Kapitel 5

Transaktionsmanagement in Datenbanksystemen

5.1 Mehrbenutzerkontrolle - Concurrency Control

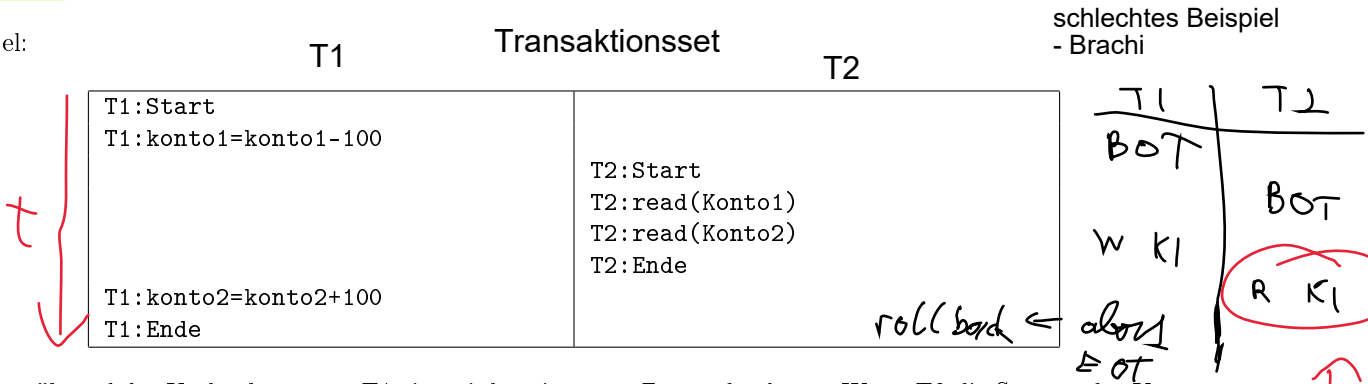
Transaktionen garantieren die Einhaltung der ACID-Eigenschaften[1]. Aufgabe der Concurrency-Kontroll-Komponente von Datenbanksystemen ist es, die Isolation nebenläufig ablaufender Transaktionen zu garantieren.

5.1.1 Fehlerklassen bei nebenläufigen Transaktionen

Es gibt verschiedene Fehlerklassen, bei denen die Isolationseigenschaft der Transaktionen verletzt ist. Diese Fehler soll die Concurrency-Kontroll-Komponente verhindern.

dirty read

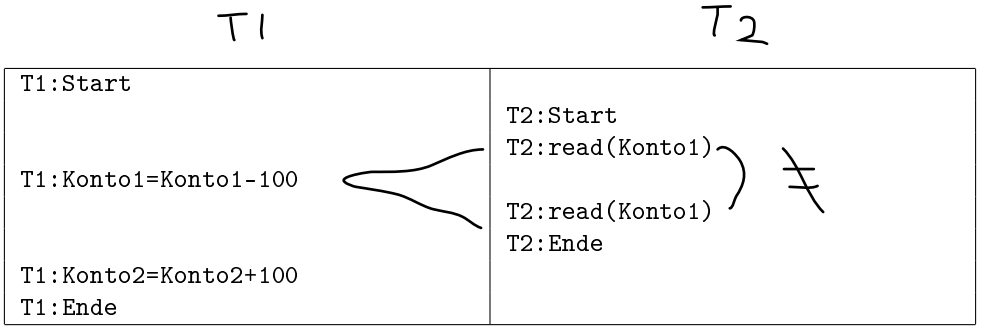
Beispiel:



T2 hat während der Umbuchung von T1 einen inkonsistenten Zustand gelesen. Wenn T2 die Summe der Konten Konto1 und Konto2 berechnet hat, hat T2 ein falsches Ergebnis geliefert.

nonrepeatable read

Beispiel:

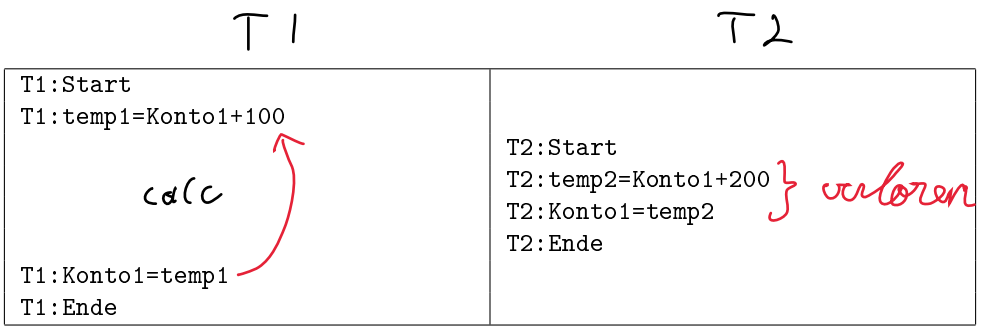


[1] Atomicity-Consistency-Isolation-Durability

T2 hat zweimal Konto1 gelesen, bekam aber verschiedene Werte, obwohl T2 den Wert von Konto1 nicht geändert hat. Die Isolationseigenschaft ist verletzt.

lost update

Beispiel:



Beide Transaktionen T1 und T2 wollen Beträge auf dasselbe Konto Konto1 einzahlen, berechnen aber den neuen Kontostand in eigenen lokalen Variablen temp1 und temp2. Durch den angezeigten parallelen Ablauf geht die Änderung von T2 verloren.

Phantomproblem

Werden nur physisch vorhandene Tupel gesperrt, kann das sogenannte Phantomproblem auftreten.

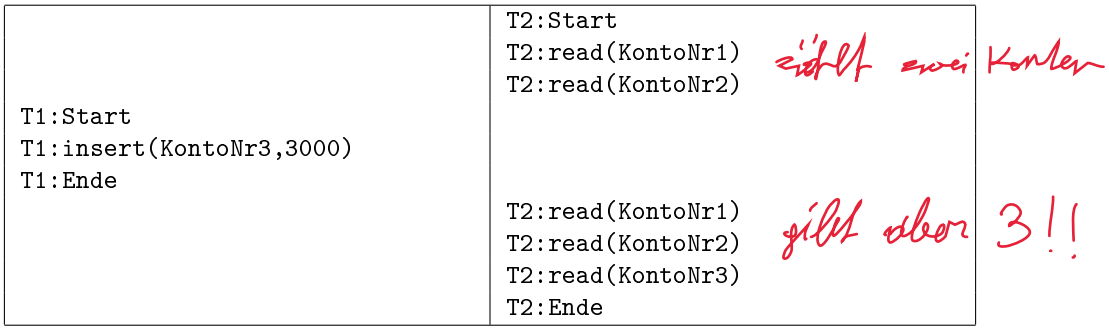
Beispiel:

Die zwei Transaktionen T1 und T2

```
1  T1:  EXEC SQL insert into Konten values(KontoNr3,3000);

3  T2:  EXEC SQL select COUNT(KontoNr) into :Kontenanzahl from Konten
4       if (Kontenanzahl<10) //wenige Konten, zeige Durchschnittsbetrag
5       {
6          EXEC SQL select SUM(KontoStand) into :Summe from Konten
7          printf("Durchschnittsbetrag: %f\n", Summe/Kontenanzahl);
8       }
```

könnten zu folgendem parallelen Ablauf von Transaktionen T1 und T2 führen (wir nehmen an, es gibt bereits 2 Konten mit KontoNr1 und KontoNr2):



Die Durchschnittsberechnung liefert den falschen Wert, weil nach der Berechnung der Anzahl (2) der Konten noch ein neues Konto eingefügt wird, dessen Anfangsbetrag mit in die Summe einfließt. Das Phantomproblem läßt sich auch nicht dadurch beheben, dass man bereits eingetragene Datenbanktupel oder bereits gelesene Datenbanktupel sperrt, da das neue Tupel (Konto3,3000) noch nicht existierte, als das erste Select-Statement ausgeführt wurde, also auch nicht gesperrt werden konnte.

5.1.2 Serialisierbarkeit

beliebige Reihenfolge $T_1 < T_2$ oder $T_2 < T_1$ Korrekt ausführbar

Jeder serielle Ablauf von Transaktionen wird als korrekt angesehen, weil die Transaktionen vollständig isoliert voneinander ablaufen. Andererseits will man mehr Parallelität von Transaktionen zulassen. Deshalb soll das Datenbanksystem serialisierbare Abläufe von Transaktionen erlauben.

Serialisierbar ist ein paralleler Ablauf von Transaktionen, wenn er einem seriellen Ablauf entspricht.

Die Serialisierbarkeit von parallelen Ausführungen kann auch mit graphentheoretischen Mitteln überprüft werden: Der **Präzedenzgraph** für eine Ausführung über einer Transaktionsmenge T ist ein gerichteter Graph, dessen Knoten die Transaktionen sind, und in dem eine Kante von T_i nach T_j führt, wenn:

- T_j liest oder schreibt ein Objekt, das zuletzt von T_i geschrieben wurde, oder
- T_i liest ein Objekt, auf das T_j danach als erste Transaktion schreibt.

Eine parallele Ausführung ist genau dann serialisierbar, wenn der Präzedenzgraph zyklenfrei ist.

Betrachten wir 3 verschiedene parallele Ausführungen A1, A2, A3 von 2 Transaktionen T1 und T2:

A1

T1	T2
read A	
write A	
	read A
	write A
read B	
write B	
	read B
	write B

A2

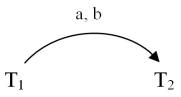
T1	T2
read A	
write A	
read B	
write B	
	read A
	write A
	read B
	write B

A3

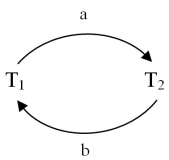
T1	T2
read A	
write A	
	read A
	write A
	read B
	write B
read B	
write B	

Wenn wir die zugehörigen Präzedenzgraphen betrachten, stellen wir fest, dass die Ausführungen A1 und A2 serialisierbar sind, während A3 einen zyklischen Präzedenzgraphen besitzt und somit keine serialisierbare Ausführung darstellt.

A1, A2:



A3:



5.1.3 Sperrprotokolle

Die am weitesten verbreitete Methode, Serialisierbarkeit von Ausführungen zu erlangen ist dynamisches Sperren und Freigeben von Objekten einer Datenbank. Folgende Bedingungen müssen erfüllt werden:

- Eine Transaktion greift auf ein Objekt nur zu, wenn sie dieses zuvor gesperrt hat (`LOCK`).
- Eine Transaktion sperrt niemals ein Objekt, das sie bereits selbst gesperrt hat.
- Eine Transaktion versucht niemals, ein Objekt freizugeben, das sie nicht zuvor gesperrt hat.
- Vor Beendigung einer Transaktion werden die Sperren auf alle Objekte, die von ihr gesperrt wurden, wieder freigegeben (`UNLOCK`).

Sperren alleine ist nicht hinreichend, damit eine Transaktion im allgemeinen Fall serialisierbar ist, wie das folgende Beispiel zeigt:

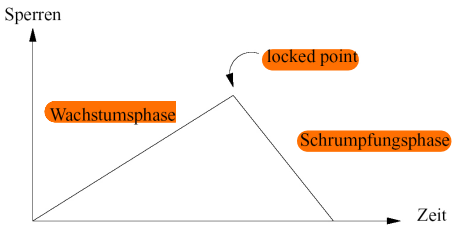
Zeit	T1	T2
1	LOCK A	
2	LOCK C	
3	WRITE A	
4	WRITE C	
5	UNLOCK A	
6		LOCK A
7		WRITE A
8		LOCK B
9		UNLOCK A
10		WRITE B
11		UNLOCK B
12	UNLOCK C	
13	LOCK B	
14	WRITE B	
15	UNLOCK B	
16		LOCK C
17		WRITE C
18		UNLOCK C

Zeichnen Sie dazu den Präzedenzgraphen!

Das 2-Phasen-Sperrverfahren (2-Phase-Locking)

Am Beginn jeder Transaktion werden die Sperren auf alle benötigten Objekte gesetzt. Der Zeitpunkt des Belegens der letzten Sperre wird als Sperrpunkt (locked point) bezeichnet. Ab diesem Zeitpunkt darf keine weitere Sperre gesetzt werden. Weiters dürfen erst ab diesem Zeitpunkt Sperren wieder aufgehoben werden.

In der Praxis werden meist alle UNLOCK-Befehle bis zum Ende der Transaktion verzögert und/oder zu einem Befehl (`COMMIT`) zusammengefasst, der gleichzeitig das Ende der Transaktion darstellt.



Wenn alle Transaktionen einer Transaktionsmenge 2-Phasen-Transaktionen sind, dann ist jede gültige Ausführung serialisierbar.

Deadlock

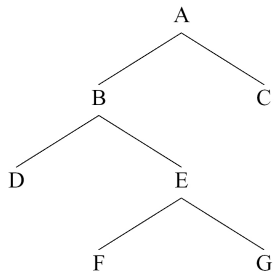
Dynamisches Sperren von Objekten führt zum Problem des Deadlocks. Deadlocks müssen entweder von vornherein vermieden oder erkannt und aufgelöst werden.[2]

T1	T2	
LOCK A		
	LOCK B	
LOCK B		← T1 wartet auf B
	LOCK A	← T2 wartet auf A

[2] In der Praxis ist Deadlockumgehung zu zeitaufwendig zu implementieren. Deadlockerkennung und Abbruch von Transaktionen wird bevorzugt (z.B. in Oracle). Wenn eine Transaktionsprozedur aber Benutzereingaben verarbeitet, könnte Deadlock-Vermeidung durch Anordnung der Betriebsmittel vorteilhafter sein, weil so verhindert wird, dass ein Benutzer seine Eingaben wiederholen muss.

Das Baumprotokoll

Gehen wir davon aus, dass wir eine Ansammlung von Objekten, wie z.B. Datenbanken, Anwendungen (Menge von Relationen), Relationen, oder Tupel als Knoten eines Baumes gespeichert haben. Dieser Baum kann z.B. ein binärer Baum (siehe: Programmieren 2. Jahrgang) sein.



Eine Transaktion T erfüllt das Baumprotokoll, wenn gilt:

- Ein Objekt O wird nur dann von T gesperrt, wenn T bereits eine Sperre auf den Vater von O hält.
- Regel 1 gilt nicht für das erste Objekt im Baum, das von T gesperrt wird.
- Ein Objekt O, das von einer Transaktion T gesperrt und wieder freigegeben wurde, kann von T anschließend nicht mehr gesperrt werden.

Dieses Sperrverfahren garantiert Serialisierbarkeit und vermeidet Deadlock.

Zeitstempelverfahren

Beim Zeitstempelverfahren wird ein Konflikt nicht durch das Setzen von Sperren vermieden, sondern wenn ein Konflikt entdeckt wird, wird die Transaktion, die zu dem Konflikt geführt hat abgebrochen und neu gestartet. Zum Erkennen von Konflikten werden Transaktionen und Datenbankobjekten Zeitstempel (timestamps) zugeordnet:

- Jede Transaktion erhält als eindeutigen Zeitstempel den Zeitpunkt des Transaktionsbeginns.
- Jedes Datenbankobjekt übernimmt sowohl den Zeitstempel der letzten zugreifenden Schreib-Transaktion, als auch den der letzten Lese-Transaktion.

Durch folgende Bedingungen wird die Konsistenz der Datenbank sichergestellt:

1. Eine Transaktion mit Zeitstempel t_1 darf ein Objekt mit Schreibstempel t_w nicht lesen, wenn $t_w > t_1$ gilt, also das Objekt bereits von einer jüngeren Transaktion verändert wurde. Die Transaktion muss in diesem Fall abgebrochen werden.
2. Eine Transaktion mit Zeitstempel t_1 darf ein Objekt mit Lese-Zeitstempel t_r nicht verändern (schreiben), wenn $t_r > t_1$ oder $t_w > t_1$ gilt, also das Objekt bereits von einer jüngeren Transaktion gelesen oder geschrieben wurde. Die Transaktion muss in diesem Fall abgebrochen werden.

Eine abgebrochene Transaktion wird neu gestartet und erhält die dem Zeitpunkt des Neustarts entsprechende Zeitmarke.

Zu beachten ist, dass zwei Lesezugriffe keinen Konflikt darstellen, und von zwei Schreibzugriffen nur der jüngere - d.h., der von der später gestarteten Transaktion durchgeführte - erhalten bleibt, da der ältere Wert überschrieben bzw. gar nicht geschrieben wird.

Die Serialisierungsreihenfolge der Transaktionen entspricht der Reihenfolge der Zeitstempel, also der Startzeitpunkte.

Um beim Abbruch einer Transaktion alle bereits erfolgreich durchgeführten Operationen dieser Transaktion rückgängig machen zu können, werden die geänderten Werte eines Objekts vorerst in einen zu dem Objekt

gehörenden privaten Arbeitsbereich geschrieben (Phase 1) und erst bei der erfolgreichen Beendigung der Transaktion werden die geänderten Werte in die Datenbank übernommen (Phase 2).

Beispiel:
Seien T1, T2, T3 drei Transaktionen mit den Zeitstempeln 200, 150 und 175. Zu Beginn der Ausführung haben alle Objekte sowohl Lese- als auch Schreibstempel auf 0 gesetzt. Da T1 als erstes auf B lesend zugreifen will, wird der Lesestempel von B auf 200 gesetzt. Danach werden die anderen beiden Lesestempel gesetzt, T1 greift schreibend auf B zu, der Schreibstempel wird gesetzt. Dasselbe passiert auch mit A. Nun will T2 C beschreiben; der Zeitstempel von T2 ist 150, C hat aber einen Lesestempel von 175. Deshalb wird T2 abgebrochen. T3 will noch auf A schreiben, was allerdings auch zu einem Abbruch führt, da der Schreibstempel auf 200 steht. T2 und T3 bekommen einen neuen Zeitstempel, der größer als 200 ist, z.B. 250 und 260. Die Reihenfolge der Transaktionen ist nun: T1<T2<T3

	T_1	T_2	T_3	A	B	C
	200	150	175	RT = 0	RT = 0	RT =0
				WT = 0	WT = 0	WT = 0
1.	READ B				RT = 200	
2.		READ A		RT = 150		
3.			READ C			RT = 175
4.	WRITE B				WT = 200	
5.	WRITE A			WT = 200		
6.		WRITE C ⇝				⇝
7.			WRITE A ⇝	WT = 200		

Beim Zeitstempelverfahren gibt es keine Deadlocks wie beim Sperrverfahren. Allerdings kann es auch hier zu unbeschränkter Verzögerung kommen, durch eine unendliche Folge von Abbrüchen, einem sogenannten Livelock:

	T_1	T_2	T_1	T_2	A	B
	100	110	120	130	RT = 0	RT =0
					WT = 0	WT = 0
1.	WRITE B					WT = 100
2.		WRITE A			WT = 110	
3.	READ A ⇝				⇝	
4.			WRITE B			WT = 120
5.		READ B ⇝				⇝
6.				WRITE A	WT =130	
7.			READ A ⇝		⇝	

Nach dem Abbruch von T1 wird diese Transaktion neu gestartet. Es tritt wieder ein Konflikt mit T2 auf, diesmal wird T2 abgebrochen und neu gestartet. Durch zyklisches Abbrechen tritt eine unbeschränkte Verzögerung auf.

5.1.4 Transaktionen in SQL

Das Transaktionsmanagement eines Datenbankmanagementsystemes auf Multiusersystemen regelt den parallelen Zugriff mehrerer Benutzer auf die Datenbank. Eine Transaktion besteht meist aus mehreren SQL-Statements, kann aber auch nur aus einem Statement bestehen.

Beginn und Ende einer Transaktion

Mit dem ersten ausführbaren SQL-Statement nach einem **COMMIT**, **ROLLBACK** oder einer Anbindung an die Datenbank wird eine Transaktion begonnen. Von nun an gehören alle SQL-Statements zu dieser Transaktion, bis sie beendet wird.

Eine Transaktion kann beendet werden durch:

- ein COMMIT [WORK]-Statement - Damit werden alle Änderungen, die seit dem Beginn dieser Transaktion durchgeführt wurden, permanent in die Datenbank eingetragen. Mit dem nächsten ausführbaren SQL-Statement beginnt eine neue Transaktion.
- ein ROLLBACK [WORK]-Statement - Damit werden alle Änderungen, die seit dem Beginn dieser Transaktion durchgeführt wurden, rückgängig gemacht.
- ein DDL[3]-Statement (z.B. CREATE, DROP, RENAME). Bevor ein DDL-Statement ausgeführt wird, wird eine allenfalls begonnene Transaktion automatisch beendet, und es werden alle bisherigen Änderungen in die Datenbank eingetragen. DDL-Statements sind immer Transaktionen, die nur aus diesem einen Befehl bestehen. Wurde ein DDL-Statement erfolgreich beendet, so beginnt mit dem nächsten Statement wieder eine neue Transaktion.
- das reguläre Beenden von SQL*PLUS. Damit werden alle Änderungen, die seit dem Beginn der aktuellen Transaktion durchgeführt wurden, permanent in die Datenbank eingetragen.
- ein abnormales Ende von SQL*PLUS. Damit werden alle Änderungen, die seit dem Beginn der aktuellen Transaktion durchgeführt wurden, rückgängig gemacht.

Der folgende Ablauf veranschaulicht dies:

`select * from winzer;`	gibt die gesamte Relation aus.
`delete from winzer;`	leert die Relation, wie man sich mit
`select * from winzer;`	überzeugen kann.
`rollback;`	macht die Veränderung rückgängig, wie man mit
`select * from winzer;`	sieht.
`select * from wein;`	gibt diese Relation aus.
`delete from wein;`	leert die Relation, was dann durch
`commit;`	endgültige Wirkung erhält:
`select * from wein;`	Da hilft auch kein
`rollback;`	mehr:
`select * from wein;`	Mit
`delete from protokoll;`	und versehentlichem
`quit;`	gehen auch diese Daten noch ex, wie man nach
`sqlplus username@oradb`	und
`select * from protokoll;`	feststellen kann. Jetzt hilft nur noch
`@create.sql;`	um die ursprüngliche Datenbank wieder zu bekommen.

Sicherungspunkte

Für eine längere Transaktion können zwischendurch Sicherungspunkte gesetzt werden:

```
1 SQL> SAVEPOINT <savepoint>;
```

[3] DDL=Data Definition Language

Falls die Transaktion in einem späteren Stadium scheitert, kann man auf den Sicherungspunkt zurücksetzen, und von dort aus versuchen, die Transaktion anders zu einem glücklichen Ende zu bringen:

```
SQL> ROLLBACK [WORK] TO [SAVEPOINT] <savepoint>;
```

nimmt alle Änderungen bis zu <savepoint> zurück.

Was passiert in der folgenden Oracle-Sitzung?

```
create table testtab1(a integer primary key,b varchar2(5));
insert into testtab1 values (1, 'A');
commit;
insert into testtab1 values (2, 'B');
savepoint sp1;
insert into testtab1 values (3, 'C');
select * from testtab1;
rollback to sp1;
select * from testtab1;
insert into testtab1 values (4, 'D');
rollback;
select * from testtab1;
```

Transaktionslevel

Im Laufe einer Transaktion können verschiedene Probleme (wie Lost Update, Dirty Read, Nonrepeatable Read, Phantom Read) auftreten. SQL92 unterstützt zur Vermeidung dieser Phänomene verschiedene Isolationsstufen (Isolation Levels). Die folgende Tabelle zeigt, durch welche Optionen welche Phänomene vermieden werden können.

Stufe	Lost Update	Dirty Read	Nonrep. Read	Phantom Read
READ UNCOMMITTED	nicht möglich	möglich	möglich	möglich
READ COMMITTED	nicht möglich	nicht möglich	möglich	möglich
REPEATABLE READ	nicht möglich	nicht möglich	nicht möglich	möglich
SERIALIZABLE	nicht möglich	nicht möglich	nicht möglich	nicht möglich

Die Änderung des Isolationslevels erfolgt durch

```
set transaction
  {{ read only | read write }
   | isolation level [read uncommitted | read committed | repeatable read
                     | serializable]
   [name <Transaktionsname>]
   | name <Transaktionsname>};
```

am Beginn einer Transaktion.

Setzen des Isolationskonzepts für die gesamte Sitzung (Oracle):

```
Alter session set isolation_level = [read committed | serializable];
```

Beispiel:
In einer Oracle-Datenbank wurde eine Tabelle mittels

```
SQL> create table testtab1 (a integer primary key, b varchar2(10));
```

angelegt. Sie enthält zu Beginn eine Zeile mit den Werten 1 und 'A'. Nun betrachten wir zwei Transaktionen, die beide auf diese Tabelle zugreifen. Für jede dieser Transaktionen wird jeweils eine SQL*Plus-Sitzung gestartet. Die linke und rechte Spalte der nächsten Tabelle geben jeweils die in den entsprechenden Sitzungen ausgeführten Anweisungen an.

`alter session set`	
`isolation_level=serializable;`	
	`alter session set`
	`isolation_level=serializable;`
`update testtab1 set b='B' where a=1;`	
	`select * from testtab1;`
`commit;`	
	`select * from testtab1;`
	`commit;`
	`select * from testtab1;`

Ergebnis:
Erst nach dem `commit` der 2. Sitzung werden die Änderungen für diese Sitzung sichtbar.

5.1.5 Aufgaben

Beispiel 1
Ist die angegebene Ausführung der Transaktionen T1, T2, T3 und T4 serialisierbar? Warum (nicht)? (Begründung mit Hilfe des Präzedenzgraphen, oder Angabe der äquivalenten seriellen Ausführung)

$T1$:	$T2$:	$T3$:	$T4$:
	READ a		
		READ a	
	UPDATE b		
		UPDATE a	
READ b			
			READ b
UPDATE c			
			UPDATE a
UPDATE b			

Beispiel 2
Angenommen wir verwenden das Zeitstempel-Verfahren für die Synchronisierung der Transaktionen in Beispiel 1. Welche der 4 Transaktionen wird abgebrochen, wenn wir folgende Voraussetzung für die Zeitstempel der Transaktionen T1-T4 treffen:

1. 300, 310, 320 und 330
2. 250, 200, 210 und 275

Beispiel 3
Gegeben ist folgende Parallelausführung von Transaktionen:

$T1$:	$T2$:	$T3$:
UPDATE a		
	UPDATE b	
		READ a
	UPDATE c	
READ c		
		READ b

1. Stellen Sie die Abhängigkeitsbeziehungen von T1, T2 und T3 durch Angabe des Präzedenzgraphen dar.
2. Ist die angegebene Ausführung von T1, T2 und T3 serialisierbar?

Beispiel 4
Parallele Transaktionen werden mit Hilfe des Zeitstempelverfahrens synchronisiert. Gegeben sind folgende Transaktionen mit ihren Zeitstempeln:

$T1$:	$T2$:	$T3$:	a	b	c
45	65	90	$RT = WT = 0$	$RT = WT = 0$	$RT = WT = 0$
READ a			$RT = 45$		
	WRITE b			$WT = 65$	
	WRITE c				$WT = 65$
		WRITE a	$WT = \ldots$		
WRITE c					$WT = \ldots$
		READ C			$RT = \ldots$
READ b				$RT = \ldots$	

1. Tragen Sie in der obigen Darstellung die fehlenden Werte für die Lese- und Schreibstempel der Objekte a, b und c ein. Kennzeichnen Sie den Abbruch einer Transaktion.
2. Welche Serialisierungsreihenfolge von T1, T2 und T3 wird in der obigen Ausführung durch die Anwendung des Zeitstempelverfahrens erzwungen?

Beispiel 5
Testen Sie Transaktionen in Oracle mittels SQL*PLUS, im Speziellen

- Commit,
- Rollback,
- Savepoint und
- Isolation Level

anhand der 3 SQL-Beispiele aus Abschnitt 5.1.4.