

# 7. Synchronisation<sup>1</sup>

- Synchronisation – Überblick über die Verfahren

- Zweiphasen-Sperrprotokolle

- RX-Protokoll, RUX-Protokoll
- Hierarchische Sperrverfahren
- Deadlock-Behandlung

- Einführung von Konsistenzebenen

- Reduzierung von TA-Blockierungen, aber Programmierdisziplin gefordert
- Unterschiedliche Ansätze basierend auf
  - Sperrdauer für R und X
  - zu tolerierende "Fehler"

- Optimistische Synchronisation

- Eigenschaften
- BOCC, FOCC

- Optimierungen

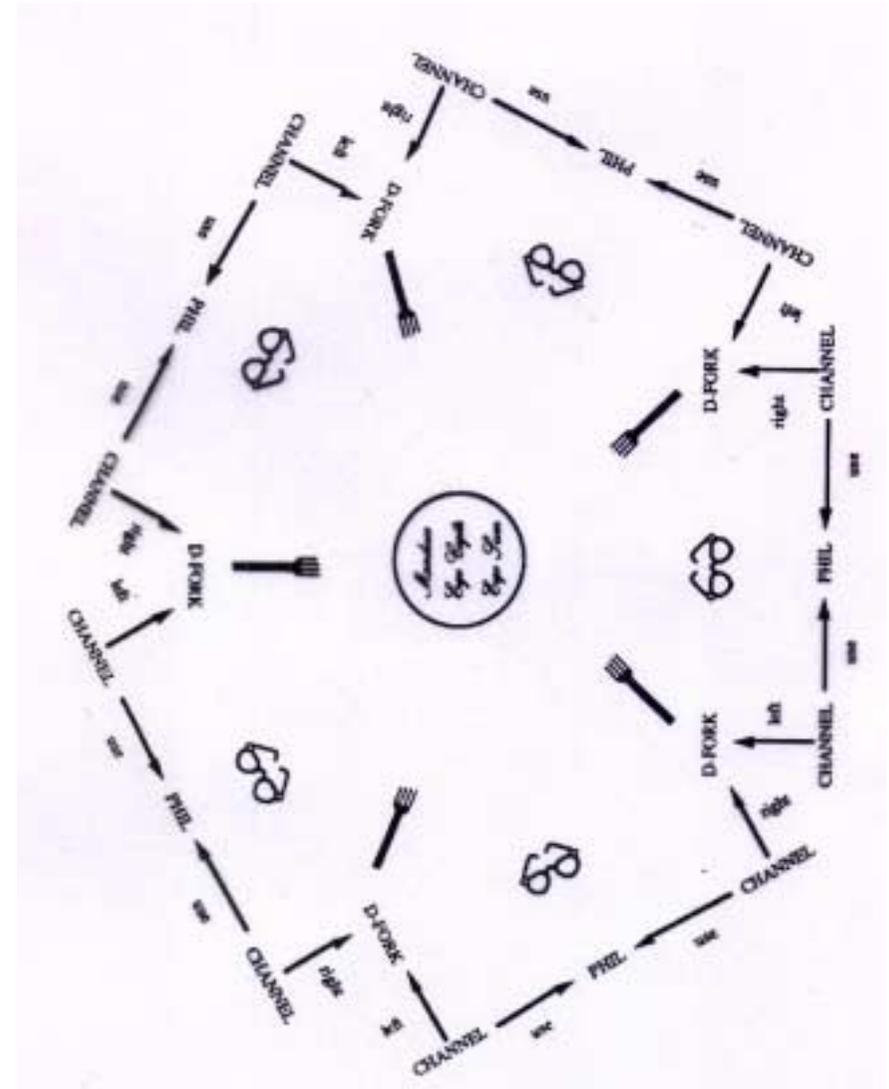
- RAX, RAC, Mehrversionen-Verfahren
- Zeitstempel-Verfahren
- Prädikatssperren
- Synchronisation auf Objekten
- Spezielle Synchronisationsprotokolle

- Leistungsanalyse und Bewertung

- Ergebnisse empirischer Untersuchungen
- Dynamische Lastkontrolle

1. Thomasian, A.: Concurrency Control: Methods, Performance, and Analysis, in: ACM Computing Surveys 30:1, 1998, 70-119.

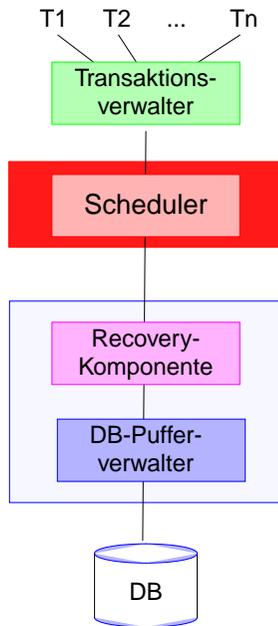
## Dining Philosophers



Wie unterscheidet sich die Synchronisation in DBS?

## Einbettung des DB-Schedulers

- **Stark vereinfachtes Modell**



- **Welche Aufgaben hat der Scheduler?**

- als Komponente der Transaktionsverwaltung zuständig für I von **ACID**
- kontrolliert die beim TA-Ablauf auftretenden Konfliktoperationen (R/W, W/R, W/W) und garantiert insbesondere, dass nur „serialisierbare“ TA erfolgreich beendet werden
- verhindert nicht serialisierbare TA.  
Dazu ist eine Kooperation mit der Recovery-Komponente erforderlich (Rücksetzen von TA)

➔ **garantiert „vernünftige“ Schedules:**

## Synchronisationsverfahren

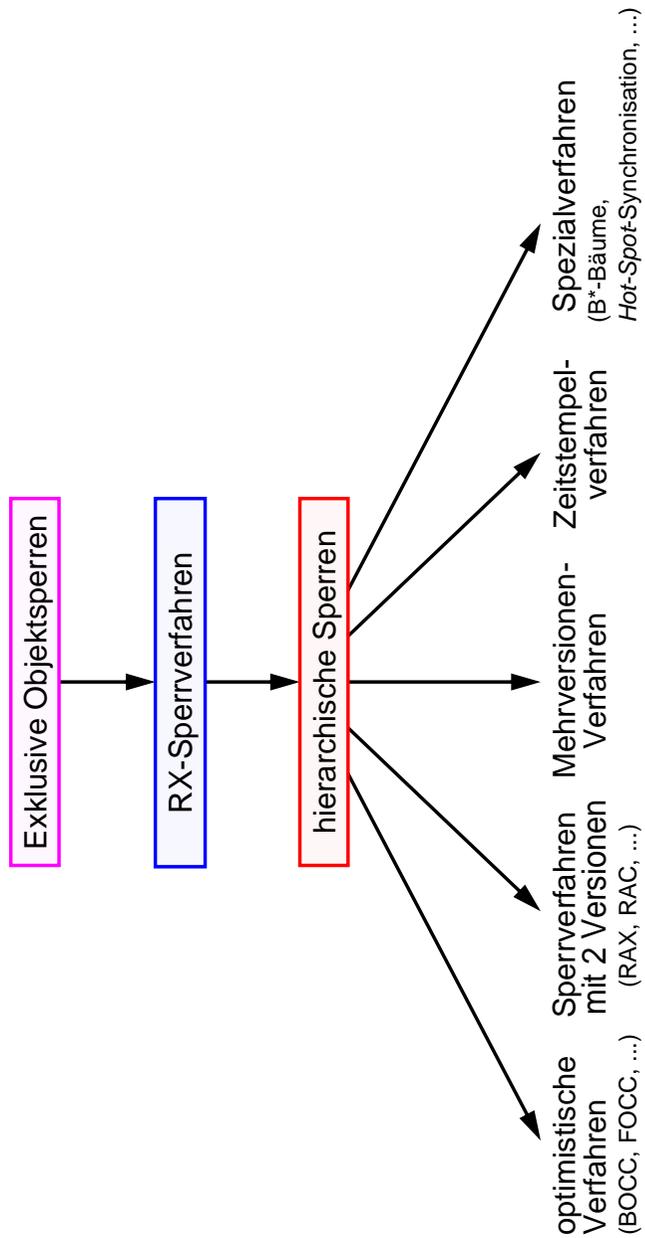
- **Zur Realisierung der Synchronisation gibt es viele Verfahren**

- **Pessimistische Verfahren:** Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt.
  - **Optimistische Verfahren:** Erst bei Commit wird überprüft, ob die TA serialisierbar ist
  - **Versionsverfahren:** Keine Behinderung der Leser durch Schreiber
  - **Zeitstempelverfahren:** Überprüfung der Serialisierbarkeit am Objekt
  - **Prädikatssperren:** Es wird die Menge der möglichen Objekte, die das Prädikat erfüllen, gesperrt
  - **Spezielle Synchronisationsverfahren:** Nutzung der Semantik von Änderungen
  - ...
- ➔ **Sperrverfahren sind pessimistisch und universell einsetzbar.**

- **Sperrbasierte Synchronisation**

- Sperren stellen während des laufenden Betriebs sicher, dass die resultierende Historie serialisierbar bleibt
- Es gibt mehrere Varianten

## Historische Entwicklung von Synchronisationsverfahren



7 - 5

## RX-Sperrverfahren

- **Sperrmodi**

- Sperrmodus des Objektes: NL (no lock), R (read), X (exclusive)
- Sperranforderung einer Transaktion: R, X

- **Kompatibilitätsmatrix:**

		aktueller Modus des Objekts		
		NL	R	X
angeforderter Modus der TA	R	+	+	-
	X	+	-	-

- Falls Sperre nicht gewährt werden kann, muss die anfordernde TA warten, bis das Objekt freigegeben wird (Commit/Abort der die Sperre besitzenden TA)
- Wartebeziehungen werden in einem *Wait-for-Graph* (WfG) verwaltet

- **Ablauf von Transaktionen**

T1	T2	a	b	Bem.
		NL	NL	
lock (a, X)		X		
...				
	lock (b, R)		R	
	...			
lock (b, R)			R	
	lock (a, R)	X		T2 wartet, WfG:
...				
unlock (a)		NL --> R		T2 wecken
...	...			
unlock(b)			R	

7 - 6

## Zweiphasen-Sperrprotokolle<sup>2</sup>

- Einhaltung folgender Regeln gewährleistet Serialisierbarkeit:

- Vor jedem Objektzugriff muss Sperre mit ausreichendem Modus angefordert werden
- Gesetzte Sperren anderer TA sind zu beachten
- Eine TA 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 benötigten Sperren gehalten werden
- Spätestens bei Commit sind alle Sperren freizugeben

- Beispiel für ein 2PL-Protokoll (2PL: two-phase locking)

```

BOT
lock (a, X)
...
lock (b, R)
...
lock (c, X)
...
unlock (b)
unlock (c)
unlock (a)
Commit
    
```

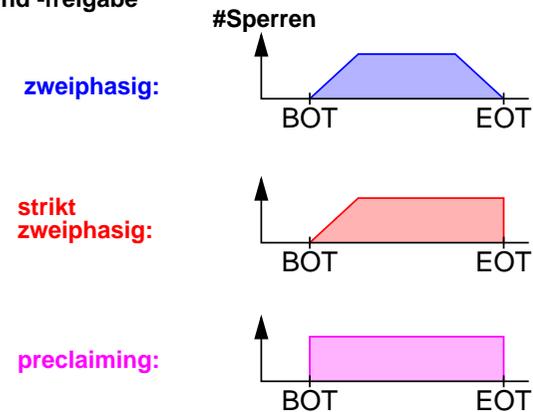
An der SQL-Schnittstelle ist die Sperranforderung und -freigabe nicht sichtbar!

2. Eswaran, K.P. et al.: The notions of consistency and predicate locks in a data base system, in: Comm. ACM 19:11, 1976, 624-633

## Zweiphasen-Sperrprotokolle (2)

- Formen der Zweiphasigkeit

Sperranforderung und -freigabe



- Anwendung des 2PL-Protokolls

T1	T2	Bem.
BOT		
lock (a, X)		
...		
lock (b, R)		
...		
lock (c, X)		
...		
unlock (b)	BOT	
unlock (c)	lock (a, X)	T2 wartet: WfG
unlock (a)		
	read (a)	T2 wecken
	write (a)	
	unlock (a)	
	commit	
unlock (b)		

➔ Praktischer Einsatz erfordert **striktes 2PL-Protokoll!**

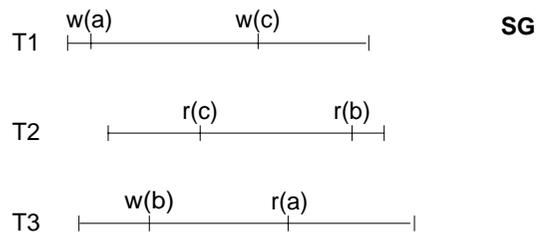
## Verklemmungen (Deadlocks)

- **Striktes 2PL-Protokoll**

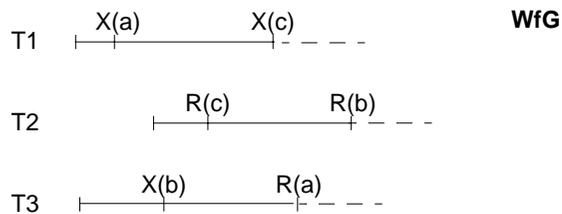
- gibt alle Sperren erst bei Commit frei und
- verhindert dadurch kaskadierendes Rücksetzen

➔ Auftreten von Verklemmungen ist **inhärent** und kann bei pessimistischen Methoden (blockierende Verfahren) nicht vermieden werden.

- **Nicht-serialisierbare Historie**



• **RX-Verfahren verhindert** das Auftreten einer nicht-serialisierbaren Historie, **aber nicht (immer) Deadlocks**

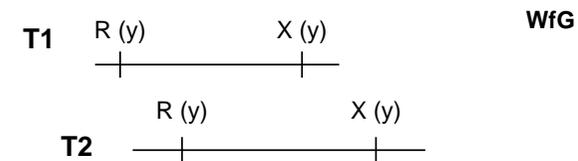


## RUX-Sperrverfahren

- **Forderung**

- Wahl des gemäß der Operation schwächst möglichen Sperrmodus
- Möglichkeit der Sperrkonversion (*upgrade*), falls stärkerer Sperrmodus erforderlich
- Anwendung: viele Objekte sind zu lesen, aber nur wenige zu aktualisieren

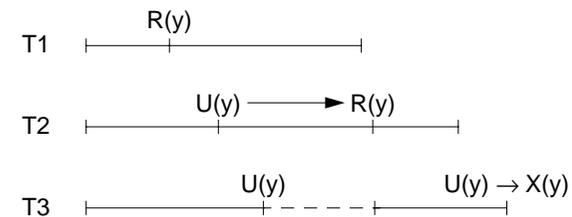
- **Problem: Sperrkonversionen**



- **Erweitertes Sperrverfahren:**

- Ziel: Verhinderung von Konversions-Deadlocks
- U-Sperre für Lesen mit Änderungsabsicht (Prüfmodus)
- bei Änderung Konversion  $U \rightarrow X$ , andernfalls  $U \rightarrow R$  (*downgrade*)

- **Wirkungsweise**



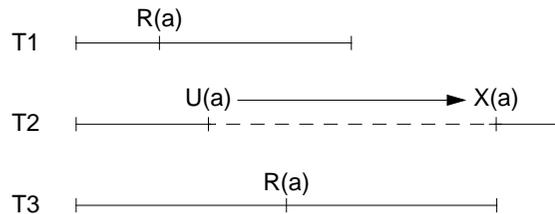
## RUX-Sperrverfahren (2)

- **Symmetrische Variante**

- Was bewirkt eine Symmetrie bei U?

	R	U	X
R	+	+	-
U	+	-	-
X	-	-	-

- **Beispiel**

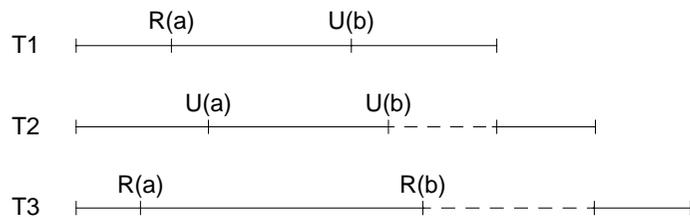


- **Unsymmetrie bei U**

	R	U	X
R	+	-	-
U	+	-	-
X	-	-	-

- u. a. in DB2 eingesetzt

- **Beispiel**



## Hierarchische Sperrverfahren

- **Sperrgranulat bestimmt Parallelität/Aufwand:**

Feines Granulat reduziert Sperrkonflikte, jedoch sind viele Sperren anzufordern und zu verwalten

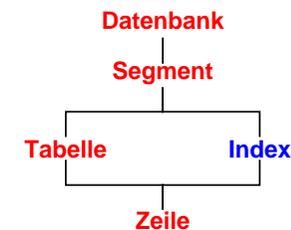
- **Hierarchische Verfahren erlauben Flexibilität bei Wahl des Granulates (*multigranularity locking*), z. B. Synchronisation**

- langer TA auf Tabellenebene
- kurzer TA auf Zeilenebene

- Kommerzielle DBS unterstützen zumeist mindestens zweistufige Objekthierarchie, z. B.

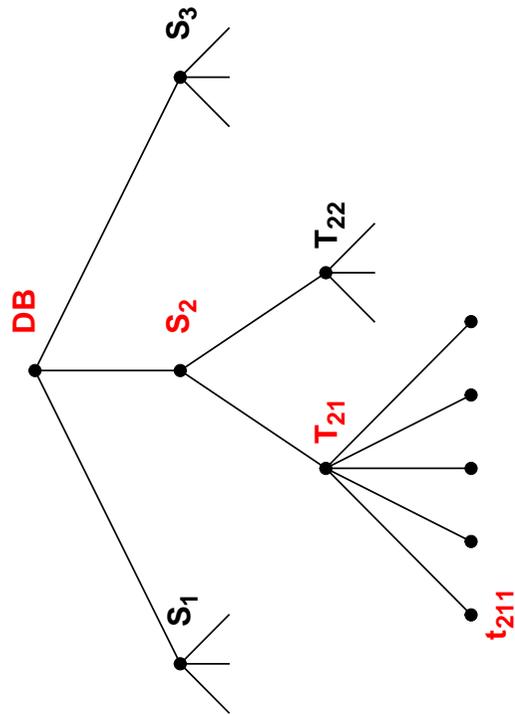


- Verfahren nicht auf reine Hierarchien beschränkt, sondern auch auf halbgeordnete Objektgruppen erweiterbar (siehe auch objektorientierte DBS).



- Verfahren erheblich komplexer als einfache Sperrverfahren (mehr Sperrmodi, Konversionen, Deadlock-Behandlung, ...)

## Beispiel einer Sperrhierarchie



Datenbank

Dateien (Segmente)

Satztypen (Tabellen)

Sätze (Zeilen)

Wieviel Aufwand zum Sperren von

- 1 Satz

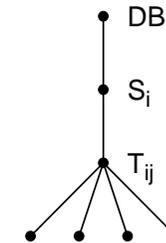
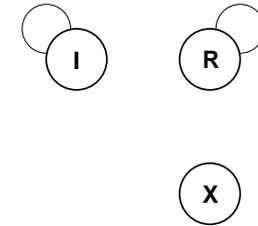
- k Sätzen

- 1 Satztyp

## 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')
- Allgemeine Anwartschaftssperre (I-Sperre)

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



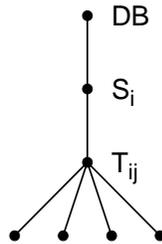
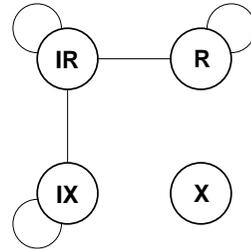
- Unverträglichkeit von I- und R-Sperren: zu restriktiv!

↳ 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, um den Fall zu unterstützen, wo alle Sätze eines Satztyps gelesen und nur einige davon geändert werden sollen

- X-Sperre auf Satztyp sehr restriktiv
- IX-Sperre auf Satztyp verlangt Sperren jedes Satzes

➔ neuer Typ von Anwartschaftssperre: **RIX = R + IX**

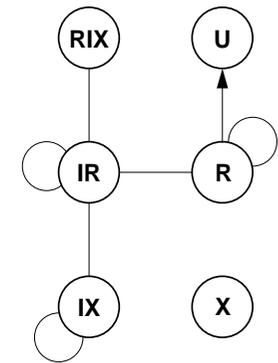
- sperrt das Objekt in R-Modus und verlangt
- X-Sperren auf tieferer Hierarchieebene nur für zu ändernde Objekte

## Anwartschaftssperren (3)

- Vollständiges Protokoll der Anwartschaftssperren

- RIX gibt ein Leserecht auf den Knoten und seine Nachfolger. Weiterhin ist damit das Recht verbunden, auf Nachfolger-Knoten IX, U und X-Sperren anzufordern.
- U gewährt ein Leserecht auf den Knoten und seine Nachfolger. Dieser Modus 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	-	-	-	-	-	-



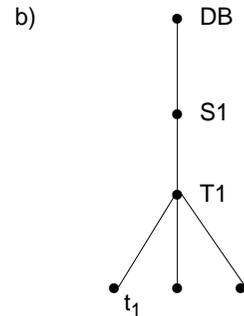
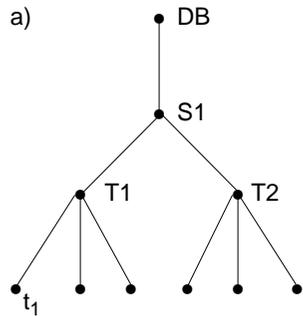
- 'Sperrdisziplin' erforderlich

- Sperranforderungen von der Wurzel zu den Blättern
- Bevor T eine R- oder IR-Sperre für einen Knoten anfordert, muss sie für alle Vorgängerknoten IX- oder IR-Sperren besitzen
- Bei einer X-, U-, RIX- oder IX-Anforderung müssen alle Vorgängerknoten in RIX oder IX gehalten werden
- Sperrfreigaben von den Blättern zu der Wurzel
- Bei EOT sind alle Sperren freizugeben

## Hierarchische Sperrverfahren: Beispiele

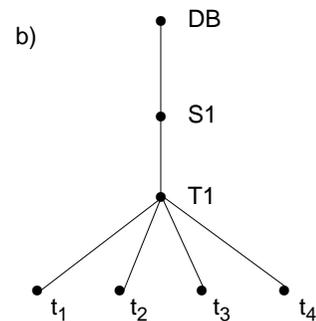
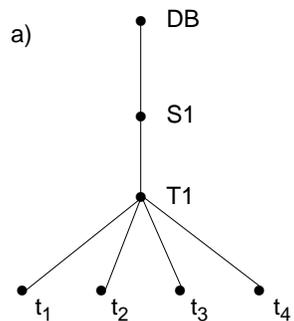
### IR- und IX-Modus

- TA1 liest  $t_1$  in T1
- a) TA2 ändert Zeile in T2
- b) TA3 liest T1



### RIX-Modus

- TA1 liest alle Zeilen von T1 und ändert  $t_3$
- a) TA2 liest T1
- b) TA3 liest  $t_2$  in T1

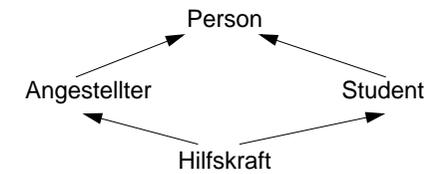


## Hierarchische Sperren in OODBS

### Übertragung der Idee hierarchischer Sperren auf Klassenhierarchie

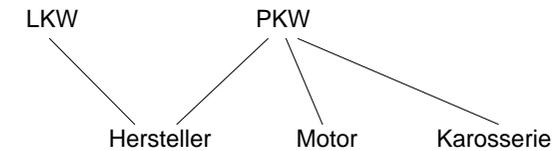
- Einsatz von Anwartschaftssperren
- Reduktion des Sperraufwandes innerhalb von Generalisierungs- und Aggregationshierarchien

### Generalisierungshierarchie



➔ Probleme durch Mehrfachvererbung

### Aggregationshierarchie



➔ Probleme durch gemeinsam genutzte Komponentenobjekte

- Explizites Sperren aller Teilobjekte sehr aufwendig!

## Deadlock-Behandlung

### • Voraussetzungen für Deadlock:

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

### • Lösungsmöglichkeiten:

#### 1. Timeout-Verfahren

- TA wird nach festgelegter Wartezeit auf Sperre zurückgesetzt
- problematische Bestimmung des Timeout-Wertes

#### 2. Deadlock-Verhütung (*Prevention*)

- keine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich
- Beispiel: *Preclaiming* (in DBS i. allg. nicht praktikabel)

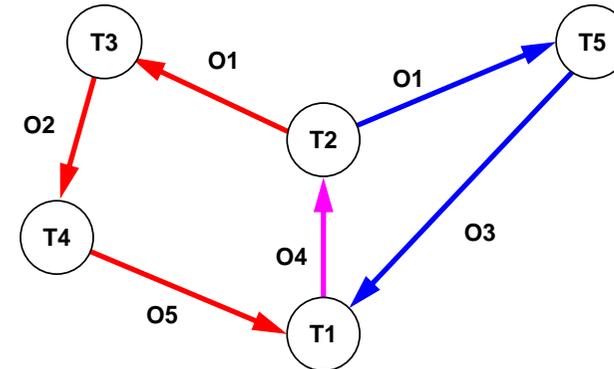
#### 3. Deadlock-Vermeidung (*Avoidance*)

- Potentielle Deadlocks werden im voraus erkannt und durch entsprechende Maßnahmen vermieden
- ⇒ Laufzeitunterstützung nötig

#### 4. Deadlock-Erkennung (*Detection*)

## Deadlock-Erkennung

- Explizites Führen eines **Wartegraphen** (*Wait-For-Graph*) und **Zyklensuche** zur Erkennung von Verklemmungen



- **Deadlock-Auflösung** durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA  
(z. B. Verursacher oder „billigste“ TA zurücksetzen)

#### • Zyklensuche entweder

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

## Sperrverfahren in Datenbanksystemen

- **Aufgabe von Sperrverfahren:** Vermeidung von Anomalien, indem man
  - zu ändernde Objekte dem Zugriff aller anderen Transaktionen entzieht
  - zu lesende Objekte vor Änderungen schützt
- **Standardverfahren:** Hierarchisches Zweiphasen-Sperrprotokoll
  - mehrere Sperrgranulate
  - Verringerung der Anzahl der Sperranforderungen
- **Häufig beobachtete Probleme** bei Sperren
  - Zweiphasigkeit führt häufig zu langen Wartezeiten (starke Serialisierung)
  - **Um Durchsatzziel zu erreichen:**  
**mehr** aktive TA → **mehr** gesperrte Objekte → **höhere** Konflikt-WS → **längere** Sperrwartezeiten, **höhere** Deadlock-Raten → **noch mehr** aktive TA
  - Häufig berührte Objekte können zu Engpässen (*Hot Spots*) werden
  - Eigenschaften des Schemas können *High-Traffic-Objekte* erzeugen
- **Einführung von Konsistenzebenen**  
zur Reduktion des Blockierungspotentials: **Programmierdisziplin gefordert!**
- **Optimierungen!?**
  - Optimistische Verfahren: Verzicht auf Sperren, dafür Validierung
  - Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
  - Nutzung mehrerer Objektversionen
  - Zeitstempelverfahren (lokale Prüfung, vor allem im verteilten Fall)
  - Prädikatssperren, Präzisionssperren
  - spezialisierte Sperren

## Konsistenzebenen

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen
  - „Schwächere“ Konsistenzebene bei der Synchronisation von Leseoperationen erlaubt höhere Parallelitätsgrade und Reduktion von Blockierungen, erfordert aber Programmierdisziplin!
    - ➔ **Inkaufnahme von Anomalien reduziert die TA-Behinderungen**

- **Konsistenzebenen** (basierend auf verkürzte Sperrdauern)

**Ebene 3:** Transaktion T sieht Konsistenzebene 3, wenn gilt:

- a) T verändert keine schmutzigen Daten anderer Transaktionen
- b) T gibt keine Änderungen vor EOT frei
- c) T liest keine schmutzigen Daten anderer Transaktionen
- d) Von T gelesene Daten werden durch andere Transaktionen erst nach EOT von T verändert

**Ebene 2:** Transaktion T sieht Konsistenzebene 2, wenn sie die Bedingungen a, b und c erfüllt

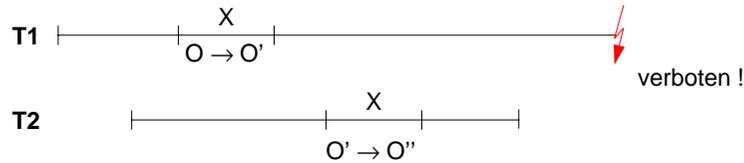
**Ebene 1:** Transaktion T sieht Konsistenzebene 1, wenn sie die Bedingungen a und b erfüllt

**Ebene 0:** Transaktion T sieht Konsistenzebene 0, wenn sie nur Bedingung a erfüllt

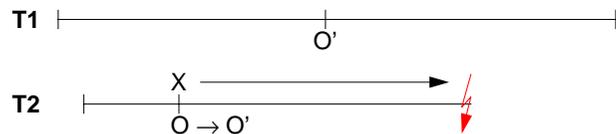
## Konsistenzebenen (2)

- **RX-Sperrverfahren und Konsistenzebenen:**  
(Beispiele für nur ein Objekt O)

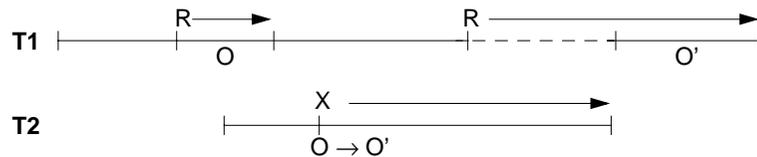
### KE0: kurze X, keine R



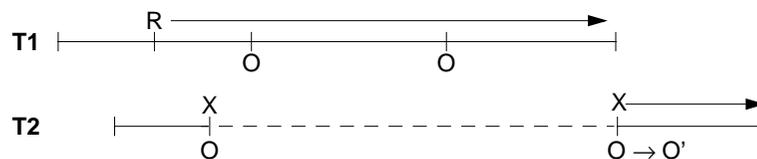
### KE1: lange X, keine R



### KE2: lange X, kurze R



### KE3: lange X, lange R



## Konsistenzebenen (3)

- **Konsistenzebene 3** (eigentlich KE 2,99):
  - wünschenswert, jedoch oft viele Sperrkonflikte wegen langer Schreib- und Lesesperren

### • Konsistenzebene 2:

- nur lange Schreibsperrern, jedoch kurze Lesesperren
- 'unrepeatable read' möglich

### • Konsistenzebene 1:

- lange Schreibsperrern, keine Lesesperren
- 'dirty read' (und 'lost update') möglich

### • Konsistenzebene 0:

- kurze Schreibsperrern ('Chaos')

➔ Kommerzielle DBS empfehlen meist Konsistenzebene 2

### • Wahlangebot

Einige DBS (DB2, Tandem NonStop SQL, ...) bieten Wahlmöglichkeit zwischen:

- 'repeatable read' (KE 3) und
- 'cursor stability' (KE 2)

Einige DBS bieten auch *BROWSE-Funktion*, d. h. Lesen ohne Setzen von Sperren (KE 1)

## Konsistenzebenen (4)

- SQL erlaubt Wahl zwischen **vier Konsistenzebenen** (Isolation Level)
- **Konsistenzebenen sind durch die Anomalien bestimmt**, die jeweils in Kauf genommen werden:
  - Abgeschwächte Konsistenzanforderungen betreffen nur Leseoperationen!
  - **Lost Update** muss generell vermieden werden, d. h., W/W-Abhängigkeiten müssen stets beachtet werden

Konsistenz- ebene	Anomalie		
	Dirty Read	Non-Repeatable Read	Phantome Read
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

- Default ist **Serialisierbarkeit**<sup>3</sup> (serializable)

## Konsistenzebenen (5)

- **SQL-Anweisung zum Setzen der Konsistenzebene:**

```
SET TRANSACTION [mode] [ISOLATION LEVEL level]
```

- Transaktionsmodus: READ WRITE (Default) bzw. READ ONLY
- **Beispiel:**  
SET TRANSACTION READ ONLY, ISOLATION LEVEL READ COMMITTED
- READ UNCOMMITTED für Änderungstransaktionen unzulässig

- **Was ist der Unterschied zwischen KE3 und "Serializable"?**

- **Repeatable Read**

Sperren von vorhandenen Objekten

Datenstruktur: O1                      O2 . .                      Zugriff mit Get Next

- **Serializable**

garantiert Abwesenheit von Phantomen

3. Repeatable Read entspricht Read Stability und Serializable entspricht Repeatable Read in DB2

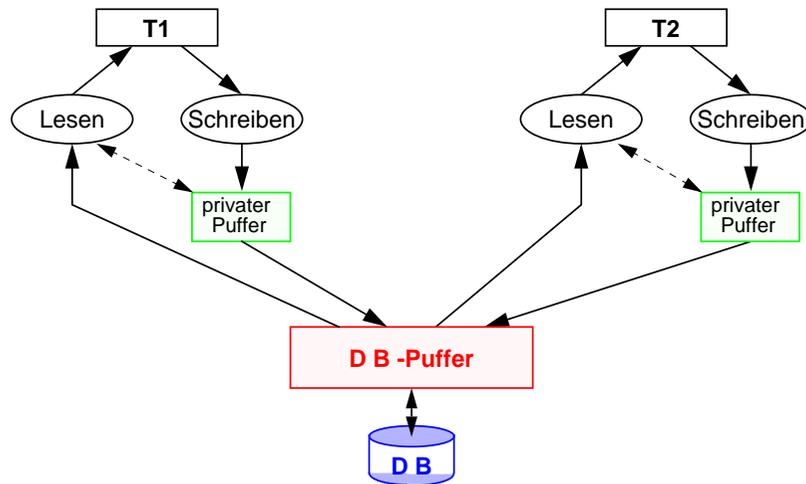
## Optimistische Synchronisation (OCC)

- Dreiphasige Verarbeitung:



- Lesephase

- eigentliche TA-Verarbeitung
- Änderungen einer Transaktion werden in privatem Puffer durchgeführt



- Validierungsphase

- Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer der parallel ablaufenden Transaktionen passiert ist
- Konfliktauflösung durch Zurücksetzen von Transaktionen

- Schreibphase

- nur bei positiver Validierung
- Lese-Transaktion ist ohne Zusatzaufwand beendet
- Schreib-Transaktion schreibt hinreichende Log-Information und propagiert ihre Änderungen

## Optimistische Synchronisation (2)

- Grundannahme: geringe Konfliktwahrscheinlichkeit

- Allgemeine Eigenschaften von OCC:

- + einfache TA-Rücksetzung
- + keine Deadlocks
- + potentiell höhere Parallelität als bei Sperrverfahren
- mehr Rücksetzungen als bei Sperrverfahren
- Gefahr des „Verhungerns“ von TA

- Durchführung der Validierungen:

Pro Transaktion werden geführt

- Read-Set (RS) und
- Write-Set (WS)

- Forderung:

Eine TA kann nur erfolgreich validieren, wenn sie alle Änderungen von zuvor validierten TA gesehen hat

➔ Validierungsreihenfolge bestimmt Serialisierbarkeitsreihenfolge

- Validierungsstrategien:

- **Backward Oriented (BOCC):**

Validierung gegenüber bereits beendeten (Änderungs-) TA

- **Forward Oriented (FOCC):**

Validierung gegenüber laufenden TA

## BOCC

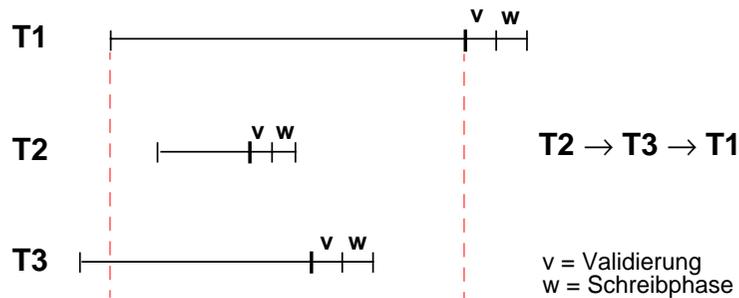
## BOCC (2)

- Erstes publizierte Verfahren zur optimistischen Synchronisation<sup>4</sup>

- **Validierung von Transaktion T:**

BOCC-Test gegenüber allen Änderungs-TA  $T_j$ , die seit BOT von T erfolgreich validiert haben:

IF  $RS(T) \cap WS(T_j) \neq \emptyset$  THEN ABORT T  
 ELSE SCHREIBPHASE

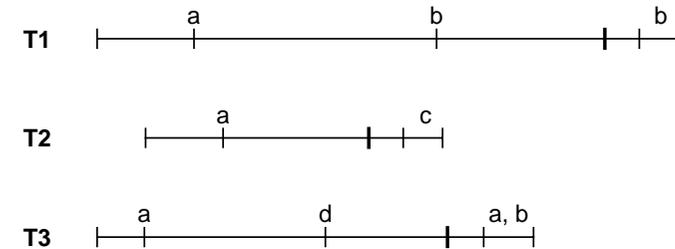


- **Nachteile/Probleme:**

- unnötige Rücksetzungen wegen ungenauer Konfliktanalyse
- Aufbewahren der Write-Sets beendeter TA erforderlich
- hohe Anzahl von Vergleichen bei Validierung
- Rücksetzung erst bei EOT → viel unnötige Arbeit
- Nur die validierende TA kann zurückgesetzt werden  
 → Gefahr von 'starvation'
- hohes Rücksetzrisiko für lange TA und bei Hot Spots

4. Kung, H.T., Robinson, J.T.: On optimistic method for concurrency control, in: ACM Trans. on Database Systems 6:2, 1981, 213-226

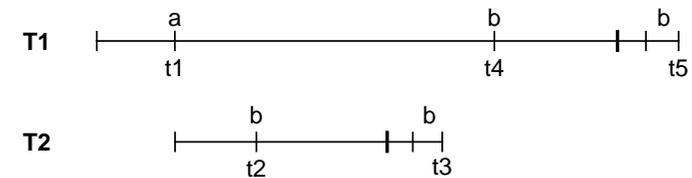
- **Ablaufbeispiel**



- **Validierung von T1:**

1.  $RS(T1) \cap WS(T2) =$
2.  $R1(T1) \cap WS(T3) =$

- **Optimierung von BOCC**



Zeitstempel in WS (Schreibzeitpunkt) und in RS (erste Referenz)

- **Validierung von T1:**

T1 : RS

T2 : WS

$RS(T1) \cap WS(T2) =$

Zusätzliche Prüfung

Write(T2) =

Ref(T1) =

## BOCC+

### • Variation des Verfahrens

- Konflikterkennung über Zeitstempel (Änderungszähler) statt Mengenvergleich
- erfolgreich validierte TA erhalten eindeutige, monoton wachsende TA-Nummer
- geänderte Objekte erhalten TA-Nummer der ändernden TA als Zeitstempel TS zugeordnet
- beim Lesen eines Objektes wird Zeitstempel  $ts$  der gesehenen Version im Read-Set vermerkt
- **Validierung** überprüft, ob gesehene Objektversionen zum Validierungszeitpunkt noch aktuell sind:

```
VALID := true
<< forall r in RS (T) do;
    if ts (r,T) < TS (r) then VALID := false;
end;
if VALID then do;
    TNC := TNC + 1; {ergibt TA-Nummer für T}
    for all w in WS (T) do;
        TS (w) := TNC;
        setze alle laufenden TA mit w in RS zurück;
    end; >>
    Schreibphase für T;
end;
else (setze T zurück);
```

- Zeitstempel TS für geänderte Objekte können zur Durchführung der Validierungen in **Objekttabelle** geführt werden

## BOCC+ (2)

### • Vorteile

- Zum Scheitern verurteilte TA können sofort zurückgesetzt werden
- keine unnötigen Rücksetzungen
- sehr schnelle Validierung

### • Probleme:

- wie bei BOCC ist 'starvation' einzelner TA möglich
- potentiell hohe Rücksetzrate

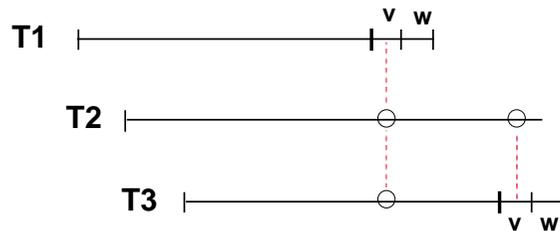
### • Lösungsmöglichkeiten:

- Reduzierung der Konfliktwahrscheinlichkeit, z. B. durch
  - geringere Konsistenzebene (Lese-TA werden bei Validierung nicht mehr berücksichtigt)
  - Mehrversionen-Verfahren
- Kombination mit Sperrverfahren

## FOCC<sup>5</sup>

- Nur Änderungs-TA validieren gegenüber laufenden TA  $T_i$

- Validierungstest:**  $WS(T) \cap RS(T_i) \stackrel{!}{=} \emptyset$



- Vorteile:**

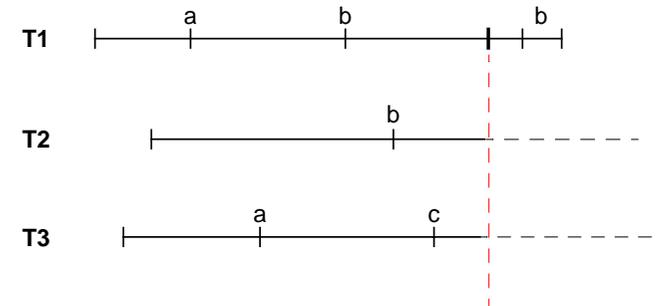
- Wahlmöglichkeit des Opfers (Kill, Abort, Prioritäten, ...)
- keine unnötigen Rücksetzungen
- frühzeitige Rücksetzung möglich
  - ➔ **Einsparen unnötiger Arbeit**
- keine Aufbewahrung von Write-Sets, geringerer Validierungsaufwand als bei BOCC

- Probleme:**

- Während Validierungs- und Schreibphase müssen die Objekte von WS (T) „gesperrt“ sein, damit sich die zu prüfenden RS ( $T_i$ ) nicht ändern (keine Deadlocks damit möglich)
- immer noch hohe Rücksetzrate möglich
- Es kann immer nur einer TA Durchkommen **definitiv zugesichert** werden

## FOCC (2)

- Ablaufbeispiel**



- ➔ **Validierung von T1:**

- $WS(T1) \cap RS(T2) =$
- $WS(T1) \cap RS(T3) =$

- Mögliche Lösungen**

5. Härder, T.: Observations on optimistic concurrency control schemes, Information Systems 9:2, 1984, 111-120

## Kombination von OCC und Sperrverfahren

- **Ziel: Vorteile beider Verfahrensklassen kombinieren**
  - geringe Rücksetzhäufigkeit von Sperrverfahren
  - hohe Parallelität (weniger Sperrwartezeiten) von OCC
- Kombination kann auf verschiedenen Ebenen realisiert werden

### 1. TA-Ebene:

- optimistisch und pessimistisch synchronisierte TA
- für lange TA, die bereits gescheitert waren, wird pessimistische Synchronisation eingesetzt
- ➔ keine Starvation

### 2. Objekt-Ebene:

- optimistisch und pessimistisch synchronisierte Datenobjekte
- pessimistische Synchronisation für *Hot-Spot*-Objekte

### 3. Kombination

#### • Erhöhte Verfahrenskomplexität

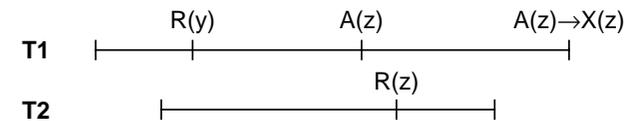
- auch bei pessimistischer Synchronisation Änderungen in privatem TA-Puffer
- erweiterte Validierung: TA scheitert, falls unverträgliche Sperre gesetzt ist
- (teilweise) pessimistisch synchronisierte TA:
  - bei EOT optimistische TA zurücksetzen, die auf X-gesperrte Objekte zugegriffen haben
  - Schreibphase mit anschließender Sperrfreigabe

## Sperrverfahren mit Versionen (RAX)

#### • Kompatibilitätsmatrix:

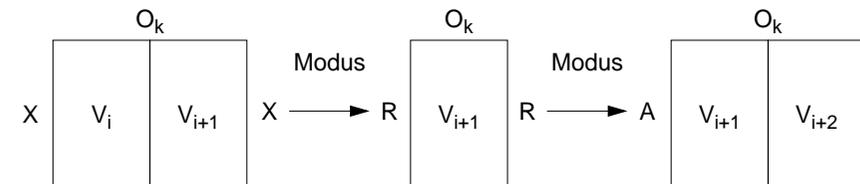
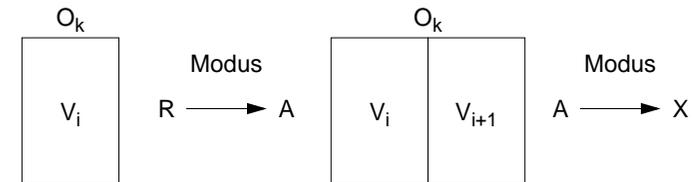
	R	A	X
R	+	⊕	-
A	⊕	-	-
X	-	-	-

#### • Ablaufbeispiel



RAX: T2 → T1

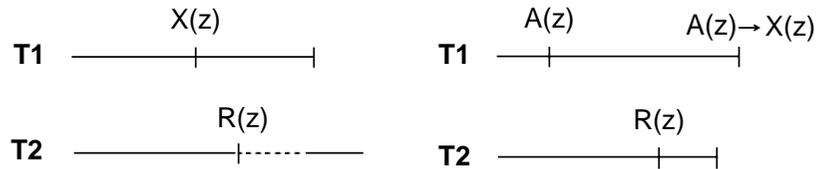
#### • Änderungen erfolgen in temporärer Objektkopie



## RAX (2)

### Eigenschaften von RAX

- Paralleles Lesen der gültigen Version wird zugelassen
- Schreiben wird nach wie vor serialisiert (A-Sperre)
- Bei EOT Konversion der A- nach X-Sperren, ggf. auf Freigabe von Lesesperren warten (Deadlock-Gefahr)
- Höhere Parallelität als beim RX-Verfahren, jedoch i. allg. andere Serialisierbarkeitsreihenfolge:



RX: T1 → T2

RAX: T2 → T1

### Nachteile

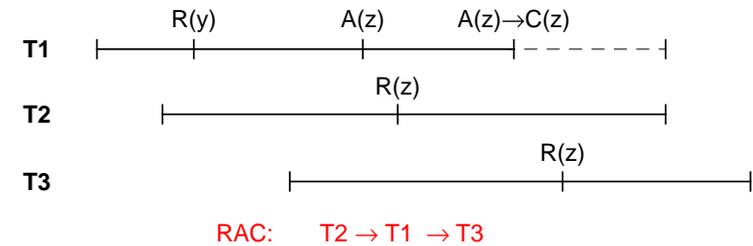
- Neue Version wird für neu ankommende Leser erst verfügbar, wenn alte Version aufgegeben werden kann
- Starke Behinderungen von Update-TA durch (lange) Leser möglich
  - ➔ Bei TA-Mix von langen Lesern und kurzen Schreibern auf gemeinsamen Objekten bringt RAX keinen großen Vorteil

## Sperrverfahren mit Versionen (RAC)

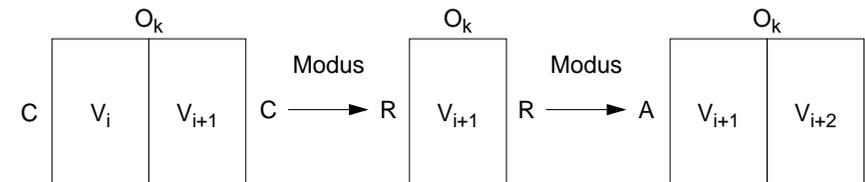
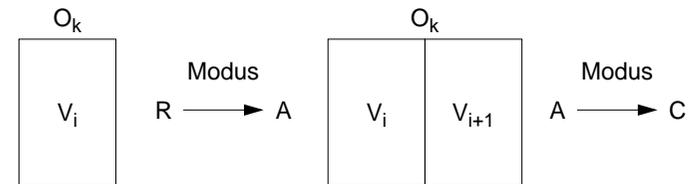
### Kompatibilitätsmatrix:

	R	A	C
R	+	+	⊕
A	+	-	-
C	⊕	-	-

### Ablaufbeispiel



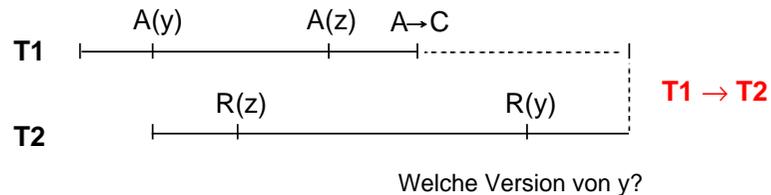
### Änderungen erfolgen ebenfalls in temporärer Objektkopie



## RAC (2)

### Eigenschaften von RAC

- Änderungen werden nach wie vor serialisiert (A-Sperre erforderlich)
- Bei EOT Konversion von A → C-Sperre
- Maximal 2 Versionen, da C-Sperren mit sich selbst und mit A-Sperren unverträglich sind
- C-Sperre zeigt Existenz **zweier gültiger Objektversionen** an



➔ **Kein Warten auf Freigabe von Lesesperren auf alter Version**  
(R- und C-Modus sind verträglich)

### Nachteile

- RAC ist nicht chronologieerhaltend
- Verwaltung komplexer Abhängigkeiten (z. B. über Abhängigkeitsgraphen)
  - ➔ **komplexere Sperrverwaltung**
- Leseanforderungen bewirken nie Blockierung/Rücksetzung, jedoch:
  - Auswahl der „richtigen“ Version erforderlich**
- Änderungs-TA, die auf C-Sperre laufen, müssen warten, **bis alle Leser der alten Version beendet**, weil nur 2 Versionen
- ➔ **ABHILFE:** allgemeines Mehrversionen-Verfahren

## Mehrversionen-Verfahren

### Änderungs-TA erzeugen neue Objektversionen

- Es kann immer nur eine neue Version pro Objekt erzeugt werden
- Sie wird bei EOT der TA freigegeben

### Lese-TA sehen den bei ihrem BOT gültigen DB-Zustand

- Sie greifen immer auf die jüngsten Objektversionen zu, die bei ihrem BOT freigegeben waren
- Sie setzen und beachten keine Sperren
- Es gibt keine Blockierungen und Rücksetzungen für Lese-TA, dafür ggf. Zugriff auf veraltete Objektversionen

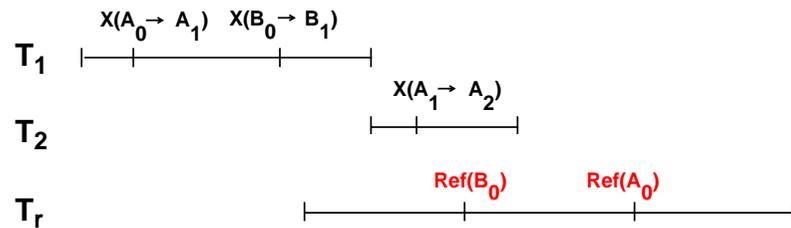
### Beispiel für Objekt $O_k$

$V_i$
$V_{i-1}$

### Zeitliche Reihenfolge der Zugriffe auf $O_k$

$T_j$ (BOT)	➔ $V_i$ (aktuelle Version)
$T_m(X)$	➔ Erzeugen $V_{i+1}$
$T_n(X)$	➔ Verzögern bis $T_m$ (EOT)
$T_m$ (EOT)	➔ Freigeben $V_{i+1}$
$T_n(X)$	➔ Erzeugen $V_{i+2}$
$T_j$ (Ref)	➔ $V_i$
$T_n$ (EOT)	➔ Freigeben $V_{i+2}$

## Mehrversionen-Verfahren (2)



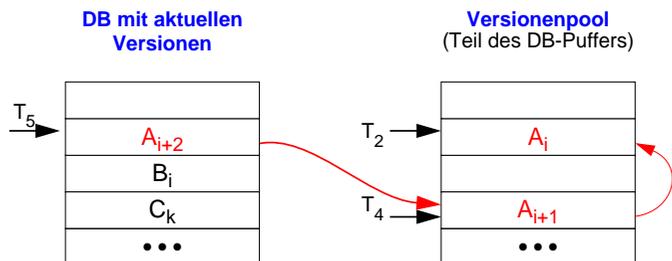
### Konsequenz

- Lese-TA werden bei der Synchronisation nicht mehr berücksichtigt
- Änderungs-TA werden untereinander über ein allgemeines Verfahren (Sperrern, OCC, . . .) synchronisiert

→ deutlich weniger Synchronisationskonflikte

### Zusätzlicher Speicher- und Wartungsaufwand

- Versionenpoolverwaltung, *Garbage Collection*
- Auffinden von Versionen



- Speicherplatzoptimierung: Versionen auf Satzebene, Einsatz von Komprimierungstechniken
- Was passiert, wenn Implementierung auf n Versionen begrenzt ist?

- Verfahren bereits in einigen **kommerziellen DBVS** eingesetzt (Oracle, RDB)

## Zeitstempelverfahren

### Grundsätzliche Idee

- TA bekommt bei BOT einen systemweit eindeutigen Zeitstempel; er legt die Serialisierbarkeitsreihenfolge fest
- TA hinterlässt den Wert ihres Zeitstempels bei jedem Objekt  $O_i$ , auf das sie zugreift
- Prüfung der Serialisierbarkeit ist sehr einfach (Zeitstempelvergleich)
  - Bei allen Objektzugriffen muss die Zeitstempelreihenfolge (Timestamp Ordering (TO)) eingehalten werden

### Prinzipielle Arbeitsweise

- Vergabe von eindeutigen TA-IDs (Zeitstempel  $ts$  der TA) in aufsteigender Reihenfolge
- „Stempeln“ des Objektes  $O$  bei Zugriffen von  $T_i$ :  $TS(O) := ts(T_i)$

### Konfliktprüfung:

if  $ts(T_i) < TS(O)$  then ABORT  
else verarbeite;

→ Wenn eine Transaktion „zu spät“ kommt, wird sie zurückgesetzt und muss wiederholt werden

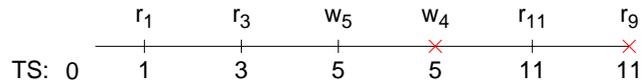
$T_7$ :  $O_k$   $TS(O_k) = 10$

$O_n$   
 $TS(O_n) = 5$

$O_m$   
 $TS(O_m) = 3$

## Zeitstempelverfahren (2)

- Zugriffsfolge auf Objekt O (nur ein allgemeiner Zeitstempel TS):



→ kein Konflikt bei r<sub>9</sub>!

- Verfeinerung: 2 Zeitstempel pro Objekt

- Erhöhung beim Schreiben: WTS
- Erhöhung beim Lesen: RTS
- Regeln für T<sub>i</sub> und O (Abk. ts(T<sub>i</sub>) = i)

R1:  $r_i \wedge (i \geq \text{WTS}(O)) \Rightarrow \text{if } \text{RTS}(O) < i \text{ then } \text{RTS}(O) := i$ ; Lesen

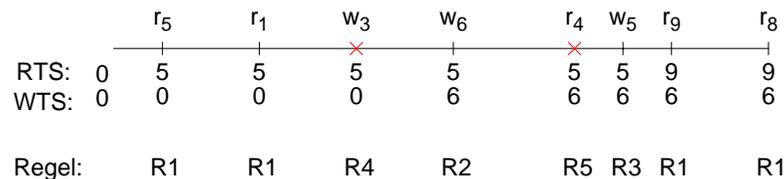
R2:  $w_i \wedge (i \geq \text{RTS}(O)) \wedge (i \geq \text{WTS}(O)) \Rightarrow \text{WTS}(O) := i$ ; Ändern

R3:  $w_i \wedge (i \geq \text{RTS}(O)) \wedge (i < \text{WTS}(O)) \Rightarrow$  kein Konflikt (*blind update*)  
– kein Schreiben – weiter<sup>6</sup>

R4:  $w_i \wedge (i < \text{RTS}(O)) \Rightarrow$  Zurücksetzen

R5:  $r_i \wedge (i < \text{WTS}(O)) \Rightarrow$  Zurücksetzen

- Zugriffsfolge auf Objekt O:

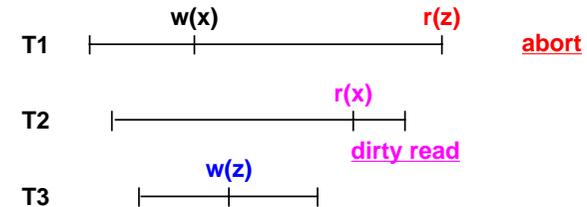


6. Diese Regel wird in der Literatur mit "Thomas' Write Rule" bezeichnet

## Zeitstempelverfahren (3)

- Wo liegt das Problem?

$$ts(T_i) = i$$



- Vorkehrungen für den ABORT-Fall

- Sofortige Zulassung aller Schreiboperationen erzeugt inkonsistente DB
- Einfrieren der Zeitstempel bis COMMIT der ändernden TA
- Basic Timestamp Ordering (BTO)<sup>7</sup> schlägt Lösung mit Sperrverfahren und Verwaltung komplexer Abhängigkeiten vor

- Eigenschaften von TO

- Serialisierbarkeitsreihenfolge einer Transaktion wird bei BOT festgelegt
- Deadlocks sind ausgeschlossen
- aber: (viel) höhere Rücksetzraten als pessimistische Verfahren
- ungelöste Probleme, z. B. wiederholter ABORT einer Transaktion

- Hauptsächlicher Einsatz

- Synchronisation in Verteilten DBS
- lokale Prüfung der Serialisierbarkeit direkt am Objekt O<sub>i</sub> (geringer Kommunikationsaufwand)

7. Bernstein, P.A., Goodman, N.: Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems, in: Proc. 6th Int. Conf. on VLDB, 1980, 285-300

## Zeitstempelverfahren (4)

### • Implementierung des Protokolls BTO<sup>8</sup>

- Alle Prüfungen/Entscheidungen müssen lokal erfolgen
- Erwerb von Anwartschaften: Prewrites
- **Prewrite i verzögert r<sub>j</sub>, w<sub>j</sub> mit j > i**
- Einführung von *Read-Queues* (R-Q), *Prewrite-Queues* (P-Q) und *Write-Queues* (W-Q)

### • Zugriffsprotokoll auf Objekt O:

	r <sub>3</sub>	p <sub>3</sub>	r <sub>6</sub>	r <sub>4</sub>	p <sub>5</sub>	w <sub>3</sub>	p <sub>7</sub>	w <sub>7</sub>	w <sub>5</sub>
RTS:	2 3	3	3	3	3	3 4	4	4	4 6 6
WTS:	1 1	1	1	1	1	3 3	3	3	5 5 7
R-Q:			6	4 6	4 6	6	6	6	6
P-Q:		3			3 5	5	5 7	5 7	7 7
W-Q:							7	7 7	

➔ komplexe Verwaltung von Abhängigkeiten in R-Q, P-Q, W-Q

## Prädikatssperren<sup>9</sup>

### • Logische Sperren oder Prädikatssperren

- Minimaler Sperrbereich durch geeignete Wahl des Prädikats
- **Verhütung des Phantomproblems**
- Eleganz

### • Form:

LOCK (R, P, a)

R Tabellenname

P Prädikat

a ∈ {read, write}

UNLOCK (R, P)

### - Lock (R, P, write)

sperrt alle möglichen Sätze von R exklusiv, die Prädikat P erfüllen

### • Beispiel:

<b>T1:</b> LOCK(R1, P1, read) LOCK(R2, P2, write) LOCK(R1, P5, write) ...	<b>T2:</b> ... LOCK(R2, P3, write) LOCK(R1, P4, read) ...
--	--

### • Wie kann Konflikt zwischen zwei Prädikaten festgestellt werden?

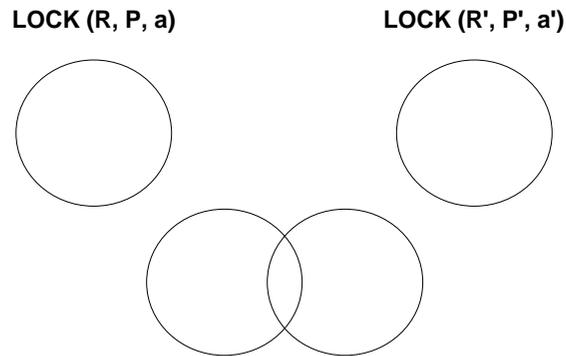
- Im allgemeinen Fall rekursiv unentscheidbar, selbst mit eingeschränkten arithmetischen Operatoren
- Entscheidbare Klasse von Prädikaten: einfache Prädikate
- ➔ (A Θ Wert) {^, v} (. . .

8. Peinl, P.: Synchronisation in zentralisierten Datenbanksystemen, Informatik-Fachberichte 161, Springer-Verlag, 1987

9. Eswaran, K.P. et al.: The notions of consistency and predicate locks in a data base system. in: Comm. ACM 19:11, 1976, 624-633

## Prädikatsperren (2)

### • Entscheidungsprozedur



1. Wenn  $R \neq R'$ , kein Konflikt
2. Wenn  $a = \text{read}$  und  $a' = \text{read}$ , kein Konflikt
3. Wenn  $P(t) \wedge P'(t) = \text{TRUE}$  für irgendein  $t$ , dann besteht ein Konflikt

T1: LOCK (Pers, Alter > 50, read)      T2: LOCK (Pers, Pnr = 4711, write)

➔ **Entscheidung:**

### • Nachteile

- Erfüllbarkeitstest:  
Aufwendige Entscheidungsprozedur mit vielen Prädikaten ( $N > 100$ )  
(wird in innerer Schleife des Lock-Mgr häufig aufgerufen)
- **Pessimistische** Entscheidungen ➔ **Einschränkung der Parallelität**  
(es wird auf Erfüllbarkeit getestet!)
- Einsatz nur bei deskriptiven Sprachen!
- Sonderfall:  $P=\text{TRUE}$  entspricht einer Tabellensperre  
➔ **große Sperrgranulate, geringe Parallelität**

## Prädikatsperren (3)

### • Effizientere Implementierung: Präzisionssperren<sup>10</sup>

- sperren nur die **gelesenen Daten** durch Prädikate
- setzen für aktualisierte Sätze Schreibsperren  
➔ Es ist kein Disjunktheitstest für Prädikate mehr erforderlich, sondern es ist lediglich zu überprüfen, ob der Satz ein Prädikat erfüllt

### • Datenstrukturen:

- **Prädikatsliste:**  
Lesesperren laufender TA werden durch **Prädikate** beschrieben

(Pers: Alter > 50 and Beruf = 'Prog.')

(Pers: Pname = 'Meier' and Gehalt > 50000)

(Abt: Anr=K55)

...

### - **Update-Liste:**

enthält geänderte **Sätze** laufender TA

(Pers: 4711, 'Müller', 30, 'Prog.', 70000)

(Abt: K51, 'DBS', ...)

...

### • Leseanforderung (Prädikat P):

- für jeden Satz der Update-Liste ist zu prüfen, ob es P erfüllt
- wenn ja ➔ **Sperrkonflikt**

### • Schreibanforderung (Satz T):

- für jedes Prädikat P der Prädikatliste ist zu prüfen, ob T es erfüllt
- wenn T keines erfüllt ➔ **Schreibsperre wird gewährt**

10. J.R. Jordan, J. Banerjee, R.B. Batman: Precision Locks, in: Proc. ACM SIGMOD, 1981, 143-147

## Synchronisation auf Objekten

- **Erhöhung der Parallelität** durch Einführung kommutativer („semantischer“) DB-Operationen als Einheit für die Synchronisation
  - Synchronisation erfolgt auf abstrakterer Ebene (**Objektebene**)
  - Realisierung muss auf niedrigerer Ebene Korrektheit gewährleisten (z. B. durch Escrow-Verfahren)

### Beispiel

Zwei Kontobuchungen auf Konten K1 und K2 durch die TA T1 und T2, die **kommutative Operationen** „erhöhe um x“ und „vermindere um x“ verwenden, könnten z.B. in folgenden Reihenfolgen ausgeführt werden:

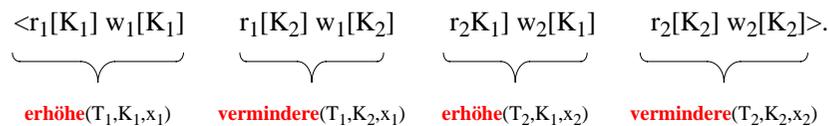
#### Schedule S1

1. erhöhe (T1,K1,x1)
2. vermindere (T1,K2,x1)
3. erhöhe (T2,K1,x2)
4. vermindere (T2,K2,x2)

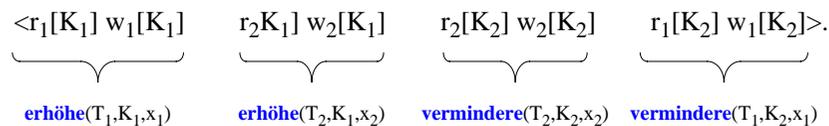
#### Schedule S2

1. erhöhe (T1,K1,x1)
2. erhöhe (T2,K1,x2)
3. vermindere (T2,K2,x2)
4. vermindere (T1,K2,x1)

➔ Schedule S1 ist auch mit rein syntaktischen Verfahren (r/w) erzeugbar:



➔ Schedule S2 ist hingegen mit rein syntaktischen Verfahren nicht erzeugbar.



## Synchronisation von High-Traffic-Objekten

### High-Traffic-Objekte

- auch *Hot Spots* genannt: Datenelemente, auf die viele TA ändernd zugreifen müssen
- können leicht einen Systemengpass bilden
- meist numerische Felder mit aggregierten Informationen z. B.
  - Anzahl freier Plätze
  - Summe aller Kontostände
- aber auch DBS-interne Strukturen und Systemressourcen
  - Wurzeln von Bäumen (erfordern spezielle Protokolle)
  - Log-Datei, Sperrtabelle, DB-Puffer usw.

➔ bei ungeschicktem TA-Scheduling (FIFO) besteht Gefahr der Ausbildung langer TA-Warteschlangen (**Konvoiphänomen**)

### Einfachste Lösung der Synchronisationsprobleme

- Vermeidung solcher Datenelemente beim DB-Entwurf
- Einsatz von speziellen Sperrern und angepassten Schedulingverfahren bei Systemressourcen

## Synchronisation von *High-Traffic*-Objekten (2)

- Was lässt sich auf SQL-Ebene tun?

- Modifikation eines Feldes FreiePlätze

```
exec sql  select FreiePlätze
         into :anzahl
         from Flüge
         where FlugNr = 'LH127';
anzahl := anzahl-2;
exec sql  update Flüge
         set FreiePlätze = :anzahl
         where FlugNr = 'LH127';
```

- ➔ **Deadlock-Gefahr wegen Sperrkonversion.**  
Erhöhung dieser Gefahr wegen bei *Hot-Spot*-Verhalten

- **Verbesserung**

```
exec sql  update Flüge
         set FreiePlätze = FreiePlätze-2
         where FlugNr = 'LH127';
```

- ➔ **keine Deadlock-Gefahr, jedoch möglicherweise**  
langes Warten auf Commit der Vorgänger-TAs.

## Synchronisation von *High-Traffic*-Objekten (3)

- **Spezielle Behandlung von *Hot Spots***

- erfordert Systemmodifikation
- Aktion auf *Hot Spot* wird in zwei Teile zerlegt
  - Testen eines Prädikats
  - Durchführung der Transformation

- Beispiel:

```
exec sql  update hotspot Flüge
         set      FreiePlätze = FreiePlätze-2
         where   FlugNr = 'LH127'
         and     FreiePlätze > 2;
```

- **Ablauf des Feldzugriffs**

1. Beim Zugriff erfolgt der Test des Prädikats unter einer kurzen Lesesperre
2. Falls der Test zu 'false' evaluiert wird, wird abgebrochen (Fehlerbehandlung der Anwendung)
3. Sonst wird eine Redo-Log-Satz mit Prädikat und Modifikationsoperation angelegt
4. **Beim Erreichen von Commit** werden die Transformationen von *Hot-Spot*-Feldern in zwei Phasen durchgeführt:
  - **Phase 1:** Alle Redo-Log-Sätze der TA, die Commit ausführt, werden abgearbeitet. Für Feldzugriffe, die keine Transformation erfordern, werden Lesesperren angefordert und für die alle anderen Feldzugriffe Schreibsperrern. Danach werden alle Prädikate noch einmal evaluiert. Falls mindestens ein Prädikat zu 'false' evaluiert, wird die TA zurückgesetzt. Sonst Eintritt in Phase 2.
  - **Phase 2:** Alle Transformationen werden angewendet und die Sperren werden freigegeben.

## Synchronisation von *High-Traffic*-Objekten (4)

- **Beispiel:**  
3 TA werden gleichzeitig aufgerufen, aber sequentiell abgewickelt

**T1:** FP > 5 ?  
FP := FP-5  
Commit

**T2:** FP > 2 ?  
FP := FP-2  
Commit

**T3:** FP > 8 ?  
FP := FP-8  
Commit

- **Beispiel:**  
Spezielle Behandlung von *Hot Spots*

T1	T2	T3	FP in DB
FP > 5 ?			12
	FP > 2 ? Commit FP > 2 FP := FP-2		
		FP > 8 ?	10
Commit FP > 5 ? FP := FP-5		Commit FP > 8 ?	5

## Ansatz von IMS Fast Path

- **Spezielle Operationen für *High-Traffic*-Objekte:**

VERIFY      *FreiePlätze* > *Anforderung*  
MODIFY      *FreiePlätze* := *FreiePlätze* – *Anforderung*

- **Quasi-optimistische Synchronisation:**

- Zunächst werden keine (langen) Sperren gesetzt
- Änderungen werden nicht direkt vorgenommen, sondern nur in '*intention list*' vermerkt
- ➔ Eine TA sieht bei wiederholtem Zugriff auf ein solches Feld ihre eigenen Änderungen nicht!
- Bei EOT Validierung- und Schreibphase:
  - Überprüfung, ob VERIFY-Prädikate noch erfüllt sind (geringe Rücksetzwahrscheinlichkeit)
  - MODIFY-Operation vornehmen
  - Sperren werden nur für Dauer der EOT-Behandlung gehalten

➔ **Verkürzung der Dauer exklusiver Sperren, weit geringere Konfliktgefahr als bei normalen Schreibsperren**

- **Alternative:**

Nutzung von semantischem Wissen zur Synchronisation wie Kommutativität von Änderungsoperationen auf solchen Feldern

- **Beispiel: Inkrement-/Dekrement-Operation**

	R	X	Inc/Dec
R	+	-	-
X	-	-	-
Inc/Dec	-	-	+

➔ Wie sieht das mit der Bereichskontrolle bei solchen Feldern aus?

## Escrow-Ansatz<sup>11</sup>

- **High-Traffic-Objekte**

- Deklaration als Escrow-Felder
- Benutzung spezieller Operationen
  - Anforderung einer bestimmten Wertemenge

```
IF ESCROW (field=F1, quantity=C1, test=(condition))
  THEN 'continue with normal processing'
  ELSE 'perform exception handling'
```

- Benutzung der reservierten Wertmengen:

```
USE (field=F1, quantity=C2)
```

- Optionale Spezifizierung eines Bereichstest bei Escrow-Anforderung
- Wenn Anforderung erfolgreich ist, kann Prädikat nicht mehr nachträglich invalidiert werden

➔ keine spätere Validierung/Zurücksetzung

- **Aktueller Wert eines Escrow-Feldes**

- ist unbekannt, wenn laufende TA Reservierungen angemeldet haben

➔ Führen eines Wertintervalls, das alle möglichen Werte nach Abschluss der laufenden TA umfasst

- für Wert  $Q_k$  des Escrow-Feldes  $k$  gilt:

$$LO_k \leq INF_k \leq Q_k \leq SUP_k \leq HI_k$$

- Anpassung von INF, Q, SUP bei Anforderung, Commit und Abort einer TA

## Escrow-Ansatz (2)

- **Beispiel:**

Zugriffe auf Feld mit LO=0, HI=500 (Anzahl freier Plätze)

Anforderungen/Rückgaben				Wertintervall		
T1	T2	T3	T4	INF	Q	SUP
				15	15	15
-5						
	-8					
		+4				
			-3			
commit						
		commit				
	abort					

- **Eigenschaften**

- Durchführung von Bereichstests bezüglich des Wertintervalls
- Minimal-/Maximalwerte (LO, HI) dürfen **nicht überschritten** werden
- hohe Parallelität ändernder Zugriffe möglich

- **Nachteile:**

- spezielle Programmierschnittstelle
- tatsächlicher Wert ggf. nicht abrufbar

11. P. O'Neil: The Escrow Transactional Method, in: ACM Trans. on Database Systems 11: 4, 1986, 405-430

## Klassifikation von Synchronisationsverfahren

### • Gemeinsame Ziele

- Erhöhung der Parallelität
- Reduktion von Behinderungen/Blockierungen
- Einfache Verwaltung

### • Erhöhung der Parallelität durch Objektreplication

- **Kopien:** temporär, privat, nicht freigeben
- **Versionen:** permanent, mehrbenutzbar, freigeben

#Versionen #Kopien	1	2	N	∞
0				
1				
P				

### • Beobachtung

- Einsatz in existierenden DBS: vor allem **hierarchische Sperrverfahren** und Varianten, aber auch **Mehrversionen-Verfahren**
- Es existieren eine Fülle von allgemeingültigen und spezialisierten Synchronisationsverfahren (zumindest in der Literatur)
- Es kommen (ständig) Verfahren durch **Variation der Grundprinzipien** dazu!

## Leistungsanalyse – Simulationsverfahren

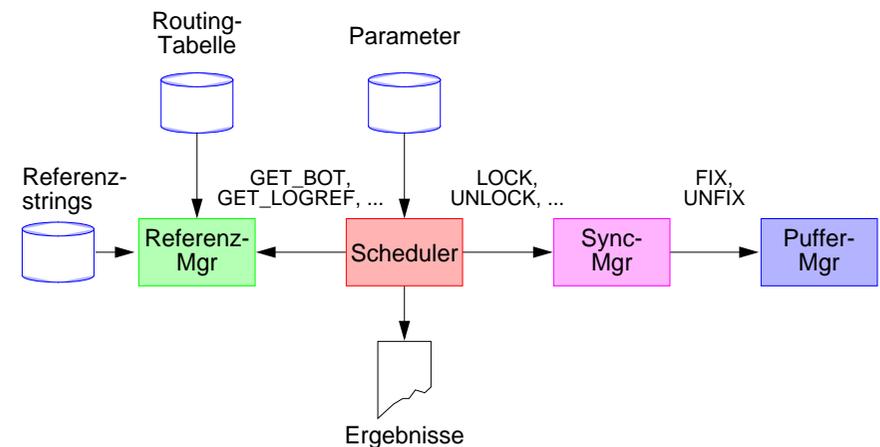
### • Analyse von Synchronisationsverfahren

- **pessimistisch:** RX, RX2
- **optimistisch:** BOCC, FOCC-K (Kill), FOCC-H (Hybrid)
- **Versionen:** RAC, Mehrversionen-Verfahren (MVC)

### • Nachbildung der DB-Last

- Aufzeichnung der Seitenreferenzen realer Anwendungen im DBS
- Nutzung verschiedenartiger TA-Mixe in Form von Referenzstrings
- Simulation des DB-Puffers und der benötigten E/A-Zeiten
- Ermittlung der Durchlaufzeiten unter den verschiedenen Synchronisationsverfahren und den eingestellten Sollparallelitäten

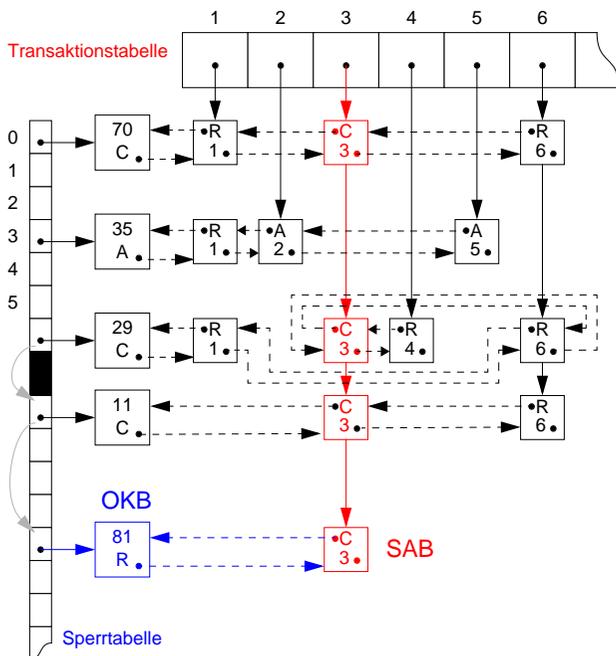
### • Simulationsverfahren



## Implementierungsaspekte – Datenstrukturen

### • Probleme bei der Implementierung von Sperren

- Kleine Sperreinheiten (wünschenswert) erfordern hohen Aufwand
- Sperranforderung und -freigabe sollten sehr schnell erfolgen, da sie sehr häufig benötigt werden → **Sperrtabelle ist High-Traffic-Objekt!**
- Explizite, satzweise Sperren führen u. U. zu umfangreichen Sperrtabellen und großem Zusatzaufwand



### → Beispiel: Sperrtabelle / TA-Tabelle für RAC-Verfahren

- Hash-Tabelle erlaubt schnellen Zugriff auf Objektkontrollblöcke (OKB)
- Matrixorganisation der Sperrtabelle mit Sperranforderungsblöcken (SAB)
- Spezielles Sperrverfahren: **Kurzzeitsperren** für Zugriffe auf Sperrtabelle (Semaphor pro Hash-Klasse reduziert Konflikt-/Konvoi-Gefahr)

## Leistungsanalyse und Bewertung von Synchronisationsverfahren

### • Wie bewertet man Parallelität?

- hoher Parallelitätsgrad – viele Rücksetzungen und Wiederholungen
- moderate Parallelität, dafür geringerer Zusatzaufwand

### • Durchsatztest

- alle Transaktionen inkl. (mehrfache) Wiederholungen sind bearbeitet (Ermittlung der Durchlaufzeit)
- einstellbarer Grad der maximalen Parallelität

### • Messung der effektiven Parallelität

- $n$  = nominale Parallelität (MPL)
- $n'$  = durchschnittliche Anzahl aktiver Transaktionen (berücksichtigt Wartesituationen)
- $q$  = tatsächliche Arbeit (Referenzen) / minimale Arbeit (berücksichtigt Rücksetzungen und Wiederholungen)

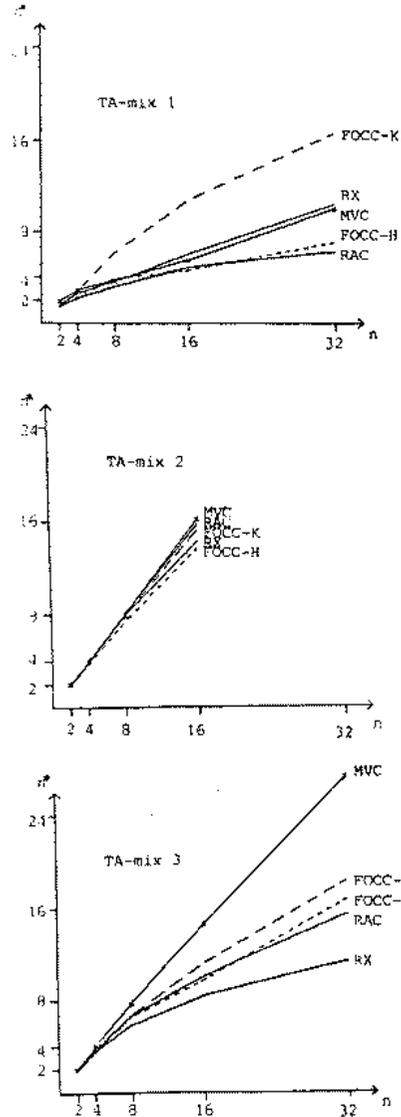
### - effektive Parallelität

$$n^* = n'/q$$

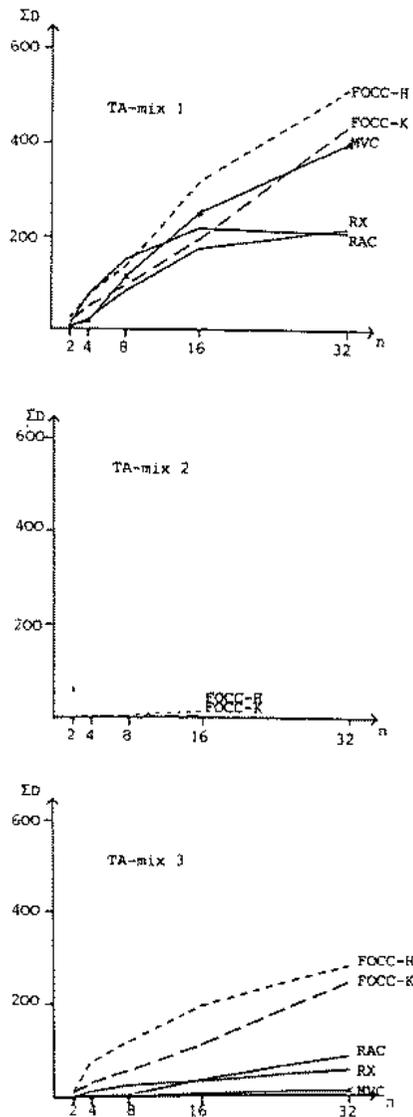
### • Zählung der Deadlocks

## Synchronisationsverfahren – Vergleich

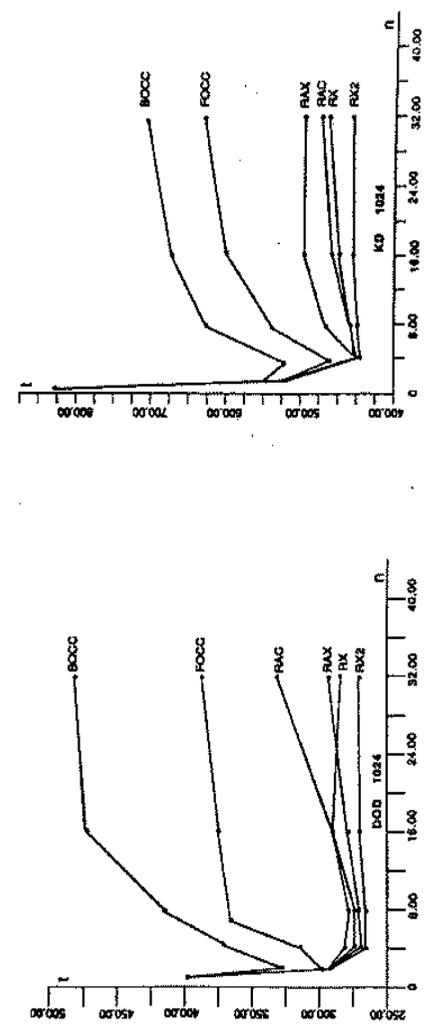
Effektive Parallelität



Summe der Deadlocks



## Vergleich verschiedener Synchronisationsverfahren



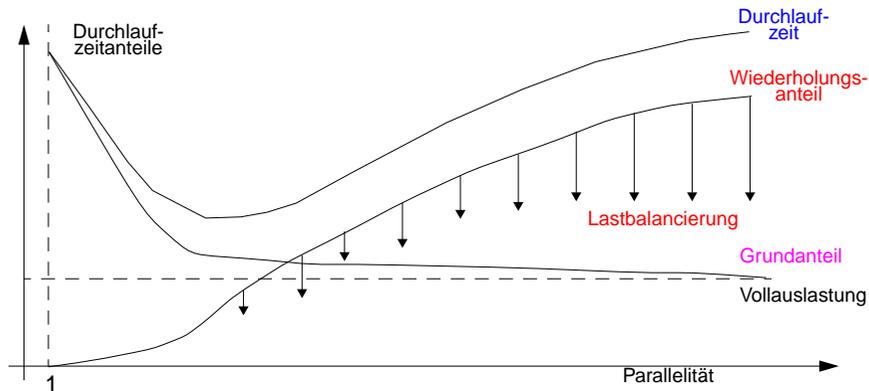
• **Schlussfolgerungen**

- Sehr geringe Parallelität → keine effektive Nutzung der Ressourcen
- Geringe Parallelität → bester Durchsatz, nicht notwendigerweise kürzeste Antwortzeiten
- Pessimistische Methoden gewinnen: Blockierung vermeidet häufig Deadlocks
- Optimistische Methoden geraten leicht in ein Threading-Verhalten
- RX2 reduziert effektiv den Wettbewerb um gemeinsam genutzte Daten

## Synchronisation und Lastkontrolle

- **Charakteristische Wannenförmigkeit (idealisiert)**

- Sie ergibt sich bei vielen Referenzstrings und Synchronisationsverfahren



- Sie wird von **zwei gegenläufigen Faktoren** bestimmt
  - Grundanteil der Durchlaufzeit beschreibt die Zeit, die zur Verarbeitung durch den Referenzstring vorgegebener Last bei fehlender Synchronisation nötig wäre
  - Wiederholungsanteil umfasst die Belegung des Prozessors zur nochmaligen Ausführung zurückgesetzter TA

➔ **Rolle der Lastkontrolle und Lastbalancierung!**

- **Empirische Bestätigung**

- Theoretische Untergrenze des Grundanteils wird bei Vollauslastung des Prozessors erreicht
- Häufigkeit von Leerphasen des Prozessors

DOD 1024							
PAR	BOCCT	FOCC	BOCC	RX2	RAX	RAC	RX
1	4864	4864	4864	4864	4864	4864	4864
2	1623	1734	1584	1719	1725	1717	1788
4	59	105	50	149	206	94	256
8	108	37	17	17	26	31	53
16	36	23	8	27	5	40	90
32	70	40	5	28	8	70	46

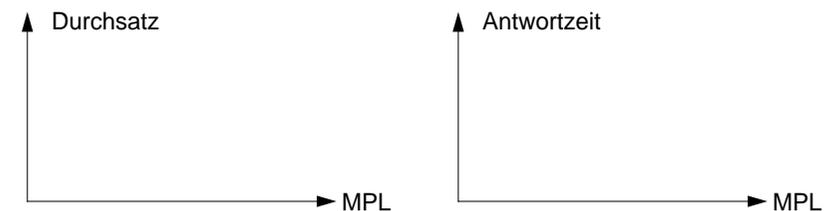
## Dynamische Lastkontrolle

- **Was nützt „blinde“ Durchsatzmaximierung?**

**mehr** aktive TA → **mehr** gesperrte Objekte → **höhere** Konflikt-WS → **längere** Sperrwartezeiten, **höhere** Deadlock-Raten → **noch mehr** aktive TA

- **Parallelitätsgrad (multiprogramming level, MPL)**

- Er hat wesentlichen Einfluss auf das Leistungsverhalten, bestimmt Umfang der Konflikte bzw. Rücksetzungen
- Gefahr von *Thrashing* bei Überschreitung eines **kritischen MPL-Wertes**



- **Statische MPL-Einstellung unzureichend:**

wechselnde Lastsituationen, mehrere Transaktionstypen

- **Idee:**

dynamische Einstellung des MPL zur Vermeidung von *Thrashing*

- **Ein möglicher Ansatz -**

**Nutzung einer Konfliktquote bei Sperrverfahren<sup>12</sup>:**

$$\text{Konfliktquote} = \frac{\text{\# gehaltener Sperren}}{\text{\# Sperren nicht-blockierter Transaktionen}}$$

**kritischer Wert:** ca. 1,3 (experimentell bestimmt)

- Zulassung neuer TA nur, wenn kritischer Wert noch nicht erreicht ist
- Bei Überschreiten erfolgt Abbrechen von TA

12. Weikum, G. et al.: The Comfort Automatic Tuning Project, in: Information Systems 19:5, 1994, 381-432

## Zusammenfassung

- **Serialisierbare Abläufe**
  - gewährleisten „**automatisch**“ Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen
- **Realisierung der Synchronisation durch Sperrverfahren**
  - Sperren stellen während des laufenden Betriebs sicher, dass die resultierende Historie serialisierbar bleibt
  - Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt
  - Es gibt mehrere Varianten (RX, RUX, ...)
  - Prädikatssperren verkörpern eine elegante Idee, sind aber in praktischen Fällen nicht direkt einsetzbar, ggf. Nutzung in der Form von Präzisionssperren
  - DBS-Standard: multiple Sperrgranulate durch hierarchische Sperrverfahren
    - ➔ **Sperrverfahren sind pessimistisch und universell einsetzbar**
- **Deadlock-Problem ist bei blockierenden Verfahren inhärent!**
- **Einführung von Konsistenzebenen**
  - zwei (geringfügig) unterschiedliche Ansätze
    - basierend auf Sperrdauer für R und X
    - basierend auf zu tolerierende "Fehler"
  - „**Schwächere**“ **Synchronisation** von Leseoperationen erlaubt **höhere Parallelitätsgrade** und Reduktion von Blockierungen, erfordert aber **Programmierdisziplin!**
    - ➔ **Inkaufnahme von Anomalien reduziert die TA-Behinderungen**

## Zusammenfassung (2)

- **Implementierung der Synchronisation: weitere Verfahren**
  - RAX und RAC begrenzen Anzahl der Versionen und reduzieren Blockierungsdauern nur für bestimmte Situationen
  - **Mehrversionen-Verfahren** liefert hervorragende Werte bei der effektiven Parallelität und bei der Anzahl der Deadlocks, verlangt jedoch höheren Aufwand (Algorithmus, Speicherplatz)
  - **Reine OCC- und Zeitstempel-Verfahren** erzeugen zu viele Rücksetzungen
- **Generelle Optimierungen:**
  - reduzierte Konsistenzebene
  - Mehrversionen-Ansatz
- **'Harte' Synchronisationsprobleme:**
  - **Hot Spots / High-Traffic-Objekte**
  - **lange (Änderung-) TA**
  - Wenn Vermeidungsstrategie nicht möglich ist, sind zumindest für Hochleistungssysteme Spezialprotokolle anzuwenden
  - Nutzung semantischen Wissens über Operationen / Objekte zur Reduzierung von Synchronisationskonflikten
  - allerdings
    - ggf. Erweiterung der Programmierschnittstelle
    - begrenzte Einsetzbarkeit
    - Zusatzaufwand
- **Dynamische Lastkontrolle erforderlich**
  - Vermeidung von *Thrashing* bei wechselnden Lastsituationen, mehreren Transaktionstypen usw.
  - Nutzung einer Konfliktrate (1.3) zur dynamischen Einstellung der MPL