4 CONCURRENCY

4.4 Einleitung

Probleme mit der Concurrency (engl. Gleichzeitigkeit) in Datenbanksystemen treten immer dann auf, wenn „gleichzeitig“ mehrere Benutzer oder Prozesse auf die Datenbank zugreifen können. Wird dabei auf die gleichen Daten-Items zugegriffen, so muss dies vom Datenbankmanagementsystem (DBMS) kontrolliert werden. Fehlt ein geeigneter Kontrollmechanismus, so können inkonsistente Zustände in Datenbanksystemen auftreten.

4.5 Transaktionen

Untrennbar mit dem Begriff Concurrency sind weitere Begriffe verbunden: die Transaktion, die Logical Unit of Work (LUW) und das Locking:

Problem „Fehlende Gegenbuchung“

Betrachten wir ein Beispiel aus dem Bankbereich – eine Umbuchung eines bestimmten Betrages von einem Konto auf ein anderes. Die zwei notwendigen SQL-Statements könnten lauten:

```
update Konto                    update Konto
set Saldo = Saldo – 500         set Saldo = Saldo + 500
where KontoNr = 4711            where KontoNr = 0815
```

Abbildung 1: Problem „Fehlende Gegenbuchung“

Wird nur eines der beiden Statements durchgeführt, so „stimmt etwas nicht“ – entweder die Bank verliert Geld oder sie gewinnt ungerechtfertigt welches auf Kosten des Kunden. Den Zustand der dann erreicht wird („wundersame Geldvermehrung“ bzw. „spurloser Geldschwund“) definieren wir als inkonsistent.

Daraus ergibt sich auch die Definition der LUW bzw. der Transaktion, die ja von einem konsistenten Zustand in den anderen führen muss: die Transaktion umfasst jeweils Buchung und Gegenbuchung.

Eine Transaktion ist eine logisch zusammenhängende Sequenz von Operationen (I/O-Befehlen), die eine Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt; man nennt alle diese Operationen zusammen eine Logical Unit of Work (LUW), da sie nur gemeinsam gut gehen oder fehlschlagen können.
Achtung: der o.a. „konsistente Zustand“ gilt per definitionem als erreicht, sobald eine Transaktion abgeschlossen wird (egal ob mit „commit“ oder „rollback“), denn das DBMS kann Konsistenz ja gar nicht überprüfen; diesen Fakten muss das Transaktionsdesign Rechnung tragen.

Wir haben soeben eine Transaktion erfolgreich entworfen (designed); in einem Anwendungsprogramm könnte das dann wie folgt aussehen:

```
exec sql      whenever sqlerror goto UnDo;
...
Betrag = ...
KontoQuelle = ...
KontoZiel = ...
...
exec sql      update Konto
     set Saldo = Saldo – Betrag
     where KontoNr = KontoQuelle
…
exec sql      update Konto
     set Saldo = Saldo + Betrag
     where KontoNr = KontoZiel
...
exec sql      commit work;
...
UnDo:
exec sql      rollback work;
```

Abbildung 2: Konto-Umbuchung

Funktionsweise

Zwischen den verschiedenen Operationen (im Beispiel: zwischen den `update`) wäre die Datenbank nicht konsistent - sie durchwandert also während einer solchen LUW inkonsistente Zwischenzustände. Es darf daher nicht erlaubt werden, dass nur ein Teil der Operationen (nur ein `update`) durchgeführt wird.

Da aber nicht garantiert werden kann, dass bei herkömmlicher Verarbeitung alle Operationen erfolgreich durchgeführt werden (z.B. Stromausfall im ungünstigsten Moment) muss auf andere Mittel zurückgegriffen werden: Systeme, die Transaktionsverarbeitung unterstützten, garantieren, dass Transaktionen entweder „ganz oder gar nicht“ durchgeführt werden.

Die `commit` und `rollback` Operationen sind das „Werkzeug“ für diese Definition: das `commit` signalisiert (und definiert) ein erfolgreiches Ende einer Transaktion - eine LUW wurde erfolgreich beendet und alle Änderungen, die in dieser LUW durchgeführt worden sind, können / müssen fixiert werden.

Im Gegensatz dazu definiert ein `rollback` ein nicht erfolgreiches Ende: das DBMS muss deshalb alle Operationen der noch „offenen“ LUW rückgängig machen.

4.6 Locks

Die meisten DBMS sind Multiuser-Systeme, was bedeutet, dass es erlaubt ist, dass mehrere Transaktionen zur selben Zeit an der gleichen Datenbank ablaufen. In solchen Systemen wird ein Concurrency-Kontrollmechanismus benötigt, damit sich zeitlich parallel ablaufende Transaktionen gegenseitig nicht beeinträchtigen.

Ohne einen solchen Mechanismus können verschiedene Probleme auftreten. Diese typischen Probleme wollen wir nun betrachten:

4.6.1 Problem „Lost Update“

Zeit	Transaktion A		Preis	Transaktion B	
1	Transaktion liest einen bestimmten Wert „Preis“ (100)...	select	100,-	-	-
2	... um ihn zu analysieren und dann ...	-	100,-	select	In der Transaktion wird genau das gleiche Anwendungsprogramm durchgeführt: also Preis lesen (100)...
3	... abhängig von vielen Faktoren zu verändern. In diesem Fall soll es eine Erhöhung um 10% sein.	update	110,-	-	... analysieren ...
4	...	...	120,-	update	... und um 20% erhöhen.
5	...	...	...	...	...

Abbildung 3: Lost Update

Eine Transaktion A holt einen Datensatz R zur Zeit t1. Transaktion B holt den gleichen Datensatz zu einem späteren Zeitpunkt t2. Die Transaktion A schreibt den veränderten Datensatz zur Zeit t3 zurück und die Transaktion B schreibt den gleichen Datensatz gemäß den getätigten Veränderungen basierend auf den Werten, die zum Zeitpunkt t2 gültig waren, zum Zeitpunkt t4 zurück. Damit ist das Update der Transaktion A verlorengegangen.

Obwohl der Preis von ATS 100,- einmal um 10% und einmal um 20% erhöht werden sollte, was zu einem Preis von ATS 132,- führt, steht in der Datenbank ein Preis von 120,-; die Veränderung der Transaktion A ist verloren gegangen!

4.6.1.1 Pessimistische Methode

Die Transaktion B geht von einem Zustand aus, der eigentlich nicht gültig ist, denn A ist ja gerade dabei, den Wert zu verändern. Das DBMS muss also dafür sorgen, dass Transaktion B den Wert nicht lesen darf, solange A in unter Bearbeitung hat – der Wert (Satz, Tabelle, ...) muss für den Zugriff durch andere gesperrt werden!

Dies ist allerdings gar nicht so einfach zu entscheiden: wie soll es denn wissen, ob A nur zur Information liest, oder ob sie vor hat, den Wert zu verändern? Der Anwendungsprogrammierer tut das mit einem speziellen Statement („`for update`“).

4.6.1.2 Optimistische Methode

Neben dieser „pessimistischen Methode“ (man nimmt den schlimmsten Fall an, nämlich, dass einen andere Transaktion den gleichen Wert verändern will), die manchmal wegen eines geringen Kollisionsprozentsatzes nicht optimal scheint, gibt es auch noch eine optimistische Methode (man nimmt an, dass schon nichts passieren wird und reagiert im nachhinein).

Unmittelbar vor dem eigentlichen `update` vergleicht man den ursprünglich gelesenen Wert mit dem Wert der sich in der Datenbank befindet: stimmen diese nicht überein, so hat inzwischen jemand die Daten verändert und die Transaktion ist mit `rollback` zu beenden.

Zum Vergleich besonders gut eignen sich hierfür ein Attribute, dass immer jenen eindeutigen Timestamp enthält, an dem der Satz zuletzt verändert wurde …

4.6.2 Problem "Uncommitted Dependency"

Zeit	Transaktion A		Preis	Transaktion B	
0	-	-	100,-	-	-
1	Die Transaktion verändert den Preis (120) ...	update	120,-	-	-
2	... kommt in einen Ausnahmezustand ...	-	120,-	select	Die Transaktion liest denn Preis (120) ...
3	... und macht die Veränderung wieder rückgängig um in einen konsistenten Zustand zu kommen.	rollback	100,-	-	... um ihn zu analysieren und zu verarbeiten.
4	-	-	100,-	-	-
5	-	-	100,-	... 120 ...	Obwohl inzwischen die Veränderung schon wieder zurückgenommen worden ist, rechnet die Transaktion mit 120 statt mit 100!

Abbildung 4: Uncommitted Dependency

Dieses Problem tritt auf, weil die Transaktion B den Wert / Datensatz / Tabelle verwendet (`select`, `update`), der von der anderen Transaktion A zwar verändert, aber nicht `committ`ed wurde. Ein derartiges „dirty read" (= lesen, vor der Änderungsbestätigung) kann verhindert werden, in dem „uncommited updates" vor anderen Transaktionen verborgen werden.

4.6.3 Problem „Inconsistent Analysis"

Zeit	Transaktion A		Kto 1	Kto 2	Transaktion B	
0	-	-	100,-	200,-	-	-
1	Transaktion summiert alle Kontostände: sie liest Kontostand 1 (100,-) ...	select	100,-	200,-	-	-
2	-	-	100,-	200,-	select	Transaktion bucht 10,- von Konto 2 auf Konto 1: sie liest Kontostand 2 (200) ...
3	-	-	100,-	190,-	update	... verändert ihn, ...
4	-	-	100,-	190,-	select	... liest Kontostand 1 ...
5	-	-	110,-	190,-	update	... und verändert auch diesen.
6	-	-	110,-	190,-	commit	Ein commit bestätigt das Ganze – aus Sicht dieser Transaktion ist alles ok!
7	... und Kontostand 2 (190,-) ...	select	110,-	190,-	-	-
8	... und gibt die falsche (!) Summe (290,-) aus!		110,-	190,-	-	-

Abbildung 5: Inconsistent Analysis

Das Problem besteht auch in diesem Fall darin, dass die Transaktion B Werte / Datensätze / Tabellen verändert, die die Transaktion A gerade analysiert. Beachte, dass das trotz `commit` in Transaktion B passiert!

Zu verhindern ist das wohl nur dann, wenn A bekannt gibt, welche Sätze sie zu lesen gedenkt, und diese gegen Veränderung gesperrt werden. Manchmal ist es dann allerdings gleich besser, die gesamte Tabelle

gegen Veränderungen zu sperren und derartige Auswertungen nicht in der Hauptgeschäftszeit zu machen. In unserem Beispiel würde das ja alle Kontobewegungen der Bank während der Auswertung verhindern …

Anmerkung: Sonderfall ist das **Phantom-Problem**. Hier ändert sich während der Lese-Transaktion die Menge der zu lesenden Objekte durch INSERT- oder DELETE-Operationen einer parallelen Transaktion.

4.6.4 Funktionsweise

Die Grundidee des Sperren von Objekten (Locking) ist folgende: wenn eine Transaktion A ein Objekt (Tabelle, Datensatz) benötigt und sichergestellt werden soll, dass keine andere Transaktion eine Veränderung an diesem Objekt vornehmen kann, dann wird von der Transaktion A ein Lock für dieses Objekt angemeldet. Damit können andere Transaktionen solange keine Veränderungen an diesem Objekt vornehmen, bis das Objekt von A wieder freigegeben wird (z.B. bei Transaktionsende).

Beachten Sie: es gibt keinen Zugriff auf Objekte der Datenbank ohne Locking!

Grundsätzlich gibt es 2 Arten von Locks:

Exclusive Lock (X Lock):	Falls die Transaktion A den Datensatz R durch einen X Lock sperrt, wird jeder Lock-Versuch einer anderen Transaktion für den gleichen Datensatz abgelehnt - egal, ob die andere Transaktion einen X Lock oder S Lock versucht. Die letztere Transaktion wird in einen Wartezustand versetzt: sie wartet, bis der Datensatz R von A freigegeben wird.
Shared Lock (S Lock):	Falls eine Transaktion A einen S Lock für den Datensatz R aufrechterhält, muß jede andere Transaktion warten, die einen X Lock anfordert - bis A das Objekt freigibt; S Locks werden jedoch gewährt. Es können also mehrere Transaktionen auf ein und dasselbe Objekt einen S Lock unterhalten.

Abbildung 6: Arten von Locks

Folgende Tabelle zeigt, wann ein Lock vom DBMS gewährt werden kann (✓) und wann die Transaktion auf die Freigabe des Lock-Holders warten muss (✗):

Versuch \ Zustand	X Lock	S Lock	unlocked
X Lock	✗	✗	✓
S Lock	✗	✓	✓

Abbildung 7: Lock-Kompatibilitätsmatrix

Aus dieser Matrix ist ersichtlich, dass eine extensive Verwendung von Locks – speziell von X Locks – sehr schlecht für die Performance der Datenbank ist. Es gilt daher, minimal zu sperren: also genau jene Objekte mit genau jenem Lock-Typ, so dass die Konsistenz der Datenbank gerade noch gewährleistet ist.

Bei den meisten DBMS wird die Datensatzsperre automatisch durchgeführt. Eine Lesezugriff führt automatisch zu einem S Lock, ein Update zu einem X Lock. Es gibt also de facto keinen Zugriff ohne einen Lock!

Sperren dürfen nicht ohne weiteres beispielsweise nach einem erfolgreichen Update oder Select freigegeben werden, sondern erst bei Ende der LUW; andernfalls könnte es bei einem notwendigen Rollback Probleme geben.

Dieses Wissen muss unser Transaktions-Design beeinflussen: unnötig lange Transaktionen führen demnach ja zu unnötig langen Locks, die wiederum die Performance unnötigerweise nach unten drücken (wegen des Wartens auf die Freigabe).

Beispiel:
Betrachten wir eine Überweisung von 100,- von Konto A auf ein Konto B:

Variante 1	**Lock A**	**Lock B**
`transaction Ü1 begin`		
`  X-Lock (KtoA);`		
`  X-Lock (KtoB);`	X	
`  if KtoA > 100 then`	X	X
`  begin`		
`  KtoA = KtoA - 100;`	X	X
`  KtoB = KtoB - 100;`	X	X
`  end`		
`  else fehlermeldung ("KtoA überzogen!");`	X	X
`end-transaction;`		

Variante 2	**Lock A**	**Lock B**
`transaction Ü2 begin`		
`  S-Lock (KtoA);`		
`  if KtoA > 100 then`	S	
`  begin`		
`  X-Lock (KtoA);`	S	
`  X-Lock (KtoB);`	X	
`  KtoA = KtoA - 100;`	X	X
`  KtoB = KtoB - 100;`	X	X
`  end`		
`  else fehlermeldung ("KtoA überzogen!");`	X	X
`end-transaction;`		

Abbildung 8: Lock-Varianten

In der Variante 2 brauchen wir deutlich weniger X Locks: sie hat den Vorteil, dass im Falle eines ungenügenden Kontostandes überhaupt kein X Lock verlangt wird, der Nachteil daran ist allerdings, dass mitten in der Transaktion eine Sperränderung verlangt wird, welche unter Umständen (falls auf KtoA wegen einer parallelen Transaktion ein zweiter S Lock liegt) zu einer Unterbrechung des Transaktionsablaufes führen kann.

Welche Strategie zu wählen ist, hängt von den Prioritäten des Anwendungssystems ab. In diesem Fall stellt sich die Frage, wie wichtig Überweisungen sind, wie schnell sie durchgeführt werden sollen und mit welchen anderen Transaktionen sie konkurriert. Die richtige Strategie ist oft, es dem DBMS zu überlassen und erst bei unerwünschtem Verhalten zu versuchen, es zu ändern.

Probleme gelöst?

Wir untersuchen nun, ob die vorhin genannten Probleme durch Locks gelöst wurden:

Lost Update:
- Transaktion A setzt zu Zeitpunkt 1 ein X Lock („for update“) ab, bekommt es und liest den Inhalt.
- Transaktion B beantragt zu ZP 2 ebenfalls ein X Lock, bekommt es aber nicht und wartet daher auf die kritische Ressource.
- Transaktion A verändert den gesperrten Preis zu ZP 3 und beendet die LUW und damit auch den Lock durch ein commit.
- Transaktion B bekommt erst jetzt den X Lock, beendet sein wait, und liest den korrekten Preis.
- ...

Uncommited Dependency:
- Transaktion A beantragt zu ZP 1 einen X Lock („update“), bekommt exklusive Kontrolle, und verändert den Preis.
- Transaktion B versucht zu ZP 2 einen S Lock zu bekommen, scheitert jeduch und wird in einen Wartezustand versetzt.
- Transaktion A beendet den Lock und die LUW mit einem Rollback.
- Transaktion B erhält nun den S Lock und liest den korrekten Preis.
- ...

Inconsistent Analysis:
- Transaktion A bekommt einen S Lock auf Kto 1 und Kto 2 (ZP 1).
- Der Versuch von Transaktion B die Ressourcen exklusiv zu sperren schlägt zu ZP 2 fehl (ist ja schon durch A gesperrt); B wartet ...
- Transaktion A wertet die Konti richtig aus und gibt sie anschließend frei (ZP 3 – 8)
- Erst jetzt darf B weitermachen, bekommt den X Lock und verändert die Kontostände.

Abbildung 9: Probleme gelöst?

Wir sehen, diese Probleme hätten wir gelöst – wir haben uns dadurch aber andere eingehandelt ...

4.7 Deadlocks

Beim Versuch, dieses Problem mit Hilfe von Locks zu lösen, ist ein neues Problem aufgetreten: ein Deadlock.

Hier ein kleines Beispiel:

ZP	Transaktion A		Preis	Transaktion B	
1	SELECT	S	100	-	
2	-	S	100	SELECT	S
3	UPDATE 10%	Anfrage X → warten	110	-	S
4	-	warten	120	UPDATE 20%	Anfrage X → warten
5	…	warten	...	...	warten

Abbildung 10: Deadlock

1: Nehmen wir an, wir können / wollen zu diesem Zeitpunkt noch nicht festlegen, ob wir den Preis später ändern werden, oder nicht – und um möglichst wenig möglichst „gering“ (S Lock vor X Lock) zu sperren, verwenden wir richtigerweise das „for update“ nicht: das Objekt bekommt daher einen S Lock seitens der Transaktion A.

2: Transaktion B verarbeitet die Daten mit dem selben Programm (ist ja nicht so unwahrscheinlich) und verlangt daher ebenfalls nur einen S Lock: sie kriegt ihn, da A ja ebenfalls nur einen S Lock hält.

3: A versucht nun einen X Lock zu bekommen: das wird abgewiesen, da B noch einen S Lock hält. A begibt sich in einen Wartezustand ...sie wartet auf die Freigabe der Ressource seitens B

4: B versucht nun ebenfalls einen X Lock zu bekommen: auch das wird abgewiesen und auch B wartet auf die Freigabe der Ressource seitens A!

Der Deadlock ist perfekt: die beiden Transaktion warten zirkulär auf die Freigabe der Ressource!

Ein Deadlock entsteht, wenn zwei Transaktionen sich gegenseitig sperren und daher zirkulär darauf warten, dass die jeweils andere Transaktion die benötigte Ressource freigibt.

Wir wissen also, wie ein Deadlock zustande kommt – aber was kann man dagegen tun? Habt ihr eine Idee?

Hier einige Strategien, um Deadlocks zu vermeiden, bzw. die Situation zu retten, nachdem einer aufgetreten ist:

4.7.1 Two Phase Locking

Falls alle Transaktionen folgende zwei Regeln einhalten, kann es zu keinem Deadlock kommen:

1. Alle in einer Transaktion benötigten Objekte werden auf einmal gesperrt (gelockt). Gelingt es, so wird die Transaktion durchgeführt.
2. Gelingt es nicht, so werden die bereits gesperrten Objekte wieder freigegeben und der Vorgang zu einem späteren Zeitpunkt (wait) erneut versucht.

Warum funktioniert das (weil wartende Transaktionen keine Ressourcen blockieren)?

Das Problem dabei ist, dass bereits beim Start der Transaktion bekannt sein muss, welche Objekte zu sperren sein werden. In der Praxis ist das oft nicht oder nur eingeschränkt möglich.

Weiters wird bei diesen Überlegungen vorausgesetzt, dass die einzelnen Transaktionen voneinander unabhängig sind; eine Abhängigkeit der Art Transaktion A muss vor Transaktion B laufen, kann damit nicht realisiert werden (vgl. Kapitel „Zeitmarken-Verfahren“ auf Seite 9).

4.7.2 Transaktions-Scheduling

Dieses Verfahren beschränkt die parallele Durchführung: nur solche Transaktionen dürfen sich zeitlich überlappen, die keine gemeinsamen Datenobjekte behandeln. Damit behindern sich die Transaktionen gegenseitig niemals.

Auch hier ist Voraussetzung, dass bereits vor dem Ausführen der Operationen bekannt ist, auf welche Datenobjekte zugegriffen wird. Da man dies i.a. nicht „sagen“ bzw. automatisch feststellen kann, sind solche

Verfahren unnötigerweise pessimistisch: es werden daher oft unnötigerweise Transaktionen von der Verarbeitung ausgeschlossen.

Als Konsequenz ergibt sich z.B. statt einem einfachen Lock auf Datensatzebene oft ein Lock der gesamten Tabelle, da eben nicht genau bekannt ist, welcher Datensatz verändert werden wird.

4.7.3 Zeitmarken-Verfahren

Die Grundidee dabei ist, dass jede Transaktion einen Zeitstempel (timestamp) zur eindeutigen Identifikation erhält. Diese Zeitmarke (= Transaktions-Startzeitpunkt) ordnet der Transaktion gleichzeitig eine bestimmte Priorität zu: je älter oder jünger eine Transaktion ist, desto eher kommt sie zum Zuge.

Konflikte werden nach einem genau definierten Schema durch eine Kombination aus Warten (Wait) und Restart (Rollback and Retry) jeweils einer der im Konflikt befindlichen Transaktion gelöst.

Wenn die Transaktion A einen von B bereits gesperrten Datensatz sperren will, dann passiert je nach gewähltem Verfahren folgendes:

Wait-Die:	Falls A älter ist als B, wartet A sonst wird A zurückgenommen
Wound-Wait:	Falls A älter ist als B, wird B zurückgenommen sonst wartet A

Abbildung 11: Zeitmarken-Verfahren

Beispiel 1: Wait-Die

Zeit-punkt	Transaktion A (älter)		Preis	Transaktion B (jünger)	
1	SELECT	S	100	-	
2	-	S	100	SELECT	S
3	UPDATE 10%	Anfrage X → warten	110	-	S
4	-	warten	120	UPDATE 20%	Anfrage X → ROLLBACK & Retry
5	…	X	...	Retry zu späterem Zeitpunkt	

Abbildung 12: Beispiel „Wait-Die“

1: Die Transaktion A bekommt den Timestamp 1 und sperrt die Ressource mit einem S Lock.

2: B wird mit Timestamp 2 markiert und sperrt die Ressource ebenfalls mit einem S Lock

3: A versucht einen X Lock zu bekommen. Lt. o.a. Verfahren beginnt sie zu warten (sie ist ja die ältere Transaktion: 1 < 2 (!)).

4: B versucht einen X Lock zu bekommen. Lt. o.a. Verfahren wird sie zurückgenommen und zu einem späteren Zeitpunkt erneut gestartet (jünger, weil 2 > 1). Damit ist auch der S Lock von B aufgehoben.
A beendet gleichzeitig den Wartezyklus (der störende S Lock ist aufgehoben) und setzt seine Verarbeitung fort.

Beispiel 2: Wound-Wait

Zeit-punkt	Transaktion A (älter)		Preis	Transaktion B (jünger)	
1	SELECT	S	100	-	
2	-	S	100	SELECT	S
3	UPDATE 10%	X	110	-	Rollback & Retry
4	...	X		Retry zu späterem Zeitpunkt	

Abbildung 13: Beispiel „Wound-Wait“

1: Die Transaktion A bekommt den Timestamp 1 und sperrt die Ressource mit einem S Lock.

2: B wird mit Timestamp 2 markiert und sperrt die Ressource ebenfalls mit einem S Lock

3: A versucht einen X Lock zu bekommen. Lt. o.a. Verfahren wird dadurch B zurückgenommen und zu einem späteren Zeitpunkt erneut gestartet (sie ist ja die ältere Transaktion: 1 < 2 (!)). Damit ist die Bahn für A frei ...

4: n/a, da B bereits beendet wurde.

In beiden Fällen werden ältere Transaktionen bevorzugt. Die Bevorzugung älterer Transaktionen ist bei Wound-Wait jedoch besonders ausgeprägt, wo sogar der Sperrbesitzer abgebrochen wird, falls er jünger ist als die anfordernde Transaktion. Wound-Wait hat dadurch besseres Leistungsverhalten als Wait-Die.

4.7.3.1 Plausibilitätsnachweis

Es kommt jede Transaktion deshalb an die Reihe, weil sie durch das Verstreichen der Zeit eine höhere Priorität bekommt, und somit die Wahrscheinlichkeit steigt, dass sie beendet werden kann, je länger sie schon wartet. Anm.: Transaktionen behalten ihren Timestamp ...

Es kann nicht wieder zu einem Deadlock kommen, da jeder Transaktion eine eindeutige Priorität zugewiesen wird und daher ein zyklisches Warten unmöglich ist; Transaktionen niederer Priorität werden ja zurückgenommen und geben daher ihre Ressourcen frei. Auch ein Lock „höherer Ordnung“ (Transaktion T_1 wartet auf T_2, T_2 auf T_3, ..., T_{n-1} auf T_n und **T_n auf T_1**) kann nicht vorkommen ...

4.7.4 Timeout

Eine weitere recht gebräuchliche Methode ist die Angabe eines sogenannten Time-Outs: auch hier wird für jede Transaktion vermerkt, wann sie gestartet wurde. Ist sie nach einer bestimmten, als Systemparameter einstellbarer Dauer nicht beendet, so wird angenommen, dass sie in einem Deadlock steckt – sie wird zurückgenommen.

Was hältst du von dieser Methode?

Garantiert sie, dass Deadlocks nicht zustande kommen (nein)? Garantiert sie, dass Deadlocks gelöst werden (ja)?

Garantiert sie, dass jede Transaktion ausgeführt wird (Nur wenn das Timeout groß genug gewählt wurde)?

Was ist bei der Wahl des Timeout zu bedenken (Zu klein: viele Rollback & Retry, evtl. manche Transaktionen nie; zu groß: sehr spätes Erkennen eines Deadlocks und daher schlechte Performance)?

4.8 Konsistentes Lesen in Oracle

Wir wollen uns noch einmal vor Augen führen, dass selbst ein Lesen ohne Sperren zu Inkonsistenzen führen kann:

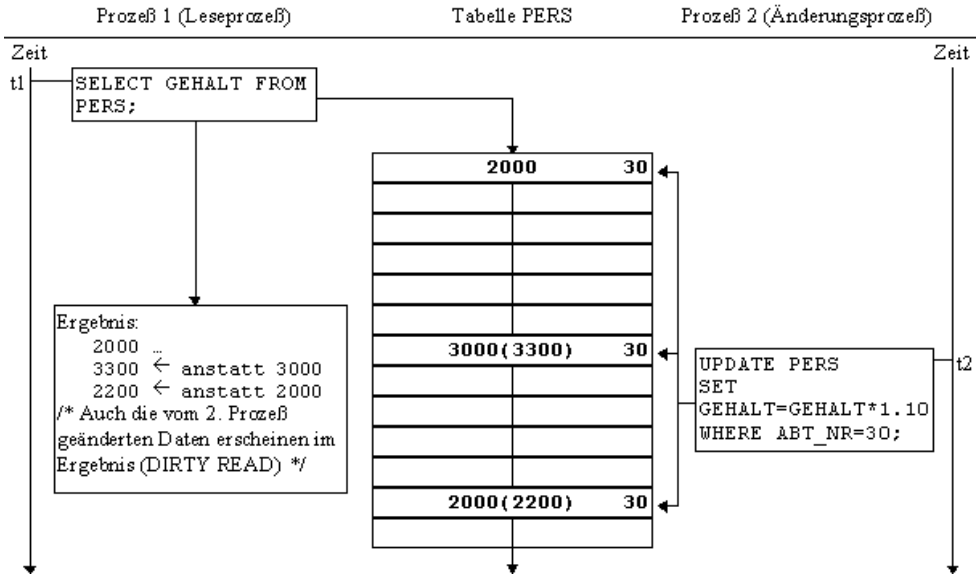

Abbildung 14: Inkonsistentes Lesen

Nehmen wir an, dass der Leseprozess die ganze Tabelle lesen muss, was unter Umständen eine längere Zeit in Anspruch nimmt; dies ist typischerweise für Berichte der Fall. Der Änderungsprozess hingegen kann über Index auf die entsprechenden Datensätze zugreifen und hat dadurch die Änderung in relativ kurzer Zeit abgeschlossen.

Wegen der beliebigen Verschachtelung von Prozessen kann der Änderungsprozess den Änderungsvorgang beginnen, während der Leseprozess erst einen Teil der Tabelle gelesen hat. Das führt dazu, dass der Leseprozess zum Teil die alten, zum Teil aber bereits veränderte Datenwerte liest - was in keiner Weise einem konsistenten Ergebnis entspricht.

4.8.1 Dirty Read

Man nennt dieses Phänomen „Dirty Read“, dessen Auswirkungen in der Praxis oft bewusst in Kauf genommen werden. Diese Methode hat zwar den Vorteil der hohen Gleichzeitigkeit, da sich lesende und ändernde Operationen nicht gegenseitig aussperren können. Dieser Vorteil muss jedoch dadurch erkauft werden, dass das Ergebnis aus einer solchen Leseoperation mit Fehlern behaftet sein kann – aber nicht sein muss (Wahrscheinlichkeit).

Ok, das ist das Dilemma: was kann man dagegen tun? (Sperren)

4.8.2 ANSI-Verfahren

Ein konsistentes Ergebnis bei lesendem Zugriff wird dadurch erreicht, dass auch für lesende Zugriffe entsprechende Sperrmechanismen (S Lock) wirken. Eine gebräuchliche Realisierung ist das Sperren der ganzen Tabelle gegen Änderungen, solange ein Lesebefehl aktiv ist.

Soll ein Lesebefehl auf eine bestimmte Tabelle zur Ausführung kommen, dann ist zu beachten, dass:

- keine Änderungsoperation auf dieser Tabelle aktiv ist; sonst muss der Lesebefehl bis auf das Ende dieser Operation warten;
- in dieser Zeit kein anderer Prozess auf diese Tabelle ändernd zugreifen kann, wenn ein Lesebefehl aktiv ist. Weitere lesende Prozesse sind erlaubt.

Das ist ausreichend für konsistentes Lesen, weshalb es auch vom ANSI-SQL-Komitee als Standard-Sperrverfahren gewählt wurde. Es hast allerdings den Nachteil der stark eingeschränkten Gleichzeitigkeit (Concurrency) – und damit schlechter Performance.
Man stelle sich vor, dass auf diese Weise alle Konten einer Bank für Veränderungen gesperrt werden würden! Eine Möglichkeit, diesen Nachteil zu umgehen ist, solche Applikationen / Berichte nur nachts bzw. dann zu erstellen, wenn die Änderungshäufigkeit sehr gering ist.

4.8.3 Oracle-Read-Consistency-Model

Auf Grund der offensichtlichen Mängel beider Verfahren wich Oracle vom Standard ab und entwickelte ein eigenes Verfahren, das:

- exakte Leseergebnisse, bei
- hoher Gleichzeitigkeit

vereint.

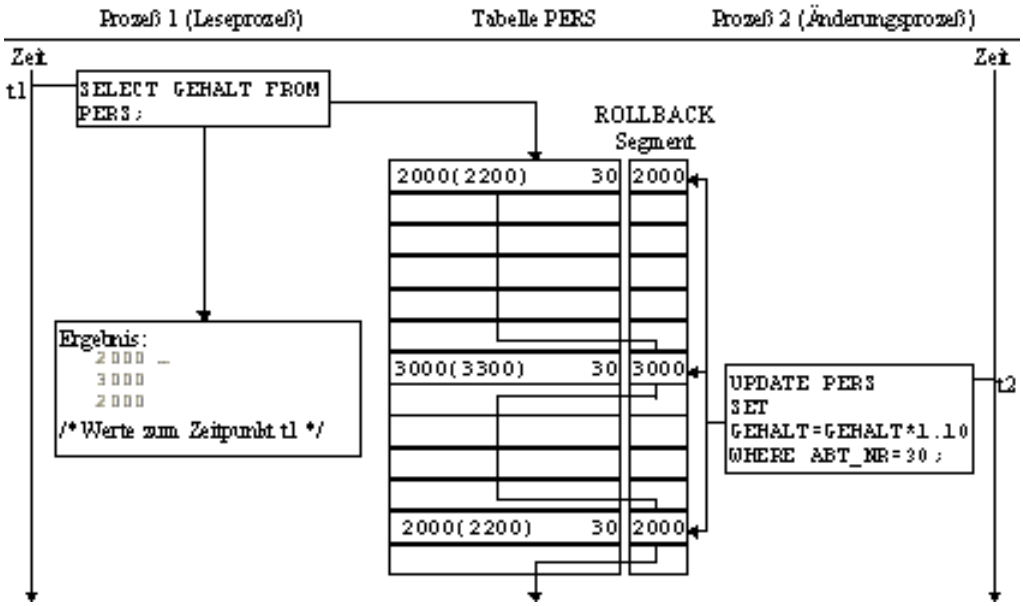

Abbildung 15: Konsistentes Lesen

Oracle verwendet dazu die sogenannten Rollback-Segmente: diese wurden ursprünglich geschaffen um im Falle eines Rollback den ursprünglichen Datenbankzustand wiederherstellen zu können; ihnen kann daher auch im Falle eines Select der jeweils letzte konsistente Zustand der Datenbank entnommen werden.

Durch Einbeziehung der Rollback-Segmente auch für Lesevorgänge, besteht bei Oracle keine Notwendigkeit mehr während Lesevorgänge die Datenbank zu sperren – allerdings sind wiederum Performance-Einbussen im Falle einer Kollision zu verzeichnen: jedoch nur dann!

Damit ist es möglich, dass beliebig viele Lesevorgänge und beliebig viele Änderungsoperationen gleichzeitig auf dieselbe Tabelle wirken können, ohne dass sich die einzelnen Prozesse gegenseitig aussperren.