

3.4 Transaktionen (1)

Definition

- Eine *Transaktion* ist eine logische Einheit von
- Datenbankoperationen, die einen konsistenten
- Zustand in einen konsistenten Zustand überführt.

Bemerkungen

- in SQL wird eine Transaktion begonnen mit
 - *begin [work | transaction]* und beendet mit
 - *commit* oder *rollback*
- Transaktionsbegriff unabhängig vom Datenmodell
 - (relational, hierarchisch, objektorientiert, ...)

1

3.4.1 Anforderungen (1)

Themen in diesem Abschnitt

- Probleme bei der Transaktionsverarbeitung
 - *Error Recovery*
 - Fehlerbehandlung schon beim Einbenutzerbetrieb wichtig
 - *Concurrency Control*
 - Nebenläufigkeit von Transaktionen mehrere Benutzer
 - *Backup/Restore*
 - Online Backup bei laufenden Transaktionen
- Anforderungen an Transaktionsverarbeitung
 - Atomicity, Consistency, Isolation, Durability (ACID)
 - Isolationsgrad (Transaction Isolation Level)

3

3.4 Transaktionen (2)

Einfaches Modell einer Datenbank

- Sammlung benannter Datenobjekte
 - alle folgenden Betrachtungen sind unabhängig von der Größe eines Datenobjekts (*Granularität*)
 - Granularität kann z.B. Feld, Tupel, Plattenblock sein
- zwei grundlegende Operationen
 - *read(X)* liest Objekt *X* der Datenbank in Programmvariable
 - *write(X)* schreibt Wert Programmvariable in Objekt *X*

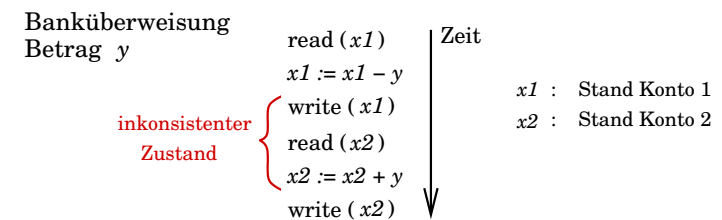
Bemerkung

- ein SQL-Kommando beinhaltet i.allg. mehrere grundlegende Operationen
- Beispiel: *update produkt set preis=preis+1 where pnr='P1'*;

2

3.4.1 Anforderungen (2)

Error Recovery

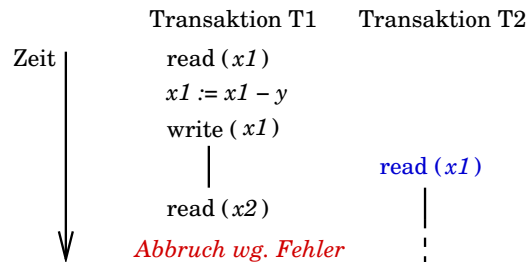


- Wie kann auf logischen Fehler innerhalb Transaktion reagiert werden? Nötig: Möglichkeit zum Rollback
- Was ist bei Systemabsturz im inkonsistenten Zustand?
- Was ist bei Systemabsturz nach Beendigung Transaktion?

4

3.4.1 Anforderungen (3)

Concurrency Control (1)

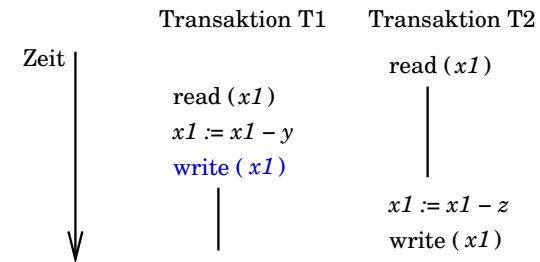


- T2 liest Daten, die nie committed werden (*dirty data*)
=> Überweisung T2 führt zu falschem Kontostand
- Phänomen wird als *Dirty Read* bezeichnet

5

3.4.1 Anforderungen (5)

Concurrency Control (3)

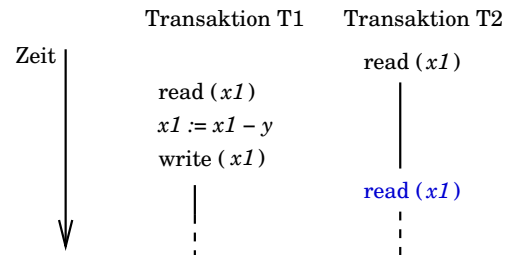


- Update von T1 geht verloren (*Lost Update*)
=> falscher Kontostand nach Abschluss beider Transaktionen
- tritt in Tateinheit mit Nonrepeatable Read auf (Warum?)

7

3.4.1 Anforderungen (4)

Concurrency Control (2)

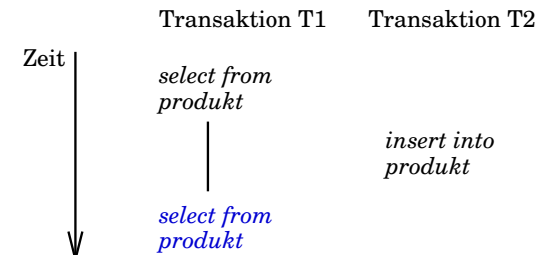


- T2 liest mehrmals hintereinander verschiedene Werte desselben Datensatzes, ohne ihn zu verändert zu haben
- Phänomen wird als *Nonrepeatable Read* bezeichnet

6

3.4.1 Anforderungen (6)

Concurrency Control (4)

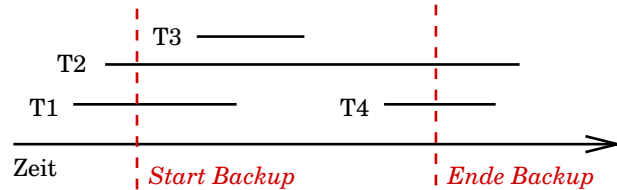


- T1 sieht beim zweiten Select zusätzliche Tupel die beim ersten Select nicht da waren (*Phantomtupel*)
- Unterschied zum Nonrepeatable Read: alle Daten unverändert

8

3.4.1 Anforderungen (7)

Backup/Recovery



- Offline Backup unproblematisch: alle Transaktionen beendet
- Online Backup problematisch: Backup muss konsistente Momentaufnahme (*Snapshot*) der Datenbank sichern
=> Backupoperation muss selber Transaktion sein, in der kein Nonrepeatable Read und keine Phantomtupel auftreten

9

3.4.1 Anforderungen (9)

Gelegentlich macht man Abstriche bzgl. Isolation

- vollständige Isolation (*Serialisierbarkeit*) verringert Durchsatz nebenläufiger Transaktionen
- für manche Transaktionen (z.B. nur lesendes Backup) keine vollständige Isolation erforderlich

SQL2 definiert vier Isolationsgrade

- *read uncommitted*, *read committed*, *repeatable read*, *serializable*
- nur *serializable* garantiert tatsächliche Transaktionsisolation
- nicht alle DBS implementieren alle Isolationsgrade (PostgreSQL: *read committed*, *serializable*)
- kann gesetzt werden nach Transaktionsbeginn mit `set transaction isolation level <isolationsgrad>;`

11

3.4.1 Anforderungen (8)

Wünschenswerte ACID-Eigenschaften

- *Atomicity*
Transaktion wird entweder ganz oder gar nicht ausgeführt
- *Consistency*
Transaktion überführt konsistenten Zustand in konsistenten Zustand. Innerhalb Transaktion Inkonsistenz möglich.
- *Isolation*
Änderungen in einer Transaktion sind bis zum Abschluss unsichtbar für andere Transaktionen.
- *Durability*
Nach Abschluss Transaktion bleiben Änderungen bestehen, auch im Fall eines folgenden Systemabsturzes

10

3.4.1 Anforderungen (10)

Wie die Isolationsgrade die Isolation verletzen:

Isolationsgrad	Dirty Read	Nonrepeatable Read	Phantomtupel
READ UNCOMMITTED	J	J	J
READ COMMITTED	N	J	J
REPEATABLE READ	N	N	J
SERIALIZABLE	N	N	N

Bemerkung:

- DBS, das Levels ungleich *serializable* unterstützt, muss zusätzliche Kontrollmechanismen anbieten
- solche Mechanismen sind aber nicht in SQL2 spezifiziert
- typischerweise sind das die zwei Befehle

```
LOCK TABLE
SELECT FOR UPDATE
```

12

3.4.2 Serialisierbarkeit (1)

Ziel:

- Isolation von Transaktionen, d.h. Transaktionen sollen nichts voneinander merken

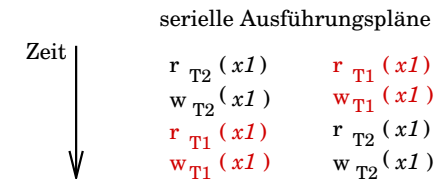
Fragestellungen:

- Wie können wir Isolation formal definieren? (Begriffe *Ausführungsplan*, *Serialisierbarkeit*)
- Wie können wir fehlende Isolation erkennen? (Algorithmus auf Basis unserer Definition)

13

3.4.2 Serialisierbarkeit (3)

ideale Isolation bei *serieller* Ausführung:
jede Transaktion hat Datenbank für sich alleine



Nachteil:

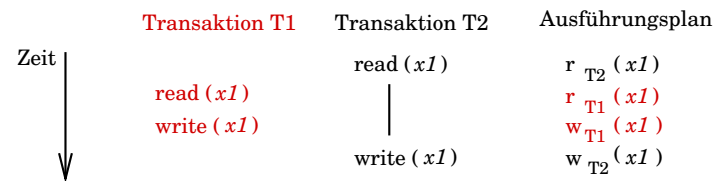
- Transaktionen müssen aufeinander warten
=> keine Nebenläufigkeit
=> geringer Durchsatz an Transaktionen

15

3.4.2 Serialisierbarkeit (2)

Ausführungsplan (*Schedule*)

= zeitliche Abfolge elementarer Operationen



Bemerkungen

- Annahme: elementare Operationen können nicht gleichzeitig durchgeführt werden
- irreführende Bezeichnung "plan": nicht DBS bestimmt Schedule, sondern Clientanwendungen bzw. Anwender M.a.W. am "Ausführungsplan" ist nichts "geplant"!

14

3.4.2 Serialisierbarkeit (4)

Serielle Ausführung eigentlich nicht nötig

- es muss für jede Transaktion nur so aussehen
· als wäre sie isoliert
- dazu reicht Existenz eines *äquivalenten seriellen* Ausführungsplans

Solcher Ausführungsplan heißt *serialisierbar*.

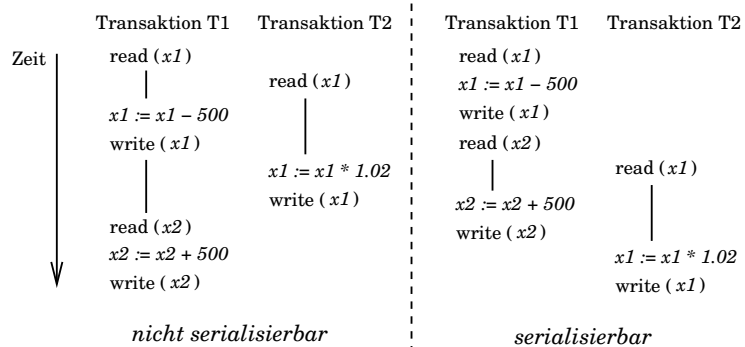
Bemerkung:

- Serialisierbarkeit ist abhängig von Definition der "Äquivalenz" von Ausführungsplänen (siehe weiter unten)
- verschiedene Definitionen der Schedule-Äquivalenz möglich.
Anschaulich: Schedules haben dieselben Auswirkungen in DB

16

3.4.2 Serialisierbarkeit (5)

Beispiel für (nicht) serialisierbaren Schedule:



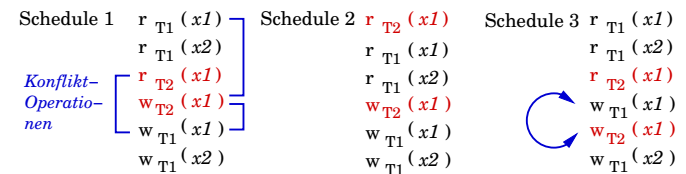
17

3.4.2 Serialisierbarkeit (7)

Konfliktäquivalenz (2)

Zwei Schedules heißen *konfliktäquivalent*, wenn die Reihenfolge in Konflikt stehender Operationen in beiden Schedules gleich ist.

Beispiel:



- Schedule 1 und 2 sind konfliktäquivalent
- Schedule 1 und 3 (und auch 2 und 3) nicht

19

3.4.2 Serialisierbarkeit (6)

Konfliktäquivalenz (1)

- suche hinreichendes Kriterium für Äquivalenz von Ausführungsplänen
- Frage: welche Operationen dürfen gefahrlos, d.h. ohne Auswirkung auf Endergebnis vertauscht werden?
 - Operationen auf verschiedenen Objekten immer vertauschbar
 - zwei read-Operationen desselben Objekts auch vertauschbar
 - ist bei zwei Operationen auf demselben Objekt eine write-Operation dabei, so kann Vertauschung das Endergebnis verändern (Beispiel: obiger nicht serialisierbarer Schedule) Solche Operationen stehen *im Konflikt* zueinander

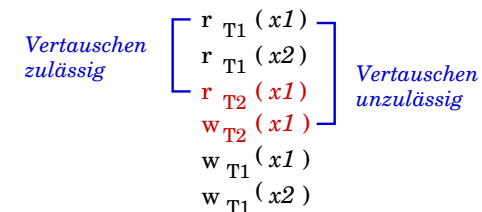
18

3.4.2 Serialisierbarkeit (8)

Anwendung Konfliktäquivalenz auf Serialisierbarkeit:

Schedule ist (*konflikt-*) *serialisierbar*, wenn er ohne Vertauschung von Konflikt-Operationen in einen seriellen Schedule umgeformt werden kann.

Beispiel



Übung:

Kriterium überprüfen an Schedules von Folie (5)

20

3.4.2 Serialisierbarkeit (9)

Prüfung auf Serialisierbarkeit

- bei zwei nebenläufigen Transaktionen können wir leicht auf Konfliktäquivalenz mit den zwei möglichen seriellen Ausführungsplänen prüfen
- Was ist aber bei n nebenläufigen Transaktionen?
 $n!$ mögliche serielle Schedules => Durchprobieren ineffizient

Frage:

- Können wir einfacher auf Serialisierbarkeit prüfen?

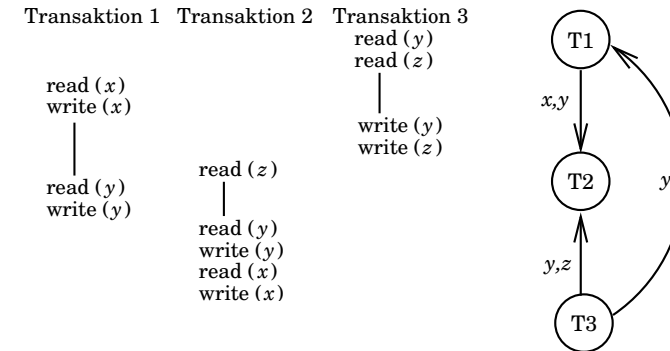
Antwort:

- Ja! Suche nach Zyklen im "Präzedenzgraphen".

21

3.4.2 Serialisierbarkeit (11)

Beispiel mit drei Transaktionen:



23

3.4.2 Serialisierbarkeit (10)

Präzedenzgraph (*precedence graph*)

- Idee:
 - stelle Reihenfolge der Konfliktoperationen durch Kanten in gerichtetem Graphen dar
- Konstruktion:
 - jede Transaktion ist ein Knoten
 - für jeden Konflikt zwischen zwei Transaktionen wird eine Kante zwischen den Transaktionsknoten gezeichnet. Die Kante geht von der früheren zur späteren Operation

Transaktion T1 Transaktion T2

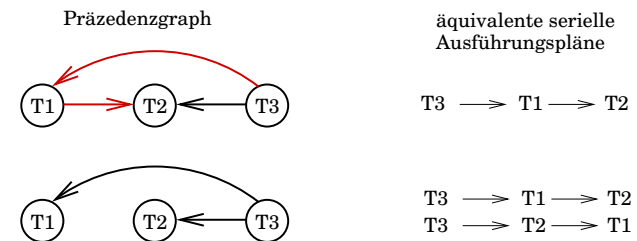


22

3.4.2 Serialisierbarkeit (12)

Anschauliche Bedeutung Präzedenzgraph

- Konfliktoperationen dürfen nicht vertauscht werden
=> Kante $T_i \rightarrow T_j$ bedeutet, dass Transaktion T_i im äquivalenten seriellen Ausführungsplan vor Transaktion T_j kommen muss
- Graph gibt also *Präzedenz* (Rangfolge) der Transaktionen im äquivalenten seriellen Schedule an



24

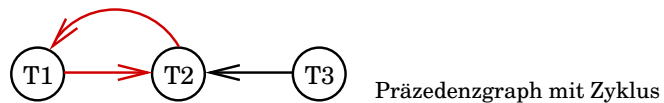
3.4.2 Serialisierbarkeit (13)

Frage:

- Wann kann Reihenfolge nicht angegeben werden?

Antwort:

- Wenn Präzedenzgraph Zyklen hat



Theorem

Ein Ausführungsplan ist (konflikt-) serialisierbar
 \Leftrightarrow Sein Präzedenzgraph hat keine Zyklen

25

3.4.2 Serialisierbarkeit (15)

Probleme bei Anwendung Serialisierbarkeitstest:

- Transaktionen beginnen und enden permanent
 \Rightarrow Wo beginnt und endet Ausführungsplan?
- Was, wenn Schedule nicht serialisierbar?
 \Rightarrow Rollback aller beteiligten Transaktionen

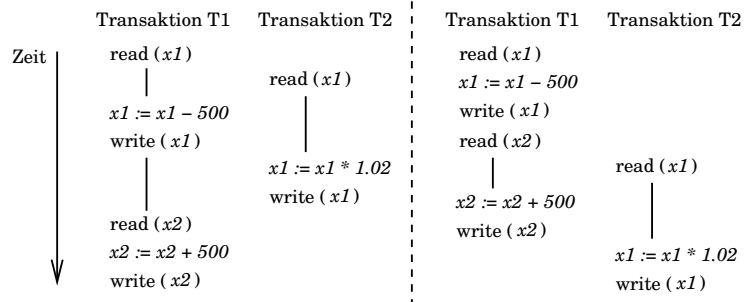
Bessere Ansätze für die Praxis:

- lasse nur serialisierbare Schedules zu
 einzelne Operation müssen dann ggf. warten
- treten Konflikte auf, dann setze nur wenige der beteiligten Transaktionen zurück

27

3.4.2 Serialisierbarkeit (14)

Anwendung auf Beispiel von Folie (5)



- Wie sehen die Präzedenzgraphen aus?
- Was sagt unser Theorem über Serialisierbarkeit?

26

3.4.2 Serialisierbarkeit (16)

Klassifikation Verfahren zur Transaktionsisolation:

- pessimistische Verfahren
 - verhindern von vorneherein nichtserialisierbare Schedules
 - Beispiel: Zwei-Phasen Sperrprotokoll
 - Nachteile:
 - Transaktionen müssen warten \Rightarrow geringere Parallelität
 - Möglichkeit von Deadlocks (gegenseitiges Warten aufeinander)
- optimistische Verfahren
 - lassen zunächst beliebige Schedules zu, beim Auftreten von Konflikten wird eine Transaktion zurückgesetzt
 - Beispiel: Multi Version Concurrency Control (MVCC)
 - Nachteil:
 - bei Abbruch wegen Konflikt muss ganze Transaktion wiederholt werden

28

3.4.3 Sperrverfahren (1)

Binäre Sperren

- jedes Objekt hat zwei mögliche Zustände: *gesperrt*, *ungesperrt*
- zwei weitere elementare Operationen
 - ▷ *lock(X)* setzt Zustand von Objekt *X* auf *gesperrt*
 - ▷ *unlock(X)* setzt Zustand von Objekt *X* auf *ungesperrt*
- vor jedem Zugriff auf Objekt *X* muss *lock(X)* erfolgen
- in Transaktion muss auf *lock(X)* irgendwann *unlock(X)* folgen

Was macht *lock(X)*, wenn *X* schon gesperrt ist?

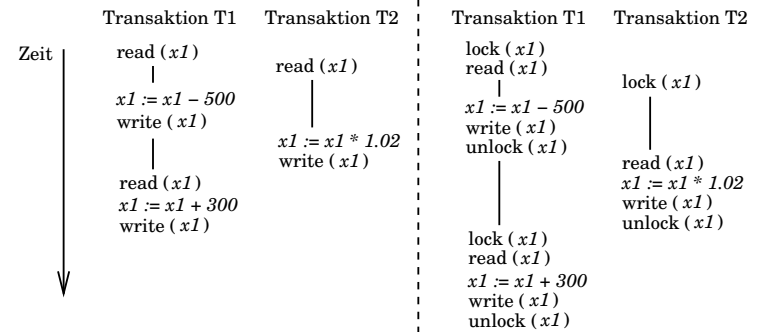
- *lock* wartet bis *X* wieder freigegeben ist
- Transaktionen die auf Freigabe von *X* warten, werden in eine Warteschlange eingereiht

29

3.4.3 Sperrverfahren (3)

Beispiel b)

Locking bewirkt nicht immer Serialisierbarkeit

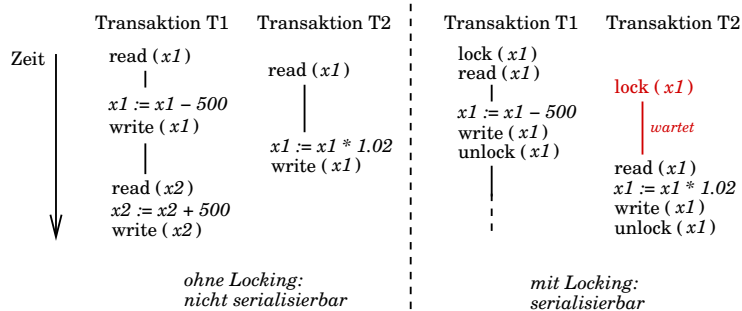


31

3.4.3 Sperrverfahren (2)

Beispiel a)

nicht serialisierbarer Schedule von Folie (5) wird durch binären Sperrmechanismus serialisierbar



30

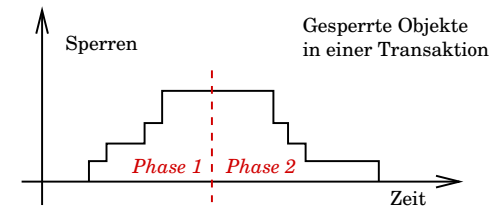
3.4.3 Sperrverfahren (4)

Zwei-Phasen Sperrprotokoll (Two-Phase Locking, 2PL)

- jede Transaktion führt alle *lock*'s vor allen *unlock*'s aus.
M.a.W. nach dem ersten *unlock* darf kein *lock* mehr folgen

- Was sind die "zwei Phasen"?

Phase 1: Anforderung von Sperren
Phase 2: Freigabe von Sperren



32

3.4.3 Sperrverfahren (5)

Theorem

- Wenn sich jede Transaktion an das 2PL hält, ist der resultierende Schedule konfliktserialisierbar.

Anmerkung

- Umkehrung gilt nicht, d.h. es gibt konfliktserialisierbare Schedules, die sich nicht an das 2PL halten (siehe Beispiel a)

Anwendung Theorem auf Beispiel b)

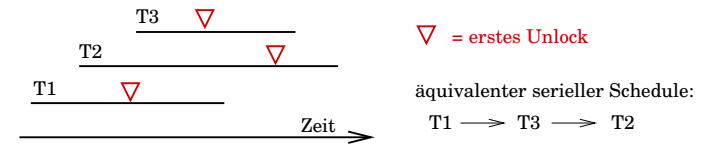
- Sperren in Beispiel b) führen nicht zu serialisierbarem Schedule => 2PL muss irgendwo verletzt werden (Warum?)
- Wenn Beispiel b) auf 2PL umgestellt wird, müsste der resultierende Schedule serialisierbar sein

33

3.4.3 Sperrverfahren (7)

Beweis des 2PL Theorems

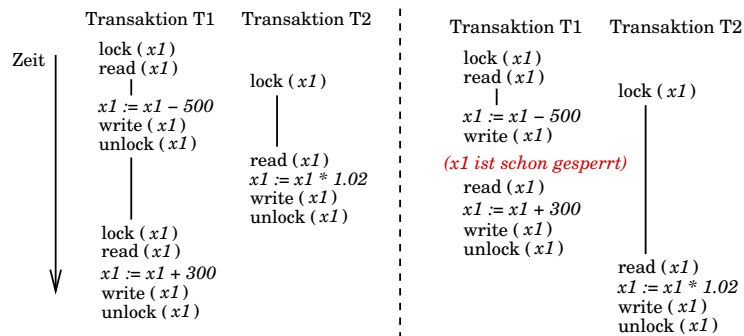
- konstruiere äquivalenten seriellen Schedule zu gegebenem Schedule, der sich an 2PL hält
- im seriellen Schedule treten die Transaktionen in der Reihenfolge ihrer *ersten Unlockoperation* auf



- Beweis durch vollst. Induktion über Anzahl Transaktionen: Garcia-Molina, Ullman, Widom: *Database Systems - The Complete Book*. Prentice Hall (2002), Abschnitt 18.3.4

35

3.4.3 Sperrverfahren (6)



- links: T1 verletzt 2PL (Wo?)
- rechts: Schedule wird serialisierbar durch 2PL

34

3.4.3 Sperrverfahren (8)

Nachteil binärer Sperren

- auch die konfliktfreien *read/read* Operationen sperren einander

Lösung: mehrere Sperrmodi

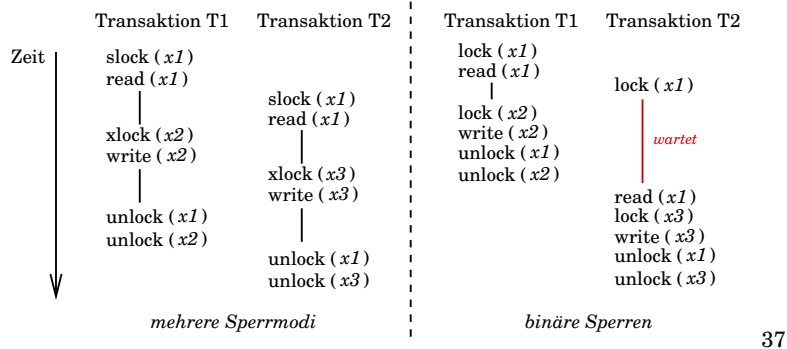
- read lock, shared lock*
lässt zu dass andere Transaktionen auch lesen (d.h. ebenfalls *read lock* anfordern), aber nicht schreiben
- write lock, exclusive lock*
lässt keine Zugriffe anderer Transaktionen zu
- führt zu drei Objektzuständen und damit zu drei elementaren Sperr-Operationen: *slock, xlock, unlock*

Kompatibilitätsmatrix der Sperroperationen		angeforderte Sperre	S	X
		gehaltene Sperre	S	X
	S	Ja	Nein	
	X	Nein	Nein	

36

3.4.3 Sperrverfahren (9)

Beispiel für Erhöhung Nebenläufigkeit durch mehrere Lockmodi:



37

3.4.3 Sperrverfahren (11)

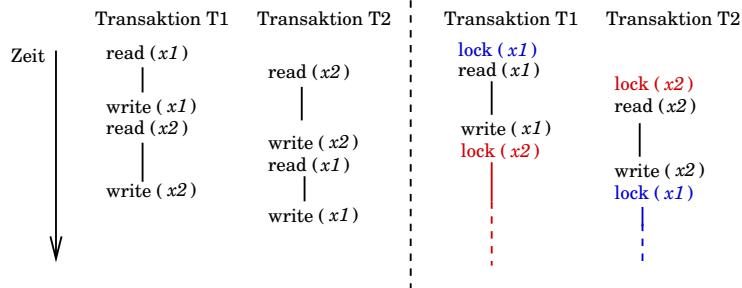
Ansätze zur Deadlockbehandlung

- **Deadlock Erkennung**
 - Suche nach Zyklen im Wartegraphen (=> Deadlock)
 - Setze eine am Deadlock beteiligte Transaktion zurück
- **Timeouts**
 - definiere obere Grenze für Wartezeit; wenn überschritten, wird Transaktion abgebrochen
 - einfach zu realisieren => in vielen DBS'en implementiert
- **verhindernde Protokolle**
 - Abbruch oder Neustart von Transaktionen, wenn ein *lock()* zu Verklemmung führen wird
 - Nachteil: brechen oft Transaktionen ab, die nie zu Verklemmung geführt hätten

39

3.4.3 Sperrverfahren (10)

Grundsätzliches Problem bei Sperrverfahren:



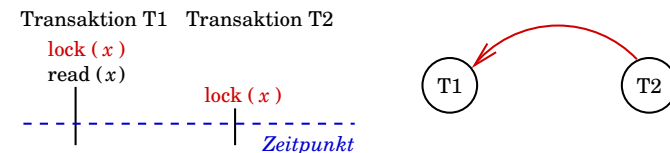
- Transaktionen blockieren sich gegenseitig: jede wartet auf Freigabe einer Sperre der anderen Transaktion
- Phänomen heißt *Deadlock* (Verklemmung)

38

3.4.3 Sperrverfahren (12)

Der Wartegraph (*wait-for graph*)

- gerichteter Graph mit Transaktionen als Knoten
- Kante $T_i \rightarrow T_j$ bedeutet, dass Transaktion T_i versucht Objekt zu sperren, das schon von Transaktion T_j gesperrt ist



Frage:

- Woran erkennt man Deadlock?

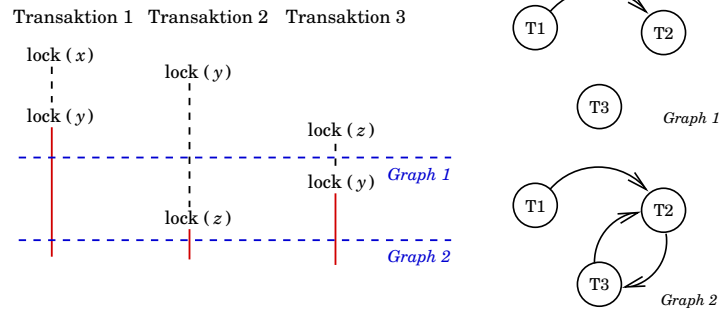
Antwort:

- Zyklus im Wartegraphen.

40

3.4.3 Sperrverfahren (13)

Beispiel mit drei Transaktionen:



o Zyklus in Wartegraph 2 zeigt Verklemmung

41

3.4.3 Sperrverfahren (15)

Protokolle zur Deadlock-Vermeidung

- o für Beispiele siehe Elmasri, Navathe: Kap. 20.1.3 und Literaturhinweise am Ende Kap. 20
- o ein einfaches Protokoll ist z.B. "Cautious Waiting":
 - o lock(x) darf nur warten, wenn x nicht von einer selber wartenden Transaktion gesperrt ist
 - o andernfalls wirft lock(x) einen Fehler
- o werden in der Praxis selten eingesetzt
 - o z.B. führt "Cautious Waiting" auch zu Abbrüchen in Nicht-Verklemmungssituationen => ggf. Schaden größer als Nutzen

43

3.4.3 Sperrverfahren (14)

Probleme bei der Deadlockerkennung:

- o **Opferauswahl**
 - o welche Transaktion soll abgebrochen werden?
 - o wünschenswert:
 - o wenig fortgeschrittene Transaktionen sollten bevorzugt abgebrochen werden
 - o Opferauswahl sollte nicht unfair sein (mehrmals dieselbe Transaktion treffen)
- o **Wann prüfen?**
 - o naheliegender Ansatz (z.B. in PostgreSQL realisiert): starte Algorithmus wenn eine Transaktion bestimmte Zeit wartet

Primitive Alternative: Timeouts

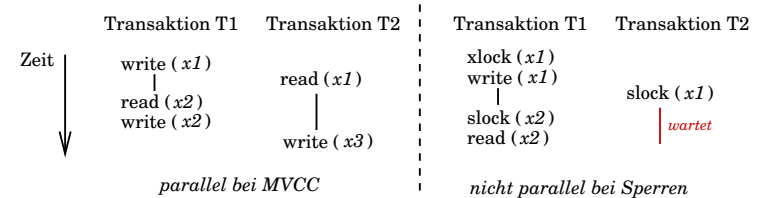
- o setze Transaktion zurück, die länger als *Timeout* wartet
- o schlägt auch zu, wenn kein Deadlock vorhanden

42

3.4.4 MVCC (1)

Multi Version Concurrency Control

- o **Idee:**
 - o write(x) überschreibt nicht, sondern erzeugt neue Version von x
 - o Transaktion arbeitet mit Snapshot der committed Daten zu geeignetem Zeitpunkt (z.B. erstes DML-Statement)
- o **Vorteil:**
 - o höhere Parallelität von Transaktionen



44

3.4.4 MVCC (2)

Speichern mehrerer Objektversionen:

- Transaktionen bekommen fortlaufende ID (*Zeitstempel*)
- jede Objektversion bekommt zwei Zeitstempel:
 - *read_ts*: Zeitstempel der letzten lesenden Transaktion
 - *write_ts*: Zeitstempel der letzten schreibenden Transaktion
- beim Lesen Objektversion wird *read_ts* entsprechend angepasst
- Daten der Objektversionen werden nie verändert; statt dessen wird eine neue Version des Objekts mit $read_ts = write_ts$ angelegt

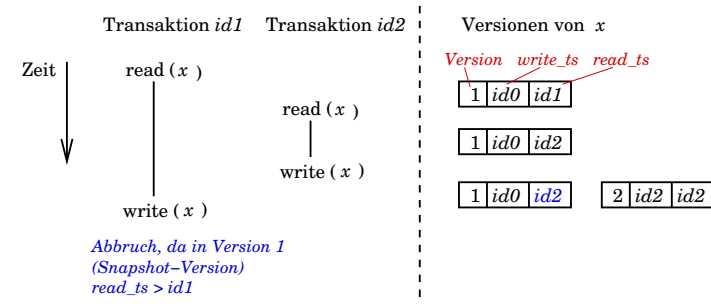
Bemerkung:

- je nach MVCC Verfahren werden ggf. auch andere Zeitstempel bei den einzelnen Objektversionen protokolliert

45

3.4.4 MVCC (4)

Was passiert bei nicht serialisierbarem Schedule?



problematische Transaktionen werden automatisch abgebrochen

47

3.4.4 MVCC (3)

Regeln für Serialisierbarkeit:

- *read(x)* von Transaktion mit ID *trans_ts*
 - liefert Version von *x* mit maximalem $write_ts \leq trans_ts$
 - *read_ts* der Version wird auf $\max\{trans_ts, read_ts\}$ gesetzt
- *write(x)* von Transaktion mit ID *trans_ts*
 - untersuche Version von *x* mit maximalem $write_ts \leq trans_ts$: hat sie $read_ts > trans_ts$, wird Transaktion abgebrochen
Denn: dann wurde Snapshot von weiterer, später gestarteter Transaktion gelesen (=> möglicher Konflikt)
 - andernfalls wird neue Version von *x* angelegt

Bemerkung:

- bei Sperrverfahren konnte auch ein *read* zum Abbruch führen (warum?); bei MVCC klappt *read* dagegen immer

46

3.4.5 Concurrency in SQL (1)

PostgreSQL und Oracle unterstützen die Isolationsgrade *read committed* und *serializable*

- Auswahl
 - für Transaktion: *set transaction isolation level <level>;*
 - für Session: *set default_transaction_isolation to <level>;*
- unterschiedliches Verhalten beim Update
 - Im *read committed* Modus wird Änderung auf Basis zuletzt committeter Werte durchgeführt.
 - Im *serializable* Modus kann es zum Abbruch mit "can't serialize" Fehler kommen

read committed Modus erfordert explizites Locking:

- Tupel-Ebene: *select ... from ... for update;*
- Tabellen-Ebene: *lock table;*

48

3.4.5 Concurrency in SQL (2)

Zwei mögliche Programmierstrategien:

- arbeite im *read committed* Modus und sperre alle zu ändernden Sätze mit *select for update*

```
BEGIN;  
SELECT betrag FROM konto WHERE ... FOR UPDATE;  
UPDATE konto SET betrag = '...' WHERE ...;  
COMMIT;
```

- arbeite im *serializable* Modus und fange bei Abbrüchen wieder von vorne an (Retry Loop)

```
BEGIN;  
SELECT betrag FROM konto WHERE ...;  
UPDATE konto SET betrag = '...' WHERE ...;  
if (no error) COMMIT;  
else          ROLLBACK;
```

49

Rückblick

Was haben wir gelernt?

- welche Features relationales DBS bereitstellt
 - Datenobjekte und Integritätsbedingungen
 - Transaktionsverwaltung
- wie man Daten "vernünftig" modelliert
 - Qualitätsmerkmal "Normalformen"
 - Modellierung über das anschauliche ER-Modell
- wie man DBS'e programmiert
 - SQL
 - clientseitige Programmierschnittstellen
 - serverseitige Programmierung

51

3.4.5 Concurrency in SQL (3)

Retry-Loop tatsächlich in *beiden* Fällen nötig:

- bei *select for update* können Lese- und Schreiboperationen fehlschlagen wegen Deadlock
- bei *serializable* können nur Schreiboperationen fehlschlagen

Was ist besser?

- *select for update* (pessimistisches Verfahren) wenn viele Konflikte, denn *serializable* führt dann häufig zum Abbruch
- *serializable* (optimistisches Verfahren) wenn wenig Konflikte

Gute Einführung:

http://conferences.oreilynet.com/cs/os2002/view/e_sess/2681

50

Ausblick

Was könnten wir noch lernen?

- Objektorientierte Ansätze
 - objektorientierte Datenbanken
 - objekt-relationale Datenbanken
- verteilte Datenbanken
- Transaktionen
 - Error Recovery
- Implementierung von DBS'en
 - Speicherstrukturen und Indizes
 - Abfrageoptimierung
 - allgemeine Architektur eines DBS

52