Prof. Dr. Gunter Schlageter et. al.

Kurs 01672

Datenbanken II

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Leseprobe zum Kurs 1672, Datenbanken 2, entnommen aus KE 1

4.5 Sperrverfahren

4.5.1 Sperren als Synchronisationsmittel

Bei Sperrverfahren geschieht die Synchronisation der Transaktionen dadurch, dass für jede Transaktion diejenigen Teile der Datenbank, auf denen sie arbeiten will, gesperrt werden. Solange eine *Sperre* (*lock*) auf ein Objekt gesetzt ist, können andere Transaktionen auf dieses Objekt nicht zugreifen. Je nach den beabsichtigten Operationen wird man Objekte in unterschiedlicher Weise sperren: will eine Transaktion nur lesen, so braucht sie andere Transaktionen, die ebenfalls nur lesen wollen, nicht am Zugriff zu hindern. Zur Vereinfachung der Diskussion betrachten wir jedoch für den Moment nur eine einzige Art von Sperren, nämlich *exklusive Sperren,* bei denen auf das gesperrte Objekt keine andere Transaktion zugreifen kann.

Sperre
lock

Betrachten wir noch einmal das Beispiel 4.1: Würden T_1 und T_2 die Objekte a, b und c sperren, bevor sie diese lesen, und würden sie die Sperren erst nach dem Schreiben der Objekte wieder aufheben, so könnte Ablauf 2 nicht entstehen. T_1 könnte b zu dem angegebenen Zeitpunkt nicht lesen, da T_2 eine Sperre auf b hätte. T_1 müsste warten, bis b geschrieben und freigegeben wäre, so dass der Ablauf serialisierbar würde.

Die Aufgabe des Transaktionsmanagers besteht nun darin, bei jedem ankommenden Operationswunsch festzustellen, ob die Transaktion die erforderlichen Sperren besitzt. Falls nicht, müssen diese zunächst erworben werden. Falls die benötigten Objekte schon für eine andere Transaktion gesperrt sind, muss die anfordernde Transaktion in einen Wartezustand versetzt werden.

Die Verwaltung der Sperren innerhalb des DBMS erfolgt durch den *Lock-Manager* (siehe Abschnitt 4.5.7). Vom Lock-Manager wird eine Sperre auf Objekt a mit LOCK a angefordert, mit UNLOCK a wird die Sperre freigegeben.

Lock-Manager

LOCK
UNLOCK

Bei der Verwendung von Sperren sind die folgenden grundsätzlichen Probleme zu klären:

(1) *Sperrprotokoll*: Wie müssen die Sperren gesetzt und freigegeben werden, damit serialisierbare Schedules entstehen?

(2) *Sperrmodi*: Welche Arten von Sperren benötigt man?

(3) *Sperreinheit*: Welches sind die sperrbaren Einheiten in der Datenbank?

(4) *Deadlock*: Sobald exklusive Sperren Verwendung finden, kann ein Deadlock auftreten. Was ist zu tun?

(5) *Livelock*: Durch wiederholte ungünstige Zuteilung von Sperren nach dem Neustart einer Transaktion T kann T permanent blockiert werden. Was ist zu tun?

4.5.2 Sperrprotokoll

Betrachten wir wieder eine Bankentransaktion

T_1: a := a - 1000
b := b + 1000

Nehmen wir an, T_1 setzt eine Sperre auf a, erledigt a:=a-1000 und gibt a wieder frei. Danach sperrt T_1 das Objekt b, führt die Operation auf b aus und gibt b frei. Trotz der Sperren erhält nun eine Transaktion T_2 eine inkonsistente Sicht, wenn sie ihrerseits nach dem Freigeben von a, aber vor dem Sperren von b auf a und b zugreift (sperrt, liest, freigibt).

Das Beispiel zeigt, dass das Setzen und Freigeben von Sperren gewissen Regeln, sogenannten *Protokollen*, unterworfen werden muss, um Serialisierbarkeit zu erreichen.

Das folgende einfache Protokoll garantiert Serialisierbarkeit paralleler Transaktionen:

Zwei-Phasen-Sperrprotokoll (two-phase lock protocol):

(1) Vor dem ersten Zugriff auf ein Objekt muss die Transaktion das Objekt sperren.
(2) Nach dem ersten UNLOCK einer Transaktion T darf von T kein LOCK mehr ausgeführt werden.
(3) Spätestens mit ihrem Ende muss die Transaktion alle Sperren freigeben.

Jede Transaktion ist also zweiphasig in dem Sinne, dass in einer Wachstumsphase alle Sperren gesetzt und in einer dann folgenden Schrumpfungsphase die Sperren wieder freigegeben werden. In der Schrumpfungsphase dürfen keine weiteren Sperren angefordert werden (vgl. Bild 4.1).

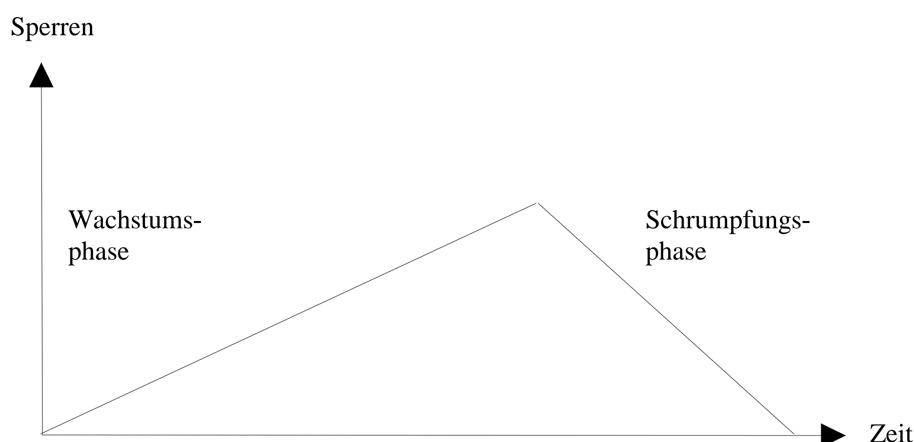

Bild 4.1: Zwei-Phasen-Sperrprotokoll

Um zu sehen, dass jedes System zweiphasiger Transaktionen serialisierbar ist, muss man zeigen, dass bei Anwendung dieses Protokolls ein Zyklus im Graphen G nicht existieren kann. Dies sieht man durch folgenden Widerspruchsbeweis (statt der Operationen betrachten wir jetzt einfach die Folge der LOCK-UNLOCK-Befehle):

Man nehme an, dass bei einem System zweiphasiger Transaktionen ein Zyklus entstanden sei: T_{i1} -> T_{i2} -> ... -> T_{ip} -> T_{i1}

Dies kann nur passieren, wenn ein LOCK von T_{i2} auf ein UNLOCK von T_{i1} folgen, ein LOCK von T_{i3} auf ein UNLOCK von T_{i2}, usw. Schließlich muss ein LOCK von T_{i1} auf ein UNLOCK von T_{ip} folgen. Dies bedeutet aber, es muss ein LOCK von T_{i1} auf ein UNLOCK von T_{i1} folgen. Dies ist ein Widerspruch zur Annahme der Zweiphasigkeit.

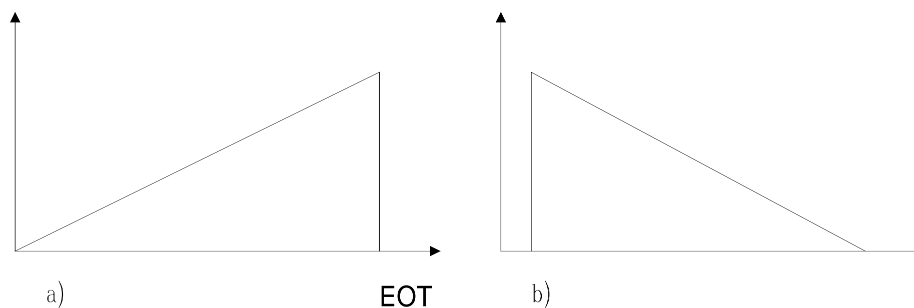

Bild 4.2: a) Strikte Zweiphasigkeit b) Preclaiming
EOT = end of transaction

Strikte Zweiphasigkeit

Das Zwei-Phasen-Sperrprotokoll wird meist in der Weise verschärft, dass alle Sperren erst zum Transaktionsende freigegeben werden dürfen. Wir sprechen von *strikter Zweiphasigkeit* (vgl. Bild 4.2 a). Der Grund hierfür ist, dass wir in der Lage sein wollen, eine Transaktion abzubrechen, ohne dass davon andere Transaktionen betroffen sind. Werden die Sperren nicht bis zum Transaktionsende gehalten, so kann *fortgepflanzter Rollback* auftreten: Transaktion T_1 gibt ein Objekt o vor ihrem Ende frei. Transaktion T_2 sperrt dieses Objekt, sobald es frei wird und beginnt mit ihrer Arbeit. Jetzt stürzt T_1 ab (irgendeine *abort* Situation), muss also zurückgesetzt und neu gestartet werden. Damit ist auch der von T_2 gelesene Wert von o nicht mehr gültig, d.h. T_2 arbeitet auf ungültigen Daten! Konsequenz: auch T_2 muss zurückgesetzt werden. Man beachte, dass T_2 beim Absturz von T_1 sogar schon beendet sein kann! (Wir kommen auf fortgepflanzten Rollback im Zusammenhang mit der Recovery noch einmal zurück.)

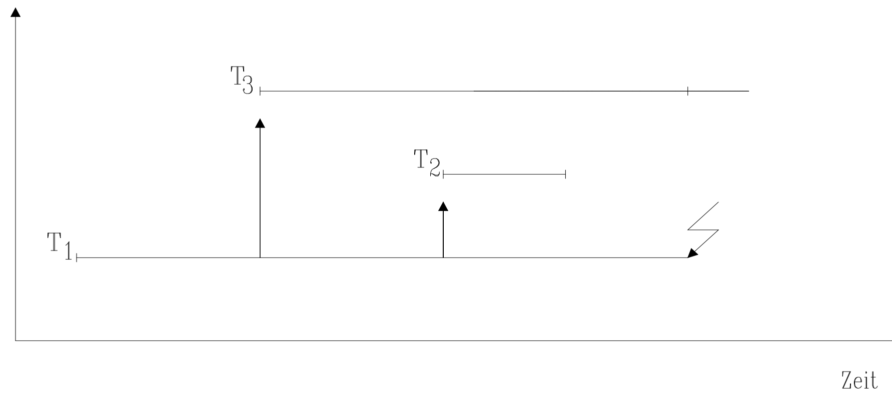

Bild 4.3: Fortgepflanzter Rollback: Transaktionen T_2 und T_3 lesen Objekte, die T_1 verändert hat, vor dem Ende von T_1. Bei Abbruch von T_1 müssen deshalb auch T_2 und T_3 zurückgesetzt werden (T_2 war sogar schon beendet!).

Die Fortpflanzung von Rollback ist nicht nur technisch aufwendig, sie kann insbesondere unangenehme oder sogar nicht akzeptable organisatorische Auswirkungen haben. Im Beispiel könnte T_2 den Verkauf eines Flugscheines realisieren: man sieht sofort das Problem, das entsteht, wenn T_2 zum Zeitpunkt des Absturzes von T_1 schon beendet sein kann (Flugschein schon in der Hand des Kunden!).

Preclaiming

Bei Preclaiming werden alle Objekte, die eine Transaktion T möglicherweise verwenden wird, zu Beginn von T, d.h. vor jeder Verarbeitung, gesperrt (vgl. Bild 4.2.b). Mit Preclaiming ist gewährleistet, dass eine Transaktion, die überhaupt zur Verarbeitung kommt, mit Sicherheit auch beendet werden kann: Da sie bereits über alle Objekte verfügt, kann sie nicht in einen Deadlock oder sonstigen Abbruch aus Synchronisationsgründen hineinlaufen.

In der Datenbankumgebung ist Preclaiming sehr problematisch, da die Spezifikation oder automatische Bestimmung der benötigten Objektmenge oft sehr schwierig ist. Die Sperrung einer zu großen Obermenge führt zu hohen Kosten und schwerwiegenden Einbußen an möglicher Parallelität. Oft ist die Sperrung einer viel zu großen Obermenge gar nicht vermeidbar, insbesondere dann, wenn das weitere Zugriffsverhalten einer Transaktion von Zwischenergebnissen abhängt. Kombiniert mit strikter Zweiphasigkeit führt Preclaiming in jedem Falle zu einer deutlichen Reduzierung der möglichen Parallelität.

Übung 4.4:

Oben wurden die drei Fehlerfälle Lost Update, inkonsistente Sicht bzw. inkonsistente Datenbank und Phantome beschrieben. Überlegen Sie, welches dieser Probleme durch das Zwei-Phasen-Sperrprotokoll gelöst wird.

4.5.3 Sperrmodi

Die exklusive Sperre, mit der wir bisher gearbeitet haben, ist offensichtlich sehr restriktiv: sie verhindert unnötigerweise, dass zwei Transaktionen gleichzeitig lesenden Zugriff auf ein Objekt bekommen können. Aus diesem Grunde unterscheidet man meist verschiedene Arten von Sperren; jede Sperre hat dann einen bestimmten *Modus*. Grundlegend ist die Unterscheidung von Lese- und Schreibsperren:

Die *Lesesperre (shared lock, S-Lock)* erlaubt der besitzenden Transaktion nur lesenden Zugriff auf das gesperrte Objekt. Sie erlaubt anderen Transaktionen das Setzen weiterer Lesesperren, verbietet aber das Setzen einer Schreibsperre.

Die *Schreibsperre (exclusive lock, X-Lock)* erlaubt der besitzenden Transaktion lesenden wie schreibenden Zugriff. Andere Transaktionen können keine gleichzeitigen Sperren für das Objekt erhalten.

Lesesperren sind also zueinander *kompatibel*, Lese- und Schreibsperre sowie Schreib- und Schreibsperre nicht.

Im Zusammenhang mit Sperrhierarchien werden weitere Sperrmodi, sogenannte *Intentions-Sperren*, eingeführt (behandeln wir etwas später).

In einigen Systemen gibt es die Möglichkeit, den Modus von Sperren im Laufe der Verarbeitung zu verändern, beispielsweise *upgrade* einer Lesesperre auf eine Schreibsperre. Diese Sperrumwandlung ist häufig auch durch den Transaktionsmanager selbst notwendig, nämlich immer dann, wenn der Transaktionsmanager beim Zugriff auf ein Objekt keine Information über die Absichten des Anwendungsprogramms hat. Liest die Transaktion ein Objekt a, so wird a mit einer Lesesperre belegt. Schreibt die Transaktion dann später das Objekt, so muss die Lesesperre zunächst in eine Schreibsperre umgewandelt werden. Diese Umwandlung scheitert natürlich dann, wenn eine andere Transaktion ebenfalls eine Lesesperre auf a hat (Verzögerung oder gar Deadlock!).

hot spots

In vielen Anwendungen gibt es Objekte, auf denen periodisch viele Transaktionen gleichzeitig schreiben müssen, sog. *hot spots*. In der Bankenwendung können beispielsweise Summenfelder *hot spots* darstellen, auf die bei jeder Ein- und Auszahlung gebucht werden muss. Solche *hot spots* können dazu führen, dass viele Transaktionen warten müssen, obwohl alle übrigen Daten für diese Transaktionen verfügbar sind und das System insgesamt nicht überlastet ist.

Dieses Performance-Problem kann oft dadurch vermieden werden, dass spezielle Operationen mit speziellen Sperrmodi eingeführt werden. Der Engpass entsteht in unserem Beispiel durch die Einzahlungs- und Auszahlungstransaktionen, die sich durch Schreibsperren auf dem Summenfeld gegenseitig immer wieder blockieren. Wir können die Situation folgendermaßen deutlich verbessern: Wir führen die speziellen atomaren Operationen ein:

Add(x, *wert*) Addiere *wert* zu x
Sub(x, *wert*) Subtrahiere *wert* von x

Atomar heißt, dass diese Operationen nicht durch andere Operationen unterbrochen werden können. Die beiden Operationen sind kommutativ, d.h. die Reihenfolge ist vertauschbar, ohne dass sich das Ergebnis verändert: Add(x,a), Sub(x,b) hat denselben Effekt wie Sub(x,b), Add(x,a). Da die beiden Operationen atomar und kommutativ sind, können sie schwächere Sperren setzen als normale Schreiboperationen. Dank der schwächeren Sperren können nun Transaktionen gleichzeitig arbeiten in Situationen, in welchen normale Schreibsperren zu Wartezuständen führen würden.

Wir führen für Add und Sub die Sperren *Add-Lock* und *Sub-Lock* ein mit der folgenden Eigenschaft: Ist Add-Lock gesetzt, so kann dennoch einer anderen Transaktion ein Add- oder Sub-Lock gewährt werden. Das heißt, es können mehrere Ein- oder Auszahlungstransaktionen gleichzeitig einen Add- oder Sub-Lock auf einem Objekt besitzen und damit gleichzeitig (d.h. eine atomare Operation unmittelbar nach der anderen) auf demselben Objekt arbeiten; vergleiche Bild 4.4.

Add-Lock und Sub-Lock sind jedoch nicht kompatibel zu S- und X-Locks. Für einen S-Lock etwa ist dies deshalb klar, weil eine Leseoperation natürlich einen anderen Wert liefert, je nachdem, ob sie vor oder nach einem Add ausgeführt wird. Es kann also zu einem Add- oder Sub-Lock nicht gleichzeitig ein S-Lock auf das Objekt gesetzt werden. Dasselbe gilt für X-Locks. Wichtig ist jedoch, dass Konflikte jetzt nur noch dann auftreten, wenn Transaktionen ausdrücklich nicht mit den häufigen Operationen Add und Sub zugreifen wollen, sondern wegen seltener anderer Operationen S- oder X-Locks benötigen.

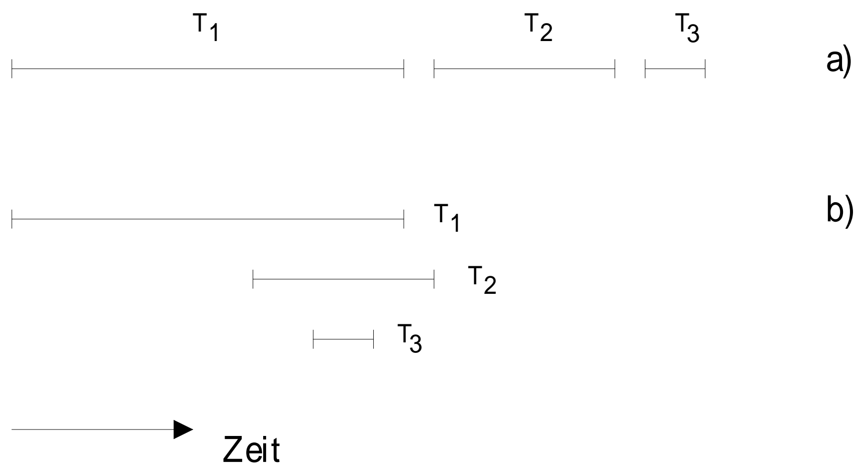

Bild 4.4: Belegung eines Objektes mit Sperren bei
(a) X-Locks
(b) Add- und Sub-Locks

Übung 4.3:

Geben Sie ein Beispiel dafür, wann eine Transaktion zum Zugriff auf solche Summenfelder S- oder X-Locks benötigt.

4.5.4 Sperreinheit

Der erreichbare Grad an Parallelität, aber auch die Kosten für die Synchronisation, hängen wesentlich von der *Granularität* (Größe) der Sperreinheiten ab. Wir haben bisher immer von Objekten gesprochen, die gesperrt werden können. Die Frage ist, welches diese Objekte sein sollen. Sperrobjekte in einer Datenbank könnten sein: logische Einheiten, wie Attribut, Satz (Tupel), Satztyp (Relation), ganze Datenbank; oder physische Einheiten wie Seiten (pages).

Bei der Wahl der Sperreinheit sind zwei gegenläufige Aspekte zu beachten:

(1) je feiner die Sperreinheiten, desto mehr Parallelität zwischen Transaktionen ist möglich;

(2) je feiner die Sperreinheiten, desto größer ist der Verwaltungsaufwand für die Sperren.

Trend (2) ergibt sich daraus, dass pro Transaktion bei höherer Feinheit mehr Sperren zu verwalten und mehr LOCK- und UNLOCK-Operationen auszuführen sind.

Eine übliche Sperrebene ist die der Sätze (Tupel). Auf physischer Ebene kann es darüber hinaus notwendig werden, zusätzlich zu den Sperren auf logischer Ebene kurzzeitig Sperren auf physische Objekte zu setzen. Beispielsweise kann das Einfügen eines Satzes dazu führen, dass verschiedene Sätze in einer Seite verschoben werden müssen. Für die Dauer einer solchen Operation „Reorganisation einer Seite“ muss die ganze Seite gesperrt sein.

Granularität

Tatsächlich müsste die Wahl der Sperreinheit natürlich von der jeweiligen Anwendung abhängen, da Transaktionen ganz unterschiedliche Datenanforderungen und damit Sperranforderungen haben. Eine Transaktion, die viele Tupel einer Relation verändern möchte, sollte in der Lage sein, einfach die ganze Relation zu sperren; umgekehrt sollte eine Transaktion, die nur auf ein einziges Tupel zugreift, auch nur dieses Tupel sperren. Sind auf diese Weise Sperren auf unterschiedlichen Ebenen setzbar, so spricht man von einer *Sperrhierarchie*.

Ein Beispiel für eine Sperrhierarchie ist das Folgende:

- Datenbank
- Area (eine Datenbank ist oft in größere disjunkte Bereiche, sog. *Areas*, eingeteilt)
- Relation (Datei)
- Tupel (Satz)

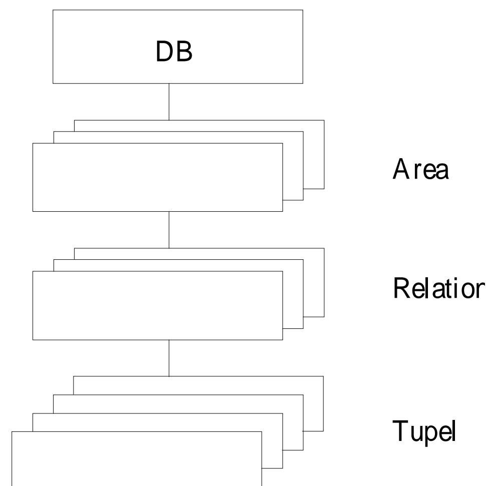

Bild 4.5: Eine Sperrhierarchie

Eine S-Sperre auf die Relation R bedeutet, dass die ganze Relation für schreibenden Zugriff gesperrt ist. Auf die einzelnen Tupel von R braucht jetzt keine Sperre mehr gesetzt zu werden.

Sperrhierarchien sind komplizierter zu verwalten als Sperren auf nur einer Ebene. Im letzteren Fall sind die Sperrobjekte alle voneinander verschieden und können ganz unabhängig voneinander betrachtet werden. Im Falle der Sperrhierarchie können sich Sperrobjekte überlappen. Die Entscheidung, ob ein Objekt O gesperrt werden kann, hängt in einer Sperrhierarchie nicht nur davon ab, ob dieses Objekt schon gesperrt ist, sondern auch davon, ob ein Objekt, das in O enthalten ist oder das O enthält, für eine andere Transaktion gesperrt ist.

Die grundsätzliche Lösung zu diesem Problem ist die Folgende: Wird eine Sperre auf ein Objekt O gesetzt, so wird eine *Intentions-Sperre* auf alle übergeordneten Objekte gesetzt. Damit ist es möglich, auf jeder Ebene zu sehen, ob irgendwo auf einer tieferen Ebene ein Objekt gesperrt ist. Ist auf einer tieferen Ebene ein Objekt gesperrt, so darf das Objekt auf der höheren Ebene nicht gesperrt werden.

Das Sperrprotokoll verlangt also, dass beim Sperren eines Objektes zunächst alle übergeordneten Objekte mit Intentions-Sperren belegt werden. Das zu sperrende Objekt selbst wird mit der tatsächlich benötigten Sperre versehen. Entsprechend den Sperr-Modi Lesen (S) und Schreiben (X) werden zumindest die Intentions-Sperren IS (intention shared) und IX (intention exklusive) benötigt. Mit IS erhält die Transaktion das Recht, auf einer tieferliegenden Ebene Objekte zu lesen; sie darf dort IS- und S-Sperren setzen. Mit IX erhält die Transaktion entsprechend das Recht, auf der tieferen Ebene IS-, S, IX- und X-Sperren zu setzen.

Mittels hierarchischer Sperren kann, allerdings auf relativ grobe Weise, das Phantom-Problem gelöst werden: Sollen Phantome auf der Ebene der Tupel vermieden werden, so muss die Transaktion einfach die entsprechende Relation als Ganzes sperren.