Isolationslevel in SQL

Zu den ACID-Eigenschaften von Transaktionen gehört auch das „I“, also Isolation. Streng genommen versteht man unter Isolation *Serialisierbarkeit*, d.h. dass der verschränkte Ablauf mehrerer Transaktionen äquivalent ist zu einem *seriellen* Ablauf, in dem jede Transaktion alle ihre Teilschritte durchführt, ehe eine andere Transaktion am Zuge ist.

Tatsächlich gibt es jedoch Situationen, in denen Transaktionen durchaus „sehen“ möchten, ob und wie andere Transaktionen parallel Daten verändern. Deshalb sieht der SQL-Standard verschiedene Level der Isolation zwischen Transaktionen vor und es ist dann die Sache der Programmierung der jeweiligen Transaktion, dasjenige Isolationslevel zu wählen, das für die Anwendung geboten ist.

Phänomene bei verschränkten Transaktionen

Werden Transaktionen nicht seriell, sondern verschränkt durchgeführt, dann können verschiedene *Phänomene* (man könnte auch sagen: *Anomalien*) der gegenseitigen Beeinflussung von Transaktionen auftreten. Der SQL-Standard definiert solche Phänomene.

Wir gehen in den folgenden Beispielen davon aus, dass unser Datenbankschema eine Tabelle `Konto` mit den Spalten `KtoNr` und `Saldo` hat. Im `Saldo` wird das Guthaben auf dem Konto mit der Kontonummer `KtoNr` gespeichert.

Dirty Write

Ausgangslage: Das Konto `1` hat einen Saldo von `100`.

Tabelle 1: Beispiel für Dirty Write

T_1	T_2	Saldo für Konto `1`
		`100`

`update Konto`
`set Saldo = 200`
`where KtoNr = 1`
		200

	`update Konto`
	`set Saldo = 250`
	`where KtoNr = 1`
		250

`commit`		200
	`commit`	250

Im Beispiel dargestellt in Tabelle 1 und Abbildung 1 führt das Bestätigen der Transaktion T_2 dazu, dass die Änderung von Transaktion T_1 verloren ist, obwohl T_1 ihre Änderung bestätigt hat.

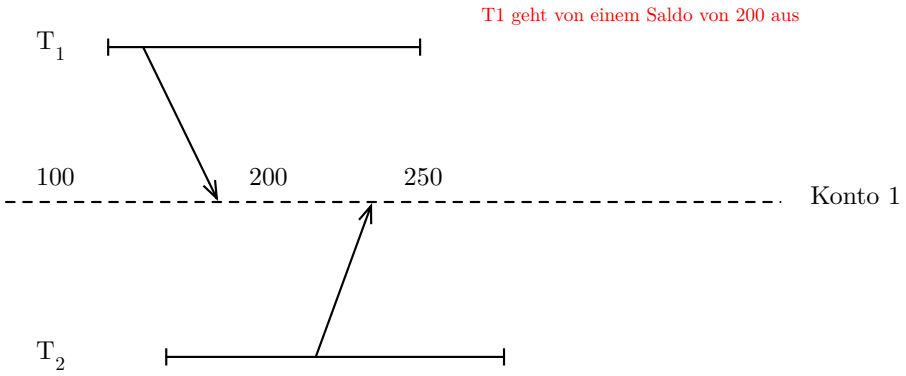

Abbildung 1: Beispiel für Dirty Write

Das Phänomen *Dirty Write* wäre fatal, weil selbst einfache Integritätsbedingungen nicht garantiert sind, selbst wenn jede einzelne Transaktion sie einhält. Nehmen wir an, dass in eine Tabelle eine Postleitzahl und ein Ort geändert werden soll und die Postleitzahl zu diesem Ort gehören muss.

Transaktion T_1:

```
T1: update Adresse set PLZ = 60312 where Id = 1;
T1: update Adresse set Ort = 'Frankfurt am Main' where Id = 1;
T1: commit;
```

Transaktion T_2:

```
T2: update Adresse set PLZ = 35390 where Id = 1;
T2: update Adresse set Ort = 'Gie ß en' where Id = 1;
T2: commit;
```

Nach der seriellen Ausführung ist die Adresse in Frankfurt oder Gießen, je nach Reihenfolge, die Zuordnung der Postleitzahlen ist in beiden Fällen korrekt.

Wäre nun *Dirty Write* erlaubt, könnte folgender Ablauf entstehen:

```
T1: update Adresse set PLZ = 60312 where Id = 1;
T2: update Adresse set PLZ = 35390 where Id = 1;
T2: update Adresse set Ort = 'Gie ß en' where Id = 1;
T2: commit;
T1: update Adresse set Ort = 'Frankfurt am Main' where Id = 1;
T1: commit;
```

Nach diesem Ablauf hätte wir eine Adresse in Frankfurt mit einer Gießener Postleitzahl!

Bemerkung Dieses Phänomen wird im SQL-Standard als *Lost Update* bezeichnet. Die Bezeichnung *Dirty Write* ist vorzuziehen, weil es noch ein anderes Phänomen gibt, das oft auch als *Lost Update* bezeichnet wird (siehe Abschnitt zur Diskussion der Isolationslevel). Der SQL-Standard definiert dieses Phänomen nicht explizit, sondern sagt etwas lapidar (und unpräzise): „The four isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that no updates will be lost“ [Abschnitt 4.35.4. in Part 2 Foundation von SQL:2003]

Dirty Read

Ausgangslage: Das Konto `1` hat einen Saldo von `100`.

Tabelle 2: Beispiel für Dirty Read

T_1	T_2	Saldo für Konto `1`
		`100`

	`update Konto`
	`set Saldo = 200`
	`where KtoNr = 1`
		`200`

`select Saldo`
`from Konto`
`where KtoNr = 1`
`...`
T_2 verwendet den Wert `200`		

	`rollback`	`100`
`commit`		`100`

Im Beispiel (dargestellt in Tabelle 2 und Abbildung 2) verwendet die Transaktion T_1 den Wert von `200` als gültigen Saldo für Konto `1`, obwohl dies niemals korrekt war, da Transaktion T_2 diesen Wert nicht bestätigt hat. Dieses Phänomen nennt man *Dirty Read*.

Das Phänomen kann auch auftreten, wenn die Transaktion T_2 die Änderungen am Ende durch ein `commit` bestätigt. Sie könnte ja zunächst den Saldo auf `200` gesetzt haben unn später nach etwaigen Berechnungen eines zusätzlichen Zinses den Saldo innerhalb der Transaktion auf `202` erhöht haben. Auch in diesem Fall würde T_1 einen Wert verwenden, der niemals gültig war.

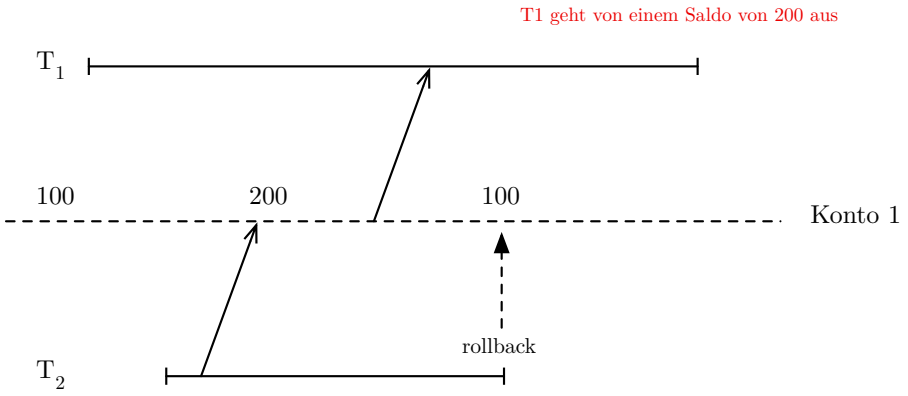

Abbildung 2: Beispiel für Dirty Read

Non-repeatable Read

Ausgangslage: Das Konto `1` hat einen Saldo von `100`.

Tabelle 3: Beispiel für Non-repeatable Read

T_1	T_2	Saldo für Konto `1`
		`100`

`select Saldo`
`from Konto`
`where KtoNr = 1`
T_1 liest den Wert `100`		

	`update Konto`
	`set Saldo = 200`
	`where KtoNr = 1`
	`...`
	`commit`
		`200`

`select Saldo`
`from Konto`
`where KtoNr = 1`
T_1 liest den Wert `200`		

Das Phänomen *Non-repeatable Read* besteht darin, dass innerhalb einer Transaktion der Zugriff auf dieselben Datenobjekte zu einem späteren Zeitpunkt in der Transaktion unterschiedliche Ergebnisse liefern kann.

In unserem Beispiel in Tabelle 3 und Abbildung 3 liest Transaktion T_1 zu-

nächst als Saldo von Konto `1` den Wert `100`, bei einem weiteren Zugriff ist der Saldo dann `200`.

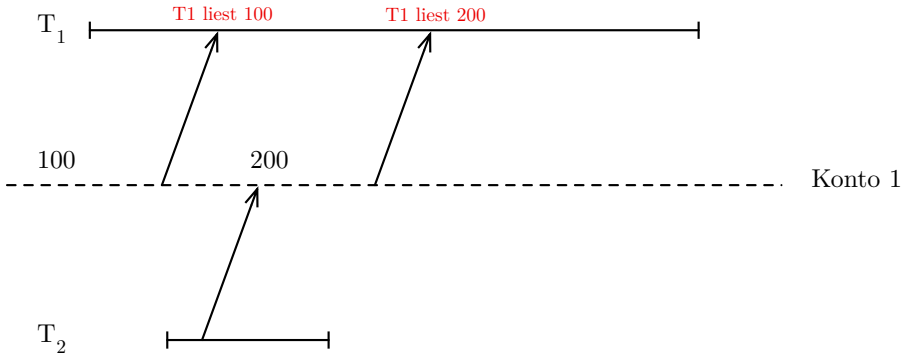

Abbildung 3: Beispiel für Non-repeatable Read

Es kann auch vorkommen, dass eine andere Transaktion den Datensatz zum Konto mit der Kontonummer `1` gelöscht hat, so dass der spätere Zugriff von Transaktion T_1 gar keinen Wert mehr zum Saldo ergeben würde. Ebenso ist es möglich, dass beim erneuten Lesen einer Ergebnismenge neue Datensätze erscheinen, die bisher nicht vorhanden waren.

Das bedeutet, dass in Transaktion T_1 Änderungen „sichtbar“ werden, die zwischenzeitlich von einer anderen Transaktion durchgeführt wurden. Dies kann in einer Anwendung durchaus erwünscht sein.

Ein weiteres Beispiel (Tabelle 4) zeigt, dass das Phänomen Non-repeatable Read zu falschen Ergebnissen führen kann.

Als Ausgangspunkt nehmen wir drei Konten mit den Salden `40`, `50`, `30`. Transaktion T_1 summiert die Salden auf und ermittelt dadurch das Gesamtguthaben der drei Konten.

In diesem Beispiel ermittelt Transaktion T_1 als Summe der Salden der Konten `1`, `2`, `3` den Wert `110`, obwohl dieser Wert weder vor noch nach der Transaktion T_2 korrekt war.

Phantom Row

Tabelle 5 zeigt ein Beispiel für das Phänomen *Phantom Row*.

Ausgangslage sind zwei Konten mit einem Saldo von jeweils `100`. Die Transaktion T_1 liest zunächst die Summe der Guthaben und erhält `200`, später liest sie nochmals die Summe. Nun erhält T_1 `250`, weil wie ein „Phantom“

Tabelle 4: Beispiel für falsche Ergebnisse bei Non-Repeatable Read

T_1	T_2	Saldo Konten `1-3`
		`40, 50, 30` Summe: `120`
T_1 liest Saldo von Konto `1` und schreibt den Wert in `sum`, also `40`.		
	T_2 ändert Saldo von Konto `3` auf `20` und von Konto `1` auf `60` `commit`	`60, 50, 20` Summe: `130`
T_1 liest Saldo von Konto `2` und addiert den Wert zu `sum`, also `90`.		
T_1 liest Saldo von Konto `3` und addiert den Wert zu `sum`, also `110`.		

plötzlich ein weiteres Konto in die Summe mit einbezogen wurde. Diese neue Zeile in der Tabelle wurde von Transaktion T_2 eingefügt.

Tabelle 5: Beispiel für Phantom Row

T_1	T_2	Saldo für Konten
		`100, 100`

`select sum(Saldo)`
`from Konto`
T_1 liest den Wert `200`		

	`insert into Konto`
	`values (3,50)`
	`commit`
		`100, 100, 50`

`select sum(Saldo)`
`from Konto`
T_1 liest den Wert `250`		

T_2 war während der Laufzeit der Transaktion T_1 in der Lage, die Summe der Salden zu verändern, indem sie ein „Phantom untergeschoben“ hat.

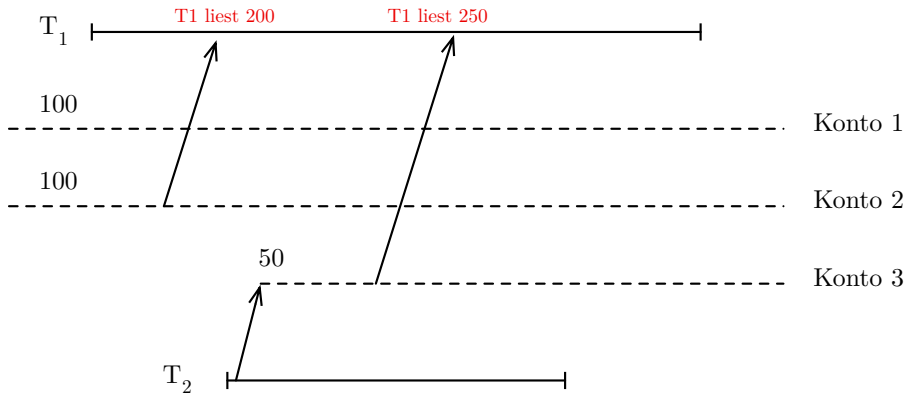

Abbildung 4: Beispiel für Phantom Row

Bemerkung Die Beispiele sind so konstruiert, dass für die einzelnen Schritte der beteiligten Transaktionen SQL-Anweisungen verwendet werden. Eine einzelne SQL-Anweisung besteht in der Regel aus vielen elementaren Operationen des Datenbankmanagementsystems. Der SQL-Standard schreibt jedoch fest, dass jede einzelne SQL-Anweisung atomar und isoliert gegenüber anderen Transaktionen durchgeführt werden muss [2, S. 877].[1]

Definition der Isolationslevel in SQL

Der SQL-Standard formuliert die Phänomene folgendermaßen [Abschnitt 4.35.4. in Part 2 Foundation von SQL:2003]:

1. „P1 (“Dirty read”): SQL-transaction T1 modifies a row. SQL-transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed and that may thus be considered to have never existed.“
2. „P2 (“Non-repeatable read”): SQL-transaction T1 reads a row. SQL-transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.“
3. „P3 (“Phantom”): SQL-transaction T1 reads the set of rows N that satisfy some <search condition>. SQL-transaction

[1] Berenson et al. sagen: „If one looks carefully at the SQL standard, it defines each statement as atomic. It has a serializable sub-transaction (or timestamp) at the start of each statement.“ ([1, S. 10])

> T2 then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same <search condition>, it obtains a different collection of rows.“

In SQL werden die Isolationslevel dadurch *definiert*, dass festgelegt wird, welche der Phänomene des konkurrierenden Zugriffs von Transaktionen *garantiert nicht auftreten* können und welche eventuell auftreten können.

Das Phänomen *Dirty Write* darf niemals auftreten.

In Tabelle 6 wird spezifiziert, welche Phänomene in welchem Isolationslevel auftreten können bzw. garantiert *nicht* auftreten.

Der Isolationslevel `SERIALIZABLE` ist natürlich durch die Abwesenheit aller Phänomene — wie in der Tabelle angegeben — nicht wirklich hinreichend spezifiziert. Der SQL-Standard verlangt, dass eine Transaktion, die dieses Transaktionslevel verlangt, die Garantie erhält, dass ihr Ablauf gegenüber anderen Transaktionen wie ein serieller Ablauf ist.

> „The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.“ [Abschnitt 4.35.4. in Part 2 Foundation von SQL:2003]

Tabelle 6: Definition der Isolationslevel in SQL

	Dirty Read	Non-Repeatable Read	Phantom Row
`READ UNCOMMITTED`	möglich	möglich	möglich
`READ COMMITTED`	*nicht* möglich	möglich	möglich
`REPEATABLE READ`	*nicht* möglich	*nicht* möglich	möglich
`SERIALIZABLE`	*nicht* möglich	*nicht* möglich	*nicht* möglich

Es handelt sich bei den Festlegungen in der Tabelle 6 um eine *Zusicherung* des Datenbankmanagementsystems für eine Transaktion: Stellt eine Transaktion z.B. das Isolationslevel `READ COMMITTED` ein, garantiert das Datenbankmanagementsystem, dass für diese Transaktion niemals das Phänomen „Dirty Read“ eintreten kann, möglicherweise aber eines der Phänomene „Non-repeatable Read“ oder „Phantom Row“.

Bemerkung Der SQL-Standard legt in der Spezifikation der Isolationslevel das *Verhalten* eines Datenbankmanagementsystems fest, *nicht* eine bestimmte Art der Implementierung der Isolationslevel.

Tatsächlich gibt es verschiedene Möglichkeiten die Isolationslevel in einem Datenbankmanagementsystem zu implementieren. Wir werden unten die beiden wichtigsten Konzepte kennenlernen.

Es wird sich dabei zeigen, dass die Unterschiede der Konzepte zu subtilen Unterschieden im Verhalten von Datenbankmanagementsystemen in Konkurrenzsituationen von Transaktionen führen kann. Werden solche Unterschiede in einer Anwendung ausgenutzt, kann diese nicht mehr leicht auf ein anderes Datenbankmanagementsystem portiert werden, sofern dieses ein anderes Konzept der Isolationslevel implementiert.

Diskussion

Tabelle 7: Beispiel für Lost Update

T_1	T_2	Saldo für Konto `1`
		`100`

`select Saldo`
`from Konto`
`where KtoNr = 1`
T_1 liest den Wert `100`		

	`select Saldo`
	`from Konto`
	`where KtoNr = 1`
	T_2 liest den Wert `100`	

`update Konto`
`set Saldo = 100+100`
`where KtoNr = 1`
`...`
`commit`
		`200`

	`update Konto`
	`set Saldo = 100+50`
	`where KtoNr = 1`
	`...`
	`commit`
		`150`

Man kann den Eindruck haben, dass im SQL-Standard *alle* möglichen Phänomene in Konkurrenzsituationen berücksichtigt werden. Dies ist jedoch nicht der Fall.

Das Phänomen *Lost Update*[2] kann auftreten, wenn das Isolationslevel `READ COMMITTED` verlangt wird. Ein Beispiel wird in Tabelle 7 dargestellt.

In diesem Beispiel liest zunächst Transaktion T_1 den Wert `100` als Saldo von Konto `1`, dann liest T_2 denselben Wert. T_1 verwendet den gelesenen Wert für eigene Berechnungen, addiert `100` und schreibt den neuen Wert `100+100` als neuen Saldo. Auch T_2 verwendet den gelesenen Wert und schreibt kurz nach T_1 dem Konto `50` zu.

Im Ergebnis sind `100`, die Addition von Transaktion T_1, verloren, und das obwohl beide Transaktionen durch `commit` bestätigt wurden.

Dieses Phänomen tritt nicht mehr auf, wenn man den Isolationslevel `REPEATABLE READ` einstellt. Man hätte jedoch auch einen eigenen Isolationslevel für den Ausschluss dieses Phänomens definieren können, in der Literatur oft *Cursor-Stabilität* genannt.

Zur kritischen Diskussion der Definition der Isolationslevel in SQL siehe: [1].

Konzepte der Implementierung von Isolationsleveln

Es gibt zwei grundlegende Techniken, die Isolationslevel zu implementieren: durch Sperrverfahren und durch Multiversionierung.

Sperrverfahren

Bei Sperrverfahren werden Datenobjekte mit Sperren (*locks*) versehen, die den Zugriff von Transaktionen auf diese Datenobjekte einschränken. Diese Sperren haben in einem Datenbankmanagementsystem in der Regel unterschiedliche Granularität: Datenzeile, Tabelle o.ä. Dies wollen wir aber nicht näher betrachen.

Wir unterscheiden Arten von Sperren oder ihren *Modus*:

- Eine *Lesesperre*, auch nicht exklusive Sperre (*read lock*), bedeutet, dass eine Transaktion ein Datenobjekt lesen kann. Mehrere Transaktion können gleichzeitig Lesesperren auf demselben Datenobjekt halten.
- Eine *Schreibsperre*, auch exklusive Sperre (*write lock*), bedeutet, dass eine Transaktion ein Datenobjekt verändern kann. Eine Schreibsperre auf einem Datenobjekt kann nur eine Transaktion haben, es dürfen also keine anderen Sperren vorhanden sein.

Weiter unterscheidet man bezüglich der *Dauer* einer Sperre:

- Eine *kurze* Sperre wird von einer Transaktion auf einem Datenobjekt

[2] Das hier beispielhaft beschriebene Phänomen wird in der Literatur oft als *Lost Update* bezeichnet, etwa in [2], es ist nicht zu verwechseln mit dem oben beschriebenen Phänomen *Dirty Write*, das manchmal auch als *Lost Update* bezeichnet wird.

nur während des Zugriffs gehalten und danach gleich wieder freigegeben.
- Eine *lange* Sperre wird im Verlauf einer Transaktion angefordert und dann bis zum Ende der Transaktion gehalten. Die Freigabe erfolgt erst beim Bestätigen oder Verwerfen der Transaktion.

Für unsere Diskussion sind auch die sogenannten *Prädikatsperren* von Interesse: Bei einer SQL-Anweisung ist mit den Tabellen, auf die sie zugreift, ein Prädikat verbunden. Dieses Prädikat ist ein logische Aussage, die auf alle Tupel zutrifft, die in der Anweisung verwendet werden. Die Menge der Datenobjekte, auf die dieses Prädikat zutrifft, besteht nicht nur aus den im Moment in den Tabellen gespeicherten Datensätzen, sondern auch denjenigen, die den Tabellen hinzugefügt werden können und dann dieses Prädikat erfüllen.

Eine *Prädikatsperre* betrifft alle diejenigen Datensätze, die das Prädikat erfüllen sowie jene, die durch eine Modifikation (`insert`, `update` oder `delete`) das Prädikat erfüllen würden.

Verhalten der Transaktion

- im Isolationslevel `READ UNCOMMITTED`: Die Transaktion berücksichtigt beim Lesen keine Sperren. Gemäß SQL-Standard darf eine Transaktion im Isolationslevel `READ UNCOMMITTED` nur lesende Zugriffe machen.
- im Isolationslevel `READ COMMITTED`: Die Transaktion verwendet beim Lesen kurze Lesesperren; sie verwendet beim Schreiben lange exklusive Prädikatsperren.
- im Isolationslevel `REPEATABLE READ`: Die Transaktion verwendet beim Lesen lange Lesesperren (auch auf Ergebnismengen); sie verwendet beim Schreiben lange exklusive Prädikatsperren.
- im Isolationslevel `SERIALIZABLE`: Die Transaktion verwendet beim Lesen lange nicht-exklusive Prädikatsperren; sie verwendet beim Schreiben lange exklusive Prädikatsperren.

Diskussion

- Eine Transaktion im Level `READ UNCOMMITTED` liest Datenobjekte ohne Sperren zu berücksichtigen. Das bedeutet, dass sie auch Daten lesen kann, auf die andere Transaktionen Sperren halten, die erst bei der Bestätigung der Transaktion freigegeben werden, d.h. eine solche Transaktion kann ein „Dirty Read“ machen.
- Eine Transaktion im Level `READ COMMITTED` liest Datenobjekte mit einer kurzen Lesesperre. D.h. sie kann nur Daten lesen, die andere Transaktionen bestätigt haben. Aber ein neues Lesen desselben Datenobjekts innerhalb der Transaktion kann zwischenzeitliche Änderungen anderer Transaktionen sichtbar machen.

- Eine Transaktion im Level `REPEATABLE READ` hält lange Lesesperren auf alle Datensätze der Ergebnismenge, d.h. ein erneutes Lesen eines bereits gelesenen Datenobjekts ergibt dasselbe Ergebnis, weil durch die lange Sperre verhindert wird, dass andere Transaktionen Änderungen an Datensätzen der Ergebnismenge vornehmen können.
- Da eine Transaktion im Level `SERIALIZABLE` länge Prädikatsperren beim Lesen hält, können auch keine Phantomzeilen entstehen.

Bemerkung Mit Prädikatsperren kann man eine Implementierung der in SQL geforderten Isolationslevel erreichen, ohne dass man zu dem Mittel greifen muss, komplette Tabellen exklusiv zu sperren. Allerdings ist eine Implementierung von Prädikatsperren sehr aufwändig. Deshalb werden in den heute üblichen Datenbankmanagementsystemen keine Prädikatsperren eingesetzt. Aber es gibt eine Technik, bei der man mit Sperren feinerer Granularität auskommt als das Sperren eines Tabelle. Dies ist dann möglich, wenn im Zugriffspfad ein *Index* verwendet wird. Solche Indexsperren werden z.B. beschrieben in [2, S. 887ff].

Multiversionierung

Wir betrachten drei Varianten von Verfahren, die Multiversionierung einsetzen: (1) *Read-Only Multiversion Concurrency Control*, (2) *Read-Consistency Multiversion Concurrency Control* und (3) *Snapshot Isolation*. Allen diesen Verfahren ist gemeinsam, dass jedem Datenobjekt der Datenbank eine Versionsnummer zugeordnet wird. Auf diese Weise kann das DBMS verschiedene Versionen eines Datenobjekts haben und Transaktionen zur Verfügung stellen.

Read-Only Multiversion Concurrency Control

Die Besonderheit besteht darin, dass eine Transaktion zu Beginn bekannt gibt, ob sie nur lesende Zugriffe oder auch schreibende Zugriffe machen möchte.

Wird die Transaktion mit der Eigenschaft `READ ONLY` eröffnet, erhält sie einen Schnappschuss der Datenbank zum Zeitpunkt des ersten Zugriffs.

Wird die Transaktion jedoch als schreibende eröffnet, führt der Transaktionsmanager ein striktes 2-Phasen-Lock-Protokoll durch.

Dieses Verfahren hat die Eigenschaft, dass Transaktionen, die nur lesen, niemals Sperren benötigen, da sie ja einen Schnappschuss der Datenbank verwenden. Deshalb müssen schreibende Transaktionen auch niemals auf die Freigabe einer Lesesperre der rein lesenden Transaktionen warten.

Read-Consistency Multiversion Concurrency Control

Auch bei diesem Verfahren wird zwischen rein lesenden (`READ ONLY`) Transaktion und schreibenden Transaktionen unterschieden.

Rein lesende Transaktionen erhalten einen Schnappschuss der Datenbank.

Schreibende Aktionen innerhalb einer Transaktion verwenden eine lange Schreibsperre auf den Datenobjekten, die sie verändern.

Lesende Aktionen einer Transaktion erhalten stets die aktuellste Version des gefragten Datenobjekts.

Dieses Verfahren hat die Eigenschaft, dass keine lesende Aktion eine Sperre benötigt, d.h. niemals kann es vorkommen, dass eine Transaktion beim Schreiben eines Datenobjekts auf eine andere Transaktion warten muss, die dieses Datenobjekt nur liest.

Dieses Verfahren ist die Implementierung des Isolationslevels `READ COMMITTED` in Oracle.

Snapshot Isolation

In diesem Fall muss nicht zu Beginn der Transaktion unterschieden werden, ob sie nur lesende Zugriffe oder auch schreibende Zugriffe macht.

Jede lesende Aktion erhält die Werte der Version, die zu Beginn der Transaktion aktuell war. D.h. alle lesenden Zugriffe beziehen sich auf einen Schnappschuss der Datenbank zum Zeitpunkt des Beginns der Transaktion.

Zwei parallel ablaufende Transaktion müssen in Bezug auf schreibende Zugriffe die sogenannte *disjoint-write property* haben: sie dürfen nur unterschiedliche Datenobjekte verändern. Sobald eine Transaktion versucht, Datenobjekte zu verändern, die die andere Transaktion bereits verändert hat (d.h. die eine höhere Version haben als die der eigenen Transaktion), wird eine der beiden konfligierenden Transaktionen abgebrochen. Diesen Konflikt kann man zum Beispiel dadurch feststellen, dass bei Änderungen von Datenobjekten Schreibsperren eingesetzt werden. (Oracle tut das z.B. so.)

Dieses Verfahren ist die Implementierung des Isolationslevels `SERIALIZABLE` in Oracle.

Anomalie bei *Snapshot Isolation*

Bei *Snapshot Isolation* kann ein Phänomen auftreten, das *Write Skew*[3] genannt wird.

[3] Eine übliche deutsche Übersetzung habe ich nicht gefunden, man könnte vielleicht „Schieflage beim Schreiben“ sagen.

Folgender Ablauf ist nicht serialisierbar, in *Snapshot Isolation* aber erlaubt:

$$r_1(a); r_1(b); r_2(a); r_2(b); w_2(a); c_2; w_1(b); c_1;$$

Was kann in einem solchen Beispiel passieren? Stellen wir uns vor, wir haben *zwei* Konten und die Bank hat die Bedingung, dass die Summe beider Salden immer >= 0 sein soll. Bei *Snapshot Isolation* kann nun folgendes geschehen:

Tabelle 8: Beispiel für Write Skew

T_1	T_2	Salden der Konten
		Konto 1: `100` Konto 2: `100`

`select Saldo`
`from Konto`
`where KtoNr in (1,2)`
T_1 liest die Werte `100,100`		

	`select Saldo`
	`from Konto`
	`where KtoNr in (1,2)`
	T_2 liest den Wert `100,100`	

`update Konto`
`set Saldo = 100-120`
`where KtoNr = 1`
`...`
`commit`
		Konto 1:`-20`

	`update Konto`
	`set Saldo = 100-120`
	`where KtoNr = 2`
	`...`
	`commit`
		Konto 2: `-20`

Beide Transaktionen denken nach dem Lesen der Salden, dass ein Überziehen eines der Konten erlaubt ist, weil das andere Konto die geforderte Deckung aufweist. Bei *Snapshot Isolation* wird kein Konflikt entdeckt, weil die beiden Transaktionen auf unterschiedliche Datenobjekte schreibend zugreifen. Ergebnis: die verlangte gemeinsame Deckung der beiden Konten wird nicht garantiert!

Implementierung Man sieht also, dass die *Snapshot Isolation* zwar alle Phänomene verhindert, wie sie im SQL-Standard definiert sind, gleichwohl aber nicht Serialisierbarkeit garantiert. MS SQL Server unterscheidet deshalb korrekterweise zwischen dem Isolationslevel `SNAPSHOT ISOLATION` und dem Isolationslevel `SERIALIZABLE`. Oracle tut das nicht. Eine Anwendung, entwickelt auf MS SQL Server, die wirkliche Serialisierbarkeit erwartet, kann, portiert auf Oracle, deshalb eventuell ein subtil anderes Verhalten zeigen!

In PostgreSQL wird seit Version 9.1 eine Implementierung namens *Serializable Snapshot Isolation (SSI)* verwendet, die diese Anomalie verhindert, siehe [3].

Literaturverzeichnis

[1] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995.*, pages 1–10, 1995.

[2] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems: An Application-Oriented Approach.* Boston, 2nd edition, 2006. Complete Version.

[3] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, 8 2012.