Semantikgestützte Concurrency-Control-Verfahren

Udo Kelter

01.12.2001

Zusammenfassung dieses Lehrmoduls

Die gängigen Concurrency-Control-Verfahren basieren auf einem "syntaktischen" Konfliktbegriff: Es werden nur lesende und schreibende Operationen unterschieden. In manchen Fällen liegt es nahe, semantische Eigenschaften, insb. Kommutativitätseigenschaften der Operationen auszunutzen und zu einem semantischen Konfliktbegriff überzugehen. Typischerweise handelt es sich um numerische Datenfelder, auf denen Beträge addiert und subtrahiert werden. Es zeigt sich allerdings, daß eine Reihe nicht offensichtlicher Probleme gelöst werden müssen, die durch Rollback, Bereichsgrenzen und "Ausnahmen" von der Kommutativität entstehen und die teilweise sehr aufwendige Gegenmaßnahmen erforderlich machen.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Sperrverfahren

empfohlen:
- Recovery
- Concurrency-Control-Theorie

Stoffumfang in Vorlesungsdoppelstunden: 1.2

Inhaltsverzeichnis

1 Einführung

Die gängigen Concurrency-Control- (CC-) Verfahren basieren auf einem "syntaktischen" Begriff Konflikt: Es wird nur unterschieden, ob Objekte geschrieben oder nur gelesen werden, von weitergehenden Details der Zugriffsoperationen wird abstrahiert. Bei den Sperrverfahren (s. [SPV]) äußert sich dies darin, daß nur Lese- und Schreibsperren unterschieden werden. Auch theoretische Analysen des Serialisierbarkeitsproblems führen zu der Erkenntnis, daß man i.a. Korrektheitsbegriffe auf Basis des syntaktischen Konfliktbegriffs verwenden muß (s. [CCT]), konkret muß ein CC-Verfahren wenigstens die cp-Serialisierbarkeit realisieren. Ein Ablauf ist cp-serialisierbar, wenn beim Serialisieren keine in Konflikt stehenden Ereignisse vertauscht werden. Würde man in Konflikt stehende Ereignisse vertauschen, würde die Sicht der betroffenen Transaktionen oder der Endzustand der betroffenen Objekte verändert werden.

Manchmal kennt man die Semantik von Aktionen recht genau und möchte diese Kenntnisse dahingehend ausnutzen, die Parallelität der Transaktionsausführungen zu erhöhen. Die Anwendungsbeispiele kommen nicht nur aus klassischen Datenbankanwendungen, sondern auch aus verteilten Betriebssystemen bzw. allgemeiner verteilten, parallel benutzbaren Objektsystemen, wo z.B. Datentypen wie Listen oder Mengen auftreten.

In vielen Fällen scheint die Ausnutzung semantischer Eigenschaften (zumindest auf den ersten Blick) relativ einfach möglich zu sein. Ein Beispiel: Wir nehmen an, zwei Umbuchungen seien wie in Bild 1 angegeben verzahnt. Wir benutzen eine tabellenförmige Notation für Abläufe; X und Y seien lokalisierte Objekte, "X:=X+a" steht als Abkürzung für "u:=X; u:=u+a; X:=u", worin u eine lokale Variable sei.

Bei diesem Ablauf liest T2 einen ungesicherten Wert von T1, beide Transaktionen erhalten eine inkonsistente Sicht der Datenbank, und dennoch wird man ihn als "korrekt" empfinden, denn die Umbuchungen werden korrekt durchgeführt. Im nächsten Abschnitt wollen wir zunächst klären, in welchem Sinne dieser Ablauf korrekt ist, um dann im übernächsten Abschnitt konkrete Sperrverfahren zu entwickeln.

T1	T2	Werte von X	Werte von Y
		1000	500
X:=X-500		500	
	Y:=Y-300		200
Y:=Y+500			700
	X:=X+300	800	

Abbildung 1: Verzahnte Ausführung zweier Umbuchungen

2 Modifikationen

2.1 Definition

Daß man den oben gezeigten Ablauf als "korrekt" empfindet, liegt u.a. an der Art, wie die Objekte gelesen und zurückgeschrieben werden. Wesentlich sind folgende Merkmale:

1. Die Veränderungen an den Datenbank-Objekten, also das Lesen des alten Inhalts, Berechnen und zurückschreiben des neuen Inhalts ("X:=X+a") sind atomar, d.h. das veränderte Datenbank-Objekt wird zwischenzeitlich nicht von anderen Transaktionen gelesen oder verändert. Wir nennen eine solche Veränderung eine **Modifikation**.

   Wir fassen Modifikationen als eine dritte Art von Aktionen auf (neben Lese- und Schreibaktionen).

   Im Gegensatz zu konventionellen Aktionen sind Modifikationen ggf. *keine* Elementaroperation des jeweiligen Datenbankmodells, d.h. sie sind applikationsspezifisch und oberhalb des APIs der Datenbank individuell zu programmieren, genauso wie Transaktionen. Man kann sie daher auch "**Subtransaktionen**" nennen.
2. Gewisse Integritätstests müssen innerhalb der Modifikationen durchgeführt werden. In unserem obigen Beispiel könnte es einen unteren Grenzwert für den Kontostand geben. Da der Wert des Objekts bei der Rückkehr zur aufrufenden Transaktion schon verändert ist, kommt ein anschließender Test innerhalb der Transaktion schon zu

spät. Das Ergebnis von Integritätstests wird durch einen Fehlercode an die aufrufende Transaktion mitgeteilt.

3. Die innerhalb einer Modifikation gelesenen Werte zählen *nicht* zur Sicht einer Transaktion. Typischerweise, so auch im obigen Beispiel, werden sie nur innerhalb dieser Modifikation benutzt, also nicht direkt oder in transformierter Form an den Benutzer weitergegeben oder in ein anderes Objekt geschrieben. In vielen Fällen liefern Modifikationen dennoch einen Wert an die aufrufende Transaktion ab, z.B. einen Fehlercode. Dieser zählt (neben den Werten, die bei normalen Lese-Aktionen gelesen werden) zur Sicht der Transaktion.

2.2 Semantische Konfliktfreiheit von Modifikationen

Das Modifikationskonzept allein erklärt noch nicht, warum der obige Ablauf "korrekt" sein soll. Er ist nicht cp-serialisierbar, sogar nicht einmal Sicht-serialisierbar (vgl. 5.1). Diese Korrektheitsbegriffe sind hier unnötig streng[1], da sie die vollständigen Sichten berücksichtigen, während tatsächlich nur die reduzierten Sichten relevant sind.

Sofern man die Semantik der Rechenoperationen außer Betracht läßt, ist der obige Ablauf auch nicht Endzustands-serialisierbar (s. Abschnitt 3.2). Man findet leicht Interpretationen (also Semantiken der Datentypen und Operationen), bei denen der Endzustand von X und Y im obigen Beispiel mit keinem der Endzustände übereinstimmt, die bei den beiden seriellen Abläufen entstehen. Bei der Endzustands-Serialisierbarkeit werden beliebig "ungünstige" Interpretationen berücksichtigt, während wir oben wissen, daß eine "günstige" Interpretation vorliegt.

Günstig ist hier, daß die Modifikationen in einem gewissen Sinn kommutieren. Um diesen Begriff zu präzisieren, fassen wir eine Modifikation, die oben als *Algorithmus* eingeführt wurde, stattdessen als eine *Funktion* auf dem Wertebereich des modifizierten Datenbank-Objektes auf. Der Hintereinanderausführung von Modifikationen entspricht die Komposition der Funktionen.

[1] Wobei wir sie in kanonischer Weise auf das n-Schritt-Modell für Transaktionen erweitern könnten.

So wird im obigen Beispiel zunächst die Modifikation "Erniedrige um 500" auf X ausgeführt, danach "Erhöhe um 300". Die entsprechenden Funktionen sind $x \mapsto x - 500$ und $x \mapsto x + 300$; ihre Komposition ist $x \mapsto x - 200$. Diese Funktion läßt sich sogar direkt durch eine Modifikation erzeugen, nämlich "Erniedrige um 200"; notwendig ist dies aber nicht.

Das entscheidende Merkmal des Konfliktbegriffs ist folgendes: Wenn zwei Ereignisse nicht in Konflikt stehen, darf ihre Reihenfolge vertauscht werden, ohne daß die Vertauschung Einfluß hat auf

1. den Endzustand der betroffenen Objekte
2. die Sichten der beiden beteiligten Transaktionen.

Zwei derartige Ereignisse nennen wir **semantisch konfliktfrei**.

Die bisherige Definition des Konfliktbegriffs war insofern *syntaktisch*, als sie sich nur auf die Identität von Objekten und die Unterscheidung in verändernde und nicht verändernde Zugriffe bezog, also Merkmale, die auf dem syntaktischen Niveau angesiedelt sind. Die syntaktische Konfliktfreiheit ist hinreichend, aber nicht notwendig für die semantische Konfliktfreiheit.

Offensichtlich stehen Modifikationen i.d.R. in Konflikt mit Lese- und Schreibereignissen.

2.3 Sperrmodi für Modifikationen

Wir können semantisch konfliktfreie Aktionen genauso behandeln wie herkömmlich konfliktfreie Aktionen, sowohl bei den Korrektheitsbegriffen für Logs wie auch bei Sperrverfahren.

Wenn eine Transaktion eine Sperre für ein Objekt hält und somit das Recht hat, gewisse Aktionen auszuführen, können wir dies so interpretieren, daß während der Sperrzeiten nur konfliktfreie Aktionen anderer Transaktionen auf diesem Objekt eintreten dürfen. Für Modifikationen gilt dies analog. Im einfachsten Fall gehört zu einer Modifikation M ein eigener Sperrmodus. Dieser ist kompatibel mit solchen Sperrmodi, die nur das Recht zu Ausführung solcher Aktionen implizieren, die mit M nicht in Konflikt stehen.

Kompatibilität|seeVerträglichkeit

Allgemeiner definieren wir zwei Sperrmodi als **kompatibel**, wenn jede der beim ersten Sperrmodus zulässigen Aktionen mit jeder beim zweiten Sperrmodus zulässigen Aktionen (auf dem gleichen Objekt) semantisch konfliktfrei ist. Die bisherige Definition der Verträglichkeit von Lese- und Schreibsperren ist ein Sonderfall dieser allgemeineren Definition[2].

Im obigen Beispiel ergeben sich folgende Verträglichkeiten, wobei die üblichen Modi S und X hinzugenommen wurden:

vorhandene	beantragte Sperre:					
Sperre:	S	X	“+300”	“-300”	“+500”	“-500”
S	+	-	-	-	-	-
X	-	-	-	-	-	-
“+300”	-	-	+	+	+	+
“-300”	-	-	+	+	+	+
“+500”	-	-	+	+	+	+
“-500”	-	-	+	+	+	+

2.4 Undo von Modifikationen

Das Undo einer Aktion wird üblicherweise durch Rückschreiben des vorher vorhandenen Werts realisiert. Da ein Objekt von mehreren gleichzeitig aktiven Transaktionen verändert werden kann, würden beim Rückschreiben des alten Werts anläßlich des Rollbacks einer Transaktion auch zwischenzeitliche Änderungen anderer, eventuell bereits abgeschlossener Transaktionen verloren gehen.

Konventionelle Sperrprotokolle halten die Isolation ein: es werden keine ungesicherten Werte gelesen. Die Isolation muß bei Modifikationen aufgegeben werden, wenn man mit ihnen überhaupt einen Parallelitätsgewinn erzielen will. Hierdurch entsteht wieder das Problem der Rollbackfortpflanzung.

[2] Die Konstruktion von Update- und Warnsperrmodi kann ebenfalls auf beliebige Basis-Sperrmodi verallgemeinert werden, s. [Ko83]).

Zur Lösung dieses Problems benutzt man ein anderes (Rückwärts-) Recovery-Prinzip, nämlich Kompensation. Zu jeder Modifikation wird eine **inverse Modifikation** (kurz: **Invertierung**) vorgesehen, welche vom Programmierer der Modifikation zu liefern ist. Hieraus resultiert, daß für Modifikationen spezielle Einträge im Log vorgesehen werden müssen und daß die inversen Modifikationen jederzeit dem Recovery-Manager in einer Bibliothek zur Verfügung stehen müssen.

Sofern ein Objekt innerhalb einer Transaktion mehrfach modifiziert wurde, müssen die Invertierungen in umgekehrter Reihenfolge durchgeführt werden, sofern sie nicht semantisch konfliktfrei miteinander sind.

Eine zurückgesetzte Transaktion wird durch die Invertierungen gedanklich fortgesetzt, bis sie am Ende den Effekt Null hat. Hieraus folgt:

- Sperren für Modifikationen dürfen *nicht* vor Commit freigegeben werden[3]. Damit die entstehenden Logs korrekt sind, müssen die Invertierungen nämlich konfliktfrei in die Verarbeitungsphase der Transaktion verschoben werden können.
- Die Invertierung zu einer Modifikation M muß mit allen Modifikationen konfliktfrei sein, mit denen M konfliktfrei ist bzw. die beim zu M gehörigen Sperrmodus für andere Transaktionen aufgerufen werden dürfen.

Wir nennen daher zwei Modifikationen nur dann **konfliktfrei**, wenn zusätzlich jede Modifikation mit der Invertierung der anderen und die beiden Invertierungen zueinander konfliktfrei sind.

Die Verwendung des Kompensationsprinzips führt zu weiteren Problemen beim Neustart nach Systemfehlern, die wir hier nicht näher diskutieren.

[3]Bei Sperren für konventionelle Aktionen ist dies im Rahmen des 2-Phasen-Protokolls im Prinzip erlaubt, wegen der Fortpflanzung von Rollback ist jedoch generell davon abzuraten.

2.5 Atomarität von Modifikationen

Wir hatten oben vereinbart, Modifikationen als eine dritte Art von Aktionen aufzufassen. Aktionen müssen die gleichen Atomaritätseigenschaften erfüllen wie Transaktionen, nämlich Fehler-Atomarität und Serialisierbarkeit (bzw. funktionale Atomarität).

Ein Unterschied von Modifikationen zu Lese- oder Schreibaktionen ist, daß letztere durch das DBMS implementiert sind, während Modifikationen ggf. vom Anwender zu implementieren sind.

Bei der Realisierung der funktionalen Atomarität ist zu bedenken, daß Modifikationen typischerweise kurz sind und nur ein einziges Objekt betreffen. Durch komplizierte Protokolle ist daher nur wenig zu gewinnen; wir gehen davon aus, daß das modifizierte Objekt wechselseitig ausgeschlossen benutzt wird.

Bezüglich der Fehler-Atomarität gelten im Prinzip die gleichen Überlegungen wie für Transaktionen. Ursachen für den Abbruch einer Modifikation können sein:

- unbeabsichtigte Laufzeitfehler
- programmiertes Rollback nach einem Integritätstest mit negativem Ausgang. Dies kann, muß aber nicht bedeuten, daß auch die zugehörige Transaktion abbricht. Dies wird innerhalb der Transaktion anhand des von der Modifikation zurückgegebenen Fehlercodes entschieden.
- die zugehörige Transaktion wird aus äußeren Gründen zurückgesetzt.

Diese Fehler treten innerhalb der Modifikation auf, d.h. zwischen dem Lesen des Objekts und dem Zurückschreiben des neuen Inhalts. Für das Rollback einer Modifikation können im Prinzip die gleichen Techniken angewandt werden wie für Transaktionen[4].

[4] Für die Invertierung von Modifikationen müssen hingegen spezielle Recovery-Mechanismen vorgesehen werden.

3 Parametrisierte Modifikationen

3.1 Definition

Bei den Modifikationen in den obigen Beispielen wurde stets um einen festen Wert erhöht oder erniedrigt. Programmtechnisch wird man diesen Wert natürlich als Parameter übergeben, d.h. es handelt sich um **parametrisierte Modifikationen**. Diese entsprechen Funktionalen, durch Einsetzen eines zulässigen Parameters ergibt sich eine Modifikation bzw. eine Funktion. Es können auch mehrere Parameter vorhanden sein, so daß man ein zulässiges Tupel von Parametern einsetzen müßte. Die Menge aller zulässigen Parameterbelegungen definiert **die zu einer parametrisierten Modifikation gehörige Menge von Modifikationen**.

Beispiele von parametrisierten Modifikationen, die oben auftraten, sind "Erhöhe um ..." und "Erniedrige um ...", kurz **incr()** und **decr()**.

Diesen Funktionalen entsprechen numerische Operatoren. Deren Kommutativität impliziert sofort, daß Modifikationen bei *beliebigen* Parameterwerten semantisch konfliktfrei sind (Probleme, die bzgl. Kommutativität durch Bereichsüberlauf verursacht werden, behandeln wir später), denn für alle Parameterwerte a, b gilt (* steht für Komposition):

$$\begin{aligned} \text{incr(a)} * \text{incr(b)} &= x \mapsto (x + a) + b \\ &= x \mapsto (x + b) + a = \text{incr(b)} * \text{incr(a)} \end{aligned}$$

"Einfache" Modifikationen fassen wir i.f. als Sonderfall von parametrisierten auf. Die "Menge von zugehörigen Modifikationen" besteht nur aus ihr selbst.

3.2 Vollständige Konfliktfreiheit

Durch eine parametrisierte Modifikation wird i.a. eine so große Menge von zugehörigen Modifikationen definiert, man man nicht mehr sinnvoll nach der oben vorgestellten Methode verfahren kann, jeder Modifikation einen eigenen Sperrmodus zuzuordnen. Wie eben am Beispiel "incr()"

gezeigt wurde, sind sowieso alle zugehörigen Modifikationen paarweise semantisch konfliktfrei, so daß man offensichtlich mit einem einzigen Sperrmodus auskäme. Dies gilt sogar dann noch, wenn man zusätzlich alle Modifikationen hinzunimmt, die zu "decr()" gehören. Diese Gegebenheit läßt sich wie folgt formaler beschreiben:

Sei **PM** eine Menge von parametrisierten (und "einfachen") Modifikationen. Wir bilden die Gesamtmenge aller zugehörigen Modifikationen. Wenn in dieser Gesamtmenge alle Modifikationen paarweise konfliktfrei sind, dann heißt PM **vollständig (semantisch) konfliktfrei**.

Offensichtlich reicht für eine Menge vollständig konfliktfreier parametrisierter Modifikationen ein einziger Sperrmodus aus. Eine Sperre in diesem Modus berechtigt dazu, auf dem jeweiligen Objekt beliebige Modifikationen aus dieser Menge auszuführen. Der Modus ist mit sich selbst verträglich.

Im obigen Beispiel können wir also für "incr()" und "decr()" einen einzigen Sperrmodus "incr" vergeben. Wir fügen noch einen weiteren Sperrmodus "mult" hinzu, der für Multiplikationen oder Divisionen steht. Die Verträglichkeitsmatrix ist:

vorhandene Sperre:	beantragte Sperre: S	X	incr	mult
S	+	-	-	-
X	-	-	-	-
incr	-	-	+	-
mult	-	-	-	+

Eine Sperre im Modus "incr" berechtigt nun dazu, beliebig viele Beträge zu einem Objekt hinzuzuaddieren.

In dem Fall, daß man innerhalb der gleichen Transaktion dieses Objekt außerdem mit einem Faktor multiplizieren will, benötigt man einen Sperrmodus, der die Zugriffsrechte von "incr" und "mult" vereinigt. Ein Verfahren zur Konstruktion derartiger Kombinations-Sperrmodi ist in

[Ko83] für die normalen Sperrmodi vorgestellt worden; es kann im Prinzip auf beliebige Arten von Sperrmodi verallgemeinert werden. Es stellt sich allerdings die Frage, unter welchen Randbedingungen so komplexe Mengen von Sperrmodi und zugehörigen Kompatibilitätstest sowie Höherstufungsregeln noch sinnvoll sind.

4 Bereichsgrenzen

Bei der Feststellung, daß die beiden Modifikationen im obigen Beispiel semantisch konfliktfrei sind, haben wir ein Problem ausgeklammert: nach der ersten Modifikation könnte ein Bereichsüberlauf eintreten.

Hierzu ein Beispiel: das veränderte Objekt stellt ein Konto, einem Lagerbestand o.ä. dar, der Stand darf nicht unter 0 fallen, anfangs sei der Stand 300. Nun wollen zwei Transaktionen jeweils 200 Einheiten abbuchen. Bei der zweiten Abbuchung wird die Bereichsgrenze unterschritten, was durch einen Integritätstest innerhalb der Modifikation festgestellt wird. Die Modifikation wird abgebrochen und zurückgesetzt. Die aufrufende Transaktion erhält einen entsprechenden Fehlercode als Ergebnis. Dieser Fehlercode zählt zur Sicht der Transaktionen; er wird benutzt, um über das weitere Vorgehen in der Transaktion zu entscheiden, z.B., ob die Transaktion ebenfalls abgebrochen werden soll. *Die Sicht der Transaktion* und der Endzustand des modifizierten Objekts sind also durch die Reihenfolgevertauschung *verändert worden*!

In vielen Fällen wird nach einer Bereichsüberschreitung die Transaktion ebenfalls abgebrochen werden; man kann sich dann auf den Standpunkt stellen, daß die Sicht der Transaktion dann ohnehin eine Rolle gespielt hat, das Risiko des Abbruchs der Transaktion bestand auf jeden Fall und der Abbruch war somit ein "korrekter" Ausgang der Transaktion.

4.1 Inkonsistente Zwischenzustände

Das folgende Beispiel zeigt allerdings, daß die logische Atomarität dennoch verloren gehen kann: zwei Transaktionen soll zwei Teilbeträge von zwei Konten abbuchen und die Summe einem dritten Konto

gutschreiben. Transaktion T1 würde die Sequenz X:=X-200; Y:=Y-200; Z:=Z+400 ausführen, T2 die Sequenz Y:=Y-200; X:=X-200; U:=U+400. Innerhalb jeder Modifikation "-200" wird getestet, ob der Wert negativ werden würde; falls ja, wird die Modifikation abgebrochen. Die Transaktion löst daraufhin ein Rollback aus.

Nehmen wir an, die Konten X und Y enthalten beide anfangs 300. Dann würde bei serieller Ausführung genau eine der beiden Transaktionen erfolgreich ausgeführt werden, die andere nicht mehr. Im folgenden Ablauf wird aber keine Transaktion erfolgreich ausgeführt:

T1	T2	Werte von		Fehler
		X	Y	
		300	300	
X:=X-200		100		
	Y:=Y-200		100	
Y:=Y-200			(-100??) 100	Y < 0
	X:=X-200	(-100??) 100		X < 0
Rollback (X:=X+200)		300		
	Rollback (Y:=Y+200)		300	

Die Ursache des Problems im vorigen Beispiel liegt in folgendem: nach den beiden ersten Rechenschritten haben die Daten einen *temporären* Zustand erreicht, den man als inkonsistent bezeichnen kann. Er würde bei einer seriellen Ausführung nicht erreicht und liegt sozusagen "zu nahe" an den Bereichsgrenzen, er manifestiert sich in Form von überflüssigen Fehlermeldungen bzw. Rollbacks. Man kann auf das Problem unterschiedlich reagieren:

1. Man kann solche Abläufe verhindern: Dann sind allerdings komplizierte Algorithmen zu deren Erkennung erforderlich (s.u.), die u.U.

die Absicht, durch Modifikationen die Performance des DBMS zu verbessern, durchkreuzen.

2. Man kann solche Abläufe dulden: Dann verzichtet man auf die vollständige Atomarität, man toleriert die (seltenen) Abweichungen, sofern nur geringe Folgeschäden auftreten. Die Vorstellung ist typischerweise, daß aus Sicht jeder einzelnen Transaktion der erfolglose Ausgang ein im Prinzip denkbares, also korrektes Ergebnis war und daß der Benutzer eine erfolglose Transaktion später und einmal wiederholen kann.

4.2 Unsichere Zwischenzustände und inverse Modifikationen

Bereichsgrenzen werfen zusätzliche Probleme bei den inversen Modifikationen auf, die im Rahmen eines Rollbacks fällig werden. Betrachten wir hierzu folgendes Beispiel: Ein Konto X hat einen Stand von 200 und darf nicht negativ werden; Transaktion T1 erhöht das Konto um 500 Einheiten und führt danach weitere Aktionen auf anderen Objekten aus, die zu einem Rollback führen; Transaktion T2 vermindert das Konto um 400 Einheiten. Unterstellt sei folgender Ablauf:

T1	T2	Werte von X	Fehler
		200	
X:=X+500		700	
	X:=X-400	300	
Rollback (X:=X-500)		-200??	X<0 !!

Die im Rahmen des Rollbacks von T1 erforderliche inverse Modifikation X:=X-500 würde zu einer Überschreitung der Bereichsgrenze führen! Verursacht wird das Problem dadurch, daß T2 auf einem unsicheren Wert gearbeitet hat. T2 hätte an dieser Stelle gar nicht ausgeführt werden dürfen, denn T2 ist im gegebenen Zustand nicht semantisch konfliktfrei mit der inversen Modifikation zu X:=X+500.

Anders gesehen hatte bei der Ausführung von T2 X zwar den Wert 700, und bei diesem Wert war die Bereichsgrenzen noch weit genug entfernt, aber es war ein Rollback von T1 möglich, wonach X=200 gewesen wäre, und in diesem Zustand war T2 nicht mehr erfolgreich ausführbar.

Allgemeiner gesehen stellt sich das Problem folgendermaßen dar: Ein Wert kann unsicher sein, weil mehrere nicht abgeschlossene Transaktionen Modifikationen auf ihm ausgeführt haben. Jede der Transaktionen kann unabhängig von den anderen zurückgesetzt werden; bei n Transaktionen bestehen somit 2^n Möglichkeiten, daß eine Teilmenge der Transaktionen zurückgesetzt wird. Jede Teilmenge entspricht einem bestimmten Wert des Objekts, der bei Rücksetzung dieser Transaktionen entstehen würde. Eine Transaktion darf nur dann ausgeführt (bzw. die entsprechende beantragte Sperre zugeteilt) werden, wenn sie bei *allen* Werten erfolgreich ausgeführt werden kann[5].

Wegen der kombinatorischen Explosion der Zahl der Teilmengen ist es praktisch nahezu ausgeschlossen, ab einem Parallelitätsgrad von ca. 5 alle Werte tatsächlich einzeln zu berechnen. Ein noch halbwegs effizient realisierbares Verfahren ist die nachfolgend beschriebene Überwachung von Unsicherheitsbereichen.

4.3 Überwachung von Unsicherheitsbereichen

Das folgende Verfahren ist nur bei linear geordneten Wertebereichen anwendbar, bei denen sich die Bereichsgrenzen einfach als Intervall (minimaler und maximaler Wert des Objekts) ausdrücken lassen. In der Praxis kommen wohl nur numerische Wertebereiche und Modifikationen wie incr() und decr() infrage, von denen wir i.f. auch ausgehen.

Die Idee des Verfahrens besteht darin, daß man gar nicht alle Teilmengen von Rücksetzungen zu berechnen braucht, sondern sich wegen der linearen Ordnung auf den ungünstigsten Fall beschränken kann. Dieser ungünstigste Fall sieht wie folgt aus:

- bei einer Inkrementierung:

[5] Analog kann man dies auch für nicht erfolgreiche Ausführungen definieren, an diesen ist man i.a. aber nicht interessiert, so daß wir diesen Fall nicht weiter betrachten.

für die obere Grenze: kein Rollback
für die untere Grenze: Rollback
- bei einer Dekrementierung:
für die obere Grenze: Rollback
für die untere Grenze: kein Rollback

Zu einem numerischen Objekt X seien **deltaMin** und **deltaMax** die Differenzen zu den Werten von X, die infolge von Rücksetzungen im ungünstigsten Fall eintreten können. Die beiden Werte stellen den Unsicherheitsbereichen nach oben bzw. unten dar und werden wie folgt berechnet:

- bei einer Inkrementierung incr(a) (a≥0):
deltaMax bleibt unverändert
deltaMin := deltaMin + a
- bei einer Dekrementierung decr(a) (a≥0)::
deltaMax := deltaMax + a
deltaMin bleibt unverändert

Sobald eine Transaktion endet (Commit oder Rollback), werden für alle von dieser Transaktion durchgeführten Modifizierungen die vorstehenden Änderungen von deltaMax bzw. deltaMin wieder rückgängig gemacht.

Seien XMin und XMax die minimal bzw. maximal für X zulässigen Werte. Zugelassen wird incr(a) nur noch dann, wenn

X + deltaMax + a ≤ XMax

ist. Analog dazu wird decr(a) nur noch dann, wenn

X - deltaMin - a ≥ XMin

ist.

5 Konfliktfreiheit mit Parametereinschränkungen

Das folgende Beispiel zeigt zwei parametrisierte Modifikationen, die nicht für alle Parameterwerte semantisch konfliktfreie Modifikationen

ergeben: Wertebereich seien Mengen über einer Basismenge (z.B. set of char), Funktionale sind das Hinzufügen eines Elements (oder einer Menge) zu einer Menge und das Wegnehmen im Sinne der unsymmetrischen Differenz. Unter der Bedingung, daß als Parameter verschiedene Elemente (bzw. disjunkte Mengen) verwendet werden, sind die beiden entstehenden Modifikationen semantisch konfliktfrei, sonst nicht.

Für eine genauere Definition dieses Sachverhalts bilden wir wieder zu einer gegebenen Menge PM von (parametrisierten) Modifikationen die Gesamtmenge der zugehörigen Modifikationen. Wenn sich eine Menge von Paaren von konfliktfreier Modifikationen aus dieser Gesamtmenge durch eine Bedingung an die Parameter, die in den beteiligten Modifikationen gelten, angeben läßt, dann heißt PM **(semantisch) konfliktfrei mit Parametereinschränkungen**.

Im obigen Beispiel bestand PM aus "einfügen(x)" und "ausfügen(y)", die Parametereinschränkung war $x \neq y$.

Die Menge der Paare in der obigen Definition sollte eine sinnvolle Größe haben, also in der gleichen Größenordnung wie das Quadrat der Größe der Gesamtmenge der Modifikationen liegen; letztlich ist diese Bewertung etwas subjektiv und auch von der Häufigkeit des Auftretens einzelner Parameter abhängig. Als Negativbeispiel sei genannt: incr(x) und mult(y) sind mit der Parametereinschränkung x=0 oder y=1 konfliktfrei.

Einheitliche Sperrmodi für die Gesamtmenge von Modifikationen sind nun leider nicht mehr anwendbar. Wir müssen daher zu modifikationsbezogenen Sperrmodi zurückkehren. (In gewissen Fällen kann die Zahl der Sperrmodi aber verringert werden, indem man gleichwertige Modi geschickt zusammenfaßt.) In der Kompatibilitätsmatrix notieren wir anstelle von + oder - die Bedingung, die die Parameter der Modifikationen erfüllen müssen.

vorhandene Sperre:	beantragte Sperre: S	X	insert(a)	remove(b)
S	+	-	-	-
X	-	-	-	-
insert(c)	-	-	+	b≠c
remove(d)	-	-	a≠d	+

Beispielsweise darf eine Sperre in Modus "remove(x)" nicht zugeteilt werden, solange eine Sperre im Modus insert(x) für diese Menge besteht.

Die Realisierung solcher Sperren erfordert flexiblere Sperroperationen als bisher; genauer müssen neben den bisherigen Sperrmodi (S, X, IS, ...) Darstellungen für alle Modifikationen incl. Parameterwerte verarbeitet werden können. Hierzu müssen die Datenstrukturen in einer Sperrtabelle erweitert werden. Die Kompatibilität von Sperren muß durch spezielle Algorithmen festgestellt werden, die vom *Benutzer* (bzw. dem Programmierer der Modifikationen) zu liefern sind. Dieser Algorithmus benötigt als Eingabedaten die Namen der Modifikationen und ggf. deren Parameter. Daher bietet es sich an, diese Angaben direkt in der Sperrtabelle zu speichern. Die Sperroperation hätte dann drei Parameter:

1. Objektidentifikation
2. Identifikation der Modifikation
3. ggf. Parameter der Modifikation

Derartige äußere Eingriffe in die Sperrenverwaltung werden in den meisten Fällen völlig undenkbar sein; selbst wenn sie in einem speziellen Fall zulässig sind, bleibt das Effizienzproblem: Die Sperroperation darf trotz der deutlich höheren Flexibilität nicht wesentlich ineffizienter werden, sonst wird je nach den Umständen insgesamt kein Gewinn an Performance durch diese Sperrmodi erzielt.

6 Konfliktfreiheit mit Objektzustandseinschränkungen

Beim letzten Beispiel hing die semantische Konfliktfreiheit zweier parametrisierter Modifikationen von den Parametern ab; im nächsten Beispiel wird sie statt dessen vom Zustand des modifizierten Objekts abhängen. Der Typ des Objekts sei hier eine (Warte-) Schlange über irgendeinem Basistyp; wir betrachten nur die Operationen:

append(x)
: ein Element mit Inhalt x hinten an die Schlange anhängen

remove():x
: eine Element vorne entnehmen; x ist Rückgabewert. Ist die Schlange leer, wird ein Fehlercode zurückgegeben.

Zwei append-Operationen sind nicht konfliktfrei, da die Reihenfolge der Anfügungen relevant ist. Zwei remove-Operationen sind ebenfalls nicht konfliktfrei, da i.a. verschiedene Werte in den Elementen der Schlange enthalten sind und der gelesene Wert bei remove zur Sicht dieser Operation zählt!

Je eine append- und remove-Operation sind offensichtlich genau dann konfliktfrei, wenn die Schlange nicht leer ist. Hierbei handelt es sich um eine neue Bedingung an die semantische Konfliktfreiheit, die unabhängig von Bedingungen an die Parameter von parametrisierten Modifikationen auftreten kann und die auch für nichtparametrisierte Modifikationen sinnvoll ist. Wir definieren daher:

- Zwei (nichtparametrisierte) Modifikationen heißen **(semantisch) konfliktfrei mit Objektzustandseinschränkung P**, wenn ihre beiden seriellen Ausführungen dieselbe Funktion ergeben, vorausgesetzt der Zustand des modifizierten Objekts erfüllt zu Beginn ein Prädikat P.
- Zwei parametrisierte Modifikationen heißen **(semantisch) konfliktfrei mit Objektzustandseinschränkung P**, wenn alle zugehörigen Paare von Modifikationen konfliktfrei mit Objektzustandseinschränkung P sind.

- Zwei parametrisierte Modifikationen heißen **(semantisch) konfliktfrei mit Parametereinschränkung P1 und Objektzustandseinschränkung P2**, wenn für alle Paare von Modifikationen, die unter Einhaltung der Bedingung P1 an die Parameter abgeleitet werden können, gilt, daß deren beide Hintereinanderausführungen dieselbe Funktion ergeben, vorausgesetzt der Zustand des modifizierten Objekts erfüllt zu Beginn ein Prädikat P2.

Man kann nun analog zum letzten Beispiel eine Verträglichkeitsmatrix, z.B. für append und remove, konstruieren, in der Bedingungen auftreten, die sich auf den Zustand des Objekts beziehen. Die obigen Bemerkungen zu solchen Einträgen gelten hier verstärkt. Hinzu kommen allerdings weitere Probleme mit derartigen Modifikationen und ihren Invertierungen, die es zweifelhaft erscheinen lassen, ob sie wirklich praktisch verwertbar sind:

1. Die erforderlichen inversen Modifikationen, z.B. das Zurückstellen eines Eintrags "vorne" in eine Schlange, können i.d.R. nicht durch schon vorhandene Modifikationen realisiert werden; stattdessen sind zusätzliche Operationen zu realisieren, die ggf. eine völlige Neuspezifikation und Neuimplementierung des Typs erforderlich machen.
2. Es können unsichere Teilobjekte entstehen. So kann eine Transaktion T1 ein Element in die Schlange einfügen, Transaktion T2 will dieses Element entnehmen, bevor T1 beendet ist. Wenn nun T1 zurückgesetzt wird, muß auch T2 zurückgesetzt werden. Wie schon in [TID] erwähnt ist die Fortpflanzung von Rollback äußerst problematisch.

   Anders gesagt ist bei unsicheren Teilobjekten das Ausfügen nicht mit der Invertierung des Einfügens konfliktfrei[6].

[6] Hierbei handelt es sich übrigens um ein allgemeineres Bereichsüberwachungsproblem: Der Wert des Objekts ist durch eine erste Modifikation verändert worden, anschließend durch eine konfliktfreie Modifikation, und hat einen Wertebereich erreicht, in dem die Invertierung der ersten Modifikation nicht mehr anwendbar ist. In unserem Beispiel kann das unsicher eingefügte Element nicht mehr ausgefügt werden, weil es schon von der anderen Transaktion ausgefügt ist.

3. Alle bisherigen Sperrmodi berechtigten dazu, die zulässigen Operationen beliebig oft auszuführen, die Atomarität der Transaktion ist hierdurch nicht gefährdet. Bei konfliktfreien Modifikationen galt dies deshalb, weil der Objektzustand keine Rolle spielte. Im Gegensatz dazu kann eine Bedingung an den Objektzustand vor einer zulässigen Modifikation erfüllt, danach aber verletzt sein. Z.B. ist die Bedingung "Schlange nicht leer" erfüllt, wenn die Schlange genau ein Element enthält; nach Ausführung von remove ist sie es nicht mehr.

Literatur

[Ko83] Korth, H.F.: Locking primitives in a database system; JACM 30:1, p.55-79; 1983/01

[CCT] Kelter, U.: Lehrmodul "Concurrency-Control-Theorie"; 2003

[SPV] Kelter, U.: Lehrmodul "Sperrverfahren"; 2003

[TID] Kelter, U.: Lehrmodul "Transaktionen und die Integrität von Datenbanken"; 2003

Index