Sperrverfahren

Udo Kelter

26.06.2007

Zusammenfassung dieses Lehrmoduls

Sperrverfahren sind die am meisten verbreiteten Concurrency-Control-Verfahren. Wir motivieren zunächst die parallele Ausführung von Transaktionen und den Begriff Serialisierbarkeit. Das simpelste Sperrverfahren ist der wechselseitige Ausschluß. An diesem erläutern wir die Grundprinzipien der Funktion von Sperren. Wir stellen einige weitere Protokolle vor, von denen das allgemeinste das 2-Phasen-Protokoll ist. Von diesem betrachten wir noch die Sonderfälle Sperren bis EOT und Preclaiming. Weiter werden die Isolationsstufen von SQL skizziert.

Vorausgesetzte Lehrmodule:

obligatorisch: – Transaktionen und die Integrität von Datenbanken

Stoffumfang in Vorlesungsdoppelstunden: 1.2

Inhaltsverzeichnis

1 Serialisierbarkeit **3**

2 On-line-Scheduler **6**

3 Grundlagen der Sperrverfahren **7**

4 Protokolle **10**
4.1 Wechselseitiger Ausschluß 12
4.2 Protokolle mit höherer Parallelität 13
4.3 Das Zwei-Phasen-Protokoll 15
4.3.1 Sperren bis EOT 17
4.3.2 Preclaiming 18

5 Isolationsstufen **19**

Literatur 22
Glossar 22
Index 23

©2006 Udo Kelter **Stand: 26.06.2007**
Dieser Text darf für nichtkommerzielle Nutzungen als Ganzes und unverändert in elektronischer oder gedruckter Form beliebig weitergegeben werden und in WWW-Seiten, CDs und Datenbanken aufgenommen werden. Jede andere Nutzung, insb. die Veränderung und Überführung in andere Formate, bedarf der expliziten Genehmigung. Die jeweils aktuellste Version ist über http://kltr.de erreichbar.

1 Serialisierbarkeit

Betriebliche Informationssysteme müssen i.d.R. eine große Anzahl an Benutzern gleichzeitig bedienen. Die Benutzer rufen Transaktionsprogramme auf und führen so einzelne Transaktionen auf der Datenbank durch.

Wenn man diese Transaktionsprogramme ungeschützt auf den gleichen Daten arbeiten läßt, kommt es zu Interferenzen und diversen Fehlern. Das bekannteste Beispiel ist eine verlorene Änderung: zwei Transaktionen lesen das gleiche Datenelement, berechnen einen neuen Wert und schreiben ihren jeweiligen neuen Wert in die Datenbank zurück. Der zuerst geschriebene Wert wird dann überschrieben, d.h. der Effekt der zugehörigen Transaktion geht verloren.

In manchen Fällen kann man das Problem lösen, indem man die parallele Ausführung von Transaktionen komplett vermeidet und sie strikt hintereinander ausführt. Dieses Vorgehen ist indessen bei größeren Systemen unannehmbar. Wenn eine Transaktionsausführung durchschnittlich 0.5 Sekunden dauert, kann man pro Minute maximal 120 Transaktionen ausführen; dieser Durchsatz ist oft zu wenig. Noch gravierender sind die entstehenden Antwortzeiten; wenn viele Benutzer gleichzeitig Transaktionen starten, können leicht Wartezeiten im Bereich von Minuten auftreten; üblicherweise wird eine Obergrenze von ca. 2 Sekunden verlangt.

Es muß also möglich sein, Transaktionen parallel auszuführen, ohne daß Interferenzen auftreten. Bild 1 zeigt ein Szenario, in dem zwei Benutzer je eine Transaktion aufrufen. Aus Sicht der Benutzer überlappen die Zeiträume, in denen die Transaktionen ausgeführt werden. Innerhalb der Transaktionen werden einzelne Aktionen aufgerufen. Unter einer **Aktion** verstehen wir hier irgendeinen lesenden oder schreibenden Zugriff auf die Datenbank. Im Beispiel rufen die beiden Transaktionen jeweils eine Folge von Aktionen auf: a11-a12-a13 bzw. a21-a22-a23-a24. Wir nehmen hier zur Vereinfachung an, daß die Aktionen innerhalb des Kerns strikt seriell ausgeführt werden. Aus Sicht des Kerns werden die Aktionsfolgen der einzelnen Transaktionen verzahnt ausgeführt, im Beispiel ist es die Verzahnung a11-a21-a12-

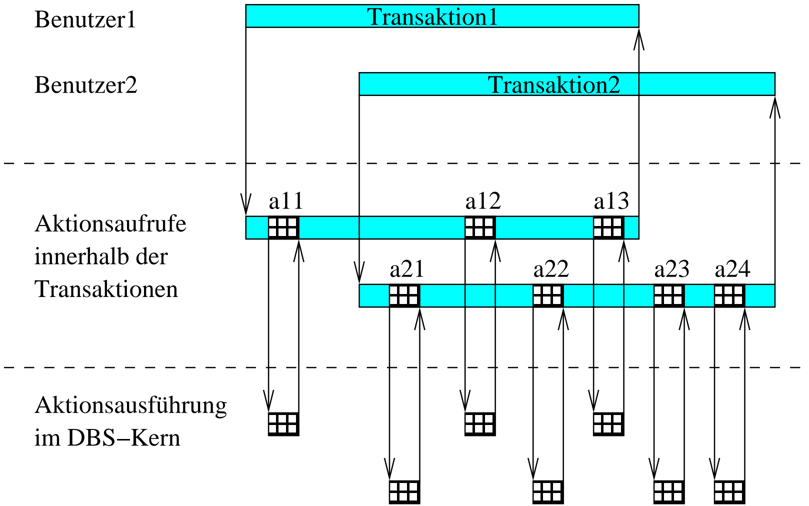


Abbildung 1: Parallelität von Transaktionen

a22-a13-a23-a24.

Das DBMS steht nun vor der Aufgabe, nur solche Verzahnungen zuzulassen, bei denen keine Interferenzen auftreten. Man spricht hier von der **Nebenläufigkeitssteuerung—seeConcurrency Control** (*Concurrency Control*, Abk.: **CC**) im DBMS.

Informell kann man "Abwesenheit von Interferenzen" so definieren, daß die Transaktionen *scheinbar* eine nach der anderen jeweils auf einen Schlag, sozusagen atomar, ausgeführt worden sind. Eine exakte Definition der "Abwesenheit von Interferenzen" ist alles andere als trivial und Gegenstand eines eigenen Zweigs der Theoretischen Informatik, der Concurrency-Control-Theorie. Das letztliche Resultat dieser Theorie kann wie folgt zusammengefaßt werden: eine Verzahnung verursacht keine Interferenzen, wenn es in ihr für jede Transaktion einen Serialisierungspunkt gibt; dann nennt man die Verzahnung **serialisierbar**. Ein **Serialisierungspunkt** einer Transaktion ist ein Zeitpunkt, zu dem alle Aktionen der Transaktion "konfliktfrei" hingeschoben werden können. Aus Sicht der Benutzer findet die

©2006 Udo Kelter

ganze Transaktion scheinbar atomar auf einen Schlag am Serialisierungspunkt statt.

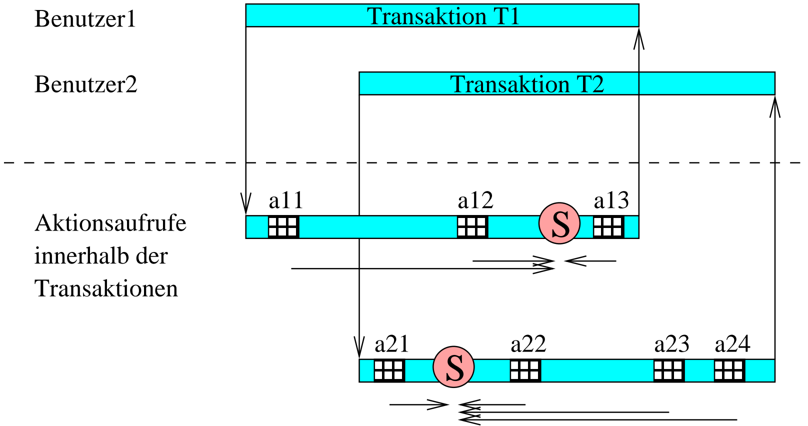


Abbildung 2: Serialisierungspunkte von Transaktionen

Bild 2 zeigt dies an einem Beispiel: die Serialisierungspunkte der beiden Transaktionen sind durch ein eingerahmtes S gekennzeichnet. Aus Sicht der Benutzer liegt der Serialisierungspunkt der Transaktion T1 hinter dem von T2, d.h. die Wirkung von T2 tritt logisch gesehen zuerst ein. Daß T1 früher angefangen hat und vielleicht sogar als erstes eine Aktion ausgeführt hat, sagt nichts über die logische Reihung aus.

Wenn die Aktionen einer Transaktion zum Serialisierungspunkt verschoben werden müssen, dann kann der Fall auftreten, daß die Reihenfolge von zwei Aktionen, die zu verschiedenen Transaktionen gehören, vertauscht werden muß. In unserem Beispiel muß u.a. die Reihenfolge von a11 und a22 vertauscht werden. Eine solche Reihenfolgevertauschung ist nur zulässig, wenn sie *keinen Einfluß auf die Wirkung der Transaktionen* hat.

Wir gehen i.f. von Aktionen der Form `read(X)` und `write(X)` aus, worin X ein unabhängig les- bzw. schreibbares Datengranulat (Tupel, Objekt, Satz o.ä.) ist. Unter dieser Annahme kann die Reihenfolge

von zwei Aktionen ohne Auswirkungen vertauscht werden, wenn

- sie verschiedene Datengranulate betreffen oder
- beide Aktionen lesende Aktionen sind.

In diesem Fall nennen wir die beiden Aktionen **konfliktfrei**.

2 On-line-Scheduler

Die Serialisierbarkeit ist zunächst ein Korrektheitskriterium für die verzahnte Ausführung *vollständiger* Transaktionen. Wir können also erst, nachdem alle laufenden Transaktion beendet sind, entscheiden, ob die aufgetretene Verzahnung zulässig war oder nicht. Bei den meisten CC-Verfahren müssen wir aber sofort entscheiden, ob eine Aktion ausgeführt werden kann, wir können nicht warten, bis alle laufenden Transaktion beendet sind. Betrachten wir hierzu Bild 3. Jede einzelne Transaktion verursacht eine Sequenz von Aktionsaufrufen.

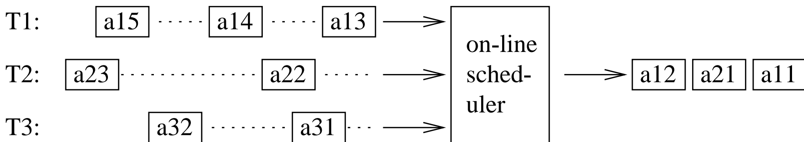


Abbildung 3: Parallelität von Transaktionen

Immer dann, wenn eine Transaktion erneut eine Aktion aufruft, muß eine DBMS-Komponente, die wir **Scheduler** nennen, entscheiden, ob diese Aktion sofort oder erst später ausgeführt wird. Diese Entscheidungen bestimmen, wie die einzelnen Sequenzen von Aktionsaufrufen zu einer einzigen Sequenz von effektiven Aktionsausführungen zusammengesetzt werden. Die Entscheidungen können nur auf Informationen über die neuen Aktionen und über die bisher schon durchgeführten Aktionen der Transaktionen basieren, nicht auf zukünftig vielleicht folgenden Aktionsaufrufen, denn der Scheduler kann nicht hellsehen. Wir sprechen daher von einem **On-line-Scheduler**.

Da jede Transaktion jederzeit ein Commit verlangen kann, folgt hieraus, daß die Serialisierbarkeit auch für beliebige Anfangsstücke der

effektiven Verzahnung gelten muß. Die Aufgabe lautet also, für jede Transaktion ständig einen Serialisierungspunkt zu garantieren.

Wenn man annimmt, daß zu einem Zeitpunkt immer nur ein Aktionsaufruf eintrifft und dieser sofort ausgeführt wird, definieren die parallelen Transaktionen eine gemeinsame **Aufrufsequenz**. Die Aufgabe des Schedulers kann darin gesehen werden, diese Aufrufsequenz in eine effektive **Ausführungssequenz** umzuformen.

3 Grundlagen der Sperrverfahren

Serialisierungspunkte können durch eine sehr einfache (und relativ grobe) Technik garantiert werden. Alle Objekte, die eine Transaktion benutzt (aber nicht die ganze DB), werden während ihrer gesamten Laufzeit *exklusiv* für sie reserviert. Die Objekte werden somit von den parallelen Transaktionen unter **wechselseitigem Ausschluß** benutzt.

Bei den so entstehenden Ausführungssequenzen ist jeder Zeitpunkt zwischen Beginn und Ende einer Transaktion ein gültiger Serialisierungspunkt für diese Transaktion.

Der wechselseitige Ausschluß kann dadurch erreicht werden, daß jede Transaktion die von ihr benutzten Objekte gleich zu Beginn *sperrt*. Eine andere Transaktion, die zu einem gesperrten Objekt zugreifen will, muß *warten*, bis dieses Objekt wieder freigegeben ist. Damit haben wir bereits ein sehr einfaches Sperrverfahren, bei dem garantiert ist, daß die entstehenden Ausführungssequenzen serialisierbar sind.

Grundprinzip des Sperrens. Zu einem gesperrten Objekt sollen andere Transaktionen als die, die es gesperrt hat, nicht zugreifen. Doch wer hindert sie daran? Um die Beachtung von Sperren durchzusetzen, führen wir folgendes *Grundprinzip* ein:

> *Nur Inhaber von Sperren dürfen zu Datenbankobjekten zugreifen.*

Verantwortlich für die Durchsetzung dieses Grundprinzips ist das DBMS. Das DBMS führt nur dann Zugriffe zu einem Objekt durch,

©2006 Udo Kelter

wenn die Transaktion eine Sperre hält.

Sperrmodi. Eine Sperre verleiht dem Inhaber sozusagen das temporäre Recht, Zugriffe durchzuführen. Man klassifiziert die Zugriffe üblicherweise in lesende und schreibende und unterscheidet dazu passend Sperren. Die Art einer Sperre nennt man auch **Sperrmodus**. Die Sperrmodi unterscheiden sich durch folgende Merkmale:

- durch die Zugriffsrechte eines Inhabers und
- durch die Verträglichkeit (Kompatibilität) mit anderen Sperren.

Die Sperren, die in dem oben vorgestellten wechselseitigen Ausschluß benutzt wurden, waren **Schreibsperren** (bzw. **exklusive Sperren**, **X-Sperren**, *exclusive locks*). Ihre Merkmale sind:

- ein Inhaber darf beliebig zu einem Objekt zugreifen.
- Schreibsperren sind mit keiner anderen Sperre verträglich.

Eine andere sehr häufige Art von Sperren sind **Lesesperren** (bzw. *shared locks* oder **S-Sperren**). Ihre Merkmale sind:

- Ein Inhaber darf nur lesend zum gesperrten Objekt zugreifen.
- Eine Lesesperre ist verträglich mit anderen Lesesperren, nicht jedoch mit Schreibsperren.

Die Verträglichkeit verschiedener Arten von Sperren kann übersichtlich durch eine **Verträglichkeitsmatrix** dargestellt werden. Ein + bzw. − zeigt an, ob die beantragte Sperre mit der vorhandenen Sperre verträglich ist oder nicht.

vorhandene	beantragte Sperre:	
Sperre:	S-Sperre	X-Sperre
keine	+	+
S-Sperre	+	−
X-Sperre	−	−

Anforderung und Freigabe von Sperren. Das DBMS prüft bei jedem Zugriffsversuch, ob eine Sperre vorhanden ist, die für diesen

Zugriff ausreichende Rechte verleiht. Falls nicht, wird normalerweise *implizit* eine Sperre angefordert, d.h. die Applikation bemerkt – außer einer eventuellen Verzögerung – nichts von dieser Sperrenanforderung.

Am Ende einer Transaktion werden stets automatisch alle Sperren freigegeben, die eine Transaktion noch hält. Diese Automatik ist besonders für den Fall erforderlich, wo eine Transaktion durch Rollback beendet wird.

Die implizite Anforderung und Freigabe von Sperren ist sehr häufig und für die Entwickler von Applikationen am bequemsten, weil man sich überhaupt nicht explizit um Sperren kümmern muß.

Sperren können aber auch *explizit* angefordert oder freigegeben werden. Hierzu muß das DBMS geeignete Kommandos zur Verfügung stellen. Deren Einzelheiten hängen von den Besonderheiten des Datenbankmodells und der Schnittstellen des DBMS ab. In unseren Beispielen benutzen wir die folgenden einfachen Befehle:

- `XLOCK(o)`: Aufforderung zur Einrichtung einer exklusiven Sperre für das Objekt o.
- `SLOCK(o)`: Aufforderung zur Einrichtung einer Lesesperre für o.
- `REL(o)`: Freigabe der Sperre auf o. (Wir können davon ausgehen, daß die Art der Sperre dem DBMS bekannt ist, da eine Transaktion zu einem Zeitpunkt höchstens eine Sperre für o innehat.)

Statt o kann auch eine Liste von Identifikationen von Objekten angegeben werden, die alle gesperrt werden sollen.

In den folgenden Beispielen gehen wir immer davon aus, daß beantragte Sperren bei Verträglichkeit ohne Verzug eingerichtet werden. (Dies ist nicht immer sinnvoll, wie wir später sehen werden.) Bei Unverträglichkeit muß die anfordernde Transaktion auf die Freigabe der vorhandenen Sperren *warten*. Das Warten findet innerhalb der LOCK-Operation statt, d.h. eine Transaktion bemerkt selbst gar nicht, ob und wie lange sie bis zur Zuteilung der Sperre warten muß. Sie kann sich darauf verlassen, daß nach Beendigung der LOCK-Operation die gewünschten Sperren für sie eingerichtet sind.

©2006 Udo Kelter

Statt beliebig lange zu warten, kann man bei vielen DBMS eine maximale Wartezeit vorgeben. Wird die Sperre nicht innerhalb dieser Frist eingerichtet, wird die explizite oder implizite Sperrenanforderung abgebrochen, und die auslösende Aktion endet mit einem entsprechenden Fehlercode. Das Problem des Wartens wird so letztlich auf die Applikation übertragen.

4 Protokolle

Nachdem wir im letzten Abschnitt die Wirkung von Sperren definiert haben, ist noch offen, nach welchen Regeln Sperren von einer Transaktion beantragt und freigegeben werden. Solche Regeln nennt man ein **Protokoll**[1].

Da jedes Protokoll die Grundregel beachten muß, daß Objekte vor ihrer Benutzung gesperrt und am Ende freigegeben werden müssen, unterscheiden sich Protokolle vor allem dadurch, mit welchen Arten von Sperren sie arbeiten und wann Sperren angefordert und freigegeben werden.

Protokolle können auf zwei Arten eingehalten werden:

- "von Hand": Der Programmierer der Transaktion sorgt selbst dafür, daß Sperren durch explizite Kommandos richtig angefordert und freigegeben werden.
- automatisch: Das DBMS und ggf. der Compiler, mit dem die Transaktionsprogramme übersetzt werden, sorgt automatisch für die richtige Anforderung und Freigabe von Sperren.

Die automatische Realisierung der Protokolle ist bequemer und sicherer. Sofern ein DBMS bzw. Compiler nur ein einziges Protokoll unterstützt, braucht sich der Programmierer überhaupt nicht um das Protokoll zu kümmern, er muß nicht einmal seine Funktionsweise kennen. Sofern mehrere Protokolle unterstützt werden, muß bei

[1] Diese Benennung ist nicht sehr anschaulich, aber allgemein üblich. Aus der Sicht eines Objekts regelt ein Protokoll die Reihenfolge von Sperrungen, Zugriffen verschiedener Art und Freigaben.

der Übersetzung einer Transaktion oder auf andere Weise eines ausgewählt werden. Auf die Verträglichkeit verschiedener Protokolle muß der Programmierer selbst achten.

Die oben vorgestellten Kommandos zur Handhabung von Sperren sind nur bei der "von-Hand"-Realisierung der Protokolle sinnvoll. Bei automatischer Realisierung der Protokolle benötigt ein Programmierer entweder gar keine Sprachmittel oder nur Anweisungen zur Auswahl eines Protokolls.

In den nachfolgenden Beispielen soll vor allem die Wirkung der Protokolle gezeigt werden. Ob die Sperren automatisch oder von Hand angefordert und freigegeben werden, ist dabei unerheblich. Wir gehen davon aus, daß das jeweils diskutierte Protokoll befolgt wurde.

In den folgenden Beispielen benutzen wir die folgenden Notationen:

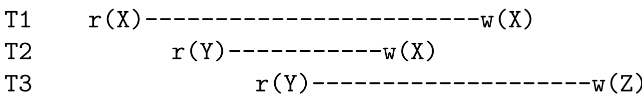
`r(X)` Lesen des Objekts X

`w(X)` Schreiben des Objekts X

`?...` Sperrenanforderung und nachfolgende Wartezeit

Anforderungen von Sperren werden nicht dargestellt, wenn sie sofort erfüllt werden können.

Wir nehmen i.f. immer die folgende Aufrufsequenz an:



```
T1     r(X)------------------------w(X)
T2           r(Y)-----------w(X)
T3                 r(Y)--------------------w(Z)
```

Sofern alle Zugriffe sofort ausgeführt werden, ist die resultierende Ausführungssequenz nicht serialisierbar. T2 müßte nämlich entweder logisch vor oder nach T1 liegen, dazu muß die Aktion `w(X)` von T2 entweder nach vorne verschoben, also mit `r(X)` von T1 vertauscht werden oder nach hinten verschoben, also mit `w(X)` von T1 vertauscht werden. Beides ist nicht konfliktfrei.

4.1 Wechselseitiger Ausschluß

Das Protokoll XPE. Das Protokoll, welches dem wechselseitigen Ausschluß zugrunde lag, war:

> **Protokoll XPE**: Zu Beginn jeder Transaktion werden alle Objekte, zu denen die Transaktion zugreifen wird, exklusiv gesperrt. (Am Ende der Transaktion werden alle Objekte automatisch freigegeben.)

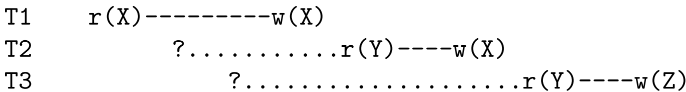
In der Abkürzung XPE bedeuten die Zeichen:

X Das Protokoll arbeitet nur mit exklusiven Sperren.

P Alle Sperren werden zu Beginn der Transaktion sozusagen im voraus angefordert (*preclaiming*).

E Alle Sperren werden erst bei EOT (*end of transaction*) freigegeben.

Die Wirkung des Protokolls XPE sei an unserem Standardbeispiel erläutert. Nehmen wir an, alle Transaktionen befolgen das Protokoll XPE. T2 beantragt dann anfangs eine Schreibsperre für Y *und* X. Wir gehen davon aus, daß Y dann sofort schreibgesperrt wird, obwohl die Sperre für X noch nicht eingerichtet werden kann. T3 beantragt anfangs eine Schreibsperre für Y und Z. Das folgende Bild zeigt die entstehende Ausführungssequenz:



Die entstandene Ausführungssequenz ist seriell, also mit Sicherheit korrekt. In der Tat ist die Aufrufsequenz unnötig stark umgeformt worden, denn Y wird von T2 und T3 nur gelesen.

Das Protokoll SPE. Wir erweitern das Protokoll XPE daher auf Lesesperren und erhalten das

> **Protokoll SPE**: Alle Objekte, zu denen im Laufe der Transaktion zugegriffen wird, werden zu Beginn gesperrt und bei EOT freigegeben. Alle Objekte, die nur gelesen werden, werden lesegesperrt, alle anderen schreibgesperrt.

Die obenstehende Aufrufsequenz führt bei Protokoll SPE zu folgender Ausführungssequenz:

```
T1    r(X)------------w(X)
T2          ?..............r(Y)--------w(X)
T3              r(Y)-------------w(Z)
```

T2 fordert zu Beginn eine Lesesperre für Y und eine Schreibsperre für X an; letztere kann nicht sofort zugeteilt werden.

Die Ausführungssequenz ist wieder serialisierbar, aber nicht mehr seriell. Im Vergleich zum Protokoll XPE muß T3 nicht mehr warten. Das Protokoll SPE erlaubt daher ein höheres Maß an Parallelität.

Endlosblockierungen. Wir hatten oben festgelegt, daß beantragte Sperren bei Verträglichkeit mit den vorhandenen Sperren ohne Verzug eingerichtet werden. Nehmen wir nun an, eine Transaktion T beantragt eine Schreibsperre für ein lesegesperrtes Objekt X, muß also warten. Während der Wartezeit beantragt eine andere Transaktion wieder eine Lesesperre für X und erhält sie, da X lesegesperrt ist. Die Zuteilung der beantragten Schreibsperre wird unendlich lange hinausgezögert, wenn immer wieder neue Transaktionen X in überlappenden Zeiträumen lesesperren. Dies nennt man eine **Endlosblockierung**. T ist endlos blockiert, obwohl das System weiterhin arbeitet, T "verhungert" sozusagen. Blockierungen dieser Art können nur vermieden werden, wenn irgendwann eine beantragte Lesesperre nicht eingerichtet wird, obwohl es sofort möglich wäre.

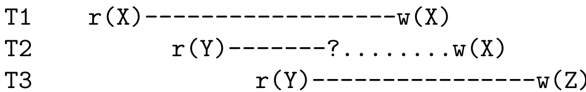
4.2 Protokolle mit höherer Parallelität

Das Protokoll SE. Eine genauere Inspektion der Ausführungssequenz, die bei SPE entstanden war, zeigt, daß auch T1 und T2 zumin-

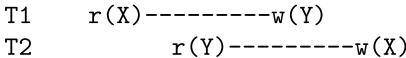
dest teilweise parallel ausgeführt werden könnten: Der Zugriff von T2 zu Y wird unnötig verzögert, da noch auf die Freigabe von X durch T1 gewartet werden muß. X wird von T2 schon zu Beginn gesperrt, obwohl erst am Ende zugegriffen wird. Der Zugriff zu Y würde nicht unnötig verzögert, wenn X erst unmittelbar vor seiner Benutzung gesperrt würde. Wir erhalten dann das neue Protokoll:

> **Protokoll SE**: wie Protokoll SPE, abweichend von diesem werden jedoch alle Objekte erst unmittelbar vor ihrer ersten Benutzung gesperrt.

Unsere Beispielaufrufsequenz führt bei Protokoll SE zu folgender Ausführungssequenz. T2 wird, nachdem es bereits lief, während der Anforderung der Schreibsperre für X in einen Wartezustand versetzt, der beendet wird, sobald T1 seine Sperre auf X freigibt.



Beim Protokoll SE wurde gegenüber SPE die Parallelität erhöht und die Wartezeit (von T2) verkürzt. Dieser Vorteil ist allerdings mit einem erheblichen Nachteil verbunden: Beim Protokoll SE sind Deadlocks möglich. Die folgende Aufrufsequenz führt beim Protokoll SE zu einem Deadlock:



Nachdem T1 und T2 jeweils ihren ersten Zugriff ohne Verzögerung ausführen konnten, warten sie später beide darauf, daß der andere ein Objekt freigibt. Auf das Deadlock-Problem und seine Lösungen kommen wir später zurück. Festzuhalten ist, daß beim Protokoll SE spezielle Vorsorgemaßnahmen gegen Deadlocks nötig sind. Der Aufwand hierfür kann durchaus den Leistungsgewinn durch höhere Parallelität gegenüber dem Protokoll SPE übersteigen.

Der Deadlock im vorstehenden Beispiel würde nicht eintreten, wenn die Sperren nicht "unnötig lange", d.h. bis zum Commit, gehalten würden, also nach der letzten Benutzung eines Objekts freigegeben würden. Das so entstehende Protokoll würde allerdings die Korrektheit der entstehenden Verzahnungen nicht mehr garantieren! So würde die obige Aufrufsequenz ohne Verzögerungen ausgeführt; die identische Ausführungssequenz ist aber nicht serialisierbar.

4.3 Das Zwei-Phasen-Protokoll

Die Grundform. Die Zeitdauer, während der ein Objekt gesperrt ist, soll natürlich so kurz wie möglich sein. Das letzte Beispiel zeigt, daß es allerdings nicht ausreicht, ein Objekt nur für die Dauer seiner Benutzung zu sperren. Bei den anderen bisher vorgestellten Protokollen wurde die Sperrzeit bis zum Commit oder zum Beginn der Transaktion verlängert, was u.U. zu lang ist. Das folgende Protokoll vermeidet eine solche starre Festlegung und garantiert dennoch die Serialisierbarkeit:

> **2-Phasen-Protokoll (2PL,** *2-phase locking*): Wenn eine Sperre freigegeben worden ist, werden keine neuen Sperren mehr angefordert.

Die Grundregel des Sperrens legt ohnehin fest, daß ein Objekt vor seiner ersten Benutzung gesperrt und nach seiner letzten Benutzung und spätestens bei Commit freigegeben werden muß. Ferner nehmen wir an, daß Objekte, die nur gelesen werden, lesegesperrt werden, und daß alle anderen Objekte schreibgesperrt werden.

Eine Transaktion, die ihre Sperren unter Beachtung des 2-Phasen-Protokolls anfordert und freigibt, heißt auch **2-Phasen-gesperrt** (**two-phase-locked**).

Die beiden typischen Phasen, durch die die Bezeichnung für dieses Protokoll motiviert ist, sind:

1. **Wachstumsphase**: Die Zeit vom Beginn der Transaktion bis zur letzten Anforderung einer Sperre. In dieser Zeit werden keine Sperren freigegeben.

2. **Schrumpfungsphase**: Die Zeit von der ersten Freigabe einer Sperre bis Commit. In dieser Zeit werden keine Sperren mehr angefordert.

In der Zeit zwischen den beiden Phasen werden Sperren weder angefordert noch freigegeben; sofern sie überhaupt interessiert, kann man sie **Verarbeitungsphase** nennen.

Die Zahl der gehaltenen Sperren hat beim 2-Phasen-Protokoll den in Bild 4 gezeigten typischen Verlauf. Dieses Bild soll natürlich *nicht* andeuten, daß Sperren in umgekehrter Reihenfolge angefordert und freigegeben werden müssen.

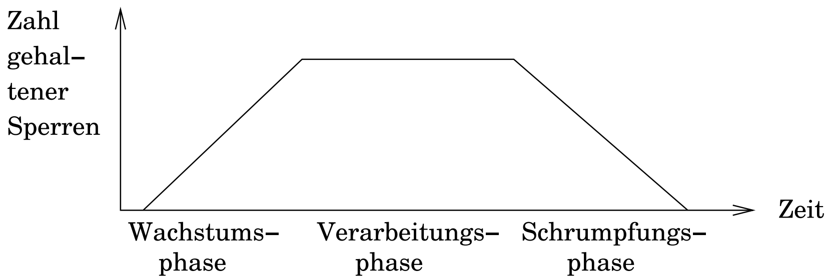


Abbildung 4: Zwei-Phasen-Protokoll

Korrektheit des 2-Phasen-Protokolls. Bei 2-Phasen-gesperrten Transaktionen sind alle Ausführungssequenzen serialisierbar. Jeder Zeitpunkt in der Verarbeitungsphase ist nämlich ein Serialisierungspunkt für diese Transaktion. Dies ergibt sich daraus, daß jedes Objekt von seiner ersten bis zu seiner letzten Benutzung und während der gesamten Verarbeitungsphase ununterbrochen gesperrt ist. Jeder Zugriff kann also konfliktfrei in die Verarbeitungsphase verschoben werden.

Die Korrektheit des 2-Phasen-Protokolls ist insofern wichtig, als alle bisher vorgestellten Protokolle Sonderfälle des 2-Phasen-Protokolls sind und damit auch ihre Korrektheit gezeigt ist.

Bei den bisherigen Protokollen wurde im Vergleich zum 2-Phasen-Protokoll die Sperrzeit der Objekte in eine oder beide Richtungen maximal verlängert. Betrachten wir beide Optionen genauer:

4.3.1 Sperren bis EOT

Alle Sperren werden bis EOT gehalten. Explizit freigegeben wird keine Sperre, da bei Commit bzw. Rollback automatisch alle vorhandenen Sperren freigegeben werden. (Durch die Regel: "keine Sperre freigeben" ist das 2-Phasen-Protokoll bereits erfüllt!) Die oben gezeigte Phasenkurve bekommt eine senkrecht absteigende Flanke:

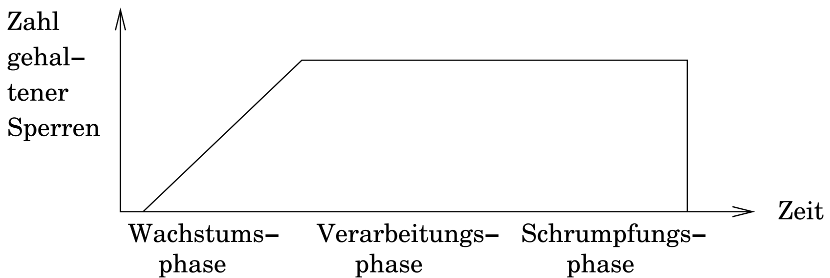


Abbildung 5: Sperren bis EOT beim Zwei-Phasen-Protokoll

Nachteil des Sperrens bis EOT ist die gegenüber der Grundform des 2-Phasen-Protokolls verlängerte Sperrzeit, die im Endeffekt zu einer geringfügigen Verminderung der Parallelität und damit zu einer Verlängerung der mittleren Wartezeit einer Transaktion führt. Außer bei sehr langen Transaktionen ist dieser Nachteil jedoch unbedeutend.

Ein Vorteil ist eine Verringerung des Verwaltungsaufwandes: Die Abarbeitung vieler einzelner Freigaben von Sperren ist i.a. aufwendiger als die automatische Freigabe aller Sperren auf einen Schlag.

Ein entscheidender Vorteil liegt darin, daß keine Fortpflanzung von Rollback auftreten kann. Wenn man eine Schreibsperre vor EOT freigibt, dann enthält das entsprechende Objekt einen **unsicheren Wert**:

möglicherweise wird die Transaktion später zurückgesetzt, und dann wird dieser Wert wieder zurückgesetzt. Wenn eine andere Transaktion den unsicheren Wert liest, muß sie bei einem Rollback der ersten Transaktion auch zurückgesetzt werden; dies nennt man eine **Rollback-Fortpflanzung**. Es sind sogar Ketten von Rollback-Fortpflanzungen denkbar. Rollback-Fortpflanzungen sind normalerweise aus vielen Gründen völlig inakzeptabel, d.h. zumindest Schreibsperren müssen schon deshalb bis EOT gehalten werden.

Insgesamt ist das Sperren bis EOT sehr zu empfehlen. Wenn überhaupt, dann sollten höchstens Lesesperren vor EOT freigegeben werden.

4.3.2 Preclaiming

Bei dieser Technik werden alle Objekte schon zu Beginn der Transaktion gesperrt. Entscheidend ist, daß alle Objekte auf einmal gesperrt werden. Die Phasenkurve bekommt eine senkrechte "steigende" Flanke:

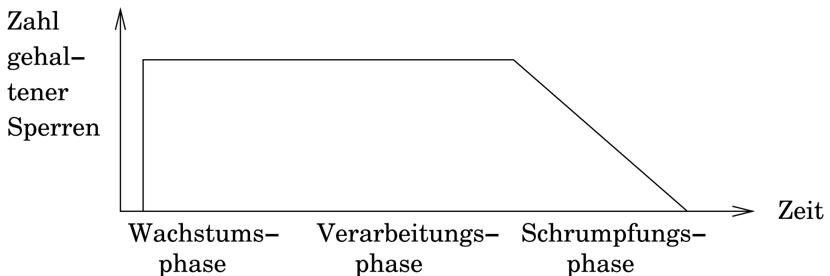


Abbildung 6: Preclaiming beim Zwei-Phasen-Protokoll

Nachteil des Preclaiming ist (wie beim Sperren bis EOT) die Verlängerung der Sperrzeit, die aber höchstens bei langen Transaktionen ins Gewicht fällt.

Sehr vorteilhaft beim Preclaiming ist, daß kein Deadlock auftreten

kann. Deadlocks sind im Prinzip zyklische Wartesituationen, die durch das DBMS mittels Rollback und Neustart einer der beteiligten Transaktionen aufgelöst werden müssen. Um zu erkennen, ob ein Deadlock vorliegt, müssen aufwendige Zyklustests durchgeführt werden. Der Aufwand hierfür ist so hoch, daß in vielen DBMS eine Transaktion einfach nach Überschreiten einer maximalen Wartezeit auf Verdacht zurückgesetzt und neu gestartet wird.

Für das Preclaiming muß spätestens beim ersten Zugriff in einer Transaktion bekannt sein, zu welchen Objekten während der gesamten Transaktion zugegriffen werden wird. Die Objektmenge muß von der Transaktion selbst herausgefunden und durch eine explizite Sperr-Anweisung dem DBMS bekanntgegeben werden. Leider ist diese Vorausbestimmung der Objektmenge nicht immer möglich. Dies kann verschiedene Ursachen haben:

- Zugriffe können datenabhängig sein, d.h. die bei früheren Zugriffen gelesenen Werte bestimmen, zu welchen Objekten später zugegriffen werden wird.
- Die Objekte werden sukzessive durch Navigieren erreicht, wobei Sperren auf den vorher besuchten Objekten angefordert werden.

In diesen Fällen muß man, sofern man am Preclaiming festhalten will, eine Obermenge der tatsächlich benötigten Objekte sperren, z.B. eine ganze Relation. (Diese Sperrung läßt sich mit Hilfe besonderer Techniken effizient durchführen.) Die Zahl der gesperrten Objekte wird hierdurch oft um Größenordnungen erhöht. Deshalb und wegen der zusätzlichen Verlängerung der Sperrzeit kann die Parallelität erheblich reduziert werden, so daß das Preclaiming eher nachteilig wird.

Insgesamt führen die Probleme dazu, daß man das Preclaiming nicht generell voraussetzen kann. Mechanismen zur Deadlock-Behandlung müssen daher im DBMS auf jeden Fall vorhanden sein.

5 Isolationsstufen

Die Serialisierbarkeit ist sozusagen der perfekte Schutz paralleler Transaktionen untereinander. Diese Perfektion hat ihren Preis, näm-

©2006 Udo Kelter

lich Wartezeiten und schlechteren Durchsatz. Wenn beispielsweise in einem Informationssystem regelmäßig komplexe Abfragen auftreten, für die ganze Relationen gelesen werden müssen, und wenn diese Abfragen serialisierbar ausgeführt werden sollen, können währenddessen keine Daten eingegeben oder verändert werden. Wenn die Abfragen länger dauern (z.B. 30 Sekunden), ist das fast so schlimm wie eine Betriebsstörung und kann je nach Umständen inakzeptabel sein. In diesem Fall muß die Serialisierbarkeit zugunsten einer besseren Performanz des Gesamtsystems geopfert werden. SQL bietet hierzu mehrere Isolationsstufen an, die pro Transaktion individuell gewählt werden kann; im einzelnen:

`read uncommitted`: Transaktionen mit dieser Isolationsstufe müssen Lese-Transaktionen sein[2]; sie werden überhaupt nicht vor parallelen Transaktionen geschützt. Daher brauchen keine Sperren gesetzt zu werden, es werden auch keine anderen Transaktionen durch die (Lese-) Sperren gebremst, und es entsteht kein Aufwand zur Verwaltung der Sperren. Der Name der Isolationsstufe rührt daher, daß sogar unsichere Werte gelesen werden.

Im Prinzip können hier beliebige Fehler auftreten, in der Praxis können je nach Umständen die Fehler aber selten und ihre Effekte vernachlässigbar sein, z.B. bei unkritischen statistischen Auswertungen.

`read committed`: Solche Transaktionen können keine unsicheren Werte lesen, d.h. ein entsprechender Leseversuch würde zum Warten auf das Commit der anderen Transaktion führen.

Es werden aber keine Lesesperren für gelesene Werte angefordert. Wenn der gleiche Datenwert zweimal innerhalb der Transaktion gelesen wird, kann er zwischendurch von einer anderen Transaktion verändert worden sein. Dies bezeichnet man als nichtwiederholbares Lesen[3].

---

[2]SQL erlaubt es, eine Transaktion als `read only` anzugeben; sie kann dann nicht mehr schreiben.

[3]Der Ausdruck ist nicht wörtlich zu nehmen. Die Leseoperation kann natürlich wiederholt werden. Allerdings wird nicht wiederholt der gleiche Wert zurückgeliefert.

`repeatable read`: Auf dieser Stufe wird eine Lesesperre für gelesene Tupel angefordert und bis EOT gehalten, das Problem des nicht wiederholbaren Lesens kann nicht mehr auftreten. Es können aber Phantome auftreten (Detail s.u.)

`serializable`: Diese Stufe garantiert die Serialisierbarkeit. Phantome können hier nicht auftreten.

In allen Fällen werden für erzeugte oder veränderte Tupel Schreibsperren angefordert und bis EOT gehalten.

Die Isolationsstufe `repeatable read` garantiert überraschenderweise nicht, daß eine SQL-Abfrage, die zweimal unverändert ausgeführt wird, jedesmal das gleiche Ergebnis liefert. Die bei der ersten Ausführung selektierten Tupel werden zwar alle einzeln lesegesperrt und sind daher unverändert im Ergebnis der zweiten Ausführung enthalten[4], eine andere Transaktion könnte aber inzwischen weitere Tupel erzeugt haben, die die Selektionskriterien der Abfrage erfüllen. Derartige Tupel nennt man **Phantome**, weil sie bei der ersten Ausführung noch nicht existierten und bei der zweiten Ausführung sozusagen aus dem Nichts auftauchen. Phantome können ebenfalls zu Parallelitätsanomalien führen.

Phantome bzgl. einer bestimmten Abfrage in einer Transaktion zu verhindern bedeutet, für alle Relationen, die in der Abfrage vorkommen, die Bereiche gegen Veränderungen zu sperren, die für das Abfrageergebnis relevant sind. Änderungs- und Löschoperationen sind kein Problem, denn sie betreffen existierende Tupel und werden schon durch die Lesesperren an diesen Tupeln verhindert. Zusätzlich verhindert werden müssen nur Einfügungen.

Die betroffenen Bereiche sind gerade durch die Selektionsprädikate der Abfrage gegeben und können sehr groß sein. Die resultierenden Blockierungen können daher sehr störend sein.

[4] Wir unterstellen hier eine Abfrage ohne Aggregation.

©2006 Udo Kelter

Literatur

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

Glossar

2-Phasen-Protokoll (*2-phase locking protocol*): Sperrprotokoll mit der Hauptregel, daß keine neuen Sperren mehr angefordert werden, sobald irgendeine Sperre freigegeben worden ist

Concurrency Control: Verhinderung von Interferenzen zwischen parallelen Transaktionen oder Behebung unzulässiger Effekte eingetretener Interferenz

Deadlock (bzw. Verklemmung): Wartezyklus bestehend aus einer Menge von Prozessen, von denen jeder wenigstens eine Sperre hält, auf deren Freigabe der “nächste” Prozeß wartet, und wenigstens eine zweite Sperre anfordert, die nicht zugeteilt werden kann, weil das betroffene Objekt für den “vorigen” Prozeß gesperrt ist

Isolationsstufen: Ausführungsvarianten von Folgen von Aktionen, bei denen mehr oder weniger umfangreich auf die Transaktionseigenschaften verzichtet wird (zugunsten besserer Performance)

kompatibel: Synonym zu verträglich

konfliktfrei: zwei Aktionen sind konfliktfrei, wenn ihre Reihenfolge ohne äußerlich sichtbare Auswirkungen vertauscht werden kann, also wenn sie verschiedene Datengranulate betreffen oder beide Aktionen nur lesen

Lesesperre: Sperre mit Sperrmodus “Lesen”, d.h. dem Inhaber sind nur lesende Zugriffe erlaubt

On-line-Scheduler: Komponente des DBMS-Kerns, der ankommende Aktionsaufrufe entgegennimmt und entscheidet, ob sie sofort oder später ausgeführt werden; on-line drückt aus, daß sofort entschieden wird und nicht auf das Ende der involvierten Transaktionen gewartet wird und, wenn möglich, aufgerufene Aktionen sofort ausgeführt werden

Preclaiming: Anforderung *aller* Sperren, die im Verlauf einer Transaktion benötigt werden, zu Beginn der Transaktion auf einen Schlag

Protokoll: im Kontext von Sperrverfahren: Menge von Regeln, wann Sperren in welchem Modus angefordert bzw. freigegeben werden

©2006 Udo Kelter

Rollback-Fortpflanzung: tritt ein, wenn eine Transaktion T1 eine Schreibsperre vor Transaktionsende freigibt, eine Transaktion T2 einen unsicheren Wert aus dem betroffenen Objekt liest, T1 zurückgesetzt wird und deshalb auch T2 zurückgesetzt werden muß

Schreibsperre: Sperre mit Sperrmodus "Schreiben", d.h. dem Inhaber sind beliebige Zugriffe erlaubt

serialisierbar: eine verzahnte Ausführung mehrere Transaktionen ist serialisierbar, wenn es für jede Transaktion einen Serialisierungspunkt gibt, zu dem alle Aktionen der Transaktion "konfliktfrei" hingeschoben werden können

Sperre: eine Sperre bezieht sich auf ein sperrbares Objekt und hat einen Prozeß als Besitzer; beruht auf dem Konzept, daß ein Prozeß nur dann zu einem zu sperrbaren Objekt zugreifen darf, wenn er Inhaber einer Sperre für dieses Objekt ist; wenn ein Objekt gesperrt ist, können keine weiteren Sperren eingerichtet werden, die mit den vorhandenen Sperren inkompatibel sind

Sperrmodus: definiert die mit einer Sperre verbundenen Rechte des Inhabers (d.h. die Menge der erlaubten Operationen auf dem Objekt)

Transaktion: Folge von Datenbankzugriffen (Aktionen), die einen konsistenten Datenbankzustand in einen neuen konsistenten Datenbankzustand überführt; Transaktionsausführungen haben folgende Eigenschaften (sofern nicht eine reduzierte Isolationsstufe gewählt wird): Fehler-Atomarität, Dauerhaftigkeit, Serialisierbarkeit, endliche Ausführungszeit

Transaktion: Folge von Datenbankzugriffen (Aktionen), die mit den Transaktionseigenschaften ausgeführt wird (oder Programm, das eine solche Folge von Aktionen auslöst) bei denen mehr oder weniger umfangreich auf die Transaktionseigenschaften verzichtet wird (zugunsten besserer Performance)

unsicherer Wert: Datenwert, der von einer nicht beendeten Transaktion geschrieben worden ist und bei dem jederzeit der Fall eintreten kann, daß er auf einen früheren Zustand zurückgesetzt wird, weil die Transaktion zurückgesetzt worden ist

verträglich: Beziehung zwischen Sperrmodi; wenn die Sperrmodi A bzw. B verträglich sind, können zwei Sperren mit diesen Modi gleichzeitig an einem Objekt vorhanden sein

©2006 Udo Kelter

Index

2-Phasen-Protokoll, 15, 22
2PL, 15

Aktion, 3
Aufrufsequenz, 7
Ausführungssequenz, 7

Concurrency Control, 4, 22

Deadlock, 14, 18, 22
Durchsatz, 3

Endlosblockierung, 13
EOT, 12
exclusive lock, 8

Interferenz, 3, 4
Isolationsstufe, 19, 22

kompatibel, *siehe Sperrmodus*
konfliktfrei, 6, 22

Lesesperre, *siehe Sperre*

Nebenläufigkeitssteuerung, *siehe Concurrency Control*

On-line-Scheduler, 6, 22

Phantome, 21
Preclaiming, 18, 22
Protokoll, 10, 22
  2PL, 15
  SE, 13
  SPE, 12
  XPE, 12

Rollback, 9
  Fortpflanzung, 17, 23

Scheduler, 6
Schreibsperre, *siehe Sperre*
Schrumpfungsphase, 16
Serialisierbarkeit, 3, 4, 21, 23
Serialisierungspunkt, 4, 7, 16
shared lock, 8
Sperre, 7, 23
  Anforderung, 8
    Endlosblockierung, 13
    explizite, 9
    implizite, 9
  exklusive, 8
  Freigabe, 8
  Kompatibilität, *siehe Sperrmodus*
  Lesesperre, 8, 12, 20, 22
  Schreibsperre, 8, 23
  Verträglichkeit, *siehe Sperrmodus*
Sperren bis EOT, 17
Sperrmodus, 8, 23
  Kompatibilität, *siehe Verträglichkeit*
  Verträglichkeit, 8, 23
  Verträglichkeitsmatrix, 8

Transaktion, 23

unsicher, 17, 20, 23

verträglich, *siehe Sperrmodus*
Verzahnung, 4

Wachstumsphase, 15
wechselseitiger Ausschluß, 7