Transaktionen

2

In betrieblichen Anwendungen sind die Daten häufig unternehmensweit – mindestens jedoch abteilungsweit – organisiert, und die Nutzer haben Zugriff über ein lokales Netzwerk oder über das Internet.

Hierbei ergibt sich das Problem, dass verschiedene Nutzer gleichzeitig auf dieselben Daten zugreifen und sie sogar gleichzeitig ändern. Das Datenbanksystem muss diese Zugriffe so organisieren, dass durch gleichzeitige Zugriffe nicht irgendwelche Tupel in unerwünschter Weise geändert werden.

2.1 Eigenschaften von Transaktionen

Als Transaktionen[1] werden Folgen von Datenmanipulationen bezeichnet, die eine logische Einheit bilden (englisch: *logical unit of work*). Eine einzelne Operation könnte, für sich betrachtet, die Konsistenz der Datenbank zerstören. Werden sie aber alle ausgeführt, geht die Datenbank wieder in einen konsistenten Zustand über.

Folgende Merkmale werden mit dem Begriff der Transaktion verbunden:

- *Atomarität*:
  Eine Transaktion wird entweder vollständig oder gar nicht, keinesfalls aber nur teilweise ausgeführt.
- *Konsistenz*:
  Eine Transaktion überführt eine Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand.

[1] Eine umfangreiche Darstellung findet sich zum Beispiel in [MeSi02].

M. Unterstein und G. Matthiessen, *Anwendungsentwicklung mit Datenbanken*, eXamen.press,
DOI: 10.1007/978-3-642-39003-6_2, © Springer-Verlag Berlin Heidelberg 2013

- *Isoliertheit*:
  Andere gleichzeitig ablaufende Transaktionen haben keinen Einfluss auf das Ergebnis einer aktiven Transaktion. Diese Forderung, die für einen reibungslosen Mehrbenutzerbetrieb entscheidend ist, wird durch die *Serialisierbarkeit* erreicht. Dies bedeutet, dass mehrere gleichzeitig ablaufende Transaktionen dasselbe Resultat bewirken, als hätten sie nacheinander stattgefunden.
- *Dauerhaftigkeit*:
  Nach Beendigung einer Transaktion sind die durch sie bewirkten Änderungen in der Datenbank dauerhaft gespeichert.[2]

In Anlehnung an die englischen Begriffe *atomicity*, *consistency*, *isolation* und *durability* fasst man diese Eigenschaften unter dem Kürzel „ACID“ zusammen.

Beispiel: Aktionen der referenziellen Integrität

Aus irgendeinem Grund soll der Wert von `kunden_nr` in der Tabelle kunde geändert werden. Wenn zu diesem Kunden Bestellungen existieren, muss konsequenterweise auch in der Tabelle `bestellung` die Änderung der `kunden_nr` nachvollzogen werden. Sonst verlieren alle Bestellungen, in denen dieser Wert von `kunden_nr` als Fremdschlüssel erscheint, ihre Zuordnung. Die Datenbank wäre dann inkonsistent. Durch *eine einzige* SQL-Anweisung wird nun eine Folge von Datenbankoperationen ausgelöst:

```
UPDATE kunde
SET  kunden_nr ....
...
```

Dass hieraus mehrere Update-Aktionen folgen, liegt an der Deklaration der `kunden_nr` als Fremdschlüssel in `bestellung`, wobei die Weitergabe aller Wertänderungen des Primärschlüssels an den Fremdschlüssel verlangt wird:

```
FOREIGN KEY (kunden_nr)
  REFERENCES kunde
    ON UPDATE CASCADE
```

Beispiel: Banküberweisung

Dabei müssen zwei SQL-Anweisungen ausgeführt werden: Vom Konto 11 werden beispielsweise 100,- Euro auf das Konto 22 überwiesen.

[2] Sie können natürlich durch eine spätere Transaktion wieder geändert werden.

```
UPDATE konto
SET    stand = stand + 100
WHERE  konto_nr =22
```

und

```
UPDATE konto
SET    stand = stand - 100
WHERE  konto_nr = 11
```

Die Transaktionsmerkmale stellen sich an diesem Beispiel wie folgt dar:

- *Konsistenz*:
  Zwischen der ersten und der zweiten Anweisung ist die Datenbank in einem inkonsistenten Zustand. Die Bank hat in ihrem Gesamtsaldo 100,- Euro zu viel. Nach der Ausführung der gesamten Transaktion ist die Datenbank wieder konsistent.
- *Atomarität*:
  Die beiden Buchungen werden komplett oder gar nicht ausgeführt.
- *Dauerhaftigkeit*:
  Nach der zweiten Buchung werden die neuen Kontostände gespeichert. Die Änderungen können nun nicht mehr verloren gehen.
- *Isoliertheit/Serialisierbarkeit*:
  Wenn während dieser Transaktion ein zweiter Buchungsvorgang beginnt, der das Konto 22 betrifft, muss dessen Stand am Ende denselben Wert haben, als wenn die Buchungen nacheinander stattgefunden hätten (in diesem Fall wäre die Reihenfolge beliebig).

2.1.1 Gefahren für die Konsistenz

Gefährungen für die Datenkonsistenz können einmal auf der physikalischen Ebene liegen:

- Programmfehler in Applikationen, die z. B. zu Endlosschleifen mit Überlauf des Stacks führen
- Betriebssystembedingte Fehler, z. B. unerlaubt hoher Verbrauch an Ressourcen wie Hauptspeicher, Plattenspeicher, Puffer
- Zusammenbruch des Datenbanksystems
- Hardware-Fehler, z. B. Zerstörung eines Datenträgers
- Stromausfall

Ist die gesamte Datenbank durch einen Hardware-Fehler oder aufgrund eines nicht reversiblen Betriebssystemfehlers physisch zerstört, ist nichts anderes möglich als die Wiederherstellung aus einer – hoffentlich nicht weit zurückliegenden – Datensicherung.

Eine andere Ursache für Inkonsistenzen liegt auf der logischen Ebene: die Datenbank ist nicht physisch zerstört, aber ihr Inhalt ist inkonsistent geworden, weil eine Reihe zusammenhängender Operationen nicht vollständig ausgeführt wurde. Hauptursache ist hier:

- der gleichzeitige Zugriff mehrerer Benutzer auf dieselben Daten

Für Fehler der ersten Kategorie ist die regelmäßige Datensicherung auf einem anderen Medium die sicherste Vorsorge. Auf Basis einer solchen Sicherung kann das Transaktionsmanagement eines DBMS unter Umständen einen relativ aktuellen Stand der Datenbank wiederherstellen. Dazu wird ein so genanntes „After-Image"-Protokoll verwaltet, das die nochmalige Ausführung abgeschlossener Transaktionen ermöglicht.

Fehler der zweiten Art können beseitigt werden, indem der Zustand der Daten wiederhergestellt wird, den diese vor Beginn der Transaktion hatten. Hierzu werden „Before-Image"-Protokolle verwendet.[3]

2.1.2 Transaktionsmanagement

Die Aufgabe des Transaktionsmanagements eines DBMS ist es, die ordnungsgemäße Beendigung von Transaktionen zu kontrollieren. Dabei muss es Situationen erkennen und bewältigen, die die Konsistenz der Datenbank gefährden. Falls der erfolgreiche Abschluss einer Transaktion nicht möglich ist, muss es die Datenbank auf einen bekanntermaßen konsistenten Zustand zurückführen, d. h. alle seit Beginn der Transaktion vorgenommenen Einzeländerungen rückgängig machen.

2.2 Transaktionen in SQL

Transaktionen können nicht „verschachtelt" sein, es kann also innerhalb einer laufenden Transaktion keine weitere Transaktion (etwa wie ein Unterprogramm) gestartet werden.

2.2.1 Start von Transaktionen

Eine Transaktion beginnt implizit, wenn gerade keine Transaktion aktiv ist und eine DML-Anweisung gestartet wird (also z. B. SELECT oder UPDATE).

Des weiteren kann eine Transaktion durch die START TRANSACTION-Anweisung[4] begonnen werden. Voraussetzung dazu ist, dass gerade keine Transaktion aktiv ist.[5]

[3] Ausführungen zu Recovery Management und Logdateien sind in Abschn. 1.2.7 und 1.2.8 zu finden.

[4] Die Optionen dieser Anweisung werden in Abschn. 2.4 eingeführt.

[5] Dieses würde zu einer Ausnahmebedingung führen.

Durch die START TRANSACTION-Anweisung können der Transaktion Parameter mitgegeben werden.

2.2.2 Beendigung von Transaktionen

Die Beendigung einer Transaktion ist auf verschiedene Weise möglich. Im Allgemeinen wird ein Transaktionsende durch den Benutzer selbst (respektive durch ein Anwendungsprogramm) explizit angefordert. Dazu stehen die beiden SQL-Anweisungen `COMMIT` und `ROLLBACK` zur Verfügung. Daneben können Transaktionen implizit beendet werden, worauf wir weiter unten zurückkommen.

Ist der Benutzer sicher, dass alle Operationen richtig ausgeführt wurden, muss er dem DBMS mitteilen, dass die Änderungen in der Datenbank dauerhaft niedergelegt werden sollen. Dies geschieht mit der Anweisung:

```
COMMIT [WORK]
```

Erst ab diesem Moment sind die geänderten Daten für andere Benutzer sichtbar, sofern der Isolation Level von ihrer Seite nicht auf READ UNCOMMITTED eingestellt wurde.[6] Solange eine Transaktion noch nicht abgeschlossen ist, sind alle Änderungen sozusagen „privat“, das heißt nur für den Prozess existent, der sie durchgeführt hat. Dies gilt im Übrigen auch, wenn solche zusammenhängenden Änderungen mit Datenbankprozeduren ausgeführt werden (vgl. Kap. 4).

Des Weiteren wird eine Transaktion durch eine `DDL`-Anweisung implizit beendet. Dieses wirkt wie eine zuvor ausgefürte COMMIT-Anweisung.

Die Anweisung

```
ROLLBACK [WORK]
```

macht die Änderungen seit Beginn der Transaktion rückgängig. Wurden beispielsweise irrtümlich für alle Artikel die Verkaufspreise um 10 % erhöht, so kann die Wirkung einer solchen Anweisung, z. B.:

```
UPDATE artikel
   SET listenpreis = listenpreis * 1.1
```

mit `ROLLBACK` aufgehoben werden. Die Artikeltabelle enthält dann dieselben Werte in der Spalte `listenpreis` wie vor der Änderung.

Technisch liegt dieser Möglichkeit, Datenänderungen rückgängig zu machen, eine zeitweise Duplizierung der Daten zugrunde. Bei ORACLE beispielsweise werden alle Tupel vor der Änderung in die bereits erwähnte Before-Image-Datei kopiert. Aus dieser werden sie mit `ROLLBACK` in die Datenbank zurückkopiert.

[6] Isolation level behandeln wir in Abschn. 2.4.

2.2.3 Autocommit

Verschiedene DBMS können über Systemvariablen so konfiguriert werden, dass sie nach jeder UPDATE-, INSERT- oder DELETE-Anweisung automatisch ein `COMMIT` durchführen; bei einigen Systemen[7] ist dies sogar die Voreinstellung. Dies ist in vielen Fällen sinnvoll, wenn nämlich alle Änderungen mit einer einzigen Anweisung erledigt werden. Es hat zudem den Vorteil, dass andere Benutzer möglichst bald mit dem aktualisierten Datenbestand arbeiten können. Sollen aber mehrere Änderungen, Einfügungen oder Löschungen im Zusammenhang betrachtet werden, muss dieses automatische `COMMIT` unterbunden werden.

Die zugehörigen Systemvariablen sind nicht standardisiert.

Bei ORACLE heißt die Systemvariable `AUTOCOMMIT` und wird durch die Anweisung

```
SET AUTOCOMMIT ON | OFF
```

eingestellt.

Bei SQL Anywhere heißt sie `AUTO_COMMIT`, die entsprechende Anweisung sieht wie folgt aus:

```
SET OPTION AUTO_COMMIT ON | OFF
```

2.2.4 Start des DBMS nach Systemzusammenbruch

Wenn einer der oben aufgezählten Fälle von Systemfehlern eintritt, ist keine ordnungsgemäße Beendigung der Transaktion mehr möglich. Wegen der Eigenschaft Atomarität muss das DBMS dann beim nächsten Laden automatisch die Datenbank in den letzten konsistenten Zustand zurückversetzen. Die Aufgabe der Anwender ist es dann, die unvollständig gebliebenen Änderungen noch einmal vorzunehmen.

Transaktionsabschnitte Mit so genannten „Sicherungspunkten“ (englisch „*savepoint*“) wird in SQL ein Konzept realisiert, das es erlaubt, im Fehlerfall möglichst viele der bereits erfolgreich ausgeführten Eingaben und Änderungen zu erhalten. Eine Transaktion kann damit in einzelne Abschnitte aufgeteilt werden – mit dem Effekt, dass bei einem Zurücksetzen nicht alle Änderungen seit Beginn der Transaktion rückgängig gemacht werden, sondern nur die, die seit dem letzten oder einem anderen ausgewählten Sicherungspunkt erfolgten. Beispielsweise kann bei einer zeitaufwändigen Änderung über mehrere Tabellen nach jeder UPDATE-Anweisung ein solcher Abschnittspunkt gesetzt werden, um im Fehlerfall nur den Teil der Datenmanipulationen wiederholen zu müssen, bei dem der Fehler aufgetreten ist. Es scheint so, als würde damit die Forderung nach Atomarität verletzt; es ist aber nur ein teilweises Zurücksetzen einer Transaktion,

[7] Zum Beispiel bei MySQL.

kein teilweises COMMIT möglich. Auch verhindert das Zurücksetzen auf einen Sicherungspunkt nicht, dass eine Transaktion vollständig ungeschehen gemacht wird, wenn sie insgesamt nicht vollständig abgeschlossen werden kann.
Mit der Anweisung

```
SAVEPOINT name
```

wird ein Transaktionsabschnitt definiert. Soll nach einem Fehler die unterbrochene Transaktion an dieser Stelle fortgesetzt werden, so fordert man dies mit der Anweisung

```
ROLLBACK TO SAVEPOINT name
```

an. Man kann einen zuvor gesetzten Sicherungspunkt auch ohne Rollback aufheben, und zwar mit

```
RELEASE SAVEPOINT name
```

Die Entscheidung über Zeitpunkte, an denen die Datenbank konsistent ist, ist nicht trivial. Wenn im interaktiven Modus mit SQL oder einer anderen Zugriffsmethode wie Query-By-Example gearbeitet wird, muss sich der Benutzer darüber klar sein, wann eine zusammenhängende Folge von Manipulationen die Konsistenz seiner Daten gefährdet, und er muss mit SET AUTOCOMMIT OFF, COMMIT und ROLLBACK sowie etwaigen über SAVEPOINT gesetzten Transaktionsabschnitten zweckentsprechend umgehen. Bei interaktiven Applikationen obliegt es dem Programmierer, durch die explizite Angabe des Beginns und Endes der Transaktionen im Programmcode für die ständige Wiederherstellbarkeit von Konsistenz und Integrität der Datenbank Sorge zu tragen. Dazu gehört auch die Einrichtung von Fehlerbehandlungsroutinen, die ein differenziertes Fortsetzen, Abbrechen oder Wiederholen einer ganz oder teilweise gescheiterten Transaktion vorsehen.

2.3 Nebenläufige Ausführung von Transaktionen

Wenn durch konkurrierende Zugriffe auf eine Datenbasis gleichzeitig Änderungen von Daten durch verschiedene Benutzer stattfinden, ist es Aufgabe des Transaktionsmanagements, mögliche Inkonsistenzen zuverlässig zu verhindern. Dies können unter anderem Sperrkonzepte leisten, die die gleichzeitige Verfügbarkeit der Daten für alle Benutzer einschränken. Hierzu gehören auch „Nur-Lese-Transaktionen“ und das in SQL vorgesehene Konzept der „Isolation-Level“.

Die Nebenläufigkeit von Prozessen im Mehrbenutzerbetrieb („Concurrency“) und die Erhaltung der Datenkonsistenz („Consistency“) stellen widersprüchliche Anforderungen an ein DBMS. Einerseits ist es eine der Hauptaufgaben, einen gemeinsamen Datenbestand für die Verarbeitung an verschiedenen Orten und durch verschiedene Personen zugleich verfügbar zu machen. Andererseits können durch die gleichzeitige oder fast

gleichzeitige Änderung von Daten Konflikte entstehen, die die Datenbasis inkonsistent und damit unbrauchbar werden lassen. Die Extrempunkte machen den Widerspruch deutlich: Der Standpunkt der Konsistenz würde in radikaler Form realisiert, wenn die gesamte Datenbank jeweils zu einem bestimmten Zeitpunkt nur einem Benutzer zur Verfügung steht. Schrankenlose Konkurrenz würde bedeuten, dass alle Daten zu jedem Zeitpunkt allen Benutzern für Lese- und Schreibzugriffe zur Verfügung stehen. Damit wäre jegliche Kontrolle über gleichzeitige Änderungen unmöglich. Die Aufgabe der „Concurrency Control" ist es, diesen Widerspruch so zu regeln, dass beiden Anforderungen Genüge getan wird.

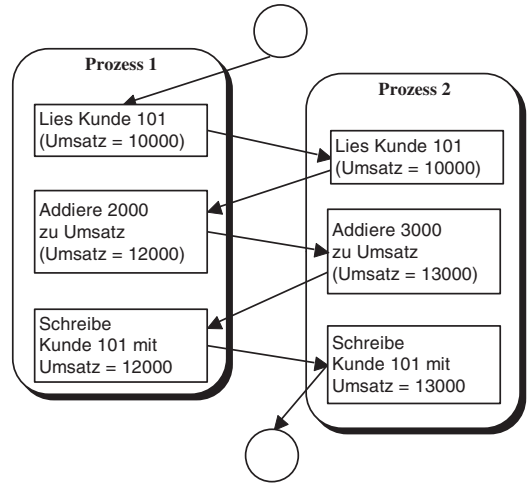
Dabei gibt es prinzipiell zwei Varianten: pessimistische Verfahren, die mit vorsorglichen Sperren arbeiten und optimistische Verfahren, die im Nachhinein feststellen, ob durch nebenläufige Änderungen Konsistenzprobleme aufgetreten sind. Die pessimistischen Verfahren entziehen die zu verändernden Daten zeitweise der allgemeinen Verfügbarkeit. Die auf dem Markt angebotenen Datenbanksysteme unterscheiden sich teilweise erheblich darin, wie fein und differenziert die zu sperrende Datenmenge und die Zeitdauer der Sperre festgelegt werden können und welche konkurrierenden, aber nicht die Konsistenz gefährdenden Zugriffe durch andere Benutzer noch möglich sind. Beispielsweise kann sich die Sperre auf einen physikalischen Speicherbereich, eine ganze Tabelle oder auf eine einzelne Zeile beziehen. Die Zeitdauer einer Sperre kann sich auf eine einzelne Schreib- oder Leseoperation, aber auch auf eine längere Transaktion beziehen. Anderen Benutzern kann unter Umständen erlaubt werden, die gesperrten Daten zu lesen. „Optimistische Methoden" gehen hingegen von der Annahme aus, dass gleichzeitige Zugriffe auf Daten recht selten vorkommen, so dass nicht im Vorhinein, sondern erst nachträglich, im Falle eines tatsächlich eingetretenen Konflikts, reagiert werden muss.

Mehrere Beispiele sollen die Problematik illustrieren.

Situation 1: „Lost Update" Wir nehmen an, eine Tabelle `kunde` enthalte die Spalte `umsatz_kumuliert`, die nach jeder Rechnungserstellung zu aktualisieren ist. Es ist dafür nötig, den alten Wert von `umsatz_kumuliert` zu lesen, im Hauptspeicher den neuen Nettobetrag der Rechnung zu addieren und das Resultat in die Kundentabelle zurückzuschreiben. Prozess 1 erhöhe den Ausgangsbetrag von 10.000 auf diese Weise um 2.000. Nachdem der Prozess den alten Betrag gelesen hat (aber bevor er den geänderten Betrag von 12.000 in die Datenbank schreibt), liest der Prozess 2 für dieselbe Zeile der Kundentabelle ebenfalls den Wert 10000 aus `umsatz_kumuliert`, um den Betrag um 3.000 zu erhöhen. Prozess 1 führt inzwischen die Änderung mit 12.000 durch. Prozess 2 berechnet neu: $10.000 + 3.000 = 13.000$ und schreibt diesen Wert in die Datenbank zurück; damit überschreibt er die Änderung von Prozess 1. Resultat: `umsatz_kumuliert` enthält den Wert 13.000, obwohl zwei Umsätze verbucht wurden und damit der richtige Wert 15.000 hieße. Abbildung 2.1 stellt die beiden Transaktionen und einen möglichen Zeitablauf *ohne* Transaktionsmanagement dar.

Das lost update Problem beschreibt eine Dateninkonsistenz, die in der Datenbank selbst herbeigeführt würde, wenn das Datenbanksystem keine entsprechenden Maßnahmen zur

Abb. 2.1 Lost Update



Konsistenzsicherung durchführen würde. Die betroffenen Daten wären für alle Benutzer, die sie künftig verarbeiten, falsch und somit unbrauchbar. Ein Benutzer einer SQL-Datenbank kann sich aber darauf verlassen, dass solche inkosistenten Zustände nicht auftreten.

Die folgenden Beispiele beschreiben Situationen, in denen die Daten selbst korrekt und konsistent sein können, und wo ein Benutzer trotzdem falsche Auswertungsergebnisse erhält.

Situation 2: Inkonsistente Analyse Wir nehmen an, Benutzer A will die Bestandswerte der Lagerbestände auf Basis der Tabelle `artikel` auswerten, und zwar für jedes Lager einzeln. Dazu gibt er folgende Anweisung

```
SELECT lagerplatz, sum(listenpreis * bestand)
FROM  artikel
GROUP BY lagerplatz
```

Während das Datenbanksystem den Wert für Lager 2 bereits ermittelt hat und nun die Berechnungen für das Lager 4 vornimmt, möchte ein zweiter Benutzer einen Artikel aus Lager 2 nach Lager 4 verlegen. Hierzu gibt er folgende Anweisung

```
UPDATE artikel
  SET  lagerplatz = 4
  WHERE artikel_nr = 'L005'
```

ein. In der Gesamtsumme ist nun der Bestandswert des Artikels mit der Artikelnummer `L005` zweimal enthalten, da er einmal als Teil von Lager 2 und danach noch einmal als Teil von Lager 4 behandelt wurde.

Situation 3: „Dirty Read“ Der Ablauf in beschreibt eine Situation, in der Prozess A Daten ändert, diese Änderung aber später zurücknimmt, während Prozess B die geänderten

Tab. 2.1 Dirty Read

Zeitpunkt	Prozess A	Prozess B
1	`UPDATE kunde`	
	`SET umsatz = umsatz + 1000`	
2		`SELECT SUM(umsatz)`
		`FROM kunde`
3	`ROLLBACK`	

Tab. 2.2 Nonrepeatable Read

Zeitpunkt	Prozess A	Prozess B
1	`UPDATE kunde`	
	`SET umsatz =` `  umsatz + 1000`	
	`WHERE kunden_nr=101`	
2		`SELECT AVG(umsatz)`
		`INTO durchschnitt`
		`FROM kunde`
3	`UPDATE kunde`	
	`SET umsatz = umsatz - 500`	
	`WHERE kunden_nr = 101`	
4	`COMMIT`	
5		`SELECT`
		`AVG((umsatz-durchschnitt)^2)`
		`INTO varianz`
		`FROM kunde`

Daten in seine Berechnung einbezieht, bevor diese in der Datenbank dauerhaft gemacht werden. Die Berechnung wird dadurch falsch. Diese Situation wird als „dirty read“ oder „read uncommitted“ bezeichnet (Tab. 2.1).[8]

Situation 4: „Nonrepeatable Read“ Diese Situation (Tab. 2.2) tritt auf, wenn eine Transaktion A einen Wert mehrfach verändert, und diese Änderung mit COMMIT dauerhaft macht, während eine andere Transaktion den Wert in irgendeinem Zwischenzustand in eine Auswertung einbezieht. Im folgenden Beispiel wird von A erst der Umsatz eines Kunden erhöht, und anschließend – warum auch immer – diese Erhöhung um die Hälfte korrigiert. B ermittelt die Varianz vom Durchschnittsumsatz in zwei Schritten, zwischen denen sich die Berechnungsgrundlage geändert hat.

[8] Genauer wird diese Situation in [SaSH10, S. 382] beschrieben.

Tab. 2.3 Phantom Read

Zeitpunkt	Prozess A	Prozess B
1	`UPDATE artikel`	
	`SET lagerplatz = 2`	
	`WHERE artikel_nr = 'G001'`	
2		`SELECT MAX(listenpreis)`
		`INTO maxpreis`
		`FROM artikel`
		`WHERE lagerplatz = 7`
3	`COMMIT`	
4		`SELECT *`
		`FROM artikel`
		`WHERE listenpreis = maxpreis`

Situation 5: „Phantom Read“ Diese Situation (Tab. 2.3) tritt auf, wenn in einer Abfrage Sätze betroffen sind, und bei einer weiteren Abfrage mit denselben Kriterien neue Sätze dazu kommen oder verschwinden. Dieses kann aufgrund von Einfügungen oder Löschungen passieren oder weil Sätze bezüglich eines Auswahlkriteriums in der WHERE-Klausel eine andere Zuordnung erfahren.

In dem angegebenen Beispiel ist der Artikel ‚G001‘ zum Zeitpunkt 2 (da UPDATE noch nicht mit COMMIT abgeschlossen) noch auf dem Lagerplatz 7 und sein Preis wird als maxpreis zwischengespeichert. Zum Zeitpunkt 4 ist er aber auf Lagerplatz 2 und somit wird in diesem Fall kein Artikel gefunden, der diesen Preis hat.

2.4 Isolation Level in Transaktionen

Die SQL Norm definiert zur Steuerung von Transaktionen so genannte „Isolation-Level“. Dabei wird dem Datenbanksystem mitgeteilt, welche möglichen *Wirkungen* nebenläufiger Transaktionen zulässig sind bzw. ausgeschlossen werden sollen. Es wird dem Benutzer von SQL also keine „technische“ Spezifikation der erforderlichen Sperre unter Benutzung der Begriffe S-Lock und X-Lock abverlangt[9] – das DBMS muss selbst den Typ von Sperre ermitteln, der die Benutzeranforderungen realisiert.

Folgende Anweisung stellt SQL zur Verfügung, um *vor* einer Transaktion deren Eigenschaften festzulegen:

[9] Diese behandeln wir im nächsten Abschnitt.

```
SET TRANSACTION optionsliste
```

Dieselbe Optionsliste kann auch als Parameter der Anweisung

```
START TRANSACTION
```

übergeben werden.

Die Optionsliste umfasst einen Zugangsmodus („access mode“) und den Isolation-Level. Der Zugangsmodus wird mit `READ ONLY` oder `READ WRITE` angegeben und legt fest, ob die Transaktion nur Lesevorgänge enthält oder auch Schreibvorgänge. Im ersten Fall sind Änderungen auf temporäre Tabellen beschränkt. Für den Isolation-Level können folgende Werte eingetragen werden:

```
isolation level ::=  READ UNCOMMITTED
                   | READ COMMITTED
                   | REPEATABLE READ
                   | SERIALIZABLE
```

Die strengste Anforderung wird mit `SERIALIZABLE` spezifiziert. Nur sie gewährleistet die volle Erfüllung aller Eigenschaften einer Transaktion, die mit dem ACID-Prinzip umschrieben werden. Die anderen Werte definieren, welche Abweichungen davon zulässig sind.

Wir bezeichnen mit A und B zwei nebenläufige Transaktionen.

- `READ UNCOMMITTED`
  Hier kann A Tupel lesen, die von einer zweiten Transaktion B geändert werden, auch wenn B diese Änderungen noch nicht durch `COMMIT` dauerhaft an die Datenbank übergeben hat oder sie sogar mit `ROLLBACK` zurücknimmt. Dies wird auch mit dem Begriff *„dirty read“* umschrieben.

- `READ COMMITTED`
  Bei dieser Variante ist das *dirty read* ausgeschlossen. A sieht nur solche Daten, die von B per COMMIT an die Datenbank übergeben werden. Es ist dann aber möglich, dass zwei identische Lesevorgänge innerhalb von A für dasselbe Tupel zu unterschiedlichen Ergebnissen führen. Der Lesevorgang innerhalb von A ist also nicht beliebig wiederholbar (*„non repeatable read“*).

- `REPEATABLE READ`
  Letzteres ist bei diesem Isolation-Level ausgeschlossen. Es kann aber noch der Fall eintreten, dass B ein neues Tupel in die von A gelesene Tabelle einfügt, so dass ein zweiter Lesevorgang innerhalb A ein Tupel findet, das vorher nicht vorhanden war (*„phantom read“*) – oder umgekehrt.

- `SERIALIZABLE`
  Hier sehen alle Auswertungen innerhalb der Transaktion dieselben Daten.

Tab. 2.4 Definition der Isolation-Level

Isolation-Level	dirty read	nonrepeatable read	Phantom
`READ UNCOMMITTED`	J	J	J
`READ COMMITTED`	N	J	J
`REPEATABLE READ`	N	N	J
`SERIALIZABLE`	N	N	N

Eine Übersicht über die Isolation-Level und die von ihnen erlaubten Verstöße gegen das ACID-Prinzip gibt Tab. 2.4 wieder.

Zur Vermeidung der inkonsistenten Analyse muss der Isolation-Level `SERIALIZABLE` angefordert werden.

Bei *SQL Anywhere* werden die Isolation-Level nahe an der Standardvorgabe mit der Anweisung

```
SET OPTION ISOLATION_LEVEL = zahl
```

eingestellt, wobei für `zahl` die Werte 0 bis 3 für die Einstellungen `READ UNCOMMITTED` bis `SERIALIZABLE` anzugeben sind. Bei ORACLE muss der Benutzer selbst dafür sorgen, dass die entsprechenden Resultate erreicht werden, indem er mit der Anweisung `LOCK TABLE` explizit Sperren setzt.

2.5 Sperrmechanismen

Die einfachste Lösung bestünde für alle Fälle darin, die zu verändernde Tabelle vollständig für andere Benutzer zu sperren und erst nach der Beendigung der Arbeit wieder freizugeben. Dieses Verfahren ist sehr radikal – es läuft auf eine zeitweise Aussetzung des Mehrbenutzerbetriebs hinaus – und würde zu völlig unakzeptablen Wartezeiten bei der gemeinsamen Nutzung einer Datenbank führen. Es ist aber für die Gewährleistung der Konsistenz lediglich erforderlich, dass kollidierende Zugriffe auf die gerade zu verändernden Objekte verhindert werden.

In der lost update Situation wäre es sinnvoll, dass Prozess 1 die zu aktualisierende Zeile in der Tabelle `kunde` mit einer Sperre belegt, so dass sie ihm für die Zeitdauer seiner Transaktion exklusiv zur Verfügung steht. Prozess 2 müsste dann so lange warten, bis Prozess 1 sein `COMMIT` zur Beendigung der Transaktion ausgeführt hat, und könnte dann erst die – nun schon aktualisierten – Daten lesen und verändern. Prozess 2 könnte aber inzwischen einen anderen Kunden bearbeiten, ohne dabei in Konflikt mit Prozess 1 zu geraten.

Eine Sperre für Situation inkonsistente Analyse müsste wie folgt wirken: Während A seine Salden bildet, müsste jegliche Veränderung an den Werten der saldierten Spalte für die gesamte Tabelle `artikel` verhindert werden. Welche Operationen wären ohne

Schaden von anderen Benutzern ausführbar? Unproblematisch ist das gleichzeitige Lesen der Tabelle. Auch Veränderungen an Werten (Updates) könnten vorgenommen werden, sofern die Spalten, die für die Auswertung von A gerade relevant sind, nicht verändert werden. Verhindert werden müssen daher nur Änderungsoperationen, bei denen die Spalten `lagerplatz`, `listenpreis` und `bestand` im Wert verändert werden. Die Änderung der Artikelbezeichnung wäre hingegen unproblematisch.

In der Situation nonrepeatable read müsste B entweder jegliche Änderung in den für die Summe relevanten Umsatzdaten unterbinden oder solange warten, bis A seine Transaktion beendet hat. Oder aber B entscheidet, dass für einen statistischen Überblick über die Umsatzdaten auf kleine Abweichungen nicht ankommt und toleriert ein möglicherweise falsches Ergebnis. Eine weitere Möglichkeit wäre, die gelesenen Datensätze bis zum Ende der Transaktion lokal zu kopieren und weitere Auswertungen auf der Kopie zu machen.[10]

Granularität von Sperren Die dargestellten Sperrmöglichkeiten ergeben sich aus der logischen Sicht der Dinge. Je feiner die „Granularität" der Sperrung, das heißt, je begrenzter die Anzahl der von der Sperrung betroffenen Datenbankobjekte ist, desto geringer sind die Einschränkungen konkurrierender Zugriffe. Umso höher ist aber der Verwaltungsaufwand, den das DBMS zu leisten hat. Das bedeutet sowohl Ressourcenverbrauch (Hauptspeicher und Plattenzugriffe) als auch merkbare Einbußen der Performanz, so dass alle praktisch vorhandenen Lösungen auf Kompromisse hinauslaufen. Dabei muss aber zuverlässig gewährleistet werden, dass die Granularität der Sperrung mindestens so groß ist, dass sie alle von einer Transaktion benötigten Datenobjekte umschließt.

Sperrbare Objekte Bei einer relationalen Datenbank sind Relationen und Tupel die Objekte von Datenmanipulationen, also auch Kandidaten für Sperrungen. Theoretisch sinnvoll wäre auch noch die nächst feinere Granularität, nämlich die Sperrung lediglich einer Spalte eines Tupels. Da sich die Sperrung von Daten auf der physikalischen Schicht eines Datenbanksystems abspielt, ist aber nicht jede Granularität mit vertretbarem Aufwand realisierbar. In der Realität finden sich außer Sperrmöglichkeiten für Tabellen oder Tupel auch noch Zwischenformen, bei denen größere Speichereinheiten (Seiten) im Ganzen dem allgemeinen Zugriff zeitweise entzogen werden. Davon sind dann alle Tupel betroffen, die physikalisch auf derselben Seite abgelegt sind.

Typen von Sperren Zwei Haupttypen von Sperren (englisch "*locks*") sind zu unterscheiden (wir gebrauchen hier die englischen Begriffe):

- „exclusive locks" (X-Lock)
- „shared locks" (S-Lock)

[10] Hier sieht man auch, dass diese Technik nicht gegen phantom read wirken würde.

S-Locks erlauben die gleichzeitige Bearbeitung desselben Objekts durch eine Transaktion, die ebenfalls ein S-Lock anfordert. X-Locks entziehen das Objekt dem Zugriff jeglicher anderen Transaktion. Objekte können Zeilen einer Tabelle oder ganze Tabellen sein.

Das Setzen von Sperren sollte für den Benutzer transparent sein und ohne sein Zutun erfolgen, wenn die Konsistenz der Datenbank dies bei Schreib- oder Lesevorgängen erfordert. Das DBMS setzt also implizite Sperren unmittelbar vor jedem Zugriff auf ein Datenobjekt. Der Zugriff kann erst dann erfolgen, wenn die Sperre erfolgreich gesetzt werden konnte.

- Jede schreibende Transaktion (`UPDATE, INSERT, DELETE`) beantragt ein X-Lock auf die betreffenden Objekte.
- Eine lesende Operation setzt per se keine Sperre.

Weitere Sperren (Typ S oder X) können explizit vom Benutzer oder vom Anwendungsprogramm gesetzt werden. Insbesondere, wenn Probleme wie die inkonsistente Analyse vermieden werden sollen, muss dies auch für lesende Zugriffe erfolgen.

Es sind folgende Fälle zu unterscheiden, wobei wir von einer Transaktion (A) ausgehen, die eine Sperre angefordert hat, während eine zweite Transaktion (B) versucht, ebenfalls eine Sperre für dasselbe Objekt anzufordern:

1. A hat S-Lock gesetzt:
   - B kann ebenfalls ein S-Lock setzen und den Zugriff ausführen.
   - B kann kein X-Lock setzen. Der beabsichtigte Zugriff kann nicht ausgeführt werden.
2. A hat ein X-Lock gesetzt:
   - B kann weder ein S-Lock noch ein X-Lock auf dasselbe Objekt setzen.

Beide Typen von Sperren werden jeweils am Ende der Transaktion, in der sie angefordert worden sind, gelöscht – im Allgemeinen also durch `COMMIT` oder `ROLLBACK`. Aber auch jede andere Art der Beendigung (`LOGOUT, EXIT`, Systemabsturz etc.) von Transaktionen sollte gesetzte Sperren aufheben, wenngleich dies nicht bei allen SQL-Implementierungen der Fall ist.

Für die Entwicklung von Anwendungsprogrammen sind Sprachkonstrukte erforderlich, um auf die Sperrung Einfluss zu nehmen. Dazu gibt es in verschiedenen DMBS den Befehl LOCK TABLE, der die von uns vorgestellten Sperrmechanismen abbildet. In der SQL Norm hingegen gibt es diesen Befehl nicht, hier ist eine implizite Angabe des Transaktionslevels möglich. Wir kommen weiter unten darauf zurück.

Zur Vermeidung von „lost update“ müsste jede der beteiligten Transaktionen bereits beim Lesen vorsorglich eine exklusive Sperre für die betreffende Zeile anfordern. B würde dann schon vor dem Lesevorgang in den Wartezustand gesetzt und könnte erst dann fortfahren, wenn A mit `COMMIT` die Transaktion beendet hat. Diese vorsorgliche Sperre muss im Allgemeinen aber explizit durch den Benutzer oder das Anwendungsprogramm

erfolgen, da das DBMS bei einem Lesevorgang innerhalb einer Dialoganwendung ja noch nicht „wissen“ kann, ob diesem ein Schreibvorgang folgt. In unserer obigen Formulierung könnte allerdings die Notwendigkeit eines X-Locks vom DBMS selbst noch erkannt werden, da mit der SQL-Anweisung `UPDATE` immer Lese- und Schreibvorgänge verbunden sind.

Die Vermeidung einer inkonsistenten Analyse erfordert die explizite Anforderung einer Sperre vom Typ „S“ durch A, die auf die gesamte Tabelle wirkt, auch wenn es sich „nur“ um eine lesende Transaktion handelt.

2.6 Explizite Sperrung mit LOCK TABLE

Die explizite Anforderung einer Sperre mit der Anweisung `LOCK TABLE` ist im Standard nicht vorgesehen. Es gibt sie aber beispielsweise bei ORACLE. Die Anweisung erfordert einen direkten Bezug des Benutzers auf die Sperrtypen S bzw. X und das zu sperrende Objekt (Granularität). Die Syntax lautet in etwas vereinfachter Form:

```
LOCK TABLE tabellenliste IN lockmodus MODE
```

Folgende Lockmodi werden angeboten:

- `EXCLUSIVE`
  erlaubt nur Abfragen an die Tabelle durch andere Benutzer. Jegliche andere Aktivität (`UDPATE, DELETE, INSERT`) wird unterbunden. Es können auch keine anderen Sperren gleichzeitig existieren.
- `SHARE`
  erlaubt Abfragen und andere S-Locks, aber keine Datenänderungen und exklusiven Sperren. Die Daten in der Tabelle können nicht verändert werden.

Die Wirkung einer Sperre kann auf einzelne Tupel eingeschränkt werden, wenn der Zusatz ROW angebracht wird.

- `ROW SHARE` (oder `SHARE UPDATE`):
  erlaubt den konkurrierenden Zugriff und die gleichzeitige Änderung anderer Zeilen durch andere Benutzer. Es verhindert jegliche exklusive Sperre auf die gesamte Tabelle. Mehrere Sperren dieses Typs können zugleich existieren.
- `ROW EXCLUSIVE`:
  arbeitet wie `ROW SHARE`, verbietet aber auch gleichzeitige Sperren auf die gesamte Tabelle im Share-Modus. Mehrere Sperren dieses Typs können gleichzeitig und neben `ROW SHARE` bestehen.
- `SHARE ROW EXCLUSIVE`:
  Hier sind gleichzeitig lediglich Sperren im Modus `ROW SHARE` erlaubt.

Der Modus ROW SHARE ist im Hinblick auf eine möglichst geringe Behinderung anderer Transaktionen der effektivste. Wir hatten für die Bewältigung der Situation 1 (*lost update*) erwähnt, dass eine vorsorgliche Anmeldung einer Exklusivsperre bereits beim Aufsuchen der betreffenden Zeile aus der Datenbank erforderlich wäre. Dies ist durch eine besondere Anweisung möglich, die die zu ändernden Datensätze reserviert. Sie sieht wie folgt aus:

```
SELECT  spaltenliste
  FROM  tabellenausdruck
 [WHERE bedingung]
  FOR UPDATE OF spaltenliste
```

Das Phänomen *dirty read* wird übrigens bei ORACLE dadurch umgangen, dass alle Lesevorgänge auf die Before-Image-Datei umgeleitet werden, soweit sie Daten betreffen, die von gerade laufenden Transaktionen verändert werden.

2.7 Deadlock

Wenn mehrere Transaktionen wechselseitig auf die Freigabe von Ressourcen warten, spricht man von einer *Verklemmung* oder einem *Deadlock*. Dies kommt dadurch zustande, dass (mindestens zwei) Transaktionen zyklisch jeweils auf die Freigabe einer Ressource durch eine andere warten, um selbst mit COMMIT abschließen zu können. Da aber erst mit COMMIT oder ROLLBACK die gesetzten Sperren aufgehoben werden, liegt ein Widerspruch vor, der nur durch Abbruch einer der beteiligten Transaktionen aufgehoben werden kann. Eine typische Situation zeigt folgendes Szenario, bei dem zwei Tabellen betroffen sind:

Prozess A	Prozess B
`LOCK TABLE kunde` `  IN EXCLUSIVE MODE;`	`LOCK TABLE auftrag` `  IN EXCLUSIVE MODE;`
`. . .`	`. . .`
`LOCK TABLE auftrag`	`LOCK TABLE kunde`

Prozess A muss auf die Freigabe von auftrag warten, während B auf die Freigabe von kunde durch A warten muss. Beide Transaktionen versperren sich gegenseitig den Zugriff auf Ressourcen.

Automatische Deadlock-Behandlung Es gehört zu den Aufgaben eines DBMS, solche Deadlock-Situationen zu erkennen und einige Sperren zu löschen, damit wenigstens eine der Transaktionen abgeschlossen werden kann. Das Mindeste ist ein automatisches Rollback gegen eine der beteiligten Transaktionen und eine entsprechende Mitteilung an den Benutzer bzw. das Anwendungsprogramm. Von diesem Rollback sollte nach Möglichkeit diejenige Transaktion betroffen sein, die noch am wenigsten fortgeschritten, also am leichtesten zu wiederholen ist.

Optimistische Verfahren zur Steuerung der Nebenläufigkeit Bei optimistischen Kontrollmethoden wird davon ausgegangen, dass die Wahrscheinlichkeit auftretender Konflikte relativ gering ist, so dass auf die vorsorgliche Sperre ganzer Tabellen oder einzelner Reihen verzichtet wird. Erst im Konfliktfall werden dann Maßnahmen ergriffen, die die Konsistenzerhaltung gewährleisten. Dabei geht es dann im Wesentlichen um ein kontrolliertes Zurücksetzen von Änderungen, verbunden mit einer automatischen Wiederholung der Operationen zu einer anderen Zeit.[11]

2.8 Kompensation statt ACID

Die vielfach praktizierte Realisierung von Anwendungssystemen auf Basis einer Serviceorientierten Architektur[12] (SOA) steht vielfach im Widerspruch zum ACID Prinzip, zumal dann, wenn einzelne Entitäten jeweils durch einen eigenen Service verwaltet werden und dieser autonom und unabhängig von anderen Services sein immer gleiches Verhalten implementiert. Deswegen kann eine Transaktion nur innerhalb jeweils einer Serviceoperation stattfinden, und nicht übergreifend über mehrere Operationen. Werden logisch zusammenhängenden Operationen aber von mehreren Serviceoperationen ausgeführt, dann ist keine Transaktionssteuerung und kein ACID möglich. Hier wird dann anstelle des ROLLBACK das Konzept der Kompensation vorgeschlagen, wobei das Ungeschehenmachen von Änderungen durch eine entgegengesetzte Operation, die natürlich programmiert werden muss, erfolgt. Wir sehen das mit Skepsis und raten nicht dazu, leichtfertig auf das ausgefeilte Transaktionsmanagement eines DBMS zu verzichten.[13]

[11] Genauer bei [Rako07, S. 271].

[12] SOA haben wir am Ende von Kap. 1 kurz eingeführt.

[13] Vgl. den Diskussionsbeitrag in [Unte10].



http://www.springer.com/978-3-642-39002-9

Anwendungsentwicklung mit Datenbanken
Unterstein, M.; Matthiessen, G.
2013, X, 182 S. 20 Abb., Softcover
ISBN: 978-3-642-39002-9