Teil II – Concurrency Control

Kapitel 2: Serialisierbarkeitstheorie – Korrektheitskriterien
Kapitel 3: Konservative Concurrency Control-Protokolle
Kapitel 4: Optimistische Concurrency Control-Protokolle
Kapitel 5: Mehrversionen-Concurrency Control
Kapitel 6: Mehrversionen-Protokolle

Kapitel 2 – Serialisierbarkeitstheorie

2.1 Grundlagen der Serialisierbarkeitstheorie

Die Serialisierbarkeitstheorie befasst sich mit den Korrektheitskriterien für die korrekte parallele Ausführung von Transaktionen

Gegeben: $\tau = \{ T^1, T^2, \ldots , T^n \}$ Menge von Transaktionen, die korrekt parallel ausgeführt werden sollen.

Grundannahme G1: Jede Transaktion T^i für sich genommen (einzeln, in Isolation ausgeführt) ist korrekt.

Grundannahme G2: Jede serielle Ausführung aller n Transaktionen (in irgendeiner Reihenfolge) ist korrekt.

Wir benötigen also Kriterien die überprüfen, ob eine gegebene parallele Ausführung mehrerer Transaktionen mit einer solchen seriellen Ausführung übereinstimmt!

Schedule ...

Die parallele Ausführung der Transaktionen in τ wird durch einen Schedule reflektiert:

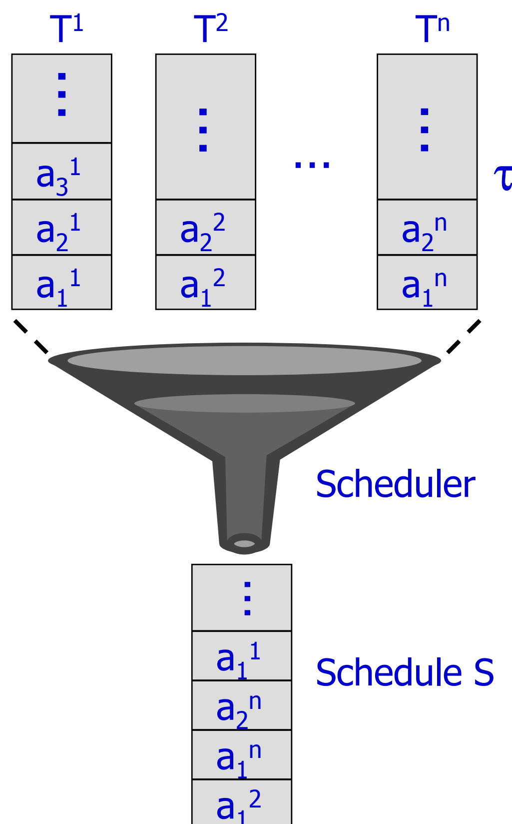
Vollständiger Schedule S Ein vollständiger Schedule S ist ein Tripel $S = (\tau, \mathbb{A}, <)$. S enthält die Ausführungsfolge aller Aktionen sämtlicher Transaktionen aus τ unter Wahrung der Präzedenzrelationen.

Dabei ist:

- $\tau = \{ T^1, T^2, \ldots, T^n \}$ Menge von Transaktionen
- $\mathbb{A} = \cup\ ACT^i$ Vereinigung der Aktionen aller Transaktionen in τ (*)
- $< \subseteq (\mathbb{A} \times \mathbb{A})$ partielle Ordnung, wobei gilt: $\ll^i \subseteq <$ für alle $T^i \in \tau$, d.h. die Ausführungsfolge der Aktionen in S beachtet die vorgeschriebene Folge innerhalb jeder Transaktion.

(*) Ein vollständiger Schedule enthält genau ein C oder ein A für jede Transaktion

... Schedule ...



Schedule: Ein Schedule ist ein Präfix eines vollständigen Schedules

Ein Schedule S enthält

- abgeschlossene (C in S)
- abgebrochene (A in S)
- aktive (weder C noch A in S)

Transaktionen.

In einem vollständigen Schedule existieren keine aktiven Transaktionen.

Für die Ordnung < eines Schedules gilt:

- < total: sequentielle Bearbeitung
- < nicht total: parallele Bearbeitung, z.B. in verteilten Datenbanken

... Schedule

Serieller Schedule: In einem seriellen Schedule ist < total und für beliebige i, k $\in$ {1, .., n} gilt: alle Aktionen von T^i werden vor allen Aktionen von T^k ausgeführt

$$S_{ser} = \underbrace{a_1^2\ a_2^2\ \ldots\ C^2}_{T^2}\ \underbrace{a_1^n\ a_2^n\ \ldots\ C^n}_{T^n}\ \underbrace{a_1^1\ a_2^1\ \ldots\ C^n}_{T^1}$$

Zwei serielle Schedules werden jeweils als korrekt angesehen, liefern i.a. aber verschiedene Datenbank-(Zwischen- oder/und End-) Zustände.

Committed Projection: Die committed projection C(S) eines Schedules S erhält man, wenn man aus S alle Aktionen von Transaktionen streicht, die nicht mit commit beendet sind, d.h. τ einschränkt auf: $\cup\ T^i$: C^i in S

C(S) enthält also weder aktive noch abgebrochene Transaktionen.

Semantik von Aktionen

Die Semantik der Aktionen einer Transaktion in einem Schedule ist wie folgt:

Leser: $r^i(x)$ liest x von $w^k(x)$, wobei $w^k(x)$ die letzte Schreibaktion in S vor $r^i(x)$ ist. Dabei ist $i \neq k$ und T^k ist nicht abgebrochen.

Schreiber: $w^i(x)$ schreibt x neu. Dieses Schreiben hängt von allen Leseaktionen von T^i vor $w^i(x)$ ab. Diese Leser (wie oben) lesen von nicht abgebrochenen Transaktionen.

... Das Ergebnis einer Schreibaktion hängt (in unbekannter Weise; dies ist Sache des Anwendungsprogramms) von allen vorhergehenden Leseaktionen innerhalb der betrachteten Transaktion ab.

2.2 Final-State Serialisierbarkeit

- Final-State-Serialisierbarkeit ist ein allgemeines Korrektheitskriterium
- Zugrundeliegende Idee: ein Schedule S ist korrekt, wenn er zu demselben Datenbankzustand führt wie ein beliebiger serieller Schedule über derselben Transaktionsmenge
- Aber: wie können wir den Datenbankzustand ermitteln?
- Abhilfe: wir betrachten den Abschluss eines Schedules und vermeiden somit Datenbankzustände

Abschluss eines Schedules S: $\hat{S} = \langle T^0, S, T^\omega \rangle$

T^0 (künstliche) Anfangstransaktion, die alle Objekte schreibt, die in S gelesen werden

T^ω (künstliche) Abschlusstransaktion, die alle Objekte liest, die in S geschrieben wurden

Beispiel: $S_1 = \langle w_1^1(a)\ r_1^2(a)\ w_2^2(b)\ C^2\ r_2^1(b)\ C^1 \rangle$

$$\hat{S}_1 = \langle \underbrace{w_1^0(a)\ w_2^0(b)\ C^0}_{T^0}\ \underbrace{w_1^1(a)\ r_1^2(a)\ w_2^2(b)\ C^2\ r_2^1(b)\ C^1}_{\text{ursprüngliches } S_1}\ \underbrace{r_1^\omega(a)\ r_2^\omega(b)\ C^\omega}_{T^\omega} \rangle$$

Liest-von-Relation & nützlich

Liest-von-Beziehung: Sei $\hat{S}$ ein abgeschlossener Schedule.
$r^i(x)$ liest-von $w^k(x)$, $i \neq k$, falls:
- (a) $w^k(x) < r^i(x)$
- (b) $\neg\exists\ w^m(x)$ mit $w^k(x) < w^m(x) < r^i(x)$
- (c) T^k ist nicht abgebrochen

Reads-From-Relation: (Liest-von-Relation) $RF = \{ (T^k, x, T^i) \mid r^i(x) \text{ liest-von } w^k(x) \}$

Mit der gegebenen Semantik der einzelnen Aktionen ist nicht nur das letzte Schreiben vor T^ω für das abschliessende Lesen entscheidend, sondern auch die Einflüsse auf diese Schreiber. Dies wird durch direkt nützlich und nützlich ausgedrückt:

Direkt nützlich: Eine Aktion a ist direkt nützlich für a' ($a \rightarrow a'$), falls gilt:
- entweder a' liest-von a
- oder a ist Lesen und a' nachfolgendes Schreiben derselben Transaktion

Nützlich: $\hookrightarrow$, die transitive Hülle von $\rightarrow$ heisst nützlich

Lebendig & Live-Reads-From-Relation

Da Final-State-Serialisierbarkeit die Gleichheit der Datenbankzustände betrachtet, sollen natürlich nur solche Aktionen berücksichtigt werden, die auch wirklich Auswirkungen auf diesen Zustand haben, das heisst, die Auswirkungen auf Leser aus T^ω haben. Zum Beispiel ist dies für $w_1^1(x)$ im Schedule $\hat{S} = \langle T^0\ w_1^1(x)\ w_1^2(x)\ T^\omega \rangle$ nicht der Fall.

Lebendig: Eine Aktion a heisst lebendig, falls a entweder
- für eine Aktion aus T^ω nützlich ist, also $(\exists a' \in T^\omega)\ a \hookrightarrow a'$
- oder zur Abschlusstransaktion T^ω gehört.

Live-Reads-From-Relation (lebendig-liest-von-Relation):

$$LRF = \{ (T^k, x, T^i) \mid r^i(x) \text{ ist lebendig} \wedge r^i(x) \text{ liest-von } w^k(x) \}$$

Beispiel 1 (Live-Reads-From-Relation)

$\hat{S}_1 = \langle w^0(x)\ w^0(y)\ C^0\ r^1(x)\ r^2(y)\ w^1(y)\ w^2(y)\ C^1\ C^2\ r^\omega(x)\ r^\omega(y)\ C^\omega \rangle$

direkt nützlich:	nützlich:
$w^0(x) \rightarrow r^1(x)$	$w^0(x) \rightarrow w^1(y)$
$w^0(y) \rightarrow r^2(y)$	$w^0(y) \rightarrow w^2(y)$
$w^0(x) \rightarrow r^\omega(x)$	$w^0(y) \rightarrow r^\omega(y)$
$w^2(y) \rightarrow r^\omega(y)$	$r^2(y) \rightarrow r^\omega(y)$
$r^1(x) \rightarrow w^1(y)$	+ direkt nützlich
$r^2(y) \rightarrow w^2(y)$	

lebendig: $w^0(x)$, $w^2(y)$, $w^0(y)$, $r^2(y)$, $r^\omega(x)$, $r^\omega(y)$

tot: $r^1(x)$, $w^1(y)$

$RF = \{ (T^0, x, T^1), (T^0, x, T^\omega), (T^0, y, T^2), (T^2, y, T^\omega) \}$

$LRF = \{ (T^0, x, T^\omega), (T^0, y, T^2), (T^2, y, T^\omega) \}$

Beispiel 2 (Live-Reads-From-Relation)

$\hat{S}_2 = \langle\ w^0(x)\ w^0(y)\ C^0\ r^1(x)\ w^1(y)\ r^2(y)\ w^2(y)\ C^2\ C^1\ r^\omega(x)\ r^\omega(y)\ C^\omega\ \rangle$

direkt nützlich:

$w^0(x) \rightarrow r^1(x)$
$w^1(y) \rightarrow r^2(y)$
$w^0(x) \rightarrow r^\omega(x)$
$w^2(y) \rightarrow r^\omega(y)$
$r^1(x) \rightarrow w^1(y)$
$r^2(y) \rightarrow w^2(y)$

lebendig: $w^0(x)$, $r^1(x)$, $w^1(y)$, $r^2(y)$, $w^2(y)$, $r^\omega(x)$, $r^\omega(y)$
tot: $w^0(y)$

RF = LRF = $\{ (T^0, x, T^1) , (T^0, x, T^\omega) , (T^1, y, T^2) , (T^2, y, T^\omega) \}$

Final-State Serialisierbarkeit: Definition

Idee: Gleichheit der LRF zweier Schedules bedeutet, dass die Ausführungen äquivalent sind.

Final-State-Äquivalenz: Zwei Schedules S und S′ sind FS-äquivalent, wenn sie dieselbe LRF besitzen, also LRF($\hat{S}$) = LRF($\hat{S}'$)

Final-State-Serialisierbarkeit (FSSR)

Ein vollständiger Schedule S ist Final-State-serialisierbar, wenn es einen seriellen Schedule S′ gibt, der zu S FS-äquivalent ist.

Die Klasse der Final-State-serialisierbaren Schedules wird FSSR genannt.

Beispiel (FSSR)

$\hat{S}_3 = \langle\ w^0(x)\ w^0(y)\ w^0(z)\ C^0\ r^1(x)\ r^2(y)\ w^1(y)\ r^3(z)\ w^3(z)\ C^3\ r^2(x)\ w^2(z)\ C^2$
$w^1(x)\ C^1\ r^\omega(x)\ r^\omega(y)\ r^\omega(z)\ C^\omega\ \rangle$

direkt nützlich:

$w^0(x) \rightarrow r^1(x)$ $\quad r^1(x) \rightarrow w^1(x)$ $\quad w^1(x) \rightarrow r^\omega(x)$
$r^1(x) \rightarrow w^1(y)$ $\quad w^1(y) \rightarrow r^\omega(y)$
$w^0(x) \rightarrow r^2(x)$ $\quad r^2(x) \rightarrow w^2(z)$
$w^0(y) \rightarrow r^2(y)$ $\quad r^2(y) \rightarrow w^2(z)$ $\quad w^2(z) \rightarrow r^\omega(z)$
$w^0(z) \rightarrow r^3(z)$ $\quad r^3(z) \rightarrow w^3(z)$

lebendig: $w^0(x)$, $w^0(y)$, $r^1(x)$, $r^2(y)$, $w^1(y)$, $r^2(x)$, $w^2(z)$, $w^1(x)$, $r^\omega(x)$, $r^\omega(y)$, $r^\omega(z)$

tot: $w^0(z)$, $r^3(z)$, $w^3(z)$

LRF = $\{(T^0, x, T^1), (T^1, x, T^\omega), (T^1, y, T^\omega), (T^0, x, T^2), (T^0, y, T^2), (T^2, z, T^\omega)\}$

Probeweise "T^3 vor T^2 vor T^1"

$\hat{S}_3' = w^0(x)\ w^0(y)\ w^0(z)\ r^3(z)\ w^3(z)\ r^2(y)$ …

Beispiel (FSSR) – Fortsetzung

$\hat{S}_3' = \langle\ w^0(x)\ w^0(y)\ w^0(z)\ C^0\ r^3(z)\ w^3(z)\ C^3\ r^2(y)\ r^2(x)\ w^2(z)\ C^2\ r^1(x)\ w^1(y)$
$w^1(x)\ C^1\ r^\omega(x)\ r^\omega(y)\ r^\omega(z)\ C^\omega\ \rangle$

direkt nützlich:

$w^0(x) \rightarrow r^1(x)$ $\quad r^1(x) \rightarrow w^1(x)$ $\quad w^1(x) \rightarrow r^\omega(x)$
$r^1(x) \rightarrow w^1(y)$ $\quad w^1(y) \rightarrow r^\omega(y)$
$w^0(x) \rightarrow r^2(x)$ $\quad r^2(x) \rightarrow w^2(z)$
$w^0(y) \rightarrow r^2(y)$ $\quad r^2(y) \rightarrow w^2(z)$ $\quad w^2(z) \rightarrow r^\omega(z)$
$w^0(z) \rightarrow r^3(z)$ $\quad r^3(z) \rightarrow w^3(z)$

lebendig: $w^0(x)$, $w^0(y)$, $r^2(y)$, $r^2(x)$, $w^2(z)$, $r^1(x)$, $w^1(y)$, $w^1(x)$, $r^\omega(x)$, $r^\omega(y)$, $r^\omega(z)$

tot: $w^0(z)$, $r^3(z)$, $w^3(z)$

LRF' = $\{(T^0, x, T^1), (T^1, x, T^\omega), (T^1, y, T^\omega), (T^0, x, T^2), (T^0, y, T^2), (T^2, z, T^\omega)\}$

LRF $\equiv$ LRF' $\quad$ S_3 ist also FSSR !

Bemerkungen zu FSSR

- RF und LRF berücksichtigen jeweils nur Schreibaktionen von nicht abgebrochenen Transaktionen
  Recovery ist ausgeklammert!
- LRF ist an toten Leseaktionen nicht interessiert.
  Die "Semantik" eines Schedules berücksichtigt nur lebendige Leser, d.h. solche, die ein Schreiben beeinflusst haben
  Reine Lesetransaktionen werden dabei ausgeklammert!
- Die Begriffe "nützlich" und "lebendig" reizen die syntaktische Fassung der (unbekannten) Semantik aus.
  komplizierte Verfahren!

2.3 View–Serialisierbarkeit

Das Problem, dass tote Aktionen bei FSSR ausgeklammert werden, lässt sich beheben, wenn man die Gleichheit der Reads-From-Relation eines gegebenen Schedules mit einem seriellen Schedule fordert.

View-Äquivalenz: Zwei Schedules S und S′ sind view-äquivalent, wenn $RF(\hat{S}) = RF(\hat{S}')$

View-Serialisierbarkeit (VSR):
Ein vollständiger Schedule S ist view-serialisierbar, wenn es einen seriellen Schedule S′ gibt, der zu S view-äquivalent ist.
Die Klasse der View-serialisierbaren Schedules wird VSR genannt.

Satz: $VSR \subset FSSR$

Beweis:
1) Da $LRF(S) \subseteq RF(S)$ ist jeder VSR-Schedule auch FSSR
2) Gegenbeispiel

Beispiel: VSR ist echte Teilmenge von FSSR

$\hat{S} = \langle w^0(x)\ w^0(y)\ r^1(x)\ w^2(x)\ w^2(y)\ r^1(y)\ r^\omega(x)\ r^\omega(y) \rangle$

Annahme: FSSR $\subseteq$ VSR

nützlich: $w^0(x) \rightarrow r^1(x)$, $w^2(y) \rightarrow r^1(y)$, $w^2(x) \rightarrow r^\omega(x)$, $w^2(y) \rightarrow r^\omega(y)$
lebendig: $w^2(x)$, $w^2(y)$, $r^\omega(x)$, $r^\omega(y)$
tot: $r^1(x)$, $r^1(y)$, $w^0(x)$, $w^0(y)$
LRF = $\{ (T^2, x, T^\omega) , (T^2, y, T^\omega) \}$

Serieller Schedule: $T^1 < T^2$ $\hat{S}' = \langle \ldots r^1(x)\ r^1(y)\ w^2(x)\ w^2(y) \ldots \rangle$ nützlich:
$w^0(x) \rightarrow r^1(x)$, $w^0(y) \rightarrow r^1(y)$, $w^2(x) \rightarrow r^\omega(x)$, $w^2(y) \rightarrow r^\omega(y)$
lebendig: $w^2(x)$, $w^2(y)$, $r^\omega(x)$, $r^\omega(y)$
tot: $r^1(x)$, $r^1(y)$, $w^0(x)$, $w^0(y)$
LRF(S) = LRF(S'), also S $\in$ FSSR

RF(S) = $\{ (T^0, x, T^1) , (T^2, y, T^1) , (T^2, x, T^\omega) , (T^2, y, T^\omega) \}$
$\Rightarrow$ Es gibt keinen seriellen Schedule, der diese RF hat

Untersuche alle möglichen seriellen Schedules:

$T^1 < T^2$: $\{ (T^0, x, T^1) , (T^0, y, T^1) , (T^2, x, T^\omega) , (T^2, y, T^\omega) \}$
$T^2 < T^1$: $\{ (T^2, x, T^1) , (T^2, y, T^1) , (T^2, x, T^\omega) , (T^2, y, T^\omega) \}$

Widerspruch zur Annahme FSSR $\subseteq$ VSR !

View–Serialisierbarkeit: Komplexität

Satz: Sei S ein Schedule ohne tote Leser. Dann gilt:
VSR = FSSR

Beweis: Ohne tote Leser gilt: RF $\equiv$ LRF, daraus folgt dann sofort, dass VSR = FSSR

Satz: Der Test, ob ein gegebener Schedule S aus VSR ist, ist NP-vollständig!

Überlegung :

1) Test auf View-Äquivalenz:
Ist linear in der Länge des Schedules

2) "Einfacher" VSR-Test (Enumeration der möglichen seriellen Schedules):
Es gibt n! serielle Schedules (n = Anzahl Transaktionen in S)

2.4 Konflikt-Serialisierbarkeit

Konflikt-Serialisierbarkeit basiert auf der grundlegenden Beobachtung, dass manche Aktionen eines Schedules vertauscht werden können, ohne dass sich dabei etwas ändert (sowohl aus der Sicht dieser Aktionen als auch aus der Sicht der jeweiligen Objekte). Dies ist nicht nur im einfachen read/write-Modell der Fall sondern kann auch auf beliebige (semantisch reiche) Aktionen verallgemeinert werden.

Beispiel:

$r^1(x) < r^2(x) \sim r^2(x) < r^1(x)$
$w^1(x) < w^2(y) \sim w^2(y) < w^1(x)$
$incr^1(x) < incr^2(x) \sim incr^2(x) < incr^1(x)$

Umgekehrt gibt es jedoch auch Paare von Aktionen, die nicht vertauscht werden können (da diese Vertauschung Auswirkungen auf die Aktionen bzw. die jeweiligen Objekte hat). In solchen Fällen bestehen Abhängigkeiten zwischen diesen Aktionen (bzw. müssen wir Abhängigkeiten annehmen).

Beispiel:

$r^1(x) < w^2(x) \not\sim w^2(x) < r^1(x)$
$w^1(x) < w^2(x) \not\sim w^2(x) < w^1(x)$
$decr^1(x) < decr^2(x) \not\sim decr^2(x) < decr^1(x)$
(wenn Untergrenze auf Zählerobjekt x gegeben)

Konfliktrelation

Aktionen, die nicht vertauscht werden können, sind in Konflikt. Die Konfliktrelation con gibt an, wann dies für Paare von Aktionen a und a' der Fall ist (solche Paare werden auch Konfliktpaare genannt)

Konfliktrelation con:

a con a' $\Leftrightarrow$ (1) a und a' werden auf dasselbe DB-Objekt x angewandt

(2) $a = read(x) \wedge a' = write(x)$ r/w-Konflikt

$\vee\ a = write(x) \wedge a' = read(x)$ w/r-Konflikt

$\vee\ a = write(x) \wedge a' = write(x)$ w/w-Konflikt

Im Read/Write-Modell sind also zwei Aktionen in Konflikt, wenn sie auf dasselbe Objekt zugreifen und mindestens eine Aktion ein Schreiben ist.

Für die Anwendung des Konfliktbegriffes auf beliebige Aktionen (für welche die einfache read/write-Semantik nicht mehr ausreicht), werden wir die Konfliktdefinition später entsprechend erweitern und verallgemeinern müssen.

Abhängigkeitsrelation

Die Abhängigkeitsrelation eines Schedules S enthält sämtliche Konfliktpaare aller Transaktionen aus S.

Abhängigkeit & Abhängigkeitsrelation:

Sei S ein Schedule. Aktion a^k ist abhängig von Aktion a^i in S, kurz $a^i \rightarrow a^k \Leftrightarrow$

(1)	$i \neq k$	T^i und T^k sind verschiedene Transaktionen,
(2)	$a^i < a^k$	a^i kommt vor a^k in S,
(3)	a^i con a^k	a^i und a^k sind in Konflikt,
(4)	C^i, C^k in S	beide Transaktionen sind abgeschlossen.

Die Abhängigkeitsrelation dep(S) ist die Menge aller abhängigen Paare in S: $dep(S) = \{(a^i, a^k) \in S \mid a^i \rightarrow a^k \}$

Beispiele:

$S_1 = \langle w^1(a)\ r^2(a)\ r^1(b)\ w^2(b)\ C^1\ C^2 \rangle$

abhängig: $w^1(a) \rightarrow r^2(a)$, $r^1(b) \rightarrow w^2(b)$

$dep(S_1) = \{(w^1(a), r^2(a)), (r^1(b), w^2(b))\}$

$S'_1 = \langle w^1(a)\ r^1(b)\ C^1\ r^2(a)\ w^2(b)\ C^2 \rangle$

abhängig: $w^1(a) \rightarrow r^2(a)$, $r^1(b) \rightarrow w^2(b)$

$dep(S'_1) = \{(w^1(a), r^2(a)), (r^1(b), w^2(b))\}$

Konflikt-Äquivalenz

Die Abhängigkeitsrelation ist ein Mittel, um die Semantik eines Schedules zu erfassen (und daher, um zwei Schedules zu vergleichen → siehe Vorlesung IS-G).

Konflikt-Äquivalenz: Zwei Schedules S und S′ sind konflikt-äquivalent, wenn beide Schedules dieselben Aktionen beinhalten und wenn die Abhängigkeitsrelation identisch ist, also dep(S) = dep(S′) ist.

Beispiel:

$S_1 = \langle w^1(a)\ r^2(a)\ r^1(b)\ w^2(b)\ C^1\ C^2 \rangle$

$S'_1 = \langle w^1(a)\ r^1(b)\ C^1\ r^2(a)\ w^2(b)\ C^2 \rangle$

S_1 und S'_1 sind konflikt-äquivalent, da $dep(S_1) = dep(S'_1)$

CPSR

Konflikt-Serialisierbarkeit (CPSR):

Ein Schedule S ist konflikt-serialisierbar, wenn seine Committed Projection C(S) konflikt-äquivalent zu einem seriellen Schedule S_{ser} ist.

Die Klasse der konflikt-serialisierbaren Schedules wird CPSR (Conflict Preserving Serializable) genannt.

Konflikt-äquivalente Schedules sind also erzeugbar durch das Kommutieren von Aktionen, die nicht abhängig sind.

Beobachtung: Durch die Betrachtung der Committed Projection wird erneut – analog zu FSSR und VSR – der Einfluss abgebrochener Transaktionen nicht berücksichtigt.

Beispiel:

$S_1 = \langle w^1(a)\ r^2(a)\ r^1(b)\ w^2(b)\ C^1\ C^2 \rangle$

$S'_1 = \langle w^1(a)\ r^1(b)\ C^1\ r^2(a)\ w^2(b)\ C^2 \rangle$

S_1 ist CPSR, da S_1 und S'_1 konflikt-äquivalent sind und da S'_1 seriell ist.

Vertauschen von Aktionen

Beobachtung: Bei serieller Ausführung werden alle Aktionen einer Transaktion T^i vor/nach allen Aktionen einer Transaktion T^k ausgeführt.

Beispiel:

$S = \langle a_1^1\ a_2^1\ a_3^1\ a_1^2\ a_1^3\ a_4^1 \rangle$ falls kein Konflikt

$\equiv \langle a_1^1\ a_2^1\ a_3^1\ a_1^2\ a_4^1\ a_1^3 \rangle$ falls kein Konflikt

$\equiv \langle a_1^1\ a_2^1\ a_3^1\ a_4^1\ a_1^2\ a_1^3 \rangle$

… seriell !

Wir benötigen daher ein Verfahren zum Nachweis der Existenz eines seriellen Schedules, der durch Vertauschung von Aktionen, die nicht in Konflikt sind, entsteht.

Serialisierungsgraph

Serialisierungsgraph (Abhängigkeitsgraph)

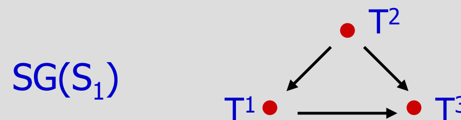
Der Serialisierungsgraph (Abhängigkeitsgraph) SG(S) eines Schedules S ist ein Graph dessen

- Knoten alle mit Commit beendeten Transaktionen in S und dessen
- Kanten alle Paare (T^i, T^k) sind für die gilt: es gibt Aktionen $a^i \in T^i$ und a^k in T^k mit $(a^i, a^k) \in dep(S)$.

Beispiele:

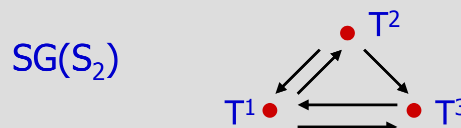
$S_1 = \langle r_1^1(a)\ r_1^2(a)\ w_2^1(a)\ r_1^3(a)\ w_2^2(b)\ w_2^3(b)\ C^1\ C^2\ C^3 \rangle$

$SG(S_1)$



$S_2 = \langle r_1^1(a)\ r_1^2(a)\ w_2^2(a)\ r_1^3(a)\ w_2^1(a)\ w_2^3(a)\ C^1\ C^2\ C^3 \rangle$

$SG(S_2)$



Serialisierbarkeitstheorem

Satz: Ein Schedule S ist genau dann konflikt-serialisierbar (S ∈ CPSR), wenn sein Serialisierungsgraph SG(S) zyklenfrei ist.

Beweis: i) S serialisierbar ⇒ SG(S) zyklenfrei

… siehe Übung (Übungsblatt 2)

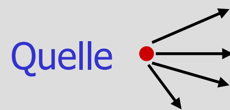
ii) SG(S) zyklenfrei ⇒ S serialisierbar

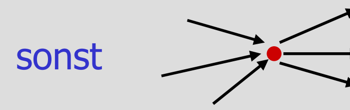
Idee: Gegeben ein azyklischer Serialisierungsgraph SG(S). Dann liefert die topologische Sortierung von SG(S) äquivalente serielle Schedules

Topologische Sortierung von SG(S)

Hilfssatz: SG(S) zyklenfrei ⇒ SG(S) topologisch sortierbar

Vorüberlegung: Wenn SG(S) zyklenfrei ist, dann gibt es mindestens eine Quelle/Senke

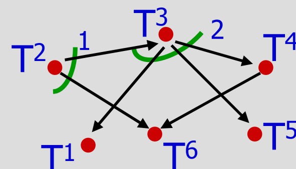




Topologische Sortierung:

Eine topologische Sortierung von SG(S) ist die Folge aller Knoten $\langle i_1 \; i_2 \ldots i_n \rangle$ wobei gilt: $i_r < i_s \Rightarrow$ es gibt keinen Pfad von i_s nach i_r in SG(S).

Beispiel:



$T^2 \rightarrow T^3 \rightarrow$ $T^4\,T^5\,T^6\,T^1$

$T^1\,T^4\,T^6\,T^5$

$T^4\,T^5\,T^1\,T^6$

… mit $T^4 \rightarrow T^6$

CPSR ⊂ VSR

View-Serialisierbarkeit hat den entscheidenden Nachteil, dass der Test $S \in VSR$ NP-vollständig ist.

Der Zyklentest für SG(S) und damit der Test $S \in CPSR$ ist jedoch in polynomialer Zeit möglich.

Aber: was sind die Konsequenzen?

Satz: CPSR ⊂ VSR

Beweis: siehe nächste Folie

Die Konsequenz ist also, dass es view-serialisierbare Schedules gibt, die nicht konflikt-serialisierbar sind (also Schedules, die eigentlich korrekt sind werden durch CPSR nicht als solche erkannt)!

Wichtige Fragen sind daher: wie viele sind dies?
welche? und sind diese praktisch relevant?

Beweis: Aus CPSR folgt VSR

Zu zeigen: $S \in CPSR \Rightarrow S \in VSR$, also alle CPSR-Schedules sind auch VSR

Wenn $S \in CPSR$, dann $\exists S_{ser}$ mit gleicher Abhängigkeitsrelation;

wir zeigen: $\hat{S}_{ser}$ hat auch identische RF-Relation wie $\hat{S}$.

Fälle: $i, k \neq 0, \omega$

Sei $(T^i, x, T^k) \in RF(\hat{S})$

$\Rightarrow$ $w^i(x) <_S r^k(x)$; kein $w^m(x)$ dazwischen

weil $w^i(x) \rightarrow r^k(x)$ ist $(w^i(x), r^k(x))$ in dep(S)

$\Rightarrow$ $w^i(x) <_{ser} r^k(x)$

Da jedes $w^m(x)$ in Konflikt mit $w^i(x)$ und $r^k(x)$ wäre,

ist auch in $\hat{S}_{ser}$ kein $w^m(x)$ dazwischen

$\Rightarrow$ $(T^i, x, T^k) \in RF(\hat{S}_{ser})$

Fall i=0:

Sei $(T^0, x, T^k) \in RF(\hat{S})$, also kein $w^m(x)$ vor $T^k(x)$ in $\hat{S}$

$\Rightarrow$ kein $w^m(x)$ vor $r^k(x)$ in $\hat{S}_{ser}$

Fall $k=\omega$:

Sei $(T^i, x, T^{\omega}) \in RF(\hat{S})$, also $w^i(x)$ ist letzter Schreiber

$\Rightarrow$ (wie oben) auch $w^i(x)$ in $\hat{S}_{ser}$ letzter Schreiber

$\Rightarrow RF(\hat{S}) = RF(\hat{S}_{ser})$

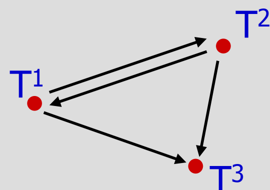
Beispiel: VSR enthält CPSR

Es gibt Schedules S mit: $S \in VSR$ und $S \notin CPSR$

Beweis durch Angabe eines Beispiels:

$S = \langle r_1^1(a)\ w_1^2(a)\ w_2^1(a)\ w_1^3(a)\ C^1\ C^2\ C^3 \rangle$

$S \notin CPSR$:



Aber: $S \in VSR$

$S_{ser} = \langle T^1\ T^2\ T^3 \rangle$ hat gleiche RF wie S

Monotonie von CPSR

Die Monotonie einer Klasse $\mathcal{K}$ von Schedules besagt, dass für die Projektion von S = (τ, $\mathbb{A}$, <) auf einen beliebigen Subschedule S′ = (τ', $\mathbb{A}'$, <′) mit $\tau' \subset \tau$ gilt: Falls S in $\mathcal{K}$ ist, dann gilt dies auch für S′. Dies ist vor allem für das Design von Protokollen zum dynamischen Scheduling wichtig.

Ist VSR monoton?

Sei S_τ View-serialisierbar und sei $\tau' \subset \tau$ so dass $S_{\tau'}$ Subschedule von S ist (durch Streichen der Aktionen von $\tau \backslash \tau'$). Ist $S_{\tau'}$ View-serialisierbar?

$S = \langle w_1^1(x)\ w_1^2(x)\ w_2^2(y)\ w_2^1(y)\ w_1^3(x)\ w_2^3(y)\ C^1\ C^2\ C^3 \rangle$

$S' = \langle w_1^1(x)\ w_1^2(x)\ w_2^2(y)\ w_2^1(y)\ C^1\ C^2 \rangle$

$RF(\langle T^1 T^2 \rangle) \neq RF(S') \neq RF(\langle T^2 T^1 \rangle) \Rightarrow S' \notin VSR$

Aber: $S \in VSR$ weil $RF(S) = RF(\langle T^1 T^2 T^3 \rangle)$ … VSR ist nicht monoton!

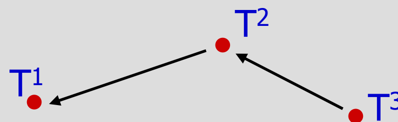
Satz: CPSR ist die grösste monotone Teilklasse in VSR
(ohne Beweis)

2.5 Ordnungserhaltende Serialisierbarkeit

CPSR ist ein sehr intuitives Korrektheitskriterium. Allerdings gibt es auch hier noch unerwartete (?) Effekte.

Beispiel:

$S = \langle r^1(x)\ r^2(y)\ w^2(y)\ r^1(y)\ C^1\ r^3(z)\ C^3\ r^2(z)\ w^2(z)\ C^2 \rangle$



Offensichtlich ist S aus CPSR mit
Serialisierungsreihenfolge: $T^3 < T^2 < T^1$

Allerdings gilt für die
Commitreihenfolge: $T^1 < T^3 < T^2$

Insbesondere ist T^1 komplett beendet, bevor T^3 startet; allerdings wird T^3 VOR T^1 serialisiert!

OPSR

Ordnungserhaltende Serialisierbarkeit:

Ein Schedule S ist (transaktions-) ordnungserhaltend serialisierbar, wenn $S \in CPSR$ und wenn es mindestens einen seriellen äquivalenten Schedule S′ gibt, in dem die in S vollständig geordneten Transaktionen auch in S′ die gleiche Reihenfolge haben.

Die Klasse der ordnungserhaltend serialisierbaren Schedules wird OPSR (Order Preserving Serializable) genannt.

Satz: $OPSR \subset CPSR$

Beweis:

– OPSR Teilmenge von CPSR: Klar, nach obiger Definition (jeder OPSR-Schedule ist auch CPSR)

– OPSR ist echte Teilmenge von CPSR: Beispiel hierfür (Schedule aus CPSR, aber nicht aus OPSR) siehe vorige Folie.

2.6 Commit-Ordnungserhaltung

OPSR betrachtet nur die Erhaltung der Reihenfolge von Transaktionen, die in einem Schedule vollständig geordnet sind, aber nicht die Reihenfolge, in der die Commits erfolgen. Die Commit-Ordnungserhaltung ist ein weiteres Korrektheitskriterium, das genau diese Ordnung zusätzlich betrachtet.

Grundlegende Idee:

Man will nicht nur sicherstellen, dass parallele Transaktionen garantiert korrekt ausgeführt werden, sondern möchte auch noch die (interne) Serialisierungsreihenfolge kennen (bzw. vorgeben). Dies ist über die Commit-Reihenfolge, die nach aussen hin bekannt ist, möglich.

Commit-ordnungserhaltende Serialisierbarkeit:

Ein Schedule S ist commit-ordnungserhaltend serialisierbar wenn für jedes Paar mit Commit abgeschlossener Transaktionen T^i, T^k aus S gilt:
falls (a^i, a^k) in dep(S), dann ist $C^i < C^k$.

Die Klasse der commit-ordnungserhaltend serialisierbaren Schedules wird COPSR (Commit Order Preserving Serializable) genannt.

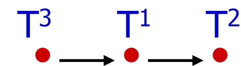
COPSR

COPSR verlangt also, dass die Commit-Ordnung eines Schedules mit dessen Serialisierungsordnung übereinstimmt. Durch das “Bekanntmachen” bzw. das “Vorgeben” der Serialisierungsordnung ist COPSR ein Korrektheitskriterium, das insbesondere in verteilten Umgebungen (verteilte Transaktionen) von grosser Bedeutung ist.

Satz: COPSR $\subset$ OPSR

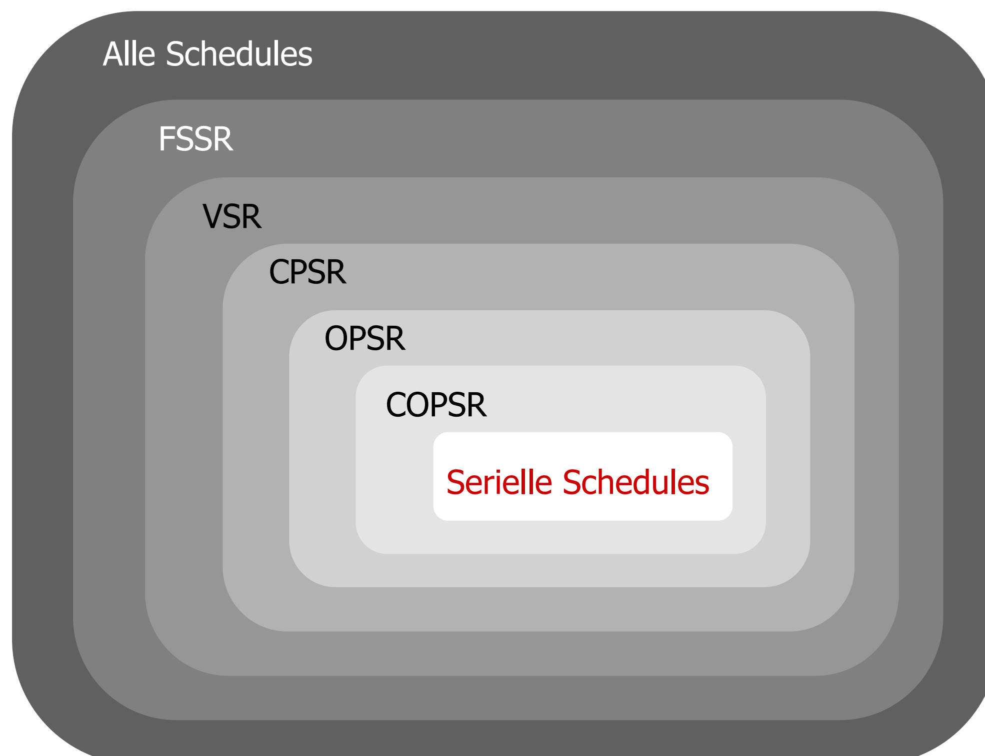
Beweis:

- COPSR Teilmenge von OPSR: Jeder COPSR-Schedule ist zunächst CPSR (alle Konfliktpaare zweier Transaktionen werden gleich geordnet). Zudem respektiert COPSR die vorgegebene Commit-Reihenfolge, darf also eine Transaktion T^i, die komplett vor einer Transaktion T^k ausgeführt (und damit committed) wurde, nicht danach serialisieren.
- COPSR ist echte Teilmenge von OPSR:
  Beispiel: $S = \langle w_1^3(y)\ C^3\ w_1^1(x)\ r_1^2(x)\ C^2\ w_2^1(y)\ C^1 \rangle$



  $S \in$ OPSR (T^3 wird vor T^2 und T^1 serialisiert)
  $S \notin$ COPSR (Commit-Reihenfolge ist $C^3 < C^2 < C^1$)

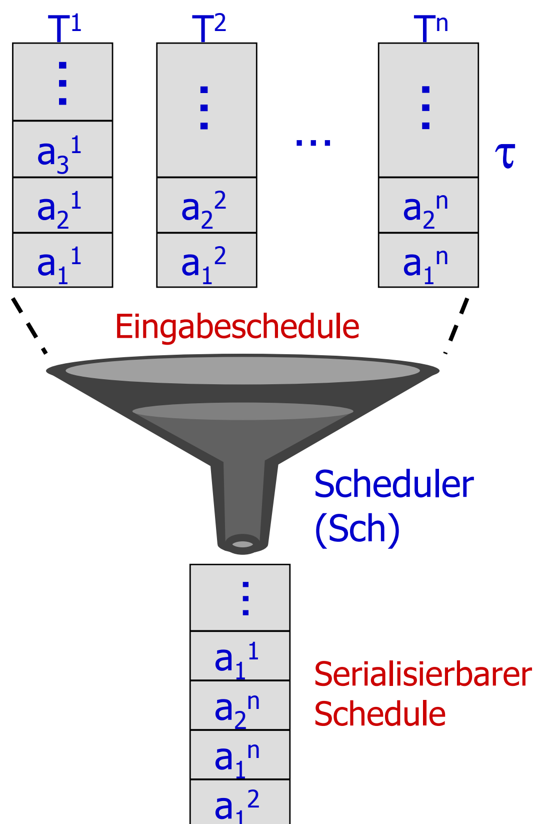
2.7 Abschliessender Vergleich



Teil II – Concurrency Control

Kapitel 2: Serialisierbarkeitstheorie – Korrektheitskriterien
Kapitel 3: Konservative Concurrency Control-Protokolle
Kapitel 4: Optimistische Concurrency Control-Protokolle
Kapitel 5: Mehrversionen-Concurrency Control
Kapitel 6: Mehrversionen-Protokolle

Aufgabe des Schedulers



- Die Aufgabe eines Schedulers ist die Transformation eines wohlgeformten Eingabeschedules in einen serialisierbaren Schedule.
- Ein Scheduler kann also als Abbildung
  Sch: ES $\mapsto$ AS
  angesehen werden, wobei ES die Menge aller Eingabeschedules und AS die Menge aller serialisierbaren Ausgabeschedules ist, d.h. AS $\subseteq$ CPSR
- Die Güte eines Schedulers bestimmt sich aus der "Grösse" der Ausgabeschedule-Menge AS, d.h. je mehr korrekte Schedules ein Scheduler zulässt, umso besser ist er.
- Ein weiteres Gütekriterium eines Schedulers ist dessen Fixpunktmenge: je grösser diese ist, umso besser ist der Scheduler.
  (die Fixpunktmenge eines Schedulers ist die Menge der korrekten Eingabeschedules, die nicht verändert werden, also
  Fixpunkt: S' = Sch(S) = S)

Concurrency Control-Protokolle

- Die Spielregeln des Schedulers, d.h. sein Algorithmus, werden als Concurrency Control Protokoll bezeichnet.
- Für die Praxis relevant sind nur dynamische ("on-line") Scheduler.
  - Das bedeutet, dass Concurrency Control-Protokolle die Entscheidung, ob Aktion a_i^k der Transaktion T^k ausgeführt werden kann oder verzögert werden muss, oder ob T^k zurückgesetzt werden muss ausschliesslich mit dem vor a_i^k liegenden Präfix des Schedules fällen können.
  - Je nach Art der Protokolle wird dabei diese Entscheidung zumeist entweder nur für die Aktionen auf Datenobjekten (im r/W-Modell: r(x) bzw. w(x)), oder nur für die Terminierungsaktionen (Commit bzw. Abort) durchgeführt. (einige wenige Protokolle verlangen eine Entscheidung für beide Arten von Aktionen)
- Ausgeschlossen ist auf jeden Fall die Analyse aller Aktionen der Transaktionen τ vor deren Ausführung.

Einteilung: konservativ vs. optimistisch

Konservative Protokolle (Kapitel 3):

Ein konservativer Scheduler setzt voraus, dass vor jeder Aktion überprüft wird, ob diese Aktion ausgeführt werden darf oder nicht. Ein Commit einer Transaktion hingegen ist in konservativen Protokollen immer erlaubt.

Konservative Scheduler arbeiten häufig mit Sperrprotokollen. Da solche Verfahren mit Konflikten rechnen und vorsorglich Sperren setzen, werden sie auch als "pessimistische" Verfahren bezeichnet. Eine Besonderheit dieser sperrbasierten Verfahren ist die Möglichkeit, die Ausführung von Aktionen zu verzögern um damit Aktionen umzuordnen.

Andere konservative Protokolle, die keine Umordung vorsehen, können nur feststellen, ob eine Aktion erlaubt ist bzw. die zugehörige Transaktion abbrechen.

Optimistische Protokolle (Kapitel 4):

Ein optimistischer Scheduler erlaubt die Ausführung von Aktionen, ohne dass zuvor überprüft werden muss, ob diese Aktion zulässig ist (solche Verfahren werden teilweise auch als "aggressive" Verfahren bezeichnet). Allerdings müssen optimistische Scheduler vor dem Commit eine Validierung bzw. Konfliktanalyse durchführen, um Korrektheit zu garantieren.

Es sind auch hybride Verfahren möglich, die zwar Sperren für Aktionen verwenden, aber trotzdem eine Validierung zum Commit-Zeitpunkt erfordern.

Kapitel 3: Konservative Protokolle

3.1 Zwei-Phasen-Sperrprotokoll (2PL)
3.2 striktes Zwei-Phasen-Sperrprotokoll (S2PL)
3.3 Deadlocks
3.4 Timestamp Ordering
3.5 Serialisierungsgraph-Test

3.1 Zwei-Phasen-Sperrprotokoll (2PL)

Idee: Vor dem Zugriff (sowohl lesend als auch schreibend) auf ein Datenobjekt muss eine Sperre erworben werden.
Dabei werden die Sperren so gewählt, dass keine Konflikt-aktionen auf demselben Objekt möglich sind.

Dabei sollen diese Sperren transparent für die Transaktionen (und damit für den Benutzer bzw. das Anwendungsprogramm) vom System gesetzt werden.

Beispiel:

Transaktion $T^1 = \langle a^1_1 \; a^1_2 \; \ldots \; a^1_k \rangle$

wird automatisch transformiert in

$T^{1*} = \langle L(a^1_1)\, a^1_1\, L(a^1_2)\, a^1_2 \ldots L(a^1_k)\, a^1_k\, U(a^1_1)\, U(a^1_2) \ldots U(a^1_k) \rangle$

wobei $L(a^i_k)$ eine Sperre auf das Objekt der Aktion a^i_k erwirbt (Lock)
$U(a^i_k)$ die gehaltene Sperre wieder freigibt (Unlock)

Sperrmodi und -verträglichkeit

Das Ziel des 2PL-Verfahrens ist, keine Konfliktaktionen auf demselben Objekt zuzulassen, aber zugleich Aktionen, die nicht in Konflikt stehen, zu erlauben. Zwei parallele Transaktionen dürfen also dasselbe Objekt lesen, wenn aber auch auf das Objekt geschrieben wird, darf dies nur eine Transaktion (und keine andere darf das Objekt lesen).

Um dies zu realisieren, benötigen wir unterschiedliche Sperrmodi:

- S-lock (= Shared Lock) für lesenden Zugriff, also für r(x)
- X-lock (= Exclusive Lock) für schreibenden Zugriff, also für w(x)

Für diese Sperrmodi gilt folgende Verträglichkeit (die aus der Konfliktrelation abgeleitet ist):

		Transaktion fordert	
		Shared	eXclusive
Objekt belegt mit	Shared	+	—
	eXclusive	—	—

\+ angeforderte Sperre kann vergeben werden
(Reihenfolge von Lesern vertauschbar. Kein Konflikt im r/r-Fall)

— angeforderte Sperre kann NICHT vergeben werden
(da sonst w/r, r/w, w/w-Konflikt)

Zwei-Phasen-Sperrprotokoll

Zwei-Phasen-Sperrprotokoll (2PL)

Ein Scheduler hält das Zwei-Phasen-Sperrprotokoll (2PL) ein, wenn:

(1) Vor der Ausführung jeder Aktion a^i_k Sperren $L(a^i_k)$ erworben werden (S-Sperre für Leseaktionen, X-Sperre für Schreibaktionen)

(2) Eine Sperre $L(a^i_k)$ mindestens solange gehalten wird, bis die Aktion a^i_k ausgeführt wurde

(3) Nach dem Freigeben einer Sperre keine weitere angefordert wird.

Erklärungen

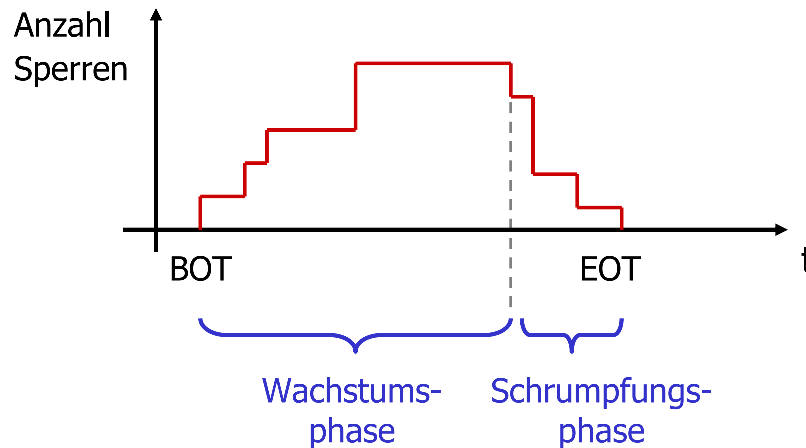
Zu (1): Hiermit werden Konfliktaktionen in die Reihenfolge der Sperranforderung gebracht.

Falls eine Sperranforderung für Aktion a^i_k von T^i nicht gewährt werden kann, muss T^i warten!

Zu (2): Die Aktionen müssen auch wirklich in der vom Scheduler vorgeschriebenen Reihenfolge ausgeführt werden.

Zweiphasigkeit

Zu (3): Sperranforderungen und –freigaben sind also in zwei Phasen aufgeteilt. Zunächst erwirbt eine Transaktion in der Wachstumsphase Sperren für alle Aktionen. Mit der ersten Sperrfreigabe tritt die Schrumpfungsphase ein, es dürfen dann keine neuen Sperren mehr angefordert werden.



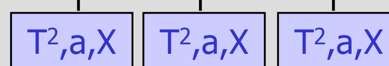
Beispiel für einen 2PL-Ablauf

Ausführung des Eingabeschedules ⟨ $r^1(a)$ $w^2(a)$ $w^1(b)$ C^1 $w^2(b)$ C^2 ⟩

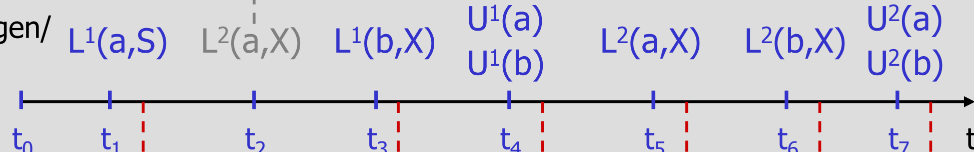
Locktabelle des 2PL-Schedulers zu folgenden Zeitpunkten:

Zeit / Objekt	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
a	—	(T^1, S)	(T^1, S)	(T^1, S)	—	(T^2, X)	(T^2, X)	—
b	—	—	—	(T^1, X)	—	—	(T^2, X)	—

Warteschlange für den Sperr-Erwerb



Sperranforderungen/ -freigaben des Schedulers



Ausgabeschedule ⟨ $r^1(a)$ $w^1(b)$ C^1 $w^2(a)$ $w^2(b)$ C^2 ⟩

Korrektheit von 2PL

Satz (2PL): Ein 2PL-Scheduler erzeugt nur serialisierbare (CPSR) Schedules.

Beweis: Zu zeigen: S durch 2PL enstanden $\Rightarrow$ S $\in$ CPSR
d.h. SG(S) hat keinen Zyklus

Vorbereitung: Kante $T^i \rightarrow T^k$ in SG(S)
$\Rightarrow \exists$ Objekt x_i für das $U^i(x_i)$ vor $L^k(x_i)$ in S

Annahme: Sei $T^1 \rightarrow T^2 \rightarrow \ldots \rightarrow T^n \rightarrow T^1$ Zyklus in SG(S)

$$\Rightarrow \quad U^1(x_1) < L^2(x_1)$$
$$U^2(x_2) < L^3(x_2)$$
$$\ldots$$
$$U^n(x_n) < L^1(x_n)$$

$\Rightarrow U^1(x_1) < L^1(x_n) \Rightarrow \neg$2PL Widerspruch!

2PL & OPSR: Das 2PL-Protokoll erzeugt nicht nur CPSR Schedules, sondern garantiert auch noch Ordnungserhaltung (OPSR)

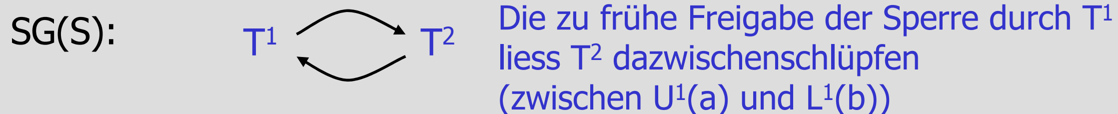
Ist die Zweiphasigkeit zu streng?

Zu (3): Zweiphasigkeit garantiert, dass alle Konfliktaktionen in derselben Reihenfolge ausgeführt werden.

Beispiel: $T^1 = \langle r^1(a)\ w^1(b)\ C^1 \rangle$
$T^2 = \langle w^2(a)\ w^2(b)\ C^2 \rangle$

$S = \langle L^1(a,S)\ r^1(a)\ U^1(a)\ L^2(a,X)\ w^2(a)\ L^2(b,X)\ w^2(b)\ U^2(a)\ U^2(b)\ C^2$
$L^1(b,X)\ w^1(b)\ U^1(b)\ C^1 \rangle$

SG(S):



Die zu frühe Freigabe der Sperre durch T^1 liess T^2 dazwischenschlüpfen (zwischen $U^1(a)$ und $L^1(b)$)

$\Rightarrow$ Aufhebung der Zweiphasigkeit nicht möglich

$\Rightarrow$ 2PL kann nicht ohne weitere Voraussetzungen verbessert werden !

Ausblick: Erweiterung von 2PL

2PL kann nur durch Berücksichtigung zusätzlicher Kenntnisse (mehr Semantik) erweitert verbessert werden.

Beispiele:

- Eine Transaktion legt beim Lesen eine private Kopie an. Wiederholbares Lesen kann dann über diese private Kopie garantiert werden, erfordert keine Sperren.
  - Verallgemeinerung: Mehrversionen-Concurrency-Control (Kapitel 5)
- Berücksichtigung der Kenntnis der Reihenfolge von Aktionen auf Datenobjekten
  - Beispiel Indexseiten: Einstieg immer über Wurzel, Zugriff in Richtung Wurzel → Blatt. Dadurch können Sperren problemlos früher freigegeben werden.
  - Verallgemeinerung: Tree-Locking

3.2 Striktes Zwei-Phasen-Sperrprotokoll

Das 2PL-Protokoll erlaubt das frühe Freigeben von Sperren (vor dem Commit). Da aber ein (on-line) Scheduler in der Regel nur den Präfix eines Schedules zur Verfügung hat, ist dieses frühzeitige Freigeben risikoreich!

Abhilfe: Sperren müssen immer bis zum Ende der jeweiligen Transaktion gehalten werden.

Striktes Zwei-Phasen-Sperrprotokoll (S2PL)

Ein Scheduler hält das strikte Zwei-Phasen-Sperrprotokoll (S2PL) ein, wenn:

(1) Vor der Ausführung jeder Aktion a^i_k Sperren $L(a^i_k)$ erworben werden (S-Sperre für Leseaktionen, X-Sperre für Schreibaktionen)

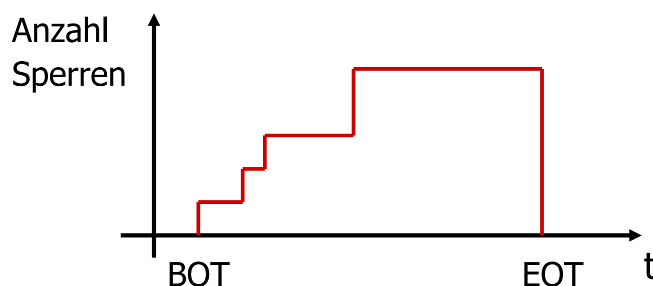
(2) Alle Sperren $L(a^i_k)$ bis zum Ende der jeweiligen Transaktion gehalten werden.

Korrektheit von S2PL

Offensichtlich ist das S2PL-Protokoll strenger als das 2PL-Protokoll. Also garantiert auch S2PL, dass jeder erzeugte Schedule OPSR und damit auch CPSR ist. Zusätzlich:

Satz (S2PL): Ein S2PL-Scheduler erzeugt commit-ordnungserhaltende (COPSR) Schedules.

Beweis: Da alle Sperren bis zum Commit gehalten werden müssen, gilt im S2PL-Protokoll für alle Konfliktpaare (a^i, a^k):
$L^i < a^i < U^i < C^i < L^k < a^k$ und damit natürlich auch $C^i < C^k$.



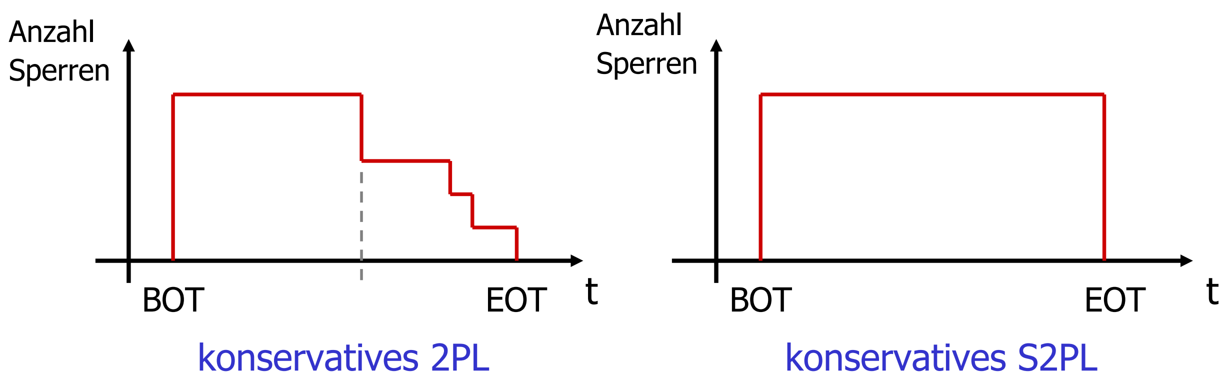
Die Schrumpfungsphase fällt also im S2PL mit dem Commit bzw. dem Abort zusammen.

Konservatives 2PL

Eine Variante des 2PL (bzw. des S2PL)-Protokolls, das konservative 2PL (konservative S2PL), verlangt dass alle Sperren, die im Lauf einer Transaktion benötigt werden, vor der ersten Aktion erworben werden.

Dieses Frühzeitige Anfordern von Sperren wird auch als Preclaiming bezeichnet.

Die Wachstumsphase beim konservativen (S)2PL-Protokoll fällt also mit dem BOT zusammen.



konservatives 2PL   konservatives S2PL

3.3 Deadlocks

Ein entscheidender Nachteil des 2PL-Protokolls (und des S2PL-Protokolls) ist die Tatsache, dass zyklische Wartebeziehungen entstehen können. Das bedeutet, Transaktionen warten (prinzipiell unendlich lange) auf Sperren, die von anderen Transaktionen gehalten werden, dort aber nicht freigegeben werden können (weil auch diese Transaktionen auf weitere Sperren warten).

Solche zyklischen Wartebeziehungen werden auch als Deadlocks bezeichnet.

Beispiel: Gegeben sei folgender Eingabeschedule ES

$ES = \langle r^1(x)\ w^2(y)\ w^2(x)\ C^2\ w^1(y)\ C^1 \rangle$

Ein S2PL-Scheduler produziert folgenden Ausgabeschedule S:

$S = \langle L^1(x,S)\ r^1(x)\ L^2(y,X)\ w^2(y) \quad \rangle$

Transaktion T^2 wartet auf eine Sperre, die von T^1 gehalten wird (und umgekehrt); keine Transaktion kann also fortfahren. Wegen der Zweiphasigkeit kann auch keine Transaktion bereits jetzt eine Sperre freigeben ⇒ unendliches Warten!

Deadlock-Erkennung: Wait-For-Graph

Eine wesentliche Voraussetzung zur Behebung von Deadlocks ist zunächst ein Mechanismus, um zyklisches Warten überhaupt zu erkennen.

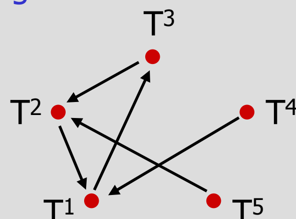
Dies geschieht zumeist mit Hilfe eines Wartegraphen (Wait-For-Graph, WFG). Knoten dieses Graphen sind Transaktionen, Kanten $T^i \rightarrow T^k$ werden genau dann eingefügt, wenn T^i auf eine Sperre wartet, die von T^k gehalten wird (T^i wartet auf Unlock von T^k)

Beispiel: Gegeben sei folgender Eingabeschedule ES

$ES = \langle r^3(a)\ w^1(b)\ w^1(c)\ w^1(a)\ r^2(d)\ w^2(e)\ r^4(b)\ w^2(c)\ r^3(e)\ r^5(e)\ C^2\ C^3\ C^5\ C^1\ C^4 \rangle$

Die Erzeugung des Ausgabeschedules S zu ES durch einen 2PL-Scheduler führt dann zu folgenden Wartebeziehungen:

WFG(S):



Deadlock-Auflösung

Deadlocks entsprechen einem Zyklus im Wartegraphen. Es werden also Graphen-algorithmen zur effizienten Zyklenerkennung in gerichteten Graphen benötigt.

Wann sollte diese Zyklenerkennung vorgenommen werden? Mögliche Strategien:

- Regelmässig, nach bestimmter Zeit (periodisch)
- Nach jedem Einfügen einer Kante in den WFG, also bei jeder nicht gewährten Sperranforderung (kontinuierlich)

Deadlocks können nur aufgelöst werden, indem Transaktionen aus dem WFG-Zyklus zurückgesetzt werden (schliesslich wartet jede dieser Transaktionen und kann nicht fortfahren). Allerdings entstehen dadurch "Kosten" (Transaktionen, die abgebrochen werden, haben bereits Ressourcen verbraucht).

Welche Transaktion(en) soll(en) zurückgesetzt werden (Auswahl des "Deadlock-Opfers")? Mögliche Strategien:

- Letzte Transaktion, die eine Kante in den WFG eingefügt hat (last blocked)
- Transaktion, die am wenigsten Sperren hält (minimum locks)
- Transaktion, die am wenigsten Rücksetzkosten verursacht, z.B. diejenige, die am wenigsten Update-Aktionen durchgeführt hat (minimum work)
- Transaktion, welche die meisten Zyklen auflöst (most cycles); dies ist vor allem beim periodischen Check des WFG relevant
- Transaktion, deren Rücksetzen am meisten Kanten aus dem WFG entfernt (most edges)

Deadlock-Vermeidung

Das Ziel von Protokollen zur Deadlock-Vermeidung ist es, den Aufwand des Führens eines WFG und den Zyklentest zu vermeiden, indem Deadlocks erst gar nicht erlaubt werden. Hierzu muss das (S)2PL also noch zusätzlich eingeschränkt werden.

Folgende Erweiterungen des (S)2PL garantieren, dass Deadlocks nicht auftreten können:

- Konservatives (S)2PL, also Preclaiming
- Prioritäten/Zeitstempel-Verfahren
- Einhalten einer fest vorgegebenen Reihenfolge bei der Sperranforderung (dies ist vor allem dann sinnvoll, wenn Datenobjekte einer bestimmten Ordnung gehorchen –z.B. baumartig strukturiert sind– und die Reihenfolge des Zugriffs dadurch schon festgelegt ist → Tree-Locking)

Preclaiming

Deadlockfreiheit beim Preclaiming setzt voraus, dass alle Sperranforderungen zu Beginn der Transaktion (und damit vor der ersten Aktion) atomar erfolgen.

Der Nachteil des Preclaiming ist, dass zumeist mehr gesperrt wird als unbedingt nötig ist

- Schliesslich weiss eine Transaktion zu Beginn nicht unbedingt, auf welche Datenobjekte sie wirklich zugreift und muss daher vom "worst-case"-Fall ausgehen

Dadurch wird jedoch auch der Parallelitätsgrad reduziert

- Andere Transaktionen müssen auf die Freigabe von Sperren warten, die eigentlich gar nicht benötigt würden.

Neben der Tatsache, dass keine Transaktionen wegen zyklischem Warten zurückgesetzt werden (und damit Zusatzkosten verursachen) ist ein entscheidender Vorteil des Preclaimings, dass keine Lock-Konversionen nötig sind.

- Lock-Konversionen entstehen, wenn eine Transaktion T^k ein Objekt a zunächst liest (und dafür eine S-Sperre erwirbt) und später schreiben möchte (dann also eine X-Sperre benötigt). Natürlich kann Letztere nur gewährt werden, wenn T^k alleiniger Leser ist. Das bedeutet letztendlich, dass auch Lock-Konversionen Deadlocks erzeugen können.

Prioritäten/Zeitstempel-Verfahren

Grundlegende Idee:

Falls einer Transaktion T^k eine Sperre nicht gewährt wird, weil eine andere Transaktion T^i sie hält, besteht Deadlockgefahr. Also bekommt jede Transaktion T^i eine Priorität $P(T^i)$ zugewiesen. Je nach Priorität der Transaktion, die eine Sperre anfordert und derjenigen der Transaktion, die diese Sperre hält, wird gewartet oder zurückgesetzt.

Häufig werden für diese Prioritäten Zeitstempel verwendet:

Jeder Transaktion T^i wird eindeutig eine Zeitmarke $ts(T^i)$ zugeordnet. Zumeist ist dies der BOT-Zeitpunkt. Dann bestimmen sich die Prioritäten wie folgt:

$$P(T^i) = \frac{1}{ts(T^i)}$$

Alte Transaktionen haben demnach eine höhere Priorität als jüngere Transaktionen.

WAIT-DIE und WOUND-WAIT-Strategien

Auf der Basis der Prioritäten bzw. Zeitstempel lassen sich nun Strategien zur Deadlock-Vermeidung definieren ("Wild West"-Methoden).

Wenn eine Transaktion T^k eine Sperranforderung nicht gewährt bekommt, weil eine andere Transaktion T^i eine dazu inkompatible Sperre hält, dann wird je nach Prioritäten wie folgt verfahren:

WAIT-DIE:
- $P(T^k) > P(T^i)$: T^k wartet
- Sonst (also $P(T^k) < P(T^i)$): T^k stirbt (wird zurückgesetzt)

T^k wartet also auf jüngere Transaktionen.

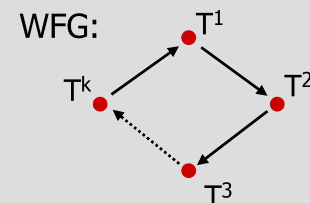
WOUND-WAIT:
- $P(T^k) > P(T^i)$: T^i wird "verwundet", muss zurückgesetzt werden
- Sonst (also $P(T^k) < P(T^i)$): T^k wartet

T^k wartet also auf ältere Transaktionen und "schiesst" jüngere Transaktionen ab

Deadlock-Vermeidung in W-D und W-W

Satz DLV: Sowohl durch WAIT-DIE als auch durch WOUND-WAIT werden Deadlocks vermieden.

Beweis: Annahme: der WFG eines Schedules, der durch WAIT-DIE (W-D) bzw. durch WOUND-WAIT (W-W) entstanden ist, enthalte einen Zyklus $T^1 \rightarrow T^2 \rightarrow \ldots \rightarrow T^k \rightarrow T^1$



Eine Kante $T^k \rightarrow T^i$ bedeutet "T^k wartet auf T^i" und
- $P(T^k) > P(T^i)$ bei WAIT-DIE
- $P(T^k) < P(T^i)$ bei WOUND-WAIT

Der Zyklus im WFG bedeutet damit:
- $P(T^1) > P(T^2) > \ldots > P(T^k) > P(T^1)$ und damit also $P(T^1) > P(T^1)$ bei W-D
- $P(T^1) < P(T^2) < \ldots < P(T^k) < P(T^1)$ und damit also $P(T^1) < P(T^1)$ bei W-W

In beiden Fällen führt dies daher zu einem Widerspruch

$\Rightarrow$ Zyklen im WFG können nicht auftreten

Eigenschaften von W-D bzw. W-W

Beide Verfahren sind unnötig restriktiv, da man bei jeder nicht erfüllten Sperranforderung einen Deadlock annimmt (dies muss aber nicht immer der Fall sein) und gleich eine der beiden involvierten Transaktionen zurücksetzt.

In keiner der beiden Strategien wird die ältere der beiden konfligierenden Transaktionen zurückgesetzt.

WAIT-DIE: Je älter eine Transaktion wird, desto grösser ist die Wahrscheinlichkeit für Warten!

Verhindern von unendlichem Warten:

- Eine freiwerdende Sperre wird immer an die wartende Transaktion mit höchster Priorität vergeben

WOUND-WAIT: Das Rücksetzen von jüngeren Transaktionen kann zu Starvation (Verhungern) führen: eine Transaktion wird abgebrochen, neu gestartet, wegen derselben Sperrkonflikte wieder abgebrochen, usw. Dieses Phänomen wird zuweilen auch als Livelock bezeichnet.

Vermeidung von Starvation:

- Eine zurückgesetzte Transaktion behält beim Neustart ihre ursprüngliche Zeitmarke.

3.4 Zeitstempel-Verfahren (TO)

Bisher haben wir Zeitstempel nur für die Vermeidung von Deadlocks (und für die Auswahl von Rücksetz-Opfern) verwendet. Die Idee des Zeitstempel-Verfahrens (Timestamp Ordering, TO) ist nun, gänzlich auf Sperren zu verzichten und Konfliktaktionen gemäss der Zeitstempel der jeweiligen Transaktionen zu ordnen.

Voraussetzung: Jeder Transaktion T^i wird eine eindeutige Zeitmarke (Zeitstempel) $ts(T^i)$ zugeordnet.

Auch hier muss das $ts(T^i)$ Element einer streng monotonen Zahlenfolge sein. Häufig wird die Systemuhr für die Bestimmung der $ts(T^i)$ zum BOT-Zeitpunkt verwendet.

TO-Regel: Für jedes Konfliktpaar $p^i(x)$ und $q^k(x)$ mit $i \neq k$ muss gelten:

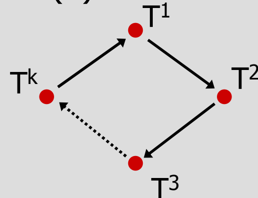
$$p^i(x) < q^k(x) \quad \Leftrightarrow \quad ts(T^i) < ts(T^k)$$

TO und CPSR

Satz TO: Ein Schedule, der unter Einhaltung der TO-Regel entsteht, ist serialisierbar (CPSR).

Beweis:

SG(S):



Annahme: Ein durch TO entstandener Schedule S enthalte einen Zyklus im Serialisierungsgraphen SG(S):

$T^1 \rightarrow T^2 \rightarrow \ldots \rightarrow T^k \rightarrow T^1$

Da S durch das TO-Protokoll entstanden ist, muss für die Zeitstempel der zugehörigen Transaktionen gelten:

$ts(T^1) < ts(T^2) < \ldots < ts(T^k) < ts(T^1)$

Widerspruch!

Bei einem Zyklus in SG kann die TO-Regel also nicht eingehalten worden sein.

Realisierung eines TO-Schedulers

Voraussetzung: Wir betrachten das Read/Write-Modell, Aktionen sind also r(x) bzw. w(x)

Für jedes Objekt x der DB gibt es:

- max_w_ts(x) grösste Zeitmarke einer Transaktion, die x geschrieben hat
- max_r_ts(x) grösste Zeitmarke einer Transaktion, die x gelesen hat

Für jede aktive Transaktion T^i wird gesetzt:

- Commit (T^i) false zu Beginn der Transaktion; true nach Commit
- WS(T^i) Write-Set der Transaktion T^i. Dies ist die Menge aller Objekte, die T^i bisher geschrieben hat

Arbeitsweise eines TO-Schedulers

T^i möchte Objekt x lesen:

- falls $ts(T^i) < max_w_ts(x)$

  T^i benötigt einen Wert, der bereits veraltet ist
  (also eine "Version", die schon überschrieben wurde)

  ⇒ T^i muss zurückgesetzt werden

- sonst (also $ts(T^i) \geq max_w_ts(x)$):

  Lesen erlaubt. Aktualisiere max_r_ts(x), falls $ts(T^i)$ grösser ist als der bisherige Wert

T^i möchte Objekt x schreiben

- falls $ts(T^i) < max_r_ts(x)$

  Es gibt eine jüngere Transaktion, die x bereits gelesen hat. Schreiben ist daher nicht erlaubt

  ⇒ T^i muss zurückgesetzt werden

- falls $ts(T^i) < max_w_ts(x)$

  Schreiben unnötig! (Wert wurde bereits wieder überschrieben)

- sonst (also $ts(T^i) \geq max_r_ts(x) \wedge ts(T^i) \geq max_w_ts(x)$)

  Schreiben erlaubt; aktualisiere max_w_ts(x)

Problem: was passiert, wenn andere Transaktionen bereits von T^i gelesen haben (und vielleicht sogar schon Commit durchgeführt haben)?

Beispielausführung eines TO-Schedulers

Beispiel: $T^1 = \langle r^1(a)\ r^1(c)\ C^1 \rangle$

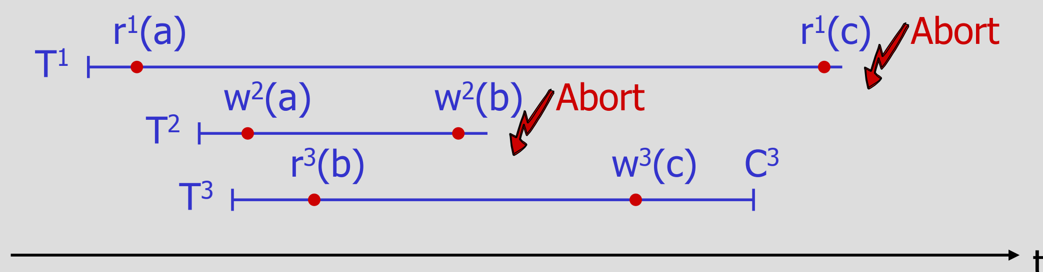
$T^2 = \langle w^2(a)\ w^2(b)\ C^2 \rangle$

$T^3 = \langle r^3(b)\ w^3(c)\ C^3 \rangle$

Mit dem Eingabeschedule

$ES = \langle r^1(a)\ w^2(a)\ r^3(b)\ w^2(b)\ C^2\ w^3(c)\ C^3\ r^1(c)\ C^1 \rangle$

Es gilt: $ts(T^1) < ts(T^2) < ts(T^3)$



3.5 Serialisierungsgraph-Test (SGT)

Grundlage des Serialisierungsgraph-Test-Verfahrens (SGT) ist die naheliegende Idee, dass der Scheduler einen Serialisierungsgraphen pflegt und sicherstellt, dass dieser azyklisch bleibt.

Dadurch ist dann sofort garantiert, dass jeder SGT-Schedule CPSR ist!

SGT-Algorithmus:

Voraussetzung: zu jedem Zeitpunkt wird benötigt

- $\mathcal{A}^k$: die Menge der Aktionen aller aktiven und korrekt abgeschlossenen Transaktionen T^k
- SG: der bislang aufgebaute Serialisierungsgraph

Aktion p^i von T^i steht zur Ausführung an

- Falls T^i noch nicht in SG
  - Füge T^i in SG ein
- Für alle T^k, also für alle aktiven und committeten Transaktionen
  - Füge Kante $T^k \rightarrow T^i$ in SG ein, wenn ein $\mathcal{A}^k$ eine Aktion q^k enthält, die mit p^i in Konflikt ist.
- Prüfe, ob SG azyklisch ist
- Falls ja, führe p^i aus und füge p^i in $\mathcal{A}^i$ ein, sonst: setze T^i zurück

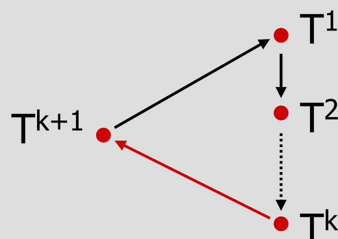
Grösse des SG bei SGT

- Ein Problem beim SGT-Verfahren ist, dass der Serialisierungsgraph immer grösser wird
  - Wann können Knoten sicher eliminiert werden?
- Naive Lösungsvermutung: Eine Transaktion T^k kann nach dem Commit entfernt werden
  - Diese Annahme ist allerdings falsch (wie das folgende Beispiel zeigt)

Beispiel:

$S = \langle r^{k+1}(x)\ w^1(x)\ w^1(y_1)\ C^1\ w^2(y_1)\ w^2(y_2)\ C^2 \ldots w^k(y_{k-1})\ w^k(y_k)\ C^k\ w^{k+1}(y_k)\ C^{k+1} \rangle$

SG(S):



Zyklus wird erst nach dem Commit von $T^1 \ldots T^k$ erkannt.

Entfernen von Transaktionen aus SGT

Das Beispiel zeigt, dass im Prinzip beliebig viele abgeschlossene Transaktionen im Serialisierungsgraph behalten werden müssen.

Sicher richtiges und korrektes Entfernen von Transaktionen aus SG:
T^k kann entfernt werden, wenn

1) T^k abgeschlossen ist und
2) T^k Quelle in SG ist

Wenn T^k abgeschlossen ist, dann können keine Kanten nach T^k gerichtet hinzukommen. Wenn T^k zudem noch Quelle ist, dann kann T^k nicht mehr in einen Zyklus verwickelt werden, darf also aus SG entfernt werden.

SGT wird heute in Datenbanken praktisch nicht verwendet. Das Problem des Verfahrens ist, dass bei kurzen r/w-Transaktionen der Aufwand recht gross ist, die $\mathcal{A}^k$-Mengen zu verwalten (zudem ist für r/w-Transaktionen die Verwendung von Sperren einfacher und auch intuitiver). Allerdings besitzt SGT den Vorteil, sehr generisch zu sein (im Gegensatz zu den Sperrverfahren, da S- bzw. X-Sperren sehr stark auf der Semantik von Lese- und Schreibaktionen basieren).
Daher ist das SGT-Verfahren für semantisch reiche Aktionen in komplexeren Umgebungen durchaus sinnvoll.

Kapitel 4: Optimistische Protokolle

Die Besonderheit optimistischer Verfahren ist das uneingeschränkte Zulassen von Aktionen, verbunden mit einem Korrektheitstest vor dem Commit. Hybride Verfahren wie das OSL verwenden Sperren und führen zusätzlich eine Validierung durch.

4.1 Validierende Verfahren
- Backward-oriented Concurrency Control (BOCC)
- Forward-oriented Concurrency Control (FOCC)

4.2 Ordered Shared Locking (OSL)

4.1 Validierende Verfahren

Grundlegende Idee der optimistischen Verfahren ist die Annahme, dass Konflikte selten auftreten. Daher führt man die Aktionen von Transaktionen zunächst ohne Einschränkung aus mit dem bewussten Risiko, im Zweifelsfall rücksetzen zu müssen.

Beispiel: Produktkataloge, in denen 99% aller Transaktionen nur lesen (Preise und/oder Produktbeschreibungen) und in denen recht selten Updates vorgenommen werden.

Natürlich muss vor dem Ende der Transaktion eine Überprüfung stattfinden. Daher werden drei Phasen einer Transaktion T^k unterschieden:

1. Lesephase: T^k liest Daten und schreibt (falls nicht reiner Leser) zunächst private Versionen. Private Versionen sind für andere Transaktionen nicht sichtbar. Wichtig dabei ist, dass am Ende dieser Phase das Read-Set RS^k und das Write-Set WS^k bekannt sind (Menge der Objekte, die gelesen bzw. geschrieben wurden).
2. Validierungsphase: Vor dem eigentlichen Schreiben wird überprüft, ob die Übernahme der bislang privaten Versionen in die DB gestattet werden kann (Konfliktanalyse).
3. Schreibphase: Falls die Validierung positiv war, werden alle privaten Versionen geschrieben; bei negativer Validierung: Zurücksetzen!

Kritische Phase: Validierung & Schreiben

Essentiell für die Korrektheit des Verfahrens ist die Voraussetzung, dass Validierungsphase und Schreibphase zusammen als unteilbare "kritische Phase" ausgeführt werden. Das bedeutet, dass in der Zeit, in der eine Transaktion T^i in der kritischen Phase ist, keine andere Transaktion T^k validieren oder schreiben darf.

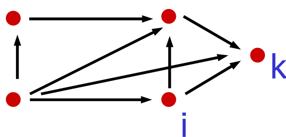
Diese starken Einschränkungen sind möglich, weil beide Phasen, Validierung und Schreiben, eher kurz sind. Die eigentliche Abarbeitung der Transaktion erfolgt in der Lesephase.

Validierungsphase

Das Ziel der Validierungsphase ist klar das Erzwingen eines azyklischen Serialisierungsgraphen, also eines korrekten (CPSR-) Schedules.

Obwohl im Fehlerfall (bei negativer Validierung) Transaktionen abgebrochen werden müssen ist dies unproblematisch, da alle noch nicht abgeschlossenen Transaktionen bis zu diesem Zeitpunkt nur private Versionen geschrieben haben, also keinen Einfluss auf andere Transaktionen besitzen (Rücksetzen ist isoliert).

Grundlage der Validierung ist die Tatsache, dass ein azyklischer gerichteter Graph G azyklisch bleibt, wenn beim Einfügen eines neuen Knotens k keine Kanten (k, i) von diesem ausgehen.



Wenn G bereits azyklisch ist, dann kann ein Zyklus nur entstehen, beim Einfügen eines Knotens k, wenn k in diesen Zyklus involviert ist. Dies ist aber nicht möglich, wenn von k keine Kanten ausgehen.

Diese Eigenschaft wird von optimistischen Verfahren angewandt: Die Validierung betrachtet (durch die kritische Phase) nur eine einzige Transaktion. Validierungsregeln müssen dann sicherstellen, dass von dieser Transaktion keine Kanten im Serialisierungsgraphen ausgehen, dass dieser also azyklisch bleibt.

Rückwärts orientierte Validierung

In der backward-oriented Concurrency Control (BOCC) validiert eine Transaktion T^k rückwärts gerichtet gegen alle bisher bereits korrekt abgeschlossenen Transaktionen. Dabei bedeutet validieren, dass ein Konflikttest vorgenommen wird.

Für jede Transaktion T^i wird dabei benötigt:

- RS^i: Menge aller gelesener Objekte (Read-Set)
- WS^i: Menge aller geschriebener Objekte (Write-Set)

BOCC-Validierung:

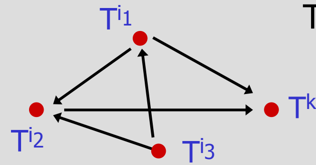
Eine Transaktion T^k wird im BOCC-Verfahren zugelassen ("positiv validiert"), wenn für alle bisher zugelassenen Transaktionen T^i gilt: entweder

1. T^i wurde beendet bevor T^k anfing (wenn also $T^i < T^k$ seriell ausgeführt werden), oder
2. $RS^k \cap WS^i = \varnothing$ (falls T^i und T^k parallel ausgeführt wurden)
   T^k darf also kein Objekt gelesen haben, das von T^i geschrieben wurde.

Korrektheit des BOCC-Verfahrens ...

Satz: Die BOCC-Validierung erzeugt einen azyklischen Serialisierungs-graphen (und daher CPSR-Schedules)

Beweisüberlegung: Man zeigt, dass für jede validierende (und damit schreibende) Transaktion T^k folgendes gilt:



falls Kanten zwischen einer Transaktion T^{im} und T^k existieren, dann sind diese nach T^k hin orientiert

Fall 1: (T^{im} und T^k seriell)

T^{im} Validierung + Schreiben

T^k Val+Write

Kante $T^k \rightarrow T^{im}$ ist nicht möglich, da T^k erst nach Beendigung von T^{im} beginnt.

Fall 2: (T^{im} und T^k parallel)

T^{im} Val+Write

T^k Val+Write

a) $R^k(x) \rightarrow W^{im}(x)$ unmöglich wegen $RS^k \cap WS^{im} = \emptyset$

b) $W^k(x) \rightarrow W^{im}(x)$ } unmöglich, da Schreibphase von T^k

c) $W^k(x) \rightarrow R^{im}(x)$ } erst nach Commit von T^{im} beginnt

Der Fall: T^{im} / T^k ist ausgeschlossen, da nur jeweils eine Transaktion in der kritischen Phase sein darf.

... Korrektheit des BOCC-Verfahrens

BOCC garantiert nicht nur, dass die entstehenden Schedules CPSR sind, sondern auch, dass COPSR eingehalten wird.

- Konflikte werden nur in der Reihenfolge zugelassen, in der Transaktionen ihre Validierung durchführen (die Validierungsreihenfolge stimmt daher auch mit der Serialisierungsordnung überein).
- Da das Commit zu dieser kritischen Phase gehört, sind zwangsläufig alle Konflikte in der Commit-Reihenfolge geordnet, also COPSR erfüllt!

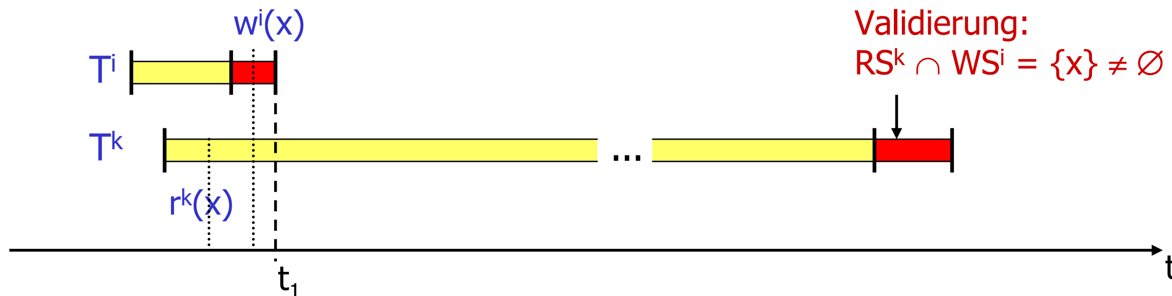
Negative Validierung im BOCC-Verfahren

Im Falle einer Verletzung der BOCC-Validierungsregel ist eine Strategie benötigt, die angibt, wie dieses Problem zu lösen ist.

Dabei ist zu beachten, dass alle Transaktionen T^i, gegen die eine Transaktion T^k validiert, abgeschlossen sind.

$\Rightarrow$ die einzige Möglichkeit besteht also im Rücksetzen von T^k

Allerdings wird die Notwendigkeit zum Rücksetzen erst in der kritischen Phase einer Transaktion, also sehr spät, erkannt:



Eine (lange) Transaktion T^k wird erst an deren Ende zurückgesetzt, obwohl sinnloses Weiterarbeiten schon zum Zeitpunkt t_1 erkennbar gewesen wäre!

Vorwärts orientierte Validierung

In der forward-oriented Concurrency Control (FOCC) validiert eine Transaktion T^k vorwärts gerichtet gegen alle parallel laufenden Transaktionen T^i. Durch die exklusive Betrachtung der kritischen Phase bedeutet dies, dass alle T^i noch in der Lesephase (und damit vor der Validierung) sind.

Für jede Transaktion T^i wird dabei benötigt:

- $RS^i(t)$: Menge aller gelesener Objekte der Transaktion T^i zum Zeitpunkt t (dies muss nicht vollständig sein, da die Lesephase noch nicht beendet ist)
- WS^i: Menge aller geschriebener Objekte (zum Zeitpunkt der Validierung)

FOCC-Validierung:

Eine Transaktion T^k wird im FOCC-Verfahren zum Zeitpunkt t zugelassen ("positiv validiert"), wenn für alle Transaktionen T^i, die zum Zeitpunkt t in der Lesephase sind, gilt:

- $WS^k \cap RS^i(t) = \emptyset$
  T^k darf also kein Objekt schreiben, das von einem T^i bisher bereits gelesen wurde.

Eine Konsequenz aus dem FOCC-Verfahren ist, dass reine Leser T^k immer positiv validiert werden, da $WS^k = \emptyset$

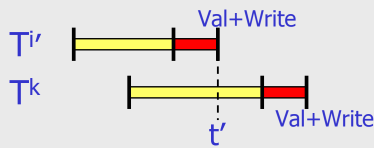
Korrektheit des FOCC-Verfahrens ...

Satz: Die FOCC-Validierung erzeugt einen azyklischen Serialisierungsgraphen (und daher CPSR-Schedules)

Beweisüberlegung: Man zeigt wieder, dass beim Einfügen (korrekten Validieren) von T^k kein Zyklus im Serialisierungsgraphen entsteht:

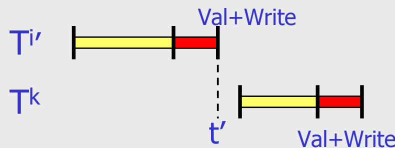
Sei $T^{i'}$ eine Transaktion, die vor T^k zum Zeitpunkt t′ positiv validierte (daher existiert ein Knoten $T^{i'}$ im bisherigen Serialisierungsgraphen).

Fall 1:
(T^k war zur Zeit t′ in Lesephase)



Da $T^{i'}$ korrekt validiert wurde, gilt zum Zeitpunkt t′ gemäss den FOCC-Regeln: $WS^{i'} \cap RS^k(t') = \varnothing$ (d.h. es gibt keine Kante zwischen $T^{i'}$ und T^k)

Fall 2:
(T^k war noch nicht aktiv)

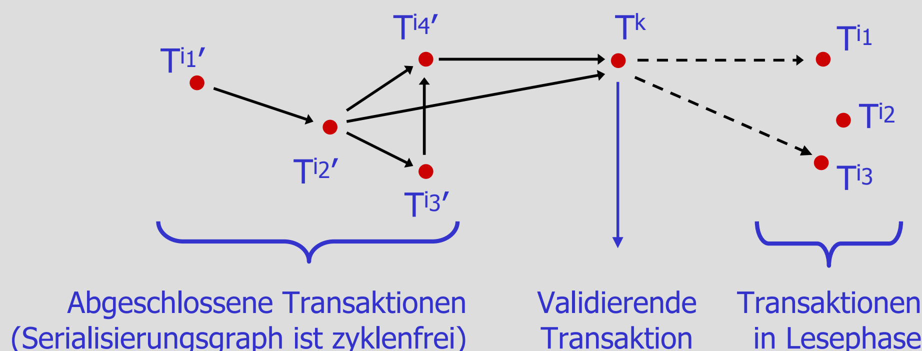


Da T^k zum Zeitpunkt t′ nicht aktiv ist, existiert auch kein Knoten T^k, und damit auch keine Kante (i′,k).

... Korrektheit des FOCC-Verfahrens

Also können nach dem Zeitpunkt t′, falls überhaupt, nur Kanten nach T^k hin orientiert eingeführt werden, nicht aber von T^k hin zu $T^{i'}$.
Zum Validierungszeitpunkt t von T^k können lediglich in folgenden Fällen Kanten zwischen T^k und $T^{i'}$ entstehen:

- falls $WS^{i'} \cap RS^k(t) \neq \varnothing$: Kante $T^{i'} \rightarrow T^k$
- falls $WS^{i'} \cap WS^k \neq \varnothing$ oder $RS^{i'} \cap WS^k \neq \varnothing$ entsprechend: Kante $T^{i'} \rightarrow T^k$



Abgeschlossene Transaktionen (Serialisierungsgraph ist zyklenfrei) | Validierende Transaktion | Transaktionen in Lesephase

Bei FOCC gilt ebenfalls, dass die entstehenden Schedules nicht nur CPSR, sondern auch COPSR sind.

Negative Validierung im FOCC-Verfahren

Im Gegensatz zum BOCC-Verfahren gibt es beim FOCC mehrere Möglichkeiten, eine Verletzung der Validierungsregel zu beheben:

1. Transaktion T^k wartet
   - Konflikt-Transaktion T^i, wegen der T^k nicht korrekt validieren kann, sei reine Lese-Transaktion:
     - T^k wartet ab, bis T^i fertig ist (schliesslich brauchen Leser nicht zu validieren bzw. validieren immer korrekt). Aber:
       - neue Lese-Transaktionen können hinzukommen oder bisherige Transaktionen lesen weitere Objekte
       - T^k wartet dann immer länger (es kann also zu einem Livelock kommen)
   - Konflikt-Transaktion T^i sei Schreiber
     - Im Gegensatz zum vorigen Fall ist hier nicht garantiert, dass T^i korrekt validiert (ausführlichere Diskussion siehe Übung)
2. T^i, wegen der T^k nicht korrekt validieren kann, wird zurückgesetzt
3. Transaktion T^k setzt selbst zurück

4.2 Ordered Shared Locks

2PL (und die diversen Varianten) garantieren CPSR, verhindern aber nicht, dass eigentlich korrekte Schedules nicht durchgelassen werden.

Der Grund hierfür ist, dass Konfliktaktionen bisher durch exklusive Sperren streng voneinander getrennt und damit geordnet werden.

- S-Sperre: Reihenfolge von Aktionen beliebig (es gibt auch keinen Konflikt zwischen zwei Lesern)
- X-Sperre: Falls T^k einen X-Lock auf einem Datenobjekt a hält, muss eine andere Transaktion T^i warten, bis T^k den X-Lock freigibt
  Konfliktaktionen werden seriell ausgeführt
  (nur eine Transaktion hat einen X-Lock zu einem Zeitpunkt)

Beispiel: $S = \langle w^1(x)\ r^2(x)\ r^3(y)\ C^3\ w^1(y)\ C^1\ r^2(y)\ C^2 \rangle$

S ist CPSR, aber nicht 2PL, da T^1 seine X-Sperre für T^2 freigeben muss, aber danach keine Sperre mehr für y erwerben kann.

Sperren mit Ordnung

Grundidee des Ordered Shared Locking:

Auch bei Konflikten Parallelität zulassen, aber dabei Ordnung der Sperranforderungen einhalten!

Dazu bedarf es allerdings weiterer Sperrmodi:

- S **S**hared: wie bisher (uneingeschränkte Parallelität)
- X e**X**klusiv: wie bisher
- OS **O**rdered **S**hared: eingeschränkte Parallelität
  - Sperren auf demselben Objekt können parallel gehalten werden, jedoch wird mit den Sperranforderungen eine Ordnung ⇒ verbunden: $L^i(x, OS) \Rightarrow L^k(x, OS)$ wenn T^i und T^k gleichzeitig Sperre auf x halten (T^i zuerst), also $L^i(x, OS) < \ldots < L^k(x, OS) < \ldots < U^i(x)$ ist möglich
  - Diese Ordnung muss auch bei der Ausführung der Konflikt-Aktionen (a^i, a^k) eingehalten werden, d.h. Konfliktpaare müssen in der Reihenfolge der Sperranforderungen ausgeführt werden: wenn $L^i(x, OS) \Rightarrow L^k(x, OS)$, dann muss auch $a^i(x) < a^k(x)$ gelten.

Verträglichkeitsmatrix

OS-Locks können als Alternative zu X-Locks für schreibende Zugriffe auf Objekte verwendet werden. Dies führt zu folgender Verträglichkeitsmatrix:

			Transaktion T^k fordert Lock zum		
			Lesen	**Schreiben**	
			Shared	Ordered Shared	eXclusive
Transaktion T^i hält Lock zum	**Lesen**	Shared	+	→	—
	Schreiben	Ordered Shared	→	→	—
		eXclusive	—	—	—

\+ Sperren kompatibel

— Sperren NICHT kompatibel

→ Sperren unter Einhaltung von Ordnung kompatibel

Ordered Shared Locking (OSL)

Die Sperrverträglichkeitsmatrix garantiert, dass jedes Konfliktpaar in der Sperr-Reihenfolge geordnet ist. Allerdings wird (noch) nicht garantiert, dass die Reihenfolge aller Konfliktpaare gleich ist.

Beispiel : $S = \langle w^1(x)\ r^2(x)\ w^2(y)\ r^1(y)\ C^2\ C^1 \rangle$
S ist nicht CPSR, wäre aber mit OS-Locks erlaubt, wenn man nicht wichtige Zusatzregeln hinzunimmt.

Eine Transaktion T^k hängt ab von T^i ("T^k has locks on hold"), wenn

- T^k einen OS-Lock hält und T^i vorher eine Sperre (S oder OS) auf demselben Objekt erworben, aber noch nicht freigegeben hat, oder wenn
- T^k einen S-Lock hält und vorher von T^i ein noch nicht freigegebener OS-Lock erworben wurde.

OSL-Regeln: Ein Scheduler hält das OSL-Protokoll ein, wenn
– Alle Transaktionen die Zweiphasigkeit des 2PL respektieren
– Eine abhängige Transaktion keine Locks freigibt

Die wichtige zweite Regel heisst mit anderen Worten: Falls T^k ein Commit durchführen möchte, müssen zuvor alle Transaktionen, von denen T^k abhängt, mit Commit (oder Abort) abgeschlossen sein.

OSL: Mögliche Protokolle in Kurzform

Die OSL-Regeln besagen lediglich, dass Zweiphasigkeit eingehalten wird, dass Schreiber entweder einen S- oder einen OS-Lock anfordern, und dass Abhängigkeiten bei der Sperrfreigabe beachtet werden.

Offen ist jetzt noch, wann S- und wann OS gewählt werden kann.

Dies führt zu einer Familie von insgesamt 8 verschiedenen Protokollen, $P_1 – P_8$, wobei P_8 das Gebräuchlichste des Ordered Shared Lockings ist.

P_1	r	w
r	S	X
w	X	X

= 2PL

P_2	r	w
r	S	X
w	OS	X

P_3	r	w
r	S	OS
w	X	X

P_4	r	w
r	S	X
w	X	OS

P_5	r	w
r	S	OS
w	OS	X

P_6	r	w
r	S	OS
w	X	OS

P_7	r	w
r	S	X
w	OS	OS

P_8	r	w
r	S	OS
w	OS	OS

OSL: Beispiele

Beispiel 1:

$S_1 = \langle w^1(x)\ r^2(x)\ r^3(y)\ C^3\ w^1(y)\ C^1\ r^2(y)\ C^2 \rangle$

Bei OSL-Protokoll P_8 sieht S_1 mit Sperranforderungen wie folgt aus:

P_8	r	w
r	S	OS
w	OS	OS

$L^1(x,OS)\ w^1(x)\ L^2(x,S)\ r^2(x)\ L^3(y,S)\ r^3(y)\ U^3(y)\ C^3\ L^1(y,OS)\ w^1(y)\ U^1(x,y)\ C^1$
$L^2(y,S)\ r^2(y)\ U^2(x,y)\ C^2$

T^1 ist nicht abhängig

T^2 ist ebenfalls nicht abhängig

Beispiel 2: (ebenfalls unter Verwendung von P_8)

$S_2 = \langle w^1(x)\ r^2(x)\ r^3(y)\ C^3\ w^1(y)\ r^2(y)\ C^2\ C^1 \rangle$

$L^1(x,OS)\ w^1(x)\ L^2(x,S)\ r^2(x)\ L^3(y,S)\ r^3(y)\ U^3(y)\ C^3\ L^1(y,OS)\ w^1(y)$

$L^2(y,S)\ r^2(y)\ U^2(x,y)$

Nicht möglich! Da T^2 von T^1 abhängig ist, muss T^2 auf die Beendigung von T^1 warten

OSL: Korrektheit

Satz OSL: Alle OSL-Protokolle (P_1-P_8) garantieren CPSR.

Beweis-Skizze:

Annahme: Schedule S, durch OSL erzeugt (mit einem der acht Protokolle P_1-P_8), besitze einen Zyklus $T^1 \rightarrow T^2 \rightarrow \dots \rightarrow T^k \rightarrow T^1$ im Serialisierungsgraphen SG(S).

Bei der Verwendung von OS-Locks bedeutet $T^i \rightarrow T^k$, dass T^k von T^i abhängig ist, und damit dass $U^i < U^k$ erfolgen muss. Also müsste durch den Zyklus gelten: T^1 ist abhängig von T^1 und damit auch $U^1 < U^1$, was jedoch zu einem Widerspruch führt.

(Zusätzlich benötigt: der Fall ohne OS-Locks. Aber dies entspricht genau der Argumentation, wie wir sie bereits beim "klassischen" 2PL geführt haben.)

OSL-Protokoll P_8, das gänzlich ohne X-Locks auskommt, besitzt zusätzlich die Eigenschaft, ordnungserhaltende Serialkisierbarkeit (OPSR) zu garantieren.

Wird OSL-Protokoll P_8 schliesslich noch erweitert durch die Forderung von strikter Zweiphasigkeit, dann wird auch commit-odnungserhaltende Serialisierbarkeit (COPSR) garantiert (das strikt zweiphasige P_8-Protokoll erlaubt sogar alle überhaupt möglichen COPSR-Schedules).

OSL: Eigenschaften

Offensichtlich kombiniert OSL (zumindest in den Varianten P_2-P_7) das pessimistische Locking (X-Locks) mit einer zusätzlichen Validierung, besitzt also gleichzeitig konservative und optimistische Elemente.
OSL-Protokoll P_1 hingegen betrachtet keine Validierung, ist also konservativ (klar, da P_1 gleich dem 2PL ist).
OSL-Protokoll P_8 schliesslich verwendet keine konservativen X-Locks sondern nur OS-Sperren ist damit ein rein optimistisches Verfahren.

Bei allen OSL-Protokollen ist unendliches Warten möglich (bei P_1 bzw. P_8 aus einem der folgenden Gründe, bei P_2-P_7 können beide auftreten):

- Bei der X-Lock-Anforderung wird eine Transaktion in den Wartezustand versetzt, wenn eine andere Transaktion bereits eine Sperre hält.
- Bei der Commit-Durchführung wird eine Transaktion in den Wartezustand versetzt, wenn eine andere Transaktion davor ihr Commit noch nicht durchgeführt hat.

Es muss bei OSL mit häufigem Rücksetzen von Transaktionen gerechnet werden; bei Protokollen, die OS-Locks für das Lesen von nicht verbindlichem Schreiben erlauben, sogar kaskadierend. Durch den grossen Verwaltungsaufwand für die Sperr-Ordnungen hat OSL keine Verbreitung in kommerziellen Datenbanksystemen gefunden. Allerdings spielt das Protokoll in komplexeren Anwendungen eine wichtige Rolle (siehe z.B. transaktionale [Geschäfts-]Prozesse).

Teil II – Concurrency Control

Kapitel 2: Serialisierbarkeitstheorie – Korrektheitskriterien
Kapitel 3: Konservative Concurrency Control-Protokolle
Kapitel 4: Optimistische Concurrency Control-Protokolle
Kapitel 5: Mehrversionen-Concurrency Control
Kapitel 6: Mehrversionen-Protokolle

Kapitel 5 – Mehrversionen-CC

Bisher sind wir immer von der Grundannahme ausgegangen, dass jedes Datenobjekt x nur in einer Version vorliegt. Die Konsequenz daraus war, dass durch jedes Schreiben der Wert von x überschrieben wurde und das dadurch Lesern nur der jeweils jüngste Wert von x verfügbar war.

Eine Abkehr von diesen Grundannahmen sieht für jedes Datenobjekt x mehrere Versionen vor (jeder Schreiber erzeugt eine neue Version, alte Versionen bleiben erhalten). Damit besitzen Leser die zusätzliche Freiheit, eine passende Version zu lesen (gemäss der Serialisierungsordnung), sind also nicht mehr auf die jüngste Version angewiesen. Dies führt zur Mehrversionen-Concurrency Control (MVCC).

MVCC ist von grosser praktischer Relevanz (es gibt kommerzielle Systeme, die diesen Ansatz verfolgen), da eine Art Versionierung von den meisten Systemen zu Recovery-Zwecken (siehe Teil III) sowieso durchgeführt werden muss.

MVCC: Annahmen & Eigenschaften

Idee: Jedes Schreiben eines Objekts x durch Transaktion T^k erzeugt eine neue Version x_k von x.

Dabei werden wir aber einschränkend annehmen, dass jede Transaktion T^k höchstens einmal x_k schreibt, d.h. T^k erzeugt zwischen BOT^k und EOT^k nur ein x_k, auch wenn x von T^k mehrfach geändert wird.

Notation: $w^k(x_k)$: T^k schreibt Version x_k
(dabei ist $w^k(x_i)$ für $i \neq k$ nicht definiert)
Für jedes Datenobjekt x nehmen wir an, dass eine Initialversion x_0 existiert.

Vorteil: Mit mehreren Versionen gibt es keine w/w-Konflikte und auch keine r/w-Konflikte:

- Falls T^k Version x_k schreibt und T^i auch x ändert, dann schreibt T^i x_i, also eine andere Version
- $r^m(x_k)$ kann nicht vor $w^k(x_k)$ stattfinden;
  daher gibt es keine r/w, höchstens w/r-Konflikte

⇒ Wahrscheinlichkeit für Konflikte grob um Faktor 1/3 reduziert (!), also höhere Parallelität.

MVCC – Motivation

Beispiel: $T^1 = \langle r^1_1(x)\ w^1_2(x)\ w^1_3(z)\ r^1_4(y)\ C^1 \rangle$

$T^2 = \langle r^2_1(x)\ w^2_2(y)\ w^2_3(z)\ C^2 \rangle$

Gegeben sei der (unvollständige) Schedule S über T^1 und T^2:

$S = \langle r^1_1(x)\ \underbrace{w^1_2(x)\ r^2_1(x)}\ w^1_3(z)\ w^2_2(y) \rangle$ mit: SG(S): $T^1 \rightarrow T^2$

Fortsetzung von S:

a) Im konventionellen Modell (jedes Datenobjekt in einer Version vorhanden): S_{1V} besitzt zyklischen Serialisierungsgraph, da zu $(w^1_2(x)\ r^2_1(x))$ auch noch das Konfliktpaar $(w^2_2(y),\ r^1_4(y))$ kommt.

$SG(S_{1V})$: $T^1 \rightleftarrows T^2$

b) Im Mehrversionen-Modell (jedes Schreiben erzeugt neue Version): Der Scheduler hat jetzt die Möglichkeit, $r^1_4(y)$ eine alte, aber "passende" Version zuzuweisen, die Ausführung ist jetzt also korrekt:

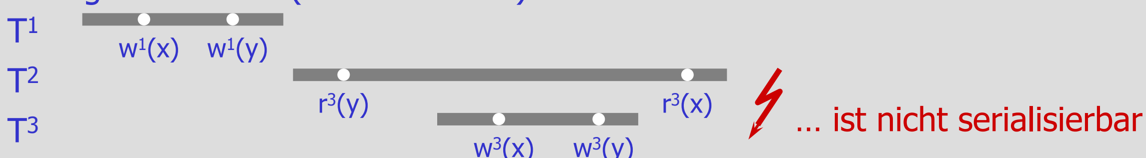
$S_{MV} = \langle r^1_1(x_0)\ \underbrace{w^1_2(x_1)\ r^2_1(x_1)}\ w^1_3(z_1)\ \underbrace{w^2_2(y_2)\ w^2_3(z_2)\ r^1_4(y_0)}_{/\!/}\ C^1\ C^2 \rangle$

MVCC: Versionenzuordnung

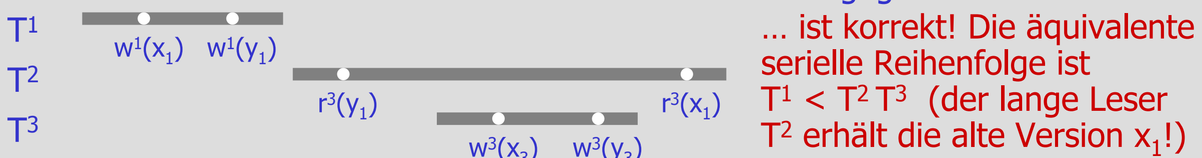
Aktionen von Transaktionen (von Benutzern oder Anwendungsprogrammen an die DB geschickt) beziehen sich auf Objekte, nicht auf Versionen. Die Aufgabe des Systems ist es nun, zu jeder Lese-Aktion $r^k(x)$ die zu lesende Version x_i so zu bestimmen, dass $w^i(x_i)$ vor $r^k(x)$ ausgeführt wurde, genauer, dass Serialisierbarkeit garantiert wird. Dies wird auch als Versionenzuordnung (Versionenfunktion) bezeichnet und muss transparent für den Benutzer / das Anwendungsprogramm erfolgen, bietet dem System aber einen zusätzlichen Freiheitsgrad. Schedules, in denen jeder einzelnen Aktion eine Version eines Datenobjektes zugewiesen ist, werden als Mehrversionen-Schedules bezeichnet.

Beispiel:

Der "gewöhnliche" (Ein-Versionen-) Schedule …



Die Transformation in einen Mehrversionen-Schedule hingegen …



… ist korrekt! Die äquivalente serielle Reihenfolge ist $T^1 < T^2\ T^3$ (der lange Leser T^2 erhält die alte Version x_1!)

Lösen mehrere Versionen alle Probleme?

Ist jeder Schedule durch die Verfügbarkeit mehrerer Versionen korrekt?

Beispiel: $T^1 = \langle r^1_1(x)\ w^1_2(x)\ C^1 \rangle$
$T^2 = \langle r^2_1(x)\ w^2_2(x)\ C^2 \rangle$

Gegeben sei der (unvollständige) Schedule S über T^1 und T^2 mit der Versionenzuordnung: $r^1_1(x) \rightarrow r^1_1(x_0)$ sowie $r^2_1(x) \rightarrow r^2_1(x_0)$

$S = \langle r^1_1(x_0)\ r^2_1(x_0) \rangle$

Fortsetzung von S:

Folgende zwei Möglichkeiten für einen vollständigen Schedule über T^1 und T^2 mit Präfix S′ existieren:

$S' = \langle r^1_1(x_0)\ r^2_1(x_0)\ w^1_2(x_1)\ w^2_2(x_2) \rangle$ bzw.
$S'' = \langle r^1_1(x_0)\ r^2_1(x_0)\ w^2_2(x_2)\ w^1_2(x_1) \rangle$

Für die beiden möglichen seriellen Ausführungen $\langle T^1 < T^2 \rangle$ bzw. $\langle T^2 < T^1 \rangle$ gilt:
$\langle T^1 < T^2 \rangle$: $r^2_1(x)$ muss x_1 lesen!
$\langle T^2 < T^1 \rangle$: $r^1_1(x)$ muss x_2 lesen!
} S ist also "nicht zu retten"

Mehrversionen-View-Serialisierbarkeit

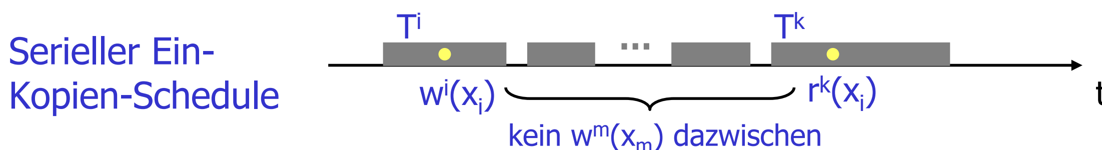
Eine wichtige Erkenntnis bei der Mehrversionen-Concurrency Control ist, dass ein serieller Mehrversionenschedule ist nicht notwendigerweise korrekt ist, sondern nur der Ein-Kopien-Schedule!

Beispiel: Schedule S mit $S = \langle r^1_1(x_0)\ w^1_2(x_1)\ C^1\ r^2_1(x_0)\ w^2_2(x_2)\ C^2 \rangle$ ist ein serieller Mehrversionen-Schedule, aber nicht korrekt!

Serieller Ein-Kopien-Schedule:

Ein serieller Ein-Kopien-Schedule ist ein serieller Schedule, in dem jeder Leser jeweils vom jüngsten Schreiber liest.

Serieller Ein-Kopien-Schedule



Mehrversionen-View-Serialisierbarkeit (MVSR)

Ein Mehrversionen-Schedule ist mehrversionen-view-serialisierbar, wenn seine Reads-From-Relation identisch zu einem seriellen Ein-Kopien-Schedule ist. Die Klasse dieser Schedules heisst MVSR.

MV-Serialisierungsgraph

Unter der Versionenordnung eines Objektes $x \in DB$ wird eine totale Ordnung aller in S über τ und DB erzeugten Versionen von x verstanden. Die Vereinigung aller Versionenordnungen der Objekte aus DB heisst Versionenordnung von S und wird mit $\lhd$ bezeichnet. Allerdings ist diese Versionenordnung nicht unbedingt gleich der zeitlichen Reihenfolge, in der die Versionen erzeugt werden!

MV-Serialisierungsgraph:

Sei S ein Mehrversionen-Schedule über τ und DB und $\lhd$ eine Versionenordnung zu S. Der MV-Serialisierungsgraph MVSG(S, $\lhd$) zu S und $\lhd$ ist ein Graph für den folgendes gilt:

Knoten: Jede Transaktion $T \in \tau$ ist ein Knoten

Kanten:
1. Falls $w^i(x_i)$ und $r^k(x_i)$ aus S mit $w^i(x_i) < r^k(x_i)$, dann enthält MVSG(S, $\lhd$) eine Kante $T^i \rightarrow T^k$ (liest-von-Beziehung, Kante analog einem w/r-Konflikt im gewöhnlichen Serialisierungsgraphen)
2. Falls $w^i(x_i)$ und $r^k(x_m)$ $(i \neq m \neq k)$ aus S und $x_i \lhd x_m$, dann enthält MVSG(S) die Kante $T^i \rightarrow T^m$;
   sonst (also bei $x_m \lhd x_i$) eine Kante $T^k \rightarrow T^i$.
   (Versionenkanten)

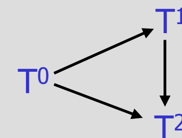
Eigenschaften von MVSG

Die Zyklenfreiheit des Serialisierungsgraphen, der nur aus den "liest-von-Abhängigkeiten" besteht, der also allein durch Regel 1 gebildet wird, reicht für MVSR nicht aus!

Beispiel:

$S = \langle w^0(x_0)\ w^0(y_0)\ C^0\ r^1(x_0)\ r^1(y_0)\ w^1(x_1)\ w^1(y_1)\ C^1\ r^2(x_0)\ r^2(y_1)\ C^2 \rangle$

S ist serieller MV-Schedule mit azyklischem SG (bezüglich der liest-von-Abhängigkeiten):



Allerdings ist S nicht äquivalent zu einem seriellen Ein-Versionen-Schedule

Die beiden seriellen Ein-Versionen-Schedules S' bzw. S'' sind:

$S' = \langle w^0(x)\ w^0(y)\ C^0\ r^1(x)\ r^1(y)\ w^1(x)\ w^1(y)\ C^1\ r^2(x)\ r^2(y)\ C^2 \rangle$
$S'' = \langle w^0(x)\ w^0(y)\ C^0\ r^2(x)\ r^2(y)\ C^2\ r^1(x)\ r^1(y)\ w^1(x)\ w^1(y)\ C^1 \rangle$

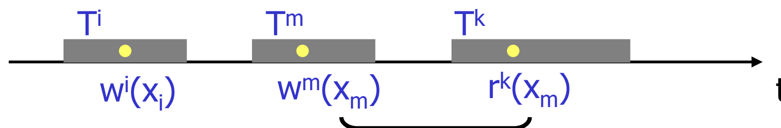
beide besitzen jedoch unterschiedliche RF-Relationen!

MVSG und Versionenkanten

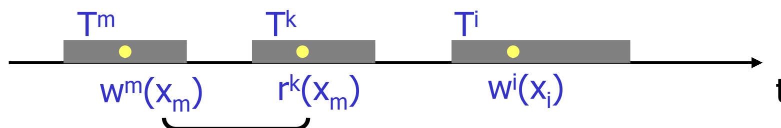
Da die liest-von-Abhängigkeiten alleine nicht ausreichen, müssen durch Regel 2 weitere Kanten (Versionenkanten) in den MVSG eingeführt werden, so dass auch die Reihenfolge der Versionen berücksichtigt wird.

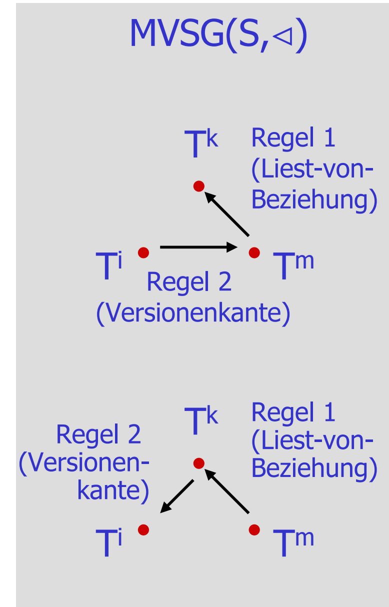
Für jedes Paar $w^i(x_i)$ und $r^k(x_m)$ gilt daher:

- $x_i \lhd x_m$ führt zu Kante $T^i \rightarrow T^m$ und erzwingt damit $w^i(x_i)$ vor $w^m(x_m)$ in S_{ser}



- $x_m \lhd x_i$ führt zu Kante $T^k \rightarrow T^i$ und erzwingt damit $w^i(x_i)$ nach $r^k(x_m)$ in S_{ser}

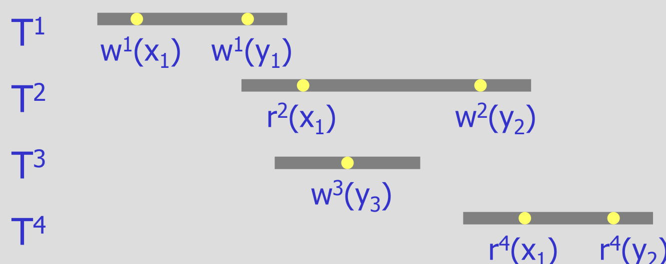




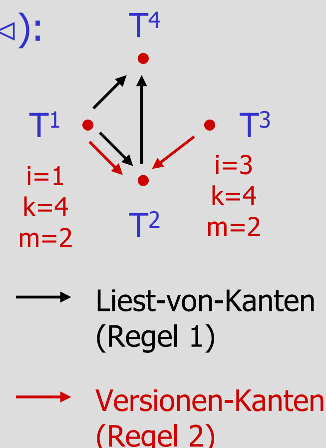
Korrektheitstest von MVSR

Satz: Ein MV-Schedule S ist MVSR $\Leftrightarrow$ es gibt eine Versionenordnung $\lhd$ zu S, so dass der MV-Serialisierungsgraph zu S und $\lhd$ azyklisch ist.

Beispiel: Gegeben sei folgender Schedule über den Transaktionen T^1-T^4 sowie die Versionenordnung $y_1 \lhd y_3 \lhd y_2$



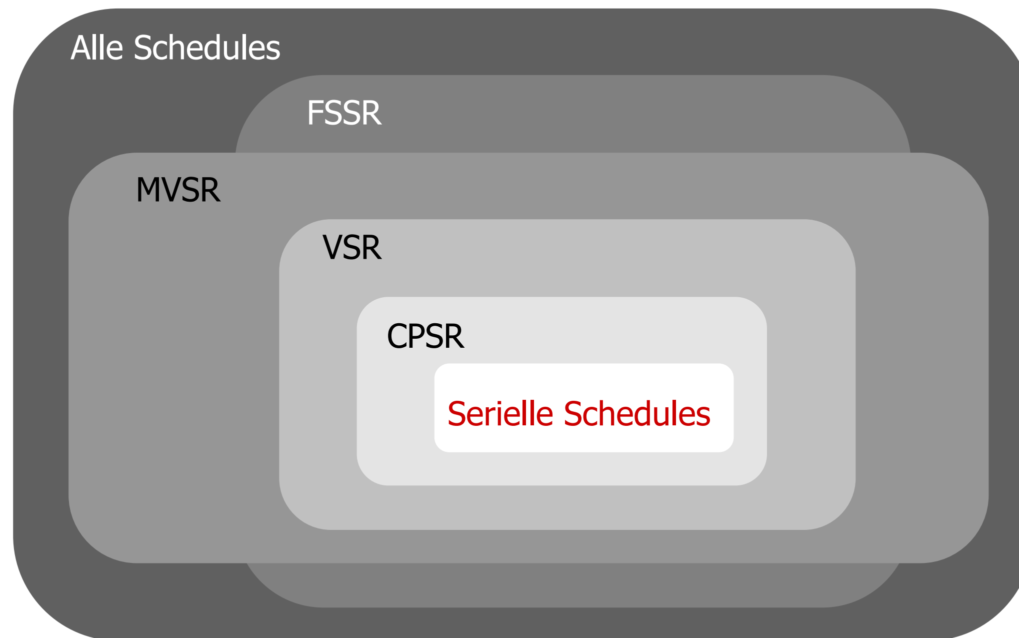
MVSG(S,⊲):



Da MVSG(S,⊲) azyklisch ist, ist S MVSR. Äquivalente serielle Ein-Kopien-Schedules sind:
$\langle T^1 < T^3 < T^2 < T^4 \rangle$ oder $\langle T^3 < T^1 < T^2 < T^4 \rangle$

Einordnung von MVSR

MVSR enthält die Klasse der view-serialisierbaren Schedules VSR
(und damit auch die Klasse CPSR der konflikt-serialisierbaren Schedules),
ist aber nicht vergleichbar mit FSSR:



Kapitel 6 – Mehrversionen-Verfahren

In der theoretischen Betrachtung geht man bei der Mehrversionen-Concurrency Control davon aus, dass jedes Schreiben eine neue Version erzeugt und dass eine unbegrenzte Anzahl Versionen pro Datenobjekt verfügbar ist.
In der Praxis schränken Mehrversionen-Verfahren dies zumeist ein, indem Versionen nur bei Bedarf erzeugt werden oder indem die Anzahl Versionen pro Objekt streng limitiert ist. Alle Verfahren besitzen jedoch die Eigenschaft, dass die Versionierung nach aussen hin, also zum Benutzer bzw. Anwendungsprogramm transparent ist (d.h. Benutzer- bzw. Anwendungsprogramm-Aktionen beinhalten immer noch Objekte, nicht Versionen).

Unbegrenzte Anzahl Versionen:

6.1 Mehrversionen-Timestamp-Ordering (MVTO)

Versionen werden "on demand" erstellt:

6.2 Snapshot Isolation (SSI)

Streng begrenzte Anzahl Versionen

6.3 Zwei-Versionen-2PL (2V2PL)

6.1 Mehrversionen-TO (MVTO)

Im "klassischen" TO-Verfahren muss eine Transaktion T^i zurückgesetzt werden, wenn

1. T^i durch eine Schreib-Aktion ein Objekt ändern will, das bereits von einer jüngeren Transaktion gelesen wurde, oder
2. T^i für eine Lese-Aktion ein Objekt benötigen würde, das bereits überschrieben wurde.

Wenn das TO-Verfahren zum Mehrversionen-Timestamp-Ordering (MVTO) erweitert wird, so dass Datenobjekte in mehreren Versionen vorliegen und jedes Schreiben eine neue Version generiert, dann lässt sich zumindest Fall 2 beheben. Anstelle eines Abbruchs von T^i muss dem Leser einfach die passende Version zugewiesen werden.

Die prinzipielle Arbeitsweise von MVTO basiert daher auf dem "klassischen" TO-Verfahren, mit der folgenden Erweiterung:

– es muss nicht mehr pro Objekt die grösste Zeitmarke max_w_ts(x) der Transaktion, die x geschrieben hat, verwaltet werden (es gibt durch die Versionierung ja nur noch einen Schreiber für x_k so dass der zugehörige Zeitstempel $ts(T^k)$ bekannt ist)

MVTO: Protokoll

T^i möchte Objekt x lesen:

– Es wird dem Leser $r^i(x)$ die Version x_k zugewiesen, so dass für den Zeitstempel $ts(T^k)$ des Schreibers von x_k gilt: $ts(T^k)$ ist der grösste Zeitstempel mit $ts(T^k) < ts(T^i)$

⇒ Leser werden nie zurückgesetzt!

T^i möchte Objekt x schreiben (also neue Version x_i produzieren):

– Falls eine Aktion $r^k(x_m)$ mit $ts(T^m) < ts(T^i) < ts(T^k)$ bereits ausgeführt wurde, dann wird T^i zurückgesetzt (T^i würde eine Version produzieren, die ein jüngerer Leser hätte lesen müssen)

– Sonst (es gibt kein solches $r^k(x_m)$): $w^i(x_i)$ ist erlaubt, T^i erzeugt also eine Objektversion mit Zeitstempel $ts(T^i)$.

Satz MVTO:	Ein Schedule, der durch das MVTO-Protokoll entsteht, ist mehrversionen-view-serialisierbar (MVSR).

Die –zumindest in der theoretischen Betrachtung existierende– noch frei wählbare Versionenordnung ist daher im MVTO durch die Zeitstempel-Reihenfolge festgelegt.

6.2 Snapshot Isolation (SSI)

Bisher haben wir für jede Transaktion T^k nur die Aktionen $r^k(x)$ bzw $w^k(x)$ sowie die Beendigung von T^k betrachtet (entweder durch Commit C^k oder Abort A^k)

Für die Formalisierung der Snapshot Isolation benötigen wir jetzt zusätzlich noch folgende Informationen für jede Transaktion T^k

- Read-Set $RS(T^k)$: Menge der Datenobjekte, welche von T^k gelesen werden
- Write-Set $WS(T^k)$: Menge der Datenobjekte, welche von T^k geschrieben werden
- Start von T^k: BOT^k (Begin-of-Transaction)

In den folgenden Beispielschedules werden wir den Beginn BOT^k einer Transaktion T^k nicht explizit einfügen. Die erste Aktion von T^k soll gleichzeitig auch der Start der Transaktion sein. Später (im praktischen Teil) werden wir dann noch eine andere Alternative für BOT kennen lernen.

Definition der Snapshot-Isolation

Idee: Jede Transaktion T^k erhält eine eigene Version (Snapshot) der Datenbank beim Transaktionsbeginn zugewiesen. Dieser Snapshot ist für die komplette Lebensdauer von T^k gültig.

Snapshot Isolation (SSI):

Ein Mehrversionen-Schedule über einer Menge τ von Transaktionen erfüllt das Kriterium der Snapshot-Isolation (SSI), wenn die folgenden Bedingungen eingehalten werden:

(SSI-V) $\forall\, r^k(x_i)$ gilt: $w^i(x_i) < C^i < BOT^k < r^k(x_i)$ und
$\nexists\, w^m(x_m)$ mit: $w^m(x_m) < BOT^k$ und $C^i < C^m < BOT^k$

(SSI-W) $\forall\, T^i, T^k$ mit $(BOT^i < BOT^k < C^i) \vee$
$(BOT^k < BOT^i < C^k)$
gilt: $WS(T^i) \cap WS(T^k) = \emptyset$

SSI-V: Ordnet einem Leser r^k immer den jüngsten, zum Startzeitpunkt von T^k committeten Wert zu (= Versionenzuordnung)

SSI-W: Je zwei parallele Transaktionen müssen paarweise disjunkte Write-Sets besitzen

SSI – Beispiele

Beispiel 1: $S_1 = \langle r^1_1(x)\ r^2_1(y)\ w^1_2(x)\ r^2_2(x)\ C^2\ w^1_3(y)\ C^1 \rangle$

Mehrversionenschedule: $S'_1 = \langle r^1_1(x_0)\ r^2_1(y_0)\ w^1_2(x_1)\ r^2_2(x_0)\ C^2\ w^1_3(y_1)\ C^1 \rangle$

Sei: $x_0 \lhd x_1$ und $y_0 \lhd y_1$

Test auf MVSR: MVSG azyklisch $\Rightarrow S'_1 \in$ MVSR

Test auf SSI: (SSI-W): $WS(T^1) = \{x,y\}$, $WS(T^2) = \emptyset$

$\Rightarrow WS(T^1) \cap WS(T^2) = \emptyset$

(SSI-V): ebenfalls erfüllt $\Rightarrow S'_1 \in$ SSI

Beispiel 2: $S_2 = \langle r^1_1(x)\ r^2_1(y)\ w^2_2(y)\ C^2\ r^1_2(y)\ w^1_3(y)\ C^1 \rangle$

Mehrversionenschedule: $S'_2 = \langle r^1_1(x_0)\ r^2_1(y_0)\ w^2_2(y_2)\ C^2\ r^1_2(y_2)\ w^1_3(y_1)\ C^1 \rangle$

Sei: $y_0 \lhd y_2 \lhd y_1$

Test auf MVSR: MVSG azyklisch $\Rightarrow S'_2 \in$ MVSR

view-äquivalenter Ein-Kopien-Schedule ist $\langle T^0 < T^2 < T^1 \rangle$

Test auf SSI: (SSI-W): NICHT erfüllt, da $WS(T^1) = WS(T^2) = \{y\}$

(SSI-V): ebenfalls nicht erfüllt. $\Rightarrow S'_2 \notin$ SSI.

Versionenzuordnung in SSI

Die Wahl der Versionenzuordnung bot bei der Mehrversionen-View-Serialisierbarkeit einen zusätzlichen Freiheitsgrad und die Möglichkeit, Lesern geeignete Versionen eines Datenobjektes zuzuordnen.

Bei SSI ist die freie Wahl der Versionenzuordnung jedoch nicht möglich (da Vorgabe durch SSI-V).

Zusätzlich ist die Versionenordnung $\lhd_{SSI}$ durch SSI-V und SSI-W bereits fest vorgegeben:

Versionenordnung $\lhd_{SSI}$

Aus SSI-V und SSI-W folgt für die Versionenordnung $\lhd_{SSI}$:

$$x_i \lhd_{SSI} x_k \Leftrightarrow C^i < C^k$$

SSI – Beispiele (Fortsetzung)

Beispiel 3: $S_3 = \langle r^1_1(x)\ r^2_1(y)\ r^2_2(x)\ r^1_2(y)\ w^2_3(x)\ w^1_3(y)\ C^1\ C^2 \rangle$

Mehrversionenschedule (einzige mögliche Versionenordnung) :

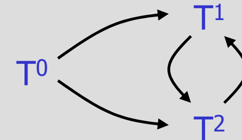
$S'_3 = \langle r^1_1(x_0)\ r^2_1(y_0)\ r^2_2(x_0)\ r^1_2(y_0)\ w^2_3(x_2)\ w^1_3(y_1)\ C^1\ C^2 \rangle$

Versionenordnung: $x_0 \lhd x_2$ und $y_0 \lhd y_1$

Test auf SSI: (SSI-W): $WS(T^1) = \{y\}$, $WS(T^2) = \{x\}$

$\Rightarrow WS(T^1) \cap WS(T^2) = \emptyset$

(SSI-V): ebenfalls erfüllt $\Rightarrow S'_3 \in SSI$



Test auf MVSR: $MVSG(S'_3, \lhd)$ enthält Zyklus

$\Rightarrow S'_3 \notin MVSR$!!!

⇒ SSI erlaubt also Schedules, die –entgegen unserer ursprünglichen Annahme– keine äquivalente serielle (Ein-Kopien-)Ausführung besitzen !!!

Praktisches Beispiel ...

Relationen

```
Konto    (KontoNr, Saldo, KontoInhaber),
Zins     (KontoNr, Zinssatz)
```

Transaktion T^1 (Zinsbestimmung):

- Lese Kontostand
- Falls 5000 < Kontostand < 10000 erhöhe Zins um 1%, falls Kontostand > 10000 erhöhe Zins um 2%

Transaktion T^2 (Zinsauszahlung):

- Lese kontospezifischen Zinssatz
- Buche Zins auf Konto

Initialwert für KontoNr 4711:

- Kontostand = 9999
- Zins = 3%

... Praktisches Beispiel ...

Transaktion T^1:

```
> Select Saldo from Konto
  where KontoNr = 4711;
> Select Zinssatz from
  Zins where KontoNr=4711;
> Update Zins Set Zinssatz
  = Zinssatz+1 where
  KontoNr = 4711;
> commit;
```

Transaktion T^2:

```
> Select Zinssatz from Zins
  where KontoNr = 4711;
> Select Saldo from Konto
  where KontoNr = 4711;
> Update Konto set Saldo =
  Saldo*Zinssatz where
  KontoNr = 4711;
> commit;
```

t ↓

$$S_3 = \langle r^1_1(x)\ r^2_1(y)\ r^2_2(x)\ r^1_2(y)\ w^2_3(x)\ w^1_3(y)\ C^1\ C^2 \rangle$$

... Praktisches Beispiel

- S_3 ist nicht MVSR (es existiert nur eine mögliche Versionenzuordnung; für diese existiert Zyklus im MV-Serialisierungsgraphen)
- Allerdings ist S_3 "korrekt" bezüglich SSI da sowohl (SSI-V) als auch (SSI-W) erfüllt sind!
- Beide möglichen seriellen (und damit korrekten) Abläufe liefern jedoch andere Ergebnisse (entweder höheren Zinssatz oder höheren Kontostand)
  - $T^1 \rightarrow T^2$: Saldo = 10399 und Zinssatz = 4%
  - $T^2 \rightarrow T^1$: Saldo = 10299 und Zinssatz = 5%
  - Hier jedoch: Saldo = 10299 und Zinssatz = 4%
- SSI wird in der Isolationsstufe `serializable` vom Datenbanksystem Oracle verwendet!

Test auf SSI

Der Test auf Mehrversionen-Sicht-Serialisierbarkeit (MVSR) lässt sich auf elegante Art und Weise mit Hilfe des MV-Serialisierungsgraphen MVSG durchführen.

Ist ein ähnliches Verfahren auch für SSI möglich?

- Unter der Einschränkung, dass keine Blind Writes erlaubt werden (d.h. $\forall T^i \in S: WS(T^i) \setminus RS(T^i) = \emptyset \wedge r^i(x_k) < w^i(x_i)$) ist dies möglich!
- Grundlage dieses graphisch orientierten Tests ist der SSI-Serialisierungsgraph (SSI-SG).
  SSI-SG entspricht dem MV-Serialisierungsgraphen mit der Ausnahme, dass alle Kanten den Bezeichner des jeweiligen Datenobjektes, das die Abhängigkeit verursacht, tragen.

SSI-Serialisierungsgraph

SSI-Serialisierungsgraph:

Sei S ein Mehrversionenschedule über τ und DB, der die Bedingung SSI-V erfüllt. Der SSI-Serialisierungsgraph SSI-SG(S, $\lhd_{SSI}$) zu S und $\lhd_{SSI}$ ist definiert durch

- Knoten: Jede Transaktion $T \in \tau$ ist ein Knoten
- Kanten: 1. Für jedes $r^k(x_i)$ in S existiert eine Kante $T^i \rightarrow T^k$ mit dem Bezeichner x (liest-von-Kante)
  2. Für jedes Paar $r^k(x_m)$ und $w^i(x_i)$ gibt es eine Kante
     - $T^i \rightarrow T^m$, mit Bezeichner x, falls $x_i \lhd_{SSI} x_m$
     - $T^k \rightarrow T^i$, mit Bezeichner x, falls $x_m \lhd_{SSI} x_i$

     (Versionen-Kanten)

Zyklenfreiheit von SSI-SG

Satz (x-Zyklenfreiheit des SSI-SG):

Ein Mehrversionenschedule, der SSI-V erfüllt und der keine Blind Writes beinhaltet ist in SSI genau dann, wenn es kein Objekt x gibt, für das ein x-Zyklus in SSI-SG existiert.

Beweis siehe R. Schenkel, G. Weikum, N. Weissenberg, X. Wu "Federated Transaction Management with Snapshot Isolation". In: Proceedings of the TDD'99 Workshop, Dagstuhl, Sept. 1999. http://www-dbs.cs.uni-sb.de/public_html/papers/tdd99.ps.Z

Das Verbot der Blind Writes ist in diesem Fall wichtig, denn es garantiert, dass durch die zusätzlichen Leseaktionen, die vor jedem Schreiben gefordert werden, durch Bedingung 2 des SSI-SG auch Kanten für w/w-Abhängigkeiten eingefügt werden.
Dadurch lässt sich dann SSI-W via x-Zyklenfreiheit überprüfen.

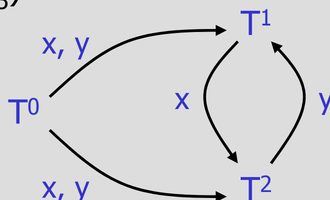
Zyklenfreiheit von SSI-SG – Beispiele

Beispiel 3 (revisited):

$S`_3 = \langle r^1_1(x_0)\ r^1_2(y_0)\ r^2_1(x_0)\ r^2_2(y_0)\ w^2_3(x_2)\ w^1_3(y_1)\ C^1\ C^2 \rangle$

Der zugehörige SSI-SG sieht folgendermassen aus:

SSI-SG($S`_3, \lhd_3$):



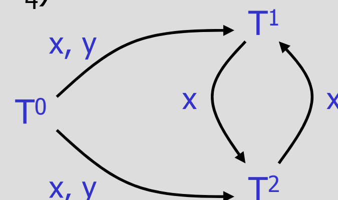
SSI-SG($S`_3, \lhd_3$) enthält weder x- noch y-Zyklen, daher SSI. Durch Zyklus ohne Labels folgt, wie bereits festgestellt, dass $S`_3 \notin$ MVSR.

Beispiel 4:

$S_4 = \langle r^1_1(x_0)\ r^1_2(y_0)\ r^2_1(x_0)\ r^2_2(y_0)\ w^1_3(x_1)\ w^2_3(x_2)\ C^1\ C^2 \rangle$

Der zugehörige SSI-SG sieht folgendermassen aus:

SSI-SG($S_4, \lhd_4$):



SSI-SG($S_4, \lhd_4$) enthält jetzt einen x-Zyklus. Das bedeutet, dass für x (SSI-W) verletzt ist.
$\Rightarrow S_4 \notin$ SSI.

Vergleich: SSI vs. MVSR

MVSR vs. SSI: Begründung siehe Beispiele 1-3

6.3 Zwei-Versionen-2PL (2V2PL)

Idee: Pro Datenobjekt existieren (höchstens) zwei Versionen: eine "gültige" Version (einer Transaktion, die mit Commit beendet ist) und evtl. zusätzlich eine "private" Version einer Transaktion, die noch aktiv ist (diese beiden Versionen werden ohnehin für Recovery-Zwecke benötigt). Da die gültige Version immer verfügbar ist, können Leser bevorzugt werden (Versionenauswahl).

2V2PL erlaubt, dass gleichzeitig gelesen und geschrieben werden kann (kein w/r und r/w-Konflikt), muss aber auch garantieren, dass pro Objekt nur eine nicht-committete Version existiert (w/w-Konflikt). Zudem wird eine Validierung (certify) aller privaten Versionen zum Commit-Zeitpunkt benötigt um zu überprüfen, wann private Objekte gültig gemacht werden.

Realisierung: 3 Lock-Modi

- Read (S)
- Write (X)
- Certify (CX)
  dieser Modus wird benötigt für alle privaten Versionen, erfordert also Konversion X → CX vor dem Commit.

Kompatibilitäts-matrix		Transaktion T^k fordert Lock an		
		Shared	eXclusive	Certify (CX)
T^i hält Lock	Shared	+	+	–
	eXclusive	+	–	–
	Certify (CX)	–	–	–

2V2PL: Protokoll

1. Transaktion T^k möchte x schreiben. X-Lock für $w^k(x)$ verfügbar?
   - Nein, falls andere Transaktion T^i zertifiziert oder schreibt (also X- oder CX-Lock hält)
     ⇒ T^k muss warten
   - Sonst (X-Lock wird gewährt):
     ⇒ Schreiben erlaubt, x_k wird durch $w^k(x_k)$ erzeugt

2. Transaktion T^k möchte x lesen. S-Lock für $r^k(x)$ verfügbar?
   - Nein, falls andere Transaktion T^i zertifiziert, also CX-Lock hält
     ⇒ T^k muss warten
   - Sonst (S-Lock wird gewährt):
     ⇒ Lesen erlaubt, T^k liest $r^k(x_m)$; x_m ist die gültige ("committed") Version von x

3. Transaktion T^k möchte Commit durchführen (C^k), d.h. die vorbereiteten Versionen x_k, y_k,... sollen in die DB übernommen werden. Können alle X-Locks in CX-Locks umgewandelt werden?
   - Nein, falls andere Transaktion T^i S-Sperre hält
     ⇒ T^k muss auf alle Leser von x, y, ... warten! Leser werden also bevorzugt!
   - Sonst (alle CX-Locks können gewährt werden):
     ⇒ T^k kann die gültigen Versionen von x_k, y_k, … erzeugen, die alten Versionen eliminieren und alle Sperren freigeben.

2V2PL: Eigenschaften

- Zusätzlich zu den Regeln (1.)-(3.) ist im 2V2PL Zweiphasigkeit gefordert (nach der ersten Sperrfreigabe darf keine neue Sperre mehr angefordert werden).
- 2V2PL ist nicht Deadlock-frei. Es müssen die bekannten Ansätze zur Deadlock-Auflösung angewandt werden.
- Während der Zertifizierung (Umwandlung von X-Locks in CX-Locks) ist auch "Verhungern" möglich (Sperren können wegen parallelen Lesern nicht umgewandelt werden; es kommen immer wieder neue Leser hinzu). Auch hier ist eine Zusatzbehandlung nötig.
- Allerdings erlaubt das 2V2PL im Vergleich zum herkömmlichen 2PL, exklusive (CX) Sperren nur kurz zu halten und somit andere Transaktionen weniger lange zu blockieren.

Satz: Jeder durch das 2V2PL-Protokoll entstandene Schedule ist mehrversionen-view-serialisierbar (MVSR).

2V2PL: Beispiel

Beispiel: Gegeben ist der folgende Eingabeschedule S_{ES}

$S_{ES} = \langle r^1{}_1(x)\ w^2{}_1(y)\ r^1{}_2(y)\ w^1{}_3(x)\ C^1\ r^3{}_1(y)\ r^3{}_2(z)\ w^3{}_3(z)\ w^2{}_2(x)\ C^2\ w^4{}_1(z)\ C^4\ C^3 \rangle$

Unter der Annahme, dass Initialversionen x_0, y_0, z_0 bereits vorhanden sind, erzeugt das 2V2PL folgenden Ausgabeschedule:

$S_{2V2PL} = \langle L^1(x,S)\ r^1{}_1(x_0)\ L^2(y,X)\ w^2{}_1(y_2)\ L^1(y,S)\ r^1{}_2(y_0)\ L^1(x,X)\ w^1{}_3(x_1)\ L^1(x,CX) \ldots$

Gemäss Kompatibilitäts-matrix des 2V2PL erlaubt! (zu $L^2(y,X)\ w^2{}_1(y_2)\ L^1(y,S)$)

Gültige Version (zu $r^1{}_2(y_0)$)

ok, da kein paralleler Leser von x existiert (zu $L^1(x,CX)$)

$\ldots U^1(x)\ U^1(y)\ C^1\ L^3(y,S)\ r^3{}_1(y_0)\ L^3(z,S)\ r^3{}_2(z_0)\ L^3(z,X)\ w^3{}_3(z_3)\ L^2(x,X)\ w^2{}_2(x_2) \ldots$

y_2 ist immer noch "privat" (zu $r^3{}_1(y_0)$)

C² muss verzögert werden: $L^2(y,CX)$ ist nicht möglich wegen $L^3(y,S)$

Nach $U^3(y)$ kann jetzt $L^2(y,CX)$ gewährt werden

$\ldots\ L^3(z,CX)\ U^3(y)\ U^3(z)\ C^3\ L^2(x,CX)\ L^2(y,CX)\ U^2(x)\ U^2(y)\ C^2 \ldots$

$w^4(z)$ muss verzögert werden, da $L^3(z,X)$ noch gehalten wird.

$\ldots L^4(z,X)\ w^4{}_1(z_4)\ L^4(z,CX)\ U^4(z)\ C^4 \rangle$