Datenbanken

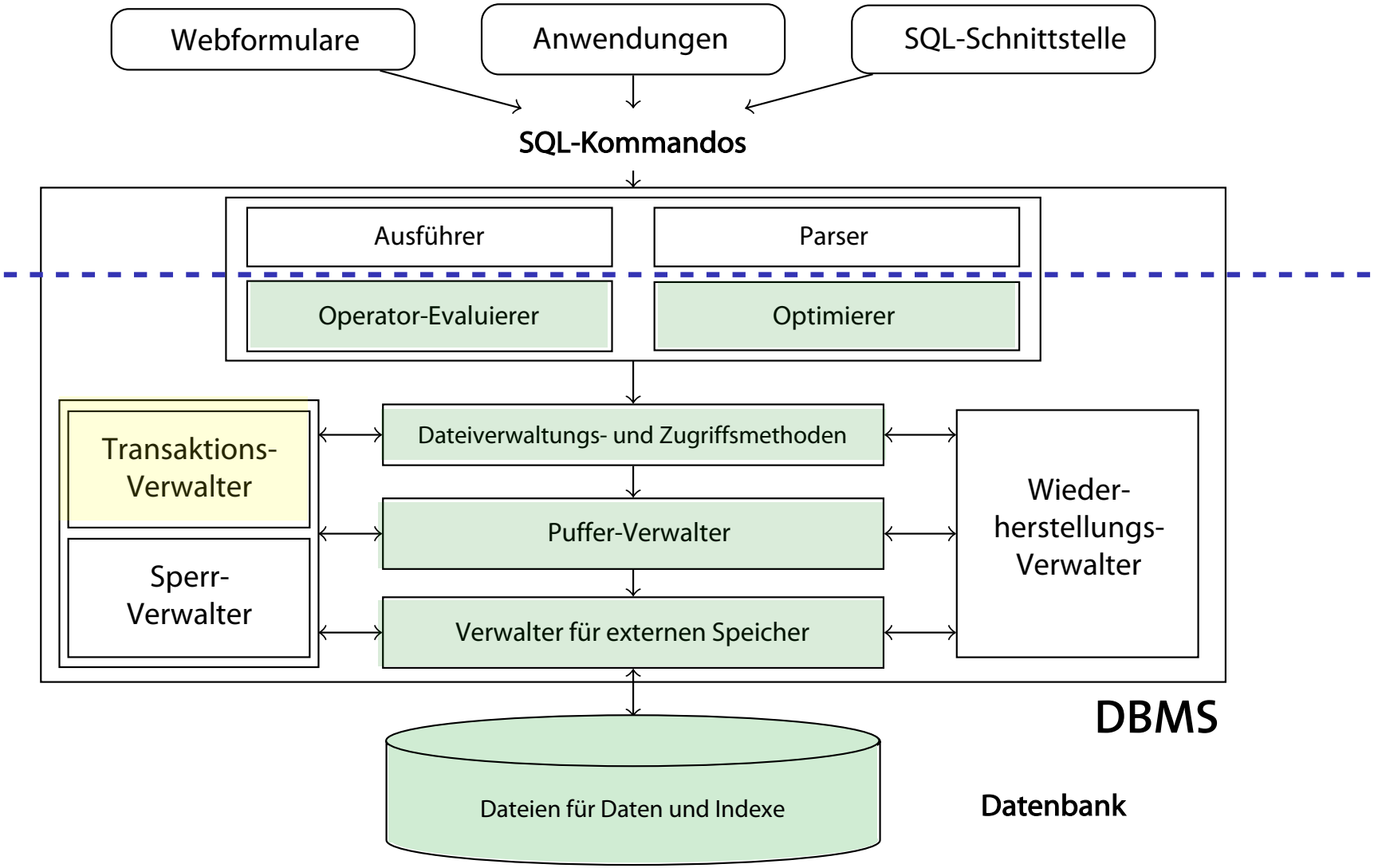
Transaktionsmanagement Teil 1

Dr. Özgür Özçep

Universität zu Lübeck

Institut für Informationssysteme

Transaktionsverwaltung



Danksagung

- Diese Vorlesung ist inspiriert von den Präsentationen zu dem Kurs:

  „Architecture and Implementation of Database Systems“ von Jens Teubner an der ETH Zürich

- Graphiken und Code-Bestandteile wurden mit Zustimmung des Autors (und ggf. kleinen Änderungen) aus diesem Kurs übernommen

Motivation

Eine einfache Transaktion

- Ab und zu verwende ich meine Kreditkarte, um Geld von meinem Konto abzuheben
- Der Bankautomat führt folgende Transaktion auf der Datenbasis der Bank aus

```
1 bal ← read_bal (acct_no) ;
2 bal ← bal − 100 CHF ;
3 write_bal (acct_no, bal) ;
```

- Wenn alles fehlerfrei abläuft, wird mein Konto richtig verwaltet

Nebenläufiger Zugriff

Mein Frau verwendet eine Karte für das gleiche Konto...

- Eventuell verwenden wir unsere Karten zur gleichen Zeit

me	my wife	DB state
		1200
bal ← `read` (*acct*) ;		1200
	bal ← `read` (*acct*) ;	1200
bal ← *bal* − 100 ;		1200
	bal ← *bal* − 200 ;	1200
		1100
`write` (*acct*, *bal*) ;		1000
	`write` (*acct*, *bal*) ;	

- Die erste Aktualisierung des Kontos ist verlorengegangen: Mich freut's! Allerdings...

... kann es auch nach hinten losgehen

- Diesmal wird Geld von einem Konto auf ein anderes transferiert

```
   // Subtract money from source (checking) account
1  chk_bal ← read_bal (chk_acct_no) ;
2  chk_bal ← chk_bal − 500 CHF ;
3  write_bal (chk_acct_no, chk_bal) ;
   // Credit money to the target (saving) account
4  sav_bal ← read_bal (sav_acct_no) ;
5  sav_bal ← sav_bal + 500 CHF ;
6  write_bal (sav_acct_no, sav_bal) ;
```

- Ann.: Bevor die Transaktion zum Schritt 6 kommt, wird die Ausführung abgebrochen (Stromversorgungsproblem, Plattenproblem, Softwarefehler, ...).
- Mein Geld ist verschwunden ☹

ACID-Eigenschaften und Anomalien

ACID-Eigenschaften für Transaktionen

Um die im Beispiel genannten und viele andere Effekte zu vermeiden, stellen DMBS folgende Eigenschaften sicher

- **A**tomicity: Entweder werden alle oder keine Werteänderungen einer Transaktion in den Datenbankzustand übernommen
- **C**onsistency: Eine Transaktion überführt einen konsistenten Zustand (FDs, Integritätsbedingungen) in einen anderen
- **I**solation: Eine Transaktion berücksichtigt bei der Berechnung keine Effekte anderer parallel laufender Transaktionen
- **D**urability: Effekte einer erfolgreichen Transaktion werden persistent gemacht

Anomalien: Lost Update

- Wir haben schon „Lost Update“ im Beispiel betrachtet
- Effekte einer Transaktion gehen verloren, weil eine andere Transaktion geänderte Werte unkontrolliert überschreibt

Anomalien: Inconsistent Read

Betrachten wir die Überweisung in SQL

Transaction 1

```
UPDATE Accounts
  SET balance = balance - 500
  WHERE customer = 4711
    AND account_type = 'C';

UPDATE Accounts
  SET balance = balance + 500
  WHERE customer = 4711
    AND account_type = 'S';
```

Transaction 2

```
SELECT SUM(balance)
  FROM Accounts
 WHERE customer = 4711;
```

➢ Transaktion 2 sieht einen inkonsistenten Zustand

Aufgabe:

Geben Sie ein Beispiel für eine Integritätsbedingung (constraint) an, die beim "Inconsistent Read" des vorigen Beispiels verletzt wäre.

Aufgabe:

Geben Sie ein Beispiel für eine Integritätsbedingung (constraint) an, die beim "Inconsistent Read" des vorigen Beispiels verletzt wäre.

Lösung:

- Dispositionskredit-Constraint über Gesamtvermögen SUM(balance) $\geq$ -200;
- Das Constraint wäre verletzt z.B.
  - bei Girokonto balance = 100 und
  - Sparkonto balance = 100
  - Tranksaktion 2 sieht DB-Zustand mit SUM(balance)= 100-500 + 100 = -300 < -200

Anomalien: Dirty Read

An einem anderen Tag heben meine Frau und ich zur gleichen Zeit Geld vom Automaten ab

me	my wife	DB state
$bal \leftarrow$ `read` $(acct)$;		1200
$bal \leftarrow bal - 100$;		1200
`write` $(acct, bal)$;		1100
	$bal \leftarrow$ `read` $(acct)$;	1100
	$bal \leftarrow bal - 200$;	1100
`abort` ;		1200
	`write` $(acct, bal)$;	900

- Die Transaktion meiner Frau hat schon einen geänderten Zustand gelesen, bevor meine Transaktion zurückgerollt wird

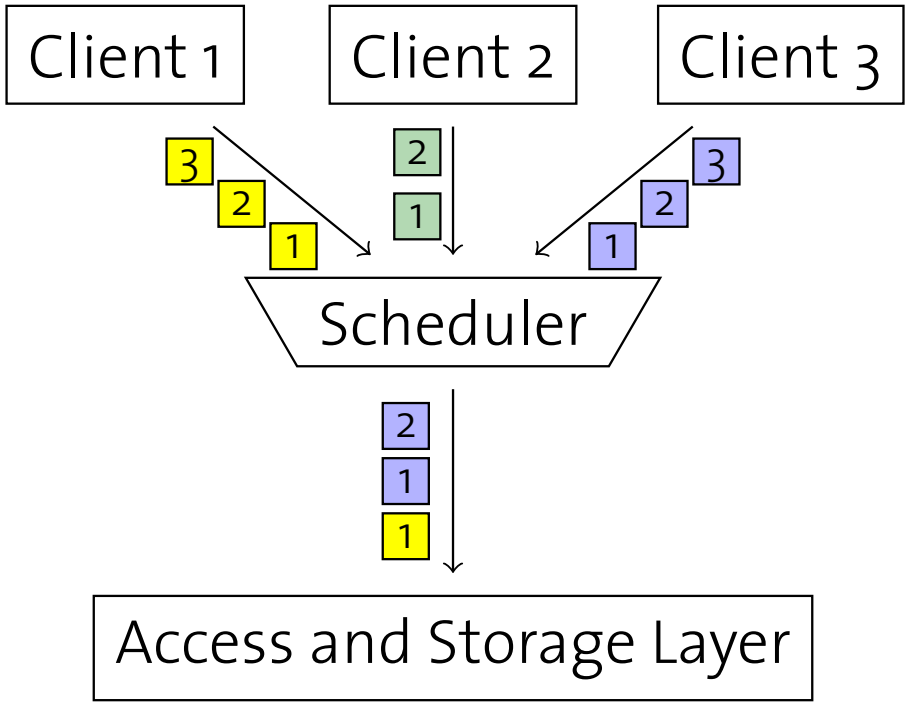
How to have the cake and eat it

- Anomalien ließen sich durch einfache sequentielle Ausführung der Transaktionen vermeiden
- Wir wollen aber beides: Keine Anomalien, dabei aber Nebenläufigkeit

Transaktionen und Serialisierbarkeit

Nebenläufige Ausführung

- Ein Steuerprogramm (Scheduler) entscheidet über die Ausführungsreihenfolge der nebenläufigen Datenbankzugriffe



Datenbankobjekte und Zugriffe darauf

- Wir nehmen ein vereinfachtes Datenmodell an
  - Eine Datenbank besteht aus einer Menge von benannten Objekten/Entitäten. In jedem Zustand hat ein Objekt einen Wert.
  - Transaktionen greifen auf ein Objekt o mit den Operationen read und write zu

- In einer relationalen DB haben wir:
  Objekt $\widehat{=}$ Komponente eines Tupels

Transaktion: Definition

- Eine Datenbanktransaktion ist eine (strikt geordnete) Folge von Schritten, wobei ein Schritt eine Zugriffsoperation auf ein Objekt/Entität ist
  - Transaktion $T = <s_1, ..., s_n>$
  - Schritt $s_i = a_i(e_i)$
  - Zugriffsoperation $a_i \in \{ r(ead), w(rite) \}$
- Die Länge einer Transaktion ist definiert als die Anzahl der Schritte $|T| = n$
- Beispiel: $T = <r(A), w(A), r(B), w(B)>$
- Abarbeitung durch Mischung der Schritte mehrerer Transaktionen (Sequenzieller Plan, Sequenz, S)

Sequenzielle Pläne

- Ein (sequenzieller) Plan S für eine Menge von Transaktionen $\{T_1, ..., T_n\}$ ist eine beliebige Folge von Schritten $S(k) = (T_k, a_i, e_i)$, so dass
  1. S genau die Schritte aus den Transaktionen T_i $(1 \leq i \leq n)$ enthält und
  2. die Ordnung der Schritte aus den Transaktionen jeweils erhalten bleibt, d.h.: Wenn $(a_p,e_p) < (a_q,e_q)$ in T_j so auch $(T_j, a_p,e_p) < (T_j,a_q,e_q)$ in S

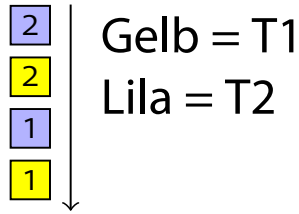
Im Plan wird die zu einem Schritt gehörige Transaktion als Index vermerkt also z.B. statt $(T_j, read,e_i)$ kürzer $r_j(e_i)$ geschrieben

Serielle Ausführung

Ein spezieller sequenzieller Plan ist die serielle Ausführung

- Ein Plan heißt seriell genau dann, wenn für jede Transaktion T_j alle ihre Schritte direkt aufeinanderfolgen (ohne Schritte anderer Transaktion dazwischen)

Betrachten wir das Geldautomatenbeispiel:



- $S = <r_1(B), r_2(B), w_1(B), w_2(B)>$
- Dieser Plan ist nicht seriell

Wenn meine Frau später zum Automaten geht, ergibt sich

- $S = <r_1(B), w_1(B), r_2(B), w_2(B)>$
- Dieser Plan ist seriell

Korrektheit der seriellen Ausführung

- Anomalien können nur auftreten, wenn die Schritte mehrerer Transaktionen verschränkt ausgeführt werden (Multi-User-Modus)
- Falls alle Transaktionen bis zum Ende ausgeführt werden (keine Nebenläufigkeit), treten keine Anomalien auf
- Jede serielle Ausführung ist korrekt
- Verzicht auf nebenläufige Ausführung nicht praktikabel, da zu langsam (Wartezeit auf Platten)
- Jede verschränkte Ausführung, die einen gleichen Zustand wie eine serielle erzeugt, ist korrekt

Abarbeitungsreihenfolge

- Vorstellung: Sequenzieller Plan gegeben
- Manchmal kann man einfach Teilschritte aus verschiedenen Transaktionen in einem Plan umordnen
  - Nicht jedoch die Teilschritte innerhalb einer einzelnen Transaktion (sonst eventuell anderes Ergebnis, siehe Definition eines sequenziellen Plans)
- Jeder Plan S‘, der durch legale Umordnung von S generiert werden kann, heißt äquivalent zu S
- Falls Umordnung nicht möglich, weil das Ergebnis verfälscht werden könnte → Konflikt
- Wie ist das definierbar?

Konflikte

Definition eines Konflikts:

- Zwei Operationen $a_i(e)$ und $a'_j(e')$ stehen in Konflikt zueinander in S, wenn
  - sie zu zwei verschiedenen Transaktionen gehören $i \neq j$,
  - sie das gleiche Objekte referenzieren $(e = e')$ und
  - mindestens eine der Operationen a oder a' eine Schreiboperation ist
- Hierdurch ist eine sog. Konfliktmatrix definiert

	`read`	`write`
`read`		×
`write`	×	×

Serialisierbarkeit

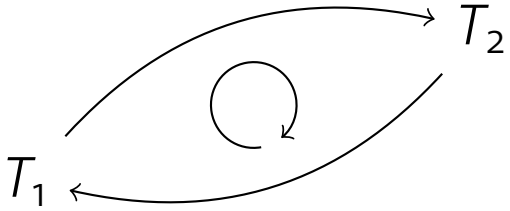
- Ein Plan S heißt serialisierbar gdw. er zu einem seriellen Plan S‘ äquivalent ist
- Die Ausführung eines serialisierbaren Plans S ist korrekt (S braucht nicht seriell zu sein)
- (Konflikt)-Korrektheit eines Plans kann anhand des Konfliktgraphen (Serialisierungsgraph) G_S gezeigt werden
  - Knoten von G_S sind die Transaktionen T_i aus S
  - Kanten $T_i \longrightarrow T_j$ werden hinzugefügt gdw. S konfligierende Operationen $a_i(e)$ und $a'_j(e')$ enthält ($i \neq j$), so dass $a_i(e)$ vor $a'_j(e')$
  - S ist serialisierbar, wenn G_S zyklenfrei ist
  - Serielle Ausführung bestimmbar durch topologische Sortierung

Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

($w_1(A)$ kann nicht an $r_2(A)$ vorbeigeschoben werden, da Konflikt)



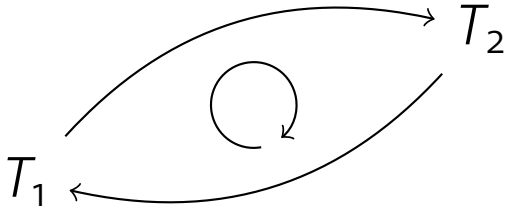
nicht serialisierbar

Serialisierungsgraph

Beispiel: ATM-Transaktion

- $S = \langle r_1(A), r_2(A), w_1(A), w_2(A) \rangle$

($w_1(A)$ kann nicht an $r_2(A)$ vorbeigeschoben werden, da Konflikt)

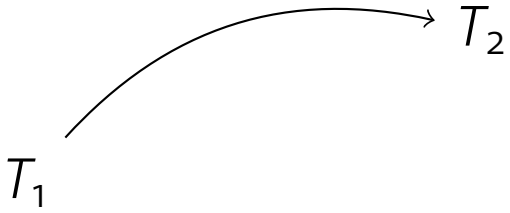


nicht serialisierbar

Beispiel: Zwei Geldtransfers

- $S = \langle r_1(C), w_1(C), r_2(C), w_2(C), r_1(S), w_1(S), r_2(S), w_2(S) \rangle$

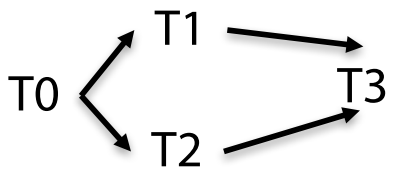
($r_1(S)$ und $w_1(S)$ können an $r_2(C)$ und $w_2(C)$ vorbeigeschoben werden, da kein Konflikt)



serialisierbar

Erinnerung Topologische Sortierung

Toplogische Sortierung eines gerichteten azyklischen Graphen G = Lineare Ordnung < der Knoten von G, so dass für alle Knoten v_i, v_j aus v_i -> v_j folgt v_i < v_j



Mögliche topologische Sortierungen

1. T0 < T1 < T2 < T3
2. T0 < T2 < T1 < T3

Aufgabe:
Ist dieser Plan serialisierbar?

Schritt	T1	T2
1	BOT	
2	r(A,a1)	
3	a1 = a1 -50	
4	w(A,a1)	
5		BOT
6		r(A,a2)
7		a2 = a2 -100
8		w(A,a2)
9		r(B,b2)
10		b2 = b2 + 100
11		w(B,b2)
12		commit
13	r(B,b1)	
14	b1 = b1 + 50	
15	w(B,b1)	
16	commit	

Aufgabe:

Ist dieser Plan serialisierbar?

Schritt	T1	T2
1	BOT	
2	r(A,a1)	
3	a1 = a1 -50	
4	w(A,a1)	
5		BOT
6		r(A,a2)
7		a2 = a2 -100
8		w(A,a2)
9		r(B,b2)
10		b2 = b2 + 100
11		w(B,b2)
12		commit
13	r(B,b1)	
14	b1 = b1 + 50	
15	w(B,b1)	
16	commit	

Lösung:

Obwohl der Serialisierbarkeitsgraph zyklisch ist, ist die Verzahnung von T1 und T2 unproblematisch (wegen der Kommutativität der Updateoperation). Aber der Planer sieht nur die Ebene der read-write-Operationen.

Tatsächlich gilt: Serialisierbarkeit ist **nicht entscheidbar**.
Mit dem graphentheoretischen Kriterium haben wir also nur eine Approximation, die unter der Bezeichnung "Konflikt-Serialisierbarkeit" firmiert

Aufgabe:

Ist dieser Plan serialisierbar?

Lösung (Forts.):

Der Plan rechts ist auf der read-write Ebene genau so aufgebaut wie der erste Plan, allerdings ist der Plan rechts nicht serialisierbar.

Schritt	T1	T2
1	BOT	
2	r(A,a1)	
3	a1 = a1 -50	
4	w(A,a1)	
5		BOT
6		r(A,a2)
7		a2 = a2 * 1.03
8		w(A,a2)
9		r(B,b2)
10		b2 = b2 * 1.03
11		w(B,b2)
12		commit
13	r(B,b1)	
14	b1 = b1 + 50	
15	w(B,b1)	
16	commit	