Mehrbenutzerbetrieb

8

Bestimmte Folgen von Datenbankanweisungen müssen als eine Einheit ausgeführt werden. Eine solche Einheit nennen wir Transaktion. In diesem Kapitel führen wir die ACID Postulate für Transaktionen ein und zeigen wie Transaktionen in SQL behandelt werden.

Theoretisch sollten parallel ausgeführte Transaktionen einander nicht beeinflussen, d. h. sie werden isoliert ausgeführt. In der Praxis wird diese Bedingung aus Performance-Gründen jedoch gelockert. Wir studieren die verschiedenen Isolationsgrade, welche SQL anbietet. Dazu betrachten wir drei Phänomene, die bei der parallelen Transaktionsverarbeitung auftreten können: Dirty Reads, Non-repeatable Reads und Phantom Reads.

Wir zeigen auch, wie PostgreSQL die verschiedenen Isolationsgrade mittels einer Multiversion Concurrency Control (MVCC) Architektur implementiert. Bei diesem Ansatz werden für jedes Tupel mehrere Versionen abgespeichert und eine Sichtbarkeitsbedingung definiert, welche Transaktion welche Version eines Tupels sieht.

8.1 Transaktionen

Eine *Transaktion* ist eine Folge von Datenbankanweisungen, welche zusammen eine *Einheit* bilden. Die Anweisungen einer Transaktion werden entweder vollständig ausgeführt (d. h. alle Anweisungen werden erfolgreich beendet) oder gar nicht. Falls also im Verlauf einer Transaktion eine fehlerhafte Situation auftritt, so werden die bisher erfolgten Anweisungen rückgängig gemacht und es wird der Zustand zu Beginn der Transaktion wieder hergestellt.

Das Standardbeispiel im Zusammenhang mit Transaktionen ist die Überweisung eines Betrags von einem Bankkonto `A` auf ein anderes Bankkonto `B`. Dazu sind zwei Schritte nötig:

T. Studer, *Relationale Datenbanken*,
https://doi.org/10.1007/978-3-662-58976-2_8

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

Es darf in diesem Beispiel nicht vorkommen, dass nur eine der beiden Anweisungen ausgeführt wird. Falls bei der Gutschrift ein Fehler auftritt, so muss die Lastschrift rückgängig gemacht werden. Es werden damit alle Kontostände auf den Zustand zu Beginn der Transaktion zurückgesetzt.

Die Eigenschaften von Transaktionen lassen sich mit den ACID Postulaten zusammenfassen:

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

Um die Eigenschaften von Transaktionen zu studieren, verwenden wir die beiden folgenden 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)
```

An diesem Beispiel wollen wir nun die ACID-Postulate etwas genauer untersuchen.

Atomicity Wir nehmen an, dass zu Beginn der Transaktion die Kontostände von `A` und `B` die Beträge CHF 1000 und CHF 2000 aufweisen. Betrachten wir dann die Transaktion nach der Ausführung von `WRITE(A)` aber vor der Ausführung von `WRITE(B)`, so

ergeben sich die Kontostände von CHF 950 für A und CHF 2000 für B. Damit ist also die Summe von A und B vermindert worden. Dies ist ein inkonsistenter Zustand, der im Datenbanksystem nicht sichtbar werden darf. Er wird aber durch die weiteren Operationen dieser Transaktion wieder in einen konsistenten Zustand überführt.

Dies ist die Grundidee von *Atomicity*: Entweder werden alle Operationen der Transaktion in der Datenbank reflektiert oder keine. Die Datenbank speichert die Daten vor Ausführung der Transaktion, und falls die Transaktion nicht erfolgreich durchgeführt worden ist, werden die alten Daten wieder aktiviert. Die Sicherstellung von Atomicity muss vom Datenbanksystem selbst gewährleistet werden.

Consistency Dieses Prinzip verlangt, dass eine konsistente Datenbank durch eine Transaktion in eine konsistente Datenbank überführt wird. In unserem Beispiel lautet eine Konsistenzbedingung, dass die Summe von A und B durch die Transaktion nicht verändert werden darf. Andernfalls würde Geld vernichtet oder neu erzeugt. Falls die Transaktion schief geht und eine Konsistenzbedingung verletzt wird, so müssen alle Änderungen der Transaktion rückgängig gemacht werden und der Ursprungszustand muss wieder hergestellt werden.

Die Sicherstellung der Konsistenz für eine individuelle Transaktion gehört zur Verantwortung des Anwendungsprogrammierers. Diese Aufgabe kann durch automatische Testverfahren in Zusammenhang mit Integritätsbedingungen unterstützt werden. Dabei dürfen Constraints innerhalb einer Transaktion verletzt sein. Am Ende der Transaktion müssen aber alle Constraints erfüllt sein, siehe dazu auch das Beispiel 8.2.

Isolation Selbst wenn Consistency und Atomicity sichergestellt sind, könnten durch simultane Ausführung weiterer Transaktionen Probleme auftreten. Betrachten wir folgende verschränkte Ausführung von zwei Transaktion, welche beide eine Gutschrift auf Konto A vornehmen.

```
T1              T2
READ(A)
A := A+200
                READ(A)
                A := A+500
WRITE(A)
                WRITE(A)
```

Transaktion T1 nimmt also eine Gutschrift von CHF 200 vor und T2 eine Gutschrift von CHF 500. Nehmen wir an, zu Beginn der beiden Transaktionen betrage der Kontostand CHF 100. Nach Beendigung der beiden Transaktion sollte der Kontostand also CHF 800 betragen. Falls die beiden Transaktionen wie oben angegeben verschränkt ausgeführt werden, so beträgt der Kontostand jedoch nur CHF 600. Das Update von T1 geht durch die parallele Ausführung der beiden Transaktionen verloren.

Das Isolationsprinzip verlangt, dass solche Effekte nicht auftreten dürfen. Eine Möglichkeit dies zu erreichen besteht etwa darin, Transaktionen nur seriell (d. h. hintereinander, ohne Parallelität) ausführen zu lassen; allerdings werden dadurch grosse Nachteile bei der Performance in Kauf genommen. Andere Aspekte und Möglichkeiten im Zusammenhang mit Isolation werden wir später betrachten.

Durability Ist eine Transaktion erfolgreich abgeschlossen worden, so bleiben alle dadurch bewirkten Updates bestehen, selbst wenn nach erfolgreicher Ausführung der Transaktion ein Systemfehler auftritt. Um dies sicherzustellen, werden die durch eine Transaktion verursachten Änderungen vor Beendigung der Transaktion z. B. persistent in den Sekundärspeicher geschrieben.

Aus den obigen Überlegungen sehen wir, dass es verschiedene Zustände für Transaktionen gibt. Eine erfolgreich durchgeführte und abgeschlossene Transaktion heisst *committed*. Ihre Auswirkung auf die Daten kann nicht mehr aufgehoben werden.

Transaktionen, die nicht erfolgreich terminieren, bezeichnen wir als *abgebrochen* (aborted). Aufgrund der Atomicity-Eigenschaft dürfen solche Transaktionen keinen Einfluss auf den Zustand der Datenbank haben. Das bedeutet, dass alle Änderungen, die durch eine abgebrochene Transaktion hervorgerufen worden sind, aufgehoben werden. Man spricht dann vom *Rollback* der entsprechenden Transaktion.

Wir betrachten nun ein einfaches Transaktionsmodell mit den folgenden sechs Zuständen:

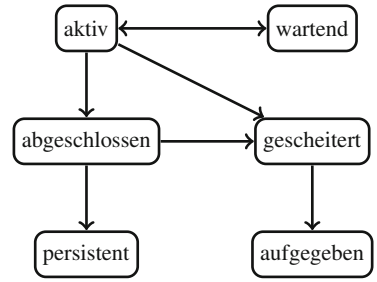
aktiv	Anfangszustand; während die Operationen der Transaktion ausgeführt werden, bleibt sie in diesem Zustand.
wartend	Zustand, in welchem die Transaktion warten muss, bis benötigte Ressourcen (durch andere Transaktionen) freigegeben werden.
abgeschlossen	Zustand nach Ausführung der letzten Operation 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.

Wir erhalten damit das Zustandsübergangsdiagramm in Abb. 8.1.

Wir wollen nun beschreiben, wie man in SQL mehrere Anweisungen zu einer Transaktion zusammenfassen kann. Dazu betrachten wir eine Tabelle `Konti` mit den Attributen `KId` (Konto-Id) und `Stand` (der aktuelle Kontostand).

In PostgreSQL wird standardmässig jede Anweisung als einzelne Transaktion ausgeführt. In diesem Fall wird die Transaktion automatisch abgeschlossen, sobald die Anweisung ausgeführt wurde. Man spricht in diesem Fall von `AUTO-COMMIT`. Dies

Abb. 8.1 Transaktionsmodell



bedeutet, dass eine *Anweisung* als eine Transaktion ausgeführt wird. Diese Anweisung kann jedoch mehrere Änderungsoperationen durchführen. Wir betrachten folgende Anweisung, welche jedem Konto 2 % Zins gutschreibt:

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

Diese Anweisung wird sequentiell abgearbeitet. Die Tabelle `Konti` wird Tupel für Tupel durchgegangen und bei jedem Tupel wird der Zins gutgeschrieben. Tritt nun nach der Hälfte der Tupel ein Fehler auf, so hat das zur Folge, dass alle Änderungen rückgängig gemacht werden. In diesem Fall müsste dann die ganze Transaktion wiederholt werden. Erst nach dem erfolgreichen Update des letzten Tupels wird die Transaktion automatisch (wegen dem `AUTO-COMMIT` Modus) abgeschlossen und die geänderten Daten persistent gespeichert.

Um mit Transaktionen zu arbeiten, welche aus mehreren Anweisungen bestehen, bietet SQL folgende Anweisungen an:

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.

Beispiel 8.1. Für eine Überweisung von CHF 50 von Konto `A` auf Konto `B` verwenden wir also folgenden SQL Anweisungen:

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

Nur so werden die beiden Updates als eine Transaktion ausgeführt.

Beispiel 8.2. In Beispiel 6.2 haben wir gesehen, wie wir zwei Tabellen mit gegenseitigen Fremdschlüsseln erzeugen können. Sollen nun Tupel, bei denen `PaarId` nicht `Null` ist, in diese Tabellen eingefügt werden, so müssen die Einträge eines Paares in beide Tabellen sozusagen gleichzeitig (d. h. als eine Einheit) erfolgen. Dies können wir mit einer Transaktion durch folgende Anweisungen erreichen:

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

Die Anweisung

```
SET CONSTRAINTS ALL DEFERRED
```

sagt, dass alle Constraints, welche `DEFERRABLE` sind, erst am Ende der Transaktion überprüft werden sollen. Damit müssen die references Constraints erst am Ende der Transaktion erfüllt sein. Defaultmässig müssen alle Constraints nach jeder Anweisung gelten, d. h. auch innerhalb von Transaktionen. Falls ein anderes Verhalten gewünscht wird, so muss das wie in diesem Beispiel angegeben werden.

8.2 Phantome und Inkonsistenzen

Gemäss den ACID Postulaten soll jede Transaktion isoliert ablaufen. Wenn jedoch mehrere Transaktionen isoliert durchgeführt werden müssen, so können sie schlecht parallelisiert werden. Eine strenge Auslegung dieses Isolationsbegriffs hat also grosse Auswirkungen auf die Performance des Datenbanksystems. Aus diesem Grund definiert der SQL Standard vier verschiedene Isolationsgrade. Um diese zu beschreiben, betrachten wir zuerst drei Phänomene, welche im Mehrbenutzerbetrieb eines Datenbanksystems auftreten können.

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 ergibt beim zweiten Mal ein anderes Resultat. Das erste Resultat ist also nicht wiederholbar. Dies kann auftreten, wenn zwischen den beiden identischen Abfragen eine zweite Transaktion Änderungen am Datenbestand durchführt und persistent schreibt. Im folgenden Ausführungsplan werden die beiden `READ(A)` Anweisungen in `T1` verschiedene Resultate liefern, obwohl sie nur Daten lesen, welche mit `COMMIT` persistent geschrieben wurden.

```
T1          T2
READ(A)
            WRITE(A)
            WRITE(B)
            COMMIT
READ(B)
READ(A)
...
```

Phantom Reads

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. Dies kann beispielsweise vorkommen, wenn zwischen den beiden Abfragen eine zweite Transaktion neue Tupel, welche die Selektionsbedingung erfüllen, eingefügt und committed hat. Die Datensätze, die in der zweiten Abfrage neu (aus dem Nichts) dazukommen, heissen *Phantome*.

Wir betrachten folgenden Ausführungsplan, in welchem die Transaktion `T1` die Anzahl der Konti und den durchschnittlichen Kontostand berechnet.

```
T1                    T2
SELECT COUNT(*)
FROM Konti;
                      INSERT INTO Konti VALUES ('C', 10);
                      COMMIT;
SELECT AVG(Stand)
FROM Konti;
```

Die beiden Berechnungen in der Transaktion `T1` beruhen auf zwei verschiedenen Zuständen der Tabelle `Konti`. Wenn wir also die Summe aller Kontostände berechnen als das Produkt der Anzahl Konti mit dem durchschnittlichen Kontostand, so erhalten wir eine Zahl, welche zu keinem Zeitpunkt der tatsächlichen Summe der Kontostände entsprochen hat.

Der Isolationsgrad einer Transaktion gibt an, welche von diesen drei Phänomenen das Datenbanksystem zulassen und welche es verhindern soll. Tab. 8.1 spezifiziert die SQL-Isolationsgrade. Dabei bedeutet *Ja*, dass das Phänomen möglich ist; *Nein* heisst, dass es verhindert wird.

In PostgreSQL kann der Isolationsgrad mit folgender Anweisung gesetzt werden:

```
SET TRANSACTION ISOLATION LEVEL < Isolationsgrad >
```

Der Isolationsgrad muss dabei zwischen der `BEGIN` Anweisung und der ersten Anweisung der Transaktion gesetzt werden. Für die Transaktion `T1` im obigen Beispiel verwenden wir also:

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

PostgreSQL unterstützt den Isolationsgrad `READ UNCOMMITTED` nicht. Falls dieser Grad gesetzt wird, so wird intern `READ COMMITTED` verwendet. Nach dem SQL Standard ist dies zulässig, da die Isolationsgrade nur Minimalgarantien dafür sind, welche Phänomene nicht auftreten dürfen.

Tab. 8.1 SQL Isolationsgrade

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

Anmerkung 8.3. Der SQL Standard definiert `SERIALIZABLE` als Default Wert für den Isolationsgrad. In PostgreSQL wird jedoch üblicherweise `READ COMMITTED` als Default verwendet.

8.3 Multiversion Concurrency Control Architektur

Die Grundidee hinter einer Multiversion Concurrency Control (MVCC) Architektur ist es, dass jeder Benutzer der Datenbank einen eigenen *Snapshot* der Datenbank zu einem bestimmten Zeitpunkt sieht. Dazu werden intern mehrere Versionen eines Objektes (z. B. eines Tupels) gehalten, welche durch fortlaufend erhöhte Transaktionsnummern voneinander unterschieden werden. 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.

Um MVCC zu implementieren, fügt PostgreSQL zu jeder Tabelle zwei interne Attribute `Xmin` und `Xmax` hinzu. Für jedes Tupel enthält `Xmin` die Transaktionsnummer derjenigen Transaktion, welche diese Version des Tupels erzeugt hat. Das Attribut `Xmax` enthält die Nummer derjenigen Transaktion, welche diese Version des Tupels als ungültig markiert hat (wegen einer Update oder Delete Operation). `Xmax` enthält den Wert 0, falls diese Version des Tupels noch gültig ist.

Wenn ein Tupel durch eine Update Operation mit der Transaktionsnummer `Xid` geändert wird, so werden nicht die alten Daten durch die neuen überschrieben, sondern in der aktuellen Version des Tupels wird `Xmax` auf `Xid` gesetzt und es wird eine neue Version des Tupels (mit dem Resultat der Update Operation) erzeugt. In dieser neuen Version hat `Xmin` den Wert `Xid` und `Xmax` ist 0.

Eine Transaktion mit Transaktionsnummer `Xid` sieht also ein Tupel `t`, falls gilt:

$$\texttt{t[Xmin]} \leq \texttt{Xid} \quad \text{und} \quad \big(\texttt{t[Xmax]} = 0 \quad \text{oder} \quad \texttt{t[Xmax]} > \texttt{Xid}\big). \tag{8.1}$$

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

Die Attribute `Xmin` und `Xmax` dienen zwar internen Zwecken von PostgreSQL, man kann sich diese Werte aber dennoch anzeigen lassen (allerdings nur für sichtbare Tupel gemäss (8.1)). Die entsprechende Anweisung lautet:

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

Auch die aktuelle Transaktionsnummer kann man sich mit einem `SELECT` Statement anzeigen lassen:

```
SELECT txid_current()
```

Wir betrachten nun unsere `Konti` Tabelle und nehmen an, das folgende Statement werde mit der Transaktionsnummer 1 ausgeführt:

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

Wir erhalten:

Konti

KId	Stand	Xmin	Xmax
A	1000	1	0

Eine Delete Operation entfernt nicht Tupel aus einer Tabelle, sondern setzt nur die entsprechenden `Xmax` Werte. Mit der Anweisung

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

erhalten wir (die Transaktionsnummer der `DELETE` Anweisung war 2)

Konti

KId	Stand	Xmin	Xmax
A	1000	1	2

Wir können eine Update Operation auffassen als eine Kombination einer Delete und einer Insert Operation. Wir betrachten folgende `Konti` Tabelle:

Konti

KId	Stand	Xmin	Xmax
A	2000	3	0

Wir führen nun folgende Update Operation mit der Transaktionsnummer 4 aus:

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

Damit erhalten wir:

Konti			
KId	**Stand**	**Xmin**	**Xmax**
A	2000	3	4
A	3000	4	0

Wir betrachten nun den Fall, dass eine Transaktion mit Nummer `Xid` eine Delete Operation ausführt und danach abgebrochen wird. Wie wir gesehen haben, setzt die Delete Operation die `Xmax` Werte der zu löschenden Tupel auf `Xid`. Das Rollback wird nun nicht diese Werte auf 0 zurücksetzen, sondern merkt sich nur, dass die Transaktion `Xid` abgebrochen wurde. Dazu führt PostgreSQL eine interne Datenstruktur `pg_clog` (*clog* steht für *commit log*), welche den Status jeder Transaktion enthält. Dabei sind folgende Stati möglich: 0 (in Bearbeitung), 1 (abgebrochen) und 2 (committed).

Nach dem obigen Update Statement hat `pg_clog` folgenden Inhalt:

pg_clog	
XId	**Status**
3	2
4	2

Wir betrachten nun folgende Transaktion mit `Xid` 5:

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

Damit erhalten wir:

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

und

pg_clog	
XId	**Status**
3	2
4	2
5	1

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

Abb. 8.2 Sichtbarkeitsbedingung in PostgreSQL

Die Formulierung der Sichtbarkeitsbedingung in (8.1) ist also noch zu simpel, da sie den Transaktionsstatus nicht berücksichtigt. Abb. 8.2 zeigt den Kommentar aus dem PostgreSQL Source Code,[1] welcher die Implementierung der Sichtbarkeitsbedingung beschreibt. Wir bezeichnen mit `T` die aktuelle Transaktion und mit `t` das Tupel, dessen Sichtbarkeit wir feststellen wollen. Achtung: Eigentlich sollte es im Kommentar `Xmax == 0` anstelle von `Xmax is null` heissen. Wie oben beschrieben, verwendet Postgre SQL den `Xmax` Wert 0, um anzuzeigen, dass ein Tupel noch gültig ist.

Der MVCC Ansatz führt dazu, dass PostgreSQL sehr viele Datensätze speichert, welche eigentlich obsolet sind, bspw. weil diese als gelöscht markiert wurden. Um diese Datensätze physikalisch zu löschen gibt es in PostgreSQL den Befehl `VACUUM`, welcher periodisch ausgeführt werden muss.

Der `VACUUM` Befehl kümmert sich auch um das korrekte Handling der Transaktionsnummern. Diese werden nämlich als 32 bit Zahl gespeichert, d. h. es gibt etwa 4 Milliarden verschiedene Transaktionsnummern. Wenn es nun ganz alte Tupel gibt, so können sich die Transaktionsnummern wiederholen. Dies kann den Effekt haben, dass Tupel, welche sehr alt sind, plötzlich aus der Zukunft zu kommen scheinen. `VACUUM` markiert diese alten Tupel, so dass sie beim Sichtbarkeitscheck richtig behandelt werden.

8.4 Implementierung der Isolationsgrade

In diesem Abschnitt werden wir zeigen, wie die verschiedenen Isolationsgrade mit Hilfe der MVCC Architektur in PostgreSQL realisiert werden. Der wichtigste Begriff dabei ist der eines Snapshots. Dabei wird mit Hilfe der Sichtbarkeitsbedingung (Abb. 8.2) ein Snapshot der Datenbankinstanz zu einem bestimmten Zeitpunkt erstellt. Auf Basis dieses Snapshots (und nicht mit den aktuellen Daten der Instanz) werden dann SQL Anweisungen (`INSERT`, `SELECT`, `UPDATE` und `DELETE`) bearbeitet. Je nach Isolationsgrad werden die Snapshots zu unterschiedlichen Zeiten erstellt und unterschiedlich lange verwendet.

[1] aus `src/backend/utils/time/tqual.c`.

Read Committed

Dieser Isolationslevel verlangt, dass keine Dirty Reads auftreten, d. h. es werden keine Daten gelesen, die noch nicht committed sind. PostgreSQL implementiert diesen Level dadurch, dass vor jeder SQL Anweisung ein aktueller Snapshot erzeugt wird und die Anweisung dann mit den Daten dieses Snapshots ausgeführt wird. Die Sichtbarkeitsbedingung garantiert, dass jeder Snapshot nur Daten enthält, welche bereits committed wurden. Damit ist sichergestellt, dass es keine Dirty Reads geben kann.

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

```
T2                          T3
BEGIN;
                            BEGIN;
UPDATE Konti
SET Stand = Stand*2;
                            UPDATE Konti
                            SET Stand = Stand*3;
COMMIT;
                            COMMIT;
```

Wir nehmen an, dass vor Beginn der beiden Transaktionen das Konto `A` den `Stand` 1000 hat. Nach der `UPDATE` Anweisung der Transaktion 2 haben wir folgende Situation. Zur Erinnerung, wir haben folgende Transaktionsstati: 0 (in Bearbeitung), 1 (abgebrochen) und 2 (committed).

Konti

KId	Stand	Xmin	Xmax
A	1000	1	2
A	2000	2	0

pg_clog

XId	Status
1	2
2	0

Gemäss der Sichtbarkeitsbedingung (Abb. 8.2) sieht also Transaktion 3 für Konto `A` den `Stand` 1000. Dies ist richtig, weil ja keine Dirty Reads auftreten sollen. Es bedeutet aber auch, 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 3 gewartet werden, bis die erste Transaktion abgeschlossen ist (entweder mit COMMIT oder ROLLBACK). Dann erst wird die Abarbeitung der zweiten Transaktion fortgesetzt. Natürlich muss dann für die UPDATE Anweisung ein aktueller Snapshot erstellt werden, welcher die Tupel, die neu committed wurden, enthält.

Betrachten wir ein Tupel mit einem Xmax Wert von einer Transaktion die *in Bearbeitung* ist. Wir können diesen Xmax Wert somit als Schreibsperre auf dem Tupel auffassen. In der Tat werden damit alle Transaktionen, welche dieses Tupel ändern wollen, angehalten, bis die Schreibsperre aufgehoben ist. Das heisst, bis die sperrende Transaktion abgeschlossen wurde.

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

Repeatable Read

In diesem Isolationslevel wird zu Beginn der Transaktion ein Snapshot erstellt. Mit diesem Snapshot wird dann die ganze Transaktion abgearbeitet. Es wird also nicht für jede Operation ein neuer, aktueller Snapshot erzeugt, sondern alle Operationen der Transaktion gehen vom ursprünglichen Snapshot aus. Es sei T eine Transaktion mit Isolationslevel REPEATABLE READ. Dann bleiben alle Änderungen, welche parallel zur laufenden Transaktion T committed werden, unsichtbar innerhalb von T.

Da immer vom ursprünglichen Snapshot ausgegangen wird (und damit parallele Änderungen unsichtbar sind), ist garantiert, dass Non-repeatable Reads nicht auftreten können. Zwei identische Leseanweisungen werden also dasselbe Resultat liefern. Mit dem Ansatz, alle Anweisungen auf demselben Snapshot auszuführen, wird auch verhindert, dass Phantome auftreten können.

Wie wir oben gesehen haben, können parallele UPDATE Operationen im Isolationslevel READ COMMITTED dazu führen, dass eine Transaktion warten muss, bis die andere abgeschlossen ist. Wenn wir REPEATABLE READ verlangen, dann ist dies nicht unbedingt ausreichend. Wir betrachten folgende Situation:

```
T1                          T2
BEGIN;
                            BEGIN;
                            SET TRANSACTION ISOLATION LEVEL
                                REPEATABLE READ;
UPDATE Konti
SET Stand = Stand*2;
                            UPDATE Konti
                            SET Stand = Stand*3;
COMMIT;
                            COMMIT;
```

Wenn die zweite Transaktion das `UPDATE` ausführen will, dann stellt sie fest, dass eine Schreibsperre besteht. Sie wird dann warten, bis `T1` abgeschlossen ist. Nun müssen wir unterscheiden, 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 (trotz des erfolgreichen `COMMIT`s von `T1`). Deshalb kann `T2` ihr `UPDATE` nie ausführen und wird automatisch zurückgesetzt (mit einem Rollback). PostgreSQL erzeugt in diesem Fall 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, wenn ein automatisches Rollback von `T2` durchgeführt wird, so heisst das nur, dass `T2` nicht parallel mit `T1` ausgeführt werden konnte. Jetzt, wo `T1` abgeschlossen ist, kann `T2` nochmals neu gestartet werden. Dies geschieht jedoch nicht automatisch, sondern muss vom Benutzer initiiert werden. Für diese neue Transaktion wird auch ein neuer Snapshot erstellt, welcher jetzt die Änderungen von `T1` enthält. Damit kann diese neue Transaktion die `UPDATE` Operation wie gewünscht ausführen.

Wir sehen, dass PostgreSQL eine *optimistische* Strategie bezüglich parallelen Transaktionen verfolgt. 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.

Eine *pessimistische* Strategie, welche von anderen Datenbanksystemen implementiert wird, ist es, zu Beginn einer Transaktion alle Objekte, welche in der Transaktion benötigt werden, zu 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. Dafür muss sie am Anfang so lange warten, bis alle Sperren verfügbar sind.

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

Serializable

Wie wir oben gesehen haben, können mit dem Isolationsgrad `REPEATABLE READ` in PostgreSQL Phantome ausgeschlossen werden. Somit erfüllt dieser Level auch

die Bedingung an SERIALIZABLE des SQL Standards. Tatsächlich wurde die oben beschriebene Methode in älteren PostgreSQL Versionen zur Implementierung von SERIALIZABLE benutzt. Seit Version 9.1 gibt es jedoch eine neue Implementierung für den Isolationsgrad SERIALIZABLE, welcher *echte Serialisierbarkeit* garantiert. Folgendes Beispiel illustriert das Problem der echten Serialisierbarkeit.

Beispiel 8.4. Wir gehen aus von einer Tabelle Aerzte. Der Primärschlüssel dieser Tabelle ist das Attribut Name. Daneben gibt es ein weiteres Attribut HatDienst, das speichert, ob der entsprechende Arzt zur Zeit Dienst hat oder nicht. Wir nehmen an, dass der Anfangszustand der Tabelle Aerzte nur zwei Ärzte Eva und Tom enthält, welche Dienst haben. Wir betrachten nun zwei Transaktionen, die im Isolationsgrad REPEATABLE READ wie angegeben parallel ablaufen.

```
T1                              T2
BEGIN;
                                BEGIN;
x:= SELECT COUNT(*)
    FROM Aerzte
    WHERE HatDienst=TRUE;
                               x:= SELECT COUNT(*)
                                    FROM Aerzte
                                    WHERE HatDienst=TRUE;
IF x>1 THEN
  UPDATE Aerzte
  SET HatDienst=FALSE
  WHERE Name='Eva';
                               IF x>1 THEN
                                 UPDATE Aerzte
                                 SET HatDienst=FALSE
                                 WHERE Name='Tom';
COMMIT;
                               COMMIT;
```

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.

Wir hatten gesehen, dass REPEATABLE READ in PostgreSQL bereits garantiert, dass weder Dirty Reads, Non-repeatable Reads noch Phantome auftreten können. Das obige Beispiel zeigt, dass es dennoch möglich ist, dass bei paralleler Ausführung zweier Transaktionen Inkonsistenzen auftreten können. Der Grund ist, dass die Snapshots im

Transaktionslevel `REPEATABLE READ` keine vollständige Isolierung der Transaktionen garantieren.

Gegeben sei eine parallele Ausführung von Transaktionen $T_1, \ldots, T_n$. Diese parallele Ausführung heisst *echt serialisierbar*, falls es eine Reihenfolge $T_{i_1}, \ldots, T_{i_n}$ dieser Transaktionen gibt, so dass das Resultat der gegebenen parallelen Ausführung gleich ist wie das Resultat der sequentiellen Ausführung $T_{i_1}, \ldots, T_{i_n}$ dieser Transaktionen. Sequentiell heisst, dass sich die Transaktionen zeitlich nicht überlappen (keine Parallelität).

Echte Serialisierbarkeit bedeutet also, dass die Transaktionen sich in keiner Art und Weise beeinflussen und somit vollständig isoliert sind. Betrachten wir Transaktionen $T_1, \ldots, T_n$ und nehmen an, dass jede einzelne Transaktion T_i $(1 \leq i \leq n)$ korrekt ist (eine Integritätsbedingung erfüllt). Damit ist auch jede serielle Ausführung dieser Transaktionen korrekt (erfüllt die Integritätsbedingung). Wenn also eine parallele Ausführung von $T_1, \ldots, T_n$ serialisierbar ist, so ist auch diese parallele Ausführung korrekt (erfüllt die Integritätsbedingung).

Um im Isolationsgrad `SERIALIZABLE` echte Serialisierbarkeit zu gewährleisten, verwendet PostgreSQL Snapshots, so wie sie bei `REPEATABLE READ` verwendet werden. Zusätzlich wird noch Buch geführt über gewisse Abhängigkeiten zwischen den Transaktionen. Im Allgemeinen gibt es drei Möglichkeiten, wie eine Transaktion `T2` von einer anderen Transaktion `T1` abhängig sein kann:

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.

Mit Hilfe dieser Abhängigkeiten kann man nun den sogenannten *Serialisierbarkeitsgraphen* definieren. Dies ist ein 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`.

Für den Ablauf aus Beispiel 8.4 erhalten wir also den Graphen aus Abb. 8.3.

Es gibt einen wichtigen Zusammenhang zwischen dem so definierten Graphen und dem Konzept der Serialisierbarkeit, welcher in folgendem Theorem ausformuliert ist.

Theorem 8.5. *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.

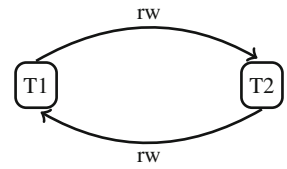


Abb. 8.3 Serialisierbarkeitsgraph zu Beispiel 8.4

Theorem 8.6. *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, vergleiche Abb. 8.3. Wenn wir diese beiden Theoreme kombinieren, so erhalten wir folgendes Resultat.

Korollar 8.7. *Gegeben sei eine parallele Ausführung von Transaktionen. Diese ist serialisierbar, falls gilt:*

$$\begin{array}{l} \textit{der Serialisierbarkeitsgraph enthält keine Sequenz} \\ \textit{der Form } T1 \xrightarrow{rw} T2 \xrightarrow{rw} T3. \end{array} \tag{8.2}$$

Dabei ist (8.2) hinreichend aber nicht notwendig für die Serialisierbarkeit.

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. Damit ist (8.2) immer erfüllt und mit Korollar 8.7 ist garantiert, dass der parallele Ablauf der Transaktionen serialisierbar ist.

Da (8.2) nur hinreichend aber nicht notwendig für die Serialisierbarkeit ist, kann es sein, dass unnötige Rollbacks ausgeführt werden. Der Vorteil dieses Verfahrens ist aber, dass es deutlich effizienter ist, als den ganzen Serialisierbarkeitsgraphen aufzubauen und auf Zyklen zu testen. Insbesondere müssen wr- und ww-Abhängigkeiten nicht beachtet werden.

Dies ergibt folgendes Verhalten. Wir nehmen an, wir haben die Ausgangssituation aus Beispiel 8.4 und betrachten den folgenden Ablauf:

```
T1                                T2
BEGIN;
                                  BEGIN;
SET TRANSACTION ISOLATION
LEVEL SERIALIZABLE;
                                  SET TRANSACTION ISOLATION
                                  LEVEL SERIALIZABLE;
```

```
SELECT COUNT(*)
FROM Aerzte
WHERE HatDienst=TRUE;
                                SELECT COUNT(*)
                                FROM Aerzte
                                WHERE HatDienst=TRUE;
UPDATE Aerzte
SET HatDienst=FALSE
WHERE Name='Eva';
                                UPDATE Aerzte
                                SET HatDienst=FALSE
                                WHERE Name='Tom';
COMMIT;
                                COMMIT;
```

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 die Transaktion `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. Diese Möglichkeit muss bei der Anwendungsprogrammierung berücksichtigt und angemessen behandelt werden.

Weiterführende Literatur[2]

1. ANSI: Database language SQL (2011). Dokument X3.135-2011
2. Hartwig, J.: PostgreSQL professionell und praxisnah. Addison-Wesley, München/Boston (2001)
3. Ports, D.R.K., Grittner, K.: Serializable snapshot isolation in postgresql. Proc. VLDB Endow. **5**(12), 1850–1861 (2012). https://doi.org/10.14778/2367502.2367523
4. The PostgreSQL Global Development Group: Postgresql documentation, concurrency control (2018). https://www.postgresql.org/docs/current/static/mvcc.html. Zugegriffen am 11.06.2019

[2]Im SQL Standard [1] werden vier verschiedene Isolationsgrade definiert. Die PostgreSQL Dokumentation [4] beschreibt, wie diese Isolationsgrade in PostgreSQL mittels der Multiversion Concurrency Control Architektur realisiert werden. Serializable Snapshot Isolation ist ein relativ neues Feature von PostgreSQL. Die Funktionsweise und Implementierung dieses Isolationsgrades wird detailiert ausgeführt in [3]. Weitere Beispiele zur parallelen Transaktionsverarbeitung unter verschiedenen Isolationsgraden finden sich in [2].