



Inhalt

1. Anomalien im Mehrbenutzerbetrieb
2. Serialisierbarkeit von Transaktionen
3. Sperrverfahren (Locking)
4. Behandlung von Verklemmungen
5. Synchronisation ohne Sperren



Synchronisation ohne Sperren

- **Synchronisation mit Sperren**
 - *pessimistische* Annahme: Konflikte sind möglich und treten (oft) auf
 - Vorgehen: *Verhinderung* von Konflikten
 - Methode: *Blockierung* von Transaktionen
 - reale Gefahr von Verklemmungen
 - Sperrenverwaltung ist sehr aufwändig
 - mögliche Leistungseinbußen durch lange Wartezeiten
- **Nicht-sperrende Synchronisation**
 - *optimistische* Annahme: Konflikte sind seltene Ereignisse
 - Vorgehen: *Auflösung* von Konflikten
 - Methode: *Rücksetzen* von Transaktionen
 - keine Verklemmungen
 - aufwändige Konfliktprävention wird eingespart
 - mögliche Leistungseinbußen durch häufige Rücksetzungen



Zeitstempel statt Sperren (1)

- Motivation:
 - Jede TA bekommt zu BOT einen Zeitstempel
 - Hierdurch Definition des äqv. seriellen Schedule (äsS)
 - Bei jedem Zugriff: Test, ob Verletzung des äsS
 - Keine Sperren, sondern über Zeitstempel auf Objekten
- Beispiel: äsS: älteste T_1 T_2 T_3 T_4 T_5 jüngste TA
- Welche Zugriffe müssen verhindert werden?
 - Bei Lesezugriff von T_3 auf ein Objekt x :
 - Lesezugriff ist verboten, wenn *vorher* T_4/T_5 x geschrieben hat
 - *nachher* dürfen T_1 und T_2 das Objekt x nicht mehr schreiben
 - Bei Schreibzugriff durch T_3 auf ein Objekt x :
 - T_4/T_5 dürfen x nicht *vorher* gelesen oder geschrieben haben
 - T_1/T_2 dürfen x nicht *nachher* das Objekt lesen oder schreiben



Zeitstempel statt Sperren (2)

- Nicht nur Transaktionen, sondern auch Objekte O tragen Zeitstempel:
 - $readTS(O)$: Zeitstempel der jüngsten TA, die das Objekt O gelesen hat.
 - $writeTS(O)$: Zeitstempel der jüngsten TA, die das Objekt O geschrieben hat.
- Prüfungen beim **Lesezugriff** von T_i auf ein Objekt O :
 - Falls $TS(T_i) < writeTS(O)$:
 T_i ist älter als die TA, die O geschrieben hat $\rightarrow T_i$ zurücksetzen
 - Falls $TS(T_i) \geq writeTS(O)$:
 T_i ist jünger als die TA, die O geschrieben hat $\rightarrow T_i$ darf O lesen, Lesemarke wird aktualisiert: $readTS(O) = \max(TS(T_i), readTS(O))$



Zeitstempel statt Sperren (3)

- Prüfungen beim **Schreibzugriff** von T_i auf ein Objekt O :

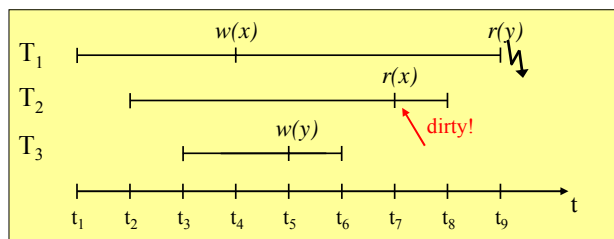
- Falls $TS(T_i) < readTS(O)$:
 T_i ist älter als eine TA, die O gelesen hat $\Rightarrow T_i$ zurücksetzen
- Falls $TS(T_i) < writeTS(O)$:
 T_i ist älter als die TA, die O geschrieben hat $\Rightarrow T_i$ zurücksetzen
- Sonst:
 T_i darf O schreiben,
Schreibmarke wird aktualisiert: $writeTS(O) = TS(T_i)$

- Keine Garantie von kaskadenfrei rücksetzbaren Schedules



Zeitstempel statt Sperren (4)

- Beispiel



Seien $writeTS(x)$, $writeTS(y)$, $readTS(x)$ und $readTS(y)$ kleiner als t_1

- $t1$: $TS(T_1) = t_1$
- $t2$: $TS(T_2) = t_2$
- $t3$: $TS(T_3) = t_3$
- $t4$: $write(x)$ in T_1 : Da $TS(T_1) > readTS(x)$ darf T_1 auf x schreiben, dann: $writeTS(x) := t_4$
- $t5$: $write(y)$ in T_3 : Da $TS(T_3) > readTS(y)$ darf T_3 auf y schreiben, dann: $writeTS(y) := t_5$
- $t6$: keine Prüfung bei **COMMIT** von T_3
- $t7$: $read(x)$ in T_2 : Da $TS(T_2) \geq writeTS(x)$ darf T_2 auf x lesen, dann: $readTS(x) := t_7$
- $t8$: keine Prüfung bei **COMMIT** von T_2 (eigenes Problem mit dirty read siehe unten)
- $t9$: $read(y)$ in T_1 : Da $TS(T_1) < writeTS(y)$ wird T_1 zurückgesetzt



Zeitstempel statt Sperren (5)

- **Problem mit Dirty Read**

- im Beispiel: T_2 liest x , obwohl T_1 noch kein *COMMIT* hatte
- geänderte, aber noch nicht festgeschriebene Daten müssen noch gegen Lesen bzw. Überschreiben gesichert werden (z.B. durch *dirty*-Bit) → damit aber wieder Deadlocks möglich

- **Auswirkungen**

- Methode garantiert Serialisierbarkeit bis auf Dirty Read
- es treten keine Deadlocks auf (möglicherweise jedoch durch *dirty*-Bit)
- äquivalente serielle Reihenfolge entspricht den Zeitstempeln der TAs



Zeitstempel statt Sperren (6)

- **Nachteil für lange TAs**

- Rücksetzgefahr steigt mit Dauer der TA
- Verhungern von TAs durch wiederholtes Zurücksetzen wird nicht verhindert

- **Bewertung**

- Verwaltung der Objektmarken ist sehr aufwändig und nicht feingranularer als auf Seitenebene praktikabel
- Zeitstempel müssen für jedes Objekt verwaltet werden, während Sperren nur bei Zugriff auf Objekte angelegt werden

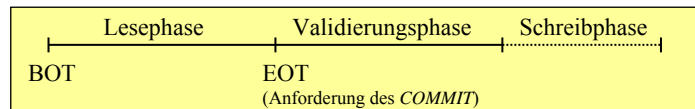


Optimistische Synchronisation (1)

- **Konzept**

- Keine Konfliktprävention
- Konflikte werden erst bei *COMMIT* festgestellt
- Im Konfliktfall werden Transaktionen zurückgesetzt
- nahezu beliebige Parallelität, da TAs nicht blockiert werden

- **Drei Phasen einer TA**



- *Lesephase*:
eigentliche TA-Verarbeitung, Änderungen nur im lokalen TA-Puffer
- *Validierungsphase*:
Prüfung, ob die abzuschließende TA mit nebenläufigen TAs in Konflikt geraten ist; im Konfliktfall wird die TA zurückgesetzt
- *Schreibphase*:
nach erfolgreicher Validierung werden die Änderungen dauerhaft gespeichert



Optimistische Synchronisation (2)

- **Validierungstechniken**

- Für jede Transaktion T_i werden zwei Mengen geführt:
 - $RS(T_i)$: die von T_i gelesenen Objekte (*Read Set*)
 - $WS(T_i)$: die von T_i geschriebenen Objekte (*Write Set*)
- Konflikterkennung
 - Konflikt zwischen T_i und T_j liegt vor, wenn $WS(T_i) \cap RS(T_j) \neq \emptyset$
 - Annahme: $WS(T_i) \subseteq RS(T_i)$, d.h. jedes Objekt wird vor dem Schreiben gelesen (neue Objekte müssen nicht in die Lesemenge eingetragen werden)
- Zwei Validierungsstrategien
 - *Backward-Oriented Optimistic Concurrency Control (BOCC)*:
Validierung nur gegenüber bereits beendeten TAs
 - *Forward-Oriented Optimistic Concurrency Control (FOCC)*:
Validierung nur gegenüber noch laufenden TAs

- **Bemerkungen**

- Serialisierungsreihenfolge ist durch Validierungsreihenfolge gegeben
- Validierung und Schreiben muss sequenziell durchgeführt werden



BOCC (1)

• Validierung von T_i

- “Wurde eines der während der Lesephase von T_i gelesenen Objekte von einer anderen (bereits beendeten) Transaktion T_j geändert?”
- D.h. Read-Set $RS(T_i)$ wird mit allen Write-Sets $WS(T_j)$ von Transaktionen T_j verglichen, die während der Lesephase von T_i validiert haben

• Algorithmus

```

VALID := true;
for (alle während Ausführung von  $T_i$  beendeten  $T_j$ ) do
  if  $RS(T_i) \cap WS(T_j) \neq \emptyset$  then VALID := false;
end;
if VALID then Schreibphase( $T_i$ ); Commit ( $T_i$ );
else Rollback( $T_i$ ); // Nothing to do

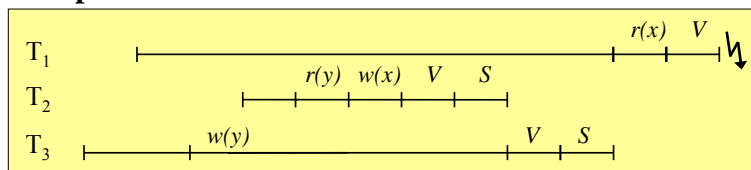
```



BOCC (2)

• Beispiel

V: Validierung
S: Schreibphase



• Ablauf

- T_2 wird erfolgreich validiert, da es noch keine validierten TAs gibt
 - T_3 wird erfolgreich validiert, da für $y \in WS(T_3) \subseteq RS(T_3)$ gilt: $y \notin WS(T_2)$
 - T_1 steht wegen $x \in WS(T_2)$ in Konflikt mit T_2 und wird abgebrochen
- Zurücksetzen war unnötig, da T_1 bereits die aktuelle Version von x gelesen hat.



BOCC (3)

- **Abhilfe:** BOCC+
 - Objekte bekommen Änderungszähler oder Versionsnummern
 - TAs werden nur zurückgesetzt, wenn sie tatsächlich veraltete Daten gelesen haben
- **Nachteile für lange TAs**
 - Verhungern von Transaktionen wird nicht verhindert
 - Anzahl der zu vergleichenden Write-Sets steigt mit TA-Dauer
 - TAs mit großen Read-Sets können in viele Konflikte geraten
 - spätes Zurücksetzen erst bei der Validierung verursacht hohen Arbeitsverlust



FOCC (1)

- **Validierung von T_i**
 - “Wurde eines der von T_i geänderten Objekte von einer anderen (noch laufenden) Transaktion T_j gelesen?”
 - D.h. Write-Set $WS(T_i)$ wird mit allen Read-Sets $RS(T_j)$ von Transaktionen T_j verglichen, die sich gerade in der Lesephase befinden
- **Algorithmus**

```
VALID := true;  
for (alle laufenden  $T_j$ ) do  
  if  $WS(T_i) \cap RS(T_j) \neq \emptyset$  then VALID := false;  
end;  
if VALID then Schreiphase( $T_i$ ) ; commit ( $T_i$ );  
else löse Konflikt auf;
```



FOCC (2)

• Bewertung

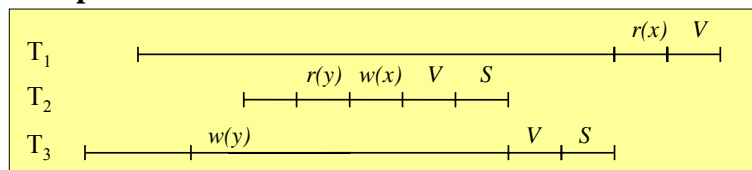
- Validierung muss nur von ändernden Transaktionen durchgeführt werden.
- die überflüssigen Rücksetzungen von BOCC werden vermieden
- überflüssige Rücksetzungen wegen der vorgegebenen Serialisierungs-Reihenfolge sind weiterhin möglich
- mehr Freiheiten bei der **Konfliktauflösung**: beliebige TA kann abgebrochen werden, z.B.
 - **Kill**-Ansatz: die noch laufenden TAs werden abgebrochen.
 - **Die**-Ansatz: die validierende TA wird abgebrochen (“stirbt”).
 - Verhindern von Verhungern: z.B. Anzahl der Rücksetzungen einer TA beachten.



FOCC (3)

• Beispiel

V: Validierung
S: Schreibphase



• Ablauf:

- T₂ wird erfolgreich validiert, da x noch von keiner TA gelesen wurde.
- T₃ wird erfolgreich validiert, da y von keiner (noch) laufenden TA gelesen wurde.
- T₁ ist eine reine Lese-TA und muss nicht validiert werden.
- Hätte T₂ das Objekt y auch geändert, so wäre der Konflikt mit T₃ bei der Validierung von T₂ erkannt worden, und eine der beiden TAs hätte abgebrochen werden müssen



Abschließende Bemerkungen

- **Qualitätsmerkmale von Synchronisationsverfahren**
 - Effektivität: Serialisierbarkeit, Vermeidung von Anomalien
 - Parallelitätsgrad (Blockierung nebenläufiger TAs)
 - Verklemmungsgefahr
 - Häufigkeit von Rücksetzungen; Vermeidung überflüssiger Rücksetzungen
 - Benachteiligung bestimmter (z.B. langer) TAs (“Verhungern”) durch lange Blockierungen oder häufige Rücksetzungen
 - Verwaltungsaufwand für die Synchronisation (Sperrern, Zeitstempel, ...)
- **Praktische Bewertung**
 - Oft Implementierungsprobleme für feinere Granul. als DB-Seiten
 - Kombinationen der Verfahren teilweise möglich (z.B. “*Optimistic Locking*”, IMS Fast Path)
 - Synchronisation von Indexstrukturen als eigenes Problem
 - nahezu alle kommerziellen DBS setzen auf Sperrverfahren