page in cache. Automatically executed in connection with read and allocate.

- *unpin (pageno)*

  Unpins the page. Done at the appropriate time.

pin/unpin is entirely unrelated to lock/unlock.

Model actions during normal operation (5)

Log actions:

- *force ()*

  Copies *all* log entries from the log buffer to the stable log.

Unpin/flush strategies

Caching actions:

- *fetch (pageno)*

  Copies a page that currently is not cached from the stable database to the database cache.

- *flush (pageno)*

  Copies an unpinned page from the database cache to the stable database provided the page is marked **dirty** and the version in cache is younger than the version in the stable database.
  Page remains in cache but is re-marked as **clean**.

Force: flush all (remaining) pages on commit.
Noforce: flush whenever convenient.

For purely technical reasons:

- *pin (pageno)*

  Pins the page in cache. Automatically executed in connection with read and allocate.

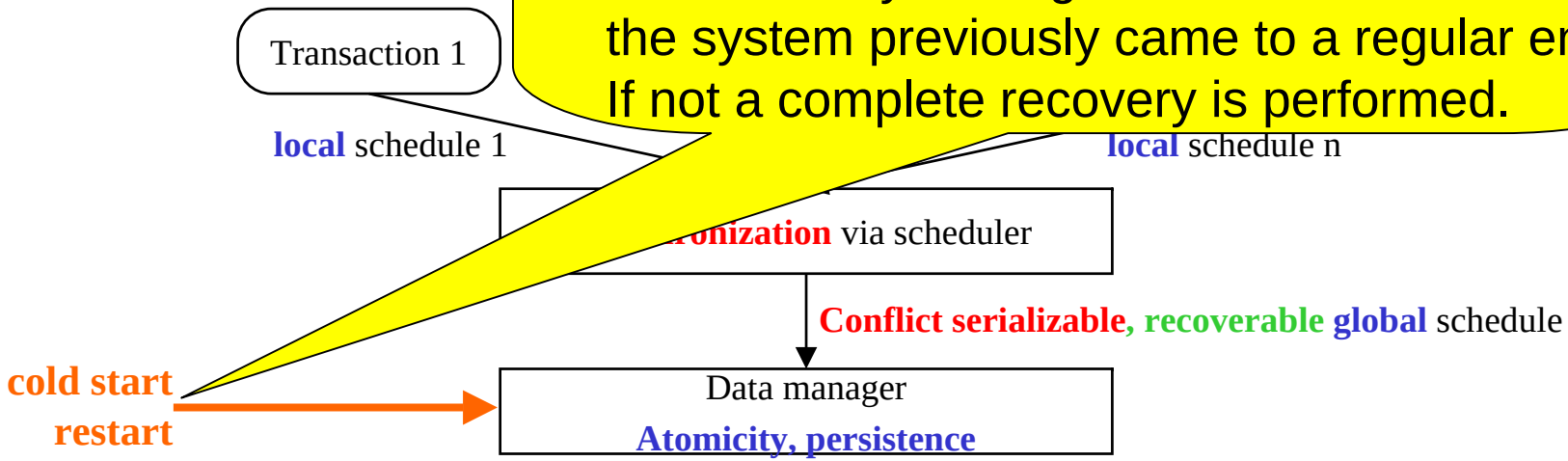
- *unpin (pageno)*

  Unpins the page. Done at the appropriate time.

Steal: Unpin on write, full-write, unfix. Flush of changed pages may take place anytime thereafter.
Nosteal: Unpin on commit.

System start

- On system start the recovery manager is automatically started, i.e., it assumes it is being restarted after a crash.
- The recovery manager determines whether the system previously came to a regular end. If not a complete recovery is performed.



Model actions during restart

Recovery actions:

- *redo ()*

  Changes of *all* committed transactions are reproduced.

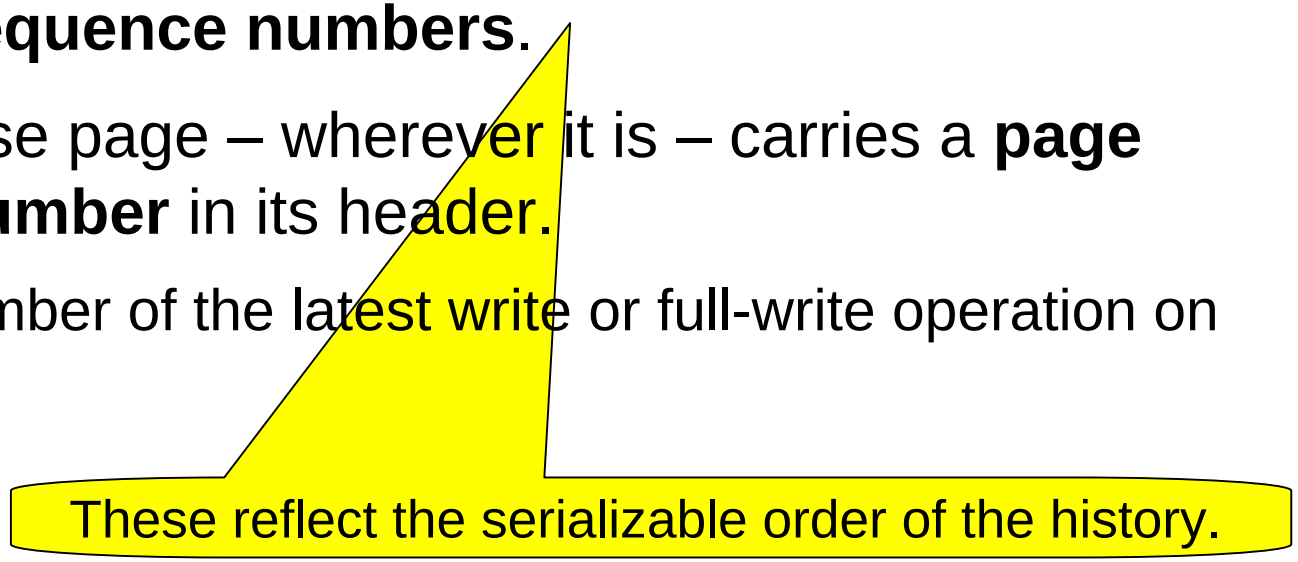
- *undo ()*

  Changes of *all* uncommitted (active or aborted) transactions are removed. Special case: undo(*t*) removes changes of a single transaction.

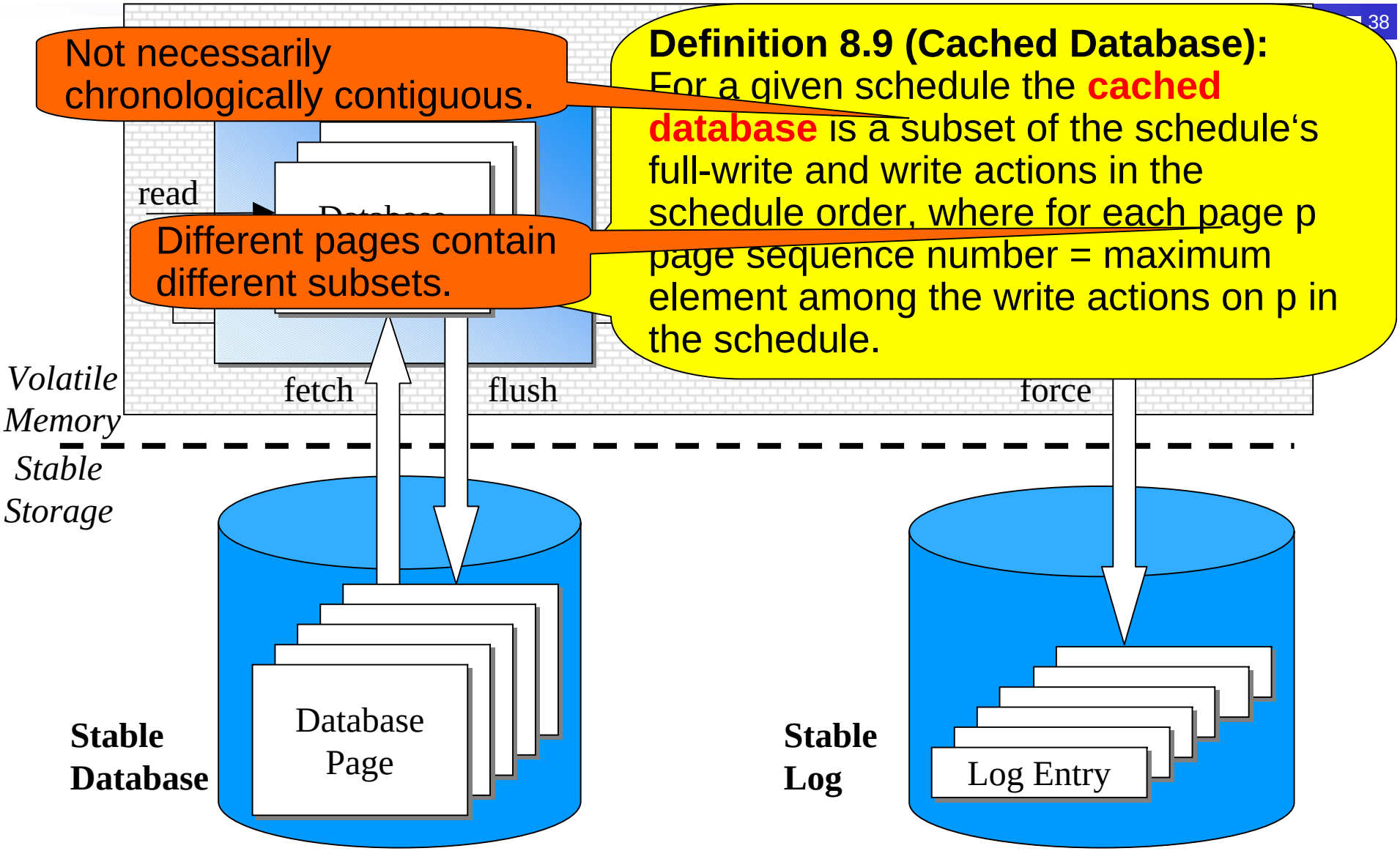
Logging and caching

Page sequence numbers

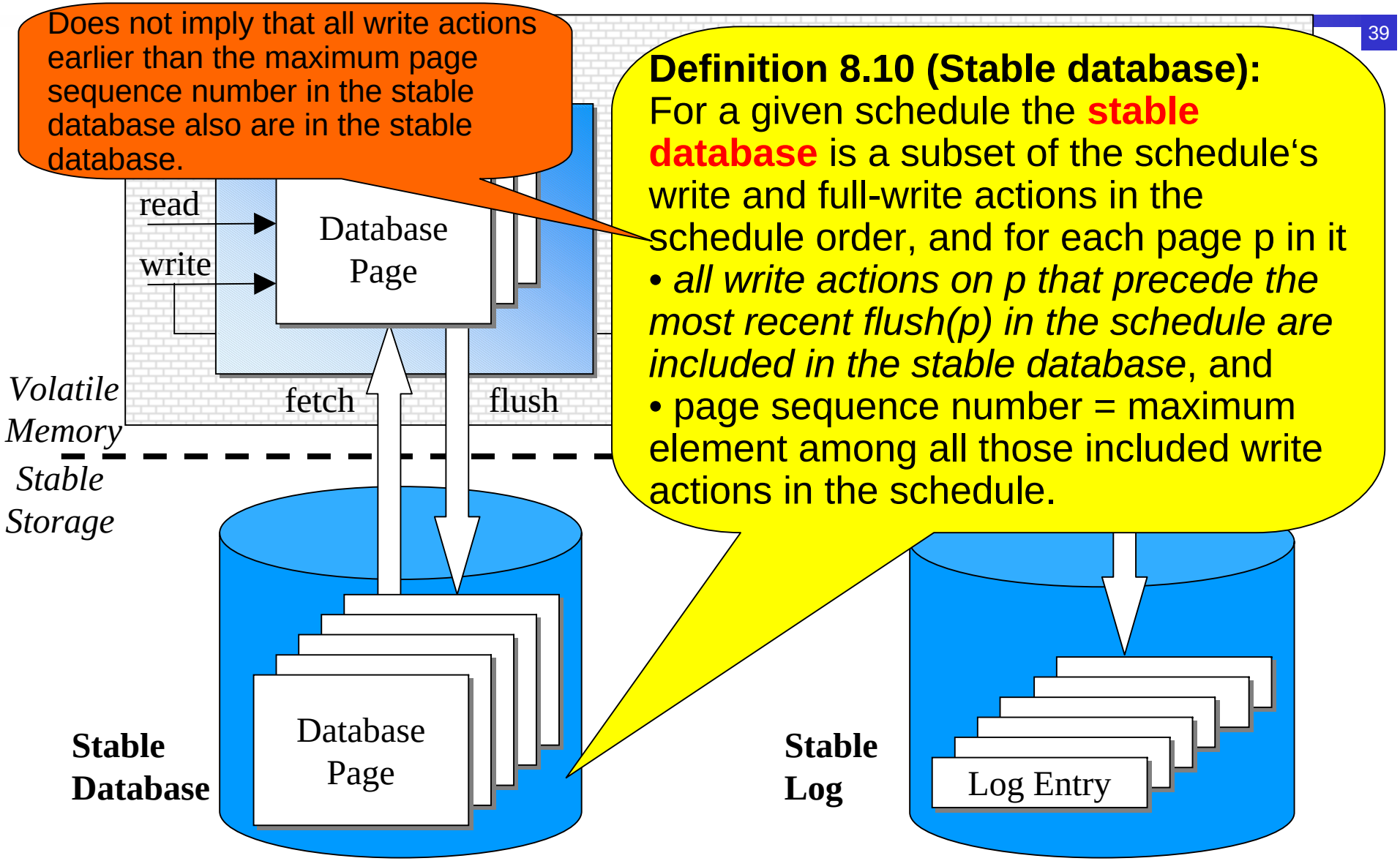
- All data actions are “tagged” with unique, monotonically increasing **sequence numbers**.
- Each database page – wherever it is – carries a **page sequence number** in its header.
  - It is the number of the latest write or full-write operation on this page.



Overview of System Architecture

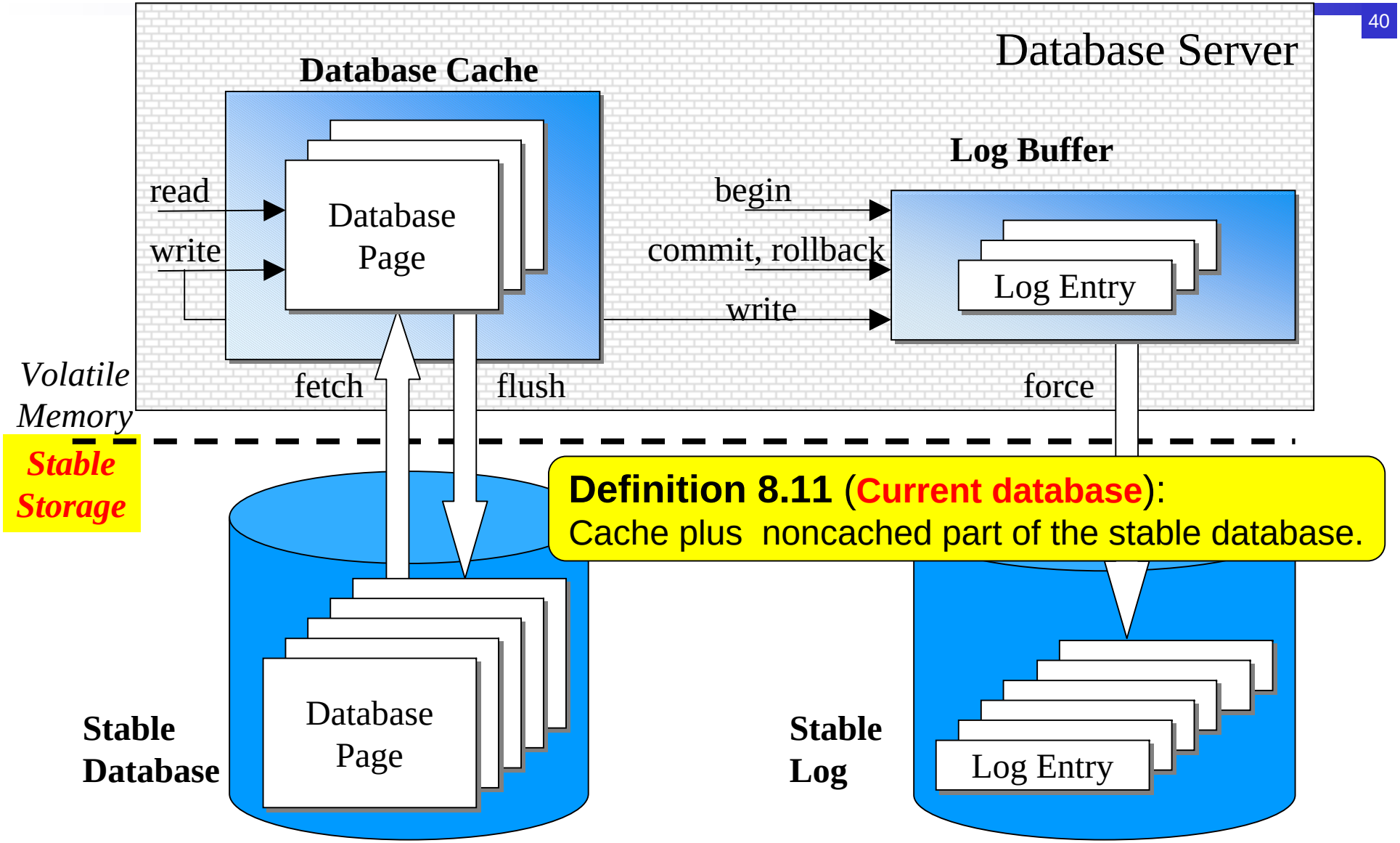


Overview of System Architecture



Definition 8.10 (Stable database):
For a given schedule the **stable database** is a subset of the schedule's write and full-write actions in the schedule order, and for each page p in it
• *all write actions on p that precede the most recent flush(p) in the schedule are included in the stable database*, and
• page sequence number = maximum element among all those included write actions in the schedule.

Overview of System Architecture



Guaranteeing the history order

On commit of a transaction

- **Phase 1**: Make sure that the results of all its write operations have safely been stored in non-volatile (*stable*) *storage*.
- **Phase 2**: Release all (remaining) locks.

RC!

Correctness Criterion

Definition 8.12 (Correct Crash Recovery):

A crash recovery algorithm is **correct** if it guarantees that, after a system failure,

the *current database* will eventually, i.e., possibly *after repeated failures and restarts*,

be equivalent to a serial order of the committed transactions

that coincides with the *serialization order of history CP*(schedule).

Logging Rules

Definition 8.13 (Logging Rules):
During normal operation, a recovery algorithm satisfies

- the **redo logging rule** if for every committed transaction *t*, all data actions of *t* are in stable storage (the stable log or the stable database),
- the **undo logging rule** if for every data action *p* of an uncommitted transaction *t* the presence of *p* in the stable database implies that *p* is in the stable log,
- the **garbage collection rule** if for every data action *p* of transaction *t* the absence of *p* from the stable log implies that *p* is in the stable database if and only if *t* is committed.

Ensures that the results of committed transactions
- are either already contained in the stable database
- or are reflected in stable log such that they cannot get lost on system crash and, hence, can be re-executed.

Definition 8.13 (Logging Rules):
During normal operation, a recovery algorithm satisfies

- the **redo logging rule** if for every committed transaction *t*, all data actions of *t* are in stable storage (the stable log or the stable database),
- the **undo logging rule** if for every data action *p* of an uncommitted transaction *t* the presence of *p* in the stable database implies that *p* is in the stable log,
- the **garbage collection rule** if for every data action *p* of transaction *t* the absence of *p* from the stable log implies that *p* is in the stable database if and only if *t* is committed.

Ensures that the results of not yet committed transactions
- are either not yet contained in the stable database
- or are reflected in stable log such that they cannot get lost on system crash and, consequently, can be removed from the stable database.

Definition 8.13 (Log

During normal ope ecovery algorithm satisfies

- the **redo l g rule** if for every committed transac t, all data actions of *t* are in stable stora e (the stable log or the stable database),
- the **undo logging rule** if for every data action *p* of an uncommitted transaction *t* the presence of *p* in the stable database implies that *p* is in the stable log,
- the **garbage collection rule** if for every data action *p* of transaction *t* the absence of *p* from the stable log implies that *p* is in the stable database if and only if *t* is committed.

Ensures
- that nothing is being removed from the log that is still needed
- implying what safely can be removed from the log to keep it short.

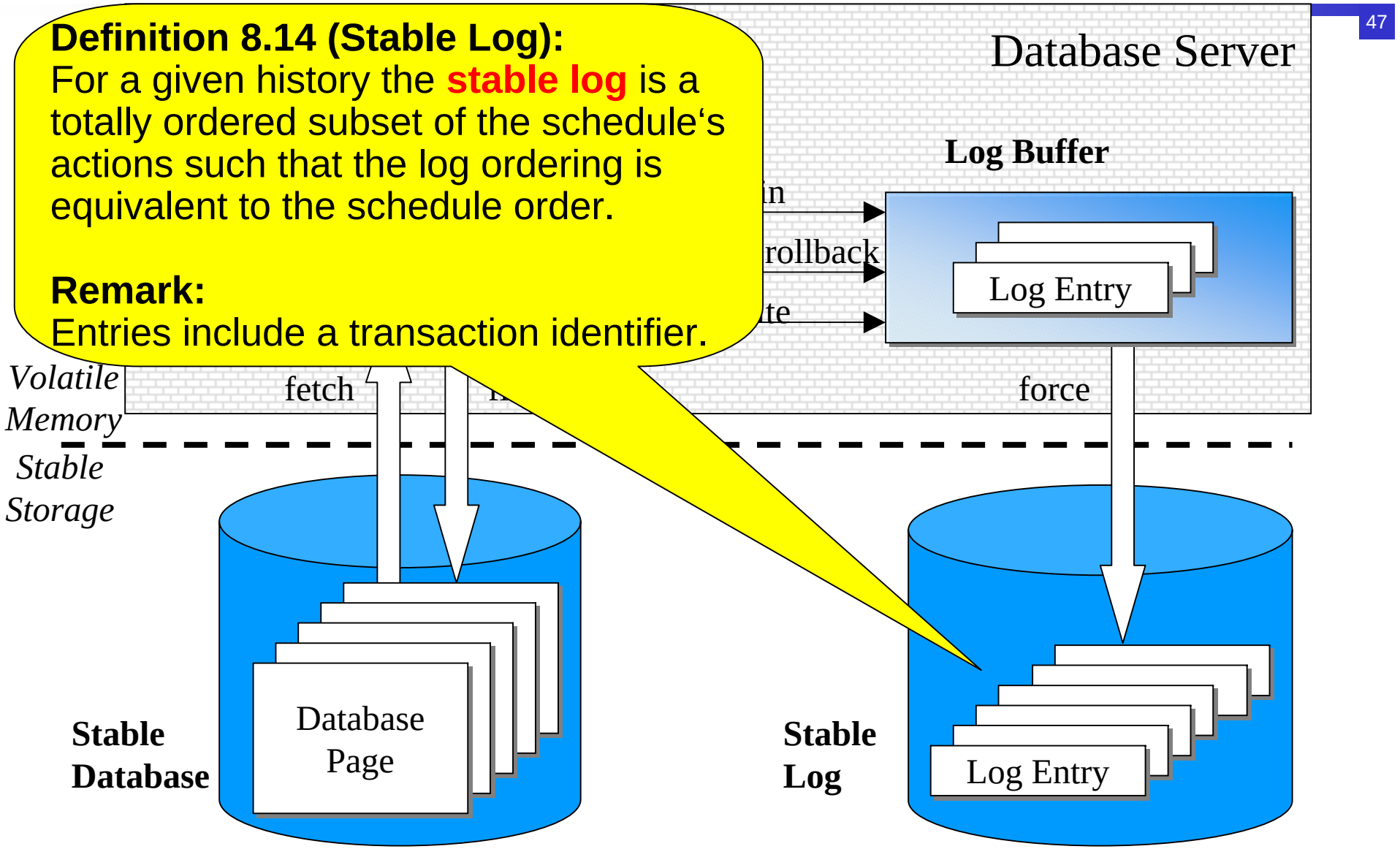
Definition 8.13 (I… …es):
During normal o… …recovery algorithm satisfies

- the **redo l… …ule** if for every committed transacti… …data actions of *t* are in stable storage… …table log or the stable database),
- the **u… …logging rule** if for every data action *p* of an unc… …mitted transaction *t* the presence of *p* in the st… …le database implies that *p* is in the stable log,
- the **garbage collection rule** if for every data action *p* of transaction *t* the absence of *p* from the stable log implies that *p* is in the stable database if and only if *t* is committed.

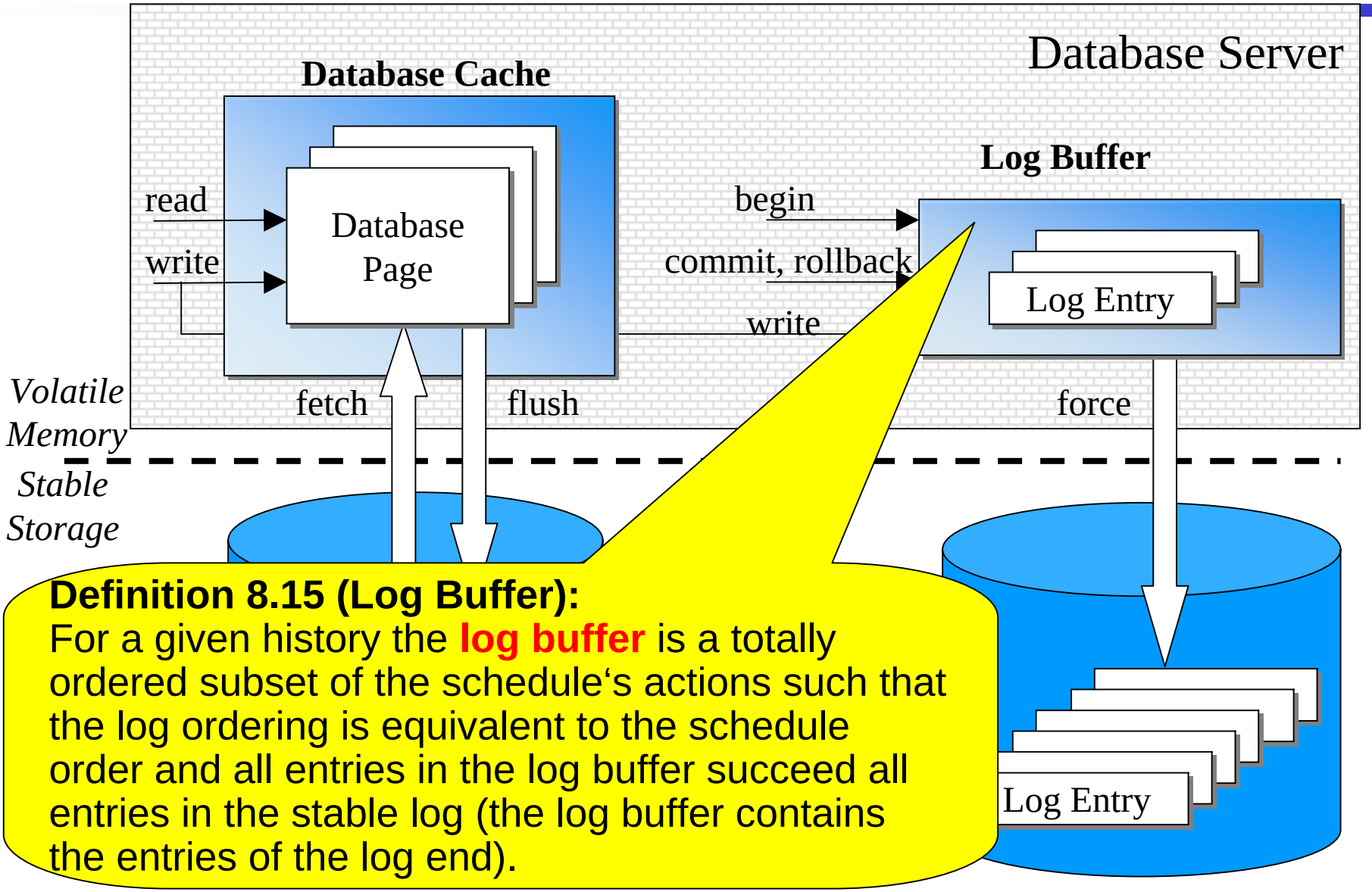
Overview of System Architecture

Definition 8.14 (Stable Log):
For a given history the **stable log** is a totally ordered subset of the schedule's actions such that the log ordering is equivalent to the schedule order.

Remark:
Entries include a transaction identifier.



Overview of System Architecture



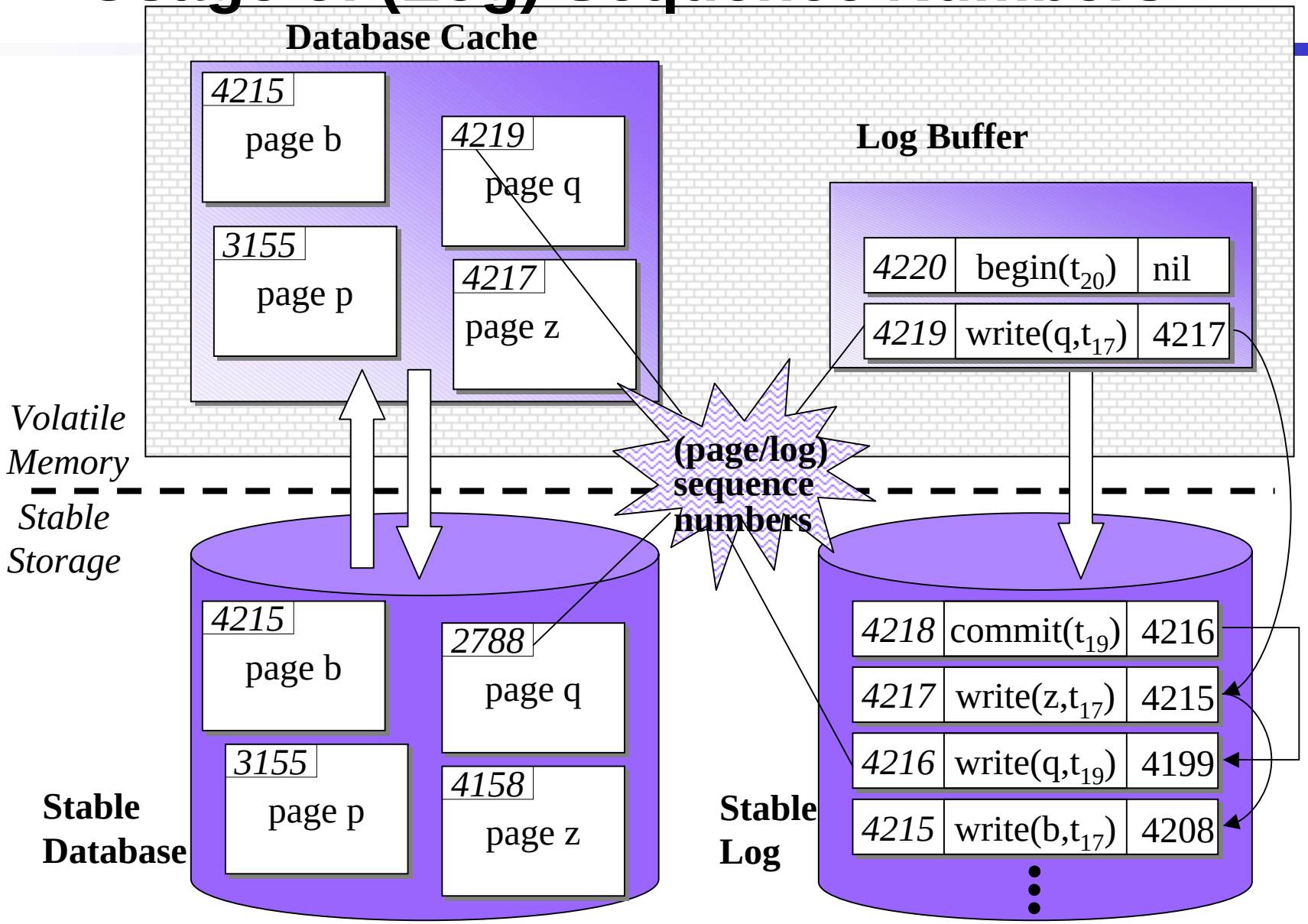
Definition 8.15 (Log Buffer):
For a given history the **log buffer** is a totally ordered subset of the schedule‘s actions such that the log ordering is equivalent to the schedule order and all entries in the log buffer succeed all entries in the stable log (the log buffer contains the entries of the log end).

Log Sequence Numbers

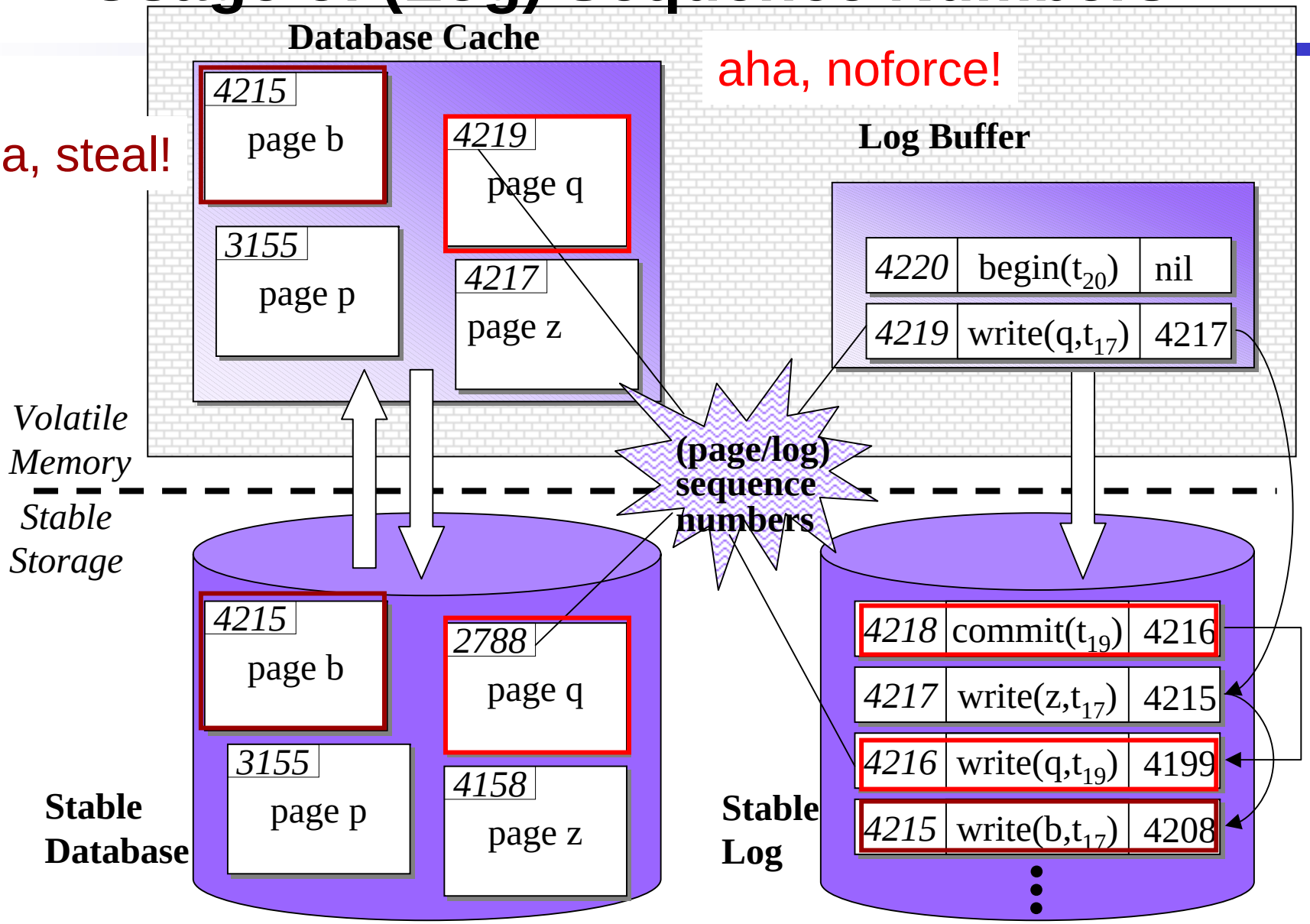
- For a given history the stable log is a totally ordered subset of the schedule‘s actions such that the log ordering is equivalent to the corresponding history order.

⇒ The (action) sequence numbers determine the order of the log entries ⇒ assign them as log sequence numbers (LSN).

Usage of (Log) Sequence Numbers

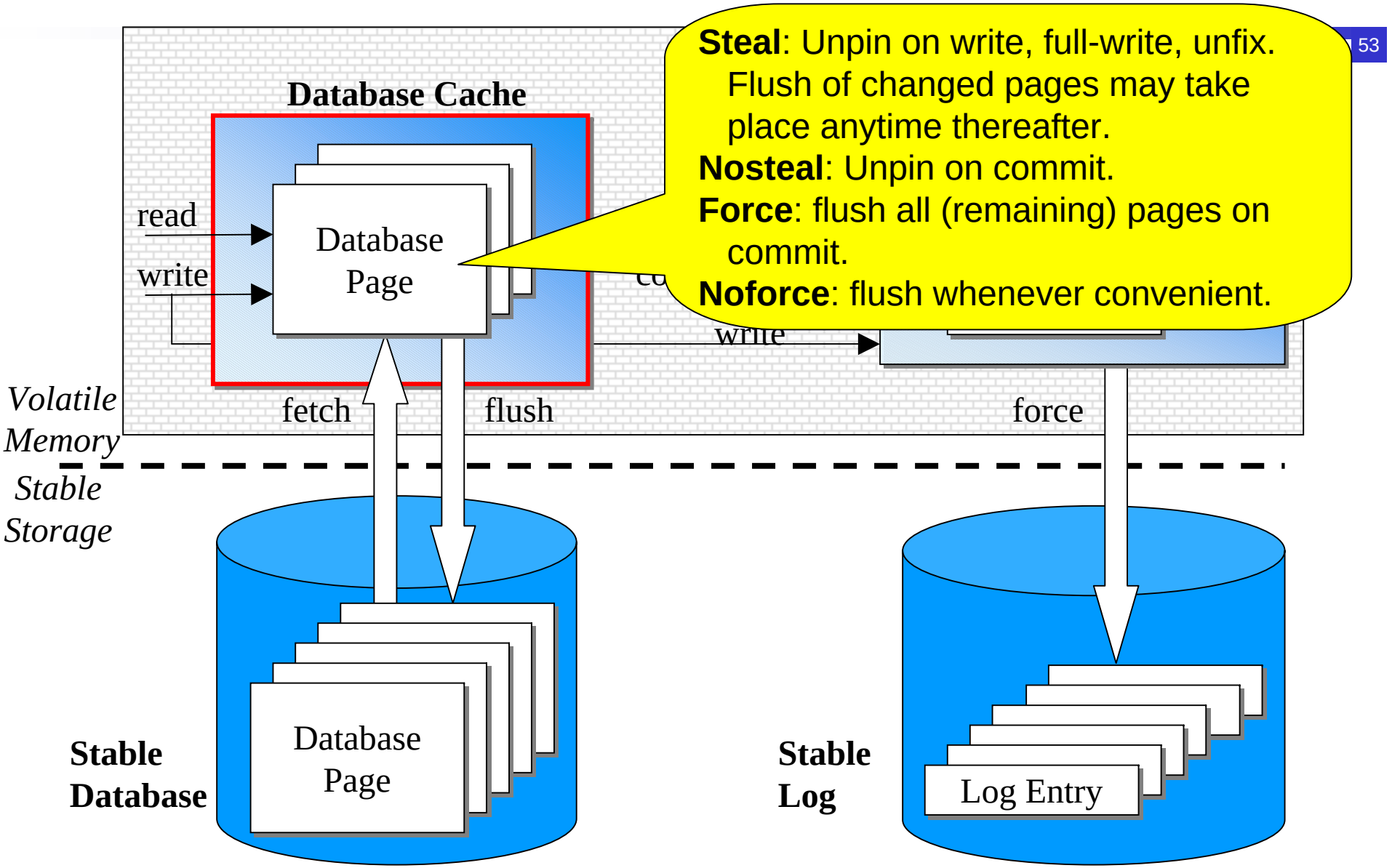


Usage of (Log) Sequence Numbers

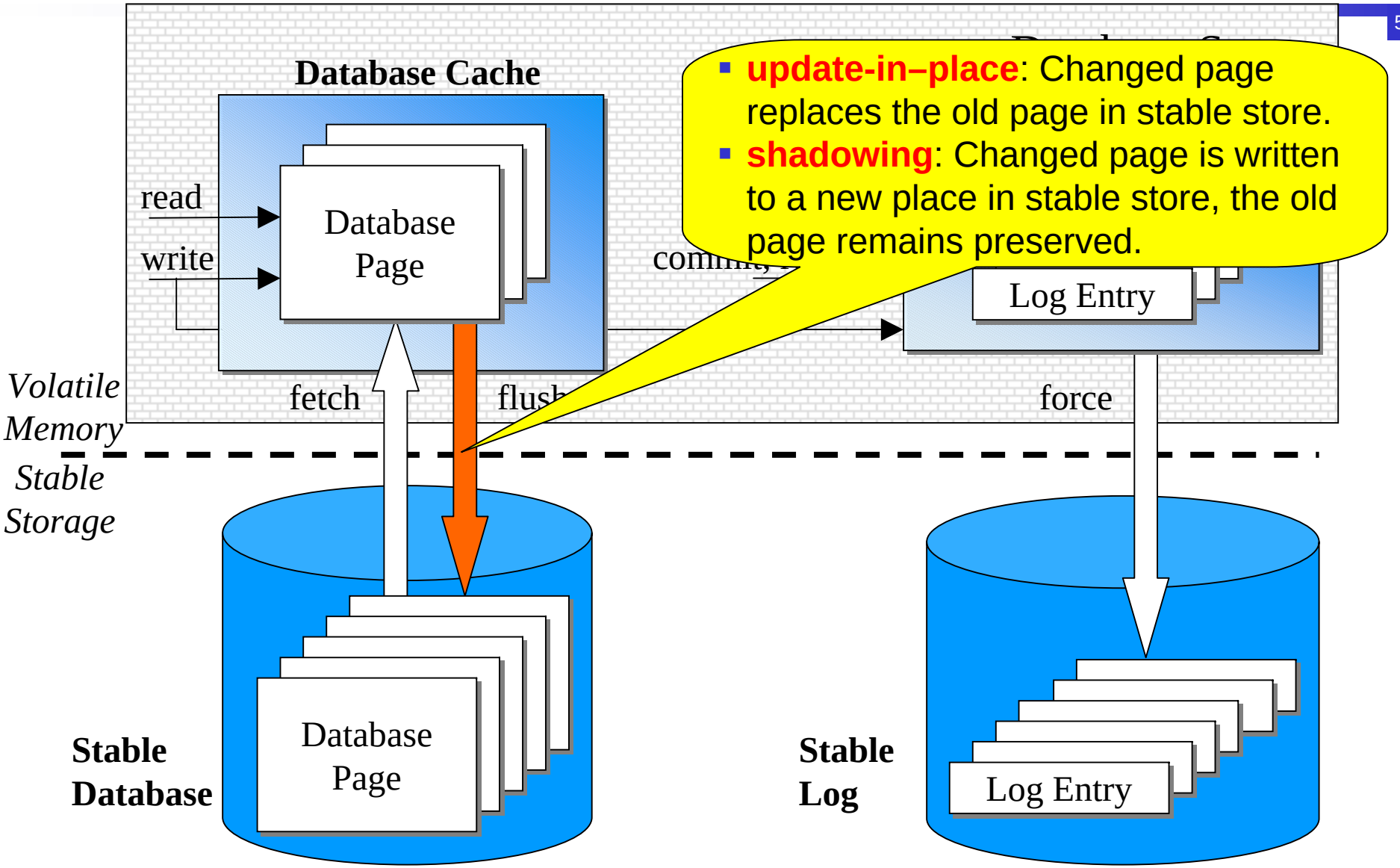


Crash recovery options

Unpin/flush strategies

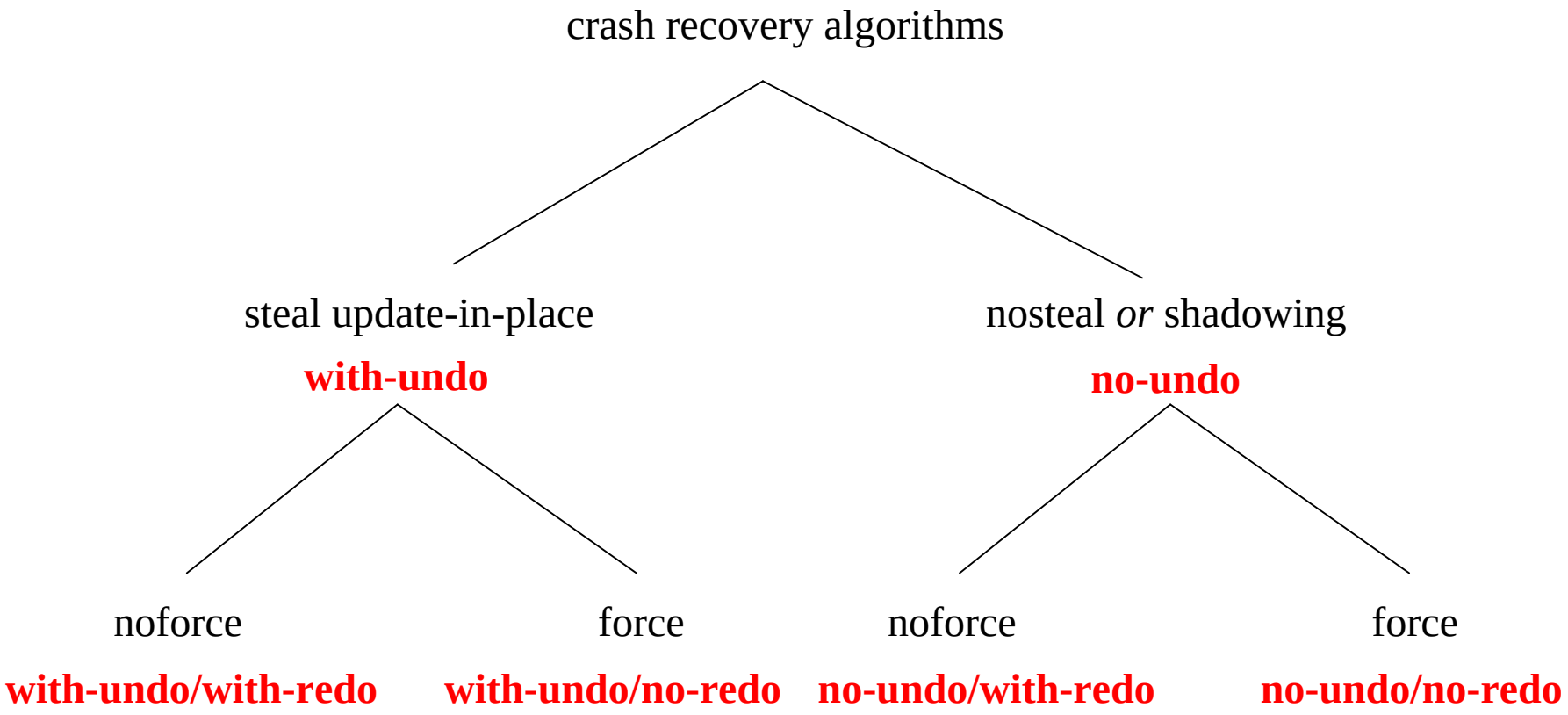


Replacement strategy



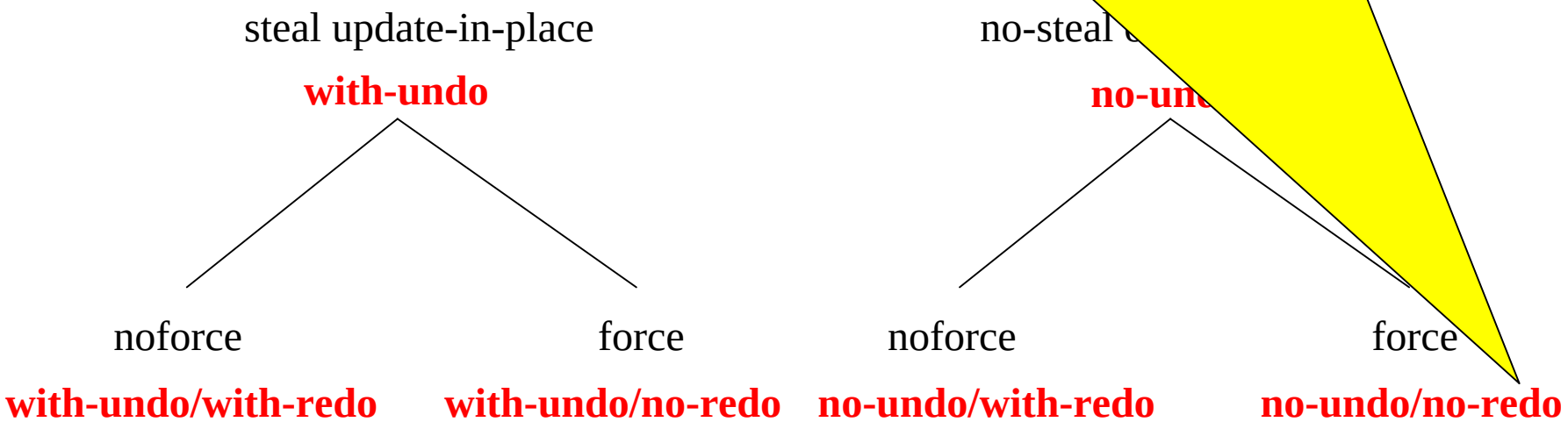
- **update-in–place**: Changed page replaces the old page in stable store.
- **shadowing**: Changed page is written to a new place in stable store, the old page remains preserved.

Taxonomy of Crash-Recovery Algorithms



Taxonomy of Crash-Recovery Algorithms

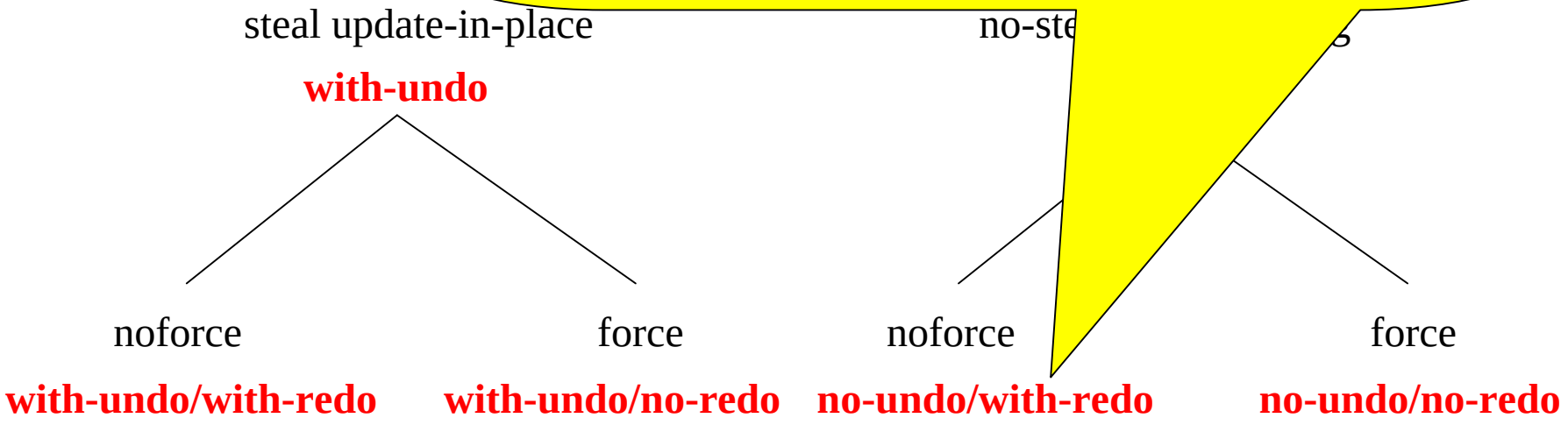
- No recovery actions needed if it can be guaranteed that
  - an action of transaction *t* is reflected in the stable database only if commit appears in stable log,
  - if stable log includes a commit of *t*, all actions of *t* are reflected in the stable database.



Taxonomy of Crash-Recovery Algorithms

- If algorithm can guarantee that
  - an action of transaction *t* is reflected in the stable database only if commit appears in stable log,

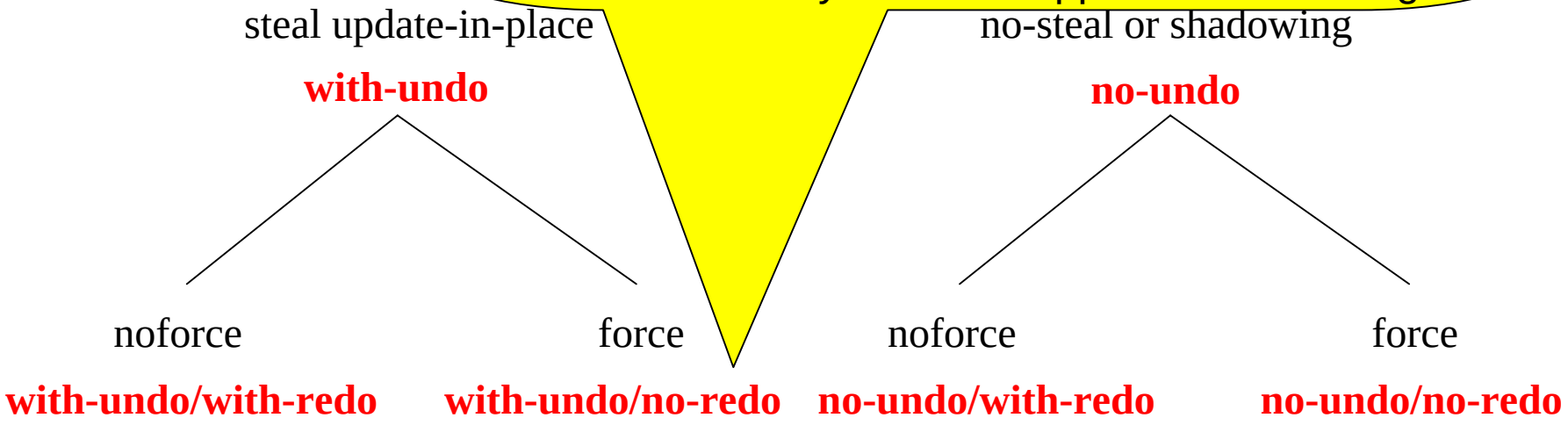
  a redo may be required to guarantee that
  - if stable log includes a commit of *t*, all actions of *t* are reflected in the stable database.



Taxonomy of Crash-Recovery Algorithms

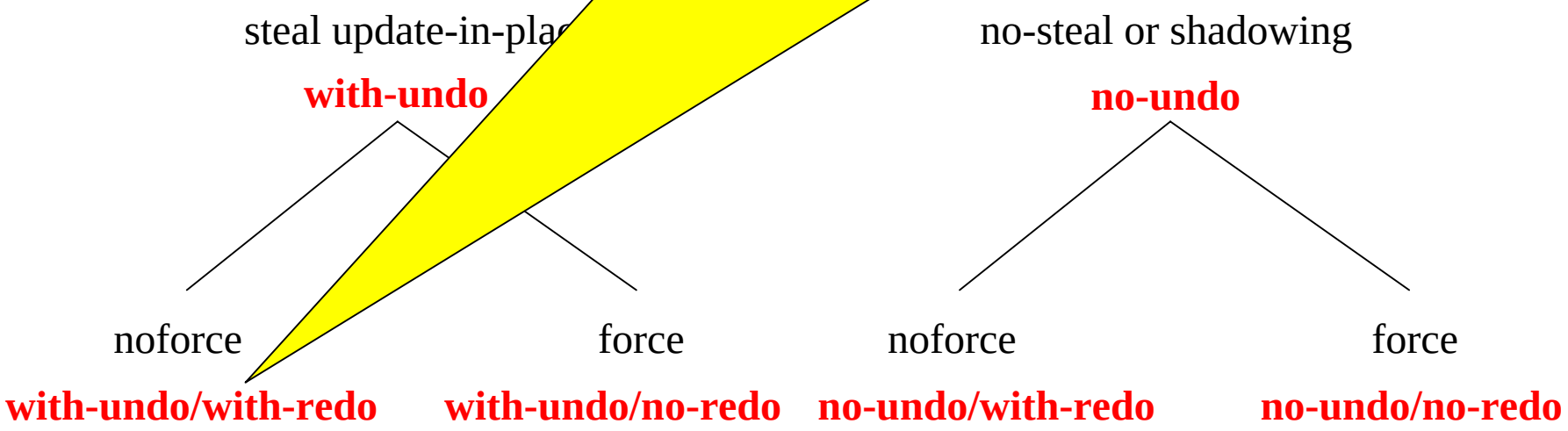
- If algorithm can guarantee that
  - if stable log includes a commit of *t*, all actions of *t* are reflected in the stable database,

  an undo may be required to guarantee that
  - an action of transaction *t* is reflected in the stable database only if commit appears in stable log.



Taxonomy of Crash-Recovery Algorithms

- Combined undo and redo necessary to guarantee that
  - an action of transaction *t* is reflected in the stable database only if commit appears in stable log,
  - if stable log includes a commit of *t*, all actions of *t* are reflected in the stable database.

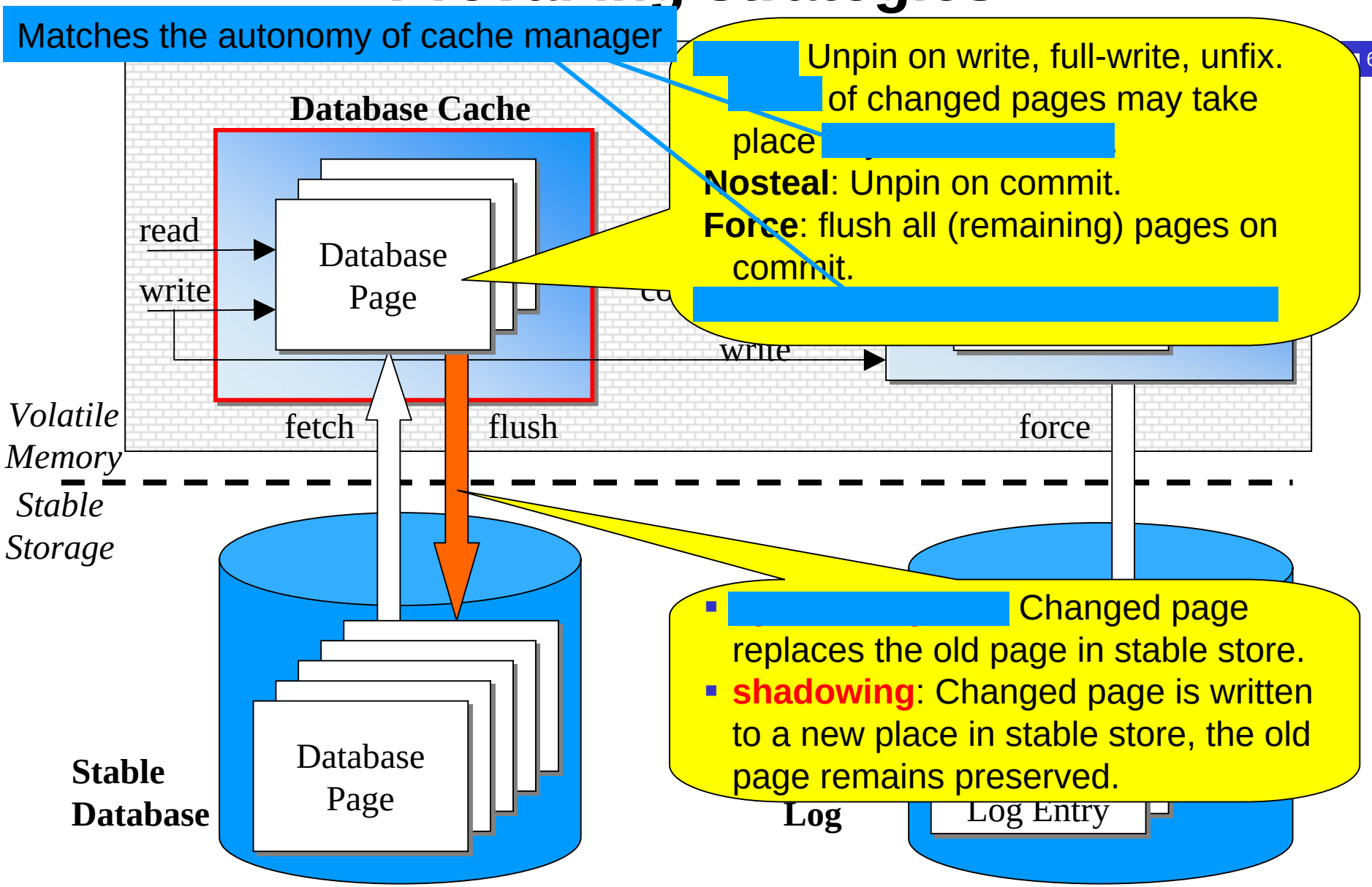


Evaluation

option	cost
no-undo / no-redo	Immediate restart possible, but extremely costly during normal operation!
no-undo	Inexpensive with shadowing, but restricts available locking options and performance.
no-redo	Heavy page transfer load on commit, more efficient to force the log.

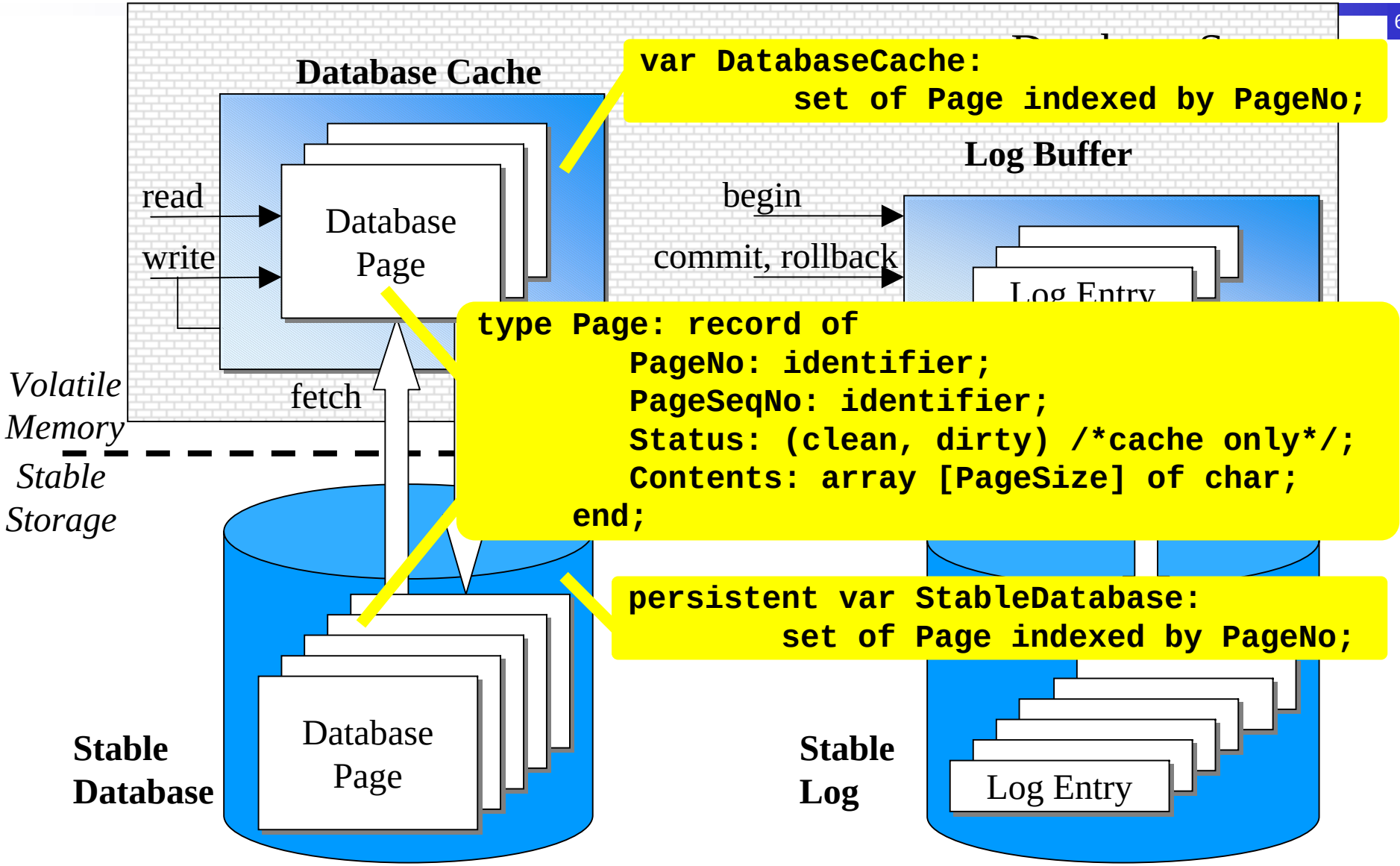
Conclusion: with-undo/with-redo algorithms are the common choice because the higher cost on restart is more than offset by lower cost during normal operation!

Prevailing strategies



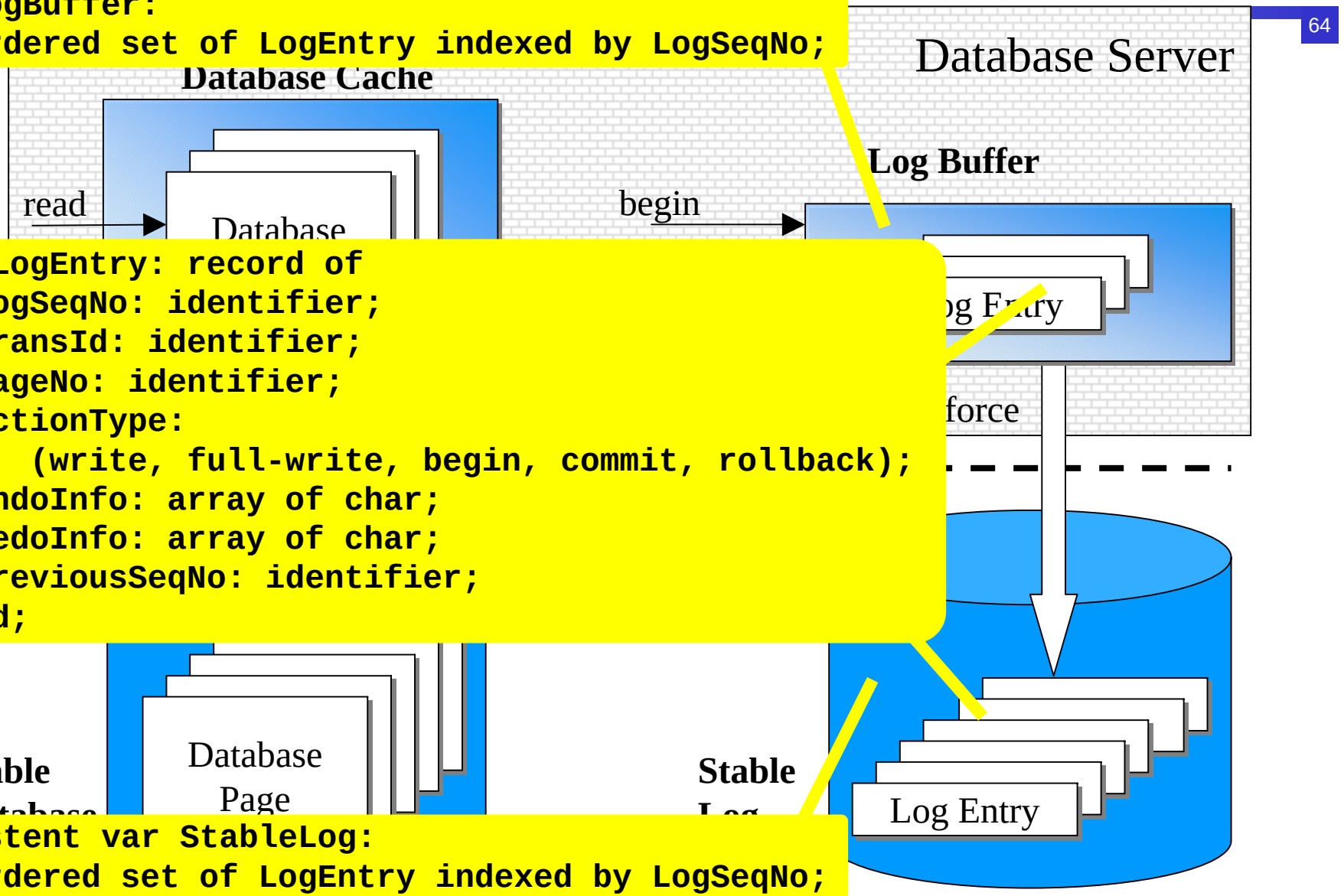
Normal operations

Basic Data Structures for Crash Recovery (1)



Basic Data Structures for Crash Recovery (2)


```
var LogBuffer:
    ordered set of LogEntry indexed by LogSeqNo;
```

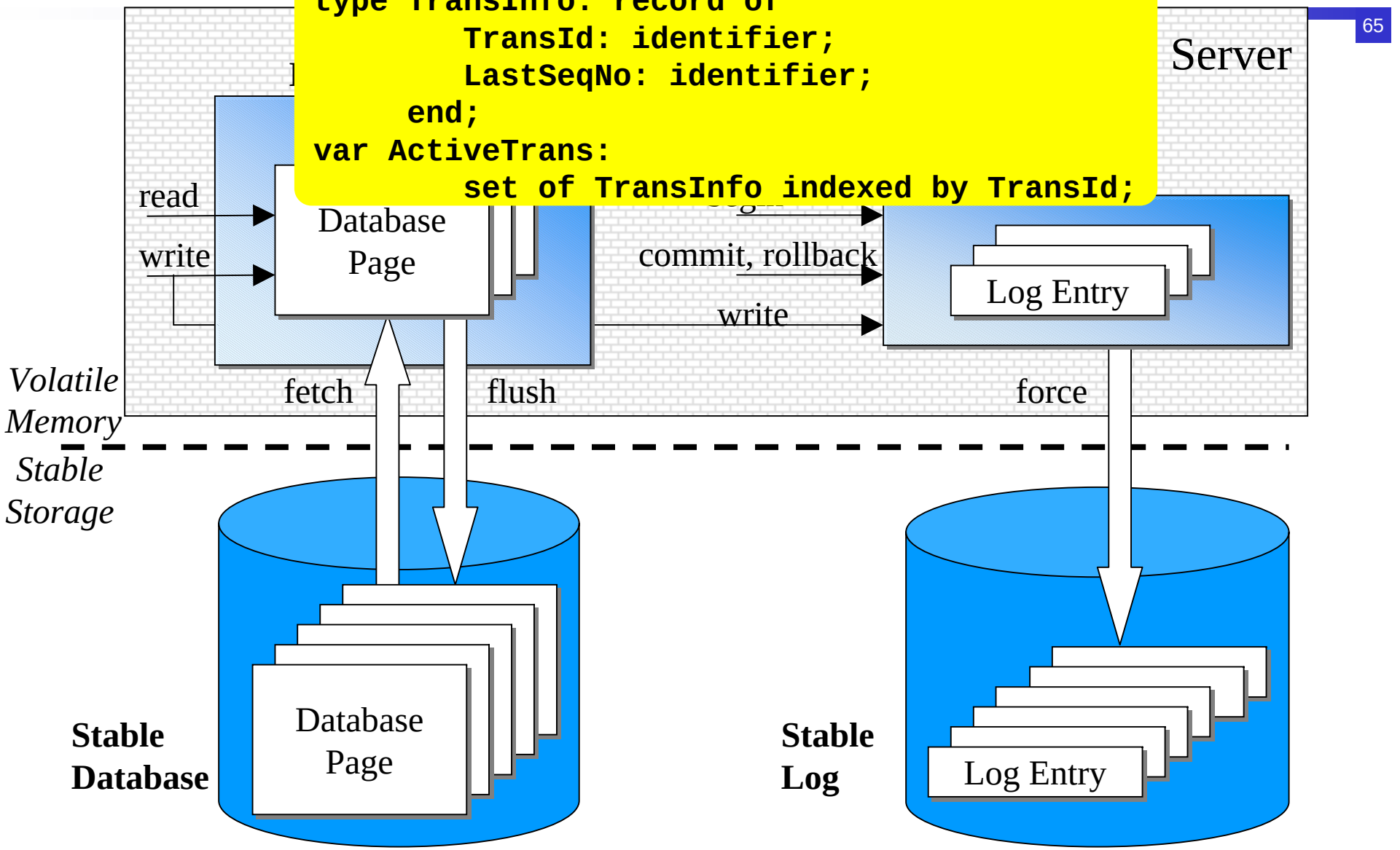



```
type LogEntry: record of
    LogSeqNo: identifier;
    TransId: identifier;
    PageNo: identifier;
    ActionType:
       (write, full-write, begin, commit, rollback);
    UndoInfo: array of char;
    RedoInfo: array of char;
    PreviousSeqNo: identifier;
   end;
```

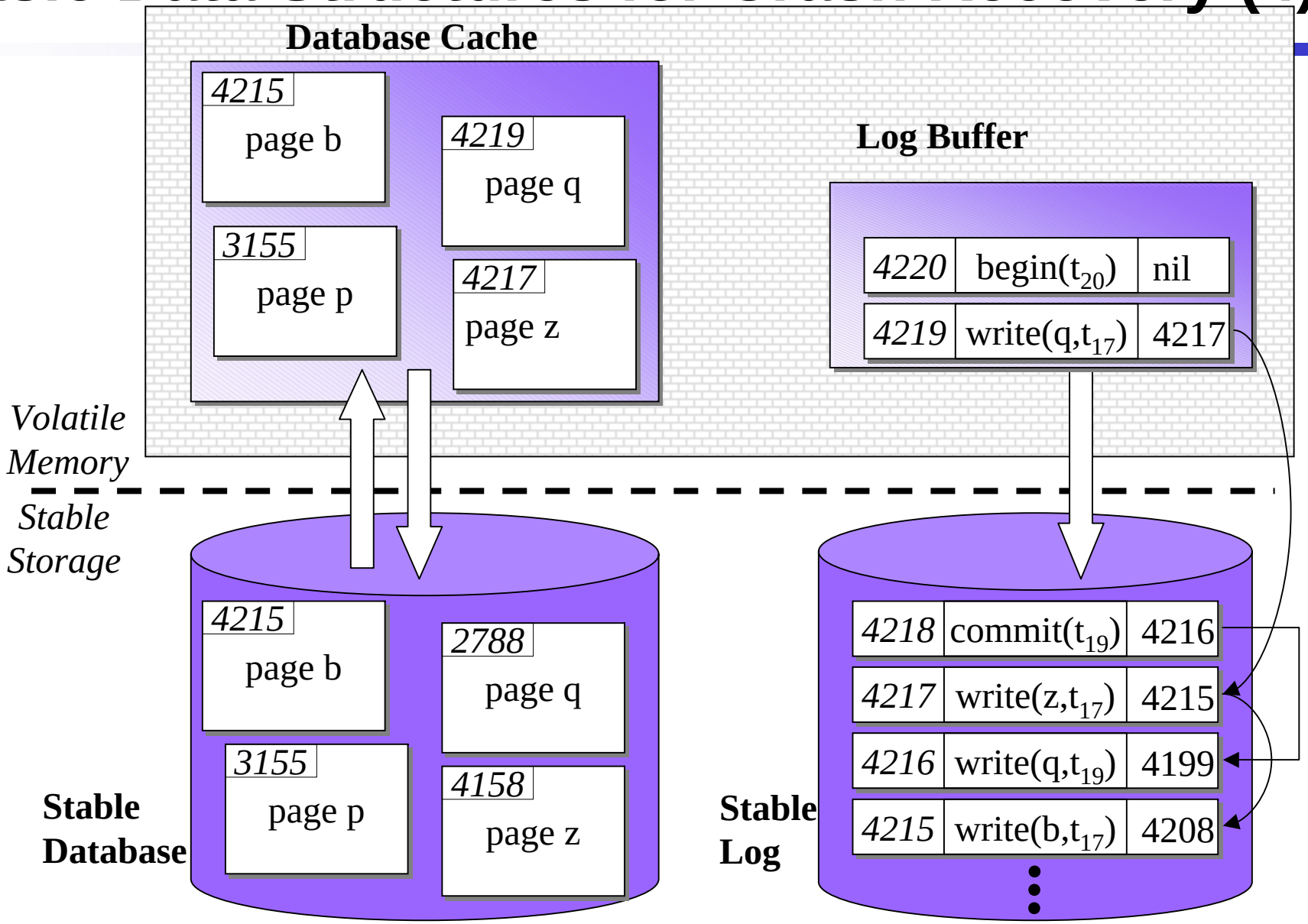
```
persistent var StableLog:
    ordered set of LogEntry indexed by LogSeqNo;
```

Basic Data Structures for Crash Recovery (3)

```
type TransInfo: record of
        TransId: identifier;
        LastSeqNo: identifier;
    end;
var ActiveTrans:
        set of TransInfo indexed by TransId;
```

Basic Data Structures for Crash Recovery (4)



Interrelationships


```
type TransInfo: record of
        TransId: identifier;
        LastSeqNo: identifier;
    end;
var ActiveTrans:
        set of TransInfo indexed by TransId;
```

Maximum LSN of a log entry for the transaction.

```
type LogEntry: record of
    LogSeqNo: identifier;
    TransId: identifier;
    PageNo: identifier;
    ActionType:
       (write, full-write, begin, commit, rollback);
    UndoInfo: array of char;
    RedoInfo: array of char;
    PreviousSeqNo: identifier;
   end;
```

LSN: Assigned in chronological order. Agrees with log ordering.

Chaining within transaction.

```
type Page: record of
        PageNo: identifier;
        PageSeqNo: identifier;
        Status: (clean, dirty) /* only cache*/;
        Contents: array [PageSize] of char;
    end;
```

equals maximum LSN of a log entry for the page.

Summary of data structures


```
var DatabaseCache:
      set of Page indexed by PageNo;
type Page: record of
        PageNo: identifier;
        PageSeqNo: identifier;
        Status: (clean, dirty) /* only cache*/;
        Contents: array [PageSize] of char;
     end;
var LogBuffer:
    ordered set of LogEntry indexed by LogSeqNo;
type LogEntry: record of
    LogSeqNo: identifier;
    TransId: identifier;
    PageNo: identifier;
    ActionType:
       (write, full-write, begin, commit, rollback);
    UndoInfo: array of char;
    RedoInfo: array of char;
    PreviousSeqNo: identifier;
   end;
```

```
type TransInfo: record of
        TransId: identifier;
        LastSeqNo: identifier;
     end;
var ActiveTrans:
    set of TransInfo indexed by TransId
```

Actions During Normal Operation (1)


```
write or full-write (pageno, transid, s):
    DatabaseCache[pageno].Contents := modified contents;
    DatabaseCache[pageno].PageSeqNo := s;
    DatabaseCache[pageno].Status := dirty;
    newlogentry.LogSeqNo := s;
    newlogentry.ActionType := write or full-write;
    newlogentry.TransId := transid;
    newlogentry.PageNo := pageno;
    newlogentry.UndoInfo := information to undo update
         (before-image for full-write);
    newlogentry.RedoInfo := information to redo update
         (after-image for full-write);
    newlogentry.PreviousSeqNo :=
          ActiveTrans[transid].LastSeqNo;
    ActiveTrans[transid].LastSeqNo := s;
    LogBuffer += newlogentry;
```

log data entry

Actions During Normal Operation (2)


```
fetch (pageno):
   Find slot in DatabaseCache and set its
       PageNo := pageno;
   DatabaseCache[pageno].Contents :=
        StableDatabase[pageno].Contents;
   DatabaseCache[pageno].PageSeqNo :=
        StableDatabase[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;

flush (pageno):
   if there is logentry in LogBuffer
      with logentry.PageNo = pageno
   then force ( ); end /*if*/;
   StableDatabase[pageno].Contents :=
        DatabaseCache[pageno].Contents;
   StableDatabase[pageno].PageSeqNo :=
        DatabaseCache[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;

force ( ):
     StableLog += LogBuffer;
     LogBuffer := empty;
```

write-ahead

May have been flushed earlier by log buffer manager!

Actions During Normal Operation (3)


```
begin (transid, s):
    newtransentry.TransId := transid;
    ActiveTrans += newtransentry;
    ActiveTrans[transid].LastSeqNo := s;
    newlogentry.LogSeqNo := s;
    newlogentry.ActionType := begin;
    newlogentry.TransId := transid;
    newlogentry.PreviousSeqNo := nil;
    LogBuffer += newlogentry;

commit (transid, s):
    newlogentry.LogSeqNo := s;
    newlogentry.ActionType := commit;
    newlogentry.TransId := transid;
    newlogentry.PreviousSeqNo :=
        ActiveTrans[transid].LastSeqNo;
    LogBuffer += newlogentry;
    ActiveTrans -= transid;
    force ( );
```

log BOT entry

log commit entry

stable log!

Correctness

Definition 8.13 (Logging Rules):
During normal operation, a recovery algorithm satisfies

- the **redo logging rule** if for every committed transaction *t*, all data actions of *t* are in stable storage (the stable log or the stable database),
- the **undo logging rule** if for every data action *p* of an uncommitted transaction *t* the presence of *p* in the stable database implies that *p* is in the stable log,
- the **garbage collection rule** if for every data action *p* of transaction *t* the absence of *p* from the stable log implies that *p* is in the stable database if and only if *t* is committed.

Theorem 8.16:
During normal operation, the redo logging rule, the undo logging rule, and the garbage collection rule are satisfied.

Efficiency

Forced log I/O is potential bottleneck during normal operation
→ **group commit** for log I/O batching

Redo-Winners:
Three-pass recovery algorithm

Winners and losers

- **Basic assumption:**
  - Transactions that have been aborted have been rolled back during normal operation and, hence, can be disregarded.
- **Winner:** Winner transactions are those transactions for which a commit log entry is encountered.
- **Loser:** Loser transactions are those transactions for which no commit entry exists in the stable log, i.e., that were still active during system crash.
- **Technical assumption:**
  - All write actions during normal operation are full-writes.

Simple Three-Pass Algorithm

- **Analysis pass:**
  - Determine start of stable log from master record.
  - Perform forward scan to determine winners and losers.
- **Redo pass:**

  Perform forward scan to redo all winner actions in chronological (LSN) order (until end of log is reached).
  - Re-executes committed history.

RG (COCSR)!

- **Undo pass:**

  Perform backward scan to traverse all loser log entries in reverse chronological order and undo the corresponding actions.
  - These must follow the last committed transaction in the serial order, hence their effects can be undone without endangering the committed transactions.

ST or RG!

Simple Three-Pass Algorithm


```
restart ( ):
    analysis pass ( ) returns losers;
    redo pass ( );
    undo pass ( );
    for each page p in DatabaseCache do
        if DatabaseCache[p].Status = dirty then flush (p);
        end /*if/;
        end /*for*/;
    reinitialize StableLog;
```

Three passes

Because undo/redo use the cache, the cache must be cleared before resuming normal operation.

Analysis Pass


```
analysis pass ( ) returns losers:
var losers: set of record
                TransId: identifier;
                LastSeqNo: identifier;
              end /*indexed by TransId*/;
    losers := empty;
    min := LogSeqNo of oldest log entry in StableLog;
    max := LogSeqNo of most recent log entry in StableLog;
    for i := min to max do
        case StableLog[i].ActionType:
            begin: losers += StableLog[i].TransId;
                   losers[StableLog[i].TransId].LastSeqNo := nil;
            commit: losers -= StableLog[i].TransId;
            full-write: losers[StableLog[i].TransId].LastSeqNo := i;
        end /*case*/;
    end /*for*/;
```

We register losers only.

Each ta is a potential loser.

ta is a winner!

Forward scan

Redo Pass


```
redo pass ( ):
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max
   do
       if StableLog[i].ActionType = full-write and
          StableLog[i].TransId not in losers
       then
          pageno = StableLog[i].PageNo;
          fetch (pageno);
          full-write (pageno)
             with contents from StableLog[i].RedoInfo;
       end /*if*/;
   end /*for*/;
```

Restore the winner no matter how much of it is already in stable database!

Forward scan: restore latest valid state.

Undo Pass


```
undo pass ( ):
    while there exists t in losers
          such that losers[t].LastSeqNo <> nil
    do
        nexttrans = TransId in losers
           such that losers[nexttrans].LastSeqNo =
           max {losers[x].LastSeqNo | x in losers};
        nextentry = losers[nexttrans].LastSeqNo;
        if StableLog[nextentry].ActionType = full-write
        then
           pageno = StableLog[nextentry].PageNo;
           fetch (pageno);
           full-write (pageno)
              with contents from StableLog[nextentry].UndoInfo;
           losers[nexttrans].LastSeqNo :=
              StableLog[nextentry].PreviousSeqNo;
        end /*if*/;
    end /*while*/;
```

Set of losers not yet exhausted?

Simulate backward scan on operations

Restore the loser no matter how much of it still is in stable database!

Loser's next entry

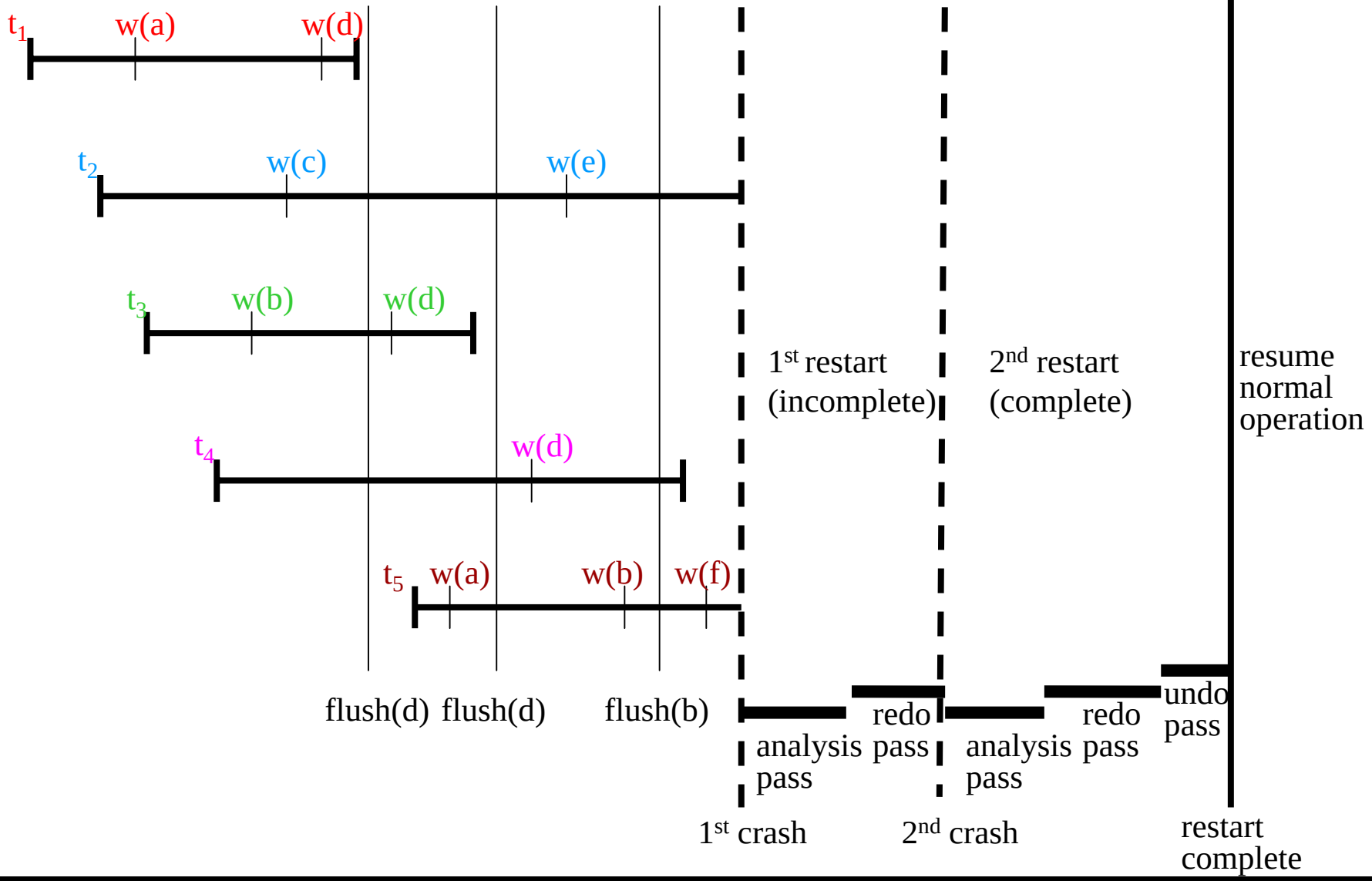
Backward scan: restore latest valid state.

Correctness

Theorem 8.17:
When restricted to full-writes as data actions, the simple three-pass recovery algorithm performs correct recovery.

For those interested in the proof :
Weikum & Vossen, pp.461-465

Sample Scenario



Sample Scenario Data Structures

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
↯SYSTEM CRASH↯				

First restart

Analysis pass: losers = $\{t_2, t_5\}$

Redo pass + ↯:

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
redo (6)	b: 6			
flush (a)		a: 3		
redo (8)	d: 8			
flush (d)		d: 8		
redo (11)	d: 11			
↯SECOND SYSTEM	CRASH↯			

Second restart

Analysis pass: losers = $\{t_2, t_5\}$

Redo pass + undo pass:

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
redo (6)	b: 6			
redo (8)	d: 8			
redo (11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 6			
undo(17)	e: 0			
undo(13)	a: 3			
undo(7)	c: 0			
SECOND RESTART	COMPLETE: RESUME	NORMAL OPERATION		

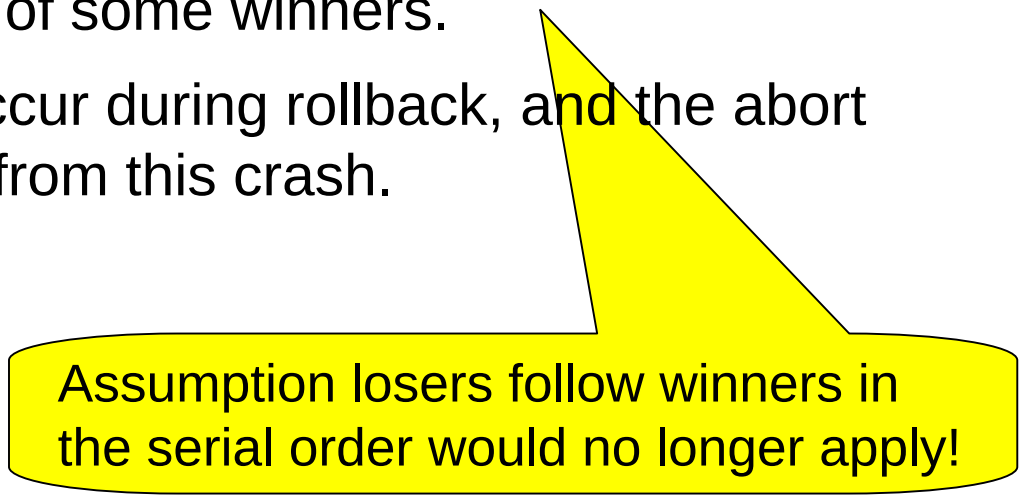
General writes

- **Full-write:** Entire page is rewritten no matter how small the updated part of the page.
  - When writing/ replacing the page all earlier writes on the page are included.
- **General write:** Shorter log by logging only the local change on the page.
  - More complicated redo/undo based on the page sequence numbers.
  - Local change must be intercepted by cache manager.

- We stay with full-write.
- See Weikum/Vossen 467-473, where all examples thereafter are based on general write.

Transaction aborts

- **Problem:** Write actions are logged before the outcome of the transaction becomes known.
  - Transaction abort requires a rollback.
  - All logging information would then have to be removed. Otherwise there would be a loser whose undo on recovery must precede the redo of some winners.
  - A system crash may occur during rollback, and the abort would have to recover from this crash.



Transaction rollback

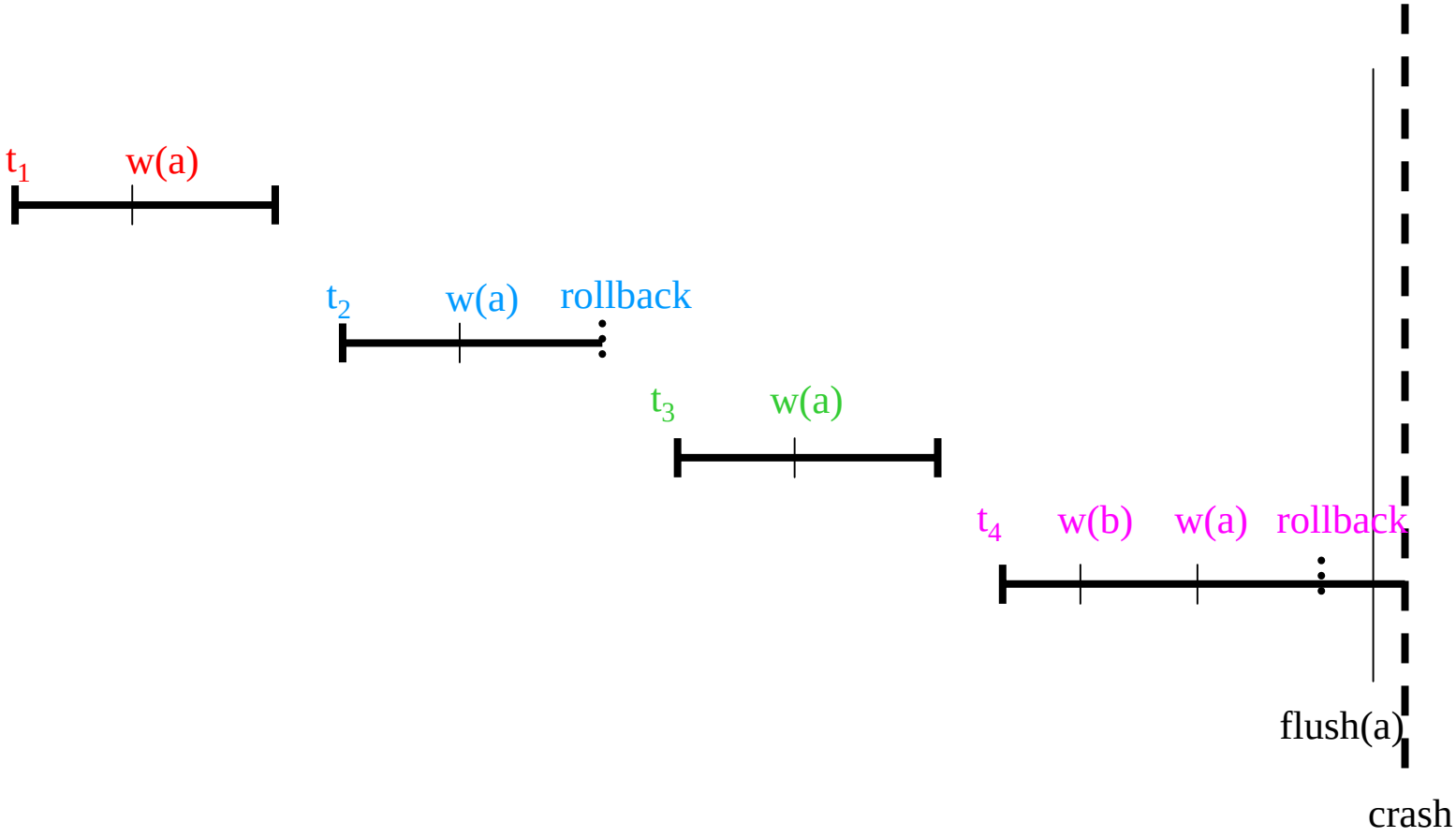
- **Solution:** Treat a rolled back transaction as a winner.
  - Create compensation log entries for inverse operations during transaction rollback.
  - Complete rollback by creating rollback log entry.
  - During crash recovery, aborted transactions with complete rollback are winners (and are redone in the right serial order), incomplete aborted transactions are losers.

Theorem 8.18:
The extension for handling transaction rollbacks during normal operation preserves the correctness of the three-pass algorithm.

Transaction rollback


```
abort (transid):
  logentry := ActiveTrans[transid].LastSeqNo;
  while logentry is not nil and
        logentry.ActionType = write or full-write do
    newlogentry.LogSeqNo := new sequence number;
    newlogentry.ActionType := compensation;
    newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
    newlogentry.RedoInfo :=
       inverse action of the action in logentry;
    newlogentry.UndoInfo :=
       inverse action of inverse action of action in logentry;
    ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
    LogBuffer += newlogentry;
    write (logentry.PageNo) according to logentry.UndoInfo;
    logentry := logentry.PreviousSeqNo;
   end /*while*/
   newlogentry.LogSeqNo := new sequence number;
   newlogentry.ActionType := rollback;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   LogBuffer += newlogentry; ActiveTrans -= transid; force ( );
```

Sample Scenario



Sample Scenario Data Structures

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: write (a, t_1)	a: 2		2: write (a, t_1)	
3: commit (t_1)			3: commit (t_1)	1, 2, 3
4: begin (t_2)			4: begin (t_2)	
5: write (a, t_2)	a: 5		5: write (a, t_2)	
6: abort (t_2)				
7: compensate(5)	a: 7		7: compensate (a, t_2)	
8: rollback (t_2)			8: rollback (t_2)	4, 5, 7, 8
9: begin (t_3)			9: begin (t_3)	
10: write (a, t_3)	a: 10		10: write (a, t_3)	
11: commit (t_3)			11: commit (t_3)	9, 10, 11
12: begin (t_4)			12: begin (t_4)	
13: write (b, t_4)	b: 13		13: write (b, t_4)	
14: write (a, t_4)	a: 14		14: write (a, t_4)	
15: abort (t_4)				
16: compensate(14)	a: 16		16: compensate (a, t_4)	
17: flush (a)		a: 16		12, 13, 14, 16
↯SYSTEM CRASH↯				

Restart

Analysis pass: losers = $\{t_4\}$

Redo pass + undo pass:

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
redo (2)	a: 2			
redo (5)	a: 5			
redo (7)	a: 7			
redo (10)	a: 10			
undo(16)	a: 14			
undo(14)	a: 10			
undo(13)	b: 0			
RESTART	COMPLETE: RESUME	NORMAL OPERATION		

Redo-Winners: Checkpoints

Bottlenecks

- The analysis pass has to scan the entire stable log. ⇒ Hours of server operation even though often only the last five minutes contain loser transactions (were active on crash).
- The redo pass has to scan the entire stable log. ⇒ Hours of server operation even though often most winner transactions are already in the stable database due to flush actions.
- The redo pass incurs many expensive page fetches from the stable database, many of them unnecessary.

Minimum: Log truncation (1)

- Truncate redo log entries:
  - RedoLSN(*p*) = sequence number of write of *p* right after last flush.
  - For redo, can drop all log entries for *p* that precede RedoLSN(*p*) because corresponding actions are already in the stable database
  - Still needed: Start pointer SystemRedoLSN = min{RedoLSN(*p*) | dirty page *p*}.
- Truncate undo log entries:
  - OldestUndoLSN = sequence number of oldest write of an active transaction.
  - For undo, can drop all log entries that precede OldestUndoLSN because entry is only needed as long as the corresponding transaction is not yet completed.
- Periodic log truncation: Flush and move forward.

Minimum: Log truncation


```
log truncation ( ):
    OldestUndoLSN :=
         min {i | StableLog[i].TransId is in ActiveTrans};
    SystemRedoLSN := min {DatabaseCache[p].RedoLSN};
    OldestRedoPage := page p such that
         DatabaseCache[p].RedoLSN = SystemRedoLSN;
    NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    OldStartPointer := MasterRecord.StartPointer;
    while NewStartPointer - OldStartPointer
                is not sufficiently large
       and SystemRedoLSN < OldestUndoLSN
    do
         flush (OldestRedoPage);
         SystemRedoLSN := min{DatabaseCache[p].RedoLSN};
         OldestRedoPage := page p such that
               DatabaseCache[p].RedoLSN = SystemRedoLSN;
         NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    end /*while*/;
    MasterRecord.StartPointer := NewStartPointer;
```

Extension!

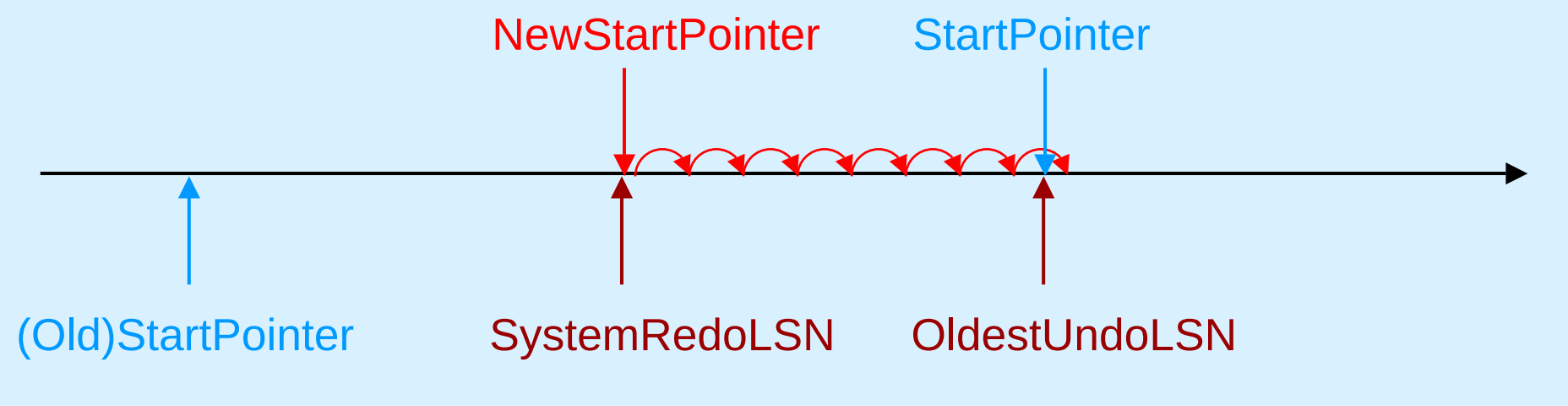
start truncation here

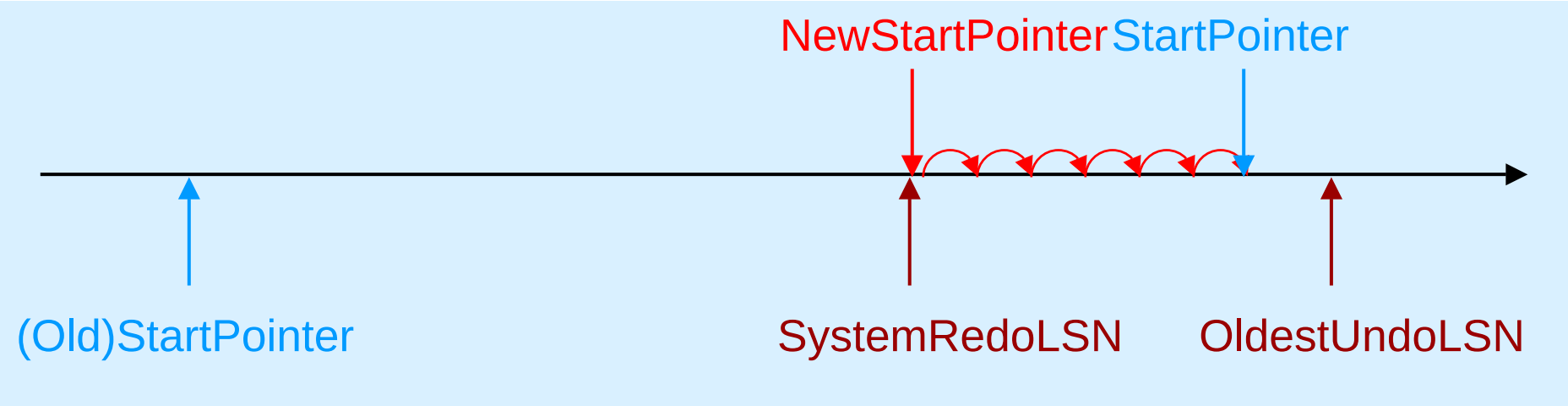
last truncation stopped here

make sure page is in stable database

move forward and try again

Minimum: Log truncation



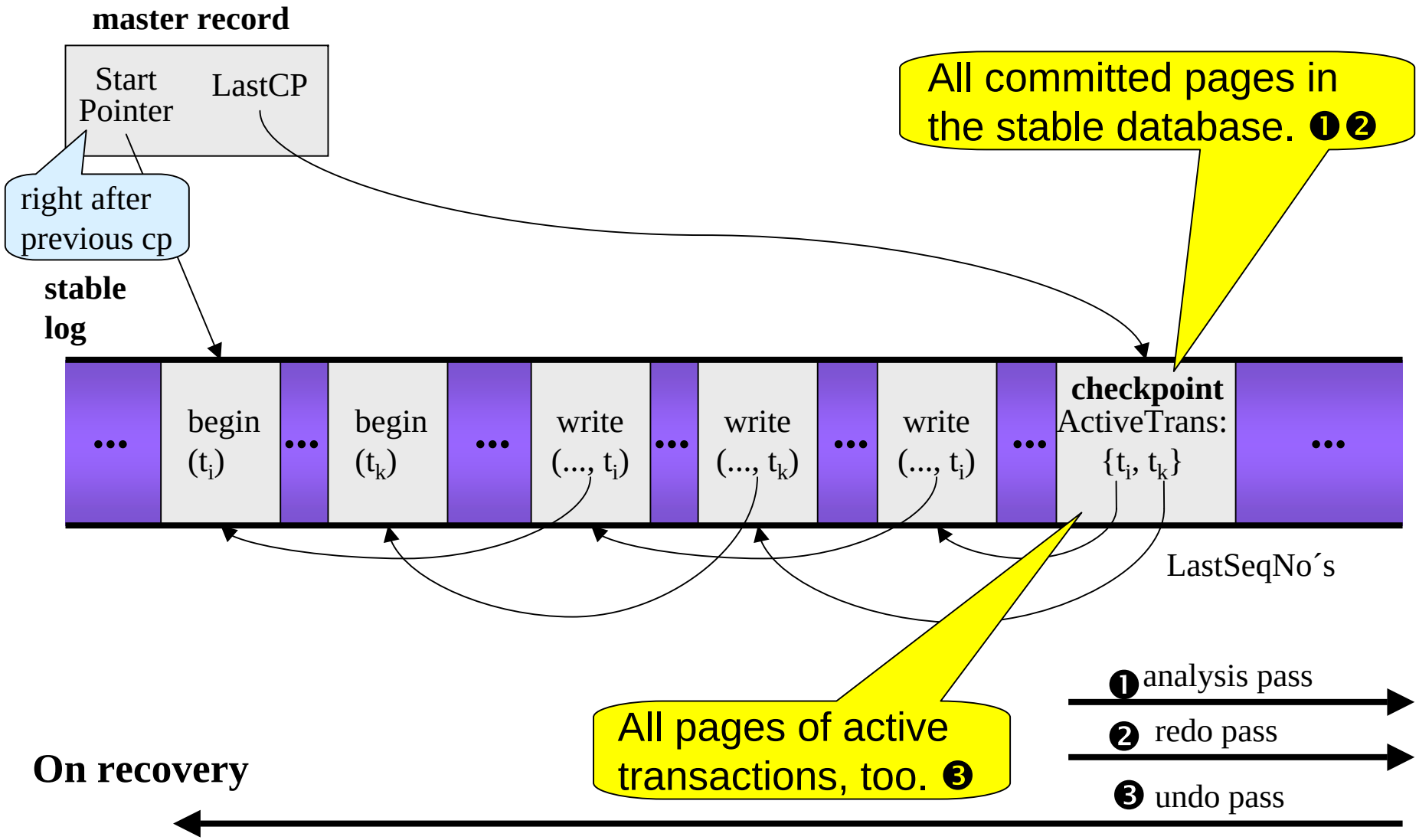


Improvement: Checkpoint

Checkpoint: log truncation followed by flushing all dirty pages to the stable database.

- **Heavy checkpoint:** log truncation is immediately followed by flushing.
  - „heavy“: flushing incurs substantial work during normal operation.
  - taken periodically (system parameter!).
- **Light checkpoint:** log truncation is followed by a subsequent background process (write-behind daemon).
  - can be taken continuously.

Heavy-Weight Checkpoints



Heavy-Weight Checkpoints

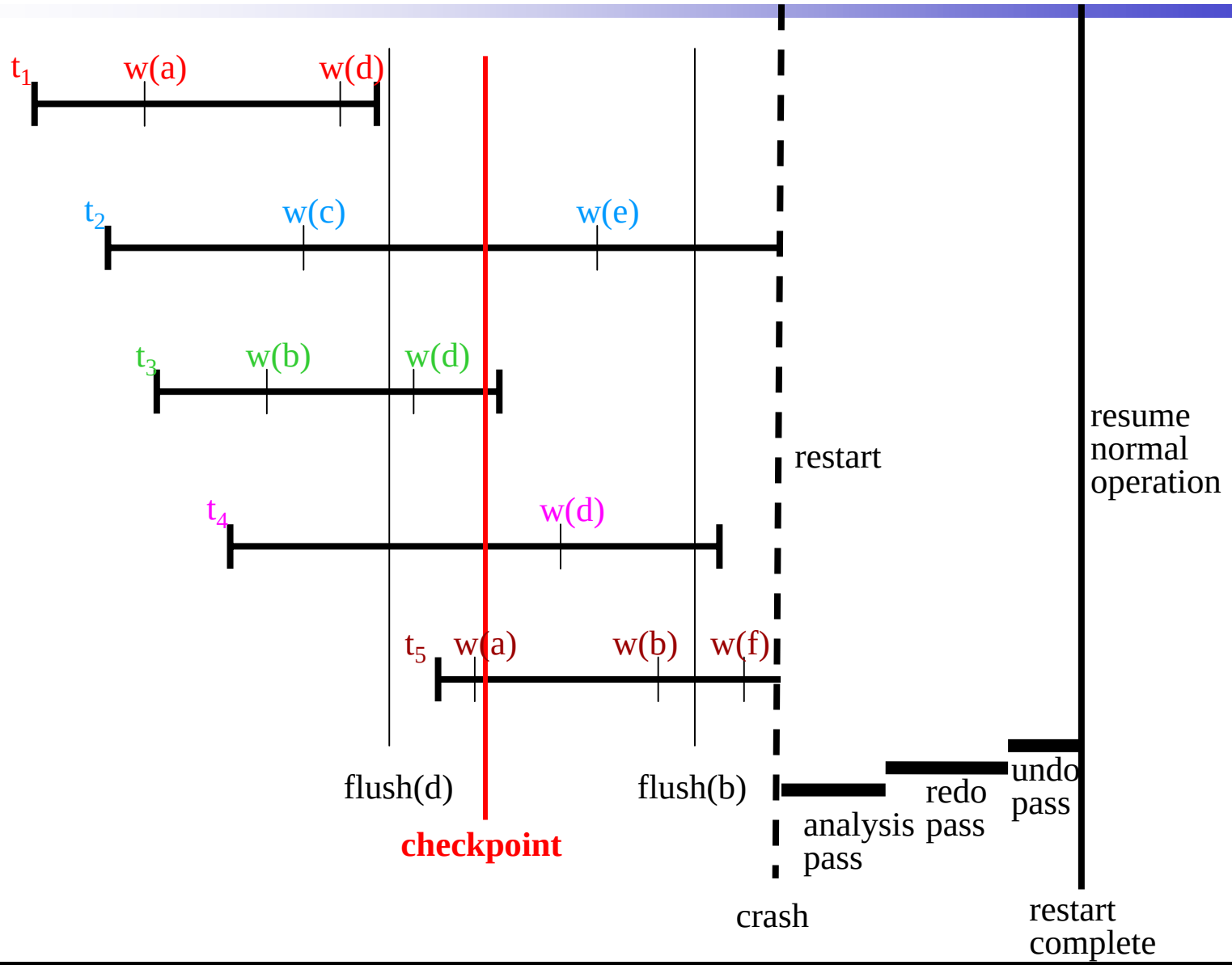

```
checkpoint ( ):
    for each p in DatabaseCache do
        if DatabaseCache[p].Status = dirty
        then flush (p);
        end /*if*/;
     end /*for*/;
    logentry.ActionType := checkpoint;
    logentry.ActiveTrans :=
        ActiveTrans (as maintained in memory);
    logentry.LogSeqNo := new sequence number;
    LogBuffer += logentry;
    force ( );
    MasterRecord.LastCP := logentry.LogSeqNo;
```

Remember: flush implies a preceding force

Heavy-Weight Checkpoints


```
analysis pass ( ) returns losers:
   cp := MasterRecord.LastCP;
   losers := StableLog[cp].ActiveTrans;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
        case StableLog[i].ActionType:
             ...
             maintenance of losers
                  as in the algorithm without checkpoints
             ...
        end /*case*/;
   end /*for*/;
redo pass ( ):
   cp := MasterRecord.LastCP;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
         ...
         page-state-testing and redo steps
              as in the algorithm without checkpoints
         ...
   end /*for*/;
```

Sample Scenario



Sample Scenario Data Structures

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: checkpoint		a: 13, b: 6, c: 7, d: 11	14: CP ActiveTrans: {t_2, t_3, t_4, t_5}	11, 12, 13 14
15: commit (t_3)			15: commit (t_3)	15
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
↯SYSTEM CRASH↯				

Restart

Analysis pass: losers = $\{t_2, t_5\}$

Redo pass + undo pass:

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
redo(16)	d: 16			
undo(18)	b: 6			
undo(17)	e: 0			
undo(13)	a: 3			
undo(7)	c: 0			
RESTART	COMPLETE: RESUME	NORMAL OPERATION		

Light-Weight Checkpoints

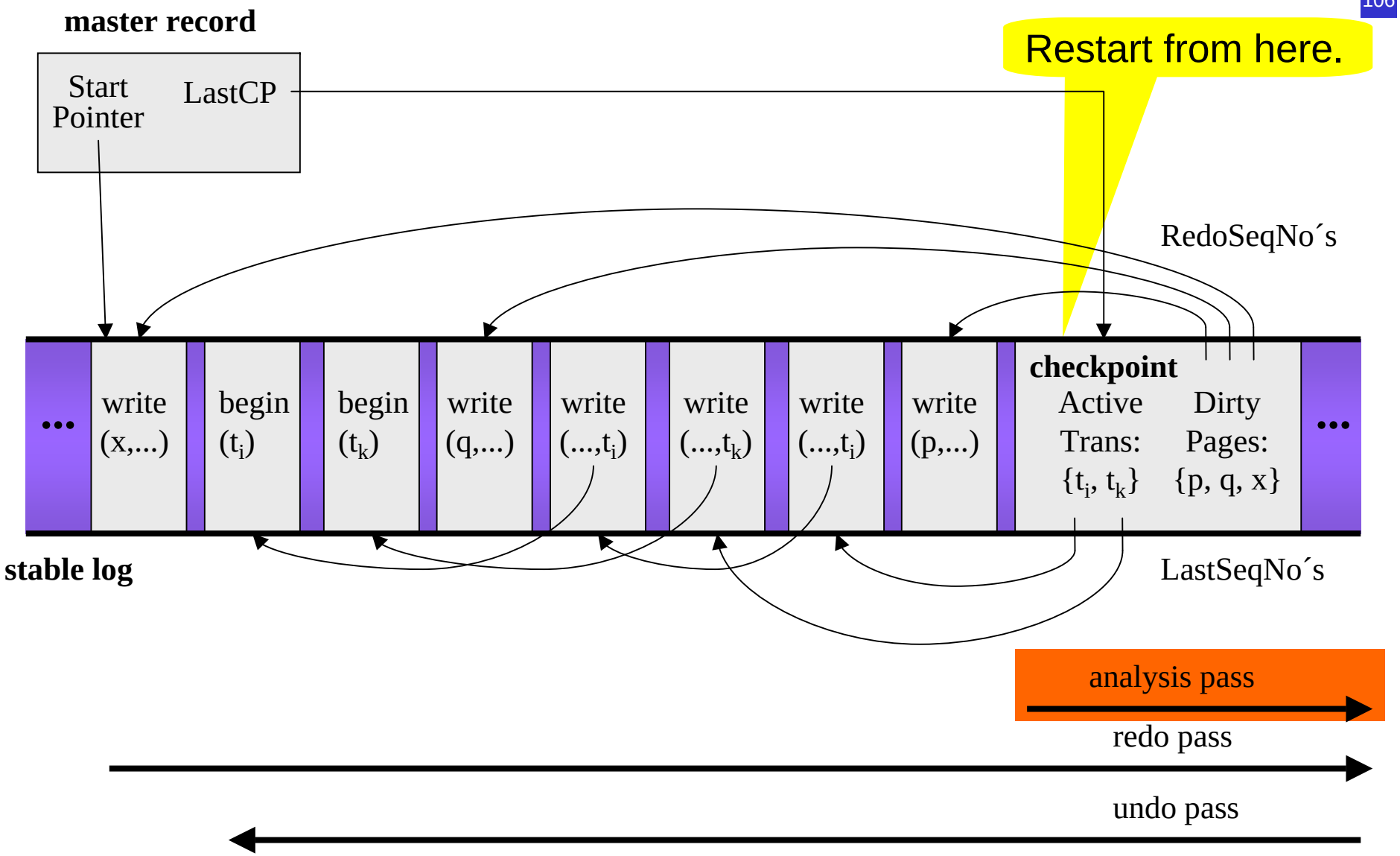

```
checkpoint ( ):
    DirtyPages := empty;
    for each p in DatabaseCache do
        if DatabaseCache[p].Status = dirty
        then
            DirtyPages += p;
            DirtyPages[p].RedoSeqNo :=
                DatabaseCache[p].RedoLSN;
        end /*if*/;
     end /*for*/;
    logentry.ActionType := checkpoint;
    logentry.ActiveTrans :=
        ActiveTrans (as maintained in memory);
    logentry.DirtyPages := DirtyPages;
    logentry.LogSeqNo := new sequence number;
    LogBuffer += logentry;
    force ( );
    MasterRecord.LastCP := logentry.LogSeqNo;
```

Determine dirty pages from the cache together with their RedoLSNs

Construct checkpoint log entry

Write to stable log

Light-Weight Checkpoints



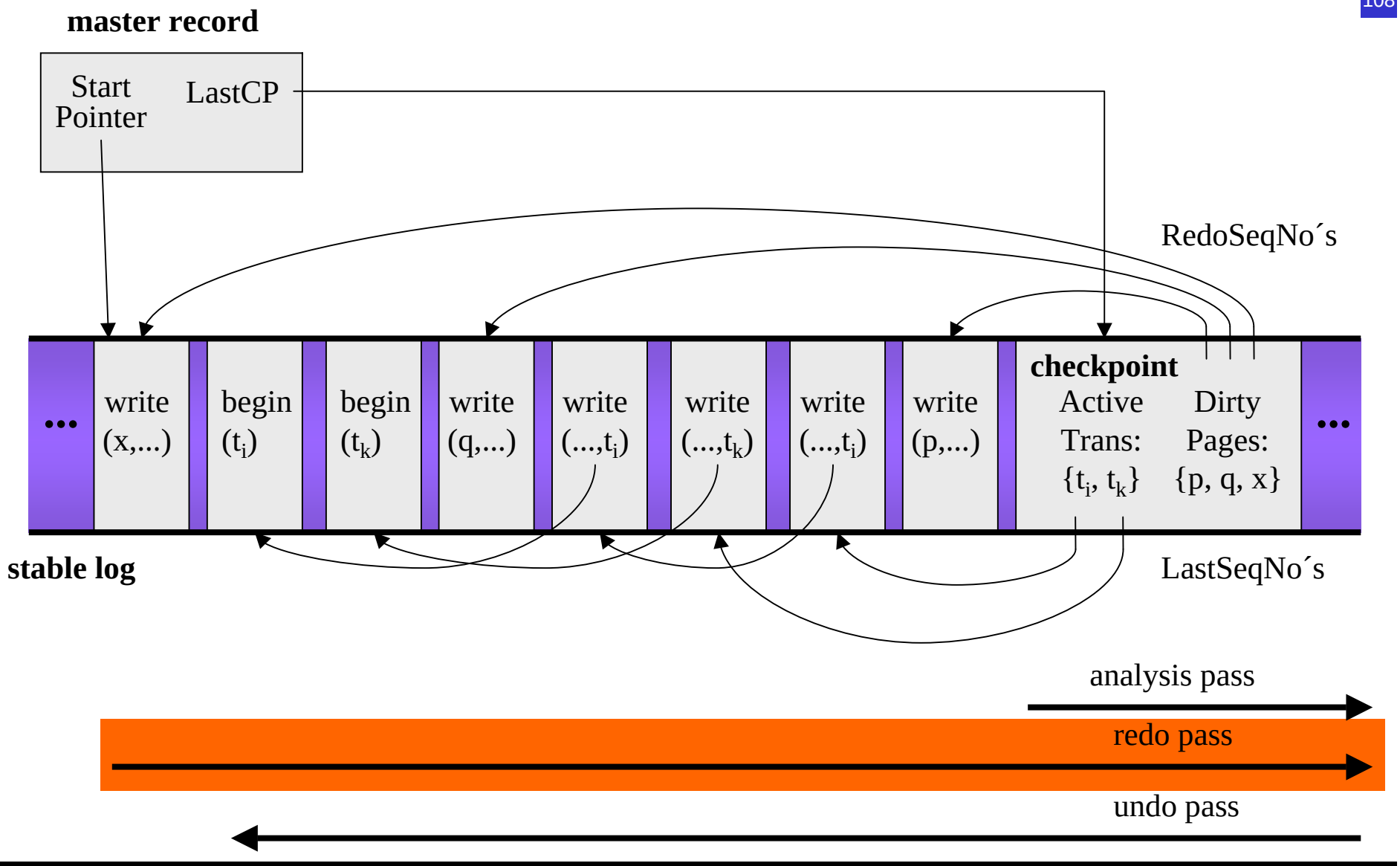
Light-Weight Checkpoints


```
analysis pass ( ) returns losers, DirtyPages:
   cp := MasterRecord.LastCP;
   losers := StableLog[cp].ActiveTrans;
   DirtyPages := StableLog[cp].DirtyPages;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
        case StableLog[i].ActionType:
             ...
             maintenance of losers
                  as in the algorithm without checkpoints
             ...
        end /*case*/;
        if StableLog[i].ActionType = write or full-write
              and StableLog[i].PageNo not in DirtyPages
        then
              DirtyPages += StableLog[i].PageNo;
              DirtyPages[StableLog[i].PageNo].RedoSeqNo := i;
        end /*if*/;
   end /*for*/;
```

Determine losers and chain their entries as before

Add new dirty pages but record only the oldest

Light-Weight Checkpoints



Light-Weight Checkpoints


```
redo pass ( ):
    cp := MasterRecord.LastCP;
    SystemRedoLSN := min{cp.DirtyPages[p].RedoSeqNo};
    max := LogSeqNo of most recent log entry in StableLog;
    for i := SystemRedoLSN to max do
        if StableLog[i].ActionType = write or full-write
             and StableLog[i].TransId not in losers
        then
            pageno := StableLog[i].PageNo;
            if pageno in DirtyPages
               and i >= DirtyPages[pageno].RedoSeqNo
            then
                fetch (pageno);
                if DatabaseCache[pageno].PageSeqNo < i
                then
                    read and write (pageno)
                       according to StableLog[i].RedoInfo;
                    DatabaseCache[pageno].PageSeqNo := i;
                else
                    DirtyPages[pageno].RedoSeqNo :=
                         DatabaseCache[pageno].PageSeqNo + 1;
                end/*if*/; end/*if*/; end/*if*/; end/*for*/;
```

check if log content is a candidate for redo

page is not in latest state, redo it

daemon was already here, record latest flush

Sample Scenario Data Structures

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: checkpoint			14: CP DirtyPages: {a, b, c, d} RedoLSNs: {a: 3, b: 6, c: 7, d: 11} ActiveTrans: {t_2, t_3, t_4, t_5}	11, 12, 13 14
15: commit (t_3)			15: commit (t_3)	15
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
↯SYSTEM CRASH↯				

Restart

Analysis pass: losers = {t_2, t_5}
DirtyPages = {a, b, c, d, e}
RedoLSNs: a: 3, b: 6, c: 7, d: 11, e: 18

Redo pass + undo pass:

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
redo (6)	b: 6			
skip-redo (8)	d: 8			
redo (11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 6			
undo(17)	e: 0			
undo(13)	a: 3			
undo(7)	c: 0			
RESTART	COMPLETE: RESUME	NORMAL OPERATION		

Correctness

Theorem 8.18:
Extending the simple three-pass recovery algorithm with log truncation, heavy-weight or light-weight checkpoints preserves the correctness of crash recovery.

Redo-history

Weakness of redo-winners

- On completion of restart there must be no trace of losers within the system.
- Consequence: Costly flush of database cache.

```
restart ( ):
   analysis pass ( ) returns losers;
   redo pass ( );
   undo pass ( );
   for each page p in DatabaseCache do
      if DatabaseCache[p].Status = dirty then flush (p);
      end /*if/;
      end /*for*/;
   reinitialize StableLog;
```

Basic idea

- In Redo-History, *all* actions are repeated in chronological order, i.e.,
  1. it first reconstructs the cached database,
  2. then treats losers as if they were still active and had just been aborted.
  - Expense is in double work of undoing losers.
  - But: Stable log can be reused $\Rightarrow$ no flushes needed after restart.

Simple algorithm

- *Optional* **analysis** pass
  - determines losers and reconstructs DirtyPages list,
  - using the analysis algorithm of the redo-winners paradigm.
- **Redo** pass starts from SystemRedoLSN and
  - redoes *both* winner and loser updates, with LSN-based state testing for idempotence,
  - to reconstruct the current database close to the time of the crash.
- **Undo** pass initiates *rollback* for all loser transactions, using the code for rollback during normal operation, with undo steps (without page state testing)
  - creating *compensation log entries* and
  - *advancing* page sequence numbers

Simple algorithm


```
redo pass ( ):
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max do
          pageno = StableLog[i].PageNo;
          fetch (pageno);
          if DatabaseCache[pageno].PageSeqNo < i
          then
             read and write (pageno)
                  according to StableLog[i].RedoInfo;
             DatabaseCache[pageno].PageSeqNo := i;
          end /*if*/;
   end /*for*/;
```

Just replace page, though only if the entry is younger

Simple algorithm


```
undo pass ( ):
    ActiveTrans := empty;
    for each t in losers do
          ActiveTrans += t;
          ActiveTrans[t].LastSeqNo := losers[t].LastSeqNo;
    end /*for*/;
    while there exists t in losers
          such that losers[t].LastSeqNo <> nil do
      nexttrans := TransNo in losers
           such that losers[nexttrans].LastSeqNo
           max {losers[x].LastSeqNo | x in losers};
      nextentry := losers[nexttrans].LastSeqNo;
```

activate all losers

backward scan

Simple algorithm


```
if StableLog[nextentry].ActionType in {write, compensation}
then
   pageno := StableLog[nextentry].PageNo; fetch (pageno);
   if DatabaseCache[pageno].PageSeqNo >= nextentry.LogSeqNo;
   then
     newlogentry.LogSeqNo := new sequence number;
     newlogentry.ActionType := compensation;
     newlogentry.PreviousSeqNo :=
          ActiveTrans[transid].LastSeqNo;
     newlogentry.RedoInfo :=
          inverse action of the action in nextentry;
     newlogentry.UndoInfo := inverse action of the
          inverse action of the action in nextentry;
     ActiveTrans[transid].LastSeqNo :=
          newlogentry.LogSeqNo;
     LogBuffer += newlogentry;
     read and write (StableLog[nextentry].PageNo)
          according to StableLog[nextentry].UndoInfo;
     DatabaseCache[pageno].PageSeqNo:=newlogentry.LogSeqNo;
   end /*if*/;
   losers[nexttrans].LastSeqNo :=
      StableLog[nextentry].PreviousSeqNo;
end /*if*/;
```

Simple algorithm


```
    if StableLog[nextentry].ActionType = begin
    then
        newlogentry.LogSeqNo := new sequence number;
        newlogentry.ActionType := rollback;
        newlogentry.TransId := StableLog[nextentry].TransId;
        newlogentry.PreviousSeqNo :=
            ActiveTrans[transid].LastSeqNo;
       LogBuffer += newlogentry;
       ActiveTrans -= transid;
       losers -= transid;
    end /*if*/;
end /*while*/;
force ( );
```

Sample Scenario Data Structures

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin (t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
↯SYSTEM CRASH↯				

Restart

Analysis pass: losers = $\{t_2, t_5\}$

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
redo (6)	b: 6			
redo(7)	c: 7			
redo (8)	d: 8			
redo(13)	a: 13			
redo (11)	d: 11			
redo(16)	d: 16			
redo(17)	e: 17			
redo(18)	b: 18			
22: compensate(18)	b: 22		22: compensate(18: b, t_5)	
23: compensate(17)	e: 23		23: compensate(17: e, t_2)	
24: compensate(13)	a: 24		24: compensate(13: a, t_5)	
25: rollback(t_5)			25: rollback(t_5)	
26: compensate(7)	c: 26		26: compensate(7: c, t_2)	
27: rollback(t_2)			27: rollback(t_2)	
force				22, 23, 24, 25, 26, 27
RESTART	COMPLETE: RESUME	NORMAL OPERATION		

Example with system crash during restart see Weikum/Vossen p. 506

Further remarks

- **Theorem 8.19:**

  The simple three-pass redo-history recovery algorithm performs correct recovery.

- Further enhancements and optimizations see Weikum/Vossen pp.510-518.
- Redo-history algorithm preferable
  - because of uniformity, no need for page flush during restart,
  - simplicity and robustness,
  - Inclusion of light-weight checkpoints for log truncation.

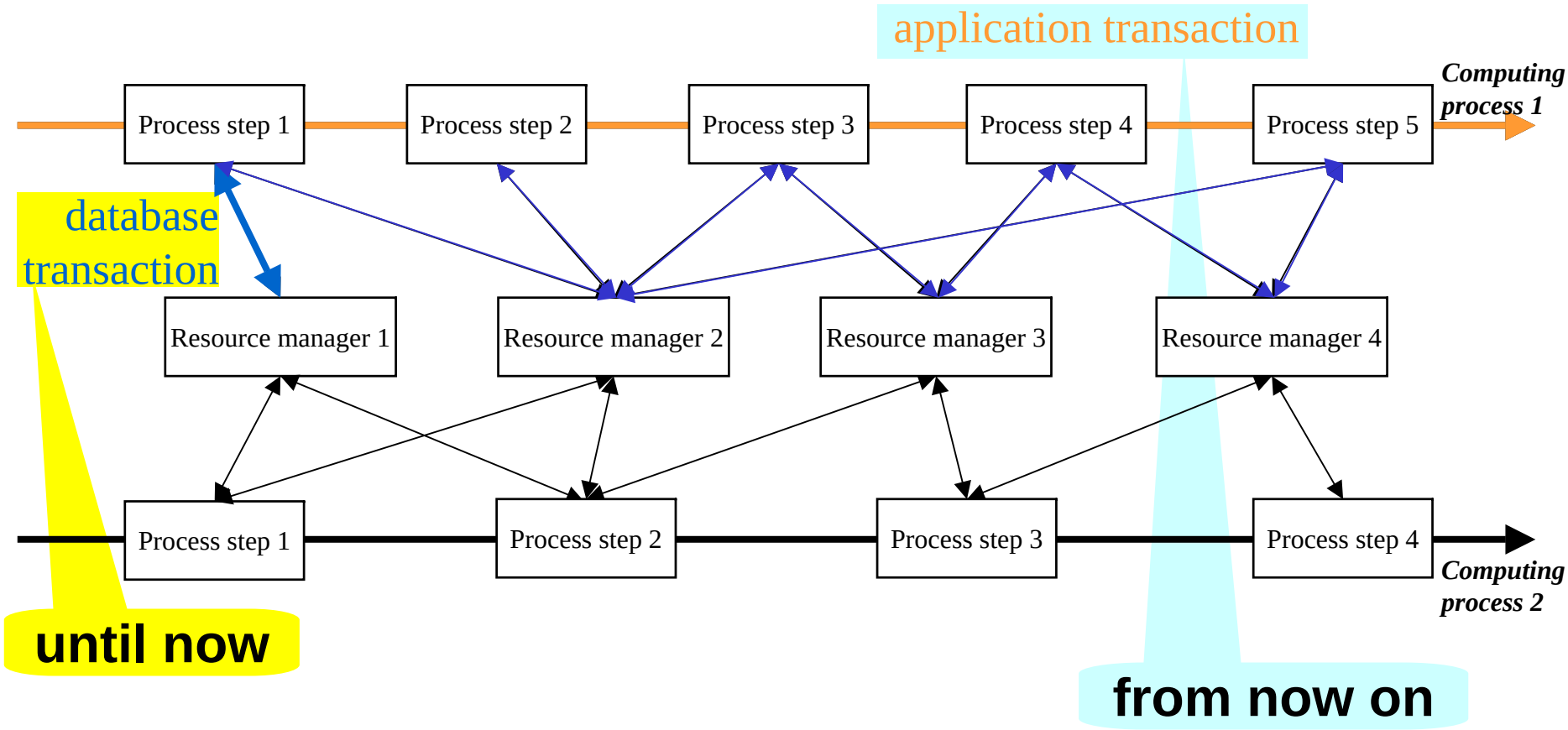
DBS

Universität Karlsruhe (TH)

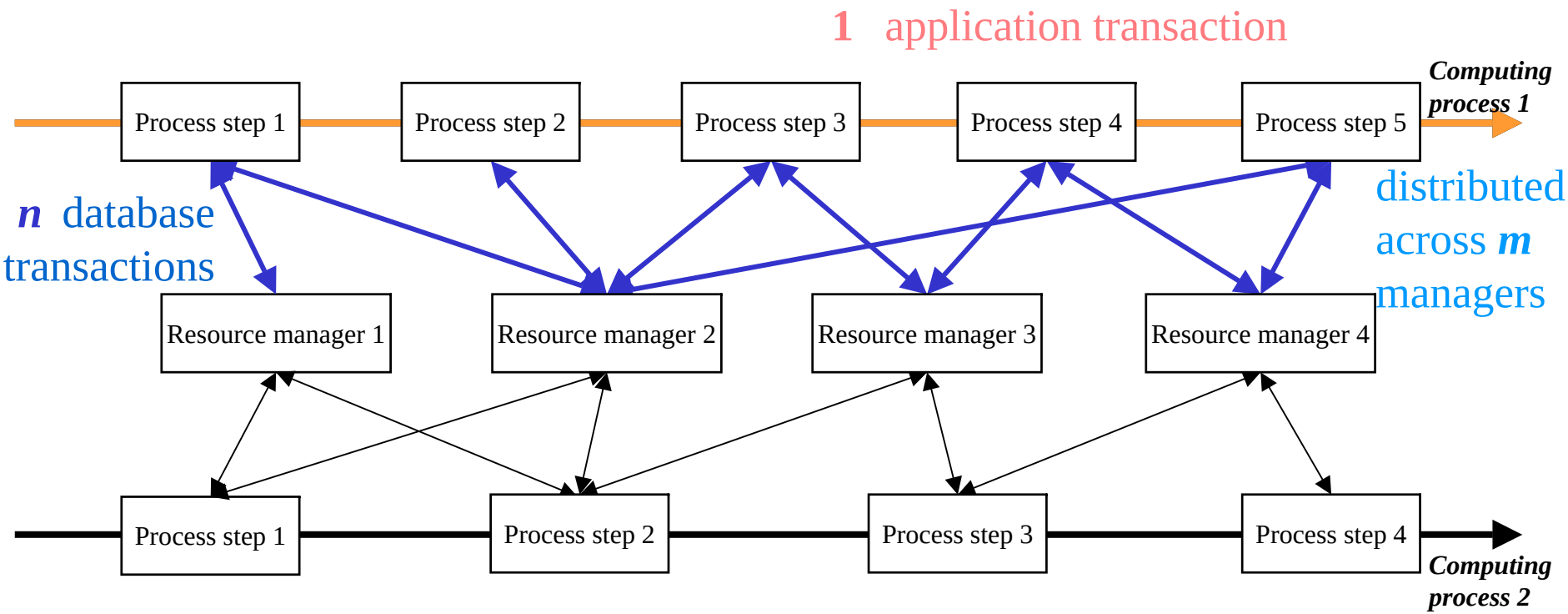
Chapter 9

Distributed Transactions: Characteristics

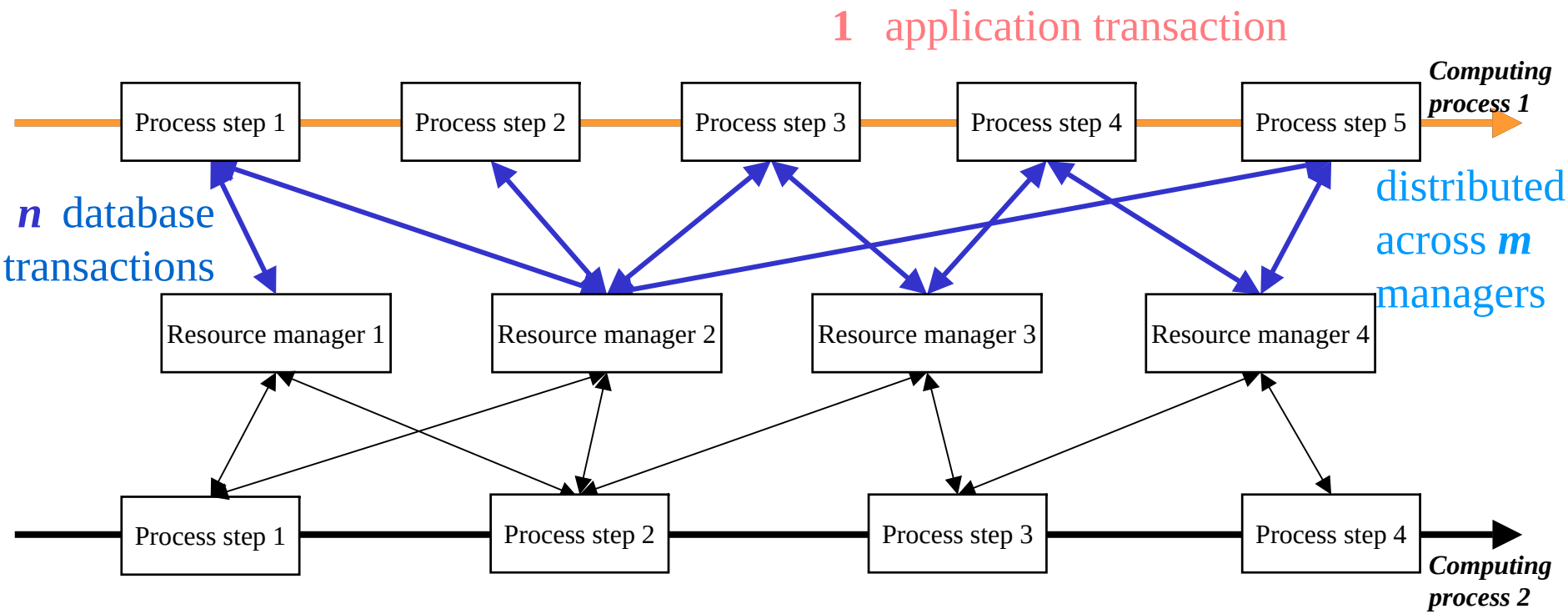
Transactions



Distributed transactions



Complication: Autonomy



Modern complication: All servers – hence, all data servers – are autonomous!

Autonomie

Jedes DBMS ist in *dreierlei* Hinsicht autonom:

- **Entwurfs-Autonomie**: Unabhängigkeit bzgl. verwendetem Datenmodell, logischem und physikalischem Entwurf, Anfragesprache, Protokollen für Nebenläufigkeitskontrolle und Recovery.
- **Ausführungs-Autonomie**: Lokale Transaktionen werden von den DBMS selbständig und ohne Kommunikation bearbeitet. Teile von globalen Transaktionen, die ein DBMS bearbeitet, werden wie lokale Transaktionen behandelt.
- **Kommunikations-Autonomie**: DBMS können mit anderen Systemen kommunizieren, können aber weder zur Kommunikation an sich, noch zu einem Zeitpunkt für die Kommunikation gezwungen werden.

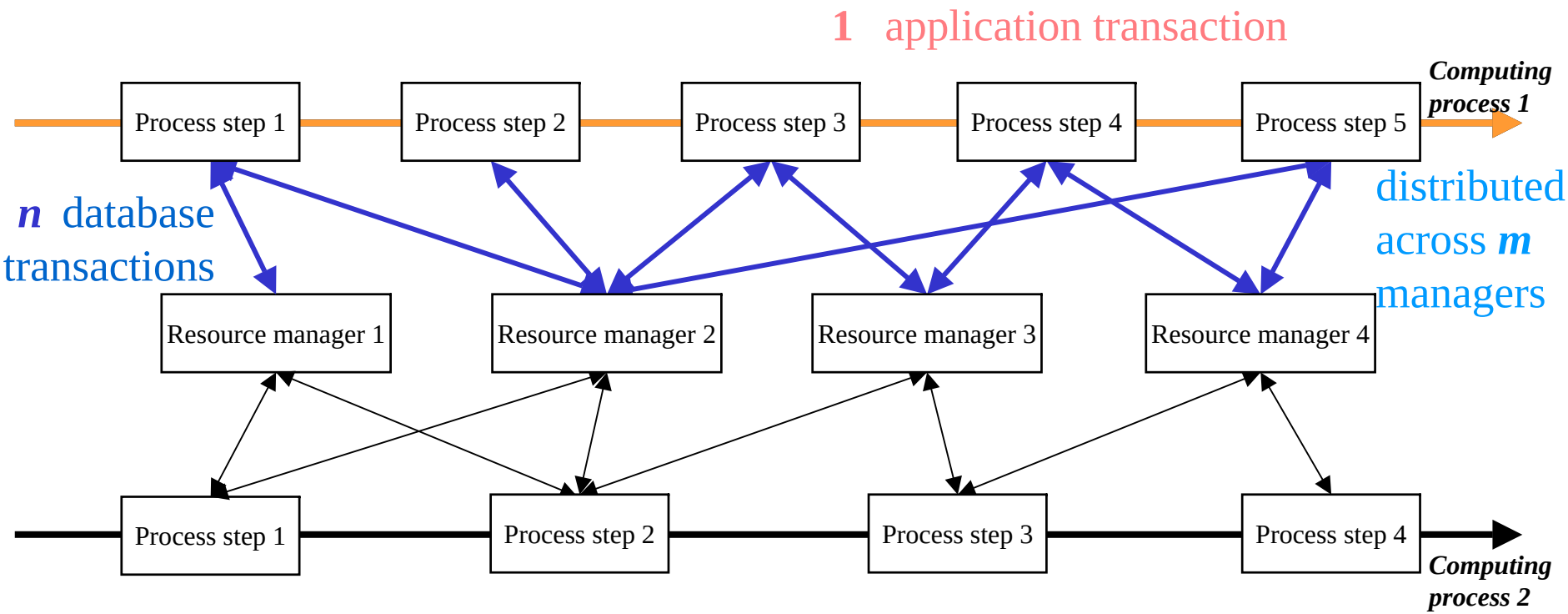
DBMS Federations

Modern focus:

Heterogeneous federation: Participating servers

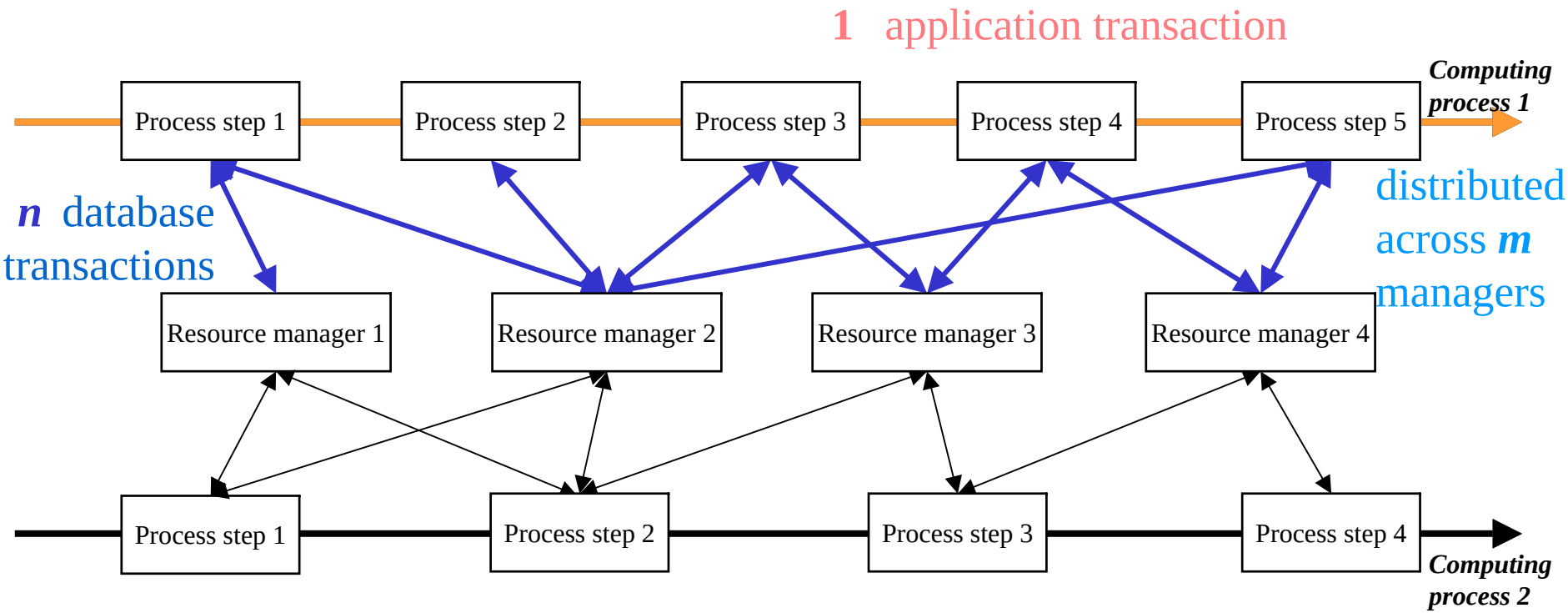
- are **autonomous** and independent;
- have no uniformity of protocols;
- their distribution is **transparent** to users.

Complication: More failure potential



Not only data servers may fail,
but also communication links!

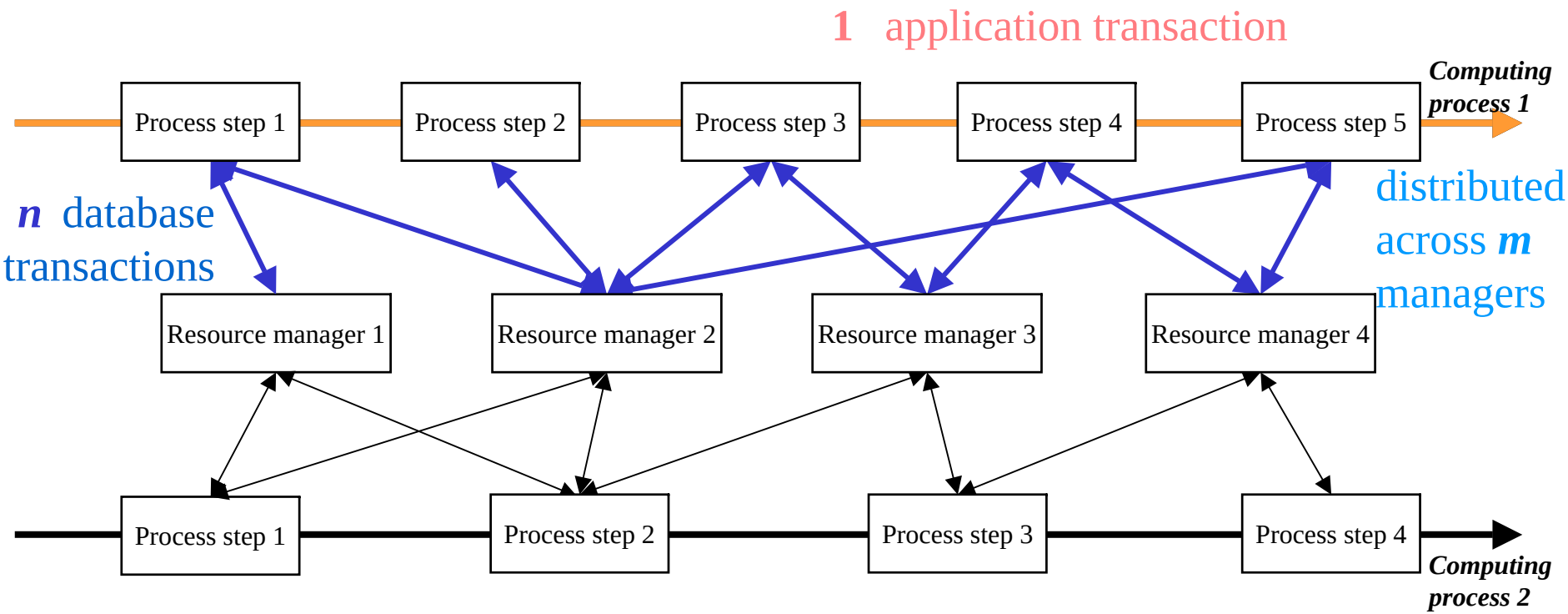
Objective and problems (1)



Ideally: Application transactions are ACID.

Consistency problem 1: Isolation of application transactions ⇒ Can we still enforce serializability?

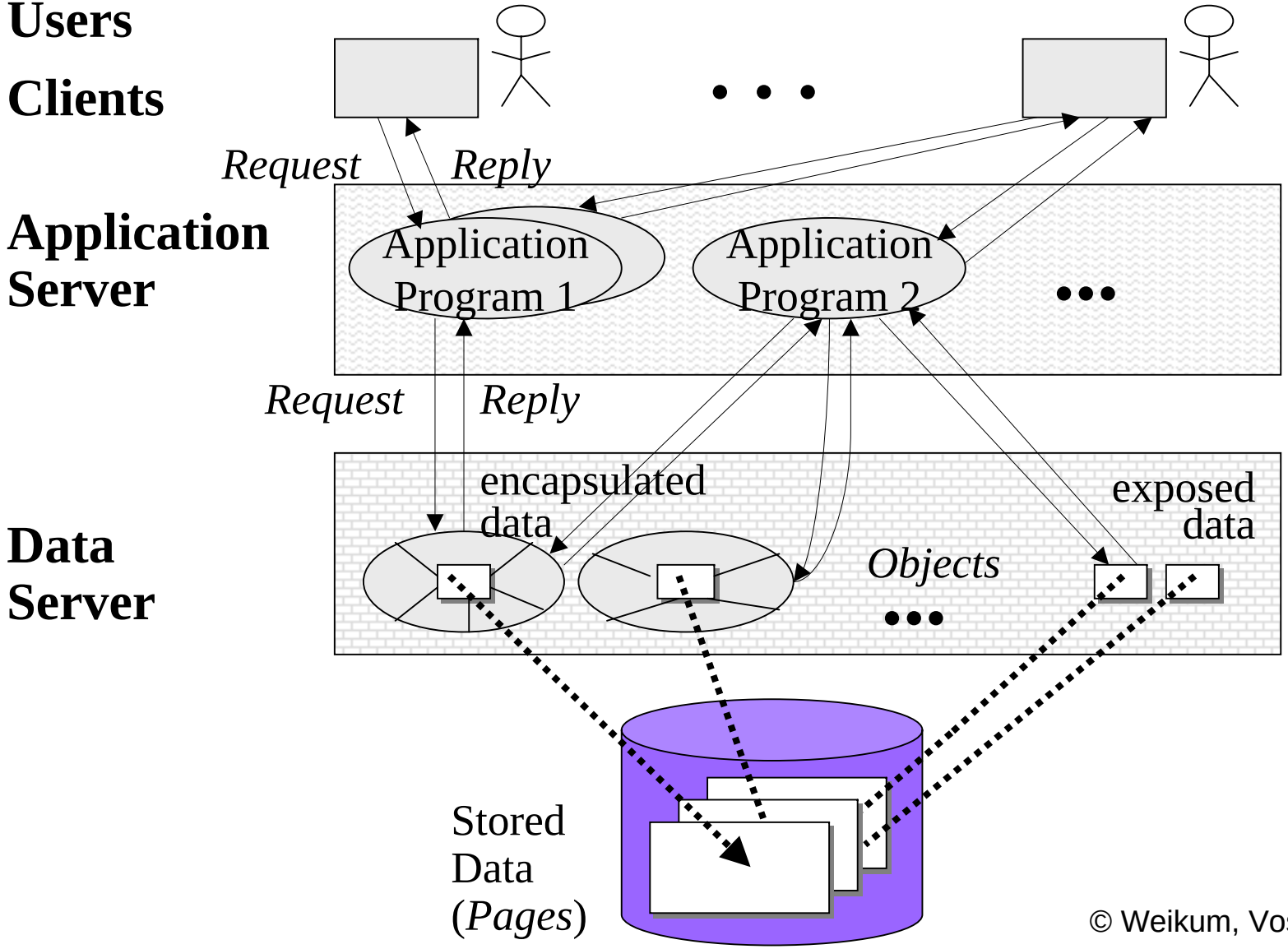
Objective and problems (2)



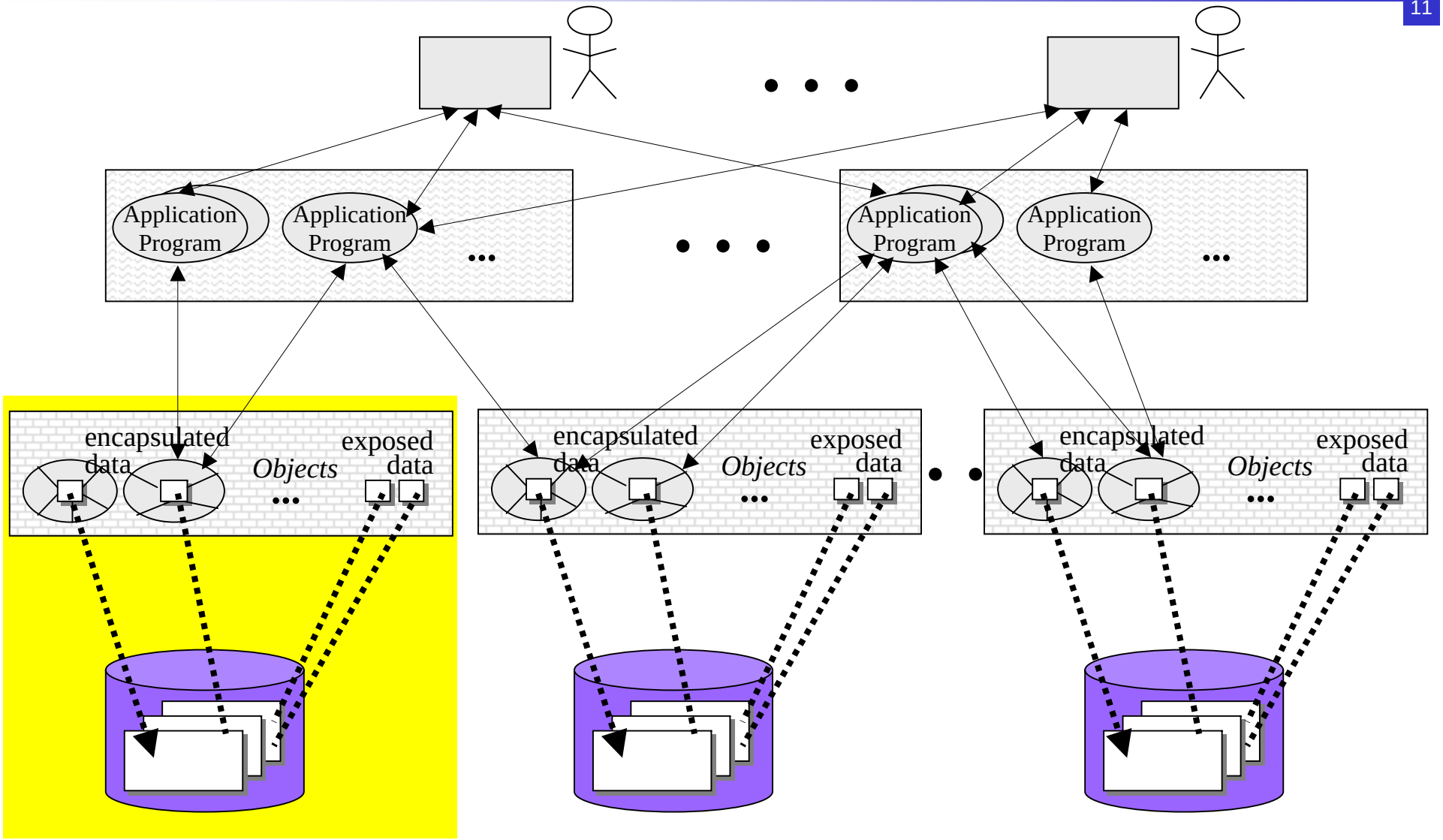
Ideally: Application transactions are ACID.

Consistency problem 2: Atomicity of application transactions ⇒ Can we still enforce all-or-nothing?

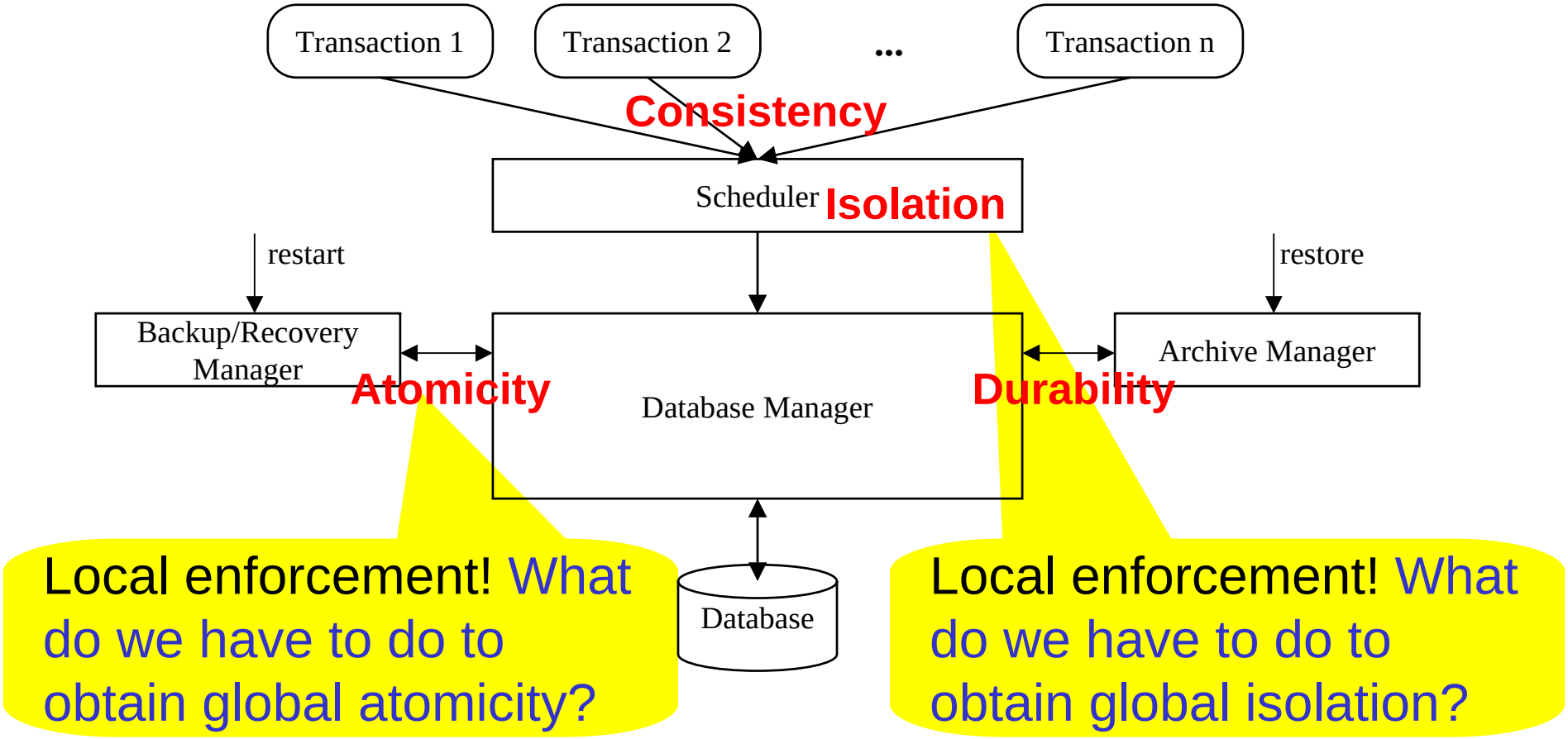
3-Tier Reference Architecture (1)



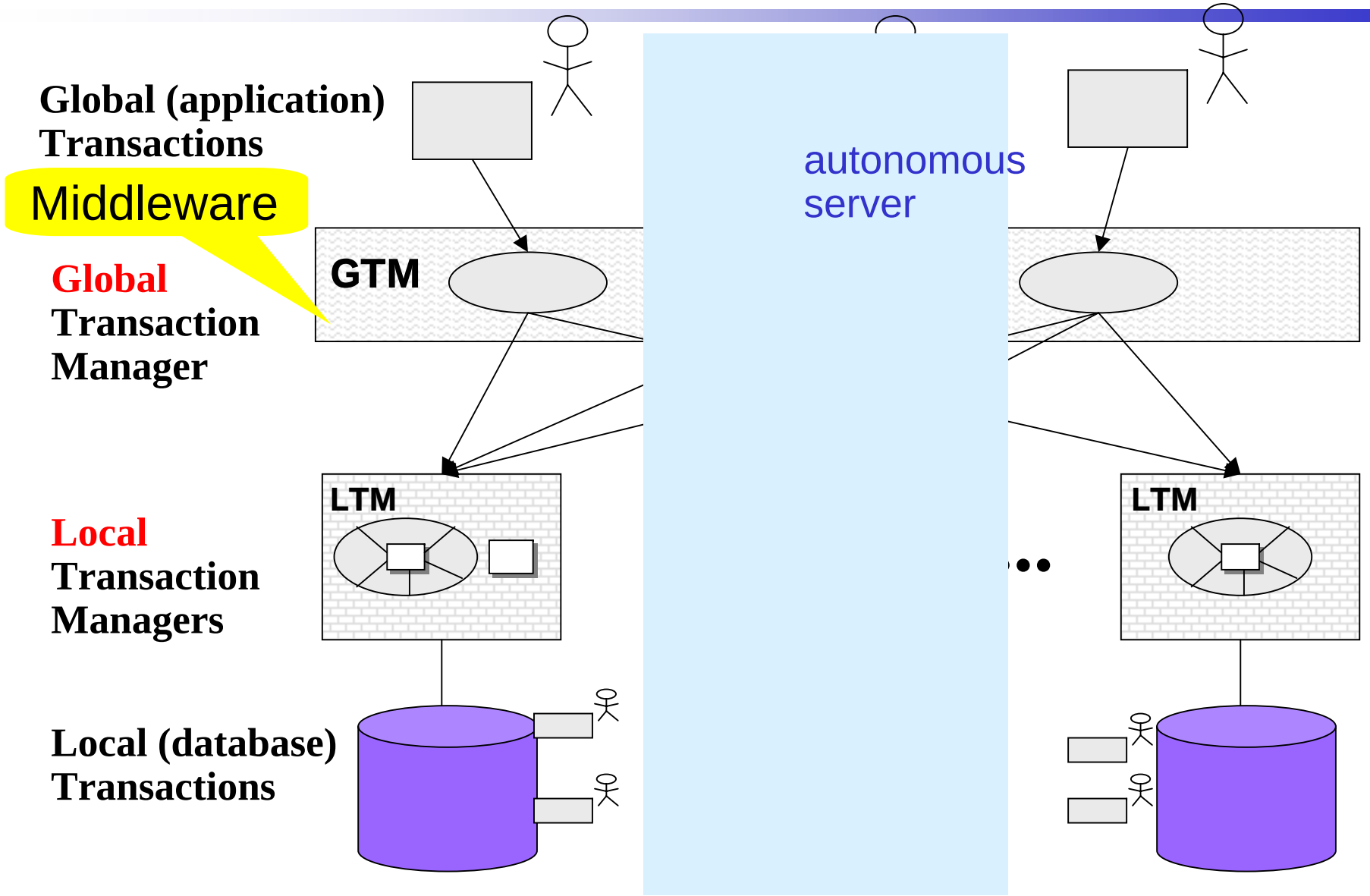
3-Tier Reference Architecture (2)



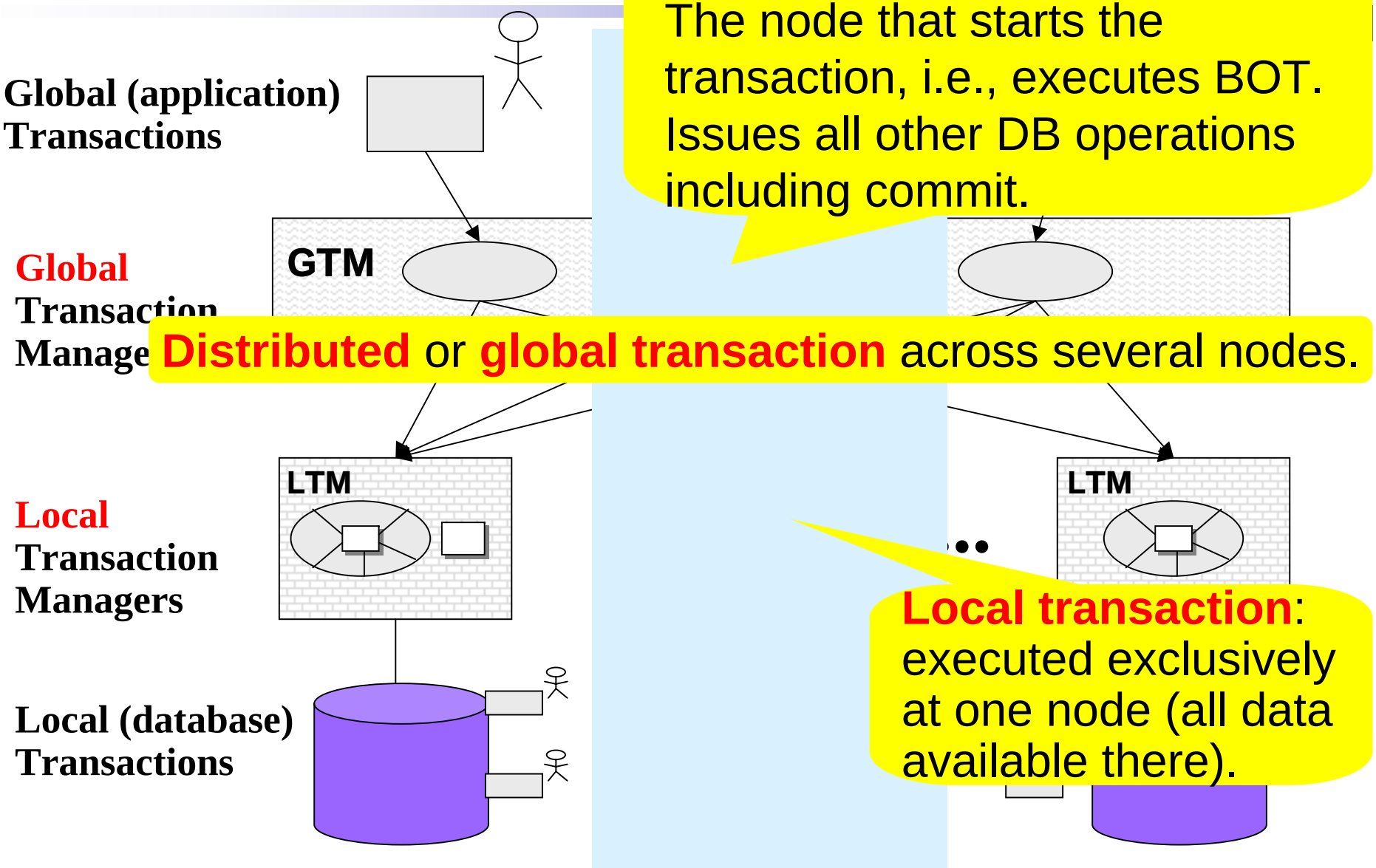
Local server architecture



Federation Model



Distributed Transaction Model (1)



home/coordinator node: The node that starts the transaction, i.e., executes BOT. Issues all other DB operations including commit.

Distributed or **global transaction** across several nodes.

Local transaction: executed exclusively at one node (all data available there).

Distributed Transaction Model (2)

- Local and global transactions are sequential (total order of DB operations).
- Each global transaction starts in each local DBMS at most one local (sub-)transaction.
- Global transactions invoke commit in each local DBMS where they are active.
- Replications of data are not recognized as such.

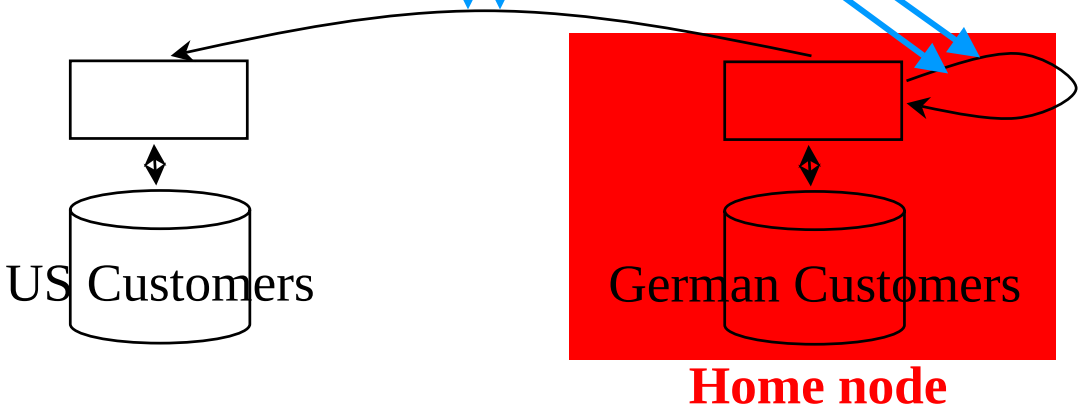
Distributed Transaction Model (3)

Global transaction:

Transfer USD 500,--
from Klemens‘ account **to** Jim‘s account.

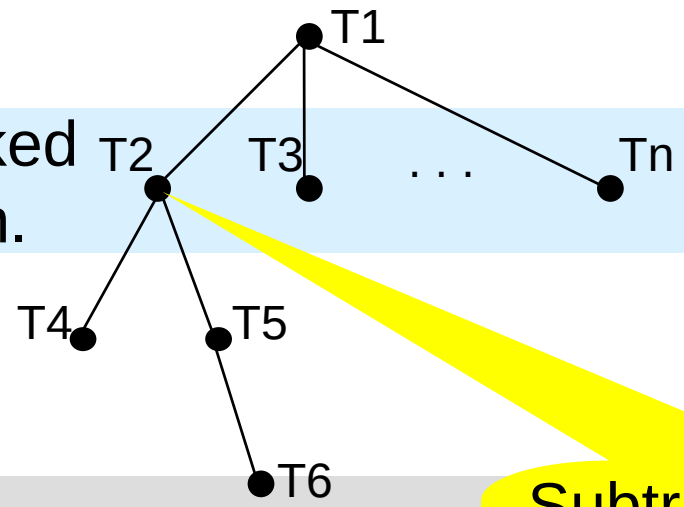
- Increment Jim‘s balance.
- Record receipt of money from Klemens.
- Decrement Klemens‘ balance.
- Record transfer of money to Jim.

(local) subtransactions



Distributed Transaction Model (4)

Primary transaction (root transaction) – subtransaction executed at home node.



Subtransactions invoked by primary transaction.

Subtransaction invokes further subtransactions at other nodes.

Transaction tree

- not balanced,
  height only limited by number of nodes,
- subtransactions may execute in parallel.

Transaction execution

- Transaction starts from the home site, results are returned from there.
- If the LTM at the home site cannot execute an operation, GTM passes them on to the LTM of some other suitable node.
- Subtransaction may concurrently execute at the various nodes.

Einfluss der Autonomie

- Es kann lediglich unterstellt werden, dass jedes lokale System für lokale Isolation und Atomizität sorgt, aber nicht wie.
- Der GTM hat keinen Einfluss auf die Ausführung lokaler Transaktionen. Er kann Kollisionen zwischen globalen und lokalen Transaktionen nicht verhindern.
- Subtransaktionen können in einem lokalen DBMS abstürzen und in anderen nicht.
- Der Abbruch lokaler Transaktionen wird außen nicht bemerkt.

- How to adapt scheduling protocols to maintain serializability?
- How to adapt recovery protocols to maintain atomicity?

DBS

Universität Karlsruhe (TH)

Chapter 10

Distributed Transactions: Synchronization

Conflict serializability

Global History

Assumption: Each global transaction affects each site.

Assumptions:
- Each site holds a partial database.
- These databases are mutually disjoint.

Definition 1(Global History):

Let the heterc eous federation consist of n sites, let T_1 , ... , T_n be sets of loca ransactions at sites 1 , ... n, and let T be a set of global transact ons. Finally, let h_1 , ... , h_n be local histories s.t. $T_i \subseteq trans(h_i)$ and $T \cap trans(h_i) \neq \emptyset$, $1 \leq i \leq n$.

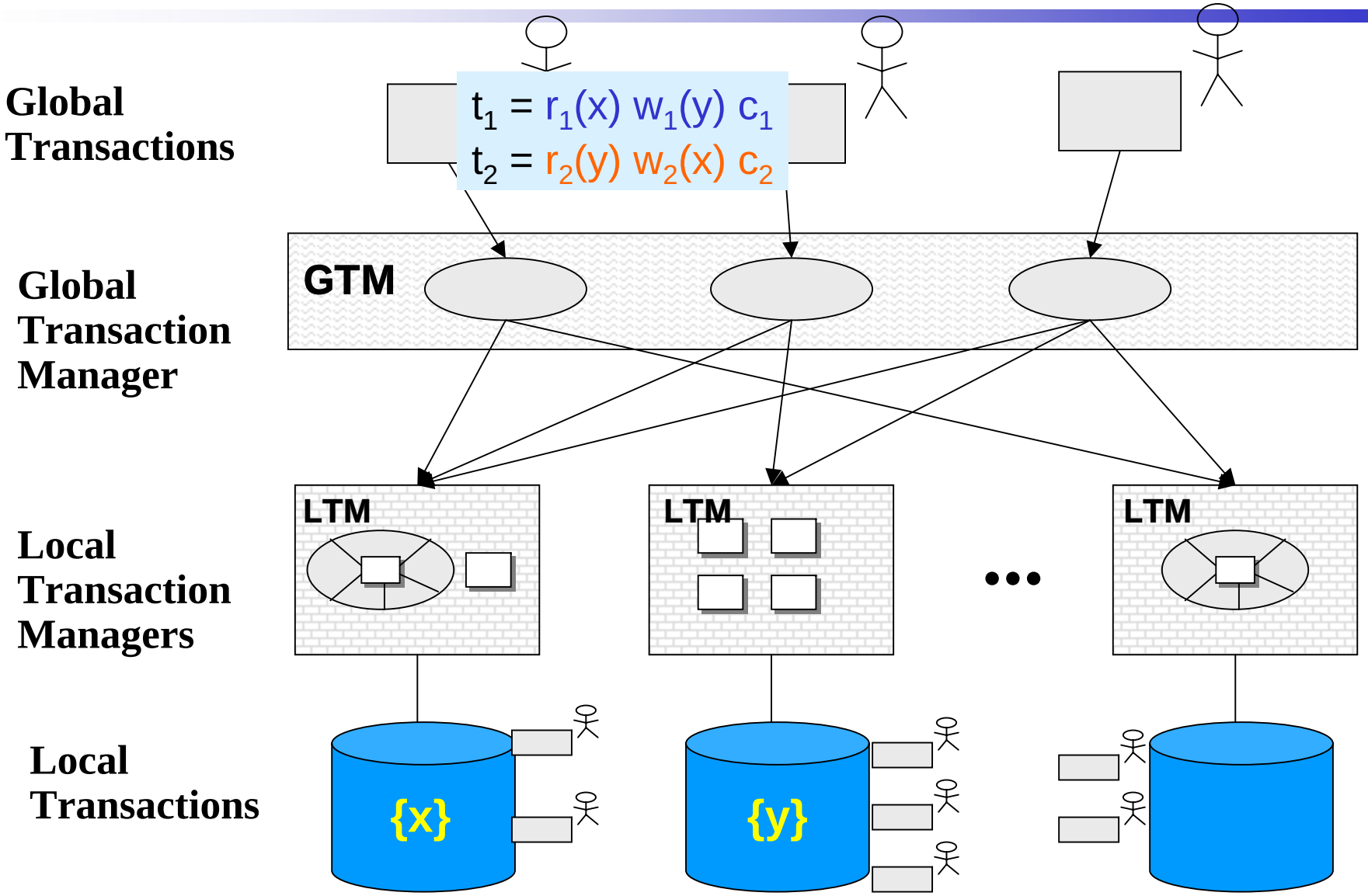
A (heterogeneous) **global history** for h_1 , h$_n$ is a history h for all the T_i and T s.t. its local projection equ he local history at each site, i.e.,

$$\Pi_i(h) = h_i \quad \text{for} \quad 1 \leq i \leq n$$

Projection on the operations in h that are executed at site i.

Local history contains the operations executed at that site.

Global vs. Local Serializability



Global vs. Local Serializability

Input sequence: $r_1(x)\ w_1(y)\ r_2(y)\ w_2(x)\ c_1\ c_2$

But: Delays and overtaking may occur!

h_1:

Server 1:	$r_1(x)$	$w_2(x)\ c_1$	c_2	CSR: $t_1 < t_2$
Server 2:	$w_1(y)$	$c_1\ r_2(y)\ c_2$		CSR: $t_1 < t_2$

Global history: $h_1 = r_1(x)\ w_1(y)\ w_2(x)\ c_1\ r_2(y)\ c_2$ CSR: $t_1 < t_2$

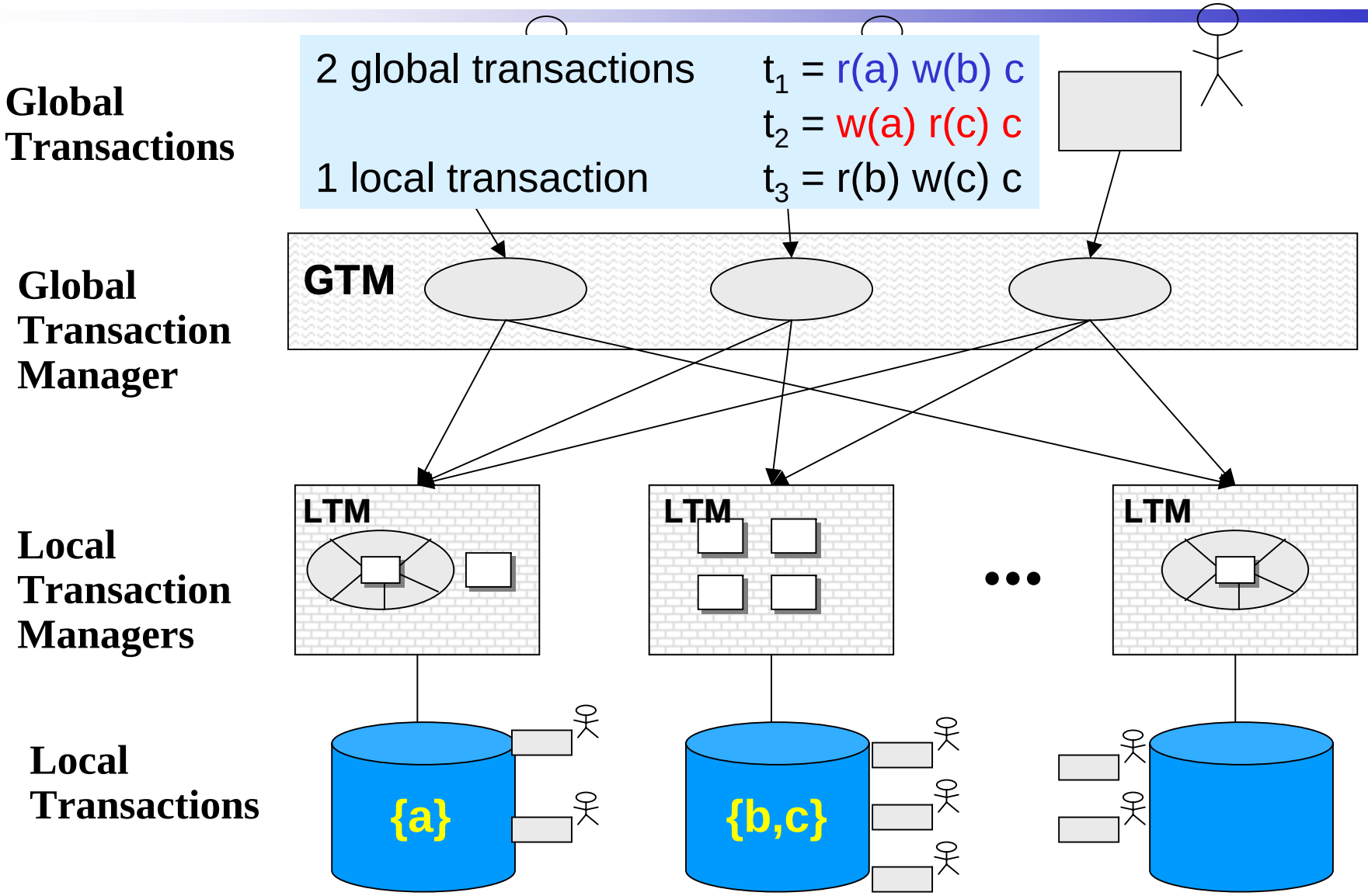
h_2:

Server 1:	$r_1(x)$	$w_2(x)\ c_1$	c_2	CSR: $t_1 < t_2$
Server 2:	$r_2(y)$	$c_2\ w_1(y)\ c_1$		CSR: $t_2 < t_1$

Global history: $h_2 = r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_1\ c_2$ CSR: $t_1 < t_2 < t_1$

Local CSR on all involved servers does not imply global CSR

Global vs. Local Serializability



Global vs. Local Serializability

The GTM decides to execute global transactions t_1 and t_2 serially.
CSR: $t_1 < t_2$

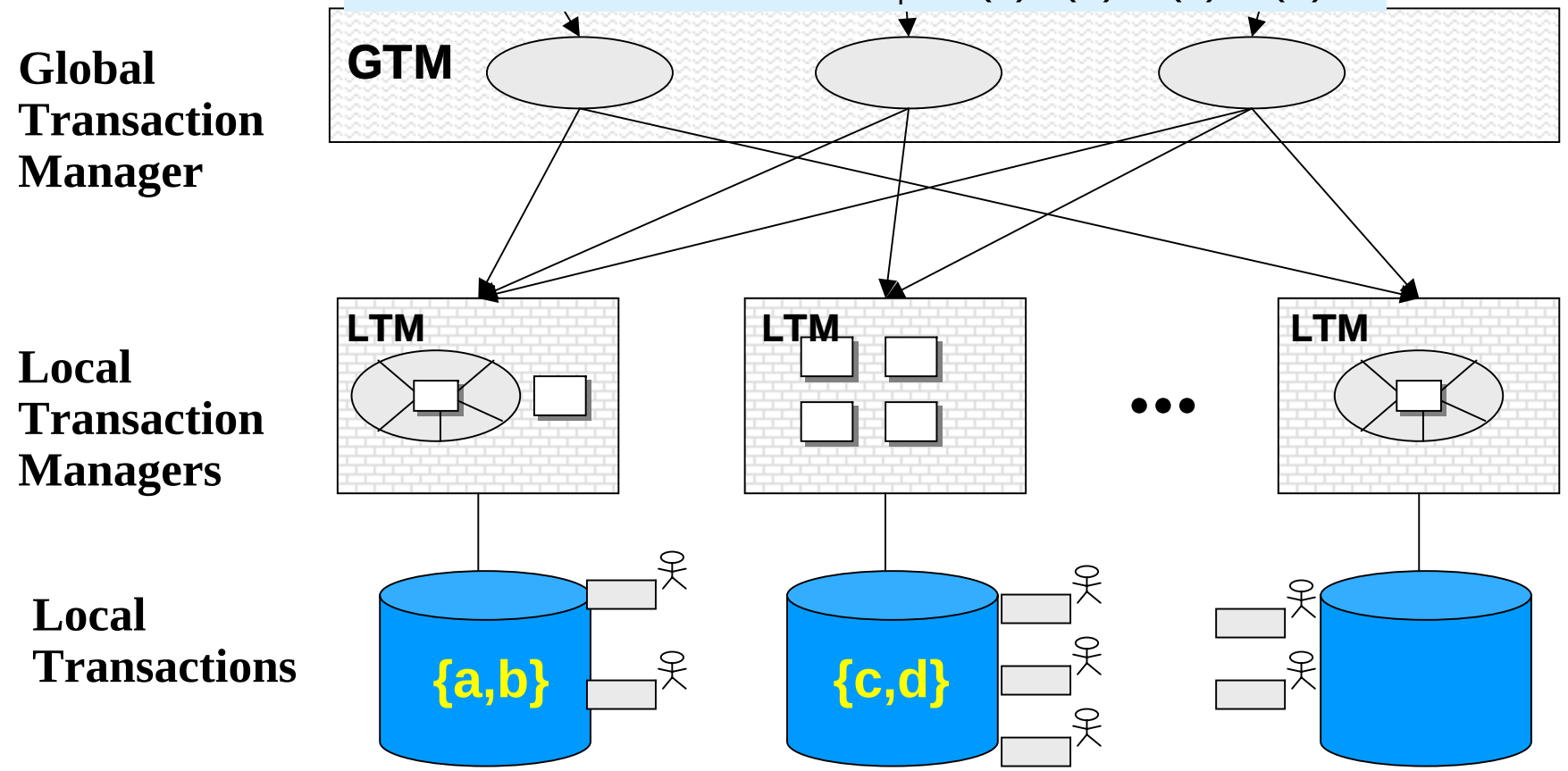
Server 1:	$r_1(a)\ c_1$	$w_2(a)\ c_2$	CSR: $t_1 < t_2$
Server 2:	$r_3(b)\ w_1(b)\ c_1$	$r_2(c)\ c_2\ w_3(c)\ c_3$	CSR: $t_2 < t_3 < t_1$

The global schedule is not CSR.
Problem: Local transactions may introduce further conflicts between global transactions.

Global vs. Local Serializability

2 global transactions
t_1 = r(a) r(d) c
t_2 = r(c) r(b) c
read-only, independent

2 local transactions
t_3 = r(a) r(b) w(a) w(b) c
t_4 = r(c) r(d) w(c) w(d) c



Global vs. Local Serializability

Server 1: $r_1(a)$ $r_3(a)$ $r_3(b)$ $w_3(a)$ $w_3(b)$ $r_2(b)$ CSR: $t_1 < t_3 < t_2$

Server 2: $r_2(c)$ $r_4(c)$ $r_4(d)$ $w_4(c)$ $w_4(d)$ $r_1(d)$ CSR: $t_2 < t_4 < t_1$

Problem (as before):

Local transactions may introduce (further) conflicts between global transactions

even if the global transactions cannot be in conflict.

Indirect Conflicts

Definition 10.2 (Direct and Indirect Conflicts):

Let h_i be a local history, and let t, t‘ $\in$ trans(h_i), t $\neq$ t‘.

(i) t and t‘ are in **direct** conflict in h_i if there are two data operations p $\in$ t and q $\in$ t‘ in h_i that access the same data item and at least one of them is a write.

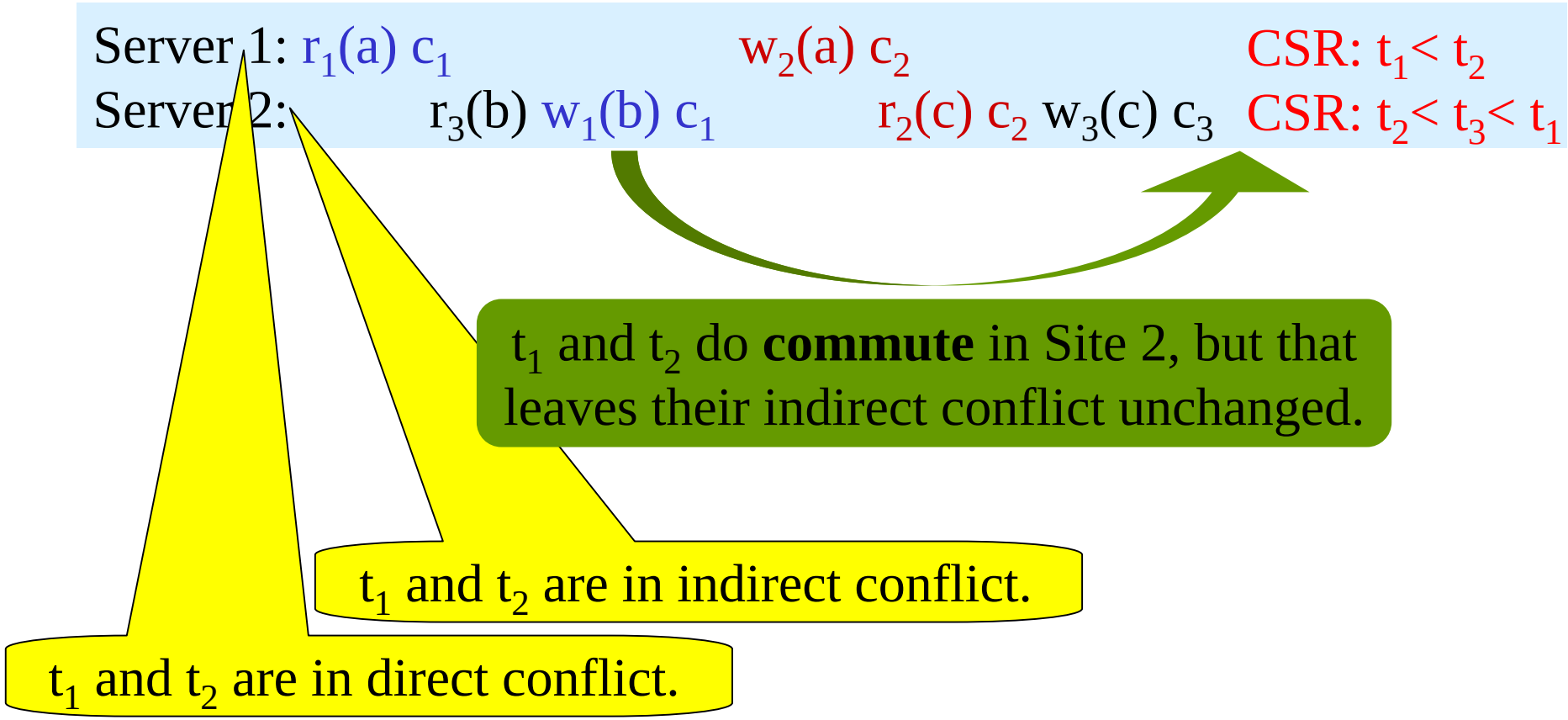
(ii) t and t‘ are in **indirect** conflict in h_i if there exists a sequence t_1 , ... , t_r of transactions in trans(h_i) s.t. t is in h_i in direct conflict with t_1, t_j is in h_i in direct conflict with t_{j+1}, $1 \leq j \leq r-1$, and t_r is in h_i in direct conflict with t‘.

(iii) t and t‘ are in **conflict** in h_i if they are in direct or indirect conflict.

Note: “Conflict” from now on means “direct or indirect conflict.”

Global vs. Local Serializability

Previous example: The GTM decides to execute global transactions t_1 and t_2 serially. CSR: $t_1 < t_2$



Global Conflict Graph

Definition 10.3 (Global Conflict Graph):
Let h be a global history for local histories h_1 , ... , h_n; let $G(h_i)$ denote the conflict graph of h_i, $1 \leq i \leq n$.

The **global conflict graph** of h is the graph union of all $G(h_i)$.

Theorem 10.4:
Let h_1 , ... , h_n be local histories s.t. each $G(h_i)$ is acyclic. Let h be a global history for the h_i.

Then h is **globally conflict serializable** iff G(h) is acyclic.

Global Conflict Serializability

Theorem 10.5:
Let h be a global history with local histories h_1 , ... , h_n involving a set T of transactions s.t. each h_i is conflict serializable.
Then h is globally conflict serializable iff there exists a total order "<" on T that is consistent with each local serialization order of the transactions.

Thus, the crucial point in all protocols is to make sure that such a total ordering among the transactions can be established.

Distributed 2-Phase Locking

Distributed 2PL

Perform standard 2PL locally at each site:

For each step the scheduler **requests a lock** on behalf of the step‘s transaction.
Each lock is requested in a specific **mode (read or write)**.
If the data item is not yet locked in an **incompatible mode** the lock is granted;
otherwise there is a **lock conflict** and the transaction becomes **blocked** (suffers a **lock wait**) until the current lock holder **releases the lock**.

Also recall:

A locking protocol is **two-phase (2PL)** if for every output schedule s and every transaction $t_i \in$ trans(s) no ql_i step follows the first ou_i step (q, o $\in$ {r, w}).

Consistent Global Ordering

All subtransactions of a given transaction must have identical equivalence times. ⇒ **Enforce global equivalence time**

- Local equivalence time can be controlled from outside:
  - Local equivalence time = commit time ⇒ all global transactions keep their locks up to the local end ⇒ all local histories must be RG.
- GTM must ensure that all local commits of a global transaction take place simultaneously.
  - Requires commit synchronization.

Synchronization

Definition 10.6 (commit-deferred)

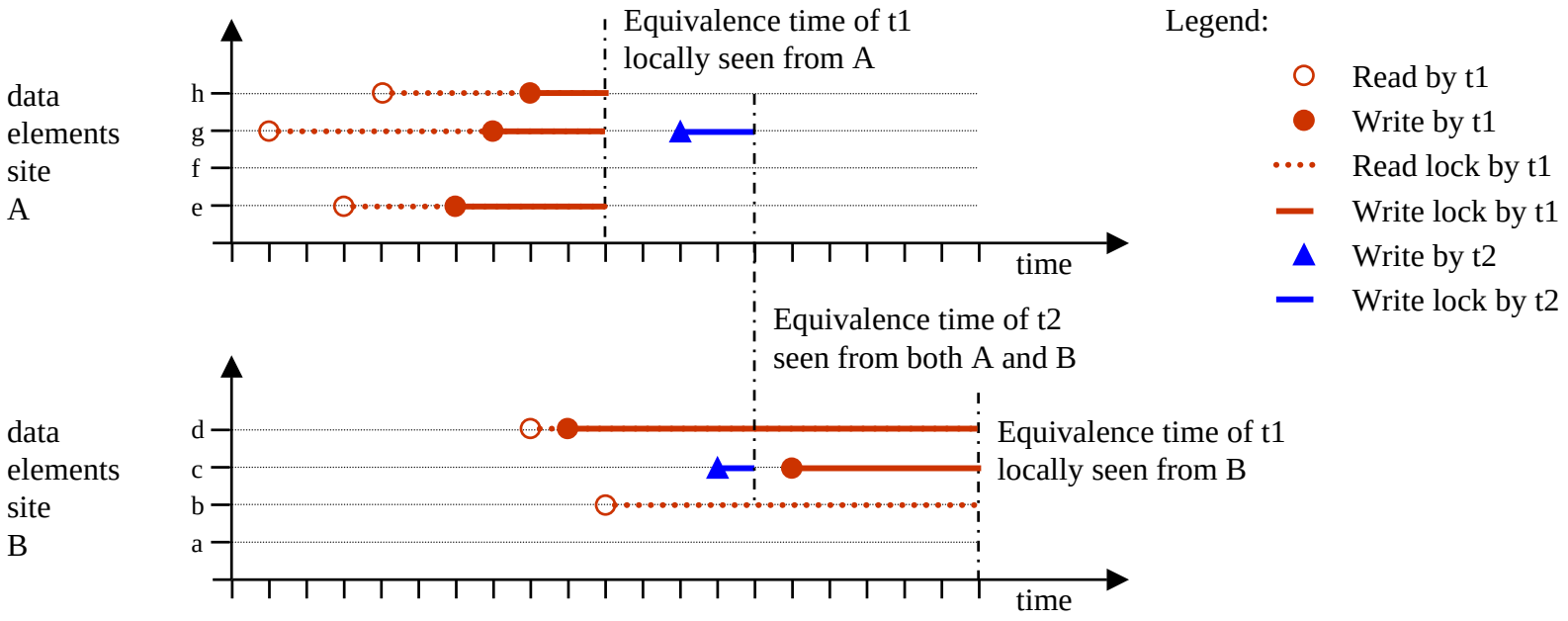
A global transaction *t* is **commit-deferred** if its commit is only communicated by GTM to all LTM after the local completion of all data operations of *t* was acknowledged.

Theorem 10.7

Let *h* be a global history for $h_1,\ldots,h_n$. If $h_i \in RG$, $1 \leq i \leq n$, and all global transactions are commit-deferred, then *h* is **global conflict serializable**.

Need for commit-deferred

- Consider transactions t1 und t2 that locally follow 2PL, but are not globally serializable:



Global vs. Local Serializability

Previous example: The GTM decides to execute global transactions t_1 and t_2 serially. CSR: $t_1 < t_2$

Server 1: $r_1(a)$ c_1 $w_2(a)$ c_2 CSR: $t_1 < t_2$
Server 2: $r_3(b)$ $w_1(b)$ c_1 $r_2(c)$ c_2 $w_3(c)$ c_3 CSR: $t_2 < t_3 < t_1$

RG

CD

Server 1: $r_1(a)$ c_1 $w_2(a)$ c_2
Server 2: $r_3(b)$ $r_2(c)$ c_2 $w_3(c)$ c_3 $w_1(b)$ c_1

serial

CD

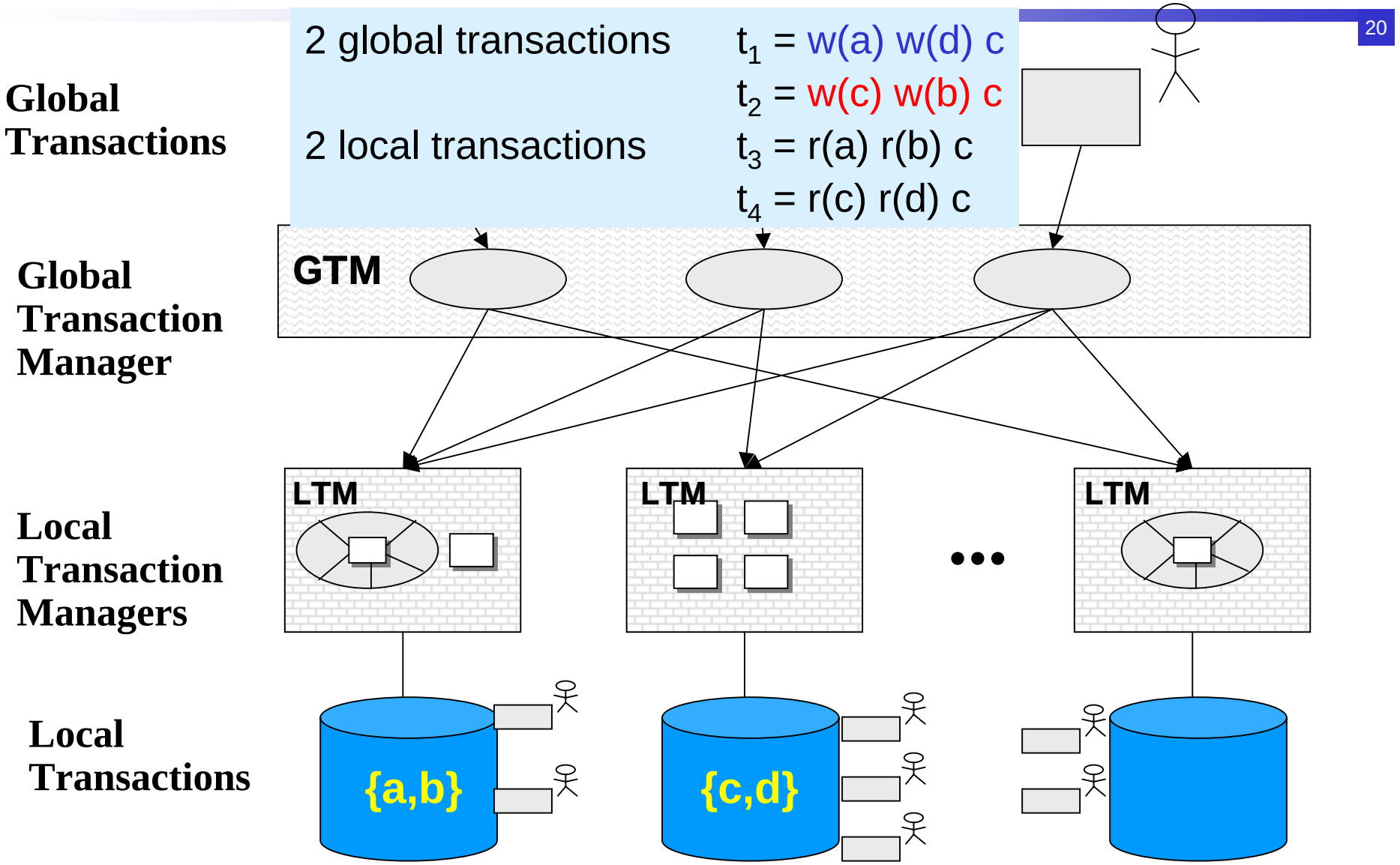
Server 1: $r_1(a)$ c_1 $w_2(a)$ c_2
Server 2: $r_3(b)$ $w_3(c)$ c_3 $w_1(b)$ c_1 $r_2(c)$ c_2

Server 1: $r_1(a)$ c_1 $w_2(a)$ c_2 CSR: $t_1 < t_2$
Server 2: $r_3(b)$ $w_3(c)$ c_3 $w_1(b)$ c_1 $r_2(c)$ c_2 CSR: $t_3 < t_1 < t_2$

Need for RG (1)

2 global transactions
t_1 = w(a) w(d) c
t_2 = w(c) w(b) c

2 local transactions
t_3 = r(a) r(b) c
t_4 = r(c) r(d) c



Need for RG (2)

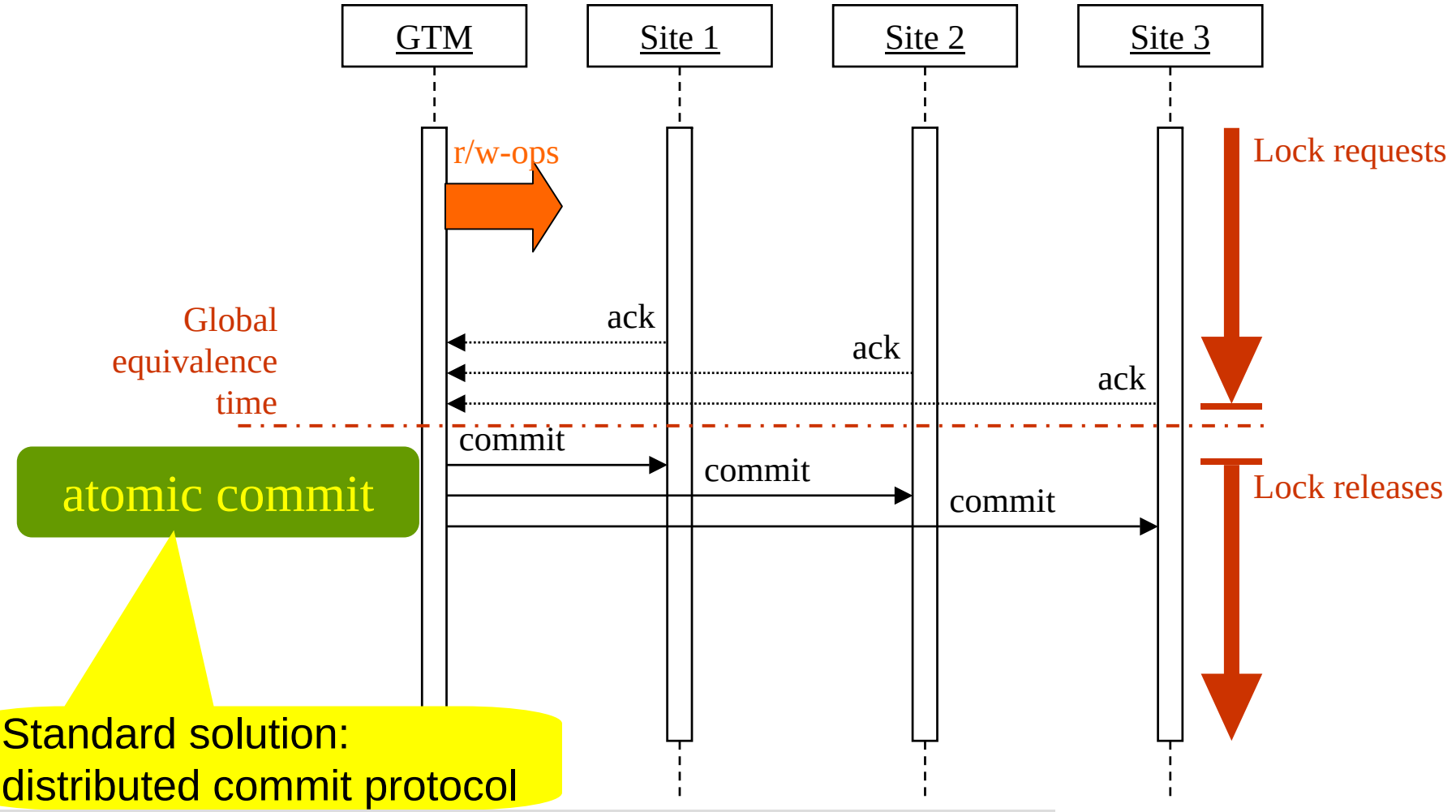
The GTM decides to execute global transactions t_1 and t_2 in the order $w_1(a)\ w_1(d)\ w_2(c)\ w_2(b)\ c_1\ c_2$. CSR: $t_1 < t_2$

But there may be different communication delays:
Server 1: $h_1 = w_1(a)\ r_3(a)\ r_3(b)\ c_3\ w_2(b)$
Server 2: $h_2 = w_2(c)\ r_4(c)\ r_4(d)\ c_4\ w_1(d)$

GTM issues $c_1\ c_2$:
Server 1: $h_1 = w_1(a)\ r_3(a)\ r_3(b)\ c_3\ w_2(b)\ c_1\ c_2$ CSR: $t_1 < t_3 < t_2$
Server 2: $h_2 = w_2(c)\ r_4(c)\ r_4(d)\ c_4\ w_1(d)\ c_1\ c_2$ CSR: $t_2 < t_4 < t_1$

Permitted under (simple) 2PL.
Under RG: $w_2(c)\ w_1(d)\ c_1\ c_2\ r_4(c)\ r_4(d)\ c_4$ CSR: $t_2 < t_1 < t_4$ or $t_1 < t_2 < t_4$
and similarly for Server 1.

Rigorous / commit-deferred



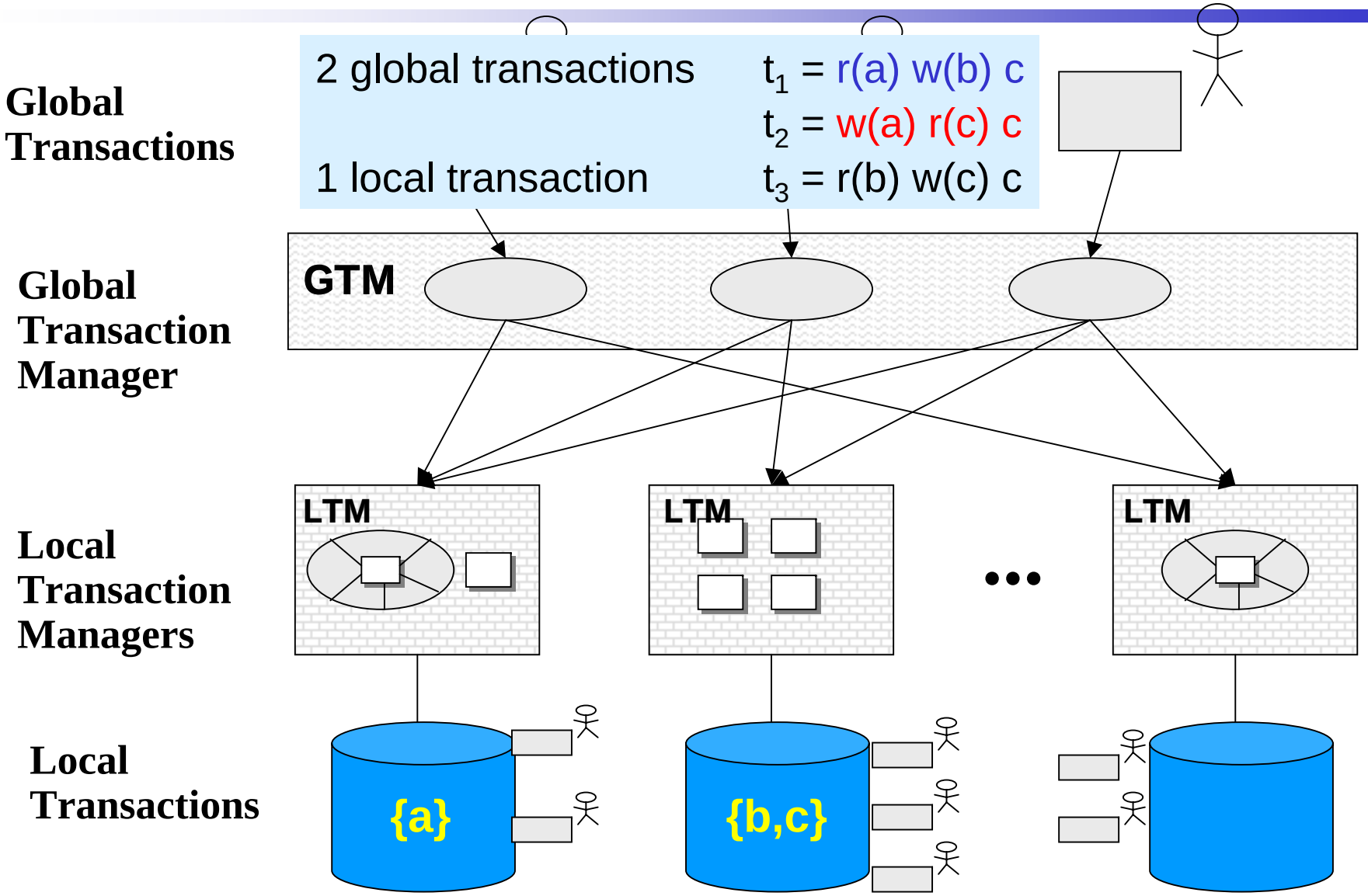
Solves distributed recovery problem as well, see there!

Light-weight solution: Tickets (1)

- Idea: Avoid different ordering of serially submitted global transactions by forcing direct conflicts no matter whether there are indirect conflicts.
- Only minimal LTM requirements: CSR.
- Additional data object in each database („ticket“).
- Each global transaction
  - reads ticket,
  - writes back incremented value.

„Take-a-ticket“ operation.

Light-weight solution: Tickets (2)



Light-weight solution: Tickets (3)

- Assume t_1 t_2.
- Possible submissions:
  - $h_1 = r_1(a)\ c_1\ w_2(a)\ c_2$ CSR: $t_1 < t_2$
  - $h_2 = r_3(b)\ w_1(b)\ c_1\ r_2(c)\ c_2\ w_3(c)\ c_3$ CSR: $t_2 < t_3 < t_1$
- 2PL without tickets:
  - $h_1 = r_1(a)\ c_1\ w_2(a)\ c_2$ CSR: $t_1 < t_2$
  - $h_2 = r_3(b)\ r_2(c)\ c_2\ w_3(c)\ c_3\ w_1(b)\ c_1$ CSR: $t_2 < t_3 < t_1$
- Use of tickets:
  - $h_1 = r_1(tc_1)\ w_1(tc_1+1)\ r_1(a)\ c_1\ r_2(tc_1)\ w_2(tc_1+1)\ w_2(a)\ c_2$
  - $h_2 = r_3(b)\ r_1(tc_2)\ w_1(tc_2+1)\ w_3(c)\ c_3\ w_1(b)\ c_1\ r_2(tc_2)\ w_2(tc_2+1)\ r_2(c)\ c_2$ CSR: $t_3 < t_1 < t_2$

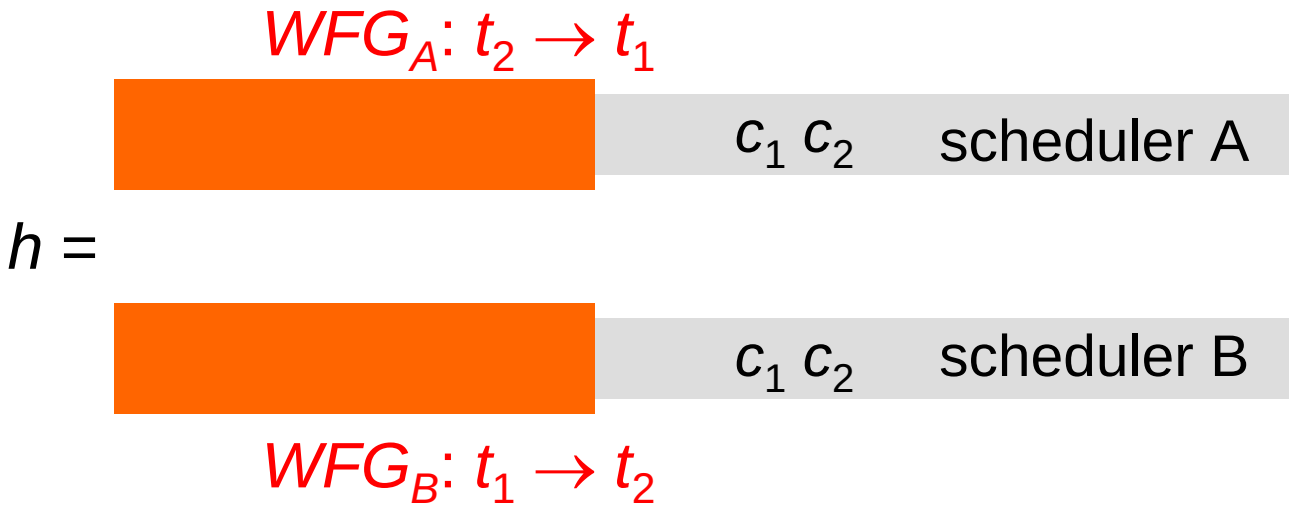
Distributed Deadlocks

How they occur

- Since nodes are autonomous, each local scheduler S_i maintains its own local waits-for graph WFG_i.
- Local WFGs do not suffice!
- Example:

$t_1 = r_1(x)\ w_1(y)\ c_1$

$t_2 = r_2(y)\ w_2(x)\ c_2$



Centralized deadlock detection

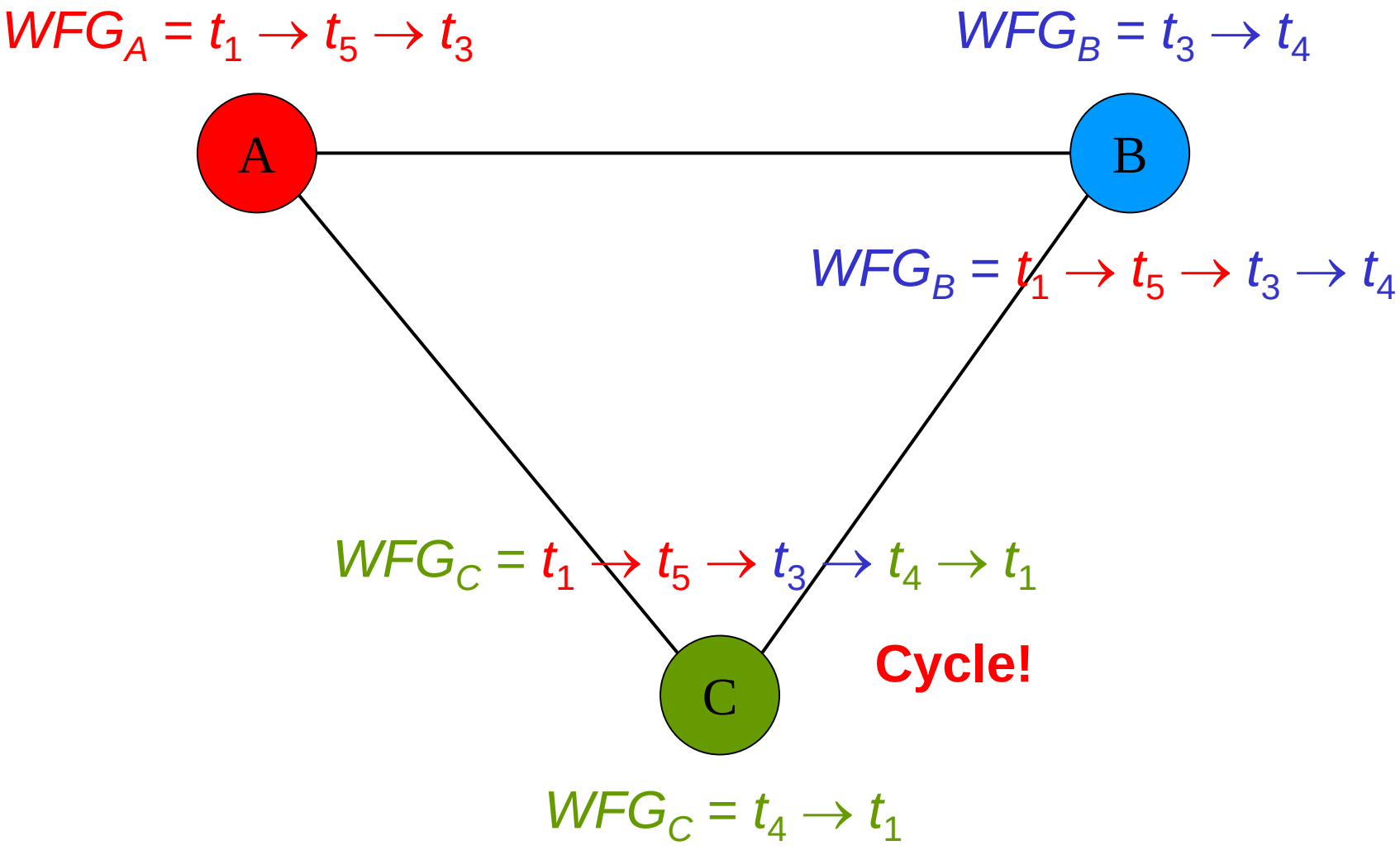
Centralized detection of global deadlocks:

- A singular site maintains the global WFG as the union of all local WFG_i. Schedulers S_i send (parts of) their WFG_i to the central site.
- Clearly: Deadlocks are guaranteed to be detected.
- Price:
  - communication overhead,
  - central site as a single point of failure, performance bottleneck
  - delayed detection due to only periodic transmission of the WFG_i,
  - false positives (phantom deadlocks) due to delayed communication of aborts.

Distributed deadlock detection (1)

- If cycle is detected in local WFG, abort one TA in the cycle $\Rightarrow$ global abort necessary (see “Recovery”).
- Otherwise **Path Pushing**:
  - Suppose WFG_i includes path $t_i \rightarrow \ldots \rightarrow t_j$.
  - Send the path to each node suspected of blocking t_j.
  - Each node adds the path to its own WFG.
  - If no deadlock but a new suspicious path, push the new path.
  - Each deadlock will ultimately be detected.

Distributed deadlock detection (2)



Distributed Timestamp Ordering

Distributed TO

- Local TO as known before.
- Each scheduler assigns to each transaction arriving at its site a time stamp.
  - Global effect requires that time stamps are globally unique.
    $\Rightarrow$ Only then can each scheduler apply the TO protocol locally and independently. $\Rightarrow$ Distribution does not raise new problems. $\Rightarrow$ No communication needed among the schedulers.
- Distributed TO is particularly simple and requires little overhead. $\Rightarrow$ We pay the usual price of a small subset of *CSR*.

Uniqueness of time stamps (1)

Required: Method for assigning globally unique time stamps without the need for a central authority or for communication among the sites.

Solution:

- Within a node achieve uniqueness by simply running a counter and assigning its value to the next (local) time stamp (*relative time*).
- Compute the global time stamp of transaction *t* from the relative time z_t, the total number *k* of nodes and the unique node ID (e.g., a number *i*) of the affected site K_i:

$$ts(t) = z_t * k + i$$

Uniqueness of time stamps (2)

Example:

t_1 starts at K_1 and t_2 at K_2, and let $z_{t_1} = z_{t_2} = 0$ (i.e., t_1 and t_2 are the first transactions executed at each node). Then

$$ts(t_1) = 0 * 2 + 1 = 1$$

$$ts(t_2) = 0 * 2 + 2 = 2$$

Remarks:

- In general, order of time stamps does not correspond to order of arrival.

  $\Rightarrow$ Method is in general not fair.

  - Strict fairness only if nodes communicate.

Weaker serializability

Resume

- Rigorous/commit-deferred:
  - Minus: Harmful to autonomy because enforcement of *RG* and potential waits for global commit.
  - Plus: Natural combination with recovery.
  - But: Local transactions participate in the Minus without profiting from the Plus!
- Tickets and Distributed TO:
  - Plus: Autonomy practically preserved.
  - Minus: Separate solution for recovery needed, small *CSR* subset.

Relax serializability?

Quasi-Serialisierbarkeit (1)

Definition 10.8

Eine Menge $\{h_1,\ldots,h_n\}$ lokaler Historien heißt **quasi-seriell**, falls gilt

- $h_k \in CSR$, $1 \leq k \leq n$, und
- es gibt eine Totalordnung $<$ auf der Menge T globaler Transaktionen, so dass $t_i < t_j$ für $t_i,t_j \in T$, $i \neq j$ impliziert, dass in jedem h_k, $1 \leq k \leq n$, die t_i-Subtransaktion *vollständig* vor der t_j-Subtransaktion steht (*falls beide in h_k vorkommen*).

Quasi-Serialisierbarkeit (2)

Definition 10.9

Eine Menge $\{h_1,\ldots,h_n\}$ lokaler Historien heißt **quasi-serialisierbar**, falls es eine quasi-serielle Menge $\{h'_1,\ldots,h'_n\}$ lokaler Historien gibt, so dass $h_i \equiv_C h'_i$ für $1 \leq i \leq n$.

Definition 10.10

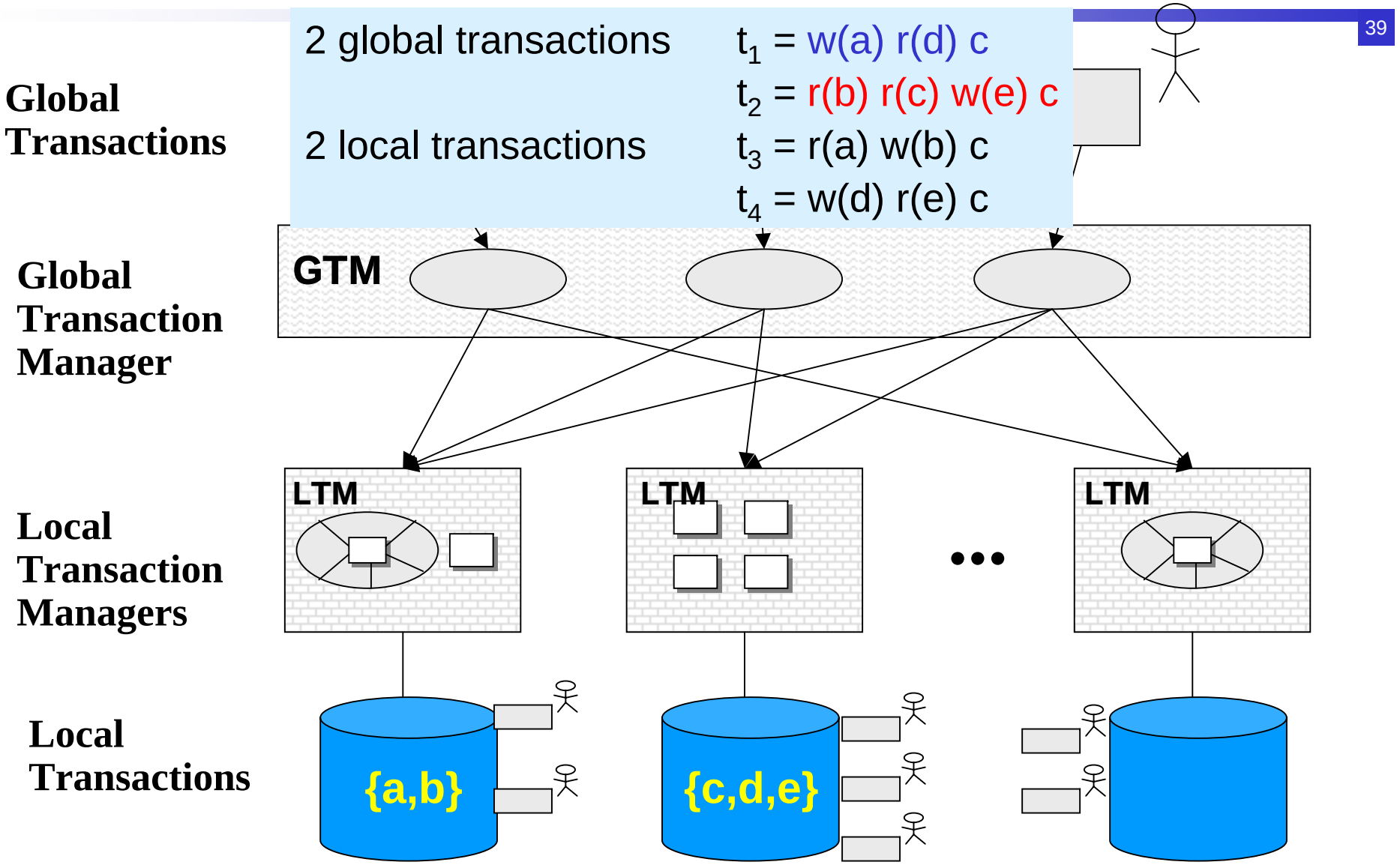
Eine globale Historie heißt **quasi-serialisierbar**, falls die Menge ihrer Lokalprojektionen quasi-serialisierbar ist.

Definition quasi-seriell betrachtet nur globale Transaktionen
$\Rightarrow$ Quasi-Serialisierbarkeit ignoriert lokale Transaktionen

Quasi-Serialisierbarkeit (3)

2 global transactions
t_1 = w(a) r(d) c
t_2 = r(b) r(c) w(e) c

2 local transactions
t_3 = r(a) w(b) c
t_4 = w(d) r(e) c



Quasi-Serialisierbarkeit (4)

Server 1: $h_1 = w_1(a)\ c_1\ r_3(a)\ w_3(b)\ c_3\ r_2(b)\ c_2$ CSR: $t_1 < t_3 < t_2$

Server 2: $h_2 = r_2(c)\ w_4(d)\ r_1(d)\ c_1\ w_2(e)\ c_2\ r_4(e)\ c_4$ CSR: $t_2 < t_4 < t_1$

$\equiv_C w_4(d)\ r_1(d)\ c_1\ r_2(c)\ w_2(e)\ c_2\ r_4(e)\ c_4$

seriell: $t_1 < t_3 < t_2$

quasi-seriell (in beiden t_1 vor t_2)

h_1 und h_2 sind Projektionen der (somit quasi-serialisierbaren) Historie

$h = w_1(a)\ r_3(a)\ r_2(c)\ w_4(d)\ r_1(d)\ c_1\ w_3(b)\ c_3\ r_2(b)\ w_2(e)\ c_2\ r_4(e)\ c_4$

Zweistufige Serialisierbarkeit (1)

Weitere Abschwächung der Quasi-Serialisierbarkeit:

Der GTM kann die Serialisierbarkeit globaler Historien garantieren, die LTMs können die Serialisierbarkeit lokaler Schedules gewährleisten. Daher:

Definition 10.11

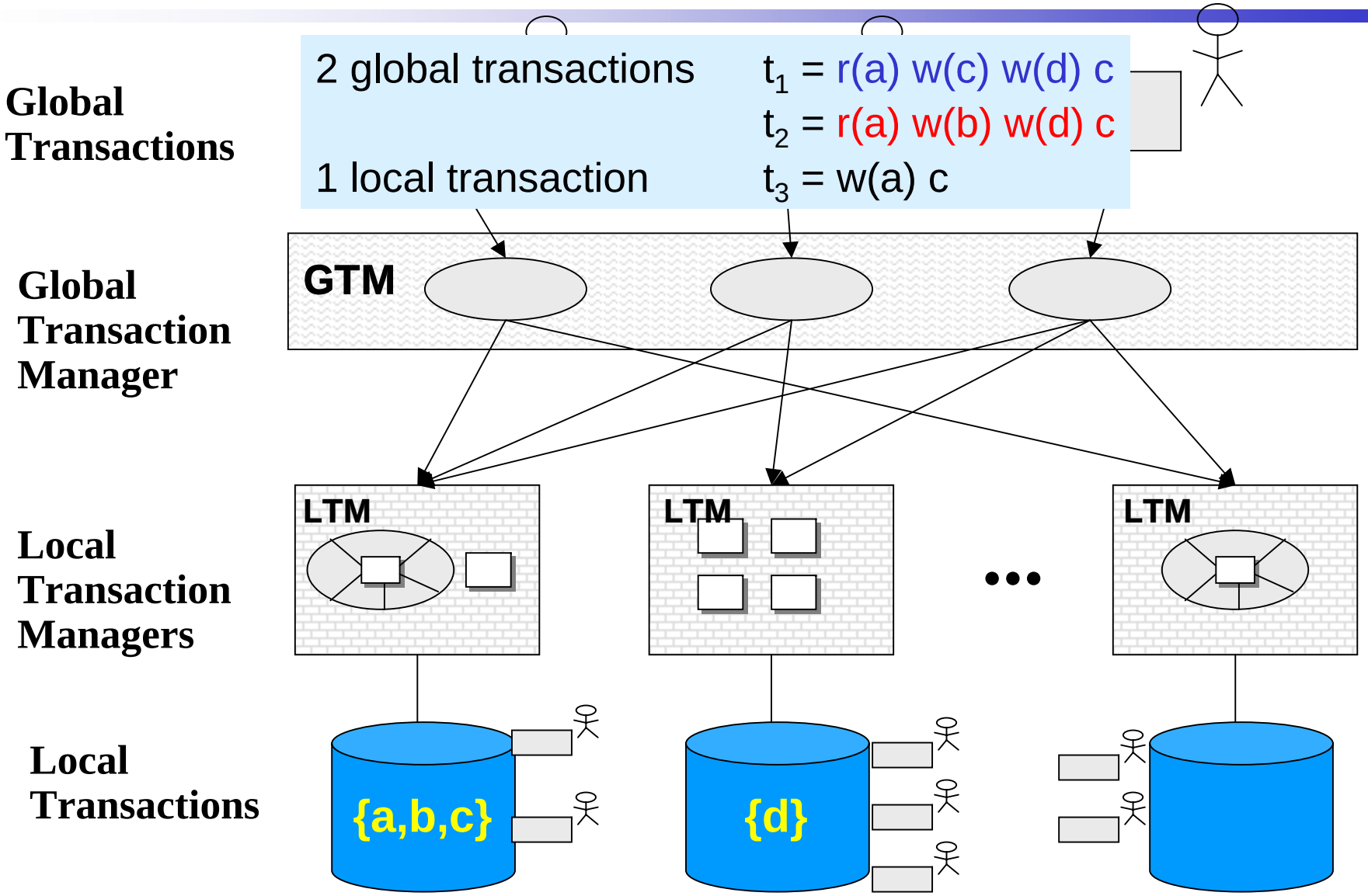
Eine globale Historie *h* heißt **zweistufig serialisierbar**, falls gilt:

- $\forall i, 1 \leq i \leq n \;\; \Pi_i(h) = h_i \in CSR$
- Ist *T* die Menge der in *h* vorkommenden globalen Transaktionen, so gilt $\Pi_T(h) \in CSR$

Anmerkung:

Jede global serialisierbare Historie ist zweistufig serialisierbar. Die Umkehrung dieses Satzes gilt nicht.

Zweistufige Serialisierbarkeit (2)



Zweistufige Serialisierbarkeit (3)

Server 1: $h_1 = r_1(a)\ w_1(c)\ c_1\ w_3(a)\ c_3\ r_2(a)\ w_2(b)\ c_2$ CSR: $t_1 < t_3 < t_2$

Server 2: $h_2 = w_2(d)\ c_2\ w_1(d)\ c_1$ CSR: $t_2 < t_1$

h_1, h_2 seriell

h_1 und h_2 sind Projektionen der Historie $\Pi_T(h) \in$ CSR: $t_2 < t_1$

$h = r_1(a)\ w_1(c)\ w_2(d)\ w_3(a)\ c_3\ w_1(d)\ c_1\ r_2(a)\ w_2(b)\ c_2$

h nicht *CSR*!

DBS

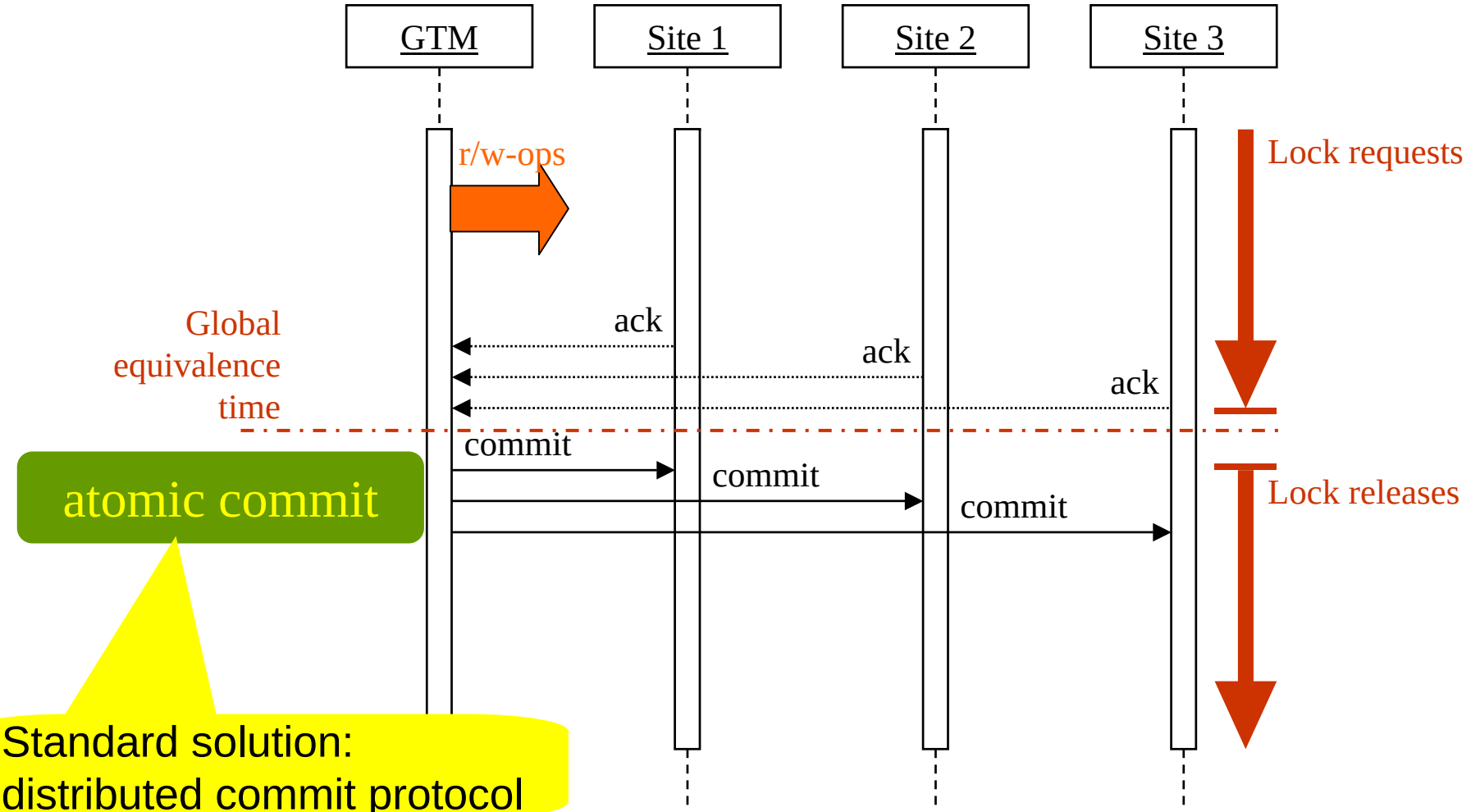
Universität Karlsruhe (TH)

Chapter 11

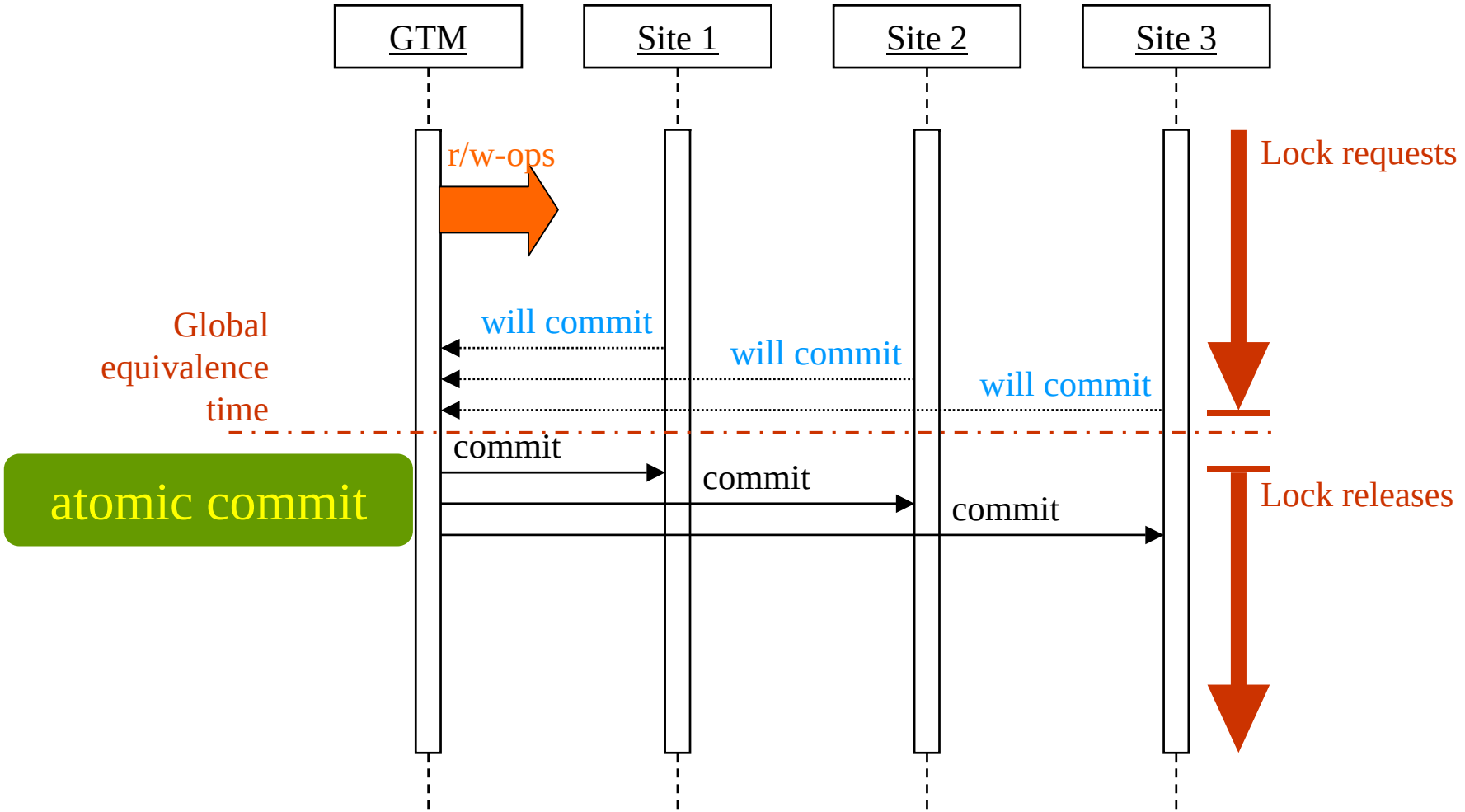
Distributed Transactions: Recovery

Global Atomicity

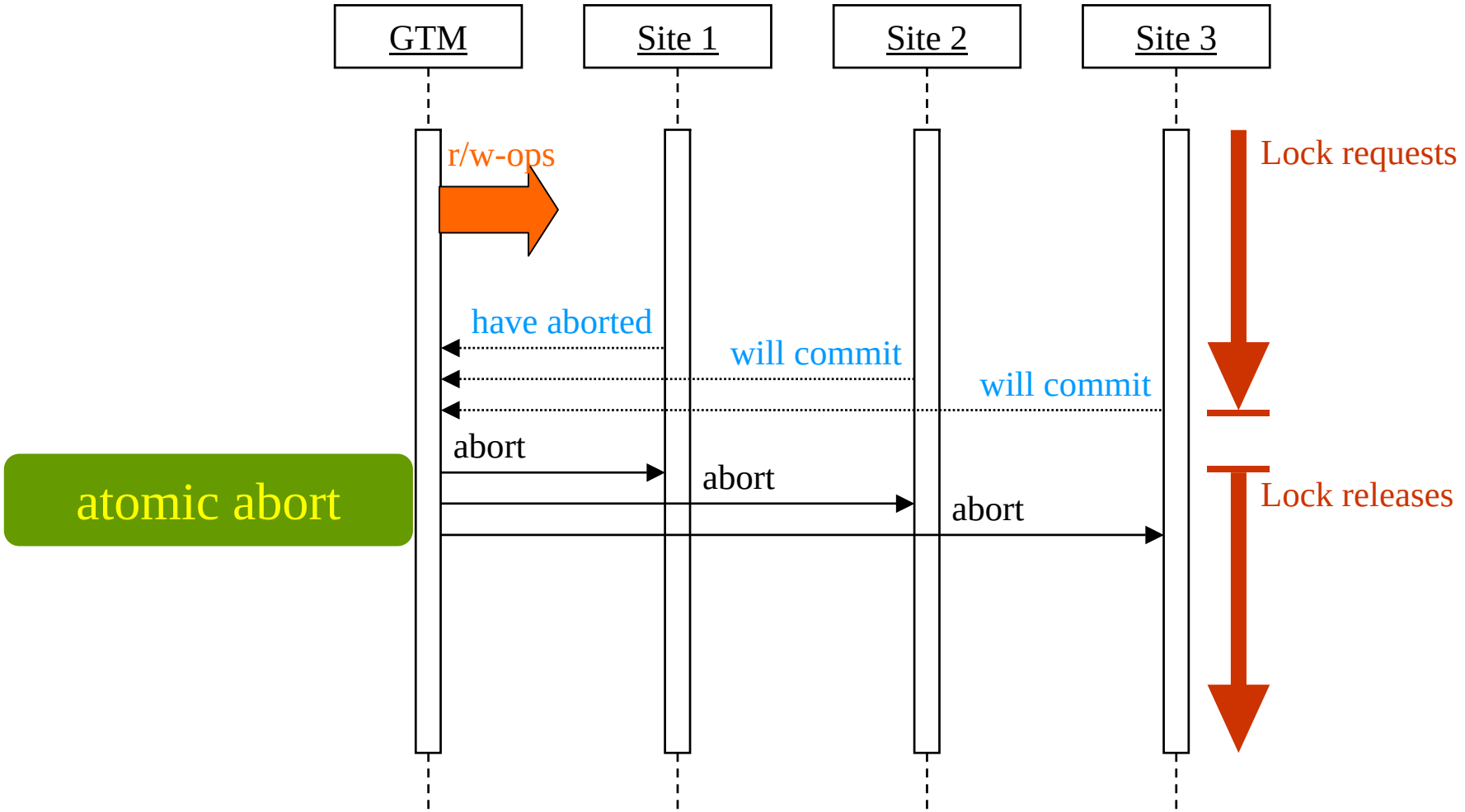
Remember: Rigorous / commit-deferred



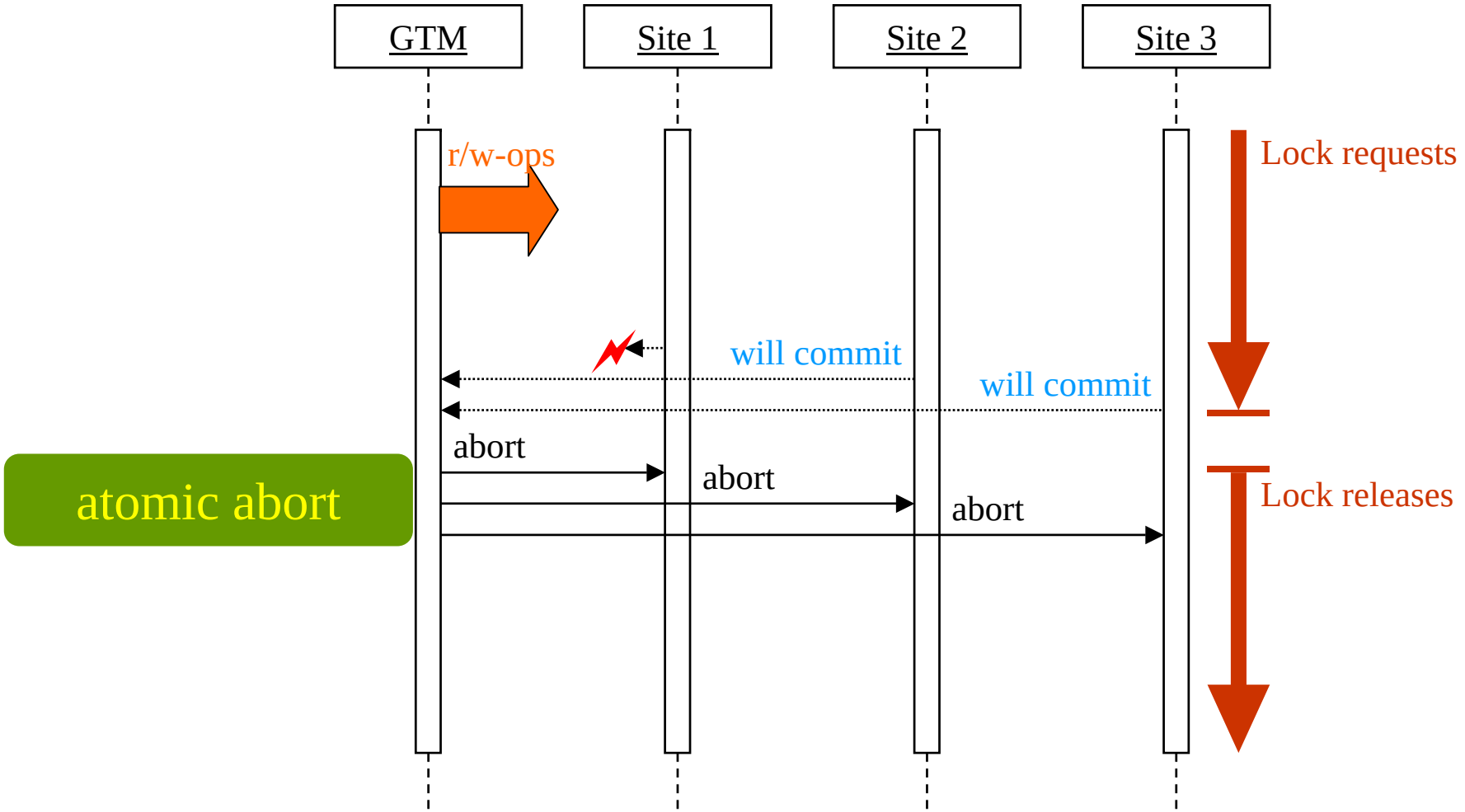
Atomic commitment



Atomic abort



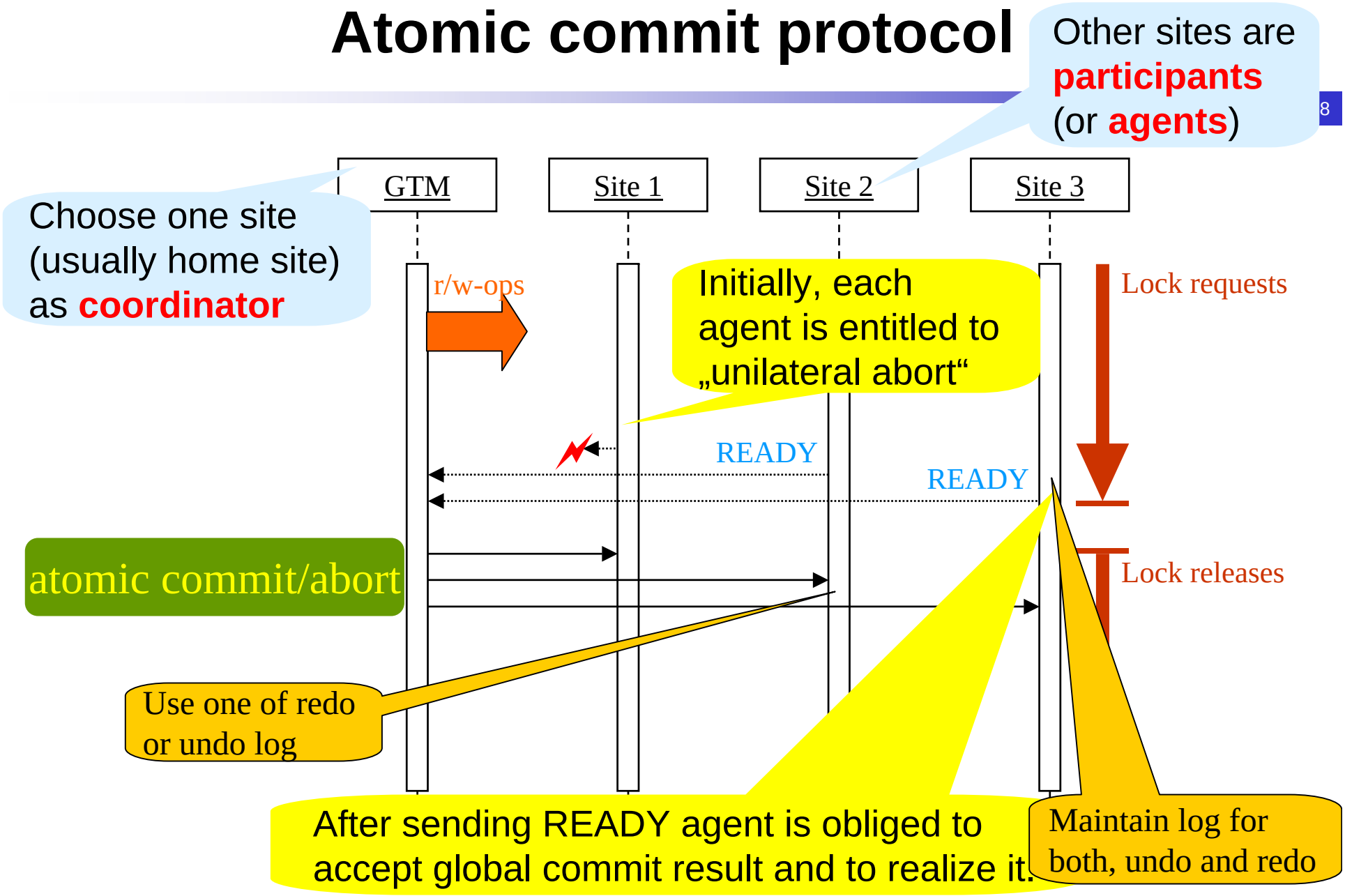
Presumed abort



Atomicity

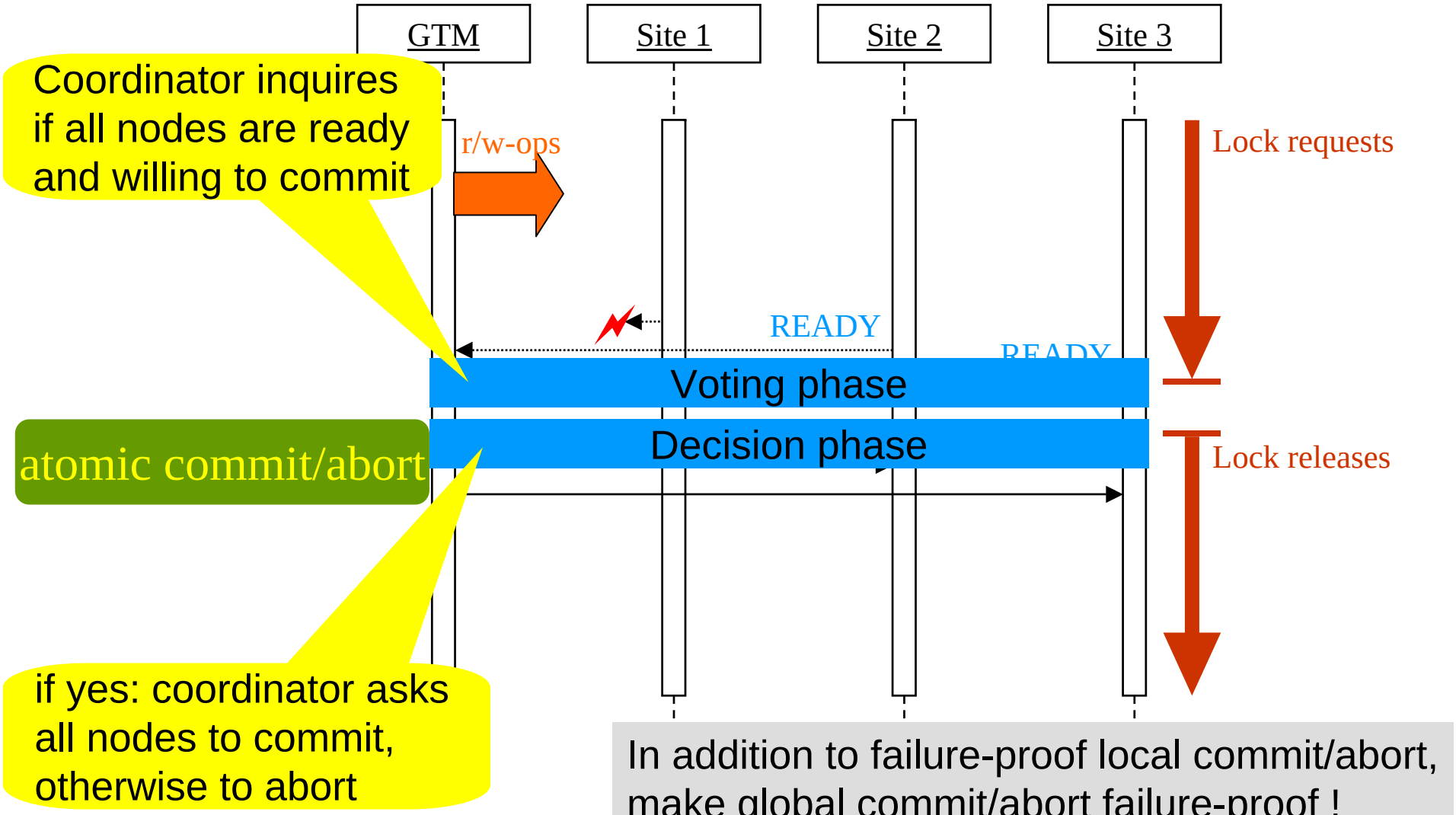
- Prerequisite of **local atomicity**: All transactions at all nodes are atomic. ⇒ Each node with a complete transaction manager as discussed earlier.
- Problem of **global atomicity**: All nodes decide exactly the same for all distributed transactions, i.e., global commit or abort, and enforce the decision locally.
- **Atomic Commit Protocols (ACP)** ensure that all nodes participating in a global transaction all either commit or abort this transaction.

Atomic commit protocol



Basis: Two-phase commit protocol (2PC)

Two-phase commit protocol (2PC)



Preconditions (1)

Rule 1: No site may be forced to commit a transaction.

- ⇒ All involved participants must consent to a *global commit* for the TA coordinator to issue such a commit.
- ⇒ Otherwise the coordinator must issue a *global abort*.
- ◆ Participants where *t* only invoked read operations can safely be skipped.

Preconditions (2)

Rule 2: A transaction *t* may only be committed if all written items already are locally persistent.

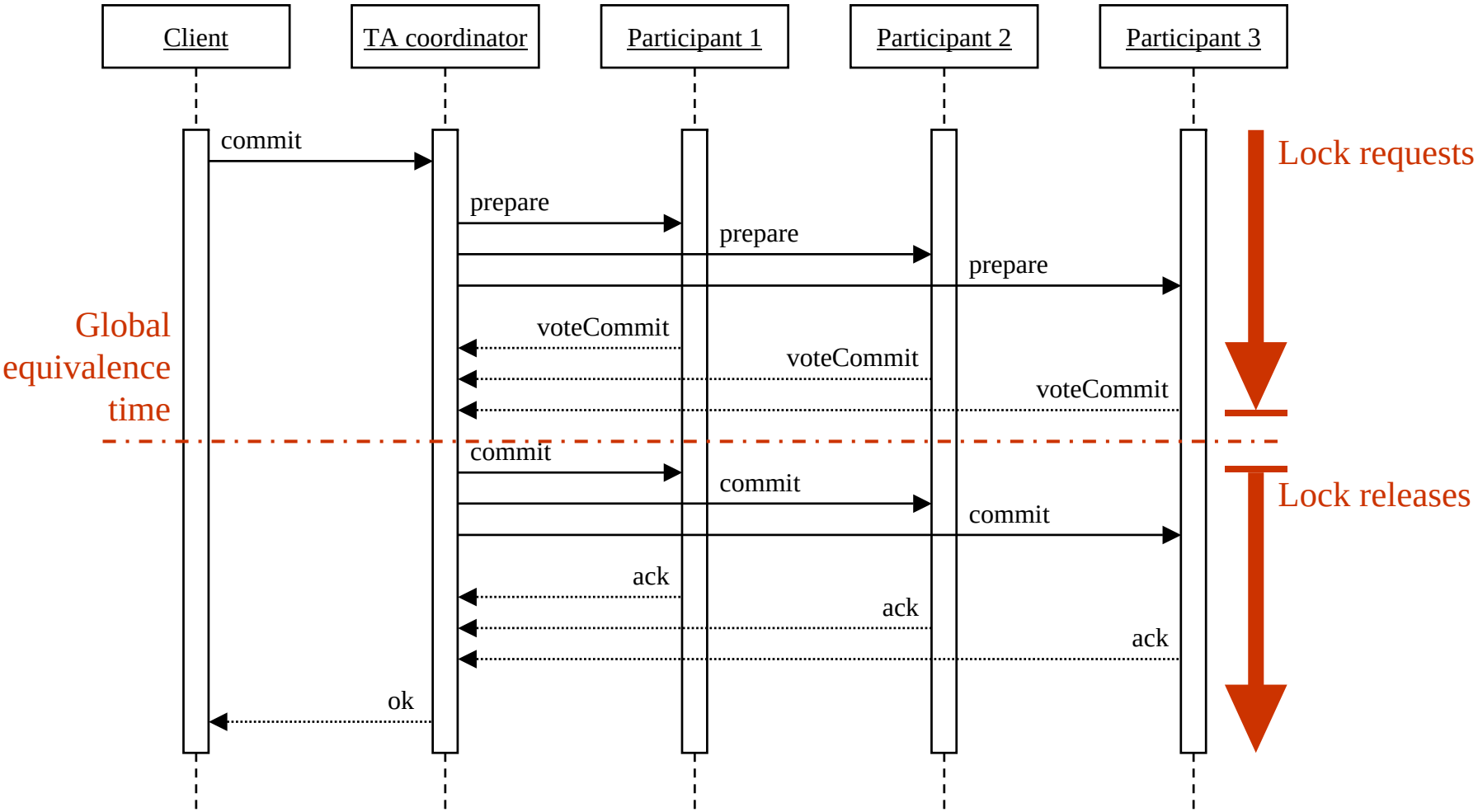
- Each site guarantees with its consent to *t* that
  - RC is satisfied,
  - Undo/Redo rule is satisfied,
  - the site will not commit other transactions that follow *t* in the local equivalent serial history before it received the coordinator‘s final decision,
  - If the site fails it can determine the outcome of *t* and if need be will be able to redo *t*.

Preconditions (3)

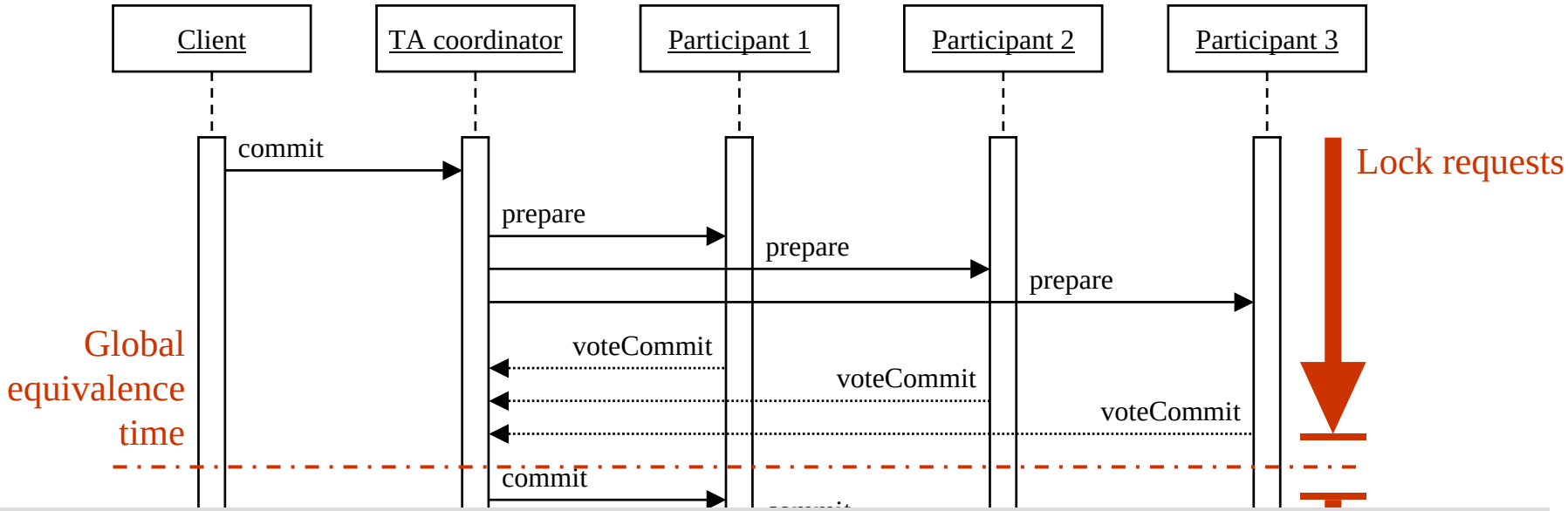
Rule 3: Participants must be informed of a global commit.

- Coordinator guarantees this to the consenting participants by maintaining a list of participants and the transaction commit in stable store until all participants have been informed.
- Presumed abort:
  - Participants that did not consent guarantee that they undo the local changes of the transaction and release the locks. They are not included in the further communication.
  - Participants that receive *global abort* from the coordinator do likewise.

Global commit via 2PC



Global commit via 2PC



Voting phase:

a) Coordinator sends *prepare* message to each participant.

b) After arrival of *prepare*, the participant responds with *voteCommit* or *voteRollback*. In case of *voteRollback* the participant locally aborts the transaction.
   - Each participant has exactly one vote.
   - The participant must not revise its vote.

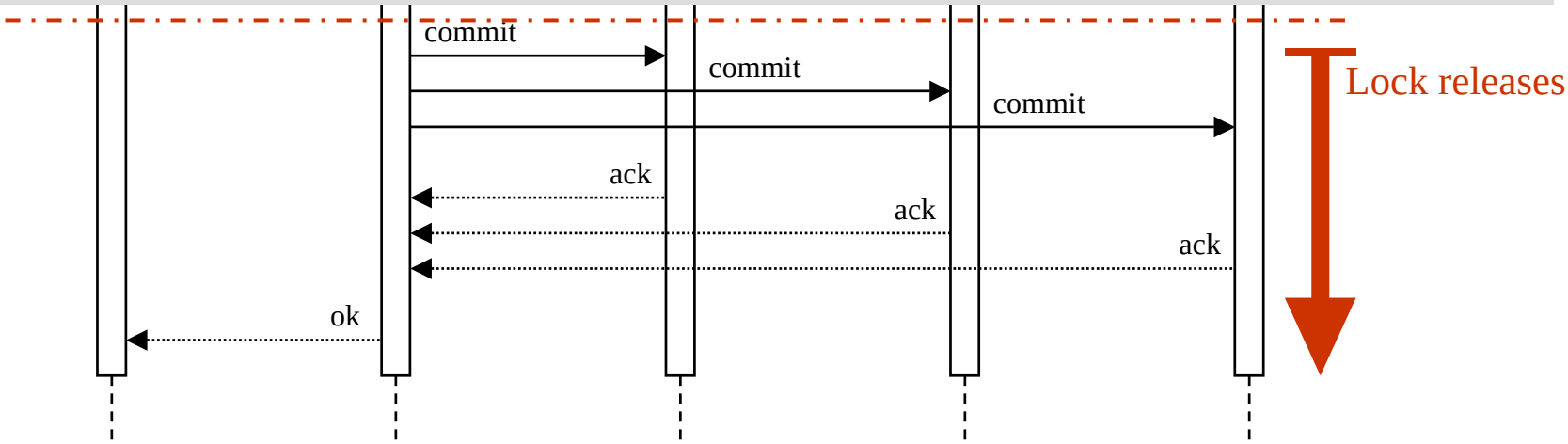
Global commit via 2PC

Decision phase:

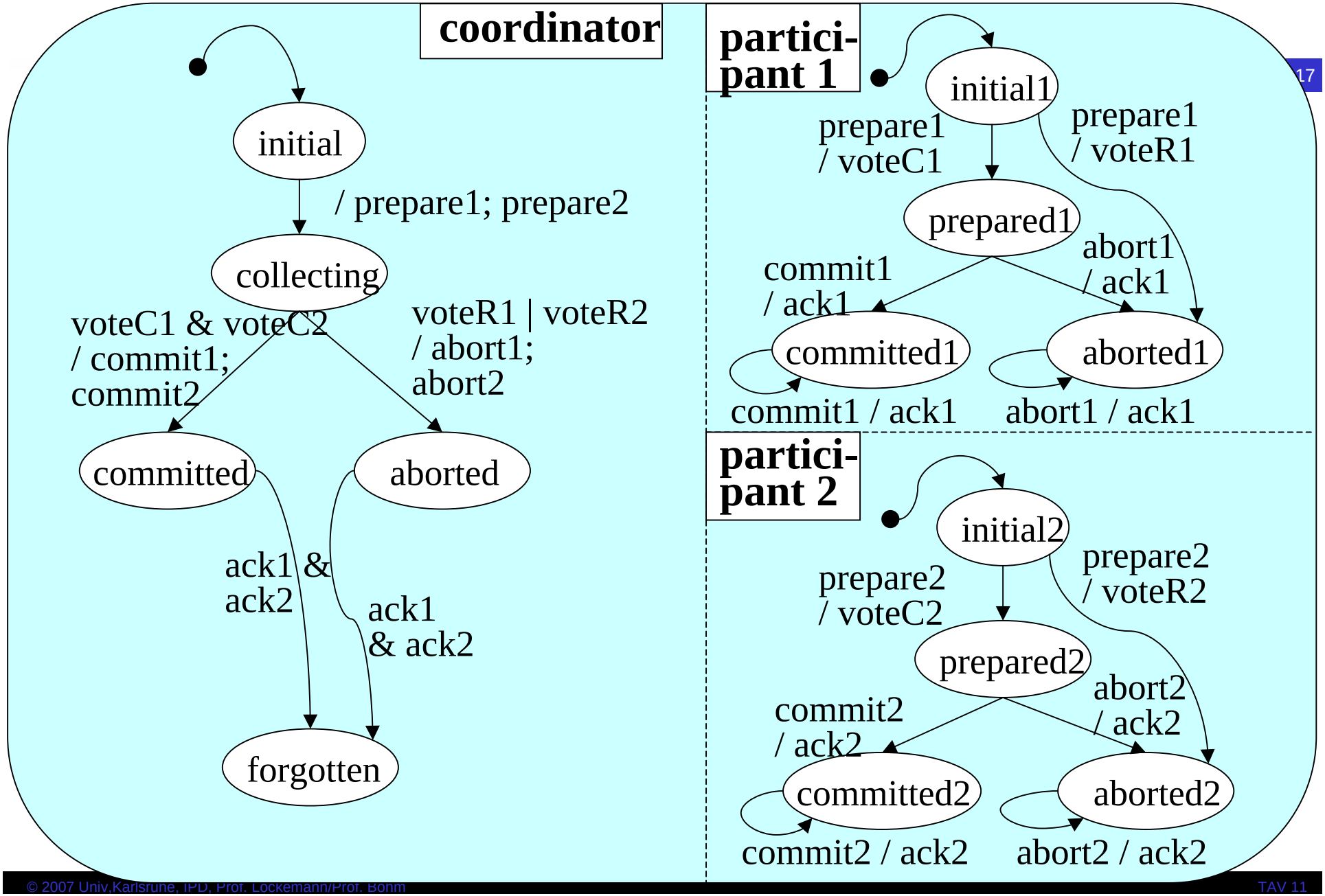
a) The coordinator collects the votes including its own. If all are *voteCommit* it decides on global commit and sends *commit* to all participants.

b) Otherwise it decides on global abort and sends *abort* to all participants that voted *voteCommit*.

c) Each participant that voted *voteCommit* waits for a *commit* or *abort* message, and after receiving it processes it accordingly.



Statechart for Basic 2PC



Scenario

Scenario

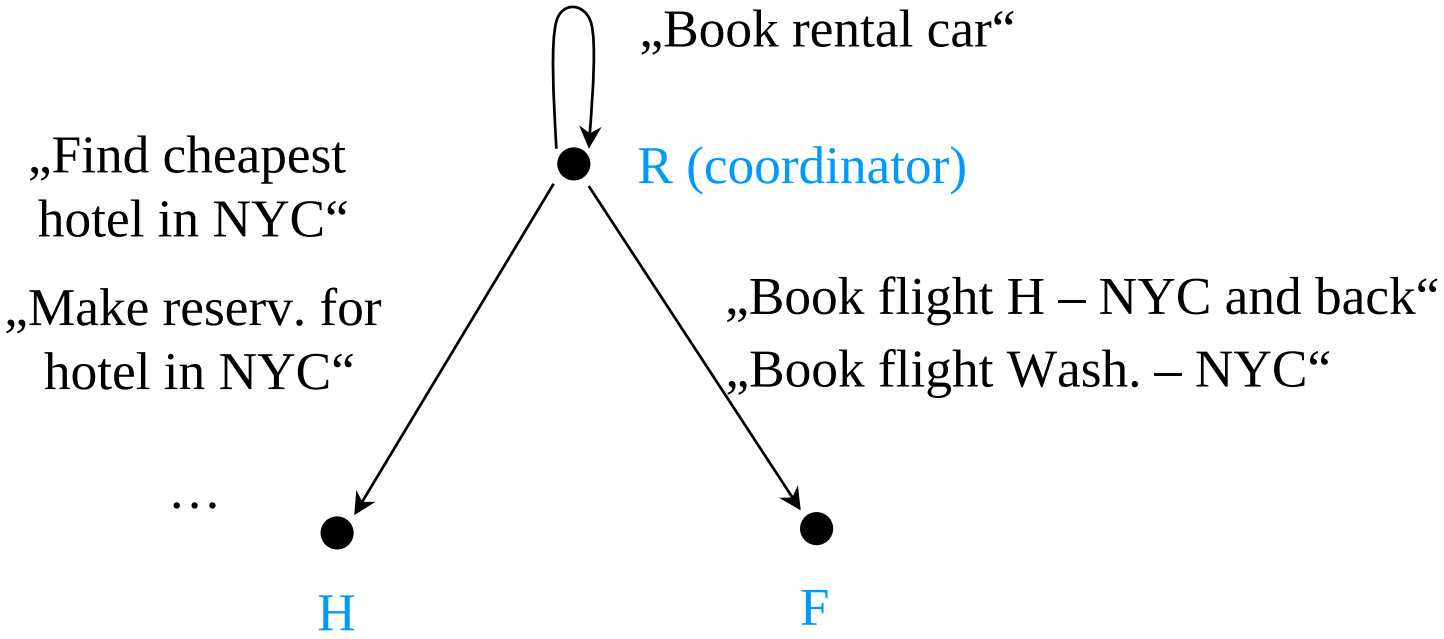
- Transaction *t* that consists of several operations:
  - "Book flight from Hannover to New York City and back."
  - "Book flight from Washington DC to NYC."
  - "Find cheapest hotel in NYC."
  - "Make reservation for it."
  - "Make reservation for any hotel in Philadelphia."
  - dto. Washington DC
  - "Book rental car NYC – Washington DC."

Flight DB (Node F)

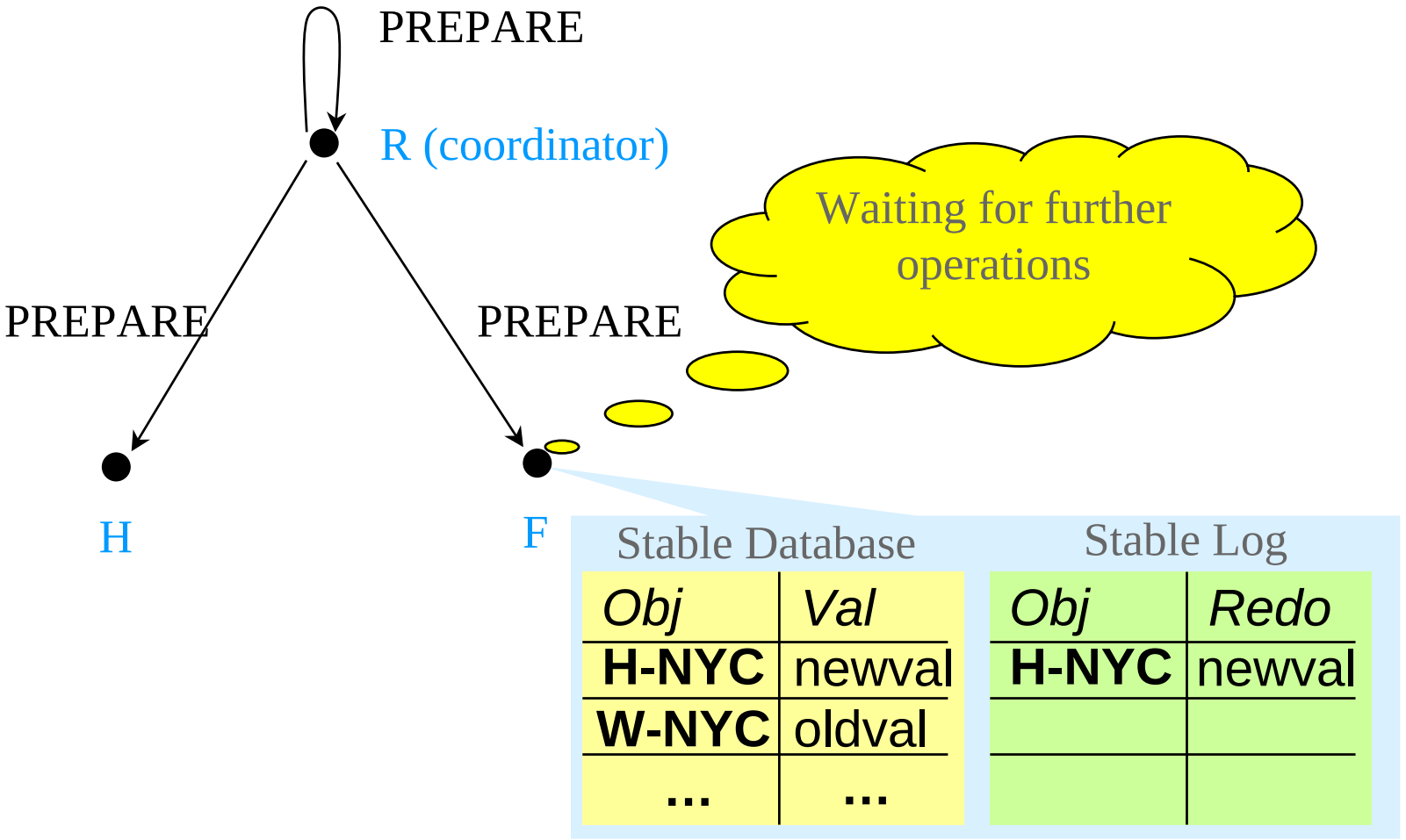
Hotel DB (Node H)

Rental Car DB (Node R)

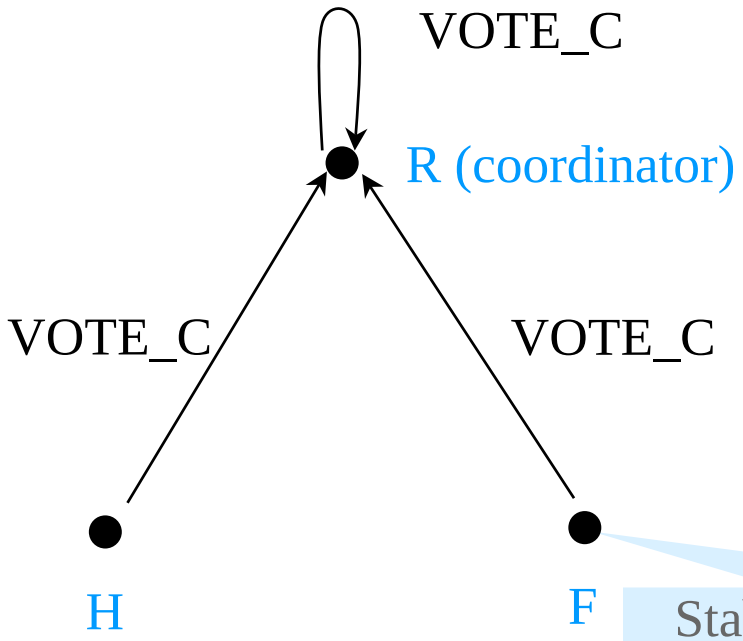
Transaction execution - no failures



2PC execution – regular commit (1)



2PC execution – regular commit (2)



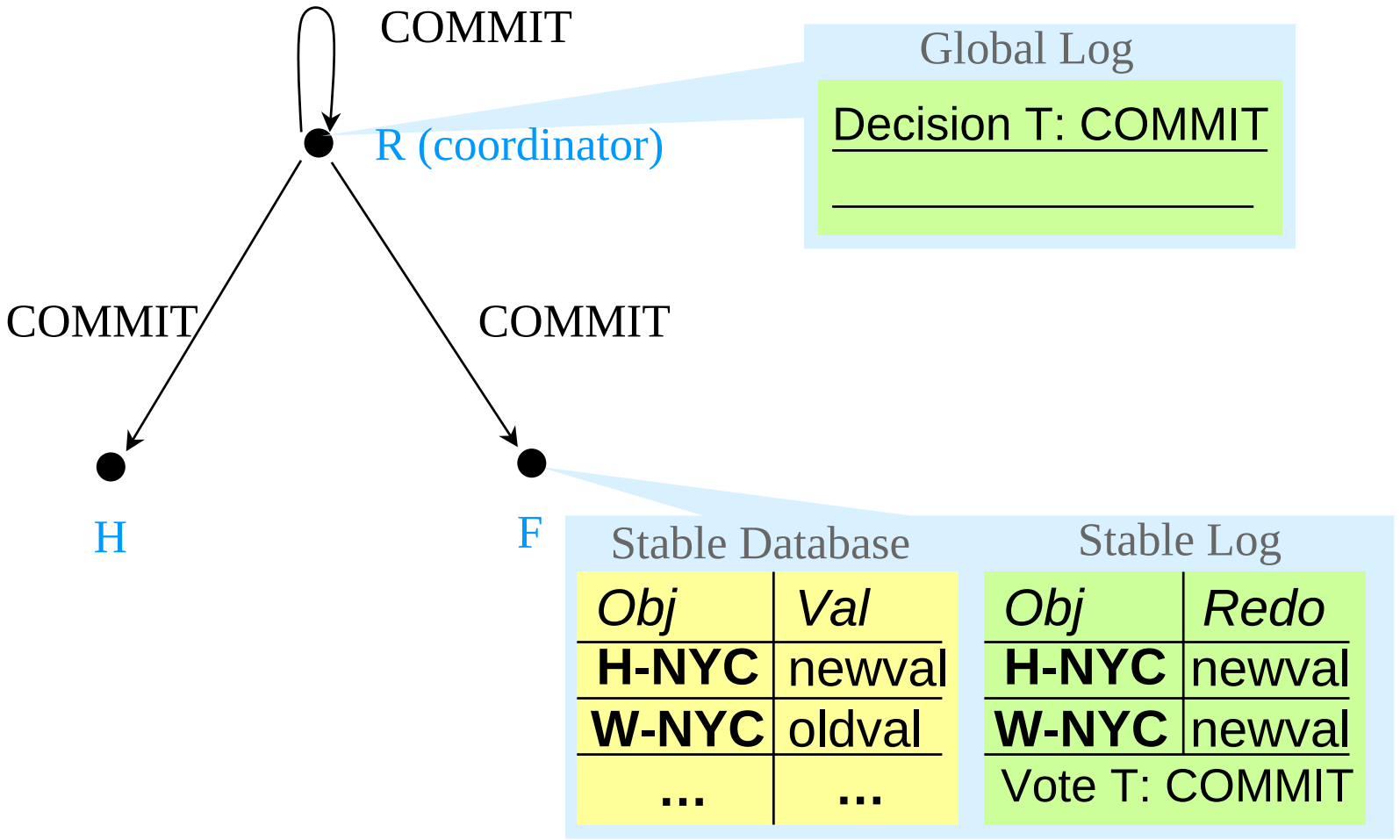
Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

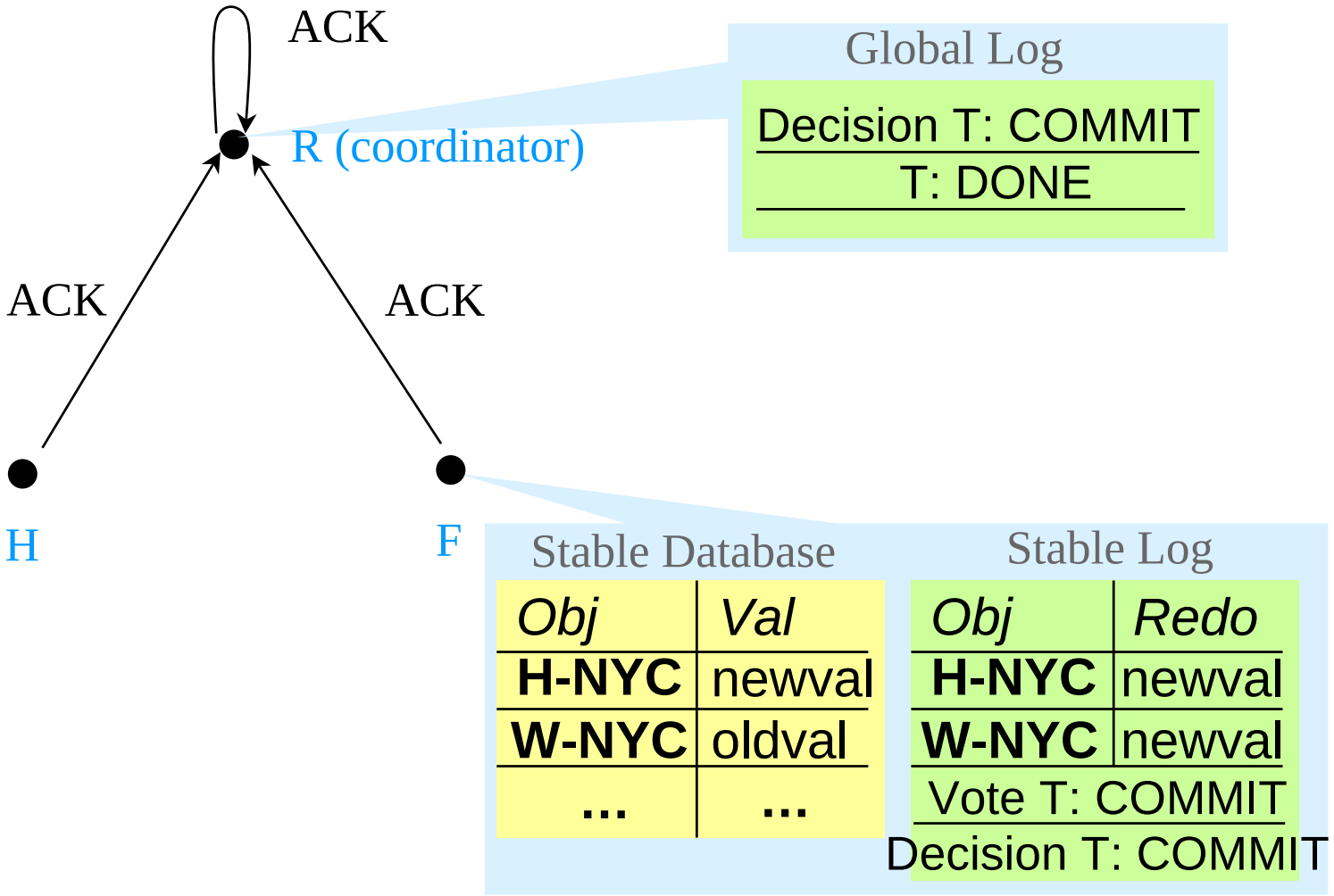
Stable Log

Obj	*Redo*
H-NYC	newval
W-NYC	newval
Vote T: COMMIT	

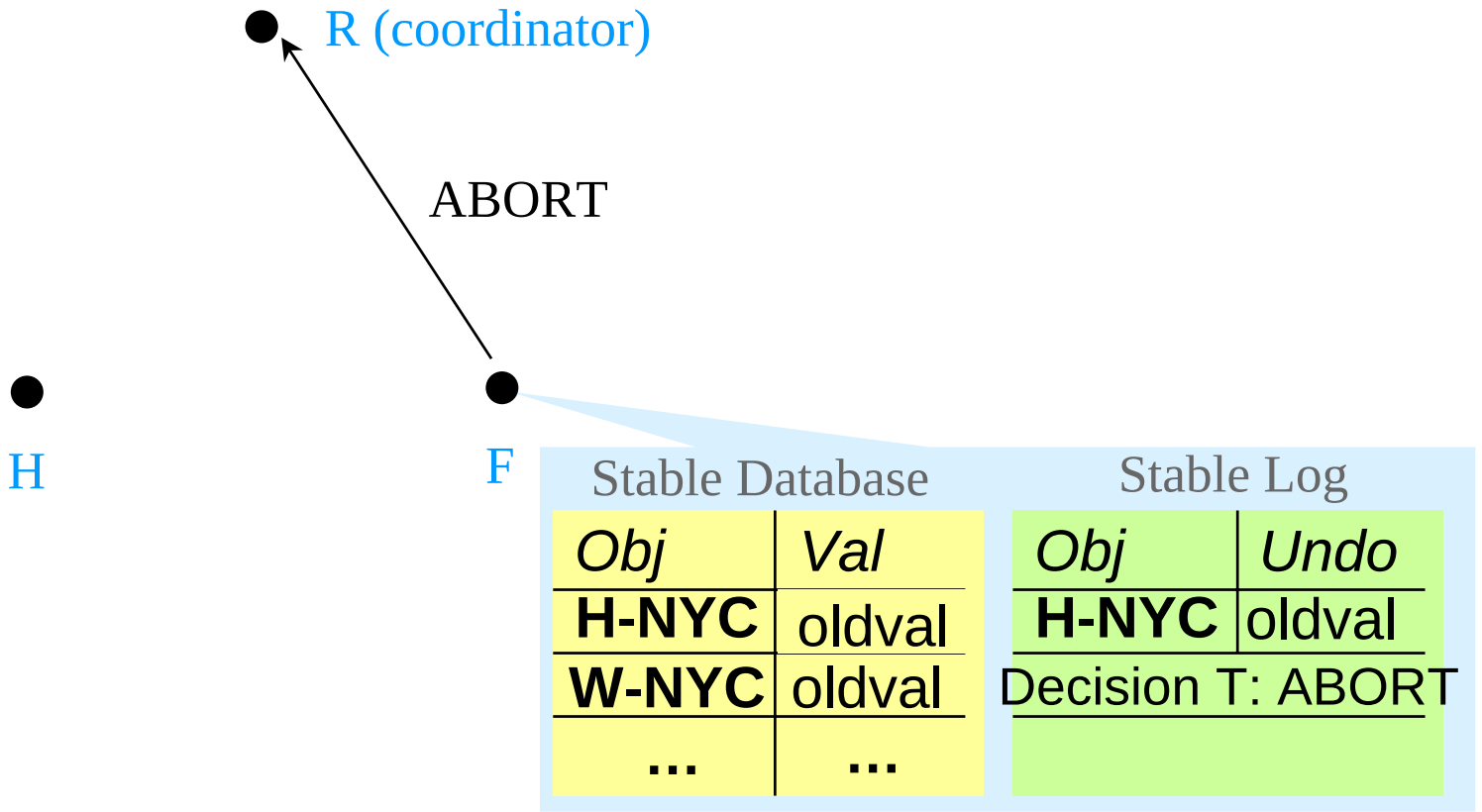
2PC execution – regular commit (3)



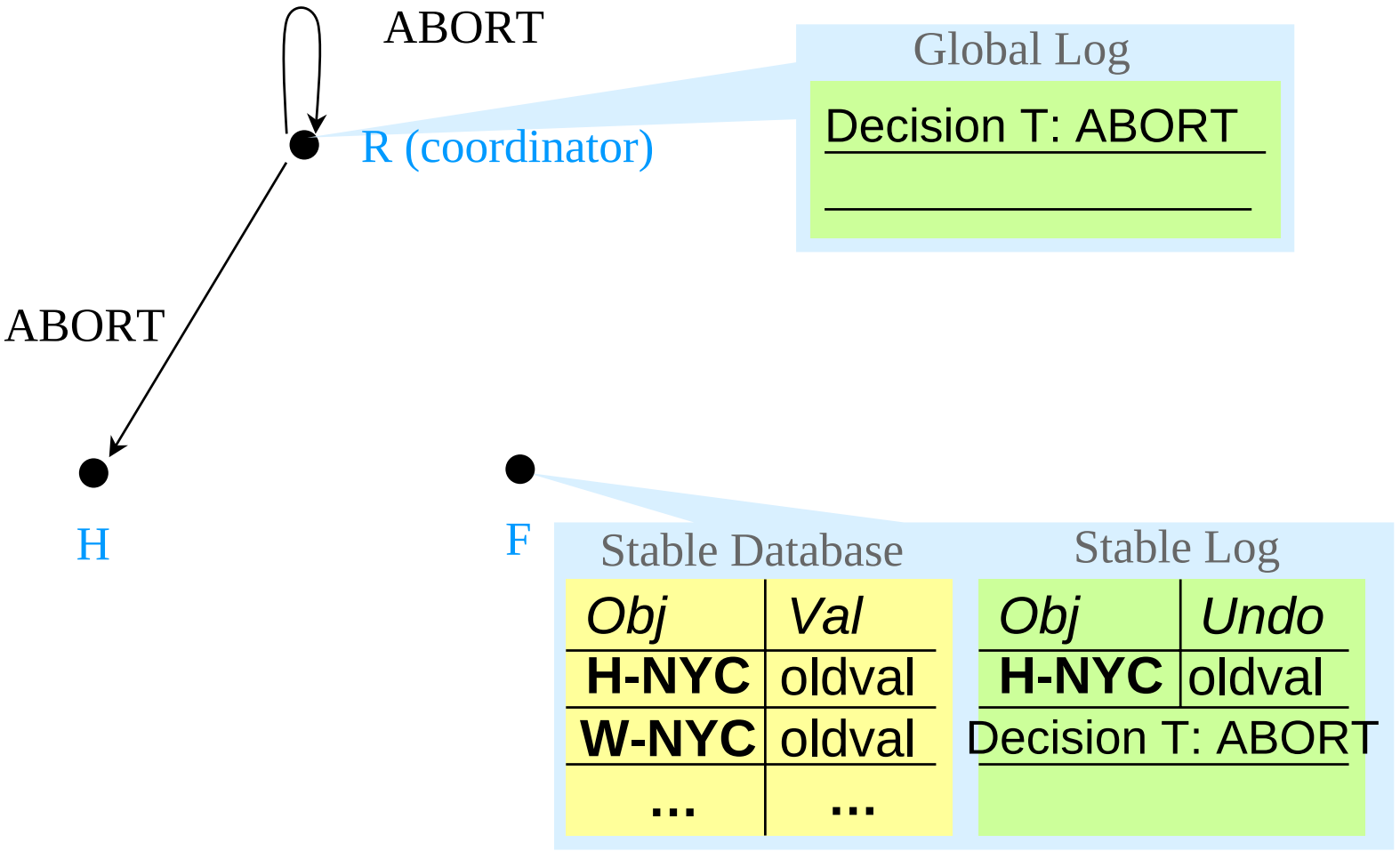
2PC execution – regular commit (4)



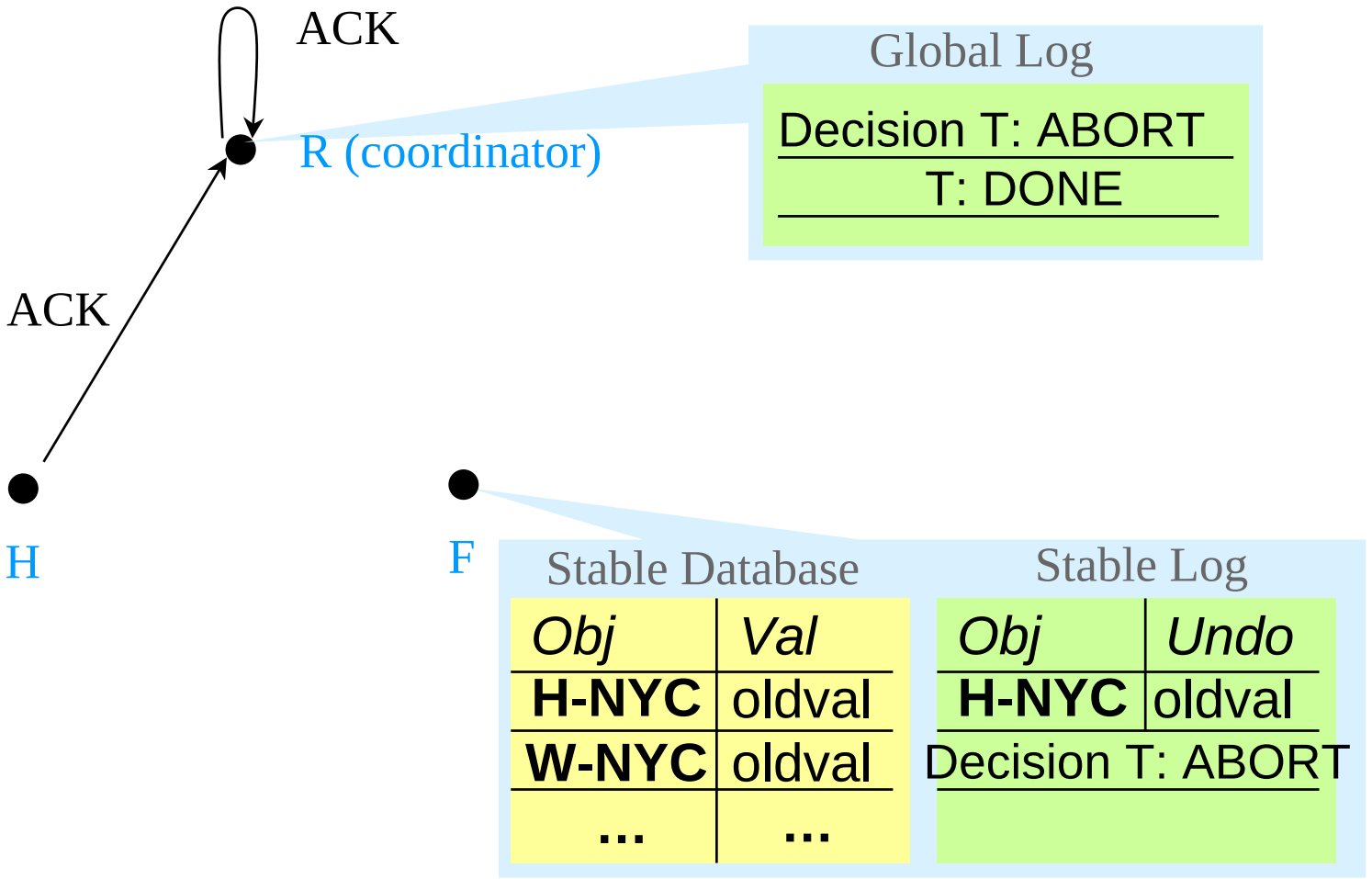
2PC execution – regular abort (1)



2PC execution – regular abort (2)



2PC execution – regular abort (3)



2PC under failures

Failure types

Site failure

- Local system crash. Correctly functioning sites are up, otherwise down.
- If all sites are down: total failure.
- If only some sites are down: partial failure.

Communication failure

- Connection loss
  - Assumption 1: Message simply disappears, no attempt at repeated delivery.
  - Assumption 2: No other failures: no message corruption or loss.

⇒ Common effect: undeliverable message

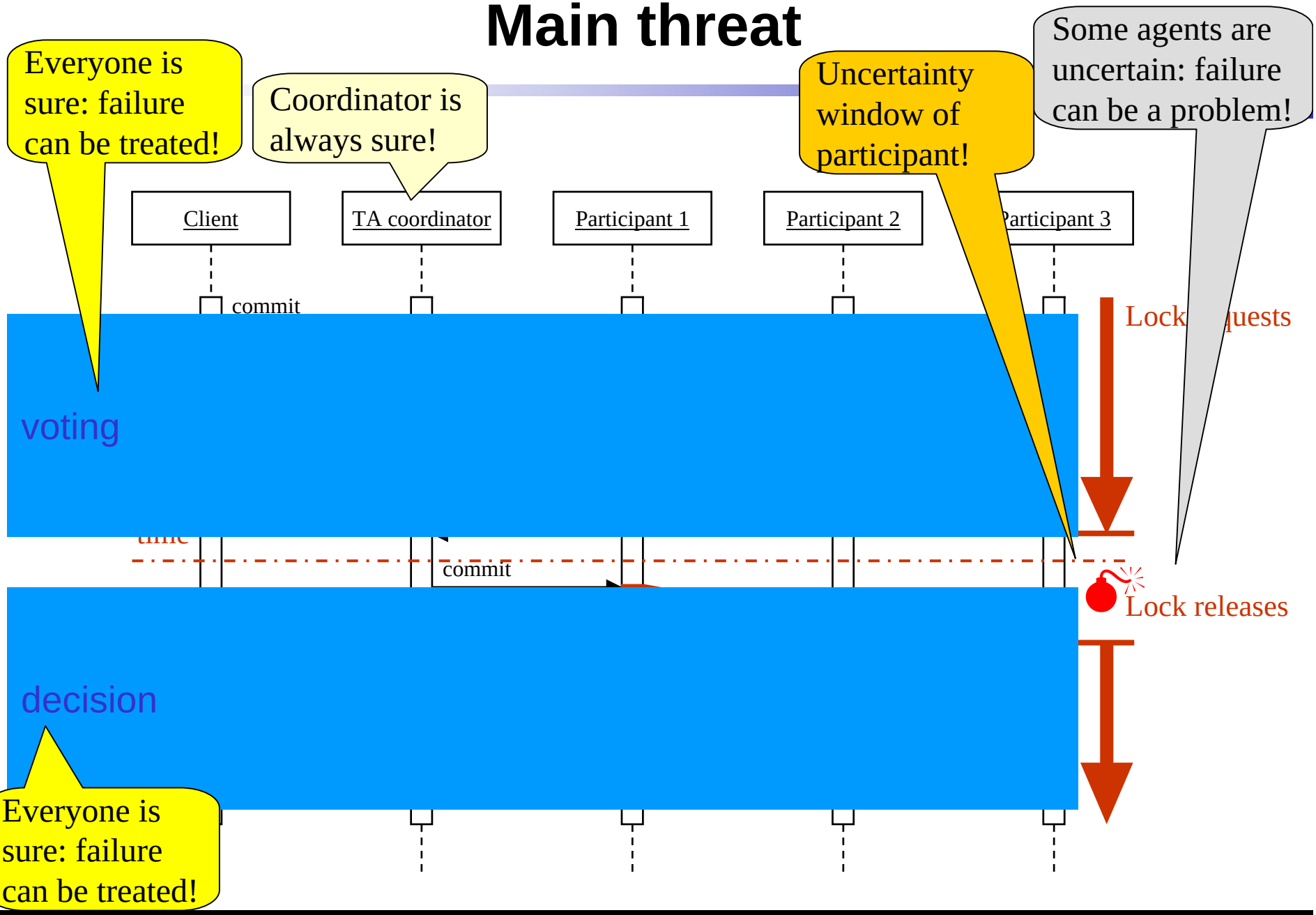
- Terminology: Site is up if it can be reached, site is down otherwise (site failure or connection loss).

Atomicity: constituent protocols

- **Completion protocol** aspects:
  - **Global commit**.
  - **Global abort**.

  } combined in 2PC ✓
- **Termination protocol** for failure recovery. ⬅

Main threat

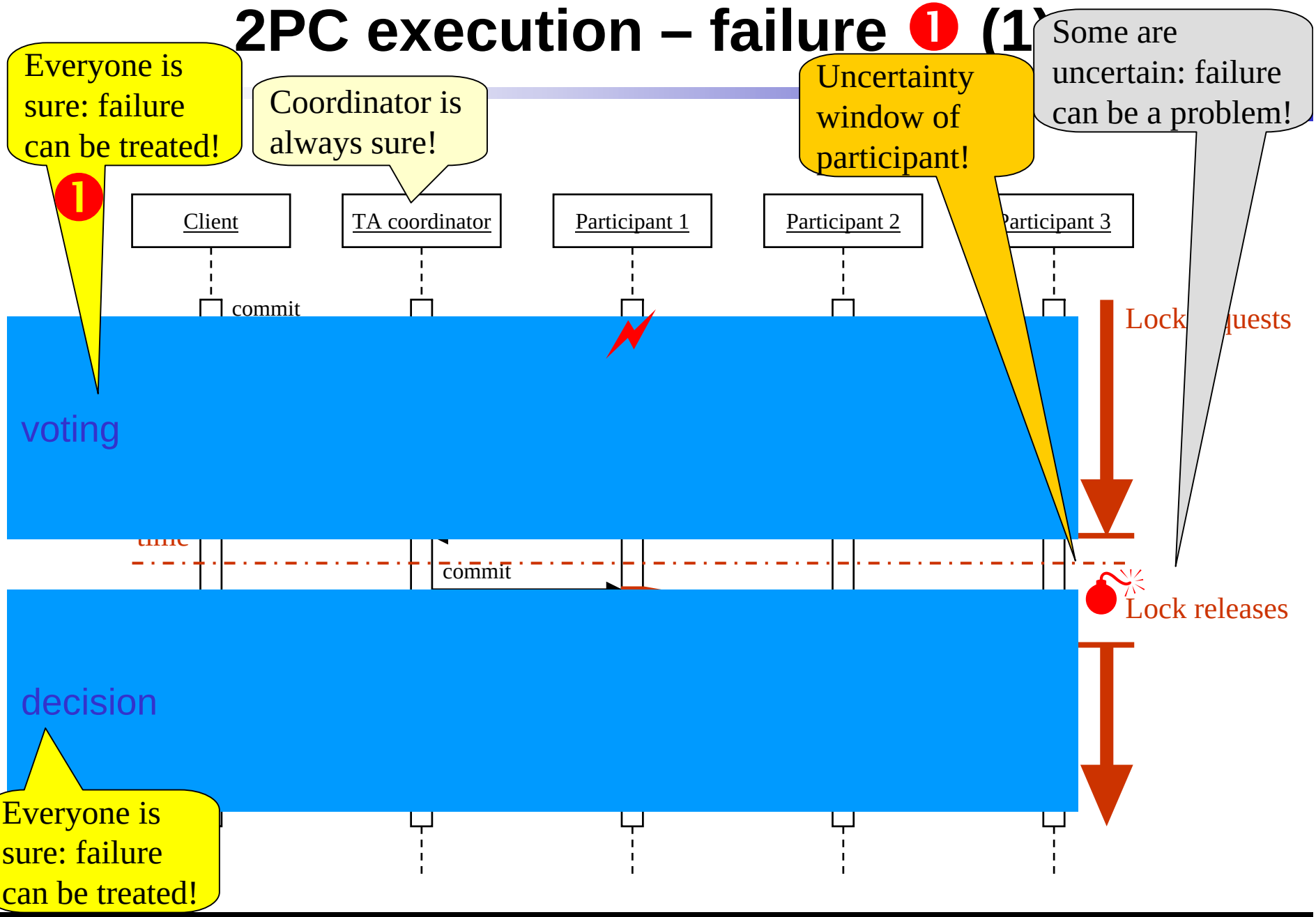

Everyone is sure: failure can be treated!
Coordinator is always sure!
Uncertainty window of participant!
Some agents are uncertain: failure can be a problem!
Client
TA coordinator
Participant 1
Participant 2
Participant 3
commit
voting
Lock requests
commit
Lock releases
decision
Everyone is sure: failure can be treated!

Distributed Transaction Log

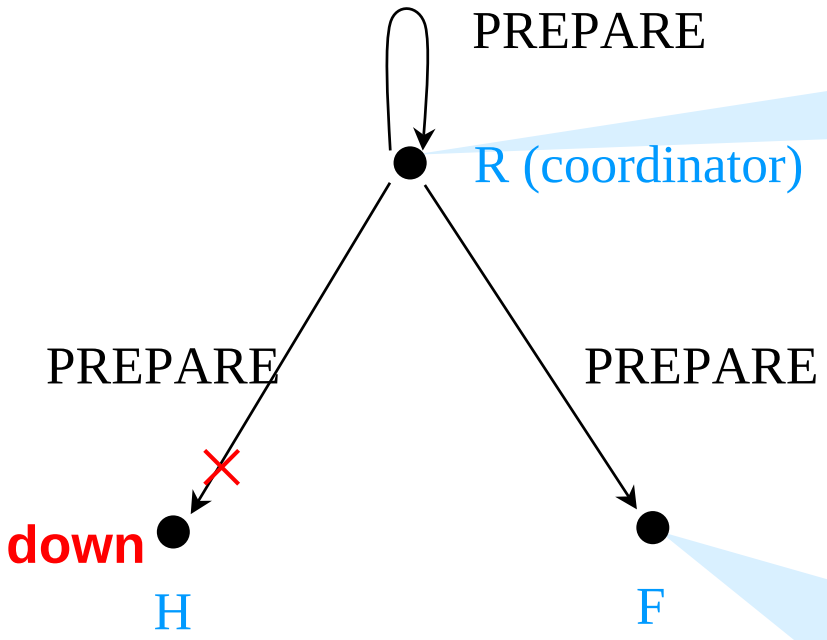
Failure of a site ⇒ After its restart its TM must take a decision (commit or abort) that agrees with the decision of all other nodes (coordinator and participants).

- **Refined**: Each site maintains a **Distributed Transaction Log** (DTL) and updates it as follows:
  - Coordinator writes Start-2PC with IDs of the participants into its DTL.
  - A consenting participant writes voteCommit and the IDs of the other nodes to its DTL. A disagreeing participant writes abort to its DTL.
  - Coordinator writes commit or abort to its DTL before sending the respective message.
  - The participants write the received decision to their DTL.
  - Coordinator writes done after receiving the acknowledgements of all participants.

2PC execution – failure ❶ (1)



2PC execution – failure ❶ (2)



Global Log

Start-2PC

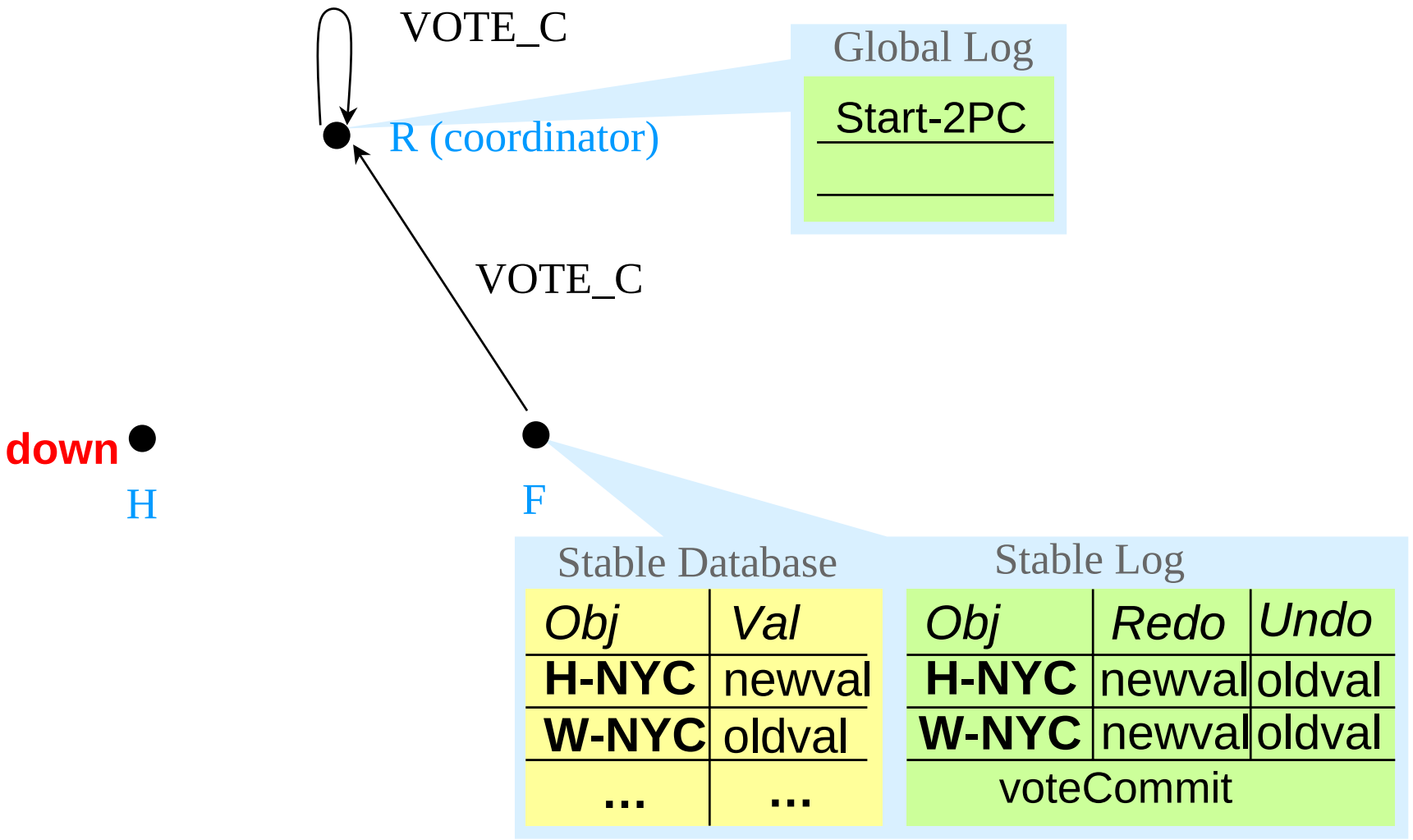
Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

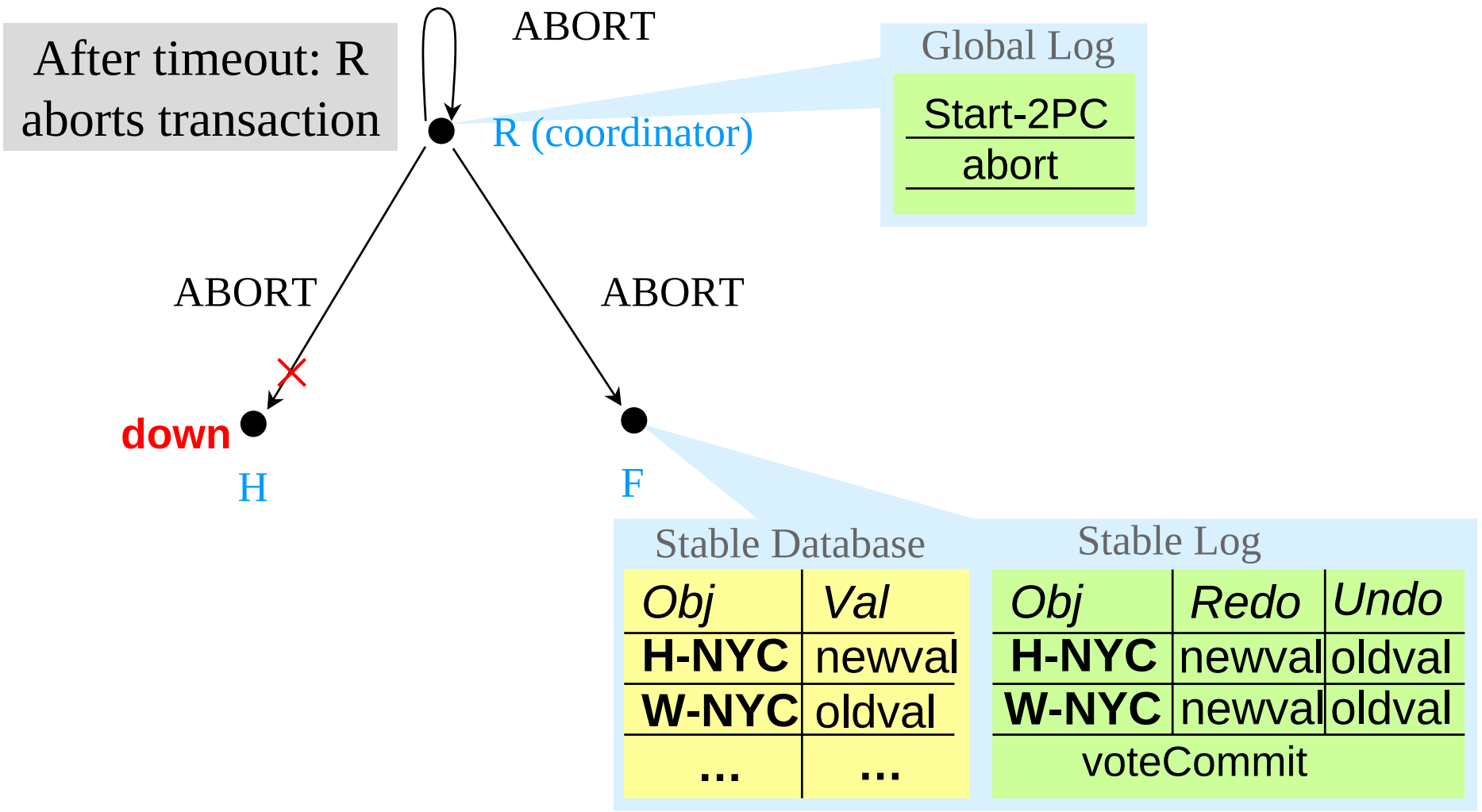
Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval

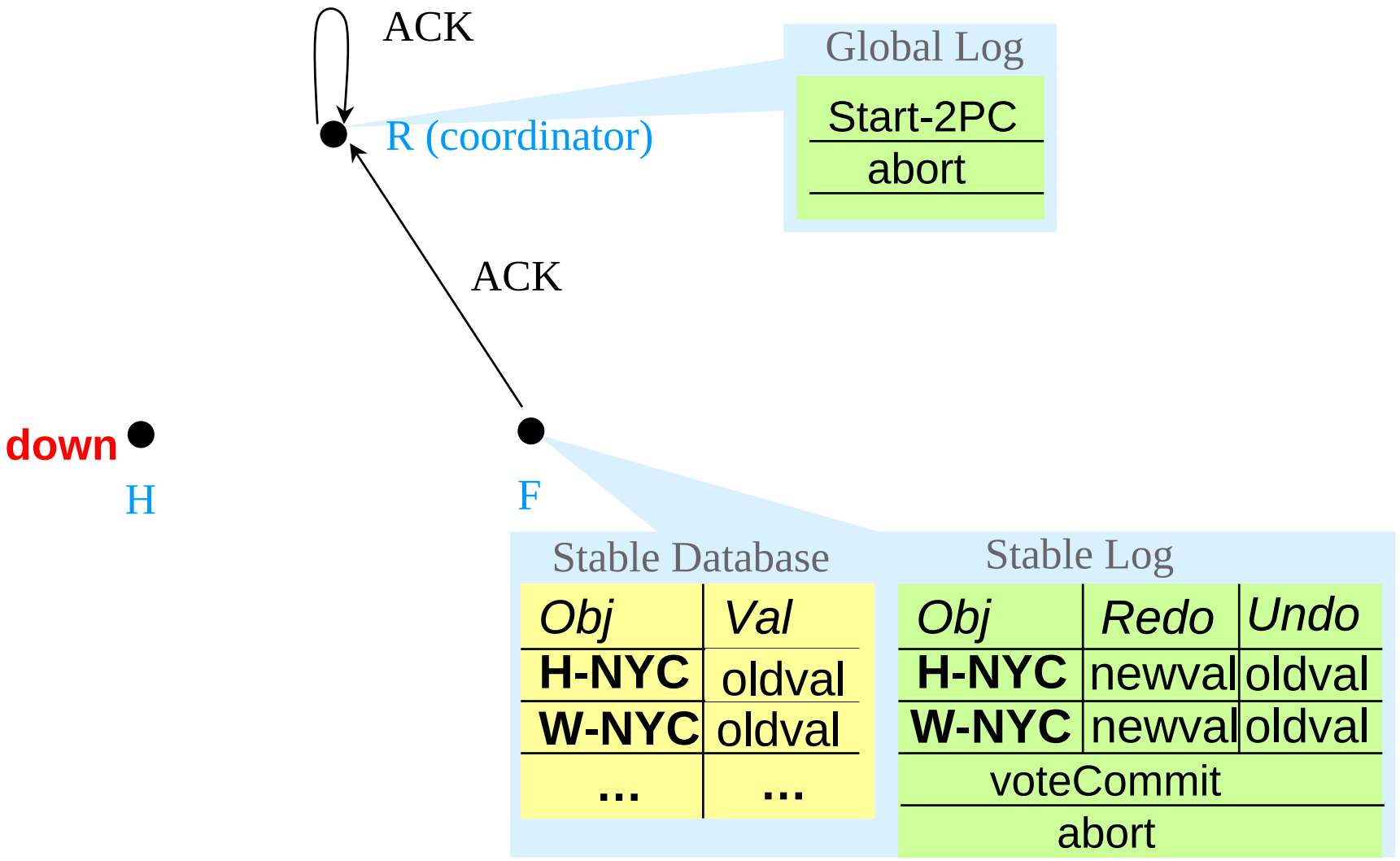
2PC execution – failure ❶ (3)



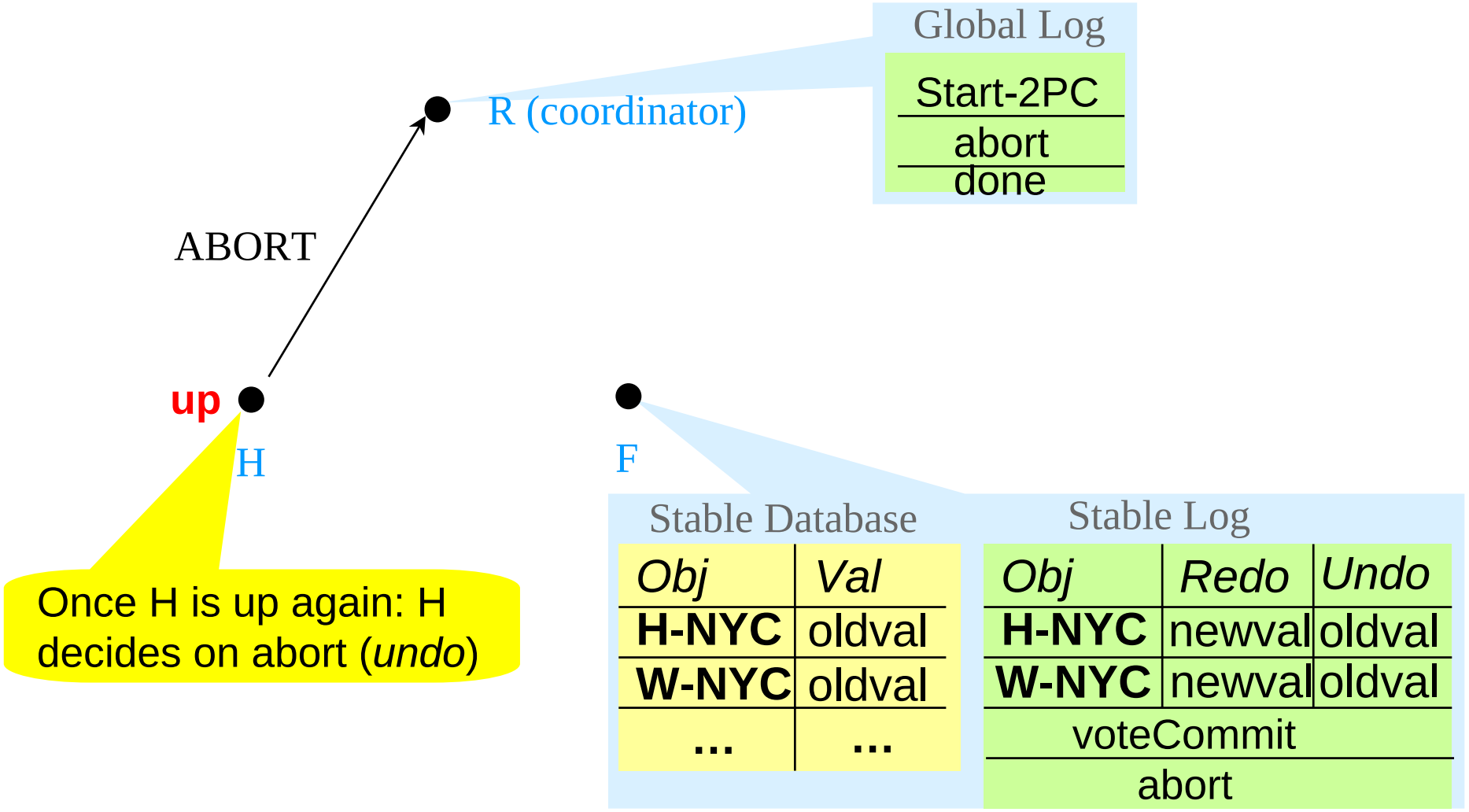
2PC execution – failure ❶ (4)



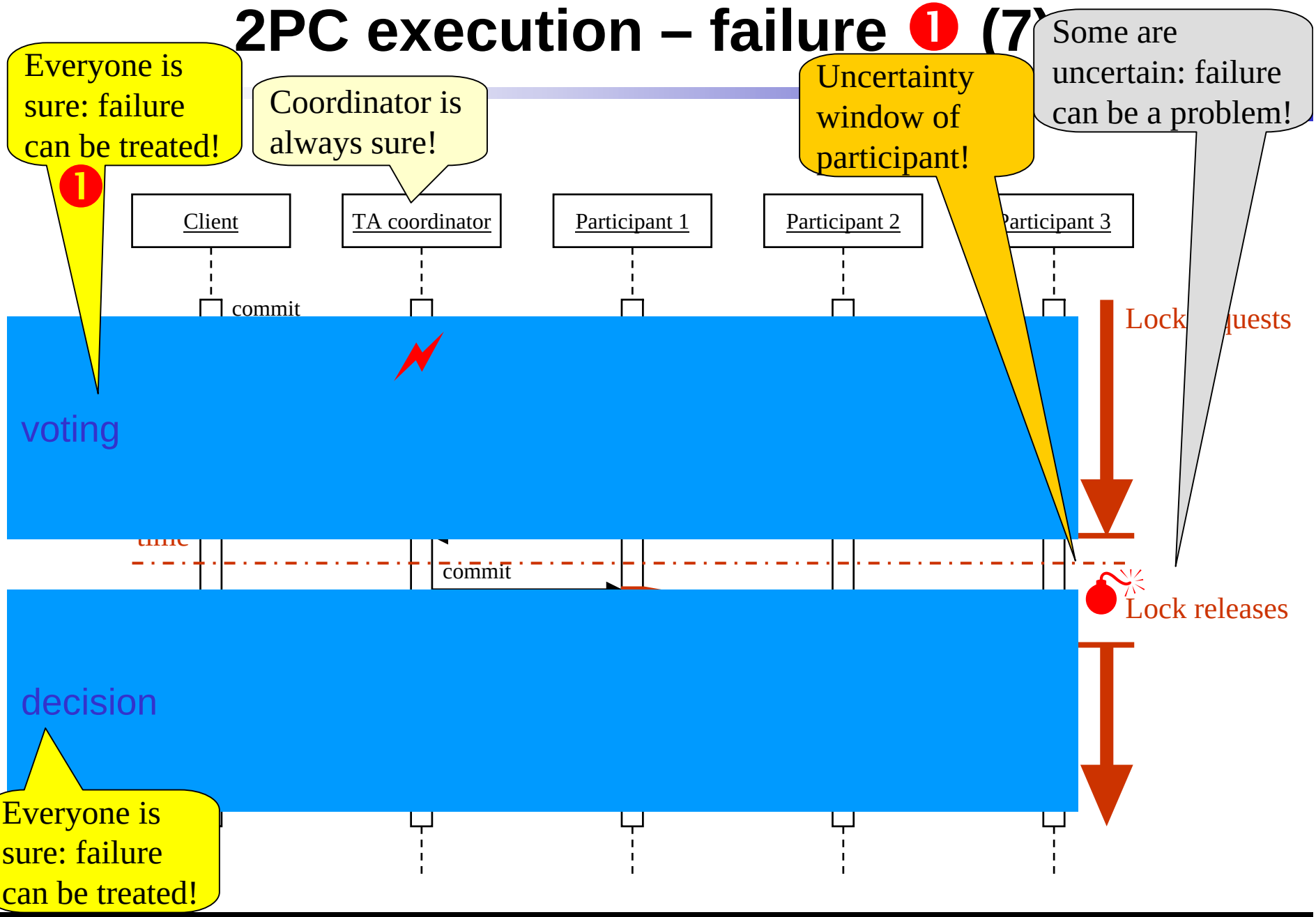
2PC execution – failure ❶ (5)



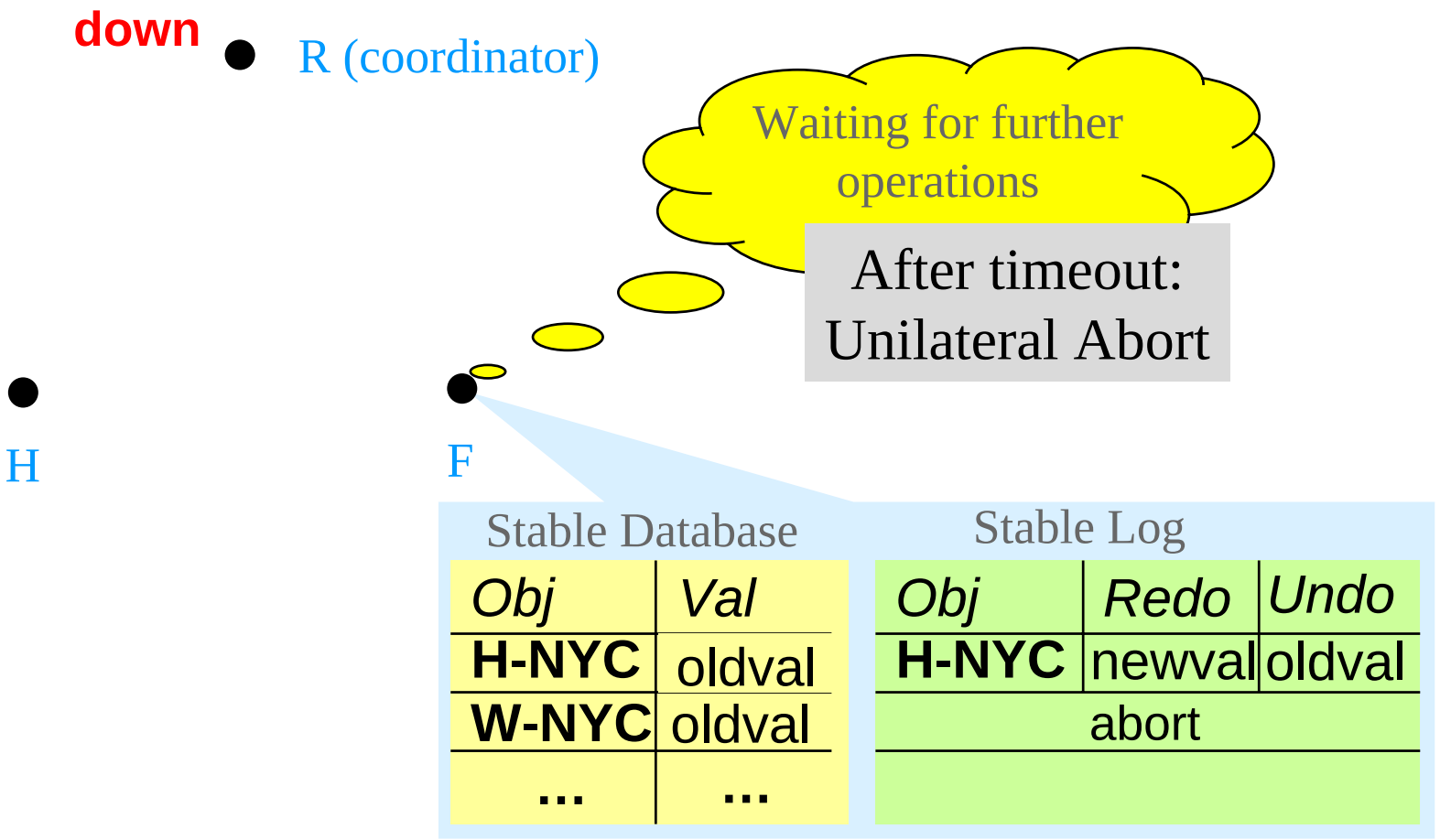
2PC execution – failure ❶ (6)



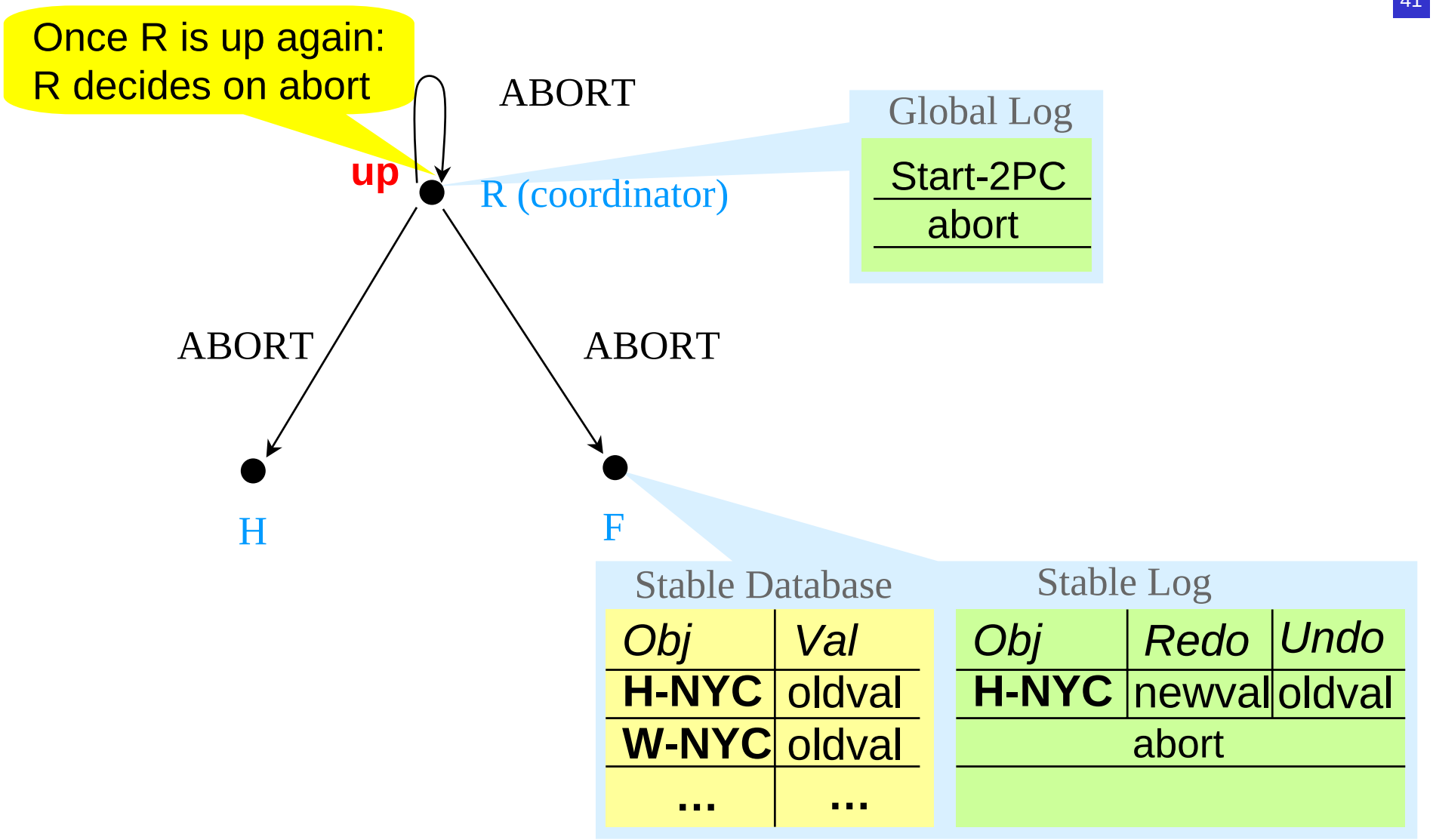
2PC execution – failure ❶ (7)



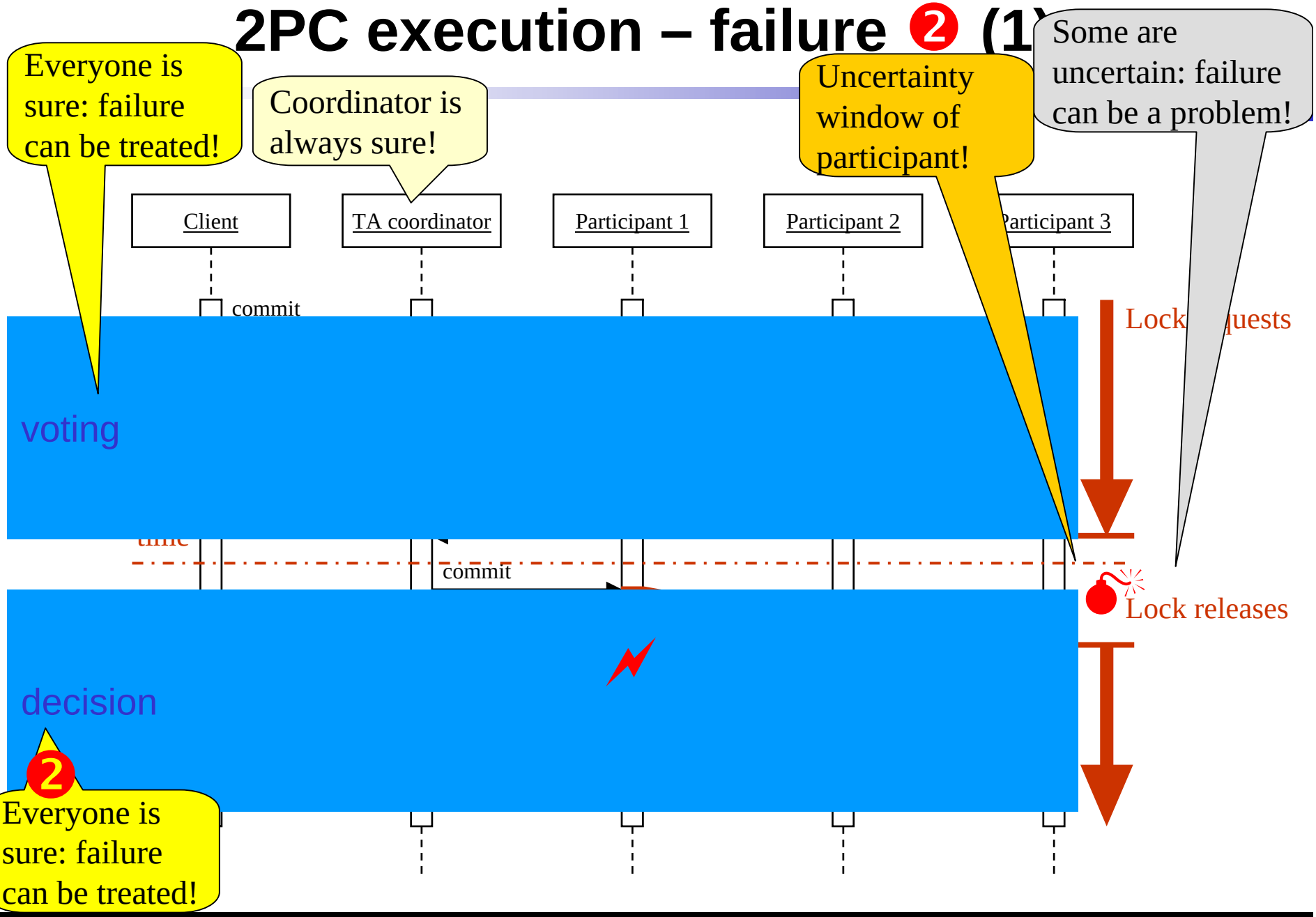
2PC execution – failure ❶ (8)



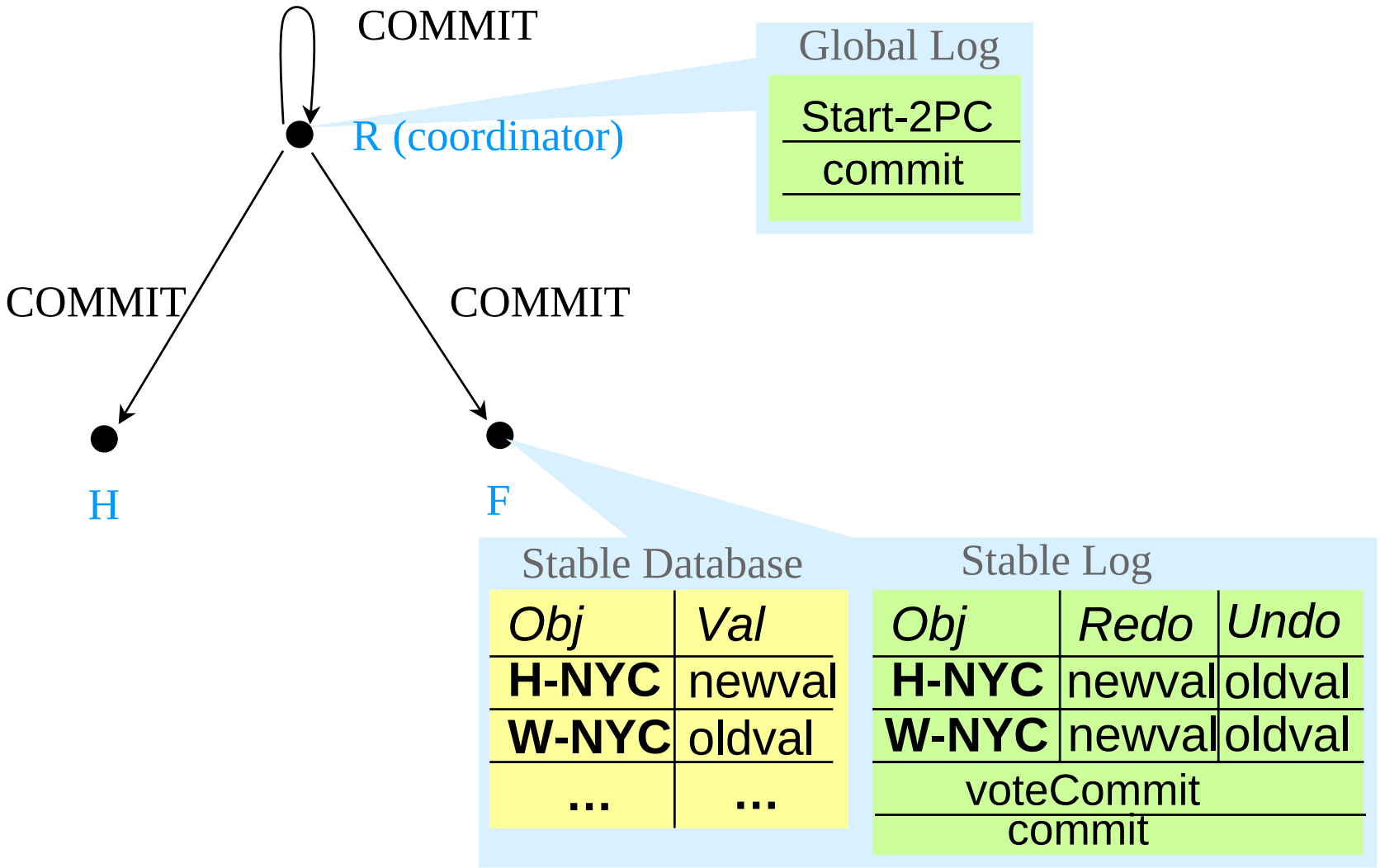
2PC execution – failure ❶ (9)



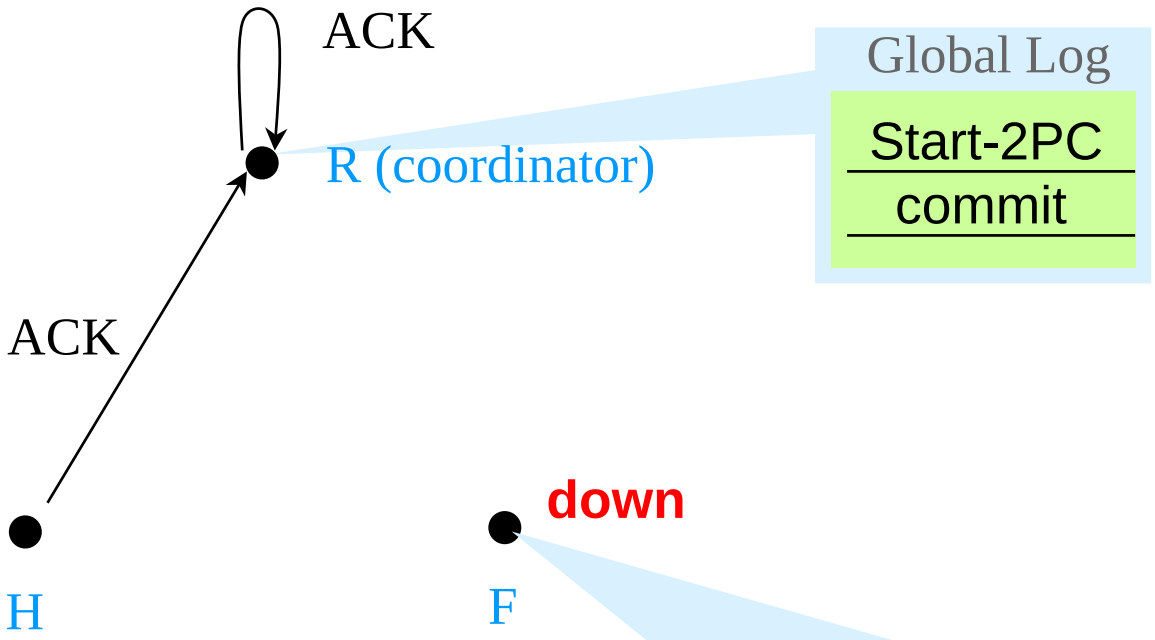
2PC execution – failure ❷ (1)



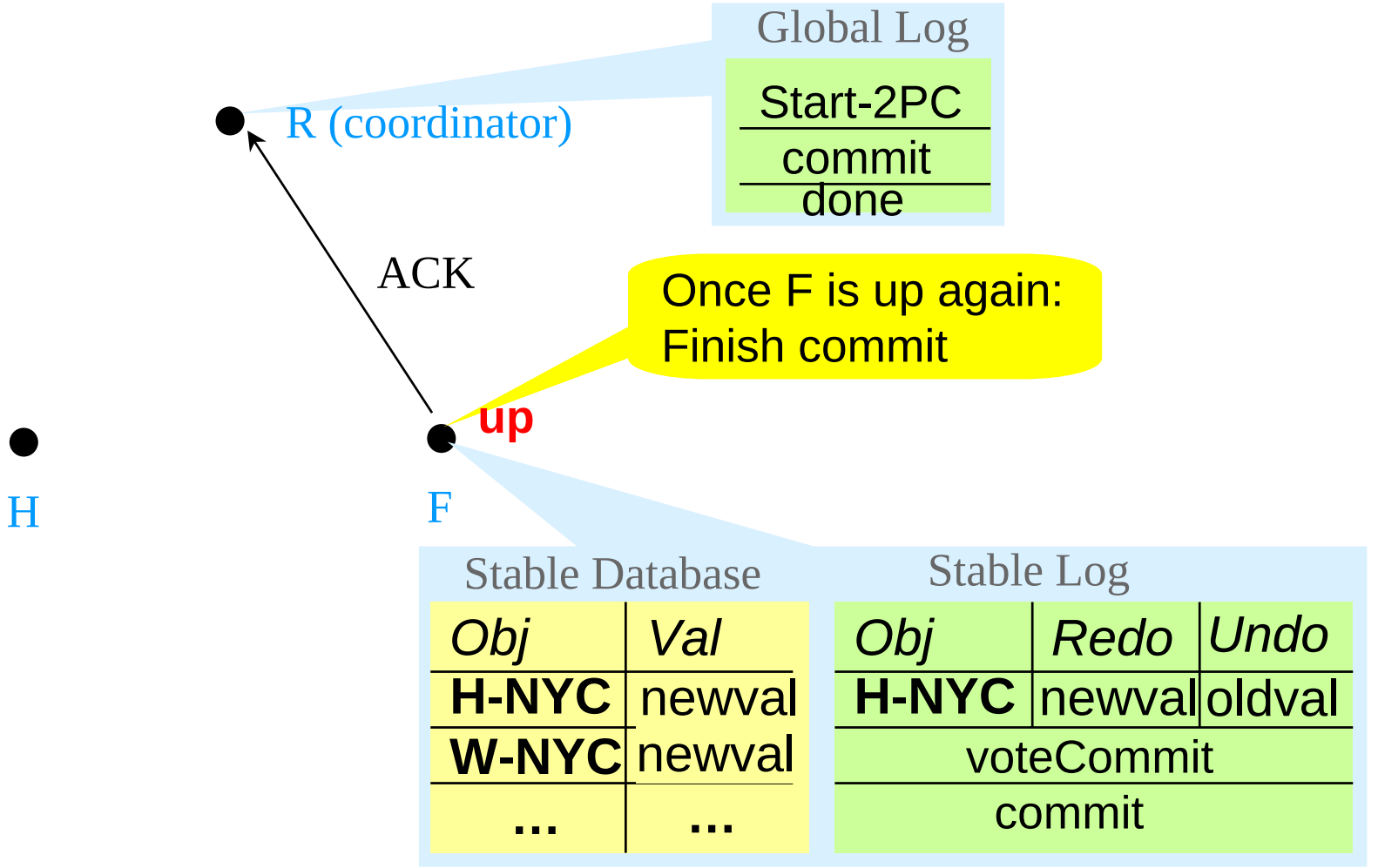
2PC execution – failure ❷ (2)



2PC execution – failure ❷ (3)



2PC execution – failure ❷ (4)



2PC execution – site down

- While a site fails its TM cannot decide ⇒ site remains blocked.
- Sites waiting for responses from sites that are down:
  - To avoid waiting forever, set **timer**.
  - Coordinator: After timeout, if it waits for *voteCommit*, signal *abort* to all up-sites and later to the down-site. No timer set for *ack*, last *ack* causes *done* to be written to log.
  - Participant: After timeout, if waiting for *prepare*, decide on abort.

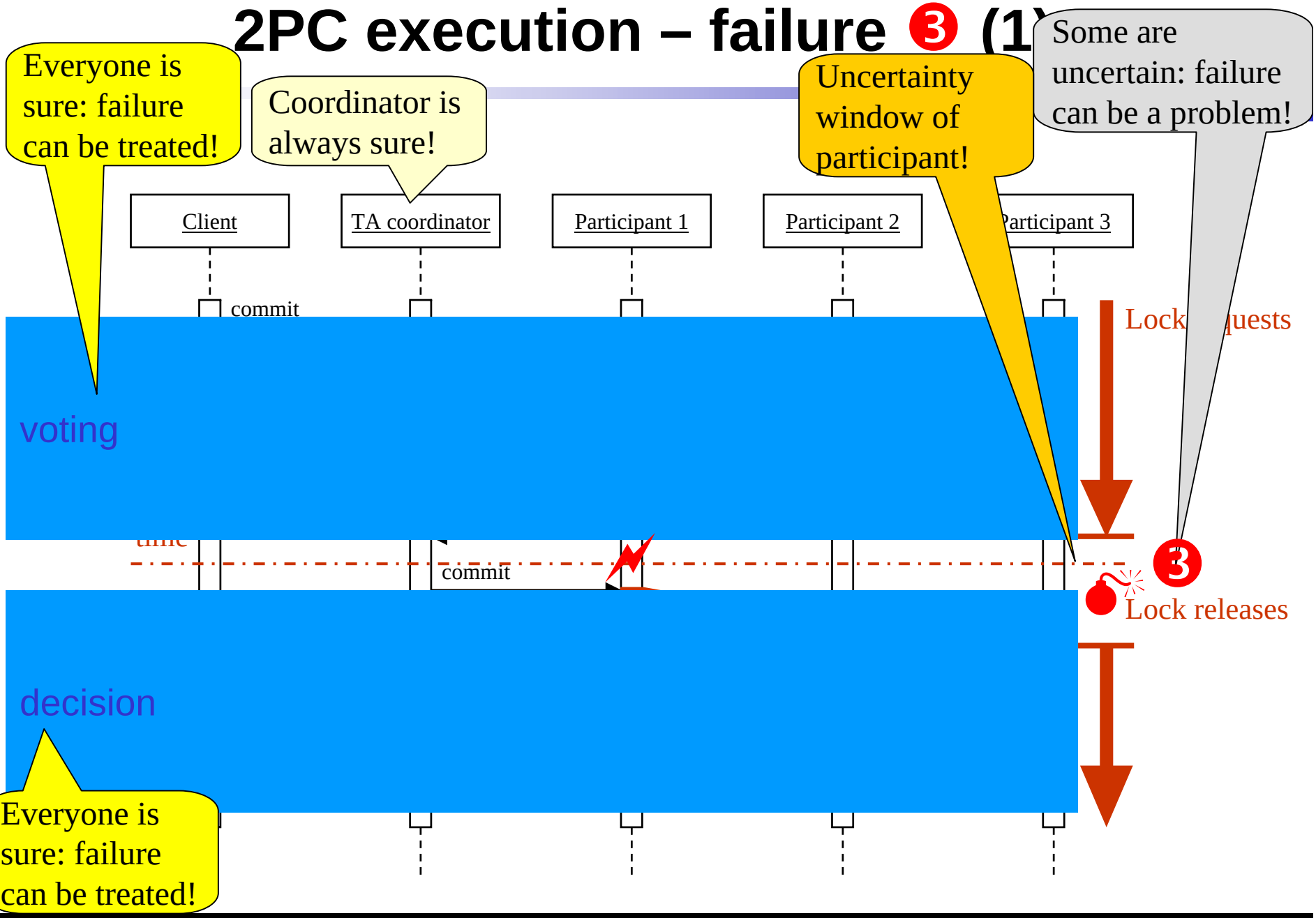
2PC execution – site recovery

Once site *S* has been reconnected or restarted:

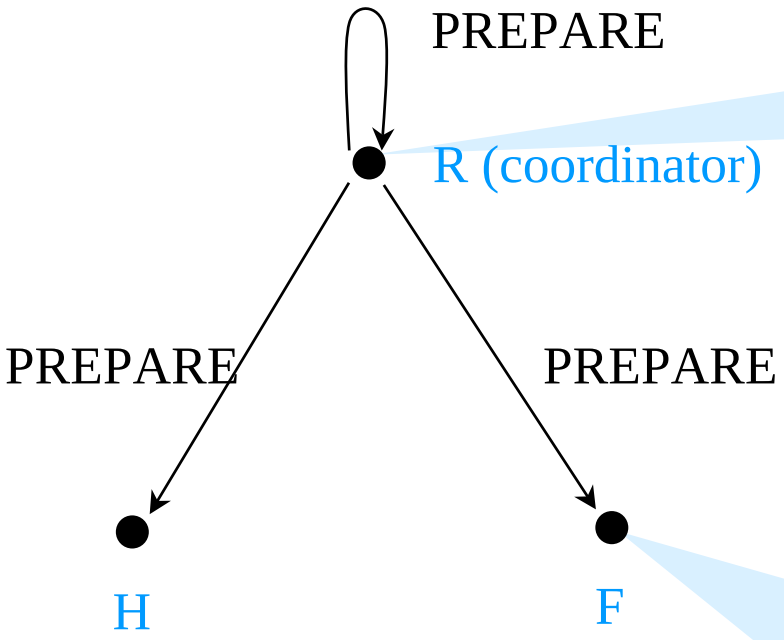
- DTL without Start-2PC for TA *t*: *S* was participant in *t*.
  - DTL contains commit or abort: Decision was taken. Repeat acknowledgement to coordinator or wait for reminder. ❷
  - DTL contains **no** voteCommit: Decide on abort and inform coordinator of the decision. ❶
- DTL with Start-2PC for TA *t*: *S* was coordinator of *t*.
  - DTL contains **only** Start-2PC: No decision was taken yet. Coordinator decides on abort and sends the decision. ❶
  - DTL contains Start-2PC **and** commit or abort **but not** done: Decision was taken but not necessarily sent to all: Retransmit the decision. ❷
  - DTL contains Start-2PC, commit or abort **and** done: Decision was taken and sent to all: Nothing left to do. ❷

2PC Termination problems

2PC execution – failure ❸ (1)



2PC execution – failure ❸ (2)



Global Log

Start-2PC

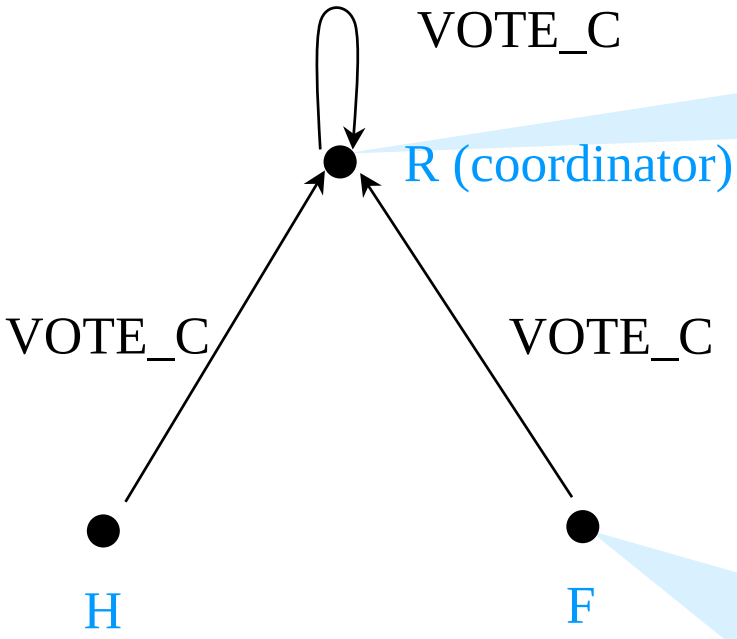
Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval

2PC execution – failure ❸ (3)



Global Log

Start-2PC

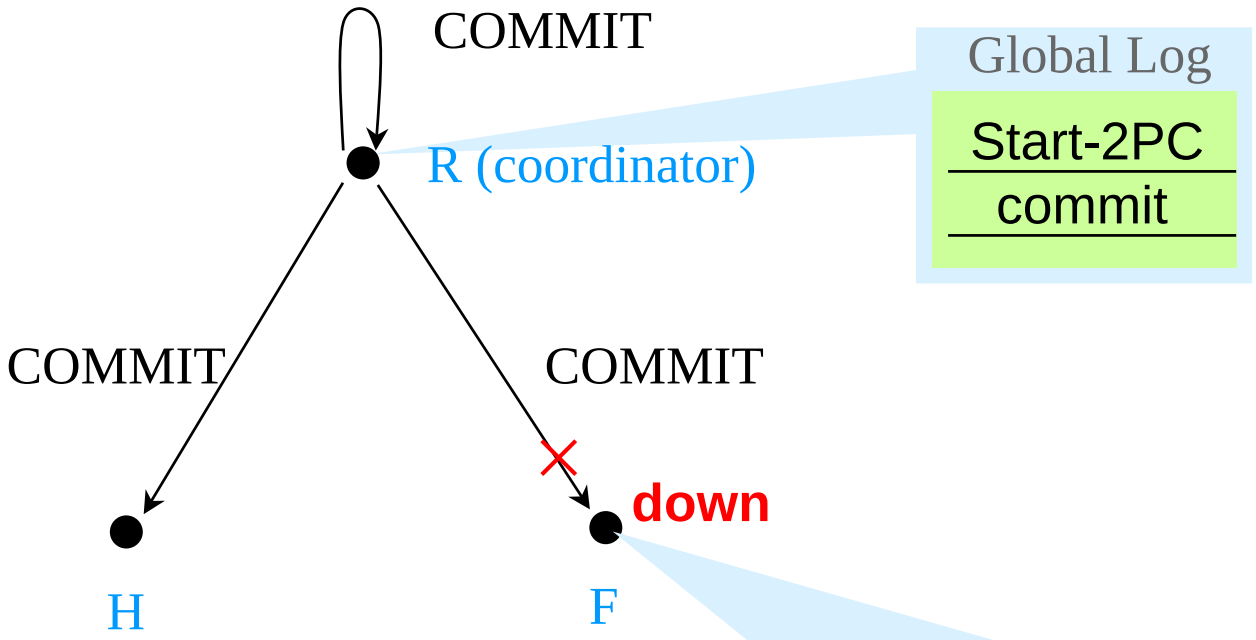
Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval
W-NYC	newval	oldval
voteCommit		

2PC execution – failure ❸ (4)



Global Log

Start-2PC
commit

Stable Database

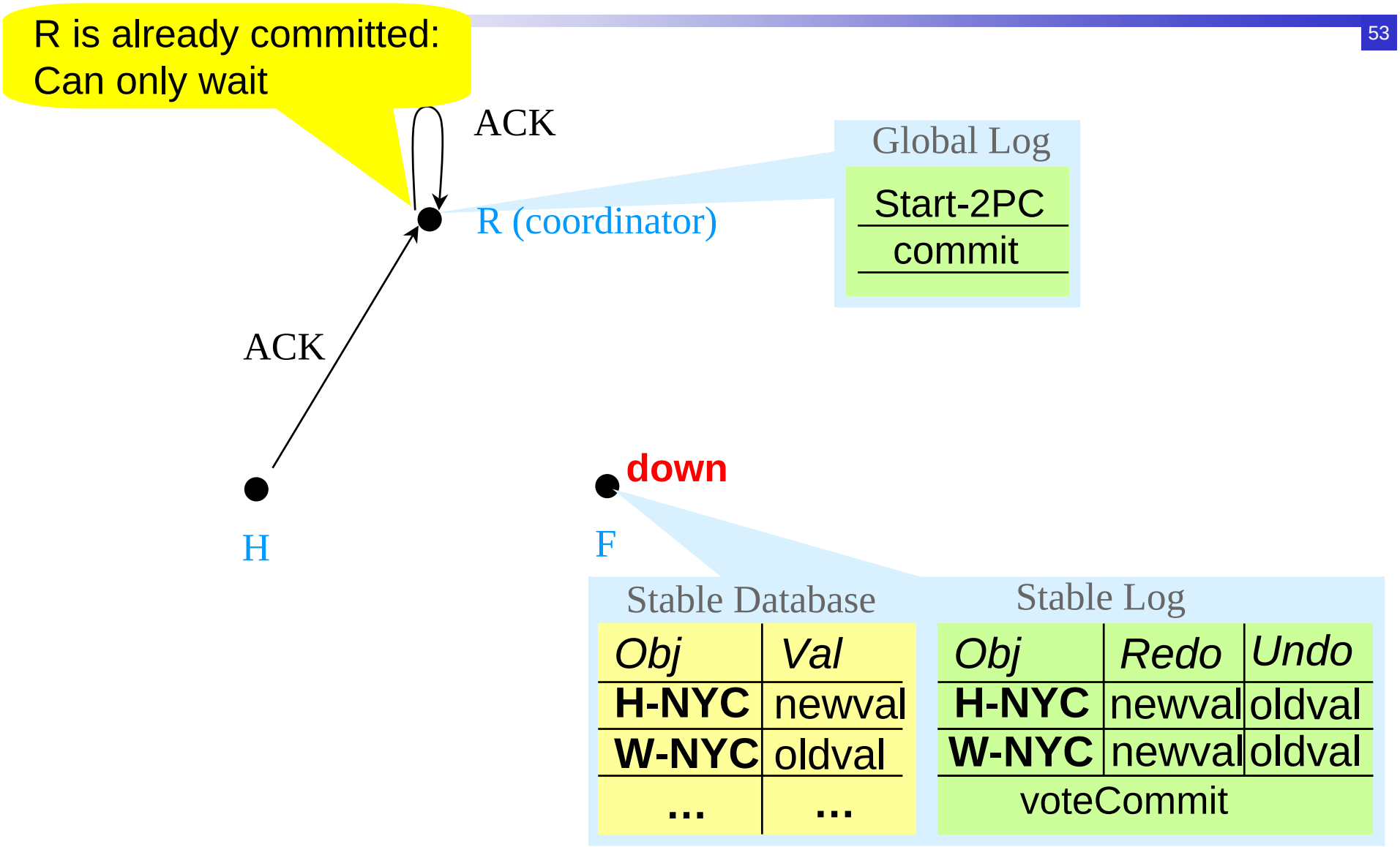
Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval
W-NYC	newval	oldval
voteCommit		

2PC execution – failure ❸ (5)

R is already committed:
Can only wait



Global Log

Start-2PC
commit

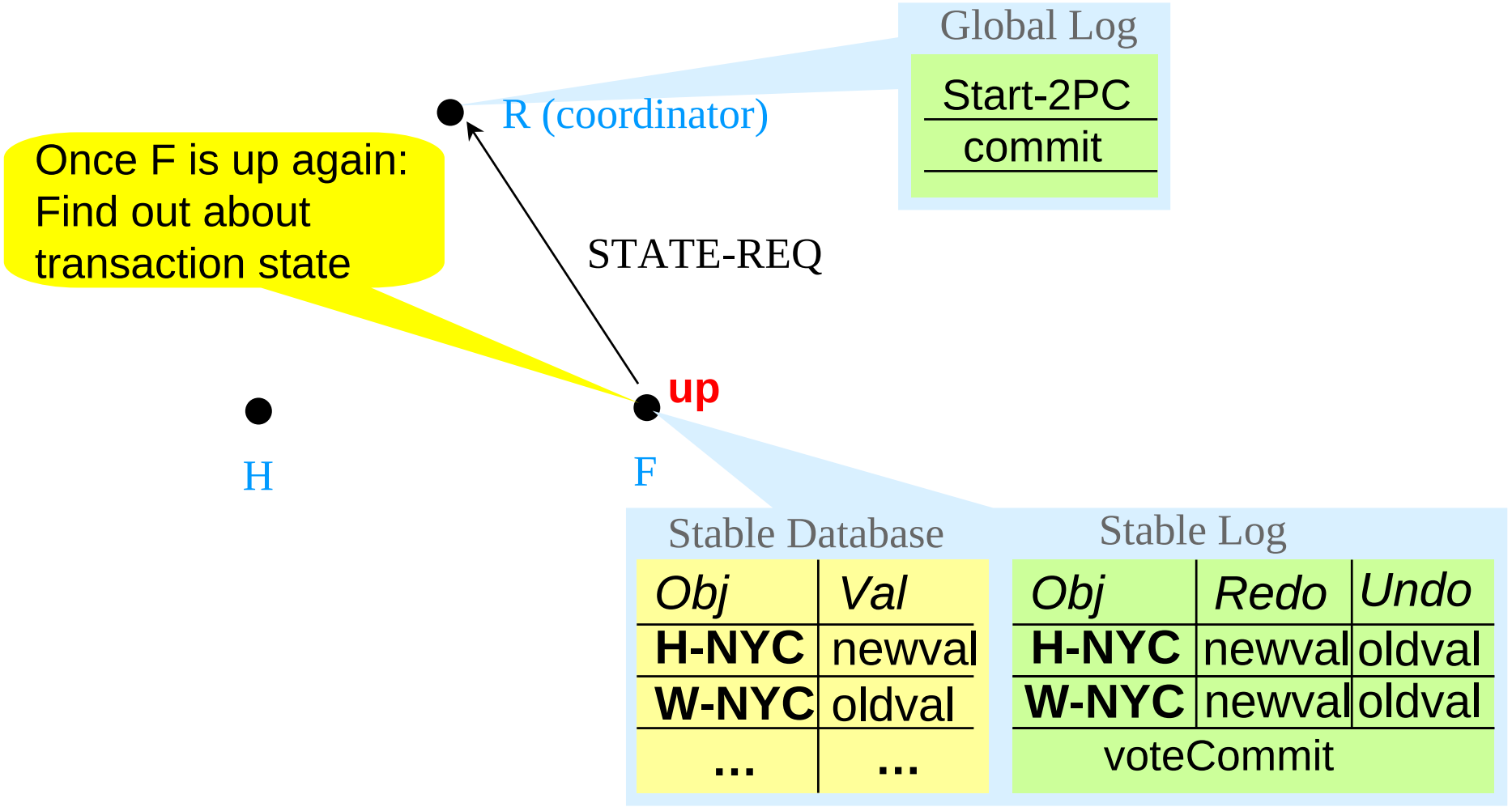
Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

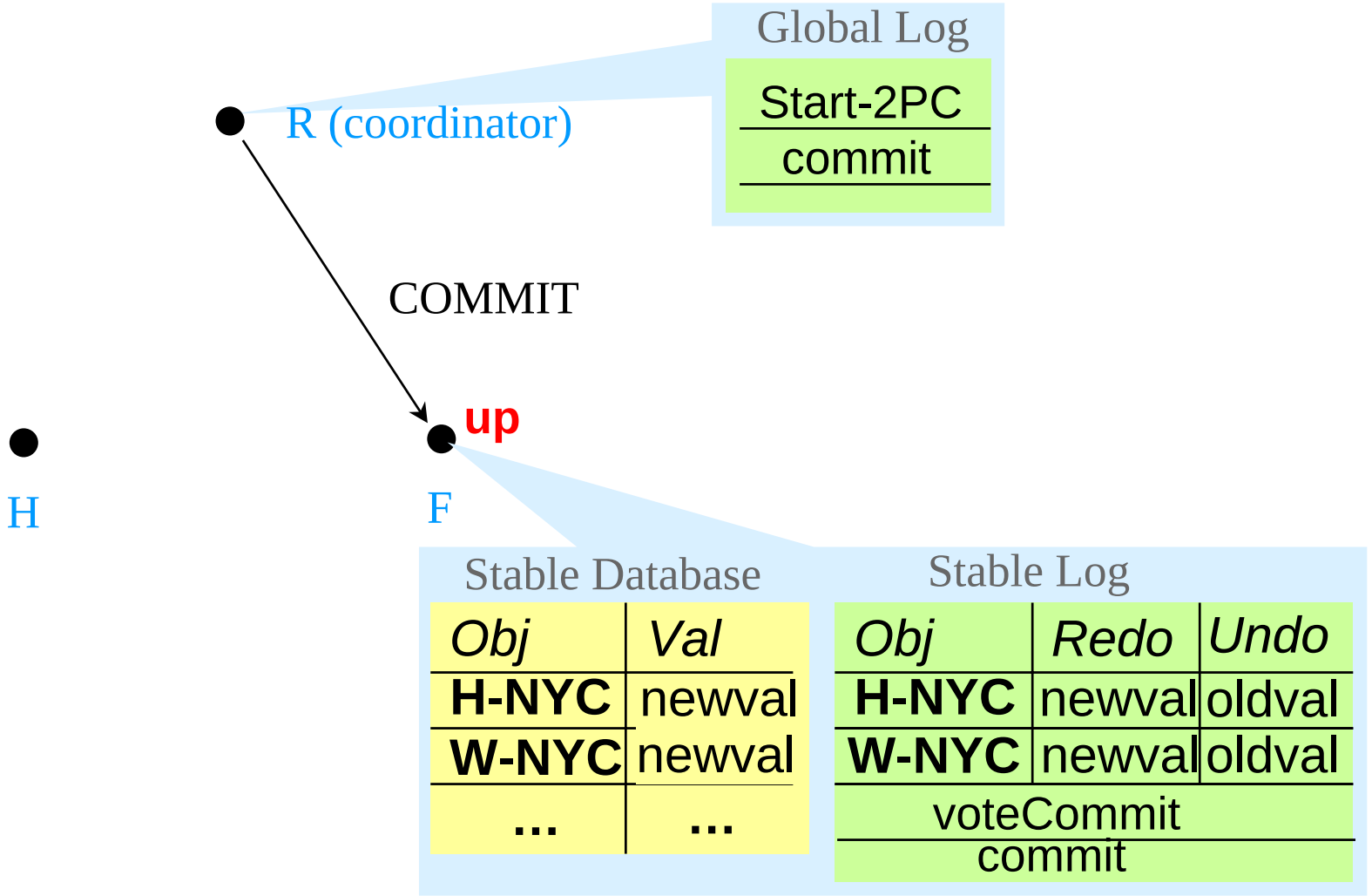
Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval
W-NYC	newval	oldval
voteCommit		

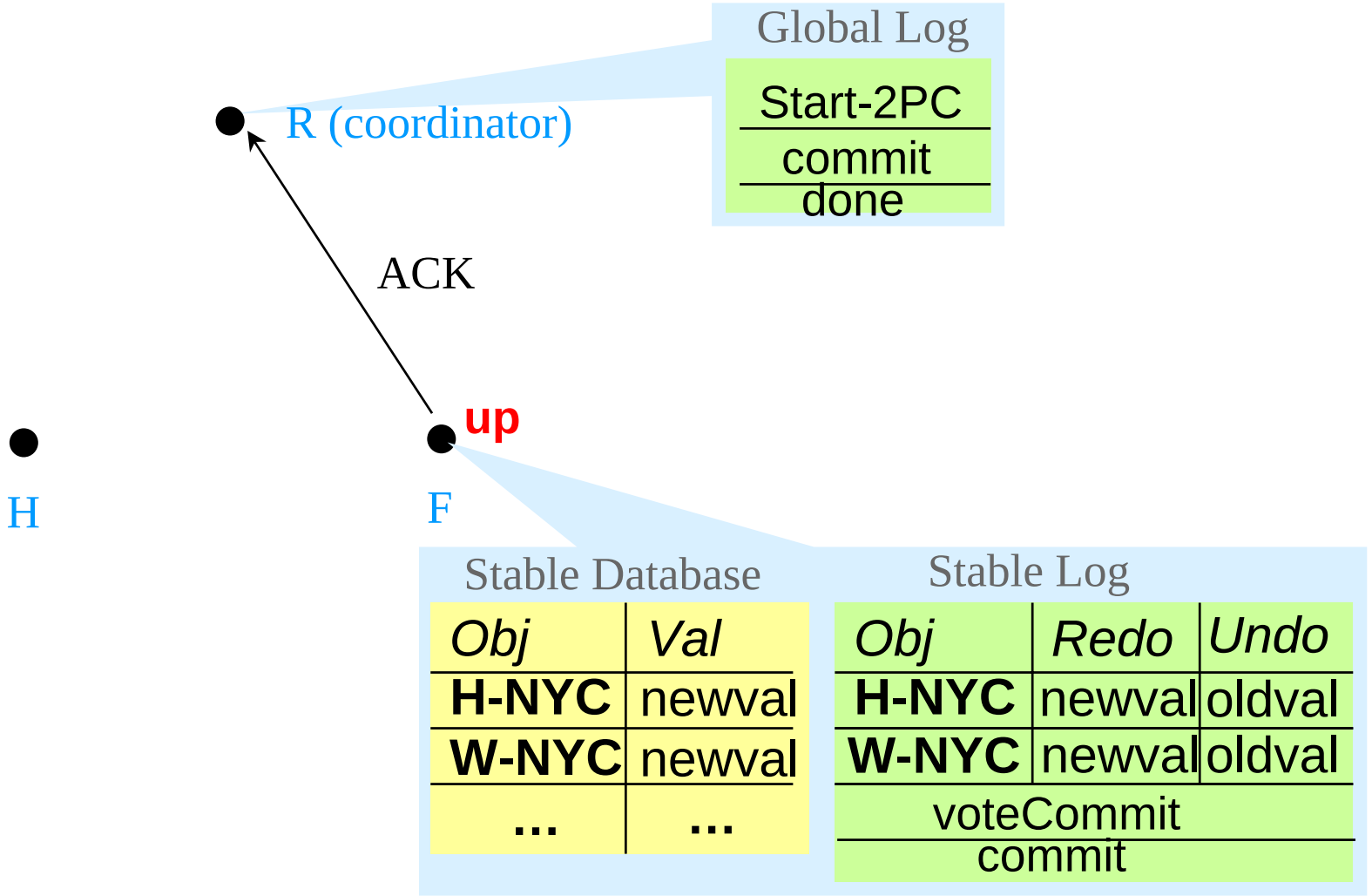
2PC execution – failure ❸ (6)



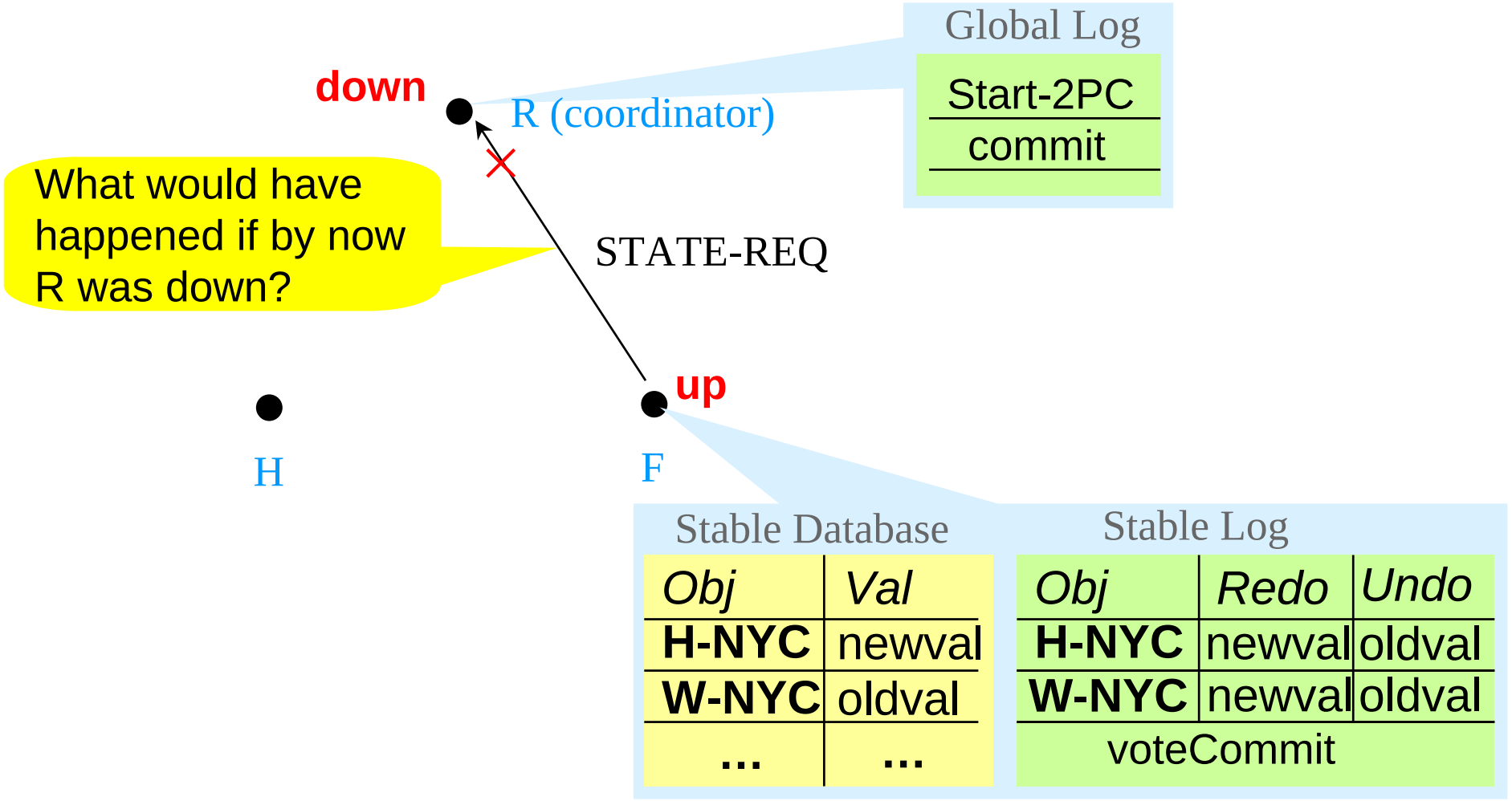
2PC execution – failure ❸ (7)



2PC execution – failure ❸ (8)



2PC termination problem ❸



2PC execution – site recovery (cont'd)

Once site *S* has been reconnected or restarted:

- DTL without Start-2PC for TA *t*: *S* was participant in *t*.
  - DTL contains commit or abort: Decision was taken. Repeat acknowledgement to coordinator or wait for reminder. ❷
  - DTL contains **no** voteCommit: Decide on abort and inform coordinator of the decision. ❶
  - DTL contains voteCommit **but no** commit or abort: Participant is uncertain. ❸

Ask others what the decision was:

- Try coordinator: It is always sure.
- Otherwise try to reach participants that are sure. If none or only uncertain nodes can be reached: Protocol is blocked!

Termination during uncertainty window (1)

S asks others what the decision was:

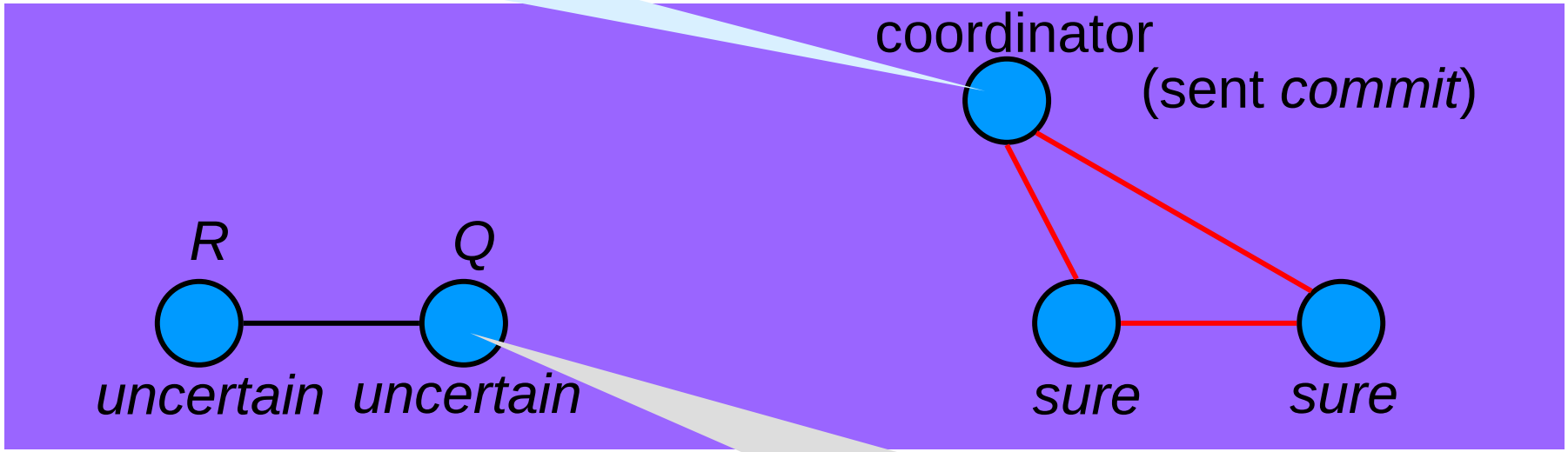
- Try coordinator: It is always sure.
- Otherwise try to reach participants that are sure. If none or only uncertain nodes can be reached: Protocol is blocked!

- Other site *Q* decided on commit/abort due to coordinator message: pass decision on to *S*.
- *Q* voted abort: pass decision on to *S*, *S* infers the global decision.
- *Q* did not yet vote: *S* decides on unilateral abort.

- *Q* voted *voteCommit* but received no answer and, hence, is uncertain.

Termination during uncertainty window (2)

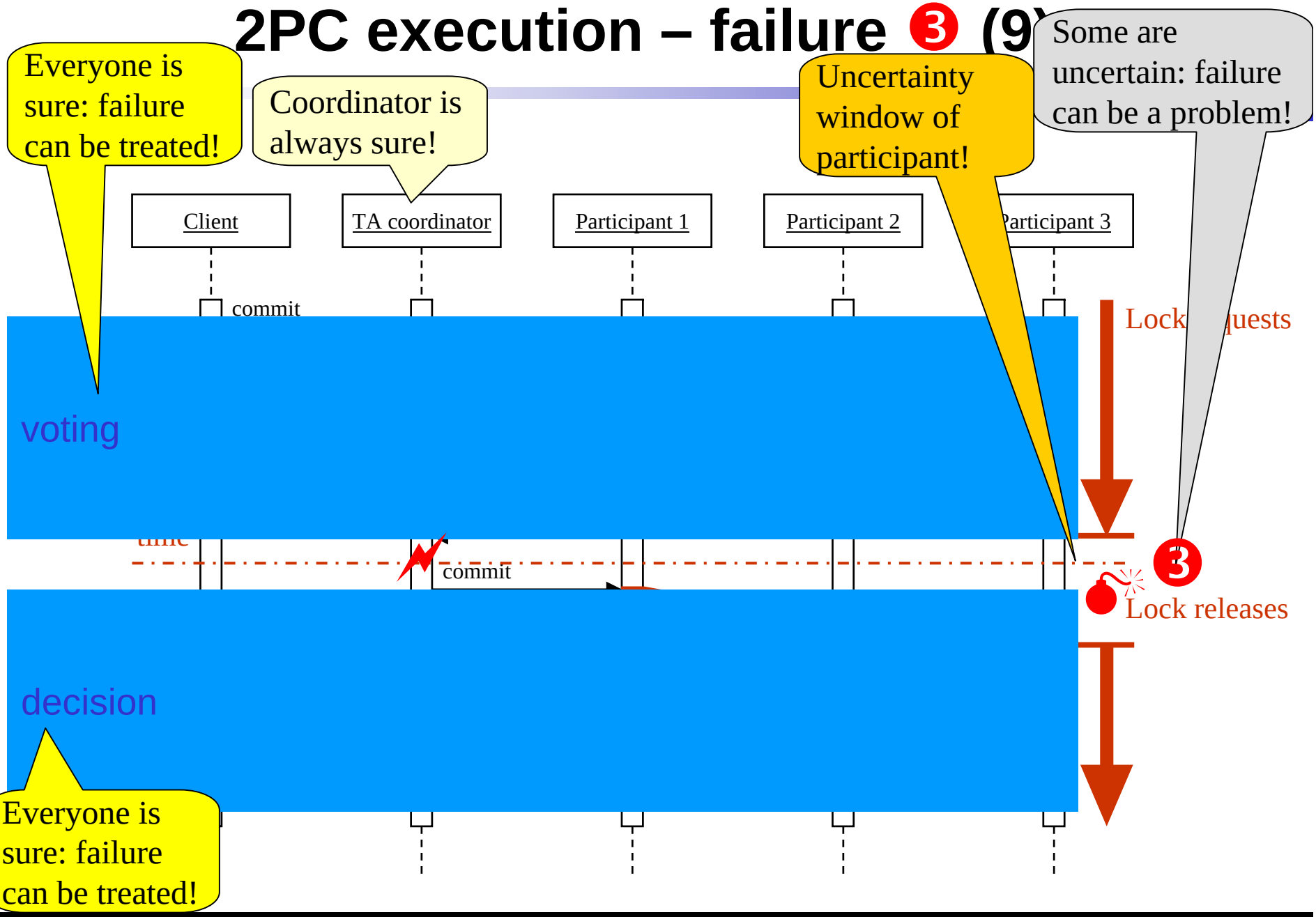
t remains blocked until all *ack*s have been received



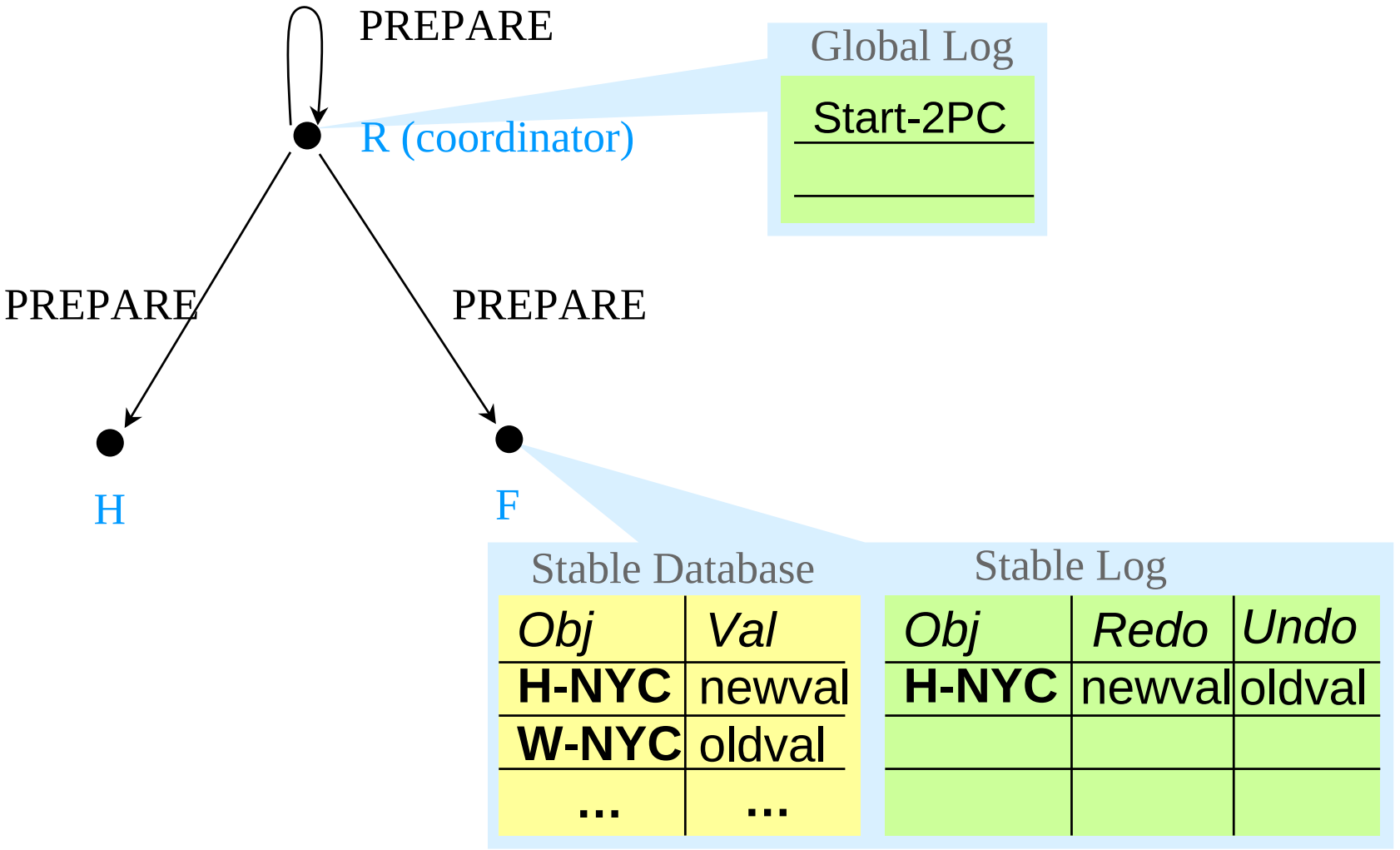
R and Q remain blocked until one of them has been reconnected to the other subnet

Blocking is of unknown duration !

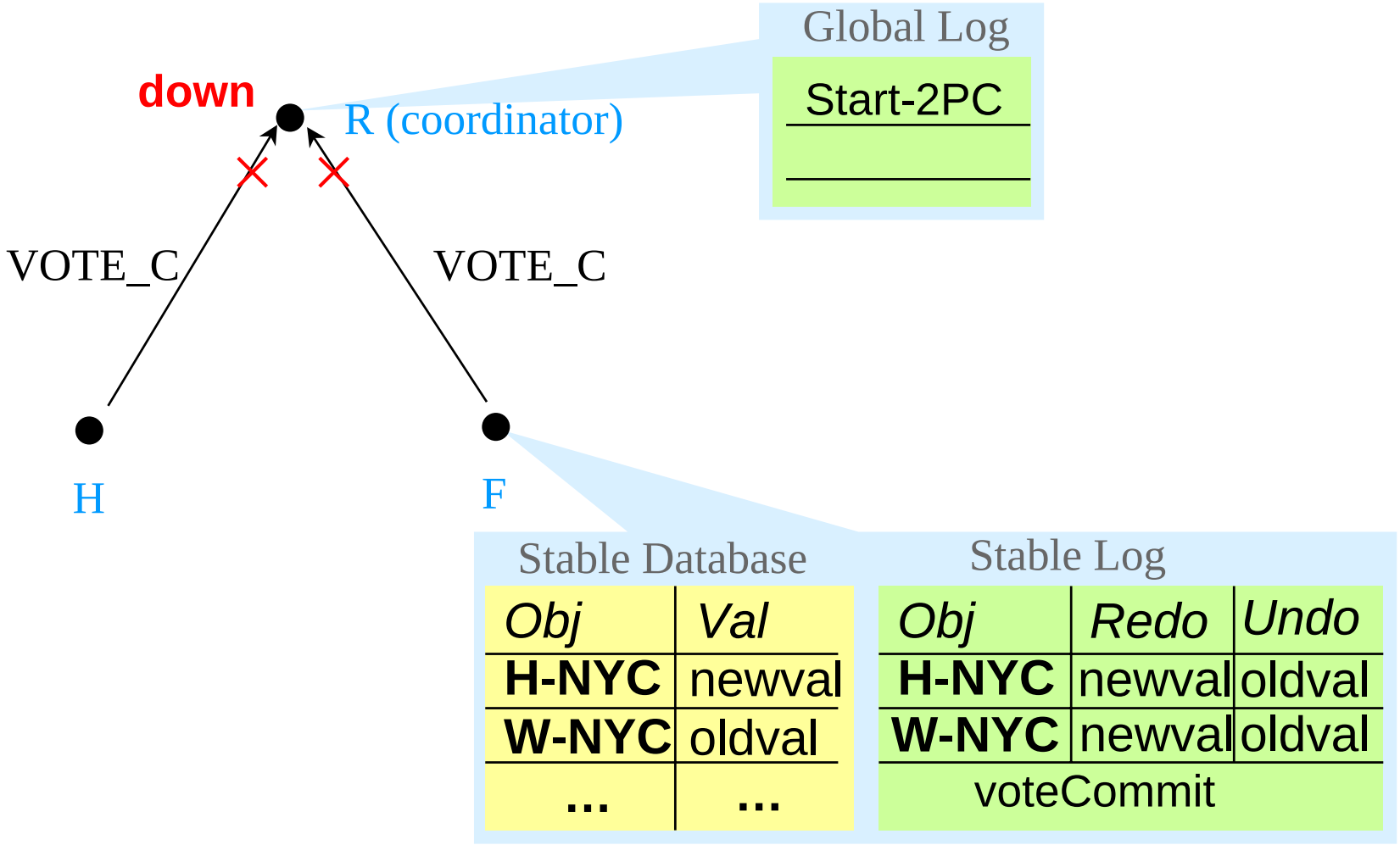
2PC execution – failure ❸ (9)



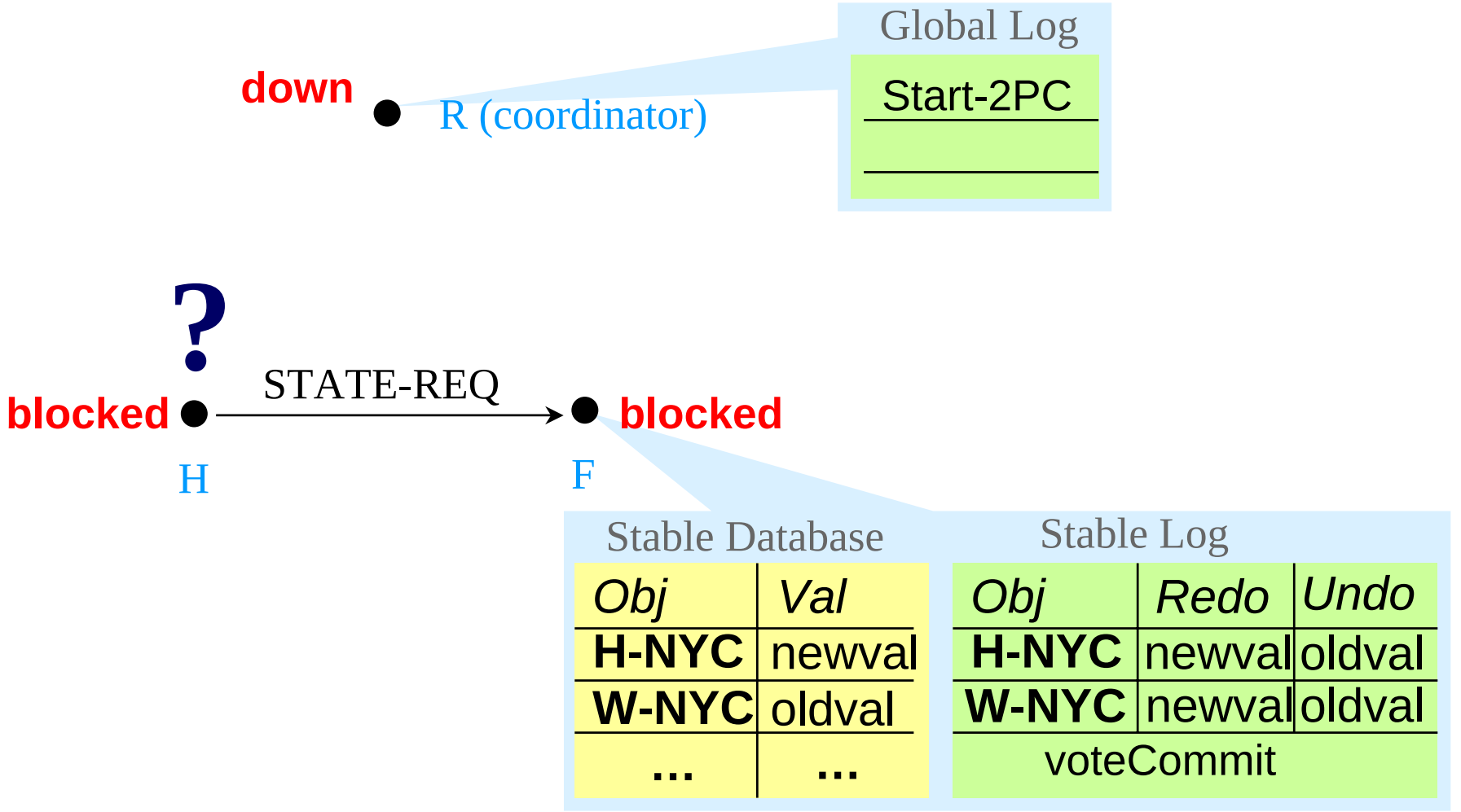
Termination during uncertainty window (3)



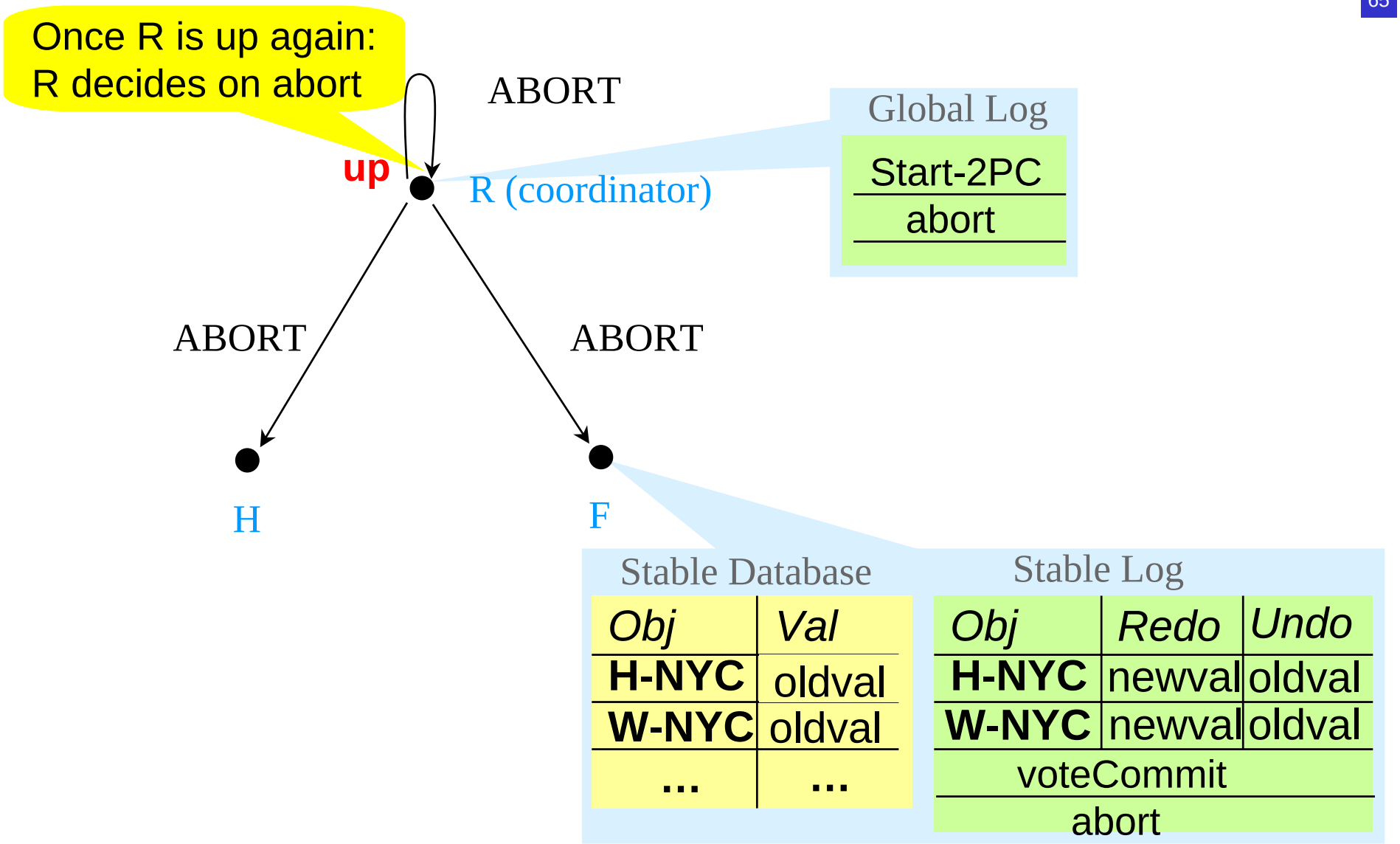
Termination during uncertainty window (4)



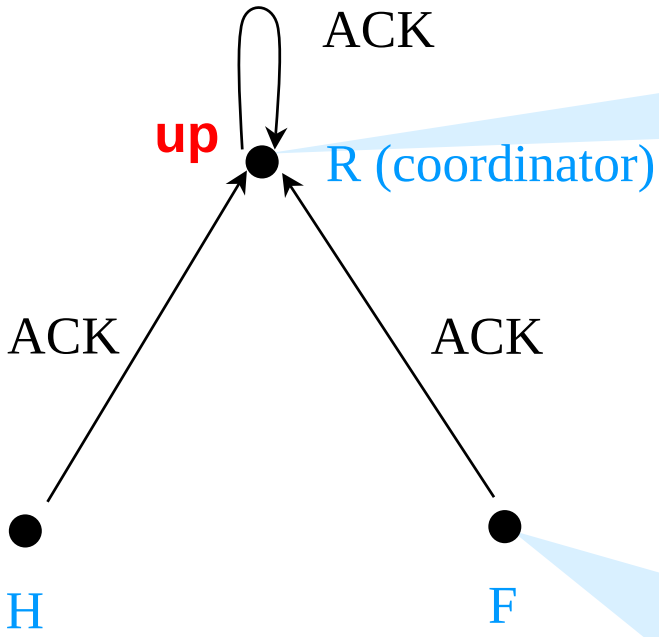
Termination during uncertainty window (5)



Termination during uncertainty window (6)



Termination during uncertainty window (7)



Global Log

Start-2PC
abort
done

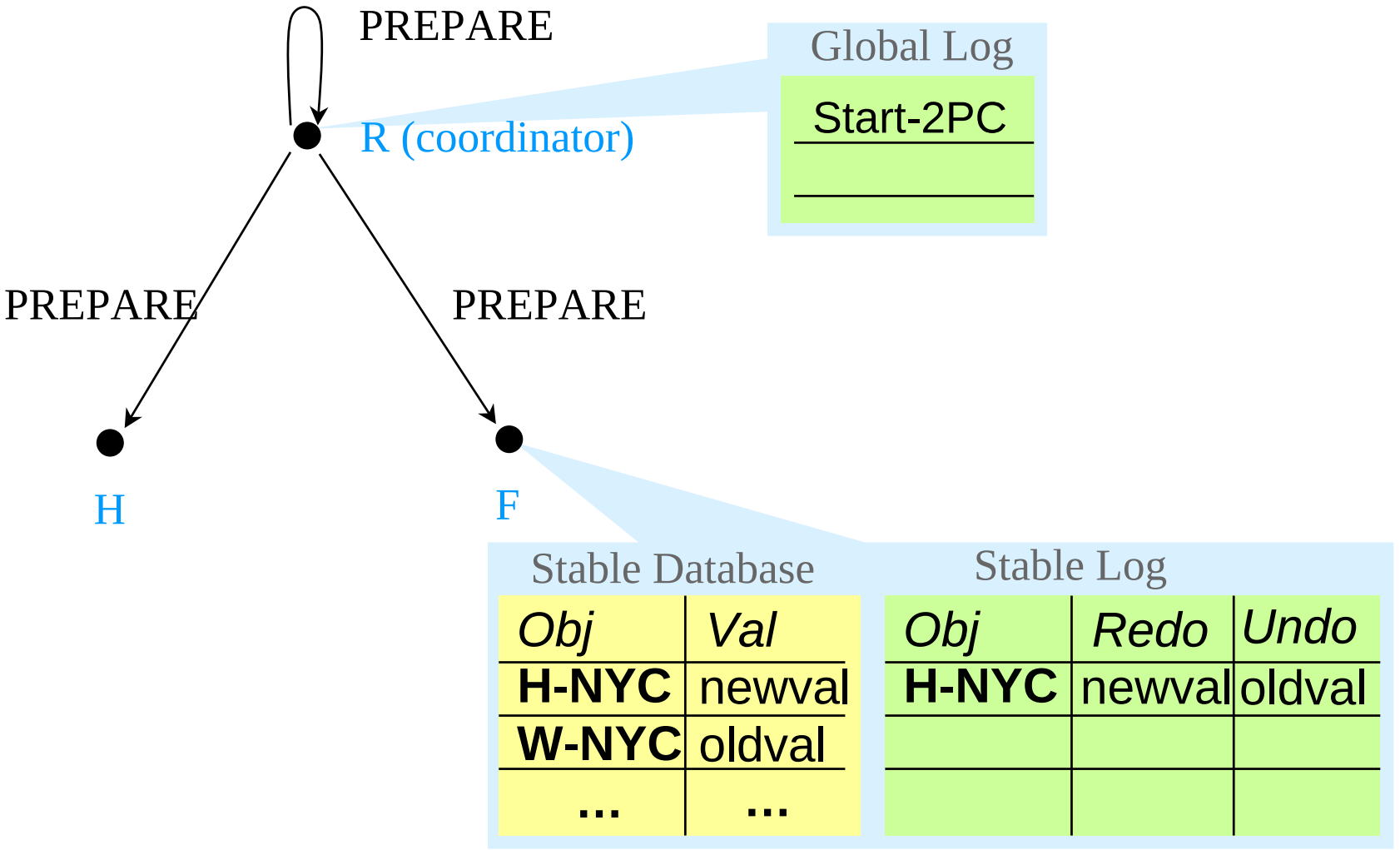
Stable Database

Obj	*Val*
H-NYC	oldval
W-NYC	oldval
...	**...**

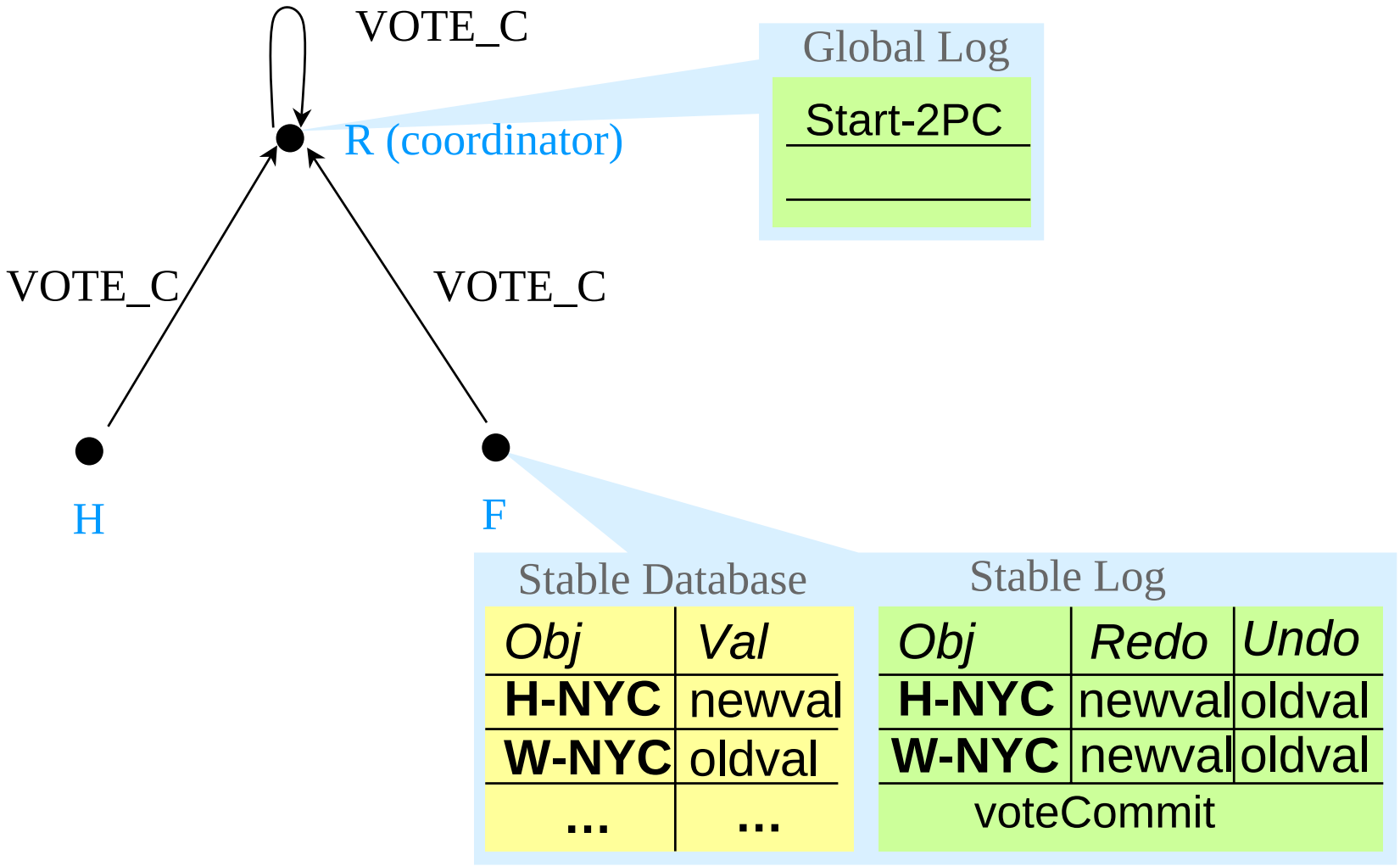
Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval
W-NYC	newval	oldval
voteCommit		
abort		

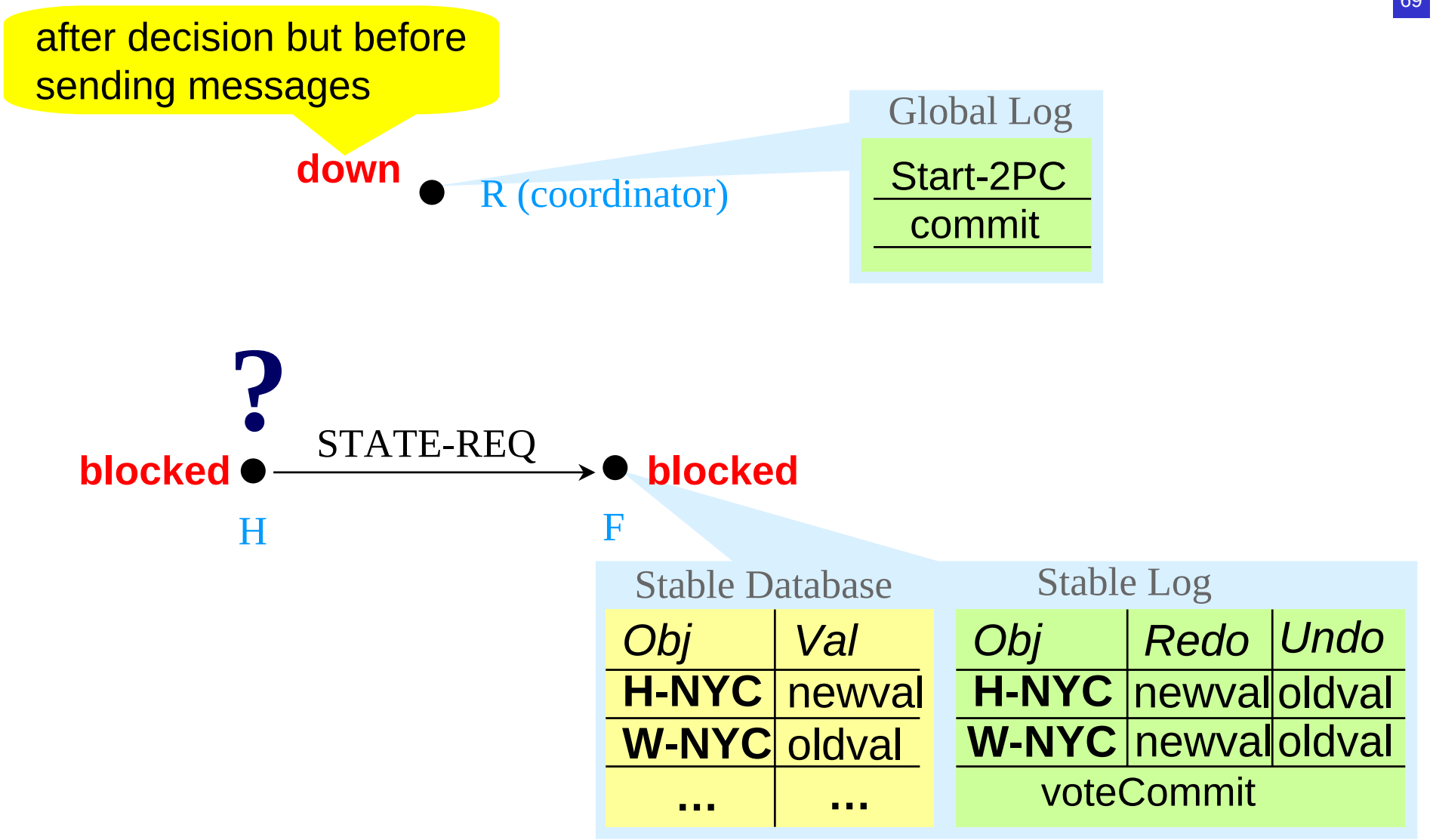
Termination during uncertainty window (8)



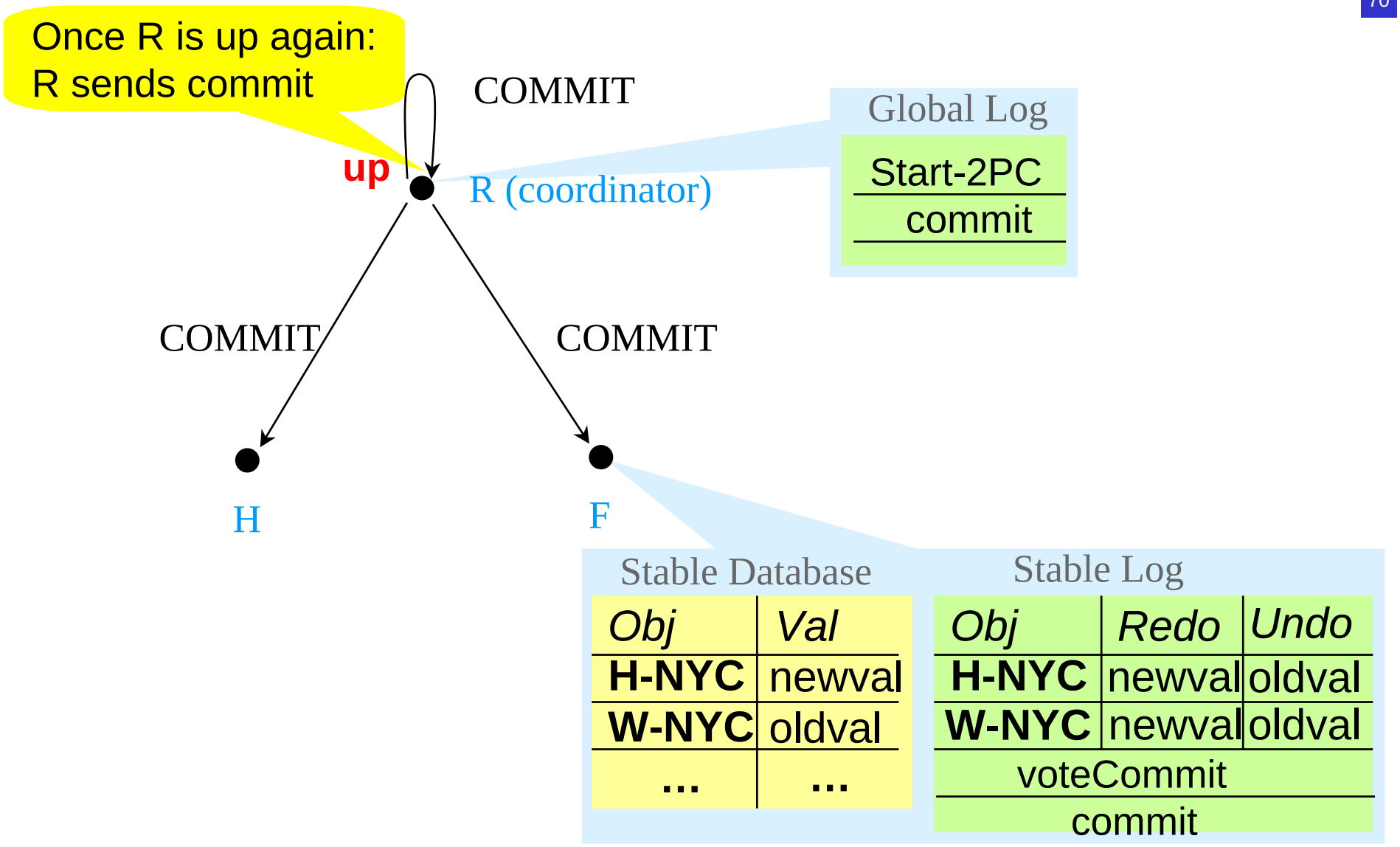
Termination during uncertainty window (9)



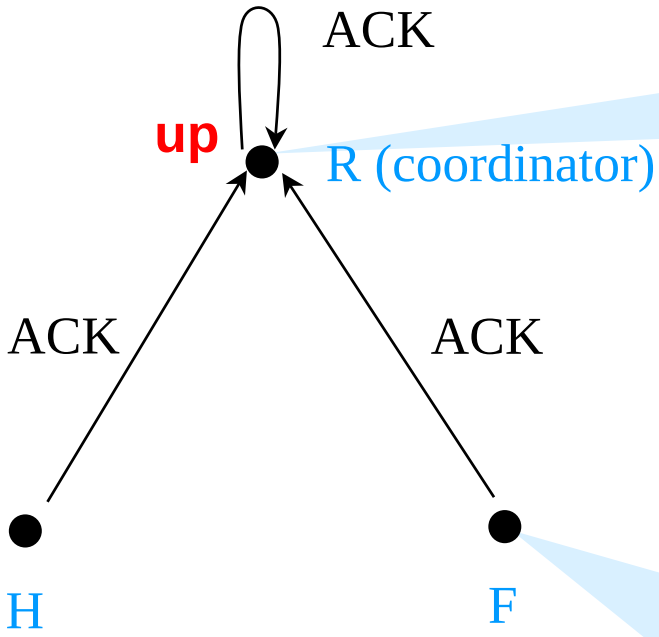
Termination during uncertainty window (10)



Termination during uncertainty window (11)



Termination during uncertainty window (12)



Global Log

Start-2PC
commit
done

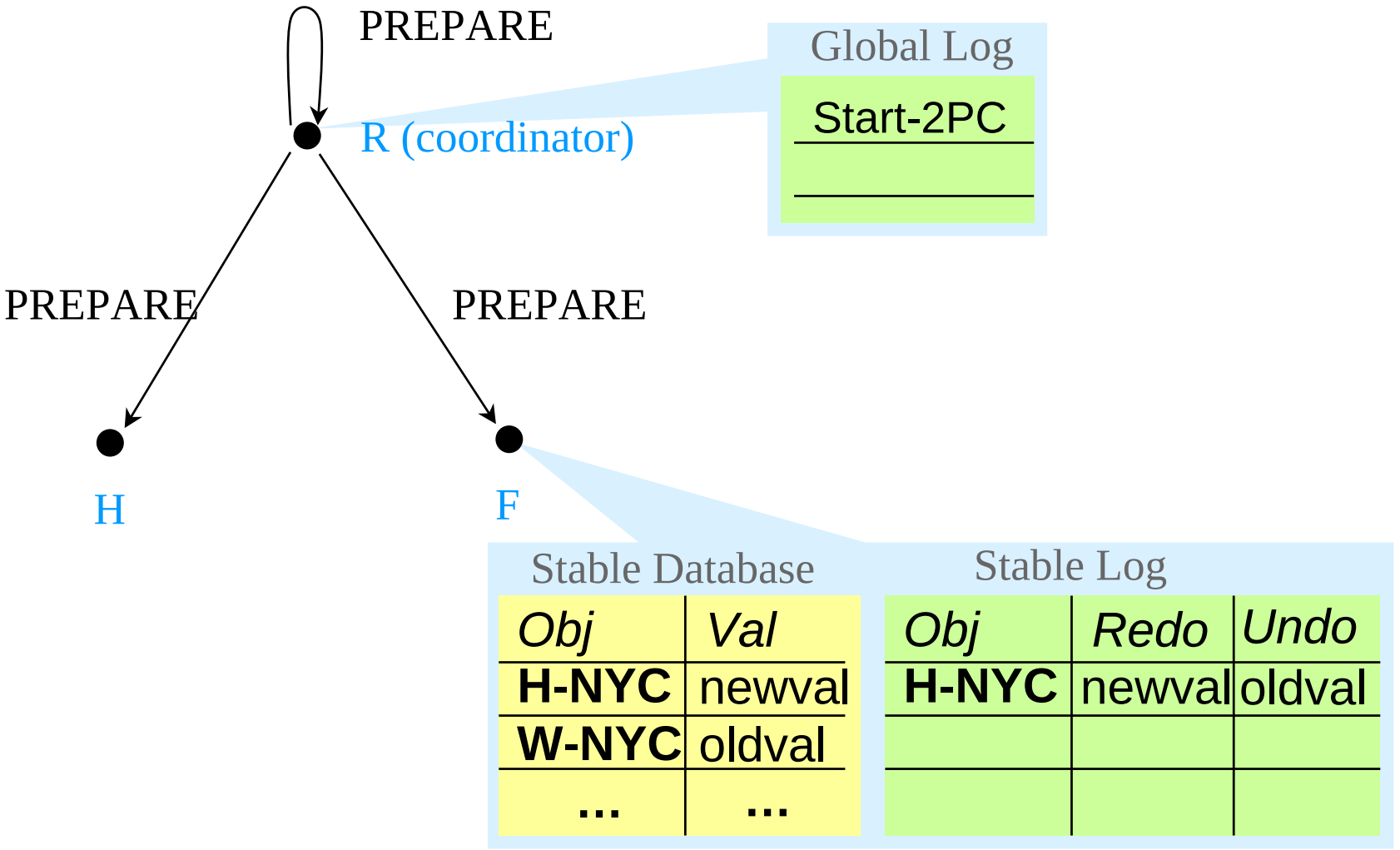
Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

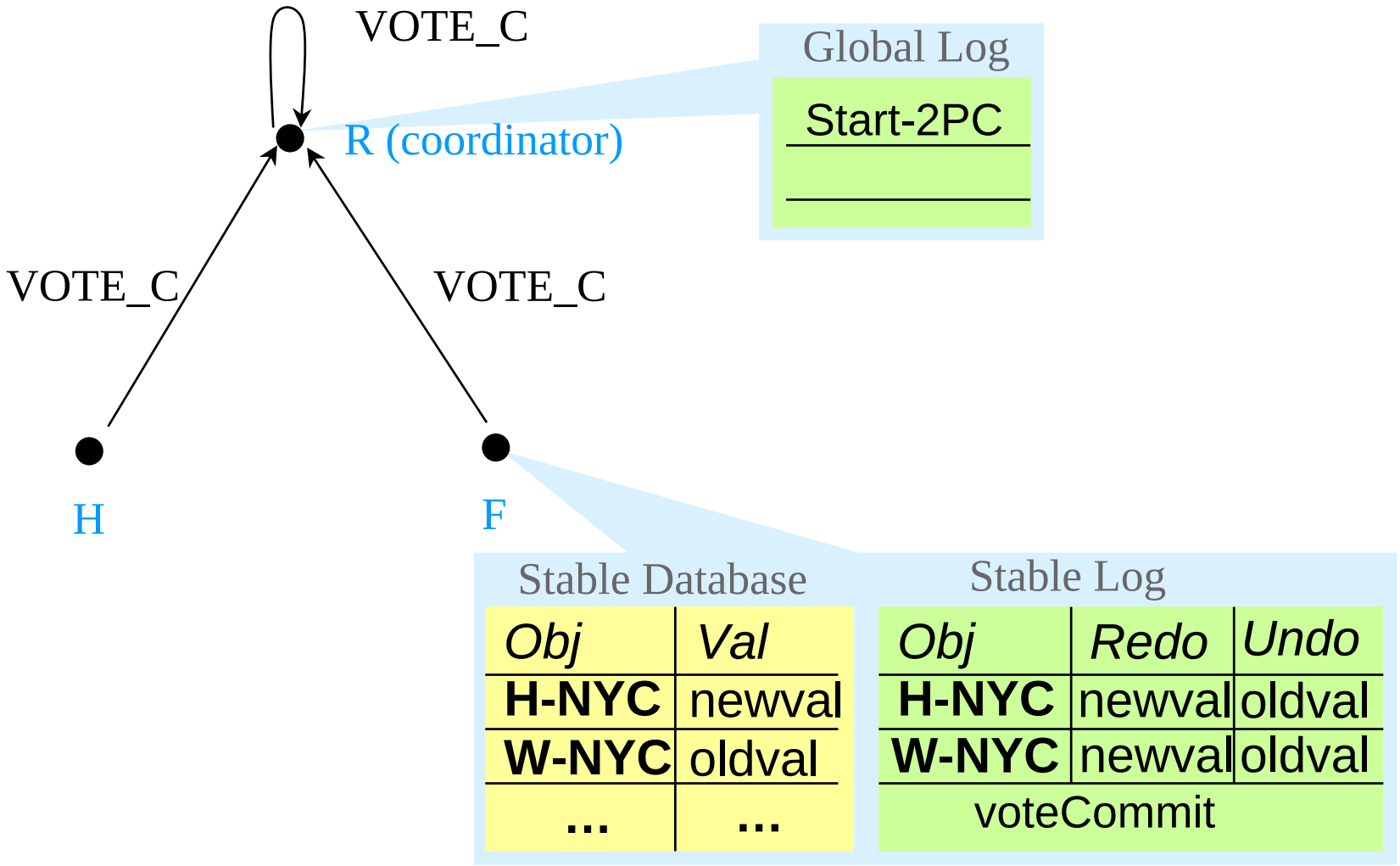
Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval
W-NYC	newval	oldval
voteCommit		
commit		

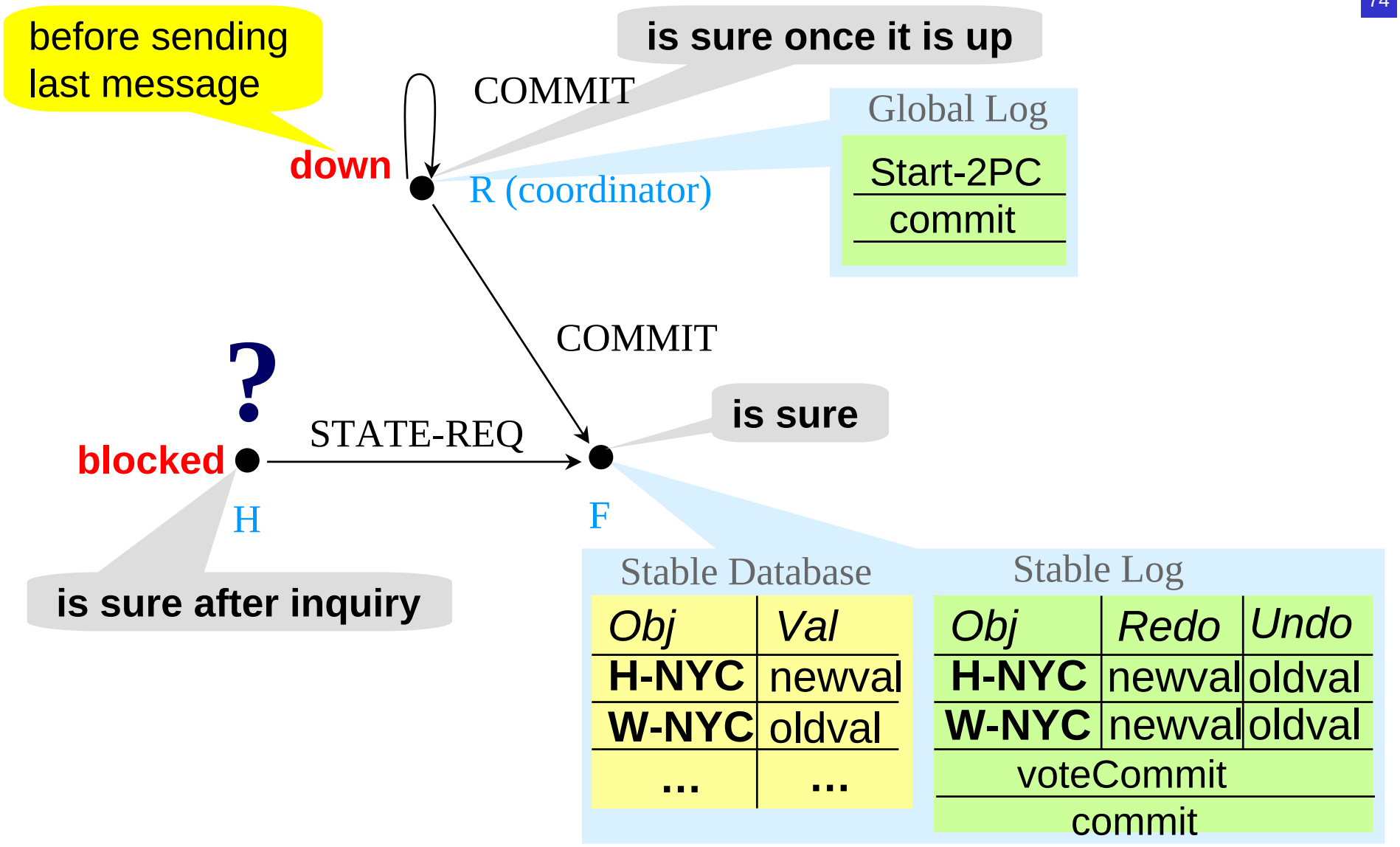
Termination during uncertainty window (13)



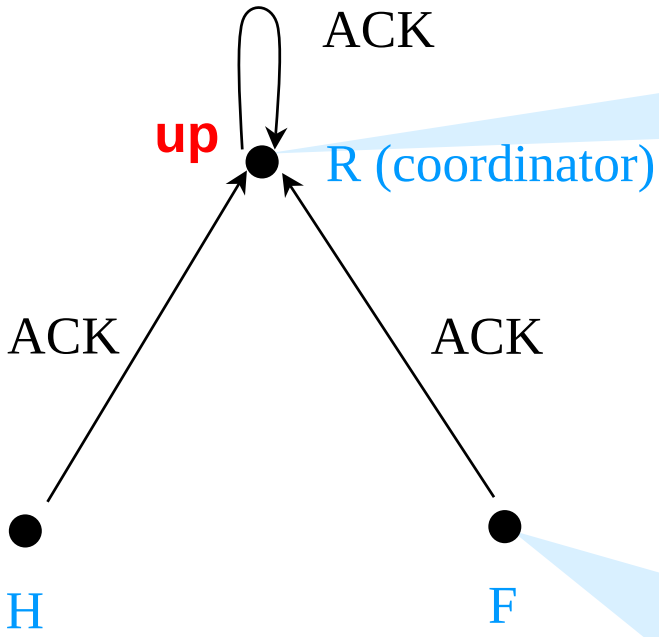
Termination during uncertainty window (14)



Termination during uncertainty window (15)



Termination during uncertainty window (16)



Global Log

Start-2PC
commit
done

Stable Database

Obj	*Val*
H-NYC	newval
W-NYC	oldval
...	**...**

Stable Log

Obj	*Redo*	*Undo*
H-NYC	newval	oldval
W-NYC	newval	oldval
voteCommit		
commit		

2PC: Performance

Performance

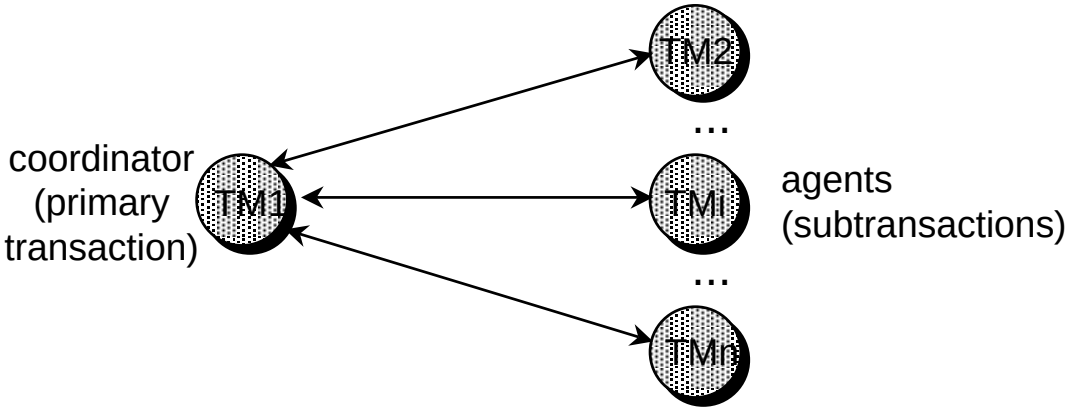
Requirements:

- As few messages as possible, and minimal number of writes to the log file.
- High robustness of protocol wrt failures: Keep probability of „blocking“ low.

Assumption: failures are rare
→ optimize design of system for normal mode.

Centralized commit

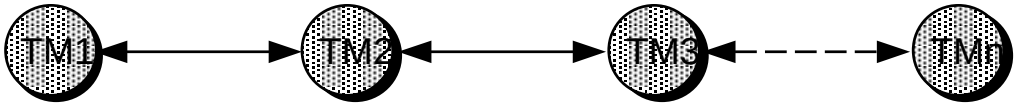
Basic protocol (treated so far):



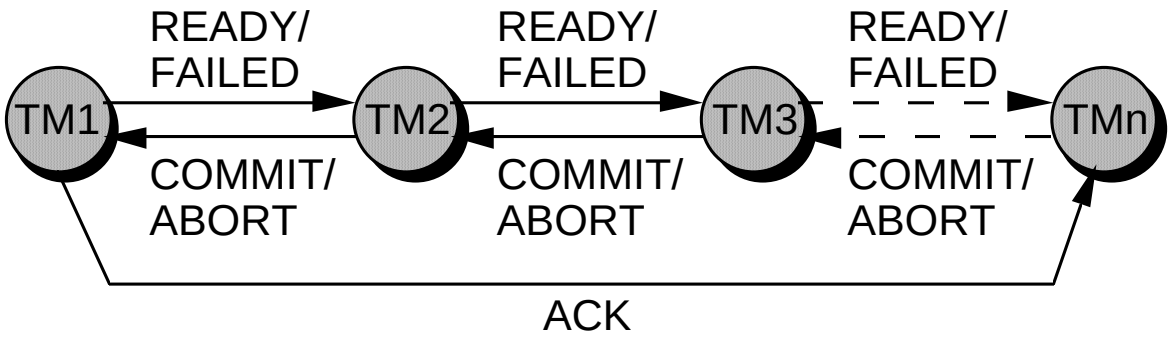
- Typically no communication between agents.
- Advantage: Commit processing in parallel at all agents.

- Four *messages* per agent $\rightarrow 4 \cdot (n\text{-}1)$ messages.
- For coordinator and each agent two *log writes* $\rightarrow 2 \cdot n$ log operations. (Processing delayed by duration of I/O.)

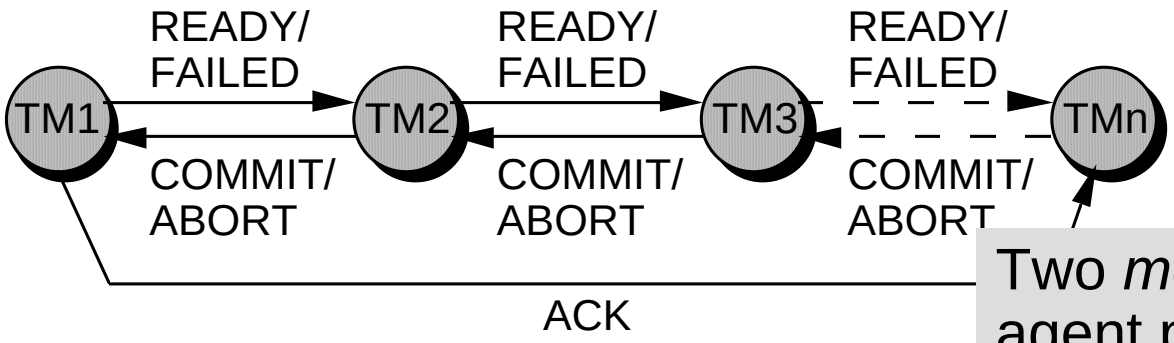
Linear commit (1)



- Commit is processed sequentially → takes longer.
- Phase 1: communication downstream from coordinator (TM1) to last agent (TMn); Phase 2: upstream.



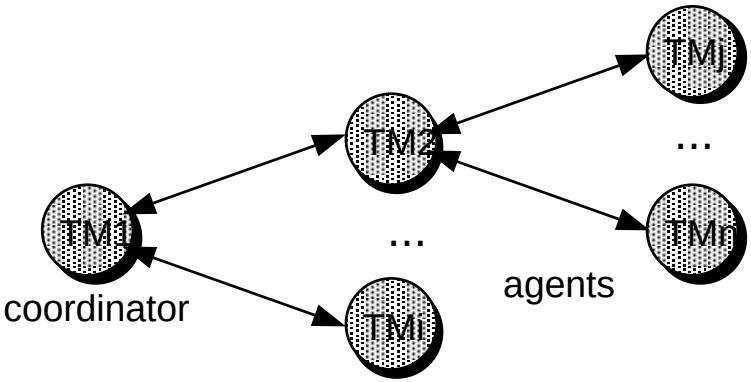
Linear commit (2)



Two *messages* per agent plus one *ack* → 2·*n*-1 messages.

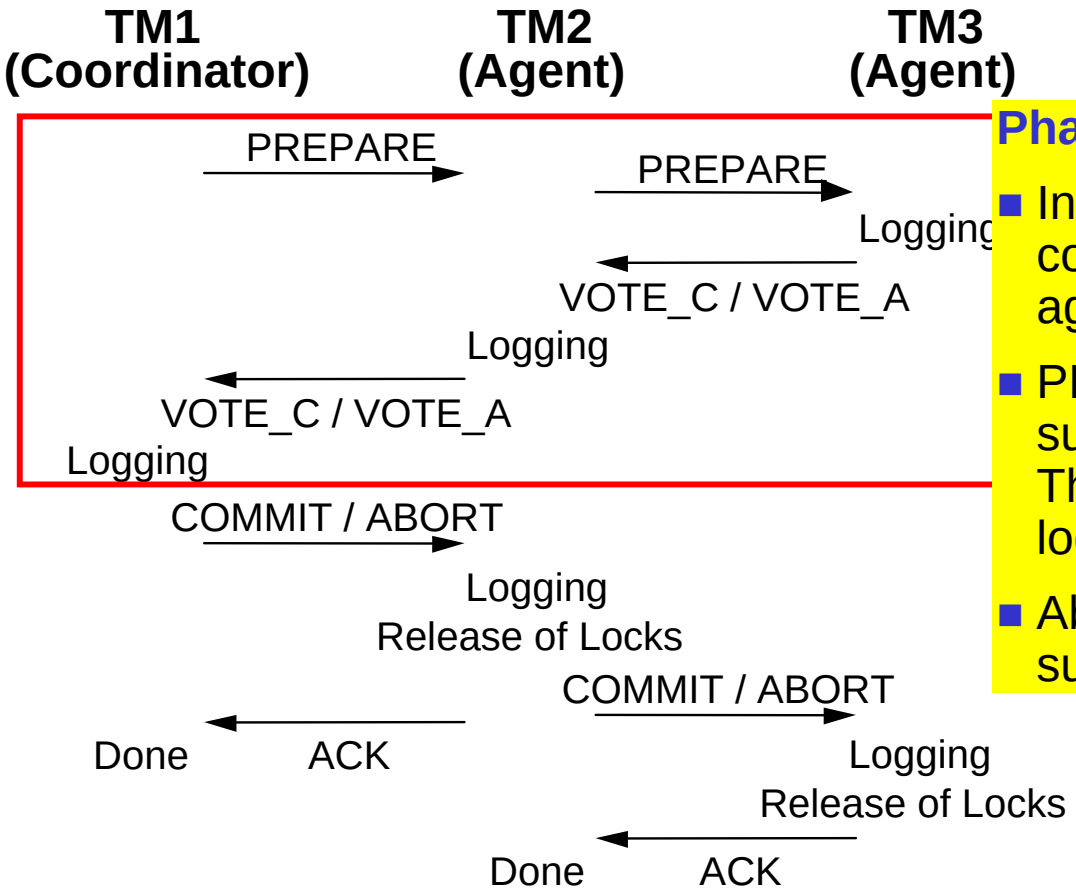
- Coordinator enters PREPARED state, passes its local commit decision (READY) to TM2.
  - Agent enters PREPARED state after having received READY, sends READY to next agent.
  - Successful termination of transaction is decided once last agent TMn has received READY and has written commit log entry.
  - Commit decision goes to agents in reverse order; logging and release of locks.
  - TM1 then sends ACK to TMn; TMn writes *done* log entry.
- Abort of transaction if one of the sites decides on abort; FAILED message is passed on.
  - Last agent (TMn) becomes coordinator: it logs global abort result and passes it on.

Hierarchical commit (1)



- Takes hierarchical invocation structure within global transaction into account, analogous to transaction tree.
- Each agent communicates only with direct ancestor and direct successors.
- Most general structure – previous ones are special case of this one.

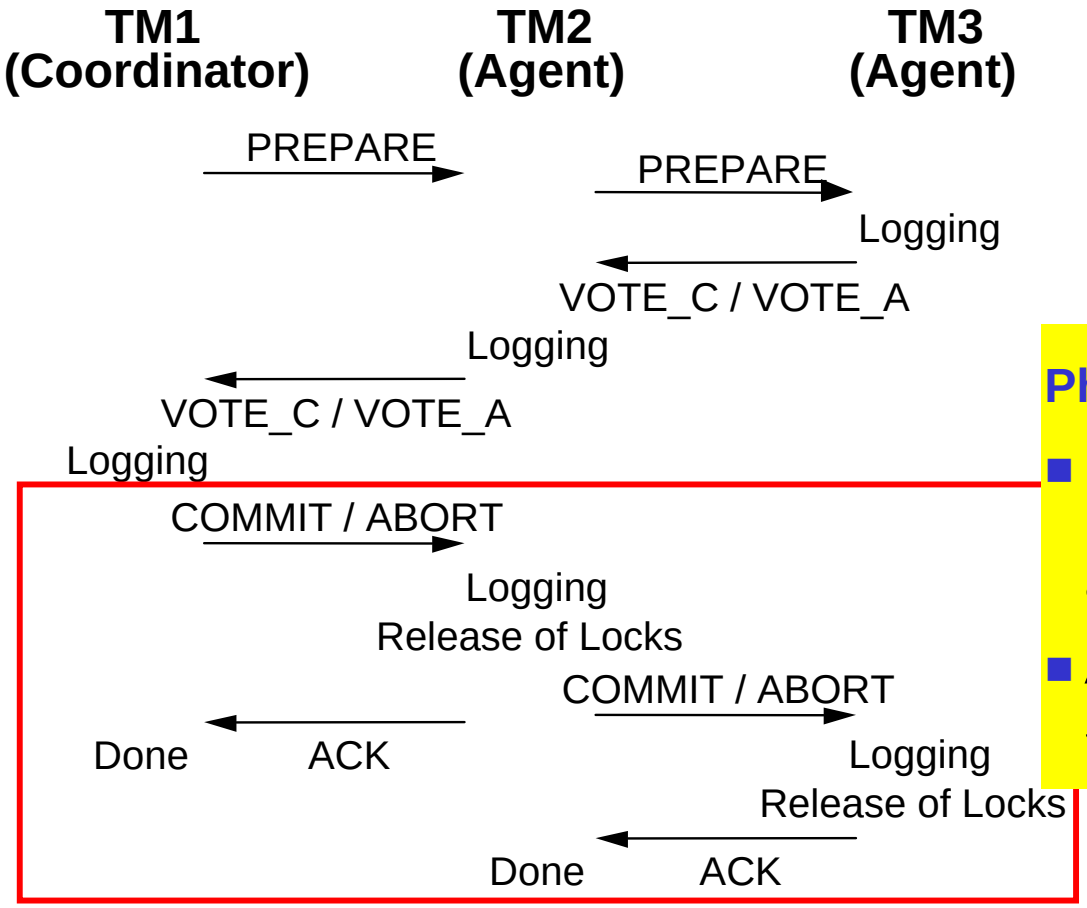
Hierarchical commit (2)



Phase 1:

- Intermediate nodes: coordinator for successors in tree, agent from its ancestor‘s perspective.
- PREPARE messages go to all successors, wait for their commit votes. Then commit decision for entire subtree, logging + sending it to ancestor.
- Abort – immediately inform all successors having voted commit.

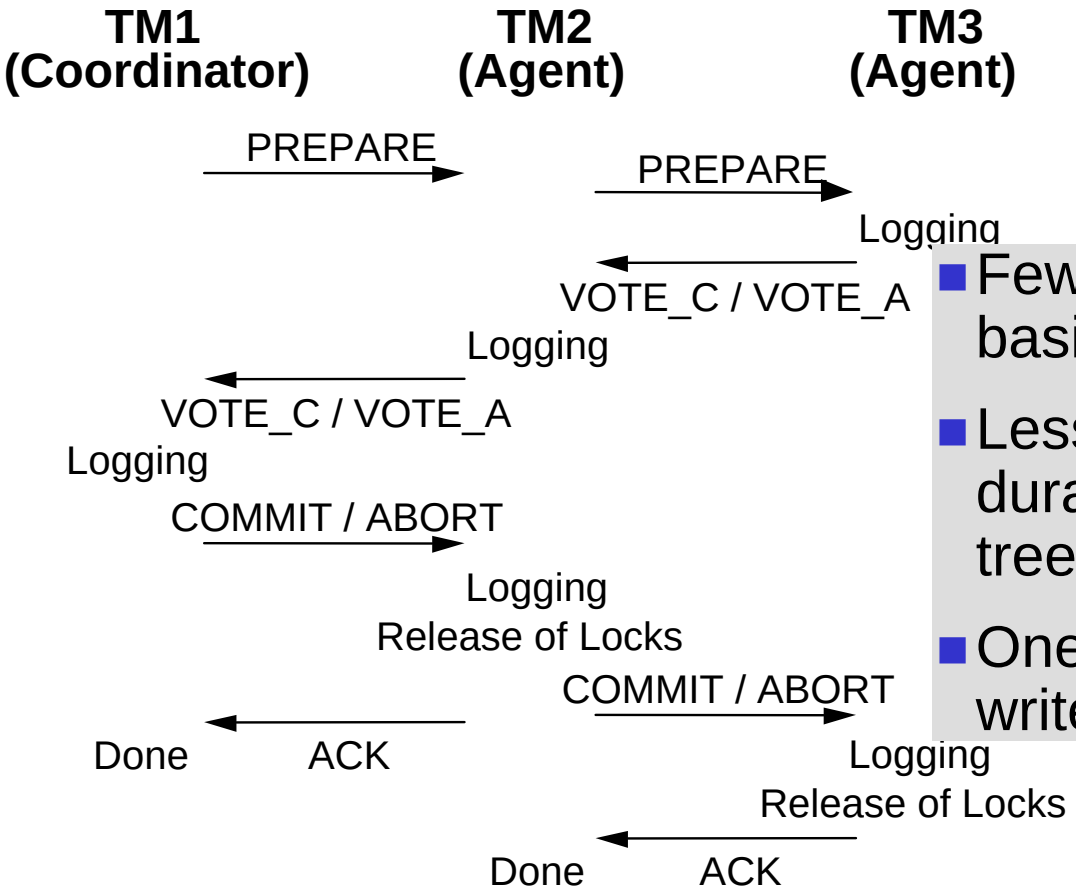
Hierarchical commit (3)



Phase 2:

- Receive decision from ancestor, log it, pass it on to successors, and confirm it immediately.
- After having received all *acks* from successors write done log entry.

Hierarchical commit (4)



- Fewer messages compared to basic scheme.
- Less parallelism and longer duration (proportional to height of tree).
- One additional (asynchronous) log write in intermediate node (done).

Presumed Abort

Also note:

Presumed abort improves performance on abort:

- Coordinator can write *abort* log entry asynchronously (no wait!).
- *ack* messages for failed transactions superfluous.
- *done* log entries superfluous at coordinator and intermediate sites.

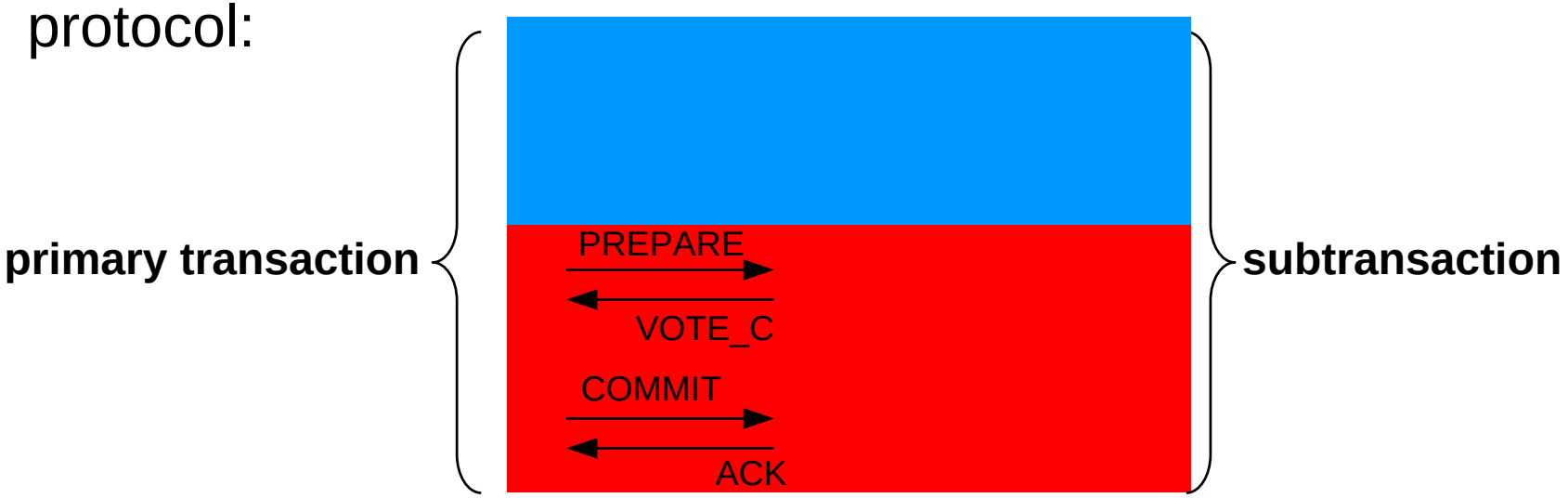
Incorporated in several products and in standards (ISO/OSI TP, X/Open DTP).

Read-only transactions

- Read-only site:
  - Neither recovery nor logging needed.
  - Only release of locks: Already possible in Phase 1 of 2PC, irrespective of success of global transaction → save entire second commit phase.
- If *m* subtransactions are read-only (of *n*-1),
  - number of messages reduced by $2m$ to $4 \cdot (n-1)-2m$,
  - number of log writes reduced to $2n-m$.
  - If global transaction is read-only ($m = n$), only $2 \cdot (n-1)$ messages and no log writes.

One-Phase Commit (1)

- So far: work in subtransactions separated from commit protocol:

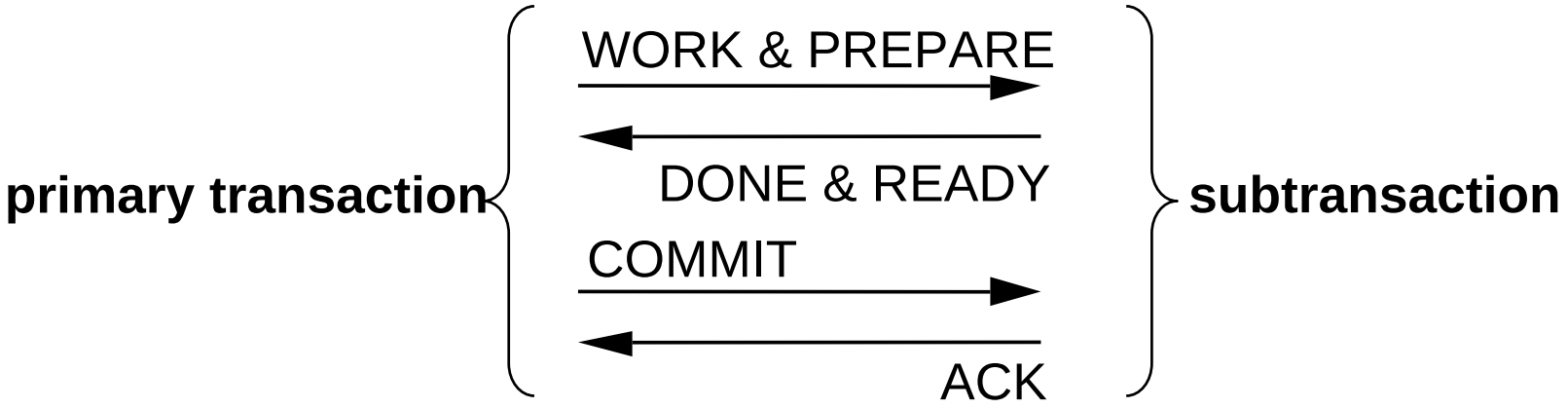


- Short distributed transactions with only one external database operation (e.g., money transfer to bank): commit processing is more expensive than transaction itself.

➔ Combine PREPARE message with WORK message.

One-Phase Commit (2)

- Subtransaction enters PREPARE state immediately after having executed operation and before replying to primary transaction.



- We save first commit phase, commit processing consists of only one phase to communicate global result.

➔ Only two messages per agent.

Why does it only work with short transactions?

Three-phase commit protocol (3PC)

Atomic Commitment Protocol - Summary

Atomic Commitment Protocol (ACP) is an algorithm that takes a (distributed) decision *Commit or Abort* such that:

- All sites that come to a decision come to the same decision.
- A site cannot change its decision once it has been taken.
- Decision for commit can only be taken if all sites have voted commit.
- If no failures occur and all sites vote commit the decision will be commit.
- Failures may occur but the algorithm can handle them. After all failures have been fixed and no new failures occur for a certain period of time a decision will be taken.

Atomic Commitment Protocol - Theorems

- **Theorem 1**:

  If communication failures and total site failures (all sites down) are possible, each ACP may lead to blocking.

- **Theorem 2**:

  No ACP can guarantee independent recovery of failed sites.

  (Independent recovery: Global state can be recovered without communication.)

Atomic Commitment Protocol – 2PC

- **Theorem 1**:

  If communication failures and total site failures (all sites down) are possible, each ACP may lead to blocking.

- **Theorem 2**:

  No ACP can guarantee independent recovery of failed sites.

2PC may even be blocking if no communication failures occur!

- All sites fail.
- Failing site splits network.

Long waits if agents are in *prepared* state and coordinator is down.

Site failures only

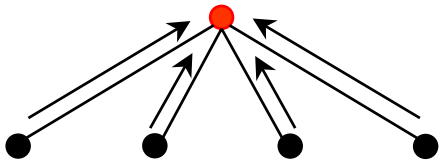
Invest more effort into reduction of blocking situations!

Solution 1: Tolerate site failures only:

- Non-blocking, except for total failures (all nodes down).
- Since only the sites may fail, **we can safely assume** that
  - all up-sites can communicate, and
  - the only reason a site may time out is that a partner site is down.

Introduce Non-Blocking Characteristic (1)

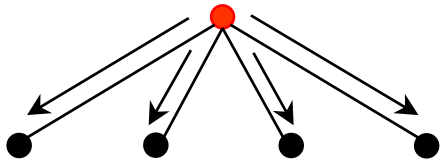
If an operational site is uncertain, no other site has yet committed.



VOTE_C-Messages

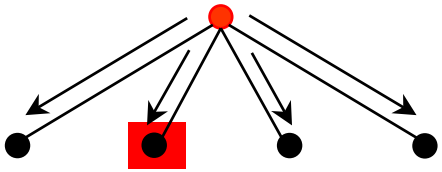
Non-blocking characteristic is not violated.

But 2PC may block.



COMMIT-Messages

Non-blocking characteristic is typically violated, since COMMIT messages do not arrive at same time.



COMMIT-Messages

Non-blocking characteristic is violated.

Introduce Non-Blocking Characteristic (2)

Conclusions:

- Non-blocking characteristic is not violated as long as all sites are uncertain.
- Sites that aborted cannot contribute to blocking because they did not decide on commit.

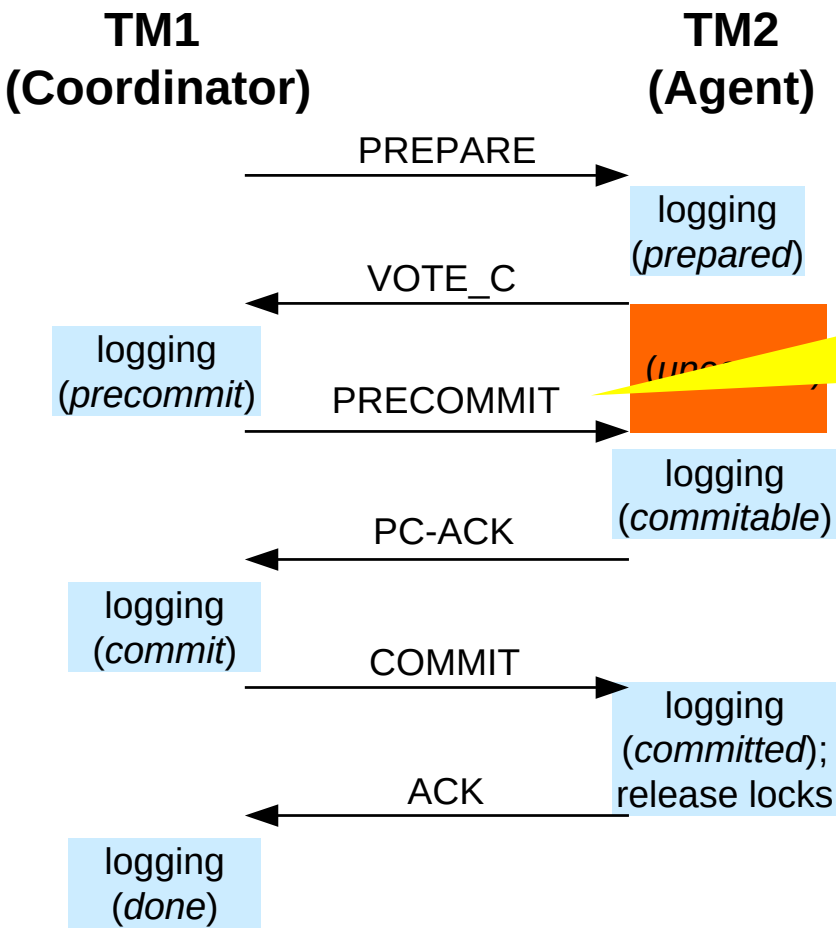
Desirable characteristic:

- Uncertain sites are allowed to abort.

Approach:

- End uncertainty, but do not yet commit $\Rightarrow$ for a while sites that are uncertain and those that are certain but did not commit may co-exist.

3PC General scheme (1)



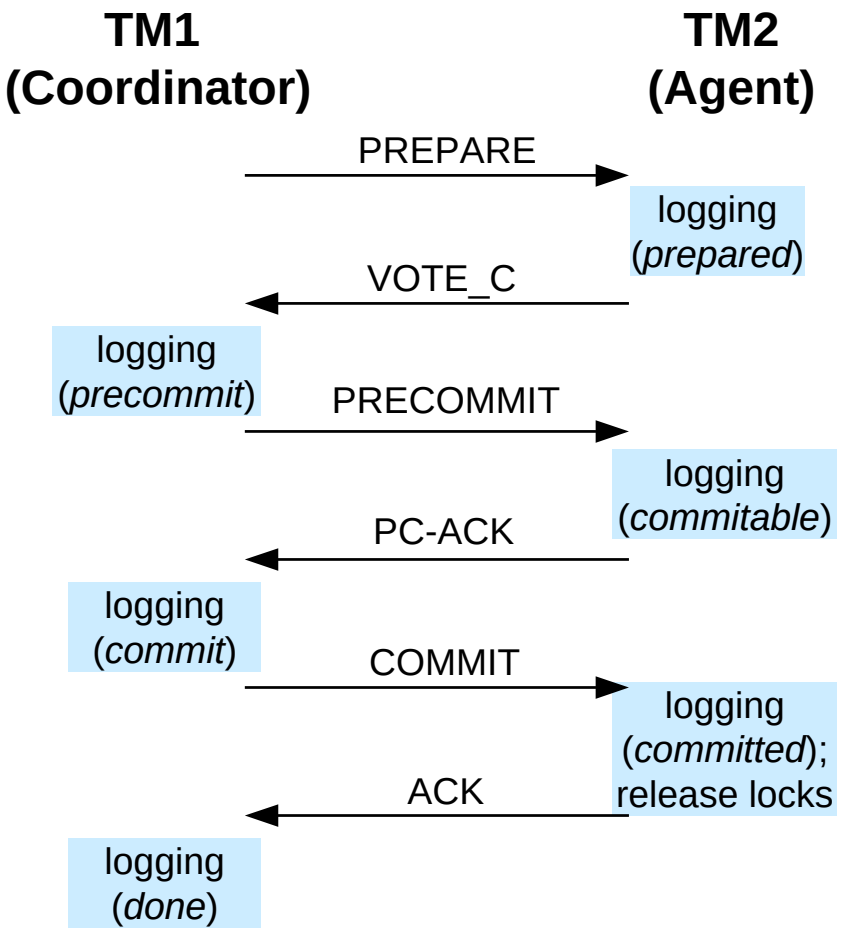
Phase 1 same as in 2PC.

PRECOMMIT tells site that it will eventually receive COMMIT message if coordinator does not fail.

Phase 2 new.

Phase 3 same as Phase 2 of 2PC.

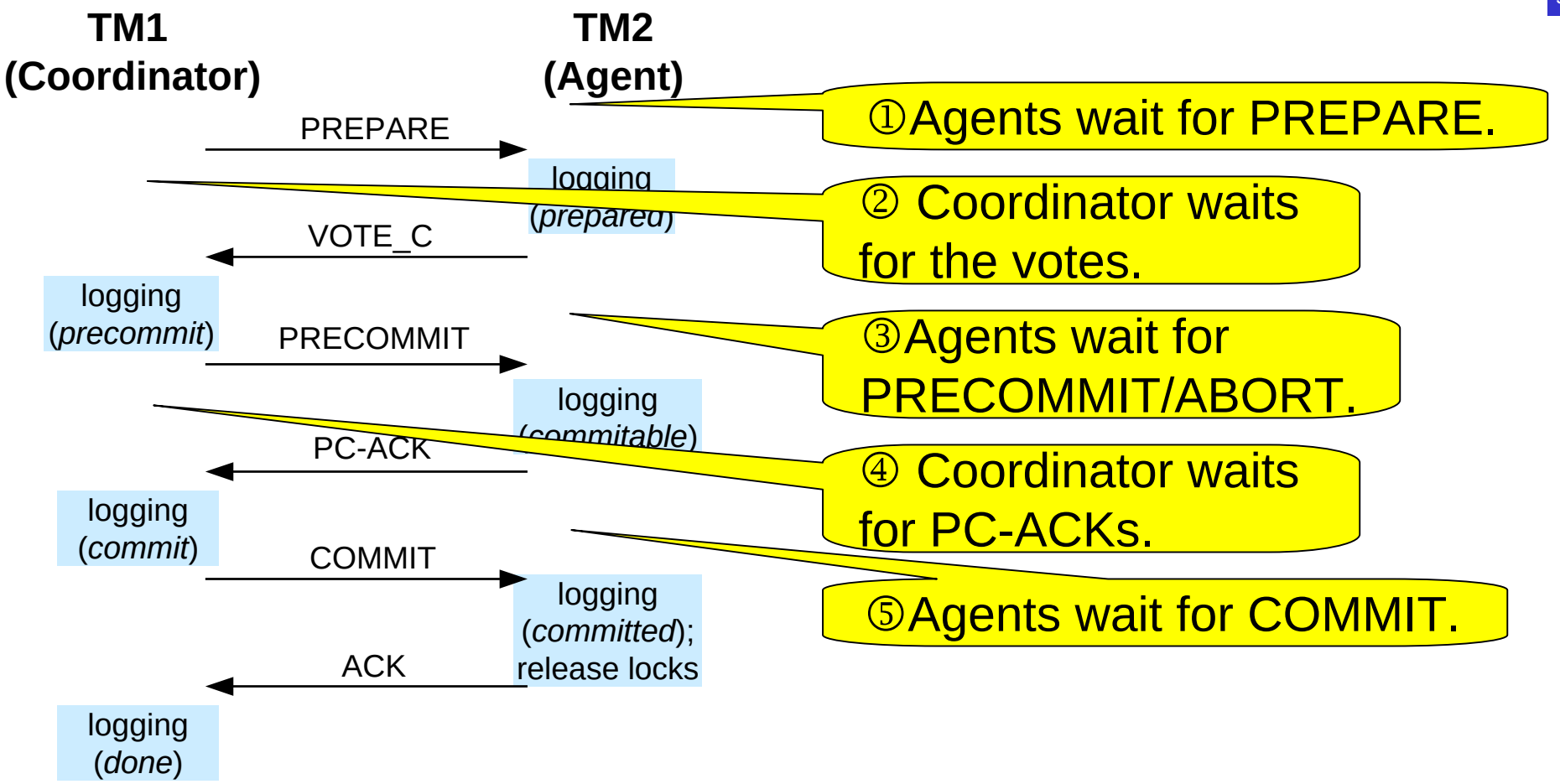
3PC General scheme (2)



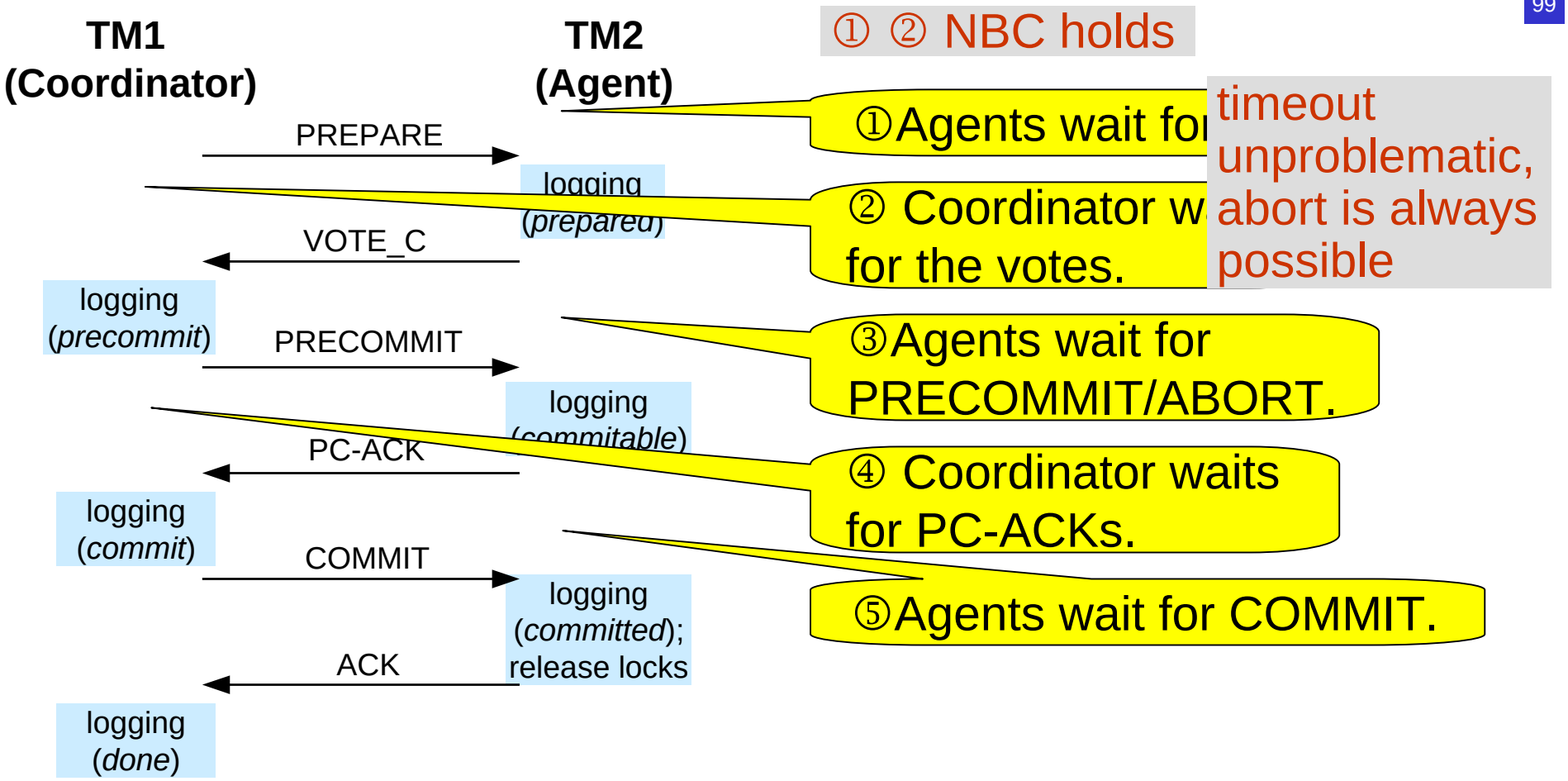
Phase 2:

- PRECOMMIT: Coordinator asserts that it will not abort transaction in future.
  - Site is no longer uncertain.
  - Site knows that it will eventually receive COMMIT message if coordinator does not fail.
  - TA may still abort if this coordinator fails.
- Agents write log entry and confirm.
- After PC-ACK messages have arrived, coordinator decides on commit and writes respective log entry.

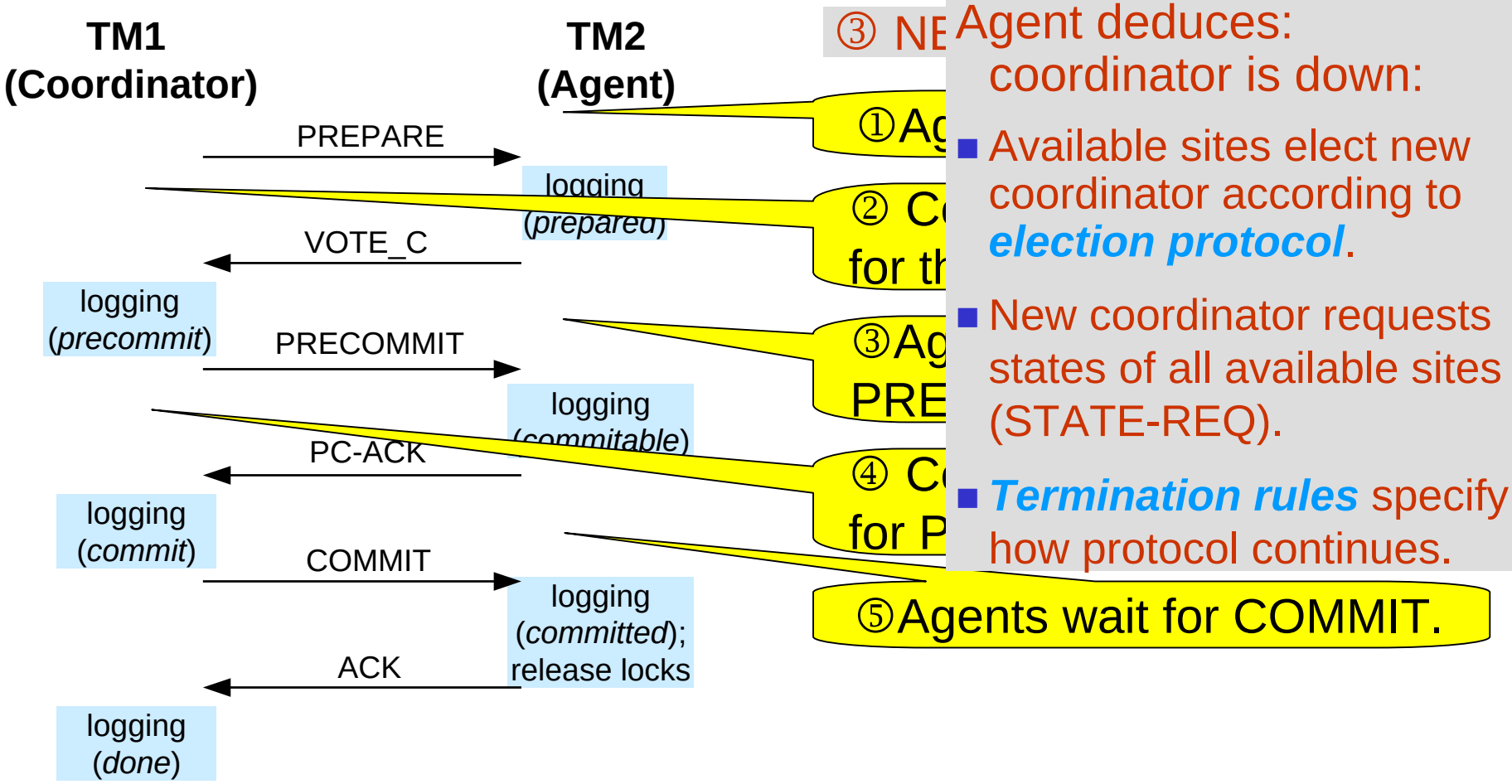
3PC Failure model (1)



3PC Failure model (2)



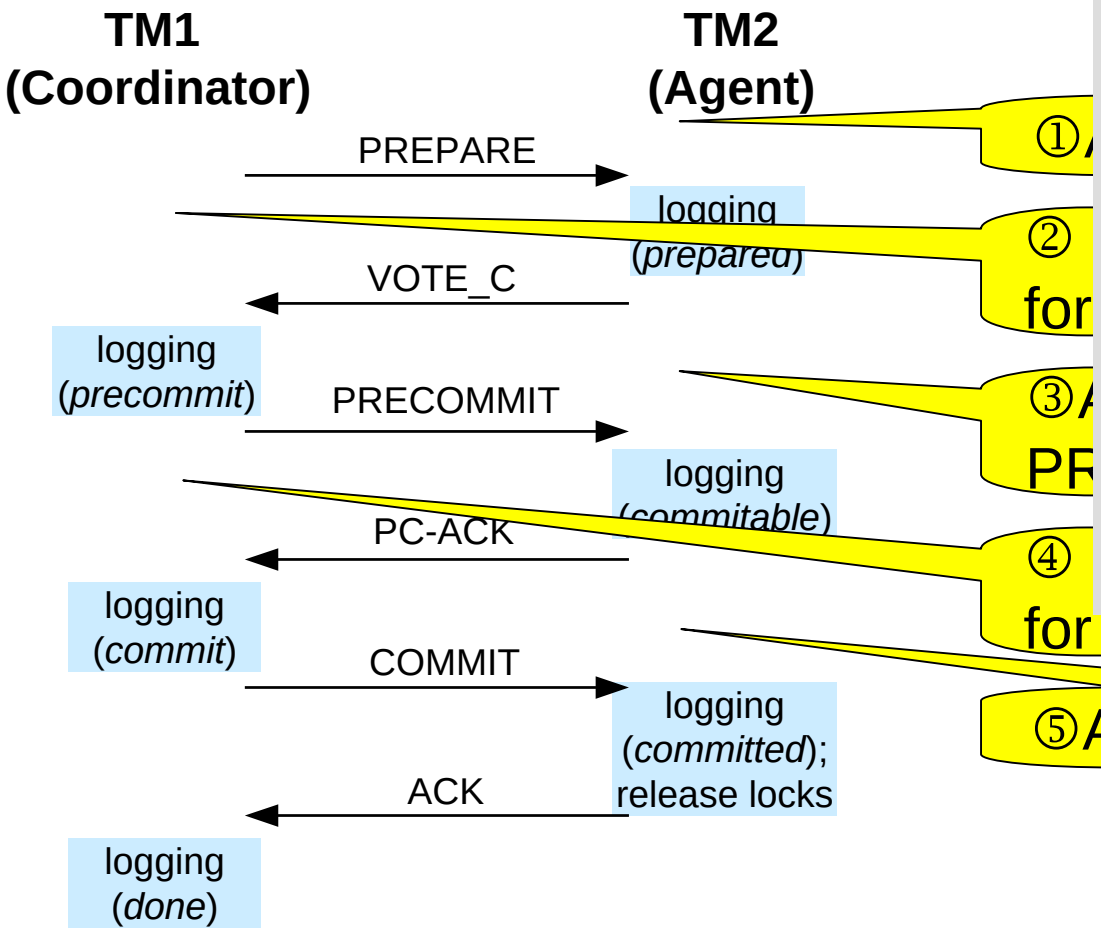
3PC Failure model (3)



Agent deduces:
coordinator is down:

- Available sites elect new coordinator according to ***election protocol***.
- New coordinator requests states of all available sites (STATE-REQ).
- ***Termination rules*** specify how protocol continues.

3PC Failure model (4)



① A

② for

③ A PR

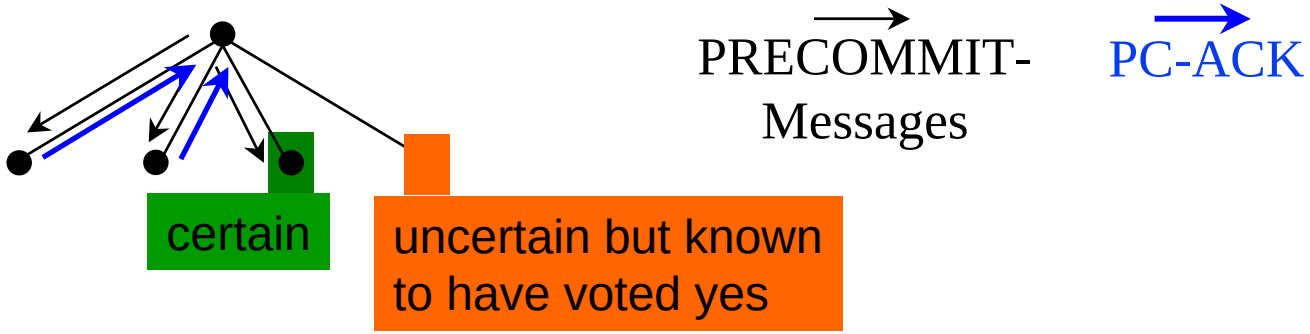
④ for PC-ACKs.

⑤ Agents wait for COMMIT.

④ NBC satisfied: Failed agent may still be uncertain, but no site committed yet.

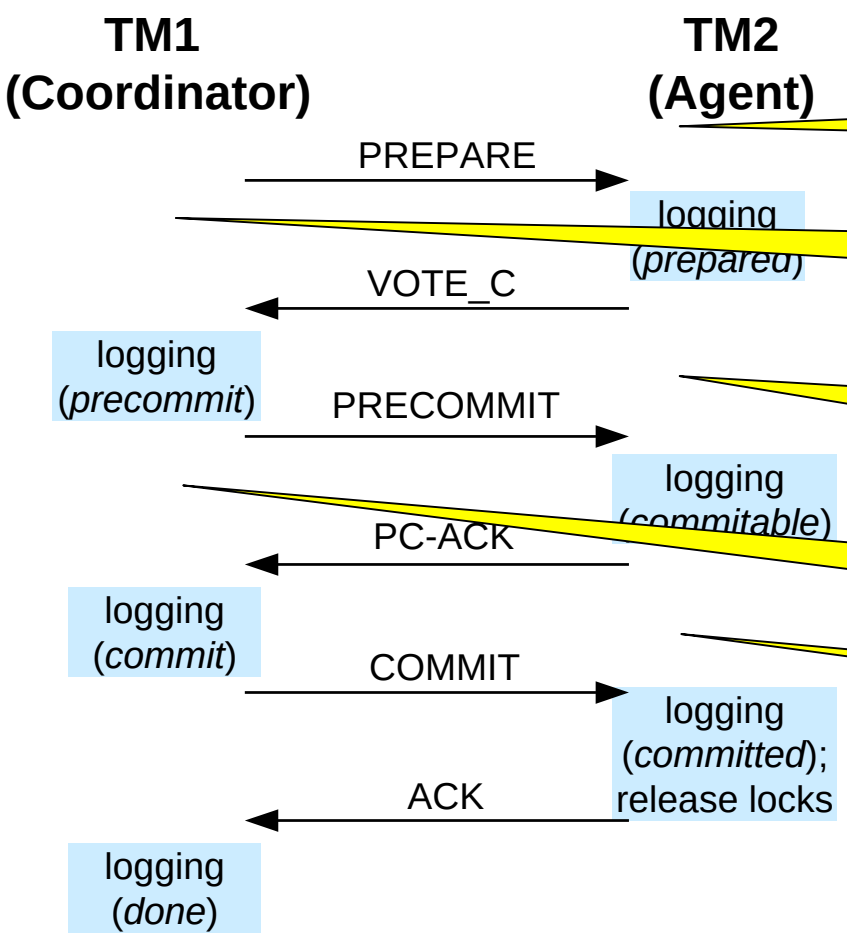
- Coordinator ignores missing PC-ACK messages; goes on after timeout period. ⇒ NBC remains satisfied.
- Failed agent must find out state of protocol after recovery.

3PC Failure model (4)



(message delays are always possible)

3PC Failure model (5)



⑤ NB Agent deduces:
Coordinator is down:

- Available sites elect new coordinator according to ***election protocol***.
- New coordinator requests states of all available sites (STATE-REQ).
- ***Termination rules*** specify how protocol continues.

①Ag

② C
for th

③Ag
PRE

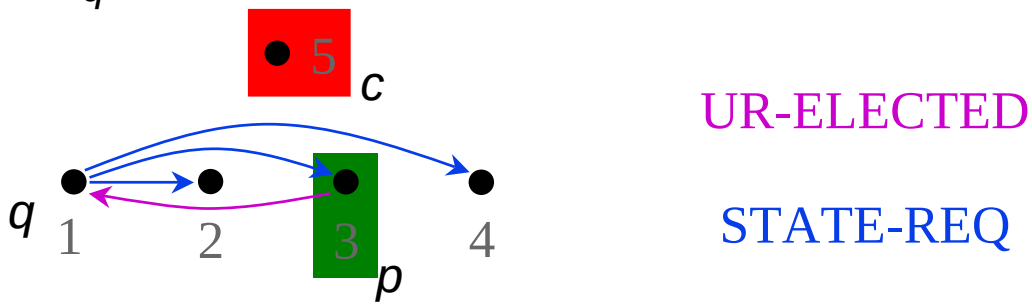
④ C
for P

⑤Agents wait for COMMIT.

Election Protocol (1)

- Prerequisites:
  - Linear ordering of sites (‚<‘).
  - Each site *p* maintains set UP_p of sites which *p* believes to be up.
- Agent *p* times out due to failed present coordinator *c*.
  - Remove *c* from UP_p and select smallest site *q*.
  - If $p \neq q$ send message UR-ELECTED to *q*.
- Agent *q* considers itself the new coordinator unless it can communicate with processes with smaller ID
  - Removes *c* from UP_q.
  - Send messages STATE-REQ (in lieu of PC-ACK) to agents in its UP_q.

$UP_p \neq UP_q$ possible!



Election Protocol (2)

- Let *p*‘ be such an agent receiving STATE-REQ from *q*.
- If *p*‘ did not timeout, it deduces the failure of *c* and recognizes *q* as the new coordinator.
  - Remove *c* and all *q*’<*q* from $UP_{q'}$.
- Due to message delay *p*‘ may have received STATE-REQ from a new coordinator *q*’ before that one failed, too.
  - We know *q*’<*q* $\Rightarrow$ ignore STATE-REQ messages from all sites *q*”<*q* .

Termination rules (1)

After new coordinator has collected states of available sites:

- TR1: a site has aborted ③
  ⇒ coordinator decides abort and sends out ABORT.
- TR2: a site has committed ⑤ some sites received COMMIT
  ⇒ coordinator decides commit and sends out COMMIT.
- TR3: all sites that have reported their state are uncertain (PREPARED state) ③
  ⇒ coordinator decides abort and sends out ABORT.
- TR4: a process is committable, none is committed ⑤
  ⇒ PRE-COMMIT messages to sites in uncertain state and wait for ACKs.
  Then commit decision, and COMMIT messages are sent out.

Termination rules (2)

How to deal with failures during termination:

- Ignore failures of agents.
- Failures of coordinator: Termination algorithm is repeated, but with fewer nodes $\Rightarrow$ nodes do not need to remain available.

Lemma: If NBC holds before the termination protocol starts, it will hold after even a partial execution of that protocol.

Site recovery after failure

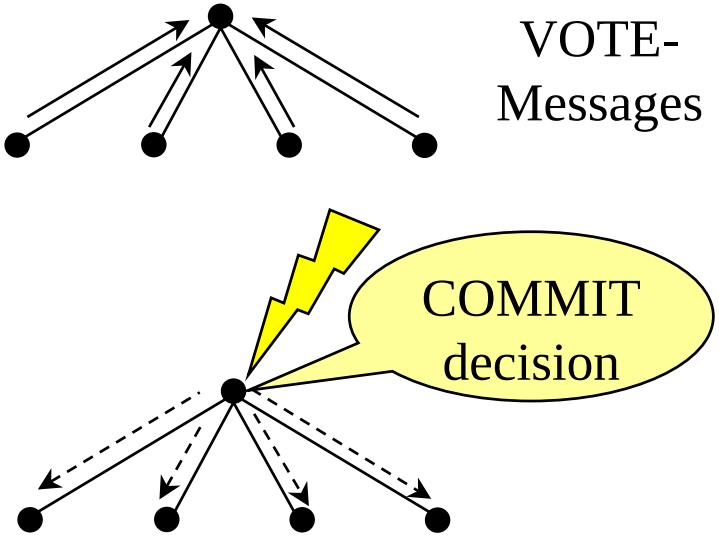
- Failure before VOTE_C: Unilateral abort and independent recovery.
- Failure after COMMIT or ABORT: Decision was taken, independent recovery.
- Failure while *uncertain* or *committable*: Contact other sites.
  - NBC guarantees that the site will eventually receive and adopt a decision.

Total failure

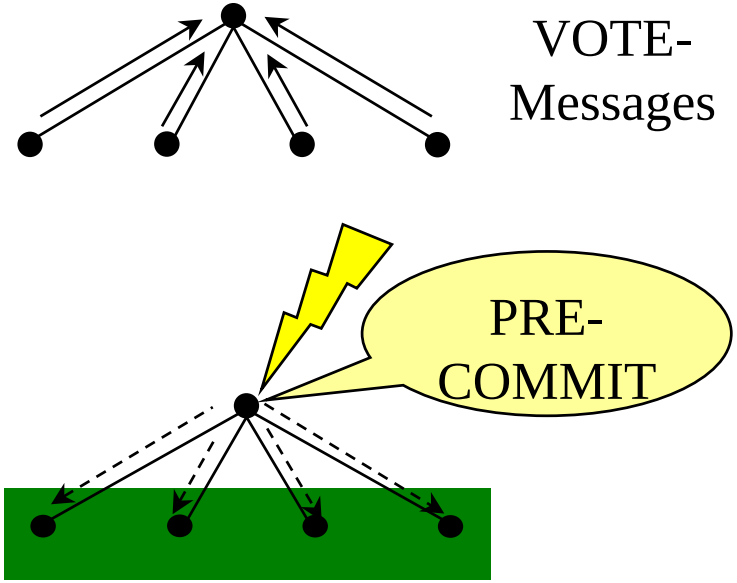
Protocol blocks in case of total failures. Unblocking proceeds as follows:

- Suppose that site *p* that has just recovered.
- As before, independent recovery before VOTE_C or after COMMIT or ABORT $\Rightarrow$ Communicate decision to the other sites.
- As before, autonomous decision is not possible if the failure occurred while *uncertain* or *committable*.
  - Decision for commit or abort could have been taken by a site failing after *p*.
  - However, it can take a decision if it was the last one to fail $\Rightarrow$ invoke the termination protocol.

2PC vs. 3PC



Blocked if coordinator takes decision but fails to send messages.

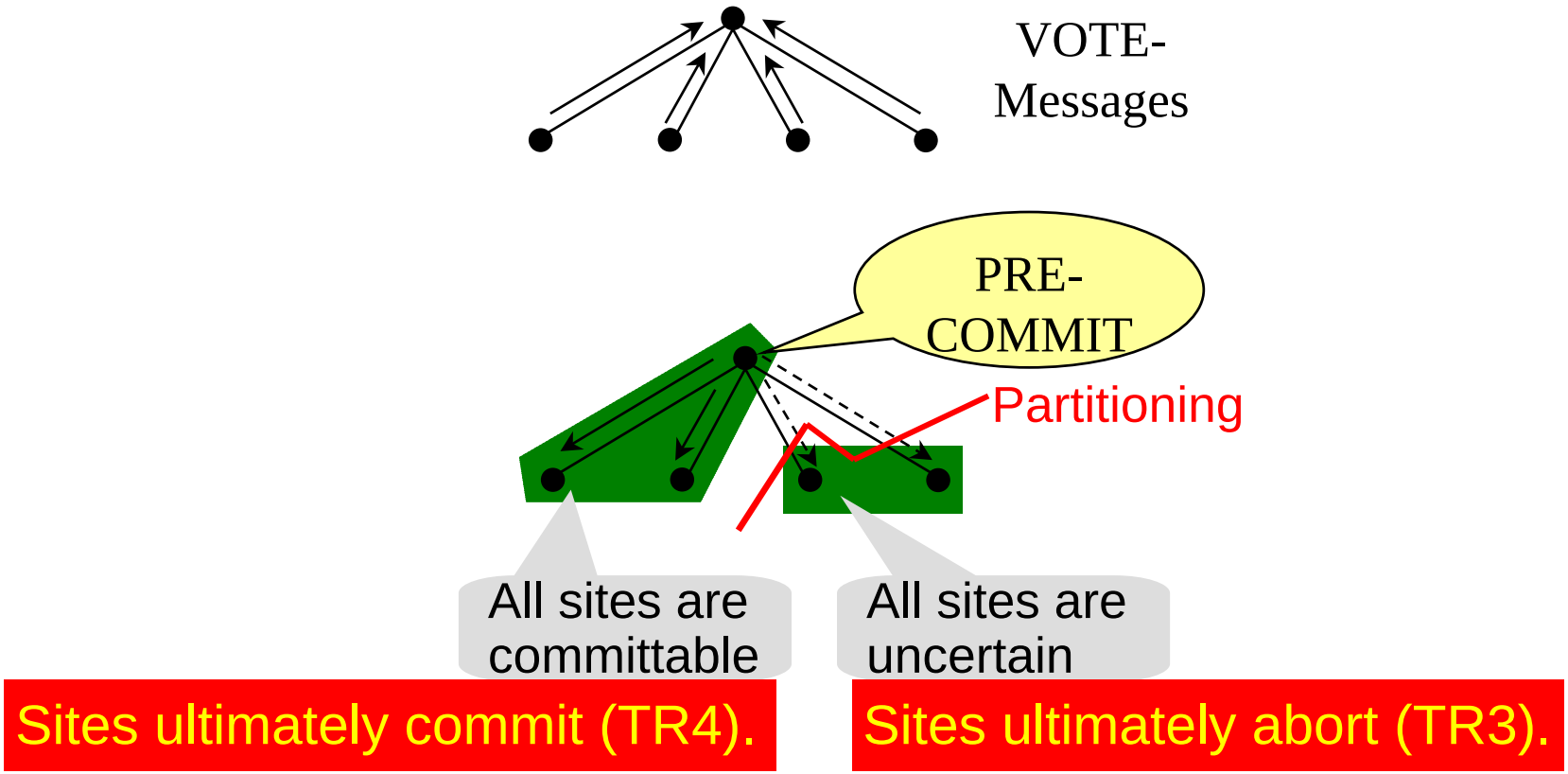


If coordinator initiates decision but fails to send messages:

- No wait, instead agents elect new coordinator.
- We know votes of all other nodes and that no decision has yet been taken.

Communication failures (1)

Present solution may result in inconsistent global state:



Problem: Election of multiple coordinators!

Communication failures (2)

Invest more effort into reduction of blocking situations!

Solution 1: Tolerate site failures only:

- Non-blocking, except for total failures (all nodes down).
- Since only the sites may fail, **we can safely assume** that
  - all up-sites can communicate, and
  - the only reason a site may time out is that a partner site is down.

None is true any longer!

Solution 2: Tolerate both, site and communication failures.

3PC extended (1)

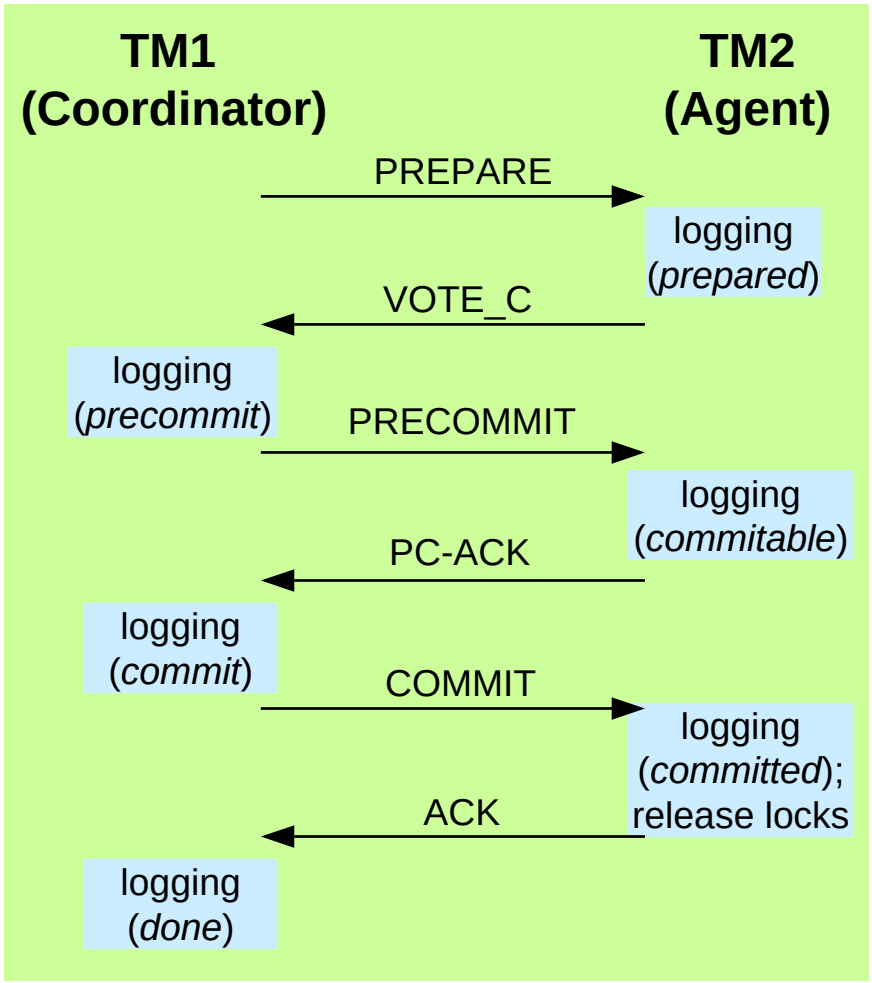
Problem: Election of multiple coordinators!

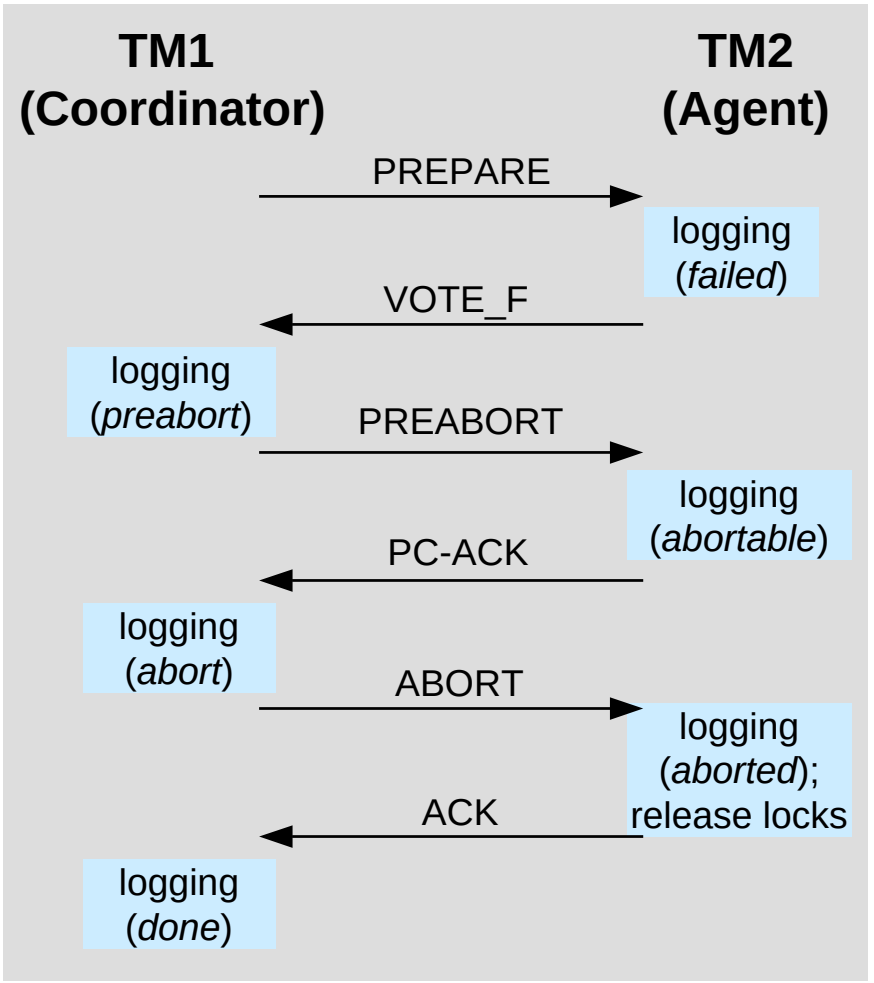
Remedy: A coordinator is only allowed to reach a decision if it can communicate with a majority of the sites.
⇒ Ensures that decision reached by two different coordinators is based on at least one site in common.

Conflict: A common site detects a conflict if its one coordinator intends to decide on commit and another on abort.

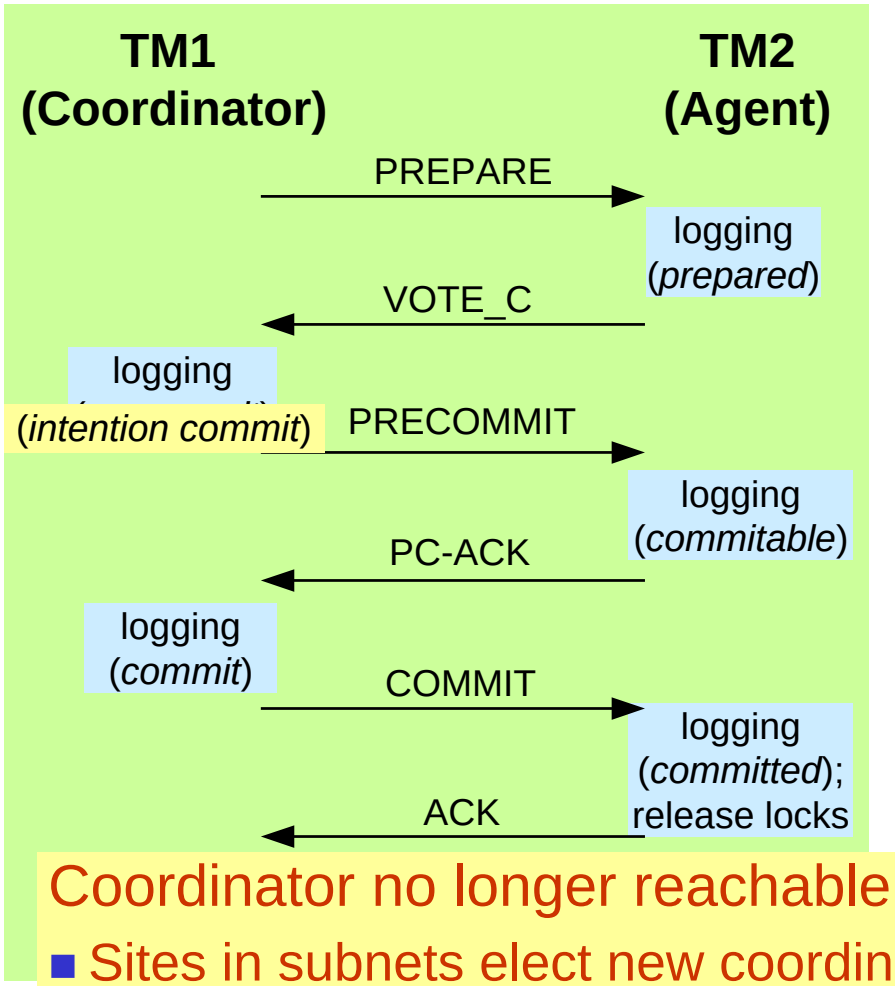
Required: Both, PRE-COMMIT and PRE-ABORT.

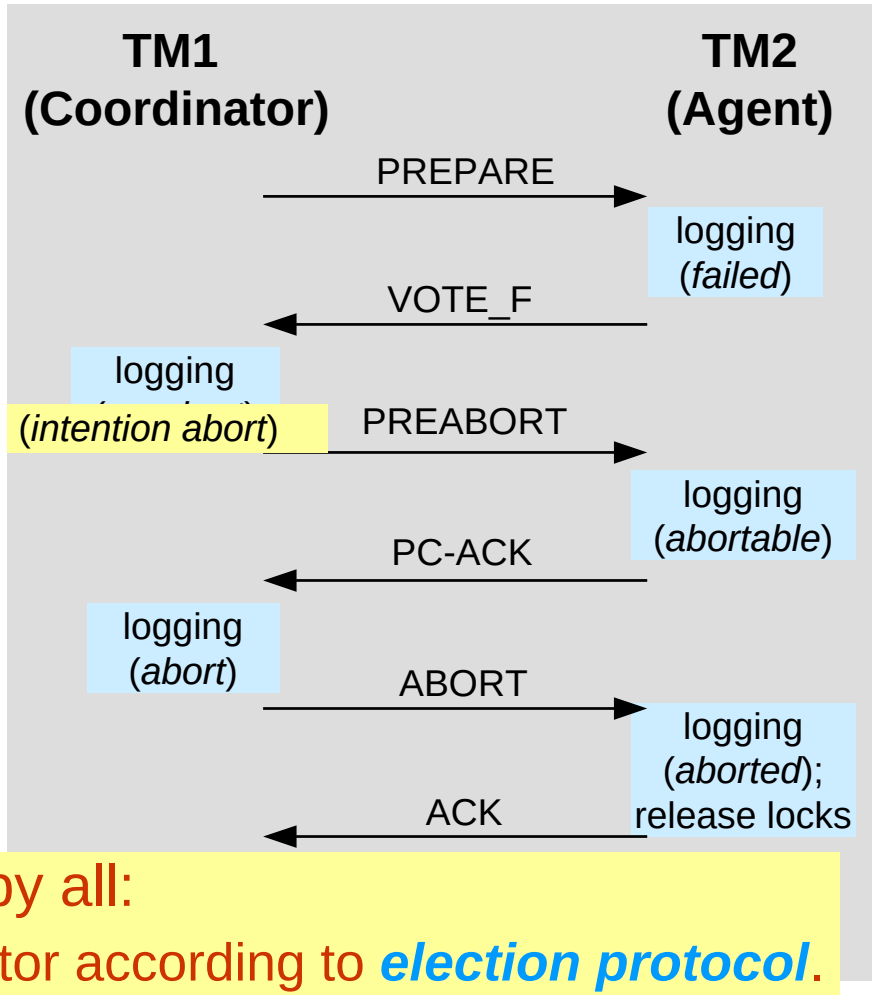
3PC extended (2)





3PC extended failure model





Coordinator no longer reachable by all:

- Sites in subnets elect new coordinator according to ***election protocol***.
- New coordinators request states of all available sites (STATE-REQ).
- ***Termination rules*** specify how protocol continues.

Majority termination rules (1)

After new coordinator has collected states of available sites:

- TR1: a site has aborted
  $\Rightarrow$ coordinator decides abort and sends out ABORT.
- TR2: a site has committed
  $\Rightarrow$ coordinator decides commit and sends out COMMIT.
- TR3:
  1. Coordinator: One committable and a majority non-abortable states (prepared or committable) $\Rightarrow$ *PRE-COMMIT* messages to all agents that have not sent commitable.
  2. Agent: *PRE-COMMIT* $\Rightarrow$ new state committable
     $\Rightarrow$ *PRE-COMMIT-ACK*
  3. Coordinator: If agents in state commitable form a majority: commit; otherwise blocking.

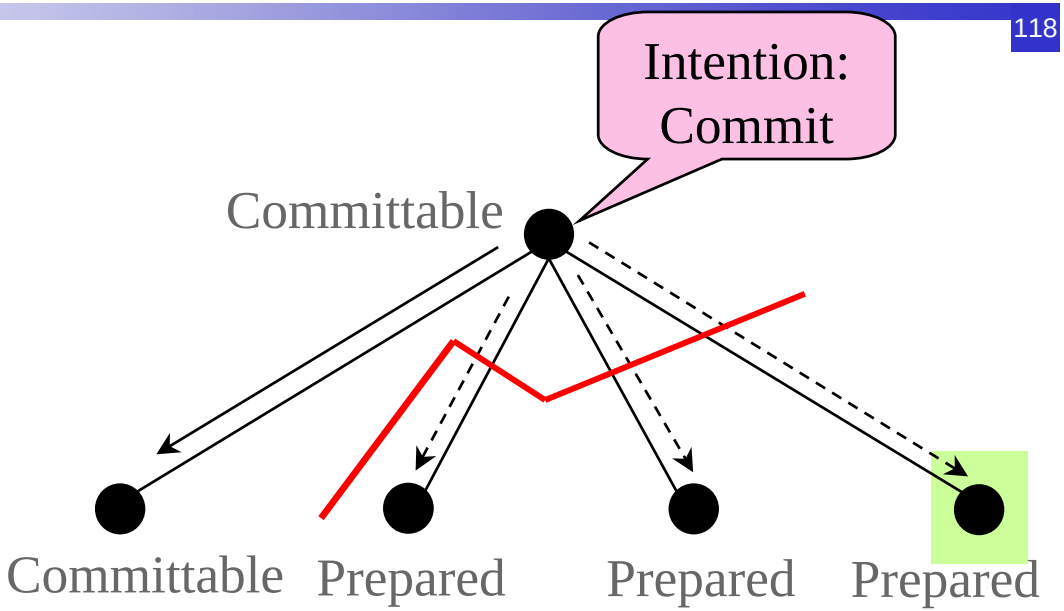
Majority termination rules (2)

After new coordinator has collected states of available sites:

- TR4:
  1. Coordinator: Majority of non-committable states (prepared or abortable) $\Rightarrow$ *PRE-ABORT* messages to all agents that have not replied abortable.
  2. Agent: *PRE-ABORT* $\Rightarrow$ new state abortable $\Rightarrow$ *PRE-ABORT-ACK*.
  3. Coordinator: if agents in state abortable form a majority: abort; otherwise blocking.
- TR5: Otherwise: blocking.

Illustration (1)

- After timeout, green component elects new coordinator.
- New coordinator collects states from nodes in component.
- New coordinator decides on ‘intention abort‘ (TR4).



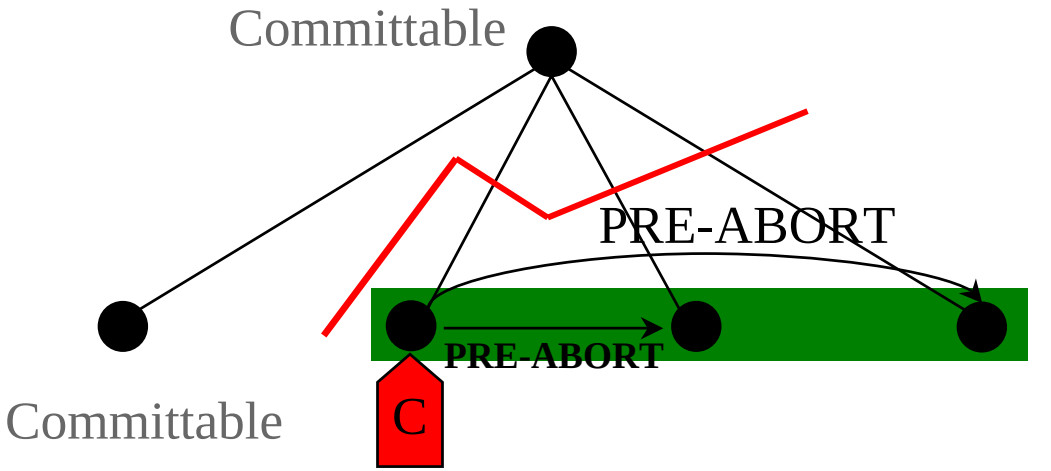


Illustration (2)

- What happens after communication has been re-established?

⇒Agents in ‘abortable’ state will not react to PRE-COMMIT message, coordinator will not be able to form a commit majority ⇒ abort (TR4).

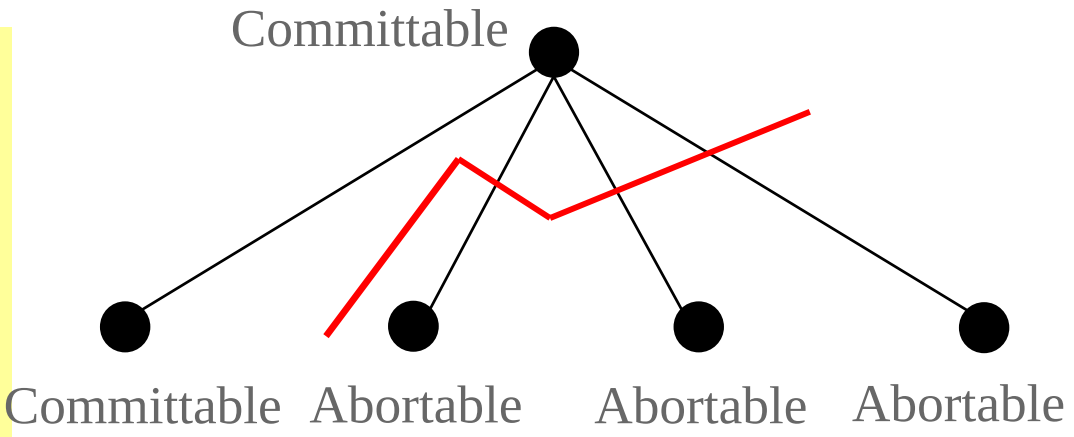
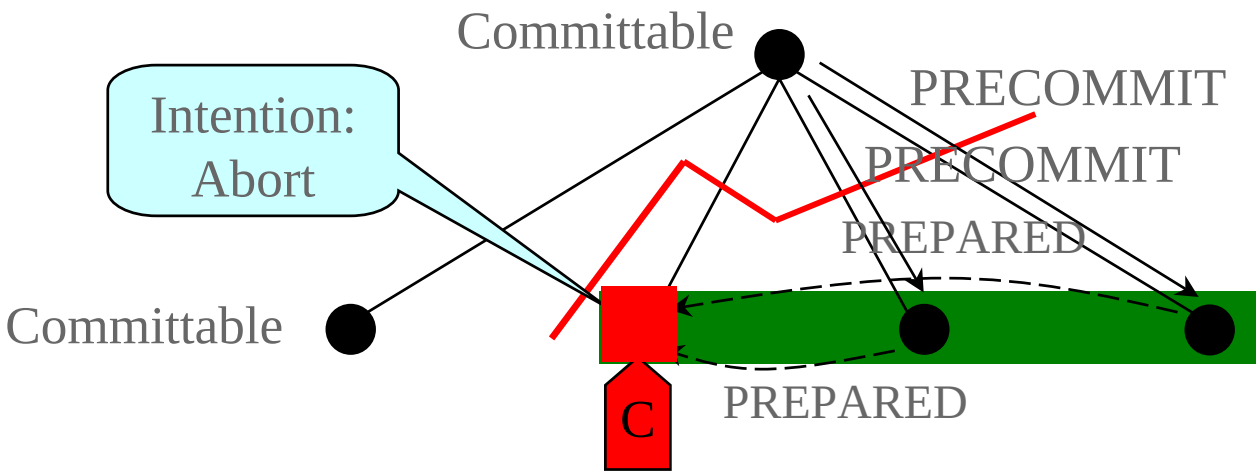


Illustration (3)

- What if there were no PRE-ABORT messages?



Performance comparison

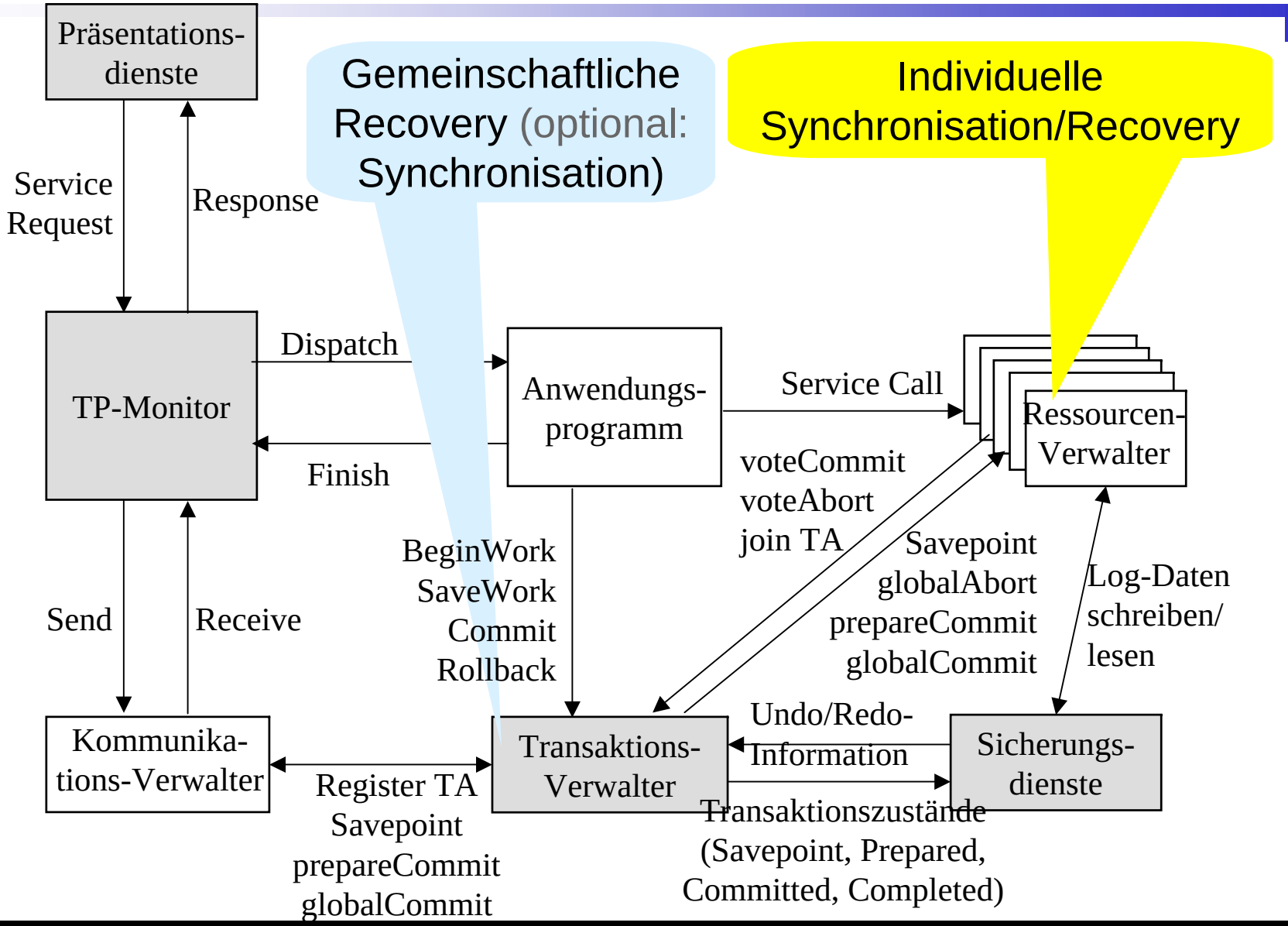
- **Number of messages**
  (including optimized treatment of read-only subtransactions)
  - transaction has executed at n nodes ($n > 1$),
    m nodes with read-only subtransactions ($m < n$)

	general	Example 1 (n=2, m=0)	Example 2 (n=10, m=5)
1PC	2(n-1)	2	18
Linear 2PC	2n-1	3	19
Centralized/ hierarchical 2PC	4(n-1)-2m	4	26
3PC	6(n-1)-4m	6	34

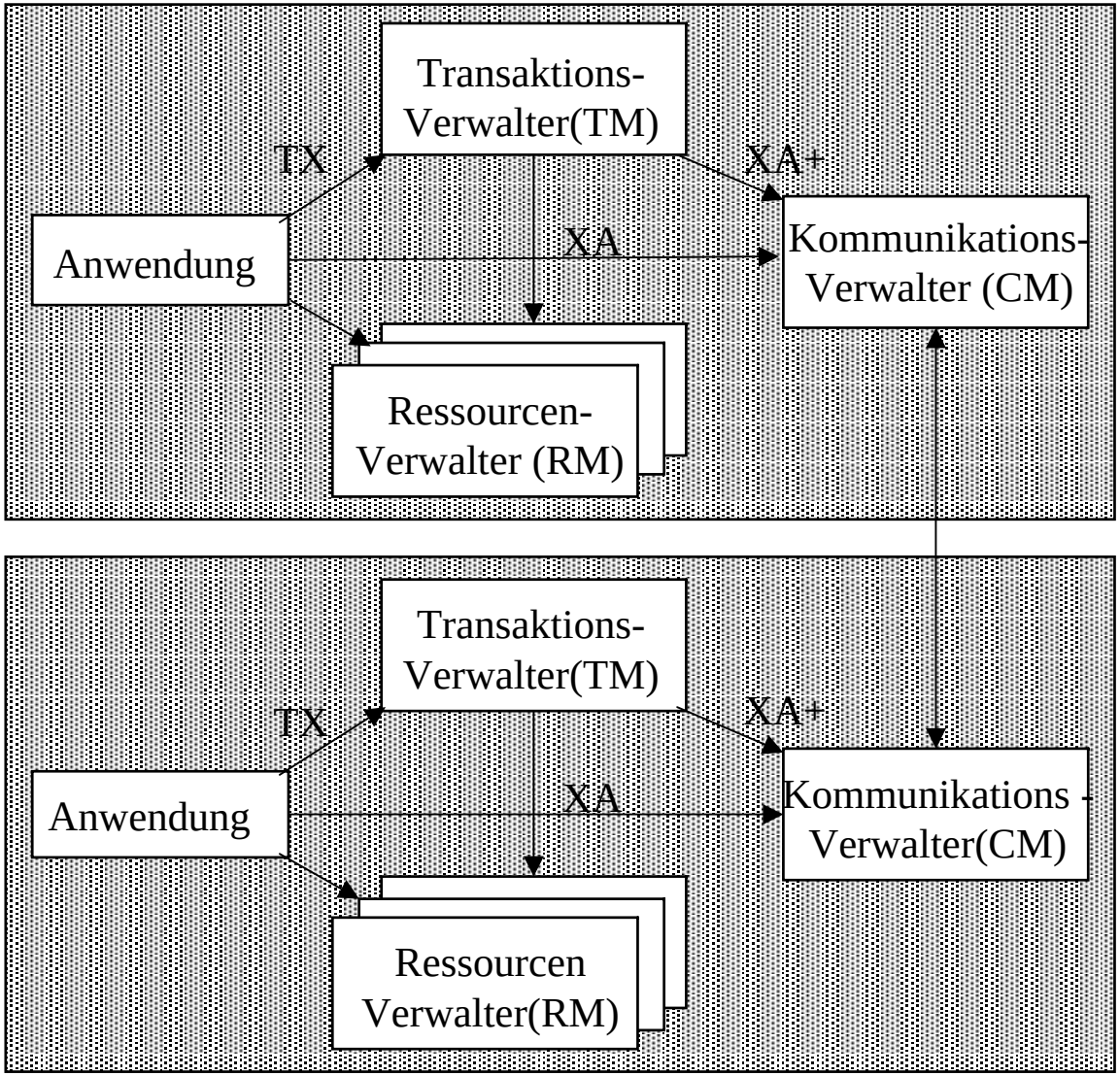
Also more log writes (3n)

Architectures

Transaktionsverwaltungs-Monitore



X/Open-Modell eines verteilten Transaktionssystems



DBS

Universität Karlsruhe (TH)

Chapter 12

Long Transactions

Special problems

Three broad application classes where long transactions occur:

- Data analysis and maintenance tasks for ‚conventional' databases (example of maintenance: index reconstruction) $\Rightarrow$ more efficient recovery needed in case of failure.
- Workflow systems with humans in the loop $\Rightarrow$ execution times become unpredictable, exclusive locks impede collaborative progress.
- Design systems $\Rightarrow$ exclusive lock for objects checked out impedes concurrent, collaborative progress; efficient recovery needed.

Efficient recovery

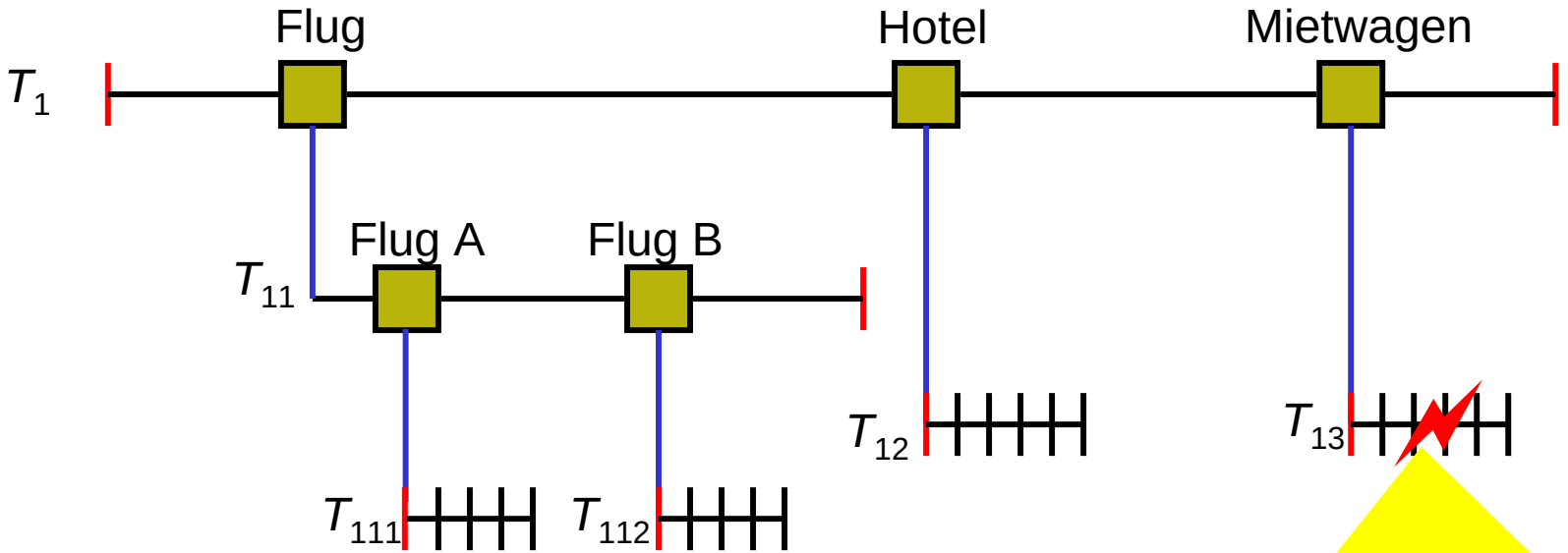
Motivation

- Outside appearance: ACID
- Internal structure:
  - Finer-grained recovery.
  - Parallelism.
- Approach: Organize long transaction as a tree of subtransactions.

⇒ **Closed nested transaction**

Beispiel (1)

Buchung einer Reise: Hierzu seien z.B. die Buchung unterschiedlicher Flugabschnitte, eines Hotels und eines Mietwagens notwendig.



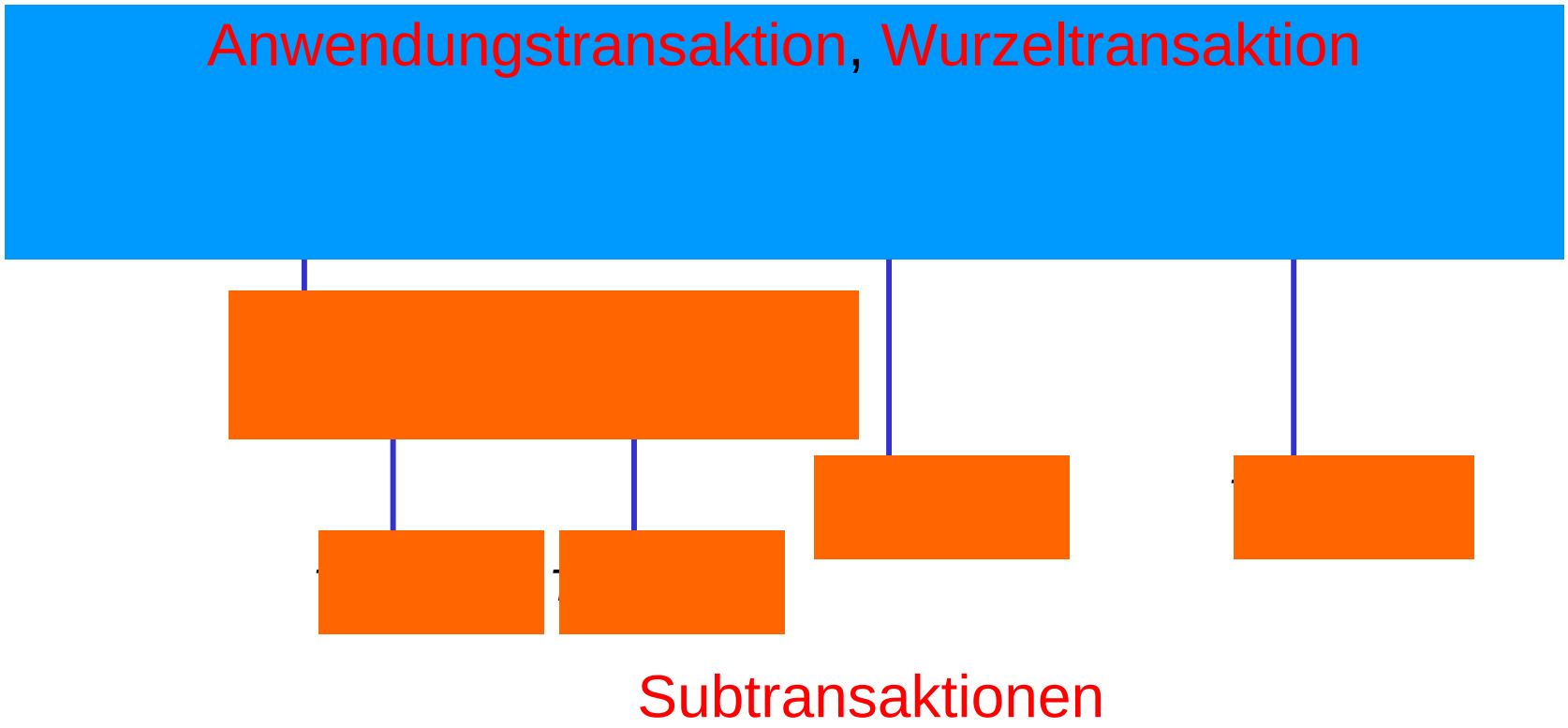
Sinnvoller wäre es, lediglich die Mietwagenbuchung (evtl. modifiziert) zu wiederholen.

Konventioneller TM: Es muss die gesamte Transaktion zurückgesetzt werden

Beispiel (2)

Lösung:

Die Teilaktivitäten werden als eigene **Subtransaktionen** betrachtet. Es entsteht ein **Transaktionsbaum**.



Vorgehen

Ziel:

Weitgehende Verwendung des konventionellen TM.

Aufgabe:

Bestimme notwendige Erweiterungen.

Recovery (1)

Differenziertere Recovery: Schrittweises, hierarchisches Vorgehen soll sich widerspiegeln.

- Blatt-Transaktionen sind ACID.
- Zusammenhänge zwischen Transaktionen entlang eines Pfades:
  - Fehlschlag einer Subtransaktion → Störung, aber Fehlschlag der Vatertransaktion nicht zwingend.

    Vatertransaktion bestimmt *Stelle* in ihrer Ausführung, *ab der* Fortführung möglich ist.
  - Störung in der Vatertransaktion → Fehlschläge aller Subtransaktionen *ab einer* von der Vatertransaktion zu bestimmenden *Stelle*.
  - Fehlschlag der Vatertransaktion → Fehlschlag *aller* bereits ausgeführten Subtransaktionen.

Recovery (2)

Differenziertere Recovery: Schrittweises, hierarchisches Vorgehen soll sich widerspiegeln.

- Blatt-Transaktionen sind ACID.

konventioneller TM

- Zusammenhänge zwischen Transaktionen entlang eines Pfades:
  - Fehlschlag einer Subtransaktion → Störung, aber Fehlschlag

1. Jede Subtransaktion bildet eine eigene Commit-Sphäre: Sie gibt Änderungen an ihrem Transaktionsende, d.h. vor der Beendigung der umgebenden Transaktion, frei.
2. Fehlschlag einer Subtransaktion: Zurücksetzen.
3. Bei Fehlschlag der umgebenden Transaktion kein Rücksetzen möglich (Subtransaktion ist abgeschlossen!), stattdessen Aufruf von Kompensationstransaktionen.

Nebenläufigkeit (1)

Nebenläufigkeit innerhalb der Transaktion:

- Subtransaktionen innerhalb einer Anwendungstransaktion lassen sich parallel bearbeiten (Intratransaktions-Parallelität).
- Im Beispiel könnte etwa die Hotel- und die Mietwagenbuchung u.U. gleichzeitig geschehen.
- Es gilt unverändert die Konfliktregelung:
  - Serialisierbarkeit innerhalb der Anwendungstransaktion
  - Sperrenfreigabe nach außen erst am Ende der Anwendungstransaktion

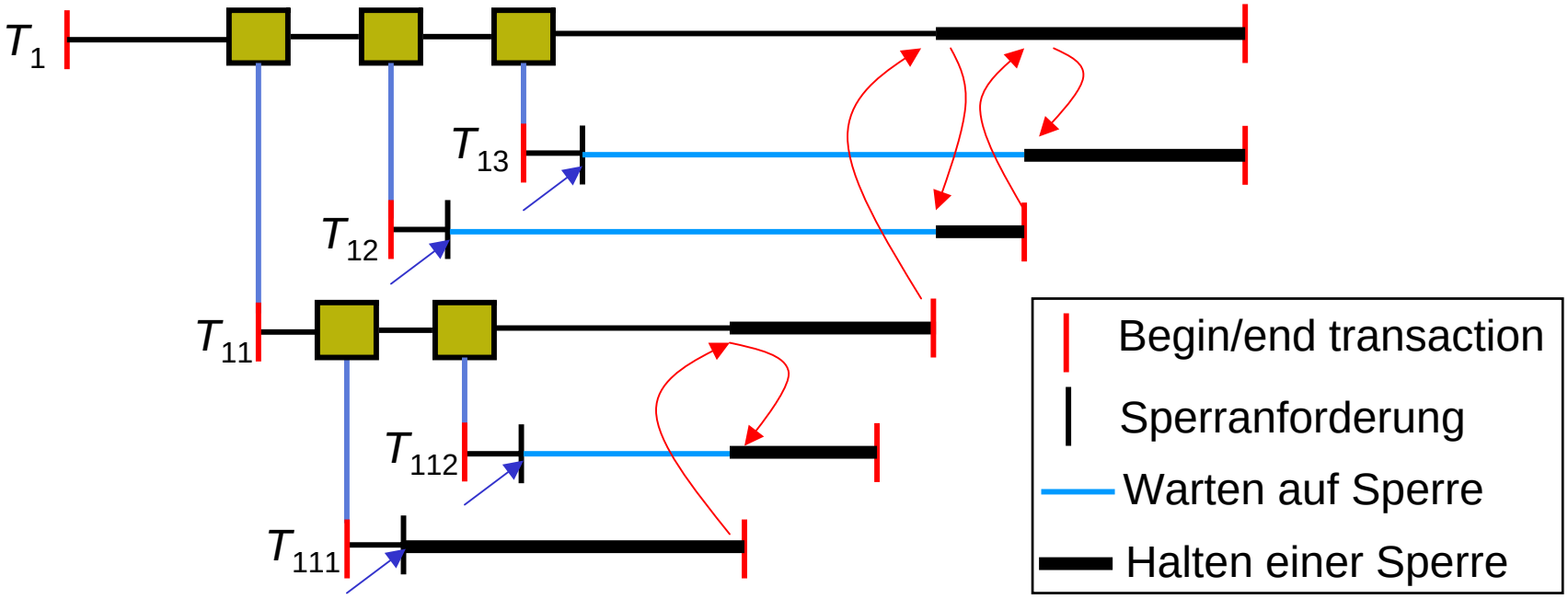
Nebenläufigkeit (2)

- Annahme: Verwendung des 2PL-Protokolls.
- Folge: Jede Subtransaktion gibt ihre Sperren beim Abschluss frei. Aber: Vatertransaktion ist noch nicht abgeschlossen, darf Sperren noch nicht freigeben.
- Lösung: Bei Beendigung einer Subtransaktion werden die Sperren an die Vatertransaktion vererbt. Damit wird die Lebensdauer der Sperren bis zum Ende der Anwendungstransaktion zugesichert.
- Zusätzliche Regel: Sperranforderungen einer Subtransaktion sind mit den gehaltenen Sperren ihrer Vorfahren (nicht jedoch mit denen ihrer möglicherweise parallelen Geschwister) verträglich.

Nebenläufigkeit (3)

Ergebnis: Maximale Nebenläufigkeit unter der Einschränkung, dass Subtransaktionen derselben Vatertransaktion nur nacheinander auf dasselbe Objekt zugreifen.

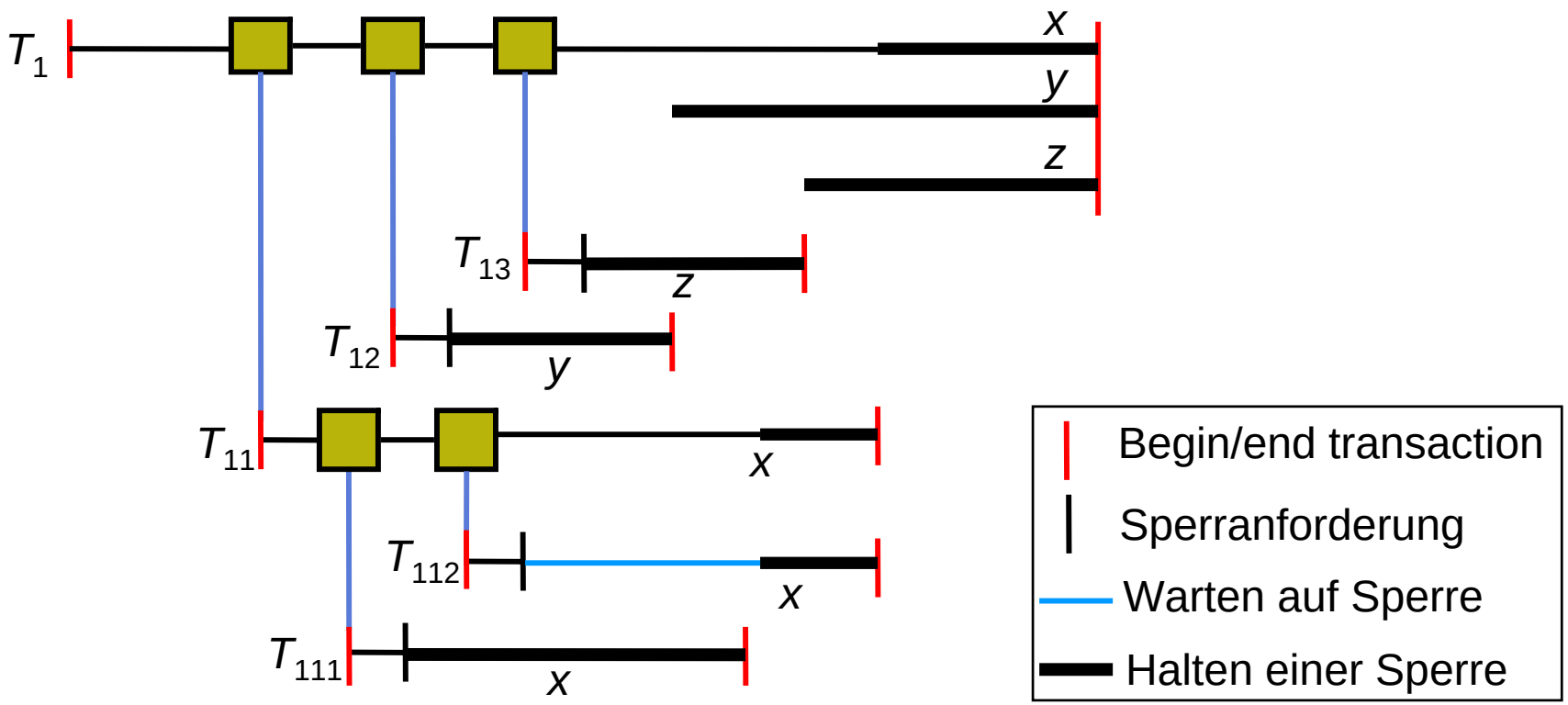
Beispiel: Alle Blatt-Transaktionen schreiben dasselbe Objekt *x*.



Nebenläufigkeit (4)

Ergebnis: Maximale Nebenläufigkeit unter der Einschränkung, dass Subtransaktionen derselben Vatertransaktion nur nacheinander auf dasselbe Objekt zugreifen.

Beispiel: T_{11} schreibt Objekt *x*, T_{12} Objekt *y* und T_{13} Objekt *z*.



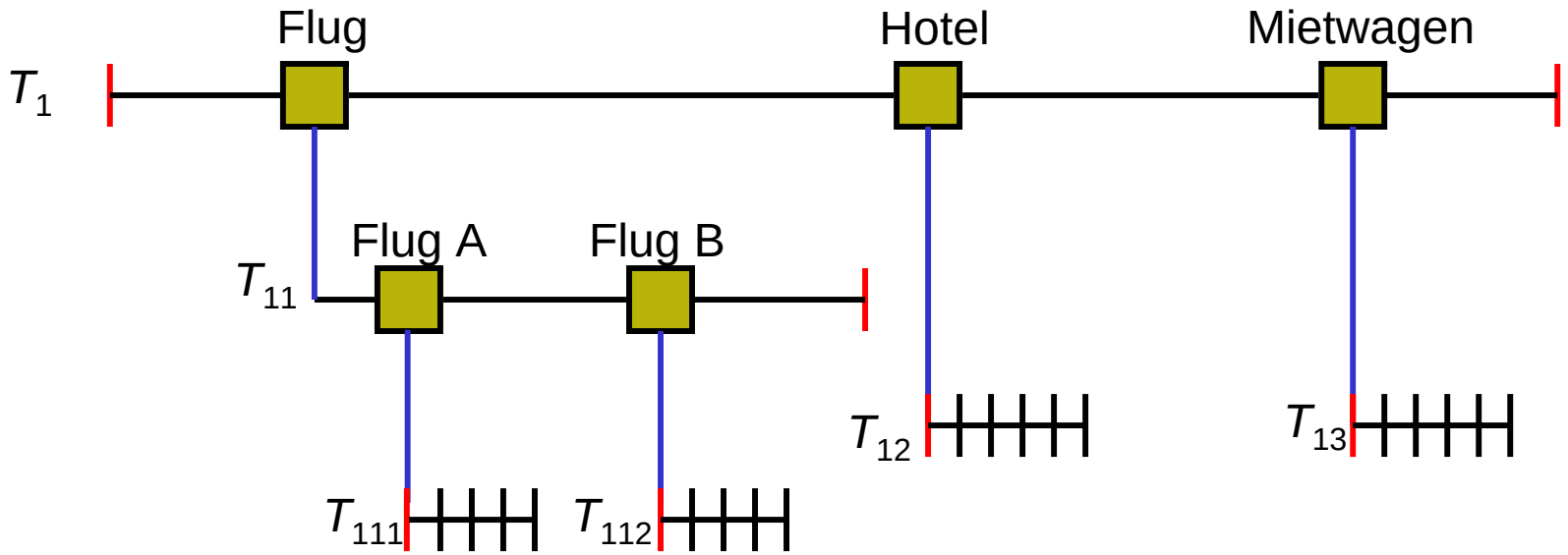
Rücksetzpunkte (1)

Kompensationstransaktionen: Wegen vollständiger Isolation aller Subtransaktionen und der Wurzeltransaktion ist die einfachste Lösung das Rücksetzen auf den Beginn der Transaktion.

Damit ist einheitliche Behandlung von Kompensation und Rücksetzen (fehlgeschlagener Transaktionen) möglich.

⇒ **Regel:** Bei geschlossen geschachtelten Transaktionen stellt der Beginn jeder Subtransaktion (sowie der Beginn der Wurzeltransaktion) einen **Rücksetzpunkt** dar.

Rücksetzpunkte (2)



- Scheitert die Buchung von Flug B, so muss u.U. auch die Buchung von Flug A zurückgenommen werden. Ein geeigneter Rücksetzpunkt ist dann der Beginn von Transaktion T_{11}.
- Soll der Effekt von Transaktion T_{111} hingegen aufrechterhalten werden, so ist der Beginn von T_{112} und damit das Ende von T_{111}der geeignete Rücksetzpunkt.

Rücksetzpunkte (3)

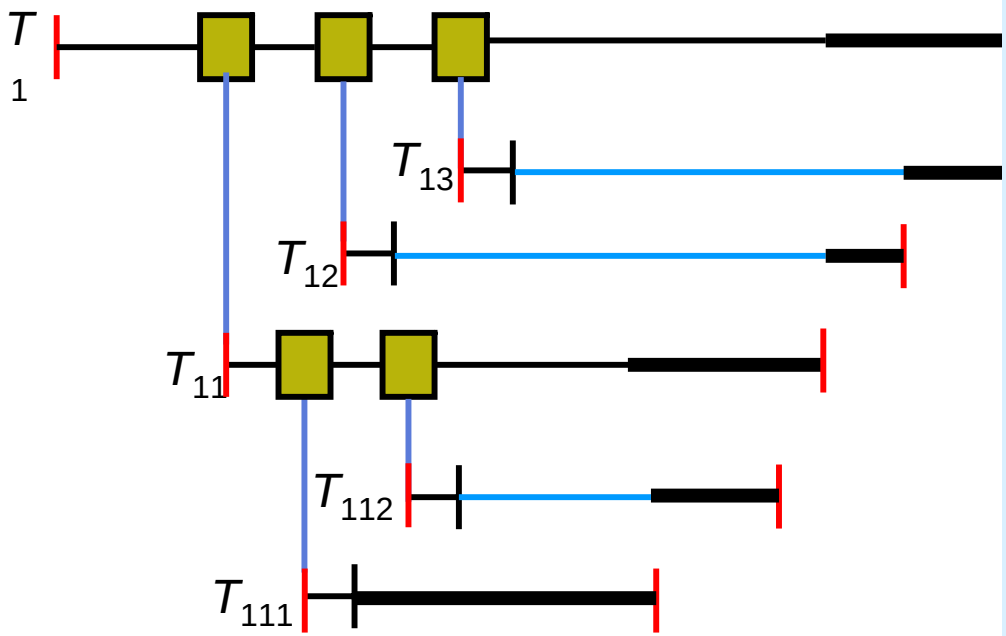
Vorgehen:

- Für jedes von einer Subtransaktion veränderte Objekt wird ein Before-Image angelegt.
- Beim commit einer Subtransaktion werden diese Before-Images an die Vatertransaktion vererbt.
- Die (ererbten) Before-Images (evtl. mehrere!) einer (Vater-) Transaktion müssen so lange gehalten werden, bis die (Vater-)Transaktion beendet ist.

  Grund: Welcher von i.d.R. mehreren möglichen Rücksetzpunkten gewählt wird, ist abhängig von der Anwendungssemantik.

Rücksetzpunkte (4)

All Subtransaktionen T_{111}, T_{112}, T_{12} und T_{13} schreiben das Objekt x. Jeweilige Before-Images: x^{111}, x^{112}, x^{12} und x^{13}.



Zeitpunkt	Before-Images
$b(T_1)$	-
$b(T_{11})$	-
$b(T_{111})$	$x^{111}(T_{111})$
$e(T_{111})$	$x^{111}(T_{11})$
$b(T_{112})$	$x^{111}(T_{11})\,x^{112}(T_{112})$
$e(T_{112})$	$x^{111}(T_{11})\,x^{112}(T_{11})$
$e(T_{11})$	$x^{111}(T_1)$
$b(T_{12})$	$x^{111}(T_1)\,x^{12}(T_{12})$
$e(T_{12})$	$x^{111}(T_1)\,x^{12}(T_1)$
$b(T_{13})$	$x^{111}(T_1)\,x^{12}(T_1)\,x^{13}(T_{13})$
$e(T_{13})$	$x^{111}(T_1)\,x^{12}(T_1)\,x^{13}(T_1)$
$e(T_1)$	-

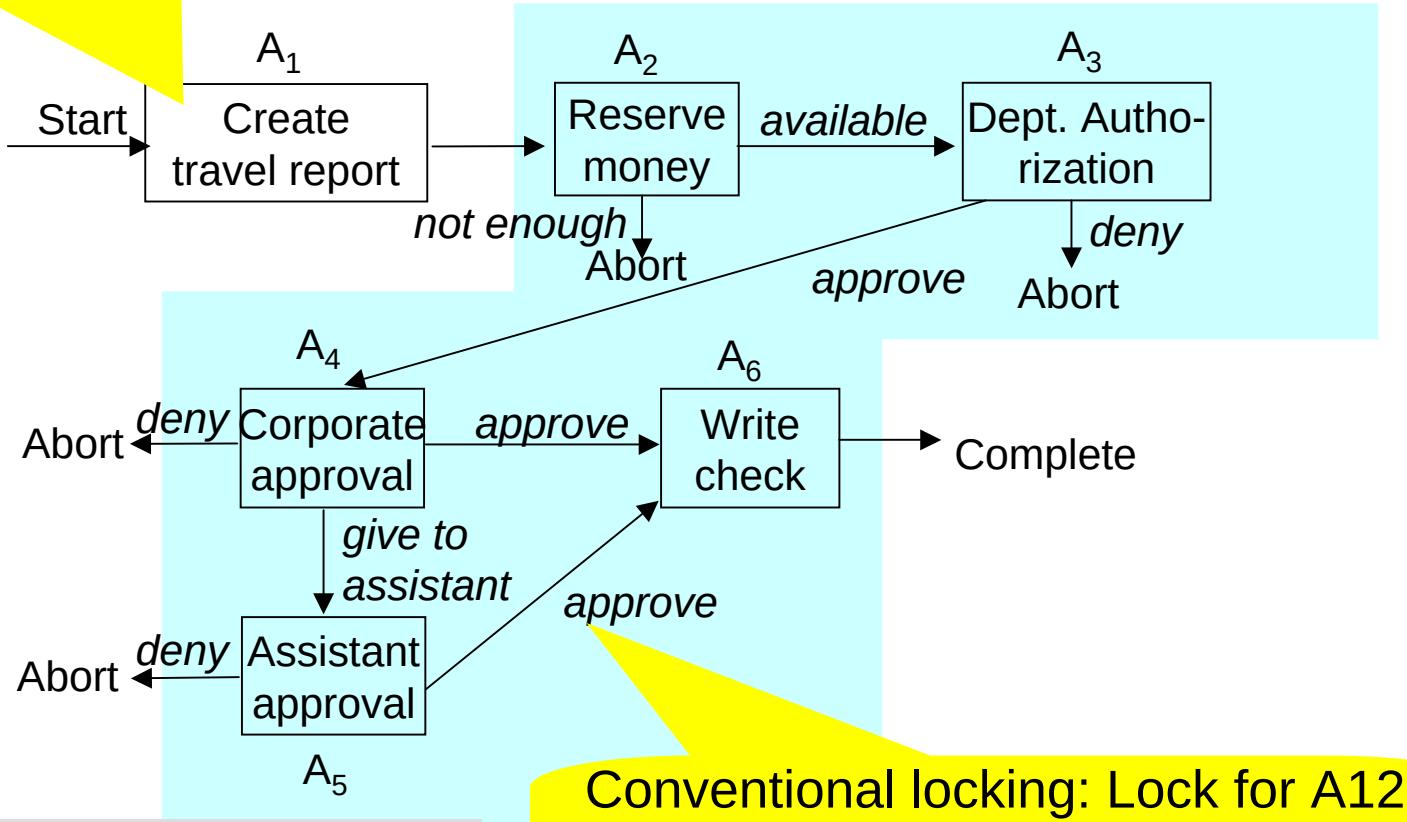
Controlled interaction

Motivation

- Usually, design activities are a collaborative effort with well-defined points of interaction.
- Isolation can then only be maintained over parts of a transaction.
  - Also permits higher degree of internal parallelism.
  - Threatens global and local consistency.
  - Complicates recovery.
- Several known approaches that differ in how well they are suited to different application scenarios.
- We study two general approaches:
  - Open nested transactions.
  - Sagas.

Example

Traveler requests expense reimbursement (USD 1000,--) from Account A123.



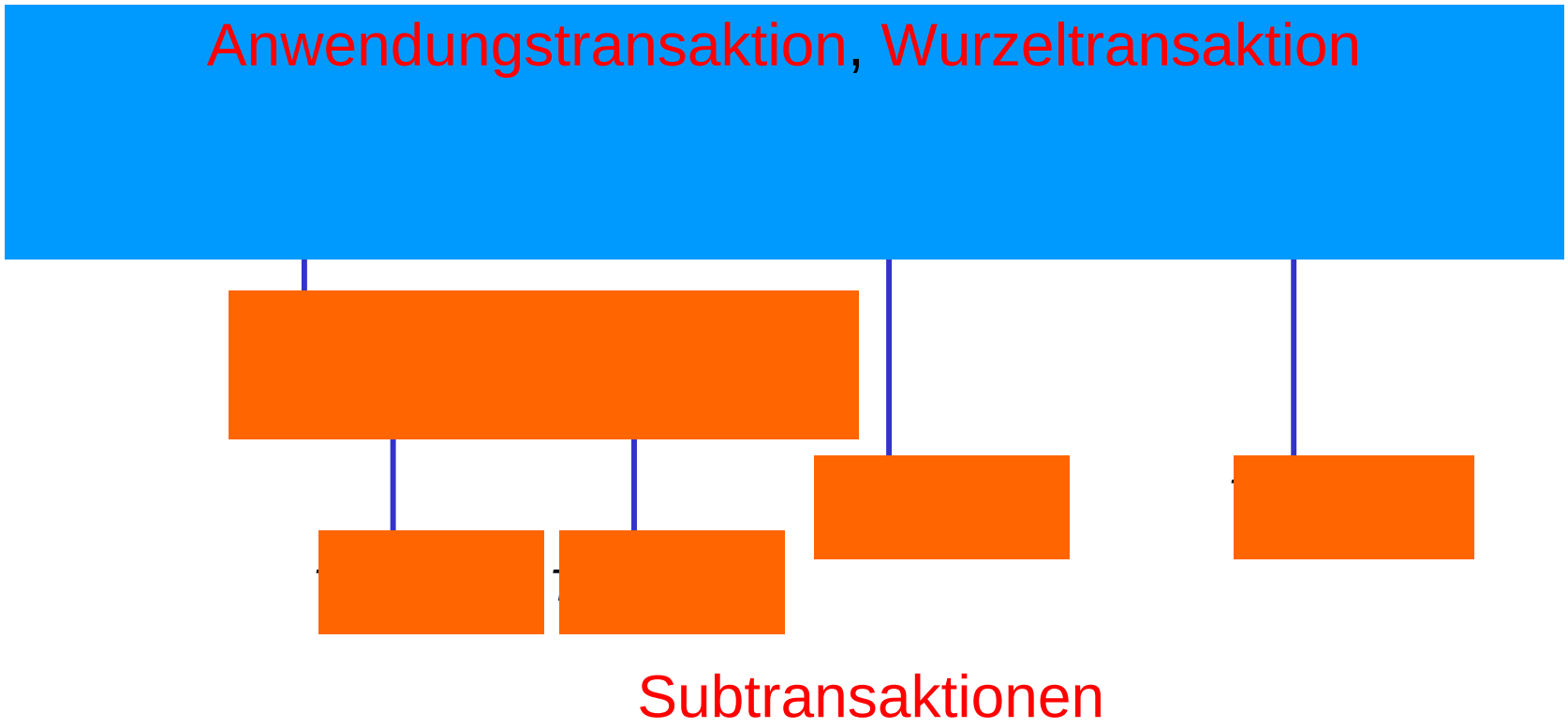
Conventional locking: Lock for A123 may be held for days, blocking everything else.

But: No concurrency control → overdraft.

Geschachtelte Transaktionen

Lösung:

Unverändert: **Transaktionsbaum**.



Isolation / Nebenläufigkeit (1)

Nebenläufigkeit innerhalb der Transaktion:

- Subtransaktionen innerhalb einer Anwendungstransaktion lassen sich parallel bearbeiten (Intratransaktions-Parallelität).
- Es gilt unverändert die Konfliktregelung:

  on

  Anwendungstransaktion

Keine Isolation über Blatt-Transaktionen hinaus.

⇒ Keine Isolation der Gesamttransaktion

⇒ Reduzierte Konsistenz und Isolation schon innerhalb der Transaktion

⇒ Hohe Nebenläufigkeit über alle Transaktionen

Recovery

Wirkung kann bereits sichtbar sein falls nicht Blatt!

Wirkung des gestörten oder fehlgeschlagenen Teils können bereits sichtbar sein!

hrit

Vorgehen soll sich widerspieg

- Blatt-Transaktionen sind ACID.
- Zusammenhänge zwischen Transaktionen entlang eines Pfades:
  - Fehlschlag einer Subtransaktion → Störung, aber Fehlschlag der Vatertransaktion nicht zwingend.

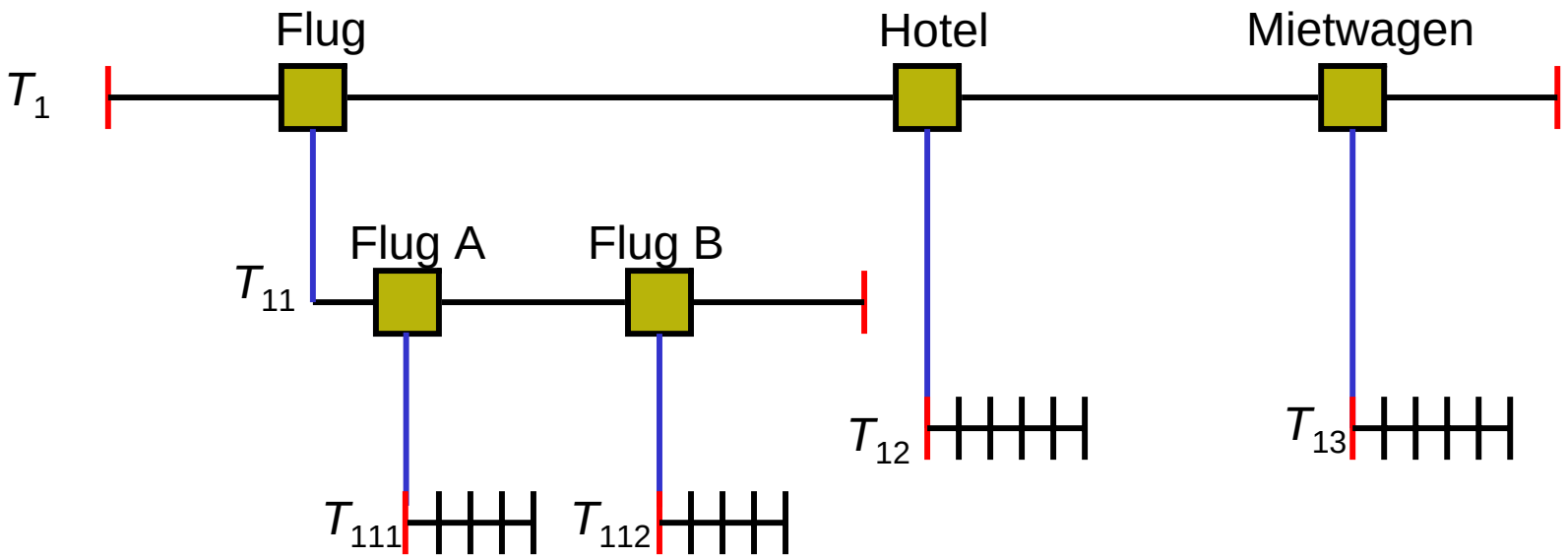
    Vatertransaktion bestimmt *Stelle* in ihrer Ausführung, *ab der* Fortführung möglich ist.
  - Störung in der Vatertransaktion → Fehlschläge aller Subtransaktionen *ab einer* von der Vatertransaktion zu bestimmenden *Stelle*.
  - Fehlschlag der Vatertransaktion → Fehlschlag *aller* bereits ausgeführten Subtransaktionen.

Summary

- Outside appearance: A~~X~~CID

⇒ **Open nested transaction**

Beispiel (1)



Beispiel (2)

- Angenommen Transaktion T_1 buche einen Platz in Flug A. Dieser sei damit ausgebucht. Sie buche dann Flug B.
- Die parallel ablaufende Transaktion T_2 versuche ebenfalls einen Platz in Flug A zu buchen. Dieser erscheint jedoch ausgebucht. T_2 kommt also nicht weiter.
- Wird jetzt allerdings T_1 zurückgesetzt, so stellt sich im Nachhinein heraus, dass Flug A doch nicht ausgebucht gewesen wäre.

Fazit:

Die Serialisierbarkeit der Transaktionen T_1 und T_2 ist also nicht gewährleistet. Der Ablauf der einzelnen Blatt-Transaktionen ist zwar serialisierbar, nicht aber der Ablauf der Wurzeltransaktionen T_1 und T_2.

Isolation / Nebenläufigkeit (2)

Lösungen:

- Subtransaktionen sollten statt der vollständigen Sperrfreigabe Sperren mit einer differenzierteren Bedeutung setzen, um die Isolation wieder zu gewährleisten (**semantische Concurrency-Control**) und trotzdem die Inter-Transaktions-Parallelität nicht unnötig einzuschränken.

Kooperation durch semantische Konfliktrelationen

- Kompensationstransaktionen setzen nicht mehr einfach zurück.

Kooperation durch globale Informierung

Vergleich

Geschlossen geschachtelte Transaktionen

- garantieren Isolation, verhindern somit Erhöhung der Intertransaktionsparallelität,
- haben alle ACID-Eigenschaften, bieten aber intern differenzierte Rücksetzmöglichkeiten.

Offen geschachtelte Transaktionen

- bieten intern differenzierte Rücksetzmöglichkeiten,
- schränken die Isolation und damit die Serialisierbarkeit zu Gunsten einer Kooperation ein,
- bieten eine Grundlage für unterschiedliche Kooperationsmodelle,
- stellen hohe und situationsabhängige Ansprüche an semantische Sperren oder Kompensationstransaktionen.

Systematisches Vorgehen für Kompensationstransaktionen durch einfacheres Modell ⇒ Sagas

Sagas (1)

- **Saga**: Anwendungstransaktion s_i bestehend aus einer linearen Folge von Subtransaktionen t_{ij}:
  - Die Subtransaktionen besitzen ACID-Eigenschaften.
  - Finalzustände der t_{ij} werden nicht nur innerhalb s_i, sondern auch den t_{kl}, $k \neq i$, sichtbar.
  - $\Rightarrow$Die Anwendungstransaktion besitzt keine ACID-Eigenschaften.
- **Folge für Recovery**:
  - Atomizität bleibt erhalten: Saga s_i wird bei Fehlschlag einer t_{ij} in ihrer Gesamtheit zurückgesetzt.
  - Dabei wird t_{ij} wie üblich zurückgesetzt.
  - Die früheren Subtransaktionen bedürfen eines differenzierteren Rücksetzens, da ihr Ergebnisse sichtbar waren.

Sagas (2)

- **Lösung**:
  - Jede Subtransaktion t_{ij} wird mit einer Kompensationstransaktion c_{ij} ausgestattet („Best effort“).
  - Über die c_{ij} werden keine Annahmen gemacht. Insbesondere wird also nicht garantiert, dass der Zustand zu Beginn von t_{ij} wiederhergestellt wird.
  - Bei Rücksetzen werden die c_{ij} in umgekehrter Reihenfolge der t_{ij} ausgeführt.
  - Sei $s = ((t_1,c_1),\ldots,(t_n,c_n))$. Dann wird entweder die Folge

    $t_1,t_2,\ldots,t_n$

    oder die Folge

    $t_1,t_2,\ldots,t_j,c_j,\ldots,c_2,c_1 \ (0 \leq j < n)$

    ausgeführt.

Scheduling

Saga-Verwalter (SM)

- Notiere begin-saga in SM-Log
- Für jede Untertransaktion t_j, notiere Ausführung in SM-Log und reiche t_j weiter an Transaktions-Verwalter (TM).
- Falls Abort-Meldung vom TM, schreibe abort-saga in SM-Log und beende normale Ausführung. Sonst vermerke erfolgreichen Abschluss von t_j.
- Bei erfolgreichem Ende notiere end-saga in SM-Log.

Transaktions-Verwalter (TM)

- Menge der t_{ij} für eine Menge von Sagas s_i bilden wieder eine Historie wie gewohnt.
- TM arbeitet konventionell wie bisher beschrieben. Führt insbesondere sein eigenes TM-Log.

Rücksetzen

TM melde Abort von t_j:

- TM hat bereits t_j konventionell zurückgesetzt.
- Daher anschließend schrittweise Anweisung an TM zur Durchführung von $c_{j-1},\ldots,c_2,c_1$ als konventionelle Transaktionen mit entsprechendem Notieren in SM-Log.
- Problem: Falls eine der c_i einen Fehler aufweist, kann die Saga nicht abgeschlossen werden. Erfordert besondere Zusatzmaßnahmen.

Restart (1)

Backward Recovery:

1. TM führt seine (konventionelle) Recovery durch.
2. SM inspiziert die Einträge im SM-Log
   (a) Es existiert zu *begin-saga* ein *end-saga*: Keine Aktion.
   (b) Es existiert zu *begin-saga* ein *abort-saga*: Sei c_j die Kompensationstransaktion, die vom TM noch nicht als beendet gemeldet war. Dann schrittweise Anweisung an TM zur Durchführung von $c_j,\ldots,c_2,c_1$.
   (c) Andernfalls: Letzte im SM-Log vermerkte Untertransaktion t_j war an TM gegangen. Falls noch nicht als abgeschlossen vermerkt, wurde t_j von TM zurückgesetzt, an den TM sind der Reihe nach $c_{j-1},\ldots,c_2,c_1$ zu senden. Falls als abgeschlossen vermerkt, gehen $c_j,\ldots,c_2,c_1$ an TM.

Restart (2)

Backward/Forward Recovery:

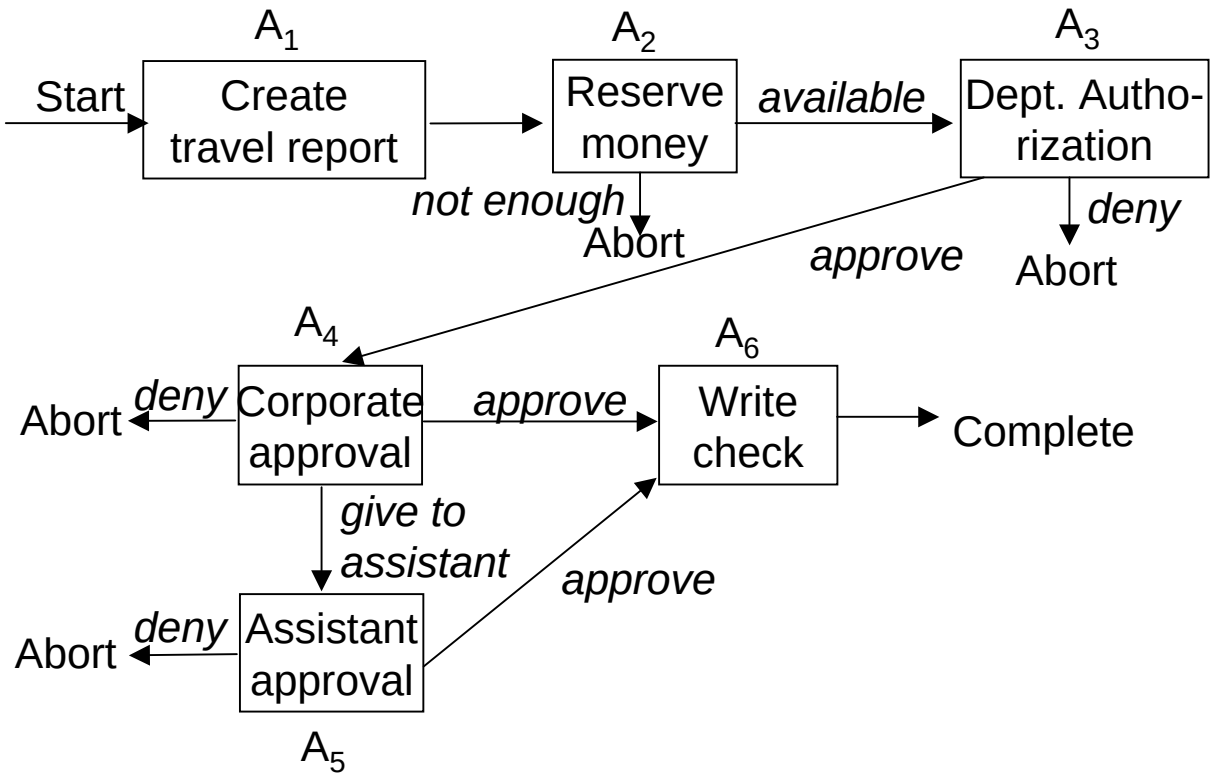
Abwandlung à la geschachtelte Transaktionen für den Fall 2.(c) (weder *end-saga* noch *abort-saga*).

- Es gibt es keinen eigentlichen Grund zum Rücksetzen (eingeschränkte Isolation und Atomizität!), man könnte die Saga auch fortsetzen.
- Erforderlich: Savepoints zwischen einigen oder allen der t_{ij}.
- Sei $s = t_1,t_2,t_3,t_4,\ldots,t_n$ und das System während der Ausführung von t_4 abgestürzt. Existiere Savepoint zwischen t_2 und t_3. Dann Restart-Abfolge (über das SM-Log sicherzustellen):

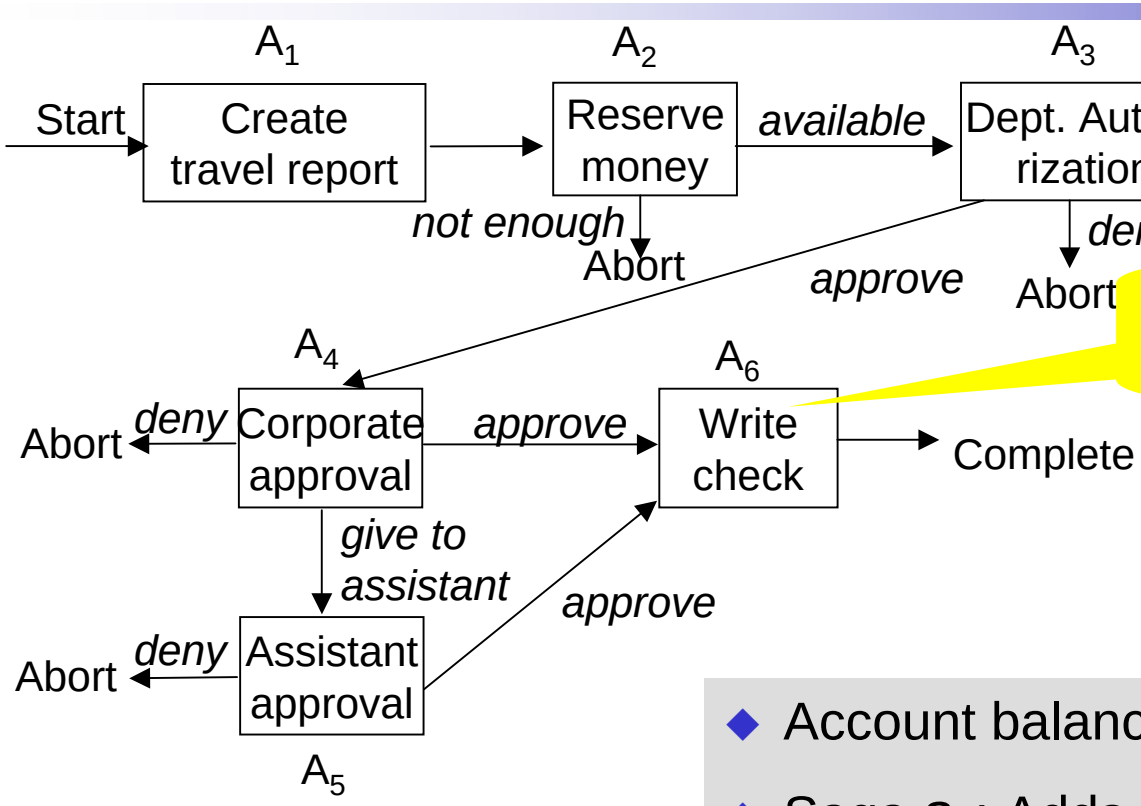
  c_3, *restore-savepoint*,$t_3,t_4,\ldots,t_n$

Sagas (3)

Generalization to graphs is possible:



Compensation



How would one compensate for a real-world action?

- Account balance is USD 2000,--.
- Saga s_1: Adds USD 1000,-- to account.
- Saga s_2: Deletes USD 2500,-- from account.
- Rollback of s_1: **Account goes negative.**
- Compensation should know of s_2.