**Transaktionen**

Datenbanken müssen für den gleichzeitigen Zugriff mehrerer Benutzer ausgelegt sein. Daraus können sich mannigfaltige Probleme ergeben. DBMS versuchen mit dem Prinzip der **Transaktion** diese Probleme in den Griff zu bekommen.

Eine Transaktion ist als eine Folge von Anweisungen zu sehen, die entweder komplett übernommen (committ) oder abgebrochen (rollback) werden.

Im Kontext dieses **Alles oder Nichts** kann es unterschiedlichen Spielarten geben. Insbesondere muss man sich darüber im Klaren sein, wie man den Zugriff anderer Clients auf die durch eine Transaktion gesperrten Datensätze regelt.

**Probleme bei gleichzeitigen Zugriff**

**Lost Updates**

Mehrere Transaktionen wollen zur gleichen Zeit einen einzelnen Datensatz bearbeiten.

| **Zeit** | **Transaktion 1** | **Transaktion 2** |
| --- | --- | --- |
| 1 | read(x) |  |
| 2 |  | read(x) |
| 3 |  | x = x + 100 |
| 4 |  | write(x) |
| 5 | x = x + 1 |  |
| 6 | write(x) |  |
| 7 |  |  |

In diesem Szenario ergibt sich für T2 der Verlust seiner Änderung

**Dirty Reads**

Ein User sieht Änderungen, die noch nicht von ihm selbst oder von anderen Usern committed oder rollbacked wurden.

| **Zeit** | **Transaktion 1** | **Transaktion 2** |
| --- | --- | --- |
| 1 | read(x) |  |
| 2 | x = x + 100 |  |
| 3 | write(x) |  |
| 4 |  | read(x) |
| 5 |  | x = x - 100 |
| 6 | ROLLBACK |  |
| 7 |  | write(x) |

In diesem Falle konnte T2 schon auf Daten zugreifen, die von T1 noch nicht endgültig freigegeben wurden.

**Non repeatable Reads**

User wählen wiederholt Zeilen aus, die andere User ändern oder löschen. Ob dies ein Problem darstellt, hängt von den jeweiligen Umständen ab (Inventur vs. Reisebüro)

**Ausgangsbedingung**

x =40 y =50 z = 30

| **Zeit** | **Transaktion 1** | **ransaktion 2** |
| --- | --- | --- |
| 1 | sum = 0 |  |
| 2 | read( x ) |  |
| 3 | read( y ) |  |
| 4 | sum = sum + x |  |
| 5 | sum = sum + y |  |
| 6 |  | read( z ) |
| 7 |  | z = z 10 |
| 8 |  | write( z ) |
| 9 |  | read( x ) |
| 10 |  | x = x + 10 |
| 11 |  | write( x ) |
| 12 | read( z ) |  |
| 13 | sum = sum + z |  |

**Phantom Rows**

Ein User kann einige, aber nicht alle neuen Datensätze lesen, die ein anderer User eingegeben hat.

| **Zeit** | **Transaktion 1** | **Transaktion 2** |
| --- | --- | --- |
| 1 | Select Counter |  |
|  | from PassCounter |  |
| 2 |  | Update Passengers |
|  |  | set Flight = 4711 |
|  |  | where Name = 'Phantom' |
| 3 |  | Update PassCounter |
|  |  | set Counter = Counter + 1 |
| 4 | select \* |  |
|  | from Passengers |  |

**Konzept Transaktion**

Als Transaktion (von lateinisch trans „(hin-)über“, agere „treiben, handeln, führen“: also wörtlich: Überführung; dt. hier besser: Durchführung) bezeichnet man in der Informatik eine Folge von Programmschritten, die als eine logische Einheit betrachtet werden.

**ACID**

Transaktionen werden durch die sog. ACID - Eigenschaften beschrieben:

* **Atomarität**

Eine Transaktion wird entweder ganz oder gar nicht ausgeführt

* **Konsistenz** (Serialisierbarkeit)

Transaktionen überführen die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand. Dies wird durch das Prinzip der Serialisierung erreicht.

* **Isolation**

Nebenläufige (gleichzeitige) Transaktionen laufen jede für sich so ab, als ob sie alleine ablaufen würden.  
Verschiedene Isolationslevel sind möglich

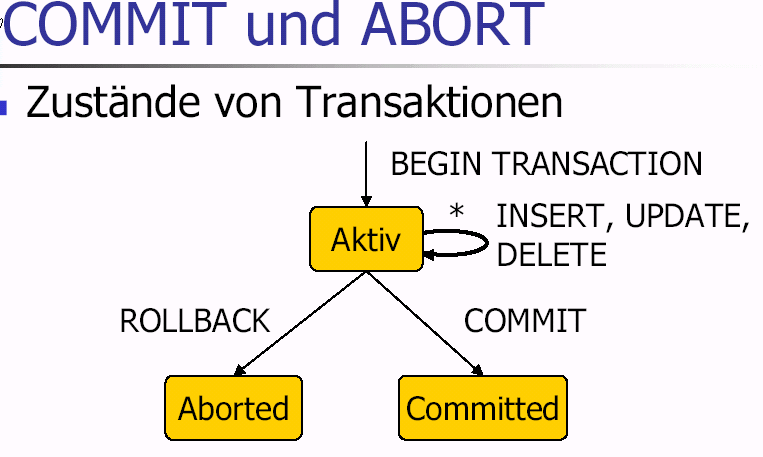
* **Dauerhaftigkeit**

Die Wirkung einer abgeschlossenen Transaktion bleibt (auch nach einem Systemausfall) erhalten. Dies wird durch spezielle Recovery-Mechanismen, z.B. WAL (Write Ahead Logs) erreicht

**Zustände**

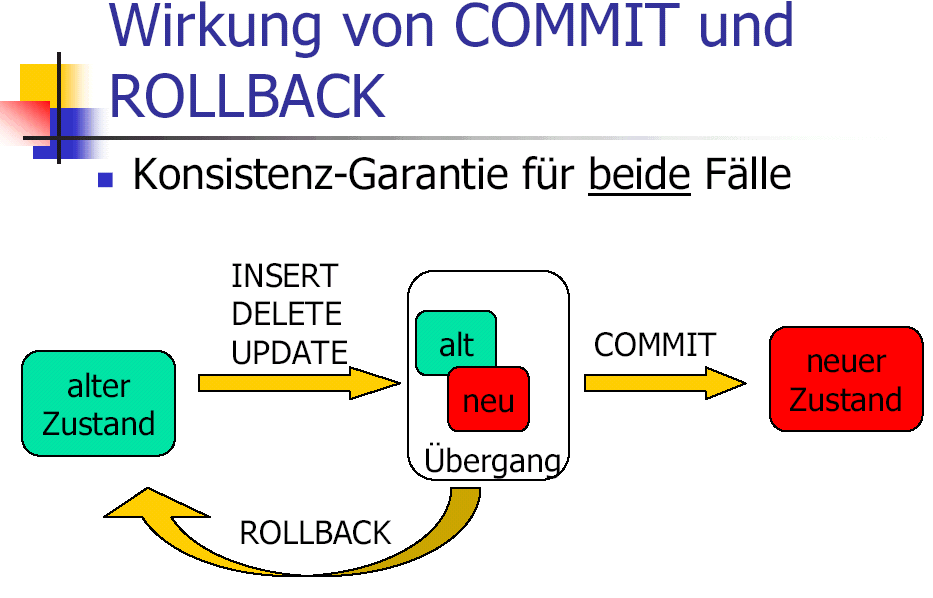
Eine Transaktion befindet sich immer in einem von drei Zuständen.

* Aktiv: Die Transaktion läuft ab; es finden die entsprechenden INSERT, UPDATE, DELETE - Anweisungen statt.
* Aborted: Aufgrund eines Fehlers wird die Transaktion zurückgesetzt (ROLLBACK)
* Committed: Bei fehlerfreier Ausführung gilt die Transaktion als abgeschlossen (COMMITTED)



**Konsistenz**

Das DBMS garantiert dafür, dass die Daten sich immer in einem konsistenten Zustand befinden. Sie hält dafür häufig mehrere Versionen der gleichen Daten, um auf einen vorhergehenden Datenbestand zurückgreifen zu können.



**Serialisierung**

Serialisierung prüft, ob gleichzeitig stattfindende Transaktionen sich gegenseitig behindern würden

**Pessimisistisch/Streng** Die einzelnen Transaktionen werden nur hintereinander ausgeführt

* keine Konflikte
* langsam

**Optimistisch/Weich**

Erst beim Abschluss einer Transaktion wird geprüft, ob sie sich serialisieren lässt. Sie kann deshalb abgebrochen werden und muss von vorne beginnen. Sie durchläuft normalerweise drei Phasen:

* Lesephase (Daten lesen und Berechnungen durchführen)
* Validierungsphase (überprüft Einhaltung der Konsistenzkriterien)
* Schreibphase (die in der Lesephase berechneten Änderungen werden eingetragen)

Optimistische Serialisierung lässt alle Reihenfolgen zu, die keinen Schaden anrichten können.

**überlappend, aber unschädlich**

| **Zeit** | **Transaktion 1** | **Transaktion 2** |
| --- | --- | --- |
| 1 | read(x) |  |
| 2 |  | read(y) |
| 3 | x = x + 100 |  |
| 4 |  | y = y - 100 |
| 5 | write(x) |  |
| 6 |  | write(y) |

**überlappend und schädlich, nicht serialisierbar**

| **Zeit** | **Transaktion 1** | **Transaktion 2** |
| --- | --- | --- |
| 1 | read(x) |  |
| 2 |  | read(x) |
| 3 |  | x = x + 100 |
| 4 |  | write(x) |
| 5 | x = x + 1 |  |
| 6 | write(x) |  |

**Isolations-Level**

Die Isolation-Level beschreiben, welche möglichen Konflikte ein Client beim gleichzeitigen Zugriff akzeptiert, d.h. inwieweit er bereit ist, mit inkonsistenten Daten zu arbeiten.

Mysql kennt folgende Arten (<https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html>)

Siehe dazu auch: <http://www.ovaistariq.net/597/understanding-innodb-transaction-isolation-levels/#.WRn7bcakK00>

* **READ UNCOMMITTED**

Eine Transaktion2 sieht sofort alle Änderungen einer Transaktion1, auch wenn Transaktion1 noch nicht committed hat. Dies erlaubt den **Dirty\_Read**, d.h. das Lesen von noch nicht endgültig abgeschlossenen Anweisungen einer anderen Transaktion.

* **READ COMMITTED**

Eine Transaktio2 sieht nur die Änderungen einer Transaktion1, wenn Transaktion1 sie committed hat. Dies vermeidet den Dirty Read. Da jede Transaktion den zuletzt committeten Zustand erhält, kann es sein, dass zwischen zwei Selects einer Transaktion verschiedene Ergebnisse zurückgeliefert werden. Dies wird auch als **Unrepeatable Read** bezeichnet

* **REPEATABLE READ**

In diesem Falle behält jede Transaktion seinen eigenen Snapshot der Daten, der beim Beginn der Transaktion existiert hat. Dieser bleibt während der gesamten Dauer der Transaktion bestehen. Somit wird das Problem des Unrepeatable reads behoben, es bleibt jedoch das Problem der **Phantom Rows**

* **SERIALIZABLE**

In diesem Isolationslevel erzeugen die Transaktionen Locks auf alle zugegriffenen Daten sowie auf die benutzten Tabellen. Neue Daten können somit nicht hinzugefügt werden. Dies ist der strengste IsolationLevel und hat damit auch die meisten Auswirkungen auf die Geschwindigkeit einer Datenbank.

Isolationlevel haben auch Auswirkungen auf die Replikation einer Datenbank, da diese per default statement-basiert ist (die sql-Anweisungen werden auf den Slaves nochmals ausgeführt). Höhere Isolationslevel wie Repeatable-Read bzw. Serializable werden hier benötigt, um die Konsistenz der Daten sicherzustellen.

**Beispiel zu Isolation Level**

%load\_ext sql

%%sql

-- aus https://adayinthelifeof.nl/2010/12/20/innodb-isolation-levels/

Ein paar Beispiele.

Sie benötigen dazu eine kleine Tabelle mit zwei Spalten und zwei Datensätzen.

create table test(

id integer not null,

val integer

) engine = innodb;

insert into test(id, val) values(1,8);

insert into test(id, val) values(2,8);

+----+-----+

| id | val |

+----+-----+

| 1 | 8 |

| 2 | 8 |

+----+-----+

Wir greifen nun mit 2 Transaktionen auf die Tabelle zu und beobachten das Verhalten des Datenbankservers in

Abhängigkeit vom gewählten Isolationlevel.

read uncommited

---------------

TX A: start transaction;

TX B: set session transaction isolation level read uncommitted;

TX B: start transaction;

TX A: select \* from test; -- val = 8

TX B: select \* from test; -- val = 8

TX A: update test set val = val + 1; -- val = 9

TX B: select \* from test; -- val = 9, dirty read

TX A: rollback;

TX B: select \* from test; -- val = 8

TX B: commit;

Wie man sieht, ist es für die Transaktion B möglich, Daten zu sehen, die von der Transaktion A geändert wurden.

Nach dem Rollback der Transaktion A sind aber die Änderungen rückgängig gemacht worden..

read committed

--------------

TX A: start transaction;

TX B: set session transaction isolation level read committed;

TX B: start transaction;

TX A: select \* from test; -- val = 8

TX B: select \* from test; -- val = 8

TX A: update test set val = val + 1; -- val = 9

TX B: select \* from test; -- val = 8, No dirty read!

TX A: commit

TX B: select \* from test; -- val = 9, commited read

Ein Dirty Read ist nun nicht mehr möglich. Erst nach dem Commit durch Transaktion A sind die Daten

für Transaktion B sichtbar

Repeatable read

---------------

TX A: start transaction;

TX B: set session transaction isolation level repeatable read;

TX B: start transaction;

TX A: select \* from test; -- val = 8

TX B: select \* from test; -- val = 8

TX A: update test set val = val + 1; -- val = 9

TX B: select \* from test; -- val = 8

TX A: commit;

TX B: select \* from test; -- val = 8, repeatable read! -- Fehler, ist nicht so

TX B: commit;

TX B: select \* from test; -- val = 9 (from tx A)

Auch nach dem Commit durch Transaktion A sehen wir noch die unveränderten Ausgangsdaten.

Nur nach einem eigenen Commit bzw. Rollback sehen wir, dass sich die Daten geändert haben.

serializable

------------

TX A: start transaction;

TX B: set session transaction isolation level serializable;

TX B: start transaction;

TX A: select \* from test; -- val = 8

TX A: update test set val = val + 1; -- val = 9

TX B: select \* from test; -- LOCKED, NO OUTPUT

TX A: commit; -- Unlocked TX B

TX B: select \* from test; -- val = 8 (repeatable read!) -- ab hier ist das Beispiel falsch

TX B: commit;

TX B: select \* from test; -- val = 9 (now we see TX A)

Wie sie sehen, können wir nach einer Änderung in Transaktion A die Daten in der anderen Transaktion B

überhaupt nicht mehr sehen. Wir müssen warten, bis die Transaktion die Daten committed oder eine Rollback

durchführt.

Nachdem Transktion A seine Daten committed hat, kann Transaktion B die geänderten Daten immer noch nicht sehen,

da es ansonsten ein Non-Repeatable-Read wäre. Wir müssen unsere eigene Transaktion committen,

um die Änderunen zu sehen.