Teil IX

Transaktionen, Integrität und Trigger

Transaktionen, Integrität und Trigger

Integrität

- Integritätsbedingung (engl. *integrity constraint* oder *assertion*): Bedingung für die „Zulässigkeit“ oder „Korrektheit“
- in Bezug auf Datenbanken:
  - (einzelne) Datenbankzustände,
  - Zustandsübergänge vom alten in den neuen Datenbankzustand,
  - langfristige Datenbankentwicklungen

Klassifikation von Integrität

Bedingungsklasse		**zeitlicher Kontext**
statisch		Datenbankzustand
dynamisch	transitional	Zustandsübergang
	temporal	Zustandsfolge

Inhärente Integritätsbedingungen im RM

1. *Typintegrität*:
   - SQL erlaubt Angabe von Wertebereichen zu Attributen
   - Erlauben oder Verbieten von Nullwerten
2. *Schlüsselintegrität*:
   - Angabe eines Schlüssels für eine Relation
3. *Referentielle Integrität*:
   - die Angabe von Fremdschlüsseln

Beispielszenarien

- Platzreservierung für Flüge gleichzeitig aus vielen Reisebüros
  $\rightarrow$ Platz könnte mehrfach verkauft werden, wenn mehrere Reisebüros den Platz als verfügbar identifizieren
- überschneidende Kontooperationen einer Bank
- statistische Datenbankoperationen
  $\rightarrow$ Ergebnisse sind verfälscht, wenn während der Berechnung Daten geändert werden

Transaktion

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

ACID-Eigenschaften

- **A**tomicity (Atomarität):
  Transaktion wird entweder ganz oder gar nicht ausgeführt
- **C**onsistency (Konsistenz oder auch Integritätserhaltung):
  Datenbank ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand
- **I**solation (Isolation):
  Nutzer, der mit einer Datenbank arbeitet, sollte den Eindruck haben, dass er mit dieser Datenbank alleine arbeitet
- **D**urability (Dauerhaftigkeit / Persistenz):
  nach erfolgreichem Abschluss einer Transaktion muss das Ergebnis dieser Transaktion „dauerhaft“ in der Datenbank gespeichert werden

Kommandos einer Transaktionssprache

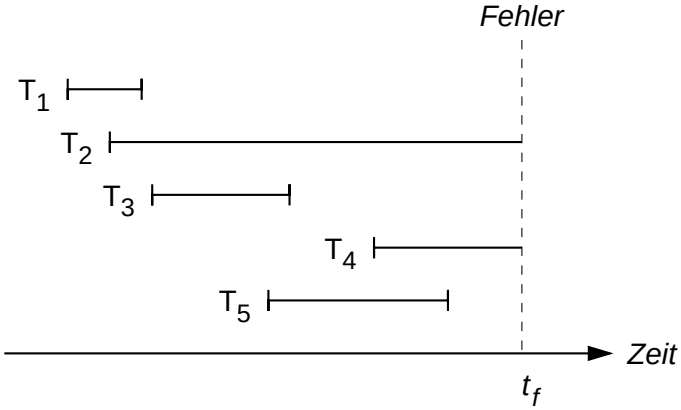
- Beginn einer Transaktion: Begin-of-Transaction-Kommando **`BOT`** (in SQL implizit!)
- **`commit`**: die Transaktion soll erfolgreich beendet werden
- **`abort`**: die Transaktion soll abgebrochen werden

Transaktion: Integritätsverletzung

- Beispiel:
  - Übertragung eines Betrages `B` von einem Haushaltsposten `K1` auf einen anderen Posten `K2`
  - Bedingung: Summe der Kontostände der Haushaltsposten bleibt konstant
- vereinfachte Notation

  Transfer = < K1:=K1−B; K2:=K2+B >;
- Realisierung in SQL: als Sequenz zweier elementarer Änderungen $\rightsquigarrow$ Bedingung ist zwischen den einzelnen Änderungsschritten nicht unbedingt erfüllt!

Transaktion: Verhalten bei Systemabsturz



Transaktion: Verhalten bei Systemabsturz /2

- **Folgen:**
  - Inhalt des flüchtigen Speichers zum Zeitpunkt t_f ist unbrauchbar $\rightarrow$ Transaktionen in unterschiedlicher Weise davon betroffen
- **Transaktionszustände:**
  - zum Fehlerzeitpunkt noch aktive Transaktionen (T_2 und T_4)
  - bereits vor dem Fehlerzeitpunkt beendete Transaktionen (T_1, T_3 und T_5)

Vereinfachtes Modell für Transaktion

- Repräsentation von Datenbankänderungen einer Transaktion
  - **read**(A,x): weise den Wert des DB-Objektes A der Variablen x zu
  - **write**(x, A): speichere den Wert der Variablen x im DB-Objekt A
- Beispiel einer Transaktion T:

$$\begin{array}{l} \textbf{read}(A,\ x);\ x := x - 200;\ \textbf{write}(x,\ A); \\ \textbf{read}(B,\ y);\ y := y + 100;\ \textbf{write}(y,\ B); \end{array}$$

- Ausführungsvarianten für zwei Transaktionen T_1, T_2:
  - seriell, etwa T_1 vor T_2
  - „gemischt“, etwa abwechselnd Schritte von T_1 und T_2

Probleme im Mehrbenutzerbetrieb

- Inkonsistentes Lesen: Nonrepeatable Read
- Abhängigkeiten von nicht freigegebenen Daten: Dirty Read
- Das Phantom-Problem
- Verlorengegangenes Ändern: Lost Update

Nonrepeatable Read

Beispiel:

- Zusicherung $x = A + B + C$ am Ende der Transaktion T_1
- x, y, z seien lokale Variablen
- T_i ist die Transaktion i
- Integritätsbedingung $A + B + C = 0$

Beispiel für inkonsistentes Lesen

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;	

Dirty Read

T_1	T_2
read(A, x);	
$x := x + 100$;	
write(x, A);	
	read(A, x);
	read(B, y);
	$y := y + x$;
	write(y, B);
	commit;
abort;	

Das Phantom-Problem

T_1	T_2
select count (*) **into** X **from** Kunde;	
	insert **into** Kunde **values** ('Meier', 0, ...); **commit**;
update Kunde **set** Bonus = Bonus $+10000/X$; **commit**;	

Lost Update

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

Serialisierbarkeit

- Einführung in die Thematik
- Formalisierung von Abläufen (Schedules)
- Serialisierbarkeitsbegriffe
- Vergleich der Serialisierbarkeitsbegriffe

Einführung in die Serialisierbarkeit

$$
\begin{aligned}
T_1 \quad : \quad & \mathbf{read}(A);\ A := A - 10;\ \mathbf{write}(A);\ \mathbf{read}(B); \\
& B := B + 10;\ \mathbf{write}(B); \\
T_2 \quad : \quad & \mathbf{read}(B);\ B := B - 20;\ \mathbf{write}(B);\ \mathbf{read}(C); \\
& C := C + 20;\ \mathbf{write}(C);
\end{aligned}
$$

- Ausführungsvarianten für zwei Transaktionen:
  - seriell, etwa T_1 vor T_2
  - „gemischt“, etwa abwechselnd Schritte von T_1 und T_2

Beispiele für verschränkte Ausführungen

Ausführung 1		**Ausführung 2**		**Ausführung 3**	
T_1	T_2	T_1	T_2	T_1	T_2
read(A)		**read**(A)		**read**(A)	
$A - 10$			**read**(B)	$A - 10$	
write(A)		$A - 10$			**read**(B)
read(B)			$B - 20$	**write**(A)	
$B + 10$		**write**(A)			$B - 20$
write(B)			**write**(B)	**read**(B)	
	read(B)	**read**(B)			**write**(B)
	$B - 20$		**read**(C)	$B + 10$	
	write(B)	$B + 10$			**read**(C)
	read(C)		$C + 20$	**write**(B)	
	$C + 20$	**write**(B)			$C + 20$
	write(C)		**write**(C)		**write**(C)

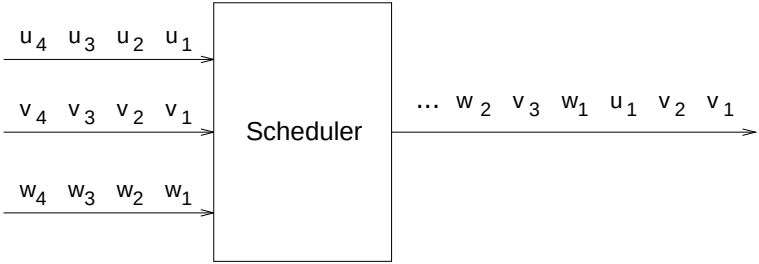
Effekt unterschiedlicher Ausführungen

	A	B	C	$A+B+C$
initialer Wert	10	10	10	30
nach Ausführung 1	0	0	30	30
nach Ausführung 2	0	0	30	30
nach Ausführung 3	0	20	30	50

Serialisierbarkeit

Eine verschränkte Ausführung mehrerer Transaktionen heißt **serialisierbar**, wenn ihr Effekt identisch zum Effekt einer (beliebig gewählten) seriellen Ausführung dieser Transaktionen ist.

Der Begriff des Schedules



Das Read/Write-Modell

- *Transaktion* T ist eine endliche Folge von Operationen (Schritten) p_i der Form $r(x_i)$ oder $w(x_i)$:

$$T = p_1 p_2 p_3 \cdots p_n \text{ mit } p_i \in \{r(x_i), w(x_i)\}$$

- Vollständige Transaktion T hat als letzten Schritt entweder einen Abbruch a oder ein Commit c:

$$T = p_1 \cdots p_n a$$

oder

$$T = p_1 \cdots p_n c.$$

Verschränkte Ausführungen

SHUFFLE($\boldsymbol{T}$): **Menge aller verschränkten Ausführungen** der Einzelschritte aller in der Menge $\boldsymbol{T}$ enthaltenen Transaktionen T_i

- alle Schritte der Transaktionen T_i sind genau einmal in jedem Element enthalten
- relative Reihenfolge der Einzelschritte einer Transaktion wird beibehalten

$$
\begin{aligned}
T_1 &:= r_1(x)w_1(x) \\
T_2 &:= r_2(x)r_2(y)w_2(y) \\
\text{SHUFFLE}(\boldsymbol{T}) = \quad & \{[r_1(x)w_1(x)r_2(x)r_2(y)w_2(y)], [r_2(x)r_1(x) \\
& w_1(x)r_2(y)w_2(y)], \ \ldots \}
\end{aligned}
$$

Schedule

Ein **Schedule** ist ein Präfix eines vollständigen Schedules.

$$\underbrace{\underbrace{r_1(x)r_2(x)w_1(x)}_{\text{ein Schedule}}\, r_2(y)a_1w_2(y)c_2}_{\text{ein vollständiger Schedule}}$$

Serieller Schedule

- Ein **serieller Schedule** s für $\boldsymbol{T}$ ist ein vollständiger Schedule der folgenden Form:

$$s := T_{\rho(1)} \cdots T_{\rho(n)} \qquad \text{für eine Permutation } \rho \text{ von } \{1, \ldots, n\}$$

- resultierende serielle Schedules für zwei Transaktionen $T_1 := r_1(x)w_1(x)c_1$ und $T_2 := r_2(x)w_2(x)c_2$:

$$s_1 := \underbrace{r_1(x)w_1(x)c_1}_{T_1}\underbrace{r_2(x)w_2(x)c_2}_{T_2}$$
$$s_2 := \underbrace{r_2(x)w_2(x)c_2}_{T_2}\underbrace{r_1(x)w_1(x)c_1}_{T_1}$$

Korrektheitskriterium

Ein verzahnter Schedule s ist **korrekt**, wenn der Effekt des Schedules s (Ergebnis der Ausführung des Schedules) äquivalent dem Effekt eines (beliebigen) seriellen Schedules s' bzgl. derselben Menge von Transaktionen ist (in Zeichen $s \approx s'$).

Ist ein Schedule s äquivalent zu einem seriellen Schedule s', dann ist s **serialisierbar** (zu s').

Konfliktserialisierbarkeit

- Konfliktrelation C von Schedule s:

$$C(s) := \{ \ (p, q) \ \mid p, q \text{ sind in Konflikt und } p \rightarrow_s q \}$$

- Konfliktmatrix:

	$r_i(x)$	$w_i(x)$
$r_j(x)$	$\surd$	–
$w_j(x)$	–	–

Bereinigte Konfliktrelation

- Mit conf(s) wird „bereinigte“ Konfliktrelation bezeichnet, in der keine abgebrochenen Transaktionen mehr vorkommen

$$\text{conf}(s) := C(s) - \{\ (p,q) \mid (p \in t' \vee q \in t') \wedge t' \in \texttt{aborted}(s)\}$$

- **`aborted`**(s): Menge der abgebrochenen Transaktionen des Schedules s

Beispiel Konfliktrelation

- Geg.: Schedule s:

$$s = r_1(x)w_1(x)r_2(x)r_3(y)w_2(y)c_2a_1c_3$$

- Konfliktrelation zu s:

$$C(s) := \{(w_1(x), r_2(x)), (r_3(y), w_2(y))\}$$

- Entfernen der abgebrochene Transaktion T_1 aus s:

$$\text{conf}(s) := \{(r_3(y), w_2(y))\}$$

Konfliktäquivalenz

- Zwei Schedules s und s' heissen *konfliktäquivalent* ($s \approx_c s'$) falls gilt:
  1. $op(s) = op(s')$
  2. $\mathsf{conf}(s) = \mathsf{conf}(s')$

Konfliktserialisierbarkeit

Ein Schedule s ist genau dann **konfliktserialisierbar**, wenn s konfliktäquivalent zu einem seriellen Schedule ist.

- Klasse aller konfliktserialisierbaren Schedules: **CSR** (für engl. *conflict serializable*)

Beispiel Konfliktserialisierbarkeit

- Geg.: zwei Schedules s und s':

$$s = r_1(x)r_1(y)w_2(x)w_1(y)r_2(z)w_1(x)w_2(y)$$
$$s' = r_1(y)r_1(x)w_1(y)w_2(x)w_1(x)r_2(z)w_2(y)$$

- Frage:
  Sind die Schedules s und s' konfliktäquivalent?
- 1. Schritt:
  $op(s) = op(s')$ gilt, da alle in s vorkommenden Datenbankoperationen auch in s' vorkommen; gilt auch umgekehrt

Beispiel Konfliktserialisierbarkeit /2

- 2. Schritt: bereinigte Konfliktelationen

$$\begin{aligned}\text{conf}(s) &= \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)),\\ &\quad (w_1(y), w_2(y))\}\\ \text{conf}(s') &= \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)),\\ &\quad (w_1(y), w_2(y))\}\end{aligned}$$

  - Es gilt $\text{conf}(s) = \text{conf}(s')$; somit stimmen auch die Konfliktrelationen überein und damit sind s und s' konfliktäquivalent

Beispiel Konfliktserialisierbarkeit /3

- Test auf Konfliktserialisierbarkeit durch Vergleich mit den seriellen Schedules
- Geg.: Schedule s

$$s = r_1(x)r_1(y)w_2(x)w_1(y)r_2(z)w_1(x)w_2(y)$$

Beispiel Konfliktserialisierbarkeit /4

- bereinigte Konfliktrelation für s:

$$\begin{aligned}\text{conf}(s) \quad = \quad & \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), \\ & w_2(y)), (w_1(y), w_2(y))\}\end{aligned}$$

- möglicher serieller Schedule s_1

$$s_1 = T_1 T_2 = r_1(x) r_1(y) w_1(y) w_1(x) c_1 w_2(x) r_2(z) w_2(y) c_2$$

  - Konfliktrelation von s_1 stimmt *nicht* mit der von s überein:

$$\begin{aligned}\text{conf}(s_1) \quad = \quad & \{(r_1(x), w_2(x)), (w_1(x), w_2(x)), \\ & (r_1(y), w_2(y)), (w_1(y), w_2(y))\}\end{aligned}$$

Beispiel Konfliktserialisierbarkeit /5

- möglicher serieller Schedule s_2 s Kandidat:

$$s_2 = T_2T_1 = w_2(x)r_2(z)w_2(y)c_2r_1(x)r_1(y)w_1(y)w_1(x)c_1$$

  - auch Konfliktrelation von s_2 stimmt *nicht* mit der von s überein:

$$\begin{aligned} \text{conf}(s_2) \quad = \quad & \{(w_2(x), r_1(x)), (w_2(y), r_1(y)), \\ & (w_2(y), w_1(y)), (w_2(x), w_1(x))\} \end{aligned}$$

- somit gilt: $s \notin \textbf{CSR}$, d.h., der Schedule s ist nicht konfliktserialisierbar

Beispiel Konfliktserialisierbarkeit /6

- Schedule

$$s_3 = r_1(x) r_2(x) w_2(y) c_2 w_1(x) c_1$$

ist trivialerweise konfliktserialisierbar, da nur ein einziger Konflikt auftritt

Graphbasierter Test

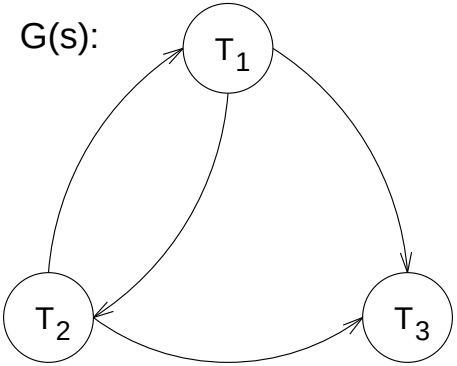
- Konfliktgraph $G(s) = (V, E)$ von Schedule s:
  1. Knotenmenge V enthält alle in s vorkommende Transaktionen
  2. Kantenmenge E enthält alle gerichteten Kanten zwischen zwei in Konflikt stehenden Transaktionen, also:
     $(t, t') \in E \Leftrightarrow t \neq t' \wedge (\exists p \in t)(\exists q \in t')$ mit $(p, q) \in \text{conf}(s)$

Zeitlicher Verlauf dreier Transaktionen

T_1	T_2	T_3
r(y)		
		r(u)
	r(y)	
w(y)		
w(x)		
	w(x)	
	w(z)	
		w(x)

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

Konfliktgraph



Eigenschaften von Konfliktgraph $G(s)$

1. Ist s ein serieller Schedule, dann ist der vorliegende Konfliktgraph ein azyklischer Graph.
2. Für jeden azyklischen Graphen $G(s)$ lässt sich ein serieller Schedule s' konstruieren, sodass s konfliktserialisierbar zu s' ist (Test bspw. durch *topologisches Sortieren*)
3. Enthält ein Graph Zyklen, dann ist der zugehörige Schedule nicht konfliktserialisierbar.

Konfliktgraphen und -serialisierbarkeit

Für jeden Schedule s gilt:

$$G(s) \text{ azyklisch} \Leftrightarrow s \in \textbf{CSR}$$

Probleme zur Laufzeit

- zur Laufzeit nur unvollständige Schedules verfügbar $\rightsquigarrow$ Überwachung unvollständiger Schedules notwendig
- Transaktionen, die noch kein **`commit`** gemacht haben, können noch jederzeit abgebrochen werden

Konservative Scheduler

- ein Scheduler arbeitet **konservativ**, wenn er Konflikte möglichst vermeidet, dafür aber Verzögerungen von Transaktionen in Kauf nimmt
- erlauben nur eine geringe Parallelität von Transaktionen
- minimieren den Rücksetzungsaufwand für abgebrochene Transaktionen
- im Extremfall findet keine Parallelisierung von Transaktionen mehr statt, d.h., es werden immer alle Transaktionen bis auf eine verzögert

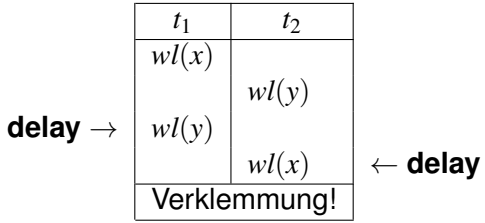
Sperrmodelle

- Schreib- und Lesesperren in folgender Notation:
  - $rl(x)$: Lesesperre (engl. *read lock* bzw. *shared lock*) auf einem Objekt x
  - $wl(x)$: Schreibsperre (engl. *write lock* bzw. *exclusive lock*) auf einem Objekt x
- *Entsperren* $ru(x)$ und $wu(x)$, oft zusammengefasst $u(x)$ für engl. *unlock*

Sperrdisziplin

- Schreibzugriff $w(x)$ nur nach Setzen einer Schreibsperre $wl(x)$ möglich
- Lesezugriffe $r(x)$ nur nach $rl(x)$ oder $wl(x)$ erlaubt
- nur Objekte sperren, die nicht bereits von einer anderen Transaktion exklusiv gesperrt
- nach $rl(x)$ nur noch $wl(x)$ erlaubt, danach auf x keine Sperre mehr; Sperren derselben Art werden maximal einmal gesetzt
- nach $u(x)$ durch t_i darf t_i kein erneutes $rl(x)$ oder $wl(x)$ ausführen
- vor einem **`commit`** müssen alle Sperren aufgehoben werden

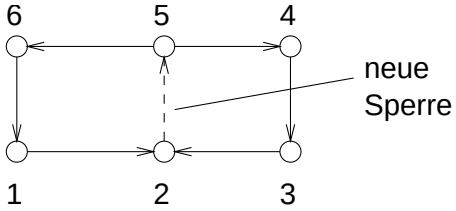
Verklemmungen



- Alternativen:
  - Verklemmungen werden erkannt und beseitigt
  - Verklemmungen werden von vornherein vermieden

Erkennung und Auflösung

- Wartegraph



- Auflösen durch Abbruch einer Transaktion, Kriterien:
  - Anzahl der aufgebrochenen Zyklen,
  - Länge einer Transaktion,
  - Rücksetzaufwand einer Transaktion,
  - Wichtigkeit einer Transaktion, ...

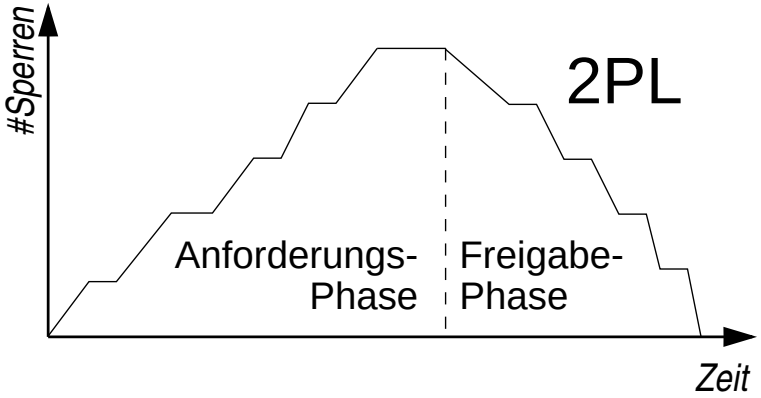
Livelock-Problem

1. T_1 sperrt A
2. T_2 will A sperren, muss aber warten
3. T_3 will danach A sperren, muss auch warten
4. T_1 gibt A frei
5. T_3 kommt vor T_2 an eine Zeitscheibe, sperrt A
6. T_2 will weiterhin A sperren, muss aber warten
7. T_4 will danach A sperren, muss auch warten
8. T_3 gibt A frei
9. T_4 kommt vor T_2 an die nächste Zeitscheibe ...

Sperrprotokolle: Notwendigkeit

T_1	T_2
$wl(x)$	
$w(x)$	
$u(x)$	
	$wl(x)$
	$w(x)$
	$u(x)$
	$wl(y)$
	$w(y)$
	$u(y)$
$wl(y)$	
$w(y)$	
$u(y)$	

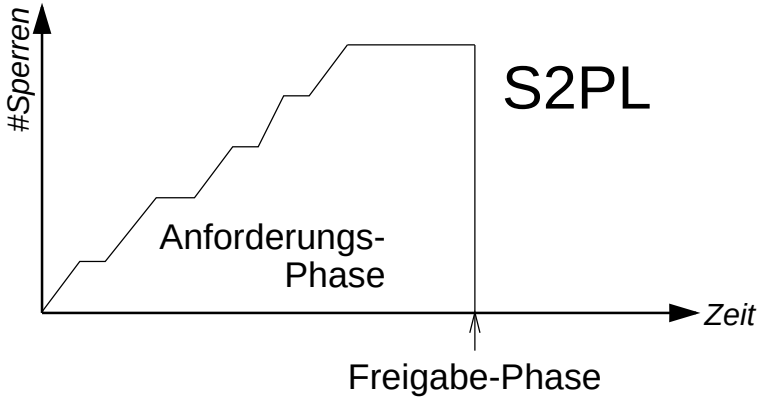
Zwei-Phasen-Sperr-Protokoll



Konflikt bei Nichteinhaltung des 2PL

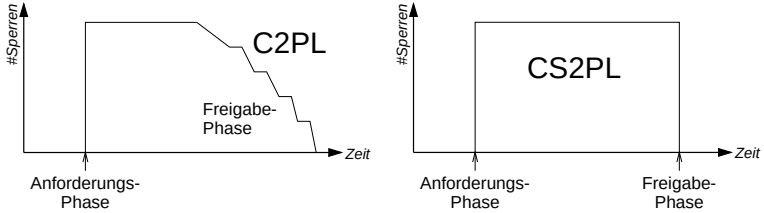
T_1	T_2
$u(x)$	
	$wl(x)$
	$wl(y)$
	$\vdots$
	$u(x)$
	$u(y)$
$wl(y)$	
$\vdots$	

Striktes Zwei-Phasen-Sperr-Protokoll



vermeidet kaskadierende Abbrüche!

Konservatives 2PL-Protokoll



vermeidet Deadlocks!

Transaktionen in SQL-DBS

Aufweichung von ACID in SQL: Isolationsebenen

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

Standardeinstellung:

```
set transaction read write,
   isolation level serializable
```

Bedeutung der Isolationsebenen

- **`read uncommitted`**
  - schwächste Stufe: Zugriff auf nicht geschriebene Daten, nur für **`read only`** Transaktionen
  - statistische und ähnliche Transaktionen (ungefährer Überblick, nicht korrekte Werte)
  - keine Sperren $\rightarrow$ 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

Isolationsebenen: `read committed`

	T_1	T_2
	set transaction **isolation level** **read committed**	
1	**select** Name **from** WEINE **where** WeinID = 1014 $\longrightarrow$ *Riesling*	
2		**update** WEINE **set** Name = 'Riesling Superiore' **where** WeinID = 1014
3	**select** Name **from** WEINE **where** WeinID = 1014 $\longrightarrow$ *Riesling*	
4		**commit**
5	**select** Name **from** WEINE **where** WeinID = 1014 $\longrightarrow$ *Riesling Superiore*	

read committed /2

	T_1	T_2
	set transaction **isolation level** **read committed**	
1	**select** Name **from** WEINE **where** WeinID = 1014	
2		**update** WEINE **set** Name = 'Riesling Superiore' **where** WeinID = 1014
3	**update** WEINE **set** Name = 'Superiore Riesling' **where** WeinID = 1014 $\longrightarrow$ *blockiert*	
4		**commit**
5	**commit**	

Isolationsebenen: **serializable**

	T_1	T_2
	set transaction isolation level serializable	
1	**select** Name **into** N **from** WEINE **where** WeinID = 1014 ⟶ N := *Riesling*	
2		**update** WEINE **set** Name = 'Riesling Superiore' **where** WeinID = 1014
4		**commit**
5	**update** WEINE **set** Name = 'Superior' || N **where** WeinID = 1014 ⟶ *Abbruch*	

Integritätsbedingungen in SQL-DDL

- **`not null`**: Nullwerte verboten
- **`default`**: Angabe von Default-Werten
- **`check`** (*search-condition*): Attributspezifische Bedingung
  (in der Regel *Ein-Tupel-Integritätsbedingung*)
- **`primary key`**: Angabe eines Primärschlüssel
- **`foreign key`** (*Attribut(e)*)
  `references` *Tabelle*(*Attribut(e)*):
  Angabe der referentiellen Integrität

Integritätsbedingungen: Wertebereiche

- **`create domain`**: Festlegung eines benutzerdefinierten Wertebereichs
- Beispiel

```
create domain WeinFarbe varchar(4)
   default 'Rot'
   check (value in ('Rot', 'Weiß', 'Rose'))
```

- Anwendung

```
create table WEINE (
   WeinID int primary key,
   Name varchar(20) not null,
   Farbe WeinFarbe,
   ...)
```

Integritätsbedingungen: `check`-Klausel

- **`check`**: Festlegung weitere lokale Integritätsbedingungen innerhalb der zu definierenden Wertebereiche, Attribute und Relationenschemata
- Beispiel: Einschränkung der zulässigen Werte
- Anwendung

```
create table WEINE (
   WeinID int primary key,
   Name varchar(20) not null,
   Jahr int check(Jahr between 1980 and 2010),
   ...
)
```

Erhaltung der referentiellen Integrität

- Überprüfung der Fremdschlüsselbedingungen nach Datenbankänderungen
- für $\pi_A(r_1) \subseteq \pi_K(r_2)$,
  z.B. $\pi_{\text{Weingut}}(\texttt{WEINE}) \subseteq \pi_{\text{Weingut}}(\texttt{ERZEUGER})$
  - Tupel t wird eingefügt in $r_1 \Rightarrow$ überprüfen, ob $t' \in r_2$ existiert mit: $t'(K) = t(A)$, d.h. $t(A) \in \pi_K(r_2)$
    falls nicht $\Rightarrow$ abweisen
  - Tupel t' wird aus r_2 gelöscht $\Rightarrow$ überprüfen, ob $\sigma_{A=t'(K)}(r_1) = \{\}$, d.h. kein Tupel aus r_1 referenziert t'
    falls nicht leer $\Rightarrow$ abweisen oder Tupel aus r_1, die t' referenzieren, löschen (bei kaskadierendem Löschen)

Überprüfungsmodi von Bedingungen

- **`on update | delete`**
  Angabe eines Auslöseereignisses, das die Überprüfung der Bedingung anstößt
- **`cascade`** | **`set null`** | **`set default`** | **`no action`**
  Kaskadierung: Behandlung einiger Integritätsverletzungen pflanzt sich über mehrere Stufen fort, z.B. Löschen als Reaktion auf Verletzung der referentieller Integrität
- **`deferred`** | **`immediate`** legt Überprüfungszeitpunkt für eine Bedingung fest
  - **`deferred`**: Zurückstellen an das Ende der Transaktion
  - **`immediate`**: sofortige Prüfung bei jeder relevanten Datenbankänderung

Überprüfungsmodi: Beispiel

- Kaskadierendes Löschen

```
create table WEINE (
   WeinID int primary key,
   Name varchar(50) not null,
   Preis float not null,
   Jahr int not null,
   Weingut varchar(30),
   foreign key (Weingut) references ERZEUGER (Weingut)
      on delete cascade)
```

Die `assertion`-Klausel

- Assertion: Prädikat, das eine Bedingung ausdrückt, die von der Datenbank immer erfüllt sein muss
- Syntax (SQL:2003)

```
create assertion name check ( prädikat )
```

- Beispiele:

```
create assertion Preise check
   ( ( select sum (Preis)
       from WEINE) < 10000 )
```

```
create assertion Preise2 check
   ( not exists (
       select * from WEINE where Preis > 200) )
```

Trigger

- Trigger: Anweisung/Prozedur, die bei Eintreten eines bestimmten Ereignisses automatisch vom DBMS ausgeführt wird
- Anwendung:
  - Erzwingen von Integritätsbedingungen („Implementierung“ von Integritätsregeln)
  - Auditing von DB-Aktionen
  - Propagierung von DB-Änderungen
- Definition:

```
create trigger ...
after <Operation>
<Anweisungen>
```

Beispiel für Trigger

- Realisierung eines berechneten Attributs durch zwei Trigger:
  - Einfügen von neuen Aufträgen

```
create trigger Auftragszählung+
   on insertion of Auftrag A:
   update Kunde
   set AnzAufträge = AnzAufträge + 1
   where KName = new A.KName
```

  - analog für Löschen von Aufträgen:

```
create trigger Auftragszählung-
   on deletion ...:
   update ...- 1 ...
```

Trigger: Entwurf und Implementierung

- Spezifikation von
  - *Ereignis* und *Bedingung* für Aktivierung des Triggers
  - *Aktion(en)* zur Ausführung
- Syntax in SQL:2003 festgelegt
- verfügbar in den meisten kommerziellen Systemen (aber mit anderer Syntax)

SQL:2003-Trigger

- Syntax:

```
create trigger <Name: >
after | before <Ereignis>
on <Relation>
[ when <Bedingung> ]
begin atomic < SQL-Anweisungen > end
```

- Ereignis:
  - **insert**
  - **update** [**of** <Liste von Attributen>]
  - **delete**

Weitere Angaben bei Triggern

- **`for each row`** bzw. **`for each statement`**: Aktivierung des Triggers für *jede* Einzeländerungen einer mengenwertigen Änderung oder nur einmal für die gesamte Änderung
- **`before`** bzw. **`after`**: Aktivierung *vor* oder *nach* der Änderung
- **`referencing new as`** bzw. **`referencing old as`**: Binden einer Tupelvariable an die neu eingefügten bzw. gerade gelöschten („alten'“) Tupel einer Relation
  $\rightsquigarrow$ Tupel der *Differenzrelationen*

Beispiel für Trigger

- *Kein Kundenkonto darf unter 0 absinken:*

```
create trigger bad_account
after update of Kto on KUNDE
referencing new as INSERTED
when (exists
   (select * from INSERTED where Kto < 0)
)
begin atomic
   rollback;
end
```

$\rightsquigarrow$ ähnlicher Trigger für **insert**

Beispiel für Trigger /2

- *Erzeuger **müssen** gelöscht werden, wenn sie keine Weine mehr anbieten:*

```
create trigger unnützes_Weingut
after delete on WEINE
referencing old as O
for each row
when (not exists
   (select * from WEINE W
    where W.Weingut = O.Weingut))
begin atomic
   delete from ERZEUGER where Weingut = O.Weingut;
end
```

Integritätssicherung durch Trigger

1. Bestimme Objekt o_i, für das die Bedingung ϕ überwacht werden soll
   - i.d.R. mehrere o_i betrachten, wenn Bedingung relationsübergreifend ist
   - Kandidaten für o_i sind Tupel der Relationsnamen, die in ϕ auftauchen
2. Bestimme die elementaren Datenbankänderungen u_{ij} auf Objekten o_i, die ϕ verletzen können
   - Regeln: z.B. Existenzforderungen beim Löschen und Ändern prüfen, jedoch nicht beim Einfügen etc.

Integritätssicherung durch Trigger /2

3. Bestimme je nach Anwendung die Reaktion r_i auf Integritätsverletzung
   - Rücksetzen der Transaktion (**`rollback`**)
   - korrigierende Datenbankänderungen
4. Formuliere folgende Trigger:

`create trigger` `t-phi-ij` **`after`** u_{ij} **`on`** o_i
`when` $\neg\phi$
`begin` r_i **`end`**

5. Wenn möglich, vereinfache entstandenen Trigger

Trigger in Oracle

- Implementierung in PL/SQL
- Notation

```
create [ or replace ] trigger trigger-name
   before | after
   insert or update [ of spalten ]
      or delete on tabelle
   [ for each row
   [ when ( prädikat ) ] ]
   PL/SQL-Block
```

Trigger in Oracle: Arten

- Anweisungsebene (*statement level trigger*): Trigger wird ausgelöst vor bzw. nach der DML-Anweisung
- Tupelebene (*row level trigger*): Trigger wird vor bzw. nach jeder einzelnen Modifikation ausgelöst (*one tuple at a time*)

Trigger auf Tupelebene:

- Prädikat zur Einschränkung (**`when`**)
- Zugriff auf altes (**`:old.`***col*) bzw. neues (**`:new.`***col*) Tupel
  - für **`delete`**: nur (**`:old.`***col*)
  - für **`insert`**: nur (**`:new.`***col*)
  - in **`when`**-Klausel nur (**`new.`***col*) bzw. (**`old.`***col*)

Trigger in Oracle /2

- Transaktionsabbruch durch **`raise_application_error`**(*`code, message`*)
- Unterscheidung der Art der DML-Anweisung

```
if deleting then ... end if;
if updating then ... end if;
if inserting then ... end if;
```

Trigger in Oracle: Beispiel

- *Kein Kundenkonto darf unter 0 absinken:*

```
create or replace trigger bad_account
after insert or update of Kto on KUNDE
for each row
when (:new.Kto < 0)
begin
   raise_application_error(-20221,
      'Nicht unter 0');
end;
```

Zusammenfassung

- Zusicherung von Korrektheit bzw. Integrität der Daten
- inhärente Integritätsbedingungen des Relationenmodells
- zusätzliche SQL-Integritätsbedingungen: **`check`**-Klausel, **`assertion`**-Anweisung
- Trigger zur „Implementierung“ von Integritätsbedingungen bzw. -regeln