Beispiel-Transaktion

1. Lese den Kontostand von A in die Variable a: **read**(A,a);
2. Reduziere den Kontostand um 50,– DM: $a := a - 50$;
3. Schreibe den neuen Kontostand in die Datenbasis: **write**(A,a);
4. Lese den Kontostand von B in die Variable b: **read**(B,b);
5. Erhöhe den Kontostand um 50,– DM: $b := b + 50$;
6. Schreibe den neuen Kontostand in die Datenbasis: **write**(B,b);

Anforderungen an die Transaktionsverwaltung

- gleichzeitig (nebenläufig) ablaufende Transaktionen
- Synchronisation
- Datenbanken gegen Soft- und Hardwarefehler schützen
- Abgeschlossene Transaktionen müssen erhalten bleiben
- Nicht abgeschlossene Transaktionen müssen vollständig revidiert (zurückgesetzt) werden.

Operationen auf Transaktions-Ebene

- **begin of transaction** (**BOT**): Mit diesem Befehl wird der Beginn einer eine Transaktion darstellenden Befehlsfolge gekennzeichnet.
- **commit**:
  - Alle Änderungen der Datenbasis werden durch diesen Befehl *festgeschrieben*,
  - sie werden **dauerhaft** in die Datenbank eingebaut.
- **abort**:
  - Selbstabbruch der Transaktion
  - Datenbanksystem muß sicherstellen, daß die Datenbasis wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.
- **define savepoint**:
  - Sicherungspunkt definiert, auf den sich die (noch aktive) Transaktion zurücksetzen läßt.
- **backup transaction**:
  - die noch aktive Transaktion wird auf den jüngsten – also den zuletzt angelegten – Sicherungspunkt zurückgesetzt.
  - evtl. ist auch ein Rücksetzen auf weiter zurückliegende Sicherungspunkte möglich.

Abschluß einer Transaktion

1. erfolgreicher Abschluß durch ein **commit**
2. erfolgloser Abschluß durch ein **abort** oder durch Fehler

BOT

op_1

op_2

$\vdots$

op_n

commit

BOT

op_1

op_2

$\vdots$

op_j

abort

BOT

op_1

op_2

$\vdots$

op_k

$\sim\sim\sim\sim\sim\sim$ Fehler

ACID-Paradigma

Atomicity (Atomarität)

- Transaktion ist kleinste, nicht mehr weiter zerlegbare Einheit
- Entweder werden alle Änderungen der Transaktion festgeschrieben oder gar keine
- Man kann sich dies auch als „alles-oder-nichts“-Prinzip merken

Consistency

- Transaktion hinterläßt einen konsistenten Datenbasiszustand
- Anderenfalls wird sie komplett (siehe *Atomarität*) zurückgesetzt
- Zwischenzustände während der TA-Bearbeitung dürfen inkonsistent sein
- Endzustand muß die im Schema definierten Konsistenzbedingungen (z.B. referentielle Integrität) erfüllen

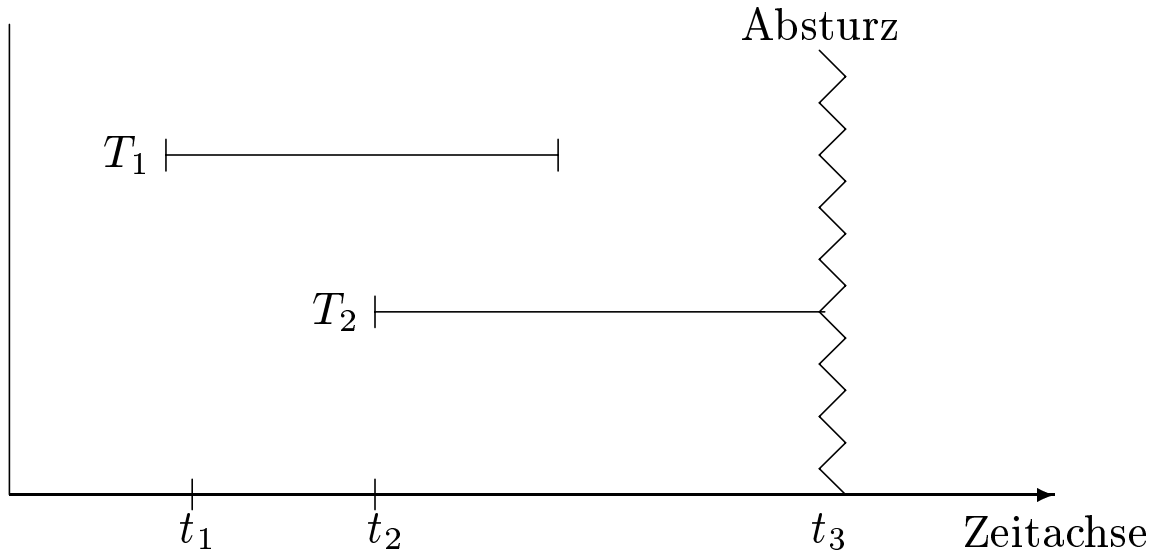
Isolation

- nebenläufig (parallel, gleichzeitig) ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen
- alle anderen parallel ausgeführten Transaktionen bzw. deren Effekte dürfen nicht sichtbar sein

Durability (Dauerhaftigkeit)

- Wirkung einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten
- Transaktionsverwaltung muß sicherstellen, daß dies auch nach einem Systemfehler (Hardware oder Systemsoftware) gewährleistet ist
- Wirkungen einer einmal erfolgreich abgeschlossenen Transaktion kann nur durch eine sogenannte kompensierende Transaktion aufgehoben werden

Transaktionsbeginn und -ende



1. Die Wirkungen der zum Zeitpunkt t_3 abgeschlossenen Transaktion T_1 müssen in der Datenbasis vonhanden sein.

2. Die Wirkungen der zum Zeitpunkt des Systemabsturzes noch nicht abgeschlossenen Transaktion T_2 müssen vollständig aus der Datenbasis entfernt sein. Diese Transaktion kann man nur durch ein erneutes Starten durchführen.

Transaktionsverwaltung in SQL

- **commit work**:
  - Änderungen werden – falls keine Konsistenzverletzungen oder andere Probleme aufgedeckt werden – festgeschrieben.
  - Das Schlüsselwort **work** ist optional,
  - d.h. das Transaktionsende kann auch einfach mit **commit** „befohlen“ werden.
- **rollback work**:
  - Alle Änderungen sollen zurückgesetzt werden.
  - Anders als der **commit**-Befehl muß das DBMS die „erfolgreiche“ Ausführung eines **rollback**-Befehls immer garantieren können.
- **Beispiel-Transaktion**

insert into Vorlesungen
    values (5275, 'Kernphysik', 3, 2141);

insert into Professoren
    values (2141, 'Meitner', 'C4', 205);

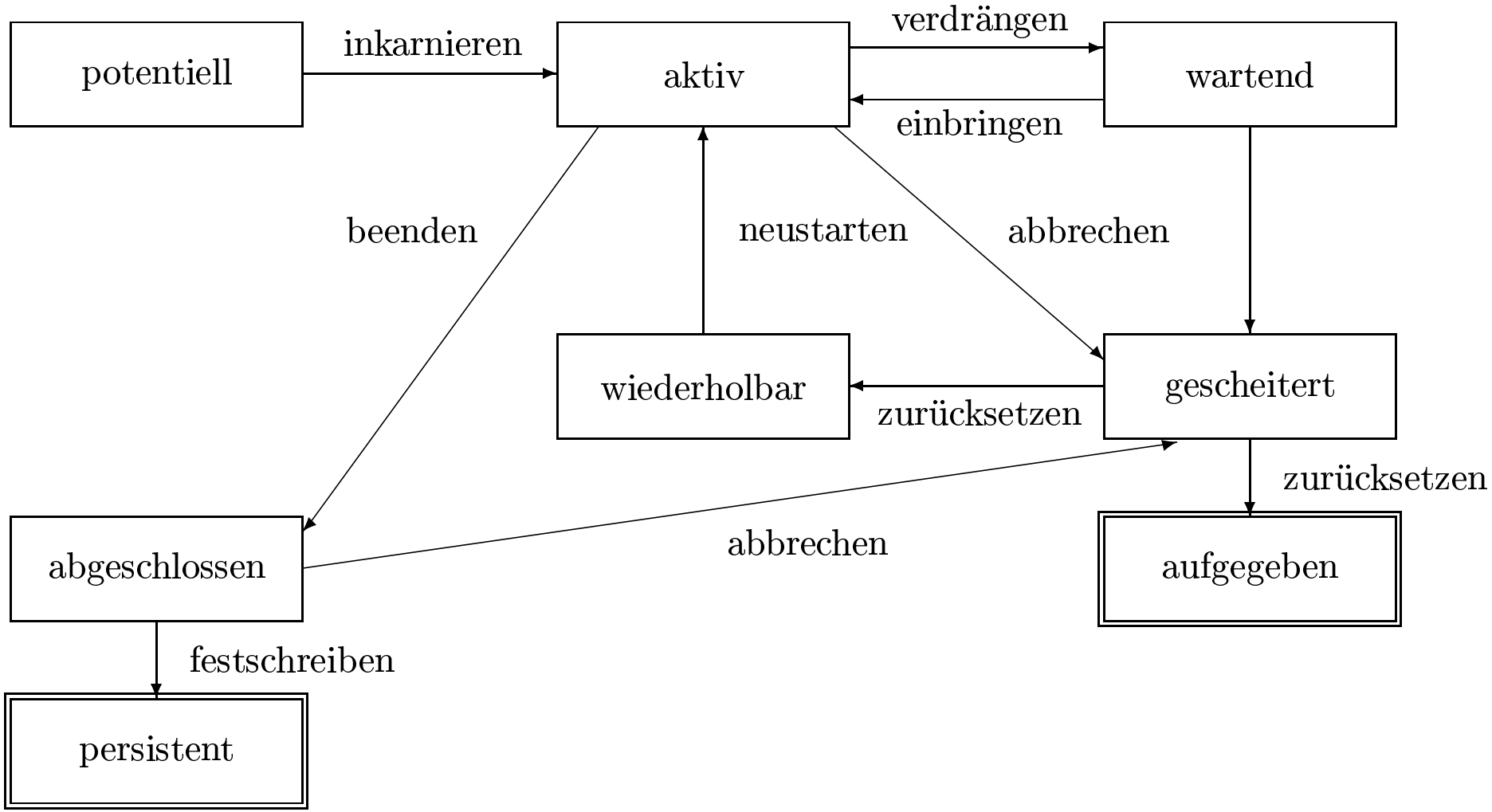
commit work

- Man beachte: ein **commit**-Versuch nach dem ersten **insert** könnte nicht erfolgreich durchgeführt werden, da zu diesem Zeitpunkt die referentielle Integrität verletzt ist.

Zustandsübergänge einer Transaktion

- *potentiell*:
- *aktiv*:
- *wartend*:
- *abgeschlossen*:
- *persistent*:
- *gescheitert*:
- *wiederholbar*:
- *aufgegeben*:

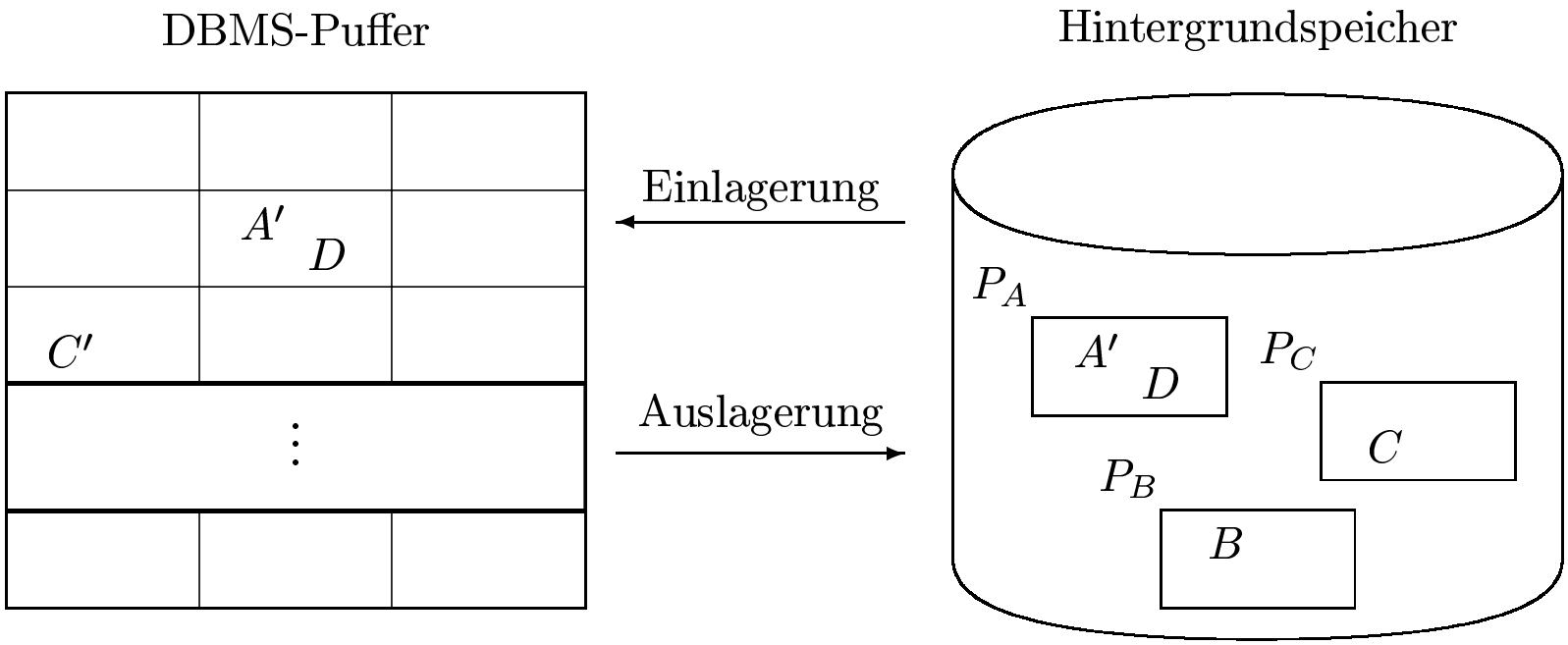
Zustandsübergangs-Diagramm für Transaktionen



Fehlerklassifikation

1. Lokaler Fehler in einer noch nicht festgeschriebenen (committed) Transaktion
   - Wirkung muß zurückgesetzt werden
   - $R1$-Recovery
2. Fehler mit Hauptspeicherverlust
   - abgeschlossene TAs müssen erhalten bleiben (R2-Recovery)
   - noch nicht abgeschlossene TAs müssen zurückgesetzt werden (R3-Recovery)
3. Fehler mit Hintergrundspeicherverlust
   - R4-Recovery

Zweistufige Speicherhierarchie



Ersetzung von Puffer-Seiten

- ¬*steal*: Bei dieser Strategie wird die Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ausgeschlossen.
- *steal*: Jede nicht fixierte Seite ist prinzipiell ein Kandidat für die Ersetzung, falls neue Seiten eingelagert werden müssen.

Einbringen von Änderungen abgeschlossener TAs

- *force*-Strategie: Änderungen werden zum Transaktionsende auf den Hintergrundspeicher geschrieben.
- ¬*force*-Strategie: geänderte Seiten können im Puffer verbleiben.

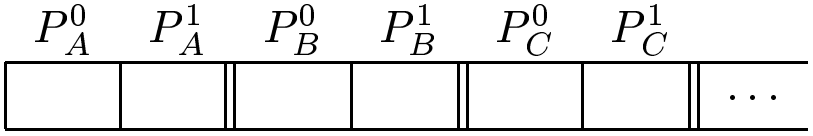
Auswirkung auf Recovery

	force	¬force
¬steal	• kein Redo • kein Undo	• Redo • kein Undo
steal	• kein Redo • Undo	• Redo • Undo

Update in Place

- jede Seite hat genau eine „Heimat“ auf dem Hintergrundspeicher
- der alte Zustand der Seite wird überschrieben

Twin-Block-Verfahren
Anordnung der Seiten P_A, P_B und P_C.



Schattenspeicherkonzept

- nur geänderte Seiten werden dupliziert
- weniger Redundanz als beim Twin-Block-Verfahren

Hier zugrunde gelegte Systemkonfiguration

- *steal*:
  - „dreckige Seiten“ können in die Datenbank (auf Platte) geschrieben werden

- $\neg$*force*:
  - geänderte Seiten sind möglicherweise noch nicht auf die Platte geschrieben

- *update-in-place*:
  - Es gibt von jeder Seite nur eine Kopie auf der Platte

- *Kleine Sperrgranulate*:
  - auf Satzebene
  - also kann eine Seite gleichzeitig „dreckige“ Daten (einer noch nicht abgeschlossenen TA) und „commited updates“ enthalten
  - das gilt sowohl für Puffer – als auch Datenbankseiten

Struktur der Log-Einträge

[LSN, TransaktionsID, PageID, Redo, Undo, PrevLSN]

- *LSN* (*Log Sequence Number*),
  - eine eindeutige Kennung des Log-Eintrags.
  - *LSNs* müssen monoton aufsteigend vergeben werden,
  - die chronologische Reihenfolge der Protokolleinträge kann dadurch ermittelt werden.
- *Transaktionskennung TA* der Transaktion, die die Änderung durchgeführt hat.
- *PageID*
  - die Kennung der Seite, auf der die Änderungsoperation vollzogen wurde.
  - Wenn eine Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Einträge generiert werden.

- Die *Redo*-Information gibt an, wie die Änderung nachvollzogen werden kann.
- Die *Undo*-Information beschreibt, wie die Änderung rückgängig gemacht werden kann.
- *PrevLSN*, einen Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion. Diesen Eintrag benötigt man aus Effizienzgründen.

Beispiel einer Log-Datei

Schritt	T_1	T_2	Log
			[LSN,TA,PageID,Redo,Undo,PrevLSN]
1.	**BOT**		[#1, T_1,**BOT**, 0]
2.	$r(A, a_1)$		
3.		**BOT**	[#2, T_2, **BOT**, 0]
4.		$r(C, c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A, a_1)$		[#3, T_1, P_A, A–=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		$w(C, c_2)$	[#4, T_2, P_C, C+=100, C–=100, #2]
9.	$r(B, b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B, b_1)$		[#5, T_1, P_B, B+=50, B–=50, #3]
12.	**commit**		[#6, T_1, **commit**, #5]
13.		$r(A, a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A, a_2)$	[#7, T_2, P_A, A–=100, A+=100, #4]
16.		**commit**	[#8, T_2, **commit**, #7]

Physische Protokollierung

Es werden Inhalte/Zustände protokolliert:

1. **before-image** enthält den Zustand vor Ausführung der Operation
2. **after-image** enthält den Zustand nach Ausführung der Operation

Logische Protokollierung

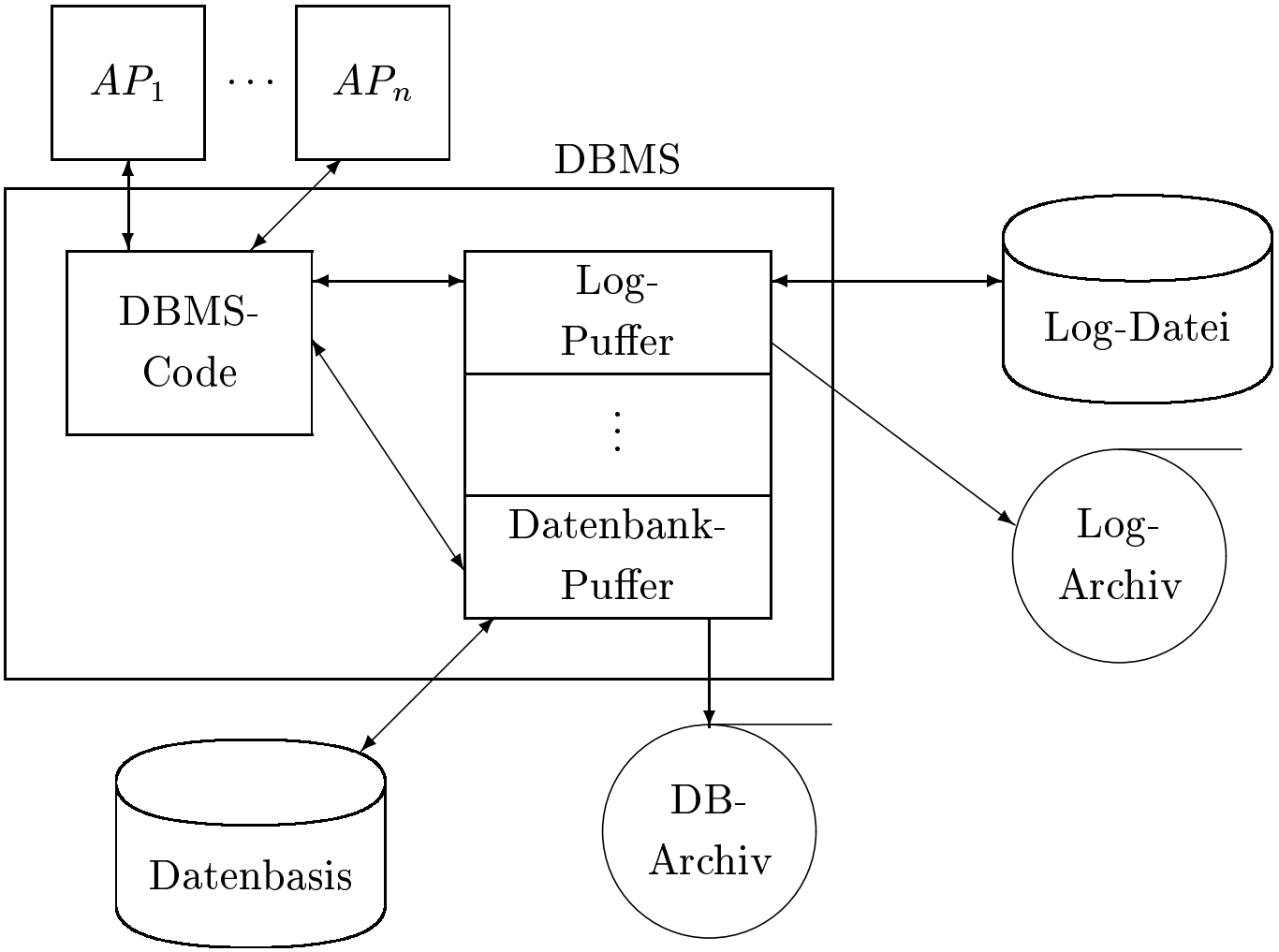
- das *Before-Image* wird durch Ausführung des *Undo*-Codes aus dem *After-Image* generiert und
- das *After-Image* durch Ausführung des *Redo*-Codes aus dem *Before-Image* berechnet.

Speicherung der Seiten-LSN

Die "Herausforderung" besteht darin, beim Wiederanlauf zu entscheiden, ob man das Before- oder das After-Image auf dem Hintergrundspeicher vorgefunden hat.

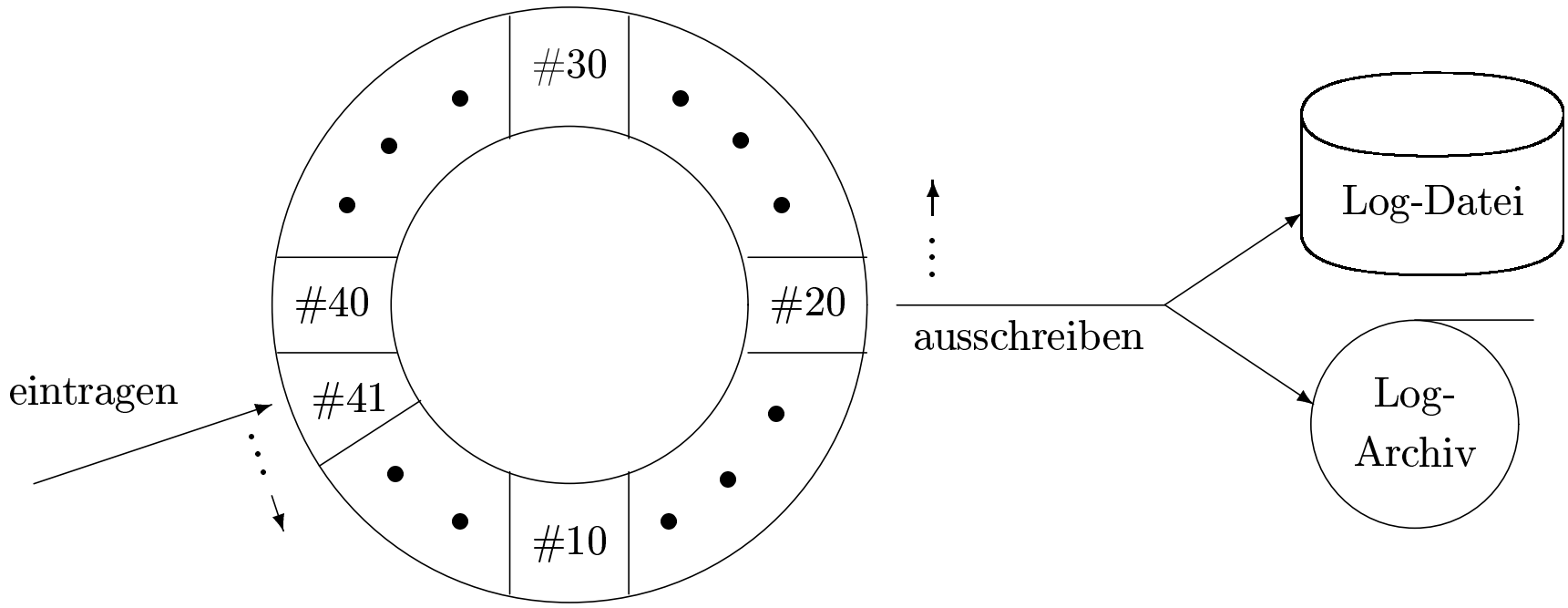
Dazu wird auf jeder Seite die LSN des jüngsten diese Seite betreffenden Log-Eintrags gespeichert.

Schreiben der Log-Information



- Die Log-Information wird zweimal geschrieben
  1. Log-Datei für schnellen Zugriff
     - R1, R2 und R3-Recovery
  2. Log-Archiv
     - R4-Recovery

Anordnung des Log-Ringpuffers

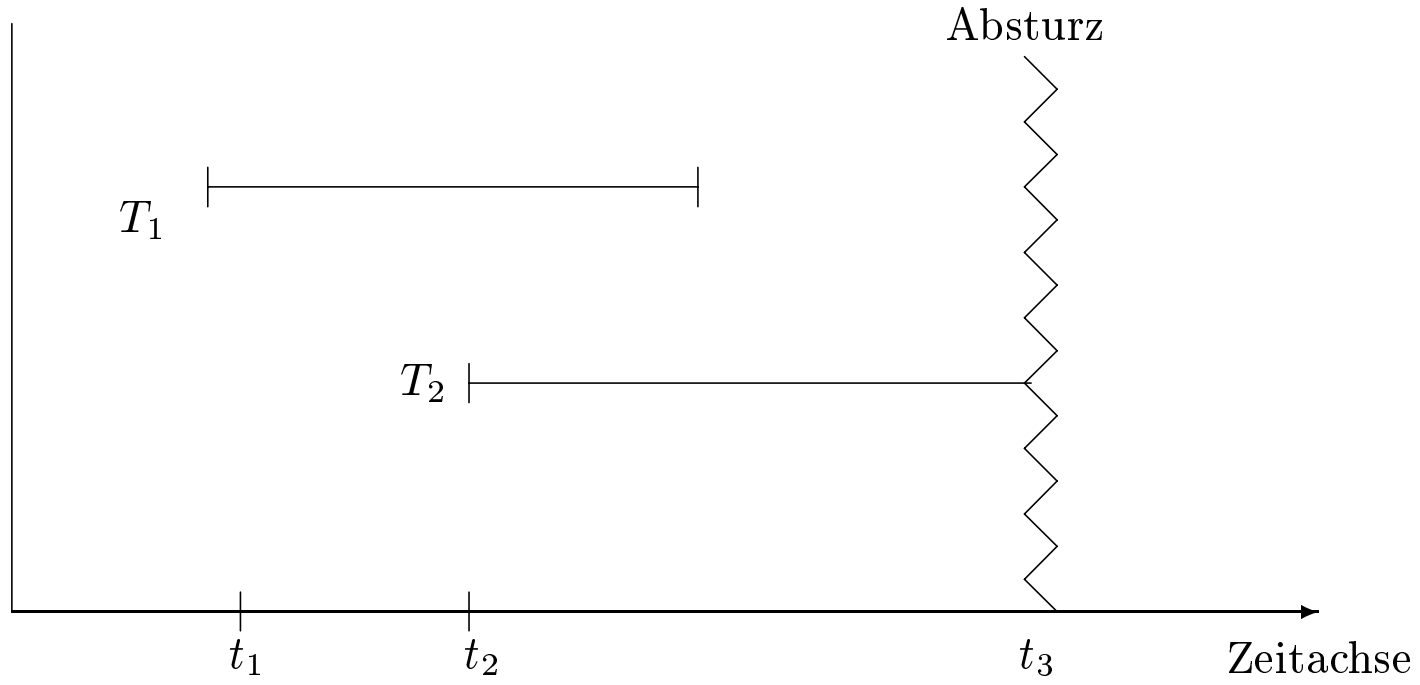


Write Ahead Log-Prinzip

1. Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle „zu ihr gehörenden“ Log-Einträge ausgeschrieben werden.

2. Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das temporäre und das Log-Archiv ausgeschrieben werden.

Transaktionsbeginn und -ende relativ zu einem Systemabsturz

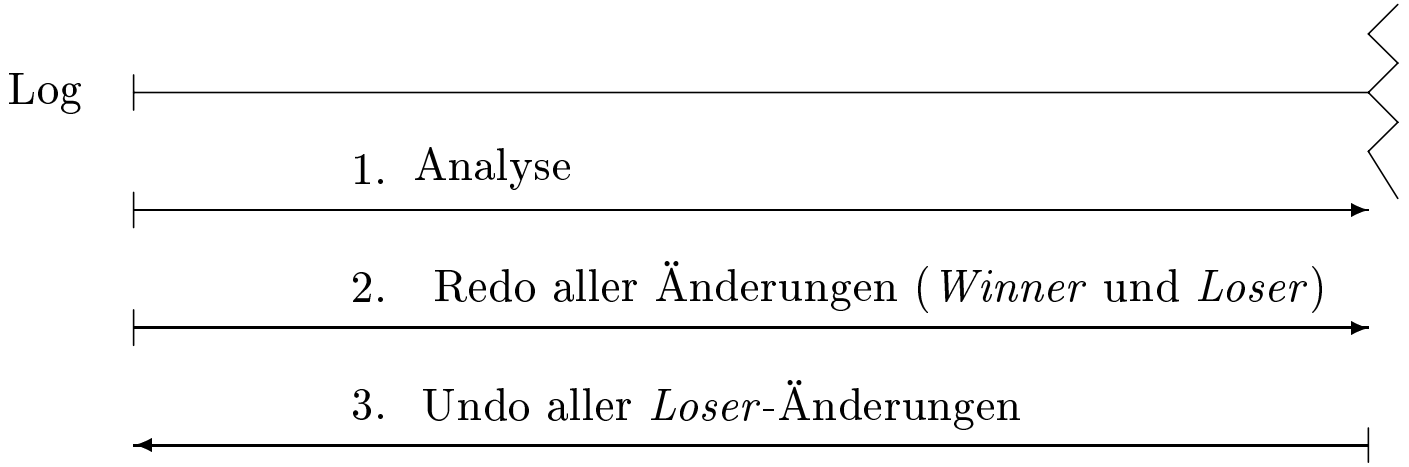


- Transaktionen der Art T_1 müssen hinsichlich ihrer Wirkung vollständig nachvollzogen werden. Transaktionen dieser Art nennt man *Winner*.
- Transaktionen, die wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese Transaktionen bezeichnen wir als *Loser*.

Drei Phasen des Wiederanlaufs

1. *Analyse*:
   - Die temporäre Log-Datei wird von Anfang bis Ende analysiert,
   - Ermittlung der *Winner*-Menge von Transaktionen des Typs T_1
   - Ermittlung der *Loser*-Menge von Transaktionen der Art T_2.
2. *Wiederholung der Historie*:
   - *alle* protokollierten Änderungen werden in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht.
3. *Undo der Loser*:
   - Die Änderungoperationen der *Loser*-Transaktionen werden in umgekehrter Reihenfoge ihrer ursprünglichen Ausführung rückgängig gemacht.

Wiederanlauf in drei Phasen

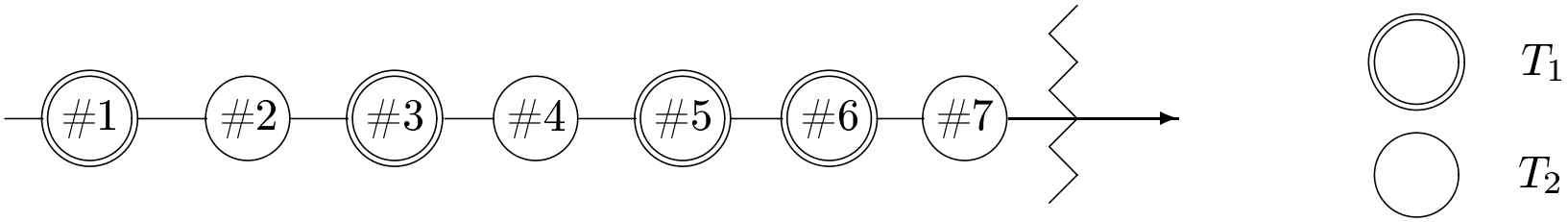


Fehlertoleranz (Idempotenz) des Wiederanlaufs

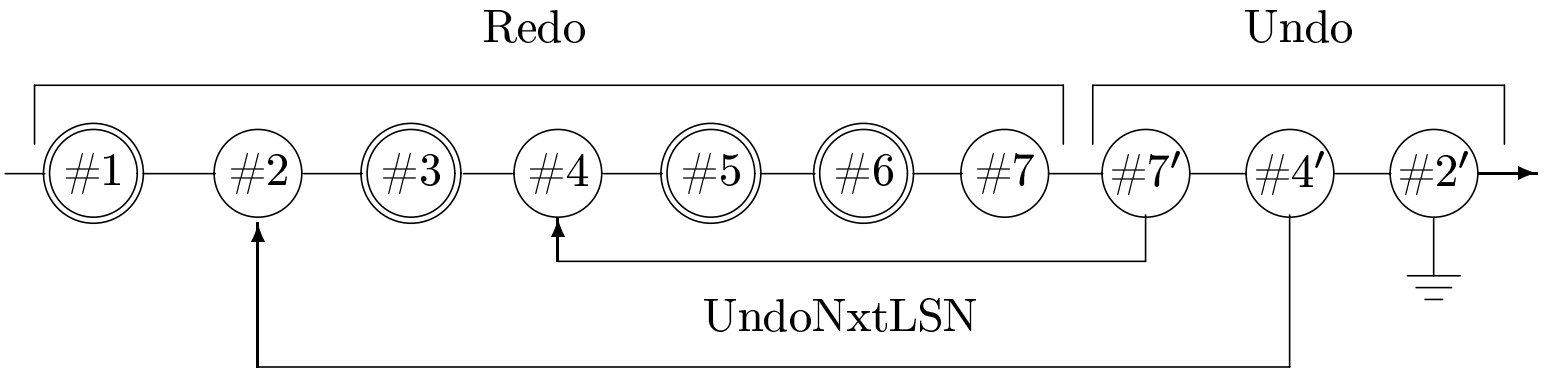
$$undo(undo(\cdots(undo(a))\cdots)) = undo(a)$$
$$redo(redo(\cdots(redo(a))\cdots)) = redo(a)$$

- auch während der Recoveryphase kann des System abstürzen

Kompensationseinträge im Log



Wiederanlauf und Log



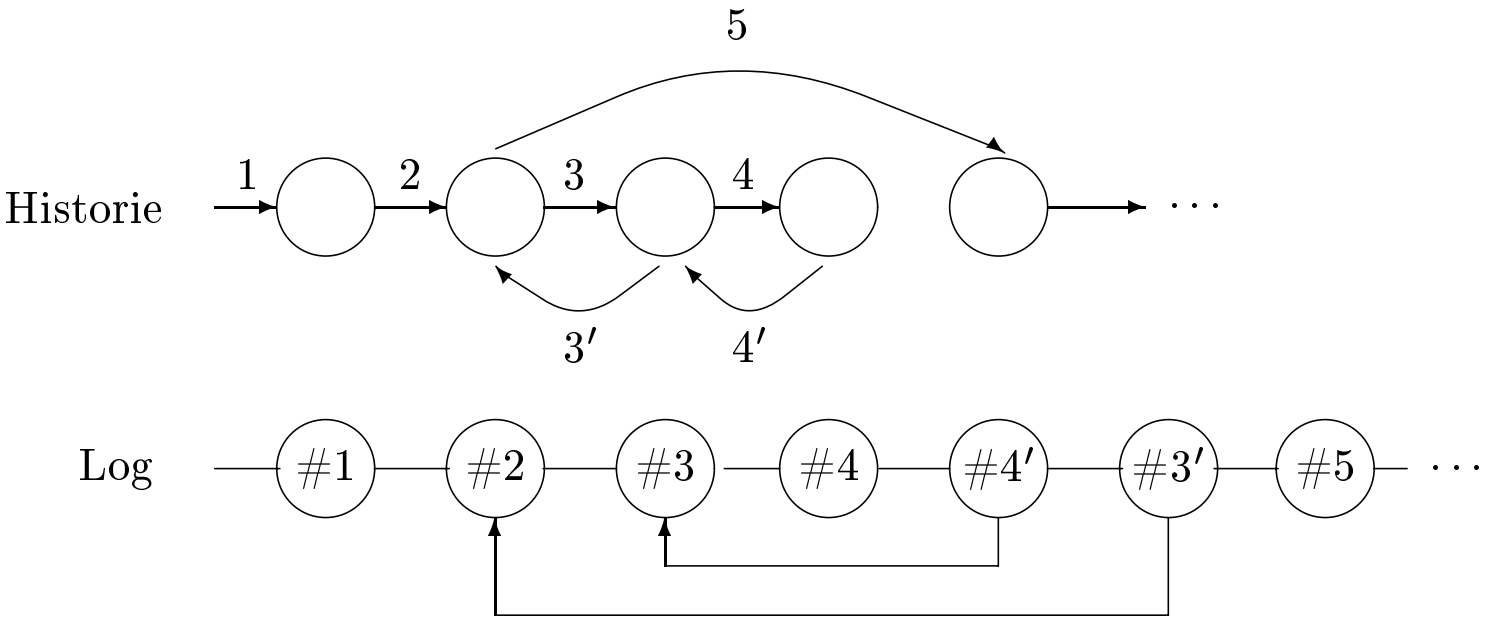
- Kompensationseinträge (CLR: compensating log record) für rückgängig gemachte Änderungen.
  - #7 ist CLR für #7
  - #4 ist CLR für #4

Logeinträge nach abgeschlossenem Wiederanlauf

$$[\#1, T_1, \mathbf{BOT}, 0]$$
$$[\#2, T_2, \mathbf{BOT}, 0]$$
$$[\#3, T_1, P_A, A{-}{=}50, A{+}{=}50, \#1]$$
$$[\#4, T_2, P_C, C{+}{=}100, C{-}{=}100, \#2]$$
$$[\#5, T_1, P_B, B{+}{=}50, B{-}{=}50, \#3]$$
$$[\#6, T_1, \mathbf{commit}, \#5]$$
$$[\#7, T_2, P_A, A{-}{=}100, A{+}{=}100, \#4]$$
$$\langle \#7', T_2, P_A, A{+}{=}100, \#7, \#4 \rangle$$
$$\langle \#4', T_2, P_C, C{-}{=}100, \#7', \#2 \rangle$$
$$\langle \#2', T_2, -, -, \#4', 0 \rangle$$

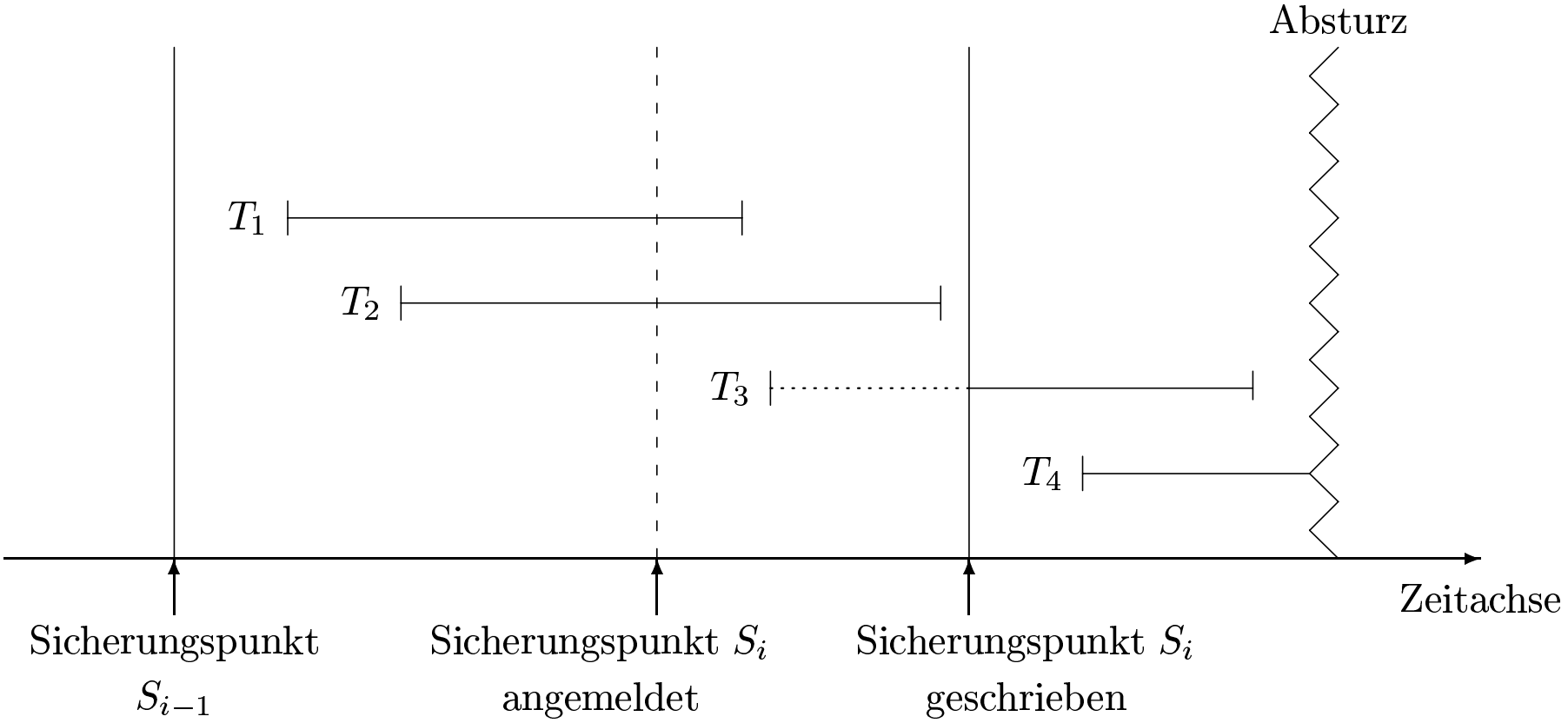
- CLRs sind durch spitze Klammern $\langle \ldots \rangle$ gekennzeichnet.
- der Aufbau einer CLR ist wie folgt
  - LSN
  - TA-Identifikator
  - betroffene Seite
  - Redo-Information
  - PrevLSN
  - UndoNxtLSN (Verweis auf die nächste rückgängig zu machende Änderung)
- CLRs enthalten keine Undo-Information
  - warum nicht?

Partielles Zurücksetzen einer Transaktion

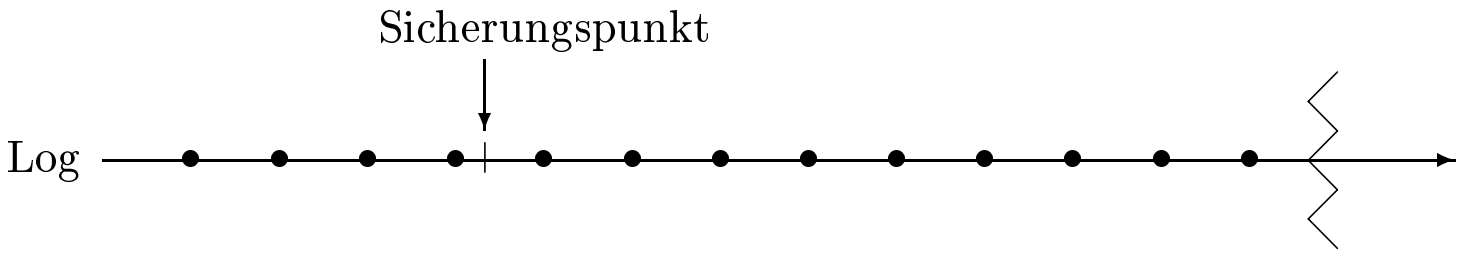


- Schritte 3 und 4 werden zurückgenommen
- notwendig für die Realisierung von Sicherungspunkten innerhalb einer TA

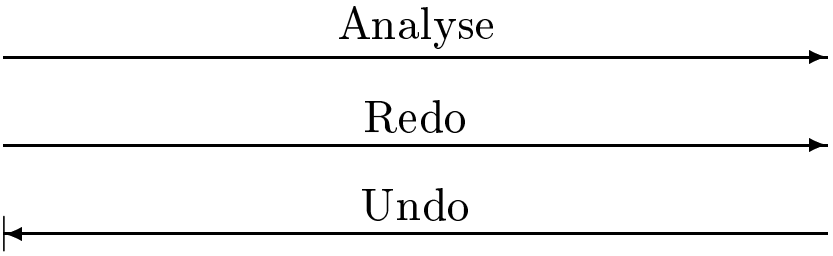
Transaktionskonsistente Sicherungspunkte



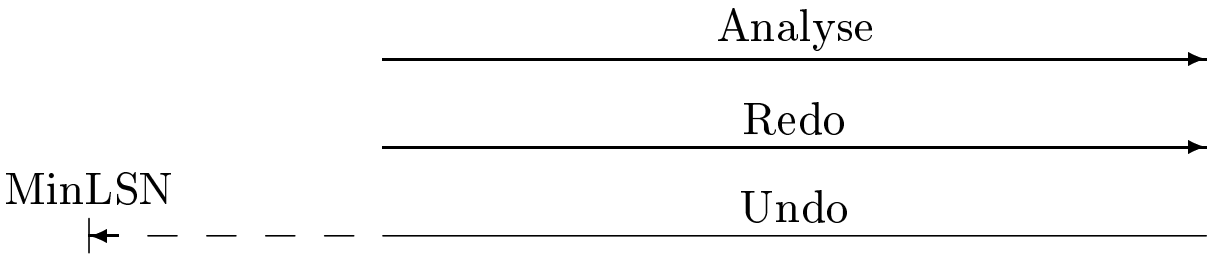
Drei unterschiedliche Sicherungspunkt-Qualitäten



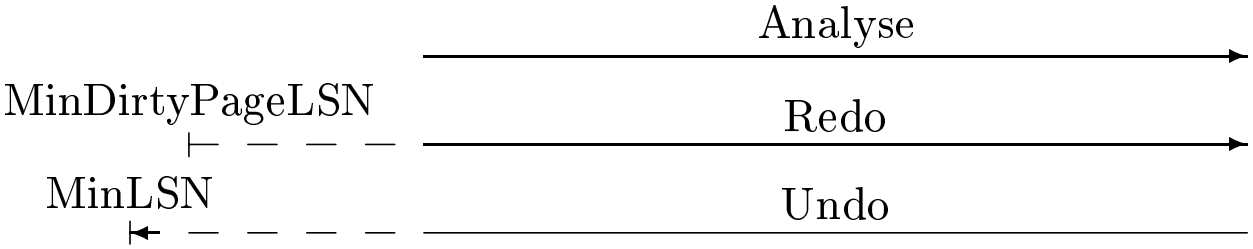
(a) **transaktionskonsistent**



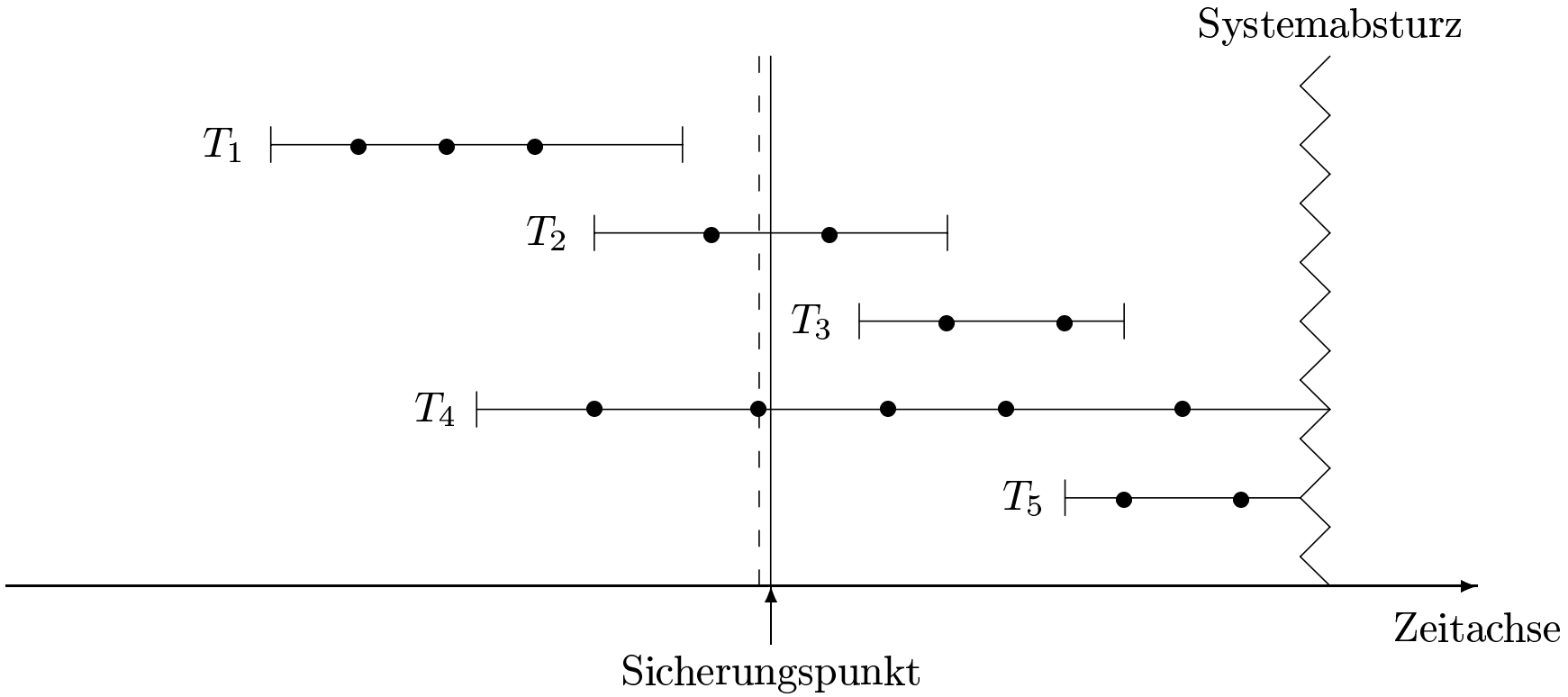
(b) **aktionskonsistent**



(c) **unscharf (fuzzy)**



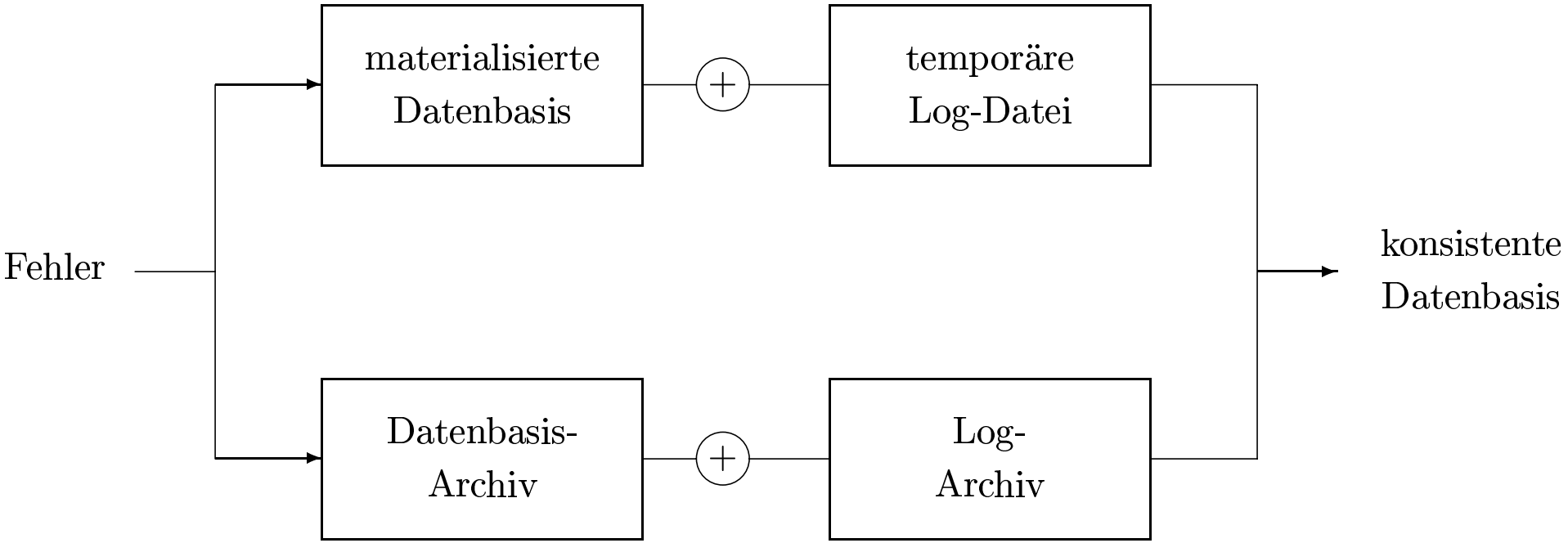
Transaktionsausführung relativ zu einem aktionskonsistenten Sicherungspunkt und einem Systemabsturz



Unscharfe (fuzzy) Sicherungspunkte

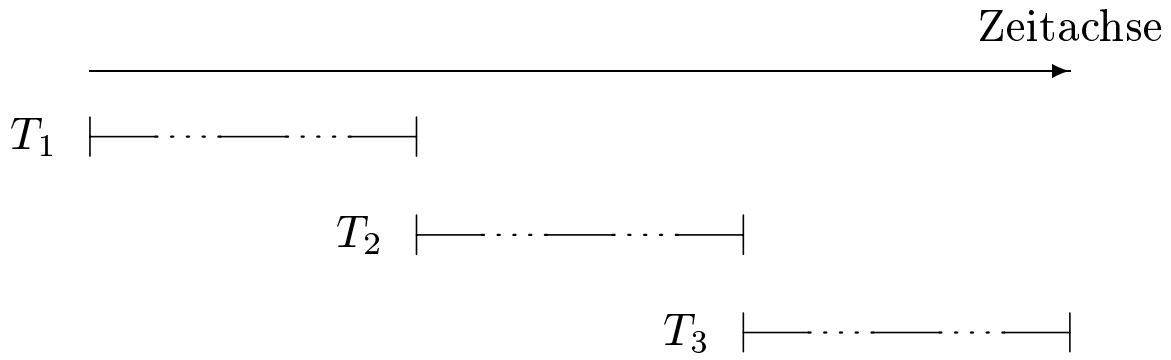
- modifizierte Seiten werden nicht ausgeschrieben
- nur deren Kennung wird ausgeschrieben
  - *Dirty Pages*=Menge der modifizierten Seiten
- *MinDirtyPageLSN*: die minimale LSN, deren Änderungen noch nicht ausgeschrieben wurde
- *MinLSN*: die kleinste LSN der zum Sicherungszeitpunkt aktiven TAs

Recovery nach einem Verlust der materialisierten Datenbasis

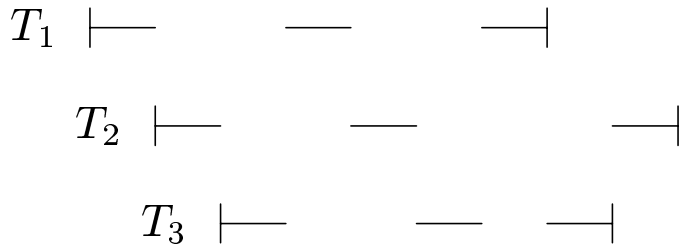


Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einbenutzerbetrieb und



(b) im (verzahnten) Mehrbenutzerbetrieb (gestrichelte Linien repräsentieren Wartezeiten)



Verlorengegangene Änderungen (*lost update*)

Schritt	T_1	T_2
1.	$\text{read}(A, a_1)$	
2.	$a_1 := a_1 - 300$	
3.		$\text{read}(A, a_2)$
4.		$a_2 := a_2 * 1.03$
5.		$\text{write}(A, a_2)$
6.	$\text{write}(A, a_1)$	
7.	$\text{read}(B, b_1)$	
8.	$b_1 := b_1 + 300$	
9.	$\text{write}(B, b_1)$	

Abhängigkeit von nicht freigegebenen Änderungen

Schritt	T_1	T_2
1.	$\text{read}(A, a_1)$	
2.	$a_1 := a_1 - 300$	
3.	$\text{write}(A, a_1)$	
4.		$\text{read}(A, a_2)$
5.		$a_2 := a_2 * 1.03$
6.		$\text{write}(A, a_2)$
7.	$\text{read}(B, b_1)$	
8.	$\cdots$	
9.	**abort**	

Fehler . . . (Forts.)

Phantomproblem

T_1	T_2
	select sum(KontoStand)
	from Konten
insert into Konten	
values $(C, 1000, \ldots)$	
	select sum(KontoStand)
	from Konten

Serialisierbarkeit

- Historie ist „äqivalent“ zu einer seriellen Historie
- dennoch parallele (verzahnte) Ausführung möglich

Serialisierbare Historie von T_1 und T_2

Schritt	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.		**BOT**
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	**commit**	
10.		read(A)
11.		write(A)
12.		**commit**

Serielle Ausführung von T_1 vor T_2, also $T_1 \mid T_2$

Schritt	T_1	T_2
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	**commit**	
7.		**BOT**
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		**commit**

Nicht serialisierbare Historie

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A)	
3.	write(A)	
4.		**BOT**
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		**commit**
10.	read(B)	
11.	write(B)	
12.	**commit**	

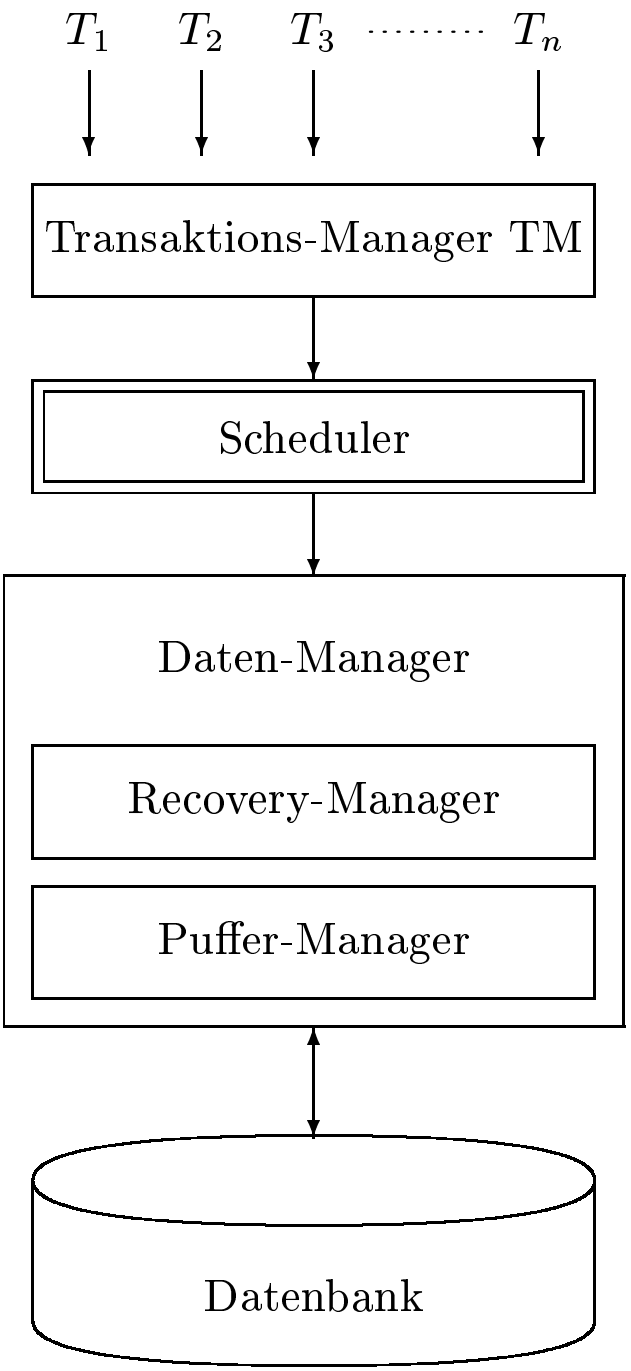
Zwei verzahnte Überweisungs-Transaktionen

Schritt	T_1	T_3
1.	**BOT**	
2.	$\text{read}(A, a_1)$	
3.	$a_1 := a_1 - 50$	
4.	$\text{write}(A, a_1)$	
5.		**BOT**
6.		$\text{read}(A, a_2)$
7.		$a_2 := a_2 - 100$
8.		$\text{write}(A, a_2)$
9.		$\text{read}(B, b_2)$
10.		$b_2 := b_2 + 100$
11.		$\text{write}(B, b_2)$
12.		**commit**
13.	$\text{read}(B, b_1)$	
14.	$b_1 := b_1 + 50$	
15.	$\text{write}(B, b_1)$	
16.	**commit**	

Eine Überweisung (T_1) und eine Zinsgutschrift (T_3)

Schritt	T_1	T_3
1.	**BOT**	
2.	$\text{read}(A, a_1)$	
3.	$a_1 := a_1 - 50$	
4.	$\text{write}(A, a_1)$	
5.		**BOT**
6.		$\text{read}(A, a_2)$
7.		$a_2 := a_2 * 1.03$
8.		$\text{write}(A, a_2)$
9.		$\text{read}(B, b_2)$
10.		$b_2 := b_2 * 1.03$
11.		$\text{write}(B, b_2)$
12.		**commit**
13.	$\text{read}(B, b_1)$	
14.	$b_1 := b_1 + 50$	
15.	$\text{write}(B, b_1)$	
16.	**commit**	

Der Datenbank-Scheduler



Zwei Sperrmodi

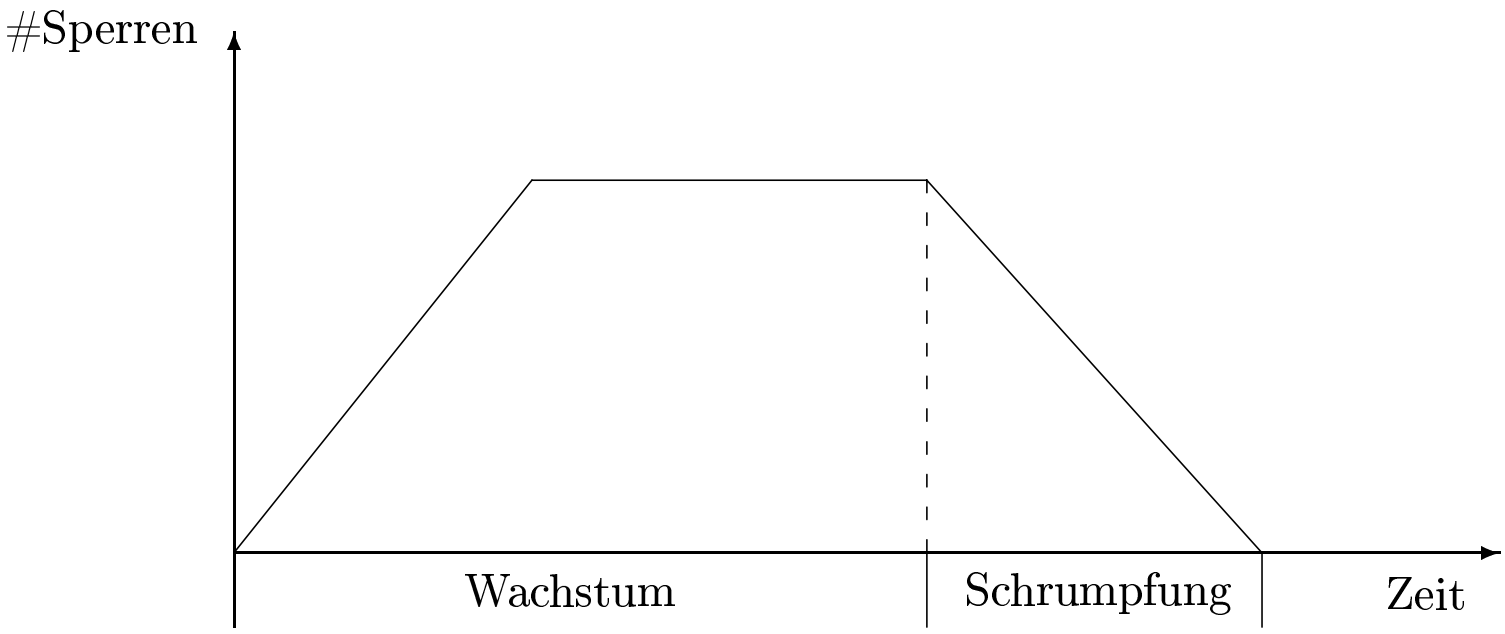
- S (shared, read lock, Lesesperre):
- X (exclusive, write lock, Schreibsperre):
- *Verträglichkeitsmatrix* (auch *Kompatibilitätsmatrix* genannt)

	NL	S	X
S	$\surd$	$\surd$	–
X	$\surd$	–	–

Zwei-Phasen-Sperrprotokoll: Definition

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muß vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. eine Transaktion muß die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann.
4. Jede Transaktion durchläuft zwei Phasen:
   - Eine *Wachstumsphase*, in der sie Sperren anfordern, aber keine freigeben darf und
   - Eine *Schrumpfungsphase*, in der sie ihre bisher erworbenen Sperren freigibt, aber keine weiteren anfordern darf.
5. Bei EOT (Transaktionende) muß eine Transaktion alle ihre Sperren zurückgeben.

Zwei-Phasen Sperrprotokoll: Graphik



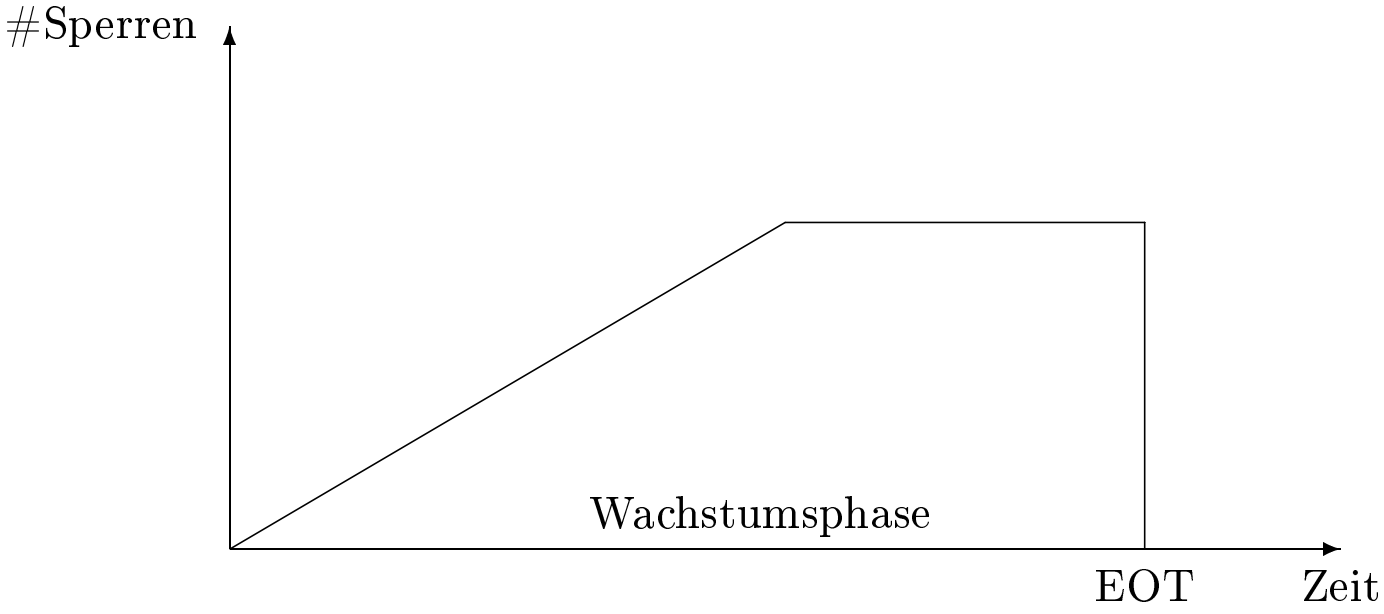
Verzahnung zweier TAs gemäß 2PL

- T_1 modifiziert nacheinander die Datenobjekte A und B (z.B. eine Überweisung)
- T_2 liest nacheinander dieselben Datenobjekte A und B (z.B. zur Aufsummierung der beiden Kontostände).

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 muß warten
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 wecken
10.		read(A)	
11.		**lockS**(B)	T_2 muß warten
12.	write(B)		
13.	**unlockX**(B)		T_2 wecken
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

Strenges Zwei-Phasen Sperrprotokoll

- 2PL schließt kaskadierendes Rücksetzen nicht aus
- Erweiterung zum *strengen* 2PL:
  - alle Sperren werden bis EOT gehalten
  - damit ist kaskadierendes Rücksetzen ausgeschlossen



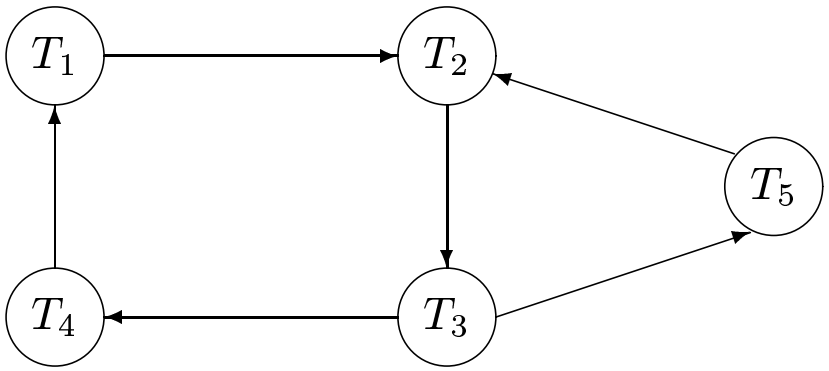
Ein verklemmter Schedule

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.		**BOT**	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 muß warten auf T_2
9.		**lockS**(A)	T_2 muß warten auf T_1
10.	...	...	$\Rightarrow$*Deadlock*

Erkennung von Verklemmungen

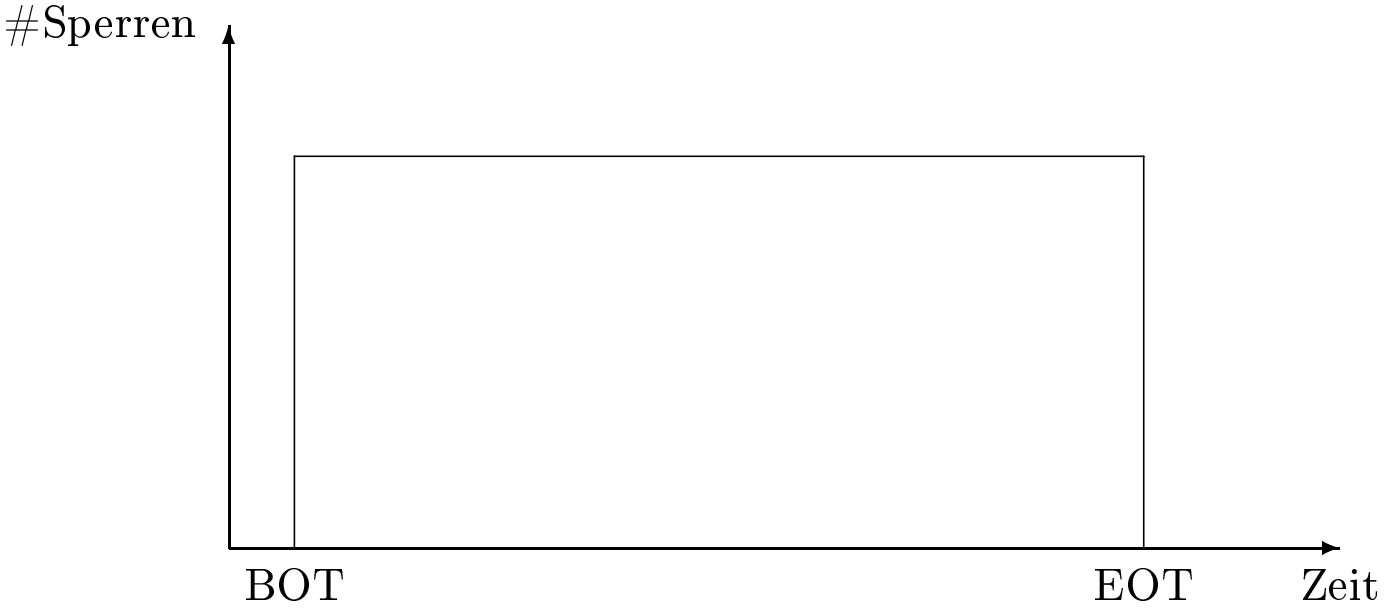
Wartegraph mit zwei Zyklen:

- $T_1 \to T_2 \to T_3 \to T_4 \to T_1$
- $T_2 \to T_3 \to T_5 \to T_2$



- beide Zyklen können durch Rücksetzen von T_3 „gelöst“ werden
- Zyklenerkennung durch Tiefensuche im Wartegraphen

Preclaiming in Verbindung mit dem strengen 2 PL-Protokoll



Verklemmungsvermeidung durch Zeitstempel

- Jeder Transaktion wird ein eindeutiger Zeitstempel (TS) zugeordnet
- ältere TAs haben einen kleineren Zeitstempel als jüngere TAs
- TAs dürfen nicht mehr „bedingungslos“ auf eine Sperre warten

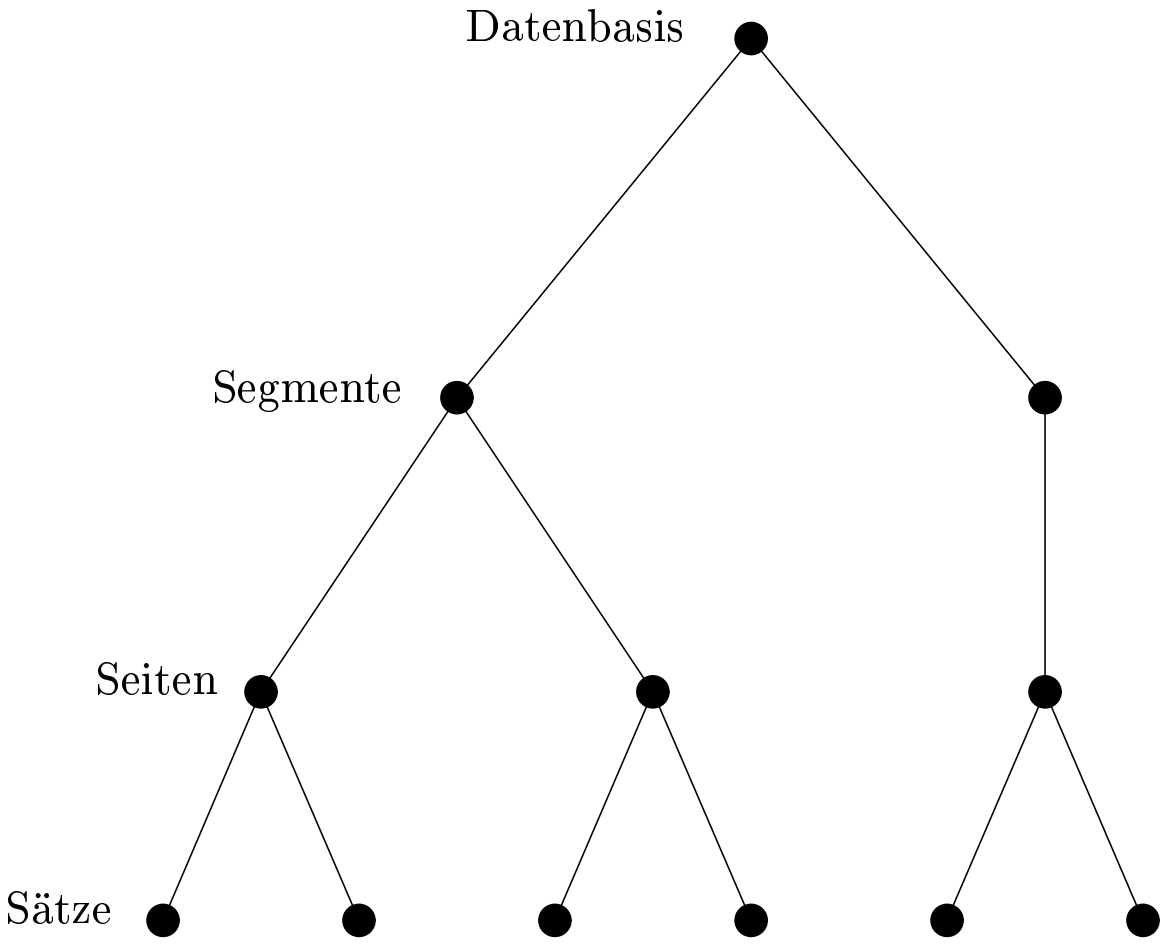
wound-wait Strategie

- T_1 will Sperre erwerben, die von T_2 gehalten wird
- Wenn T_1 älter als T_2 ist, wird T_2 abgebrochen und zurückgesetzt, so daß T_1 weiterlaufen kann.
- Sonst wartet T_1 auf die Freigabe der Sperre durch T_2.

wait-die Strategie

- T_1 will Sperre erwerben, die von T_2 gehalten wird
- Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre.
- Sonst wird T_1 abgebrochen und zurückgesetzt.

Hierarchische Anordnung möglicher Sperrgranulate



Erweiterte Sperrmodi

- *NL*: keine Sperrung (no lock),
- S: Sperrung durch Leser,
- X: Sperrung durch Schreiber,
- *IS* (intention share): Weiter unten in der Hierarchie ist eine Lesesperre (S) beabsichtigt,
- *IX* (intention exclusive): Weiter unten in der Hierarchie ist eine Schreibsperre (X) beabsichtigt.

Kompatibilitätsmatrix

	NL	S	X	IS	IX
S	$\surd$	$\surd$	–	$\surd$	–
X	$\surd$	–	–	–	–
IS	$\surd$	$\surd$	–	$\surd$	$\surd$
IX	$\surd$	–	–	$\surd$	$\surd$

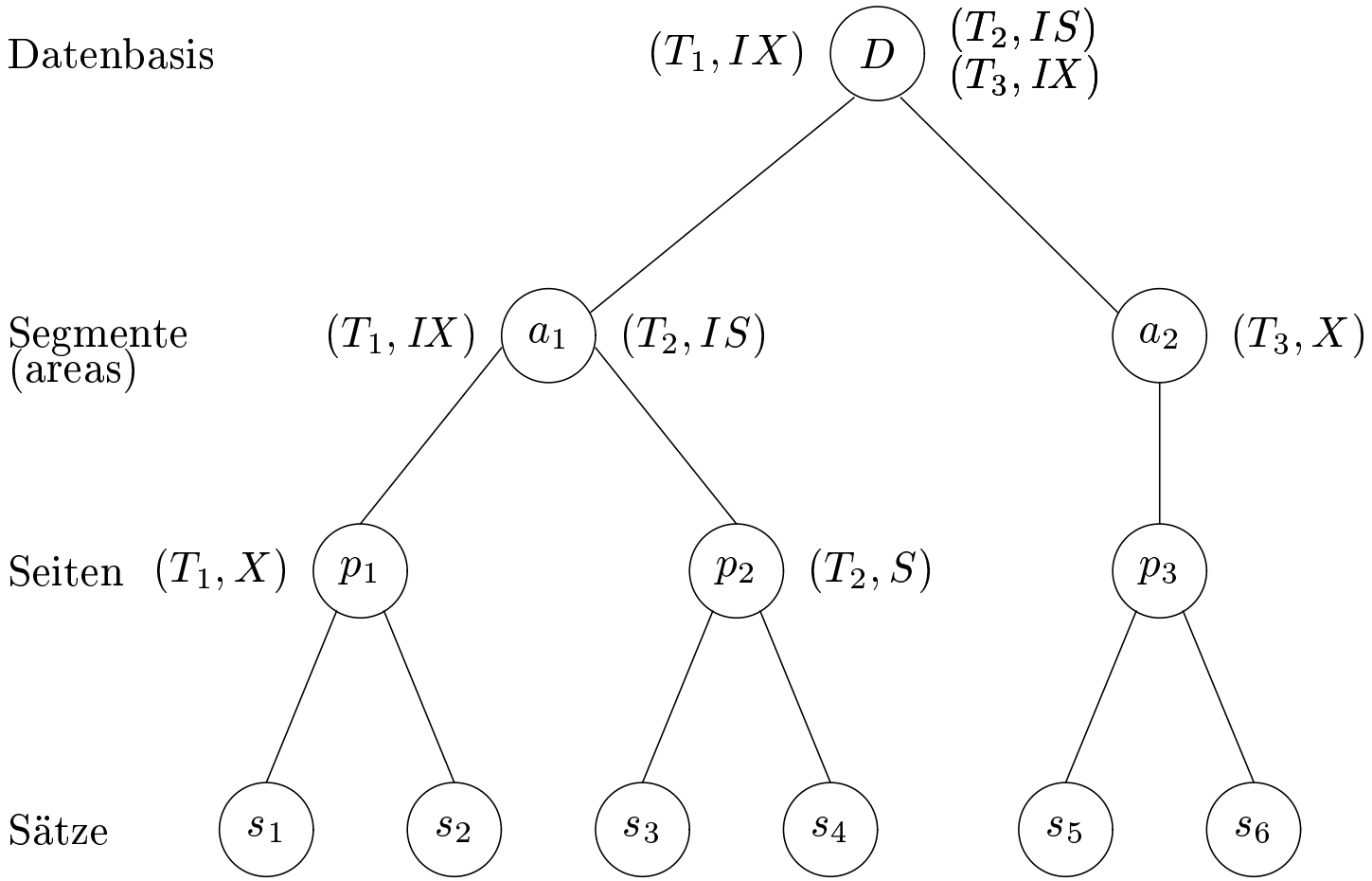
Sperrprotokoll des MGL

1. Bevor ein Knoten mit S oder IS gesperrt wird, müssen alle Vorgänger in der Hierarchie vom Sperrer (also der Transaktion, die die Sperre anfordert) im IX- oder IS- Modus gehalten werden.

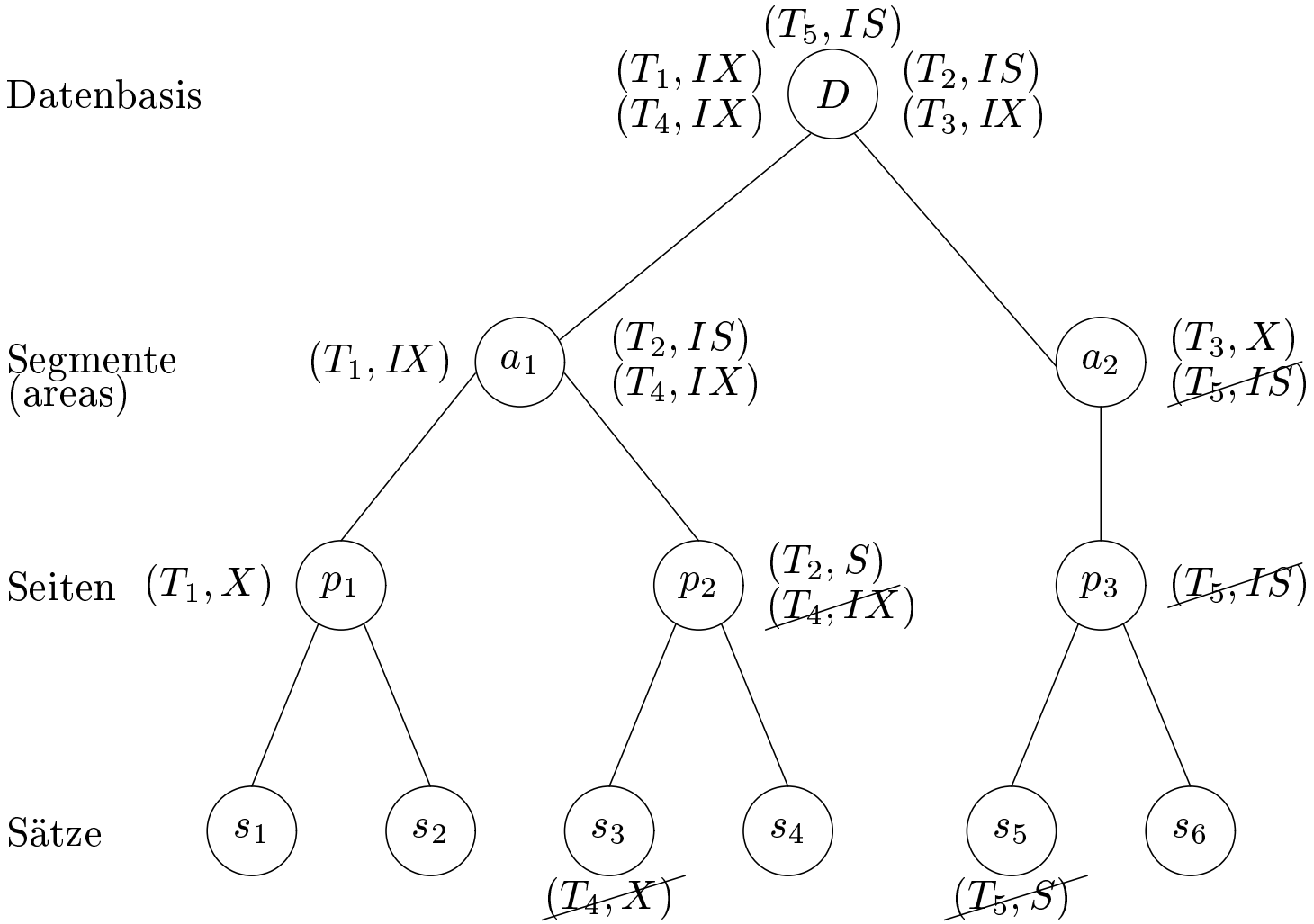
2. Bevor ein Knoten mit X oder IX gesperrt wird, müssen alle Vorgänger vom Sperrer im IX-Modus gehalten werden.

3. Die Sperren werden von unten nach oben (bottom up) freigegeben, so daß bei keinem Knoten die Sperre freigegeben wird, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat.

Datenbasis-Hierarchie mit Sperren



Datenbasis-Hierarchie mit blockierten Transaktionen



- die TAs T_4 und T_5 sind blockiert (warten auf Freigabe von Sperren)
- es gibt aber in diesem Beispiel (noch) keine Verklemmung
- Verklemmungen sind aber auch bei MGL möglich

Einfüge- und Löschoperationen, Phantome

- Vor dem Löschen eines Objekts muß die Transaktion eine X-Sperre für dieses Objekt erwerben. Man beachte aber, daß eine andere TA, die für dieses Objekt ebenfalls eine Sperre erwerben will, diese nicht mehr erhalten kann, falls die Löschtransaktion erfolgreich (mit **commit**) abschließt.
- Beim Einfügen eines neuen Objekts erwirbt die einfügende Transaktion eine X-Sperre.

Phantomprobleme

T_1	T_2
select count$(*)$	
from prüfen	
where Note **between** 1 **and** 2;	
	insert into prüfen
	values(29555, 5001, 2137, 1);
select count$(*)$	
from prüfen	
where Note **between** 1 **and** 2;	

- Das Problem läßt sich dadurch lösen, daß man zusätzlich zu den Tupeln auch den Zugriffsweg, auf dem man zu den Objekten gelangt ist, sperrt
- Wenn also ein Index für das Attribut *Note* existiert, würde der Indexbereich $[1, 2]$ für T_1 mit einer S-Sperre belegt
- Wenn jetzt also Transaktion T_2 versucht, das Tupel

  [29555, 5001, 2137, 1] in *prüfen* einzufügen, wird die TA blockiert

Jedem Datum A in der Datenbasis werden bei diesem Synchronisationsverfahren zwei Marken zugeordnet:

1. $readTS(A)$:
2. $writeTS(A)$:

Synchronisationsverfahren

- T_i will A lesen, also $r_i(A)$
  - Falls $TS(T_i) < writeTS(A)$ gilt, haben wir ein Problem:
    - Die Transaktion T_i ist älter als eine andere Transaktion, die A schon geschrieben hat.
    - Also muß T_i zurückgesetzt werden.
  - Anderenfalls, wenn also $TS(T_i) \geq writeTS(A)$ gilt, kann T_i ihre Leseoperation durchführen und die Marke $readTS(A)$ wird auf $max(TS(T_i), readTS(A))$ gesetzt.

- T_i will A schreiben, also $w_i(A)$
  - Falls $TS(T_i) < readTS(A)$ gilt, gab es eine jüngere Lesetransaktion, die den neuen Wert von A, den T_i gerade beabsichtigt zu schreiben, hätte lesen müssen. Also muß T_i zurückgesetzt werden.
  - Falls $TS(T_i) < writeTS(A)$ gilt, gab es eine jüngere Schreibtransaktion. D.h. T_i beabsichtigt einen Wert einer jüngeren Transaktion zu überschreiben. Das muß natürlich verhindert werden, so daß T_i auch in diesem Fall zurückgesetzt werden muß.
  - Anderenfalls darf T_i das Datum A schreiben und die Marke $writeTS(A)$ wird auf $TS(T_i)$ gesetzt.

1. *Lesephase*:
   - In dieser Phase werden alle Operationen der Transaktion ausgeführt – also auch die Änderungsoperationen.
   - Gegenüber der Datenbasis tritt die Transaktion in dieser Phase aber nur als Leser in Erscheinung, da alle gelesenen Daten in lokalen Variablen der Transaktion gespeichert werden.
   - alle Schreiboperationen werden (zunächst) auf diesen lokalen Variablen ausgeführt.
2. *Validierungsphase*:
   - In dieser Phase wird entschieden, ob die Transaktion möglicherweise in Konflikt mit anderen Transaktionen geraten ist.
   - Dies wird anhand von Zeitstempeln entschieden, die den Transaktionen in der Reihenfolge zugewiesen werden, in der sie in die Validierungsphase eintreten.
3. *Schreibphase*:
   - Die Änderungen der Transaktionen, bei denen die Validierung positiv verlaufen ist, werden in dieser Phase in die Datenbank eingebracht.

Validierung bei der optimistischen Synchronisation

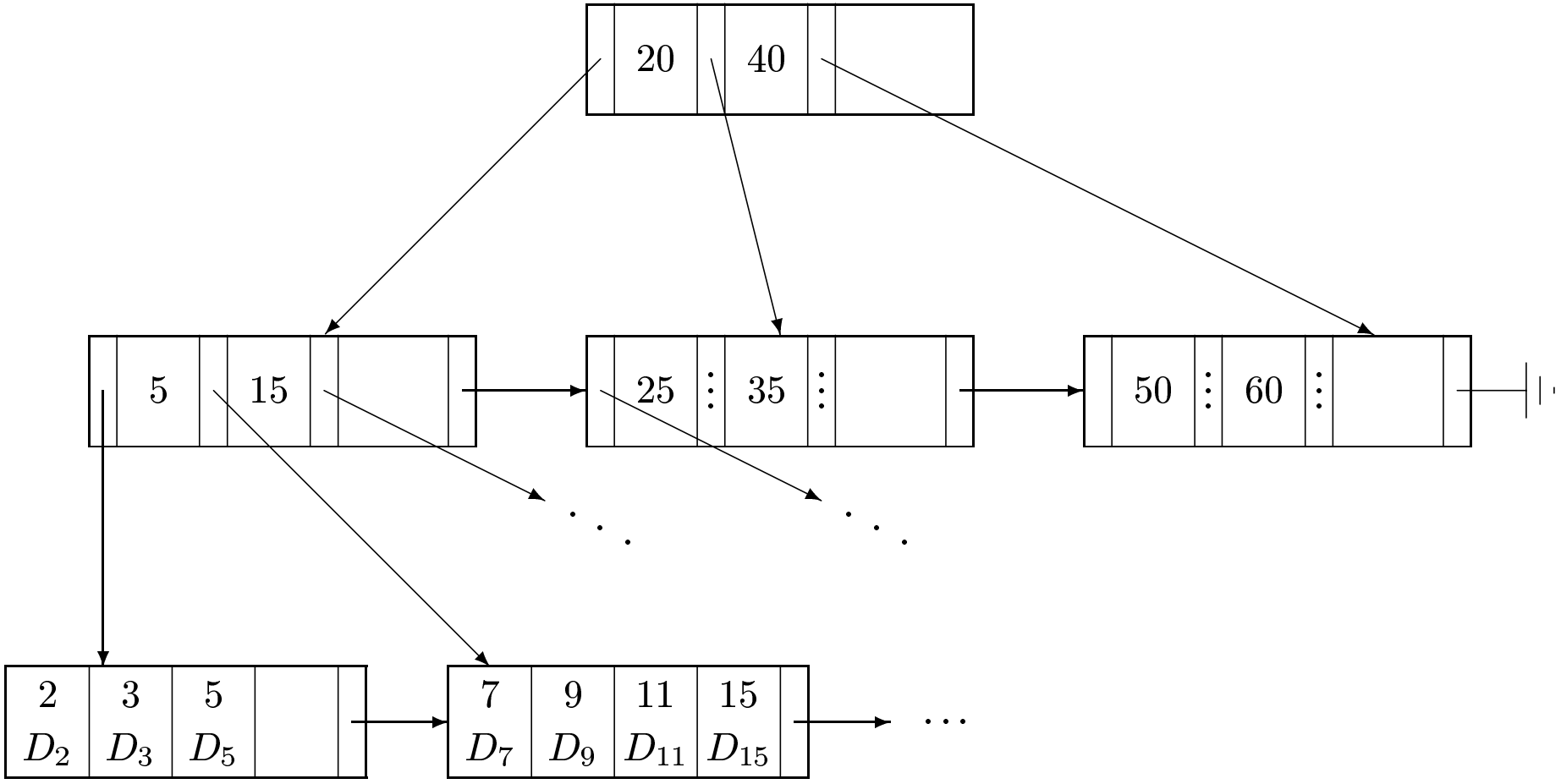
Vereinfachende Annahme: Es ist immer nur eine TA in der Validierungsphase!

Wir wollen eine Transaktion T_j validieren. Die Validierung ist erfolgreich falls für **alle** älteren Transaktionen T_a – also solche die früher ihre Validierung abgeschlossen haben – eine der beiden folgenden Bedingungen gelten:

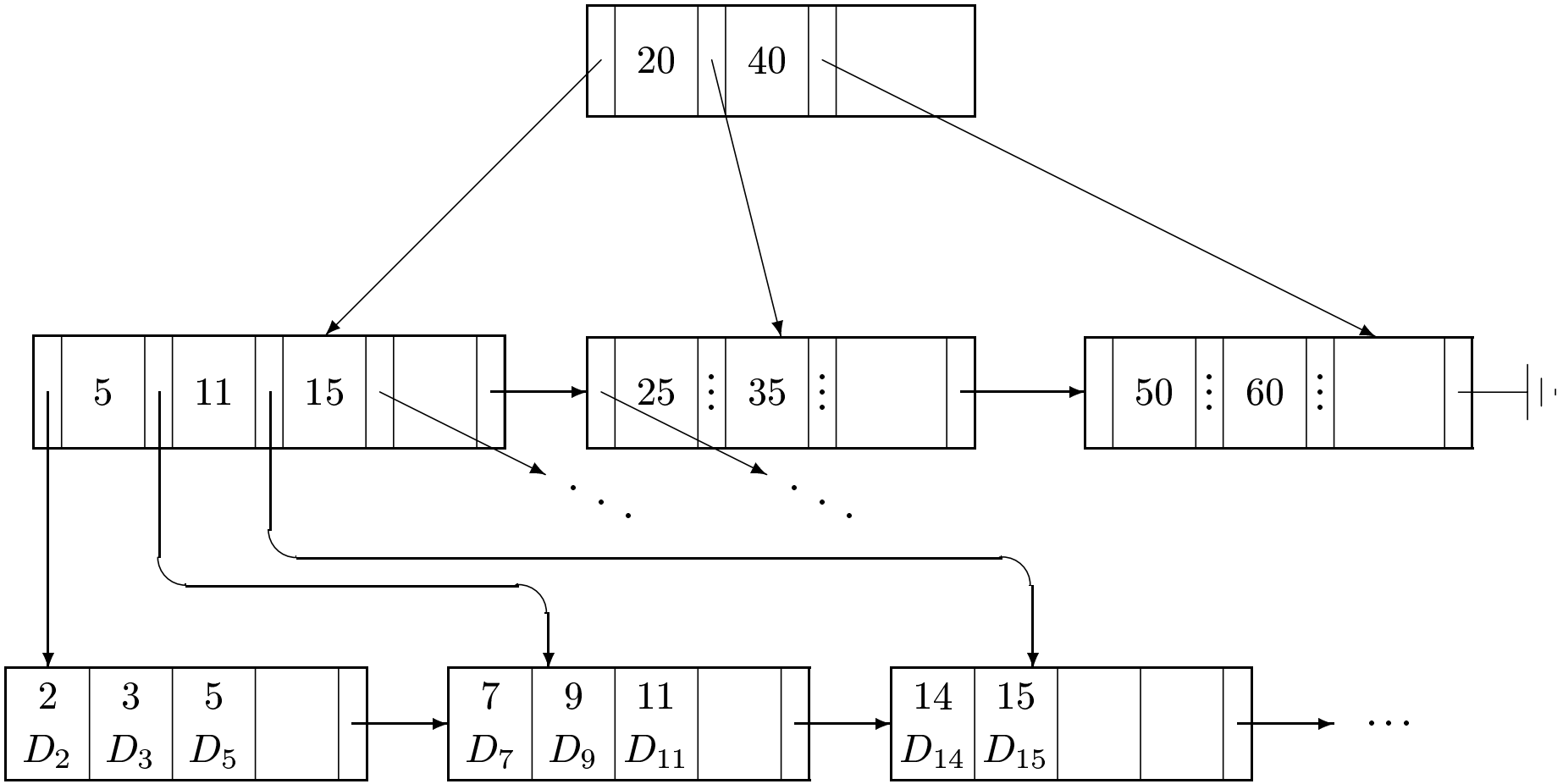
1. T_a war zum Beginn der Transaktion T_j schon abgeschlossen – einschließlich der Schreibphase.
2. Die Menge der von T_a geschriebenen Datenelemente, genannt $\mathit{WriteSet}(T_a)$, enthält keine Elemente der Menge der gelesenen Datenelemente von T_j, genannt $\mathit{ReadSet}(T_j)$. Es muß also gelten:

$$\mathit{WriteSet}(T_a) \cap \mathit{ReadSet}(T_j) = \emptyset$$

B^+-Baum mit *rechts*-Verweisen zur Synchronisation



B$^+$-Baum mit $rechts$-Verweisen nach Einfügen von 14



Transaktionsverwaltung in SQL92

set transaction
[read only, | read write,]
[**isolation level**
read uncommitted, |
read committed, |
repeatable read, |
serializable,]
[**diagnostics size** ...,]

- **read uncommitted**: Dies ist die schwächste Konsistenzstufe. Sie darf auch nur für **read only**-Transaktionen spezifiziert werden. Eine derartige Transaktion hat Zugriff auf noch nicht festgeschriebene Daten. Zum Beispiel ist folgender Schedule möglich:

T_1	T_2
	read(A)
	...
	write(A)
read(A)	
...	
	rollback

- **read committed**: Diese Transaktionen lesen nur festgeschriebene Werte. Allerdings können sie unterschiedliche Zustände der Datenbasis-Objekte zu sehen bekommen:

T_1	T_2
read(A)	
	write(A)
	write(B)
	commit
read(B)	
read(A)	
...	

- **repeatable read**: Das oben aufgeführte Problem des *non repeatable read* wird durch diese Konsistenzstufe ausgeschlossen. Allerdings kann es hierbei noch zum Phantomproblem kommen. Dies kann z.B. dann passieren, wenn eine parallele Änderungstransaktion dazu führt, daß Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.

- **serializable**: Diese Konsistenzstufe fordert die Serialisierbarkeit. Dies ist der Default.