Transaktionsmanagement - Einführung

Einführung

- Nebenläufige Ausführung von Benutzerprogrammen wesentlich für gute Performance des DBMS
  - Weil Plattenzugriffe häufig und relativ langsam sind: CPU muß ausgelastet werden durch nebenläufige Abarbeitung mehrerer Benutzerprogramme
- Ein Benutzerprogramm kann viele Operationen auf den Daten ausführen, die aus der Datenbank gelesen werden
- DBMS kümmert sich nur darum, welche Daten in der Datenbank gelesen/geschrieben werden
- Eine Transaktion ist die abstrakte Sicht des DBMS auf ein Benutzerprogramm: eine Folge von Reads und Writes (deren Semantik ist für das DBMS irrelevant)
  - Read: Lesens der´s Datums von der Platte in den Puffer und von da in die Programmvariable
  - Write: Eine Kopie des Datums wird im Speicher modifiziert und später auf Platte geschrieben

Nebenläufigkeit in einem DBMS

- Benutzer schicken Transaktionen ab und verstehen jede Transaktion einzeln
  - Nebenläufigkeit wird ermöglicht durch das DBMS, welches Aktionen verschiedener Transaktionen (reads/writes auf DB-Objekten) ineinander verschachtelt
  - Jede Transaktion muß die Datenbank in einem konsistenten Zustand hinterlassen, wenn die DB zum Beginn der Transaktion schon konsistent war
    - DBMS kontrolliert die Einhaltung benutzerdefinierter Integritäts-bedingungen (diese ergeben sich aus der Schemadefinition)
    - Darüber hinaus “versteht“ das DBMS die Semantik der Daten nicht (z.B. versteht nicht, wie der Zins auf einem Bankkonto berechnet wird)
- Probleme:
  - Welche Effekte kann die nebenläufige Ausführung von Transaktionen haben?
  - Was passiert bei einem Crash?

Transaktionsparadigma

- Definition
  - Eine Transaktion ist eine ununterbrechbare Folge von Aktionen (Lese- oder Schreibaktionen), welche die Datenbank von einem logisch konsistenten Zustand in einen neuen logisch konsistenten Zustand überführt.
  - Diese Definition besonders für kurze Transaktionen (z.B. Geldüberweisungen) geeignet,
- ACID-Prinzip
  - Atomicity: ‘Alles-oder-Nichts‘-Eigenschaft
  - Consistency: Eine erfolgreiche Transaktion erhält die DB-Konsistenz (Menge der definierten Integritäts-bedingungen)
  - Isolation: Alle Aktionen innerhalb einer TA müssen vor parallel ablaufenden TAs verborgen werden
  - Durability: Sobald eine TA ihre Änderungen freigegeben hat, muß das System das Überleben dieser Änderungen trotz beliebiger (erwarteter) Fehler garantieren (*Persistenz*)

Atomarität von Transaktionen

- Wichtige Eigenschaft, die vom DBMS garantiert wird
- Eine Transaktion kann beendet werden (*commit*), nach der Beendigung aller ihrer Aktionen.
- Eine Transaktion kann zurückgesetzt werden (oder durch das DBMS abgebrochen werden), *abort*.
- Ein Benutzer versteht seine Transaktion als atomare Einheiten. Transaktion führt alle ihre Aktionen in einem Schritt aus oder führt überhaupt keine Aktion aus
  - Benutzer kann Transaktionsgrenzen in Anwendung festlegen
  - DBMS loggt alle Aktionen, so daß es die Aktionen von abgebrochen Transaktionen (abort) rückgängig machen kann (undo).

Beispiel

- 2 parallel verlaufende Transaktionen:

```
T1:    BEGIN   A=A+100,   B=B-100   END
T2:    BEGIN   A=1.06*A,   B=1.06*B   END
```

- Die erste Transaktion transferiert 100 € vom Konto B zum Konto A. Die zweite Transaktion schreibt beiden Konti 6 % Zinsen gut.
- Es gibt keine Garantie, daß T1 vor T2 ausgeführt wird (oder umgekehrt), wenn beide zusammen gestartet werden. Jedoch gilt: Der Nettoeffekt *muß* äquivalent zu beiden Transaktionen sein, wenn sie seriell in irgendeiner Reihenfolge ablaufen würden.

Beispiel (Forts.)

- Betrachte folgenden Ablauf mit ineinander geschachtelten Transaktionen (*Schedule*):

```
T1:    A=A+100,                    B=B-100
T2:                 A=1.06*A,                  B=1.06*B
```

- Kein Problem, aber nimm dieses Beispiel:

```
T1:    A=A+100,                               B=B-100
T2:                 A=1.06*A, B=1.06*B
```

- Zweiter Schedule aus Sicht des DBMS:

```
T1:    R(A), W(A),                            R(B), W(B)
T2:                 R(A), W(A), R(B), W(B)
```

Scheduling von Transaktionen

- *Serieller Schedule:*
  Schedule, in dem keine Aktionen aus unterschiedlichen Transaktionen ineinandergeschachtelt sind
- *Äquivalente Schedules*:
  In jedem Datenbankzustand ist der Effekt der Ausführung des einen Schedules (auf der Menge der Objekte in der Datenbank) identisch mit dem Effekt der Ausführung eines anderen Schedules
- *Serialisierbarer Schedule*:
  Ein Schedule, der äquivalent zu irgendeiner seriellen Ausführung der Transaktionen ist

(Hinweis: Wenn jede einzelne Transaktion die Konsistenz in der Datenbank bewahrt, kann somit auch jeder serialisierbare Schedule die Konsistenz bewahren)

Anomalien im Mehrbenutzerbetrieb

- Verlorengegangene Änderungen (*Lost Update*)
  - WW-Konflikt
  - Gleichzeitige Änderung desselben Objekts durch 2 Transaktionen
  - Erste Änderung (aus nicht beendeter Transaktion) wird durch die zweite überschrieben

```
T1:     W(A),                    W(B), C
T2:             W(A), W(B), C
```

- Zugriff auf schmutzige Daten (*Dirty Read*)
  - WR-Konflikt
  - “schmutzige“ Daten = geänderte Objekte, deren Änderungen von Transaktionen stammen, die noch nicht beendet sind
  - Dauerhaftigkeit der Änderungen nicht garantiert, da Transaktionen noch zurückgesetzt werden
  - Ungültige Daten werden somit durch andere Transaktion gelesen und (schlimmer noch!) vielleicht noch weiterpropagiert

```
T1:     R(A), W(A),                    R(B), W(B), Abort
T2:                   R(A), W(A), C
```

Anomalien im Mehrbenutzerbetrieb (Forts.)

- Nicht-wiederholbares Lesen (*Unrepeatable Read*)
  - RW-Konflikt
  - Eine Transaktion sieht (bedingt durch parallele Änderungen) während ihrer Ausführung unterschiedliche Zustände des Objekts. Erneutes Lesen in der Transaktion liefert somit anderen Wert

T1:	R(A),	R(A), W(A), C
T2:	R(A), W(A), C	

- Phantom-Problem
  - Spezielle Form des Unrepeatable Read
  - Lesetransaktion: Mengenorientiertes Lesen über ein bestimmtes Suchprädikat P
  - Parallel läuft Änderungstransaktion, die die Menge der sich für das Prädikat qualifizierenden Objekte ändert
  - Folge: Phantom-Objekte, die die durch parallele Einfüge- oder Löschvorgänge in Ergebnismenge auftauchen oder daraus verschwinden

Lost Update Beispiel

Gehaltsänderung T1

```
SELECT GEHALT
   INTO :gehalt
FROM PERS
WHERE PNR=2345

gehalt:=gehalt+2000;


UPDATE PERS
SET GEHALT=:gehalt
WHERE PNR=2345
```

Gehaltsänderung T2

```
SELECT GEHALT
   INTO :gehalt
FROM PERS
WHERE PNR=2345


gehalt:=gehalt+1000;


UPDATE PERS
SET GEHALT=:gehalt
WHERE PNR=2345
```

Zeit

DB-Inhalt
(PNR, GEHALT)

2345	39.000
2345	41.000
2345	40.000

Dirty Read Beispiel 1

Gehaltsänderung T1

```
UPDATE PERS
SET GEHALT=
   GEHALT+1000
WHERE PNR=2345

. . .

ROLLBACK
```

Gehaltsänderung T2

```
SELECT GEHALT
   INTO :gehalt
FROM PERS
WHERE PNR=2345

gehalt:=gehalt*1.05;

UPDATE PERS
SET GEHALT=:gehalt
WHERE PNR=3456
COMMIT
```

Zeit

DB-Inhalt
(PNR, GEHALT)

2345	39.000
2345	40.000
3456	42.000
2345	39.000

Dirty Read Beispiel 2

Gehaltsänderung T1

```
UPDATE PERS
SET GEHALT=
   GEHALT+1000
WHERE PNR=2345

        . . .

UPDATE PERS
SET GEHALT=
   GEHALT+2000
WHERE PNR=3456

COMMIT
```

Gehaltssumme T2

```
SELECT SUM(GEHALT)
   INTO :summe
FROM PERS
WHERE PNR IN
   (2345,3456)
```

Inkonsistente Analyse

Zeit

DB-Inhalt
(PNR, GEHALT)

2345	39.000
3456	45.000
2345	40.000
3456	47.000

Unrepeatable Read Beispiel

Gehaltsänderung T1

```
UPDATE PERS
SET GEHALT=
   GEHALT+1000
WHERE PNR=2345

UPDATE PERS
SET GEHALT=
   GEHALT+2000
WHERE PNR=3456

COMMIT
```

Gehaltssumme T2

```
SELECT GEHALT INTO :g1
FROM PERS
WHERE PNR=2345

SELECT GEHALT INTO :g2
FROM PERS
WHERE PNR=3456
summe:=g1+g2
```

DB-Inhalt
(PNR, GEHALT)

2345	39.000
3456	45.000
2345	40.000
3456	47.000

Zeit

Inkonsistente Analyse ohne schmutziges Lesen

Phantom-Problem Beispiel

Lesetransaktion
(Gehaltssumme bestimmen)

```
SELECT SUM(GEHALT) INTO :summe1
FROM PERS
WHERE ANR=17

                . . .


SELECT SUM(GEHALT) INTO :summe2
FROM PERS
WHERE ANR=17

IF summe1<> summe2 THEN
     <Fehlerbehandlung>
```

Änderungstransaktion
(Einfügen eines neuen Angestellten)

```
INSERT INTO PERS
  (PNR,ANR,GEHALT)
VALUES(4567,17,55.000)
```

Zeit

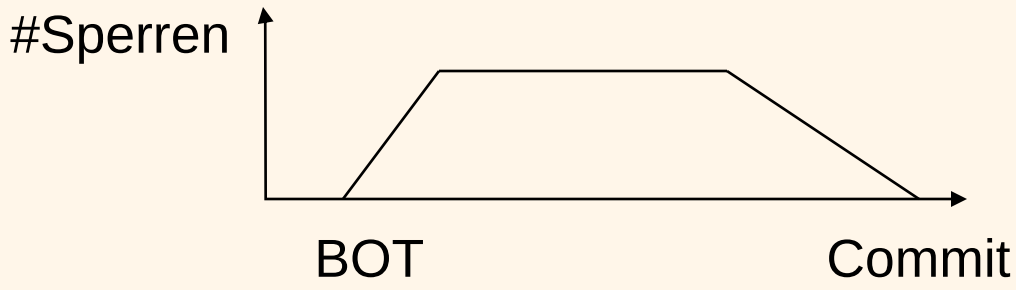
Sperrenbasierte Synchronisation

- Idee von Sperrverfahren:
  - Vor dem Zugriff auf ein Objekt muß dafür durch die Transaktion eine Sperre erworben werden
- Fundamentalsatz des Sperrens (nach Eswaran 1976)
  - Jedes zu referenzierende Objekt muß vor dem Zugriff mit einer Sperre belegt werden.
  - Die Sperren anderer Transaktionen sind zu beachten. Somit muß eine Sperranforderung, die mit gesetzten Sperren unverträglich ist, auf die Freigabe dieser Sperren warten.
  - Keine Transaktion fordert eine Sperre an, die sie bereits besitzt.
  - Sperren werden zweiphasig angefordert und freigegeben.
  - Spätestens bei Transaktionsende gibt eine Transaktion alle Sperren zurück.

Es liegt die Annahme einer fehlerfreien Betriebsumgebung zugrunde.

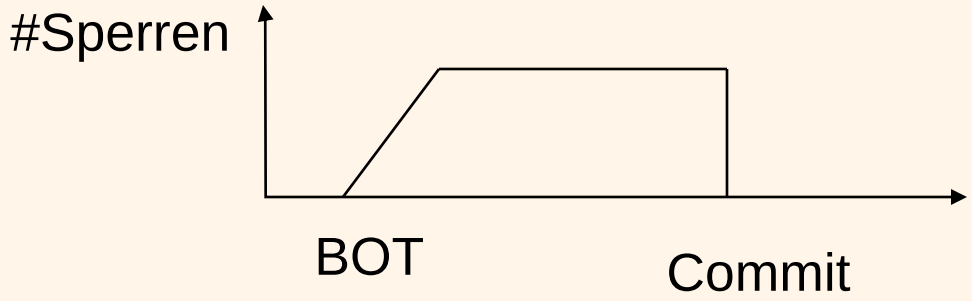
Zwei-Phasen-Sperrverfahren

- *2-Phase-Locking (2PL*)
- 1. Phase: Wachstumsphase
  - Transaktion darf Sperren anfordern, aber keine freigeben
- 2. Phase: Schrumpfungsphase
  - Transaktion darf Sperren freigeben, aber keine neuen anfordern
- Problem:
  - Zugrunde liegt Annahme, daß Betrieb fehlerfrei
  - Bei Systemausfall kann es zu *kaskadierendem Abort* komme
    - Transaktionen, die schon freigegebene Daten haben (und vielleicht selbst schon normal beendet wurden) müssen zurückgesetzt werden



Striktes 2-Phasen-Sperren

- Idee:
  - Vermeidung von kaskadierendem Abort, falls Systemabsturz in der Freigabephase
- Protokoll:
  - Jede TA muß eine S-Sperre (*shared lock*) auf dem Objekt vor dem Lesen erwerben, und eine X-Sperre (*exclusive lock*) vor dem Schreiben
  - Alle Sperren werden bei Beendigung der Transaktion (d.h. zum Commit-Zeitpunkt) freigegeben
  - Wenn eine Transaktion eine X-Sperre auf einem Objekt hält, kann keine andere Transaktion eine (S oder X-) Sperre auf diesem Objekt erwerben
- Nur Striktes 2PL erlaubt serialisierbare Schedules
  - Beweis in einschlägiger DB-Literatur



Zurücksetzen einer Transaktion

- Kaskadierender Abort:
  - Wenn eine Transaktion *Ti* zurückgesetzt wird (abort), müssen alle ihre Aktionen rückgängig gemacht werden (undo).
  - Außerdem muß eine Transaktion *Tj* zurückgesetzt werden, das ein Objekt liest, das zuletzt von *Ti* geschrieben wurde
- Lösung durch Sperrenfreigabe nur zur Commit-Zeit
  - Wenn *Ti* ein Objekt schreibt, kann *Tj* dieses nur lesen, nachdem *Ti* commit gemacht hat
- Undo aller Aktionen beim Abort:
  - Führen von Log-Daten einer Transaktion: Aufzeichnung aller Write-Aktionen
  - Auch nutzbar für die Wiederherstellung der Daten bei System-Crah (Recovery): alle Transaktionen, die zum Zeitpunkt des System-Crashs aktiv waren, werden beim Wiederanlauf zurückgesetzt

Log-Datei

- Logisches Logging: Speicherung von Änderungsaktionen mit Parametern
  - Log-Sätze zur Beschreibung von Änderungen: alter und neuer Wert
    - Log-Satz muß auf Platte gespeichert werden vor der geänderten Seite
  - Transaktionsbeginn, Transaktions-Commit sowie Transaktions-Rollback
- Log-Sätze
  - Haben variable Länge
  - Haben eindeutige Adresse, *Log Sequence Number (LSN*)
  - Sind über Transaktions-ID verkettet (unterstützt schnelles Transaktions-Undo)
- Logs sind oft *dupliziert* und *archiviert* auf stabilem Speicher
- Alle log-bezogenen Aktivitäten (bzw. alle Synchronisations-maßnahmen wie Sperrenverwaltung, Deadlock-Behandlung) werden transparent durch das DBMS behandelt!

Wiederherstellung nach einem Crash

- 3 Phasen im Recovery-Algorithmus (*Aries*-Ansatz)
  - Analyse-Lauf
    - Sequentielles Lesen (Scan) der Log-Datei vom letzten Checkpoint bis zu ihrem Ende
    - Bestimme die zum Checkpoint-Zeitpunkt laufenden Transaktionen
    - Bestimme davon Gewinner (TA mit Commit-Eintrag im Log) und Verlierer (TA mit Rollback-Satz bzw. ohne Commit-Satz)
    - Ermittle die Seiten, die nach dem Checkpoint geändert wurden (dirty pages)
  - Redo-Lauf
    - Wiederholung der Änderungen, welche noch nicht in den betroffenen Seite vorliegen (Lese Log von hinten nach vorne)
    - Somit sichergestellt, daß alle protokollierten Updates wirksam und auf Platte geschrieben sind
  - Undo-Lauf
    - Zurücksetzen der Verlierer-Transaktionen, die während des Crash aktiv waren (durch Schreiben des *before value*, der im Log-Satz steht)
    - Lesen des Logs von vorne nach hinten bis zum Beginn der ältesten Transaktion, die beim letzten Checkpoint aktiv war

Zusammenfassung

- Synchronisation (Concurrency Control) und Recovery gehören zu den wichtigsten Funktionen, die durch ein DBMS bereitgestellt werden
- Benutzer brauchen sich (fast) nicht um Nebenläufigkeit zu kümmern
  - System erzeugt automatisch Anforderungen und Freigaben von Sperren (*Lock Management*)
  - System plant Aktionen unterschiedlicher Transaktionen in einer Weise, um sicherzustellen, daß die resultierende Ausführung äquivalent zu irgendeiner seriellen Transaktionsausführung ist (*Scheduling*)
- *Write-ahead-Logging* (WAL) wird genutzt, um Aktionen von zurückgesetzten Transaktionen rückgängig zu machen und das System nach einem Crash wieder in einen konsistenten Zustand zu bringen
  - *Konsistenter Zustand*: Nur die Effekte von beendeten Transaktionen sind sichtbar