Kapitel 11: Transaktionsverwaltung – Concurrency–Control –

Transaktionskonzept:

Eine *Transaktion* ist eine Folge von (Lese– und Änderungs–) Operationen auf einer oder mehreren Datenbanken, für die die zugrundeliegenden Datenbanksysteme die folgenden Eigenschaften (häufig *ACID–Eigenschaften* genannt) garantieren:

- *Atomarität (engl.: atomicity):*
  Die Änderungen einer Transaktion werden entweder vollständig oder gar nicht durchgeführt (Alles–oder–nichts–Eigenschaft). Im Fehlerfall oder auf "Wunsch" eines Programms werden bereits durchgeführte Änderungen einer Transaktion rückgängig gemacht.

- *Dauerhaftigkeit / Persistenz (engl.: durability / persistence):*
  Die Änderungen einer abgeschlossenen Transaktion gehen auch im Fall von Fehlern der Hardware oder Systemsoftware nicht verloren.

- *Isolation (engl.: isolation):*
  Die Transaktion sieht die Datenbanken so, als gäbe es keine parallelen Transaktionen, und die Änderungen der Transaktion werden bis zum Abschluß der Transaktion vor parallelen Transaktionen verborgen (Äquivalenz zum Einbenutzerbetrieb).

- *Integritätserhaltung (engl.: consistency–preservation, integrity–preservation):*
  Die Transaktion transformiert die Datenbanken von einem – bezüglich der spezifizierten Integritätsbedingungen – konsistenten Zustand in einen anderen konsistenten Zustand.

Die Eigenschaften der Atomarität und Persistenz bedeuten, daß Datenbankanwendungsprogramme größtenteils so geschrieben werden können, als gäbe es keine Fehler der Hardware oder Systemsoftware. Die Transaktionsverwaltung eines DBS stellt mit der **Recovery–Komponente** weitreichende Fehlertoleranzmaßnahmen zur Verfügung, und zwar transparent für die Anwendungsentwicklung (und natürlich erst recht transparent für den Endbenutzer).

Die Eigenschaft der Isolation bedeutet, daß Datenbankanwendungsprogramme so geschrieben werden können, als würde das DBS (ggf. sogar ein verteiltes DBS oder mehrere DBS in einem verteilten System) im Einbenutzerbetrieb arbeiten, obwohl ein DBS in der Regel aus Leistungsgründen effektiv im Mehrbenutzerbetrieb arbeitet. Die Transaktionsverwaltung eines DBS stellt mit der **Concurrency–Control–Komponente** automatische Synchronisationsmaßnahmen zur Verfügung, und zwar transparent für die Anwendungsentwicklung (und natürlich erst recht für den Endbenutzer).

Die Eigenschaft der Integritätserhaltung wird durch die Transaktionsverwaltung nur unterstützt, nicht aber voll erzwungen. Sie kann nur in Verbindung mit entsprechenden Maßnahmen der Anwendungsentwicklung erzwungen werden. Es wird erwartet, daß jedes Anwendungsprogramm bei Beendigung einer Transaktion entsprechende Integritätsprüfungen durchführt oder anstößt (z.B. über Trigger). Falls dann eine Transaktion im fehlerlosen Einbenutzerbetrieb integritätserhaltend ist, garantiert die Transaktionsverwaltung, daß die Integritätsbewahrung auch im Mehrbenutzerbetrieb und bei Auftreten von Hardware– und Systemsoftwarefehlern gilt.

Transaktionsprogramme:

Transaktionen werden dynamisch durch entsprechende DBS−Aufrufe des Anwendungsprogramms erzeugt. In SQL gibt es dafür drei Anweisungen:

- CONNECT ... (BOT = Begin−of−Transaction):
  kennzeichnet den Beginn einer Transaktion.
- COMMIT WORK (EOT = End−of−Transaction):
  kennzeichnet das (”erfolgreiche”) Ende einer Transaktion
  und erklärt alle Änderungen der Transaktion für gültig.
- ROLLBACK WORK (RBT = Rollback−Transaction):
  kennzeichnet das (”erfolglose”) Ende einer Transaktion
  und erklärt alle Änderungen der Transaktion für ungültig.

Implizit kennzeichnen EOT und RBT außerdem den Beginn einer neuen Transaktion, die bis zum nächsten EOT, RBT oder DISCONNECT dauert.

Beispiel eines Transaktionsprogramms (in Pseudocode−Notation)**:**

Datenbankschema:
Konto (Kontonr, Inhaber, Adresse, Stand, Geschäftsstelle)
Buchung (Kontonr, Datum, Zeit, Betrag, Text)

```
    MODULE Transfer;
    ...
    EXEC SQL BEGIN DECLARE SECTION;
      VAR    K1, K2: INTEGER;
             S1, S2, DM: REAL;
    EXEC SQL END DECLARE SECTION;

1   get (K1, K2, DM) from terminal;
2   EXEC SQL CONNECT ...;
3   EXEC SQL SELECT Stand INTO :S1 FROM Konto WHERE Kontonr = :K1;
4   S1 := S1 − DM;
5   EXEC SQL UPDATE Konto SET Stand = :S1 WHERE Kontonr = :K1;
6   EXEC SQL SELECT Stand INTO :S2 FROM Konto WHERE Kontonr = :K2;
7   S2 := S2 + DM;
8   EXEC SQL UPDATE Konto SET Stand = :S2 WHERE Kontonr = :K2;
9   EXEC SQL INSERT INTO Buchung VALUES (:K1, today, now, − :DM, 'Überweisung');
10  EXEC SQL INSERT INTO Buchung VALUES (:K2, today, now, :DM, 'Überweisung');
11  IF S1 < 0 THEN
12     EXEC SQL ROLLBACK WORK;
13     send 'Transfer gescheitert' to terminal;
14  ELSE
15     EXEC SQL COMMIT WORK;
16     send 'Transfer erfolgt' to terminal;
17  END (*IF*);

    END Transfer.
```

11.1 Probleme bei unkontrollierter (Quasi−) Parallelität

Die (quasi−) parallele Ausführung von Transaktionen ist wünschenswert, um möglichst gute Antwortzeiten zu erzielen. Dazu müssen Plattenzugriffe und CPU−Arbeit für verschiedene Transaktionen parallel erfolgen. Um aber auch korrekte Resultate zu gewährleisten, muß die Parallelität u.U. wieder eingeschränkt werden.

Problemtyp "Verlorene Änderung" ("Lost Update")

Prozeß 1: Transfer von a nach c	*Prozeß 2: Transfer von b nach c*
...	...
(* K2 von Prozeß 1 = c *) SELECT Stand INTO :S2 FROM Konto WHERE Kontonr = :K2; (* S2 von Prozeß 1 = 1000.00 DM *)	
	(* K2 von Prozeß 2 = c *) SELECT Stand INTO :S2 FROM Konto WHERE Kontonr = :K2; (* S2 von Prozeß 2 = 1000.00 DM *)
	S2 := S2 + 100.00; (* S2 von Prozeß 2 = 1100.00 DM *) UPDATE Konto SET Stand = :S2 WHERE Kontonr = :K2;
S2 := S2 + 100.00; (* S2 von Prozeß 1 = 1100.00 DM *) UPDATE Konto SET Stand = :S2 WHERE Kontonr = :K2;	
...	...

Abstrakte Darstellung des (quasi−) parallelen Ablaufs als *Schedule*
(Reihenfolge der Read− und Write−Operationen auf den Tupeln oder Seiten der DB):
R1 (a) W1 (a) R2 (b) W2 (b) R1 (c) R2 (c) W2 (c) W1 (c)

Problemtyp "Inkonsistentes Lesen"

Prozeß 1: Transfer von a nach b	*Prozeß 2: Prüfen der Summe von a und b*
...	...
(* a.Stand = 1000.00 DM *) UPDATE Konto SET Stand = Stand − 3000.00 WHERE Kontonr = a (* a.Stand = − 2000.00 DM *)	
	SELECT Stand INTO :A FROM Konto WHERE Kontonr = a (* A = − 2000.00 DM *)
	SELECT Stand INTO :B FROM Konto WHERE Kontonr = b (* B = 1000.00 DM *)
(* b.Stand = 1000.00 DM *) UPDATE Konto SET Stand = Stand + 3000.00 WHERE Kontonr = b (* b.Stand = 4000.00 DM *)	
...	...

Abstrakte Darstellung des (quasi−) parallelen Ablaufs als *Schedule*:
R1 (a) W1 (a) R2 (a) R2 (b) R1 (b) W1 (b)

Problemtyp ”Phantomproblem” (Spezialfall des inkonsistenten Lesens)

Datenbankschema:
Abteilungen (AbtNr, AbtName, Budget, ...)
Angestellte (AngNr, Name, Adresse, Gehalt, ..., AbtNr)
Integritätsbedingung:
Abteilungen.Budget ≤ 200 000.00 * (SELECT COUNT(*) FROM Angestellte
WHERE Angestellte.AbtNr = Abteilungen.AbtNr)

Prozeß 1: Finanzprüfung	*Prozeß 2: Einstellung eines Mitarbeiters*
...	...
SELECT COUNT(*) FROM Angestellte WHERE AbtNr = 1 (* Resultat: 10 Angestellte *)	
	INSERT INTO Angestellte VALUES (111, ’Hans Meier’, ..., 1)
	UPDATE Abteilungen SET Budget = Budget + 200 000.00 WHERE AbtNr = 1
SELECT Budget FROM Abteilungen WHERE AbtNr = 1 (* Resultat: 2 200 000.00 DM *)	
...	...

Zusätzliches Problem mit der Semaphorlösung:
− Semaphore auf Tupeln alleine genügen nicht.

Problemtyp ”Domino−Effekt” (Kaskadierendes Rücksetzen)

Prozeß 1: Einzahlung auf a	*Prozeß 2: Überweisung von a nach b*
...	...
(* a.Stand = 0.00 DM *) UPDATE Konto SET Stand = Stand + 100.00 WHERE Kontonr = a (* a.Stand = 100.00 DM *)	
	UPDATE Konto SET Stand = Stand − 100.00 WHERE Kontonr = a (* a.Stand = 0.00 DM *)
	(* b.Stand = 0.00 DM *) UPDATE Konto SET Stand = Stand + 100.00 WHERE Kontonr = b (* b.Stand = 100.00 DM *)
	COMMIT WORK
ROLLBACK WORK	
...	...

Lösungsansätze mit Semaphoren (durch die Anwendungsprogramme)

Variante 1: ein globaler Semaphor für die Datenbank
(für das Beispiel des Problems des inkonsistenten Lesens)

Prozeß 1: Transfer von a nach b	*Prozeß 2: Prüfen der Summe von a und b (* Integritätsbedingung: a + b ≥ 0.0 *)*
VAR mutex: shared semaphore;	
...	...
P (mutex) (* Resultat: ok *) UPDATE Konto SET Stand = Stand − 3000.00 WHERE Kontonr = a	
	P (mutex) (* Resultat: Prozeß muß warten *)
UPDATE Konto SET Stand = Stand + 3000.00 WHERE Kontonr = b **V (mutex)**	|
	(* Prozeß wird fortgesetzt *) SELECT Stand INTO :A FROM Konto WHERE Kontonr = a SELECT Stand INTO :B FROM Konto WHERE Kontonr = b **V (mutex)**
...	...

Probleme:
- Der globale Semaphor verhindert auch eine parallele Ausführung beispielsweise eines Transfers von a nach b und einer Prüfung von c und d.
- Es ist unklar, wann der Semaphor jeweils frühestens freigegeben werden darf. Vermutlich hängt dies von der jeweiligen Programmlogik und den spezifizierten Integritätsbedingungen ab.

Variante 2: ein Semaphor für jedes Tupel (oder jede Seite)
(für das Beispiel des Problems des inkonsistenten Lesens)

Prozeß 1: Transfer von a nach b | *Prozeß 2: Prüfen der Summe von a und b*

```
VAR mutex_a: shared semaphore;
    mutex_b: shared semaphore;
    ... ;
```

```
...                                              ...

P (mutex_a) (* Resultat: ok *)
UPDATE Konto SET Stand = Stand − 3000.00
WHERE Kontonr = a

                                                 P (mutex_a) (* Resultat: Prozeß muß warten *)

P (mutex_b) (* Resultat: ok *)                          |
UPDATE Konto SET Stand = Stand + 3000.00                |
WHERE Kontonr = b                                       |
V (mutex_a); V (mutex_b);                               |

                                                 (* Prozeß wird fortgesetzt *)
                                                 SELECT Stand INTO :A FROM Konto
                                                 WHERE Kontonr = a

                                                 P (mutex_b) (* Resultat: ok *)
                                                 SELECT Stand INTO :B FROM Konto
                                                 WHERE Kontonr = b
                                                 V (mutex_a); V (mutex_b);

...                                              ...
```

Probleme:
- Die Programmierung der Synchronisationsmaßnahmen erschwert die Anwendungs−entwicklung und dürfte die meisten Anwendungsprogrammierer überfordern.
- Es ist unklar, wann jeweils ein Semaphor frühestens freigegeben werden darf.
- Es ist nicht einfach ersichtlich, ob es durch die Verwendung mehrerer Semaphore nicht womöglich zu Deadlocks (Verklemmungen) kommen kann. Falls es dazu kommt, ist unklar, wie man solche Situationen erkennt und behebt.
- Die Verwaltung sehr vieler Semaphore (z.B. als Unix−Semaphore) ist sehr aufwendig.
- Das Phantom−Problem bleibt ungelöst.

Konsequenz:
Das DBS sollte die notwendigen Semaphore und die P− und V−Operationen darauf selbst automatisch generieren!

11.2 Korrektheitskriterium für den Mehrbenutzerbetrieb

Intuitives Kriterium:

Es dürfen nur (quasi−) parallele Abläufe von Transaktionen zugelassen werden, die zu einer sequentiellen Ausführung der mit Commit beendeten Transaktionen äquivalent sind. Dadurch werden alle Anomalien wie ”verlorene Änderungen”, ”inkonsistentes Lesen”, usw. vermieden.

Formalisierung des intutitiven Korrektheitskriteriums:

- Modellbildung für (quasi−) parallele Abläufe von Transaktionen
- Definition der Äquivalenz von Abläufen
- Entwicklung (beweisbar) korrekter Concurrency−Control−Verfahren zur automatischen Synchronisation durch das DBS

Begriffsbildung und Definitionen:

Einschränkung für den Rahmen der Vorlesung:
Es wird angenommen, daß stets alle Transaktionen mit Commit beendet werden.
→ für den allgemeinen Fall siehe Theorie der Rücksetzbarkeit und verwandte Theorien

Ein **Schedule** s ist ein Tripel (T, A, <) mit:
- T ist eine Menge von Transaktionen (eigentlich nur Transaktionsnummern).
- A ist eine Menge von (Datenbankzugriffs−) Aktionen der Form Ri (x) oder Wi (x), wobei
  - R und W für Lesezugriffe (Reads) und Schreibzugriffe (Writes) stehen,
  - x ein Objekt der Datenbank ist, auf das sich der Zugriff bezieht (z.B. eine Seite) und
  - jede Aktion zu genau einer Transaktion Ti gehört.
- < ist eine partielle Ordnung auf der Aktionenmenge A, wobei für jedes Aktionenpaar $<a, b> \in A \times A$ gelten soll:
  Wenn a und b auf dasselbe Datenbankobjekt zugreifen und a oder b ein Schreibzugriff ist, dann müssen a und b bezüglich < geordnet sein; es muß also $(a < b) \vee (b < a)$ gelten.

Zwei Aktionen $a, b \in A$ in einem Schedule $s = (T, A, <)$ sind in **Konflikt**, wenn
- a und b auf dasselbe Datenbankobjekt zugreifen und a oder b ein Schreibzugriff ist und
- a und b zu verschiedenen Transaktionen gehören.

Die beiden Aktionen a und b werden auch als ”Konfliktpaar” bezeichnet.

Bemerkungen:
- Es ist wichtig, daß die Aktionen eines Konfliktpaars geordnet sind, damit überhaupt eine Aussage über deren Effekt möglich ist.
- Aktionen, die nicht in Konflikt sind, können innerhalb eines Schedules in beliebiger Reihenfolge ausgeführt werden, ohne daß dadurch unterschiedliche Effekte entstehen.
  Mit anderen Worten:
  Um zu entscheiden, ob zwei Schedules bezüglich ihrer Effekte äquivalent sind, ist nur die Reihenfolge der Aktionen in den Konfliktpaaren von Bedeutung.

Ein Schedule $s = (T, A, <)$ heißt **seriell**, wenn für je zwei Transaktionen Ti, $Tj \in T$ gilt, daß entweder alle Aktionen von Ti bezüglich < vor allen Aktionen von Tj liegen oder alle Aktionen von Tj vor allen Aktionen von Ti.
In einem seriellen Schedule gibt es somit keine (quasi−) parallelen Transaktionen.

Beispiele:
(Es wird in den Beispielen nur A angegeben; T steckt implizit in den Indizes der Aktionen, und < ist implizit durch die Anordnung der Aktionen von links nach rechts gegeben.
In den Beispielen ist < immer eine totale Ordnung.)

Schedule s1: R1 (a) W1 (a) R2 (a) R2 (b) R1 (b) W1 (b)

Schedule s2: R1 (a) W1 (a) R2 (a) R3 (b) R2 (b) W2 (b) R3 (c) W3 (c) R1 (c)

Schedule s2': R3 (b) R3 (c) W3 (c) R1 (a) W1 (a) R1 (c) R2 (a) R2 (b) W2 (b)

Konfliktpaare für s1: <W1(a), R2(a)>, <R2(b), W1(b)>

Konfliktpaare für s2: <W1(a), R2(a)>, <R3(b), W2(b)>, <W3(c), R1(c)>

Konfliktpaare für s2': <W3(c), R1(c)>, <R3(b), W2(b)>, <W1(a), R2(a)>

Definition der Äquivalenz von Schedules:

Zwei Schedules s und s' sind **(konflikt−) äquivalent**, wenn sie dieselben Transaktionsmengen und Aktionsmengen haben und in beiden Schedules dieselben Konfliktpaare (jeweils mit derselben Aktionsreihenfolge) vorliegen.

Definition der Korrektheit von nichtseriellen Schedules:

Ein Schedule s ist **(konflikt−) serialisierbar**, wenn er zu einem seriellen Schedule äquivalent ist.

Bemerkungen:
- Serialisierbarkeit ist die Formalisierung des intuitiven Kriteriums der Äquivalenz zum Einbenutzerbetrieb.
- Damit Serialisierbarkeit als Korrektheitskriterium Sinn macht, sind folgende (nicht beweisbare) Annahmen implizit zugrundegelegt:
  - Ein serieller Schedule ist korrekt (unter der wiederum impliziten Annahme, daß das Transaktionsprogramm an sich korrekt, insbesondere integritätserhaltend ist.
  - Es wird angenommen, daß alle Transaktionen von verschiedenen, unabhängigen Benutzern stammen und daß für diese Benutzer jede beliebige sequentielle Ausführung der Transaktionen akzeptabel ist.

Algorithmische Überprüfung der Serialisierbarkeit eines Schedules:

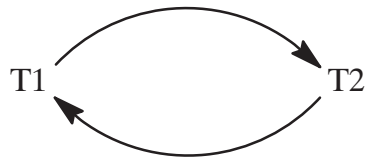
Sei s = (T, A, <) ein Schedule. Der **Abhängigkeitsgraph** für diesen Schedule ist ein gerichteter Graph mit
- Transaktionen als Knoten und
- einer Kante zwischen Ti und Tj genau dann, wenn es ein Konfliktpaar <a, b> in s gibt, so daß a zu Ti und b zu Tj gehört und a < b gilt.

Satz:
Ein Schedule ist serialisierbar genau dann, wenn sein Abhängigkeitsgraph azyklisch ist, d.h. keine Zyklen enthält.

Man erhält einen zum Schedule äquivalenten seriellen Schedule − eine **Serialisierung** − durch topologische Sortierung des Abhängigkeitsgraphen
(eine totale Ordnung der Transaktionen, die die durch die Kanten des Graphen gegebene partielle Ordnung beinhaltet):
- Man wähle eine beliebige Quelle des Graphen (d.h. einen Knoten ohne eingehende Kanten) und entferne diese sowie alle davon ausgehenden Kanten.
- Falls der Graph noch Knoten hat,
  wiederholt man diese Prozedur für den verbleibenden Graphen.

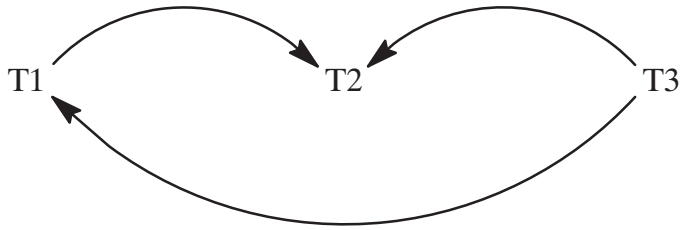
Beispiele:

Abhängigkeitsgraph für s1:



hat einen Zyklus
⇒ s1 ist nicht serialisierbar

Abhängigkeitsgraph für s2:



azyklisch
⇒ s2 ist serialisierbar

topologische Sortierung: T3 T1 T2

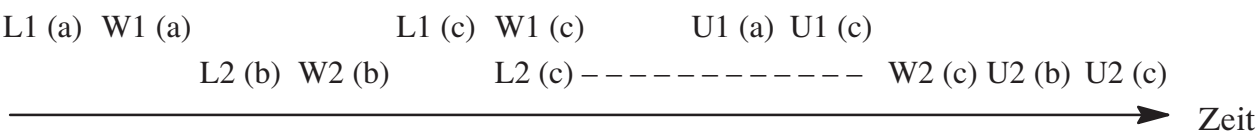
11.3 Concurrency−Control mit Sperrprotokollen

Das DBS fordert für eine Transaktion vor dem Zugriff auf ein Objekt eine Sperre auf dem Objekt an. Sperren sind im Prinzip Semaphore, die durch das DBS selbst implementiert werden. Dazu stellt die Concurrency−Control−Komponente die folgenden zwei Operationen für Sperren zur Verfügung:

- Lock (Transaktion Ti, Objekt x, Modus m) und
- Unlock (Transaktion Ti, Objekt x).

Das DBS kann nie zwei Transaktionen gleichzeitig eine Sperre auf demselben Objekt (in unverträglichen Sperrmodi) geben. Im Sperrkonfliktfall muß die "spätere" Transaktion bis zur Sperrfreigabe warten. Die Concurrency−Control−Komponente legt in diesem Fall die Transaktion schlafen und weckt sie bei Sperrfreigabe automatisch wieder auf. Zu diesem Zweck verwaltet die Concurrency−Control verschiedene Warteschlangen.

Beispiel:
(mit der Notation Li (x) für Lock (Ti, x, Exclusive) und Ui (x) für Unlock (Ti, x)**:**



Zweiphasen−Sperrprotokoll (Two−Phase Locking, 2PL):

Regel 1: Bevor eine Transaktion auf ein Objekt zugreift, muß das Objekt für die Transaktion gesperrt werden ("Wachstumsphase" der Transaktion).

Regel 2: Nachdem eine Transaktion eine Sperre freigegeben hat, darf sie keine weiteren Sperren mehr erwerben ("Schrumpfungsphase" der Transaktion).

Satz:
Das Zweiphasen−Sperrprotokoll garantiert, daß alle entstehenden Schedules serialisierbar sind.

Beweis:

Sei s ein Schedule, der unter dem 2PL entstanden ist.
Annahme: s ist nicht serialisierbar.
⇒ es gibt einen Zyklus im Abhängigkeitsgraphen
$T1 \rightarrow T2 \rightarrow T3 \rightarrow ... \rightarrow Tk \rightarrow T1$,
also Konfliktpaare
$<a1(x1), a2(x1)>, <a2(x2), a3(x2)>, ..., <ak(xk), a1(xk)>$
Da Objekte vor dem Zugriff gesperrt werden und andere Transaktionen erst nach der entsprechenden Sperrfreigabe auf das Objekt zugreifen können, gilt bezüglich der Reihenfolge der Lock− und Unlock−Operationen:
$U1(x1) < L2(x1),\ U2(x2) < L3(x2),\ ...,\ Uk(xk) < L1(xk)$
Wegen Regel 2 des 2PL gilt ferner:
$L2(x1) < U2(x2), L3(x2) < U3(x3), ...$
⇒ $U1(x1) < L2(x1) < U2(x2) < L3(x2) < U3(x3) < ... < Uk(xk) < L1(xk)$
Dann aber wäre wegen $U1(x1) < L1(xk)$ die Regel 2 für Transaktion T1 verletzt.
Widerspruch ! ⇒ Behauptung

Beobachtungen:

- Das 2PL ist nicht ausreichend, um Domino−Effekte zu verhindern (siehe Beispiel für Domino−Effekte in Kapitel 11.1).
- Um das 2PL korrekt anwenden zu können, muß man für jede Transaktion wissen, ab welchem Zeitpunkt im Laufe ihrer Ausführung keine weiteren Objekte mehr benötigt werden, die nicht bereits für die Transaktion gesperrt sind. Ab diesem Zeitpunkt können Sperren freigegeben werden.
  Dieses A−priori−Wissen ist unrealistisch, da die von der Transaktion benötigten Objekte von der Ausführungsdynamik des zugrundeliegenden Transaktionsprogramms abhängen.

Striktes Zweiphasen−Sperrprotokoll (Strict Two−Phase Locking, Strict 2PL):

Regel 1: wie beim allgemeinen 2PL

Regel 2: Halte alle jemals für eine Transaktion erworbenen Sperren bis zum Ende der Transaktion (Abschluß des Commit oder Rollback) und gebe sie dann auf einmal frei. (Die Schrumpfungsphase fällt also zu einem einzigen Zeitpunkt zusammen.)

Bemerkungen:

- Das strikte 2PL gewährleistet die Äquivalenz (in einem weitergehenden Sinne) eines parallelen Ablaufs von Transaktionen (die mit Commit oder Rollback beendet sind) zu einem sequentiellen Ablauf derjenigen Transaktionen, die mit Commit beendet sind (ohne Beweis). Das strikte 2PL verhindert also auch Domino−Effekte.
- Das strikte 2PL ist als Concurrency−Control−Protokoll in praktisch allen kommerziellen DBS implementiert.
- Das 2PL kann auf verschiedenen Sperrgranulaten implementiert werden:
  - Seitensperren (in den meisten DBS)
  - Relationssperren oder Tablespace−Sperren (sehr grob und damit restriktiv)
  - Tupelsperren und Sperren auf Indexeinträgen
    (sehr feines Granulat und daher bezüglich der Mehrbenutzerleistung am besten, aber auch sehr komplex und daher nur in den besseren DBS implementiert)
  - Kombinationen (die meisten Transaktionen sperren z.B. Seiten, einige lange Transaktionen aber sperren ganze Relationen) → Mehrgranulats−Sperrverfahren
- Das 2PL kann mit verschiedenen Sperrmodi realisiert werden. In der Regel sind die Sperrmodi
  Exclusive − für Writes − und *Shared* − für Reads −
  vorgesehen.
  Der Sperrmodus für einen Zugriff wird jeweils vom DBS automatisch gewählt.
  Für Sperrmodi muß eine *Kompatibilitätsmatrix (Verträglichkeitsmatrix)* spezifiziert sein, die festlegt, ob eine Sperranforderung im Modus m gewährt werden darf, wenn das zu sperrende Objekt bereits für eine andere Transaktion im Modus k gesperrt ist.

  Beispiel einer Kompatibilitätsmatrix:

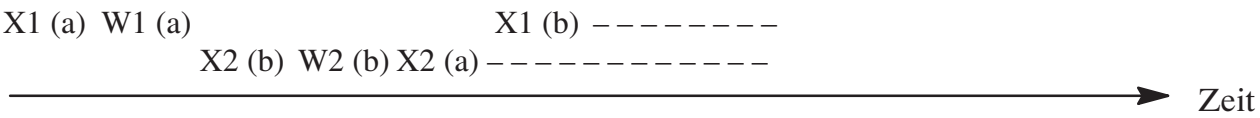
		Transaktion fordert eine Sperre an im Modus	
		Shared	*Exclusive*
Objekt ist gesperrt im Modus	*Shared*	+	−
	Exclusive	−	−

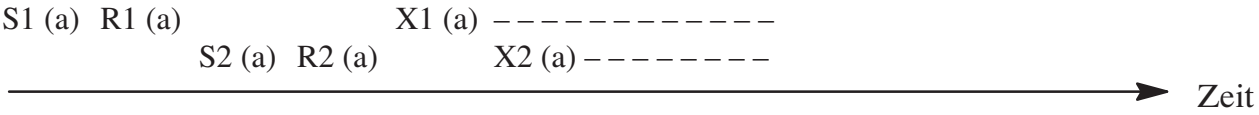
Deadlocks (Verklemmungen):

Das 2PL kann zu Deadlocks führen, bei denen zwei oder mehr Transaktionen in einem Zyklus wechselseitig auf die Freigabe von Sperren warten.
Die Concurrency−Control−Komponente erkennt solche Deadlocks, indem sie über alle Wartebeziehungen in Form eines Wartegraphen Buch führt und − periodisch oder bei jedem Sperrkonflikt − einen Zyklustest für den Wartegraphen durchführt. Falls ein Zyklus gefunden wird, wird eine der am Zyklus beteiligten Transaktionen als Deadlock−Opfer ausgewählt und zurückgesetzt (von der Recovery−Komponente des DBS).

Das Anwendungsprogramm erhält in diesem Fall einen speziellen SQLCODE (−60 in Oracle) für die SQL−Anweisung, bei der der Deadlock entstand. Das Programm sollte in einem solchen Fall selbst den erneuten Start der Transaktion veranlassen (ggf. nach erneuten Initialisierungen), damit der Deadlock und seine Behandlung gegenüber dem Endbenutzer transparent ist.

Beispiele:
(mit der Notation Xi (x) für Lock (Ti, x, Exclusive) und Si (x) für Lock (Ti, x, Shared))





Der zweite Fall − Deadlock aufgrund von Sperrkonversionen (Verschärfung einer Shared−Sperre in eine Exclusive−Sperre) − tritt in der Praxis potentiell häufig auf.
Er kann durch zusätzliche Sperrmodi weitgehend vermieden werden. Diese erfordern aber u.U. spezielle Hinweise durch das Programm (die der Programmierer vorsehen muß).
Aus diesem Grund hat SQL spezielle Anweisungszusätze wie z.B.
DECLARE C CURSOR FOR SELECT ... *FOR UPDATE* ...

Ergänzende Literatur zu Kapitel 10:

P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison−Wesley, 1987

G. Weikum, Transaktionen in Datenbanksystemen, Addison−Wesley, 1988

J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

C.H. Papadimitriou, The Theory of Concurrency Control, Computer Science Press, 1986

N. Lynch, M. Merritt, W. Weihl, A. Fekete, Atomic Transactions, Morgan Kaufmann, 1994

A.J. Bernstein, P.M. Lewis, Concurrency in Programming and Database Systems, Jones and Bartlett Publishers, 1993

J. Bacon, Concurrent Systems, Addison−Wesley, 1993