

Ein Konflikt liegt vor : - auf dasselbe Objekt zugreifen
- aus verschiedenen Transaktionen
- min. ein write

1 Conflict Graphs

Betrachten Sie die folgenden Schedules $S_1 - S_3$, die in einem kompakteren Format dargestellt sind als auf den Folien. Zum Beispiel:

- $r_x(A)$ bedeutet, dass Transaktion T_x das Datenelement A liest.
- Wir lassen lokale Variablen weg, um Informationen zu repräsentieren, die gelesen oder geschrieben werden, da diese Informationen für den Conflict Graph nicht relevant sind.
- $r_x(A) \rightarrow w_x(A)$ stellt die Reihenfolge der Operationen dar, d.h., die Leseoperation in diesem Beispiel wird zuerst ausgeführt, die Schreiboperation danach.
- c_x steht für die Commit-Operation der Transaktion T_x .

1. $S_1 := r_1(A) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow r_3(B) \rightarrow w_3(C) \rightarrow c_1 \rightarrow r_2(C) \rightarrow c_2 \rightarrow c_3$
2. $S_2 := r_3(B) \rightarrow r_3(A) \rightarrow w_4(B) \rightarrow r_4(C) \rightarrow r_2(D) \rightarrow w_1(C) \rightarrow r_1(C) \rightarrow r_1(A) \rightarrow c_1 \rightarrow w_2(D) \rightarrow c_2 \rightarrow w_3(A) \rightarrow c_3$
3. $S_3 := r_1(A) \rightarrow w_1(A) \rightarrow r_3(C) \rightarrow r_2(A) \rightarrow w_3(C) \rightarrow c_3 \rightarrow w_1(B) \rightarrow w_1(C) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2 \rightarrow c_1$

Für jeden dieser Schedules erstellen Sie bitte einen Conflict Graph und entscheiden Sie, ob der Schedule conflict serializable ist. Wenn er conflict serializable ist, geben Sie ein Beispiel für einen conflict equivalent serial Schedule an und ein weiteres Beispiel für einen seriellen Schedule der nicht conflict equivalent ist.

2 Recoverable and Cascadeless Schedules

Gegeben sind die folgenden Schedules S_4 and S_5 . Bitte entscheiden und erklären Sie ob jeder dieser Schedules recoverable und/oder cascadeless sind.

1. $S_4 := w_1(A) \rightarrow r_1(C) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_3(B) \rightarrow r_3(A) \rightarrow w_2(C) \rightarrow c_3 \rightarrow c_2$
2. $S_5 := r_2(A) \rightarrow w_2(B) \rightarrow r_1(A) \rightarrow r_1(B) \rightarrow c_2 \rightarrow c_1$

3 Gleichzeitige Ausführung von Transaktionen

Begründe die folgende Aussage: Die gleichzeitige Ausführung von Transaktionen ist wichtiger, wenn Daten von (langsamen) Festplatten abgerufen werden müssen oder wenn Transaktionen lange dauern, und weniger wichtig, wenn Daten im Speicher sind und Transaktionen sehr kurz sind.

4 Zustände von Transaktionen

Während ihrer Ausführung durchläuft eine Transaktion mehrere Zustände, bis sie schließlich abgeschlossen (commit) oder abgebrochen (abort) wird. Listen Sie alle möglichen Zustandssequenzen auf, durch die eine Transaktion laufen kann. Erklären Sie, warum jeder Zustandswechsel auftreten kann.

5 ACID Implementation

Datenbanksystem-Implementierer:innen haben den ACID-Eigenschaften deutlich mehr Aufmerksamkeit gewidmet als Dateisystem-Implementierer:innen. Warum könnte das der Fall sein?

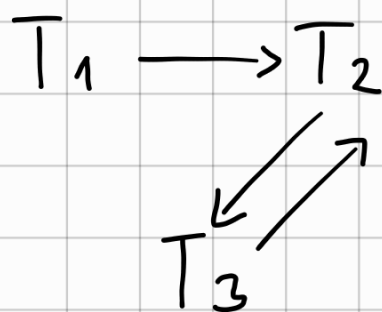
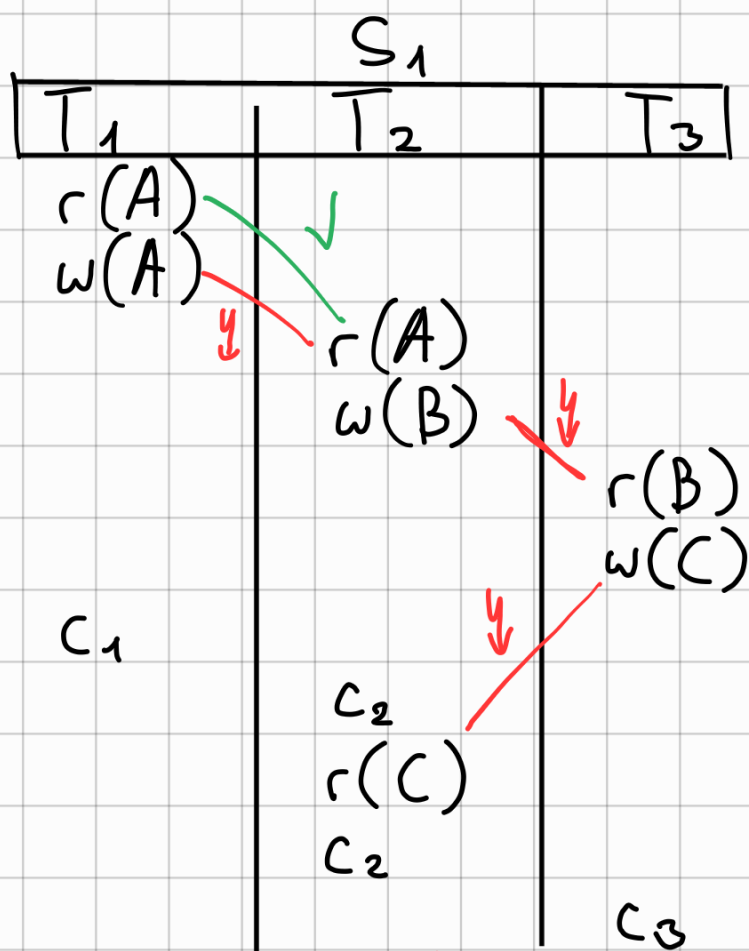
konfliktäquivalent := S_1 & S_2 sind so, wenn sie dieselben Transaktionen & Operationen enthalten und die Reihenfolge aller konfliktverursachenden Operationen ist in beiden gleich

conflict serializable := wenn der Schedule konfliktäquivalent zu einem seriellen Schedule ist
 prüfen mit **zyklusfreier Konfliktgraph**
 durch Umformen \Rightarrow **seriellen Schedule**

Commit-Operationen haben keinen Einfluss auf Konflikte

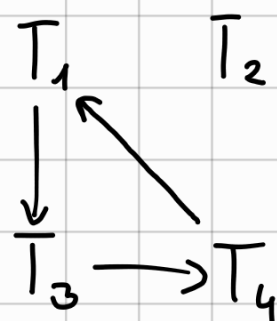
1)

$S_1 := r_1(A) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow r_3(B) \rightarrow w_3(C) \rightarrow c_1 \rightarrow c_2 \rightarrow r_2(C) \rightarrow c_2 \rightarrow c_3$



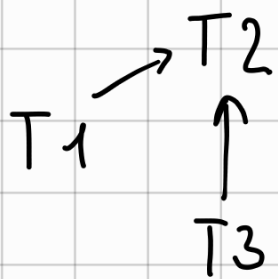
nicht conflict serializable, da ein Zyklus auftaucht

$S_2 := r_3(B) \rightarrow r_3(A) \rightarrow w_4(B) \rightarrow r_4(C) \rightarrow r_2(D) \rightarrow w_1(C) \rightarrow$
 $r_1(C) \rightarrow r_1(A) \rightarrow c_1 \rightarrow w_2(D) \rightarrow c_2 \rightarrow w_3(A) \rightarrow c_3$



nicht conflict serializable, da ein Zyklus auftaucht

$S_3 := r_1(A) \rightarrow w_1(A) \rightarrow r_3(C) \rightarrow r_2(A) \rightarrow w_3(C) \rightarrow c_3 \rightarrow$
 $w_1(B) \rightarrow w_1(C) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2 \rightarrow c_1$



conflict serializable,
da kein Zyklus auftritt

$T_3 \rightarrow T_1 \rightarrow T_2$ conflict equivalent serial Schedule

$T_1 \rightarrow T_2 \rightarrow T_3$ KEIN
conflict equivalent serial Schedule

Not needed ↓

Man darf zwei Operationen tauschen, wenn sie keinen Konflikt haben.

$r_1(A) \rightarrow w_1(A) \rightarrow r_3(C) \rightarrow r_2(A) \rightarrow w_3(C) \rightarrow c_3 \rightarrow$
 $w_1(B) \rightarrow w_1(C) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2 \rightarrow c_1$

$r_1(A) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow r_3(C) \rightarrow w_3(C) \rightarrow c_3 \rightarrow$
 $w_1(B) \rightarrow w_1(C) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2 \rightarrow c_1$

$r_1(A) \rightarrow w_1(A) \rightarrow w_1(C) \rightarrow w_1(B) \rightarrow w_3(C) \rightarrow c_3 \rightarrow$
 $r_3(C) \rightarrow r_2(A) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2 \rightarrow c_1$

$r_1(A) \rightarrow w_1(A) \rightarrow w_1(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow$
 $w_3(C) \rightarrow r_3(C) \rightarrow c_3 \rightarrow$
 $r_2(A) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2$

conflict equivalent serial schedule

$r_1(A) \rightarrow r_2(A) \rightarrow w_1(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow w_3(C) \rightarrow r_3(C) \rightarrow c_3 \rightarrow$
 $w_1(A) \rightarrow r_2(B) \rightarrow w_2(B) \rightarrow w_2(C) \rightarrow c_2$

man darf sie nicht
tauschen

Wenn T_i fehlschlägt $\Rightarrow T_i$ zurücksetzen

Wenn eine andere T_j Daten gelesen hat, die von T_i geschrieben wurden, dann muss auch T_j zurückgesetzt werden.

Wenn kein c_i da steht, weiß man nicht sicher, ob die Transaktion erfolgreich war

Recoverable

! Wenn T_j Daten von T_i liest, dann muss c_i vor c_j kommen

S2

$S_4 := w_1(A) \rightarrow r_1(C) \rightarrow \underline{c_1} \rightarrow r_2(A) \rightarrow w_3(B) \rightarrow r_3(A) \rightarrow w_2(C) \rightarrow c_3 \rightarrow c_2$

$w_1(A) \rightarrow \dots \rightarrow r_2(A)$	c_1 vor c_2	} c_1 muss an erster Stelle ✓
$w_1(A) \rightarrow \dots \rightarrow r_3(A)$	c_1 vor c_3	

es ist egal, ob c_3 oder c_2 als nächstes kommen

\Rightarrow recoverable

Da vor c_1 nichts eingelesen wird, muss in anderen Transaktionen nichts zurückgesetzt werden

\Rightarrow cascadeless

$S_5 := r_2(A) \rightarrow w_2(B) \rightarrow r_1(A) \rightarrow r_1(B) \rightarrow c_2 \rightarrow c_1$

$w_2(B) \rightarrow \dots \rightarrow r_1(B)$ c_2 vor c_1 ✓ \Rightarrow recoverable

da $r_1(B)$ vor c_2 kommt, müsste man in T_1 auch B zurücksetzen, wenn T_2 fehlschlägt

\Rightarrow nicht cascadeless

3 Gleichzeitige Ausführung von Transaktionen

Begründe die folgende Aussage: Die gleichzeitige Ausführung von Transaktionen ist wichtiger, wenn Daten von (langsamen) Festplatten abgerufen werden müssen oder wenn Transaktionen lange dauern, und weniger wichtig, wenn Daten im Speicher sind und Transaktionen sehr kurz sind.

Gleichzeitige Ausführung **wichtiger**

Daten von langsame Festplatten
oder
langandauernde Transaktionen

} mehr
Zeit

Gleichzeitige Ausführung **weniger wichtig**

Daten im Arbeitsspeicher
oder
kurze Transaktionen

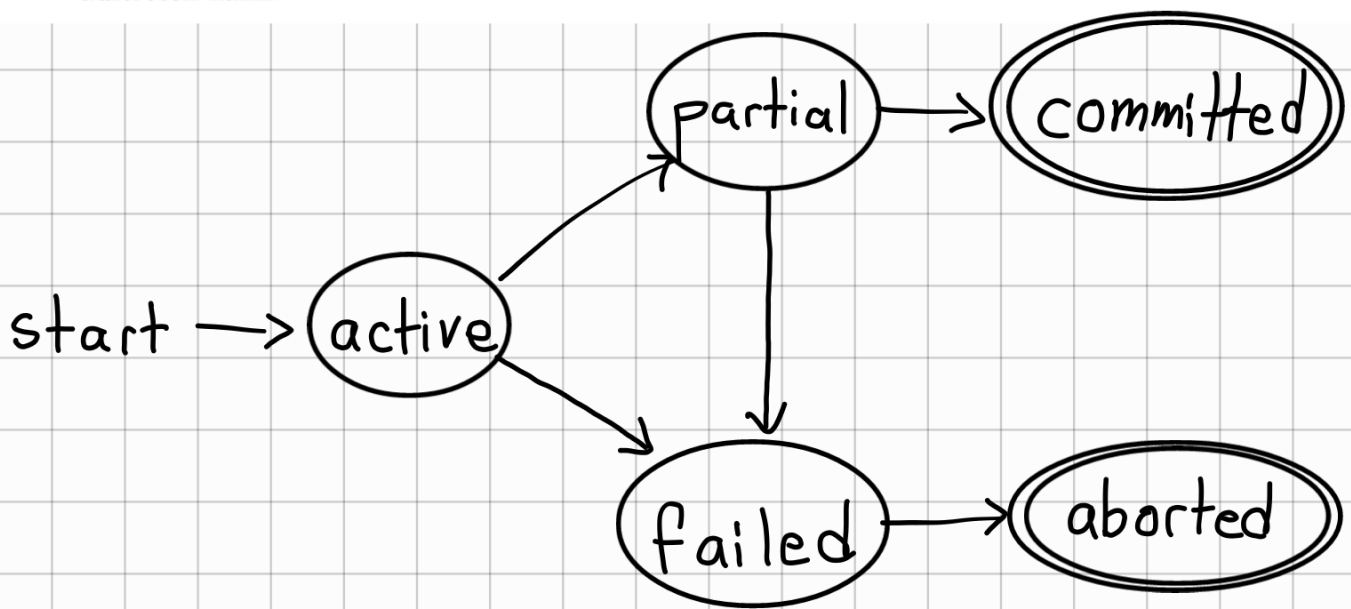
} weniger
Zeit

Das Ziel der nebenläufige Prozesse, ist es den Zeitaufwand für die Prozeduren zu minimieren.

Kurz: Verbesserte Effizienz

4 Zustände von Transaktionen

Während ihrer Ausführung durchläuft eine Transaktion mehrere Zustände, bis sie schließlich abgeschlossen (commit) oder abgebrochen (abort) wird. Listen Sie alle möglichen Zustandssequenzen auf, durch die eine Transaktion laufen kann. Erklären Sie, warum jeder Zustandswechsel auftreten kann.



Erfolgloser Abschluss der Transaktion: **Abort / Fehler**

Active: Transaktion gestartet
wechselt zu **partial**, wenn alle Operationen ausgeführt sind
wechselt zu **failed**, wenn ein Fehler auftritt
z.B. Constraint-Verletzung

Partial: alles außer Commit ausgeführt
wechselt zu **committed**, wenn erfolgreich gespeichert
wechselt zu **failed**, Systemfehler / Stromausfall bevor Speichern

Failed: Fehler tritt auf & Transaktion kann nicht weitergeführt werden

Aborted: Transaktion wurde vollständig zurückgesetzt

Committed: Transaktion wurde erfolgreich abgeschlossen

5 ACID Implementation

strukturiert

Datenbanksystem-Implementierer:innen haben den ACID-Eigenschaften deutlich mehr Aufmerksamkeit gewidmet als Dateisystem-Implementierer:innen. Warum könnte das der Fall sein?

eher unstrukturiert

A
C
I
D
atomicity
consistency
isolation
durability

Datenbanksysteme (Sicherheit & Einheitlichkeit)

- kritische, strukturierte Daten z.B. Banktransaktionen

kleine Fehler

Struktur muss erhalten bleiben,
damit Abfragen & Transaktionen sich
auf die richtigen Daten verlassen können

hoher Schaden

ACID => Korrektheit bei kritischen Transaktionen

Dateisysteme (Effizienz beim Datenzugriff)

- Dateisysteme wie NTFS verwalten Dateien
wie z.B. Bilder, Word-Dokumente usw.
↳ diese sind nicht verbunden

- Beim speichern wird nur die betroffene Datei betrachtet,
man muss nicht hinweg schauen und nebenbei
auch andere Dateien in Betracht ziehen

- Bei einem Fehler ist nur diese Datei betroffen und
nicht das gesamte System

DBMS

DS

A alles oder nichts

Stromausfall -> beschädigte Datei

C nur gültige Zustände

keine semantische Bedeutung

I Keine zwei Prozesse greifen
gleichzeitig auf dieselben Ressourcen

Zwei Prozesse können
gleichzeitig auf Datei zugreifen

D dauerhaft speichern

dauerhaft speichern