

2. Synchronisation in DBS: Grundlagen, Sperrverfahren

- Anomalien im Mehrbenutzerbetrieb
- Serialisierbarkeit
 - Abhängigkeiten, Abhängigkeitsgraph
- Zweiphasen-Sperrprotokolle
 - RX-Verfahren
 - Sperrkonversionen
- Konsistenzstufen von Transaktionen
- hierarchische Sperrverfahren
- Deadlock-Behandlung
 - Timeout
 - Wait/Die, Wound/Wait, WDL
 - Erkennung
- Implementierung der Datenstrukturen (Sperrtabelle)



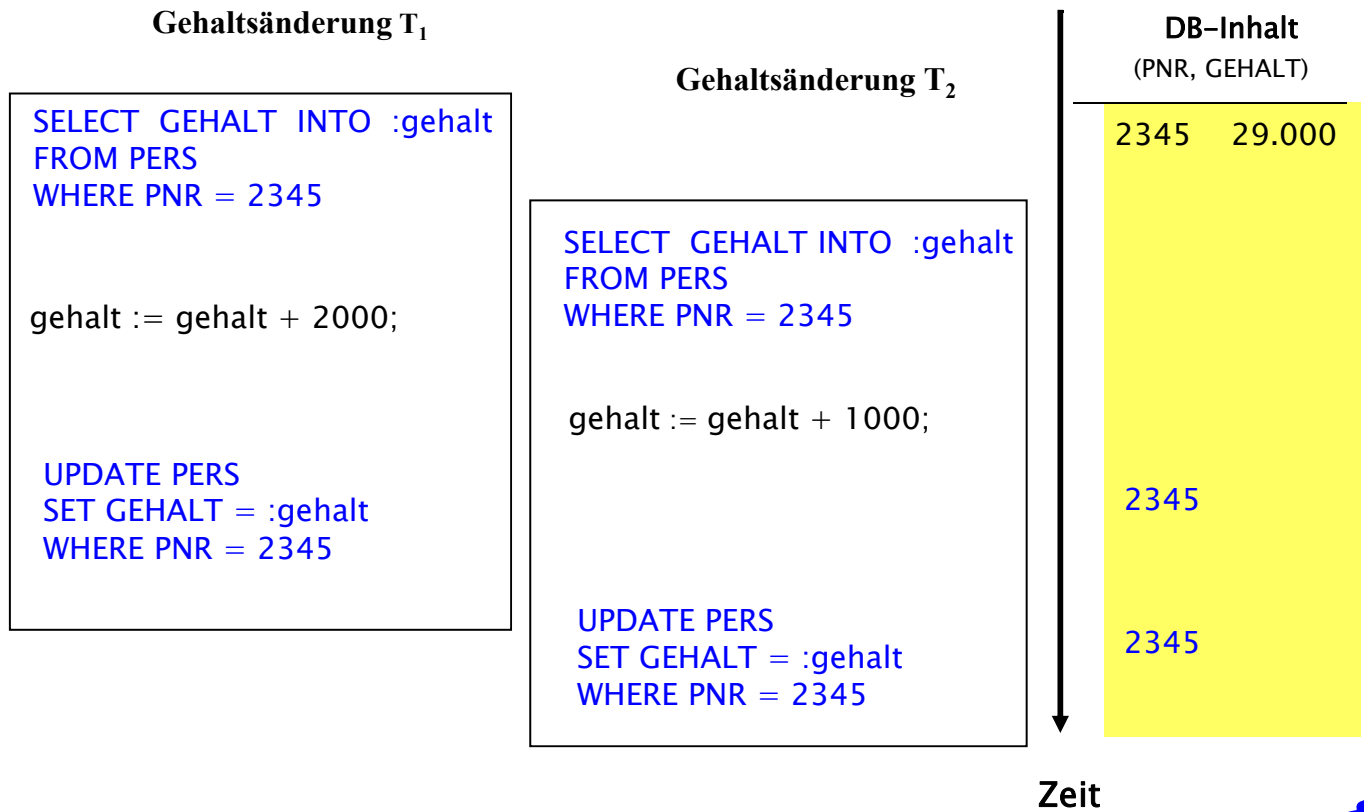
Mehrbenutzerbetrieb

- viele gleichzeitige Nutzer auf derselben DB
 - Mehrbenutzerbetrieb mit paralleler Ausführung unabhängiger Transaktionen
- serielle (sequenzielle) Ausführung von Transaktionen inakzeptabel
 - lange Wartezeiten für neue Transaktionen/DB-Anfragen bis laufende Transaktion und andere bereits wartende Transaktionen beendet sind
 - sehr schlechte CPU-Nutzung aufgrund zahlreicher Transaktionsunterbrechungen: E/A, Kommunikationsvorgänge
- Anomalien im Mehrbenutzerbetrieb ohne Synchronisation
 1. verlorengangene Änderungen (*lost updates*)
 2. Abhängigkeiten von nicht freigegebenen Änderungen (*dirty read, dirty overwrite*)
 3. inkonsistente Analyse (*non-repeatable read*)
 4. Phantom-Problem

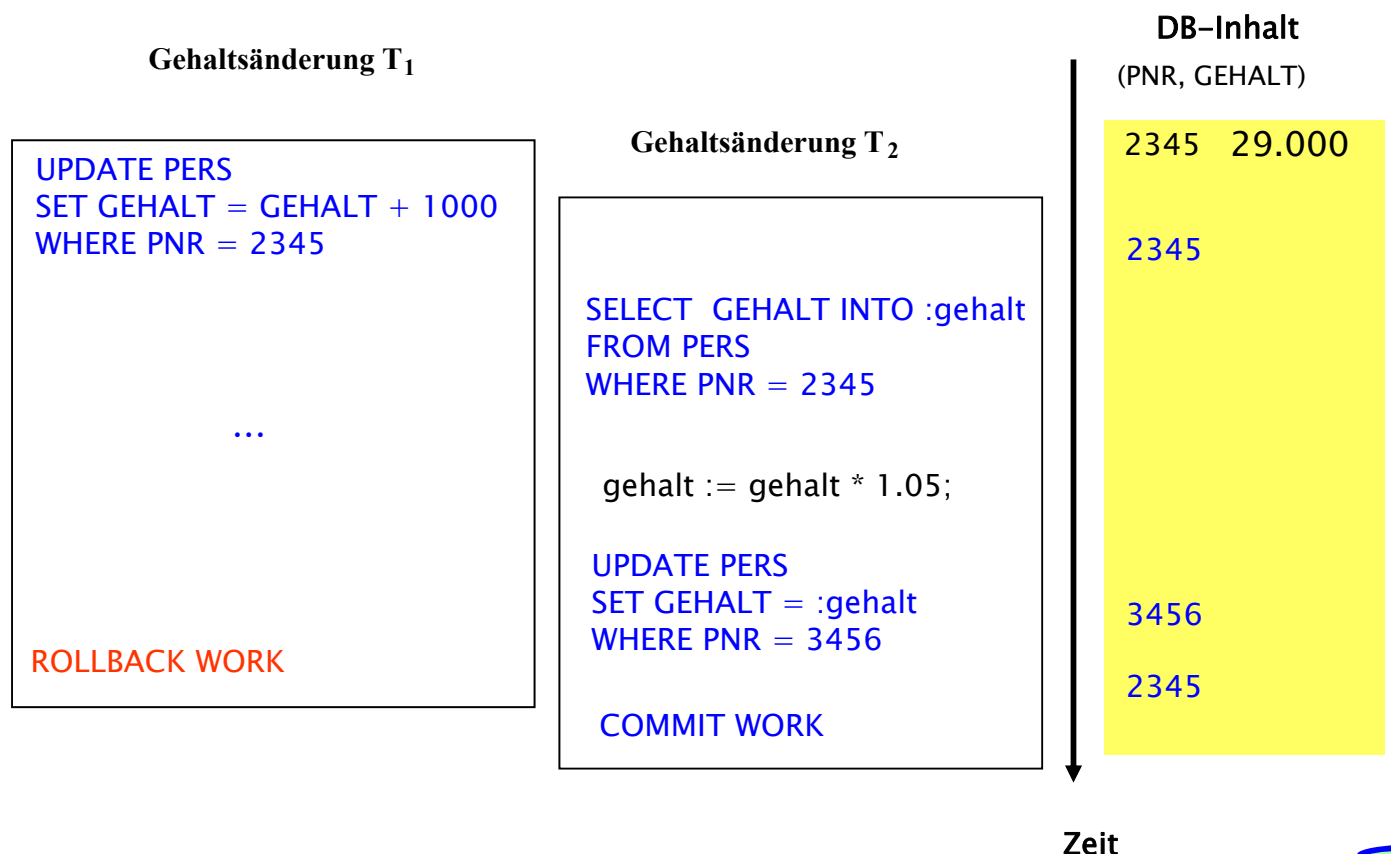
→ nur durch Änderungstransaktionen verursacht



Verloren gegangene Änderung (Lost Update)



Schmutziges Lesen (Dirty Read)



Inkonsistente Analyse (Non-repeatable Read)

Lesetransaktion

(Gehaltssumme berechnen)

```
SELECT GEHALT INTO :gehalt  
FROM PERS  
WHERE PNR = 2345  
summe = summe + gehalt
```

```
SELECT GEHALT INTO :gehalt  
FROM PERS  
WHERE PNR = 3456  
summe = summe + gehalt  
COMMIT WORK
```

Änderungstransaktion

```
UPDATE PERS  
SET GEHALT = GEHALT + 1000  
WHERE PNR = 2345  
  
UPDATE PERS  
SET GEHALT = GEHALT + 2000  
WHERE PNR = 3456  
  
COMMIT WORK
```

DB-Inhalt

(PNR, GEHALT)

2345	29.000
3456	38.000

2345	30.000
------	--------

3456	40.000
------	--------

Zeit



Phantom-Problem

Lesetransaktion

(Gehaltssumme überprüfen)

```
SELECT SUM (GEHALT) INTO :summe  
FROM PERS  
WHERE ANR = 17
```

```
SELECT SUM (GEHALT) INTO :summe2  
FROM PERS  
WHERE ANR = 17
```

```
IF summe <> summe2 THEN  
  <Fehlerbehandlung>
```

Änderungstransaktion

(Einfügen eines neuen Angestellten)

```
INSERT INTO PERS (PNR, ANR, GEHALT)  
VALUES (4567, 17, 55.000)
```

```
COMMIT WORK
```

Zeit



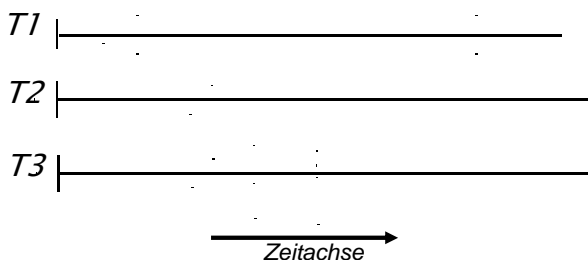
Synchronisation von Transaktionen: Modellannahmen

- Transaktion: Programm T mit DB-Anweisungen
- Annahme: Wenn T allein auf einer konsistenten DB ausgeführt wird, dann terminiert T (irgendwann) und hinterlässt DB in einem konsistenten Zustand
 - keine Konsistenzgarantien während der Transaktionsverarbeitung
- wenn mehrere Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten
- DB-Anweisungen lassen sich nachbilden durch READ- und WRITE-Operationen
- Transaktion besteht aus
 - BOT (Begin Of Transaction)
 - Folge von READ- und WRITE-Anweisungen auf Objekte
 - EOT (End of Transaction): Commit oder Rollback (Abort)



Modellannahmen (2)

- die Ablauffolge von Transaktionen mit ihren Operationen kann durch einen *Schedule* beschrieben werden:
 - BOT ist implizit
 - $r_i(x)$ bzw. $w_i(x)$: Read- bzw. Write-Operation durch Transaktion i auf Objekt x
 - EOT wird durch c_i (Commit) oder a_i (Abort / Rollback) dargestellt
- Beispiel: $r_1(x), r_2(x), r_3(y), w_1(x), w_3(y), r_1(y), c_1, r_3(x), w_2(x), a_2, w_3(x), c_3, \dots$



- Beispiel eines *seriellen Schedules*:

$r_1(x), w_1(x), r_1(y), c_1, r_3(y), w_3(y), r_3(x), c_3, r_2(x), w_2(x), c_2, \dots$



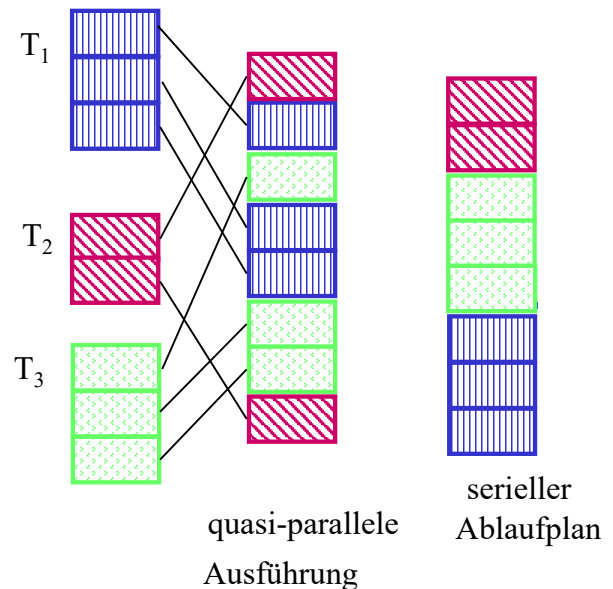
Korrektheitskriterium der Synchronisation: Serialisierbarkeit

- Ziel der Synchronisation: logischer Einbenutzerbetrieb, d.h. Vermeidung aller Mehrbenutzeranomalien
- gleichbedeutend mit formalem Korrektheitskriterium der *Serialisierbarkeit*:

Die parallele Ausführung einer Menge von n Transaktionen ist **serialisierbar**, wenn es eine **serielle** Ausführung der selben Transaktionen gibt, die für einen Ausgangszustand der DB den gleichen Endzustand der DB wie die parallele Transaktionsausführung erzielt.

■ Hintergrund:

- serielle Ablaufpläne sind korrekt
- jeder Ablaufplan, der den selben Effekt wie ein serieller erzielt, ist akzeptierbar



Abhängigkeiten

- Nachweis der Serialisierbarkeit kann über Betrachtung von Abhängigkeiten zwischen Transaktionen geführt werden
- **Abhängigkeit** (Konflikt) $T_1 \rightarrow T_2$ besteht, wenn Transaktion T_1 zeitlich vor Transaktion T_2 auf das selbe Objekt zugreift und die Zugriffe mit nicht reihenfolgeunabhängigen Operationen erfolgten
 - zwei Lesezugriffe sind reihenfolgeunabhängig
 - Schreibzugriffe auf dem selben Objekt verletzen Reihenfolgeunabhängigkeit (unterschiedliche Ergebnisse für Zugriffe vor vs. nach dem Schreibzugriff)
- Konfliktarten:
 - Schreib-/Lese (WR)-Konflikt
 - Lese-/Schreib (RW)-Konflikt
 - Schreib-/Schreib(WW)-Konflikt

- Beispiel: $r_1(x), w_2(x), w_3(y), w_1(y), r_3(x)$

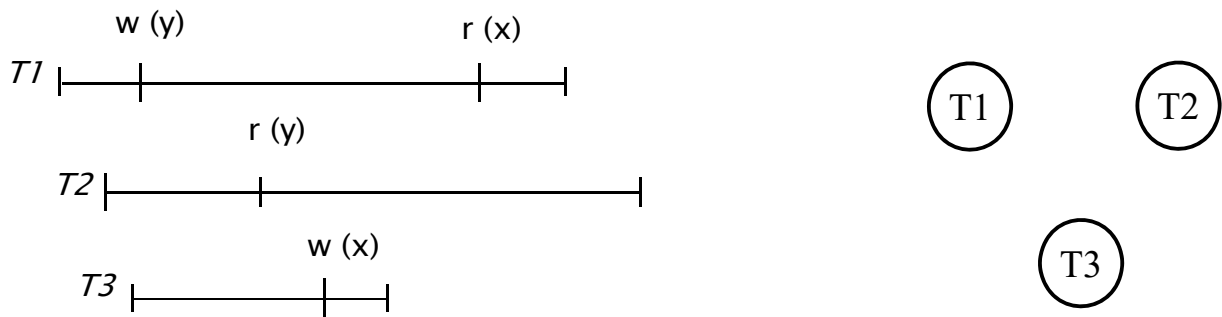
RW-Konflikt:

WW-Konflikt:

WR-Konflikt:

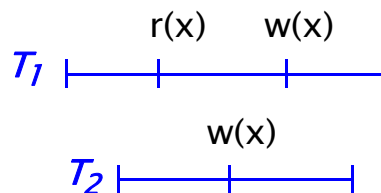
Nachweis der Serialisierbarkeit

- Führen von zeitlichen Abhängigkeiten zwischen Transaktionen in einem *Abhängigkeitsgraphen* (*Konfliktgraphen*)
 - Knoten: Transaktionen (laufend oder bereits beendet)
 - Kante $T_1 \rightarrow T_2$: Konflikt auf einem Objekt, T_1 hat vor T_2 Objekt bearbeitet
- Serialisierbarkeit liegt vor, wenn der Abhängigkeitsgraph keine Zyklen enthält
 - => Abhängigkeitsgraph beschreibt partielle Ordnung zwischen Transaktionen, die sich zu einer vollständigen erweitern lässt (*Serialisierungsreihenfolge*)

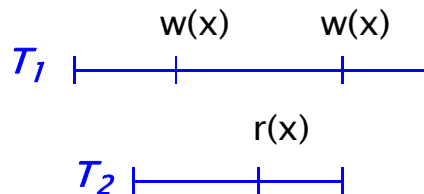


Anomalien im Schreib/Lese-Modell

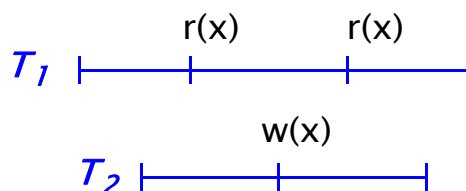
Lost Update



Dirty Read



Non-repeatable Read



Konsistenzzerhaltende Ablaufpläne

■ bei n Transaktionen bestehen n! mögliche serielle Schedules

- z.B. für drei Transaktionen T1, T2, T3 muss so synchronisiert werden, dass der resultierende DB-Zustand gleich dem ist, der bei der seriellen Ausführung in einer der folgenden Sequenzen zustande gekommen wäre.

T1 < T2 < T3

T2 < T1 < T3

T3 < T1 < T2

T1 < T3 < T2

T2 < T3 < T1

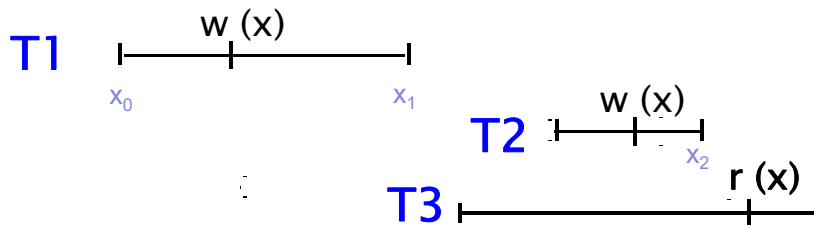
T3 < T2 < T1

■ sinnvolle Einschränkungen:

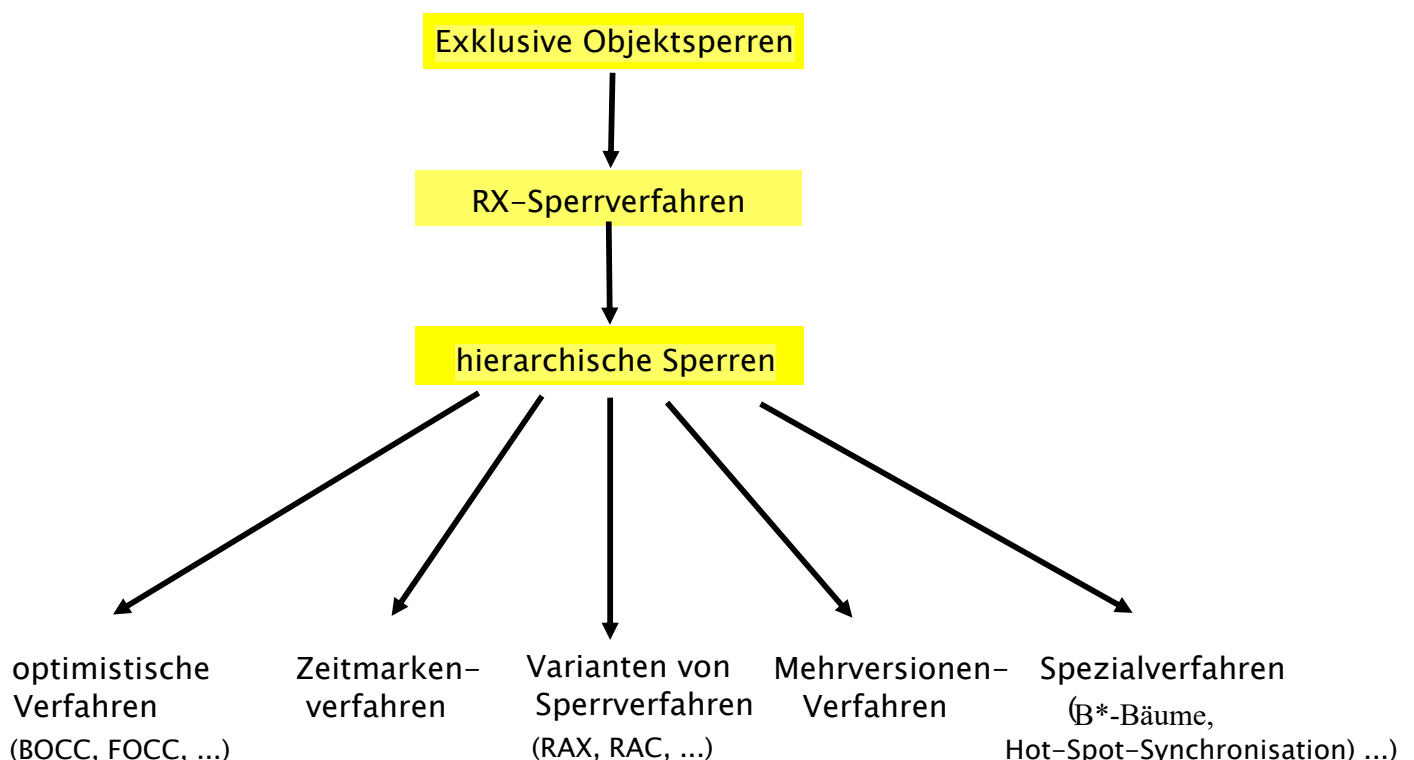
- *reihenfolgeerhaltende Serialisierbarkeit*: jede Transaktion sollte wenigstens alle Änderungen sehen, die bei ihrem Start (BOT) bereits beendet waren
- *chronologieerhaltende Serialisierbarkeit*: jede Transaktion sollte stets die aktuellste Objektversion sehen

Welche Version von x soll T3 lesen?

■ Beispiel



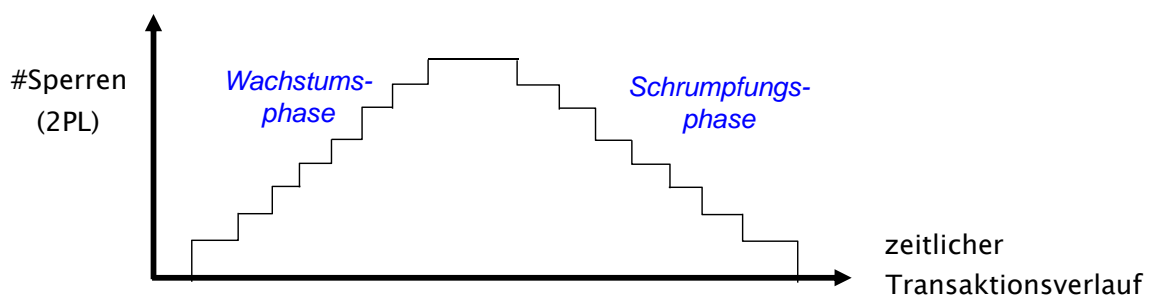
Historische Entwicklung von Synchronisationsverfahren



Zweiphasen-Sperrprotokolle (2 Phase Locking, 2PL)

■ Einhaltung folgender Regeln gewährleistet Serialisierbarkeit:

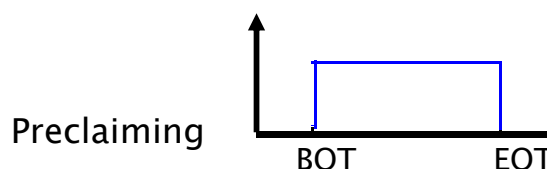
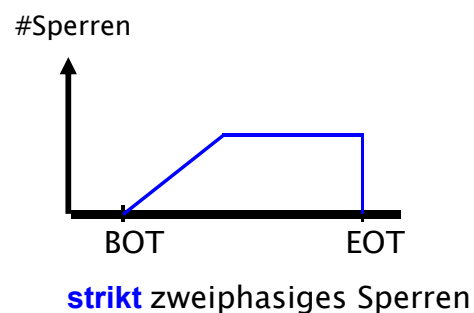
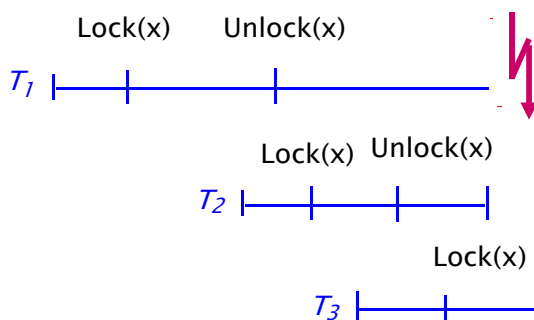
1. vor jedem Objektzugriff muss Sperre mit ausreichendem Modus angefordert werden
2. gesetzte Sperren anderer Transaktionen sind zu beachten
3. eine Transaktion darf nicht mehrere Sperren für ein Objekt anfordern
4. Zweiphasigkeit:
 - Anfordern von Sperren erfolgt in einer *Wachstumsphase*
 - Freigabe der Sperren in *Schrumpfungsphase*
 - Sperrfreigabe kann erst beginnen, wenn alle Sperren gehalten werden
5. spätestens bei EOT sind alle Sperren freizugeben



Striktes Zwei-Phasen-Sperren

- 2PL garantiert Serialisierbarkeit lediglich in fehlerfreier Umgebung
- Fehler während Schrumpfungsphase können zu "Dirty Read" etc. führen
- Lösungsalternativen

- Lesen schmutziger Daten und Abhängigkeiten bei Commit überprüfen (Problem: kaskadierende Rollbacks)
- besser: strikte Zwei-Phasen-Sperrverfahren mit Sperrfreigabe nach Commit (in Commit-Phase 2 eines Zwei-Phasen-Commits)



RX-Sperrverfahren

- Sperranforderung einer Transaktion: **R** (Read) oder **X** (eXclusive bzw. Write)
- gewährter Sperrmodus des Objektes: NL, R, X
- Kompatibilitätsmatrix:

		aktueller Modus		
		NL	R	X
angeforderter Modus	R	+	+	-
	X	+	-	-

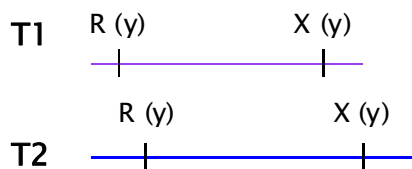
+ verträglich (kompatibel)
- unverträglich

NL (no lock) wird in Kompatibilitätsmatrix meist weggelassen

- unverträgliche Sperranforderung (Sperrkonflikt) führt zur **Blockierung**
 - anfordernde Transaktion muss warten bis Sperre verfügbar wird



Problem von Sperrkonversionen



- Sperrkonversionen führen oft zu Deadlocks
- erweitertes Sperrverfahren
 - Ziel: Verhinderung von Konversions-Deadlocks
 - **U-Sperre (Update)** für Lesen mit Änderungsabsicht
 - bei Änderung Konversion $U \rightarrow X$, andernfalls $U \rightarrow R$ (Downgrading)

		aktueller Modus		
		R	U	X
angeforderter Modus	R	+	-	-
	U	+	-	-
	X	-	-	-

- u.a. in DB2 eingesetzt (**SELECT FOR UPDATE**)
- das Verfahren ist unsymmetrisch - was würde eine Symmetrie bei U bewirken?



Konsistenzstufen von Transaktionen

ursprüngliche Definition von Gray et al. (1976)

- **Konsistenzstufe 0:**
 - Transaktionen halten kurze Schreibsperrern auf den Objekten, die sie ändern
- **Konsistenzstufe 1:**
 - Transaktionen halten lange Schreibsperrern auf den Objekten, die sie ändern (keine Lesesperrern)
- **Konsistenzstufe 2:**
 - Transaktionen halten lange Schreibsperrern auf geänderten Objekten sowie *kurze Lesesperrern* auf Objekten, die sie lesen
- **Konsistenzstufe 3:**
 - Transaktionen halten lange Schreibsperrern auf den Objekten, die sie ändern, sowie lange Lesesperrern auf Objekten, die sie lesen.



Konsistenzebenen im SQL-Standard

- vier Konsistenzebenen (Isolation Level) bzgl. Synchronisation
 - Konsistenzebenen sind durch die Anomalien bestimmt, die jeweils in Kauf genommen werden
 - Lost-Update muss generell vermieden werden
 - Default ist Serialisierbarkeit (serializable)

Konsistenzebene	Anomalie		
	Dirty Read	Non-Repeatable Read	Phantome
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

- **SQL-Anweisung zum Setzen der Konsistenzebene:**
`SET TRANSACTION <tx mode>, ISOLATION LEVEL <level>`
 - tx mode: READ WRITE (Default) bzw. READ ONLY
 - Beispiel: SET TRANSACTON READ ONLY, ISOATION LEVEL READ COMMITTED
- **READ UNCOMMITTED für Änderungstransaktionen unzulässig**



JDBC: Transaktionskontrolle

■ Transaktionskontrolle durch Methodenaufrufe der Klasse Connection

- `setAutoCommit`: Ein-/Abschalten des Autocommit-Modus (jedes Statement ist eigene Transaktion)
 - Default: Autocommit eingeschaltet
- `setReadOnly`: Festlegung ob lesende oder ändernde Transaktion
- `setTransactionIsolation`: Festlegung der Synchronisationsanforderungen (None, Read Uncommitted, Read Committed, Repeatable Read, Serializable)
- `commit` bzw. `rollback`: erfolgreiches Transaktionsende bzw. Transaktionsabbruch

■ Beispiel

```
try {
    con.setAutoCommit (false);
    // einige Änderungsbefehle (z.B. Insert, Update, ...)
    con.commit ();
} catch (SQLException e) {
    try { con.rollback (); } catch (SQLException e2) {}
} finally {
    try { con.setAutoCommit (true); } catch (SQLException e3) {}
}
```

■ Autocommit-Modus kann leicht zu Mehrbenutzeranomalien führen

- inkl. Lost Updates



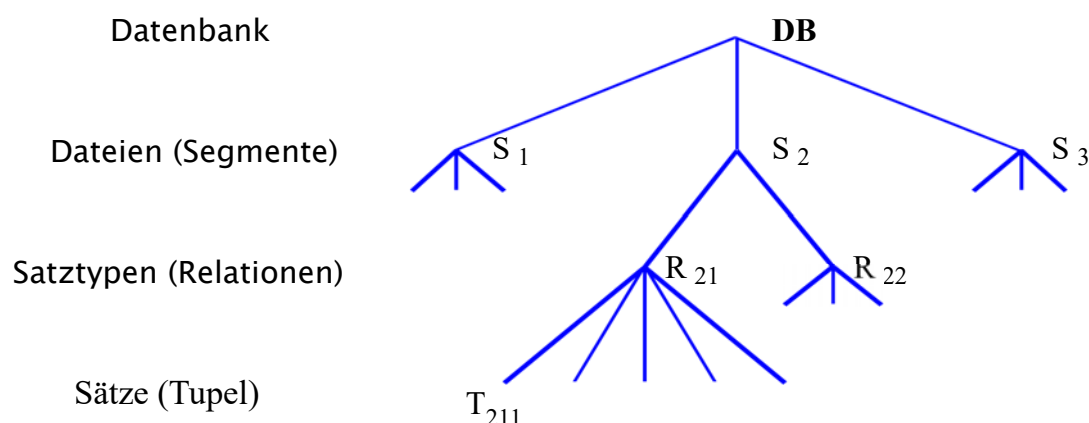
Hierarchische Sperrverfahren

■ Sperrgranulat bestimmt Parallelität/Aufwand

- feines Granulat reduziert Sperrkonflikte
- jedoch sind viele Sperren anzufordern und zu verwalten

■ hierarchische Verfahren erlauben Flexibilität bei Wahl des Granulates ('multigranularity locking')

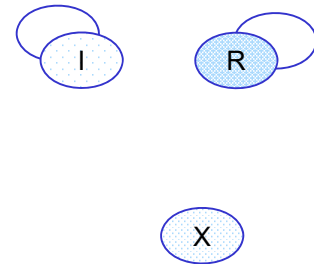
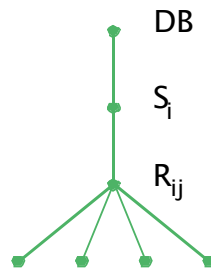
- z.B. lange Transaktionen (Anfragen) auf Relationenebene und kurze Transaktionen auf Satzebene synchronisieren
- kommerzielle DBS unterstützen zumeist mindestens 2-stufige Objekthierarchie, z.B. Segment-Seite bzw. Satztyp (Relation) - Satz (Tupel)



Hierarchische Sperrverfahren: Anwartschaftssperren

- mit R- und X-Sperre werden alle Nachfolgerknoten implizit mitgesperrt => *Einsparungen möglich*
- alle Vorgängerknoten sind ebenfalls zu sperren, um Unverträglichkeiten zu vermeiden => Verwendung von **Anwartschaftssperren** ('intention locks')
- einfachste Lösung: Nutzung eines Sperrtyps (I-Sperre)

	I	R	X
I	+	-	-
R	-	+	-
X	-	-	-

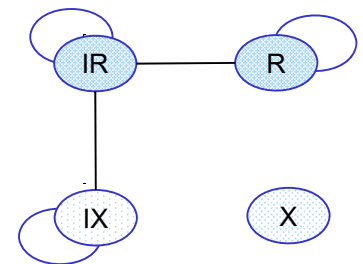
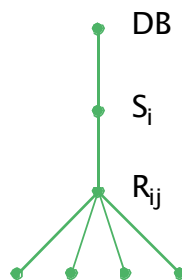


- Unverträglichkeit von I- und R-Sperren zu restriktiv => zwei Arten von Anwartschaftssperren (IR und IX)

Anwartschaftssperren (2)

- Anwartschaftssperren für Leser und Schreiber

	IR	IX	R	X
IR	+	+	+	-
IX	+	+	-	-
R	+	-	+	-
X	-	-	-	-



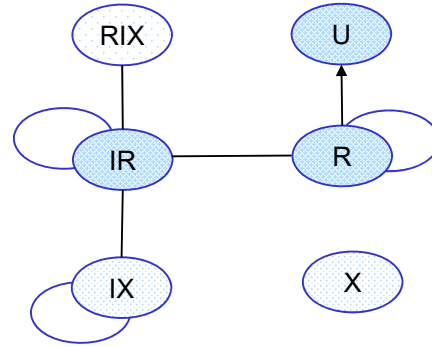
- IR- Sperre (intent read), falls auf untergeordneten Objekten nur lesend zugegriffen wird, sonst IX-Sperre
 - weitere Verfeinerung sinnvoll, für Situationen, in denen alle Tupel eines Satztyps gelesen und nur einige davon geändert werden sollen
 - X-Sperre auf Satztyp sehr restriktiv
 - IX-auf Satztyp verlangt Sperren jedes Tupels
- => **neuer Typ von Anwartschaftssperre: RIX = R + IX**
- nur für zu ändernde Sätze muss (X-)Sperre auf Tupelebene angefordert werden

Anwartschaftssperren (3)

■ vollständiges Protokoll der Anwartschaftssperren

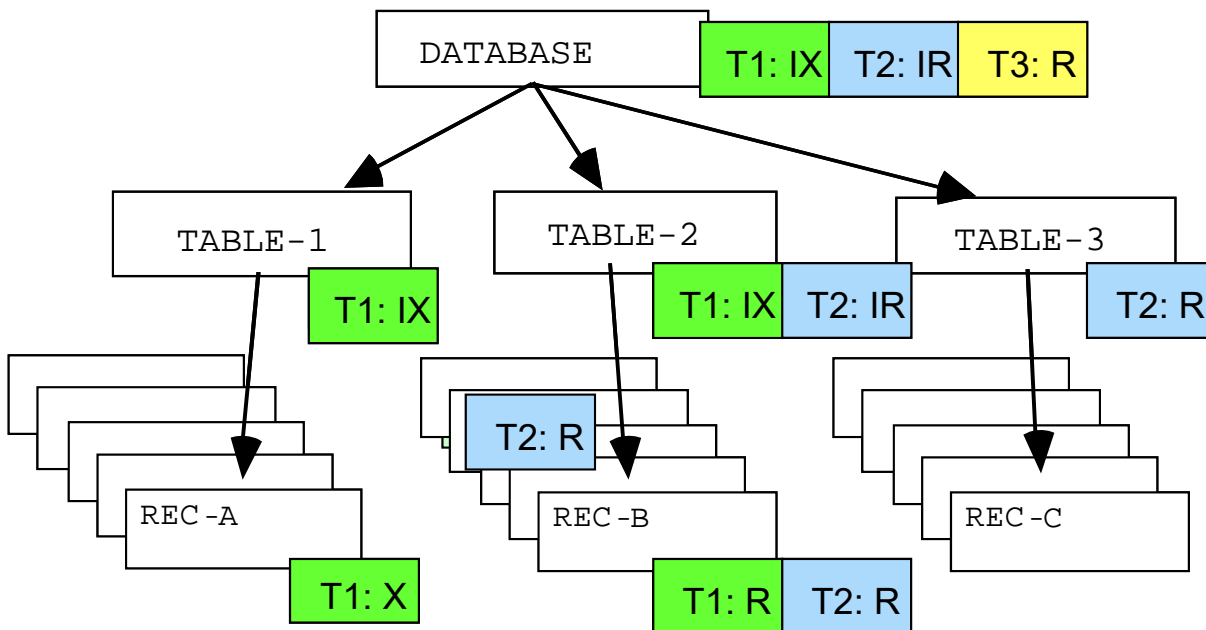
- RIX gibt Leserecht auf den Knoten und seine Nachfolger. Weiterhin ist damit das Recht verbunden, auf Nachfolger-Knoten IX, U und X-Sperren anzufordern.
- U gewährt Leserecht auf den Knoten und seine Nachfolger - repräsentiert die Absicht, den Knoten in der Zukunft zu verändern. Bei Änderung Konversion $U \rightarrow X$, sonst $U \rightarrow R$.

	IR	IX	R	RIX	U	X
IR	+	+	+	+	-	-
IX	+	+	-	-	-	-
R	+	-	+	-	-	-
RIX	+	-	-	-	-	-
U	-	-	+	-	-	-
X	-	-	-	-	-	-



- Sperranforderungen von der Wurzel zu den Blättern
- bei R- oder IR-Anforderung müssen für alle Vorgänger IX- oder IR-Sperren erworben werden
- bei X-, U-, RIX- o. IX-Anforderung müssen alle Vorgänger in RIX oder IX gehalten werden
- Sperrfreigaben von den Blättern zu der Wurzel
- bei EOT sind alle Sperren freizugeben

Hierarchische Sperren: Beispiel



- T3 wartet
- T2 hat Lesesperre auf der gesamten Tabelle 3
- Lesesperren auf Satzebene in Tabelle 2
- T1 hat Satzsperrungen in Tabelle 1 und 2

Wie werden folgende Anforderungen behandelt

- R-Sperranforderung auf REC-D:
- X-Anforderung auf REC-E

Hierarchische Sperren: (De-) Escalation

■ Lock Escalation

- falls Transaktion sehr viele Sperren benötigt -> dynamisches Umschalten auf gröberes Granulat
- Bsp.: nach 200 Satzsperrern auf einer Tabelle -> 1 Tabellensperre erwerben
- Schranke ist typischer Tuning-Parameter

■ manchmal kann *Lock De-Escalation* sinnvoll sein

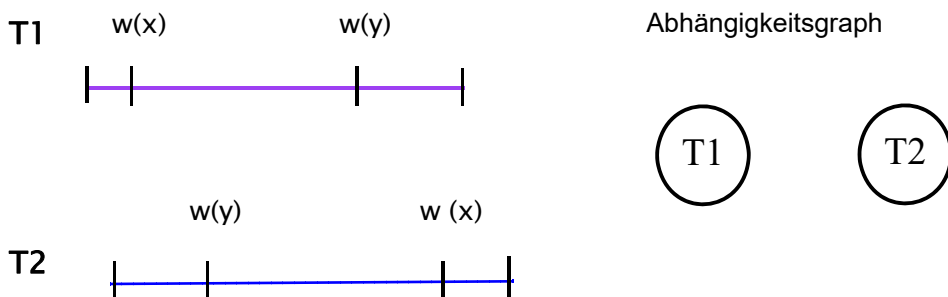
- Erwerbe grob-granulare Sperre (z.B. für Tabelle)
- Vermerke referenzierte Objekte auf fein-granularer Ebene (z.B. Sätze)
- bei Konflikt(en): Umschalten auf feine Sperren

Deadlock-Behandlung

■ 5 Voraussetzungen für Deadlock

1. paralleler Objektzugriff
2. exklusive Zugriffsanforderungen
3. anfordernde Transaktion besitzt bereits Objekte/Sperren
4. keine vorzeitige Freigabe von Objekten/Sperren (non-preemption)
5. zyklische Wartebeziehung zwischen zwei oder mehr Transaktionen

■ Beispiel



- für DBS: Deadlock-Behandlung erfordert Rücksetzung (Rollback) von Transaktionen

Lösungsmöglichkeiten zur Deadlock-Behandlung

1. Timeout-Verfahren

- Transaktion wird nach festgelegter Wartezeit auf Sperre zurückgesetzt
- problematische Bestimmung des Timeout-Wertes
- viele unnötige Rücksetzungen bei kleinem Timeout-Wert
- lange Blockaden bei großem Timeout-Wert

2. Deadlock-Verhütung (Prevention)

- keine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich
- Bsp.: Preclaiming (in DBS i.a. nicht praktikabel)

3. Deadlock-Vermeidung (Avoidance)

- potenzielle Deadlocks werden durch entsprechende Maßnahmen vermieden
- Laufzeitunterstützung nötig

4. Deadlock-Erkennung (Detection)

- Zyklensuche innerhalb eines Wartegraphen
- gestattet minimale Anzahl von Rücksetzungen



Deadlock-Vermeidung

- Zuweisung einer eindeutigen *Transaktionszeitmarke* bei BOT
- im Konfliktfall darf nur ältere (bzw. jüngere) Transaktion warten
=> kein Zyklus möglich

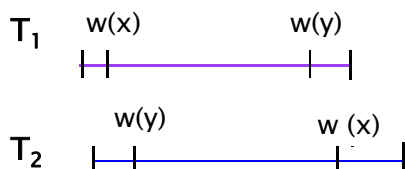
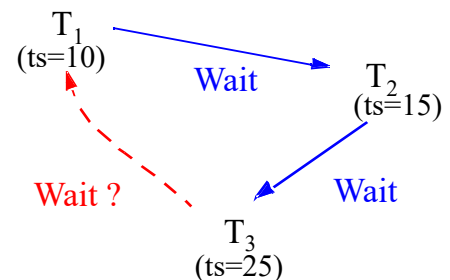
- in Verteilten DBS : Behandlung globaler Deadlocks ohne Kommunikation

■ WAIT/DIE-Verfahren

- anfordernde Transaktion wird zurückgesetzt, falls sie jünger als Sperrbesitzer ist
- ältere Transaktionen warten auf jüngere

T_i fordert Sperre, Konflikt mit T_j

if $ts(T_i) < ts(T_j)$ { T_i älter als T_j }
then WAIT (T_j)
else ROLLBACK (T_i) { "Die" }



Deadlock-Vermeidung (2)

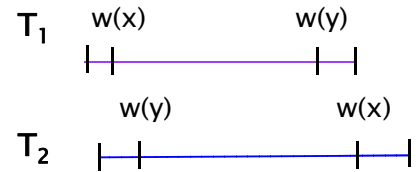
WOUND / WAIT-Verfahren:

- Sperrbesitzer wird zurückgesetzt, wenn er jünger als anfordernde Transaktion ist: jüngere Transaktionen warten auf ältere
- preemptiver Ansatz

T_i fordert Sperre, Konflikt mit T_j :

```

if  $ts(T_i) < ts(T_j)$  {  $T_i$  älter als  $T_j$  }
then ROLLBACK ( $T_j$ ) { "Wound" }
else WAIT ( $T_j$ )
    
```



Verbesserung für Wait/Die und Wound/Wait:

- statt BOT-Zeitmarke Zuweisung der Transaktionszeitmarke erst bei erstem Sperrkonflikt ("dynamische Zeitmarken")
- erster Sperrkonflikt kann stets ohne Rücksetzung behandelt werden



Deadlock-Vermeidung: Wait Depth Limited

"Wartetiefe" (Wait Depth):

- eine nicht-blockierte Transaktion hat Wartetiefe 0
- blockierte Transaktion T hat Wartetiefe $i+1$, falls i die maximale Wartetiefe derjenigen Transaktionen ist, die T blockieren

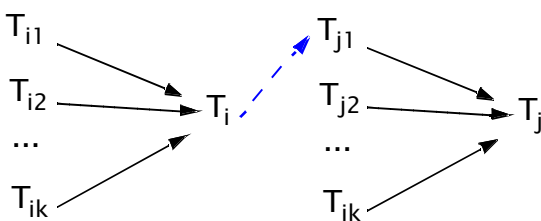
Wait Depth Limited (WDL): Begrenzung der maximalen Wartetiefe d

- $d=0$: kein Warten (Immediate Restart) -> keine Deadlocks möglich
- $d=1$: Warten erfolgt nur auf nicht-blockierte (laufende) Transaktionen -> keine Deadlocks möglich

Problemfälle mit drohender Wartetiefe > 1

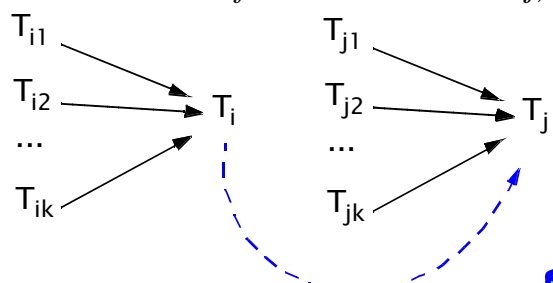
Fall 1:

(Konflikt mit blockierter Transaktion)



Fall 2

(Konflikt von T_i mit laufender Transaktion T_j)



Wait-Depth Limited (2)

■ WDL1-Variante "Running Priority"

T_i fordert Sperre, Konflikt mit T_j :

```
if ( $T_j$  blockiert) then KILL ( $T_j$ ) {Bevorzugung der laufenden Transaktion}
else if ( $T_k$  wartet auf  $T_i$ ) {es existiert ein  $T_k$ , die auf  $T_i$  wartet }
then ROLLBACK( $T_i$ )
else WAIT ( $T_i$ )
```

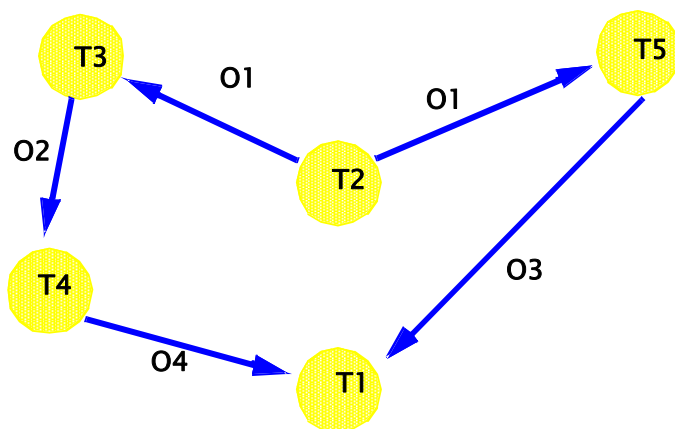


- bei hohen Konflikttraten zeigte WDL in Simulationen besseres Leistungsverhalten als 2PL

Deadlock-Erkennung

- explizites Führen eines *Wartegraphen* (wait-for graph) und Zyklensuche zur Erkennung von Verklemmungen

- $T_1 \rightarrow T_2$: T_1 wartet auf T_2 wegen unverträglicher Sperranforderung
- Wartegraphen enthält nur laufende Transaktionen (im Gegensatz zu Abhängigkeitsgraphen)



- Deadlock-Auflösung durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter Transaktion (z.B. Verursacher oder 'billigste' Transaktion)

- Zyklensuche entweder

- bei jedem Sperrkonflikt bzw.
- verzögert (z.B. über Timeout gesteuert)

Implementierungsaspekte: Datenstrukturen

■ Hash-Tabelle zur Realisierung der Sperrtabelle

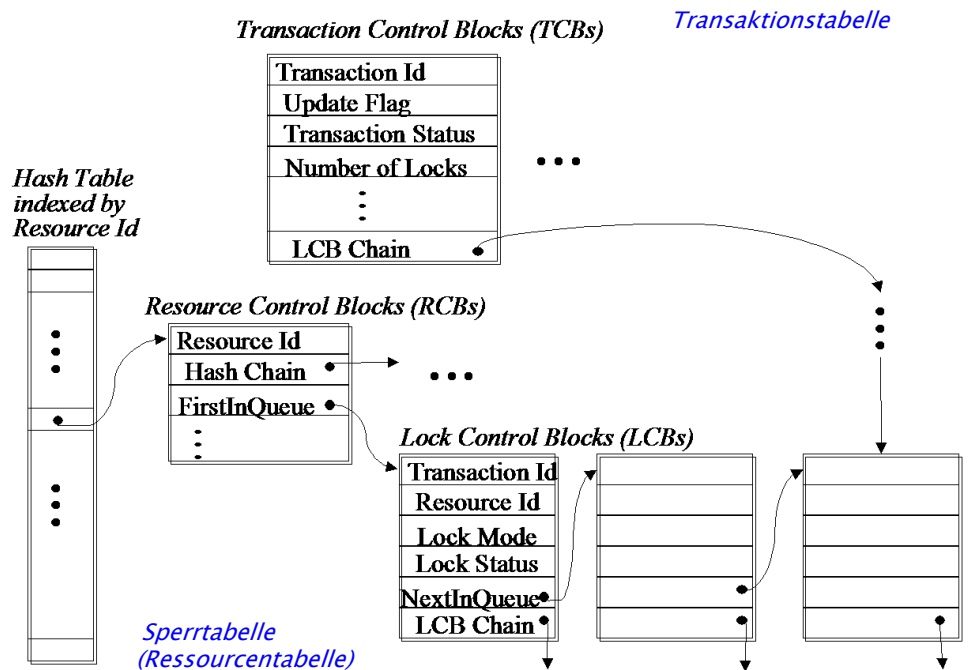
- schneller Zugriff auf Objekt/Ressourcenkontrollblöcke für Lock-Aufrufe

■ Matrixorganisation Objekt-/Transaktions-tabelle

- schnelle Bestimmung freizugebender Sperren bei Commit

■ Kurzzeitsperren („Latch“) für Zugriffe auf Sperrtabelle

- Semaphore pro Hash-Klasse reduziert Konflikt-Gefahr



Sperrverfahren in Datenbanksystemen

■ Sperrverfahren: Vermeidung von Anomalien, in dem

- zu ändernde Objekte dem Zugriff aller anderen Transaktionen entzogen werden
- zu lesende Objekte vor Änderungen geschützt werden

■ Standardverfahren: hierarchisches Zweiphasen-Sperrprotokoll

- mehrere Sperrgranulate
- Verringerung der Anzahl der Sperranforderungen

■ Probleme bei der Implementierung von Sperren

- Zweiphasigkeit der Sperren führt häufig zu langen Wartezeiten (starke Serialisierung)
- häufig benutzte Indexstrukturen können zu Engpässen werden
- Eigenschaften des Schemas können "hot spots" erzeugen

■ Optimierungen:

- Begnügen mit reduzierter Konsistenzstufe
- Verkürzung der Sperrdauer, insbesondere für Änderungen
- Nutzung mehrerer Objektversionen
- spezialisierte Sperren (Nutzung der Semantik von Änderungsoperationen)