

2.3 Transaktionen

Eine *Transaktion* bezeichnet eine Gruppe von Operationen. Im Kontext von Datenbankmanagementsystemen geht es darum, dass mehrere Datenbankoperationen entweder vollständig und fehlerfrei ausgeführt werden oder gar nicht. Auf keinen Fall darf nur ein Teil einer Transaktion durchgeführt werden, also z. B. die ersten beiden von insgesamt fünf Kommandos.

Warum sind Transaktionen so wichtig?

Das klassische Beispiel für die Notwendigkeit von Transaktionen ist die Umbuchung von einem Konto auf ein anderes. Beispielsweise will Bankkunde A 1.000 EUR auf das Konto von Kunde B überweisen. Die Konten beider Kunden sind in einer Tabelle eines DBMS abgebildet. Im Detail sind die folgenden Datenbankoperationen erforderlich:

1. Befinden sich auf dem Konto von A zumindest 1.000 EUR?
2. Reduziere den Kontostand von A um 1.000 EUR.
3. Erhöhe den Kontostand von B um 1.000 EUR.

Ich denke, ich muss hier nicht ausführen, welche katastrophalen Auswirkungen es für das Vertrauen in die Bank hätte, wenn ein derartiger Prozess nur bis zum zweiten Punkt ausgeführt würde (siehe Abbildung 2.3). Die 1.000 EUR würden auf Nimmerwiedersehen verschwinden. Der finanzielle Nutznießer wäre die Bank – zumindest bis dieses Fehlverhalten öffentlich bekannt würde.

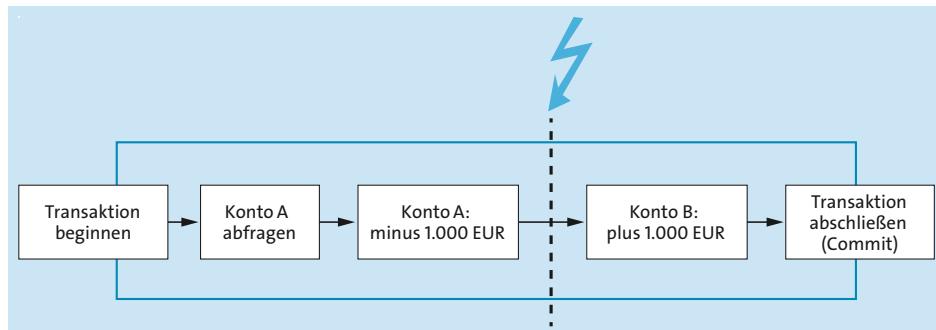


Abbildung 2.3 Das Datenbankmanagementsystem muss garantieren, dass die Transaktion vollständig ausgeführt wird – oder gar nicht. Auch ein äußeres Ereignis wie ein Blitzschlag darf nicht dazu führen, dass nur ein Teil der Kommandos innerhalb einer Transaktion ausgeführt wird.

Gerne übersehen wird, dass auch die Grenze zwischen Punkt 1 und 2 kritisch ist: Vielleicht versucht genau zu dem Zeitpunkt, zu dem A das Geld an B überweisen möchte, die Versicherung C die ihr zustehende monatliche Versicherungsprämie abzubuchen. Das Konto von A ist aber nicht ausreichend für beide Aktionen gedeckt. Das DBMS muss sicherstellen, dass die eine oder die andere Aktion vollständig durchgeführt

wird – nicht aber beide. Deswegen kann es am Ende einer Transaktion vorkommen, dass das DBMS meldet, dass die Transaktion gescheitert ist (im hier skizzierten Fall also deswegen, weil die andere Transaktion schneller war bzw. vorher abgeschlossen wurde und das Konto nun nicht mehr gedeckt ist).

Transaktionen sind schließlich ein wichtiges Hilfsmittel zur Absicherung von Code in Client-Programmen: SQL bietet die Möglichkeit, eine einmal gestartete Transaktion jederzeit abzubrechen. Damit werden alle bisher im Rahmen der Transaktion ausgeführten Kommandos widerrufen.

Um beim Eingangsbeispiel zu bleiben: Nehmen Sie an, die Umbuchung würde per Onlinebanking erfolgen. Aber genau während Kunde A seine Onlinebanking-App bedient, fällt das Internet aus. (Vielleicht befindet sich A in einem Zug, das gerade durch einen Tunnel fährt.) Die Onlinebanking-App hat die ersten beiden Kommandos bereits an den Datenbank-Server gesendet, das dritte Kommando aber nicht. Auch in diesem Fall scheitert die gesamte Transaktion, weil das DBMS vergebens auf die Bestätigung der Transaktion wartet (also auf das COMMIT-Kommando, siehe unten).

Transaktionen in SQL

In der Structured Query Language gibt es zur Steuerung von Transaktionen eigene Kommandos:

- ▶ START TRANSACTION oder BEGIN starten eine neue Transaktion.
- ▶ COMMIT schließt die Transaktion ab.
- ▶ ROLLBACK oder ABORT widerrufen alle Kommandos der Transaktion.

Je nach DBMS gibt es darüber hinaus weitere Kommandos, um Transaktionen zu verschachteln oder um innerhalb von Transaktionen sogenannte *Savepoints* zu setzen. Daraus ergibt sich die Möglichkeit, eine Transaktion bis zum Savepoint zu bestätigen, alle ab diesem Punkt durchgeführten Kommandos aber zu widerrufen. Eine Menge praktischer Beispiele zum Einsatz der Transaktionskommandos folgen in [Kapitel 12, »Transaktionen«](#).

Auch wenn es an dieser Stelle ein Vorgriff auf Teil III dieses Buchs ist, also auf »Structured Query Language (SQL)«, möchte ich hier kurz in Pseudocode skizzieren, aus welchen SQL-Kommandos sich die Transaktion für die Umbuchung zusammensetzt (siehe [Abbildung 2.3](#)). Dabei gehe ich davon aus, dass Kunde A die ID 5432 hat, Kunde B die ID 8787.

Die in Blockbuchstaben formulierten Kommandos entsprechen regulärem SQL-Code. if, else und print deuten Python-ähnlichen Code der Onlinebanking-App an, wobei die Programmiersprache hier nicht relevant ist. @commit_ok ist eine Statusvariable, die angibt, ob die letzte Transaktion erfolgreich abgeschlossen wurde oder nicht. (In der Praxis gibt es so eine Variable nicht. Stattdessen tritt bei COMMIT ein Fehler auf, wenn

das DBMS die Transaktion aus irgendeinem Grund nicht ausführen kann. Ich wollte den Code hier aber nicht mit try/catch oder einer anderen Fehlerabsicherung unnötig kompliziert machen.)

```
BEGIN;
SELECT balance FROM accounts WHERE id=5432 INTO @currentbalance;
if @currentbalance >= 1000:
    UPDATE accounts SET balance = balance - 1000 WHERE id = 5432;
    UPDATE accounts SET balance = balance + 1000 WHERE id = 8787;
    COMMIT;
    if @commit_ok:
        print("Transaktion durchgeführt.")
    else:
        ROLLBACK;
        print("Transaktion gescheitert, versuchen
              Sie es später wieder.")
else:
    ROLLBACK;
    print("Transaktion gescheitert, zu wenig Geld
          auf dem Konto.")
```

Sonderfälle und Varianten

Zu gewöhnlichen Transaktionen gibt es einige Varianten, die aber nicht jedes DBMS unterstützt:

- ▶ **Verschachtelte Transaktionen** liegen vor, wenn innerhalb einer Transaktion eine weitere Transaktion (wenn Sie so wollen: eine Subtransaktion) gestartet werden kann. Die innere Transaktion kann dann bestätigt oder widerrufen werden. Aber erst wenn auch die äußere Transaktion erfolgreich abgeschlossen wird, werden sämtliche Änderungen tatsächlich in der Datenbank gespeichert. (Diese Vorgehensweise wird auch als *geschlossene Transaktion* bezeichnet. Demgegenüber gibt es auch *offene Transaktionen*, bei denen der Abschluss einer inneren Transaktion sofort für andere Transaktionen wirksam wird. Das widerspricht allerdings den im Abschnitt 2.4, »Datensicherheit und ACID«, vorgestellten ACID-Regeln.)

Je nach DBMS ist auch eine mehrstufige Verschachtelung zulässig. Auf jeder Ebene können Transaktionen durchgeführt werden. Zu einer endgültigen Speicherung kommt es wiederum erst, wenn die äußere Transaktion bestätigt wird.

- ▶ Von **verteilten Transaktionen** ist die Rede, wenn mehr als ein DBMS beteiligt ist. Dazu käme es beispielsweise, wenn Sie eine Geldüberweisung von einer Bank auf eine andere durchführen wollten und beide Banken jeweils ihre eigene IT-Infrastruktur nutzen.

Die Durchführung verteilter Transaktionen erfordert einen Transaktionsmanager. Das ist ein Programm, das eine Ebene über der von SQL bzw. der beteiligten DBMS agiert. Insofern sind verteilte Transaktionen trotz ihrer großen praktischen Bedeutung außerhalb der Reichweite dieses Buchs.

Die meisten gängigen Transaktionsmanager entsprechen dem Standard *X/Open XA* (*eXtended Architecture*, kurz XA). Diese Spezifikation wurde 1991 von der Unix-nahen *X/Open Company* entwickelt. XA sieht für verteilte Transaktionen einen Zwei-Phasen-Commit vor. Dazu sind allerdings Locking-Mechanismen erforderlich, die sich negativ auf die Performance auswirken und fehleranfällig sind.

Transaktionen parallel ausführen

Aus Sicht des DBMS wäre es am einfachsten, eine Transaktion nach der anderen abzuarbeiten. Diese serielle Vorgehensweise ist aber äußerst ineffizient, wenn mehrere Clients parallel zueinander Transaktionen starten.

Die parallele Verarbeitung von Transaktionen birgt leider eine großes Fehlerpotenzial und ist nur dann sicher, wenn die Datenbank ACID-konform ist. Was das bedeutet, verrate ich Ihnen im nächsten Abschnitt.

2.4 Datensicherheit und ACID

Ein entscheidendes Argument für die Einführung einer relationalen Datenbank ist die Sicherheit der Daten. Was aber bedeutet »sicher«? Es gibt je nach Kontext viele Spielarten, inwiefern Ihre Daten sicher sein können:

- ▶ Sicher vor einem Datenverlust durch eine defekte Festplatte oder SSD.
- ▶ Sicher vor einem Datenverlust, wenn der Server-Raum durch ein Feuer zerstört wird.
- ▶ Sicher vor einer gezielten Manipulation durch einen Mitarbeiter, der sich bereichern will.
- ▶ Sicher vor unabsichtlichen Änderungen durch Programmierfehler in einem Client-Programm.
- ▶ Sicher vor Inkonsistenzen (dass also z. B. eine Bestellung im Onlineshop auf eine Produktnummer verweist, die es in der Produktetabelle gar nicht gibt).
- ▶ Sicher vor Konflikten beim parallelen/gleichzeitigen Zugriff auf die Daten durch mehrere Clients.

Das Thema Sicherheit werde ich daher in diesem Buch immer wieder aus unterschiedlichen Sichtweisen behandeln. Besonders relevant sind die folgenden Abschnitte bzw. Kapitel:

- ▶ Abschnitt 5.2, »Foreign Keys (Fremdschlüssel)«
(Einhaltung der referenziellen Integrität)
- ▶ Kapitel 15, »Benutzerverwaltung«
- ▶ Kapitel 16, »Logging und Backups«
- ▶ Kapitel 17, »Replikation und High Availability«

An dieser Stelle stehen die sperrigen Begriffe *Atomicity*, *Consistency*, *Isolation* und *Durability* im Vordergrund. Sie ergeben zusammen das Kürzel ACID, das als eine zentrale Eigenschaft von DBMS gilt. Wenn ein Datenbankmanagementsystem ACID-konform ist, dann kann auch bei der gleichzeitigen Ausführung mehrerer Transaktionen (Stichwort: die Buchung des letzten noch freien Platzes für einen Flug) nichts schiefgehen. ACID hat also damit zu tun, wie gut ein DBMS Datenbankoperationen parallelisieren kann. Wie Sie noch sehen werden, muss dabei oft der richtige Kompromiss zwischen maximaler Sicherheit und bestmöglicher Performance gesucht werden. Beide Wünsche lassen sich gleichzeitig kaum erfüllen.

Atomicity

Atomicity bzw. Abgeschlossenheit oder Atomarität bedeutet, dass das DBMS eine Transaktion entweder ganz oder gar nicht ausführt (»alles oder nichts«). Der gesamte Vorgang ist unteilbar. Atomicity ist somit die Grundvoraussetzung für Transaktionen in Datenbankmanagementsystemen.

Das DBMS muss in der Lage sein, Atomicity selbst in Extrempfälten zu garantieren. Auch wenn bei den drei Schritten einer Überweisung von Konto A auf B zwischen dem zweiten und dem dritten Schritt der Blitz einschlägt, der Strom ausfällt, die Festplatte versagt etc., muss sichergestellt werden, dass die gesamte Transaktion blockiert und nicht etwa nur Schritt 1 und 2 wirksam wird (siehe auch Abschnitt 2.3, »Transaktionen«, sowie Abbildung 2.3).

Auf technischer Ebene wird die Forderung nach Atomicity in der Regel durch einen zweiphasigen Speicherprozess sichergestellt. Die Änderungen, die sich durch die Transaktion ergeben, werden zuerst im sogenannten *Transaktions-Log* gespeichert, also in einer eigenen Datei oder einem speziellen Bereich innerhalb der Datenbankdatei.

Nach einem COMMIT-Kommando erhält das Client-Programm erst dann die Bestätigung für die erfolgreiche Durchführung der Transaktion, wenn diese im Transaktions-Log gespeichert wurde. In einem zweiten Schritt wird die Änderung in den Dateien des DBMS durchgeführt. Sollte es dabei zu einem Problem kommen (Stromausfall), kann die Transaktion beim Neustart des DBMS zu einem späteren Zeitpunkt nachgeholt werden; alle dafür erforderlichen Informationen befinden sich ja im Transaktions-Log.

Tritt der Stromausfall auf, *bevor* der Vorgang im Transaktions-Log gespeichert wurde, bekommt das Client-Programm keine positive Rückmeldung. In diesem Fall ist die Transaktion eben gescheitert. Zu dieser Situation kann es immer kommen; jedes Client-Programm muss damit zurechtkommen.

Der entscheidende Punkt bei Atomicity ist also nicht die Garantie, dass eine Transaktion durchgeführt werden kann (die gibt es nicht), sondern die Garantie, dass die Transaktion bei einer positiven Rückmeldung als Ganzes verarbeitet wird, bei einer negativen Rückmeldung dagegen überhaupt nicht. Dazwischen gibt es keinen Graubereich.

Consistency

Die Forderung nach *Consistency* verlangt, dass eine Datenbank, die sich vor einer Transaktion in einem konsistenten Zustand befunden hat, danach ebenfalls wieder konsistent ist. Wenn diese Bedingung nicht erfüllt werden kann, muss das DBMS die Transaktion ablehnen. (Das Client-Programm erhält also eine Fehlermeldung, dass die gesamte Transaktion nicht durchgeführt werden konnte.)

Was bedeutet nun aber »konsistent«? Die Tabellen der Datenbank müssen bestimmte Regeln erfüllen:

- ▶ Jeder Datensatz einer Tabelle muss anhand einer Spalte eindeutig identifizierbar sein. In dieser Primärschlüsselspalte darf es also keine Doppelgänger geben. (Der Primärschlüssel kann auch aus mehreren Spalten zusammengesetzt sein. Eine Menge Details dazu, was der Primärschlüssel ist, wozu er dient und wie er angewendet wird, folgen in Kapitel 5, »Primary Keys, Foreign Keys und referentielle Integrität«.)
- ▶ In relationalen DBMS können Sie eine Spalte mit einem Unique-Index ausstatten. Das bedeutet, dass es auch in dieser Spalte keine Doppelgänger geben darf.
- ▶ Alle Werte einer Spalte müssen den vorgegebenen Datentyp und gegebenenfalls eine maximale Länge einhalten. Wenn eine Spalte beispielsweise den Datentyp VARCHAR(100) hat, dann darf es nicht vorkommen, dass dort eine Zeichenkette mit mehr als 100 Zeichen gespeichert wird. Wenn eine Spalte den Datentyp DATE hat, muss das gespeicherte Datum gültig sein, darf also beispielsweise nicht 30.2.2022 lauten.
- ▶ Es besteht die Möglichkeit, für die Spalten einer Tabelle den Wertebereich einzuschränken, z.B. auf Zahlen von 0 bis 100. Dann darf kein Wert diesen Bereich unter- oder überschreiten.
- ▶ Die Spalten einer Tabelle können mit NOT NULL deklariert werden. Dann muss das DBMS sicherstellen, dass in dieser Spalte wirklich kein NULL-Zustand gespeichert wird.

- Schließlich darf es keine Querverweise auf nicht existierende Datensätze geben. Bei diesem Punkt muss ich nochmals auf Abschnitt 5.2, »Foreign Keys (Fremdschlüssel)«, vorgreifen. Sie wissen ja schon, dass relationale Datenbankmanagementsysteme Verknüpfungen zwischen Tabellen erlauben. Wenn Sie also in der Tabelle *orders* die ID eines Kunden und die ID eines Produkts speichern, bedeutet das, dass dieser Kunde ein bestimmtes Produkt bestellt hat. Durch sogenannte *Foreign-Key-Regeln* kann die Datenbank garantieren, dass es den betreffenden Kunden in der *customers*-Tabelle tatsächlich gibt und dass auch das bestellte Produkt mit der angegebenen ID in der *products*-Tabelle enthalten ist.

Wenn es in irgendeiner Tabelle der Datenbank einen Verweis auf einen Datensatz einer anderen Tabelle gibt, der dort gar nicht existiert, dann gilt die Datenbank als *inkonsistent* (siehe Abbildung 2.4).

Tabelle orders				Tabelle products		
id	quantity	customerId	productId	id	name	price
...				1	Spülmaschinen-Tabs	11,36
745	1	348	3	2	Maschineneinreigner	3,98
746	2	725	2	3	Geschirrspülmittel	2,90
747	1	725	17	4	Fensterputzmittel	4,90
748	1	412	1	...		
...				15	Essigreiniger	2,99

??? (id nicht vorhanden)

Abbildung 2.4 Inkonsistente Datenbank (es gibt kein Produkt mit der Nummer 17)

Die Forderung nach Consistency besagt, dass keine Transaktion aus einer vormals konsistenten Datenbank eine inkonsistente Datenbank machen darf.

Keine 404-Fehler mehr

Wer kennt das nicht? Sie klicken auf einen Link. Aber anstatt dass Ihr Webbrowser die gewünschte Seite anzeigt, kommt es zum Fehler 404 (*not found*). Der Link ist also »tot«, die Seite existiert nicht mehr oder hat eine neue Adresse erhalten.

Wenn das gesamte Internet durch ein relationales DBMS abgebildet würde, das die Einhaltung der Consistency-Forderung garantiert, gäbe es beim Klicken auf Links keine 404-Fehler mehr!

Isolation

Das Isolation-Prinzip (im Deutschen oft als »Abgrenzung« bezeichnet) verlangt, dass sich parallel ausgeführte Transaktionen nicht gegenseitig beeinflussen. Um die Bedeutung dieser Forderung zu veranschaulichen, präsentiere ich Ihnen ein weiteres Bankkontobeispiel: Diesmal ist nur ein Konto im Spiel, dafür laufen aber zwei Transaktionen parallel ab:

- ▶ Transaktion 1 beginnt zur Zeit t_0 . Der Kontostand von Matthias beträgt gerade 4.756 EUR. Nun wird das monatliche Gehalt überwiesen. Im Rahmen einer Transaktion wird der aktuelle Kontostand ermittelt. Dieser wird um 2.500 EUR erhöht und zum Zeitpunkt t_2 in der Datenbank gespeichert.
- ▶ Transaktion 2 beginnt zur Zeit t_1 zwischen t_0 und t_2 . Matthias hebt bei einem Bankautomaten 200 EUR ab. Zum Zeitpunkt t_1 beträgt der Kontostand noch immer 4.756 EUR. Im Rahmen der Transaktion wird dieser Wert ermittelt, um 200 EUR vermindert und gespeichert. Die Transaktion endet zum Zeitpunkt t_3 , also ein wenig später als Transaktion 1. Der neu errechnete Kontostand beträgt $4.756 - 200$, also 4.556 EUR. Dieser Betrag wird gespeichert.

Ohne Isolation hätte Matthias gerade 2.500 EUR verloren (siehe Abbildung 2.5). Dieses Fiasko lässt sich zum Glück recht einfach lösen. Transaktion 2 darf nicht starten, bevor Transaktion 1 endet. Sämtliche Transaktionen müssen also serialisiert werden. Transaktion 2 kann erst beginnen, wenn Transaktion 1 fertig ist.

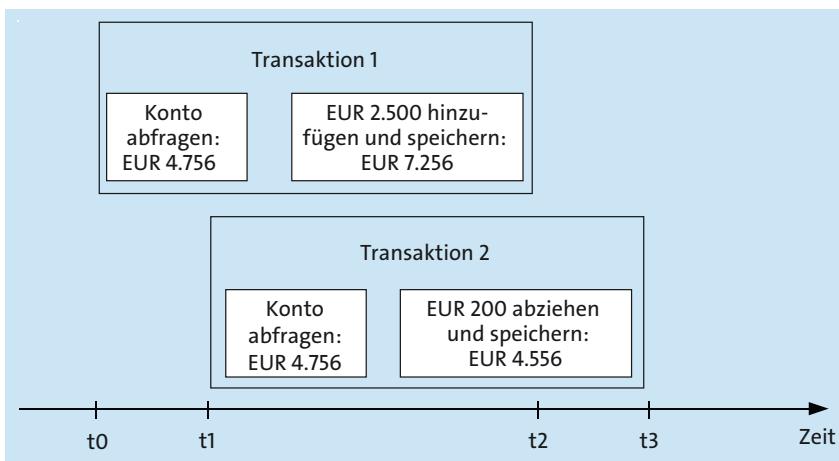


Abbildung 2.5 Wenn ein Datenbankmanagementsystem Transaktionen nicht voneinander isoliert, kann es wie in diesem Fall zur Katastrophe kommen: Der Bankkunde verliert gerade ein ganzes Monatsgehalt.

Wenn eine Lösung allzu einfach ist, gibt es meistens einen Haken. In diesem Fall ist es die Geschwindigkeit: Eine Serialisierung aller Transaktionen würde die maximale Performance des DBMS stark limitieren. (Die Anzahl der pro Sekunde ausführbaren Transaktionen ist ein wichtiges Maß für die Effizienz von DBMS.)

Deswegen wurden in DBMS im Laufe der Zeit diverse Optimierungsmaßnahmen vorgenommen. Die Grundidee ist dabei, einerseits durch Locking-Mechanismen die gegenseitige Beeinflussung durch Transaktionen auszuschließen, andererseits aber das Locking nicht auf die ganze Datenbank anzuwenden, sondern auf möglichst kleine Bereiche davon:

- ▶ Transaktionen, die unterschiedliche Tabellen betreffen, beeinflussen sich gegenseitig normalerweise nicht. Die parallele Ausführung von Transaktionen ist deswegen möglich. (Bei verknüpften Tabellen gilt das allerdings nur mit Einschränkungen.)
- ▶ Selbst bei Transaktionen, die die gleiche Tabelle betreffen, ist Parallelität gefahrlos möglich, wenn unterschiedliche Datensätze betroffen sind. Besonders einfach ist das vorliegende Beispiel zu begründen: Wenn sich die Transaktionen 1 und 2 auf unterschiedliche Kunden beziehen (d. h., Kunde A erhält sein Gehalt, Kundin B hebt bei einem Bankomaten Geld ab), dann spricht nichts gegen eine parallele Ausführung der Transaktionen.

In der Praxis ist das leider nicht immer so einfach: Nehmen Sie an, Sie wollten innerhalb einer Transaktion den Zinssatz von allen Kunden, die mehr als 100.000 EUR auf ihrem Konto liegen haben, verändern. Diese Operation betrifft zwar vermutlich nur einen kleinen Prozentanteil aller Kunden. Es ist aber schwierig, nur diese Datensätze vorübergehend vor Zugriffen durch andere Transaktionen zu sperren. Am einfachsten wäre es in diesem Fall, kurzzeitig die gesamte Tabelle zu blockieren.

Wie die Locking-Mechanismen im Detail durchgeführt werden, hängt stark von der Implementierung des jeweiligen DBMS ab und ist zum Teil ein Firmengeheimnis. Zum Beispiel können zum Locking ganze Gruppen (*Pages*) von Indexeinträgen verwendet werden. Dazu müssen Sie aber verstehen, wie ein Index funktioniert. Ich greife dieses Thema in Abschnitt 6.2, »Index-Interna und B-Trees«, nochmals auf.

Eine andere Strategie besteht darin, die gerade blockierten Datensätze in einer Bitmap zu kennzeichnen. Pro Datensatz, der durch Locking gesperrt ist, kostet das nur ein Bit im Arbeitsspeicher. Es geht hier also um die Frage, wie der erforderliche Arbeitsspeicher für Locking-Operationen minimiert werden kann. Sie sehen schon, sobald man sich ein wenig in die Interna von DBMS einlässt, wird die Sache sehr technisch.

- ▶ Reine Leseoperationen (also SELECT-Kommandos), die nicht in der Folge mit Datenänderungen verbunden sind, können gefahrlos parallelisiert werden. Hier ist aber zu beachten, dass es in der Verantwortung des Softwareentwicklers liegt, Transaktionen korrekt einzugrenzen. Bei der Umbuchung von Konto A auf Konto B (siehe Abbildung 2.3) wäre es ein grober Fehler, wenn die Transaktion nur die beiden Umbuchungen, nicht aber die vorher notwendige Kontoabfrage umfasst.

Isolation Level: Der Kompromiss zwischen Performance und Sicherheit

Bei einigen DBMS können Sie den sogenannten *Isolation Level* einstellen, also gewissermaßen ein Maß dafür, wie exakt das Isolation-Prinzip garantiert werden soll.

Der zugrundeliegende Gedanke ist einfach: Nicht immer bilden die Daten Bankkonten ab, nicht immer ist eine perfekte Trennung jeder Transaktion erforderlich. Manchmal ist mehr Performance wichtiger.

Stellen Sie sich vor, Ihr DBMS speichert die Beiträge eines großen Diskussionsforums. Die exakte Einhaltung der Isolation könnte in diesem Fall garantieren, dass zwei exakt gleichzeitig durchgeführte »Likes« für einen besonders gelungenen Diskussionsbeitrag korrekt verarbeitet werden. In der Praxis wird das zum einen extrem selten vorkommen, zum anderen sind die Konsequenzen einer fehlerhaften Verarbeitung, bei der vielleicht ein »Like« verloren geht, vernachlässigbar.

Es gibt vier im SQL-Standard definierte Isolation Level, nämlich *Serializable* (vollständig ACID-konform), *Repeatable Read*, *Read Committed* und *Read Uncommitted* (am schwächsten, aber auch am schnellsten). In [Kapitel 12, »Transaktionen«](#), gehe ich näher auf diese Level ein und zeige anhand von Beispielen, welche Auswirkungen die verschiedenen Einstellungen haben können.

Durability

Ohne lange Erläuterungen zu verstehen ist die Forderung nach *Durability*, also Dauerhaftigkeit: Wenn ein Datenbankmanagementsystem meldet, dass es eine Transaktion erfolgreich abgeschlossen hat, dann muss es garantieren, dass die betroffenen Daten tatsächlich gespeichert wurden. Und diese Garantie muss in jedem Fall gelten, auch wenn der schon erwähnte Blitz im Rechenzentrum einschlägt, wenn aus anderen Gründen der Strom ausfällt, wenn ein Defekt bei einem Datenträger eintritt oder wenn ein Programm (im schlimmsten Fall: der Datenbank-Server selbst) abstürzt.

Auch für die Durability spielt das bei der Beschreibung des Atomicity-Begriffs bereits erwähnte Transaktions-Log eine Rolle: Jede Änderung wird zuerst in einer speziellen Logging-Datei außerhalb der eigentlichen Datenbank gespeichert. Sobald das Betriebssystem meldet, dass diese Datei wirklich auf dem Datenträger gespeichert wurde (also nach einer *File Synchronization*), bestätigt das DBMS den erfolgreichen Abschluss der Transaktion. Die Überführung der Änderungen vom Transaktions-Log in die Datenbank kann dann zu einem späteren Zeitpunkt erfolgen, zur Not eben auch nach einem Stromausfall oder nach der Reparatur des Datenbank-Servers.

Wie so oft ist vieles, was in der Theorie einfach klingt, in der Praxis schwer zu realisieren. Das DBMS betrachtet das Betriebssystem und die Server-Hardware als idealisiertes, fehlerfreies System. Alles, was ab dieser Ebene passiert, fällt nicht mehr in die Verantwortung des DBMS. Tatsächlich kann auf dieser Ebene natürlich noch einiges schiefgehen. Deswegen sollten Sie bei der Speicherung wichtiger Daten bei der Hardware nicht sparen. Grundbedingungen sind:

- ▶ redundante Datenträger (*Redundant Array of Independent Disks* = RAID)
- ▶ unterbrechungsfreie, redundante Stromversorgung
- ▶ ECC RAM (also Arbeitsspeicher mit *Error Correction Code*, der das Umkippen eines Bits durch Alphastrahlung oder einen Defekt erkennt und im Idealfall sogar korrigieren kann)
- ▶ regelmäßige Hardware-Upgrades bzw. regelmäßiger Austausch von Komponenten, *bevor* diese ausfallen

Aber selbst wenn die Hardware, das Betriebssystem und der Datenbank-Server fehlerfrei funktionieren, ist Durability nicht ohne Weiteres zu garantieren. Das Problem, das jetzt noch verbleibt, heißt *Caching*. Letzten Endes geht es wieder einmal darum, dass ein DBMS nicht nur sicher, sondern auch schnell sein soll. Deswegen werden Schreiboperationen üblicherweise auf allen Ebenen zwischengespeichert (siehe Abbildung 2.6):

- ▶ Um möglichst viele Transaktionen pro Sekunde auszuführen, verwalten viele Datenbank-Server einen Zwischenspeicher (Cache), der sich die durchzuführenden Datenbankoperationen merkt und diese so schnell wie möglich an das Betriebssystem weiterleitet. Das steigert die Geschwindigkeit, widerspricht aber natürlich der Durability-Forderung. Ob es einen derartigen Zwischenspeicher gibt, wie groß er ist, wie lange offene Transaktionen maximal dort bleiben dürfen (z. B. eine Sekunde), sind DBMS-spezifische Parameter, die Datenbankadministratoren einstellen können. Oft ist bei Geschwindigkeitsproblemen der Druck groß, an diesen Schrauben zu drehen, statt einen viel teureren Server anzuschaffen.
- ▶ Wenn das DBMS vom Betriebssystem eine *File Synchronization* (kurz *fsync*) für die Logging-Datei anfordert, kann es sein, dass das Betriebssystem sie bestätigt, obwohl die zu schreibenden Dateien sich erst in einem – diesmal auf Betriebssystemebene verwalteten – Zwischenspeicher befinden. Im Normalfall wird sich das Betriebssystem in den nächsten Sekunden darum kümmern, alle offenen Operationen tatsächlich auszuführen. Wenn genau in dieser Zeitspanne ein Stromausfall stattfindet, kommt es dazu nicht mehr, und die Transaktion ist verloren.
- ▶ Falls der Datenbank-Server in einer virtuellen Umgebung oder in einem Container ausgeführt wird, kann es auch auf dieser Ebene zu Performanceoptimierungen kommen, die der Durability wenig zuträglich sind.
- ▶ Sogar wenn sämtliche *Write Caches* auf Betriebssystem-, Virtualisierungs- oder Containerebene explizit deaktiviert sind, kann es immer noch zu einer Zwischenspeicherung auf der Festplatte oder SSD kommen! Die meisten marktüblichen Datenträger besitzen einen extrem schnellen Cache, in dem Schreiboperationen zwischengespeichert werden. Die Festplatte oder SSD meldet an das Betriebssystem, dass es die Daten erhalten hat; tatsächlich kann es danach aber noch

Sekunden dauern, bis die Schreiboperation tatsächlich auf der Festplatte oder in einem nicht volatilen Bereich der SSD gespeichert ist.

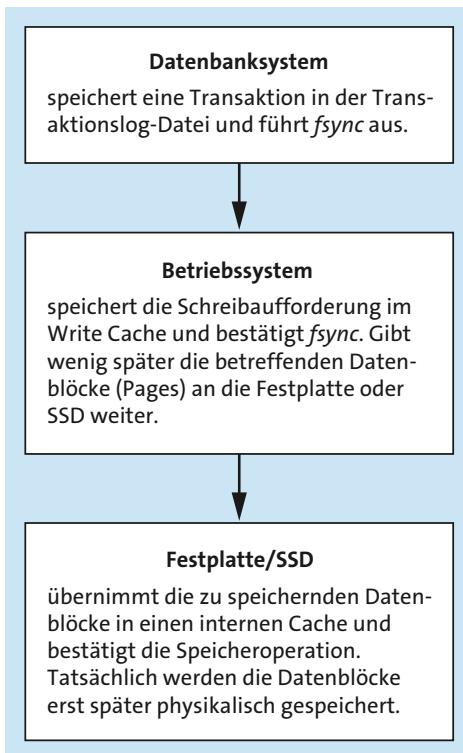


Abbildung 2.6 Zwischenspeicher (Write Caches) auf allen Ebenen verbessern die Performance, mindern aber die Durability.

Sie sehen schon, dass sich die Dauerhaftigkeit einer Transaktion bzw. einer Schreiboperation gar nicht so leicht garantieren lässt. Erschwerend kommt hinzu, dass auf dieser Ebene selten Tests gemacht werden – einerseits, weil ein Stromausfall oder ähnliche Fehlerursachen sehr selten vorkommen, andererseits, weil derartige Tests schwer durchzuführen sind und mit etwas Pech zu defekten Hardwarekomponenten führen. (Kein Administrator zieht gerne wiederholt bei einem 10.000 oder 20.000 Euro teuren Server einfach den Stromstecker.) Umgekehrt ist mit diesem Hintergrundwissen klar, dass Sie allein mit einer unterbrechungsfreien Stromversorgung schon einigen Problemen aus dem Weg gehen.

Auswirkungen, wenn ACID nicht funktioniert (»Dirty Read« etc.)

Auf den vergangenen Seiten habe ich Ihnen bereits zwei konkrete Beispiele dafür geliefert, was schiefgehen kann, wenn Transaktionen nicht korrekt verarbeitet werden (Umbuchung von einem Bankkonto auf ein anderes, zwei parallele Buchungen

bei einem Konto). In der Datenbanktheorie gibt es eigene Fachbegriffe für typische Fehler bzw. Anomalien:

- ▶ **Dirty Read:** Von einem *Dirty Read* spricht man, wenn eine Transaktion A veränderte Daten sieht, die sich durch eine zweite, noch gar nicht abgeschlossene Transaktion B ergeben.

Beispiel: Transaktion B erhöht den Preis eines Produkts. Diese Transaktion ist noch nicht abgeschlossen, d. h., es ist noch nicht klar, ob diese Preiserhöhung je dauerhaft in der Datenbank gespeichert wird. (Möglicherweise wird die Transaktion vom Client-Programm später durch ROLLBACK widerrufen. Oder das DBMS verzögert die Ausführung der Transaktion, weil dadurch eine Konsistenzregel gebrochen wird oder weil es einen Konflikt zu einer anderen Transaktion sieht.)

Dennoch sieht Transaktion A – der Kauf eines Produkts durch einen Kunden – bereits den erhöhten Preis.

- ▶ **Lost Update:** In diesem Fall verändern zwei Transaktionen unabhängig voneinander denselben Datensatz. Letztendlich wird aber nur eine Änderung dauerhaft gespeichert.

Beispiel: Genau während das Gehalt überwiesen wird, hebt der Kunde Geld ab. Beide Transaktionen beginnen mit dem ursprünglichen Kontostand. Die Transaktion, die später durchgeführt wird, »gewinnt« (siehe [Abbildung 2.5](#)).

- ▶ **Non-repeatable Read:** Von diesem Fehler spricht man, wenn mehrere gleichzeitige SQL-Abfragen innerhalb einer Transaktion zu unterschiedlichen Ergebnissen kommen. Da sich Transaktionen gegenseitig nicht beeinflussen dürfen, ist ein *Non-repeatable Read* bei einer vollständigen ACID-Konformität ausgeschlossen.

Beispiel: Die Bank ermittelt in Transaktion A alle Konten mit negativem Kontostand. Zugleich findet Transaktion B statt, wobei ein weiteres Konto überzogen wird. Transaktion A läuft weiter, eine neue Abfrage nach überzogenen Konten liefert nun ein anderes Ergebnis.

- ▶ **Phantom Read bzw. Inconsistent Read:** Ein *Phantom Read* ist eine Variante zum Non-repeatable Read, allerdings ergibt sich der Fehler durch zwei *unterschiedliche* SQL-Kommandos.

Beispiel: In Transaktion A ermittelt die Bank zuerst die Anzahl aller negativen Konten. Zugleich findet Transaktion B statt, ein weiteres Konto wird überzogen. In der noch aktiven Transaktion A wird nun die Summe aller negativen Konten ermittelt, wobei nun auch das frisch überzogene Konto berücksichtigt wird. Anzahl und Summe passen daher nicht zusammen, die Berechnung des durchschnittlichen Fehlbetrags liefert ein falsches Ergebnis.

12.1 Transaktionen in SQL

Grundsätzlich reichen drei simple Kommandos aus, um Transaktionen zu steuern:

- ▶ START TRANSACTION leitet eine neue Transaktion ein. Bei einigen DBMS können Sie stattdessen auch das Kommando BEGIN [TRANSACTION] verwenden, bei manchen müssen Sie sogar, weil das Kommando START TRANSACTION fehlt. (Laut SQL-Standard ist START TRANSACTION korrekt.)
- ▶ COMMIT [TRANSACTION] schließt die Transaktion ab. Beachten Sie, dass bei COMMIT ein Fehler auftreten kann: Wenn die eigene Transaktion einen Konflikt mit einer anderen Transaktion verursacht, die vor Ihrer Transaktion abgeschlossen wurde, blockiert das DBMS die Ausführung Ihrer Transaktion und widerruft stattdessen sämtliche seit START bzw. BEGIN durchgeführten Änderungen in der Datenbank.
- ▶ ROLLBACK [TRANSACTION] widerruft die Transaktion und damit alle Kommandos, die Sie seit dem letzten START TRANSACTION ausgeführt haben.

Die folgenden Zeilen zeigen, wie 100 € von einem Konto auf ein anderes Konto überwiesen werden. Die Transaktion garantiert, dass entweder beide Kommandos korrekt ausgeführt werden oder aber gar keines.

`START TRANSACTION`

`UPDATE accounts SET amount = amount - 100 WHERE id = 127`

`UPDATE accounts SET amount = amount + 100 WHERE id = 288`

`COMMIT -- bestätigt beide UPDATES`

Transaktionen sind auch bei der Entwicklung von Client-Programmen enorm wichtig. Sie ermöglichen es, mehrere Datenbankenoperationen zu einer Gruppe zusammenzufassen und erst am Schluss entscheiden zu können, ob die Operationen bestätigt werden oder nicht. Das erleichtert die Fehlerabsicherung. Außerdem kann die Transaktion zum Schluss von einer Bedingung abhängig gemacht werden – z. B. ob der Bezahlvorgang erfolgreich war.

Markierungspunkte innerhalb einer Transaktion (SAVEPOINT)

Mit `SAVEPOINT spname` markieren Sie innerhalb einer laufenden Transaktion einen Zeitpunkt. Innerhalb der Transaktion kann nun `ROLLBACK TO SAVEPOINT spname` ausgeführt werden. Damit werden die Kommandos bestätigt, die vor der Definition des Markierungspunkts durchgeführt wurden; alle späteren Kommandos werden dagegen widerrufen. Die Transaktion als Ganzes läuft weiter. Sie muss weiterhin mit `COMMIT` bestätigt oder mit `ROLLBACK` vollständig widerrufen werden.

```

START TRANSACTION

INSERT INTO mytable (cola, colb) VALUES (17, 'efg')

SAVEPOINT mysavepoint

DELETE FROM mytable WHERE colc > 100

ROLLBACK TO mysavepoint -- widerruft DELETE, die
-- Transaktion geht weiter

UPDATE mytable SET cola = 27 WHERE id = 123

COMMIT -- schließt die Transaktion ab, bestätigt
-- also INSERT und UPDATE

```

Locking beeinflussen (SELECT FOR UPDATE)

Der SQL-Standard sieht das Kommando `SELECT ... FOR UPDATE` vor, um am Beginn einer Transaktion ausgewählte Datensätze gleichsam zu markieren. Das hat zur Folge, dass andere Transaktionen, die diese Datensätze verändern möchten, blockiert werden. Das Kommando `SELECT ... FOR UPDATE` ist unter anderem in MySQL, Oracle und PostgreSQL implementiert.

```

START TRANSACTION

SELECT * FROM mytable FOR UPDATE
WHERE id BETWEEN 100 AND 200

UPDATE mytable SET cola = 456 WHERE id = 123
...

COMMIT

```

Ob `SELECT ... FOR UPDATE` überhaupt eine Wirkung hat, hängt vom aktiven Isolation Level ab (siehe [Abschnitt 12.2](#)). Im Level *Serializable* ist das Kommando überflüssig oder bestenfalls eine Hilfe für den Query Optimizer.

Bei schwächeren Isolation Level kann `SELECT ... FOR UPDATE` Veränderungen der Daten durch andere Transaktionen verhindern. Das Kommando vergrößert so die Unabhängigkeit von mehreren parallel ablaufenden Transaktionen über das durch den aktiven Isolation Level vorgesehene Maß. Lesen Sie das Verhalten dieses Kommandos unbedingt im Handbuch Ihres DBMS nach, die Details der Implementierung variieren stark.

Auto-Commit-Modus

Vielleicht fragen Sie sich, wie das DBMS mit SQL-Kommandos umgeht, die nicht explizit in einer Transaktion ausgeführt werden. In diesem Fall wird normalerweise jedes Kommando als winzige Einzeltransaktion ausgeführt.

Viele DBMS sprechen in diesem Zusammenhang vom Auto-Commit-Modus, das heißt, jedes Kommando impliziert einen sofortigen Commit. Bei manchen DBMS bzw. Client-Programmen können Sie den Auto-Commit-Modus deaktivieren. Dann startet das erste Kommando eine Transaktion, deren Ende aber offen ist. Sämtliche ausgeführten Kommandos werden erst durch ein explizites COMMIT dauerhaft ausgeführt.

DBMS-spezifische Varianten und Erweiterungen

Oracle, SQL Server und SQLite kennen das Kommando START TRANSACTION nicht. Bei Oracle müssen leiten Sie Transaktionen mit SET TRANSACTION ein. Bei SQLite verwenden Sie BEGIN, bei SQL Server BEGIN TRANSACTION.

Bei MySQL startet COMMIT [AND] CHAIN bzw. ROLLBACK [AND] CHAIN sofort die nächste Transaktion.

Beim SQL Server setzen Sie Savepoints mit SAVE statt mit SAVEPOINT. Bei PostgreSQL und SQLite können gesetzte Savepoints mit RELEASE SAVEPOINT sname wieder aufgelöst werden.

Bei PostgreSQL und MySQL gibt es als Alternative zu SELECT ... FOR UPDATE das Kommando SELECT ... FOR SHARE. Damit werden ebenfalls andere Transaktionen blockiert, die die betroffenen Datensätze ändern wollen. Der Unterschied besteht darin, dass das DBMS mehrere parallele Transaktionen erlaubt, die mit SELECT ... FOR SHARE die gleichen Datensätze sperren:

<https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html>

<https://www.postgresql.org/docs/current/sql-select.html#SQL-FOR-UPDATE-SHARE>

Oracle kennt SELECT ... FOR SHARE nicht, dafür können Sie die Wirkung von SELECT ... FOR UPDATE durch weitere Schlüsselwörter (NOWAIT, WAIT n, SKIP LOCKED) modifizieren:

<https://markjbobak.wordpress.com/2010/04/06/unintended-consequences>

SQL Server unterstützt weder SELECT ... FOR UPDATE noch SELECT ... FOR SHARE. Stattdessen erlaubt SELECT ... FOR BROWSE explizit, dass Datensätze während des Auslesens durch andere Datensätze verändert werden. Dieses Kommando reduziert also die Sicherheit, erhöht dafür aber die Effizienz.