Transaktionssysteme

Guido Moerkotte

20. April 2004

Inhaltsverzeichnis

Vorbemerkung

Diese Skript wurde anhand des Buches [1] erstellt und hält sich im Wesentlichen an dessen Aufbau. Der Beweis der NP-Vollständigkeit für die Sichtenserialisierbarkeit wurde [3] entnommen (leicht modifiziert). Der Teil über Hot-Spots entstammt [2].

Kapitel 1

Wiederholung

1.1 (Ungerichtete) Graphen

Definition 1.1.1 (Ungerichteter Graph)
Ein ungerichteter Graph *ist ein Paar $G = (N, E)$, wobei N eine Menge von Knoten ist und $E \subseteq (N \times N)$ eine Menge von Kanten. Die Paare in E seien ungerichtet.*

Geordnete Tupel, insbesondere also Paare werden wir als $[v_1, \ldots, v_n]$ bezeichnen.

Definition 1.1.2 (Pfad)
Gegeben sei ein Graph $G = (N, E)$. Ein Pfad in G *ist eine endliche Folge $v_1, \ldots, v_n$ von Knoten, so dass für alle $1 \leq k < n$*

$$[v_k, v_{k+1}] \in E.$$

In diesem Fall verbindet *der Pfad v_1 und v_n.*

Definition 1.1.3 (Verbunden)
Ein Graph $G = (N, E)$ heißt verbunden *genau dann, wenn für alle $v, v' \in N$ gilt:*

Es existiert ein Pfad, der v und v' verbindet.

Definition 1.1.4 (Teilgraph)
Ein Graph $G' = (N', E')$ ist ein Teilgraph *eines Graphen $G = (N, E)$:≺≻ $N' \subseteq N$ und $E' \subseteq E$.*

Definition 1.1.5 (Partitionierung)
Eine Familie $G_i = (N_i, E_i)$ mit $i \in \{1, \ldots, n\}$ von Teilgraphen heißt Partitionierung *eines Graphen $G = (N, E)$:≺≻*

1. *Jeder Graph G_i ist verbunden und*
2. *$\forall i, j \in \{1, \ldots, n\} i \neq j \succ N_i \cap N_j = \emptyset$.*

Jeder Graph G_i wird Komponente *von G genannt.*

Bemerkung 1.1.6 *Man beachte, dass in vorhergehender Definition die Teilgraphen den Gesamtgraphen nicht umfassen müssen. Es wurde also weder*

$$\cup_{i=1}^{n} N_i = N$$

noch

$$\cup_{i=1}^{n} E_i = E$$

gefordert.

1.2 Gerichtete Graphen

Definition 1.2.1 (Gerichteter Graph)
Ein gerichteter *Graph ist ein Paar $G = (N, E)$, wobei N eine Menge von Knoten ist und $E \subseteq (N \times N)$ eine Menge von Kanten. Die Paare in E seien geordnet.*

Geordnete Tupel, insbesondere also Paare, werden wir als $(v_1, \ldots, v_n)$ bezeichnen.

Definition 1.2.2 (Pfad)
Gegeben sei ein gerichteter Graph $G = (N, E)$. Ein Pfad in G *ist eine endliche Folge $v_1, \ldots, v_n$ von Knoten, so dass für alle $1 \leq k < n$*

$$(v_k, v_{k+1}) \in E.$$

In diesem Fall verbindet *der Pfad v_1 und v_n. Jedes Paar (v_k, v_{k+1}) ist ein* Element *des Pfades. Ist $v_n = v_1$ und $n > 1$, so handelt es sich um einen* Zyklus.

Ein einzelner Knoten stellt den trivialen Pfad dar (n=1). Dies ist zu unterscheiden von einer *Schlinge*, also einem Pfad v, v.

Definition 1.2.3 (Vorgänger/Nachfolger)
Sei $G = (N, E)$ ein gerichteter Graph und $v, v' \in N$. Ist $(v, v') \in E$, so heißt v direkter Vorgänger *von v' und v'* direkter Nachfolger *von v. (Auch als $v \to v'$ notiert.) Falls ein Pfad von v nach v' führt, so heißt v* Vorgänger *von v' und v'* Nachfolger *von v. (Auch als $v \to^* v'$ notiert.)*

Definition 1.2.4 (Unabhängig) *Zwei Knoten $v, v' \in E$ eines Graphen $G = (N, E)$ heißen unabhängig voneinander, falls weder v ein Vorgänger von v' ist, noch umgekehrt.*

1.3 Gerichtete unzyklische Graphen

Definition 1.3.1 (Gerichteter unzyklischer Graph)
Ein gerichteter unzyklischer Graph *ist ein gerichteter Graph $G = (N, E)$, der keine Zyklen enthält.*

Anstelle der Bezeichnung gerichteter unzyklischer Graph wird oft *DAG* (*directed acyclic graph*) verwendet.

Bemerkung 1.3.2 *Man beachte, dass in einem DAG ein Knoten nicht gleichzeitig sein direkter Vorgänger und direkter Nachfolger sein kann.*

Definition 1.3.3 (Wurzel)
Ein Knoten v eines DAG heißt Wurzel*, falls es keine auf ihn gerichteten Kanten gibt. Hat ein DAG nur eine Wurzel, so heißt er* Wurzelgraph*. Man kann in diesem Fall von seiner Wurzel sprechen.*

Definition 1.3.4 (Topologische Ordnung)
Gegeben sei ein gerichteter Graph $G = (N, E)$. Eine Folge $v_1, \ldots, v_n$ ($v_i \in N$) aller Knoten aus N ist eine topologische Ordnung*, falls*

$$i < j \succ \text{es existiert kein Pfad von } v_j \text{ nach } v_i.$$

Wir bezeichnen die topologische Ordnung eines Graphen G mit $T(G)$.

Proposition 1.3.5 *Ein gerichteter Graph hat eine topologische Ordnung $:\prec\succ$ er ein DAG ist.*

Definition 1.3.6 (Transitive Hülle)
Ein Graph $G^+(N, E^+)$ ist eine transitive Hülle *eines Graphen $G = (N, E)$ $:\prec\succ$*

$$v \to_{G^+} v' \prec\succ v \to_G v'.$$

Proposition 1.3.7 *G^+ ist ein DAG $\prec\succ$ G ein DAG ist.*

Definition 1.3.8
Ist $G = G^+$, so heißt G transitiv abgeschlossen*.*

Definition 1.3.9
Ein Baum *ist ein Wurzelgraph, in dem zu jedem Knoten genau ein Pfad von der Wurzel aus führt.*

Pfeilspitzen laufen also von “oben” nach “unten”. In einem Baum haben alle Knoten außer der Wurzel einen eindeutigen Vorgänger.

1.4 Partielle Ordnung

Definition 1.4.1 (Partielle Ordnung)
Eine partielle Ordnung *ist ein Paar $L = (\Sigma, <)$, wobei Σ eine Menge ist und $<$ eine irreflexive transitive binäre Relation auf $\Sigma \times \Sigma$. Σ wird auch* Domäne *oder* Bereich *der partiellen Ordnung genannt.*

Seien $a, b \in \Sigma$. Falls $a < b$, so ist a *vor* b und b *nach* a; b *folgt* a. Gilt weder $a < b$ noch $b < a$, so heißen a und b *unvergleichbar*.

Definition 1.4.2 (Einschränkung)
Eine partielle Ordnung $L' = (\Sigma', <')$ heißt Einschränkung *einer partiellen Ordnung $L = (\Sigma, <)$ auf den Bereich Σ' $:\prec\succ$*

1. *$\Sigma' \subseteq \Sigma$ und*

2. $\forall a, b \in \Sigma' \; a <' b \prec\succ a < b$.

Definition 1.4.3 (Präfix)
Eine partielle Ordnung $L' = (\Sigma', <')$ heißt Präfix *einer partiellen Ordnung $L = (\Sigma, <)$ (notiert als $L' \leq L$) $:\prec\succ$*

1. *L' eine Einschränkung von L ist und*

2. $\forall a \in \Sigma' \; \forall b \in \Sigma \; b < a \succ b \in \Sigma'$.

Durch Identifikation der Elemente aus Σ mit den Knoten N eines DAGs $G = (N, E)$ kann jede partielle Ordnung als DAG aufgefasst werden mit $(a, b) \in N \prec\succ a < b$. Es gibt keine Probleme mit der Zyklenfreiheit, da eine partielle Ordnung transitiv geschlossen und irreflexiv ist. (Umkehrung gilt natürlich auch.)

Kapitel 2

Einleitung

2.1 Motivation

- Viele Benutzer teilen sich eine Datenbank
- Unerwünschte Seiteneffekte durch Nebenläufigkeit sollen vermieden werden

Beispiele:

- Kontenverwaltung
- Platzreservierung

2.2 Transaktion

2.2.1 Syntax

Eine Transaktion klammert eine Menge von Operationen durch

- **begin transaction** und
- **end transaction** oder
- **abort**

Wir benutzen abkürzend b, c und a.

2.2.2 Semantik

ACID:

- atomicity
- consistency
- isolation
- durability

Einheit der Interaktion, der Konsistenz und der Recovery.

- concurrency control (Nebeläufigkeitskontrolle)
- recovery (Wiederanlauf)

2.2.3 Nur Lesen und Schreiben

Operationen:

- read
- write

Beispiel:

Step	TA_1	TA_2
1.	temp := x.read;	
2.		temp := x.read;
3.	temp := temp+1;	
4.		temp := temp+1;
5.	x.write(temp);	
6.		x.write(temp);

$$x.read_1; x.read_2; x.write_1(val_1); x.write_2(val_2);$$

2.2.4 Einfluss der Einschränkung auf ACID

Konsistenzbegriff eingeschränkt, da höhere Semantik dem Datenbanksystem nicht bekannt. Diese muss durch Anwendung sichergestellt werden. Dadurch Einschränkung des Konsistenzbegriffs wie folgt:

Konsistenzprobleme:

- no lost updates
- no dirty reads
- repeatable reads

Konsistenzebenen:

Level 1: no lost updates

Level 2: level 1 + no dirty reads

Level 3: level 2 + repeatable reads

kein kaskadierendes Rücksetzen

Beispiel 2.2.1
Beispiele

2.3 Wiederanlauf

1. Ausfall (failure)

   (a) Persistenz von abgeschlossenen Transaktionen (nach *commit*)

   (b) Aufheben von Effekten partiell ausgeführter Transaktionen (vor *commit*)

2. Abbruch (abort)

   (a) Ts Effekt muss rückgängig gemacht werden

   (b) Falls T ein x beschrieben hat und ein T' es gelesen hat, so muss T' zurückgesetzt werden (transitive Hülle dieser Beziehung).

   Letzteres wird als *kaskadierendes Rcksetzen* bezeichnet.

Beispiel 2.3.1
Betrachte folgende Anweisungsfolge/Ausführung, in der die Subscripte der Operationen die zugehörige Transaktion bezeichnen:

$x.write_1(1);$
$x.read_2;$
$y.write_2(2);$
$y.read_3;$

Falls T_1 abgebrochen wird, muss auch T_2 zurückgenommen werden. Dann auch T_3.

Problem:
Eine Transaktion, die committet ist, muss persistent sein, kann also nicht wieder zurückgenommen werden. Falls sie von einer Transaktion abhängt, die noch nicht committet ist (bspw. hat sie Daten gelesen, die letztere geschrieben hat), so kann sie also auch nicht committen. Betrachte dazu folgende Sequenz/Ausführung:

$b_1; b_2; x.write_1(1); x.read_2; c_2; \ldots$

Definition 2.3.1 ($\mathbf{T}_i$ liest von $\mathbf{T}_j$)
Seien T_i und T_j Transaktionen. T_i liest x von *T_j, falls*

1. *es ein Datenelement x gibt, so dass ein $x.write_j(\cdot)$ vor einem $x.read_i$ ausgeführt wurde,*

2. *T_j nicht vor dem ersten Lesen von x abbricht und*

3. *jede Transaktion, die x beschreibt, nachdem T_j es beschreibt und bevor T_i es liest, abbricht.*

Eine Transaktion T_i liest von *T_j, falls es ein Datenelement x gibt, so dass T_i x von T_j liest. (Notiert als $T_i \rhd T_j$ bzw. $T_i \rhd_x T_j$.)*

Definition 2.3.2 *[wiederanlaufbar]*
Eine Ausführung ist wiederanlaufbar *(*recoverable*) falls für jede Transaktion T_j, die abschließt (committet), jede Transaktion T_i mit $T_i \lhd T_j$ vor T_j abschließt.*
Anders ausgedrückt:

$$T_i \lhd T_j \quad \succ \quad c_i < c_j$$

Dabei verstehen wir unter einer Ausführung die verzahnte Ausführung der Operationen verschiedener Transaktionen. Eine formale Definition hierzu folgt noch.

Beispiel 2.3.2
Betrachte folgende Beispiele:

$$x.write_1(1); x.read_2; y.write_2(2); c_1; \ldots ; c_2$$
$$x.write_1(1); x.read_2; y.write_2(2); c_2; \ldots ; c_1$$

Die erste Ausführung ist wiederanlaufbar, die zweite nicht. Im zweiten Beispiel hätte das DBMS die Anweisung c_2 bis nach c_1 zu verzögern.

Bemerkung 2.3.3 *Bildschirmausgaben können nicht rückgängig gemacht werden.*

Definition 2.3.4
Eine Ausführung vermeidet kaskadierendes Rücksetzen*, falls in ihr nur Datenelemente gelesen werden, die von bereits abgeschlossenen Transaktionen geschrieben wurden.*
Anders ausgedrückt:

$$T_i \lhd_x T_j \quad \succ \quad c_i < x.read_j$$

Dies kann erreicht werden, falls das DBMS alle Leseoperationen verzögert, bis die Transaktion, die das zu lesende Datenelement geschrieben hat, abschließt.

Bemerkung 2.3.5 (Zusammenhang kaskadierendes Rücksetzen/wiederanlaufbar)

Es gilt:

> *Eine Ausführung, die kaskadierendes Rücksetzen vermeidet, ist wiederanlaufbar.*

Dies ist offensichtlich, da x.read$_j$ $< c_j$ *gelten muss.*
Es gilt **nicht***:*

> *Wiederanlaufbar* $\succ$ *vermeidet kaskadierendes Rücksetzen.*

Betrachte

$$x.write_1(1); x.read_2; y.write_2(2); a_1$$

Diese Auführung ist wiederanlaufbar (keine Aussagen über commits). Aber: T_2 muss abgebrochen werden, da die resultierende Ausführung sonst nicht mehr wiederanlaufbar wäre. Also:

Um Wiederanlaufbarkeit zu gewährleisten, müssen unter Umständen Abbrüche stattfinden.

Die Forderung nach Vermeidung kaskadierenden Rücksetzens ist nicht immer scharf genug. Betrachte

$$x.write_1(1); y.write_1(3); y.write_2(1); c_1; x.read_2; a_2;$$

Entfernen von T_2 ergibt:

$$x.write_1(1); y.write_1(3); c_1;$$

Es gilt $y = 3$ nach Ausführung. Dies ist also der Wert, auf den zurückgesetzt werden muss, falls T_2 abbricht. Wir bezeichnen den Wert, den ein Datenelement *direkt* vor einer Schreibanweisung besitzt, als *before image*. Diese werden üblicherweise benutzt, um das Rücksetzen zu implementieren. In unserem Beispiel war es genau das before image von $y.write_2(1)$, das benutzt wurde, um den Wert von y zurückzusetzen.

Unglücklicherweise ist dies nicht immer möglich, wenn nicht gewisse Anforderungen an eine Ausführung gestellt werden. Betrachte

$$x.write_1(1); x.write_2(2); a_1; a_2;$$

Annahme: $x = 0$ vor der Ausführung von $x.write_1(1)$. Dies sollte also auch der Wert sein, den x bei Rücksetzen von T_2 erhält. Dies ist aber nicht der Fall, wenn man das before image von $x.write_2(2)$, nämlich $x = 1$ benutzt.

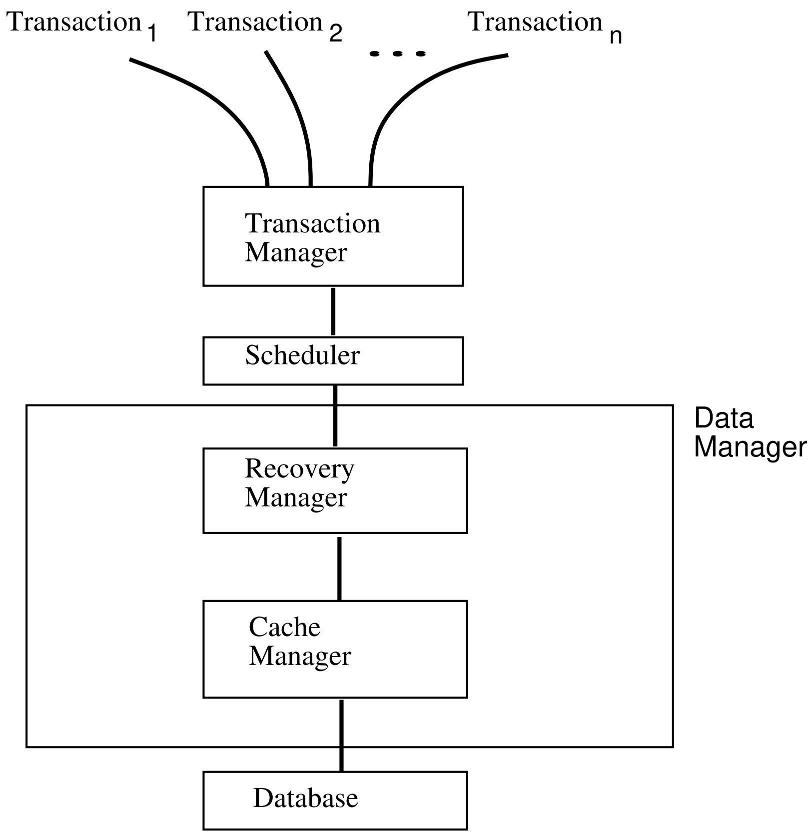
Ähnlich wie bei der Vermeidung kaskadierenden Rücksetzens, wo Leseoperationen bis nach dem Beenden einer Transaktion verzögert wurden, lässt sich auch dieses Problem durch Verzögern der Schreiboperationen lösen.

Definition 2.3.6
Eine Ausführung heißt strikt *gdw*

1. $i \neq j \succ$
   $x.write_i(\cdot) < x.read_j \succ c_i \vee a_i < x.read_j(\cdot)$

2. $i \neq j \succ$
   $x.write_i(\cdot) < x.write_j(\cdot) \succ c_i \vee a_i < x.write_j(\cdot)$

2.4 Abstrakte DBMS-Architektur



Transaction 1
Transaction 2
Transaction n
Transaction Manager
Scheduler
Data Manager
Recovery Manager
Cache Manager
Database

Kapitel 3

Serialisierbarkeit

Das Thema dieses Kapitels ist die Serialisierbarkeitstheorie. Sie wird uns die Grundlagen liefern, um die Korrektheit von Schedulern beurteilen zu können.

3.1 Historien

Wir haben bereits gesagt, dass wir Transaktionen nur als Folgen von Lese- und Schreiboperationen modellieren werden. Bisher schrieben wir *read* und *write*. Wir werden jetzt nur noch die Anfangsbuchstaben verwenden. Weiterhin bedeuten die Subskripte die zugehörige Transaktion. Also:

- $r_i[x]$ bedeutet, dass Transaktion i das Datenelement x liest.
- $w_i[x]$ bedeutet, dass Transaktion i das Datenelement x schreibt.

Um auszudrücken, dass eine Transaktion eine Operation vor einer anderen ausführt, verwenden wir $\rightarrow$. Bsp:

$$r_i[x] \rightarrow w_i[x] \rightarrow c_i$$

Da eine Transaktion auch parallele Prozesse umspannen kann, muss keine totale Ordnung vorliegen. Es ist also auch denkbar:

$$\begin{tikzcd} r_i[x] \arrow[dr] & & \\ & w_i[z] \arrow[r] & c_i \\ r_i[y] \arrow[ur] & & \end{tikzcd}$$

um bspw. die Summe von x und y in z zu speichern.

Falls eine Transaktion i das gleiche Datenelement x liest und schreibt, muss entweder $r_i[x] \rightarrow w_i[x]$ oder $w_i[x] \rightarrow r_i[x]$ gelten. Warum? Weil der Wert von $r_i[x]$ davon abhängt, ob x vorher geschrieben wurde oder nicht.

Wir werden Transaktionen als partielle Ordnungen $(\Sigma_i, <_i)$ modellieren. Dabei ist der Grundbereich überflüssig, da er aus der Transaktion herleitbar ist: Er umfasst alle Operationen (r und w) von Transaktion i. Falls eine Operation $r_i[x]$ in Transaktion i vorkommt, schreiben wir $r_i[x] \in T_i$. (Analog für w.)

Definition 3.1.1 (Transaktion) *Eine* Transaktion T_i *ist eine partielle Ordnung* $<_i$*, wobei*

1. $T_i \subseteq \{r_i[x], w_i[x] | x \quad \textit{ist ein Datenelement}\} \cup \{a_i, c_i\}$
2. $a_i \in T_i \quad \prec\succ \quad c_i \notin T_i$
3. $\forall t \in T_i \; t = c_i \vee t = a_i \succ (\forall s \in T_i \; s <_i t)$
4. $r_i[x], w_i[x] \in T_i \succ (r_i[x] <_i w_i[x] \vee w_i[x] <_i r_i[x])$

Zur Veranschaulichung werden wir Transaktionen oft als DAG zeichnen.(bsp)

Wir haben nicht erfasst:

- before images
- after images
- alle anderen Operationen, Zuweisungen, Kontrollanweisungen
- Ein/Ausgabe

In den folgenden Beweisen müssen wir also aufpassen, dass wir keine impliziten Annahmen über diese Dinge machen. Insbesondere müssen unsere Beweise für alle before images und after images gelten.

Bspw. können wir nicht sagen, ob für

$r_i[x] \rightarrow w_i[x] \rightarrow c_i$
oder
$r_i[x] \rightarrow w_i[y] \rightarrow c_i$

der Wert, der geschrieben wird, vom gelesenen Wert abhängt oder nicht. Wir nehmen daher an, dass ein geschriebener Wert von allen voher gelesenen Werten abhängt.

Bemerkung 3.1.2 *x kann nicht nur ein Integer oder Datenfeld sein, sondern, sogar sehr gebräuchlich, eine Seite. Wie stellt man fest, ob so eine Seite gelesen oder geschrieben wird, ohne jedesmal ein entsprechende Funktion aufzurufen, das heißt, ein bestimmtes Protokoll zu fahren? Hier gibt es Betriebssystemunterstützung für r/w-Rechte (Memory Management). Werden die gesetzten Zugriffsrechte verletzt, dann wird ein Interrupt erzeugt, der dann durch das DBMS abgefangen werden kann.*

Wenn eine Menge von Transaktionen parallel ausgeführt wird, so ist die Ausführung verzahnt/verschachtelt in dem Sinne, dass Operationen verschiedener Transaktionen gemischt werden. Eine solche Ausführung nennen wir Historie. Man beachte, dass auch hier einige Operationen nebenläufig ausgeführt werden können. Auch hier handelt es sich also um eine partielle Ordnung. Wir benötigen noch eine Zusatzdefinition, bevor wir Historien definieren können:

Definition 3.1.3 (unverträglich)
Zwei Operationen p und q sind unverträglich (kommutieren nicht)*, falls beide auf das gleiche Datenelement zugreifen und mindestens eine von ihnen ein Schreibzugriff ist. Wir notieren dies als $q \not\parallel p$.*

Beachte: Unverträglichkeit ist symmetrisch.

Definition 3.1.4 (Historie)
Sei $T = \{T_1, \ldots, T_n\}$ eine Menge von Transaktionen. Eine vollständige Historie über T *ist eine partielle Ordnung $<_H$ mit:*

1. $H = \cup_{i=1}^{n} T_i$
2. $\cup_{i=1}^{n} <_i \subseteq <_H$
3. $\forall p, q \in H \ p \not\parallel q \succ p <_H q \vee q <_H p$

Eine Historie *ist ein Präfix einer vollständigen Historie.*

Bemerkung 3.1.5 *ad 1: Die Ausführung der Historie H muss genau alle Operationen aus den T_i umfassen, nicht mehr und nicht weniger.*

ad 2: Die Ordnungsrelation von H, $<_H$, muss mindestens alle Operatoren ordnen, die auch schon in mindestens einem der $<_i$ einer T_i enthalten waren.

ad 3: Alle unverträglichen Operationen müssen durch $<_H$ geordnet sein.

Historie: Eine Historie ist also eine möglicherweise unvollständige Ausführung. Dies ist notwendig, um Systemabstürze und andere Unarten behandeln zu können.

Wir betrachten nun ein Beispiel:

Beispiel 3.1.1 *Gegeben sei $T = \{T_1, T_3, T_4\}$ mit:*

$$
\begin{array}{rcl}
T_1 & = & r_1[x] \rightarrow w_1[x] \rightarrow c_1 \\
T_3 & = & r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_3 \\
T_4 & = & r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_4
\end{array}
$$

Eine *vollständige Historie H_1 über T ist:*

$$
\begin{array}{ccccccccccc}
 & & & & r_3[x] & \rightarrow & w_3[y] & \rightarrow & w_3[x] & \rightarrow & c_3 \\
 & & & & \uparrow & & \uparrow & & & & \\
H_c & = & r_4[y] & \rightarrow & w_4[x] & \rightarrow & w_4[y] & \rightarrow & w_4[z] & \rightarrow & c_4 \\
 & & & & \uparrow & & & & & & \\
 & & r_1[x] & \rightarrow & w_1[x] & \rightarrow & c_1 & & & &
\end{array}
$$

Eine *Historie über T, die gleichzeitig ein Präfix von H_1 ist, ist:*

$$
\begin{array}{ccccccc}
 & & & & r_3[x] & \rightarrow & w_3[y] \\
 & & & & \uparrow & & \uparrow \\
H_c & = & r_4[y] & \rightarrow & w_4[x] & \rightarrow & w_4[y] \\
 & & & & \uparrow & & \\
 & & r_1[x] & \rightarrow & w_1[x] & \rightarrow & c_1
\end{array}
$$

Man beachte, dass wir keine transitiven Pfeile gezeichnet haben. Diese werden wir uns auch in Zukunft schenken. Oftmals werden wir auch die Pfeile innerhalb einer Folge weglassen. Wir schreiben also beispielsweise statt

$$r_1[x] \rightarrow w_2[y] \rightarrow w_1[x] \rightarrow c_2$$

nur noch

$$r_1[x]w_2[y]w_1[x]c_2.$$

Definition 3.1.6
Eine Transaktion T_i heißt beendet *in einer Historie H, falls $c_i \in H$. Sie heißt* abgebrochen *in H, falls $a_i \in H$. Sie heißt* aktiv *in H, falls sie weder beendet noch abgebrochen ist.*

Definition 3.1.7 (abgeschlossene Projektion)
Sei H eine Historie über $T = \{T_1, \ldots, T_n\}$. Dann ist die abgeschlossene Projektion *$(C(H))$ von H definiert als:*

$$C(H) \quad := \quad H|_{\cup_{c_i \in H,\ i=1,\ldots,n} T_i}$$

$C(H)$ besteht also nur noch aus den Operationen abgeschlossener Transaktionen.

Bemerkung 3.1.8
Beachte: $C(H)$ ist eine vollständige Historie für alle in H abgeschlossenen Transaktionen.

3.2 Serialisierbare Historien

Historien modellieren (repräsentieren) nebenläufige Ausführungen einer Menge von Transaktionen. Eine Historie ist seriell, wenn alle Transaktionen hintereinander ausgeführt werden. Also:

Definition 3.2.1 (serielle Historien)
Eine Historie H über $T = \{T_1, \ldots, T_n\}$ heißt seriell *(S) $:\prec\succ$*

$$\forall 1 \leq i, j \leq n \ i \neq j \succ (\forall p_i, p_j \ p_i <_H p_j) \vee (\forall p_i, p_j \ p_j <_H p_i)$$

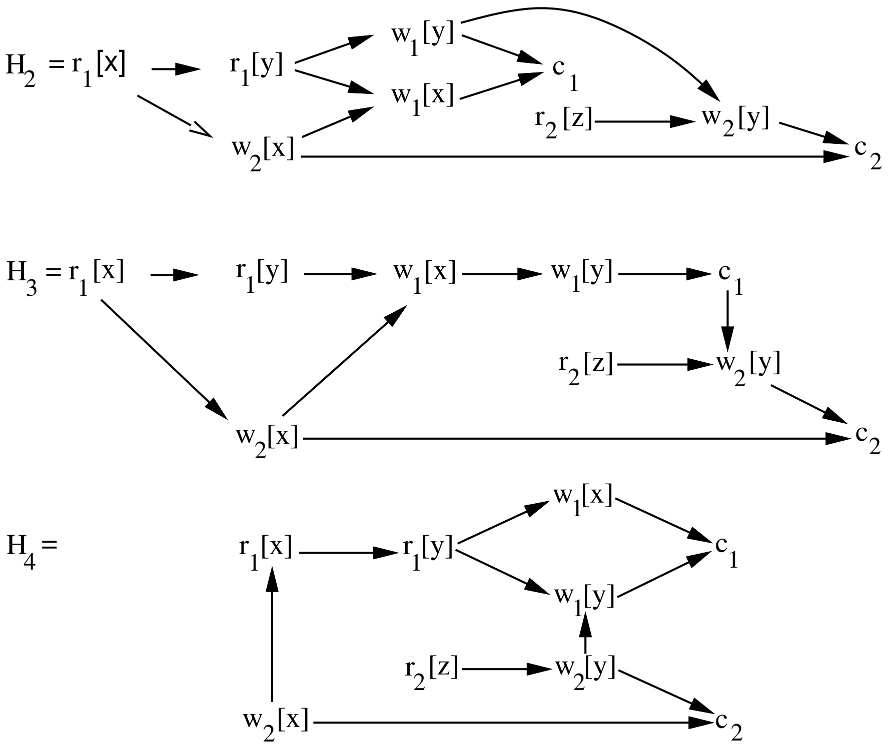
Eine serielle Historie über T werden wir oft als $T_{i_1}, \ldots, T_{i_n}$ bezeichnen, wobei die i_j eine Permutation von $1 \ldots, n$ bedeuten.

Bei seriellen Historien kann nichts schief gehen, sie sind für uns der Inbegriff einer korrekten Historie. Wir werden im folgenden den Begriff der serialisierbaren Historie definieren und zeigen, dass mehr Historien serialisierbar sind als seriell, aber die serialisierbaren Historien äquivalent zu den seriellen sind, also insbesondere korrekt. Dies begründet unser Interesse an serialisierbaren Historien.

Wir beginnen mit der Definition der Äquivalenz zweier Historien. Die zentrale Idee hierbei ist, dass das Ergebnis einer Historie nur von der Ordnung der unverträglichen Operationen abhängt. Wenn diese also in zwei Historien gleich geordnet sind, erzeugen diese beiden Historien dasselbe Ergebnis.

Definition 3.2.2 ((Konflikt-) Äquivalenz)
Zwei Historien H und H' heißen (konflikt-) äquivalent $:\prec\succ$

1. *Beide sind über der gleichen Transaktionsmenge T definiert,*
2. *beide umfassen die gleiche Menge von Operationen und*



3. *für alle Operationen $p_i \in T_i$ und $p_j \in T_j$ mit $p_i \not\parallel p_j$ gilt:*
   $a_i, a_j \notin H, p_i <_H p_j \ \succ \ p_i <_{H'} p_j$

Beachte: Die letzte Bedingung impliziert $p_i <_H p_j \ \prec\succ \ p_i <_{H'} p_j$

Beispiel 3.2.1
Beispielhistorien über $t = \{T_1, T_2\}$:

$$
\begin{array}{cccccccccccccc}
 & & & & & & w_1[y] & \rightarrow & \rightarrow & \rightarrow & \rightarrow & & & \\
 & & & & & \nearrow & & \searrow & & & & & & \\
H_2 & = & r_1[x] & \rightarrow & r_1[y] & & & & c_1 & & \downarrow & & & \\
 & & & & & \searrow & & \nearrow & & & & & & \textit{und} \\
 & & & & & & w_1[x] & & & & & & & \\
 & & & \searrow & & & & & r_2[z] & \rightarrow & w_2[y] & \searrow & & \\
 & & & & w_2[x] & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & c_2 &
\end{array}
$$

andere Bilder (BeHaGoFig2-2p31).

Für diese Historien H_2, H_3 und H_4 gilt:

$$
\begin{array}{ccc}
H_2 & \equiv & H_3 \\
H_4 & \not\equiv & H_2 \\
H_4 & \not\equiv & H_3
\end{array}
$$

Definition 3.2.3 (Serialisierbarkeit)
Eine Historie H heißt serialisierbar*, falls eine serielle Historie H_s existiert, so dass*

$C(H) \equiv H_s$

Wir kürzen serialisierbar oft als SR ab.

Warum enthält diese Definition eine etwas komplexere Bedingung, als die vielleicht erwartete Bedingung

$$H \equiv H_s \ ?$$

Hierfür gibt es zwei Gründe. Der erste ist technischer Natur, der zweite wichtig für unsere Ziele:

1. Äquivalente Historien müssen die gleichen Operationen umfassen. Damit wäre die Definition nur für vollständige Historien geeignet.

2. Eine nichtvollständige Historie enthält unvollständig ausgeführte Transaktionen. Unvollständig ausgeführte Transaktionen erhalten aber nicht notwendigerweise (oder in den seltensten Fällen) die Konsistenz.

Man erinnere sich, dass die abgeschlossene Projektion einer Historie eine vollständige Historie ist.

3.3 Das Serialisierbarkeitstheorem

Wir werden jetzt einen einfachen Test angeben, mit dem bestimmt werden kann, ob eine Historie serialisierbar ist. Dazu benötigen wir:

Definition 3.3.1 (Serialisierbarkeitsgraph(SG))
Sei H eine Historie über $T = \{T_1, \ldots, T_n\}$. Der Serialisierbarkeitsgraph $SG(H)$ ist ein gerichteter Graph $SG(H) = (N, E)$ mit

1. $N = \{T_i | T_i \in T, c_i \in H\}$

2. $E = \{T_i \rightarrow T_j | \exists p_i \in T_i, p_j \in T_j (i \neq j) \ p_i \not\parallel p_j \ \wedge \ p_i <_H p_j\}$

Eine Kante $T_i \rightarrow T_j$ bedeutet hier, dass zumindest eine Operation aus T_i vor einer Operation aus T_j ausgeführt werden sollte. Falls die Historie H äquivalent zu einer seriellen Historie sein soll, so sollte in dieser seriellen Historie T_i vor T_j ausgeführt werden. Dies geht natürlich nur solange gut, wie $SG(H)$ keinen Zyklus enthält. Allgemein gilt:

Satz 3.3.2 (Serialisierbarkeitstheorem)
Eine Historie ist serialisierbar $\prec\succ$ $SG(H)$ ist azyklisch.

Beweis:

$\prec$: Sei H eine Historie über $T = \{T_1, \ldots, T_n\}$.
oBdA: $C(H) = T' = \{T_1, \ldots, T_m\}$, $m \leq n$
$\succ$ Die Menge aller Knoten von $SG(H)$ ist T'
$SG(H)$ azyklisch
$\succ^{1.3.5}$ Es existiert topologische Ordnung von $SG(H)$.
Diese sei: $T_{i_1}, \ldots, T_{i_j} =: H_s$.
z.z: $H_s \equiv H$.

Bedingung 1 und 2 sind klar.
zu 3: Seien $c_i, c_j \in H$, $p_i \in T_i$, $p_j \in T_j$ mit $p_i \nparallel p_j$.
$\succ^{3.1.4}$ $p_i <_H p_j$.
$\succ^{3.3.1}$ $T_i \rightarrow_{SG(H)} T_j$
$\succ$ $\forall T(SG(H))$ $T_i <_{T(SG(H))} T_j$
$\succ$ $\forall T(SG(H))$ $p_i <_{T(SG(H))} p_j$
$\succ$ $p_i <_{H_s} p_j$
Also $H_s \equiv H$. Da H_s nach Konstruktion seriell, gilt: H ist serialisierbar.

$\succ$: Sei H serialisierbar, H_s eine serielle Historie und $H \equiv H_s$.
Betrachte $T_i \rightarrow T_j \in SG(H)$.
$\succ$ $\exists p_i \in T_i, p_j \in T_j$ $p_i <_H p_j$
$\succ$ $(C(H) \equiv H_s)$ $p_i <_{H_s} p_j$
$\succ$ (H_s seriell) $T_i <_{H_s} T_j$
Falls nun $SG(H)$ einen Zyklus hat, so auch H_s. Widerspruch.
Also: $SG(H)$ ist DAG.

$\square$

Im $\prec$-Teil des Beweises haben wir gesehen: Falls eine vollständige H azyklisch ist, dann ist H äquivalent zu jeder seriellen Historie, die eine topologische Ordnung von H ist. Da mehrere solche existieren können, kann H zu mehreren seriellen Historien äquivalent sein.

Weiter folgt, dass Serialisierbarkeit in polynomialer Zeit berechnet werden kann.

Wir betrachten nun einige Beispiele:

Beispiel 3.3.1
Die Historie

$$
\begin{array}{ccccccccccc}
 & & & & & & r_3[x] & \rightarrow & w_3[x] & \rightarrow & c_3 \\
 & & & & & \nearrow & & & & & \\
H_5 & = & r_1[x] & \rightarrow & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 & & \\
 & & & \nearrow & & \nearrow & & & & & \\
 & & r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 & & & &
\end{array}
$$

hat den Serialisierbarkeitsgraphen

$$
\begin{array}{ccccccc}
 & & & \rightarrow & \rightarrow & & \\
 & & \nearrow & & & \searrow & \\
SG(H_5) & = & T_2 & \rightarrow & T_1 & \rightarrow & T_3
\end{array}
$$

H_5 ist also serialisierbar, also äquivalent zu einer seriellen Historie. Hier ist T_2, T_1, T_3 die einzige.

Man beachte, dass im Allgemeinen $T_i \rightarrow T_j, T_j \rightarrow T_k \not\succ T_i \rightarrow T_k$. Ersetzt man beispielsweise in H_5 $w_3[x]$ durch $w_3[z]$, so ist

$$SG(H_5) \quad = \quad T_2 \quad \rightarrow \quad T_1 \quad \rightarrow \quad T_3,$$

da nun keine Unverträglichkeiten zwischen T_2 und T_3 mehr existieren.

Beispiel 3.3.2
Die Historie

$$H_6 = w_1[x]w_1[y]c_1r_2[x]r_3[y]w_2[x]c_2w_3[y]c_3$$

hat den Serialisierbarkeitsgraphen

$$\begin{array}{ccccccc} & & \nearrow & \rightarrow & \rightarrow & \searrow & \\ SG(H_6) & = & T_1 & \rightarrow & T_3 & & T_2 \end{array}$$

Für diesen existieren zwei topologische Ordnungen, nämlich:

1. T_1, T_2, T_3
2. T_1, T_3, T_2

Folgerichtig ist H_6 äquivalent zu beiden.

3.4 Wiederanlaufbare Historien

Man erinnere sich an das vorangegangene Kapitel, in dem Überlegungen über günstige Eigenschaften für den Wiederanlauf einer Transaktion zum Begriff der strikten Ausführung führte. Wir werden dies jetzt mit Historien formalisieren.

Wir formalisieren zunächst die Definitionen 2.3.1, 2.3.2, 2.3.4 und 2.3.6.

Definition 3.4.1 (liest)
Seien T_i und T_j Transaktionen und H eine Historie mit $T_i, T_j \in H$. T_i liest *x* von *T_j, wenn*

- $\exists x \; w_j[x] <_H r_i[x]$
- $a_j \not<_H r_i[x]$
- $\forall k \neq i, j$: $\forall \; w_k[x] \quad w_j[x] <_H w_k[x] <_H r_i[x] \succ a_k <_H r_i[x]$.

Eine Transaktion T_i liest von *T_j, falls es ein Datenelement x gibt, so dass T_i x von T_j liest. (Notiert als $T_i \rhd_H [x]T_j$ bzw. $T_i \rhd_H T_j$.)*

Definition 3.4.2 (wiederanlaufbar/rücksetzbar)
Eine Historie H heißt wiederanlaufbar *(*rücksetzbar, recoverable*, RC), falls*

- $T_i \rhd_H T_j, c_i \in H \succ c_j <_H c_i$

für alle Transaktionen T_i und T_j $(i \neq j)$ aus H.

Definition 3.4.3 (vermeidet kaskadierendes Rücksetzen)
Eine Historie H vermeidet kaskadierendes Rücksetzen *(ACA), falls*

- $T_i \rhd_H [x]T_j \succ c_j <_H r_i[x]$

für alle Transaktionen T_i und T_j $(i \neq j)$ aus H.

Man beachte, dass für alle Transaktionen gilt: $r_i[x] < c_i$.

Definition 3.4.4 (strikt)
Seien T_i und T_j $(i \neq j)$ Transaktionen und H eine Historie mit $T_i, T_j \in H$. Eine Historie H heißt strikt *(ST), falls*

- $w_j <_H p_i[x] \succ a_j <_H p_i[x] \vee c_j <_H p_i[x]$

für alle Transaktionen T_i und T_j $(i \neq j)$ aus H, $p \in \{r, w\}$.

Wir haben alle Definitionen auch mit Kürzeln (SR, RC, ACA, ST) versehen. Diese werden nicht nur dazu verwendet, um den entsprechenden Ausdruck zu verkürzen, sondern auch, um die entsprechende Klasse zu bezeichnen. Wir bezeichen also bspw. mit SR die Menge aller serialisierbaren Historien.

Wir betrachten jetzt einige Beispiele, an denen die obigen Definitionen und ihre Beziehungen zueinander beleuchtet werden.

Beispiel 3.4.1
Sei $T = \{T_1, T_2\}$ mit

$$\begin{aligned} T_1 &= w_1[x]w_1[y]w_1[z]c_1 \\ T_2 &= r_2[u]w_2[x]r_2[y]w_2[y]c_2 \end{aligned}$$

Dazu betrachten wir folgende Historien:

$$\begin{array}{lcccccccccc} H_7 & = & w_1[x] & w_1[y] & r_2[u] & w_2[x] & r_2[y] & w_2[y] & c_2 & w_1[z] & c_1 \\ H_8 & = & w_1[x] & w_1[y] & r_2[u] & w_2[x] & r_2[y] & w_2[y] & w_1[z] & c_1 & c_2 \\ H_9 & = & w_1[x] & w_1[y] & r_2[u] & w_2[x] & w_1[z] & c_1 & r_2[y] & w_2[y] & c_2 \\ H_{10} & = & w_1[x] & w_1[y] & r_2[u] & w_1[z] & c_1 & w_2[x] & r_2[y] & w_2[y] & c_2 \end{array}$$

Zunächst sei angemerkt, dass alle obigen Historien vollständig sind. Weiter gilt folgendes:

	S	SR	RC	ACA	ST
H_1					
H_2					
H_3					
H_4					
H_5					
H_6					
H_7	+		-		
H_8	+		+	-	
H_9	+		+	+	-
H_{10}	+		+	+	+

Bemerkung: Wir vereinbaren $A \subset B \succ A \neq B$.

Satz 3.4.5

$ST \subset ACA \subset RC$

Beweis:

$ST \subset ACA$**:** Sei $H \in ST$ und $T_i, T_j \in H$ $(i \neq j)$.
Annahme: $T_i \rhd_H [x]T_j$
$\succ^{3.4.1}$ $w_j[x] <_H r_i[x]$ für ein x und $a_j \not<_H r_i[x]$
$\succ^{3.4.4}$ $c_j <_H r_i[x]$
$\succ^{3.4.3}$ $H \in ACA$
Die Echtheit der Teilmengenbeziehung folgt mit H_9.

$ACA \subset RC$**:** Sei $H \in ACA$ und $T_i, T_j \in H$ $(i \neq j)$.
Annahme: $T_i \rhd_H [x]T_j$ und $c_i \in H$.
$\succ^{3.4.3}$ $c_j <_H r_i[x]$.
Da $c_i \in H$, gilt $r_i[x] <_H c_i$ und damit $c_j <_H c_i$.
Also: $H \in RC$
Die Echtheit der Teilmengenbeziehung folgt mit H_8.

□

Bemerkung 3.4.6
Über die Beziehungen zwischen SR und RC, ACA und ST kann man sagen, dass der Schnitt zwischen SR und X für $X \in \{RC, ACA, ST\}$ nicht leer, aber SR auch unvergleichbar zu X ist, also keinerlei Teilmengenbeziehung gilt. Ein Scheduler muss daher Serialisierbarkeit und Wiederanlaufbarkeit bzw. Vermeidung von kaskadierendem Rücksetzen oder Striktheit getrennt garantieren.

Wir fassen die Ergebnisse in folgendem Venn-Diagramm zusammen:

Für den Bau eines realen Scheduler ist die folgende Eigenschaft von Historien von entscheidender Bedeutung:

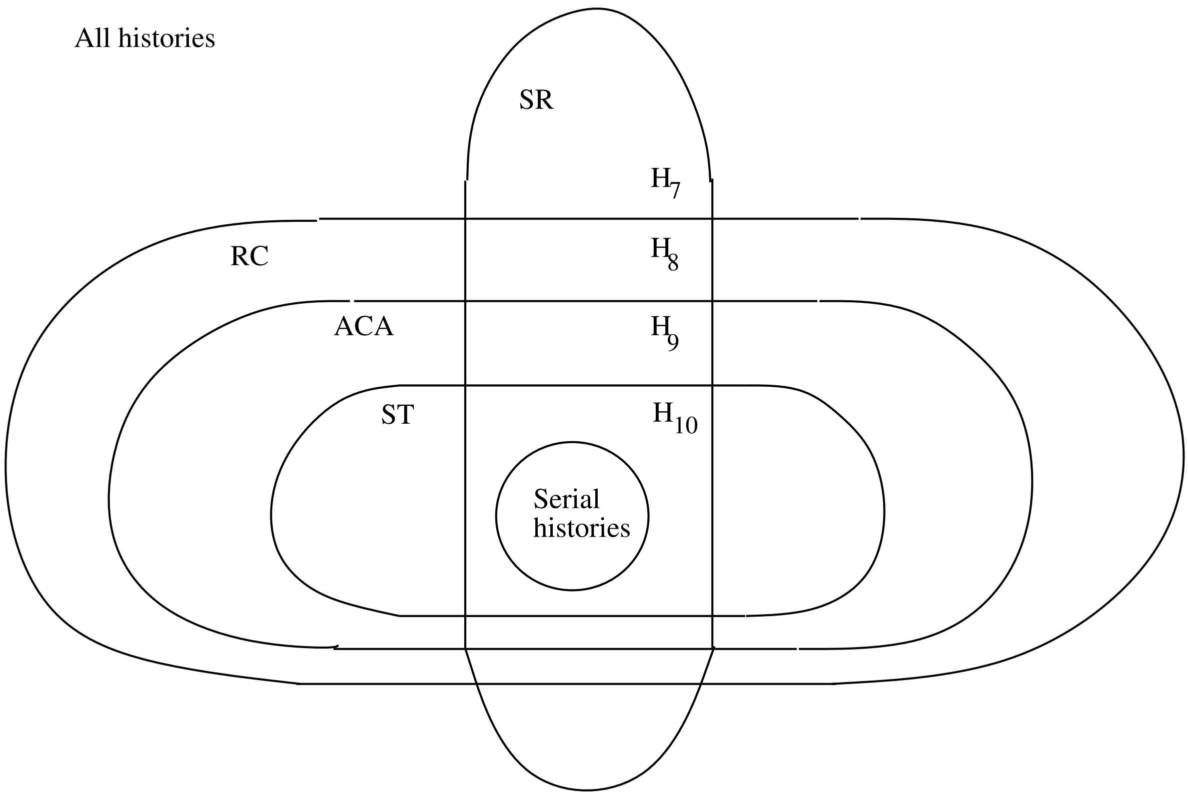
Definition 3.4.7 (Präfix-Commit-Abgeschlossenheit)
Sei $\mathcal{E}$ eine Eigenschaft von Historien. $\mathcal{E}$ heißt präfix-commit-abgeschlossen *(pcc) , falls*

$$\forall H \; H \in \mathcal{E} \succ \forall H' \leq H \; C(H') \in \mathcal{E}$$

Satz 3.4.8 *SR, RC, ACA und ST sind pcc.*

Beweis:
Wir zeigen nur: SR ist pcc. (Rest ist Übungsaufgabe.)
Annahme: $H \in SR$
$\succ$ $SG(H)$ ist DAG
Sei $H' \leq H$ ein Präfix von H
Offensichtlich ist $SG(C(H'))$ ein Teilgraph von $SG(H)$. Da $SG(H)$ ein DAG ist, ist auch $SG(C(H'))$ einer, also $SG(C(H')) \in SR$.

□



3.5 Weitere Operationen

Wir haben bis jetzt die Annahme gemacht, dass wir nur Lese- und Schreiboperationen betrachten. Diese Annahme wird auch weiterhin gelten. In diesem Abschnitt werden wir kurz erläutern, warum dieses keine wesentliche Einschränkung ist und alles, was wir über Lese- und Schreiboperationen haben, leicht übertragen werden kann auf eine beliebige Menge von Operationen. Die wesentliche Definition, auf der alles aufbaut(e), war die der Unverträglichkeit ($\not\parallel$). In dieser nahmen wir explizit Bezug auf die Operationen Lesen und Schreiben. Das muss aber nicht sein. Im Allgemeinen ist es möglich, eine Tabelle spezifizieren zu lassen, aus der hervorgeht, welche Operationen kommutieren und welche unverträglich sind. Alle nachfolgenden Definitionen können dann so verallgemeinert werden, dass sie nicht mehr direkt auf Lese- und Schreiboperationen Bezug nehmen, sondern nur noch noch auf Operationen, die im Konflikt stehen, also unverträglich sind. Dies gilt im Wesentlichen für die Definitionen im Zusammenhang mit Wiederanlauf. Die Definition der Serialisierbarkeit kann unverändert bleiben. Das Serialisierbarkeitstheorem gilt nach wie vor.

Wir werden das Hinzufügen von Operationen an einem kleinen Beispiel diskutieren.

Wir fügen die Operationen Inkrement (inc) und Dekrement (dec) hinzu. Wir nehmen weiter an, dass keiner dieser Operationen einen Wert zurückliefert. Damit können wir folgende *Kompatibilitätsmatrix* KM aufstellen:

	r	w	inc	dec
r	y	n	n	n
w	n	n	n	n
inc	n	n	y	y
dec	n	n	y	y

Sie enthält ein y an einer Stelle, falls die beiden Operationen kommutieren ($\|$) und ein n, falls die Operationen nicht kommutieren ($\not\|$), also im Konflikt miteinander stehen, d.h. unverträglich sind. Allgemein:

$$KM(p_1, p_2) = n \prec\succ p_1 \not\| p_2$$

Der Sinn liegt im Erreichen von mehr Parallelität, das heißt weniger Konflikten.

Bei der Definition muss man aber sehr aufpassen. Betrachte:

- Zähler, der von Null nichts weiter abzieht (Platzreservierung)
- Strafgebühren bei Kontoüberzug (debit, credit)

Wir betrachten beispielhaft die folgende Historie:

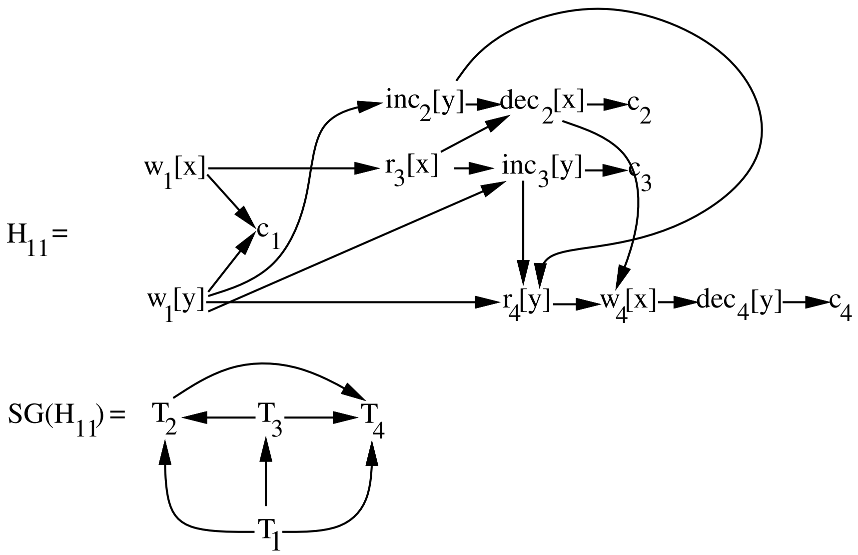
Beispiel 3.5.1

$$H_{11} \quad = \quad \begin{array}{ccccccccccccc}
 & & & & & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & & & \\
 & & & & \nearrow & & & \rightarrow & & & \searrow & & \\
 & & & & & & \nearrow & & \searrow & & & & \\
 & & & & inc_2[y] & \rightarrow & dec_2[x] & \rightarrow & c_2 & \searrow & & \downarrow & \\
 & & & & & \nearrow & & & & & \downarrow & \downarrow & \\
w_1[x] & \rightarrow & \rightarrow & \rightarrow & r_3[x] & \rightarrow & inc_3[y] & \rightarrow & c_3 & & \downarrow & \downarrow & \\
 & \searrow & & & & \nearrow & \downarrow & & & & \downarrow & \downarrow & \\
 & & c_1 & & & & \downarrow & & & & \downarrow & \downarrow & \\
 & & \uparrow & & \nearrow & & \downarrow & & & & \downarrow & \downarrow & \\
 & & \rightarrow & \nearrow & & & \downarrow & & & & \downarrow & (r_4[y]) & \\
 & \nearrow & & & & & \downarrow & & & \swarrow & & & \\
w_1[y] & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & r_4[y] & \rightarrow & w_4[x] & \rightarrow & dec_4[y] & \rightarrow & c_4
\end{array}$$

mit

$$SG(H_{11}) \quad = \quad \begin{array}{ccccc}
 & & \rightarrow & & \\
 & \nearrow & & \searrow & \\
T_2 & \leftarrow & T_3 & \rightarrow & T_4 \\
\uparrow & & \uparrow & & \uparrow \\
 & \nwarrow & \uparrow & \nearrow & \\
 & & T_1 & &
\end{array}$$

$SG(H_{11})$ ist azyklisch, also ist H_{11} serialisierbar. Es gilt:



$H_{11} \equiv T_1, T_3, T_2, T_4$

Da der wesentliche Begriff, auf dem unserer Serialisierbarkeitstheorie bisher aufbaut, der Konflikt ist, spricht man auch von Konfliktserialisierbarkeit (CSR). Dies ist insbesondere im nächsten Abschnitt von Bedeutung, in dem wir Sichtenserialisierbarkeit untersuchen.

3.6 Sichtenserialisierbarkeit

Die Definition der Konfliktserialisierbarkeit benutzte einen Äquivalenzbegriff, der forderte, alle in Konflikt stehenden Operationen in beiden Historien gleich zu ordnen. Dies ist, wenn man will, eine etwas indirekte Definition, da sie sich nicht auf den Effekt bezieht, den man haben will, sondern auf eine Eigenschaft, die diesen Effekt impliziert.

Ein von einer Transaktion R_i geschriebener Wert ist eine Funktion von der Menge aller gelesenen Variablen auf diesen Wert. Mit anderen Worten, wenn eine Transaktion die gleichen Werte liest, so produziert sie auch die gleichen Ergebnisse. Also:

1. Falls eine Transaktion innerhalb zweier Historien alle Werte von denselben Schreiboperationen liest, dann schreiben alle Schreiboperationen in beiden Historien den gleichen Wert.

2. Falls in zwei Historien die letzten Schreiboperationen übereinstimmen, so ist der resultierende Datenbankzustand beider Historien der gleiche.

Wenn sich aber alle Transaktionen in zwei Historien gleich verhalten und die resultierenden Datenbankzustände gleich sind, so sind die Historien äquivalent. Wir formalisieren nun diesen Äquivalenzbegriff:

Definition 3.6.1 (Letztes Schreiben)
Sei H eine Historie und x ein Datenelement. Das letzte Schreiben *des Datenelementes x in der Historie H ist die Operation $w_i[x] \in H$ mit*

1. $a_i \notin H$

2. $\forall w_j[x] \in H \; i \neq j \succ w_j[x] <_H w_i[x] \vee a_j \in H$

Wir schreiben dann auch $w_i[x] = FIN_H(x)$.

Definition 3.6.2 (Sichtenäquivalenz)
Zwei Historien H *und* H' *sind* sichtenäquivalent :≺≻

1. *beide sind über der gleichen Transaktionsmenge* T *definiert,*
2. *beide umfassen die gleiche Menge von Operationen,*
3. $\forall T_i, T_j \in T \forall x \; a_i, a_j \notin H, T_i \rhd_H [x] T_j \prec\succ T_i \rhd_{H'} [x] T_j$
4. $\forall x FIN_H(x) = FIN_{H'}(x)$

Bem.: $a_i, a_j \notin H$ (in 1.) impliziert $a_i, a_j \notin H'$.

Wir können nun den Begriff der Sichtenserialisierbarkeit einführen:

Definition 3.6.3 (Sichtenserialisierbarkeit)
Eine Historie H *heißt* sichtenserialisierbar *(VSR)* :≺≻

$$\forall H' \leq H \; \exists H_s \text{ seriell } \; C(H') \equiv_V H_s$$

Die Forderung, dass jedes Präfix einer Historie zu einer seriellen Historie äquivalent sein muss, garantiert uns, dass Sichtenserialisierbarkeit eine präfix-commit-abgeschlossene Eigenschaft ist.

Beispiel 3.6.1
Die folgende Historie

$$H_{12} = w_1[x] w_2[x] w_2[y] c_2 w_1[y] c_1 w_3[x] w_3[y] c_3$$

hat sich selbst als abgeschlossene Projektion. Es gilt also

$$C(H_{12}) = H_{12}.$$

Desweiteren gilt

$$H_{12} \equiv_V T_1 T_2 T_3$$

Aber das Präfix

$$H'_{12} = w_1[x] w_2[x] w_2[y] c_2 w_1[y] c_1$$

ist weder zu $T_1 T_2$ *noch* $T_2 T_1$ *äquivalent.*

Beispiel 3.6.2 *Es gibt Historien, die sichtenserialisierbar, aber nicht konfliktserialisierbar sind. Betrachte:*

$$H_{13} = w_1[x] w_2[x] w_2[y] c_2 w_1[y] w_3[x] w_3[y] c_3 w_1[z] c_1$$

H_{13} ist sichtenserialisierbar, aber nicht konfliktserialisierbar: Es gilt

$$\begin{aligned}
w_1[x]w_2[x]w_2[y] &\equiv_V \lambda \\
w_1[x]w_2[x]w_2[y]c_2 &\equiv_V T_2 \\
w_1[x]w_2[x]w_2[y]c_2w_1[y]w_3[x]w_3[y]c_3 &\equiv_V T_2T_3 \\
w_1[x]w_2[x]w_2[y]c_2w_1[y]w_3[x]w_3[y]c_3w_1[z]c_1 &\equiv_V T_1T_2T_3
\end{aligned}$$

und

$$SG(H_{13}) \quad = \quad \begin{array}{ccccc}
 & \swarrow & \leftarrow & \nwarrow & \\
T_1 & & & & T_2 \\
\downarrow & \searrow & \rightarrow & \nearrow & \downarrow \\
\searrow & & & & \swarrow \\
 & & T_3 & &
\end{array}$$

Satz 3.6.4 $CSR \subset VSR$

Beweis:

$$\begin{aligned}
H \in CSR &\succ \forall H' \leq H \exists H_s seriell\ H' \equiv H_s \\
& \text{ zu zeigen:} C(H') \equiv_V H_s. \\
& \text{ 1 und 2 } \surd \\
T_i \rhd_{C(H')} [x] T_j &\succ w_j[x] <_{C(H')} r_i[x] \text{ und} \\
& \not\exists w_k[x]\ w_j[x] <_{C(H')} w_k[x] <_{C(H')} r_i[x] \\
&\succ w_j[x] <_{H_s} r_i[x] \text{ und} \\
& \not\exists w_k[x]\ w_j[x] <_{H_s} w_k[x] <_{H_s} r_i[x] \\
&\succ T_i \rhd_{H_s} [x] T_j
\end{aligned}$$

Da für alle i, j $w_i[x] \not\parallel w_j[x]$ gilt und beide Historien $C(H')$ und H_s unverträgliche Operationen in der gleichen Reihenfolge ordnen, sind ihre letzten Schreiboperationen identisch.

Die Echtheit der Teilmengenbeziehung folgt aus Historie H_{13}.

□

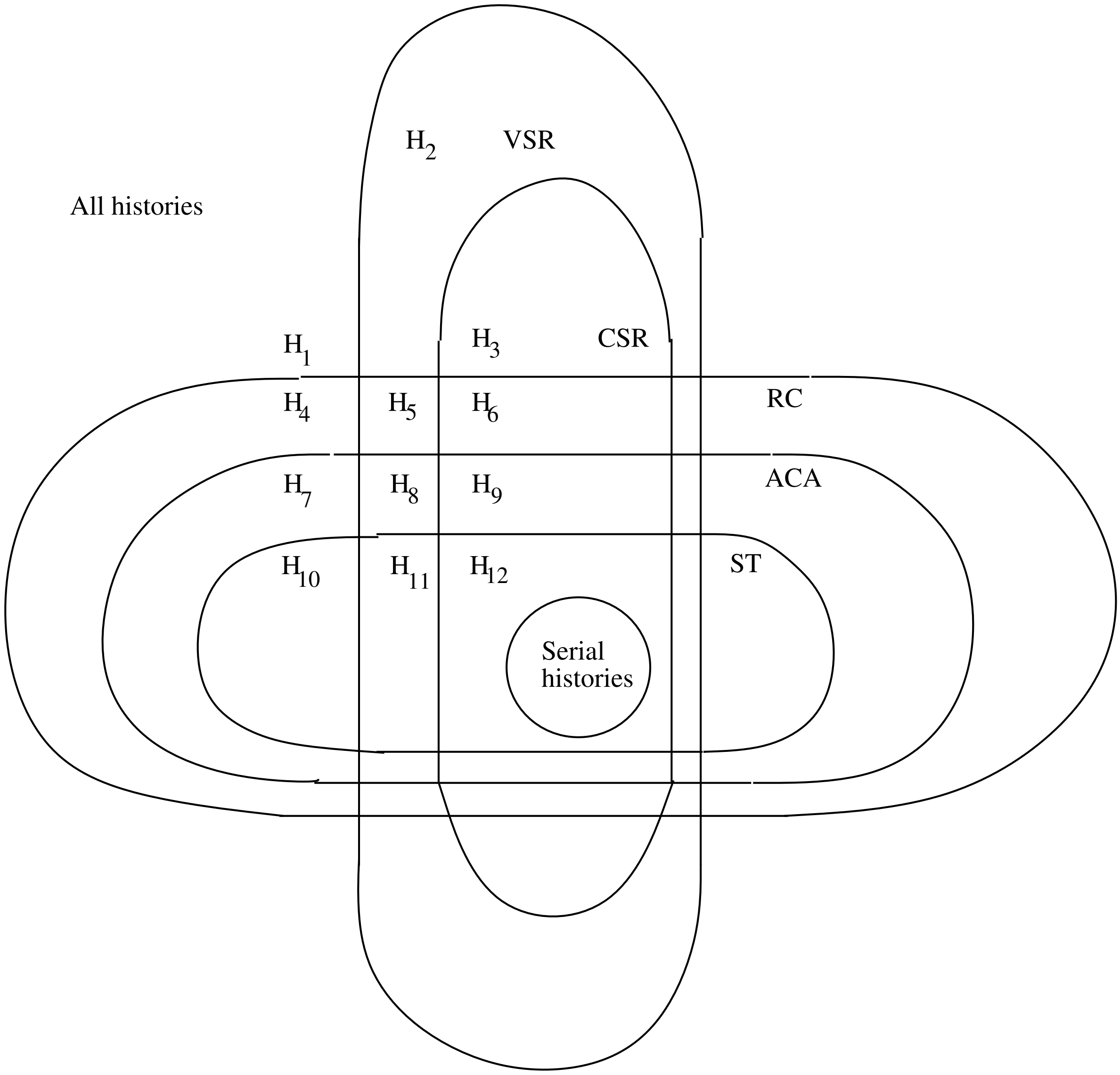
Auch das folgende Beispiel zeigt, dass die Ordnung von Transaktionen innerhalb der zu Präfixen einer Historie äquivalenten seriellen Historien nicht immer gleich ist:

$$w_1[x]w_2[x]w_3[x]w_4[x]w_2[y]r_3[y]w_3[z]r_1[z]c_1c_2c_3c_4$$

Es gilt:

$$\begin{aligned}
w_1[x]w_2[x]w_4[x]w_2[y]r_3[y]w_3[z]r_1[z]c_1c_2 &\equiv_V T_1T_2 \\
w_1[x]w_2[x]w_4[x]w_2[y]r_3[y]w_3[z]r_1[z]c_1c_2c_3c_4 &\equiv_V T_2T_3T_1T_4
\end{aligned}$$

Durch Einschieben weiterer Transaktionen kann dann eine Historie konstruiert werden, die bei jedem weiteren zum Präfix hinzugefügten Commit die Ordnung von T_1 und T_2 umkehrt.



Bei der Konfliktserialisierbarkeit besagte ein Satz, dass eine Historie genau dann konfliktserialisierbar ist, wenn der Serialisierbarkeitsgraph azyklisch ist. Ein analoges Ergebnis streben wir nun für die Sichtenserialisierbarkeit an. Der wesentliche Unterschied für die Konstruktion eines Graphen ergibt sich aus der Bedingung über die liest-von-Beziehung zweier äquivalenter Historien. Sie enthielt den wesentlichen Passus, dass keine Transaktion zwischen dem Schreiben und dem Lesen schreiben durfte. Umgekehrt ausgedrückt heißt dies, dass alle anderen Schreiboperationen entweder *vor* dem Lesen oder *nach* dem Schreiben auftauchen müssen. Um diesen Sachverhalt erfassen zu können, benötigen wir einen verallgemeinerten Graphenbegriff:

Definition 3.6.5 (Polygraph)
Ein Polygraph ist ein Tripel $P = (N, E, B)$ mit

1. *(N, E) ist ein gerichteter Graph und*

2. B ist eine Menge von Pfaden der Länge zwei (Bipfad), also von der Form $((v,u),(u,w))$.

Ein Polygraph kann auch als Familie $\mathcal{D}(N,E,B)$ von gerichteten Graphen angesehen werden, wobei

1. $(N,E') \in \mathcal{D}(N,E,B) \prec\succ E \subseteq E'$ und
2. $\forall (e_1,e_2) \in B\ e_1 \in E' \vee e_2 \in E'$.

Definition 3.6.6 (azyklischer Polygraph)
Ein Polygraph $P = (N,E,B)$ heißt azyklisch, falls mindestens ein azyklischer gerichter Graph in $\mathcal{D}(N,E,B)$ existiert.

Definition 3.6.7 (Vervollständigung einer Historie)
Gegeben sei eine Historie H über der Transaktionsmenge $T = \{T_1,\ldots,T_n\}$. Die vervollständigte Historie $\tilde{H}$ ist definiert als Historie über $T \cup \{T_{n+1}\}$. Dabei enthält T_{n+1} für alle in H auftauchenden Datenelemente x die Operationen $r_{n+1}[x]$. Ferner gilt für $<_{\tilde{H}}$:

$$\forall p \in H, x \in H : p <_{\tilde{H}} r_{n+1}[x]$$

Wir haben also eine letzte Transaktion hinzugefügt, die alle Datenelemente liest. Die Bedingung des letzten Schreibens ist damit durch die liest-von-Beziehung abgedeckt.

Wir definieren nun zu jeder Historie ihren Polygraphen $P(H)$.

Definition 3.6.8 (P(H))
Gegeben sei eine Historie H über der Transaktionsmenge $T = \{T_1,\ldots,T_n\}$. Wir definieren $P(H) = (N,E,B)$ mit

- $N = T$
- $E = \{(T_j,T_i) | T_i \rhd_{\tilde{H}} [x] T_j\}$
- $B = \{((T_i,T_k),(T_k,T_j)) | T_i \rhd_{\tilde{H}} [x] T_j, w_k[x] \in H\}$

Eine Kante (T_j,T_i) drückt aus, dass T_j vor T_i stattfinden muss. Ein Bipfad $((T_i,T_k),(T_k,T_j))$ drückt aus, dass T_k entweder vor T_j oder nach T_i stattfinden muss.

Lemma 3.6.9
Seien H und H' zwei Historien. Dann gilt:

$$H \equiv_V H' \ \prec\succ\ P(H) = P(H').$$

Der Beweis folgt direkt aus der Definition der Sichtenäquivalenz und der Definition von $P(H)$.

Jetzt folgt das Analogon zum Serialisierbarkeitstheorem für Konfliktserialisierbarkeit.

Lemma 3.6.10
Eine Historie H ist genau dann sichtenserialisierbar, wenn $P(H)$ azyklisch ist.

Beweis:
$\succ$

Sei H sichtenserialisierbar.

$\succ$ $\exists H_s$ seriell $H \equiv_V H_s$

$\succ$ $P(H) = P(H_s)$

Noch zu zeigen: $P(H_s) = (N, E, B)$ azyklisch.
Sei $H_s = T_1, \ldots, T_n$. Definiere $(N, E') \in \mathcal{D}(P(H_s))$ als:

$$\begin{array}{rcl} E' & = & E \\ & & \cup\{(T_i, T_k) | i < k, ((T_i, T_k), (T_k, T_j)) \in B\} \\ & & \cup\{(T_k, T_j) | k < j, ((T_i, T_k), (T_k, T_j)) \in B\} \end{array}$$

Es gilt $i < k \ \vee \ k < j$, da $T_i \rhd_{H_s} T_j$, daher kann nicht $j < k < i$ gelten. Es gilt also $(N, E') \in \mathcal{D}(P(H_s))$. (N, E') ist azyklisch, da er ein Teilgraph von $H_s = T_1, \ldots, T_n, T_{n+1}$ ist.

$\prec$

Sei $(N, E') \in \mathcal{D}(N, E, B)$ ein azyklischer gerichteter Graph. Sei H_s die topologische Ordnung von (N, E'). Dann ist H_s eine serielle Historie mit $H_s \equiv_V H$.

$\square$

Wir haben gesehen, dass das Entscheidungsproblem, ob eine Historie konfliktserialisierbar ist oder nicht, in polynomialer Zeit berechnet werden kann. Wir werden nun zeigen, dass das entsprechende Entscheidungsproblem für Sichtenserialisierbarkeit NP-vollständig ist. Dazu benötigen wir zunächst folgende Hilfslemmata: (Siehe auch C. H. Papadimitriou: The Serializability of Concurrent Database Updates, JACM 26(4), 631ff, Oct. 79, und Garey, Johnson).

Lemma 3.6.11 *Seien H und H' zwei Historien, die auf Datenelemente D zugreifen. Beide haben n Transaktionen und m Operationen. Es kann in polynomialer Zeit ($O(m \log(m) + m * |D| + n^2)$) entschieden werden, ob zwei Historien sichtenäquivalent sind.*

Beweis:

ad 1: ob beide Historien die gleichen Transaktionen umfassen: mit 2.

ad 2: ob beide Historien die gleichen Operationen umfassen: $O(m \log(m))$. Gleichzeitig merkt man sich, welche Transaktionen abschließen und welche abbrechen.

ad 3: wer liest von wem (durch Notieren der last write pro Datenelement): $O(n * |D|)$, ($|D|$ kommt durch die Verzweigung in Historien. Pro Pfad ein Array mit allen Datenelementen, ein Array mit $T \times T$ Elementen vom Typ Bool. Immer wenn man auf ein $r_i[x]$ stößt, schaut man nach, wer dieses als letztes geschrieben hat.) Anschließend kann man in $O(n^2)$ nachsehen, ob Bedingung 3 erfüllt ist.

ad 4: mit 3. Nachsehen, ob die letzten $w_i[x]$ übereinstimmen: $O(|D|)$.

□

Lemma 3.6.12 *3-SAT ist NP-vollständig*

Definition 3.6.13 *Eine Klausel heißt* gemischt, *falls sie sowohl positive als auch negative Literale enthält. Eine Klauselmenge heißt* nicht-zirkulär, *falls jede Variable höchstens einmal in einer gemischten Klausel vorkommt.*

Bsp.: $\{\{abc\}\{a\overline{b}\}\}$ und $\{\{ab\overline{c}\}\{a\overline{b}\}\}$

Lemma 3.6.14 *Sei* $3 - SAT^{nz}$ *das Entscheidungsproblem für nicht-zirkuläre Klauselmengen.* $3 - SAT^{nz}$ *ist NP-vollständig.*

Beweis:
Für jede Variable v in einer Klauselmenge K führe folgendes durch: Sei m die Anzahl der Vorkommen von v in K. Führe $v_1, \ldots, v_m$ neue Variablen ein. Ersetze das erste Vorkommen von v in K durch v_1, das zweite durch $\overline{v_2}$, das dritte durch v_3 etc. Dann füge

$$\{\{v_1, v_2\}, \{\overline{v_1}, \overline{v_2}\}, \{v_2, v_3\}, \{\overline{v_2}, \overline{v_3}\} \ldots\}$$

zu K hinzu. Diese entspricht der Klauseldarstellung von

$$v_1 \equiv \overline{v_2} \equiv v_3 \equiv \overline{v_4} \ldots$$

Diese Modifikation von K, durchgeführt für alle Variablen in K, resultiere in K'. Dann gilt:

1. $K \equiv K'$,
2. K' ist nicht-zirkulär und
3. K' kann aus K in polynomialer Zeit gewonnen werden.

□

Satz 3.6.15 *Das Entscheidungsproblem, ob zu einer gegebene Historie H eine serielle Historie H_s existiert mit $H \equiv_V H_s$, ist NP-vollständig.*

Beweis:
Falls eine serielle Historie gegeben ist, so können wir die Sichtenäquivalenz in polynomialer Zeit entscheiden. Noch zu zeigen: $3 - SAT^{nz}$ kann in polynomialer Zeit auf obiges Entscheidungsproblem reduziert werden.

Sei

$$K = \{C_1, \ldots, C_m\}$$

eine Klauselmenge in Variablen $v_1, \ldots, v_n$, mit o.B.d.A.

$C_i = \{l_1^i, \ldots, l_{m_i}^i\}$,

wobei $m_i \leq 3$.

Wir konstruieren nun einen Polygraphen $P_K = (N, E, B)$. Zunächst das Gerüst:

$$\begin{aligned} N &= \{x_j, y_j, z_j | 1 \leq j \leq n\} \text{ pro } v_j \text{ (für jede Variable)} \\ &\quad \cup \{c_k^i, d_k^i | 1 \leq i \leq m, 1 \leq k \leq m_i\} \text{ pro } C_i \text{ (für jede Klausel)} \end{aligned}$$

$$\begin{aligned} E &= \{(x_j, y_j) | 1 \leq j \leq n\} \text{ pro } v_j \\ &\quad \cup \{(c_k^i, d_{k+1(mod\ m_i)}^i) | 1 \leq i \leq m, 1 \leq k \leq m_i\} \text{ pro } C_i \\ B &= \{((y_j, z_j), (z_j, x_j)) | 1 \leq j \leq n\} \text{ pro } v_j \end{aligned}$$

Jetzt berücksichtigen wir das Vorzeichen der Literale:
Für jedes Literal $l_k^i = v_j$:

$$\begin{aligned} E &\cup= \{(z_j, c_k^i), (y_j, d_k^i)\} \\ B &\cup= \{((d_k^i, c_k^i), (c_k^i, y_j))\} \end{aligned}$$

Für jedes Literal $l_k^i = \overline{v_j}$:

$$\begin{aligned} E &\cup= \{(d_k^i, z_j), (c_k^i, x_j)\} \\ B &\cup= \{((x_j, d_k^i), (d_k^i, c_k^i))\} \end{aligned}$$

Zum Schluss fügen wir noch folgende Knoten hinzu, die der Konstuktion von $\tilde{H}$ aus H entsprechen:

$$\begin{aligned} N &\cup= \{n_f\} \\ E &\cup= \{(n, n_f) | n \in N, n \neq n_f\} \end{aligned}$$

Man beachte, dass die letzte Konstruktion nichts an der Azyklizität bzw. Zyklizität von P_K ändert. Offensichtlich ist diese Konstruktion in polynomialer Zeit durchführbar.

Zwischenlemma:

P_K azyklisch $\prec\succ$ K erfüllbar.

"$\succ$"

P_K azyklisch

$\succ$ $\exists (N, E') \in \mathcal{D}(N, E, B)$ mit (N, E') azyklisch

$\succ$ $\forall 1 \leq j \leq n$: entweder $(z_j, x_j) \in E'$ oder $(y_j, z_j) \in E'$
Eines ist mindestens drin, nach Definition. Das zweite wird ausgeschlossen, da (N, E') azyklisch.

Es repräsentiere (z_j, x_j) die Tatsache, dass der Variablen v_j der Wahrheitswert T zugeordnet wird.
Falls $l_k^i = v_j$ zu F evaluiert wird, gilt

$$(d_k^i, c_k^i) \in E',$$

da sonst der Zyklus z_j, c_k^i, y_j, z_j existieren würde.
Falls $l_k^i = \overline{v_j}$ zu F evaluiert wird, gilt

$$(d_k^i, c_k^i) \in E',$$

da sonst der Zyklus d_k^i, z_j, x_j, d_k^i existieren würde.

Die einzige Möglichkeit, dass (N, E') keinen Zyklus der Form

$$d_1^i, c_1^i, d_2^i, \ldots, c_{m_i}^i, d_1^i$$

enthält, ist also diejenige, dass mindestens ein Literal zu T evaluiert. Daraus folgt, dass K erfüllbar ist.

$\prec$

K sei erfüllbar durch eine Interpretation I. Wir werden daraus eine azyklischen gerichteten Graphen $(N, E') \in \mathcal{D}(P_K)$ gewinnen.

$$\begin{aligned} E' &= E \\ &\cup\{(z_j, x_j) | I(v_j)\} \\ &\cup\{(y_j, z_j) | \neg I(v_j)\} \\ &\cup\{(d_k^i, c_k^i) | \neg I(l_k^i)\} \\ &\cup\{(c_k^i, y_j) | l_k^i = \overline{v_j}, I(v_j)\} \\ &\cup\{(x_j, d_k^i) | l_k^i = v_j, \neg I(v_j)\} \end{aligned}$$

Dann gilt: $(N, E') \in \mathcal{D}(P_K)$.
Da K azyklisch ist, ist (N, E) azyklisch, da durch die Konstruktion von E Folgendes impliziert wird:

- Klauseln, die nur Variablen enthalten (unnegiert), korrespondieren zu Knotenmengen mit nur einlaufenden Kanten.
- Klauseln, die nur negierte Variablen enthalten, korrespondieren zu Knotenmengen mit nur auslaufenden Kanten.
- Knotenmengen, die zu gemischten Klauseln korrespondieren, haben beides, einlaufende und auslaufende Kanten, aber keine zwei solchen Knotenmengen sind untereinander erreichbar in (N, E), da K azyklisch ist.

$E' \setminus E$ kann nur einen Zyklus in den gerichteten Graphen (N,E) einführen, der die Form $d_1^i, c_1^i, d_2^i, \ldots$ hat. Das aber würde bedeuten, dass die korrespondierende Klausel C_i kein Literal enthält, das zu wahr evaluiert. $\square$

Damit haben wir das Zwischenlemma bewiesen. Um den Beweis zu vervollständigen, benötigen wir nur noch eine Historie H mit $P(H) = P_K$. Damit können wir dann jede Klauselmenge K in $3 - SAT^{nz}$ in polynomialer Zeit auf eine Historie reduzieren, die genau dann sichtenserialisierbar ist, wenn K erfüllbar ist.

Dazu wird jeder Knoten in P_K eine Transaktion. Sei $|N| = n + 1$. Dann bezeichnen wir auch n_i als T_i.

Die Menge der Operationen wird wie folgt festgelegt:

- $\forall\ (T_i, T_j) \in E$:
  - $w_i[x_{T_i,T_j}] \in T_i$
  - $r_j[x_{T_i,T_j}] \in T_j$
- $\forall\ ((T_i, T_k), (T_k, T_j)) \in B$:
  - $w_k[x_{T_j,T_i}] \in T_k$

Der letzte Schritt legt $<_H$ fest. Wir werden eine totale Ordnung konstruieren.

Bemerkung:

> Man erinnere sich, dass von rein negativen Klauseln nur Kanten ausgingen, und dass rein positive Klauseln nur eingehende Kanten besitzen. Weiter konnte keine Variable zweimal in einer gemischten Klausel vorkommen. Es liegt daher nahe, die den Klauseln zugeordneten Teilgraphen dementsprechend zu ordnen: zuerst die rein negativen, dann die gemischten und dann die positiven. Dies entspricht auch der Vorgehensweise, wobei allerdings jeweils Teile vorgezogen bzw. zurückgestellt werden.

Seien

$$R(T_i) = r_i[x_1], \ldots, r_i[x_{l_i}]$$

die Leseoperationen von T_i, und

$$W(T_i) = w_i[y_1], \ldots, w[y_{k_i}]$$

die Schreiboperationen von T_i, dann bezeichnen wir mit $H(T_i)$ die partielle Historie

$$H(T_i) = r_i[x_1], \ldots, r_i[x_{l_i}], w_i[y_1], \ldots, w[y_{m_i}]c_i$$

Man beachte, dass diese der Definition einer Transaktion genügt und die Ordnung total ist.

Wir konstruieren die Historie von vorne nach hinten, es werden also immer Teile angehängt. Definiere für jede Klausel $C_i = \{l_1^i, \ldots, l_{m_i}^i\}$, die nur negative Literale enthält, die partielle Historie $H(C_i)$:

$$H(C_i) = H(c_1^i), \ldots, H(c_{m_i}^i)$$

Man beachte, dass dies auch eine Historie ist, und zwar eine total geordnete.

Weiter fügen wir für jede Variable v_j, die in einer gemischte Klausel C_l vorkommt, der Historie folgendes hinzu:

Falls v_j positiv als $l_k^l = v_j$ vorkommt:

$$\begin{array}{lcllllllll} H(v_j) & = & R(x_j) & & z_j & W(x_j) & R(y_j) & & & W(y_j) \quad \text{RAHMEN} \\ H(v_j) & = & R(x_j) & d_m^i & z_j & W(x_j) & R(y_j) & c_k^l & c_q^p & W(y_j) \end{array}$$

- Die d_m^i kommen nur vor, falls es eine rein negative Klausel C_i gibt, mit $l_m^i = \overline{x_j}$.
- Die c_q^p kommen nur vor, falls es eine rein positive Klausel C_p gibt, mit $l_q^p = x_j$.

Man erinnere sich, dass eine Variable nur in einer gemischten Klausel vorkommen kann. Es bleiben also nur die beiden obigen Fälle übrig.

Wir fahren auf dieselbe Art und Weise für die anderen Variablen fort.

Falls v_j in allen gemischten Klauseln nur negativ vorkommt, fügen wir folgende Historie hinzu:

$$\begin{array}{lcllllllll} H(v_j) & = & R(x_j) & & & z_j & W(x_j) & R(y_j) & & W(y_j) \quad \text{RAHMEN} \\ H(v_j) & = & R(x_j) & d_m^i & d_q^p & z_j & W(x_j) & R(y_j) & c_k^l & W(y_j) \end{array}$$

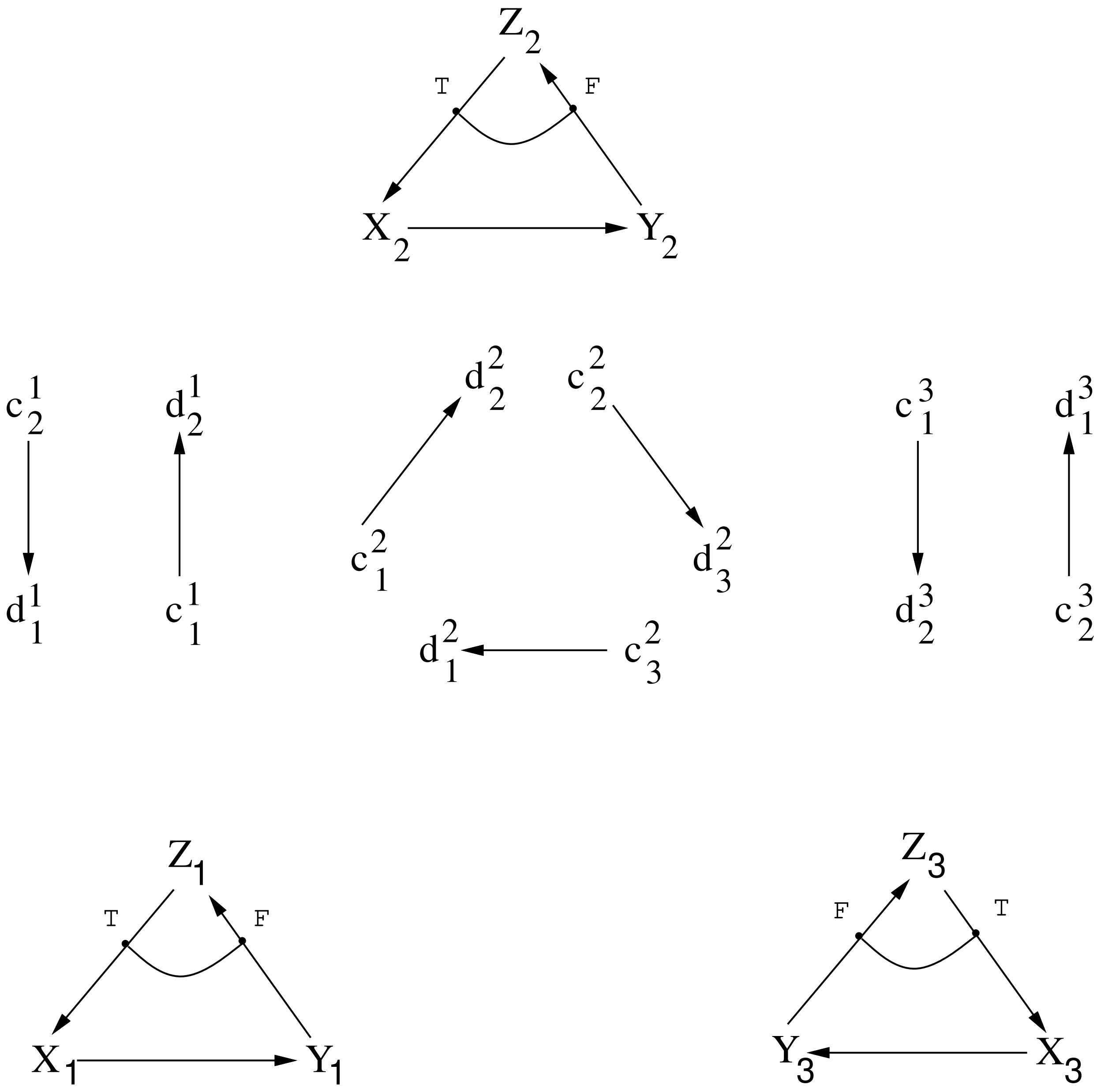
- Die c_k^l kommen nur vor, falls es eine rein positive Klausel C_l gibt, mit $l_k^l = v_j$
- Die d_q^p kommen nur vor, falls es eine rein negative Klausel C_p gibt, mit $l_q^p = \overline{v_j}$

Wir fahren auf dieselbe Art und Weise für die anderen Variablen fort.

Wir fügen jetzt noch die noch nicht abgearbeiteten c_j^i und d_j^i sowie commit-Operationen an das Ende der Historie an.

Daß $P_f(P(H))$ gilt, sieht man wie folgt. Zunächst werden alle $(c_j^i, d_{j+1(mod m_i)}^i)$ Kanten durch H eingefügt. Der Teilpolygraph, bestehend aus den Knoten $\{x_j, y_j, z_j, c_k^i, d_k^i\}$, ergibt sich für jedes $v_j = l_k^i$, der analoge Teilpolygraph aus $\overline{v_j} = l_k^i$. Des Weiteren kann man leicht überprüfen, dass keine weiteren Kanten oder Bipfade hinzugefügt werden.

□



$$K = \{ c_1 , c_2 , c_3 \}$$

$c_1 = \{ v_1, v_2 \}$ $\quad$ $c_2 = \{ v_1, \overline{v_2}, v_3 \}$ $\quad$ $c_3 = \{ \overline{v_2}, \overline{v_3} \}$

Kapitel 4

Sperren-basierte Synchronisationsverfahren

Ein Scheduler hat für jede Operation, die ihn erreicht, drei Möglichkeiten:

1. sofortiges Ausführen (Weitergabe an den Data Manager)
2. Verzögern (Einreihen in eine Warteschlange)
3. Zurückweisen (Zurücksetzen der zugehörigen Transaktion)

Scheduler unterscheiden sich dann zunächst einmal dadurch, dass sie eine oder zwei von diesen Möglichkeiten bevorzugen.

Wir können also zwei Arten von Schedulern unterscheiden:

aggressiv bevorzugt sofortiges Ausführen. Dies führt dazu, dass weniger umsortiert werden kann.

konservativ bevorzugt Verzögerung. Bewahrt sich also die Möglichkeit umzusortieren, verzögert aber den Fortgang von Transaktionen.

Extrem eines konservativen Schedulers: Alle Transaktionen müssen vor Beginn ihre *read* und *write* sets bekannt geben (preclaiming). Da dies sehr schwierig ist (Schleifen, Verzweigungen), kann oft nur eine Obermenge angegeben werden. Deklarative Anfragesprachen machen die Sache noch schwieriger. Es muss aber nie zurückgesetzt werden.

4.1 Das einfache Zwei-Phasen-Sperrprotokoll

Gemäß den betrachteten Operationen (read und write), führen wir zwei Arten von Sperren ein:

- $rl[x]$: Lesesperre
- $wl[x]$: Schreibsperre

Falls die Operation entweder Lesen oder Schreiben sein kann, schreiben wir $ol[x]$ ($pl[x]$, $ql[x]$). Transaktionszugehörigkeit wird wieder durch Index ausgedrückt. Wir bezeichnen mit $ol_i[x]$ nicht nur die Sperren selbst, sondern gegebenenfalls auch die Operation, die diese Sperre anfordert. Die Freigabe wird mit $ou_i[x]$ bezeichnet.

Definition 4.1.1 *Zwei Sperren $pl_i[x]$ und $ql_j[y]$ stehen in Konflikt miteinander ($pl_i[x] \not\parallel ql_j[y]$) $:\prec\succ$*

1. $x = y$
2. $i \neq j$
3. $p \not\parallel q$

Alle Sperren werden in einer zu Anfang leeren Sperrtabelle gehalten. Der *Basis-2PL-Scheduler* arbeitet nach folgenden drei Regeln:

1. Bei Eintreffen von $p_i[x]$: Falls $p_i[x]$ nicht mit einer bereits vergebenen Sperre in Konflikt steht, wird $pl_i[x]$ an T_i vergeben und $p_i[x]$ ausgeführt. Sonst wird $p_i[x]$ verzögert, bis die entsprechende Sperre freigegeben ist.
2. Eine Sperre $pl_i[x]$ wird nicht eher freigegeben, bis nicht wenigstens $p_i[x]$ ausgeführt wurde. Sperren werden spätestens am Transaktionsende wieder freigegeben.
3. Nachdem für eine Transaktion mindestens eine Sperre freigegeben wurde, dürfen keine Sperren mehr an diese Transaktion vergeben werden.

Die letzte Regel ist die 2-Phasen-Regel. Sie teilt den Verlauf der Transaktion in eine sperrenanfordernde und eine sperrenfreigebende Phase. Dies kann durch folgendes Beispiel motiviert werden:

Beispiel 4.1.1
Betrachte die Transaktionen

$$\begin{aligned} T_1 &: r_1[x]w_1[y]c_1 \\ T_2 &: w_2[x]w_2[y]c_2 \end{aligned}$$

zusammen mit der Historie

$$H_1 = rl_1[x]r_1[x]ru_1[x]wl_2[x]w_2[x]wl_2[y]w_2[y]wu_2[x]wu_2[y]c_2wl_1[y]w_1[y]wu_1[y]c_1$$

Wegen $r_1[x] <_{H_1} w_2[x]$ und $w_2[y] <_{H_1} w_1[y]$ ist $SG(H_1)$ zyklisch, also H_1 nicht serialisierbar. Trotzdem traten keine Sperrkonflikte auf. Grund: Freigabe durch $ru_1[x]$ vor $wl_1[y]$.

Unglücklicherweise können durch dieses Sperrprotokoll Deadlocks entstehen:

Beispiel 4.1.2 *Betrachte die Transaktionen*

$$\begin{aligned} T_1 &: r_1[x]w_1[y]c_1 \\ T_3 &: w_3[y]w_3[x]c_3 \end{aligned}$$

und die folgende Historie:

$$H_2 = rl_1[x]r_1[x]wl_3[y]w_3[y]$$

Diese kann durch kein 2-PL-Protokoll fortgesetzt werden.

Deadlocks treten auch bei Sperrverschärfung auf:

Beispiel 4.1.3 *Betrachte die Transaktionen*

$$\begin{array}{rcl} T_4 & : & r_4[x]w_4[x]c_1 \\ T_5 & : & r_5[x]w_5[x]c_2 \end{array}$$

und folgende Historie

$$H_3 = rl_4[x]r_4[x]rl_5[x]r_5[x]$$

Wieder kann die Historie nicht ohne Protokollverletzung fortgeführt werden.

Deadlocks werden durch Zyklen im Wartegraphen erkannt. Aufhebung kann nur durch Abbruch einer am Zyklus beteiligten Transaktion erfolgen. Zur Auswahl können folgende Kriterien dienen:

1. Bereits durch eine Transaktion entstandene Kosten.
2. Kosten, die entstehen, wenn eine Transaktion zu Ende geführt wird.
3. Kosten, die entstehen, wenn eine Transaktion abgebrochen wird.
4. Anzahl der Zyklen, an denen eine Transaktion beteiligt ist.
5. Anzahl der durch Deadlocks bedingten Abbrüche einer Transaktion.

Erkennungsalgorithmen für Deadlocks werden als bekannt vorausgesetzt.

4.2 Korrektheit des einfachen 2-Phasen-Sperrprotokolls

Wir müssen nun beweisen, dass alle durch das 2-PL-Protokoll generierten Historien serialisierbar sind. Dazu gehen wir in zwei Schritten vor:

1. Nachweis von Eigenschaften, die alle 2-PL-Historien aufweisen.
2. Nachweis, dass aus diesen Eigenschaften die Serialisierbarkeit folgt.

Zunächst erweitern wir die Definition der Historie um Sperranforderungen und Sperrfreigaben. Wir beobachten

Proposition 4.2.1
Sei H eine 2-PL-Historie. Dann gilt:

1. $\forall p_i[x] \in C(H) \quad pl_i[x] \in C(H) \quad \wedge \quad pu_i[x] \in C(H)$
2. $\forall p_i[x] \in C(H) \quad pl_i[x] <_H p_i[x] <_H pu_i[x]$

Diese Eigenschaften lassen sich aus den Regeln 1 und 2 des 2-PL-Protokolls herleiten.

Proposition 4.2.2
Sei H eine 2-PL-Historie. Dann gilt:

- $\forall p_i[x], q_j[x] \in C(H)(i \neq j) \quad p_i[x] \not\parallel q_j[x] \succ pu_i[x] <_H ql_j[x] \vee qu_j[x] <_H pl_i[x]$

Dies folgt direkt aus Regel 1 des 2-PL-Protokolls.

Proposition 4.2.3
Sei H eine 2-PL-Historie. Dann gilt:

- $\forall p_i[x], q_i[y] \in C(H) \quad pl_i[x] <_H qu_i[y]$

Dies folgt direkt aus der Regel 3, die eine Transaktion klar in eine sperrenanfordernde und eine sperrenfreigebende Phase teilt.

Lemma 4.2.4
Sei H eine 2-PL-Historie mit $T_i \rightarrow T_j \in SG(H)$. Dann gilt:

- $\exists p_i[x], q_j[x] \in H \quad p_i[x] \not\parallel q_j[x] \quad \wedge \quad pu_i[x] <_H ql_j[x]$

Beweis:

$$
\begin{array}{rcl}
T_i \rightarrow T_j \in SG(H) & \succ & \exists p_i[x], q_j[x] \in H \quad p_i[x] \not\parallel q_j[x] \quad \wedge \quad p_i[x] <_H q_j[x] \\
Prop.\ 4.2.1 & \succ & pl_i[x] <_H p_i[x] <_H pu_i[x] \\
 & & \wedge \\
 & & ql_j[x] <_H q_j[x] <_H qu_j[x] \\
Prop.\ 4.2.2 & \succ & pu_i[x] <_H ql_j[x] \\
 & & \vee \\
 & & qu_j[x] <_H pl_i[x]
\end{array}
$$

$qu_j[x] <_H pl_i[x]$ impliziert $q_j[x] <_H p_i[x]$, ein Widerspruch zu $p_i[x] <_H q_j[x]$. Also $pu_i[x] <_H ql_j[x]$.

□

Lemma 4.2.5
Sei H eine 2-PL-Historie und $T_1 \rightarrow \ldots \rightarrow T_n$ ein Pfad in $SG(H)$ mit $n > 1$. Dann gilt:

- $\exists x, y \ p_1[x], q_n[y] \in H \quad pu_1[x] < ql_n[y]$

Beweis:
Durch Induktion nach n:

$n = 2$**:** folgt direkt aus Lemma 4.2.4.

$n > 2$**:** Die Behauptung stimme für $k = n - 1$. Aus der Induktionshypothese:

1. $\exists x, z \ p_1[x], o_k[z] \in H \quad pu_1[x] <_H ol_k[z]$

Aus $T_k \rightarrow T_n$ folgt mit Lemma 4.2.4

2. $\exists y, o'_k[y], q_n[y] \in H \quad o'_k[y] \not\| q_n[y] \wedge o'u_k[y] <_H ql_n[y]$

Proposition 4.2.3 folgt

3. $ol_k[z] <_H o'u_k[y]$

Aus 1,2 und 3 folgt mit Hilfe der Transitivität von '$<_H$':

- $pu_1[x] <_H ql_n[y]$

□

Satz 4.2.6
Jede 2-PL-Historie H ist serialisierbar.

Beweis:
Annahme: $SG(H)$ enthält einen Zyklus der Form $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ mit $n > 1$. Aus Lemma 4.2.5 folgt

- $\exists x, y \; p_1[x], q_1[y] \in H \quad pu_1[x] <_H ql_1[y]$

Widerspruch zu Proposition 4.2.3.

□

Übungsaufgabe: Zeige, dass die Umkehrung des Satzes falsch ist.

Deadlocks können mit dem *konservativen* 2-PL-Protokoll vermieden werden. Bei diesem müssen alle Sperren *vor* der ersten Operation angefordert werden.

Fast alle realen Scheduler implementieren das *strikte* 2-PL-Protokoll. Bei diesem werden die Sperren erst beim Commit freigegeben. Genauer: nachdem die Commit-Operation c_i durch den DM wahrgenommen wurde. Hierfür sprechen zwei Gründe:

1. Es ist schwer vorherzusagen, ob eine Sperre noch benötigt wird.

2. Strikte 2-PL-Protokolle ergeben strikte Historien.

Satz 4.2.7
Jede H, die dem strikten 2-PL-Protokoll gehorcht, ist strikt.

Beweis:
Aus

- $w_i[x] <_H o_j[x]$

folgt mit Proposition 4.2.1

1. $wl_i[x] <_H w_i[x] <_H wu_i[x]$

2. $ol_j[x] <_H o_j[x] <_H ou_j[x]$

Da $wl_i[x] \not\parallel ol_j[x]$, folgt aus Proposition 4.2.2

3. $wu_i[x] <_H ol_j[x]$ oder $ou_j[x] <_H wl_i[x]$

Letzteres ergibt mit 1 und 2 einen Widerspruch zu $w_i[x] <_H o_j[x]$. Daher gilt

4. $wu_i[x] <_H ol_j[x]$

Da H dem strikten 2-PL-Protokoll gehorcht, gilt:

5. entweder $a_i <_H wu_i[x]$ oder $c_i <_H wu_i[x]$.

Daraus folgt mit 2, 3 und 5

- entweder $a_i <_H o_j[x]$ oder $c_i <_H o_j[x]$.

Also ist H strikt.

$\square$

Aus dem Beweis folgt weiter, dass man eigentlich nur Schreibsperren bis zum Commit-Zeitpunkt aufschieben muss. Lesesperren könnten also bei Start der Commitoperation freigegeben werden, während Schreibsperren bis zum Ende der Commitoperation gehalten werden müssen. (analog für abort)

Implementierungshinweise:

- LockManager:
  - schnell: < 100 Op's pro Sperrenanforderung/-freigabe.
  - spezielle Operationen, um alle Sperren einer Transaktion freizugeben, um eine Sperre, die nicht in Konflikt mit einer vergebenen steht, zu vergeben.
  - Üblich: Hashtabelle nach Datenelementen, Verkettung der Sperren einer Transaktion.
- Blockieren: wichtig: fairness
  - FCFS
  - Leseoperationen on-block durchlassen
- Atomizität von Lese- und Schreiboperationen, Sperroperationen: bspw. Semaphore
  - Falls die Granularität eine Seite ist, dann ist es am einfachsten.
    Falls Records, extra Sperren (kurz, kein 2-PL) für Seiten.
  - Sperrtabelle kann Flaschenhals werden: Partitionieren

Da bis jetzt nur Lese- und Schreiboperationen berücksichtigt wurden, kann es zum sogenannten *Phantomproblem* bei nicht statischen Datenbanken kommen. (Also solche, bei denen Einfüge- und Löscheoperationen vorkommen).

Beispiel 4.2.1 *Annahme: Es existieren zwei Files, die je eine der folgenden Relationen gleichen Namens beinhalten:*

Konto *(Nr, Ort, Guthaben)*

Einlagen *(Ort, SummeGuthaben)*

Eine Transaktion T_1 lese alle Guthaben *von Konten in Karlsruhe und summiere diese auf. Das Ergebnis wird dann mit der Summe in* Einlagen *abgeglichen. Transaktion T_2 fügt ein neues Konto mit einem Anfangsguthaben von 50 ein. Folgende Historie ist nicht serialisierbar, aber 2-PL:*

1. $r_1[Konto(1-9), Khe]$ */* summe ergebe X */*
2. $insert_2[Konto(10), Khe, 50]$
3. $r_2[Einlagen(2), Khe]$ */* returns X */*
4. $w_2[Einlagen(2), Khe]$ */* writes X+50 */*
5. $r_1[Einlagen(2), Khe]$ */* reads X+50 */*

Das Problem ist, dass bei der insert*-Operation keine Sperre gesetzt wird.*

Zur Lösung können *Index-Sperren* benutzt werden. Für jeden Schlüssel existiere ein Indexeintrag, der nur Verweise auf Datenelemente mit assoziiertem zugehörigem Schlüssel besitzen. Dann sind diese Einträge natürlich zu sperren, sobald eine Einfüge- oder Löscheoperation stattfindet.

Für unser Beispiel nehmen wir an, dass ein Index auf *Ort* existiert. Wenn man nun alle Orte aus Karlsruhe liest und den Index hierfür benutzt, so erhält der entsprechende Indexeintrag eine Lesesperre. Damit werden dann nachfolgende *insert*-Operationen durch das 2-PL-Protokoll verzögert.

4.3 Sperrgranulate

oder *multiple granularity locking*

Bis jetzt war die Menge alle Datenelemente völlig unstrukturiert. Dies ist aber in einer Datenbank nicht der Fall. Verschiedene Granulate sind in einer Datenbank erkennbar. Zum Beispiel (auf eher physischer Ebene)

- (ganze) Datenbank
- Segment, Area, DB-Space
- Datei
- Seite
- Record

oder in einer objektorientierten Datenbank auf semantischer Ebene:

- Datenbank
- Schema

- Extension
- Objekt
- Attribut

Plakativ:

- Großes Sperrgranulat $\succ$ wenig Sperren $\succ$ geringe Nebenläufigkeit.
- Kleines Sperrgranulat $\succ$ viele Sperren $\succ$ hohe Nebenläufigkeit

Beispiele:

- Gehaltserhöhung aller Angestellten nach Tarifrunde (alle schreiben)
- Berechnen der Gesamtpersonalkosten (alle lesen)
- Auswertung eines Wettbewerbs, bei dem die besten drei Vertreter eine Prämie bekommen. (alle lesen, drei schreiben)

Diese längeren Transaktionen haben eine größere Chance durchzukommen, wenn sie nur eine Sperre auf der Menge aller Angestellten setzen. Es muss allerdings durch diese Sperre verhindert werden, dass andere Transaktionen einzelne Angestellten ändern.

Wir benötigen zusätzliche Sperren, erläutert an den beiden Granularitäten Extension und Objekt:

r : share lock

- für ein Granulat: Sperrt zum Lesen.

w : exclusive lock

- für ein Granulat: Sperrt zum Schreiben (wie vorher).

ir : intention share lock

- für ein Granulat: zeigt an, dass (ein) Objekt(e) dieses Granulats gelesen werden soll(en). Die share-Sperren für die Objekte müssen noch einzeln explizit geordert werden.

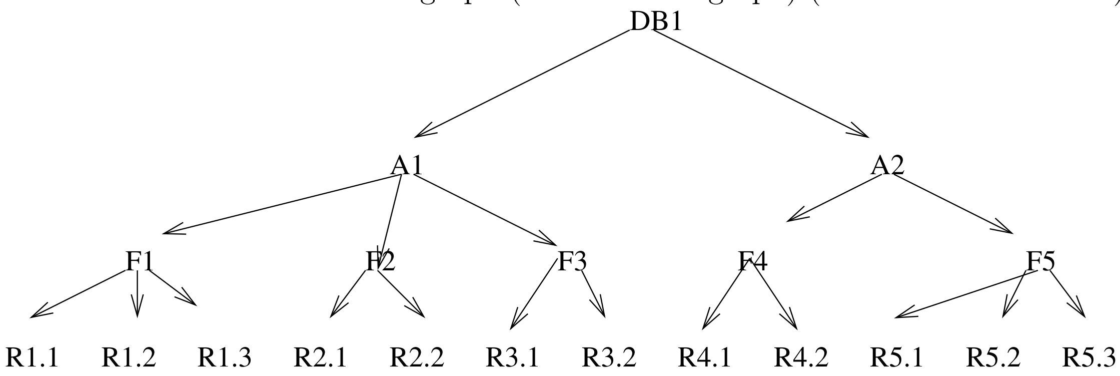
iw : intention exclusive lock

- analog zum intention share lock

riw : share intention exclusive lock

- für ein Granulat: Zeigt an, dass alle Objekte gelesen und einige geschrieben werden sollen. Die X-Sperren zum Schreiben müssen noch explizit angefordert werden.

Dies gilt natürlich für je zwei benachbarte Sperrgranulate. Eine Sperre auf ein Granulat X sperrt *implizit* alle unter X befindlichen Granulate. Dies macht man sich am besten an einem Bild klar. Instanzhierarchiegraph (lock instance graph) (auf Baum beschränkt):



Kompatibilitätsmatrix:

	r	w	ir	iw	riw
r	yes	no	yes	no	no
w	no	no	no	no	no
ir	yes	no	yes	yes	yes
iw	no	no	yes	yes	no
riw	no	no	yes	no	no

Wir führen nun folgendes Sperrprotokoll (MGL) ein:

1. Falls eine *r*- oder *ir*-Sperre für ein Granulat benötigt wird, müssen alle Vorgänger *ir* oder *iw* gesperrt werden.

2. Falls eine *w*- oder *iw*-Sperre für ein Granulat benötigt wird, müssen alle Vorgänger mit *iw* gesperrt werden.

3. Falls T_i ein Datenelement lesen (schreiben) will, so benötigt sie eine r (w) Sperre für das Datenelement oder einen Vorgänger.

4. Eine *ix*-Sperre kann nur freigegeben werden, falls keine Sperre mehr für Nachfolger gehalten wird.

Sperren werden natürlich nur dann vergeben, falls keine andere Transaktion bereits eine in Konflikt stehende Sperre hält.
Beispiele: Sperren eines einzelnen Objektes:

1. lock database in *ir* mode

2. lock extension E of the type of object o in *ir* mode

3. lock object o in r mode

Alle Attribute von o sind implizit gesperrt.

Sperren einer Extension zum Schreiben:

1. lock database in *iw* mode

2. lock extension E in w mode

Dieses Beispiel steht mit dem ersten in Konflikt.

Sperren einer Extension zum vollständigen Lesen und partiellen Schreiben:

1. lock database in iw mode

2. lock E in riw mode

Die zu schreibenden Objekte von E sind noch explizit zu sperren. Kein Konflikt zum ersten Beispiel.

Satz 4.3.1
Falls alle Transaktionen dem MGL-Protokoll gehorchen, so halten keine zwei Transaktionen zwei in Konflikte stehende Sperren.

Beweis:
Es genügt, den Satz für Blattknoten zu zeigen. Denn falls zwei Transaktionen für ein Datenelement in Konflikt stehende Sperren halten, werden für alle Nachfolgerknoten ebenfalls in Konflikt stehende (implizite) Sperren gehalten.

Wir nehmen also an, dass T_i und T_j zwei in Konflikt stehende Sperren halten. Es sind 7 Fälle zu unterscheiden:

	T_i hält:	T_j hält:	
1	implizite r-Sperre	explizite w-Sperre	auf x
2	implizite r-Sperre	implizite w-Sperre	auf x
3	explizite r-Sperre	explizite w-Sperre	auf x
4	explizite r-Sperre	implizite w-Sperre	auf x
5	implizite w-Sperre	explizite w-Sperre	auf x
6	implizite w-Sperre	implizite w-Sperre	auf x
7	explizite w-Sperre	explizite w-Sperre	auf x

ad 1 Wegen Regel 3 des MGL-Protokolls: T_i hält $rl_i[y]$ für einen Vorgänger y von x.
Wegen Regel 2 des MGL-Protokolls: T_j hält $iwl_j[z]$ für jeden Vorgänger z von x.
Insbesondere hält T_j also $iwl_j[y]$. Widerspruch.

ad 2 Wegen Regel 3 des MGL-Protokolls: T_i hält $rl_i[y]$ für einen Vorgänger y von x.
Wegen Regel 3 des MGL-Protokolls: T_j hält $wl_j[y']$ für einen Vorgänger y' von x.
Wir unterscheiden drei Unterfälle:

$y = y'$ Widerspruch.

$y > y'$ Widerspruch, da T_j $iwl_j[y]$ halten muss.
(Konflikt zu $rl_i[y]$)

$y' > y$ Widerspruch, da T_i $irl_j[y']$ halten muss.
(Konflikt zu $wl_j[y']$)

ad 3, 7 offensichtlicher Widerspruch.

ad 4, 5 analog zu 1.

ad 6 analog zu 2.

□

Um Serialisierbarkeit zu erreichen, muss MGL mit 2-PL kombiniert werden. MGL sagt lediglich, wie Sperren zu vergeben sind, 2-PL hingegen, wann.

Sperrenkonversion wird jetzt schwieriger, da mehr Sperren existieren. Hierfür ist eine Sperrkonversionstabelle sinnvoll. Diese ist so aufgebaut, dass, falls eine Sperre x gehalten und eine Sperre y angefordert wird, der Eintrag eine Sperre ist, die stärker als x und stärker als y ist:

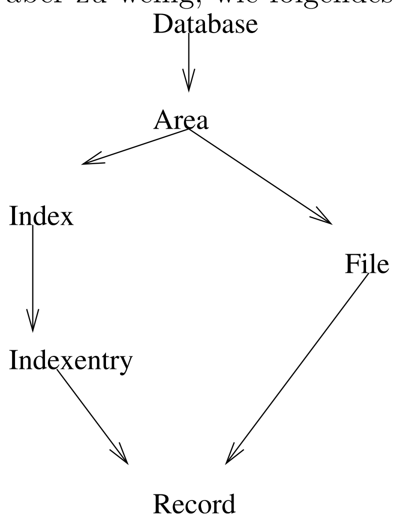
	ir	iw	r	riw	w
ir	ir	iw	r	riw	w
iw	iw	iw	riw	riw	w
r	r	riw	r	riw	w
riw	riw	riw	riw	riw	w
w	w	w	w	w	w

Senkrecht: angefordert, waagerecht: gehalten.

Es bleibt das Problem, welche Sperre auf welchem Sperrgranulat angefordert wird. Bei kleinem Sperrgranulat ergeben sich keine Probleme. Bei großem ist ein gewisses Wissen darüber notwendig, auf wie viele Datenelemente kleinerer Granularität zugegriffen wird. Dazu gibt es drei Möglichkeiten:

- Explizites Setzen der Sperre durch Transaktion
- Compiler versucht Vorhersagen
- Sperreskalation: Falls eine bestimmte Anzahl von Zugriffen überschritten wird, wird eine entsprechende Sperre für das nächsthöhere Granulat angefordert.

Bis jetzt hatten wir uns auf Bäume für die Granularitätshierarchie beschränkt. Das ist aber zu wenig, wie folgendes Beispiel zeigt:



Wir können aber das MGL-Protokoll nicht unverändert übernehmen. Insbesondere die Regeln für w und iw nicht. Wir nehmen wieder das Bankenbeispiel, um dies zu illustrieren:

Konto (Nr, Ort, Guthaben)

Einlagen (Ort, SummeGuthaben)

Weiter existiere ein Index auf *Ort* von *Konto*. Der Index-File und der Konten-File seien in Area A1 gespeichert. Transaktion T_1 halte folgende Sperren:

- *ir* auf Datenbasis
- *ir* auf A1
- *ir* auf Index
- *r* auf den Indexeintrag von Khe.

Transaktion T_2 halte

- *iw* auf Datenbasis
- *iw* auf A1
- *w* auf Konten-File

Dies ist fehlerhaft, da T_1 implizit auf allen Konten aus Karlsruhe eine Lesesperre besitzt und T_2 eine implizite Schreibsperre auf allen Konten. Für eine implizite oder explizite Sperre müssen also *ix*-Sperren für alle Vorgänger gehalten werden. Im Beispiel hieße dies, dass die Indexeinträge, Indizes, Files und Areas intentional gesperrt sein müssen, um die Kontenrecords zu sperren.

Wir ersetzen Regel 3 des MGL-Protokolls durch folgende Regeln:

3 a Falls eine Transaktion T_i ein Datenelement x lesen will, dann muss sie eine r- oder w-Sperre für x oder einen Vorgänger von x halten.

3 b Falls eine Transaktion T_i ein Datenelement x schreiben will, dann muss sie eine w-Sperre auf jedem Pfad von x zur Wurzel halten.

Der Beweis, dass keine zwei Transaktionen unter dem MGL'-Protokoll zwei in Konflikt stehende Sperren halten können, ist ähnlich zum Beweis von Satz 4.3.1.

4.4 Hotspots

[2]

4.4.1 Motivation

Beispiele für Hotspots:

- Milchflaschen
- 6mm und 8mm Schrauben

Dies sind die in einem entsprechenden Laden am häufigsten verkauften Dinge. Oft gibt es Dinge, die bei jedem Einkauf verkauft werden. Für die Datenbank heißt dies, dass (fast) alle Transaktionen darauf zugreifen. Solche Datensätze bezeichnet man als *Hotspot*.

Was kann der Programmierer tun (SCHLECHT):

```
exec sql select bestand
         into   :bestand
         from   Lager
         where  produkt_id = :id;

bestand -= 2;

exec sql update Lager
         set    bestand = :bestand
         where  produkt_id = :id;
```

Schlecht, da eine andere Transaktion die Leseanweisung auch bereits hinter sich haben kann. Es entsteht ein Deadlock, sobald die Schreibanweisungen die w-Sperre anfordern. Untersuchungen haben gezeigt, dass fast alle Deadlocks auf diese Art und Weise zustande kommen. Bei Hotspots verklemmen sich bei dieser Programmierung fast alle Transaktionen.

Eine bessere Formulierung (GUT):

```
exec sql update Lager
         set    bestand = :bestand - 2
         where  produkt_id = :id;
```

Jetzt entstehen keine Deadlocks mehr, diese wurden umgewandelt in “normales” Warten auf Sperrfreigabe. Immer noch müssen allerdings alle Transaktionen warten, falls es sich bei *produkt_id* um einen Hotspot handelt. Die Transaktionen werden rein sequentiell abgearbeitet.

4.4.2 Feldzugriffe (Field Calls)

Betrachten wir noch einmal obiges Beispiel, insbesondere die Anweisung

```
exec sql update Lager
         set    bestand = :bestand - 2
         where  produkt_id = :id;
```

Die Idee von Feldzugriffen ist es, eine Aktion auf einem Hotspot-Record in zwei Teile zu zerlegen,

1. ein Prädikat und
2. eine Transformation.

Für unser Beispiel:

1. Prädikat: bestand > 2
2. Transformation: bestand = :bestand -2;

In manchen Fällen benötigt man kein Prädikat, in anderen keine Transformation.

Das Protokoll von Feldzugriffen kann dann wie folgt beschrieben werden:

1. Sofortiger Test des Prädikates unter kurzer Lesesperre.
   (Die Sperre auf das ungeänderte Datenelement wird freigegeben, sobald der Test beendet ist.)

2. Falls der Test zu falsch evaluiert, wird ein Fehler gemeldet.

3. Ansonsten wird ein REDO Log-Eintrag angelegt, der bei Transaktionsende ausgeführt wird. Dieser enthält sowohl das Prädikat als auch die Transformation.

4. Das Commit teilt sich in 2 Phasen:

   Phase 1 Alle REDO Log-Einträge der beendenden Transaktion werden bearbeitet, Lesesperren werden angefordert für Feldzugriffe, die keine Transformation beinhalten, Schreibsperren für alle anderen Feldzugriffe. Danach werden alle Prädikate noch einmal evaluiert. Falls nur eines zu falsch evaluiert, wird die Transaktion zurückgesetzt. Ansonsten Eintritt in Phase 2 des Commits.

   Phase 2 Alle Transformationen werden angewendet und die Sperren freigegeben.

Bei dem Milchverkauf im Lebensmittelgeschäft wird kein Prädikat benötigt, da der Kassierer ja die Milch direkt sieht, sie also noch vorhanden sein muss. Also genügt:

```
exec sql update hotspot Lager
         set            bestand = :bestand - 2
         where          produkt_id = :milch_id;
```

Bei der Bearbeitung von Bestellungen, in denen Aufträge an ein “nicht unmittelbar sichtbares” Lager weitergereicht werden, werden Prädikate benötigt. Also benötigen wir folgende Anweisung:

```
exec sql update hotspot Lager
         set            bestand = :bestand - 2
         where          produkt_id = :id
         and            bestand > 2;
```

Nachteile:

- Die Prädikatevaluierung in Phase 1 des Commits kann immer noch fehlschlagen, wodurch die Transaktion dann zurückgesetzt werden muss.

- Mehrfache updates auf demselben Eintrag (100 mal nur 2 Milchflaschen kaufen, bei einem Bestand von nur 100 Flaschen).
  Dieses Problem kann einfach gelöst werden, indem man die Prädikatüberprüfungen und die Transformationen sortiert.

- Falls die Transaktion den Eintrag liest, so erhält sie immer den ursprünglichen Wert. Konsistentes Lesen von Hotspots ist immer noch ungelöst, aber vgl. nächsten Abschnitt.

4.4.3 Escrow-Sperren

Eine einfache Verbesserung verhindert das Fehlschlagen des Prädikattests. Sie funktioniert nur für geordnete Domänen, also beispielsweise Zahlen. Ein Escrow gibt dann ein Intervall an, in dem sich der tatsächliche Wert befinden muss:

TA1	TA2	TA3	qoh	Escrow
			1000	[1000,1000]
qoh > 150?				[850,1000]
qoh -= 150;				
	qoh > 800?		1000	[50,1000]
	qoh -= 800			
	commit		200	[50,200]
		qoh > 100?F	200	
		qoh -= 100		
		commit		

Transaktion 3 wird verzögert. Man beachte, dass TA1 hätte zurückgesetzt werden müssen, wenn TA2 ausgeführt worden wäre, da der Test ja zu wahr evaluiert. Dies wäre beim einfachen Feldzugriff der Fall gewesen.

Es werden 2 verschiedene Leseoperationen unterschieden:

1. Standardlesen, das Wert zurückliefert
2. Escrowlesen, das wahr oder falsch zurückliefert, also nur für Prädikatauswertung genutzt werden kann.

Escrows lösen nicht alle Probleme, schon gar nicht das Serial-number-Problem. Alle Vorgänge sollen durch f nummeriert werden. Hier werden folgende Forderungen aufgestellt:

Monotonie falls $c_i < c_j$, dann $f(i) < f(j)$

Lesbarkeit Jede Transaktion T_i erfährt ihre Nummer $f(i)$ vor Commit

Dense Die Nummern $f(i)$ sollen dicht liegen

Rollback Transaktionen können zurückgesetzt werden

High Throughput 10-100 Transaktionen pro Sekunde sollen möglich sein

Wie sieht es mit der Lösbarkeit aus?

Kapitel 5

Sperrverfahren für Bäume

Bis jetzt haben wir nur Sperrverfahren für einzelne Datenelemente betrachtet. In diesem Kapitel betrachten wir spezielle Verfahren für Bäume. Dies ist besonders wichtig, da Bäume besonders häufig eingesetzte Indexverfahren sind (Bsp.: B^+-Baum). Da viele Transaktionen vorhandene Indizes benutzen werden, besteht hier häufig die Gefahr eines Engpasses, genauer verminderter Nebenläufigkeit.

5.1 Einfaches Baumsperrverfahren

Wir unterscheiden nicht zwischen Lese- und Schreiboperationen. Stattdessen gibt es nur noch eine Zugriffsfunktion $a_i[x]$, die besagt, dass eine Transaktion T_i auf ein Datenelement x, hier ein Knoten eines Baumes, zugreift. Falls $i \neq j$, gilt $a_i[x] \not\parallel a_j[x]$. Die zugehörige Sperre ist $al_i[x]$. Für $i \neq j$ sind $al_i[x]$ und $al_j[x]$ konsequenterweise unverträglich. Sperrfreigabe wird durch $au_i[x]$ bezeichnet.

Wir setzen voraus, dass die Daten hierarchisch in einem Datenbaum (DT) organisiert sind. Die Wurzel bezeichen wir mit r. Damit ergibt sich folgendes einfaches Baumsperrprotokoll (TL):

1. Vor Ausführung von $a_i[x]$ muss $al_i[x]$ gesetzt werden.

2. $al_i[x]$ kann nur gesetzt werden, falls kein $al_j[x]$ gesetzt ist für $i \neq j$.

3. Falls $x \neq r$ nicht die Wurzel ist, so kann $al_i[x]$ nur gesetzt werden, falls $al_i[y]$ gesetzt ist, 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 folgen, das heißt, dass eine Sperre nach Freigabe nicht noch einmal angefordert werden kann.

Regeln 3 und 5 implizieren, dass der Scheduler $al_i[x]$ nur freigeben kann, falls die Transaktion alle benötigten Sperren auf den benötigten Söhnen von x bereits hält. Dies heißt auch *Sperrkopplung* (*lock coupling*). Man beachte, dass eine Sperranforderungsrichtung von der Wurzel zu den Blättern impliziert wird.

Proposition 5.1.1
Falls T_i einen Knoten x vor T_j sperrt, so sperrt T_i auch jeden Nachfolger y von x, der sowohl von T_i als auch von T_j gesperrt wird, vor T_j.

Satz 5.1.2
Jede TL-Historie ist serialisierbar.

Beweis:
Betrachte $T_i \to T_j \in SG(H)$ für TL-Historie H.
$\succ \exists a_i[x], a_j[x] \in H \quad a_i[x] \not\parallel a_j[x] \wedge a_i[x] <_H a_j[x]$
$\succ au_i[x] <_H al_j[x]$
$\succ au_i[r] <_H al_j[r]$
Dies gilt nicht nur für die Wurzel, sondern für alle Pfade zur Wurzel. Annahme: $SG(H)$ hat einen Zyklus der Form

$$T_1 \to \ldots \to T_n \to T_1$$

$\succ au_1[r] <_H al_1[r]$ Widerspruch zu TL-Regel 5.

□

Satz 5.1.3 *Das TL-Protokoll vermeidet Deadlocks.*

Beweis:
Falls T_i auf die Wurzelsperre wartet, so kann sie nicht in einem Deadlock involviert sein, das sie keine Sperren besitzt.
Annahme: T_i wartet auf $au_j[x]$, $x \neq r$. $\succ au_j[r] <_H al_i[r]$
Durch Induktion: Falls der Wartegraph einen Zyklus hat, der T_i beinhaltet, so $au_i[r] <_H al_i[r]$. Widerspruch.

□

Neben der Verklemmungsfreiheit besteht ein weiterer Vorteil des TL-Protokolls:

- Sperren können eher als bei 2-PL freigegeben werden.

Falls alle benötigten Nachfolger von x gesperrt sind, kann die Sperre von x aufgegeben werden.
Problem:

- Wie kann dies entschieden werden?

Antworten:

- falls alle Nachfolger gesperrt sind
- durch Hinweise des Transaktionsmanagers

Falls weder das eine noch das andere vorliegt, degeneriert TL zu striktem 2-PL.

Diskussion: Frühe Freigabe von Sperren erhöht aber die Nebenläufigkeit: Weniger Sperren werden gehalten, weniger Konflikte, weniger Warten. Dies wird durch TL aber nur erreicht, wenn der Baum von der Wurzel zu den Blättern durchlaufen wird. Falls dies nicht der Fall ist, kann TL die Nebenläufigkeit verringern.

Um Wiederanlauf zu ermöglichen und kaskadierendes Rücksetzen zu vermeiden, muss TL so verschärft werden, dass Sperren bis zum Transaktionsende gehalten werden. Dadurch verringert TL die Nebenläufigkeit.

5.2 Variationen von TL

5.2.1 Beliebiger Einstieg

TL verlangt den Einstieg in einen Baum bei der Wurzel. Dies ist nicht notwendig, jeder beliebige Knoten kann dazu verwendet werden. Hieraus ergibt sich mehr Nebenläufigkeit.

5.2.2 Lese- und Schreibsperren

TL kann so verallgemeinert werden, dass Lese- und Schreibsperren behandelt werden können. Falls jede Transaktion nur Lesesperren oder nur Schreibsperren anfordert, so genügen die normalen Konfliktregeln, um Serialisierbarkeit zu gewährleisten. Dies ist nicht der Fall, falls eine Transaktion Lese- und Schreibsperren setzt.

Beispiel 5.2.1
Betrachte folgenden Baum

$$\begin{array}{c} x \\ \downarrow \\ y \\ \downarrow \\ z \end{array}$$

und die Historie

$$H = wl_1[x]rl_1[y]wu_1[x]wl_2[x]rl_2[y]wu_2[x]wl_2[z]ru_2[y]wu_2[z]wl_1[z]ru_1[y]wu_1[z]$$

Man beachte, dass $wl_1[x]$ vor $wl_2[x]$ ausgeführt wird, aber $wl_2[z]$ vor $wl_1[z]$. H ist also nicht SR. Dies ist möglich, da T_2 T_1 in dem Moment überholt hat, da T_1 nur noch eine Lesesperre hielt.

Die Lösung dieses Problems: Für jeden Pfad $x_1, \ldots, x_n$, falls eine Transaktion eine Schreibsperre auf x_1 und x_n hält und Lesesperren auf den dazwischen liegenden Elementen, so erhält sie Schreibsperren auf allen Elementen.

5.2.3 DAG-Sperren

Hier sollen Wurzel-DAGS gesperrt werden. Das zugehörige Protokoll DL unterscheidet sich von TL nur in einem Punkt, nämlich Regel 3. Alle anderen (1,2,4-6) bleiben unverändert. Die modifizierte Regel 3 lautet:

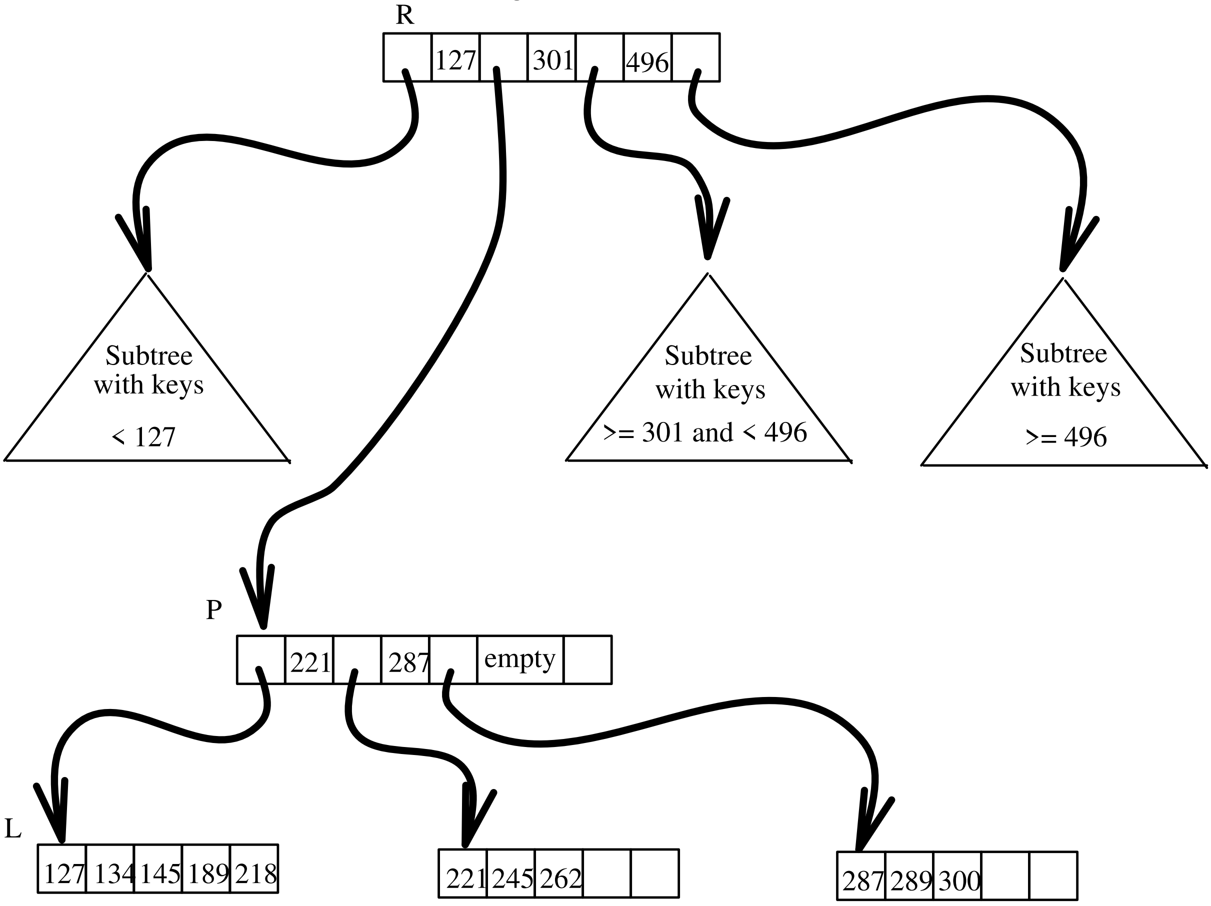
3. Für Nicht-Wurzelknoten x, falls T_i eine Sperre für x anfordert, so muss T_i eine Sperre auf *einem* der Vorgängerknoten halten und zu einem Zeitpunkt Sperren auf *allen* Vorgängerknoten besessen haben.

Selbstverständlich kann man hier die vorher genannten Variationen einbringen.

5.3 B-Baum-Protokolle

5.3.1 Einfache Möglichkeiten

Die für uns wichtigen Operationen sind *search, insert* und *delete*. Da *delete* analog zu *insert* synchronisiert wird, interessieren uns nur *search* und *insert*. Um uns an den B-Baum zu erinnern, betrachten wir folgendes Beispiel:



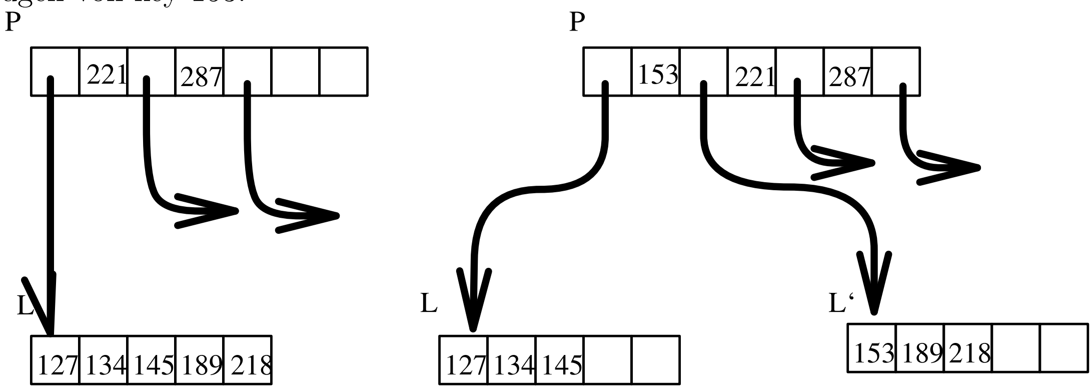
Wir suchen nach dem Eintrag mit dem Schlüssel 134:

1. Die Wurzel wird besucht, und der Eintrag $[127, 301[$ ist der gesuchte.

2. Knoten P wird besucht, und der Eintrag $[127, 221[$ ist der gesuchte.

3. Knoten L wird besucht, und der Eintrag mit Schlüssel 134 wird gefunden.

Beim Einfügen wird zunächst die passende Blattseite L gesucht. Falls dort Platz ist, kann das Paar [key,data] dort gespeichert werden. Falls nicht, so muss eine Spaltung stattfinden. Hierbei wird eine neue Seite L' angelegt und die auf L enthaltenen Paare gleichmäßig auf L und L' verteilt. Die Vorgängerseite von L muss nun um den Eintrag von L' erweitert werden. Falls dort kein Platz ist, setzt sich die Spaltung fort. Bsp.: Einfügen von key 153:



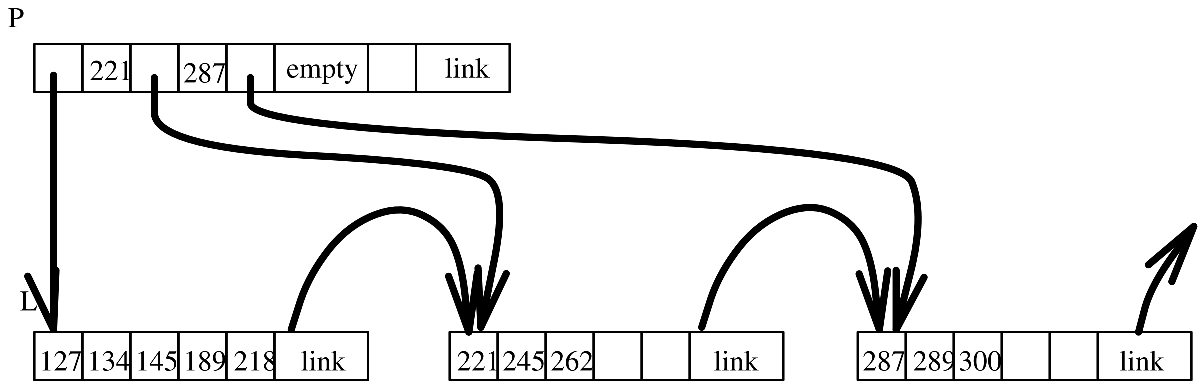
Wir könnten mit 2PL arbeiten, aber da jede Operation bei der Wurzel startet, würde sie gleich den ganzen Baum sperren. Was benötigt wird, ist Sperrfreigabe für nicht mehr benötigte Sperren. Da die inneren Knoten eigentlich redundant sind, reicht es aus, sich auf die Sperren der Einträge in den Blattseiten zu verlassen. Der Rest könnte mit kurzfristigen Sperren (oder Semaphoren) erledigt werden.

TL hilft hier ein bisschen weiter. Es ergibt sich nur folgendes Problem: Eine *insert*-Operation weiß erst bei Erreichen der Blattseite, ob ein Split nötig ist. Eine Möglichkeit ist, zunächst Schreibsperren auf jede zu besuchende Seite zu setzen. Falls diese sich dann als nicht voll erweist, kann sie in eine Lesesperre umgewandelt werden. Dies kann aber zu erheblich weniger Nebenläufigkeit führen. Eine andere Möglichkeit ist, zunächst nur Lesesperren zu setzen und diese bei Bedarf in Schreibsperren umzuwandeln. Das kann aber leicht Deadlocks verursachen. Eine Möglichkeit, diese zu verhindern, wäre dann die Einführung von *intention write*-Sperren. Obiges entspricht der *seitenorientierten Synchronisation.*

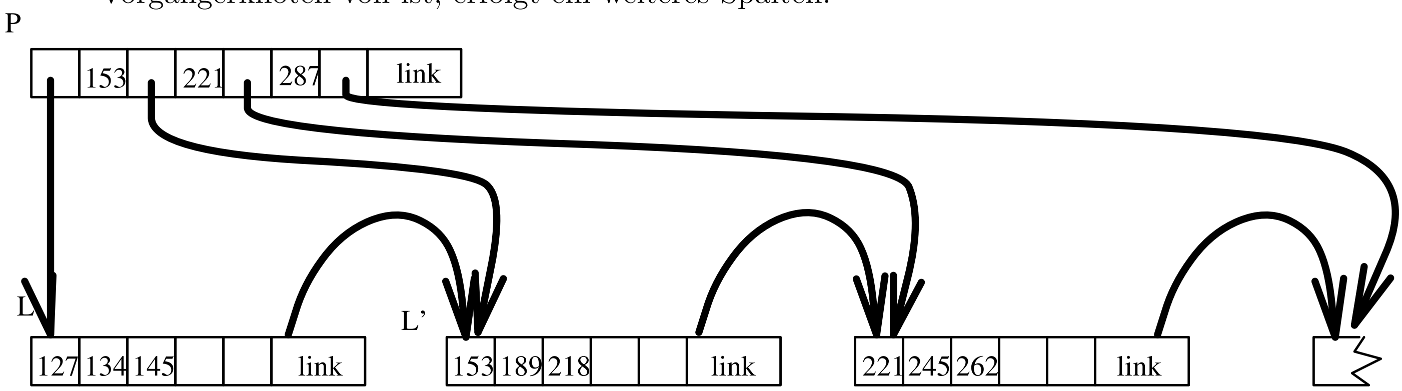
Wir betrachten noch kurz die *tupelorientierte Synchronisation.* Ein Problem, wenn man sich nur auf das Sperren der Einträge in einem B-Baum verlässt, wie es ja möglich ist, da der Überbau redundant ist, ist das Phantomproblem. Hier helfen Schlüsselbereichssperren (key range locks), die über die gesamte Transaktionsdauer gehalten werden. Dieser Bereich kann auch nur ein einzelner Schlüsselwert sein, falls es sich um einen Sekundärindex handelt (mehrere Einträge pro Schlüssel).

5.3.2 Benutzung von Links

Wir wollen nun die Sperrkopplung von TL ganz aufgeben. Hierfür werden wir die Verkettung der Seiten einer Ebene des B-Baums ausnutzen:



Zunächst modifizieren wir *insert* so, dass es einen Update auf einer Seite durchführen kann, ohne eine Sperre auf dem Vorgängerknoten zu besitzen. Falls kein Spalten notwendig ist, ist dies trivial. Falls es notwendig ist, so wird zunächst eine neue Seite L' erzeugt, dann die "obere Hälfte" der Einträge der vollen Seite L in die neue Seite eingetragen. Die Links von L und L' werden aktualisiert. Jetzt kann *insert* die Sperren auf L und L' freigeben. Es hält keine weiteren Sperren! Im zweiten Schritt wird eine w-Sperre auf dem Vorgängerknoten angefordert und der Zeiger von L' dort eingetragen. Falls der Vorgängerknoten voll ist, erfolgt ein weiteres Spalten.



Die Operation *search* verläuft wie bisher, nur dass direkt nachdem eine Seite gelesen wurde, die entsprechende Sperre wieder freigegeben wird. Danach wird die Sperre der Nachfolgerseite angefordert und diese gelesen. Es gibt also einen Moment, in dem auch *search* keine Sperre hält. Dies kann nur zu Problemen führen, wenn es von einem *insert* überholt wird. Ist dies der Fall, so kann es passieren, dass *search* den gesuchten Schlüssel auf einer Blattseite nicht findet. Dann sind aber alle auf dieser Seite befindlichen Schlüssel kleiner als der von *search* gesuchte. Also kann *search* dem Link folgen und auf der Nachbarseite nachsehen.

Da beide Operationen, *search* und *insert*, nur dann Sperren anfordern, wenn sie keine halten, kann es zu keinem Deadlock kommen.

Kapitel 6

Nicht-sperrende Synchronisationsverfahren

6.1 Zeitstempel-basierte Synchronisation

Bei diesem Verfahren bekommt jede Transaktion T_i einen eindeutigen Zeitstempel $ts(T_i)$ zugeordnet. Wir werden davon ausgehen, dass jede Operation einer Transaktion denselben Zeitstempel trägt.

TO-Scheduler ordnen in Konflikt stehende Operationen gemäß ihrer Zeitstempel. Das TO-Protokoll enthält folgende Regel:

1. $\forall p_i[x] \not\parallel q_j[x] \in H (i \neq j) \;\; p_i[x] <_H q_j[x] \prec\succ ts(T_i) < ts(T_j)$

Historien, die TO genügen, sind SR:

Satz 6.1.1 *Jede Historie, die TO genügt, ist serialisierbar.*

Beweis:
$T_i \rightarrow T_j \in SG(H)$
$\succ \;\; \exists p_i[x] \not\parallel q_j[x] \in H \;\; p_i[x] <_H q_j[x]$
$\succ \;\; ts(T_i) < ts(T_j)$
Annahme: existiert Zyklus $T_1, \ldots, T_n, T_1 \in SG(H)$
durch Induktion: $ts(T_1) < ts(T_1)$ Widerspruch.

□

Wir werden jetzt Möglichkeiten zur Einhaltung von TO vorstellen.

6.1.1 Basis-TO

Basis-TO ist eine einfache und aggressive Implementierung von TO. Alle Operationen werden unmittelbar an den DM weitergeleitet (FCFS-Ordnung). Operationen, die zu spät kommen, werden zurückgewiesen. Dabei ist eine Operation $p_i[x]$ zu spät, falls gilt:

$$\exists\, q_j[x] \not\parallel p_i[x] \quad q_j[x] <_H p_i[x] \quad \wedge \quad ts(T_j) > ts(T_i)$$

Da $q_j[x]$ in so einem Fall schon ausgeführt ist, muss $p_i[x]$ zurückgewiesen werden, um TO zu garantieren. Dies bedeutet aber, dass T_i abgebrochen werden muss. Bei erneuter Ausführung von T_i muss T_i natürlich einen neuen, höheren Zeitstempel erhalten.

Zur Feststellung, ob eine Operation zu spät ist, hält der Basis-TO-Scheduler für jedes x das Maximum aller Zeitstempel für ausgeführte r- und w-Operationen:

- max-r-scheduled[x]
- max-w-scheduled[x]

Bei Eintreffen von $p_i[x]$ vergleicht der Scheduler $ts(T_i)$ mit allen $max - q - schedule[x]$, wobei q mit p in Konflikt steht. Falls $ts(T_i) < max{-}q{-}scheduled[x]$ gilt, wird $p_i[x]$ zurückgewiesen. Andernfalls wird $p_i[x]$ an den DM weitergereicht und $max - p - scheduled[x]$ auf $ts(T_i)$ gesetzt.

Es ist darauf zu achten, dass Operationen in der Reihenfolge ausgeführt werden, wie sie an den DM gesendet werden. Hierzu ist ein hand-shake-Verfahren notwendig. (Dieses war bei 2PL nicht notwendig, da Operationen automatisch verzögert wurden, falls sie in Konflikt standen.) Für dieses Hand-shake-Verfahren hält der Scheduler für jedes x ein Zähler für an den DM gesendete r- und w-Operationen, deren Ausführung noch nicht bestätigt wurde:

- r-in-transit[x]
- w-in-transit[x]

Dabei ist w-in-transit[x] entweder 0 oder 1, da zwei Schreiboperationen in Konflikt stehen. Die Zähler werden erniedrigt, sobald das ack für die entsprechende Operation vom DM eintrifft. Weiter gibt es für jedes x noch eine *queue-[x]* für noch einzuplanende Operationen.

6.1.2 Striktes TO

Betrachte T_1, T_2 mit $ts(T_i) = i$ und

$$H = w_1[x]r_2[x]w_2[y]c_2$$

Diese Historie erfüllt TO, ist aber nicht wiederanlaufbar. Wir werden nun einen TO-Scheduler vorstellen, der strikte Historien garantiert.

Der strikte TO-Scheduler arbeitet wie der Basis-TO-Scheduler, außer dass w-in-transit[x] erst bei a_i oder c_i auf 0 gesetzt wird. Hierdurch werden $r_j[x]$ und $w_j[x]$ mit $ts(T_j) > ts(T_i)$ verzögert, bis T_i beendet ist. w-in-transit[x] verhält sich wie eine Sperre. Trotzdem: Deadlocks können nicht auftreten, da T_j nur auf T_i wartet, falls $ts(T_j) > ts(T_i)$.

Betrachte

$$H = r_2[x]w_3[x]c_3w_1[y]c_1r_2[y]w_2[z]c_2$$

H ist SR und strikt. Letzteres, da zwar $w_1[y] <_H r_2[y]$, aber auch $c_1 <_H r_2[y]$.

Falls $ts(T_1) < ts(T_2) < ts(T_3)$, ist H strikt TO. H ist aber nicht 2PL, da $ru_2[x] < wl_3[x]$ gelten müsste, aber damit würde T_2 eine Sperre freigeben und danach weitere anfordern.

6.1.3 Konservatives TO

Falls Basis-TO die Operationen in einer Reihenfolge erhält, die stark von der Zeitstempelordnung abweicht, so müssen viele Transaktionen abgebrochen werden. Dies wird durch die Aggressivität von Basis-TO bedingt. Ein konservativer TO-Scheduler verzögert Operationen künstlich. Dies führt zu weniger Zurücksetzen.

Ein ultimativ konservativer Scheduler könnte beispielsweise die Vorankündigung der Lese- und Schreibmengen verlangen (analog zu 2PL). Wir betrachten hier jedoch einen ultimativ konservativen Scheduler, der auf einer anderen Annahme aufbaut. Diese ist:

> Der TM sendet die Operationen in Zeitstempelreihenfolge.

Dies kann erreicht werden, wenn jeder TM zu jedem Zeitpunkt nur eine Transaktion durchführt. Außerdem erhöhen die TMs den Zeitstempel bei jedem Aufruf. Also führt jeder TM die Transaktionen seriell durch. (Dies ist keine besonders harte Annahme (client/server).) Da es aber viele TMs gibt, erhält der Scheduler die TAs nicht seriell.

Ein ultimativ konservativer Scheduler managt jetzt eine Warteschlange *unsched-queue*, in der alle noch nicht eingeplanten Operationen gehalten werden. Diese ist sortiert nach aufsteigender Zeitstempelordnung. Eine neue Operation $p_i[x]$ wird dann entsprechend einsortiert. Die nächste Operation $q_j[x]$ ist immer eine vom Kopf. Diese kann eingeplant werden, falls

1. unsched-queue mindestens eine Operation eines jeden TMs enthält und

2. alle mit $q_j[x]$ in Konflikt stehenden Operationen, die schon zum DM gesendet wurden, bestätigt wurden.

Für Punkt 1 verwaltet der Scheduler Zähler *op-count[v]* für jeden TM_v. *unsched-queue* enthält dann Paare der Form $(v, o_i[x])$.

Punkt 1 kann zu sehr langem Warten führen. Daher senden die TMs in gewissen Abständen NULL-Operationen. Aber selbst dann führt der ultimativ konservative TO nur zu seriellen Schedules. Es gibt verschiedene Techniken, dies zu ändern. Eine basiert auf einer Klassifikation der Transaktionen entsprechend ihrer READ- und WRITE-Mengen. Wir gehen jedoch nicht näher darauf ein.

6.1.4 Serialisierbarkeitsgraph-basiertes Scheduling

Bis jetzt wurden Scheduler vorgestellt, die auf Sperren oder Zeitstempeln arbeiten. Eine dritte Art von Schedulern basiert auf Serialisierbarkeitsgraphtests. Ein SGT-Scheduler baut den SG der aktuellen Historie explizit auf. Operationen ändern den SG. Der SGT-Scheduler passt auf, dass der SG immer zyklenfrei bleibt. Es sind aber Anpassungen des SG's notwendig:

- Der SG enthält nur committete TAs. Diese Bedingung fällt weg.

- Der SG enthält alle committeten TAs. Diese Bedingung fällt auch weg.

Wir bezeichnen einen solchen "SG" dann als SSG.

Basis SGT

Falls der SGT-Scheduler eine Operation $p_i[x]$ von einem TM erhält, so

1. falls $TA_i \notin SSG$: einfügen
2. $\forall q_j[x] <_H p_i[x], i \neq j, q_j[x] \not\parallel p_i[x] \quad SSG \cup T_j \to T_i$
3. falls SSG jetzt Zyklus enthält: abort T_i. nach acknowledgement von DM: $SSG \backslash T_j \to T_i$
4. sonst: schedule $p_i[x]$, falls alle in Konflikt stehenden Operationen schon acknowledged. Hierzu wieder q-in-transit[x].

Zusätzlich für jede TA_i

r-scheduled[TA_i] und w-scheduled[TA_i]

in denen die Lese- und Schreibmengen von TA_i gehalten werden.

Wann können TAs aus SSG gelöscht werden? Nicht direkt nach commit! Betrachte

$$H = r_{k+1}[x]w_1[x]w_1[y_1]c_1w_2[x]w_2[y_2]c_2 \ldots w_k[x]w_k[y_k]c_k$$

Jetzt erreicht $w_{k+1}[z]$ den Scheduler. Dieser muss testen, ob $z \in \{x, y_1, \ldots, y_k\}$. Hierzu benötigt er jedoch genau diese Menge, obwohl die entsprechenden Transaktionen schon committet sind.

Transaktionen können aus SSG entfernt werden, wenn sie nicht mehr an einem Zyklus teilnehmen können. Dafür benötigen sie aber mindestens eine einfallende und eine ausgehende Kante. Committete Transaktionen ohne einfallende Kanten können aus SSG entfernt werden, da nur ausgehende Kanten nach einen Commit entstehen können.

Es gibt auch wieder eine konservative Variante. Um Striktheit zu gewährleisten, geht man analog zu striktem TO vor.

6.1.5 Optimistische Verfahren

Optimistische Verfahren prüfen nicht jede Operation, sondern nur zum Commit-Zeitpunkt. Es gibt verschiedene optimistische Verfahren zu 2PL, TO und SGT.

6.1.6 Integrierte Scheduler

Scheduler aus verschiedenen Verfahren zusammensetzen.

Kapitel 7

Mehrversionen-Synchronisation

Jedes Schreiben eines Datenelementes x erzeugt eine neue Version. Daher ist es bspw. möglich, Leseoperationen auf alten Versionen zu erlauben, anstatt diese zurückzuweisen. Dies ist nicht unbedingt kostspieliger, da für den Wiederanlauf oft mehrere Versionen (bspw. before-images) verwaltet werden müssen.

Falls eine Transaktion abbricht, so werden die von ihr erzeugten Versionen vernichtet. Eine Version bezieht sich immer auf den Wert eines Datenelementes. Dieser kann von einer abgeschlossenen oder aktiven Transaktion stammen. Falls der Scheduler eine bestimmte Version von x einer Leseoperation $r_j[x_i]$ ($i =$ Versionsnummer $=$ Transaktionsnummer der schreibenden TA) zuweist, darf dieser Wert nicht von einer abbgebrochenen Transaktion stammen. Falls die schreibende Transaktion T_i noch aktiv ist, so muss a_j verzögert werden bis nach a_i. Falls T_i abbricht, so muss auch T_j abgebrochen werden.

Die verschiedenen Versionen sind für den Benutzer nicht sichtbar! Sie haben nichts mit den Versionen zu tun, die bspw. in der OO-Vorlesung zur Sprache kamen.

7.1 Einleitung

Für eine Serialisierbarkeitstheorie benötigen wir Erweiterungen der bisherigen Serialisierbarkeitstheorie:

- Mehrversionenhistorien (MV)
  Diese repräsentieren die Ausführung des DM.

- Einversionenhistorien (1V)
  Diese repräsentieren die Interpretation von MV-Historien für eine einzelne Benutzertransaktion

Serielle 1V-Historien sind dann diejenigen, die vom Benutzer als korrekt erachtet werden.

Um den Nachweis der Korrektheit eines Synchronisationsverfahrens zu liefern, müssen wir nachweisen, dass die MV-Historie äquivalent ist zu einer seriellen 1V-Historie. Hierzu benötigen wir zunächst den Begriff der Äquivalenz.

Für jedes Datenelement x bezeichne $w_i[x_i]$ das Schreiben der Version i von x. Dabei wird der Versionsidentifikator gleich dem Transaktionsidentifikator gesetzt. Lesen wird wie üblich als $r_j[x_i]$ bezeichnet.

Nehmen wir an, eine MV-Historie H_{MV} ist äquivalent zu einer 1V-Historie H_{1V}, falls in Konflikt stehende Operationen in beiden Historien die gleiche Ordnung aufweisen. Betrachte die MV-Historie

$$H_1 = w_0[x_0]c_0w_1[x_1]c_1r_2[x_0]w_2[y_2]c_2$$

Dann sind $w_0[x_0]$ und $r_2[x_0]$ die einzigen in Konflikt stehenden Operationen. Betrachte zusätzlich die 1V-Historie

$$H_2 = w_0[x]c_ow_1[x]c_1r_2[x]w_2[y]c_2$$

H_2 wurde aus H_1 durch Eliminieren der Versionsnummern gewonnen. Die Operationen $w_0[x_0]$ und $r_2[x_0]$ sind in der gleichen Reihenfolge geordnet. Demnach währen die beiden Historien äquivalent. Dennoch ist dies schlecht, da $T_2 \rhd_{H_1[x]} T_0$ und $T_2 \rhd_{H_2[x]} T_1$ gilt. Also liest T_2 in H_1 und H_2 unterschiedliche Werte. Durch das Weglassen von Versionsnummern werden also Konflikte in der 1V-Historie erzeugt, die in der MV-Historie nicht vorkamen.

Der Äquivalenzbegriff, der zu verwenden ist, ist derjenige der Sichtenäquivalenz. Mit diesem sind H_1 und H_2 nicht mehr äquivalent.

Jetzt benötigen wir eine Möglichkeit zu zeigen, dass jede MV-Historie äquivalent zu einer 1V-Historie ist. Ansatz: $SG(H)$ ist azyklisch. Dies funktioniert nicht, da nicht jede serielle MV-Historie äquivalent zu einer seriellen 1V-Historie ist. Betrachte die MV-Historie

$$H_3 = w_0[x_0]w_0[y_0]c_0r_1[x_0]r_1[y_0]w_1[x_1]w_1[y_1]c_1r_2[x_0]r_2[y_1]c_2$$

Obwohl H_3 seriell ist und $SG(H_3) =$

$$\begin{array}{ccccc} T_0 & \rightarrow & T_1 & \rightarrow & T_2 \\ & \searrow & \rightarrow & \nearrow & \end{array}$$

keinen Zyklus enthält, ist sie zu keiner seriellen 1V-Historie äquivalent.

Betrachte als Beispiel folgende serielle 1V-Historie:

$$H_4 = w_0[x]w_0[y]c_0r_1[x]r_1[y]w_1[x]w_1[y]c_1r_2[x]r_2[y]c_2$$

Es gilt:

- $T_2 \rhd_{H_4} [x, y]T_1$
- $T_2 \rhd_{H_3} [x]T_0$
- $T_2 \rhd_{H_3} [y]T_1$

Also sind H_3 und H_4 nicht sichtenäquivalent. Betrachte jetzt

$$H_5 = w_0[x]w_0[y]c_0r_2[x]r_2[y]c_2r_1[x]r_1[y]w_1[x]w_1[y]c_1$$

Es gilt:

- $T_1 \rhd_{H_3[x,y]} T_0$
- $T_1 \rhd_{H_5[x,y]} T_2$

und auch

- $T_2 \rhd_{H_5} [x,y]T_0$
- $T_2 \rhd_{H_3} [x]T_0$
- $T_2 \rhd_{H_3} [y]T_1$

Also sind H_3 und H_5 nicht sichtenäquivalent. Gleiches gilt für alle seriellen 1V-Historien.

Nur eine Teilmenge aller seriellen MV-Historien, die sogenannten *1-seriellen MV-Historien*, sind äquivalent zu seriellen 1V-Historien. Sie können wie folgt charakterisiert werden:

> $T_i \rhd_x T_j$, dann ist T_j die letzte Transaktion, die irgendeine Version von x schreibt.

Diese Bedingung war in H_3 verletzt: $T_2 \rhd_{H_3} [x]T_0$, obwohl T_1 x nach T_0 schreibt.

Alle 1-seriellen MV-Historien sind dann äquivalent zu 1V-Historien. Um zu zeigen, dass eine MV-Historie äquivalent zu einer 1-seriellen MV-Historie ist, benutzen wir Mehrversionenserialisierbarkeitsgraphen (MVSG). Dann gilt:

> MV ist 1-seriell $\prec\succ$ MVSG ist azyklisch.

Wir formalisieren dieses jetzt.

7.2 MV-Serialisierbarkeitstheorie

7.2.1 MV-Historien und Äquivalenz

Definition 7.2.1 (Übersetzung) *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. h wird als Übersetzung *bezeichnet.*

Definition 7.2.2 (vollständige MV-Historie)
Eine vollständige MV-Historie H über T ist eine partielle Ordnung $<_H$, wobei gilt

1. *$H = h(\cup_{i=0}^{n} T_i)$ für eine Übersetzung h*
2. *$\forall T_i \forall p_i, q_i \in T_i \quad p_i <_i q_i \succ h(p_i) <_H h(q_i)$*
3. *$h(r_j[x]) = r_j[x_i] \quad \succ \quad w_i[x_i] <_H r_j[x_i]$*
4. *$w_i[x] <_i r_i[x] \quad \succ \quad h(r_i[x]) = r_i[x_i]$*

5. $h(r_j[x]) = r_j[x_i], i \neq j, c_j \in H \succ c_i <_H c_j$

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

Bedingung 4 besagt, dass H reflexive liest-von-Beziehungen bewahrt.
Bedingung 5 besagt, dass H wiederanlaufbar ist.

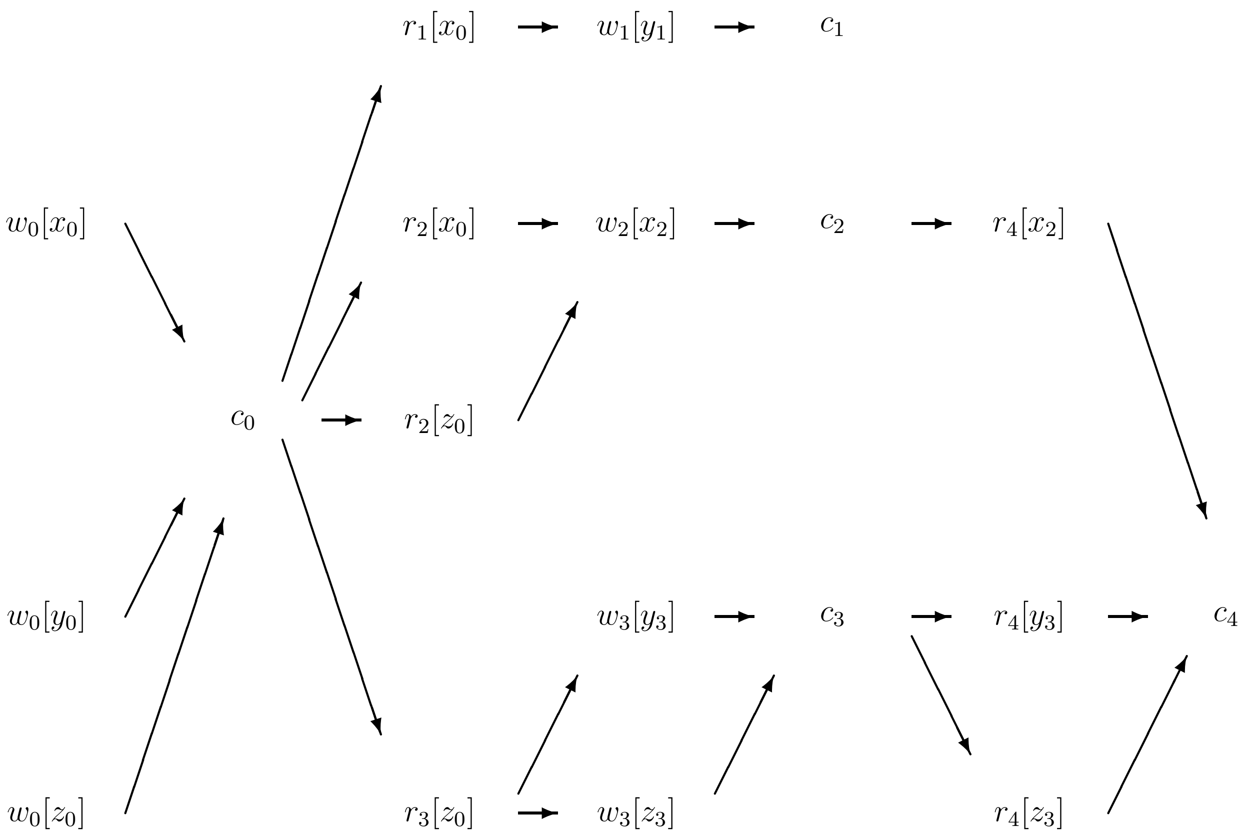
Die abgeschlossene Projektion $C(H)$ wird wie vorher definiert. Alle vollständigen Historien über einer gegebenen Menge von Transaktionen haben die gleichen Schreiboperationen, nicht aber unbedingt die gleichen Leseoperationen, deren Versionsidentifikator kann unterschiedlich sein.

$$T_0 = \begin{array}{ccc} w_0[x] & \searrow & \\ w_0[y] & \rightarrow & c_0 \\ w_0[z] & \nearrow & \end{array} \qquad T_3 = \begin{array}{ccccc} & \nearrow & w_3[y] & \searrow & \\ r_3[z] & & & & c_3 \\ & \searrow & w_3[z] & \nearrow & \end{array}$$

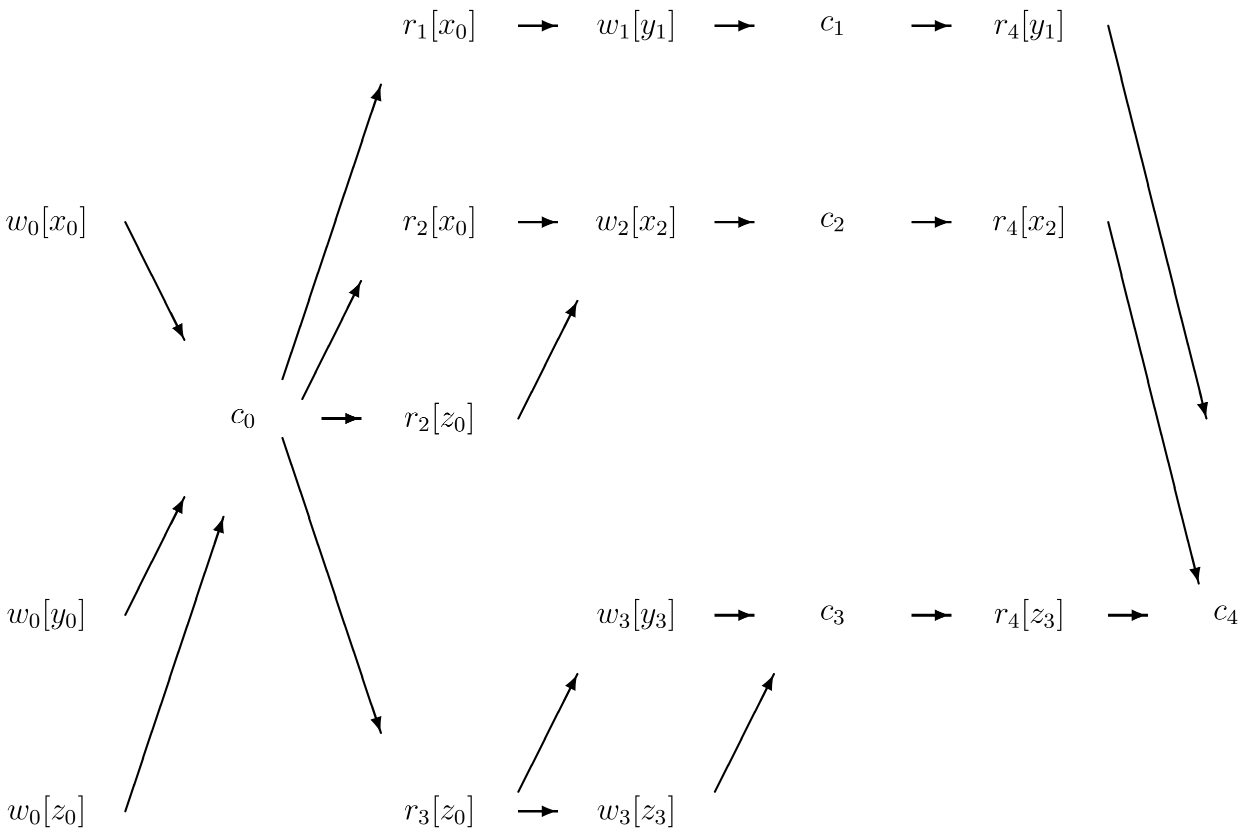
$$T_1 = r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_4 = \begin{array}{ccc} r_4[x] & \searrow & \\ r_4[y] & \rightarrow & c_4 \\ r_4[z] & \nearrow & \end{array}$$

$$T_2 = \begin{array}{ccccc} r_2[x] & \searrow & & & \\ & & w_2[x] & \rightarrow & c_2 \\ r_2[z] & \nearrow & & & \end{array}$$

H_6 ist eine vollständige MV-Historie über $\{T_0, \ldots, T_4\}$:



H_7 ist eine vollständige MV-Historie über $\{T_0, \ldots, T_4\}$, enthält aber $r_4[y_1]$ anstelle von $r_4[y_3]$:



Bemerkung 7.2.3

- *Sichtenserialisierbarkeit verlangt die gleichen Letzten Schreiben. Da jede Transaktion ein anderes Datenelement schreibt, ist dies immer erfüllt, wenn die Historien die gleichen Transaktionen umfassen.*
- *Da nicht Datenelemente, sondern Versionen gelesen werden, müssen wir die Definition der liest-von-Relation ändern. Wir ersetzen in der Definition Datenelement durch Version.*

Definition 7.2.4 (liest-von)
Sei H eine MV-Historie. T_j liest x von T_i ($T_j \rhd_{H[x]} T_i$) $\prec\succ$

$$T_j \rhd_{H[x_i]} T_i.$$

Definition 7.2.5 (Äquivalenz von MV-Historien)
Zwei MV-Historien über einer Menge T von Transaktionen sind äquivalent ($\equiv_V$, falls sie

1. *die gleichen Operationen umfassen und*
2. *die gleiche liest-von-Beziehung haben.*

Man beachte, dass die Bedingung über das letzte Schreiben entfallen ist. Dies ist nach obiger Bemerkung zulässig.

Da keine Version überschrieben wird, ist jedes Schreiben ein letztes Schreiben. Daher kann diese Bedingung entfallen. Die gleiche liest-von-Beziehung zu haben, heißt im vorliegenden Fall, die gleichen Leseoperationen zu besitzen. In anderen Worten:

> Sei H eine MV-Historie. Eine Transaktion T_j liest x von T_i genau dann, wenn T_j x_i liest, also genau dann, wenn $r_j[x_i] \in H$.

Proposition 7.2.6 *Zwei MV-Historien über einer Menge von Transaktionen T sind äquivalent, falls sie die gleichen Operationen umfassen.*

$\equiv_{MV}$ ist also sehr leicht entscheidbar.

Wir sind nur an 1V-Historien H_{1V} interessiert, die gültige Ein-Versionen-Sichten einer MV-Historie H_{MV} sind. D.h.: Es existiert eine bijektive Übersetzung von H_{1V} nach H_{MV}. Die Bijektion erlaubt es uns nun, die liest-von-Beziehungen von H_{1V} und H_{MV} zu vergleichen. Die Letzten Schreiben spielen keine Rolle, weil alle Letzten Schreiben von H_{1V} Teil des von H_{MV} produzierten Zustandes sind, da H_{MV} alle geschriebenen Versionen aufbewahrt. Wir können also definieren:

Definition 7.2.7
Eine 1V-Historie H_{1V} ist eine gültige Ein-Versions-Sicht *einer MV-Historie H_{MV}, falls*

1. *H_{1V} und H_{MV} über der gleichen Menge von Transaktionen definiert sind und*

2. *$\exists h$, h bijektiv und h bildet die Operationen von H_{MV} auf diejenigen von H_{1V} ab.*

Definition 7.2.8 (Äquivalenz von MV und 1V)
Sei H_{1V} eine gültige Ein-Versions-Sicht der MV-Historie H_{MV}. $H_{MV} \equiv_V H_{1V} \prec\succ$

- *H_{MV} und H_{1V} haben die gleiche liest-von-Beziehung.*

7.2.2 Serialisierbarkeitsgraphen

Definition 7.2.9 (Konflikt) *Sei H eine MV-Historie. Zwei Operationen stehen in Konflikt miteinander ($\not\parallel$), falls sie auf dieselbe Version zugreifen und eine Operation eine Schreiboperation ist.*

Falls $p_i <_H q_j$ und $p_i \not\parallel q_j$, dann gilt:

$$\begin{aligned} p_i &= w_i[x_i] \\ q_j &= r_j[x_i] \end{aligned}$$

für ein Datenelement x. Andere Konflikte sind unmöglich:

- $w_i[x_i] <_H w_j[x_i]$ ist unmöglich, da Versionen nie überschrieben werden.

- $r_j[x_i] <_H w_i[x_i]$ ist unmöglich, da Versionen erst nach dem Erzeugen gelesen werden können.

Alle Konflikte können daher auf die liest-von-Beziehung zurückgeführt werden.

Definition 7.2.10 *Sei H eine MV-Historie. Der Serialisierbarkeitsgraph $SG(H)$ ist wie folgt definiert:*

1. *Die Knoten sind abgeschlossene Transaktionen.*
2. $T_i \to T_j \in SG(H)$ $(i \neq j)$ $\prec\succ$ $T_j \rhd_{H[x]} T_i$ $(\prec\succ$ $r_j[x_i] \in C(H), i \neq j)$.

Es folgt unmittelbar:

Proposition 7.2.11
Seien H und H' zwei MV-Historien. Dann gilt:

$$H \equiv_V H' \quad \succ \quad SG(H) = SG(H').$$

Beispiel:

$$\begin{array}{ccccccc} & & & \nearrow & T_2 & \searrow & \\ SG(H_6) & = & T_0 & \to & T_3 & \to & T_4 \\ & & & \searrow & T_1 & & \end{array}$$

$$\begin{array}{ccccccc} & & & \nearrow & T_2 & \searrow & \\ SG(H_7) & = & T_0 & \to & T_3 & \to & T_4 \\ & & & \searrow & T_1 & \nearrow & \end{array}$$

7.2.3 Ein-Versionen-Serialisierbarkeit

Definition 7.2.12
Eine vollständige MV-Historie ist seriell $\prec\succ$

$$\begin{array}{l} \forall T_i, T_j \in H \\ (\forall p_i \in T_i q_j \in T_j \quad p_i <_H q_j) \vee (\forall p_i \in T_i q_j \in T_j \quad q_j <_H p_i) \end{array}$$

Nicht alle seriellen MV-Historien verhalten sich wie gewöhnliche 1V-Historien. Betrachte

$$\begin{array}{rcl} H_3 & = & w_0[x_0]w_0[y_0]c_0 \\ & & r_1[x_0]r_1[y_0]w_1[x_1]w_1[y_1]c_1 \\ & & r_2[x_0]r_2[y_1]c_2 \end{array}$$

Definition 7.2.13 (1-seriell)
Eine serielle MV-Historie H ist 1-seriell $\prec\succ$ $\forall i, j, x$: *falls* $T_i \rhd_H [x] T_j$, *dann gilt*

- $i = j$ *oder*
- T_j *ist die letzte Transaktion vor* T_i, *die irgendeine Version von* x *schreibt.*

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

H_3 ist nicht 1-seriell, da $T_2 \rhd_{H_3} [x]T_0$ aber $w_0[x_0] <_{H_3} w_1[x_1] <_{H_3} r_2[x_0]$.

Die folgende Historie ist 1-seriell:

$$\begin{aligned} H_8 \quad = \quad & w_0[x_0]w_0[y_0]w_0[z_0]c_0 \\ & r_1[x_0]w_1[y_1]c_1 \\ & r_2[z_0]w_2[z_2]w_2[x_2]c_2 \\ & r_3[z_2]w_3[y_3]w_3[z_3]c_3 \\ & r_4[x_2]r_4[y_3]c_4 \end{aligned}$$

Definition 7.2.14 (1-serialisierbar)
Eine MV-Historie H ist 1-serialisierbar (1SR), falls $C(H)$ MV-äquivalent zu einer 1-seriellen Historie ist.

1-Serialisierbarkeit ist eine prefix-commit-abgeschlossene Eigenschaft. Anders als bei Sichtenserialisierbarkeit können wir also auf den Zusatz '*für alle Präfixe*' verzichten.

Eine serielle Historie kann 1SR sein, obwohl sie nicht 1-seriell ist:

$$H_9 = w_0[x_0]c_0r_1[x_0]w_1[x_1]c_1r_2[x_0]c_2$$

H_9 ist nicht 1-seriell, da $T_2 \rhd_{H_9[x]} T_0$ (statt von T_1). H_9 ist 1-serialisierbar, da H_9 äquivalent ist zu

$$H_{10} = w_0[x_0]c_0r_2[x_0]c_2r_1[x_0]w_1[x_1]c_1$$

Zur Bestätigung des Korrektheitskriteriums 1-serialisierbar zeigen wir, dass die abgeschlossene Projektion jeder 1-serialisierbaren Historie äquivalent ist zu einer seriellen 1V-Historie.

Satz 7.2.15
Sei H eine MV-Historie über T. Dann gilt:

$$C(H) \text{ ist äquivalent zu einer seriellen 1V-Historie über } T \prec\!\succ H \text{ ist 1SR}$$

Beweis:
$\prec$: Annahme: H ist 1SR
$\succ$ $\exists$ 1-serielle MV-Historie H_s $\quad C(H) \equiv_{MV} H_s$
Wir übersetzten H_s durch h in eine serielle 1V-Historie H'_s durch Eliminieren der Versionsindizes auf den Datenelementen. Noch zu zeigen: $H_s \equiv_{MV} H'_s$.

Betrachte $T_j \rhd_{H_s} [x]T_i$.

Da H_s 1-seriell, gibt es kein $w_k[x_k]$ mit $w_i[x_i] <_{H_s} w_k[x_k] <_{H_s} r_j[x_j]$.
$\succ$ gibt es auch kein solches in H'_s.
$\succ$ $T_j \rhd_{H'_s} [x]T_i$.

Betrachte $T_j \rhd_{H'_s} [x]T_i$.

Falls $r_j[x_i]$ das Urbild unter h von $r_j[x]$ ist, dann gilt $T_j \rhd_{H_s} [x]T_i$. Annahme: $r_j[x_k], k \neq i$ ist das Urbild von $r_j[x]$. Falls $i = j$, dann gilt $k = i$ (Bed. 4 einer MV-Historie) und wir sind fertig.

Falls $i \neq j$, dann gilt entweder $w_i[x_i] <_{H_s} w_k[x_k]$ oder $r_j[x_k] <_{H_s} w_i[x_i]$, da H_s 1-seriell. Dann: Die Übersetzung dieser Operationen impliziert, dass nicht $T_j \rhd_{H_s'} [x]T_i$ gilt. Widerspruch. Also $T_j \rhd_{H_s} [x]T_i$. Also $H_s' \equiv_{MV} H_s$. Da $H_s \equiv_{MV} C(H)$, $C(H) \equiv_{MV} H_s'$.

$\succ$: Annahme: Sei H_s' eine zu $C(H)$ MV-äquivalente 1V-Historie. Übersetze H_s' in eine serielle MV-Historie: $c_i \rightsquigarrow c_i$, $w_i[x] \rightsquigarrow w_i[x_i]$, $r_j[x] \rightsquigarrow r_j[x_i]$, wobei i so gewählt ist, dass $T_j \rhd_{H_s'} [x]T_i$.

Zu zeigen:

a H_s ist vollständige MV-Historie

b $C(H) \equiv_{MV} H_s$

c H_s ist 1-seriell.

ad a: Bed. 1+2 sind klar.

Für Bed. 3 genügt zu zeigen: Für $r_j[x] \in H_s'$ existiert ein $w_i[x] \in H_s'$ mit $w_i[x] <_{H_s'} r_j[x]$. Da H eine MV-Historie ist, gibt es vor jedem $r_j[x_k]$ ein $w_k[x_k]$ in H. Da H und H_s' die gleiche liest-von-Beziehung haben, gibt es auch ein entsprechendes Schreiben in H_s'.

Bed. 4: Falls $w_j[x] <_{H_s'} r_j[x]$, dann gilt $T_j \rhd_{H_s'} [x]T_j$ und $r_j[x]$ wird in $r_j[x_j]$ in H übersetzt, da H_s' seriell ist.

Bed. 5: zu zeigen: $i \neq j$, $r_j[x_i] \in H_s \succ c_i <_{H_s} c_j$.

$r_j[x_i] \in H_s \succ T_j \rhd_{H_s'} [x]T_i$. Da H_s' seriell gilt $c_i <_{H_s'} c_j$. Die Übersetzung ergibt dann $c_i <_{H_s} c_j$.

Also ist H_s eine MV-Historie.

ad b: Da die Übersetzung die liest-von-Beziehung bewahrt, gilt $H_s \equiv_{MV} H_{s'}$. Transitivität von $\equiv_{MV}$ ergibt $C(H) \equiv_{MV} H_s$.

ad c: Betrachte $T_j \rhd_{H_s} [x]T_i$ für $i \neq j$. Da H_s' eine 1V-Historie ist, liegt kein $w_k[x]$ zwischen $w_i[x]$ und $r_i[x]$ in H_s'. Also liegt auch kein $w_k[x_k]$ zwischen $w_i[x_i]$ und $r_j[x_i]$ in H_s. Also ist H_s 1-seriell.

□

7.2.4 Das 1-SR-Theorem

Um zu garantieren, dass ein MV-Protokoll korrekt ist, müssen wir zeigen, dass alle erzeugten Historien 1SR sind. Hierzu verwenden wir einen modifizierten SG. Die Modifikation ist motiviert durch die Tatsache, dass alle MV-Scheduler die Versionen eines Datenelementes x total ordnen.

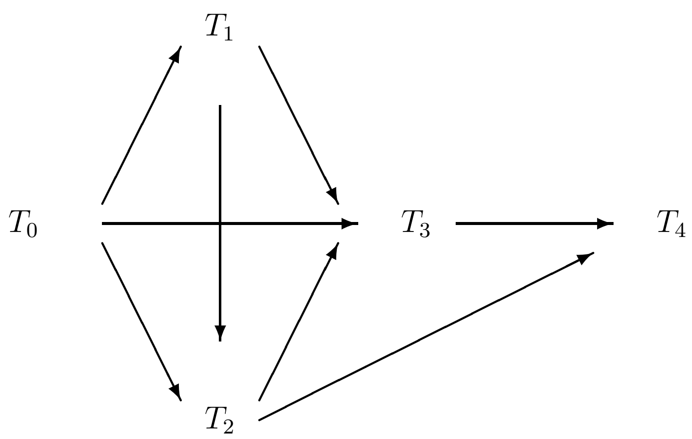
Definition 7.2.16 (Versionsordnung)
Sei H eine MV-Historie und x ein Datenelement. Eine Versionsordnung $\ll$ *für x ist eine totale Ordnung auf den Versionen von x in H. Eine* Versionsordnung *für H ist die Vereinigung der Versionsordnungen aller in H vorkommenden Datenelemente x.*

Beispiel: Eine mögliche Versionsordnung für H_6 ist $x_0 \ll x_2$, $y_0 \ll y_1 \ll y_3$, $z_0 \ll z_3$.

Definition 7.2.17 (MVSG)
Sei MV eine Historie und $\ll$ eine Versionsordnung von H. Der Mehrverisonenserialisierbarkeitsgraph *für H und $\ll$ ($MVSG(H, \ll)$) ist definiert als $SG(H)$, wobei folgende* Versionsordnungskanten *hinzugefügt werden:*

> *Für jedes $r_k[x_j]$ und $w_i[x_i]$ in H, mit $i \neq j \neq k$: falls $x_i \ll x_j$, so wird $T_i \to T_j$ hinzugefügt, sonst $T_k \to T_i$.*

Beispiel 7.2.1 *Für H_6 zusammen mit der obigen Versionsordnung ist $MSVG(H_6, \ll)$:*



Zusätzlich zu $SG(H_6)$: $T_1 \to T_2$, $T_1 \to T_3$ und $T_2 \to T_3$. Außer $T_0 \to T_1$, sind alle anderen Kanten auch Versionsordnungskanten.

Satz 7.2.18 (1-Serialisierbarkeitstheorem)
Eine MV-Historie H ist 1SR $\prec\succ$ es existiert eine Versionsordnung $\ll$ mit $MVSG(H, \ll)$ ist azyklisch.

Beweis:
$\prec$:
Sei H_s eine serielle MV-Historie $T_{i_1}, \ldots, T_{i_n}$,
wobei $T_{i_1}, \ldots, T_{i_n}$ eine topologische Ordnung von $MVSG(H, \ll)$ ist. Da $C(H)$ eine MV-Historie ist, ist H_s auch eine MV-Historie. Da H_s die gleichen Operationen hat wie $C(H)$, gilt $H_s \equiv_{1V} C(H)$.

Noch zu zeigen: H_s ist 1-seriell. Betrachte ein Lesen $T_k \rhd_{H_s} [x_j] T_j$ mit $i \neq j$. Sei $w_i[x_i]$ ($i \neq j$, $i \neq k$) irgendein anderes Schreiben von x.

Falls $x_i \ll x_j$, enthält $MVSG(H, \ll)$ $T_i \to T_j$, also folgt $T_i <_{H_s} T_j$.

Falls $x_j \ll x_i$, enthält $MVSG(H, \ll)$ $T_k \to T_i$, also folgt $T_k <_{H_s} T_i$.

Also schreibt keine Transaktion x zwischen T_j und T_k in H_s.

$\succ$: Für H und $\ll$ sei $MV(H, \ll)$ der Graph, der nur Versionsordnungskanten enthält. Diese hängen nur von den Operationen, nicht aber von deren Ordnung in H ab. Falls also

H und H' die gleichen Operationen beinhalten, so gilt $MV(H, \ll) = MV(H', \ll)$ für alle $\ll$.

Sei H_s eine 1-serielle MV-Historie, $H_s \equiv_{1V} C(H)$. Alle Kanten in $SG(H_s)$ laufen "von links nach rechts": $T_i \rightarrow_{H_s} T_j \succ T_i <_{H_s} T_j$. Definiere $\ll$ wie folgt: $x_i \ll x_j$ falls $T_i <_{H_s} T_j$. Alle Kanten in $MV(H_s, \ll)$ laufen dann auch von links nach rechts. Also auch alle Kanten in $MVSG(H_s, \ll) = SG(H_s) \cup MV(H_s, \ll)$. Also ist $MVSG(H_s, \ll)$ azyklisch.

Prop. 7.2.6 $\succ$: $C(H)$ und H_s haben die gleichen Operationen. Also $MV(C(H), \ll) = MV(H_s, \ll)$.

Prop.7.2.6,7.2.11 $\succ$: $SG(C(H)) = SG(H_s)$.
Also $MVSG(C(H), \ll) = MVSG(H_s, \ll)$.
Da $MVSG(H_s, \ll)$ azyklisch ist es auch
$MVSG(C(H), \ll)$.
Mit $MVSG(C(H), \ll) = MVSG(H, \ll)$ folgt die Behauptung.

□

7.3 Mehrversionen-Protokolle

Für jedes der drei Basisprotokolle 2PL, TO und SGT, können wir nun MV-Protokolle definieren. Wir fangen mit TO an, da dies am einfachsten ist.

Für MVTO: Jede Transaktion T hat einen eindeutigen Zeitstempel $ts(T)$. Jede Operation hat den Zeitstempel ihrer Transaktion. Jede Version ist durch den Zeitstempel der schreibenden Transaktion markiert.

7.3.1 Mehrversionen TO

Ein Mehrversionen-TO-Scheduler (MVTO) bearbeitet Operationen in FCFS-Ordnung. Er übersetzt Operationen auf Datenelementen in Operationen auf Versionen, so dass es aussieht, als würden die Operationen in Zeitstempelordnung auf einer einzigen Version ausgeführt.

$r_i[x]$ wird in $r_i[x_k]$ übersetzt, wobei x_k die Version mit dem größten Zeitstempel kleiner oder gleich $ts(T_i)$ ist. $r_i[x]$ wird dann zum DM gesendet.

Für $w_i[x]$ werden zwei Fälle unterschieden:

1. Falls bereits ein $r_j[x_k]$ mit $ts(T_k) < ts(T_i) < ts(T_j)$ ausgeführt wurde, so wird $w_i[x]$ zurückgewiesen.

2. Sonst wird $w_i[x]$ in $w_i[x_i]$ übersetzt und zum DM gesendet.

Für die Wiederanlaufbarkeit wird c_i solange verzögert, bis c_j für alle T_j mit $T_i \rhd T_j$ ausgeführt wurde.

Man vergleiche die Ausführungen von MVTO mit denen einer 1-Version-Historie H_{1V} in Zeitstempelordnung. Dort liest $r_i[x]$ auf den Wert von x mit dem größten Zeitstempel kleiner als $ts(T_i)$. Dies geschieht auch bei MVTO.

Bemerkung 7.3.1 *Sei $ts(T_i) = i$. Sei $w_0[x_0] < r_2[x_0]$ eine MV-Historie. Es erreiche nun $w_1[x]$ den Scheduler. Übersetzung nach $w_1[x_1]$ ergibt ein Problem, da in $w_0[x_0] < r_2[x_0] < w_1[x_1]$ $r_2[x]$ den Wert von x_0 liest, es aber den von x_1 lesen sollte.* $w_1[x_1]$ invalidiert $r_2[x_0]$. *Also muss $w_1[x]$ zurückgewiesen werden. Allgemein: $w_i[x]$ wird zurückgewiesen, falls bereits ein $r_j[x_k]$ ausgeführt wurde mit $ts(T_k) < ts(T_i) < ts(T_j)$.*

MVTO Versionsauswahl Zur Auswahl der zu lesenden Versionen und um Invalidierungen zu vermeiden, verwaltet der Scheduler Zeitstempelintervalle. Für jede Version x_i wird ein Zeitstempelintervall $interval(x_i) = [wts, rts]$ geführt, mit

- wts ist der Zeitstempel von x_i und
- rts ist der größte Zeitstempel eines Lesens von x_i. Falls kein solches Lesen existiert, so gilt $wts = rts$.

Sei $intervals(x) = \{\ interval(x_i) \mid x_i \textit{ ist eine Version von } x\ \}$.

Zur Bearbeitung von $r_i[x]$ untersucht der Scheduler $intervals(x)$, um diejenige Version x_j zu finden, deren Intervall $[wts, rts]$ das größte wts kleiner oder gleich $ts(T_i)$ hat. Falls $rts < ts(T_i)$, so wird rts auf $ts(T_i)$ gesetzt.

Zur Bearbeitung von $w_i[x]$ untersucht der Scheduler $intervals(x)$, um diejenige Version x_j zu finden, deren Intervall $[wts, rts]$ das größte wts kleiner als $ts(T_i)$ hat. Falls $rts > ts(T_i)$, wird $w_i[x]$ zurückgewiesen, sonst wird $w_i[x_i]$ an den DM gesendet und ein neues Intervall $interval(x_i) = [ts(T_i), ts(T_i)]$ erzeugt.

MVTO Garbage Collection Alte Versionen und deren Intervalle müssen vor den neuen gelöscht werden. Warum? Betrachte H_{11}:

$$w_0[x_0]c_0r_2[x_0]w_2[x_2]c_2r_4[x_2]w_4[x_4]$$

mit $ts(T_i) = i$. Annahme: x_2 wird gelöscht, aber nicht x_0. $r_3[x]$ erreiche den Scheduler. Der übersetzt es fälschlicherweise in $r_3[x_0]$.

Annahme: x_0 wird gelöscht. $r_1[x]$ erreiche den Scheduler. Dieser findet keine Version mit $wts < ts(T_1)$. Diese Bedingung zeigt an, dass die zu lesende Version bereits gelöscht wurde. $r_1[x]$ wird zurückgewiesen.

MVTO Korrektheit Zunächst die formalen Eigenschaften einer MVTO-Historie über $\{T_0, \ldots, T_n\}$.

MVTO1: $ts(T_i) = ts(T_j) \prec\succ i = j$

MVTO2: $\forall r_k[x_j] \in H :$
$w_j[x_j] <_H r_k[x_j] \wedge ts(T_j) \leq ts(T_k)$

MVTO3: $\forall r_k[x_j], w_i[x_i] \in H, i \neq j :$

(a) $ts(T_i) < ts(T_j)$ XOR

(b) $ts(T_k) < ts(T_i)$ XOR

(c) $i = k \wedge r_k[x_j] <_H w_i[x_i]$

MVTO4: $r_j[x_i] \in H, i \neq j, c_j \in H \succ c_i <_H c_j$.

Obige Bedingungen erhalten die reflexiven liest-von-Relationen. Nehme das Gegenteil an: $w_k[x_k] < r_k[x_j]$, $j \neq k$.

MVTO2 $\succ ts(T_j) \leq ts(T_k)$.
MVTO3 $\succ$

(a) $ts(T_k) < ts(T_j)$ XOR

(b) $ts(T_k) < ts(T_k)$ XOR

(c) $i = k \wedge r_k[x_j] <_H w_k[x_k]$

Alle Fälle sind unmöglich. Widerspruch.

Satz 7.3.2
Jede MVTO-Historie ist 1SR.

Beweis: Definiere eine Versionsordnung $\ll$ wie folgt: $x_i \ll x_j \prec\succ ts(T_i) < ts(T_j)$. Wir zeigen nun, dass $MVSG(H, \ll)$ azyklisch ist. Hierfür zeigen wir, dass $\forall T_i \to T_j \in MVSG(H, \ll) : ts(T_i) < ts(T_j)$.

Sei $T_i \to T_j \in SG(H)$. $\succ T_j \rhd_H [x]T_i$ für ein x. MVTO2 $\succ ts(T_i) \leq ts(T_j)$. MVTO1 $\succ ts(T_i) \neq ts(T_j)$. Also $ts(T_i) < ts(T_j)$. $\surd$

Seien $r_k[x_j], w_i[x_i] \in H$. i, j, k ungleich. Betrachte die Versionsordnungskante, die hiervon erzeugt wird. Es sind zwei Fälle zu unterscheiden:

1. $x_i \ll x_j$
   $\succ T_i \to T_j \in MVSG(H, \ll)$
   Def. von $\ll$: $ts(T_i) < ts(T_j)$ $\surd$

2. $x_j \ll x_i$
   $\succ T_k \to T_i \in MVSG(H, \ll)$

ad 2: MVTO3 $\succ$:

- $ts(T_i) < ts(T_j)$ XOR
- $ts(T_k) < ts(T_i)$

Ersteres ist unmöglich, da $x_j \ll x_i \succ ts(T_j) < ts(T_i)$. Also: $ts(T_k) < ts(T_i)$. $\surd$.

Da alle Kanten in $MVSG(H, \ll)$ in Zeitstempelordnung sind, ist $MVSG(H, \ll)$ azyklisch. Nach Satz 7.2.18 ist H 1SR.

□

7.3.2 2-Versionen 2-PL

Vorbemerkungen:

2PL: Eine X-Sperre schließt alle Leseoperationen aus. Dies kann durch die Verwendung von zwei Versionen vermieden werden. Falls T_i x schreibt, so wird eine neue Version x_i angelegt und x gesperrt. Dies verhindert das Lesen von x_i und das Erzeugen weiterer Versionen von x.

Aber: Andere Transaktionen können weiterhin das alte x lesen.

Bei einem commit von T_i wird x_i die aktuelle Version, und auf die alte Version von x kann nicht mehr zugegriffen werden.

Wir verwenden also 2PL für die ww-Synchronisation und Versionenauswahl für rw-Synchronisation.

Kopplung mit Recovery: alte Version = before image, neue Version = after image.

2VPL benutzt 3 Sorten von Sperren: Lese-, Schreib- und Certifysperren. Kompatibilitätsmatrix:

	read	write	ceritify
read	y	y	n
write	y	n	n
certify	n	n	n

Sperrvergabe: wie bisher für rl und wl, durch Kompatibilitätsmatrix gesteuert. Bei commit: alle w-Sperren werden Certifysperren (cl).

- $w_i[x]$: Anfordern $wl_i[x]$. Verzögern, falls Vergabe nicht möglich. Sonst: $wl_i[x]$ und Übersetzung in $w_i[x_i]$.
- $r_i[x]$: Anfordern $rl_i[x]$. Verzögern, falls Vergabe nicht möglich. Falls T_i hält $wl_i[x]$: Übersetzung in $r_i[x_i]$. Sonst: Übersetzung in $r_j[x_j]$ x_j neueste (einzige) freigegebene Version von x.
- c_i: Alle Schreibsperren in Certifysperren umwandeln. (Bedingt Warten, bis alle Leser beendet.) Keine Sperrfreigabe, bevor nicht alle Schreibsperren in Certifysperren umgewandelt wurden.

Da nur freigegebene Versionen gelesen werden: Scheduler vermeidet kaskadierendes Rücksetzen.

Durch Sperrkonversion (-verschärfung) können natürlich wieder Deadlocks auftreten.

$$rl_i[x]wl_j[x]$$

Konversion von $rl_i[x]$ in $wl_i[x]$ und $wl_j[x]$ in $cl_j[x]$ resultiert in Deadlock.

Certifysperren verhalten sich wie Schreibsperren in 2PL, da aber die Zeit, in der sie gehalten werden, viel kleiner ist, ist 2VPL günstiger. Ausserdem: concurrent reads. Nachteil: Warten auf Beenden der concurrent reads bei commit der Schreibtransaktion.

Mehrversionen 2PL Der einzige Grund für die Schreibsperren war es, nur eine einzige neue Version zuzulassen. Wenn man also die Konfliktmatrix ändert, so dass zwei Schreibsperren verträglich werden, so können mehrere Versionen geschrieben werden. Die übrigen Regeln von 2VPL bleiben gleich. Insbesondere kann nur die letzte freigegebene Version gelesen werden. Die Änderung beeinflusst nicht die Korrektheit von 2VPL.

Mehr Flexibilität kann man erreichen, wenn man kaskadierendes Rücksetzen erlaubt. Der Scheduler wird wie folgt geändert:

1. Eine Transaktion kann nicht zertifiziert werden, solange nicht alle gelesenen Datenelemente zertifiziert wurden.

2. Eine Schreibsperre für x kann nur in eine Certifysperre umgewandelt werden, wenn keine Lesesperren auf zertifizierten Versionen von x sind.

2VPL (Korrektheit) Wir bezeichnen mit f_i die Zertifizierung von T_i. Historien enthalten diese Operationen. Sei H eine 2VPL Historie über $T_0, \ldots, T_n$. Dann erfüllt H folgende Eigenschaften:

2VPL1: $\forall T_i : r_i[x], w_i[x] <_H f_i \wedge f_i <_H c_i$

2VPL2: $\forall r_k[x_j] \in H : (k \neq j \succ c_j <_H r_k[x_j]) \wedge (k = j \succ w_k[x_k] <_H r_k[x_k]$

2VPL3: $\forall w_k[x_k], r_k[x_j] \in H : w_k[x_k] <_H r_k[x_j] \succ j = k$

2VPL4: $\forall r_k[x_j], w_i[x_i] \in H : f_i < r_k[x_j] \vee r_k[x_j] < f_i$

2VPL5: $\forall r_k[x_j], w_i[x_i] \in H, i \neq j \neq k :$
$f_i <_H r_k[x_j] \succ f_i <_H f_j$

2VPL6: $\forall r_k[x_j], w_i[x_i] \in H, i \neq j :$
$r_k[x_j] < f_i \succ f_k < f_i$

2VPL7: $\forall w_i[x_i], w_j[x_j] \in H : f_i <_H f_j \vee f_j <_H f_i$

Anmerkungen:

ad 2VPL4: Die Zertifizierungen aller x schreibenden Operationen mit den x lesenden Operationen müssen geordnet sein.

ad 2VPL5: Zusammen mit 2VPL2 ergibt sich, dass jedes Lesen $r_k[x_j]$ entweder eine Version x_k oder die letzte zertifizierte Version von x liest.

ad 2VPL6: Eine x schreibende Transaktion T_i kann nicht zertifiziert werden, solange nicht alle Transaktionen, die x vorher lasen zertifiziert wurden.

ad 2VPL7: Zertifizierungen von Transaktionen, die beide ein x lesen, sind atomar.

Satz 7.3.3
Jede 2VPL-Historie ist 1SR.

Beweis: 2VPL1-3 $\succ$

- H bewahrt reflexive liest-von-Beziehungen
- H ist wiederanlaufbar

Also: H ist eine MV-Historie.

Definiere eine Versionsordnung $\ll$ mit $x_i \ll x_j \prec\succ f_i <_H f_j$. 2VPL7 $\succ \ll$ ist eine Versionsordnung.

Wir werden beweisen: $T_i \to T_j \in MVSG(H, \ll) \succ f_i <_H f_j$. Da $f_i <_H f_j$ eine Teilordnung von H ist, H per definitionem azyklisch, gilt: $MVSG(H, \ll)$ ist azyklisch. Mit Satz 7.2.18 folgt dann die Behauptung.

Sei

$$\begin{array}{rl} T_i \to T_j \in SG(H) & \\ \succ & \exists x : T_j \rhd_H [x]T_i \\ \succ [2VPL2] & f_i <_H r_j[x_i] \\ \succ [2VPL1] & r_j[x_i] <_H f_j \\ \succ & f_i <_H f_j \end{array}$$

Betrachte eine Versionsordnungskante, die durch $w_i[x_i]$, $w_j[x_j]$ und $r_k[x_j]$ $(i \neq j \neq k)$ induziert wurde. Zwei Fälle:

1. $x_i \ll x_j$: $\succ T_i \to T_j \succ [def \ll]\ f_i <_H f_j$
2. $x_j \ll x_i$

Im zweiten Fall ist die Versionsordnungskante $T_k \to T_i$.

$$\begin{array}{rcl} x_j \ll x_i & \succ & f_j <_H f_i \\ 2VPL4 & \succ & f_i <_H r_k[x_j] \vee r_k[x_j] <_H f_i \end{array}$$

Im ersten Fall impliziert 2VPL5 $f_i <_H f_j$ (Widerspruch). Also $r_k[x_j] <_H f_i$. 2VPL6 $\succ$ $f_k <_H f_i$ $\surd$.

Bemerkung 7.3.4

- *Multi Version Mixed Method: siehe Bernstein*
  Keine Behinderungen zwischen Anfragen und Änderern.
- *Timestampvergabe kann Problem sein. (teuer).*
  Ersetzen durch commit-Listen (s. Bernstein).
  verteilte commit-Listen: (s. Bernstein)
- *MVSGT: Ähnlich zu SGT, nur jetzt MVSG benutzen. Siehe Vossen.*

Literaturverzeichnis

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] P. Gray and A. Reuter. *Transaction Processing: Concepts and Technology.* Morgan Kaufmann Publishers, San Mateo, Ca, 1993.

[3] C. H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.