edv-buchversand.de

Datenbanksysteme

Das umfassende Lehrbuch

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

Kapitel 1
Wozu Datenbanken?

In diesem Kapitel möchte ich Sie mit der Welt und der Denkweise von Datenbanken bzw. Datenbankmanagementsystemen vertraut machen, ohne Sie mit allzu viel Details und theoretischem Wissen zu überwältigen. Ich beginne damit, Ihnen die Funktionsweise eines Onlineshops aus Datenbanksicht zu schildern. Obwohl die Beschreibung vereinfacht ist, wird Ihnen danach klar sein, dass ein typischer Onlineeinkauf unzählige (Hunderte, womöglich Tausende) winzige Datenbankoperationen mit sich bringt.

Im nächsten Abschnitt gehe ich auf den »Todfeind« jeder relationalen Datenbank ein: auf Excel-Tabellen. Ich versuche, Ihnen klarzumachen, dass Excel trotz seiner einfachen Bedienung nur in Ausnahmefällen eine richtige Datenbanklösung ersetzen kann. Sobald größere Datenmengen im Spiel sind, sobald mehrere Personen einer Organisation oder eines Unternehmens gleichzeitig Daten lesen und verändern wollen, gerät jedes Tabellenkalkulationsprogramm an seine Grenzen. Dieser Abschnitt beantwortet also die zentrale Frage dieses Kapitels: »Wozu Datenbanken?«

Zuletzt möchte ich Ihnen in der Art eines »Hello World!«-Beispiels zeigen, wie Sie Ihre erste eigene Datenbank einrichten und mit Daten füllen. Ich nutze die Gelegenheit, um Sie mit den ersten Begriffen rund um das Thema »Datenbankdesign« vertraut zu machen. Gleichzeitig lernen Sie das Programm kennen, das in diesem Buch eine zentrale Rolle für alle Beispiele spielt: die MySQL Workbench. Dieses Programm hilft Ihnen dabei, eigene Datenbanken zu entwerfen, sie auf einem Datenbank-Server einzurichten, SQL-Kommandos auszuführen und administrative Aufgaben zu erledigen.

1.1 Datenbanken sind allgegenwärtig

Haben Sie sich schon einmal Gedanken darüber gemacht, was hinter den Kulissen passiert, wenn Sie bei Amazon oder einem beliebigen anderen Onlinehändler eine Bestellung durchführen? Aus Benutzersicht sieht es so aus:

- Sie melden sich an bzw. sind im Webbrowser dank eines Cookies bereits angemeldet.
- Sie suchen im Webbrowser nach den gewünschten Produkten.

- Sie legen diese in einen virtuellen Einkaufskorb.
- Sie schließen den Einkauf mit der Bezahlung ab. Eventuell können dabei früher hinterlegte Daten genutzt werden. Andernfalls geben Sie Ihre Kreditkartennummer oder ähnliche Informationen an.
- Der Onlinehändler verständigt Sie per Mail zuerst über den erfolgten Einkauf und später über den Versand.
- Beim nächsten Besuch der Website des Onlinehändlers empfiehlt Ihnen dieser Produkte, die auf wunderbare Weise gut zur letzten Bestellung passen. Wenn Sie also vorhin die erste Staffel von »Games of Thrones« gekauft haben, ist es kein Zufall, dass Sie beim nächsten Besuch die zweite Staffel angeboten bekommen, sicherlich zu einem besonders günstigen Preis ...

Alles gut und recht, werden Sie sagen, aber das kenne ich schon. Worum es mir an dieser Stelle geht, ist, Ihnen zu zeigen, dass bei jedem der oben aufgezählten Schritte Datenbankoperationen im Spiel sind.

Anmeldung

Beginnen wir mit der Anmeldung: Wenn es sich um die erste Bestellung handelt, müssen Sie sich im Onlineshop einmalig registrieren. (Üblicherweise erfolgt diese Registrierung erst beim Abschluss der ersten Bestellung, aber das tut hier nichts zur Sache.)

Sie geben also Ihren Namen, Ihre E-Mail-Adresse und Ihre Postadresse an und legen ein Passwort fest. Diese Daten werden – Sie haben es sicherlich schon erraten – in einer Datenbank gespeichert (siehe Abbildung 1.1). Genau genommen landen die Daten in einer *Tabelle* der Datenbank. Diese und andere Spitzfindigkeiten wollen wir aber vorerst beiseitelassen.

Bei einer neuerlichen Anmeldung geben Sie nur noch die E-Mail-Adresse und das Passwort an. Anhand dieser Informationen kann der Onlineshop alle weiteren Daten aus der Datenbank auslesen und gleichzeitig Ihr Passwort verifizieren. (Es ist übrigens sicherheitstechnisch absolut verpönt, Passwörter im Klartext zu speichern. Stattdessen werden sogenannte *Hash-Codes* gespeichert. Warum das zweckmäßig ist, erkläre ich Ihnen in Abschnitt 4.9, »Besondere Datentypen«.)

Die meisten Onlineshops versuchen, den Anmeldeprozess so komfortabel wie möglich zu machen, und bieten Ihnen die (oft unausgesprochene) Möglichkeit, angemeldet zu bleiben. Dazu wird in einem *Cookie* (einer winzigen Datei, die Ihr Webbrowser auf Ihrem Rechner speichert) ein eindeutiger Code hinterlegt. Datenbanktechnisch muss sich der Onlineshop jetzt zusätzlich merken, welchem Cookie-Code welcher Kunde zugeordnet ist.

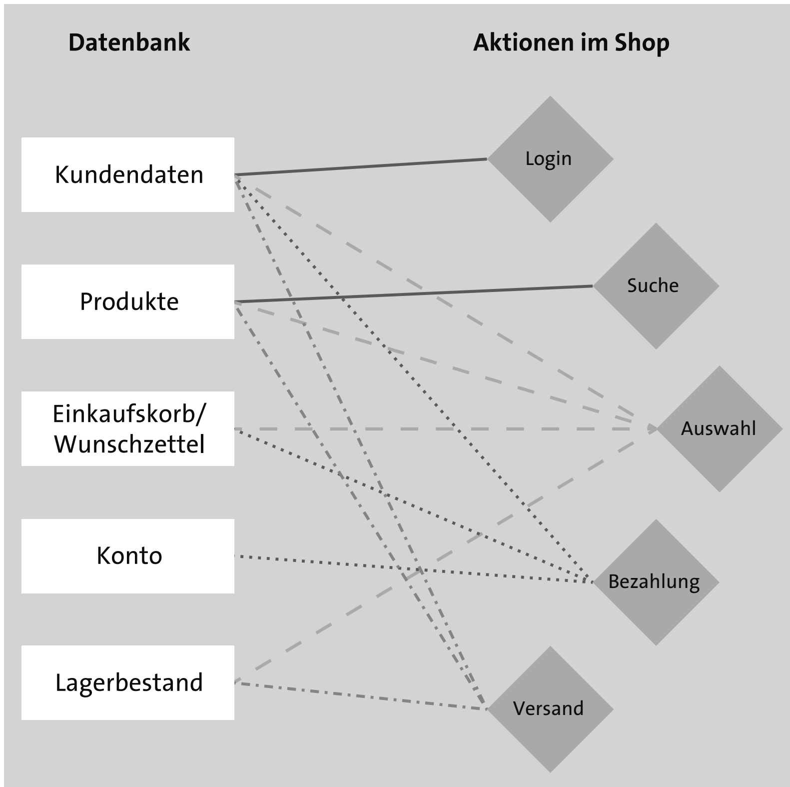

Abbildung 1.1 Stark vereinfachte Darstellung einer Datenbank für einen Onlineshop

Anhand dieses Codes kann die Website Sie wiedererkennen. Das ist nicht nur für Sie bequem, sondern auch für den Onlineshop! Während Sie das riesige Angebot durchstöbern, erkennt der Shop gleich, ob Sie sich eher für Kleider und Parfüm interessieren oder für technische Produkte und Autozubehör. Dieses Wissen hilft, jederzeit passende Empfehlungen und Angebote einzublenden.

Lustigerweise werden die Cookies client-seitig oft ebenfalls in einer Datenbank gespeichert. Viele Webbrowser greifen dazu auf die Bibliothek *SQLite* zurück. Damit arbeiten zwei Datenbankprogramme – ohne voneinander zu wissen – zusammen, um Ihnen ein angenehmes Weberlebnis und dem Onlineshop optimale Werbemöglichkeiten zu bieten.

Das Produktangebot

Sie besuchen den Onlineshop, um ein Produkt zu finden. Vielleicht wollen Sie mehrere Alternativen vergleichen oder ein besonders preisgünstiges Angebot entdecken.

Das setzt voraus, dass der Onlineshop Informationen zu allen lagernden Produkten hat (oder zu Produkten, die ein kooperierender Dritthändler liefern kann).

Selbst wenn nicht wie in diesem Buch Datenbanken im Fokus stehen, spricht man in diesem Kontext oft von einer *Produktdatenbank*. Das ist eine geordnete Liste aller Produkte, wobei in der Regel zu jedem Produkt eine ausführliche Beschreibung (Text), diverse Abbildungen, die Zuordnung zu Produktkategorien, der aktuelle Preis, die Verfügbarkeit usw. gespeichert werden. Mit einer Suchfunktion können Sie nun nach Markennamen (»Sony«), nach Kategorien (»Systemkamera«), Modellnummern (»Alpha 7«), Identifikationscodes (»B00Q2KEVA2«), Eigenschaften (»1080p Full HD«) etc. suchen.

All diese Daten befinden sich natürlich ebenfalls in einer Datenbank. Nicht ganz so selbstverständlich ist, wie der Onlineshop mit Ihrer Suche umgeht: Vielleicht erkennt der Suchalgorithmus, dass die von Ihnen gesuchte Kamera gerade nicht lieferbar ist. Dann wäre es zielführend, Ihnen alternative Angebote schmackhaft zu machen, vielleicht das nächstbessere Modell des gleichen Herstellers oder ein Modell mit ähnlichen technischen Eigenschaften eines anderen Herstellers. Die Suchergebnisse könnten so gereiht sein, dass jene Modelle zuerst angezeigt werden, bei denen die Gewinnspanne für den Shop höher ist. Zweckmäßig ist auch, ergänzend zum eigentlichen Produkt gleich sinnvolle Erweiterungen anzubieten – bei einer Kamera z. B. ein passendes Objektiv oder eine Kameratasche.

Wir setzen derartige Strategien heute fast schon selbstverständlich voraus. Tatsächlich erfordern solche »Tricks« aber, dass die Datenbank nicht nur Produktdaten enthält, sondern auch eine Menge Zusatzinformationen (Metadaten): Welche Produkte haben Käufer in der Vergangenheit angesehen, bevor sie sich für eines entschieden haben? Welche Produkte werden oft gemeinsam gekauft? Welche Produkte waren in der letzten Zeit besonders beliebt, sind also »im Trend«?

Der Einkaufskorb

Vergleichsweise einfach ist es, einen Einkaufskorb zu implementieren: Wenn Sie auf den Button In den Einkaufskorb legen klicken, merkt sich der Onlineshop das zuletzt angesehene Produkt und verbindet es mit Ihren Kundendaten. Oder, etwas technischer ausgedrückt: Der Shop speichert eine Zuordnung zwischen dem Produkt und Ihrem Kunden-Account.

Jetzt kommt noch eine Prise Verkaufspsychologie hinzu: Der Onlineshop wird Sie darauf hinweisen, dass Sie sich mit dem Kauf eines weiteren Produkts die Versandkosten sparen könnten. Es wird auch den Lagerbestand in Ihren Fokus rücken (»nur noch 3 Stück auf Lager«). Diesbezüglich noch viel aggressiver sind Hotelbuchungsseiten, die Ihnen ständig den Eindruck vermitteln, dass Sie das beste Angebot verpassen, wenn Sie sich nicht sofort entscheiden.

Egal, ob Sie sich nun zu einem schnellen Verkaufsabschluss verführen lassen – uns interessieren hier die datenbanktechnischen Details: Verkaufsfördernde Argumente funktionieren umso besser, je glaubhafter sie sind, d. h. auf je mehr Daten der Shop zurückgreifen kann: Wann wurde das Produkt zuletzt gekauft? Wie viel Stück werden typischerweise pro Stunde zu einer bestimmten Uhrzeit verkauft? Wenn solche und ähnliche Informationen in der Datenbank des Shops bereitstehen, gelingt die Manipulation der Kunden umso besser ...

Bezahlung und Versand

Wenn Sie das erste Mal in einem Shop zahlen, müssen Sie Zahlungsinformationen, wie eine Kreditkartennummer, angeben. Viele Shops erlauben auch die Verwendung externer Dienstleister wie PayPal. Die Bezahlungsdaten werden dann zusammen mit Ihren übrigen persönlichen Daten gespeichert, damit Sie die Bezahlung in Zukunft möglichst bequem erledigen können.

Sofern die Bezahlung durchgeführt werden kann, wenn also die Software-Schnittstelle zwischen dem Shop und dem Zahlungsdienstleister den Bezahlprozess ohne Fehler abschließt, werden sowohl der Rechnungsbetrag als auch der Zahlungseingang in einem weiteren Bereich der Datenbank gespeichert. Üblicherweise wird der Kunde per Mail davon informiert.

Jetzt startet die Logistik für den Versand: Die bestellten Waren müssen im Lager aufgefunden, verpackt und an einen Paketdienst übergeben werden. Wiederum müssen dabei diverse Informationen aus der Datenbank gelesen werden: An welche Adresse soll das Paket versendet werden? Welcher Paketdienst ist für den Versand am besten geeignet bzw. am günstigsten? Gleichzeitig muss mit der Entnahme, der Lagerbestand aktualisiert werden.

Sonstiges

Ich habe in diesem Abschnitt die datenbanktechnische Komplexität eines Verkaufsprozesses nur angedeutet. Tatsächlich gäbe es natürlich noch mehr Details, die je nach Shopgröße in der Datenbank zu berücksichtigen wären:

- der Umgang mit Retouren
- das Speichern von tagesaktuellen Top-10-Listen für verschiedene Kategorien
- die Berücksichtigung unterschiedlicher Mehrwertsteuersätze je nach Produkt und Adresse des Käufers
- eine dynamische Preisbildung, die sich an Angebot und Nachfrage orientiert und auf die Preise von konkurrierenden Shops Rücksicht nimmt
- der Versand von Marketingmails mit Produktvorschlägen, die auf das Kaufverhalten der Kunden abgestimmt sind

1.2 Warum eine Excel-Tabelle nicht ausreicht

Gerade in kleinen oder mittelgroßen Unternehmen haben viele Datenbanken das Licht der Welt in Form einer Excel-Tabelle erblickt. In diesem Abschnitt erkläre ich Ihnen, warum Excel kein richtiges Datenbankmanagementsystem ersetzen kann. (Von mir aus darf es anstelle von Excel natürlich gerne auch *LibreOffice Calc* oder *Apple Numbers* sein. Die hier beschriebenen strukturellen Probleme gelten auch für diese Programme. Wegen der großen Verbreitung schreibe ich im Folgenden aber einfach von Excel.)

Nehmen wir an, Sie arbeiten in einer Tischlerei, die jährlich gut 100 Rechnungen ausstellt. Ihre Aufgabe ist es, die Rechnungen zu versenden, den Zahlungseingang zu kontrollieren und bei Bedarf auch Mahnungen zu verschicken. Ein naheliegendes Hilfsmittel ist dabei eine simple Excel-Tabelle, in der Sie jede neue Rechnung sowie später den Zahlungseingang eintragen. Solange es bei 100 Rechnungen im Jahr bleibt, spricht nichts gegen diese Vorgehensweise.

Es kommt, wie es für ein Beispiel in diesem Buch kommen muss: Die Tischlerei wächst, und die Chefin entschließt sich, parallel zum bisherigen Geschäft in einem Shop auch Holzspielzeug und andere Kleinartikel zu verkaufen. Damit steigt die Anzahl der pro Jahr ausgestellten Rechnungen auf 1.000. Obwohl Sie mit ein paar VBA-Funktionen versuchen, die Administration des Rechnungswesen unter Kontrolle zu halten, häufen sich die Probleme. Und die haben keineswegs damit zu tun, dass Excel mit 1.000 Zeilen überfordert wäre.

Excel ist nicht netzwerktauglich

Das größte Hindernis im hier angegebenen Szenario besteht darin, dass Excel weder netzwerktauglich noch für den Multi-User-Einsatz optimiert ist. Natürlich können Sie die Datei in einem Netzwerkverzeichnis ablegen, sodass mehrere Mitarbeiter die Datei lesen und auch verändern können; aber sollten unglücklicherweise zwei Personen zugleich Änderungen vornehmen – Sie bei der Durchsicht des Zahlungseingangs und das Verkaufspersonal im Shop beim Verkauf einer Spielzeugeisenbahn –, wird es zu Kollisionen und Datenverlusten kommen. (Diesbezüglich besser für den parallelen Zugriff durch mehrere Nutzer optimiert sind webbasierte Tabellenkalkulationsprogramme wie *Google Sheets*.)

Ein echtes Datenbankmanagementsystem (in Zukunft oft kurz *DBMS*) ist hingegen von vornherein für den Client-Server-Betrieb konzipiert. Der Datenbank-Server akzeptiert also über das Netzwerk Client-Verbindungen und verarbeitet die Datenbankaufträge. (Mit einem *Client* ist im Datenbankkontext ein Programm gemeint, das mit dem Datenbank-Server kommuniziert.) Sogenannte *Transaktionen* stellen ein

klar definiertes Verhalten auch für den Fall sicher, dass zwei Operationen parallel oder im Konflikt zueinander ausgeführt werden.

Um hier eine Konfliktsituation möglichst einfach darzustellen: In so einem Fall wird für einen Client alles gut gehen. Der zweite erhält eine Fehlermeldung und kann den Vorgang dann wiederholen. Transaktionen und andere DBMS-Sicherheitsmaßnahmen rund um Atomicity, Consistency, Isolation und Durability behandle ich in den weiteren Kapiteln dieses Buchs noch ausführlich.

Features von Datenbankmanagementsystemen

Natürlich gibt es außer der Netzwerktauglichkeit noch mehr grundlegende Unterschiede zwischen einem Tabellenkalkulationsprogramm und einem Datenbankmanagementsystem:

- **Daten verknüpfen:** Datenbankmanagementsysteme brillieren, wenn es darum geht, unterschiedliche Daten zu verbinden. Bleiben wir beim Beispiel dieses Abschnitts: Es wird Kunden geben, die Ihre Tischlerei immer wieder beauftragen. Es ist nicht zweckmäßig, die Kundendaten (Name, Adresse usw.) immer wieder neu zu speichern. In einem relationalen DBMS legen Sie in diesem Fall *zwei* Tabellen an, eine für die Kundendaten und eine zweite für die Rechnungen.

  Diese Vorgehensweise mindert die Redundanz und stellt sicher, dass zentrale Daten nur *einmal* vorliegen. Wenn sich also die E-Mail-Adresse oder Telefonnummer des Kunden ändert, speichern Sie die neue Adresse einmalig in der Kundentabelle. *Alle* Anwendungen, die auf die Datenbank zurückgreifen, verwenden nun automatisch die richtige E-Mail-Adresse.

  Die schlechte Verknüpfbarkeit zusammengehörender Daten macht sich umso stärker bemerkbar, je weiter sich eine Excel-Lösung von der ursprünglichen Zielsetzung entfernt. Vielleicht hat sich Excel vor der Expansion der Tischlerei zur Kontrolle des Zahlungseingangs bewährt. Deswegen ist die Idee aufgekommen, auch gleich das Personalwesen, alle Aufträge, das Bestellwesen und die Lagerverwaltung mit Excel zu verwalten. Während jede Aufgabe für sich lösbar ist, scheitert die Lösung am Zusammenspiel der Daten: Wo im Lager befinden sich die Eichenbretter, die für den Auftrag 137 im Rahmen der Bestellung vom 12. Mai angeschafft wurden? Wer hat diese Bestellung durchgeführt? Wer hat die Lieferung angenommen?

- **Zugriffskontrolle:** Für eine Excel-Datei gilt im Wesentlichen ein Alles-oder-nichts-Ansatz: Wer Zugriff auf die Datei hat, kann alles ändern, im schlimmsten Fall auch alles überschreiben – sei es in böser Absicht oder aus Versehen. Mit »gesperrten« Zellen können Sie dieses Risiko minimieren, aber auch das ist nur in Einzelfällen eine echte Lösung. Ein DBMS bietet dagegen die Möglichkeit, fein differenzierte Zugriffsregeln festzulegen.

- **Konsistenz:** Bei einem DBMS besteht die Möglichkeit, durch Regeln die Konsistenz von Daten in einem gewissen Ausmaß sicherzustellen. Dann ist es verboten, eine Kundennummer zu verwenden, die es gar nicht gibt, oder als Rechnungsbetrag versehentlich einen Text einzugeben.
- **Programmierbarkeit:** In einer Excel-Datei können Sie VBA-Code verankern, der bestimmte Vorgänge automatisiert. Es gibt aber nur wenige etablierte Schnittstellen (APIs) oder Bibliotheken, mit denen Programme von *außen* auf die in einer Excel-Datei gespeicherten Daten zugreifen oder diese ändern können – und schon gar nicht, wenn die Datei gerade geöffnet ist.

  Ganz anders sieht die Lage bei einem DBMS aus: Hier stehen für nahezu jede Programmiersprache Bibliotheken zur Verfügung für die Programmierung von Client-Apps. Egal, ob Sie ein Desktop-Programm, eine Smartphone-App oder eine Website entwickeln möchten – Sie können über das Netzwerk mit dem Datenbank-Server kommunizieren. (Es sei hier aber nicht verschwiegen, dass die Programmierung von Datenbank-Clients, egal für welche Plattform, mit erheblichem Aufwand verbunden ist.)
- **Logging und Backup:** Zusammen mit einem DBMS wird in der Regel ein automatisiertes Logging-System eingerichtet. Dieses kann bei geeigneter Konfiguration festhalten, wer wann was geändert hat. Regelmäßige Backups geben zudem die Möglichkeit, alte Versionen einer Datenbank wiederherzustellen.
- **Skalierbarkeit:** Aus einer Tischlerei wird üblicherweise nicht ein Großunternehmen wie Amazon. Bei vielen IT-Projekten schwingt aber immer die Hoffnung mit, den ganz großen Hit zu landen. Sollte dieser Glücksfall eintreten, ist es entscheidend, dass die Performance des Datenbank-Backends rasch und unkompliziert gesteigert werden kann.

  Zugegebenermaßen ist dies auch bei Datenbankmanagementsystemen nicht ganz unkompliziert. Mit etwas Know-how können aber die meisten DBMS über mehrere Rechner verteilt werden (Hochverfügbarkeit, Cluster-Datenbanken etc.). Im Gegensatz dazu gilt: Eine simple Excel-Datei skaliert überhaupt nicht.

Raus aus der Excel-Falle!

Hoffentlich konnte ich Sie davon überzeugen, dass Excel aus vielerlei Gründen kein geeignetes Werkzeug ist, um große Datenmengen zu verwalten. (Dabei stelle ich gar nicht in Frage, dass die restlichen Features großartig sind. Aber das Programm ist nun einmal zum Rechnen und zum Visualisieren von Daten gedacht.)

Warum wird Excel dennoch so häufig zweckentfremdet? Weil der Start so leichtfällt! Das Eingangsbeispiel dieses Kapitels, also eine Excel-Tabelle zur Überwachung des Zahlungseingangs, ist in fünf Minuten eingerichtet. Dass das Setup nicht ideal ist, stellt sich oft erst nach Jahren heraus.

Eine vergleichbare Datenbank steht mit etwas Übung zwar auch in einer Stunde, aber damit sind Sie von einer praxistauglichen Lösung noch meilenweit entfernt. Jetzt brauchen Sie ein Programm (einen Client), um die Rechnungsdaten einzugeben, um nach offenen Rechnungen zu suchen, eine Liste der betreffenden E-Mails zu exportieren usw. So ein Programm zu entwickeln, dauert selbst im Idealfall mehrere Tage, in der Realität eher ein paar Wochen. Selbst für die hier skizzierte recht simple Aufgabenstellung verursacht das hohe Kosten.

Natürlich kann man argumentieren, dass Sie dafür eine zukunftsfähige Lösung erhalten, die sich später erweitern lässt (z. B. um Web-Clients, Smartphone-Apps) und die die Anfangskosten mit der Zeit von selbst einspielt. Diese Argumentation trifft umso mehr zu, wenn Sie bereit sind, gleich noch einen Schritt weiterzugehen, und das mit Datenbanken verbundene Automatisierungspotenzial nutzen: Beispielsweise könnte ein Programm täglich den Bankeingang kontrollieren, bezahlte Rechnungen selbstständig als erledigt markieren und bei nicht bezahlten Rechnungen nach einem Monat ebenso automatisch eine Mahnung per Mail versenden.

Diese Überlegungen führen aber von einer simplen Excel-Tabelle zu einem komplexen Programm. Für die kleine Tischlerei von nebenan ist das weder zielführend noch bezahlbar. Je größer ein Unternehmen oder eine Organisation ist, je vielfältiger die Anforderungen (Daten, Client-Programme) sind, je mehr Daten zu verwalten sind, desto mehr Argumente sprechen für eine »richtige« Datenbanklösung und gegen einen Excel-Hack.

1.3 Die erste eigene Datenbank

Das Design einer Datenbank ist entscheidend dafür, wie gut sich die Daten später finden lassen, wie leicht sich Client-Anwendungen programmieren lassen und wie effizient das System letztlich läuft. Deswegen beschäftigt sich circa die Hälfte dieses Buchs mit DBMS-Grundlagen sowie mit Regeln zum perfekten Datenbankdesign. Natürlich bemühe ich mich, Ihnen diese Grundlagen so praxisnah wie möglich zu vermitteln; ganz lässt sich der eine oder andere Ausflug in die Datenbanktheorie aber nicht vermeiden.

Umso wichtiger erscheint es mir, in diesem Einführungskapitel mit einem einfachen praktischen Beispiel zu starten: Auf den nächsten Seiten zeige ich Ihnen, wie Sie eine ganz einfache Adressdatenbank einrichten, in dieser Datenbank einige Datensätze speichern und die Daten dann mit SQL-Kommandos abfragen.

Dabei geht es mir nicht um das bisschen Datenbank-Know-how, das in diesem Minibeispiel steckt. Viel wichtiger ist es mir, Ihnen vorweg einmal zu zeigen, was es eigentlich heißt, mit Datenbanken zu arbeiten und welche Werkzeuge Ihnen dabei

Indizes in der MySQL Workbench einrichten

SQL bietet mit `CREATE TABLE` bzw. `ALTER TABLE` einen allgemeingültigen Weg, Spalten mit einem Index auszustatten. Häufig werden Sie das Datenbankdesign aber mit einer Benutzeroberfläche erstellen. Dann reichen ein paar Mausklicks, um eine Spalte mit einem Index auszustatten.

In der MySQL Workbench statten Sie Spalten im Dialogblatt COLUMNS mit den Optionen PK oder UQ mit einem Primär- oder Unique-Index aus. Andere Indizes sowie ihre Optionen stellen Sie im Dialogblatt INDEXES ein (siehe Abbildung 6.1).

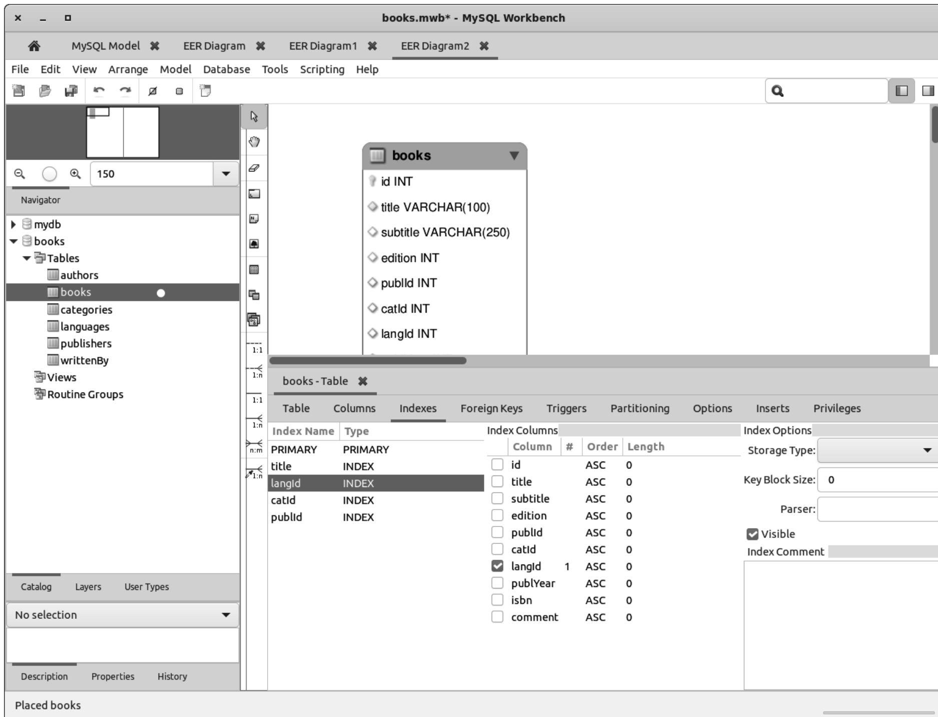

Abbildung 6.1 Indizes eines Datenbankmodells in der MySQL Workbench einstellen

6.2 Index-Interna und B-Trees

Dieser Abschnitt beschreibt, wie Datensätze und die dazugehörenden Indizes durch das DBMS physikalisch auf Datenträgern gespeichert werden. Wenn Sie sich ausschließlich für die Anwendung von Datenbankmanagementsystemen interessieren, können Sie die folgenden Seiten natürlich einfach überspringen. Ich glaube aber, dass ein minimales Wissen über die Interna hilfreich ist, DBMS besser zu verstehen und somit auch effizienter zu einzusetzen. (Wenn Sie dieses Buch als Begleitlektüre zu

einer Datenbankvorlesung verwenden, haben Sie vermutlich gar keine Wahl: Die Themen »Clustered versus Non-clustered Index« und »B-Trees« eignen sich einfach zu gut für Prüfungsfragen ...)

Speicherblöcke (Pages)

Jedes Mal, wenn Sie Datensätze in eine Tabelle Ihrer Datenbank einfügen, müssen diese physikalisch auf der Festplatte oder SSD gespeichert werden. Bei einer späteren Änderung eines Datensatzes muss auch diese Änderung gespeichert werden. Falls der geänderte Datensatz mehr Platz braucht als der bisherige (z. B. weil eine Zeichenkette länger geworden ist), muss womöglich ein neuer Ort für den Datensatz gefunden werden.

Die meisten DBMS organisieren den verwalteten Datenspeicher in Blöcken, die als *Pages* bezeichnet werden (siehe Abbildung 6.2). Die Blockgröße variiert je nach DBMS; 4 KiB, 8 KiB und 16 KiB sind die gebräuchlichsten Werte. Jeder *Page* ist eine Aufgabe zugeordnet – z. B. die Speicherung von Datensätzen einer bestimmten Tabelle, von großen binären Objekten oder von Zusatzinformationen (Indizes, Metadaten zum Inhalt der Pages etc.).

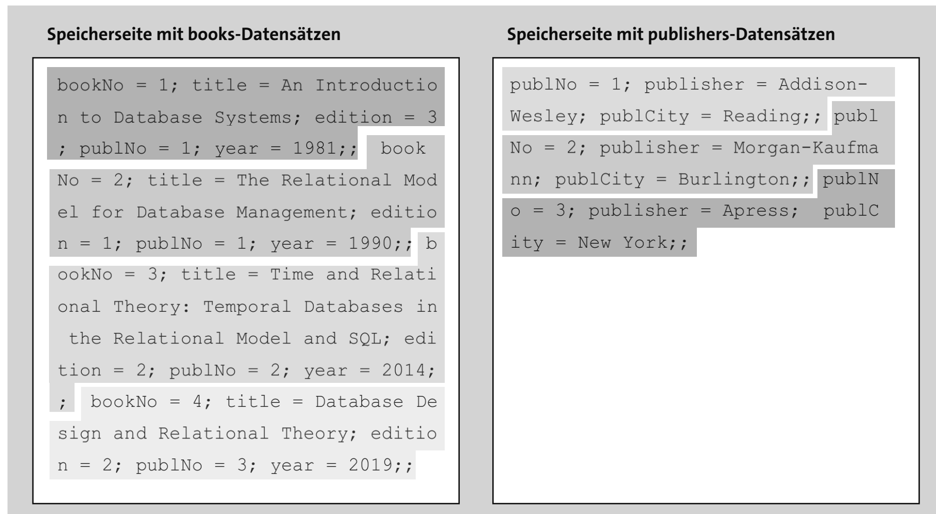

Abbildung 6.2 Stark vereinfachte Darstellung der Speicherung von Datensätzen in Blöcken (Pages)

Nach Möglichkeit versucht das DBMS, jeden Datensatz vollständig in einer Page zu speichern. Reicht der verbleibende Platz für den nächsten Datensatz nicht mehr aus, landet dieser in einer neuen Page; der restliche Speicherplatz der vorigen Page bleibt (vorerst) ungenutzt.

Längere Texte oder größere binäre Objekte werden aus Performancegründen getrennt vom restlichen Datensatz gespeichert. Denken Sie etwa an eine Personaltabelle, wo Name, Geburtsdatum, Versicherungsnummer und ein Foto jedes Mitarbeiters gespeichert werden. Die Basisdaten umfassen vielleicht 300 Byte, das Foto dagegen viele kByte. Deswegen ist es zweckmäßig, das Foto anders zu behandeln, zumal es für viele Datenbankoperationen – z. B. die monatliche Verrechnung des Gehalts – vollkommen irrelevant ist.

Die Pages befinden sich in einer gewöhnlichen Datei. Am gängigsten ist es, pro Datenbank eine große Datei vorzusehen, die sowohl Daten als auch Indizes aufnimmt. Je nach DBMS kann es sein,

- dass Daten und Indizes in getrennten Dateien gespeichert werden,
- dass jeder Tabelle (und womöglich sogar jedem Index) eine eigene Datei zugewiesen wird oder
- dass *sämtliche* Datenbanken inklusive aller Tabellen und Indizes in einer gemeinsamen, riesigen Datei gespeichert werden.

Bei der letzten Option verwaltet das DBMS die zentrale Datendatei in der Art eines eigenen Dateisystems. In logischer Konsequenz bieten manche Systeme die Option, ganze Datenträger als *Raw Devices* zu nutzen, also auf die Funktionen des Betriebssystems zur Verwaltung des Dateisystems komplett zu verzichten. Diese Vorgehensweise ist aber unüblich, weil sie Backups erschwert und sich nur schwer mit Netzwerkdatenträgern kombinieren lässt.

Alles eine Frage der Implementierung!

In diesem Abschnitt vermittle ich Ihnen Grundlagenwissen, wobei ich bei vielen Details zu Vereinfachungen greife. Datenbankmanagementsysteme funktionieren so ähnlich, wie ich es hier beschreibe, aber eben nicht exakt so. Das hat nicht nur den Grund, dass es mir hier primär um eine kompakte Darstellung elementarer Techniken geht, sondern auch, dass jedes DBMS ein wenig anders implementiert und optimiert ist. Jeder Hersteller bzw. Entwickler versucht, noch raffiniertere Wege als die Konkurrenz zu finden, um in der Praxis oder zumindest in marktüblichen Benchmarktests ein paar Prozent mehr Performance herauszuholen.

Fragmentierung

Durch das Löschen des Datensatzes entsteht dort, wo sich die Daten ursprünglich befanden, ungenutzter freier Speicherplatz. Je mehr derartige »Löcher« es in den Pages gibt, desto stärker gilt der Speicher als fragmentiert.

Bei Datenbanken, in denen vorhandene Daten häufig geändert werden (also nicht nur ständig neue Daten hinzugefügt werden), muss das DBMS nach Wegen suchen, den

ungenutzten Speicherplatz zu »recyclen«, also wiederverwendbar zu machen. Das Fragmentierungsproblem wird zumeist so gelöst, dass Seiten mit einem hohen Anteil ungenutzter Bytes komplett reorganisiert werden: Die noch relevanten Datensätze werden in eine neue Seite übertragen, danach wird die alte Seite als Speicherplatz für andere Daten freigegeben.

Es hängt von der Implementierung des DBMS ab, ob sich das System selbstständig im Hintergrund um derartige Aufräumarbeiten kümmert oder ob Sie als Systemadministrator(in) diesen Prozess gelegentlich selbst anstoßen müssen. Die meisten DBMS bieten darüber hinaus eine Möglichkeit, das Ausmaß der Fragmentierung je nach Tabelle festzustellen. Beachten Sie, dass die Fragmentierung nicht nur die eigentlichen Tabellendaten betreffen kann, sondern auch die dazugehörigen Pages mit dem Index.

Die Rolle des Index

Damit das DBMS beim Zugriff auf einen bestimmten Datensatz nicht jedes Mal alle Pages mit Datensätzen der jeweiligen Tabelle durchsuchen muss, benötigt es eine Art Inhaltsverzeichnis. Dieses gibt Auskunft darüber, an welchem Ort welcher Datensatz gespeichert ist. In der Nomenklatur der Datenbankwelt wird dieses Inhaltsverzeichnis als *Index* bezeichnet.

Auf den ersten Blick erscheint das Erstellen des Index eine triviale Aufgabe zu sein: Das DBMS reserviert eine Page für eine geordnete Minitabelle, deren Einträge wie die folgenden Beispiele aussehen:

Der Datensatz mit bookNo=1234 befindet sich in Page 57 ab Byte 478.
Der Datensatz mit bookNo=1235 befindet sich in Page 61 ab Byte 2998.

Bei einer kompakter Darstellung (4 Byte für die ID plus 4 Byte für die Ortsangabe) gelingt es, 1.000 oder 2.000 solcher Querverweise in einer Page zu speichern. Spannend wird es, wenn eine Page nicht mehr für alle Indexeinträge ausreicht: Sie müssen die Einträge jetzt über zwei Pages verteilen, und zwar so, dass die Sortierordnung erhalten bleibt.

Denken Sie bei der Organisation des Index nicht nur an IDs, die ständig größer werden und daher von Anfang an geordnet sind! Die Verwaltung des Index muss auch dann effizient funktionieren, wenn die Werte der betreffenden Spalte in ungeordneter Reihenfolge eingefügt werden, wenn Werte nachträglich verändert werden und wenn Datensätze (und damit auch die dazugehörenden Indexeinträge) später wieder gelöscht werden. Stellen Sie sich vor, Sie organisieren den Index für die Namen eine Mitarbeitertabelle einer großen Firma, die ständig neue Mitarbeiter einstellt und wieder entlässt. Für jeden neuen Mitarbeiter, für jede Mitarbeiterin muss ein Eintrag mitten im vorhandenen Index eingefügt werden.

Sie merken schon: Die Verwaltung eines Index, der auch dann noch effizient funktioniert, wenn Millionen von sich ständig ändernden Datensätzen im Spiel sind, ist alles andere als trivial.

B-Trees

Um große Mengen von Daten geordnet zu organisieren, greift die Informatik auf Baumstrukturen (*Trees*) zurück. In der Datenbankwelt hat sich eine besondere Variante durchgesetzt, der sogenannte *B-Tree*. Seine Besonderheit besteht darin, dass jeder Knoten der Baumstruktur einer ganzen Page entspricht und dementsprechend mehrere Hundert oder sogar einige Tausend Indexeinträge aufnehmen kann.

Hintergrund

Bei vielen anderen in der IT üblichen Datenstrukturen werden pro Knoten nur ein oder höchstens zwei Elemente gespeichert. Das ist zweckmäßig, wenn die Daten vollständig im RAM abgebildet werden können.

Bei einem DBMS befindet sich aber typischerweise immer nur eine Teilmenge der Daten im Arbeitsspeicher. Die restlichen Daten liegen in Pages auf der Festplatte oder SSD. Weil jeder Disk-Zugriff vergleichsweise langsam ist, muss eine DBMS-taugliche Datenstruktur dahingehend optimiert sein, die Anzahl der Page-Operationen, also Lesen/Ändern/Schreiben/Löschen, zu minimieren. Diese Überlegungen haben zur Entwicklung des B-Trees geführt.

Der wichtigste Parameter eines B-Trees ist die Ordnungszahl m, die die Anzahl der Elemente und Subknoten (*Kinder*) pro Knoten limitiert. Für einen B-Tree der Ordnung m gelten die folgenden Regeln:

- Jeder Knoten kann maximal $m-1$ Indexeinträge (Elemente) aufnehmen und auf maximal m Subknoten (Kinder) verweisen. Ein Knoten eines B-Trees mit der Ordnungszahl 3 kann also maximal 2 Elemente (Indexeinträge) aufnehmen und 3 Links auf untergeordnete Knoten haben (siehe Abbildung 6.3).

  Ich verwende in diesem Buch zur einfacheren Visualisierung B-Trees mit ganz kleinen Ordnungszahlen (meist 3). Bei echten DBMS ist die Ordnungszahl natürlich viel größer. Bei einem B-Tree der Ordnung 100 sind es dementsprechend 99 Elemente und 100 Verweise auf Subknoten.
- Jeder Knoten muss mindestens $m/2$ Subknoten haben. Eine Ausnahme gilt für den Startknoten (*Root-Knoten*), der auf weniger Subknoten verweisen darf. Auch die Endknoten (die »Blätter« des Baums) verweisen auf keine weiteren Knoten.
- Jeder Knoten, der auf k Subknoten verweist, muss $k-1$ Elemente (Indexeinträge) enthalten.

Das bedeutet, dass sich zwischen zwei Verweisen auf Subknoten immer ein Element befinden muss (siehe Abbildung 6.3). Umgekehrt bedeutet das aber auch, dass es nicht erlaubt ist, dass die Links auf Subknoten einseitig sind. Wenn ein Knoten ein Element enthält, muss es zwei Links auf Subknoten geben. Einer reicht nicht. Wenn der Knoten 20 Elemente enthält, *muss* es 21 Links geben, also jeweils links und rechts von jedem Element.

In Kombination mit der vorigen Regel ergibt sich damit auch die Mindestfüllung jedes Knotens: Bei einem Baum der Ordnung 3 muss jeder Knoten zumindest ein Element (einen Indexeintrag) enthalten. Bei einem Baum der Ordnung 100 darf es keinen Knoten geben, der weniger als 49 Elemente enthält.

- Jedes Element in einem Knoten dient als Trennwert für die Subknoten. Der linke Subknoten darf nur kleinere Elemente haben, der rechte nur größere.
- Für sämtliche Endknoten ist der Abstand zum Startknoten gleich. Der B-Tree muss also immer »ausbalanciert« sein. Wenn eine Einfüge- oder Löschoperation die Balance durcheinanderbringt, müssen Teile des Baums reorganisiert werden.

  Die Anzahl der Ebenen eines B-Trees wird *Tiefe* genannt.

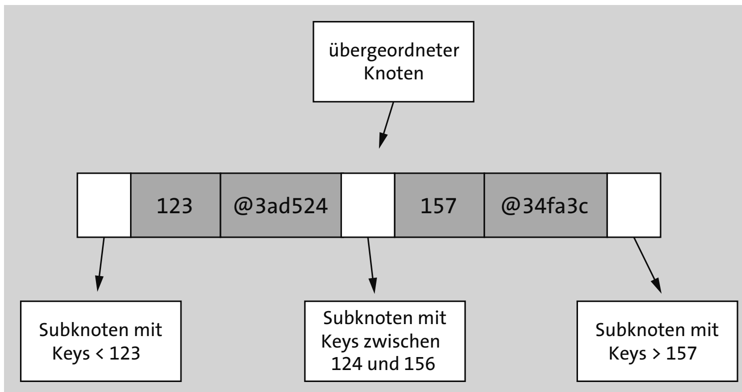

Abbildung 6.3 B-Tree-Knoten der Ordnung 3

Beachten Sie, dass die Literatur zu B-Trees uneinheitlich ist: Es variiert nicht nur der Buchstabe für die Ordnungszahl der Struktur (hier *m*), sondern auch die Bedeutung dieser Zahl. Je nach Autor bezieht sich die Ordnungszahl auf die maximale Anzahl der Subknoten, auf die maximale Anzahl der Elemente (die ist um eins kleiner!) oder auf die Hälfte der Maximalanzahl der Elemente.

Abbildung 6.3 zeigt einen voll besetzten Knoten für einen B-Tree mit der Ordnungszahl 3. Der Knoten enthält zwei Indexeinträge für die IDs 123 und 157 samt der hexadezimal codierten Information, in welcher Page und wo dort sich der zugehörende Datensatz befindet. Außerdem gibt es drei Links auf untergeordnete Knoten, einmal für IDs kleiner 123, einmal von 124 bis 156 und einmal größer als 157.

Suchvorgänge im B-Tree starten immer an der Spitze des Baums. Wenn das gesuchte Element nicht direkt gefunden wird, ist in jedem Fall klar, in welchem Subknoten die Suche fortgesetzt wird. Es müssen also immer nur so viele Knote durchsucht bzw. Pages vom Datenträger in den Arbeitsspeicher übertragen werden, wie der Baum tief ist.

Dazu ein kleines Rechenbeispiel:

- Nehmen Sie an, die Page-Größe beträgt 8 KiB = 8192 Byte.
- Pro Element (also pro Indexeintrag) benötigen Sie 16 Byte Speicherplatz (je 8 Byte für eine 64-Bit-ID sowie für eine Adressangabe).
- Außerdem benötigen Sie für jeden Link zu einem anderen Knoten 4 Byte.
- Damit können Sie maximal 409 Elemente (Indexeinträge) sowie 410 Links pro Knoten/Page speichern. $409 \times 16 + 410 \times 4 = 8184$. Sie haben damit einen B-Tree der Ordnung 410.
- Gehen Sie davon aus, dass jeder Knoten (jede Page) ca. zu 60 Prozent gefüllt ist. Das ergibt 245 Elemente bzw. Indexeinträge pro Knoten.
- Unter diesen Voraussetzungen reicht ein Baum der Tiefe 3 aus, um 245^3 = 14.706.125 Datensätze zu indizieren. Der Speicherplatz für den Index beträgt dann 1 + 245 + 245^2 = 60.271 Pages mit jeweils 8 KiB, das sind ca. 471 MiB.

  Wenn wir annehmen, dass jeder Datensatz 1 KiB Platz beansprucht, dann beträgt der Speicherplatz für die Tabelle knapp 15 GiB. Für den Index kommt ca. ein halbes GiB hinzu. Der Preis für eine schnelle Suche ist somit gut 3 Prozent mehr Speicherplatz und natürlich viel Verwaltungsarbeit zur regelmäßigen Aktualisierung des Index.
- Mit einem Baum der Tiefe 4 kommen Sie bereits auf 3,6 Milliarden Datensätze. Trotz dieser enormen Anzahl reicht es aus, nur vier Knoten zu durchsuchen, um die Adresse des gesuchten Datensatzes herauszufinden!

B-Trees sind also Baumstrukturen, die sich im Vergleich zu anderen Baumtypen durch eine geringe Tiefe auszeichnen. Dafür sind B-Trees typischerweise sehr breit, haben also pro Knoten sehr viele Subknoten.

Grundsätzlich funktionieren B-Trees unabhängig vom Datentyp der indizierten Spalte. Dass ich hier mit relativ kleinen ID-Zahlen arbeite, dient nur der Übersichtlichkeit. Ebenso gut wären Datum- und Zeitangaben sowie Zeichenketten geeignet. Allerdings bringen Zeichenketten wegen ihrer variablen Länge zusätzliche Komplikationen mit sich. Die Anzahl der Indexeinträge, die pro Knoten Platz haben, ist bei Zeichenketten nicht exakt vorhersehbar. Bei vielen DBMS können oder müssen Sie deswegen beim Einrichten eines Index für eine Textspalte festlegen, wie viele Zeichen maximal indiziert werden.

Elemente in B-Trees einfügen

Für das Einfügen von Elementen (also Indexeinträgen) in einen B-Tree gibt es klare Regeln:

- Zuerst wird der Knoten gesucht, in das das neue Element gehört.
- Wenn der Knoten noch Platz für das neues Element hat, wird es dort eingefügt.
- Andernfalls wird der Knoten beim mittleren Element (Median) in zwei neue Knoten zerlegt. Das Median-Element wird in den übergeordneten Knoten eingebaut. Die beiden neuen Knoten sind jetzt exakt halb voll.
- Sollte der übergeordnete Knoten aufgrund des zusätzlichen Median-Elements ebenfalls seine Kapazität überschreiten, wird auch dieser Knoten zerlegt. Sein Median-Element rutscht eine Ebene nach oben – und so weiter …

Im folgenden Beispiel (siehe Abbildung 6.4) werden in einen anfangs leeren B-Tree die Indexeinträge 17, 12, 8, 10, 11, 13, 14 und 15 eingefügt. Der Baum wächst dabei von einer auf schließlich drei Ebenen. Um Platz zu sparen, zeigt die Abbildung nur die Werte der indizierten (ID-)Spalte. Tatsächlich müssen natürlich weiterhin auch der Ort des dazugehörenden Datensatzes sowie die Verweise auf die untergeordneten Knoten gespeichert werden.

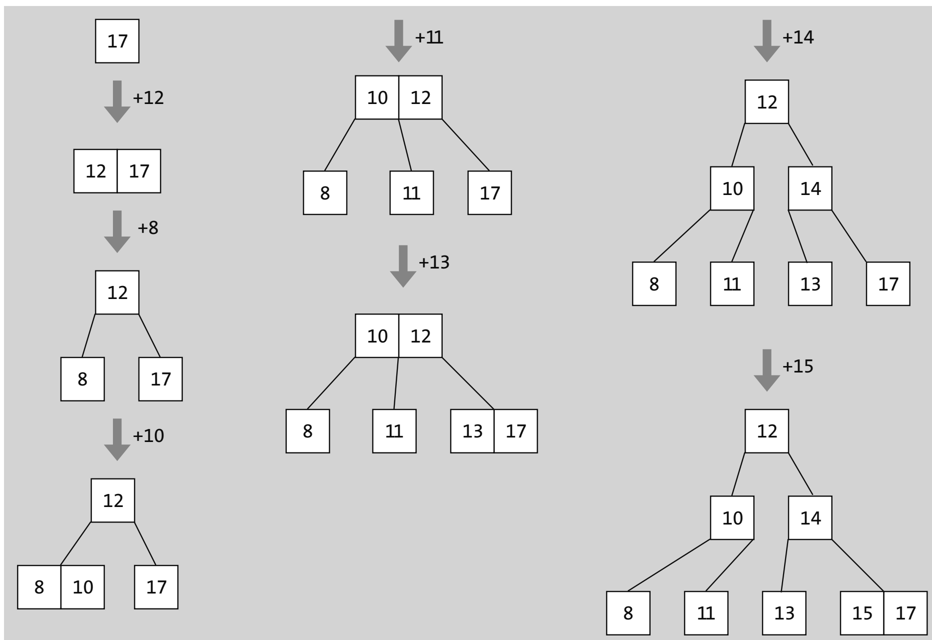

Abbildung 6.4 Einfügeoperationen in einen B-Tree der Ordnung 3

Elemente aus B-Trees löschen

Analog gibt es auch Regeln für das Löschen von Elementen. Diese unterscheiden sich, je nachdem, ob sich der Knoten in der Mitte des Baums oder am Ende befindet (ob es sich also um ein Blatt des Baumes, ein *Leaf*, handelt). Bei Blättern kann das Element einfach entfernt werden. Wenn der Knoten danach weniger als halb voll ist, muss der Baum neu ausbalanciert werden.

Das Entfernen von Elementen aus den inneren Knoten ist insofern schwieriger, als der Baum bei jedem Element in zwei Teile zerlegt wird. Als neues Trennelement eignet sich das letzte Element des linken Subknotens oder das erste Element des rechten Subknotens. Eines davon wird aus dem Subknoten entfernt und anstelle des gelöschten Elements eingebaut. (Grundsätzlich haben Sie die freie Wahl, aus welchem Subknoten Sie das Element verschieben. Damit die Nutzung der Subknoten möglichst ausgeglichen bleibt, werden Sie dasjenige Element aus dem Subknoten nehmen, das mehr Elemente enthält.) Wenn durch das Verschieben des Elements der betroffene Subknoten weniger als halb voll ist, müssen Sie den Baum neu ausbalancieren.

Damit kommen wir zu den Regeln für das Ausbalancieren: Der Vorgang beginnt immer bei einem Endknoten (*Leaf*), der weniger als *m/2* Elemente enthält. Wenn es einen benachbarten Knoten (*Sibling*) mit genug Elementen gibt, können Sie einfach ein Element in den übergeordneten Knoten und dafür dessen Trennelement in den zu kleinen Knoten verschieben. Dieser Vorgang wird *Rotation* genannt und lässt sich einfach bewerkstelligen.

Wenn es keinen Nachbarknoten gibt, der für eine Rotation in Frage kommt, müssen Sie zwei benachbarte Knoten zu einem zusammenfügen (*Merge*). Allerdings müssen Sie dazu dem übergeordneten Knoten ein Element entnehmen und in den neuen Knoten einbauen. Es kann nun dazu kommen, dass auch der übergeordnete Knoten zu wenig Elemente hat – dann müssen Sie das Ausbalancieren an dieser Stelle fortsetzen.

B-Tree-Algorithmen

Falls Sie selbst Klassen zur Verwaltung von B-Trees programmieren möchten, finden Sie auf der englischen Wikipedia-Seite detaillierte Algorithmen:

https://en.wikipedia.org/wiki/B-tree

Am leichtesten sind Löschvorgängen in B-Trees anhand von Beispielen zu verstehen (siehe Abbildung 6.5): Vollkommen unkompliziert ist das Löschen von Element 15. Die Elementanzahl des Knotens 15/17 sinkt dadurch zwar, unterschreitet aber nicht das Minimum.

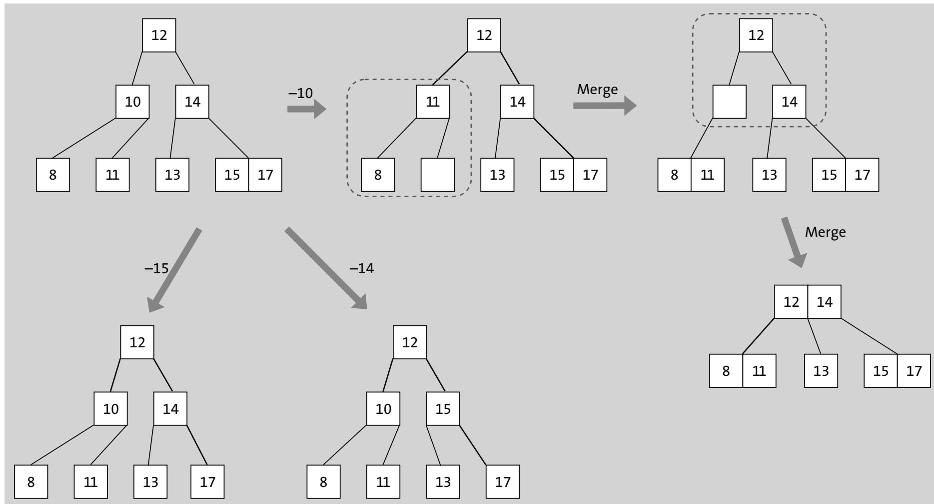

Abbildung 6.5 Löschoperationen in einem B-Tree der Ordnung 3

Beinahe ebenso einfach ist der Löschvorgang für Element 14. Damit geht das Trennelement des Knotens verloren. Es kann aber unkompliziert durch das erste Element des rechten Subknotens ersetzt werden, also durch 15 (Rotation).

Deutlich komplizierter ist der Löschvorgang für Element 10. Egal, ob Element 8 oder 11 nach oben bewegt (rotiert) wird – in jedem Fall unterschreitet einer der unteren Knoten die minimale Elementanzahl. Im Zuge des nun notwendigen Ausbalancierens werden Knoten 8 und der leere Knoten zusammengefügt. Damit wandert Element 11 wieder nach unten, der Knoten, der ursprünglich 10 enthielt, ist jetzt leer. Die Lösung ist ein zweiter Merge-Prozess, durch den die Tiefe des Baums von 3 auf 2 sinkt.

B-Tree-Simulator

Am besten lernen Sie die Funktionsweise von B-Trees interaktiv in einem Simulator kennen. Es gibt im Internet mehrere Seiten, die alle Vorgänge (Suche, Einfügen, Löschen, erneutes Ausbalancieren) visualisieren können – beispielsweise hier:

https://www.cs.usfca.edu/~galles/visualization/BTree.html
https://yangez.github.io/btree-js

Aus Gründen der Übersichtlichkeit werden B-Trees in Büchern oder im Internet zumeist mit einer niedrigen Ordnung dargestellt (in diesem Buch: Ordnung 3). Beim Arbeiten mit solchen Bäumen entsteht der Eindruck, dass nahezu jede Einfüge- oder Löschoperation eine umfassende Reorganisation des Baums mit sich bringt.

In einem DBMS ist die Ordnung von B-Trees aber viel größer und liegt oft im Bereich 200 bis 2.000. Erst jetzt offenbart sich die Effizienz von B-Trees. Eine Reorganisation des Baums tritt nun viel seltener auf – und kann vergleichsweise effizient (d. h. mit nur wenigen I/O-Operationen) durchgeführt werden. Zugleich müssen bei einem Suchvorgang nach einem singulären Wert nur drei oder vier Pages geladen werden – selbst bei Tabellen mit Millionen vor Datensätzen.

B steht für ...?

Es ist nicht bekannt, welche Bedeutung der Buchstabe »B« in B-Trees hat. Auf jeden Fall nicht für *Binary*! (Binäre Bäume sind eine andere etablierte Datenstruktur. Im Unterschied zu B-Trees kann sich aber jeder Knoten nur in zwei weitere Elemente aufteilen.)

Mögliche Kandidaten für »B« sind *Balanced*, *Block-based*, *Boeing* (die Firma, in der B-Trees entwickelt wurden) oder *Bayer* (einer der Entwickler). *McCreight*, der B-Trees zusammen mit *Bayer* erfunden hat, meint dazu mit feiner Ironie: »The more you think about what the B in B-trees means, the better you understand B-trees.«

Alternativen und Varianten zum B-Tree

Neben dem klassischen B-Tree gibt es zur Organisation des Index einer Tabelle diverse Varianten bzw. Alternativen, die ich hier nur kurz aufzählen möchte:

- Beim **B+-Tree** enthalten nur die Endknoten (»Blätter«) Key-Value-Einträge, wobei als Schlüssel die zu indizierenden Werte dienen, als Werte die Querverweise auf Pages der Datenbank. Die Knoten in den übergeordneten Ebenen enthalten dagegen nur Keys sowie Verweise auf andere Knoten. Außerdem sind die Knoten einer Ebene durch direkte Links (ähnlich einer *Linked List*) miteinander verbunden.

  Aufgrund dieser Änderungen gibt es in B+-Trees eine gewisse Redundanz, die in gewöhnlichen B-Trees nicht auftritt. Dafür können aber häufig vorkommende Operationen schneller durchgeführt werden. B+-Trees kommen in einigen marktüblichen relationalen DBMS zum Einsatz, unter anderem in Microsoft SQL Server, in Oracle und in SQLite. Auch manche NoSQL-Systeme (z. B. CouchDB) verwenden intern B+-Trees.

- Beim **B*-Tree** müssen Knoten zu mindestens 2/3 gefüllt werden. Knoten, die voll sind, werden nicht sofort in zwei neue Teilknoten getrennt; stattdessen wird versucht, Elemente in Nachbarknoten zu verschieben. Diese beiden Maßnahmen führen dazu, dass Knoten im Schnitt dichter befüllt sind und weniger häufig in zwei neue Knoten zerlegt werden müssen.

  Beachten Sie, dass Original-B-Trees, B+-Trees, B*-Trees und andere Varianten oft gesammelt als »B-Trees« bezeichnet werden.

- **Prefix Trees** bzw. **Tries** eignen sich besonders gut zur platzsparenden Indizierung von Zeichenketten. Dabei bildet jede Ebene des Baums jeweils einen Buchstaben des zu indizierenden Worts ab. Damit müssen nicht jeweils ganze Zeichenketten als Indexwerte gespeichert werden, sondern immer nur der nächste Buchstabe (siehe Abbildung 6.6).

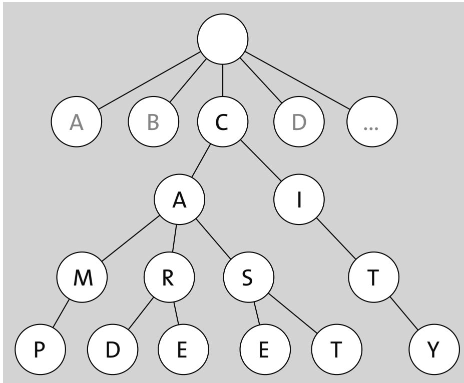

Abbildung 6.6 Prefix Tree für die Wörter »camp«, »card«, »care«, »case«, »cast« und »city«

Das Konzept von *Prefix Trees* ist auch für andere Datentypen geeignet. Beispielsweise kann jede Ebene des Baums einige Bits oder ein Byte der betreffenden Daten abbilden.

- **Hash Tables:** Anstatt die betreffende Spalte direkt zu indizieren, kann ein Hash-Wert des Spalteninhalts errechnet und dieser indiziert bzw. in einer sogenannten *Hash Table* gespeichert werden. Das bietet sich abermals für lange Zeichenketten oder für binäre Daten (BLOBs) an, weil Hash-Werte wenig Platz beanspruchen und extrem schnelle Suchvorgänge ermöglichen. Dem stehen aber mehrere Nachteile entgegen:
  - Die Berechnung von Hash-Werten kostet zusätzliche Zeit.
  - Unterschiedliche Daten können zum selben Hash-Wert führen. Das Zugriffssystem muss mit solchen Doppelgängern zurechtkommen.
  - Hash-Werte ermöglichen zwar den schnellen Zugriff auf *ein* gesuchtes Element, bieten aber keine Möglichkeit, Datensätze zu ordnen oder alle Datensätze zu finden, die in einem Wertebereich liegen bzw. die dem Suchwert ähnlich sind.

Clustered Index versus Non-clustered Index

Bis jetzt habe ich Ihnen das Konzept eines Index präsentiert, der von den eigentlichen Daten einer Tabelle vollkommen losgelöst ist: Es gibt also Pages, die Datensätze enthalten, und andere Pages mit Indexeinträgen, die auf die Datensätze verweisen

(siehe Abbildung 6.7). Diese Art des Index wird als *Non-clustered Index* bezeichnet, im Deutschen mitunter auch als *nicht-gruppierter Index*.

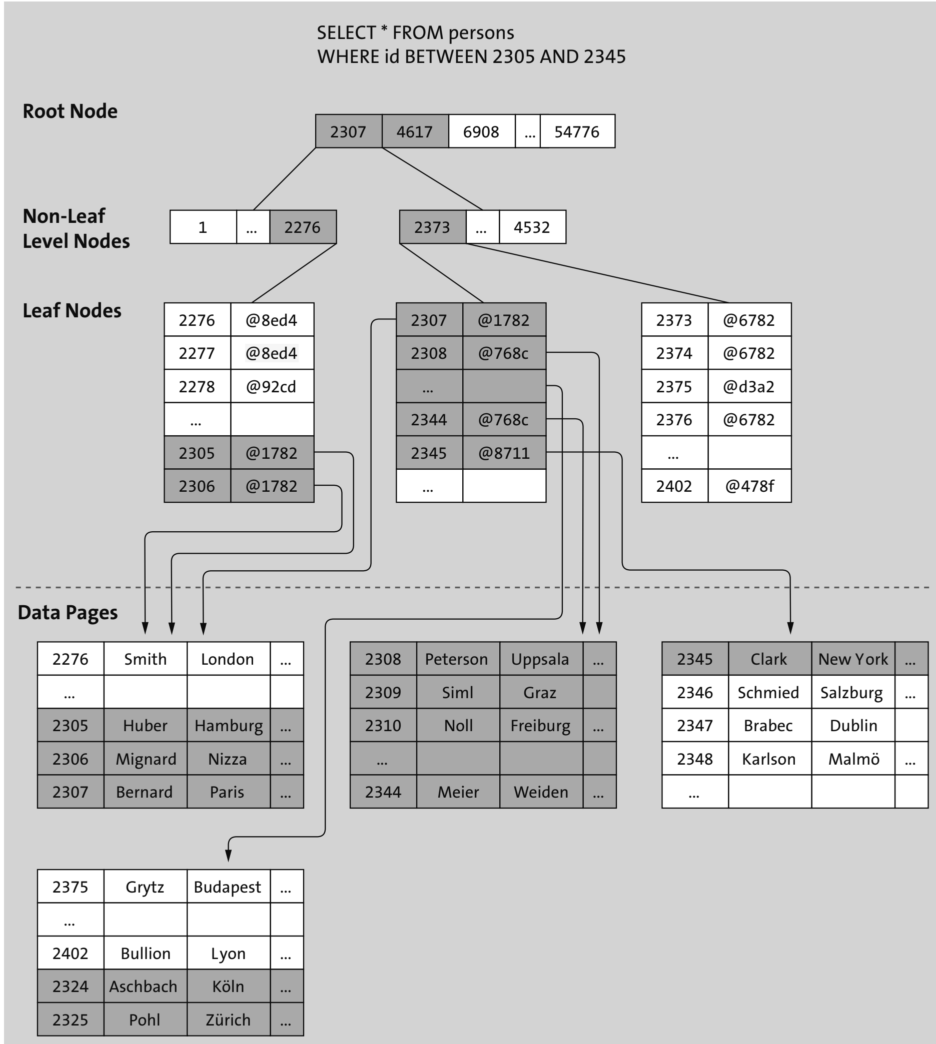

Abbildung 6.7 Schematische Darstellung der physikalischen Speicherung einer Personentabelle mit einem Non-clustered Index. Jeder Knoten des B+-Trees entspricht einer Page. Die durch das SELECT-Kommando gesuchten Datensätze sind hervorgehoben.

Beachten Sie, dass die Datensätze trotz der durchlaufenden IDs nicht geordnet in den Pages gespeichert sind: Wenn sich Datensätze später ändern und dann mehr Platz brauchen, müssen sie aus ihrer bisherigen Page entfernt und in einer neuen Page gespeichert werden. Ich habe diesen Fall für die beiden Datensätze mit den IDs 2324 und 2325 dargestellt (weiterhin in Abbildung 6.7).

Ein entscheidendes Kriterium für die Geschwindigkeit eines DBMS ist die Anzahl der I/O-Vorgänge (*Input/Output*), die notwendig sind, um einen Datensatz vom Datenträger in den Arbeitsspeicher zu bringen. Die `SELECT`-Abfrage für alle Personen mit einer ID von 2305 bis 2345 erfordert in der symbolischen Darstellung den Zugriff auf neun Pages. Das Verfolgen von Querverweisen von der Index-Page in eine Datensatz-Page scheint da eine unnötige Bremse zu sein. Besteht denn keine Möglichkeit, die Datensätze und den Index zusammenzuführen?

Tatsächlich gibt es einen Lösungsansatz, der unter dem Namen *Clustered Index* weite Verbreitung gefunden hat. Oracle nennt derartige Tabellen *Index-organized Tables*. Kurz gefasst sieht das Konzept so aus (wie üblich mit Implementierungsvariationen je nach DBMS):

- Die Endknoten (*Leaf Nodes*) enthalten Indexeinträge samt den dazugehörigen Datensätzen. Im Hinblick auf die indizierte Spalte werden die Datensätze also immer geordnet abgespeichert. Damit gilt also wie bei einem altmodischen Telefonbuch: *The data is the index.*
- Die Knoten auf allen darüberliegenden Ebenen enthalten ausschließlich Indexeinträge plus Querverweise auf andere Knoten des B-Trees (siehe Abbildung 6.8). Anders als beim Original-B-Tree dienen die Root- und Non-Leaf-Knoten also ausschließlich zur Navigation.

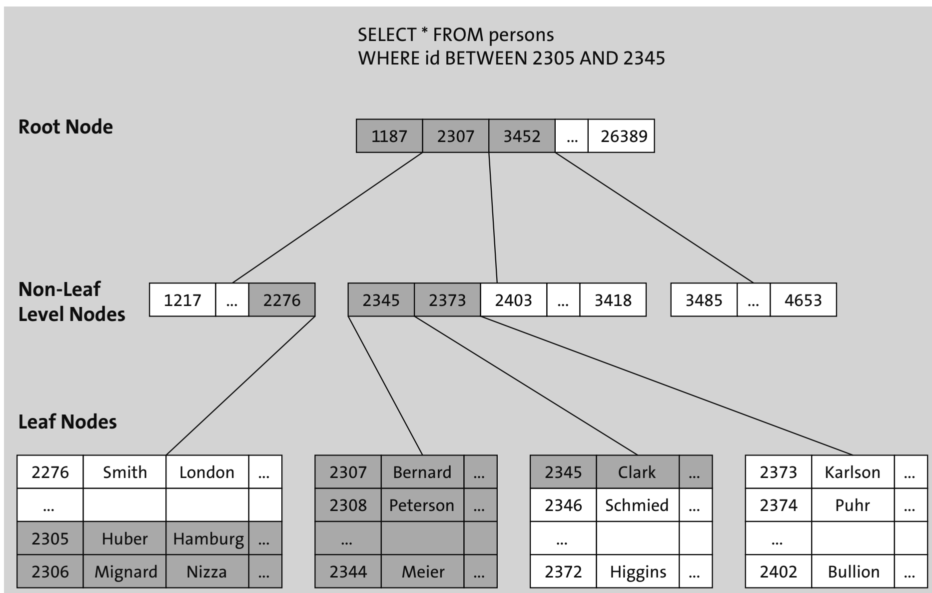

Abbildung 6.8 Personentabelle mit einem Clustered Index. Jeder Knoten des B+-Trees entspricht einer Page.

Am besten funktioniert ein Clustered Index, wenn der Speicherbedarf pro Datensatz relativ klein ist (also viele Datensätze in einer Page Platz finden) und wenn es häufig Abfragen gibt, bei denen – gemäß der durch die indizierten Spalte vorgegebenen Ordnung – aufeinanderfolgende Datensätze gelesen werden sollen. In der schematischen Darstellung der Personentabelle (siehe Abbildung 6.8) müssen insgesamt nur sechs Pages verarbeitet werden, um alle Datensätze zu lesen, deren ID im Bereich 2305 bis 2345 liegt.

Bei vielen DBMS wird der Primärindex immer als Clustered Index realisiert. Andere Programme bieten Ihnen durch zusätzliche Schlüsselwörter die Option, einen Index Ihrer Wahl als Clustered Index zu kennzeichnen. Beachten Sie aber, dass eine nachträgliche Änderung des Clustered Index extrem aufwändig ist: Dazu müssen sämtliche Datensätze der Tabelle reorganisiert werden. Bei großen Tabellen kann das Stunden dauern.

Ein Clustered Index bringt also in bestimmten Fällen spürbare Vorteile mit sich. Aber auch diese Indexvariante ist mit Einschränkungen und Nachteilen verbunden:

- Pro Tabelle kann es nur einen Clustered Index geben. Sämtliche Datensätze werden ja physikalisch geordnet gespeichert. Dabei kann es nur *ein* Ordnungskriterium geben, das durch den Clustered Index vorgegeben ist. Alle weiteren Indizes sind weiterhin »gewöhnliche« Non-clustered Indizes, deren Einträge auf die Leaf Nodes des Clustered Index verweisen.
- Veränderungen von Datensätzen (`INSERT`, `UPDATE` und `DELETE`) verursachen mehr Schreibzugriffe. Im Vergleich zu einem Non-clustered Index ist es häufiger notwendig, die Leaf Nodes samt den darin enthaltenen Datensätzen zu reorganisieren (z. B. auf zwei Pages aufzuteilen). In diesem Fall müssen auch die Querverweise auf diese Datensätze in allen Non-clustered Indizes aktualisiert werden.

Locking

Bereits in Abschnitt 2.4, »Datensicherheit und ACID«, habe ich Ihnen den Begriff *Isolation* vorgestellt: Im Kontext von relationalen DBMS bedeutet Isolation, dass sich zwei parallel durchgeführte Transaktionen nicht gegenseitig beeinflussen dürfen. Diese Forderung ist schnell aufgestellt – aber wie wird sie tatsächlich garantiert?

In irgendeiner Form muss sich das DBMS »merken«, welche Datensätze eine Transaktion bearbeitet. Im einfachsten Fall werden diese Datensätze für alle andere Transaktionen vorübergehend durch einen Locking-Mechanismus gesperrt. Solange eine Transaktion nur einen einzigen Datensatz oder zumindest nur wenige Datensätze betrifft (`UPDATE tablename WHERE id=1234 ...`), bereitet das Locking wenig Arbeit. Wesentlich aufwändiger wird es, wenn eine Transaktion mehrere Tausend Datensätze betrifft.

Und genau hier kommen Indizes ins Spiel: Wenn das Auswahlkriterium für die durch eine Transaktion veränderten Datensätze einem geordneten Bereich von Werten einer indizierten Spalte entspricht, dann kann das DBMS den Index für das Locking verwenden und z. B. vorübergehend alle Datensätze für andere Transaktionen sperren, auf die einige Pages des Index verweisen. (Natürlich wird die Grenze des Bereichs der zu sperrenden Datensätze selten exakt mit den Index-Pages übereinstimmen. Es werden dann einfach alle Datensätze gesperrt, auf die die betroffenen Index-Pages verweisen – also vielleicht deutlich mehr als notwendig. Das ist zwar nicht optimal, aber immer noch besser, als die gesamte Tabelle zu sperren.)

Indizes können nicht nur als ein Hilfsmittel zum Locking verwendet werden, sie müssen auch selbst durch Locking geschützt werden: Es darf nicht sein, dass zwei Transaktionen unabhängig voneinander Änderungen in einer Index-Page vornehmen. Dieser Fall muss durch ein sogenanntes *Index Locking* abgesichert werden (siehe auch *https://en.wikipedia.org/wiki/Index_locking*).

In diesem Buch greife ich Locking im Kontext von Transaktionen noch einmal auf. In Kapitel 12, »Transaktionen«, beschreibe ich verschiedene Locking-Typen und -Verfahren.

6.3 Indizes – Pro und Kontra

Vielleicht haben Sie nach der Lektüre der vorangegangenen Seiten den Eindruck gewonnen, dass Sie sämtliche Performanceprobleme lösen, indem Sie einfach jede Spalte Ihrer Tabellen mit einem Index ausstatten. Diese Idee ist leider total falsch! Wie Medikamente haben leider auch Indizes Nebenwirkungen:

- Indizes erfordern Platz, sowohl im Arbeitsspeicher als auch auf den Datenträgern (Festplatten oder SSDs). Wie groß der Platz relativ zu den eigentlichen Daten ist, hängt natürlich stark von der Art der Daten und von der Anzahl der Indizes ab. Bei Datenbanken, in denen überwiegend kurze Texte und Zahlen gespeichert werden, können Indizes durchaus 20 bis 50 % zusätzlichen Platz beanspruchen, in Extremfällen auch mehr. Wenn Ihre Datenbank dagegen viele BLOBs (binäre Daten) enthält, die üblicherweise nicht indiziert werden (können), werden die Indizes im Vergleich zum gesamten Datenbedarf vernachlässigbar sein.
- Indizes beschleunigen Abfragen, also Leseoperationen. Sie verlangsamen aber den Schreibbetrieb, ganz egal, ob Daten eingefügt, geändert oder gelöscht werden. In allen Fällen müssen die Indizes aktualisiert werden, wobei der Aufwand je nach Setup und Art der Daten pro Index ähnlich hoch wie zur Speicherung der eigentlichen Daten ist.

Welcher Isolation Level für welchen Zweck?

Viele Anwender belassen den vom DBMS vorgegebenen Defaultlevel und machen sich weiter keine Gedanken über die Isolation. Das geht oft gut – Transaktionen funktionieren einfach. Bei allen im nächsten Abschnitt beschriebenen Phänomenen ist das Timing der entscheidende Faktor: Wenn der zeitliche Ablauf nicht gerade ausgesprochen ungünstig ist, wenn Sie also nicht besonderes Pech haben, werden selbst im schwächsten Isolation Level des DBMS, ja selbst bei einem Verzicht auf Transaktionen, keine Fehler auftreten.

Probleme treten in der Regel erst auf, wenn richtig viele Transaktionen zur gleichen Zeit ausgeführt werden müssen. Wenn Sie zum ersten Mal in Ihrer Datenbank Daten entdecken, die nicht korrekt sein können – was dann? Ihr Verdacht als Datenbankadministrator richtet sich dann vielleicht an die Anwendungsentwickler, deren Code natürlich Fehler haben kann. Aber es wird sehr schwer sein, einen Fehler aufgrund unzureichender Isolierung der Transaktionen auszuschließen.

Insofern empfehle ich Ihnen, Ihr DBMS für Datenbanken mit unternehmenskritischen Daten so zu konfigurieren, dass Transaktionen standardmäßig im höchstmöglichen Isolation Level (also *Serializable*) ausgeführt werden. Oft wird das DBMS durch diese Einstellung nur um wenige Prozent langsamer.

Egal, ob Sie den Isolation Level ändern oder nicht: Hinterfragen Sie, wie gut die Trennung von Transaktionen tatsächlich funktioniert. Machen Sie sich die Mühe, und probieren Sie die im folgenden Abschnitt vorgestellten Beispiele aus!

12.3 Dirty Read, Phantom Read und andere Isolation-Probleme

In diesem Beispiel zeige ich Ihnen, wie je nach gültigem Isolation Level verschiedene Fehler auftreten können. Die folgenden Beispiele vermitteln gleichzeitig eine gute Vorstellung davon, wie Transaktionen funktionieren. Sie sollten sich unbedingt die Mühe machen, die Beispiele selbst nachzuvollziehen!

Nachdem ich in den vorigen Abschnitten auf fünf verschiedene DBMS eingegangen bin, setze ich im Folgenden wieder voraus, dass Sie mit MySQL arbeiten. Als Ausgangspunkt für die Beispiele verwende ich die Tabelle *t3* aus der Musterdatenbank *books*. Diese Tabelle ist denkbar einfach aufgebaut:

```
SELECT * FROM t3 ORDER BY id

  id   data
  ---  ----
   1    100
   2    100
   3    250
   4   NULL
```

```
   5     200
   6     300
   7    1000
   8     500
   9    NULL
  10     700

SELECT SUM(data) FROM t3
  3150
```

id ist der Primärschlüssel. *data* hat einfach den Datentyp `INT`, wobei NULL erlaubt ist. Wenn Sie den Inhalt der Tabelle durch eine Transaktion verändert haben, können Sie so den Grundzustand wiederherstellen:

```
-- alle Datensätze löschen ...
DELETE FROM t3

-- ... und wieder herstellen
INSERT INTO t3 (id, data)
VALUES (1, 100), (2, 100), (3, 250), (4, NULL), (5, 200),
       (6, 300), (7, 1000), (8, 500), (9, NULL), (10, 700)
```

Transaktionen testen

Um das Verhalten von Transaktionen und das Ausmaß der Isolation zwischen parallel ausgeführten Transaktionen zu testen, benötigen Sie zumindest zwei voneinander getrennte Datenbankverbindungen. In der MySQL Workbench ist das nicht möglich; auch wenn Sie mehrere Query Tabs öffnen, teilen diese eine gemeinsame Verbindung zur Datenbank. Sie können sich davon mit `SELECT CONNECTION_ID()` überzeugen.

Deswegen ist es besser, zwei Terminal-Fenster zu öffnen und in jedem Fenster mit dem Kommando `mysql` eine Verbindung zum lokalen MySQL-Server herzustellen (siehe Abbildung 12.1).

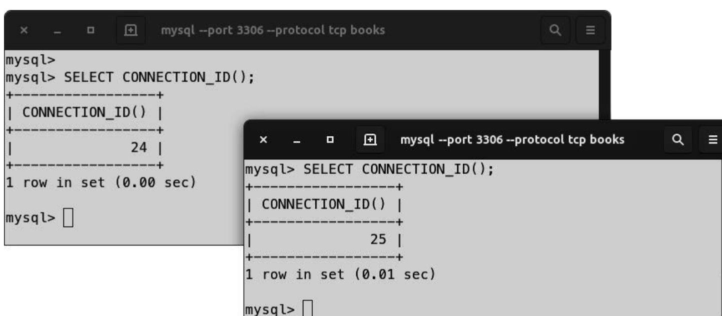

Abbildung 12.1 Zum Testen von Transaktionen brauchen Sie zwei Fenster mit eigenen Verbindungen zum Datenbank-Server.

Was ist wirklich wahr?

Bevor ich auf die klassischen Isolation-Probleme eingehe, möchte ich Ihnen mit dem folgenden Beispiel klarmachen, dass bei vielen DBMS alle Transaktionen eine eigene Sicht auf die Datenbank haben. Das folgende Beispiel funktioniert so in MySQL im Default-Level *Repeatable Read*. Das zweispaltige Listing gibt den Zeitverlauf an, in dem die Kommandos in den beiden Verbindungen ausgeführt werden.

```
-- Verbindung 1                  -- Verbindung 2
SET SESSION TRANSACTION          SET SESSION TRANSACTION
ISOLATION LEVEL                  ISOLATION LEVEL
REPEATABLE READ                  REPEATABLE READ

START TRANSACTION
                                 START TRANSACTION
-- Zeitpunkt 1
DELETE FROM t3
WHERE id > 8

SELECT SUM(data) FROM t3
  2450
                                 -- Zeitpunkt 2
                                 SELECT SUM(data) FROM t3
                                   3150

                                 -- Zeitpunkt 3
                                 UPDATE t3 SET data=data+200
                                 WHERE id=5

                                 SELECT SUM(data) FROM t3
                                   3350

-- Zeitpunkt 4
COMMIT

SELECT SUM(data) FROM t3
  2450
                                 -- Zeitpunkt 5
                                 SELECT SUM(data) FROM t3
                                   3350

                                 -- Zeitpunkt 6
                                 COMMIT

SELECT SUM(data) FROM t3         SELECT SUM(data) FROM t3
  2650                             2650
```

Was geht hier vor? Es gibt zwei Verbindungen zur Datenbank. In beiden Verbindungen beginnen ziemlich gleichzeitig Transaktionen. Zum Zeitpunkt 1 werden in Verbindung 1 einige Datensätze aus *t3* gelöscht. Die Summe über die Spalte *data* für alle verbleibenden Datensätze ergibt `2450`.

Verbindung 2 ist von Verbindung 1 isoliert, sie weiß also nichts davon, welche Datenbankoperationen in anderen Verbindungen gerade passieren. Zum Zeitpunkt 2 ist die Summe über *data* noch `3150`.

Zum Zeitpunkt 3 ändert auch Verbindung 2 einen Datensatz. Die neuerliche Summenberechnung berücksichtigt diese Änderung.

Zum Zeitpunkt 4 wird die Transaktion in Verbindung 1 mit `COMMIT` abgeschlossen. In der »echten« Datenbank werden jetzt alle Datensätze aus *t3* mit *id* > 8 gelöscht.

Erstaunlicherweise liefert die neuerliche Summenberechnung zum Zeitpunkt 5 in der noch immer laufenden Transaktion 2 weiterhin `3350` – als wären die Datensätze nie gelöscht worden. Das erscheint unlogisch, entspricht aber der zentralen Aussage des Levels *Repeatable Read*: Zwei gleiche `SELECT`-Kommandos innerhalb einer laufenden Transaktion müssen reproduzierbar immer zum gleichen Ergebnis kommen (es sei denn, Sie haben die betreffenden Daten im Rahmen Ihrer eigenen Transaktion selbst verändert)!

Zum Zeitpunkt 6 wird die Transaktion in Verbindung 2 beendet. Jetzt werden auch die Änderungen der zweiten Transaktion dauerhaft gespeichert. Die Summenberechnung kommt nun in beiden Verbindungen zu neuen Ergebnissen.

Während der Ausführung von Transaktionen hat also jede Transaktion ihre eigene »Wahrheit«, die sich vom Zustand der Datenbank zum Beginn der Transaktion sowie von den seither innerhalb der eigenen Transaktion ausgeführten Kommandos ergibt.

Die Parallelität von Transaktionen hat allerdings Grenzen. Sobald ein DBMS Konflikte erkennt, blockiert es Transaktionen. Bei diesem Beispiel lässt MySQL die parallele Bearbeitung der Tabelle deswegen zu, weil Datensätze mit unterschiedlichen *id*-Werten betroffen sind.

Wenn Sie das gleiche Beispiel im Level *Serializable* ausprobieren, wird die zweite Transaktion dagegen zum Zeitpunkt 3 blockiert. Das `UPDATE`-Kommando wird erst ausgeführt, nachdem die Transaktion in Verbindung 1 abgeschlossen ist.

Zu unterschiedlichen Ergebnissen werden Sie auch kommen, wenn Sie die obige Kommandoabfolge in anderen DBMS ausprobieren. Ein wichtiger Faktor neben dem Isolation Level ist die Art und Weise, wie intern das Locking durchgeführt wird. Viele DBMS blockieren bei Updates einfach sämtliche Datensätze, die sich in den von einem Update betroffenen Speicherseiten oder Indexseiten befinden.

Wie weit parallele Transaktionen möglich sind, kann also auch davon abhängen, ob sich die Datensätze oder deren Indexeinträge physikalisch in unterschiedlichen Speicherseiten befinden oder nicht.

Die Minitabelle *t3* hat vollständig in einer Speicherseite (Page) Platz. Bestünde *t3* aber aus einer Million Datensätzen und beträfe das eine Update Datensätze mit IDs von 889480 bis 889520, die andere Datensätze mit IDs von 347120 bis 347160, dann wären die Chancen groß, dass sich die Datensätze in unterschiedlichen Seiten befinden und unabhängig voneinander geändert werden können.

Dirty Read

In den folgenden Beispielen demonstriere ich Ihnen die drei »klassischen« Lesefehler laut ANSI-Standard, die dann auftreten können, wenn die Isolierung zwischen Transaktionen nicht ausreichend gut ist. Ausgangspunkt für jedes Beispiel ist der Originalzustand von *t3*.

Von einem *Dirty Read* spricht man, wenn eine Transaktion Änderungen sieht, die in einer anderen Transaktion vorgenommen, aber noch gar nicht abgeschlossen wurden. (Es gibt also noch gar keinen `COMMIT`.) Für einen *Dirty Read* müssen Sie den schwächsten Isolation Level, *Read Uncommitted*, wählen.

```
-- Verbindung 1                  -- Verbindung 2
SET SESSION TRANSACTION          SET SESSION TRANSACTION
ISOLATION LEVEL                  ISOLATION LEVEL
READ UNCOMMITTED                 READ UNCOMMITTED

-- Zeitpunkt 1
START TRANSACTION
                                 START TRANSACTION

SELECT SUM(data) FROM t3         SELECT SUM(data) FROM t3
  3150                             3150

-- Zeitpunkt 2
UPDATE t3 SET data=0
WHERE id = 7

-- Zeitpunkt 3                   -- Dirty Read
SELECT SUM(data) FROM t3         SELECT SUM(data) FROM t3
  2150                             2150

ROLLBACK
                                 SELECT SUM(data) FROM t3
                                   3150

                                 ROLLBACK
```

Zum Zeitpunkt 1 starten ziemlich gleichzeitig zwei Transaktionen. Zum Zeitpunkt 2 ändert die erste Transaktion einen Datensatz in *t3*. Im Widerspruch zur Isolation-Forderung ist diese Änderung nicht nur in Transaktion 1 sichtbar, sondern auch in Transaktion 2!

Zum Zeitpunkt 3 widerruft die erste Transaktion die Änderung. Ab sofort ist auch in Transaktion 2 der ursprüngliche Zustand der Tabelle *t3* sichtbar.

Aus datenbanktheoretischer Sicht liegt bei einem Dirty Read eine verletzte Schreib-Lese-Abhängigkeit vor. Transaktion 1 führt eine Datenänderung durch (*Write*). Transaktion dürfte diese Daten aber erst sehen (*Read*), nachdem Transaktion 1 abgeschlossen ist. Abhilfe schafft es, dass Transaktionen Änderungen in den Daten generell erst dann sehen, wenn die Änderungen vollzogen (*committed*) sind.

Non-repeatable Read

Ein *Non-repeatable Read* findet statt, wenn innerhalb einer Transaktion dasselbe SELECT-Kommando zu unterschiedlichen Ergebnissen kommt, weil eine andere Transaktion die zugrundeliegenden Daten verändert hat. Dieses Phänomen können Sie nur in den Leveln *Read Uncommitted* oder *Read Committed* ausprobieren.

```
-- Verbindung 1                  -- Verbindung 2
SET SESSION TRANSACTION          SET SESSION TRANSACTION
ISOLATION LEVEL                  ISOLATION LEVEL
READ COMMITTED                   READ COMMITTED

-- Zeitpunkt 1
START TRANSACTION
                                 START TRANSACTION

SELECT SUM(data) FROM t3         SELECT SUM(data) FROM t3
  3150                             3150

-- Zeitpunkt 2
UPDATE t3 SET data=0
WHERE id = 7

-- Zeitpunkt 3                   -- OK
SELECT SUM(data) FROM t3         SELECT SUM(data) FROM t3
  2150                             3150

-- Zeitpunkt 4
COMMIT
                                 -- Non-repeatable Read!
                                 SELECT SUM(data) FROM t3
                                   2150

                                 COMMIT
```

Wieder starten zwei Transaktionen zum Zeitpunkt 1. Zum Zeitpunkt 2 ändert die erste Transaktion einen Datensatz von *t3*. Zum Zeitpunkt 3 ist die Änderung in der Transaktion 1 sichtbar (korrektes Verhalten), in Transaktion 2 nicht (auch korrekt).

Zum Zeitpunkt 4 bestätigt Transaktion 1 die Änderung. Ab sofort sind die geänderten Daten aber auch in Transaktion 2 sichtbar, was nicht passieren sollte. Es ist ein *Nonrepeatable Read* aufgetreten.

Phantom Read

Als *Phantom Read* wird das Phänomen bezeichnet, dass eine Transaktion Daten sieht, die eine andere Transaktion neu durch ein `INSERT`-Kommando erzeugt hat, bzw. Daten nicht mehr sieht, die eine andere Transaktion durch `DELETE` gelöscht hat. (Der zweite Fall ist selten; die meisten DBMS sind in der Lage, diese Situation zu erkennen und zu verhindern.)

Mühelos lässt sich ein Phantom Read im Level *Read Committed* auslösen:

```
-- Verbindung 1                   -- Verbindung 2
SET SESSION TRANSACTION          SET SESSION TRANSACTION
ISOLATION LEVEL                  ISOLATION LEVEL
READ COMMITTED                   READ COMMITTED

-- Zeitpunkt 1
START TRANSACTION

SELECT SUM(data) FROM t3
  3150                           -- Zeitpunkt 2
                                 START TRANSACTION

                                 INSERT INTO t3 (id, data)
                                 VALUES (11, 100)

                                 COMMIT
--- Zeitpunkt 3
SELECT SUM(data) FROM t3
  3250 -- berücksichtigt auch
       -- den 11. Datensatz
```

Im obigen Beispiel berücksichtigt `SELECT SUM(data) FROM t3` zum Zeitpunkt 1 nur zehn Datensätze, zum Zeitpunkt 2 aber elf Datensätze. Besonders problematisch ist, wenn zum Zeitpunkt 1 `SELECT SUM(data) FROM t3` ausgeführt wird, zum Zeitpunkt 2 aber `SELECT COUNT(*) FROM t3`. Bei einer nachfolgenden Berechnung des Mittelwerts würden zwei Werte verwendet, die sich auf eine unterschiedliche Datenbasis beziehen.

Bei vielen DBMS kann ein Phantom Read aber auch im Level *Repeatable Read* auftreten, was eigentlich ein Widerspruch zur Bezeichnung dieses Levels ist. Das hat mit den Locking-Verfahren zu tun: Es ist für ein DBMS möglich, alle Datensätze vor Änderungen zu schützen (also andere Transaktionen solange blockieren), die explizit in SELECT-Kommandos einer Transaktion angesprochen wurden. Das Locking umfasst aber in der Regel nicht Datensätze, die später neu hinzukommen.

Bei MySQL schützt der Level *Repeatable Read* an sich sehr gut gegen Phantom Reads. Ein Fehlverhalten lässt sich nur mit einem Trick provozieren. Das folgende Beispiel geht davon aus, dass sich *t3* wieder im Grundzustand befindet (10 Datensätze, Summe über *data* ergibt 3150).

```
-- Verbindung 1                    -- Verbindung 2
SET SESSION TRANSACTION            SET SESSION TRANSACTION
ISOLATION LEVEL                    ISOLATION LEVEL
REPEATABLE READ                    REPEATABLE READ

-- Zeitpunkt 1
START TRANSACTION

SELECT SUM(data) FROM t3
  3150
SELECT COUNT(*) FROM t3
  10

                                   -- Zeitpunkt 2
                                   START TRANSACTION

                                   INSERT INTO t3 (id, data)
                                   VALUES (11, 100)

                                   COMMIT
--- Zeitpunkt 3
SELECT SUM(data) FROM t3
  3150 -- ok

--- Zeitpunkt 4
UPDATE t3 SET data = data + 50

SELECT SUM(data) FROM t3
  3700
SELECT COUNT(*) FROM t3
  11  --

COMMIT
```

Zum Zeitpunkt 1 ermittelt die erste Transaktion Daten aus *t3*. Es gibt 10 Datensätze, die Summe über die Spalte *data* ergibt `2150`. Zum Zeitpunkt 2 startet die zweite Transaktion. Sie fügt einen Datensatz zu *t3* hinzu.

Werden die `SELECT`-Kommandos zum Zeitpunkt 3 in der ersten Transaktion wiederholt, ergeben sich dieselben Daten. Die Transaktion sieht den hinzugefügten elften Datensatz nicht. Das DBMS verhält sich korrekt.

Kritisch wird es zum Zeitpunkt 4: Das `UPDATE`-Kommando addiert bei allen Datensätzen in *t3* , bei denen *data* nicht NULL ist, `50`. Dieses `UPDATE`-Kommando erfasst auch den elften Datensatz, den es aus der Sicht von Transaktion 1 eigentlich gar nicht geben dürfte. Ab jetzt berücksichtigt `SELECT` alle elf Datensätze.

Grundsätzlich ist sowohl die berechnete Summe als auch die Anzahl der Datensätze korrekt. Allerdings liefert `SELECT COUNT(*) FROM t3` innerhalb einer Transaktion unterschiedliche Ergebnisse, was dem Anspruch des Levels *Repeatable Read* widerspricht. Transaktion 1 wurde also durch Transaktion 2 beeinflusst.

In MySQL lässt sich dieses Problem nicht einmal durch den Level *Serializable* verhindern. Selbst in diesem Level berücksichtigt das `UPDATE`-Kommando in Transaktion 1 den elften Datensatz. Abhilfe schafft dagegen das Kommando `SELECT * FROM t3 FOR UPDATE` am Beginn von Transaktion 1. Damit wird das `INSERT`-Kommando in Transaktion 2 blockiert.

Lost Update

Als Lost Update wird der Fall beschrieben, dass zwei Transaktionen, ohne voneinander zu wissen, gleichzeitig denselben Datensatz ändern. Dabei gehen Daten verloren, mit etwas Pech Geld auf einem Konto (siehe auch Abbildung 2.5 in Abschnitt 2.4, »Datensicherheit und ACID«). In MySQL ist es selbst im schwächsten Isolation Level unmöglich, in *t3* ein Lost Update zu provozieren, sofern Sie den betroffenen Datensatz direkt per `UPDATE` ändern:

```
-- Verbindung 1                     -- Verbindung 2
SET SESSION TRANSACTION             SET SESSION TRANSACTION
ISOLATION LEVEL                     ISOLATION LEVEL
READ UNCOMMITTED                    READ UNCOMMITTED

SELECT * FROM t3 WHERE id=8
  500

START TRANSACTION
                                    START TRANSACTION
UPDATE t3
SET data = data + 100
WHERE id = 8
```

```
                                  -- Transaktion wird hier
                                  -- blockiert, bis COMMIT oder
                                  -- ROLLBACK in Verbindung 2
                                  UPDATE t3
                                  SET data = data - 50
                                  WHERE id = 8
COMMIT

SELECT * FROM t3 WHERE id=8
  600
                                  COMMIT

SELECT * FROM t3 WHERE id=8       SELECT * FROM t3 WHERE id=8
  550                               550 -- korrekt
```

Anders sieht es aus, wenn Sie den zu ändernden Betrag zuerst mit `SELECT` einlesen und dann weiterverarbeiten. In diesem Beispiel verwende ich zur Demonstration eine SQL-Variable. Denkbar wäre aber auch die Verarbeitung der Daten durch ein Client-Programm. Bei einem ungünstigen Timing kann ein Lost Update sogar im relativ »sicheren« Level *Repeatable Read* passieren:

```
-- Verbindung 1                   -- Verbindung 2
SET SESSION TRANSACTION           SET SESSION TRANSACTION
ISOLATION LEVEL                   ISOLATION LEVEL
REPEATABLE READ                   REPEATABLE READ

START TRANSACTION
                                  START TRANSACTION
-- @var1 = 500
SELECT data FROM t3
WHERE id=8 INTO @var1             -- @var2 = 500
                                  SELECT data FROM t3
                                  WHERE id=8 INTO @var2
UPDATE t3
SET data = @var1 + 100
WHERE id=8
                                  -- Transaktion wird hier
                                  -- blockiert, bis COMMIT oder
                                  -- ROLLBACK in Verbindung 1
                                  UPDATE t3
                                  SET data = @var2 - 50
                                  WHERE id=8
COMMIT
                                  -- jetzt wird das UPDATE
                                  --- ausgeführt und bestätigt
                                  COMMIT
```

```
SELECT data FROM t3                SELECT * FROM t3
WHERE id=8                         WHERE id=8
  450                                450 -- falsch!
```

MySQL-spezifischer Umgang mit Variablen

Die Verwendung von Variablen in der Form `@var` sowie Variablenzuweisungen in der Form `SELECT ... INTO @var` funktioniert in dieser Form nur in MySQL bzw. MariaDB.

Erst der Level *Serializable* verhindert diesen Fehler. Das liegt daran, dass der betroffene Datensatz allein durch das Ausführen von `SELECT` gesperrt wird. Die erste Verbindung wird beim `UPDATE`-Kommando blockiert. Bei `UPDATE` in der zweiten Transaktion erkennt MySQL einen Deadlock. Transaktion 1 würde für immer auf das Ende von Transaktion 2 warten, während Transaktion 2 auf das Ende von Transaktion 1 wartet. Weil es aus dieser Situation keinen Ausweg gibt, bricht MySQL Transaktion 2 ab. Damit kann Transaktion 1 fortgesetzt und mit `COMMIT` abgeschlossen werden.

```
-- Verbindung 1                    -- Verbindung 2
SET SESSION TRANSACTION            SET SESSION TRANSACTION
ISOLATION LEVEL                    ISOLATION LEVEL
SERIALIZABLE                       SERIALIZABLE

START TRANSACTION
                                   START TRANSACTION

-- @var1 = 500
SELECT data FROM t3
WHERE id=8 INTO @var1              -- @var2 = 500
                                   SELECT data FROM t3
                                   WHERE id=8 INTO @var2

-- wird blockiert, wartet
-- auf Verbindung 2
UPDATE t3
SET data = @var1 + 100
WHERE id=8
                                   -- MySQL erkennt Deadlock und
                                   -- bricht diese Transaktion ab
                                   -- (entspricht ROLLBACK)
                                   UPDATE t3
                                   SET data = @var2 - 50
                                   WHERE id=8

-- UPDATE wird ausgeführt
-- und bestätigt
COMMIT
```

Anstelle den Level *Serializable* zu aktivieren, können Sie auch `SELECT ... FOR UPDATE` ausführen. Auch so verhindern Sie das Lost Update:

```
-- Verbindung 1                    -- Verbindung 2
SET SESSION TRANSACTION            SET SESSION TRANSACTION
ISOLATION LEVEL                    ISOLATION LEVEL
REPEATABLE READ                    REPEATABLE READ

START TRANSACTION
                                   START TRANSACTION
-- @var1 = 500
SELECT data FROM t3
WHERE id=8 INTO @var1              -- MySQL blockiert die
FOR UPDATE                         -- Ausführung
                                   SELECT data FROM t3
                                   WHERE id=8 INTO @var2
                                   FOR UPDATE
UPDATE t3
SET data = @var1 + 100
WHERE id=8

COMMIT                             -- erst jetzt wird SELECT
                                   -- ausgeführt, @var2 = 600
                                   UPDATE t3
                                   SET data = @var2 - 50
                                   WHERE id=8

                                   COMMIT

SELECT data FROM t3                SELECT data FROM t3
WHERE id=8                         WHERE id=8
  550                                550 -- korrekt!
```

Ganz allgemein gilt: Lost Updates treten auf, wenn zwei Transaktionen gleichzeitig gemeinsame Daten ändern. Abhilfe schaffen exklusive Schreib-Locks auf die betroffenen Datensätze.

12.4 Locking-Verfahren

Jetzt bleibt noch die Frage offen, *wie* ein DBMS die verschiedenen Isolation Level garantieren kann. Grundsätzlich ist das ein Implementierungsdetail, wenn auch zugegebenermaßen ein sehr wichtiges. Verschiedene DBMS differenzieren sich durch Locking-Mechanismen von konkurrierenden Systemen. Damit kann jeder Hersteller