Teil I – Einführung & Grundlagen

Kapitel 1: Einführung in das Transaktionskonzept

1.1 Was ist eine Transaktion?

- Eine Transaktion ist eine Menge von logisch zusammengehörenden Operationen (zumeist in ein Programm eingebunden, aber auch interaktiv möglich)
  - Im Fall von klassischen Datenbanktransaktionen beziehen sich alle Operationen (Lesen bzw. Schreiben von Datenobjekten) auf dasselbe Datenbanksystem
  - Verteilte Transaktionen erlauben Operationen in verschiedenen Datenbanksystemen
  - Verallgemeinerte Sichtweise: Operationen sind beliebige Dienstaufrufe (semantisch reiche Operationen), nicht unbedingt innerhalb von Datenbanken
- Transaktionen basieren auf einem generischen Abstraktionskonzept zum Kapseln der enthaltenen Operationen, um für diese bestimmte Ausführungsgarantien durchzusetzen
  - Ausführungsgarantien abgeleitet von ACID-Eigenschaften
  - Diese Garantien werden vom System transparent für den Anwendungsentwickler/ Benutzer bereitgestellt
  - Anwendungsentwickler/Benutzer muss lediglich die Transaktionsgrenzen festlegen (BOT = Begin-of-Transaction bzw. EOT = End-of-Transaction), wird aber von allen anderen Aspekten zur Realisierung der Garantien befreit

1.2 Transaktionseigenschaften: ACID

- **A**tomicity (Atomarität)
  - Eine Transaktion wird entweder komplett ausgeführt ("commit") oder erscheint so, als wäre sie nie gestartet worden, d.h. sie hinterlässt keine Effekte ("rollback"). Die Semantik einer Transaktion entspricht also einem "Alles-oder-Nichts".
- **C**onsistency (Konsistenz)
  - Eine Transaktion führt die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand über
- **I**solation
  - Transaktionen, auch wenn parallel ausgeführt, erscheinen so als ob sie isoliert voneinander (d.h. sequentiell) ausgeführt werden
- **D**urability (Dauerhaftigkeit/Persistenz)
  - Nach dem Commit einer Transaktion sind ihre Effekte dauerhaft, d.h. überleben auch Systemabstürze

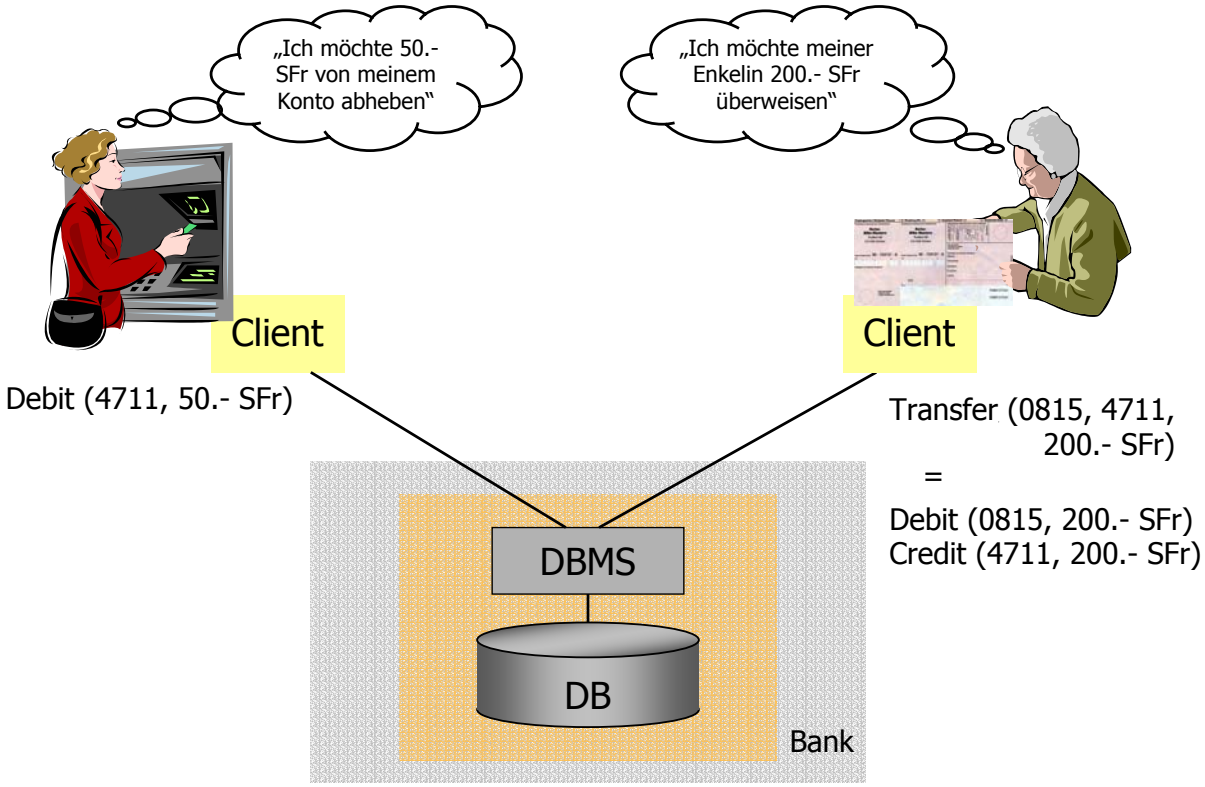
Durchsetzen von Transaktionseigenschaften

- Transaktionen sind eine Art Vertrag zwischen einem Anwendungsprogramm und dem Laufzeitsystem
  - Die ACID-Eigenschaften können dabei als Vertragsinhalt angesehen werden, durch welche die Aufgaben der Laufzeitumgebung festgeschrieben sind
- Die wichtigsten Aufgaben einer Infrastruktur zur Transaktionsverwaltung bestehen aus der
  - korrekten Synchronisation paralleler, konkurrierender Transaktionen (Concurrency Control → Isolation) sowie der
  - korrekten Behandlung von System- und Anwendungsfehlern (Recovery → Atomarität, Persistenz).
  - Diese Unterscheidung spiegelt sich auch in den Komponenten der Infrastruktur zur Transaktionsverwaltung wieder
- Für beide Bereiche (Concurrency Control und Recovery) bedarf es zunächst einer eingehenden Untersuchung, was jeweils "korrekt" bedeutet
  - ➲ Korrektheitskriterien (Concurrency Control: Serialisierbarkeitstheorie, Recovery: Fehlererholung / Crash-Recovery)
- Auf der Basis dieser Korrektheitskriterien müssen dann Systeme entwickelt werden, welche die verlangten Garantien durchsetzen
  - ➲ Concurrency Control- bzw. Recovery-Protokolle

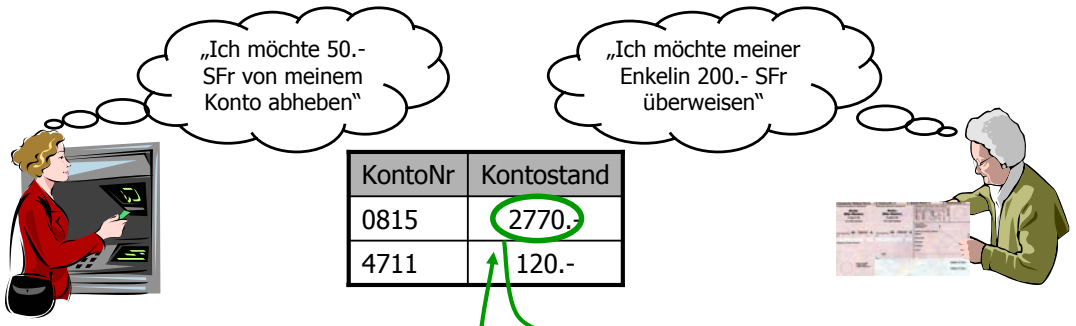
1.3 Transaktionsverwaltung – Beispiele

- Beispiel 1: Banküberweisung
  - Klassischer Vertreter von Datenbanktransaktionen
    - Kurzlebige Transaktionen, operieren auf wenigen Datenobjekten
  - Parallele Zugriffe auf Konto-Relationen (bzw. Objekte)
    `Konto (KontoNr, KundenNr, FilialNr, Kontostand)`
  - Typische Operationen
    `Debit (KontoNr, Betrag)` bucht **`Betrag`** von Konto **`KontoNr`** ab
    `Credit (KontoNr, Betrag)` bucht **`Betrag`** auf Konto **`KontoNr`**
    sowie `Transfer (KontoNrA, KontoNrB, Betrag)`, das aus einem **`Debit(KontoNrA, Betrag)`** und einem **`Credit(KontoNrB, Betrag)`** besteht.
- Beispiel 2: Geschäftsprozesse (Workflow Management)
  - Transaktionskonzepte (in erweiterter Form) für transaktionale Prozesse, ausserhalb von Datenbanksystemen
    - Prozesse sind äusserst langlebig, greifen auf autonome, verteilte und evtl. auch heterogene Ressourcen zu
  - Operationen (Aktivitäten) = Aufrufe von Anwendungsdiensten
  - Komplexe Struktur durch flexiblere Fehlerbehandlung (Alternativen)

Beispiel 1 – Banküberweisung



Beispiel 1 – Banküberweisung: Atomarität



KontoNr	Kontostand
0815	2770.-
4711	120.-

SELECT kontostand INTO :balanceTransA
FROM Konto WHERE kontoNr = 0815;

2570.-

balanceTransA := balanceTransA - 200;

Update Konto
SET kontostand = :balanceTransA
Where kontoNr = 0815;

SELECT kontostand INTO :balanceTransB
FROM Konto WHERE kontoNr = 4711;

KontoNr	Kontostand
0815	2570.-
4711	120.-

*** Programmabbruch ***

Beispiel 1 – Banküberweisung: Atomarität

- Im vorigen Ablauf ist die Alles-oder-Nichts-Semantik der Atomarität verletzt
  - Ein inkonsistenter Zwischenzustand bleibt erhalten
    (bei der fehlgeschlagenen Überweisung ging Geld verloren!)
  - Abhilfe: für die durch den Systemabsturz abgebrochene Überweisungstransaktion sollte das bereits erfolgte Debit automatisch durch das Datenbanksystem rückgängig gemacht werden
  - ➲ Gefragt sind also geeignete Recovery-Protokolle, die beispielsweise den Zustand einspielen, der direkt vor Beginn der Überweisungstransaktion vorlag

Beispiel 1 – Banküberweisung: Isolation

„Ich möchte 50.- SFr von meinem Konto abheben"

„Ich möchte meiner Enkelin 200.- SFr überweisen"

KontoNr	Kontostand
0815	2770.-
4711	120.-

SELECT Kontostand INTO :balanceATM
FROM Konto WHERE NontoNr = 4711;

SELECT Kontostand INTO :balanceTransB
FROM Konto WHERE KontoNr = 4711;

320.-

balanceTransB := balanceTransB + 200;

balanceATM := balanceATM – 50;

70.-

Update Konto
SET Kontostand = :balanceTransB
WHERE KontoNr = 4711;

Update Konto
SET Kontostand = :balanceATM
WHERE KontoNr = 4711;

KontoNr	Kontostand
0815	2570.-
4711	70.-

Beispiel 1 – Banküberweisung: Isolation

- Im vorigen Ablauf überschneiden sich die beiden Transaktionen in inkorrekter Weise
  - Die Änderungsoperation der Überweisungstransaktion geht komplett verloren (und damit auch das überwiesene Geld)
  - Der korrekte Endzustand nach kompletter Ausführung beider Transaktionen wäre:

KontoNr	Kontostand
0815	2570.-
4711	270.-

➲ Das System muss daher die beiden Transaktionen durch geeignete Concurrency Control-Protokolle voneinander isolieren, indem z.B. die Überweisungstransaktion den Kontostand von 4711 erst lesen und verändern kann, nachdem die ATM-Transaktion beendet ist.

Beispiel 1 – Banküberweisung

Weitere Anforderungen der Beispielanwendung

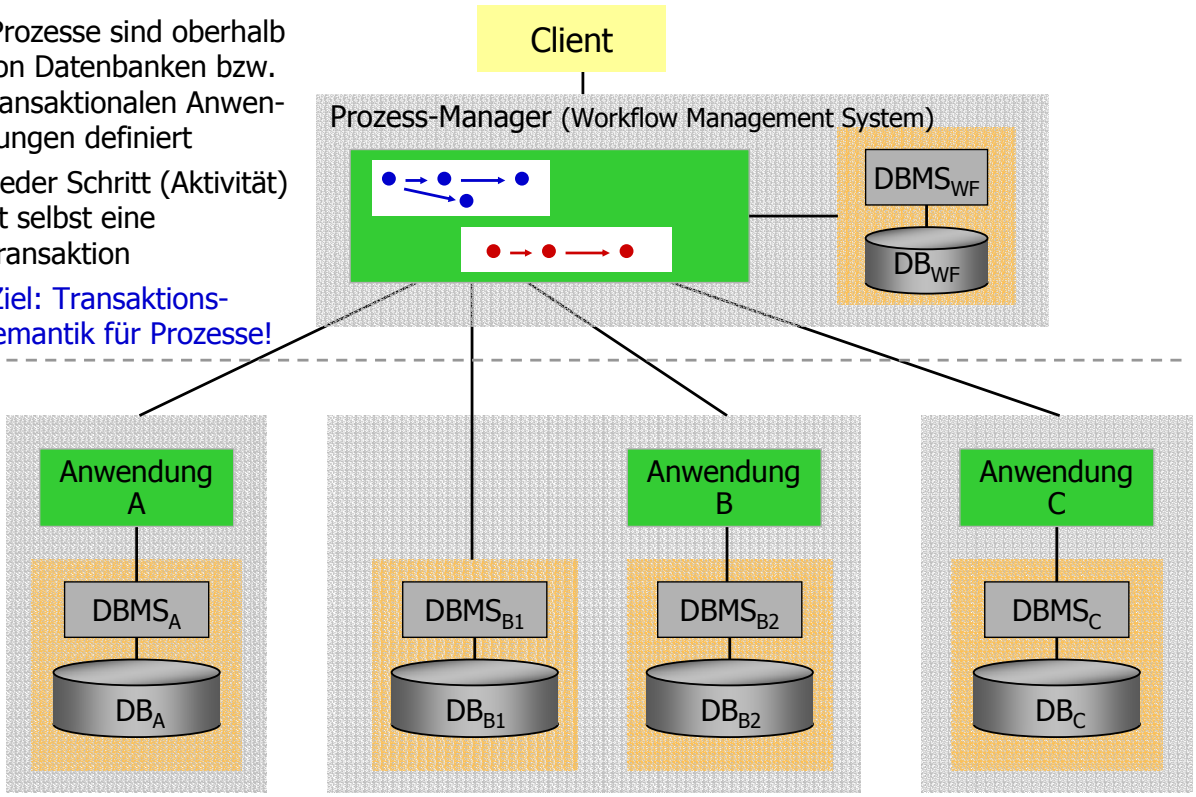
- Konsistenz
  - z.B. Einschränkung, dass der Kontostand niemals negativ werden kann
  - Dies wird in der Regel durch Integritätsbedingungen zugesichert

```
CREATE ASSERTION positiverKontostand
CHECK (NOT EXISTS (
       SELECT * FROM Konto
       WHERE Kontostand < 0 ))
```

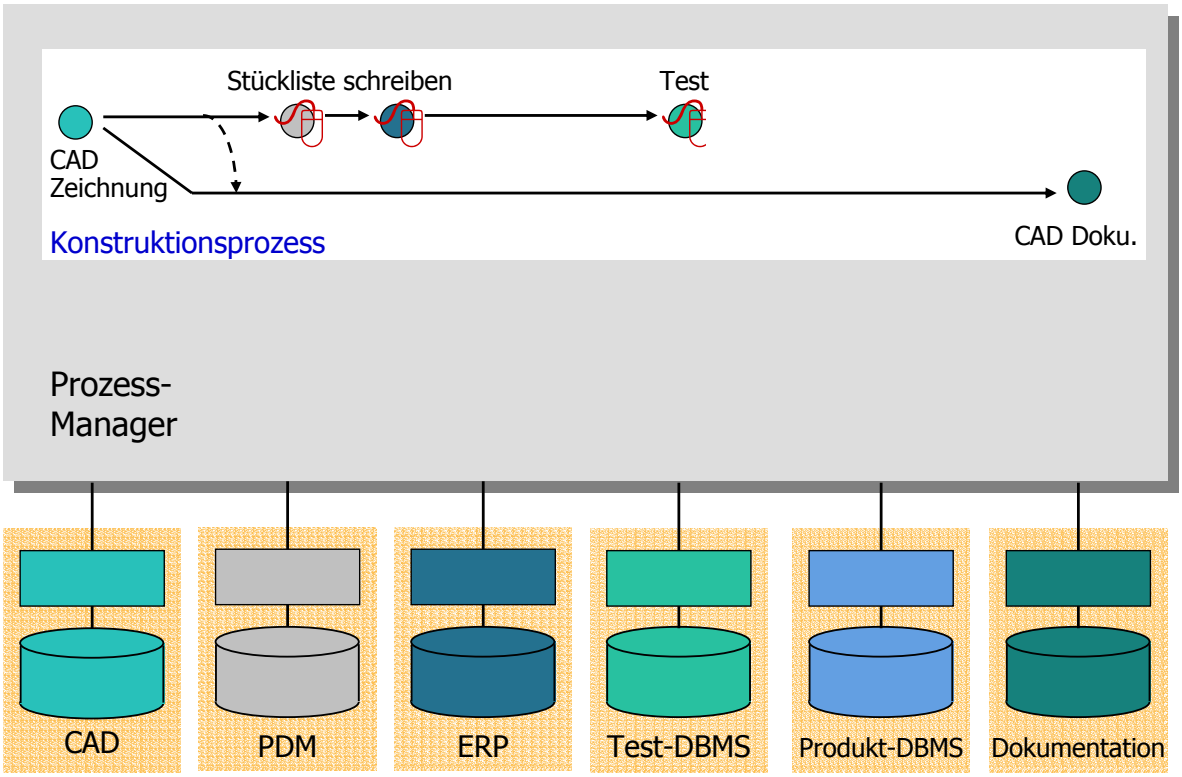
  - ➲ Auch die Konsistenz wird vom System garantiert und muss nicht vom Anwendungsentwickler berücksichtigt werden
- Dauerhaftigkeit
  - Änderungen am Kontostand durch korrekt (mit Commit) beendete Transaktionen müssen dauerhaft sein, also z.B. auch Systemabstürze überstehen
- Performanz
  - Natürlich muss das System mehr als zwei Benutzer parallel unterstützen, ohne dass dabei die Synchronisation konkurrierender Transaktionen zu signifikanten Performance-Einbussen führt

Beispiel 2 – Geschäftsprozesse

- Prozesse sind oberhalb von Datenbanken bzw. transaktionalen Anwendungen definiert
- Jeder Schritt (Aktivität) ist selbst eine Transaktion
- Ziel: Transaktionssemantik für Prozesse!



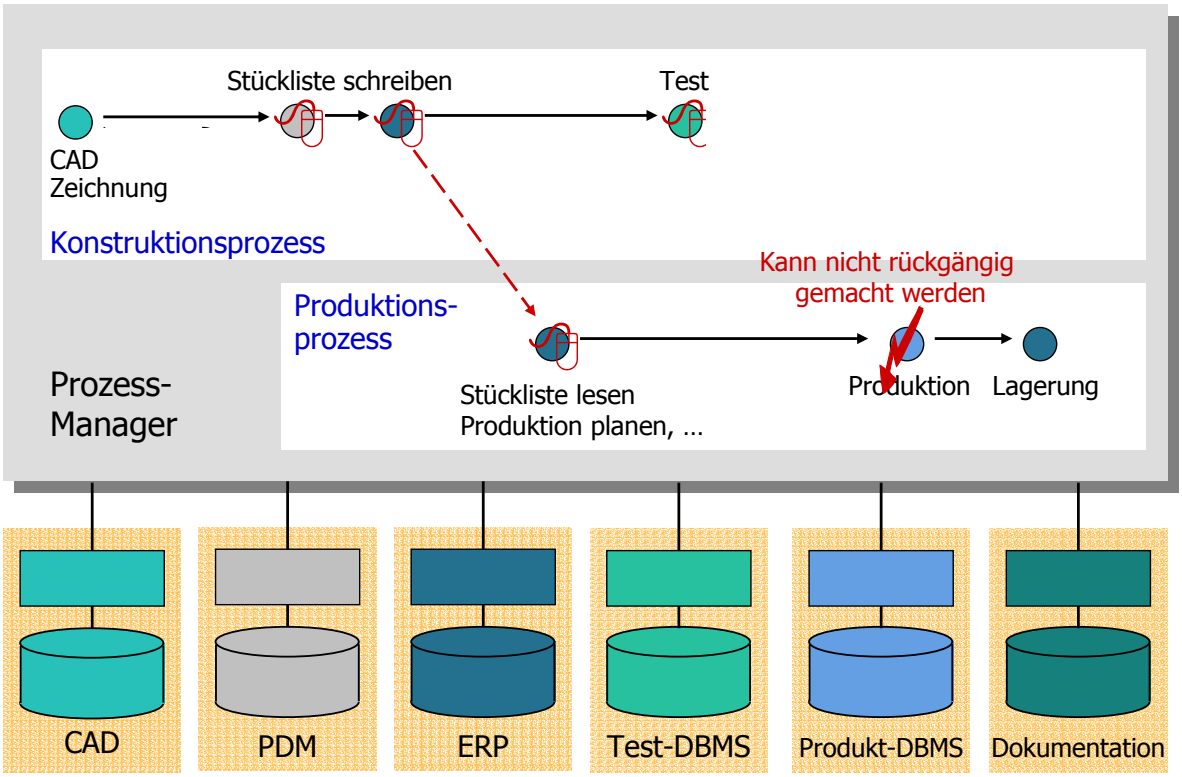
Beispiel 2 – Geschäftsprozesse: Atomarität



Beispiel 2 – Geschäftsprozesse: Atomarität

- Im Bezug auf Atomarität für kurze (Buchungs-)Transaktionen war die Alles-oder-Nichts-Semantik sehr erstrebenswert
- Für Prozesse ist dies jedoch nicht mehr der Fall
  - Beim Auftreten eines Fehlers sollten nicht automatisch die gesamten Änderungen, d.h. alle bisher ausgeführten Dienste, eines Prozesses verworfen werden (Kosten optimieren!)
  - Manchmal kann dies auch gar nicht erlaubt sein durch die Semantik der einzelnen Prozess-Schritte (wenn zum Beispiel die Effekte einer Aktivität nicht rückgängig gemacht werden können, da kein zusätzlicher Dienst hierzu verfügbar ist)
- Prozesse benötigen flexiblere Fehlerbehandlungsstrategien
  - z.B. alternative Ausführungen, die im Fehlerfall zur Anwendung kommen
  - Bei Systemfehlern sollte der Prozess weitergeführt werden als hätte es keinen Fehler gegeben (Vorwärts-Recovery)

Beispiel 2 – Geschäftsprozesse: Isolation



Beispiel 2 – Geschäftsprozesse: Isolation

- Parallelität zwischen Prozessen ist sehr wichtig
- Aber: Datenbanktransaktionen können jederzeit abgebrochen werden, sowohl bei Anwendungsfehlern als auch bei Systemfehlern, bei Prozessen ist dies nicht immer der Fall
- Dies hat auch Konsequenzen für die Concurrency Control, da Fälle auftreten können, die sich durch Abbruch nicht mehr beheben lassen
  - Concurrency Control muss zusätzliche Randbedingungen berücksichtigen
  - Gleichzeitig darf aber der Parallelitätsgrad nicht zu stark eingeschränkt werden (Prozesse können sehr langdauernd sein!)
  - ➲ Abhilfe: Berücksichtigung von Ausführungskosten (sowohl für normale Prozess-Schritte als auch für deren Kompensation) um zu entscheiden, wie restriktiv der Prozess-Manager agieren sollte

Beispiel 2 – Geschäftsprozesse

Weitere Anforderungen der Beispielanwendung

- Dauerhaftigkeit
  - Der Prozess-Manager verwaltet keine eigenen Daten (diese liegen alle in den unterliegenden transaktionalen Anwendungen; dort sorgen lokale Transaktionen für die ACID-Eigenschaften)
  - Aber: um Vorwärts-Recovery zu unterstützen muss immer der aktuelle Zustand eines Prozesses persistent verwaltet werden
  - ➲ Datenbank des Prozess-Managers für Prozess-Zustände

1.4 Read/Write-Modell

- Die wesentlichen Aspekte von Concurrency Control und Recovery werden wir zunächst anhand des klassischen Read/Write-Modells von Datenbanken einführen
  - Später (vor allem in Teil V) wird dieses Modell dann erweitert

Grundlegende Begriffe

Datenbank (DB) Menge von Objekten: DB = $\{o_1, o_2, \ldots\}$
Diese Menge ist statisch, Objekte sind in der Regel einzelne Datenseiten

Operation (OP) Menge von Operationen: OP = {read, write}

Aktion Aktion $a \in (OP \times DB)$, also entweder read(o_i) oder write(o_i).
Dies wird später Gegenstand der Verallgemeinerung werden

read(o_i) liefert das Datenobjekt o_i zurück
write(o_i) schreibt Datenobjekt o_i

Im Folgenden werden wir die Aktionen in abgekürzter Schreibweise verwenden:
- r(o_i) für read(o_i)
- w(o_i) für write(o_i)

Transaktion

Transaktion (T) Eine Transaktion T ist ein Tupel mit $T = (ACT , \ll)$ bestehend aus einer Menge von Aktionen ACT, zwischen denen eine (partielle) Ordnungsrelation « vorhanden ist

Dabei ist: $ACT = \{ a_1, a_2, \ldots, a_k \} \cup term$

$a_1, \ldots, a_k$ sind Aktionen

$term \in \{C, A\}$ ist Terminierungsaktion

$\ll \subseteq (ACT \times ACT)$ ist Präzedenzrelation

- Jede Transaktion T ist entweder durch C (commit) oder A (abort) abgeschlossen. C bzw. A ist hinter allen Aktionen der jeweiligen Transaktion geordnet (bezüglich «), also $a_i \ll term$ für alle $1 \leq i \leq k$.
- Die Präzedenzrelation « legt die Ausführungsreihenfolge der Aktionen einer Transaktion fest
  - « partiell: Parallelität innerhalb einer Transaktion (intra-transaktionale Parallelität) erlaubt
  - « total: sequentieller Ablauf (dies ist der Normalfall)

  $T = \langle a_1 \ll a_2 \ll \ldots \ll a_k \ll C \rangle$

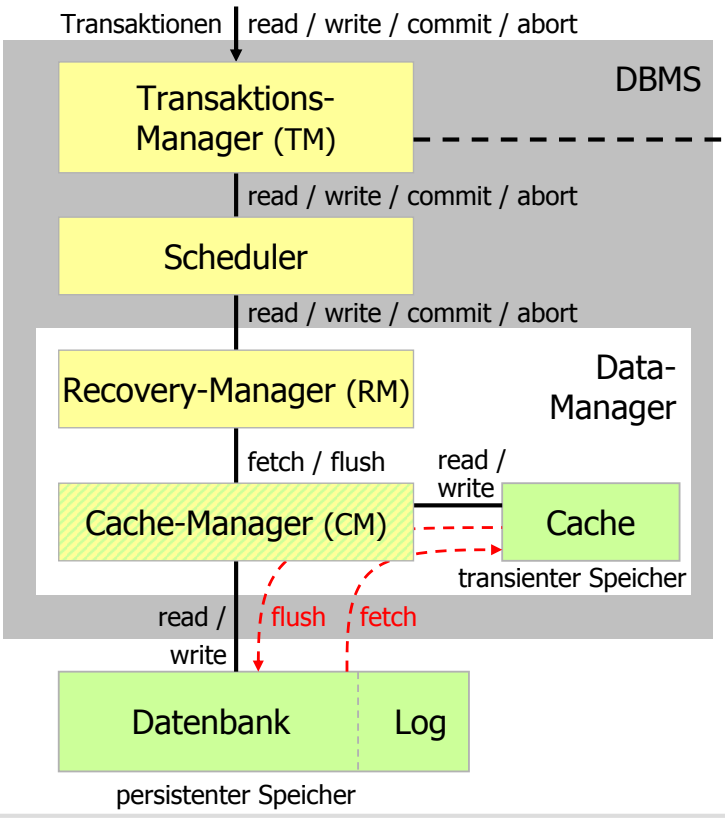
DB-System-Modell und Komponenten ...

Der **TM** leitet eingehende Operationen an den Scheduler weiter. Im Falle von verteilten DBMSen beinhaltet dies die Kommunikation sowie die Koordination verteilter Transaktionen.

Der **Scheduler** entscheidet, ob Operationen ausgeführt, zurückgewiesen oder verzögert werden.

Aufgabe des **RM** ist, die DB vor Systemfehlern zu schützen (z.B. wenn der transiente Speicher verloren geht) und zu garantieren, dass die Effekte abgeschlossener Transaktionen dauerhaft sind bzw. dass abgebrochene TA keine Effekte hinterlassen.

Der **CM** sorgt für einen effizienten Zugriff auf Daten.



... DB-System-Modell und Komponenten

- Module
  - In der Praxis sind die einzelnen Module stark voneinander abhängig, deren Implementierung also zumeist ineinander verzahnt (speziell um Performance-Gewinne zu erzielen)
  - Trennung ist also eher konzeptioneller (didaktischer) Natur
  - Weitergehende Aspekte sind in diesem abstrakten Modell komplett ausgeblendet (wie z.B. die Anfrageoptimierung, die Abbildung einer SQL-Schnittstelle auf read/write-Operationen, etc.)
- Handshaking
  - Ordnungen auf Aktionen müssen zwischen den einzelnen Modulen korrekt weitergegeben werden
  - Dies geschieht in diesem Modell strikt seriell mit einem Handshaking-Verfahren, d.h. im Falle von a « b wird b erst an die nächste Komponente gegeben, nachdem a ein Ergebnis zurückgeliefert hat
- Anwendbarkeit des Modells
  - Das System-Modell ist zwar auf die Read/Write-Semantik von Transaktionen zugeschnitten (read, write, flush, fetch), die Hauptkomponenten und deren Interaktionen lassen sich aber sehr gut verallgemeinern
  - Im Fall der transaktionalen Prozesse sind beispielsweise TM, Scheduler, und RM wichtige Komponenten des Prozess-Managers