Zusammenfassung Transaktionsverwaltung (SS ’99)

Marc Schanne (Marc.Schanne@gmx.de)

1 Konsistenz, Interaktivität, Koordination

1.1 Begriffe

Transaktion: Informationsverarbeitender Leistungsprozess, der einem festen Ablaufmuster folgt und für deren Wirkungen nach innen und in die Realwelt hinein Qualitätszusicherungen gegeben werden.

Transaktionsmanager (TM): Konsistenz

Scheduler: garantiert Isolation durch Synchronisation

Data-Manager (DM): Recovery-Manager + Cache-Manager: Wiederherstellbarkeit, Dauerhaftigkeit

Recovery: Wiederherstellung der Konsistenz

Interaktion: Einflussnahme eines Prozesses auf das Fortschreiten eines anderen

Konkurrenz: Behinderung des Fortschreitens eines Prozesses duch einen anderen

Konflikt: Konkrete Behinderungssituation

Kooperation: Gegenseitiges Lenken des Fortschreitens

Koordination: Geordnetes Zusammenwirken mehrerer Ressourcen-Verwalter innerhalb desselben Leistungsprozesses (= Synchronisation)

Integration: engere Kopplung über gemeinsame Datenbestände

Kongruenz: Datenbasis ist kongruentes Abbild der Miniwelt. Idealvorstellung, nicht kontrollierbar!

1.2 Qualitäten von Transaktionen

1.2.1 Isolation

Unterbinden unerwünschter Interaktion zwischen nebenläufigen Transaktionen

1.2.2 Konsistenz

innere Konsistenz: vorausgesetzt

externe Konsistenz: Ressourcen Manager (Isolation, Recovery)

Datenbasis folgt stets einer Menge von Regeln, die denen der Miniwelt nachempfunden sind. Deskriptive oder prozedurale Verallgemeinerung: Kongruenz ist ein Spezialfall der der Konsistenz.
mehr Konsistenz oder mehr Nebenläufigkeit

1.2.3 Atomizität

Transaktion wirkt als ganzes oder garnicht nach aussen. Wirkung erst nach erfolgreichem Abschluss sichtbar.

1.2.4 Dauerhaftigkeit

Wirkung erfolgreich abgeschlossener Transaktionen bleibt sichtbar, es sei denn eine andere Transaktion macht sie explizit rückgangig.

1.3 Transaktionsklassen

Qualitätsklassen: Umfang der zugesicherten externen Konsistenz; Lauf der Transaktion; Aufwand für Durchsetzung der Konsistenz; Aufwand für Recovery; Interaktion

Einflussfaktoren: Dauer; Volumen der geänderten Daten; Störanfälligkeit; Art der Störungen; Möglichkeiten der Recovery, Kompensation & Korrektur von Prozessschritten; verwendeter Konsistenzbegriff; gewünschte Interaktionsmöglichkeit

1.3.1 ACID-Transaktionen (klassische Transaktion)

Eigenschaften: kurze Zeitdauer; geringes Datenvolumen; vollständiges Rückgängigmachen

Qualitäten: ACID: Atomizität; Konsistenz; Isolation; Dauerhaftigkeit

1.3.2 Konversationstransaktionen (VLSI Entwurf)

Eigenschaften: längere Dauer; hohes Datenvolumen; Korrekturnotwendigkeiten

Qualitäten: Konsistenz; Dauerhaftigkeit; Abschwächung der Atomizität (Einführung dauerhafter Sicherungspunkte, auf die zurückgesetzt werden kann)

1.3.3 Entwurfstransaktionen (Entwurf im Team)

Eigenschaften: lange Dauer; hohes Datenvolumen; Kooperation mehrerer Personen

Qualitäten: Konsistenz & Dauerhaftigkeit (ACID); Atomizität (Konv.trans); keine Isolation (Benutzer kooperieren und beeinflussen sich gegenseitig) => Aufspalten in ACID/ACI-Untertransaktionen

1.3.4 Stapeltransaktionen (Buchungssätze bei Platzbuchung)

Eigenschaften: lange Dauer, hohes Datenvolumen; gleichartige Behandlung grosser Datenmengen; zeitlich abgesetzte Ausführung

Qualitäten: Konsistenz & Dauerhaftigkeit (ACID); Problem Isolation: an Sicherungspunkten werden Daten, deren Bearbeitung abgeschlossen ist, freigegeben

1.4 Scheduler

Arten: aggressiv / optimistisch - konservativ / pessimistisch

Möglichkeiten:

1. sofortiges Ausführen (Weitergabe an Data-Manager)
2. Verzögern (Warteschlage)
3. Zurückweisen (der zugehörigen Transaktion)

2 Grundlagen der Graphentheorie

2.1 Ungerichteter Graph

$G = (N, E)$ mit N einer Menge von Knoten und $E \subseteq (N \times N)$ eine Menge von Kanten; Paare in E <u>ungeordnet</u>.

Eigenschaften:

Verbunden $\Leftrightarrow \forall\, v, v' \in N :\ \exists$ Pfad, der v und v' verbindet.
Teilgraph G' von $G \ \Leftrightarrow\ N' \subseteq N \bigwedge E' \subseteq E$
Pfad: endliche Folge $v_1, ..., v_n$ von Kanten, so dass $\forall\, 1 \leq k \leq n :\ [v_k, v_{k+1}] \in E$
Partitionierung: Famile G_i von Teilgraphen von G

2.2 Gerichteter Graph

$G = (N, E)$ mit N einer Menge von Knoten und $E \subseteq (N \times N)$ eine Menge von Kanten; Paare in E seien <u>geordnet</u>.

zusätzl. Eigenschaften:

Zyklus: Pfad mit $v_1 = v_n$ und $n > 1$
Vorgänger/Nachfolger: $(v, v') \in E$, so heißt v direkter Vorgänger von v'. Falls ein Pfad von v nach v' existiert heißt v Vorgänger von v' $(v \rightarrow^* v')$
Unabhängig $\Leftrightarrow v \not\rightarrow v' \bigwedge v' \not\rightarrow v$

2.3 Gerichteter azyklische Graph

Gerichteter Graph ohne Zyklen: directed acyclic graph (DAG)
Wurzel-Knoten $v \ \Leftrightarrow \nexists$ auf v gerichtete Kante
Topologische Ordnung $T(G)$ eines Graphen G ist die Folge $v_1, ..., v_n \in N$ aller Knoten aus N mit $i < j \Rightarrow \nexists$ Pfad von v_j nach v_i

2.4 Transitive Hülle

Graph $G^+ = (N, E^+)$ ist eine transitive Hülle eines Graphen $G = (N, E)$ gdw. $v \rightarrow_G^+ v' \ \Leftrightarrow v \rightarrow_G v'$

2.5 Partielle Ordnung

$L = (\sum, <)$ mit $\sum$ Menge und $<$ irreflexive, transitive binäre Relation $\sum \times \sum$

Einschränkung L' **einer partiellen Ordnung** L auf den Bereich Σ' :

$\Sigma' \subseteq \Sigma \ \wedge\ \forall a, b \in \Sigma' :\ a <' b \Leftrightarrow a < b$

Präfix L' **einer partiellen Ordnung** L $(L' \leq L)$:

L' ist Einschränkung von $L \wedge \forall a \in \Sigma' \forall b \in \Sigma :\ b < a \Rightarrow b \in \Sigma'$

3 Transaktionsmodell

3.1 (externe) Konsistenz

Verwendung eines einfachen R/W-Modells $\Rightarrow$ von außen sicherzustellender Konsistenzbegriff eingeschränkt:

Level 1: no lost update ($r_1[x]r_2[x]w_1[x]w_2[x]$ ... $w_1[x]$)

Level 2: Level 1 + no dirty reads ($r_1[x]w_1[x]r_2[x]a_1$ ... $r_2[x]$ illegitimer Wert x)

Level 3: Level 2 + repeatable reads ($r_1[x]r_2[x]w_2[x]r_1[x]$ ... $r_1[x]$ vor oder nach Änderung?)

3.2 Transaktionsbeschreibung

Definition (Bernstein, allgemeiner): Transaktion T_i ist eine partielle Ordnung $<_i$, wobei

1. $T_i \subseteq \{r_i[x],\, w_i[x] \mid x$ ist ein Datenelement $\} \cup \{a_i,\, c_i\}$
2. $a_i \in T_i$ gdw $c_i \notin T_i$
3. $\forall t \in T_i\, t = c_i \lor t = a_i \Rightarrow (\forall s \in T_i <_i t)$
4. $r_i[x],\, w_i[x] \in T_i \Rightarrow (r_i[x] <_i w_i[x] \lor w_i[x] <_i r_i[x])$

Sequentielle Transaktion: Transaktion mit totaler Ordnung (Definition Transaktion Vossen)

3.3 (Vollständige) Historie

Wenn eine Menge von Transaktionen ($T = \{T_1, \ldots, T_n\}$) parallel ausgeführt wird, so ist die Ausführung verschachtelt, Operationen verschiedener Transaktionen sind gemischt. Eine solche Ausführung nennen wie Historie.

Vollständige Historie ist eine partielle Ordnung $<_H$ mit:

1. $H = \bigcup_{i=1}^{n} T_i$

2. $\bigcup_{i=1}^{n} <_i \subseteq <_H$

3. $\forall p, q \in H :\, p \not\parallel q \Rightarrow p <_H q$ oder $q <_H p$

Historie ist ein Präfix einer vollständigen Historie.

3.4 Serielle Historie

Historie H über $T = \{T_1, \ldots, T_n\}$ heißt seriell :⇔
$\forall 1 \leq i, j \leq n : i \neq j \Rightarrow (\forall p_i, p_j : p_i <_H p_j) \vee (\forall p_i, p_j : p_j <_H p_i)$

3.5 Serialisierbarkeit

serialisierbar: bezügl. eines Verträglichkeitskriteriums äquivalent zu seriell.

Konfliktserialisierbarkeit, Sichten-Serialsierbarkeit, Final-State-Serialisierbarkeit

3.6 Korrekte Historien

- serielle Historien
- serialisierbare Historen ≡ serielle Historien

3.7 Strikte Historie (ST)

Historie H mit $\forall T_i, T_j \, (i \neq j) \in H : w_i[x] <_H p_j[x] \Rightarrow a_i <_H p_j[x] \vee c_i <_H p_j[x], p \in \{r, w\}$
Jede Historie, die dem strikten 2PL Protokoll gehorcht, ist strikt.

3.8 Vollständige Schedule (Full)

Vollständige Historie mit vollständiger Ordnung

3.9 Schedule

Präfix eines vollständigen Schedules

Jede serielle Historie ist ein Schedule (serieller Schedule)

$trans(S) := \{T_i \,|\, T_i \in S\} := \{T_i \,|\,$ S enthält Schritte von $T_i\}$

$commit(S) := \{T_i \,|\, c_i \in S\}$

$abort(S) := \{T_i \,|\, a_i \in S\}$

$active(S) := trans(S) - (commit(S) \vee abort(S))$

$op(S) := \{op \,|\, op \in S\}$ Menge aller in S vorkommenden Schritte

3.10 Lesen und Schreiben in Transaktion

liest von: T_j liest x von T_j ($T_j \rhd_H [x] T_i$)
* $\exists x : w_i[x] <_H r_j[x]$
* $a_i \not<_H r_j[x]$
* $\forall k \neq i, j : w_i[x] <_H w_k[x] <_H r_j[x] \Rightarrow a_k <_H r_j[x]$

letztes Schreiben in H von x:
Operation $w_i[x] \in H$ mit
* $a_i \in H$
* $\forall w_j[x] \in H : i \neq j \Rightarrow w_j[x] <_H w_i[x] \vee a_j \in H$
$w_i[x] = FIN_H[x]$

4 X-Serialisierbarkeit

4.1 HERBRAND-Semantik von Schedules (nicht Historien!)

Annahmen: Jedes Datenobjekt in jeder Transaktion wird höchstens einmal gelesen und/oder geschrieben, in keiner Transaktion nach schreiben nochmal gelesen. In jedem Schedule gibt es eine Datenbasis initialisierende Transaktion $T_0 = w_0[x_1] \dots w_0[x_n]\, c_0$

Definition:
$H_S(r_i[x]) := H_S(w_j[x]),\ j \neq i$ wobei $r_i[x] \rhd_S w_j[x]$
$H_S(w_i[x]) := f_{i,x}(H_S(r_i[y_1], \dots, H_S(r_i[y_m]))$ wobei $r_i[y_j]\,(1 \leq j \leq m)$ sämtliche Leseschritte von T_i in S vor $w_i[x]$ und $f_{i,x}$ ein uninterpretiertes m-stelliges Funktionssymbol.

HERBRAND-Universum (HU)...
... für Transaktionen $T_i,\ i > 0$ ist die kleinste Menge von Symbolen:
* $F_{0,x}() \in HU$ für jedes $x \in D$, wobei $f_{0,x}$ 0-stelliges Funktionssymbol
* Falls
- $w_i[x] \in op(T_i)$
- $|\{r_i[y] \mid \exists y \in D : r_i[y] <_{T_i} w_i[x]\}| = m$
- $v_1, \dots, v_m \in HU$
dann $f_{i,x}(v_1, \dots, v_m) \in HU$ wobei $f_{i,x}$ ein m-stelliges Funktionssymbol.

Semantik eines Schedules: $H(S) : D \rightarrow HU$
Def.: $H(S)(x) := H_S(FIN_S(x)))\ (x \in D)$
D.h. Menge derjenigen Werte, die von nicht-abgebrochenen transaktionen zuletzt geschrieben werden.

4.2 Unverträglichkeit

Zwei Operationen p und q sind unverträglich (kommutieren nicht), falls beide auf das gleiche Datenelement zugreifen und mindestens eine von ihnen ein Schreibzugriff ist.
Schreibweise:
$p \not\| q$: p und q stehen in Konflikt miteinander

4.3 Nützlichkeit

Operation p ist in S nützlich für Operation q ($p \rightsquigarrow_S q$), falls $q \triangleright_S p$ oder $p = r_i[x]$, $q = w_i[y]$ und $p <_S q$

4.4 Lebendigkeit

Operation p heißt lebendig in $S :\Leftrightarrow \exists q \in T_\infty\, p \rightsquigarrow^* q$ (sonst tot!)
mit T_∞ Transaktion zur Vervollständigung einer Historie, wobei T_∞ für alle in H auftauchende Datenelemente x die Operation $r_\infty[x]$ (per Definition lebendig) enthält mit $\forall p \in H,\, x \in D:\ p <_H r_\infty[x]$

4.5 Lebendig-Liest-von-Relation

$RF(S) := \{(T_i, x, T_j) \,|\, r_j[x] \triangleright_S w_i[x]\}$
$LRF(S) := \{(T_i, x, T_j) \,|\, r_j[x]$ lebendig, $r_j[x] \triangleright_S w_i[x]\}$
Schreibweise: $T_j \triangleright_S^L [x]\, T_i$

4.6 Final-State-Äquivalenz ($S \equiv_F S'$)

1. $op(S) = op(S')$ gleiche Operationen
2. $H[S] = H[S']$ gleicher Endzustand

Folgerungen/Sätze:

$op(S) = op(S')$
Final-State-Äquivalenz kann in polynominaler Zeit entschieden werden

FSR: Menge der final-state-serialisierbaren Schedule

4.7 Konflikt-Relation

$C(S) = \{(p,q) \mid p \not\parallel q,\, p <_S q\}$
Abgeschlossene Projektion: $CP(H) := H \mid \bigcup_{c_i \in H,\, i=1,\dots,n} T_i$
$conf$: $conf(S) := C(CP(S))$ um abgebrochene Transaktionen bereinigte Konfliktrelation. Nur die nicht abgebrochenen Transaktionen und deren Konflikte sind für die Serialisierbarket interessant.

4.8 Konflikt-Äquivalenz ($H \equiv_C H'$)

Gleiche Reihenfolge in Konflikt stehender Operationen wie in serieller Historie H'.

1. $trans(H) = trans(H')$
2. $op(H) = op(H')$
3. $\forall p_i \in T_i,\, p_j \in T_j$ mit $p_i \not\parallel p_j :\ a_i, a_j \notin H,\, p_i <_H p_j \Leftrightarrow p_i <_{H'} p_j$

4.9 Konflikt-Serialisierbarkeit

Historie H heist serialisierbar, falls eine serielle Historie H_S existiert, so dass $CP(H) \equiv H_S$
CSR Menge alle konfliktserialisierbaren Historien $\Leftrightarrow$ $conf(H) = conf(H_S)$

4.10 Serialisierbarkeitsgraph (SG)

$SG(H)$ ist ein gerichteter Graph (N, E) mit $N = \{T_i \mid T_i \in T,\, c_i \in H\}$ und
$E = \{T_i \rightarrow T_j \mid \exists p_i \in T_i,\, p_j \in T_j\, (i \neq j)\, p_i \not\parallel p_j \wedge p_i <_H p_j\}$

4.11 Serialisierbarkeitstheorem

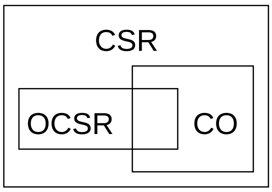
Eine Historie H ist serialsisierbar $\Leftrightarrow$ $SG(H)$ ist azyklisch.

4.12 Ordnungserhaltende konfliktserialisierbare Schedules

$\forall T, T' \in commit(S) : \forall p \in T : p <_S T' \Rightarrow p <_{S'} T'$ mit S vollständiger Schedule, ordnungserhaltend konfliktserislisierbar und S' serieller Schedule:
$OCSR \subset CSR$ echte Teilmenge: $SG(S) \notin OSCR$

4.13 Commit-Ordering

Schedule S ist geordnet bezgl. Freigabeoperation, falls gilt:
$\forall T_i, T_j \in commit(S), i \neq j, p \in T_i, q \in T_j : (p,q) \in conf(S) \Rightarrow c_i <_S c_j$ Reihenfolge von je zwei in Konflikt stehenden Operationen wie Reihenfolge der Commit-Operationen der zugehörigen Transaktionen.
$CO \subset CSR$ Echtheit der Inklusion: $S = r_1[x]w_2[x]c_2c_1 \in CSR \notin CO$



Beispiele für serielle Reihenfolge: $T_3T_1T_2$:
$S_1 = w_1[x]r_2[x]c_2w_3[y]c_3w_1[y]c_1 \in CSR \setminus (OCSR \cup CO)$
$S_2 = w_3[y]c_3w_1[x]r_2[x]c_2w_1[y]c_1 \in OCSR \setminus CO$
$S_3 = w_1[x]r_2[x]w_3[y]c_3w_1[y]c_1c_2 \in CO \setminus OCSR$
$S_4 = w_3[y]c_3w_1[x]r_2[x]w_1[y]c_1c_2 \in OCSR \cap CO$

4.14 Sichten-Äquivalenz ($H \equiv_V H'$)

$H \equiv_V H' :\Leftrightarrow$

1. T_i in H und H' liest von denselben Schreiboperationen
2. Letztes Schreiben in H und H' stimmen überein

Definition nach Bernstein:
1. $trans(H) = trans(H')$
2. $op(H) = op(H')$
3. $\forall T_i, T_j \in T : \forall x : a_i, a_j \notin H, T_i \triangleright_H [x]T_j \Leftrightarrow T_i \triangleright_{H'} [x]T_j$
4. $\forall x : FIN_H(x) = FIN_{H'}(x)$

alternativ: Definition nach Vossen:
1. $op(S) = op(S')$
2. $H(S) = H(S')$
vgl. Final-State-Äquivalenz! Verschärfung.
3. $H_S(p) = H_{S'}(p)$ für jede Operation p einer nicht abgebrochenen Transaktion

4.15 Sichtenserialisierbarkeit

Historie H heißt sichtenserialisierbar (VSR) $:\Leftrightarrow \forall H' \leq H \, \exists H_S \, seriell : CP(H') \equiv_V H_S$
d.h. jeder Präfix einer Historie muss zu einer seriellen Historie äquvalent sein.
$CSR \subset VSR \subset FSR$

4.16 Abschätzung O

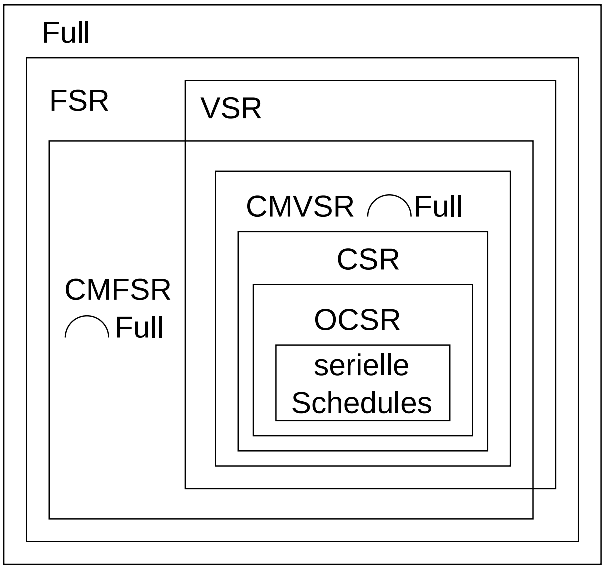
FSR-Nachweis: polynomer Zeitaufwand
CSR-Nachweis: polynominaler Zeitaufwand (Azyklität in gerichtetem Graph)
VSR-Nachweis: NP-vollständig (Azyklität in Polygraph (= Familie von gerichteten Graphen))

4.17 Präfix-Commit-Abgeschlossenheit (PCC)

$\forall H : H$ präfixabgeschlossen $\Rightarrow \forall H' \leq H : CP(H')$ präfixabgeschlossen.
CSR ist PCC: Wenn H serialisierbare Historie, dann ist für jedes Präfix H' von H $CP(H')$ serialisierbar. FSR und VSR (Vossen-Def.) nicht zwingend PCC. VSR (Bernstein-Def.) ist PCC.

4.18 Commit-serialisierbar (CM)

S heißt CM, falls $CP(S')$ serialisierbar für jeden Präfix S' von S
CSR ist CM
$CMFSR$: Klasse der commit-FSR Schedules
$CMVSR$: Klasse der commit-VSR Schedules
$CMCSR$: Klasse der commit-CSR Schedules
$CMCSR \subset CMVSR \subset CMFSR$ alle PCC
$CMFSR \cap Full \subset FSR$
$CMVSR \cap Full \subset VSR$
$CMCSR \cap Full = CSR$



5 Einversionen-Scheduler

5.1 Einfaches 2-Phasen-Sperrprotokoll (2PL)

- $rl[x]$ read lock (shared)
- $wl[x]$ write lock (exclusiv)

$pl_i[x] \not\parallel ql_j[y] \;:\Leftrightarrow\; x = y \wedge i \neq j \;\wedge p \not\parallel q$
Protokoll bei Eintreffen von $p_i[x]$:

1. Falls $p_i[x]$ nicht mit vergebenen Sperren in Konflikt wird $pl_i[x]$ an T_i vergeben und $p_i[x]$ ausgeführt, sonst $p_i[x]$ verzögert.
2. $\forall x$ bearbeitet von T_i $\exists$ Operation $rl_i[x]$ bzw. $wl_i[x]$ vor.
3. $pl_i[x]$ wird erst freigegeben wenn $p_i[x]$ ausgeführt. Alle Sperren werden bzw. sind zu Transaktionsende freigegeben.
4. Nachdem für Transaktion eine Sperre freigegeben keine mehr.

Sperrenvergabe mit "growing phase" und "shrinking phase".

Erzeugnis eines 2PL-Scheduler: $\varepsilon(2Pl) \subset OCSR$

5.2 Verklemmungen (einfaches 2PL)

Aufheben nur durch Abbruch einer an Verklemmung beteiligten Transaktion

Behandlung von Verklemmungen:

* Timeout: bevorzugt bei verteilten Systemen
Problem: vermutungsbasiert, geeignetes Timeoutintervall
* Wartegraph $WG = (N, E)$ mit $N = active(H),\ E = \{(T_i, T_j) \,|\, T_i$ wartet auf Sperrfreigabe von $T_j\} \Rightarrow$ Verklemmung bei Zyklus
Problem: Leistungseinbuße, geeignetes Testintervall

Auswahlkriterien (Wartegraph):

1. Bereits durch Transaktion entstandene Kosten
⇒ billige: nochmal laufen lassen
2. Kosten Transaktion zu Ende zu führen
⇒ fast zu Ende: durchziehen
3. Kosten Transaktion abzubrechen
⇒ Info über DM nötig
4. Anzahl der Zyklen, an denen Transaktion beteiligt
⇒ MAX(Anzahl Beteiligung an Zyklen)
5. Anzahl der durch Verklemmungen bedingte Abbrüche einer Transaktion
⇒ Fairness! nicht immer die selbe

5.3 (Ultra-) Konservatives 2PL

Verklemmungen werden vermieden, weil vor der ersten Operation alle Sperren der Transaktion angefordert werden.
- kaum Nebenläufigkeit
+ bessere Optimierungsmöglichkeiten (mehr Infos)
+ keine Deadlocks

5.4 Striktes 2PL

Fast alle realen Scheduler implementieren es.
(Schreib-)Sperren werden erst am Ende der Transaktion freigegeben. $\forall x,\, p[x] :\ a_i <_H pu_i[x] \vee c_i <_H pu_i[x]$
Gründe: schwer vorhersagbar ob Sperre nochmal benötigt wird.
Striktes 2PL ergibt strikte Historien
- geringe Nebenläufigkeit
+ einfache Scheduler möglich
+ Isolation

$\varepsilon(S2PL) \subseteq CSR \cap ST$

Wenn alle Sperren bis zum Ende der jeweiligen Transaktion gehalten werden: starker S2PL

$\varepsilon(SS2PL) \subset \varepsilon(S2PL) \subset \varepsilon(2PL)$
$\varepsilon(SS2PL) = RG$

5.5 Multigranularity Locking (MGL) Protokoll

Organisation: physische DB-Struktur:
(ganze) DB $\supset$ Segmente, Area, DB-Space $\supset$ Datei $\supset$ Seite $\supset$ Record $\supset$ Attribut

Sperren von der Wurzel aus!

1. Falls eine r- oder ir-Sperre benötigt wird für ein Granulat $\Rightarrow$ alle Vorgänger müssen ir oder iw gesperrt.
2. Falls eine w- oder iw-Sperre benötigt wird für ein Granulat $\Rightarrow$ alle Vorgänger müssen mit iw gesperrt werden.
3. Falls T_i ein Datenelement lesen (schreiben) will, so benötigt es eine r- (w-) Sperre für das Datenelement oder einen Vorgänger.
4. ix-Sperre kann freigegeben werden, falls keine Sperre für Nachfolger gehalten wird.

Kompatibilitätsmatrix (x: Konflikt):

Sperre geh.\angef.	rl	wl	irl	iwl	riwl
rl		x		x	x
wl	x	x	x	x	x
irl		x			
iwl	x	x			x
riwl	x	x		x	x

rl: share lock; wl: exclusive lock; irl: intention share lock; iwl: intention exclusive lock; riwl: share intention exclusive lock

Sperrgranulate (Sperrenanzahl : Nebenläufigkeit):

große Sperrgranulate $\Rightarrow$ wenig Sperren $\Rightarrow$ geringe Nebenläufigkeit
kleine Sperrgranulate $\Rightarrow$ viele Sperren (hoher Aufwand) $\Rightarrow$ hohe Nebenläufigkeit

Sperre auf ein Granulat X sperrt implizit alle unter X befindlichen Granulate.

$\varepsilon(MGL) \subseteq CSR$

5.6 Einfaches Baumsperrverfahren (TL)

Organisation: Daten selbst.

Keine Unterscheidung zw. Lese- und Schreiboperationen, nur Zugriffsoperation $a_i[x]$. Falls $i \neq j$ gilt $a_i[x] \not\parallel a_j[x]$. Daten werden hierarchisch in Datenbaum (DT) organisiert. Durchlaufrichtung: von oben nach unten! Wurzel: r

1. vor $a_i[x]$ muss $al_i[x]$ stattfinden.
2. $al_i[x]$ kann nur stattfinden, falls entweder kein $al_j[x]$ stattfand für $i \neq j$ oder $\forall al_j[x]$ auch $au_j[x]$.
3. Falls $x \neq r$ nicht Wurzel, kann $al_i[x]$ nur stattfinden, falls auch $al_j[y] \ <_H \ al_i[x] \wedge au_i[y] \ \not<_H \ al_i[x]$, wobei y der Vorgänger von x ist.
4. $au_i[x]$ kann nur nach $a_i[x]$ stattfinden.
5. Nach $au_i[x]$ kann kein $al_i[x]$ mehr erfolgen.

aus 3. 5. $\Rightarrow$ $al_i[x]$ kann nur freigegeben werden, falls die Transaktion alle benötigten Sperren auf den benötigten Söhnen bereits hält.

Mindestens eine Sperre pro Pfad wird garantiert: Keine Überholung durch andere Transaktionen möglich.

Das TL-Protokoll vermeidet Verklemmungen

Sperren können eher als bei 2PL freigegeben werden: Falls nötige Nachfolger von x gesperrt sind, kann Sperre auf x aufgegeben werden.
Problem: Wann ist dies der Fall?
Antwort: rein mechanisch (alle Nachfolger gesperrt) oder Zusatzkenntnisse durch TM (ohne degeneriert TL zu striktem 2PL).

$\varepsilon(TL) \subseteq CSR$

5.7 Verallgemeinerungen des TL-Protokoll

1. Irgendein Objekt im Datenbaum (nicht Wurzel) am Anfang gesperrt und danach nur noch Objekte im dort beginnenden Teilbaum
2. Unterscheidung zw. Lese- und Schreibsperren:
   Bei nur Lese- oder nur Schreibsperren genügen die normalen Konfliktregeln um Serialisierbarkeit zu gewährleisten.
3. Datenbaum ein gerichteter azyklischer Graph (DAG)

Synchronisation und Zugriff auf baumstrukturierte Indexe wie z.B. B^*-Bäume: spez. Protokoll.

Problem mit TL: Wegen mögl. Split wird der Baum nicht zwingend nur von der Wurzel an abwärts durchlaufen.

5.8 Zeitstempelbasierte Protokolle (TO)

Jeder Transaktion T_i wird ein eindeutiger Zeitstempel $ts(T_i)$ zugeordnet.

In Konflikt stehende Operationen geordnet gemäß Zeitstempel:
$\forall p_i[x] \not\parallel p_j[x] \in H \; (i \neq j) \; p_i[x] \; <_H \; p_j[x] \; \Leftrightarrow \; ts(T_i) < ts(T_j)$

TO Historien sind SR.

$\varepsilon(TO) \subseteq CSR$

5.9 Basis-TO (BTO)

Alle Operationen werden direkt an DM weitergeleitet. Operationen, die zu spät kommen werden zurückgewiesen: $p_i[x]$ zu spät $\Leftrightarrow$ $\exists q_j[x] \; <_H \; p_i[x] : \; ts(T_j) > ts(T_i)$
Im Vergleich zu 2PL beschränkt Basis-TO die Menge der akzeptierten CSR-Schedules.

Zur Feststellung, ob Operation zu spät ist, hält der Basis-TO-Scheduler für jedes Objekt x das Maximum aller Zeitstempel für ausgeführte r- & w-Operationen:
- $max - r - scheduled[x]$
- $max - w - scheduled[x]$

Eintreffen von $p_i[x]$: Vergleich $ts(T_i)$ mit allen $max - q - scheduled[x]$ für die q, mit p in Konflikt:

```
if (ts(T_i) < max − q − scheduled[x]) {
     p_i[x] zurückweisen
} else {
     p_i[x] an DM weiterleiten
     max − p − scheduled[x] := ts(T_i)
}
```

Datenmanager (DM) muss Reihenfolge garantieren $\Rightarrow$ Hand-shake-Verfahren

5.10 Konservatives TO

Operationen werden nicht sofort bei falschem Zeitstepel zurückgewiesen, sondern TO-Scheduler verzögert Operationen $\Rightarrow$ weniger Rücksetzen.
Problem: Leistungsverluste; wahl der Verzögerung; Abbruch von Transaktionen weiterhin möglich

5.11 Striktes TO (STO)

TO-Scheduler, der strikte Historien garantiert:
Verschärfung des Basis-TO-Hand-shake-Verfahrens:
Erst bei a_i oder c_i wird $w-in-transit[x]$ auf 0 gesetzt.

$\varepsilon(STO) \subseteq CSR \cap ST$

5.12 Ultimativ konservatives TO

Ziel: kein Abbruch von Transaktionen

5.13 SG-basiertes Scheduling

Serialisierbarkeitsgraphtest (SGT)

GSGT-Scheduler kontrolliert, dass der SG immer zyklenfrei $SG \Rightarrow SSG = (N, E)$
N enthält nicht nur committete Transaktionen, sondern auch aktive.
N enthält nicht alle committete Transaktionen (großer Teil Vergangenheit interessiert nicht)

5.14 Basis SGT

1. Falls $T_i \notin SSG:\ p_i[x]$ erhalten $\Rightarrow\ T_i$ einfügen.
2. $\forall q_j[x] <_H p_i[x],\ i \neq j,\ q_j[x] \not\parallel p_i[x] :\ SSG \cup \{T_j \rightarrow T_i\}$
3. Falls SSG jetzt Zyklus enthält:
   abort T_i und nach Empfangsbestätigung von DM: $SSG \setminus \{T_j \rightarrow T_i\}$
4. sonst: Schedule $p_i[x]$, fallsalle in Konflikt stehenden Operationen schon Empfang bestätigt (hierzu $q-in-transit[x]$).

Zur Bestimmung von Bedingung 2. zusätzlich für jede Transaktion T_i $r-$ und $w-scheduled[T_i]$ in denen die Lese- und Schreibmengen (Datenelemente) von T_i gehalten werden.

Transaktionen können aus SSG entfernt werden, wenn sie nicht mehr an einem Zyklus teilnehmen können. Für zyklus mindestens eine einfallende und eine ausfallende Kante nötig. Committete transaktionen ohne einfallende Kanten können aus SSG entfernt werden, da nach Commit nur ausfallende Kanten entstehen können.

5.15 Konservatives SGT

5.16 Striktes SGT

6 Mehrversionen-Scheduler

Jedes Schreiben von x erzeugt eine neue Kopie von x.

Mehrversionen-Historien repräsentieren die Ausführung des DM.
Einversionen-Historien repräsentieren die Interpretation von MV-Historien für eine einzelne Benutzertransaktion.

Serielle 1V-Historien erachtet der Benutzer als korrekt.

6.1 Übersetzung h

Sei $T = \{T_0, \ldots, T_n\}$ eine Menge von Transaktionen, wobei die Operationen von T_i durch $<_i$ geordnet sind. Um Operationen von T_i auszuführen, müssen sie auf Versionen übersetzt werden. Dies geschieht durch eine Abbildung h, die

1. $w_i[x]$ auf $w_i[x_i]$,
2. $r_i[x]$ auf $r_i[x_j]$ für ein geeignetes j,
3. c_i auf c_i und
4. a_i auf a_i

abbildet.

6.2 Mehrversionen-Historie (MV-Historie)

Eine vollständige MV-Historie H über T ist eine partielle Ordnung $<_H$, wobei gilt

1. $H = h(\bigcup_{i=0}^{n} T_i)$ für eine Übersetzung h
2. $\forall T_i : \forall p_i, q_i \in T_i : p_i <_i q_i \Rightarrow h(p_i) <_H h(q_i)$
3. $h(r_j[x]) = r_j[x_i] \Rightarrow w_i[x_i] <_H r_j[x_i]$
4. $w_i[x] <_i r_i[x] \Rightarrow h(r_i[x]) = r_i[x_i]$
5. $h(r_j[x]) = r_j[x_i],\ i \neq j,\ c_j \in H \Rightarrow c_i <_H c_j$

Bedingung 4.: H bewahrt reflexive Liest-von-Beziehung.
Bedingung 5.: Eine Transaktion liest nur Versionen von transaktionen, die vor ihr committen.

Eine MV-Historie ist ein Präfix einer vollständigen MV-Historie.

6.3 1-serielle MV-Historie

Eine vollständige MV-Historie ist seriell :⇔ $\forall T_i, T_j \in H$:
$(\forall p_i \in T_i \, q_j \in T_j : p_i <_H q_j) \vee (\forall p_i \in T_i \, q_j \in T_j : q_j <_H p_i)$

Eine serielle MV-Historie H ist 1-seriell :⇔ $\forall i, j, x$: falls $T_i \triangleright_H [x] \, T_j$, dann gilt
- $i = j$ oder
- T_j ist die letzte Transaktion vor T_i, die irgendwine Version von x schreibt.

Definition ist wohldefiniert, da die letzte Transaktion eindeutig bestimmt ist, weil H seriell ist.

Serielle 1V-Historie ≡ 1-serielle MV-Historie (somit korrekt synchronisiert)

Zu zeigen: MV-Historie erzeugt durch MV-Scheduler äquivalent zu 1-serieller MV-Historie über den Mehrversionenserialisierbarkeitsprahen (MVSG):
MV ist 1-seriell ⇔ MVSG ist azyklisch

6.4 Serialisierbarkeitsgraph (SG)

Sei H eine MV-Historie: SG(H) ist wie folgt definiert:

1. Die Knoten sind abgeschlossene Transaktionen.
2. $T_i \rightarrow T_j \in SG(H) \, (i \neq j) \Leftrightarrow T_j \triangleright_H [x] \, T_i$
   $(\Leftrightarrow r_j[x_i] \in CP(H), \, i = j)$

6.5 Versionsordnung

Sei H eine MV-Historie und x ein Datenelement. Eine Versionsordung $\ll$für x ist eine totale Ordung auf den Versionen von x in H. Eine Versionsordnung für H ist die Vereinigung der Versionsordungen aller in H vorkommenden Datenelmente x.

6.6 Mehrversionenserialisierbarkeitsgraph (MVSG)

$MVSG(H, \ll) := SG(H)$ erweitert um die folgenden Versionsordungskanten:
Für jedes $r_k[x_j]$ und $w_i[x_i]$ in $CP(H)$, mit $i \neq j \neq k$: falls $x_i \ll x_j$, so wird $T_i \rightarrow T_j$ hinzugefügt, sonst $T_k \rightarrow T_i$.
Entweder "T_i muss bzgl. x vor T_j stattgefunden haben"
oder "x_i kann erst nach $r_k[x_j]$ geschrieben werden (T_k muss bzgl. x vor T_i stattfinden)"

6.6.1 Beisiel

$$T_0 = \begin{array}{l} w_0[x] \backslash \\ w_0[y] - \; c_0 \\ w_0[z] / \end{array}$$

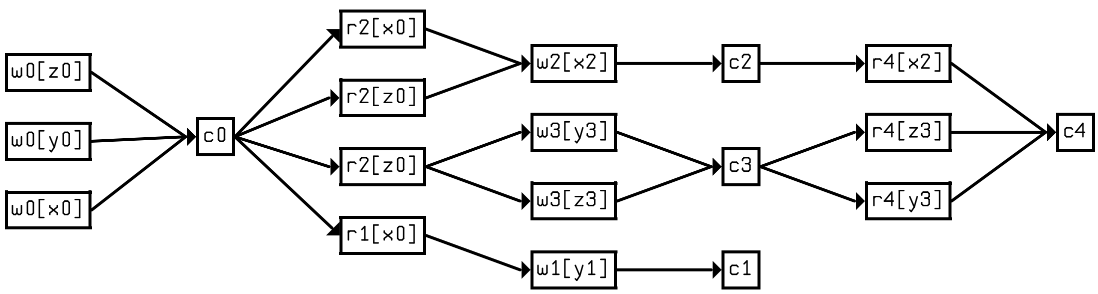
$$T_1 = r_1[x] - w_1[y] - c_1$$

$$T_2 = \begin{array}{l} r_2[x] \backslash \\ r_2[z] / \end{array} w_2[x] - c_2$$

$$T_3 = r_3[z] \begin{array}{l} / w_3[y] \backslash \\ \backslash w_3[z] / \end{array} c_3$$

$$T_4 = \begin{array}{l} r_4[x] \backslash \\ r_4[y] - \; c_4 \\ r_4[z] / \end{array}$$

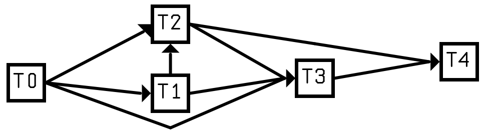
Vollständige MV-Historie H über $\{T_0, T_1, T_2, T_3, T_4\}$:



MVSG(H):

$r_1[x_0] w_2[x_2] \; x_0 \ll x_2 \Rightarrow T_1 \rightarrow T_2$
$r_2[x_0] -$
$r_2[z_0] w_3[z_3] \; z_0 \ll z_3 \Rightarrow T_2 \rightarrow T_3$
$r_3[z_0] -$
$r_4[x_2] w_0[x_0] \; x_0 \ll x_2 \Rightarrow T_0 \rightarrow T_2$
$r_4[y_3] w_1[y_1] \; y_1 \ll y_3 \Rightarrow T_1 \rightarrow T_3$
$r_4[y_3] w_0[y_0] \; y_0 \ll y_3 \Rightarrow T_0 \rightarrow T_3$
$r_4[z_3] w_0[z_0] \; z_0 \ll z_3 \Rightarrow T_0 \rightarrow T_2$

zusätzlich zu SG(H): $T_1 \to T_2$, $T_1 \to T_3$ und $T_2 \to T_3$:



6.7 MV-Äquivalenz ($H \equiv_{MV} H'$)

Zwei MV-Historien ($H,\ H'$) über eine Menge T von Transaktionen sind äquivalent, falls gilt:

1. $op(H) = op(H')$
2. gleiche liest-von-Beziehung

6.8 MV-Zeitstempelverfahren

6.9 2-Versionen 2PL

Anpassung von 2PL

7 Verteilte Scheduler

7.1 Verteilte Datenbanken

Annahmen:

1. Rechnerknoten lose gekoppelt, kommunikation über Netz
2. Knoten lokal autonom, kein “Master”-Knoten mit “globalem” Wissen
3. Datenverteilung für Benutzer transparent (Ort, Replikation, Ausführung)

Folgerungen und Einschränkungen:

1. DB-Architektur (Scheduler, DM, etc.) existiert in jedem Knoten
2. Homogenität

3. zunächst keine Datenreplikation
4. Transaktion wird am Haus-Knoten gestartet und dort das Ergebis abgeliefert
5. Was der hausknoten nicht bearbeiten kann gibt er weiter
6. Abarbeitung in mehreren Knoten kann parallel erfolgen (Historien mit nur patieller Ordnung nötig)
7. TM betroffener Knoten müssen zusammenwirken ⇒ höherer Verwaltungsaufwand

7.2 Verteiltes Zeitstempelbasiertes Protokoll

Global eindeutige Zeitstempel ohne zentrale Instanz oder Kommunikation zwischen den Knoten. Wird T an Knoten S_i zur relativen Zeit z_T gestartet und gibt es insgesamt k Knoten: $ts(T) = z_T * k + i$

Zeitstempel sind global eindeutig und aufeinanderfolgende Transaktionen eines Hausknotens erhalten aufsteigende Zeitstempel. Globale Monotomie der Zeitstempel ist allerdings nicht gewährleistet - somit ist das Verfahren nicht zwingend fair.

7.3 Verteiltes 2PL

Transaktionsmanager des Haus-Knotens sendet $r[x]$ und $w[x]$ an Scheduler des x führenden Knoten. Jeder Scheduler unterhält lokal die Sperren für seine Objekte gemäß 2PL.

Problem: Sperre erst aufhebbar, wenn keine Sperre in keinem Scheduler mehr gesetzt wird.

Lösung: Verwendung des strikten 2PL, auch die Lesesperren werden bis zum Commit-Zeitpunkt gehalten (verteiltes, atomares Commit).

Vorgehen: Koordinierte Freigabe aller Sperren. TM des Haus-Knotens koordiniert beteiligte Scheduler.

7.4 Verteilte Verklemmungen

Ein Knoten verwaltet den Globalen Wartegraph WG, die Vereinigung aller lokalen WG_i. Probleme: Verzögerte Erkennung (perodische Übertragung der WG_i); Ausfallmöglichkeit der zentralen Komponente.

Zeitstempelbasierte Verklemmungsvermeidung: Priorität umgekehrt zum Zeitstempel gesetzt.

7.5 (Vollständige) Mehrkopien-Historien

Eine vollständige MC-Historie H über T ist eine partielle Ordnung $<_H$, wobei gilt:

1. $H = h(\bigcup_{i=0}^{n} T_i)$ für eine Übersetzung h
2. $\forall T_i \, \forall p_i, \, q_i \in T_i : \, p_i \, <_i \, q_i \, \Rightarrow \, h(p_i) \, <_H \, h(q_i)$
3. $h(r_j[x]) = r_j[x_k] \, \Rightarrow \, w_i[x_k] \, <_H \, r_j[x_k]$
4. $w_i[x] \, <_i \, r_i[x] \, \wedge \, h(r_i[x]) = r_i[x_k] \, \Rightarrow \, w_i[x_k] \in h(w_i[x])$
5. Alle Paare von unverträglichen Operationen sind durch $<_H$ geordnet.

Bedingung 4. besagt, dass eine Transaktion welche x schreibt, bevor sie x liest, die Kopie von x schreiben muss, welche sie anschließend liest.

8 Fehlersicherheit

8.1 Wiederanlaufbarkeit / Rücksetzbarkeit (RC)

Historie $H \in RC$, falls $T_i \triangleright_H T_j, \, c_i \in H \, \Rightarrow \, c_j \, <_H \, c_i$ für alle Transaktionen T_i, T_j $(i \neq j)$ aus H.

"Falls T_i von T_j liest, darf T_i nur nach T_j erfolgreich enden."

8.2 Vermeidung von kaskadierendem Rücksetzen (ACA)

Historue $H \in ACA$, falls $T_i \triangleright_H [x] \, T_j \, \Rightarrow \, c_j \, <_H \, r_i[x]$ für alle Transaktionen T_i, T_j $(i \neq j)$ aus H.

"Falls T_i von T_j liest, darf das lesen erst nach dem erfolgreichen Ende von T_j erfolgen."

8.3 Striktheit (ST)

$\Rightarrow$ einfacher beim Algorithmus.

$ST \subset ACA \subset RC$

Falls $w_j[x] \, <_H \, p_i[x] \, \Rightarrow \, a_j \, <_H \, p_i[x] \, \vee \, c_j \, <_H \, p_i[x]$ für alle Transaktionen T_i, T_j $(i \neq j)$ aus H.

"Transaktion darf Datenelement erst schreiben oder lesen nachdem die Transaktion, die es zuvor schrieb abbrach oder erfolgreich endete."

CSR, RC, ACA, ST sind PCC (präfix-commit-abgeschlossen)

8.4 RC, ACA, ST orthogonal zu CSR, VSR

Schnitt zwischen $\{CSR, VSR\}$ und $\{RC, ACA, ST\}$ nicht leer, aber keine Teilmengenbeziehung (VSR nach Definition Bernstein!)

Scheduler muss Serialisierbarkeit und Wiederanlaufbarkeit/Vermeidung kaskadierenden Rücksetzens/Striktheit getrennt garantieren!

8.5 Rigorosität (RG)

Historie H heißt rigoros, falls sie strikt ist und gilt:
$\forall T_i, T_j \in trans(H) : r_j[x] <_H w_i[x], i \neq j \Rightarrow a_j <_H w_i[x] \vee c_j <_H w_i[x]$

"Rigorose Historien verschärfen Bedingung für Striktheit:
Es darf kein Datum überschriebn werden, solange die Transaktionen, die es zuletzt lasen, noch nicht endeten."

$RG \subset ST$, $RG \subset CSR$, $RG \subset CO$ d.h.: $RG \subset (CO \cap ST)$

9 Recovery

9.1 Recovery-Strategien

Transaktions-Fehler:
Bei Abbruch einer Transaktion stellt DM die alten Werte aller Datenelemente, die die Transaktion veränderte, wieder her.

System- oder Media-Fehler:
DM bricht alle Transaktionen ab, die zum Fehlerzeitpunkt aktiv waren und stellt sicher, dass die Ergebnisse aller bereits committeten Transaktionen in der Datenbasis festgeschrieben sind.

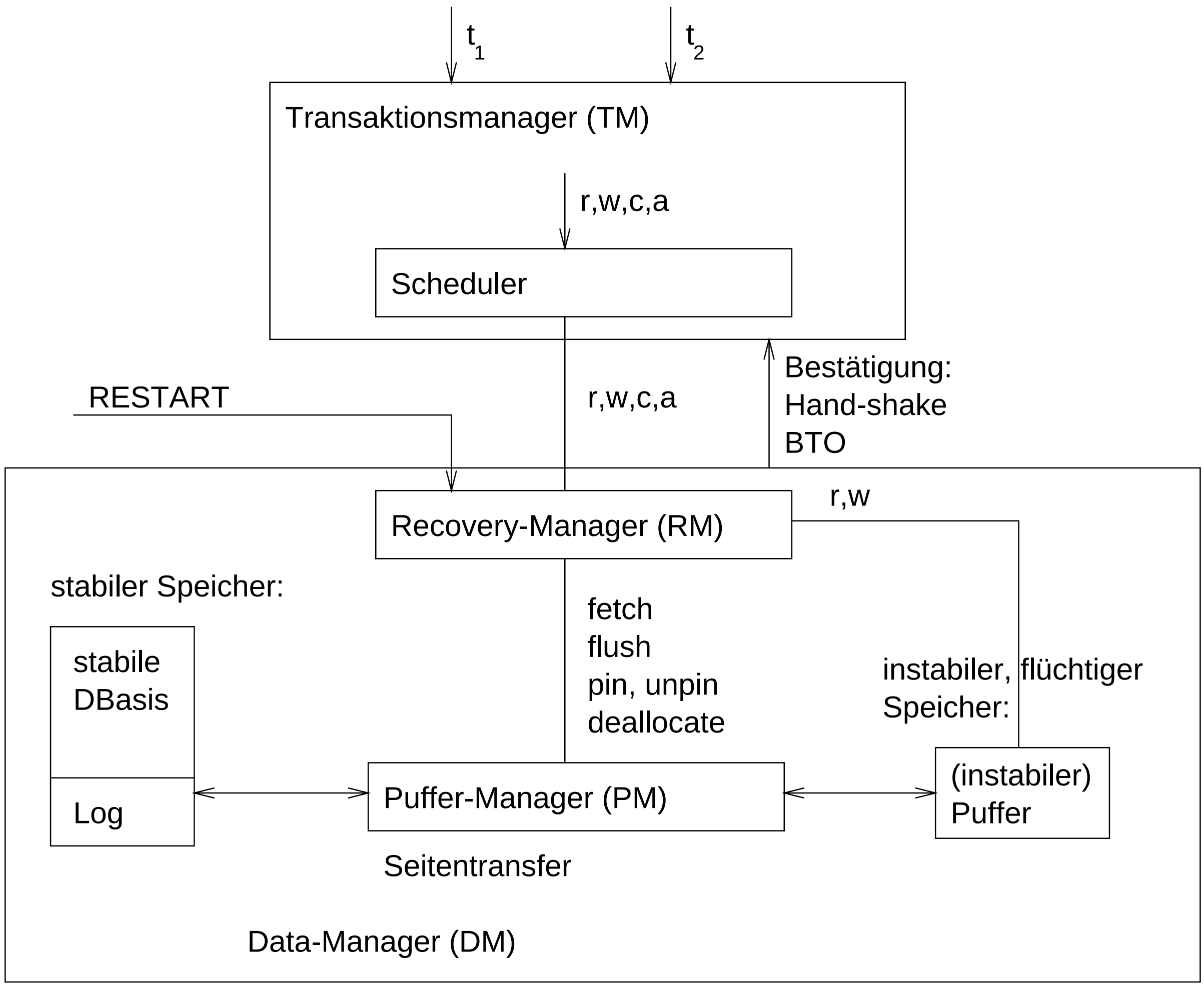
Redundante Kopie der Daten im (Haupt- und Hintergrund-) Speicher $\Rightarrow$ Log.

9.2 Freigegebener Datenbankzustand

Letzter freigegebener Wert von Datenelement x ist der letzte von einer abgeschlossenen transaktion geschriebene Wert von x. Freigegebener DB-Zustand $:= \forall x$ freigegebener Wert.

Restart: Freigegebener DB-Zustand herstellen.
- Abort(T_i) für alle T_i zum Fehlerzeitpunkt aktiv
- $\forall T_i \in commit(H)$, Ergebnisse nur im Log: Eintragen in stabile DB

9.3 Architektur



Stabile Datenbasis: Kopie jeder Datenbankseite (Ergebnis jüngster flush-Operation)

Puffer: Kopie einiger datenbankseiten. Seiten, die Änderungen enthalten, die noch nicht in den stabilen Speicher übertragen wurden =: dirty pages

Log: Sequentielle Datei - meist im stabilen Speicher - enthält Einträge, die Änderungen der Datenbasis beschreiben: Adresse geänderte Seite; Transaktions-ID; Wert nach Beschreiben (after image); Wert vor Beschreiben (before image) $\Rightarrow$ 2 Datenbankseiten zu ineffizient.

9.4 Operationen des Recovery-Managers (RM)

Commit(T_i):
* von T_i geänderte Seite in stabilen Speicher (nicht unbedingt Datenbasis)

* atomar, selbst im (System-) Fehlerfall
* endgültig: nachdem T_i committet kein Abort möglich

Abort(T_i):
* Änderungen durch T_i in stabiler Datenbasis rückgängig machen (nicht unbedingt auch Log)
* atomar
* endgültig: nach Abort kein Commit möglich
* idenpodenz: garantiert immer durchführbar

9.5 Flush-Strategie des Puffers

steal: flush vor RM-Commit möglich

nosteal: kein flush vor RM-Commit

force: flush bei RM-Commit

noforce: kein flush bei RM-Commit erforderlich

"undo": mache unwirksam

"redo": Wiederherstellung ≡Wiederholung

Restart-Strategien:

steal/noforce	undo	redo
steal/force	undo	%
nosteal/noforce	%	redo
nosteal/force*)	%	%

*) widersprüchlich, nur besondere Techniken! ⇒ praktisch: entweder undo oder redo nötig
RM kann vollständig ohne undo & redo auskommen, wenn alle durch eine transaktion gemachten Änderungen genau zum Commit-Zeitpunkt (in einer einzigen, atomaren Operation) in die DB geschrieben werden.

RM erfordert undo, falls nicht abgeschlossene Transaktionen veränderte werte bereits in die Datenbasis schreiben können (steal)

RM erfordert redo, falls von abgeschlossenen transaktionen veränderte Werte nicht zwingend in stabile Datenbasissein müssen (noforce)

Write-Ahead Log-Protokoll (undo-Regel): Nicht-Committete Änderungen ("dirty pages") dürfen erst zur stabilen Datenbais geflusht werden, nachdem der Log-Eintrag mit dem zugehörigen Before-Image in die Log-Datei geflusht wurde.

Commit-Regel (redo-Regel): Bevor eine Transaktion enden kann muss jeder von ihr geschriebene wert im stabilen Speicher (stabile DB oder Log) stehen.

9.6 Log-Eintrag

1. Name der Transaktion T_i
2. Position von x in der stabilen Datenbasis
3. Offset/Länge der Änderung
4. alter Wert von x (steal)
5. neuer Wert von x (noforce)
6. Log-sequnece Number (LSN) des Logeintrags der vorherigen Änderungsoperation von T_i

10 Verteilte Recovery

Atomare Commit Protokolle (ACP) nötig!

Koordinator für Transaktion regelt mit beteiligten Knoten (Teilnehmern) ein atomares Commit oder Abort. Anforderungen:

- Alle TMs, die eine Entscheidung auf Commit oder Abort fällen (d.h. nicht alle müssen eine Entscheidung fällen!), fällen die gleiche Entscheidung
- Einmal getroffene Entscheidung wird nicht revidiert.
- Commit-Entscheidung wird nur getroffen, wenn alle Knoten-TMs mit "ja" stimmen.
- Treten keine Fehler auf $\Rightarrow$ Commit
- Falls bestimmte Fehler behandelt werden können, und falls deren Behandlung abgeschlossen ist, und falls hinreichend lange keine weiteren Fehler auftreten, so kommen TMs zu einer Entscheidung ($\Rightarrow$ Protokoll terminiert)

10.1 2-Phasen-Commit-Protokoll (2PC)

1. Phase (Wahlphase):

   (a) Koordinator sendet vote-request an alle Teilnehmer

   (b) Trifft vote-request bei TN ein, so antwortet er mit "ja" (ready-to-commit) oder "nein". Stimmt er auf "nein" führt er gleich autonom, lokal ein Abort der Transaktion durch. Bei "ja" bleibt Entscheidung offen ⇒ alten & neuen Zustand halten.

2. Phase (Entscheidungsphase):

   (a) Koordinator sammelt Stimmen. Sind alle (incl. seiner) "ja", so wird auf Commit entschieden und an alle Teilnehmer geschickt.
   Ansonsten wird auf Abort entschieden und alle "ja"-TN informiert.

   (b) TN der "ja" gestimmt har wartet auf Commit oder Abort. Trifft dies ein, wird entsprechend entschieden und die Ausführung an Koordinator gemeldet

Nachteil des 2PC

- keine autonome Entscheidung während der Unsicherheitsphase
- blockierendes Terminierungsprotokoll

Mögliche Fehler bei 2PC

Unsicherheitsphase des Knoten-TM: Zeit zwischen Absetzen des Votums und Erhalt der endgültigen Entscheidung.

Mögliche Fehler in dieser Phase ⇒ Gefahr: Nicht-Terminieren

1. Kommunikationsfehler
   ⇒ Timeout-Aktion nach festgelegter Wartezeit:

   (a) TN wartet auf vote-request
   Aktion: TN entscheidet (autonom) auf Abort

   (b) Koordinator wartet auf Stimme eines TN
   Aktion: Koordinator votiert auf Abort und teilt dies allen "ja"-TN mit.

   (c) TN, mit "ja" votiert, wartet auf Entscheidung: TN ist unsicher, kann nicht alleine entscheiden ⇒ Terminierungsprotokoll: Anfrage bei anderem TN:

i. anderer TN hat mit Nachricht von Koordinator auf Commit/Abort entschieden
ii. anderer TN hat mit “nein” gestimmt.
iii. anderer TN hat mit “ja” votiert, auch keine antwort ⇒ warten oder noch einen TN befragen, mindestens Koordinator hat Antwort.

2. Knotenfehler
⇒ bei Wiederanlauf Distributed Transaction Log nötig!

10.2 Distributed Transaction Log (DTL)

Um nach (lokalem) Restart konsistente Entscheidungen treffen zu können: Nötige Infos:

1. Koordinator schreibt Start-2PC mit ID der TN in sein DLT
2. Votiert TN mit “ja”, schreibt er dies und die ID der anderen beteiligten TMs in sein DLT. Votiert er mit nein, schreibt er Abort in sein DLT
3. Koordinator schreibt Commit/Abort in sein DLT, bevor er diese Nachricht abschickt
4. TN protokolliert empfangene Entscheidung
5. Koordinator schreibt Done in sein DLT, wenn er die Bestätigung von allen TN erhalten hat

10.3 3-Phasen-Commit-Protokoll (3PC)

Erweiterung des 2PC: 1. Phase identisch, 2. Phase in 2 Phasen eingeteilt

1. Phase (Wahlphase)
2. Phase (Entscheidungsvorbereitungsphase):
   (a) Koordinator sammelt alle Voten. Sind alle “ja” (incl. eigener) sendet er prepare-to-commit an alle TN. Ansonsten wird Abort beschlossen und die “ja”-TN informiert.
   (b) Jeder “ja”-TN wartet auf prepare-to-commit- oder Abort-Nachricht. trifft Abort ein entscheidet er auf Abort, sonst sendet er Koordinator eine Bestätigung.
3. Phase (Entscheidungsphase):

(a) Koordinator sammelt Bestätigungen. Hat er alle bekommen entscheidet er auf Commit und sendet dies an alle TN

(b) TN warten auf Commit-Entscheidung des Koordinators und führen diese bei Entreffen aus.

nach 2 b): Jeder prepare-to-commit-Knoten: die nächste Nachricht, die er erhalten wird: Commit. Er ist sicher. ⇒ alle Up-Knoten in der Sicherheitsphase.

nach 3 b): Er ist sicher, dass auch alle anderen Knoten sicher sind. ⇒ alle, auch Down-Knoten

Blockaden werden nicht ganz verhindert, nur reduziert!

Timeout-Aktionen

11 Geschachtelte Transaktionen

11.1 Geschachtelte Transaktionen

1. Isolation der Gesamttransaktion
   Konsistenzwahrung innerhalb der Transaktion
   Nebenläufigkeit innerhalb der Transaktion
   ⇒ Geschlossen geschachtelte Transaktion

2. Kooperation von Transaktionen
   Reduzierte Konsistenz und Isolation
   Hohe Nebenläufigkeit über alle Transaktionen
   ⇒ Offen geschachtelte Transaktionen

Wurzel-/ Anwendungstransaktion mit Subtransaktionen/Blatt-Transaktionen in herkömmlichen R-/W-Sinne

Entscheidungsräume:

1. Zusammenhänge zwischen Transaktionen eines Pfades

   (a) Fehlschlag Subtransaktion ⇒ Fehlschlag der Vatertansaktion?

   (b) Fehlschlag der Vatertransaktion Fehlschlag Subtransaktion ⇒ Fehlschlag aller Subtransaktionen?

2. Zusammenhänge zwischen Transaktionen verschiedener Pfade

   (a) Sichtbarkeit der Ergebnisse bereits abgeschlossener Subtransaktionen?

3. Eigenschaften der Transaktion
   (a) A, C, I, D bei Blatt-Transaktionen?
   (b) A, C, I, D bei Nicht-Blatt-Transaktionen?

Vorteile gegenüber konventionellen (ACID-) Transaktionen:

- Bei Fehlschlägen von Teilbuchungen muss nicht die gesamte Transaktion zurückgesetzt werden.
- Sperren auf die gesamte Datenbasis müssen nicht die ganze Transaktion gehalten werden (!= geschlossen!)

11.1.1 Sagas

Einstufiger Spezialfall geschachtelter Transaktionen. Hierachische Strukturierung, um schrittweises Vorgehen in der Anwendung wiederzuspiegeln. ⇒ Pfade

Langdauernde Transaktionen behindern sich bei Isolation gegenseitig ⇒ Abschwächung ACID:

1. Begrenzte Isolation/Konsistenz:
   - Zerlegung einer transaktion S_i (Saga) in Folge von ACID-Untertransaktionen T_{ij}.
   - Finalzustände der T_{ij} werden nicht nur T_{ij+1}, sondern auch den T_{kl}, $k \neq i$ anderer Sagas sichtbar.
2. "schwächer" definierte Atomizität (keine Wiederherstellbarkeit, keine Unsichtbarkeit):
   - Saga S_i wird bei Fehlschlag einer T_{ij} in ihrer Gesamtheit zurückgesetzt.
   - Jede Untertransaktion T_{ij} ist mit einer Kompensationstransaktion C_{ij} ausgestattet.
   - Bei Rücksetzen: C_{ij} in umgekehrter Reihenfolge der T_{ij} ausgeführt.
   - Andere Sagas S_k, $k \neq i$ werden nicht zwingend informiert und daher üblicherweise nicht zurückgesetzt.
     Eine Saga $S = ((T_1, C_1), \ldots, (T_n, C_n))$ wird entweder erfolgreich abgearbeitet: $T_1, T_2, \ldots, T_n$ oder bei einem Fehler in T_j kompensiert: $T_1, T_2, \ldots, T_j[Fehler], C_{j-1}, \ldots, C_2,$
     Über die C_{ij} werden keine Annahmen gemacht, insbesondere nicht der Zustand zu Beginn von S_i garantiert. Auf Grund der begrenzten Isolation nicht sinnvoll.

Menge der T_{ij} für eine Menge von Sagas S_i bilden wie gewohnt eine Historie ⇒ Saga-Manager (SM) führt zusätzlich zu TM ein Log!

11.1.2 Geschlossen geschachtelte Transaktionen

1. Zusammenhang zw. Transaktionen eines Pfades:
    - Fehlschlag Subtransaktiion ⇒ Störung, aber Fehlschlag der Vatertransaktion nicht zwingend. Vatertransaktion bestimmt Stelle in Ausführung, von der fortsetzung möglich ist.
    - Störung Vatertransaktion ⇒ Fehlschlag aller Subtransaktionen nach Sicherungspunkt.
    - Fehlschlag Vatertransaktion ⇒ Fehlschlag aller bereits ausgeführten Subtransaktionen.
2. Zusammenhang zw. Transaktionen verschiedener Pfade:
    - Ergebnis abgeschlossener Transaktionen nicht sichtbar, weder innerhalb noch außerhalb der Anwendungstransaktion. Ausnahme für sequentielle Subtransaktionen derselben Vatertransaktion: Sichtbarkeit.
3. Eigenschaften der Transaktionen
    - Blatt-Transaktionen sind ACID

Rücksetzpunkte: Regel: Für jede transaktion stellt deren Beginn und der Transaktionsbeginn jeder Subtransaktion einen Rücksetzpunkt dar.

Für jedes von einer Subtransaktion veränderte Objekt wird ein Before-Image angelegt, bei Commit der Subtransaktion an Vatertransaktion vererbt, die alle (ererbten) Before-Images bis Ende hält.

Synchronisation: 2PL-Protokoll: Sperren abgeschlossener Subtransaktionen dürfen nicht freigegeben werden, sondern werden an Vatertransaktion vererbt. Sperranforderungen einer Subtransaktion: Verträglichkeit mit gehaltenen Sperren ihrer Vorfahren (nicht mit parallelen Geschwistern).

Intratransaktions-Parallelität: Konditional unabhängige Subtransaktionen einer Anwendungstransaktion können parallel bearbeitet werden. ⇒ Modellierung als geschachtelte Subtransaktionen mit Garantie betreffs Isolation.

Konsequenzen

1. Subtransaktion bildet eine eigene Commit-Sphäre.
   Änderungen werden am transaktionsende, d.h. vor Beendigung der umgebenden Transaktion, freigegeben.

2. Bei Fehlschlag der umgebenden Transaktion kein Rücksetzen möglich, stattdessen Aufruf von Kompensationstransaktion

3. Fehlschlag innerhalb Transaktion (außer Blatt-Transaktionen) ⇒ Rücksetzpunkte erforderlich.

4. Besondere Regelung zur Isolation!

11.1.3 Offen geschachtelte Transaktionen

Offen: Wegfall der Isolation aller Transaktionen außer Blatt-Transaktionen.
Anders als bei geschlossen geschachtelten Transaktionen da keine Isolationsgarantie hier Erhöhung der Intertransaktions-Parallelität.

1. Zusammenhang zw. Transaktionen eines Pfades:
   - Fehlschlag einer Subtransaktion ⇒ Störung der Vatertransaktion, Kompensationstransaktion
   - Störung/Fehlschlag der Vatertransaktion ⇒ keine Auswirkungen auf Subtransaktionen, jedoch zusätzliche Kompensationstransaktion

2. Zusammenhang zw. Transaktionen verschiedener Pfade:
   - volle Sichtbarkeit

Vorteil: Hohe Intertransaktions-Parallelität. Andere Transaktionen können früher auf die geänderten Daten zugreifen.

Problem: Falls die Anwendungstransaktion abgebrochen wird, kann eine abgeschlossene Subtransaktion, da möglicherweise auf veränderte Daten zugegriffen wurde, nicht zurückgesetzt werden.
Parallel ablaufende Anwendungstransaktionen nicht serialisierbar (nur Subtransaktionen).

Lösung: Statt Rücksetzen Kompensationstransaktionen.
Subtransaktionen setzen Sperren mit differenzierteren Bedeutungen. ⇒ Isolation wieder gewährleistet (semantic Concurrency Control).

Konsequenzen:
vgl. 1. - 3. bei Geschlossen geschachtelte Transaktion!

4. Besondere Behandlung zur Isolation entfällt, da keine Isolation

Einordnung/Zusammenfassung

1. Sagas:
   völlig unabhängige Subtransaktionen, keine Isolationsgarantie, abgeschwächter Konsistenzbegriff

2. Geschlossen geschachtelte Transaktionen:
   + Intratransaktionsparallelität
   - Intertransaktionsparallelität
   Wurzeltransaktion: ACID

3. Offen geschachtelte Transaktionen:
   Einschränkung der Isolation aber evtl. stärkere Garantie als Sagas

12 Transaktionen in Multidatenbanksystemen

12.1 Multidatenbanksysteme (MDBMS)

Föderation unabhängiger lokaler Datenbanksysteme (LDBMS)
Jedes LDBMS ist autonom:

- Entwurf: Unabhängigkeit bezüglich verwendetem Datenmodell, logischem und physischem Entwurf, Anfragesprache, Concurrency-Control- und recovery-Protokoll.
- Ausführung: Lokale Transaktionen werden von LDBMS selbständig und ohne Kommunikation bearbeitet. Teile globaler Transaktionen werden wie lokale bearbeitet.
- Kommunikation: LDBMS können miteinander kommunizieren, können aber zur Kommunikation an sich oder Zeitpunkt nicht gezwungen werden.

Definitionen

Globale Transaktion: Transaktion die Daten mehrerer LDBMS benutzt.

Globale Datenbasis: Vereinigung der lokalen Datenbasen, LDBMS autonom, keine Replikation von Daten.

12.2 Transaktionsmodell für MDBMS

mit Gewalt durchzusetzen!

- Lokale Transaktionen sind in sich sequentiell (also totale Ordnung der DB-Operationen).
- Operationen in lokalen Transaktionen: Read, Write & Commit.
- Abbruch lokaler Transaktionen wird nicht betrachtet. ("Es findet kein Abbruch statt.")
- Globale Transaktionen ebenfalls sequentiell (totale Ordnung statt Halbordnung).
- Operationen in globalen Transaktionen: Read, Write, Abort, Commit.
- Jede globale Transaktion startet für einen LDM_i höchstens eine Subtransaktion.
- Globale Transaktionen müssen in allen LDBMS, in denen sie aktiv sind, Freigabeoperationen enthalten.

12.3 Globaler Transaktionsmanager (GTM)

- GTM erzeugt global korrekte Schedules (konsistenzerhaltend für die globale Datenbasis).
- GTM hat nach Voraussetzung keinen Einfluss auf die ausführung lokaler transaktionen.
- Bei lokalen Fehlern könnten Transaktionen in einzelnen LDBMS abstürzen bzw. weiterlaufen ⇒ Behebung des Problems über Protokolle für verteilte Datenbanken.
- Erkennung globaler Deadlocks schwierig!
- GTM_i: Komponenten in jedem $LDBMS_i$ mit denen GTM zur Ausführung von globalen Transaktionen kommuniziert.

12.4 Globaler Scheduler

MDBMS bestehend aus n LDBMS, $T_1, \ldots, T_n$ Mengen lokaler Transaktionen in Knoten $1, \ldots, n$, T sei Menge globaler Transaktionen. $H_1, \ldots, H_n$ lokale Schedules mit $T_i \subseteq trans(H_i)$, $T \cap trans(H_i) = \emptyset$ für $1 \leq i \leq n$:

Globaler Schedule (für $H_1, \ldots, H_n$) ist ein Schedule H mit:
1) $trans(H) = \bigcup_{i=1}^{n} T_i \cup T$ und
2) $\forall i,\ 1 \leq i \leq n : \prod_i(H) = H_i$

$\prod_i(H)$ bezeichne die Projektion von H auf Knoten i, d.h. lokale Transaktionen und ausgeführte Teile globaler Transaktionen im Knoten i.

Korrektheitskrierium: Für globale Transaktionen (auch für solche, zw. denen keine Konflikte bestehen) muss in allen LDBMS die gleiche relative Serialisierungsordung eingehalten werden.

Globale Konflikt-Serialisierbarkeit

Sei H globaler Schedule mit einer Menge T von Transaktionen

H heißt global konflikt-serialisierbar, falls eine Totalordnung $\leq_H$ auf der Menge $commit(H) \cap T$ existiert, welche mit den lokalen Serialisierbarkeitsordungen der freigegebenen globalen Transaktionen konsisitent ist, d.h.:

$\forall T_1, T_2 \in commit(H) \cap T, T_1 \neq T_2 :$
$T_1 <_H T_2 \Rightarrow \forall H_i, 1 \leq i \leq n, T_1, T_2 \in commit(H_i)\ \exists H_i', H_i'$ seriell, $H_i \equiv_C H_i' : T_1 <_{H_i'} T_2$

Klasse global konflikt-serialisierbarer Schedule GCSR

12.5 Globaler Konflikt-Graph

Globaler Schedule H für die lokalen Schedules $H_1, \ldots, H_n$; $G(H_i)$bezeichne den Konflikt-Graphen von H_i. Der globale Konflikt-Graph von H ist definiert als die Vereinigung aller $G(H_i),\ 1 \leq i \leq n$, d.h. $G(H) = \bigcup_{i=1}^{n} G(H_i)$

$H \in GCSR \Leftrightarrow G(H)$ ist azyklisch.

Möglichkeiten wie Verzögerung einzelner Transaktionen, damit kein Zyklus im globalen Konflikt-Graph entsteht, widersprciht der Autonomie-Voraussetzung einzelner LDBMS.

Globale Serialisierungsordungen kann der GTM nur dadurch bestimmen, dass er die Reihenfolge, in welcher Operationen an die LDBMS übergeben und von diesen bestätigt werden, kontrolliert.

Inhaltsverzeichnis