

Database system concepts

by Abraham Silberschatz, Henry F. Korth, S. Sudarshan · Language: DE

Summary by booksummaryai.com —

<https://booksummaryai.com/country/de/database-system-concepts-zusammenfassung>

Summary

In 60 Sekunden

Dies ist der Standardtext an den Universitäten, wenn es darum geht, wie Datenbanken wirklich funktionieren, aufgebaut um eine einzige Universitätsdatenbank mit Dozenten, Studierenden, Kursen und Fachbereichen. Es beginnt damit, warum Datenbanken Dateien schlagen, lehrt das relationale Modell und SQL von einfachen Abfragen bis zu Triggern und Rekursion und zeigt dann, wie man mit E-R-Diagrammen und Normalisierung gute Schemata entwirft. Die zweite Hälfte blickt unter die Haube: Speicherung, Indizes, Abfrageoptimierung, Transaktionen mit ACID-Garantien, Nebenläufigkeitskontrolle, Recovery und verteilte Systeme. Am Ende kannst du einen Abfrageplan lesen und weißt, warum deine Banküberweisung nie zur Hälfte passiert.

Warum es sich lohnt

Fast jede App, die du benutzt, sitzt auf einer Datenbank, und dies ist das Buch, das die Menschen ausgebildet hat, die sie gebaut haben. Es ist wichtig für dich, wenn du ein Informatikstudent vor Prüfungen bist, ein Entwickler, der SQL schreibt, aber nie die Theorie gelernt hat, oder ein Ingenieur, der sich auf Systemdesign-Interviews vorbereitet. Du bekommst Vokabular und mentale Modelle – Schlüssel, ACID, Serialisierbarkeit –, die das Verhalten in Produktion vorhersehbar statt geheimnisvoll machen.

Die Kernideen

Dateien verrotten, Datenbanken nicht: das Plädoyer für ein DBMS

Eine Datenbank ist eine Sammlung zusammenhängender Daten, und das DBMS ist die Gesamtheit der Programme, die darauf zugreifen. Das Buch stellt dem Dateiverarbeitungssysteme gegenüber, die unter Redundanz, Inkonsistenz, schwierigem Datenzugriff, Datenisolation und Integritätsproblemen leiden. Ein DBMS bietet stattdessen Datenunabhängigkeit, effizienten Datenzugriff, Datenintegrität und Sicherheit, gleichzeitigen Zugriff sowie Wiederherstellung nach Abstürzen. Datenabstraktion auf physischer, logischer und Sichtebeine hält den Anwendungscode stabil, selbst wenn sich die Speicherung ändert. Datenbankbenutzer werden eingeteilt in naive Benutzer, Anwendungsprogrammierer und erfahrene Benutzer, dazu die Datenbankadministratoren (DBAs), die das ganze Spiel leiten.

Eine Datenbank ist eine Sammlung zusammenhängender Daten, und das DBMS ist die Gesamtheit der Programme, die darauf zugreifen. Die Eingangsargumentation des Buches

lautet: Dateiverarbeitungssysteme scheitern auf vorhersehbare Weise – Redundanz, Inkonsistenz, mühsam formulierte Abfragen, Isolation, Integritätsprobleme –, während dir ein DBMS Datenunabhängigkeit, effizienten Zugriff, Integrität, Sicherheit, Nebenläufigkeit und Wiederherstellung nach Abstürzen bietet. Datenabstraktion auf physischer, logischer und Sichtebeine hält den Anwendungscode stabil, selbst wenn sich die Speicherung ändert. Die Benutzer teilen sich auf in naive Benutzer, Anwendungsprogrammierer und erfahrene Benutzer, dazu den DBA, der das ganze Spiel leitet.

Probier es aus: Such dir in deinem Leben eine Tabellenkalkulation, in der zwei Kopien derselben Daten auseinanderlaufen, und überführe sie in eine einzige Tabelle mit einem Primärschlüssel.

Überall nur Tabellen: das relationale Modell

Das ganze Buch dreht sich um eine kleine Tabelle: die Relation instructor mit den Attributen ID, name, dept_name und salary, wobei ID der Primärschlüssel ist. Zeilen sind Tupel, Spalten sind Attribute, die Reihenfolge der Zeilen bedeutet nichts, und jeder Wert ist atomar – keine Listen, die in eine Zelle gestopft werden. Dazu die Bedingung, die alles vertrauenswürdig macht: referentielle Integrität, veranschaulicht daran, dass instructor.dept_name auf department.dept_name verweist, sodass du keinen Dozenten einer Abteilung zuweisen kannst, die es nicht gibt. Auf diesem schlichten Tabellenfundament baut das Buch die relationale Algebra auf – select, project, union, set difference, Cartesian product, rename –, wobei der Join aus diesen Grundelementen definiert wird. Jede Abfrage, die du je schreiben wirst, ist eine Komposition dieser sechs Grundzüge.

Eine Relation ist einfach eine Tabelle: Zeilen sind Tupel, Spalten sind Attribute, die Tupelreihenfolge ist egal, und Attributwerte sind atomar. Die Relation instructor – ID, name, dept_name, salary, mit ID als Primärschlüssel – ist das durchgehende Beispiel für das gesamte Buch. Schlüssel kommen in einer Familie daher: Superschlüssel, Kandidatenschlüssel, Primärschlüssel, Fremdschlüssel, und sie erzwingen Integrität, wobei referentielle Integrität bedeutet, dass instructor.dept_name zu einem echten department.dept_name passen muss. Die sechs Grundoperationen der relationalen Algebra – select, project, union, set difference, Cartesian product, rename – definieren alles andere, einschließlich des Joins.

Probier es aus: Öffne eine beliebige Tabelle, die dir gehört, und prüfe: Enthält jede Spalte genau einen atomaren Wert, und gibt es eine Spalte, die jede Zeile eindeutig identifiziert?

SELECT-FROM-WHERE: SQL von den ersten Worten bis zu Unterabfragen

Die grundlegende SQL-Abfragestruktur ist SELECT-FROM-WHERE: Die SELECT-Klausel wählt Spalten aus, FROM nennt die Relationen, WHERE filtert Tupel mit Prädikaten – zum Beispiel das Auffinden von Dozentennamen nach Abteilung. Die SQL-Datendefinition verwendet CREATE TABLE, um Attribute, Datentypen (wie char, varchar, int, numeric) und Integritätsbedingungen wie PRIMARY KEY, FOREIGN KEY und NOT NULL festzulegen. Dann vertieft sich die Sprache: Aggregatfunktionen (avg, min, max, sum, count) mit group by und having, Nullwerte,

Mengenoperationen (union, intersect, except), verschachtelte Unterabfragen (in, some, all, exists), Join-Ausdrücke (natural, inner, outer), Sichten, die with-Klausel, grant und revoke, Trigger sowie rekursive Abfragen mit with recursive.

SQLs grundlegende Abfragestruktur ist SELECT-FROM-WHERE: Die SELECT-Klausel wählt Spalten aus, FROM nennt die Relationen, WHERE filtert Tupel mit Prädikaten. Von dort aus schichtet das Buch auf: Aggregate (avg, min, max, sum, count) mit group by und having, Nullwerte, Mengenoperationen (union, intersect, except) sowie verschachtelte Unterabfragen mit in, some, all, exists. Das SQL der mittleren Stufe ergänzt Join-Varianten – natural, inner, left/full outer –, Sichten über CREATE VIEW, die with-Klausel für temporäre benannte Unterabfragen, Transaktionen, Integritätsbedingungen bis hin zu CREATE ASSERTION und die Autorisierung über grant. Fortgeschrittenes SQL bringt gespeicherte Funktionen und Prozeduren, Trigger, die vor oder nach Zeilen- oder Anweisungsereignissen auslösen, und rekursive Abfragen mit with recursive.

Probier es aus: Schreibe diese Woche eine Abfrage mit group by und having – bilde irgendetwas pro Gruppe als Durchschnitt und filtere dann die Gruppen.

Papier vor Code: das E-R-Modell

Das E-R-Modell arbeitet mit Entitäten, Attributen, Beziehungen, Entitätsmengen und Beziehungsmengen, um Datenbanken zu entwerfen, bevor man sich auf Tabellen festlegt – mit der Universitätsdatenbank (instructor, student, course, department) als durchgängigem Beispiel. Der Kern sind die Constraints: 'takes' ist eine n:m-Beziehung zwischen Studierenden und Kursen. Schwache Entitätsmengen wie 'section', die von 'course' abhängen, werden über ein Diskriminator-Attribut (sec_id) identifiziert und haben einen Primärschlüssel (course_id, sec_id, semester, year). Zu den Constraints im E-R-Modell gehören Kardinalitäten (1:1, 1:n, n:1, n:m), Partizipationsconstraints und Schlüssel. Spezialisierung, Generalisierung und Aggregation behandeln Hierarchien und Beziehungen zwischen Beziehungen.

Das E-R-Modell lässt dich mit Entitäten, Attributen und Beziehungen entwerfen, bevor du dich auf Tabellen festlegst – mit der Universitätsdatenbank (instructor, student, course, department) als durchgängigem Beispiel. Der Kern sind die Constraints: Kardinalitäten (1:1, 1:n, n:m), Partizipationsconstraints und Schlüssel, wobei takes eine n:m-Beziehung zwischen Studierenden und Kursen ist. Schwache Entitätsmengen wie section hängen von course ab und tragen einen Diskriminator (sec_id) sowie einen zusammengesetzten Primärschlüssel (course_id, sec_id, semester, year). Spezialisierung, Generalisierung und Aggregation behandeln Hierarchien und Beziehungen zwischen Beziehungen.

Probier es aus: Skizziere die Daten deines nächsten Projekts zuerst als Entitäten und Beziehungen und markiere jede Beziehung als 1:1, 1:n oder n:m, bevor du auch nur eine Zeile DDL schreibst.

Zerleg die Tabelle, bis es wehtut: Normalisierung

Funktionale Abhängigkeiten (etwa ID name, dept_name, salary in der Relation instructor) dienen dazu, Relationen zu zerlegen und durch Normalisierung Redundanz zu vermeiden. Die Boyce-Codd-Normalform verlangt, dass bei jeder nichttrivialen funktionalen Abhängigkeit $X \rightarrow Y$ die linke Seite X ein Superschlüssel ist; die dritte Normalform lockert das auf und erlaubt etwas Redundanz, damit Abhängigkeiten durchsetzbar bleiben. Dekompositionsalgorithmen, die auf der kanonischen Überdeckung basieren, überführen nicht-BCNF-Schemas in BCNF-Relationen – wobei die BCNF-Zerlegung Abhängigkeiten verlieren kann, während die 3NF-Zerlegung sie erhält. Entweder wird zur begründeten Entscheidung statt zur Gewohnheit.

Funktionale Abhängigkeiten – etwa dass ID in der Relation instructor name, dept_name und salary bestimmt – sind das formale Werkzeug, um Redundanz aufzuspüren und die Zerlegung zu steuern. BCNF verlangt, dass bei jeder nichttrivialen Abhängigkeit $X \rightarrow Y$ die linke Seite X ein Superschlüssel ist; 3NF lockert das bewusst auf und toleriert etwas Redundanz, damit Abhängigkeiten innerhalb einzelner Relationen durchsetzbar bleiben. Dekompositionsalgorithmen, die auf der kanonischen Überdeckung aufbauen, transformieren nicht-BCNF-Schemas, und der Trade-off ist real: Die BCNF-Zerlegung erhält Abhängigkeiten womöglich nicht, während die 3NF-Zerlegung sie erhält. Entweder wird zur begründeten Entscheidung statt zur Gewohnheit.

Probier es aus: Nimm deine breiteste Tabelle und liste auf, was was bestimmt; wenn eine Nicht-Schlüssel-Spalte eine andere Nicht-Schlüssel-Spalte bestimmt, hast du einen Kandidaten für die Zerlegung gefunden.

Wo die Bytes wohnen: Festplatten und RAID

Der Blick zoomt von der Logik zur Physik: Die Speicherhierarchie umfasst Cache, Hauptspeicher, Flash-Speicher, Magnetplatte, optische Platte und Magnetband. Magnetplatten sind physische Objekte mit Plattern, Spuren und Sektoren – deshalb wird die Zugriffszeit von mechanischer Bewegung dominiert. Und die Frage, vor der jede Administratorin steht – was passiert, wenn eine Platte stirbt? – beantwortet RAID: Die Level 0 bis 6, einschließlich Kombinationen wie RAID 1+0, liefern Redundanz über mehrere Platten hinweg. Jede spätere Entscheidung über Indizierung und Pufferung im Buch wird an dieser Hierarchie bemessen.

Die Speicherhierarchie reicht von Cache und Hauptspeicher über Flash-Speicher, Magnetplatte, optische Platte und Magnetband, wobei Volatilität und Kosten pro Byte bestimmen, wohin welche Daten gehören. Magnetplatten sind physische Objekte – Plattern, Spuren, Sektoren –, deshalb wird die Zugriffszeit von mechanischer Bewegung dominiert. RAID-Level 0 bis 6, einschließlich Kombinationen wie RAID 1+0, liefern Redundanz über mehrere Platten hinweg und machen aus unabhängigen Ausfallraten ein beherrschbares Risiko. Jede spätere Entscheidung über Indizierung und Pufferung im Buch wird an dieser Hierarchie bemessen.

Probier es aus: Prüfe, welches RAID-Level dein Produktiv- oder Backup-Speicher tatsächlich nutzt, und ob er einen Ausfall von zwei Platten übersteht.

B-Bäume und Buckets: Daten schnell finden

Das Buch baut zuerst geordnete Indizes auf – primäre und sekundäre, dichte und spärliche, zu mehrstufigen Indizes gestapelt – und dann seinen Star: den B+-Baum, eine balancierte Baumstruktur, bei der alle Blattknoten auf derselben Tiefe liegen und die effiziente Bereichsanfragen und Gleichheitssuchen unterstützt. Hashing schlägt den anderen Weg ein: Statisches Hashing verwendet eine feste Anzahl von Buckets und leidet, wenn die Daten wachsen, während dynamische Verfahren wie erweiterbares Hashing die Hashtabelle dynamisch wachsen und schrumpfen lassen, mit linearem Hashing als Alternative. Die Wahl zwischen beiden ist die Wahl zwischen Bereichsscans und Punktabfragen.

Geordnete Indizes gibt es in verschiedenen Spielarten – primär und sekundär, dicht und spärlich, mehrstufig – und der B+-Baum ist das Arbeitstier: ein balancierter Baum, bei dem alle Blattknoten auf derselben Tiefe liegen, und genau deshalb eignet er sich sowohl für Gleichheitssuchen als auch für Bereichsanfragen effizient. Hashing tauscht Ordnung gegen Geschwindigkeit: Statisches Hashing verwendet eine feste Anzahl von Buckets, während dynamische Verfahren wie erweiterbares Hashing die Hashtabelle dynamisch wachsen und schrumpfen lassen, wenn sich die Daten ändern, mit linearem Hashing als Alternative. Die Wahl zwischen beiden ist die Wahl zwischen Bereichsscans und Punktabfragen.

Probier es aus: Nimm deine langsamste Anfrage und prüfe, ob ihre WHERE-Spalten einen Index haben; falls nicht, füge einen hinzu und miss den Unterschied.

Wie eine Anfrage tatsächlich läuft: Verarbeitung und Optimierung

Die Anfrageverarbeitung umfasst Parsen und Übersetzung, Optimierung und Auswertung. Algorithmen für Selektion, Sortierung und Join-Operationen – Nested-Loop, Block-Nested-Loop, indizierter Nested-Loop, Merge-Join, Hash-Join – werden anhand der Datenträger-E/A-Kosten gemessen, und die falsche Wahl kann Größenordnungen kosten. Der Trick des Optimierers sind Äquivalenzregeln, die einen relationalen Algebra-Ausdruck in einen anderen umformen, plus kostenbasierte Optimierung, die die Kosten verschiedener Auswertungspläne anhand von Statistiken wie Histogrammen und Selektivität schätzt – sie sagt den Preis jedes Plans voraus, bevor auch nur einer läuft.

Die Anfrageverarbeitung hat drei Phasen: Parsen und Übersetzung, Optimierung und Auswertung, wobei Algorithmen für Selektion, Sortierung und Joins – Nested-Loop, Block-Nested-Loop, indizierter Nested-Loop, Merge-Join, Hash-Join – an den Datenträger-E/A-Kosten gemessen werden. Die Optimierung ist der interessante Teil: Äquivalenzregeln erlauben dem System, einen relationalen Algebra-Ausdruck in eine billigere Gestalt umzuschreiben, und die kostenbasierte Optimierung schätzt die Plankosten aus Statistiken wie Histogrammen und Selektivität. Dasselbe SQL kann je nachdem, welchen Plan der Optimierer wählt, um Größenordnungen in der Laufzeit auseinanderliegen.

Probier es aus: Führe diese Woche EXPLAIN (oder das Äquivalent deiner Datenbank) auf einer langsamen Anfrage aus und finde heraus, welche Join-Methode sie gewählt hat.

ACID: Warum eine fehlgeschlagene Überweisung dein Geld nicht verlieren kann

Das kanonische Beispiel ist eine Banküberweisung von Konto A auf Konto B. Eine Transaktion ist eine Ausführungseinheit eines Programms, die auf verschiedene Datenelemente zugreift und sie möglicherweise aktualisiert, und sie muss die ACID-Eigenschaften erfüllen: Atomarität (entweder passieren alle Schritte oder keiner), Konsistenz (die Bücher bleiben im Gleichgewicht), Isolation (keine andere Transaktion sieht den halbfertigen Zustand) und Dauerhaftigkeit (einmal bestätigt, überlebt die Überweisung jeden späteren Ausfall). Die Banküberweisung von Konto A auf Konto B ist das Beispiel des Buches, und es ist dieselbe Garantie, die deine Bank dir täglich gibt.

Eine Transaktion ist eine Ausführungseinheit eines Programms, die auf verschiedene Datenelemente zugreift und sie möglicherweise aktualisiert, und sie muss ACID erfüllen: Atomarität, Konsistenz, Isolation, Dauerhaftigkeit. Das kanonische Beispiel ist eine Banküberweisung von Konto A auf Konto B – zwei Aktualisierungen, die zusammen gelingen oder scheitern müssen, die Bücher nie inkonsistent hinterlassen, anderen nie mitten im Vorgang sichtbar werden und nach einem Absturz nie einfach verpuffen. Jedes Nebenläufigkeitsprotokoll und jeder Wiederherstellungsmechanismus in den späteren Kapiteln existiert, um diese vier Eigenschaften mit Geschwindigkeit zu liefern.

Probier es aus: Packe deine nächste mehrstufige Datenänderung in eine einzige Transaktion – BEGIN, beide Aktualisierungen, COMMIT – und teste, dass ein Fehler beide zurückrollt.

Sperren, Zeitstempel und der Deadlock-Tanz

Die Nebenläufigkeitskontrolle behandelt Serialisierbarkeit (Konflikt- und Sicht-Serialisierbarkeit) als Korrektheitsmaßstab für die gleichzeitige Ausführung. Sperrbasierte Protokolle wie das Zwei-Phasen-Sperrprotokoll (2PL) mit geteilten und exklusiven Sperren gewährleisten Serialisierbarkeit, doch das Halten von Sperren kann zu Deadlocks führen. Deshalb lehrt das Buch den Umgang damit: Präventionsschemata (wait-die, wound-wait), Erkennung mittels Wartegraphen (ein Zyklus zeigt einen Deadlock an) und Auflösung typischerweise durch Zurückrollen einer der Transaktionen. Jenseits von Sperren weisen zeitstempelbasierte Protokolle jeder Transaktion einen eindeutigen Zeitstempel zu, und auch validierungsbasierte (optimistische) Protokolle und Multiversion Concurrency Control (MVCC) werden behandelt.

Serialisierbarkeit—Konflikt und Sicht—ist der Korrektheitsmaßstab für die gleichzeitige Ausführung, und das Zwei-Phasen-Sperrprotokoll erreicht sie mit geteilten und exklusiven Sperren, wobei das Halten von Sperren in wachsender und schrumpfender Phase Deadlocks erzeugen kann. Die Deadlock-Behandlung zerfällt in Prävention (wait-die, wound-wait), Erkennung über Wartegraphen, bei denen ein Zyklus einen Deadlock anzeigt, und Auflösung durch Zurückrollen einer Transaktion. Alternativen tauschen Sperren gegen andere Annahmen ein: Zeitstempelbasierte Protokolle geben jeder Transaktion einen eindeutigen Zeitstempel und

ordnen die Arbeit entsprechend an, während validierungsbasierte (optimistische) Protokolle und Multiversion Concurrency Control die Leser ohne Sperren halten.

Probier es aus: Prüfe in dieser Woche die aktuellen Sperrwartezeiten oder Deadlocks deiner Datenbank—die meisten Systeme legen sie offen—and identifiziere, welche zwei Abfragen kollidiert sind.

Das Protokoll, das alles überlebt: Recovery

Die protokollbasierte Recovery zeichnet jede Änderung in einem Protokoll auf stabilen Speicher auf. Die Regel, die es vertrauenswürdig macht, ist Write-Ahead-Logging (WAL): Protokollsätze müssen auf stabilen Speicher geschrieben werden, bevor die zugehörige Datenseite geschrieben wird. Deferred und immediate database modification bezeichnen die beiden Strategien, und Checkpoints begrenzen, wie viel Protokoll wiedergegeben werden muss. Der ARIES-Recovery-Algorithmus verwendet eine Log Sequence Number (LSN) und drei Phasen: Analyse (identifiziert schmutzige Seiten und aktive Transaktionen), Redo (wendet alle Änderungen aus dem Protokoll erneut an) und Undo (rollt nicht festgeschriebene Transaktionen mit Kompensationsprotokollsätzen zurück). Crash-Recovery hört auf, Magie zu sein, und wird zu einem Algorithmus, den du implementieren könntest.

Die protokollbasierte Recovery zeichnet jede Änderung in einem Protokoll auf stabilen Speicher auf, mit deferred und immediate database modification als den beiden Strategien, Checkpoints, die begrenzen, wie viel Protokoll wiedergegeben werden muss, und der Write-Ahead-Logging-Regel als Invariante: Protokollsätze erreichen den stabilen Speicher, bevor die zugehörige Datenseite geschrieben wird. ARIES macht aus dem Protokoll ein präzises Recovery-Verfahren mit Log Sequence Numbers und drei Phasen—Analyse, um schmutzige Seiten und aktive Transaktionen zu finden, Redo, um alle Änderungen erneut anzuwenden, Undo, um nicht festgeschriebene Transaktionen mit Kompensationsprotokollsätzen zurückzurollen. Crash-Recovery hört auf, Magie zu sein, und wird zu einem Algorithmus, den du implementieren könntest.

Probier es aus: Vergewissere dich, dass deine Datenbank Full-Page-Logging und aktuelle Checkpoints aktiviert hat—diese beiden Einstellungen entscheiden, wie schnell du dich erholst.

Eine Maschine reicht nie: Architekturen, Parallelität, Verteilung

Zu den Datenbanksystem-Architekturen gehören zentralisierte und Client-Server-Systeme, mit der Unterscheidung zwischen Anwendungsservern und Datenbankservern sowie dem Konzept einer Drei-Schichten-Architektur. Für reinen Durchsatz umfassen parallele Datenbankarchitekturen Shared Memory, Shared Disk und Shared Nothing, gemessen an Speedup und Scaleup, mit Ausnutzung von Inter- und Intraquery-Parallelität. Wenn sich Daten über mehrere Standorte erstrecken, nutzen verteilte Datenbanken Datenreplikation, Fragmentierung (horizontal und vertikal) und Partitionierung (Round-Robin, Hash, Range); das Two-Phase-Commit-Protokoll (2PC) gewährleistet Atomizität, indem ein Koordinator alle

Teilnehmer auffordert, sich vor dem Festschreiben vorzubereiten, sodass eine verteilte Transaktion überall oder nirgends atomar ist.

Datenbanksystem-Architekturen reichen von zentralisiert bis Client-Server, unterscheiden Anwendungsserver von Datenbankservern und fügen das Drei-Schichten-Modell hinzu. Parallele Datenbanken wählen zwischen Shared-Memory-, Shared-Disk- und Shared-Nothing-Entwürfen, gemessen an Speedup und Scaleup, mit Ausnutzung von Inter- und Intraquery-Parallelität. Verteilte Datenbanken verteilen Daten über Standorte mittels Replikation, horizontaler und vertikaler Fragmentierung und Partitionierung nach Round-Robin, Hash oder Range—während das Two-Phase-Commit-Protokoll, bei dem ein Koordinator alle Teilnehmer auffordert, sich vor dem Festschreiben vorzubereiten, dafür sorgt, dass eine Transaktion über Maschinen hinweg atomar bleibt.

Probier es aus: Skizziere, welche Schicht deines aktuellen Systems die Geschäftslogik enthält und welche die Daten—wenn sie verschmolzen sind, hast du deinen Skalierungsengpass gefunden.

Die besten Zitate

- "Eine Datenbank ist eine Sammlung zusammenhängender Daten, und ein Datenbankmanagementsystem (DBMS) ist eine Menge von Programmen zum Zugriff auf diese Daten." - Abraham Silberschatz, Henry F. Korth, S. Sudarshan
- "instructor"-Relation mit den Attributen ID, name, dept_name und salary, wobei ID der Primärschlüssel ist" - Abraham Silberschatz, Henry F. Korth, S. Sudarshan
- "with recursive"-Konstrukt, mit einem Beispiel, das alle Kurse findet, die (direkt oder indirekt) Voraussetzungen eines gegebenen Kurses sind" - Abraham Silberschatz, Henry F. Korth, S. Sudarshan
- "Write-Ahead-Logging"-Regel (WAL), die verlangt, dass Protokollsätze vor der entsprechenden Datenseite auf stabilen Speicher geschrieben werden" - Abraham Silberschatz, Henry F. Korth, S. Sudarshan

Kritik

Es ist ein Lehrbuch, und es liest sich auch so: geduldig, systematisch und streckenweise trocken, wobei die Übungen einen Großteil der Lehre übernehmen. Das durchlaufende Universitätsbeispiel ist hilfreich, kann sich aber über fünfzehnhundert Seiten Abdeckung hinweg repetitiv anfühlen. Die Implementierungsdetails driften zu den Systemen, die zum Zeitpunkt des Schreibens aktuell waren, weshalb manche Details auf Produktebene schneller altern als die Theorie. Und das Buch lehrt Prinzipien mehr als operatives Handwerk – Tuning, echte Kapazitätsplanung und Cloud-native Muster kommen knapp weg. Kombiniere es mit praktischer Übung, sonst weißt du, was ein B+-Baum ist, ohne jemals einen abgestimmt zu haben.

In einem Satz

Jede zuverlässige Anwendung ruht auf Tabellen, Schlüsseln, Transaktionen und Logs – und dieses Buch bringt dir alle vier von Grund auf bei.

Für wen sich das Buch eignet

Alle, die SQL schreiben und ahnen, dass die Datenbank darunter etwas Magisches treibt – Anwendungsentwickler, Data Engineers und Informatikstudierende, die wissen wollen, warum ihre Abfrage langsam ist und warum ihre Banküberweisung nie zur Hälfte passiert. Auch Manager, die in Architekturbesprechungen mitnicken, während sie sich insgeheim wünschen, jemand würde ihnen Indizes, Transaktionen und Normalisierung in klaren Worten erklären.