Datenbank-Programmierung

Kapitel 9: (Kurze) Theorie der Serialisierbarkeit

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2021

http://www.informatik.uni-halle.de/~brass/dbp21/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- Gegebene Schedules auf „Konflikt-Serialisierbarkeit“ prüfen.
- Die wesentlichen Vereinfachungen des theoretischen Modells nennen.
- Die Beziehung zwischen Konflikt-Serialisierbarkeit und semantischer Serialisierbarkeit erklären.

Inhalt

1. Transaktionen
2. Schedules
3. Serialisierbarkeit

Transaktionen (1)

- Transaktionen können formalisiert werden als Folgen von Operationen der folgenden Typen:
  - `read(`x`)`: Eine Kopie von Objekt x aus der DB wird der Transaktion zur Verfügung gestellt.

    Objekte können z.B. Tabellenzeilen oder Plattenblöcke sein.
  - `write(`x`)`: Ersetze die aktuelle Version von Objekt x in der Datenbank durch eine neue Version.

    Natürlich hat `write` einen zweiten Parameter für den neuen Wert.

    Für diese Theorie ist aber nur wichtig, dass x geschrieben wird.
  - `rollback`: Alle Änderungen zurücknehmen.
  - `commit`: Alle Änderungen dauerhaft machen.

Transaktionen (2)

- Jede Transaktion muss mit einem der Kommandos `commit` oder `rollback` enden, und diese Kommandos sind auch nur als letztes Element der Folge erlaubt.

  Für die Definition der (Konflikt-)Serialisierbarkeit könnte man das Kommando `rollback` durch `write(`x`)` für alle in der Transaktion geschriebenen Objekte ersetzen (es setzt die Werte ja auf die alten Werte zurück). Das Kommando `commit` könnte man ganz weglassen (es ändert die Werte der Objekte nicht). Daher ist es auch möglich, als einzige Operationen in Schedules `read(`x`)` und `write(`x`)` zu betrachten. Allerdings werden die Operationen `commit` und `rollback` in der Praxis verwendet, und wären z.B. für einen Transaktionsmanager, der mit Sperren arbeitet, wichtig. In den hier folgenden Definitionen sind sie dagegen nicht wirklich wichtig.

Transaktionen (3)

- Der Transaktions-Manager im DBMS weiß nicht, wie bei `write(x)` der neue Wert von x von der Transaktion berechnet wurde.

  Er kennt den Programmcode nicht, hat keine Formel für diesen Wert.

- Er muss daher annehmen, dass der neue Wert von x möglicherweise von allen Objekten abhängt, die die Transaktion vorher gelesen hat.

Transaktionen (4)

- Dieses formale Modell nimmt an, dass jede Transaktion explizit angibt, welche Objekte (Tupel) sie lesen oder schreiben will.
- Dies ist eine Vereinfachung der Realität: In SQL gibt man eine Bedingung für die zu lesenden oder zu ändernden Tupel an.
- Z.B. kann das Phantom-Problem in diesem Modell gar nicht untersucht werden.

  Natürlich gibt es kompliziertere formale Modelle, die auch Operationen für das Lesen oder Schreiben einer Menge von Objekten haben, die über eine Bedingung spezifiziert wird.

Inhalt

1 Transaktionen

2 Schedules

3 Serialisierbarkeit

Schedules (1)

- Sei eine endliche Menge $\{T_1, \ldots, T_n\}$ gegeben, deren Elemente Transaktionen heißen, und für jedes T_i eine Folge $c_{i,1} \ldots c_{i,m_i}$ von Kommandos.
- Sei weiter $\mathcal{S}$ die Menge der Tripel $s = (T_i, j, c_{i,j})$ mit $1 \leq i \leq n$ und $1 \leq j \leq m_i$ (auszuführende Schritte).

  Menge der Kommandos aller Transaktionen mit Positionsinformation.

- Die Transaktionen definieren eine partielle Ordnung auf $\mathcal{S}$: $s \prec s'$ gdw. s und s' zu der gleichen Transaktion gehören und s vor s' in der Transaktion kommt, d.h. $s = (T_i, j, c_{i,j})$ und $s' = (T_i, k, c_{i,k})$ mit $j < k$.

Schedules (2)

- Ein Schedule (Historie) dieser Transaktionen ist eine lineare Ordnung $<$ auf $\mathcal{S}$, die mit $\prec$ verträglich ist (d.h. wenn $s \prec s'$, dann $s < s'$).
- Ein Schedule definiert also eine Reihenfolge $s_1 \ldots s_l$ von Schritten, die jedes Element von $\mathcal{S}$ genau einmal enthält, und die Ordnung der Schritte innerhalb einer Transaktion berücksichtigt (wenn $s_i \prec s_j$, dann $i < j$).
- Ein Schedule ist also eine Verschachtelung der einzelnen Schritte der Transaktionen.

Schedules (3)

- Beispiel: Angenommen, T_1 und T_2 wollen beide ein Objekt `A` ändern, also haben sie beide die Folge von Operationen: `read(A)`, `write(A)`, `commit`.
- Dann ist ein Schedule (Beispiel für Lost Update):

T_1: `read(A)`,
T_2: `read(A)`,
T_2: `write(A)`,
T_2: `commit`,
T_1: `write(A)`,
T_1: `commit`.

T_1	T_2
`read(A)`	
	`read(A)`
	`write(A)`
	`commit`
`write(A)`	
`commit`	

Schedules (4)

- Serielle Schedules sind Schedules, die jede Transaktion in einem Stück ausführen, d.h. für alle Schritte $s_1 < s_2 < s_3$ gilt: Wenn s_1 und s_3 zur gleichen Transaktion T_i gehören, dann muss s_2 auch zu T_i gehören.

T_1	T_2
`read(A)`	
`write(A)`	
`commit`	
	`read(A)`
	`write(A)`
	`commit`

T_1	T_2
	`read(A)`
	`write(A)`
	`commit`
`read(A)`	
`write(A)`	
`commit`	

Inhalt

1 Transaktionen

2 Schedules

3 Serialisierbarkeit

Serialisierbarkeit

- Informell ist ein Schedule serialisierbar gdw. er äquivalent zu einem seriellen Schedule ist.
- Äquivalent bedeutet:
  - Die Leseoperationen aller Transaktionen liefern die gleichen Werte in beiden Schedules.
  - Am Ende wird der gleiche DB-Zustand erreicht (identische Werte für alle Objekte).
- Das DBMS muss obige Serialisierbarkeit garantieren.
  Da es aber die Berechnung der neuen Werte nicht kennt, kann es die Schedules weiter einschränken.

Konflikt-Serialisierbarkeit (1)

- Man definiert nun eine Konflikt-Relation zwischen Schritten in Transaktionen. Zwei Schritte stehen in Konflikt, wenn die Operationen nicht vertauscht werden können, also ihre Reihenfolge wichtig ist:
  - T_i: `write(`x`)` steht in Konflikt mit T_j: `write(`x`)`,
  - T_i: `write(`x`)` steht in Konflikt mit T_j: `read(`x`)`,
  - T_i: `rollback` steht in Konflikt mit T_j: `read(`x`)` und T_j: `write(`x`)`, wenn `write(`x`)` in T_i enthalten ist.

    D.h. `rollback` schreibt alle Objekte, die in der Transaktion modifiziert wurden: Es muss sie auf auf den alten Wert zurücksetzen.
  - und jeweils auch umgekehrt.

Konflikt-Serialisierbarkeit (2)

- Sei $s_1 \ldots s_k$ ein Schedule. Eine erlaubte elementare Modifikation des Schedules ist es, zwei Schritte zu vertauschen, die direkt aufeinander folgen, also $s_1 \ldots s_i \; s_{i+1} \ldots s_k$ in $s_1 \ldots s_{i+1} \; s_i \ldots s_k$ umzuwandeln, wobei s_i and s_{i+1} nicht in Konflikt stehen.
- Ein Schedule heißt konflikt-serialisierbar gdw. er durch erlaubte elementare Modifikationen in einen seriellen Schedule überführt werden kann.
- Konflikt-Serialisierbarkeit impliziert die Äquivalenz zu einem seriellen Schedule.
- Das Umgekehrte gilt nicht, weil z.B. zwei Schreiboperationen zufällig den gleichen Wert schreiben können.

Konflikt-Serialisierbarkeit (3)

Beispiel/Aufgabe:

- Der folgende Schedule ist konflikt-serialisierbar:

T_1	T_2
	`read(A)`
`read(A)`	
`read(B)`	
	`write(A)`
`write(B)`	
	`commit`
`commit`	

- Überführen Sie diesen Schedule durch erlaubte elementare Modifikationen in einen seriellen Schedule.

Konflikt-Serialisierbarkeit (4)

- Ein einfacher Test für die Konflikt-Serialisierbarkeit eine Schedules ist es, seinen Konflikt-Graphen zu konstruieren (und diesen auf Zyklen zu testen):
  - Die Knoten des Graphen sind die Transaktionen.
  - Es gibt eine Kante von T_i zu T_j ($i \neq j$) gdw. es im Schedule Schritte s von T_i und s' von T_j mit $s < s'$ gibt, so dass s and s' in Konflikt stehen.

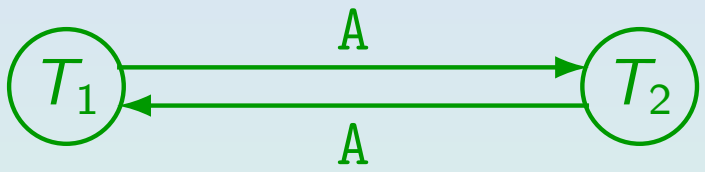
    Man sollte an die Kante die Objekte schreiben, die den Konflikten zugrunde liegen (oder mindestens eines dieser Objekte, um die Kante zu begründen). In Übungsaufgaben und Klausuren ist das oft verlangt, für den eigentlichen Serialisierbarkeitstest ist es nicht wichtig.

Konflikt-Serialisierbarkeit (5)

- Um keinen Konflikt zu übersehen, kann man den Graphen z.B. folgendermaßen aufbauen:
  - Man geht die beteiligten Objekte nacheinander durch, und sucht zuerst die `write`-Anweisungen für das aktuelle Objekt.
  - Für jede `write`-Anweisung erzeugt man Kanten von/zu den anderen Transaktionen, die auf das gleiche Objekt zugreifen.
  - Die Reihenfolge im Schedule bestimmt die Richtung der Kante.

Konflikt-Serialisierbarkeit (6)

- Beispiel: Konfliktgraph für den Schedule auf Folie 11 (mit Lost Update):



- Satz: Ein Schedule ist konflikt-serialisierbar gdw. sein Konfliktgraph keine Zyklen enthält.

  Und topologische Sortierung liefert äquivalente serielle Schedules.

  Beachte: Es ist nicht verlangt, dass das gleiche Objekt an allen Kanten des Zyklus steht (die Kantenbeschriftung ist für diesen Test irrelevant, sie dokumentiert nur den Grund für die Kante). Es ist natürlich auch nicht verlangt, dass der Zyklus alle Knoten des Graphen beinhaltet.

Konflikt-Serialisierbarkeit (7)

Aufgabe:

- Ist dieser Schedule konflikt-serialisierbar?

T_1	T_2	T_3
read(A)		
	read(B)	
	write(B)	
		read(A)
		write(C)
	read(C)	
read(B)		
		write(D)
		commit
commit		
	commit	

Literatur/Quellen

- Kemper/Eickler: Datenbanksysteme (in German), 7th Ed., Oldenbourg, 2009.
  Kapitel 11: Mehrbenutzersynchronisation
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Lewis/Bernstein/Kifer: Databases and Transaction Processing,
  An Application-Oriented Approach. Addison-Wesley, 2002.
- H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil:
  A critique of ANSI SQL isolation levels. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1–10, 1995.