Datenbanksysteme

Wintersemester 2016/17

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Übersicht Recovery: Einfacher Redo-Winners Ansatz

Im Folgenden eine Übersicht zu verschiedenen Ansätzen / Konfigurationen des Crash-Recovery

- Annahme: Volle (physikalische) Before- und After-Images
- Sperren von Seiten
- Kein Rollback
- Wie sieht Redo aus? Einfaches Anwenden der Redo Informationen der Gewinner Transaktionen
- Wieso reicht das aus? Letztes After-Image “gewinnt”.
- Wegen Locking werden Verlierer-TA (ohne Konflikte) rein am Ende vor Crash ausgeführt.
- Undo der Verlierer Transaktionen.
- Keine PageLSN, keine CLRs nötig.

Übersicht Recovery: Nun (Physio)logisches Logging beim Redo-Winners Ansatz

- Es werden nicht komplette physikalische Images (byte[]) gespeichert im Log
- Sondern Beschreibung wie Änderungsoperation gearbeitet hat (A+=10 oder diff)
- Kompakter, aber Redo nicht mehr idempotent!
- Also nicht mehr (wie auf vorheriger Folie) einfaches Anwenden der Gewinner Redos
- Sondern: Wir brauchen nun PageLSNs
- Was ist mit Idempotenz der Undos? PageLSNs funktionieren auch hier (Achtung, Annahme: Seitensperren)

Übersicht Recovery: Nun auch: Rollback beim Redo-Winners Ansatz

- Wo ist das Problem?
- Zurücksetzen von Transaktionen führt dazu, dass nun nicht mehr die Verlierer "am Ende" des Logs (der Ausführung) stehen.
- Lösung: Kompensations-Operationen bei abort.
- Vollständig abgebrochene TA werden zu Gewinnern.
- Was passiert bei Absturz während Rollback?
- Man braucht CLRs um Idempotenz zu gewährleisten.

Übersicht Recovery: Nun: Sperrgranularität: Datensatz

- Redo-Winners (Selektives Redo)? Funktioniert nicht (siehe Erklärung weiter vorne)
- Also: vollständiges Redo (Redo-History): Sowohl Gewinner als auch Verlierer werden nachvollzogen

Wer es noch genauer wissen möchte, inkl. Pseudocode und Korrektheitsbeweise, Details gibt es im Buch von Weikum und Vossen.

Sicherungspunkte (=checkpoints)

Achtung: checkpoint $\neq$ savepoint (Terminologie)

Beobachtung

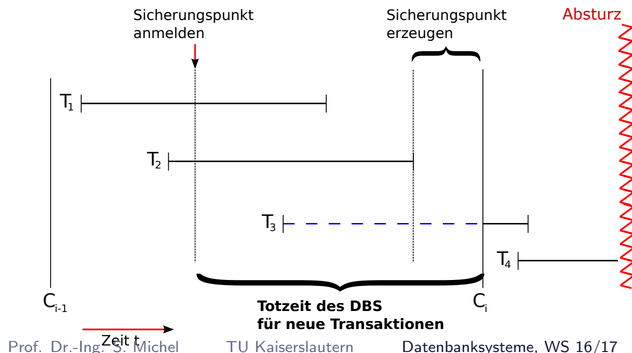
- Mit zunehmender Betriebszeit des Datenbanksystems wird Wiederanlauf immer langwieriger, da die Log-Datei immer umfangreicher wird.

Sicherungspunkte

- Idee: "Markiere" im Log Zeitpunkt, über die man beim Wiederanlauf nicht hinausgehen muss.
- Verschiedene Ansätze.
  - (globale) transaktionskonsistente Sicherungspunkte
  - aktionskonsistente Sicherungspunkte und
  - unscharfe (fuzzy) Sicherungspunkte
- Achtung: "cut-off" Punkt ist nicht unbedingt Zeitpunkt an dem Sicherungspunkt angelegt wird, kann auch älter sein; kleinste noch benötigte LSN wird angegeben.

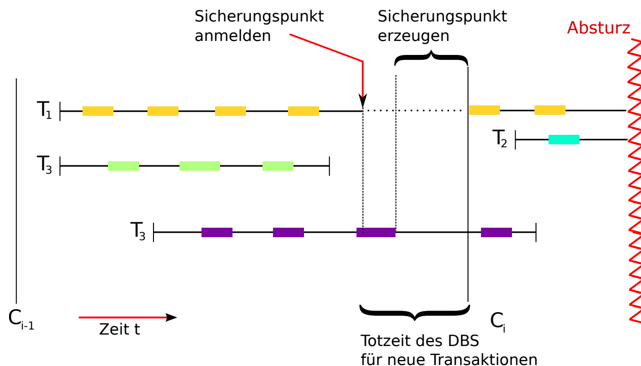
Transaktionskonsistente Sicherungspunkte (TCC)

- **Transaction-Consistent Checkpoint (TCC)**
- Neu ankommende TA (hier T_3) müssen warten. Laufende TA (hier T_1 und T_2) werden zu Ende ausgeführt.
- Dann schreiben aller modifizierten Seiten auf Hintergrundspeicher.
- Redo und Undo höchstens für TA nach Sicherungspunkt nötig.
- Führt i.a. zu einer sehr langen Totzeit des Systems für den Änderungsbetrieb.



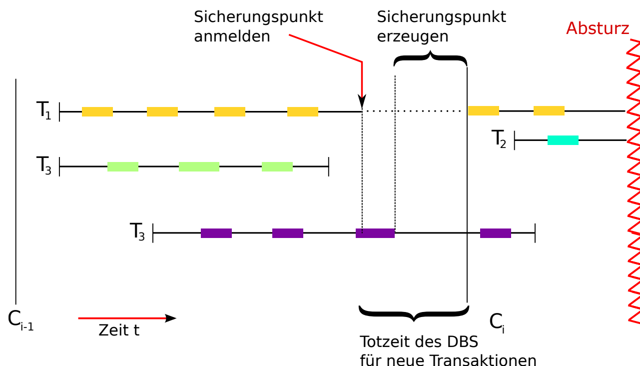
Aktionskonsistente Sicherungspunkte (ACC)

- **Action-Consistent Checkpoints (ACC)**
- Transaktionsausführung relativ zu einem aktionskonsistenten Sicherungspunkt und einem Systemabsturz.
- Aktive Änderungsoperationen (hier, bei T_4) werden noch ausgeführt und ausgeschrieben.
- Neue Änderungsoperationen müssen warten (hier, bei T_1)



Aktionskonsistente Sicherungspunkte (2)

- Man braucht keine Redo-Informationen, die älter sind als der Zeitpunkt des Schreibens auf Hintergrundspeicher
- Aber man braucht u. U. Undo-Informationen
- Also: Kleinste LSN aller zum Zeitpunkt noch aktiven LSN speichern (=MinLSN) + Liste aller noch aktiven TA



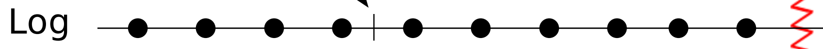
Unscharfe (fuzzy) Sicherungspunkte

- Idee: modifizierte Seiten werden **nicht** ausgeschrieben
- Sondern nur deren Kennung (PageID)
- Und die älteste LSN, die diese Seite hat dreckig werden lassen werden notiert (in Log-Datei).

Terminologie:

- **MinDirtyPageLSN:**
  - Die kleinste LSN, deren Änderungen noch nicht ausgeschrieben wurde
  - D.h. die älteste Änderungsoperation, die eine Seite geändert hat (hat "dreckig" werden lassen).
  - Die kleinste all dieser LSNs wird MinDirtyPageLSN genannt.
  - D.h. bis dahin muss Redo Phase laufen.
- **MinLSN:**
  - Die kleinste LSN der zum Sicherungszeitpunkt aktiven TA
  - Erste/älteste Änderungsoperation aller Loser-TA
  - Die älteste Änderung, die rückgängig gemacht werden muss (im Undo)

Zusammenfassung der drei Arten von Sicherungspunkten



(1) transaktionskonsistent

Analyse

Redo

Undo

(2) aktionskonsistent

Analyse

Redo

MinLSN

Undo

(3) unscharf (fuzzy)

Analyse

MinDirtyPageLSN

Redo

MinLSN

Undo

Anmerkung zur Analyse-Phase

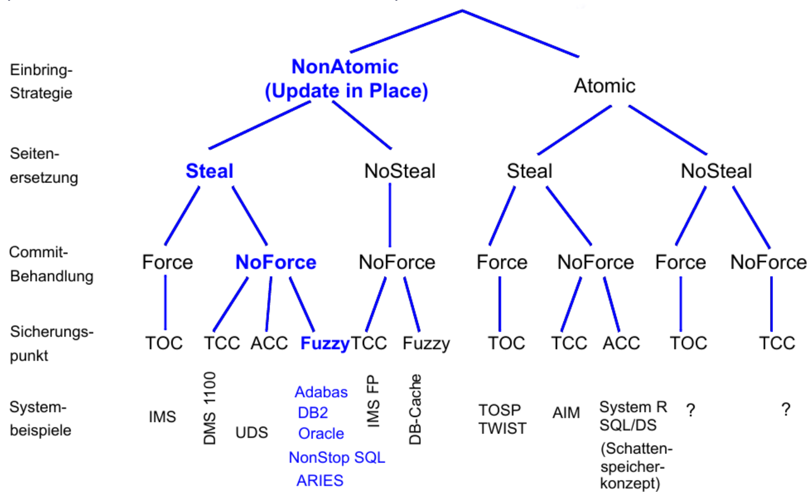
- Analyse-Phase ist mehr oder weniger optional bei dem vollständigen Redo (Redo-History)
- Für Optimierungen der Redo-Phase ist die Analyse-Phase allerdings notwendig (z.B. Prefetching von bekannten DirtyPages zum Redo-Zeitpunkt)
- Ob nun 2 oder 3 Phasen günstiger sind hängt von einigen Dingen ab, z.B. Häufigkeit der Checkpoints.

Weitere Anmerkungen

- Relativ gesehen gibt es sehr wenige Rollbacks und TAs, die zurückgenommen (Undo) werden müssen.
- Anzahl nebenläufiger TA ist obere Schranke für Verlierer-Transaktionen.

Klassifikation der DB Recovery-Verfahren

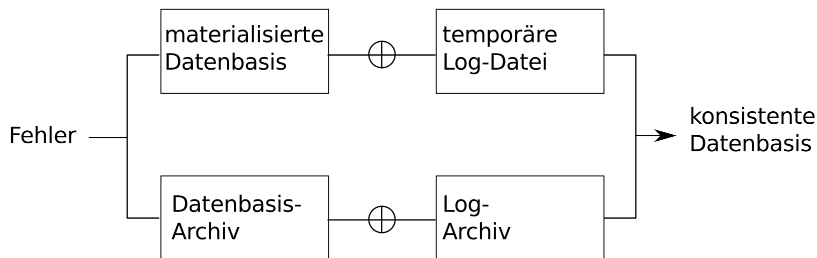
(nur zum Teil hier angesprochen)



In blau hervorgehoben: Hauptsächlich betrachtete Konfiguration.

Recovery

Recovery nach einem Verlust der materialisierten Datenbasis.



Dies kann auch auf einzelne Seiten angewandt werden z.B. bei einzelnen fehlerhaften Seiten (Media Recovery)

Zusammenfassung Recovery

- Motivation: ACID (Speziell Atomicity und Durability)
- Aber auch: High Availability (durch kleine MTTR), also Effizienz
- Es gibt verschiedene Arten von Recovery (je nach Fehler)
- Haben (hauptsächlich) über Crash-Recovery gesprochen
- Essenz: Speichern von Log-Informationen
- WAL-Prinzip und Commit-Regel
- Redo von Gewinner-TA, Undo von Verlierer-TA
- Idempotenz des Wiederanlaufs
- Sicherungspunkte zum schnelleren Wiederanlauf (nicht das ganze Log betrachten müssen)
- Kurz: Zurücksetzen von Transaktionen

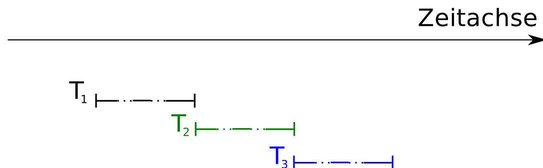
Konzept der Mehrbenutzersynchronisation

Wiederholung: Mehrbenutzersynchronisation

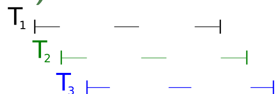
Das "**I**" in AC**I**D.

Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einzelbetrieb



(b) im (verzahnten) Mehrbenutzerbetrieb



Ziel: Semantik von seriell ausgeführten Transaktionen zusammen mit Performance von verzahnter Ausführung.

Mehrbenutzersynchronisation – Aspekte

Konzept der Serialisierbarkeit

- Final-State-Serialisierbarkeit (FSR)
- View-Serialisierbarkeit (VSR)
- Konflikt-Serialisierbarkeit (CSR)

Synchronisation

- Basierend auf Sperren
- Basierend auf Zeitstempeln
- Behandlung von Deadlocks

Wiederholung: Das lost-update Problem

t_1	Time	t_2
	/* $x = 100$ */	
$r(x)$	1	
	2	$r(x)$
/*update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$*/	

Wiederholung: Das lost-update Problem / Notation

- Die Essenz dieses Problems kann durch folgende Sequenz von Lese- und Schreiboperationen ausgedrückt werden:

$$r_1(x)r_2(x)w_1(x)w_2(x)$$

Wiederholung: Das inconsistent-read Problem

Beispiel aus z.B. Anwendung in Bank. Aktueller Stand $x = y = 50$, also $x + y = 100$. Transaktion t_1 berechnet die Summe von x und y, während t_2 einen Wert von 10 von x nach y transferiert.

t_1	Time	t_2
	1	$r(x)$
	2	/* $x := x - 10$ */
	3	$w(x)$
/* $sum := 0$ */	4	
$r(x)$	5	
$r(y)$	6	
/* $sum := sum + x$ */	7	
/* $sum := sum + y$ */	8	
	9	$r(y)$
	10	/* $y := y + 10$ */
	11	$w(y)$

Wiederholung: Das inconsistent-read Problem (2)

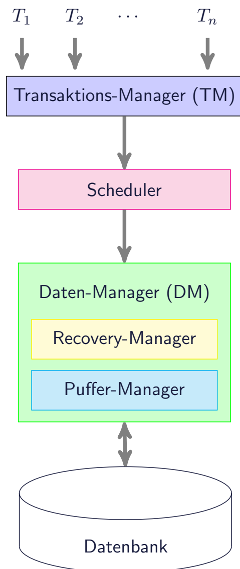
- Offensichtlich ist auch hier wieder das Problem, dass Lese- und Schreiboperationen der einzelnen Transaktionen gemischt ablaufen

$$r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)$$

Wiederholung: Das dirty-read Problem

t_1	Time	t_2
$r(x)$	1	
/* $x := x + 100$ */	2	
$w(x)$	3	
	4	$r(x)$
	5	/* $x := x - 100$ */
failure & rollback	6	
	7	$w(x)$

Der Datenbank-Scheduler



- TM: Informationen über TA, welche Schritte als nächstes ausgeführt werden können.
- Scheduler: Bekommt Schedules vom TM und muss diese in einen serialisierbaren Schedule umwandeln.
- Welcher verarbeitet wird von Daten-Manager (DM)

Aktionen des Schedulers

Scheduler empfängt Schritte der Form r, w, a und c als Eingabe. Diese müssen in einen serialisierbaren Schedule überführt werden. Dazu kann Scheduler folgende Aktionen durchführen:

1. **Output:** Schritt (r, w, a oder c) wird an Output-Schedule (am Anfang leer) angehängt.
2. **Reject:** Schritt wird nicht ausgegeben (weil z.B. dies die Serialisierbarkeit des Outputs zerstören würde). Also muss die dazugehörige TA abgebrochen werden.
3. **Block:** Schritt wird nicht ausgegeben und auch nicht rejected, sondern als "momentan nicht ausführbar" angesehen und deshalb auf einen späteren Zeitpunkt verschoben.

Es gibt optimistische und pessimistische Scheduler. Sperrbasierte Scheduler (z.B. nach 2PL) gehören zur Klasse der pessimistischen Scheduler.

Ziele

- Formale Betrachtung von verschiedenen Klassen von Schedules
- Welche Probleme mit welcher Klasse behoben werden und welche nicht

Wir betrachten erstmal nicht:

- Wie ein Protokoll aussieht, das z.B. durch Sperren nur Schedules einer bestimmten Klasse erzeugt.
- Dazu später mehr.

Transaktion

- Eine Transaktion t ist eine Partialordnung von Schritten der Form

$$p_i \in \{r(x), w(x)\}$$

  mit $x \in D$ ein Datenobjekt.
- Lese- und Schreiboperationen sowie mehrfache Schreiboperationen auf demselben Datenobjekt sind geordnet.
- Eine vollständige TA hat als letzte Operation entweder einen Abbruch a oder ein Commit c

$$t = p_1 \dots p_n a$$

  oder

$$t = p_1 \dots p_n c$$

- Man kann für Partialordnungen vollständige Ordnungen angeben, in dem man nicht geordnete Elements einfach in eine (beliebige) Reihenfolge bringt.
- Achtung: in der Schreibweise $t = p_1\ p_2\ p_3\ \dots$ ist die Ordnung aller Operationen gegeben (von links nach rechts).

Historien und Schedules

- Es sei $T = \{t_1, \dots, t_n\}$ eine Menge von Transaktionen, wobei jedes $t_i \in T$ die Form $t_i = \{op_i, <_i\}$ besitzt, op_i die Menge der Operationen von t_i und $<_i$ ihre Ordnung $(1 \leq i \leq n)$ bezeichnen.
- Eine **Historie** für T ist ein Paar $s = (op(s), <_s)$, so dass:
  (a) $op(s) \subseteq \bigcup_{i=1}^{n} op_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$ und $\bigcup_{i=1}^{n} op_i \subseteq op(s)$
  (b) $(\forall i, 1 \leq i \leq n) c_i \in op(s) \Leftrightarrow a_i \notin op(s)$
  (c) $\bigcup_{i=1}^{n} <_i \subseteq <_s$
  (d) $(\forall i, 1 \leq i \leq n)(\forall p \in op_i)\ p <_s a_i$ oder $p <_s c_i$
  (e) Jedes Paar von Operatoren $p,q \in op(s)$ von verschiedenen Transaktionen, die auf dasselbe Datenobjekt zugreifen und von denen wenigstens eine davon eine Schreiboperation ist, sind so geordnet, dass $p <_s q$ oder $q <_s p$ gilt.
- Ein **Schedule** ist ein Präfix einer Historie

Historien und Schedules (2)

Erläuterungen zu den zuvor genannten Punkten (a) bis (e):

- (a) Historie enthält alle Operationen aller Transaktionen
- (b) Historie benötigt eine Terminierungsoperation für jede TA
- (c) Historie bewahrt alle Ordnungen innerhalb der TA
- (d) Terminierungsoperation ist letzte Operation in jeder TA
- (e) Konfliktoperationen sind geordnet.

Historien und Schedules (3)

Bemerkung

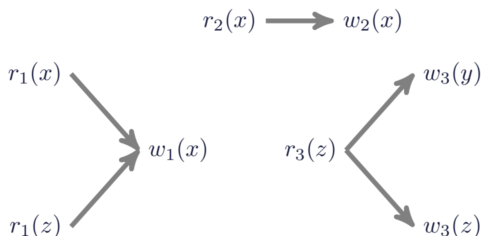
- Wegen (a) und (b) wird eine Historie auch als vollständiger Schedule bezeichnet
- Ein Präfix einer Historie kann die Historie selbst sein
- Historien lassen sich als Spezialfälle von Schedules betrachten; es genügt deshalb meist, einen gegebenen Schedule zu betrachten

Definition: Serielle Historie

- Eine Historie s ist **seriell**, wenn für jeweils zwei TA t_i und t_j $(i \neq j)$ alle Operationen von t_i vor allen Operationen von t_j in s auftreten oder umgekehrt.

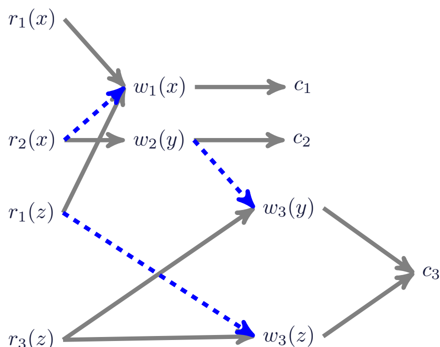
Beispiel

- Wir sehen hier drei Beispiel-Transaktionen, t_1, t_2 und t_3.
- Hier, jeweils, dargestellt als ein gerichteter azyklischer Graph, dessen Kanten die (partielle) Ordnung der Schritte beschreibt.
- Partiell, weil z.B. in Transaktion t_1 nicht festgelegt ist ob $r_1(x)$ vor oder nach $r_1(z)$ ausgeführt werden soll.
- Aber beide Operationen müssen vor $w_1(x)$ ausgeführt werden!



Beispiel (Fortsetzung)

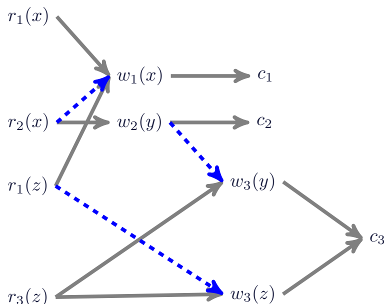
- Nehmen wir noch an, dass diese drei TA auch committen
- Folgende partielle Ordnung ist möglich:



- Die normalen Kanten stammen aus den ursprünglichen TA
- Die blauen gestrichelten Kanten mussten aufgrund von Regel (e) eingefügt werden, da Konfliktoperationen geordnet sein müssen!

Beispiel (Fortsetzung)

Schauen wir uns nochmal die partielle Ordnung von zuvor an:



- Eine mögliche **totale Ordnung** darauf ist gegeben durch (topologisches Sortieren):

$$r_1(x)\ r_2(x)\ r_1(z)\ w_1(x)\ w_2(y)\ r_3(z)\ w_3(y)\ c_1\ c_2\ w_3(z)\ c_3$$

Historien und Schedules (4)

Definition: TA-Mengen eines Schedules

- Alle TA in s:

$$trans(s) := \{t_i | s \text{ enthält Schritte von } t_i\}$$

- Alle TA die in s committen:

$$commit(s) := \{t_i \in trans(s) | c_i \in s\}$$

- Alle TA die in s aborten:

$$abort(s) := \{t_i \in trans(s) | a_i \in s\}$$

- Alle aktiven TA in s:

$$active(s) := trans(s) - (commit(s) \cup abort(s))$$

Historien und Schedules (5)

Für jede Historie s gilt

- $trans(s) = commit(s) \cup abort(s)$
- $active(s) = \emptyset$

Beispiel

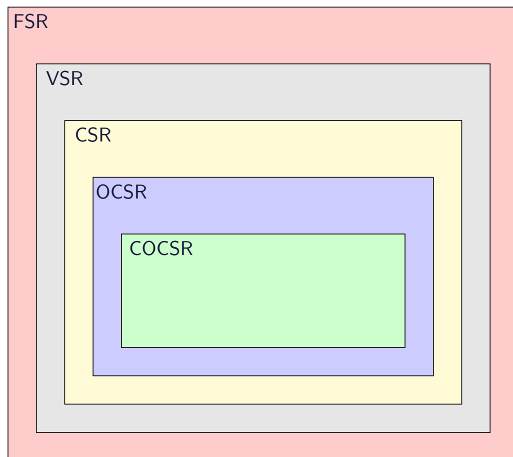
- $s_1 =$ $r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ c_2\ a_3$
    - $trans(s_1) = \{t_1, t_2, t_3\}$
    - $commit(s_1) = \{t_1, t_2\}$
    - $abort(s_1) = \{t_3\}$
    - $active(s_1) = \emptyset$
- $s_2 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1$
    - $trans(s_1) = \{t_1, t_2, t_3\}$
    - $commit(s_1) = \{t_1\}$
    - $abort(s_1) = \emptyset$
    - $active(s_1) = \{t_2, t_3\}$

Serialisierbarkeitsklassen

Ziel

- formale Betrachtung des Serialisierbarkeitsbegriffs

Klassen



Historien und Schedules - Monotonie

Definition: Monotone Klassen von Historien

Eine Klasse E von Historien heißt monoton, wenn Folgendes gilt:

- Wenn s in E ist, dann ist die Projektion s' von s auf T $s' = \Pi_T(s)$ mit $op(s') = op(s) - \cup_{t \notin T} op(t)$, in E für jedes $T \subseteq trans(s)$
- Mit anderen Worten: E ist unter beliebigen Projektionen abgeschlossen

Monotonie

- Monotonie einer Historienklasse E ist eine wünschenswerte Eigenschaft, da sie E unter beliebigen Projektionen bewahrt!

Serialisierbarkeitsklassen

Akzeptable Klasse von Schedules ...

- muss mindestens **Lost Update** und **Inconsistent Read** ausschließen
- muss Zugehörigkeit eines Schedules effizient entscheiden können
- muss bei Annahme von Fehlern (Aborts) Abhängigkeit von nicht freigegebenen Änderungen (**Dirty Read**) vermeiden:

$$r_1(x)\, w_1(x)\, r_2(x)\, a_1\, w_2(x)\, c_2$$

Klasse FSR (Final-State-Serialisierbarkeit)

Definition: Final-State-Serialisierbarkeit

- Eine Historie s ist final-state-serialisierbar, wenn eine serielle Historie s' existiert, so dass $s \approx_f s'$.

  FSR bezeichnet die Klasse aller final-state serialisierbaren Historien.

Final-State-Serialisierbarkeit

- Final-State-Äquivalenz: $s \approx_f s'$, wenn sie ausgehend vom selben Ausgangszustand denselben Endzustand der DB erzeugen.
- Konsistenter DB-Zustand wird nur am Ende der Historie gewährleistet. FSR macht deshalb nur Sinn für Historien (vollständige Schedules).

Hinweis: $\approx_f$ und später auch $\approx_v$ sind hier nicht formal definiert, da recht komplex; siehe Buch von Weikum und Vossen, Kapitel 3.

Klasse FSR - Beispiel

Ist Historie s_{FSR}, die einen Zyklus enthält, final-state-serialisierbar?

$$s_{FSR} = w_1(x)\ \ r_2(x)\ \ w_2(y)\ \ c_2\ \ r_1(y)\ \ w_1(y)\ \ c_1\ \ w_3(x)\ \ w_3(y)\ \ c_3$$

- Ja, denn es gilt $s_{FSR} \approx_f s'$, mit $s' = t_1 t_2 t_3$
- Wieso?

Annahmen für Beispiele mit konkreten Werten

- Initiale DB $= \{x = 0, y = 0\}$
- r liest aktuellen Wert a
- r vor w: w schreibt $a + 1$
- blindes schreiben: w schreibt irgendeinen Wert

Beispiel von zuvor mit konkreten Werten:

$$s_{FSR} = \quad w_1(x=5)\ r_2(x=5)\ w_2(y=7)\ c_2\ r_1(y=7)\ w_1(y=8)\ c_1\ w_3(x=1)\ w_3(y=1)\ c_3$$

$$s' = \quad w_1(x=5)\ r_1(y=0)\ w_1(y=1)\ c_1\ r_2(x=5)\ w_2(y=7)\ c_2\ w_3(x=1)\ w_3(y=1)\ c_3$$

Also: $s_{FSR} \approx_f s' = t_1\ t_2\ t_3$

Klasse FSR - Plausibilitätstest

Plausibilitätstest: FSR ist nicht ausreichend!

Lost Update

$$L = r_1(x)\ \ r_2(x)\ \ w_1(x)\ \ w_2(x)\ \ c_1\ \ c_2$$

- Mit konkreten Beispielwerten:

- In dem Schedule oben haben wir:

$$L = r_1(x=0)\ \ r_2(x=0)\ \ w_1(x=1)\ \ w_2(x=1)\ \ c_1\ \ c_2$$

- Für die seriellen Schedules haben wir aber:

$$t_1 t_2 \equiv r_1(x=0)\ \ w_1(x=1)\ \ c_1\ \ r_2(x=1)\ \ w_2(x=2)\ \ c_2$$

$$t_2 t_1 \equiv r_2(x=0)\ \ w_2(x=1)\ \ c_2\ \ r_1(x=1)\ \ w_1(x=2)\ \ c_1$$

- Also $L \notin$ FSR, da $t_1 t_2$ oder $t_2 t_1$ andere Endzustände erzeugen würden.
- OK! Aber was ist mit Inconsistent Read?

Klasse FSR - Plausibilitätstest (2)

Inconsistent Read

- $I = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$
- Mit konkreten Beispielwerten:

- Für Schedule I haben wir:

$$I = r_2(x=0)\ w_2(x=1)\ r_1(x=1)\ r_1(y=0)\ r_2(y=0)\ w_2(y=1)\ c_1\ c_2$$

- Für die seriellen Schedules haben wir:

$$t_2 t_1 \equiv r_2(x=0)\ w_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2\ r_1(x=1)\ r_1(y=1)\ c_1$$

$$t_1 t_2 \equiv r_1(x=0)\ r_1(y=0)\ c_1\ r_2(x=0)\ w_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2$$

- Also $I \in$ FSR , da $t_1 t_2$ oder $t_2 t_1$ denselben Endzustand erzeugen, obwohl t_1 inkonsistente Werte liest. **FSR verhindert also nicht inkonsistentes Lesen.**

Klasse FSR - Abschließende Bermerkungen

- Wie wir gerade gesehen haben genügt die Klasse FSR nicht unseren Anforderungen bzw. der Vermeidung von Inconsistend Read.
- Bzgl. Entscheidbarkeit: Für zwei gegebene Schedules s und s' kann in polynomialer Zeit (in der Länge der beiden Schedules) entschieden werden ob $s \approx_f s'$ gilt.
- Aber es gibt für Schedule s mit n Transaktionen $n!$ verschiedene serielle Schedules. Testen ist nicht einfach (Literatur!).
- Ist aber auch nicht so wichtig, da ja auch bereits die Anforderungen oben nicht erfüllt worden sind und FSR in der Praxis daher nicht relevant.

Klasse VSR (View-Serialisierbarkeit)

Definition: View-Serialisierbarkeit

- Ein Schedule s ist view-serialisierbar, wenn ein serieller Schedule s' existiert, so dass $s \approx_v s'$. VSR bezeichnet die Klasse aller view-serialisierbaren Historien.

View-Serialisierbarkeit

- s erfüllt VSR, wenn eine view-äquivalente serielle Historie erzeugt werden kann und
- die gesamte Historie einen konsistenten DB-Zustand hinterlässt
- View-äquivalent bedeutet, dass alle Leseoperationen Werte liefern wie in einem seriellen Schedule.

- Verhindert inkonsistentes Lesen; da gewährleistet wird, dass die Sicht (View) jeder TA konsistent ist.

Klasse VSR - Beispiel

Ist Historie s_{VSR}, die einen Zyklus enthält, view serialisierbar?

$$s_{VSR} = r_1(x)w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c1\ w_3(x)\ w_3(y)\ c_3$$

Beispiel mit konkreten Werten:

- Annahme wie bisher: Initiale DB $= \{x = 0, y = 0\}$ usw.

  $$s_{VSR} = r_1(x=0)w_1(x=1)\ w_2(x=5)\ w_2(y=7)\ c_2$$
  $$w_1(y=3)\ c1\ w_3(x=1)\ w_3(y=1)\ c_3$$

  $$s' = r_1(x=0)\ w_1(x=1)\ w_1(y=3)\ c_1\ w_2(x=5)\ w_2(y=7)\ c_2$$
  $$w_3(x=1)\ w_3(y=1)c_3$$

- Also $s_{SVR} \approx_v s' = t_1 t_2 t_3$

Klasse VSR - Plausibilitätstest

Plausibilitätstest: Ist VSR ausreichend?

Lost Update

$$L = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$$

Mit konkreten Beispielwerten:

$$L = r_1(x=0)\ r_2(x=0)\ w_1(x=1)\ w_2(x=1)\ c_1\ c_2$$

- $L \notin$ VSR, da keine view-äquivalente serielle Historie erzeugt werden kann

$$t_1t_2 \equiv r_1(x=0)\ w_1(x=1)\ c_1\ r_2(x=1)\ w_2(x=2)\ c_2$$
$$t_2t_1 \equiv r_2(x=0)\ w_2(x=1)\ c_2\ r_1(x=1)\ w_1(x=2)\ c_1$$

Klasse VSR - Plausibilitätstest (2)

Plausibilitätstest: Ist VSR ausreichend?

Inconsistent Read

$$I = r_2(x)\ \ w_2(x)\ \ r_1(x)\ \ r_1(y)\ \ r_2(y)\ \ w_2(y)\ \ c_1\ \ c_2$$

Mit konkreten Beispielwerten:

$$I = r_2(x=0)\ \ w_2(x=1)\ \ r_1(x=1)\ \ r_1(y=0)$$
$$r_2(y=0)\ \ w_2(y=1)\ \ c_1\ \ c_2$$

- $I \notin$ VSR , da **keine** view-äquivalente serielle Historie erzeugt werden kann

$$t_2 t_1 \equiv r_2(x=0)\ w_2(x=1)\ r_2(y=0)\ w_2(y=1)\ c_2$$
$$r_1(x=1)\ r_1(y=1)\ c_1$$
$$t_1 t_2 \equiv r_1(x=0)\ r_1(y=0)\ c_1 r_2(x=0)\ w_2(x=1)$$
$$r_2(y=0)\ w_2(y=1)\ c_2$$

Klasse VSR - Eignung

Eignung bzw. Konsistenz

- VSR erfüllt Anforderungen für das Lost-Update-Problem und das Inconsistent-Read-Problem.

Aber:

Theorem zu Komplexität des Entscheidungsproblem

- Das Entscheidungsproblem, ob für einen gegebenen Schedule $s \in$ VSR gilt, ist NP-vollständig.