



Informations- und Wissensmanagement

Kapitel 6: **Nebenläufigkeit und Transaktionen**



Gliederung

- Einleitung/Probleme,
- Definitionen
(Transaktion, History, Konflikt, Äquivalenz,
Serialisierbarkeit, ...),
- Locking.

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)



Transaktionseigenschaften (1)

- **ACID-Eigenschaften:**

- **Atomicity**

- **Consistency**

- **Isolation**

- **Durability**

Constraints

Logging,
Recovery

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)



Transaktionseigenschaften (2)

- **Atomarität**

- Beispiel Bank:
- | Name | Konto | Guthaben |
|----------|-------|----------|
| Erik B. | 1892 | 1000 |
| Mirco S. | 2434 | 2000 |
- Überweisung besteht technisch aus zwei Elementaroperationen.
 - Abbuchung(Erik B., 500),
 - Einzahlung(Mirco S., 500).

- **Isolation**

- was passiert, wenn parallele Transaktionen
 - die gleichen Datensätze schreiben?
 - Datensätze von nicht abgeschlossenen Transaktionen lesen?

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

```
SQL> UPDATE Bankkonten SET Stand=Stand-200 WHERE Kontonr=1234;
```

```
1 row updated.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	500

```
SQL> UPDATE Bankkonten SET Stand=Stand+200 WHERE Kontonr=5678;
```

```
1 row updated.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	700

```
SQL> █
```

Benutzer 1

1

```
SQL> SET AUTOCOMMIT OFF;  
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

2

```
SQL> UPDATE Bankkonten SET Stand=Stand-200 WHERE Kontonr=1234;  
1 row updated.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	500

3

```
SQL> UPDATE Bankkonten SET Stand=Stand+200 WHERE Kontonr=5678;  
1 row updated.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	700

```
SQL> COMMIT;
```

4

```
Commit complete.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	700

```
SQL> █
```

Benutzer 2

1

```
SQL> SET AUTOCOMMIT OFF;  
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

2

```
SQL>  
SQL>  
SQL>  
SQL>  
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

3

```
SQL>  
SQL>  
SQL>  
SQL>  
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

4

```
SQL>  
SQL>  
SQL>  
SQL>  
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	700

```
SQL> █
```

```
X Desktop
SQL> SET AUTOCOMMIT OFF;
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

```
SQL> UPDATE Bankkonten SET Stand=Stand-200 WHERE Kontonr=1234;
1 row updated.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	500

```
SQL> UPDATE Bankkonten SET Stand=Stand+200 WHERE Kontonr=5678;
1 row updated.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	300
5678	Gunter Saake	700

```
SQL> ROLLBACK;
```

```
Rollback complete.
```

```
SQL> SELECT * FROM Bankkonten;
```

KONTONR	INHABER	STAND
1234	Klemens Boehm	500
5678	Gunter Saake	500

```
SQL> █
```

Rollback
als Gegensatz
zum **Commit**.



Synchronisation in Datenbanken

- Zentrales Leistungsmerkmal von Datenbanken: Viele Benutzer können die gleichen Daten gleichzeitig sowohl lesend als auch schreibend zugreifen.
- Konsistenz muß sichergestellt sein – Aufgabe der **Synchronisationskomponente**.
- Benutzer sollen vom Multi-User Betrieb so wenig wie möglich merken.
Nebenläufigkeit soll transparent sein.
,Illusion', daß man der einzige Nutzer ist.
- Engl. *concurrency, concurrency control*.

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)



Synchronisationsprobleme

- Unkontrollierte **nicht-serielle Ausführung** führt zu Inkonsistenz:
 - **Lost Updates**,
 - **Non-repeatable reads**, inkonsistente Lesezugriffe, unterschiedliche Ergebnisse bei identischer Anfrage
 - **Dirty Reads**, Lesen von Updates, die noch nicht committet sind.
 - **Phantome**, Lesen von neu angelegten Datentupeln aus einer Transaktion, die noch nicht committet hat

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)

Lost Update

- Programm T_1 transferiert EUR 300,- von A auf B, Programm T_2 schreibt Konto A 3 % Zinsen gut.
- Zinsen aus Schritt 5 von Programm T_2 gehen verloren, weil T_1 den Wert Schritt 6 überschreibt.

Schritt	T_1	T_2
1	Read(A, a1)	
2	a1 := a1-300	
3		Read(A, a2)
4		a2 := a2 *1.03
5		Write(A, a2)
6	Write(A, a1)	
7	Read(B, b1)	
8	b1 := b1 + 300	
9	Write(B, b1)	

Dirty Read

- *Commit, Abort.*
- Programm T_2 berechnet Zinsen auf einem Wert aus einem inkonsistenten Zustand

Schritt	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 * 1.03
6		Write(A, a2)
7		commit
8	Read(B, b1)	
9	...	
10	abort	

Non-Repeatable Reads

- Programm liest Datenobjekt mehr als einmal und sieht Änderung, die eine andere Transaktion durchgeführt hat.

Schritt	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 *1.03
6		Write(A, a2)
7	Read(A, a3)	
8	...	



Phantom Reads

- während einer Transaktion wiederholte Anfragen ergeben unterschiedliche Ergebnismengen
- entspricht Non-repeatable Reads auf Mengen statt Werten

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)

T1	T2
select count(*) from Bank	insert into Bank (Name, Konto, Wert) values ('Erik B', 29384, 100.00)
select count(*) from Bank	
...	
...	...





Lösbar ohne Synchronisation, aber...

- **Serielle Ausführung** von Transaktionen
 - neue Transaktion kann erst starten, wenn vorhergehende alle Daten geschrieben hat
 - Datenbank ist am Ende jeder Transaktion konsistent.
 - kein Aufwand durch Sperren und Scheduling
 - **Aber:** Extreme Wartezeiten und unbefriedigende Ausnutzung der Ressourcen.
 - während Festplatte arbeitet, steht der Rest des Rechners ungenutzt still

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)





Maßnahmen zur Synchronisation

- Zusammenfassen von mehreren Zugriffen zu **Transaktionen**
 - commit/rollback bestätigt/verwirft **alle** Änderungen der Transaktion
- Sperren von einzelnen Lese / Schreibzugriffen
- Umsortieren der Reihenfolge von parallelen Transaktionen
- Abbrechen von Transaktionen, die mit anderen in Konflikt stehen
 - DBMS kann Transaktion selbst zurücksetzen

Problem: Tradeoff zwischen Leistung der DB und Garantien zur Isolation!

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)

Nebenläufigkeit in existierenden DBMS

- Beispiel: Isolation Levels in Oracle
Read committed ist die Voreinstellung

Anm.: Lost Updates immer ausgeschlossen

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)



Transaktionen und Histories



Transaktionen

- *Transaktion* := Ausführung eines Programms, das auf die Datenbank (lesend oder schreibend) zugreift.
- Programmausführung \neq Programm-Code.
- Genauer: Repräsentation der Ausführung, die folgende Charakteristika umfaßt:
 - Gelesene und geschriebene Datenobjekte,
 - Reihenfolge ihrer Ausführung,
 - kommt am Ende ein Commit (bzw. Abort) oder nicht?

Einleitung/
Probleme

Definitionen

Locking



Zielstellung dieses Kapitels

- Wie muss ein Ablaufplan (***Schedule***) für mehrere Transaktionen aussehen, der
 - das gleiche Resultat zeigt wie eine serielle Ausführung, aber
 - die Ressourcen des Rechners durch parallele Ausführung besser ausnutzt?

Einleitung/
Probleme

Definitionen

Locking



Beispiel für Transaktionen

- Beispiel:

```
Procedure  $P$  begin
```

```
  Start;
```

```
  temp := Read(x);
```

```
  temp := temp + 1;
```

```
  Write(x, temp);
```

```
  Commit
```

```
end
```

Operationen



Einleitung/
Probleme

[Definitionen](#)

[Locking](#)

- Repräsentation: $r_1[x] \rightarrow w_1[x] \rightarrow c_1$
- Transaktion ist partielle Ordnung $(\Sigma, <)$
(Σ wird im Folgenden meistens weggelassen.)



Konflikte

- **Zwei Operationen p, q konfliktieren** :=
p, q greifen auf das gleiche Datenobjekt zu,
und p oder q ist eine Schreiboperation.
- Weitere Operationen –
Definition von ‘Konflikt’ muß erweitert werden.
- Beispiel. Kompatibilitätsmatrix:

	Read	Write	Increment	Decrement
Read	y	n	n	n
Write	n	n	n	n
Increment	n	n	y	y
Decrement	n	n	y	y

Einleitung/
Probleme

[Definitionen](#)

[Locking](#)



Transaktion – formale Definition

Transaktion ist partielle Ordnung
mit Ordnungsrelation $<$, so daß gilt

1. $T_i \subseteq \{r_i[x], w_i[x] | x \text{ ist ein Datenobjekt}\} \cup \{a_i, c_i\}$,
2. $a_i \in T_i \Leftrightarrow c_i \notin T_i$;
3. wenn es sich bei t um c_i oder a_i handelt, dann gilt für jede andere Operation $p \in T_i$: $p <_i t$; und
4. wenn $r_i[x], w_i[x] \in T_i$,
dann $r_i[x] <_i w_i[x]$ oder $w_i[x] <_i r_i[x]$.

Ordnungsbeziehung
ist essentiell um
Konflikte zu
bestimmen!

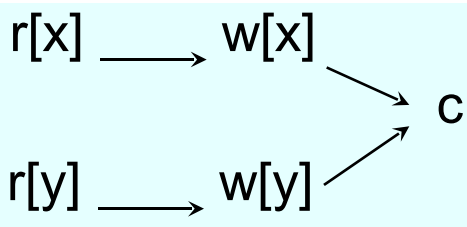
Transaktion – weitere Beispiele

- Gegeben: Halbordnung von Operationen

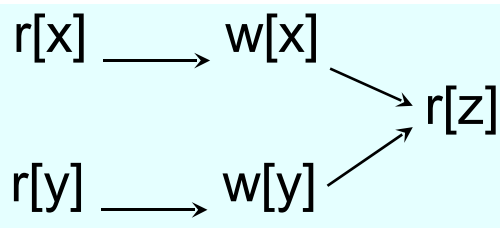
Einleitung/
Probleme

Definitionen

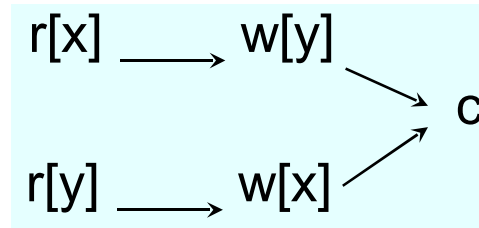
Locking



ist Transaktion.



ist keine TA.



ist keine TA.



Beziehungen zwischen Transaktionen

- Im DBMS viele parallele Transaktionen möglich
- **Reads-from-Beziehung:**
Transaktion T_i liest von Transaktion T_j wenn
 1. T_i liest x , nachdem T_j x geschrieben hat;
 2. T_j abortet nicht, bevor T_i x liest; und
 3. Jede Transaktion, die x schreibt, bevor T_i x liest, und nachdem T_j x überschreibt, abortet, bevor T_i x liest.

Einleitung/
Probleme

[Definitionen](#)

Locking



Beispiele für Beziehungen

- $w_1[x] w_2[x] r_3[x] r_4[x] c_1 c_2 c_3 c_4$
reads-from Beziehungen:
 T_3 von T_2 , T_4 von T_2 .
- $w_1[x] w_2[x] r_3[x] r_4[x] c_1 a_2 c_3 c_4$
reads-from Beziehungen:
 T_3 von T_2 , T_4 von T_2 .
- $w_1[x] w_2[x] a_2 r_3[x] r_4[x] c_1 c_3 c_4$
reads-from Beziehungen:
 T_3 von T_1 , T_4 von T_1 .

Einleitung/
Probleme

Definitionen

Locking





Histories

- Ausführung der Operationen mehrerer Transaktionen, die miteinander ‘verzahnt’ sind, d.h. nebenläufig ablaufen.
- Formal: $H = \{T_1, T_2, \dots, T_n\}$ ist eine Menge von Transaktionen.
- Anm.: in der Literatur werden *Histories* auch häufig als *Schedules* bezeichnet

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)





Vollständige Histories

- **Vollständige History** H über $T :=$
partielle Ordnung mit Ordnungsbeziehung $<_H$,
so dass
 1. $H = \bigcup_{i=1}^n T_i$
 2. $<_H \supseteq \bigcup_{i=1}^n <_i$
 3. $p, q \in H$ konfliktieren $\rightarrow p <_H q$ oder $q <_H p$

Einleitung/
Probleme

Definitionen

Locking



Probleme mit vollst. Histories

- vollständige Histories erst nach Abschluss aller Transaktionen definiert
 - *für Scheduling nutzlos*
- praktische Herangehensweise im folgenden:
 - **History**: Präfix einer vollständigen History.



„Arbeitsversion“, um eine vollständige History zu bekommen

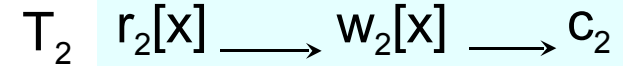
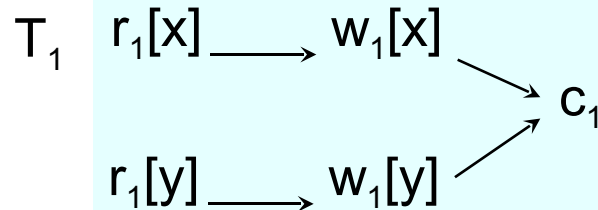
Einleitung/
Probleme

Definitionen

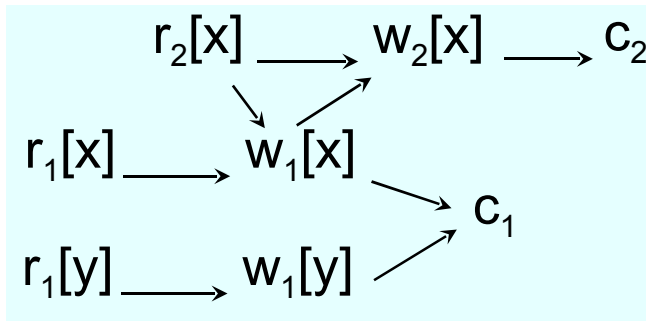
Locking

Histories – Beispiele (1)

- Gegeben zwei Transaktionen:



- Vollständige History:



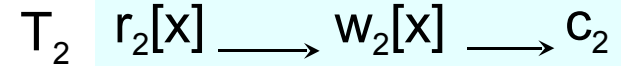
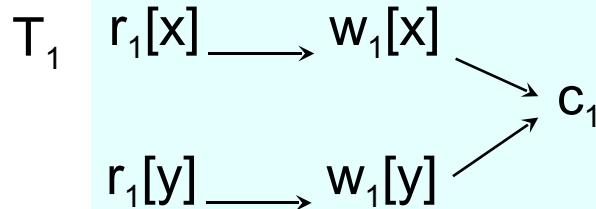
Einleitung/
Probleme

[Definitionen](#)

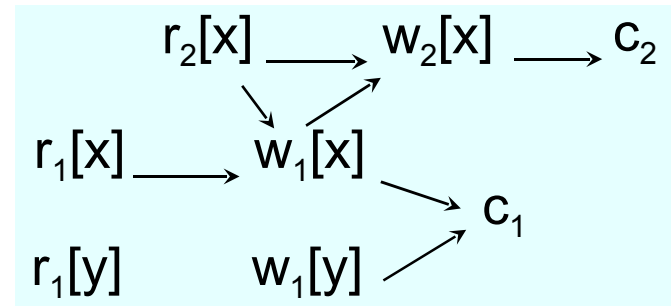
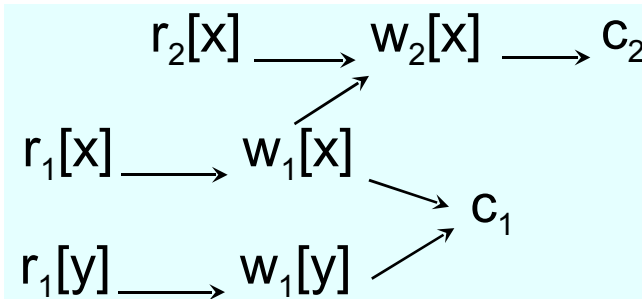
Locking

Histories – Beispiele (2)

- Gegeben zwei Transaktionen:

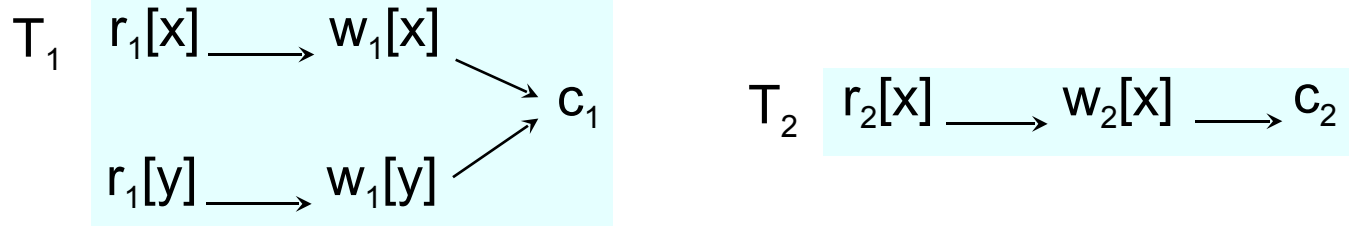


- Keine Histories:

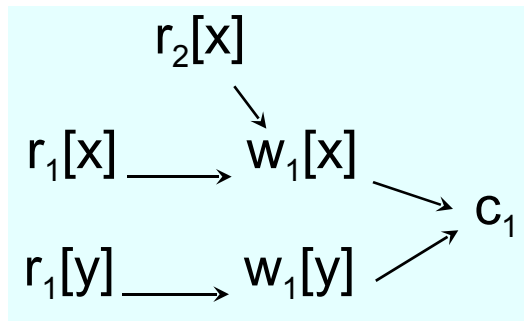


Histories – Beispiele (3)

- Gegeben zwei Transaktionen:



- History, aber nicht *vollständig*:





Äquivalenz von Histories

- mehrere Definitionen möglich:
 - (Konflikt-)Äquivalenz,
 - (Sicht-)Äquivalenz.
- im Folgenden nur Konflikt-Äquivalenz betrachtet
- Definition **Konflikt-Äquivalenz**:
Histories H, H' sind Konfliktäquivalent, wenn
 1. gleiche Transaktionen, gleiche Operationen;
 2. gleiche Ordnung konfligierender Operationen.
z.B. gehören p_i und q_j zu T_i bzw T_j .
 $a_i, a_j \notin H$. Wenn $p_i <_H q_j$, dann $p_i <_{H'} q_j$.

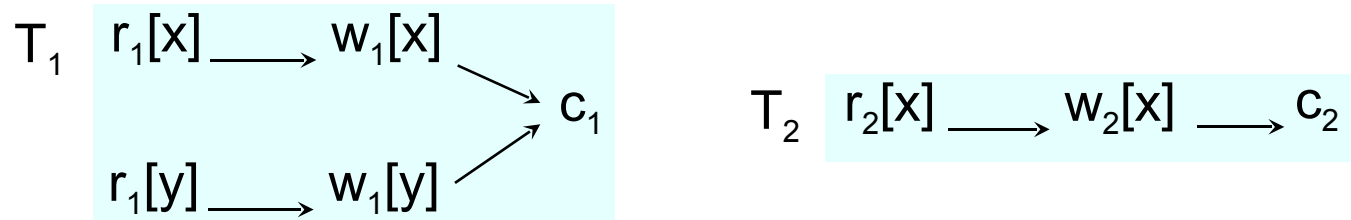
Einleitung/
Probleme

Definitionen

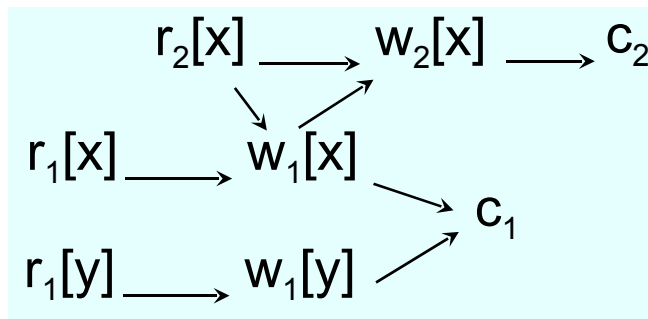
Locking

Äquivalenz von Histories – Beispiele (1)

- Gegeben zwei Transaktionen:



- Eine vollständige History, OK:



- Beispiel für eine Konflikt-äquivalente History, die nicht identisch ist?

Äquivalenz von Histories – Beispiele (2)

- History 1:

Schritt	T1	T2	T3
1	Read(A)		
2		Write(A)	
3	Write(A)		
4			Write(A)

Alle
Transaktionen
committen

- History 2:

Schritt	T1	T2	T3
1	Read(A)		
2	Write(A)		
3		Write(A)	
4			Write(A)

- Sind diese Histories Konflikt-äquivalent?

Einleitung/
Probleme

Definitionen

Locking



Überlegungen zur Korrektheit

- History muß nicht korrekt sein, kann z. B. Lost Updates etc. enthalten.
- Ziel im folgenden:
suche eine Definition 'Korrektheit' für Histories.

[Einleitung/
Probleme](#)

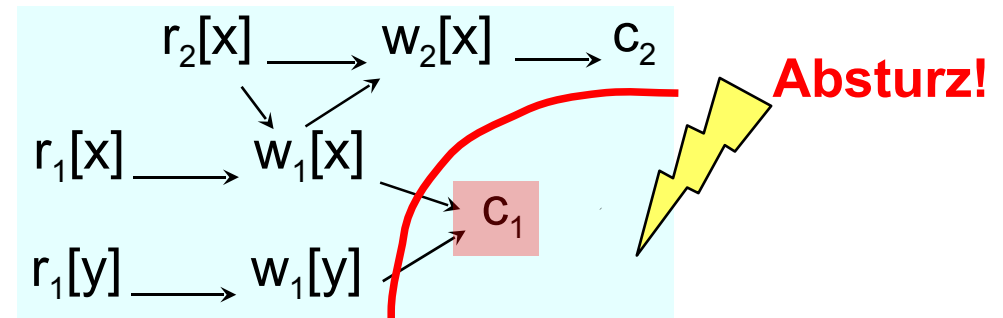
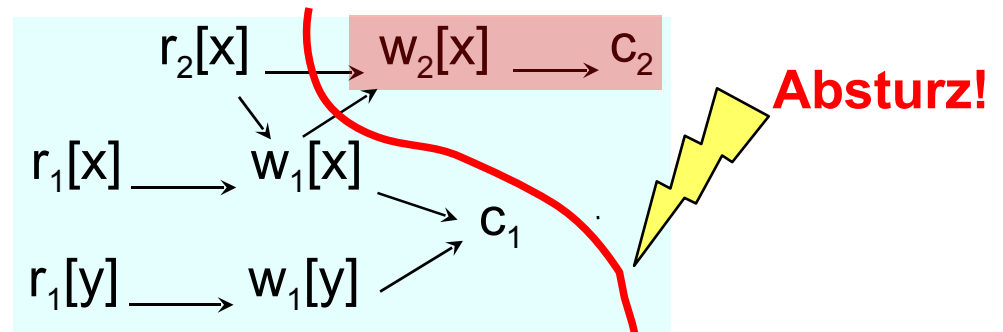
[Definitionen](#)

[Locking](#)



Committed Projections

- *Committed projection einer History H* –
Abkürzung: $C(H)$:= resultiert aus H durch Löschen aller Operationen, die nicht committed sind.
- Illustration:



Präfixe von Histories

Einleitung/
Probleme

[Definitionen.](#)

Locking

$H = o_1 \dots o_n$
(linear,
der Einfachheit
halber)

Präfixe:
 $H' = o_1 \dots o_i$
 $H'' = o_1 \dots$
 o_m

α = "History
enthält weniger
als 10
Operationen"

Wenn $H \alpha$
erfüllt, erfüllen
es auch H' , H''
und jeder
andere Präfix. α
ist *prefix*
commit-closed.

β = "Alle
Operationen
sind Lese-
Operationen"

Wenn $H \beta$ erfüllt,
erfüllen es auch
 H' , H'' und jeder
andere Präfix. β
ist *prefix*
commit-closed.

γ = "History
enthält mehr als
10 Operationen"

H' muß γ nicht
erfüllen, selbst
wenn H es
erfüllt. γ ist nicht
prefix commit-
closed.

Weitere Beispiele für Eigenschaften,
die *prefix commit-closed* sind?



Präfixe von Histories (2)

- Eigenschaft von Histories ist *prefix commit-closed*, wenn gilt:
H erfüllt die Eigenschaft.
 $\Rightarrow C(H')$ erfüllt die Eigenschaft, H' ist Präfix von H.

$C(H) :=$ *committed projection*, d. h.
nur Operationen von Transaktionen,
die committed sind.

- Vorangegangene Folie hat so getan,
als ob man alle Präfixe betrachtet.
Es reicht, committed projections der Präfixe
zu betrachten.

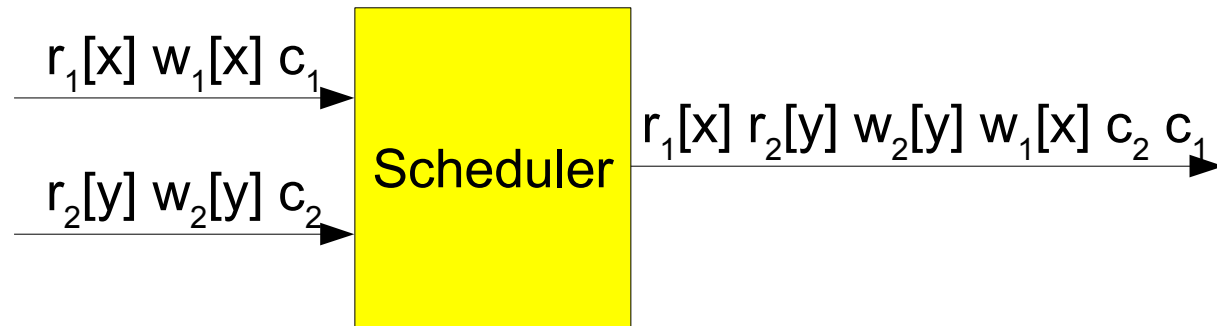
Einleitung/
Probleme

[Definitionen.](#)

Locking

Korrektheit von Histories

- Überlegung hinter vorangegangener Definition:
 - Korrektheitskriterium für Histories muss *prefix commit-closed* Eigenschaft haben
 - Scheduler generiert History; generiert aber auch jedes Präfix.



- Absturz des DBMS –
History nach Wiederaufnahme des Betriebs hat diese Eigenschaft → *History ist korrekt!*



Eigenschaften von Histories

- Conflict-Serializability (**CSR**)
 - Konflikt-Serialisierbarkeit
- Recoverability (**RC**)
 - Rücksetzbarkeit
- Avoids Cascading Aborts (**ACA**)
 - keine kaskadierenden Abbrüche
- Strictness (**ST**)
 - keine Zwischenergebnisse von uncommitted Transaktionen lesen

Einleitung/
Probleme

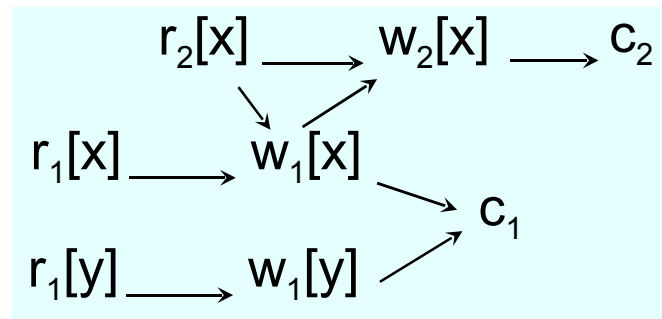
Definitionen

Locking



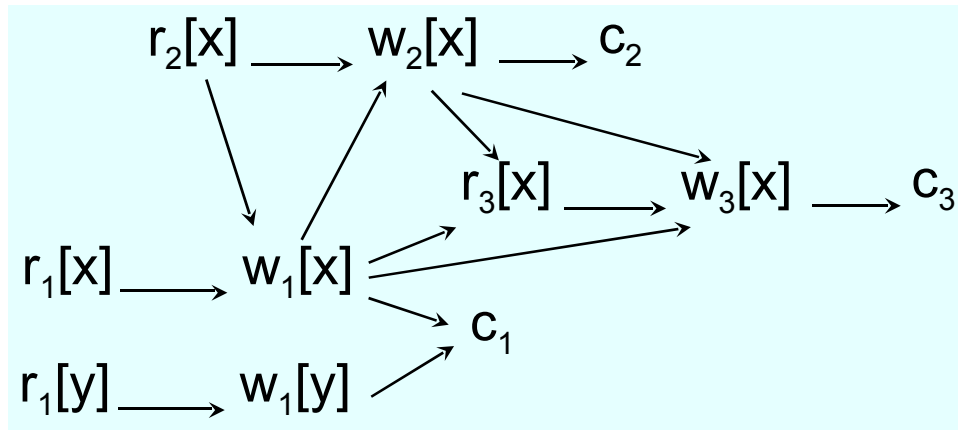
Konflikt-Serialisierbarkeit (1)

- Committed projection einer History H :
 $C(H) :=$ resultiert aus H , indem alle Operationen gelöscht werden, die nicht committed sind.
- H ist serialisierbar, wenn $C(H)$ zu serieller History H_s äquivalent ist.
- Ist diese History serialisierbar?
Wenn ja, wie sieht äquivalente serielle History aus?



Konflikt-Serialisierbarkeit (2)

- Konflikt-Serialisierbarkeit ist offensichtlich prefix commit-closed.
- Illustration:



Recoverability (1)

- ‘Commit einer Transaktion’ – kein Abort mehr mgl.
- Commit nur, wenn alle Änderungen an Datenobjekten, die T gelesen hat, committet sind.
- Gegenbeispiel:
Dirty Read.

Serialisierbar?

Schritt	T1	T2
1	Read(A, a1)	
2	a1 := a1 - 300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 * 1.03
6		Write(A, a2)
7		commit
8	Read(B, b1)	
9	...	
10	abort	



Recoverability (2)

- *Ausführung ist recoverable* :=
Commit von T nach Commit aller Transaktionen,
von denen T gelesen hat.
- folgendes darf **nie** passieren:
 $r_1[x] w_1[x] \underline{r_2[x] w_2[x] c_2} a_1$

Einleitung/
Probleme

[Definitionen](#)

Locking





Cascading Aborts (1)

- Transaktion T abortet, wenn sie nicht korrekt beenden kann.
- T muß zurückgesetzt werden:
 - Writes von T,
 - dto. alle anderen Transaktionen, die solche Änderungen gelesen haben.*Undo* kann zu *cascading abort* führen.

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)





Cascading Aborts (2)

- Beispiel für cascading abort:
 - x und y haben zunächst Wert 1.
 - Zwei Transaktionen T_1 und T_2
 $r_1[x] w_1[x] r_2[x] a_1 \dots$
 - Abort von T_1 , undo $w_1[x]$
 $\Rightarrow T_2$ muß ebenfalls aborten.
- Cascading aborts sind möglich, auch wenn die History recoverable ist.

Einleitung/
Probleme

Definitionen

Locking



Cascadelessness

- Cascading aborts sind unerwünscht:
 - Sie ziehen ‘Buchhaltung’ nach sich,
 - Zahl der Transaktionen, die aborten müssen, ist nicht beschränkt.
- *DBMS ist cascadeless* :=
Jede Transaktion liest Datenobjekte nur von committeten Transaktionen.

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)





Strictness

- *Strictness* := keinen Wert einer nicht abgeschlossenen Transaktion lesen oder überschreiben
- Problem: Undo basiert auf *Before-Images*.
- Veranschaulichung:
 1. $w_1(x, 2); w_2(x, 3); a_1$
 2. $w_1(x, 2); w_2(x, 3); a_1; a_2$
- Strikte History: $w_i(x, v)$ wird verzögert, bis alle Transaktionen, die x geschrieben haben, entweder committet oder abortet haben
 1. $w_1(x, 2); a_1; w_2(x, 3)$

Einleitung/
Probleme

Definitionen

Locking



Beispiele

- $T_1 = w_1[x] w_1[y] w_1[z] c_1$
- $T_2 = r_2[u] w_2[x] r_2[y] w_2[y] c_2$
- $H_7 = w_1[x] w_1[y] r_2[u] w_2[x] r_2[y] w_2[y] c_2 w_1[z] c_1$
- $H_8 = w_1[x] w_1[y] r_2[u] w_2[x] r_2[y] w_2[y] w_1[z] c_1 c_2$
- $H_9 = w_1[x] w_1[y] r_2[u] w_2[x] w_1[z] c_1 r_2[y] w_2[y] c_2$
- $H_{10} = w_1[x] w_1[y] r_2[u] w_1[z] c_1 w_2[x] r_2[y] w_2[y] c_2$
- Welche Histories sind recoverable,
welche kommen ohne cascading abort aus,
welche sind strict?

Einleitung/
Probleme

Definitionen

Locking





Serialisierbarkeitsgraph (1)

- Test, ob ein Schedule (= Transaktionen, zusammen mit ihren Operationen) serialisierbar ist:
 - Erzeuge Serialisierbarkeitsgraphen bzw. **Abhängigkeitsgraphen**.
 - Knoten = Transaktionen,
 - (gerichtete) Kante = Abhängigkeit zwischen zwei Transaktionen: Transaktionen greifen auf das gleiche Datenobjekt zu, und Operationen konfliktieren.

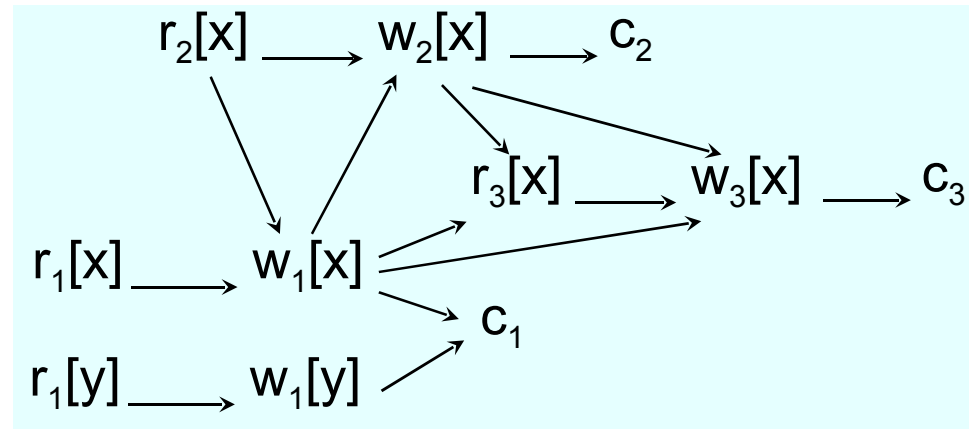
Einleitung/
Probleme

[Definitionen](#)

[Locking](#)

Serialisierbarkeitsgraph (2)

- Wie sieht der Serialisierbarkeitsgraph aus?



Einleitung/
Probleme

[Definitionen](#)

Locking



Serialisierbarkeitsgraph (3)

- Theorem: Schedule ist serialisierbar, wenn entsprechender Abhängigkeitsgraph *zykelfrei* ist.
- Denn: Partielle Ordnung, zu totaler Ordnung erweiterbar – äquivalenter serieller Schedule.

[Einleitung/
Probleme](#)

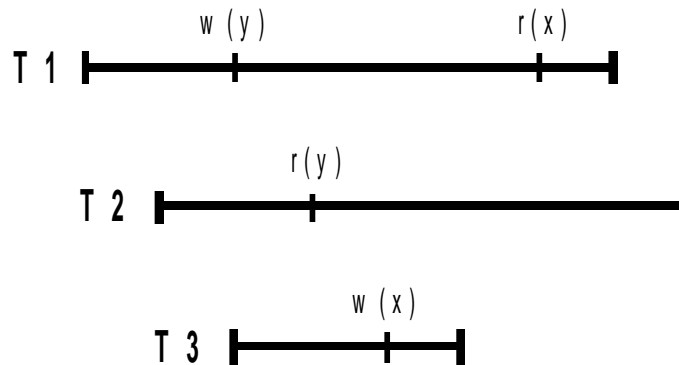
[Definitionen](#)

[Locking](#)

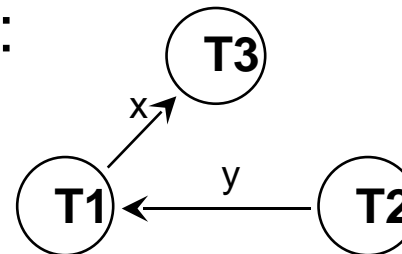


Serialisierbarkeitsgraph – Beispiel

- $r(x)/w(x)$ – Lese-/Schreibzugriff auf Datenobjekt x .
- Schedule:



- Abhängigkeitsgraph:



- azyklisch \rightarrow Schedule ist serialisierbar.
- Serialisierungsreihenfolge: $T3 < T1 < T2$



Serialisierbarkeitsgraph – Diskussion

Ansatz ist nicht praktikabel.

- Serialisierbarkeit von Schedules nur im nachhinein überprüfbar.
- Administrativer Overhead ist zu hoch: Abhängigkeiten zu bereits terminierten Transaktionen müssen ebenfalls berücksichtigt werden. z.B. wird Beziehung zwischen T1 und T3 erst nach Commit von T3 klar.

[Einleitung/
Probleme](#)

[Definitionen](#)

[Locking](#)



Sperren und Sperrprotokolle

Verzögern und Zurücksetzen

- Verzögern und Zurücksetzen mittels Locking sind übliche Techniken in der Transaktionsverwaltung.

- Locking:
 - Datenobjekte werden zum Schreiben und/oder Lesen gesperrt
 - Verzögerte Transaktionen müssen warten bis Lock aufgehoben
 - zahlreiche Strategien zum Locking möglich, im folgenden: 2-Phase-Locking

Im Allgemeinen immer noch schneller als rein serielle Ausführung der Transaktionen

Einleitung/
Probleme

Definitionen

Locking



Locking

- Transaktion i setzt Lock für Operation o auf Datenobjekt x – Notation: $ol_i[x]$
- Einfachster Fall: Nur Read Locks und Write Locks („RX Locking Scheme“).
 - read lock (rl): Lesesperre
 - write lock (wl): Schreibsperre
 - read/write unlock (ru/wu): Sperre aufheben

Einleitung/
Probleme

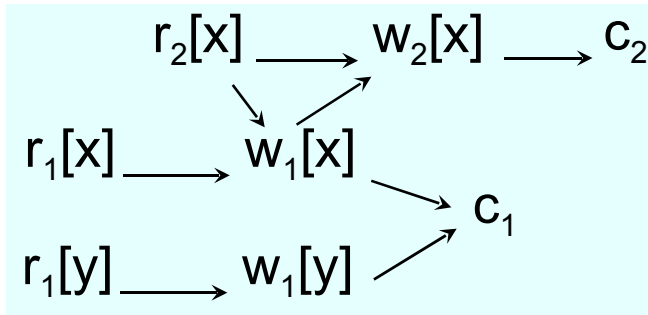
Definitionen

Locking

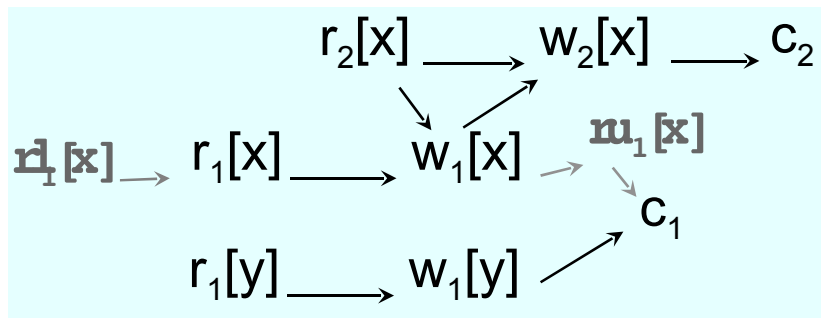


Locking - Beispiel

- ohne Locking



- T_1 setzt Read-Lock auf x





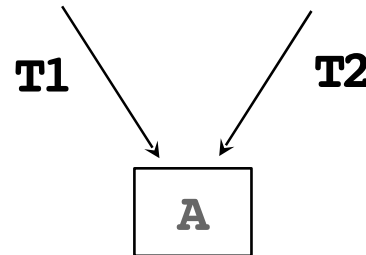
Sperrdisziplin

- Schreibzugriff $w[x]$ nur nach $wl[x]$
- Lesezugriff $r[x]$ nur nach $rl[x]$ oder $wl[x]$
- Transaktion darf nur Objekte sperren, die keine Locks von anderen Transaktionen tragen, sonst: verzögert bis ru/wu
- nach $rl[x]$ ist noch ein $wl[x]$ erlaubt, sonst keine weiteren Sperren
- nach dem Entsperren darf eine Transaktion das selbe Objekt nicht erneut sperren
- vor Commit alle Sperren aufheben

Einleitung/
Probleme
Definitionen
[Locking](#)

Konflikte beim Locking

- Locks können konfliktieren.



- Locks konfliktieren in gleicher Weise wie entsprechende Operationen.

	rl_i	wl_i
rl_j	y	n
wl_j	n	n

→ mehrere Lesesperren auf dem gleichen Objekt möglich

Zwei-Phasen-Sperrprotokoll

- Zwei-Phasen-Sperrprotokoll (two-phase locking, 2PL) stellt Serialisierbarkeit sicher.
 - Zwei Phasen:
 - Locks hinzunehmen
 - Locks freigeben

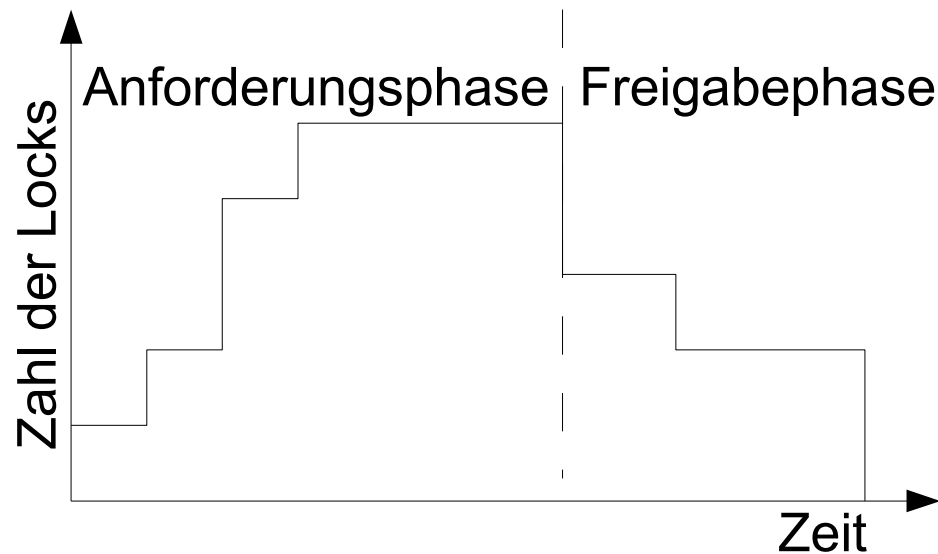


Illustration 2PL

- $T_1: r_1[x] \rightarrow w_1[y] \rightarrow c_1; T_2: w_2[x] \rightarrow w_2[y] \rightarrow c_2$

Einleitung/
Probleme

Definitionen

Locking

- Ohne 2PL:

$H_1 = r_1[x] \quad r_1[x] \quad ru_1[x] \quad wl_2[x] \quad w_2[x] \quad wl_2[y] \quad w_2[y] \quad wu_2[x]$
 $wu_2[y] \quad c_2 \quad wl_1[y] \quad w_1[y] \quad wu_1[y] \quad c_1$

- $r_1[x] < w_2[x] \wedge w_2[y] < w_1[y]$
 $\Rightarrow \text{SG}(H_1)$ hat zyklische Abhängigkeit $T_1 \rightarrow T_2 \rightarrow T_1$

- Mit 2PL:

$H_1 = r_1[x] \quad r_1[x] \quad wl_1[y] \quad w_1[y] \quad c_1 \quad ru_1[x] \quad wu_1[y] \quad wl_2[x] \quad w_2[x]$
 $wl_2[y] \quad w_2[y] \quad c_2 \quad wu_2[x] \quad wu_2[y]$

Beispiel für Deadlock mit 2PL

- $T_1: r_1[x] \rightarrow w_1[y] \rightarrow c_1; T_2: w_2[y] \rightarrow w_2[x] \rightarrow c_2$
- Abfolge:
 1. Beide Transaktionen starten ohne Locks.
 2. TM sendet $r_1[x]$ an Scheduler.
 $rl_1[x]$, Scheduler sendet $r_1[x]$ an DM.
 3. TM sendet $w_2[y]$ an Scheduler.
 $wl_2[y]$, Scheduler sendet $w_2[y]$ an DM.
 4. TM sendet $w_2[x]$ an Scheduler.
 $wl_2[x]$ nicht möglich. Verzögerung.
 5. TM sendet $w_1[y]$ an Scheduler.
 $wl_1[y]$ nicht möglich. Verzögerung.

Deadlock!
Muss extern
zurückgesetzt
werden

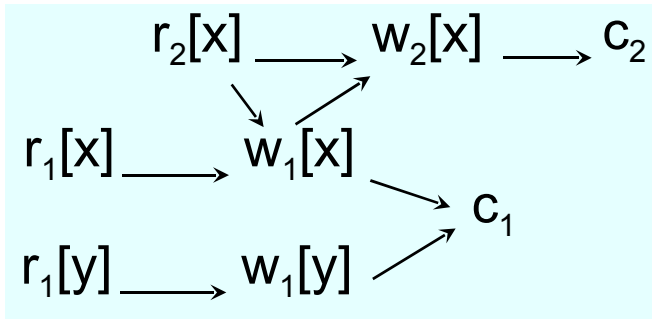
Einleitung/
Probleme

Definitionen

Locking

Beispiel 2PL

- Illustration:



- Wie läuft es ab, wenn zuerst $r_2[x]$?

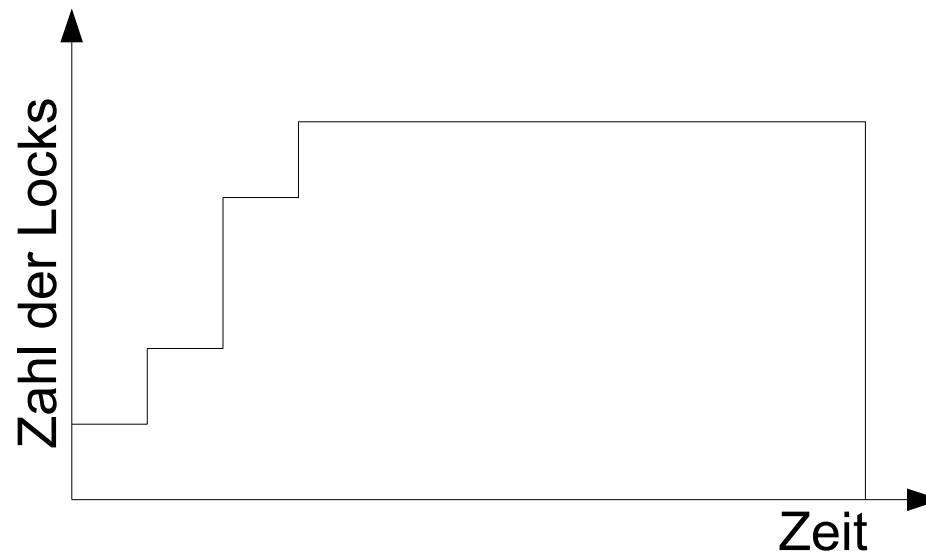
Einleitung/
Probleme

Definitionen

Locking

Strenges 2PL

- stellt Cascadelessness sicher:
Freigabe der Locks erst am Ende der Transaktion.



Einleitung/
Probleme

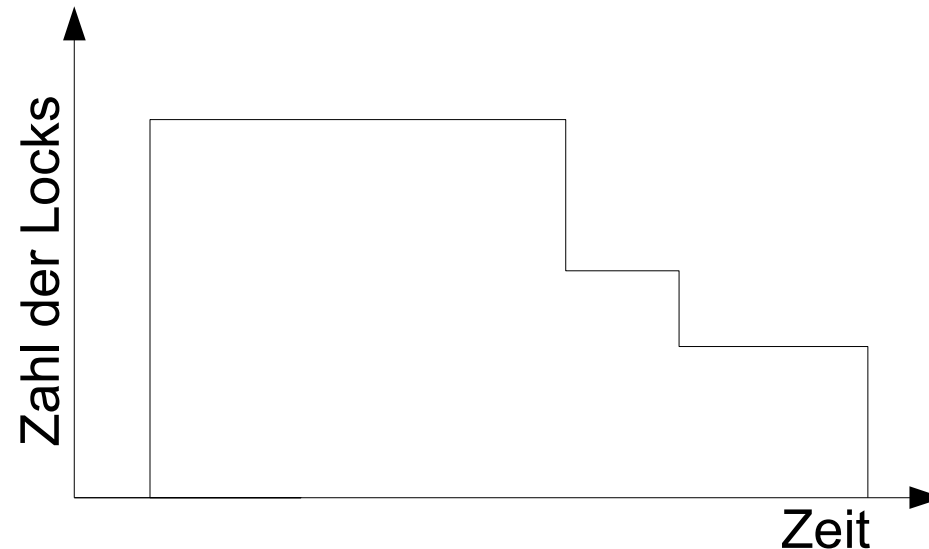
Definitionen

Locking



Konservatives 2PL

- stellt Deadlock-Freiheit sicher:
Alle Locks werden am Anfang der Transaktion
gesetzt



- Kombination möglich: **Konvervatives Strenges Zwei-Phasen-Sperrprotokoll**





Zusammenfassung

- Nebenläufiger Zugriff
 - fundamental wichtiges Anliegen.
- Serielle Ausführung wäre *immer* korrekt, ist wegen mangelhafter Performance aber nicht akzeptabel.
- Korrektheitskriterium
 - Äquivalenz zu serieller Ausführung.
- Two-Phase Locking stellt Serialisierbarkeit sicher.

Einleitung/
Probleme

Definitionen

Locking

