

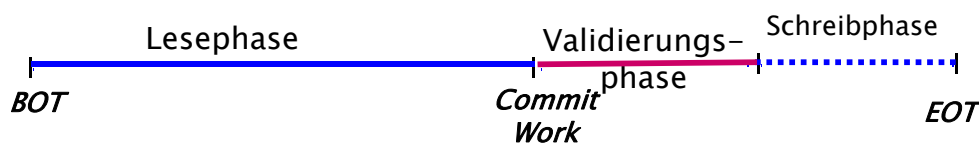
3. Synchronisation: Weitere Verfahren, Leistungsbewertung

- Optimistische Synchronisation
 - BOCC, FOCC
 - Verwendung von Zeitstempeln (BOCC+)
- Mehrversionen-Synchronisation
 - allgemeines Konzept, Versionenpool-Verwaltung
 - Snapshot Isolation
 - Nutzung für In-Memory-DBS
- Spezialverfahren
 - Prädikatsperren (Behandlung Phantom-Problem)
 - Synchronisation auf "High Traffic"-Elementen (IMS, Escrow-Verfahren)
 - Sperren von B*-Bäumen
- Leistungsbewertung, Lastkontrolle



Optimistische Synchronisation (OCC)

- Grundannahme: geringe Konfliktwahrscheinlichkeit
- Transaktionsverarbeitung in 3 Phasen



- **Lesephase**: eigentliche Transaktionsverarbeitung
 - Änderungen einer Transaktion erfolgen in privatem Puffer
- **Validierungsphase**: Serialisierbarkeitsprüfung
 - Test auf (RW-, WR-, WW-) Konflikte mit parallelen Transaktionen
 - Konfliktauflösung durch Zurücksetzen von Transaktionen
- **Schreibphase**
 - nur für Änderungstransaktionen und nur bei positiver Validierung
 - Logging und Propagieren der Änderungen



Optimistische Synchronisation (2)

■ allgemeine Eigenschaften von OCC:

- + keine Dirty Reads möglich
- + einfache Transaktionsrücksetzung
- + keine Deadlocks
- mehr Rücksetzungen als bei Sperrverfahren
- Gefahr des „Verhungerns“ (starvation) von Transaktionen



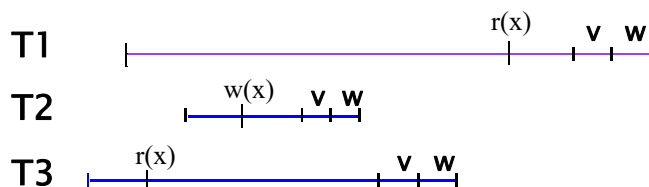
BOCC

■ ursprünglich vorgeschlagenes Verfahren zur optimistischen Synchronisation [Kung&Robinson, ACM TODS 1981]

■ Validierung von Transaktion T:

BOCC-Test gegenüber allen Änderungstransaktionen T_j , die seit BOT von T erfolgreich validiert haben (Annahme $WS(T) \subseteq RS(T)$):

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



v = Validierung
w = Schreibphase



BOCC (2)

■ Nachteile/Probleme:

- unnötige Rücksetzungen wegen ungenauer Konfliktanalyse
- Rücksetzung erst bei EOT => viel unnötige Arbeit
- es kann nur die validierende Transaktion zurückgesetzt werden
=> Gefahr des „Verhungerns“
- hohes Rücksetzrisiko für lange Transaktionen und bei Hot-Spots

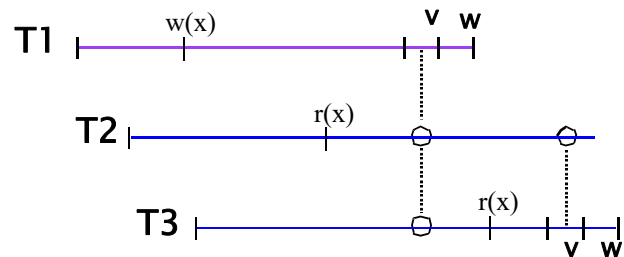


FOCC

■ nur Änderungstransaktionen validieren gegenüber laufenden Transaktionen T_i

Validierungstest:

$$WS(T) \cap RS(T_i) = \emptyset$$



■ Vorteile:

- Wahlmöglichkeit des Opfers (Kill, Rollback, Prioritäten, ...)
- keine unnötigen Rücksetzungen
- frühzeitige Rücksetzung möglich => Einsparen unnötiger Arbeit

■ Probleme:

- während Validierungs- und Schreibphase ist $WS(T)$ zu 'sperren', damit sich die $RS(T_i)$ nicht ändern (keine Deadlocks damit möglich)
- immer noch hohe Rücksetzrate möglich



BOCC+

■ Verwendung von Zeitstempeln (Änderungszähler)

- erfolgreich validierte Transaktionen erhalten eindeutige, monoton wachsende *Transaktionsnummer*
- geänderte Objekte erhalten *Zeitstempel TS* (=Transaktionsnummer der ändernden Transaktion)
- beim Lesen eines Objektes wird Zeitstempel *ts* der gesehenen Version im Read-Set vermerkt

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

■ Validierung

- Zeitstempelvergleich, ob gesehene Objektversionen zum Validierungszeitpunkt noch aktuell
- Verwalten einer *Objekttabelle* mit TS-Zeitstempel zuletzt geänderter Objekte



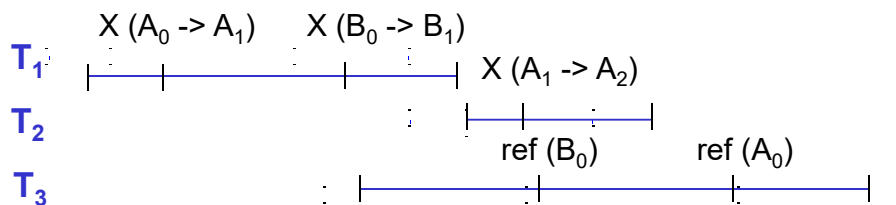
BOCC+ (2)

- zum Scheitern verurteilte Transaktionen können sofort zurückgesetzt werden
- Vorteile BOCC+
 - keine unnötigen Rücksetzungen
 - frühzeitiges Abbrechen zum Scheitern verurteilter Transaktionen
 - schnelle Validierung
- Probleme:
 - wie bei BOCC können einzelne Transaktionen verhungern
 - potenziell hohe Rücksetzrate
- Verbesserungsmöglichkeit durch Reduzierung der Konfliktwahrscheinlichkeit, z.B. durch
 - geringere Konsistenzebene (Lesetransaktionen werden bei Validierung nicht mehr berücksichtigt)
 - Mehrversionen-Konzept



Mehrversionen-Konzept (MVCC)

- jede Änderung erzeugt neue Objektversion
- Lesetransaktionen sehen den bei ihrem BOT gültigen DB-Zustand
=> *reihenfolgeerhaltende Serialisierbarkeit*
- keine Synchronisation mehr für Lese-Transaktionen !
 - keine Blockierungen und Rücksetzungen für Leser, dafür ggf. Zugriff auf leicht veraltete Objektversionen
 - Änderungstransaktionen werden untereinander über allgemeines Verfahren (Sperren, OCC, ...) synchronisiert
 - erheblich weniger Synchronisationskonflikte
- mittlerweile starke Nutzung in kommerziellen DBS
 - Oracle, PostgreSQL, fast alle NewSQL-Systeme (SAP Hana, VoltDB, Google Spanner)

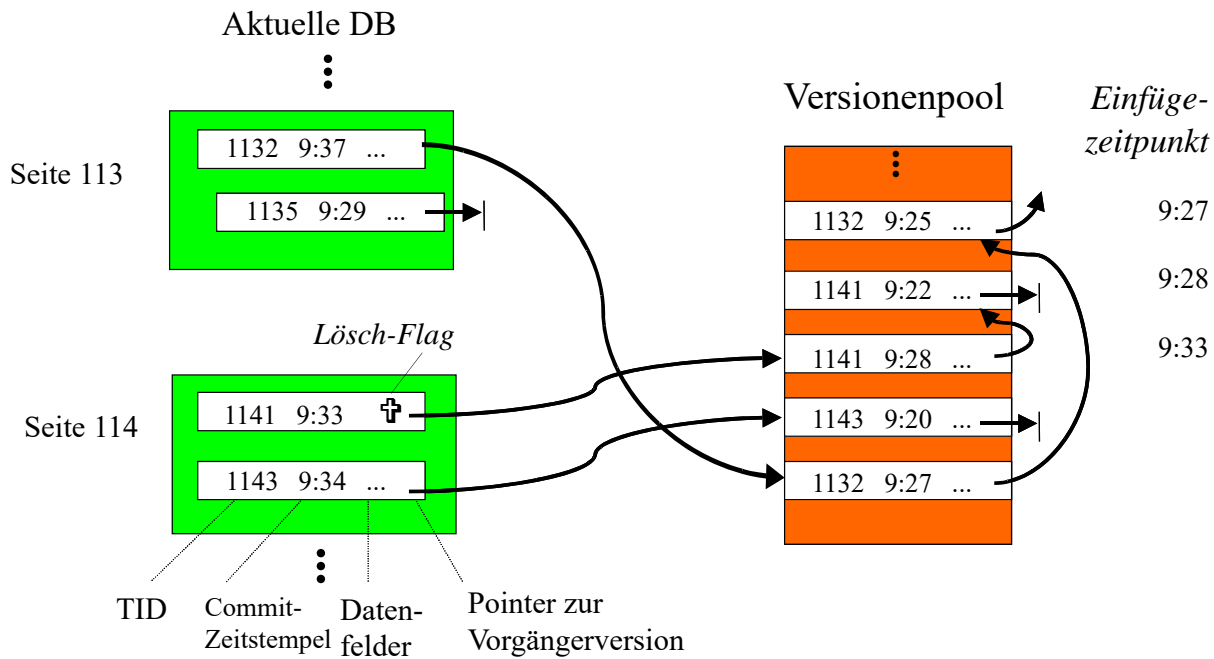


Versionenpool-Verwaltung

- zusätzlicher Speicher- und Wartungsaufwand
 - Versionenpool-Verwaltung
 - Auffinden von Versionen
 - Garbage Collection
- Realisierungsmöglichkeit
 - Lesetransaktionen erhalten BOT-Zeitstempel
 - Änderungstransaktionen kennzeichnen geänderte Objekte mit Commit-Zeitstempel
 - alte Objektversionen kommen in Versionenpool; Verkettung der Versionen eines Objektes
- Lesetransaktionen greifen auf jüngste Objektversionen mit Zeitstempel kleiner dem Transaktionsbeginn zu
- Objektversion V kann freigegeben werden
 - sobald jüngere Version V_j existiert mit Zeitstempel kleiner dem *Beginn der ältesten Lesetransaktion* im System



Versionenpool-Verwaltung: Beispiel



Lesetransaktion T mit BOT-Zeitstempel 9:35

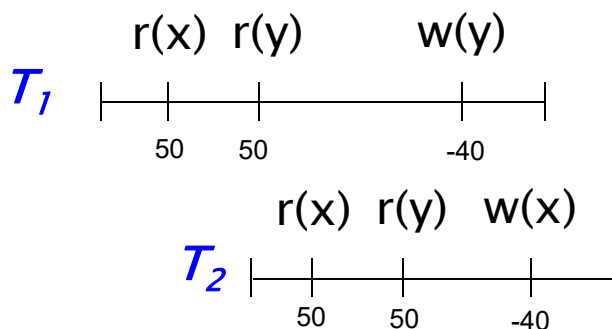
Zugriff auf Objekt 1132:

Objekt 1143:

neue älteste Lesetransaktion T – Freigabe von

Snapshot Isolation (SI)

- MVCC-Variation auch für Update-Transaktionen
- quasi-optimistische Synchronisation für Änderer („First Committer Wins“)
 - zunächst private Änderungen für Änderungstransaktion T
 - Commit erfolgreich, wenn keine andere Transaktion mit überlappendem Write-Set seit Start von T Commit vorgenommen hat
- i.a. keine Serialisierbarkeit



Serializable Snapshot Isolation*

- erweiterte Validierung von Update-Transaktionen, um neben WW-Konflikten auch RW-Konflikte zu erkennen
 - nicht-serialisierbare Situationen für SI entstehen durch Zyklen mit mind. zwei RW-Konflikten zwischen parallel laufenden Transaktionen
- approximative Lösung
 - zwei Flags für Transaktion T: *inConflict* (falls RW-Konflikt $T_x \rightarrow T$) und *OutConflict* (falls RW-Konflikt $T \rightarrow T_y$)
 - potenzielle Serialisierbarkeitsverletzung für T, wenn sowohl *inConflict* als auch *OutConflict* vorliegt (-> Abbruch von T)
 - Setzen der Flags, wenn T Objektversion liest, für die neuere Version durch T_y vorliegt (*OutConflict* für T, *InConflict* für T_y)
 - zusätzlich SI-Read- und Write-Locks zu setzen : Write-Sperre von T bei gesetztem SI-Read durch T_x führt zu *InConflict* für T, *OutConflict* für T_x
 - SI-Read-Sperren auch nach Transaktionsende zu halten (solange bis alle parallele Transaktionen beendet)

*Cahill, Röhm, Fekete: *Serializable isolation for Snapshot Databases*. Proc. SIGMOD 2008



Mehrversionen-Synchronisation für In-Memory-DBS*

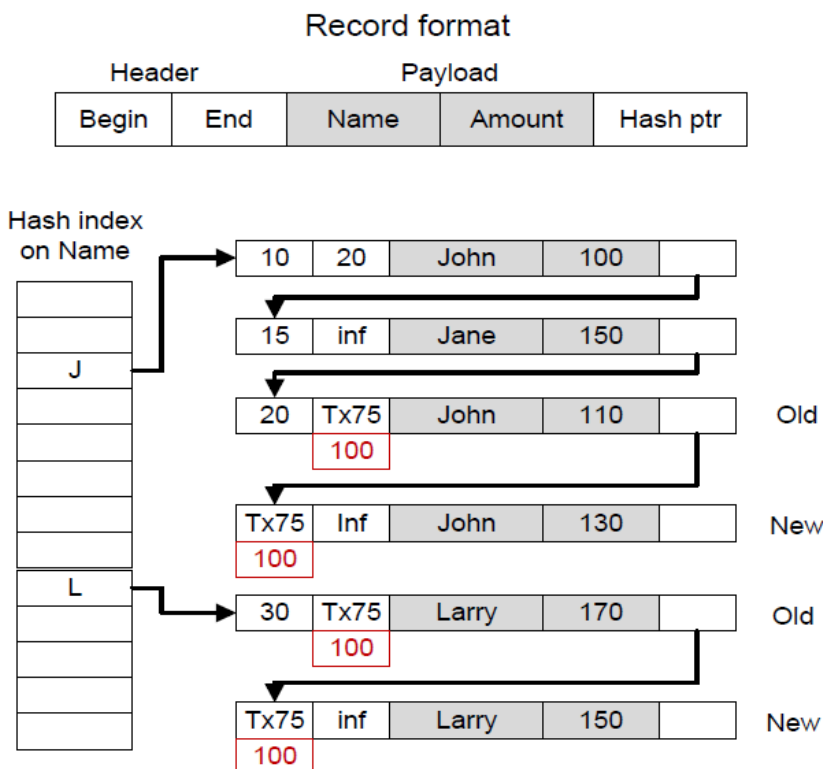


Figure 1: Example account table with one hash index. Transaction 75 has transferred \$20 from Larry's account to John's account but has not yet committed.

■ Zeitstempelvergabe

- Transaktionen erhalten Beginn- und Commit-Timestamp (von Zähler abgeleitet)
- Objekte haben Beginn/Ende-Zeitstempel bzgl ihrer Gültigkeit
- Beginn-Zeitstempel entspricht Commit-TS der Änderungs-Transaktion
- Ende-Zeitstempel offen (infinite) für aktuelle Version, sonst Beginn-Zeitstempel der Nachfolgeversion
- in Vorbereitung befindliche Objektversionen erhalten temporär Transaktionsid als Beginn-Timestamp (frühzeitige Erkennung von Konflikten)

*P. Larson et al.: *High-Performance Concurrency Control Mechanisms for Main-Memory Databases*. Proc. VLDB 2012.

C. Diaconu et al.: *Hekaton: SQL Server's memory-optimized OLTP engine*. Proc. Sigmod 2013



Mehrversionen-Synchronisation für In-Memory-DBS (2)

■ Lesen durch Transaktion T:

- Lesen der Objektversion, für die BOT-TS (T) im Beginn/Ende-Intervall des Objektes liegt (nur für max. 1 Objektversion möglich)

■ Änderung:

- Erzeugen einer neuer Objektversion mit Beginn-Zeitstempel = Tx-ID; Ende-Zeitstempel der Vorgängerversion wird auch auf Tx-ID gesetzt
- Transaktionsabbruch falls bereits ein anderes Update im Gange (d.h. es existiert bereits eine Version mit Tx-Id als Beginn-Zeitstempel)

■ Validierung/Commit (nur für Änderungs-Transaktionen)

- Validierung: prüfe für jedes gelesene Objekt im Read-Set, ob gelesene Version weiterhin gültig. Wenn nicht erfolgt Abbruch der Transaktion
- Logging der Änderungen
- Markieren der vorbereiteten Objektversionen mit Commit-TS der Transaktion (anstelle der Transaktions-ID) -> Änderungen sind für andere Transaktionen nutzbar



Logische Sperren (Prädikate)

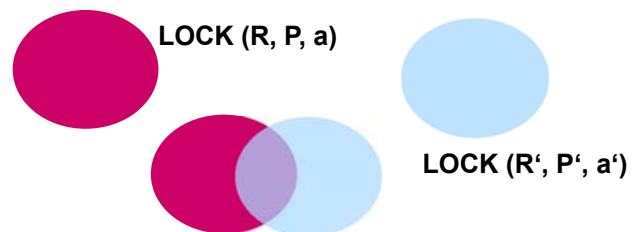
■ Verhütung des Phantomproblems \Rightarrow Prädikatssperren

■ Operationen: LOCK(R, P, a), UNLOCK (R, P)

R Relationenname, P Prädikat, $a \in \{\text{read, write}\}$

■ wie kann Konflikt zwischen Prädikaten festgestellt werden?

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



■ i.a. unentscheidbar, selbst mit arithmetischen Prädikaten

Bsp.: T1: LOCK (PERS, ALTER > 50, read)

T2: LOCK (PERS, PNR = 4711, write)

■ pessimistische Entscheidungen \Rightarrow Einschränkung d. Parallelität

- Sonderfall: P = TRUE entspricht einer Relationensperre



Prädikatsperren (2)

- effizientere Implementierung durch sog. *Präzisionssperren*
- nur gelesene Daten werden durch Prädikat gesperrt, Schreibsperren werden für Sätze gesetzt
 - kein Disjunktheitstest für Prädikate mehr, sondern lediglich Test, ob Satz ein Prädikat erfüllt
- Datenstrukturen:
 - Prädikatliste: pro Relation werden Lesesperren laufender Transaktionen durch Prädikate beschrieben
 - Update-Liste: geänderte Sätze laufender Transaktionen
- Leseanforderung (Prädikat P):
 - für jeden Satz der Update-Liste ist zu prüfen, ob es P erfüllt
 - wenn ja -> Sperrkonflikt
- Schreibenanforderung (Satz t):
 - Schreibsperre wird gewährt, wenn t keines der Leseprädikate erfüllt



Synchronisation von High-Traffic-Objekten

- High-Traffic-Objekte:
 - meist numerische Felder mit aggregierten Informationen
 - z.B. Anzahl freier Plätze, Summe aller Kontostände
- einfachste Lösung: 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	-	-	+

- Problem:
 - Inkrementieren/Dekrementieren erfordert i.a. vorheriges Lesen des Objektes
 - R-Sperre jedoch inkompatibel mit Inc/Dec-Sperre



IMS Fast Path-Ansatz

- spezielle Operationen für High-Traffic-Objecte:

VERIFY #Plätze \geq Anforderung

MODIFY #Plätze := #Plätze - Anforderung

- quasi-optimistische Synchronisation:

- zunächst werden keine Sperren gesetzt
- Änderungen zunächst nur in 'work-to-do-list' vermerkt
- bei EOT Validierung- und Schreibphase:
Überprüfung, ob VERIFY-Prädikate noch erfüllt sind (geringe Rücksetzwahrscheinlichkeit) sowie Durchführung von Inkrement/Dekrement

- Sperren werden nur für Dauer der EOT-Behandlung gehalten

- weit geringere Konfliktgefahr als mit normalen Schreibsperren



Escrow-Ansatz

- Deklaration von High-Traffic-Attributen als Escrow-Felder

- Benutzung spezieller Operationen auf Escrow-Feldern

- Reservierung 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 *Bereichstests* bei Escrow-Anforderung

- es wird garantiert, dass Prädikat nachträglich nicht mehr invalidiert wird (keine spätere Validierung/Zurücksetzung)

- gleichzeitiger Objektzugriff durch mehrere Änderer, solange Prädikate erfüllt bleiben

- Nachteil: spezielle Programmierschnittstelle (wie bei IMS FP)



Escrow-Ansatz (2)

- aktueller Wert eines Escrow-Feldes ist unbekannt, solange Reservierungen bestehen
 - Führen eines Wertintervalls für Escrow-Felder, das alle möglichen Werte nach Abschluss der laufenden Transaktionen 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 Rollback einer Transaktion
 - Minimal-/Maximalwerte (LO, HI) sind einzuhalten
 - Durchführung von Bereichstests bezüglich des Wertintervalls
- Beispiel: Zugriffe auf Feld mit $LO=0$, $HI=500$ (# freier Plätze)

Anforderung/Rückgaben				Wertintervall		
T1	T2	T3	T4	INF	Q	SUP
				15	15	15
-5				10	10	15
	-8					
		+4				
Commit						
	Rollback					
		Commit				



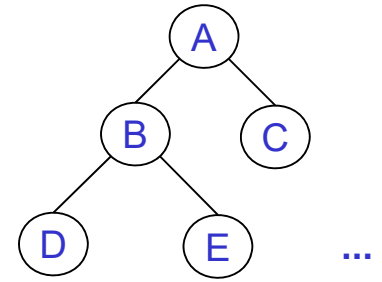
Escrow-Ansatz (3)

- Escrow-Operationen führen zur Vergrößerung des Intervalls
 - Anforderungen ($C1 < 0$): INF wird um $C1$ dekrementiert, falls LO nicht unterschritten wird
 - Rückgabe ($C1 > 0$): SUP wird um $C1$ inkrementiert, falls HI nicht überschritten wird
- Commit/Rollback führen zur Reduzierung des Intervalls
 - Dekrementierung von SUP: Commit für Anforderung ($C1 < 0$) bzw. Rollback für Rückgabe
 - Inkrementierung von INF: Commit für Rückgabe ($C1 > 0$) bzw. Rollback für Anforderung
- nach Intervallanpassung ggf. wartende Escrow-Operationen bedienen

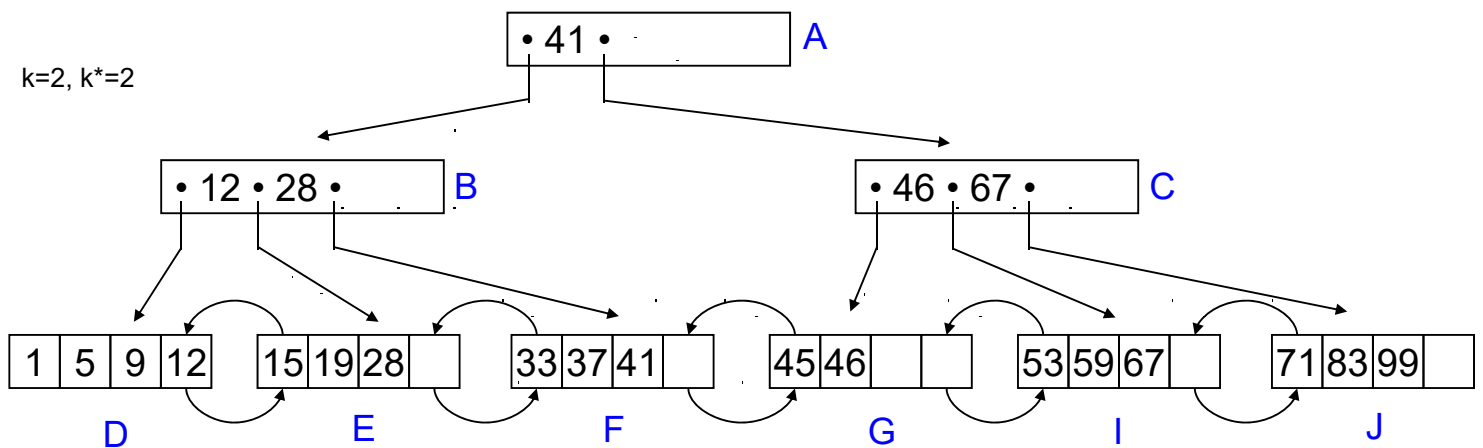


Sperren in B*-Bäumen

- (lange) Seitensperren führen zu inakzeptablen Behinderungen
 - v.a. Wurzelknoten und Zwischenknoten werden zu Engpässen
 - zahlreiche Einträge auch pro Blatt
- Nutzung der Baum-Charakteristika
 - alle Operationen starten bei Wurzel
 - Zwischenknoten im Baum steuern nur die Suche nach Blattknoten
 - nur Inserts/Deletes, die Baumstruktur ändern, erfordern exklusive Sperren des Pfades von der Wurzel
- Baumsperrrprotokoll (Lock Coupling)
 - Seite (außer Wurzel) kann erst gesperrt werden, wenn Vorgänger gesperrt ist („lock coupling“)
 - Sperre auf Vorgänger kann freigegeben werden, wenn seine spätere Änderung - z.B. wegen Splitting - ausgeschlossen
 - garantiert Serialisierbarkeit trotz fehlender Zweiphasigkeit



Sperren in B*-Bäumen : Beispiel



1. Suche 37

R-Lock A
R-Lock B

2. Insert 27

X-Lock A
X-Lock B

3. Insert 10

X-Lock A
X-Lock B



Methoden zur Leistungsbewertung von DBS

1.) Kostenformeln

- nur für grobe Abschätzungen brauchbar („back of the envelope“)

2.) analytische Modelle

- z.B. Warteschlangenmodelle
- oft starke Vereinfachungen erforderlich

3.) Simulationen

- sehr geeignet zum Vergleich verschiedener Realisierungsalternativen (Verfahren können direkt implementiert werden)
- synthetische Lasten vs. Trace-getrieben (reale Lasten)

4.) Benchmarks

- Messungen mit DBS-Prototyp bzw. realem DBS
- Bewertung / Vergleich bestimmter Systeme



Wahrscheinlichkeit von Sperrkonflikten und Deadlocks

■ Annahmen

- m Datenbankobjekte
- n parallele Transaktionen (Multiprogramming Level)
- k Objektzugriffe pro Transaktion
- nur exklusive Sperren
- Gleichverteilung der DB-Zugriffe

■ mittlere Anzahl von Sperren pro (laufender) Transaktion: $\bar{L} \approx \frac{k}{2}$

■ Wahrscheinlichkeit eines Sperrkonfliktes für den i-ten Objektzugriff einer Transaktion T:

$$P_c = \frac{\text{\#Sperren anderer Transaktionen}}{\text{\#DB-Objekte, die nicht von T gesperrt}} = \frac{(n-1) \bar{L}}{m - (i-1)} \approx \frac{(n-1) k}{2 m}$$

■ Wahrscheinlichkeit, dass eine Transaktion einen Sperrkonflikt erfährt:

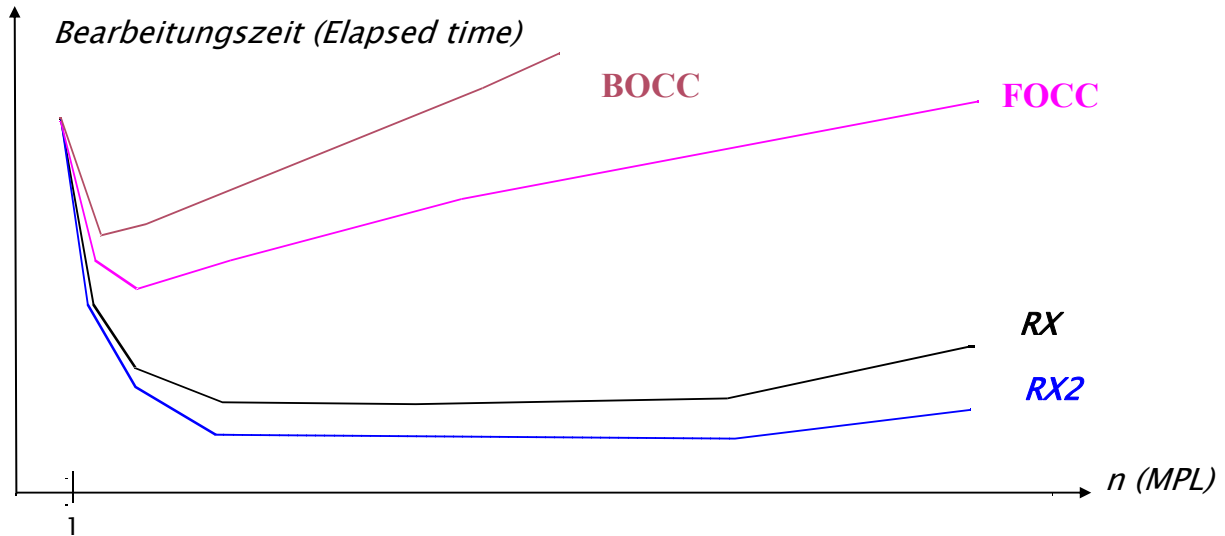
$$P_w \approx k \cdot P_c \approx \frac{(n-1) k^2}{2 m}$$

■ Deadlock-Wahrscheinlichkeit zwischen zwei Transaktionen T1 und T2:

$$\text{Pr [T1 -> T2]} \cdot \text{Pr [T2 -> T1]} \cdot \text{\#Kandidaten für T2} = \left(\frac{P_w}{n-1} \right)^2 (n-1) = \frac{(n-1) k^4}{4 m^2}$$



Bewertung von Synchronisationsverfahren mit Trace-basierter Simulation (Peinl, 1987)

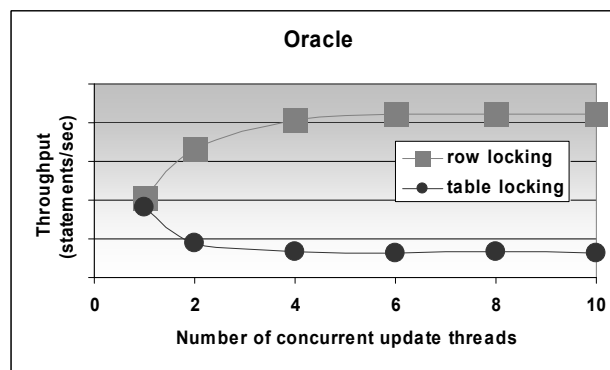
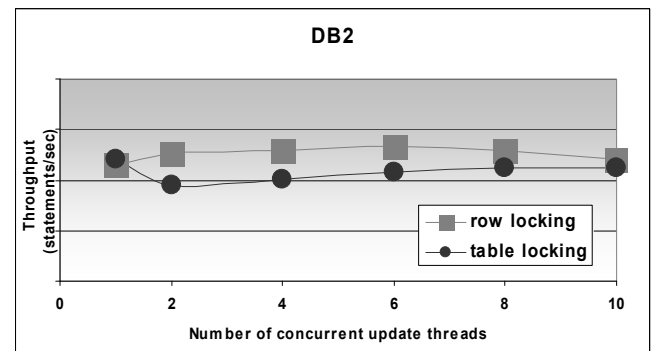
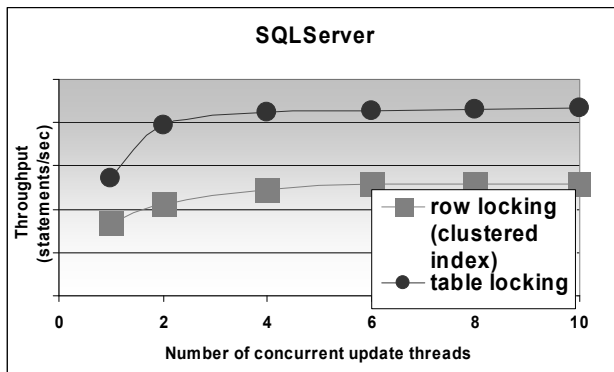


- sehr geringe Parallelität → keine effektive Nutzung der Ressourcen
- geringe Parallelität → bester Durchsatz, nicht notwendigerweise kürzeste Antwortzeiten
- pessimistische Methoden gewinnen: Blockierung vermeidet häufig Deadlocks
- optimistische Methoden geraten leicht in ein Thrashing-Verhalten
- RX2 (kurze Lesesperren) reduziert effektiv Konfliktrate

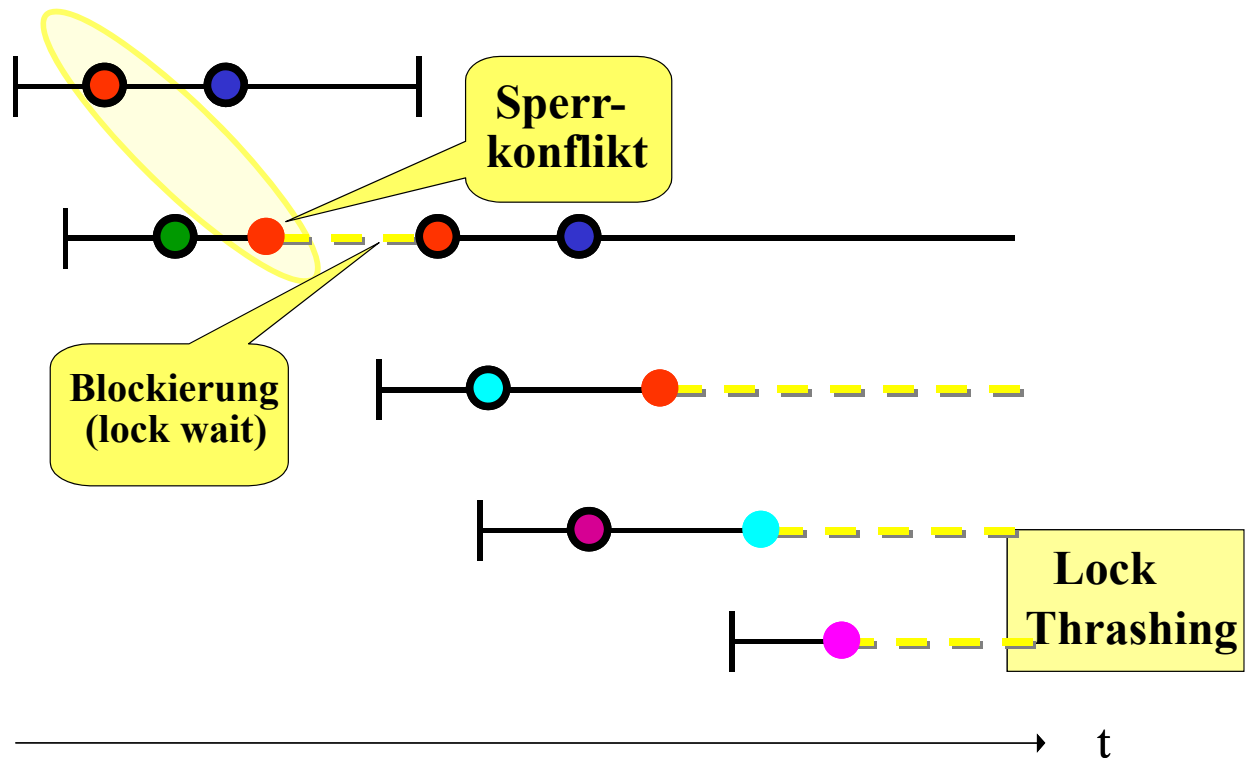


Benchmark-Ergebnisse (Shasha, 2002)

■ table vs. row (record) locking

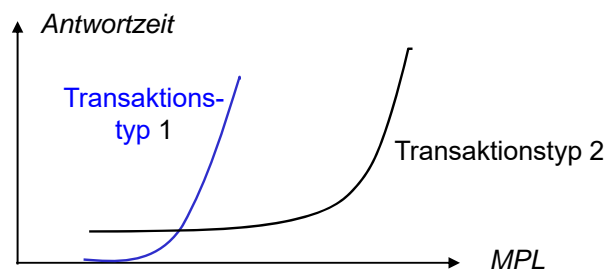
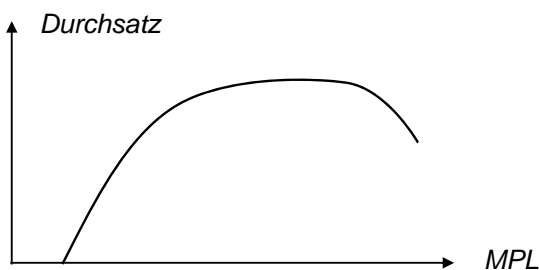


Lock Thrashing



Dynamische MPL-Kontrolle

- Parallelitätsgrad (multiprogramming level, MPL) bestimmt wesentlich Leistungsverhalten
 - insbesondere Umfang an Konflikten bzw. Rücksetzungen
 - Gefahr von "lock thrashing" bei Überschreiten eines kritischen MPL-Wertes



- statische MPL-Einstellung problematisch: wechselnde Lastsituationen, mehrere Transaktionstypen
- Idee: dynamische MPL-Einstellung zur Vermeidung von "lock thrashing"

Dynamische MPL-Kontrolle (2)

- Nutzung einer aktuellen, systemweiten Konfliktrate:

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

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



Zusammenfassung

- Sperrverfahren dominieren in traditionellen (disk-basierten) DBS
 - reine OCC- Verfahren erzeugen i.a. zu viele Rücksetzungen
 - hierarchische Sperrverfahren erlauben Begrenzung des Verwaltungs-Overheads
- hoher Bedeutungszuwachs für Mehrversionen-Synchronisation / Snapshot Isolation, oft in Kombination mit OCC
 - Lesetransaktionen bleiben konfliktfrei
 - stark reduzierte Konfliktgefahr für Änderungstransaktionen
- „harte“ Synchronisationsprobleme durch Hot Spots und lange (Änderungs-) Transaktionen
 - möglichst Vermeidungsstrategie anwenden
 - Spezialprotokolle nutzen semantisches Wissen über Operationen/Objekte zur Reduzierung von Synchronisationskonflikten
 - allerdings: ggf. Erweiterung der Programmierschnittstelle
- dynamische MPL-Kontrolle kann Lock Thrashing vermeiden

