Kapitel 8: Transaktionen und ihre Realisierung

Lernziele und Überblick:

- ❏ Transaktionen als Kooperationsmodell, Eigenschaften von Transaktionen
- ❏ Isolation von Transaktionen:
  - Terminologie und Formalisierung
  - Klassifikation der Probleme durch Verletzung der Isolation
  - Isolation durch Sperrprotokolle
- ❏ Diskussion: Vor- und Nachteile niedriger Isolationsgrade

Datenmodelle und Kooperationsmodelle

Datenmodelle:

- ❏ Datenbanksprachen unterscheiden sich hauptsächlich in ihrer Mächtigkeit bei der Beschreibung der Struktur und des Zustandes der Datenbanken.
- ❏ Ziel: (vgl. Kapitel 3-6): Problemangepaßte Datenstrukturen und deklarative Anfragen für Nutzer der Datenbank

Kooperationsmodelle:

- ❏ Weitgehend orthogonal zur Wahl des Datenmodells ist das Kooperationsmodell.
- ❏ Ziel: Problemangepaßte Mechanismen zur Synchronisation und Fehlererholung von nebenläufigen Aktivitäten auf dem gemeinsamen Datenbankzustand
- ❏ *Transaktionen* sind das allgemein in Theorie und Praxis akzeptierte Kooperationsmodell.
- ❏ Andere Ansätze: Private & öffentliche Arbeitsumgebungen (*workspaces* bei CASE), *Message Passing* (nebenläufige Programmierung), Replizierte Datenbanken mit manueller Konfliktauflösung (*Groupware*, sehr gut geeignet für autonome mobile Aktoren).

Transaktionen für kooperierende Aktivitäten

- ❏ Kooperation mehrerer Geschäftsprozesse über *Seiteneffekte* auf gemeinsamen Daten
- ❏ Idee: Der gemeinsame Datenbankzustand ist ein Abbild des aktuellen Zustands der gemeinsamen Realität der nebenläufigen Geschäftsprozesse.
  - Beispiele: Buchhaltung, Lagerwirtschaft, Logistik, ...
- ❏ Das Datenbanksystem sorgt *implizit* für Synchronisation und Fehlererholung der *Zustandsänderungen*, die durch die Geschäftsprozesse ausgelöst werden.
- ❏ Der Programmierer ergreift keine expliziten Maßnahmen zur Synchronisation und Fehlererholung.

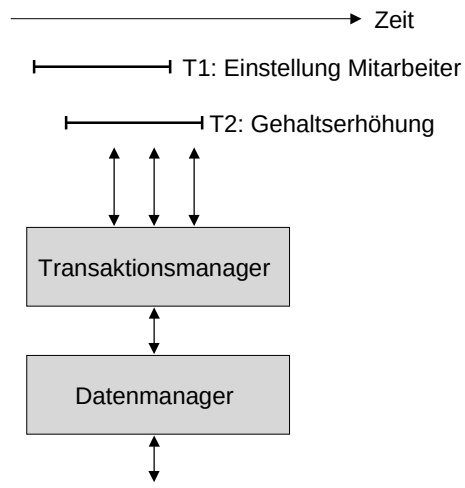
Grundsätzlich sind Transaktionen auch als Bausteine für komplexere, anwendungsbezogene Kooperationsmodelle geeignet
(s. Beispiel "betriebswirtschaftliche Transaktionen" in SAP R/3).

Definition: Transaktion

Eine Transaktion ist eine Folge von "zusammengehörigen" Operationen auf dem Systemzustand, die die folgenden Kriterien erfüllt (**ACID**-Eigenschaften):

- ❏ **Atomarität**: Die Änderungen des Systemzustands sind atomar, d.h. entweder finden alle oder keine statt. Diese Änderungen umfassen Datenbankzustandsänderungen, Nachrichten und Aktionen auf externen Geräten.
- ❏ **Konsistenz**: Eine Transaktion stellt eine konsistente Transformation des Datenbankzustands dar, wenn die Gesamtheit ihrer Aktionen keine der Integritätsbedingungen, die für den Datenbankzustand definiert sind, verletzt.
- ❏ **Isolation**: Obwohl Transaktionen nebenläufig ausgeführt werden, erscheint es für jede Transaktion *T*, als ob jede einzelne der nebenläufigen Transaktionen entweder als ganzes vor oder als ganzes nach *T* ausgeführt wird.
  (→ *Serialisierbarkeit*)
- ❏ **Dauerhaftigkeit**: Wenn eine Transaktion erfolgreich abgeschlossen ist, überleben ihre Änderungen des Datenbankzustands auch alle späteren Systemausfälle.

Beispiel: Projektdatenbank



Einstellung Mitarbeiter:

```
BEGIN_WORK
   SQL-Anfragen
   SQL-Änderungsoperationen
END_WORK
```

Vorteile von Transaktionen:

- Programmierer können Nebenläufigkeit und Fehlersituationen weitgehend "ignorieren". Sie sehen nur die Schlüsselworte `BEGIN_WORK`, `END_WORK` und `ROLLBACK_WORK`.
- Mit `ROLLBACK_WORK` können bei Bedarf in der Datenbank ausgeführte Änderungen durch die laufende Transaktion rückgängig gemacht werden.

ACID: Atomarität

Vorteile:

- ❏ Klare Fehlersemantik in komplexen Systemen.
- ❏ Schutz des gesamten Datenbestands, der meist wesentlich wertvoller als die einzelnen Zustandsänderungen einer Transaktion ist.
- ❏ Möglichkeit zum Rücksetzen von Anwendungstransaktionen im Falle von Verklemmungen, Überlastsituationen, ...

Probleme:

- ❏ Rücksetzbarkeit von "realen" Operationen (Geld auszahlen, Rakete abschießen, ...)
- ❏ Programmgesteuerte Reaktion auf `ROLLBACK`
  - Restaurieren des Bildschirminhalts, des Zustands von Dateien, ...
  - Benachrichtigung des Benutzers, ...

ACID: Konsistenz

- In der Praxis ein rein syntaktisches Kriterium: Jede sequentielle Anweisungsfolge zwischen `BEGIN_WORK` und `COMMIT_WORK` erzeugt im Einbenutzerbetrieb aus einem konsistenten Datenbankzustand einen konsistenten Datenbankzustand.
- Alternativ: Test statischer semantischer Konsistenzbedingungen durch den Datenmanager (Schlüsselintegrität, ...)
- Außerdem: Transaktionen wahren dynamische Konsistenzbedingungen (z.B. "Gehälter fallen nie").
- Das Rücksetzen von Transaktionen führt immer zu einem konsistenten Zustand.

Probleme:

- Wer sichert die Konsistenz zwischen Datenbankzustand und Realität ("Korrektheit von Transaktionen")?

ACID: Isolation

Durch die ungeschützte parallele Ausführung der Transaktionensanweisungen können Inkonsistenzen entstehen, auch wenn die einzelnen Transaktionen konsistent sind.

Idee:

- Transaktionen enthalten keine expliziten Synchronisationsbefehle (Zugriff auf Semaphore, Mutexe, Sperren, Nachrichtenschlangen, ...)
- Es existieren keine relevanten Reihenfolgeabhängigkeiten zwischen nebenläufigen Transaktionen (s. Beispiel: Einstellung / Gehaltserhöhung), d.h. jede serielle Ausführungsreihenfolge ist "erlaubt".

Überblick: (s. folgende Folien)

- Modellbildung (Transaktionsabhängigkeiten)
- Klassifikation der Probleme aufgrund fehlender Isolation (Anomalien)
- Formalisierung des Begriffs der Isolation (Serialisierbarkeit)
- Techniken zur Isolation von Transaktionen (Sperren)

ACID: Dauerhaftigkeit und Fehlererholung

Rücksetzen der Transaktion (ROLLBACK, UNDO), falls

- Systemfehler oder Integritätsverletzung während der Transaktionsausführung oder während `COMMIT_WORK`
- ROLLBACK_WORK Befehl

Wiederherstellen der Transaktionseffekte (REDO), falls

- Systemfehler nach erfolgreichem `COMMIT_WORK` basierend auf Logdateien, Archivkopien, dem Zustand von Spiegelsystemen, ...

Betrifft eine Transaktion mehr als einen Datenmanager, so muß am Transaktionsende ein *Zwei-Phasen-Commit-Protokoll* stattfinden:

- **Phase 1**: Feststellen der Commitfähigkeit:
  Der Transaktionsmanager schickt allen Datenmanagern eine PREPARE TO COMMIT Nachricht. Nur wenn alle Datenmanager mit "JA" antworten, wird die Commitfähigkeit festgestellt.
- **Phase 2**: Propagieren der getroffenen Entscheidung:
  Der Transaktionsmanager schickt allen Datenmanagern eine COMMIT bzw. eine ABORT Nachricht.

Transaktionen

Transaktionen sind Folgen von Einzelaktionen, die zwischen `begin` und `commit` oder `rollback` stattfinden.

```
T1    begin
      slock        A
      xlock        B
      read         A
      write        B
      commit
```

```
T2    begin
      slock        A
      read         A
      xlock        B
      write        B
      rollback
```

Symbolische Repräsentation einer Transaktion:

$< < t, a_i, o_k > | i = 1, ..., n; k = 1, ..., m >$

Der i-te Schritt der Transaktion *t* führt die Einzelaktion a_i auf dem Objekt o_k durch.

Isolation von Transaktionen

Ziel:

- *Datenbankhistorien*: Beschreibung der nebenläufigen ("verzahnten") Ausführung von Transaktionen.

Hilfsbegriffe:

- *Datenbankzustand*
- *Operation* auf *Objekt* (Komponente des Datenbankzustands)
- *Abhängigkeiten* zwischen Operationen
- *Transaktion* (mit impliziter und expliziter Fehlererholung)

Parallel wird das Konzept der *Sperren* auf Objekten eingeführt.

Anschließend wird gezeigt, daß durch geeignete Sperrprotokolle (= Sperralgorithmen) nur "erwünschte" Datenbankhistorien zugelassen werden.

Einzelaktionen

Generische Aktionen:

❑ `begin`	Legt den Anfang einer Transaktion fest.
❑ `commit`	Beendet eine erfolgreich ausgeführte Transaktion.
❑ `rollback`	Auf Objekten innerhalb einer Transaktion durchgeführte Manipulationen werden rückgängig gemacht, anschließend wird die Transaktion beendet.

Aktionen auf Objekten:

❑ `read A`	Liefert den Wert des benannten Objekts *A.*
❑ `write B`	Weist dem benannten Objekt *B* einen Wert zu und erzeugt neue Objektversion.
❑ `xlock B`	Setzt X-Sperre (*exklusiv*) für Manipulation des Objekts *B.*
❑ `slock A`	Setzt S-Sperre (*shared*) für Lesen des Objekts *A.*
❑ `unlock B`	Gibt Sperre für Objekt *B* frei.

Sperrkompatibilitätsmatrix

Ziel von Sperrprotokollen:

- Erzeugung von serialisierbaren Historien
- Dazu müssen Lese- und Schreibaktionen in einer bestimmten Systematik verwendet werden (Zweiphasen-Sperrprotokoll).

Kompatibilität		Sperrmodus	
		shared	exklusiv
Sperran-forderung	shared	kompatibel	Konflikt
	exklusiv	Konflikt	Konflikt

- Ist eine Sperranforderung gewährt worden, wird die Exekution fortgesetzt.
- Ist eine Sperranforderung nicht gewährt worden, wird die Exekution vorerst angehalten.

Notation

- Folgen *S, T* — S = < a, b, c > T = < d, e, f, >
- Konkatenation *S* || *T* — S || T = < a, b, c, d, e, f >
- i-tes Element einer Folge *S* — S [3] = c
- Teilfolge *S'* von *S* — S' = < S [i] | Prädikat (S [i]) >
- Geordnetes Paar als zweielementige Folge — < Name, Wert > < A, 3 >, < B, 4 >
- Zustand *Z* eines Systems — Z = { < Name, Wert > } Z = { < A, 3 >, < B, 4 > }

Transaktionsvereinfachung (1)

Transaktionen können dadurch vereinfacht werden, daß die Aktionen `begin`, `commit` und `rollback` durch die Aktionen `read`, `write`, `lock` und `unlock` substituiert sind:

Transaktionsvereinfachung:

- Verzichte auf `begin`.
- Ersetze `commit` durch:

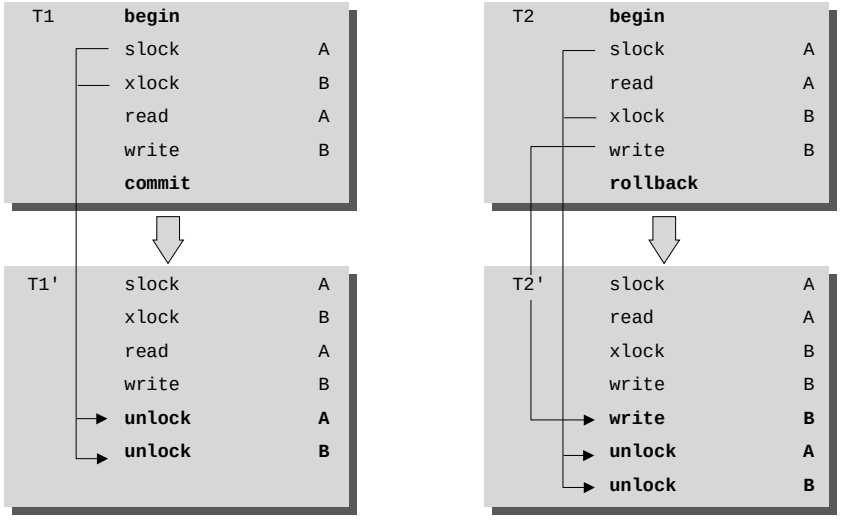
  < `unlock A` | falls `slock A` oder `xlock A` in *T* vorkommt >
- Ersetze `rollback` durch:

  < `write A` | falls `write A` in *T* vorkommt > ||

  < `unlock A` | falls `slock A` oder `xlock A` in *T* vorkommt >
- Beispiel: *T1'* und *T2'* auf der nächsten Folie

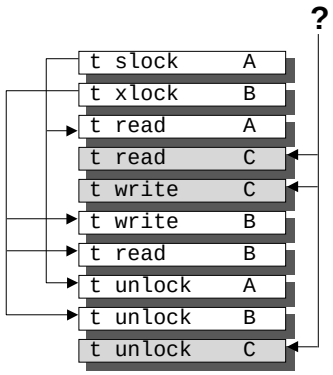
Transaktionsvereinfachung (2)

Beispiel:



Zweiphasige und wohlgeformte Transaktionen (1)

Wohlgeformte Transaktion:



- ❏ Jede lock Aktion korrespondiert in systematischer Weise mit den Lese- und Schreibaktionen der Transaktion.
  - Vor Leseaktionen erfolgt ein slock.
  - Vor Schreibaktionen erfolgt ein xlock.
- ❏ Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
- ❏ Sperren anderer Transaktionen müssen beachtet werden (Sperrkompatibilitätsmatrix).
- ❏ Jede read, write und unlock Aktion korrespondiert mit einer lock Aktion.
- ❏ Auf jede lock Aktion erfolgt eine korrespondierende unlock Aktion.
- ❏ Bei EOT muß eine Transaktion spätestens alle ihre Sperren zurückgeben.

Zweiphasige und wohlgeformte Transaktionen (2)

Zweiphasen-Sperrprotokoll:

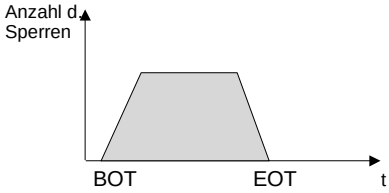
- ❏ Alle lock Aktionen gehen ihren unlock Aktionen voraus.
- ❏ In der *Wachstumsphase* T[1], ..., T[j] der Transaktion *T* werden Sperren angefordert (lock).
- ❏ In der *Schrumpfungsphase* T[j + 1], ..., T[n] werden die Sperren zurückgegeben (unlock).
- ❏ Wohlgeformte, zweiphasige Transaktionen genügen dem Zweiphasen-Sperrprotokoll.

Striktes Zweiphasen-Sperrprotokoll:

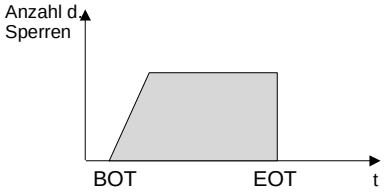
- ❏ Spezielle Zusatzanforderungen an den Verlauf der Phasen (s. nächste Folie) lösen Probleme, die über die Anforderungen der Isolation von Transaktionen hinausgehen (Dauerhaftigkeit, Verklemmung, ...).

Zweiphasige und wohlgeformte Transaktionen (3)

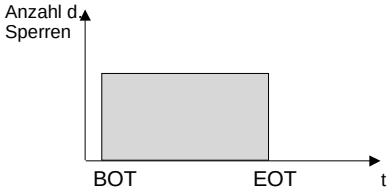
Zweiphasen-Sperrprotokoll (Dauerhaftigkeit, Verklemmung?)



Striktes Zweiphasen-Sperrprotokoll



Striktes Zweiphasen-Sperrprotokoll mit preclaiming



Historien

- ❏ Ziel: Modellierung der parallelen Ausführung von Transaktionen
- ❏ Folge von Aktionen bestehend aus einer Menge von Transaktionen unter Beibehaltung der Teilfolgen innerhalb der einzelnen Transaktionen
- ❏ Notation: $H = < < t, a_i, o_k > | i = 1, ..., n; k = 1, ..., m >$

  Ein Schritt einer Historie entspricht einer Aktion a_i der Transaktion *t* auf dem Objekt o_k.
- ❏ *Serielle Historie:* Die Transaktionen werden vollständig nacheinander ausgeführt. Es entstehen keine Inkonsistenzen aufgrund von Nebenläufigkeit.
- ❏ *Serialisierbare Historie*: Eine Historie, die äquivalent zu einer seriellen Historie ist. (Äquivalenz noch zu definieren!)
- ❏ *Sperr-Bedingungen* schränken die Menge der erlaubten Historien ein. Z.B. kann eine Exklusivsperre auf ein Objekt für eine Aktion nur dann genehmigt werden, wenn für dieses Objekt nicht schon eine Sperre im Rahmen einer anderen Transaktion besteht (s. Kompatibilität zwischen *Anforderungsmodus* und *Sperrmodus* auf Folie Sperrkompatibilitätsmatrix).
- ❏ *Legale Historie:* Sperren werden berücksichtigt (s. Sperrkompatibilitätsmatrix).

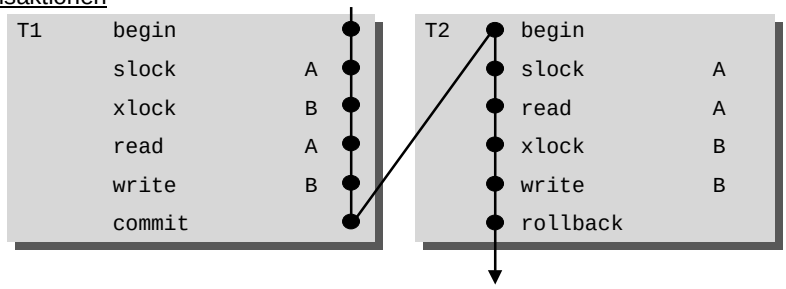
Ausführungshistorien, Beispiele (1)

Legal und seriell:

Historie

T1	slock	A	T2	slock	A
T1	xlock	B	T2	read	A
T1	read	A	T2	xlock	B
T1	write	B	T2	write	B
T1	unlock	A	T2	unlock	A
T1	unlock	B	T2	unlock	B

Transaktionen



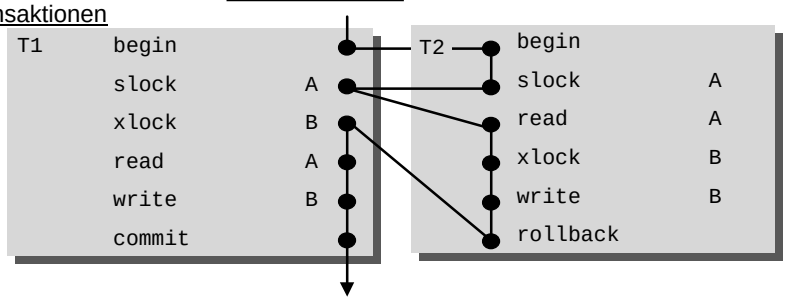
Ausführungshistorien, Beispiele (2)

Legal und nicht seriell:

Historie

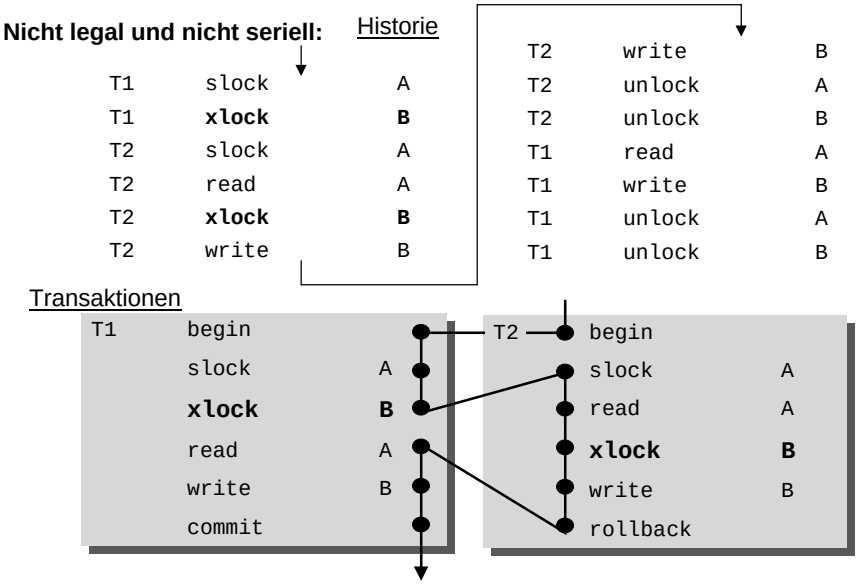
T2	slock	A	T2	unlock	A
T1	slock	A	T2	unlock	B
T2	read	A	T1	xlock	B
T2	xlock	B	T1	read	A
T2	write	B	T1	write	B
T2	write	B	T1	unlock	A
			T1	unlock	B

Transaktionen



Ausführungshistorien, Beispiele (3)

Nicht legal und nicht seriell:



Abhängigkeiten zw. Transaktionen in Historien

- Ziel: Definition des Datenflusses zwischen Transaktionen und Vergleich zweier Historien miteinander
- Eine Transaktion *T* hängt von einer Transaktion *T'* innerhalb einer Historie *H* ab, falls
  - *T'* Objekte beschrieben hat, die anschließend von *T* gelesen oder beschrieben werden oder
  - *T'* Objekte gelesen hat, die anschließend von *T* beschrieben werden.
- *Äquivalente Historie:* Zwei Historien besitzen die selben Abhängigkeiten, d.h. ihre Transaktionen greifen lesend und schreibend in beiden Historien in der gleichen Reihenfolge auf die gleichen Objekte zu.
- *Abhängigkeitsgraphen* beschreiben die Abhängigkeiten zwischen Transaktionen in Historien (s. nächste Folie).
- *Schmutzige Daten* sind Daten, auf die Transaktion *T* zugreifen kann, die aber potentiell von nebenläufigen Transaktionen *T'* noch geändert werden können.

Abhängigkeitsgraphen (1)

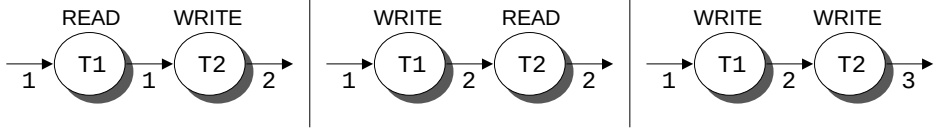
Kritische Elementarsituationen:

`A (i)`: Objekt *A* mit Versionsnummer *i*

Ausführungssequenzen:

READ→WRITE	WRITE→READ	WRITE→WRITE
`T1 read A (1)` `T2 write A (2)`	`T1 write A (2)` `T2 read A (2)`	`T1 write A (2)` `T2 write A (3)`

Abhängigkeitsgraphen:



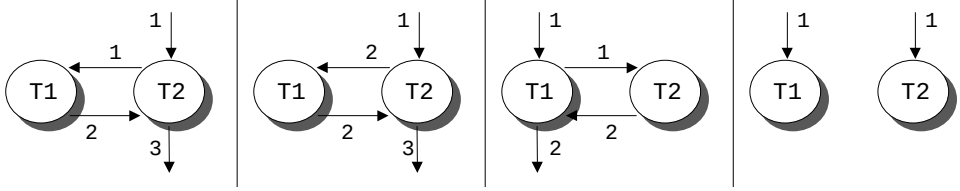
Leseabhängigkeiten (READ→READ) werden nicht berücksichtigt.

Abhängigkeitsgraphen (2)

Anormale T1/T2-Abhängigkeiten (Zyklen):

Lost Update Verlorenes Aktualisieren	Dirty Read Schmutziges Lesen	Unrepeatable Read Unwiederholbares Lesen	O.K.

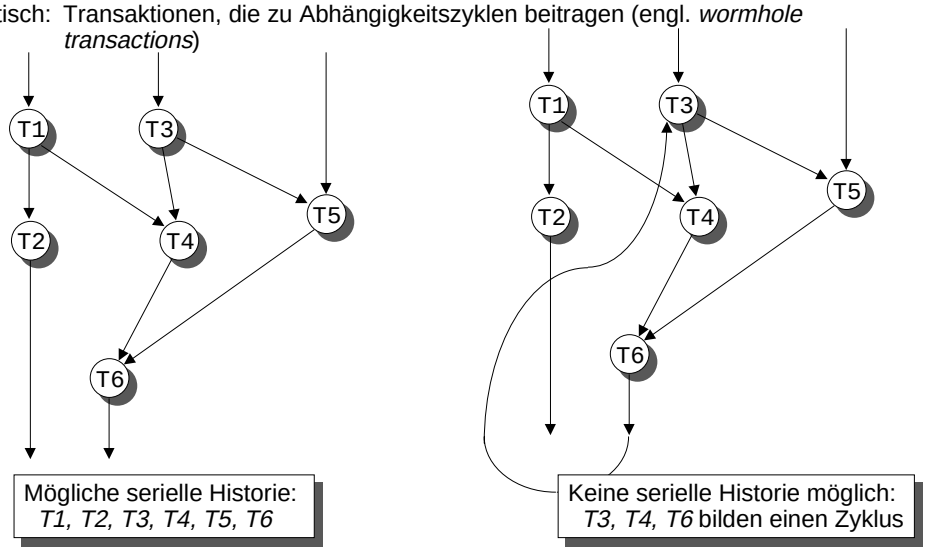
`T2 read A (1)`
`T1 write A (2)`
`T2 write A (3)`
	`T2 write A (2)`
	`T1 read A (2)`
	`T2 write A (3)`
		`T1 read A (1)`
		`T2 write A (2)`
		`T1 read A (2)`
			`T1 read A (1)`
			`T2 read A (1)`
			`T1 read A (1)`



`A (i)`: Objekt *A* mit Versionsnummer *i*

Abhängigkeitszyklen (1)

Kritisch: Transaktionen, die zu Abhängigkeitszyklen beitragen (engl. *wormhole transactions*)



Abhängigkeitszyklen (2)

Für die linke Seite (s. vorige Folie) gilt:

- ❏ VOR(T4) = {T1, T3} und NACH(T4) = {T6}. *T2* und *T5* sind deshalb weder vor noch nach *T4* und können deshalb nebenläufig zu *T4* ausgeführt werden. *T2, T4* und *T5* können in beliebiger Reihenfolge und beliebig überlappend ausgeführt werden.
- ❏ Die Relation <<< hat keine Zyklen und definiert eine partielle Ordnung der Transaktionen.

Für die rechte Seite (s. vorige Folie) gilt:

- ❏ VOR(T4) = {T1, T3, T4, T5, T6}
- ❏ Die Relation <<< hat Zyklen → keine Serialisierbarkeit

Abhängigkeitszyklen (3)

Folgende Aussagen sind äquivalent (ohne Beweis):

- Wenn für zwei Transaktionen *T* und *T'* gilt, daß Teile von *T* vor *T'* und Teile von *T'* vor *T* ausgeführt werden, dann liegt eine Nebenläufigkeitsanomalie vor, anderenfalls gibt es keine Nebenläufigkeitsanomalie.
- Hat der Abhängigkeitsgraph Zyklen, gibt es keine äquivalente serialisierbare Historie; ist er zyklenfrei, so existiert mindestens eine serialisierbare Historie.
- Eine serialisierbare Historie entsteht (genau) dann, wenn alle nebenläufigen Transaktionen dem Zweiphasen-Sperrprotokoll genügen.

Niedrige Grade der Isolation (1)

- Isolation wird von den meisten Systemen aus Performanzgründen nicht in vollem Umfang geboten. Es werden nur WRITE→WRITE und WRITE→READ Abhängigkeiten berücksichtigt (→ *cursor stability, repeatable reads*). READ→WRITE Abhängigkeiten werden meist ignoriert.
- Der ANSI SQL-Standard schreibt volle Isolation vor (Grad 3).

- **Grad 0:** Eine 0° Transaktion überschreibt keine schmutzigen Daten einer anderen Transaktion, wenn diese mindestens Grad 1 hat.
- **Grad 1:** 1° Transaktionen haben keine verlorengegangenen Änderungen.
- **Grad 2:** 2° Transaktionen haben keine verlorengegangenen Änderungen und keine schmutzige Leseoperationen.
- **Grad 3:** 3° Transaktionen haben keine verlorengegangenen Änderungen und nur wiederholbare Leseoperationen.

Niedrige Grade der Isolation (2)

Sperrprotokolle der Isolationsgrade haben die folgenden Eigenschaften:

- **Grad 0:** Wohlgeformt in bezug auf Schreiboperationen (*anarchy*)
- **Grad 1:** Zweiphasig in bezug auf exklusive Sperren und wohlgeformt in bezug auf Schreiboperationen (*browse*)
- **Grad 2:** Zweiphasig in bezug auf exklusive Sperren und wohlgeformt (*cursor stability*)
- **Grad 3:** Zweiphasig und wohlgeformt (*isolated, serializable, repeatable reads*)

Die Isolationsgrade entsprechen den Abhängigkeiten:

1° ↔ WRITE→WRITE

2° ↔ WRITE→WRITE, WRITE→READ

3° ↔ WRITE→WRITE, WRITE→READ, READ→WRITE

Niedrige Grade der Isolation (3)

Isolationstheorem:
Erfüllt eine Transaktion das 0°, 1°, 2° oder 3° Sperrprotokoll, so erreicht sie Isolationsgrad 1°, 2° oder 3° in jeder legalen Historie, wenn alle anderen Transaktionen mindestens vom Grad 1° sind.

- In der Praxis bedeutet dies: Der Grad der eigenen Transaktion kann unabhängig von anderen Transaktionen gewählt werden.
- **Ausnahme:** Eingabedaten von 1° Transaktionen sind möglicherweise schmutzig. Werden diese Daten zum Aktualisieren der Datenbank verwendet, können andere Transaktionen inkonsistente Daten bekommen. Das Theorem geht deshalb davon aus, daß 1° Transaktionen "wissen, was sie tun".
- **Konklusion:** Auch wenn eine Transaktion schmutzig Daten liest, nimmt man an, daß sie die Datenbank in einen konsistenten Zustand transformiert. Viele Systeme lassen 1° Isolationsverhalten nur für Lese-Transaktionen zu.

Grade der Isolation in SQL

- Transaktionen mit niedrigen Isolationsgraden benötigen weniger Sperren, halten diese für eine kürzere Zeit und sorgen somit für bessere Performanz.
- 2° Isolation erlaubt es Transaktionen, die gesamte Datenbank abzusuchen und dabei wenig Sperren zu halten und somit andere Transaktionen nicht zu stören.

```
select count(*)
from mitarbeiter
where gehalt > 10000;
```

3°: Alle Datensätze aus `mitarbeiter` bzw. die gesamte Tabelle werden bis Transaktionsende gesperrt.

2°: Jeder Datensatz wird nach dem Lesen vom System automatisch freigegeben.

- Volle Isolation kann nur bei Grad 3 erreicht werden.
- SQL stellt die folgenden Optionen bereit: **`set transaction isolation level`**

`read uncommitted`	Grad 1
`read committed`	Grad 2
`repeatable read`	Grad 2,999 (Phantome)
`serializable`	Grad 3

Cursor stability

- In SQL wird üblicherweise eine Islolation implementiert, die mehr Konsistenz als Grad 2 bietet (*cursor stability*).

Cursor stability vermeidet verlorengegangene Änderungen:

```
exec sql declare cursor c for
         set gehalt
         from mitarbeiter
         where no = nr;
exec sql open c;
exec sql fetch c into :gehalt;
gehalt = gehalt * 1,5;
exec sql update mitarbeiter
         set gehalt = :gehalt;
         where no = nr;
exec sql close c;
```

Verlorengegangene Änderungen mit 2° Isolation:

```
exec sql select gehalt
         into :gehalt
         from mitarbeiter
         where no = nr;
gehalt = gehalt * 1,5;
exec sql update mitarbeiter
         set gehalt = :gehalt
         where no = nr;
```

Searched Update is OK 1° oder 2°:

```
exec sql update mitarbeiter
     set gehalt = gehalt * 1,5
     where no = nr;
```

Zusammenfassung: Niedrige Isolationsgrade

Vorteile:

- ❏ Daten sind nur kurze Zeit gesperrt.
- ❏ Weniger Daten werden gesperrt.
- ❏ Potentiell höhere Nebenläufigkeit

Nachteile:

- ❏ Möglicherweise inkonsistente Eingabe- und Ausgabedaten
- ❏ Eine ROLLBACK Operation muß für ihre Aktionen Sperren setzen.
- ❏ 0° Transaktionen unterstützen kein ROLLBACK.
- ❏ Für reine Lesetransaktionen gilt 0° = 1°.

Die Eigenschaften niedriger Isolationsgrade sind in den folgenden Tabellen nochmals gegenübergestellt.

Überblick über die Isolationseigenschaften (1)

	Grad 0	**Grad 1**
Name	*Chaos*	*Browse*
Schutz	Stört keine Transaktionen höheren Grades	0° und keine verlorengegangenen Änderungen
Commit Zeitpunkt	Schreiboperation sofort sichtbar	Schreiboperation am Ende der Transaktion sichtbar
Schmutzige Daten	Schmutzige Daten werden nicht überschrieben	Eigene schmutzige Daten werden nicht überschrieben
Sperrprotokoll	Kurze exklusive Sperren beim Schreiben	Lange exklusive Sperren beim Schreiben
Transaktionsstruktur	Wohlgeformt	Wohlgeformt und zweiphasig
Nebenläufigkeit	Größtmöglich: kurze Schreibsperren	Groß: nur Schreibsperren
Aufwand	Kleinstmöglich: kurze Schreibsperren	Klein: nur Schreibsperren
Rücksetzen	Nicht möglich	Nicht beendete Transaktionen
Fehlererholung	Änderungen können verlorengehen und 3° verletzten	
Beachtete Abhängigkeiten	Keine	WRITE→WRITE

Überblick über die Isolationseigenschaften (2)

	Grad 2	**Grad 3**
Name	*Cursor Stability*	*(Serialisierbar) Repeatable Read*
Schutz	Keine verlorengeg. Änderungen Keine schmutzigen Leseoperationen	Keine verlorengeg. Änderungen,keine schmu. Leseoper., wiederh. Leseoper.
Commit Zeitpunkt	Schreiboperation am Ende der Transaktion sichtbar	Schreiboperation am Ende der Transaktion sichtbar
Schmutzige Daten	0°, 1° und es werden keine schmutzigen Daten gelesen	0°, 1°, 2° und andere lesen keine geschriebenen schmutzige Daten
Sperrprotokoll	1°und nicht exklusive Sperren auf lesende Daten	1° und lange Lesesperren
Transaktions-struktur	wohlgeformt und zweiphasig	Wohlgeformt und zweiphasig
Nebenläufig-keit	Mittel: wenige Lesesperren	Klein: alle betroffenen Daten werden gesperrt
Aufwand	Mittel: Scheib- und Lesesperren, Lesesperren werden nicht gehalten	Mittel: hält beide Arten von Sperren
Rücksetzen	Nicht beendete Transaktionen	Nicht beendete Transaktionen
Fehlererholung	Wie 1°	Wie 1°, in jeder <<< Reihenfolge wiederholbar
Beachtete Abhängigkeiten	WRITE→WRITE, WRITE→READ	WRITE→WRITE , WRITE→READ READ→WRITE