Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Datenbanksysteme II

Transaktionsmanagement

(Kapitel 19)

9.7.2007

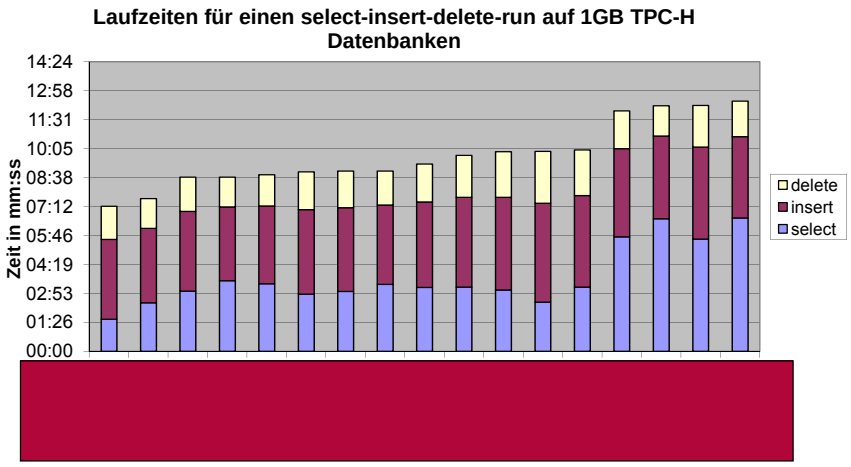
Felix Naumann

Ankündigungen

- Ergebnisse des TPC-H Benchmarks
- Bachelorprojekt „Datenfusion – Konsolidierung widersprüchlicher Daten
  - mit der FUZZY! Informatik AG
- Workshop zur Datenreinigung
  - mit der FUZZY! Informatik AG
- Bachelorprojekt „HighQ - Informationsintegration mit dem IBM Information Server"
  - mit IBM F&E Böblingen
- Veranstaltungen im nächsten Semester
  - Bachelorseminar „Beauty is our Business"
  - Bachelorseminar „www.ProminentPeople.info"
  - VL Datenbanksysteme I
  - VL Data Warehouses (Master)
  - Masterseminar „Schema Matching"

Ergebnisse des TPC-H Benchmarks

3



Bachelorprojekt
Datenfusion – Konsolidierung widersprüchlicher Daten

4

- Anwendung entwickeln, die Unternehmen bei der Datenfusion unterstützt
  - Partitionierung von Duplikatgruppen in Gruppen vergleichbarer Konfliktsituationen
  - Bereitstellen einer Vielzahl von teils selbst entworfenen Konfliktlösungsfunktionen und deren effizienter Implementierung
  - Visuelle Darstellung von Duplikatgruppen und manuelle Konfliktlösung darin
  - Automatische Auswahl von Konfliktlösungsfunktionen pro Attribut
  - Effiziente Fusion großer Datenbestände
  - Interaktion mit anderen FUZZY! Produkten, insbesondere FUZZY!Double
- Zusammenarbeit mit der FUZZY! Informatik AG

5

Workshop "Datenreinigung" für Studenten und Doktoranden

Prof. Felix Naumann
FUZZY! Informatik AG

8. Oktober - 10. Oktober 2007 Neu: Mo - Mi
(Mo. – Mi. direkt vor dem Wintersemester)

Innerhalb eines Unternehmens werden Kundendaten häufig in unterschiedlichen Systemen gehalten. Die Gründe dafür können in der Struktur des Unternehmens (getrennte Sparten), in unterschiedlichen Vertriebskanälen oder in einer Unternehmensfusion liegen. Um eine einheitliche Sicht auf den Kunden zu bekommen, müssen die Daten aus diesen Systemen zusammengeführt werden. Ein wichtiges Ziel ist dabei die automatische Erkennung von Dubletten, d.h. die Tatsache, dass ein Kunde in mehreren Systemen vorkommt, also in mehreren Beziehungen zum Unternehmen steht.

Sie sollen erkennen, welche Arten von Problemen beim Zusammenführen von Datenbeständen auftreten, welche Probleme sich mit einfachen Mitteln (SQL, Skripte, Text-Editor, etc.) lösen lassen und welche nicht. In praktischer Teamarbeit implementieren Sie Algorithmen zur Dublettenerkennung für große Datenmengen (1 Mio. Kundendatensätze). Das Team mit den meisten richtig gefundenen Dubletten gewinnt! Die in den beiden ersten Tagen gewonnenen Erkenntnisse und Lösungen sollen am Abschlusstag präsentiert werden.

Weitere Informationen und Programm: http://www.hpi.uni-potsdam.de/naumann/

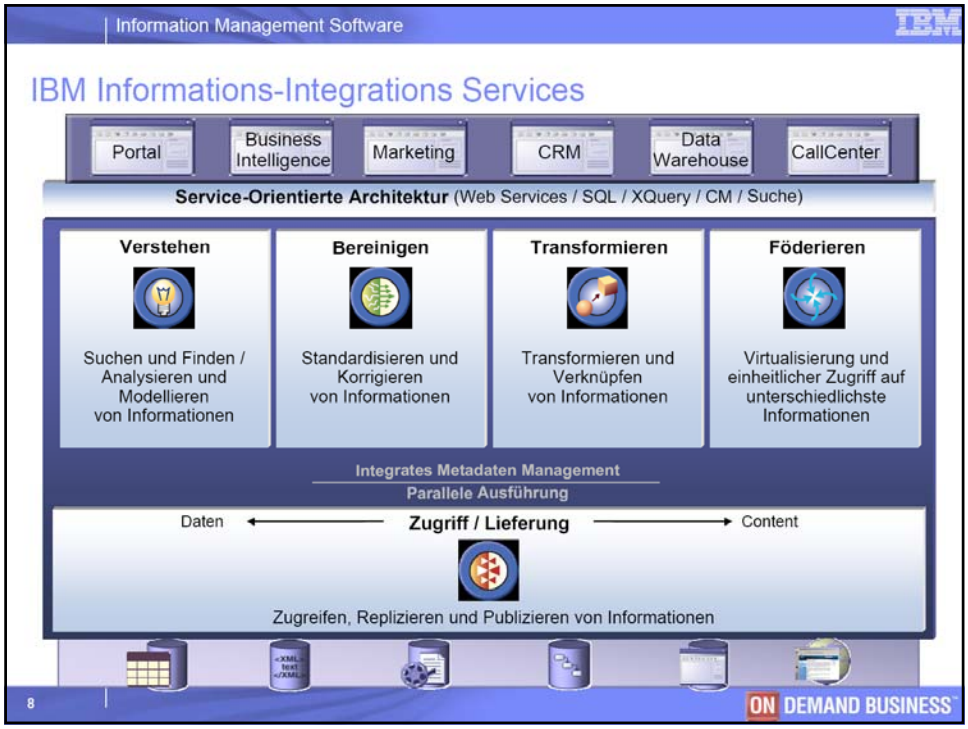
Anmeldung

- Formlose Anmeldung per Email bis zum 25. September an office-naumann@hpi.uni-potsdam.de.
- Es können maximal 20 Teilnehmer (Bachelor- und Master-Studenten und Doktoranden) mitmachen.

6

Bachelorprojekt
HighQ - Informationsintegration mit dem IBM Information Server

- Donnerstag: Gewinn eines IBM Shared University Research Grant
  - Einrichtung eines Information Server Lab am HPI
  - 2x Database server: Intel Xeon 5130, 8GB RAM, 5x73GB HDD RAID
  - 10x Developer client: Intel Core 2 Duo E6400, 3GB RAM, 160GB HDD, 19" TFT Display
- IBM Information Server
  - Im Kern: ehemals Ascential Produkte
  - ETL – Extract, Transform, and Load
  - DataStage, ProfileStage, QualityStage, SurviveStage
  - Föderiert: WebSphere Information Integrator
- SOA: Integration as a service



Bachelorprojekt

„HighQ - Informationsintegration mit dem IBM Information Server"

HPI Hasso Plattner Institut

8

- Erkenntnis: Wissenslücke zwischen ETL-Produkten und der Forschung
- Idee: Erweiterung des IIS mit Methoden der Forschung
  - Duplikaterkennung
  - Data Profiling
  - Data Fusion
- Use cases
  - Kooperation zwischen IBM und SAP
  - Customer Relationship Management

Bachelorseminar
Beauty is our Business

- *... Wenn wir uns klarmachen, dass der Kampf gegen Chaos, Durcheinander, und unbeherrschte Kompliziertheit eine der größten Herausforderungen der Informatik ist, müssen wir zugestehen: Beauty is our Business.*

  Edsger W. Dijkstra, 1978
- Termine
  - Einführung
  - Wissenschaftliche Texte lesen
  - Literaturkritik / Diskussion
  - Vortragstechniken
  - Vorträge: Mariposa und Fagins Algorithmus
  - Einführung in LaTeX
  - Vorträge: Enough Already in SQL und Sorted Neighborhood
  - Vorstellung der Gliederungen und Tipps zur Ausarbeitung
  - Vorträge: Source Capabilities und Data Mining

Bachelorseminar
www.ProminentPeople.info

- Idee
  - Personensuchmaschine / -datenbank
    - Aufgrund von Nachrichten
  - Mit Metadaten: Insbesondere Alter und Titel
    - Anfrage nach Alter (mit Widersprüchen)
    - Anfrage nach Geburtstag (aus statistischer Analyse
    - Beziehungen zwischen Personen
    - uvam.
- Zwei Themenkomplexe
  - Named Entity Recognition (NER)
  - Datenanalyse
- Vortrag und Implementierung
  - Domain ist bereits reserviert!

Masterveranstaltungen

- VL Data Warehouses
  - Architektur zur Integration von Unternehmensdatenbeständen
    - Weit verbreitet
  - Mehrdimensionale Modellierung
    - Star Schema
  - OLAP Anfragen
  - Optimierung
- SE „Schema Matching“
  - Korrespondenzen zwischen Schemata und Ontologien finden
  - Automatisiert
  - Label-basiert: Analyse der Schemata
  - Instanz-basiert: Analyse der zugehörigen Daten

Überblick

- Transaktionsmanagement aus DBS I
- Serialisierbarkeit und Recovery
- View-Serialisierbarkeit
- (Deadlocks (Verklemmungen))
- (Transaktionen in verteilten Datenbanken)
- (Verteiltes Commit)
- (Verteilte Sperren)
- (Langdauernde Transaktionen)

Die Transaktion

13

- Eine Transaktion ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten (eventuell veränderten) Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.
- Probleme im Mehrbenutzerbetrieb
  - Inkonsistentes Lesen: Nonrepeatable Read
  - Abhängigkeiten von nicht freigegebenen Daten: Dirty Read
  - Das Phantom-Problem
  - Verlorengegangenes Ändern: Lost Update

ACID

14

Atomicity (Atomarität)

- Transaktion wird entweder ganz oder gar nicht ausgeführt.

Consistency (Konsistenz oder auch Integritätserhaltung)

- Datenbank ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand.

Isolation (Isolation)

- Nutzer, der mit einer Datenbank arbeitet, sollte den Eindruck haben, dass er mit dieser Datenbank alleine arbeitet.

Durability (Dauerhaftigkeit / Persistenz)

- Nach erfolgreichem Abschluss einer Transaktion muss das Ergebnis dieser Transaktion „dauerhaft" in der Datenbank gespeichert werden.

Isolationsebenen

15

- **`read uncommitted`**
  - Schwächste Stufe: Zugriff auf nicht geschriebene Daten
  - Statistische Transaktionen (ungefährer Überblick, nicht korrekte Werte)
  - Keine Sperren: effizient ausführbar, keine anderen Transaktionen werden behindert
- **`read committed`**
  - Nur Lesen endgültig geschriebener Werte, aber nonrepeatable read möglich
- **`repeatable read`**
  - Kein nonrepeatable read, aber Phantomproblem kann auftreten
- **`serializable`**
  - Garantierte Serialisierbarkeit (default)
  - Transaktion sieht nur Änderungen, die zu Beginn der Transaktion committed waren (plus eigene Änderungen).

Schedules

16

Ein Schedule ist eine geordnete Abfolge wichtiger Aktionen, die von einer oder mehreren Transaktionen durchgeführt werden.

- Wichtige Aktionen: READ und WRITE eines Elements
- „Ablaufplan" für Transaktion, bestehend aus Abfolge von Transaktionsoperationen

Schritt	T_1	T_3
1.	**BOT**	**BOT**
2.	read(A,a_1)	read(A,a_2)
3.	$a_1 := a_1 - 50$	$a_2 := a_2 - 100$
4.	write(A,a_1)	write(A,a_2)
5.	read(B,b_1)	read(B,b_2)
6.	$b_1 := b_1 + 50$	$b_2 := b_2 + 100$
7.	write(B,b_1)	write(B,b_2)
8.	**commit**	**commit**

Schedules

Serieller Schedule

Schritt	T_1	T_2
1.	**BOT**	
2.	read(*A*)	
3.	write(*A*)	
4.	read(*B*)	
5.	write(*B*)	
6.	**commit**	
7.		**BOT**
8.		read(*C*)
9.		write(*C*)
10.		read(*A*)
11.		write(*A*)
12.		**commit**

Serialisierbarer Schedule

Schritt	T_1	T_2
1.	**BOT**	
2.	read(*A*)	
3.		**BOT**
4.		read(*C*)
5.	write(*A*)	
6.		write(*C*)
7.	read(*B*)	
8.	write(*B*)	
9.	**commit**	
10.		read(*A*)
11.		write(*A*)
12.		**commit**

Beispiel: Alfons Kemper (TU München)

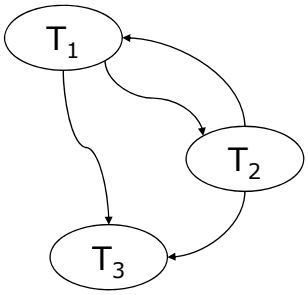
Konfliktserialisierbarkeit

- Idee: So lange nicht-konfligierende Aktionen tauschen bis aus einem Schedule ein serieller Schedule wird.
  - Falls das klappt ist der Schedule serialisierbar.
- Zwei Schedules *S* und *S'* heißen konfliktäquivalent, wenn die Reihenfolge aller Paare von konfligierenden Aktionen in beiden Schedules gleich ist.
- Ein Schedule *S* ist genau dann konfliktserialisierbar, wenn *S* konfliktäquivalent zu einem seriellen Schedule ist.
- Schedule:
  $r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$
- Variante 1: Serieller Schedule T_1T_2
  $r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$
- Variante 2: Serieller Schedule T_2T_1
  $r_2(A)w_2(A)r_2(B)w_2(B)r_1(A)w_1(A)r_1(B)w_1(B)$

Graph-basierter Test

19

T_1	T_2	T_3
r(y)		
		r(u)
	r(y)	
w(y)		
w(x)		
	w(x)	
	w(z)	
		w(x)



$S = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)$

Sperren

20

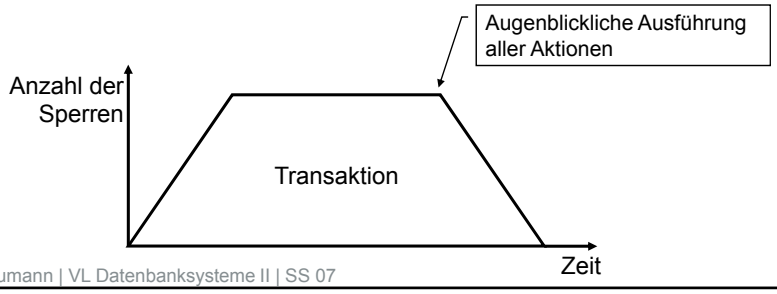
Idee: Transaktionen müssen zusätzlich zu den Aktionen auch Sperren anfordern und freigeben.

- Bedingungen
  - **Konsistenz** einer Transaktion
    - Lesen oder Schreiben eines Objektes nur nachdem Sperre angefordert wurde und bevor die Sperre wieder freigegeben wurde.
    - Nach dem Sperren eines Objektes muss später dessen Freigabe erfolgen.
  - **Legalität** des Schedules
    - Zwei Transaktionen dürfen nicht gleichzeitig das gleiche Objekt sperren.
- Zwei neue Aktionen
  - $l_i(X)$: Transaktion i fordert Sperre für X an.
  - $u_i(X)$: Transaktion i gibt Sperre auf X frei.
- Konsistenz: Vor jedem $r_i(X)$ oder $w_i(X)$ kommt ein $l_i(X)$ (mit keinem $u_i(X)$ dazwischen) und ein $u_i(X)$ danach.
- Legalität: Zwischen $l_i(X)$ und $l_j(X)$ kommt immer ein $u_i(X)$

2-Phasen Sperrprotokoll

2-Phase-Locking (2PL): Einfache Bedingung an Transaktionen garantiert Konfliktserialisierbarkeit.

- Alle Sperranforderungen geschehen vor allen Sperrfreigaben
- Die Phasen
  - Phase 1: Sperrphase
  - Phase 2: Freigabephase
- Wichtig: Bedingung an Transaktionen, nicht an Schedule



Mehrere Sperrmodi

Idee: Mehrere Arten von Sperren erhöhen die Flexibilität und verringern die Menge der abgewiesenen Sperren.

- Sperren obwohl nur gelesen wird, ist übertrieben
  - Sperre ist dennoch nötig
  - Aber: Mehrere Transaktionen sollen gleichzeitig lesen können.
- Schreibsperre
  - *Exclusive lock*: $xl_i(X)$
  - Erlaubt auch das Lesen
- Lesesperre
  - *Shared lock*: $sl_i(X)$
- Kompatibilität
  - Für ein Objekt darf es nur eine Schreibsperre oder mehrere Lesesperren geben.
- Freigabe
  - Unlock: $u_i(X)$ gibt alle Arten von Sperren frei

Bedingungen

- Konsistenz von Transaktionen
  - Schreiben ohne Schreibsperre ist nicht erlaubt.
  - Lesen ohne irgendeine Sperre ist nicht erlaubt.
  - Jede Sperre muss irgendwann freigegeben werden.
- 2PL von Transaktionen
  - Wie zuvor: Nach der ersten Freigabe dürfen keine Sperren mehr angefordert werden.
- Legalität von Schedules
  - Auf ein Objekt mit einer Schreibsperre darf es keine andere Sperre einer anderen Transaktion geben.
  - Auf ein Objekt kann es mehrere Lesesperren geben.

Überblick

- Transaktionsmanagement aus DBS I
- Serialisierbarkeit und Recovery
- View-Serialisierbarkeit
- (Deadlocks (Verklemmungen))
- (Transaktionen in verteilten Datenbanken)
- (Verteiltes Commit)
- (Verteilte Sperren)
- (Langdauernde Transaktionen)

Motivation

25

Bisher waren Serialisierbarkeit und Logging orthogonale Konzepte.

- Logging stellt „blind" eine Datenbank wieder her
  - Alle committed Transaktionen
  - Serialisierbarkeit wird nicht berücksichtigt
- Serialisierbarkeit berücksichtigt Logging nicht
  - Serialisierbarkeit erlaubt ein Element zu schreiben
    - Abhängig von Sperren
    - Unabhängig vom Commit
  - Serialisierbarkeit erlaubt ABORT ohne bereits geschriebene Elemente zurückzusetzen!

Das Dirty-Data Problem

26

- Dirty data
  - Daten, die bereits geschrieben wurden,
    - Disk oder Hauptspeicher
  - aber Transaktion noch nicht committed
- Im Beispiel
  - T_2 liest inkonsistenten Zustand: A neu, B alt
  - Disk oder Puffer ist egal
  - Am Ende: A ≠ B
  - Undo T_2 (und T_1) wird nötig!

Dirty read!

T_1	T_2	A	B
		25	25
l(A); read(A,a_1)			
$a_1 := a_1 + 100$		125	
write(A,a_1); l(B); u(A)			
	l(A); read(A,a_2)		
	$a_2 := a_2 * 2$	250	
	write(A,a_2);		
	l(B); abgelehnt!		
read(B,b_1)			
ABORT			
u(B)			
	l(B); u(A); read(B,b_2)		
	$b_2 := b_2 * 2$		250
	write(B,b_2); u(B)		

Antwort auf Frage in VL

27

- Ein **Schedule** kann serialisierbar sein
  - Verhindert somit Fehler wie nonrepeatable read, dirty read, Phantom-Problem, Lost Update
  - Aber der Plan kann durch ein Abort „durchkreuzt werden."
    - Von ABORTs war nie die Rede…
  - D.h. Schedule (Plan) ist serialisierbar, aber **tatsächliche Ausführung** nicht.
- Problem „Abort" und Dirty Data
  - Wichtige Annahme: Daten werden erst geschrieben, wenn eine Transaktion sicher ist, kein ABORT auszuführen
    - Sowohl Hauptspeicher als auch Disk
  - Dirty Data trotz Serialisierbarkeit

Kaskadierendes Rollback

28

- Bei einem *Dirty Read* muss kaskadierendes Rollback erfolgen
- Falls Transaktion T abortet:
  - Suche Transaktionen, die von T geschriebene Werte gelesen hat.
  - Aborte diese
  - Rekursiv weiter
- Daten für Rollback
  - Logdatei
    - Falls undo oder undo/redo
  - Werte auf Disk (falls neue Werte noch nicht auf Disk geschrieben)

Rücksetzbare Schedules

29

- *Recoverable Schedules*
- Bei Recovery nach Logging: Die Menge der committed Transaktionen nach Recovery muss konsistent sein.
  - Falls T_1 nach Recovery als committed angesehen wird,
  - und falls T_1 einen von T_2 geschriebenen Wert verwendete,
  - dann muss auch T_2 nach Recovery als committed gelten
- Ein Schedule heißt „rücksetzbar", falls jede Transaktion erst committed nachdem alle Transaktionen committed haben, von denen sie gelesen hat.
- Notation im Schedule: c_i wenn T_i committet

Schedules – Beispiele

30

- Rücksetzbarer Schedule: S_1: $w_1(A)$, $w_1(B)$, $w_2(A)$, $r_2(B)$, c_1, c_2
  - T_2 liest B nach T_1 B geschrieben hat. Also committed T_2 auch erst nach T_1.
  - Serialisierbar?
    - Ja, sogar seriell
- Weiterer Schedule: S_2: $w_2(A)$, $w_1(B)$, $w_1(A)$, $r_2(B)$, c_1, c_2
  - Rücksetzbar? Serialisierbar?
    - Rücksetzbar (T_2 liest B nach T_1, committed aber auch danach)
    - Nicht serialisierbar! (Wo ist der Konflikt?)
- Weiterer Schedule: S_3: $w_1(A)$, $w_1(B)$, $w_2(A)$, $r_2(B)$, c_2, c_1
  - Rücksetzbar? Serialisierbar?
    - Nicht rücksetzbar (T_2 liest B nach T_1, committed aber eher)
    - Serialisierbar (sogar seriell)
- Rücksetzbarkeit und Serialisierbarkeit sind unabhängig.

Commit Reihenfolge

- Beispiele von eben
  - S_1: $w_1(A)$, $w_1(B)$, $w_2(A)$, $r_2(B)$, c_1, c_2
    - Rücksetzbar
  - S_3: $w_1(A)$, $w_1(B)$, $w_2(A)$, $r_2(B)$, c_2, c_1
    - Nicht Rücksetzbar
  - Einziger Unterschied: Commit-Reihenfolge
- Weitere Annahme um tatsächlich Rücksetzbarkeit zu erreichen:
  - Commit-Datensätze im Log erreichen Disk in der Reihenfolge, in der sie geschrieben wurden.

ACR Schedules

- ACR: *Avoid cascading rollbacks*
- Kaskadierende Rollbacks sind mitunter (natürlich) auch bei rücksetzbaren Schedules nötig
  - Aber teuer ☹
- ACR Schedules
  - Ein Schedule ist **ACR**, falls Transaktionen nur Werte von committed Transaktionen lesen.
- Reminder
  - Ein Schedule heißt **rücksetzbar**, falls jede Transaktion erst committed nachdem alle Transaktionen committed haben, von denen sie gelesen hat.
- ACR => Dirty read ist verboten
- Rücksetzbar => Dirty read ist erlaubt, TA darf aber noch nicht commiten
- „Committed" genauer: Commit-Datensatz im Log ist bereits auf Disk

ACR Schedules – Beispiel

33

- Vorige Schedules
  - S_1: $w_1(A)$, $w_1(B)$, $w_2(A)$, $r_2(B)$, c_1, c_2
  - S_2: $w_2(A)$, $w_1(B)$, $w_1(A)$, $r_2(B)$, c_1, c_2
  - S_3: $w_1(A)$, $w_1(B)$, $w_2(A)$, $r_2(B)$, c_2, c_1
  - ACR?
    - Alle nicht ACR!
- S_4: $w_1(A)$, $w_1(B)$, $w_2(A)$, c_1, $r_2(B)$, c_2
  - Rücksetzbar und ACR
- Beobachtung: ACR => rücksetzbar
  - Wenn eine Transaktion noch liest nach dem eine andere committed ist, kann sie auch erst danach aborten oder committen.

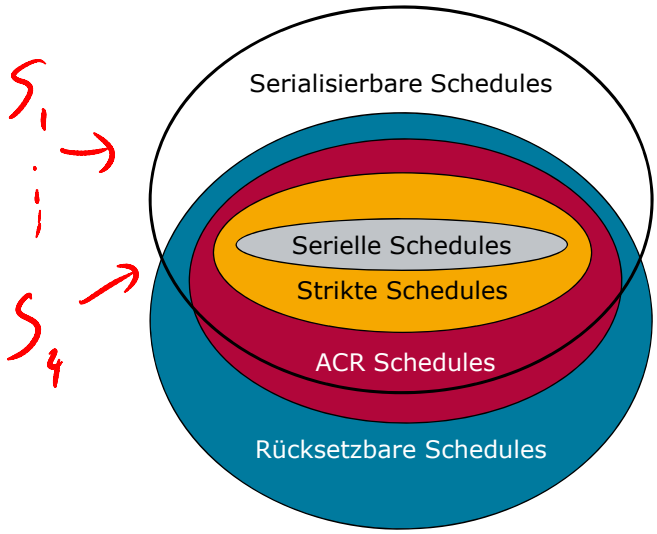
Strikte Schedules

34

- Sperrprotokolle dienen zur Einhaltung der Serialisierbarkeit
- Einhaltung von ACR durch verbessertes Sperrprotokoll
  - Striktes Sperren: Eine Transaktion gibt keine xl Sperren frei, bis sie committed oder aborted…
    - …und entsprechendes COMMIT oder ABORT im Log auf Disk geschrieben wurde.
- Strikter Schedule: Schedule, der den strikten Sperren folgt
- Eigenschaften
  - Strikt => ACR
    - T2 kann nichts xl-gesperrtes von T1 lesen. Sperre wird erst nach COMMIT freigegeben.
  - Strikt => Serialisierbar
    - Strikter Schedule ist äquivalent mit einem seriellen Schedule, an dem alle TAs augenblicklich zu ihrem COMMIT ausgeführt werden

Schedules – Beziehungen

35



Rollback für Blöcke

36

- Sperrgranularität: Blöcke
- Rollback einfach
  - Ohne Verwendung des Logs
- T habe xl auf Block A, schreibt neues A in Puffer und abortet.
- Annahme:
  - Blöcke aus uncommitted Transaktionen werden nicht auf Disk geschrieben.
- Rollback:
  - Block A ignorieren, also nicht auf Disk zurückschreiben.
  - Entsprechender Puffer wird als „frei" deklariert
    - Zum Bufferpool

Rollback für kleine Elemente

37

- Sperrgranularität: Tupel, Objekte, Attributwerte
- Problem
  - Ein Puffer enthält Änderungen mehrerer Transaktionen.
  - Eine abortet, die andere committet…
- Einfache Strategie, nämlich Puffer verwerfen, funktioniert nicht.
- Varianten
  1. Originalwert von A von Disk lesen und Pufferinhalt anpassen
  2. Bei undo oder undo/redo logging: Alten Wert aus Log lesen
     - Gleicher Ablauf wie bei recovery von uncommitted TAs
  3. Kleiner Hauptspeicherlog für Transaktionen. Aufbewahrung nur solange Transaktion aktiv.
- Nachteile: Zusätzliche I/Os bzw. Speicherverbrauch

Group Commit

38

- Wdh.: Rücksetzbarkeit
  - Ein Schedule heißt „rücksetzbar", falls jede Transaktion erst committed nachdem alle Transaktionen committed haben, von denen sie gelesen hat.
- Wdh.: Commit Reihenfolge
  - Commit-Datensätze im Log erreichen Disk in gleicher Reihenfolge wie sie geschrieben wurden.
- Wdh.: Striktes Sperren
  - Eine Transaktion gibt keine xl Sperren frei, bis sie committed oder aborted und entsprechendes COMMIT oder ABORT im Log auf Disk geschrieben wurde.
- Manchmal kann man sich Flush-log sparen bzw. Sperren früher aufheben.
- *Group commit*
  1. xl Sperren bereits aufheben wenn commit-Datensatz im Log ist (aber noch nicht auf Disk).
  2. Flush-log schreibt Log-Blöcke in Entstehungsreihenfolge

Group Commit – Beispiel

- Transaktion T1
  - Schreibt X; Schreibt COMMIT-Datensatz in Log
    - Log-Datensatz bleibt (zunächst) im Puffer
  - T1 gibt sämtliche Sperren ab.
- Transaktion T2
  - Liest X (potentiell dirty); Schreibt COMMIT-Datensatz in Log
    - Log-Datensatz bleibt auch im Puffer
    - Wegen Reihenfolgenerhaltung
- Recovery: T2 kann niemals als committed angesehen werden, wenn nicht auch T1 committed ist.
  - Fall 1: T1 und T2 Commit-Datensätze erreichen Disk nicht. Kein Problem
    - Beide werden bei Recovery abortet.
  - Fall 2: T1 ist committed, T2 nicht. Kein Problem
    - T2 hat nicht von uncommitted TA gelesen
    - T2 ist sowieso aborted
  - Fall 3: Beide Commit-Datensätze erreichen Disk. Kein Problem
    - Lesen von X von T2 war nicht dirty
- Fälle falls Reihenfolge nicht eingehalten wird?

Logisches Logging

- Probleme bei Logging
  - Logging verlangt die Speicherung alter, neuer oder beider Werte: Ganzer Block trotz vielleicht nur kleiner Änderung
    - Viel Redundanz
  - Rücksetzbarkeit verlangt viel: Sperren erst nach commit aufheben
- Logisches Logging hilft: Nur Änderungen auf Blöcke werden beschrieben
  - Hier nur Intuition!

Komplexität von logischem Logging

- Fall 1: Nur wenige Byte ändern sich
  - z.B. update eines Feldes fester Länge
  - Logging trivial: Veränderte Bytes und deren Position
- Fall 2: Beschreibung einfach, aber Auswirkungen überall im Block
  - Z.B. update eines Feldes variabler Länge
    - Datensatz ändert sich.
    - Andere Datensätze auf dem Block verschieben sich.
  - Neuer und alter Wert sehen verschieden aus.
  - Idee: Nicht die Byte-Struktur eines Blocks ist interessant, sondern dessen logischer Inhalt: Eine Tupelmenge
  - Logging der Änderungsoperation, nicht der Änderung selbst
- Fall 3: Änderung betrifft viele Bytes und weitere Änderungen verhindern Wiederherstellung völlig
  - Z.B. Logging für B-Baum Blöcke
- Probleme beim Recovery: Änderungsoperationen sind nicht idempotent.
  - Idee: Zu jeder Änderung wird die kompensierende Änderung gespeichert.
  - Recovery erzeugt nicht identische Datenbank aber äquivalente Datenbank.

Überblick

- Transaktionsmanagement aus DBS I
- Serialisierbarkeit und Recovery
- View-Serialisierbarkeit
- (Deadlocks (Verklemmungen))
- (Transaktionen in verteilten Datenbanken)
- (Verteiltes Commit)
- (Verteilte Sperren)
- (Langdauernde Transaktionen)

Wdh: Serialisierbarkeit

- Serieller Schedule
  - Schedule in dem Transaktionen hintereinander ausgeführt werden
- Serialisierbarer Schedule
  - Schedule dessen Effekt identisch zum Effekt eines (beliebig gewählten) seriellen Schedules ist.
- Konfliktserialisierbarer Schedule
  - Zwei Schedules *S* und *S'* heißen konfliktäquivalent, wenn die Reihenfolge aller Paare von konfligierenden Aktionen in beiden Schedules gleich ist.
  - Ein Schedule *S* ist genau dann konfliktserialisierbar, wenn *S* konfliktäquivalent zu einem seriellen Schedule ist.
  - Konflikt herrscht falls zwei Aktionen
    - das gleiche Datenbankelement betreffen und
    - mindestens eine der beiden Aktionen ein *write* ist.
  - Idee: So lange nicht-konfligierende Aktionen tauschen bis aus einem Schedule ein serieller Schedule wird.
- Problem: Konflikt ist übertrieben streng definiert.

Sicht-Serialisierbarkeit

- Konflikt-serialisierbar => Serialisierbar
- Sicht-Serialisierbar => Serialisierbar
  - Aber weniger streng
- Idee: Sichtserialisierbarkeit untersucht alle Verbindungen zwischen Transaktionen T und U, bei denen T schreibt was später U liest.
- Hintergrund: Wenn T und U das gleiche Element schreiben, ist dies kein Problem.
  - Write-Aktion kann flexibler positioniert werden.

Sicht-Äquivalenz

45

- Gegeben zwei verschiedene Schedules S_1 und S_2 für gleiche Transaktionen.
- Zusätzlich
  - Hypothetische Transaktion T_0, die initial jedes Element schreibt, das später gelesen wird.
  - Hypothetische Transaktion T_f, die am Ende jedes Element liest, das von einer oder mehr Transaktionen geschrieben wurde.
- Für jede Lese-Aktion (auch in T_f) kann man die dichteste vorige Schreib-Aktion im Schedule finden.
  - Spätestens T_0
  - Diese Aktion ist die „Quelle" der Lese-Aktion
- Wenn für S_1 und S_2 für jede Lese-Aktion die Quelle gleich ist, sind S_1 und S_2 **Sicht-äquivalent**.
- Wenn Schedule S Sicht-äquivalent zu einem seriellen Schedule ist, ist S **Sicht-serialisierbar**.

Sicht-Äquivalenz – Beispiel

46

- T_1: r1(A) w1(B)
- T_2: r2(B) w2(A) w2(B)
- T_3: r3(A) w3(B)

- Konfliktserialisierbar?
  - Nein! Wo sind die Konflikte?
  - Aber: T_1 und T_2 schreiben B-Werte, die sowieso verloren gehen.
  - Nur der T_3-Wert für B wird von (hypothetischem) T_f gelesen.
- Quellen der Read-Aktionen im Schedule?
  - r2(B): T_0; r1(A): T_2; r3(A): T_2
  - Lese-Aktion von A in T_f: T_2; Lese-Aktion von B in T_f: T_3
- Zugehöriger serieller Schedule?
  - T2, T1, T3
  - Probe: Sind die Quellen alle gleich?

Graphbasierter-Test

47

- Wiederholung: Graphbasierter Test für Konflikt-Serialisierbarkeit
  - Konfliktgraph *G(S) = (V,E)* von Schedule *S*:
    - Knotenmenge *V* enthält alle in *S* vorkommende Transaktionen.
    - Kantenmenge *E* enthält alle gerichteten Kanten zwischen zwei konfligierenden Transaktionen.
      - » Kantenrichtung entspricht zeitlichem Ablauf im Schedule.
  - Eigenschaften
    - *S* ist ein konfliktserialisierbarer Schedule gdw der vorliegende Konfliktgraph ein azyklischer Graph ist.

Graphbasierter-Test

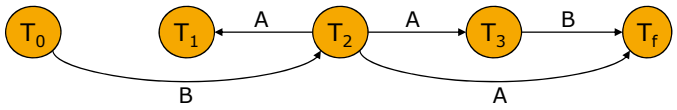
48

- „Polygraph" für Test auf Sicht-Serialisierbarkeit
  1. Knoten wie zuvor für jede Transaktion
     - Zusätzlich Knoten für T_0 und T_f
  2. Kante für jede Lese-Aktion von Quelle zur Leseaktion
  3. Spezielles Kantenpaar (Interventionskanten):
     - Sei T_j Quelle von $r_i(X)$
     - T_k habe (auch) ein $w_k(X)$
     - => T_k muss vor T_j oder nach T_i erscheinen
     - Gestrichelte Kanten von T_k -> T_j und T_i -> T_k.
     - Eine der beiden Kanten ist „echt", es ist aber egal welche (um Polygraph azyklisch zu machen).
     - Sonderfälle
       - » T_j ist T_0 => T_k kann nicht früher => nur eine Kante
       - » T_i ist T_f => T_k kann nicht später => nur eine Kante

Graphbasierter-Test – Beispiel

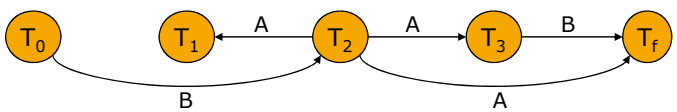
- T_1: r1(A) w1(B)
- T_2: r2(B) w2(A) w2(B)
- T_3: r3(A) w3(B)

- Zunächst Graph mit normalen Kanten
  - Quellen für jede Leseaktion



Graphbasierter-Test – Beispiel

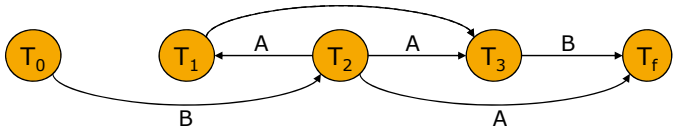
- T_1: r1(A) w1(B)
- T_2: r2(B) w2(A) w2(B)
- T_3: r3(A) w3(B)
- Nun Interventionskanten
  - Können sich Schreibaktionen zwischen eine Kante schieben?
  - Vorgehen pro Datenbankelement und dann pro Kante
    - Schreiber von A: T_0 und T_2
    - Kante T_2->T_1
      - » T_0 kann nicht verschoben werden
      - » T_2 ist schon an der Kante beteiligt
    - Ebenso für Kanten T_2->T_3 und T_2->T_f



Graphbasierter-Test – Beispiel

51

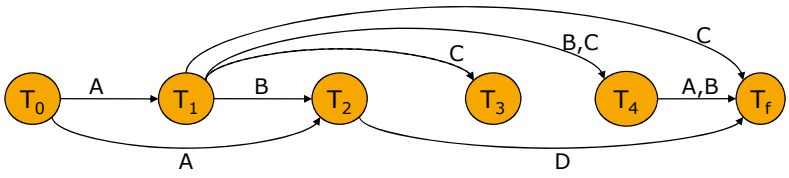
- T_1: r1(A) w1(B)
- T_2: r2(B) w2(A) w2(B)
- T_3: r3(A) w3(B)
- Nun Interventionskanten
  - Können sich Schreibaktionen zwischen eine Kante schieben?
  - Vorgehen pro Datenbankelement und dann pro Kante
    - Schreiber von B: T_0, T_1, T_2, T_3
    - Z.B. Kante T_0->T_2
      - » T_1 darf sich nicht zwischenschieben => Gestrichelte Kanten
      - » T_1->T_0 (sowieso nicht erlaubt) und T_2->T_1 (schon da)
    - Ebenso für Kanten T_3->T_f



Graphbasierter-Test – Beispiel

52

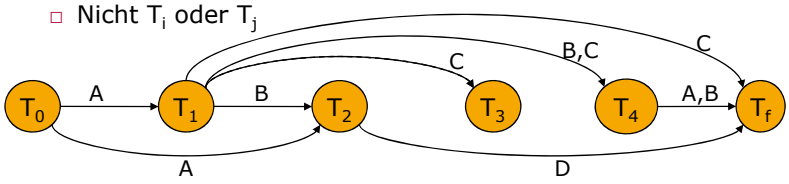
T1	T2	T3	T4
	r2(A)		
r1(A), w1(C)			
		r3(C)	
w1(B)			
			r4(A)
		w3(A)	
			r4(C)
	w2(D); r2(B)		
			w4(A); w4(B)



Graphbasierter-Test – Beispiel

53

- Interventionskanten
  - T_j Quelle von $r_i(X)$; T_k habe (auch) ein $w_k(X)$ => T_k muss vor T_j oder nach T_i erscheinen
  - Gestrichelte Kanten: T_k -> T_j und T_i -> T_k
- Nur berücksichtigen
  - Schreiber eines Elements, die Kante T_j -> T_i verursacht haben
  - Nicht T_0 oder T_f
  - Nicht T_i oder T_j

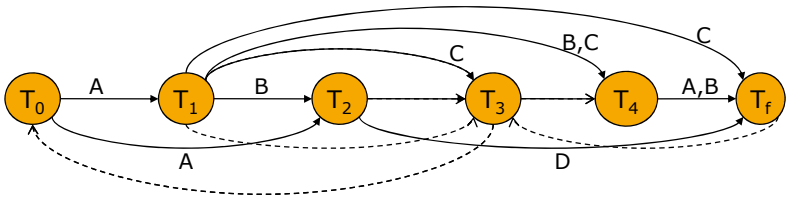


Graphbasierter-Test – Beispiel

54

T1	T2	T3	T4
	r2(A)		
r1(A), w1(C)			
		r3(C)	
w1(B)			
			r4(A)
		w3(A)	
			r4(C)
	w2(D); r2(B)		
			w4(A); w4(B)

Schreiber von A: T_0, T_3, T_4
z.B.: T_3 darf nicht zwischenT_4 und T_f
Schreiber von B: T_0, T_1, T_4
Schreiber von C: T_0, T_1
Schreiber von D: T_0, T_2

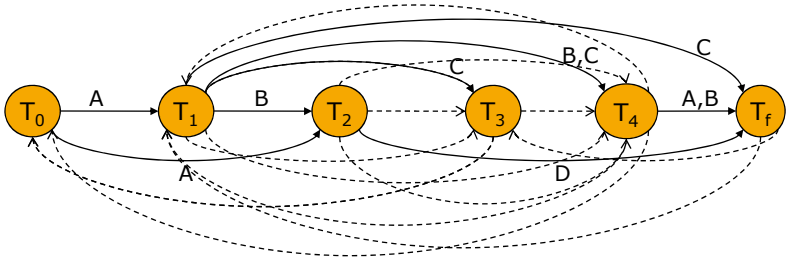


Graphbasierter-Test – Beispiel

55

T1	T2	T3	T4
	r2(A)		
r1(A), w1(C)			
		r3(C)	
w1(B)			
			r4(A)
		w3(A)	
			r4(C)
	w2(D); r2(B)		
			w4(A); w4(B)

Schreiber von A: T_0, T_3, T_4
Schreiber von B: T_0, T_1, T_4
Schreiber von C: T_0, T_1
Schreiber von D: T_0, T_2

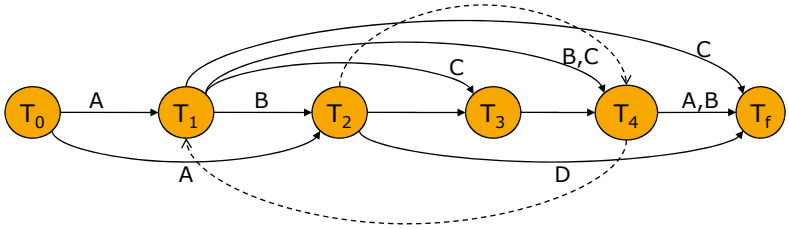


Graphbasierter-Test – Beispiel

56

T1	T2	T3	T4
	r2(A)		
r1(A), w1(C)			
		r3(C)	
w1(B)			
			r4(A)
		w3(A)	
			r4(C)
	w2(D); r2(B)		
			w4(A); w4(B)

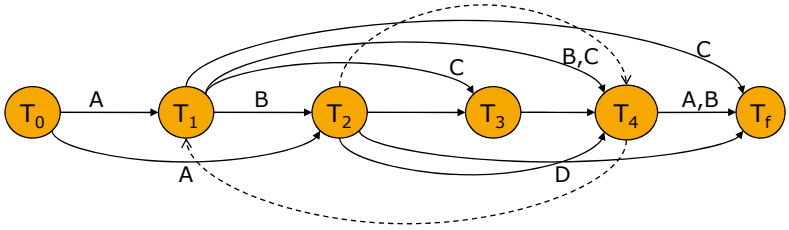
Schreiber von A: T_0, T_3, T_4
Schreiber von B: T_0, T_1, T_4
Schreiber von C: T_0, T_1
Schreiber von D: T_0, T_2



Graphbasierter Test

- Für jedes Interventionskantenpaar muss eine Wahl getroffen werden.
- Wähle so, dass Graph azyklisch wird (falls möglich).
- Graph ist azyklisch ⇔ Schedule ist sicht-serialisierbar
  - Wieder: Topologische Sortierung findet äquivalenten seriellen Schedule.
- Graph ist azyklisch => Schedule ist sicht-serialisierbar
  - Azyklisch => Topologische Sortierung ist möglich
  - und Schreiber sind nicht zwischen Leser und dessen Quelle
  - Und Schreiber sind vor Leser
  - => Leser–Quelle Verbindungen sind in gleicher Reihenfolge wie im seriellen Graph
- Graph ist azyklisch <= Schedule ist sicht-serialisierbar
  - selbst

Graphbasierter-Test – Beispiel



Frage: Welche Kante sollte gewählt werden?

Äquivalenter serieller Schedule: T_1, T_2, T_3, T_4