Datenbanksysteme

VU 184.686, WS 2022

Mehrbenutzerinsynchronisation

Johannes Fichte, Felix Winter

Institut für Logic and Computation
Technische Universität Wien

Acknowledgments

Die Folien sind eine Weiterentwicklung der Folien von Reinhard Pichler und Sebastian Skritek.

Der Inhalt basiert auf und behandelt Kapitel 11 des Lehrbuchs
(Kemper, Eickler: Datenbanksysteme – Eine Einführung).
Die meisten Beispiele und Abbildungen sind von dort entnommen.

Mehrbenutzerinsynchronisation (Concurrency Control)

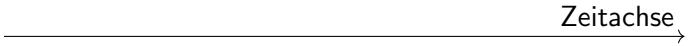
1. Nebenläufigkeit und mögliche Fehler
2. Klassifikation von Historien
3. Concurrency Control
4. Transaktionsverwaltung in SQL

Inhalt

Inhalt

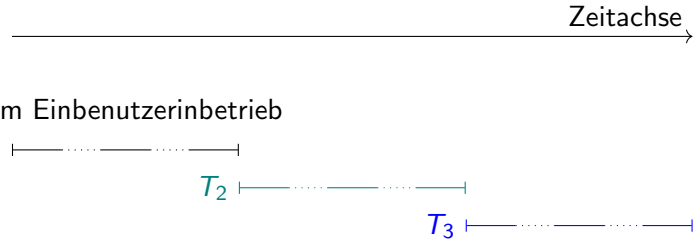
Nebenläufigkeit

Ausführung der drei Transaktionen T_1, T_2 und T_3:



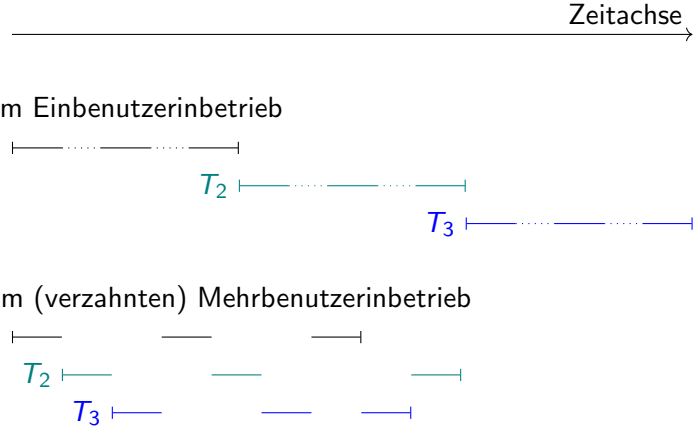
Nebenläufigkeit

Ausführung der drei Transaktionen T_1, T_2 und T_3:



Nebenläufigkeit

Ausführung der drei Transaktionen T_1, T_2 und T_3:



Verzahnte Ausführung

Idee:

- CPU- und I/O-Aktivitäten können parallel geschehen.
- Verzahnte Ausführung mehrerer Transaktionen führt zu besserer Auslastung der Ressourcen.

Verzahnte Ausführung

Idee:

- CPU- und I/O-Aktivitäten können parallel geschehen.
- Verzahnte Ausführung mehrerer Transaktionen führt zu besserer Auslastung der Ressourcen.

Vorteile:

- Durchsatz des DBMS kann erhöht werden (durchschnittliche Anzahl der abgeschlossenen Transaktionen pro Zeiteinheit).
- Unvorhersehbare Verzögerungen der Antwortzeit lassen sich dadurch reduzieren.

Inhalt

Mögliche Fehler bei unkontrollierter Nebenläufigkeit

- Lost Update
- Dirty Read
- Unrepeatable Read
- Phantomproblem

Lost Update

Beispiel

T_1: Überweisung von A nach B

T_2: Zinsgutschrift für A

Lost Update

Beispiel

T_1: Überweisung von A nach B

T_2: Zinsgutschrift für A

Schritt	T_1	T_2
1.	read(A,a_1)	
2.		read(A,a_2)
3.		write(A,$a_2 * 1.03$)
4.	write(A,$a_1 - 300$)	
5.	read(B,b_1)	
6.	write(B,$b_1 + 300$)	

Lost Update

Beispiel

T_1: Überweisung von A nach B

T_2: Zinsgutschrift für A

Schritt	T_1	T_2
1.	read(A,a_1)	
2.		read(A,a_2)
3.		write(A,$a_2 * 1.03$)
4.	write(A,$a_1 - 300$)	
5.	read(B,b_1)	
6.	write(B,$b_1 + 300$)	

Problem: T_1 überschreibt die Änderung von T_2

$\Rightarrow$ die Änderungen von T_2 gehen verloren.

Dirty Read

Lesen nicht freigegebener (committed) Änderungen

Beispiel

Schritt	T_1	T_2
1.	read(A,a_1)	
2.	write(A,$a_1 - 300$)	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.	read(B,b_1)	
6.	...	
7.	**abort**	

Dirty Read

Lesen nicht freigegebener (committed) Änderungen

Beispiel

Schritt	T_1	T_2
1.	read(A,a_1)	
2.	write(A,$a_1 - 300$)	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.	read(B,b_1)	
6.	...	
7.	**abort**	

Problem:

- Änderungen in T_2 basieren auf inkonsistentem DB-Zustand.
- Selbst bei `commit` statt `abort`

Unrepeatable Read

Beispiel

Schritt	T_1	T_2
1.	read(A,a_1)	
2.	...	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.		**commit**
6.	read(A,a_1)	
7.	...	

Unrepeatable Read

Beispiel

Schritt	T_1	T_2
1.	read(A,a_1)	
2.	...	
3.		read(A,a_2)
4.		write(A,$a_2 * 1.03$)
5.		**commit**
6.	read(A,a_1)	
7.	...	

Problem: Wiederholtes Lesen durch T_1 liefert unterschiedliche Ergebnisse, obwohl T_1 keine Änderung vorgenommen hat.

Phantomproblem

Beispiel

Schritt	T_1	T_2
1.		`SELECT SUM(KontoStand)` `FROM Konten`
2.	`INSERT INTO Konten` `VALUES (C,1000,...)`	
3.		`SELECT SUM(KontoStand)` `FROM Konten`

Phantomproblem

Beispiel

Schritt	T_1	T_2
1.		`SELECT SUM(KontoStand)` `FROM Konten`
2.	`INSERT INTO Konten` `VALUES (C,1000,...)`	
3.		`SELECT SUM(KontoStand)` `FROM Konten`

Problem: T_2 berechnet unterschiedliche Werte, da „Phantom“ mit den Werten $(C, 1000, \ldots)$ eingefügt wurde.

Phantomproblem

Beispiel

Schritt	T_1	T_2
1.		`SELECT SUM(KontoStand) FROM Konten`
2.	`INSERT INTO Konten VALUES (C,1000,...)`	
3.		`SELECT SUM(KontoStand) FROM Konten`

Problem: T_2 berechnet unterschiedliche Werte, da „Phantom“ mit den Werten $(C, 1000, \ldots)$ eingefügt wurde.

Bei Zugriff auf Mengen von Objekten mit bestimmter Eigenschaft

Konflikte zwischen Transaktionen

Konflikt:

- Zwei Transaktionen greifen auf das selbe Objekt zu
- Mindestens ein Zugriff ist schreibend
- Mögliche Konflikte: W-W, W-R, R-W

Konflikte zwischen Transaktionen

Konflikt:

- Zwei Transaktionen greifen auf das selbe Objekt zu
- Mindestens ein Zugriff ist schreibend
- Mögliche Konflikte: W-W, W-R, R-W

Fehler durch Konflikte verursacht:

- Lost Update: W-W
- Dirty Read: W-R
- Unrepeatable Read: R-W
- Phantomproblem: R-W

Inhalt

Inhalt

Transaktionen

Transaktion: besteht aus einer Menge an Operationen sowie einer (partiellen) Ordnung dieser Operationen.

Transaktionen

Transaktion: besteht aus einer Menge an Operationen sowie einer (partiellen) Ordnung dieser Operationen.

Elementare Operationen einer Transaktion T_i:

- $r_i(A)$: Liest das Datenobjekt A
- $w_i(A)$: Schreibt das Datenobjekt A
- a_i: Abbruch der Transaktion (`abort`)
- c_i: Festschreiben der Transaktion (`commit`)

(wir betrachten kein insert oder delete)

Transaktionen

Transaktion T_i: besteht aus einer Menge an Operationen sowie einer (partiellen) Ordnung dieser Operationen.

Transaktionen

Transaktion T_i: besteht aus einer Menge an Operationen sowie einer (partiellen) Ordnung dieser Operationen.

Mindestanforderung an die **Ordnung** $<_i$ auf den Operationen $o_i \in \{r_i, w_i\}$:

- Falls die Transaktion `abort` enthält:
  $o_i(A) <_i a_i$ für alle Operationen $o_i(A)$ (`abort` letzte Aktion)
- Falls die Transaktion `commit` enthält:
  $o_i(A) <_i c_i$ für alle Operationen $o_i(A)$ (`commit` letzte Aktion)
- Muss zwischen allen Paaren von Operationen auf dem selben Datenobjekt definiert sein.

Im Allgemeinen: totale Ordnung

Historien (Schedules)

Historie: zeitliche Anordnung der elementaren Operationen einer Menge von Transaktionen.

Historien (Schedules)

Historie: zeitliche Anordnung der elementaren Operationen einer Menge von Transaktionen.

Eigenschaften einer Historie H mit Ordnung $<_H$:

- H enthält genau die elementaren Operationen aller beteiligten Transaktionen
- $<_H$ ist verträglich mit allen $<_i$ (Reihenfolge innerhalb Transaktion nicht geändert)
- für Konfliktoperationen p, q gilt entweder $p <_H q$ oder $q <_H p$

Historien (Schedules)

Historie: zeitliche Anordnung der elementaren Operationen einer Menge von Transaktionen.

Eigenschaften einer Historie H mit Ordnung $<_H$:

- H enthält genau die elementaren Operationen aller beteiligten Transaktionen
- $<_H$ ist verträglich mit allen $<_i$ (Reihenfolge innerhalb Transaktion nicht geändert)
- für Konfliktoperationen p, q gilt entweder $p <_H q$ oder $q <_H p$

Konfliktoperationen: Paare von Operationen die auf das selbe Datenobjekt zugreifen, mindestens eine davon schreibend:

Historien (Schedules)

Historie: zeitliche Anordnung der elementaren Operationen einer Menge von Transaktionen.

Eigenschaften einer Historie H mit Ordnung $<_H$:

- H enthält genau die elementaren Operationen aller beteiligten Transaktionen
- $<_H$ ist verträglich mit allen $<_i$ (Reihenfolge innerhalb Transaktion nicht geändert)
- für Konfliktoperationen p, q gilt entweder $p <_H q$ oder $q <_H p$

Konfliktoperationen: Paare von Operationen die auf das selbe Datenobjekt zugreifen, mindestens eine davon schreibend:
$(r_i(A), w_j(A)), \quad (w_i(A), r_j(A)), \quad (w_i(A), w_j(A))$

Historie für drei Transaktionen

Beispiel

$$
H = \begin{array}{ccccccc}
 & & r_2(A) & \longrightarrow & w_2(B) & \longrightarrow & w_2(C) \longrightarrow c_2 \\
 & & \uparrow & & \uparrow & & \uparrow \\
r_3(B) & \longrightarrow & w_3(A) & \longrightarrow & w_3(B) & \longrightarrow & w_3(C) \longrightarrow c_3 \\
 & & \uparrow & & & & \\
r_1(A) & \longrightarrow & w_1(A) & \longrightarrow & c_1 & &
\end{array}
$$

Inhalt

Serialisierbarkeit

Serielle Historie: Jede Transaktion wird vollständig abgearbeitet bevor die nächste beginnt.

Schreibweise: $T_1 \mid T_2 \mid T_3 \ldots$ (für „T_1 vor T_2 vor T_3 $\ldots$“).

Serialisierbarkeit

Serielle Historie: Jede Transaktion wird vollständig abgearbeitet bevor die nächste beginnt.

Schreibweise: $T_1 \mid T_2 \mid T_3 \ldots$ (für „T_1 vor T_2 vor T_3 …“).

Serialisierbare Historie: Eine (verzahnte) Historie welche den selben Effekt hat wie irgendeine serielle Ausführung (informell)

Serialisierbarkeit

Serielle Historie: Jede Transaktion wird vollständig abgearbeitet bevor die nächste beginnt.

Schreibweise: $T_1 \mid T_2 \mid T_3 \ldots$ (für „T_1 vor T_2 vor T_3 …“).

Serialisierbare Historie: Eine (verzahnte) Historie welche den selben Effekt hat wie irgendeine serielle Ausführung (informell)

- Auf allen Ausprägungen (Instanzen)
- Aus Sicht der DBMS; DBMS sieht nur Elementaroperationen (`read`, `write`, …) aber keine Anwendungslogik

Serialisierbare Historie

Beispiel

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**

Äquivalente serielle Ausführung: $T_1 \mid T_2$

Beispiel

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**

Nicht serialisierbare Historie

Beispiel

Schritt	T_1	T_2
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**	

Anmerkung: Serialisierbarkeit immer aus Sicht des DBMS

- Vorherige Historie aus Sicht des DBMS nicht serialisierbar
- DBMS sieht keine Anwendungslogik
- Auf Grund von Anwendungslogik kann es sein dass ...

Anmerkung: Serialisierbarkeit immer aus Sicht des DBMS

- Vorherige Historie aus Sicht des DBMS nicht serialisierbar
- DBMS sieht keine Anwendungslogik
- Auf Grund von Anwendungslogik kann es sein dass ...
  - ... zwei zur Historie passende Transaktionen den selben Effekt bei serieller Ausführung hätten (siehe nächstes Beispiel)

Anmerkung: Serialisierbarkeit immer aus Sicht des DBMS

- Vorherige Historie aus Sicht des DBMS nicht serialisierbar
- DBMS sieht keine Anwendungslogik
- Auf Grund von Anwendungslogik kann es sein dass ...
  - ... zwei zur Historie passende Transaktionen den selben Effekt bei serieller Ausführung hätten (siehe nächstes Beispiel)
  - ... es für zwei zur Historie passende Transaktionen keine serielle Ausführung mit dem selben Effekt gibt (siehe übernächstes Beispiel)

Serialisierbar: Zwei Überweisungen

Beispiel

Schritt	T_1	T_2
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**	

Serialisierbar: Zwei Überweisungen

Beispiel

Schritt	T_1	T_2
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**	

Eine Überweisung und eine Zinsgutschrift

Beispiel

Schritt	T_1	T_2
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.		$a_2 := a_2 * 1.03$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 * 1.03$
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**	

Eine Überweisung und eine Zinsgutschrift

Beispiel

Schritt	T_1	T_2
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.		$a_2 := a_2 * 1.03$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 * 1.03$
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**	

Serialisierbare Historien

Definition (Serialisierbar —keine abgebrochenen Transaktionen—)

Eine Historie über einer Menge erfolgreich abgeschlossener Transaktionen ist *serialisierbar* falls ihre Auswirkung auf jeder konsistenten Ausprägung identisch ist zu irgendeiner seriellen Historie derselben Transaktionen.

Serialisierbare Historien

Definition (Serialisierbar —keine abgebrochenen Transaktionen—)

Eine Historie über einer Menge erfolgreich abgeschlossener Transaktionen ist *serialisierbar* falls ihre Auswirkung auf jeder konsistenten Ausprägung identisch ist zu irgendeiner seriellen Historie derselben Transaktionen.

Definition (Serialisierbar —mit abgebrochenen Transaktionen—)

Eine Historie über einer Menge Transaktionen ist *serialisierbar* falls ihre Auswirkung auf jeder konsistenten Ausprägung identisch ist zu irgendeiner seriellen Historie der erfolgreichen (`committed`) Transaktionen.

Konfliktserialisierbare Historien

Gesucht: Möglichkeit zu entscheiden:

- Ist eine Historie serialisierbar?
- Welche serielle Ausführung führt zum selben Ergebnis?

Konfliktserialisierbare Historien

Gesucht: Möglichkeit zu entscheiden:

- Ist eine Historie serialisierbar?
- Welche serielle Ausführung führt zum selben Ergebnis?

$\Rightarrow$ Konfliktserialisierbare (*conflict-serializable*) Historien

Konfliktserialisierbare Historien

Definition (konfliktäquivalent/*conflict-equivalent*)

Zwei Historien (über der selben Menge an Transaktionen) sind *konfliktäquivalent* wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge ausführen.

Konfliktserialisierbare Historien

Definition (konfliktäquivalent/*conflict-equivalent*)

Zwei Historien (über der selben Menge an Transaktionen) sind *konfliktäquivalent* wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge ausführen.

Beispiel (Konfliktäquivalente Historien)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Konfliktserialisierbare Historien

Definition (konfliktäquivalent/*conflict-equivalent*)

Zwei Historien (über der selben Menge an Transaktionen) sind *konfliktäquivalent* wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge ausführen.

Beispiel (Konfliktäquivalente Historien)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Konfliktserialisierbare Historien

Definition (konfliktäquivalent/*conflict-equivalent*)

Zwei Historien (über der selben Menge an Transaktionen) sind *konfliktäquivalent* wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge ausführen.

Beispiel (Konfliktäquivalente Historien)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Konfliktserialisierbare Historien

Definition (konfliktäquivalent/*conflict-equivalent*)

Zwei Historien (über der selben Menge an Transaktionen) sind *konfliktäquivalent* wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge ausführen.

Beispiel (Konfliktäquivalente Historien)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Konfliktserialisierbare Historien

Definition (konfliktäquivalent/*conflict-equivalent*)

Zwei Historien (über der selben Menge an Transaktionen) sind *konfliktäquivalent* wenn sie sämtliche Konfliktoperationen der nicht abgebrochenen Transaktionen in der selben Reihenfolge ausführen.

Beispiel (Konfliktäquivalente Historien)

$$r_1(A) \to r_2(C) \to w_1(A) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_2(C) \to w_2(C) \to r_1(B) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to r_2(C) \to w_2(C) \to w_1(B) \to c_1 \to r_2(A) \to w_2(A) \to c_2$$

$$r_1(A) \to w_1(A) \to r_1(B) \to w_1(B) \to c_1 \to r_2(C) \to w_2(C) \to r_2(A) \to w_2(A) \to c_2$$

Konfliktserialisierbare Historien

Definition (Konfliktserialisierbar (*conflict-serializable*))

Eine Historie ist *konfliktserialisierbar* wenn sie konfliktäquivalent zu einer seriellen Historie ist.

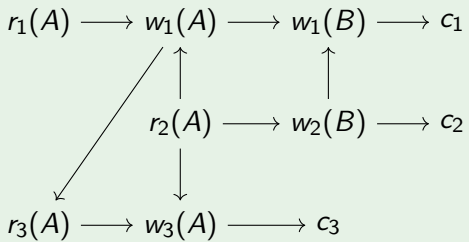
Konfliktserialisierbare Historien

Definition (Konfliktserialisierbar (*conflict-serializable*))

Eine Historie ist *konfliktserialisierbar* wenn sie konfliktäquivalent zu einer seriellen Historie ist.

Beispiel (Konfliktserialisierbare Historie)

$H =$



Serialisierbarkeitsgraph

Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \rightarrow T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

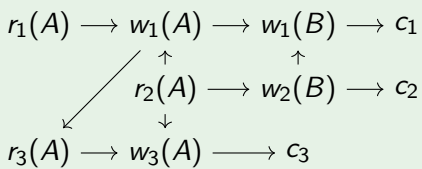
Serialisierbarkeitsgraph

Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel



Serialisierbarkeitsgraph

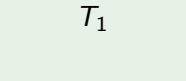
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$$
$$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$$
$$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$$



Serialisierbarkeitsgraph

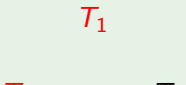
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$$
$$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$$
$$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$$



Serialisierbarkeitsgraph

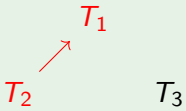
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$$
$$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$$
$$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$$



Serialisierbarkeitsgraph

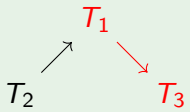
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$
\begin{array}{ccccccc}
r_1(A) & \longrightarrow & w_1(A) & \longrightarrow & w_1(B) & \longrightarrow & c_1 \\
 & & \uparrow & & \uparrow & & \\
 & & r_2(A) & \longrightarrow & w_2(B) & \longrightarrow & c_2 \\
 & & \downarrow & & & & \\
r_3(A) & \longrightarrow & w_3(A) & \longrightarrow & c_3 & &
\end{array}
$$



Serialisierbarkeitsgraph

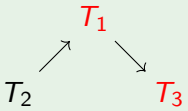
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$
\begin{array}{ccccccc}
r_1(A) & \longrightarrow & w_1(A) & \longrightarrow & w_1(B) & \longrightarrow & c_1 \\
 & \searrow & \uparrow & & \uparrow & & \\
 & & r_2(A) & \longrightarrow & w_2(B) & \longrightarrow & c_2 \\
 & \swarrow & \downarrow & & & & \\
r_3(A) & \longrightarrow & w_3(A) & \longrightarrow & c_3 & &
\end{array}
$$



Serialisierbarkeitsgraph

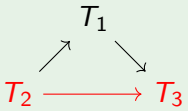
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$$
$$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$$
$$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$$



Serialisierbarkeitsgraph

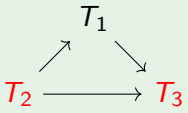
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$$
$$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$$
$$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$$



Serialisierbarkeitsgraph

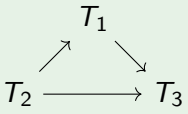
Definition (Serialisierbarkeitsgraph)

Serialisierbarkeitsgraph $SG(H)$ einer Historie H:

- Knoten: erfolgreiche Transaktionen $T_1, T_2, \ldots$ von H
- Gerichtete Kante $T_i \to T_j$ falls für (mindestens) ein Paar (p_i, q_j) von Konfliktoperationen gilt: $p_i <_H q_j$

Beispiel

$$r_1(A) \longrightarrow w_1(A) \longrightarrow w_1(B) \longrightarrow c_1$$
$$r_2(A) \longrightarrow w_2(B) \longrightarrow c_2$$
$$r_3(A) \longrightarrow w_3(A) \longrightarrow c_3$$



Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Theorem

Jede topologische Sortierung von $SG(H)$ gibt eine konfliktäquivalente serielle Historie an.

Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Theorem

Jede topologische Sortierung von $SG(H)$ gibt eine konfliktäquivalente serielle Historie an.

Beispiel

$H = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Theorem

Jede topologische Sortierung von $SG(H)$ gibt eine konfliktäquivalente serielle Historie an.

Beispiel

$H = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(H) = \begin{array}{ccc} & & T_2 \\ T_1 & & \\ & & T_3 \end{array}$$

Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Theorem

Jede topologische Sortierung von $SG(H)$ gibt eine konfliktäquivalente serielle Historie an.

Beispiel

$H = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(H) = \quad T_1 \nearrow \begin{matrix} T_2 \\ \\ T_3 \end{matrix}$$

Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Theorem

Jede topologische Sortierung von $SG(H)$ gibt eine konfliktäquivalente serielle Historie an.

Beispiel

$H = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(H) = \quad T_1 \begin{array}{l} \nearrow T_2 \\ \searrow T_3 \end{array}$$

Serialisierbarkeitstheorem

Theorem

Eine Historie H ist konfliktserialisierbar gdw der Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Theorem

Jede topologische Sortierung von $SG(H)$ gibt eine konfliktäquivalente serielle Historie an.

Beispiel

$H = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

$$SG(H) = \begin{array}{ccc} & \nearrow & T_2 \\ T_1 & & \\ & \searrow & T_3 \end{array}$$

$$H_s^1 = T_1 \mid T_2 \mid T_3$$
$$H_s^2 = T_1 \mid T_3 \mid T_2$$
$$H \equiv H_s^1 \equiv H_s^2$$

Berechnung einer topologischen Sortierung

- Beobachtung: In einem azyklischen, gerichteten Graphen gibt es mindestens einen Knoten ohne eingehende Kante.

Berechnung einer topologischen Sortierung

- Beobachtung: In einem azyklischen, gerichteten Graphen gibt es mindestens einen Knoten ohne eingehende Kante.
- Algorithmus:

In: azyklischer Serialisierbarkeitsgraph $SG(H)$
Out: eine mögliche topologische Sortierung

1: Wähle in $SG(H)$ einen Knoten T_i ohne eingehende Kante
2: Schreibe T_i in den Output
3: Lösche T_i und alle von T_i ausgehenden Kanten aus $SG(H)$
4: **if** es ist noch ein Knoten übrig **then**
5: **goto** 1
6: **end if**

Inhalt

Weitere Eigenschaften von Historien

Falls keine Transaktion abbricht:

- Jede konfliktserialisierbare Historie ist serialisierbar

Weitere Eigenschaften von Historien

Falls keine Transaktion abbricht:

- Jede konfliktserialisierbare Historie ist serialisierbar

Achtung: Gegenteil muss nicht der Fall sein:

Weitere Eigenschaften von Historien

Falls keine Transaktion abbricht:

- Jede konfliktserialisierbare Historie ist serialisierbar

Achtung: Gegenteil muss nicht der Fall sein:

Beispiel

	T_1	T_2	T_3
1.	read(A)		
2.		write(A)	
3.		**commit**	
4.	write(A)		
5.	**commit**		
6.			write(A)
7.			**commit**

Weitere Eigenschaften von Historien

Falls Transaktionen abbrechen können:

- Nicht alle konfliktserialisierbaren Historien sind serialisierbar

Weitere Eigenschaften von Historien

Falls Transaktionen abbrechen können:

- Nicht alle konfliktserialisierbaren Historien sind serialisierbar

Beispiel

	T_1	T_2
1.	read(A)	
2.	write(A)	
3.		read(A)
4.		write(A)
5.		read(B)
6.		write(B)
7.		**commit**
8.	**abort**	

Historien bei abgebrochenen Transaktionen

Konfliktserialisierbar in DBMS meist die Minimalanforderung

Historien bei abgebrochenen Transaktionen

Konfliktserialisierbar in DBMS meist die Minimalanforderung

Zusätzliche Eigenschaften von/Anforderungen an Historien:

- Rücksetzbare Historien
- Historien ohne kaskadierendes Rücksetzen
- Strikte Historien

Rücksetzbare Historien

Minimalanforderung bzgl. Recovery:

- Abbruch einer aktiven Transaktion möglich ohne Einfluss auf erfolgreich abgeschlossene Transaktionen.

Rücksetzbare Historien

Minimalanforderung bzgl. Recovery:

- Abbruch einer aktiven Transaktion möglich ohne Einfluss auf erfolgreich abgeschlossene Transaktionen.

Rücksetzbare Historien (informell): Eine Transaktion darf erst mit `commit` abschließen nachdem alle Transaktionen von denen sie gelesen hat ebenfalls abgeschlossen sind.

Rücksetzbare Historien

Minimalanforderung bzgl. Recovery:

- Abbruch einer aktiven Transaktion möglich ohne Einfluss auf erfolgreich abgeschlossene Transaktionen.

Rücksetzbare Historien (informell): Eine Transaktion darf erst mit `commit` abschließen nachdem alle Transaktionen von denen sie gelesen hat ebenfalls abgeschlossen sind.

Schreib/Leseabhängigkeiten zwischen Historien

Definition (T_i liest von T_j)

Eine Transaktion T_i *liest von einer Transaktion* T_j in einer Historie H wenn es mindestens ein Datum A gibt das folgendes erfüllt:

Schreib/Leseabhängigkeiten zwischen Historien

Definition (T_i liest von T_j)

Eine Transaktion T_i *liest von einer Transaktion* T_j in einer Historie H wenn es mindestens ein Datum A gibt das folgendes erfüllt:

1. $w_j(A) <_H r_i(A)$:
   T_j schreibt mindestens ein Datum A das T_i später liest.

Schreib/Leseabhängigkeiten zwischen Historien

Definition (T_i liest von T_j)

Eine Transaktion T_i *liest von einer Transaktion* T_j in einer Historie H wenn es mindestens ein Datum A gibt das folgendes erfüllt:

1. $w_j(A) <_H r_i(A)$:
   T_j schreibt mindestens ein Datum A das T_i später liest.
2. $a_j \not<_H r_i(A)$:
   T_j wird nicht zurückgesetzt bevor T_i Datum A gelesen hat.

Schreib/Leseabhängigkeiten zwischen Historien

Definition (T_i liest von T_j)

Eine Transaktion T_i *liest von einer Transaktion* T_j in einer Historie H wenn es mindestens ein Datum A gibt das folgendes erfüllt:

1. $w_j(A) <_H r_i(A)$:
   T_j schreibt mindestens ein Datum A das T_i später liest.
2. $a_j \not<_H r_i(A)$:
   T_j wird nicht zurückgesetzt bevor T_i Datum A gelesen hat.
3. Wenn ein $w_k(A)$ mit $w_j(A) <_H w_k(A) <_H r_i(A)$ existiert, dann auch $a_k <_H r_i(A)$:
   Alle zwischenzeitlichen Schreibvorgänge anderer Transaktionen auf A werden vor dem Lesen von T_i zurückgesetzt.

Schreib/Leseabhängigkeiten zwischen Historien

Definition (T_i liest von T_j)

Eine Transaktion T_i *liest von einer Transaktion* T_j in einer Historie H wenn es mindestens ein Datum A gibt das folgendes erfüllt:

1. $w_j(A) <_H r_i(A)$:
   T_j schreibt mindestens ein Datum A das T_i später liest.
2. $a_j \not<_H r_i(A)$:
   T_j wird nicht zurückgesetzt bevor T_i Datum A gelesen hat.
3. Wenn ein $w_k(A)$ mit $w_j(A) <_H w_k(A) <_H r_i(A)$ existiert, dann auch $a_k <_H r_i(A)$:
   Alle zwischenzeitlichen Schreibvorgänge anderer Transaktionen auf A werden vor dem Lesen von T_i zurückgesetzt.

Intuition: T_i liest genau den Wert von A den T_j geschrieben hat.

Rücksetzbare Historien

Definition (Rücksetzbare Historien)

Eine Historie H heißt *rücksetzbar*, wenn für alle Paare T_i, T_j von Transaktionen in H gilt:

- Wenn T_i von T_j liest und T_i ein `commit` durchführt, so muss diesem das `commit` von T_j vorausgegangen sein.
  (T_i liest von T_j und $c_i \Rightarrow c_j <_H c_i$)

Rücksetzbare Historien

Definition (Rücksetzbare Historien)

Eine Historie H heißt *rücksetzbar*, wenn für alle Paare T_i, T_j von Transaktionen in H gilt:

- Wenn T_i von T_j liest und T_i ein `commit` durchführt, so muss diesem das `commit` von T_j vorausgegangen sein.
  (T_i liest von T_j und $c_i \Rightarrow c_j <_H c_i$)

Alternativ: Eine Transaktion darf erst dann ihr `commit` ausführen wenn alle Transaktionen, von denen sie gelesen hat, beendet sind.

Problem: Kaskadierendes Rücksetzen

Beispiel (Historie mit kaskadierendem Rücksetzen)

Schritt	T_1	T_2	T_3	T_4	T_5
0.	...				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	a_1 (**abort**)				

Historien ohne kaskadierendes Rücksetzen

Definition (Historie ohne kaskadierendes Rücksetzen)

Eine Historie H *vermeidet kaskadierendes Rücksetzen* wenn

- $c_j <_H r_i(A)$ gilt, wann immer T_i ein Datum A von T_j liest.
  (T_i liest von T_j erst nachdem T_j `committed` wurde)

Historien ohne kaskadierendes Rücksetzen

Definition (Historie ohne kaskadierendes Rücksetzen)

Eine Historie H *vermeidet kaskadierendes Rücksetzen* wenn

- $c_j <_H r_i(A)$ gilt, wann immer T_i ein Datum A von T_j liest.
  (T_i liest von T_j erst nachdem T_j `committed` wurde)

Alternativ: Änderungen einer Transaktion werden erst nach einem `commit` zum Lesen freigegeben.

Historien ohne kaskadierendes Rücksetzen

Definition (Historie ohne kaskadierendes Rücksetzen)

Eine Historie H *vermeidet kaskadierendes Rücksetzen* wenn

- $c_j <_H r_i(A)$ gilt, wann immer T_i ein Datum A von T_j liest.
  (T_i liest von T_j erst nachdem T_j `committed` wurde)

Problem (Lesen nicht ausreichend)

- Was passiert, wenn T_1 schreibt, dann schreibt T_2, T_2 liest aber nicht von T_1?
- Problem mit physische Protokollierung,
  da Ergebnis von T_2 nach Rücksetzen von T_1 verloren

$\Rightarrow$ Benötigt strengere Anforderung (strikte Historie)

Strikte Historien

Definition (Strikte Historie)

Eine Historie H heißt *strikt* wenn für alle Paare T_i, T_j von Transaktionen in H gilt:

- Falls $w_j(A) <_H o_i(A)$ (mit $o_i \in \{r_i, w_i\}$), dann gilt entweder
  - $c_j <_H o_i(A)$ oder
  - $a_j <_H o_i(A)$.

Strikte Historien

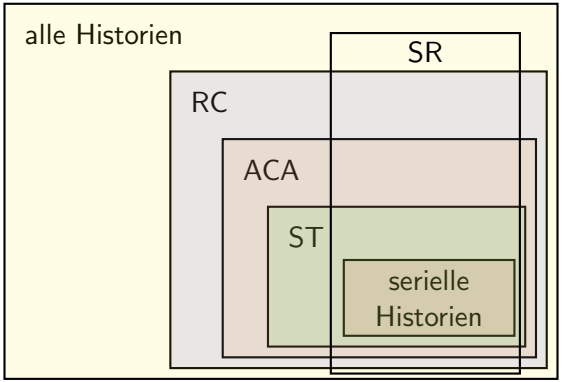
Definition (Strikte Historie)

Eine Historie H heißt *strikt* wenn für alle Paare T_i, T_j von Transaktionen in H gilt:

- Falls $w_j(A) <_H o_i(A)$ (mit $o_i \in \{r_i, w_i\}$), dann gilt entweder
  - $c_j <_H o_i(A)$ oder
  - $a_j <_H o_i(A)$.

Alternativ: Auf ein von einer Transaktion geschriebenes Datum dürfen anderen Transaktionen erst nach deren Beendigung (`commit` oder `abort`) schreibend oder lesend zugreifen.

Beziehungen zwischen den Klassen von Historien



SR (Konflikt-) Serialisierbare Historien (*SeRializable*)

RC Rücksetzbare Historien (*ReCoverable*)

ACA Historien ohne kaskadierendes Rücksetzen (*Avoid Cascading Abort*)

ST Strikte Historien (*STrict*)

Inhalt

Datenbank-Scheduler

Bestimmt Ausführungsreihenfolge der Einzeloperationen

- Mindestanforderung: Resultierende Historie sollte konfliktserialisierbar sein.
- Üblicherweise werden nur strikte, konfliktserialisierbare Historien erzeugt.
- Realisierung:
  - weit verbreitet: sperrbasierte Synchronisation
  - weitere Methoden: Zeitstempel-basierte Synchronisation, optimistische Synchronisation, MVCC

Inhalt

Sperrbasierte Synchronisationsprotokolle

- Zwei-Phasen-Sperrprotokoll (*2PL*)
- Strenges Zwei-Phasen-Sperrprotokoll (*strict 2PL*)
- Zwei-Phasen-Sperrprotokoll mit Preclaiming (*conservative 2PL*)
- Sperrprotokolle mit Zeitstempel
- Hierarchische Sperrgranulate (*MGL*)

Sperrmodi

Zwei Sperrmodi:

S Shared, read lock, Lesesperre

Sperrmodi

Zwei Sperrmodi:

- S Shared, read lock, Lesesperre
- X eXclusive, write lock, Schreibsperre

Sperrmodi

Zwei Sperrmodi:

S Shared, read lock, Lesesperre

X eXclusive, write lock, Schreibsperre

Verträglichkeitsmatrik (Kompatibilitätsmatrix)

Sperre		aktuell		
		NL	S	X
angefordert	S	✓	✓	–
	X	✓	–	–

NL . . . *No Lock*

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion beachtet folgende Regeln:

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion beachtet folgende Regeln:

1. Jedes von einer Transaktion benützte Objekt muss vor Verwendung entsprechend gesperrt werden.

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion beachtet folgende Regeln:

1. Jedes von einer Transaktion benützte Objekt muss vor Verwendung entsprechend gesperrt werden.
2. Eine Transaktion fordert keine Sperre an, die sie schon besitzt.

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion beachtet folgende Regeln:

1. Jedes von einer Transaktion benützte Objekt muss vor Verwendung entsprechend gesperrt werden.
2. Eine Transaktion fordert keine Sperre an, die sie schon besitzt.
3. Sperren werden nach der Kompatibilitätsmatrix gewährt. Kann eine Sperre nicht gewährt werden, wartet die Transaktion bis die Sperre gewährt werden kann.

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion beachtet folgende Regeln:

1. Jedes von einer Transaktion benützte Objekt muss vor Verwendung entsprechend gesperrt werden.
2. Eine Transaktion fordert keine Sperre an, die sie schon besitzt.
3. Sperren werden nach der Kompatibilitätsmatrix gewährt. Kann eine Sperre nicht gewährt werden, wartet die Transaktion bis die Sperre gewährt werden kann.
4. Nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren mehr anfordern.

Zwei-Phasen-Sperrprotokoll (2PL)

Jede Transaktion beachtet folgende Regeln:

1. Jedes von einer Transaktion benützte Objekt muss vor Verwendung entsprechend gesperrt werden.
2. Eine Transaktion fordert keine Sperre an, die sie schon besitzt.
3. Sperren werden nach der Kompatibilitätsmatrix gewährt. Kann eine Sperre nicht gewährt werden, wartet die Transaktion bis die Sperre gewährt werden kann.
4. Nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren mehr anfordern.
5. Am Transaktionsende (EOT) müssen alle Sperren freigegeben worden sein.

Zwei Phasen: Wachstum und Schrumpfung

Wichtig zu beachten:

4. Nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren mehr anfordern.

Zwei Phasen: Wachstum und Schrumpfung

Wichtig zu beachten:

4. Nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren mehr anfordern.

Wachstumsphase Transaktion darf Sperren anfordern, aber keine Sperren freigeben.

Zwei Phasen: Wachstum und Schrumpfung

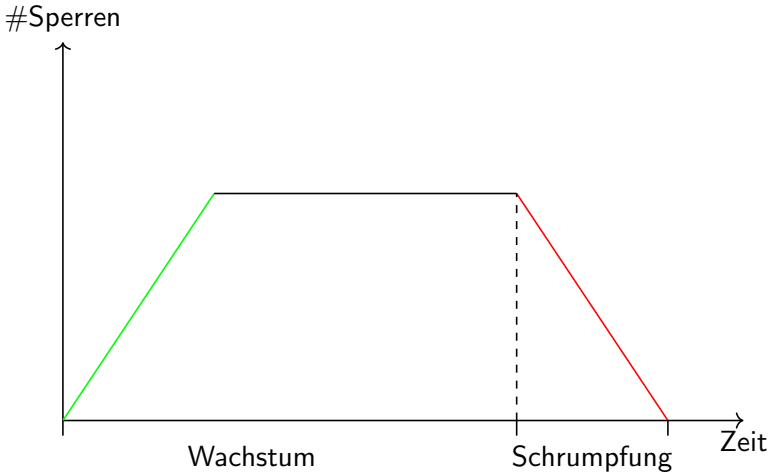
Wichtig zu beachten:

4. Nachdem eine Transaktion eine Sperre freigegeben hat darf sie keine weiteren Sperren mehr anfordern.

Wachstumsphase Transaktion darf Sperren anfordern, aber keine Sperren freigeben.

Schrumpfungsphase Transaktion darf erworbene Sperren freigeben, aber keine weiteren mehr anfordern.

Zwei Phasen: Wachstum und Schrumpfung



Verzahnung zweier Transaktionen gemäß 2PL

Beispiel (Einhaltung des 2PL)

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 muss warten
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 wecken
10.		read(A)	
11.		**lockS**(B)	T_2 muss warten
12.	write(B)		
13.	**unlockX**(B)		T_2 wecken
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

Zwei Phasen garantieren Konfliktserialisierbarkeit

Beispiel (Bei Nichteinhaltung des 2PL)

Schritt	T_1	T_2
1.		
2.	read(A)	
3.	write(A)	
4.		
5.		
6.		
7.		read(A)
8.		write(A)
9.		read(B)
10.		write(B)
9.		
6.		
10.		
11.	read(B)	
12.	write(B)	
13.		

Zwei Phasen garantieren Konfliktserialisierbarkeit

Beispiel (Bei Nichteinhaltung des 2PL)

Schritt	T_1	T_2
1.	**lockX**(A)	
2.	read(A)	
3.	write(A)	
4.	**unlockX**(A)	
5.		**lockX**(A)
6.		**lockX**(B)
7.		read(A)
8.		write(A)
9.		read(B)
10.		write(B)
9.		**unlockX**(A)
6.		**unlockX**(B)
10.	**lockX**(B)	
11.	read(B)	
12.	write(B)	
13.	**unlockX**(B)	

Eigenschaften von Historien nach 2PL

2PL garantiert Konfliktserialisierbarkeit:

Eigenschaften von Historien nach 2PL

2PL garantiert Konfliktserialisierbarkeit:

- Die Reihenfolge der Transaktionen in einer (konflikt-) äquivalenten seriellen Ausführung ergibt sich aus der Reihenfolge der Sperranforderungen bei Konflikten.

Eigenschaften von Historien nach 2PL

2PL garantiert Konfliktserialisierbarkeit:

- Die Reihenfolge der Transaktionen in einer (konflikt-) äquivalenten seriellen Ausführung ergibt sich aus der Reihenfolge der Sperranforderungen bei Konflikten.
- Widersprüchliche Reihenfolgen der Sperranforderungen führen zu einem Deadlock

Eigenschaften von Historien nach 2PL

2PL garantiert Konfliktserialisierbarkeit:

- Die Reihenfolge der Transaktionen in einer (konflikt-) äquivalenten seriellen Ausführung ergibt sich aus der Reihenfolge der Sperranforderungen bei Konflikten.
- Widersprüchliche Reihenfolgen der Sperranforderungen führen zu einem Deadlock

2PL garantiert nicht Rücksetzbarkeit:

- Freigaben von Sperren vor Transaktionsende möglich
  $\Rightarrow$ Andere Transaktion kann zugreifen und committen.

Nicht rücksetzbare Historie bei 2PL

Beispiel

Schritt	T_1	T_2	Bemerkung
1.			
2.			
3.			
4.			
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Nicht rücksetzbare Historie bei 2PL

Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.			
6.			
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Nicht rücksetzbare Historie bei 2PL

Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 muss warten
7.			
8.			
9.			
10.			
11.			
12.			
13.			
14.			

Nicht rücksetzbare Historie bei 2PL

Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 muss warten
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wecken
9.			
10.			
11.			
12.			
13.			
14.			

Nicht rücksetzbare Historie bei 2PL

Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 muss warten
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wecken
9.	read(B)	read(A)	„T_2 liest von T_1“
10.			
11.			
12.			
13.			
14.			

Nicht rücksetzbare Historie bei 2PL

Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 muss warten
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wecken
9.	read(B)	read(A)	„T_2 liest von T_1“
10.		write(A)	
11.		**unlockX**(A)	
12.		**commit**	
13.			
14.			T_2 nicht mehr rücksetzbar!

Nicht rücksetzbare Historie bei 2PL

Beispiel

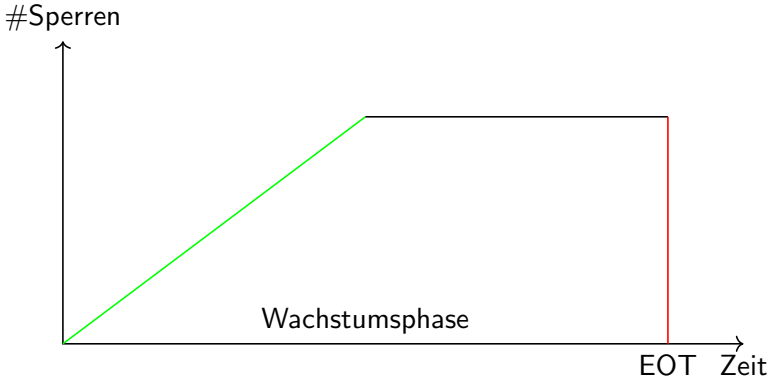
Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 muss warten
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wecken
9.	read(B)	read(A)	„T_2 liest von T_1“
10.		write(A)	
11.		**unlockX**(A)	
12.	write(B)	**commit**	
13.	**unlockX**(B)		
14.	**abort**		T_2 nicht mehr rücksetzbar!

Strenges Zwei-Phasen-Sperrprotokoll (strict 2PL)

- Alle Sperren werden bis Transaktionsende gehalten.

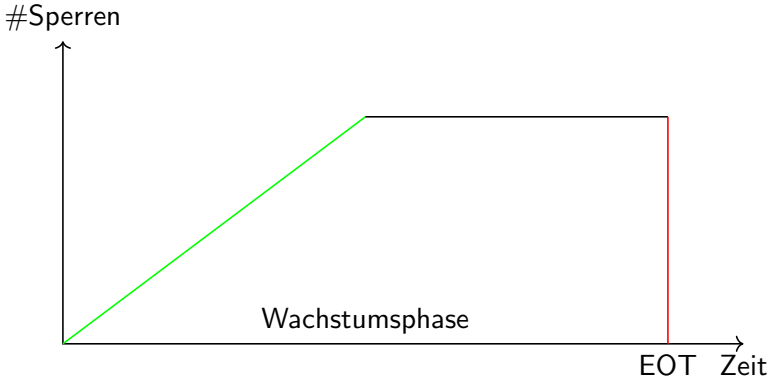
Strenges Zwei-Phasen-Sperrprotokoll (strict 2PL)

- Alle Sperren werden bis Transaktionsende gehalten.



Strenges Zwei-Phasen-Sperrprotokoll (strict 2PL)

- Alle Sperren werden bis Transaktionsende gehalten.
- Das strenge 2PL-Protokoll lässt nur strikte Historien zu.



Inhalt

Verklemmungen (Deadlocks)

Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.		**BOT**	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 muss auf T_2 warten
9.		**lockS**(A)	T_2 muss auf T_1 warten
10.	...	...	$\Rightarrow$ *Deadlock*

Deadlock-Erkennung

Einfache Methode: Time-out Strategie

- Transaktion wird zurückgesetzt wenn sie innerhalb einer festgesetzten Zeit keine Fortschritte macht.

Deadlock-Erkennung

Einfache Methode: Time-out Strategie

- Transaktion wird zurückgesetzt wenn sie innerhalb einer festgesetzten Zeit keine Fortschritte macht.
- Problem: “gute” Wahl des Timeouts:
  - zu klein: unnötiges Rücksetzen
  - zu groß: Deadlocks werden spät erkannt

Deadlock-Erkennung

Einfache Methode: Time-out Strategie

- Transaktion wird zurückgesetzt wenn sie innerhalb einer festgesetzten Zeit keine Fortschritte macht.
- Problem: "gute" Wahl des Timeouts:
  - zu klein: unnötiges Rücksetzen
  - zu groß: Deadlocks werden spät erkannt

Exakte Methode: Wartegraph

- Knoten: aktive Transaktionen $T_1, T_2, \ldots$
- Kanten: gerichtete Kante $T_i \rightarrow T_j$ falls T_i auf T_j wartet.

Deadlock-Erkennung

Einfache Methode: Time-out Strategie

- Transaktion wird zurückgesetzt wenn sie innerhalb einer festgesetzten Zeit keine Fortschritte macht.
- Problem: "gute" Wahl des Timeouts:
  - zu klein: unnötiges Rücksetzen
  - zu groß: Deadlocks werden spät erkannt

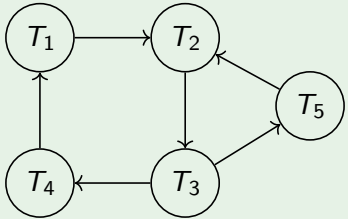
Exakte Methode: Wartegraph

- Knoten: aktive Transaktionen $T_1, T_2, \ldots$
- Kanten: gerichtete Kante $T_i \rightarrow T_j$ falls T_i auf T_j wartet.
- Deadlock: gdw. Wartegraph einen Zyklus enthält.

Wartegraph

Beispiel (Wartegraph mit zwei Zyklen)

1. $T_1 \to T_2 \to T_3 \to T_4 \to T_1$
2. $T_2 \to T_3 \to T_5 \to T_2$



Beide Zyklen können durch Rücksetzen von T_3 aufgelöst werden.

Deadlock-Vermeidung

"Preclaiming": Transaktionen werden nur gestartet wenn alle benötigten Sperren bereits am Beginn gewährt werden können.

Deadlock-Vermeidung

"Preclaiming": Transaktionen werden nur gestartet wenn alle benötigten Sperren bereits am Beginn gewährt werden können.

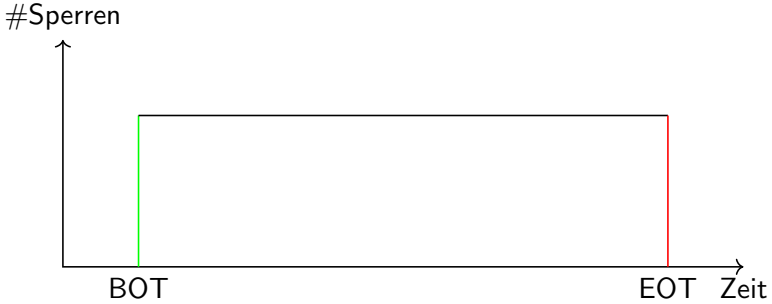
Zeitstempelverfahren: Jeder Transaktion wird ein Zeitstempel zugeordnet welcher (abhängig von der konkreten Strategie) entscheidet ob eine Transaktion auf eine Sperre wartet oder nicht.

Conservative 2PL (Preclaiming)

- Variante von strict 2PL
- Transaktion fordert alle Sperren die sie brauchen wird bereits zu Beginn, und hält alle Sperren bis zum Schluss.

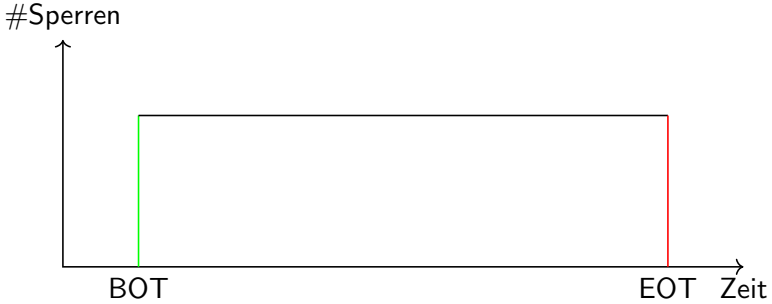
Conservative 2PL (Preclaiming)

- Variante von strict 2PL
- Transaktion fordert alle Sperren die sie brauchen wird bereits zu Beginn, und hält alle Sperren bis zum Schluss.



Conservative 2PL (Preclaiming)

- Variante von strict 2PL
- Transaktion fordert alle Sperren die sie brauchen wird bereits zu Beginn, und hält alle Sperren bis zum Schluss.



- Problem: Benötigte Sperren sind im allgemeinen zu Transaktionsbeginn nicht (exakt) bekannt.

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

wound-wait Strategie Ältere Transaktionen setzen sich durch

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

wound-wait Strategie Ältere Transaktionen setzen sich durch

- T_1 älter als T_2: T_2 wird abgebrochen.

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

wound-wait Strategie Ältere Transaktionen setzen sich durch

- T_1 älter als T_2: T_2 wird abgebrochen.
- Sonst: T_1 wartet auf Freigabe der Sperre durch T_2.

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

wound-wait Strategie Ältere Transaktionen setzen sich durch

- T_1 älter als T_2: T_2 wird abgebrochen.
- Sonst: T_1 wartet auf Freigabe der Sperre durch T_2.

wait-die Strategie Jüngere Transaktionen sind "schüchtern"

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

wound-wait Strategie Ältere Transaktionen setzen sich durch

- T_1 älter als T_2: T_2 wird abgebrochen.
- Sonst: T_1 wartet auf Freigabe der Sperre durch T_2.

wait-die Strategie Jüngere Transaktionen sind "schüchtern"

- T_1 älter als T_2: T_1 wartet auf die Freigabe der Sperre

Zeitstempelverfahren – zwei Strategien

- Jede Transaktion T_i erhält eindeutigen Zeitstempel $TS(T_i)$
- Jüngere Transaktionen erhalten größere Zeitstempel

T_1 will eine Sperre erwerben die von T_2 gehalten wird:

wound-wait Strategie Ältere Transaktionen setzen sich durch

- T_1 älter als T_2: T_2 wird abgebrochen.
- Sonst: T_1 wartet auf Freigabe der Sperre durch T_2.

wait-die Strategie Jüngere Transaktionen sind "schüchtern"

- T_1 älter als T_2: T_1 wartet auf die Freigabe der Sperre
- Sonst: T_1 wird abgebrochen.

Inhalt

Hierarchische Sperrgranulate

Multiple Granularity Locking (MGL):

- Bislang: Einheitliche Sperrgranulate
- Effizienter: Unterschiedliche Sperrgranulate

Hierarchische Sperrgranulate

Multiple Granularity Locking (MGL):

- Bislang: Einheitliche Sperrgranulate
- Effizienter: Unterschiedliche Sperrgranulate

Benötigt:

- Erweiterte Sperrmodi
- Erweitertes Sperrprotokoll

MGL: Multiple-Granularity Locking

Problem: Einheitliche Sperrgranulate (z.B.: pro Datensatz, pro Tabelle, pro Seite, ...) können ineffizient sein. Ist die Granularität

MGL: Multiple-Granularity Locking

Problem: Einheitliche Sperrgranulate (z.B.: pro Datensatz, pro Tabelle, pro Seite, ...) können ineffizient sein. Ist die Granularität

- zu klein: langsam bei Transaktionen mit vielen Datenzugriffen

MGL: Multiple-Granularity Locking

Problem: Einheitliche Sperrgranulate (z.B.: pro Datensatz, pro Tabelle, pro Seite, ...) können ineffizient sein. Ist die Granularität

- zu klein: langsam bei Transaktionen mit vielen Datenzugriffen
- zu groß: blockiert womöglich mehr Transaktionen als nötig

Idee: Mehr Flexibilität durch unterschiedliche Sperrgranulate.

- DBMS oder Benutzerin wählt passende Granularität (Datensatz, Tabelle, Seite, Segmente, Datenbasis)

MGL: Multiple-Granularity Locking

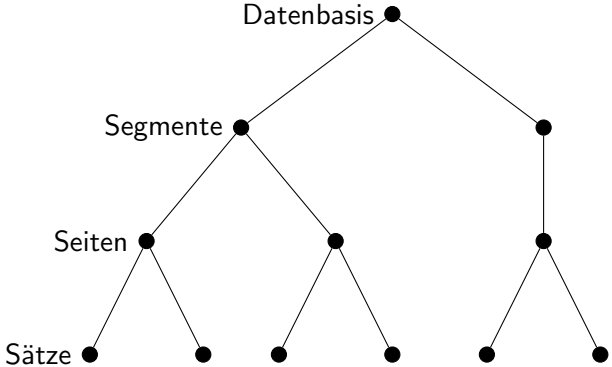
Problem: Einheitliche Sperrgranulate (z.B.: pro Datensatz, pro Tabelle, pro Seite, ...) können ineffizient sein. Ist die Granularität

- zu klein: langsam bei Transaktionen mit vielen Datenzugriffen
- zu groß: blockiert womöglich mehr Transaktionen als nötig

Idee: Mehr Flexibilität durch unterschiedliche Sperrgranulate.

- DBMS oder Benutzerin wählt passende Granularität (Datensatz, Tabelle, Seite, Segmente, Datenbasis)
- "lock escalation": Anfangs Sperren kleiner Granularität; mit steigender Menge Sperren höherer Granularität

Hierarchische Anordnung möglicher Sperrgranulate



Segment:
mehrere (logisch zusammengehörende) Seiten (insb. Tabelle)

Erweiterte Sperrmodi bei MGL

Neues Problem: "Übersicht" über Sperren kleiner Granularität.

- Ohne weitere Vorkehrung: Vor Vergabe einer Sperre müssten alle darunterliegenden Objekte auf Sperren geprüft werden.

Erweiterte Sperrmodi bei MGL

Neues Problem: "Übersicht" über Sperren kleiner Granularität.

- Ohne weitere Vorkehrung: Vor Vergabe einer Sperre müssten alle darunterliegenden Objekte auf Sperren geprüft werden.

Lösung: Zusätzliche Sperrmodi (IS, IX) welche anzeigen dass weiter unten in der Hierarchie bestimmte Sperren existieren.

Sperrmodi bei MGL

Sperrmodi bei MGL:

- NL Keine Sperre (*No Lock*)
- S Lesesperre (*Shared lock*)
- X Schreibsperre (*eXclusive lock*)

Sperrmodi bei MGL

Sperrmodi bei MGL:

- NL Keine Sperre (*No Lock*)
- S Lesesperre (*Shared lock*)
- X Schreibsperre (*eXclusive lock*)
- IS Tiefer in der Hierarchie ist eine Lesesperre (S) beabsichtigt (*Intention Shared lock*)

Sperrmodi bei MGL

Sperrmodi bei MGL:

- NL Keine Sperre (*No Lock*)
- S Lesesperre (*Shared lock*)
- X Schreibsperre (*eXclusive lock*)
- IS Tiefer in der Hierarchie ist eine Lesesperre (S) beabsichtigt (*Intention Shared lock*)
- IX Tiefer in der Hierarchie ist eine Schreibsperre (X) beabsichtigt (*Intention eXclusive lock*)

Verträglichkeitsmatrix (Kompatibilitätsmatrix) bei MGL

Sperre		aktuell				
		NL	S	X	IS	IX
angefordert	S	✓	✓	–	✓	–
	X	✓	–	–	–	–
	IS	✓	✓	–	✓	✓
	IX	✓	–	–	✓	✓

Sperrprotokoll des MGL

Anforderung benötigter Sperren: top-down

Sperrprotokoll des MGL

Anforderung benötigter Sperren: top-down

- Bevor eine Transaktion einen Knoten mit S oder IS sperren kann, benötigt sie für alle Vorgänger in der Hierarchie eine IS- oder IX- Sperre.

Sperrprotokoll des MGL

Anforderung benötigter Sperren: top-down

- Bevor eine Transaktion einen Knoten mit S oder IS sperren kann, benötigt sie für alle Vorgänger in der Hierarchie eine IS- oder IX- Sperre.
- Bevor eine Transaktion einen Knoten mit X oder IX sperren kann, benötigt sie für alle Vorgänger eine IX- Sperre.

Sperrprotokoll des MGL

Anforderung benötigter Sperren: top-down

- Bevor eine Transaktion einen Knoten mit S oder IS sperren kann, benötigt sie für alle Vorgänger in der Hierarchie eine IS- oder IX- Sperre.
- Bevor eine Transaktion einen Knoten mit X oder IX sperren kann, benötigt sie für alle Vorgänger eine IX- Sperre.

Freigabe von Sperren: bottom-up

Sperrprotokoll des MGL

Anforderung benötigter Sperren: top-down

- Bevor eine Transaktion einen Knoten mit S oder IS sperren kann, benötigt sie für alle Vorgänger in der Hierarchie eine IS- oder IX- Sperre.
- Bevor eine Transaktion einen Knoten mit X oder IX sperren kann, benötigt sie für alle Vorgänger eine IX- Sperre.

Freigabe von Sperren: bottom-up

- Eine IS- oder IX-Sperre an einem Knoten darf erst freigegeben werden wenn alle Sperren auf Nachfolgeknoten bereits freigegeben worden sind.

Datenbasishierarchie mit Sperren

Beispiel (MGL Protokoll)

Level:
Datenbasis, Segmente (*areas*), Seiten (*pages*), Datensätze

3 Transaktionen:

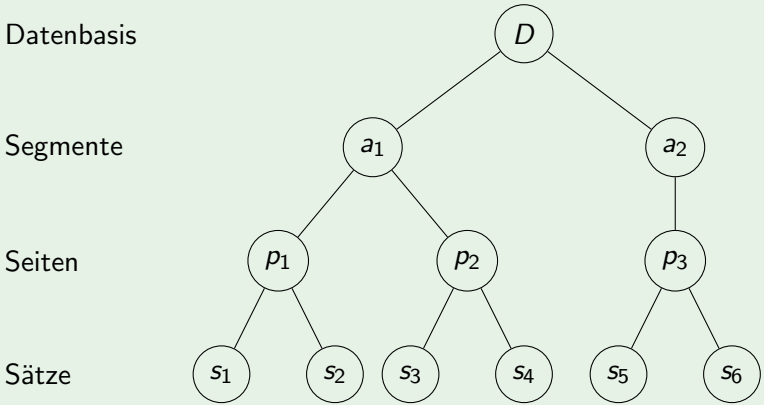
T_1 exclusive lock für Seite p_1 unterhalb von Segment a_1

T_2 shared lock für Seite p_2 unterhalb von Segment a_1

T_3 exclusive lock für Segment a_2

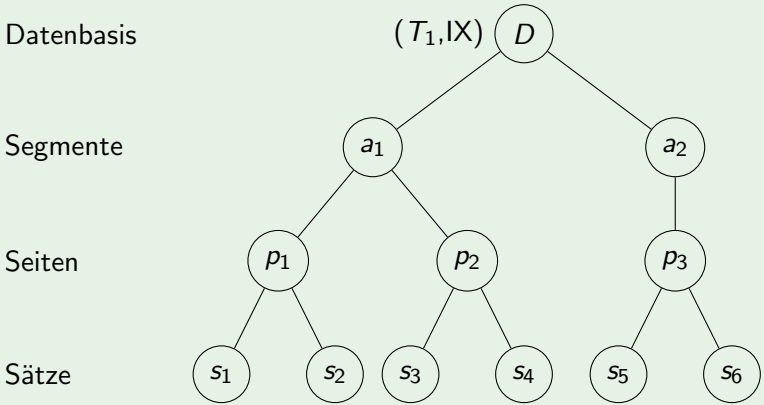
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



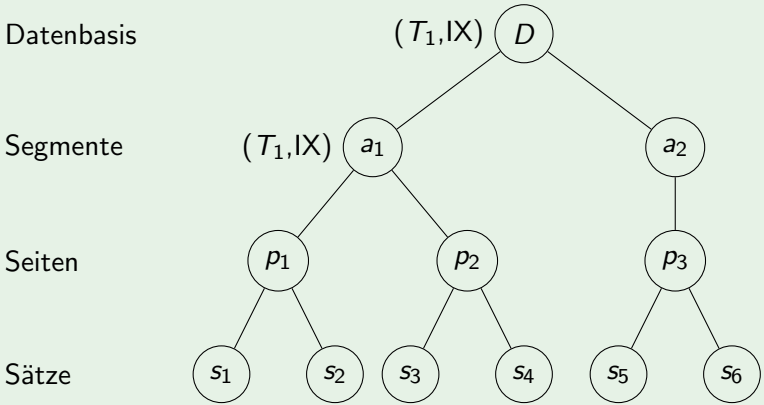
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



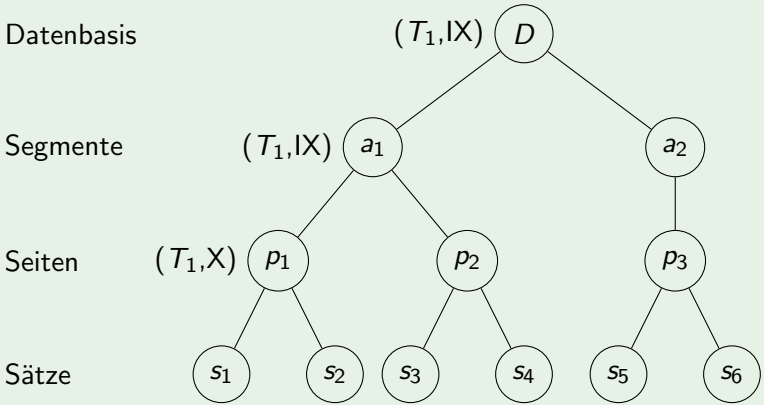
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



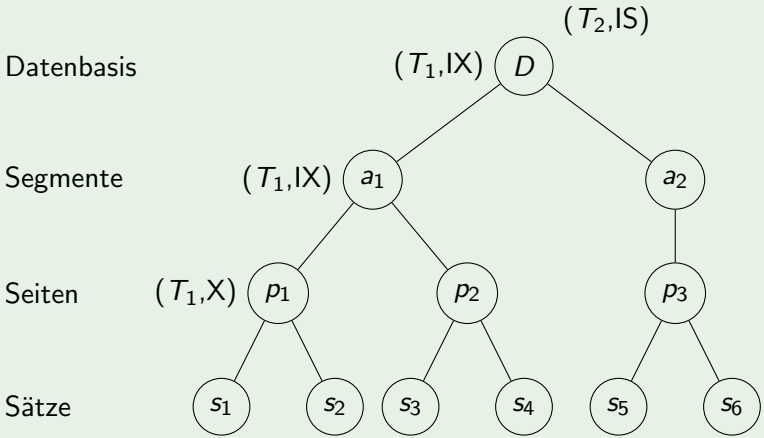
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



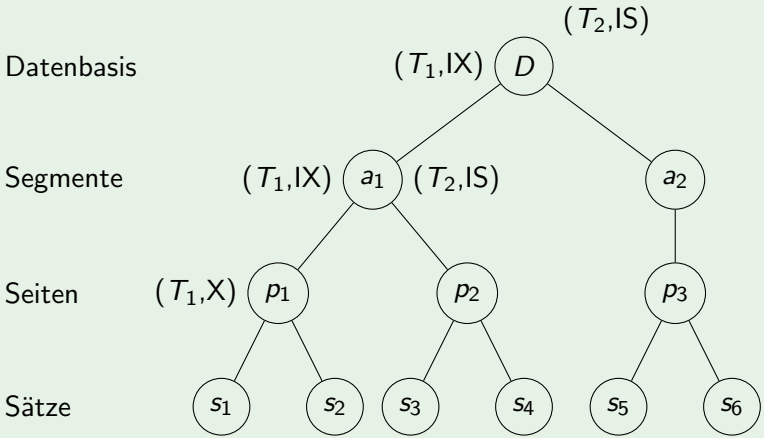
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



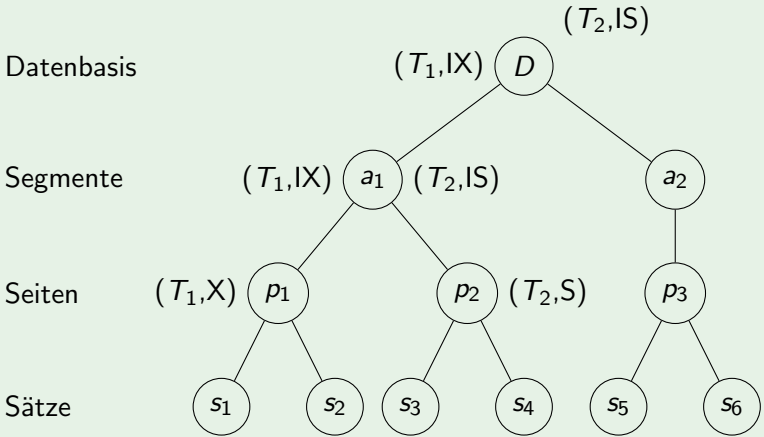
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



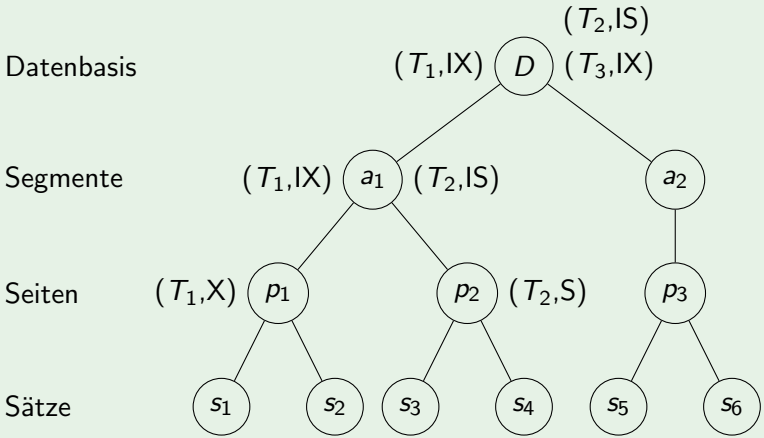
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



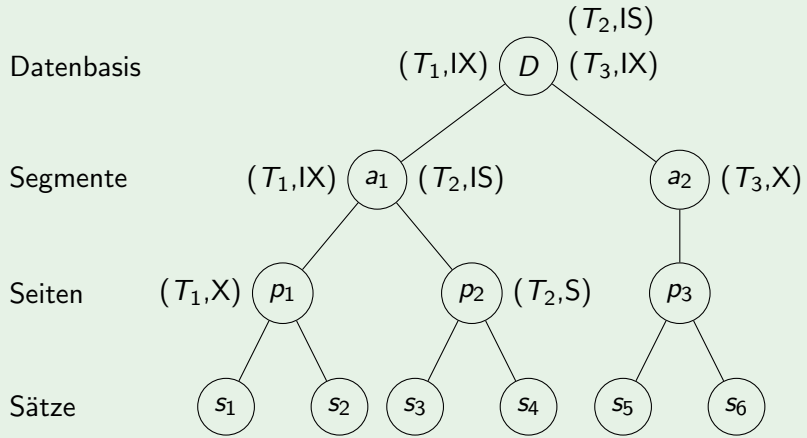
Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



Datenbasishierarchie mit Sperren

Beispiel (Anwendung des MGL Protokolls)



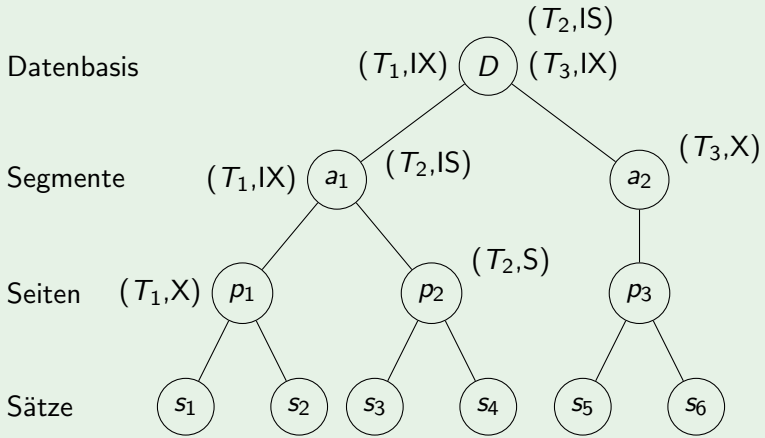
MGL blockiert Transaktionen

Beispiel (Blockierung bei MGL)

- 3 Transaktionen:
  - T_1 exclusive lock für Seite p_1 unterhalb von Segment a_1
  - T_2 shared lock für Seite p_2 unterhalb von Segment a_1
  - T_3 exclusive lock für Segment a_2
- Fortsetzung:
  - T_4 exclusive lock für Satz s_3 unterhalb von Seite p_2: IX-lock Anforderung für p_2 scheitert wegen S-Sperre von T_2.
  - T_5 shared lock für Satz s_5 unterhalb von Seite p_3 unterhalb von Segment a_2: IS-lock Forderung für a_2 scheitert wegen X-Sperre von T_3.

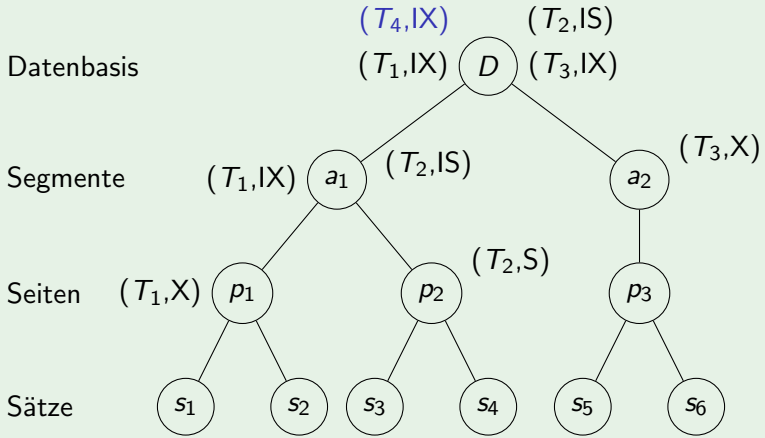
MGL blockiert Transaktionen

Beispiel (MGL blockiert Transaktionen)



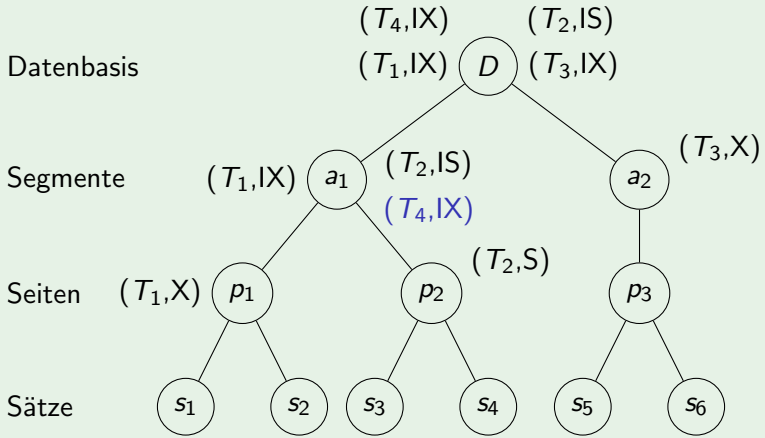
MGL blockiert Transaktionen

Beispiel (MGL blockiert Transaktionen)



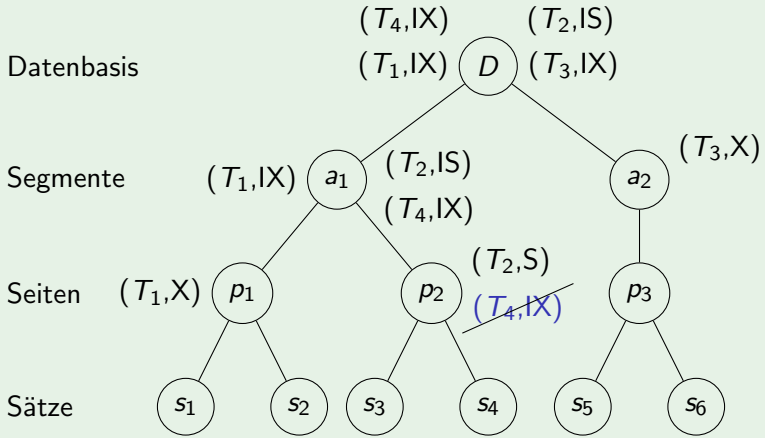
MGL blockiert Transaktionen

Beispiel (MGL blockiert Transaktionen)



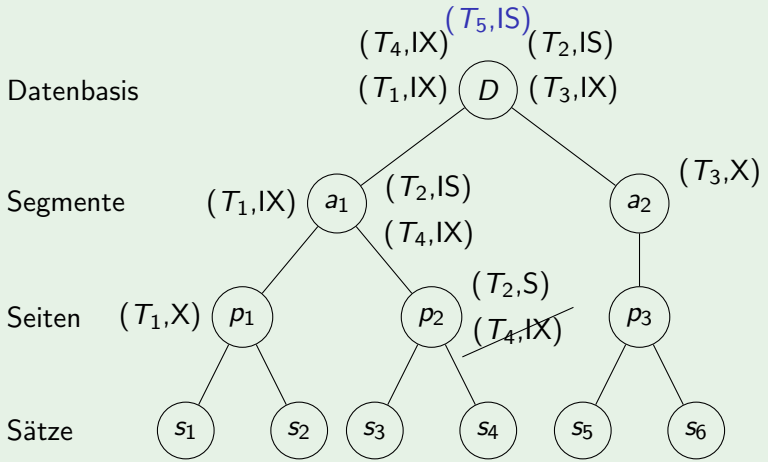
MGL blockiert Transaktionen

Beispiel (MGL blockiert Transaktionen)



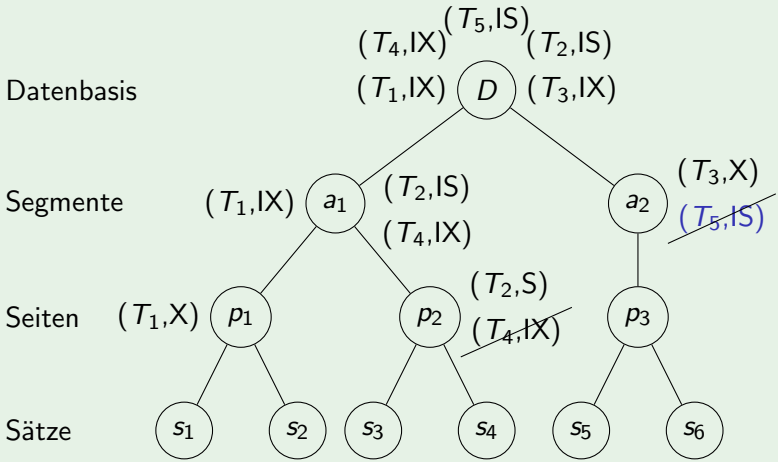
MGL blockiert Transaktionen

Beispiel (MGL blockiert Transaktionen)



MGL blockiert Transaktionen

Beispiel (MGL blockiert Transaktionen)



Inhalt

Insert/Delete – Operationen

Bisher: Annahme dass keine Tupel eingefügt/gelöscht werden.

Insert/Delete – Operationen

Bisher: Annahme dass keine Tupel eingefügt/gelöscht werden.

Besonderheiten bei Einfüge/Löschoperationen:

- Sperren beim Einfügen/Löschen
- Phantomproblem

Sperren beim Einfügen/Löschen

Löschen:

- Erfordert eine X-Sperre auf das zu löschende Objekt.
- Andere Transaktionen die eine Sperre für das Objekt wollen können diese nach erfolgreichem Abschluss (`commit`) nicht mehr erhalten.

Sperren beim Einfügen/Löschen

Löschen:

- Erfordert eine X-Sperre auf das zu löschende Objekt.
- Andere Transaktionen die eine Sperre für das Objekt wollen können diese nach erfolgreichem Abschluss (`commit`) nicht mehr erhalten.

Einfügen:

- Durch das Einfügen erhält Transaktion eine X-Sperre.
- Vermeidung des Phantomproblems erfordert zusätzliche Maßnahmen.

Phantomproblem

Beispiel (Auftreten des Phantomproblems)

T_1	T_2
`SELECT COUNT(*)`	
`FROM pruefen`	
`WHERE Note BETWEEN 1 AND 2`	
	`INSERT INTO pruefen`
	`VALUES`
	`(29555,5001,2137,1)`
`SELECT COUNT(*)`	
`FROM pruefen`	
`WHERE Note BETWEEN 1 AND 2`	

Phantomproblem

Problem:

Phantomproblem

Problem:

- "*Falls keine Transaktion abbricht ist jede konfliktserialisierbare Historie auch serialisierbar*" gilt nur wenn keine neuen Einträge eingefügt werden.

Phantomproblem

Problem:

- "*Falls keine Transaktion abbricht ist jede konfliktserialisierbare Historie auch serialisierbar*" gilt nur wenn keine neuen Einträge eingefügt werden.
- (Strenges) 2PL sperrt in dem Fall zu wenig Datensätze:
  - Transaktion sperrt nur vorhandene Datensätze
  - Korrektheit von 2PL erfordert Sperrung aller benötigten Datensätze

Phantomproblem

Problem:

- "*Falls keine Transaktion abbricht ist jede konfliktserialisierbare Historie auch serialisierbar*" gilt nur wenn keine neuen Einträge eingefügt werden.
- (Strenges) 2PL sperrt in dem Fall zu wenig Datensätze:
  - Transaktion sperrt nur vorhandene Datensätze
  - Korrektheit von 2PL erfordert Sperrung aller benötigten Datensätze

Lösung: Das Anlegen neuer, relevanter Datensätze muss verhindert werden

Phantomproblem

Problem:

- "*Falls keine Transaktion abbricht ist jede konfliktserialisierbare Historie auch serialisierbar*" gilt nur wenn keine neuen Einträge eingefügt werden.
- (Strenges) 2PL sperrt in dem Fall zu wenig Datensätze:
  - Transaktion sperrt nur vorhandene Datensätze
  - Korrektheit von 2PL erfordert Sperrung aller benötigten Datensätze

Lösung: Das Anlegen neuer, relevanter Datensätze muss verhindert werden

- Tabelle ohne Index: Anlegen neuer Seiten/Datensätze verhindern (z.B. IS-Sperre auf Tabelle)
- Tabelle mit Index: Zusätzlich muss der betroffene Indexbereich gesperrt werden

Inhalt

Weitere Synchronisationsmethoden

Weitere Synchronisationsmethoden

- Zeitstempel-basierende Synchronisation

Weitere Synchronisationsmethoden

- Zeitstempel-basierende Synchronisation
- Optimistische Synchronsiation

Zeitstempel-basierende Synchronisation

- Jede Transaktion T erhält eindeutigen Zeitstempel $TS(T)$
  - $TS(T_i) < TS(T_j)$ wenn T_i älter als T_j ist

Zeitstempel-basierende Synchronisation

- Jede Transaktion T erhält eindeutigen Zeitstempel $TS(T)$
  - $TS(T_i) < TS(T_j)$ wenn T_i älter als T_j ist
- Jedem Datum A werden zwei Werte zugewiesen:

Zeitstempel-basierende Synchronisation

- Jede Transaktion T erhält eindeutigen Zeitstempel $TS(T)$
  - $TS(T_i) < TS(T_j)$ wenn T_i älter als T_j ist
- Jedem Datum A werden zwei Werte zugewiesen:
  - `readTS(`A`)`: TS der jüngsten Transaktion die A gelesen hat

Zeitstempel-basierende Synchronisation

- Jede Transaktion T erhält eindeutigen Zeitstempel $TS(T)$
  - $TS(T_i) < TS(T_j)$ wenn T_i älter als T_j ist
- Jedem Datum A werden zwei Werte zugewiesen:
  - `readTS(`A`)`: TS der jüngsten Transaktion die A gelesen hat
  - `writeTS(`A`)`: TS der jüngsten Transaktion die A geschrieben hat

Zeitstempel-basierende Synchronisation

- Jede Transaktion T erhält eindeutigen Zeitstempel $TS(T)$
  - $TS(T_i) < TS(T_j)$ wenn T_i älter als T_j ist
- Jedem Datum A werden zwei Werte zugewiesen:
  - `readTS(`A`)`: TS der jüngsten Transaktion die A gelesen hat
  - `writeTS(`A`)`: TS der jüngsten Transaktion die A geschrieben hat

Idee:

- Erzeuge serielle Historie in der Reihenfolge der Zeitstempel
  - Vor Zugriff von T_i auf A wird $TS(T_i)$ mit `readTS(`A`)` bzw. `writeTS(`A`)` verglichen und bei einem Konflikt T_i zurückgesetzt.

Lesezugriff bei Zeitstempel-basierender Synchronisation

Lesezugriff $r_i(A)$ (T_i will A lesen):

Lesezugriff bei Zeitstempel-basierender Synchronisation

Lesezugriff $r_i(A)$ (T_i will A lesen):

$TS(T_i) < \texttt{writeTS}(A)$: T_i ist älter als eine andere Transaktion die A schon geschrieben hat.

Lesezugriff bei Zeitstempel-basierender Synchronisation

Lesezugriff $r_i(A)$ (T_i will A lesen):

$TS(T_i) < \texttt{writeTS}(A)$: T_i ist älter als eine andere Transaktion die A schon geschrieben hat.

- T_i wird zurückgesetzt

Lesezugriff bei Zeitstempel-basierender Synchronisation

Lesezugriff $r_i(A)$ (T_i will A lesen):

$TS(T_i) < \texttt{writeTS}(A)$: T_i ist älter als eine andere Transaktion die A schon geschrieben hat.

- T_i wird zurückgesetzt

$TS(T_i) \geq \texttt{writeTS}(A)$: T_i ist jünger als eine andere Transaktion die A schon geschrieben hat.

Lesezugriff bei Zeitstempel-basierender Synchronisation

Lesezugriff $r_i(A)$ (T_i will A lesen):

$TS(T_i) < \texttt{writeTS}(A)$: T_i ist älter als eine andere Transaktion die A schon geschrieben hat.

- T_i wird zurückgesetzt

$TS(T_i) \geq \texttt{writeTS}(A)$: T_i ist jünger als eine andere Transaktion die A schon geschrieben hat.

- T_i darf A lesen

Lesezugriff bei Zeitstempel-basierender Synchronisation

Lesezugriff $r_i(A)$ (T_i will A lesen):

$TS(T_i) < \texttt{writeTS}(A)$: T_i ist älter als eine andere Transaktion die A schon geschrieben hat.

- T_i wird zurückgesetzt

$TS(T_i) \geq \texttt{writeTS}(A)$: T_i ist jünger als eine andere Transaktion die A schon geschrieben hat.

- T_i darf A lesen
- $\texttt{readTS}(A) = \max(TS(T_i), \texttt{readTS}(A))$

Schreibzugriff bei Zeitstempel-basierender Synchronisation

Schreibzugriff $w_i(A)$ (T_i will auf A schreiben):

Schreibzugriff bei Zeitstempel-basierender Synchronisation

Schreibzugriff $w_i(A)$ (T_i will auf A schreiben):

$TS(T_i) < \texttt{readTS}(A)$: Eine jüngere Transaktion hat bereits A gelesen, hätte aber den Wert von A lesen müssen den T_i schreiben will.

Schreibzugriff bei Zeitstempel-basierender Synchronisation

Schreibzugriff $w_i(A)$ (T_i will auf A schreiben):

$TS(T_i) < \texttt{readTS}(A)$: Eine jüngere Transaktion hat bereits A gelesen, hätte aber den Wert von A lesen müssen den T_i schreiben will.

- T_i wird zurückgesetzt

Schreibzugriff bei Zeitstempel-basierender Synchronisation

Schreibzugriff $w_i(A)$ (T_i will auf A schreiben):

$TS(T_i) < \texttt{readTS}(A)$: Eine jüngere Transaktion hat bereits A gelesen, hätte aber den Wert von A lesen müssen den T_i schreiben will.

- T_i wird zurückgesetzt

$TS(T_i) < \texttt{writeTS}(A)$: Eine jüngere Transaktion hat auf A geschrieben und wird jetzt von T_i überschrieben.

Schreibzugriff bei Zeitstempel-basierender Synchronisation

Schreibzugriff $w_i(A)$ (T_i will auf A schreiben):

$TS(T_i) < \texttt{readTS}(A)$: Eine jüngere Transaktion hat bereits A gelesen, hätte aber den Wert von A lesen müssen den T_i schreiben will.

- T_i wird zurückgesetzt

$TS(T_i) < \texttt{writeTS}(A)$: Eine jüngere Transaktion hat auf A geschrieben und wird jetzt von T_i überschrieben.

- T_i wird zurückgesetzt

Schreibzugriff bei Zeitstempel-basierender Synchronisation

Schreibzugriff $w_i(A)$ (T_i will auf A schreiben):

$TS(T_i) < \texttt{readTS}(A)$: Eine jüngere Transaktion hat bereits A gelesen, hätte aber den Wert von A lesen müssen den T_i schreiben will.

- T_i wird zurückgesetzt

$TS(T_i) < \texttt{writeTS}(A)$: Eine jüngere Transaktion hat auf A geschrieben und wird jetzt von T_i überschrieben.

- T_i wird zurückgesetzt

Sonst:

- T_i darf auf A schreiben
- $\texttt{writeTS}(A) = TS(T_i)$

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4			

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$		

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5			

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$		

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6			

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7			

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7	$\texttt{reset}_1\ [r_1(A)]$	3	2

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7	$\texttt{reset}_1\ [r_1(A)]$	3	2
8	$w_3(A)$	3	3

Zeitstempel-basierende Synchronisation: Beispiel

Beispiel

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$	$\texttt{writeTS}(A)$
1	$\texttt{BOT}_1$	0	0
2	$\texttt{BOT}_2$	0	0
3	$\texttt{BOT}_3$	0	0
4	$r_1(A)$	1	0
5	$w_2(A)$	1	2
6	$r_3(A)$	3	2
7	$\texttt{reset}_1\ [r_1(A)]$	3	2
8	$w_3(A)$	3	3
9	$\texttt{reset}_2\ [w_2(A)]$	3	3

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.
- **Aber:** Historien können nicht rücksetzbar sein.

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.
- **Aber:** Historien können nicht rücksetzbar sein.

Lösung:

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.
- **Aber:** Historien können nicht rücksetzbar sein.

Lösung:

Rücksetzbar: Verzögere `commit` bis alle Transaktionen von denen gelesen wurde beendet sind (verwalte Liste).

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.
- **Aber:** Historien können nicht rücksetzbar sein.

Lösung:

Rücksetzbar: Verzögere `commit` bis alle Transaktionen von denen gelesen wurde beendet sind (verwalte Liste).

Strikt: Führe alle Schreibvorgänge atomar am Ende der Transaktion durch.

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.
- **Aber:** Historien können nicht rücksetzbar sein.

Lösung:

Rücksetzbar: Verzögere `commit` bis alle Transaktionen von denen gelesen wurde beendet sind (verwalte Liste).

Strikt: Führe alle Schreibvorgänge atomar am Ende der Transaktion durch.

Alternativ: Markiere noch nicht festgeschriebene Daten und verzögere Zugriff darauf.

Eigenschaften Zeitstempel-basierender Synchronisation

- Es entstehen keine Deadlocks.
- Liefert konfliktserialisierbare Historien.
- Reihenfolge in serieller Historie entspricht Zeitstempeln.
- **Aber:** Historien können nicht rücksetzbar sein.

Lösung:

Rücksetzbar: Verzögere `commit` bis alle Transaktionen von denen gelesen wurde beendet sind (verwalte Liste).

Strikt: Führe alle Schreibvorgänge atomar am Ende der Transaktion durch.

Alternativ: Markiere noch nicht festgeschriebene Daten und verzögere Zugriff darauf.

Strikte Historien durch Zeitstempel

Vorgehen:

- *"dirty bit":* Vermerke falls ein Datensatz von einer noch aktiven Transaktion geschrieben wurde.
- Falls dirty bit gesetzt werden andere Transaktionen die auf das Datum zugreifen wollen verzögert.
  - Nur Lesezugriffe: Vermeidet kaskadierendes Rücksetzen
  - Schreib- und Lesezugriffe: Garantiert strikte Historien

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6				

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	readTS(A):	writeTS(A)	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$r_3(A)$			

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗
7				

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗
7	$\texttt{reset}_1\ [r_1(A)]$	1	2	✗

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗
7	$\texttt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗
7	$\texttt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗
9	c_2	1	2	

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗
7	$\texttt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗
9	c_2	1	2	
10	$r_3(A)$	3	2	

Strikte Historie durch Zeitstempel und Dirty Bit

$\texttt{BOT}_1 \rightarrow \texttt{BOT}_2 \rightarrow \texttt{BOT}_3 \rightarrow r_1(A) \rightarrow w_2(A) \rightarrow r_3(A) \rightarrow r_1(A) \rightarrow w_3(A) \rightarrow w_2(A) \rightarrow c_2$

TS	Aktion	$\texttt{readTS}(A)$:	$\texttt{writeTS}(A)$	$d(A)$
1	$\texttt{BOT}_1$	0	0	
2	$\texttt{BOT}_2$	0	0	
3	$\texttt{BOT}_3$	0	0	
4	$r_1(A)$	1	0	
5	$w_2(A)$	1	2	✗
6	$\texttt{block}_3\ [r_3(A)]$	1	2	✗
7	$\texttt{reset}_1\ [r_1(A)]$	1	2	✗
8	$w_2(A)$	1	2	✗
9	c_2	1	2	
10	$r_3(A)$	3	2	
11	$w_3(A)$	3	3	✗

Strikte Historien durch Zeitstempel

`Commit` von T_i:

- Aktualisiere die dirty bits

Strikte Historien durch Zeitstempel

`Commit` von T_i:

- Aktualisiere die dirty bits

`Rollback` von T_i:

- Für jeden von der Transaktion geschriebenen Datensat A:
  Setze `writeTS`(A) zurück auf den Wert vor dem Schreibzugriff von T_i
- Aktualisiere die dirty bits
- *Anmerkung:* Die letzte vor T_i geschriebene Transaktion hat erfolgreich abgeschlossen

Optimistische Synchronisation

Generelle Idee:

- Führe Transaktionen erst einmal aus
- Beobachte Aktionen und entscheide auf Grund dessen ob Konflikte aufgetreten sind
- Bei Problemen: Setze Transaktion zurück

Optimistische Synchronisation

Generelle Idee:

- Führe Transaktionen erst einmal aus
- Beobachte Aktionen und entscheide auf Grund dessen ob Konflikte aufgetreten sind
- Bei Problemen: Setze Transaktion zurück

Im Folgenden: Eine konkrete Methode

Optimistische Synchronisation

Transaktion wird in 3 Phasen aufgeteilt:

1. Lesephase
   - Alle Operationen der Transaktion werden ausgeführt
   - Schreiboperationen aber nur auf lokalen Kopien
2. Validierungsphase
   - Überprüfung ob die Transaktion committed werden darf
   - Konflikte werden mittels Zeitstempels (vergeben nach Reihenfolge des Eintritts in die Validierungsphase) erkannt
3. Schreibphase
   - Nach erfolgreicher Validierung werden Änderungen in die Datenbasis eingebracht

Validierungsphase bei Optimistischer Synchronisation

Validierung von T_i:
Für alle Transaktionen T_a mit $TS(T_a) < TS(T_i)$ (Achtung: TS wird beim Erreichen der Validierungsphase vergeben!) muss eine von zwei Bedingungen erfüllt sein:

1. T_a war zum Beginn von T_i schon vollständig abgeschlossen.
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$

Validierungsphase bei Optimistischer Synchronisation

Validierung von T_i:
Für alle Transaktionen T_a mit $TS(T_a) < TS(T_i)$ (Achtung: TS wird beim Erreichen der Validierungsphase vergeben!) muss eine von zwei Bedingungen erfüllt sein:

1. T_a war zum Beginn von T_i schon vollständig abgeschlossen.
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
   - $WriteSet(T_a)$ alle von T_a geschriebenen Datenobjekte

Validierungsphase bei Optimistischer Synchronisation

Validierung von T_i:
Für alle Transaktionen T_a mit $TS(T_a) < TS(T_i)$ (Achtung: TS wird beim Erreichen der Validierungsphase vergeben!) muss eine von zwei Bedingungen erfüllt sein:

1. T_a war zum Beginn von T_i schon vollständig abgeschlossen.
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
   - $WriteSet(T_a)$ alle von T_a geschriebenen Datenobjekte
   - $ReadSet(T_i)$ alle von T_i gelesenen Datenobjekte

Validierungsphase bei Optimistischer Synchronisation

Validierung von T_i:
Für alle Transaktionen T_a mit $TS(T_a) < TS(T_i)$ (Achtung: TS wird beim Erreichen der Validierungsphase vergeben!) muss eine von zwei Bedingungen erfüllt sein:

1. T_a war zum Beginn von T_i schon vollständig abgeschlossen.
2. $WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$
   - $WriteSet(T_a)$ alle von T_a geschriebenen Datenobjekte
   - $ReadSet(T_i)$ alle von T_i gelesenen Datenobjekte

Wichtig für Korrektheit:
Validierungs- und Schreibphase müssen atomar ablaufen
(Nur jeweils eine Transaktion darf in Phase 2+3 sein)

Inhalt

Transaktionsverwaltung in SQL

```
SET TRANSACTION
  [READ WRITE | READ ONLY]
  [ISOLATION LEVEL { READ UNCOMMITED |
                     READ COMMITED |
                     REPEATABLE READ |
                     SERIALIZABLE }]
```

Transaktionsverwaltung in SQL

```
SET TRANSACTION
  [READ WRITE | READ ONLY]
  [ISOLATION LEVEL { READ UNCOMMITED |
                     READ COMMITED |
                     REPEATABLE READ |
                     SERIALIZABLE }]
```

Access Mode:

- read only: Nur Lesezugriff ($\Rightarrow$ nur Lesesperren)
- read write: Default

Transaktionsverwaltung in SQL

```
SET TRANSACTION
  [READ WRITE | READ ONLY]
  [ISOLATION LEVEL { READ UNCOMMITED |
                     READ COMMITED |
                     REPEATABLE READ |
                     SERIALIZABLE }]
```

Access Mode:

- read only: Nur Lesezugriff ($\Rightarrow$ nur Lesesperren)
- read write: Default

Isolation Level:

- Bieten unterschiedliche Grade der Isolation, erlauben Parallelisierbarkeit zu erhöhen.

Isolation Levels in SQL

`READ UNCOMMITED` Schwächste Stufe. Kann nicht festgeschriebene Änderungen lesen und daher auch inkonsistente Datenbankzustände sehen. Nur für `READ ONLY` Transaktionen erlaubt.

Isolation Levels in SQL

`READ UNCOMMITED` Schwächste Stufe. Kann nicht festgeschriebene Änderungen lesen und daher auch inkonsistente Datenbankzustände sehen. Nur für `READ ONLY` Transaktionen erlaubt.

`READ COMMITED` Jede Operation sieht nur Datensätze die vor dem Beginn der Operation committed waren.

Isolation Levels in SQL

`READ UNCOMMITED` Schwächste Stufe. Kann nicht festgeschriebene Änderungen lesen und daher auch inkonsistente Datenbankzustände sehen. Nur für `READ ONLY` Transaktionen erlaubt.

`READ COMMITED` Jede Operation sieht nur Datensätze die vor dem Beginn der Operation committed waren.

`REPEATABLE READ` Alle Operationen der Transaktion sehen nur Datensätze die vor der ersten Aktion der Transaktion committed waren.

Isolation Levels in SQL

`READ UNCOMMITED` Schwächste Stufe. Kann nicht festgeschriebene Änderungen lesen und daher auch inkonsistente Datenbankzustände sehen. Nur für `READ ONLY` Transaktionen erlaubt.

`READ COMMITED` Jede Operation sieht nur Datensätze die vor dem Beginn der Operation committed waren.

`REPEATABLE READ` Alle Operationen der Transaktion sehen nur Datensätze die vor der ersten Aktion der Transaktion committed waren.

`SERIALIZABLE` Höchste Stufe.

Isolation Levels und mögliche Anomalien

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
`READ UNCOMMITTED`	möglich	möglich	möglich
`READ COMMITTED`	–	möglich	möglich
`REPEATABLE READ`	–	–	möglich
`SERIALIZABLE`	–	–	–

Zusammenfassung

Protokolle und ihre wichtigsten Eigenschaften

Protokoll	Äquivalente serielle Historie	Weitere Eigenschaften	Deadlock möglich
2PL	Reihenfolge der Sperranforderungen bei Konflikten	im allgemeinen nicht rücksetzbar	ja
Strict 2PL	wie 2PL	strikt	ja
Strict 2PL + Deadlock-Vermeidung	wie 2PL	strikt	nein
Zeitstempel-basierend	Zeitpunkt von BOT	strikte Variante existiert	nein
optimistisch	Zeitpunkt der Validierung	strikt	nein