

---

# Datenbanken

Prof. Dr. Ralf Möller

**Universität zu Lübeck**

**Institut für Informationssysteme**

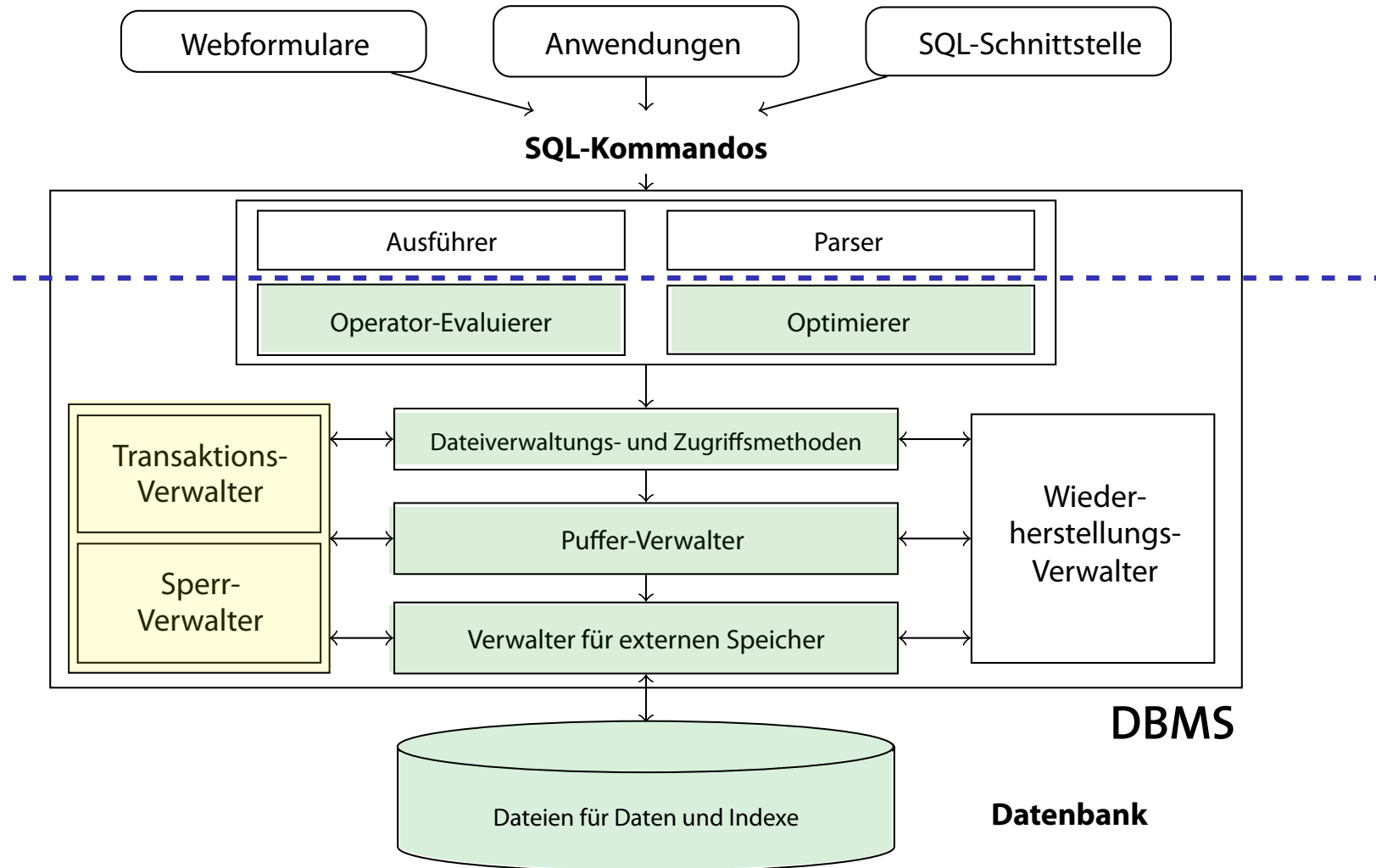
Marc Stelzner (Übungen)

Torben Matthias Kempfert (Tutor)

Maurice-Raphael Sambale (Tutor)



# Transaktionsverwaltung



# Danksagung

---

- Diese Vorlesung ist inspiriert von den Präsentationen zu dem Kurs:

„Architecture and Implementation of Database Systems“  
von Jens Teubner an der ETH Zürich

- Graphiken und Code-Bestandteile wurden mit Zustimmung des Autors (und ggf. kleinen Änderungen) aus diesem Kurs übernommen

# Eine einfache Transaktion

---

- Ab und zu verwende ich meine Kreditkarte, um Geld von meinem Konto abzuheben
- Der Bankautomat führt folgende Transaktion auf der Datenbasis der Bank aus

```
1 bal ← read_bal (acct_no) ;  
2 bal ← bal – 100 CHF ;  
3 write_bal (acct_no, bal) ;
```



- Wenn alles fehlerfrei abläuft, wird mein Konto richtig verwaltet



# Nebenläufiger Zugriff

Mein Frau verwendet eine Karte für das gleiche Konto...

- Eventuell verwenden wir unsere Karten zur gleichen Zeit

me	my wife	DB state
$bal \leftarrow read(acct);$		1200
	$bal \leftarrow read(acct);$	1200
$bal \leftarrow bal - 100;$		1200
	$bal \leftarrow bal - 200;$	1200
$write(acct, bal);$		1100
	$write(acct, bal);$	1000

- Die erste Aktualisierung des Kontos ist verlorenggegangen: Mich freut's! Allerdings...

## ... kann es auch nach hinten losgehen

---

- Diesmal wird Geld von einem Konto auf ein anderes transferiert

```
// Subtract money from source (checking) account
1 chk_bal ← read_bal (chk_acct_no) ;
2 chk_bal ← chk_bal - 500 CHF ;
3 write_bal (chk_acct_no, chk_bal) ;

// Credit money to the target (saving) account
4 sav_bal ← read_bal (sav_acct_no) ;
5 sav_bal ← sav_bal + 500 CHF ;
6 write_bal (sav_acct_no, sav_bal) ;
```

- Bevor die Transaktion zum Schritt 6 kommt, wird die Ausführung abgebrochen (Stromversorgungsproblem, Plattenproblem, Softwarefehler, ...).

Mein Geld ist verschwunden ☹️



# ACID-Eigenschaften und Transaktionen

---

Um diese und viele andere Effekte zu vermeiden, stellen DMBS folgende Eigenschaften sicher

- **Atomicity:** Entweder werden alle oder keine Werteänderungen einer Transaktion in den Datenbankzustand übernommen
- **Consistency:** Eine Transaktion überführt einen konsistenten Zustand (FDs, Integritätsbedingungen) in einen anderen
- **Isolation:** Eine Transaktion berücksichtigt bei der Berechnung keine Effekte andere parallel laufender Transaktionen
- **Durability:** Effekte einer erfolgreichen Transaktion werden persistent gemacht

# Anomalien: Lost Update

---

- Wir haben schon „Lost Update“ im Beispiel betrachtet
- Effekte einer Transaktion gehen verloren, weil eine andere Transaktion geänderte Werte unkontrolliert überschreibt

# Anomalien: Inconsistent Read

Betrachten wir die Überweisung in SQL

```
Transaction 1
UPDATE Accounts
SET balance = balance - 500
WHERE customer = 4711
  AND account_type = 'C';

Transaction 2
SELECT SUM(balance)
FROM Accounts
WHERE customer = 4711;

UPDATE Accounts
SET balance = balance + 500
WHERE customer = 4711
  AND account_type = 'S';
```

➤ Transaktion 2 sieht einen inkonsistenten Zustand

# Anomalien: Dirty Read

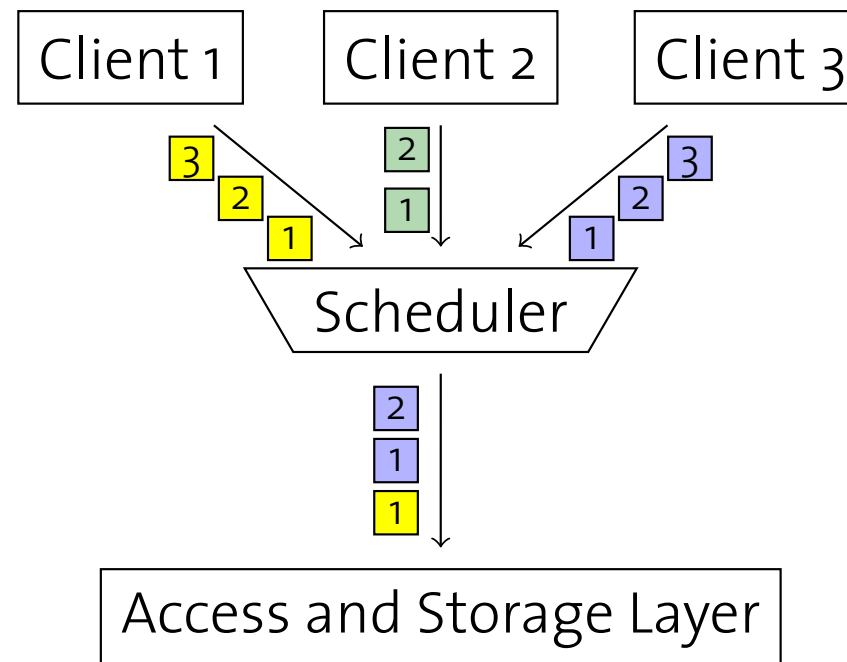
An einem anderen Tag heben meine Frau und ich zur gleichen Zeit Geld vom Automaten ab

me	my wife	DB state
<i>bal</i> ← read ( <i>acct</i> );		1200
<i>bal</i> ← <i>bal</i> - 100;		1200
write ( <i>acct</i> , <i>bal</i> );		1100
	<i>bal</i> ← read ( <i>acct</i> );	1100
	<i>bal</i> ← <i>bal</i> - 200;	1100
abort;		1200
	write ( <i>acct</i> , <i>bal</i> );	900

- Die Transaktion meiner Frau hat schon einen geänderten Zustand gelesen bevor meine Transaktion zurückgerollt wird

# Nebenläufige Ausführung

- Ein Steuerprogramm (Scheduler) entscheidet über die Ausführungsreihenfolge der nebenläufigen Datenbankzugriffe



# Datenbankobjekte und Zugriffe daraus

---

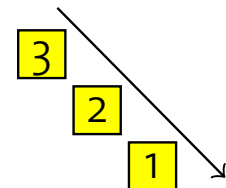
- Wir nehmen ein vereinfachtes Datenmodell an
  - Eine Datenbank besteht aus einer Menge von benannten Objekten. In jedem Zustand hat ein Objekt einen Wert.
  - Transaktionen greifen auf ein Objekt  $o$  mit den Operationen **read** und **write** zu
- In einer relationalen DB haben wir:  
Objekt  $\hat{=}$  Komponente eines Tupels



# Transaktion: Definition

---

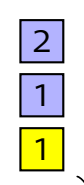
- Eine **Datenbanktransaktion** ist eine (strikt geordnete) **Folge von Schritten**, wobei ein Schritt eine Zugriffsoperation auf ein Objekt ist
  - **Transaktion**  $T = \langle s_1, \dots, s_n \rangle$
  - **Schritt**  $s_i = (a_i, e_i)$
  - **Zugriffsoperation**  $a_i \in \{ r(\text{ead}), w(\text{rite}) \}$
- Die Länge einer Transaktion ist definiert als die Anzahl der Schritte  $|T| = n$
- **Beispiel:**  $T = \langle (\text{read}, \text{Checking}), (\text{write}, \text{Checking}), (\text{read}, \text{Saving}), (\text{write}, \text{Saving}) \rangle$
- **Kurzform:**  $T = \langle r(C), w(C), r(S), w(S) \rangle$



# Ausführungspläne

Ein (Ausführungs-)Plan (Schedule)  $S$  für eine gegebene Menge von Transaktionen  $\mathbf{T} = \{T_1, \dots, T_n\}$  ist eine Folge von Ausführungsschritten

$$S(k) = (T_j, a_i, e_i) \quad k = 1 \dots m$$



so dass

1.  $S$  genau die Schritte aller Transaktionen enthält und die
2. Ordnung der Schritte jeder Transaktion beibehalten wird  
( $a_p, e_p$ ) < ( $a_q, e_q$ ) in  $T_j$ , dann ( $T_j, e_p, e_p$ ) < ( $T_j, a_q, e_q$ ) in  $S$

Wir schreiben  $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$  für

$$S(1) = (T_1, \text{read}, B) \quad S(3) = (T_1, \text{write}, B)$$

$$S(2) = (T_2, \text{read}, B) \quad S(4) = (T_2, \text{write}, B)$$

# Serielle Ausführung

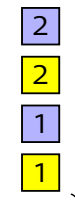
---

Ein spezieller Plan ist die serielle Ausführung

- Ein Plan heißt **seriell** genau dann, wenn für jede Transaktion  $T_j$  alle ihre **Schritte direkt aufeinanderfolgen** (ohne Schritte anderer Transaktion dazwischen)

Betrachten wir das Geldautomatenbeispiel:

- $S = \langle r_1(B), r_2(B), w_1(B), w_2(B) \rangle$
- Dieser Plan ist nicht seriell



Wenn meine Frau später zum Automaten geht, ergibt sich

- $S = \langle r_1(B), w_1(B), r_2(B), w_2(B) \rangle$
- Dieser Plan ist seriell

# Korrektheit der seriellen Ausführung

---

- Anomalien können nur auftreten, wenn die Schritte mehrerer Transaktionen verschränkt ausgeführt werden (Multi-User-Modus)
- Falls alle Transaktionen bis zum Ende ausgeführt werden (keine Nebenläufigkeit), treten keine Anomalien auf
- **Jede serielle Ausführung ist korrekt**
- Verzicht auf nebenläufige Ausführung nicht praktikabel, da zu langsam (Wartezeit auf Platten)
- Jede verschränkte Ausführung, die einen gleichen Zustand wie eine serielle erzeugt, ist korrekt

# Konflikte

---

- Was bedeutet es, dass ein Plan  $S$  äquivalent zu einem anderen  $S'$  ist?
- Manchmal kann man einfach Teilschritte aus verschiedenen Transaktionen in einem Plan umordnen
  - Nicht jedoch die Teilschritte innerhalb einer einzelnen Transaktion (sonst eventuell anderes Ergebnis)
- Zwei Operationen  $(a, e)$  und  $(a', e')$  stehen in Konflikt zueinander  $(a, e) \bowtie (a', e')$ , wenn ihre Ausführungsreihenfolge bedeutsam ist
  - Umordnung in einem Plan dann nicht möglich
- Jeder Plan  $S'$ , der durch legale Umordnung von  $S$  generiert werden kann, heißt konfliktäquivalent zu  $S$

# Konflikte

Ausführbare Definition eines Konflikts:

- Zwei Operationen  $(T_i, a, e)$  und  $(T_j, a', e')$  sind in Konflikt zueinander in  $S$ , wenn
  - sie zu zwei verschiedenen Transaktionen gehören ( $T_i \neq T_j$ )
  - sie das gleiche Objekte referenzieren ( $e = e'$ ) und
  - mindestens eine der Operationen  $a$  oder  $a'$  eine Schreiboperation ist
- Hierdurch ist eine sog. Konfliktmatrix definiert

	read	write
read		×
write	×	×

- Konfliktrelation  $\prec_S$ :

$$(T_i, a, e) \prec_S (T_j, a', e')$$

:=

$$(a, e) \leftrightarrow (a', e') \wedge (T_i, a, e) \text{ kommt vor } (T_j, a', e') \text{ in } S \wedge T_i \neq T_j$$



# Konflikt-Serialisierbarkeit

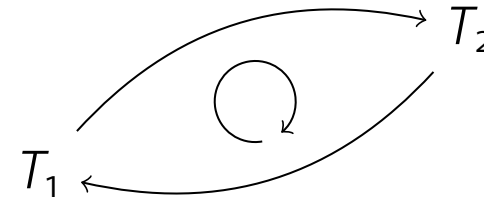
---

- Ein Plan  $S$  heißt **Konflikt-serialisierbar**, gdw. er Konflikt-äquivalent ist zu **einem** seriellen Plan  $S'$
- Die Ausführung eines Konflikt-serialisierbaren Plans  $S$  ist **korrekt** ( $S$  braucht nicht seriell zu sein)
- Korrektheit eines Plans kann anhand des **Konfliktgraphen**  $G_S$  gezeigt werden (auch **Serialisierungsgraph** genannt)
  - Knoten von  $G_S$  sind die Transaktionen  $T_i$  aus  $S$
  - Kanten  $T_i \rightarrow T_j$  werden hinzugefügt, gdw.  $S$  Operationen  $(T_i, a, e)$  und  $(T_j, a', e')$  enthält, so dass  $(T_i, a, e) <_S (T_j, a', e')$
- $S$  ist Konflikt-serialisierbar, wenn  $G_S$  **zyklenfrei** ist
  - Serielle Ausführung bestimmbar durch topologische Sortierung

# Serialisierungsgraph

## Beispiel: ATM-Transaktion

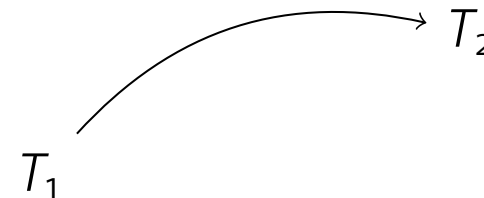
- ▶  $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$
- ▶ Konfliktrelation:
  - $(T_1, r, A) \prec_S (T_2, w, A)$
  - $(T_2, r, A) \prec_S (T_1, w, A)$
  - $(T_1, w, A) \prec_S (T_2, w, A)$



nicht serialisierbar

## Beispiel: Zwei Geldtransfers

- ▶  $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$
- ▶ Konfliktrelation:
  - $(T_1, r, C) \prec_S (T_2, w, C)$
  - $(T_1, w, C) \prec_S (T_2, r, C)$
  - $(T_1, w, C) \prec_S (T_2, w, C)$
  - ⋮



serialisierbar



# Sperren im Anfrageplan

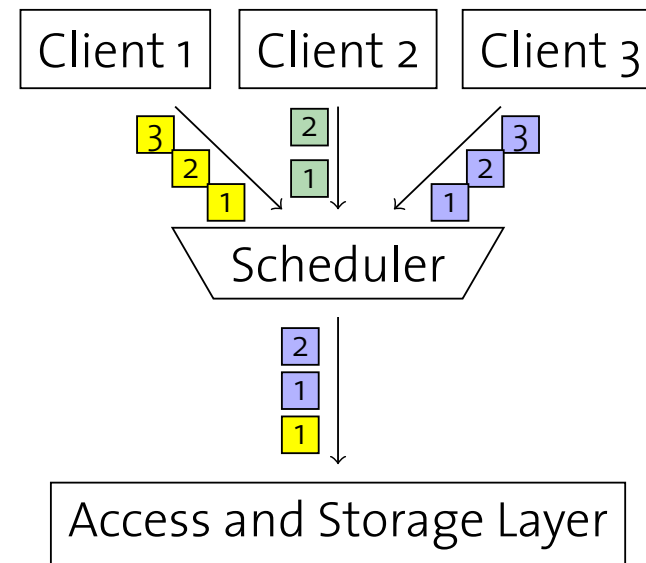
Können wir einen Scheduler bauen, der immer einen serialisierbaren Plan generiert?

Idee:

- Lasse jede Transaktion eine Sperre akquirieren, bevor auf ein Datum zugegriffen wird

```
1 lock o ;  
2 ...access o ...;  
3 unlock o ;
```

- Dadurch wird ein nebenläufiger Zugriff auf o verhindert



# Sperr-Verwaltung

---

- Falls eine Sperre nicht zugeteilt wird (z.B. weil eine andere Transaktion  $T'$  die Sperre schon hält), wird die anfragende Transaktion  $T$  **blockiert**
- Der Verwalter **setzt** die Ausführung von Aktionen einer blockierten Transaktion  $T$  **aus**
- Sobald  $T'$  die Sperre **freigibt**, kann sie an  $T$  vergeben werden (oder an eine andere Transaktion, die darauf wartet)
- Eine Transaktion, die eine Sperre erhält, wird **fortgesetzt**
- Sperren regeln die **relative Ordnung der Einzeloperationen** verschiedener Transaktionen

# Aufgabe:

- Reicht die Idee der Sperrverwaltung aus, um Serialisierbarkeit zu garantieren?

## Transaction 1

```
lock (acct) ;  
read (acct) ;  
unlock (acct) ;
```

```
lock (acct) ;  
write (acct) ;  
unlock (acct) ;
```

## Transaction 2

```
lock (acct) ;  
read (acct) ;  
unlock (acct) ;
```

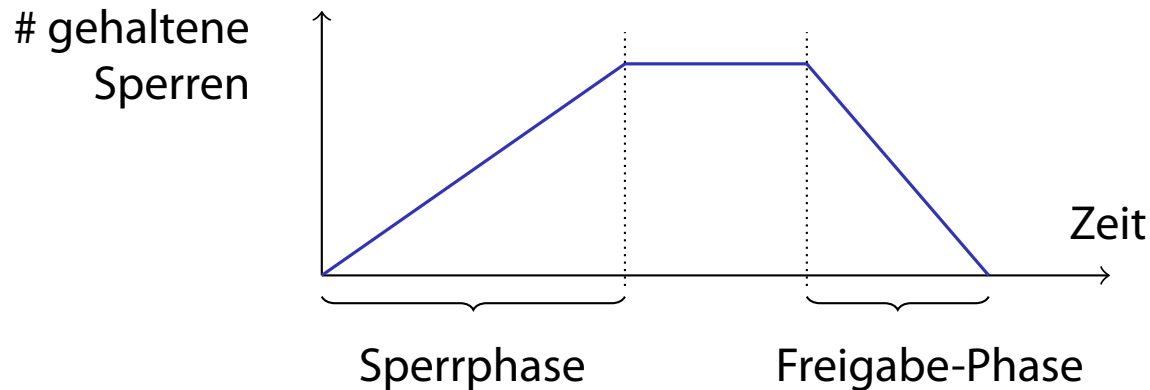
```
lock (acct) ;  
write (acct) ;  
unlock (acct) ;
```

# ATM-Transaktion mit Sperren

Transaction 1	Transaction 2	DB state
<code>lock (acct) ;</code> <code>read (acct) ;</code> <code>unlock (acct) ;</code>		1200
<code>lock (acct) ;</code> <code>write (acct) ;</code> <code>unlock (acct) ;</code>	<code>lock (acct) ;</code> <code>read (acct) ;</code> <code>unlock (acct) ;</code>	1100
	<code>lock (acct) ;</code> <code>write (acct) ;</code> <code>unlock (acct) ;</code>	1000

# Zwei-Phasen-Sperrverwaltung

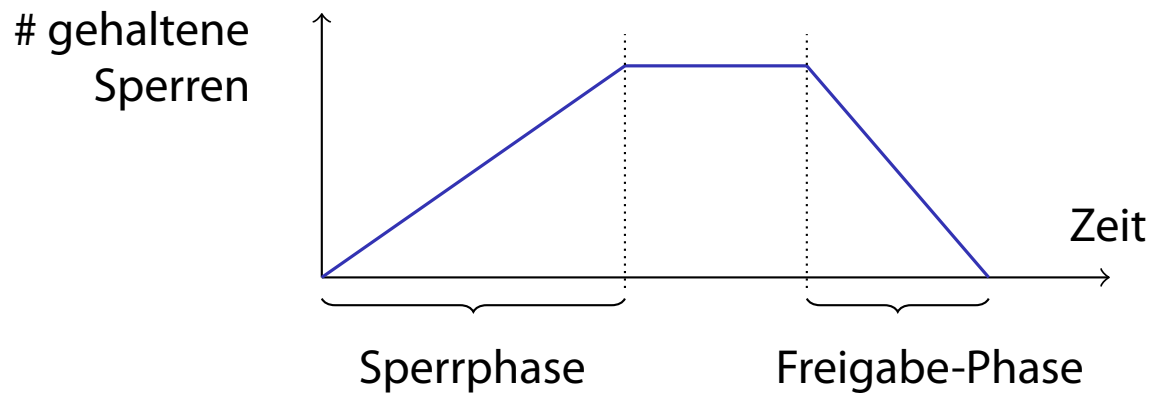
- Das Zwei-Phasen-Sperrprotokoll (Two-Phase Locking, **2PL**) führt eine weitere Einschränkung ein
- Sobald eine Transaktion eine Sperre freigegeben hat, darf sie keine weiteren Sperren anfordern



- Zwei-Phasen-Sperrprotokoll wird in jedem Datenbanksystem verwendet

# Aufgabe:

- Warum werden Sperren nicht auf einmal angefordert bzw. zurückgegeben?



# Noch einmal: ATM-Transaktion

Transaction 1	Transaction 2	DB state
<code>lock (acct) ;</code> <code>read (acct) ;</code> <code>unlock (acct) ;</code>		1200
<code>lock (acct) ;</code> ⚡ <code>write (acct) ;</code> <code>unlock (acct) ;</code>	<code>lock (acct) ;</code> <code>read (acct) ;</code> <code>unlock (acct) ;</code>	1100
	<code>lock (acct) ;</code> ⚡ <code>write (acct) ;</code> <code>unlock (acct) ;</code>	1000

# Eine 2PL-konforme Transaktion

---

- Zweiphasigkeit muss eingehalten werden

```
1 lock (acct) ;           } lock phase
2 bal ← read_bal (acct) ;
3 bal ← bal - 100 CHF ;
4 write_bal (acct, bal) ;
5 unlock (acct) ;        } unlock phase
```



# Entstehender Abarbeitungsplan

Transaction 1	Transaction 2	DB state
lock ( <i>acct</i> ) ; read ( <i>acct</i> ) ;		1200
write ( <i>acct</i> ) ; unlock ( <i>acct</i> ) ;	lock ( <i>acct</i> ) ; ↓ Transaction ↓ blocked	1100
	read ( <i>acct</i> ) ; write ( <i>acct</i> ) ; unlock ( <i>acct</i> ) ;	900

- Verwendung von Sperren führt zu einem korrekten (und serialisierbaren) Plan

# Sperrarten (Sperrmodi)

- Wir haben gesehen, dass zwei Leseoperationen nicht in Konflikt zueinander stehen
- Systeme verwenden verschiedene Arten von Sperren
  - Lesesperren (read locks, shared locks): Modus S
  - Schreibsperren (write locks, exclusive locks): Modus X
- Lock stehen nur in Konflikt zueinander, wenn eines davon eine X-Sperre ist:

	shared (S)	exclusive (X)
shared (S)		×
exclusive (X)	×	×

- Es ist eine sichere Operation beim 2PL, während der Sperrphase eine S-Sperre in eine X-Sperre zu **konvertieren**

# Verklemmungen (Deadlocks)

- Wie bei vielen Sperrprotokollen, kann es beim Zwei-Phasen-Sperrprotokoll zu Verklemmungen kommen:

Transaction 1	Transaction 2
lock (A) ;	
⋮	lock (B)
do something	⋮
⋮	do something
lock (B)	⋮
[ wait for $T_2$ to release lock ]	lock (A)
	[ wait for $T_1$ to release lock ]

- Durch Sperren ist der Ressourcenzugriff exklusiv
- Beide Transaktionen warten wechselseitig aufeinander
- Ressource werden nicht einzeln entzogen (no preemption)

# Behandlung von Verklemmungen

---

## Erkennung von Verklemmungen:

- System verwaltet einen **Wartet-auf-Graphen**, in dem eine Kante  $T_1 \rightarrow T_2$  bedeutet, dass  $T_1$  blockiert ist durch eine Sperre, die von  $T_2$  gehalten wird
- Periodisch wird nach **Zyklen** in diesem Graph gesucht
- Wenn Zyklus entdeckt, wird die **Verklemmung** durch Abbruch (abort) einer der Transaktionen **aufgelöst**
- Wahl des **Opfers** durchaus schwierig:
  - Abbruch **junger** Transaktionen führt zu Mehrfachausführung
  - Abbruch **alter** Transaktionen führt zu Verlust von vielen ausgeführten Operationen

# Behandlung von Verklemmungen

## Zeitüberschreitung (Timeout):

- Warte nur auf Sperre bis Zeitüberschreitung eintritt, sonst nehme Verklemmung an und breche ab



IBM DB2 UDB:

```
db2 => GET DATABASE CONFIGURATION;  
:  
Interval for checking deadlock (ms)      (DLCHKTIME) = 10000  
Lock timeout (sec)                      (LOCKTIMEOUT) = -1
```

## Vermeidung von Verklemmungen:

- Wait-Die
- Wound-Wait

	Wait/Die	Wound/Wait
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

Wikipedia

## Bankier-Algorithmus (leider Ressourcen nicht a priori bekannt)



# Wait-Die

---

- Fordert eine Transaktion eine Sperre an, die von einer jüngeren gehalten wird, so wartet die ältere bis die Sperre von der jüngeren freigegeben wird.
- Fordert eine Transaktion eine Sperre an, die von einer älteren gehalten wird, so bricht sie sich selber ab (genauer: sie startet neu, allerdings behält sie ihren alten Zeitstempel bei, um so „älter“ zu wirken und ihre Chancen zu erhöhen, diesmal komplett durchlaufen zu können)

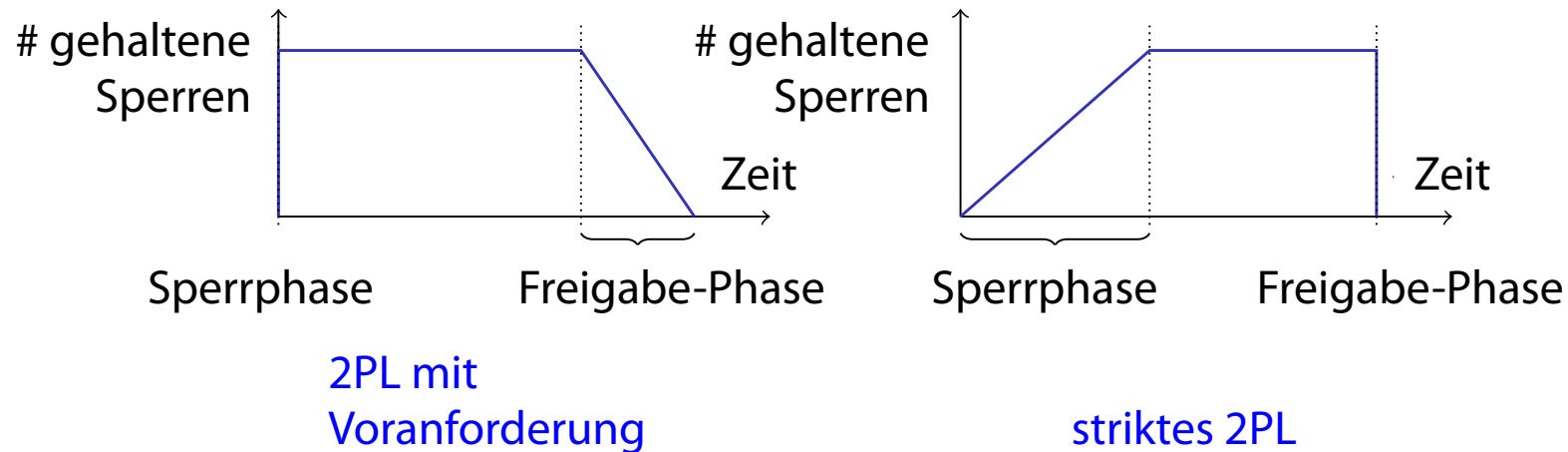
# Wound-Wait

---

- Fordert eine Transaktion eine Sperre an, die von einer jüngeren gehalten wird, so wird die jüngere abgebrochen (genauer: neu gestartet) und die ältere bekommt die Sperre zugewiesen.
- Fordert eine Transaktion eine Sperre an, die von einer älteren gehalten wird, so wartet sie, bis die Sperre von der älteren wieder freigegeben wird.

# Varianten des Zwei-Phasen-Sperrprotokolls

- Es gibt **Freiheitsgrade** bzgl. der Akquise- und Rückgabezeit von Sperren
- Mögliche Varianten

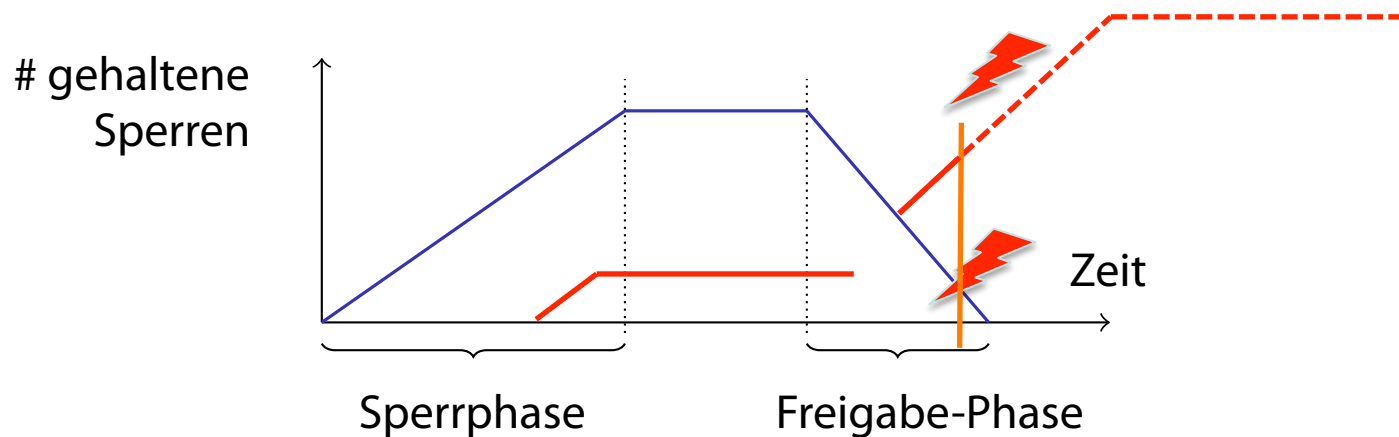


- Wodurch könnten die Varianten motiviert sein?



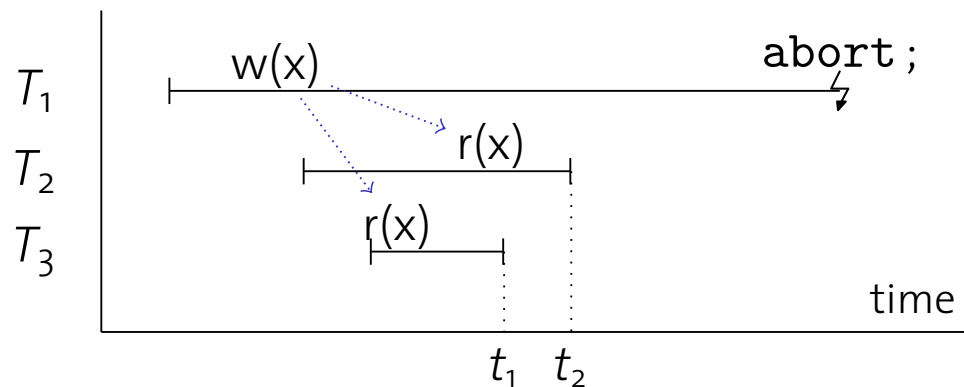
# Aufgabe:

- Was kann passieren, wenn Sperren nicht in einem Schritt zurückgegeben werden?



# Kaskadierendes Rücksetzen

Betrachten wir drei Transaktionen



- Wenn Transaktion  $T_1$  abgebrochen und zurückgerollt wird, haben Transaktionen  $T_2$  und  $T_3$  schon Daten gelesen, die von  $T_2$  erzeugt wurden (Dirty Read)
- $T_2$  und  $T_3$  müssen ebenfalls zurückgerollt werden
- $T_2$  und  $T_3$  können nicht abgeschlossen werden bis  $T_1$  fertig ist
- Zwei-Phasen-Sperrung vs. strikte Zwei-Phasen-Sperrung

# Phantom-Problem

---

Transaction 1	Transaction 2	Effect
scan relation $R$ ;	insert new row into $R$ ;	$T_1$ locks all rows
scan relation $R$ ;	commit;	$T_2$ locks new row
		$T_2$ 's lock released
		reads <b>new</b> row, too!

- Obwohl beide Relationen dem 2PL-Protokoll folgen, sieht  $T_1$  einen Effekt von  $T_2$
- Ursache des Problems:  
 $T_1$  kann nur **existierende** Tupel sperren

# Implementierung eines Sperrverwalters

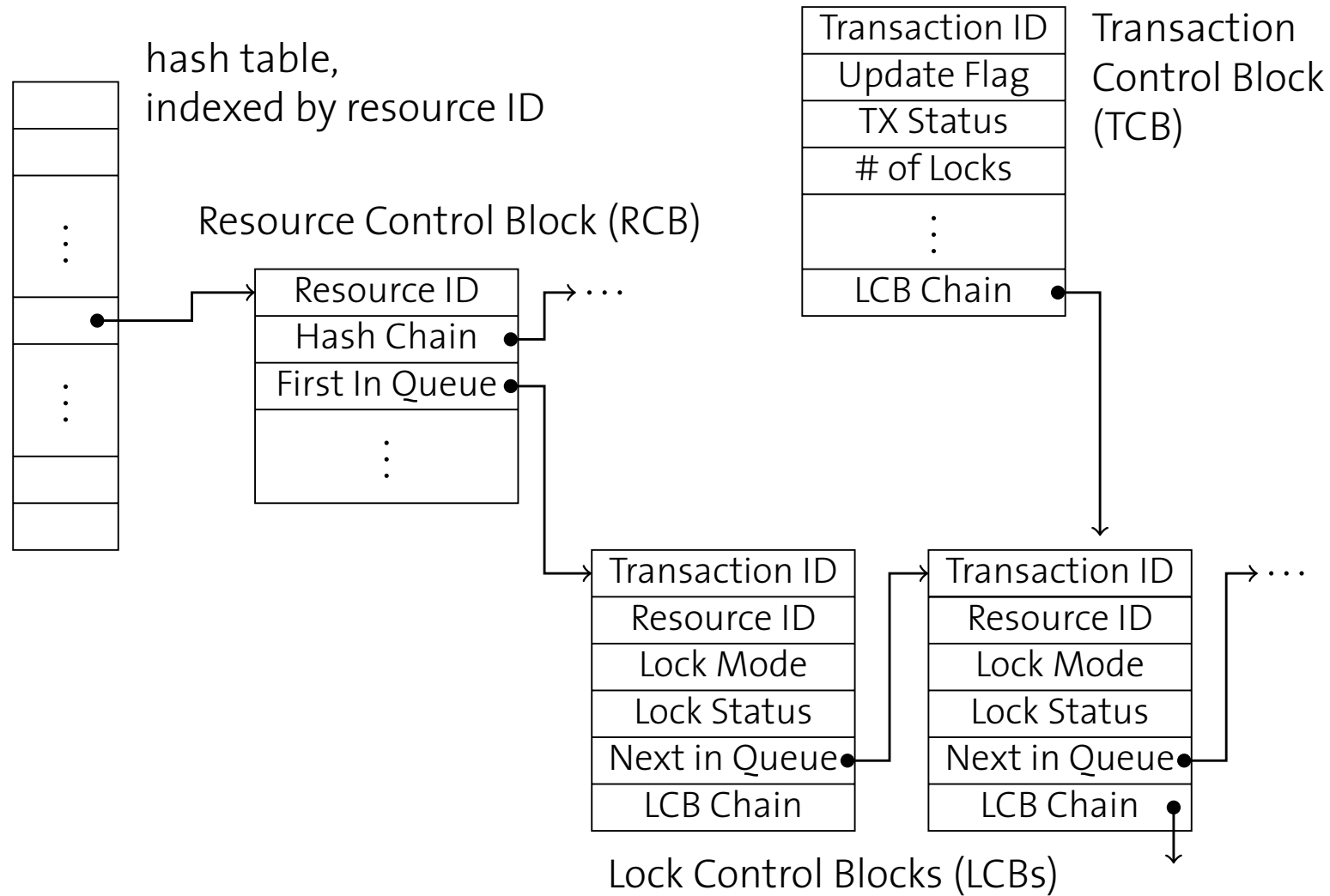
---

Ein Sperrverwalter muss drei Aufgaben effektiv erledigen:

1. Prüfen, welche **Sperren für eine Ressource** gehalten werden (um eine Sperranforderung zu behandeln)
2. Bei Sperr-Rückgabe müssen die **Transaktionen**, die die Sperre haben wollen, schnell **identifizierbar** sein
3. Wenn eine Transaktion beendet wird, müssen alle von der Transaktion angeforderten und gehaltenen **Sperren zurückgegeben** werden

Wie muss eine Datenstruktur aussehen, mit der diese Anforderungen erfüllt werden können?

# Datenstruktur zur Buchführung



# Implementierung von Aufgaben

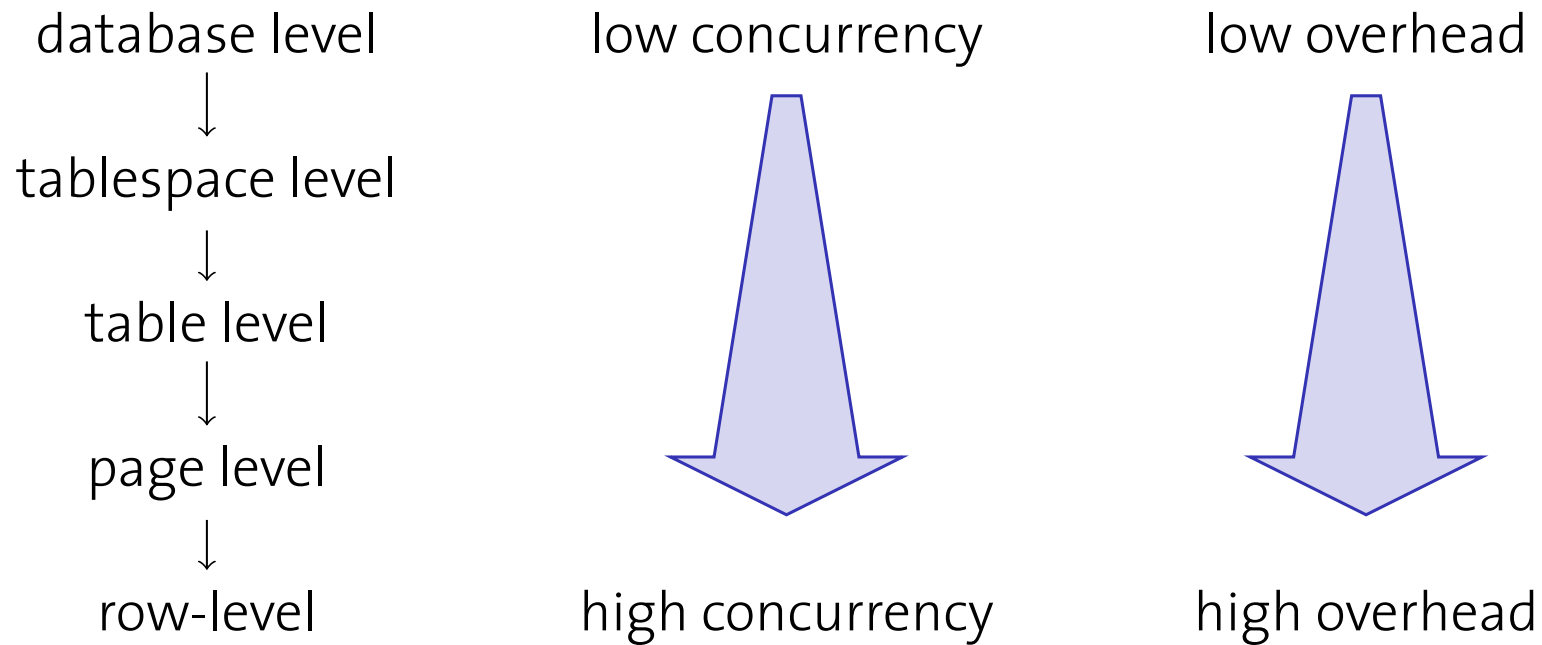
---

1. Sperren für eine Ressource können über Hashzugriff gefunden werden
  - Verkettete Liste der Lock Control Blocks über ‚First In Queue/Next in Queue‘ (alle Anfragen enthalten, stattgegeben oder nicht)
  - Transaktion(en) am Kopf der Liste hält/halten Sperre für die Ressource
2. Wenn eine Sperre zurückgegeben wird (LCB aus der Liste entfernt), können die nächsten Transaktionen berücksichtigt werden
3. Sperren einer beendeten Transaktion können über ‚LCB Chain‘ identifiziert und zurückgegeben werden

# Granularität des Sperrens

---

Die Granularität des Sperrens unterliegt Abwägung



➤ Sperren mit multipler Granularität

# Sperren mit multipler Granularität

- Entscheide die Granularität von Sperren für jede Transaktion (abhängig von ihrer Charakteristik)
  - Tupel-Sperre z.B. für

```
SELECT *  
FROM CUSTOMERS Q1  
WHERE C_CUSTKEY = 42
```

- und eine Tabellen-Sperre für

```
SELECT * FROM CUSTOMERS Q2
```

- Wie können die Sperren für die Transaktionen koordiniert werden?
  - Für Q<sub>2</sub> sollen nicht für alle Tupel umständlich Sperrkonflikte analysiert werden



# Vorhabens-Sperren

Datenbanken setzen Vorhabens-Sperren (intention locks) für verschiedenen Sperrgranularitäten ein

- Sperrmodus **Intention Share**: IS
- Sperrmodus **Intention Exclusive**: IX
- Konfliktmatrix:

	S	X	IS	IX
S		×		×
X	×	×	×	×
IS		×		
IX	×	×		

- Eine Sperre **I**  auf einer größeren Ebene bedeutet, dass es eine Sperre  auf einer niederen Ebene gibt

# Vorhabens-Sperren

---

Protokoll für Sperren auf mehreren Ebenen:

1. Eine Transaktion kann jede Ebene  $g$  in Modus  $\square \in \{S, X\}$
2. Bevor Ebene  $g$  in Modus  $\square$  gesperrt werden kann, muss eine Sperre  $I\square$  für alle größeren Ebenen gewonnen werden

Anfrage  $Q_1$  würde

- eine IS-Sperre für Tabelle **CUSTOMERS** anfordern (auch für Tablespace und die Datenbank) und dann
- eine S-Sperre auf dem Tupel mit **C\_CUSTKEY=42** akquirieren

Anfrage  $Q_2$  würde eine

- S-Sperre für die Tabelle **CUSTOMERS** anfordern (und eine IS-Sperre auf dem Tablespace und der Datenbank)

# Entdeckung von Konflikten

---

Nehmen wir an, folgende Anfrage ist zu bearbeiten

```
UPDATE CUSTOMERS
SET NAME = 'John Doe'
WHERE C_CUSTKEY = 17
```

Hierfür wird

- eine IX-Sperre auf Tabelle **CUSTOMERS** (und ...) sowie
- eine X-Sperre auf dem Tupel mit Kunde 17

Diese Anfrage ist

- kompatibel mit  $Q_1$  (kein Konflikt zw. IX und IS auf der Tabellenebene)
- aber inkompatibel mit  $Q_2$  (die S-Sperre von  $Q_2$  steht in Konflikt mit der IX-Sperre bzgl.  $Q_3$ )



# Konsistenzgarantien in SQL-92

---

In einigen Fall kann man mit einigen kleinen Fehlern im Anfrageergebnis leben

- „Fehler“ bezüglich einzelner Tupel machen in Aggregatfunktionen evtl. kaum bemerkbar
  - Lesen inkonsistenter Werte (inconsistent read anomaly)
- In SQL-92 kann man Isolations-Modi spezifizieren:

```
SET ISOLATION SERIALIZABLE;
```

- Es gibt weniger strikte Modi, unter denen die Performanz höher ist (weniger Verwaltungsaufwand z.B. für Sperren)

# SQL-92 Isolations-Modi

---

- **Read uncommitted** (auch: 'dirty read' oder 'browse')
  - Nur Schreibsperrern akquiriert (nach 2PL)
- **Read committed** (auch 'cursor stability')
  - Lesesperrern werden nur gehalten, sofern der Zeiger auf das betreffende Tupel zeigt, Schreibsperrern nach 2PL
- **Repeatable read** (auch 'read stability')
  - Lese- und Schreibsperrern nach 2PL akquiriert
- **Serializable**
  - Zusätzliche Sperranforderungen, um Phantomproblem zu begegnen

# Resultierende Konsistenzgarantien

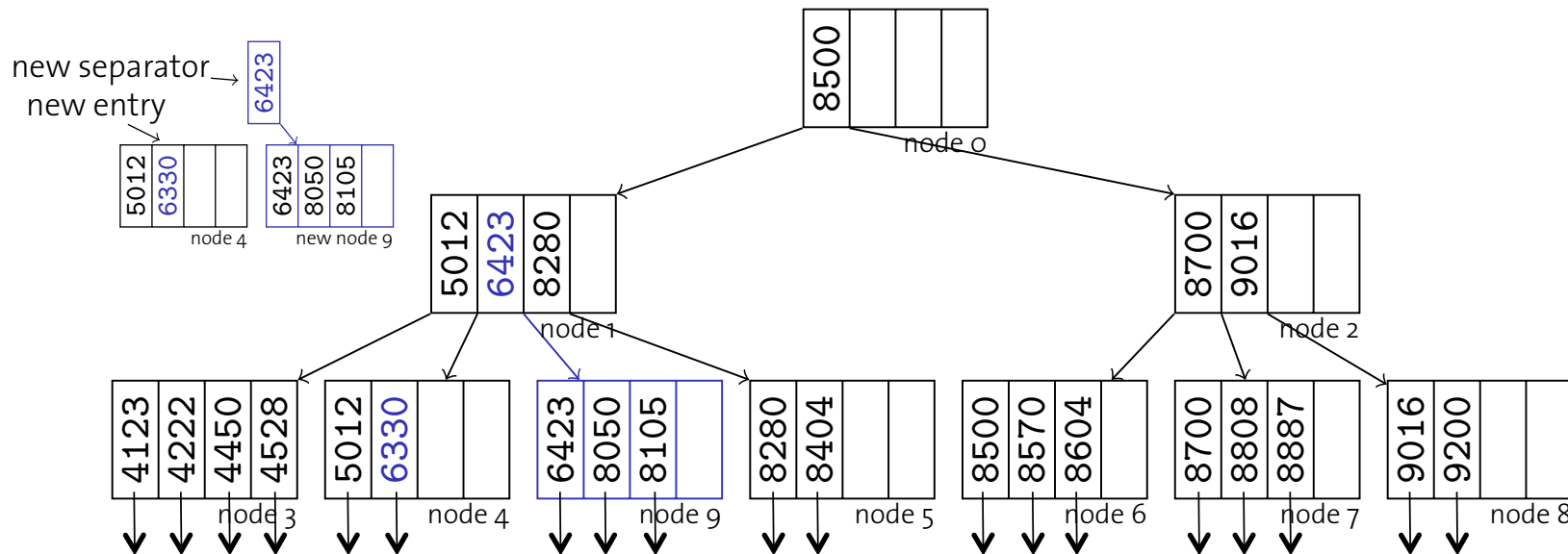
---

isolation level	dirty read	non-repeat. rd	phantom rd
read uncommitted	possible	possible	possible
read committed	not possible	possible	possible
repeatable read	not possible	not possible	possible
serializable	not possible	not possible	not possible

- Einige Implementierungen unterstützen mehr, weniger oder andere Isolationmodi (isolation levels)
- Nur wenige Anwendung benötigen (volle) Serialisierbarkeit

# Nebenläufigkeit beim Indexzugriff

- Betrachten wir eine Transaktion  $T_w$ , die etwas in einen B-Baum einführt, was zu einer Splitoperation führt



- Einfügung eines Eintrags mit Schlüssel **6330**
  - Knoten 4 aufgespalten
  - Neuer Separator in Knoten 1

# Nebenläufigkeit beim Indexzugriff

---

- Nehme an, die Aufspaltung ist gerade erfolgt, aber der neue Separator **6423** ist noch nicht etabliert
- Nehme weiterhin an, ein nebenläufiges Lesen in Transaktion  $T_r$  sucht nach **8085**
  - Die Verzeigerung weist auf Knoten node1
  - Knoten node1 enthält aber **8050** nicht mehr, also wird der entsprechende Datensatz nicht gefunden
- Auch in B-Bäumen muss beim Umbau mit Sperren gearbeitet werden!



# Sperren und B-Baum-Indexe

---

Betrachten wir B-Baum-Operationen

- Für die Suche erfolgt ein Top-Down-Zugriff
- Für Aktualisierungen ...
  - erfolgt erst eine Suche,
  - dann werden Daten ggf. in ein Blatt eingetragen und
  - ggf. werden Aufspaltungen von Knoten nach oben propagiert
- Nach dem Zwei-Phasen-Sperrprotokoll ...
  - müssen S/X-Sperren auf dem Weg nach unten akquiriert werden (Konversion provoziert ggf. Verklemmungen)
  - müssen alle Sperren bis zum Ende gehalten werden

# Sperren und B-Baum-Indexe

---

- Diese Strategie reduziert die Nebenläufigkeit drastisch
- Alle Transaktionen müssen warten, um die Sperre für die Wurzel des Index zu erhalten
- Wurzel wird dadurch zum Flaschenhals und serialisiert alle (Schreib-)Transaktionen
- Zwei-Phasen-Sperrprotokoll nicht angemessen für B-Baum-Indexstrukturen

# Sperrkopplung

---

Betrachten wir den Schreibe-Fall (alle Sperren mit Konflikt)

- Das Protokoll Write-Only-Tree-Locking (WTL) garantiert Serialisierbarkeit
  - Für alle Baumknoten  $n$  außer der Wurzel kann eine Sperre nur akquiriert werden, wenn die Sperre für den Elternknoten akquiriert wurde
  - Sobald ein Knoten entsperrt wurde, kann für ihn nicht erneut eine Sperre angefordert werden (2PL)

Und damit gilt:

- Alle Transaktionen folgen Top-Down-Zugriffsmuster
- Keine Transaktion kann dabei andere überholen
- WTL-Protokoll ist verklemmungsfrei

# Aufspaltungssicherheit

---

- Wir müssen auf dem Weg nach unten in den B-Baum Schreibsperrern wegen möglicher Aufspaltungen halten
- Allerdings kann man leicht prüfen, ob ein Spaltung von Knoten  $n$  die Vorgänger überhaupt erreichen kann
  - Wenn  $n$  weniger als  $2d$  Einträge enthält kommt es nicht zu einer Weiterreichung der Aufspaltung nach oben
- Ein Knoten, der diese Bedingung erfüllt, heißt aufspaltungssicher (split safe)
- Ausnutzung zur frühen Sperrrückgabe
  - Wenn ein Knoten auf dem Weg nach unten als aufspaltungssicher gilt, können alle Sperren der Vorgänger zurückgegeben werden
  - Sperren werden weniger lang gehalten

# Sperrkopplungsprotokoll (Variante 1)

---

```
1 place S lock on root ; readers
2 current ← root ;
3 while current is not a leaf node do
4   | place S lock on appropriate son of current ;
5   | release S lock on current ;
6   | current ← son of current ;
```

```
1 place X lock on root ; writers
2 current ← root ;
3 while current is not a leaf node do
4   | place X lock on appropriate son of current ;
5   | current ← son of current ;
6   | if current is safe then
7   |   | release all locks held on ancestors of current ;
```

# Erhöhung der Nebenläufigkeit

---

- Auch mit Sperrkopplung werden eine beträchtliche Anzahl von Sperren für innere Knoten benötigt (wodurch die Nebenläufigkeit gemindert wird)
- Innere Knoten selten durch Aktualisierungen betroffen
  - Wenn  $d=50$ , dann Aufspaltung bei jeder 50. Einfügung (2% relative Auftretenshäufigkeit)
- Eine Einfügetransaktion könnte optimistisch annehmen, dass keine Aufspaltung nötig ist
  - Bei inneren Knoten werden während der Baumtraversierung nur Lesesperren akquiriert (inkl. einer Schreibsperre für das betreffende Blatt)
  - Wenn die Annahme falsch ist, traversiere Indexbaum erneut unter Verwendung korrekter Schreibsperren

# Sperrkopplungsprotokoll (Variante 2)

---

## Modifikationen nur für Schreibvorgänge

```
1 place S lock on root ;
2 current ← root ;
3 while current is not a leaf node do
4   | son ← appropriate son of current ;
5   | if son is a leaf then
6   |   | place X lock on son ;
7   |   | else
8   |   |   | place S lock on son ;
9   |   |   | release lock on current ;
10  |   |   | current ← son ;
11  | if current is unsafe then
12  |   | release all locks and repeat with protocol Variant 1 ;
```

# Zusammenfassung

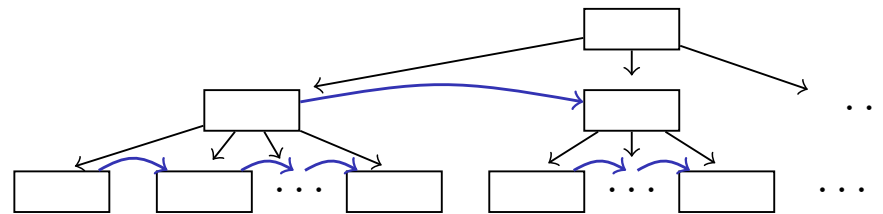
---

- Wenn eine Aufspaltung nötig ist, wird der Vorgang abgebrochen und erneut aufgesetzt
- Die resultierende Verarbeitung ist korrekt, obwohl es nach einem erneuten Sperren aussieht (was für WTL nicht erlaubt ist)
- Der Nachteile von Variante 2 ist, das im Falle einer Blattaufspaltung Arbeit verloren ist
- Es gibt viele Varianten dieser Sperrprotokolle



# B+-Bäume ohne Lesesperren

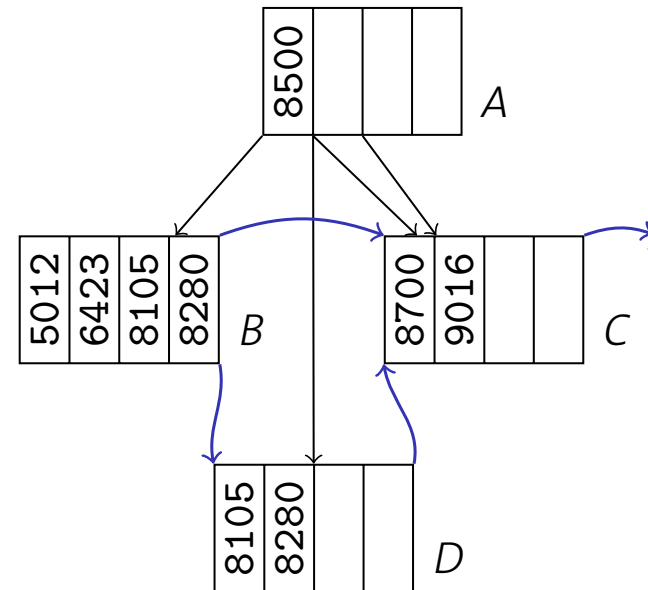
- Es gibt Vorschläge, ohne Lesesperren auf B-Baum-Knoten zu operieren
- Anforderung: ein Next-Zeiger zeigt auf rechten Geschwisterknoten



- Vorher schon betrachtet: Verkettete Liste auf Blattebene
- Zeiger stellen zweiten Pfad auf Knoten bereit
- Bei nebenläufigen Zugriffen und Aufspaltung von Knoten bleibt Zugriff auf Gesamtinformation möglich

# Einfügung mit Aufspaltung eines inneren Knotens

- 1 lock & read page  $B$  ;
- 2 create new page  $D$  and lock it ;
- 3 populate page  $D$  ;
- 4 set next pointer  $D \rightarrow C$  ;
- 5 write  $D$  ;
- 6 set next pointer  $B \rightarrow D$  ;
- 7 adjust content of  $B$  ;
- 8 write  $B$  ;
- 9 lock & read  $A$  ;
- 10 adjust content of  $A$  ;
- 11 write  $A$  ;



- Alle Indexeinträge sind zu jedem Zeitpunkt erreichbar

# B-Baum-Zugriff beim Lesen

---

- Die Next-Zeiger ermöglichen Leseoperationen, Einträge sogar inmitten einer Aufspaltung zu finden, auch wenn einige Einträge schon auf eine neue Seite verschoben wurden
- Während `tree_search()` können Lesetransaktionen auf die Geschwister eines Knotens `n` zugreifen sofern
  - in `n` kein entsprechender Eintrag gefunden wird und
  - `next` kein Nullzeiger ist
- Es wird nur auf Geschwister mit gleichem Elternteil verwiesen
- Schreibsperrern halten Lesetransaktionen nicht vom Zugriff auf eine Seite ab (Leser fordern keine Sperren an)
- Dieses Protokoll wird von PostgreSQL verwendet

# Sperren (Locks) und Indexsperren (Latches)

---

- Welches Annahmen müssen wir machen für Sperrfreien Lesezugriff?
- Sperren sollte wir ja nicht verwenden
- Leichtgewichtige Sperren für kurzfristige atomare Ops
- Indexsperren induzieren wenige Verwaltungsaufwand (meist als Spinlocks implementiert)
- Sie sind nicht unter der Kontrolle des Sperrverwalters (es gibt keine Verklemmungsüberwachung oder automatisches Zurückgeben der Sperren bei Transaktionsabbruch)

# Optimistische Organisation der Nebenläufigkeit

---

- Bisher waren wir pessimistisch
  - Wir haben uns immer den schlimmste Fall vorgestellt und durch Sperrverwaltung vermieden
- In der Praxis kommt der schlimmste Fall gar nicht sehr oft vor (siehe auch die Isolationsmodi)
- Wir können auch das beste hoffen und nur im Fall eines Konflikts besondere Maßnahmen ergreifen
- Führt auf die Idee der Optimistischen Kontrolle der Nebenläufigkeit

# Optimistische Organisation der Nebenläufigkeit

---

## Behandle Transaktionen in drei Phasen

- Lesephase: Führe Transaktion aus, aber schreibe Daten nicht sofort auf die Platte, halte Kopien in einem privaten Arbeitsbereich
- Validierungsphase: Wenn eine Transaktion erfolgreich abgeschlossen wird (commit), teste ob Annahmen gerechtfertigt waren. Falls nicht, führe doch noch einen Abbruch durch
- Schreibphase: Transferiere Daten vom privaten Arbeitsbereich in die Datenbasis

# Validierung von Transaktionen

---

Validierung wird üblicherweise implementiert durch Betrachtung der

- Gelesenen Attribute (read set  $RS(T_i)$ )
- Geschriebene Attribute (write set  $WS(T_i)$ )

# Validierung von Transaktionen

---

- Rückwärtsorientierte optimistische Nebenläufigkeitsverwaltung
  - Vergleich T bezüglich aller erfolgreich beendeter (committed) Transaktionen
  - Test ist erfolgreich, wenn  $T_c$  beendet wurde bevor T gestartet wurde oder  $RS(T) \cap WS(T_c) = \emptyset$
- Vorwärtsorientierte optimistische Nebenläufigkeitsverwaltung
  - Vergleiche T bezüglich aller laufenden Transaktionen  $T_r$
  - Test ist erfolgreich, wenn  $WS(T) \cap RS(T_r) = \emptyset$



# Multiversionen-Nebenläufigkeitsorganisation

---

Betrachten wir den folgenden Abarbeitungsplan

$r_1(x), w_1(x), r_2(x), w_2(y), r_1(y), w_1(z)$

t  
↓

Ist dieser Plan serialisierbar?

- Angenommen, wenn T1 y lesen möchte, sei der „alte“ Wert vom Zeitpunkt **t** noch verfügbar
- Dann könnten wir eine Historie wie folgt erzeugen

$r_1(x), w_1(x), r_2(x), r_1(y), w_2(y), w_1(z)$

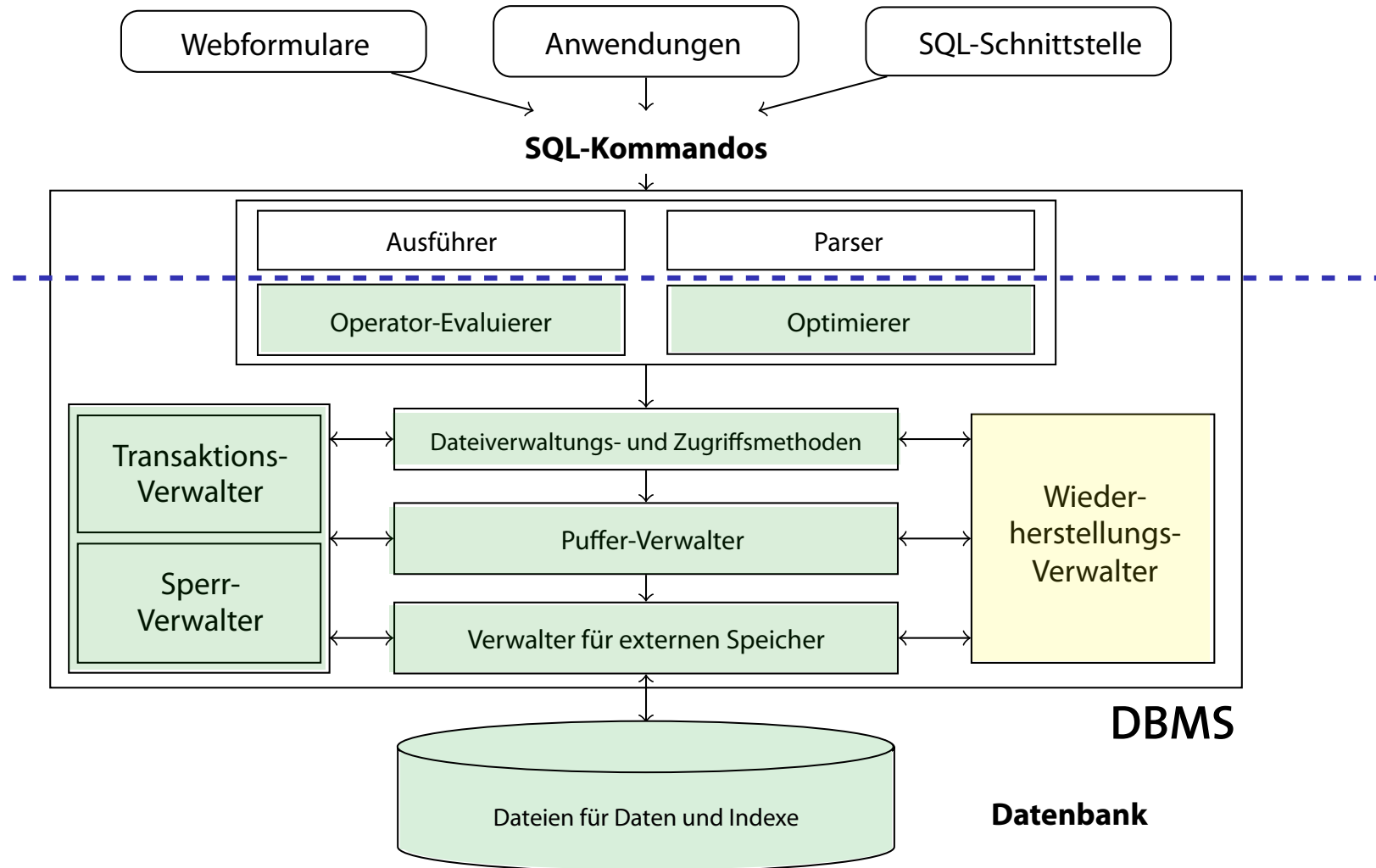
die serialisierbar ist

# Multiversionen-Nebenläufigkeitsorganisation

---

- Mit verfügbaren alten Objektversionen müssen Leseschritte nicht länger blockiert werden
- Es sind „abgelaufene“, aber konsistente Werte verfügbar
- Problem: Versionierung benötigt Raum und erzeugt Verwaltungsaufwand (Garbage Collection)
  
- Einige Systeme unterstützen Schnappschussisolation
  - Oracle, SQL Server, PostgreSQL

# Wiederherstellung (Recovery)



# Wiederherstellung nach Fehlern

---

## Drei Typen von Fehlern

- **Transaktionsfehler (Prozessfehler)**
  - Eine Transaktion wird abgebrochen (abort)
  - Alle Änderungen müssen ungeschehen gemacht werden
- **Systemfehler**
  - Datenbank- oder Betriebssystem-Crash, Stromausfall, o.ä.
  - Änderungen im Hauptspeicher sind verloren
  - Sicherstellen, dass keine Änderungen mit Commit verloren gehen (oder ihre Effekte wieder herstellen) und alle anderen Transaktionen ungeschehen gemacht werden
- **Medienfehler (Gerätefehler)**



# Wiederherstellung nach Fehlern

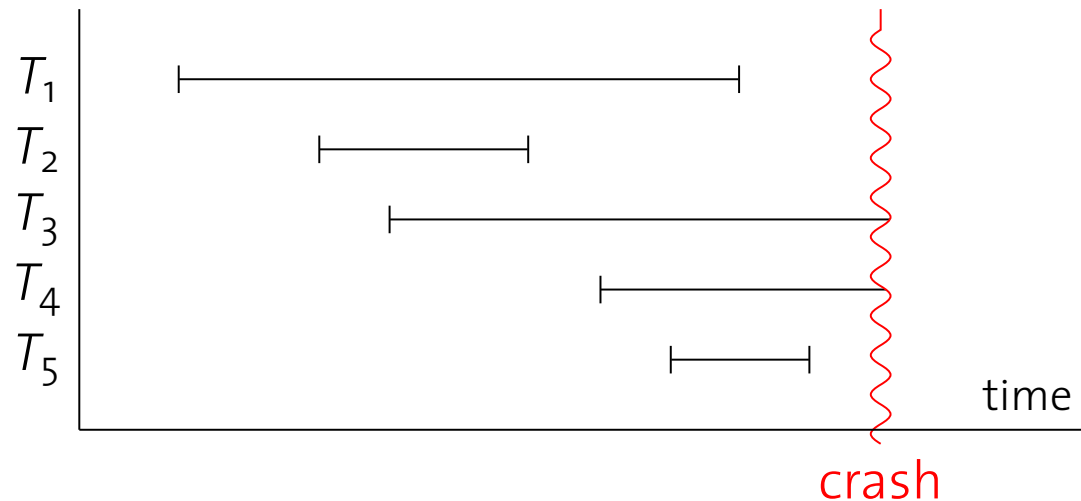
---

## Drei Typen von Fehlern

- Transaktionsfehler (Prozessfehler)
- Systemfehler
- Medienfehler (Gerätefehler)
  - Crash von Festplatten, Feuer, Wassereinbruch
  - Wiederherstellung von externen Speichermedien

Trotz Fehler müssen Atomarität und Durabilität garantiert werden (ACID-Bedingungen)

# Beispiel: Systemausfall (oder Medienfehler)



- Transaktionen  $T_1$ ,  $T_2$  und  $T_3$  wurden vor dem Ausfall erfolgreich beendet → Dauerhaftigkeit: Es muss sichergestellt werden, dass die Effekte beibehalten werden oder wiederhergestellt werden können (redo)
- Transaktionen  $T_3$  und  $T_4$  wurden noch nicht beendet → Atomarität: Alle Effekte müssen rückgängig gemacht werden

# Arten von Speichern

---

Wir nehmen an, es gibt drei Arten von Speichern

- Flüchtige Speicher
  - Wird vom Pufferverwalter verwendet (auch für Cache und Write-Ahead-Log, s.u.)
- Nicht-flüchtiger Speicher
  - Festplatten
- Stabiler Speicher
  - Nicht-flüchtiger Speicher, der alle drei Arten von Fehlern überlebt. Stabilität kann durch Replikation auf mehrere Platten erhöht werden (auch: Bänder)

Vergleiche Arten von Fehler und Arten von Speichern

# Schatten-Seiten-Verwaltung

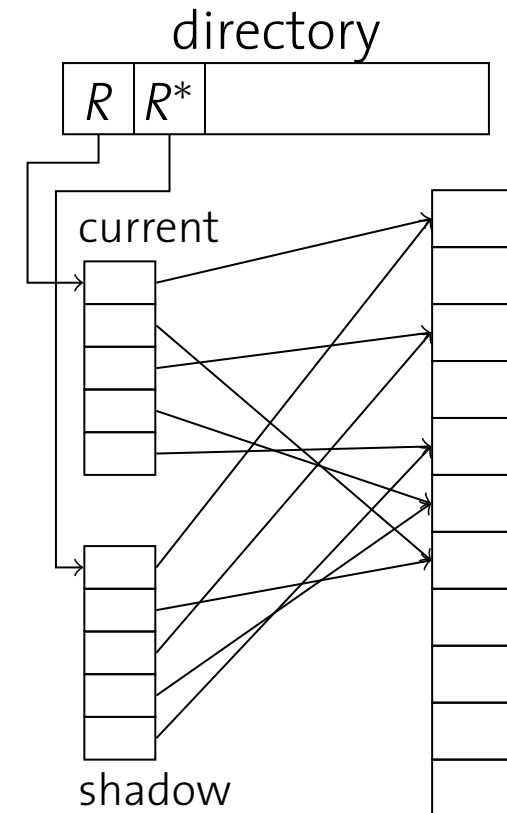
---

- Fehler können zu jeder Zeit auftreten, also muss das System jederzeit in einen konsistenten Zustand zurückgeführt werden können
- Dies kann durch Redundanz erreicht werden
- Schatten-Seiten (eingeführt durch IBMs „System R“)
  - Von jeder Seite werden zwei Versionen geführt
  - Aktuelle Version (Arbeitskopie, copy-on-write)
  - Schatten-Seite (konsistente Version auf nicht-flüchtigem Speicher zur Wiederherstellung)
- Operation *SAVE*, um aktuellen Zustand als Schatten-Version zu sichern (für Commit)
- Operation *RESTORE*, um Wiederherstellung einzuleiten (für Abort)



# Schatten-Seiten-Verwaltung

- Am Anfang: shadow = current
- Ein Transaktion **T** ändert nun die aktuelle Version
  - Aktualisierung nicht in situ
  - Neue Seiten anfordern, kopieren, ändern und Seitentabelle anpassen
- Wenn **T** abgebrochen wird, überschreibe aktuelle Version mit Schatten-Version
- Wenn **T** beendet wird, aktualisiere Info in Verzeichnis, um aktuelle Version persistent zu machen
- Gewinne ggf. Seiten durch Garbage Collection wieder



# Schatten-Seiten: Diskussion

---

- Wiederherstellung schnell für ganze Relationen/Dateien
- Um Persistenz (Durabilität) sicherzustellen, müssen modifizierte Seiten bei einem Commit in nicht-flüchtigen Speicher (z.B. Festplatte) geschrieben werden (force to disk)
- Nachteile:
  - Hohe I/O-Kosten, keine Verwendung von Cache möglich
  - Langsame Antwortzeiten
- Besser: No-Force-Strategie, Verzögerung des Schreibens
- Transaktion muss neu abgespielt werden können (Redo), auch für Änderungen, die nicht gespeichert wurden

# Schatten-Seiten: Diskussion

---

- Schatten-Seiten ermöglichen das Stehlen der Rahmen im Pufferverwalter (frame stealing): Seiten werden möglicherweise sofort auf die Platte geschrieben (sogar bevor Transaktion erfolgreich beendet wird)
  - Stehlen erfolgt durch andere Transaktionen
  - Stehlen kann nur erfolgen, wenn Seite nicht gepinnt
  - Geänderte Seiten (dirty pages) werden auf die Platte geschrieben
- Diese Änderungen müssen ungeschehen gemacht werden während der Wiederherstellung
  - Leicht möglich durch Schatten-Seiten

# Effekte für die Wiederherstellung

---

- Entscheidungen zur Strategie haben Auswirkungen auf das, was bei der Wiederherstellung erfolgen muss

	force	no force
no steal	no redo no undo	must redo no undo
steal	no redo must undo	must redo must undo

- Bei **steal** und **no force** wird zur Erhöhung der Nebenläufigkeit und der Performanz ein **redo** und ein **undo** implementiert

# Write-Ahead-Log (WAL)

---

- Die ARIES<sup>1</sup>-Wiederherstellungsmethode verwendet ein Write-Ahead-Log zur Implementierung der notwendigen redundanten Datenhaltung
- Datenseiten werden in situ modifiziert
- Für ein Undo müssen Undo-Informationen in eine Logdatei auf nicht-flüchtigem Speicher geschrieben werden bevor eine geänderte Seite auf die Platte geschrieben wird
- Zur Persistenzsicherung muss zur Commit-Zeit Redo-Information sicher gespeichert werden (No-Force-Strategie: Daten auf der Platte enthalten alte Information)

<sup>1</sup> Algorithm for Recovery and Isolation Exploiting Semantics  
Mohan et al. ARIES: A Transaction Recovery Method Supporting  
Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead  
Logging. ACM TODS, vol. 17(1), March 1992.

# Inhalte des Write-Ahead-Logs

---

LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
:	:	:	:	:	:	:	:

## LSN (Log Sequence Number)

- Monoton steigende Zahl, um Einträge zu identifizieren  
Trick: Verwende Byte-Position of Log-Eintrag

## Typ (Log Record Type)

- Repräsentiert, ob es ein Update-Eintrag (UPD), End-of-Transaction-Eintrag (EOT), Compensation-Log-Record (CLR)

## TX (Transaktions-ID)

- Transaktionsbezeichner (falls anwendbar)

# Inhalte des Write-Ahead-Logs (Forts.)

---

## Prev (Previous Log Sequence Number)

- LSN des vorigen Eintrags von der gleichen Transaktion (falls anwendbar, am Anfang steht '-')

## Page (Page Identifier)

- Seite, die aktualisiert wurde (nur für UPD und CLR)

## UNxt (LSN Next to be Undone)

- Nur für CLR: Nächster Eintrag der Transaktion, der während des Zurückrollens bearbeitet werden muss

## Redo

- Information zum erneuten Erzeugen einer Operation

## Undo

- Information zum Ungeschehenmachen einer Operation

# Beispiel

Transaction 1	Transaction 2	LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
$a \leftarrow \text{read}(A);$	$c \leftarrow \text{read}(C);$								
$a \leftarrow a - 50;$	$c \leftarrow c + 10;$								
$\text{write}(a,A);$	$\text{write}(c,C);$	1	UPD	$T_1$	-	...		$A := A - 50$	$A := A + 50$
$b \leftarrow \text{read}(B);$		2	UPD	$T_2$	-	...		$C := C + 10$	$C := C - 10$
$b \leftarrow b + 50;$									
$\text{write}(b,B);$		3	UPD	$T_1$	1	...		$B := B + 50$	$B := B - 50$
$\text{commit};$		4	EOT	$T_1$	3	...			
	$a \leftarrow \text{read}(A);$								
	$a \leftarrow a - 10;$								
	$\text{write}(a,A);$	5	UPD	$T_2$	2	...		$A := A - 10$	$A := A + 10$
	$\text{commit};$	6	EOT	$T_2$	5	...			



# Redo/Undo-Information

---

ARIES nimmt **seitenorientiertes Redo** an

- Keine anderen Seiten müssen angesehen werden, um eine Operation erneut zu erzeugen
- Z.B.: Physikalisches Logging
  - Speicher von Byte-Abbildern von (Teilen von) Seiteninhalten
  - Vorher-Abbild (Abbild vor der Operation)
  - Nachher-Abbild (Abbild nach der Operation)
- Wiederherstellung unabhängig von Objekten
  - Struktur braucht nicht bekannt zu sein, nur Seitenstruktur relevant
- Gegensatz: Logisches Redo („Setze Tupelwert auf v’“)
  - Redo muss vollständig durchgeführt werden, auch Indexeinträge werden neu generiert, inkl. Aufspaltung usw.

# Redo/Undo-Information

---

- ARIES unterstützt **logisches Undo**
- Seitenorientiertes Undo kann kaskadierende Rückroll-Situationen heraufbeschwören
  - Selbst wenn eine Transaktion  $T_1$  nicht direkt Tupel betrachtet hat, die von anderer Transaktion  $T_2$  beschrieben wurden, ist doch das physikalische Seitenlayout, das  $T_1$  sieht, von  $T_2$  beeinflusst
  - $T_1$  kann nicht von  $T_2$  erfolgreich beendet werden
- Logisches Undo erhöht also die Nebenläufigkeit
- Durch Operator-Logging können Transaktion voll nebenläufig arbeiten, wenn ihre Operationen semantisch kompatibel sind
  - Z.B.: Dekrement- und Inkrementoperationen
  - Dieses ist für Index und Metadaten relevant

# Schreiben von Log-Einträgen

---

- Aus Performanzgründen werden Log-Einträge zunächst in flüchtigen Speicher geschrieben
- Zu bestimmten Zeiten werden die Einträge bis zu einer bestimmten LSN in stabilen Speicher geschrieben
  - Alle Einträge bis zum EOT einer Transaktion T werden auf die Platte geschrieben wenn T erfolgreich beendet (um ein Redo von Ts Effekten vorzubereiten)
  - Wenn eine Datenseite p auf die Platte geschrieben wird, werden vorher die letzten Modifikationen von p im WAL auf die Platte geschrieben (zur Vorbereitung eines Undo)
- Die Größe des Logs nimmt immer zu (s. aber Schnappschüsse weiter unten)

# Normaler Verarbeitungsmodus

---

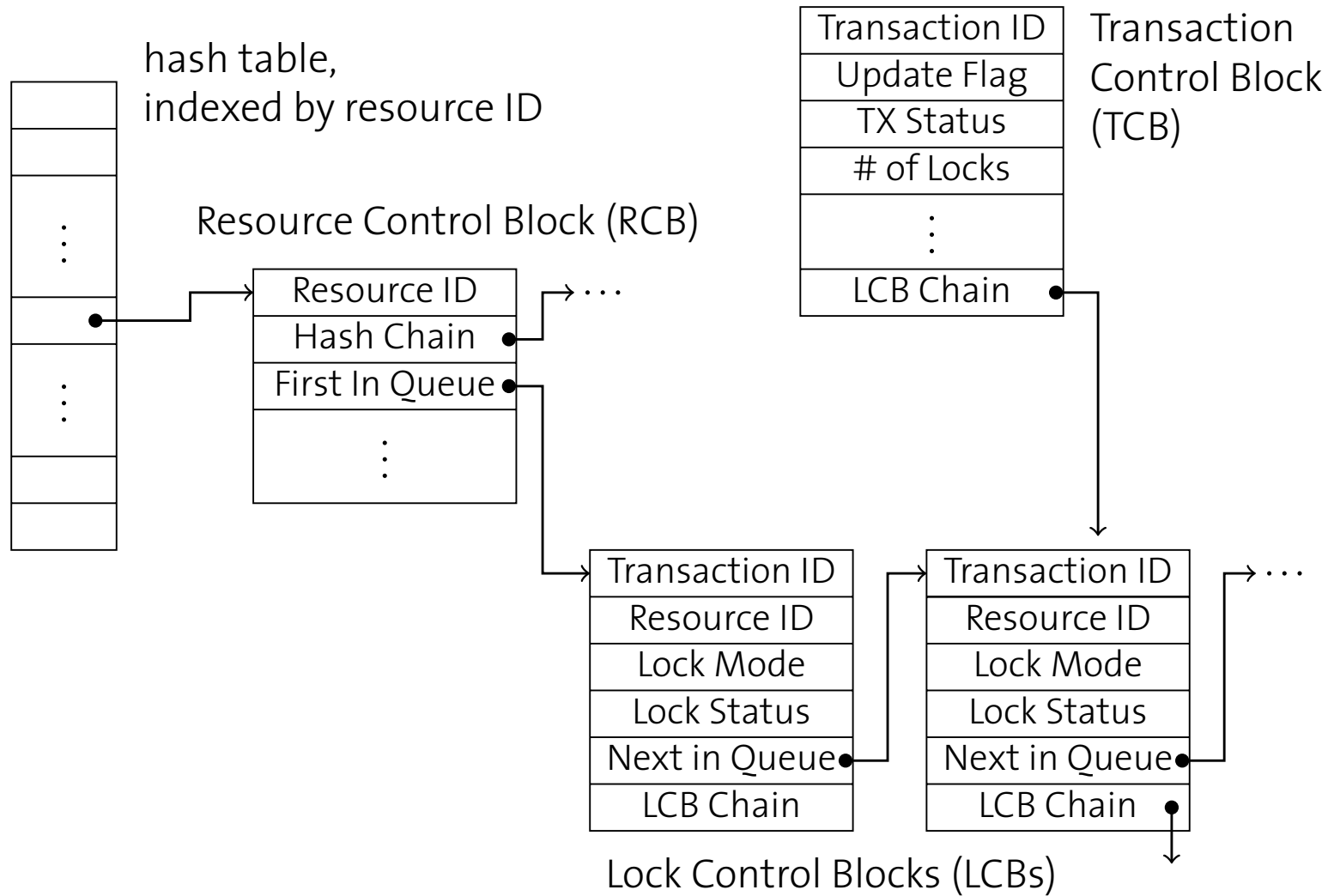
Während der normalen Transaktionsverarbeitung, werden zwei Dinge im Transaktionskontrollblock gespeichert

- LastLSN (Last Log Sequence Number)
  - LSN des letzten geschriebenen Log-Eintrags für die Transaktion
- UNxt (LSN Next to be Undone)
  - LSN des nächsten Eintrags, der beim Rückrollen betrachtet werden muss

Wenn eine Aktualisierung einer Seite  $p$  durchgeführt wird

- wird ein Eintrag  $r$  ins WAL geschrieben und
- die LSN von  $r$  im Seitenkopf von  $p$  gespeichert

# Datenstruktur zur Buchführung



# Rückrollen einer Transaktion

---

Schritte zum Rückrollen einer Transaktion T:

- Abarbeiten des WAL in Rückwärtsrichtung
- Beginn bei Eintrag, auf den UNxt im Transaktionskontrollblock von T zeigt
- Finden der übrigen Einträge von T durch Verfolgen der Prev- und UNxt-Einträge im Log

Undo-Operationen modifizieren ebenfalls Seiten

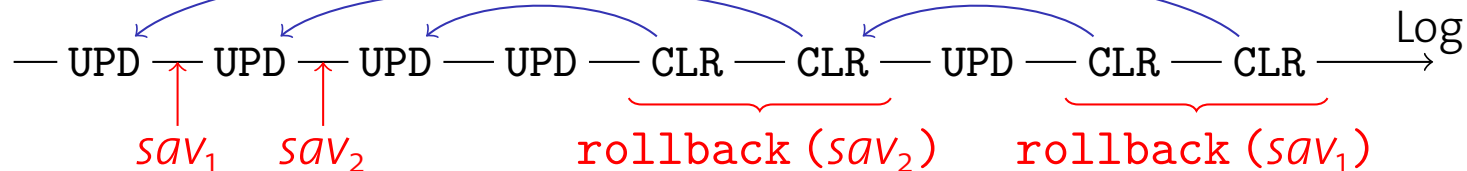
- Logging der Undo-Operationen im WAL
- Verwendung von Compensation-Log-Record (CLRs) für diesen Zweck

# Rückrollen einer Transaktion

```
1 Function: rollback (SaveLSN, T)
2 UndoNxt ← T.UNxt;
3 while SaveLSN < UndoNxt do
4   LogRec ← read log entry with LSN UndoNxt;
5   switch LogRec.Type do
6     case UPD
7       perform undo operation LogRec.Undo on page LogRec.Page;
8       LSN ← write log entry
9         ⟨CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, ..., ∅⟩;
10      set LSN = LSN in page header of LogRec.Page;
11      T.LastLSN ← LSN;
12     case CLR
13       UndoNxt ← LogRec.UNxt;
14   T.UNxt ← UndoNxt;
```

# Rückrollen einer Transaktion

- Transaktionen können auch partiell zurückgerollt werden (zur SaveLSN)
  - Wozu könnte das nützlich sein?
- Das UNxt-Feld in einem CLR zeigt auf den Logeintrag vor demjenigen, der ungeschehen gemacht wurde





# Wiederherstellung nach Ausfall

---

Neustart nach einem Systemabsturz in drei Phasen

## 1. Analyse-Phase:

- Lese Log in Vorwärtsrichtung
- Bestimmt Transaktionen, die aktiv waren als der Absturz passierte (solche Transaktionen nennen wir Pechvögel)

## 2. Redo-Phase:

- Spiele Operation im Log erneut ab (in Vorwärtsrichtung), um das System in den Zustand vor dem Fehler zu bringen

## 3. Undo-Phase:

- Rolle Pechvögel-Transaktionen zurück, in dem das Log in Rückwärtsrichtung abgearbeitet wird (wie beim normalen Zurückrollen)



# Analyse-Phase

---

```
1 Function: analyze ()
2 foreach log entry record LogRec do
3   switch LocRec.Type do
4     create transaction control block for LogRec.TX if necessary ;
5     case UPD or CLR
6       LogRec.TX.LastLSN ← LogRec.LSN ;
7       if LocRec.Type = UPD then
8         | LogRec.TX.UNxt ← LogRec.LSN ;
9       else
10        | LogRec.TX.UNxt ← LogRec.UNxt ;
11     case EOT
12     | delete transaction control block for LogRec.TX ;
```

# Redo-Phase

---

```
1 Function: redo ()
2 foreach log entry record LogRec do
3     switch LogRec.Type do
4         case UPD or CLR
5              $v \leftarrow \text{pin}(\text{LogRec.Page}) ;$ 
6             if  $v.\text{LSN} < \text{LogRec.LSN}$  then
7                 perform redo operation LogRec.Redo on  $v ;$ 
8                  $v.\text{LSN} \leftarrow \text{LogRec.LSN} ;$ 
9             unpin( $v, \dots$ ) ;
```

Auch beim Wiederherstellen können Abstürze eintreten

- Undo und Redo einer Transaktion müssen idempotent sein
  - $\text{undo}(\text{undo}(T)) = \text{undo}(T)$
  - $\text{redo}(\text{redo}(T)) = \text{redo}(T)$
- Prüfe LSN vor der Redo-Operation

# Redo-Phase

---

- Beachte, dass alle Operationen (auch solche von Pechvögeln) in chronologischer Ordnung erneut durchgeführt werden
- Nach der Redo-Phase ist das System im gleichen Zustand, wie zum Fehlerzeitpunkt
  - Einige Log-Einträge sind noch nicht auf der Platte, obwohl erfolgreich beendete Transaktionen ihre Änderung geschrieben hätten. Alle anderen müssten ungeschehen gemacht werden
- Wir müssen hinterher alle Effekte von Pechvögeln ungeschehen machen
- Als Optimierung kann man den Puffermanager instruieren, geänderte Seiten vorab zu holen (Prefetch)

# Undo-Phase

---

- Die Undo-Phase ist ähnlich zum Rückrollen im normalen Betrieb
- Es werden mehrere auf einmal Transaktionen zurückgerollt (all Pechvögel)
- Alle Pechvögel werden vollständig zurückgerollt (nicht nur bis zu einem Sicherungspunkt)

# Undo-Phase

```
1 Function: undo ()
2 while transactions (i.e., TCBs) left to roll back do
3    $T \leftarrow$  TCB of loser transaction with greatest UNxt ;
4    $LogRec \leftarrow$  read log entry with LSN  $T.UNxt$  ;
5   switch  $LogRec.Type$  do
6     case UPD
7       perform undo operation  $LogRec.Undo$  on page  $LogRec.Page$  ;
8        $LSN \leftarrow$  write log entry
9          $\langle CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, \dots, \emptyset \rangle$  ;
10      set  $LSN = LSN$  in page header of  $LogRec.Page$  ;
11       $T.LastLSN \leftarrow LSN$  ;
12     case CLR
13        $UndoNxt \leftarrow LogRec.UNxt$  ;
14      $T.UNxt \leftarrow UndoNxt$  ;
15     if  $T.UNxt = '-'$  then
16       write EOT log entry for  $T$  ;
17       delete TCB for  $T$  ;
```

# Checkpointing

---

- WAL ist eine immer-wachsendes Log-Datei, die bei der Wiederherstellung gelesen wird
- In der Praxis sollte die Datei nicht zu groß werden (Wiederherstellung dauert zu lange)
- Daher wird ab und zu ein sog. **Checkpoint** erstellt
  - **Schwergewichtiger Checkpoint:**  
Speicherung aller geänderter Seiten auf der Platte, dann Verwaltungsinformation für Checkpoint schreiben (Redo kann dann ab Checkpoint erfolgen)
  - **Leichtgewichtiger Checkpoint:**  
Speichere Information bzgl. geänderter Seiten in Log, aber keine Seiten (Redo ab Zeitpunkt kurz vor Checkpoint)

# Medien-Wiederherstellung

---

- Um Medienfehler zu kompensieren, muss periodisch eine Sicherungskopie erstellt werden (nicht-flüchtiger Speicher)
- Kann während des Betriebs erfolgen, wenn WAL auch gesichert wird
- Wenn Pufferverwalter verwendet wird, reicht es, das Log vom Zeitpunkt des Backups zu archivieren
  - Pufferverwalter hat aktuelle Seiten
  - Sonst muss bis zum ältesten Write der aktualisierten Seiten zurückgegangen werden
- Anderer Ansatz:  
Spiegeldatenbank auf anderem Rechner



# Leichtgewichtiger Checkpoint

---

Schreibe periodisch Checkpoint in drei Phasen

1. Schreibe Begin-Checkpoint-Logeintrag BCK
2. Sammle Information
  - über geänderte Seiten im Pufferverwalter und dem LSN ihrer letzten Modifikationsoperation und
  - über alle aktiven Transaktionen (und ihrer LastLSN und UNxt TCB-Einträge)

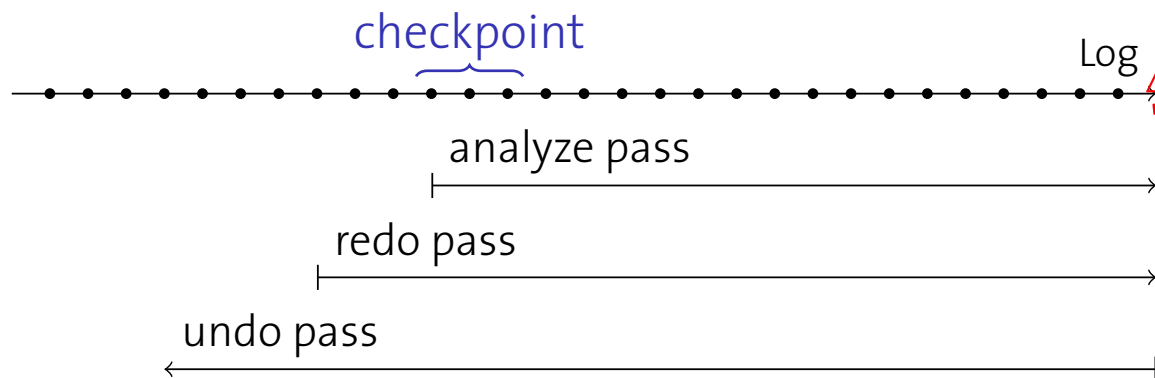
Schreibe diese Information in End-Checkpoint Eintrag ECK

3. Setze Master-Record auf bekannte Position auf der Platte und zeige auf LSN und BCK Logeinträge

# Wiederherstellung mit leichtgew. Checkpoint

## Während der Wiederherstellung

- Starte Analyse beim BCK-Eintrag im Master-Record (anstelle vom Anfang des Logs)
- Wenn der ECK-Eintrag gelesen wird
  - Bestimme kleinste LSN für die die Redo-Verarbeitung und
  - Erzeuge TCBs für alle Transaktionen im Checkpoint



# Zusammenfassung

---

- **ACID und Serialisierbarkeit**
  - Vermeiden von Anomalien durch Nebenläufigkeit
  - Serialisierbarkeit reicht für Isolation
- **Zwei-Phasen-Sperrprotokoll**
  - 2PL ist eine praktikable Technik, um Serialisierbarkeit zu garantieren (meist wird strikte 2PL verwendet)
  - In SQL-92 können sog. Isolationsgrade eingestellt werden (Abschwächung der ACID-Bedingungen)
- **Nebenläufigkeit in B-Bäumen**
  - Spezialisierte Protokolle (WTL) zur Flaschenhalsvermeidung
- **Wiederherstellung (ARIES)**
  - Write-Ahead-Log, Checkpoints

# Das Gesamtbild der Architektur

