

Kap. 5: Transaktionsmanagement

Serialisierbarkeit

Lösung von Synchronisationsproblemen

T1	T2
	Read(X);
	X := X+10;
	Write(X);
Read(X);	
X := X*1.1;	
Write(X);	
Read(Y);	
Y := Y*1.1;	
Write(Y);	
	Read(Y);
	Y := Y+10;
	Write(Y);

Welche Werte für X und Y nach den Transaktionen sind denn richtig?

Startwerte X=Y=10

Definition korrekter Ausführung:

- Ausführung konkurrierender Transaktionen ist **korrekt** das Ergebnis gleich **einem** der Ergebnisse ist, das durch sequentielle Abarbeitung der Transaktionen entstehen würde

- Ausführungsplan, Schedule:
 - Geordnete Folge der Aktionen (read, write, commit, abort/rollback) einer **Menge** von Transaktionen
- Schedule heißt **seriell**, wenn die Schritte je einer Transaktion unmittelbar aufeinander folgen und nicht mit Schritten anderer Transaktionen verschachtelt sind
- Ist aus praktischen Gründen nicht akzeptabel, da echter Mehrbenutzerbetrieb erforderlich!
- Schedule heißt **serialisierbar**, wenn das Ergebnis äquivalent zu dem eines seriellen Schedules ist
- Hier werden zunächst nur erfolgreiche Transaktionen betrachtet
 - Die Frage nach der Rücksetzbarkeit kommt später

Beispiele für Transaktionsschedules

Bestimme den Typ dieser Schedules!

nicht serialisierbar		seriell		serialisierbar	
T1	T2	T1	T2	T1	T2
Read(X);		Read(X);		Read(X);	
X -= 10;		X -= 10;		X -= 10;	
	Read(X);	Write(X);		Write(X);	
	X += 15;	Read(Y);			Read(X);
Write(X);		Y += 10;			X += 15;
Read(Y);		Write(Y);		Read(Y);	
	Write(X);		Read(X);	Y += 10;	
Y += 10;			X += 15;		Write(X);
Write(Y);			Write(X);	Write(Y);	

T1	T2	T1	T2	T1	T2
Read(X);			Read(X);	Read(X);	
X += 10;			X *= 1.1;	X += 10;	
Write(X);			Write(X);	Write(X);	
Read(Y);		Read(X);			Read(X);
Y += 10;		X += 10;			X *= 1.1;
Write(Y);		Write(X);			Write(X);
	Read(X);		Read(Y);	Read(Y);	
	X *= 1.1;		Y *= 1.1;	Y += 10;	
	Write(X);		Write(Y);	Write(Y);	
	Read(Y);	Read(Y);			Read(Y);
	Y *= 1.1;	Y += 10;			Y *= 1.1;
	Write(Y);	Write(Y);			Write(Y);

Abstraktes Modell eines DBMS

Vereinfachtes Modell eines DBMS für die Einführung in die Theorie der Serialisierbarkeit

Die Datenbank besteht aus Objekten, auf die die Transaktionen lesend und schreibend zugreifen.

Eine Transaktion ist aus der Sicht des DBMS eine Folge von

- Atomaren Lese operationen
- Atomaren Schreib operationen
- Ihrem Beginn BOT und Abschluss (COMMIT oder ABORT)
- **DBMS weiß nichts über die von der Transaktion durchgeführten Berechnungen**

Notation eines Schedule (Beispiel):

- $R_1(X)$; $W_1(X)$; $R_2(X)$; $W_2(X)$;
 $R_1(Y)$; $W_1(Y)$; $R_2(Y)$; $W_2(Y)$;
 C_1 ; C_2 ;

Abkürzungen

- R Lesen
- W Schreiben
- C Commit
- A Abort
- Index Transaktionsnummer
- Objekte in Klammern

T1	T2
Read(X);	
X += 10;	
Write(X);	
	Read(X);
	X *= 1.1;
	Write(X);
Read(Y);	
Y += 10;	
Write(Y);	
	Read(Y);
	Y *= 1.1;
	Write(Y);

Gegeben seien die Transaktionen:

- T1: $R_1(a)$ $W_1(a)$ $W_1(b)$ und T2: $R_2(a)$ $W_2(a)$

Beispiel für T1: $R_1(a)$, $b := a$, $a := a + 100$, $W_1(a)$, $W_1(b)$

Beispiel für T2: $R_2(a)$, $a := a + 200$, $W_2(a)$

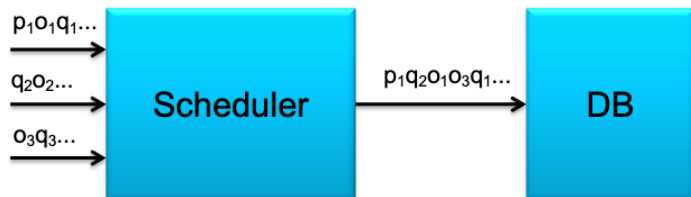
Anfangswert $a = 10$

Gegeben seien die Operationsfolgen

	a	b
Ablauf 1: $R_1(a)$ $W_1(a)$ $W_1(b)$ $R_2(a)$ $W_2(a)$	310	10
Ablauf 2: $R_2(a)$ $W_2(a)$ $R_1(a)$ $W_1(a)$ $W_1(b)$	310	210
Ablauf 3: $R_1(a)$ $R_2(a)$ $W_1(a)$ $W_1(b)$ $W_2(a)$?	?
Ablauf 4: $R_1(a)$ $W_1(a)$ $R_2(a)$ $W_1(b)$ $W_2(a)$?	?
Ablauf 5: $R_1(a)$ $R_2(a)$ $W_1(a)$ $W_2(a)$ $W_1(b)$?	?
Ablauf 6: $R_1(a)$ $W_1(a)$ $R_2(a)$ $W_2(a)$ $W_1(b)$?	?

- Ergänzen Sie die Tabelle. Welche Abläufe sind serialisierbar? (5 Minuten)

Modell eines Schedulers



Wie entscheidet der Scheduler, welche Reihenfolge serialisierbar ist?

Es sind nur die Operationen Read, Write, Commit, Abort sichtbar!

Konflikte zwischen Operationen

- Transaktionen T_i und T_j greifen auf dasselbe Objekt A zu
- $R_i(A)$ und $R_j(A)$ stehen nicht in Konflikt zueinander:
 - Ihre Reihenfolge ist irrelevant, da beide TAs in jedem Fall denselben Zustand lesen.
- $R_i(A)$ und $W_j(A)$ stehen in Konflikt zueinander:
 - T_i liest entweder den alten oder den neuen Wert von A. Es muss also $R_i(A)$ vor $W_j(A)$ oder $W_j(A)$ vor $R_i(A)$ angegeben werden.
- $W_i(A)$ und $R_j(A)$ entsprechend
- $W_i(A)$ und $W_j(A)$ stehen in Konflikt zueinander: Die Reihenfolge der Ausführung ist entscheidend für den Zustand der Datenbank nach Ausführung beider Transaktionen.

Die Reihenfolge von Leseoperationen, die unmittelbar aufeinanderfolgen, ist unwesentlich

Zwei Operationen verschiedener Transaktionen stehen in Konflikt zueinander, wenn sie auf dasselbe Objekt zugreifen und mindestens eine der beiden eine Schreiboperation ist.

Äquivalenz zweier Schedules

- Bei Änderungen Ergebnisse weiterhin gleich

Zwei Schedules S und S' über der gleichen Menge von Transaktionen sind **konfliktäquivalent**, wenn sie die Konfliktoperationen in derselben Reihenfolge ausführen.

Beispiele äquivalenter Umformungen:

- $R_1(A)$ $R_2(B)$ $W_1(A)$ $W_2(C)$ $R_1(B)$ $W_1(B)$ $R_2(A)$ $W_2(A)$
- $R_1(A)$ $W_1(A)$ $R_2(B)$ $W_2(C)$ $R_1(B)$ $W_1(B)$ $R_2(A)$ $W_2(A)$
- $R_1(A)$ $W_1(A)$ $R_2(B)$ $R_1(B)$ $W_2(C)$ $W_1(B)$ $R_2(A)$ $W_2(A)$
- $R_1(A)$ $W_1(A)$ $R_1(B)$ $R_2(B)$ $W_2(C)$ $W_1(B)$ $R_2(A)$ $W_2(A)$

Serialisierbarkeit

Ein Schedule S ist **konfliktserialisierbar**, wenn er konfliktäquivalent zu einem seriellen Schedule ist.

Das bedeutet auch: es gibt einen seriellen Schedule S', der das gleiche Ergebnis wie S liefert

Erklärung der Definition:

- Wenn man S durch Vertauschung von Operationen, die nicht in Konflikt zueinander stehen, in einen seriellen Schedule umwandeln kann, ist S konfliktserialisierbar.

Beispiel:

Folgendes Beispiel ist nicht konfliktserialisierbar. Grund:

- Read(B) in T1 und Write(B) in T2 stehen in Konflikt
- Write(B) in T2 und Write(B) in T1 stehen in Konflikt
- Man kann keine serielle Ausführung erlangen, ohne eines dieser Paare zu tauschen

Anders ausgedrückt:

- Der erste Konflikt erzwingt, dass T2 nach T1 ausgeführt wird
- Der zweite Konflikt erzwingt, dass T1 nach T2 ausgeführt wird

T1	T2
Read(A);	
	Read(B)
A += 10	
Write(A);	
	B += 10
Read(B);	Write(B);
B -= 10;	Read(C);
Write(B);	C -= 10;
	Write(C);

Abhängigkeitsgraph - Allgemeine Vorgehensweise

Eine Transaktion T_j heißt **abhängig** von T_i , wenn es zwei Operationen a_i und b_j gibt, die in Konflikt stehen und a_i wird in S vor b_j ausgeführt.

Definition des (gerichteten) **Abhängigkeitsgraphen** zu S:
 $G(S) = (T, U)$

T Knotenmenge, repräsentiert die Transaktionen aus S,
 U Kantenmenge mit $(T_i, T_j) \in U \Leftrightarrow T_j$ ist abhängig von T_i

- Graph: Zyklus ist ein Problem, roter Pfeil bedeutet T1 muss vor T2 und blauer Pfeil T2 muss vor T1.
- Bei einem konfliktserialisierbaren Schedule darf es keine Zyklen geben

T1	T2
Read(A);	
	Read(B)
A += 10	
Write(A);	
	B += 10
Read(B);	Write(B);
B -= 10;	Read(C);
Write(B);	C -= 10;
	Write(C);



T1	T2
Read(A);	
	Read(B)
A += 10	
	B += 10
Write(A);	
	Write(B);
Read(B);	Read(C);
B -= 10;	C -= 10;
Write(B);	Write(C);



Serialisierbarkeitskriterium als Algorithmus

Aufbau eines Abhängigkeitsgraphen für einen gegebenen Schedule S zur Ausführung von n Transaktionen T_1, T_2, \dots, T_n :

- Für jede Transaktion T_i wird ein Knoten erzeugt
- Es wird eine Kante (T_i, T_j) erzeugt, wenn es in S ein $R_j(X)$ nach einem $W_i(X)$ gibt.
- Es wird eine Kante (T_i, T_j) erzeugt, wenn es in S ein $R_j(X)$ nach einem $R_i(X)$ gibt.
- Es wird eine Kante (T_i, T_j) erzeugt, wenn es in S ein $W_j(X)$ nach einem $W_i(X)$ gibt.
- Zur verbesserten Übersicht kann die Kante mit dem jeweiligen Objekt, das den Konflikt hervorruft, beschriftet werden

**Ist der entstandene Graph zyklensfrei,
so ist der Schedule S konfliktserialisierbar.**

Beispiel zur Übung

T1	T2	T3
		Read(Y);
		Read(Z);
Read(X);		
Write(X);		
		Write(Y);
		Write(Z);
	Read(Z);	
Read(Y);		
Write(Y);		
	Read(Y);	
	Write(Y);	

T1

T2

T3

Serialisierbarkeitstheorem

- Wenn Transaktionen nichts machen sind sie serialisierbar, aber nicht konfliktserialisierbar

Ein Schedule S ist **genau dann** konfliktserialisierbar, wenn der zugehörige Abhängigkeitsgraph zyklensfrei ist.

Vergleich Serialisierbarkeit / Konfliktserialisierbarkeit:

- Ein konfliktserialisierbarer Schedule ist auch serialisierbar
- Aber: Nicht alle serialisierbaren Schedules sind konfliktserialisierbar:



Abbruch von Transaktionen

- Dirty Read entsteht bei allen, aber 1. und 2. Tabelle sind ok. Bei der 2. Tabelle macht das der dbms automatisch einen abort => Kaskadierender Abbruch
- Bei der 3. Tabelle Problem commit von T2 vor abort von T1

ok		kaskadierender Abbruch		nicht rücksetzbar	
T1	T2	T1	T2	T1	T2
	Read(X);		Read(X);		Read(X);
	Write(X);		Write(X);		Write(X);
Read(X);		Read(X);		Read(X);	
Write(X);		Write(X);		Write(X);	
	Read(Y);		Read(Y);		Read(Y);
	Write(Y);		Write(Y);		Write(Y);
Read(Y);		Read(Y);		Read(Y);	
Write(Y);		Write(Y);		Write(Y);	
	Commit;		Abort/ Rollback;	Commit;	
Abort/ Rollback;		→ Abort;			Abort/ Rollback;

Rücksetzbarkeit

Eine Transaktion i **liest** einen Wert x **von** einer anderen Transaktion j , wenn $W_j(x)$ vor $R_i(x)$ in dem Schedule steht, und dazwischen keine andere Transaktion x schreibt:

- $W_j(x) \dots R_i(x)$ [... hier kommt kein $W_k(x)$...]

Ein Schedule ist **rücksetzbar**, wenn jede Transaktion, die einen Wert von einer anderen Transaktion liest, erst nach dieser festgeschrieben wird:

- $W_j(x) \dots C_j \dots R_i(x) \dots C_i$ oder
- $W_j(x) \dots R_i(x) \dots C_j \dots C_i$

Ein Schedule **vermeidet kaskadierende Abbrüche**, wenn jede Transaktion nur Werte von bereits festgeschriebenen Transaktionen liest:

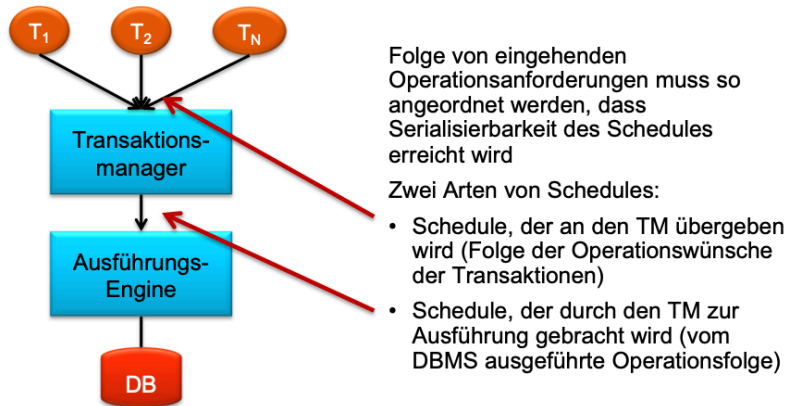
- **Ok:** $W_j(x) \dots C_j \dots R_i(x) \dots C_i$
- **Verboten:** $W_j(x) \dots R_i(x) \dots C_j \dots C_i$

Zusammenfassung

- Ein Scheduler muss die inhaltlich korrekte Ausführung von nebenläufigen Transaktionen garantieren.
- Dazu braucht er bestimmte Prüfbedingungen:
 - Allgemeine Serialisierbarkeit kann nicht sinnvoll geprüft werden
 - Daher verwendet man in der Praxis Konfliktserialisierbarkeit
- Zusätzlich muss der Scheduler die Rücksetzbarkeit garantieren, damit fehlerhafte Transaktionen abgebrochen werden können
- Weiter ist es sinnvoll, kaskadierende Abbrüche zu vermeiden

b) Synchronisationsverfahren

Aufgabe des Transaktionsmanagers



Modell: Transaktionen bestehen aus den Schritten

- **Read**
- **Write**
- **Abort**
- **Commit**

Möglichkeiten des Transaktionsmanagers:

- Operation
 - ausführen (execute) → TA running
 - verzögern (delay) → TA delayed
 - zurückweisen (reject) → TA aborted

Scheduling-Verfahren

Pessimistische Verfahren

- Greifen ein, bevor eine nicht-serialisierbare Situation entsteht
- Beispiel: **Sperrverfahren**
Es werden präventiv Sperren gesetzt, um anderen Transaktionen bestimmte Objekte vorübergehend nicht zugänglich zu machen
Transaktionen werden ggf. verzögert

Optimistische Verfahren

- Greift ein, wenn der bisher entstandene Schedule nicht serialisierbar ist
- Einziges Mittel zur Gewährleistung der Serialisierbarkeit ist der Abbruch und Neustart einer geeigneten Transaktion
- Beispiel: **Zeitstempel-Verfahren**
Testen anhand von Zeitmarken, ob das Schedule noch serialisierbar ist. Wenn nicht, wird die Transaktion abgebrochen

Sperrverfahren

Datenelemente und Sperren

Probleme entstehen durch den gleichzeitigen Zugriff mehrerer Transaktionen auf gemeinsame Datenelemente

Durch Sperren werden die Zugriffe gesteuert

Datenelement (Item): Teil der Datenbank, zu dem der Zugriff kontrolliert werden kann

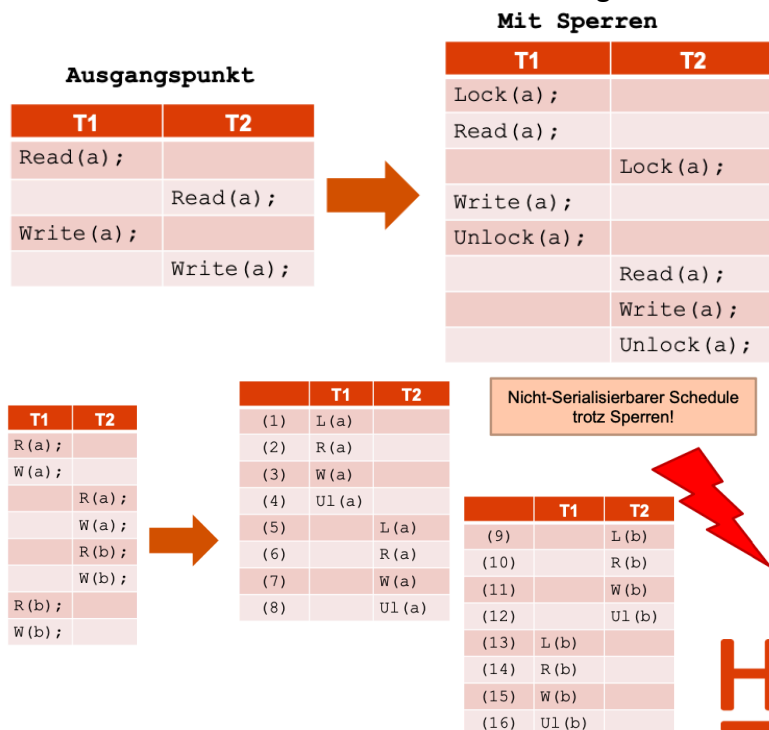
Größe der Datenelemente: Granularität

Zugriffskontrolle durch Sperren: lock - unlock

- Jede Transaktion sperrt jedes Datenelement vor dem Bearbeiten und gibt es danach wieder frei.
- Keine Transaktion darf auf ein von einer anderen Transaktion gesperrtes Element zugreifen
- Operationen: lock (x) ; und unlock (x) ;
 - sog. Binäre Sperren

Wann funktionieren Sperren?

- Erst nach unlock darf T2 weiter ausgeführt werden



Grundsätzliche Probleme beim Sperren

Sperrprotokoll

- Nach welchen Regeln müssen die Sperren gesetzt und freigegeben werden, damit serialisierbare Schedules entstehen? Einfaches Modell von oben garantiert nicht immer serialisierbare Schedules!

Sperrmodi

- Welche Arten von Sperren braucht man?

Sperreinheit

- Welches sind die sperrbaren Einheiten?

Deadlock

- Verklemmung zweier Transaktionen: Jede wartet auf die andere.

Zwei-Phasen-Sperrprotokoll


Zwei-Phasen-Sperrprotokoll (two-phase lock protocol, 2PL):

1. Vor dem ersten Zugriff auf ein Objekt muss die Transaktion das Objekt sperren
2. Nach dem ersten `unlock(X)` einer Transaktion T (auf irgendein Objekt X) darf von T kein `lock(Y)` (auf irgendein Objekt Y) mehr ausgeführt werden
3. Spätestens bei EOT (Transaktionsende) muss eine Transaktion alle ihre Sperren zurückgeben

Das Beispiel oben folgt diesem Protokoll nicht, da die Transaktionen bereits wieder Sperren freigeben, bevor sie alle jemals benötigten Sperren angefordert haben!

Einsatz von 2PL

T1		T2	
L(a);	R(a);		
	W(a);		
Ul(a);			
	L(a); R(a);		
	W(a);		
	Ul(a);		
	L(b); R(b);		
	W(b);		
	Ul(b);		
L(b); R(b);			
W(b);			
Ul(b);			



T1		T2	
L(a);	R(a);		
	W(a);		
		L(a);	
L(b); Ul(a);		...	
		R(a);	
		W(a);	
		L(b);	
	R(b);	...	
	W(b);		
Ul(b);			
		Ul(a); R(b);	
		W(b);	
		Ul(b);	

Allgemeine Variante

Sperren



Bei Anwendung des Zwei-Phasen-Sperrprotokolls kann es keinen Zyklus im Abhängigkeitsgraphen G geben!

Bedeutet: Anwendung des Zwei-Phasen-Sperrprotokolls stellt Serialisierbarkeit des Schedules sicher!

Angenommen, es gäbe einen Zyklus

$$T_i \rightarrow T_j \rightarrow \dots \rightarrow T_n \rightarrow T_i$$

Dann liegen folgende lock- und unlock-Befehle vor:

$\text{unlock}_i(X), \text{lock}_j(X), \text{unlock}_j(Y), \dots, \text{unlock}_n(Z), \text{lock}_i(Z)$

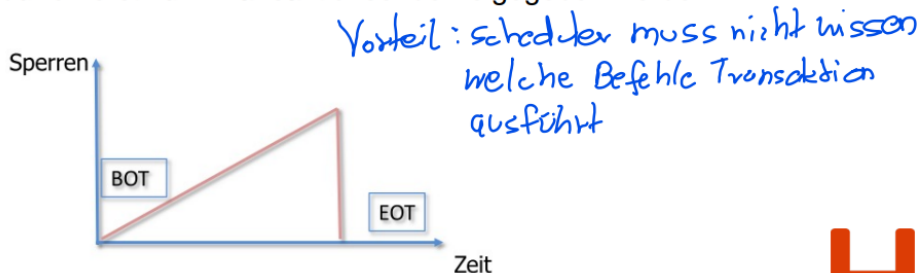
Insgesamt bedeutet dies: unlock_i vor lock_i

Widerspruch zur Annahme der Zweiphasigkeit (T_i muss alle Sperren bereits halten, wenn das erste unlock ausgeführt wird)

Strikte Zweiphasigkeit

Strikte Zweiphasigkeit:

- Alle Sperren dürfen erst zum Transaktionsende freigegeben werden:



- Motivation: Die Transaktion soll möglichst **keine kaskadierenden Abbrüche erzeugen**, da Objekte bis zum EOT gesperrt bleiben



Preclaiming

Alle Objekte, die eine Transaktion T möglicherweise verwenden wird, werden zu Beginn von T gesperrt:



Motivation: Eine Transaktion, die begonnen wird, soll mit Sicherheit zügig und ohne Warten beendet werden können

Programm müsste alle benötigten Zugriffe ankündigen!



Mehrfachmodussperren

Bisher: Binäre Sperren:

- `lock(x) ; und unlock(x) ;`
- **Beispiel:**
 - `lock(x) ; read(x) ; write(x) ; unlock(x) ;`

Nachteil: auch gleichzeitiges Lesen wird verhindert

Lösung dafür:

- Mehrfachmodussperren
- Idee: Schreib- und Lesesperre unterscheiden

Operationen für Mehrfachmodussperren

`read_lock(X) :`

- Sperrt ein Item X für den Lesezugriff
- Wartet ggf., falls Zugriff bereits gesperrt

`write_lock(X) :`

- Sperrt ein Item X für den Schreibzugriff
- Wartet ggf., falls Zugriff bereits gesperrt

`unlock(X) :`

- Gibt eine gesetzte Sperre wieder frei

Algorithmus read_lock(X)

B: `if LOCK(X) = UNLOCKED then`

`LOCK(X) := READ_LOCKED`

`No_Of_Reads(X) := 1`

`else if LOCK(X) = READ_LOCKED then`

`No_Of_Reads(X) := No_Of_Reads(X) + 1`

`else`

`do wait (until LOCK(X) != WRITE_LOCKED and the lock
 manager wakes up the transaction)`

`end wait`

`goto B`

`end if`

Algorithmus write_lock(X)

```
C: if LOCK(X) = UNLOCKED then
    LOCK(X) := WRITE_LOCKED
else
    do wait (until LOCK(X)=UNLOCKED and the
            lock manager wakes up the transaction)
    end wait
    goto C
end if
```

Algorithmus unlock(X)

```
if LOCK(X) = WRITE_LOCKED then
    LOCK(X) := UNLOCKED
    if any transactions are waiting
        then wake up one or more of the transactions
    end if
else if LOCK(X) = READ_LOCKED then
    No_Of_Reads(X) := No_Of_Reads(X) - 1
    if (No_Of_Reads(X) = 0) then
        LOCK(X) := UNLOCKED
        if any transactions are waiting
            then wake up one or more of the transactions
        end if
    end if
end if
```

Transaktionsmanagement bei Mehrfachmodussperren 1/2

System verwaltet eine Sperrtabelle

Inhalt für ein Objekt X:

- ID/Name des Datenbankobjekts

	T _i hat read_lock	T _j hat write_lock
T _i hat read_lock	✓	✗
T _i hat write_lock	✗	✗

- Wert für LOCK (UNLOCKED, READ_LOCKED, WRITE_LOCKED)

- No_Of_Reads

- ID(s) der sperrenden Transaktion(en)

- Warteschlange für Transaktionen, die auf das Objekt X warten

Kompatibilitäts- bzw. Verträglichkeitsmatrix (siehe oben rechts)



Regeln für Mehrfachmodussperren

- Jede Transaktion T muss `read_lock(X)` oder `write_lock(X)` aufrufen, bevor T ein `read(X)` ausführt
- Jede Transaktion T muss `write_lock(X)` aufrufen, bevor T ein `write(X)` ausführt
- Jede Transaktion T muss `unlock(X)` aufrufen, nachdem T alle `read(X)` und `write(X)` abgeschlossen hat
- Keine Transaktion T führt ein `read_lock(X)` aus, wenn T bereits eine (beliebige) Sperre auf Objekt X besitzt
- Keine Transaktion T führt ein `write_lock(X)` aus, wenn T bereits eine Schreibsperre auf Objekt X besitzt
- Hat T bereits eine Lesesperre und ist diese exklusiv, so ist eine **Verschärfung** auf eine Schreibsperre erlaubt; ist sie nicht exklusiv, so muss die T in die Warteschlange eingereiht werden
- Jede Transaktion T führt nur dann ein `unlock(X)` aus, wenn T eine gemeinsame Lesesperre oder exklusive Schreibsperre auf das Objekt X besitzt

Transaktionsmanagement bei Mehrfachmodussperren 2/2

- Frage: nein, weil Scheduler nicht in Zukunft sehen kann, dass danach auch geschrieben wird

Transaktionsablauf:

- `read(A); A := A + 1; write(A);`

Scheduler macht daraus:

- `read_lock(A);`
`read(A); A := A + 1;`
`write_lock(A);`
`write(A);`
`unlock(A);`

Oder besser:

- `write_lock(A);`
`read(A); A := A + 1;`
`write(A);`
`unlock(A);`

Frage: Kann der Scheduler dies umsetzen?

Ebenen von Sperren

Sperreinheiten

Granularitätshierarchien

- logische Einheiten: Attribute, Tupel, Relation, Datenbank
- physische Einheiten: Seite, Datei, Datenbank

Gegenläufige Aspekte:

- je feiner die Sperreinheit, desto mehr Parallelität zwischen Transaktionen möglich
- je feiner die Sperreinheiten, desto größer der Verwaltungsaufwand für die Sperren

Übliche Sperrebene: Tupel

- kann zusätzliche kurzzeitige Sperren physischer Objekte notwendig machen

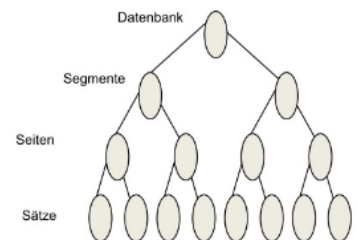
Beispiel:

- Einfügen eines Satzes in eine Tabelle erfordert Verschiebung anderer Sätze und Reorganisation einer Seite

Wahl der Sperreinheit

Möchte eine Transaktion viele Tupel verändern, so sollte die Möglichkeit bestehen, die ganze Relation zu sperren.

- Greift eine Transaktion nur auf ein einziges Tupel zu, so soll nur dieses Tupel gesperrt werden.
- Sperrhierarchie: Sperren auf unterschiedlichen Ebenen setzbar



Multiple-Granularity Locking (MGL)

Multiple-Granularity Locking (MGL):

Verwendung verschiedener Sperrgranulate

- Sperrobjekte können sich überlappen
- Entscheidung, ob ein Objekt O gesperrt werden kann, hängt auch davon ab, ob ein in O enthaltenes Objekt bereits von einer anderen Transaktion gesperrt ist

Einführung von Intentionssperren, um die flexible Auswahl eines bestimmten Sperrgranulats pro Transaktion zu ermöglichen

Wird eine Sperre auf ein Objekt O gesetzt, so muss zuvor eine Intentionssperre auf alle übergeordneten Objekte gesetzt werden

- irl (intentionale Lesesperre): weiter unten in der Hierarchie ist eine Lesesperre (rl) beabsichtigt
- iwl (intentionale Schreibsperre): weiter unten in der Hierarchie ist eine Schreibsperre (wl) beabsichtigt

Kompatibilität der Sperrmodi des MGL

Sperrung „top-down“, Freigabe „bottom-up“

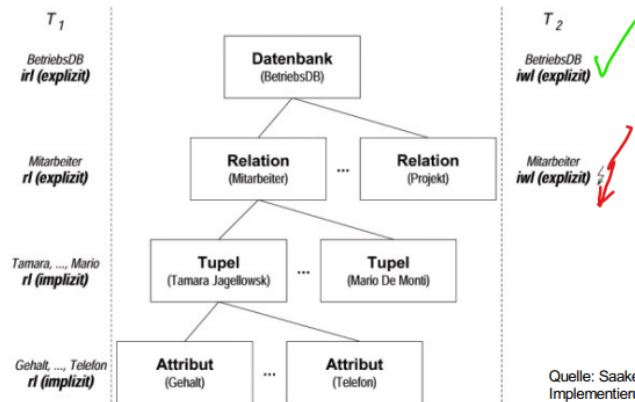
1. Sperren werden auf einem Pfad in der Reihenfolge von der Wurzel zum Zielobjekt gesetzt.
2. Das Datenobjekt, auf dem gearbeitet werden soll, wird gesperrt: Schreib- oder Lesesperre. Dabei Sperrenverträglichkeitsmatrix beachten!
3. Alle anderen Knoten auf dem Pfad bekommen intentionale Sperren.
4. Sperren können verschärft werden, das heißt ein rl kann zum wl werden, ein irl zum rl und ein irl zum iwl .
5. Die Freigabe erfolgt in umgekehrter Reihenfolge.
6. Protokolle zur Vermeidung von Konflikten auch hier erforderlich (z.B. 2PL).

	rl_i	wl_i	irl_i	iwl_i
rl_j	✓	✗	✓	✗
wl_j	✗	✗	✗	✗
irl_j	✓	✗	✓	✓
iwl_j	✗	✗	✓	✓

Hierarchisches Sperren (Beispiel 1)

T1 liest die gesamte Relation Mitarbeiter

T2 will einen Mitarbeiter aktualisieren



Probleme bei Sperrverfahren

T1	T2
WL (a) ; Read (a)	
	WL (b) ; Read (b)
Berechnungen	
WL (b) ; Read (b), muss warten auf T2	
	WL (a) ; Read (a), muss warten auf T1
Write (a)	
	Write (b)
Write (b)	
	Write (a)

- Deadlock tritt auf, wenn zwei Transaktionen jeweils auf die andere warten
- Beispiel: T1 und T2 wollen Objekte a und b verändern.



Behandlung von Deadlocks

Zyklisches Warten

- Kann nur aufgelöst werden, indem eine der beiden beteiligten Transaktionen von außen (Transaktionsmanager) gezwungen wird, ein Objekt freizugeben
- TA-Manager veranlasse bei einer der Transaktionen: **Abbruch und Rücksetzen**
- **Späterer Neustart** erforderlich (durch TA-Manager oder Client)

Erkennung von Deadlocks

Time - out: Wartezeit einer Transaktion T auf ein Objekt „zu lange“

- Transaktionsmanager schließt auf Beteiligung an Deadlock, bricht T ab
- kritisch: Wahl der Wartezeit

Wartegraph: Deadlock = Zyklus im Wartegraph

- kritisch: Prüfzeitintervalle, Auswahl der Transaktion, die abgebrochen werden soll (Kostenfunktionen)

Zeitstempelverfahren

- Transaktionen werden auf Basis der zeitlichen Reihenfolge, in der sie in das DBMS kommen, synchronisiert
- Jede Transaktion T erhält einen eindeutigen Zeitstempel $TS(T)$
(Transaktionszeitstempel siehe oben)
- Jede Operation wird mit dem Zeitstempel der Transaktion versehen
- Jedes Datenbankobjekt X besitzt zusätzlich die Zeitstempel
 - $TSR(X)$ time stamp for read = TS der zuletzt gestarteten lesenden Transaktion
 - $TSW(X)$ time stamp for write = TS der zuletzt gestarteten schreibenden Transaktion
- Mittels der Zeitstempel wird erkannt, falls eine nicht-serialisierbare Situation entsteht,
- und mit Abbruch darauf reagiert.
- Problem des Tradeoff zwischen Aufwand und Parallelität bei Wahl der Granularität für X bleibt

Timestamp Ordering

Basis – Timestamp Ordering (TO) – Algorithmus

- Fordert eine Transaktion T eine Lese- oder Schreib-Operation auf ein Objekt X an, so wird wie folgt verfahren:
- **Lese-Operation** (geht nur, wenn X noch nicht von einer später gestarteten Transaktion geschrieben wurde):

```
if TS(T) < TSW(X) then
    abort T
else
    execute read
    TSR(X) := max{TSR(X), TS(T)}
end
```

Schreib-Operation (geht nur, wenn X noch nicht von einer später gestarteten Transaktion gelesen oder geschrieben wurde):

```
if TS(T) < max{TSR(X), TSW(X)} then
    abort T
else
    execute write
    TSW(X) := TS(T)
end
```

Kaskadierender Abbruch möglich: Abbruch einer Transaktion führt zu Abbruch einer anderen Transaktion, die bereits von dieser geschriebene Wert gelesen hatte, usw.

Beispiel:

T1 ($t_s=150$)	T2 ($t_s=175$)	T3 ($t_s=200$)	Objekt A		Objekt B		Objekt C	
			TSR	TSW	TSR	TSW	TSR	TSW
Read(B)			0	0	150	0	0	0
	Read(A)		175	0	150	0	0	0
		Read(C)	175	0	150	0	200	0
Write(B)			175	0	150	150	200	0
Read(C)			175	0	150	150	200	0
	Write(C)		175	0	150	150	200	ABORT

Multi-Version Concurrency Control

MVCC

Idee: Wenn ältere Versionen eines Objektes aufgehoben werden, müssen nur Schreibzugriffe synchronisiert werden Implementierung (Achtung: dies beinhaltet nicht die Synchronisation)

- Jede Transaktion hat einen Anfangszeitstempel
- Jedes Objekt ist in mehreren Versionen vorhanden, mit Zeitstempel
- Jede Transaktion liest nur die zu ihrem Anfangszeitstempel passende Version
 - Dies bedeutet: die aktuellste Version mit einem Zeitstempel kleiner oder gleich dem Anfangszeitstempel der Transaktion

T1	T2
Read(x);	
Write(x);	
	Read(x);
	Write(y);
Read(y);	
Write(z);	

- Dieser Ablauf ist nicht konfliktserialisierbar
- Wenn T1 eine alte Version von y lesen kann (Version vor dem Write(y) von T2), ist das Problem geheilt
- Lösung wird z.B. in Oracle oder PostgreSQL verwendet
- Bedeutet aber, dass ggf. über längere Zeit mehrere Versionen eines Objektes vorgehalten werden müssen
 - Eine veraltete Version kann erst gelöscht werden, wenn alle noch aktiven Transaktionen einen neueren Start-Zeitstempel haben
 - Erfordert aufwändige Garbage Collection

c) Transaktionsmanagement in SQL und Oracle

Transaktionsverwaltung in SQL

- Normalerweise ist Transaktionsverwaltung transparent für den Nutzer des DBS
 - Manchmal ist manuelles Eingreifen erforderlich
 - Volle Serialisierbarkeit kostet Performance
- Daher: Aufweichung von ACID in SQL-Systemen: **Isolationsebenen**

```
set transaction
    [{read only | read write},]
    [isolation level
        {
            read uncommitted |
            read committed |
            repeatable read |
            serializable}]
```

Read Uncommitted

schwächste Konsistenzstufe

- darf auch nur für read only- Transaktionen spezifiziert werden.
- hat Zugriff auf noch nicht geschriebene Daten

T1	T2
	Read (A)
	...
	Write (A)
Read (A)	
...	
	Rollback

Read Committed

- Transaktionen lesen nur endgültig geschriebene Werte
- Können unterschiedliche Zustände der Datenbank-Objekte zu sehen bekommen
- Non-repeatable read kann auftreten und muss verarbeitet sein
- Beispiel – Schedule:

T1	T2
Read (A)	
	Write (A)
	Write (B)
	Commit
Read (B)	
Read (A)	
...	

Repeatable Read und Serializable

repeatable read: non-repeatable read wird ausgeschlossen

- Phantom problem kann auftreten
- Wenn eine parallele Änderungstransaktion dazu führt, dass Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.

serializable: garantiert Serialisierbarkeit

Isolationsebenen in SQL - Systemen

Isolationsebene	Dirty Read	Non-repeatable Read	Phantom Read
read uncommitted	Möglich	Möglich	Möglich
read committed	Nicht möglich	Möglich	Möglich
repeatable read	Nicht möglich	Nicht möglich	Möglich
serializable	Nicht möglich	Nicht möglich	Nicht möglich

Transaktionsverwaltung in ORACLE

Isolationsstufen read committed (default) und serializable, zusätzlich read only

- set transaction isolation level ...
- set transaction read only

Isolationslevel für jede Transaktion einstellbar oder für eine Menge von Transaktionen

- alter session set isolation_level ...

Schreibsperren-Verwaltung für Tables und Rows

Explizite Kommandos zum Setzen von Sperren möglich

- **SELECT * FROM movie WHERE movie = 123456
FOR UPDATE;**

Multi-Version Concurrency Control

- Dadurch im Normalbetrieb keine Lesesperren nötig

Anwendungsbeispiele


```
conn.setAutoCommit(false);
...
try {
    PreparedStatement prep = conn.prepareStatement
        ("UPDATE KONTO SET kontostand = kontostand - ? WHERE id = ?");
    prep.setFloat(1, amount); prep.setInt(2, from);
    if (prep.executeUpdate() != 1)
        throw new Exception("Konto " + from + " wurde nicht gefunden!");

    prep = conn.prepareStatement
        ("UPDATE KONTO SET kontostand = kontostand + ? WHERE id = ?");
    prep.setFloat(1, amount);
    prep.setInt(2, to);
    if (prep.executeUpdate() != 1)
        throw new Exception("Konto " + to + " wurde nicht gefunden!");

    conn.commit();
} catch (Exception e) {
    conn.rollback();
    //...
}
```

Was mache ich, wenn der Kontostand nicht negativ werden darf?

```
SELECT kontostand FROM KONTO WHERE id = :from;

Prüfung, ob Kontostand minus Abbuchungsbetrag < 0 ist
Wenn < 0
    ROLLBACK
    Abbrechen
Sonst
     andere Transaktion verringert Konto "from"

    Update wie bisher durchführen
```

Lösung 1: Isolationsebene erhöhen.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT kontostand FROM KONTO WHERE id = :from;

Prüfung, ob der Kontostand minus Abbuchungsbetrag < 0 ist
Wenn < 0
    ROLLBACK
    Abbrechen
Sonst
    Update wie bisher durchführen
```

Lösung 2: Sperren verwenden

```
SELECT kontostand FROM KONTO WHERE id = :from
    FOR UPDATE [NOWAIT];

Prüfung, ob der Kontostand minus Abbuchungsbetrag < 0 ist
Wenn < 0
    ROLLBACK
    Abbrechen
Sonst
    Update wie bisher durchführen
```

Zusammenfassung

- Transaktionen fassen eine zusammenhängende Menge von Basisoperationen auf einer Datenbank zusammen
- Transaktionen genügen dem ACID-Prinzip
- Durch parallele Ausführung von Transaktionen in einem DBS können Synchronisationsprobleme entstehen
- Theoretische Lösung der Synchronisationsprobleme durch serialisierbare Schedules
- Prüfung auf (allgemeine) Serialisierbarkeit unmöglich
 - Daher wird Konfliktserialisierbarkeit verwendet
- Konkret werden meist Sperrverfahren verwendet
 - Unter Nutzung des 2-Phasen-Sperrprotokolls
 - Meist kombiniert mit MVCC