Kapitel 7: Nebenläufigkeit und Transaktionen

Gliederung

Einleitung/ Probleme
Definitionen
Locking

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

Weitere Probleme ohne Datenbanken (2)

– Folie aus Einleitung –

Einleitung/ Probleme
Definitionen
Locking

- Transaktionseigenschaften – insbesondere *Atomarität* und *Isolation*.
- Atomarität
  - Beispiel, „Bank-Szenario“:

Nummer	Inhaber	Stand
	Klemens	5000
	Rudi	200

  - Überweisung – zwei Elementaroperationen.
    – Abbuchung(Klemens, 500),
    – Einzahlung(Rudi, 500).
- Isolation – auch an diesem Beispiel erklärbar.
- Transaktionen.

X Desktop

```
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>
```

```
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> COMMIT;

Commit complete.

SQL> SELECT * FROM Bankkonten;

   KONTONR INHABER                   STAND
---------- -------------------- ----------
      1234 Klemens Boehm               300
      5678 Gunter Saake                700

SQL>
```

Markers: 1, 2, 3, 4

```
SQL> SET AUTOCOMMIT OFF;
SQL> SELECT * FROM Bankkonten;   1

   KONTONR INHABER                   STAND
---------- -------------------- ----------
      1234 Klemens Boehm               500
      5678 Gunter Saake                500

SQL>
SQL>
SQL>
SQL>
SQL> SELECT * FROM Bankkonten;   2

   KONTONR INHABER                   STAND
---------- -------------------- ----------
      1234 Klemens Boehm               500
      5678 Gunter Saake                500

SQL>
SQL>
SQL>
SQL>
SQL> SELECT * FROM Bankkonten;   3

   KONTONR INHABER                   STAND
---------- -------------------- ----------
      1234 Klemens Boehm               500
      5678 Gunter Saake                500

SQL>
SQL>
SQL>
SQL>
SQL> SELECT * FROM Bankkonten;   4

   KONTONR INHABER                   STAND
---------- -------------------- ----------
      1234 Klemens Boehm               300
      5678 Gunter Saake                700

SQL>
```

```
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 Alternative zu Commit.

Synchronisation in Datenbanken (1)

- 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*.

Synchronisation in Datenbanken (2)

Einleitung/ Probleme
Definitionen
Locking

- **Serielle Ausführung** von Anwendungsprogrammen.
  - Jene Illusion läßt sich ohne jeglichen Synchronisationsaufwand erreichen.
  - Datenbank ist nach Ende jeder Programmausführung konsistent.
  - Aber: Extreme Wartezeiten und unbefriedigende Ausnutzung der Ressourcen. (CPU ist während Kommunikation und I/O nicht aktiv.)

Synchronisation im Allgemeinen

Einleitung/ Probleme
Definitionen
Locking

Unkontrollierte **nicht-serielle Ausführung** führt zu anderen Problemen, insbesondere Inkonsistenz:

- Lost Updates,
- inkonsistente Lesezugriffe („non-repeatable read"),
- Dirty Reads, d. h. Reads von Updates, die noch nicht committet sind.
- Phantome.

Lost Update

Einleitung/ Probleme
Definitionen
Locking

- Programm T_1 transferiert EUR 300,- von Konto A auf Konto 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 in Schritt 6 überschreibt.

Schritt	*T1*	*T2*
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

Einleitung/ Probleme
Definitionen
Locking

- *Commit*, *Abort*.
- Programm T_2 schreibt Zinsen gut, basierend auf einem Wert, der nicht zu einem konsistenten Zustand gehört.
- Denn später erfolgt Abort von T_1.

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**	

Einleitung/
Probleme
Definitionen
Locking

Non-Repeatable Reads

Programm liest Datenobjekt mehr als einmal und sieht Änderung, die anderes Programm 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	…	

z

Einleitung/
Probleme
Definitionen
Locking

Transaktionen (1)

- *Transaktion* := Ausführung eines Programms, das auf die Datenbank (lesend oder schreibend) zugreift.
- Programmausführung ≠ 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

Transaktionen (2)

- Beispiel:

```
Procedure P begin
    Start;
    temp := Read(x);
    temp := temp + 1;
    Write(x, temp);
    Commit
end
```

Operationen (→ temp := Read(x); Write(x, temp);)

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

Einleitung/
Probleme
Definitionen
Locking

Konflikt

- *Zwei Operationen p, q konfligieren* :=
  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
Read	**y**	**n**
Write	**n**	**n**

Transaktion – formale Definition (1)

Einleitung/ Probleme
Definitionen
Locking

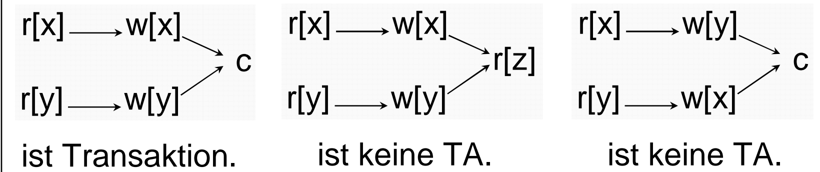
Transaktion ist partielle Ordnung
mit Ordnungsrelation <, so daß gilt

1. $T_i \subseteq \{r_i[x], w_i[x] | x$ 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 $w_i[x] <_i r_i[x]$.

Transaktion – formale Definition (2)

Einleitung/ Probleme
Definitionen
Locking

Beispiele:



ist Transaktion. ist keine TA. ist keine TA.

Histories (1)

Einleitung/ Probleme
Definitionen
Locking

- Ausführung der Operationen mehrerer Transaktionen, die miteinander ‘verzahnt’ sind, d. h. nebenläufig ablaufen.
- Formal –
  $T = \{T_1, T_2, \ldots, T_n\}$ ist Menge von Transaktionen.
  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. $p, q \in H$ konfligieren $\Rightarrow p <_H q$ oder $q <_H p$

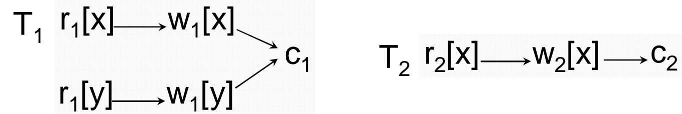
Histories (2)

Einleitung/ Probleme
Definitionen
Locking

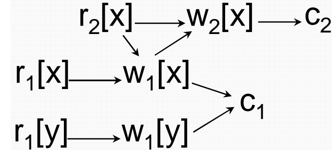
- *History* := Präfix einer vollständigen History.

Histories – Beispiele

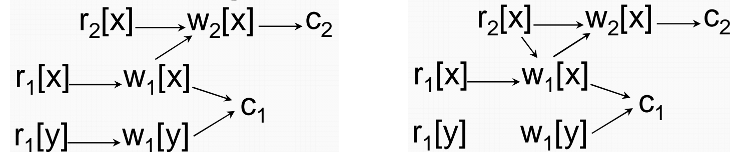
- Gegeben zwei Transaktionen:



- Eine vollständige History, OK:



- Nicht vollständige Histories:

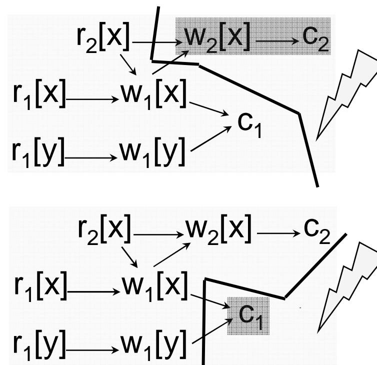


Histories (3)

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

Histories (4)

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



Histories (5)

$H=o_1 \dots o_n$ (linear, der Einfachheit halber)	α=”History enthält weniger als 10 Operationen”	β=”Alle Operationen sind Lese-Operationen”	γ=”History enthält mehr als 10 Operationen”
Präfixe: $H'=o_1 \dots o_l$ $H''=o_1 \dots o_m$	Wenn H α erfüllt, erfüllen es auch H’, H” und jeder andere Präfix. α ist *prefix commit-closed.*	Wenn H β erfüllt, erfüllen es auch H’, H” und jeder andere Präfix. β ist prefix commit-closed.	H’ muß γ nicht erfüllen, selbst wenn H es erfüllt. γ ist nicht prefix commit-closed.

Histories (6)

- 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.

Histories (7)

- Überlegung hinter vorangegangener Definition:
  - Korrektheitskriterium für Histories
    muß diese Eigenschaft haben.
  - Scheduler generiert History;
    generiert aber auch jedes Präfix.
  - Absturz des DBMS –
    History nach Wiederaufnahme des Betriebs
    hat diese Eigenschaft ebenfalls.
    D. h. wir wissen, daß History korrekt ist.

Reads-from Beziehung zwischen Transaktionen (1)

- *Transaktion T_i liest von Transaktion T_j in einer bestimmten History*, 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.

Reads-from Beziehung zwischen Transaktionen (2)

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

Äquivalenz von Histories

- Mehr als eine Definition:
  - (Konflikt-)Äquivalenz,
  - (Sicht-)Äquivalenz.
- Im Folgenden nur Konflikt-Äquivalenz.
- Definition Konflikt-Äquivalenz':
  Histories H, H' sind (Konflikt-)äquivalent, 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 $p_i <_{H'} q_j$.

Äquivalenz von Histories – Beispiele (1)

- Gegeben zwei Transaktionen:

T_1: $r_1[x] \rightarrow w_1[x] \rightarrow c_1$; $r_1[y] \rightarrow w_1[y] \rightarrow c_1$

T_2: $r_2[x] \rightarrow w_2[x] \rightarrow c_2$

- Eine vollständige History, OK:

$r_2[x] \rightarrow w_2[x] \rightarrow c_2$; $r_2[x] \rightarrow w_1[x] \rightarrow w_2[x]$
$r_1[x] \rightarrow w_1[x] \rightarrow c_1$
$r_1[y] \rightarrow w_1[y] \rightarrow c_1$

- 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 werden committen. (Unten ebenso.)

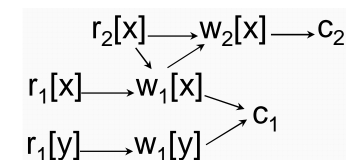
- History 2:

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

- Sind diese Histories Konflikt-äquivalent?

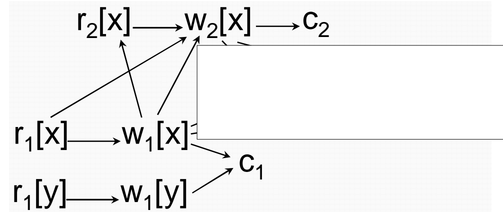
Serialisierbarkeit (1)

- *Committed projection einer History H* – Abkürzung: 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?



Serialisierbarkeit (2)

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



z

Recoverability (1)

- 'Commit einer Transaktion' –
  DBMS kann sie nicht mehr aborten.
- 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.

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.

Cascading Aborts (2)

Einleitung/ Probleme
Definitionen
Locking

- Beispiel für cascading abort:
  - ◆ x und y haben zunächst Wert 1.
  - ◆ Zwei Transaktionen T_1 und T_2.
  - ◆ Reihenfolge der Operationen: $Write_1(x, 2); Read_2(x); Write_2(y, 3)$.
  - ◆ Undo von $Write_1(x, 2)$ $\Rightarrow$ T_2 muß ebenfalls aborten.
- Cascading aborts sind möglich, auch wenn die History recoverable ist.

Cascadelessness

Einleitung/ Probleme
Definitionen
Locking

- 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.

Strictness

Einleitung/ Probleme
Definitionen
Locking

- Undo basiert üblicherweise auf *Before-Images*.
- Veranschaulichung des Problems – ursprünglicher Wert von x: 1
  1. $w_1(x, 2); w_2(x, 3); abort_1$
  2. $w_1(x, 2); w_2(x, 3); abort_1; abort_2$
- *Strictness* :=
  write(x, val) wird verzögert,
  bis alle Transaktionen, die x geschrieben haben,
  entweder committet oder abortet haben,
  \+ Cascadelessness.

Beispiele

Einleitung/ Probleme
Definitionen
Locking

- $T_1 = w_1[x]\ w_1[y]\ w_1[z]\ c_1$
- $T_2 = r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$

- $H_7 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ c_2\ w_1[z]\ c_1$
- $H_8 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ r_2[y]\ w_2[y]\ w_1[z]\ c_1\ c_2$
- $H_9 = w_1[x]\ w_1[y]\ r_2[u]\ w_2[x]\ w_1[z]\ c_1\ r_2[y]\ w_2[y]\ c_2$
- $H_{10} = w_1[x]\ w_1[y]\ r_2[u]\ w_1[z]\ c_1\ w_2[x]\ r_2[y]\ w_2[y]\ c_2$

- Welche Histories sind recoverable,
  welche kommen ohne cascading abort aus,
  welche sind strict?

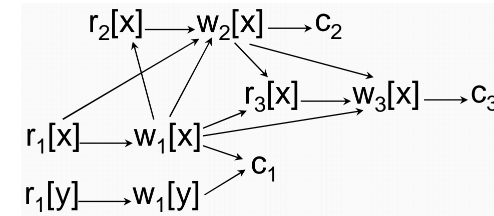
z

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 konfligieren.

Serialisierbarkeitsgraph (2)

- Wie sieht Serialisierbarkeitsgraph aus?

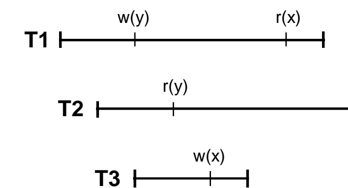


Serialisierbarkeitsgraph (3)

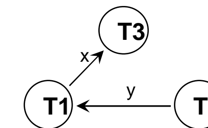
- Theorem: Schedule ist serialisierbar, wenn entsprechender Abhängigkeitsgraph *zykelfrei* ist.
- Denn: Partielle Ordnung, zu totaler Ordnung erweiterbar – äquivalenter serieller Schedule.

Serialisierbarkeitsgraph – Beispiel

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



- Abhängigkeitsgraph:



- azyklisch → Schedule ist serialisierbar.
- Serialisierungsreihenfolge: T3 < T1 < T2

Serialisierbarkeitsgraph – Diskussion

Einleitung/
Probleme
Definitionen
Locking

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.

Verzögern und Zurücksetzen

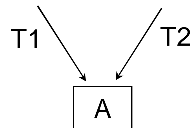
Einleitung/
Probleme
Definitionen
Locking

- Verzögern und Zurücksetzen sind übliche Techniken in der Transaktionsverwaltung.

Locking (1)

Einleitung/
Probleme
Definitionen
Locking

- Lock für jedes Datenobjekt und für jede Art von Operation – Notation: $ol_i[x]$
- Transaktion verschafft sich Lock, Transaktion gibt Lock frei.
- Locks können konfligieren.



- Locks konfligieren in gleicher Weise wie entsprechende Operationen.
- Locking für sich betrachtet ist nicht ausreichend! (Wird gleich deutlich.)

Locking (2)

Einleitung/
Probleme
Definitionen
Locking

- Zwei Phasen:
  - Phase, in der Locks hinzugenommen werden,
  - Phase, in der Locks freigegeben werden.
- **Zwei-Phasen Sperrprotokoll** (two-phase locking, 2PL) stellt Serialisierbarkeit sicher.

Locking (3)

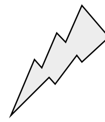
- Einfachster Fall: Nur Read Locks und Write Locks („RX Locking Scheme“).

Illustration 2PL

- T_1: $r_1[x] \rightarrow w_1[y] \rightarrow c_1$; T_2: $w_2[x] \rightarrow w_2[y] \rightarrow c_2$
- $H_1 = rl_1[x]\ r_1[x]\ ru_1[x]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ wu_2[x]$ $wu_2[y]\ c_2\ wl_1[y]\ w_1[y]\ wu_1[y]\ c_1$
- $r_1[x] < w_2[x] \wedge w_2[y] < w_1[y]$
  $\Rightarrow$ SG(H_1) hat zyklische Abhängigkeit $T_1 \rightarrow T_2 \rightarrow T_1$
- T_1 verletzt two-phase Regel.
- History, die two-phase Regel nicht verletzt:
  $H_1 = rl_1[x]\ r_1[x]\ wl_1[y]\ w_1[y]\ c_1\ ru_1[x]\ wu_1[y]\ wl_2[x]\ w_2[x]$ $wl_2[y]\ w_2[y]\ c_2\ wu_2[x]\ wu_2[y]$

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 halten zunächst keine 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.

Locking (4)

- Illustration:

  $r_2[x] \rightarrow w_2[x] \rightarrow c_2$
  $r_1[x] \rightarrow w_1[x] \rightarrow c_1$
  $r_1[y] \rightarrow w_1[y] \rightarrow c_1$
  ($r_2[x] \rightarrow w_1[x]$, $w_1[x] \rightarrow w_2[x]$)

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

Locking (5)

Einleitung/ Probleme
Definitionen
Locking

- **Strenges Zwei-Phasen Sperrprotokoll** stellt Cascadelessness sicher: Freigabe der Locks erst am Ende der Transaktion.

Zusammenfassung

Einleitung/ Probleme
Definitionen
Locking

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

Mögliche Prüfungsfragen (1)

- Was ist Isolation? Was ist der Zusammenhang zwischen Isolation und Serialisierbarkeit?
- Welche Probleme können bei unkontrollierter nebenläufiger Ausführung von Transaktionen auftreten?
- Wieso ist der Begriff 'prefix commit closed' im aktuellen Kontext wichtig?

Mögliche Prüfungsfragen (2)

- Ist eine gegebene History serialisierbar/recoverable/cascadeless?
- Haben zwei Konflikt-äquivalente Histories stets die gleichen Reads-from Beziehungen?
- Warum verwendet man i. d. R. nicht den Serialisierbarkeitsgraphen, um Serialisierbarkeit sicherzustellen?
- Bei Deadlocks wird i. d. R. eine Transaktion zurückgesetzt. Kann es vorkommen, daß die gleiche Transaktion (a) mehrmals (b) beliebig oft zurückgesetzt wird? Wenn ja, was kann man jeweils dagegen tun?