# DATENBANKANWENDUNG

## Wintersemester 2013/2014

Holger Schwarz
Universität Stuttgart, IPVS
holger.schwarz@ipvs.uni-stuttgart.de

| | |
|---|---|
| **Beginn:** | **23.10.2013** |
| **Mittwochs:** | **11.45 – 15.15 Uhr, Raum 46-268 (Pause 13.00 – 13.30)** |
| **Donnerstags:** | **10.00 – 11.30 Uhr, Raum 46-268** |
| | **11.45 – 13.15 Uhr, Raum 46-260** |

http://wwwlgis.informatik.uni-kl.de/cms/courses/datenbankanwendung/

# 8. Sperrverfahren – Implementierung und Leistungsanalyse[1]



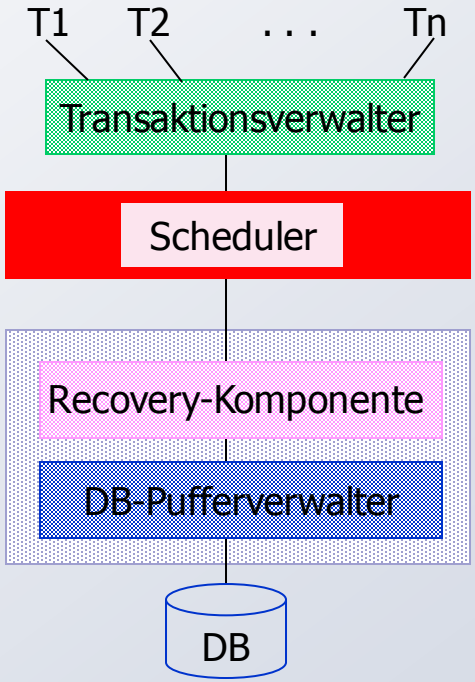
- **Synchronisation – Überblick über die Verfahren**
- **Zweiphasen-Sperrprotokolle**
  - RUX-Protokoll
  - Hierarchische Sperrverfahren
- **Einführung von Konsistenzebenen**
  - Reduzierung von TA-Blockierungen, aber Programmierdisziplin gefordert
  - Unterschiedliche Ansätze basierend auf
    - Sperrdauer für R und X
    - zu tolerierende "Fehler"
- **Optimierungen**
  - RAX und RAC
  - Mehrversionen-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.

# Einbettung des DB-Schedulers



- **Stark vereinfachtes Modell**

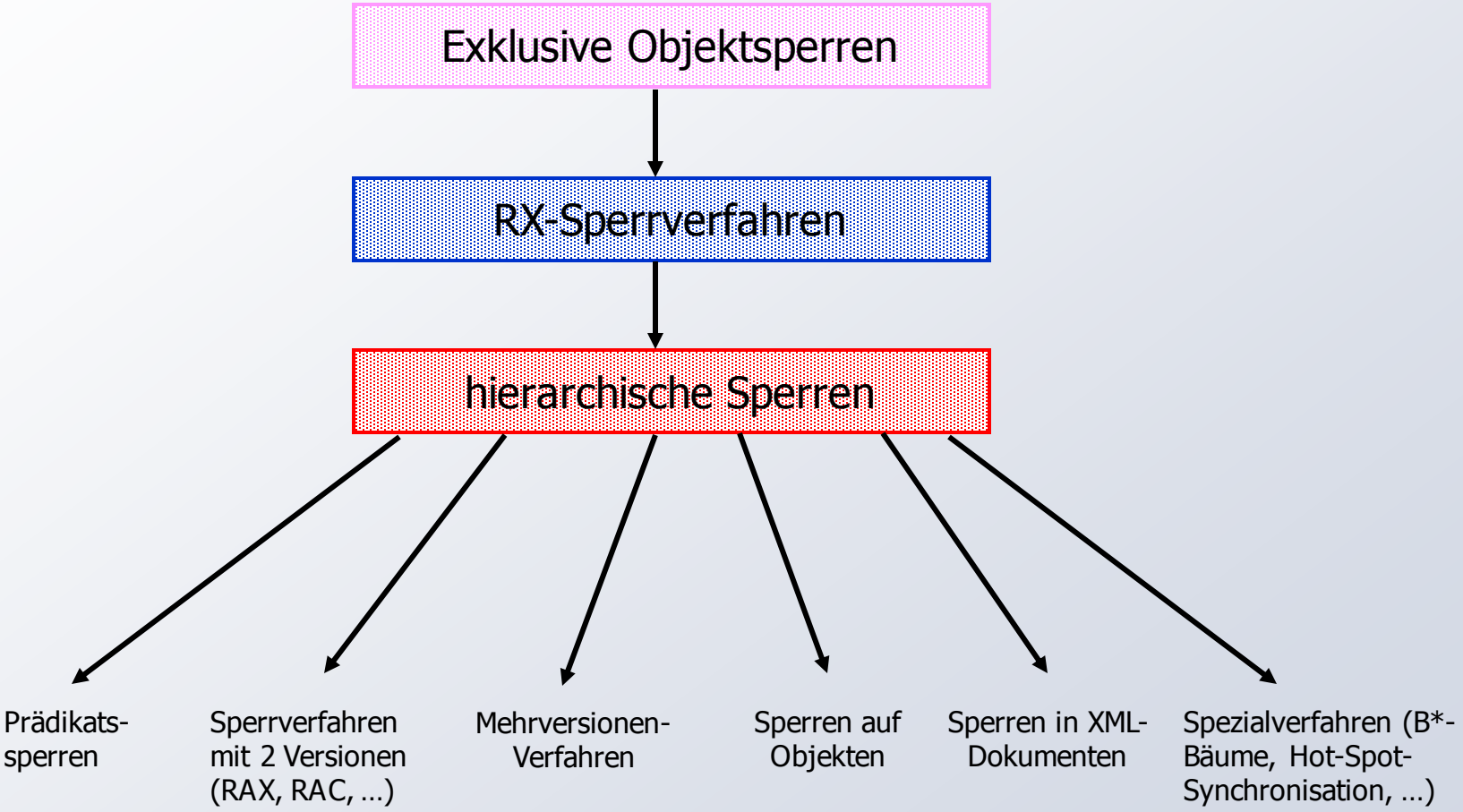



- **Welche Aufgaben hat der Scheduler (Sperrverwalter, Lock-Mgr)?**
  - als Komponente der Transaktionsverwaltung zuständig für I von ACID
  - kontrolliert die beim TA-Ablauf auftretenden Konfliktoperationen (R/W, W/R, W/W) und garantiert insbesondere, dass nur „serialisierbare" TAs erfolgreich beendet werden
  - verhindert nicht-serialisierbare TAs
  - kann jedoch keine Deadlocks verhindern. Zur Deadlock-Auflösung ist eine Kooperation mit der Recovery-Komponente erforderlich (Rücksetzen von TAs)

⇨ Garantiert „vernünftige" Schedules

# Historische Entwicklung von Synchronisationsverfahren





# Phantom-Problem



- **Einfügungen oder Löschungen können zu falschen Schlussfolgerungen verleiten**

| **Lesetransaktion** (Gehaltssumme überprüfen) | **Änderungstransaktion** (Einfügen eines neuen Angestellten) |
|---|---|
| SELECT SUM (Gehalt) INTO :summe<br>FROM Pers<br>WHERE Anr = 17 | |
| | INSERT INTO Pers (Pnr, Anr, Gehalt)<br>VALUES (4567, 17, 55.000) |
| | UPDATE Abt<br>SET Gehaltssumme = Gehaltssumme +55.000<br>WHERE Anr = 17 |
| SELECT Gehaltssumme INTO :gsumme<br>FROM Abt<br>WHERE Anr = 17 | |
| IF gsumme <> summe THEN<br><Fehlerbehandlung> | |

**Zeit**

# Sperrverfahren



- **Sperrenbasierte Synchronisation**
  - Sperren stellen während des laufenden Betriebs sicher, dass die resultierende Historie serialisierbar bleibt
  - Es gibt mehrere Varianten
- **Zur Realisierung der Synchronisation gibt es viele verschiedenartige Sperrverfahren**
  - **Sperrenbasierte Verfahren auf physischen DB-Objekten**
    Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt.
    Verfeinerungen: Konversion des Sperrmodus, multiple Sperrgranulate
  - **Versionsverfahren**
    Es werden zwei oder mehr Versionen eines Objektes gehalten.
    Keine Behinderung der Leser durch Schreiber
  - **Prädikatssperren**
    Es wird die Menge der möglichen Objekte, die das Prädikat erfüllen, gesperrt.
    Elegante Lösung für das Phantom-Problem
  - **Sperren auf Objekten**
    Nutzung der Kommutativität von Änderungen
  - **Spezielle Synchronisationsverfahren**
    Nutzung der Semantik von Änderungen
  - . . .

⇨ **Sperrverfahren sind pessimistisch und universell einsetzbar**

# RX-Sperrverfahren



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

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

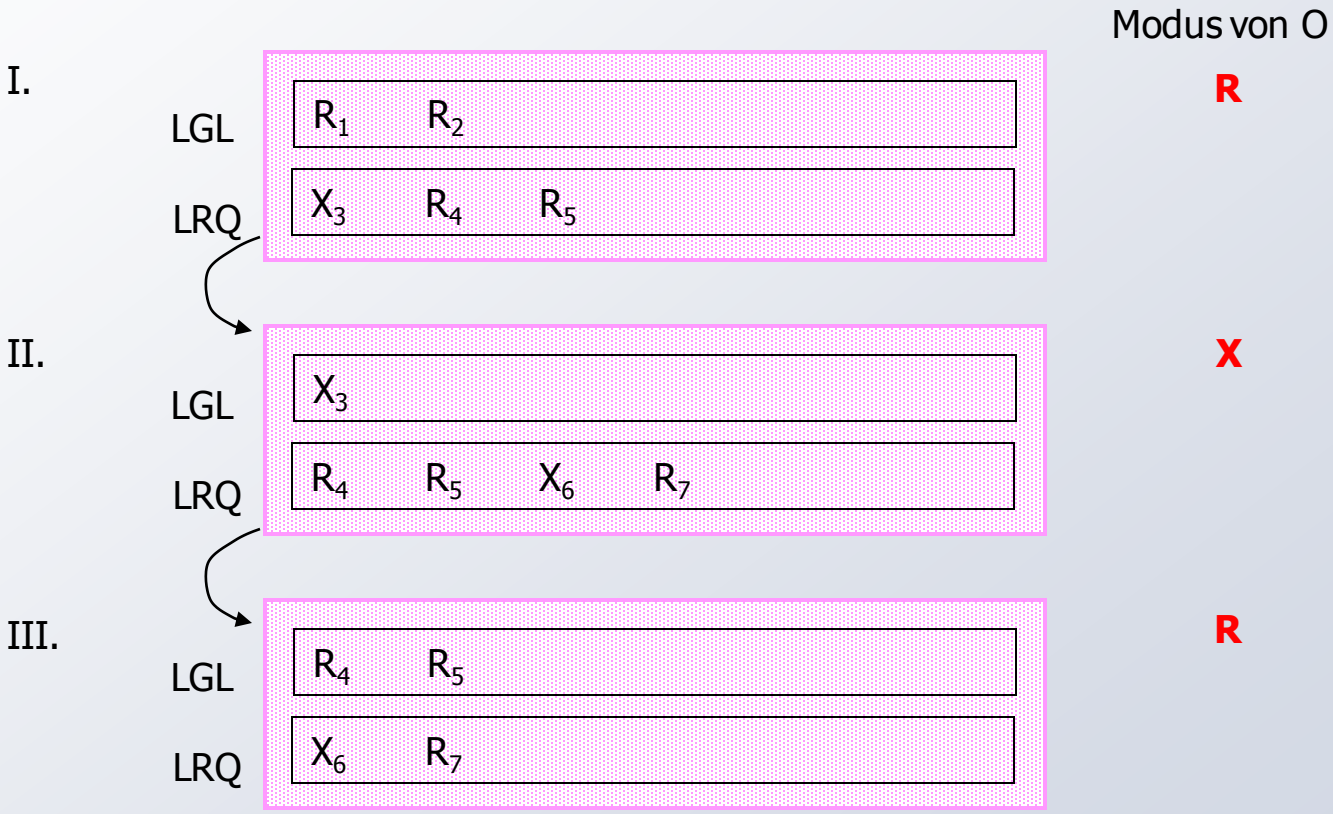
| Modus-Übergang | NL | R | X |
|---|---|---|---|
| R | R | R | - |
| X | X | - | - |

  - 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
- **Erinnerung**
  - Zweiphasigkeit: 2PL
  - Unter striktem 2PL (S2PL) werden alle exklusiven Sperren einer TA bis zu ihrer Terminierung gehalten
  - Unter starkem 2PL (SS2PL) werden alle Sperren einer TA bis zu ihrer Terminierung gehalten
- **Scheduling von Sperranforderungen**
  - pro Objekt existiert ein Liste der gewährten Sperranforderungen LGL (list of granted locks) und eine Warteschlange LRQ (lock request queue)
  - Wann wird eine Sperranforderung LR (lock request) gewährt?
  - Legende: Der Index i gibt die zeitliche Reihenfolge der Sperranforderung und gleichzeitig eine Transaktionsnummer an. Alle Sperranforderungen beziehen sich auf ein Objekt O.

# RX-Sperrverfahren (2)



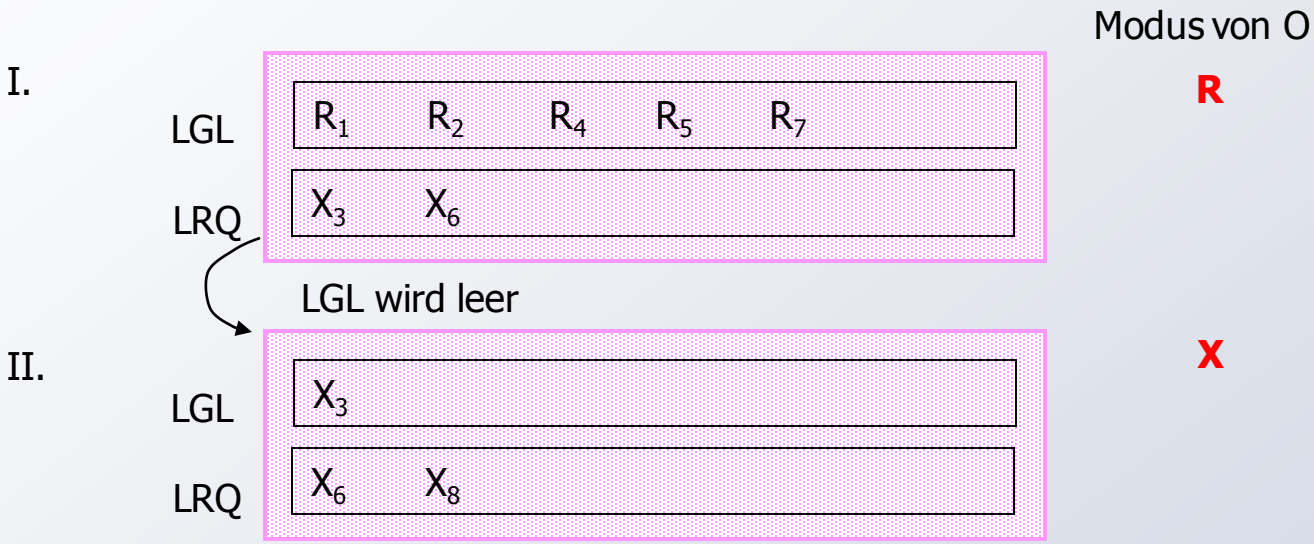
- **Faires Scheduling (First Come, First Serve, FCFS)**
  - LR wird gewährt
    - wenn LRQ leer und LR kompatibel mit dem aktuellen Modus von O,
    - wenn LGL leer, erster LR aus LRQ



# RX-Sperrverfahren (3)



- **Unfaires Scheduling**
  - LR wird gewährt, wenn LR kompatibel mit dem aktuellen Modus von O



# 2PL



## SQL-API

- An der SQL-Schnittstelle ist die Sperranforderung und -freigabe nicht sichtbar!
- Objektanforderung durch Zugriffsmodul (nach Optimierung) über OID (direkt), Scan (Fetch Next), Index usw.
- Optimierung der Sperranforderungen im DBMS
  - Mehrere Sperrgranulate anwendbar
  - Nach welchen Kriterien entscheidet das DBMS?
  - Zusätzliche Option: Sperreskalation
- Wie wird das Phantom-Problem gelöst?

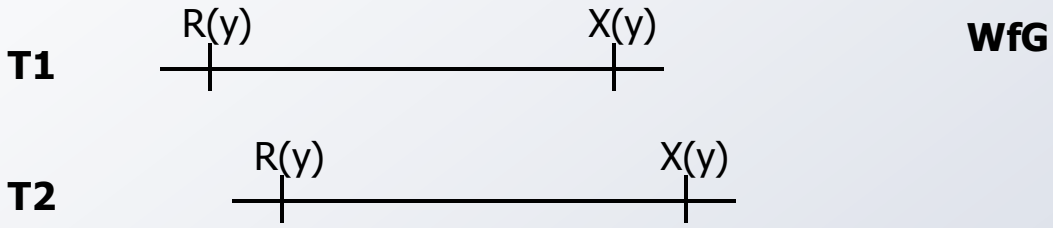
## Anwendung des 2PL-Protokolls

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

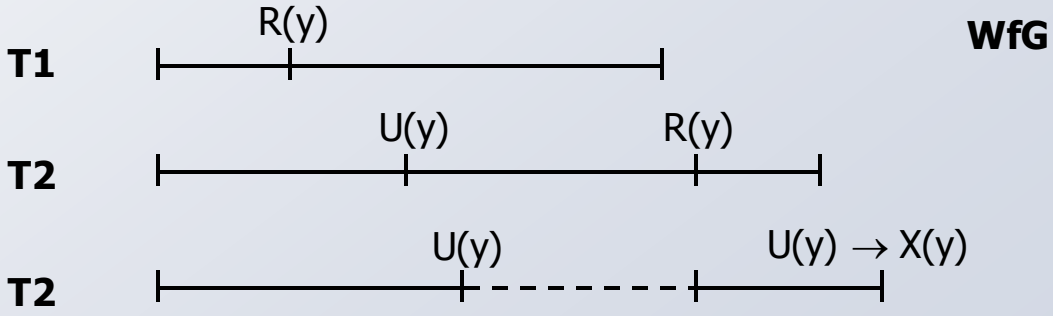
⇨ Praktischer Einsatz erfordert mindestens striktes 2PL-Protokoll

# 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 → X, andernfalls U → R (*downgrade*)
- **Wirkungsweise**



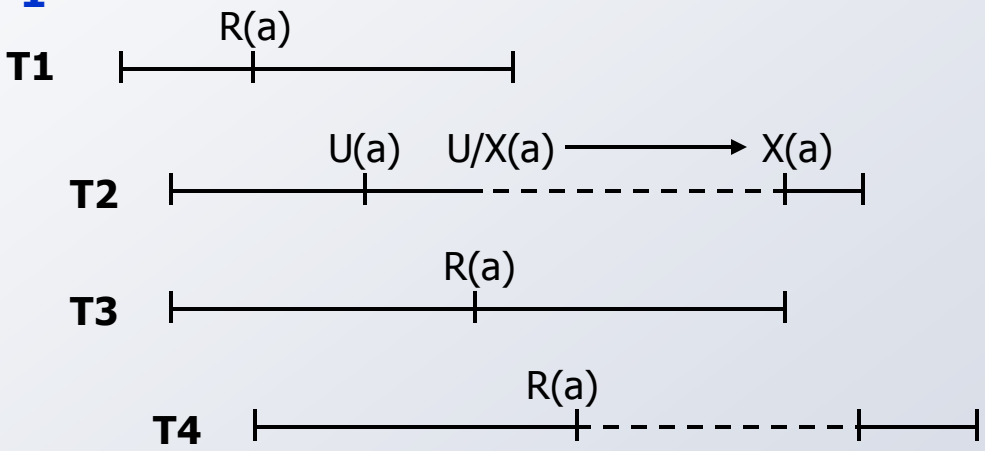
# RUX-Sperrverfahren (2)



- **Symmetrische Variante**
  - Was bewirkt eine Symmetrie bei U?

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

- **Beispiel 1**



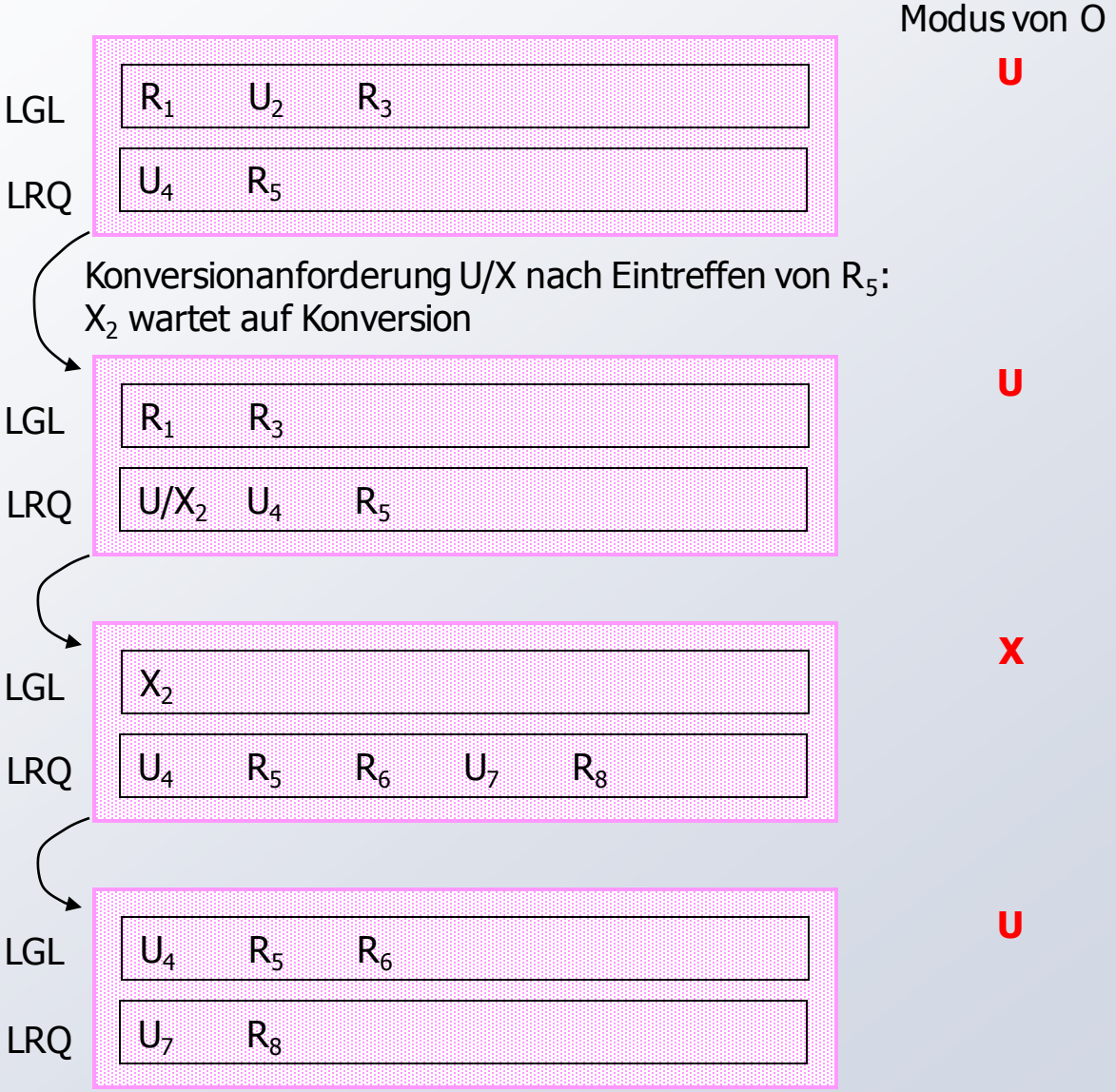

- **Beobachtung**
  - Bei Downgrade einer U-Sperre (U/R) gibt s keine Probleme, da R mit dem Modus paralleler Leser (R) verträglich ist
  - Bei Upgrade einer U-Sperre (U/X) muss die Transaktion warten, bis alle momentanen Leser ihre Sperren auf dem Objekt freigegeben haben
  - Bei Warten muss die konvertierende Transaktion an die Spitze der LRQ gesetzt werden. Warum?
  - Nach Anmeldung einer Konversion gibt es noch verschiedene Scheduling-Optionen

# RUX-Sperrverfahren (3)



- **Beispiel 2 – verschiedene Scheduling-Strategien**

Fair

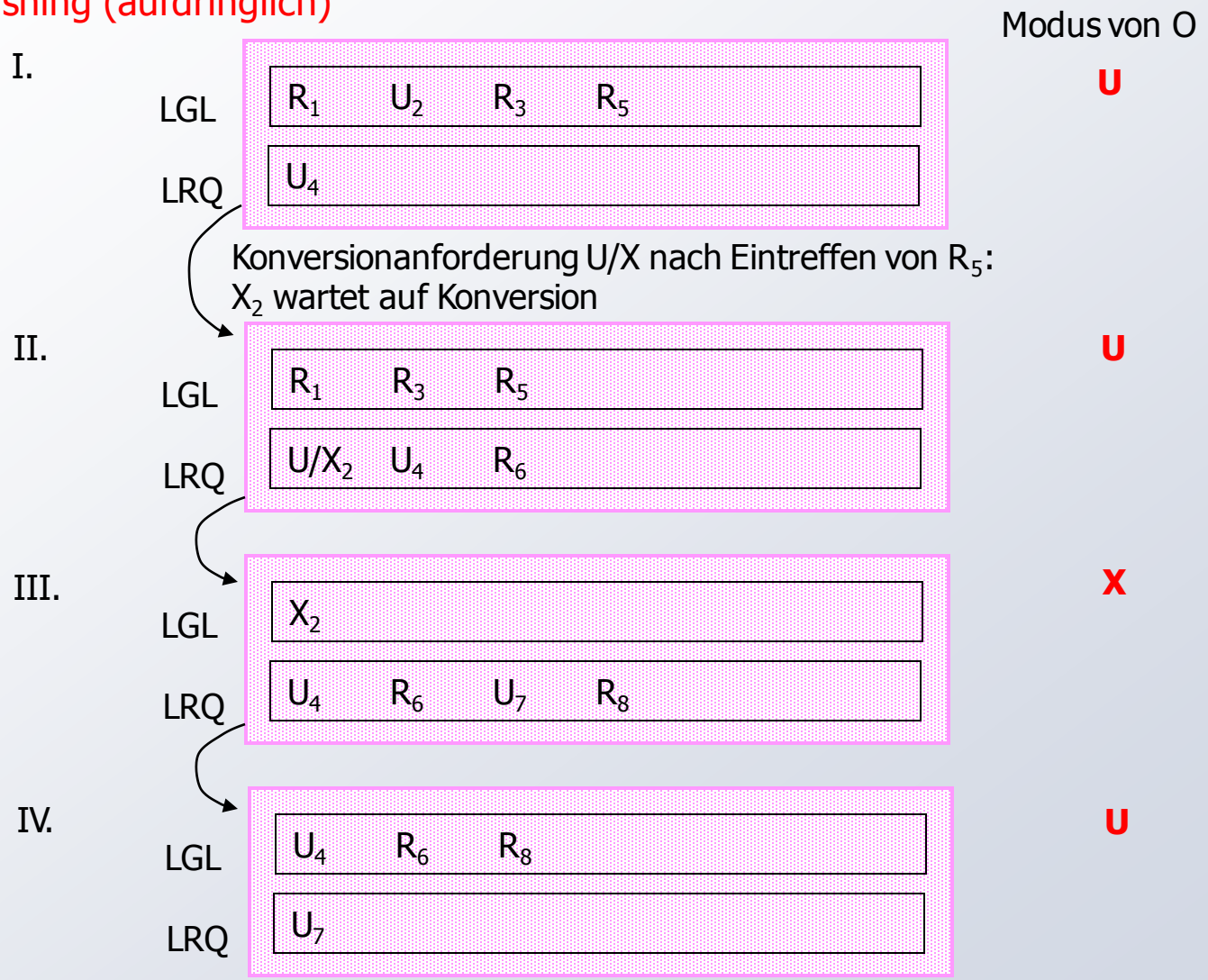


# RUX-Sperrverfahren (4)



- **Beispiel 2 – verschiedene Scheduling-Strategien (Cont.)**

Pushing (aufdringlich)



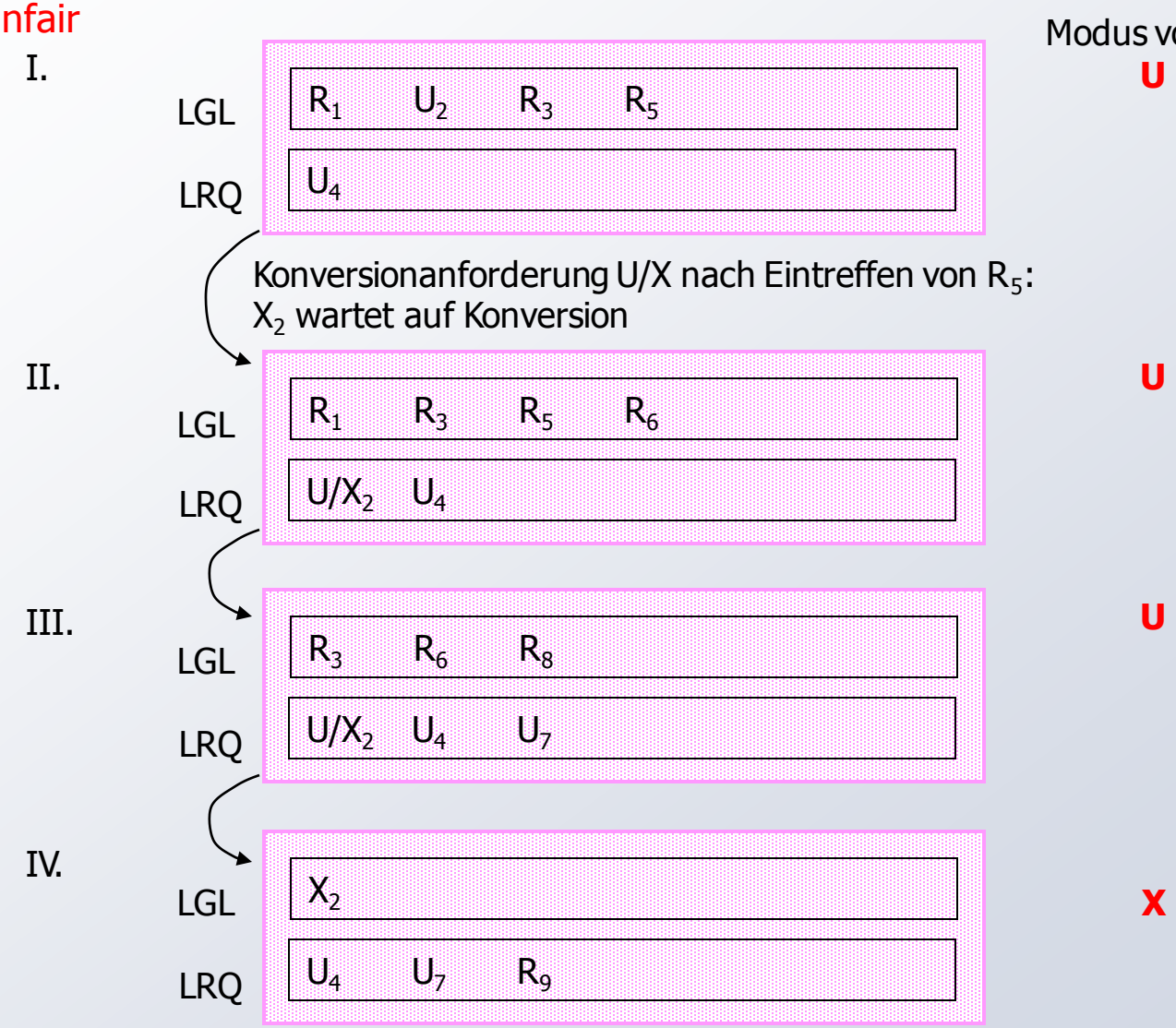

Erst wenn $U_2$ oder U4 die Konversion U/X anfordern, müssen ankommende Leser (R) in LRQ warten.

# RUX-Sperrverfahren (5)



- **Beispiel 2 – verschiedene Scheduling-Strategien (Cont.)**

Unfair




Erst wenn $U_2$ die Konversion U/X angefordert hat und LGL leer ist, müssen ankommende Leser (R) in LRQ warten.

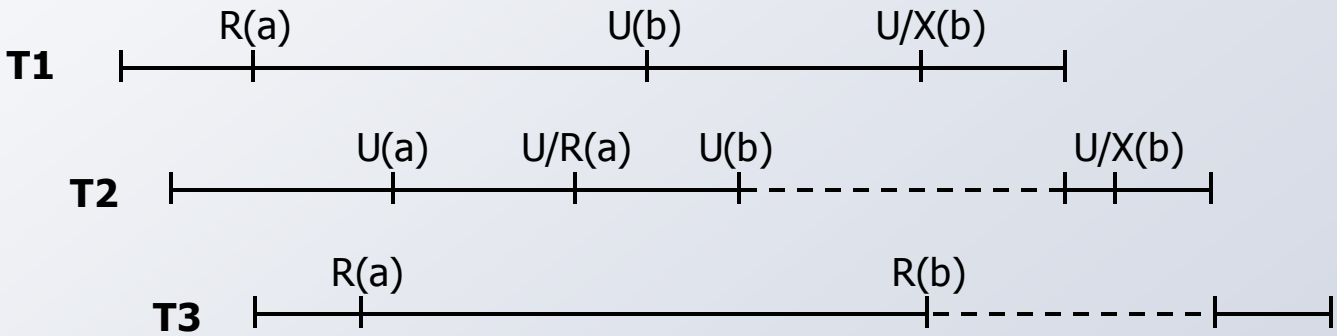
# RUX-Sperrverfahren (6)



- **Unsymmetrie bei U** (u.a. in DB2 eingesetzt)

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

- **Beispiel 3**

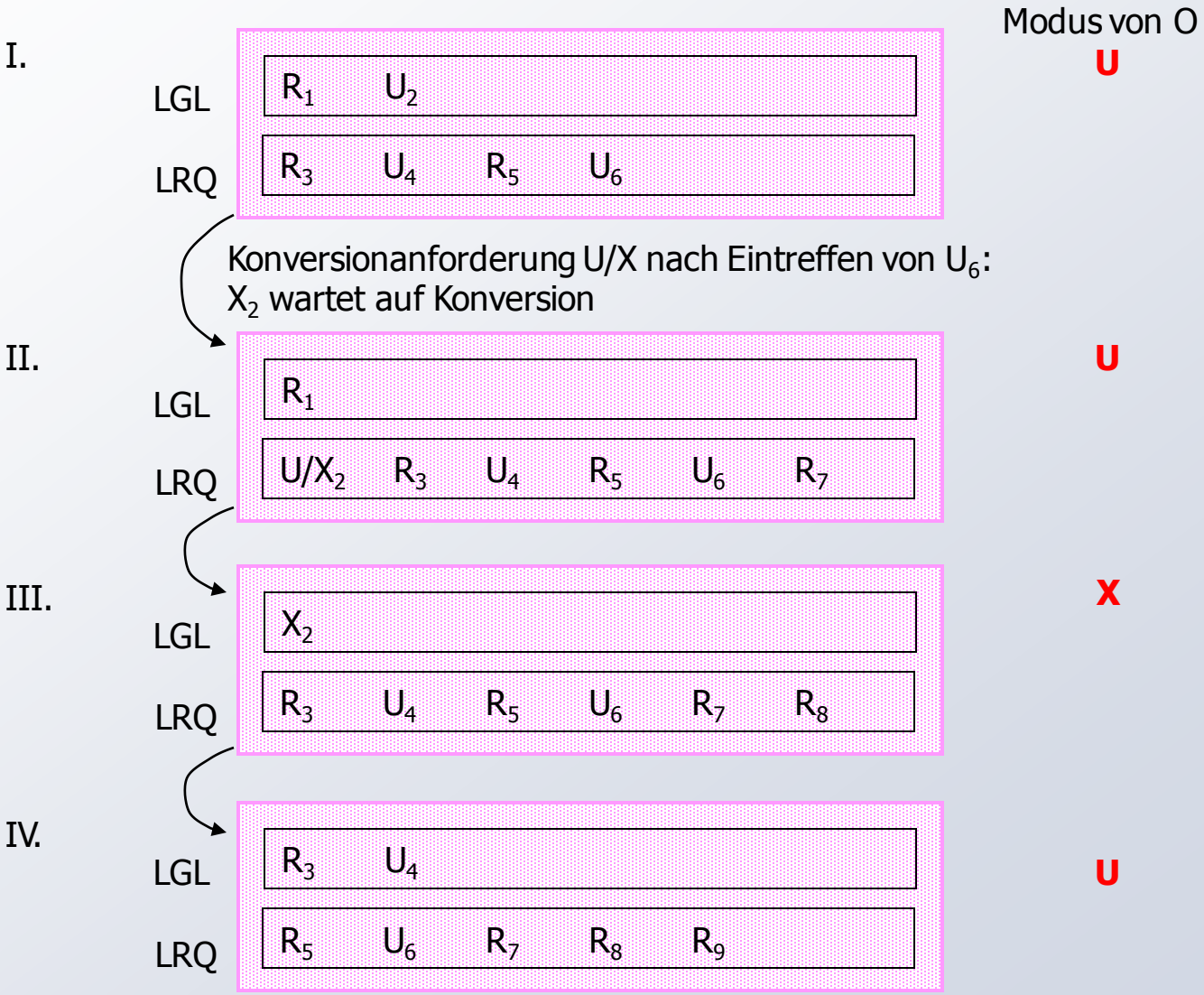



- **Beobachtung**
  - Bei Downgrade einer U-Sperre (U/R) gibt s keine Probleme, da R mit dem Modus paralleler Leser (R) verträglich ist
  - Bei Upgrade einer U-Sperre (U/X) muss die Transaktion warten, bis alle momentanen Leser ihre Sperren auf dem Objekt freigegeben haben
  - RUX-Verfahren reduziert Konversions-Deadlocks, ist aber nicht frei von ihnen. $R3(a)$ $U_2(a)$ $U_2(b)$ $U/X_2(a)$ $R_3(b)$ hätte in Beispiel 3 zu einem Deadlock geführt

# RUX-Sperrverfahren (7)



- **Beispiel 4**
  Faire Scheduling-Strategie

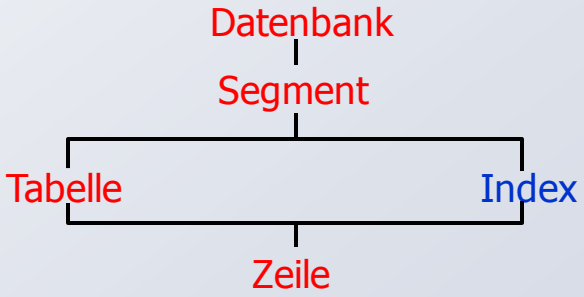


# Hierarchische Sperrverfahren



- **Sperrgranulat bestimmt Parallelität/Aufwand**
  - Feines Granulat reduziert Sperrkonflikte
  - Jedoch sind viele Sperren anzufordern und zu verwalten
- **Hierarchische Verfahren erlauben Flexibilität bei Wahl des Granulates** (*multi-granularity locking*), z. B. Synchronisation
  - langer TAs auf Tabellenebene
  - kurzer TAs auf Zeilenebene
- **Kommerzielle DBS unterstützen zumeist mindestens zweistufige Objekthierarchie, z. B**


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




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

# Beispiel einer Sperrhierarchie

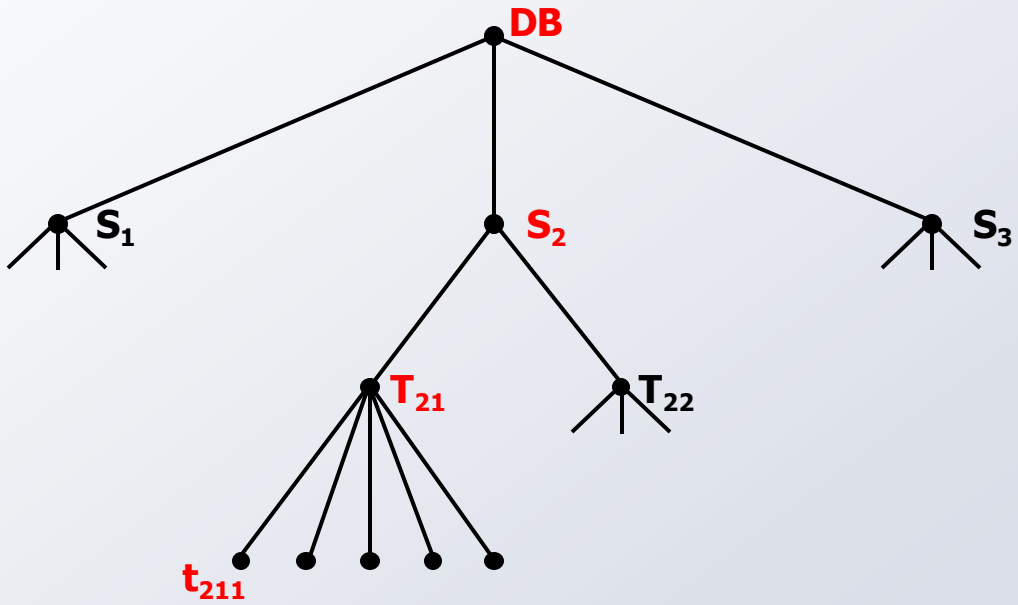


Datenbank

Dateien (Segmente)

Satztypen (Tabellen)

Sätze (Zeilen)




Wieviel Aufwand zum Sperren von

- 1 Satz
- k Sätzen
- 1 Satztyp

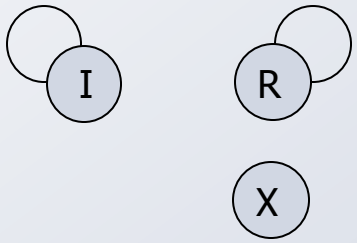
Was ist Sperr-Eskalation?

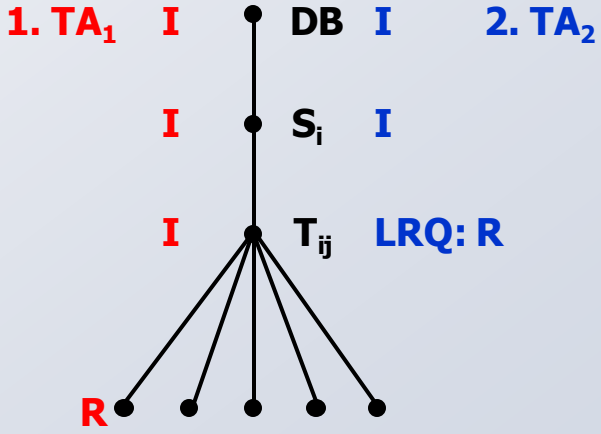
# Anwartschaftssperren



- **Mit R- und X-Sperre werden alle Nachfolgerknoten implizit mit gesperrt**
  ⇨ 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 | - | - | - |




- **Unverträglich von I- und R-Sperren**
  - I-Sperre bei parallelem Lesen zu restriktiv!
  - $TA_2$ wartet (LRQ: Lock Request Queue)

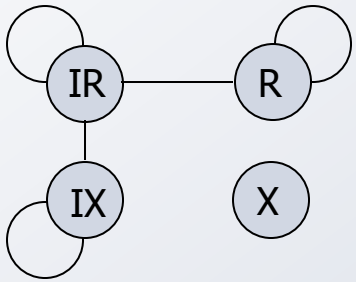
  ⇨ Zwei Arten von Anwartschaftssperren (IR und IX)

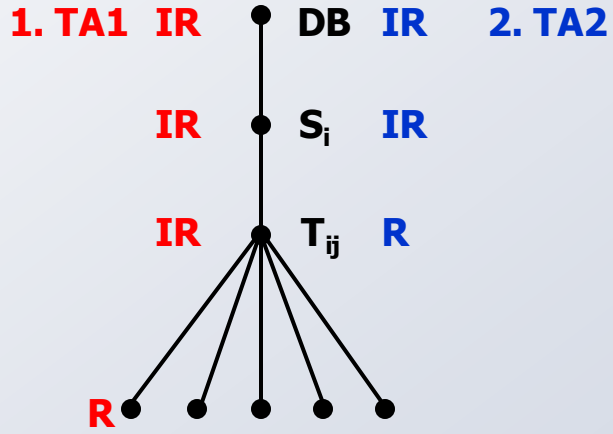
# Anwartschaftssperren (2)



- **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

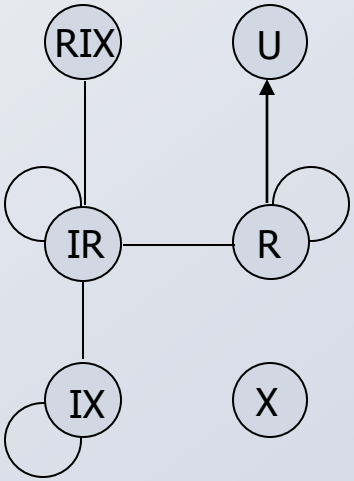
# Anwartschaftssperren (3)



## 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 → X, sonst U → R.

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




## Sperrdisziplin erforderlich

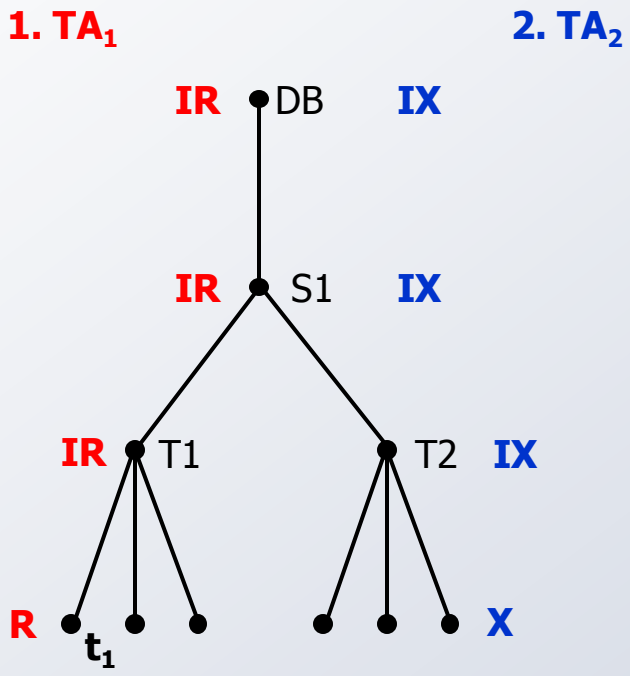
- Sperranforderungen von der Wurzel zu den Blättern
- Bevor TA eine R- oder IR-Sperre für einen Knoten anfordert, muss sie für alle Vorgängerknoten IX- oder IR-Sperren besitzen
- Bei einer X-, U-, RIX- oder IX-Anforderung müssen alle Vorgängerknoten in RIX oder IX gehalten werden
- Sperrfreigaben von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

# Hierarchische Sperrverfahren: Beispiele

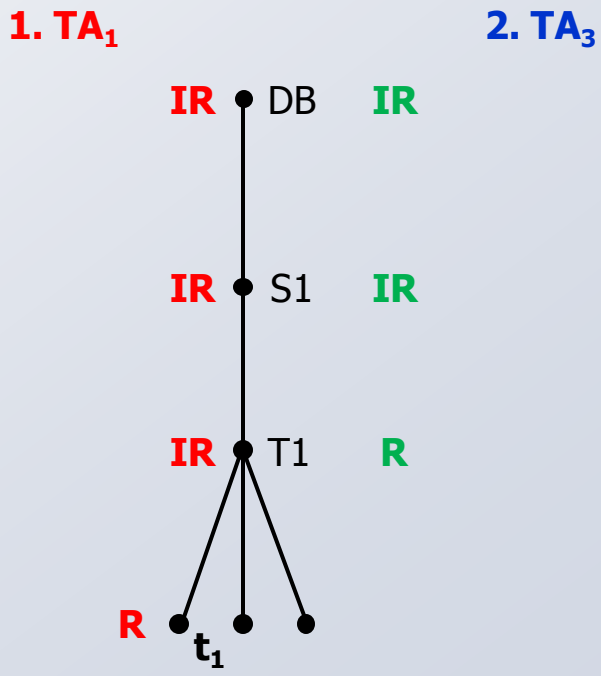


- **IR- und IX-Modus**
  - $TA_1$ liest $t_1$ in T1
  - $TA_2$ ändert Zeile in T2 (a)
  - $TA_3$ liest T1 (b)

(a)




(b)

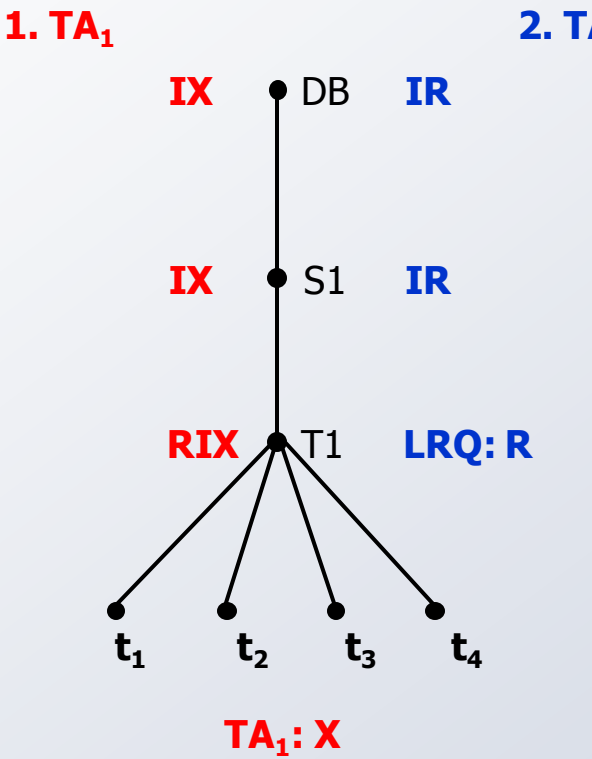


# Hierarchische Sperrverfahren: Beispiele (2)

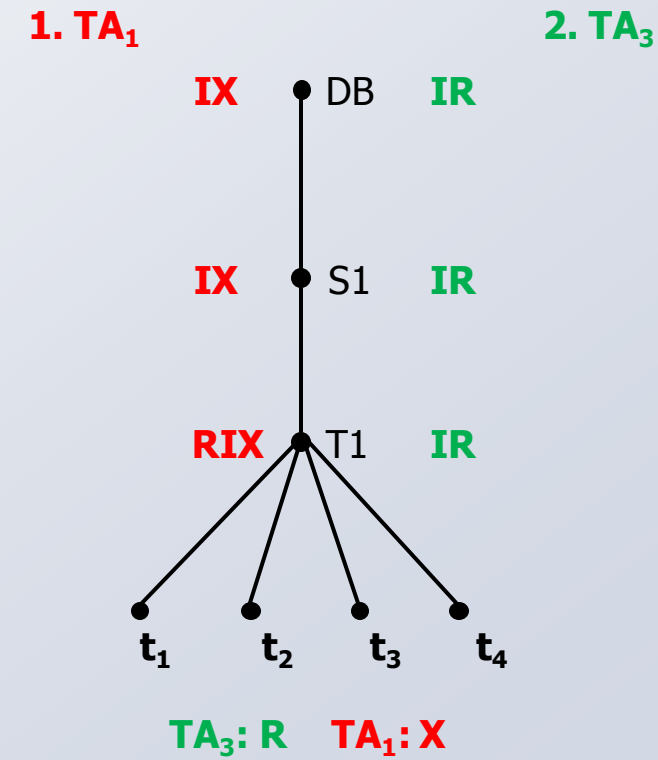


- **RIX-Modus**
  - $TA_1$ liest alle Zeilen von T1 und ändert $t_3$
  - $TA_2$ liest T1 (a)
  - $TA_3$ liest $t_2$ in T1 (b)

(a)




(b)



# 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)
  - blinde Durchsatzmaximierung?

    **Um Durchsatzziel zu erreichen:**
    **mehr** aktive TAs → **mehr** gesperrte Objekte → **höhere** Konflikt-WS →
    **längere** Sperrwartezeiten, **höhere** Deadlock-Raten → **noch mehr** aktive TAs
  - 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!?**
  - Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
  - Nutzung mehrerer Objektversionen
  - 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 verkürzten 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

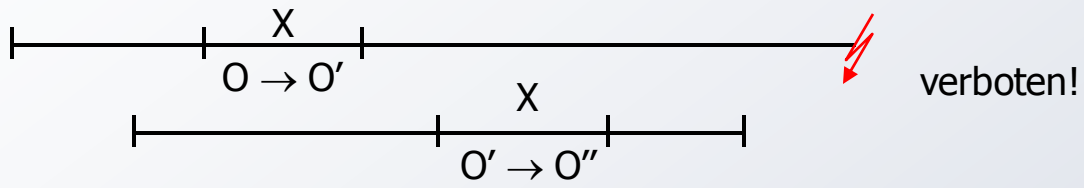
  **Ebene 0:** Transaktion T sieht Konsistenzebene 0, wenn sie nur Bedingung a erfüllt

# Konsistenzebenen (2)

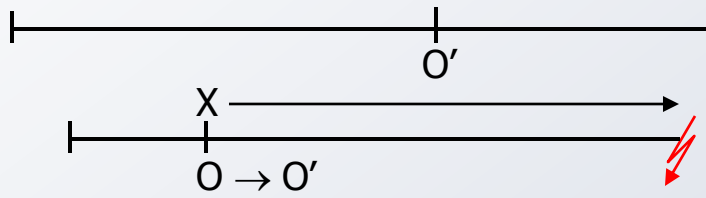


- **RX-Sperrverfahren und Konsistenzebenen** (Beispiele für nur ein Objekt O)

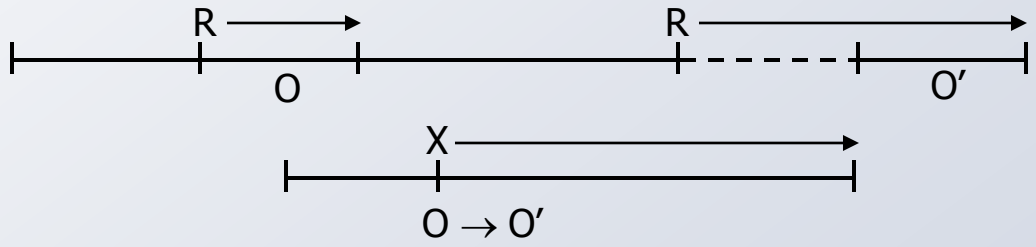
**KE 0: kurze X, keine R**



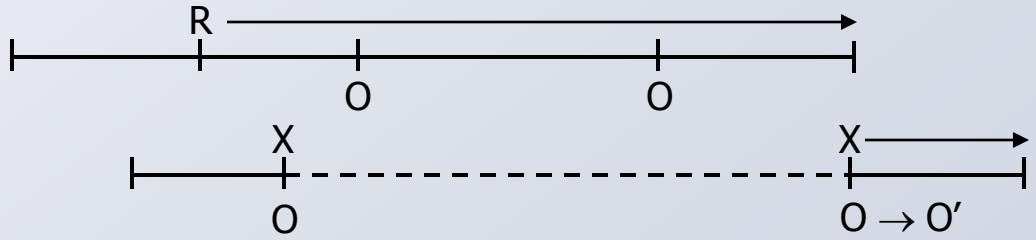

**KE 1: lange X, keine R**




**KE 2: lange X, kurze R**




**KE 3: 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 Schreibsperren, jedoch kurze Lesesperren
  - *'unrepeatable read'* möglich
- ***Konsistenzebene 1***
  - lange Schreibsperren, keine Lesesperren
  - *'dirty read'* (und *'lost update'*) möglich
- ***Konsistenzebene 0***
  - kurze Schreibsperren ('Chaos')

⇨ Kommerzielle DBS empfehlen meist Konsistenzebene 2

- **Wahlangebot**
  - Einige DBS (Oracle, DB2, MS SQL Server, ...) 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 Uncommitted** | + | + | + |
| **Read Committed** | - | + | + |
| **Repeatable Read** | - | - | + |
| **Serializable** | - | - | - |

- Default ist Serialisierbarkeit[2] (serializable)



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

# 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 KE 3 und "Serializable"?**
  - **Repeatable Read** - Sperren von vorhandenen Objekten

    Datenstruktur: O1, O2, . . . Zugriff mit Get Next

    1. Lesefolge von T1:

       Einfügen von T2:

    2. Lesefolge von T1:
  - **Serializable** - garantiert Abwesenheit von Phantomen

    1. Lesefolge von T1:

       Einfügen von T2:

    2. Lesefolge von T1:

# Sperrverfahren mit Versionen (RAX)

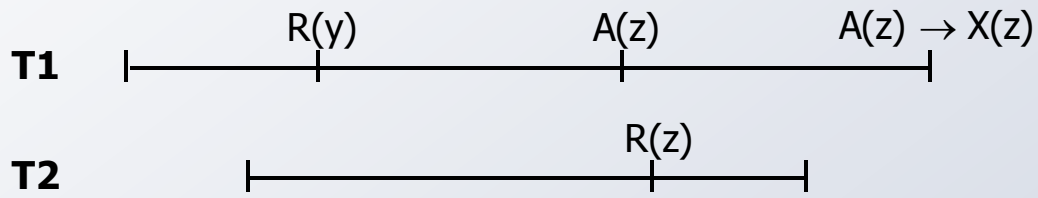


- **Kompatibilitätsmatrix**

| | R | A | X |
|---|---|---|---|
| R | + | ⊕ | - |
| A | ⊕ | - | - |
| X | - | - | - |

A = Analyse

- **Ablaufbeispiel**

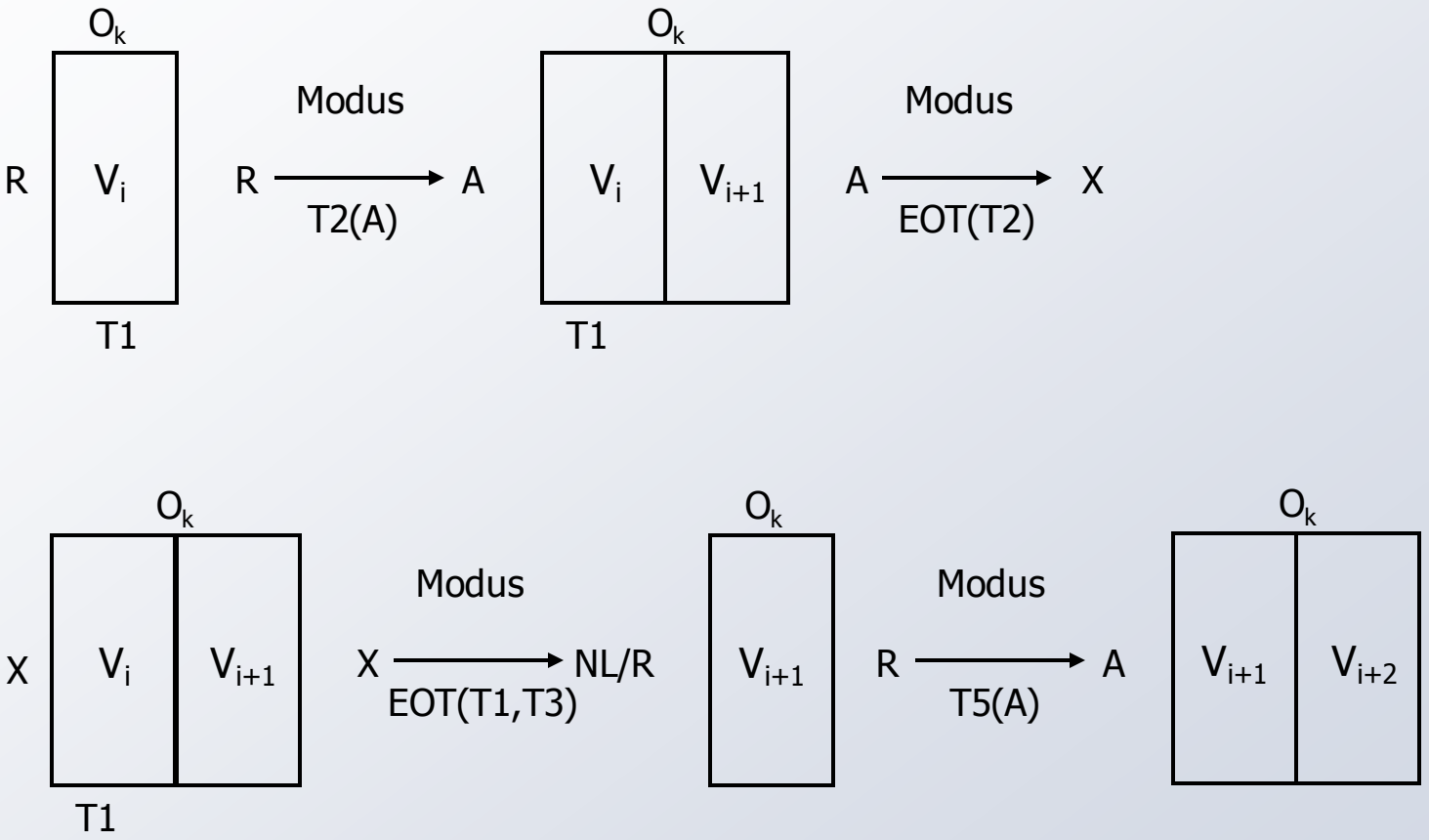



**RAX: T2 → T1**

# RAX (2)



- **Änderungen erfolgen in temporärer Objektkopie**




LRQ(R):

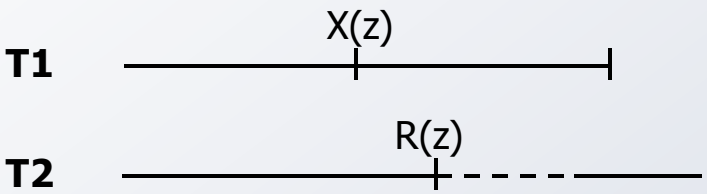
LRQ(A):

# RAX (3)

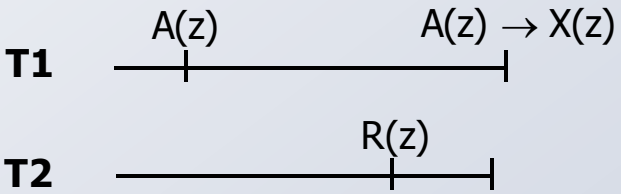


## Eigenschaften von RAX

- 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 Serialisierbarkeits-reihenfolge




**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-TAs 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)

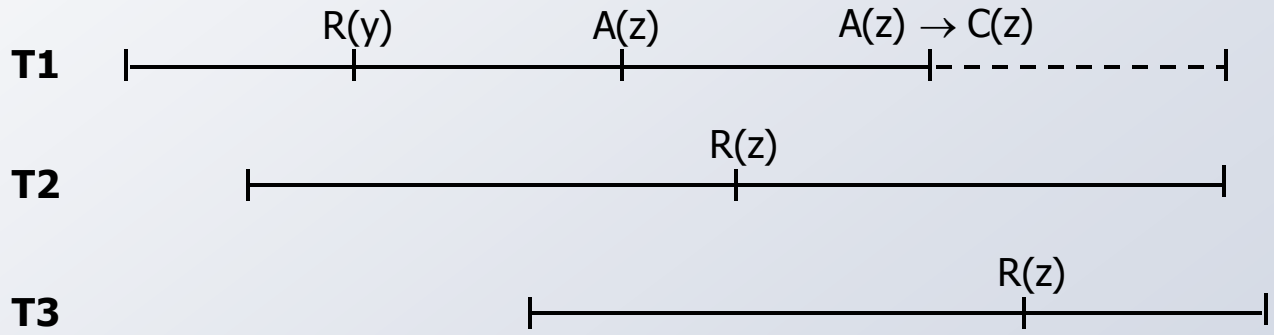


- **Kompatibilitätsmatrix**

| | R | A | C |
|---|---|---|---|
| R | + | + | ⊕ |
| A | + | - | - |
| C | ⊕ | - | - |

C = Commit

- **Ablaufbeispiel**

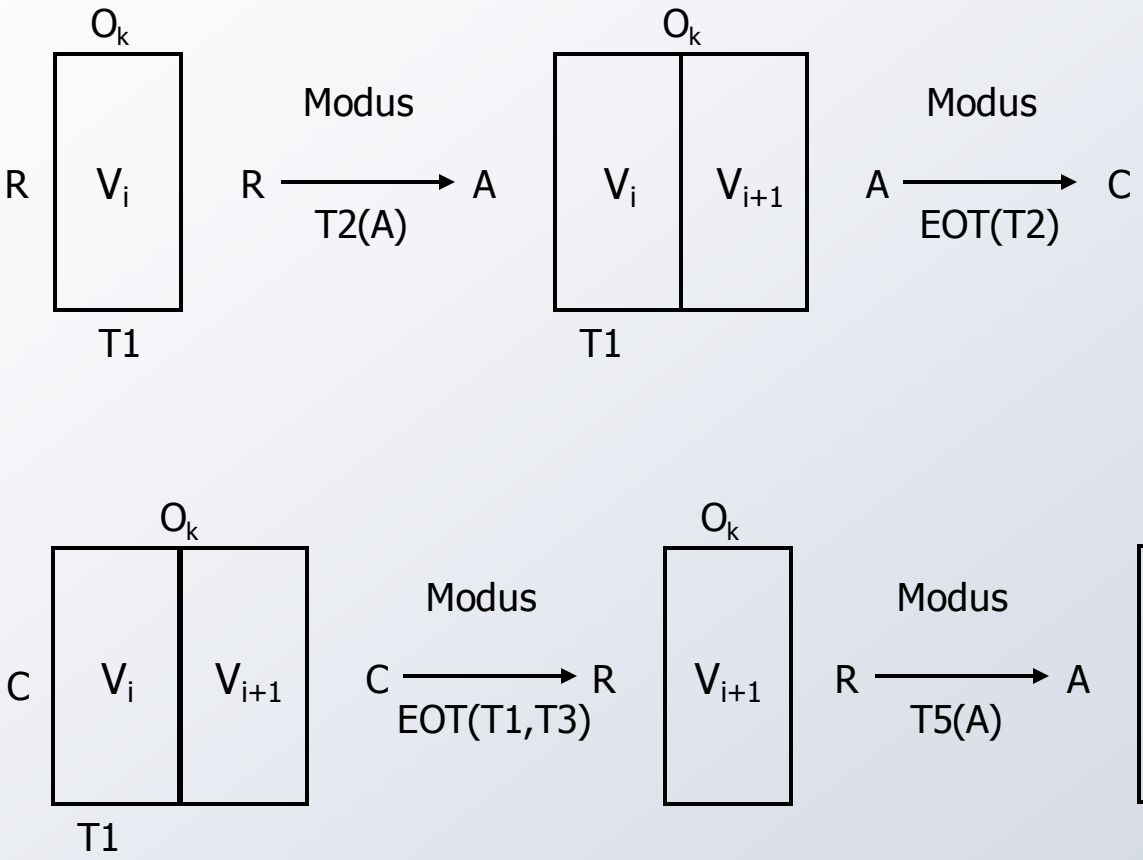



**RAC: T2 → T1 → T3**

# RAC (2)



- **Änderungen erfolgen ebenfalls in temporärer Objektkopie**



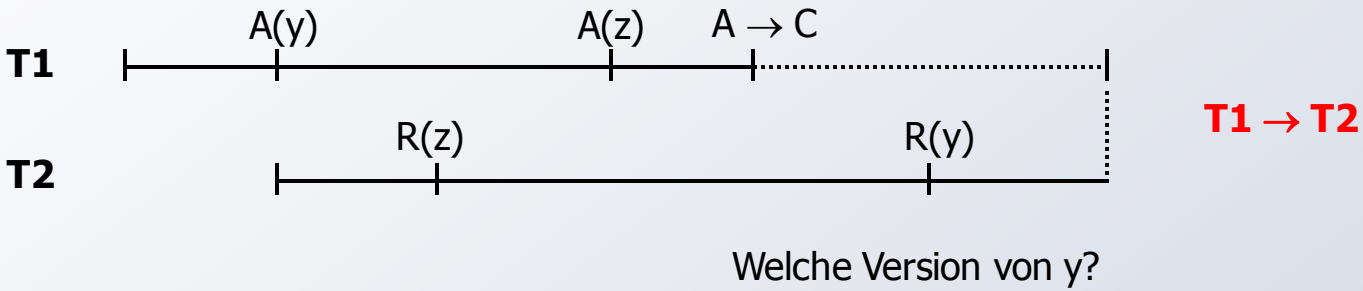

LRQ(A):

# RAC (3)



## Eigenschaften von RAC

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




⇨ 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-TAs, die auf C-Sperre laufen, müssen warten, bis alle Leser der alten Version beendet sind - weil nur 2 Versionen

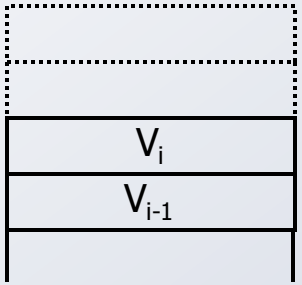
⇨ ABHILFE: allgemeines Mehrversionen-Verfahren

# Mehrversionen-Verfahren



- **Änderungs-TAs erzeugen neue Objektversionen**
  - Es kann immer nur eine neue Version pro Objekt erzeugt werden
  - Sie wird bei EOT der TA freigegeben
- **Lese-TAs sehen den bei ihrem BOT gültigen DB-Zustand**
  - Sie greifen immer auf die jüngsten Objektversionen zu, die bei ihrem BOT freigegeben waren

    ⇨ Eine Objektversion, auf die ein Leser noch zugreifen könnte, darf nicht gelöscht werden
  - Sie setzen und beachten keine Sperren
  - Es gibt keine Blockierungen und Rücksetzungen für Lese-TAs, dafür ggf. Zugriff auf veraltete Objektversionen
  - Wie erkennt das DBS Leser?
- **Beispiel für Objekt $O_k$**




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)



- **Kompatibilitätsmatrix für Objekt O**

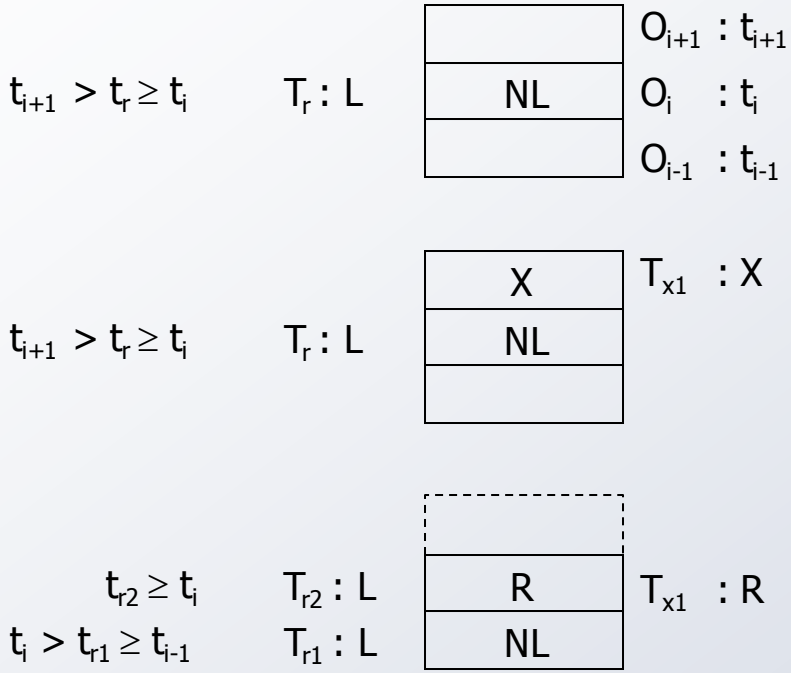
| | NL | R | X |
|---|---|---|---|
| L | + | + | - |
| R | + | + | - |
| X | + | - | - |

- **Auswahl der Objektversion**
  - Der Zugriff im Modus L bezieht sich auf die bei BOT($T_r$) freigegebene Version:
    - $T_r$: Lese-TA (Read-only-Operationen)
    - $T_x$: Schreib-TA (Read/Write-Operationen)
    - Commit-Zeitpunkte der Versionen: ... $O_i = t_i$, $O_{i+1} = t_{i+1}$ ...
    - BOT($T_r$) = $t_r$
      Wenn $t_{i+1} > t_r \geq t_i$, dann bezieht sich der L-Zugriff auf = $O_i$

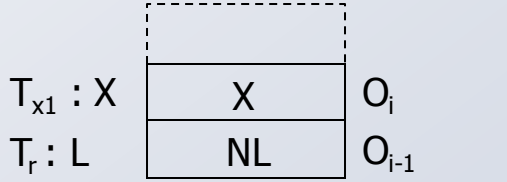
# Mehrversionen-Verfahren (3)



## Sperrsituationen




Zugriff auf O in der Reihenfolge $T_{x1}$, $T_{x2}$, $T_r$ ($t_r \geq t_{i-1}$)

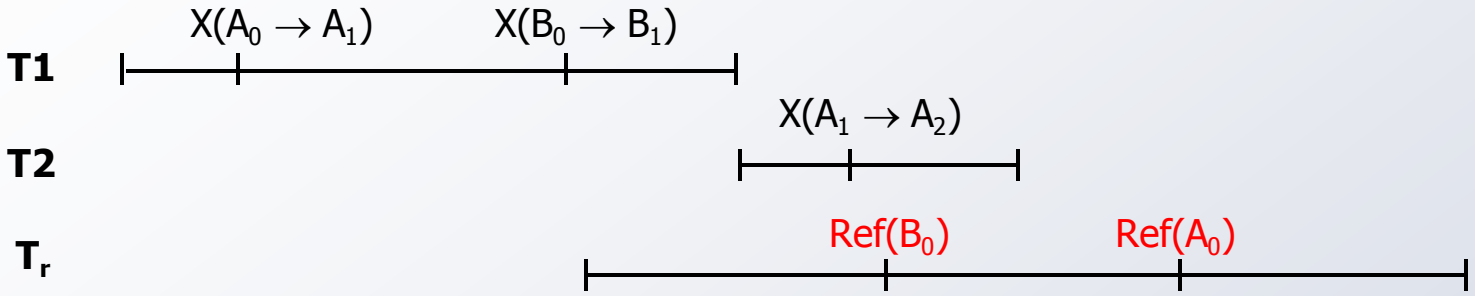



⇨ Mit R wird immer die jüngste freigegebene Version gelesen; solange Lock(O,R) gilt, kann keine neue Version erzeugt werden

# Mehrversionen-Verfahren (4)



- **Ablaufszenario**

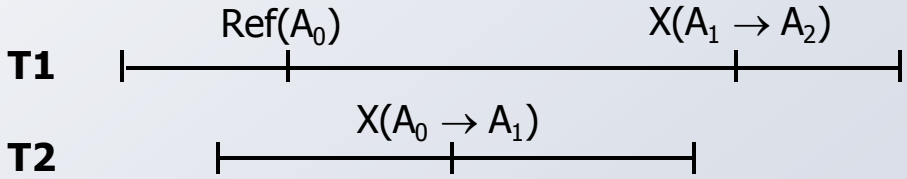



- **Konsequenz**
  - Lese-TAs werden bei der Synchronisation nicht mehr berücksichtigt
  - Schreib-TAs werden mit ihren Lese- und Schreib-Operationen untereinander über ein allgemeines Verfahren (Sperren, OCC, . . .) synchronisiert

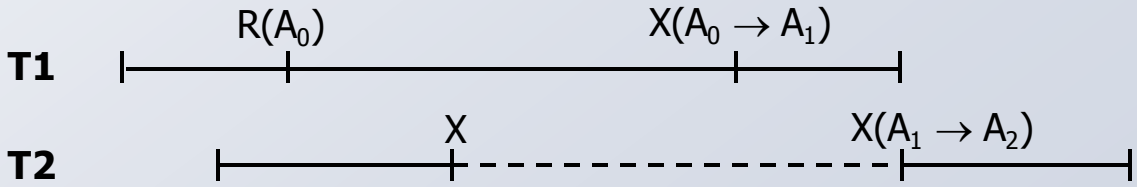
  ⇨ deutlich weniger Synchronisationskonflikte

  - Hypothetische Annahme: Ein Schreiber greift bei seinen Lese-Operationen wie ein Leser zu (ohne Synchronisation). Ist der Schreiber serialisierbar?

**L:**



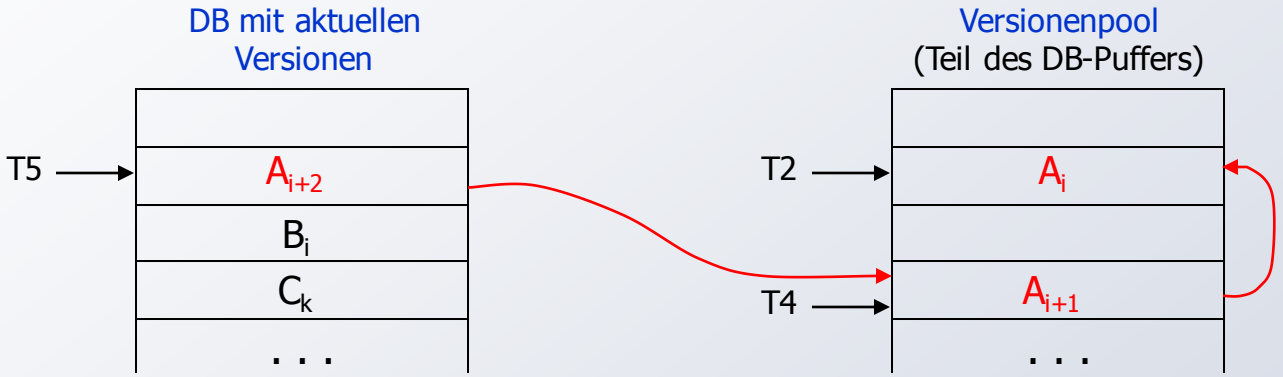

**R:**



# Mehrversionen-Verfahren (5)



- **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?

# Snapshot Isolation (SI)



## Regeln

1) Wenn eine TA T einen Datenobjekt x liest, dann liefert das System immer den jüngsten transaktionskonsistenten Wert von x zum Zeitpunkt von BOT(T). Dieser Wert von x stammt von der jüngsten erfolgreich beendeten TA unter allen TA, die x geschrieben haben und beendet waren, bevor T gestartet wurde. Wenn T selbst den Wert von x verändert, sieht es diesen geänderten Wert.

2) Wenn zwei TA $T_i$ und $T_j$ zeitlich überlappen und dasselbe Datenobjekt schreiben, verhindert das System, dass beide erfolgreich Commit ausführen können. In einem solchen Fall gilt „First Committer Wins".

## Notation

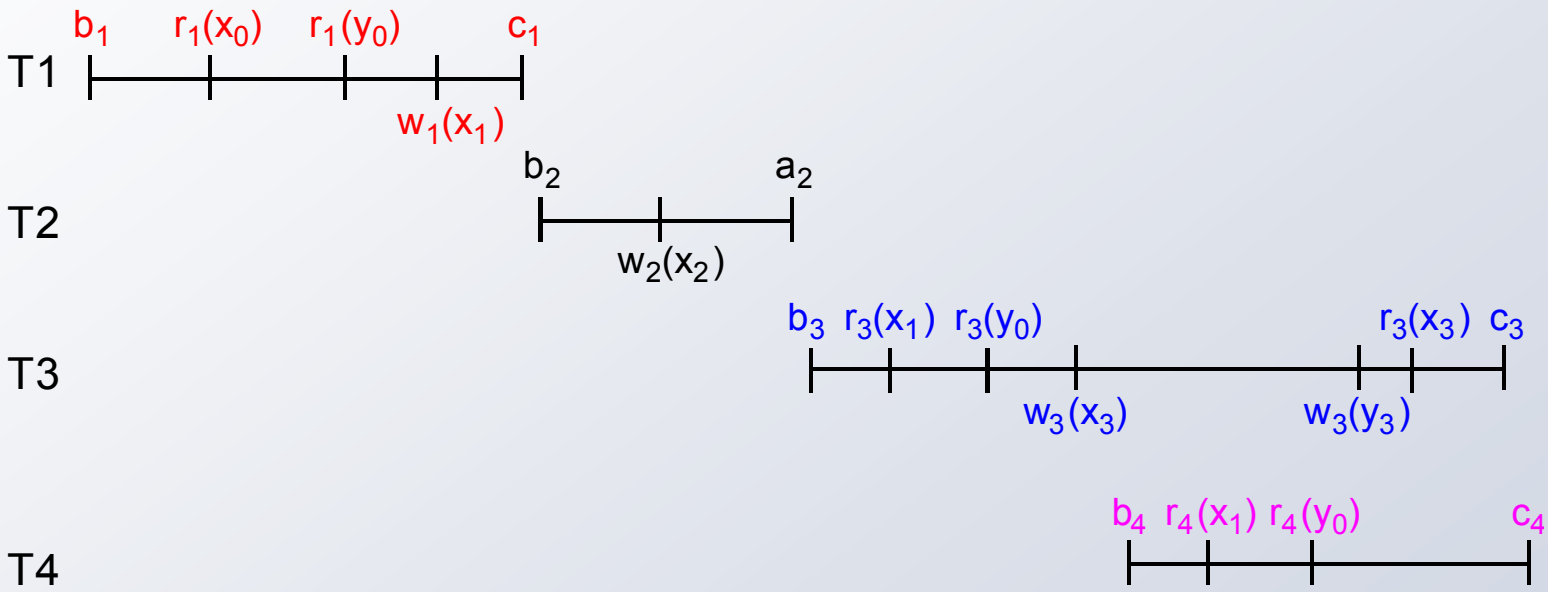
- $r_i$ ist eine Leseoperation innerhalb von $T_i$
- Wenn $T_j$ das Datenobjekt x schreibt, wird das durch $w_j(x_j)$ ausgedrückt
- Achtung: Der Index bei $x_j$ zeigt nicht die Versionsnummer an. In einem Schedule bedeutet $r_5(x_3)$, dass TA $T_5$ den Wert von x liest, den $T_3$ geschrieben hat

# Snapshot Isolation (2)



- **Allgemeine Wirkungsweise der SI (Synonym: Row Versioning)**

S1: $b_1$ $r_1(x_0)$ $r_1(y_0)$ $w_1(x_1)$ $c_1$ $b_2$ $w_2(x_2)$ $a_2$ $b_3$ $r_3(x_1)$ $r_3(y_0)$ $w_3(x_3)$
$b_4$ $r_4(x_1)$ $r_4(y_0)$ $w_3(y_3)$ $r_3(x_3)$ $c_3$ $c_4$



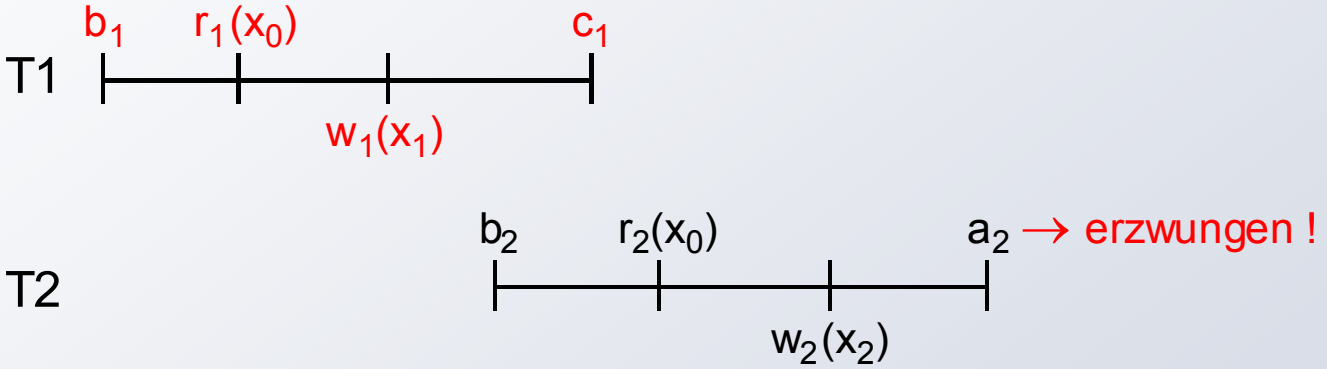

- **Besonderheiten**
  - Repeatable Read von T3 (Regel 1)
  - T4 liest $x_1$, obwohl $x_3$ schon existiert (kein Warten wie bei Sperren)

# Snapshot Isolation (3)



- **First Committer Wins!**

S2: $b_1$ $r_1(x_0)$ $w_1(x_1)$ $b_2$ $c_1$ $r_2(x_0)$ $w_2(x_2)$ $a_2$

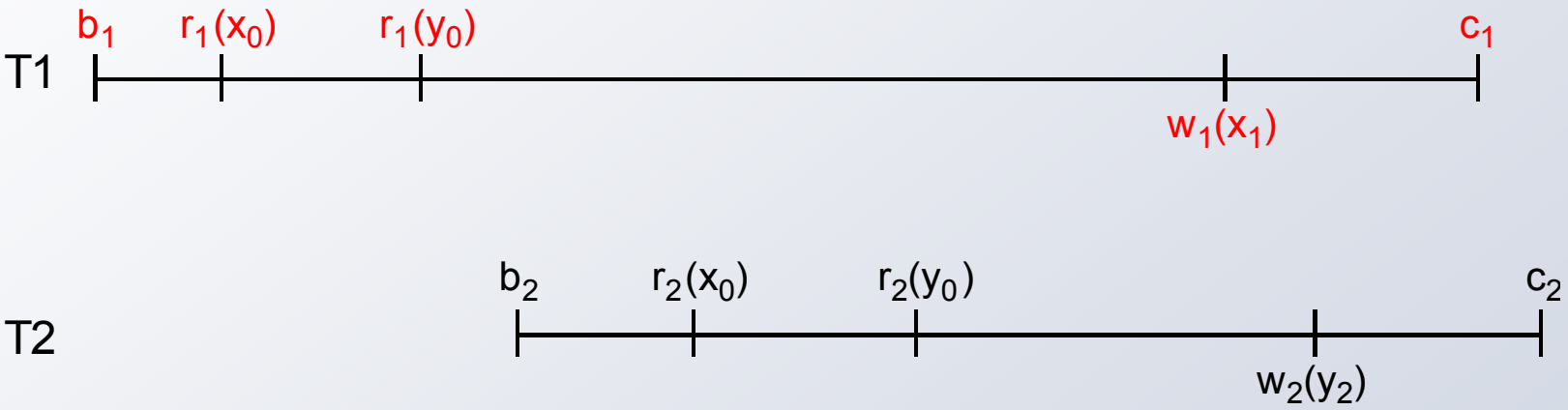



➥ Commit von T2 würde ein Lost Update von T1 implizieren!

# Snapshot Isolation (4)

- **Anomalie „Write Skew“:**
  SI garantiert nicht Serlialisierbarkeit!

  S3: $b_1$ $r_1(x_0)$ $r_1(y_0)$ $b_2$ $r_2(x_0)$ $r_2(y_0)$ $w_1(x_1)$ $w_2(y_2)$ $c_1$ $c_2$




- **Annahme**
  - x und y seien die Kontostände zweier Konten eines Kunden. Eine Geschäftsregel verlangt, dass die Gesamtsumme solcher zusammengehöriger Konten immer positiv ist.
  - Wenn eine TA erkennt, dass die Gesamtsumme negativ ist, führt sie Abort aus
  - Beispiel: $x_0 = 100$; $y_0 = 200$; T1 --> x - 150; T2 --> y - 175;

# Snapshot Isolation (5)



- **Einsatz der SI in kommerziellen DBMS**
  - "SET TRANSACTION ISOLATION LEVEL SNAPSHOT" in Microsoft SQL Server 2005
  - Ähnliche Einsatzmöglichkeit in Interbase und Oracle Berkeley DB
  - Andere Plattformen wie PostgreSQL (ab Version 7) und Oracle wenden Snapshot Isolation an, wenn der Benutzer "SET ISOLATION LEVEL SERIALIZABLE" spezifiziert
  - Achtung: SI erlaubt nicht-serialisierbare TA-Ausführungen!

# Prädikatssperren[3]

- **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 $\in$ {read, write}

  UNLOCK (R, P)

  - Lock (R, P, write) sperrt alle möglichen Sätze von R exklusiv, die Prädikat P erfüllen
- **Beispiel**

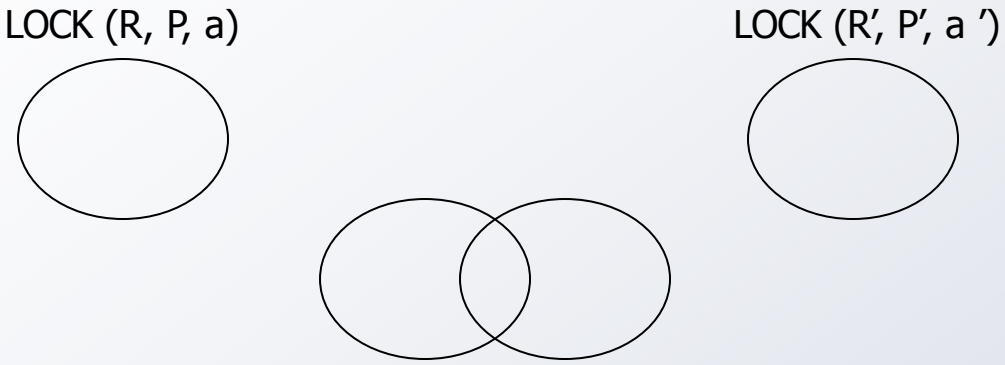
| **T1:** | | **T2:** | |
|---|---|---|---|
| | LOCK(R1, P1, read) | | . . . |
| | LOCK(R2, P2, write) | | LOCK(R2, P3, write) |
| | LOCK(R1, P5, write) | | LOCK(R1, P4, read) |
| | . . . | | . . . |

- **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 $\Theta$ Wert) {$\wedge$, $\vee$} (. . .

3. Eswaran, K.P. et al.: The Notions of Consistency and Predicate Locks in a Database System, in: Communications of the ACM 19:11, 1976, 624-633.

# Prädikatssperren (2)

- **Entscheidungsprozedur**




1. Wenn R ≠ R', kein Konflikt
2. Wenn a = read und a' = read, kein Konflikt
3. Wenn P(t) ∧ P'(t) = TRUE für irgendein t, dann besteht ein Konflikt

T1:
LOCK (Pers, Alter > 50, read)

T2:
LOCK (Pers, Pnr = 4711, write)

⇨ Entscheidung:

- **Nachteile**
  - Erfüllbarkeitstest: Aufwändige 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=TRUE entspricht einer Tabellensperre ⇨ große Sperrgranulate, geringe Parallelität

# Prädikatssperren (3)

- **Effizientere Implementierung: Präzisionssperren**[4]
  - 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 TAs 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 TAs

    (Pers: 4711, 'Müller', 30, 'Prog.', 70000)
    (Abt: K51, 'DBS', . . .)
    . . .
- **Leseanforderung (Prädikat P):**
  - für jeden Satz der Update-Liste ist zu prüfen, ob er 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



4. 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 TAs T1 und T2, die **kommutative Operationen** „erhöhe um x" und „vermindere um x" verwenden, könnten z.B. in folgenden Reihenfolgen ausgeführt werden:

| **Schedule S1** | **Schedule S2** |
|---|---|
| 1. erhöhe (T1,K1,x1) | 1. erhöhe (T1,K1,x1) |
| 2. vermindere (T1,K2,x1) | 2. erhöhe (T2,K1,x2) |
| 3. erhöhe (T2,K1,x2) | 3. vermindere (T2,K2,x2) |
| 4. vermindere (T2,K2,x2) | 4. vermindere (T1,K2,x1) |

# Synchronisation auf Objekten (2)



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

$\underbrace{<r_1[K_1]\ w_1[K_1]}_{\text{erhöhe}(T_1,K_1,x_1)}$ $\underbrace{<r_1[K_2]\ w_1[K_2]}_{\text{vermindere}(T_1,K_2,x_1)}$ $\underbrace{<r_2[K_1]\ w_2[K_1]}_{\text{erhöhe}(T_2,K_1,x_2)}$ $\underbrace{<r_2[K_2]\ w_2[K_2]>.}_{\text{vermindere}(T_2,K_2,x_2)}$

⇨ Schedule S2 ist hingegen mit rein syntaktischen Verfahren nicht erzeugbar

$\underbrace{<r_1[K_1]\ w_1[K_1]}_{\text{erhöhe}(T_1,K_1,x_1)}$ $\underbrace{<r_2[K_1]\ w_2[K_1]}_{\text{erhöhe}(T_2,K_1,x_2)}$ $\underbrace{<r_2[K_2]\ w_2[K_2]}_{\text{vermindere}(T_2,K_2,x_2)}$ $\underbrace{<r_1[K_2]\ w_1[K_2]>.}_{\text{vermindere}(T_1,K_2,x_1)}$

# Synchronisation von High-Traffic-Objekten



- ***High-Traffic*-Objekte**
  - auch *Hot Spots* genannt: Datenelemente, auf die viele TAs 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 Sperren und angepassten Scheduling-Verfahren 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';

if anzahl ≥ 2 then anzahl := anzahl-2;

exec sql    update Flüge
            set FreiePlätze = :anzahl
            where FlugNr = 'LH127';
```

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

  - Verbesserung

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

aber: Prüfung der Integritätsbedingung (FreiePlätze ≥ 0) durch das DBMS

⇨ 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 Attributzugriffs**

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 ein 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 alle anderen Feldzugriffe Schreibsperren. 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 TAs 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 ?<br>Commit<br>FP > 2<br>FP := FP-2 | | |
| | | FP > 8 ? | 10 |
| Commit<br>FP > 5 ?<br>FP := FP-5 | | | |
| | | Commit<br>FP > 8 ? | 5 |

# Escrow-Ansatz[5]

- **High-Traffic-Objekte**
  - Deklaration als Escrow-Felder
  - Benutzung spezieller Operationen
    - Anforderung einer bestimmten Wertmenge

      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 Spezifikation eines Bereichstests 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 TAs 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

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

# Escrow-Ansatz (2)



- **Beispiel**
  - Zugriffe auf Feld mit LO=0, HI=100 (Anzahl freier Plätze)

| Anforderungen/Rückgaben | | | | Wertintervall | | |
|---|---|---|---|---|---|---|
| T1 | 2 | T3 | T4 | INF | Q | SUP |
| | | | | 15 | 15 | 15 |
| -5 | | | | 10 | 10 | 15 |
| | -8 | | | 2 | 2 | 15 |
| | | +4 | | 2 | 6 | 19 |
| | | | -3 | | | |
| commit | | | | 2 | 6 | 14 |
| | | commit | | 6 | 6 | 14 |
| | abort | | | 14 | 14 | 14 |

- **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

# Klassifikation von Synchronisationsverfahren



- **Gemeinsame Ziele**
  - Erhöhung der Parallelität
  - Reduktion von Behinderungen/Blockierungen
  - Einfache Verwaltung
- **Erhöhung der Parallelität durch Objektreplikation**
  - Kopien: temporär, privat, nicht freigegeben
  - Versionen: permanent, mehrbenutzbar, freigegeben

| #Kopien \ #Versionen | 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!

# Implementierungsaspekte - Datenstrukturen

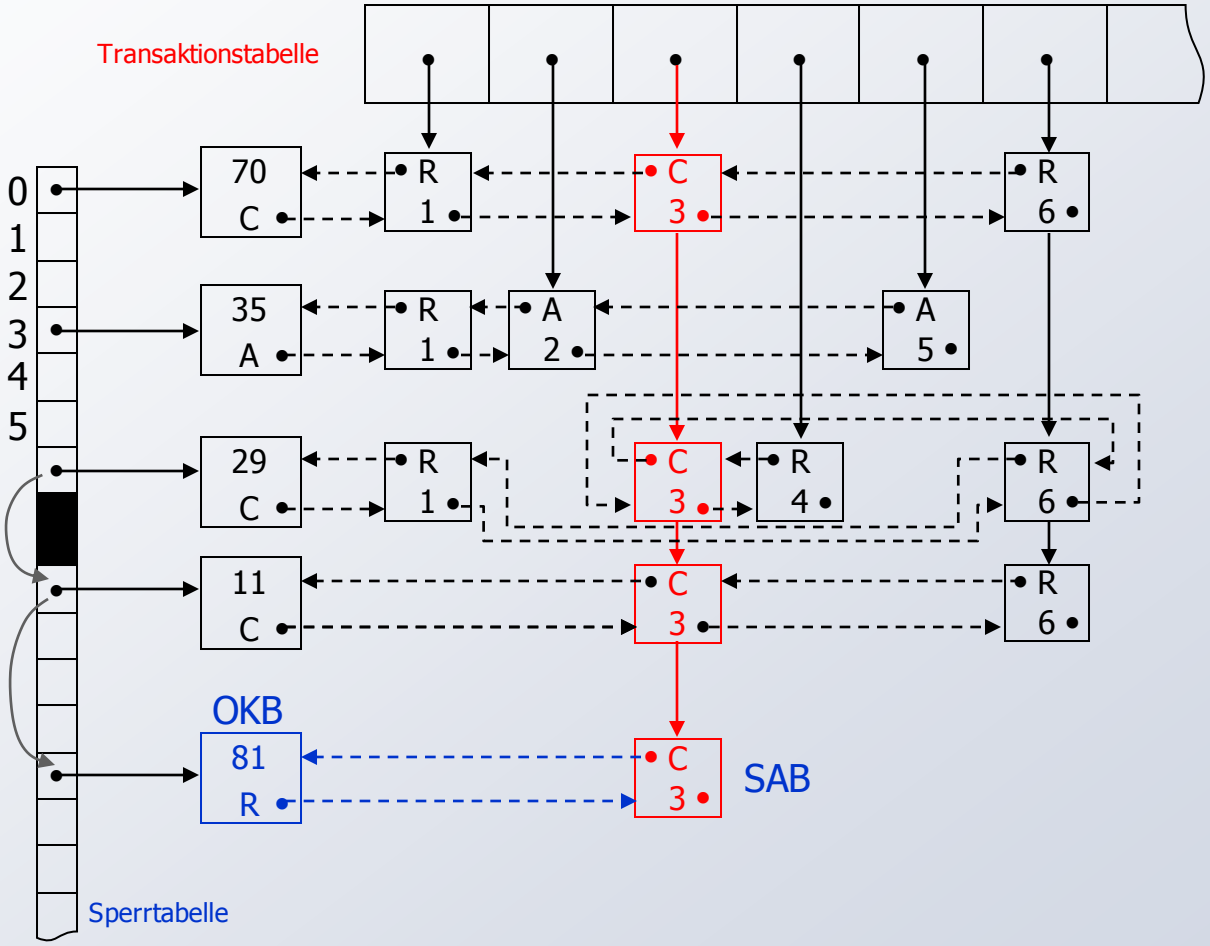


- **Probleme bei der Implementierung von Sperren**
  - Kleine Sperreinheiten (wünschenswert) erfordern hohen Aufwand
  - Sperranforderung und -freigabe sollten sehr schnell erfolgen, da sie sehr häufig benötigt werden ⇨ Sperrtabelle ist *High-Traffic*-Objekt!
  - Explizite, satzweise Sperren führen u. 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)

# Implementierungsaspekte - Datenstrukturen



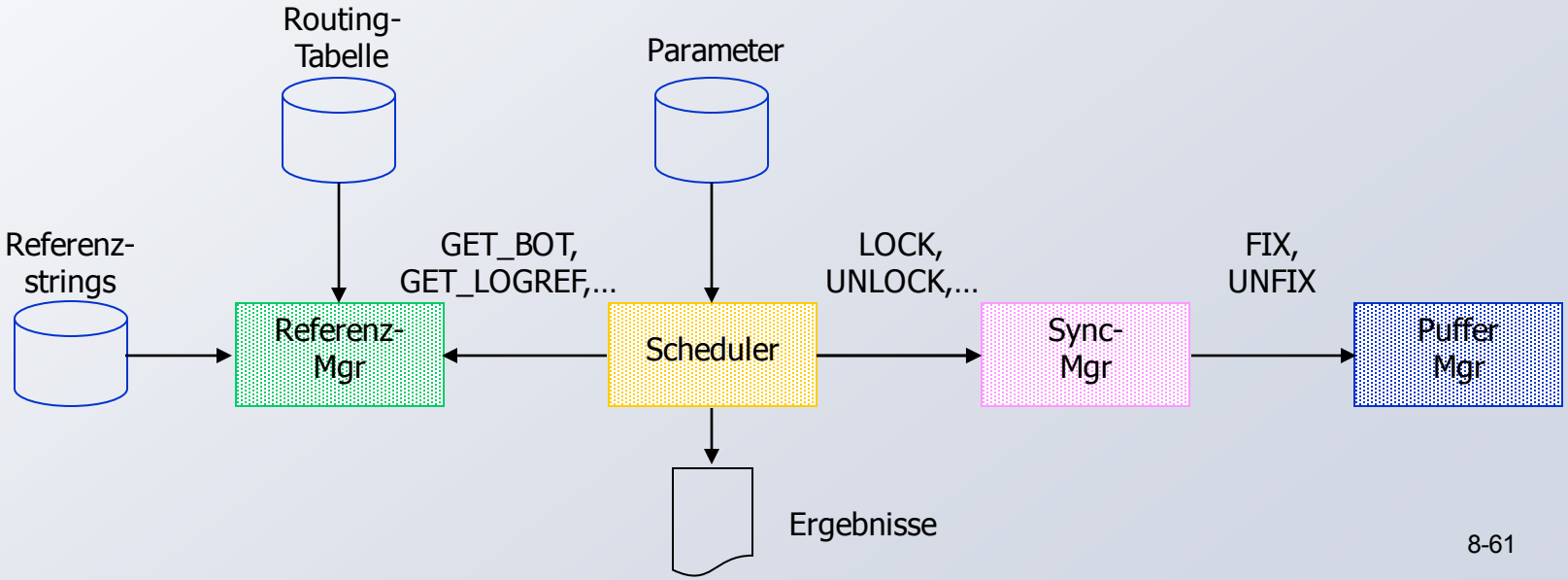
- **Beispiel: Sperrtabelle / TA-Tabelle für RAC-Verfahren**



# 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**



# Leistungsanalyse und Bewertung von Synchronisationsverfahren



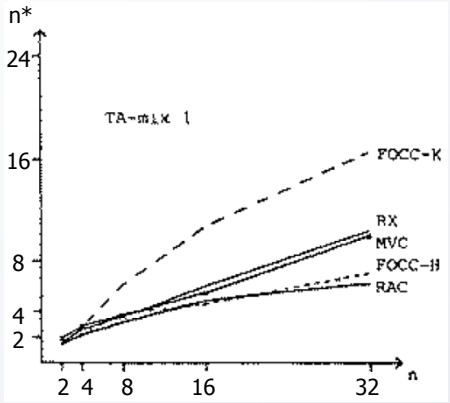
- **Wie bewertet man Parallelität?**
  - hoher Parallelitätsgrad – viele Rücksetzungen und Wiederholungen
  - moderate Parallelität, dafür geringerer Zusatzaufwand
- **Durchsatztest**
  - alle Transaktionen inkl. (mehrfache) Wiederholungen sind bearbeitet (Ermittlung der Durchlaufzeit)
  - einstellbarer 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

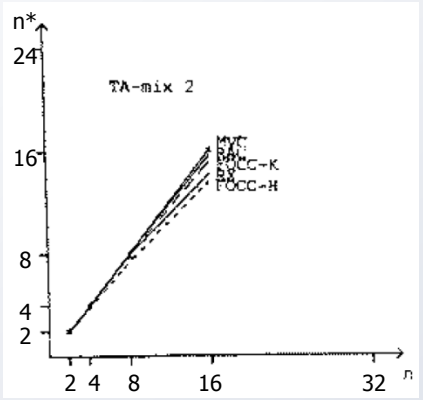
    $n^* = n'/q$
- **Zählung der Deadlocks**

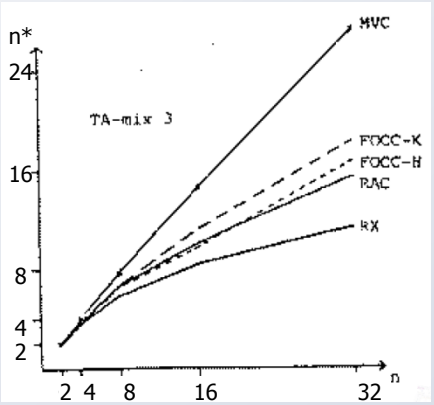
# Synchronisationsverfahren - Vergleich



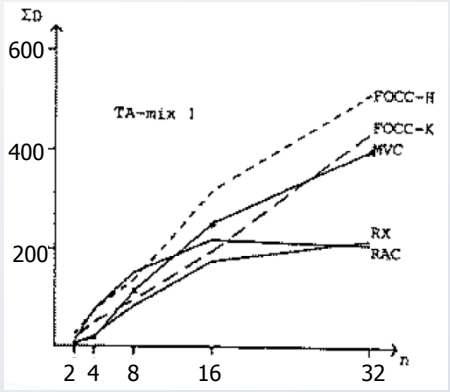
Effektive Parallelität

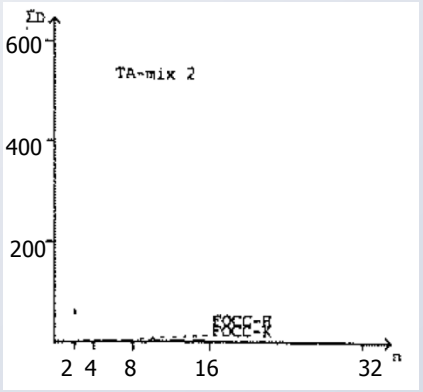


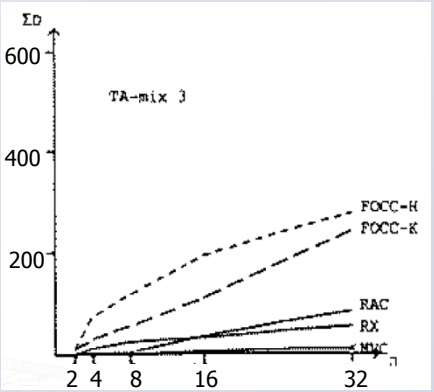




Summe der Deadlocks

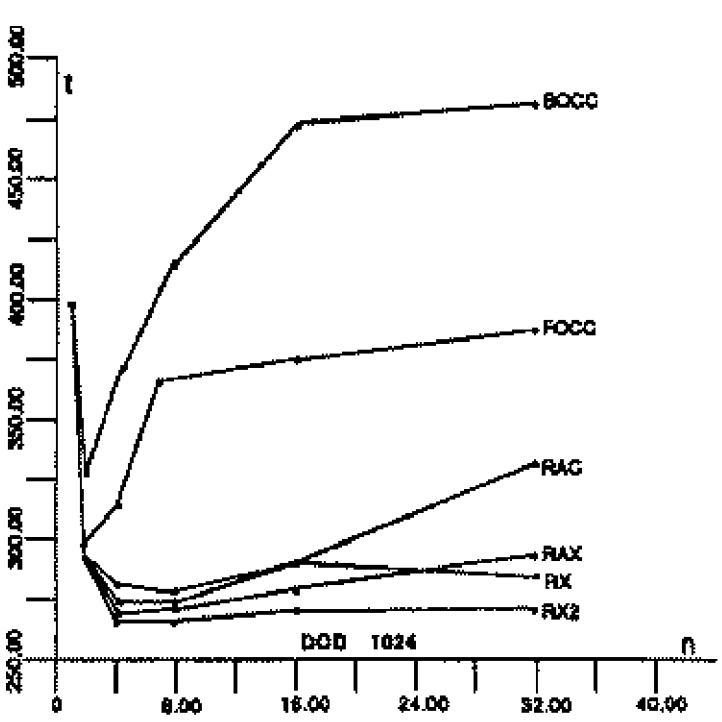


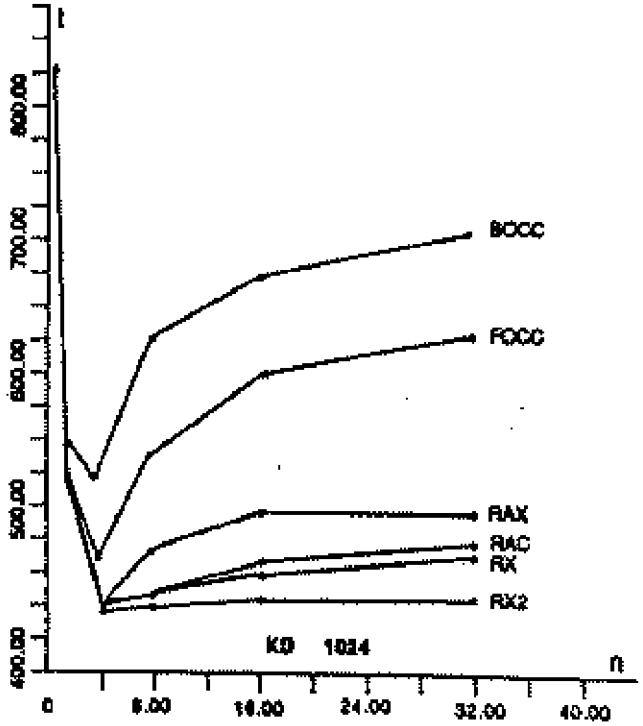



# Synchronisationsverfahren - Vergleich





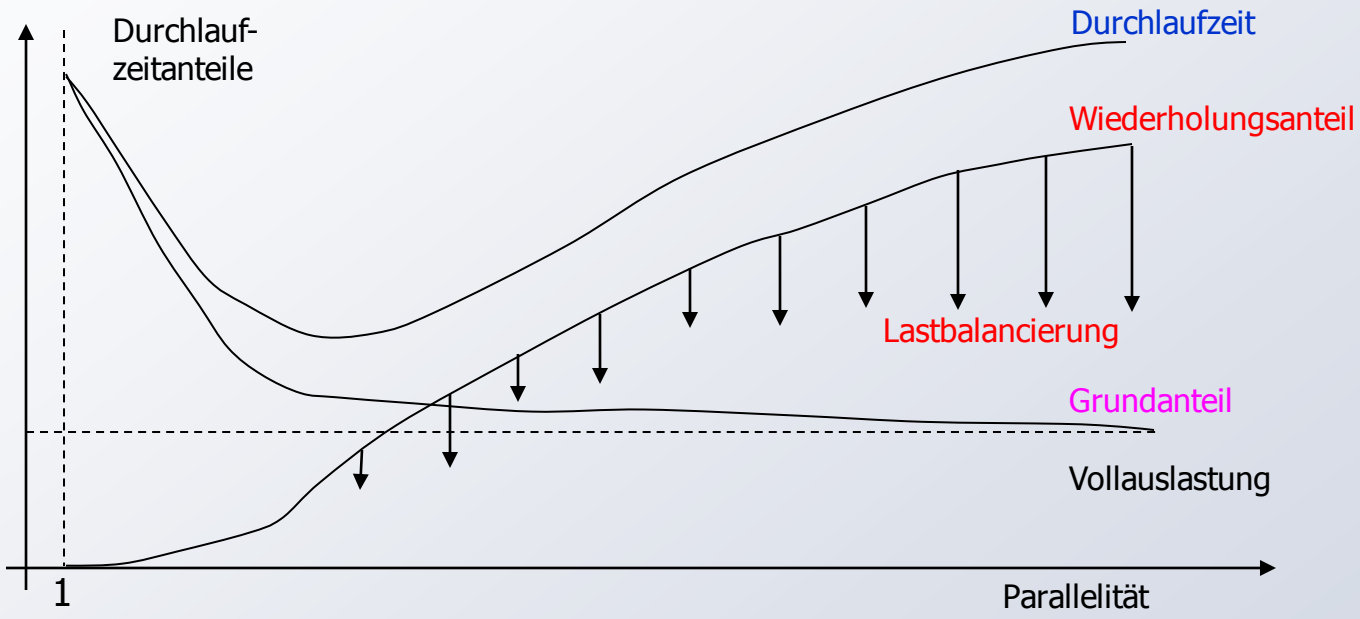




- **Schlussfolgerungen**
  - Sehr geringe Parallelität ⇨ keine effektive Nutzung der Ressourcen
  - Geringe Parallelität ⇨ bester Durchsatz, nicht notwendigerweise kürzeste Antwortzeiten
  - Pessimistische Methoden gewinnen: Blockierung vermeidet häufig Deadlocks
  - Optimistische Methoden geraten leicht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

# Synchronisation und Lastkontrolle



- **Charakteristische Wannenform (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 der durch den Referenzstring vorgegebenen Last bei fehlender Synchronisation nötig wäre
    - Wiederholungsanteil umfasst die Belegung des Prozessors zur nochmaligen Ausführung zurückgesetzter TAs

    ⇨ Rolle der Lastkontrolle und Lastbalancierung!

# Synchronisation und Lastkontrolle (2)



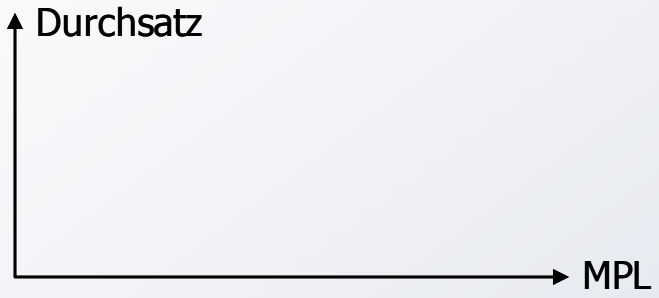
- **Empirische Bestätigung**
  - Theoretische Untergrenze des Grundanteils wird bei Vollauslastung des Prozessors erreicht
  - Häufigkeit von Leerphasen des Prozessors

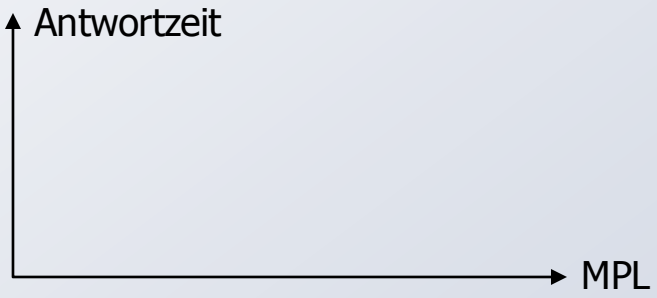
| DOD 1024 | | | | | | | |
|---|---|---|---|---|---|---|---|
| PAR | BOCCT | FOCC | BOCC | RX2 | RAX | RAC | RX |
| 1 | 4864 | 4864 | 4864 | 4864 | 4864 | 4864 | 4864 |
| 2 | 1623 | 1734 | 1584 | 1719 | 1725 | 1717 | 1788 |
| 4 | 59 | 105 | 50 | 149 | 206 | 94 | 256 |
| 8 | 108 | 37 | 17 | 17 | 26 |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

# Dynamische Lastkontrolle



- **Was nützt „blinde" Durchsatzmaximierung?**
- **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 Konfliktrate bei Sperrverfahren[6]:

  Konfliktrate = #gehaltener Sperren / #Sperren nicht-blockierter Transaktionen

  kritischer Wert: ca. 1,3 (experimentell bestimmt)
  - Zulassung neuer TAs nur, wenn kritischer Wert noch nicht erreicht ist
  - Bei Überschreiten erfolgt Abbrechen von TAs

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

# Performanz vs. Korrektheit

- **Trade-off**
  - Starke Isolation wie Serialisierbarkeit schützt die Datenintegrität
  - jedoch begrenzt den Durchsatz bei Konkurrenzsituationen (contention)
  - ➥ aber: Serialisierbarkeit garantiert immer korrekte TA-Ergebnisse!
- **Wunsch: Besserer Durchsatz auch bei „Contention"**
  - von Anfang an: Reduktion der Konsistenzebene (isolation level)
    - vor allem Read Committed (RC)
  - später: Mehrversionen-Verfahren (multiversion concurrency control algs.)
    - Snapshot Isolation (SI)
    - Multiversion Read Committed (RC_MV)*

➥ Reduktion der Konsistenzebene erlaubt dem Anwendungsentwickler Domänenwissen zu nutzen; bei sorgfältigem Einsatz bleiben dabei TA-Ergebnisse / Daten auch korrekt

- **Vergleich der Eigenschaften**
  - Eine TA T unter SI liest die Versionen der Daten, wie sie zu BOT(T) existierten.
  - RC_MV liest dagegen die jüngste freigegebene Version zum Zeitpunkt der Leseoperation (ähnlich wie das versionsfreie RC). Vergleiche mit SI!

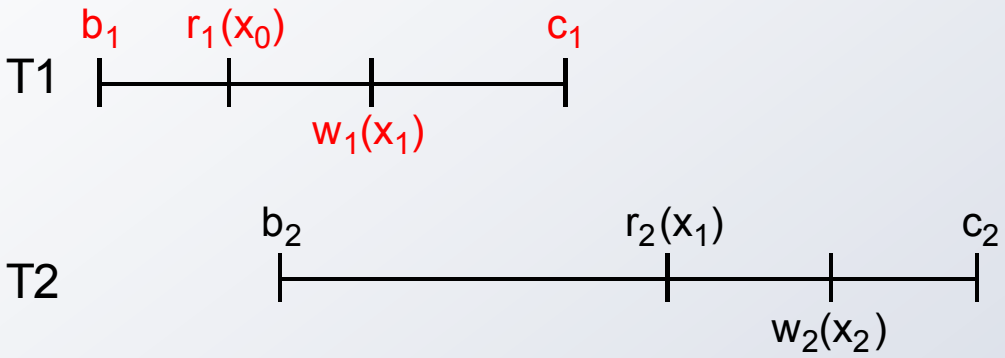
* Oracle-Dokumentationen verwenden auch „Oracle Read Consistency" für den RC_MV-Algorithmus

# Performanz vs. Korrektheit (2)



- **Beispiel eines RC_MV-Schedules:**

$b_1\ \ r_1(x_0)\ \ w_1(x_1)\ \ b_2\ \ c_1\ \ r_2(x_1)\ \ w_2(x_2)\ \ c_2$

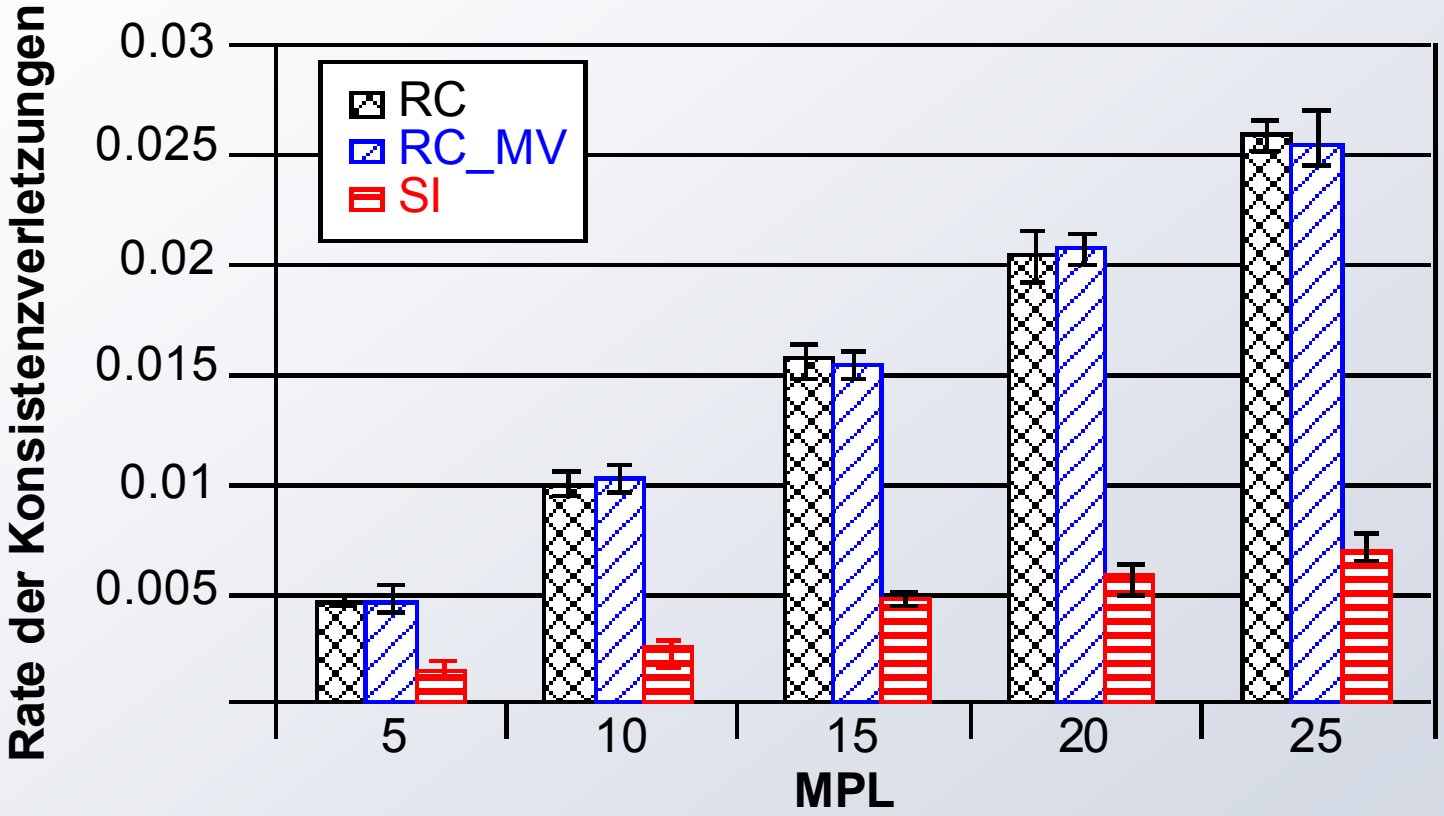



➥ Im Allgemeinen ist aber SI deutlich strikter als RC und RC_MV, bei denen „Phantome" und „inkonsistente Analysen" auftreten können

# Performanz vs. Korrektheit (3)



- **Beispiel: gemessene Rate der Konsistenzverletzungen vs MPL***




➥ SI erzeugt hier wesentlich weniger Inkonsistenzen als RC und RC_MV

* Fekete, A., Goldrei, S.N., Asenjo, J.P.: Quantifying Isolation Anomalies. Proc. VLDB 2009, Lyon, France.

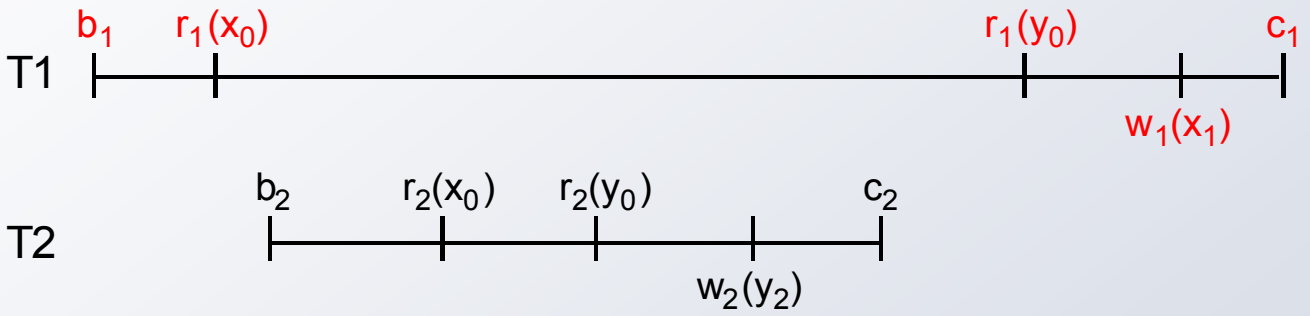
# Performanz vs. Korrektheit (4)



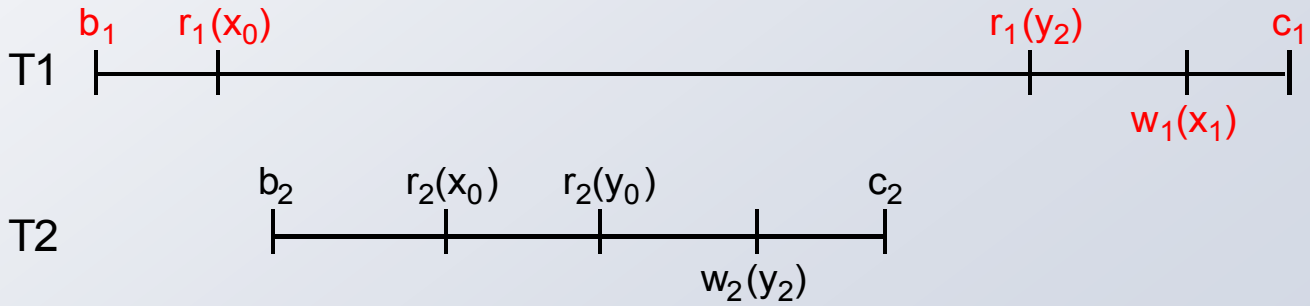
- **SI ist aber nicht durchgängig strikter als RC_MV und RC**

Beispiel: $b_1$ $r_1(x)$ $b_2$ $r_2(x)$ $r_2(y)$ $w_2(y)$ $c_2$ $r_1(y)$ $w_1(x)$ $c_1$

- Ablauf unter SI




- Ablauf unter RC und RC_MV



# Zusammenfassung

- **Realisierung der Synchronisation durch Sperrverfahren**
  - Prädikatssperren verkörpern eine elegante Idee, sind aber in praktischen Fällen nicht direkt einsetzbar, ggf. Nutzung in der Form von Präzisionssperren
  - 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)

  ⇨ Standard: multiple Sperrgranulate durch hierarchische Sperrverfahren
- **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
- **Generelle Optimierungen**
  - reduzierte Konsistenzebene und Mehrversionen-Ansatz
  - aber: Trade-off von Performanz und Korrektheit beachten
- **'Harte' Synchronisationsprobleme**
  - Hot Spots / High-Traffic-Objekte
  - lange (Änderung-) TAs
  - Wenn Vermeidungsstrategie nicht möglich ist, sind zumindest für Hochleistungs-systeme Spezialprotokolle anzuwenden
- **Dynamische Lastkontrolle erforderlich**