

6 Transaktionen

- Begriff aus der Welt der Datenbanken
 - ◆ Menge von zusammengehörigen Anweisungen mit bestimmten Eigenschaften
- Motivation
 - ◆ Koordinierung
 - gegenseitiger Ausschluss zu restriktiv: keine Nebenläufigkeit
 - Effekt des gegenseitigen Ausschlusses, aber mehr Nebenläufigkeit
 - ◆ Fehlertoleranz
 - konsistente Operationsausführung auch bei Ausfällen
 - dauerhafte Speicherung konsistenter Ergebnisse bei Ausfällen



6 Transaktionen (2)

- Lösung: Programmiermodell mit Transaktionen
 - ◆ Menge von gekennzeichneten Operationen
 - `openTransaction()` zum Beginn
 - `closeTransaction()` zum erfolgreichen Beenden (*Commit*)
 - `abortTransaction()` zum Abrechnen (*Abort*)
 - ◆ Koordinator (Transaktionsmanager, TP-Monitor)
 - verwaltet Transaktionen
 - kontrolliert Datenzugriffe der Operationen
 - realisiert Ausführungssemantik



6 Transaktionen (3)

- Eigenschaften von Transaktionen (ACID-Eigenschaften nach Härder, Reuter, 1983)
 - ◆ Atomarität (*Atomicity*)
 - Transaktion wird vollständig oder gar nicht ausgeführt
 - ◆ Konsistenz (*Consistency*)
 - Transaktion führt konsistenten Zustand in konsistenten Zustand über
 - ◆ Isolation/Serialisierbarkeit (*Isolation*)
 - Ausführungssemantik wie bei serieller Ausführung (gegenseitiger Ausschluss)
 - ◆ Dauerhaftigkeit (*Durability*)
 - nach erfolgreich beendeter Transaktion sind Ergebnisse dauerhaft gespeichert



6.1 Atomarität

- Wirkung der Transaktion
 - ◆ „Alles oder nichts“-Prinzip
- Implementierung
 - ◆ echt serielle Abarbeitung von Transaktionen
 - nachfolgende Transaktion sieht Effekte der Vorgängertransaktion erst nach erfolgreichem Beenden
 - **aber:** keine Nebenläufigkeit, keine Fehlertoleranz
 - ◆ zusätzlich: Zustandssicherung vor Beginn der Transaktion
 - im Fehlerfall: Rücksetzen auf gesicherten Zustand (Fehlertoleranz)
 - **aber:** keine Nebenläufigkeit
 - ◆ besseres Verfahren?



6.2 Konsistenz

- **Transparenz von inkonsistenten Zwischenzuständen**
 - ◆ Überführung von konsistentem in anderen konsistenten Zustand
 - ◆ durch Atomarität sind Zwischenzustände unsichtbar
- **Konsistenzsicherung**
 - ◆ Kontrolle der Konsistenzbedingungen
 - z.B. Summe aller Kontostände in Buchhaltung ist Null
 - ◆ Verhindern von Commits
 - Abbruch der Transaktion bei inkonsistentem Endzustand
- **Implementierung**
 - ◆ Konsistenzprüfung bei `closeTransaction()`



6.4 Dauerhaftigkeit

- **Permanente Zustandssicherung**
 - ◆ Rücksetzbarkeit auf gesicherte (konsistente) Zustände
- **Implementierung**
 - ◆ Zustandssicherung nach Ablauf einer Transaktion
 - ◆ persistente Speicherung der Sicherung
 - überlebt Rechner und Anwendungsausfall
 - ◆ Rücksetzen im Fehlerfall möglich



6.3 Isolation

- **Zwischenzustände sind transparent**
 - ◆ Ausführungssemantik wie bei serieller Ausführung
- **Implementierung**
 - ◆ echt serielle Ausführung
 - **aber:** keine Nebenläufigkeit, keine Fehlertoleranz
 - ◆ besseres Verfahren?



6.5 Implementierung von Transaktionen

- **Serialisierbarkeit**
 - ◆ gesucht: Abfolge der Einzelaktionen mehrerer Transaktionen, so dass Wirkung äquivalent zur seriellen Transaktionsabwicklung (seriell äquivalente Ausführung)
- **Verletzung der Isolation bei nebenläufigen Abläufen**
 - ◆ lesender oder schreibender Zugriff auf gleiches Datum (read, write)
 - ◆ $read_{T_1} \text{ — } read_{T_2}$
 - **kein Konflikt**, nicht von Ausführungsreihenfolge abhängig
 - ◆ $read_{T_1} \text{ — } write_{T_2}$
 - **Konflikt**: entweder alter oder neuer Wert gelesen
 - ◆ $write_{T_1} \text{ — } write_{T_2}$
 - **Konflikt**: welcher der Werte ist im Endergebnis enthalten?



6.5 Implementierung von Transaktionen (2)

■ Beispiel: „read–write“-Konflikt

```
accountA.balance= 50;
-----
openTransaction();          openTransaction();
s= accountA.getBalance();   accountA.setBalance( 100 );
accountB.setBalance( s );   closeTransaction();
closeTransaction();
```

◆ Zugriff auf Konto **accountA**

- Effekt von Reihenfolge abhängig
- write–read führt zu „Dirty Read“ (Lesen inkonsistenter Daten) und/oder „Unrepeatable Read“ (mehrfaches Lesen gibt verschiedenes Ergebnis)



6.5 Implementierung von Transaktionen (3)

■ Beispiel: „write–write“-Konflikt

```
accountA.balance= 50;
-----
openTransaction();          openTransaction();
accountA.setBalance( 120 );  accountA.setBalance( 100 );
closeTransaction();         closeTransaction();
```

◆ Zugriff auf Konto **accountA**

- Effekt von Reihenfolge abhängig



6.5 Implementierung von Transaktionen (4)

■ Bedingung für Serialisierbarkeit

- ◆ alle konfligierenden Zugriffspaare treten in gleicher Reihenfolge auf (z.B. erst Zugriff von T1, dann Zugriff von T2)

■ Problem bei Abbrüchen

◆ „Dirty Read“

- Abbruch einer Transaktion mit Schreibzugriff
- Folgetransaktion mit Lesezugriff

◆ „Premature Write“

- Abbruch einer Transaktion mit Schreibzugriff setzt vorherigen Wert aus Zustandssicherung
- Folgetransaktion mit Schreibzugriff
- Änderung der Folgetransaktion bleibt ohne Effekt



6.5 Implementierung von Transaktionen (5)

■ Beispiel: „Dirty Read“

```
accountA.balance= 50;
-----
openTransaction();          openTransaction();
accountA.setBalance( 100 );  s= accountA.getBalance();
abortTransaction();         accountB.setBalance( s );
                             closeTransaction();
```

- ◆ Welchen Wert hat der Saldo von **accountB** bei Abbruch der linken Transaktion?



6.5 Implementierung von Transaktionen (6)

- Beispiel: „Premature Write“

```
accountA.balance= 50;
```

```
openTransaction();           openTransaction();  
accountA.setBalance( 100 );  accountA.setBalance( 120 );  
abortTransaction();         closeTransaction();
```

- ◆ Welchen Wert hat der Saldo von **accountA** bei Abbruch der linken Transaktion?

