Datenbanken

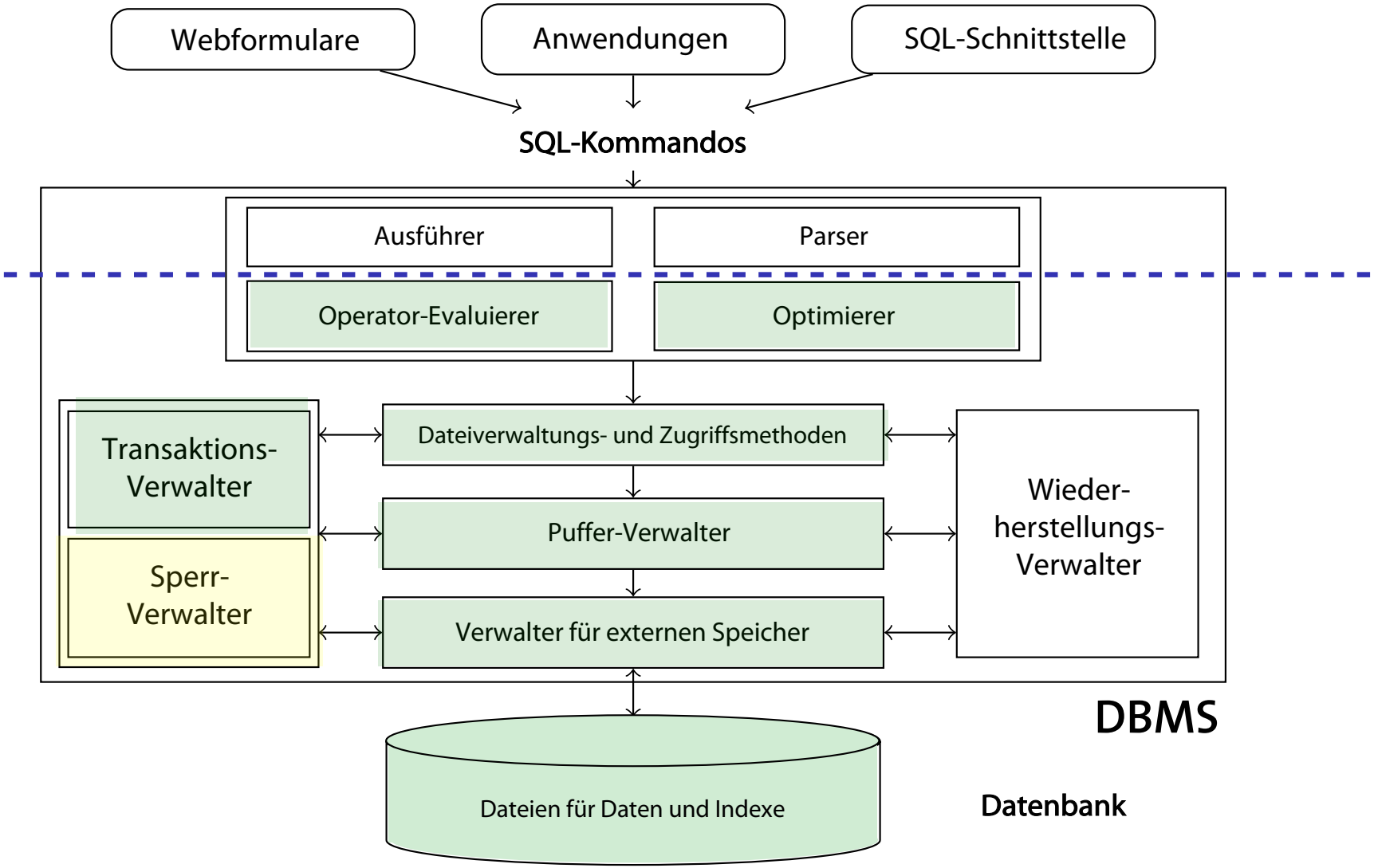
Transaktionsmanagement Teil 2

Dr. Özgür Özçep

Universität zu Lübeck

Institut für Informationssysteme

Transaktionsverwaltung



Motivation

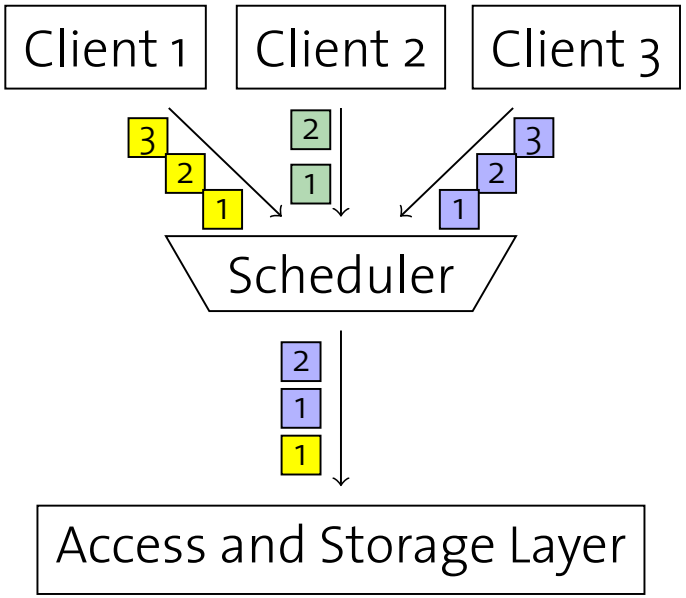
Sperren im Anfrageplan

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

Idee:

- Lass 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

- 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

- Kein Lost Update
- Aber: Sperren müssen richtig eingesetzt werden

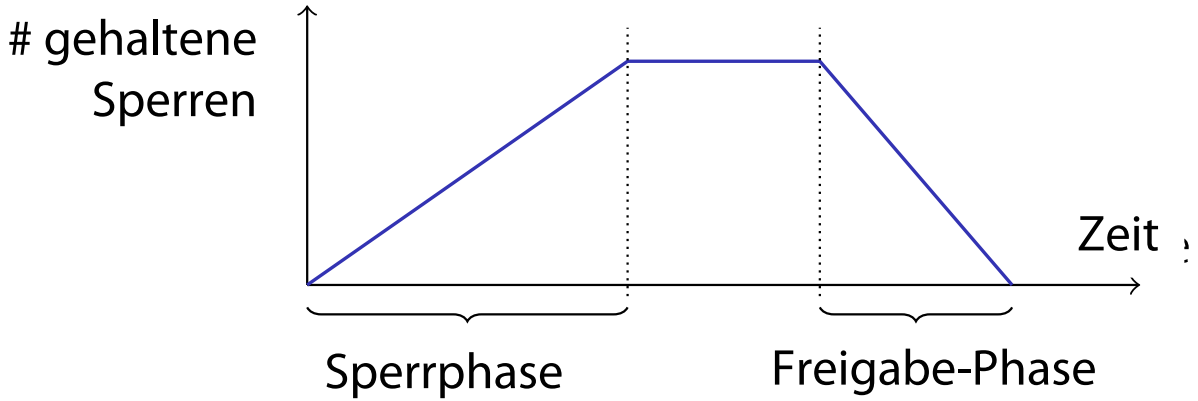
-> Sperrprotokolle

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

2-Phasen-Sperrprotokoll

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



- Serialisierbarkeit gewährleistet in fehlerfreier Umgebung

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)

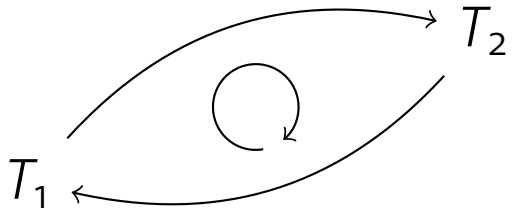
- Wir haben gesehen, dass zwei Leseoperationen nicht in Konflikt zueinander stehen
- Systeme verwenden verschiedene Arten von Sperren
  - Lesesperren (read locks, shared locks): Modus S
  - Schreibsperren (write locks, exclusive locks): Modus X
- Locks stehen nur in Konflikt zueinander, wenn eines davon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

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

Serialisierungsgraph

Beispiel: ATM-Transaktion

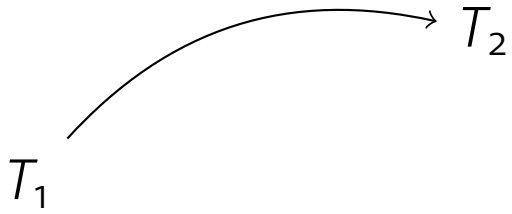
- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$



nicht serialisierbar

Beispiel: Zwei Geldtransfers

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



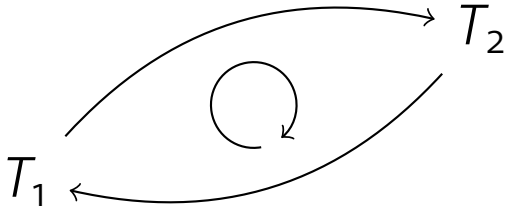
serialisierbar

Serialisierungsgraph

Beispiel: ATM-Transaktion

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

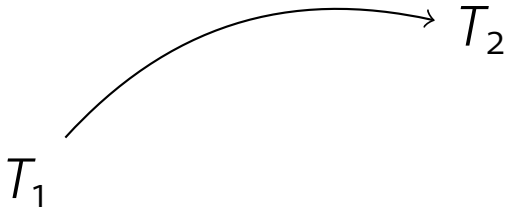
XLock



nicht serialisierbar

Beispiel: Zwei Geldtransfers

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

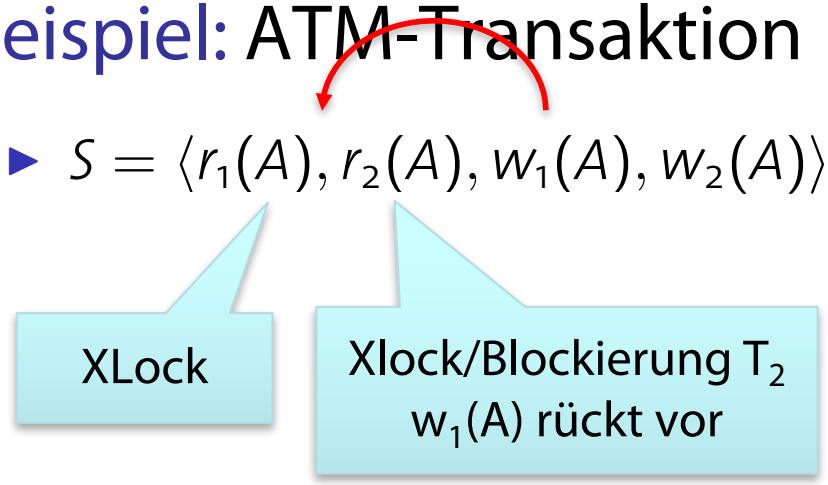


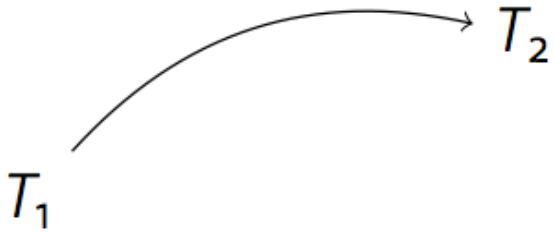
serialisierbar

Serialisierungsgraph

Beispiel: ATM-Transaktion

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

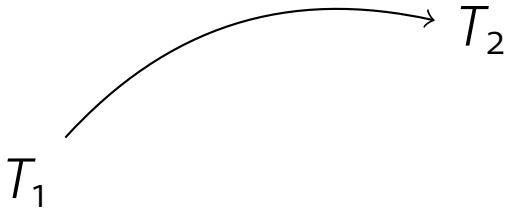




serialisierbar

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

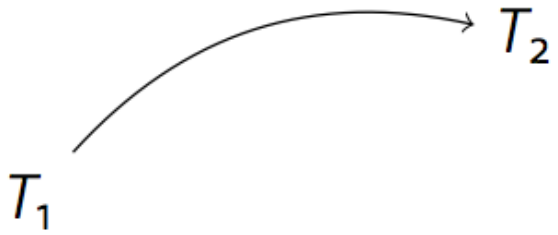
Beispiele noch einmal: Serialisierungsgraph

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

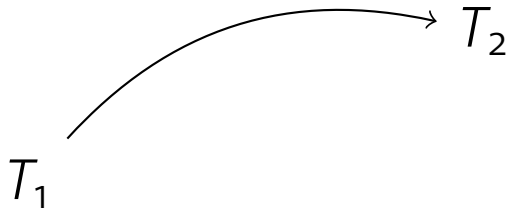


serialisierbar / korrekt

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

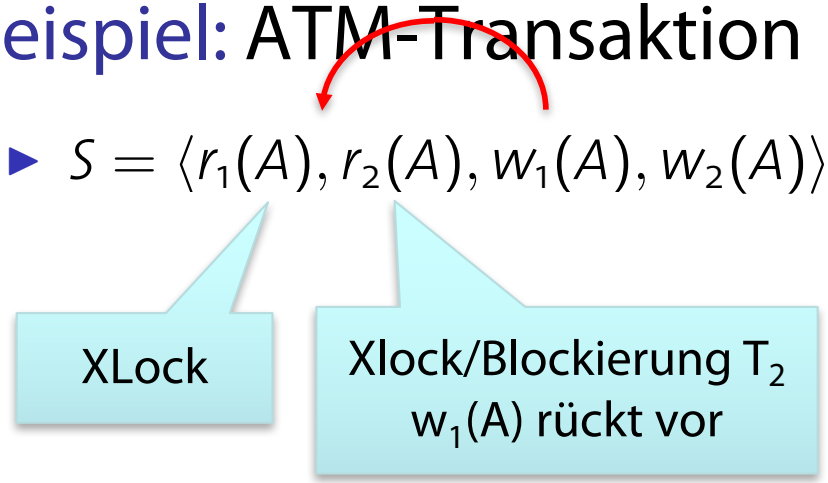


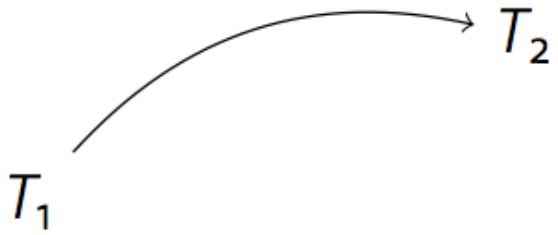
serialisierbar

Beispiele noch einmal: Serialisierungsgraph

Beispiel: ATM-Transaktion

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

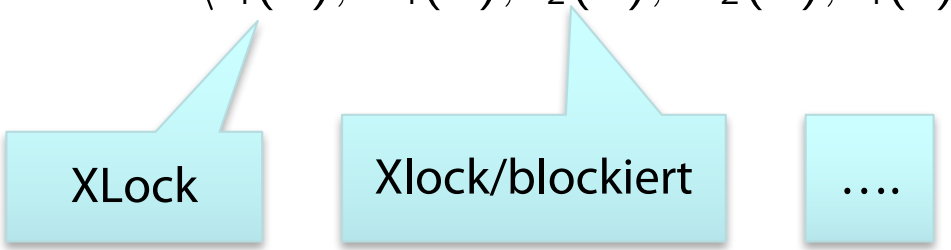


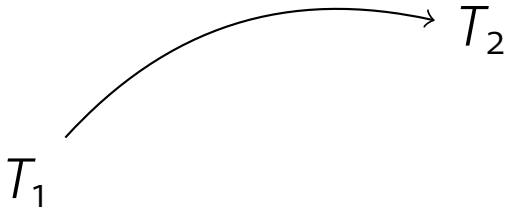


serialisierbar / korrekt

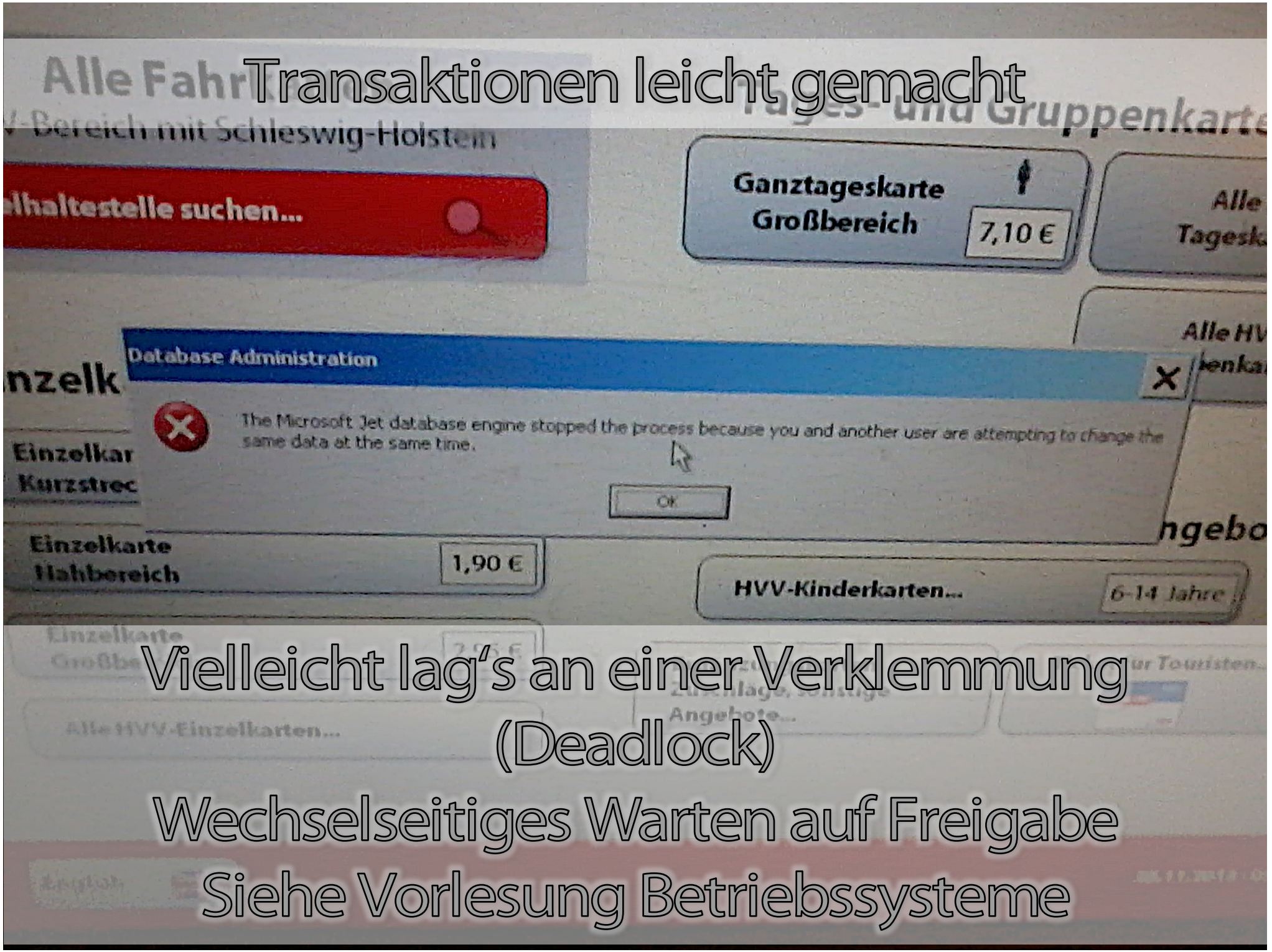
Beispiel: Zwei Geldtransfers

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



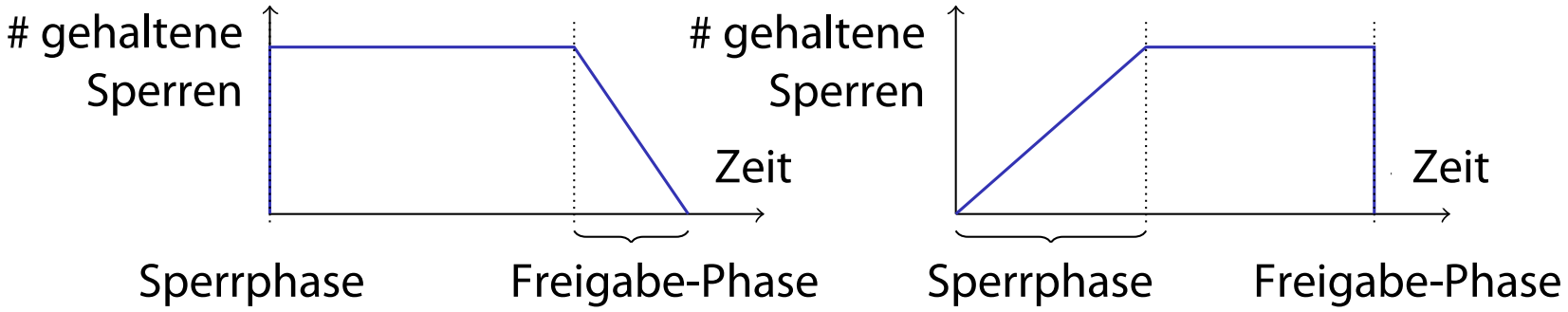


serialisierbar


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 Hahbereich
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? (Übungsaufgabe)

Implementierung eines Sperrverwalters

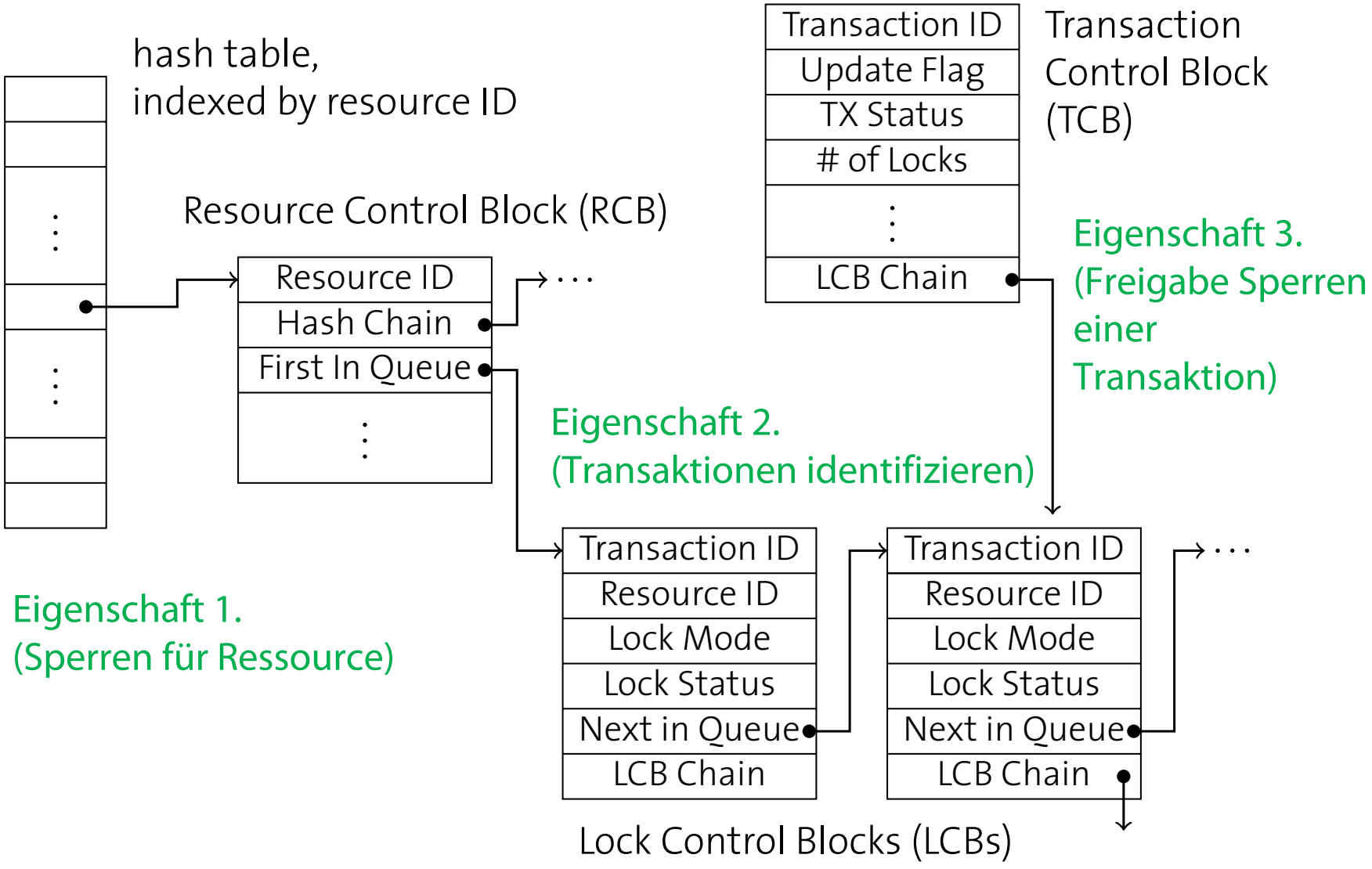
Implementierung eines Sperrverwalters

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



Granularität von Sperren

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

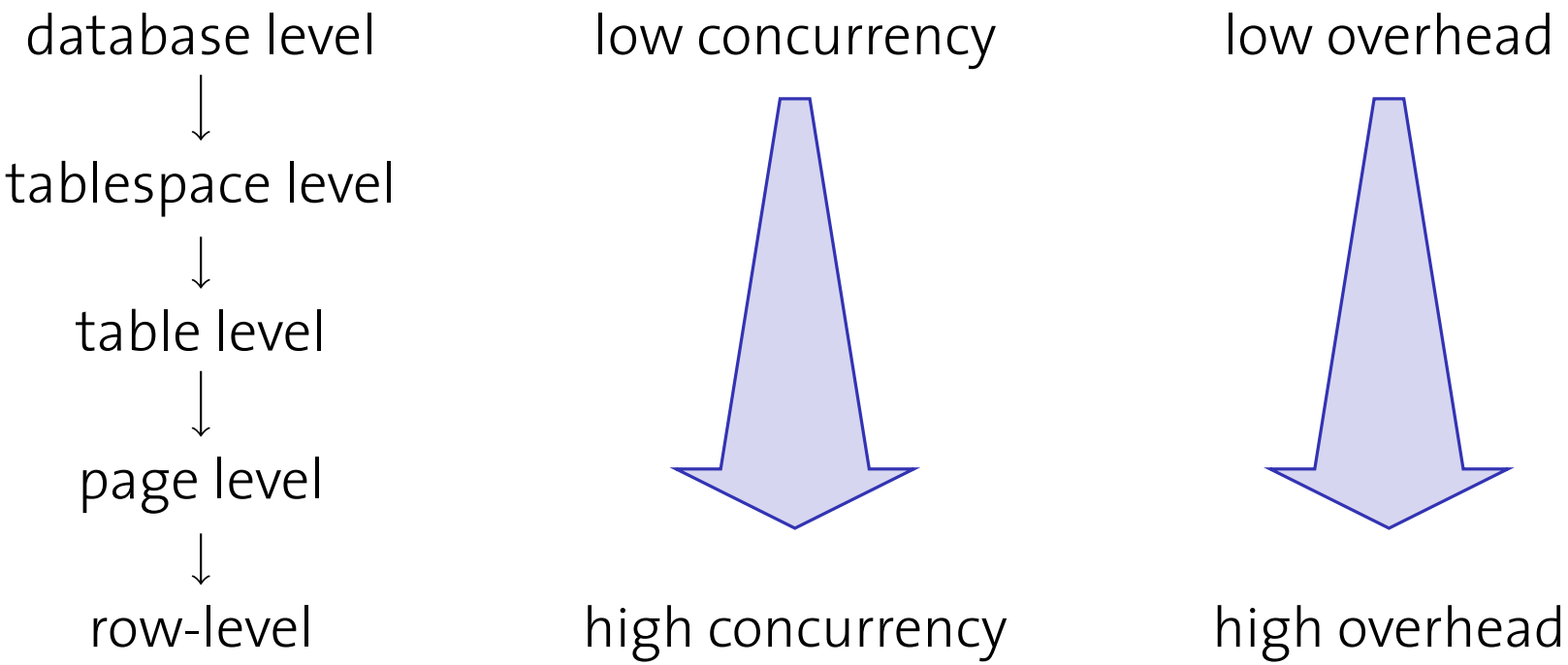
	`INSERT INTO Customers` `VALUES (..., 'Sam', ...)`	ok

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

- Ungern ganze Relation sperren (-> 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 (vorhandenen oder noch einzufügenden) Tupel mit dem Indexschlüssel „Sam“

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 $\square \in \{S, X\}$ sperren
2. Bevor Ebene g in Modus $\square$ gesperrt werden kann, muss eine Sperre I$\square$ 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) 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 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

Entdeckung von Konflikten

Nehmen wir an, folgende Anfrage ist auch noch zu bearbeiten

Q_3:

```
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)

Konsistenz und Isolationslevel

Erinnerung: ACID-Eigenschaften für Transaktionen

- **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 anderer parallel laufender Transaktionen
- **Durability:** Effekte einer erfolgreichen Transaktion werden persistent gemacht

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 (cursor) 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

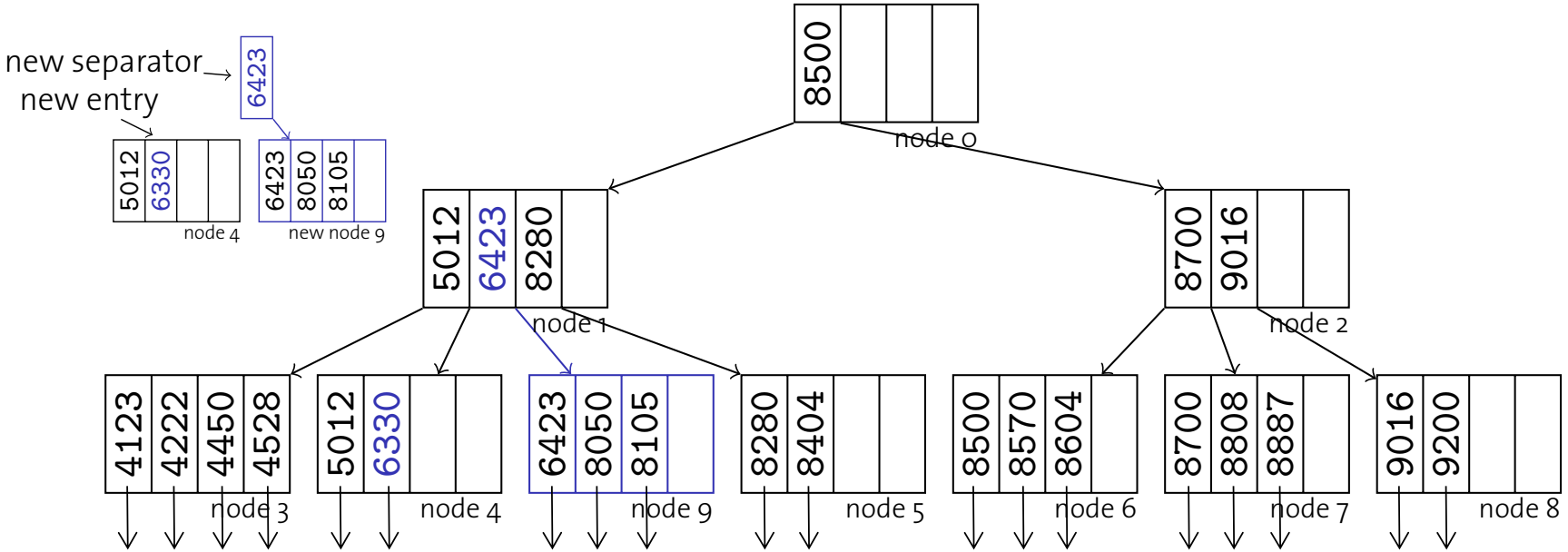
isolation level	dirty read	non-repeat. rd	phantom rd
read uncommitted	possible	possible	possible
read committed	not possible	possible	possible
repeatable read	not possible	not possible	possible
serializable	not possible	not possible	not possible

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

Kurzüberblick: Sperren von Indexen

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

- Wenn Aufspaltung gerade erfolgt ist, ist der neue Separator 6423 noch nicht etabliert
- Annahme: Nebenläufiges Lesen in Trans. T_r sucht 8050
  - Die Verzeigerung weist auf Knoten $node_4$
  - $Node_4$ enthält aber 8050 nicht mehr! Nicht auffindbar!
- Also: Auch in B-Bäumen muss beim Umbau mit Sperren gearbeitet werden!
- 2PL problematisch: reduziert nebenläufigen Zugriff
- Lösungen:
  - Sperrkopplung Write-Only-Tree-Locking (WTL)
  - Protokolle ohne Lesesperren (Indexsperren: Latches)

Kurzüberblick: Optimistische Organisation der Nebenläufigkeit

Optimistische Organisation der Nebenläufigkeit

- Bisher waren wir pessimistisch
  - Wir haben uns immer den schlimmsten 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

Multiversion-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 Platz und erzeugt Verwaltungsaufwand (Garbage Collection)