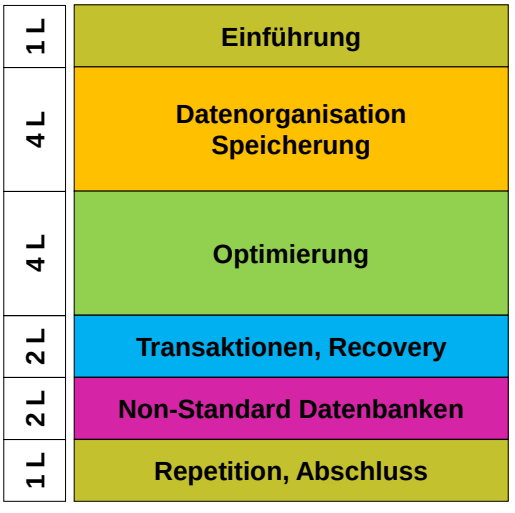
Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie

# Transaktionen

Lehrbuch Kapitel 9.1-9.3 (© bei den Autoren)

# Gliederung

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

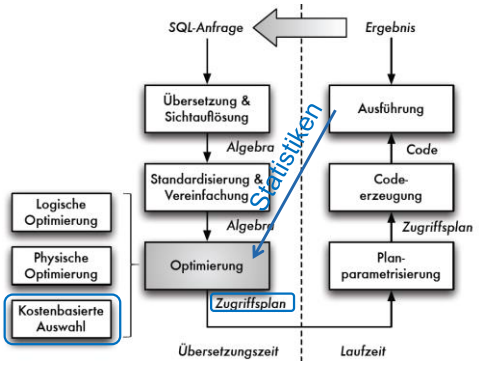



← "You are here"

2

# Rückblick

- Aspekte der Optimierung:
  - Ausführungspläne (Zugriffsplan/Anfrageplan), QEP («Query Execution Plan»)
  - Kostenbasierte Auswahl des Ausführungsplanes
  - Statistiken
  - Performanceaspekte in der Praxis



# Lernziele heute

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- Wichtigste Anforderungen an Transaktionsverarbeitung kennen
- ACID-Eigenschaften eines Transaktionssystems verstehen
- Probleme konkurrenter Transaktionen beschreiben können:
  - Lost Update
  - Dirty Read
  - Non-Repeatable Read
  - Phantom Read
- Aspekte von Nebenläufigkeit und Transaktionen in der Praxis kennen
- Begriffe Schedule und Scheduler kennen

4

## Transaktionsverarbeitung

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie

Wichtigste Anforderungen an Transaktionsverarbeitung:

1. Atomarität (Atomicity) / Unit of Work:
   Zusammengehörige Folge von Lese- und Schreibzugriffen (in sich geschlossene „Arbeitseinheit"), muss als Ganzes entweder erfolgreich abgeschlossen (Commit) oder rückgängig gemacht werden können (Rollback).
2. Consistency / Konsistenz: Alle Operationen hinterlassen die Datenbank in einem konsistenten Zustand.
3. Isolation / Nebenläufigkeit (Concurrency):
   „Gleichzeitiger" Zugriff mehrerer Benutzer ermöglichen, so dass diese Transaktionen keinen unerwünschten Einfluss aufeinander haben (bei nur einer CPU auch möglich -> Gleichzeitigkeit wird durch Zeitscheibenverfahren „simuliert").
4. Dauerhaftigkeit (Durability) / Recovery:
   Automatische Behandlung von Ausnahmesituationen (Fehlern) und schneller (möglichst automatischer) Wiederanlauf nach schwerwiegenden Fehlern. Wiederherstellung (Rollforward) verlorener Daten und Rücksetzten (Rollback) fehlerhafter Daten.

5

Der Begriff Transaktion (aus dem Buch):

> In der Datenbankforschung hat sich als Referenzdefinition für den Begriff der Transaktion die Definition nach Härder und Reuter etabliert:
>
> Eine Transaktion ist eine Folge von Operationen, die die Datenbank von einem konsistenten Zustand in einen konsistenten, eventuell veränderten, Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.

Die 4 ACID-Eigenschaften wurden schon in DAB1 eingeführt. Sie sind hier noch ein Mal zur Erinnerung kurz und vollständig aufgeführt:

> ACID-Prinzip: Welche Eigenschaften muss eine Datenbank haben um die Anforderungen an die Transaktionsverwaltung verlässlich zu erfüllen:
> - Atomicity – Atomarität: Zusammengehörige Operationen werden entweder ganz oder gar nicht ausgeführt.
> - Consistency – Konsistenzerhaltung: Alle Operationen hinterlassen die Datenbank in einem konsistenten Zustand (alle Integritätsbedingungen bleiben eingehalten).
> - Isolation – Isolation: Die Operationen nebenläufiger Transaktion laufen ohne gegenseitige Beeinflussung ab.
> - Durability – Dauerhaftigkeit: Alle Resultate bleiben nach Durchführung der Transaktion dauerhaft gespeichert.

In DAB2 werden wir Transaktionen, nebst grundsätzlichen Betrachtungen, vor allem in Bezug auf die 3 auf der Folie rot aufgeführten Gesichtspunkte genauer betrachten (Atomicity, Isolation und Durability) und zeigen, wie Datenbanksysteme diese Anforderungen technisch sicherstellen. Isolation wird vorwiegend im Zusammenhang mit Transaktionen behandelt, Atomarität und Dauerhaftigkeit vor allem beim Thema Recovery (folgt später) betrachtet.

## Nebenläufigkeit: Lost-Update-Problem

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie

Lost-Update: Überschreiben bereits getätigter Updates

| | Transaktion Benutzer 1 | Transaktion Benutzer 2 |
|---|---|---|
| 1 | SELECT Wert INTO W FROM Tbl | |
| 2 | | SELECT Wert INTO W FROM Tbl |
| 3 | UPDATE Tbl SET Wert = 100 | |
| 4 | | UPDATE Tbl SET Wert = 200 |
| 5 | SELECT Wert INTO W FROM Tbl | |
| 6 | | SELECT Wert INTO W FROM Tbl |

- Keine Isolation der Transaktionen beider Benutzer → Update-Operation von Benutzer 1 geht verloren!
- Lösung: Durch andere Transaktion gelesene Daten dürfen bis zur deren Beendigung nicht verändert werden.

6

Zunächst werden wir uns dem Thema der Nebenläufigkeit (Isolation) zuwenden. In der nächsten Lektion werden wird uns dann mit der Isolation (und Sperren) vertieft auseinander setzen.

In der Literatur werden bestimmte, typische Problemtypen, respektive Fehlertypen im Zusammenhang mit der Nebenläufigkeit aufgeführt. Wir betrachten die am häufigsten erwähnten Fehlertypen:

1. Lost-Update
2. Dirty-Read
3. Non-Repeatable-Read
4. Phantom-Read

Diese Klassifizierung ist auch für die Praxis relevant, da wir für Datenbanksysteme festlegen können, welche dieser Fehler wir zulassen möchten (über Isolationslevel festgelegt). Am einfachsten wäre es natürlich, keinerlei Fehler zuzulassen, dies hätte aber für die Performance negative Auswirkungen. Es ist daher wichtig zu verstehen, welche Folgen welcher Fehler hat und dann für eine bestimmte Applikation und/oder Transaktion festzulegen, welche Fehler zulässig sind. Wir werden daher zunächst diese vier Fehlertypen näher betrachten und dann zeigen, wie der «Fehler-Level» (Isolationslevel) im Datenbanksystem gewählt und gesetzt werden kann. Natürlich werden wir später auch untersuchen, wie ein Datenbanksystem diese Anforderungen technisch lösen kann (z.B. mittels Sperren).

Auf dieser Folie ist der erste Fehlertyp «Lost-Update» aufgeführt – dabei gehen geändert Daten verloren – dies ist in der Praxis fast immer völlig inakzeptabel.

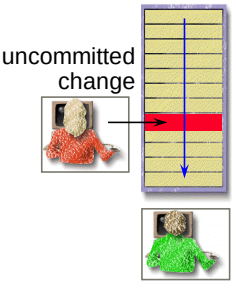
Auf den Folien fehlen übrigens die SQL-Befehle zum Setzen der Transaktionsgrenzen (BEGIN TRANSACTION und COMMIT TRANSACTION). Ohne Transaktionsgrenzen wird jeder DBMS-Befehl als eigene Transaktion ausgeführt und damit nur ganz oder gar nicht ausgeführt. Ohne das Setzen von Transaktionsgrenzen kann keiner der aufgeführten Problemtypen behandelt/gelöst werden. Die Wahl dieser Grenzen (wo beginnt und wo endet die Transaktion) muss daher immer bewusst erfolgen – auch in einem Anwendungsprogramm (ist z.B. der Befehl auf Zeile 5 noch Teil der Transaktion des Benutzer 1?)!

# Nebenläufigkeit: Dirty-Read-Problem

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie

Dirty-Read: Lesen von Veränderungen noch nicht abgeschlossener Transaktionen

| | Transaktion Benutzer 1 | Transaktion Benutzer 2 |
|---|---|---|
| 1 | SELECT Wert INTO W FROM Tbl | |
| 2 | UPDATE Tbl SET Wert = NeuerWert | |
| 3 | | SELECT Wert INTO W FROM Tbl |
| 4 | ROLLBACK | |
| 5 | | UPDATE Tbl SET Wert = W + 1 |
| 6 | | SELECT Wert INTO W FROM Tbl |
| 7 | SELECT Wert INTO W FROM Tbl | |




- Keine Isolation der Transaktionen beider Benutzer → Benutzer 2 arbeitet mit einem nicht bestätigten Wert, der später sogar zurückgenommen wird!
- Lösung: Nur Updates bestätigter Transaktionen lesen.

7

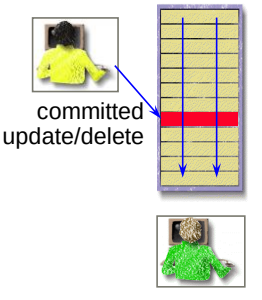
Dirty-Read ist in der Praxis zum Teil akzeptabel, die Konsequenzen müssen aber klar durchdacht werden.

# Nebenläufigkeit: Non-Repeatable-Read-Problem

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie

Non-Repeatable-Read: Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

| | Transaktion Benutzer 1 | Transaktion Benutzer 2 |
|---|---|---|
| 1 | SELECT Wert INTO W FROM Tbl | |
| 2 | | UPDATE Tbl SET Wert = Wert + 5 |
| 3 | | COMMIT |
| 4 | SELECT Wert INTO W FROM Tbl | |




2 mal gelesen

- Keine Isolation der Transaktionen beider Benutzer → Benutzer 1 erhält nicht immer denselben Wert!
- Lösung: Nur den Datenbankzustand sehen, der bei Beginn einer Transaktion vorliegt.

8

Während bei Lost-Update und Dirty-Read recht schnell klar ist, dass diese in der grossen Mehrzahl der Fälle unerwünscht sind, ist dies hier nicht mehr so klar. Es ist durchaus vorstellbar, dass dies da und dort akzeptabel ist.

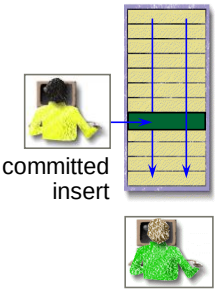
Wieso sollten wir dies zulassen und nicht einfach auf Nummer sicher gehen (den Isolationslevel möglichst hoch setzen)? Ein dazu konkurrenzierendes Ziel von uns ist, einen möglichsten hohen Grad an Parallelität im Datenbanksystem zuzulassen. Je höher wir den Isolationslevel setzen, desto geringer ist die zulässige Parallelität. Dieser Fehlertyp wird in der Praxis daher auch häufig zugelassen.

# Nebenläufigkeit: Phantom-Read-Problem

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie

Phantom-Read: Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

| | Transaktion Benutzer 1 | Transaktion Benutzer 2 |
|---|---|---|
| 1 | SELECT count(*) INTO cnt FROM Tbl | |
| 2 | N = cnt | |
| 3 | | INSERT INTO Tbl VALUES (…) |
| 4 | | COMMIT |
| 5 | SELECT count(*) INTO cnt FROM Tbl | |
| 6 | | |




- Keine Isolation der Transaktionen beider Benutzer → Benutzer 1 sieht Zwischenresultate, die von Benutzer 2 eingefügt wurden!
- Lösung: Nur den Datenbankzustand sehen, der bei Beginn einer Transaktion vorliegt

9

Auch hier ist es durchaus üblich, diesen «Fehler» zu erlauben.

## Nebenläufigkeit in der Praxis

Zusammenfassend:

- Lost Update:
  kann in einem Transaktionssystem fast nie toleriert werden.
- Dirty-Read, Non-Repeatable-Read, Phantom-Read:
  bei konkurrentem Lesen oft zu einem gewissen Ausmass tolerierbar

10

Zur Implementation der Nebenläufigkeit bestehen verschiedene Möglichkeiten, welche wir zum Teil später genauer betrachten werden:

- Sperren (pessimistische Verfahren): Dabei existieren verschiedene Arten von Sperren (z.B. Schreib- und Lese-Sperren) und unterschiedliche Sperrobjekte (z.B. Zeile oder ganze Tabelle), welche auch als Sperr-Granularität bezeichnet wird.
- Snapshot (optimistische Verfahren): Speichern den Zustand der Daten während der Ausführung der Transaktion zwischen (Multi Version Read Consistency z.B. in Oracle, SQL-Server mit Snapshot Isolation).
- Zeitstempelverfahren: Kennzeichnen den Zeitpunkt des Lesens/der Änderungen der Daten

Da verschiedene Verfahren in verschiedenen Datenbanksystemen unterschiedlich Implementiert und kombiniert sind, muss man sich beim Einsatz eines Datenbanksystems mit dessen Verfahren vertraut machen und die geeignete Variante je nach Situation auswählen.

## Isolationsebenen im SQL-Standard

School of Engineering
InIT Institut für angewandte Informationstechnologie

- Isolationsebenen im SQL-92 Standard

```
SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | …}
```

- Isolationsebenen vs. Phänomene:

| Isolationsebene | Dirty Read | Non-Repeatable Read | Phantom Read | Lost Update |
|---|---|---|---|---|
| READ UNCOMMITTED | möglich | möglich | möglich | möglich |
| READ COMMITTED (Häufig in der Praxis) | verhindert | möglich | möglich | verhindert |
| REPEATABLE READ | verhindert | verhindert | möglich | verhindert |
| SERIALIZABLE | verhindert | verhindert | verhindert | verhindert |

11

Im ANSI SQL-92 Standard sind 4 Isolationsebenen definiert. Diese 4 Isolationsebenen werden den drei Phänomenen Dirty-Read, Non-Repeatable-Read und Phantom-Read gegenüber gestellt (ohne Lost-Update). Read Uncommitted (Zugriff auf nicht endgültig geschriebener Daten) ist die schwächste Stufe. Dieser Isolationslevel verwendet keinerlei Sperren, ist damit effizient ausführbar und behindert keine anderen Transaktionen:

- nur für read only Transaktionen verwenden, da sonst Lost Update möglich
- für statistische und ähnliche Transaktionen verwenden (ungefährer Überblick, nicht korrekte Werte)

Diese schöne Übersicht ist die Vorgabe durch den SQL-92 Standard, was aber noch nicht heisst, dass der implementierte Mechanismus des Datenbanksystems sich an diese Tabelle hält. Im konkreten Fall muss die Dokumentation des Datenbanksystems konsultiert werden.

In der Praxis sind noch weitere Phänomene (z.B. der Lost-Update) bekannt, welche in dieser Gegenüberstellung im SQL-92-Standard nicht berücksichtigt werden (in der Darstellung oben ist das Lost-Update-Phänomen zusätzlich aufgeführt). Da dieses Phänomen nicht im SQL-92 Standard beschrieben ist, häufen sich hier Abweichungen. Im SQL-Server wird im READ COMMITTED der Lost-Update verhindert, anders bei Oracle.

SQL-Server unterstützt:

- Alle Isolationsebenen nach ANSI (mittels Sperren implementiert)
- Zusätzlich Isolationsgrade, die mittels Snapshots (Row Versioning) implementiert sind: SNAPSHOT und READ COMMITTED SNAPSHOT
- Voreinstellung: Read Committed

Im SQL-Server kann innerhalb von Transaktionen je Operation angegeben werden, ob und wie Sperren gesetzt werden sollen (z.B. `SELECT * FROM TEST WITH (UPDLOCK)`). Dies kann gezielt genutzt werden, um die Performance und die parallele Ausführung von Transaktionen zu optimieren.

## Transaktionen in der Praxis

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- Ein Anwendungsentwickler muss lediglich die Transaktionsgrenzen festlegen, wird aber von allen anderen Aspekten zur Realisierung der Garantien befreit:
  - „COMMIT “: Transaktionsresultat wird permanent.
  - „ROLLBACK “, „ABORT“: Annullierung aller Veränderungen, Unsichtbarkeit für andere Transaktionen.
- Transaktionsanweisungen in SQL:
  - BEGIN TRANSACTION (implizit und/oder explizit)
  - COMMIT TRANSACTION
  - ROLLBACK TRANSACTION
- Werden durch Fehler beendet die die Sitzung abbrechen, durch Systemabsturz oder durch Verlassen der Sitzung ohne Commit
  → inkonsistenter Zustand, muss durch Recovery behoben werden.

12

Der Beginn einer Transaktion wird bei einigen Systemen explizit gesetzt (d.h. es MUSS der Befehl BEGIN TRANSACTION gegeben werden, ansonsten wird bei einem COMMIT TRANSACTION ein Fehler ausgegeben). Andere DBMS (z.B. Oracle) legen den Beginn einer Transaction implizit fest. Dann muss dem Entwickler absolut klar sein, wie das DBMS diesen Startpunkt festlegt, ansonsten werden eventuell Lese-Operationen ausgeführt, die nicht der Teil der Transaktion sind (was z.B. zu Non-Repeatable-Read führt).

Wird eine Transaktion aus irgend einem Grunde abgebrochen, so müssen alle durch diese Transaktion ausgeführten Änderungsoperationen auf jeden Fall rückgängig gemacht werden. Wir werden diese Recovery-Mechanismen noch kennen lernen.

Beim SQL Server müssen die Transaktionsgrenzen explizit gesetzt werden. Jede einzelne DBMS-Operationen ausserhalb von Transaktionsgrenzen wird wie eine eigene Transaktion ausgeführt. Wird diese Operation (ausserhalb von Transaktionsgrenzen) abgebrochen, so werden allfällig ausgeführte Update-Operationen dieser einzelnen Operation zurückgesetzt.

Wird eine Sitzung ohne Commit/Rollback verlassen, weiss das DBMS ja nicht, dass die Sitzung nicht korrekt abgeschlossen wurde, es erhält einfach keine weiteren Anweisungen mehr. Es ist daher in Datenbanksystemen üblich, dass für eine Transaktion, in welcher für eine bestimmte Zeit (z.B. für 15 Minuten) keine Befehle mehr ausgeführt wurden, automatisch ein Rollback ausgelöst wird. Das selbe passiert natürlich auch, wenn der User während einer aktiven Transaktion (die auf Eingaben wartet) für längere Zeit in die Cafeteria geht.

Nested Transactions beim SQL Server: Unter bestimmten Umständen kann es sein, dass Transaktionen innerhalb von Transaktionen verschachtelt (nested) werden (z.B. wenn eine Stored Procedure eine andere Stored Procedure aufruft). Kann dann nur die innere Transaktion zurückgesetzt werden? Nein, die verschachtelten Transaktionen haben keinerlei Wirkung. Der SQL Server zählt die Anzahl Begin- und Commit-Befehle und schliesst die Transaktion erst mit dem letzten Commit-Befehl ab.

## Transaktionen in der Praxis

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- DDL-Anweisungen / je DBMS:
  - Einbettung in laufende Transaktion (z.B. SQL-Server)
  - Kapselung in eigener Transaktion (z.B. Oracle; die laufende Transaktion wird committet!)
- Constraint-Verletzung / je DBMS:
  - Laufende Überprüfung und sofortiger Rollback (Oracle, SQL-Server).
  - «deferred constraint checking»: Constraints werden am Ende der Transaktion überprüft (Oracle). SQL Server kennt keine deferred constraints.

13

DDL-Befehle sind auch Teil einer Transaktion. Wird z.B. ein Attribut in eine Tabelle eingefügt, kann diese Operation mittels Rollback wieder rückgängig gemacht werden.

Bezogen auf Konsistenzerhaltung (Constraints) und Transaktionen soll an dieser Stelle ein für die Praxis sehr wichtiger Aspekt erwähnt werden (anschliessend betrachten wir Consistency nicht mehr). Wir definieren bei der Erstellung einer Datenbank drei grundsätzliche Typen von Konsistenzbedingungen: Primärschlüssel-, Fremdschlüssel-Bedingungen und weitere beliebige Bedingungen (z.B. der Lohn muss grösser 0 sein). Es stellt sich jetzt die Frage, müssen diese Bedingungen immer oder jeweils erst am Ende der Transaktion erfüllt sein. Diese wichtige Unterscheidung wird leider nicht durch alle DBMS unterstützt. So fordert der SQL Server zum Beispiel, dass die Konsistenzbedingungen immer erfüllt sein müssen. Dies hat z.B. zur Folge, dass Daten mit Fremdschlüsseln erst eingefügt werden können, wenn die referenzierten Daten schon existieren. Die Daten müssen daher in der richtigen Reihenfolge eingefügt werden! Das erfordert zum Teil eine aufwändigere Programmierung der Anwendungen…

## Transaktionen in der Praxis

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

Zustand DB bei offener Transaktion:

- vorhergehender Zustand wird in sog. Before-Images festgehalten und kann damit wieder hergestellt werden.
- betroffenen Sätze sind durch Schreib- / Lese-Sperren blockiert; sie können in anderen Transaktionen nicht gelesen / geändert werden.
- andere Transaktionen sehen *geänderte* Daten nicht (ausser bei Isolationsgrad Uncommitted Read).

14

Nebst den Before-Images werden auch After-Images (diese werden wir später beim Recovery brauchen) in einem Log-File festgehalten. Die After-Images protokollieren den Zustand nach einer Änderung. Die Before-Images können beim Ende der Transaktion allenfalls gelöscht werden. Soll es aber auch möglich sein, einen x-beliebigen Zustand in der Vergangenheit wieder herzustellen (z.B. gestern um 13:34:15), dann müssen die Before-Images natürlich behalten werden (es ist in Datenbanksystemen üblich, dass ein x-beliebiger Zeitpunkt in der Vergangenheit wieder hergestellt werden kann).

Eine Lese-Sperre auf Daten zeigt, dass eine laufende Transaktion die Daten gelesen hat. Eine Schreib-Sperre zeigt, dass eine laufende Transaktion die Daten verändert hat. Im Zusammenspiel mit den Isolationsgraden wird anderen Transaktionen nun der lesende oder schreibende Zugriff auf diese Daten erlaubt oder verboten.

## Transaktionen in der Praxis

Zustand DB nach COMMIT:

- Die geänderten Daten sind in der Datenbank festgeschrieben.
- Alle Datenänderungen - alter Zustand (Before-Images) und neuer Zustand (After-Images) - sind in den Transaktionslogs protokolliert.
- Vorhergehender Zustand kann nicht mehr mittels ROLLBACK wieder hergestellt werden. Aber wie sonst?
- Alle Sperren der Transaktion sind frei gegeben.
- Geänderten Daten können in Transaktionen mit «Read Committed Isolation» oder in nachfolgenden Transaktionen gelesen werden.
- Geänderte Daten können in anderen Transaktionen geändert werden (falls nicht weitere Transaktionen Sperren gesetzt haben).

15

Zu Punkt 1 und 2: Die eigentlichen, geänderten Daten sind allenfalls noch nicht in der Datenbank festgeschrieben (erst im Puffer). Aus Performance-Gründen werden für das Transaktionslog möglichst effiziente Algorithmen und Speicherstrukturen verwendet. Die Daten auf die externen Speichermedien selbst zu schreiben ist dann allenfalls langsamer. Daher wird evtl. darauf verzichtet, die eigentlichen Daten bereits beim Commit auf externe Speichermedien zu schreiben (die Informationen sind ja zur Sicherheit im Transaktionslog).

Frage aus Punkt 3: Mit einen Backup und sämtlichen Before- und After-Images seit dem Backup kann jeder beliebige Zeitpunkt der Datenbank in einem konsistenten Zustand wieder hergestellt werden. Dies kann z.B. interessant sein, wenn beim Deployment eine fehlerhafte Software eingespielt wurde (welche die Daten zerstört hat).

Zu Punkt 5: Darf eine Transaktion, welche vor dem Commit der anderen Transaktion gestartet wurde, nach deren Commit Daten lesen, welche durch die andere Transaktion verändert wurden? Streng genommen, kann dadurch ein Konflikt entstehen, da die Transaktion so eine inkonsistente Sicht auf die Daten erhalten könnte (andere Daten, die sie vorher gelesen hat, passen allenfalls nicht zu diesen ‘neuen’ Daten). Ab Isolationsgrad READ REPEATABLE wird dies verhindert.

# Transaktionen in der Praxis

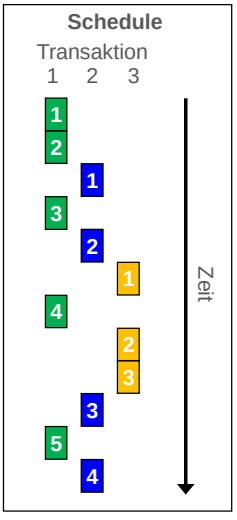
Zustand DB nach ROLLBACK:

- Geänderten Daten sind vollständig zurückgesetzt.
- Herstellen des alten Zustandes durch Einsetzen der Before-Images.
- Rollback-Vorgang selbst wird auch in den Transaktionslogs aufgezeichnet.
- Sperren der betroffenen Daten sind frei gegeben.
- Daten können in anderen Transaktionen gelesen und geändert werden.

16

# Schedules

- Schedule (Ablaufplan):
  - Folge von Lese- bzw. Schreiboperationen für die (parallele) Ausführung einer oder mehrerer Transaktionen.
  - Schedules können ACID-Eigenschaften verletzen.
- Vollständiger Schedule = History
  - Sämtliche Schritte aller anstehenden Transaktionen inklusive Terminierung (COMMIT, ABORT)
  - Für jede Transaktion ist festgehalten, ob sie erfolgreich endet oder abbricht (dies kann in einem Schedule noch offen sein).




17

Achtung: Ablaufplan (Schedule) nicht mit Ausführungsplan (Execution Plan) der Query verwechseln! Der Ausführungsplan beschreibt, welche Basisoperationen in welcher Reihenfolge zum Ausführen *einer* Query abgearbeitet werden. Der Ablaufplan beschreibt, wie mehrere Transaktionen in welcher Reihenfolge ihre Operationen ausführen/mischen.

Werden die Lese- und Schreiboperationen (auf Datenobjekten der Datenbank) der verschiedenen Transaktionen in zufälliger Reihenfolge ausgeführt, können die in der letzten Lektion eingeführten Probleme (Lost-Update, Dirty-Read, etc.) auftreten. Sprich, es werden eine oder mehrere der ACID-Eigenschaften verletzt.

Die History stellt einen vollständigen Schedule dar (wir werden nachfolgend den Begriff vollständiger Schedule verwenden). In der Praxis ist die History eigentlich ‘nie’ abgeschlossen. Für uns ist die History, der vollständige Schedule insofern wichtig, da wir untersuchen werden, wann ein vollständiger Schedule ein korrektes Resultat erzeugt.

# Schedules

School of Engineering
InIT Institut für angewandte Informationstechnologie

- Darstellung Lesen und Schreiben: $r_n(x)$ bzw. $w_n(x)$
  (r = read, w = write, x = DB-Objekt, n = Transaktionsnummer)
- Darstellung Terminierungsoperationen: $c_n$ = commit, $a_n$ = abort
- Beispiel mit drei Transaktionen T = $\{t_1,t_2,t_3\}$:
  - $t_1 = r_1(x)w_1(x)r_1(y)w_1(y)r_1(z)c_1$
  - $t_2 = r_2(x)w_2(x)w_2(y)w_2(z)c_2$
  - $t_3 = w_3(x)r_3(y)w_3(z)c_3$
  - Lost-Update Schedule; $w_1(x)$ geht verloren:
    $s_{lu} = r_1(x)r_2(x)\boldsymbol{w_1(x)}w_2(x)\ldots$
  - Vollständiger Schedule (History):
    $s_{vs} = r_1(x)w_1(x)r_2(x)r_1(y)\ w_2(x)w_3(x)w_1(y)r_3(y)w_3(z)w_2(y)r_1(z)c_1c_3w_2(z)c_2$

18

Ein BEGIN TRANSACTION ist in obiger Notation nicht nötig, da jede Transaktion eine eindeutige Nummer erhält (und mit der ersten Operation beginnt).

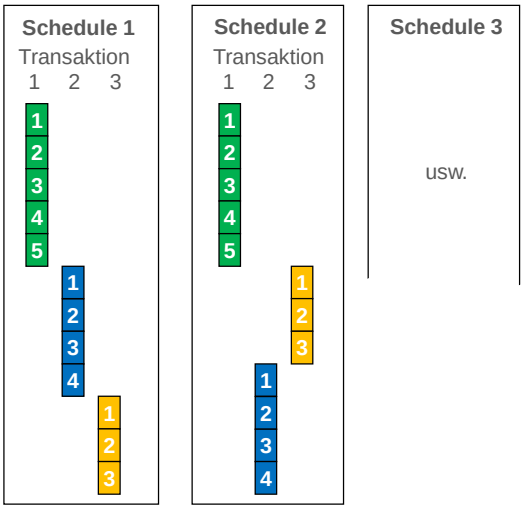
Führt der vollständige Schedule auf der Folie zu einem korrekten Resultat? Wir müssen zunächst festlegen, was korrektes Resultat überhaupt heisst – siehe nächste Folien. Wir werden dabei nicht alle möglichen Probleme (z.B. Lost-Update und Dirty-Read und und und) bestimmen, sondern einen zielstrebigeren Ansatz wählen.

# Schedules

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

Serieller Schedule:

- Alle Transaktionen nacheinander.
- Für n Transaktionen existieren n! verschiedene serielle Schedules.
- Serielle Schedules werden als konsistenzerhaltend betrachtet.
- Sicher, aber schlechte Performance → verschränkte Ausführung ist erwünscht.




19

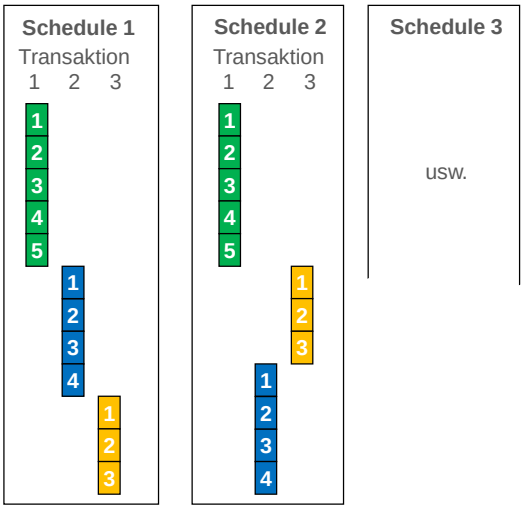
Werden n Transaktionen ausgeführt, dann führt die serielle Ausführung (= serieller Schedule, ohne zeitliche Überschneidung der Transaktionen) der Transaktionen sicher zu einem korrekten Resultat. Dies ergibt n-Fakultät mögliche Schedules. Dabei kann es auch sein, dass jeder Schedule zu einem anderen Resultat führt – dennoch sind alle Schedules, alle Resultate konsistenzerhaltend.

## Schedules

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

Serialisierbarer Schedule ist auch konsistenzerhaltend.

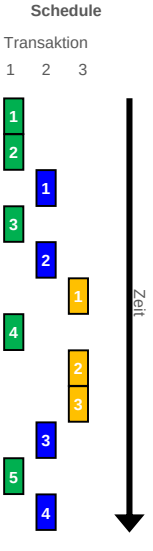
Konfliktserialisierbar: Hat denselben Effekt auf die Datenbank, wie einer der seriellen Schedules. Hierzu existieren effiziente Algorithmen.




20

Eine serielle Ausführung von Transaktionen ist natürlich unerwünscht, Transaktionen müssen parallel ausgeführt werden können (wie im Schedule unten). Wir sagen einem Schedule nun serialisierbar, falls dieser zu einem 'identischen' seriellen Schedule umgewandelt werden kann (wie einer der n-Fakultät Schedules oben), so dass das Ergebnis das selbe ist.

Wir brauchen jetzt noch einen effizienten Algorithmus um sicherzustellen, dass unsere Schedules serialisierbar sind. Auf den folgenden Folien werden wir zeigen, dass dies für konfliktserialisierbare Schedules gegeben ist.



## Serialisierbarer Schedule

Wann ist ein Schedule konfliktserialisierbar? (im Folgenden sprechen wir nur noch von serialisierbar)

- Abhängigkeit (Konflikt) besteht, wenn zwei Transaktionen auf dasselbe Objekt mit reihenfolgeabhängigen Operationen zugreifen. Konfliktarten:
  - Schreib-/Lese-Konflikt $w_1(x)$ $r_2(x)$
  - Lese-/Schreib-Konflikt $r_1(x)$ $w_2(x)$
  - Schreib-/Schreib-Konflikt $w_1(x)$ $w_2(x)$
- Ein Schedule s ist nun konfliktserialisierbar, falls ein serieller Schedule existiert, dessen Konflikte identisch sind.
- Nachweis der Serialisierbarkeit: Führen von zeitlichen Abhängigkeiten zwischen Transaktionen in einem Abhängigkeitsgraphen (Konfliktgraphen).

21

Zwei Operationen zweier Transaktionen sind konfliktär, wenn die Vertauschung der Reihenfolge zu einem anderen Ergebnis führt (eine einzelne Transaktion kann in sich nicht konfliktär sein). Wird nur lesend auf Daten zugegriffen, kann auch kein Konflikt auftreten, da die Reihenfolge dann keine Rolle spielt.

Im Schedule $s_{vs}$ unten sind z.B. $w_1(x)$ und $r_2(x)$ konfliktär, ein vertauschen dieser beiden Operationen in $s_{vs}$ führt (potentiell) zu einem anderen Resultat. Ein Schedule s ist nun konfliktserialisierbar, falls ein serieller Schedule existiert, dessen Konflikte identisch sind.

Ein Beispiel soll dass erläutern:

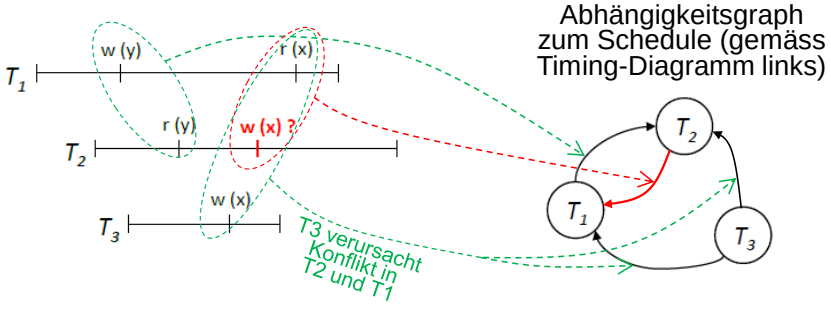
- Zu prüfender Schedule $s_{vs}$: $r_1(x)w_1(x)r_2(x)r_1(y)w_2(x)w_3(x)w_1(y)r_3(y)w_3(z)w_2(y)r_1(z)c_1a_3w_2(z)c_2$
- Serieller Schedule $s_1$: $r_1(x)w_1(x)r_1(y)w_1(y)r_1(z)c_1r_2(x)w_2(x)w_2(y)w_2(z)c_2w_3(x)r_3(y)w_3(z)a_3$

Während in $s_{vs}$ y zuerst durch $r_3(y)$ gelesen wird und anschliessend durch $w_2(y)$ geschrieben wird (rote Buchstaben), ist die Reihenfolge in $s_1$ umgekehrt – die beiden Konflikte sind nicht identisch gelöst, der Schedule $s_{vs}$ ist damit nicht konfliktserialisierbar bezogen auf den Schedule $s_1$. Natürlich müssten wir zur abschliessenden Beurteilung, ob $s_{vs}$ konfliktserialisierbar ist jetzt wieder alle möglichen seriellen Schedules überprüfen (und Transaktionen mit abort ausschliessen).

# Serialisierbarer Schedule



- Man kann zeigen, dass ein Schedule serialisierbar ist (d.h. er führt zum selben Resultat wie ein beliebiger serieller Schedule), wenn der Abhängigkeitsgraph keine Zyklen enthält (in $O(n^2)$ entscheidbar):




- Dieser Schedule ist nicht serialisierbar, da der Graph einen Zyklus aufweist.



Die Überprüfung, ob ein Schedule konfliktserialisierbar ist, kann durch einen Abhängigkeitsgraphen (auch Konflikt, oder Schedulegraphen) erfolgen. Der Name ist etwas verwirrend, im Graphen werden nicht wie vorher die konfliktären Operationen, sondern es werden die konfliktären Transaktionen aufgezeigt:

- Die Knoten im Abhängigkeitsgraphen stellen die Transaktionen des gesamten Schedule dar.
- Die Pfeile stellen die Konflikte (Abhängigkeiten) zwischen den Transaktionen dar (es wird nicht je Datenobjekt, sondern je Transaktion der Konflikt dargestellt)

Im Konfliktgrafen ist ersichtlich, dass $T_1$ mit w(y) in $T_2$ mit r(y) eine Abhängigkeit (Konflikt) erzeugen. Dies ist noch kein Problem. Erst wenn die umgekehrte Abhängigkeit in den selben Transaktionen auftritt, hier mit w(x) in $T_2$ und r(x) in $T_1$, lässt sich diese Reihenfolge nicht mehr in einen seriellen Schedule abbilden. Ein Schedule ist also nicht mehr konfliktserialisierbar, sobald ein Zyklus im Graph auftritt.

Wir können jetzt entscheiden, was ein korrekter Schedule ist und was nicht, aber wie implementieren wir das? Der Abhängigkeitsgraph ist kein geeignetes Mittel um ein Synchronisierungsverfahren zu implementieren. Erstens zeigt der Abhängigkeitsgraph meist erst nachträglich, ob der Schedule serialisierbar ist, und zweitens wäre der Verwaltungsaufwand viel zu gross, da auch bereits beendete Transaktionen im Graphen verwaltet werden müssen. Zur Synchronisation werden daher andere Verfahren / Algorithmen verwendet, für welche nachgewiesen werden kann, dass diese die Serialisierbarkeit gewährleisten. Wir werden diese Scheduling-Verfahren (Sperrverfahren) im Verlauf der nächsten Lektion einführen.

# Scheduler / Transaktionsmanager

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

Scheduler:

- Erzeugt einen serialisierbaren Ablaufplan s für parallel auszuführende Transaktionen T → welche Transaktion macht den nächsten Schritt?
- Verwaltet notwendige Synchronisationsinformationen, z.B. die Lese- und Schreibsperren

- Beispiel mit drei Transaktionen T = $\{t_1,t_2,t_3\}$:
  - $t_1 = r_1(x)w_1(x)r_1(y)w_1(y)r_1(z)c_1$
  - $t_2 = r_2(x)w_2(x)w_2(y)w_2(z)c_2$
  - $t_3 = w_3(x)r_3(y)w_3(z)a_3$
  - $s = r_1(x)w_1(x)w_3(x)r_1(y)r_2(x)w_2(x)w_1(y)r_3(y)r_1(z)c_1w_3(z)a_3w_2(y)w_2(z)c_2$

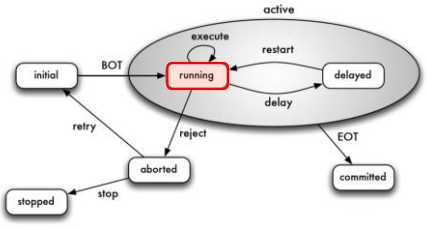
23

Im Beispiel werden drei Transaktionen parallel ausgeführt, der Scheduler hat daraus den Schedule s erzeugt.

«Kleine» Übung: zeigen Sie mittels Abhängigkeitsgraphen, dass der Schedule s tatsächlich konflikt-serialisierbar ist.

# Scheduler / Transaktionsmanager

- Eine Transaktion kann folgende Zustände einnehmen:




- Für den Scheduler bestehen folgende drei Möglichkeiten zur Behandlung eines Schrittes einer laufenden (running) Transaktion:
  - Ausführen (execute): Transaktion → Zustand running
  - Verzögern (delay): Transaktion → Zustand delayed
  - Zurückweisen (reject): Transaktion → Zustand aborted (z.B. wenn Serialisierbarkeit gefährdet würde)

# Scheduler / Transaktionsmanager

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

Scheduling-Verfahren:

1. Ein Scheduler arbeitet aggressiv, wenn er Konflikte zulässt und dann versucht, aufgetretene Konflikte zu erkennen und aufzulösen:
   - Ziel: Maximiert die Parallelität von Transaktionen
   - Risiko: Transaktionen werden erst am Ende ihrer Ausführung zurückgesetzt
   - Grenzfall: Im Extremfall ist keine Transaktion mehr erfolgreich
   - Beispiel: Oracle (Scheduling-Verfahren: Mehrversionensynchronisation)
2. Ein Scheduler arbeitet konservativ, wenn er Konflikte möglichst vermeidet, dafür aber Verzögerungen von Transaktionen in Kauf nimmt:
   - Ziel: Minimiert den Rücksetzungsaufwand für abgebrochene Transaktionen
   - Risiko: Erlauben eine geringere Parallelität von Transaktionen
   - Grenzfall: Im Extremfall findet keine Parallelisierung von Transaktionen mehr statt, d.h. die Transaktionen werden sequentiell ausgeführt
   - Beispiel: SQL Server (Scheduling-Verfahren: Sperrverfahren; nutzt 7 Haupt-Lock-Modi und diverse Untermodi zur Optimierung)

25

# Ausblick

- Das nächste Mal:
- Sperren: Konzept, Sperrprobleme (Deadlocks…)
- Recovery
- Lesen: Lehrbuch Kapitel 10.1-10.3 und 10.8 (13 Seiten)
  Zusätzlich: 11.1 und 11.2 (6 Seiten)

26

# Anhang

# Beispiele ACID-Prinzip



ACID-Prinzipien am Beispiel einer Banküberweisung:

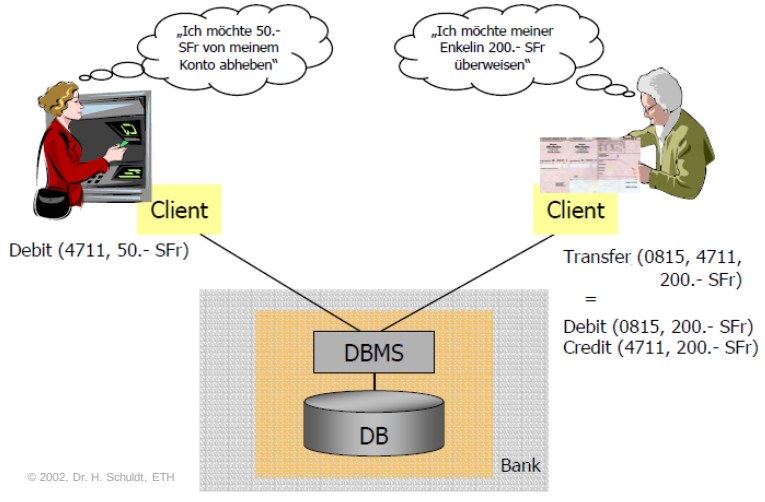
- Klassischer Vertreter von Datenbanktransaktionen.
- Kurzlebige Transaktionen, operieren auf wenigen Datenobjekten.
- Viele Parallele Zugriffe auf Konto-Tabelle:

```
Konto (KontoNr, KundenNr, Kontostand, …)
```

- Typische Operationen:
  - Debit (KontoNr, Betrag): bucht Betrag von Konto KontoNr ab.
  - Credit (KontoNr, Betrag): bucht Betrag auf Konto KontoNr.
  - Transfer (KontoNrA, KontoNrB, Betrag): besteht aus einem Debit(KontoNrA, Betrag) und einem Credit(KontoNrB, Betrag)

28

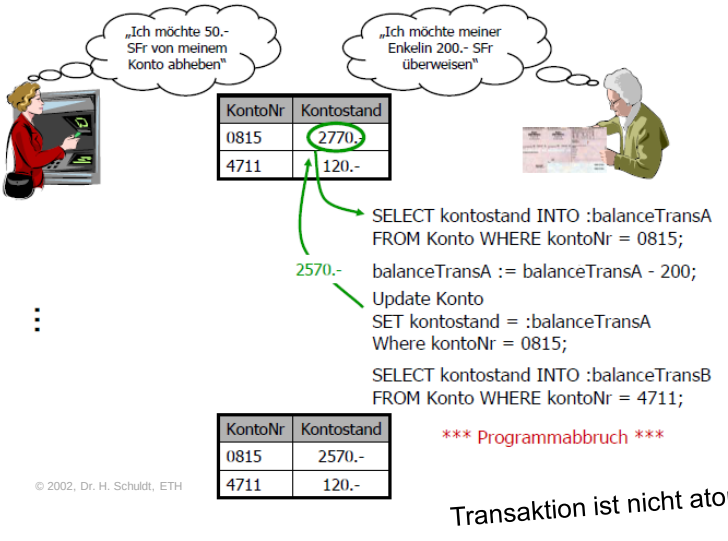
# Beispiele ACID-Prinzip




29

# Beispiele ACID-Prinzip: Atomicity

Zürcher Hochschule für Angewandte Wissenschaften
zh aw School of Engineering
InIT Institut für angewandte Informationstechnologie






Transaktion ist nicht atomar

30

# Beispiele ACID-Prinzip: Atomicity

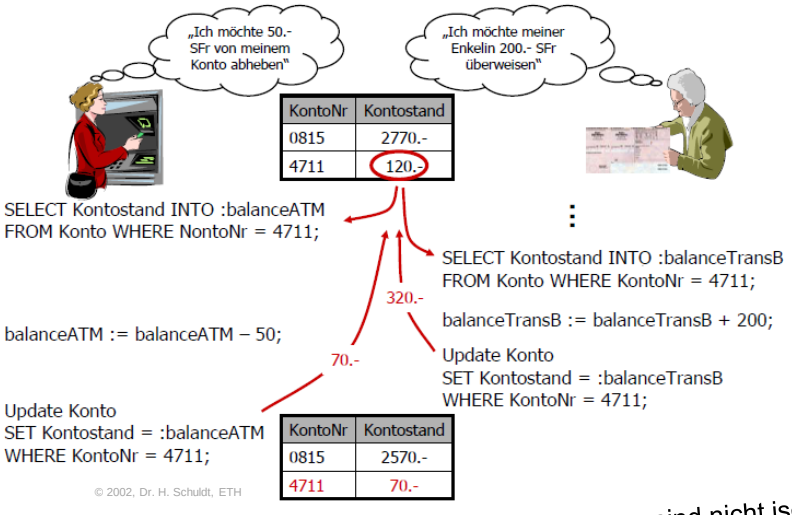
Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

- Atomarität: Im vorigen Ablauf ist die «Alles-oder-Nichts-Semantik» der Atomarität verletzt. Ein inkonsistenter Zwischenzustand bleibt zurück (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 wiederherstellen, der direkt vor Beginn der Überweisungstransaktion vorlag.

31

# Beispiele ACID-Prinzip: Isolation




Transaktionen sind nicht isoliert

32

# Beispiele ACID-Prinzip: Isolation

- Isolation: Im vorigen Ablauf überschneiden sich die beiden Transaktionen in inkorrekter Weise. Die 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.

33

# Beispiele ACID-Prinzip: Consistency & Durability

Zürcher Hochschule für Angewandte Wissenschaften
School of Engineering
InIT Institut für angewandte Informationstechnologie

Weitere Anforderungen:

- Konsistenz: z.B. Einschränkung, dass der Kontostand niemals negativ werden kann. Dies wird in der Regel durch Integritätsbedingungen zugesichert (z.B. einer CHECK-Klausel). 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 System-abstürze überstehen.

34