Datenbanken

Transaktionsmanagement

Dr. Özgür Özçep

Prof. Dr. Ralf Möller

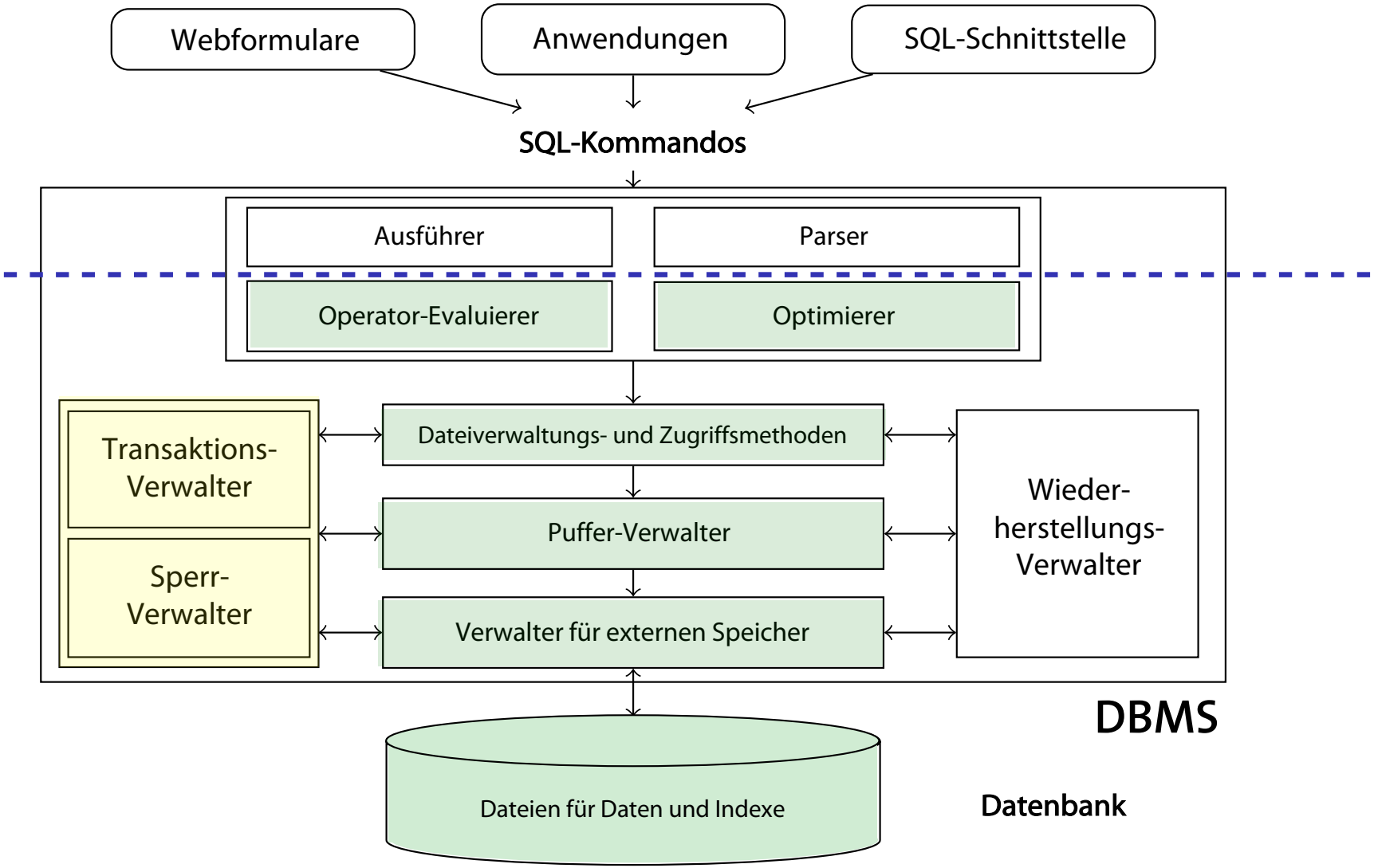
Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

und studentische Tutoren

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
		1200
bal ← `read` (*acct*) ;		1200
	bal ← `read` (*acct*) ;	1200
bal ← *bal* − 100 ;		1200
	bal ← *bal* − 200 ;	1200
		1100
`write` (*acct*, *bal*) ;		1100
	`write` (*acct*, *bal*) ;	1000

- Die erste Aktualisierung des Kontos ist verlorengegangen: 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) ;
```

- Ann.: 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

ACID-Eigenschaften und Transaktionen

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

- **A**tomicity: Entweder werden alle oder keine Werteänderungen einer Transaktion in den Datenbankzustand übernommen
- **C**onsistency: Eine Transaktion überführt einen konsistenten Zustand (FDs, Integritätsbedingungen) in einen anderen

ACID-Eigenschaften und Transaktionen

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

- **A**tomicity: Entweder werden alle oder keine Werteänderungen einer Transaktion in den Datenbankzustand übernommen
- **C**onsistency: Eine Transaktion überführt einen konsistenten Zustand (FDs, Integritätsbedingungen) in einen anderen
- **I**solation: Eine Transaktion berücksichtigt bei der Berechnung keine Effekte anderer parallel laufender Transaktionen

ACID-Eigenschaften und Transaktionen

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

- **A**tomicity: Entweder werden alle oder keine Werteänderungen einer Transaktion in den Datenbankzustand übernommen
- **C**onsistency: Eine Transaktion überführt einen konsistenten Zustand (FDs, Integritätsbedingungen) in einen anderen
- **I**solation: Eine Transaktion berücksichtigt bei der Berechnung keine Effekte anderer parallel laufender Transaktionen
- **D**urability: 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	Transaction 2

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

	`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

Aufgabe:

Geben Sie ein Beispiel für eine Integritätsbedingung (constraint) an, die beim "Inconsistent Read" des vorigen Beispiels verletzt wäre.

Aufgabe:

Geben Sie ein Beispiel für eine Integritätsbedingung (constraint) an, die beim "Inconsistent Read" des vorigen Beispiels verletzt wäre.

Lösung: Dispositionskredit-Constraint, etwa SUM(balance) >= -600 Euro

Anomalien: Dirty Read

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

me	my wife	DB state
$bal \leftarrow$ `read` $(acct)$;		1200
$bal \leftarrow bal - 100$;		1200
`write` $(acct, bal)$;		1100
	$bal \leftarrow$ `read` $(acct)$;	1100
	$bal \leftarrow bal - 200$;	1100
`abort` ;		1200
	`write` $(acct, bal)$;	900

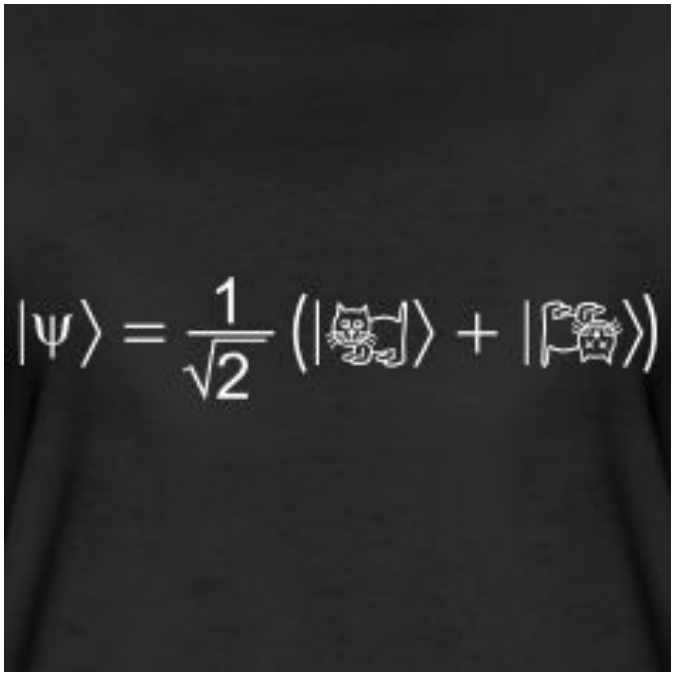
- Die Transaktion meiner Frau hat schon einen geänderten Zustand gelesen bevor meine Transaktion zurückgerollt wird

How to have the cake and eat it

- Anomalien ließen sich durch einfache sequentielle Ausführung der Transaktionen vermeiden
- Wir wollen aber beides: Keine Anomalien, dabei aber Nebenläufigkeit

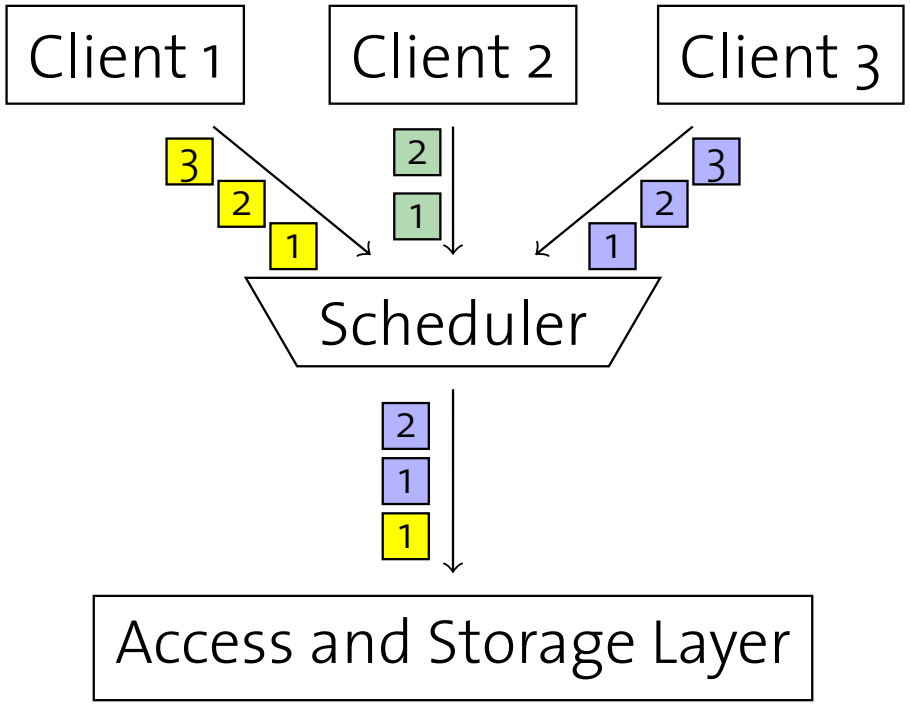
How to have the cake and eat it

- Anomalien ließen sich durch einfache sequentielle Ausführung der Transaktionen vermeiden
- Wir wollen aber beides: Keine Anomalien, dabei aber Nebenläufigkeit



Nebenläufige Ausführung

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



Datenbankobjekte und Zugriffe darauf

- 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 ≙ Komponente eines Tupels

Transaktion: Definition

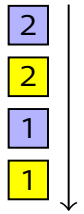
- Eine Datenbanktransaktion ist eine (strikt geordnete) Folge von Schritten, wobei ein Schritt eine Zugriffsoperation auf ein Objekt ist
  - Transaktion $T = \langle s_1, ..., s_n \rangle$
  - Schritt $s_i = a_i(e_i)$
  - Zugriffsoperation $a_i \in \{ r(ead), w(rite) \}$
- Die Länge einer Transaktion ist definiert als die Anzahl der Schritte $|T| = n$
- Beispiel: $T = \langle r(A), w(A), r(B), w(B) \rangle$
- Abarbeitung durch Mischung der Schritte mehrerer Transaktionen (Sequenzieller Plan, Sequenz, S)

Serielle Ausführung

Ein spezieller sequentieller 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:



- $S = <r_1(B), r_2(B), w_1(B), w_2(B)>$
- Dieser Plan ist nicht seriell

Wenn meine Frau später zum Automaten geht, ergibt sich

- $S = <r_1(B), w_1(B), r_2(B), w_2(B)>$
- 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

Abarbeitungsreihenfolge

- Vorstellung: Sequentieller Plan gegeben
- Manchmal kann man einfach Teilschritte aus verschiedenen Transaktionen in einem Plan umordnen
  - Nicht jedoch die Teilschritte innerhalb einer einzelnen Transaktion (sonst eventuell anderes Ergebnis)
- Jeder Plan S‘, der durch legale Umordnung von S generiert werden kann, heißt äquivalent zu S
- Falls Umordnung nicht möglich weil Ergebnis verfälscht → Konflikt
- Wie ist das definierbar?

Konflikte

Definition eines Konflikts:

- Zwei Operationen $a_i(e)$ und $a'_j(e')$ stehen in Konflikt zueinander in S, wenn
  - sie zu zwei verschiedenen Transaktionen gehören $i \neq j$,
  - 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`	×	×

Serialisierbarkeit

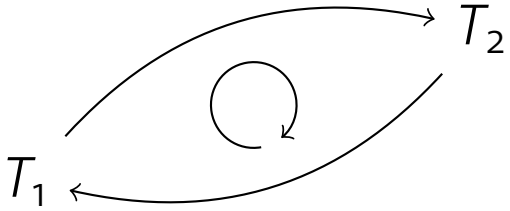
- Ein Plan S heißt serialisierbar gdw. er äquivalent ist zu einem seriellen Plan S‘
- Die Ausführung eines serialisierbaren Plans S ist korrekt (S braucht nicht seriell zu sein)
- Korrektheit eines Plans kann anhand des Konfliktgraphen G_S gezeigt werden (auch Serialisierungsgraph genannt)
  - Knoten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

Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

($w_1(A)$ kann nicht an $r_2(A)$ vorbeigeschoben werden, da Konflikt)



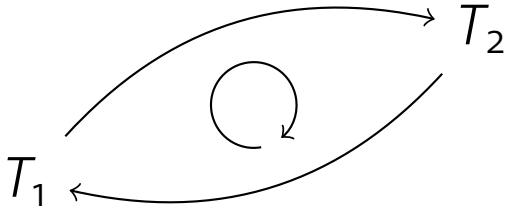
nicht serialisierbar

Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

($w_1(A)$ kann nicht an $r_2(A)$ vorbeigeschoben werden, da Konflikt)

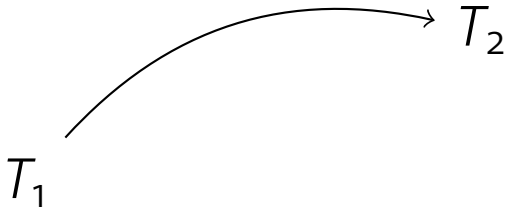


nicht serialisierbar

Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$

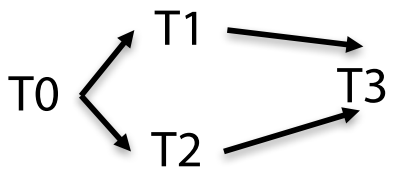
($r_1(S)$ und $w_1(S)$ können an $r_2(C)$ und $w_2(C)$ vorbeigeschoben werden, da kein Konflikt)



serialisierbar

Erinnerung Topologische Sortierung

Toplogische Sortierung eines Graphen G = Lineare Ordnung < der Knoten von G, so dass für alle Knoten v_i, v_j aus v_i -> v_j folgt $v_i < v_j$



Mögliche Topologische Sortierungen

1. T0 < T1 < T2 < T3
2. T0 < T2 < T1 < T3

Aufgabe:
Ist dieser Plan serialisierbar?

Schritt	T1	T2
1	BOT	
2	r(A,a1)	
3	a1 = a1 -50	
4	w(A,a1)	
5		BOT
6		r(A,a2)
7		a2 = a2 -100
8		w(A,a2)
9		r(B,b2)
10		b2 = b2 + 100
11		w(B,b2)
12		commit
13	r(B,b1)	
14	B1 = b1 + 50	
15	w(B,b1)	
16	commit	

Aufgabe:

Ist dieser Plan serialisierbar?

Schritt	T1	T2
1	BOT	
2	r(A,a1)	
3	a1 = a1 -50	
4	w(A,a1)	
5		BOT
6		r(A,a2)
7		a2 = a2 -100
8		w(A,a2)
9		r(B,b2)
10		b2 = b2 + 100
11		w(B,b2)
12		commit
13	r(B,b1)	
14	b1 = b1 + 50	
15	w(B,b1)	
16	commit	

Lösung:

Obwohl der Serialisierbarkeitsgraph zyklisch ist, ist die Verzahnung von T1 und T2 unproblematisch (wegen der Kommutativität der Updateoperation). Aber der Planer sieht nur die Ebene der read-write-Operationen.

Aufgabe:

Ist dieser Plan serialisierbar?

Lösung (Forts.):

Der Plan rechts ist auf der read-write Ebene genau so aufgebaut wie der erste Plan, allerdings ist dieser nicht serialisierbar.

Schritt	T1	T2
1	BOT	
2	r(A,a1)	
3	a1 = a1 -50	
4	w(A,a1)	
5		BOT
6		r(A,a2)
7		a2 = a2 * 1.03
8		w(A,a2)
9		r(B,b2)
10		b2 = b2 * 1.03
11		w(B,b2)
12		commit
13	r(B,b1)	
14	b1 = b1 + 50	
15	w(B,b1)	
16	commit	

Transaktionszustände

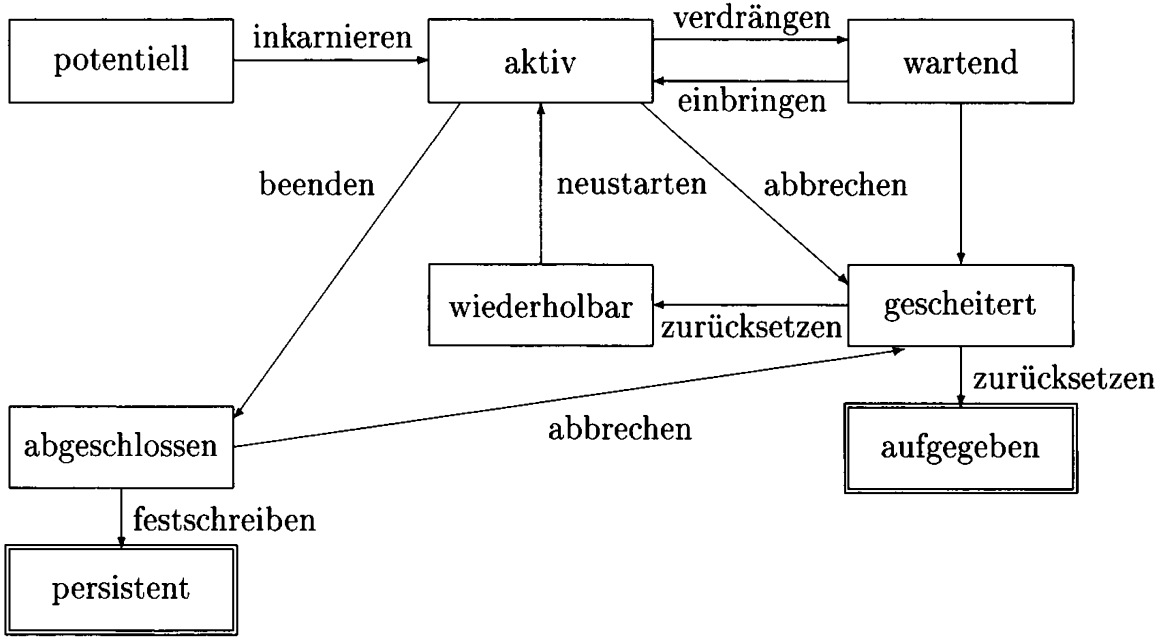


Abb. 9.2: Zustandsübergangs-Diagramm für Transaktionen

Kemper/Eickler: Datenbanksysteme, Oldenburg, 7. Auflage, 2009

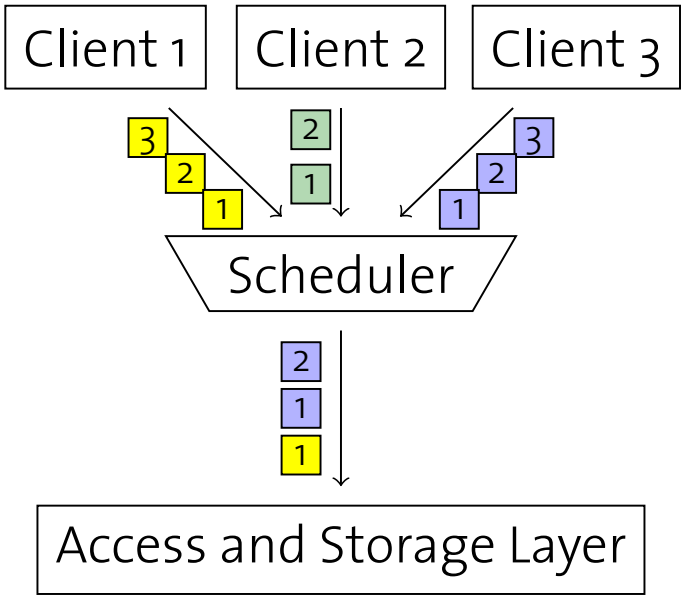
Sperren im Anfrageplan

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 soll ein unkontrollierter nebenläufiger Zugriff auf o verhindert werden

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

Verwendung von Sperren vor dem Zugriff

```
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
```

- Sperren werden automatisch in den Anfragebeantwortungsplan eingefügt

Serialisierbarkeit durch Sperren

Transaction 1	Transaction 2	DB state
`lock (`*acct*`) ;`		1200
`read (`*acct*`) ;`		
	`lock (`*acct*`) ;`	
`write (`*acct*`) ;`	Transaction	1100
`unlock (`*acct*`) ;`	blocked	
	`read (`*acct*`) ;`	
	`write (`*acct*`) ;`	900
	`unlock (`*acct*`) ;`	

- Kein Lost Update

Aufgabe:

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

Aufgabe:

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

Lösung: Nein!

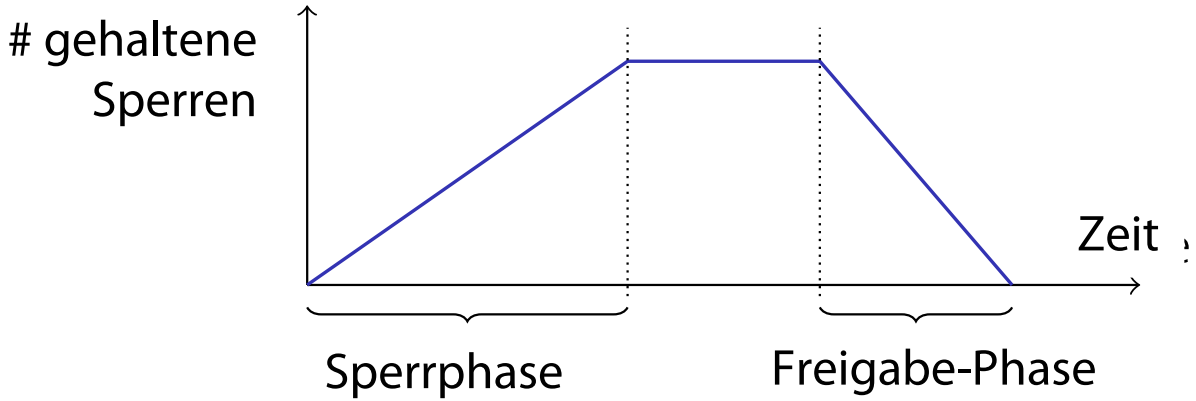
Vgl. Beispiel ATM-Zugriff ich und meine Frau (lost Update).

me	my wife	DB state
		1200
bal ← read (*acct*) ;		1200
	bal ← read (*acct*) ;	1200
bal ← *bal* − 100 ;		1200
	bal ← *bal* − 200 ;	1200
write (*acct*, *bal*) ;		1100
	write (*acct*, *bal*) ;	1000

Transaction 1	Transaction 2	DB state
		1200
lock (*acct*) ;		
read (*acct*) ;		
unlock (*acct*) ;		
	lock (*acct*) ;	
	read (*acct*) ;	
	unlock (*acct*) ;	
lock (*acct*) ;		
write (*acct*) ;		1100
unlock (*acct*) ;		
	lock (*acct*) ;	
	write (*acct*) ;	1000
	unlock (*acct*) ;	

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



- Repeatable Read und kein Inconsistent Read; Serialisierbarkeit gewährleistet in fehlerfreier Umgebung

Anmerkung (Non-repeatable read ähnlich zu Phantom Read:
Gleiche Anfrage innerhalb einer Transaktion liefert unterschiedliche Ergebnisse.
Beim Phantomproblem bedingt durch Einfügen neuer Zeilen, bei Non-repeatable
Bedingt durch insert oder update.

ATM-Beispiel mit 2PL verletzender Ausführung

Transaction 1	Transaction 2	DB state
`lock(`*acct*`);`		1200
`read(`*acct*`);`		
`unlock(`*acct*`);`		
	`lock(`*acct*`);`	
	`read(`*acct*`);`	
	`unlock(`*acct*`);`	
`lock(`*acct*`);`		
`write(`*acct*`);`		1100
`unlock(`*acct*`);`		
	`lock(`*acct*`);`	
	`write(`*acct*`);`	1000
	`unlock(`*acct*`);`	

ATM-Beispiel mit 2PL einhaltender Ausführung

Transaction 1	Transaction 2	DB state
`lock(`*acct*`);`		1200
`read(`*acct*`);`		
	`lock(`*acct*`);`	
`write(`*acct*`);`	Transaction	1100
`unlock(`*acct*`);`	blocked	
	`read(`*acct*`);`	
	`write(`*acct*`);`	900
	`unlock(`*acct*`);`	

Sperrarten (Sperrmodi)

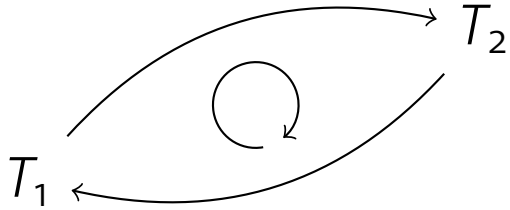
- Wir haben gesehen, dass zwei Leseoperationen nicht in Konflikt zueinander stehen
- Systeme verwenden verschiedene Arten von Sperren
  - Lesesperren (read locks, shared locks): Modus S
  - Schreibsperren (write locks, exclusive locks): Modus X
- Locks stehen nur in Konflikt zueinander, wenn eines davon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

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

Serialisierungsgraph

Beispiel: ATM-Transaktion

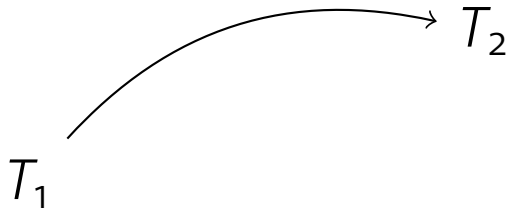
- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$



nicht serialisierbar

Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$



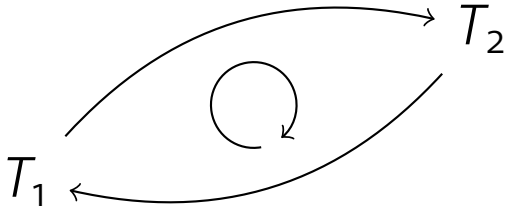
serialisierbar

Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

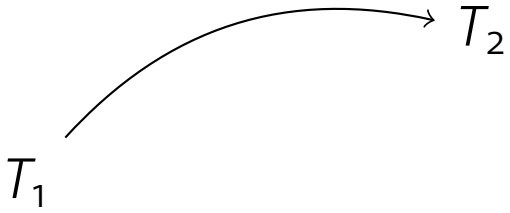
XLock



nicht serialisierbar

Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$

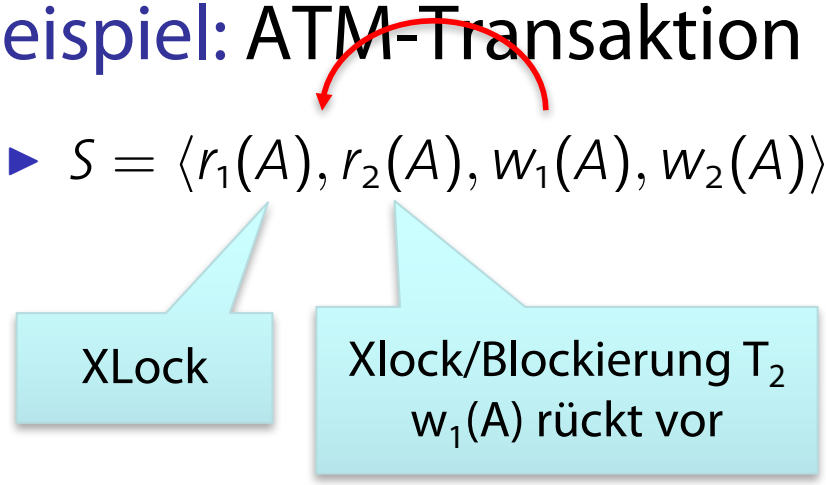


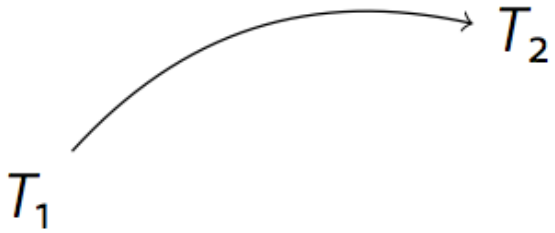
serialisierbar

Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

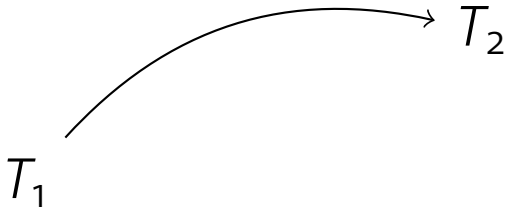




serialisierbar

Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$



serialisierbar

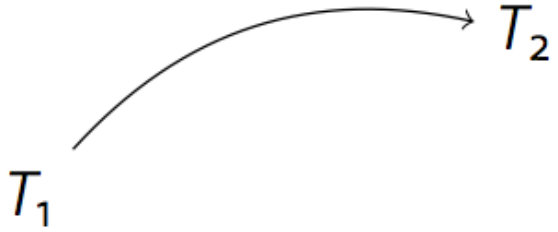
Beispiele noch einmal: Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

XLock

Xlock/Blockierung T_2
$w_1(A)$ rückt vor

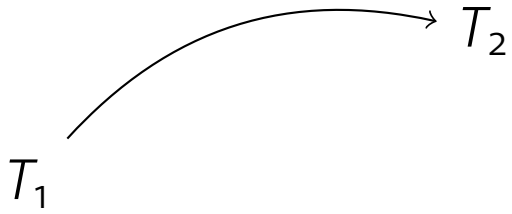


serialisierbar / korrekt

Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$

XLock

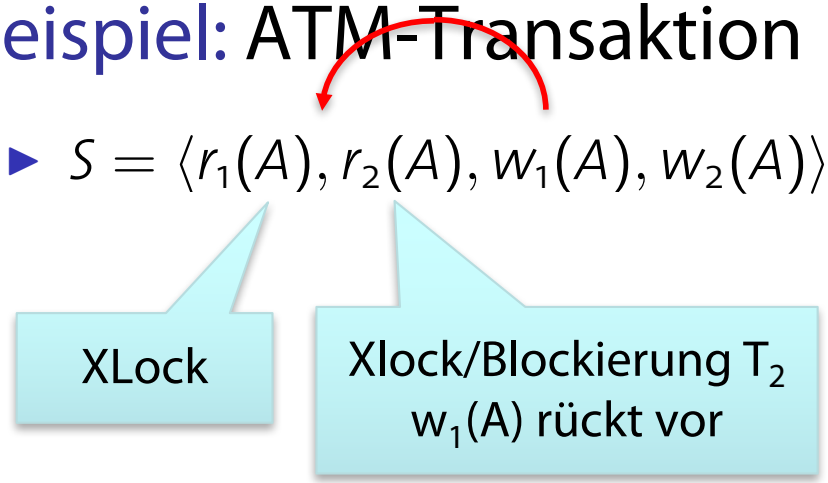


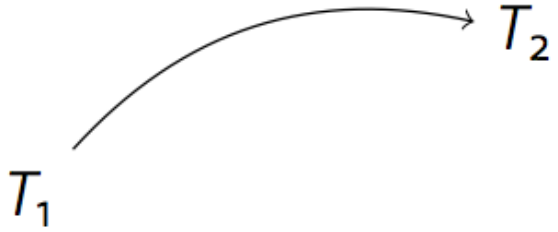
serialisierbar

Beispiele noch einmal: Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

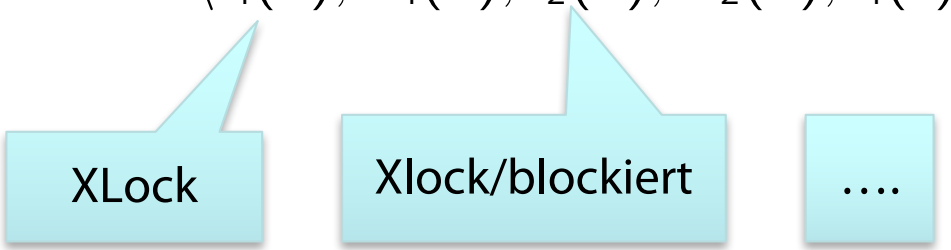


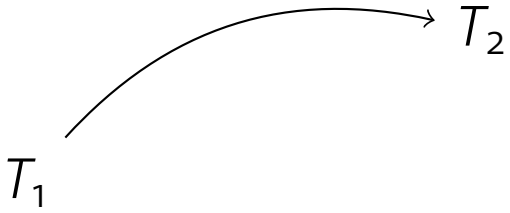


serialisierbar / korrekt

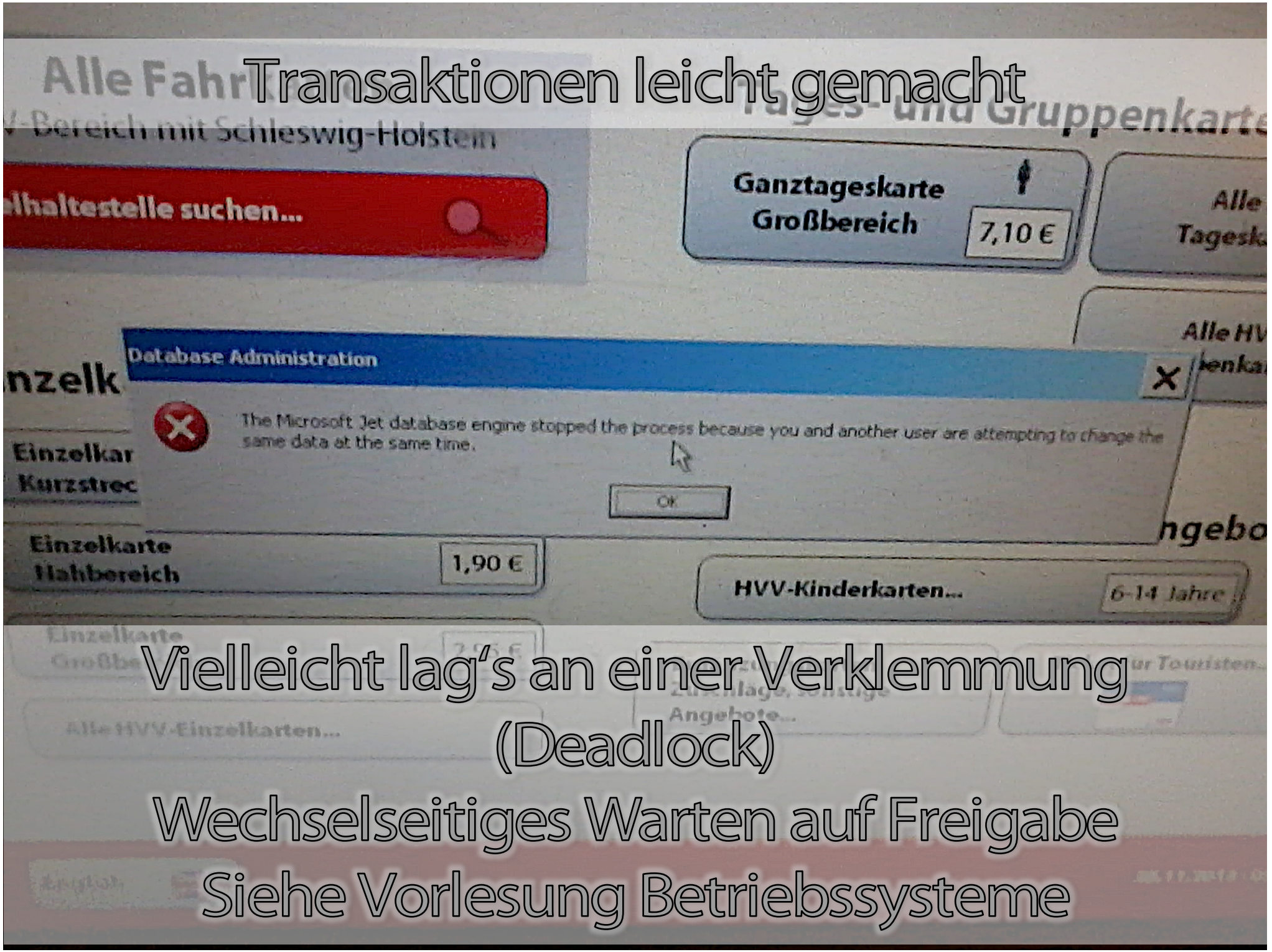
Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$



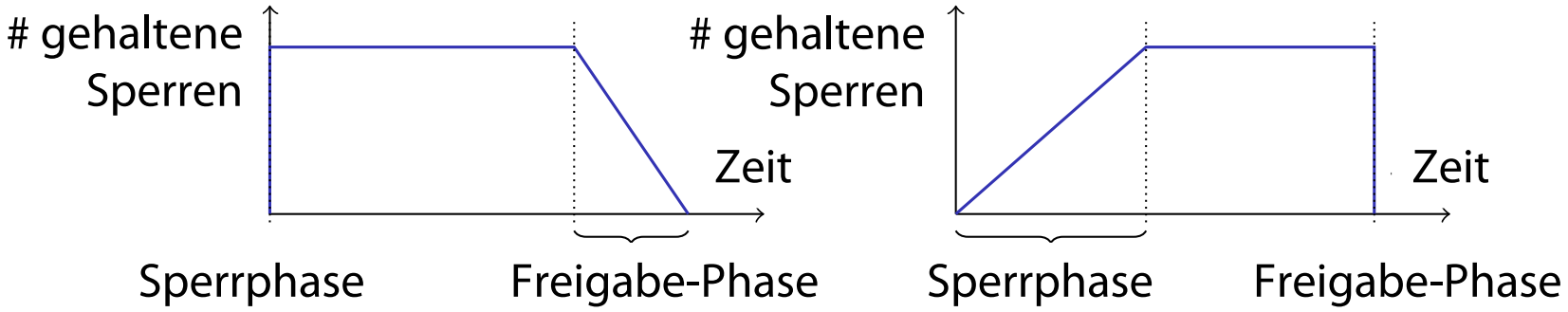


serialisierbar


Transaktionen leicht gemacht
Database Administration
The Microsoft Jet database engine stopped the process because you and another user are attempting to change the same data at the same time.
OK
Ganztageskarte Großbereich
7,10 €
Einzelkarte Nahbereich
1,90 €
HVV-Kinderkarten...
6-14 Jahre
Vielleicht lag's an einer Verklemmung
(Deadlock)
Wechselseitiges Warten auf Freigabe
Siehe Vorlesung Betriebssysteme

Varianten des Zwei-Phasen-Sperrprotokolls

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



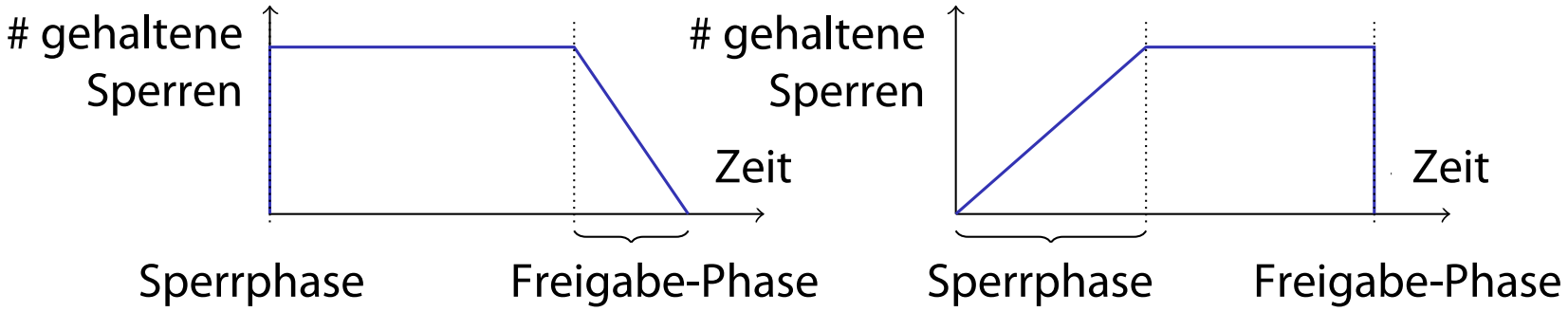
2PL mit Voranforderung

striktes 2PL

- Wodurch könnten die Varianten motiviert sein?

Varianten des Zwei-Phasen-Sperrprotokolls

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



2PL mit Voranforderung

striktes 2PL

- Wodurch könnten die Varianten motiviert sein?

2PL mit Voranforderung: Vermeide Deadlocks;

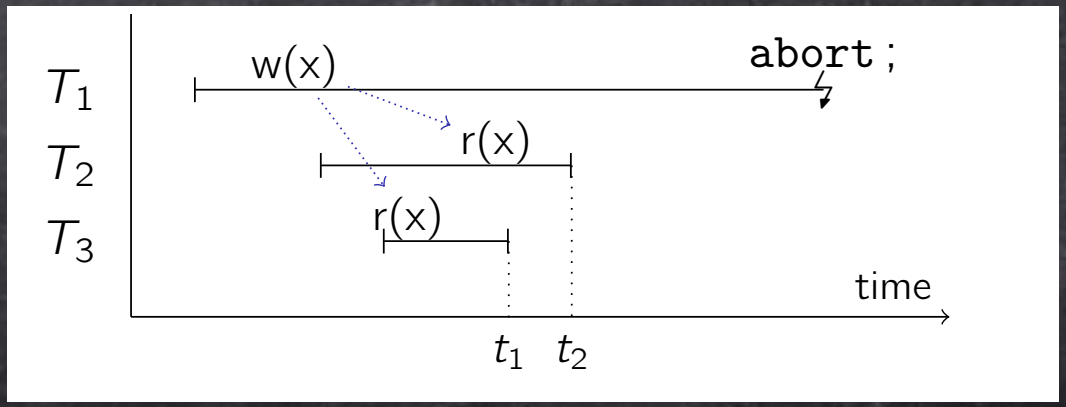
Aufgabe:

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

Aufgabe:

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

Lösung: Dirty Read droht. Ließe sich noch durch Abhängigkeitsanalyse vermeiden. Aber es ist auch kaskadierendes Rollback nötig. Letzteres durch striktes 2PL verhinderbar.



Datenbanken

Transaktionsmanagement

Dr. Özgür Özçep

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

und studentische Tutoren

ACID-Eigenschaften und Transaktionen

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

- **A**tomicity: Entweder werden alle oder keine Werteänderungen einer Transaktion in den Datenbankzustand übernommen
- **C**onsistency: Eine Transaktion überführt einen konsistenten Zustand (FDs, Integritätsbedingungen) in einen anderen
- **I**solation: Eine Transaktion berücksichtigt bei der Berechnung keine Effekte anderer parallel laufender Transaktionen
- **D**urability: Effekte einer erfolgreichen Transaktion werden persistent gemacht

Phantom-Problem

Transaction 1	Transaction 2	Effect
scan relation R ;		T_1 locks all rows
	insert new row into R ;	T_2 locks new row
	`commit` ;	T_2's lock released
scan relation R ;		reads **new** 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
- Sollen wir immer die ganze Relation sperren?

Phantom-Problem (Forts.)

Transaction 1	Transaction 2	Result

`SELECT COUNT(*)`
`FROM Customers`
`WHERE Name = 'Sam'`
		2

	`INSERT INTO Customers` `VALUES (..., 'Sam', ...)`	ok

`SELECT COUNT(*)`
`FROM Customers`
`WHERE Name = 'Sam'`
		3 ↯

- Ungern ganze Relation sperren (s. auch Granularität von Sperren)
- Sperren von Prädikaten?
  - Ungünstig, da man diese vernünftig repräsentieren und vergleichen muss (z.B. Testen auf Äquivalenz)
- Lösung: Nutze Index
  - Hier: Sperre alle (vorhanden oder noch einzufügenden) Tupel mit dem Indexschlüssel „Sam“

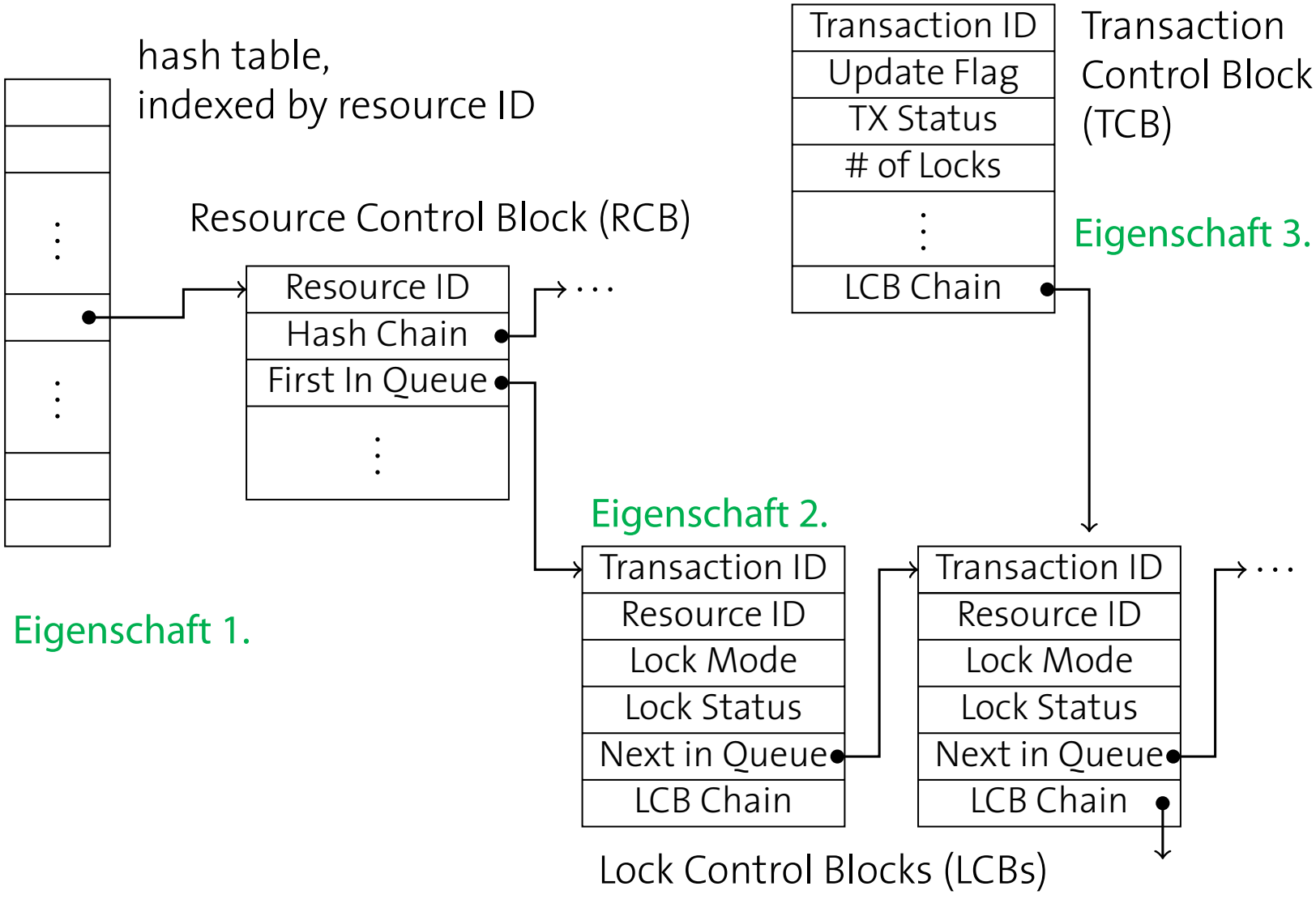
Implementierung eines Sperrverwalters

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

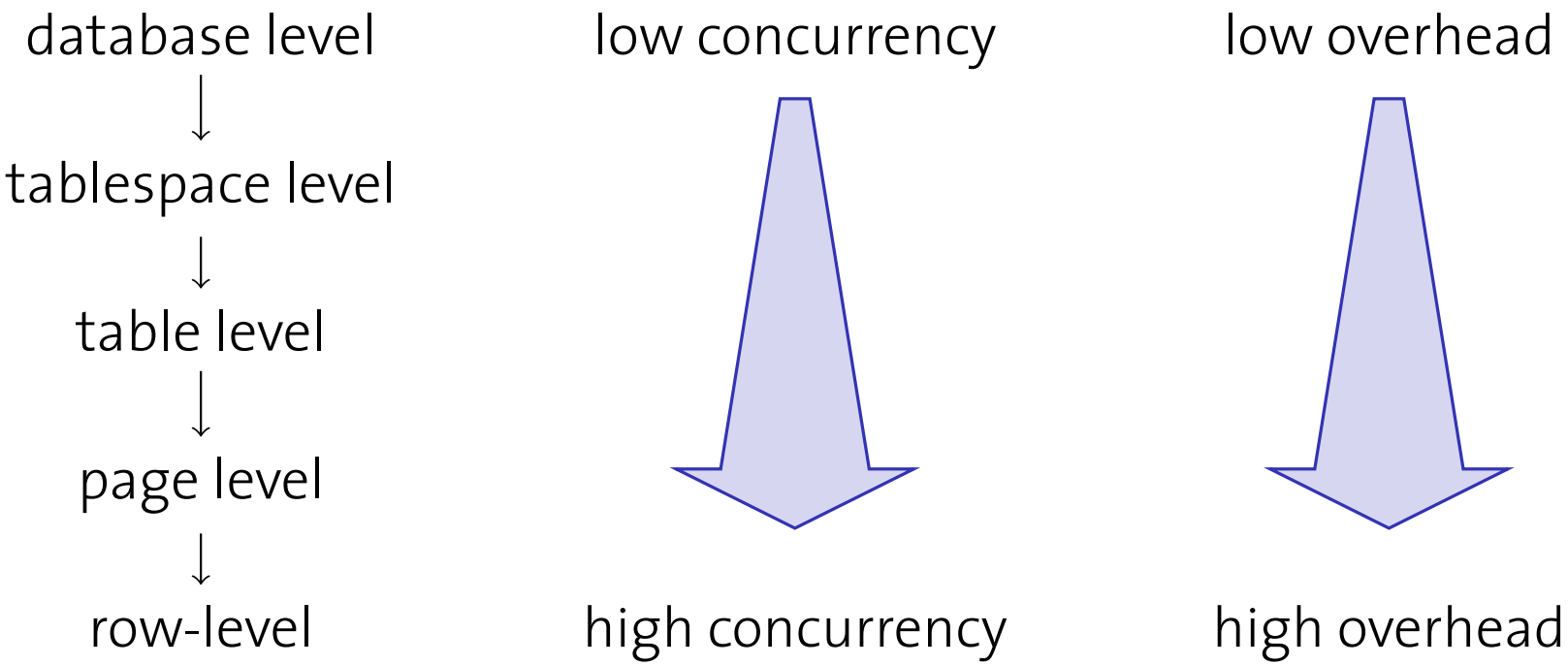


Implementierung von Aufgaben

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
- Wofür sollte man Sperren auf Seitenebene betrachten?

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_2 sollen nicht für alle Tupel umständlich Sperrkonflikte analysiert werden

Vorhabens-Sperren

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

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

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

- Eine Sperre I□ auf einer gröberen Ebene bedeutet, dass es eine Sperre □ auf einer niederen Ebene gibt

Vorhabens-Sperren

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

Protokoll für Sperren auf mehreren Ebenen:

1. Eine Transaktion kann jede Ebene g in Modus □ ∈ {S, X} sperren
2. Bevor Ebene g in Modus □ gesperrt werden kann, muss eine Sperre I□ für alle gröberen Ebenen gewonnen werden

Vorhabens-Sperren

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

Protokoll für Sperren auf mehreren Ebenen:

1. Eine Transaktion kann jede Ebene g in Modus □ ∈ {S, X} sperren
2. Bevor Ebene g in Modus □ gesperrt werden kann, muss eine Sperre I□ für alle gröberen Ebenen gewonnen werden

Anfrage Q_1 (Finde Tupel in CUSTOMERS mit Key=42) würde

- eine IS-Sperre für Tabelle CUSTOMERS anfordern (auch für Tablespace und die Datenbank) und dann

Vorhabens-Sperren

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

Protokoll für Sperren auf mehreren Ebenen:

1. Eine Transaktion kann jede Ebene g in Modus □ ∈ {S, X} sperren
2. Bevor Ebene g in Modus □ gesperrt werden kann, muss eine Sperre I□ für alle gröberen Ebenen gewonnen werden

Anfrage Q_1 (Finde Tupel in CUSTOMERS mit Key=42) 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

Vorhabens-Sperren

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

Protokoll für Sperren auf mehreren Ebenen:

1. Eine Transaktion kann jede Ebene g in Modus □ ∈ {S, X} sperren
2. Bevor Ebene g in Modus □ gesperrt werden kann, muss eine Sperre I□ für alle gröberen Ebenen gewonnen werden

Anfrage Q_1 (Finde Tupel in CUSTOMERS mit Key=42) 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 (Kopiere CUSTOMERS) wurde 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 auch noch zu bearbeiten

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

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

Hierfür wird

- eine IX-Sperre auf Tabelle CUSTOMERS (und ...) sowie
- eine X-Sperre auf dem Tupel mit Key=17 angefordert

Entdeckung von Konflikten

Nehmen wir an, folgende Anfrage ist auch noch zu bearbeiten

Q3:

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

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

Hierfür wird

- eine IX-Sperre auf Tabelle CUSTOMERS (und ...) sowie
- eine X-Sperre auf dem Tupel mit Key=17 angefordert

Diese Anfrage ist

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

Konsistenzgarantien in SQL-92

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

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

```
SET ISOLATION SERIALIZABLE;
```

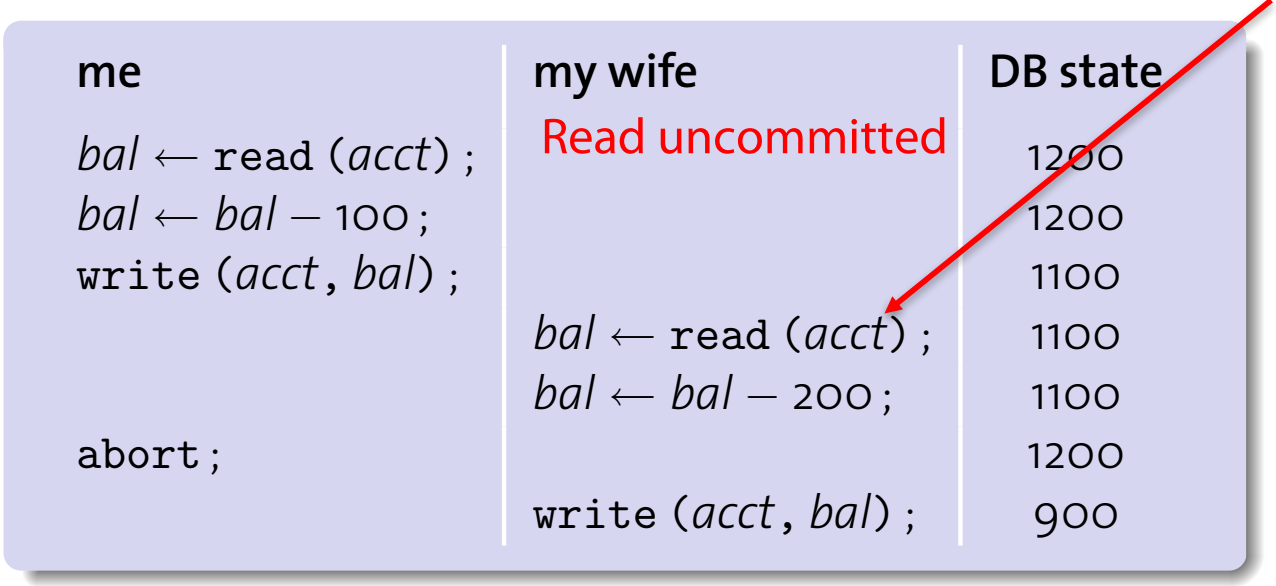
- Es gibt weniger strikte Modi, unter denen die Performanz höher ist (weniger Verwaltungsaufwand z.B. für Sperren)

SQL-92 Isolations-Modi

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

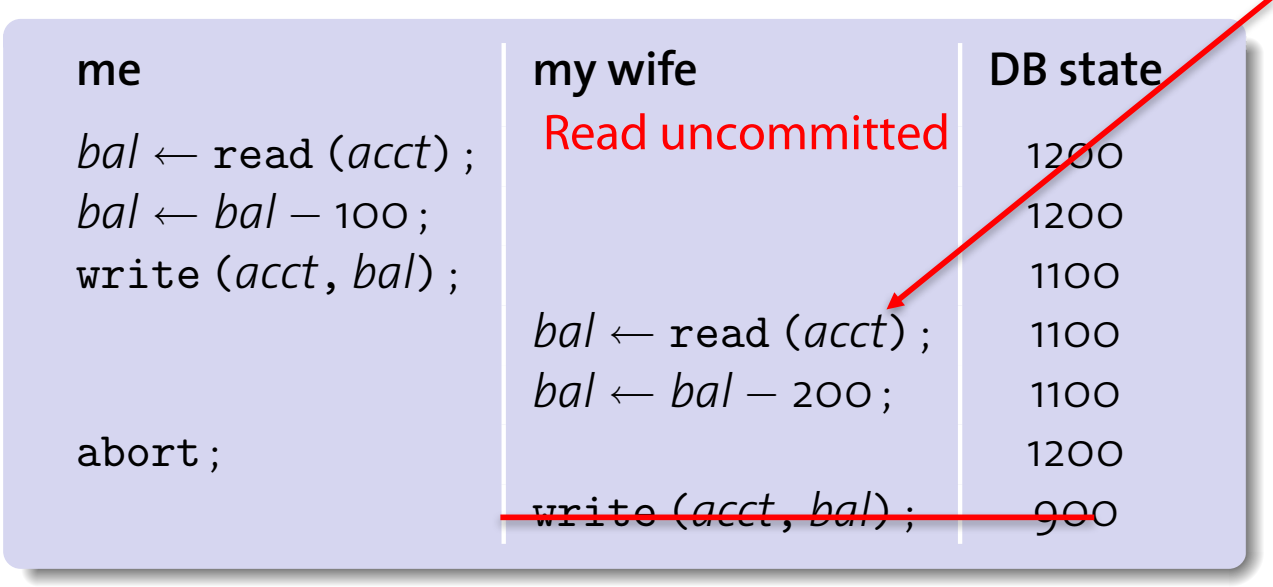
SQL-92 Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)



SQL-92 Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)



Es muss kein read lock erworben werden

Read uncommited nur für lesende Transaktion

SQL-92 Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)
- Read committed (auch ‘cursor stability‘)
  - Lesesperren werden nur gehalten, sofern der Zeiger auf das betreffende Tupel zeigt, Schreibsperren nach 2PL

SQL-92 Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)
- Read committed (auch ‘cursor stability‘)
  - Lesesperren werden nur gehalten, sofern der Zeiger auf das betreffende Tupel zeigt, Schreibsperren nach 2PL

T1(read committed)	T2
Read(A)	
	Write(A)
	Write(B)
	commit
Read(B)	
Read(A)	

Nur commited gelesen

Aber:
Unrepeatable Read
Nicht ausgeschlossen

SQL-92 Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)
- Read committed (auch ‘cursor stability‘)
  - Lesesperren werden nur gehalten, sofern der Zeiger auf das betreffende Tupel zeigt, Schreibsperren nach 2PL
- Repeatable read (auch ‘read stability‘)
  - Lese- und Schreibsperren nach 2PL akquiriert

SQL-92 Isolations-Modi

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

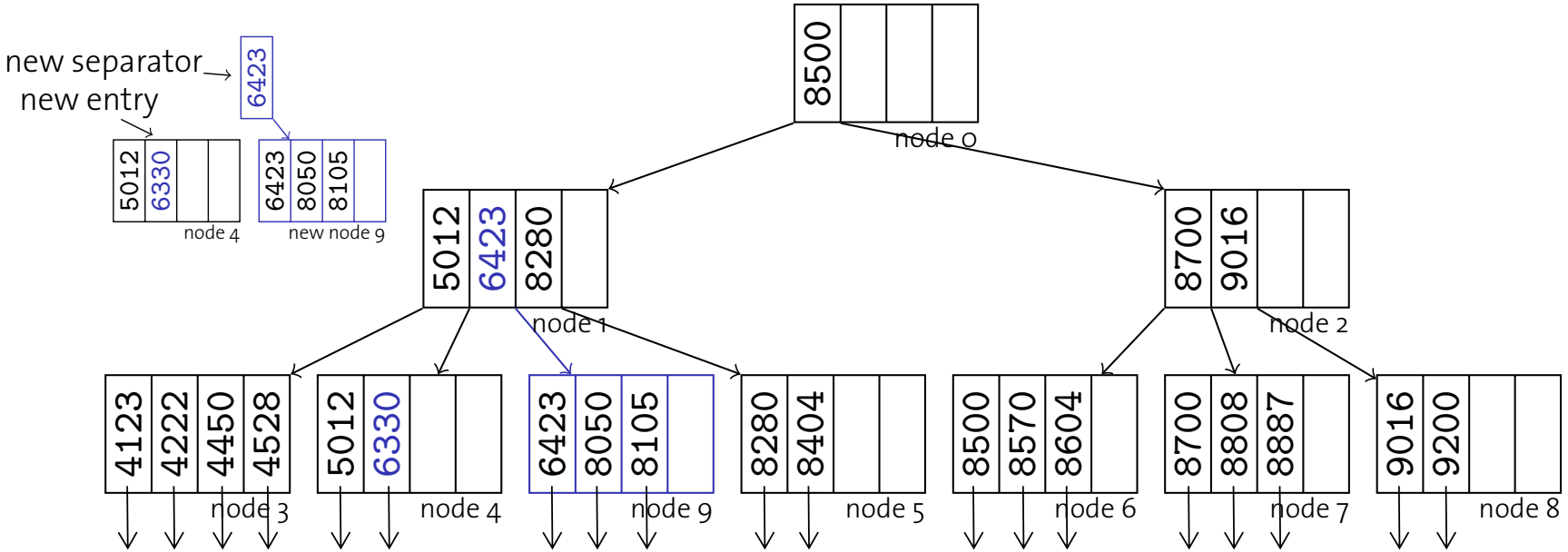
Resultierende Konsistenzgarantien

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 Isolationsmodi (isolation levels)
- Nur wenige Anwendungen benötigen (volle) Serialisierbarkeit

Nebenläufigkeit beim Indexzugriff

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



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

Nebenläufigkeit beim Indexzugriff

- Angenommen, die Aufspaltung ist gerade erfolgt, aber der neue Separator 6423 ist noch nicht etabliert
- Nimm weiterhin an, ein nebenläufiges Lesen in Transaktion T_r sucht nach 8050
  - Die Verzeigerung weist auf Knoten $node_4$
  - Knoten $node_4$ 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
- Während des Indexzugriffs einer Transaktion müssen alle anderen Transaktionen 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

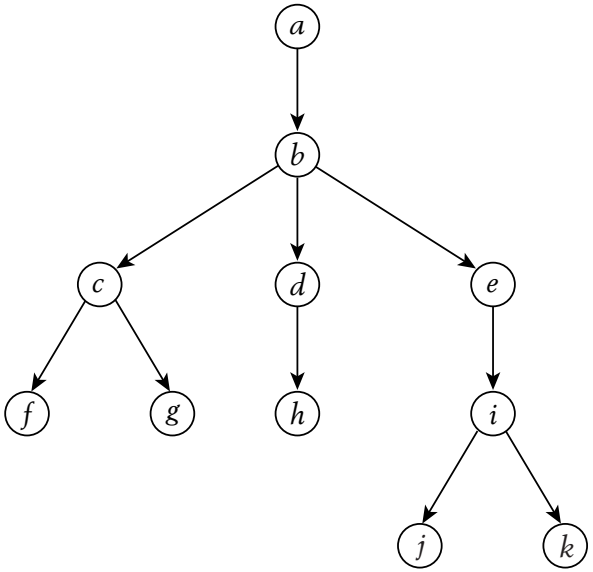
Betrachten wir den Schreibe-Fall (alle Sperren mit Konflikt)

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

Und damit gilt:

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

Beispiel WTL



Transaktion w(d), w(i), w(k)

Plan gemäß WTL:

wl(a),wl(b),wu(a),wl(d),wl(e),wu(b),w(d),wu(d),
wl(i),wu(e),w(i),wl(k),wu(i),w(k),wu(k)

Zugriffssordnung für Datenelemente
(z.B. B-Baum)

Weikum/Vossen: Transactional Information Systems, Elsevier, 2002

Aufspaltungssicherheit

- Wir müssen auf dem Weg nach unten in den B-Baum Schreibsperren wegen möglicher Aufspaltungen halten
- Allerdings kann man leicht prüfen, ob eine 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)

readers

```
1 place S lock on root ;
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 ;
```

writers

```
1 place X lock on root ;
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

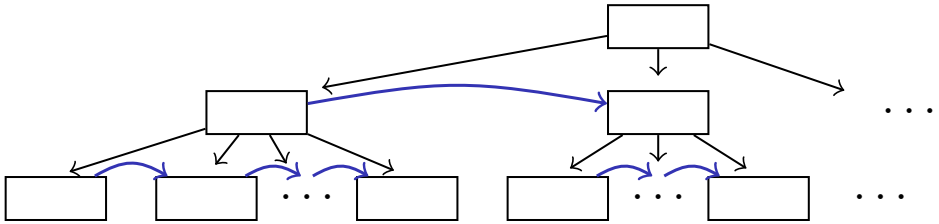
```
1  place S lock on root ;
2  current ← root ;
3  while current is not a leaf node do
4      son ← appropriate son of current ;
5      if son is a leaf then
6          place X lock on son ;
7      else
8          place S lock on son ;
9      release lock on current ;
10     current ← son ;
11 if current 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 Nachteil von Variante 2 ist, dass 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

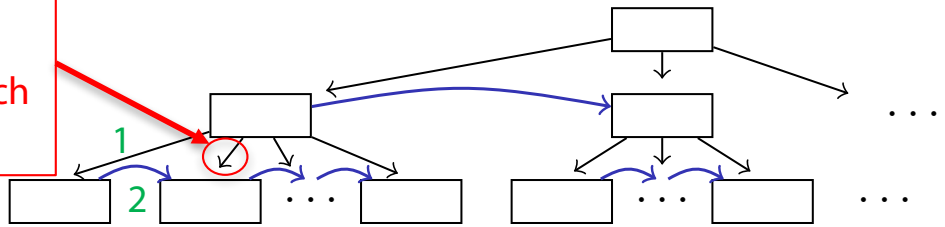


- Vorher schon betrachtet: Verkettete Liste auf Blattebene
- Zeiger stellen zweiten Pfad auf Knoten bereit (Diese werden schon beim Splitten erzeugt)
- Bei nebenläufigen Zugriffen und Aufspaltung von Knoten bleibt Zugriff auf Gesamtinformation möglich

B+-Bäume ohne Lesesperren

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

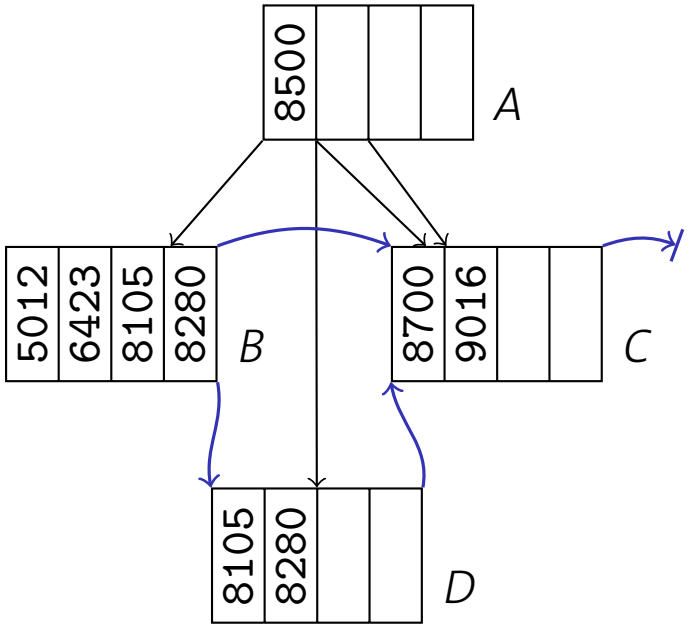
Dieser Zeiger könnte fehlen, sofern gesuchter Wert in Blattknoten durch Pfad 1-2 erreichbar



- Vorher schon betrachtet: Verkettete Liste auf Blattebene
- Zeiger stellen zweiten Pfad auf Knoten bereit (Diese werden schon beim Splitten erzeugt)
- 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 un-lock D
6 set `next` pointer $B \rightarrow D$;
7 adjust content of B ;
8 un-lock B
9 lock & read A ;
10 adjust content of A ;
11 un-lock A



- Alle Indexeinträge sind zu jedem Zeitpunkt erreichbar
- Schritt 1-8 (splitting) und 9-11 (Propagierung auf Vaterknoten) unabhängig

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
- Schreibsperren 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)

- Welche Annahmen müssen wir machen für Sperr-freien Lesezugriff?
- Sperren wollten wir ja nicht verwenden
- Leichtgewichtige Sperren für kurzfristige atomare Ops
- Indexsperren induzieren wenig Verwaltungsaufwand (meist als Spinlocks implementiert; auf der Folie vorher also eigentlich auch latches)
- 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

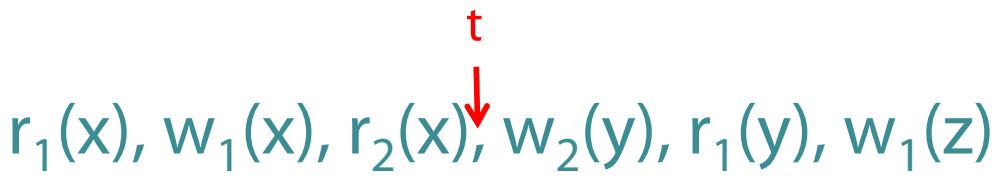
- Gelesenen Attribute (read set $RS(T_i)$)
- Geschriebene Attribute (write set $WS(T_i)$)

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 $WS(T) \cap RS(T_r) = \emptyset$

Multiversions-Nebenläufigkeitsorganisation

Betrachten wir den folgenden Abarbeitungsplan



Ist dieser Plan serialisierbar?

- Angenommen, wenn T_1 den Wert y lesen möchte, sei der „alte“ Wert vom Zeitpunkt t noch verfügbar,
- dann könnten wir eine Historie wie folgt erzeugen

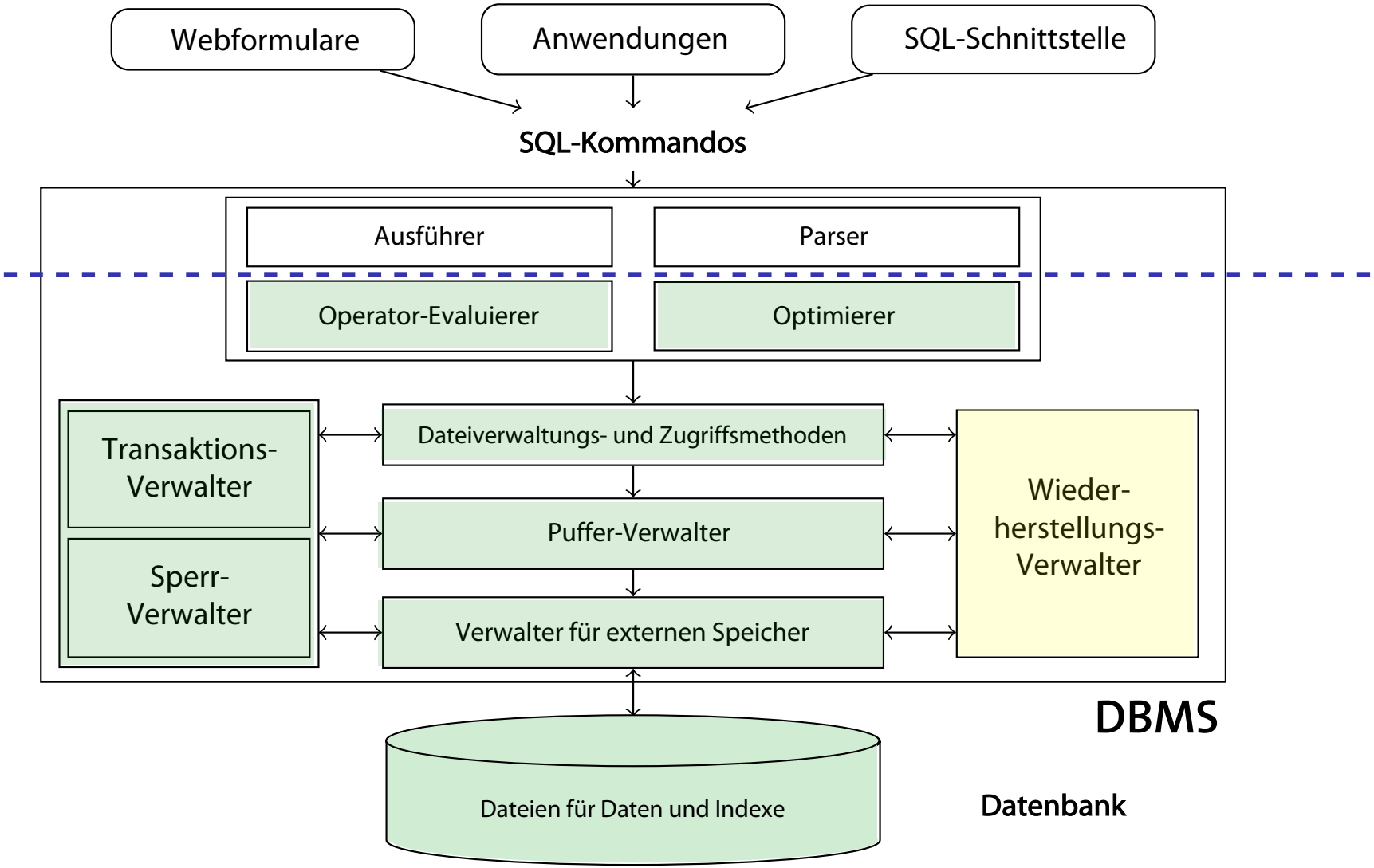
$$r_1(x), w_1(x), r_2(x), r_1(y), w_2(y), w_1(z)$$

die serialisierbar ist

Multiversions-Nebenläufigkeitsorganisation

- Mit verfügbaren alten Objektversionen müssen Leseschritte nicht länger blockiert werden
- Es sind „abgelaufene“, aber konsistente Werte verfügbar (vgl. Dirty-Read-Problematik)
- Problem: Versionierung benötigt Raum und erzeugt Verwaltungsaufwand (Garbage Collection)

Wiederherstellung (Recovery)



Wiederherstellung nach Fehlern

Drei Typen von Fehlern

- Transaktionsfehler (Prozessfehler)
  - Eine Transaktion wird abgebrochen (abort)
  - Alle Änderungen müssen ungeschehen gemacht werden

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

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)

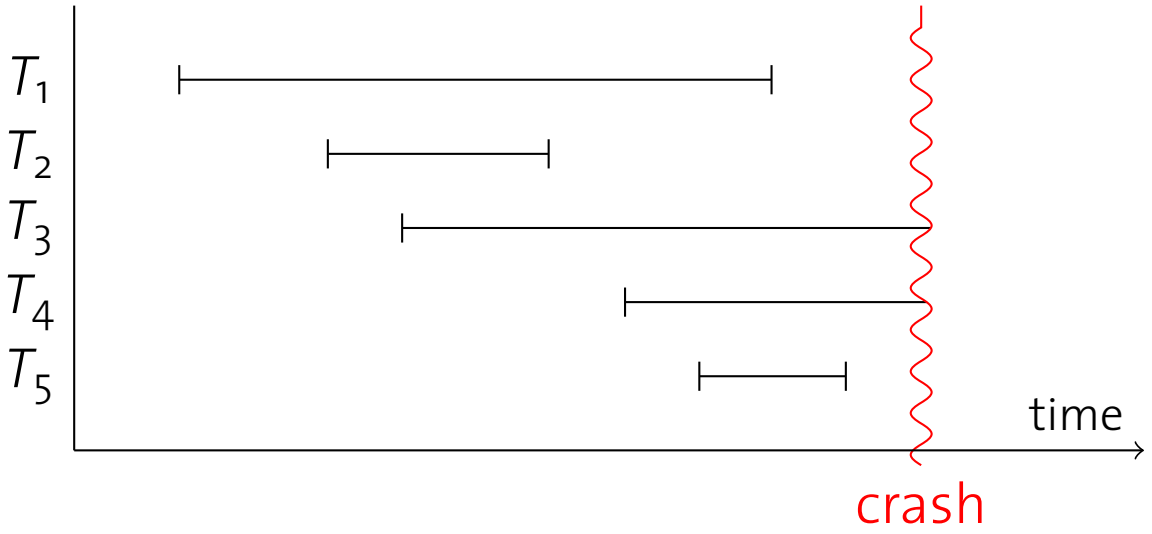
Wiederherstellung nach Fehlern

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: System- oder Medienfehler



- Transaktionen T_1, T_2 und T_5 wurden vor dem Ausfall erfolgreich beendet → Dauerhaftigkeit: Es muss sicher-gestellt 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

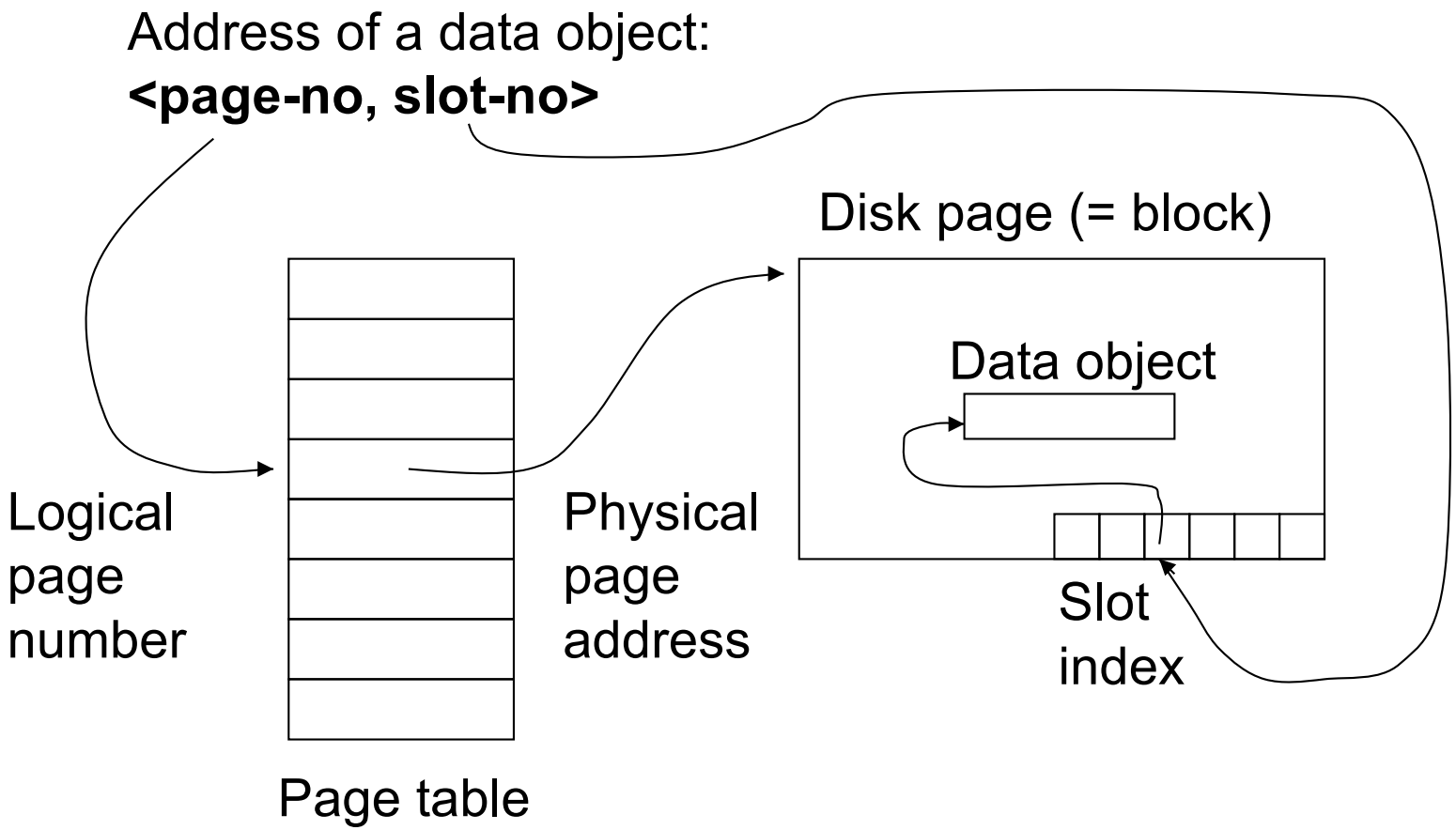
- Flüchtige Speicher
  - Wird vom Pufferverwalter verwendet
    (für Seitencache und auch Transaktions-Verwaltungsdaten)
- Nicht-flüchtige Speicher
  - Festplatten, Solid-State Drives
- Stabile Speicher
  - Nicht-flüchtiger Speicher, der alle drei Arten von Fehlern überlebt. Stabilität kann durch Replikation auf mehrere Platten erhöht werden (auch: Bänder)

Vergleiche Arten von Fehler und Arten von Speichern

Schatten-Seiten-Verwaltung

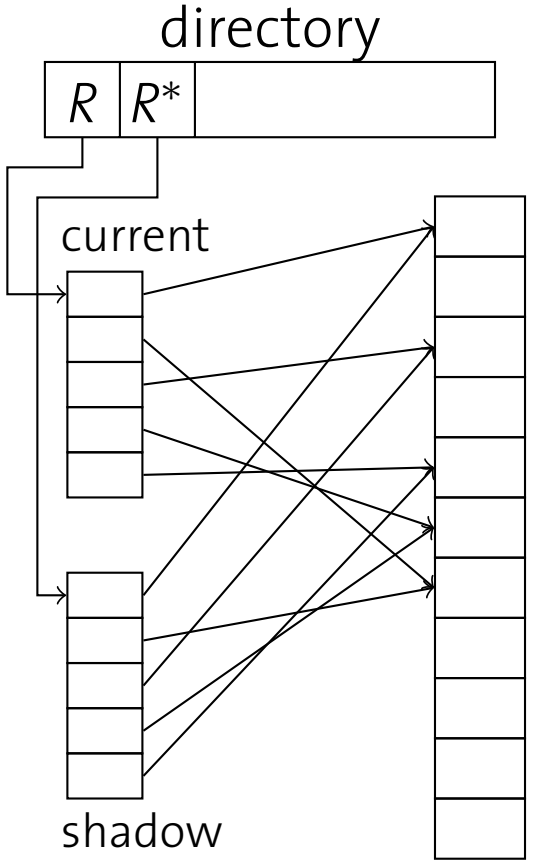
- Fehler können zu jeder Zeit auftreten, also muss das System jederzeit in einen konsistenten Zustand zurückführbar sein
- Dies kann durch Redundanz erreicht werden
- Schatten-Seiten (eingeführt durch IBMs „System R“)
  - Von jeder Seite werden zwei Versionen geführt
  - Aktuelle Version/current (Arbeitskopie, copy-on-write)
  - Schatten-Seite/shadow (konsistente Version auf nicht-flüchtigem Speicher zur Wiederherstellung)

Verwalter für externen Speicher: Indirekte Adressierung



Shadow-Paging: Funktionen

- **Anforderung einer Seite:**
  - Ergänze Seitentabelle um Eintrag
  - Kopiere neuen Seitentabelleneintrag in die Schattentabelle
- **Schreiben einer Seite** (z.B. bei Verdrängung aus Puffer)**:**
  - Fordere neue Seite an
  - Schreibe Pufferinhalt
  - Trage Zeiger auf neue Seite in aktuelle Seitentabelle ein (überschreibe darin enthaltenen Zeiger auf das Original)

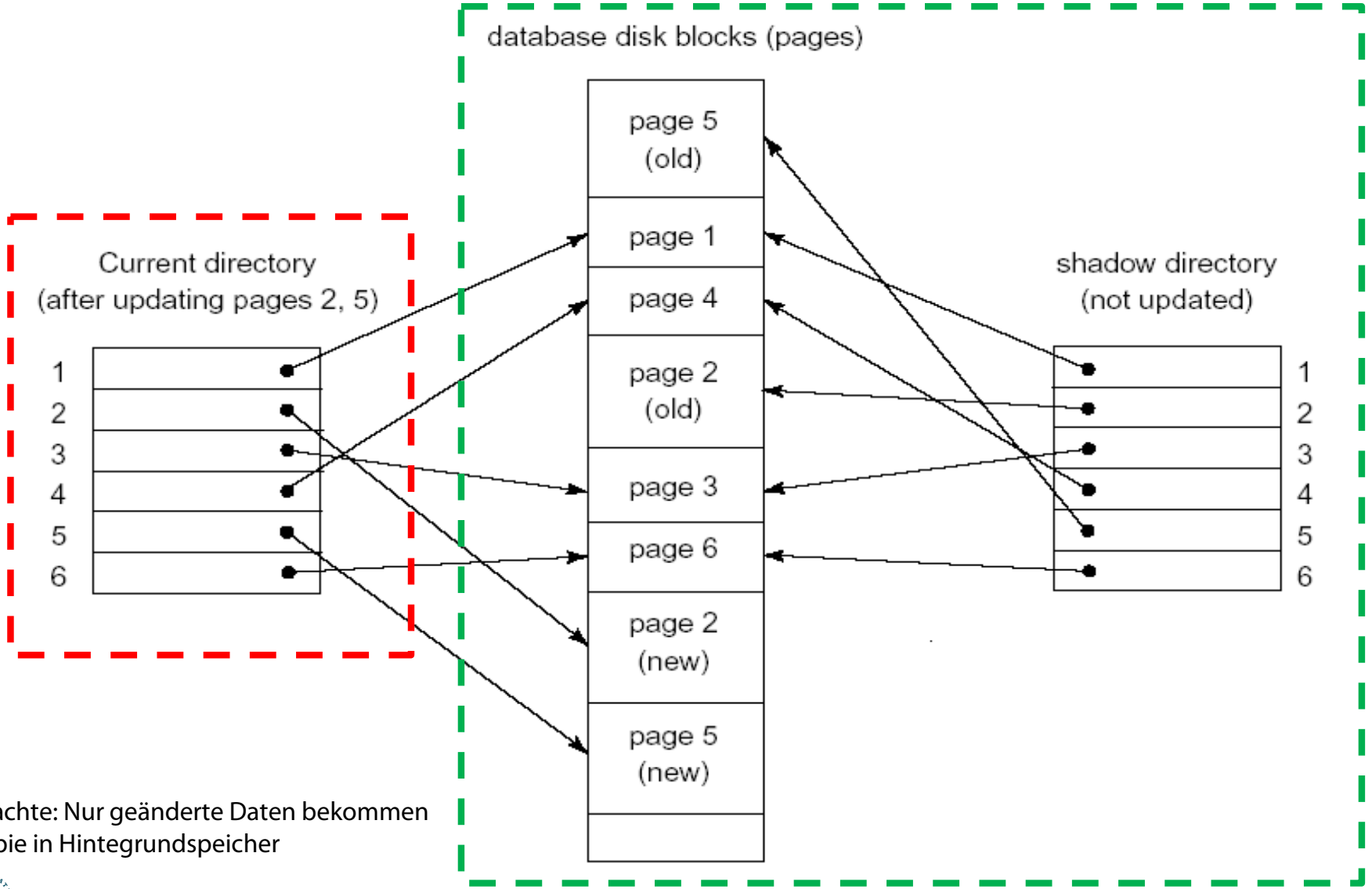


(Ausgangszustand: Current = shadow)

Beispiel

Nichtflüchtiger Speicher

Flüchtiger Speicher



Beachte: Nur geänderte Daten bekommen Kopie in Hintegrundspeicher

Shadow-Paging: Funktionen (2)

- **Commit:**
  - Übernehme aktuelle Tabelle bzw. die darin modifizierten Seiten in nichtflüchtigen Speicher
    - Atomare Ausführung ist nicht trivial
    - Kopie der Relation und atomares Umsetzen der Basisreferenz
  - Verwirf Schattentabellenseiten, d.h. gib referenzierte alte Seiten wieder frei
- **Abort/Recovery:**
  - Bzgl. alter Seiten ist nichts zu tun (es wurde auf Kopie gearbeitet)
  - Aktuelle Version überschrieben durch Schattenversion
  - Gib nicht referenzierte Seiten im nichtflüchtigen Speicher wieder frei (Garbage Collection von modifizierten Seiten)

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 -> Logging nötig

Gray et al.. The Recovery Manager of the System R Database Manager. ACM Comp. Surv., vol. 13(2), June 1981.

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/fixiert ist
  - Geänderte Seiten (dirty pages) werden auf die Platte geschrieben
- Diese Änderungen müssen ungeschehen gemacht werden während der Wiederherstellung
  - Leicht möglich durch Schatten-Seiten

Effekte, die Wiederherstellung berücksichtigen muss

- 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[1]-Wiederherstellungsmethode verwendet ein Write-Ahead-Log zur Implementierung der notwendigen redundanten Datenhaltung
- Datenseiten werden in situ (update-in-place) 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)

[1] Algorithm for Recovery and Isolation Exploiting Semantics
Mohan et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks UsingWrite-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 des Log-Eintrags

Typ (Log Record Type)

- Repräsentiert, ob 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
$a \leftarrow \texttt{read}(A)$;	
	$c \leftarrow \texttt{read}(C)$;
$a \leftarrow a - 50$;	
	$c \leftarrow c + 10$;
$\texttt{write}(a, A)$;	
	$\texttt{write}(c, C)$;
$b \leftarrow \texttt{read}(B)$;	
$b \leftarrow b + 50$;	
$\texttt{write}(b, B)$;	
$\texttt{commit}$;	
	$a \leftarrow \texttt{read}(A)$;
	$a \leftarrow a - 10$;
	$\texttt{write}(a, A)$;
	$\texttt{commit}$;

LSN	Type	TX	Prev	Page	UNxt	Redo	Undo
1	UPD	T_1	–	$\cdots$		$A := A - 50$	$A := A + 50$
2	UPD	T_2	–	$\cdots$		$C := C + 10$	$C := C - 10$
3	UPD	T_1	1	$\cdots$		$B := B + 50$	$B := B - 50$
4	EOT	T_1	3	$\cdots$			
5	UPD	T_2	2	$\cdots$		$A := A - 10$	$A := A + 10$
6	EOT	T_2	5	$\cdots$			

Logische Protokollierung

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
- Logisches Redo (A = A + 50 oder ‚Setze Tupelwert auf v')
  - Redo wird vollständig durchgeführt, auch Indexeinträge würden neu generiert, inkl. Aufspaltung usw.

Redo/Undo-Information

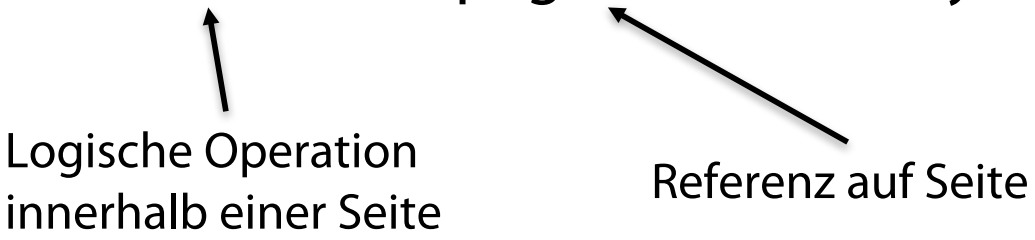
- ARIES unterstützt logisches Undo
- Seitenorientiertes Undo kann kaskadierende Rückroll-Situationen heraufbeschwören
  - Selbst wenn eine Transaktion T_1 nicht direkt Tupel betrachtet hat, die von anderer Transaktion T_2 beschrieben wurden, ist doch das physikalische Seitenlayout, das T_1 sieht, von T_2 beeinflusst
  - T_1 kann nicht vor T_2 erfolgreich beendet werden (selbst wenn es keine (logischen) Konflikte von T1 und T2 gibt
- Logisches Undo erhöht also die Nebenläufigkeit
- Aber: Eine einzelne logische Operation wie insert tuple into table R führt mehre Schritte nach sich: Insert tuple into data pages , insert into index etc.

Kompromiss:

- Kombiniere beide Ansätze: physikalisch auf Seiten zugreifen, logisch innerhalb einer Seite

- Beispiel Logeintrag

  [...,insert,...,page 4711,...,record value r]
  [...,ix insert,...,ix page 0815,...,ix key: k1, rid: v]
  [...,ix insert,...,ix page 4242,...,ix key: k2, rid: v]



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 der Effekten von T 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 weiter zu (s. aber Schnappschüsse weiter unten)

Normaler Verarbeitungsmodus

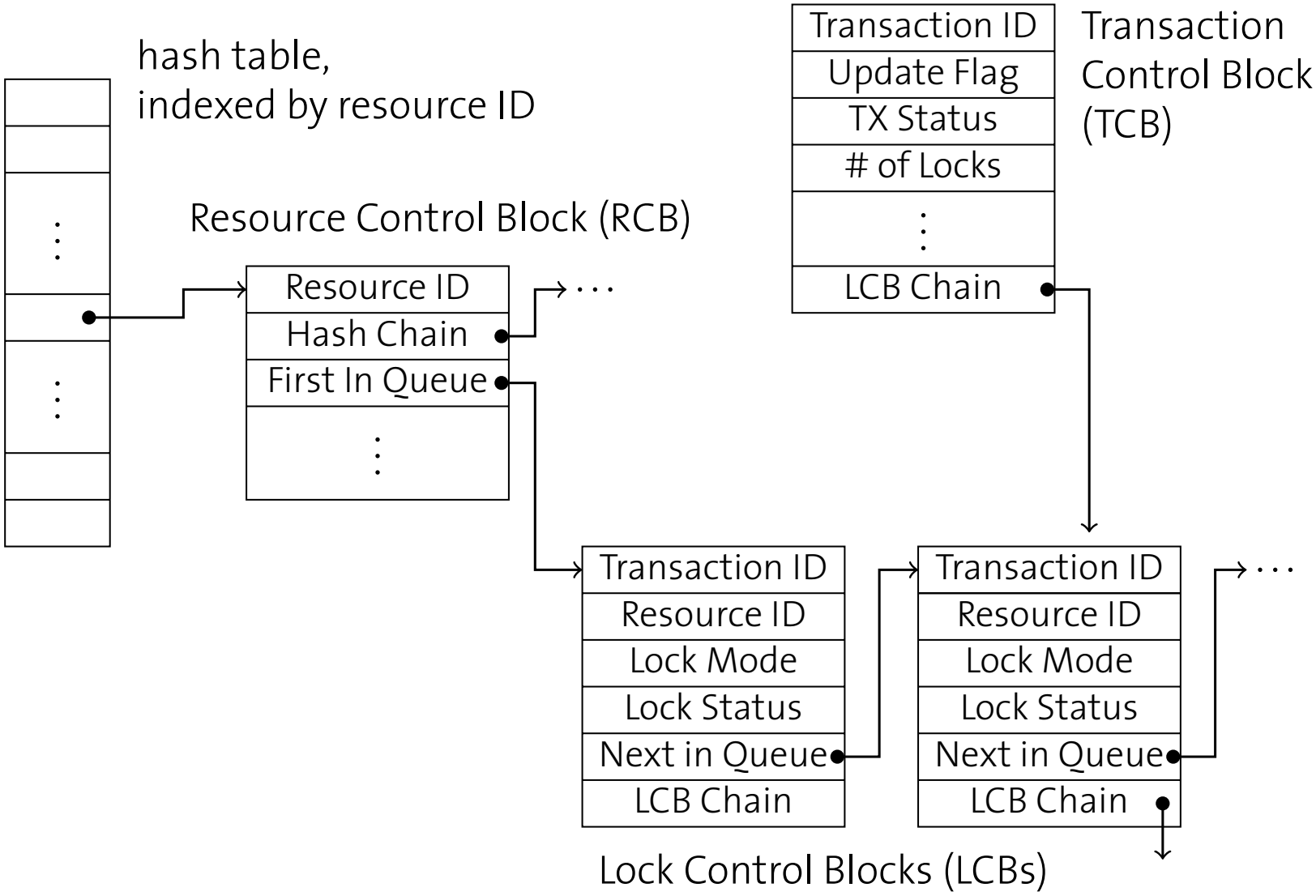
Während der normalen Transaktionsverarbeitung werden zwei Dinge im Transaktionskontrollblock gespeichert

- 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

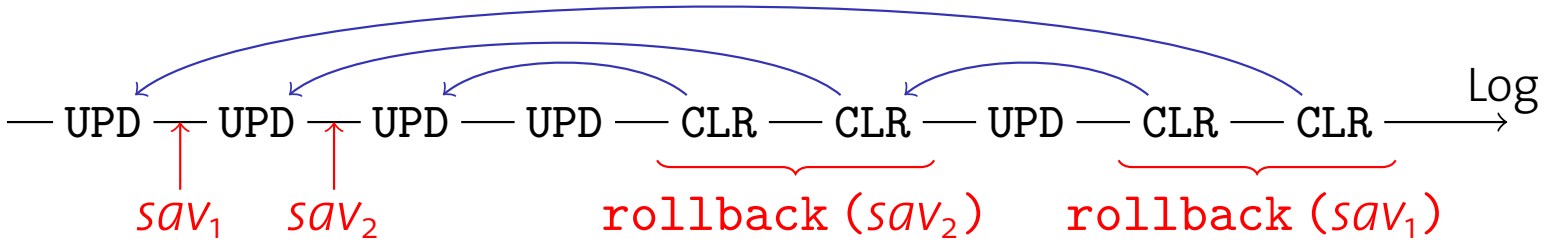
- Logging der Undo-Operationen im WAL
- Verwendung von Compensation-Log-Record (CLRs) für diesen Zweck

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
               < LSN', CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, ··· , ∅⟩ ;
9              In page header of LogRec.Page: Set LSN = LNS'
10             T.LastLSN ← LNS'
11         case CLR
12             UndoNxt ← LogRec.UNxt ;
13     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:
   - Lies Log in Vorwärtsrichtung
   - Bestimme Transaktionen, die aktiv waren als der Absturz passierte (solche Transaktionen nennen wir Pechvögel)
2. Redo-Phase:
   - Spiele 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 LocRec.Type do
4         case UPD or CLR
5             v ← pin (LogRec.Page) ;
6             if v.LSN < LogRec.LSN then
7                 perform redo operation LogRec.Redo on v ;
8                 v.LSN ← LogRec.LSN ;
9             unpin (v, ···) ;
```

Auch beim Wiederherstellen können Abstürze eintreten

- Undo und Redo einer Transaktion müssen idempotent sein
  - undo(undo(T)) = undo(T) // z.B. nicht zweimal dasselbe Tupel einfügen bei insert
  - redo(redo(T)) = redo(T)
- Prüfe LSN vor der Redo-Operation (Zeile 6)

Wiederholung

Funktion pin für Anfragen nach Seiten und
unpin für Freistellungen von Seiten nach Verwendung

- pin(pageno)
  - Anfrage nach Seitennummer pageno
  - Lade Seite in Hauptspeicher falls nötig
  - Rückgabe einer Referenz auf pageno
- unpin(pageno, dirty)
  - Freistellung einer Seite pageno zur möglichen Auslagerung
  - dirty = true bei Modifikationen der Seite

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 Transaktionen auf einmal zurückgerollt (alle Pechvögel)
- Alle Pechvögel werden vollständig zurückgerollt

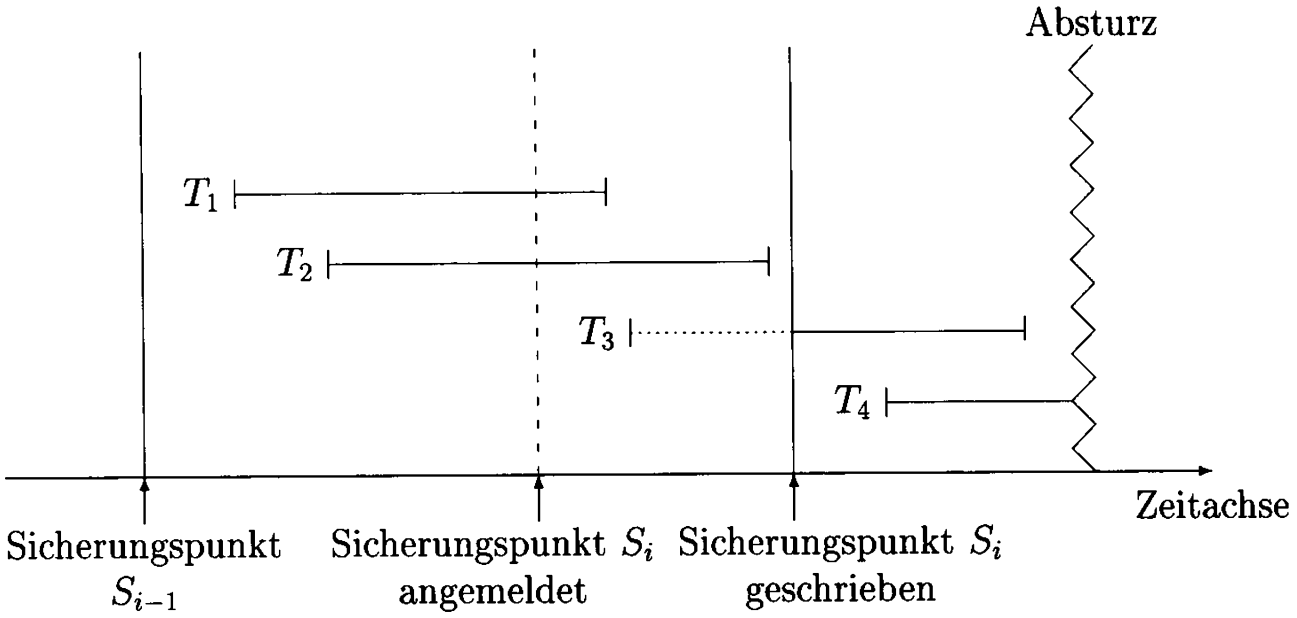
Undo-Phase

```
1  Function: undo ()
2  while transactions (i.e., TCBs) left to roll back do
3      T ← TCB of loser transaction with greatest UNxt ;
4      LogRec ← 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 ← write log entry
                       ⟨CLR, T, T.LastLSN, LogRec.Page, LogRec.Prev, · · · , ∅⟩ ;
9              set LSN = LSN in page header of LogRec.Page ;
10             T.LastLSN ← LSN ;
11         case CLR
12             UndoNxt ← LogRec.UNxt ;
13     T.UNxt ← UndoNxt ;
14     if T.UNxt = '–' then
15         write EOT log entry for T ;
16         delete TCB for T ;
```

Checkpointing

- WAL ist eine immer-wachsende 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)

Beispiel schwergewichtiger checkpoint



- Nach Absturz Logdatei ab Si nötig.
- Hierfür
  - Für T1 und T2 nichts zu tun (da bereits persistent)
  - T3 nach Anmeldung von Si verzögert;
  - Merke Änderungen von T3 für Redo
  - Merke Änderungen von T4 für Undo

Medien-Wiederherstellung

- Um Medienfehler zu kompensieren, muss peridiodisch 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

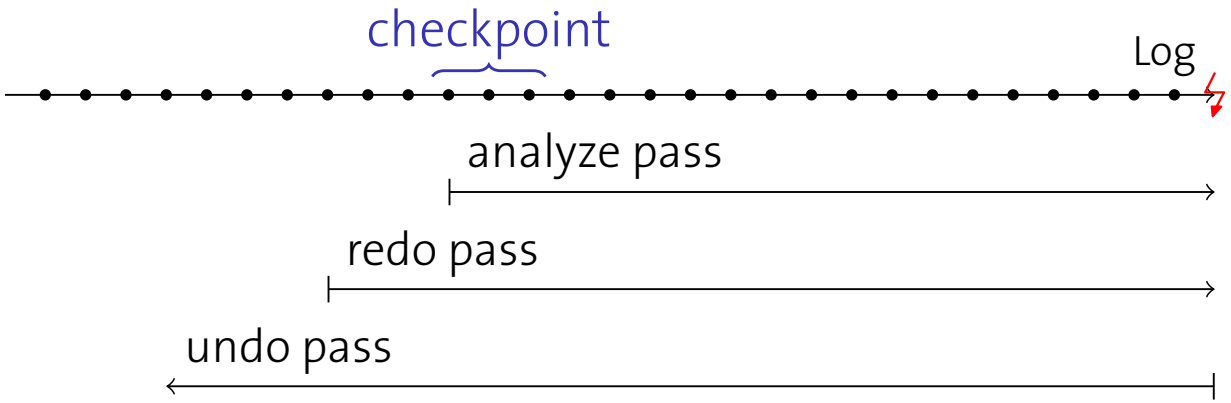
1. Schreibe Begin-Checkpoint-Logeintrag BCK
2. Sammle Information
   - über geänderte Seiten im Pufferverwalter und dem LSN ihrer letzten Modifikationsoperation und
   - über alle aktiven Transaktionen (und ihrer LastLSN und UNxt TCB-Einträge)

   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