Kap. 5: Transaktionsmanagement

Serialisierbarkeit

- "Wenn das Ergebnis des **Schedules** gleich einem Ergebnis eines **seriellen Schedules** ist"
- Ein **Schedule** ist **seriell** wenn die **Schritte je einer Transaktion unmittelbar aufeinander Folgen** und **nicht** mit anderen **Transaktionen verschachtelt** sind.
- Schedule heißt **serialisierbar**, wenn das Ergebnis äquivalent zu dem eines seriellen Schedules ist

Scheduler

- Ein **Schedule S ist konfliktserialisierbar**, wenn er konfliktäquivalent zu einem seriellen Schedule ist.
- Wenn man S durch **Vertauschung von Operationen**, die **nicht in Konflikt zueinanderstehen**, in einen **seriellen Schedule umwandeln** kann, ist S **konfliktserialisierbar**.

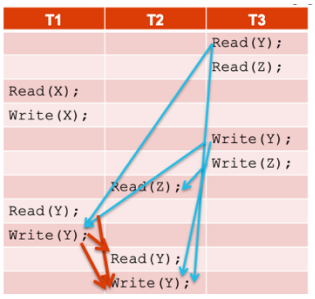
Konflikte zwischen Operationen

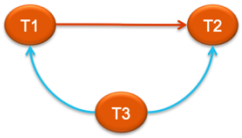
- **READ** und **WRITE** stehen im Konflikt zueinander!
- **WRITE** und **READ** stehen im Konflikt zueinander!
- **WRITE** und **WRITE** stehen im Konflikt zueinander!
- Wenn der Graph **zyklenfrei** ist, ist **S konfliktserialisierbar**

```
R - Read
W - Write
C - Commit
A - Abort
```

Serialisierbarkeitskriterium als Algorithmus

- Für jede Transaktion T_i wird ein Knoten erzeugt
- Es wird eine Kante (T_i, T_j) erzeugt, wenn es in S ein $R_j(X)$ nach einem $W_i(X)$ gibt.
- Es wird eine Kante (T_i, T_j) erzeugt, wenn es in S ein $W_j(X)$ nach einem $R_i(X)$ gibt.
- Es wird eine Kante (T_i, T_j) erzeugt, wenn es in S ein $W_j(X)$ nach einem $W_i(X)$ gibt.
- Zur verbesserten Übersicht kann die Kante mit dem jeweiligen Objekt, das den Konflikt hervorruft, beschriftet werden





- Wenn Transaktionen nichts machen, sind sie serialisibar, aber nicht konfliktserialisierbar!

Abbruch von Transaktionen

- Zusätzlich muss der Scheduler die Rücksetzbarkeit garantieren, damit fehlerhafte Transaktionen abgebrochen werden können
- Weiter ist es sinnvoll, kaskadierende Abbrüche zu vermeiden
- Eine Transaktion ist rücksetzbar, wenn diese den gelesenen Wert erst comitted nachdem die vorige Transaktion auch commitet hat.
- Ein Schedule vermeidet kaskadierende Abbrüche wenn nur commited Werte gelesen werden

ok

T1	T2
	Read(X);
	Write(X);
Read(X);	
Write(X);	
	Read(Y);
	Write(Y);
Read(Y);	
Write(Y);	
	Commit;
Abort;	

kaskadierender Abbruch

T1	T2
	Read(X);
	Write(X);
Read(X);	
Write(X);	
	Read(Y);
	Write(Y);
Read(Y);	
Write(Y);	
	Abort;
→ Abort;	

nicht rücksetzbar

T1	T2
	Read(X);
	Write(X);
Read(X);	
Write(X);	
	Read(Y);
	Write(Y);
Read(Y);	
Write(Y);	
Commit;	
	Abort;

b) Synchronisationsverfahren

Transaktionsmanager – Operationen:

- `EXECTE` → `TA ausfuehren`
- `DELAY` → `TA verzoegert`
- `REJECT` → `führt zu abort`

Sperrverfahren (pessimistische Verfahren)

- Jede Transaktion **sperrt jedes Element** vor dem Bearbeiten und **gibt es danach frei**.
- Keine Transaktion darf auf ein von einer anderen Transaktion gesperrtes Element zugreifen
- Operationen: `lock(x);` und `unlock(x);`

Ausgangspunkt

T1	T2
Read(a);	
	Read(a);
Write(a);	
	Write(a);

Mit Sperren

T1	T2
Lock(a);	
Read(a);	
	Lock(a);
Write(a);	
Unlock(a);	
	Read(a);
	Write(a);
	Unlock(a);

T1	T2
R(a);	
W(a);	
	R(a);
	W(a);
	R(b);
	W(b);
R(b);	
W(b);	

	T1	T2
(1)	L(a)	
(2)	R(a)	
(3)	W(a)	
(4)	Ul(a)	
(5)		L(a)
(6)		R(a)
(7)		W(a)
(8)		Ul(a)

Nicht-Serialisierbarer Schedule trotz Sperren!

	T1	T2
(9)		L(b)
(10)		R(b)
(11)		W(b)
(12)		Ul(b)
(13)	L(b)	
(14)	R(b)	
(15)	W(b)	
(16)	Ul(b)	

Zwei-Phasen-Sperrprotokoll

Zwei-Phasen-Sperrprotokoll (two-phase lock protocol, 2PL):

1. Vor dem ersten Zugriff auf ein Objekt muss die Transaktion das Objekt sperren
2. Nach dem ersten unlock darf kein Objekt mehr gesperrt werde
3. Am Ende alle Sperren aufheben

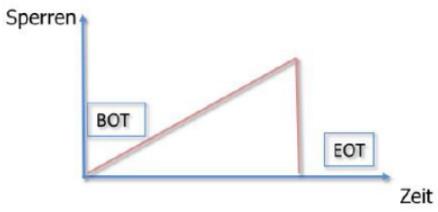
Allgemeine Variante

T1		T2	
L(a);	R(a);		
	W(a);		
Ul(a);			
		L(a);	R(a);
			W(a);
		Ul(a);	
		L(b);	R(b);
			W(b);
		Ul(b);	
L(b);	R(b);		
	W(b);		
Ul(b);			

T1		T2	
L(a);	R(a);		
	W(a);		
		L(a);	
L(b); Ul(a);		...	
			R(a);
			W(a);
		L(b);	
	R(b);	...	
	W(b);		
Ul(b);			
		Ul(a);	R(b);
			W(b);
		Ul(b);	

Strikte Zweiphasigkeit

- Alle Sperren werden erst zum Transaktionsende freigegeben.
  → keine kaskadierende Abbrüche möglich



Preclaming

- Alle Sperren werden von Beginn an gesperrt

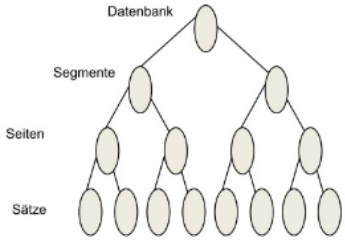
Mehrfachmodussperren

- **Idee: Schreib / Lesesperre unterscheiden**

Lesesperre: Jeder darf lesen; Es wird Counter für Anzahl Leser mitgeführt
Schreibsperre: Niemand darf lesen oder schreiben
`READ_LOCK(X);` Lesesperren!
`WRITE_LOCK(X);` Schreibsperre!
`UNLOCK(X);` Gibt eine gesezte Sperre wieder frei!

Sperreinheiten

- Greift eine Transaktion nur auf ein einziges Tupel zu, so soll nur dieses Tupel gesperrt werden.
- logische Einheiten: Attribute, Tupel, Relation, Datenbank
- physische Einheiten: Seite, Datei, Datenbank



Übliche Sperrebene: Tupel

Multiple-Granularity Locking (MGL)

- Sperrobjekte können sich überlappen
- `irl` (intentionale Lesesperre): später kommt eine Lesesperre (`rl`)
- `iwl` (intentionale Schreibsperre): später kommt eine Schreibsperre (`wl`)

Kompatibilität der Sperrmodi des MGL

Sperrung „top-down“, Freigabe „bottom-up“

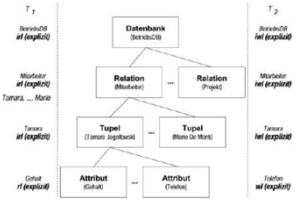
1. Sperren werden auf einem Pfad in der Reihenfolge von der Wurzel zum Ziel gesetzt
2. Das Objekt wird gesperrt
3. Alle anderen Knoten auf dem Pfad bekommen intentionale Sperren.
4. Sperren können verschärft werden (**`rl->wr, irl->rl`**)
5. Die Freigabe erfolgt in umgekehrter Reihenfolge.

Protokolle zur Vermeidung von Konflikten auch hier erforderlich (z.B. 2PL).

	rl_i	wl_i	irl_i	iwl_i
rl_j	✓	✗	✓	✗
wl_j	✗	✗	✗	✗
irl_j	✓	✗	✓	✓
iwl_j	✗	✗	✓	✓

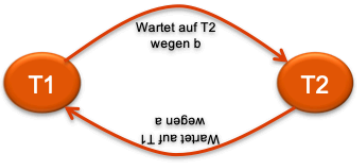
Hierarchisches Sperren

T1 liest die gesamte Relation Mitarbeiter
T2 will einen Mitarbeiter aktualisieren



Probleme bei Sperrverfahren (Zyklisches Warten)

Deadlock - Tritt auf, wenn zwei Transaktionen jeweils auf die andere warten.



Behandlung von Deadlocks

- TA-Manager veranlasse bei einer der Transaktionen: **Abbruch und Rücksetzen**
- **Späterer Neustart** erforderlich (durch TA-Manager oder Client)

Erkennung von Deadlocks

Time - out: Wartezeit einer Transaktion T auf ein Objekt „zu lange“

- Transaktionsmanager schließt auf Beteiligung an Deadlock, bricht T ab
- kritisch: Wahl der Wartezeit

Wartegraph: Deadlock = Zyklus im Wartegraph

- kritisch: Prüfzeitintervalle, Auswahl der Transaktion, die abgebrochen werden soll (Kostenfunktionen)

Zeitstempelverfahren (optimistisches Verfahren)

- Transaktionen werden auf Basis der zeitlichen Reihenfolge, in der sie in das DBMS kommen, synchronisiert
- Jede Transaktion T erhält einen eindeutigen Zeitstempel **`TS(T)`**
- Jede Operation wird mit dem **`TS`** der Transaktion versehen
- Jedes Objekt besitzt zwei **`TS`**
  - **`TSR(X)`** = `Letzter Lesezugriff`
  - **`TSW(X)`** = `Letzter Schreibzugriff`

T1 (t_s=200)	T2 (t_s=150)	T3 (t_s=175)	Objekt A		Objekt B		Objekt C	
			TSR	TSW	TSR	TSW	TSR	TSW
Read(B)			0	0	200	0	0	0
	Read(A)		150	0	200	0	0	0
		Read(C)	150	0	200	0	175	0
Write(B)			150	0	200	200	175	0
Write(A)			150	200	200	200	175	0
	Write(C)		150	200	200	200	175	ABORT

Timestamp Ordering TO) – Algorithmus

Lese-Operation

```
IF TS(T) < TSW(X) THEN abort T
ELSE execute read; TSR(X):= max{TSR(X),TS(X)} END
```

Schreib-Operation

```
IF TS(T) < max{TSR(X),TSW(X)} THEN abort T
ELSE execute write; TSW(X):= TS(X) END
```

- Kaskadierender Abbruch möglich

Multi-Version Concurrency Control

Idee: Wenn ältere Versionen eines Objektes aufgehoben werden, müssen nur Schreibzugriffe synchronisiert werden Implementierung (Achtung: dies beinhaltet nicht die Synchronisation)

- Jede Transaktion hat einen Anfangszeitstempel
- Jedes Objekt ist in mehreren Versionen vorhanden, mit Zeitstempel
- Jede Transaktion liest nur die zu ihrem Anfangszeitstempel passende Version
  - Dies bedeutet: die aktuellste Version mit einem Zeitstempel kleiner oder gleich dem Anfangszeitstempel der Transaktion

T1	T2
Read(x);	
Write(x);	
	Read(x);
	Write(y);
Read(y);	
Write(z);	

- Dieser Ablauf ist nicht konfliktserialisierbar
- Wenn T1 eine alte Version von y lesen kann (Version *vor* dem Write(y) von T2), ist das Problem geheilt
- Lösung wird z.B. in Oracle oder PostgreSQL verwendet
- Bedeutet aber, das ggf. über längere Zeit mehrere Versionen eines Objektes vorgehalten werden müssen
  - Eine veraltete Version kann erst gelöscht werden, wenn alle noch aktiven Transaktionen einen neueren Start-Zeitstempel haben
  - Erfordert aufwändige Garbage Collection

c) Transaktionsmanagement in SQL und Oracle

Transaktionsverwaltung in SQL

```
set transaction
     [{read only | read write},]
     [isolation level {
          read uncommitted |
          read committed |
          repeatable read |
          serializable }]
```

Isolationsebene	Dirty Read	Non-repeatable Read	Phantom Read
`read uncommitted`	Möglich	Möglich	Möglich
`read committed`	Nicht möglich	Möglich	Möglich
`repeatable read`	Nicht möglich	Nicht möglich	Möglich
`serializable`	Nicht möglich	Nicht möglich	Nicht möglich

Read Uncommitted

- darf auch nur für read only- Transaktionen spezifiziert werden.

Read Committed (default)

- Transaktionen lesen nur endgültig geschriebene Werte
- Können unterschiedliche Zustände der Datenbank-Objekte zu sehen bekommen
- Non-repeatbale read kann auftreten und muss verarbeitbar sein

Repeatable Read und Serializable

repeatable read: non-repeatable read wird ausgeschlossen

- Phantom problem kann auftreten
- Wenn eine parallele Änderungstransaktion dazu führt, dass Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.

serializable: garantiert Serialisierbarkeit

Transaktionsverwaltung in ORACLE

Isolationsstufen `read committed` und `serializable`, zusätzlich `read only`

- `set transaction isolation level ...`
- `set transaction read only`

Isolationslevel für jede Transaktion einstellbar oder für eine Menge von Transaktionen

- `alter session set isolation_level ...`

Lese- und Schreibsperren-Verwaltung für Tables und Rows
explizite Kommandos zum Setzen von Sperren möglich

- `select * from movie where movie = 123456 FOR UPDATE;`

Multi-Version Concurrency Control

- Dadurch im Normalbetrieb keine Lesesperren nötig