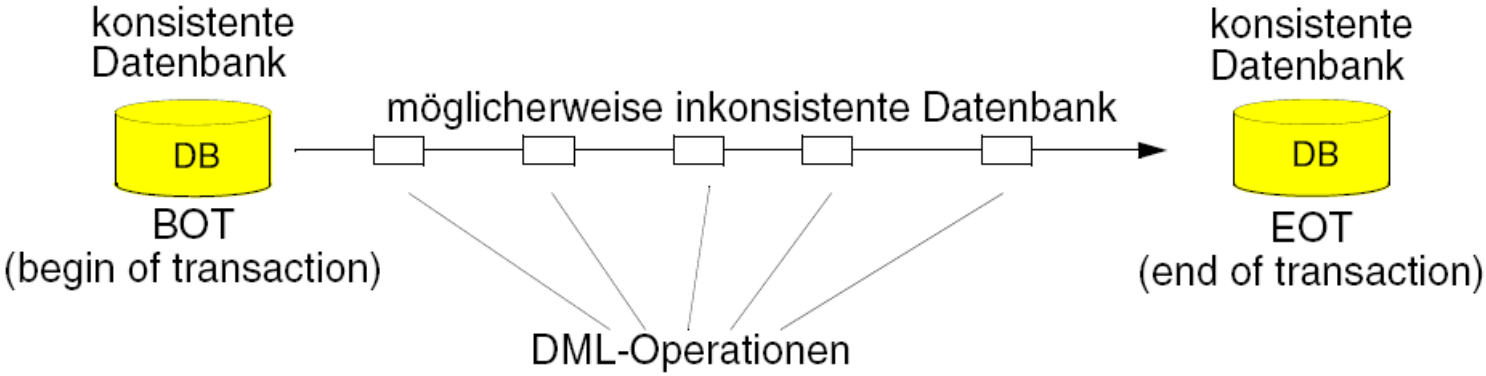
11 Concurrency Control

Transaktionen

- Zusammenfassung von aufeinanderfolgenden DB-Operationen, die eine Datenbank von einem konsistenten Zustand wieder in einen konsistenten Zustand überführt.



Merke

- das Ende einer Transaktion muss nicht notwendigerweise in einem anderen konsistenten Zustand enden

- Transaktionen werden immer beendet
  - normal (commit): Änderungen sind permanent in DB
  - abnormal (abort / rollback): bereits durchgeführte Änderungen werden zurückgenommen

Atomarität (atomicity)

- Unteilbarkeit durch Transaktionsdefinition (Begin – End)
- Alles-oder-Nichts Prinzip, d.h. das DBS garantiert
  - entweder die vollständige Ausführung einer Transaktion …
  - oder die Wirkungslosigkeit der gesamten Transaktion (und damit aller beteiligten Operationen)

Konsistenzerhaltung (consistency)

- Eine erfolgreiche Transaktion garantiert, dass alle Konsistenzbedingungen (Integritätsbedingungen) eingehalten worden sind

Isolation (isolation)

- Mehrere Transaktionen laufen voneinander isoliert ab und benutzen keine (inkonsistenten) Zwischenergebnisse anderer Transaktionen

Dauerhaftigkeit (durability)

- Alle Ergebnisse erfolgreicher Transaktionen müssen persistent gemacht werden (worden sein)

Beispiel

- vorgegebene Konsistenzbedingung: x=y

```
BOT              // DB ist konsistent
read(x)
read(y)
x := x+1
write(x)         // DB ist zeitweilig inkonsistent
y := y+1
write(y)
EOT              // DB ist wieder konsistent
```

Arten von Konsistenz

- Datenbankkonsistenz
  - alle (auf der DB definierten) Konsistenzbedingungen sind erfüllt
- Transaktionskonsistenz
  - der nebenläufige Ablauf der Transaktionen ist korrekt
  - Gefahr der Anomalien

Unterschiedliche Sichtweisen

	Korrektheit der Abbildungshierarchie	Übereinstimmung zwischen DB und Miniwelt
durch das Anwendungsprogramm	Mehrbenutzer-Anomalien	unzulässige Änderungen
	Synchronisation	Integritätsüberwachung des DBS TA-orientierte Verarbeitung
durch das DBS und die Betriebsumgebung	Fehler auf den Externspeichern, Inkonsistenzen in den Zugriffspfaden	Undefinierter DB-Zustand nach einem Systemausfall
	Fehlertolerante Implementierung	Transaktionsorientierte Fehlerbehandlung (Recovery)

Synchronisation

Ziel

- Erhaltung der Transaktionskonsistenz (= operationelle Integrität) im Mehrbenutzerbetrieb

Gründe für den Mehrbenutzerbetrieb

- Verteilung generell (z.B. EC-Automat)
- CPU-Nutzung während systemgetriebenen und benutzergetriebenen TA-Unterbrechungen
- Kommunikationsvorgänge in verteilten Systemen

Gegenstand der Synchronisation

- Vermeidung der gegenseitigen Beeinflussung von Lese- und Schreiboperationen
- Verhinderung von Anomalien im Mehrbenutzerbetrieb

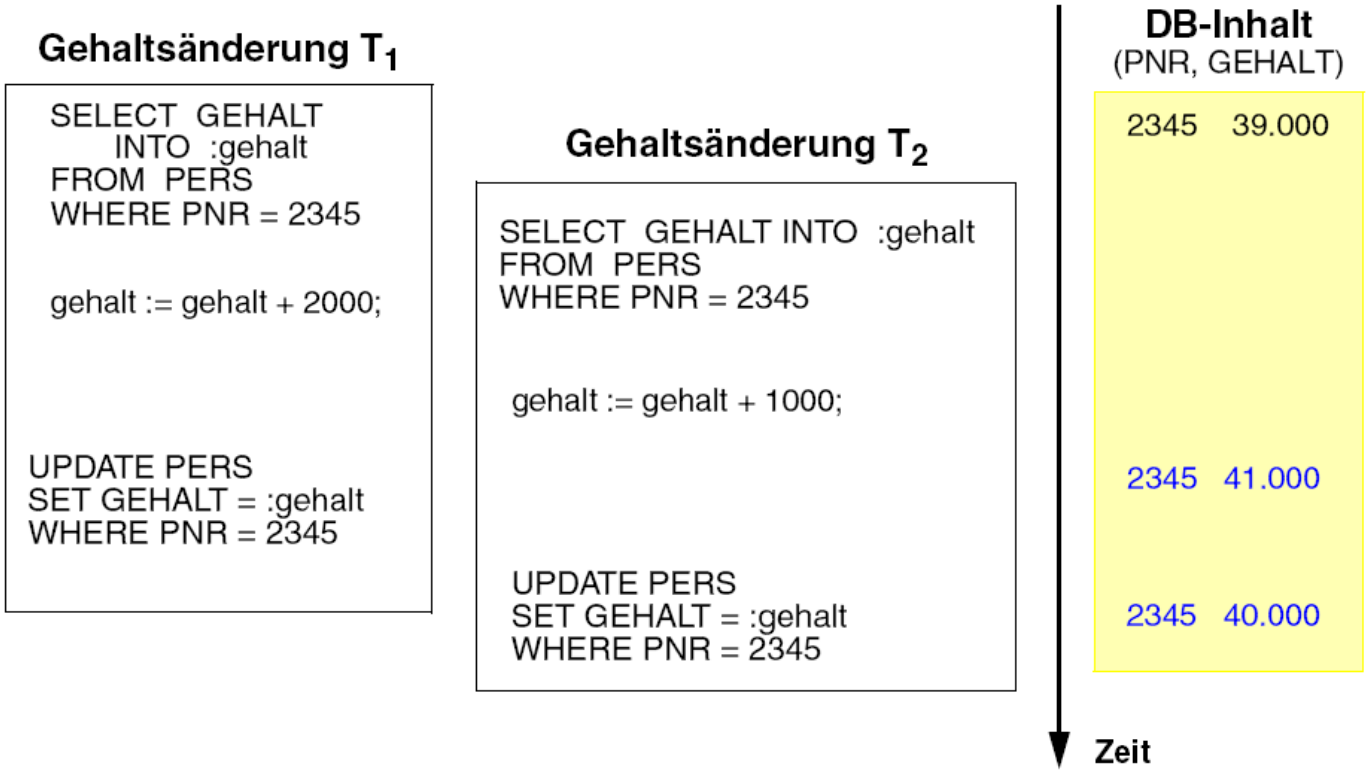
Mögliche Anomalien ohne Synchronisation

- Verlorengegangene Änderungen (lost update)
- Abhängigkeiten von nicht freigegebenen Änderungen (dirty read, dirty overwrite)
- Inkonsistente Analyse (non-repeatable read)
- Phantom-Problem

Lösung: Serialisierbarkeit

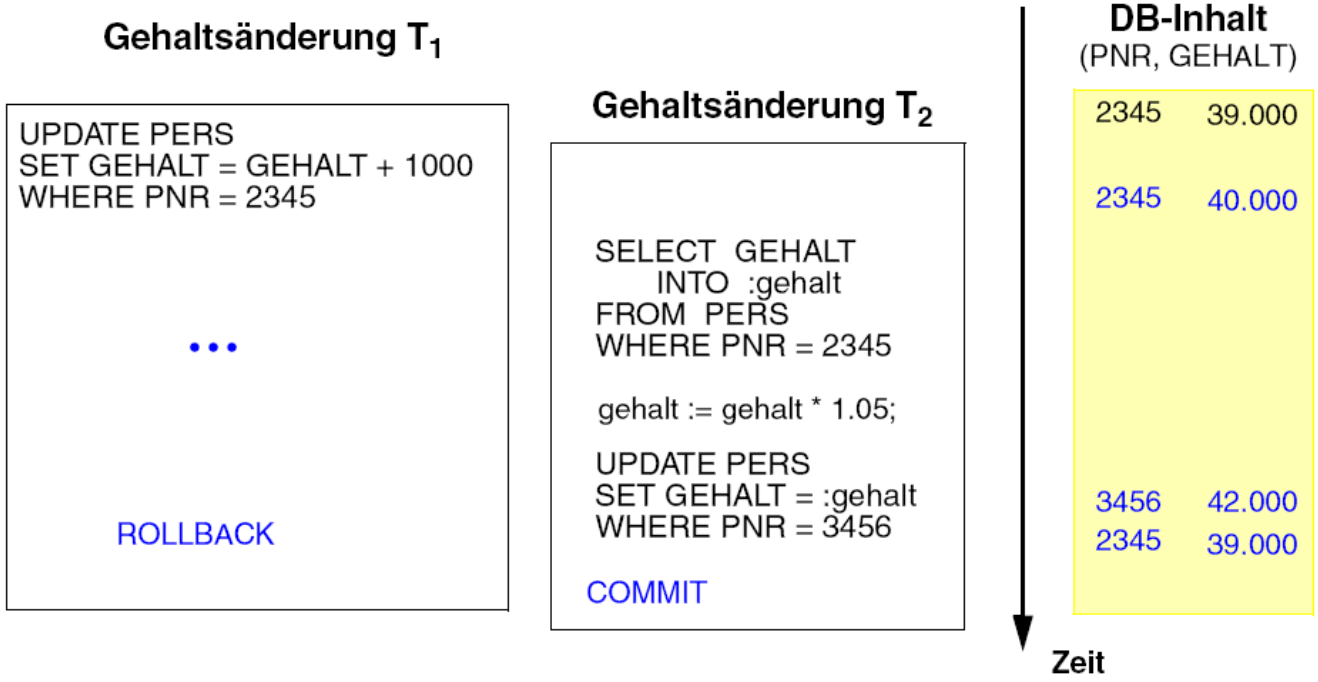
- aber: bei langen TA zu große Wartezeiten für andere TA (Scheduling-Fairness)
- Sicherstellen der operationellen Integrität durch eine „virtuelle“ serielle Ausführung der einzelnen Operationen einer Transaktion (logischer Einbenutzerbetrieb!)

Beispiel für „Lost Updates“



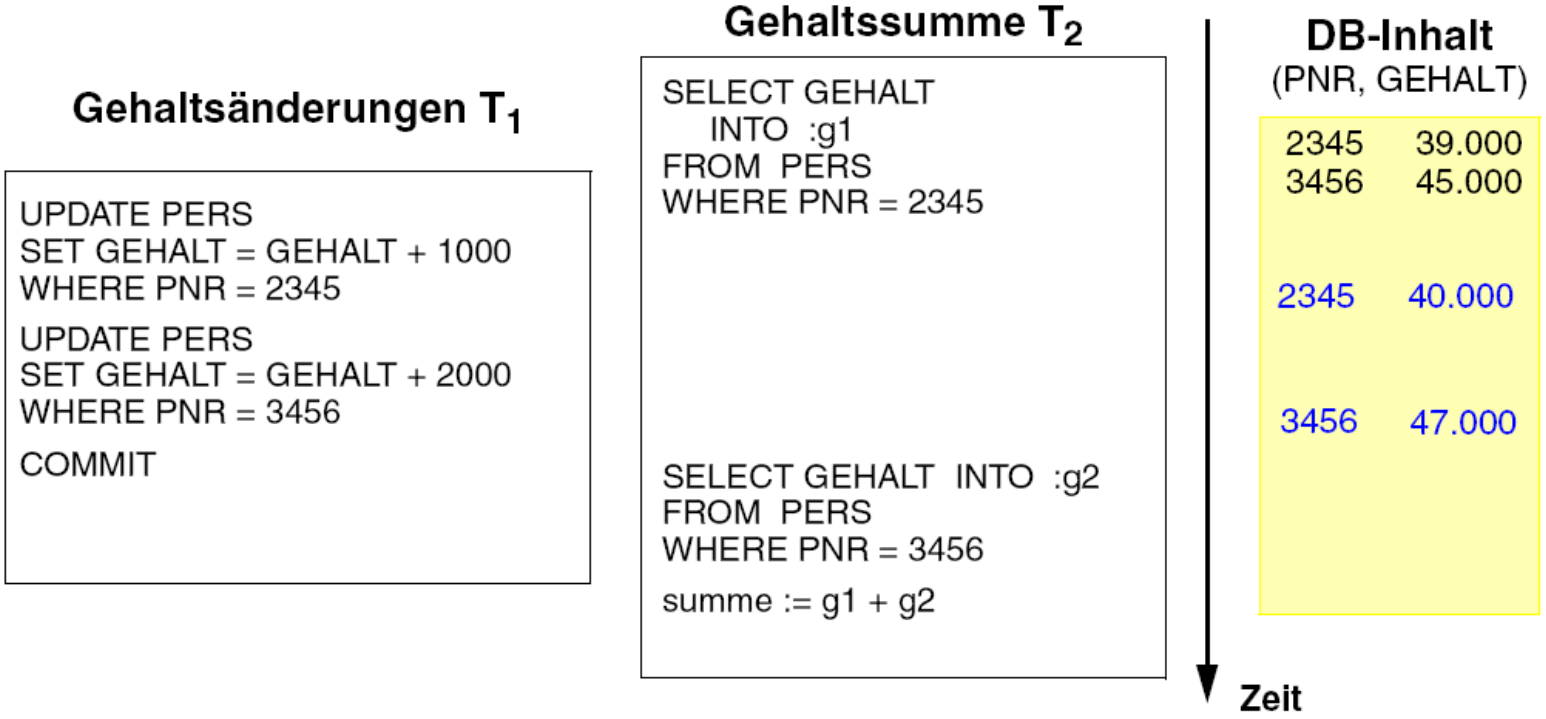
- Konkurrierendes Verändern eines Datenelementes
- write/write – dependency
- Lösung: exklusives Sperren für den Schreiber

Beispiel für „Dirty Read“



- Abhängigkeiten von nicht freigegebenen Änderungen (rekursives Zurücksetzen)
- write/read-dependency
- Lösung
  - Ermöglichen eines isolierten Zurücksetzens
  - Lesen geänderter Daten erst wenn sie freigegeben sind!

Beispiel für „non-repeatable read“



- Während der Verarbeitung von T_2 wird der Datenbestand durch T_1 verändert
- T_2 sieht konsistente Zustände, jedoch unterschiedliche!
- read/write-dependency

> Anomalien (4) – Phantom Problem

Lesetransaktion (Gehaltssumme prüfen)	Änderungstransaktion (Einfügen eines neuen Angestellten)
SELECT SUM(Gehalt) INTO :summe FROM pers WHERE anr=17	
	INSERT INTO Pers(pnr, anr, gehalt) VALUES (4567, 17, 55.000) UPDATE Abt SET gehaltssumme = gehaltssumme+55.000 WHERE anr=17
SELECT Gehaltssumme INTO :gsumme FROM abt WHERE anr=17 IF gsumme <> summe THEN <Fehlerbehandlung>	

- Interpretation: „dirty read“ auf höherer Ebene (relationenübergreifend!)

Synchronisation durch Sperren

Transaktion T_j setzt sich aus folgenden Operationen zusammen

- Leseoperation: $r_j(A)$
- Schreiboperation: $w_j(A)$
- Abbruch: a_j
  (es gibt keine andere Operation der TA, die danach ausgeführt wird)
- Commit: c_j
  (es gibt keine andere Operation der TA, die danach ausgeführt wird)
- Weitere Operationen, die aber auf die Datenbank (und für die parallele) keine Auswirkung haben.

Bemerkungen

- einzelne Operationen r_j, w_j, a_j und c_j werden sequentiell nacheinander ausgeführt
- für jede TA T_j gibt es eine zweistellige Relation $<_j$, die die Ordnung der Operationen ausdrückt:

  $op_1 <_j op_2$: op_1 wird vor op_2 ausgeführt.

Realisierung eines logischen Einbenutzerbetriebs

- Einführung von Sperren für exklusiven Zugriff auf Datenobjekte
- für jedes benutzte Datenobjekt wird zentral in einer Sperrtabelle die Nutzungsart (Sperrmodus) protokolliert

Arten von Sperren

- X (eXclusive)-Sperre (=Schreibsperre)
- S/R (shared/read)-Sperre (=Lesesperre)

Kompatibilitätsmatrix

- gibt Auskunft, ob eine Sperranforderung für ein (möglicherweise bereits gesperrtes) Objekt gewährt werden kann

angeforderter Sperrmodus	NL	R	X
R	+	+	-
X	+	-	-

aktueller Sperrmodus

(NL (no lock) wird meist weggelassen)

Wann sind Sperren zu erwerben?

- Statisches Sperren
  - zu Beginn der Transaktion alle Sperren anfordern (“preclaiming”)
  - Nachteil: Man muss alles sperren, was man brauchen könnte.
- Dynamisches Sperren
  - während der Transaktion werden Sperren nach Bedarf angefordert
  - Nachteil: Verklemmungen (deadlocks)

Wann sind Sperren freizugeben?

- Sperren müssen bis zum Ende der Transaktion gehalten werden, um Serialisierbarkeit zu garantieren
- Abschwächung zur Optimierung
  - frühzeitiges Freigeben z.B. von Lesesperren
  - Nachteil: wiederholtes Lesen desselben Satzes kann dann unterschiedliche Ergebnisse liefern

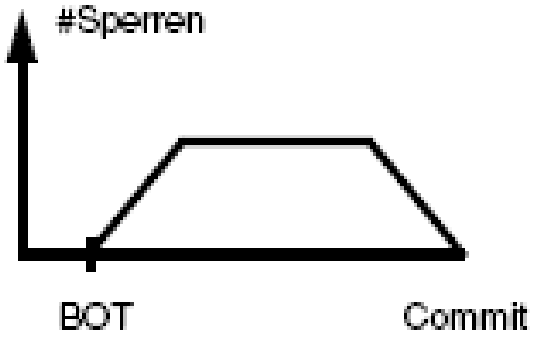
    4 standardisierte Konsistenzstufen in SQL (consistency levels)

Regeln für den Umgang mit Sperren

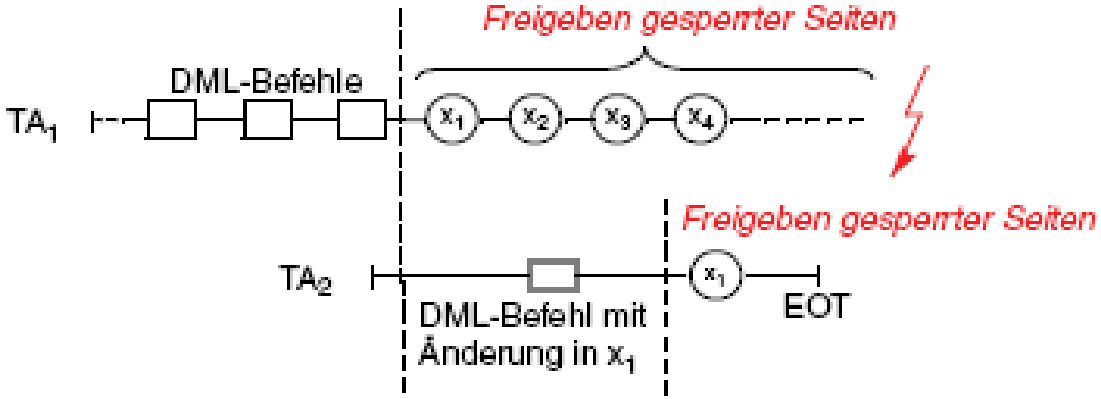
- Jedes Datenobjekt, auf das zugegriffen werden soll, muss vorher gesperrt werden
- Eine Transaktion fordert eine Sperre, die sie bereits besitzt, nicht noch einmal an
- Eine Transaktion muss die von anderen Transaktionen gesetzten Sperren beachten
- Am Ende einer Transaktion sind alle Sperren wieder freizugeben
  (Eswaran, Gray, Lorie und Traiger 1976)

Zweiphasigkeit

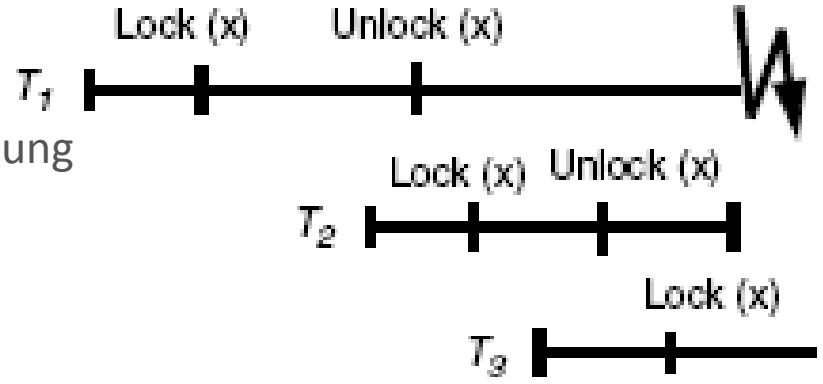
- Anfordern von Sperren erfolgt in einer **Wachstumsphase**
- Freigabe der Sperren in **Schrumpfungsphase**
- Sperrfreigabe kann erst beginnen, wenn alle Sperren gehalten werden



Mikroskopische Sichtweise auf das Freigeben von Sperren

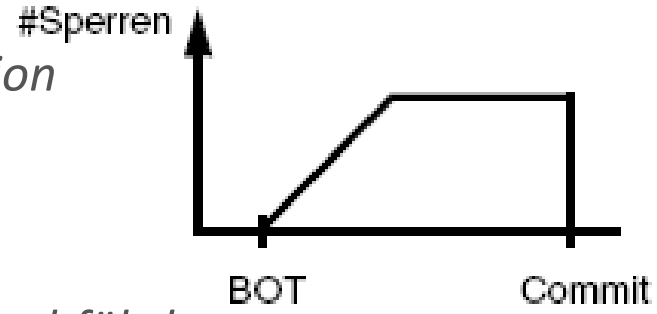


Verschärfung des Zweiphasen-Sperrprotokolls



Merke

- bei fehlerfreiem Betrieb ist diese Voraussetzung (fast) ausreichend, um einen korrekten Transaktionsablauf zu gewährleisten
- Problem des kaskadierenden Abbruchs einer Transaktion
  - Rücksetzen von TAs, die bereits freigegebene Daten gelesen haben
  - Rücksetzen von TAs, die bereits selbst schon committed sind !



Freigabe aller Sperren erst am Ende einer Transaktion

- Striktes Zweiphasen-Sperrprotokoll
- Profil der Sperranforderungen und Sperrfreigaben

Problem: Freigabe aller Sperren ist nicht atomar durchführbar

- Einführung einer zweiphasigen Commit-Behandlung
- Phase 1: Sichern der isolierten Wiederholbarkeit und Schreiben des EOT-Satzes
- Phase 2: Freigabe aller Sperren und Beenden der Aktivität

Beispiel eines elementaren Deadlocks

Transaktion T1	Transaktion T2
T1 hält X-Sperre auf A	T2 hält X-Sperre auf B
T1 benötigt B zum beenden	T2 benötigt A zum beenden

Voraussetzungen für Deadlock

- paralleler Zugriff
- exklusive Zugriffsanforderungen (X-Sperren)
- anfordernde TA besitzt bereits Sperren auf Datenobjekte
- keine vorzeitige Freigabe von Sperren auf Datenobjekte (non-preemption)
- es existieren zyklische Wartebeziehungen zwischen zwei oder mehr Transaktionen

Timeout-Verfahren

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

Deadlock-Verhütung (Prevention)

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

Deadlock-Vermeidung (Avoidance)

- potentielle Deadlocks werden im voraus erkannt und durch entsprechende Maßnahmen vermieden
  → Laufzeitunterstützung nötig

Deadlock-Erkennung (Detection)

- 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)

Konsistenzebenen

Isolation Levels in SQL

SET TRANSACTION ISOLATION LEVEL X

where X =

3 = SERIALIZABLE

2 = REPEATABLE READ

1 = READ COMMITTED

0 = READ UNCOMMITTED

Kommerzielle Datenbanksysteme implementieren viele weitere Varianten von Konsistenzebenen, insbesondere:

CURSOR STABILITY

- Erweiterung von READ COMMITTED
- Lange Schreibsperren basierend auf Prädikaten
- Kurze Lesesperren auf einzelne Tupel
- (Manchmal auch: 2.99 = REPEATABLE READ, 2 = CURSOR STABILITY)

Transaktion T1

```
(max) SELECT MAX(price)
      FROM  Products
      WHERE name = 'X301';
(min) SELECT MIN(price)
      FROM Products
      WHERE name = 'X301';
```

Transaktion T2

```
(del) DELETE
      FROM  Products
      WHERE name = 'X301';
(ins) INSERT INTO Products
      VALUES('X301',
             'Cyberport', 1350);
```

Mögliche Verzahnung der Operationen der einzelnen Transaktionen

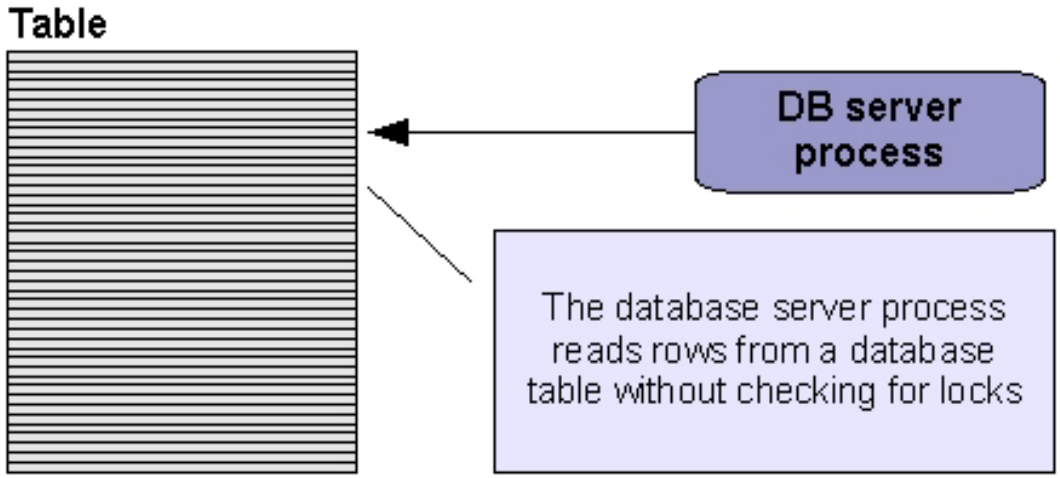
(max)(del)(ins)(min)

X301 Preise:	{1250, 1300}	{1250, 1300}		{1300}
Operationen:	(max)	(del)	(ins)	(min)
Ergebnis:	1300			1350

→ *T1 sieht MAX < MIN !!!*

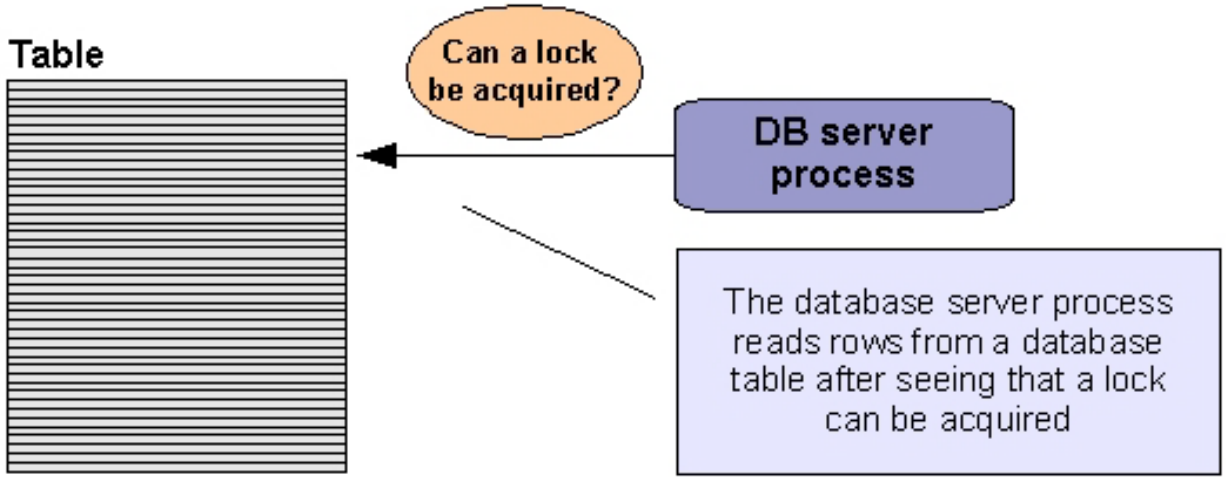
READ UNCOMMITTED

- Eine Transaktion, die unter READ UNCOMMITTED läuft, berücksichtigt keine Sperren anderer Transaktionen.
- Die Transaktion kann Daten anderer Transaktionen sehen, auch wenn diese noch nicht beendet (erfolgreich oder erfolglos) sind → Phantome
- Für Datenbanken ohne Logging ist dies normalerweise der einzig verfügbare Isolationslevel
- Anwendungsfall:
  - Der Inhalt ist statisch (keine Updates, read-only Tabellen)
  - 100%ige Korrektheit ist nicht so wichtig wie Geschwindigkeit und Garantie der Verklemmungsfreiheit
- Synonym: DIRTY READ



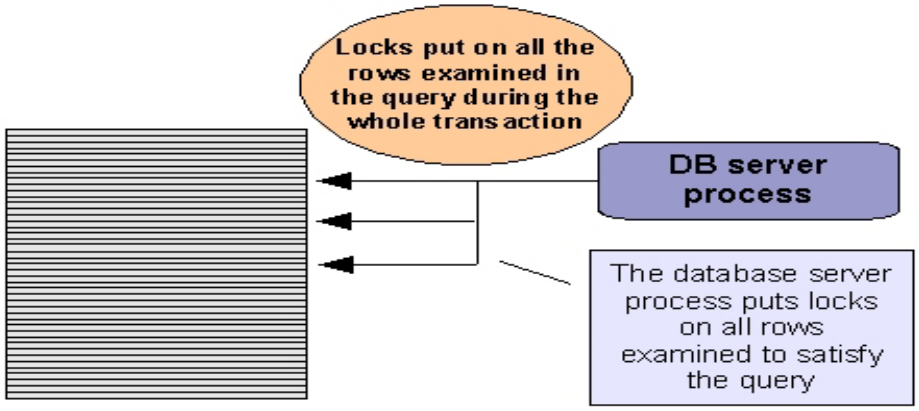
READ COMMITTED

- Falls Transaktion T1 in READ COMMITTED läuft, kann Sie nur freigegebene Datenbestände sehen, aber nicht notwendigerweise die gleichen Zustände
- Möglicher Schedule: (max)(del)(ins)(min)
  - → T1 würde Max < Min sehen!
- Kein Zugriff auf Phantome oder nicht freigegebene Daten
- Zugriff nur auf freigegebene Daten, die jedoch nach dem Lesen sofort von einer anderen Transaktion geändert werden können.
- nützlich für Lookup-Queries
- Synonym: COMMTTED READ



REPEATABLE READ

- Anforderungen sind analog zu READ COMMTTED; zusätzlich gilt, dass Alles was beim ersten Mal gelesen wird, mindestens auch bei allen weiteren Lesevorgängen gesehen wird.
- Es können jedoch mehr Tupel bei den weiteren Lesevorgängen gesehen werden!
- Möglicher Schedule: (max)(del)(ins)(min)
  - (min) würde den Zustand vor (del) und auch das neue Tupel nach (ins) sehen
- Nützlich falls alle zu lesenden Tupel als logische Einheit gesehen werden müssen bzw. wenn garantiert sein muss, dass sich ein Wert nicht ändert (z.B. Aggregatberechnung in Accounting)
- Sinnvoll, falls koordinierte Lookups über mehrere Tabellen gefahren werden müssen



CURSOR STABILITY

- Zugriff auf Daten über einen Cursor, d.h. OPEN gefolgt von mehreren FETCH-Operationen
- Jede Operation (FETCH) ist für sich atomar
- Tupel auf die ein Cursor aktuell zeigt, können von anderen Transaktionen nicht manipuliert werden
- Alle Tupel, die von einer Transaktion zugegriffen werden, aber auf die aktuell der Cursor nicht zeigt, können von anderen Transaktionen aktualisiert werden

Beispiel

T_1: fetch(t) update(t)
T_2: update(t) commit

- möglich bei READ COMMITTED (und damit Lost Updates möglich)
- nicht erlaubt bei CURSOR STABILITY, da T1 auf Tupel t über einen Cursor zugreift und Tupel t für die Dauer der Cursor-Positionierung gesperrt bleibt.

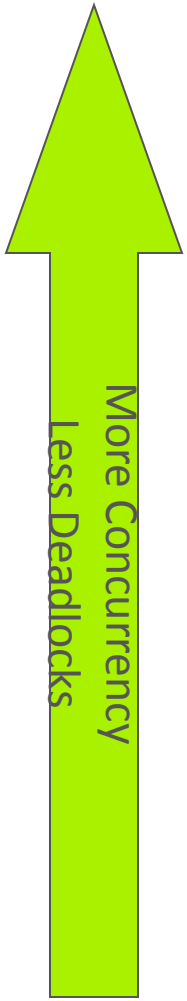
SERIALIZABLE

- Falls Transaktion T1 in SERIALIZABLE abläuft, sieht die Transaktion den Zustand entweder vor oder nach der Transaktion T2, aber keinen Zwischenzustand

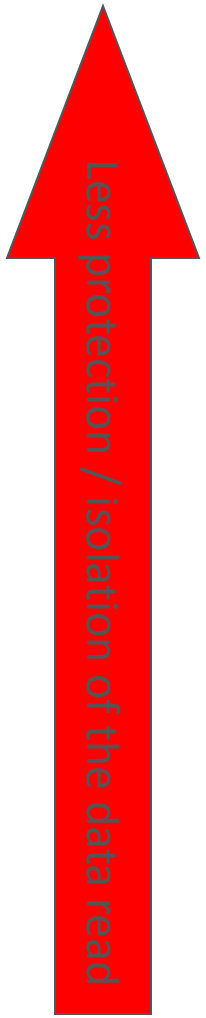
Anmerkung

- Die Wahl der Konsistenzebenen berührt nur die jeweilige Transaktion, nicht wie andere Transaktionen die Datenbankzustände sehen.
- Beispiel
  - T2 läuft serializable, T1 aber in einem schwächeren Modus
  - T1 könnte keine Preise für „X301“ ermitteln, d.h. für Sie würde es so aussehen, als würde Sie mitten innerhalb der Transaktion T2 laufen

Informix SQL	ANSI SQL	Remarks
Dirty read ***set isolation to dirty read [retain update locks]***	Read uncommitted ***set transaction isolation level read uncommitted***	**No locks are placed during reading data and no locks from other sessions will block this reader**
Last Committed read ***set isolation to committed read last committed [retain update locks]***	Not available	**Returns the most recently committed version of the rows, even if another concurrent session holds an exclusive lock**
Committed read ***set isolation to committed read [retain update locks]***	Read committed ***set transaction isolation level read committed***	**Checks for locks being held by other sessions, but does not place a lock itself**
Cursor stability ***set isolation to cursor stability [retain update locks]***	Not available	**An update lock is placed on the current fetched row. It will be promoted to an exclusive lock as soon an update is executed**
Repeatable read ***set isolation to repeatable read***	Serializable (and Repeatable read) ***set transaction isolation level serializable***	**A share lock is placed on every row read to make sure that this query returns the same result set if it is being re-executed in this transaction**



Isolation level	Dirty read Can occur?	Nonrepeatable read Can occur?	Phantom read Can occur?
Dirty read **(ANSI: Read uncommitted)**	**Yes**	**Yes**	**Yes**
Last Committed Read **(ANSI: Not supported)**	**No**	**Yes**	**Yes**
Committed read **(ANSI: Read Committed)**	**No**	**Yes**	**Yes**
Cursor Stability **(ANSI: Not supported)**	**No**	**No**	**Yes**
Repeatable Read **(ANSI: Serializable)**	**No**	**No**	**No**



Erweiterte Sperrverfahren

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
- explizite, satzweise Sperren führen u. U. zu umfangreichen Sperrtabellen und großem Zusatzaufwand
- Zweiphasigkeit der Sperren führt häufig zu langen Wartezeiten (starke Serialisierung)
- häufig berührte Zugriffspfade können zu Engpässen werden
- Eigenschaften des Schemas können „hot spots“ erzeugen

Mögliche Optimierungen

- Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
- Nutzung mehrerer Objektversionen
- spezialisierte Sperren (Nutzung der Semantik von Änderungsoperationen)

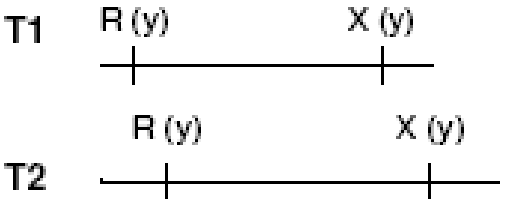
Vielzahl weiterer Sperrverfahren

- U-Sperre für Lesen mit Änderungsabsicht
  - Ziel: Verhinderung von Konversions-Deadlocks
  - bei Änderung Konversion U → X, andernfalls U → R (downgrading)

- Anwartschaftssperren: I-Sperre oder Sperranzeige
  - IS-Sperre (intention share),
    falls auf untergeordnete Objekte nur lesend zugegriffen wird
  - IX-Sperre (intention eXclusive),
    falls auf untergeordnete Objekte schreibend zugegriffen wird
  - entspricht einer Sperre für Betriebsmittel auf einer höheren Hierarchiestufe
  - die Nutzung einer Untermenge wird angezeigt, in der Untermenge werden noch explizite Sperren gesetzt

- Kombinierte Sperr-Sperranzeige: SIX = S + IX (share and intention exclusive)
  - sperrt das Objekt in S-Modus
  - verlangt X-Sperren auf tieferer Hierarchieebene nur für zu ändernde Objekte

Deadlock-Gefahr durch Sperrkonversionen



Erweitertes Sperrverfahren

- Ziel: Verhinderung von Konversions-Deadlocks
- U-Sperre für Lesen mit Änderungsabsicht
- bei Änderung Konversion U → X, andernfalls U → R (downgrading)

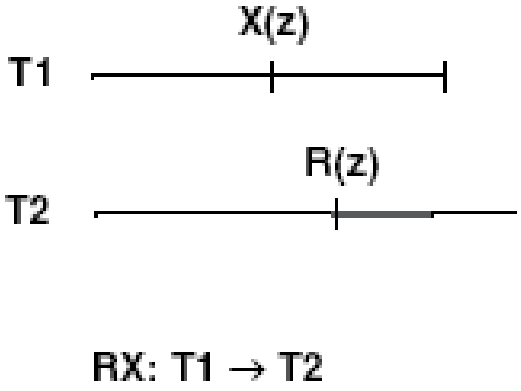
angeforderter Sperrmodus \ aktueller Sperrmodus	R	U	X
R	+	-	-
U	+	-	-
X	-	-	-

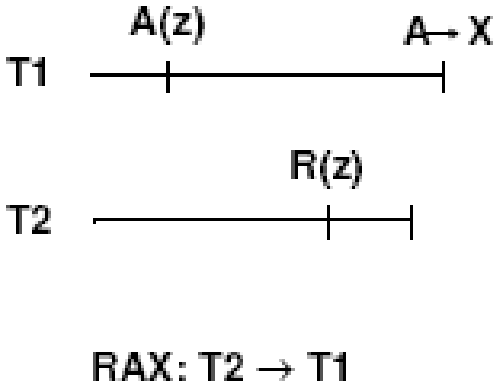
- u. a. in IBM DB2 eingesetzt
- das Verfahren ist unsymmetrisch!

	R	A	X
R	+	⊕	-
A	⊕	-	-
X	-	-	-

Prinzip

- Änderungen erfolgen in temporärer Objektkopie, paralleles Lesen der gültigen Version wird zugelassen
- Schreiben wird nach wie vor sequentialisiert (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.a. andere Serialisierungsreihenfolge:

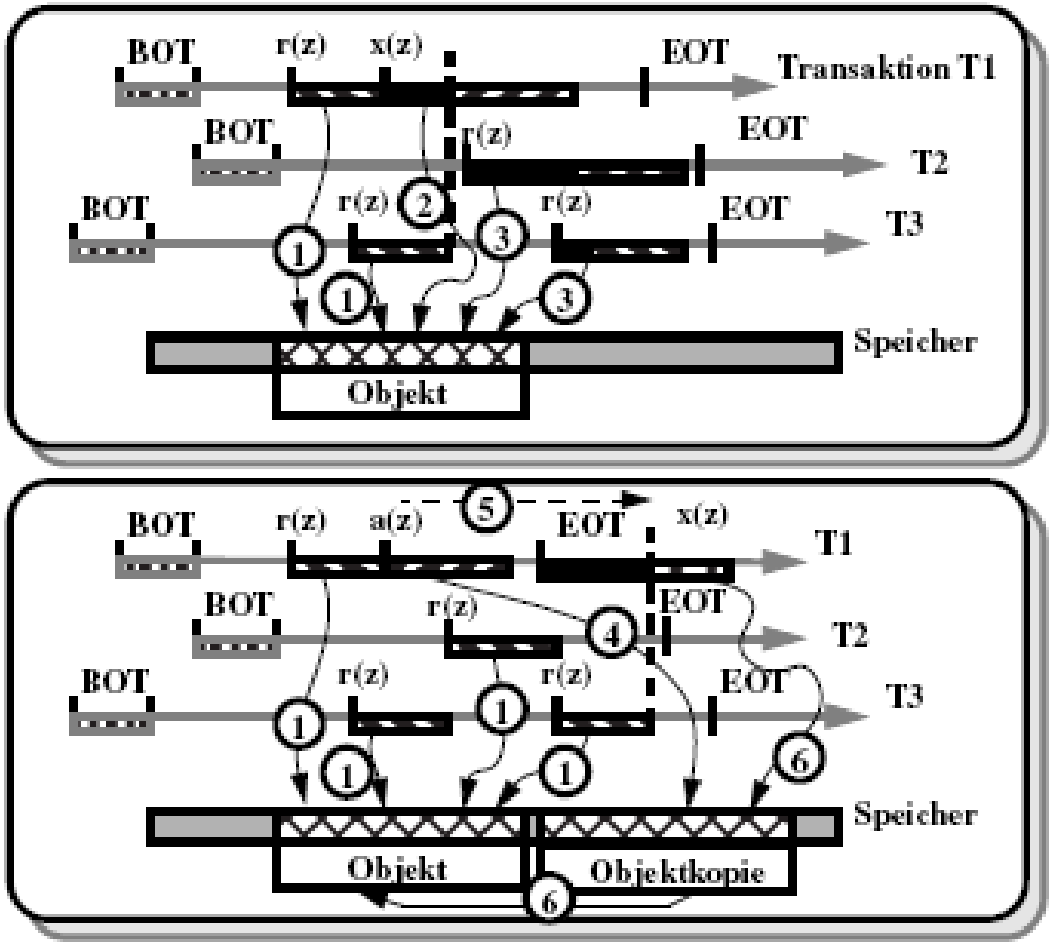




- starke Behinderungen von Update-TA durch (lange) Leser möglich

Vorgänge

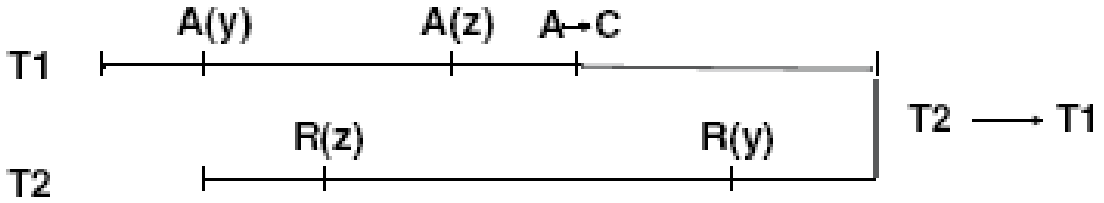
- 1 = Leseoperation ist sofort möglich
- 2 = Direktes Ändern nach Lesen
- 3 = Leseoperation muss auf das Ende der ändernden Transaktion warten
- 4 = Erzeugen einer lokal gültigen, neuen Version
- 5 = Konversion von A- in X-Sperre nach EOT der lesenden Transaktion



Prinzip

	R	A	C
R	+	+	⊕
A	+	-	-
C	⊕	-	-

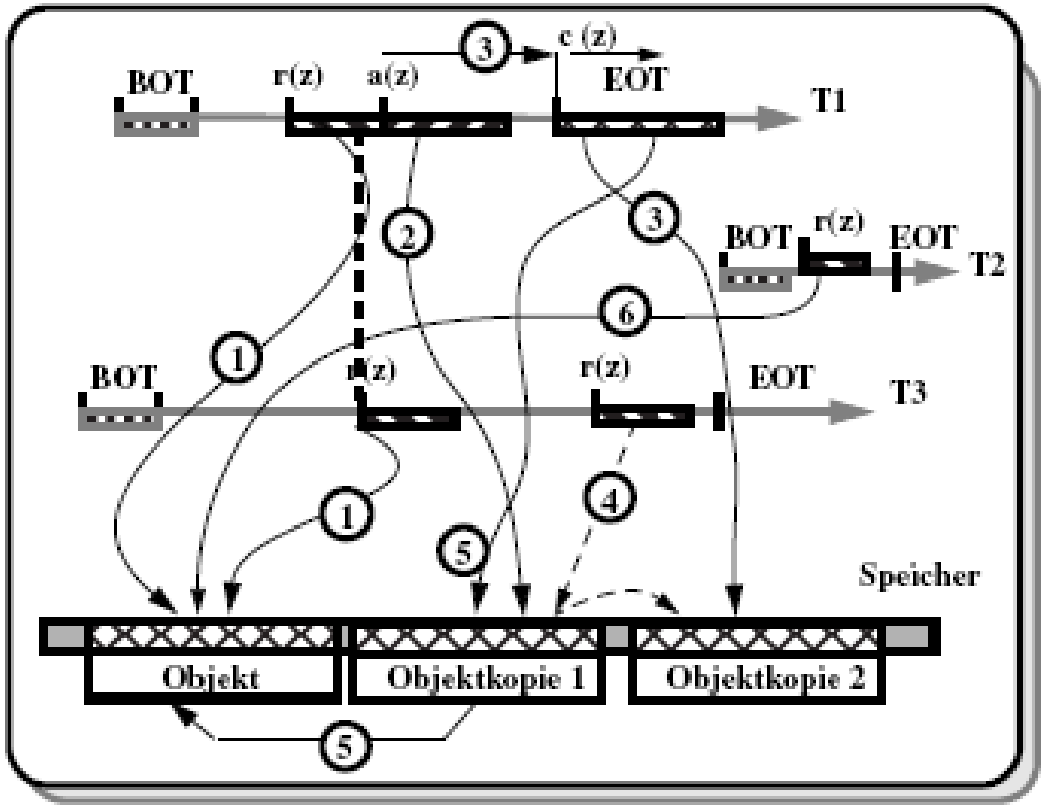
- Änderungen erfolgen ebenfalls in temporärer Objektkopie, A-Sperre erforderlich
- bei EOT Konversion von A → C-Sperre
- C-Sperre zeigt Existenz zweier gültiger Objektversionen an
  → kein Warten auf Freigabe von Lesesperren auf alter Version
  (R- und C-Modus sind verträglich)
- maximal zwei Versionen, da C-Sperren mit sich selbst und mit A-Sperren unverträglich sind



- Leseanforderungen bewirken nie Blockierung/Rücksetzung, jedoch:
  Auswahl der „richtigen" Version erforderlich
  (z. B. über Abhängigkeitsgraphen)
- Änderungs-TA, die auf C-Sperre laufen, müssen warten, bis alle Leser der alten Version beendet sind, weil nur zwei Versionen existieren
  - → **ABHILFE:** allgemeines Mehrversionen-Konzept

Vorgänge

- 1 = Leseoperation ist sofort möglich
- 2 = Erzeugen einer lokal gültigen, neuen Version
- 3 = Erzeugen einer neuen Kopie und Signalisierung durch Konversion von A- in C-Sperre nach EOT der lesenden Transaktion
- 4 = Leseoperation liest “passende” Kopie



- 5 = Einbringen der Kopie in die Datenbank zum allgemeinen Zugriff
- 6 = Leseoperation erfolgt auf die neue Version

Synchronisation von High-Traffic Objekten

- meist numerische Felder mit aggregierten Informationen
  z. B. Anzahl freier Plätze, Summe aller Kontostände

einfachste Lösung der Sperrprobleme

- Vermeidung solcher Felder beim DB-Entwurf

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	-	-	+

Escrow-Felder

- Deklaration von High-Traffic-Attributen
- 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 Bereichtests 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 Werteintervalls, das alle möglichen Werte nach Abschluss der laufenden TA umfasst

Eigenschaften

- 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

Beispiel

- Zugriffe auf Feld mit LO=0, HI=500 (Anzahl freier Plätze)
- Durchführung von Bereichstests bezüglich des Werteintervalls
- Minimal-/Maximalwerte (LO, HI) dürfen nicht überschritten werden
- hohe Parallelität ändernder Zugriffe möglich

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

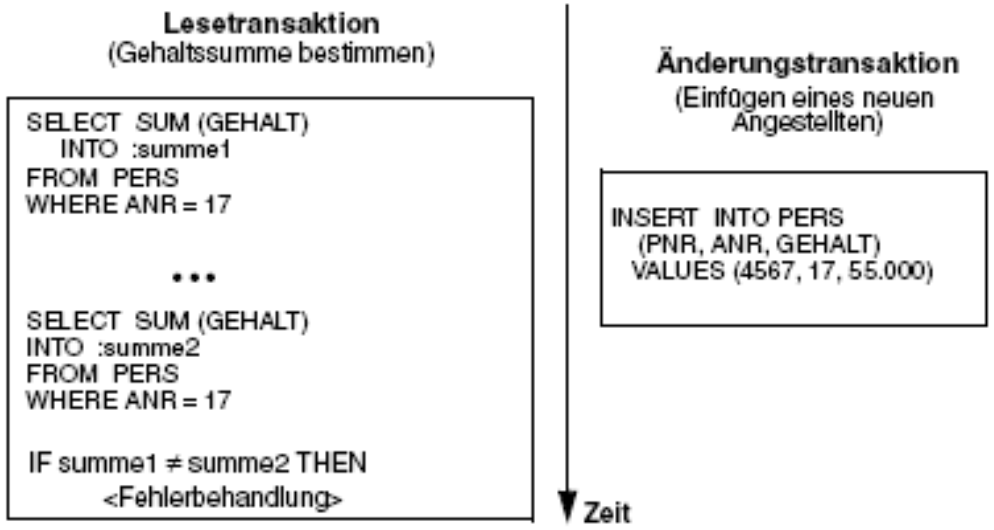
Hierarchische Sperren

Probleme mit einfachen X- und S- Sperren

- nicht effizient
  - aufwendig bei Transaktionen, die viele (oder alle) Tupel einer Relation sperren
  - große Sperrtabellen, hohe Verwaltungskosten
- nicht ausreichend, um alle Fehlerklassen auszuschließen

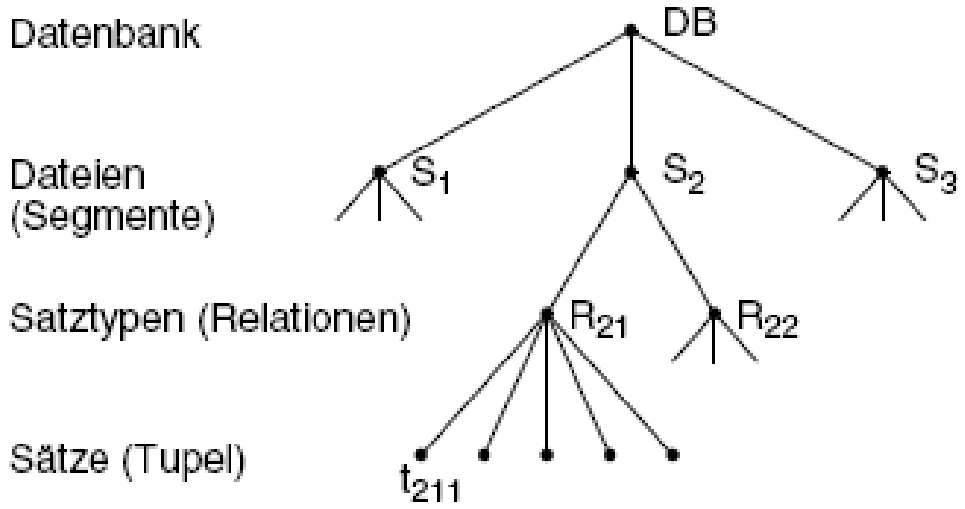
Phantom-Problem

- Sperren nur auf existierende Tupel
- Phantome
  (= scheinbar nicht existierende Tupel)

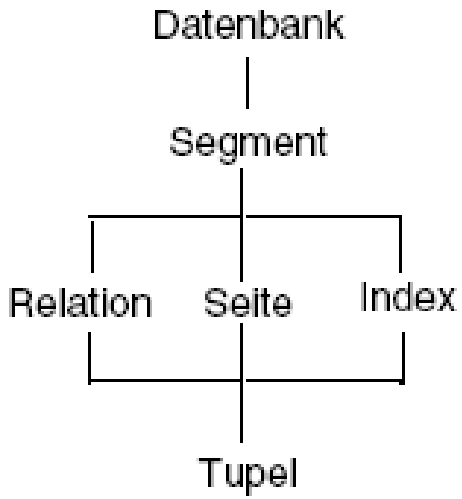


Hierarchisches Verfahren

- erlaubt Flexibilität bei der Wahl der Sperrgranulates
  - Synchronisierung langer TAs auf Relationenebene
  - Synchronisierung kurzer TAs auf Tupelebene



a) Beispiel einer Objekthierarchie



b) nicht-hierarchische Granularitäten

Nachteil von S- und X-Sperren

- alle Nachfolgeknoten werden implizit mit gesperrt
- alle Vorgängerknoten sind ebenfalls zu sperren, um Unverträglichkeiten zu vermeiden
  - X-Sperre auf DB: Einbenutzerbetrieb
  - S-Sperre auf DB: nur Lese-Transaktionen können parallel laufen

Verwendung von Anwartschaftssperren

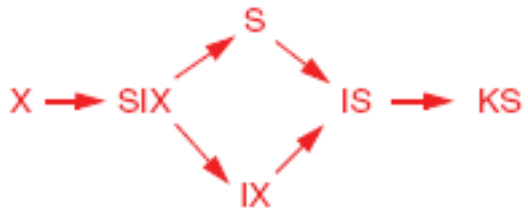
- I-Sperre oder Sperranzeige
  - IS-Sperre (intention share),
    falls auf untergeordnete Objekte nur lesend zugegriffen wird
  - IX-Sperre (intention eXclusive),
    falls auf untergeordnete Objekte schreibend zugegriffen wird
- Anwartschaftssperre
  - entspricht einer Sperre für Betriebsmittel auf einer höheren Hierarchiestufe
  - die Nutzung einer Untermenge wird angezeigt, in der Untermenge werden noch explizite Sperren gesetzt

Kombinierte Sperr-Sperranzeige

- SIX = S + IX (share and intention exclusive)
  - sperrt das Objekt in S-Modus
  - verlangt X-Sperren auf tieferer Hierarchieebene nur für zu ändernde Objekte
- sinnvoll für den Fall, in dem alle Tupel eines Satztyps gelesen und nur einige davon geändert werden
  - X-Sperre auf Satztyp sehr restriktiv
  - IX-Sperre auf Satztyp verlangt Sperren jedes Tupels

Kompatibilitätsmatrix

- Darstellung der Sperrmodi in einer Halbordnung (Dominanz-Relation)



	IS	IX	S	SIX	X
IS	+	+	+	+	-
IX	+	+	-	-	-
S	+	-	+	-	-
SIX	+	-	-	-	-
X	-	-	-	-	-

„Top-Down“ beim Erwerb von Sperren

- bevor ein Knoten mit S oder IS gesperrt wird, müssen alle Vorgänger in der Hierarchie von der Transaktion, die die Sperre anfordert, im IX oder im IS-Modus gesperrt werden
- bevor ein Knoten mit X oder IX gesperrt wird, müssen alle Vorgänger von der Transaktion, die die Sperre anfordert, im IX-Modus gehalten werden

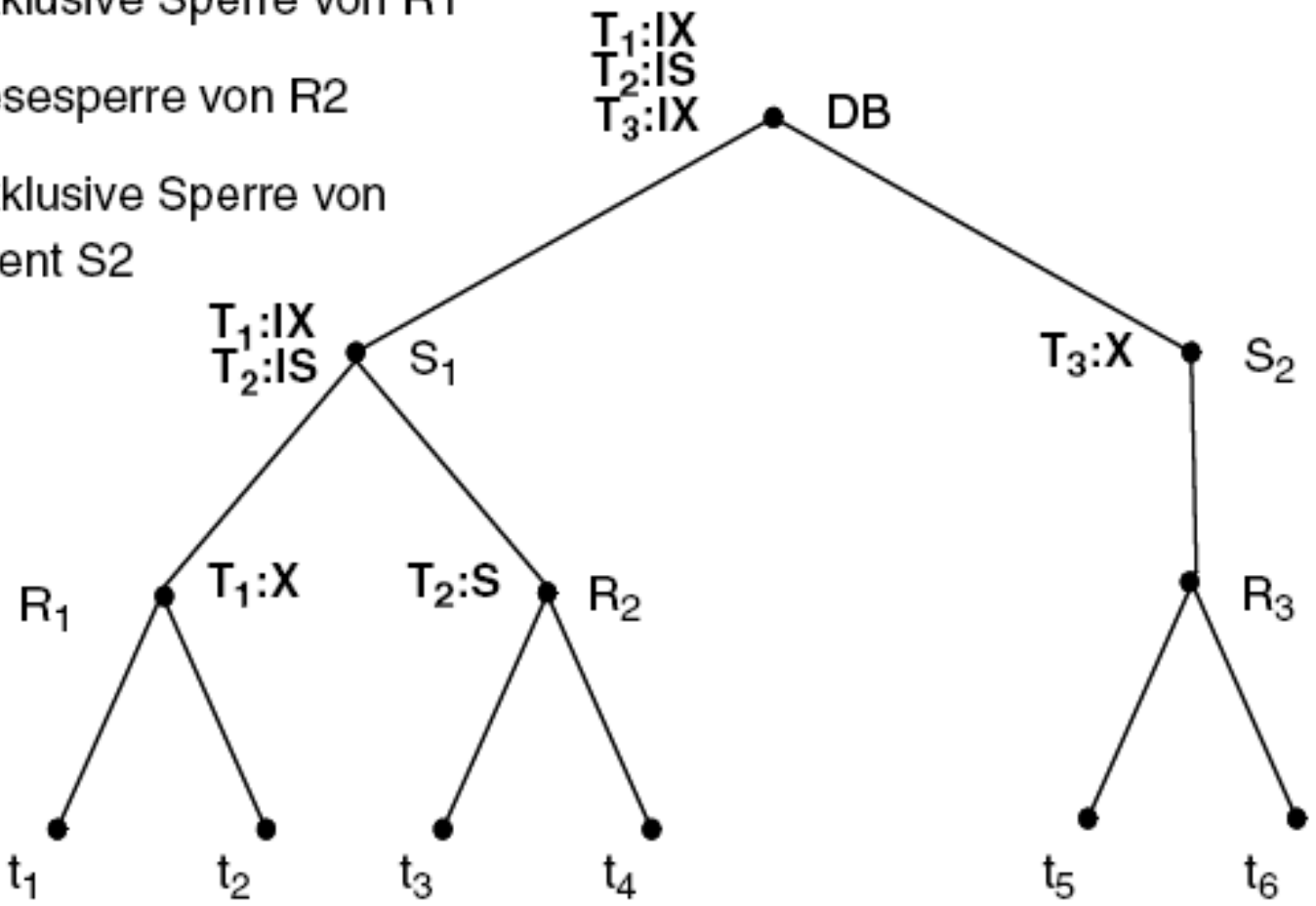
„Bottom-Up“ bei der Freigabe von Sperren

- Freigabe der Sperren von unten nach oben
- bei keinem Knoten wird die Sperre aufgehoben, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat

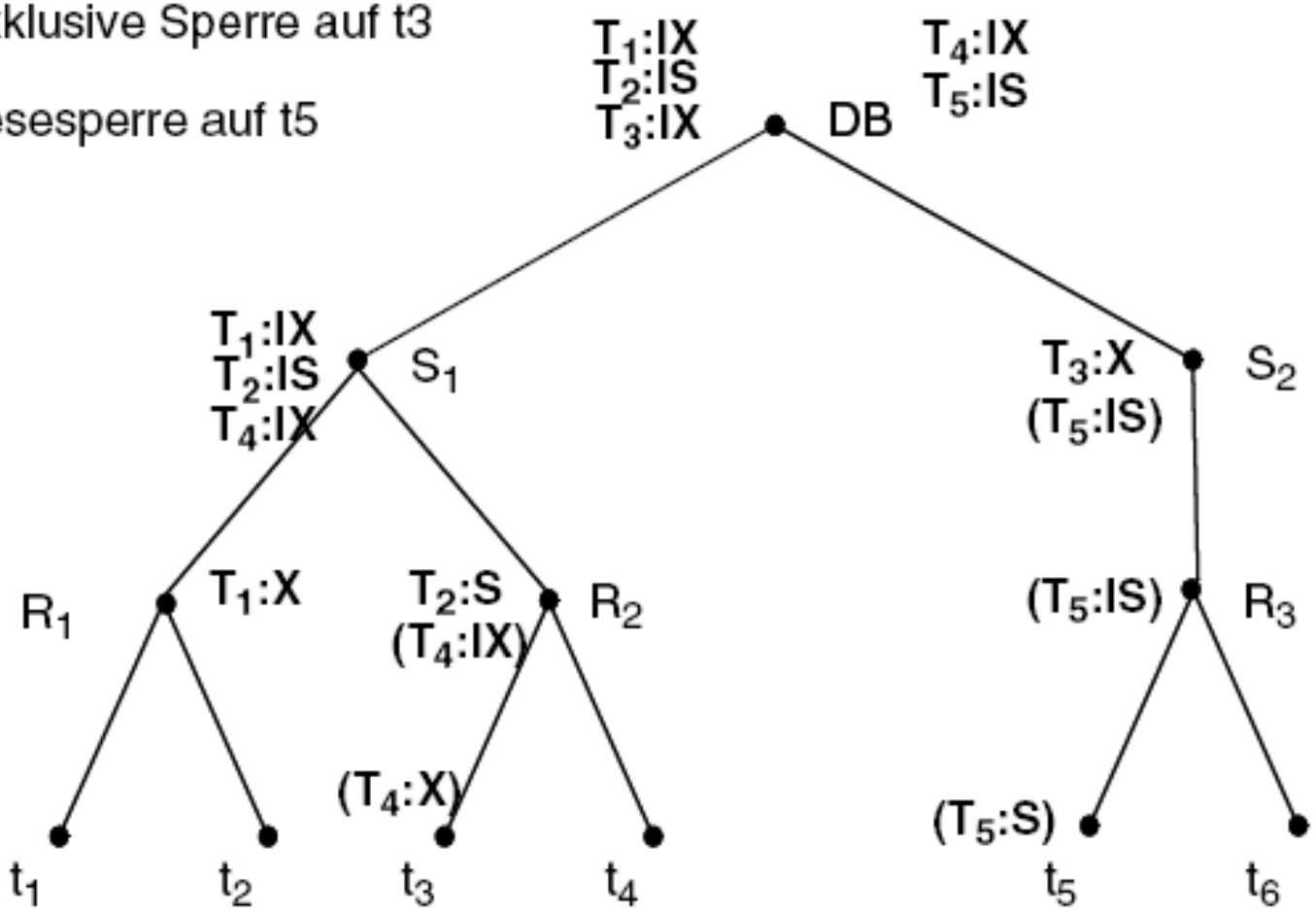
Beispiel

- *...*

- T_1: exklusive Sperre von R1
- T_2: Lesesperre von R2
- T_3: exklusive Sperre von Segment S2



- T_4: exklusive Sperre auf t3
- T_5: Lesesperre auf t5



Beispiel

- T1: SELECT SUM(gehalt)
  FROM pers

- T2: INSERT INTO pers (pnr, name, gehalt)´VALUES (4711, ‘Lehner‘, 30.000)

Sperranforderungen

- Anforderungen für T_1 Anforderungen für T_2

  a_1: Sperre DB mit IS a_2: Sperre DB mit IX
  b_1: Sperre S1 mit IS b_2: Sperre S1 mit IX
  c_1: Sperre PERS mit S c_2: Sperre PERS mit IX oder X

T_2 kann erst nach Beendigung von T_1 ausgeführt werden, da c_1 und c_2 unverträglich sind

Aber...

- Phantome treten bei kleineren Sperrgranulaten wieder auf!

Alternative Sperranforderung

- Anforderungen für T_1 Anforderungen für T_2

  a_1: Sperre DB mit IS — a_2: Sperre DB mit IX
  b_1: Sperre S1 mit IS — b_2: Sperre S1 mit IX
  c_1: Sperre PERS mit IS — c_2: Sperre PERS mit IX
  d_1: Sperre Tupel mit S — d_2: Sperre Tupel mit PNR = 4711 mit X

- *Phantome treten wieder auf, da c_1 und c_2 verträglich sind*

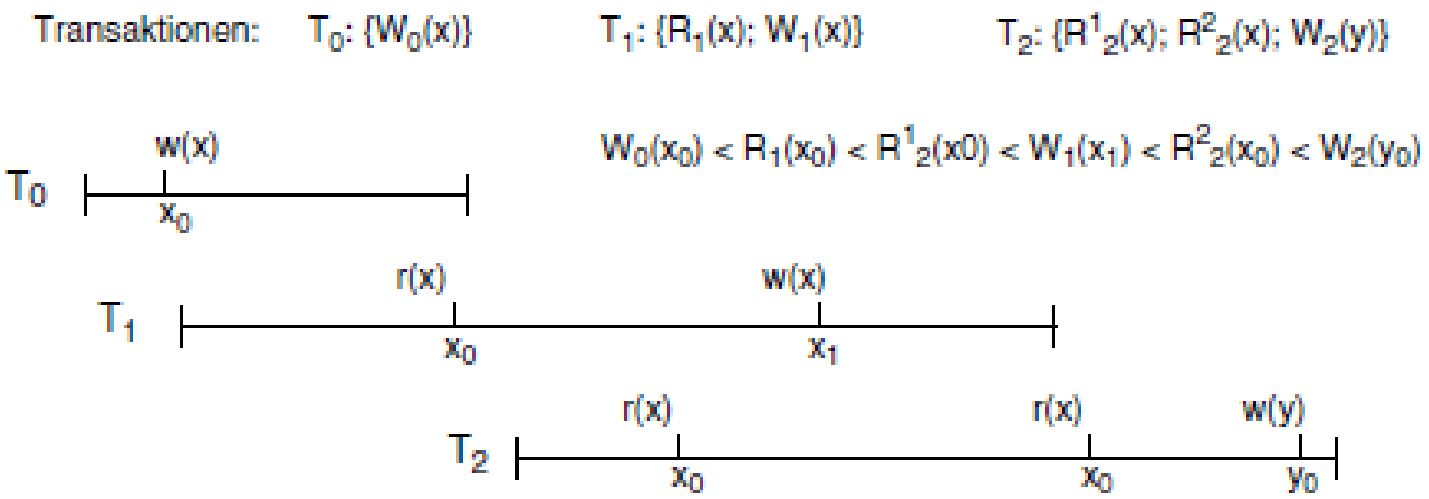
Merke

- Wahl des Sperrgranulates ist entscheidend !

Mehrversionenkonzept

Ausgangspunkt

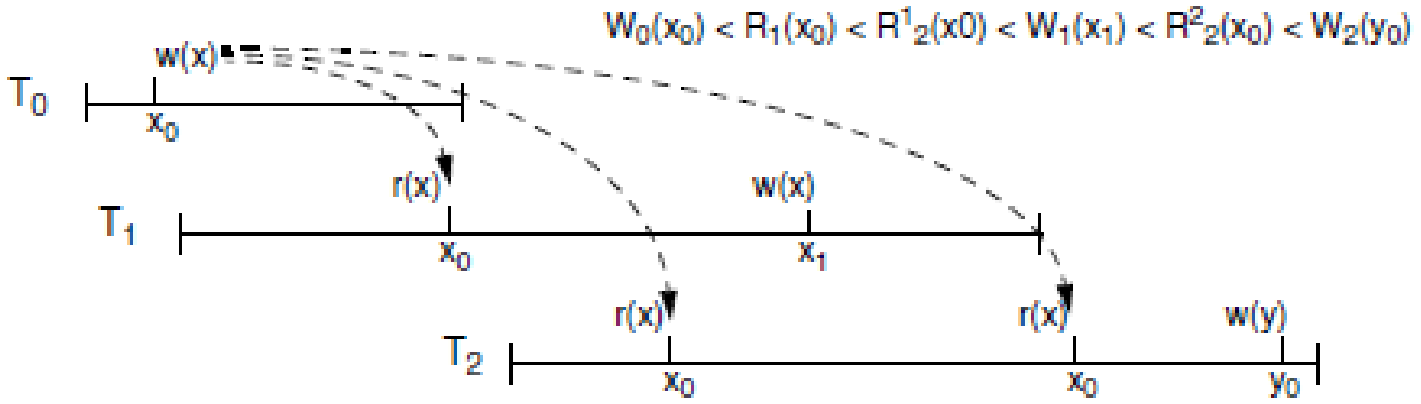
- Jede Schreiboperation auf einem Datenobjekt erzeugt eine neue Version
- DBMS verwaltet verschiedene Versionen eines Objektes
- DBMS stellt sicher, dass nur solche Versionen gelesen werden, die eine konsistente Sicht auf die Datenbank gewährleisten
- Änderungs-TA werden untereinander über ein allgemeines Verfahren (Sperren, …) synchronisiert



- MV-Historie: Es wird vermerkt, auf welche Version eines Objektes zugegriffen wurde

Äquivalenz von MV-Historien mit 1V-Historien

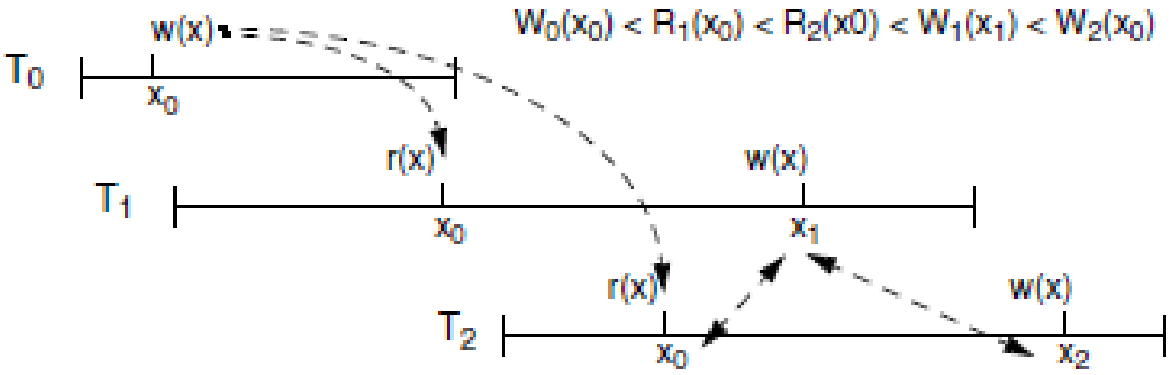
- Zeitliche Reihenfolge der Operationen spielt keine Rolle mehr!
- ”liest-von”-Relation: von welcher Schreiboperation wird gelesen?
- Beispiel
  - MV-Historie ist äquivalent mit 1V-Historie, wenn sie die gleiche ”liest-von”-Relation erfüllt



- MV-Historie ist MV-serialisierbar, wenn sie äquivalent ist mit einer seriellen 1V-Historie

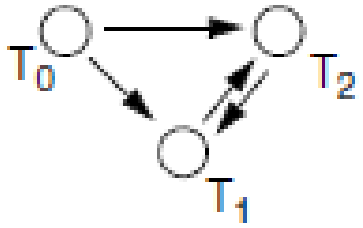
MV-Serialisierungsgraph

- basierend auf ”liest-von”-Relation
- Knoten: Abgeschlossene Transaktionen
- Zwischen zwei Knoten T_i und T_j im Serialisierungsgraphen gibt es eine gerichtete Kante von T_i nach T_j, wenn T_j eine Leseoperation enthält, die von einer Schreiboperation aus T_i liest
- Zusätzliche Kanten sind in den MV-Serialisierungsgraphen einzufügen, um die Versionsordnung abzubilden
  (Es muss eine eindeutige Versionsordnung geben!)
  - In einer 1V-Historie unterliegen alle Transaktionen, die ein Objekt x ändern, einer totalen Ordnung => Vorgegebene Versionsordnung durch Reihenfolge
  - Für jedes Paar von Lese-/Schreib-, Schreib-/Lese- und Schreib-/Schreiboperationen auf einem Objekt x ist eine Kante gemäß der Versionsordnung in den Serialisierungsgraphen einzutragen



Aufbau eines MV-Seriealisierungsgraphen für $x_0 < x_1 < x_2$

- Transaktionen = Knoten
- Kanten für „liest-von“-Relation
- $R_2(x) < W_1(x)$ wegen $x_0 < x_1$
- $W_1(x) < W_2(x)$ wegen $x_1 < x_2$



- Zyklus im MV-Serialisierungsgraphen
- Bei jeder anderen Versionsordnung ergibt sich ebenfalls ein Zyklus

Merke

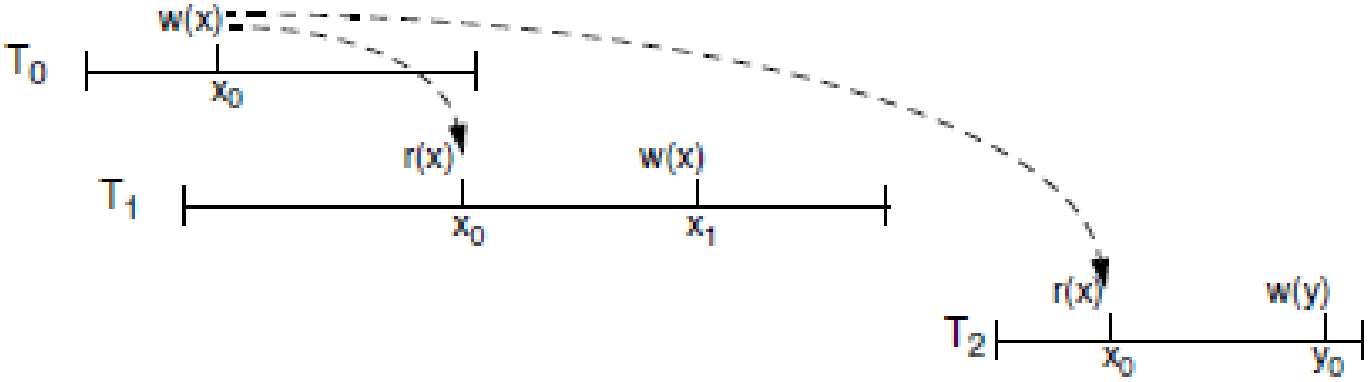
- Eine MV-Historie ist dann MV-serialisierbar, wenn es einen zyklenfreien MV-Serialisierungsgraphen gibt, bei dem für jedes vorkommende Objekt eine totale Versionsordnung definiert ist.

Fazit

- Mehrversionen-Serialisierbarkeit erlaubt Lesezugriffe auf veraltete Versionen eines Objektes!
- Tatsächlicher Lesezeitpunkt ist irrelevant für die Korrektheit!

Beispiel

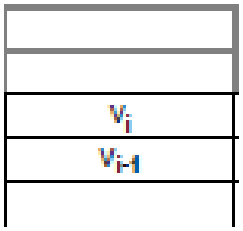
- Erhöhte Nebenläufigkeit wird durch eine verminderte Aktualität erkauft



Objekt Ok

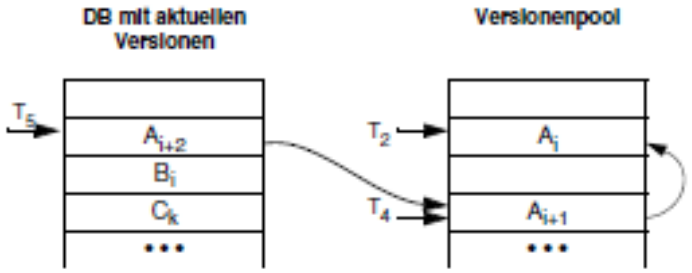
- zeitliche Reihenfolge der Zugriffe auf O_k

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}



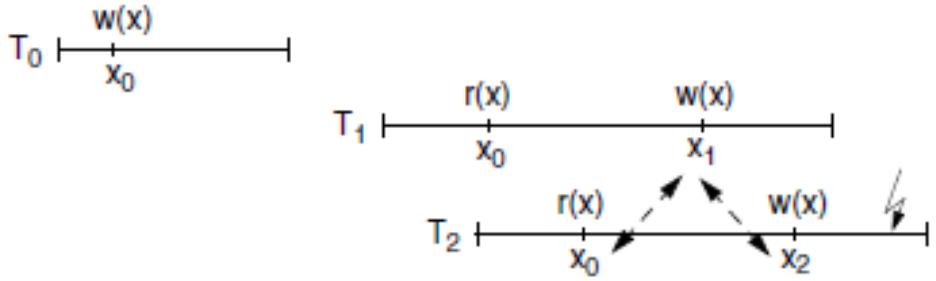
Speicherungsschema für Versionen

- Versionenpool: Teil des DB-Puffers
- Speicherplatzoptimierung:
  - Versionen auf Satzebene
  - Einsatz von Komprimierungstechniken



Eigenschaften

- Es können keine Verklemmungen auftreten
- Eine Transaktion muss nie warten, sondern erzeugt beim Schreiben immer eine neue Version
- Lese-TAs sehen den bei ihrem BOT gültigen DB-Zustand und müssen bei Synchronisation nicht mehr berücksichtigt werden
- Es kann vorkommen, dass Transaktionen zurückgesetzt werden müssen, weil keine MV-serialisierbaren Historien mehr erzeugbar sind



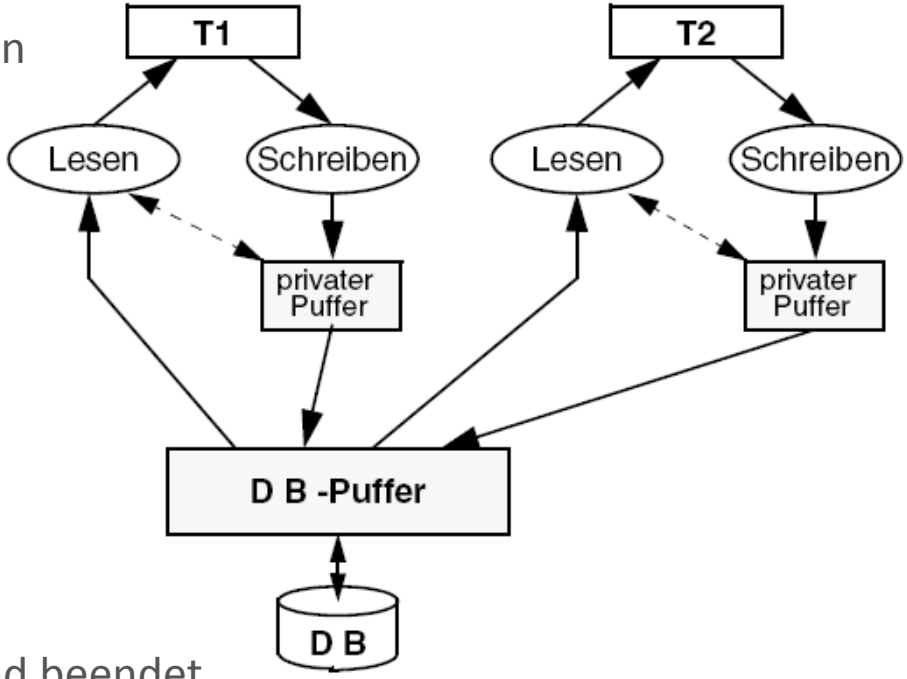
  - Normales Synchronisationsverfahren für Anderungstransaktionen
- zusätzlicher Speicher- und Wartungsaufwand für Versionenpoolverwaltung, Auffinden von Versionen, Garbage Collection, …

Optimistische Synchronisation

3-phasige Verarbeitung

Lesephase

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



Validierungsphase

- Überprüfung auf Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer der parallel ablaufenden Transaktionen
- Konfliktauflösung durch Zurücksetzen von Transaktionen

Schreibphase

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

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
  zur Durchführung der Validierungen werden pro Transaktion der Read-Set (RS) und Write-Set (WS) geführt

Forderung

- eine TA kann nur erfolgreich validieren, wenn sie alle Änderungen von zuvor validierten TA gesehen hat
  ➥ Validierungsreihenfolge bestimmt Serialisierungsreihenfolge

Validierungsstrategien

- Backward Oriented (BOCC): Validierung gegenüber bereits beendeten TA
- Forward Oriented (FOCC): Validierung gegenüber laufenden TA

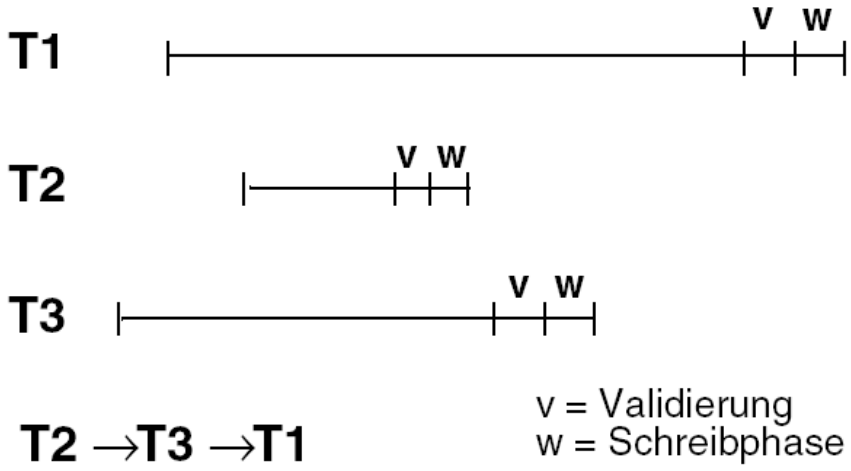
Validierung von Transaktion T

- 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

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

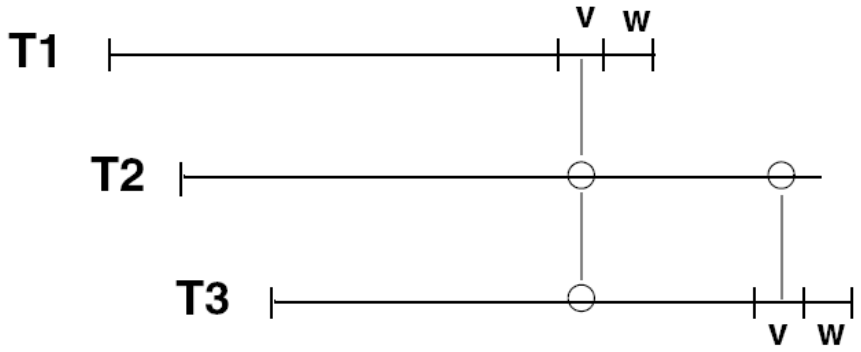


Prinzip

- nur Änderungs-TA validieren gegenüber laufenden TA T_i

Validierungstest: $(RS(T) \cap WS(T_j) \neq \emptyset)$

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 muss WS (T) 'gesperrt' sein, damit sich die $RS(T_i)$ nicht ändern (keine Deadlocks damit möglich)
- immer noch hohe Rücksetzrate möglich
- es kann immer nur einer TA Durchkommen definitiv zugesichert werden

Zusammenfassung

Mögliche Anomalien ohne Synchronisation

- Verlorengegangene Änderungen (lost updates)
- Abhängigkeiten von nicht freigegebenen Änderungen (dirty read, dirty overwrite)
- Inkonsistente Analyse (non-repeatable read)
- Phantom-Problem (in DB2)

Lösung: Serialisierung

- aber: bei langen TA zu große Wartezeiten für andere TA (Scheduling-Fairness)
- Sicherstellen der operationellen Integrität durch eine „virtuelle“ serielle Ausführung der einzelnen Operationen einer Transaktion
  (logischer Einbenutzerbetrieb!)