



**Hasso
Plattner
Institut**

IT Systems Engineering | Universität Potsdam

Datenbanksysteme I
Transaktionsmanagement

24.6.2009

Felix Naumann

Motivation - Transaktionsmanagement

2

Annahmen bisher

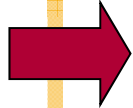
■ Isolation

- Nur ein Nutzer greift auf die Datenbank zu
 - ◇ Lesend
 - ◇ Schreibend
- In Wahrheit: Viele Nutzer und Anwendungen lesen und schreiben gleichzeitig.

■ Atomizität

- Anfragen und Updates bestehen aus einer einzigen, atomaren Aktion.
 - ◇ DBMS können nicht mitten in dieser Aktion ausfallen.
- In Wahrheit: Auch einfache Anfragen bestehen oft aus mehreren Teilschritten.

3



- Transaktionen
- Isolationsebenen
- Serialisierbarkeit
- Konfliktserialisierbarkeit
- Sperrprotokolle
- Sperren



Die Transaktion

4

- Eine Transaktion ist eine **Folge von Operationen** (Aktionen), die die Datenbank von einem **konsistenten Zustand** in einen konsistenten (eventuell veränderten) Zustand überführt, wobei das **ACID-Prinzip** eingehalten werden muss.
- Aspekte:
 - Semantische Integrität: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
 - Ablaufintegrität: Fehler durch „gleichzeitigen“ Zugriff mehrerer Benutzer auf dieselben Daten vermeiden

Transaktionen – Historie

5

- In alten DBMS kein Formalismus über Transaktionen
 - Nur Tricks
- Erster Formalismus in den 80ern
 - *System R* unter Jim Gray
 - Erste (ineffiziente) Implementierung
 - ACM Turing Award
 - ◇ *For seminal contributions to database and transaction processing research and technical leadership in system implementation from research prototypes to commercial products.*
- ARIES Project (IBM Research)
 - Alle Details
 - Effiziente Implementierungen
 - C. Mohan
- Transaktionen auch in verteilten Anwendungen und Services



- **Atomicity (Atomarität)**
 - Transaktion wird entweder ganz oder gar nicht ausgeführt.
- **Consistency (Konsistenz oder auch Integritätserhaltung)**
 - Datenbank ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand.
- **Isolation (Isolation)**
 - Transaktion, die auf einer Datenbank arbeitet, sollte den „Eindruck“ haben, dass sie allein auf dieser Datenbank arbeitet.
- **Durability (Dauerhaftigkeit / Persistenz)**
 - Nach erfolgreichem Abschluss einer Transaktion muss das Ergebnis dieser Transaktion „dauerhaft“ in der Datenbank gespeichert werden.



Beispielszenarien

7

- Platzreservierung für Flüge gleichzeitig aus vielen Reisebüros
 - Platz könnte mehrfach verkauft werden, wenn mehrere Reisebüros den Platz als verfügbar identifizieren.

- Überschneidende Konto-Operationen einer Bank
 - Beispiele später

- Statistische Datenbankoperationen
 - Ergebnisse sind verfälscht, wenn während der Berechnung Daten geändert werden.

Beispiel – Serialisierbarkeit

8

- Fluege(Flugnummer, Datum, Sitz, besetzt)
- chooseSeat() sucht nach freiem Platz und besetzt ihn gegebenenfalls
 - EXEC SQL BEGIN DECLARE SECTION;


```
int flug;
char date[10];
char seat[3];
int occ;
```
 - EXEC SQL END DECLARE SECTION;
 - void chooseSeat() {


```
/* Nutzer nach Flug, Datum und Sitz fragen */
EXEC SQL SELECT besetzt INTO :occ FROM Flights
WHERE Flugnummer=:flug
AND Datum=:date AND Sitz=:seat;
```
 - if(!occ) {


```
EXEC SQL UPDATE Fluege SET besetzt=TRUE
WHERE Flugnummer=:flight AND Datum=:date AND Sitz=:seat;
```
 - }
 - else ...
 - }



Embedded SQL

Beispiel – Serialisierbarkeit

9

- Problem: Funktion wird von mehreren Nutzern zugleich aufgerufen

Nutzer 1 findet leeren Platz	
	Nutzer 2 findet leeren Platz
Nutzer 1 besetzt Platz	
	Nutzer 2 besetzt Platz

„Schedule“

- Beide Nutzer glauben den Platz reserviert zu haben.
- Lösung gleich
 - Serielle Schedules
 - Serialisierbare Schedules

Beispiel - Atomizität

10

- Problem: Eine Operation kann Datenbank in inkonsistenten Zustand hinterlassen.

- Softwarefehler / Hardwarefehler

- EXEC SQL BEGIN SECTION;

```
int konto1, konto2;
```

```
int kontostand;
```

```
int betrag;
```

```
EXEC SQL END DECLARE SECTION;
```

```
void transfer() {
```

```
/* Nutzer nach Konto1, Konto2 und Betrag fragen */
```

```
EXEC SQL SELECT Stand INTO :kontostand FROM Konten
```

```
WHERE KontoNR=:konto1;
```

```
If (kontostand >= betrag) {
```

```
EXEC SQL UPDATE Konten
```

```
SET Stand=Stand+:betrag
```

```
WHERE KontoNR=:konto2;
```

```
EXEC SQL UPDATE Konten
```

```
SET Stand=Stand-:betrag
```

```
WHERE KontoNR=:konto1;
```

```
} else /* Fehlermeldung */
```

```
}
```

Problem: System-absturz hier

Lösung: Atomizität

Probleme im Mehrbenutzerbetrieb

11

- Inkonsistentes Lesen: *Nonrepeatable Read*
- Abhängigkeiten von nicht freigegebenen Daten: *Dirty Read*
- Berechnungen auf unvollständigen Daten: *Phantom-Problem*
- Verlorengesangenes Ändern: *Lost Update*

Dirty Read

12

T ₁	T ₂
<code>read(A,x)</code>	
<code>x:=x+100</code>	
<code>write(x,A)</code>	
	<code>read(A,x)</code>
	<code>read(B,y)</code>
	<code>y:=y+x</code>
	<code>write(y,B)</code>
	<code>commit</code>
<code>abort</code>	

Problem: T₂ liest den veränderten A-Wert, diese Änderung ist aber nicht endgültig, sondern sogar ungültig.

Nonrepeatable Read

13

- Nicht-wiederholbares Lesen
- Beispiel:
 - Zusicherung: $x = A + B + C$ am Ende der Transaktion T_1
 - x, y, z seien lokale Variablen

Problem: A hat sich im Laufe der Transaktion geändert. $x = A + B + C$ gilt nicht mehr

T_1	T_2
read(A,x)	
	read(A,y)
	y:=y/2
	write(y,A)
	read(C,z)
	z:=z+y
	write(z,C)
	commit
read(B,y)	
x:=x+y	
read(C,z)	
x:=x+z	
commit	

Das Phantom-Problem

14

T ₁	T ₂
<pre>SELECT COUNT(*) INTO X FROM Mitarbeiter</pre>	
	<pre>INSERT INTO Mitarbeiter VALUES (,Meier`, 50000, ...)</pre>
	<pre>commit</pre>
<pre>UPDATE Mitarbeiter SET Gehalt = Gehalt +10000/X</pre>	
<pre>commit</pre>	

Problem: Meier geht nicht in die Gehaltsberechnung ein. Meier ist das Phantom.

Lost Update

15

T ₁	T ₂	A
read(A,x)		10
	read(A,x)	10
x:=x+1		10
	x:=x+1	10
write(x,A)		11
	write(x,A)	11

Problem: Die Erhöhung von T₁ wird nicht berücksichtigt.

Transaktionen in SQL

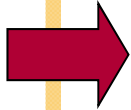
16

Idee: Gruppierung mehrerer Operationen / Anfragen in eine Transaktion

- Ausführung atomar (per Definition)
- Ausführung serialisierbar (per SQL Standard)
 - Kann aufgeweicht werden (Isolationsstufen)
- Ein SQL Befehl entspricht einer Transaktion
 - Ausgelöste **TRIGGER** werden ebenfalls innerhalb der Transaktion ausgeführt.
- Beginn einer Transaktion: **START TRANSACTION**
- Ende einer Transaktion (falls mit **START TRANSACTION** gestartet)
 - **COMMIT** signalisiert erfolgreiches Ende der Transaktion
 - **ROLLBACK** signalisiert Scheitern der Transaktion
 - ◇ Erfolgte Änderungen werden rückgängig gemacht.
 - ◇ Kann auch durch das DBMS ausgelöst werden: Anwendung muss entsprechende Fehlermeldung erkennen.

17

- Transaktionen
- Isolationsebenen
- Serialisierbarkeit
- Konfliktserialisierbarkeit
- Sperrprotokolle
- Sperren



Transaktionen

18

- Aufweichung von ACID in SQL-92: Isolationsebenen
 - `set transaction`
 - [{ `read only` | `read write` },]
 - [`isolation level` {
 - `read uncommitted` |
 - `read committed` |
 - `repeatable read` |
 - `serializable` },]
 - [`diagnostics size ...`]
- Kann pro Transaktion angegeben werden
- Standardeinstellung:
 - `set transaction read write,`
`isolation level serializable`
- Andere Ebenen als Hilfestellung für das DBMS zur Effizienzsteigerung

Bedeutung der Isolationsebenen

19

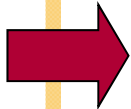
- **read uncommitted**
 - Schwächste Stufe: Zugriff auf nicht geschriebene Daten
 - Falls man schreiben will: **read write** angeben (default ist hier ausnahmsweise **read only**)
 - Statistische und ähnliche Transaktionen (ungefährer Überblick, nicht korrekte Werte)
 - Keine Sperren: Effizient ausführbar, keine anderen Transaktionen werden behindert
- **read committed**
 - Nur Lesen endgültig geschriebener Werte, aber nonrepeatable read möglich
- **repeatable read**
 - Kein nonrepeatable read, aber Phantomproblem kann auftreten
- **serializable**
 - Garantierte Serialisierbarkeit (default)
 - Transaktion sieht nur Änderungen, die zu Beginn der Transaktion committed waren (und eigene Änderungen).

Isolationsebenen

20

Isolationsebene	Dirty Read	Nonrepeatable Read	Phantom Read	Lost Update
Read Uncommitted	+	+	+	+
Read Committed	-	+	+	-
Repeatable Read	-	-	+	-
Serializable	-	-	-	-

- Transaktionen
- Isolationsebenen
- Serialisierbarkeit
- Konfliktserialisierbarkeit
- Sperrprotokolle
- Sperren



Seriell vs. Parallel

22

- Grundannahme: Korrektheit
 - Jede Transaktion, isoliert ausgeführt auf einem konsistenten Zustand der Datenbank, hinterlässt die Datenbank wiederum in einem konsistenten Zustand.

- Lösung aller obigen Probleme
 - Alle Transaktionen seriell ausführen.
- Aber: Parallele Ausführung bietet Effizienzvorteile
 - „Long-Transactions“ über mehrere Stunden hinweg
 - Cache ausnutzen
- Deshalb: Korrekte parallele Pläne (*Schedules*) finden
 - Korrekt = Serialisierbar

Schedules

23

- Ein Schedule ist eine geordnete Abfolge wichtiger Aktionen, die von einer oder mehreren Transaktionen durchgeführt werden.
 - Wichtige Aktionen: READ und WRITE eines Elements
 - „Ablaufplan“ für Transaktion, bestehend aus Abfolge von Transaktionsoperationen

Schritt	T ₁	T ₃
1.	BOT	BOT
2.	read(A,a ₁)	read(A,a ₂)
3.	a ₁ := a ₁ - 50	a ₂ := a ₂ - 100
4.	write(A,a ₁)	write(A,a ₂)
5.	read(B,b ₁)	read(B,b ₂)
6.	b ₁ := b ₁ + 50	b ₂ := b ₂ + 100
7.	write(B,b ₁)	write(B,b ₂)
8.	commit	commit

Serialisierbarkeit

24

- Schedule
 - „Ablaufplan“ für Transaktion, bestehend aus Abfolge von Transaktionsoperationen
- Serieller Schedule
 - Schedule in dem Transaktionen hintereinander ausgeführt werden
- Serialisierbarer Schedule
 - Schedule dessen Effekt identisch zum Effekt eines (beliebig gewählten) seriellen Schedules ist.

Schedules

Serieller Schedule

Serialisierbarer Schedule

25

Schritt	T ₁	T ₂
1.	BOT	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	commit	
7.		BOT
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		commit

Schritt	T ₁	T ₂
1.	BOT	
2.	read(A)	
3.		BOT
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	commit	
10.		read(A)
11.		write(A)
12.		commit

Schedules

26

Serialisierbar?

Schritt	T ₁	T ₃
1.	BOT	
2.	read(A)	
3.	write(A)	
4.		BOT
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		commit
10.	read(B)	
11.	write(B)	
12.	commit	

Aufgabe: Suche äquivalenten
seriellen Schedule.

Schritt	T ₁	T ₃
1.	BOT	
2.	read(A,a ₁)	
3.	a ₁ := a ₁ - 50	
4.	write(A,a ₁)	
5.		BOT
6.		read(A,a ₂)
7.		a ₂ := a ₂ - 100
8.		write(A,a ₂)
9.		read(B,b ₂)
10.		b ₂ := b ₂ + 100
11.		write(B,b ₂)
12.		commit
13.	read(B,b ₁)	
14.	b ₁ := b ₁ + 50	
15.	write(B,b ₁)	
16.	commit	

Schritt	T ₁	T ₃
1.	BOT	
2.	read(A,a ₁)	
3.	a ₁ := a ₁ - 50	
4.	write(A,a ₁)	
5.	read(B,b ₁)	
6.	b ₁ := b ₁ + 50	
7.	write(B,b ₁)	
8.	commit	
9.		BOT
10.		read(A,a ₂)
11.		a ₂ := a ₂ - 100
12.		write(A,a ₂)
13.		read(B,b ₂)
14.		b ₂ := b ₂ + 100
15.		write(B,b ₂)
16.		commit

Effekt: $A = A - 150, B = B + 150$

Effekt: $A = A - 150, B = B + 150$

	T ₁	T ₃	A	B
1.	BOT		100	100
2.	read(A,a ₁)			
3.	a ₁ := a ₁ - 50		50	
4.	write(A,a ₁)			
5.		BOT		
6.		read(A,a ₂)		
7.		a ₂ := a ₂ * 1.03	51,5	
8.		write(A,a ₂)		
9.		read(B,b ₂)		
10.		b ₂ := b ₂ * 1.03		103
11.		write(B,b ₂)		
12.		commit		
13.	read(B,b ₁)			
14.	b ₁ := b ₁ + 50			153
15.	write(B,b ₁)			
16.	commit			

	T ₁	T ₃	A	B
1.	BOT		100	100
2.	read(A,a ₁)			
3.	a ₁ := a ₁ - 50		50	
4.	write(A,a ₁)			
5.	read(B,b ₁)			
6.	b ₁ := b ₁ + 50			150
7.	write(B,b ₁)			
8.	Commit			
9.		BOT		
10.		read(A,a ₂)		
11.		a ₂ := a ₂ * 1.03	51,5	
12.		write(A,a ₂)		
13.		read(B,b ₂)		
14.		b ₂ := b ₂ * 1.03		154,5
15.		write(B,b ₂)		
16.		commit		

*Effekt: A = (A - 50) * 1.03
B = B * 1.03 + 50*

*Effekt: A = (A - 50) * 1.03
B = (B + 50) * 1.03*

	T ₁	T ₃	A	B
1.	BOT		100	100
2.	read(A,a ₁)			
3.	a ₁ := a ₁ - 50		50	
4.	write(A,a ₁)			
5.		BOT		
6.		read(A,a ₂)		
7.		a ₂ := a ₂ * 1.03	51,5	
8.		write(A,a ₂)		
9.		read(B,b ₂)		
10.		b ₂ := b ₂ * 1.03		103
11.		write(B,b ₂)		
12.		commit		
13.	read(B,b ₁)			
14.	b ₁ := b ₁ + 50			153
15.	write(B,b ₁)			
16.	commit			

*Effekt: A = (A - 50) * 1.03
B = B * 1.03 + 50*

	T ₁	T ₃	A	B
1.		BOT	100	100
2.		read(A,a ₂)		
3.		a ₂ := a ₂ * 1.03	103	
4.		write(A,a ₂)		
5.		read(B,b ₂)		
6.		b ₂ := b ₂ * 1.03		103
7.		write(B,b ₂)		
8.		commit		
9.	BOT			
10.	read(A,a ₁)			
11.	a ₁ := a ₁ - 50		53	
12.	write(A,a ₁)			
13.	read(B,b ₁)			
14.	b ₁ := b ₁ + 50			153
15.	write(B,b ₁)			
16.	Commit			

*Effekt: A = A * 1.03 - 50
B = B * 1.03 + 50*

Schedules

30

Serialisierbar?
Nein,
denn Effekt
entspricht weder
dem seriellen
Schedule T_1T_3
noch dem
seriellen
Schedule T_3T_1

Schritt	T_1	T_3
1.	BOT	
2.	read(A)	
3.	write(A)	
4.		BOT
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		commit
10.	read(B)	
11.	write(B)	
12.	commit	

Serialisierbar?
Nein,
obwohl es
konkrete Beispiele
solcher
Transaktionen
gibt, für die es
einen
äquivalenten
seriellen Schedule
gibt. Man nimmt
immer das
Schlimmste an.

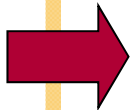
Nochmal die beiden seriellen Schedules. Ist Ihnen etwas aufgefallen?

HPI Hasso

	T ₁	T ₃	A	B		T ₁	T ₃	A	B
1.		BOT	100	100	1.	BOT		100	100
2.		read(A,a ₂)			2.	read(A,a ₁)			
3.		a ₂ := a ₂ * 1.03	103		3.	a ₁ := a ₁ - 50		50	
4.		write(A,a ₂)			4.	write(A,a ₁)			
5.		read(B,b ₂)			5.	read(B,b ₁)			
6.		b ₂ := b ₂ * 1.03		103	6.	b ₁ := b ₁ + 50			150
7.		write(B,b ₂)			7.	write(B,b ₁)			
8.		commit			8.	Commit			
9.	BOT				9.		BOT		
10.	read(A,a ₁)				10.		read(A,a ₂)		
11.	a ₁ := a ₁ - 50		53		11.		a ₂ := a ₂ * 1.03	51,5	
12.	write(A,a ₁)				12.		write(A,a ₂)		
13.	read(B,b ₁)				13.		read(B,b ₂)		
14.	b ₁ := b ₁ + 50			153	14.		b ₂ := b ₂ * 1.03		154,5
15.	write(B,b ₁)				15.		write(B,b ₂)		
16.	Commit				16.		commit		

T₁T₃ ≠ T₃T₁ Ist das schlimm?

- Transaktionen
- Isolationsebenen
- Serialisierbarkeit
- Konfliktserialisierbarkeit
- Sperrprotokolle
- Sperren



Konflikte

33

- Bedingung für Serialisierbarkeit: „Konfliktserialisierbarkeit“
- Konfliktserialisierbarkeit wird von den meisten DBMS verlangt (und hergestellt)
- Konflikt herrscht zwischen zwei Aktionen eines Schedules falls die Änderung ihrer Reihenfolge das Ergebnis verändern kann.
 - „Kann“ - nicht muss.
- Neue **Notation**: Aktion $r_i(X)$ bzw. $w_i(X)$
 - read bzw. write
 - TransaktionsID i
 - Datenbankelement X
- **Transaktion** ist eine Sequenz solcher Aktionen
 - $r_1(A)w_1(A)r_1(B)w_1(B)$
- **Schedule** einer Menge von Transaktionen ist eine Sequenz von Aktionen
 - Alle Aktionen aller Transaktionen müssen enthalten sein
 - Aktionen einer Transaktion erscheinen in gleicher Reihenfolge im Schedule

Konflikte

34

Gegeben Transaktionen T_i und T_k

- $r_i(X)$ und $r_k(X)$ stehen nicht in Konflikt
- $r_i(X)$ und $w_k(Y)$ stehen nicht in Konflikt (falls $X \neq Y$)
- $w_i(X)$ und $r_k(Y)$ stehen nicht in Konflikt (falls $X \neq Y$)
- $w_i(X)$ und $w_k(Y)$ stehen nicht in Konflikt (falls $X \neq Y$)
- $r_i(X)$ und $w_k(X)$ stehen in Konflikt
- $w_k(X)$ und $r_i(X)$ stehen in Konflikt
- $w_i(X)$ und $w_k(X)$ stehen in Konflikt
 - „No coincidences“: Man nimmt immer das schlimmste an. Die Konkrete Ausprägung der write-Operationen ist egal.

Zusammengefasst: Konflikt herrscht falls zwei Aktionen

- das gleiche Datenbankelement betreffen,
- und mindestens eine der beiden Aktionen ein *write* ist.

Konflikt – konfligieren

Wikiwörterbuch

Wiktionary
 ['vɪkʃəˌnɛʀi], *n*
 Das freie Wörterbuch
 ein Wiki-basiertes
 freies Wörterbuch

Navigation

- Hauptseite
- Themenportale
- Zufälliger Eintrag
- Inhaltsverzeichnis

Mitarbeit

- Wiktionary-Portal
- Wunschliste
- Literaturliste
- Letzte Änderungen

Hilfe

- Hilfe
- Spenden
- Nutzungshinweise
- Botschaft - embassy

Suche

Eintrag Diskussion Seite bearbeiten Versionen/Autoren

konfligieren

Aus Wiktionary, dem freien Wörterbuch

konfligieren (Deutsch) [bearbeiten]

Verb [bearbeiten]

Silbentrennung: kon-flig-ie-ren

Aussprache:
 IPA: [ˌkɔnflɪˈɡiːrən]
 Hörbeispiele: -

Bedeutungen:
 [1] widerstreiten, in Widerspruch stehen

Herkunft:
 von lateinisch: *cōnfligere*

Synonyme:
 [1] *widerstreiten*, *widersprechen*, *sich beißen*

Unterbegriffe:

Beispiele:
 [1] Meine Interessen *konfligieren* mit den Ihren!

Charakteristische Wortkombinationen:
 [1] *konfligieren mit*

Abgeleitete Begriffe:
Konflikt

Konfliktserialisierbarkeit

36

- Idee: So lange nicht-konfligierende Aktionen tauschen bis aus einem Schedule ein serieller Schedule wird.
 - Falls das klappt ist der Schedule serialisierbar.
- Zwei Schedules S und S' heißen *konfliktäquivalent*, wenn die Reihenfolge aller Paare von konfligierenden Aktionen in beiden Schedules gleich ist.
- Ein Schedule S ist genau dann *konfliktserialisierbar*, wenn S konfliktäquivalent zu einem seriellen Schedule ist.
- Schedule:

$$r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$$
- Serieller Schedule T_1T_2 :

$$r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$$
- Serieller Schedule T_2T_1 :

$$r_2(A)w_2(A)r_2(B)w_2(B)r_1(A)w_1(A)r_1(B)w_1(B)$$

Konfliktserialisierbarkeit \Rightarrow Serialisierbarkeit

- Beispiel zuvor: Serialisierbarkeit war „zufällig“ aufgrund spezieller Ausprägungen der Aktionen möglich.
- S1: $w_1(Y)w_1(X)w_2(Y)w_2(X)w_3(X)$
 - Ist seriell
- S2: $w_1(Y)w_2(Y)w_2(X)w_1(X)w_3(X)$
 - Hat (zufällig) gleichen Effekt wie S1, ist also serialisierbar.
 - Aber: Es müssten konfligierende Aktionen getauscht werden.
 - ◇ Welche?
 - S2 ist also nicht konfliktserialisierbar
- Was fällt auf?
 - T_3 überschreibt X sowieso \rightarrow Sichtserialisierbarkeit in DBS II

Graphbasierter Test

38

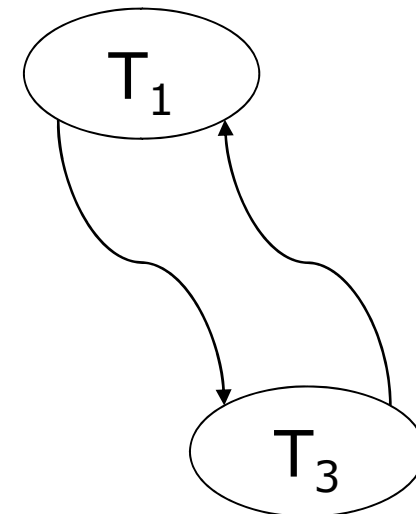
- Konfliktgraph $G(S) = (V, E)$ von Schedule S :
 - Knotenmenge V enthält alle in S vorkommende Transaktionen.
 - Kantenmenge E enthält alle gerichteten Kanten zwischen zwei konfligierenden Transaktionen.
 - ◇ Kantenrichtung entspricht zeitlichem Ablauf im Schedule.
- Eigenschaften
 - S ist ein konfliktserialisierbarer Schedule gdw. der vorliegende Konfliktgraph ein azyklischer Graph ist.
 - Für jeden azyklischen Graphen $G(S)$ lässt sich ein serieller Schedule S' konstruieren, so dass S konfliktäquivalent zu S' ist.
 - ◇ D.h. S ist konfliktserialisierbar
 - ◇ Z.B. topologisches Sortieren
 - Enthält der Graph Zyklen, ist der zugehörige Schedule nicht konfliktserialisierbar.

Schedules

39

Konflikt-serialisierbar?

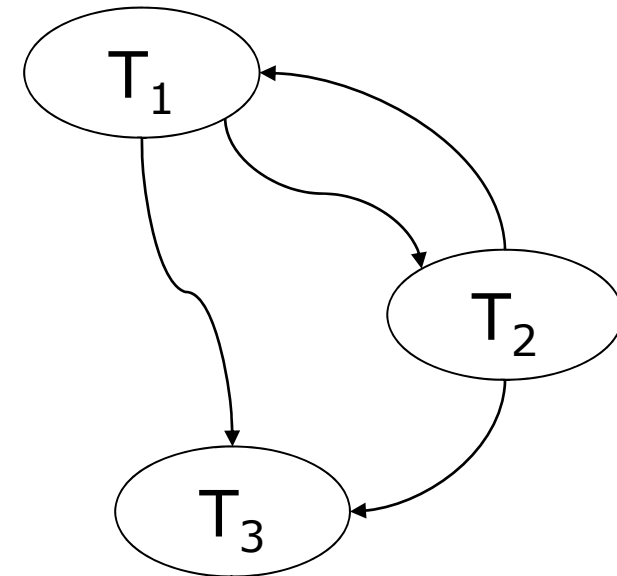
Schritt	T ₁	T ₃
1.	BOT	
2.	read(A)	
3.	write(A)	
4.		BOT
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		commit
10.	read(B)	
11.	write(B)	
12.	commit	



Schedules

40

Schritt	T ₁	T ₂	T ₃
1.	r(y)		
2.			r(u)
3.		r(y)	
4.	w(y)		
5.	w(x)		
6.		w(x)	
7.		w(z)	
8.			w(x)

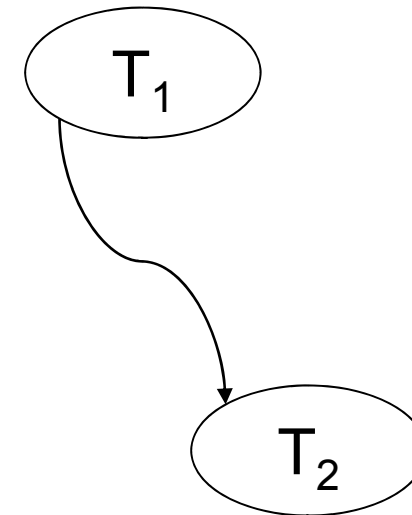


$$S = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)$$

Schedules

41

Schritt	T ₁	T ₂
1.	BOT	
2.	read(A)	
3.		BOT
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	commit	
10.		read(A)
11.		write(A)
12.		commit



Serialisierbarer Schedule

Beweis

42

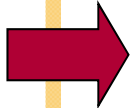
Konfliktgraph ist zyklfrei \Leftrightarrow Schedule ist konfliktserialisierbar

- Konfliktgraph ist zyklfrei \Leftarrow Schedule ist konfliktserialisierbar
 - Leicht: Konfliktgraph hat Zykel \Rightarrow Schedule ist nicht konfliktserialisierbar
 - $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$
- Konfliktgraph ist zyklfrei \Rightarrow Schedule ist konfliktserialisierbar
 - Induktion über Anzahl der Transaktionen n
 - $n = 1$: Graph und Schedule haben nur eine Transaktion.
 - $n = n + 1$:
 - ◇ Graph ist zyklfrei
 - \Rightarrow mindestens ein Knoten T_i ohne eingehende Kante
 - \Rightarrow es gibt keine Aktion einer anderen Transaktion, die vor einer Aktion in T_i ausgeführt wird und mit dieser Aktion in Konflikt steht
 - ◇ Alle Aktionen aus T_i können an den Anfang bewegt werden (Reihenfolge innerhalb T_i bleibt erhalten).
 - ◇ Restgraph ist wieder azyklisch (Entfernung von Kanten aus einem azyklischen Graph kann ihn nicht zyklisch machen).
 - ◇ Restgraph hat $n-1$ Transaktionen

Überblick

43

- Transaktionen
- Isolationsebenen
- Serialisierbarkeit
- Konfliktserialisierbarkeit
- Sperrprotokolle
- Sperren

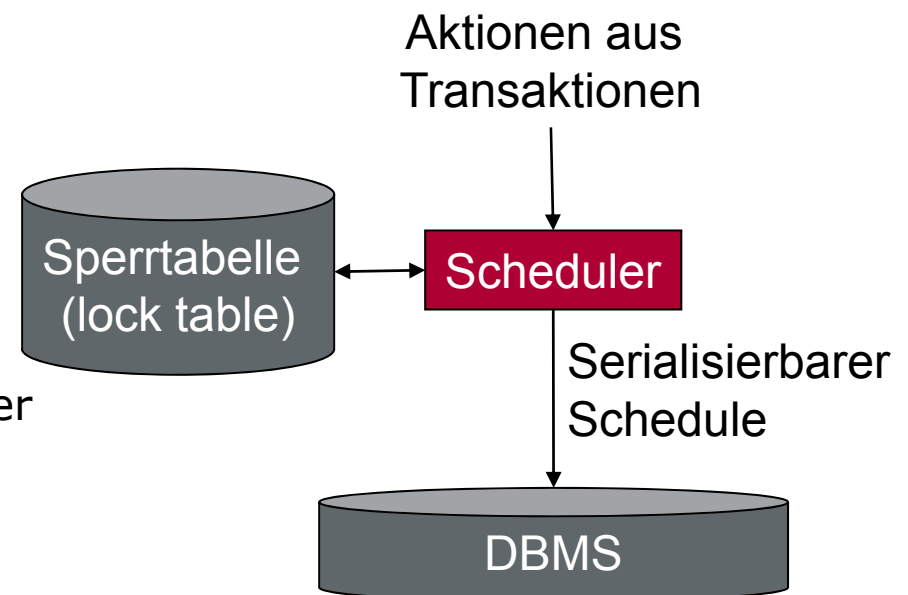


Scheduler

44

Der Scheduler in einem DBMS garantiert konfliktserialisierbare (also auch serialisierbare) Schedules bei gleichzeitig laufenden Transaktionen.

- Komplizierte Variante: Graphbasierter Test
 - Inkrementell?
- Einfachere Variante: Sperren und Sperrprotokolle
 - In fast allen DBMS realisiert
- Idee: Transaktion sperrt Objekte der Datenbank für die Dauer der Bearbeitung
 - Andere Transaktionen können nicht auf gesperrte Objekte zugreifen.



Sperrern

45

Idee: Transaktionen müssen zusätzlich zu den Aktionen auch Sperren anfordern und freigeben.

- Bedingungen
 - **Konsistenz** einer Transaktion
 - ◇ Lesen oder Schreiben eines Objektes nur nachdem Sperre angefordert wurde und bevor die Sperre wieder freigegeben wurde.
 - ◇ Nach dem Sperren eines Objektes muss später dessen Freigabe erfolgen.
 - **Legalität** des Schedules
 - ◇ Zwei Transaktionen dürfen nicht gleichzeitig das gleiche Objekt sperren.
- Zwei neue Aktionen
 - $l_i(X)$: Transaktion i fordert Sperre für X an (*lock*).
 - $u_i(X)$: Transaktion i gibt Sperre auf X frei (*unlock*).
- Konsistenz: Vor jedem $r_i(X)$ oder $w_i(X)$ kommt ein $l_i(X)$ (mit keinem $u_i(X)$ dazwischen) und ein $u_i(X)$ danach.
- Legalität: Zwischen $l_i(X)$ und $l_k(X)$ kommt immer ein $u_i(X)$

Schedules mit Sperren

46

- Zwei Transaktionen
 - $r_1(A)w_1(A)r_1(B)w_1(B)$
 - $r_2(A)w_2(A)r_2(B)w_2(B)$
- Zwei Transaktionen mit Sperren
 - $l_1(A)r_1(A)w_1(A)u_1(A)l_1(B)r_1(B)w_1(B)u_1(B)$
 - $l_2(A)r_2(A)w_2(A)u_2(A)l_2(B)r_2(B)w_2(B)u_2(B)$
 - Konsistent?
- Schedule
 - $l_1(A)r_1(A)w_1(A)u_1(A)$
 - $l_2(A)r_2(A)w_2(A)u_2(A)$
 - $l_2(B)r_2(B)w_2(B)u_2(B)$
 - $l_1(B)r_1(B)w_1(B)u_1(B)$
 - Legal?
 - Konfliktserialisierbar?

Schedules mit Sperren

47

T ₁	T ₃	A	B
		25	25
l(A); read(A,a ₁)			
a ₁ := a ₁ + 100		125	
write(A,a ₁); u(A)			
	l(A); read(A,a ₂)		
	a ₂ := a ₂ * 2	250	
	write(A,a ₂); u(A)		
	l(B); read(B,b ₂)		
	b ₂ := b ₂ * 2		50
	write(B,b ₂); u(B)		
l(B); read(B,b ₁)			
b ₁ := b ₁ + 100			150
write(B,b ₁); u(B)			

Legal?
Serialisierbar?
Konfliktserialisierbar?

Freigabe durch Scheduler

48

Scheduler speichert Sperrinformation in Sperrtabelle

- **Sperren(Element, Transaktion)**
- Anfragen, INSERT, DELETE
- Vergabe von Sperren nur wenn keine andere Sperre existiert
- Alte Transaktionen
 - $I_1(A)r_1(A)w_1(A)u_1(A)I_1(B)r_1(B)w_1(B)u_1(B)$
 - $I_2(A)r_2(A)w_2(A)u_2(A)I_2(B)r_2(B)w_2(B)u_2(B)$
- Neue Transaktionen
 - $I_1(A)r_1(A)w_1(A)I_1(\mathbf{B})u_1(\mathbf{A})r_1(B)w_1(B)u_1(B)$
 - $I_2(A)r_2(A)w_2(A)I_2(\mathbf{B})u_2(\mathbf{A})r_2(B)w_2(B)u_2(B)$
 - Konsistent?

Schedules mit Sperren

49

T ₁	T ₃	A	B
		25	25
l(A); read(A,a ₁)			
a ₁ := a ₁ + 100		125	
write(A,a ₁); l(B); u(A)			
	l(A); read(A,a ₂)		
	a ₂ := a ₂ * 2	250	
	write(A,a ₂);		
	l(B); abgelehnt!		
read(B,b ₁)			
b ₁ := b ₁ + 100			125
write(B,b ₁); u(B)			
	l(B); u(A); read(B,b ₂)		
	b ₂ := b ₂ * 2		250
	write(B,b ₂); u(B)		

Legal?

Serialisierbar?

Konfliktserialisierbar?

Zufall?

2-Phasen Sperrprotokoll

50

2-Phase-Locking (2PL): Einfache Bedingung an Transaktionen garantiert Konfliktserialisierbarkeit.

- Alle Sperranforderungen geschehen vor allen Sperrfreigaben
- Die Phasen
 - Phase 1: Sperrphase
 - Phase 2: Freigabephase
- Wichtig: Bedingung an Transaktionen, nicht an Schedule

2-Phasen Sperrprotokoll – Beispiel

51

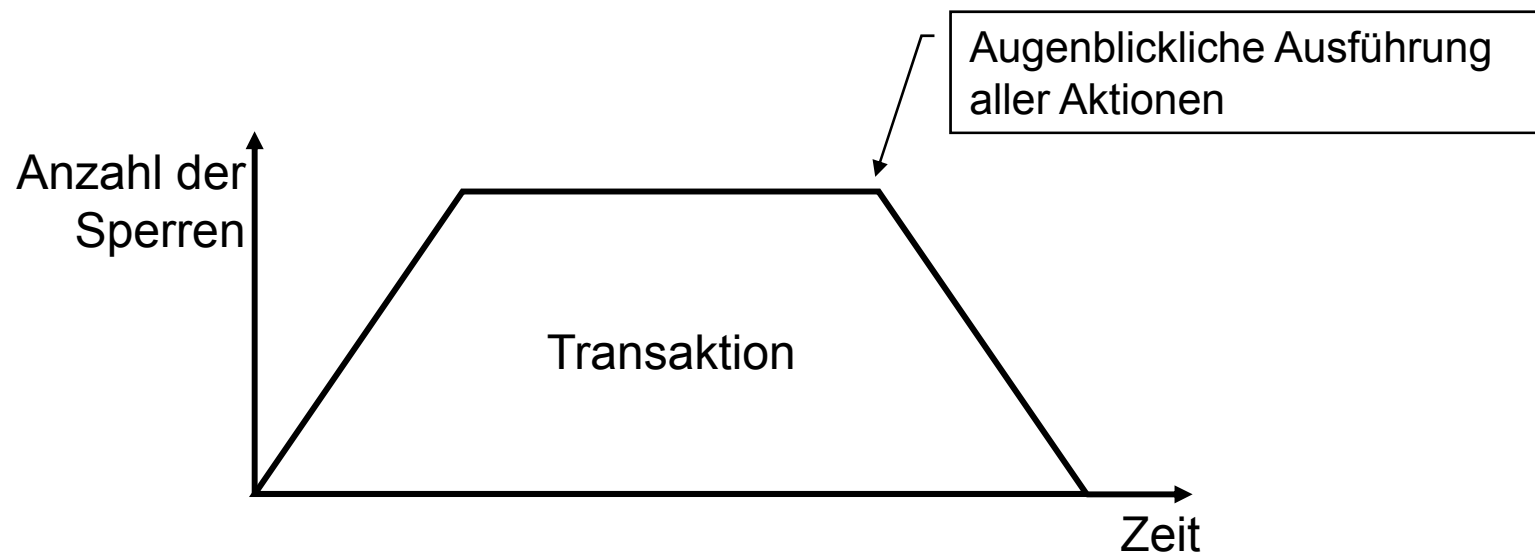
T ₁	T ₃	A	B
		25	25
l(A); read(A,a ₁)			
a ₁ := a ₁ + 100		125	
write(A,a ₁); u(A)			
	l(A); read(A,a ₂)		
	a ₂ := a ₂ * 2	250	
	write(A,a ₂); u(A)		
	l(B); read(B,b ₂)		
	b ₂ := b ₂ * 2		50
	write(B,b ₂); u(B)		
l(B); read(B,b ₁)			
b ₁ := b ₁ + 100			150
write(B,b ₁); u(B)			

T ₁	T ₃	A	B
		25	25
l(A); read(A,a ₁)			
a ₁ := a ₁ + 100		125	
write(A,a ₁); l(B); u(A)			
	l(A); read(A,a ₂)		
	a ₂ := a ₂ * 2	250	
	write(A,a ₂);		
	l(B); abgelehnt!		
read(B,b ₁)			
b ₁ := b ₁ + 100			125
write(B,b ₁); u(B)			
	l(B); u(A); read(B,b ₂)		
	b ₂ := b ₂ * 2		250
	write(B,b ₂); u(B)		

2PL – Intuition

52

- Jede Transaktion führt sämtliche Aktionen in dem Augenblick aus, zu dem das erste Objekt freigegeben wird.
- Reihenfolge der Transaktionen des äquivalenten seriellen Schedules: Reihenfolge der ersten Freigaben von Sperren



2PL – Beweis

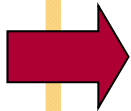
53

- Idee: Verfahren zur Konvertierung eines beliebigen, legalen Schedule S aus konsistenten, 2PL Transaktionen in einen konfliktäquivalenten seriellen Schedule
- Induktion über Anzahl der Transaktionen n
- $n = 1$: Schedule S ist bereits seriell
- $n = n + 1$:
 - S enthalte Transaktionen T_1, T_2, \dots, T_n .
 - Sei T_i die Transaktion mit der ersten Freigabe $u_i(X)$.
 - Behauptung: Es ist möglich, alle Aktionen der Transaktion an den Anfang des Schedules zu bewegen, ohne konfligierende Aktionen zu passieren.
 - Angenommen es gibt eine Schreibaktion $w_i(Y)$, die man nicht verschieben kann:
 - ◇ ... $w_k(Y)$... $u_k(Y)$... $l_i(Y)$... $w_i(Y)$...
 - Da T_i erste freigebenden Transaktion ist, gibt es ein $u_i(X)$ vor $u_k(Y)$:
 - ◇ ... $w_k(Y)$... $u_i(X)$... $u_k(Y)$... $l_i(Y)$... $w_i(Y)$...
 - T_i ist nicht 2PL

Überblick

54

- Transaktionen
- Isolationsebenen
- Serialisierbarkeit
- Konfliktserialisierbarkeit
- Sperrprotokolle
- Sperren



Mehrere Sperrmodi

55

Idee: Mehrere Arten von Sperren erhöhen die Flexibilität und verringern die Menge der abgewiesenen Sperren.

- Sperren obwohl nur gelesen wird, ist übertrieben
 - Gewisse Sperre ist dennoch nötig
 - Aber: Mehrere Transaktionen sollen gleichzeitig lesen können.
- Schreibsperre
 - *Exclusive lock*: $xl_i(X)$
 - Erlaubt das Lesen und Schreiben durch Transaktion T_i
 - Sperrt Zugriff aller anderen Transaktionen
- Lesesperre
 - *Shared lock*: $sl_i(X)$
 - Erlaubt das Lesen für Transaktion T_i
 - Sperrt Schreibzugriff für alle anderen Transaktionen
- Kompatibilität
 - Für ein Objekt darf es nur eine Schreibsperre oder mehrere Lesesperren geben.
- Freigabe
 - *Unlock*: $u_i(X)$ gibt alle Arten von Sperren frei

Bedingungen

56

- Konsistenz von Transaktionen
 - Schreiben ohne Schreibsperre ist nicht erlaubt.
 - Lesen ohne irgendeine Sperre ist nicht erlaubt.
 - Jede Sperre muss irgendwann freigegeben werden.
- 2PL von Transaktionen
 - Wie zuvor: Nach der ersten Freigabe dürfen keine Sperren mehr angefordert werden.
- Legalität von Schedules
 - Auf ein Objekt mit einer Schreibsperre darf es keine andere Sperre einer anderen Transaktion geben.
 - Auf ein Objekt kann es mehrere Lesesperren geben.

		Angeforderte Sperre	
		sl	xl
Aktuelle Sperre	sl	Ja	Nein
	xl	Nein	Nein

Beispiel

57

- T1: $sl_1(A)r_1(A)xl_1(B)r_1(B)w_1(B)u_1(A)u_1(B)$
- T2: $sl_2(A)r_2(A)sl_2(B)r_2(B)u_2(A)u_2(B)$
 - Konsistent?
 - 2PL?

T ₁	T ₂
sl(A); r(A)	
	sl(A)r(A)
	sl(B)r(B)
xl(B) – abgelehnt!	
	u(A)u(B)
xl(B)r(B)w(B)	
u(A)u(B)	

Legal?

Konfliktserialisierbar?

2PL funktioniert auch hier!

Weitere Sperrarten

58

- Upgraden einer Sperre
 - Von Lesesperre zu Schreibsperre
 - Anstatt gleich strenge Schreibsperre zu setzen
- Updatesperren
 - Erlaubt nur lesenden Zugriff
 - Kann aber Upgrade erfahren
 - Lesesperre kann dann keinen Upgrade erfahren
- Inkrementsperre
 - Erlaubt lesenden Zugriff
 - Erlaubt schreibenden Zugriff falls Wert nur inkrementiert wird.
 - Inkremente sind kommutativ.