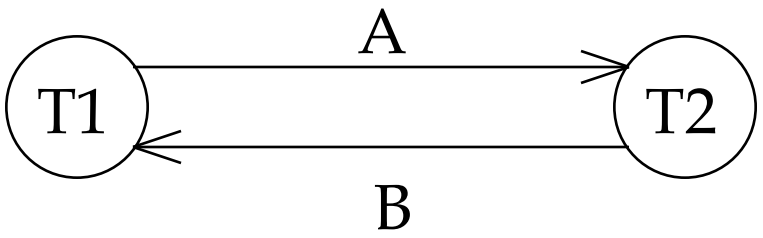
Concurrency Control

Korrektheitskriterium: Serialisierbarkeit

- Zwei Schedules sind konfliktäquivalent wenn gilt:
  - Sie enthalten genau die gleichen Transaktionen (und damit auch Aktionen)
  - Jedes Paar konfligierender Aktionen ist in der gleichen Weise in beiden Schedules geordnet
    - Konfligierende Aktionen sind read-write, write-read oder write-write auf den gleichen Objekten
- Schedule S ist konfliktserialisierbar, wenn S konfliktäquivalent zu irgendeinem seriellen Schedule S ist
- Beispiel: Schedule, der nicht konfliktserialisierbar ist

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	



Präzedenzgraph

Präzedenzgraph

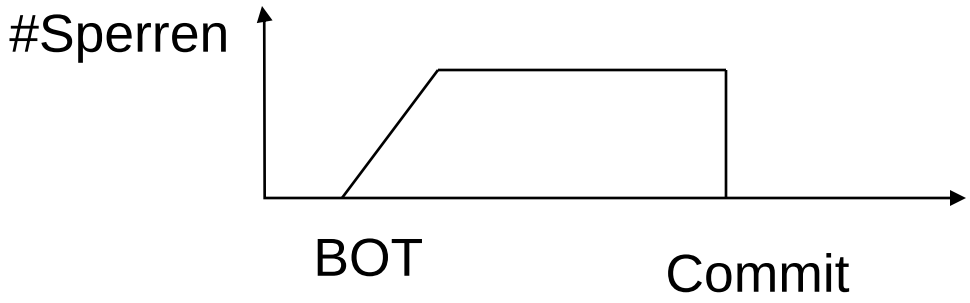
- Beispiel zeigt Zyklus im Präzedenzgraphen.
  - Problem: Output von T1 hängt vom Output von T2 ab und umgekehrt
- Präzedenzgraph:
  - Pro Transaktion ein Knoten
  - Kante von *Ti* nach *Tj*, wenn *Tj* ein Objekt Q liest/schreibt, das zuletzt von *Ti* geschrieben wurde
- Theorem:
  - Ein Schedule ist konfliktserialisierbar, genau dann wenn der zugehörige Präzedenzgraph azyklisch ist
- Komplexität des Algorithmus:
  - Zyklusfindung im Graphen ist $O(n^2)$ mit n=Anzahl Transaktionen

Striktes 2-Phasen-Sperren

Protokoll:

- Jede TA muß eine S-Sperre (*shared lock*) auf dem Objekt vor dem Lesen erwerben, und eine X-Sperre (*exclusive lock*) vor dem Schreiben
- Alle Sperren werden bei Beendigung der Transaktion (d.h. zum Commit-Zeitpunkt) freigegeben
- Wenn eine Transaktion eine X-Sperre auf einem Objekt hält, kann keine andere Transaktion eine (S oder X-) Sperre auf diesem Objekt erwerben

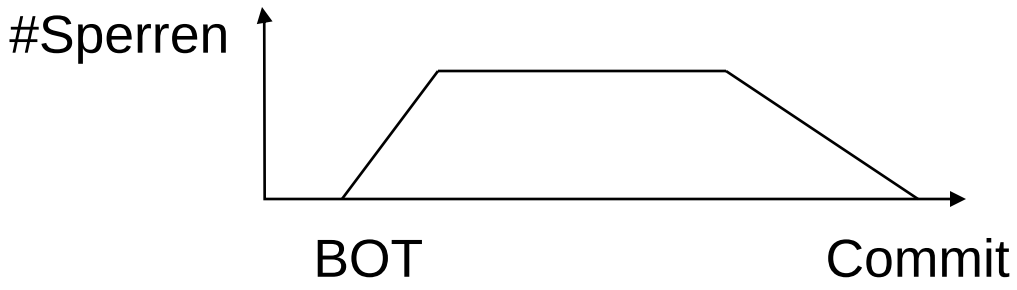
- Nur Striktes 2PL erlaubt Schedules, dessen Präzedenzgraph azyklisch ist
  - Beweis in einschlägiger DB-Literatur



2-Phasen-Sperren (2PL)

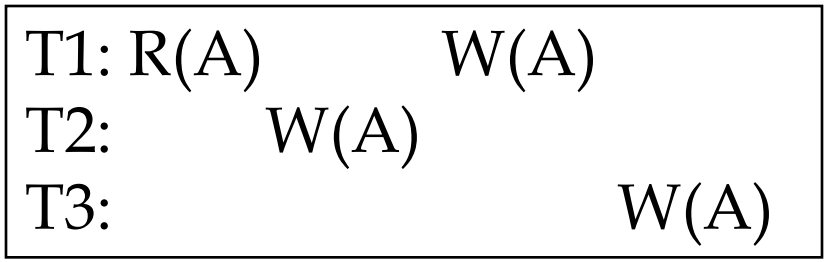
Schwächere Konsistenz als bei striktem 2PL

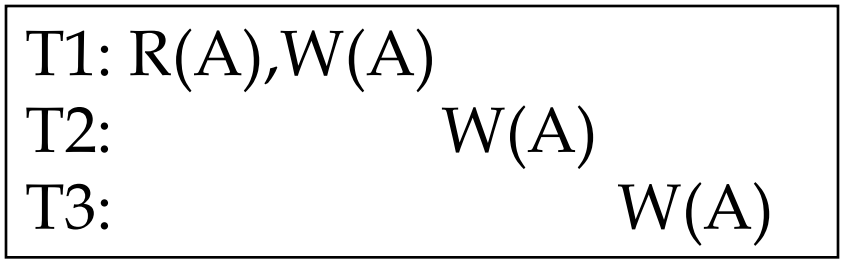
- Jede TA muß eine S-Sperre (*shared lock*) auf dem Objekt vor dem Lesen erwerben, und eine X-Sperre (*exclusive lock*) vor dem Schreiben
- Eine Transaktion kann nicht zusätzliche Sperren anfordern, wenn es erst mal welche freigegeben hat
- Wenn eine Transaktion eine X-Sperre auf einem Objekt hält, kann keine andere Transaktion eine (S oder X-) Sperre auf diesem Objekt erwerben



Sichtserialisierbarkeit

- Zwei Schedules S1 und S sind sichtäquivalent, wenn gilt:
  - Wenn Ti den initalen Wert von A in S1 liest, dann liest Ti auch den initalen Wert von A in S2
  - Wenn Ti den Wert von A liest, der von Tj in S1 geschrieben wurde, dann liest Ti auch den Wert von A, der von Tj in S2 geschrieben wurde
  - Wenn Ti den letzten Wert von A in S1 schreibt, dann schreibt Ti auch den letzten Wert von A in S2





- Ein Schedule ist sichtserialisierbar, wenn er sichtäquivalent zu irgendeinem seriellen Schedule ist
- Nicht jeder sichtserialisierbare Schedule ist auch konfliktserialierbar (siehe Beispiel)!
- Überprüfung auf Sichtserialisierbarkeit ist NP-vollständig
  - Überprüfung des strengeren Kriteriums Konfliktserialisierbarkeit einfacher

Sperrenverwaltung

- Sperrenverwaltungskomponente im DBMS (*Lock Manager*)
- Eintrag in einer Sperrtabelle:
  - Anzahl der Transaktionen, die augenblicklich eine Sperre besitzen
  - Typ der Sperre (Lese- oder Schreibsperre)
  - Pointer auf eine Warteschlange von Sperranforderungen
- Sperren und Entsperren (*Lock / Unlock*) müssen atomare Operationen sein
- Sperren können erweitert werden (*Lock Upgrade*)
  - Eine Transaktion, die eine Lese-Sperre hält, kann diese in eine Schreib-Sperre umgewandelt bekommen

Deadlock

- Voraussetzungen für die Entstehung eines Deadlocks:
  1. Paralleler Objektzugriff durch mehrere Transaktionen
  2. Exklusive Zugriffsanforderungen
  3. Die eine Sperre anfordernde Transaktion besitzt bereits Sperren
  4. Keine vorzeitige Freigabe von Sperren (non-preemption)
  5. Zyklische Wartebeziehung zwischen 2 oder mehr Transaktionen
- Hauptcharakteristikum:

  Zyklische Wartebeziehung zwischen Transaktionen, die wechselseitig auf Freigabe von Sperren warten
- 2 Wege, um Deadlocks zu behandeln:
  - Deadlock-Vermeidung
  - Deadlock-Erkennung

Vermeidung von Deadlocks

- Statische Sperrverfahren (*Preclaiming*)

  Transaktion fordert alle benötigten Sperren gleich zu Beginn an

- Zuordnung von Prioritäten basierend auf statischen Zeitstempeln (*Timestamps*)

  Annahme: Ti benötigt eine Sperre, die Tj besitzt

  2 mögliche Ansätze:

  - *Wait/Die*: Wenn Ti höhere Priorität hat, wartet Ti auf Tj; anderenfalls wird Ti zurückgesetzt (abort)
  - *Wound/Wait*: Wenn Ti höhere Priorität hat, wird Tj zurückgesetzt; anderenfalls wartet Ti
  - Beide Verfahren könnten Rücksetzungen bei Schreibkonflikten auslösen, ohne daß Deadlock vorliegt
  - Beim Re-Start der Transaktion muß diese wieder den originalen Zeitstempel haben

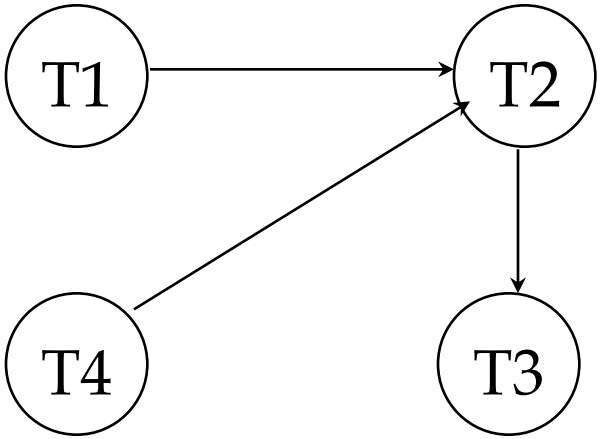
Erkennen von Deadlocks

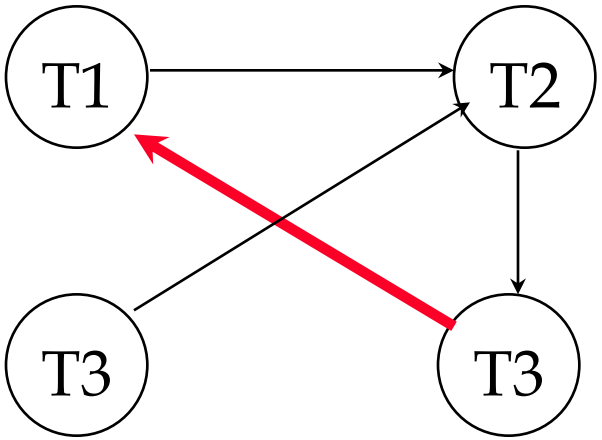
- Erzeugen eines Wartegraphen:
  - Knoten sind Transaktionen
  - Kanten von Ti nach Tj, wenn Ti auf Tj wartet, um eine Sperre freizugeben
- Überprüfung des Wartegraphen notwendig
  - Suche nach Zyklen
    - periodisch: Durchsuchen des ganzen Graphen, aber seltenere Ausführung
    - bei auftretenden Konflikten: Untersuchung eines Teilgraphen
- Auswahl der Opfer-Transaktionen nach verschiedenen Kriterien
  - Minimierung des Arbeitsverlustes
  - Einfachheit der Opferbestimmung
- Weiteres Verfahren: Timeout
  - Zurücksetzen einer Transaktion, sobald ihre Wartezeit auf eine Sperre eine festgelegte Zeitschranke überschreitet

Erkennen von Deadlocks (Forts.)

Beispiel:

T1: S(A), R(A), S(B)
T2: X(B),W(B) X(C)
T3: S(C), R(C) X(A)
T4: X(B)

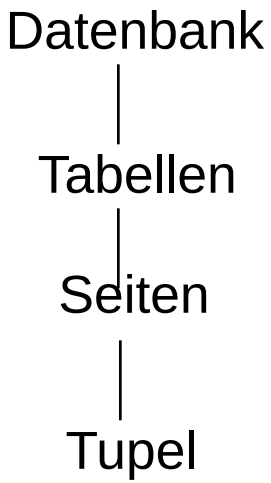




S - shared (Lese-Sperre)
R - read (Lesen)
X - exclusive (Schreib-Sperre)
W- write (Schreiben)

Hierarchisches Sperren

- Sperren können auf Betriebsmitteln der DB hierarchisch angebracht werden (*Multiple Granularity Locks*)
- Problem einfacher Satz-Sperren:
  - Hoher Verwaltungsaufwand von Satzsperren bei Transaktionen auf vielen Tupeln einer Relation (z.B. große Sperrtabelle)
  - Nicht alle Fehler zu vermeiden (z.B. Phantom-Problem)
- Problem von Datenbank-Sperren:
  - Geringere Nebenläufigkeit, wenn z.B. eine Transaktion bloß auf wenige Tupel zugreift
- Entscheidung über Sperrgranulat schwierig (Tupel vs. Seiten vs. Tabellen)
- Hierachie der Betríebsmittel



Lösung: Neue Sperrmodi

- Erlaubt Transaktionen, Sperren auf jeder Stufe zu setzen, mit einem speziellen Protokoll
- Einsatz von Anwartschaftssperren (*Intention Locks*): zeigen die Absicht an, einen Zugriff auf einer tieferen Ebene vorzunehmen und verhindern unverträgliche Sperren auf den höheren Ebenen
- Vor dem Sperren eines Objekts muß eine TA Intention Locks auf allen Vorgängerknoten innerhalb der Sperrhierachie setzen
- Freigabe der Sperren in umgekehrter Richtung (bottom up)
- IS: Absicht, niedrigere Objekte zu lesen. Nachfolger können mit S oder IS-Sperren belegt werden
- IX: Absicht, niedrigere Objekte zu schreiben. Nachfolger können mit X,S,IX oder SIX-Sperren belegt werden
- SIX: Erlaubt Lesezugriff auf Knoten und Absicht, Nachfolger zu ändern. Nachfolger können mit X,SIX oder IX belegt werden

	--	IS	IX	S	X
--	√	√	√	√	√
IS	√	√	√	√	
IX	√	√	√		
S	√	√		√	
X	√				

Protokoll

- Jede Transaktion startet an der Wurzel der Hierarchie
- Um eine S- oder IS-Sperre auf einem Knoten zu erhalten, muß TA eine IS- oder IX-Sperre auf dem Vorgängerknoten besitzen
- Um eine X- oder IX-Sperre auf einem Knoten zu erhalten, muß TA eine IX- oder SIX-Sperre auf dem Vorgängerknoten besitzen
- Freigabe der Sperren in umgekehrter Richtung von unten nach oben

Protokoll ist korrekt, weil es äquivalent zu einem direkten Setzen der Sperren auf den Blattknoten in der Sperrhierarchie ist.

Beispiel

- T1 scannt die Relation R und ändert einige Tupel:
  - T1 erhält eine SIX-Sperre auf R, erhält dann mehrfach eine S-Sperre auf Tupeln von R, die für einige Tupel (zum Ändern) in X-Sperren umgewandelt werden
- T2 verwendet einen Index, um einen Teil von R zu lesen:
  - T2 erhält eine IS-Sperre auf R, und mehrfach eine S-Sperre auf Tupeln von R
- T3 liest die gesamte Relation R:
  - T3 erhält S-Sperre auf R
  - Oder: T3 kann sich wie T2 verhalten und bei Konflikten eskalieren (d.h. Sperren auf nächsthöherer Ebene anfordern)

	--	IS	IX	S	X	SIX
--	√	√	√	√	√	
IS	√	√	√	√		
IX	√	√	√			
S	√	√		√		
X	√					
SIX		√				

Logische vs. Physische Sperren

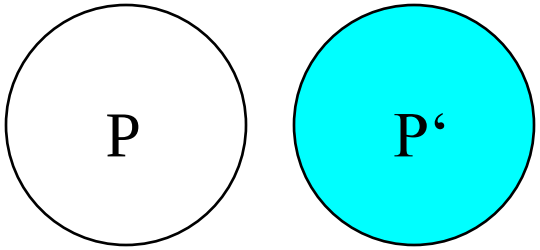
- Nachteil des strikten 2PL
  - Basiert auf der Annahme, daß DB eine feste Ansammlung von Objekten ist
  - Nur tatsächlich vorhandene Objekte können gesperrt werden, aber nicht diejenigen, die nachträglich eingefügt werden (Phantom-Problem!)
- Lösung: Verwendung von logischen Sperren (Prädikatsperren)
  - Sperr-Operation: *Lock (R,P,a)*

    R=Relation, P=Prädikat, a= { Read, Write }
  - 2 Sperr-Anforderungen *Lock (R, P, a)* und *Lock (R‘, P‘, a)* stehen genau in Konflikt, wenn gilt:
    - R = R‘ und
    - (a $\neq$ Read oder a‘$\neq$ Read) und
    - P(t) $\wedge$P(t‘) = TRUE für irgendein t aus R (Prädikate disjunkt)
- Ergebnis: Phantom-Problem umgangen, da nach einer Lesesperre für P kein Einfügen von Objekten möglich, die P erfüllen

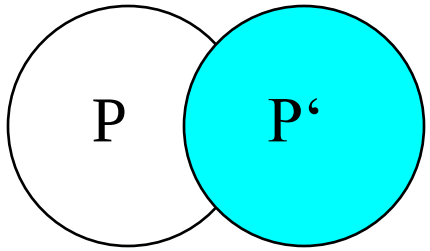
Prädikatsperren

LOCK (PERS,
P: Alter < 50,
Read)

LOCK (PERS,
P‘: PNR =
4711,
Write)



Disjunkte Objektmengen (Konfliktfreiheit)



Überlappende Objektmengen

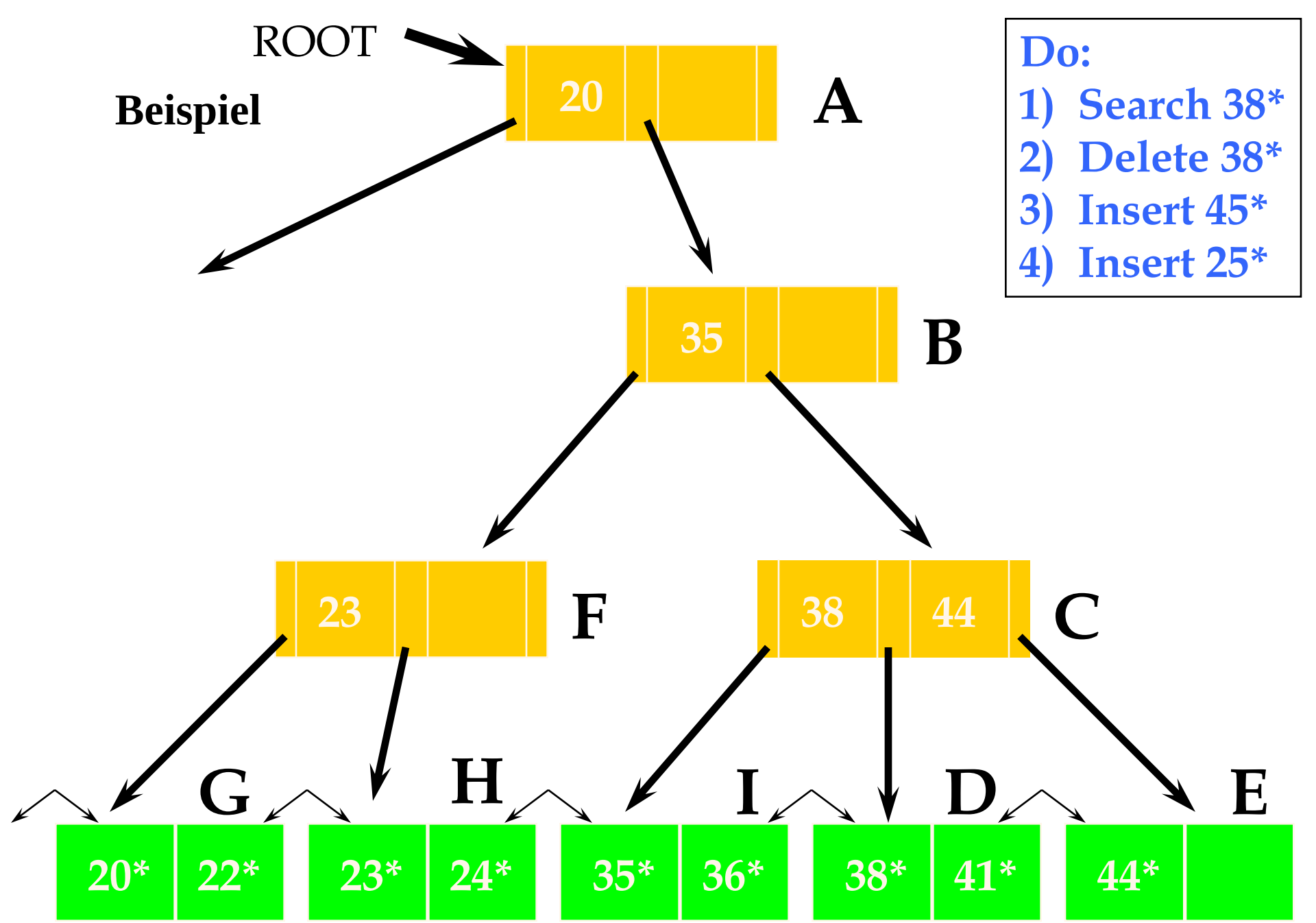
- Prädikatsperren hat hohen Overhead beim Sperren
- Bei 2 beliebigen Prädikaten unentscheidbar, ob Konflikt vorliegt
- Somit restriktivere Einschränkungen notwendig
  → Höhere Konfliktwahrscheinlichkeit
- Alternative zum Prädikatsperren: Sperren des Index denkbar

Sperren in B+ Bäumen

- Wie kann ein bestimmter Blattknoten effizient gesperrt werden?
  - Achtung: Anderes Problem als Hierarchisches Sperren?
- Lösungsansatz:
  - Ignoriere die Baumstruktur
  - Einfache Seitensperren mit 2PL Protokoll
- Problem: Performance
  - Wurzelknoten (und viele Vorgängerknoten) werden zu "Bottlenecks", weil jeder Baumzugriff an der Wurzel beginn
- Erfahrungen:
  - Zwischenknoten im Baum steuern nur die Suche nach Blattknoten
  - Bei Inserts muß ein Pfad von der Wurzel bis zum modifizierten Blatt nur dann gesperrt werden (im X-Modus), wenn ein Split mit Reorganisation des Baumes bis zur Wurzel notwendig ist (analog gültig für Delete)
- Entwurf eines effizienten Sperrprotokolls, das Serialisierbarkeit garantiert (sogar bei Verletzung des 2PL), möglich

Einfacher Baumsperr-Algorithmus

- Search: Beginne bei der Wurzel und steige ab; bei jedem Schritt: setze S-Sperre auf den Sohn und entsperre den Vater
- Insert / Delete: Beginne bei der Wurzel und steige ab; erwerbe X-Sperren soweit benötigt.
  Nach dem Sperren eines Sohn-Knotens, prüfe, ob dieser sicher ist:
  - Wenn ein Sohn-Knoten sicher ist, gebe Sperren auf allen Vorgängern frei
- Sicherer Knoten: Knoten, bei dem Änderungen nicht über den Knoten hinaus propagiert werden.
  - Insert: Knoten ist nicht voll
  - Deletes: Knoten ist nicht halb-leer.


ROOT
Beispiel
20
A
35
B
23
F
38
44
C
G
H
I
D
E
20*
22*
23*
24*
35*
36*
38*
41*
44*
Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*

Beispiel (Forts.)

- Search 38*
  - Erwerbe S-Sperre auf A
  - Erwerbe S-Sperre auf B
  - Freigabe S-Sperre auf A
  - Erwerbe S-Sperre auf C
  - Freigabe S-Sperre auf B
  - Erwerbe S-Sperre auf D
  - Freigabe S-Sperre auf C
- Insert 45*
  - Erwerbe S-Sperre auf A
  - Erwerbe S-Sperre auf B
  - Freigabe S-Sperre auf A
  - Erwerbe S-Sperre auf C (keine Freigabe der S-Sperre auf B!)
  - Erwerbe X-Sperre auf E
  - Freigabe S- Sperre auf C
  - Freigabe S-Sperre auf B
- Insert 25*
  - Erwerbe S-Sperre auf A
  - Erwerbe S-Sperre auf B
  - Freigabe S-Sperre auf A
  - Erwerbe S-Sperre auf F
  - Freigabe S-Sperre auf B
  - Erwerbe X-Sperre auf H
  - H ist voll, erfordert Splitten von H
  - Verlange Umwandlung von S- in X-Sperre auf F
  - Warten, falls Sperrkonflikt in F auftritt

Verbesserter Baumsperr-Algorithmus*

Nach Bayer/Schkolnick:

- Search: Wie gehabt
- Insert / Delete:
  - Setze Sperren wie beim Search bis zum Blattniveau
  - Setze X-Sperre auf dem Blattknoten
  - Wenn Blatt-Knoten nicht sicher ist, gebe alle Sperren frei und starte die Transaktion erneut mit dem herkömmlichen Insert/Delete-Protokoll (siehe vorigen Algorithmus)
- Setzt voraus, daß nur Blattknoten modifiziert werden, anderenfalls wäre die Vergabe von S-Sperren in einem ersten Paß überflüssiger Mehraufwand
- In der Praxis besser als vorheriger Algorithmus

- Weitere Optimierungsmöglichkeiten für Baumsperr-Algorithmen:
  - Verwendung von IX- anstelle von X-Sperren
  - Verwendung von X-Sperren nur auf unteren Ebenen nahe Blatt-Niveau

Optimistische Verfahren

- Sperren ist ein konservativer Ansatz, in dem Konflikte vermieden werden mit entsprechenden Nachteilen:
  - Hoher Aufwand für Sperrenverwaltung
  - Erkennung und Auflösung von Deadlocks notwendig
  - Unnötiger Aufwand, wenn nur wenig Konkurrenz um Resourcen besteht
- Wenn Konflikte selten sind, kann Nebenläufigkeit erhöht werden durch Verzicht auf Sperren:
  - Alternative: Prüfen auf Konflikte beim Commit von Transaktionen

Modell (nach Kung/Robinson)

- Lesephase (READ) R:
  - Eigentliche Transaktionsverarbeitung; d.h. Transaktion liest Objekte aus der Datenbank und modifiziert diese
  - Änderungen werden auf privaten Kopien in einem Transaktions-puffer durchgeführt
- Validierungsphase (VALIDATE) V:
  - Beginnt beim Commit der Transaktion
  - Prüfen, ob die beendigungswillige Transaktion mit einer parallel zu ihr laufenden Transaktion in Konflikt geraten ist
  - Auflösen von Konflikten durch Zurücksetzen einer oder mehrerer beteiligter Transaktionen
- Schreibphase (WRITE) W:
  - Wird nur von Änderungstransaktionen ausgeführt, welche die Validierungsphase erfolgreich beenden konnten
  - Sicherstellung der Wiederholbarkeit der Transaktion (Logging)
  - Alle Änderungen werden durch Einbringen der lokalen Kopien in die Datenbank sichtbar gemacht

Validierung

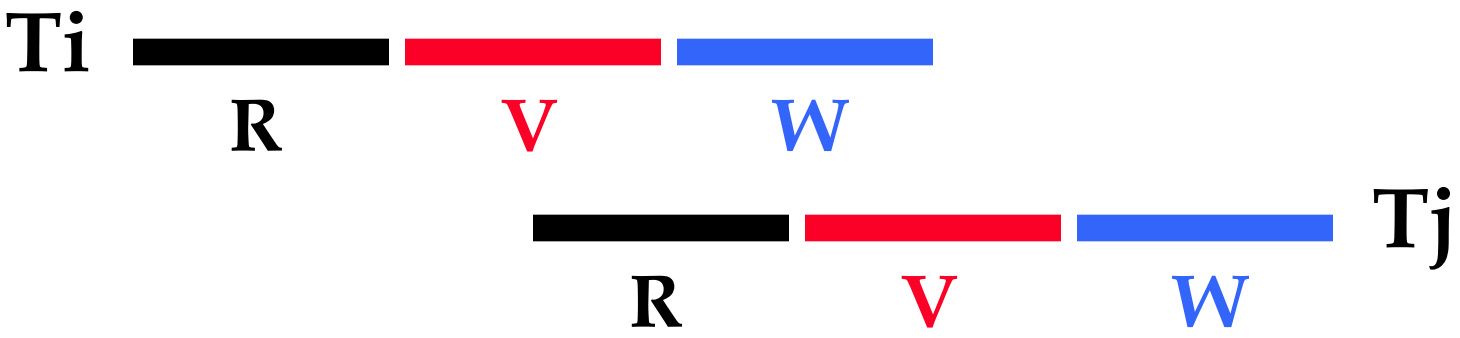
- Überprüfen von Bedingungen gewährleistet, daß kein Konflikt aufgetreten ist
- Jeder Transaktion wird eine numerische ID zugewiesen
  - Im allgemeinen Zeitstempel (Timestamp) TS(Ti)
- Timestamps werden am Ende der Lesephase zugewiesen, unmittelbar vor dem Beginn der Validierung
- ReadSet(Ti)
  Menge der Objekte, die von der Transaktion Ti gelesen wird
- WriteSet(Ti)
  Menge der Objekte, die von der Transaktion Ti geschrieben wird

Test 1

- Für jedes Paar von Transaktionen Ti und Tj mit TS(Ti) < TS(Tj) prüfe, daß Ti beendet wird, bevor Tj beginnt

Test 2

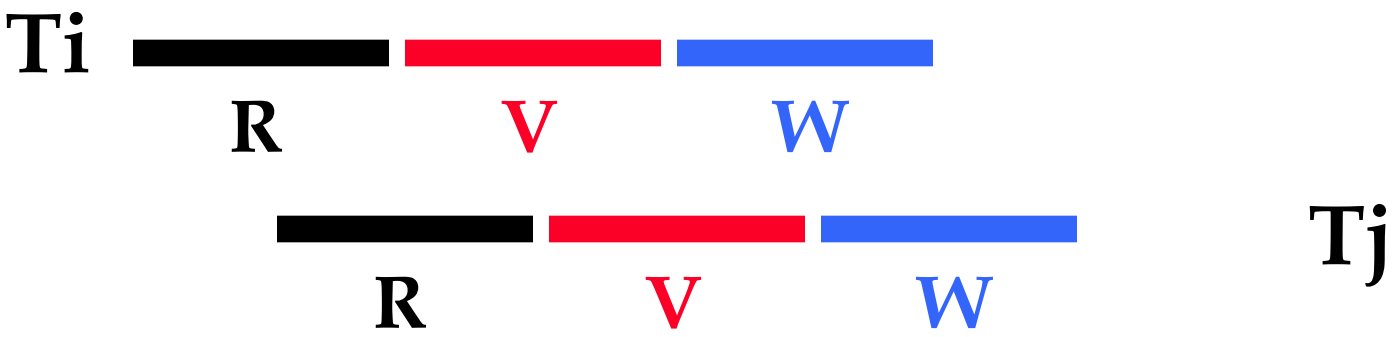
- Für jedes Paar von Transaktionen Ti und Tj mit TS(Ti) < TS(Tj) prüfe:
  - Ti ist beendet, bevor Tj mit der Schreibphase beginnt und
  - WriteSet(Ti) ∩ ReadSet(Tj) ist leer



- Erlaubt Tj, Objekte zu lesen während Ti noch modifiziert, aber kein Konflikt
- Tj könnte Objekte überschreiben, die von Ti geschrieben wurden, aber alle Writes von Ti liegen vor denen von Tj

Test 3

- Für jedes Paar von Transaktionen Ti und Tj mit TS(Ti) < TS(Tj) prüfe:
  - Ti beendet ihre Lesephase vor der Lesephase von Tj und
  - WriteSet(Ti) ∩ ReadSet(Tj) ist leer und
  - WriteSet(Ti) ∩ WriteSet(Tj) ist leer



- Stärkere zeitliche Überlappung als im Test 2 erlaubt
- Keine Überlappung der Objektmengen, somit auch keine RW-, WR- oder WW-Konflikte

Validierung einer Transaktion T

- BOCC-Ansatz (Backward Oriented Concurrency Control): Validierung nur gegenüber bereits beendeten Transaktionen

```
valid = true;
// S = Menge der Transaktionen, die nach Begin(T)
     beendet wurden
< foreach  Ts in S do {
  if ReadSet(T) ∩ WriteSet(Ts) = Ø
        then valid = false;
  }
  if valid then { install updates; // Schreibphase T
                  Commit T } >
            else Rollback T
```

Ende des kritischen Abschnitts

Bemerkungen zur Validierung

- Zuweisung von Transaktions-ID, Validierungs- und Schreibphase sind innerhalb eines kritischen Abschnitts
  - Keine weitere Transaktion kann parallel beendet werden
  - Großer Nachteil, wenn Schreibphase lang ist
- Gefahr des “Verhungerns“
  - Ständiges Scheitern von Transaktionen bei der Validierung: lange Transaktionen mit großen Read-Sets
  - Spätes Zurücksetzen
- Optimierung für Read-Only-Transaktionen:
  - Kein kritischer Abschnitt (weil keine Schreibphase)
- Bedingungsprüfung oft zu streng
  - Unnötiges Zurücksetzen von Transaktionen, die die aktuelle Objektversion gelesen haben
- Variante: vorwärtsorientiertes Verfahren (Forward Oriented Concurrency Control (FOCC)
  - Validierung gegen noch laufende Transaktionen

Kombination mit Sperrverfahren*

- *Optimistic Locking* Verfahren:
  - Setze S-Sperren wie gewöhnlich
  - Änderungen auf privaten Kopien von DB-Objekten
  - Beim Einbringen der Änderungen Anfordern von X-Sperren für die betroffenen Objekte
  - Änderung global sichtbar machen
  - Freigabe aller Sperren
- Ergebnis: Transaktionen werden blockiert, wenn auf Sperren gewartet wird
  - Keine Validierungsphase
  - Kein Zurücksetzen von Transaktionen
- Mögliche Anwendung in Client/Server-Systemen

Mehraufwand bei optimistischen Verfahren

- Aufzeichnen der Lese- und Schreibaktivitäten in Read-Sets und Write-Sets pro Transaktion
  - Erzeugen und Löschen dieser Mengen bei Bedarf
- Prüfung auf Konflikte während der Validierung und globales Sichtbarmachen validierter Writes
  - Kritischer Abschnitt reduziert Parallelität
  - Verfahren zum Zurückschreiben von Änderungen kann Clustering reduzieren
- Optimistische CC setzt die Transaktionen zurück, deren Validierung scheitert
  - Bisherige Arbeit gescheitert
  - “Aufräumarbeiten“ erforderlich
  - Nicht geeignet auf Indexstrukturen

Zeitstempelverfahren

- Idee: Jedes Objekt erhält einen Lesezeitstempel (*Read-Timestamp*) RTS und einen Schreibzeitstempel (*Write-Timestamp*) WTS. Jede Transaktion erhält zu Beginn einen Zeitstempel TS.
  - Konfliktoperationen verschiedener Transaktionen müssen stets in der Reihenfolge der Transaktionszeitstempel erfolgen
    - Wenn eine Aktion ai einer Transaktion Ti einen Konflikt mit einer Aktion aj einer Transaktion Tj hat und gilt TS(Ti) < TS(Tj), dann muß ai vor aj ablaufen
    - Anderenfalls Zurücksetzen der verletzenden Transaktion
  - Eine Transaktion muß alle Änderungen von älteren Transaktionen (d.h. Transaktionen mit kleinerem Zeitstempel) sehen
  - Eine Transaktion darf keine Änderungen von jüngeren Transaktionen sehen

Lesen eines Objekts O durch Transaktion T

- Wenn TS(T) < WTS(O), verletzt dies die Zeitstempelordnung von T bezüglich der Schreiber auf O. Keine jüngere Transaktion als T darf zuletzt das Objekt O geändert haben.
  - Erfordert Abbruch von T und Neustart mit einem neuen größeren Timestamp (Beim Restart mit dem gleichen Timestamp würde T wieder scheitern! Beachte den unterschiedlichen Gebrauch von Zeitstempeln bei der Deadlock-Vermeidung im 2PL)
- Wenn TS(T) > WTS(O):
  - Erlaubt T, das Objekt O zu lesen
  - Setze RTS(O) auf das Maximum von (RTS(O), TS(T))
- Modifikation von RTS(O) beim Lesen muß auf Platte geschrieben werden! Somit entsteht Overhead, ebenso auch beim Restart von Transaktionen.

Schreiben eines Objekts O durch Transaktion T

- Wenn TS(T) < RTS(O), verletzt dies die Zeitstempelordnung von T bezüglich der Leser von O. Keine jüngere Transaktion als T darf zuletzt das Objekt O geändert haben. Abbruch und Restart von T erforderlich.
- Wenn TS(T) < WTS(O), verletzt dies die Zeitstempelordnung von T bezüglich der Schreiber von O, d.h. aktuelles Write von T wird durch das jüngste Write überschrieben
  - Thomas Write-Regel: Wir können solche überholten Writes ignorieren; brauchen keinen Restart von T! (T's Write erfolgt effektiv vor einem anderen Write, ohne Read-Operationen dazwischen) Erlaubt einige serialisierbare, aber keine konfliktserialisierbaren Schedules:
- Anderenfalls
  erlaube T, O zu schreiben
  Setze WTS(O) auf TS(T)
- Im Schedule gilt: Keine Transaktion sieht die Write-Operation von T2

T1	T2
R(A)	
	W(A)
	Commit
W(A)	
Commit	

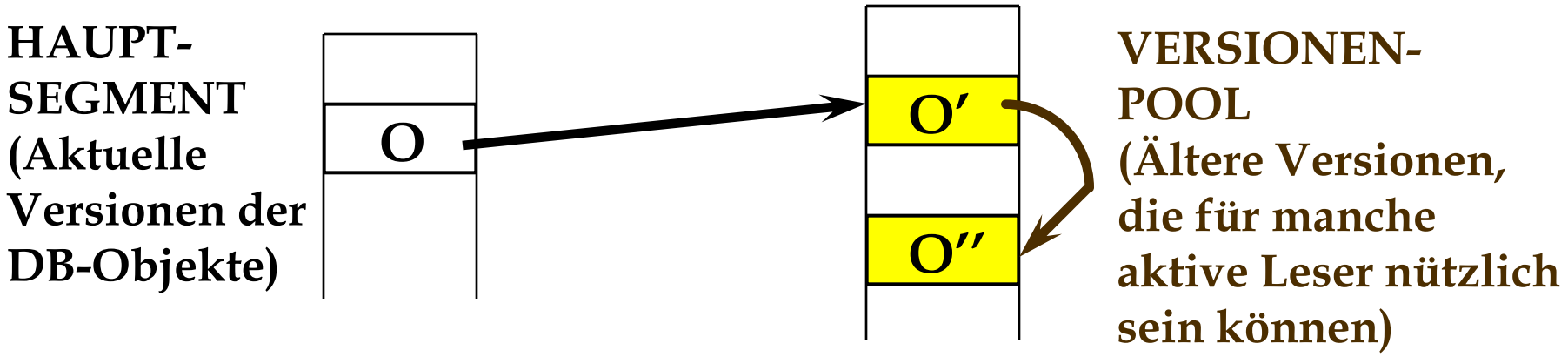
Bewertung des Zeitstempelverfahrens

T1	T2
W(A)	
	R(A)
	W(B)
	Commit

- Im Schedule liest T2 “schmutzige“ Daten (A), d.h. noch vor dem commit von T1
- Schedule nicht recovery-fähig, wenn T1 zurückgesetzt werden muß
- Zeitstempelverfahren modifizieren, um recoverable Schedules zu erlauben:
  - Puffern aller Writes, bis der Schreiber commit macht (aber Ändern aller WTS(O), wenn das Write gestattet ist)
  - Blockiere Leser T (wenn TS(T) > WTS(O) bis zum Commit des Schreibers von O
- Nachteile:
  - Rücksetzgefahr einer TA steigt mit zunehmender Verweildauer im System
  - Viele Blockierungen möglich, um Recovery-Fähigkeit zu sichern (siehe oben)
- Reine Zeitstempelverfahren praktisch wenig angewandt da zu viele Probleme (Hauptanwendung aber in verteilten Datenbanken)

Mehrversionen-Synchronisation

- Idee: Schreiber machen eine “neue” Kopie; Leser verwenden eine geeignete “alte” Kopie:

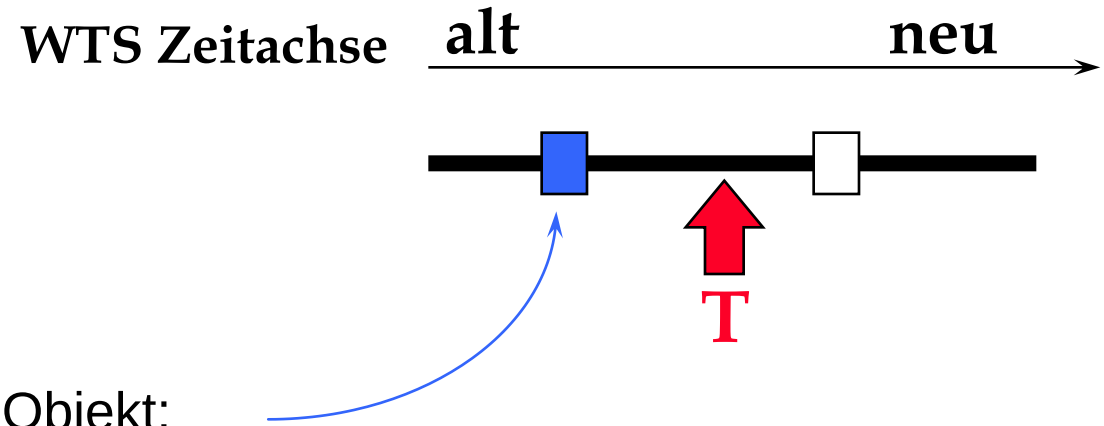


- Erhebliche Reduzierung von Synchronisationskonflikten
- Jede Transaktion klassifiziert als Leser oder Schreiber:
  - Schreiber können Objekte schreiben, Leser niemals
  - Transaktion werden bei Beginn als Leser oder Schreiber deklariert

Mehrversionen-Synchronisation (Forts.)

- Jede Version eines Objekts hat:
  - Writer-Timestamp WTS der schreibenden Transaktion und
  - Reader-Timestamp RTS der Transaktion, die zuletzt diese Version gelesen hat
- Versionen sind rückwärtsverkettet (Overhead für Versionenverwaltung)
- Zu alte Versionen sind nicht mehr interessant und können entfernt werden (benötigt *Garbage Collection*-Algorithmen)
- Lesetransaktionen sehen immer einen konsistenten DB-Zustand (den, der zu Beginn der TA gültig war)
  - Keine Synchronisation mit Lesetransaktionen notwendig
  - Kein Restart von Lesetransaktionen
- Änderungstransaktionen
  - Zugriff auf die aktuellste Version
  - Synchronisation mit beliebigem Verfahren
- Attraktiv zur Unterstützung langer Anfragen (z.B. Decision Support), welche ohne Synchronisationskonflikte bearbeitet werden können
- Genereller Nutzen sehr hoch, da Lesetransaktionen in vielen DB-Anwendungen dominieren

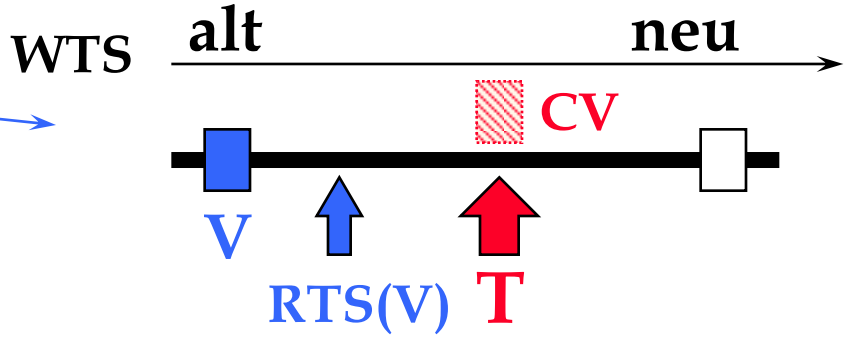
Lese-Transaktionen*



- Für jedes zu lesende Objekt:
  - Finde neueste Version mit WTS < TS(T). (Beginne mit der aktuellen Version im Hauptsegment und verfolge die Kette rückwärts zu früheren Versionen)
- Annahme, daß irgendeine Version jedes Objekts von Anfang an existiert, Lese-Transaktionen werden niemals neu gestartet.
  - Können jedoch blockiert werden bis die Schreib-Transaktion der geeigneten Version commit macht

Schreib-Transaktionen*

- Zum Lesen eines Objekts befolge das Leser-Protokoll
- Zum Schreiben eines Objekts:
  - Finde neueste Version V so daß WTS < TS(T).
  - Wenn RTS(V) < TS(T): T macht eine Kopie CV von V, mit einem Pointer auf V, mit WTS(CV) = TS(T), RTS(CV) = TS(T). (Write wird gepuffert bis zum Commit von T; andere Transaktionen können TS-Werte, aber nicht die Version CV sehen)
  - Anderenfalls: Schreiben wird zurückgewiesen



Transaktions-Unterstützung in SQL-92

- Jede Transaktion hat:
  - Access Mode: READ ONLY, READ WRITE
  - Diagnostics Size: Anzahl von aufzuzeichnenden Fehlern
  - Isolation Level

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Zusammenfassung

- Sperrenbasierte Synchronisationsverfahren (2PL, Striktes 2PL). Konflikte zwischen Transaktionen durch Abhängigkeitsgraphen aufgezeigt
- Sperrenmanager kontrolliert die Sperrenvergabe. Deadlocks können verhindert oder erkannt werden.
- Einfache Sperrmechanismen können Phantom-Problem bewirken
- Index-Sperren sehr gebräuchlich, beeinflußt die Performance signifikant
  - Benötigt, wenn Zugriff auf Sätze über einen Index
  - Benötigt zum Sperren einer logischen Satzmenge (Prädikatbasiertes Sperren)
- Baumsperr-Algorithmen:
  - Einfache Anwendung des 2PL sehr ineffizient
  - Verbesserungsmöglichkeiten (Bayer-Schkolnick-Algorithmus)

Zusammenfassung (Forts.)

- Bessere Techniken verfügbar (Satz- statt Seitensperren)
- Multiple Granularity Locks vermindern den Mehraufwand bei der Sperrenvergabe in Ressourcen-Hierarchien (z.B. Datei - Seiten)
- Optimistische Verfahren
  - minimieren den Mehraufwand der Sperrenverwaltung in Umgebungen mit vielen Reads und wenig Writes
  - haben ihren eigenen Overhead (deshalb nicht so verbreitet)
- Zeitstempelverfahren
  - sind Alternative zum 2PL
  - erlauben serialisierbare Schedules, die im 2PL nicht möglich
  - Erfordern zusätzliche Fähigkeiten, um Transaktionen zu blockieren, um Recovery-Fähigkeit von Schedules sicherzustellen
- Mehrversionenverfahren
  - Lese-Transaktionen immer ohne Verzögerung möglich
  - Vermeidet Restart von Lese-Transaktionen
  - Versionenverwaltung nötig
- Definition der gewünschten Konsistenz bei Nebenläufigkeit in SQL-92 durch *Isolation Levels*