Datenbanksysteme

Kap 9: Transaktionen

Was ist eine Transaktion?

- 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, ...)

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:

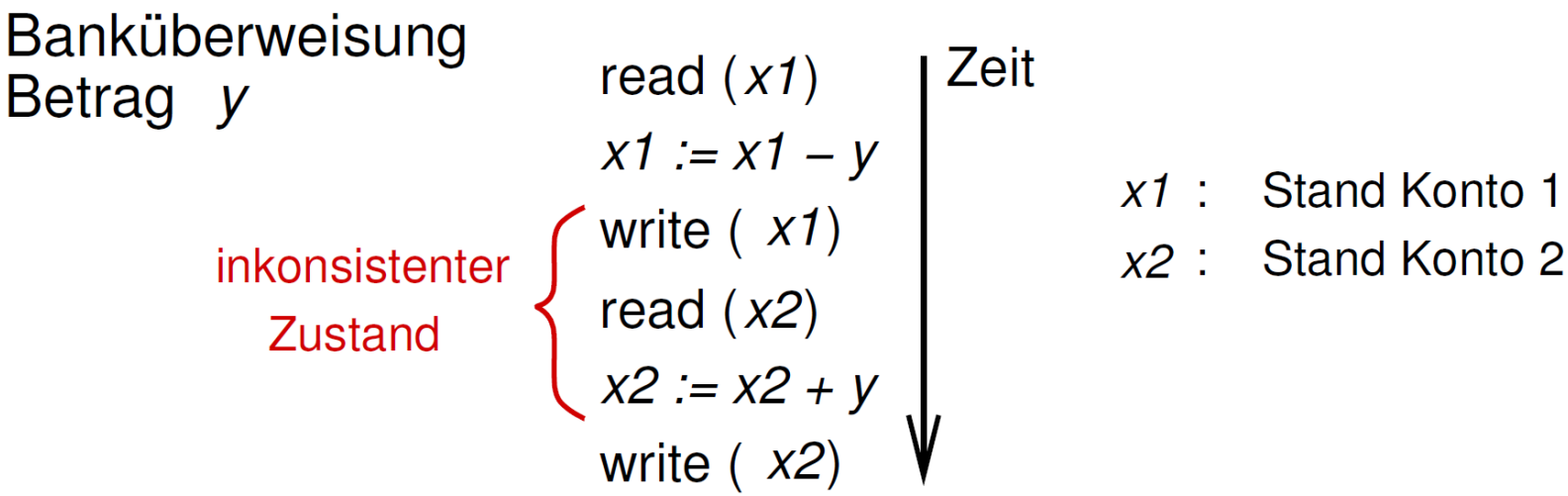
```
update produkt set preis=preis+1
where pnr='P1';
```

Transaktionsverwaltung

Themen in diesem Abschnitt:

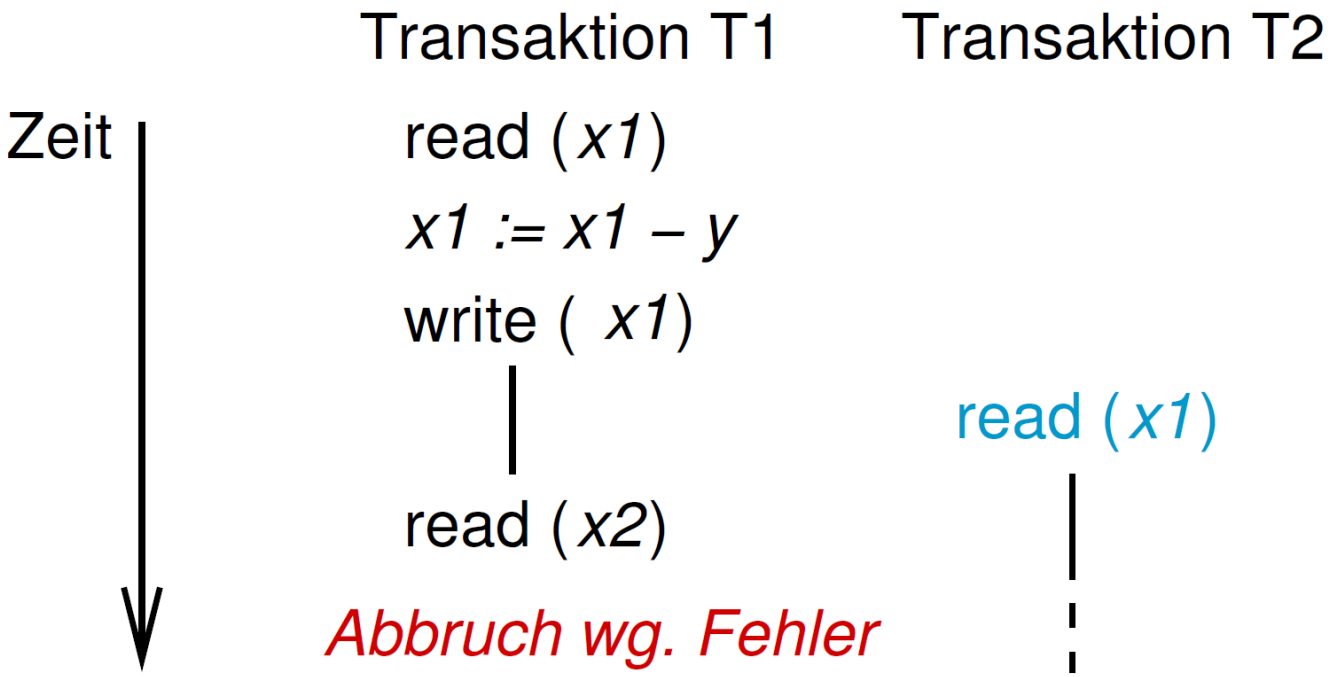
- Probleme bei der Transaktionsverarbeitung
  - *Error Recovery*
    Fehlerbehandlung schon beim Einbenutzerbetrieb wichtig
  - *Concurrency Control*
    Nebenläufigkeit von Transaktionen mehrerer Benutzer
  - *Backup/Restore*
    Online Backup bei laufenden Transaktionen
- Anforderungen an Transaktionsverarbeitung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

Error Recovery



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

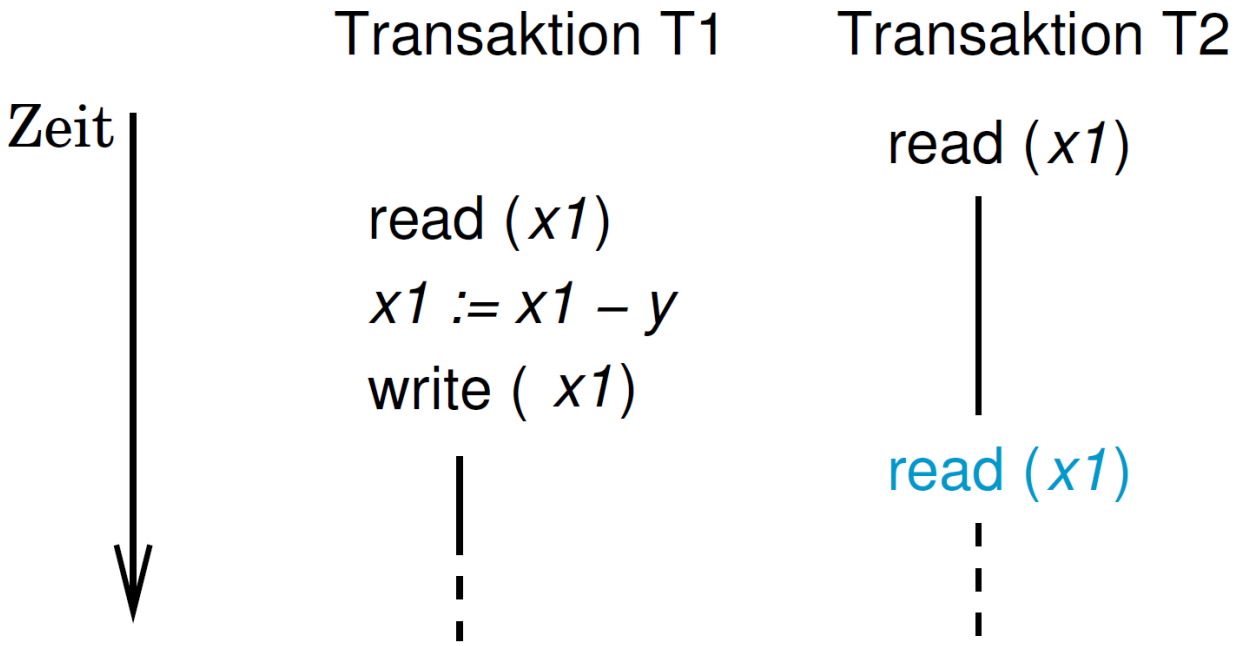
Concurrency Control – Dirty Read



- T2 liest Daten, die nie committed werden (dirty data)

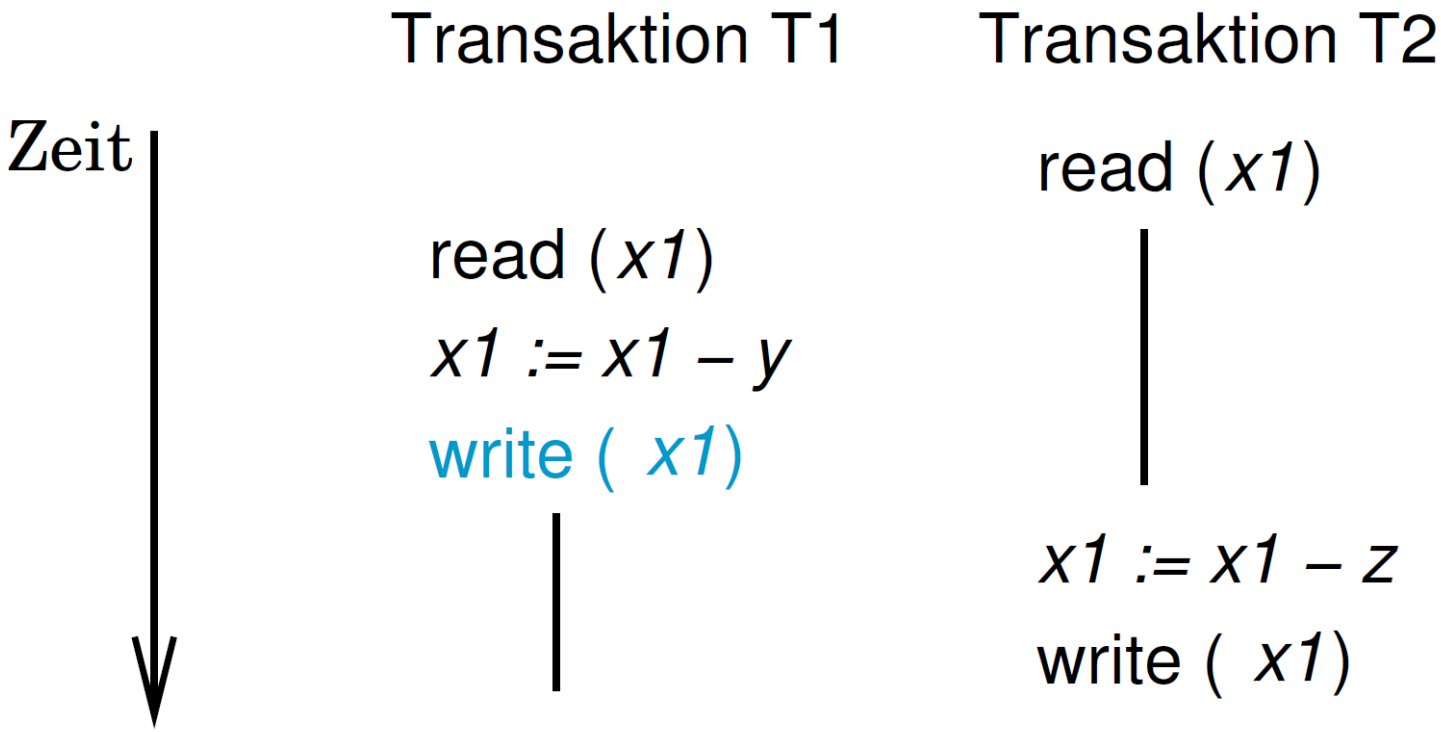
  →Überweisung T2 führt zu falschem Kontostand
- Phänomen wird als *Dirty Read* bezeichnet

Concurrency Control – Nonrepeatable Read



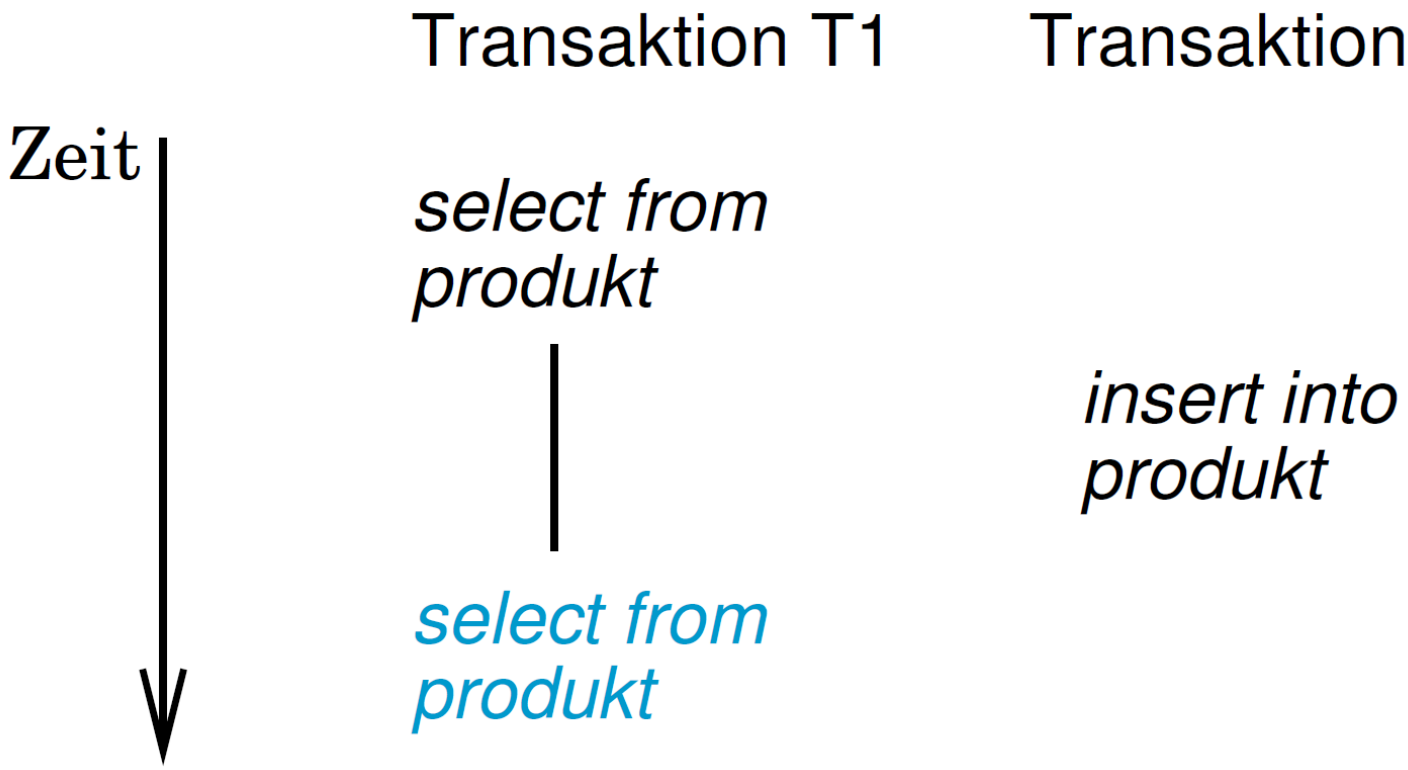
- T2 liest mehrmals hintereinander verschiedene Werte desselben Datensatzes, ohne ihn zu verändert zu haben
- Phänomen wird als *Nonrepeatable Read* bezeichnet

Concurrency Control – Lost Update



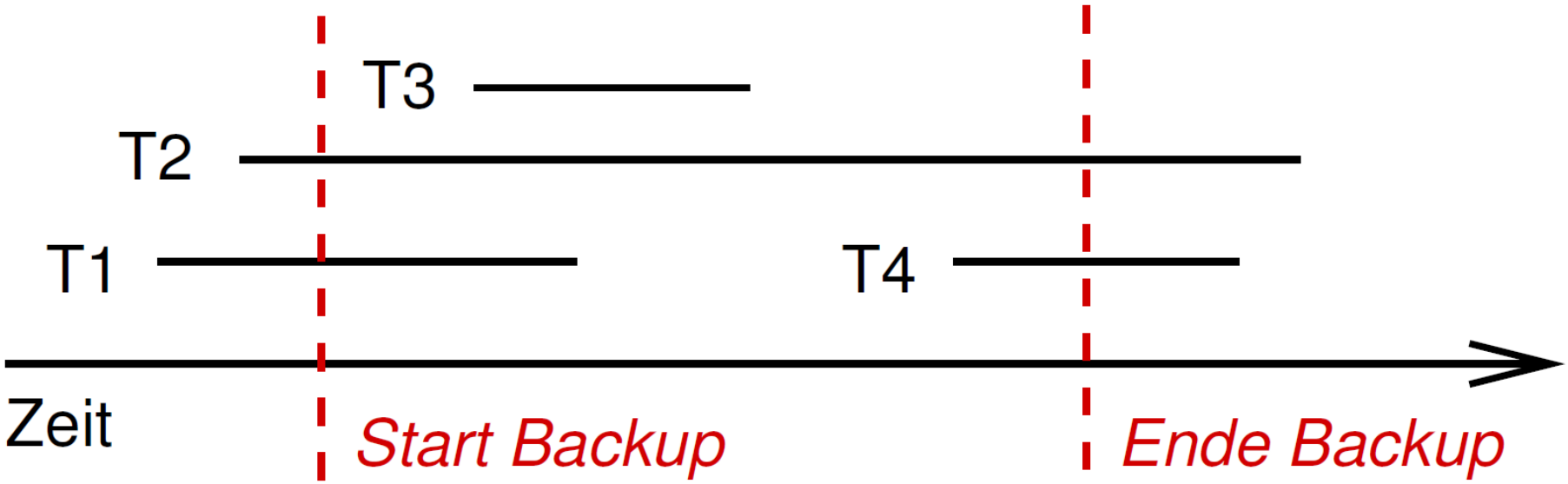
- Update von T1 geht verloren (*Lost Update*)
  →falscher Kontostand nach Abschluss beider Transaktionen
- Tritt in Tateinheit mit *Nonrepeatable Read* auf (Warum?)

Concurrency Control - Phantomtupel



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

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

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

Isolationsgrade

- 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 uncommited*, *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>;`

Isolationsgrade

- 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
```

Serialisierbarkeit

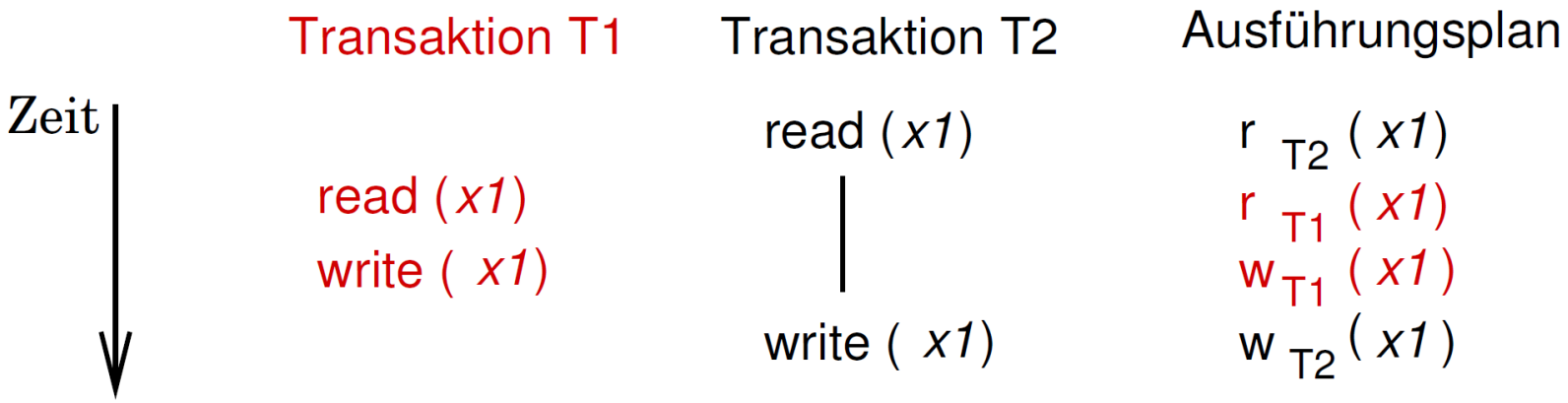
- Ziel: Isolation von Transaktionen
  - Transaktionen sollen nichts voneinander merken

Fragestellungen:

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

Ausführungsplan

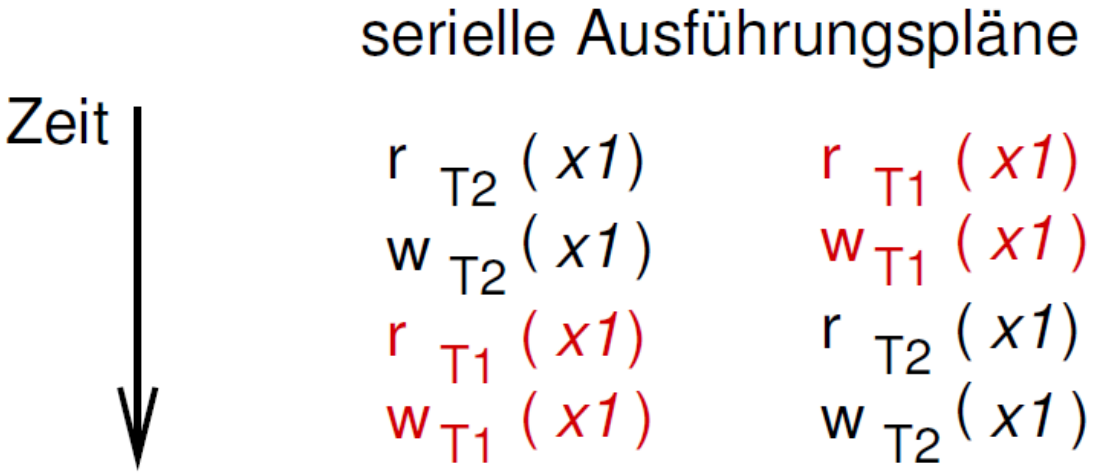
- Ausführungsplan (*Schedule*) =
  zeitliche Abfolge elementarer Operationen



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

Serielle Ausführung

- Ideale Isolation bei serieller Ausführung:
  - Jede Transaktion hat Datenbank für sich alleine



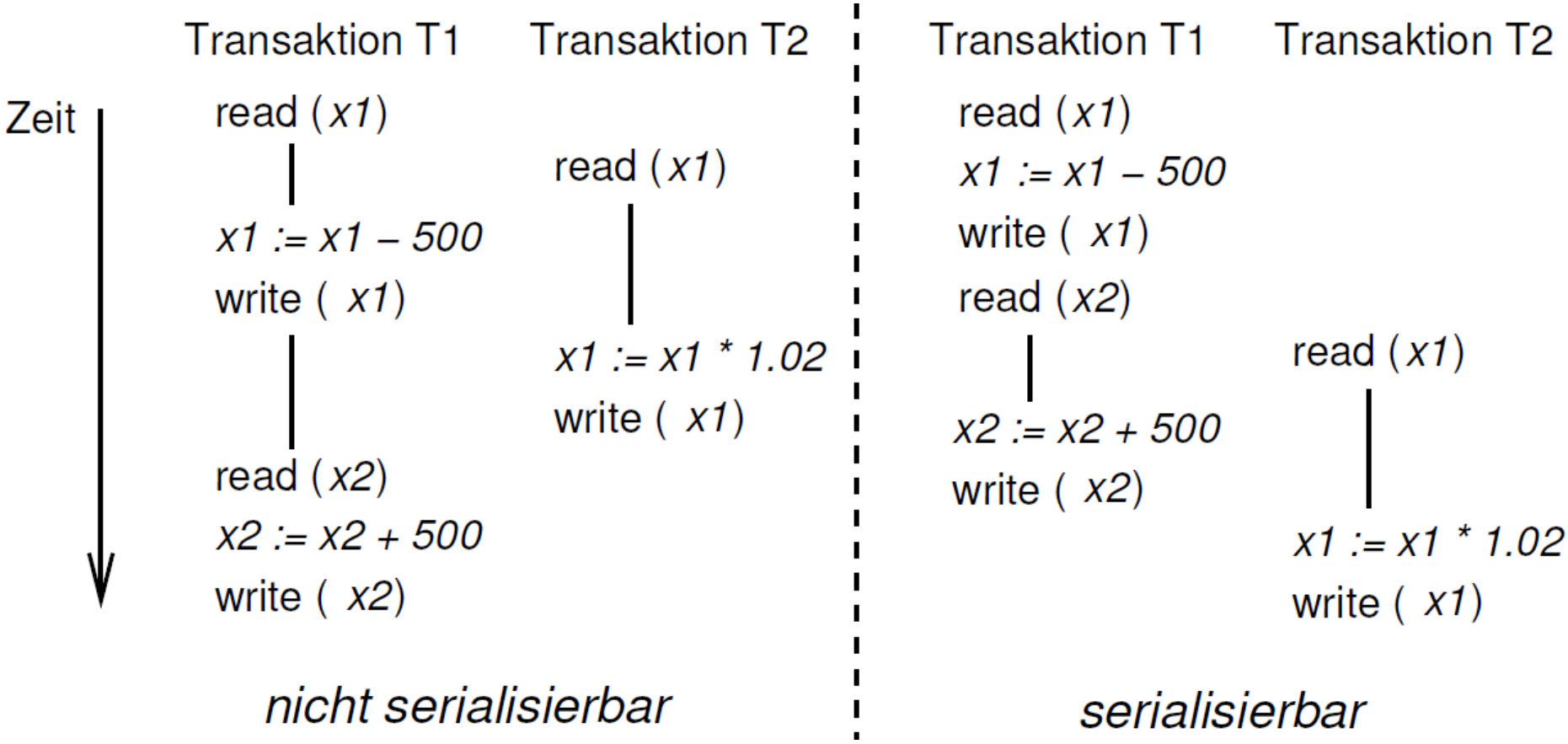
- Nachteil:
  - Transaktionen müssen aufeinander warten
    - →keine Nebenläufigkeit
    - →geringer Durchsatz an Transaktionen

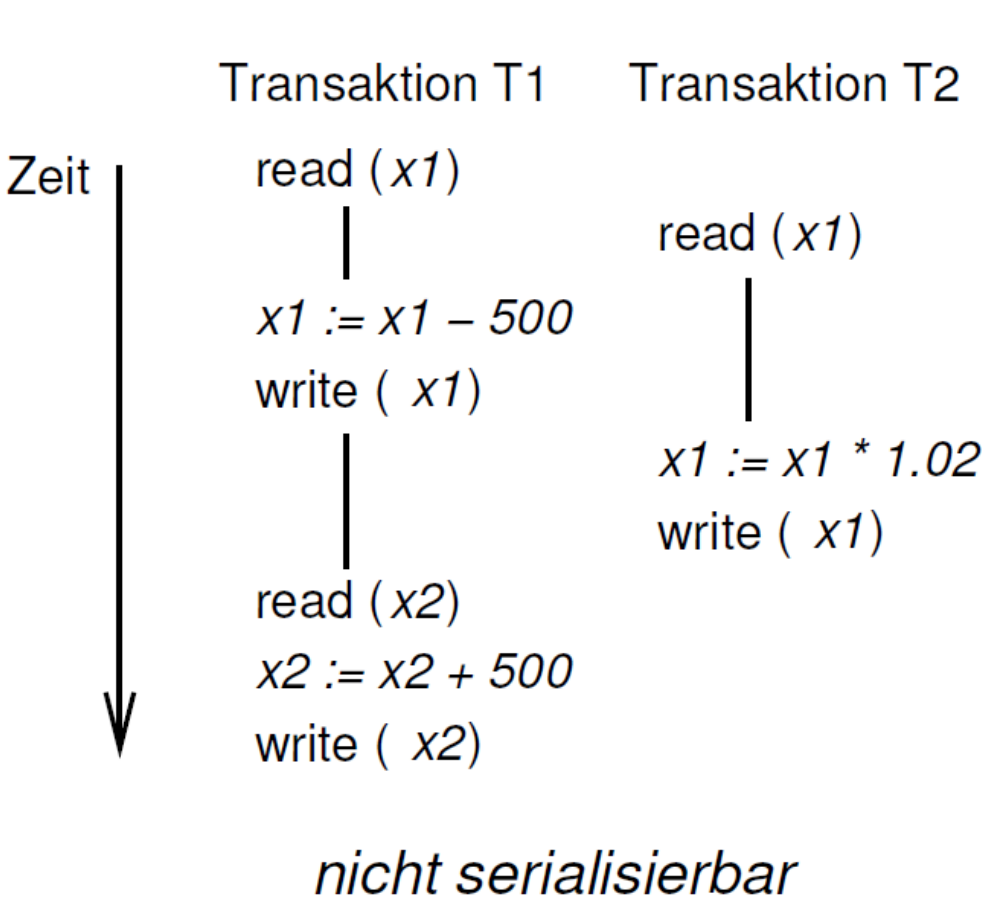
Serialisierbarkeit

- 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*.
- Bemerkungen
  - Serialisierbarkeit ist abhängig von Definition der „Äquivalenz“ von Ausführungsplänen (siehe weiter unten)
  - Verschiedene Definitionen der Schedule-Äquivalenz möglich
  - Hier: Schedules haben dieselben Auswirkungen in DB

Serialisierbarkeit

- Beispiele für (nicht) serialisierbaren Schedule
  - Vergleich der Resultate für x1 und x2 mit den Werten der seriellen Ausführungspläne → Übung



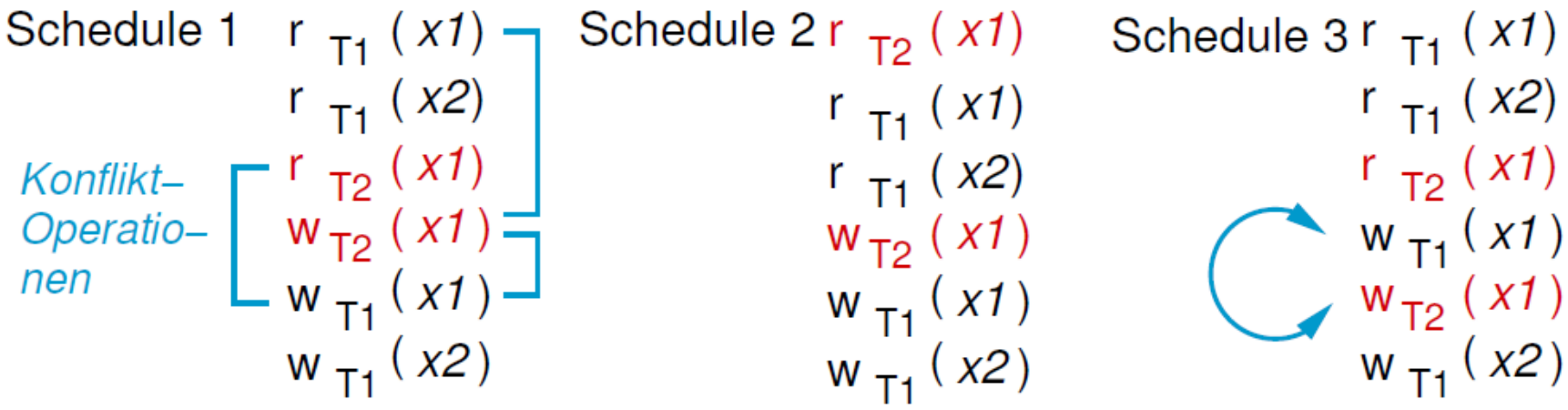

Transaktion T1
Transaktion T2
Zeit
read (x1)
read (x1)
x1 := x1 – 500
write (x1)
x1 := x1 * 1.02
write (x1)
read (x2)
x2 := x2 + 500
write (x2)
nicht serialisierbar

Konfliktäquivalenz

- Suche hinreichendes Kriterium für Äquivalenz von Ausführungsplänen
- Frage: welche Operationen zweier Transaktionen 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 (siehe nicht serialisierbaren Schedule im obigen Beispiel)
- Solche Operationen stehen im *Konflikt* zueinander

Konfliktäquivalenz

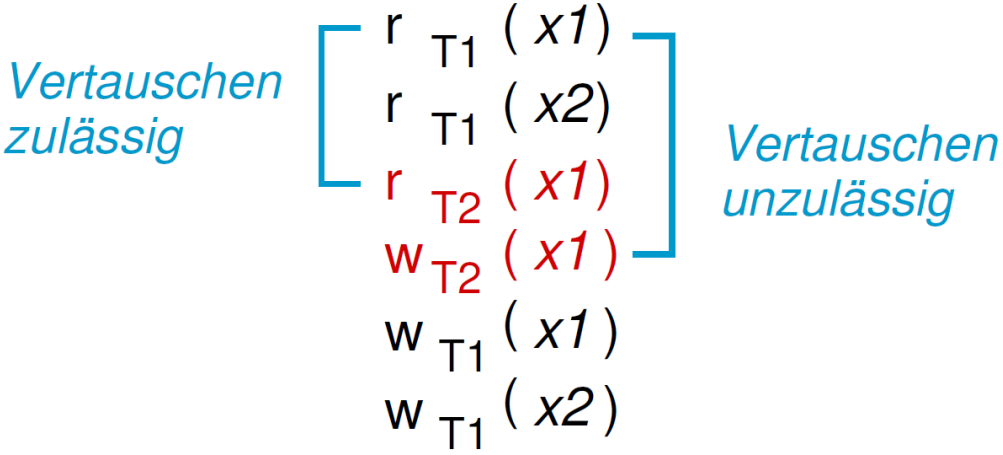
- Zwei Schedules heißen *konfliktäquivalent*, wenn die Reihenfolge in Konflikt stehender Operationen in beiden Schedules gleich ist.
- Beispiel



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

Serialisierbarkeit

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



  - Übung: Kriterium überprüfen an vorhergehenden Schedules

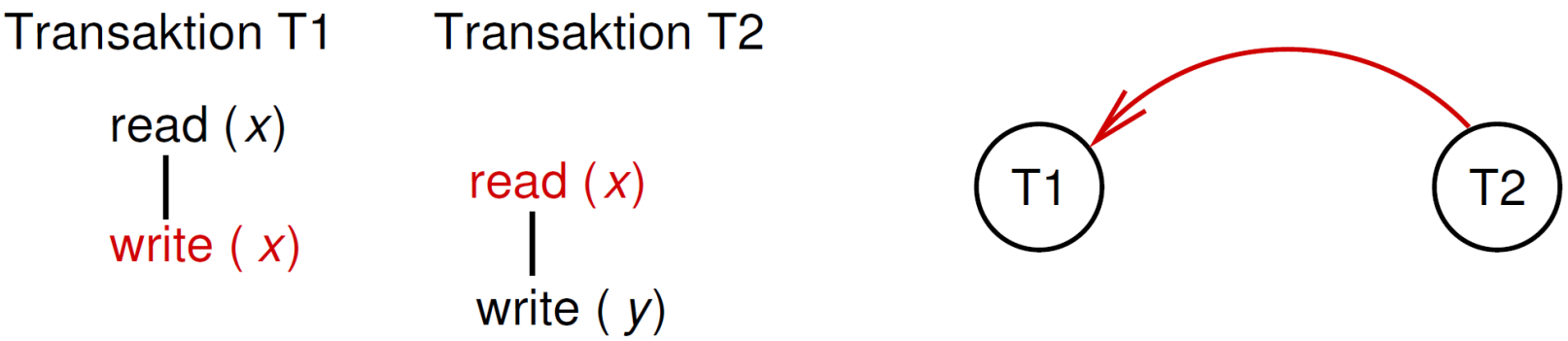
Serialisierbarkeit

- 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"

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



Beispiel mit drei Transaktionen

Transaktion 1

read (*x*)
write (*x*)

read (*y*)
write (*y*)

Transaktion 2

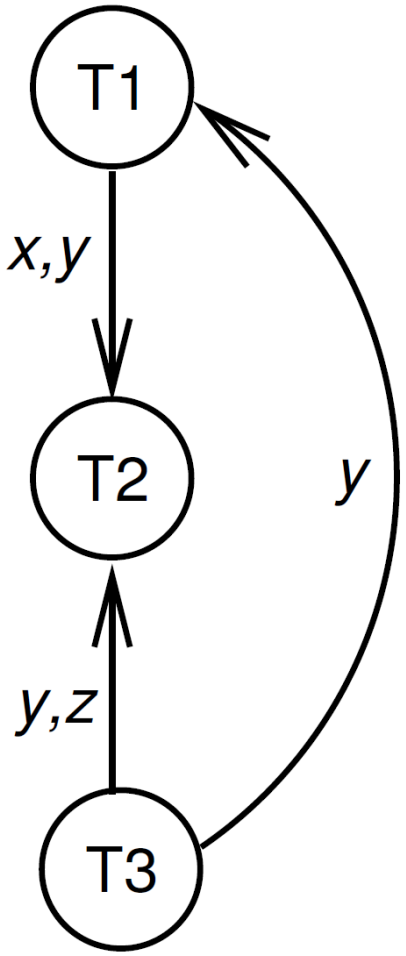
read (*z*)

read (*y*)
write (*y*)
read (*x*)
write (*x*)

Transaktion 3

read (*y*)
read (*z*)

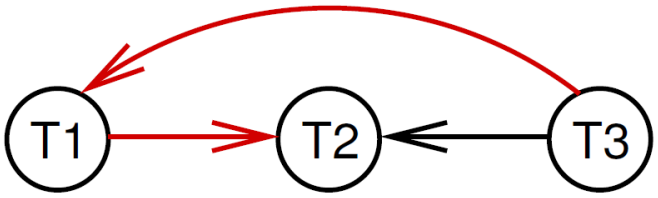
write (*y*)
write (*z*)

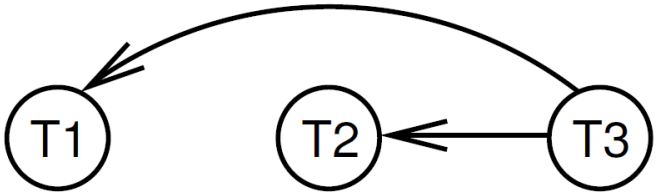


Interpretation des Präzedenzgraphen

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

Präzedenzgraph





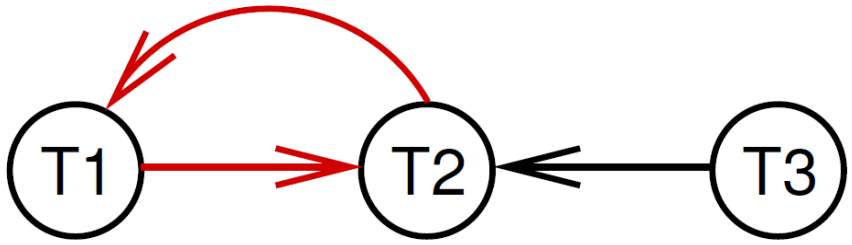
äquivalente serielle Ausführungspläne

T3 ⟶ T1 ⟶ T2

T3 ⟶ T1 ⟶ T2
T3 ⟶ T2 ⟶ T1

Serialisierbarkeit

- Frage:
  - Wann kann Reihenfolge nicht angegeben werden?
- Antwort:
  - Wenn Präzedenzgraph Zyklen hat

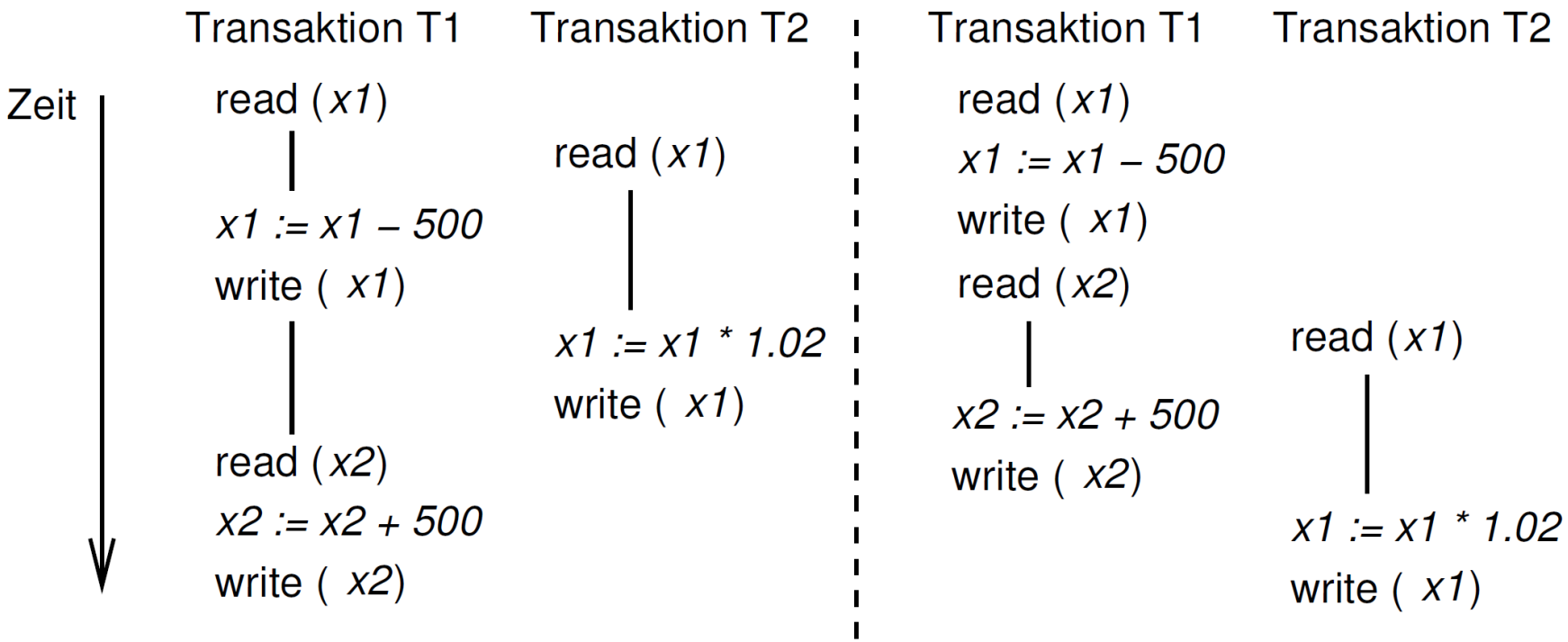


Präzedenzgraph mit Zyklus

- Theorem
  - Ein Ausführungsplan ist (konflikt-)serialisierbar
    ⇔ sein Präzedenzgraph hat keine Zyklen

Anwendung auf vorheriges Beispiel

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



Serialisierbarkeit

- Probleme bei Anwendung Serialisierbarkeitstest:
  - Transaktionen beginnen und enden permanent
  - Wo beginnt und endet Ausführungsplan?
- Was, wenn Schedule nicht serialisierbar?
  - Rollback aller beteiligten Transaktionen

- Bessere Ansätze für die Praxis:
  - Lasse nur serialisierbare Schedules zu; einzelne Operationen müssen dann ggf. warten
  - Treten Konflikte auf, dann setze nur wenige der beteiligten Transaktionen zurück

Klassifikation Verfahren zur Transaktionsisolation

- Pessimistische Verfahren
  - verhindern von vorneherein nichtserialisierbare Schedules
  - Beispiel: *Zwei-Phasen Sperrprotokoll*
  - Nachteile:
    - Transaktionen müssen warten → geringere Parallelität
    - Möglichkeit von Deadlocks (gegenseitiges Warten aufeinander)

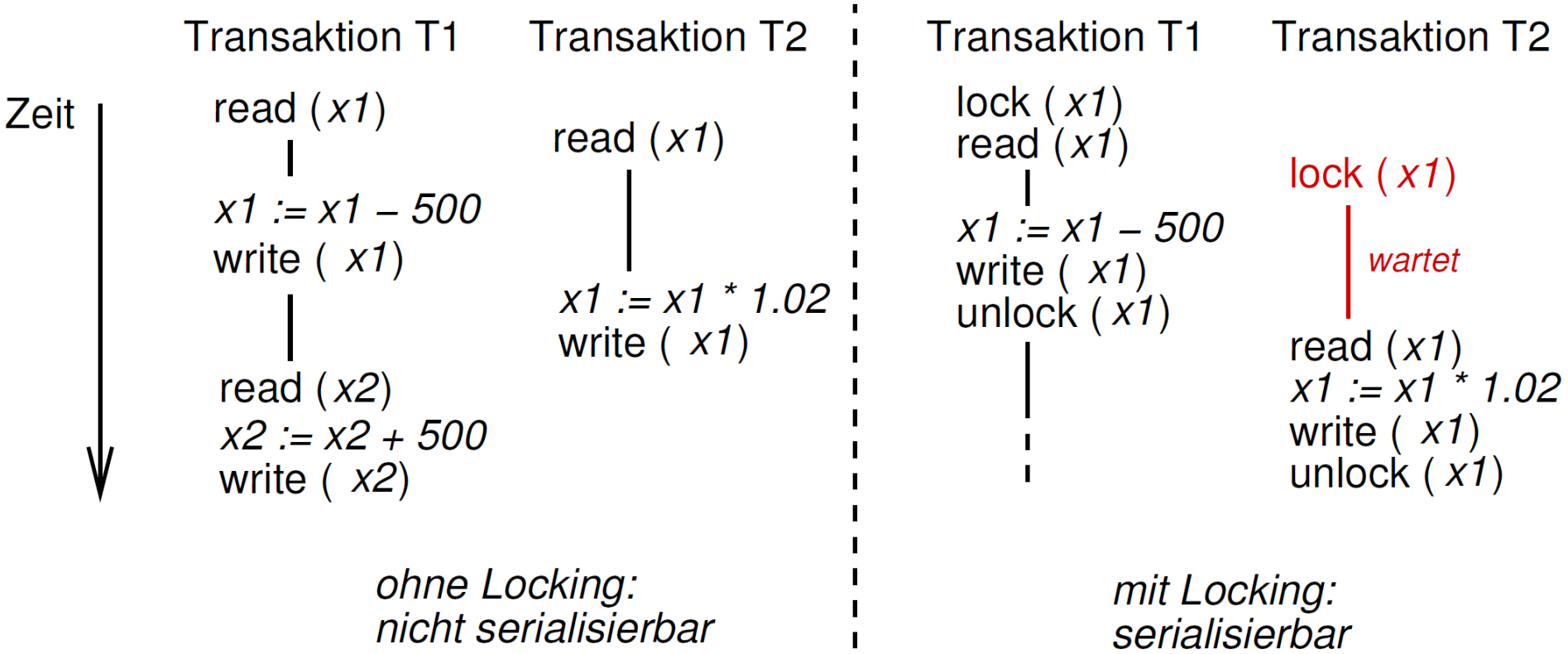
- Optimistische Verfahren
  - lassen zunächst beliebige Schedules zu, beim Auftreten von Konflikten wird eine Transaktion zurückgesetzt
  - Beispiel: *Multi Version Concurrency Control (MVCC)*
  - Nachteil:
    - bei Abbruch wegen Konflikt muss ganze Transaktion wiederholt werden

Sperrverfahren

- 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

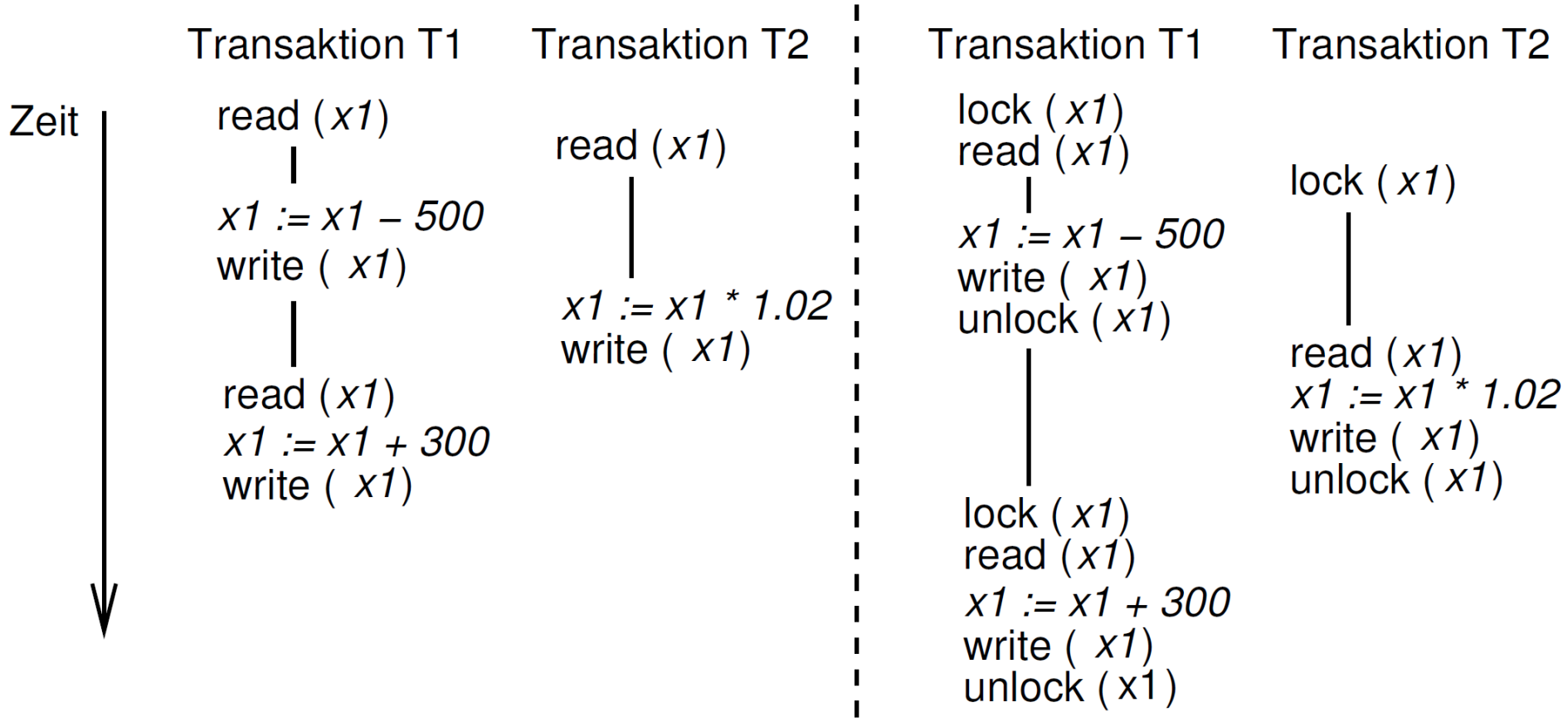
Beispiel 1

- nicht serialisierbarer Schedule wird durch binären Sperrmechanismus serialisierbar



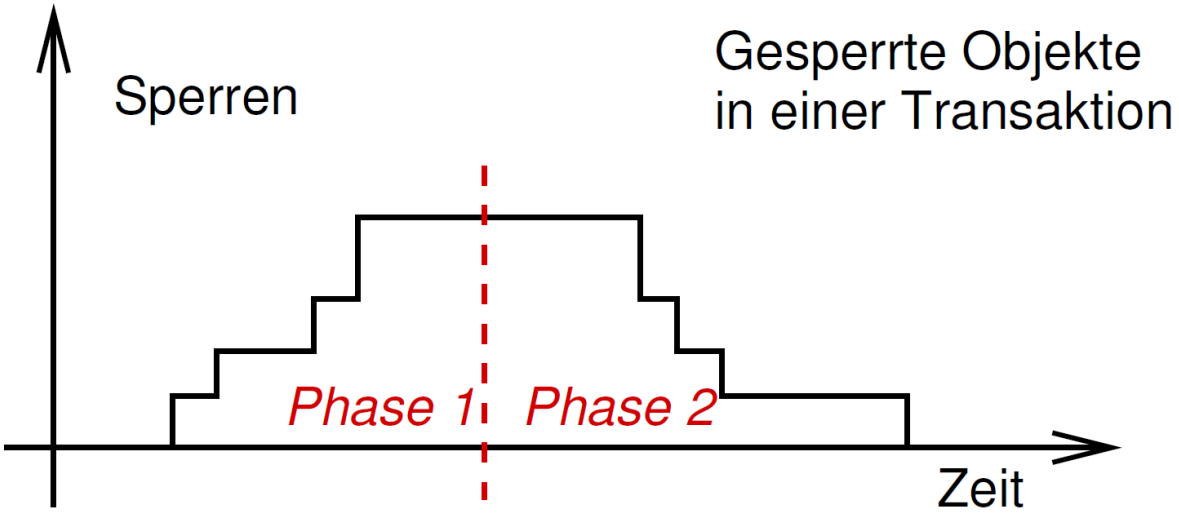
Beispiel 2

- Locking bewirkt nicht immer Serialisierbarkeit



Zwei-Phasen Sperrprotokoll

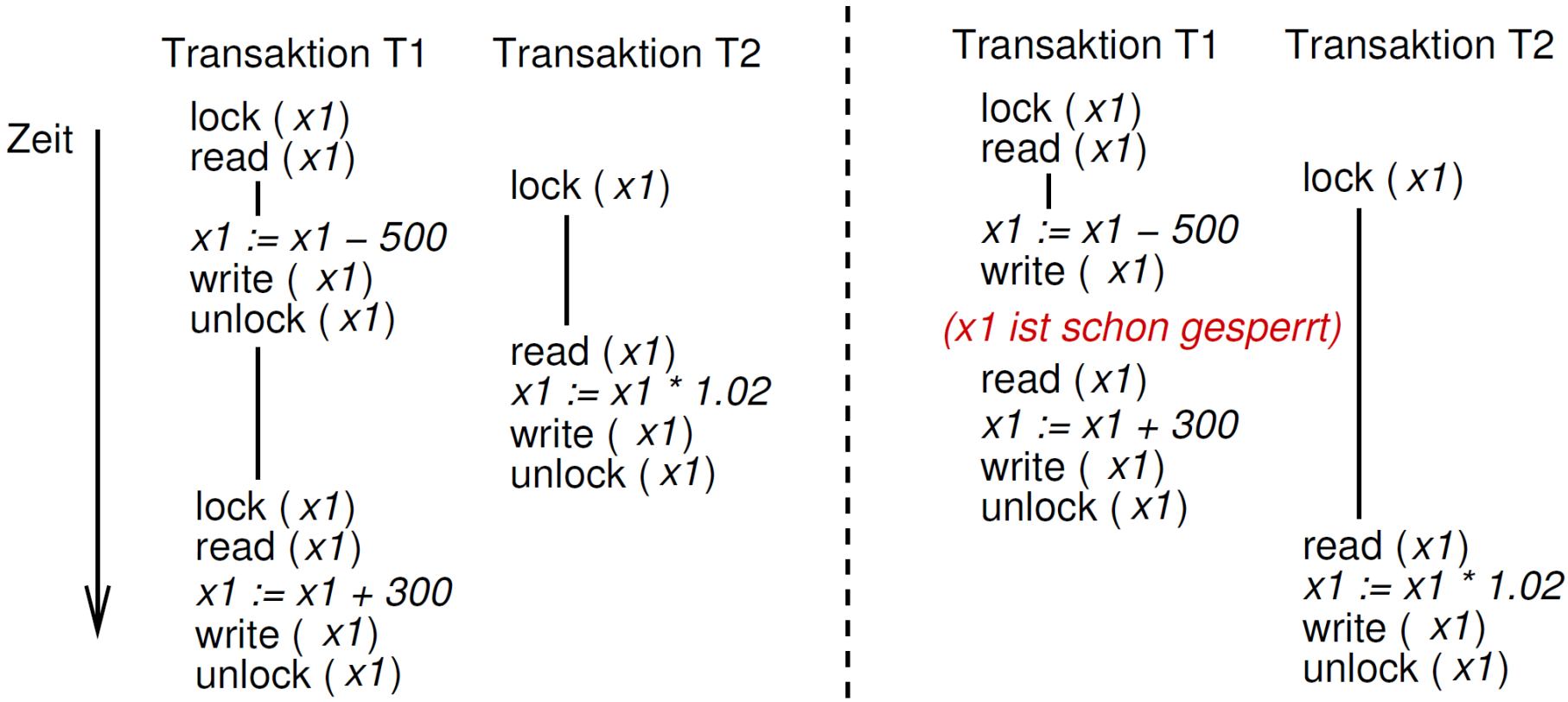
- Jede Transaktion führt alle `locks` vor allen `unlocks` aus.
  - M.a.W. nach dem ersten `unlock` darf kein `lock` mehr folgen
- Was sind die "zwei Phasen"?
  - Phase 1: Anforderung von Sperren
  - Phase 2: Freigabe von Sperren



Zwei-Phasen Sperrprotokoll

- 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 1
- Anwendung Theorem auf Beispiel 2
  - Sperren in Beispiel 2 führen nicht zu serialisierbarem Schedule → 2PL muss irgendwo verletzt werden (Wo?)
  - Wenn Beispiel 2 auf 2PL umgestellt wird, müsste der resultierende Schedule serialisierbar sein

Beispiel



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

Sperrverfahren

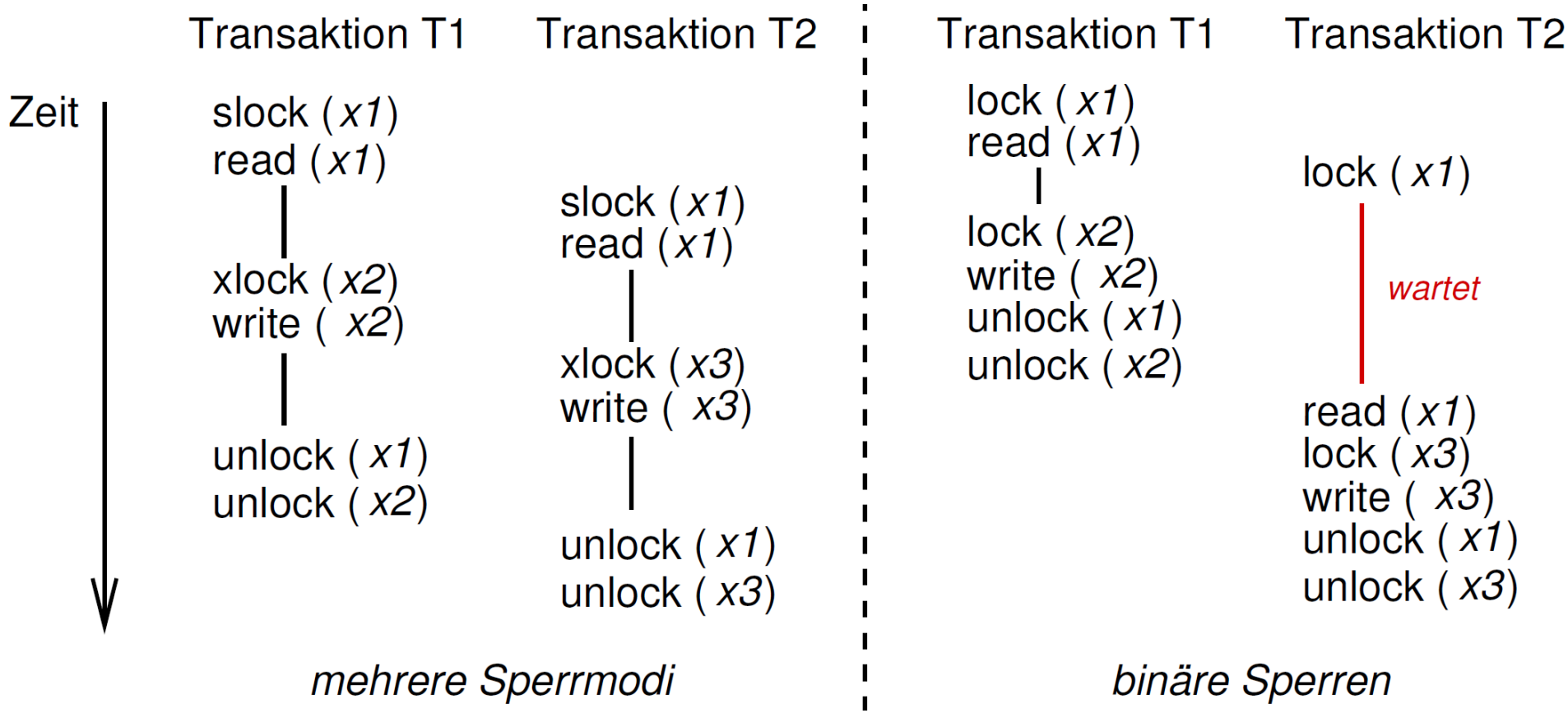
- Nachteil binärer Sperren
  - auch die konfliktfreien read/read Operationen sperren einander
- Lösung: mehrere Sperrmodi
  - read lock, shared lock (S)
    - lässt zu dass andere Transaktionen auch lesen (d.h. ebenfalls read lock anfordern), aber nicht schreiben
  - write lock, exclusive lock (X)
    - lässt keine Zugriffe anderer Transaktionen zu
  - führt zu drei Objektzuständen und damit zu drei elementaren Sperr-Operationen: slock, xlock, unlock

Kompatibilitätsmatrix der Sperroperationen

gehaltene Sperre \ angeforderte Sperre	S	X
S	Ja	Nein
X	Nein	Nein

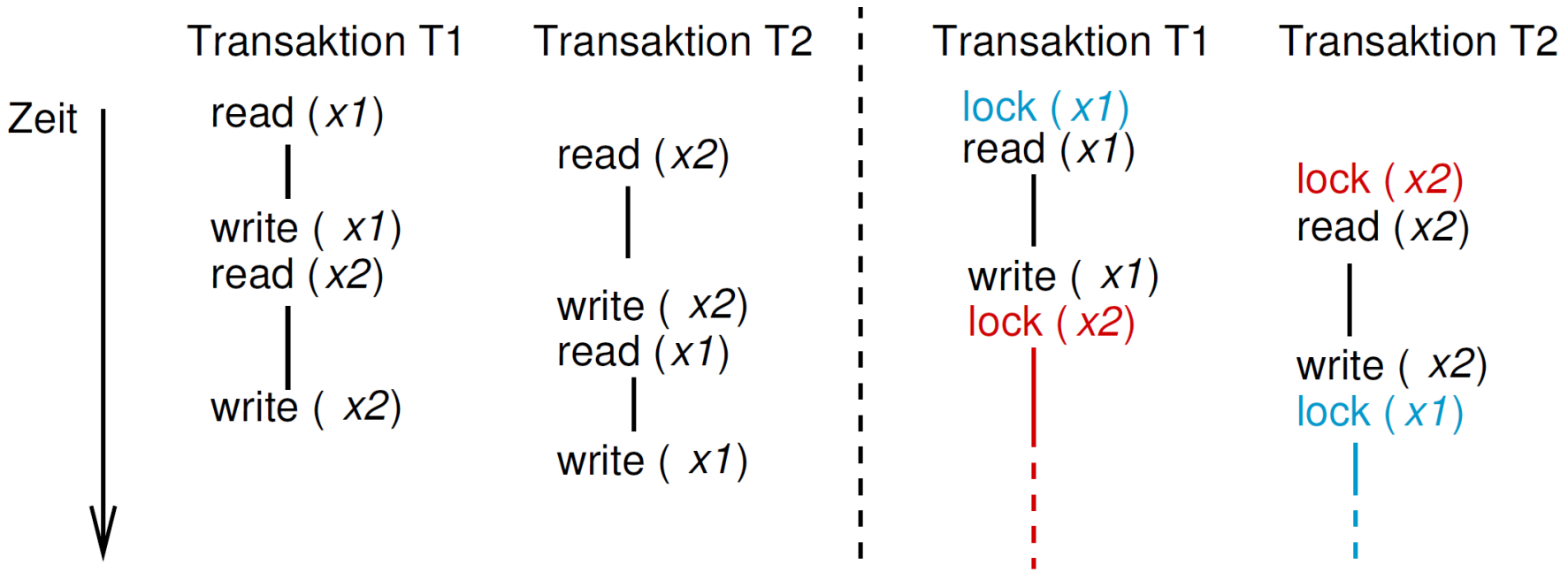
Beispiel

- Erhöhung Nebenläufigkeit durch mehrere Lockmodi



Deadlocks

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

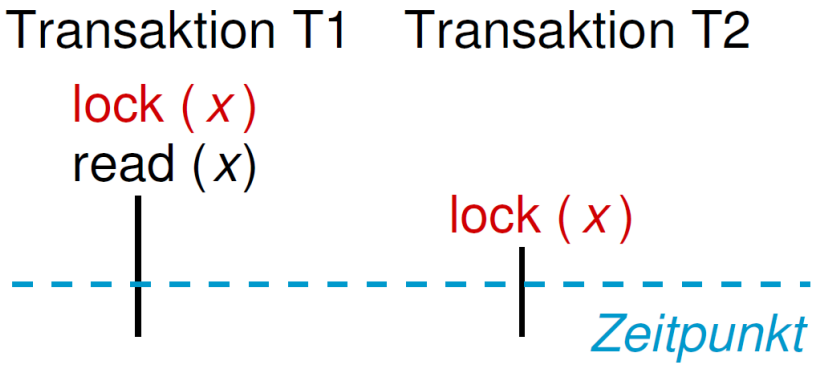


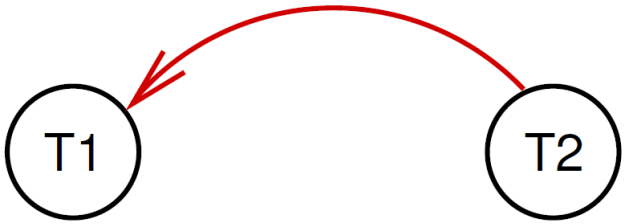
Ansätze zur Deadlock-Behandlung

- 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 könnte
  - Nachteil: brechen oft Transaktionen ab, die nie zu Verklemmung geführt hätten

Wartegraph (wait-for-graph)

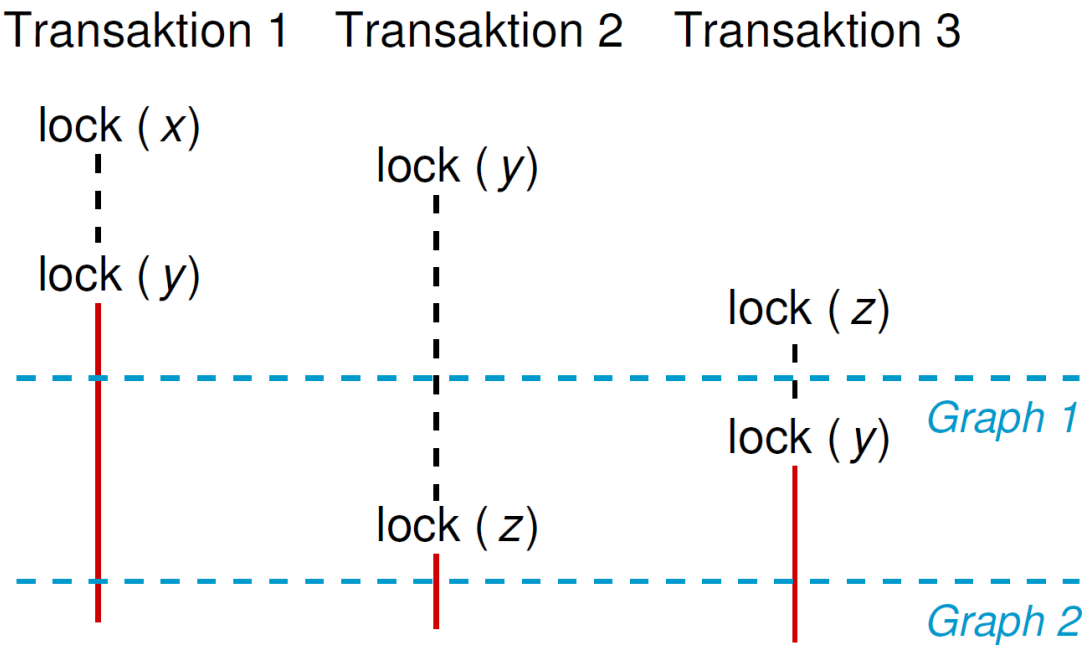
- Gerichteter Graph mit Transaktionen als Knoten
  - Kante $T_i \rightarrow T_j$ bedeutet, dass Transaktion T_i versucht Objekt zu sperren, das schon von Transaktion T_j gesperrt ist

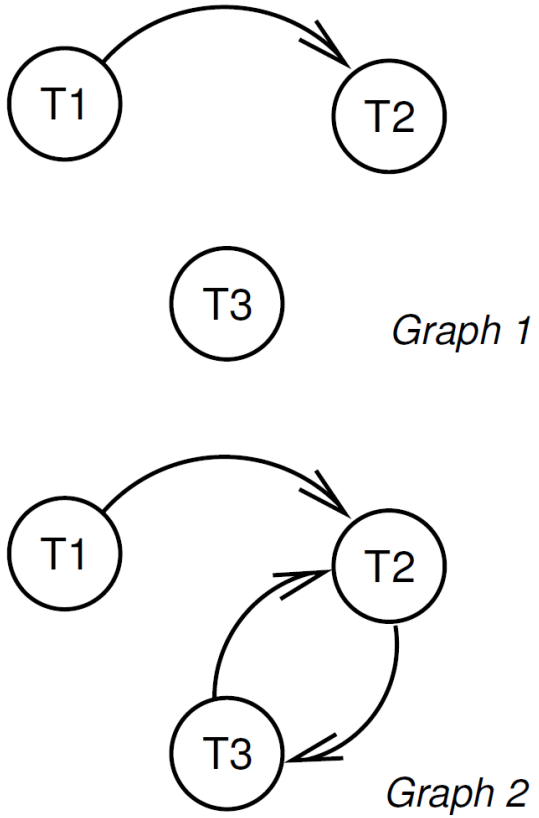




- Frage:
  - Woran erkennt man Deadlock?
- Antwort:
  - Zyklus im Wartegraphen

Beispiel mit drei Transaktionen





- Zyklus in Graph 2 zeigt Verklemmung an

Probleme bei der Deadlockerkennung

- Opferauswahl
  - welche Transaktion soll abgebrochen werden?
  - wünschenswert:
    - wenig fortgeschrittene Transaktionen bevorzugt abbrechen
    - Opferauswahl sollte nicht unfair sein, d.h. mehrmals dieselbe Transaktion treffen
- Wann prüfen?
  - naheliegender Ansatz (z.B. in PostgreSQL realisiert): starte Algorithmus wenn eine Transaktion bestimmte Zeit wartet
- Primitive Alternative: Timeouts
  - setze Transaktion zurück, die länger als Timeout wartet
  - schlägt auch zu, wenn kein Deadlock vorhanden