



Thomas Studer
Relationale Datenbanken:
Von den theoretischen Grundlagen
zu Anwendungen mit PostgreSQL
Springer, 2016
ISBN 978-3-662-46570-7

Dieser Foliensatz darf frei verwendet werden unter der Bedingung, dass diese Titelfolie nicht entfernt wird.

Datenbanken



Mehrbenutzerbetrieb

Thomas Studer

Institut für Informatik
Universität Bern

Einführung

Überweisung eines Betrags von einem Bankkonto A auf ein anderes Bankkonto B

- 1 Lastschrift des Betrags auf Konto A,
- 2 Gutschrift des Betrags auf Konto B.

ACID Postulate

- A - Die Änderungen einer Transaktion auf der Datenbank sind *atomar*, d.h. es werden alle oder keine dieser Operationen ausgeführt. (Atomicity)
- C - Eine Transaktion führt von einem *korrekten* DB-Zustand wieder in einen korrekten DB-Zustand über. (Consistency)
- I - Eine Transaktion arbeitet *isoliert* auf der DB. Dies bedeutet, sie wird bei simultaner Ausführung weiterer Transaktionen von diesen nicht beeinflusst. (Isolation)
- D - Wirksame Änderungen von T sind *dauerhaft*, d.h. sie dürfen auch im Fall einer späteren Fehlfunktion nicht mehr verloren gehen. (Durability)

Grundoperationen

- `READ(X)`: Transferiert die Daten `X` von der Datenbank auf einen lokalen Puffer, der zur Transaktion gehört, welche `READ(X)` ausführt.
- `WRITE(X)`: Transferiert die Daten `X` vom lokalen Puffer, der zur Transaktion gehört, welche `WRITE(X)` ausführt, zurück auf die Datenbank.

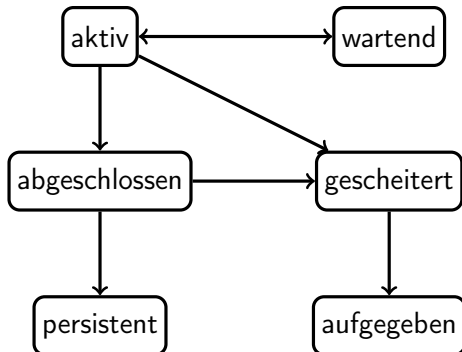
Wir können nun die Überweisung von CHF 50 von Konto A auf Konto B folgendermassen beschreiben:

```
READ(A)
A := A-50
WRITE(A)
READ(B)
B := B+50
WRITE(B)
```

Transaktionsmodell

- aktiv** Anfangszustand; während die Operationen der Transaktion ausgeführt werden, bleibt sie in diesem Zustand.
- wartend** Zustand, in welchem die Transaktion warten muss, bis benötigte Ressourcen (durch andere Transaktionen) freigegeben werden.
- abgeschlossen** Zustand nach Ausführung der letzten Operation der Transaktion. Aus diesem Zustand kann die Transaktion immer noch scheitern, da möglicherweise die Daten nicht persistent geschrieben werden können.
- gescheitert** Zustand, nachdem festgestellt worden ist, dass eine weitere Ausführung der Transaktion nicht mehr möglich ist.
- aufgegeben** Zustand nach dem Rollback der Transaktion und Wiederherstellung der DB wie vor Beginn der Transaktion.
- persistent** Zustand nach erfolgreicher Durchführung der Transaktion.

Zustandsübergangsdiagramm



AUTO-COMMIT

In PostgreSQL wird standardmässig jede Anweisung als einzelne Transaktion ausgeführt.

Eine Anweisung, eine Transaktion, aber mehrere Änderungsoperationen

```
UPDATE Konti  
SET Stand = Stand * 1.02
```


SQL

- 1 Mit BEGIN wird eine neue Transaktion gestartet. Damit ist für die folgenden Anweisungen der AUTO-COMMIT Modus ausgeschaltet.
- 2 Den erfolgreichen Abschluss einer Transaktion geben wir mit der Anweisung COMMIT an.
- 3 Wir können eine Transaktion aufgeben mit der Anweisung ROLLBACK. Die Datenbank wird dann in den ursprünglichen Zustand (vor Beginn der Transaktion) zurückgesetzt.

Nach einem COMMIT oder ROLLBACK beginnt eine neue Transaktion. Ohne erneute Angabe des Schlüsselwortes BEGIN ist diese neue Transaktion wieder im AUTO-COMMIT Modus.

```
BEGIN;  
UPDATE Konti SET Stand = Stand - 50 WHERE KId = 'A';  
UPDATE Konti SET Stand = Stand + 50 WHERE KId = 'B';  
COMMIT;
```

Gegenseitige Fremdschlüssel

```
CREATE TABLE Autos (  
    AutoId integer PRIMARY KEY,  
    PaarId integer UNIQUE );
```

```
CREATE TABLE Personen (  
    PersId integer PRIMARY KEY,  
    PaarId integer UNIQUE  
    REFERENCES Autos(PaarId) DEFERRABLE );
```

```
ALTER TABLE Autos ADD  
    FOREIGN KEY (PaarId)  
    REFERENCES Personen(PaarId) DEFERRABLE
```

Aufgeschobene Constraints

```
BEGIN;  
SET CONSTRAINTS ALL DEFERRED;  
INSERT INTO Autos VALUES (1,3);  
INSERT INTO Personen VALUES (2,3);  
COMMIT;
```

Dirty Reads

Eine Abfrage innerhalb einer Transaktion liefert das Ergebnis einer anderen Transaktion, die noch nicht persistent ist. Dies kann zu zwei Problemen führen:

- 1 Das Ergebnis kann inkonsistent sein.
- 2 Die schreibende Transaktion könnte abgebrochen werden. Dann würde das Ergebnis nicht mehr existieren (bzw. hätte gar nie existiert).

Folgende verzahnte Ausführung zweier Transaktionen zeigt das Problem.

Wir verwenden wiederum die abstrakten `READ(X)` und `WRITE(X)`

Operationen:

T1	T2
	READ(A)
	...
	WRITE(A)
READ(A)	
...	
	ROLLBACK

Non-repeatable Reads

Innerhalb einer Transaktion wird eine Abfrage wiederholt und erhält beim zweiten Mal ein anderes Resultat.

Die beiden `READ(A)` Anweisungen in T1 liefern verschiedene Resultate, obwohl sie nur committete Daten lesen.

T1	T2
<code>READ(A)</code>	
	<code>WRITE(A)</code>
	<code>WRITE(B)</code>
	<code>COMMIT</code>
<code>READ(B)</code>	
<code>READ(A)</code>	
<code>...</code>	

Phantome

Phantome treten auf, wenn innerhalb einer Transaktion eine Abfrage mit einer identischen Selektionsbedingung wiederholt wird und bei der zweiten Abfrage eine andere Menge von Tupeln selektiert wird.

```
T1
SELECT COUNT(*)
FROM Konti;
```

```
SELECT AVG(Stand)
FROM Konti;
```

```
T2
INSERT INTO Konti VALUES ('C', 10);
COMMIT;
```

Isolationsgrade gemäss SQL Standard

Isolationsgrad	Dirty R.	Non-repeatable R.	Phantome
READ UNCOMMITTED	Ja	Ja	Ja
READ COMMITTED	Nein	Ja	Ja
REPEATABLE READ	Nein	Nein	Ja
SERIALIZABLE	Nein	Nein	Nein

SET TRANSACTION ISOLATION LEVEL

```
BEGIN;  
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SELECT COUNT(*) FROM Konti;  
SELECT AVG(Stand) FROM Konti;  
COMMIT;
```

Der Isolationsgrad muss dabei zwischen der BEGIN Anweisung und der ersten Anweisung der Transaktion gesetzt werden.

PostgreSQL unterstützt den Isolationsgrad READ UNCOMMITTED nicht. Falls dieser Grad gesetzt wird, so wird intern READ COMMITTED verwendet.

Der SQL Standard definiert SERIALIZABLE als Default Wert für den Isolationsgrad. In PostgreSQL wird jedoch üblicherweise READ COMMITTED als Default verwendet.

Multiversion Concurrency Control Architektur MVCC

Grundidee: jeder Benutzer sieht einen eigenen *Snapshot* der Datenbank zu einem bestimmten Zeitpunkt.

Die MVCC Architektur stellt sicher, dass

- 1 ein Lesezugriff niemals einen parallelen Schreibzugriff blockiert und
- 2 umgekehrt ein Lesezugriff niemals auf einen parallelen Schreibzugriff warten muss.

Implementierung

Jede Transaktion hat Transaktionsnummer Xid .

Zwei interne Attribute $Xmin$ und $Xmax$ bei jeder Tabelle:

- 1 $Xmin$: Xid der Trx , welche diese Version des Tupels erzeugt hat.
- 2 $Xmax$: Xid der Trx , welche diese Version als ungültig erklärt hat.
Enthält `Null`, falls diese Version des Tupels noch gültig.

Ein Transaktion mit Transaktionsnummer Xid sieht ein Tupel t , falls gilt:

$$t[Xmin] \leq Xid \text{ und } (t[Xmax] \text{ IS NULL oder } t[Xmax] > Xid) .$$

Das heisst, das Tupel t wurde vor (oder in) der Transaktion Xid erzeugt und er wurde aus Sicht der Transaktion Xid noch nicht gelöscht.

Intern, aber sichtbar

```
SELECT txid_current()
```

```
SELECT *, Xmin, Xmax  
FROM Konti
```

Bemerkung: Diese Anweisung zeigt nur die sichtbaren Tupel.

Beispiel

Wir starten mit Transaktionsnummer 1

```
INSERT INTO Konti VALUES ('A', 1000)
```

Konti

KId	Stand	Xmin	Xmax
-----	-------	------	------

A	1000	1	-
---	------	---	---

```
DELETE FROM Konti WHERE KId='A'
```

Konti

KId	Stand	Xmin	Xmax
-----	-------	------	------

A	1000	1	2
---	------	---	---

Beispiel 2

Konti			
KId	Stand	Xmin	Xmax
A	2000	3	-

Mit Transaktionsnummer 4

```
UPDATE Konti  
SET Stand=3000  
WHERE KId='A'
```

Konti			
KId	Stand	Xmin	Xmax
A	2000	3	4
A	3000	4	-

Rollback

Ein Rollback setzt nicht `Xmax` auf `Null` zurück, sondern merkt sich, dass die Transaktion abgebrochen wurde.

Interne Datenstruktur `pg_clog` enthält Status jeder Transaktion: 0 (in Bearbeitung), 1 (abgebrochen) und 2 (committed).

<u>pg_clog</u>	
<u>XId</u>	<u>Status</u>
3	2
4	2

ROLLBACK Beispiel

Transaktion mit Xid 5:

```
BEGIN;  
DELETE FROM Konti WHERE KId='A';  
ROLLBACK;
```

Konti

KId	Stand	Xmin	Xmax
A	2000	3	4
A	3000	4	5

pg_clog

XId	Status
3	2
4	2
5	1

Sichtbarkeitsbedingung

T: aktuelle Transaktion

t: Tupel, dessen Sichtbarkeit wir feststellen wollen

<code>(Xmin == my-transaction &&</code>	eingefügt in T
<code> (Xmax is null </code>	t wurde nicht gelöscht oder
<code> Xmax != my-transaction))</code>	t wurde (nach T) gelöscht
<code> </code>	oder
<code>(Xmin is committed &&</code>	Einfügen von t ist committed
<code> (Xmax is null </code>	t wurde nicht gelöscht oder
<code> (Xmax != my-transaction &&</code>	t wurde gelöscht von T' aber
<code> Xmax is not committed)))</code>	T' ist nicht committed

Read Committed

Dieser Isolationslevel verlangt, dass keine Dirty Reads auftreten.

Vor jeder SQL Anweisung wird ein aktueller Snapshot erzeugt und die Anweisung wird dann mit den Daten dieses Snapshots ausgeführt. Die Sichtbarkeitsbedingung garantiert, dass jeder Snapshot nur Daten enthält, welche bereits committed wurden.

Die MVCC Architektur stellt sicher, dass Lese- und Schreibzugriffe sich nicht gegenseitig blockieren können. Konflikte zwischen parallelen Schreibzugriffen lassen sich aber prinzipiell nicht vermeiden.

Beispiel

```
T1
BEGIN;
```

```
UPDATE Konti
SET Stand = Stand*2;
```

```
COMMIT;
```

```
T2
```

```
BEGIN;
```

```
UPDATE Konti
SET Stand = Stand*3;
```

```
COMMIT;
```

Vor Beginn der beiden Transaktionen hat das Konto A den Stand 1000 hat.
Nach der UPDATE Anweisung der Transaktion 1:

<u>Konti</u>			
<u>KId</u>	<u>Stand</u>	<u>Xmin</u>	<u>Xmax</u>
A	1000	0	1
A	2000	1	-

<u>pg_clog</u>	
<u>XId</u>	<u>Status</u>
0	2
1	0

Konsequenzen

Gemäss der Sichtbarkeitsbedingung sieht also Transaktion 2 für Konto A den Stand 1000.

Es bedeutet, dass die Update Operation (noch) nicht ausgeführt werden darf. Es würde nämlich ein neuer Stand 3000 gesetzt, was nicht korrekt ist, falls die erste Transaktion erfolgreich abschliesst.

In dieser Situation muss mit der Ausführung der UPDATE Operation der Transaktion 2 gewartet werden, bis die erste Transaktion abgeschlossen ist (entweder mit COMMIT oder ROLLBACK).

Betrachten wir ein Tupel mit einem X_{max} Wert von einer Transaktion die *in Bearbeitung* ist. Wir können diesen X_{max} Wert somit als Schreibsperre auf dem Tupel auffassen.

Mit diesem Mechanismus werden alle Tupel, welche in einer Transaktion durch UPDATE oder DELETE Operationen modifiziert werden, implizit für parallele Änderungsoperationen gesperrt.

Repeatable Read

Erstelle zu Beginn der Transaktion einen Snapshot und benutze diesen für die ganze Transaktion.

Dieser Ansatz verhindert auch Phantome.

```
T1  
BEGIN;
```

```
UPDATE Konti  
SET Stand = Stand*2;
```

```
COMMIT;
```

```
T2
```

```
BEGIN;  
SET TRANSACTION ISOLATION LEVEL  
REPEATABLE READ;
```

```
UPDATE Konti  
SET Stand = Stand*3;
```

```
COMMIT;
```

Repeatable Read 2

Wenn die zweite Transaktion das UPDATE ausführen will, stellt sie fest, dass eine Schreibsperre besteht und wartet bis T1 abgeschlossen ist.

Dann Fallunterscheidung ob T1 erfolgreich war oder nicht.

- 1 Erfolgreiches Commit von T1. Wegen des Isolationsgrads von T2 wird der Effekt der ersten Transaktion nie in T2 sichtbar sein. Deshalb kann T2 ihr UPDATE nie ausführen und wird automatisch zurückgesetzt. PostgreSQL erzeugt die Fehlermeldung:

ERROR: could not serialize access due to concurrent update.

- 2 Rollback von T1. Durch das Rollback wird die Schreibsperre aufgehoben. Ausserdem enthält der Snapshot, der zu Beginn von T2 erstellt wurde, die aktuellen Daten, da das UPDATE von T1 rückgängig gemacht wurde. Somit kann T2 mit der Abarbeitung der UPDATE Anweisung fortfahren und muss nicht zurückgesetzt werden.

Im ersten Fall muss erneute Ausführung von T2 vom Benutzer initiiert werden.

Optimismus

Optimistische Strategie:

Wenn zwei Transaktionen nebeneinander initiiert werden, so wird mit der Abarbeitung von beiden begonnen und nur wenn es zu unlösbaren Konflikten kommt, wird eine Transaktion zurückgesetzt.

Pessimistische Strategie:

Zu Beginn einer Transaktion alle Objekte, welche in der Transaktion benötigt werden, sperren. Nur wenn alle Sperren erhältlich waren, wird mit der Abarbeitung der Transaktion begonnen. Es ist dann garantiert, dass die Transaktion ganz durchgeführt werden kann.

Der optimistische Ansatz bietet häufig eine deutlich bessere Performance als der pessimistische Ansatz. Falls nämlich nicht mit Snapshot gearbeitet wird, sondern mit Lese- und Schreibsperren, so können Schreiboperationen parallele Leseanweisungen blockieren, was in PostgreSQL nicht möglich ist.

Repeatable Read gibt nicht vollständige Isolation

T1

BEGIN;

```
x:= SELECT COUNT(*)  
      FROM Aerzte  
      WHERE HatDienst=TRUE;
```

```
IF x>1 THEN  
  UPDATE Aerzte  
  SET HatDienst=FALSE  
  WHERE Name='Eva';
```

COMMIT;

T2

BEGIN;

```
x:= SELECT COUNT(*)  
      FROM Aerzte  
      WHERE HatDienst=TRUE;
```

```
IF x>1 THEN  
  UPDATE Aerzte  
  SET HatDienst=FALSE  
  WHERE Name='Tom';  
COMMIT;
```

Beispiel

Wir nehmen an, dass der Anfangszustand der Tabelle `Aerzte` nur zwei Ärzte Eva und Tom enthält, welche Dienst haben.

Transaktionen im Isolationsgrad `REPEATABLE READ`.

Nachdem beide Transaktionen ihr `COMMIT` aufgeführt haben, enthält die Tabelle `Aerzte` keinen Arzt mehr, der Dienst hat.

Dies obwohl jede Transaktion für sich genommen garantiert, dass noch mindestens ein diensthabender Arzt übrigbleibt.

Keine vollständige Isolation im Isolationsgrad `REPEATABLE READ`

Echte Serialisierbarkeit

Gegeben sei eine parallele Ausführung von Transaktionen T_1, \dots, T_n .

Diese parallele Ausführung heisst *echt serialisierbar*, falls es eine Reihenfolge T_{i_1}, \dots, T_{i_n} dieser Transaktionen gibt, so dass das Resultat der gegebenen parallelen Ausführung gleich ist wie das Resultat der sequentiellen Ausführung T_{i_1}, \dots, T_{i_n} dieser Transaktionen.

Sequentiell heisst, dass sich die Transaktionen zeitlich nicht überlappen (keine Parallelität).

Echte Serialisierbarkeit bedeutet also, dass die Transaktionen einander in keiner Art und Weise beeinflussen und somit vollständig isoliert sind.

Serializable

Um im Isolationsgrad `SERIALIZABLE` echte Serialisierbarkeit zu gewährleisten, verwendet PostgreSQL Snapshots, so wie sie bei `REPEATABLE READ` verwendet werden. Zusätzlich werden noch gewisse Abhängigkeiten berücksichtigt.

wr-Abhängigkeit T1 schreibt einen Wert, welcher später von T2 gelesen wird.

ww-Abhängigkeit T1 schreibt eine Version eines Tupels, welche von T2 durch eine neue Version ersetzt wird.

rw-Gegenabhängigkeit T1 liest eine Version eines Tupels, welche später von T2 durch eine neue Version ersetzt wird.

In allen drei Fällen muss T2 nach T1 ausgeführt werden.

Serialisierbarkeitsgraph

Gerichteter Graph, der wie folgt aufgebaut ist:

- 1 Jede Transaktion entspricht einem Knoten des Graphen.
- 2 Wenn T2 von T1 abhängig ist, so gibt es eine Kante von T1 zu T2.

Theorem

Eine parallele Ausführung von Transaktionen ist serialisierbar, genau dann wenn der entsprechende Serialisierbarkeitsgraph azyklisch ist.

Für die Implementierung echter Serialisierbarkeit in PostgreSQL ist zusätzlich noch folgendes Theorem relevant.

Theorem

Jeder Zyklus in einem Serialisierbarkeitsgraphen enthält eine Sequenz
 $T1 \xrightarrow{rw} T2 \xrightarrow{rw} T3.$

Beachte, dass in diesem Theorem T1 und T3 dieselbe Transaktion bezeichnen können.

Implementation

Korollar

Gegeben sei eine parallele Ausführung von Transaktionen. Diese ist serialisierbar, falls gilt:

der Serialisierbarkeitsgraph enthält keine Sequenz (1)
der Form $T1 \xrightarrow{rw} T2 \xrightarrow{rw} T3$.

Dabei ist Bedingung (1) hinreichend für die Serialisierbarkeit. Sie ist jedoch nicht notwendig.

PostgreSQL macht sich dieses Korollar zunutze, um echte Serialisierbarkeit zu implementieren. Im Isolationslevel `SERIALIZABLE` wird Buch geführt über die Abhängigkeiten zwischen den Transaktionen. Falls zwei aneinanderliegende `rw`-Abhängigkeiten auftreten, so wird eine der beteiligten Transaktionen abgebrochen.

Beispiel

```
T1
BEGIN;
SET TRANSACTION ISOLATION
LEVEL SERIALIZABLE;
```

```
SELECT COUNT(*)
FROM Aerzte
WHERE HatDienst=TRUE;
```

```
UPDATE Aerzte
SET HatDienst=FALSE
WHERE Name='Eva';
```

```
COMMIT;
```

```
T2
BEGIN;
SET TRANSACTION ISOLATION
LEVEL SERIALIZABLE;
```

```
SELECT COUNT(*)
FROM Aerzte
WHERE HatDienst=TRUE;
```

```
UPDATE Aerzte
SET HatDienst=FALSE
WHERE Name='Tom';
COMMIT;
```

Beispiel

Wenn die zweite Transaktion ihre COMMIT Anweisung ausführen will, so erhalten wir folgende Fehlermeldung

```
ERROR: could not serialize access due to read/write dependencies  
among transactions.
```

und T2 wird automatisch zurückgesetzt.

Genau wie im Isolationslevel REPEATABLE READ müssen wir auch hier damit rechnen, dass eine Transaktion ihr COMMIT nicht ausführen kann und ein automatisches Rollback durchgeführt wird.