Transaktionsverwaltung

1. Schnellkurs: Serialisierbarkeit, Isolationslevel, Synchronisationsverfahren, Savepoints, Logging, Implementierungsaspekte
   → Härder, Rahm Buch
2. Lastkonstrolle
   → Mönkeberg, Weikum, VLDB 1992
3. Sperrverwaltung und Recovery für Indexe
   → Mohan, SIGMOD 1992
   → Kornacker et al., SIGMOD 1997
4. (POSTGRES: ein etwas anderes System)
   → Stonebraker, VLDB 1987
5. Erweiterte Transaktionskonzepte
   → Härder, Rahm Buch
6. (Recovery in Client/Server Umgebungen und von Anwendungsprogrammen)
   → Franklin et al., SIGMOD 1992
   → Lomet, Weikum, SIGMOD 1998

- Das Thema verdient eigentlich eine eigene Vorlesung.

Transaktionskonzept

Eine Transaktion ist eine folge von Operationen mit folgenden *ACID* Eigenschaften:

- **Atomicity:** Es werden alle Operationen oder gar keine Operation ausgeführt.
- **Consistency:** Wenn alle Operationen ausgeführt werden, dann überführt die Transaktion die Datenbasis von einem konsistenten in einen (anderen) konsistenten Zustand.
- **Isolation:** Im Mehrbenutzerbetrieb sind die Operationen der Transaktionen *serialisierbar*.
- **Durability:** Die Änderungen einer Transaktionen können nur durch eine andere Transaktionen überschrieben werden.

Komponenten der Transaktionsverwaltung:

- **Synchronisation:** Sorgt für *Isolation.*
- **Logging, Recovery, Commit-Behandlung:** Sorgt für *Atomicity* und *Durability.*
- **Integritätskontrolle:** Sorgt für *Consistency.*

Beispiele

Integritätsbedingung: Summe aller Kontostände ist eine Konstante.

Transaktion 1:

BOT;
andreas.konto += 100.000;
markus.konto −= 100.000;
COMMIT;

Transaktion 2:

BOT;
andreas.konto += 50.000;
ABORT;

Transaktion 3:

BOT;
andreas.konto += 70.000;
markus.konto −= 50.000;
COMMIT;

Synchronisation und Mehrbenutzerbetrieb

Menü

- Serialisierbarkeit, Isolationsebenen
- Sperrverfahren und Deadlock-Behandlung
- Optimistische Synchronisation
- Weitere Verfahren
  - Prädikatsperren
  - Mehrversionenkonzept
  - Synchronisation von "Hot Spots"
- Lastkontrolle

Frage: Wieso läßt man Transaktionen überhaupt im Mehrbenutzerbetrieb und nicht im Batchbetrieb ablaufen?

Nachweis der Serialisierbarkeit

- Eine Historie ist eine Folge von *(TID, Operation, Datum)* Elementen
- Beispiel:

$$(T_1, W, x); (T_2, R, y); (T_3, W, x); (T_1, R, y)$$

- Abhängigkeitsgraph zwischen Transaktionen. Kante, wenn zwei Transaktionen auf dasselbe Objekt mit nicht reihenfolgeunbhängigen Operationen zugreifen.
- Die Historie ist serialisierbar, wenn der Abhängigkeitsgraph keine Zyklen enthält.

Isolationslevel nach ANSI SQL 92

Level	**Dirty Read**	**Fuzzy Read**	**Phantom**
Read Uncommitted	möglich	möglich	möglich
Read Committed	unmöglich	möglich	möglich
Repeatable Read	unmöglich	unmöglich	möglich
Serializable	unmöglich	unmöglich	unmöglich

Dirty Read

$$x = 2; (T_1, W, x = 1); (T_2, R, x = 1); (T_1, abort, x = 2)$$

Fuzzy Read

$$(T_1, R, x = 1); (T_2, W, x = 2); (T_1, R, x = 2)$$

Phantom

$$(T_1, R, avg(x) = 2); (T_2, I, x = 1); (T_1, R, avg(x) = 1.5)$$

Zweiphasen Sperrprotokoll (2PL)

Prinzip

- vor jedem Objektzugriff muß Sperre mit ausreichendem Modus angefordert werden
- blockieren, falls Sperre nicht erteilt werden kann
- Zweiphasigkeit: Wachstums- und Schrumpfphase. D.h. Sperrfreigabe kann erst beginnen, wenn alle Sperren gehalten werden.
- Spätestens beim Commit oder Abort werden alle Sperren freigegeben.

Striktes 2PL

Beobachtungen

- Kann man Ende der Wachstumsphase genau bestimmen?
- Fehler während der Schrumpfphase können zu *Dirty Read* führen.

Striktes 2PL

- Freigabe aller Sperren erst mit dem Commit (oder Abort).

N.B.

- Striktes 2PL erzielt Isolationslevel 2.
- Serialisierbarkeit (Ausschluß von Phantomen) wird nur bei *Prädikatsperren* oder anderen besonderen Maßnahmen erreicht.

RX-Sperrverfahren

- Fordere R Sperre zum Lesen an.
- Fordere X Sperre zum Schreiben an.
- Ggf. Upgrade einer R zu einer X Sperre, wenn Objekt nach einem Lesen geschrieben werden soll.
- Kompatibilitätsmatrix:

angeforderter Modus \ aktueller Modus	NL	R	X
R	+	+	-
X	+	-	-

RUX-Sperrverfahren

- Sperrkonversionen führen oft zu Deadlocks

  $$(T_1, R, x); (T_2, R, x); (T_1, W, x); (T_2, W, x)$$

- Erweitertes Sperrverfahren:
  - U-Sperre für Lesen mit Änderungsabsicht
  - Konversion $U \rightarrow X$ bei Änderung
  - Konversion $U \rightarrow R$ ansonsten

		aktueller Modus			
		NL	R	U	X
angeforderter Modus	R	+	+	-	-
	U	+	+	-	-
	X	+	-	-	-

Frage: Wieso ist die Matrix unsymmetrisch?

RAX Sperrverfahren

- Änderungen erfolgen in temporärer Kopie sowie mit einer A Sperre
- Paralleles Lesen der gültigen Version wird zugelassen
- A Sperre bewirkt Serialisierung der Schreibvorgänge
- Beim Commit ist Konversion von $A \rightarrow X$ notwendig; dabei muß ggf. auf Commit oder Abort von Lesern gewartet werden.
- Erhöhte Parallelität aber auch andere Serialisierungsreihenfolge.

		aktueller Modus			
		NL	R	A	X
angeforderter Modus	R	+	+	+	-
	A	+	+	-	-
	X	+	-	-	-

RAC Verfahren

angeforderter Modus \ aktueller Modus	NL	R	A	C
R	+	+	+	+
A	+	+	-	-
C	+	+	-	-

- Änderungen erfolgen in lokalen Kopien.
- A Sperren zur Änderung erforderlich.
- Bei Commit, $A \rightarrow C$ Konvertierung.
- C Sperre zeigt Existenz zweier gültiger Objektversionen an:

T_1	A(y)		A(z)	$A \rightarrow C$			
T_2		R(z)			R(y)		

- Auswahl der richtigen Version erforderlich.
- C Sperre wird erst freigegeben, wenn letzter Leser fertig ist.
- Vorteil: Leseanforderungen werden nie blockiert.
- Nachteil: Schreiber müssen bei gesetzter C Sperre auf alle Leser der alten Version warten.
- Erster Schritt in Richtung Mehrversionensperrverfahren.

Mehrversionensperrverfahren

- Jede Änderung erzeugt neue Objektversion.
- Keine Blockierungen und Rücksetzungen für Lesetransaktionen; dafür ggf. Zugriff auf veraltete Objektversionen.
- Änderungstransaktionen werden durch ein allgemeines Verfahren (z.B. RX-Sperren oder optimistisch) synchronisiert.
- Vorteil: erheblich weniger Synchronisationskonflikte.
- Nachteil: Aufwand für Versionsverwaltung, Auffinden von Version und Garbage Collection.

Zugriff auf Objektversionen

- TNC zählt monoton wachsend abgeschlossene Schreibtransaktionen
- Transaktionen merken sich beim Start aktuellen TNC:
  BOT-VNR = TNC.
- Zugriff erfolgt auf die jüngste Objektversion mit Versionsnummer $\leq$ BOT-VNR.
- Beim Commit einer Schreibtransaktion erhalten alle geschriebenen Objekte als Zeitstempel TNC.
- Eine Objektversion wird freigegeben (garbage collected), wenn es nachfolgende Version gibt und die Version von keiner aktuellen Transaktion mehr benötigt wird (i.e., Versionsstempel der nachfolgenden Version ist kleiner als das BOT-VNR aller aktuellen Transaktionen).
- Speicherorganisation: aktuelle Version und Verkettung im Versionenpool.

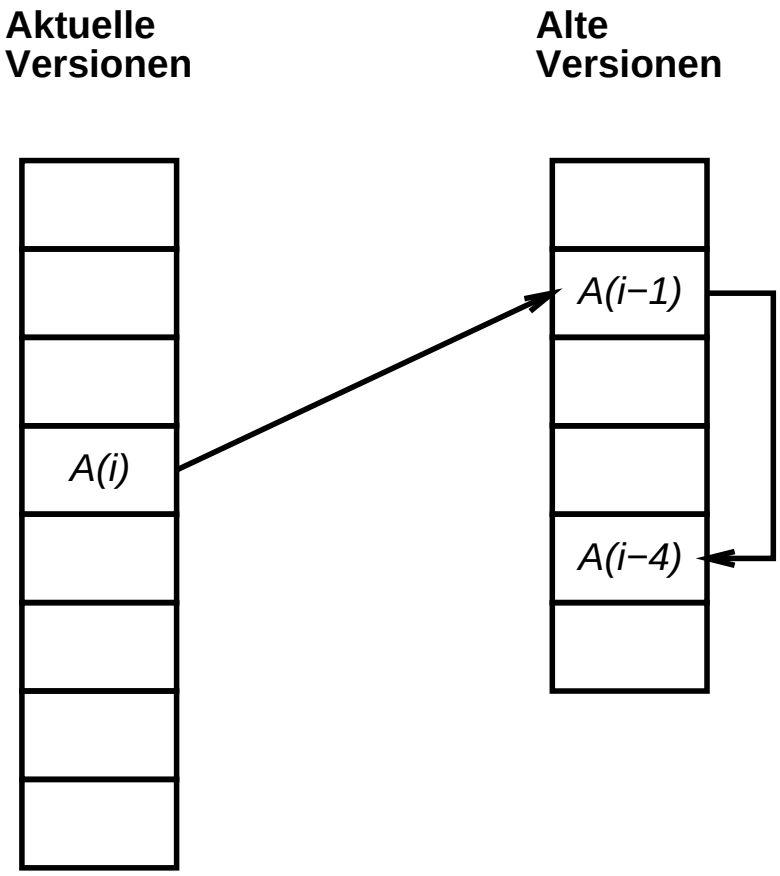
Beispiel

Ausgangszustand

- alle Objekte sind in Version 0
- TNC = 0

Ereignis	**Aktion**
T_1: BOT	BOT-VNR = 0
T_2: BOT	BOT-VNR = 0
T_1: R(X)	liest X(0)
T_2: W(X)	erzeugt neue Version für X
T_3: BOT	BOT-VNR = 0
T_1: R(X)	liest Version 0
T_2: Commit	TNC = 1, fixiert X(1)
T_3: R(X)	liest X(0) (!)
T_1: R(X)	liest X(0)
T_1: W(Y)	erzeugt neue Version für Y
T_1: Commit	TNC = 2, fixiert Y(2)
T_4: BOT	BOT-VNR = 2
T_4: R(X)	liest X(1)
T_4: R(Y)	liest Y(2)
T_3: R(Y)	liest Y(0)
T_3: Commit	löscht X(0), Y(0)
T_4: Commit	–

Versionsverwaltung



Logische Sperren (Prädikatsperren)

Zur Erinnerung: Phantomproblem.

$(T_1, R, \text{avg}(\text{salary}, \text{age} > 50))$;
$(T_2, \text{Insert}, (\text{John}, 60, 10\text{K}))$;
$(T_1, R, \text{avg}(\text{salary}, \text{age} > 50))$

Prädikatsperren

- Sperren der Form: (Relation, Prädikat, Modus)
  z.B. (Emp, age>50, R)
- Wäre eine elegante Lösung für das Phantomproblem; kein Aufwand für einzelne Objektsperren.
- Bei inkompatiblen Sperrmodi muß das Prädikat überlappungsfrei sein.
- Überlappungsfreiheit kann man einfach überprüfen wenn die Prädikate auf denselben Dimensionen arbeiten.

Prädikatsperren (ctd.)

- Im allgemeinen ist Überlappungsfreiheit nicht entscheidbar.
  - T_1 liest alle *Emp* mit $age > 50$.
  - T_2 löscht alle *Emp* mit $dept = 4711$.
- In diesem Fall muß man also doch physisch sperren. Oder "pessimistisch" Überlappung annehmen, wenn es unentscheidbar ist.
- Pessimismus wird oft mit schlechter Parallelität bestraft.

Andere Lösung für Phantome

Precision Locks

- Prädikatsperren nur für Leseanforderungen.
- Überprüfe Schreiber auf Kompatibilität mit vergangenen Lesern.
- Überprüfe Leser auf Kompatibilität mit vergangenen Schreibern.
- Literatur: Jordan et al. SIGMOD 1981

Key Value Locking

- Nutzt B∗ Bäume auf Schlüssel aus.
- Funktioniert durch Bereichssperren auf dem Schlüssel.
- (Wie genau solche Bereichssperren mit einem B∗ Baum verwaltet werden erläutern wir nächste Woche.)
- Literatur: Mohan et al. VLDB 1990.

Semantische Sperren

- Sperren auf der Ebene von Operationen.
- Beispiel: eine "Überweisung" beißt sich nicht mit einer anderen "Überweisung", obwohl beide Operationen evtl. dasselbe Konto schreiben.
- Benutzerdefinierte Sperrkompatibilitätstabellen auf Objektebene:

	überweisen	kontostand
überweisen	+	-
kontostand	-	+

- Semantisches Sperren Erhöht Parallelität.
- Semantisches Sperren nur möglich in objektorientierten Systemen und wenn die Operationen auf einem Objekt *atomar* sind.
- Semantisches Sperren setzt *logisches Logging* voraus.
- **Bitte semantisches Sperren nicht mit Prädikatsperren verwechseln.**
- Mehr zu semantischen Sperrverfahren in OO-OR Vorlesung.

Implementierung der Sperrtabelle

- Matrixorganisation; Zugriff über Hashtabellen (siehe Tafel)
- Zugriff über Objektidentifikator (Schlüssel): notwendig, um Sperranforderung zu erfüllen
- Zugriff über TID (Transaktionsid): notwendig, um alle Sperren einer Transaktion bei Commit oder Abort freizugeben
- Der Zugriff auf die Einträge der Sperrtabelle muß mit Semaphoren synchronisiert werden.
- Frage: Unter welchen Umständen reichen Semaphore auf die Anker der Objektidentifikatorhashtabelle?

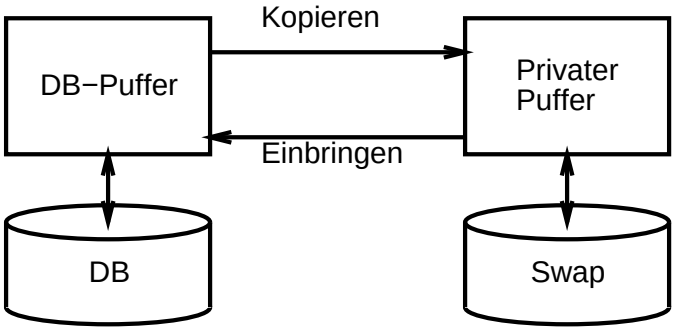
Optimistische Verfahren

Grundidee: 3-phasige Verarbeitung

- **Lesephase**
  - Durchführung der Operationen einer Transaktion.
  - Änderungen werden in einem privaten Puffer durchgeführt.
- **Validierungsphase**
  - Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer anderen Transaktion aufgetreten ist.
  - Konfliktauflösung durch Zurücksetzen der Transaktion.
  - Es wird in der Regel nur eine Transaktion gleichzeitig validiert.
- **Schreibphase** (bei positiver Validierung)
  - Propagieren der Änderungen

Optimistische Verfahren

Pufferverwaltung



Bewertung

- **Vorteile**
  - einfaches Rollback wegen NOSTEAL
  - keine Deadlocks
- **Nachteile**
  - Ressourcenverschwendung und unglückliche Benutzer durch mehr Rollbacks
  - Gefahr des Verhungerns von Transaktionen

Validierung bei optimistischen Verfahren

Voraussetzungen

- jede Transaktion merkt sich ihren *Read Set (RS)* und ihren *Write Set (WS)*
- eine Transaktion darf nur abgeschlossen werden, wenn sie alle Änderungen von zuvor abgeschlossenen Transaktionen gesehen hat
- N.B.: die Validierungsreihenfolge bestimmt die Serialisierungsreihenfolge

Backward Oriented Validierung BOCC

- Schließe T ab, nur wenn für alle T_j, die seit dem BOT von T abgeschlossen haben, gilt:

$$\mathrm{RS}(T) \cap \mathrm{WS}(T_j) = \emptyset$$

Nachteile von BOCC

- unnötige Rollbacks wegen ungenauer Konfliktanalyse (BOCC+ merkt sich zusätzlich Zeitstempel, um diesen Effekt zu vermeiden)
- Aufbewahren der Write-Sets beendeter Transaktionen notwendig
- hohe Anzahl von Vergleichen beim Validierung (i.e., $\cap$ Berechnung)
- Rücksetzung erst bei der Validierung möglich
- es kann nur die validierende Transaktion zurückgesetzt werden; Starvation möglich
- hohes Rollback Risiko bei langen Transaktionen

Forward Oriented Validierung FOCC

Prinzip

- Update-Transaktion T validiert gegen laufende Transaktionen T_i

$$\mathrm{WS}(T) \cap \mathrm{RS}(T_i) = \emptyset$$

- Setze T oder T_i zurück.

Vorteile

- Auswahl, wen man zurücksetzt.
- Keine unnötigen Rücksetzungen.
- Frühzeitige Rücksetzung möglich.
- Keine Aufbewahrung von Write-Sets.
- In der Regel geringerer Validierungsaufwand als bei BOCC.

Nachteile

- Immer noch nicht das Paradies: hohe Rollbackraten.
- Es existieren noch keine besonderen Ideen für Indexe.
- Man muß modifizierte Objekte während der Validierung und bis sie geschrieben worden sind, sperren.

Kombination: Optimistisch mit Sperren

Ziel:

Kombiniere Vorteile von optimistischen und pessimistischen Sperrverfahren.

Möglicher Ansatz

- Führe BOCC oder FOCC per Default durch.
- Falls Validierung einer Transaktion scheitert, dann starte Transaktion neu. Fordere aber gleich alle Sperren für die Transaktion an.
- Vorteil 1: Verhindert (meistens) Deadlocks.
- Vorteil 2: Zweiter Lauf (meistens) sehr schnell, da alles im Hauptspeicher ist.
- Nachteil 1: Komplizierte Implementierung. (Man muß beides realisieren.)
- Nachteil 2: Evtl. werden zuviele und/oder die falschen Sperren im zweiten Lauf angefordert. Genau kann man den zweiten Lauf ja nicht vorhersagen.

Weitere Möglichkeiten

- spezielle Verfahren für Hot Spots

Synchronisation von Hot Spots

- Hot Spots: meist numerische Felder mit aggregierten Informationen; z.B. Anzahl freier Parkplätze.
- Einfachste Lösung der Sperrprobleme: Vermeidung solcher Felder beim DB-Entwurf.
- Alternative: Semantisches Sperren (falls möglich)
- IMS Fast Path Ansatz: besondere Methoden auf Hot Spots
  - *Verify:* #Plätze $\geq$ Anforderung
  - *Modify:* #Plätze = #Plätze - Anforderung
  - Quasi-optimistische Synchronisation.
  - Zunächst werden keine Sperren gesetzt.
  - *Verify* und *Modify* werden nicht direkt vorgenommen sondern in "To-do" Listen eingetragen.
  - Beim Commit findet eine Validierungs- und Schreibphase statt.
    - Setze Sperren für die Dauer der Commit Behandlung.
    - Überprüfe *Verify* Prädikate. Abbruch, falls nicht erfüllt.
    - Führe *Modify* durch.
  - Geringe Konfliktgefahr. Klappt nur, wenn *Verify* sehr, sehr selten scheitert.

Weitere Verfahren

Zeitstempelverfahren

- Alternatives Synchronisationsverfahren
- Idee: Transaktionen erhalten Zeitstempel beim BOT; Serialisierung von Operationen auf Objekten wird anhand dieses Zeitstempels durchgeführt.
- Nur witzig in verteilten Systemen; aber auch dort kaum eingesetzt.
- Details in jedem gängigen Lehrbuch.

Hierarchische Sperrverfahren

- Sperren in feinen Granulaten möglich, um viel Parallelität zu ermöglichen.
- Möglichst niedriger Aufwand bei Sperren in groben Granulaten.
- Idee: erweiterte Sperrtabelle mit "Intention Locks".
- **Sehr, sehr wichtig!**
- Details in Einführungsvorlesung.
- N.B. Granularitätsaspekte sehr einfach bei optimistischen Synchronisationsverfahren handhabbar.

Deadlock-Behandlung

- Deadlockvermeidung durch "Preclaiming" oder "Ordnung auf Datenobjekte" (in der Regel ist beides nicht realisierbar).
- Lastkontrolle zur Verringerung der Anzahl der Deadlocks.
  (Wird in der nächsten Woche ein bißchen ausführlicher besprochen.)
- Deadlock Detection: Modelliere Wartebeziehung zwischen Transaktionen durch Graph. Deadlock bei Zyklus im Wartegraph. Verschiedene Heuristiken zum Auflösen von Deadlocks (z.B. breche jüngste Transaktion ab.
- Details in Einführungsvorlesung. Etwas interessantere Fälle in Verteilte DBMS Vorlesung.
- N.B. Deadlock Detection bei optimistischen Synchronisationsverfahren nicht notwendig.

Zusammenfassung: Synchronisation

- Vermeidung von Anomalien:
  - zu ändernde Objekte werden dem Zugriff aller anderen Transaktionen entzogen
  - zu lesende Objekte werden vor Änderungen geschützt
- Standardverfahren: striktes hierarchisches Zweiphasen-Sperrprotokoll
  - mehrere Sperrgranulate
  - Deadlock-Erkennung
  - besondere Tricks zur Reduzierung von Deadlocks (z.B. RUX)
  - besondere Tricks bei Hot Spots
  - Vermeidung von Phantomen sehr aufwendig.
- Effiziente Implementierung der Sperrtabelle ist fieselig und kritisch!
- Erhöhung des Durchsatzes (der Parallelität) durch niedrigeren Isolationslevel in vielen Anwendungen in Ordnung.

- Änderungen auf Objektkopien und Nutzung mehrerer Objektversionen in Kombination mit "optimistischen" Ansätzen wird immer populärer (z.B. Oracle); aber Vorsicht:
  - Bleibt Serialisierbarkeit wirklich gewahrt?
  - Wie groß ist der Hauptspeicherverbrauch? Wie hoch ist der Aufwand, die Kopien zu erzeugen.