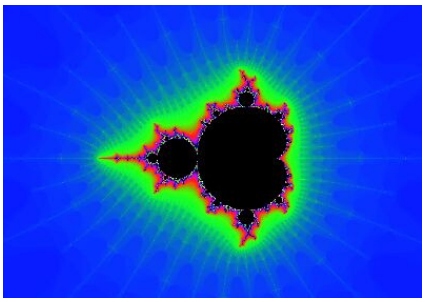
Datenbanken

11. Transaktionen und Konkurrenzsteuerung

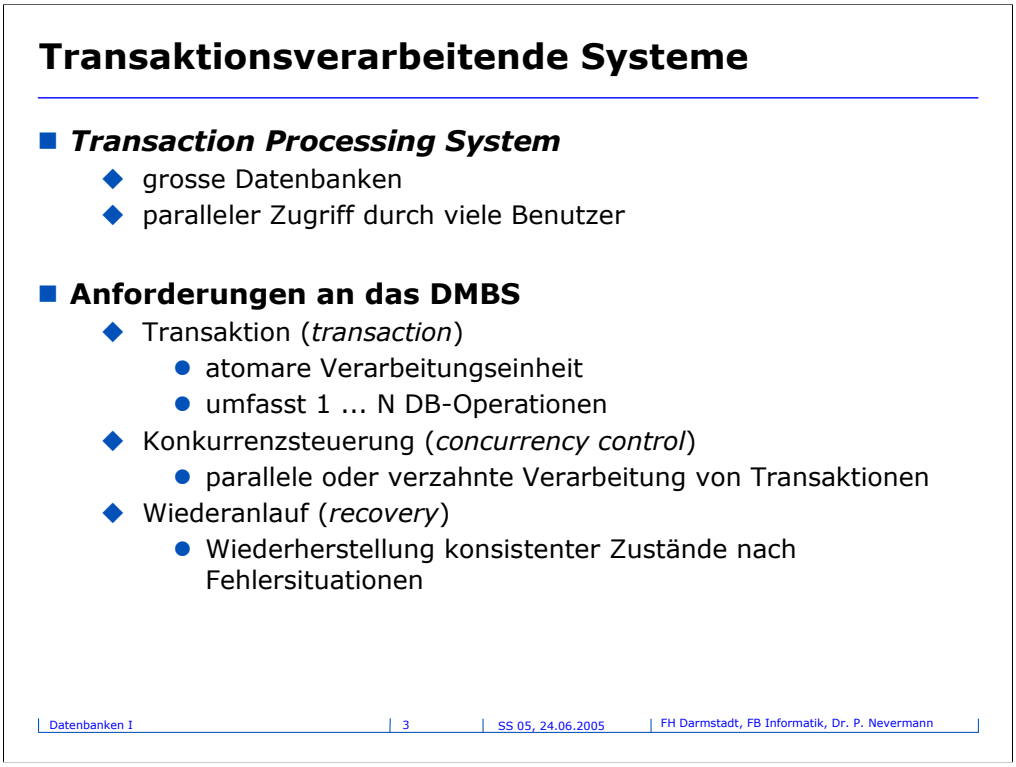


Fachhochschule Darmstadt
Fachbereich Informatik

Dr. Peter Nevermann

Rückblick

- **PHP**
  - in HTML mit **<?php** und **?>** eingebunden
  - Datei-Endung: **.php**
  - Einfach zu lernen (simpel, ähnlich zu Java/C/...)
  - Umfangreiche Funktionen-Bibliothek (http://www.php.net/docs.php)
  - Variablen sind *typfrei* und beginnen mit **$**
  - Arrays:
    - numerisch ($a[0] = "red") o. assoziativ ($b["root"] = "red")
    - **foreach** ($a as **$value**) {...} oder **foraech** ($b as **$key=>$value**) {...}
  - HTML Formular-Daten: **$_GET**, **$_POST**, **$_REQUEST**
  - MySQL Funktionen
    - **mysql_connect**, **mysql_close**
    - **mysql_query**, **mysql_fetch_row**, **mysql_fetch_assoc**



Transaction Processing System

Datenbanken sind zentraler Bestandteil von transaktionsverarbeitenden Systemen (*transaction processing system*), wie z.B. Reservierungssysteme, Konten-und Depot-Verwaltung bei Banken, etc.

Typisch bei solchen Systemen ist:

•Die Verarbeitung erfolgt in Einheiten, die grösser sind als eine einzelne SELECT-, INSERT-, UPDATE- oder DELETE -Operation. Diese Einheiten nennt man **Transaktionen** (*transactions*) und können eine Vielzahl von DB-Operationen auf mehreren Tabellen umfassen (z.B. Flug reservieren, Kontobewegung, etc.). Transaktionen sind atomar auszuführen (d.h. ganz oder garnicht) und müssen eine Datenbank von einem konsistenten Zustand ein anderern konsistenten Zustand überführen.

•Es werden viele Transaktionen parallel oder verzahnt verarbeitet. Dabei werden hohe Verfügbarkeit und schelle Antwortzeiten erwartet. Dazu ist eine geeignete **Konkurrenzsteuerung** (*concurrency control*) erforderlich, die sicherstellt, dass sich die einzelnen Transaktionen nicht gegenseitig behindern wodurch es zu inkonsistenten DB-Zuständen kommen könnte.

•Es ist in hohem Grade *Ausfallsicherheit* und *garantierte Datenkonsistenz* gefordert. Dazu sind geeignete **Wiederanlauf-Mechanismen** (*recovery*) nötig, die nach Fehlersituationen konsistente DB-Zustände wiederherstellen können.

Innerhalb eines DBMS's werden diese Aufgaben von verschiedenen Komponenten übernommen:

•Transaktions-Manager

•Konkurrenzsteuerungs-Subsystem

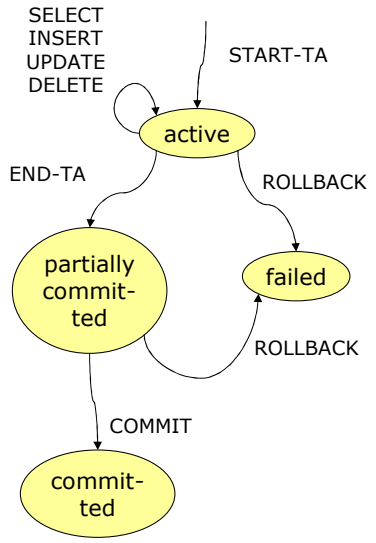
•Wiederanlauf-Subsystem (Recovery)

Parallele und verzahnte Verarbeitung

Für die parallele Verarbeitung von Transaktionen können mehrere CPUs zur Verfügung stehen (echt parallele Verarbeitung, *parallel processing*) oder aber es erfolgt eine "verzahnte" Verarbeitung (*interleaved processing*) auf einer einzigen CPU.

Transaktionen

- **Transaktions-Operationen**
  - START-TRANSACTION
  - END-TRANSACTION
  - COMMIT
  - ROLLBACK
- **ACID-Eigenschaften**
  - *Atomicity* (atomar)
  - *Consistency preserving* (konsistenzerhaltend)
  - *Isolation* (isoliert)
  - *Durability* (Dauerhaftigkeit)



Datenbanken I | 5 | SS 05, 24.06.2005 | FH Darmstadt, FB Informatik, Dr. P. Nevermann

Transaktions-Operationen

Mit dem Transaktionskonzept werden zusätliche Datenbankoperationen eingeführt:

•START-TRANSACTION – um den Anfang einer Transaktion zu markieren.

•END-TRANSACTION – um das Ende einer Transaktion zu markieren.

•COMMIT – zur Festlegung des durch die Transaktion erzeugten neuen Zustandes.

•ROLLBACK – um alle durch die Transaktion durchgeführten Änderungen zu verwerfen und zum alten Zustand (vor Transaktionsbegin) zurückzukehren.

Transaktions-Zustände

Transaktionen durchlaufen eine Vielzahl von Zuständen (siehe Diagramm auf der Folie):

•*active* (aktiv) – ist eine Transaktion nachdem sie begonnen wurde und während in ihr Operationen ausgeführt werden.

•*failed* (fehlgeschlagen) – dieser Zustand wird nach Auftreten einer Fehlersituation erreicht, die innerhalb oder ausserhalb der Anwendung festgestellt worden sein kann – die Änderungen durch die Transaktion werden zurückgesetzt und der Zustand vor Transaktionsbegin wird wiederhergestellt.

•*partially commited* (teilweise festgelegt) – dieser Zustand wird erreicht wenn die Transaktion von der Anwendung beendet wurde und dessen Festlegung bevorsteht. Die Konkurrenzsteuerung des DBMS prüft nun die Korrektheitskriterien für den COMMIT.

•*committed* (festgelegt) – die Änderungen durch die Transaktion wurden dauerhaft festgelegt.

ACID-Eigenschaften von Transaktionen

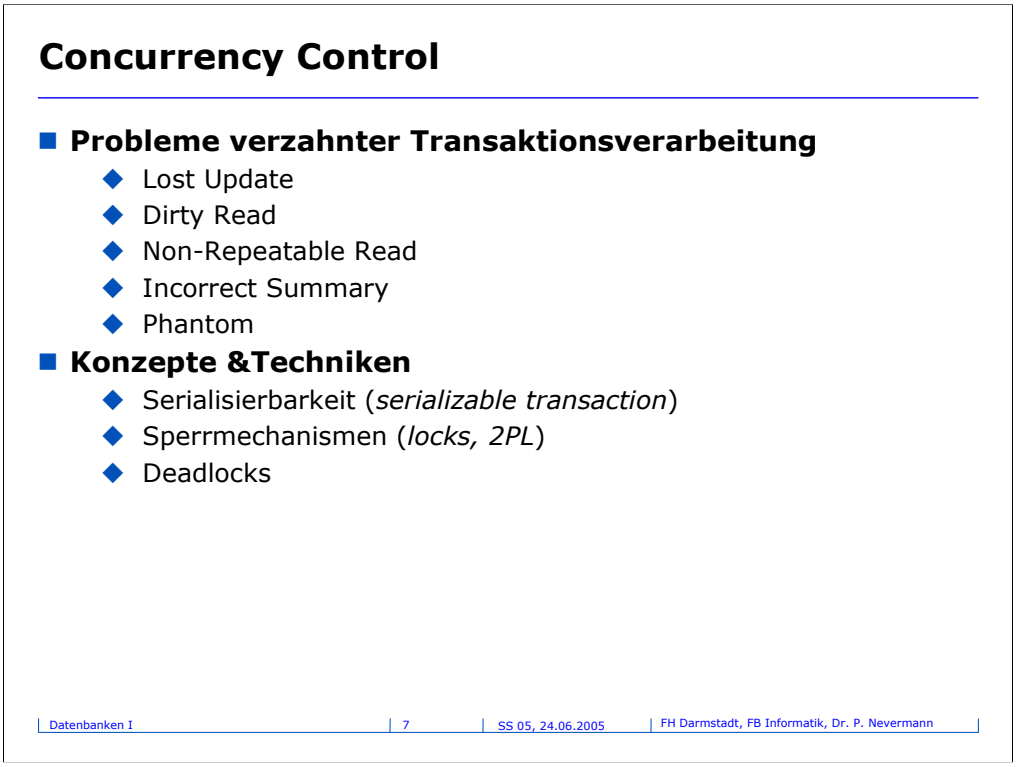
Vom Transaktionssystem eines DBMS sind bestimmte Eigenschaften gefordert:

•**A**tomicity
Eine Transaktion ist eine atomare Verarbeitungseinheit, d.h. es werden entweder alle Operationen oder keine Operation der Transaktion ausgeführt. Innerhalb eines DBMS ist hierfür der Transaktionsmanager verantwortlich.

•**C**onsistency preserving
Eine Transaktion führt eine DB von einem konsistenten Zustand in einen anderen konsistenten Zustand über. Verantwortlich für die Konsistenzerhaltung sind gemeinsam das Anwendungssystem und das subsystem zur Überwachung von Integrtätsregeln innerhalb des DBMSs.

•**I**solation
Eine Transaktion kann isoliert arbeiten und kommt sich nicht mit parallel oder verzahnt arbeitenden Transaktionen ins Gehege. Hierfür ist im DBMS das Konkurrenzsteuerungs-Subsystem zuständig, welches i.d.R. mit Sperrmechanismen arbeitet.

•**D**urability
Die Änderungen die nach Transaktionsende mit COMMIT in der DB festgesetzt wurden sind dauerhaft, auch wenn es danach zu Systemfehlern kommen sollte. Dafür sorgt im DBMS das Recovery-Subsystem (Wiederanlauf).



Konkurrenz-Probleme

Werden mehrere Transaktionen verzahnt ausgeführt, können gewaltige Konflikte auftreten, wenn diese Transaktionen auf denselben Ressourcen (Attribut, Tupel, Tabelle) operieren und nicht ausreichend voneinander isoliert sind.

Konkurrenzkontrolle erfolgt im DBMS nicht auf der BASIS komplexer SQL Operationen (SELECT, INSERT, UPDATE, DELETE), sondern auf der BASIS elementarer READ und WRITE Operationen. So setzt sich z.B. eine SQL SELECT Operation aus einer Vielzahl von elementaren READ Operationen, eine SQL UPDATE Operation aus mehreren elementaren READ und WRITE Operationen zusammen.

•Lost Update Problem

Die Änderungen durch eine der beteiligten Transaktion gehen verloren.

Beispiel:
Zwei Transaktionen (T1, T2) bearbeiten das selbe Tupel t aus der MITARBEITER Relation, T1 ändert das Gehalt, T2 die Adresse ... am Ende ist die Änderung des Gehalts verloren gegangen, weil T2 das Tupel zu früh gelesen hat.

```
T1   READ t
T2   READ t
T1   WRITE t (gehalt = gehalt * 1.03)
T2   WRITE t (adresse = 'Mozartweg 14, 65432 Daumdorf')
T1   COMMIT
T2   COMMIT
```

•Dirty Read Problem
Eine Transaktion liest einen nicht-festgelegten Zustand (*uncommitted state*), den eine andere Transaktion erzeugt hat aber später verwirft, und arbeitet damit weiter.

•Beispiel:
Zwei Transaktionen (T1, T2) bearbeiten dasselbe Tupel t aus der MITARBEITER Relation, T1 addiert einen Bonus von 500 auf das Gehalt, T2 liest das neue Gehalt erhöht nochmal um 3%, dann verwirft T1 seine Änderungen (ROLLBACK) ... davon bekommt T2 aber nichts mit und legt das berechnete Gehalt mit Bonus fest (COMMIT)

```
T1   READ t
T1   WRITE t (gehalt = gehalt + 5000)
T2   READ t
T2   WRITE t (gehalt = gehalt * 1.03)
T1   ROLLBACK
T2   COMMIT
```

•Non-Repeatable Read Problem
Im Verlauf einer Transaktion wird ein Wert mehrfach mit unterschiedlichem Ergebnis gelesen, weil andere Transaktionen in der Zwischenzeit den Wert mit COMMIT geändert haben.

Beispiel:
Zwei Transaktionen (T1, T2) bearbeiten dasselbe Tupel t aus der MITARBEITER Relation, T1 liest das Gehalt zwei Mal mit unterschiedlichem Ergebnis, weil T2 es in der Zwischenzeit geändert hat.

```
T1   READ t   (liest gehalt: 60000)
T2   READ t
T2   WRITE t (gehalt = gehalt + 3000)
T2   COMMIT
T1   READ t   (liest gehalt: 63000)
T1   ...
```

•Incorrect SummaryProblem
Eine Transaktion führt Aggregationsfunktionen aus, während andere Transaktionen die zugrundeliegenden Tupel (mit COMMIT) verändern.

•Phantom Problem
Im Verlauf einer Transaktion wird eine Selection (SELECT mit WHERE) mehrfach mit unterschiedlichem Ergebnis ausgeführt, weil andere Transaktionen in der Zwischenzeit Änderungen mit COMMIT vornehmen, welche das Ergebnis der WHERE-Auswerung beeinflussen.

Serialisierbarkeit

Um die beschriebenen Probleme bei verzahnten Transaktionen zu vermeiden, wird die Konkurrenzsteuerung des DBMS versuchen nur bestimmte Abläufe (*schedules*) für die Transaktionen zuzulassen.

Das übliche Korrektheitskriterium ist das der Serialisierbarkeit – *serializable schedule* (obwohl es auch andere gibt: *recoverable*, *cascadeless*, *strict*). Ein Ablauf verzahnter Transaktionen ist serialisierbar, wenn er äquivalent (= zu gleichen Ergebnissen führend) zu einem seriellen Ablauf ist, bei dem die Transaktionen nicht mehr verzahnt, sondern nacheinander ablaufen. Bei einem serialisierbaren Ablauf können die beschriebenen Probleme nicht auftreten.

Beispiel:

T1 READ-LOCK y
T1 READ y
T1 UNLOCK y
T2 READ-LOCK x
T2 READ x
T2 UNLOCK x
T2 WRITE-LOCK y
T2 READ y
T2 WRITE y (y.a = y.a + x.a)
T2 UNLOCK y
T1 WRITE-LOCK x
T1 READ x
T1 WRITE x (x.a = x.a + y.a)
T1 UNLOCK x

Hat man Anfangs die Werte x.a = y.a = 1, dann würde ein serieller Ablauf T1 → T2 das Ergebnis x.a = 2, y.a = 3 ergeben, ein serieller Ablauf T2 → T1 das Ergebnis x.a = 3, y.a = 2 ergeben, der gezeigte Ablauf ergibt aber x.a = 2, y.a = 2.

Dieses Problem kann man vermeiden, wenn sich alle Transaktionen an das 2-Phasen-Sperrprotokoll halten (*2-phase locking*, 2PL), nach dem alle LOCK-Operationen vor allen UNLOCK-Operatiön kommen müssen, d.h. die Transaktion hält i.d.R. alle Sperren bis ans Ende und gibt sie dann erst frei. Man kann nun zeigen, dass 2PL zu serialisierbaren Abläufen verzahnter Transaktionen führt. Der Preis den man für 2PL zahlt ist ein geringerer Durchsatz aufgrund der länger gehaltenen Sperren.

Sperren

Um serialisierbare Abläufe verzahnter Transaktionen zu gewährleisten, bedienen sich DBMS unterschiedlicher Mechanismen und Protokolle, wobei Sperrmechanismen verbreitet sind. Ressourcen (je nach Granularität: Tupel, Tabelle, DB oder auch physisch: Feld, Block, Datei) können mit Sperren (*locks*) versehen werden, um die parallel zugreifenden Transaktionen zu synchronisieren. Dabei sind zwei Arten von Sperren üblich: Lesesperre (*read-lock* oder *shared lock*) und Schreibsperre (*write lock* oder *exclusive lock*). Dazu kommen neue DB Operationen mit denen Sperren verwaltet werden können:

- *read-lock(X)* – die Ressource X wird mit einer Lesesperre versehen
- *write-lock(X)* – die Ressource X wird mit einer Schreibsperre versehen
- *unlock(X)* – die Ressource X wird entsperrt

Ist eine Ressource bereits mit einem Read-Lock belegt, so können andere Transaktionen zwar noch weitere Read-Locks aber keinen Write-Lock erwerben. Ist eine Ressource mit einem Write-Lock belegt (davon kann es dann nur einen geben!), dann kann keine weitere Transaktion Read- oder Write-Locks erwerben. Kann eine Transaktion die gewünschte Sperre (Read- oder Write-Lock) nicht erwerben, muss sie warten bis die Ressource wieder frei wird. Wird ein Write-Lock aufgehoben (UNLOCK), dann hat die Ressource keine Sperren mehr und wartende Transaktionen können notifiziert werden, daß die Ressource frei ist. Wird ein Read-Lock aufgehoben, und es handelt sich nicht um den letzten Read-Lock, so wird einfach nur der Read-Lock-Zähler der Ressource heruntergezählt. Wird das letzte Read-Lock entfernt, können wieder wartende Transaktionen notifiziert werden, dass die Ressource frei ist.

Frage: Warum gibt es keine SQL-Operationen (z.B. LOCK und UNLOCK) um Sperren zu erzeugen bzw. aufzuheben?

2-Phasen-Sperrprotokoll (2PL)

Man kann sich mit einfachen Beispielen überlegen, dass das alleinige Setzen und Aufheben von Sperren noch keine Serialisierbarkeit von Abläufen garantiert, und zwar, wenn Sperren innerhalb der Transaktion zu früh freigegeben werden:

Deadlocks

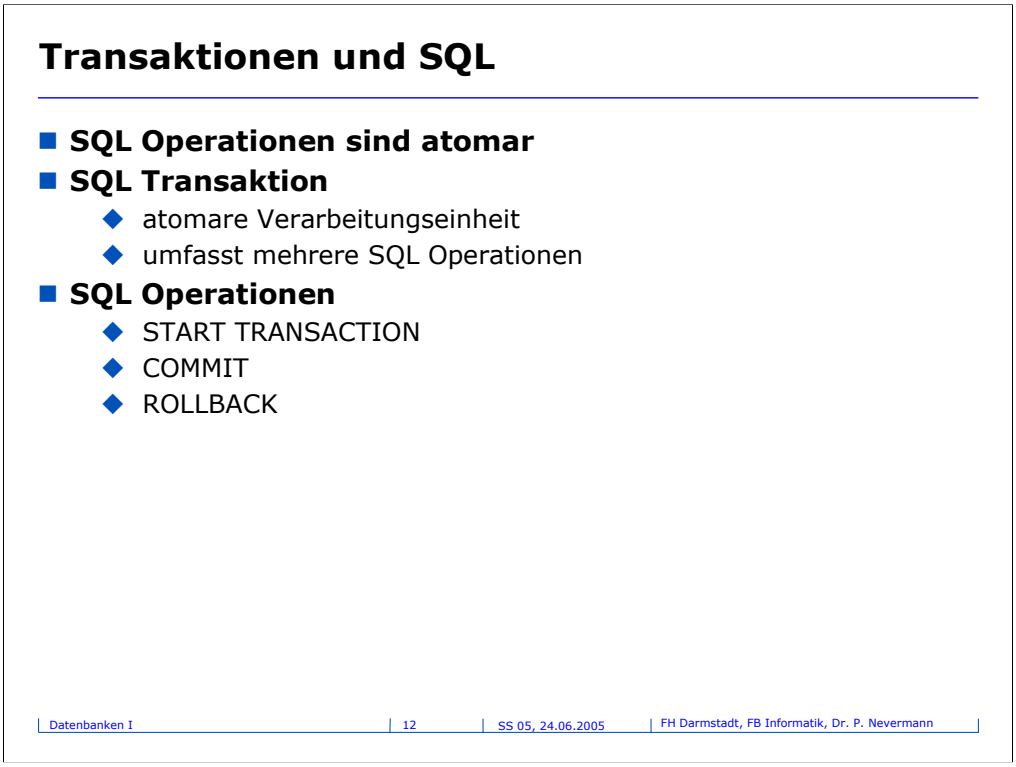
Auch mit 2PL kann es noch zu Problemen mit verzahnten Transaktionsabläufen kommen. Das bekannteste ist das der "Verklemmung" (*Deadlock*), bei dem Transaktionen gegenseitig darauf warten, dass Sperren freigegeben werden.

Beispiel:

T1 READ-LOCK y
T1 READ y
T2 READ-LOCK x
T2 READ x
T1 WRITE-LOCK x ➔ wartet ewig auf x
T2 WRITE-LOCK y ➔ wartet ewig auf y

Timeout

Der einfachste Mechanismus um Deadlocks zu vermeiden ist die Verwendung von Zeitabschaltungen bei Wartezuständen (*timeout*). Dabei gibt eine Transaktion – unabhängig davon, ob ein Deadlock besteht oder nicht – nach einer definierten Zeitspanne des Wartens einfach auf und setzt die Transaktion zurück (ROLLBACK). Die Anwendung muss dann einen neuen Versuch starten.



SQL Transaktionen

SQL Operationen sind zwar nicht elementar (setzen sich aus einer Vielzahl von READ/WRITE Operationen zusammen), werden aber durch das DBMS atomar ausgeführt. Man kann aber auch mit SQL noch grössere Verarbeitungseinheiten (*SQL Transaktionen*) erzeugen, die eine Vielzahl von SQL Operationen umfassen.

SQL-Operationen zur Transaktionsverarbeitung

SQL Transaktionen starten i.d.R. implizit mit Zugriffsoperationen (SELECT, INSERT , UPDATE, DELETE), und werden mit COMMIT (Festlegung) oder ROLLBACK (Abbruch und Zurücksetzung) beendet.

Standardmässig ist aufgrund des verwendeten Sprerr-Protokolls Serialisierbarkeit von Transaktionsabläufen gegeben. Es kann aber auch ein niedrigerer Isolierungsgrad (*isolation level*) für Transaktionen mit dem SQL-Befehl SET TRANSACTION eingestellt werden.

Isolation Level

Folgende Isolation-Levels sind definiert:

•READ UNCOMMITTED
Dirty-Read-, Non-Repeatable-Read- und Phantom-Probleme können auftreten

•READ COMMITTED
Non-Repeatable-Read- und Phantom-Probleme können auftreten

•REPEATABLE READ
Phantom-Probleme können auftreten

•SERIALIZABLE
Keines der oben genannten Probleme kann auftreten

Transaktionen in MySQL

Standardmässig werden SQL-Operationen in MySQL im sogenannten "Auto-Commit-Modus" verarbeitet, d.h. jede Operation bildet für sich eine Transaktion die MySQL implizit mit COMMIT abzuschliessen versucht.

In MySQL kann der Auto-Commit-Modus auch ausgeschaltet werden um Transaktionen mit mehreren Operationen zu verarbeiten. Voraussetzung ist die Verwendung von Transaktionssicheren Tabellen (z.B. durch ENGINE=InnoDB oder ENGINE=BDB).

Man hat in MySQL folgende zwei Möglichkeiten für explizite Transaktionen:

•SET AUTOCOMMIT=0;
Schaltet den Auto-Commit-Modus aus. Änderungen von Operationen werden ab jetzt nur dann dauerhaft, wenn sie mit COMMIT abgeschlossen werden. Mit ROLLBACK kann eine Transaktion zurückgesetzt werden. Nach COMMIT oder ROLLBACK beginnt implizit eine neue Transaktion.

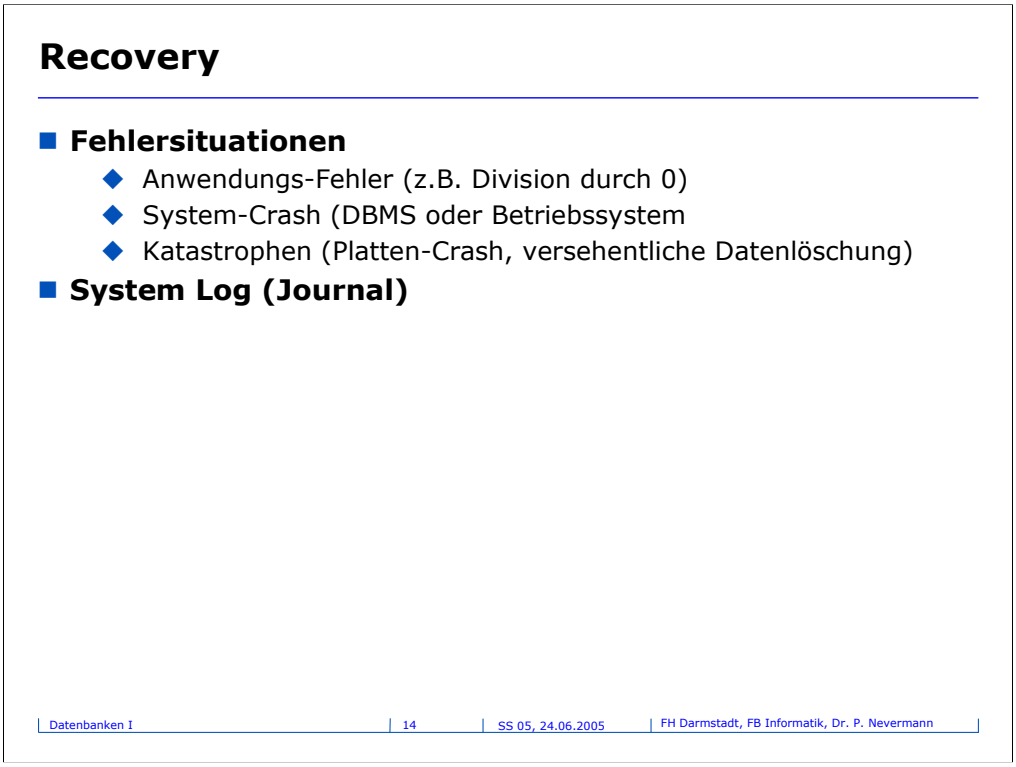
•START TRANSACTION
Startet eine ad-hoc eine Transaktion innerhalb der kein Auto-Commit stattfindet (auch wenn Auto-Commit generell aktiviert ist). Die Transaktion muss mit COMMIT oder ROLLBACK beendet werden. Danach kann wieder AUTOCOMMIT greifen, falls es aktiviert ist.

Der Isolation-Level von Transaktionen kann mit folgenden SQL Befehl gesetzt werden (Standardwert ist SERIALIZABLE):

•SET TRANSACTION [ISOLATION LEVEL {READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE}]

Transaktionen mit Access

Auch in Access kann man explizite Transaktionen mit dem SQL Befehl BEGIN TRANSACTION starten. Diese werden dann mit COMMIT oder ROLLBACK beendet.



Das System muss in der Lage sein, nach Fehlersituationen einen konsistenten und möglichst aktuellen DB-Zustand wiederherzustellen.

Fehlersituationen

Fehlersituation sind dabei vielfältig:

•Eine Transaktion schlägt aufgrung eines Anwendungsfehlers fehl (z.B. Programmierfehler): Wird möglicherweise durch den ROLLBACK-Mechanismus des Transaktionsmanagers behandelt

•System-Absturz (DBMS oder Betriebssystem):
Das Recovery Subsystem sollte genügend Informationen vorhalten (System-Log, Journal), um alle Transaktionen die COMMIT erreicht hatten (aber dessen Änderungen noch nicht vollständig persistent gemacht worden waren) zu wiederholen (REDO) ... bzw. Transaktionen die noch nicht ihr Ende erreicht hatten aber teilweise schon Änderungen geschreiben hatten zurückzusetzen (UNDO).

•Platten-Crash, versehenliche Datenlöschung (und andere Katastrophen):
Hier ist das Recovery-System sicherlich auf vorhandene Datensicherungen (BACKUP) angewiesen, um zunächst einen früheren konsistenten Zustand wiederherzustellen. Falls vorhanden, kann anschliessend das DB-Journal für REDO von post-update Transaktionen herangezogen werden. Bei einem Platten-Crash geht aber das Journal mit hoher Wahrscheinlichkeit auch verloren, so dass der Zustand des letzten Backups die bestmögliche Wiederherstellung darstellt.

System-Log (Journal)

Im System-Log (oder Journal) protokolliert das Recovery-Subsystem des DBMS alle Operationen innerhalb von Transaktionen. Dabei kommt das DBMS mit relativ wenig Informationen aus, z.B. <operation, transaktions-nr, ressource, alter-wert, neuer-wert>, um mit REDO bzw. UNDO Transaktionen wiederholen oder zurückzufahren zu können.

Die Protokollierung im System-Log erfolgt auf der Basis von elementaren Operationen (READ, WRITE) und nicht auf der Basis komplexer SQL Operationen.

Beispiel eines System-Logs:

START-TA, T1
START-TA, T2
WRITE-LOCK, T1, MITARBEITER
READ, T1,
WRITE, T1, MITARBEITER.gehalt, 60000, 60500
COMMIT, T1
ROLLBACK, T2
...