The Pragmatic Programmers

Deutsche Übersetzung von O'REILLY®

Sieben Wochen, sieben Datenbanken

Moderne Datenbanken und die NoSQL-Bewegung

Eric Redmond
& Jim R. Wilson

Übersetzt von Peter Klicman

Sieben Wochen, sieben Datenbanken

Moderne Datenbanken
und die NoSQL-Bewegung

Eric Redmond & Jim R. Wilson

Deutsche Übersetzung von
Peter Klicman

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag GmbH & Co. KG
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

Die Originalausgabe erschien 2012 unter dem Titel
Seven Databases in Seven Weeks
bei Pragmatic Bookshelf, Inc.

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Übersetzung: Thomas Demmig, Mannheim
Lektorat: Volker Bombien, Köln
Korrektorat: Dr. Dorothée Leidig, Freiburg
Produktion: Karin Driesen, Köln

Satz: le-tex publishing services GmbH, Leipzig, www.le-tex.de
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-86899-791-0

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhaltsverzeichnis

Widmung

Bei der Abfahrt am Beaver Run SuperChair in Breckenridge, Colorado, fragten wir uns, wo der frische Pulverschnee geblieben war. In Breckenridge gab es Schneekanonen und die Pisten waren hervorragend präpariert, doch zwangsläufig waren die Bedingungen auf dem Berg überall gleich. Ohne frischen Pulverschnee fehlte der letzte Kick.

Im Jahr 1994 hatte ich als Angestellter im Datenbank-Entwicklungslabor von IBM in Austin das gleiche Gefühl. Ich hatte an der University of Texas in Austin objektorientierte Datenbanken studiert, da ich nach einer Dekade relationaler Dominanz eine reale Chance dafür sah, dass objektorientierte Datenbanken Fuß fassen könnten. Und doch brachte die nächste Dekade mehr dieser immer gleichen relationalen Modelle denn je zuvor. Ich schaute deprimiert zu, wie Oracle, IBM und später durch MySQL angeführte Open-Source-Lösungen sich ausbreiteten und alle anderen aufkommenden Lösungen im Keim erstickten.

Mit der Zeit entwickelten sich die Benutzerschnittstellen von grünen Bildschirmen über Client/Server- zu Internet-basierten Anwendungen, doch die Programmierung der relationalen Schicht weitete sich zur erbarmungslosen Mauer der Gleichförmigkeit aus, die Jahrzehnte perfekter Langeweile mit sich brachte. Also warteten wir auf eine neue Schneedecke.

Und schließlich fiel der frische Pulverschnee. Zuerst reichte er nicht einmal aus, um die ersten Spuren zu bedecken, doch der Sturm kam, frischte die Landschaft auf und sorgte für das perfekte Ski-Erlebnis mit der Vielfalt und Qualität, die wir uns erwünscht hatten. Erst im letzten Jahr wachte ich auf und stellte fest, dass auch die Datenbank-Welt mit einer frischen Schneedecke überzogen wurde. Sicher, die relationalen Datenbanken sind immer noch da und man kann überraschend viel Spaß mit Open-Source-RDBMS' haben. Sie können mit Clustern arbeiten, Volltext- und sogar Fuzzy-Suchen durchführen. Doch Sie sind nicht länger auf diesen Ansatz beschränkt. Ich habe seit einem Jahr keine vollständig relationale Lösung mehr entwickelt.

Vielmehr habe ich in dieser Zeit eine dokumentenbasierte Datenbank und eine Reihe von Schlüssel/Wert-Datenspeichern genutzt.

Die Wahrheit ist, dass relationale Datenbanken nicht länger ein Monopol auf Flexibilität oder gar Skalierbarkeit besitzen. Für alle Arten der von uns entwickelten Anwendungen gibt es bessere Modelle, die einfacher, schneller und zuverlässiger sind. Als jemand, der zehn Jahre bei IBM Austin an Datenbanken im Labor und beim Kunden gearbeitet hat, ist die Entwicklung einfach überwältigend. In *Sieben Wochen, sieben Datenbanken* werden Sie sich durch Beispiele arbeiten, die eine sehr schöne Schnittmenge der wichtigsten Fortschritte bei den Datenbanken behandeln, die die Internet-Entwicklung stützen. Bei den Schlüssel/Wert-Speichern werde Sie das radikal skalierbare und zuverlässige Riak kennenlernen sowie die schönen Query-Mechanismen von Redis. Aus dem Bereich der spaltenorientierten Datenbanken werden Sie die Leistungsfähigkeit von HBase antesten, einem engen Verwandten des relationalen Datenbankmodells. Und bei den dokumentenbasierten Datenbanken werden Sie die elegante Lösung für tief verschachtelte Dokumente beim stark skalierbaren MongoDB kennenlernen. Sie werde mit Neo4J auch Graph-Datenbanken kennenlernen, die eine schnelle Verarbeitung von Beziehungen erlauben.

Um ein besserer Programmierer oder Datenbankadministrator zu werden, müssen Sie all diese Datenbanken nicht nutzen. Während Eric Redmond und Jim Wilson Sie auf diese magische Reise mitnehmen, werden Sie bei jedem Schritt schlauer und erhalten Einblicke, die für den modernen Softwareentwickler unbezahlbar sind. Sie werden wissen, wo die jeweilige Plattform besonders glänzt und wo sie am beschränktesten ist. Sie werden sehen, wo sich Ihre Branche hinbewegt, und werden die Kräfte kennenlernen, die sie dorthin treiben.

Genießen Sie die Reise.

Bruce Tate
Austin, Texas, Mai 2012

Danksagungen

Ein Buch dieser Größe und zu diesem Thema kann nicht von nur zwei Autoren geschrieben werden. Es verlangt den Einsatz vieler sehr kluger Leute mit übermenschlichen Augen, die so viele Fehler wie möglich erkennen und wichtige Einblicke in die Details der jeweiligen Techniken vermitteln.

Wir wollen, ohne besondere Reihenfolge, den folgenden Leuten Danken, die ihre Zeit und ihr Wissen mit uns geteilt haben:

Ian Dees	Mark Phillips	Jan Lenhardt
Robert Stam	Oleg Bartunov	Dave Purrington
Daniel Bretoi	Matt Adams	Sean Copenhaver
Loren Sands-Ramshaw	Emil Eifrem	Andreas Kollegger

Wir wollen auch Bruce Tate für seine Erfahrung und Beratung danken.

Wir möchten uns beim gesamten Team von Pragmatic bedanken. Danke für die Unterstützung dieses kühnen Projekts. Wir danken insbesondere unserem Lektor Jackie Carter. Sein geduldiges Feedback hat dieses Buch zu dem gemacht, was es heute ist. Dank an das gesamte Team, das hart gearbeitet hat, um das Buch aufzupolieren und alle unsere Fehler auszumerzen.

Zu guter Letzt danken wir Frederic Dumont, Matthew Flower, Rebecca Skinner und allen unerbittlichen Lesern. Wenn sie nicht diese Leidenschaft für das Lernen hätten, hätten wir keine Möglichkeit gehabt, sie zu befriedigen.

Falls wir jemanden vergessen haben, bitten wir um Entschuldigung. Jegliche Auslassungen waren sicher keine Absicht.

Von Eric: Liebe Noelle, du bist nicht nur etwas besonderes, du bist einzigartig und das ist wesentlich besser. Danke, dass du ein weiteres Buch mitgemacht hast. Dank auch an die Datenbank-Schöpfer und alle Beteiligten dafür, dass wir über etwas schreiben und davon leben können.

Von Jim: Zuerst muss ich meiner Familie danken. Ruthy, deine grenzenlose Geduld und Unterstützung waren herzerwärmend. Emma und Jimmy,

ihr seid zwei Wonneproppen und euer Vater wird euch immer lieben. Einen besonderen Dank auch an all die unbekannten Helden, die sich um IRC, Forumsnachrichten, Mailinglisten und Bugsysteme kümmern, und jedem helfen, der Hilfe braucht. Euer Einsatz für Open Source hält die Projekte am Laufen.

Vorwort

Man sagt, Daten seien das neue Öl. Wenn das so ist, sind Datenbanken die Ölfelder, die Raffinerien, die Bohrer und die Pumpen. Daten werden in Datenbanken gespeichert und wenn Sie sie nutzen wollen, ist es ein guter Einstieg, wenn man sich mit der modernen Ausrüstung auseinandersetzt.

Datenbanken sind Werkzeuge. Sie sind Mittel zum Zweck. Jede Datenbank hat ihre eigene Geschichte und ihre ganz eigene Sicht der Welt. Je besser Sie sie verstehen, desto besser können Sie die latente Macht des Ihnen zur Verfügung stehenden, stetig wachsenden Datenmaterials nutzen.

Warum sieben Datenbanken

Schon seit März 2010 wollten wir ein NoSQL-Buch schreiben. Der Begriff hat für einige Aufregung gesorgt und obwohl viele Leute über ihn reden, schien es um ihn herum recht viel Verwirrung zu geben. Was genau bedeutet der Begriff *NoSQL*? Welche Arten von Systemen sind enthalten? Wie wirkt sich das auf die Entwicklung guter Software aus? Das waren die Fragen, die wir beantworten wollten – für uns ebenso wie für andere.

Nachdem wir Bruce Tates beispielhaftes *Seven Languages in Seven Weeks: A Programming Guide to Learning Programming Languages* gelesen hatten, wussten wir, dass wir an etwas dran waren. Der progessive Stil, Progammiersprachen vorzustellen, brachte eine Saite in uns zum Klingen. Wir fühlten, dass seine Methode auch auf Datenbanken übertragen werden und helfen kann, einige dieser schwierigen NoSQL-Fragen zu beantworten.

Was Sie in diesem Buch finden

Dieses Buch richtet sich an erfahrene Entwickler, die sich ein vielseitiges Verständnis für die moderne Datenbank-Landschaft wünschen. Erfahrungen mit Datenbanken werden zwar nicht vorausgesetzt, sind aber hilfreich.

Nach einer kurzen Einführung nimmt dieses Buch Kapitel für Kapitel sieben Datenbanken in Angriff. Die Datenbanken wurden so gewählt, dass sie fünf

verschiedene Gattungen abdecken, die in Kapitel 1, *Einführung*, auf Seite 1 erläutert werden. Nach Kapiteln geordnet sind das PostgreSQL, Riak, Apache HBase, MongoDB, Apache CouchDB, Neo4J und Redis.

Jedes Kapitel wurde so konzipiert, dass man es (auf drei Tage verteilt) an einem langen Wochenende durcharbeiten kann. Jeder Tag endet mit Übungen, die auf den vorgestellten Themen und Konzepten aufbauen und jedes Kapitel endet mit einer Zusammenfassung der guten und schlechten Aspekte der Datenbank. Sie können die Dinge etwas schneller oder auch langsamer angehen, doch es ist wichtig, die Konzepte der einzelnen Tage zu begreifen, bevor Sie weitermachen. Wir haben versucht, Beispiele auszuwählen, die die charakteristischen Features einer Datenbank aufzeigen. Um wirklich zu verstehen, was diese Datenbanken zu bieten haben, müssen Sie sie eine Zeit lang nutzen, was bedeutet, dass Sie die Ärmel hochkrempeln und einiges an Arbeit erledigen müssen.

Auch wenn Sie versucht sein könnten, Kapitel zu überspringen, wurde das Buch so entworfen, dass man es von vorne bis hinten liest. Einige Konzepte wie Mapreduce werden in frühen Kapiteln ausführlich erläutert, während spätere Kapiteln nur oberflächlich auf sie eingehen. Das Ziel dieses Buches besteht darin, Ihnen ein solides Wissen über moderne Datenbanken zu vermitteln, weshalb wir Ihnen empfehlen, alles zu lesen.

Was dieses Buch nicht ist

Bevor Sie dieses Buch lesen, sollten Sie wissen, worüber wir nicht reden.

Es ist keine Installationsanleitung

Die Installation der Datenbanken in diesem Buch ist manchmal einfach, manchmal eine Herausforderung und manchmal geradezu hässlich. Bei einigen Datenbanken können Sie auf fertige Pakete zurückgreifen, während Sie bei anderen die Quellen kompilieren müssen. Wir geben hier und da ein paar nützliche Tipps, aber im Großen und Ganzen sind Sie auf sich selbst gestellt. Dadurch, dass wir die Installationsschritte außen vor lassen, haben wir mehr Platz für nützliche Beispiele und die Betrachtung von Konzepten. Und das ist es doch, was Sie eigentlich wollen, nicht wahr?

Administrationshandbuch? Eher nicht

Genau wie die Installation behandelt dieses Buch nicht jeden möglichen Punkt, den Sie im Adminstrationshandbuch finden. Alle Datenbanken besitzen unzählige Optionen, Einstellungen, Switches und Konfigurations-Details, von denen die meisten im Web gut dokumentiert sind. Wir sind mehr

daran interessiert, Ihnen nützliche Konzepte mit der nötigen Tiefe zu vermitteln, statt uns auf das Tagesgeschäft zu konzentrieren. Zwar können sich die Eigenschaften der Datenbanken durch entsprechende Einstellungen ändern – und wir besprechen diese Eigenschaften durchaus – , doch wir können nicht bei allen möglichen Konfigurationen ans Eingemachte gehen. Dafür ist einfach nicht genug Platz!

Ein Hinweis für Windows-Nutzer

In diesem Buch geht es naturgemäß um Auswahlmöglichkeiten von (vorwiegend) Open-Source-Software auf *nix-Plattformen. Microsoft-Umgebungen tendieren zu integrierten Umgebungen, wodurch die Wahlmöglichkeiten auf eine kleinere Menge beschränkt wird. Daher sind die von uns behandelten Datenbanken Open Source und von (und im Wesentlichen *für*) Benutzer von *nix-Systemen entwickelt worden. Das ist nicht so sehr unsere eigene Neigung, sondern spiegelt vielmehr den aktuellen Stand der Dinge wider. Folglich gehen wir davon aus, dass die Beispiele in einer *nix-Shell ausgeführt werden. Wenn Sie mit Windows arbeiten und es dennoch probieren wollen, empfehlen wir die Installation von Cygwin[1], um Ihre Erfolgsaussichten zu erhöhen. Sie könnten auch den Betrieb einer virtuellen Linux-Maschine in Erwägung ziehen.

Codebeispiele und Konventionen

Dieses Buch enthält Code in einer Vielzahl von Sprachen. Zum Teil liegt das an den von uns betrachteten Datenbanken. Wir haben versucht, uns bei den Sprachen auf Ruby/JRuby und JavaScript zu beschränken. Wir bevorzugen Kommandozeilen-Tools gegenüber Skripten, doch wir stellen auch andere Sprachen vor, um eine Aufgabe zu erledigen – beispielsweise PL/pgSQL (Postgres) und Gremlin/Groovy (Neo4J). Wir betrachten auch die Entwicklung serverseitiger JavaScript-Anwendungen mit Node.js.

Wenn nicht anders vermerkt, sind Code-Listings vollständig und können ganz nach Wunsch einfach ausgeführt werden. In Beispielen und Codefragmenten erfolgt eine Syntaxhervorhebung nach den Regeln der jeweils genutzten Sprache. Shell-Befehlen ist ein $ vorangestellt.

1. http://www.cygwin.com/

Online-Ressourcen

Die Pragmatic Bookshelf-Seite zu diesem Buch[2] ist eine gute Ressource. Sie finden Downloads mit dem gesamten in diesem Buch vorgestellten Quellcode. Sie finden auch Feedback-Tools wie etwa ein Community-Forum und ein Errata-Formular, in dem Sie Änderungen zukünftiger Releases vorschlagen können.

Vielen Dank, dass Sie uns auf unserer Reise durch die moderne Datenbanklandschaft begleiten.

Eric Redmond und Jim R. Wilson

2. http://pragprog.com/book/rwdata/seven-databases-in-seven-weeks

KAPITEL 1

Einführung

Dies ist eine wichtige Zeit für die Datenbankwelt. Jahrelang war das relationale Modell der *de facto*-Standard für große und kleine Probleme. Wir gehen nicht davon aus, dass relationale Datenbanken in absehbarer Zeit verschwinden, doch der RDBMS-Nebel lichtet sich zunehmend und die Leute entdecken neue Möglichkeiten wie schemafreie oder alternative Datenstrukturen, einfache Replikation, Hochverfügbarkeit, horizontale Skalierung und neue Abfragemethoden. Diese Möglichkeiten nennt man kollektiv *NoSQL* und sie machen den Großteil dieses Buches aus.

In diesem Buch untersuchen wir sieben Datenbanken, die das gesamte Spektrum verfügbarer Datenbanktypen abdecken. Im Verlauf des Buches lernen Sie die verschiedenen Funktionalitäten und Kompromisse aller Datenbanken kennen – Stabilität gegen Geschwindigkeit, absolute gegen schlussendliche Konsistenz und so weiter – und wie man die besten Entscheidungen für den jeweiligen Anwendungsfall trifft.

1.1 Es beginnt mit einer Frage

Die zentrale Frage von *Sieben Wochen, sieben Datenbanken* lautet: Welche Datenbank oder welche Kombination von Datenbanken löst Ihr Problem am besten? Wenn Sie dieses Buch gelesen haben und verstehen, wie Sie diese Entscheidung (unter Berücksichtigung Ihrer jeweiligen Bedürfnisse und Ressourcen) treffen können, sind wir zufrieden.

Doch um diese Frage beantworten zu können, müssen Sie die Möglichkeiten verstehen. Darum tauchen wir tief in jede der sieben Datenbanken ein, um die guten, aber auch die weniger guten Seiten aufzuzeigen. Sie werden sich mit CRUD die Hände schmutzig machen, Ihre Schema-Muskeln spielen lassen und Antworten auf die folgenden Fragen finden:

- *Um welche Art von Datenspeicher (Datastore) handelt es sich?* Datenbanken gibt es in einer Vielzahl von Arten wie zum Beispiel relational, Schlüssel/Wert, spaltenorientiert, dokumentenorientiert und graphenbasiert. Populäre Datenbanken – einschließlich der in diesem Buch behandelten – können grundsätzlich in eine dieser Kategorien eingeordnet werden. Sie werden alle Arten kennenlernen sowie die Arten von Problemen, für die sie am besten geeignet sind. Wir haben die Datenbanken gezielt ausgewählt, um diese Kategorien abzudecken, einschließlich einer relationalen Datenbank (Postgres), zwei Schlüssel/Wert-Speichern (Riak, Redis), einer spaltenorientierten Datenbank (HBase), zwei dokumentenorientierten Datenbanken (MongoDB, CouchDB) und einer Graph-Datenbank (Neo4J).
- *Was war die treibende Kraft?* Datenbanken werden nicht in einem Vakuum entwickelt. Sie werden entworfen, um Probleme zu lösen, die auf realen Anwendungsfällen basieren. Relationale Datenbank-Management-Systeme (RDBMS) entstanden in einer Welt, in der die Flexibilität der Abfrage (Query) wichtiger war als die Flexibilität des Schemas. Demgegenüber wurden spaltenorientierte Datenbanken entwickelt, um große Datenmengen auf mehrere Rechner verteilen zu können (während die Beziehungen der Daten in den Hintergrund traten). Wir betrachten Beispiele für den Einsatz jeder dieser Datenbanken.
- *Wie spricht man mit ihr?* Datenbanken unterstützen häufig eine Vielzahl von Verbindungsoptionen. Wenn eine Datenbank eine interaktive Kommandozeilen-Schnittstelle besitzt, fangen wir mit ihr an, bevor wir uns anderen Möglichkeiten zuwenden. Wo eine Programmierung nötig ist, halten wir uns hauptsächlich an Ruby und JavaScript, auch wenn ab und zu andere Sprachen auftauchen – wie etwa PL/pgSQL (Postgres) und Gremlin (Neo4J). Auf einer niedrigeren Ebene behandeln wir Protokolle wie REST (CouchDB, Riak) und Thrift (HBase). Im letzten Kapitel stellen wir einen etwas komplexeren Datenbank-Aufbau vor, der über eine Node.js JavaScript-Implementierung zusammengehalten wird.
- *Was macht sie einzigartig?* Alle Datenspeicher unterstützen das Schreiben von Daten und können siw natürlich auch wieder auslesen. Was sie sonst noch können, variiert von System zu System sehr stark. Einige erlauben die Abfrage beliebiger Felder. Einige bieten eine Indexierung zur schnellen Suche. Einige unterstützen Ad-hoc-Queries, bei anderen müssen die Abfragen geplant werden. Ist das Schema ein rigider Rahmen, der durch die Datenbank erzwungen wird, oder nur eine Reihe von Richtlinien, über die bei Bedarf noch einmal verhandelt werden kann? Die Fähigkeiten und Einschränkungen zu verstehen, hilft Ihnen dabei, die richtige Datenbank für den Job zu wählen.

- *Was leistet sie?* Wie funktioniert die Datenbank und zu welchem Preis? Unterstützt sie die Fragmentierung? Was ist mit Replikation? Verteilt sie Daten gleichmäßig über konsistentes Hashing oder hält sie die Daten gern beisammen? Ist die Datenbank für das Lesen optimiert oder für das Schreiben oder für irgendeine andere Operation? Wie viel Kontrolle haben Sie (wenn überhaupt) über diese Optimierung?
- *Wie skaliert sie?* Skalierbarkeit steht im Zusammenhang mit Performance. Über Skalierbarkeit zu reden, ohne den Kontext zu betrachten, in dem *skaliert* werden soll, ist üblicherweise vergebliche Liebesmüh. Dieses Buch liefert Ihnen das Hintergrundwissen, das Sie brauchen, um die richtigen Fragen stellen zu können, um diesen Kontext herzustellen. Zwar ist die Diskussion darüber, *wie* jede Datenbank skaliert, bewusst einfach gehalten, doch Sie werden lernen, ob die Datenspeicher eher horizontal (MongoDB, HBase, Riak), traditionell vertikal (Postgres, Neo4J, Redis) oder irgendwo dazwischen skalieren.

Es ist nicht unser Ziel, Neulinge zu Meistern der jeweiligen Datenbanken zu machen. Eine umfassende Betrachtung jeder einzelnen könnte ganze Bücher füllen (und tut das auch). Doch zum Schluss sollten Sie eine klare Vorstellung von den jeweiligen Stärken haben und wie sie sich unterscheiden.

1.2 Die Gattungen

Wie Musik lassen sich auch Datenbanken grob in ein oder mehrere Gattungen unterteilen. Ein einzelner Song kann die gleichen Noten aufweisen wie ein anderer und doch für bestimmte Situationen besser geeignet sein. Nur wenige Leute werden Bachs *h-Moll-Messe* voll aufdrehen, während Sie mit dem offenen Cabrio die Landstraße entlang fahren. In gleicher Weise sind einige Datenbanken für bestimmte Fälle besser geeignet als andere. Die Frage, die Sie sich selbst immer stellen müssen, lautet nicht „Kann ich diese Datenbank nutzen, um meine Daten zu speichern und zu bearbeiten?“, sondern vielmehr „Sollte ich?“

In diesem Abschnitt wollen wir die fünf Haupt-Datenbank-Gattungen untersuchen. Wir werfen auch einen Blick auf die Datenbanken, die wir uns für die jeweilige Gattung ausgesucht haben.

Es ist wichtig, daran zu denken, dass Sie die meisten Aufgaben mit den meisten (oder allen) in diesem Buch vorgestellten Datenbanken lösen können (von anderen Datenbanken ganz zu schweigen). Es geht weniger um die Frage, ob eine bestimmte Datenbank-Gattung Ihre Daten abbilden kann, sondern vielmehr darum, ob sie am besten zu Ihrem Problemfeld, Ihren Nutzungs-

mustern und den verfügbaren Ressourcen passt. Sie werden lernen herauszufinden, ob eine Datenbank wirklich für Sie geeignet ist.

Relational

Das relationale Modell kommt üblicherweise den meisten Leuten mit Datenbank-Erfahrung in den Sinn. Relationale Datenbank-Management-Systeme (RDBMSs) basieren auf der Mengenlehre und werden in zweidimensionalen Tabellen mit Zeilen und Spalten implementiert. Das vorschriftsmäßige Mittel zur Kommunikation mit einem RDBMS besteht im Schreiben von Queries in SQL (Structured Query Language). Die Datenwerte sind typisiert und es gibt numerische Typen, Strings, Datumswerte, uninterpretierte Blobs und weitere Typen. Die Typen werden durch das System vorgegeben. Zudem können Tabellen zu neuen, komplexeren Tabellen verknüpft und umgewandelt werden, da ihre mathematische Grundlage die relationale Theorie (Mengenlehre) ist.

Sie können aus einer Vielzahl an relationalen Open Source-Datenbanken wählen, darunter MySQL, H2, HSQLDB, SQLite, und vielen anderen. Wir beschäftigen uns mit Kapitel 2, *PostgreSQL*, auf Seite 9.

PostgreSQL

Das kampferprobte PostgreSQL ist die mit Abstand älteste und robusteste von uns betrachtete Datenbank. Dank des Festhaltens am SQL-Standard wird sich jeder wohlfühlen, der schon mal mit relationalen Datenbanken gearbeitet hat, und sie bildet eine solide Grundlage für Vergleiche mit den anderen hier vorgestellten Datenbanken. Wir werden auch einige der eher unbekannten SQL-Features untersuchen und Postgres-spezifische Vorteile vorstellen. Vom SQL-Einsteiger bis zum Experten ist hier für jeden etwas dabei.

Schlüssel/Wert (Key-Value)

Der Schlüssel/Wert-Speicher (engl. key-value, kurz KV) ist das einfachste von uns betrachtete Modell. Wie der Name andeutet, verknüpft ein KV-Speicher Schlüssel mit Werten, so wie das eine Map (oder Hashtabelle) bei einigen bekannten Programmiersprachen tut. Einige KV-Implementierungen erlauben komplexe Wertetypen wie Hashes oder Listen, aber das ist nicht notwendig. Einige KV-Implementierungen erlauben die Iteration über die Schlüssel, aber auch das ist ein zusätzlicher Bonus. Man kann ein Dateisystem als KV-Speicher betrachten, wenn man sich den Dateipfad als Schlüssel und den Dateiinhalt als Wert vorstellt. Da KV-Speicher so anspruchslos sind, können

sie in einer Reihe von Szenarien unglaublich performant sein, bei komplexeren Abfragen und Aggregationen sind sie jedoch wenig hilfreich.

Wie bei relationalen Datenbanken stehen auch hier viele Open-Source-Lösungen zur Verfügung. Einige der bekannteren Vertreter dieser Gattung sind memcached (und dessen Cousins memcachedb und membase), Voldemort und die beiden, die wir in diesem Buch betrachten: Redis und Riak.

Riak

Riak, das in Kapitel 3, *Riak*, auf Seite 57 behandelt wird, ist mehr als ein Schlüssel/Wert-Speicher. Es unterstützt von Grund auf Web-Konzepte wie HTTP und REST. Es stellt eine originalgetreue Implementierung von Amazons Dynamo dar, besitzt aber fortgeschrittene Features wie Vektoruhren zur Auflösung von Konflikten. Werte können bei Riak alles sein, von reinem Text über XML bis hin zu Bilddaten. Die Beziehungen zwischen Schlüsseln werden über benannte Strukturen, sog. *Links*, beschrieben. Als eine der weniger bekannten Datenbanken in diesem Buch gewinnt Riak zunehmend an Popularität, und sie ist die erste, über die wir reden, die fortgeschrittene Queries über mapreduce unterstützt.

Redis

Redis bietet komplexe Datentypen wie sortierte Sets (Mengen) und Hashes sowie grundlegende Messaging-Muster wie Publish/Subscribe und blockierende Queues. Es besitzt darüber hinaus einen der stabilsten Query-Mechanismen für KV-Stores. Und weil Schreiboperationen zuerst im Speicher gecached werden, bevor sie über einen Commit auf die Platte geschrieben werden, ist die Performance von Redis beachtlich, auch wenn bei einem Hardware-Fehler die Gefahr eines Datenverlustes steigt. Diese Eigenschaft macht sie zu einem guten Kandidaten zum Caching unkritischer Daten und als Message-Broker. Wir heben sie uns bis zum Schluss auf – siehe Kapitel 8, *Redis*, auf Seite 285 – , so dass wir mit Redis und anderen Datenbanken eine harmonische Multidatenbank-Anwendung aufbauen können.

Spaltenorientiert

Spaltenorientierte Datenbanken werden so genannt, weil der wichtigste Aspekt ihres Entwurfs darin besteht, dass die Daten einer gegebenen Spalte (im Sinne einer zweidimensionalen Tabelle) zusammen gespeichert werden. Im Gegensatz dazu werden bei zeilenorientierten Datenbanken (wie einem RDBMS) Informationen einer Zeile zusammen festgehalten. Der Unterschied mag belanglos erscheinen, doch die Auswirkungen sind enorm. Bei spaltenorientierten Datenbanken ist das Einfügen von Spalten mit geringen Kosten

verbunden und erfolgt Zeile für Zeile. Jede Zeile kann eine andere Menge an Spalten besitzen, auch gar keine, wodurch die Tabellen *dünn besetzt (sparse)* bleiben, ohne Speicherkosten für Nullwerte zu erzeugen. Hinsichtlich der Struktur ist eine spaltenorientierte Datenbank ein Mittelding zwischen relationaler und KV-Datenbank.

Auf dem Markt spaltenorientierter Datenbanken herrscht etwas weniger Wettbewerb als bei relationalen Datenbanken oder KV-Stores. Die drei bekanntesten Vertreter sind HBase (das wir in Kapitel 4, *HBase*, auf Seite 103 behandeln), Cassandra und Hypertable.

HBase

Diese spaltenorientierte Datenbank hat von allen hier behandelten nichtrelationalen Datenbanken die meisten Gemeinsamkeiten mit dem relationalen Modell. Mit Googles BigTable-Papier als Blaupause baut HBase auf Hadoop (einer mapreduce-Engine) auf und wurde entworfen, um horizontal gut auf Clustern handelsüblicher Hardware zu skalieren. HBase gibt hohe Garantien im Bezug auf die Konsistenz ab und kennt Tabellen mit Zeilen und Spalten – wodurch sich SQL-Fans gleich heimisch fühlen sollten. Durch die direkte Unterstützung von Versionierung und Komprimierung hebt sich diese Datenbank im „Big Data"-Bereich ab.

Dokumentenorientiert

Dokumentenorientierte Datenbanken speichern, nun ja, Dokumente. Kurz gesagt ist ein Dokument wie ein Hash. Es enthält ein eindeutiges ID-Feld und Werte unterschiedlichster Typen, einschließlich weiterer Hashes. Dokumente können verschachtelte Strukturen enthalten und bieten daher einen hohen Grad an Flexibilität, was variable Bereiche erlaubt. Das System legt den eingehenden Daten nur wenige Beschränkungen auf, solange sie die grundlegende Anforderung erfüllen, in Form eines Dokuments ausgedrückt werden zu können. Die verschiedenen Dokument-Datenbanken verfolgen unterschiedliche Strategien im Bezug auf Indexierung, Ad-hoc-Abfragen, Replikation, Konsistenz und andere Design-Entscheidungen. Um vernünftig zwischen ihnen wählen zu können, müssen Sie diese Unterschiede verstehen und wissen, wie sich das auf Ihren jeweiligen Anwendungsfall auswirkt.

Die beiden wichtigsten Open-Source-Player auf dem Markt der Dokumenten-Datenbanken sind MongoDB, die wir in Kapitel 5, *MongoDB*, auf Seite 147 behandeln, und CouchDB, die in Kapitel 6, *CouchDB*, auf Seite 193 diskutiert wird.

MongoDB

MongoDB ist für *riesige* Datenmengen gedacht (der Name *Mongo* entstammt dem englischen Wort hu*mongo*us, zu deutsch gigantisch). Mongo-Serverkonfigurationen versuchen, konsistent zu bleiben – wenn Sie etwas schreiben, empfangen nachfolgende Leseoperationen immer den gleichen Wert (bis zum nächsten Update). Dieses Feature macht es für Leute mit RDBMS-Hintergrund attraktiv. Sie bietet außerdem atomische Schreib-/Leseoperationen wie etwa die Inkrementierung eines Wertes und Tiefenabfragen verschachtelter Dokumentenstrukturen. Mit JavaScript als Abfragesprache unterstützt MongoDB sowohl einfache Queries als auch komplexe mapreduce-Jobs.

CouchDB

CouchDB zielt auf eine Vielzahl unterschiedlicher Anwendungsszenarien vom Rechenzentrum über den Desktop bis hinunter zum Smartphone. In Erlang geschrieben, besitzt CouchDB eine einzigartige Robustheit, die den anderen Datenbanken größtenteils fehlt. Mit nahezu unzerstörbaren Daten-Dateien bleibt CouchDB auch dann noch hochverfügbar, wenn die Verbindung verloren geht oder ein Hardware-Fehler eintritt. Wie bei Mongo ist auch bei CouchDB JavaScript die systemeigene Abfragesprache. Views bestehen aus mapreduce-Funktionen, die als Dokumente gespeichert und zwischen den Knoten wie alle anderen Daten auch repliziert werden.

Graph

Eine der nicht ganz so gängigen Datenbank-Gattungen, die Graph-Datenbank, glänzt beim Umgang mit hochgradig verbundenen Daten. Eine Graph-Datenbank besteht aus Knoten und Beziehungen zwischen diesen Knoten. Sowohl Knoten als auch Beziehungen können Eigenschaften – Schlüssel/Wert-Paare – haben, die Daten speichern. Die eigentliche Stärke von Graph-Datenbanken ist das Verarbeiten der Knoten über das Verfolgen von Beziehungen.

In Kapitel 7, *Neo4J*, auf Seite 237 behandeln wir die heute beliebteste Graph-Datenbank Neo4J.

Neo4J

Eine Operation, bei der andere Datenbanken häufig scheitern, ist die Verarbeitung selbstreferenzierender oder auf andere Weise kompliziert verknüpfter Daten. Genau hier glänzt Neo4J. Der Vorteil der Verwendung einer Graph-Datenbank ist die Fähigkeit, Knoten und Beziehungen schnell durchgehen zu können, um relevante Daten zu finden. Man findet sie häufig bei Anwendun-

gen für soziale Netzwerke und sie nehmen aufgrund ihrer Flexibilität immer mehr Fahrt auf. Neo4j ist dabei die Vorzeige-Implementierung.

Polyglot

Im richtigen Leben werden Datenbanken häufig zusammen mit anderen Datenbanken genutzt. Zwar ist es durchaus üblich, mit einer einzelnen relationalen Datenbank zu arbeiten, doch mittlerweile nutzt man gerne verschiedene Datenbanken, um ihre Stärken zu nutzen. Auf diese Weise lässt sich ein Ökosystem aufbauen, das leistungsfähiger und robuster ist als die Summe seiner Teile. Diese Praxis bezeichnet man als *polyglotte Persistenz* und ist ein Thema, dem wir uns in Kapitel 9, *Abschließende Zusammenfassung*, auf Seite 335 widmen wollen.

1.3 Vor- und aufwärts

Wir befinden uns mitten in einer kambrischen Explosion von Datenspeicher-Optionen. Es ist schwierig, genau vorherzusagen, was sich als Nächstes entwickelt. Wir können aber recht sicher sein, dass die Dominanz einer bestimmten Strategie (relational oder sonstwie) eher unwahrscheinlich ist. Vielmehr werden wir eine steigende Zahl spezialisierter Datenbanken sehen, die für eine bestimmte (wenn auch überlappende) Menge von Problembereichen ideal geeignet sind. Und so wie es heute Jobs gibt, die Fachkenntnisse bei der Administration relationaler Datenbanken voraussetzen, werden Jobs für die nicht-relationalen Pendants entstehen.

Datenbanken sind, wie Programmiersprachen und Bibliotheken, eine weitere Reihe von Werkzeugen, die jeder Entwickler kennen sollte. Jeder gute Schreiner muss verstehen, was da in seinem Werkzeuggürtel hängt. Und wie jeder gute Handwerker können Sie nicht hoffen, ein Meister zu sein, ohne die vielen Möglichkeiten zu kennen, die Ihnen zur Verfügung stehen.

Betrachten Sie das Buch als Crashkurs in einer Schreinerei. Sie werden einige Hämmer schwingen, Bohrmaschinen wirbeln lassen, mit Nagelpistolen herumspielen und am Ende weit mehr als ein Vogelhäuschen bauen können. Wenden wir uns also ohne langes Federlesen der ersten Datenbank zu: PostgreSQL.

KAPITEL 2

PostgreSQL

PostgreSQL ist sozusagen der Hammer der Datenbankwelt. Es wird gemeinhin verstanden, ist häufig direkt verfügbar, stabil und löst eine überraschende Anzahl von Problemen, wenn man den Hammer denn fest genug schwingt. Niemand darf sich für einen Meister halten, der dieses gebräuchlichste aller Werkzeuge nicht versteht.

PostgreSQL ist ein relationales Datenbank-Management-System, d. h. ein auf der Mengenlehre basierendes System, implementiert in zweidimensionalen Tabellen mit Datenzeilen und zwingend vorgeschriebenen Spaltentypen. Trotz des wachsenden Interesses an neuen Datenbank-Trends ist der relationale Stil am weitesten verbreitet und das wird sich in absehbarer Zeit wohl auch nicht ändern.

Die Vorherrschaft relationaler Datenbanken liegt aber nicht nur am riesigen Werkzeugkasten (Trigger, Stored Procedures, fortschrittliche Indizes), der Datensicherheit (durch ACID-Konformität) oder der Marktbedeutung (viele Programmierer sprechen und denken relational), sondern auch an der Flexibilität der Queries. Im Gegensatz zu einigen anderen Datenspeichern müssen Sie nicht wissen, wie Sie die Daten nutzen wollen. Wenn das relationale Schema normalisiert ist, sind die Abfragen flexibel. PostgreSQL ist das beste Open-Source-Beispiel für die Tradition relationaler Datenbank-Management-Systeme.

2.1 Das ist Post-greS-Q-L

PostgreSQL ist die mit Abstand älteste und kampferprobteste Datenbank in diesem Buch. Sie besitzt Plugins zum Parsing natürlicher Sprachen, zur mehrdimensionalen Indexierung, geographische Queries, eigene Datentypen und vieles mehr. Sie besitzt eine ausgereifte Transktionsverarbeitung, fest

Was hat es mit dem Namen auf sich?

PostgreSQL existiert in der aktuellen Projekt-Inkarnation seit 1995, doch seine Wurzeln sind deutlich älter. Das Originalprojekt entstand in den frühen 1970ern in Berkeley unter dem Namen Interactive Graphics and Retrieval System, kurz „Ingres“. In den 1980ern wurde eine verbesserte Version veröffentlicht: post-Ingres (also „nach Ingres“), kurz Postgres. Das Projekt wurde 1993 in Berkeley abgeschlossen, von der Open-Source-Gemeinde als Postgres95 aber wieder aufgenommen. 1996 wurde es dann in PostgreSQL umbenannt, um die neue SQL-Unterstützung hervorzuheben. Dabei ist es seither geblieben.

eingebaute Stored Procedures für ein Dutzend Sprachen und läuft auf vielen Plattformen. PostgreSQL unterstützt von Haus aus Unicode, Sequenzen, Tabellen-Vererbung und Subselects und ist eine der ANSI SQL – konformsten relationalen Datenbanken auf dem Markt. Sie ist schnell und zuverlässig, kann Terabytes an Daten verarbeiten und hat ihre Leistungsfähigkeit auch in prominenten Projekten wie Skype, France's Caisse Nationale d'Allocations Familiales (CNAF) und der United States' Federal Aviation Administration (FAA) unter Beweis gestellt.

Sie können PostgreSQL, je nach Betriebssystem, auf unterschiedlichste Art installieren.[1] Neben der grundlegenden Installation müssen Sie Postgres um die folgenden Pakete erweitern: `tablefunc`, `dict_xsyn`, `fuzzystrmatch`, `pg_trgm` und `cube`. Installationshinweise finden Sie auf der Website.[2]

Nachdem Postgres installiert ist, erzeugen Sie mit dem folgenden Befehl ein Buch-Schema namens `book`:

```
$ createdb book
```

Wir werden dieses `book`-Schema für den Rest dieses Kapitels nutzen. Führen Sie nun den folgenden Befehl aus, um sicherzustellen, dass die Pakete korrekt installiert wurden:

```
$ psql book -c  "SELECT '1'::cube;"
```

Informationen zu eventuellen Fehlermeldungen finden Sie in der Online-Dokumentation.

1. http://www.postgresql.org/download/
2. http://www.postgresql.org/docs/9.0/static/contrib.html

2.2 Tag 1: Relationen, CRUD und Joins

Wir gehen zwar nicht davon aus, dass Sie ein Experte für relationale Datenbanken sind, erwarten aber doch, dass Sie in der Vergangenheit mit ein oder zwei Datenbanken in Berührung gekommen sind. Und die Chancen stehen gut, dass diese Datenbank relational war. Wir wollen damit beginnen, eigene Schemata zu erzeugen und mit Daten zu füllen. Danach sehen wir uns die Abfrage von Werten an und zum Schluss das, was relationale Datenbanken so besonders macht: den Join.

Wie die meisten Datenbanken, die wir hier behandeln, stellt Postgres einen Backend-Server bereit, der die gesamte Arbeit erledigt, und eine Kommandozeilen-Shell, die die Verbindung mit dem laufenden Server herstellt. Der Server kommuniziert standardmäßig über Port 5432, über den Sie mit der `psql`-Shell die Verbindung herstellen können.

```
$ psql book
```

PostgreSQL gibt den Namen der Datenbank aus, gefolgt von einem Rautensymbol (als Administrator) bzw. einem Dollarzeichen (als normaler Benutzer). Die Shell verfügt über eine integrierte Dokumentation, die an jeder Console zur Verfügung steht. Die Eingabe von `\h` gibt Informationen zu SQL-Befehlen aus und `\?` hilft bei `psql`-spezifischen Befehlen, namentlich denen, die mit einem Backslash beginnen. Nutzungshinweise zu jedem SQL-Befehl können Sie sich also wie folgt ausgeben lassen:

```
book=# \h CREATE INDEX
Command:     CREATE INDEX
Description: define a new index
Syntax:
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | ...
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

Bevor wir tiefer in Postgres einsteigen, sollten Sie sich mit diesem Tool vertraut machen. Es ist durchaus sinnvoll, sich ein paar gängige Befehle wie `SELECT` oder `CREATE TABLE` anzusehen.

Einstieg in SQL

PostgreSQL folgt der SQL-Konvention, nach der Relationen als Tabellen (`TABLE`s), Attribute als Spalten (`COLUMN`s) und Tupel als Zeilen (`ROW`s) bezeichnet werden. Der Konsistenz halber verwenden wir diese Terminologie, auch wenn Ihnen die mathematischen Begriffe *Relationen*, *Attribute* und *Tupel* begegnen

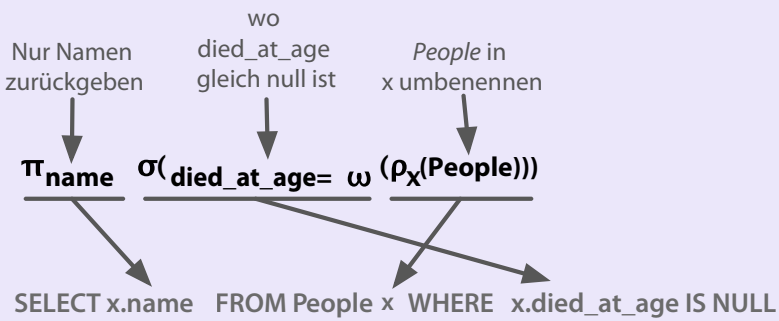
werden. Mehr zu diesen Konzepten erfahren Sie im Kasten *Mathematische Relationen*, auf Seite 12.

Mathematische Relationen

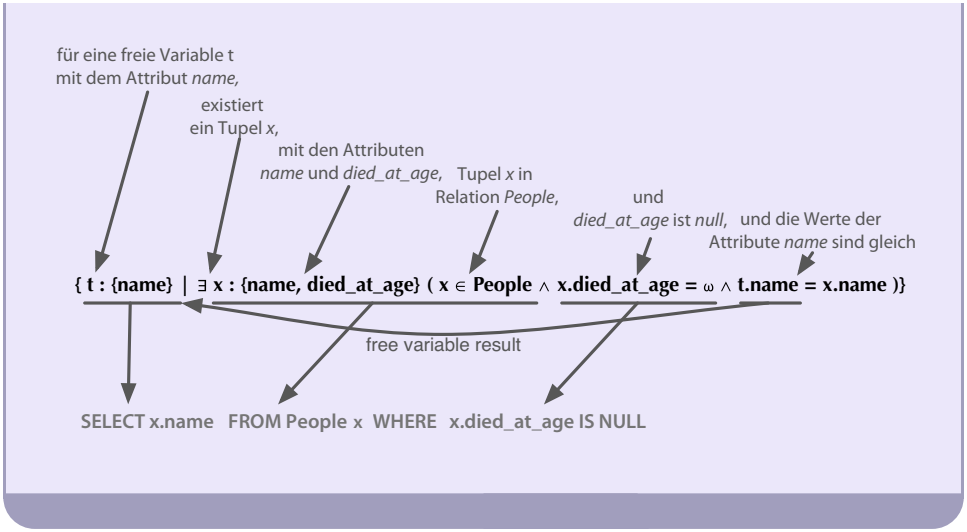
Relationale Datenbanken werden so genannt, weil sie *Relationen* (d. h. Tabellen) enthalten, die aus Gruppen (Mengen) von *Tupeln* (d. h. Zeilen) bestehen, die *Attribute* auf atomische Werte (z. B. `{name: 'Genghis Khan', p.died_at_age: 65}`) abbilden. Die verfügbaren Attribute werden durch ein *Header*-Attribut-Tupel definiert, das auf eine bestimmte Domäne (*domain*) oder einen beschränkenden Typ (d. h. Spalten; beispielsweise `{name: string, age: int}`) abgebildet wird. Das ist der Kern der relationalen Struktur.

Die Implementierungen sind wesentlich praktischer orientiert, als es die mathematischen Namen vermuten lassen. Warum erwähnen wir sie dann überhaupt? Wir wollen hervorheben, dass relationale Datenbanken *relational* im mathematischen Sinne sind. Sie sind nicht relational, weil Tabellen über Fremdschlüssel „in Relation“ zueinander stehen. (Ob es solche Beschränkungen tatsächlich gibt, ist hier nicht der Punkt.)

Auch wenn ein Großteil der Mathematik vor Ihnen versteckt wird, basiert die Leistungfähigkeit dieses Modells doch genau in dieser Mathematik. Sie erlaubt es dem Benutzer, mächtige Abfragen zu formulieren, die das System basierend auf vordefinierten Mustern optimiert. RDBMS bauen auf einem Bereich der Mengenlehre auf, der *relationale Algebra* genannt wird – einer Kombination aus Selektionen (`WHERE ...`), Projektionen (`SELECT ...`), Kartesischen Produkten (`JOIN ...`) und vielem mehr:



Sich eine Relation als physikalische Tabelle (ein Array von Arrays, das in Datenbank-Einführungskursen *ad infinitum* wiederholt wird) vorzustellen, kann einen in der Praxis richtig leiden lassen, etwa wenn man Code schreibt, der über alle Zeilen iteriert. Relationale Queries sind da wesentlich deklarativer, da sie einem Zweig der Mathematik entstammen, der als *Tupelkalkül* bezeichnet wird und in relationale Algebra umgewandelt werden kann. PostgreSQL und andere RDBMS optimieren Abfragen, indem sie diese Umwandlung vornehmen und die Algebra vereinfachen. Wie Sie sehen können, entspricht die SQL im unteren Diagramm der des obigen Diagramms.



Arbeiten mit Tabellen

PostgreSQL ist, als Vertreter des relationalen Stils, ein „Design zuerst"-Datenspeicher. Sie entwerfen zuerst das Schema und geben dann die Daten ein, die der Definition dieses Schemas entsprechen.

Um eine Tabelle zu erzeugen, müssen Sie ihr einen Namen geben sowie eine Liste von Spalten mit den jeweiligen Typen und optionalen Beschränkungen. Jede Tabelle sollte auch eine Spalte als eindeutigen Identifier festlegen, der bestimmte Zeilen eindeutig identifiziert. Dieser Identifier wird als Primärschlüssel (`PRIMARY KEY`) bezeichnet. Der SQL-Code zum Aufbau einer `countries`-Tabelle sieht wie folgt aus:

```
CREATE TABLE countries (
  country_code char(2) PRIMARY KEY,
  country_name text UNIQUE
);
```

Diese Tabelle speichert eine Reihe von Zeilen, die über einen Zwei-Zeichen-Code identifiziert werden und bei denen der Name eindeutig (unique) ist. Für beide Spalten gelten Beschränkungen (*Constraints*). Der `PRIMARY KEY` beschränkt die `country_code`-Spalte auf eindeutige Ländercodes (d. h., Duplikate sind nicht erlaubt). Nur ein `us` und ein `gb` können vorhanden sein. Wir haben `country_name` explizit eine vergleichbare UNIQUE-Beschränkung auferlegt, auch wenn es sich nicht um den primären Schlüssel handelt. Wir befüllen die `countries`-Tabelle, indem wir einige Zeilen einfügen.

```
INSERT INTO countries (country_code, country_name)
VALUES ('us','United States'), ('mx','Mexico'), ('au','Australia'),
       ('gb','United Kingdom'), ('de','Germany'), ('ll','Loompaland');
```

Probieren wir unsere „Eindeutig"-Beschränkung aus. Wenn Sie versuchen, einen `country_name`n doppelt einzugeben, schlägt die unique-Beschränkung fehl und das erneute Einfügen wird unterbunden. Mit Constraints stellen relationale Datenbanken wie PostgreSQL koschere Daten sicher.

```
INSERT INTO countries
VALUES ('uk','United Kingdom');

ERROR:  duplicate key value violates unique constraint "countries_country_name_key"
DETAIL:  Key (country_name)=(United Kingdom) already exists.
```

Wir können überprüfen, ob die Zeilen korrekt eingefügt wurden, indem wir den Tabellenbefehl `SELECT...FROM` nutzen.

```
SELECT *
FROM countries;

country_code | country_name
-------------+---------------
 us          | United States
 mx          | Mexico
 au          | Australia
 gb          | United Kingdom
 de          | Germany
 ll          | Loompaland
(6 rows)
```

Jeder halbwegs respektable Atlas wird bestätigen, dass es Loompaland gar nicht gibt – wir wollen es daher aus der Tabelle entfernen. Wir legen die zu löschende Zeile in der `WHERE`-Klausel fest, d. h., die Zeile mit dem `country_code` `ll` wird entfernt.

```
DELETE FROM countries
WHERE country_code = 'll';
```

Nachdem nur noch reale Länder in unserer `countries`-Tabelle stehen, wollen wir eine `cities`-Tabelle mit Städten anlegen. Um sicherzustellen, dass jeder eingefügte `country_code` auch in unserer `countries`-Tabelle existiert, nutzen wir das Schlüsselwort `REFERENCES`. Dass die `country_code`-Spalte den Schlüssel einer anderen Tabelle referenziert, bezeichnen wir als *Fremdschlüssel (foreign key)*-Constraint.

Über CRUD

CRUD ist eine nützliche Merkhilfe für die elementaren Operationen des Datenmanagements: *Create*, *Read*, *Update* und *Delete*. Auf deutsch also Erzeugen, Lesen, Aktualisieren und Löschen. Das entspricht grundsätzlich dem Einfügen neuer Datensätze (*Create*), der Modifikation bestehender Datensätze (*Update*) und dem Entfernen nicht mehr benötigter Datensätze (*Delete*). Alle anderen Operationen, für die Sie eine Datenbank nutzen (also jede noch so verrückte Query, die Sie sich vorstellen können), sind *Leseoperationen*. Wenn Sie CRUD können, können Sie alles.

```
CREATE TABLE cities (
  name text NOT NULL,
  postal_code varchar(9) CHECK (postal_code <> ''),
  country_code char(2) REFERENCES countries,
  PRIMARY KEY (country_code, postal_code)
);
```

Diesmal haben wir den Städtenamen so eingeschränkt, dass er keine `NULL`-Werte enthalten darf. Wir prüfen auch die Postleitzahl (`postal_code`) dahingehend, dass sie keinen Leerstring enthält (`<>` bedeutet *nicht gleich*). Und weil der `PRIMARY KEY` eine Zeile eindeutig identifiziert, haben wir einen zusammengesetzten Schlüssel (compound key) angelegt: `country_code` + `postal_code`. Diese beiden zusammen identifizieren eine Zeile eindeutig.

Postgres kennt darüber hinaus eine Vielzahl von Datentypen. Sie haben bereits drei verschiedene Arten von Strings kennengelernt: `text` (ein String beliebiger Länge), `varchar(9)` (ein String variabler Länge mit bis zu neun Zeichen) und `char(2)` (ein String mit genau zwei Zeichen). In dieses Schema wollen wir nun *Toronto, CA* einfügen.

```
INSERT INTO cities
VALUES ('Toronto','M4C1B5','ca');

ERROR:  insert or update on table "cities" violates foreign key constraint
  "cities_country_code_fkey"
DETAIL:  Key (country_code)=(ca) is not present in table "countries".
```

Dieser Fehler ist gut! Da `country_code` `countries` referenziert, muss der `country_code` in der `countries`-Tabelle vorhanden sein. Man nennt das *Pflege der referenziellen Integrität*. Sie stellt sicher, dass unsere Daten immer korrekt sind (siehe Abbildung 1, *Das Schlüsselwort REFERENCES beschränkt Felder auf den Primärschlüssel einer anderen Tabelle.*, auf Seite 16). Beachten Sie, dass `NULL` ein gültiger Wert für `cities.country_code` ist, da die `NULL` das Fehlen eines Wertes repräsentiert. Wenn Sie die Spalte in der `cities`-Tabelle wie folgt definieren: `country_code char(2) REFERENCES countries NOT NULL`.

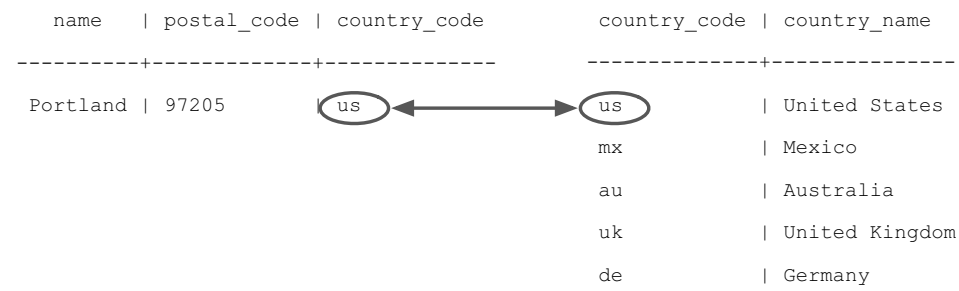


Abbildung 1: Das Schlüsselwort `REFERENCES` **beschränkt Felder auf den Primärschlüssel einer anderen Tabelle.**

Nun wollen wir es nochmal probieren, diesmal mit einer amerikanischen Stadt.

```
INSERT INTO cities
VALUES ('Portland','87200','us');

INSERT 0 1
```

Ok, dieser Insert war wohl erfolgreich. Doch wir haben versehentlich den falschen `postal_code` eingegeben. Die korrekte Postleitzahl für Portland lautet *97205*. Statt die Zeile zu löschen und wieder neu einzufügen, können wir sie aktualisieren.

```
UPDATE cities
SET postal_code = '97205'
WHERE name = 'Portland';
```

Wir haben nun die Zeilen einer Tabelle erzeugt (Create), gelesen (Read), aktualisiert (Update) und gelöscht (Delete).

Join-Reads

Alle anderen Datenbanken, über die wir in diesem Buch reden, führen ebenfalls CRUD-Operationen durch. Was relationale Datenbanken wie PostgreSQL von anderen abhebt, ist ihre Fähigkeit, Tabellen während des Lesens zusammenzufassen. Bei einer Join-Operation werden zwei Tabellen in irgendeiner Form miteinander kombiniert und in einer einzelnen Tabelle zurückgegeben. Das ist ein wenig so wie beim Scrabble, wenn man Teile existierender Wörter nutzt, um neue Wörter zu schaffen.

Die grundlegende Variante eines Joins ist der sog. *innere Join*. In der einfachsten Form legen Sie zwei Spalten (eine aus jeder Tabelle), die miteinander verglichen werden sollen, über das Schlüsselwort `ON` fest.

```
SELECT cities.*, country_name
FROM cities INNER JOIN countries
  ON cities.country_code = countries.country_code;
```

```
 country_code |   name   | postal_code | country_name
--------------+----------+-------------+--------------
 us           | Portland | 97205       | United States
```

Der Join liefert eine einzelne Tabelle zurück, die alle Spaltenwerte der `cities`-Tabelle zusammen mit dem passenden `country_name`-Wert aus der `countries`-Tabelle enthält.

Wir können Joins auch auf Tabellen wie `cities` anwenden, die einen zusammengesetzten Primärschlüssel nutzen. Um einen zusammengesetzten Join zu testen, wollen wir eine neue Tabelle mit einer Liste von Veranstaltungsorten aufbauen.

Ein Veranstaltungsort existiert sowohl unter einer bestimmten Postleitzahl (*postal_code*) als auch in einem bestimmten Land (*country*). Der *Fremdschlüssel* muss aus zwei Spalten bestehen, der beide `cities`-*primary key*-Spalten berücksichtigt. (`MATCH FULL` ist ein Constraint, der sicherstellt, dass beide Werte existieren oder beide `NULL` sind.)

```
CREATE TABLE venues (
  venue_id SERIAL PRIMARY KEY,
  name varchar(255),
  street_address text,
  type char(7) CHECK ( type in ('public','private') ) DEFAULT 'public',
  postal_code varchar(9),
  country_code char(2),
  FOREIGN KEY (country_code, postal_code)
    REFERENCES cities (country_code, postal_code) MATCH FULL
);
```

Die `venue_id`-Spalte ist ein typischer Primärschlüssel: automatisch inkrementierte Integerwerte (1, 2, 3, 4 und so weiter...). Wir erzeugen diesen Identifier mit dem Schlüsselwort `SERIAL` (MySQL kennt ein vergleichbares Konstrukt namens `AUTO_INCREMENT`).

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Crystal Ballroom', '97205', 'us');
```

Obwohl wir keinen `venue_id`-Wert angegeben haben, wird er beim Anlegen der Zeile automatisch erzeugt.

Zurück zu unserem zusammengesetzten Join. Der Join der venues-Tabelle mit der cities-Tabelle verlangt *beide* Fremdschlüssel. Um uns etwas Schreibarbeit zu ersparen, können wir für den Tabellennamen einen Alias festlegen, indem wir ihn direkt auf den Tabellennamen folgen lassen, wobei wir optional ein AS dazwischen setzen können (also beispielsweise venues v oder venues AS v).

```
SELECT v.venue_id, v.name, c.name
FROM venues v INNER JOIN cities c
  ON v.postal_code=c.postal_code AND v.country_code=c.country_code;

 venue_id |       name       |   name
----------+------------------+----------
        1 | Crystal Ballroom | Portland
```

Optional können Sie bei PostgreSQL die Spalten festlegen, die nach einer Insert-Operation zurückgeliefert werden sollen, indem Sie die Query mit einer RETURNING-Anweisung abschließen.

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Voodoo Donuts', '97205', 'us') RETURNING venue_id;

 id
- - - -
  2
```

Das liefert ohne eine zusätzliche Abfrage die neue venue_id zurück.

Äußere Joins

Neben inneren Joins kann PostgreSQL auch sog. *äußere Joins (outer joins)* durchführen. Äußere Joins sind eine Möglichkeit, zwei Tabellen zu verschmelzen, wenn das Ergebnis einer Tabelle immer zurückgegeben werden muss, unabhängig davon, ob es einen passenden Spaltenwert in der anderen Tabelle gibt.

Das erklärt man am einfachsten mit einem Beispiel, doch zu diesem Zweck müssen wir eine neue Tabelle namens events anlegen. Das ist genau die richtige Übung für Sie. Die events-Tabelle soll die folgenden Spalten besitzen: eine fortlaufende (SERIAL) ganzzahlige event_id, einen Titel (title), Anfang (starts) und Ende (ends) (vom Typ *timestamp*) und eine venue_id (ein venues referenzierender Fremdschlüssel). Ein Schema-Definitionsdiagramm aller bisher behandelten Tabellen sehen Sie in Abbildung 2, *Das Entity-Relationship-Diagramm (ERD)*, auf Seite 19.

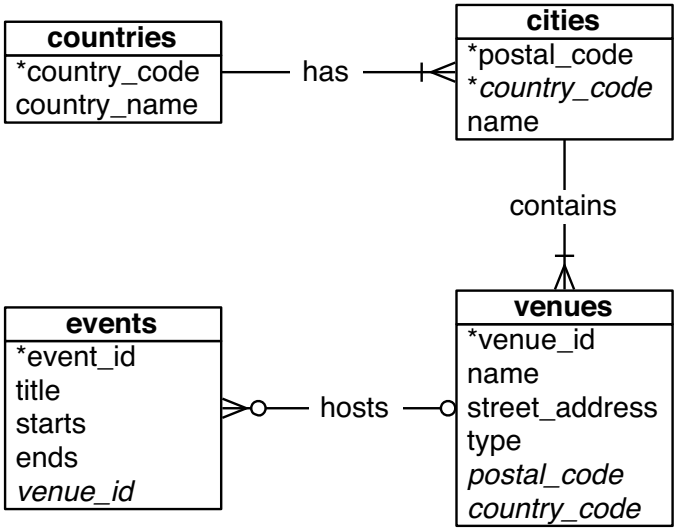


Abbildung 2: Das Entity-Relationship-Diagramm (ERD)

Nachdem Sie die `events`-Tabelle angelegt haben, fügen Sie mittels `INSERT` die folgenden Werte ein (Zeitstempel werden als Strings der Form *2012-02-15 17:30* angegeben): Urlaubstage und einen Club, über den wir *nicht reden*.

```
      title     |       starts        |        ends         | venue_id | event_id
----------------+---------------------+---------------------+----------+---------
LARP Club       | 2012-02-15 17:30:00 | 2012-02-15 19:30:00 |        2 |        1
April Fools Day | 2012-04-01 00:00:00 | 2012-04-01 23:59:00 |          |        2
Christmas Day   | 2012-12-25 00:00:00 | 2012-12-25 23:59:00 |          |        3
```

Zuerst wollen wir eine Abfrage formulieren, die uns den Titel eines Events und den Namen des Veranstaltungsorts als inneren Join zurückliefert (das Wort `INNER` von `INNER JOIN` ist optional, weshalb wir es hier weglassen).

```
SELECT e.title, v.name
FROM events e JOIN venues v
  ON e.venue_id = v.venue_id;

       title  |       name
--------------+------------------
 LARP Club    | Voodoo Donuts
```

`INNER JOIN` liefert nur dann eine Zeile zurück, *wenn die Spaltenwerte übereinstimmen*. Da es keine `venues.venue_ids` mit dem Wert `Null` geben kann, liefern die beiden `NULL-eventsvenue_ids` nichts zurück. Wenn Sie unabhängig von einem Veranstaltungsort alle Events abrufen wollen, müssen Sie einen `LEFT OUTER JOIN` (kurz `LEFT JOIN`) verwenden.

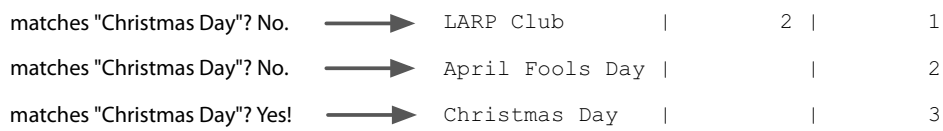
```
SELECT e.title, v.name
FROM events e
LEFT JOIN venues v
ON e.venue_id = v.venue_id;
```

```
      title      |      name
-----------------+----------------
 LARP Club       | Voodoo Donuts
 April Fools Day |
 Christmas Day   |
```

Wenn Sie das Gegenteil brauchen, alle Veranstaltungsorte und nur passende Events, verwenden Sie einen `RIGHT JOIN`. Schließlich gibt es noch den `FULL JOIN`, also die Vereinigung von `LEFT` und `RIGHT`, bei der Sie garantiert alle Werte aus beiden Tabellen zurückerhalten.

Schnelle Lookups per Indexierung

Die Geschwindigkeit von PostgreSQL (und anderen RDBMS) basiert auf der effizienten Verwaltung von Datenblöcken, der Reduzierung von Leseoperationen, der Query-Optimierung und anderen Techniken. Doch sie rufen Ergebnisse nur schnell ab. Wenn wir einenTitel wie *Christmas Day* aus der `events`-Tabelle abrufen, muss der Algorithmus jede Zeile auf einen Treffer untersuchen. Ohne einen *Index* muss jede Zeile von der Platte gelesen werden, um herauszufinden, ob die Query sie zurückgeben muss.



Ein Index ist eine spezielle Datenstruktur, die aufgebaut wird, um bei Abfragen ein vollständiges Durchsuchen der Tabelle zu vermeiden. Wenn Sie `CREATE TABLE`-Befehle ausgeführt haben, könnten Ihnen Meldungen wie die folgende aufgefallen sein:

```
CREATE TABLE / PRIMARY KEY will create implicit index "events_pkey" \
for table "events"
```

PostgreSQL erzeugt automatisch einen Index über den Primärschlüssel. Dabei ist der Primärschlüssel der Schlüssel und der Wert zeigt auf die entsprechende Zeile auf der Festplatte, wie in der Grafik unten zu sehen. Die Verwendung des Schlüsselwortes `UNIQUE` ist eine weitere Möglichkeit, einen Index für eine Tabellenspalte zu erzwingen.

Sie können explizit einen Index mit dem Befehl `CREATE INDEX` erzeugen. Dabei muss jeder Wert eindeutig sein (wie eine Hashtabelle oder eine Map).

```
CREATE INDEX events_title
  ON events USING hash (title);
```

Für Kleiner-als/Größer-als/Gleich-Vergleiche wollen wir einen Index, der etwas flexibler ist als ein einfacher Hash, etwa ein B-Tree (siehe Abbildung 3, *Ein B-Tree-Index eignet sich für bereichsorientierte Queries.*, auf Seite 22). Stellen Sie sich eine Query vor, die alle Events nach dem ersten April finden soll.

```
SELECT *
FROM events
WHERE starts >= '2012-04-01';
```

Dafür ist ein Baum die perfekte Datenstruktur. Um einen B-Tree-Index für die `starts`-Spalte anzulegen, verwenden wir Folgendes:

```
CREATE INDEX events_starts
  ON events USING btree (starts);
```

Nun können wir eine bereichsorientierte Abfrage über die Datumsangaben durchführen, ohne die ganze Tabelle durchsuchen zu müssen. Das macht bei Millionen (oder Milliarden) von Datensätzen einen gewaltigen Unterschied.

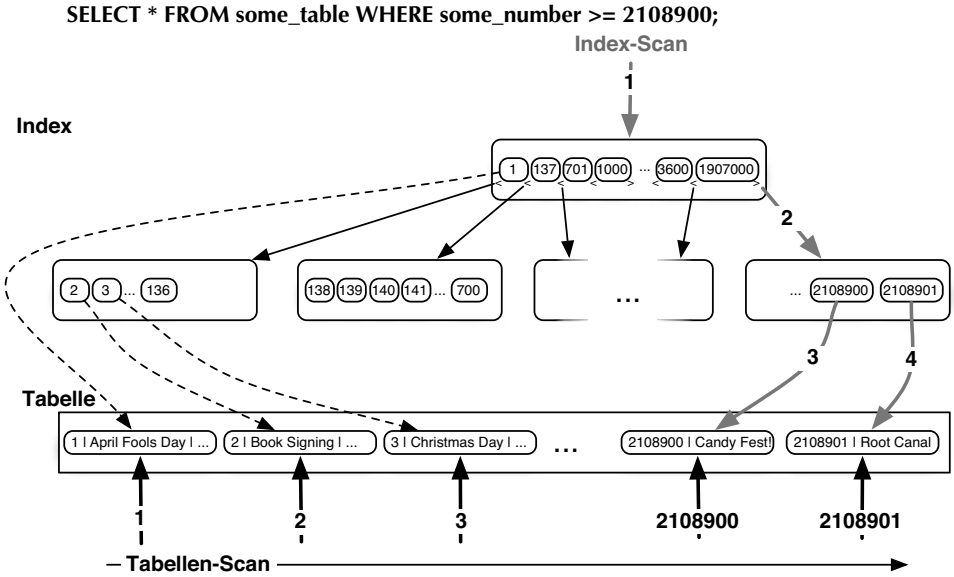


Abbildung 3: Ein B-Tree-Index eignet sich für bereichsorientierte Queries.

Wir können unsere Arbeit mit dem folgenden Befehl inspizieren, der alle Indizes des Schemas ausgibt:

```
book=# \di
```

Beachten Sie, dass PostgreSQL bei einer `FOREIGN KEY`-Constraint automatisch einen Index für die Zielspalte(n) erzeugt. Selbst wenn Sie Datenbank-Constraints nicht mögen (ja, wir sprechen von euch, Ruby on Rails-Entwickler), werden Sie sich häufig dabei ertappen, Indizes für Spalten anzulegen, über die später ein Join laufen soll, um Fremdschlüssel-Joins zu beschleunigen.

Was wir am ersten Tag gelernt haben

Wir sind heute einiges durchgegangen und haben viele Begriffe kennengelernt. Hier eine Zusammenfassung:

Begriff	Definition
Spalte (Column)	Eine Domäne mit Werten eines bestimmten Typs. Wird manchmal auch *Attribut* genannt
Zeile (Row)	Ein aus einer Reihe von Spaltenwerten zusammengesetztes Objekt. Wird manchmal auch *Tupel* genannt
Tabelle (Table)	Eine Reihe von Zeilen mit den gleichen Spalten. Wird manchmal auch *Relation* genannt
Primärschlüssel (Primary Key)	Der eindeutige Wert, der eine bestimmte Zeile identifiziert
CRUD	Create, Read, Update, Delete, also Anlegen, Lesen, Aktualisieren und Löschen
SQL	Structured Query Language, die *lingua franca* relationaler Datenbanken
Join	Zwei Tabellen an übereinstimmenden Spalten zu einer zusammenfassen
Left join	Zwei Tabellen an übereinstimmenden Spalten zu einer zusammenfassen, oder mit `NULL`, wenn es in der linken Tabelle keine Übereinstimmung gibt
Index	Eine Datenstruktur zur optimierten Auswahl einer bestimmten Menge von Spalten
B-Tree	Ein guter Standard-Index. Werte werden in einem ausbalancierten Baum gespeichert. Sehr flexibel

Relationale Datenbanken sind seit 40 Jahren der *de-facto*-Standard des Datenmanagements. Viele von uns haben ihre Karriere in der Mitte ihrer Evolution begonnen. Wir haben uns einige Kernkonzepte des relationalen Modells mithilfe elementarer SQL-Queries angesehen. Wir werden am 2. Tag auf diesen Kernkonzepten aufbauen.

Tag 1: Selbststudium

Finden Sie heraus

1. Legen Sie Bookmarks für das Online-PostgreSQL-FAQ und die Dokumente an.
2. Machen Sie sich mit den Kommandozeilen-Ausgaben von `\?` und `\h` vertraut.
3. Finden Sie in der Dokumentation unter `FOREIGN KEY` heraus, was `MATCH FULL` bedeutet.

Machen Sie Folgendes

1. Wählen Sie alle Tabellen aus `pg_class` aus, die wir angelegt haben (und zwar nur diese).
2. Schreiben Sie eine Query, die den Namen des Landes des LARP Club-Events ermittelt.
3. Erweitern Sie die `venues`-Tabelle um eine Boolesche Spalte namens `active` mit dem Standardwert `TRUE`.

2.3 Tag 2: Fortgeschrittene Queries, Code und Regeln

Am ersten Tag haben wir gesehen, wie man Schemata definiert, mit Daten füllt, Zeilen aktualisiert und löscht sowie grundlegende Leseoperationen durchführt. Heute wollen wir tiefer in die vielfältigen Möglichkeiten eintauchen, mit denen PostgreSQL Daten abfragen kann. Wir werden zeigen, wie man gleichartige Daten gruppiert, Code auf dem Server ausführt und eigene Interfaces über *Views* und *Regeln* aufbaut. Wir beenden den Tag mit einem PostgreSQL-Zusatzpaket, das die Tabellen auf den Kopf stellen kann.

Aggregatfunktionen

Eine Aggregat-Query gruppiert die Ergebnisse mehrerer Zeilen nach einem gemeinsamen Kriterium. Das können so einfache Dinge sein wie das Zählen der Zeilen einer Tabelle oder die Berechnung des Durchschnitts einer numerischen Spalte. Sie sind mächtige SQL-Werkzeuge und machen viel Spaß.

Wir wollen ein paar Aggregatfunktionen ausprobieren, brauchen aber zuerst etwas mehr Daten in unserer Datenbank. Fügen Sie Ihr eigenes Land in die `countries`-Tabelle ein, Ihre eigene Stadt in die `cities`-Tabelle und Ihre eigene Adresse als Veranstaltungsort (den wir einfach *My Place* nennen). Fügen Sie dann einige Datensätze in die `events`-Tabelle ein.

Ein SQL-Tipp am Rande: Statt die venue_id explizit anzugeben, können Sie mit einem Sub-SELECT besser lesbaren Titel verwenden. Wenn also *Moby* im *Crystal Ballroom* spielt, können Sie die venue_id wie folgt setzen:

```
INSERT INTO events (title, starts, ends, venue_id)
  VALUES ('Moby', '2012-02-06 21:00', '2012-02-06 23:00', (
    SELECT venue_id
    FROM venues
    WHERE name = 'Crystal Ballroom'
  )
);
```

Füllen Sie die events-Tabelle mit den folgenden Daten auf (um *Valentine's Day* in PostgreSQL einzugeben, ersetzen Sie das Apostroph durch zwei, also beispielsweise *Heaven''s Gate*):

```
      title      |       starts        |        ends         |      venue
-----------------+---------------------+---------------------+--------------
 Wedding         | 2012-02-26 21:00:00 | 2012-02-26 23:00:00 | Voodoo Donuts
 Dinner with Mom | 2012-02-26 18:00:00 | 2012-02-26 20:30:00 | My Place
 Valentine's Day | 2012-02-14 00:00:00 | 2012-02-14 23:59:00 |
```

Nachdem wir unsere Daten eingefügt haben, wollen wir einige Aggregat-Queries ausprobieren. Die einfachste Aggregatfunktion ist count(). Wenn Sie die Titel zählen (engl. count), die das Wort *Day* enthalten (Hinweis: % ist ein Platzhalter für LIKE-Suchen), wird der Wert 3 zurückgegeben.

```
SELECT count(title)
FROM events
WHERE title LIKE '%Day%';
```

Um die Anfangszeit des ersten und die Endzeit des letzten Events im Crystal Ballroom zu ermitteln, nutzen Sie min() (gibt den kleinsten Wert zurück) und max() (gibt den größten Wert zurück).

```
SELECT min(starts), max(ends)
FROM events INNER JOIN venues
  ON events.venue_id = venues.venue_id
WHERE venues.name = 'Crystal Ballroom';

         min         |         max
---------------------+---------------------
 2012-02-06 21:00:00 | 2012-02-06 23:00:00
```

Aggregatfunktionen sind nützlich, für sich genommen aber nur eingeschränkt sinnvoll. Wenn wir die Anzahl der Events an jedem Veranstaltungsort zählen wollen, können wir für jede venue-ID Folgendes schreiben:

```
SELECT count(*) FROM events WHERE venue_id = 1;
SELECT count(*) FROM events WHERE venue_id = 2;
SELECT count(*) FROM events WHERE venue_id = 3;
SELECT count(*) FROM events WHERE venue_id IS NULL;
```

Das wäre sehr lästig, wenn die Zahl der Veranstaltungsorte wächst. Daher nutzen wir den `GROUP BY`-Befehl.

Gruppierung

`GROUP BY` ist ein Kürzel, das die obigen Queries auf einmal ausführt. Mit `GROUP BY` weisen Sie Postgres an, die Zeilen zu gruppieren und dann eine Aggregatfunktion (wie `count()`) auf diese Gruppen anzuwenden.

```
SELECT venue_id, count(*)
FROM events
GROUP BY venue_id;
```

```
 venue_id | count
----------+-------
        1 |     1
        2 |     2
        3 |     1
          |     3
```

Das ist eine nette Liste, aber die Frage ist, ob wir sie über die `count()`-Funktion filtern können. Die Antwort lautet ja. Die `GROUP BY`-Bedingung besitzt ein eigenes Filter-Schlüsselwort: `HAVING`. `HAVING` verhält sich wie die `WHERE`-Klausel, nur dass es über Aggregatfunktionen filtert (was `WHERE` nicht kann).

Die folgende Query liefert die beliebtesten Veranstaltungsorte zurück, d. h. diejenigen mit zwei oder mehr Events:

```
SELECT venue_id
FROM events
GROUP BY venue_id
HAVING count(*) >= 2 AND venue_id IS NOT NULL;
```

```
 venue_id | count
----------+-------
        2 |     2
```

Sie können `GROUP BY` auch ohne Aggregatfunktion nutzen. Wenn Sie `SELECT... FROM...GROUP BY` für eine Spalte ausführen, erhalten Sie eine Liste aller eindeutigen Werte zurück.

```
SELECT venue_id FROM events GROUP BY venue_id;
```

Diese Art der Gruppierung ist so gängig, dass SQL dafür mit `DISTINCT` ein eigenes Schlüsselwort kennt.

```
SELECT DISTINCT venue_id FROM events;
```

Die Ergebnisse beider Queries sind identisch.

GROUP BY in MySQL

Wenn Sie bei MySQL versuchen, ein `SELECT` mit Spalten auszuführen, die nicht unter `GROUP BY` definiert sind, wird es Sie vielleicht schockieren zu sehen, dass es funktioniert. Das ließ uns ursprünglich an der Notwendigkeit von Fensterfunktionen zweifeln. Doch als wir die Daten genauer untersuchten, die MySQL zurücklieferte, stellte sich heraus, dass nur eine zufällige Auswahl von Datensätzen zurückgegeben wurde, nicht alle relevanten Ergebnisse. Das ist nicht besonders sinnvoll (und sogar potentiell gefährlich).

Window-Funktionen

Wenn Sie in der Vergangenheit irgendeine Art Produktivsystem mit relationalen Datenbanken entwickelt haben, sind Sie mit Aggregatfunktionen sehr wahrscheinlich vertraut. Sie sind ein gängiges SQL-Werkzeug. *Fensterfunktionen* (window functions) sind dagegen nicht so weit verbreitet (PostgreSQL ist eine der wenigen Open-Source-Datenbanken, die sie implementiert).

Fensterfunktionen ähneln `GROUP BY`-Queries darin, dass sie es erlauben, Aggregatfunktionen auf mehrere Spalten anzuwenden. Der Unterschied besteht darin, dass sie es erlauben, fest eingebaute Aggregatfunktionen zu nutzen, ohne dass jedes einzelne Feld in eine einzelne Zeile gruppiert werden muss.

Wenn Sie versuchen, die `title`-Spalte auszuwählen, ohne sie zu gruppieren, müssen Sie mit einer Fehlermeldung rechnen.

```
SELECT title, venue_id, count(*)
FROM events
GROUP BY venue_id;

ERROR:  column "events.title" must appear in the GROUP BY clause or \
        be used in an aggregate function
```

Wir zählen die Zeilen nach `venue_id` und im Falle von *LARP Club* und *Wedding* gibt es zwei Titel für eine `venue_id`. Postgres weiß nicht, *welcher* Titel ausgegeben werden soll.

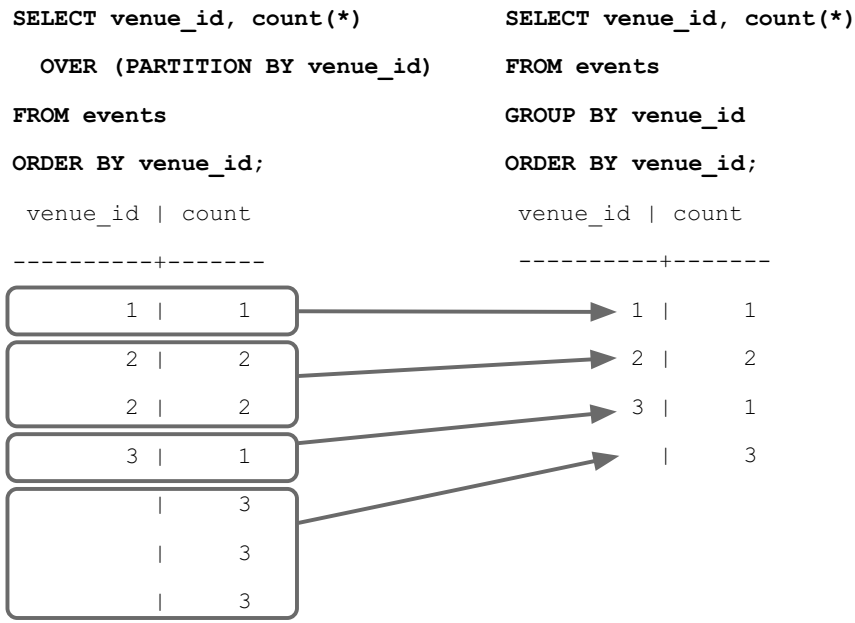


Abbildung 4: Die Ergebnisse der Fensterfunktionen lassen die Ergebnisse pro Gruppe nicht einbrechen.

Während eine `GROUP BY`-Klausel einen Datensatz für jeden passenden Gruppenwert zurückgibt, kann eine Fensterfunktion einen separaten Datensatz für jede Zeile zurückliefern. Eine visuelle Darstellung sehen Sie in Abbildung 4, *Die Ergebnisse der Fensterfunktionen lassen die Ergebnisse pro Gruppe nicht einbrechen.*, auf Seite 28. Sehen wir uns an, was Fensterfunktionen leisten können.

Fensterfunktionen geben alle Treffer zurück und replizieren die Ergebnisse jeder Aggregatfunktion.

```
SELECT title, count(*) OVER (PARTITION BY venue_id) FROM events;
```

Wir betrachten `PARTITION BY` gerne als einen Verwandten von `GROUP BY`, doch statt die Ergebnisse aus der `SELECT`-Attributliste zu erzeugen (und damit zu einer kleineren Ergebnisliste zusammenzufassen), gibt sie die gruppierten Werte wie jedes andere Feld zurück (die Berechnung erfolgt für die gruppierte Variable, ist ansonsten aber einfach ein weiteres Attribut). Oder, im SQL-

Sprachgebrauch, gibt sie das Ergebnis einer Aggregatfunktion über (`OVER`) einen Teil (ein Fenster, `PARTITION`) der Ergebnismenge zurück.

Transaktionen

Transaktionen sind das Konsistenz-Bollwerk relationaler Datenbanken. *Alles oder nichts* lautet das Motto bei Transaktionen. Sie stellen sicher, dass jeder Befehl in einer Folge von Befehlen auch ausgeführt wird. Geht dabei irgendetwas schief, wird alles wieder so zurückgedreht (Rollback), als wäre nie etwas passiert.

PostgreSQL-Transaktionen sind ACID-konform. Das steht für atomisch (Atomic, entweder alle Operationen sind erfolgreich oder keine), konsistent (Consistent, die Daten sind in einem „guten" Zustand – es gibt keine inkonsistenten Zustände), isoliert (Isolated, Transaktionen behindern sich nicht gegenseitig) und beständig (Durable, eine bestätigte Transaktion ist sicher, selbst nach einem Server-Absturz). Bitte beachten Sie, dass *Konsistenz* bei ACID eine andere Bedeutung hat als bei CAP (das in Anhang 2, *Das CAP-Theorem*, auf Seite 347 behandelt wird).

Wir können jede Transaktion in einen `BEGIN TRANSACTION`-Block packen. Um die Atomizität zu prüfen, heben wir die Transaktion mit dem `ROLLBACK`-Befehl wieder auf.

Zwangsläufige Transaktionen

Bis jetzt wurde jeder von uns in `psql` ausgeführte Befehl implizit in eine Transaktion gepackt. Wenn Sie einen Befehl wie `DELETE FROM account WHERE total < 20;` ausführen und die Datenbank stürzt auf halber Strecke ab, dann bleiben Sie nicht auf einer halb gelöschten Tabelle sitzen. Beim Neustart des Datenbankservers erfolgt ein Rollback des Befehls.

```
BEGIN TRANSACTION;
  DELETE FROM events;
ROLLBACK;
SELECT * FROM events;
```

Alle Events bleiben erhalten. Transaktionen sind nützlich, wenn Sie zwei Tabellen modifizieren, die synchron bleiben müssen. Das klassische Beispiel ist das Kontensystem einer Bank, bei dem Geld von einem Konto auf ein anderes transferiert wird:

```
BEGIN TRANSACTION;
  UPDATE account SET total=total+5000.0 WHERE account_id=1337;
  UPDATE account SET total=total-5000.0 WHERE account_id=45887;
END;
```

Wenn etwas zwischen den beiden Updates schiefgeht, hat die Bank fünf Riesen verloren. Doch innerhalb eines Transaktionsblocks wird das erste Update aufgehoben, selbst wenn der Server explodiert.

Stored Procedures

Alle Befehle, die Sie bisher gesehen haben, waren deklarativ, doch manchmal muss man auch Code ausführen. An diesem Punkt muss man eine Entscheidung treffen: Soll der Code auf Seiten des Clients oder auf der Datenbank ausgeführt werden?

Stored Procedures (also „gespeicherte Prozeduren“) bieten große Performance-Vorteile zum Preis hoher Architekturkosten. Sie können den Abruf tausender Datensätze an eine Client-Anwendung einsparen, haben Ihren Anwendungscode aber auch an diese Datenbank gebunden. Die Entscheidung für Stored Procedures sollte nicht leichtfertig getroffen werden.

Ungeachtet aller Warnungen wollen wir eine Prozedur (oder `FUNCTION`) entwickeln, die den `INSERT` eines neuen Events zu einem Veranstaltungsort erlaubt, ohne die `venue_id` kennen zu müssen. Wenn der Veranstaltungsort noch nicht existiert, wird er zuerst angelegt und dann im neuen Event angegeben. Wir wollen (als kleine Feinheit) einen Booleschen Wert zurückgeben, der dem Benutzer anzeigt, ob ein neuer Veranstaltungsort eingefügt wurde.

Was ist mit der Anbieterabhängigkeit?

Als relationale Datenbanken ihre Blütezeit hatten, waren sie das Schweizer Armeemesser der Technik. Man konnte nahezu alles in ihnen speichern – und sogar ganze Projekte mit ihnen entwickeln (z. B. mit Microsoft Access). Die wenigen Unternehmen, die diese Software anboten, förderten die Nutzung proprietärer Unterschiede und nutzten diese Abhängigkeit dann aus, um enorme Lizenz- und Beratungsgebühren zu verlangen. Das war die gefürchtete *Anbieterabhängigkeit*, die neuere Programmiermethoden in den 1990ern und frühen 2000ern zu lindern versuchten.

Doch in dem Bestreben, die Anbieter zu neutralisieren, entstanden Maximen wie *keine Logik in der Datenbank*. Das ist eine Schande, weil relationale Datenbanken über so viele verschiedene Datenbankmanagement-Optionen verfügen. Viele Aktionen, die wir in diesem Buch vorstellen, sind hochgradig implementierungspezifisch. Dennoch lohnt es sich zu wissen, wie man Datenbanken bis an die Grenzen ausreizt, statt Tools wie Stored Procedures *a priori* abzulehnen.

postgres/add_event.sql

```
CREATE OR REPLACE FUNCTION add_event( title text, starts timestamp,
  ends timestamp, venue text, postal varchar(9), country char(2) )
RETURNS boolean AS $$
DECLARE
  did_insert boolean := false;
  found_count integer;
  the_venue_id integer;
BEGIN
  SELECT venue_id INTO the_venue_id
  FROM venues v
  WHERE v.postal_code=postal AND v.country_code=country AND v.name ILIKE venue
  LIMIT 1;

  IF the_venue_id IS NULL THEN
    INSERT INTO venues (name, postal_code, country_code)
    VALUES (venue, postal, country)
    RETURNING venue_id INTO the_venue_id;

    did_insert := true;
  END IF;

  -- Hinweis: kein „Fehler“ wie bei einigen Programmiersprachen
  RAISE NOTICE 'Venue found %', the_venue_id;

  INSERT INTO events (title, starts, ends, venue_id)
  VALUES (title, starts, ends, the_venue_id);

  RETURN did_insert;
END;
$$ LANGUAGE plpgsql;
```

Sie können diese externe Datei in das aktuelle Schema mit der folgenden Kommandozeile einfügen (wenn Sie nicht den ganzen Code abtippen wollen):

```
book=# \i add_event.sql
```

Wenn Sie ihn ausführen, sollte er `t` (true, also wahr) zurückgeben, da der Veranstaltungsort *Run's House* neu ist. Das ersetzt das Senden zweier SQL-Befehle (ein select gefolgt von einem insert) vom Client durch einen.

```
SELECT add_event('House Party', '2012-05-03 23:00',
  '2012-05-04 02:00', 'Run''s House', '97205', 'us');
```

Die Sprache, in der wir die Prozedur entwickelt haben, ist PL/pgSQL (was für Procedural Language/PostgreSQL steht). Die Details einer ganzen Programmiersprache abzuhandeln, würde unseren Rahmen sprengen, aber Sie können alles darüber in der PostgreSQL-Onlinedokumentation nachlesen.[3]

Neben PL/pgSQL unterstützt Postgres drei weitere Kernsprachen für die Entwicklung von Prozeduren: Tcl, Perl und Python. Für dutzende anderer Spra-

3. http://www.postgresql.org/docs/9.0/static/plpgsql.html

chen, darunter Ruby, Java, PHP und Scheme wurden Erweiterungen geschrieben. Eine Liste weiterer Sprachen finden Sie in der Dokumentation. Geben Sie den folgenden Shell-Befehl ein:

```
$ createlang book --list
```

Er führt alle für Ihre Datenbank installierten Sprachen auf. Der Befehl `createlang` wird außerdem genutzt, um neue Sprachen hinzuzufügen. Mehr darüber erfahren Sie online.[4]

Trigger

Trigger führen Stored Procedures automatisch aus, wenn ein bestimmtes Ereignis eintritt, etwa ein Insert oder ein Update. Sie erlauben es der Datenbank, eine bestimmte Reaktion auf geänderte Daten zu erzwingen.

Lassen Sie uns eine neue PL/pgSQL-Funktion entwickeln, die in einem Log festhält, wenn ein Event aktualisiert wird (wir wollen sicherstellen, dass niemand ein Event ändern kann und es dann später abstreitet). Zuerst legen wir eine `logs`-Tabelle an, in der wir die Event-Änderungen festhalten. Ein Primärschlüssel ist hier nicht nötig, da es sich bloß um einen Log handelt.

Ausführung in der Datenbank?

An dieser Stelle stehen wir das erste Mal vor einer Frage, die sich im Buch immer wieder mal stellt: Gehört der Code in die Anwendung oder in die Datenbank? Das ist eine schwierige Frage, die Sie für jede Anwendung neu beantworten müssen.

Der Vorteil ist, dass man die Performance um etwa eine Größenordnung verbessern kann. Sie könnten zum Beispiel eine komplexe anwendungsspezifische Berechnung durchführen müssen. Wenn die Berechnung viele Datensätze umfasst, müssen Sie mit Stored Procedures nur ein einzelnes Ergebnis übertragen statt tausende von Datensätzen. Der Preis ist die Aufteilung Ihrer Anwendung, Ihres Codes und Ihrer Tests auf zwei unterschiedliche Programmierparadigmen.

```
CREATE TABLE logs (
  event_id integer,
  old_title varchar(255),
  old_starts timestamp,
  old_ends timestamp,
  logged_at timestamp DEFAULT current_timestamp
);
```

4. http://www.postgresql.org/docs/9.0/static/app-createlang.html

Als Nächstes bauen wir eine Funktion auf, die die alten Daten im Log festhält. Die Variable `OLD` enthält die zu ändernde Zeile (und `NEW` eine eingehende Zeile, was wir später noch in Aktion sehen). Bevor wir zurückspringen, geben wir noch einen Hinweis mit der `event_id` in der Console aus.

postgres/log_event.sql

```
CREATE OR REPLACE FUNCTION log_event() RETURNS trigger AS $$
DECLARE
BEGIN
  INSERT INTO logs (event_id, old_title, old_starts, old_ends)
  VALUES (OLD.event_id, OLD.title, OLD.starts, OLD.ends);
  RAISE NOTICE 'Someone just changed event #%', OLD.event_id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Abschließend legen wir der Trigger an, der die Änderungen festhält, nachdem eine Zeile aktualisiert wurde.

```
CREATE TRIGGER log_events
  AFTER UPDATE ON events
  FOR EACH ROW EXECUTE PROCEDURE log_event();
```

Wie es aussieht, endet unsere Party im Run's House früher als erhofft. Lassen Sie uns das Event entsprechend ändern.

```
UPDATE events
SET ends='2012-05-04 01:00:00'
WHERE title='House Party';

NOTICE:  Someone just changed event #9
```

Und die alte Endzeit wurde im Log festgehalten.

```
SELECT event_id, old_title, old_ends, logged_at
FROM logs;

event_id |  old_title  |      old_ends       |        logged_at
---------+-------------+---------------------+------------------------
       9 | House Party | 2012-05-04 02:00:00 | 2011-02-26 15:50:31.939
```

Trigger können auch vor Updates und vor oder nach Inserts eingerichtet werden.[5]

5. http://www.postgresql.org/docs/9.0/static/triggers.html

Views

Wäre es nicht schön, wenn wir das Ergebnis einer komplexen Query so nutzen könnten wie jede andere Tabelle? Nun, genau dafür sind VIEWs gedacht. Im Gegensatz zu Stored Procedures handelt es sich dabei nicht um auszuführende Funktionen, sondern um Aliase für Queries.

In unserer Datenbank enthalten alle Feiertage das Wort *Day* und es gibt keinen Veranstaltungsort.

postgres/holiday_view_1.sql

```
CREATE VIEW holidays AS
  SELECT event_id AS holiday_id, title AS name, starts AS date
  FROM events
  WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

Um also einen View zu erzeugen, müssen Sie nur eine Query schreiben und ihr ein `CREATE VIEW view_name AS` voranstellen. Nun können Sie `holidays` wie jede andere Tabelle abfragen. Hinter den Kulissen werkelt die gute alte `events`-Tabelle. Zum Beweis fügen Sie den *Valentinstag* am *14.2.2012* in `events` ein und fragen dann den `holidays`-View ab.

```
SELECT name, to_char(date, 'Month DD, YYYY') AS date
FROM holidays
WHERE date <= '2012-04-01';
```

```
      name       |        date
-----------------+--------------------
 April Fools Day | April     01, 2012
 Valentine's Day | February  14, 2012
```

Views sind ein mächtiges Werkzeug, um auf einfache Weise auf komplex abgefragte Daten zugreifen zu können. Die dahinterstehende Query kann noch so kompliziert sein, Sie sehen nur eine Tabelle.

Soll eine neue Spalte in den View eingefügt werden, muss das in der zugrundeliegenden Tabelle geschehen. Lassen Sie uns die `events`-Tabelle um ein Array mit assoziierten Farben erweitern.

```
ALTER TABLE events
ADD colors text ARRAY;
```

Da die Farben auch in `holidays` auftauchen sollen, müssen wir die `VIEW`-Query um das `colors`-Array ergänzen.

```
CREATE OR REPLACE VIEW holidays AS
  SELECT event_id AS holiday_id, title AS name, starts AS date, colors
  FROM events
  WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

Nun müssen wir noch ein Array mit Farbangaben zum gewünschten Feiertag hinzufügen. Leider können wir einen View nicht direkt aktualisieren.

```
UPDATE holidays SET colors = '{"red","green"}' where name = 'Christmas Day';

ERROR:  cannot update a view
HINT:  You need an unconditional ON UPDATE DO INSTEAD rule.
```

Sie so aus, als würden wir eine Regel (RULE) benötigen.

Regeln (RULEs)

Eine Regel (RULE) beschreibt, wie der geparste *Query-Baum* modifiziert werden soll. Jedes Mal, wenn Postgres eine SQL-Anweisung ausführt, wandelt es diese Anweisung zuerst in einen Query-Baum (allgemein auch *abstrakter Syntax-Baum* genannt) um. Dieser Prozess wird als Parsing bezeichnet.

Operatoren und Werte werden in diesem Baum zu Zweigen und Blättern, und vor der Ausführung wird er durchgegangen und beschnitten und auf andere Weise bearbeitet. Dieser Baum kann optional über Postgres-Regeln ungeschrieben werden, bevor er an den Query-Planer weitergegeben wird (der den Baum so umschreibt, dass er optional ausgeführt wird), und er leitet diesen letzten Befehl dann zur Ausführung weiter. Siehe Abbildung 5, *Wie SQL bei PostgreSQL ausgeführt wird*, auf Seite 36. Mehr noch, ein VIEW wie holidays *ist* eine Regel.

Wir können das nachweisen, indem wir uns den Ausführungsplan des holidays-Views mit dem EXPLAIN-Befehl ansehen (*Filter* ist die WHERE-Klausen und *Output* die Spaltenliste).

```
EXPLAIN VERBOSE
  SELECT *
  FROM holidays;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on public.events  (cost=0.00..1.04 rows=1 width=57)
   Output: events.event_id, events.title, events.starts, events.colors
   Filter: ((events.venue_id IS NULL) AND ((events.title)::text ~~ '%Day%'::text))
```

Vergleichen Sie das über EXPLAIN VERBOSE mit der Query, die wir für unseren holidays-VIEW nutzen. Sie sind funktional identisch.

```
EXPLAIN VERBOSE
  SELECT event_id AS holiday_id,
    title AS name, starts AS date, colors
  FROM events
  WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on public.events  (cost=0.00..1.04 rows=1 width=57)
   Output: event_id, title, starts, colors
   Filter: ((events.venue_id IS NULL) AND ((events.title)::text ~~ '%Day%'::text))
```

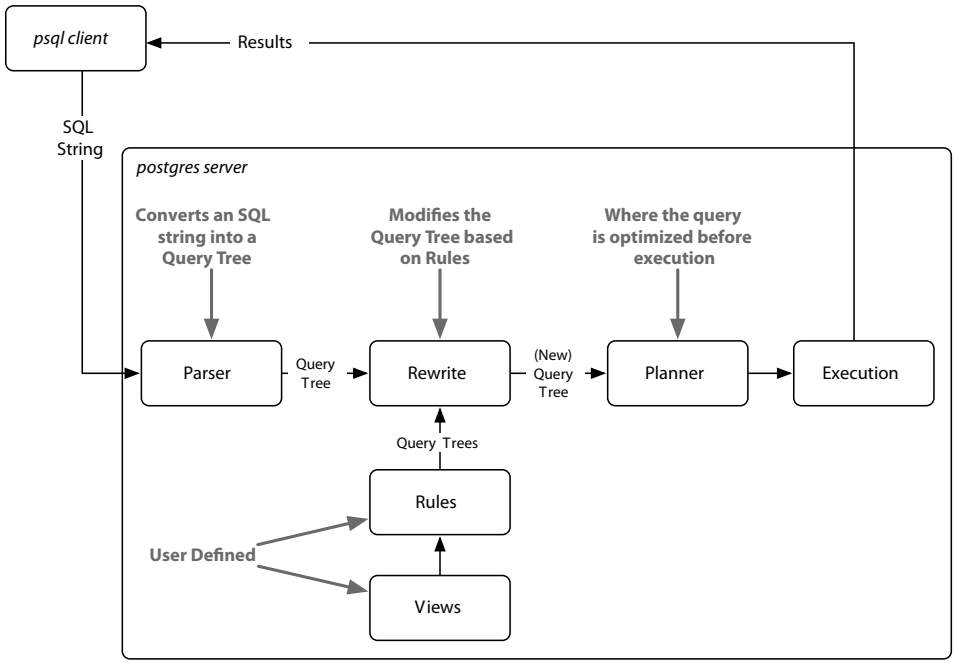


Abbildung 5: Wie SQL bei PostgreSQL ausgeführt wird

Um also Updates in unserem `holidays`-View vornehmen zu können, müssen wir eine Regel (`RULE`) entwickeln, die Postgres anweist, was bei einem `UPDATE` zu tun ist. Unsere Regel fängt Updates des `holidays`-Views ab und führt dieses Update dann über `events` aus, wobei es die Werte aus den Pseudorelationen `NEW` und `OLD` abruft. `NEW` dient funktional als Relation mit den von uns gesetzten Werten, während `OLD` die alten Werte enthält.

postgres/create_rule.sql

```
CREATE RULE update_holidays AS ON UPDATE TO holidays DO INSTEAD
  UPDATE events
  SET title = NEW.name,
      starts = NEW.date,
      colors = NEW.colors
  WHERE title = OLD.name;
```

Sobald diese Regel vorhanden ist, können wir `holidays` direkt aktualisieren.

```
UPDATE holidays SET colors = '{"red","green"}' where name = 'Christmas Day';
```

Als Nächstes wollen wir den *Neujahrstag* am *1.1.2013* in `holidays` einfügen. Wie erwartet, benötigen wir auch dafür eine Regel. Kein Problem.

```
CREATE RULE insert_holidays AS ON INSERT TO holidays DO INSTEAD
  INSERT INTO ...
```

Wir wenden uns jetzt anderen Dingen zu, aber wenn Sie mehr mit Regeln herumspielen wollen, versuchen Sie sich an einer `DELETE RULE`.

Kreuztabellen

Bei der letzten Übung für diesen Tag wollen wir einen Monatskalender mit Events aufbauen, bei dem jeder Monat des Kalenderjahrs die Anzahl der Events in diesem Monat angibt. Eine solche Operation wird üblicherweise mit einer *Pivot-Tabelle* gelöst. Sie bauen „Pivot"-gruppierte Daten um eine andere Ausgabe herum, in diesem Fall eine Liste der Monate. Wir bauen unsere Pivot-Tabelle mit der Funktion `crosstab()` auf.

Zuerst entwickeln wir eine Query, die für jedes Jahr die Anzahl der Events pro Monat ermittelt. PostgreSQL stellt die Funktion `extract()` zur Verfügung, die ein Unterfeld aus einem Datum oder Zeitstempel zurückliefert. Das hilft uns bei der Gruppierung.

```
SELECT extract(year from starts) as year,
  extract(month from starts) as month, count(*)
FROM events
GROUP BY year, month;
```

Um `crosstab()` nutzen zu können, muss die Query drei Spalten zurückliefern: `rowid`, `category` und `value`. Wir werden `year` als ID nutzen, d. h., die anderen Felder sind die Kategorie (der Monat) und der Wert (die Anzahl).

Die `crosstab`-Funktion benötigt eine weiteren Satz von Werten, der die Monate repräsentiert. So weiß die Funktion, wie viele Spalten benötigt werden. Diese Werte werden zu den Spalten (der Tabelle, über die wir die *Pivot*-Funktion laufen lassen). Wir legen daher eine temporäre Tabelle an, die eine Liste von Zahlen enthält.

```
CREATE TEMPORARY TABLE month_count(month INT);
INSERT INTO month_count VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12);
```

Nun können wir `crosstab()` mit unseren beiden Queries ausführen.

```
SELECT * FROM crosstab(
  'SELECT extract(year from starts) as year,
 extract(month from starts) as month, count(*)
 FROM events
 GROUP BY year, month',
  'SELECT * FROM month_count'
);

ERROR:  a column definition list is required for functions returning "record"
```

Ups, eine Fehlermeldung.

Das hört sich etwas kryptisch an, bedeutet aber, dass die Funktion eine Reihe von Datensätzen (Zeilen) zurückgibt und nicht weiß, wie sie sie benennen soll. Tatsächlich weiß sie nicht einmal, um welche Datentypen es sich handelt.

Denken Sie daran, dass die Pivot-Tabelle die Monate als Kategorie verwendet, aber diese Monate sind einfach nur Integerwerte. Daher definieren wir sie wie folgt:

```
SELECT * FROM crosstab(
  'SELECT extract(year from starts) as year,
 extract(month from starts) as month, count(*)
 FROM events
 GROUP BY year, month',
  'SELECT * FROM month_count'
) AS (
  year int,
  jan int, feb int, mar int, apr int, may int, jun int,
  jul int, aug int, sep int, oct int, nov int, dec int
) ORDER BY YEAR;
```

Wir haben eine Spalte für das Jahr (die Zeilen-ID) und zwölf weitere Spalten für die Monate.

```
year | jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
2012 |     |   5 |     |   1 |   1 |     |     |     |     |     |     |   1
```

Fügen Sie zusätzliche Events für ein anderes Jahr ein, um sich auch deren Anzahl ausgeben zu lassen. Führen Sie die crosstab-Funktion aus und genießen Sie Ihren Kalender.

Was wir am zweiten Tag gelernt haben

Heute haben wir die Grundlagen von PostgreSQL abgeschlossen. Wir beginnen zu erkennen, dass Postgres mehr ist, als einfach nur ein Server zum Speichern der üblichen Datentypen und ihre Abfrage. Es ist eine Datenbank-Management-Engine, mit der man die zurückgelieferten Daten umformatie-

ren, Datentypen wie Arrays speichern und Programmlogik ausführen kann. Und sie ist leistungsfähig genug, um eingehende Queries umzuschreiben.

Tag 2: Selbststudium

Finden Sie heraus

1. Finden Sie eine Liste der Aggregat-Funktionen in der PostgreSQL-Dokumentation.
2. Finden Sie ein GUI wie Navicat, mit dem Sie mit PostgreSQL interagieren können.

Machen Sie Folgendes

1. Entwickeln Sie eine Regel, die `DELETE`s für Veranstaltungsorte abfängt und stattdessen das active-Flag (das Sie im Selbststudium des ersten Tages eingefügt haben) auf `FALSE` setzt.
2. Eine temporäre Tabelle ist zur Implementierung der Eventkalender-Pivot-Tabelle nicht die beste Lösung. Die Funktion `generate_series(a, b)` gibt eine Reihe von Datensätzen von a bis b zurück. Ersetzen Sie das im `month_count-SELECT` entsprechend.
3. Bauen Sie eine Pivot-Tabelle auf, die jeden Tag eines Monats ausgibt. Dabei bildet jede Woche des Monats wie bei einem Standard-Kalender eine Reihe und jeder Tag bildet eine Spalte (sieben Tage, die bei `Sonntag` beginnen und mit dem `Samstag` enden). Jeder Tag soll die Anzahl der Events an diesem Datum enthalten bzw. leer bleiben, wenn es keine Events gibt.

2.4 Tag 3: Volltext und Mehrdimensionales

Wir wollen uns am dritten Tag die uns zur Verfügung stehenden Tools ansehen, um ein Abfragesystem für eine Filmdatenbank aufzubauen. Wir beginnen mit den vielen Möglichkeiten, mit denen PostgreSQL die Namen von Schauspielern/Filmen über „unscharfes“ (fuzzy) Stringmatching suchen kann. Dann sehen wir uns das cube-Paket an, mit dem wir ein System aufbauen, das uns Filme basierend auf den von uns bevorzugten Genres empfiehlt. Da das alles separate Pakete sind, sind die Implementierungen PostgreSQL-spezifisch und nicht Teil des SQL-Standards.

Üblicherweise beginnt man beim Entwurf eines relationalen Datenbankschemas mit einem Entitäten-Diagramm. Wir werden unser persönliches Film-Empfehlungsystem aufbauen, das Filme, ihre Genres und die Schauspieler nachhält. Das Modell ist in Abbildung 6, *Unser Film-Empfehlungssystem*, auf Seite 40 zu sehen.

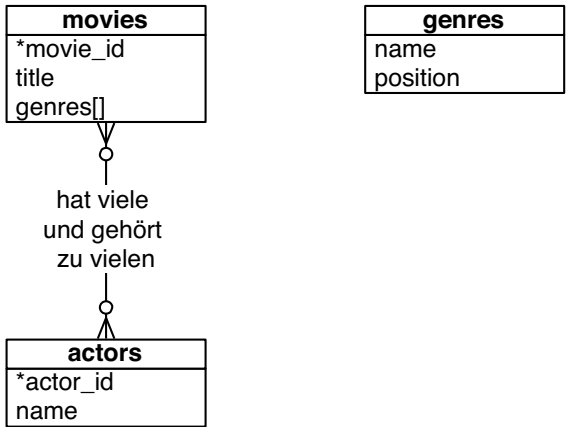


Abbildung 6: Unser Film-Empfehlungssystem

Sie werden sich erinnern, dass wir am ersten Tag verschiedene Zusatzpakete installiert haben. Heute werden wir sie alle nutzen. Hier noch einmal die Liste der benötigten Pakete: `tablefunc`, `dict_xsyn`, `fuzzystrmatch`, `pg_trgm` und `cube`.

Zuerst wollen wir die Datenbank aufbauen. Häufig baut man Indizes für Fremdschlüssel auf, um „Reverse Lookups“ (z. B. für die Filme, in denen ein Schauspieler mitgespielt hat) zu beschleunigen. Sie können auch ein `UNIQUE`-Constraint für Join-Tabellen wie `movies_actors` festlegen, um doppelte Join-Werte zu vermeiden.

postgres/create_movies.sql

```
CREATE TABLE genres (
  name text UNIQUE,
  position integer
);
CREATE TABLE movies (
  movie_id SERIAL PRIMARY KEY,
  title text,
  genre cube
);
CREATE TABLE actors (
  actor_id SERIAL PRIMARY KEY,
  name text
);

CREATE TABLE movies_actors (
  movie_id integer REFERENCES movies NOT NULL,
  actor_id integer REFERENCES actors NOT NULL,
  UNIQUE (movie_id, actor_id)
);
CREATE INDEX movies_actors_movie_id ON movies_actors (movie_id);
```

```
CREATE INDEX movies_actors_actor_id ON movies_actors (actor_id);
CREATE INDEX movies_genres_cube ON movies USING gist (genre);
```

Sie können die zum Buch passende Datei `movies_data.sql` herunterladen, um die Tabellen mit Leben zu füllen. Alle Fragen zum `genre cube` behandeln wir im Verlauf dieses Tages.

Unscharfe Suche

Ein System für die Textsuche zu öffnen, bedeutet, sich für ungenaue Eingaben zu öffnen. Sie müssen mit Tippfehlern wie „Franksteins Braut“ rechnen. Manchmal können sich die Benutzer nicht an den vollständigen Namen von „J. Roberts“ erinnern. In anderen Fällen können wir uns einfach nicht daran erinnern, wie man „Benn Aflek“ richtig schreibt. Wir wollen uns einige PostgreSQL-Pakete ansehen, die die Textsuche vereinfachen. Und je weiter wir einsteigen, desto mehr verschwimmen bei diesem String-Matching die Grenzen zwischen relationalen Queries und Such-Frameworks wie Lucene.[6] Auch wenn einige der Ansicht sind, dass Features wie die Volltextsuche in den Anwendungscode gehören, kann es im Bezug auf Performance und Administration vorteilhaft sein, diese Aufgaben der Datenbank zu überlassen, wo die Daten liegen.

Standard-String-Matching bei SQL

PostgreSQL kann Texte auf vielerlei Weise durchsuchen, aber die beiden bekanntesten Standard-Methoden sind `LIKE` und Reguläre Ausdrücke.

LIKE und ILIKE

`LIKE` und `ILIKE` (Casesensitives `LIKE`) sind die einfachsten Formen der Textsuche und bei relationalen Datenbanken nahezu immer vorhanden. `LIKE` vergleicht einen Spaltenwert mit einem gegebenen String-Muster, bei dem die Zeichen % und _ Platzhalter (Wildcards) sind. % steht dabei für eine beliebige Zahl von Zeichen, während _ für genau ein Zeichen steht.

```
SELECT title FROM movies WHERE title ILIKE 'stardust%';

       title
-------------------
 Stardust
 Stardust Memories
```

Soll der Teilstring *stardust* nicht das Ende des Strings sein, können Sie den Unterstrich (_) als kleinen Trick verwenden.

6. http://lucene.apache.org/

```
SELECT title FROM movies WHERE title ILIKE 'stardust_%';

       title
------------------
 Stardust Memories
```

Das ist in einfachen Fällen nützlich, doch `LIKE` ist auf simple Wildcards beschränkt.

Reguläre Ausdrücke

Eine wesentlich leistungsfähigere String-Matching-Syntax bieten *Reguläre Ausdrücke* (Regex). Reguläre Ausdrücke erscheinen im gesamten Buch recht oft, da viele Datenbanken sie unterstützen. Es gibt ganze Bücher, die sich nur der Entwicklung leistungsfähiger regulärer Ausdrücke widmen – das Thema ist bei Weitem zu kompliziert, um hier eingehender behandelt werden zu können. Postgres ist (größtenteils) POSIX-konform.

Bei Postgres wird ein Regex-Match durch den Operator ~ eingeleitet, dem optional ein ! (für *nicht* vorstehen) und ein * (für *schreibungsunabhängig*) folgen kann. Wenn Sie also alle Filme zählen wollen, die *nicht* mit *the* beginnen, können Sie die folgende schreibungsunabhängige Query verwenden. Die Zeichen innerhalb des Strings sind der reguläre Ausdruck.

```
SELECT COUNT(*) FROM movies WHERE title !~* '^the.*';
```

Sie können Strings für das Pattern Matching obiger Queries indexieren, indem Sie einen `text_pattern_ops`-Operatorklassen-Index erzeugen, solange die indexierten Werte kleingeschrieben sind.

```
CREATE INDEX movies_title_pattern ON movies (lower(title) text_pattern_ops);
```

Wir haben `text_pattern_ops` verwendet, weil der Titel vom Typ `text` ist. Wenn Sie varchars, chars, oder Namen indexieren müssen, verwenden Sie die entsprechenden Operatoren `varchar_pattern_ops`, `bpchar_pattern_ops` und `name_pattern_ops`.

Levenshtein

Levenshtein ist ein Stringsvergleichsalgorithmus, der die Ähnlichkeit zweier Strings vergleicht, indem er berechnet, wie viele *Schritte* notwendig sind, um einen String in den anderen zu überführen. Jedes ersetzte, fehlende oder hinzugefügte Zeichen gilt als Schritt. Die Distanz entspricht der Gesamtzahl der Schritte. Bei PostgreSQL wird die `levenshtein()`-Funktion über das Zu-

satzpaket `fuzzystrmatch` bereitgestellt. Betrachten wir mal die Strings *bat* und *fads*.

```
SELECT levenshtein('bat', 'fads');
```

Die Levenshtein-Distanz ist 3, weil – verglichen mit dem String *bat* – zwei Buchstaben ersetzt (b=>f, t=>d) und einer hinzugefügt werden mussten (+s). Jede Änderung erhöht die Distanz. Die Distanz wird kleiner, je weiter wir uns annähern. Die Distanz nimmt ab, bis sie Null erreicht (wenn beide Strings gleich sind).

```
SELECT levenshtein('bat', 'fad') fad,
  levenshtein('bat', 'fat') fat,
  levenshtein('bat', 'bat') bat;

 fad | fat | bat
-----+-----+-----
   2 |   1 |   0
```

Die Änderung der Groß-/Kleinschreibung kostet ebenfalls einen Punkt, d. h., Sie wandeln die Strings vor der Query besser in Groß- oder Kleinbuchstaben um.

```
SELECT movie_id, title FROM movies
WHERE levenshtein(lower(title), lower('a hard day nght')) <= 3;

 movie_id |       title
----------+--------------------
      245 | A Hard Day's Night
```

Damit stellen Sie sicher, dass unbedeutende Unterschiede sich nicht auf die Distanz auswirken.

Trigramm

Ein Trigramm ist eine Gruppe dreier aufeinanderfolgender Zeichen aus einem String. Das Zusatzmodul `pg_trgm` bricht einen String in soviele Trigramme wie möglich auf.

```
SELECT show_trgm('Avatar');

show_trgm
-----------------------------------
 {"  a"," av","ar ",ata,ava,tar,vat}
```

Um einen passenden String zu finden, muss man nur die Anzahl passender Trigramme zählen, die Strings mit den meisten Treffern sind am ähnlichsten.

Das ist nützlich, wenn die Suche kleine Schreibfehler oder fehlende Wörter tolerieren soll. Je länger der String, desto mehr Trigramme und desto eher findet sich ein Treffer – sie eignen sich gut für Filmtitel, da diese immer ähnlich lang sind.

Wir erzeugen einen Trigramm-Index über die Filmnamen (wozu wir Generalized Index Search Tree [GIST] nutzen, eine generische Index-API, die für die PostgreSQL-Engine zur Verfügung steht).

```
CREATE INDEX movies_title_trigram ON movies
USING gist (title gist_trgm_ops);
```

Nun können unsere Anfragen auch kleinere Schreibfehler enthalten und wir erhalten trotzdem vernünftige Ergebnisse.

```
SELECT *
FROM movies
WHERE title % 'Avatre';

 title
---------
 Avatar
```

Trigramme sind eine gute Wahl, wenn man Benutzereingaben ohne Wildcards verarbeiten möchte.

Volltextsuche

Nun wollen wir den Benutzern Volltextsuchen basierend auf passenden Wörtern (auch im Plural) erlauben. Postgres unterstützt eine einfache maschinelle Sprachverarbeitung, d. h., ein Benutzer kann nach bestimmten Wörtern in einem Filmtitel suchen, auch wenn er sich nicht an alle Wörter erinnert.

TSVector und TSQuery

Sehen wir uns einen Film an, der die Wörter *night* und *day* enthält. Das ist genau die richtige Aufgabe für den Volltext-Queryoperator @@.

```
SELECT title
FROM movies
WHERE title @@ 'night & day';

            title
------------------------------
 A Hard Day's Night
 Six Days Seven Nights
 Long Day's Journey Into Night
```

Die Query gibt Titel wie *A Hard Day's Night* zurück, auch wenn das Wort *Day* im Genitiv vorliegt und die beiden Wörter in der Query in der falschen Reihenfolge stehen. Der Operator @@ wandelt das Namensfeld in einen `tsvector` und die Query in eine `tsquery` um.

Ein `tsvector` ist ein Datentyp, der einen String in ein Array (einen *Vektor*) von Tokens zerlegt, der dann durch die Query abgesucht wird. `tsquery` repräsentiert hingegen eine Query in irgendeiner Sprache wie Englisch oder Französisch. Die Sprache entspricht einem Wörterbuch (über das wir gleich mehr erfahren). Die obige Query ist mit der folgenden Query identisch (wenn Ihre Systemsprache auf Englisch eingestellt ist):

```
SELECT title
FROM movies
WHERE to_tsvector(title) @@ to_tsquery('english', 'night & day');
```

Sie können sich ansehen, wie der Vektor und die Query aufgebrochen werden, indem Sie die Konvertierungsfunktionen direkt über die Strings laufen lassen.

```
SELECT to_tsvector('A Hard Day''s Night'), to_tsquery('english', 'night & day');

   to_tsvector             |   to_tsquery
---------------------------+-----------------
'day':3 'hard':2 'night':5 | 'night' & 'day'
```

Die Tokens eines `tsvectors` werden *Lexeme* genannt und sind mit ihren Positionen innerhalb des Textes verknüpft.

Sie werden bemerkt haben, dass der `tsvector` für *A Hard Day's Night* das Lexem *a* nicht enthält. Mehr noch, einfache englische Wörter wie *a* fehlen, wenn Sie nach ihnen suchen.

```
SELECT *
FROM movies
WHERE title @@ to_tsquery('english', 'a');

NOTICE:  text-search query contains only stop words or doesn't \
       contain lexemes, ignored
```

Gängige Wörter wie das englische *a* werden *Stoppwörter* genannt und sind für Queries generell nicht geeignet. Das englische Wörterbuch wurde vom Parser verwendet, um den String zu normalisieren und in nützliche englische Komponenten aufzuteilen. In der Console können Sie sich die englischen Stoppwörter im Verzeichnis `tsearch_data` ansehen.

```
cat 'pg_config --sharedir'/tsearch_data/english.stop
```

Wir können *a* aus der Liste entfernen oder wir können ein anderes Wörterbuch wie `simple` verwenden, das String nur bei Nichtwort-Zeichen zerlegt und in Kleinbuchstaben umwandelt. Vergleichen Sie die beiden folgenden Vektoren:

```
SELECT to_tsvector('english', 'A Hard Day''s Night');

to_tsvector
----------------------------
'day':3 'hard':2 'night':5

SELECT to_tsvector('simple', 'A Hard Day''s Night');

to_tsvector
---------------------------------------
'a':1 'day':3 'hard':2 'night':5 's':4
```

Mit `simple` können wir jeden Film abrufen, der das Lexem *a* enthält.

Weitere Sprachen

Da Postgres hier eine maschinelle Sprachverarbeitung durchführt, ist es sinnvoll, dass unterschiedliche Konfigurationen für unterschiedliche Sprachen genutzt werden. Alle installierten Konfigurationen können Sie sich mit dem folgenden Befehl ansehen:

```
book=# \dF
```

Wörterbücher sind ein Teil dessen, was Postgres zur Generierung der `tsvector`-Lexeme verwendet (zusammen mit Stoppwörtern und anderen Tokenizing-Regeln wie *Parsern* und *Templates*, auf die wir hier nicht eingehen). Eine Liste Ihres Systems können Sie sich wie folgt ansehen:

```
book=# \dFd
```

Sie können jedes Wörterbuch direkt ausprobieren, im Sie die Funktion `ts_lexize()` aufrufen. Hier ermitteln wir die englische Stammform des Worts *Day's*.

```
SELECT ts_lexize('english_stem', 'Day''s');

 ts_lexize
-----------
 {day}
```

Die Volltextsuche funktioniert natürlich auch in anderen Sprachen. Wenn Sie Deutsch installiert haben, probieren Sie Folgendes:

```
SELECT to_tsvector('german', 'was machst du gerade?');

    to_tsvector
--------------------
'gerad':4 'mach':2
```

Da *was* und *du* gängig sind, werden sie im deutschen Wörterbuch als Stoppwörter markiert, während *machst* und *gerade* abgeleitet werden.

Lexeme indexieren

Die Volltextsuche ist sehr leistungsfähig. Doch wenn wir unsere Tabellen nicht indexieren, ist sie auch langsam. Der `EXPLAIN`-Befehl ist ein mächtiges Werkzeug, wenn man sich ansehen möchte, wie Queries intern geplant werden.

```
EXPLAIN
SELECT *
FROM movies
WHERE title @@ 'night & day';

                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on movies  (cost=10000000000.00..10000000001.12 rows=1 width=68)
   Filter: (title @@ 'night & day'::text)
```

Beachten Sie die Zeile *Seq Scan on movies*. Das ist bei einer Query nur selten ein gutes Zeichen, denn es bedeutet, dass die gesamte Tabelle abgesucht wird, d. h., jede Zeile wird gelesen. Wir brauchen also einen Index.

Wir nutzen Generalized Inverted iNdex (GIN) – wie GIST eine Index-API – , um einen Index der Lexeme aufzubauen, um schnelle Queries zu ermöglichen. Der Begriff *invertierter Index* wird Ihnen bekannt vorkommen, wenn Sie schon einmal mit Suchmaschinen wie Lucene oder Sphinx gearbeitet haben. Diese Datenstruktur ist zur Indexierung bei Volltextsuchen üblich.

```
CREATE INDEX movies_title_searchable ON movies
USING gin(to_tsvector('english', title));
```

Nachdem wir den Index erzeugt haben, führen wir die Suche erneut aus.

```
EXPLAIN
SELECT *
FROM movies
WHERE title @@ 'night & day';

                                QUERY PLAN
--------------------------------------------------------------------------
 Seq Scan on movies  (cost=10000000000.00..10000000001.12 rows=1 width=68)
   Filter: (title @@ 'night & day'::text)
```

Was ist passiert? Nichts. Der Index ist da, aber Postgres nutzt ihn nicht. Das liegt daran, dass unser GIN-Index gezielt die `english`-Konfiguration zum Aufbau der `tsvector`en nutzt, während wir diesen Vektor nicht angeben. Wir müssen ihn in der `WHERE`-Klausel der Query explizit angeben.

```
EXPLAIN
SELECT *
FROM movies
WHERE to_tsvector('english',title) @@ 'night & day';

                                QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on movies  (cost=4.26..8.28 rows=1 width=68)
   Recheck Cond: (to_tsvector('english'::regconfig, title) @@ '''day'''::tsquery)
   -> Bitmap Index Scan on movies_title_searchable  (cost=0.00..4.26 rows=1 width=0)
        Index Cond: (to_tsvector('english'::regconfig, title) @@ '''day'''::tsquery)
```

`EXPLAIN` ist wichtig, um sicherzustellen, dass Indizes so genutzt werden, wie Sie es erwarten. Anderenfalls ist der Index nur unnötiger Overhead.

Metaphone

Wir haben uns in Richtung weniger spezifischer Eingaben bewegt. Bei `LIKE` und regulären Ausdrücken müssen Sie Muster angeben, die die Strings entsprechend ihrem Format genau beschreiben. Die Levenshtein-Distanz findet Treffer mit kleineren Schreibfehlern, doch letztlich muss man recht nah am gesuchten String sein. Trigramme sind eine gute Wahl für akzetabel falsch geschriebene Wörter. Die Volltextsuche erlaubt schließlich eine natürliche sprachliche Flexibilität, indem sie z. B. englische Wörter wie *a* und *the* ignoriert und mit der Pluralisierung umgehen kann. Manchmal wissen wir nicht, wie man ein Wort richtig schreibt, doch wir wissen, wie man es spricht.

Wir lieben Bruce Willis und wollen wissen, in welchen Filmen er mitgespielt hat. Unglücklicherweise können wir uns nicht genau daran erinnert, wie man seinen Namen schreibt, weshalb wir ihn so schreiben, wie es sich in etwa anhört.

```
SELECT *
FROM actors
WHERE name = 'Broos Wlis';
```

Selbst ein Trigramm hilft uns hier nicht weiter (wir verwenden % statt =).

```
SELECT *
FROM actors
WHERE name % 'Broos Wlis';
```

Nutzen Sie Metaphone, einen Algorithmus, der einen String in eine Art Lautschrift umwandelt. Sie können festlegen, wie viele Zeichen der Ausgabestring hat. Zum Beispiel ist das Sieben-Zeichen-Metaphone für den Namen Aaron Eckhart *ARNKHRT*.

Um alle Filme zu finden, in denen ein Schauspieler mitgewirkt hat, der wie „Broos Wils“ klingt, können wir die Query über die Metaphone-Ausgabe laufen lassen. Beachten Sie, dass ein natürlicher Join (`NATURAL JOIN`) ein `INNER JOIN` ist, bei dem der Join automatisch bei passenden Spaltennamen erfolgt (z. B. `movies.actor_id= movies_actors.actor_id`).

```
SELECT title
FROM movies NATURAL JOIN movies_actors NATURAL JOIN actors
WHERE metaphone(name, 6) = metaphone('Broos Wils', 6);

            title
-----------------------------
 The Fifth Element
 Twelve Monkeys
 Armageddon
 Die Hard
 Pulp Fiction
 The Sixth Sense
:
```

Wenn Sie sich die Online-Dokumentation ansehen, sehen Sie, dass das *fuzzystrmatch*-Modul weitere Funktionen enthält: `dmetaphone` (double metaphone), `dmetaphone_alt` (für die alternative Aussprache von Namen) und `soundex` (ein wirklich alter Algorithmus aus den 1880ern, entwickelt für die amerikanische Volkszählung zum Vergleich gängiger amerikanischer Nachnamen).

Sie können die Ergebnisse der Funktionen untersuchen, indem Sie sich deren Ausgabe ansehen.

```
SELECT name, dmetaphone(name), dmetaphone_alt(name),
  metaphone(name, 8), soundex(name)
FROM actors;
```

```
      name      | dmetaphone | dmetaphone_alt | metaphone | soundex
----------------+------------+----------------+-----------+--------
 50 Cent        | SNT        | SNT            | SNT       | C530
 Aaron Eckhart  | ARNK       | ARNK           | ARNKHRT   | A652
 Agatha Hurle   | AK0R       | AKTR           | AK0HRL    | A236
:
```

Es gibt nicht die beste Funktion schlechthin. Die optimale Wahl hängt von Ihren Daten ab.

String-Matches kombinieren

Da wir nun all unsere Suchtechniken beisammen haben, können wir damit beginnen, sie auf interessante Art und Weise zu kombinieren.

Einer der flexibelsten Aspekte von Metaphones ist, dass die Ergebnisse einfache Strings sind. Das erlaubt es uns, sie mit anderen String-Matchern zu kombinieren.

Zum Beispiel können wir den Trigramm-Operator auf die `metaphone`-Ausgaben anwenden und die Ergebnisse nach der Levenshtein-Distanz sortieren. Die folgende Query bedeutet: „Gib mir sortiert die Namen zurück, die sich am ehesten nach `Robin Williams` anhören“.

```
SELECT * FROM actors
WHERE metaphone(name,8) % metaphone('Robin Williams',8)
ORDER BY levenshtein(lower('Robin Williams'), lower(name));

 actor_id |      name
----------+-----------------
     2442 | John Williams
     4090 | Robin Shou
     4093 | Robin Williams
     4479 | Steven Williams
```

Wie Sie sehen, ist das nicht perfekt. Robin Williams steht an dritter Stelle. Die unkontrollierte Nutzung dieser Flexibilität kann zu sehr lustigen Ergebnissen führen, also Vorsicht.

```
SELECT * FROM actors WHERE dmetaphone(name) % dmetaphone('Ron');

 actor_id |    name
----------+-------------
     3911 | Renji Ishibashi
     3913 | Renée Zellweger
:
```

Die Kombinationsmöglichkeiten sind riesig und nur durch Ihre Experimentierfreude beschränkt.

Genre als mehrdimensionale Hypercubes

Das letzte Zusatzpaket, das wir uns ansehen wollen, ist `cube`. Wir nutzen den Datentyp `cube`, um die Filmgenres als mehrdimensionalen Vektor abzubilden. Dann verwenden wir Methoden, die effizient nach den nächstliegenden Punkten innerhalb der Grenzen eines Hypercubes suchen, um uns eine Liste ähnlicher Filme zurückzuliefern.

Ihnen wird aufgefallen sein, dass wir zu Beginn des 3. Tages eine Spalte namens `genres` vom Typ `cube` angelegt haben. Jeder Wert ist ein Punkt in einem 18-dimensionalen Raum, in dem jede Dimension ein Genre repräsentiert. Warum stellt man Filmgenres als Punkte in einem n-dimensionalen Raum dar? Die Kategorisierung von Filmen ist keine exakte Wissenschaft und viele Filme sind nicht zu 100 Prozent Komödie oder Tragödie, sondern liegen irgendwo dazwischen.

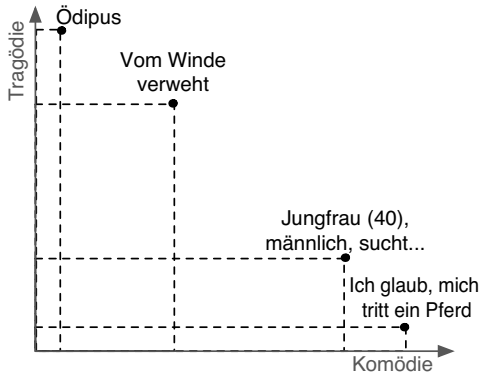
In unserem System wird jedem Genre ein (völlig willkürlicher) Wert zwischen 0 und 10 zugewiesen, je nachdem, wie stark der Film einem Genre verhaftet ist. Die 0 steht dabei für gar nicht und 10 für sehr stark.

Star Wars hat den Genre-Vektor `(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)`. Die `genres`-Tabelle beschreibt die `Position` jeder Dimension im Vektor. Wir können den Genre-Wert entschlüsseln, indem wir `cube_ur_coord(vector, dimension)` für jede `genres.position` abrufen. Der Klarheit halber filtern wir Genres mit dem Wert 0 aus.

```
SELECT name,
  cube_ur_coord('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) as score
FROM genres g
WHERE cube_ur_coord('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) > 0;

 name      | score
-----------+-------
 Adventure |     7
 Fantasy   |     7
 SciFi     |    10
```

Ähnliche Filme finden wir, indem wir die nächstliegenden Punkte aufspüren. Um zu verstehen, warum das funktioniert, können wir uns zwei Filme (wie in der nachfolgenden Abbildung) als zweidimensionales Genre-Diagramm vorstellen. Wenn Sie *Ich glaub', mich tritt ein Pferd (Animal House)* mögen, werden Sie sich wohl lieber *Jungfrau (40), männlich, sucht . . . (The 40 Year Old Virgin)* als *Oedipus* ansehen – eine Geschichte, der definitiv jeder Humor fehlt. In unserem zweidimensionalen Universum müssen wir nur eine Nearest-Neighbor-Suche durchführen, um passende Treffer zu finden.



Wir können das auf weitere Dimensionen mit mehr Genres (2, 3 oder 18) ausweiten. Das Prinzip ist das gleiche: Eine Nearest-Neighbor-Suche des nächsten Punktes im Genre-Raum liefert die nächsten Genre-Treffer zurück.

Die nächsten Treffer des Genre-Vektors können Sie mit `cube_distance (point1, point2)` ermitteln. Nachfolgend bestimmen wir die Distanz aller Filme zum *Star Wars*-Genre-Vektor und sortieren das Ergebnis nach dieser Distanz.

```
SELECT *,
  cube_distance(genre, '(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)') dist
FROM movies
ORDER BY dist;
```

Wir haben den `movies_genres_cube`-Index aufgebaut, als wir die Tabellen angelegt haben. Doch selbst mit einem Index ist die Abfrage relativ langsam, da sie eine vollständige Verarbeitung der Tabelle verlangt. Sie berechnet die Distanz für jede Zeile und sortiert sie dann.

Statt die Distanz für jeden Punkt zu berechnen, können wir uns mit Hilfe eines sog. *Bounding Cube* auf wahrscheinliche Kandidaten beschränken. Die fünf nächsten Städte für einen Punkt zu ermitteln, geht mit einer lokalen Landkarte wesentlich schneller als mit einer Weltkarte. Die Begrenzung (bounding) reduziert die Punkte, die wir uns ansehen müssen.

Wir verwenden `cube_enlarge(cube,radius,dimensions)`, um einen 18-dimensionalen Kubus aufzubauen, der etwas breiter (radius) als ein Punkt ist.

Sehen wir uns ein einfaches Beispiel an. Wenn wir ein zweidimensionales Quadrat um einen Punkt (1,1) aufbauen, liegt der untere linke Punkt des Quadrats bei (0,0) und der obere rechte Punkt bei (2,2).

```
SELECT cube_enlarge('(1,1)',1,2);
```

```
  cube_enlarge
----------------
 (0, 0),(2, 2)
```

Dieses Prinzip gilt für beliebige Dimensionen. Mit unserem begrenzten Hypercube können wir den speziellen Cube-Operator @> nutzen, der *enthält* bedeutet. Die folgende Query findet die Distanz aller Punkte, die in einem Fünf-Einheiten-Cube vom *Star Wars*-Genre-Punkt entfernt ist

```
SELECT title, cube_distance(genre, '(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)') dist
FROM movies
WHERE cube_enlarge('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'::cube, 5, 18) @> genre
ORDER BY dist;
```

```
                     title                      |       dist
------------------------------------------------+------------------
 Star Wars                                      |                0
 Star Wars: Episode V - The Empire Strikes Back |                2
 Avatar                                         |                5
 Explorers                                      | 5.74456264653803
 Krull                                          | 6.48074069840786
 E.T. The Extra-Terrestrial                     | 7.61577310586391
```

Mittels eines Subselects können wir das Genre über den Filmnamen bestimmen und unsere Berechnung über dieses Genre mit Hilfe eines Tabellen-Alias durchführen.

```
SELECT m.movie_id, m.title
FROM movies m, (SELECT genre, title FROM movies WHERE title = 'Mad Max') s
WHERE cube_enlarge(s.genre, 5, 18) @> m.genre AND s.title <> m.title
ORDER BY cube_distance(m.genre, s.genre)
LIMIT 10;
```

```
 movie_id |           title
----------+---------------------------
     1405 | Cyborg
     1391 | Escape from L.A.
     1192 | Mad Max Beyond Thunderdome
     1189 | Universal Soldier
     1222 | Soldier
     1362 | Johnny Mnemonic
      946 | Alive
      418 | Escape from New York
     1877 | The Last Starfighter
     1445 | The Rocketeer
```

Diese Methode der Film-Empfehlung ist nicht perfekt, aber ein sehr guter Anfang. Weitere dimensionale Queries werden wir in späteren Kapiteln noch sehen, z. B. die zweidimensionale geographische Suche in MongoDB (siehe 5.4, *Genre als mehrdimensionale Hypercubes*, auf Seite 187).

Was wir am dritten Tag gelernt haben

Heute sind wir tief in die Flexibilität der PostgreSQL-Stringsuche eingetaucht und haben das cube-Paket zur mehrdimensionalen Suche genutzt. Wir haben einen Blick auf die nicht standardkonformen Erweiterungen geworfen, die PostgreSQL an die Spitze der Open-Source-RDBMS gebracht haben. Ihnen stehen dutzende (wenn nicht hunderte) zusätzlicher Erweiterungen zur Verfügung, von der Speicherung geographischer Daten bis zu kryptographischen Funktionen, maßgeschneiderten Datentypen und Spracherweiterungen. Neben der Stärke von SQL sind Zusatzpakete das, was PostgreSQL brillieren lässt.

Tag 3: Selbststudium

Finden Sie heraus

1. Finden Sie die Online-Dokumentation aller bei Postgres mitgelieferten Zusatzpakete.
2. Finden Sie die Online-Dokumentation zu POSIX-Regex' (die auch für zukünftige Kapitel sehr praktisch ist).

Machen Sie Folgendes

1. Entwickeln Sie eine Stored Procedure, der Sie einen Filmtitel oder den Namen eines Schauspielers übergeben können und die fünf Empfehlungen zurückgibt. Diese Empfehlungen sollen auf den Filmen basieren, in denen der Schauspieler mitgespielt hat oder die ähnliche Genres abdecken.
2. Erweitern Sie die Filmdatenbank um Benutzerkommentare und extrahieren Sie daraus Schlüsselwörter (ohne englische Stoppwörter). Verknüpfen Sie diese Schlüsselwörter mit den Nachnamen der Schauspieler und versuchen Sie herauszufinden, über welche Schauspieler am meisten diskutiert wird.

2.5 Zusammenfassung

Wenn Sie noch nicht viel Zeit mit relationalen Datenbanken verbracht haben, sollten Sie unbedingt tiefer in PostgreSQL (oder eine andere relationale Datenbank) einsteigen, bevor Sie sie zugunsten einer anderen Variante fallen lassen. Relationale Datenbanken standen über 40 Jahre im Fokus intensiver akademischer Forschung und industrieller Verbesserungen, und PostgreSQL ist eine der besten relationalen Open-Source-Datenbanken, um von diesen Weiterentwicklungen zu profitieren.

PostgreSQLs Stärken

PostgreSQLs Stärken sind so vielfältig wie das relationale Modell: jahrelange Forschung und produktiver Einsatz in nahezu jedem Bereich der Datenverarbeitung, flexible Abfragen und sehr konsistente und beständige Daten. Die meisten Programmiersprachen besitzen erprobte Treiber für Postgres, und viele Programmiermodellen wie der objektrelationalen Abbildung (object-relational mapping, ORM) liegt eine relationale Datenbank zugrunde. Den Kern bildet die Flexibilität der Joins. Sie müssen nicht wissen, wie das Modell tatsächlich abgefragt wird, da Sie Joins, Filter, Views und Indizes beliebig nutzen können – und die Chancen stehen gut, dass Sie immer in der Lage sind, die gewünschten Daten zu extrahieren.

PostgreSQL eignet sich ausgezeichnet für sog. „Stepford-Daten“ (benannt nach *Die Frauen von Stepford*, einer Geschichte über eine Nachbarschaft, in der nahezu alle identisch sind, also ziemlich homogene Daten, die gut in ein strukturiertes Schema passen.

PostgreSQL beherrscht aber weit mehr, als die normalen Open Source RDBMS zu bieten haben, etwa einen mächtigen Constraint-Mechanismus. Sie können eigene Spracherweiterungen entwickeln, Indizes anpassen, benutzerdefinierte Datentypen entwickeln und sogar die Verarbeitung eingehender Queries ändern. Und während andere Open-Source-Datenbanken komplizierte Lizenzbestimmungen verwenden, ist PostgreSQL Open Source in seiner reinsten Form. Der Code gehört niemandem. Jeder kann so ziemlich alles mit dem Projekt machen, was er will (außer die Autoren dafür haftbar machen). Die Entwicklung und Distribution liegt völlig in der Hand der Community. Wenn Sie ein Fan freier Software sind und einen langen, buschigen Bart haben, müssen Sie den generellen Widerstand gegen die Kommerzialisierung eines tollen Produkts respektieren.

PostgreSQLs Schwächen

Zwar sind relationale Datenbanken zweifellos seit Jahren die erfolgreichsten Datenbank-Vertreter, doch es gibt Fälle, für die sie nicht sehr gut geeignet sind.

Partitionierung ist nicht die starke Seite relationaler Datenbanken wie PostgreSQL. Wenn Sie in die Breite und nicht in die Höhe wachsen (d. h. mehrere parallele Datenspeicher anstelle einer einzigen starken Maschine/eines Clusters), dann sehen Sie sich besser nach anderen Lösungen um. Wenn die Anforderungen an Ihre Daten zu vielfältig sind, um einfach in das rigide Schema relationaler Datenbanken zu passen, oder wenn Sie den Overhead einer vollwertigen Datenbank nicht benötigen, Schlüssel/Wert-Paare in großen Mengen schreiben und lesen oder nur große Blobs speichern müssen, dann ist einer der anderen Datenspeicher vielleicht die bessere Wahl.

Abschließende Gedanken

Eine relationale Datenbank ist eine ausgezeichnete Wahl, wenn es um die Flexibilität der Queries geht. Zwar verlangt PostgreSQL von Ihnen, dass Sie die Form der Daten im Vorfeld festlegen, legt Ihnen bei der Nutzung der Daten aber keinerlei Beschränkungen auf. Solange Ihr Schema halbwegs ordentlich normalisiert ist (ohne Duplikate oder der Speicherung zu berechnender Werte), sollten Sie in der Lage sein, nahezu jede benötigte Abfrage zu formulieren. Und wenn Sie die richtigen Module einbinden, die Engine und den Index gut abstimmen, dann kann es mehrere Terabyte Daten mit sehr wenig Ressourcen überraschend gut verarbeiten. Für diejenigen, denen Datensicherheit besonders wichtig ist, stellen PostgreSQLs ACID-konforme Transaktionen sicher, dass die Aktionen atomisch, konsistent, isoliert und dauerhaft sind.

KAPITEL 3

Riak

Jeder, der schon einmal am Bau gearbeitet hat weiß, dass ein Bewehrungsstab verwendet wird, um Beton zu verstärken. Genau wie Riak („Rie-ahck") nutzt man nicht einen allein, aber die vielen zusammenarbeitenden Teile machen das Gesamtsystem stabil. Jede Komponente für sich ist günstig und verzichtbar, doch richtig eingesetzt, findet man kaum eine einfachere und stärkere Struktur für ein Fundament.

Riak ist eine verteilte Schlüssel/Wert-Datenbank, bei der die Werte alles sein können – von reinem Text, JSON oder XML bis zu Images oder Video-Clips – , und alles ist über eine einfache HTTP-Schnittstelle zugänglich. Welche Daten Sie auch immer haben, Riak kann sie speichern.

Riak ist auch fehlertolerant. Server können jederzeit vom oder ans Netz gehen, ohne dass es zu einem „Single Point of Failure" kommt. Ihr Cluster läuft weiter, während Server hinzugefügt und entfernt werden oder (hoffentlich nicht) abstürzen. Riak hält Sie nachts nicht wach, weil Sie sich Sorgen um Ihr Cluster machen – der Ausfall eines Knotens ist kein Notfall und Sie können sich dem Problem auch am nächsten Morgen widmen. Wie Kern-Entwickler Justin Sheehy einmal (frei übersetzt) anmerkte, „[Das Riak-Team] konzentrierte sich stark auf Dinge wie Schreib-Verfügbarkeit. . . , um wieder schlafen gehen zu können."

Doch diese Flexibilität hat einige Nachteile. Riak fehlt es an stabiler Unterstützung für ad Ad-hoc-Queries und Schlüssel/Wert-Speicher haben prinzipbedingt Probleme, Werte miteinander zu verknüpfen (mit anderen Worten, es gibt keine Fremdschlüssel). Riak geht diese Probleme an mehreren Fronten an, die wir uns in den nächstenTagen ansehen werden.

3.1 Riak liebt das Web

Riak spricht die Sprache des *Web* besser als jede andere Datenbank in diesem Buch (auch wenn CouchDB nah dran ist). Abfragen laufen über URLs, Header und Verben, und Riak gibt entsprechende Daten und Standard-HTTP-Response-Codes zurück.

Riak und cURL

Das Ziel dieses Buches besteht darin, Ihnen sieben Datenbanken und ihre Konzepte vorzustellen, und nicht, Ihnen neue Programmiersprachen beizubringen. Deshalb vermeiden wir die Einführung neuer Programmiersprachen, wenn das möglich ist. Riak stellt ein HTTP REST-Interface zur Verfügung, weshalb wir über das URL-Tool cURL mit der Datenbank kommunizieren. Im Produktiveinsatz werden Sie nahezu immer einen Treiber für die von Ihnen bevorzugte Datenbank verwenden. Die Verwendung von cURL erlaubt es uns, einen Blick auf die zugrundeliegende API zu werfen, ohne einen bestimmten Treiber oder eine bestimmte Programmiersprache nutzen zu müssen.

Riak ist eine gute Wahl für Rechenzentren wie Amazon, die viele Requests mit geringer Latenz bedienen müssen. Wenn jede mit Warten verplemperte Millisekunde den potentiellen Verlust eines Kunden darstellt, ist Riak schwer zu schlagen. Es ist einfach zu verwalten, einfach einzurichten und kann mit Ihren Anforderungen wachsen. Wenn Sie schon einmal die Amazon Web Services wie SimpleDB oder S3 genutzt haben, werden Sie Ähnlichkeiten in Form und Funktion bemerken. Das ist kein Zufall. Riak wurde von Amazons Dynamo-Papier inspiriert.[1]

In diesem Kapitel sehen wir uns an, wie Riak Werte speichert und abruft und wie es Daten über `Link`s verknüpft. Dann untersuchen wir ein Konzept des Datenabrufs (Data Retrieval), das in diesem Buch sehr oft genutzt wird: Mapreduce. Sie werden sehen, wie Riak seine Server gruppiert ("clustert") und Requests (selbst bei einem Server-Fehler) behandelt. Abschließend sehen wir uns an, wie Riak Konflikte löst, die beim Schreiben an verteilte Server auftreten, und wir sehen uns einige Erweiterungen des Servers an.

3.2 Tag 1: CRUD, Links und MIMEs

Sie können einen Riad-Build von Basho[2] (dem Unternehmen, das die Entwicklung finanziert) herunterladen und installieren, doch wir ziehen einen

1. http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
2. http://www.basho.com/

eigenen Build vor, da man einige vorkonfigurierte Beispiele erhält. Wenn Sie keinen eigenen Build vornehmen wollen, installieren Sie einfach eine vorkompilierte Version und besorgen Sie sich dann den Quellcode, um die Beispiel-Dev-Server zu extrahieren. Erlang[3] wird ebenfalls benötigt, um Riak (R14B03 und höher) auszuführen.

Um Riak aus den Quellen zu kompilieren, benötigen Sie drei Dinge: Erlang, den Quellcode und allgemeine Unix-Build-Tools wie Make. Die Installation von Erlang ist einfach (Sie brauchen Erlang auch für CouchDB in Kapitel 6, *CouchDB*, auf Seite 193), auch wenn sie eine Weile dauern kann. Wir laden die Riak-Quellen aus dem Repository (den Link finden Sie auf der Basho-Website – wenn Sie nicht mit Git oder Mercurial arbeiten, können Sie ein gezipptes Paket herunterladen). Alle Beispiele in diesem Kapitel wurden mit der Version 1.0.2 durchgeführt.

Die Riak-Entwickler spielten für uns neue Benutzer den Weihnachtsmann und hinterließen ein cooles Spielzeug in unserem Strumpf. Im gleichen Verzeichnis, in dem Sie Riak kompiliert haben, führen Sie den folgenden Befehl aus:

```
$ make devrel
```

Wenn die Kompilierung abgeschlossen ist, finden wir drei Beispiel-Server, die wir einfach starten:

```
$ dev/dev1/bin/riak start
$ dev/dev2/bin/riak start
$ dev/dev3/bin/riak start
```

Keine Sorge, wenn ein Server nicht starten kann, weil ein Port bereits belegt ist. Sie können den Port für dev1, dev2 oder dev3 ändern, indem Sie die Konfigurationsdatei `etc/app.config` des Servers öffnen und in der entsprechenden Zeile einen anderen Port angeben:

```
{http, [ {"127.0.0.1", 8091 } ]}
```

Es sollten nun drei Erlang-Prozesse namens `beam.smp` laufen, die die einzelnen Riak-Knoten (Server-Instanzen) repräsentieren und von der Existenz der jeweils anderen nichts wissen. Um ein Cluster aufzubauen, müssen wir die Knoten über den `riak-admin`-Befehl `join` des jeweiligen Servers mit einem anderen Knoten verbinden.

3. http://www.erlang.org/

```
$ dev/dev2/bin/riak-admin join dev1@127.0.0.1
```

Es spielt dabei keine Rolle, mit welchem Server wir sie verbinden – bei Riak sind alle Knoten gleich. Nachdem dev1 und dev2 in einem Cluster liegen, können wir dev3 mit einem dieser beiden verbinden.

```
$ dev/dev3/bin/riak-admin join dev2@127.0.0.1
```

Stellen Sie sicher, dass die Server laufen, indem Sie sich die Statistiken in einem Web-Browser ansehen: `http://localhost:8091/stats`. Sie könnten zum Download der Datei aufgefordert werden, die viele Informationen über das Cluster enthält. Das sollte wie folgt aussehen (wir haben es der Lesbarkeit halber ein wenig editiert):

```
{
  "vnode_gets":0,
  "vnode_puts":0,
  "vnode_index_reads":0,
  ...
  "connected_nodes":[
    "dev2@127.0.0.1",
    "dev3@127.0.0.1"
  ],
  ...
  "ring_members":[
    "dev1@127.0.0.1",
    "dev2@127.0.0.1",
    "dev3@127.0.0.1"
  ],
  "ring_num_partitions":64,
  "ring_ownership":
    "[{'dev3@127.0.0.1',21},{'dev2@127.0.0.1',21},{'dev1@127.0.0.1',22}]",
  ...
}
```

Wir können sehen, dass alle Server gleichwertige Teilnehmer des Rings sind, indem wir die Statistiken an den Ports 8092 (dev2) und 8093 (dev3) abrufen. Fürs Erste halten wir uns aber an die Statistiken von dev1.

Suchen Sie nach der *ring_members*-Property – sie sollte die Namen aller Knoten enthalten und bei allen Servern gleich sein. Suchen Sie als Nächstes nach dem Wert für *connected_nodes*. Er sollte die anderen Server des Rings enthalten.

Wir können die von *connected_nodes* gemeldeten Werte ändern, indem wir einen Knoten anhalten

```
$ dev/dev2/bin/riak stop
```

und die `/stats` neu laden. Beachten Sie, dass `dev2@127.0.0.1` aus der *connected_nodes*-Liste verschwunden ist. Starten Sie `dev2` und er vereinigt sich selbstständig wieder mit dem *Riak-Ring* (wir behandeln den Ring am 2. Tag).

REST und cURLs

REST steht für „REpresentational State Transfer“. Das klingt stark nach Fachchinesisch, ist aber die *De-facto*-Architektur für Web-Anwendungen. Es ist also durchaus sinnvoll, wenn man darüber Bescheid weiß. REST ist eine Richtlinie zur Abbildung von Ressourcen auf URLs und zur Interaktion mit diesen Ressourcen über CRUD-Verben: `POST` (Create, Erzeugen), `GET` (Read, Lesen), `PUT` (Update, Aktualisieren) und `DELETE` (Löschen).

Wenn Sie es noch nicht installiert haben, besorgen Sie sich den HTTP-Client cURL. Wir verwenden ihn als unsere REST-Schnittstelle, da man Verben (wie `GET` und `PUT`) und HTTP-Headerinformationen (wie `Content-Type`) auf einfache Weise angeben kann. Mit dem `curl`-Befehl kommunizieren wir direkt mit dem HTTP REST-Interface des Riak-Servers, ohne eine interaktive Console, wie z. B. einen Ruby-Treiber, zu benötigen.

Sie können überprüfen, ob der `curl`-Befehl mit Riak funktioniert, indem Sie einen Knoten „anpingen“.

```
$ curl http://localhost:8091/ping
OK
```

Lassen Sie uns eine fehlerhafte Query senden. Mit `-I` teilen wir cURL mit, dass wir nur die Header-Response wollen.

```
$ curl -I http://localhost:8091/riak/no_bucket/no_key
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Thu, 04 Aug 2011 01:25:49 GMT
Content-Type: text/plain
Content-Length: 10
```

Da Riak mit HTTP-URLs und -Aktionen arbeitet, nutzt es auch HTTP-Header und -Fehlercodes. Der Response-Code 404 bedeutet das gleiche wie bei einer fehlenden Webseite: Hier ist nichts. Lassen Sie uns also mit `PUT` etwas in Riak einfügen.

Der Parameter `-X PUT` teilt cURL mit, dass wir ein HTTP `PUT` ausführen wollen, um einen expliziten Schlüssel zu speichern und abzurufen. Das Attribut `-H` verwendet den nachfolgenden Text als HTTP-Headerinformation. In diesem Fall legen wir den MIME-Typ mit HTML fest. Alles was wir an `-d` (für Body-Daten) übergeben, speichert Riak als neuen Wert.

```
$ curl -v -X PUT http://localhost:8091/riak/favs/db \
 -H "Content-Type: text/html" \
 -d "My new favorite DB is RIAK"
```

Wenn Sie sich mit einem Browser zu `http://localhost:8091/riak/favs/db` bewegen, erhalten Sie eine schöne Nachricht von sich selbst.

Werte einfügen mit PUT

Riak ist ein Schlüssel/Wert-Speicher, erwartet also die Übergabe eines Schlüssels, um einen Wert abzurufen. Riak teilt Schlüsselklassen in sog. *Buckets* ("Behälter") auf, um Schlüsselkollisionen zu vermeiden. So kollidiert zum Beispiel ein Schlüssel für `java` als *Sprache* nicht mit `java` als *Getränk*.

Wir wollen ein System aufbauen, das die Tiere in einer Hundepension nachhält. Zuerst legen wir ein Bucket namens `animals` an, das alle Details unserer pelzigen Gäste enthält. Die URL folgt dabei diesem Muster:

```
http://SERVER:PORT/riak/BUCKET/KEY
```

Eine einfache Möglichkeit, ein Riak-Bucket zu füllen, besteht darin, den Schlüssel im Voraus zu kennen. Zuerst fügen wir *Ace, The Wonder Dog* ein. Wir geben ihm den Schlüssel `ace` und den Wert `{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}`. Sie müssen den Bucket nicht explizit anlegen – mit dem ersten Wert, der unter dem Bucket-Namen abgelegt wird, wird auch der Bucket erzeugt.

```
$ curl -v -X PUT http://localhost:8091/riak/animals/ace \
 -H "Content-Type: application/json" \
 -d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}'
```

Das Einfügen eines neuen Wertes liefert den Code 204 zurück. Das `-v` (verbose) im `curl`-Befehl gibt diese Header-Zeile zurück.

```
< HTTP/1.1 204 No Content
```

Wir können uns die Liste der von uns erzeugten Buckets ansehen.

```
$ curl -X GET http://localhost:8091/riak?buckets=true
{"buckets":["favs","animals"]}
```

Optional können Sie das Set-Ergebnis zurückgeben, indem Sie den Parameter `?returnbody=true` verwenden. Das testen wir durch Einfügen eines weiteren Tieres:

```
$ curl -v -X PUT http://localhost:8091/riak/animals/polly?returnbody=true \
 -H "Content-Type: application/json" \
 -d '{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}'
```

Diesmal erhalten wir den Code 200 zurück.

```
< HTTP/1.1 200 OK
```

Wenn man beim Schlüsselnamen nicht wählerisch ist, erzeugt Riak einen, wenn man mit POST arbeitet.

```
$ curl -i -X POST http://localhost:8091/riak/animals \
 -H "Content-Type: application/json" \
 -d '{"nickname" : "Sergeant Stubby", "breed" : "Terrier"}'
```

Der erzeugte Schlüssel findet sich im Header unter Location. Beachten Sie auch den 201-Code im Header.

```
HTTP/1.1 201 Created
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Location: /riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
Date: Tue, 05 Apr 2011 07:45:33 GMT
Content-Type: application/json
Content-Length: 0
```

Ein GET-Request (den cURL ohne weitere Parameter standardmäßig nutzt) an diese Location liefert uns den Wert zurück.

```
$ curl http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
```

DELETE löscht ihn.

```
$ curl -i -X DELETE http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
HTTP/1.1 204 No Content
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Mon, 11 Apr 2011 05:08:39 GMT
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
```

DELETE gibt keinen Body zurück, aber der HTTP-Code bei Erfolg ist 204. Anderenfalls wird (Sie ahnen es schon) 404 zurückgegeben.

Falls wir einen unserer Schlüssel im Bucket vergessen haben, können wir sie alle mit keys=true abrufen.

```
$ curl http://localhost:8091/riak/animals?keys=true
```

Sie können sie mit `keys=stream` auch als Stream abrufen, was bei großen Datensätzen die sicherere Variante sein kann. Dabei werden Segmente mit Schlüssel-Array-Objekten gesendet und mit einem leeren Array abgeschlossen.

Links

`Links` sind Metadaten, die einen Schlüssel mit anderen Schlüsseln verknüpfen. Die grundlegende Struktur sieht wie folgt aus:

```
Link: </riak/bucket/key>; riaktag=\"whatever\"
```

Der Schlüssel, auf den dieser Wert verweist, steht in spitzen Klammern(<>). Darauf folgt ein Semikolon und ein Tag, der beschreibt, in welcher Beziehung der Link zu diesem Wert steht (wobei Sie einen beliebigen String verwenden können).

Link-Walking

Unsere kleine Hundepension hat nur wenige (große, komfortable und menschenwürdige) Zwinger. Um nachzuhalten, welches Tier sich in welchem Zwinger befindet, verwenden wir einen Link. In Zwinger (Cage) 1 befindet sich (`contains`) Polly. Dazu verlinken wir ihren Schlüssel (wodurch auf ein neuer Bucket namens `cages` erzeugt wird). Der Zwinger steht in Raum 101, was wir als Wert im JSON-Format festhalten.

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
 -H "Content-Type: application/json" \
 -H "Link: </riak/animals/polly>; riaktag=\"contains\"" \
 -d '{"room" : 101}'
```

Beachten Sie, dass diese Link-Beziehung nur in eine Richtung verläuft. Der gerade erzeugte Zwinger weiß zwar, dass Polly darin sitzt, aber an den Polly-Daten selbst wurden keine Änderungen vorgenommen. Wir können das überprüfen, indem wir Pollys Daten abrufen und uns die `Link`-Header ansehen.

```
$ curl -i http://localhost:8091/riak/animals/polly

HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (participate in the frantic)
Link: </riak/animals>; rel="up"
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT
ETag: "VD0ZAfOTsIHsgG5PM3YZW"
Date: Tue, 13 Dec 2011 17:54:51 GMT
```

```
Content-Type: application/json
Content-Length: 59

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
```

Sie können (durch Kommata getrennt) so viele Metadaten-`Links` angeben wie nötig. Wir tragen Ace in Zwinger 2 ein und verweisen über *next_to* auf Zwinger 1, damit wir wissen, dass sie beieinanderstehen.

```
$ curl -X PUT http://localhost:8091/riak/cages/2 \
-H "Content-Type: application/json" \
-H "Link:</riak/animals/ace>;riaktag=\"contains\",
 </riak/cages/1>;riaktag=\"next_to\"" \
-d '{"room" : 101}'
```

Was `Links` bei Riak so besonders macht, ist die Link-Traversierung, das sog. *Link-Walking* (und eine leistungsfähigere Variante, sog. „verlinkte map-reduce-Queries“, die wir uns morgen ansehen). An die verlinkten Daten gelangen Sie, indem Sie eine *Linkspezifikation* (link spec) an die URL anhängen, die wie folgt strukturiert ist: `/_,_,_`. Die Unterstriche (_) in der URL repräsentieren Platzhalter (Wildcards) für die jeweiligen Link-Kriterien: bucket, tag, keep. Wir erklären diese Begriffe gleich. Zuerst wollen wir alle Links für Zwinger 1 abrufen.

```
$ curl http://localhost:8091/riak/cages/1/_,_,_

--4PYi9DW8iJK5aCvQQrrP7mh7jZs
Content-Type: multipart/mixed; boundary=Av1fawIA4WjypRlz5gHJtrRqklD
--Av1fawIA4WjypRlz5gHJtrRqklD
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--Av1fawIA4WjypRlz5gHJtrRqklD--

--4PYi9DW8iJK5aCvQQrrP7mh7jZs--
```

Zurückgegeben wird ein `multipart/mixed`-Dump der Header sowie die Bodies aller verlinkten Schlüssel/Werte. Da bekommt man beim Hinsehen schon Kopfschmerzen. Morgen sehen wir uns eine leistungsfähigere Möglichkeit an, an diese Daten zu gelangen, die auch schönere Werte zurückliefert – doch heute wollen wir noch etwas tiefer in diese Syntax einsteigen.

Im MIME-Typ `multipart/mixed` enthält die `Content-Type`-Definition einen boundary-Eintrag, der den Anfang und das Ende eines HTTP-Headers und der Body-Daten markiert.

```
--BcOdSWMLuhkisryp0GidDLqeA64
HTTP-Header und Body-Daten
--BcOdSWMLuhkisryp0GidDLqeA64--
```

In unserem Fall sind die Daten das, womit Zwinger 1 verlinkt ist: Polly Purebred. Sie haben vielleicht bemerkt, dass die zurückgelieferten Header die eigentlichen Link-Informationen nicht enthalten. Das ist in Ordnung; diese Daten sind immer noch unter dem linked-to-Schlüssel gespeichert.

Beim Link-Walking können wir die Unterstriche in der Link-Spezifikation ersetzen, um nach von uns gewünschten Werten zu filtern. Zwinger 2 besitzt zwei Links, d. h., ein Link-Spec-Request liefert sowohl das im Zwinger enthaltene Tier Ace zurück als auch den Zwinger 1 daneben (`next_to`). Um festzulegen, dass wir nur das `animals`-Bucket verfolgen wollen, ersetzen wir den ersten Unterstrich durch den Namen des Buckets.

```
$ curl http://localhost:8091/riak/cages/2/animals,_,_
```

Oder wir können dem *next to* des Zwingers folgen, indem wir das `tag`-Kriterium einsetzen.

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,_
```

Der letzte Unterstrich – `keep` – akzeptiert die Werte 1 oder 0. `keep` ist nützlich, wenn man Links zweiter Ordnung verfolgen will oder Links, die anderen Links folgen (was möglich ist, indem man einfach eine weitere Linkspezifikation anhängt). Wir wollen den Schlüsseln neben (`next_to`) Zwinger 2 folgen, was Zwinger 1 zurückgibt. Dann wollen wir den Tieren folgen, die mit Zwinger 1 verlinkt sind. Da wir `keep` auf 0 gesetzt haben, gibt Riak den Zwischenschritt (die Daten für Zwinger 1) nicht zurück. Es gibt nur die Daten zu Polly zurück, die sich neben Aces Zwinger befindet.

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,0/animals,_,_

--6mBdsboQ8kTT6MlUHg0rgvbLhzd
Content-Type: multipart/mixed; boundary=EZYdVz9Ox4xzR4jx1I2ugUFFiZh
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
Etag: VD0ZAfOTsIHsgG5PM3YZW
```

```
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh--

--6mBdsboQ8kTT6MlUHg0rgvbLhzd--
```

Wenn Sie die Informationen zu Polly und zu Zwinger 1 wünschen, setzen Sie `keep` auf 1.

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,1/_,_,_

--PDVOEl7Rh1AP90jGln1mhz7x8r9
Content-Type: multipart/mixed; boundary=YliPQ9LPNEoAnDeAMiRkAjCbmed

--YliPQ9LPNEoAnDeAMiRkAjCbmed
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRKY+VIYo35gRfFgA=
Location: /riak/cages/1
Content-Type: application/json
Link: </riak/animals/polly>; riaktag="contains", </riak/cages>; rel="up"
Etag: 6LYhRnMRrGIqsTmpE55PaU
Last-Modified: Tue, 13 Dec 2011 17:54:34 GMT

{"room" : 101}
--YliPQ9LPNEoAnDeAMiRkAjCbmed--

--PDVOEl7Rh1AP90jGln1mhz7x8r9
Content-Type: multipart/mixed; boundary=GS9J6KQLsI8zzMxJluDITfwiUKA

--GS9J6KQLsI8zzMxJluDITfwiUKA
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--GS9J6KQLsI8zzMxJluDITfwiUKA--

--PDVOEl7Rh1AP90jGln1mhz7x8r9--
```

Das liefert die Ergebnisse auf dem Weg zum Endergebnis zurück. Mit anderen Worten wird also jeder Schritt festgehalten (engl. *keep*).

Jenseits von Links

Neben `Links` können Sie beliebige Metadaten über den Header-Präfix `X-Riak-Meta-` speichern. Wenn wir die Farbe (color) eines Zwingers festhalten wollen, ohne dass das für die täglichen Aufgaben von entscheidender Bedeutung wäre, könnten wir Zwinger 1 mit der Farbe Pink kennzeichnen. Der Abruf des URL-Headers (über das Flag -I) liefert Ihnen den Namen und den Wert der Metadaten zurück.

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
 -H "Content-Type: application/json" \
 -H "X-Riak-Meta-Color: Pink" \
 -H "Link: </riak/animals/polly>; riaktag=\"contains\"" \
 -d '{"room" : 101}'
```

MIME-Typen bei Riak

Riak speichert alles in Form binärkodierter Werte (genau wie normales HTTP). Der MIME-Typ gibt den Kontext der Binärdaten vor – wir haben bisher nur mit normalem Text gearbeitet. MIME-Typen werden auf dem Riak-Server gespeichert, sind aber eigentlich nur ein Flag für den Client, damit er beim Download der Binärdaten weiß, was er mit ihnen anfangen soll.

Wir wollen in unserer Hundepension Bilder unserer Gäste hinterlegen. Wir müssen nur das Flag `data-binary` des `curl`-Befehls nutzen, um ein Bild auf den Server hochzuladen, und den MIME-Typ mit `image/jpeg` angeben. Wir verlinken das mit dem `/animals/polly`-Schlüssel, damit wir wissen, wen wir da vor uns haben.

Erzeugen Sie zuerst ein Bild namens `polly_image.jpg` und legen Sie es im gleichen Verzeichnis ab, in dem Sie die `curl`-Befehle ausführen.

```
$ curl -X PUT http://localhost:8091/riak/photos/polly.jpg \
 -H "Content-type: image/jpeg" \
 -H "Link: </riak/animals/polly>; riaktag=\"photo\"" \
 --data-binary @polly_image.jpg
```

Nun besuchen Sie die URL im Web-Browser, die genau so abgerufen und dargestellt wird, wie Sie es von jedem Client/Server-Request erwarten.

```
http://localhost:8091/riak/photos/polly.jpg
```

Da wir das Bild mit `/animals/polly` verknüpft haben, können wir dem Link vom Image-Schlüssel *zu* Polly folgen, aber nicht umgekehrt. Im Gegensatz zu einer relationalen Datenbank gibt es keine „hat ein“- oder „ist ein“-Regel im Bezug auf Links. Sie verlinken in die Richtung, in die Sie sich bewegen wollen. Wenn wir glauben, dass wir in unserer Anwendung im `animals`-Bucket auf die Bilddaten zugreifen müssen, muss ein entsprechender Link in diesem Objekt existieren.

Was wir am ersten Tag gelernt haben

Wir hoffen, dass Sie Riaks Potential als flexible Speicheroption erkannt haben. Bisher haben wir nur die übliche Schlüssel/Wert-Praxis vorgestellt und einige Links eingestreut. Wenn Sie ein Riak-Schema entwickeln, müssen Sie

irgendwo zwischen einem Caching-System und PostgreSQL denken. Sie teilen Ihre Daten in verschiedene Gruppen (Buckets) auf und die Werte können stillschweigend miteinander in Beziehung stehen. Doch Sie gehen nicht so weit, sie in feine Komponenten zu normalisieren, wie Sie das bei einer relationalen Datenbank tun würden, da Riak keinerlei relationale Joins durchführt, um neue Werte zusammenzusetzen.

Tag 1: Selbststudium

Finden Sie heraus

1. Legen Sie ein Lesezeichen für die Online-Dokumentation des Riak-Projekts an und sehen Sie sich die Dokumentation der REST API an.
2. Finden Sie eine gute Liste browserunterstützter MIME-Typen.
3. Sehen Sie sich die Riak-Konfiguration in `dev/dev1/etc/app.config` an und vergleichen Sie diese mit den anderen dev-Konfigurationen.

Machen Sie Folgendes

1. Nutzen Sie `PUT`, um `animals/polly` um einen Link auf `photos/polly.jpg` zu ergänzen.
2. `POST`en Sie eine Datei mit einem MIME-Typ, den wir nicht genutzt haben (etwa `application/pdf`). Ermitteln Sie den generierten Schlüssel und sehen Sie sich diese URL mit dem Web-Browser an.
3. Legen Sie einen neuen Bucket-Typ namens *medicines* an, fügen Sie mittels `PUT` ein JPEG-Image (mit dem richtigen MIME-Typ) unter dem Schlüssel *antibiotics* hinzu und verlinken Sie es mit dem (armen kranken) Ace.

3.3 Tag 2: Mapreduce und Server-Cluster

Heute tauchen wir in das Mapreduce-Framework ein, mit dem leistungsfähigere Queries durchgeführt werden können, als sie das Schlüssel/Wert-Paradigma normalerweise bereitstellt. Wir gehen dann detaillierter auf diese Leistungsfähigkeit ein und kombinieren die Traversierung (Link-Walking) mit Mapreduce. Abschließend sehen wir uns die Server-Architektur von Riak an. Sie verwendet ein neues Server-Layout, um im Bezug auf Konsistenz oder Verfügbarkeit flexibel zu sein, selbst bei Netzwerk-Partitionen.

Populations-Skript

Wir benötigen für diesen Teil etwas mehr Daten. Zu diesem Zweck wenden wir uns einer anderen Art von Hotel zu: einem für Menschen, nicht für Tiere. Um

das zu erreichen, erzeugt ein kurzes Populations-Skript in Ruby die Daten für ein gigantisches Hotel mit 10.000 Zimmern.

Ruby ist (falls Sie nicht mit ihr vertraut sind) eine beliebte Allzweck-Programmiersprache. Sie ist sehr nützlich, um schnell Skripten auf einfache und gut lesbare Weise zu entwickeln. Mehr zu Ruby erfahren Sie in *Programming Ruby: The Pragmatic Programmer's Guide* von Dave Thomas und Andy Hunt sowie online.[4]

Sie benötigen außerdem den Ruby-Paketmanager namens RubyGems.[5] Sobald Ruby und RubyGems zur Verfügung stehen, installieren Sie den Riak-Treiber.[6] Sie könnten außerdem den json-Treiber benötigen.

```
$ gem install riak-client json
```

Jedes Zimmer unseres Hotels verfügt über eine zufällig gewählte Kapazität von ein bis acht Personen und einen zufälligen Typ wie Einzelzimmer oder Suite.

riak/hotel.rb

```
# Erzeugt viele Zimmer mit zufälligen Typen und Kapazitäten
require 'rubygems'
require 'riak'
STYLES = %w{single double queen king suite}

client = Riak::Client.new(:http_port => 8091)
bucket = client.bucket('rooms')
# 100 Stockwerke für das Gebäude erzeugen
for floor in 1..100
  current_rooms_block = floor * 100
  puts "Making rooms #{current_rooms_block} - #{current_rooms_block + 100}"
  # 100 Zimmer auf jedem Stockwerk (ein riesiges Hotel!)
  for room in 1...100
    # Eindeutige Zimmernummer als Schlüssel erzeugen
    ro = Riak::RObject.new(bucket, (current_rooms_block + room))
    # Zimmertyp zufällig wählen und Kapazität ergänzen
    style = STYLES[rand(STYLES.length)]
    capacity = rand(8) + 1
    # Zimmerinformationen als JSON-Wert speichern
    ro.content_type = "application/json"
    ro.data = {'style' => style, 'capacity' => capacity}
    ro.store
  end
end
```

```
$ ruby hotel.rb
```

4. http://ruby-lang.org
5. http://rubygems.org
6. http://rubygems.org/gems/riak-client

Wir haben nun die Hotel-Daten erzeugt, auf die wir Mapreduce anwenden wollen.

Einführung in Mapreduce

Einer von Googles nachhaltigsten Beiträgen zur Informatik ist die Popularisierung von Mapreduce als algorithmisches Framework zur parallelen Ausführung von Jobs auf mehreren Knoten. Es wird in Googles bahnbrechender Abhandlung[7] erläutert und wurde zu einem wertvollen Tool für die Ausführung von Queries bei partitionstoleranten Datenspeichern.

Mapreduce zerlegt Probleme in zwei Teile. Im ersten Teil wird eine Liste von Daten über eine `map()`-Funktion in eine andere Art von Liste umgewandelt. Im zweiten Teil wird diese zweite Liste über die `reduce()`-Funktion in einen oder mehrere Skalarwerte umgewandelt. Nach diesem Muster kann ein System Aufgaben in kleinere Komponenten zerlegen und parallel in einem Server-Cluster ausführen. Wir könnten alle Riak-Werte, die `{country : 'CA'}` enthalten, zählen, indem wir jedes passende Dokument auf `{count : 1}` abbilden (map) und dann die Summe aller counts berechnen (reduce).

Sind 5012 kanadische Werte in unseren Daten enthalten, wäre das Reduce-Ergebnis also `{count : 5012}`.

```
map = function(v) {
  var parsedData = JSON.parse(v.values[0].data);
  if(parsedData.country === 'CA')
    return [{count : 1}];
  else
    return [{count : 0}];
}

reduce = function(mappedVals) {
  var sums = {count : 0};
  for (var i in mappedVals) {
    sums[count] += mappedVals[i][count];
  }
  return [sums];
}
```

In gewisser Hinsicht ist Mapreduce das Gegenteil von dem, wie wir Queries normalerweise ausführen. Ein Ruby on Rails-System könnte Daten (über sein ActiveRecord-Interface) wie folgt abrufen:

```
# Baue Hash auf, der Raumkapazität und Raumtyp veknüpft
capacity_by_style = {}
rooms = Room.all
for room in rooms
  total_count = capacity_by_style[room.style]
```

7. http://research.google.com/archive/mapreduce.html

```
  capacity_by_style[room.style] = total_count.to_i + room.capacity
end
```

`Room.all` führt eine Query über die Datenbank aus, die der folgenden SQL-Query entspricht:

```
SELECT * FROM rooms;
```

Die Datenbank sendet alle Ergebnisse an den Applikations-Server und der führt irgendeine Operation mit den Daten durch. In diesem Fall gehen wir alle Zimmer des Hotels durch und zählen die Gesamtkapazität für jeden Zimmertyp (z. B. reicht die Kapazität der Suiten des Hotels für 448 Gäste). Das ist bei kleinen Datenmengen akzeptabel, doch wenn die Anzahl der Zimmer steigt, wird das System langsamer, da das System die Zimmer-Daten an die Applikation weitergeben muss.

Mapreduce verfolgt die genau entgegengesetzte Strategie. Statt die Daten aus der Datenbank abzurufen und auf dem Client (oder dem Applikations-Server) zu verarbeiten, übergibt Mapreduce einen Algorithmus an alle Datenbank-Knoten. Die sind dann dafür verantwortlich, ein Ergebnis zurückzuliefern. *Jedes Objekt auf dem Server wird auf irgendeinen gemeinsamen Schlüssel abgebildet ("map"), der die Daten gruppiert und alle passenden Schlüssel werden dann auf einen einzelnen Wert reduziert ("reduce").*

Für Riak bedeutet das, dass die Datenbankserver für das Map/Reduce der Werte auf jedem Knoten verantwortlich sind. Die reduzierten Werte werden an einen anderen Server weitergereicht (üblicherweise an den anfordernden Server), der die Werte weiter reduziert und das Endergebnis an den anfordernden Client (oder auch an einen Rails-Applikationsserver) zurückgibt.

Diese einfache Umkehrung ist eine leistungsfähige Möglichkeit, komplexe Algorithmen lokal auf jedem Server auszuführen und ein sehr kleines Ergebnis an den aufrufenden Client zurückzugeben. *Es ist schneller, den Algorithmus an die Daten zu senden, als die Daten an den Algorithmus zu schicken.* In Abbildung 7, *Ergebnisse der map-Funktion*, auf Seite 73 sehen wir, wie ein Bucket mit Telefonrechnungen, bei denen die Telefonnummer den Schlüssel bildet, die Berechnung der Gesamtkosten auf drei Server verteilt. Jeder Server verarbeitet dabei alle Nummern mit dem gleichen Präfix.

Die Ergebnisse der map-Funktion wird an die Reduce-Funktionen übergeben. Ein Kombination aus den Map-Ergebnissen *und* vorangegangenen Reduce-Aufrufen kann wiederum an nachfolgende Reduce-Funktionen übergeben werden. Wir kommen auf diesen Punkt später noch einmal zurück,

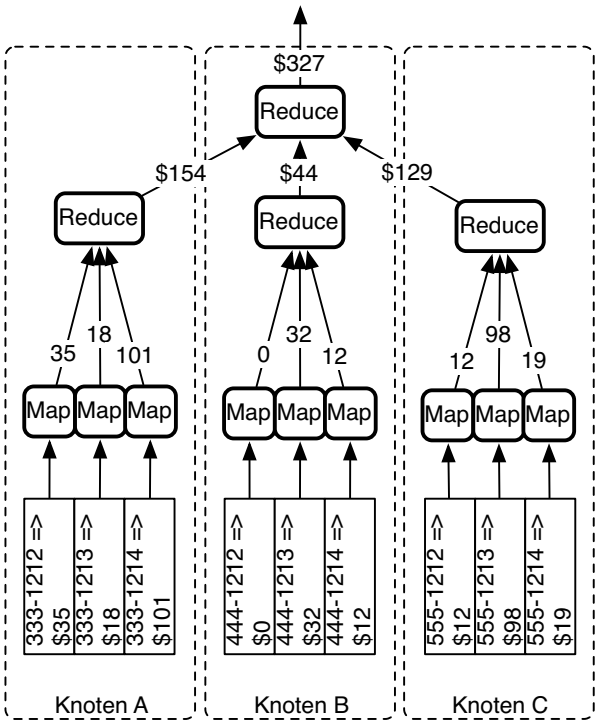


Abbildung 7: Ergebnisse der map-Funktion Die Ergebnisse der map-Funktion werden an reduce übergeben und dann an weitere Reducer.

weil sie eine wichtige (wenn auch subtile) Komponente bei der Entwicklung effektiver Mapreduce-Queries darstellt.

Mapreduce in Riak

Wir wollen Mapreduce-Funktionen für unsere Riak-Daten entwickeln, die genauso funktionieren wie der Kapazitätszähler. Ein nettes Feature bei Riaks Mapreduce ist, dass Sie die `map`-Funktion für sich ausführen können und sich alle Ergebnisse im Lauf (wenn Sie ein Reduce ausführen wollen) mit ansehen können. Wir gehen es langsam an und schauen zuerst nur die Ergebnisse für die Zimmer 101, 102 und 103.

Das Map-Setting verlangt die Sprache, die wir verwenden, sowie die Quellcode. Erst dann können wir die eigentliche JavaScript-map-Funktion angeben. (Die Funktion ist nur ein String, so dass wir alle Zeichen entsprechend kodieren müssen.)

Der cURL-Befehl @- legt fest, dass der Standard-Eingang der Console offen bleibt, bis ein `Ctrl+D` empfangen wird. Diese Daten werden im HTTP-Body an die URL gesendet, den wir an den `/mapred`-Befehl senden (Vorsicht: Die URL lautet `/mapred`, nicht `/riak/mapred`).

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
    "query":[
      {"map":{
      "language":"javascript",
      "source":
        "function(v) {
          /* Aus dem Riak-Objekt Daten extrahieren und als JSON parsen */
          var parsed_data = JSON.parse(v.values[0].data);
          var data = {};
          /* Kapazität mit Raumtyp verknüpfen */
          data[parsed_data.style] = parsed_data.capacity;
          return [data];
        }"
    }}
  ]
}

Ctrl-D
```

Der `/mapred`-Befehl verlangt gültiges JSON und hier haben wir die Form unserer Mapreduce-Befehle festgelegt. Wir wählen die drei gewünschten Räume, indem wir die „Eingangswerte“ als Array mit `[bucket, key]`-Paaren angeben. Das Wesentliche bei diesen Einstellungen findet sich aber unter dem *query*-Wert, der ein Array von JSON-Objekten akzeptiert, das Objekte enthält, bei denen *map*, *reduce* und/oder *links* die Schlüssel bilden (mehr über Links erfahren Sie später).

Damit werden die Daten abgerufen (`v.values[0].data`), die Werte als JSON-Objekt geparst (`JSON.parse(...)`) und die Kapazität (`parsed_data.capacity`) zusammen mit dem Zimmertyp (`parsed_data.style`) zurückgegeben. Das Ergebnis sieht dann etwa so aus:

```
[{"suite":6},{"single":1},{"double":1}]
```

Das sind einfach die JSON-Daten der Objekte für die Zimmer 101, 102, und 103.

Wir müssen die Daten nicht im JSON-Format ausgeben. Wir können den Wert jedes Schlüssels in das umwandeln, was wir brauchen. Wir haben nur auf die Body-Daten zugegriffen, aber wir hätten auch Metadaten, Link-Informationen, den Schlüssel oder die Daten selbst abrufen können. Letztlich ist

alles möglich, weil wir jeden Schlüssel auf irgendeinen anderen Wert abbilden.

Wenn Ihnen danach ist, können Sie alle 10000 Zimmer verarbeiten, indem Sie die input-spezifischen [bucket, key]-Arrays durch das `rooms`-Bucket ersetzen:

```
"inputs":"rooms"
```

Ein freundliche Warnung: Das liefert eine Menge Daten zurück. Abschließend ist noch erwähnenswert, dass Mapreduce-Funktionen seit Riak Version 1.0 durch ein Subsystem namens Riak Pipe verarbeitet werden. Beim alten System werden Sie das veraltete mapred_system nutzen. Als Endanwender betrifft Sie das nicht sonderlich, es wirkt sich aber spürbar auf Geschwindigkeit und Stabilität aus.

Stored Functions

Eine weitere Möglichkeit, die Riak uns bietet, ist die Speicherung der map-Funktion als Bucket-Wert. Das ist ein weiteres Beispiel dafür, den Algorithmus auf die Datenbank zu verschieben. Das ist eine Stored Procedure oder genauer gesagt, eine benutzerdefinierte Funktion – mit einer ähnlichen Philosophie, die seit Jahren bei relationalen Datenbanken verwendet wird.

```
$ curl -X PUT -H "content-type:application/json" \
http://localhost:8091/riak/my_functions/map_capacity --data @-
function(v) {
  var parsed_data = JSON.parse(v.values[0].data);
  var data = {};
  data[parsed_data.style] = parsed_data.capacity;
  return [data];
}
```

Nachdem unsere Funktion gespeichert wurde, führen wir sie aus, indem wir auf den Bucket und Schlüssel zeigen, der die Funktion enthält.

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    {"map":{
      "language":"javascript",
      "bucket":"my_functions",
      "key":"map_capacity"
    }}
  ]
}
```

Sie sollten das gleiche Ergebnis erhalten wie beim direkt integrierten JavaScript-Code.

Fest eingebaute Funktionen

Sie können einige der in Riak fest eingebauten Funktionen nutzen, die mit dem JavaScript-Objekt `Riak` verknüpft sind. Wenn Sie den folgenden Code ausführen, packen die Zimmer-Objekte die Werte in JSON und liefern sie zurück. Die Funktion `Riak.mapValuesJson` gibt Werte im JSON-Format zurück.

```
curl -X POST http://localhost:8091/mapred \
-H "content-type:application/json" --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    {"map":{
      "language":"javascript",
      "name":"Riak.mapValuesJson"
    }}
  ]
}
```

Riak stellt weitere Funktionen in einer Datei namens `mapred_builtins.js` zur Verfügung, die Sie online (oder tief im Code) finden. Sie können diese Syntax auch nutzen, um eine fest eingebaute Funktionen aufzurufen. Diesem Thema widmen wir uns morgen.

Reduce

Mapping ist nützlich, bei der Umwandlung einzelner Werte in andere individuelle Werte aber recht beschränkt. Irgendeine Form der Analyse über die Daten auszuführen, und sei es so etwas Einfaches wie das Zählen der Datensätze, verlangt einen weiteren Schritt. An dieser Stelle kommt Reduce ins Spiel.

Das SQL/Ruby-Beispiel, das wir uns in 3.3, *Reduce*, auf Seite 71 angesehen haben, zeigt, wie über alle Werte iteriert und die Gesamtkapazität für jeden Zimmertyp berechnet wird. Wir wollen genau das in unserer `reduce`-Funktion in JavaScript machen.

Der Großteil des an `/mapred` übergebenen Befehls ist gleich, doch diesmal fügen wir noch die `reduce`-Funktion hinzu.

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":"rooms",
  "query":[
    {"map":{
     "language":"javascript",
     "bucket":"my_functions",
     "key":"map_capacity"
   }},
   {"reduce":{
     "language":"javascript",
     "source":
       "function(v) {
         var totals = {};
         for (var i in v) {
           for(var style in v[i]) {
             if( totals[style] ) totals[style] += v[i][style];
             else                totals[style] = v[i][style];
           }
         }
         return [totals];
       }"
     }}
   ]
}
```

Führt man das über alle Zimmer aus, wird die Gesamtkapazität für jeden Zimmertyp zurückgegeben.

```
[{"single":7025,"queen":7123,"double":6855,"king":6733,"suite":7332}]
```

Ihre Werte werden mit diesen hier nicht genau übereinstimmen, weil wir die Zimmer-Daten zufällig generieren.

Key-Filter

Eine recht junges Riak-Feature ist das Konzept des Schlüssel-Filters (Key-Filter). Ein Key-Filter besteht aus einer Reihe von Befehlen, die den Schlüssel verarbeiten, bevor Mapreduce auf ihn angewandt wird. Damit ersparen Sie der Operation das Laden unnötiger Werte. Im folgenden Beispiel wandeln wir die Zimmernummer (als Schlüssel) in einen Integerwert um und prüfen, ob er kleiner als 1000 ist (auf einer der ersten zehn Etagen liegt; jedes Zimmer über der zehnten Etage wird ignoriert).

Reduce-Muster

Die Entwicklung einer `reduce`-Funktion wird einfacher, wenn sie dem gleichen Muster folgt wie Ihre map-Funktion. Wenn Sie einen einzelnen Wert etwa so abbilden...

```
[{name:'Eric', count:1}]
```

...dann sollte das Ergebnis von `reduce` wie folgt ausehen:

```
[{name:'Eric', count:105}, {name:'Jim', count:215}, ...]
```

Das ist nicht unbedingt notwendig, aber praktisch. Weil Reducer Daten an weitere Reducer übergeben können, wissen Sie nicht, ob die Daten bei einem `reduce`-Aufruf von einem Map-Ergebnis, einem Reduce-Ergebnis oder von beiden stammen. Folgen sie hingegen dem gleichen Objekt-Muster, spielt das keine Rolle, weil sie immer gleich sind! Anderenfalls muss Ihre `reduce`-Funktion immer den Typ der empfangenen Daten prüfen und entsprechende Entscheidungen treffen.

In unserem Mapreduce zur Bestimmung der Zimmer-Kapazitäten ersetzen wir `"inputs":"rooms"` durch den folgenden Block (der mit einem Komma enden muss):

```
"inputs":{
  "bucket":"rooms",
  "key_filters":[["string_to_int"], ["less_than", 1000]]
},
```

Zwei Dinge sind interessant: Die Query ist wesentlich schneller (weil wir nur die benötigten Werte verarbeitet haben) und die Ergebnismenge ist kleiner (weil wir nur die ersten 10 Etagen berücksichtigen).

Mapreduce ist ein mächtiges Werkzeug, um Daten zu bündeln und eine allumfassende Analyse dieser Daten durchzuführen. Wir werden dieser Idee in diesem Buch immer wieder begegnen, doch das Konzept ist immer das gleiche. Riak optimiert die grundlegende Form des Mapreduce ein wenig, indem es das Konzept um Links erweitert.

Link-Walking mit Mapreduce

Gestern haben wir Sie in das Link-Walking eingeführt. Heute wollen wir das gleiche in Verbindung mit Mapreduce tun. Der query-Abschnitt hat neben `map` und `reduce` noch einen zusätzlichen Wert: `link`.

Wir kehren noch einmal zu unserem `cages`-Bucket des gestrigen Hundepension-Beispiels zurück und schreiben einen Mapper, der nur Zwinger 2 (in dem Ace the dog sitzt) zurückgibt.

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":{
    "bucket":"cages",
    "key_filters":[["eq", "2"]]
  },
  "query":[
    {"link":{
     "bucket":"animals",
```

```
      "keep":false
    }},
    {"map":{
      "language":"javascript",
      "source":
        "function(v) { return [v]; }"
    }}
  ]
}
```

Obwohl wir die Mapreduce-Query auf den `cages`-Bucket angewandt haben, werden die Informationen zu *Ace the dog* zurückgegeben, weil wir auf Zwinger 2 verlinkt haben.

```
[{
  "bucket":"animals",
  "key":"ace",
  "vclock":"a85hYGBgzmDKBVIsrDJPfTKYEhnzWBn6LfiP80GFWVZay0KF5yGE2ZqTGPmCLiJLZAEA",
  "values":[{
    "metadata":{
      "Links":[],
      "X-Riak-VTag":"4JVlDcEYRIKuyUhw8OUYJS",
      "content-type":"application/json",
      "X-Riak-Last-Modified":"Tue, 05 Apr 2011 06:54:22 GMT",
      "X-Riak-Meta":[]},
    "data":"{\"nickname\" : \"The Wonder Dog\", \"breed\" : \"German Shepherd\"}"
  }]
}]
```

Sowohl Daten als auch Metadaten (die normalerweise im HTTP-Header zurückgegeben werden) erscheinen im values-Array.

Wenn man Map, Reduce, Link-Walking und Key-Filter miteinander verknüpft, können beliebige Queries über ein breites Spektrum von Riak-Schlüsseln ausgeführt werden. Das ist natürlich deutlich effizienter, als alle Daten mit einem Client abzurufen. Da solche Queries üblicherweise simultan auf mehreren Servern ausgeführt werden, sollten Sie auch nicht lange warten müssen. Doch wenn Sie wirklich nicht warten wollen, können Sie der Query noch eine weitere Option übergeben: `timeout`. Setzen Sie `timeout` auf einen Wert in Millisekunden (voreingestellt ist `"timeout": 60000`, also 60 Sekunden), und die Query wird abgebrochen, wenn sie nicht innerhalb der festgelegten Zeit abgeschlossen wird.

Konsistenz und Haltbarkeit

Die Riak-Serverarchitektur vermeidet einen „Single Point of Failure“ (alle Knoten sind gleichberechtigt) und erlaubt das beliebige Verkleinern und Vergrößern des Clusters. Das ist bei großen Systemen wichtig, weil Ihre Daten-

bank auch dann noch verfügbar bleibt, wenn mehrere Knoten ausfallen oder aus anderen Gründen nicht reagieren.

Die Verteilung von Daten auf mehrere Server ist von Natur aus problematisch. Soll Ihre Datenbank weiterlaufen, wenn sie vom Netzwerk getrennt wird (d. h., wenn Nachrichten verloren gehen), müssen Sie einen Kompromiss eingehen. Sie können entweder für Server-Requests *verfügbar* bleiben oder Requests ablehnen und die *Konsistenz* Ihrer Daten sicherstellen. Es ist nicht möglich, eine verteilte Datenbank aufzubauen, die vollständig konsistent, verfügbar und partitionstolerant ist. Sie können nur zwei Eigenschaften garantieren (partitionstolerant und konsistent, partitionstolerant und verfügbar oder konsistent und verfügbar, aber nicht verteilt). Das ist als CAP-Theorem bekannt (Consistency, Availability, Partition tolerance). Details finden Sie unter Anhang 2, *Das CAP-Theorem*, auf Seite 347, im Wesentlichen handelt es sich aber um ein Problem im Systemdesign.

Doch das Theorem hat ein Schlupfloch. Tatsache ist, dass man nicht zu *jedem Zeitpunkt* konsistent, verfügbar und partitionstolerant sein kann. Riak macht sich diese Tatsache zunutze. Es erlaubt Ihnen, Request-bezogen Verfügbarkeit gegen Konsistenz einzutauschen. Wir wollen uns zuerst ansehen, wie Riak seine Server clustert, und zeigen dann, wie man Lese- und Schreiboperationen im Cluster abstimmt.

Der Riak-Ring

Riak teilt seine Server in Partitionen auf, die mit einer 160-Bit-Zahl (2^160) gekennzeichnet sind. Das Riak-Team stellt diesen großen Integerwert gerne in Form eines Kreises dar, den sie den *Ring* nennen. Wird ein Schlüssel einer Partition zugeordnet, hilft der Ring dabei festzulegen, welche Riak-Server den Wert speichern.

Eine der ersten Entscheidungen, die wir beim Aufsetzen eines Riak-Clusters treffen müssen, besteht darin, die Anzahl der Partitionen festzulegen. Nehmen wir an, Sie haben 64 Partitionen (Riaks Standardwert). Wenn Sie diese 64 Partitionen auf drei Knoten (Server) verteilen, weist Riak jedem Knoten 21 oder 22 Partitionen zu (64 / 3). Jede Partition wird als virtueller Knoten bezeichnet, kurz *vnode*. Beim Booten gehen alle Riak-Services den Ring durch und beanspruchen nacheinander Partitionen, bis alle vnodes zugewiesen wurden (siehe Abbildung 8, *“Der Riak-Ring” mit 64 vnodes, verteilt auf drei physikalische Knoten*, auf Seite 81).

Node A verwaltet die vnodes 1, 4, 7, 10...63. Diese vnodes werden auf Partitionen im 160-Bit-Ring abgebildet. Wenn Sie sich den Status der drei Entwicklungs-Server ansehen (erinnern Sie sich an das `curl -H "Accept: text/`

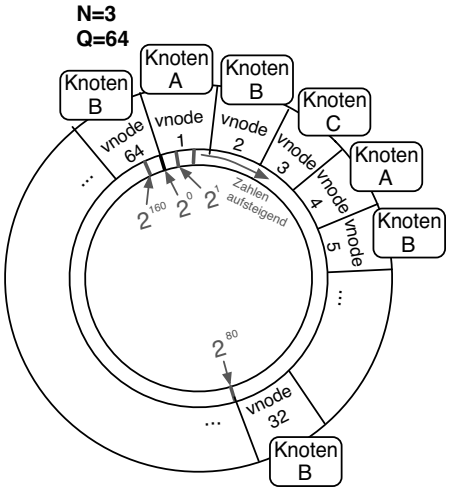
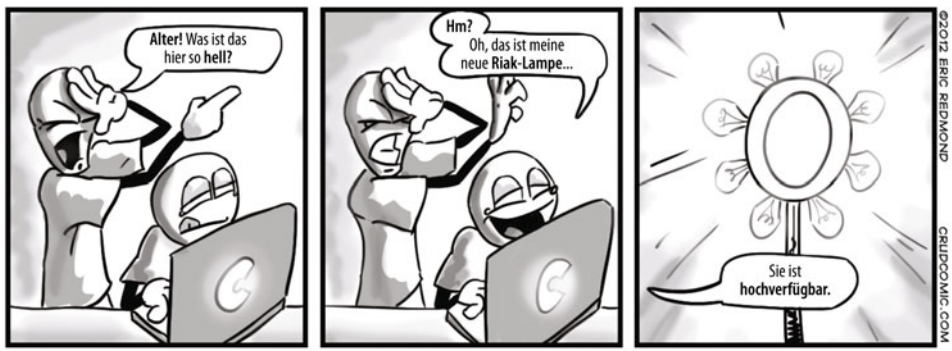


Abbildung 8: "Der Riak-Ring" mit 64 vnodes, verteilt auf drei physikalische Knoten

`plain" http://localhost:8091/stats` von gestern), finden Sie eine Zeile wie diese:

```
"ring_ownership": \
"[{'dev3@127.0.0.1',21},{'dev2@127.0.0.1',21},{'dev1@127.0.0.1',22}]"
```

Die zweite Zahl jedes Objekts gibt die Anzahl der vnodes des jeweiligen Knotens an. Das summiert sich auf insgesamt 64 (21 + 21 + 22) vnodes.



Jede vnode repräsentiert einen Teilbereich der gehashten Schlüssel. Wenn wir die Zimmer-Daten für den Schlüssel *101* einfügen, könnte er im Bereich von vnode 2 landen, d. h., das Schlüssel/Wert-Objekt würde auf Knoten B

abgelegt werden. Der Vorteil ist, dass Riak nur den Schlüssel abfragen muss, um die entsprechende vnode zu finden, wenn wir wissen wollen, auf welchem Server der Schlüssel liegt. Genauer gesagt, wandelt Riak den Hash in eine Liste potentieller vnodes um und verwendet die erste.

Nodes/Writes/Reads

Riak ermöglicht die Kontrolle über Lese- und Schreiboperationen im Cluster durch die Veränderung dreier Werte: N, W und R. *N* ist die Anzahl der Knoten, auf die eine Schreiboperation repliziert wird, d. h. die Anzahl der Kopien im Cluster. *W* ist die Anzahl der Knoten, an die erfolgreich geschrieben werden muss, bevor eine Erfolgsmeldung zurückgegeben wird. Ist W kleiner als N, wird die Schreiboperation selbst dann als erfolgreich betrachtet, wenn Riak den Wert noch kopiert. *R* ist schließlich die Anzahl der Knoten, die benötigt werden, um einen Wert erfolgreich einlesen zu können. Ist R größer als die Zahl der verfügbaren Kopien, schlägt der Request fehl.

Sehen wir uns das etwas genauer an.

Wenn wir ein Objekt in Riak einfügen, haben wir die Möglichkeit, den Wert über mehrere Knoten zu replizieren. Der Vorteil ist, dass eine Kopie auf einem anderen Server zur Verfügung steht, wenn ein Server abstürzt. Die Bucket-Eigenschaft (Property) `n_val` enthält die Anzahl von Knoten, auf die ein Wert repliziert werden soll (der N-Wert). Voreingestellt ist `3`. Wir können die Eigenschaften eines Buckets ändern, indem wir einen neuen Wert in das `props`-Objekt schreiben. Nachfolgend setzen wir bei animals den `n_val` auf 4:

```
$ curl -X PUT http://localhost:8091/riak/animals \
 -H "Content-Type: application/json" \
 -d '{"props":{"n_val":4}}'
```

N ist einfach die Anzahl von Knoten, die *schlussendlich* den richtigen Wert enthalten. Das bedeutet nicht, dass wir darauf warten müssen, bis der Wert auf *alle* Knoten kopiert wurde, um zurückkehren zu können. Manchmal soll der Client sofort zurückkehren und Riak die Daten im Hintergrund replizieren lassen. Manchmal wollen wir aber auch warten, bis Riak (nur um sicherzugehen) auf alle *N* repliziert hat, bevor wir zurückkehren.

Wir können den Wert *W* auf die Anzahl erfolgreicher Schreiboperationen setzen, bevor die Operation als erfolgreich abgeschlossen wird. Auch wenn wir die Daten schlussendlich an vier Knoten schreiben, können wir W auf 2 setzen und die Schreiboperation kehrt nach nur zwei Kopien zurück. Die beiden restlichen werden im Hintergrund repliziert.

```
curl -X PUT http://localhost:8091/riak/animals \
 -H "Content-Type: application/json" \
 -d '{"props":{"w":2}}'
```

Schließlich können wir noch den *R*-Wert setzen. *R* ist die Anzahl der Knoten, die gelesen werden müssen, um die Leseoperation erfolgreich abzuschließen. Sie können einen Standard-*R*-Wert auf die gleiche Weise festlegen, wie wir das für `n_val` und `w` getan haben.

```
curl -X PUT http://localhost:8091/riak/animals \
 -H "Content-Type: application/json" \
 -d '{"props":{"r":3}}'
```

Aber Riak bietet eine noch flexiblere Lösung. Sie können die Zahl der zu lesenden Knoten bei *jedem Request* in der URL als *r*-Parameter angeben.

```
curl http://localhost:8091/riak/animals/ace?r=3
```

Sie werden sich fragen, warum man etwas von mehr als einem Knoten lesen sollte. Letzten Endes werden die von uns geschriebenen Werte auf *N* Knoten repliziert und wir können jeden von ihnen lesen. Die Idee lässt sich visuell einfacher vermitteln.

Nehmen wir an, wir legen unsere NRW-Werte mit `{"n_val":3, "r":2, "w":1}` (siehe Abbildung 9, *Schlussendliche Konsistenz: W+R <= N*) fest. Damit reagiert unser System bei Schreiboperationen besser, weil nur an einen Knoten geschrieben werden muss, bevor zurückgekehrt wird. Doch es besteht die Möglichkeit, dass eine andere Operation etwas lesen will, bevor die Knoten die Chance zur Synchronisation hatten. Selbst wenn wir von zwei Knoten lesen, besteht die Möglichkeit, einen alten Wert empfangen zu haben.

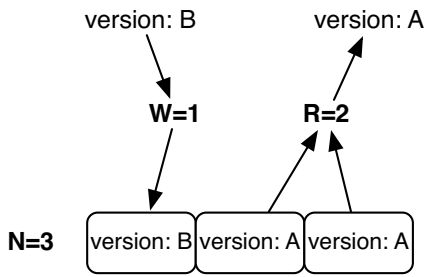


Abbildung 9: Schlussendliche Konsistenz: W+R <= N

Eine Möglichkeit sicherzustellen, dass der aktuellste Wert eingelesen wird, besteht darin, W=N und R=1 wie folgt zu setzen: `{"n_val":3, "r":1, "w":3}` (siehe Abbildung 10, *Konsistenz beim Schreiben: W=N, R=1*). Im Wesentlichen ist das das, was relationale Datenbanken machen. Sie stellen die Konsistenz her, indem sie sicherstellen, dass eine Schreiboperation abgeschlossen ist, bevor sie zurückkehren. Wir können natürlich schneller lesen, da wir nur auf einen Knoten zugreifen müssen. Schreiboperationen können dadurch aber stark verlangsamt werden.

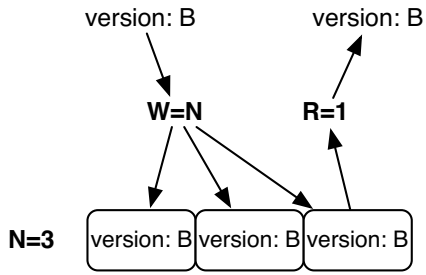


Abbildung 10: Konsistenz beim Schreiben: W=N, R=1

Oder Sie können einen Wert schreiben, aber alle einlesen. Dazu setzen Sie W=1 und R=N wie folgt: `{"n_val":3, "r":3, "w":1}` (siehe Abbildung 11, *Konsistenz beim Lesen: W=1, R=N*). Zwar könnten Sie auch einige ältere Werte einlesen, aber der aktuellste ist garantiert auch darunter. Sie müssen nur herausfinden, welcher das ist (mit Hilfe einer Vektoruhr, die wir morgen behandeln). Natürlich kehren Sie damit das Problem von eben um, d. h., jetzt werden die Leseoperationen langsamer.

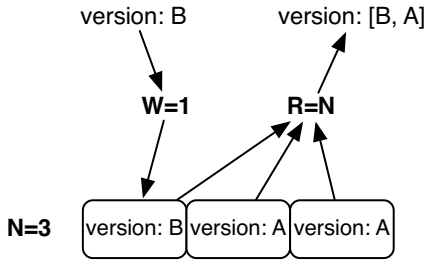


Abbildung 11: Konsistenz beim Lesen: W=1, R=N

Oder Sie setzen W=2 und R=2: `{"n_val":3, "r":2, "w":2}` (siehe Abbildung 12, *Konsistenz beim Quorum: W+R > N*). Auf diese Weise müssen Sie nur an mehr als die Hälfte der Knoten schreiben und von mehr als der Hälfte lesen, haben aber den Vorteil der Konsistenz, während Sie die Zeitverzögerungen zwischen Lese- und Schreiboperationen aufteilen. Das nennt man *Quorum* und entspricht der Minimalmenge für konsistente Daten.

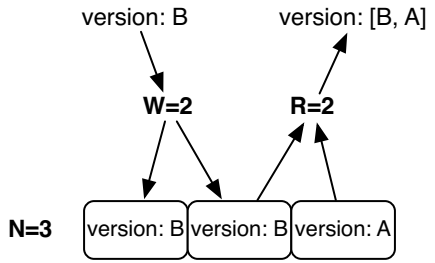


Abbildung 12: Konsistenz beim Quorum: W+R > N

Sie können R und W auf jeden Wert zwischen 1 und N setzen, werden sich üblicherweise aber an einen, alle oder ein Quorum halten. Das sind so gängige Werte, dass R und W sie repräsentierende Stringwerte akzeptieren, die in der folgenden Tabelle definiert sind:

Begriff	Definition
One	Einfach der Wert 1. W oder R auf diesen Wert zu setzen, bedeutet, dass nur ein Knoten antworten muss, damit der Request abgeschlossen wird.
All	Entspricht dem Wert von N. Wird W oder R auf diesen Wert gesetzt, müssen alle Knoten antworten.
Quorum	Entspricht N/2+1. Wird W oder R auf diesen Wert gesetzt, müssen die meisten Knoten geantwortet haben, damit der Request abgeschlossen wird.
Default	Welcher Wert für W oder R für das Bucket auch eingestellt sein mag. Standardmäßig 3.

Die obigen Werte sind nicht nur gültige Bucket-Eigenschaften, sondern können auch als Query-Parameter verwendet werden.

```
curl http://localhost:8091/riak/animals/ace?r=all
```

Wenn Sie das Lesen aller Knoten verlangen, besteht die Gefahr, dass Riak dem Request nicht nachkommen kann, wenn ein Server unten ist. Als kleines Experiment wollen wir den dritten Server herunterfahren.

```
$ dev/dev3/bin/riak stop
```

Wenn Sie nun versuchen, von allen Knoten zu lesen, sind die Chancen groß, dass der Request fehlschlägt (wenn nicht, versuchen Sie auch dev2 herunterzufahren, oder fahren Sie dev1 herunter und lesen Sie von Port 8092 oder 8093; wir können nicht kontrollieren, an welche vnodes Riak schreibt).

```
$ curl -i http://localhost:8091/riak/animals/ace?r=all
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Thu, 02 Jun 2011 17:18:18 GMT
Content-Type: text/plain
Content-Length: 10

not found
```

Wenn Ihre Anfrage nicht verarbeitet werden kann, wird der Code 404 (Objekt nicht gefunden) zurückgegeben, was aus Sicht des Requests einen Sinn ergibt. Das Objekt kann nicht gefunden werden, weil es nicht genug Kopien gibt, um den URL-Request zu erfüllen. Das ist natürlich keine gute Sache, weshalb Riak eine *Lese-Reparatur* (read repair) anstößt, die *N* Replikationen des Schlüssels über die noch verfügbaren Server verteilt. Wenn Sie erneut auf die gleiche URL zugreifen, erhalten Sie die Schlüsselwerte anstelle eines weiteren 404-Codes zurück. Die Riak-Dokumentation enthält ein ausgezeichnetes Beispiel[8] in Erlang.

Doch die sichere Variante verlangt ein Quorum (Daten von den meisten, aber nicht allen vnodes).

```
curl http://localhost:8091/riak/animals/polly?r=quorum
```

Solange Sie an ein Quorum schreiben (das Sie für jede Schreiboperation erzwingen können), sollten Ihre Lese-Operationen konsistent sein. Ein anderer Wert, den Sie direkt setzen können, ist W. Wenn Sie nicht warten wollen, bis Riak an alle Knoten geschrieben hat, setzen Sie W auf 0 (null), was so viel bedeutet, wie dass Sie Riak vertrauen und es direkt zurückkehren kann.

8. http://wiki.basho.com/Replication.html

```
curl -X PUT http://localhost:8091/riak/animals/jean?w=0 \
-H "Content-Type: application/json"
-d '{"nickname" : "Jean", "breed" : "Border Collie"}' \
```

Trotz all dieser Möglichkeiten werden Sie üblicherweise bei den Standardwerten bleiben, solange Sie keinen triftigen Grund haben, das nicht zu tun. Bei Logs können Sie W=0 setzen und Sie können W=N und R=1 für selten zu schreibende Daten verwenden, um ein extraschnelles Lesen zu ermöglichen.

Schreiben und dauerhaftes Schreiben

Wir haben Ihnen bisher etwas verheimlicht. Schreiboperationen sind bei Riak nicht notwendigerweise dauerhaft, d. h., sie werden nicht sofort auf die Festplatte geschrieben. Auch wenn eine *Knoten-Schreiboperation* erfolgreich war, kann es noch zu einem Fehler kommen, bei dem ein Knoten Daten verliert. Selbst bei W=N können Server ausfallen und Daten verlieren. Eine Schreiboperation wird für einen Moment im Speicher gepuffert, bevor sie auf der Platte gespeichert wird und dieser Bruchteil einer Millisekunde stellt eine Gefahr dar.

Das sind schlechte Nachrichten. Die gute Nachricht lautet, dass Riak eine Einstellung namens DW für *durable write*, also „dauerhaftes Schreiben“ kennt. Das ist langsamer, aber reduziert noch einmal das Risiko, weil Riak nicht zurückkehrt, bis das Objekt auf die Platten der angegebenen Anzahl von Knoten geschrieben hat. Genau wie bei Schreiboperationen können Sie diese Eigenschaft für ein Bucket festlegen. Nachfolgend setzen wir `dw` auf `one`, um sicherzugehen, dass wenigstens ein Knoten die Daten gespeichert hat.

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"dw":"one"}}'
```

Wenn Sie wollen, können Sie das auch bei jeder Schreiboperation festlegen, indem Sie den Query-Parameter `dw` in der URL nutzen.

Eine freundliche Warnung

Der Versuch, etwas an einen nicht verfügbaren Knoten zu schreiben, endet immer noch mit einem „204 No Content“. Das liegt daran, dass Riak den Wert an einen benachbarten Knoten schreibt, der die Daten vorhält, bis er sie an den nicht verfügbaren Knoten übergeben kann. Das ist auf kurze Sicht ein großartiges Sicherheitsnetz, weil ein anderer Riak-Knoten einspringt, wenn ein Server unten ist. Wenn alle Requests für Server A nun aber an Server B

weitergeleitet werden, muss Server B plötzlich mit der doppelten Last fertig werden. Es besteht die Gefahr, dass B deshalb zusammenbricht, was sich auch auf C, D und so weiter auswirkt. Man bezeichnet das als *kaskadierenden Fehler*, der zwar selten, aber möglich ist. Betrachten Sie das als freundliche Warnung, nicht jeden Riak-Server an der Leistungsgrenze zu betreiben, da Sie nicht wissen, wann er mal einspringen muss.

Was wir am zweiten Tag gelernt haben

Heute haben Sie der wichtigsten Merkmale von Riak kennengelernt: die mächtige Mapreduce-Methode und die flexible Server-Clusterung. Mapreduce wird von vielen der anderen Datenbanken in diesem Buch verwendet. Wenn Sie also noch Fragen dazu haben, sollten Sie sich den ersten Teil von Tag 2 noch einmal durchlesen und sich die Riak-Dokumentation[9] sowie die Wikipedia-Artikel[10] ansehen.

Tag 2: Selbststudium

Finden Sie heraus

1. Lesen Sie die Riak-Online-Dokumentation zu Mapreduce.
2. Finden Sie das Riak-Repository mit beigesteuerten Funktionen, das viele vorgefertigte Mapreduce-Funktionen enthält.
3. Finden Sie die Online-Dokumentation mit einer vollständigen Liste von Schlüssel-Filtern. Sie reichen von der Umwandlung von Strings in Großbuchstaben (`to_upper`) über das Auffinden numerischer Werte innerhalb eines bestimmten Wertebereichs bis hin zu einfachen Stringvergleichen per Levenshtein-Distanz und logischen UND/ODER/NICHT-Operationen.

Machen Sie Folgendes

1. Schreiben Sie map- und reduce-Funktionen für das `rooms`-Bucket, die die Gesamtkapazität je Etage berechnen.
2. Erweitern Sie diese Funktionen um einen Filter, der die Kapazitätsberechnung auf die Etagen 42 und 43 beschränkt.

3.4 Tag 3: Konflikte auflösen und Riak erweitern

Heute lernen wir einige der Ecken und Kanten von Riak kennen. Wie wir gesehen haben, ist Riak eine einfache Schlüssel/Wert-Datenbank in einem

9. http://wiki.basho.com/MapReduce.html
10. http://en.wikipedia.org/wiki/MapReduce

Cluster von Servern. Wenn mit mehreren Knoten gearbeitet wird, kann es zu Datenkonflikten kommen und manchmal müssen wir sie auflösen. Riak stellt Mechanismen bereit, mit denen sich herausfinden lässt, welche Schreiboperationen zuletzt erfolgt sind. Es verwendet dazu Vektoruhren (vector clocks) und die sog. Sibling Resolution.

Wir sehen uns auch an, wie man eingehende Daten über Pre- und Post-Commit-Hooks validieren kann. Und wir werden Riak in unsere persönliche Suchmaschine verwandeln. Dazu verwenden wir Riak-Search (mit dem SOLR-Interface) und sorgen mit Sekundärindizes für schnellere Queries.

Konflikte auflösen mit Vektoruhren

Eine *Vektoruhr*[11] ist ein Token, das verteilte Systeme wie Riak nutzen, um die richtige Reihenfolge kollidierender Schlüssel/Wert-Updates zu erhalten. Es ist wichtig nachzuhalten, in welcher Reihenfolge Updates auftreten, weil mehrere Clients sich mit verschiedenen Servern verbinden können, und während ein Client den einen Server aktualisiert, kommen von einem anderen Client Updates bei einem anderen Server an (man kann nicht kontrollieren, an welchen Server was geschrieben wird).

Nun könnten Sie denken: „Gib allen Werten einen Zeitstempel mit und der höchste Wert macht das Rennen“, doch bei einem Server-Cluster funktioniert das nur, wenn alle Server-Uhren perfekt synchron laufen. Riak stellt keine solchen Anforderungen, da die Synchronisation der Uhren bestenfalls schwierig und in vielen Fällen sogar unmöglich ist. Die Verwendung eines zentralisierten Uhrensystems wäre der Riak-Philosophie ein Gräuel, da es einen Single Point of Failure darstellt.

Mit Vektoruhren können Sie markieren, welches Schlüssel/Wert-Event (Erzeugen, Aktualisieren, Löschen) mit welchem Client in welcher Reihenfolge aufgetreten ist. Auf diese Weise können Clients – oder der Applikationsentwickler – entscheiden, wer im Falle eines Konflikts gewinnt. Wenn Sie mit Versionskontrollsystemen wie Git oder Subversion vertraut sind, dann ist das der Auflösung von Versionskonflikten (wenn zwei Leute die gleiche Datei ändern) nicht unähnlich.

Vektoruhren in der Theorie

Nehmen wir an, unsere Hundepension läuft so gut, dass wir im Bezug auf die Klientel etwas wählerischer sein müssen. Sie haben dazu drei Hundeexperten engagiert, die Ihnen dabei helfen zu entscheiden, welcher neue Hund

11. http://en.wikipedia.org/wiki/Vector_clock

gut passen würde. Die Experten bewerten jeden Hund mit einem Wert von 1 (kein guter Kandidat) bis 4 (der perfekte Kandidat). Unsere drei Hundeexperten – wir nennen Sie Bob, Jane und Rakshith – müssen eine einstimmige Entscheidung fällen.

Jeder Experte hat einen eigenen Client, der mit dem Datenbankserver verbunden ist, und jeder Client trägt eine eindeutige Client-ID in jeden Request ein. Diese Client-ID wird genutzt, um die Vektoruhr aufzubauen, und hält gleichzeitig den Client im Objekt-Header nach, der zuletzt ein Update durchgeführt hat. Wir sehen uns zuerst ein einfaches Pseudocode-Beispiel an und wollen dieses Beispiel dann später in Riak umsetzen.

Bob erzeugt das Objekt zuerst und gibt die respektable Bewertung 3 für einen neuen Hund namens Bruiser ab. Die Vektoruhr kodiert seinen Namen und die Version 1.

```
vclock: bob[1]
value:  {score : 3}
```

Jane ruft diesen Datensatz ab und bewertet Bruiser mit 2. Die für ihr Update erzeugte vclock erfolgte nach Bobs Eintrag, weshalb ihre erste Version an das Ende des Vektors angehangen wird.

```
vclock: bob[1], jane[1]
value:  {score : 2}
```

Gleichzeitig hat Rakshith die von Bob (und nicht von Jane) erzeugte Version abgerufen. Er liebt Bruiser und bewertet ihn mit einer 4. Genau wie bei Jane wird sein Client-Name an das Ende der Vektoruhr als Version 1 angehangen.

```
vclock: bob[1], rakshith[1]
value:  {score : 4}
```

Später an diesem Tag sieht sich Jane (als Leiterin der Bewertungsrunde) die Bewertungen nochmal an. Da Rakshiths Update-Vektor nicht nach Jane, sondern gleichzeitig erfolgte, stehen die beiden Updates in einem Konflikt, der aufgelöst werden muss. Sie erhält beide Werte und es liegt an ihr, den Konflikt aufzulösen.

```
vclock: bob[1], jane[1]
value:  {score : 2}
---
vclock: bob[1], rakshith[1]
value:  {score : 4}
```

Sie wählt einen Mittelwert und aktualisiert die Bewertung mit dem Wert 3.

```
vclock: bob[1], rakshith[1], jane[2]
value:  {score : 3}
```

Nachdem der Konflikt aufgelöst wurde, erhält jeder, der danach auf die Daten zugreift, den aktuellsten Wert zurück.

Vektoruhren in der Praxis

Lassen Sie uns das obige Beispiel-Szenario mit Riak durchgehen.

Bei diesem Beispiel wollen wir alle in Konflikt stehenden Versionen sehen, damit wir sie manuell auflösen können. Wir halten mehrere Versionen vor, indem wir die Eigenschaft `allow_mult` im `animals`-Bucket setzen. Schlüssel mit mehreren Werten werden *Geschwister-(sibling-)*Werte genannt.

```
$ curl -X PUT http://localhost:8091/riak/animals \
  -H "Content-Type: application/json" \
  -d '{"props":{"allow_mult":true}}'
```

Hier trägt Bob Bruiser mit der Bewertung 3 und der Client-ID *bob* in das System ein.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
  -H "X-Riak-ClientId: bob" \
  -H "Content-Type: application/json" \
  -d '{"score" : 3}'
```

Jane und Rakshith rufen beide Bruisers Daten ab, die von Bob angelegt wurden (die Header-Informationen sind wesentlich umfassender; wir beschränken uns hier auf die Vektoruhr).

Beachten Sie, dass Riak Bobs vclock kodiert hat. Dahinter verbirgt sich aber ein Client und eine Version (sowie ein Zeitstempel, d. h., Ihre Werte werden andere sein als hier zu sehen).

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true
X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==

{"score" : 3}
```

Jane aktualisiert die Bewertung auf 2 und fügt die Vektoruhr mit ein, die sie mit Bobs Version empfangen hat. Damit signalisiert sie Riak, dass ihr Wert Bobs Version aktualisiert.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
  -H "X-Riak-ClientId: jane" \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==" \
  -H "Content-Type: application/json" \
  -d '{"score" : 2}'
```

Da Jane und Rakshith Bobs Daten gleichzeitig abgerufen haben, sendet er sein Update (mit dem Wert 4) ebenfalls mit Bobs Vektoruhr.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
  -H "X-Riak-ClientId: rakshith" \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==" \
  -H "Content-Type: application/json" \
  -d '{"score" : 4}'
```

Wenn Jane die Bewertung erneut abruft, sieht sie nicht wie erwartet einen Wert, sondern einen HTTP-Code für mehrere Wahlmöglichkeiten und einen Body mit zwei „Geschwister-“ (Sibling-)Werten.

```
$ curl http://localhost:8091/riak/animals/bruiser?return_body=true
Siblings:
637aZSiky628lx1YrstzH5
7F85FBAIW8eiD9ubsBAeVk
```

Riak speichert diese Versionen in einem Multipart-Format, d. h., sie kann das gesamte Objekt abrufen, indem sie diesen MIME-Typ akzeptiert.

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true \
  -H "Accept: multipart/mixed"

HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA
Content-Type: multipart/mixed; boundary=1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Length: 409

--1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Type: application/json
Etag: 637aZSiky628lx1YrstzH5

{"score" : 4}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Type: application/json
Etag: 7F85FBAIW8eiD9ubsBAeVk

{"score" : 2}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu--
```

Beachten Sie, dass die gezeigten „Geschwister“ HTTP-Etags (die Riak vtags nennt) für spezifische Werte sind. Nebenbei bemerkt können Sie den vtag-Parameter in der URL nutzen, um nur diese Version abzurufen: `curl http://localhost:8091/riak/animals/bruiser?vtag=7F85FBAIW8eiD9ubsBAeVk` gibt `{"score" : 2}` zurück. Janes Aufgabe ist es nun, diese Information zu nut-

zen, um ein sinnvolles Update vorzunehmen. Sie entscheidet sich für den Durchschnitt der beiden Bewertungen und aktualisiert den Wert mit 3. Dabei verwendet Sie die angegebene Vektoruhr, um den Konflikt aufzulösen.

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser?return_body=true \
-H "X-Riak-ClientId: jane" \
-H "X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA" \
-H "Content-Type: application/json" \
-d '{"score" : 3}'
```

Wenn Bob und Rakshith jetzt `bruisers` Informationen abrufen, erhalten sie die aufgelöste Bewertung.

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...CpQmAkonCcHFM4CAA==

{"score" : 3}
```

Alle zukünftigen Abfragen liefern die Bewertung 3 zurück.

Zeit wächst weiter an

Sie werden bemerkt haben, dass die Vektoruhr anwächst, während die Clients Werte aktualisieren. Das ist ein fundamentales Problem von Vektoruhren, das auch die Riak-Entwickler erkannt haben. Sie haben Vektoruhren dahingehend erweitert, dass sie mit der Zeit bereinigt werden, um ihre Größe klein zu halten. Das Tempo, in dem Riak alte Vektoruhr-Werte aussortiert, wird über Bucket-Eigenschaften festgelegt, die man sich (zusammen mit allen anderen Eigenschaften) ansehen kann, indem man das Bucket abruft.

```
$ curl http://localhost:8091/riak/animals
```

Sie werden einige der folgenden Eigenschaften sehen, die festlegen, wie Riak die Uhr beschneidet, bevor sie zu groß wird.

```
"small_vclock":10,"big_vclock":50,"young_vclock":20,"old_vclock":86400
```

small_vclock und *big_vclock* bestimmen die minimale und maximale Länge des Vektors, während *young_vclock* und *old_vclock* das minimale und maximale Alter einer vclock festlegen.

Mehr über Vektoruhren und ihre Beschneidung finden Sie online.[12]

12. http://wiki.basho.com/Vector-Clocks.html

Pre-/Post-Commit-Hooks

Riak kann die Daten über sog. Hooks vor oder nach dem Speichern eines Objekts umwandeln. Pre- und Post-Commit-Hooks sind einfache JavaScript- (oder Erlang-)Funktionen, die vor oder nach dem Commit ausgeführt werden. Pre-Commit-Funktionen können das eingehende Objekt in irgendeiner Form modifizieren (und sogar einen Fehler auslösen), während Post-Commits auf einen erfolgreichen Commit reagieren (etwa in einen Log schreiben oder eine E-Mail senden) können.

Jeder Server besitzt eine `app.config`-Datei, in der die Lage eigener JavaScript-Codes festgelegt wird. Öffnen Sie diese Datei zuerst für Server dev1 unter `dev/dev1/etc/app.config`, und suchen Sie die Zeile, die `js_source_dir` enthält. Tragen Sie dort den gewünschten Verzeichnispfad ein. Beachten Sie, dass diese Zeile mit einem %-Zeichen auskommentiert sein kann, d. h., Sie müssen die Zeile evtl. zuerst aktivieren, indem Sie dieses Zeichen löschen. Unsere Zeile sieht wie folgt aus:

```
{js_source_dir, "~/riak/js_source"},
```

Sie müssen diese Änderung dreimal vornehmen, einmal für jeden dev-Server.

Wir wollen einen Validator entwickeln, der pre-commit ausgeführt wird. Er parst die Daten und stellt sicher, dass es eine Bewertung gibt und dass sie im Bereich von 1 bis 4 liegt. Wird eines dieser Kriterien nicht erfüllt, wird ein Fehler ausgelöst und unser Validator gibt ein JSON-Objekt zurück, das aus `{"fail" : message}` besteht, wobei `message` die Nachricht ist, die wir an den Benutzer zurückgeben wollen. Erfüllen die Daten die Bedingungen, geben wir nur das Objekt zurück und Riak speichert den Wert.

riak/my_validators.js

```
function good_score(object) {
  try {
    /* Daten aus dem Riak-Objekt ziehen und als JSON parsen */
    var data = JSON.parse( object.values[0].data );
    /* Fehler, wenn keine Bewertung vorliegt */
    if( !data.score || data.score === '' ) {
      throw( 'Score is required' );
    }
    /* Fehler, wenn Bewertung nicht im gültigen Wertebereich liegt */
    if( data.score < 1 || data.score > 4 ) {
      throw( 'Score must be from 1 to 4' );
    }
  } catch( message ) {
    /* Riak erwartet bei einem Fehler die folgenden JSON-Daten */
    return { "fail" : message };
  }
  /* Keine Probleme, also weitermachen */
  return object;
}
```

Speichern Sie diese Datei in dem von Ihnen festgelegten `js_source_dir`-Verzeichnis. Da wir Änderungen am Server-Kern vornehmen, müssen wir alle Entwicklungs-Server über das `restart`-Argument neu starten.

```
$ dev/dev1/bin/riak restart
$ dev/dev2/bin/riak restart
$ dev/dev3/bin/riak restart
```

Riak sucht nach allen Dateien mit der Endung `.js` und lädt sie in den Speicher. Sie können nun die `precommit`-Eigenschaft eines Buckets nutzen, um den Namen der JavaScript-*Funktion* (nicht den Dateinamen) festzulegen.

```
curl -X PUT http://localhost:8091/riak/animals \
-H "content-type:application/json" \
-d '{"props":{"precommit":[{"name" : "good_score"}]}}'
```

Wir wollen unseren neuen Hook nun ausprobieren, indem wir eine Bewertung größer 4 setzen. Unser Pre-Commit-Hook erzwingt eine Bewertung zwischen 1 und 4, weshalb der folgende Versuch mit einem Fehler endet:

```
curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "Content-Type: application/json" -d '{"score" : 5}'
HTTP/1.1 403 Forbidden
Content-Type: text/plain
Content-Length: 25

Score must be 1 to 4
```

Sie erhalten den Code 403 Forbidden zurück, sowie eine Fehlermeldung im Klartext, die wir im „fail"-Feld zurückgegeben haben. Wenn Sie mit `GET` den `bruiser`-Wert abrufen, bleibt die Bewertung bei 3. Wenn Sie versuchen, die Bewertung auf 2 zu setzen, werden Sie mehr Erfolg haben.

Post-Commit ähnelt dem Pre-Commit, wird aber erst ausgeführt, wenn der Commit erfolgreich war. Wir überspringen das an dieser Stelle, weil Post-Commit-Hooks nur in Erlang geschrieben werden können. Erlang-Entwickler führt die Online-Dokumentation durch die Installation eigener Module. Tatsächlich können Sie mit Erlang auch Mapreduce-Funktionen schreiben. Doch wir wollen uns jetzt anderen vorgefertigten Modulen und Erweiterungen zuwenden.

Riak erweitern

Riak wird mit verschiedenen Erweiterungen ausgeliefert, die standardmäßig deaktiviert sind, aber Features bieten, die Sie durchaus nützlich finden könnten.

In Riak suchen

Die Riak-Suche analysiert die Daten innerhalb ihres Riak-Clusters und baut daraus einen invertierten Index auf. An den Begriff *invertierter Index* werden Sie sich aus dem PostgreSQL-Kapitel erinnern (GIN steht für Generalisierter Invertierter Index). Genau wie GIN sorgt der Riak-Index dafür, dass viele String-Varianten schnell und effizient gesucht werden können, berücksichtigt dabei aber seine verteilte Natur.

Um die Riak-Suche nutzen zu können, müssen Sie es in Ihren `app.config`-Dateien aktivieren und die Riak-Suchkonfiguration auf `enabled, true` setzen.

```
%% Riak Search Config
  {riak_search, [
    %% To enable Search functionality set this 'true'.
    {enabled, true}
  ]},
```

Wenn Sie mit Suchmaschinen-Lösungen wie Lucene vertraut sind, ist dieser Teil ein Kinderspiel. Wenn nicht, hat man den Bogen schnell raus.

Wir müssen der Suche mitteilen, wenn sich Werte in der Datenbank ändern. Dazu nutzen wir einen Pre-Commit-Hook. Sie können `riak_search_kv_hook`, die `precommit`-Funktion des Erlang-Moduls, mit dem folgenden Befehl in das `animals`-Bucket einbinden:

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"precommit":
[{"mod": "riak_search_kv_hook","fun":"precommit"}]}}'
```

Der Aufruf `curl http://localhost:8091/riak/animals` zeigt, dass der Hook in die `precommit`-Eigenschaft des `animal`-Buckets eingefügt wurde. Wenn Sie nun JSON- oder XML-kodierte Daten in das `animals`-Bucket einfügen, indexiert die Riak-Suche die Feldnamen und Werte. Lassen Sie uns ein paar Tiere hochladen.

```
$ curl -X PUT http://localhost:8091/riak/animals/dragon \
-H "Content-Type: application/json" \
-d '{"nickname" : "Dragon", "breed" : "Briard", "score" : 1 }'
$ curl -X PUT http://localhost:8091/riak/animals/ace \
-H "Content-Type: application/json" \
-d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd", "score" : 3 }'
$ curl -X PUT http://localhost:8091/riak/animals/rtt \
-H "Content-Type: application/json" \
-d '{"nickname" : "Rin Tin Tin", "breed" : "German Shepherd", "score" : 4 }'
```

Sie haben verschiedene Möglichkeiten, die Daten abzufragen, aber wir wollen Riaks HTTP-Solr-Interface nutzen (das die Apache Solr[13] Such-Schnittstelle implementiert). Um in `/animals` zu suchen, greifen wir auf `/solr` zu, gefolgt vom Bucket-Namen `/animals` und dem `/select`-Befehl. Die Parameter legen den Suchbegriff fest. Im folgenden Beispiel wählen wir jede Rasse (*breed*) aus, die das Wort *Shepherd* enthält.

```
$ curl http://localhost:8091/solr/animals/select?q=breed:Shepherd
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="indent">on</str>
      <str name="start">0</str>
      <str name="q">breed:Shepherd</str>
      <str name="q.op">or</str>
      <str name="df">value</str>
      <str name="wt">standard</str>
      <str name="version">1.1</str>
      <str name="rows">2</str>
    </lst>
  </lst>
  <result name="response" numFound="2" start="0" maxScore="0.500000">
    <doc>
      <str name="id">ace</str>
      <str name="breed">German Shepherd</str>
      <str name="nickname">The Wonder Dog</str>
      <str name="score">3</str>
    </doc>
    <doc>
      <str name="id">rtt</str>
      <str name="breed">German Shepherd</str>
      <str name="nickname">Rin Tin Tin</str>
      <str name="score">4</str>
    </doc>
  </result>
</response>
```

Wenn die Query lieber JSON zurückliefern soll, fügen Sie den Parameter `wt=json` hinzu. Sie können mehrere Parameter in der Query kombinieren, indem Sie sie durch Leerzeichen (bzw. %20 in URL-kodierter Form) trennen und den Parameter `q.op` auf `and` setzen. Suchen Sie z. B. nach der Rasse „shepard“ und einem Spitznamen, in dem das Wort „*rin*“ enthalten ist, führen Sie die folgende Abfrage aus:

```
$ curl http://localhost:8091/solr/animals/select\
?wt=json&q=nickname:rin%20breed:shepherd&q.op=and
```

13. http://lucene.apache.org/solr/

Die Riak-Suche kennt eine umfangreichere Query-Syntax, wie z. B. Wildcards (mit * zum Matching mehrerer und ? zum Matching eines Zeichens), allerdings nur am Ende eines Suchbegriffs. Die Query `nickname:Drag*` würde Dragon erkennen, `nickname:*ragon` hingegen nicht. Bereichssuchen sind ebenfalls eine nette Option:

```
nickname:[dog TO drag]
```

Komplexere Queries mit Booleschen Operatoren, Gruppierungen und Umkreissuchen stehen ebenfalls zur Verfügung. Darüber hinaus können Sie eigene Datenkodierungen festlegen, eigene Indizes erzeugen und diese sogar bei der Suche wählen. Weitere URL-Parameter finden Sie in der folgenden Tabelle:

Parameter	Beschreibung	Standardwert
q	Der Query-String	
q.op	Verknüpfung der Query-Begriffe über `and` oder `or`	`or`
sort	Feldname, nach dem sortiert werden soll	
start	Das erste Objekt, das in der Trefferliste zurückgegeben werden soll	0
rows	Max. Anzahl zurückzuliefernder Ergebnisse	20
wt	Ausgabe im `xml`- oder `json`-Format	`xml`
index	Legt den zu verwendenden Index fest	

Es gibt noch sehr viel mehr über die Riak-Sucherweiterung zu lernen, weit mehr, als wir hier behandeln können. Dennoch sollten Sie ein Gefühl für ihre Leistungsfähigkeit bekommen haben. Sie ist eine gute Wahl, wenn Sie Suchfunktionalität für eine große Web-Anwendung bereistellen wollen, lohnt aber auch einen zweiten Blick, wenn viele einfache Ad-hoc-Abfragen durchgeführt werden müssen.

Riak indexieren

Seit der Version 1.0 unterstützt Riak Sekundärindizes. Sie ähneln (mit einer kleinen Abweichung) den Indizes, die wir bei PostgreSQL gesehen haben. Anstelle der Indexierung einer/mehrerer Spalte(n) erlaubt Riak die Indexierung von Metadaten, die mit dem Header des Objekts verknüpft sind.

Erneut müssen Sie eine Änderung an der `app.config`-Datei vornehmen. Ändern Sie das Storage-Backend (wie nachfolgend zu sehen) von bitcask in eLevelDB und starten Sie die Server neu:

```
{riak_kv, [
  %% Storage_backend specifies the Erlang module defining the
  %% storage mechanism that will be used on this node.
  {storage_backend, riak_kv_eleveldb_backend},
```

eLevelDB ist eine Erlang-Implementierung des Google Key/Value-Stores namens LevelDB.[14] Diese neue Backend-Implementierung erlaubt Sekundärindizes in Riak.

Sobald unser System bereit ist, können wir jedes Objekt mit einer beliebigen Zahl von Header-Tags indexieren. Diese *index entries* definieren, wie ein Objekt indexiert werden soll. Die Feldnamen beginnen mit `x-riak-index-` und enden mit `_int` oder `_bin` für Integer- oder Binärwerte (alles außer Integer).

Bei Blue II, dem Maskottchen der Butler Bulldogs, wollen wir über den Namen der Universität, dessen Maskottchen dieser Hund (`butler`) ist, indexieren, aber auch über die Versionsnummer (Blue 2 ist das zweite Bulldog-Maskottchen).

```
$ curl -X PUT http://localhost:8098/riak/animals/blue
-H "x-riak-index-mascot_bin: butler"
-H "x-riak-index-version_int: 2"
-d '{"nickname" : "Blue II", "breed" : "English Bulldog"}'
```

Sie werden bemerkt haben, dass die Indizes nichts mit den Werten zu tun haben, die im Schlüssel gespeichert sind. Das ist tatsächlich ein sehr mächtiges Feature, da wir Daten orthogonal zu den von uns gespeicherten Werten indexieren können. Wenn Sie also ein Video als Wert speichern wollen, können Sie es dennoch indexieren.

Den Wert über den Index abzurufen, ist eine einfache Sache.

```
$ curl http://localhost:8098/riak/animals/index/mascot_bin/butler
```

Zwar sind die Sekundärindizes bei Riak ein großer Schritt in die richtige Richtung, dennoch liegt noch ein langer Weg vor uns. Wenn Sie beispielsweise Datumswerte indexieren wollen, müssen Sie einen String speichern, der entsprechend sortiert werden kann – etwa `"JJJJMMTT"`. Die Speicherung von Fließkommawerten verlangt zuerst die Multiplikation mit einem signifikanten Vielfachen von 10 und die Speicherung als Integerwert – etwa 1.45 * 100

14. http://code.google.com/p/leveldb/

== 145. Ihr Client ist für die Konvertierung verantwortlich. Doch zwischen Mapreduce, Riak-Suche und (jetzt auch) Sekundärindizes bietet Riak viele Tools, die die klassischen Beschränkungen von Schlüssel/Wert-Speichern durch andere Zugriffsmöglichkeiten (die über einfache Schlüssel hinausgehen) aufweichen.

Was wir am dritten Tag gelernt haben

Wir haben Riak mit einigen seiner fortschrittlicheren Konzepte abgeschlossen: wie man Versionskonflikte mit Vektoruhren auflöst und wie man eingehende Daten mit Commit-Hooks absichert oder modifiziert. Wir haben uns auch einige Riak-Erweiterungen angesehen: die Aktivierung der Riak-Suche und der Datenindexierung, die uns flexiblere Queries erlauben.

Durch die Nutzung dieser Konzepte, zusammen mit Mapreduce (Tag 2) und `Links` (Tag 1), können Sie eine flexible Kombination von Tools aufbauen, die weit über den üblichen Schlüssel/Wert-Speicher hinausgeht.

Tag 3: Selbststudium

Finden Sie heraus

1. Finden das Contrib-List-Repository der Riak-Funktionen (Tipp: Sie finden es auf GitHub).
2. Lesen Sie mehr über Vektoruhren.
3. Lernen Sie, Ihre eigene Index-Konfiguration zu erzeugen.

Machen Sie Folgendes

1. Entwickeln Sie Ihren eigenen Index, der das `animals`-Schema definiert. Legen Sie dabei das `score`-Feld gezielt als Integer-Typ an und fragen Sie es als Bereichssuche ab.
2. Bauen Sie ein kleines Cluster von drei Servern (z. B. drei Laptops oder EC2-Instanzen)[15] auf und installieren Sie Riak auf jedem Server. Richten Sie die Server als Cluster ein. Installieren Sie die Google Stock-Daten, die auf der Basho-Website zu finden sind.[16]

15. http://aws.amazon.com/ec2/
16. http://wiki.basho.com/Loading-Data-and-Running-MapReduce-Queries.html

3.5 Zusammenfassung

Riak ist die erste von uns behandelte NoSQL-Datenbank. Es handelt sich um einen verteilten, datenreplizierenden, erweiterten Schlüssel/Wert-Speicher ohne Single Point of Failure.

Wenn Ihre Erfahrungen mit Datenbanken bisher nur relational waren, wird Ihnen Riak sehr fremdartig erscheinen. Es gibt keine Transaktionen, kein SQL und kein Schema. Es gibt Schlüssel, doch die Verknüpfung zwischen Buckets hat nichts mit einem Tabellen-Join gemein und Mapreduce kann eine schwer zu begreifende, einschüchternde Methodik sein.

Die Nachteile kann man für eine bestimmte Klasse von Problemen aber verschmerzen. Riaks Fähigkeit, über mehrere Server zu skalieren (statt nur auf einem einzelnen Server), und seine einfache Verwendbarkeit machen es zu einem ausgezeichneten Kandidaten zur Lösung der Skalierbarkeitsprobleme des Web. Und statt das Rad neu zu erfinden, greift Riak auf die HTTP-Struktur zurück, was jedem Framework und Web-fähigen System ein Maximum an Flexibilität erlaubt.

Riaks Stärken

Wenn Sie ein groß angelegtes Bestellsystem à la Amazon entwerfen oder in jeder Situation, in der Hochverfügbarkeit der wesentliche Aspekt ist, sollten Sie Riak in Erwägung ziehen. Eine von Riaks Stärken liegt zweifellos in seinem Bestreben, Single Points of Failure zu entfernen, um so eine maximale Betriebszeit und bedarfsgerechtes Wachsen (oder Schrumpfen) zu ermöglichen. Wenn Ihre Daten nicht komplex sind, hält Riak die Dinge einfach, erlaubt aber dennoch recht komplexe Abfragen, sollten sie nötig sein. Momentan werden über ein Dutzend Sprachen unterstützt (die Sie auf der Riak-Website finden), es ist aber bis in den Kern hinein erweiterbar, wenn Sie gerne in Erlang programmieren. Und wenn Sie eine höhere Geschwindigkeit brauchen, als sie HTTP bietet, können Sie sich auch an Protobuf versuchen,[17] einem effizient binärkodierten Transportprotokoll.

Riaks Schwächen

Wenn Sie einfach nur abfragbare, komplexe Datenstrukturen benötigen oder ein rigides Schema oder wenn Sie nicht horizontal mit Ihren Servern skalieren müssen, ist Riak wahrscheinlich nicht die beste Wahl. Eines der größten Mankos von Riak ist das Fehlen eines einfachen und robusten Frameworks für Ad-hoc-Abfragen (auch wenn man zweifellos auf dem richtigen Weg ist).

17. http://code.google.com/p/protobuf/

Mapreduce bietet phantastische und mächtige Funktionen, doch wir würden gerne mehr fest eingebaute URL- oder PUT-basierte Query-Funktionen sehen. Die Ergänzung um Indizes ist ein wichtiger Schritt in die richtige Richtung und ein Konzept, das wir liebend gern erweitert sehen würden. Und wenn Sie nicht in Erlang entwickeln wollen, werden Sie mit JavaScript einige Einschränkungen hinnehmen müssen, etwa das Fehlen von Post-Commit oder die langsamere Mapreduce-Ausführung. Allerdings arbeitet das Riak-Team an diesen relativ kleinen Problemchen.

Riak und CAP

Riak bietet eine clevere Möglichkeit, die allen verteilten Datenbanken durch CAP auferlegten Beschränkungen zu umgehen. Wie es das Problem angeht, ist erstaunlich, wenn man es mit einem System wie PostgreSQL vergleicht, das (im Wesentlichen) nur eine hohe Schreibkonsistenz unterstützt. Riak baut auf der Erkenntnis des Amazon Dynamo-Papiers auf, dass CAP auf Basis von Buckets oder Requests geändert werden kann. Das ist ein großer Schritt in Richtung robuster und flexibler Open-Source-Datenbanksysteme. Wenn Sie über die anderen Datenbanken in diesem Buch lesen, sollten Sie Riak immer im Hinterkopf haben und Sie werden von dessen Flexibilität beeindruckt sein.

Abschließende Gedanken

Wenn Sie einen riesigen Katalog mit Daten speichern müssen, gibt es schlechtere Lösungen als Riak. Relationale Datenbanken werden zwar seit über 40 Jahren erforscht und verbessert, aber nicht jedes Problem verlangt ACID-Konformität oder die Fähigkeit, ein Schema zu erzwingen. Wenn Sie eine Datenbank in ein Gerät einbetten oder Finanztransaktionen verarbeiten wollen, sollten Sie Riak meiden. Wenn Sie skalieren oder große Datenmengen im Web bereitstellen müssen, sollten Sie einen Blick riskieren.

KAPITEL 4

HBase

Apache HBase ist, wie eine Nagelpistole, für große Jobs gemacht. Sie würden HBase niemals nutzen, um die Vertriebsliste Ihres Unternehmens zu katalogisieren, ebensowenig wie Sie eine Nagelpistole nutzen würden, um ein Puppenhaus zu bauen. Wenn Ihre Daten nicht viele Gigabytes umfassen, brauchen Sie sehr wahrscheinlich ein kleineres Tool.

HBase erinnert auf den ersten Blick stark an eine relationale Datenbank, und zwar so sehr, dass man glauben könnte, es wäre eine, wenn man es nicht besser wüsste. Wenn Sie HBase erlernen wollen, ist die Technik nicht die größte Herausforderung, sondern dass viele der von HBase verwendeten Wörter einem so schmeichlerisch und trügerisch vertraut sind. Zum Beispiel speichert HBase Daten in *Tabellen*, die *Zellen* enthalten, die bei den Schnittpunkten von *Zeilen* und *Spalten* liegen. Soweit, so gut, richtig?

Falsch! Bei HBase verhalten sich Tabellen nicht wie Relationen, Zeilen dienen nicht als Datensätze und Spalten sind vollständig variabel (werden also nicht durch die Schema-Beschreibung vorgegeben). Das Schema-Design ist für die Performance-Charakteristika des System immer noch wichtig, hält Ihre Daten aber nicht in Ordnung. HBase ist der böse Zwilling (wenn Sie so wollen, der Bizarro) der RDBMS.

Warum sollte man diese Datenbank also verwenden? Neben der Skalierbarkeit gibt es verschiedene Gründe. Erstens besitzt HBase einige Features, die anderen Datenbanken fehlen, etwa Versionierung, Komprimierung, Garbage Collection (für abgelaufene Daten) und speicherbasierte (In-Memory) Tabellen. Diese Features von Haus aus zur Verfügung zu haben, bedeutet, dass man weniger Code schreiben muss, wenn die Anforderungen sie verlangen. HBase macht auch hohe Garantien in Bezug auf die Konsistenz, was den Wechsel von relationalen Datenbanken vereinfacht.

Aus all diesen Gründen glänzt HBase als Grundpfeiler eines OLAP-Systems (Online Analytical Processing) ist. Zwar können einzelne Operationen langsamer sein als vergleichbare Operationen anderer Datenbanken, doch die Verarbeitung riesiger Datenmengen ist etwas, was HBase auszeichnet. Bei wirklich großen Queries lässt HBase andere Datenbanken oft hinter sich. Das erklärt auch, warum HBase häufig bei großen Unternehmen das Rückgrat von Logging- und Suchsystemen bildet.

4.1 Einführung in HBase

HBase ist eine *spaltenorientierte* Datenbank, die auf ihre Konsistenz und Skalierbarkeit stolz ist. Sie basiert auf BigTable, einer hochperformanten, proprietären Datenbank, die von Google entwickelt und 2006 im White Paper „Bigtable: A Distributed Storage System for Structured Data“ beschrieben wurde.[1] Ursprünglich zur Verarbeitung natürlicher Sprache entwickelt, begann HBase sein Dasein als Contrib-Paket für Apache Hadoop. Seither wurde es zu einem Apache-Spitzenprojekt.

Unter Architektur-Aspekten wurde HBase fehlertolerant entworfen. Hardware-Fehler sind bei einzelnen Maschinen eher selten, doch in einem großen Cluster ist der Ausfall eines Knotens die Norm. Durch das sog. Write-Ahead-Logging und eine verteilte Konfiguration kann sich HBase von einzelnen Server-Ausfällen schnell erholen.

Darüber hinaus lebt HBase in einem Ökosystem, das seine eigenen zusätzlichen Vorteile bietet. HBase basiert auf Hadoop – einer stabilen skalierbaren Plattform, die ein verteiltes Dateisystem und Mapreduce-Fähigkeiten bietet. Wo immer Ihnen HBase begegnet, finden Sie auch Hadoop und andere Infrastrukturkomponenten, die Sie in eigenen Anwendungen nutzen können.

Es wird aktiv von einer Reihe großer Unternehmen für ihre „Big Data“-Probleme eingesetzt und weiterentwickelt. Unter anderem hat Facebook im November 2010 HBase als Kernkomponente seiner neuen Messaging-Infrastruktur angekündigt. Stumbleupon verwendet HBase seit Jahren zur Echtzeit-Datenspeicherung und Analyse. Verschiedene Features der Site werden direkt von HBase bedient. Twitter nutzt HBase ausgiebig. Das reicht von der Datengenerierung (für Applikationen wie die Personen-Suche) bis zu Speicherung von Monitoring/Performance-Daten. Die Liste der HBase nutzenden Unternehmen umfasst auch Größen wie eBay, Meetup, Ning, Yahoo! und viele andere.

1. http://research.google.com/archive/bigtable.html

Bei all dieser Aktivität kommen neue HBase-Versionen in recht schneller Folge heraus. Während diese Zeilen geschrieben werden, ist 0.90.3 die aktuelle stabile Version, die wir hier auch verwenden. Also laden Sie HBase herunter und los geht's.

4.2 Tag 1: CRUD und Tabellenadministration

Das Ziel des heutigen Tages besteht darin, die Grundlagen des Umgangs mit HBase zu lernen. Wir werden eine lokale Instanz von HBase im Standalone-Modus betreiben und dann die HBase-Shell nutzen, um Tabellen anzulegen und zu verändern, sowie Daten mit elementaren Befehlen einfügen und modifizieren. Danach wollen wir untersuchen, wie man einige dieser Operationen programmtechnisch vornimmt, indem wir die HBase Java API in JRuby nutzen. Nebenbei enthüllen wir einige HBase-Architekturkonzepte wie die Beziehung zwischen Zeilen, Spaltenfamilien und den Werten in einer Tabelle.

Ein voll funktionsfähiges HBase-Cluster für den Produktiveinsatz (so die landläufige Meinung) sollte aus nicht weniger als fünf Knoten bestehen. Für unsere Bedürfnisse wäre ein solches Setup allerdings etwas zu viel des Guten. Glücklicherweise unterstützt HBase drei Betriebsmodi:

- Im Standalone-Modus arbeitet eine einzelne Maschine allein.
- Im pseudoverteilen (pseudodistributed) Modus gibt ein einzelner Knoten vor, ein Cluster zu sein.
- Im vollständig verteilten (fully distributed) Modus arbeitet ein Cluster von Knoten zusammen.

Im Großteil dieses Kapitels betreiben wir HBase im Standalone-Modus. Doch selbst das kann eine Herausforderung sein, und obwohl wir nicht jeden Aspekt der Installation und Administration behandeln, geben wir an geeigneten Stellen Tipps zur Fehlersuche.

HBase konfigurieren

Bevor man HBase nutzen kann, muss es konfiguriert werden. Die Konfigurationseinstellungen für HBase finden Sie in einer Datei namens `hbase-site.xml`, die Sie im Verzeichnis `${HBASE_HOME}/conf/` finden. Beachten Sie, dass `HBASE_HOME` eine Umgebungsvariable ist, die auf das Verzeichnis verweist, in dem HBase installiert wurde.

Anfangs enthält diese Datei nur einen leeren `<configuration>`-Tag. Sie können in der Konfiguration eine beliebige Anzahl von Eigenschaften (Properties) im folgenden Format definieren:

```
<property>
  <name>name.der.eigenschaft</name>
  <value>Wert der Eigenschaft</value>
</property>
```

Eine vollständige Liste aller verfügbaren Properties, zusammen mit Standardwerten und Beschreibungen, finden Sie in `hbase-default.xml` unter `${HBASE_HOME}/src/main/resources`.

Standardmäßig verwendet HBase ein temporäres Verzeichnis zur Speicherung seiner Daten-Dateien. Das bedeutet, dass Sie *alle Daten* verlieren, wenn das System entscheidet, sich Plattenplatz zurückzuholen.

Um Ihre Daten dauerhaft zu speichern, müssen Sie sich eine andere Stelle aussuchen. Setzen Sie die `hbase.rootdir`-Property auf den entsprechenden Pfad:

```
<property>
  <name>hbase.rootdir</name>
  <value>file:///pfad/auf/hbase</value>
</property>
```

Um HBase zu starten, öffnen Sie ein Terminal und führen den folgenden Befehl aus:

```
${HBASE_HOME}/bin/start-hbase.sh
```

Um HBase herunterzufahren, verwenden Sie den Befehl `stop-hbase.sh` im gleichen Verzeichnis.

Wenn etwas schiefgeht, sehen Sie sich die zuletzt modifizierten Dateien im Verzeichnis `${HBASE_HOME}/logs` an. Bei *nix-basierten Systemen gibt der folgende Befehl die zuletzt geschriebenen Logdaten an der Console aus:

```
cd ${HBASE_HOME}
find ./logs -name "hbase-*.log" -exec tail -f {} \;
```

Die HBase-Shell

Die HBase-Shell ist ein JRuby-basiertes Kommandozeilen-Programm, über das Sie mit HBase kommunizieren können. In der Shell können Sie Tabellen hinzufügen und löschen, Tabellen-Schemata ändern, Daten einfügen oder löschen und viele Dinge mehr. Später werden wir andere Möglichkeiten kennenlernen, mit HBase zu kommunizieren, aber für den Augenblick wird die Shell unsere Heimat sein.

Bei laufendem HBase öffnen Sie ein Terminal und starten die HBase-Shell:

```
${HBASE_HOME}/bin/hbase shell
```

Um uns zu vergewissern, dass alles richtig funktioniert, fragen wir die Versionsinformationen ab.

```
hbase> version
0.90.3, r1100350, Sat May 7 13:31:12 PDT 2011
```

Sie können jederzeit `help` eingeben, um eine Liste verfügbarer Befehle oder Nutzungshinweise zu einem bestimmten Befehl abzurufen.

Als Nächstes führen wir den `status`-Befehl aus, um zu sehen, wie sich Ihr HBase-Server hält.

```
hbase> status
1 servers, 0 dead, 2.0000 average load
```

Wenn bei einem dieser Befehle ein Fehler auftritt oder wenn sich die Shell aufhängt, könnte es ein Problem mit der Verbindung geben. HBase tut sein Möglichstes, um seine Dienste entsprechend Ihren Netzeinstellungen automatisch zu konfigurieren, doch manchmal geht das schief. Wenn sich bei Ihnen diese Symptome zeigen, überprüfen Sie die *HBase-Netzwerkeinstellungen*, auf Seite 108.

Eine Tabelle anlegen

Eine Map ist ein Schlüssel/Wert-Paar, ähnlich einem Hash bei Ruby oder einer Hashmap bei Java. Eine Tabelle in HBase ist im Grunde eine große Map. Nun, genauer gesagt, eine Map von Maps.

In einer HBase-Tabelle sind Schlüssel beliebige Strings, die auf eine *Zeile* mit Daten zeigen. Eine Zeile ist selbst eine Map, in der als *Spalten* bezeichnete Schlüssel und Werte nicht weiter interpretierte Arrays von Bytes bilden. Spalten werden zu *Spaltenfamilien (column families)* gruppiert, so dass der vollständige Name einer Spalte aus zwei Teilen besteht: dem Namen der Spaltenfamilie und dem *Spaltenbezeichner (column qualifier)*. Sie werden häufig über einen Doppelpunkt zusammengefasst (zum Beispiel `'familie:qualifier'`).

Abbildung 13, *HBase-Tabellen bestehen aus Zeilen, Schlüsseln, Spaltenfamilien, Spalten und Werten.*, auf Seite 109 veranschaulicht diese Konzepte.

In dieser Abbildung sehen Sie eine hypothetische Tabelle mit zwei Spaltenfamilien: `color` (Farbe) und `shape` (Form). Die Tabelle enthält zwei Zeilen – gekennzeichnet durch die gestrichelten Linien: `first` und `second`. Wenn wir

HBase-Netzwerkeinstellungen

Standardmäßig versucht HBase, seine Dienste externen Clients zugänglich zu machen, doch in unserem Fall müssen wir die Verbindung nur vom gleichen Rechner aus herstellen. Es könnte daher hilfreich sein, einige (oder alle) der folgenden Properties in Ihrer `hbase-site.xml` festzulegen. Beachten Sie, dass die Werte in der folgenden Tabelle nur bei lokalen (nicht entfernten) Verbindungen helfen:

Property	Wert
hbase.master.dns.interface	lo
hbase.master.info.bindAddress	127.0.0.1
hbase.regionserver.info.bindAddress	127.0.0.1
hbase.regionserver.dns.interface	lo
hbase.zookeeper.dns.interface	lo

Die Properties teilen HBase mit, wie Verbindungen mit dem Master- und den Regions-Servern (die wir später noch behandeln) sowie dem Zookeeper-Konfigurationsdienst herzustellen sind. Die Properties mit dem Wert „lo“ verweisen auf das sog. Loopback-Interface. Bei *nix-Systemen ist das Loopback-Interface kein echtes Netzwerk-Interface (wie Ethernet- oder WLAN-Karten), sondern ein reines Software-Interface, über das der Computer die Verbindung mit sich selbst herstellen kann. Die `bindAddress`-Properties teilen HBase mit, an welchen IP-Adressen es horchen soll.

uns nur die erste Zeile (`first`) ansehen, erkennen wir, dass sie drei Spalten in der Spaltenfamilie `color` (mit den Bezeichnern `red`, `blue` und `yellow`) besitzt sowie eine Spalte in der Spaltenfamilie `shape`(`square`). Die Kombination aus Zeilenschlüssel und Spaltenname (Familie und Bezeichner) ergibt eine Adresse zum Lokalisieren von Daten. In diesem Beispiel führt uns das Tupel `first/color:red` zum Wert `'#F00'`.

Nun wollen wir das, was wir über die Tabellenstruktur gelernt haben, für etwas Unterhaltsames nutzen – wir bauen uns ein Wiki!

Es gibt sehr viele Informationen, die wir mit einem Wiki verknüpfen wollen, aber wir beginnen beim absoluten Minimum. Ein Wiki enthält Seiten, die alle einen eindeutigen Titel-String haben und irgendeinen Artikeltext enthalten.

Wir verwenden den `create`-Befehl, um unsere Wiki-Tabelle anzulegen:

```
hbase> create 'wiki', 'text'
0 row(s) in 1.2160 seconds
```

Wir haben eine Tabelle namens `wiki` mit einer einzigen Spaltenfamilie namens `text` erzeugt. Die Tabelle ist momentan leer; sie enthält keine Zeilen

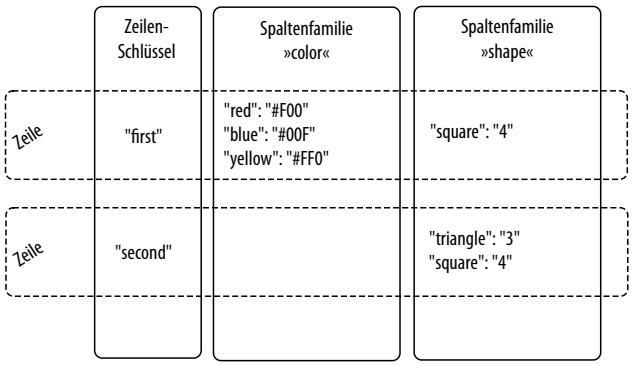


Abbildung 13: HBase-Tabellen bestehen aus Zeilen, Schlüsseln, Spaltenfamilien, Spalten und Werten.

und daher auch keine Spalten. Im Gegensatz zu einer relationalen Datenbank ist eine Spalte bei HBase nur für die Zeile spezifisch, in der sie enthalten ist. Wenn wir Zeilen einfügen, fügen wir gleichzeitig auch Spalten ein, um Daten zu speichern.

Wenn wir uns die Tabellen-Architektur vor Augen führen, haben wir so etwas wie in Abbildung 14, *Die wiki-Tabelle hat eine Spaltenfamilie.*, auf Seite 110. Nach unserer eigenen Konvention erwarten wir, dass jede Zeile genau eine Spalte innerhalb der `text`-Familie besitzt, die über den Leerstring (`''`) qualifiziert wird. Der vollständige Name der Spalte, der den Text einer Seite enthält, lautet also `'text:'`.

Damit unsere Wiki-Tabelle nützlich ist, brauchen wir natürlich Inhalte. Lassen Sie uns also etwas einfügen!

Daten einfügen, aktualisieren und abrufen

Unser Wiki braucht eine Homepage, also fangen wir damit an. Um Daten in eine HBase-Tabelle einzufügen, verwenden wir den `put`-Befehl:

```
hbase> put 'wiki', 'Home', 'text:', 'Welcome to the wiki!'
```

Dieser Befehl fügt in die Wiki-Tabelle eine neue Zeile mit dem Schlüssel `'Home'` ein und `'Welcome to the wiki!'` in die Spalte `'text:'`.

Wir können die Daten für die `'Home'`-Zeile mit `get` abrufen, wozu wir zwei Parameter benötigen: den Namen der Tabelle und den Schlüssel der Zeile.

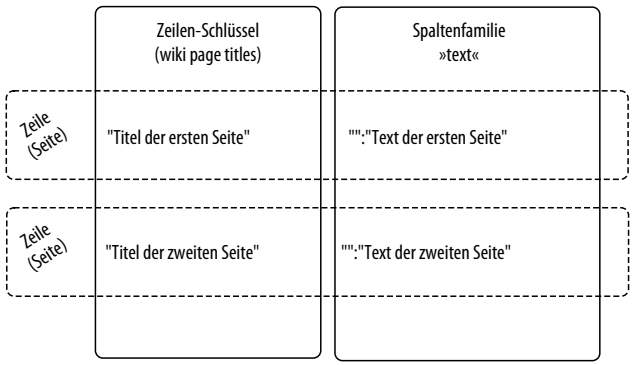


Abbildung 14: Die wiki-Tabelle hat eine Spaltenfamilie.

Optional kann auch eine Liste der zurückzuliefernden Spalten angegeben werden.

```
hbase> get 'wiki', 'Home', 'text:'
COLUMN CELL
 text: timestamp=1295774833226, value=Welcome to the wiki!
1 row(s) in 0.0590 seconds
```

Beachten Sie das `timestamp`-Feld in der Ausgabe. HBase speichert zu allen Werten einen Zeitstempel mit ab, der die Zeit in Millisekunden seit Beginn der Epoche (00:00:00 UTC am 1. Januar 1970) angibt. Wird ein neuer Wert in die gleiche Zelle geschrieben, bleibt der alte Wert (indexiert über den Zeitstempel) erhalten. Das ist ein beeindruckendes Feature. Bei den meisten Datenbanken müssen Sie sich selbst um die alten Daten kümmern, doch bei HBase ist die Versionierung mit integriert!

Put und Get

Die Befehle `put` und `get` erlauben die explizite Angabe des Zeitstempels. Wenn Millisekunden seit Epochenbeginn nicht Ihrem Geschmack entsprechen, können Sie einen anderen Integerwert Ihrer Wahl angeben. Das eröffnet Ihnen eine zusätzliche Dimension, mit der Sie arbeiten können, wenn Sie sie brauchen. Wenn Sie keinen Zeitstempel angeben, verwendet HBase beim Einfügen die aktuelle Zeit und gibt beim Lesen die aktuellste Version zurück.

Ein Beispiel dafür, wie ein Unternehmen das Zeitstempel-Feld überlädt, finden Sie in *Fallstudie: Facebooks Messaging-Index-Tabelle*, auf Seite 111. Im weiteren Verlauf des Kapitels werden wir die Standard-Interpretation des Zeitstempels verwenden.

Fallstudie: Facebooks Messaging-Index-Tabelle

Facebook nutzt HBase als grundlegende Komponente seiner Messaging-Infrastruktur, und zwar sowohl zur Speicherung von Nachrichten-Daten, als auch zur Pflege eines invertierten Indexes für die Suche.

Bei seinem Index-Tabellen-Schema:

- sind die Zeilenschlüssel Benutzer-IDs;
- sind Spalten-Bezeichner Wörter, die in den Nachrichten der Benutzer vorkommen;
- sind Zeitstempel Nachrichten-IDs, die dieses Wort enthalten.

Nachrichten zwischen Nutzern sind unveränderlich, weshalb auch die Indexeinträge für eine Nachricht statisch sind. Das Konzept versionierter Werte ist hier nicht sinnvoll.

Für Facebook bietet die Manipulation des Zeitstempels zum Matching von Nachrichten-IDs eine zusätzliche Dimension zur Speicherung von Daten.

Tabellen ändern

Bislang besteht unser Wiki-Schema aus Seiten mit Titeln, Text und einer integrierten Versions-Historie, mehr aber auch nicht. Wir wollen das wie folgt erweitern:

- In unserem Wiki wird eine Seite eindeutig über ihren Titel identifiziert.
- Eine Seite kann beliebig oft überarbeitet werden.
- Eine Überarbeitung wird über ihren Zeitstempel identifiziert.
- Eine Überarbeitung enthält Text und optional einen Commit-Kommentar.
- Eine Überarbeitung erfolgt durch einen Autor, der über seinen Namen identifiziert wird.

Visuell lassen sich unsere Anforderungen wie in Abbildung 15, *Anforderungen an eine Wiki-Seite (mit Zeitdimension)*, auf Seite 112 darstellen. Bei dieser abstrakten Darstellung unserer Anforderungen einer Seite erkennen wir, dass jede Überarbeitung einen Autor, einen Commit-Kommentar, den Artikeltext und einen Zeitstempel enthält. Der Titel der Seite ist nicht Teil der Überarbeitung, da wir ihn als ID für die Überarbeitungen einer Seite nutzen.

Die Abbildung unserer Vorstellungen in eine HBase-Tabelle verlangt eine etwas andere Form, die in Abbildung 16, *Aktualisierte Wiki-Tabellenarchitektur (ohne Zeitdimension)*, auf Seite 112 zu sehen ist. Unsere Wiki-Tabelle verwendet den Titel als Zeilenschlüssel und gruppiert weitere Seiten-Daten in zwei Spaltenfamilien namens `text` und `revision`. Die Spaltenfamilie `text` ist

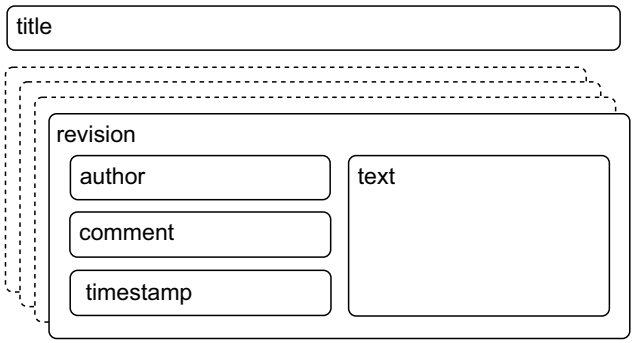


Abbildung 15: Anforderungen an eine Wiki-Seite (mit Zeitdimension)

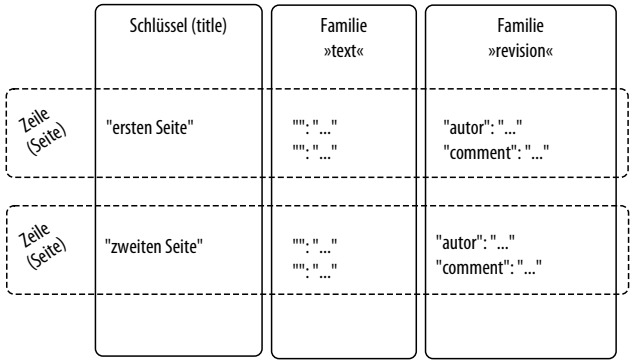


Abbildung 16: Aktualisierte Wiki-Tabellenarchitektur (ohne Zeitdimension)

unverändert. Wir erwarten in jeder Zeile genau eine Spalte, die über den Leerstring (`''`) qualifiziert wird, für den Artikel-Inhalt. Die Spaltenfamilie `revision` enthält andere revisionsspezifische Daten wie den Autor und den Commit-Kommentar.

Standardwerte

Wir haben die Wiki-Tabelle ohne besondere Optionen angelegt, so dass alle HBase-Standardwerte verwendet wurden. Einer dieser Standardwerte legt fest, dass nur drei Versionen (`VERSIONS`) der Spaltenwerte vorgehalten werden. Das wollen wir ändern. Um Änderungen am Schema vorzunehmen, müssen wir die Tabelle zuerst mit dem `disable`-Befehl offline nehmen.

```
hbase> disable 'wiki'
0 row(s) in 1.0930 seconds
```

Nun können wir die Charakteristika der Spaltenfamilie mit dem `alter`-Befehl ändern.

```
hbase> alter 'wiki', { NAME => 'text', VERSIONS =>
hbase*  org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0430 seconds
```

Hier weisen wir HBase an, für die Spaltenfamilie `text` das `VERSIONS`-Attribut zu ändern. Es gibt eine Reihe anderer Attribute, die wir setzen können, von denen wir einige am zweiten Tag behandeln werden. Die `hbase*`-Zeile ist eine Fortsetzung der vorherigen Zeile.

Eine Tabelle ändern

Operationen, die die Charakteristika der Spaltenfamilie ändern, können sehr teuer werden, weil HBase eine neue Spaltenfamilie mit der gewählten Spezifikation erzeugen und dann alle Daten kopieren muss. Bei einem Produktivsystem kann das zu signifikanten Ausfallzeiten führen. Aus diesem Grund ist es eine gute Idee, die Optionen einer Spaltenfamilie im Vorfeld festzulegen.

Während die Wiki-Tabelle noch deaktiviert ist, wollen wir die Spaltenfamilie `revision` einfügen. Dazu verwenden wir ebenfalls den `alter`-Befehl:

```
hbase> alter 'wiki', { NAME => 'revision', VERSIONS =>
hbase*  org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0660 seconds
```

Genau wie bei der `text`-Familie haben wir nur eine `revision`-*Spaltenfamilie* in das Tabellenschema eingefügt, keine individuellen *Spalten*. Auch wenn wir davon ausgehen, dass jede Zeile letztlich `revision:author` und `revision:comment` enthält, liegt es am Client, diese Erwartung zu erfüllen. In einem formalen Schema festgelegt ist das nicht. Wenn jemand `revision:foo` für eine Seite speichern möchte, wird HBase ihn nicht aufhalten.

Weiter geht's

Nachdem wir unser Schema entsprechend erweitert haben, aktivieren wir unser Wiki wieder:

```
hbase> enable 'wiki'
0 row(s) in 0.0550 seconds
```

Da die Wiki-Tabelle nun unsere gewachsenen Anforderungen erfüllt, können wir damit beginnen, Daten in die Spalten der `revision`-Spaltenfamilie einzufügen.

Daten programmatisch hinzufügen

Wie wir gesehen haben, eignet sich die HBase-Shell gut für solche Aufgaben wie die Bearbeitung von Tabellen. Leider wird das Hinzufügen von Daten in der Shell nicht besonders gut unterstützt. Mit dem `put`-Befehl können Sie nur jeweils einen Spaltenwert setzen, bei unserem aktualisierten Schema müssen wir aber mehrere Spaltenwerte gleichzeitig einfügen, damit alle den gleichen Zeitstempel haben. Wir werden also mit dem Skripting beginnen müssen.

Das folgende Skript kann direkt in der HBase-Shell ausgeführt werden, weil die Shell gleichzeitig ein JRuby-Interpreter ist. Wenn Sie es ausführen, wird eine neue Version des Textes für die Homepage gespeichert und gleichzeitig werden die Autoren- und Kommentarfelder gesetzt. JRuby läuft auf der Java Virtual Machine (JVM), was uns Zugriff auf den HBase-Java-Code gibt. Diese Beispiele funktionieren nur mit JVM-Ruby.

hbase/put_multiple_columns.rb

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

table = HTable.new( @hbase.configuration, "wiki" )

p = Put.new( *jbytes( "Home" ) )

p.add( *jbytes( "text", "", "Hello world" ) )
p.add( *jbytes( "revision", "author", "jimbo" ) )
p.add( *jbytes( "revision", "comment", "my first edit" ) )

table.put( p )
```

Die `import`-Zeilen versorgen die Shell mit Referenzen auf nützliche HBase-Klassen. Auf diese Weise müssen wir später nicht den vollständigen Namensraum angeben. Als Nächstes verarbeitet die `jbytes`-Funktion eine beliebige Anzahl von Argumenten und gibt ein in Java-Byte-Arrays umgewandeltes Array zurück (so wie es die HBase-API-Methoden verlangen).

Danach erzeugen wir eine lokale Variable (`table`), die auf unsere Wiki-Tabelle zeigt, wobei wir das `@hbase`-Administrationsobjekt für Konfigurationsinformationen nutzen.

Als Nächstes bereiten wir eine Commit-Operation vor, indem wir eine neue Instanz von `Put` erzeugen und vorbereiten. In diesem Beispiel halten wir uns an die Homepage, mit der wir bisher gearbeitet haben. Abschließend fügen wir mit `add` Properties in unsere `Put`-Instanz ein und rufen dann das `table`-Objekt auf, um die `put`-Operation auszuführen. Die `add`-Methode gibt es in unterschiedlichen Formen. In unserem Fall verwenden wir die Version mit drei Argumenten: `add(spaltenfamilie, spaltenbezeichner, wert)`.

Warum Spaltenfamilien?

Sie könnten versucht sein, die gesamte Struktur ohne Spaltenfamilien aufzubauen. Warum sollte man nicht alle Daten in einer einzelnen Spaltenfamilie speichern? Eine solche Lösung wäre einfacher zu implementieren. Doch es gibt Nachteile, wenn man auf Spaltenfamilien verzichtet, namentlich fehlende Möglichkeiten zum Performance-Tuning. Die Performance-Optionen jeder Spaltenfamilie werden unabhängig voneinander konfiguriert. Diese Einstellungen legen Dinge fest wie die Lese-/Schreibgeschwindigkeit und den Verbrauch von Plattenplatz.

Alle Operation von HBase sind auf *Zeilenebene* atomisch. Unabhängig von der Anzahl betroffener Spalten hat die Operation einen konsistenten Blick auf die Zeile, auf die der Zugriff/die Modifikation erfolgt. Diese Design-Entscheidung hilft Clients dabei, intelligente Schlussfolgerungen zu den Daten treffen zu können.

Unsere `put`-Operation wirkt sich auf mehrere Spalten aus und gibt keinen Zeitstempel an, so dass alle Spaltenwerte den gleichen Zeitstempel aufweisen (die aktuelle Zeit in Millisekunden). Das wollen wir mit `get` verifizieren.

```
hbase> get 'wiki', 'Home'
COLUMN             CELL
 revision:author   timestamp=1296462042029, value=jimbo
 revision:comment  timestamp=1296462042029, value=my first edit
 text:             timestamp=1296462042029, value=Hello world
3 row(s) in 0.0300 seconds
```

Wie Sie sehen können, weist jeder aufgeführte Spaltenwert den gleichen `timestamp` auf.

Was wir am ersten Tag gelernt haben

Heute haben wir einen ersten Blick auf einen laufenden HBase-Server geworfen. Wir haben gelernt, wie man ihn konfiguriert und Logdateien für die Fehlersuche kontrolliert. Mit Hilfe der HBase-Shell haben wir grundlegende Aufgaben der Administration und Datenverarbeitung durchgeführt.

Mit der Modellierung eines Wiki-Systems haben wir das Schema-Design bei HBase erkundet. Wir haben gelernt, wie man Tabellen erzeugt und Spaltenfamilien manipuliert. Das Design eines HBase-Schemas bedeutet, Entscheidungen zu Spaltenfamilien-Optionen zu treffen, und – genauso wichtig – die semantische Interpretation von Features wie Zeitstempel und Zeilen-Schlüssel.

Wir haben auch ein wenig in der HBase Java-API gestöbert, indem wir JRuby-Code in der Shell ausgeführt haben. Am zweiten Tag werden wir noch einen Schritt weiter gehen und die Shell nutzen, um eigene Skripten für große Jobs wie den Datenimport auszuführen.

Idealerweise haben Sie bereits einige relationale Konzepte hinter sich gelassen, die auf Begriffen wie *Tabelle*, *Zeile* und *Spalte* lasten. Der Unterschied in der Verwendung dieser Begriffe bei HBase und anderen Systemen wird noch stärker, je tiefer wir in die Features von HBase eintauchen.

Tag 1: Selbststudium

Die online verfügbare HBase-Dokumentation gibt es in zwei Varianten: extrem technisch oder gar nicht. Das ändert sich langsam, da mittlerweile „Erste Schritte"-Leitfäden auftauchen, doch Sie sollten sich darauf einstellen, einige Zeit mit Javadoc oder dem Quellcode zu verbringen, um Antworten zu finden.

Finden Sie heraus

1. Finden Sie heraus, wie man mit der Shell Folgendes macht:
   - Einzelne Spaltenwerte einer Zeile löschen
   - Eine ganze Zeile löschen
2. Setzen Sie ein Lesezeichen für die HBase API-Dokumentation der von Ihnen verwendeten HBase-Version.

Machen Sie Folgendes

1. Entwickeln Sie eine Funktion namens `put_many`, die eine `Put`-Instanz erzeugt, eine beliebige Anzahl von Spalten/Wert-Paaren hinzufügt und diese dann in die Tabelle übernimmt. Die Signatur sollte wie folgt aussehen:

```
def put_many( tabellen_name, zeile, spaltenwerte )
  # Ihr Code steht hier
end
```

2. Definieren Sie Ihre `put_many`-Funktion, indem Sie sie in die HBase-Shell einfügen und wie folgt aufrufen:

```
hbase> put_many 'wiki', 'Some title', {
hbase*  "text:" => "Some article text",
hbase*  "revision:author" => "jschmoe",
hbase*  "revision:comment" => "no comment" }
```

4.3 Tag 2: Mit großen Datenmengen arbeiten

Nachdem wir am ersten Tag gelernt haben, wie man Tabellen anlegt und manipuliert, ist es an der Zeit, ein paar ernsthafte Daten in unsere Wiki-Tabelle einzufügen. Heute wollen wir die HBase-APIs nutzen und Wikipedia-Inhalte direkt in unser Wiki einfügen! Ganz nebenbei werden wir einige Performance-Tricks kennenlernen, die für einen schnelleren Import sorgen. Abschließend wollen wir uns die HBase-Interna anschauen, um zu sehen, wie Daten in Regionen partitioniert werden, was sowohl der Performance als auch der Disaster-Recovery dient.

Daten importieren, Skripten ausführen

Ein typisches Problem, vor dem die Leute stehen, wenn sie ein neues Datenbanksystem ausprobieren wollen, ist die Migration der Daten. Von Hand durchgeführte `Put`-Operationen mit statischen Strings (wie am ersten Tag) sind schön und gut, aber wir können das besser.

Glücklicherweise ist das Einfügen von Befehlen in die Shell nicht die einzige Möglichkeit, sie auszuführen. Wenn Sie die HBase-Shell über die Kommandozeile starten, können Sie auch den Namen eines auszuführenden JRuby-Skripts angeben. HBase führt das Skript dann aus, als hätten Sie es direkt in die Shell eingetippt. Die Syntax sieht wie folgt aus:

```
${HBASE_HOME}/bin/hbase shell <Ihr_Skript> [<optionale_argumente> ...]
```

Da wir besonders an „Big Data“ interessiert sind, wollen wir ein Skript entwickeln, das Wikipedia-Artikel in unsere Wiki-Tabelle importiert. Die WikiMedia Foundation – die über Wikipedia, Wictionary und andere Projekte wacht –, veröffentlicht regelmäßig Daten-Dumps, die wir nutzen können. Diese Dumps liegen in Form riesiger XML-Dateien vor. Hier ein beispielhafter Datensatz aus der englischen Wikipedia:

```
<page>
  <title>Anarchism</title>
  <id>12</id>
  <revision>
    <id>408067712</id>
    <timestamp>2011-01-15T19:28:25Z</timestamp>
    <contributor>
      <username>RepublicanJacobite</username>
      <id>5223685</id>
    </contributor>
    <comment>Undid revision 408057615 by [[Special:Contributions...</comment>
    <text xml:space="preserve">{{Redirect|Anarchist|the fictional character|
...
[[bat-smg:Anarkėzmos]]
    </text>
  </revision>
</page>
```

Da wir so clever waren, sind hier alle Informationen enthalten, die wir in unserem Schema bereits berücksichtigt haben: Titel (Zeilenschlüssel), Text, Zeitstempel und Autor. Es sollte also nicht allzu schwierig ein, ein Skript zu schreiben, das die Artikel importiert.

XML-Streaming

Doch der Reihe nach. Das Parsing dieser riesigen XML-Dateien verlangt ein Streaming a la SAX, weshalb wir damit beginnen wollen. Die grundlegende Form des Parsings einer XML-Datei (Datensatz für Datensatz) sieht bei JRuby in etwa so aus:

hbase/basic_xml_parsing.rb

```
import 'javax.xml.stream.XMLStreamConstants'

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

while reader.has_next

  type = reader.next

  if type == XMLStreamConstants::START_ELEMENT
    tag = reader.local_name
    # verarbeite Tag
  elsif type == XMLStreamConstants::CHARACTERS
    text = reader.text
    # verarbeite den Text
  elsif type == XMLStreamConstants::END_ELEMENT
    # wie START_ELEMENT
  end

end
```

Sieht man sich das genauer an, sind hier ein paar Dinge erwähnenswert. Zuerst erzeugen wir einen `XMLStreamReader` und verknüpfen ihn mit `java.lang.System.in`, d. h., das Einlesen erfolgt über die Standardeingabe.

Als Nächstes folgt eine `while`-Schleife, die fortlaufend Token aus dem XML-Stream einliest, bis keine mehr übrig sind. Innerhalb der `while`-Schleife verarbeiten wir das aktuelle Token. Was wir genau machen, hängt davon ab, ob das Token am Anfang eines XML-Tags steht oder an dessen Ende oder ob es sich um den Text dazwischen handelt.

Die Wikipedia verarbeiten

Nun können wir diese grundlegende Verarbeitung von XML mit unserer Erläuterung der `HTable`- und `Put`-Schnittstellen kombinieren. Hier das resultierende Skript. Das meiste sollte Ihnen vertraut sein und wir gehen auf die wenigen neuen Teile genauer ein.

hbase/import_from_wikipedia.rb

```
require 'time'

import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'javax.xml.stream.XMLStreamConstants'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

① document = nil
buffer = nil
count = 0

table = HTable.new( @hbase.configuration, 'wiki' )


② table.setAutoFlush( false )

while reader.has_next
  type = reader.next

③   if type == XMLStreamConstants::START_ELEMENT

      case reader.local_name
      when 'page' then document = {}
      when /title|timestamp|username|comment|text/ then buffer = []
      end

④   elsif type == XMLStreamConstants::CHARACTERS

      buffer << reader.text unless buffer.nil?

⑤   elsif type == XMLStreamConstants::END_ELEMENT

      case reader.local_name
      when /title|timestamp|username|comment|text/
        document[reader.local_name] = buffer.join
```

```
    when 'revision'
      key = document['title'].to_java_bytes
      ts = ( Time.parse document['timestamp'] ).to_i

      p = Put.new( key, ts )
      p.add( *jbytes( "text", "", document['text'] ) )
      p.add( *jbytes( "revision", "author", document['username'] ) )
      p.add( *jbytes( "revision", "comment", document['comment'] ) )
      table.put( p )

      count += 1
      table.flushCommits() if count % 10 == 0
      if count % 500 == 0
        puts "#{count} records inserted (#{document['title']})"
      end

    end

  end
end

table.flushCommits()
exit
```

① Der erste bemerkenswerte Unterschied ist die Einführung einiger Variablen:

- `document`: Enthält die aktuellen Artikel- und Revisionsdaten
- `buffer`: Enthält Zeichendaten des aktuellen Feldes innerhalb des Dokuments (Text, Titel, Autor und so weiter)
- `count`: Hält nach, wie viele Artikel wir bislang importiert haben

② Beachten Sie insbesondere unsere Nutzung von `table.setAutoFlush (false)`. Bei HBase werden Daten *automatisch* in regelmäßigen Zeitabständen auf die Platte geschrieben (Autoflushing). Das wird von den meisten Anwendungen bevorzugt. Durch die Deaktivierung von Autoflush in unserem Skript werden alle `put`-Operationen gepuffert, bis wir `table.flushCommits` aufrufen. Auf diese Weise können wir Schreiboperationen bündeln und dann ausführen, wenn es uns gelegen kommt.

③ Als Nächstes sehen wir uns an, was beim Parsing passiert. Wenn der Start-Tag `page` ist, setzen wir `document` auf einen leeren Hash zurück. Wenn es sich um einen anderen Tag handelt, der uns interessiert, setzen wir den `buffer` zurück, um dessen Text festzuhalten.

④ Wir verarbeiten die Zeichendaten, indem wir sie an den Puffer anhängen.

⑤ Bei den meisten schließenden Tags schieben wir den gepufferten Inhalt einfach in das `document`. Ist der schließende Tag eine `</revision>`, erzeugen wir hingegen eine neue `Put`-Instanz, füllen sie mit den Feldern des `documents` und senden sie an die Tabelle. Danach nutzen wir `flushCommits`, wenn wir das eine Zeit lang nicht getan haben, und geben den Fortschritt über die Standardausgabe aus (`puts`).

Komprimierung und Bloomfilter

Wir können das Skript fast ausführen, müssen vorher aber noch ein wenig aufräumen. Die Spaltenfamilie `text` wird lange Texte enthalten und könnte von einer Komprimierung profitieren. Wir wollen daher die Komprimierung und schnelle Lookups aktivieren:

```
hbase> alter 'wiki', {NAME=>'text', COMPRESSION=>'GZ', BLOOMFILTER=>'ROW'}
0 row(s) in 0.0510 seconds
```

HBase unterstützt zwei Komprimierungsalgorithmen: Gzip (GZ) und Lempel-Ziv-Oberhumer (LZO). Die HBase-Community bevorzugt eindeutig LZO, aber wir verwenden hier GZ.

Das Problem mit LZO ist die Lizenz der Implementierung. Zwar handelt es sich um Open Source, aber sie ist nicht mit Apaches Lizenzphilosophie kompatibel, so dass LZO nicht mit HBase gebündelt werden kann. Zur Installation und Konfiguration der LZO-Unterstützung sind online detaillierte Anweisungen vorhanden. Wenn Sie eine Hochleistungskomprimierung brauchen, besorgen Sie sich LZO.

Ein Bloomfilter ist eine wirklich coole Datenstruktur, die sehr effizient die Frage „Habe ich das schon mal gesehen?“ beantworten kann. Ursprünglich in den 1970ern von Burton Howard Bloom zur Rechtschreibprüfung entwickelt, wurden sie in Datenspeicher-Anwendungen beliebt, um schnell herauszufinden, ob ein Schlüssel existiert. Wenn Sie mit Bloomfiltern nicht vertraut sind, finden Sie in *Wie arbeiten Bloomfilter?* eine kurze Einführung.

Wie arbeiten Bloomfilter?

Wir wollen nicht allzu detailliert auf die Implementierungs-Details eingehen, aber grundsätzlich verwaltet ein Bloomfilter ein statisch dimensioniertes Bit-Array, das zu Beginn auf 0 gesetzt ist. Jedes Mal, wenn der Filter neue Daten vorgesetzt bekommt, werden einige dieser Bits auf 1 gesetzt. Zum Setzen dieser Bits wird ein Hash aus den Daten erzeugt, und dieser Hash wird dann auf einen Satz von Bitpositionen abgebildet.

Soll später geprüft werden, ob der Filter bestimmte Daten schon einmal gesehen hat, berechnet er, welche Bits auf 1 gesetzt werden müssen und vergleicht sie dann. Stehen einige auf 0, kann er eindeutig mit „Nein“ antworten. Sind alle auf 1 gesetzt, antwortet er mit „Ja“. Die Chancen stehen dann gut, dass er die Daten schon einmal gesehen hat, doch die falschen Positive erhöhen sich, je mehr Daten er verarbeitet hat.

Das ist der Nachteil des Bloomfilters gegenüber einem einfachen Hash. Ein Hash erzeugt niemals falsche Positive, aber der zur Speicherung der Daten benötigte Platz ist nicht begrenzt. Bloomfilter verwenden eine konstante Menge an Platz, erzeugen aber (basierend auf der Sättigung) eine vorhersehbare Menge falscher Positive.

HBase unterstützt die Verwendung von Bloomfiltern, um zu bestimmen, ob eine bestimmte Spalte für einen gegebenen Zeilenschlüssel (`BLOOMFILTER=> 'ROWCOL'`) existiert oder einfach, ob ein gegebenen Zeilenschlüssel überhaupt existiert (`BLOOMFILTER=>'ROW'`). Die Anzahl der Spalten innerhalb einer Spaltenfamilie und die Anzahl der Zeilen sind beide potentiell unbegrenzt. Bloomfilter bieten eine schnelle Möglichkeit herauszufinden, ob Daten existieren, bevor ein teures Lesen von der Festplatten nötig wird.

Und los!

Nun sind wir bereit, das Skript zu starten. Denken Sie daran, dass diese Dateien riesig sind, d. h., das Herunterladen und Entpacken kommt nicht in Frage. Was machen wir also?

Glücklicherweise können wir Dank der Magie der *nix-Pipes die XML-Datei in einem Rutsch herunterladen, extrahieren und an das Skript übergeben. Der Befehl sieht wie folgt aus:

```
curl <dump_url> | bzcat | \
${HBASE_HOME}/bin/hbase shell import_from_wikipedia.rb
```

Den `<dump_url>` müssen Sie natürlich durch die URL des gewünschten WikiMedia Foundation-Dumps ersetzen.[2] Sie müssen `[projekt]-latest-pages-articles.xml.bz2` für die englische Wikipedia (~6GB)[3] oder das englische Wictionary (~185MB) verwenden.[4] Diese Dateien enthalten die neuesten Seiten des `Main`-Namensraums, d. h., die Seiten von Benutzern und Diskussionen sind nicht dabei.

Tragen Sie die URL ein und führen Sie das Skript aus! Die Ausgabe sollte (irgendwann) etwa so aussehen:

```
500 records inserted (Ashmore and Cartier Islands)
1000 records inserted (Annealing)
1500 records inserted (Ajanta Caves)
```

Das Skript wird zufrieden vor sich hin laufen, solange Sie es lassen oder bis ein Fehler auftritt, wahrscheinlich werden Sie es aber nach einer Weile abbrechen wollen. Wenn Sie so weit sind, das Skript zu beenden, drücken Sie `Ctrl+C`. Im Moment wollen wir es aber noch laufen lassen, um einen Blick hinter die Kulissen zu werfen und herauszufinden, wie HBase seine horizontale Skalierbarkeit erreicht.

2. http://dumps.wikimedia.org
3. http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2
4. http://dumps.wikimedia.org/enwiktionary/latest/enwiktionary-latest-pages-articles.xml.bz2

Einführung in Regionen und Überwachung der Platten-Nutzung

Bei HBase werden Zeilen nach Zeilenschlüssel sortiert vorgehalten. Eine Region ist ein Segment von Zeilen, die über den Start- (inklusive) und Endschlüssel (exklusive) identifiziert wird. Regionen überlappen sich niemals und jede Region wird einem bestimmten Regions-Server im Cluster zugewiesen. Bei unserem einfachen Standalone-Server gibt es nur einen Regions-Server und der ist immer für alle Regionen verantwortlich. Ein vollständig verteilter Cluster kann aus vielen Regions-Servern bestehen.

Sehen wir uns also mal die Plattennutzung unseres HBase-Servers an. Das zeigt uns, wie die Daten verteilt werden. Sie können die Plattennutzung von HBase untersuchen, indem Sie in der Shell in das früher festgelegte `hbase.rootdir` wechseln und den Befehl `du` ausführen. `du` ist ein Standard-Utility bei *nix-Systemen, das Ihnen rekursiv anzeigt, wie viel Platz ein Verzeichnis und seine Unterverzeichnisse nutzen. Die Option `-h` weist `du` an, die Zahlen in einer für uns Menschen verständlichen Form auszugeben.

Nach dem Einfügen von etwa 12000 Seiten und immer noch laufendem Import sah das bei uns wie folgt aus:

```
$ du -h
231M ./.logs/localhost.localdomain,38556,1300092965081
231M ./.logs
4.0K ./.META./1028785192/info
12K  ./.META./1028785192/.oldlogs
28K  ./.META./1028785192
32K  ./.META.
12K  ./-ROOT-/70236052/info
12K  ./-ROOT-/70236052/.oldlogs
36K  ./-ROOT-/70236052
40K  ./-ROOT-
72M  ./wiki/517496fecabb7d16af7573fc37257905/text
1.7M ./wiki/517496fecabb7d16af7573fc37257905/revision
61M  ./wiki/517496fecabb7d16af7573fc37257905/.tmp
12K  ./wiki/517496fecabb7d16af7573fc37257905/.oldlogs
134M ./wiki/517496fecabb7d16af7573fc37257905
134M ./wiki
4.0K ./.oldlogs
365M .
```

Diese Ausgabe sagt uns sehr viel darüber, wie viel Platz HBase nutzt, und wie er alloziert wird. Die mit `/wiki` beginnenden Zeilen beschreiben den von der Wiki-Tabelle genutzten Platz. Das Unterverzeichnis mit dem langen Namen `517496fecabb7d16af7573fc37257905` repräsentiert eine individuelle Region (die bisher einzige Region). Darunter entsprechen die Verzeichnisse `/text` und `/revision` den Spaltenfamilien `text` bzw. `revision`. Die letzte Zeile summiert schließlich all diese Werte und sagt uns, dass HBase 365MB Plattenplatz nutzt.

Eine Sache noch: Die ersten beiden mit `/.logs` beginnenden Zeilen am Anfang der Ausgabe zeigen den Platz an, der von den sog. WAL-Dateien (Write-Ahead-Log) verwendet wird. HBase nutzt das Write-Ahead-Logging als Schutz vor ausfallenden Knoten. Das ist eine recht typische Disaster-Recovery-Technik. So wird das Write-Ahead-Logging bei Dateisystemen beispielsweise *Journaling* genannt. Bei HBase werden die Logs an den WAL angehangen, bevor Editier-Operationen (put und increment) auf der Platte festgehalten werden.

Aus Performance-Gründen werden Edits nicht unbedingt direkt auf die Festplatte geschrieben. Das System läuft wesentlich besser, wenn die E/A-Operationen gepuffert und in größeren Blöcken geschrieben werden. Wenn der für die Region verantwortliche *Regions-Server* in diesem Übergangsstadium abstürzt, nutzt HBase den WAL, um herauszufinden, welche Operationen erfolgreich waren, und nimmt entsprechende Korrekturen vor.

Das Schreiben in den WAL ist optional und standardmäßig aktiviert. Edit-Klassen wie `Put` und `Increment` besitzen eine Setter-Methode namens `setWriteToWAL`, mit der man unterbinden kann, dass die Operation in den WAL geschrieben wird. Generell sollten Sie an der Standard-Einstellung festhalten, doch in manchen Fällen kann es sinnvoll sein, das zu ändern. Wenn Sie beispielsweise einen Import-Job laufen haben, den Sie jederzeit wiederholen können (wie etwa unser Wikipedia-Import-Skript), dann könnten Sie den Performance-Vorteil durch das Abschalten des WAL der Disaster-Recovery vorziehen.

Regionale Fragen

Wenn Sie das Skript lange genug laufen lassen, teilt HBase die Tabelle in mehrere Regionen auf. Hier noch einmal die Ausgabe von `du`, aber diesmal nach etwa 150000 eingefügten Seiten:

```
$ du -h
40K  ./.logs/localhost.localdomain,55922,1300094776865
44K  ./.logs
24K  ./.META./1028785192/info
4.0K ./.META./1028785192/recovered.edits
4.0K ./.META./1028785192/.tmp
12K  ./.META./1028785192/.oldlogs
56K  ./.META./1028785192
60K  ./.META.
4.0K ./.corrupt
12K  ./-ROOT-/70236052/info
4.0K ./-ROOT-/70236052/recovered.edits
4.0K ./-ROOT-/70236052/.tmp
12K  ./-ROOT-/70236052/.oldlogs
44K  ./-ROOT-/70236052
48K  ./-ROOT-
138M ./wiki/0a25ac7e5d0be211b9e890e83e24e458/text
5.8M ./wiki/0a25ac7e5d0be211b9e890e83e24e458/revision
4.0K ./wiki/0a25ac7e5d0be211b9e890e83e24e458/.tmp
144M ./wiki/0a25ac7e5d0be211b9e890e83e24e458
149M ./wiki/15be59b7dfd6e71af9b828fed280ce8a/text
6.5M ./wiki/15be59b7dfd6e71af9b828fed280ce8a/revision
4.0K ./wiki/15be59b7dfd6e71af9b828fed280ce8a/.tmp
155M ./wiki/15be59b7dfd6e71af9b828fed280ce8a
145M ./wiki/0ef3903982fd9478e09d8f17b7a5f987/text
6.3M ./wiki/0ef3903982fd9478e09d8f17b7a5f987/revision
4.0K ./wiki/0ef3903982fd9478e09d8f17b7a5f987/.tmp
151M ./wiki/0ef3903982fd9478e09d8f17b7a5f987
135M ./wiki/a79c0f6896c005711cf6a4448775a33b/text
6.0M ./wiki/a79c0f6896c005711cf6a4448775a33b/revision
4.0K ./wiki/a79c0f6896c005711cf6a4448775a33b/.tmp
141M ./wiki/a79c0f6896c005711cf6a4448775a33b
591M ./wiki
4.0K ./.oldlogs
591M .
```

Der größte Unterschied besteht darin, dass die alte Region (`517496fecabb7d16af7573fc37257905`) verschwunden und durch vier neue ersetzt wurde. Bei unserem Standalone-Server werden alle Regionen durch diesen einzigen Server bedient, doch in einer verteilten Umgebung würden sie auf die verschiedenen Regions-Server verteilt werden.

Das führt zu einigen Fragen wie etwa „Woher wissen die Regions-Server, für welche Regionen sie verantwortlich sind?“ und „Wie kann man herausfinden, welche Region (und welcher Regions-Server) eine bestimmte Zeile liefert?“

Wenn wir noch einmal zur HBase-Shell zurückkehren, können wir die `.META.`-Tabelle abfragen, um mehr über die aktuellen Regionen herauszufinden. `.META.` ist eine spezielle Tabelle, deren einziger Zweck darin besteht, alle Benutzertabellen nachzuhalten sowie die Regions-Server, die die Regionen dieser Tabellen bedienen.

```
hbase> scan '.META.', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
```

Selbst bei einer kleinen Anzahl von Regionen erhalten Sie eine sehr umfangreiche Ausgabe. Hier ein Teil unserer Ausgabe, der der Lesbarkeit halber formatiert und abgeschnitten wurde:

```
ROW
 wiki„1300099733696.a79c0f6896c005711cf6a4448775a33b.

COLUMN+CELL
 column=info:server, timestamp=1300333136393, value=localhost.localdomain:3555
 column=info:regioninfo, timestamp=1300099734090, value=REGION => {
 NAME => 'wiki„1300099733696.a79c0f6896c005711cf6a4448775a33b.',
 STARTKEY => '',
 ENDKEY => 'Demographics of Macedonia',
 ENCODED => a79c0f6896c005711cf6a4448775a33b,
 TABLE => {{...}}

ROW
 wiki,Demographics of Macedonia,1300099733696.0a25ac7e5d0be211b9e890e83e24e458.

COLUMN+CELL
 column=info:server, timestamp=1300333136402, value=localhost.localdomain:35552
 column=info:regioninfo, timestamp=1300099734011, value=REGION => {
   NAME => 'wiki,Demographics of Macedonia,1300099733696.0a25...e458.',
   STARTKEY => 'Demographics of Macedonia',
   ENDKEY => 'June 30',
   ENCODED => 0a25ac7e5d0be211b9e890e83e24e458,
   TABLE => {{...}}
```

Beide oben aufgeführten Regionen werden vom gleichen Server bedient: `localhost.localdomain:35552`. Die erste Region beginnt mit dem Leerstring (`''`) und endet mit `'Demographics of Macedonia'`. Die zweite Region beginnt bei `'Demographics of Macedonia'` und geht bis `'June 30'`.

`STARTKEY` ist inklusive, `ENDKEY` exklusive. Wenn wir also die Zeile `'Demographics of Macedonia'` suchen, finden wir sie in der zweiten Region.

Da Zeilen sortiert vorgehalten werden, können wir die in `.META.` gespeicherten Informationen nutzen, um die Region und den Server zu ermitteln, auf dem eine Zeile zu finden ist. Doch wo wird diese `.META.`-Tabelle gespeichert?

Wie sich herausstellt, wird die `.META.`-Tabelle wie jede andere Tabelle auch in Regionen aufgeteilt, die von Regions-Servern bedient werden. Um herauszufinden, welche Server welche Teile der `.META.`-Tabelle enthalten, müssen wir `-ROOT-` durchsuchen.

```
hbase> scan '-ROOT-', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
ROW  .META.,,1
COLUMN+CELL
  column=info:server, timestamp=1300333135782, value=localhost.localdomain:35552
  column=info:regioninfo, timestamp=1300092965825, value=REGION => {
    NAME => '.META.,,1',
    STARTKEY => '',
    ENDKEY => '',
```

```
ENCODED => 1028785192,
TABLE => {{...}}
```

Die Zuweisung von Regionen an Regions-Server, einschließlich der `.META.`-Regionen, übernimmt der *Master*-Knoten, der häufig als `HBaseMaster` bezeichnet wird. Der Master-Server kann auch ein Regions-Server sein und beide Aufgaben gleichzeitig übernehmen.

Fällt ein Regions-Server aus, schreitet der Master-Server ein und weist die dem ausgefallenen Knoten zugewiesenen Regionen anderen Servern zu. Die neuen Verwalter dieser Regionen sehen sich den WAL an, um festzustellen, ob irgendwelche Recovery-Schritte notwendig sind. Fällt der Master-Server aus, wird die Verantwortung an einen der anderen Regions-Server übertragen, der sich dann anschickt, die Stelle des Masters zu übernehmen.

Eine Tabelle scannen, um eine andere zu erzeugen

Wenn Sie das Import-Skript beendet haben, können wir uns der nächsten Aufgabe zuwenden: Informationen aus den importierten Wiki-Inhalten extrahieren.

Die Wiki-Syntax ist voller Links, von denen einige intern auf andere Artikel und andere auf externe Ressourcen verweisen. Diese Verlinkung enthält eine Fülle topologischer Daten, die wir abgreifen wollen.

Unser Ziel ist es, die Beziehungen zwischen Artikeln als gerichtete Links festzuhalten, die von einem Artikel *zu* einem anderen zeigen oder einen Link *von* einem anderen empfangen. Ein interner Artikel-Link sieht in Wikitext wie folgt aus: *[[<zielname>|<alttext>]]*, wobei *<zielname>* der Artikel ist, auf den verlinkt wird, und *<alttext>* der (optionale) Alternativtext, der ausgegeben wird.

Wo ist mein TABLE-Schema?

Das `TABLE`-Schema wurde aus der Beispiel-Ausgabe der `regioninfo`-Scans entfernt. Dadurch ist das Ganze nicht so überladen und wir unterhalten uns später noch über Performance-Tuning-Optionen. Wenn Sie die Schema-Definition einer Tabelle sehen wollen, verwenden Sie den `describe`-Befehl:

```
hbase> describe 'wiki'
hbase> describe '.META.'
hbase> describe '-ROOT-'
```

Wenn zum Beispiel der Text im *Star Wars*-Artikel den String *"[[Yoda|jedi master]]"* enthält, wollen wir diese Beziehung zweimal festhalten – einmal als aus-

gehenden Link von *Star Wars* und einmal als eingehenden Link bei Yoda. Die Beziehung zweimal zu speichern, macht den Lookup sowohl der ausgehenden als auch der eingehenden Links einer Seite sehr schnell.

Um die zusätzlichen Linkdaten zu speichern, legen wir eine neue Tabelle an. Wechseln Sie in die Shell und geben Sie Folgendes ein:

```
hbase> create 'links', {
 NAME => 'to', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
},{
 NAME => 'from', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
}
```

Im Prinzip hätten wir die Linkdaten auch in die existierende Spaltenfamilie einfügen oder ein oder mehrere zusätzliche Spaltenfamilien in die Wiki-Tabelle aufnehmen können. Eine separate Tabelle anzulegen, hat aber den Vorteil, dass die Tabelle separate Regionen besitzt. Das bedeutet, dass der Cluster die Regionen ganz nach Bedarf effektiver aufteilen kann.

Der generelle Rat der HBase-Community in Bezug auf Spaltenfamilien lautet, die Anzahl der Familien pro Tabelle klein zu halten. Sie erreichen das, indem Sie mehr Spalten in der gleichen Familie unterbringen oder indem Sie die Familien gleich in andere Tabellen auslagern. Die Entscheidung hängt größtenteils davon ab, ob und wie oft Clients ganze Datenzeilen (im Gegensatz zu einigen wenigen Spaltenwerten) benötigen.

Im Fall unseres Wikis müssen die `text`- und `revision`-Spaltenfamilien in der gleichen Tabelle liegen, damit beim Einfügen neuer Revisionen Metadaten und Text den gleichen Zeitstempel haben. Im Gegensatz dazu hat der `links`-Inhalt niemals den gleichen Zeitstempel wie der Artikel, von dem die Daten stammen. Darüber hinaus werden die meisten Clients am Artikeltext oder an den Informationen über die Links interessiert sein, nicht aber an beiden gleichzeitig. Das Ausgliedern der `to`- und `from`-Spaltenfamilien in eine separate Tabelle ist also durchaus sinnvoll.

Den Scanner aufbauen

Nachdem wir die `links`-Tabelle angelegt haben, können wir ein Skript implementieren, das alle Zeilen der `wiki`-Tabelle scannt. Für jede Zeile rufen wir dann den Wikitext ab und extrahieren die Links. Abschließend erzeugen wir für jeden gefundenen Link ein- und ausgehende Einträge in der `link`-Tabelle. Ein Großteil des Skripts sollte Ihnen nun vertraut sein. Das meiste haben wir recycelt, und auf die wenigen neuen Elemente gehen wir gleich ein.

code/hbase/generate_wiki_links.rb

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'org.apache.hadoop.hbase.client.Scan'
import 'org.apache.hadoop.hbase.util.Bytes'

def jbytes( *args )
  return args.map { |arg| arg.to_s.to_java_bytes }
end

wiki_table = HTable.new( @hbase.configuration, 'wiki' )
links_table = HTable.new( @hbase.configuration, 'links' )
links_table.setAutoFlush( false )

① scanner = wiki_table.getScanner( Scan.new )

linkpattern = /\[\[([^\[\]\|\:\#][^\[\]\|:]*)(?:\|([^\[\]\|]+))?\]\]/
count = 0

while (result = scanner.next())

②   title = Bytes.toString( result.getRow() )
    text = Bytes.toString( result.getValue( *jbytes( 'text', '' ) ) )
    if text

      put_to = nil
③     text.scan(linkpattern) do |target, label|
        unless put_to
          put_to = Put.new( *jbytes( title ) )
          put_to.setWriteToWAL( false )
        end

        target.strip!
        target.capitalize!

        label = '' unless label
        label.strip!

        put_to.add( *jbytes( "to", target, label ) )
        put_from = Put.new( *jbytes( target ) )
        put_from.add( *jbytes( "from", title, label ) )
        put_from.setWriteToWAL( false )
④       links_table.put( put_from )
      end

⑤     links_table.put( put_to ) if put_to
      links_table.flushCommits()
    end
    count += 1
    puts "#{count} pages processed (#{title})" if count % 500 == 0

end
links_table.flushCommits()
exit
```

① Zuerst erzeugen wir ein `Scan`-Objekt, über das wir die Wiki-Tabelle durchgehen.

② Um die Zeilen- und Spaltendaten zu extrahieren, müssen wir ein wenig mit den Bytes jonglieren, aber auch das ist generell nicht besonders schwierig.

③ Jedes Mal, wenn das `linkpattern` im Seitentext auftaucht, extrahieren wir den Zielartikel und den Text des Links und fügen diese Werte dann zu unseren `Put`-Instanzen hinzu.

④ Abschließend weisen wir die Tabelle an, die akkumulierte `Put`-Operationen auszuführen. Es ist möglich (wenn auch unwahrscheinlich), dass ein Artikel keine Links enthält. Das ist der Grund für die `if put_to`-Klausel.

⑤ Die Verwendung von `setWriteToWAL(false)` für diese Puts war eine bewusste Entscheidung. Da es sich um eine Übung mit didaktischem Hintergrund handelt und weil wir das Skript einfach erneut ausführen können, wenn etwas schiefgeht, entscheiden wir uns für den Geschwindigkeitsvorteil und akzeptieren unser Schicksal, wenn der Knoten ausfallen sollte.

Das Skript ausführen

Wenn Sie bereit sind und alle Vorsicht unbekümmert in den Wind schießen, starten Sie das Skript.

```
${HBASE_HOME}/bin/hbase shell generate_wiki_links.rb
```

Die Ausgabe sollte etwa so aussehen:

```
500  pages processed (10 petametres)
1000 pages processed (1259)
1500 pages processed (1471 BC)
2000 pages processed (1683)
...
```

Genau wie das vorige Skript können Sie es so lange laufen lassen, wie Sie wollen, sogar bis zum Ende. Wenn Sie es unterbrechen wollen, drücken Sie `Ctrl+C`.

Sie können die Plattennutzung des Skripts wie vorhin mit `du` überwachen. Sie werden neue Einträge für die von uns angelegte `links`-Tabelle finden und die Größen nehmen ständig zu, je länger das Skript läuft.

Joe fragt:
Hätten wir das nicht mit Mapreduce erledigen können?

In der Einführung haben wir erklärt, dass unsere Beispiele (J)Ruby und JavaScript nutzen. JRuby harmoniert nicht besonders gut mit Hadoop, doch wenn Sie Mapreduce mittels Java nutzen wollten, hätten Sie diesen Scanner-Code als Mapreduce-Job entwickelt und an Hadoop übergeben.

Im Allgemeinen sind Aufgaben wie diese ideal für Mapreduce-Implementierungen geeignet. Es gibt eine riesige Menge Eingangsdaten in einem regulären Format, der von einem Mapper verarbeitet werden kann (Scannen einer HBase-Tabelle) und eine riesige Menge Ausgangsoperationen, die von einem Reducer übernommenwerden können (Schreiben der Zeilen an eine HBase-Tabelle).

Die Hadoop-Architektur verlangt, dass `Job`-Instanzen in Java geschrieben und (einschließlich aller Abhängigkeiten) in einer jar-Datei gekapselt sind, die an alle Knoten des Clusters gesendet werden kann. Neuere JRuby-Versionen können Java-Klassen erweitern, doch die mit HBase ausgelieferte Version kann das nicht.

Es gibt einige Open-Source-Projekte, die die Ausführung von JRuby unter Hadoop ermöglichen, doch bisher gibt es nichts, was mit HBase besonders gut funktioniert. Es gibt Gerüchte, nach denen die HBase-Infrastruktur zukünftig Abstraktionen enthalten wird, die JRuby MR- (Mapreduce-) Jobs ermöglichen. Es gibt also Hoffnung.

Das Ergebnis prüfen

Wir haben gerade ein Scanner-Programm entwickelt, das eine anspruchsvolle Aufgabe löst. Nun wollen wir den `scan`-Befehl der Shell nutzen, um uns einfach einen Teil des Tabelleninhalts an der Console ausgeben zu lassen. Für jeden Link, den das Skript in einem `text:`-Blob findet, erzeugt es einen `to`- und einen `from`-Eintrag in der `links`-Tabelle. Um sich die Links anzusehen, die auf diese Weise erzeugt werden, wechseln Sie in die Shell und `scan`nen die Tabelle.

```
hbase> scan 'links', STARTROW => "Admiral Ackbar", ENDROW => "It's a Trap!"
```

Die Ausgabe wird recht umfangreich sein, aber natürlich können Sie den `get`-Befehl nutzen, um sich die Links für einen einzelnen Artikel anzusehen.

```
hbase> get 'links', 'Star Wars'
COLUMN CELL
 ...
 links:from:Admiral Ackbar         timestamp=1300415922636, value=
 links:from:Adventure              timestamp=1300415927098, value=
 links:from:Alamogordo, New Mexico timestamp=1300415953549, value=
```

```
links:to:"weird al" yankovic      timestamp=1300419602350, value=
links:to:20th century fox         timestamp=1300419602350, value=

links:to:3-d film                 timestamp=1300419602350, value=
links:to:Aayla secura             timestamp=1300419602350, value=
...
```

In der `wiki`-Tabelle sind die Zeilen in Bezug auf die Spalten sehr gleichmäßig. Jede Zeile besteht aus den Spalten `text:`, `revision:author` und `revision:comment`. Diese Regelmäßigkeit gibt es in der `links`-Tabelle nicht. Jede Zeile kann eine oder hunderte Spalten enthalten. Und die Vielzahl der Spaltennamen ist so unterschiedlich wie die Spaltenschlüssel selbst (Titel von Wikipedia-Artikeln). Aber das ist in Ordnung! HBase ist aus eben diesem Grund ein sog. dünnbesetzter (sparse) Datenspeicher.

Um herauszufinden, wie viele Zeilen Ihre Tabelle enthält, können Sie den `count`-Befehl verwenden.

```
hbase> count 'wiki', INTERVAL => 100000, CACHE => 10000
Current count: 100000,  row: Alexander wilson (vauxhall)
Current count: 200000,  row: Bachelor of liberal studies
Current count: 300000,  row: Brian donlevy
...
Current count: 2000000, row: Thomas Hobbes
Current count: 2100000, row: Vardousia
Current count: 2200000, row: Wörrstadt (verbandsgemeinde)
2256081 row(s) in 173.8120 seconds
```

Aufgrund seiner verteilten Architektur kann HBase nicht direkt wissen, wie viele Zeilen eine Tabelle enthält. Um das herauszufinden, muss es sie (über einen Tabellen-Scan) zählen. Glücklicherweise eignet sich die regionenbasierte Speicherarchitektur von HBase zum schnellen verteilten Scanning. Selbst wenn die fragliche Operation einen Tabellen-Scan verlangt, müssen wir uns (im Gegensatz zu anderen Datenbanken) keine allzugroßen Sorgen machen.

Was wir am zweiten Tag gelernt haben

Puh, das war ein anstrengender Tag! Wir haben gelernt, wie man ein Import-Skript für HBase entwickelt, das Daten aus einem XML-Stream verarbeitet. Dann haben wir diese Techniken genutzt, um Wikipedia-Dumps direkt in unsere `wiki`-Tabelle zu importieren.

Wir haben mehr über die HBase-API erfahren, einschließlich einiger Client-gesteuerter Performance-Hebel wie `setAutoFlush()`, `flushCommits()` und `setWriteToWAL()`. Darüber hinaus haben wir einige architektonische Features von HBase diskutiert, etwa das Disaster-Recovery durch Write-Ahead-Logs.

Apropos Architektur. Wir haben Tabellen-Regionen kennengelernt und entdeckt, wie HBase die Verantwortung für sie auf die Regions-Server im Cluster verteilt. Wir haben die Tabellen `.META.` und `-ROOT-` untersucht, um ein Gefühl für die HBase-Interna zu bekommen.

Abschließend haben wir diskutiert, wie sich das dünnbesetzte Design von HBase auf die Performance auswirkt. Dabei haben wir auch einige der „Best Practices" der Community zum Umgang mit Spalten, Familien und Tabellen kennengelernt.

Tag 2: Selbststudium

Finden Sie heraus

1. Finden Sie eine Diskussion oder einen Artikel, der die Vor- und Nachteile der Komprimierung in HBase beschreibt.
2. Finden Sie einen Artikel, der die Funktionsweise von Bloomfiltern beschreibt und wie HBase von ihnen profitiert.
3. Welche weiteren Spaltenfamilien-Optionen gibt es (neben dem verwendeten Algorithmus) im Zusammenhang mit der Komprimierung?
4. Wie beeinflussen die verwendeten Datentypen und Nutzungsmuster die Komprimierungsoptionen der Spaltenfamilien?

Machen Sie Folgendes

Bauen Sie basierend auf der Idee des Datenimports eine Datenbank mit Nährwertangaben auf.

Laden Sie die MyPyramid Raw Food-Daten von Data.gov herunter.[5] Entpacken Sie den Inhalt und suchen Sie `Food_Display_Table.xml` heraus.

Die Daten bestehen aus vielen Paaren von `<Food_Display_Row>`-Tags. Innerhalb dieser enthält jede Zeile einen `<Food_Code>` (Integerwert), einen `<Display_Name>` (String) und andere Fakten zu den Nahrungsmitteln innerhalb entsprechend benannter Tags.

1. Legen Sie eine neue Tabelle namens `foods` mit einer einzelnen Spaltenfamilie an, um die Fakten zu speichern. Was sollten Sie als Spaltenschlüssel verwenden? Welche Spaltenfamilien-Optionen sind für diese Daten sinnvoll?

5. http://explore.data.gov/Health-and-Nutrition/MyPyramid-Food-Raw-Data/b978-7txq

2. Entwickeln Sie ein neues JRuby-Skript zum Import der Daten. Nutzen Sie den SAX-Parser, den wir auch für den Wikipedia-Import genutzt haben, und passen Sie ihn an die Nahrungsmitteldaten an.
3. Übergeben Sie die Daten an Ihr Import-Skript, um die Tabelle zu füllen.
4. Nutzen Sie zum Schluss die HBase-Shell und fragen Sie die `foods`-Tabelle nach Informationen zu Ihren Lieblingslebensmitteln ab.

4.4 Tag 3: Auf dem Weg in die Cloud

An den ersten beiden Tagen haben wir viele praktische Erfahrungen mit HBase im Standalone-Modus gesammelt. Unsere Experimente haben sich bisher auf den Zugriff auf einen einzelnen lokalen Server beschränkt. Im richtigen Leben werden Sie, wenn Sie sich für HBase entscheiden, einen gut dimensionierten Cluster nutzen, um die Performance-Vorteile der verteilten Architektur ausschöpfen zu können.

Am dritten Tag wollen wir uns dem Betrieb und der Interaktion mit einem entfernten HBase-Cluster widmen. Zuerst entwickeln wir eine Client-Anwendung in Ruby und stellen damit die Verbindung zu unserem lokalen Server her. Dazu nutzen wir ein Protokoll namens Thrift. Dann richten wir einen Multinode-Cluster bei einem Cloud-Serviceprovider – Amazon EC2 – ein. Dabei verwenden wir eine Cluster-Management-Technik namens Apache Whirr.

Eine „sparsame" HBase-Anwendung entwickeln

Bisher haben wir die HBase-Shell genutzt, doch HBase unterstützt eine Reihe von Protokollen zur Client-Kommunikation. Hier die vollständige Liste:

Name	Verbindungstyp	Produktionstauglich?
Shell	Direkt	Ja
Java API	Direkt	Ja
Thrift	Binärprotokoll	Ja
REST	HTTP	Ja
Avro	Binärprotokoll	Nein (experimentell)

In der obigen Tabelle beschreibt der Verbindungstyp, ob das Protokoll direkte Java-Aufrufe vornimmt, Daten über HTTP transportiert oder ein kompaktes Binärformat zur Datenübertragung nutzt. Alle eignen sich für den Produktionseinsatz, nur Avro ist relativ neu und sollte als experimentell betrachtet werden.

Von all diesen Möglichkeiten ist Thrift zur Entwicklung von Client-Anwendungen wohl am weitesten verbreitet. Als ausgereiftes Binärprotokoll mit geringem Overhead wurde Thrift ursprünglich von Facebook entwickelt und als Open Source freigegeben. Später wurde es zu einem Apache Incubator-Projekt. Machen wir Ihren Rechner für den Einsatz von Thrift fit.

Thrift installieren

Wie bei vielen Dingen in der Datenbank-Welt ist auch bei der Arbeit mit Thrift ein Setup notwendig. Um die Verbindung zum HBase-Server über Thrift herstellen zu können, müssen wir Folgendes tun:

1. HBase den Thrift-Service ausführen lassen.
2. Das Thrift-Kommandozeilen-Tool installieren.
3. Bibliotheken für die gewählte Client-Sprache installieren.
4. HBase-Modelldateien für diese Sprache erzeugen.
5. Eine Client-Anwendung entwickeln und ausführen.

Wir starten zuerst den Thrift-Service, was eine einfache Angelegenheit ist. Sie starten den Daemon wie folgt über die Kommandozeile:

```
${HBASE_HOME}/bin/hbase-daemon.sh start thrift -b 127.0.0.1
```

Als Nächstes müssen Sie das `thrift`-Kommandozeilen-Tool installieren. Die dazu notwendigen Schritte hängen stark von Ihrer Umgebung ab und verlangen grundsätzlich die Kompilierung von Binaries. Um zu überprüfen, ob es korrekt installiert wurde, rufen Sie es über die Kommandozeile mit dem Flag `-version` auf. Die Ausgabe sollte etwa so aussehen:

```
$ thrift -version
Thrift version 0.6.0
```

Als Client-Sprache verwenden wir Ruby, auch wenn die Schritte bei anderen Sprachen vergleichbar sind. Installieren Sie das Thrift Ruby-Gem wie folgt über die Kommandozeile:

```
$ gem install thrift
```

Um zu überprüfen, ob das Gem korrekt installiert wurde, können Sie den folgenden Ruby-Einzeiler ausführen:

```
$ ruby -e "require 'thrift'"
```

Wenn es keine Ausgabe gibt, ist das eine gute Nachricht! Eine Fehlermeldung wie „`no such file to load`“ bedeutet, dass Sie mit der Fehlersuche anfangen müssen, bevor Sie weitermachen können.

Die Modelle generieren

Nun generieren wir die sprachspezifischen HBase-Modelldateien. Diese Modelldateien bilden den Kitt, der Ihre spezifische HBase-Version mit der jeweils installierten Thrift-Version verbindet. Das ist auch der Grund, warum sie generiert werden müssen und nicht einfach fertig mitgeliefert werden.

Zuerst machen Sie die Datei `Hbase.thrift` unterhalb des Verzeichnisses `${HBASE_HOME}/src` ausfindig. Der Pfad sollte etwa so aussehen:

```
${HBASE_HOME}/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift
```

Mit dem so ermittelten Pfad erzeugen Sie die Modelldateien über den folgenden Befehl:

```
$ thrift - gen rb <pfad_zu_Hbase.thrift>
```

Damit wird ein neuer Ordner namens `gen-rb` erzeugt, der die folgenden Modelldateien enthält:

- `hbase_constants.rb`
- `hbase.rb`
- `hbase_types.rb`

Wir werden diese Dateien gleich nutzen, wenn wir eine einfache Client-Anwendung entwickeln.

Eine Client-Anwendung entwickeln

Unser Programm stellt die HBase-Verbindung über Thrift her und gibt alle Tabellen, zusammen mit ihren Spaltenfamilien, aus. Das wäre der erste Schritt zur Entwicklung einer Administrationsschnittstelle für HBase. Im Gegensatz zu den früheren Beispielen wird dieses Skript vom guten alten (normalen) Ruby ausgeführt, nicht von JRuby. Es könnte sich beispielsweise in eine Ruby-basierte Web-Anwendung einfügen.

Geben Sie den folgenden Code in einer neuen Textdatei ein (wir nennen sie `thrift_example.rb`):

hbase/thrift_example.rb

```
$:.push('./gen-rb')
require 'thrift'
require 'hbase'

socket = Thrift::Socket.new( 'localhost', 9090 )
transport = Thrift::BufferedTransport.new( socket )
protocol = Thrift::BinaryProtocol.new( transport )
client = Apache::Hadoop::Hbase::Thrift::Hbase::Client.new( protocol )

transport.open()

client.getTableNames().sort.each do |table|
  puts "#{table}"
  client.getColumnDescriptors( table ).each do |col, desc|
    puts " #{desc.name}"
    puts " maxVersions: #{desc.maxVersions}"
    puts " compression: #{desc.compression}"
    puts " bloomFilterType: #{desc.bloomFilterType}"
  end
end

transport.close()
```

Im obigen Code müssen wir zuerst sicherstellen, dass Ruby die Modelldateien findet. Dazu fügen wir `gen-rb` in den Pfad ein und binden `thrift` und `hbase` ein. Danach stellen wir eine Verbindung mit dem Thrift-Server her und verknüpfen sie mit einer HBase-Client-Instanz. Über dieses `client`-Objekt kommunizieren wir mit HBase.

Nachdem der `transport` geöffnet wurde, gehen wir alle Tabellen durch, die von `getTableNames` zurückgeliefert werden. Für jede Tabelle gehen wir die Liste der Spaltenfamilien durch, die uns `getColumnDescriptors()` zurückliefert und geben einige Properties über die Standardausgabe aus.

Nun wollen wir das Programm über die Kommandozeile ausführen. Die Ausgabe sollte bei Ihnen ähnlich aussehen, da wir die Verbindung mit dem lokalen HBase-Server herstellen, den wir vorhin gestartet haben.

```
$> ruby thrift_example.rb
links
  from:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
  to:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
wiki
  revision:
    maxVersions: 2147483647
    compression: NONE
    bloomFilterType: NONE
```

```
  text:
    maxVersions: 2147483647
    compression: GZ
    bloomFilterType: ROW
```

Sie werden feststellen, dass das Thrift-API für HBase größtenteils die gleiche Funktionalität bietet wie die Java-API, viele Konzepte aber anders ausgedrückt werden. Beispielsweise erzeugen Sie bei Thrift anstelle einer `Put`-Instanz eine `Mutation` zur Aktualisierung einer einzelnen Spalte bzw. eine `BatchMutation` für das Update mehrerer Spalten in einer Transaktion.

Die Datei `Hbase.thrift`, die wir vorhin zur Generierung der Modelldateien genutzt haben – siehe Abschnitt *Die Modelle generieren*, auf Seite 136 – , ist sehr gut dokumentiert und beschreibt die Ihnen zur Verfügung stehenden Strukturen und Methoden. Sehen Sie sich die Datei also mal etwas genauer an!

Einführung in Whirr

Die Einrichtung eines funktionierenden Clusters über einen Cloud-Service war früher *sehr viel* Arbeit. Glücklicherweise ändert sich das durch Whirr. Whirr ist momentan ein Apache Incubator-Projekt und stellt Tools zur Verfügung, mit denen Sie Cluster virtueller Maschinen starten, Verbindungen mit ihnen herstellen und wieder löschen können. Es unterstützt beliebte Dienste wie Amazons Elastic Compute Cloud (EC2) und RackSpaces Cloud-Server. Whirr unterstützt momentan den Aufbau von Hadoop-, HBase-, Cassandra-, Voldemort- und ZooKeeper-Clustern und nebenbei auch Techniken wie MongoDB und ElasticSearch.

Obwohl Dienstanbieter wie Amazon häufig Mittel zur Verfügung stellen, die Daten zu erhalten, nachdem die virtuellen Maschinen beendet wurden, werden wir diese Möglichkeit nicht nutzen. Für unsere Zwecke reichen temporäre Cluster aus, die ihre gesamten Daten bei der Beendigung verlieren. Wenn Sie entscheiden, HBase in einer Produktionsumgebung einzusetzen, könnten Sie auch einen persistenten Speicher einrichten wollen. In diesem Fall stellt sich die Frage, ob dedizierte Hardware für Ihre Bedürfnisse nicht besser geeignet ist. Dynamische Dienste wie EC2 eignen sich gut als Arbeitstiere für zwischendurch, doch generell erhalten Sie bei einem Cluster dedizierter physikalischer oder virtueller Maschinen mehr für Ihr Geld.

Einrichtung für EC2

Bevor Sie Whirr nutzen können, um einen Cluster aufzusetzen, benötigen Sie einen Account bei einem unterstützten Cloud-Serviceanbieter. In diesem

Kapitel beschreiben wir, wie man Amazons EC2 nutzt, aber Sie können auch jeden anderen Provider nutzen.

Wenn Sie noch keinen Amazon-Account besitzen, wechseln Sie zu Amazons Web Services-Portal (AWS) und legen Sie einen an.[6] Melden Sie sich an und aktivieren Sie EC2 für Ihren Account, falls es noch nicht aktiviert sein sollte.[7] Zum Schluss öffnen Sie die EC2 AWS-Consolenseite unter Accounts Amazon EC2.[8] Das sollte so aussehen wie in Abbildung 17, *Amazon EC2-Console ohne Instanzen*.

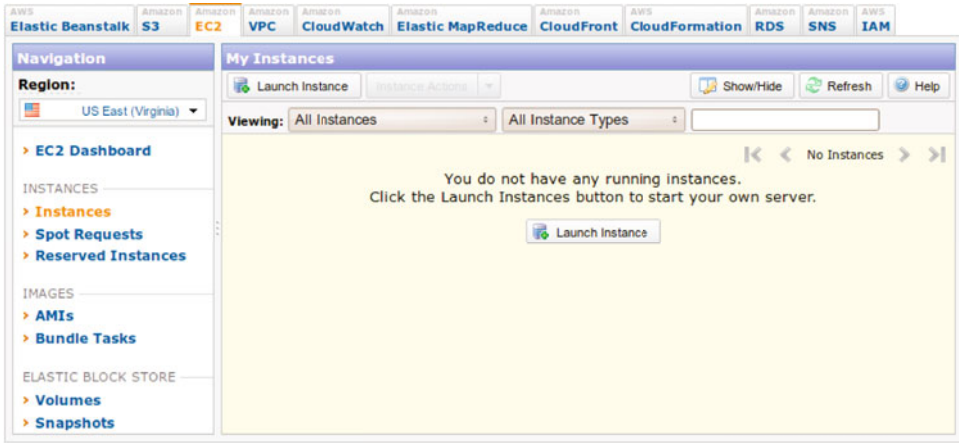


Abbildung 17: Amazon EC2-Console ohne Instanzen

Sie benötigen Ihre AWS-Credentials, um EC2-Knoten starten zu können. Kehren Sie zur AWS-Hauptseite zurück und wählen Sie Account→Security Credentials. Scrollen Sie zum Abschnitt Access Credentials herunter und notieren Sie sich Ihre Access Key ID. Unter Secret Access Key klicken Sie auf Show und notieren sich auch diesen Wert. Diese Schlüssel werden wir später unter `AWS_ACCESS_KEY_ID` und `AWS_SECRET_ACCESS_KEY` eintragen, wenn wir Whirr konfigurieren.

Whirr vorbereiten

Mit den passenden EC2-Credentials zur Hand, beschaffen wir uns Whirr. Wechseln Sie auf die Apache Whirr-Site[9]und laden Sie die neueste Version herunter. Entpacken Sie die heruntergeladene Datei und wechseln Sie in

6. http://aws.amazon.com/
7. http://aws.amazon.com/ec2/
8. https://console.aws.amazon.com/ec2/#s=Instances
9. http://incubator.apache.org/whirr/

einem Terminal in dieses Verzeichnis. In der Kommandozeile können wir mit Hilfe des `version`-Befehls überprüfen, ob Whirr bereit ist.

```
$ bin/whirr version
Apache Whirr 0.6.0-incubating
```

Als Nächstes generieren wir einige passwortfreie SSH-Schlüssel für Whirr, die wir verwenden, wenn wir Instanzen (virtuelle Maschinen) starten.

```
$ mkdir keys
$ ssh-keygen -t rsa -P '' -f keys/id_rsa
```

Das erzeugt ein Verzeichnis namens `keys` und legt darin die Dateien `id_rsa` und `id_rsa.pub` ab. Nachdem wir das erledigt haben, können wir damit beginnen, den Cluster zu konfigurieren.

Den Cluster konfigurieren

Um die Details eines Clusters zu spezifizieren, versorgen wir Whirr mit einer `.properties`-Datei, die alle relevanten Einstellungen enthält. Legen Sie eine Datei im Whirr-Verzeichnis mit dem Namen `hbase.properties` mit dem folgenden Inhalt an (und ersetzen Sie dabei `AWS_ACCESS_KEY_ID` und `AWS_SECRET_ACCESS_KEY` wie oben beschrieben):

hbase/hbase.properties

```
# service provider
whirr.provider=aws-ec2
whirr.identity=Ihr AWS_ACCESS_KEY_ID
whirr.credential=Ihr AWS_SECRET_ACCESS_KEY

# SSH-Credentials
whirr.private-key-file=keys/id_rsa
whirr.public-key-file=keys/id_rsa.pub

# Cluster-Konfiguration
whirr.cluster-name=myhbasecluster
whirr.instance-templates=\
  1 zookeeper+hadoop-namenode+hadoop-jobtracker+hbase-master,\
  5 hadoop-datanode+hadoop-tasktracker+hbase-regionserver

# Konfiguration der HBase- und Hadoop-Version
whirr.hbase.tarball.url=\
  http://apache.cu.be/hbase/hbase-0.90.3/hbase-0.90.3.tar.gz
whirr.hadoop.tarball.url=\
  http://archive.cloudera.com/cdh/3/hadoop-0.20.2-cdh3u1.tar.gz
```

Die ersten beiden Abschnitte legen den Serviceprodiver samt allen relevanten Credentials fest, während die beiden letzten Abschnitte das von uns erzeugte HBase-Cluster spezifizieren. Der `whirr.cluster-name` ist unerheb-

lich, solange Sie nicht mehr als einen Cluster gleichzeitig ausführen wollen. In diesem Fall müssen Sie jedem einen eigenen Namen geben. Die Eigenschaft `whirr.instance-templates` enthält eine kommaseparierte Liste, die beschreibt, welche Rolle jeder Knoten spielt und wie viele Knoten es geben soll. In unserem Fall wollen wir einen Master und fünf Regions-Server. Zum Schluss erzwingt `whirr.hbase.tarball.url`, dass Whirr die gleiche HBase-Version verwendet wie wir.

Das Cluster starten

Nachdem wir alle Details der Konfiguration in `hbase.properties` gespeichert haben, ist es an der Zeit, das Cluster zu starten. In der Kommandozeile führen wir im Whirr-Verzeichnis den `launch-cluster`-Befehl aus und übergeben ihm die gerade angelegte Properties-Datei.

```
$ bin/whirr launch-cluster --config hbase.properties
```

Das erzeugt eine sehr lange Ausgabe und kann eine Weile dauern. Sie können den Fortschritt des Starts überwachen, indem Sie in die AWS EC2-Console wechseln. Sie sollten so etwas sehen wie in Abbildung 18, *Amazon EC2-Console mit startenden HBase-Instanzen*.

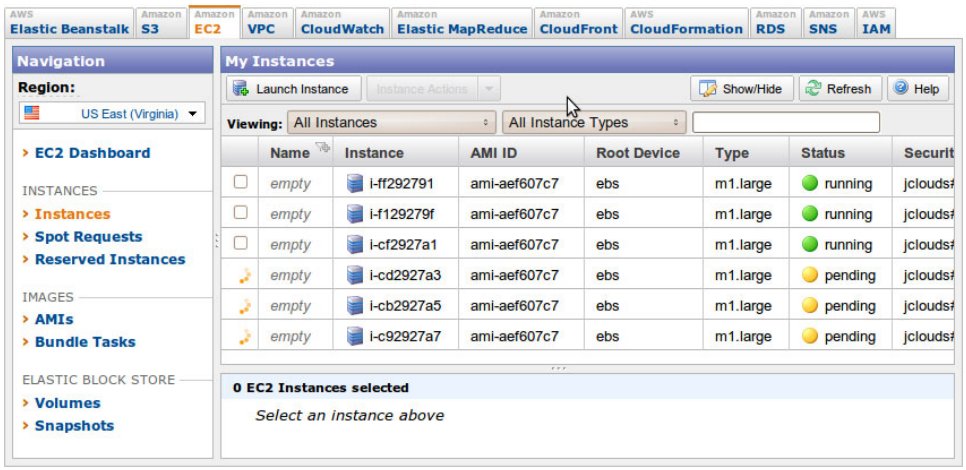


Abbildung 18: Amazon EC2-Console mit startenden HBase-Instanzen

Weitere Informationen zum aktuellen Status finden Sie in der Datei `whirr.log` im Whirr-Verzeichnis.

Verbindung zum Cluster

Standardmäßig ist nur ein gesicherter Datenverkehr mit dem Cluster erlaubt, d. h., um die Verbindung mit HBase herzustellen, müssen wir eine SSH-Session öffnen. Zuerst müssen wir den Namen eines Servers im Cluster kennen, um die Verbindung mit ihm herstellen zu können. Im Home-Verzeichnis des Benutzers hat Whirr ein Verzeichnis namens `.whirr/myhbasecluster` angelegt. Hier finden Sie eine Datei namens `instances`, die alle laufenden Amazon-Instanzen des Clusters (durch Tabulatoren getrennt) aufführt. Die dritte Spalte enthält die öffentlich adressierbaren Domainnamen der Server. Nehmen Sie sich gleich den ersten und führen Sie den folgenden Befehl aus:

```
$ ssh -i keys/id_rsa ${USER}@<SERVER_NAME>
```

Sobald die Verbindung hergestellt wurde, starten Sie die HBase-Shell:

```
$ /usr/local/hbase-0.90.3/bin/hbase shell
```

Sobald die Shell gestartet ist, können Sie den Zustand des Clusters mit dem `status`-Befehl abfragen.

```
hbase> status
6 servers, 0 dead, 2.0000 average load
```

Von hier aus können Sie die gleichen Operationen durchführen, die wir am ersten und zweiten Tag durchgeführt haben, also z. B. Tabellen anlegen und Daten einfügen. Die Anbindung unserer beispielhaften Thrift-basierten Client-Anwendung an den Cluster überlassen wir Ihnen als Übung im Selbststudium.

Bevor wir den Tag beenden, müssen wir aber noch über eine Sache sprechen: Wie man den Cluster herunterfährt.

Das Cluster herunterfahren

Wenn wir mit unserem entfernten HBase EC2-Cluster fertig sind, verwenden wir Whirrs `destroy-cluster`-Befehl, um es herunterzufahren. Beachten Sie, dass dabei alle in den Cluster eingefügten Daten verloren gehen, da wir die Instanzen nicht für die persistente Speicherung konfiguriert haben.

Über die Kommandozeile geben Sie im Whirr-Verzeichnis den folgenden Befehl ein:

```
$ bin/whirr destroy-cluster --config hbase.properties
Destroying myhbasecluster cluster
Cluster myhbasecluster destroyed
```

Das kann eine Weile dauern. Vergewissern Sie sich in der AWS-Console, dass die Instanzen heruntergefahren wurden (was so aussehen sollte wie in Abbildung 19, *Amazon EC2-Console mit herunterfahrenden HBase-Instanzen*).

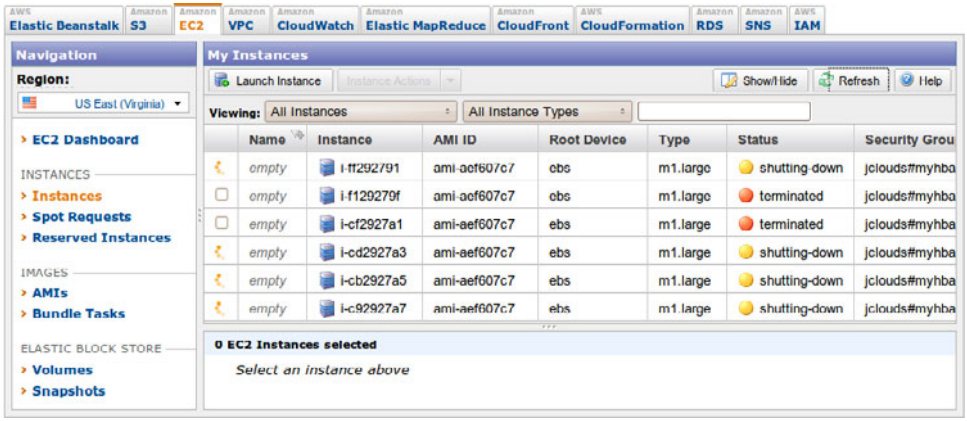


Abbildung 19: Amazon EC2-Console mit herunterfahrenden HBase-Instanzen

Falls beim Herunterfahren etwas schiefgeht, können Sie die Instanzen immer noch direkt in der AWS-Console beenden.

Was wir am dritten Tag gelernt haben

Heute haben wir die HBase-Shell verlassen, um uns andere Möglichkeiten der Anbindung anzusehen, einschließlich eines Binärprotokolls namens Thrift. Wir haben eine Thrift-Client-Anwendung entwickelt und dann ein entferntes Cluster in Amazons EC2 mit Apache Whirr angelegt und verwaltet. Im gleich folgenden Selbststudium kommen diese beiden Dinge zusammen, d. h., Sie fragen Ihr entferntes EC2-Cluster über Ihre lokal laufende Thrift-Anwendung ab.

Tag 3: Selbststudium

Beim heutigen Selbststudium werden Sie Ihre lokale Thrift-Anwendung mit dem entfernt laufenden HBase-Cluster verbinden. Um das tun zu können, müssen Sie Ihr Cluster für unsichere eingehende TCP-Verbindungen öffnen. Wenn dies eine Produktionsumgebung wäre, würden Sie im ersten Schritt

besser einen sicheren Kanal für Thrift schaffen – zum Beispiel durch die Einrichtung eines virtuellen privaten Netzwerks (VPN) mit Endpunkten in EC2 und unserem Netzwerk. Ein solches Setup liegt außerhalb des in diesem Buch behandelten Rahmens, doch Sie sollten Ihren Datenverkehr unbedingt absichern, wenn Sie an diesem Punkt angelangt sind.

Machen Sie Folgendes

1. Bei laufendem EC2-Cluster öffnen Sie eine SSH-Session mit einem Knoten, starten die `hbase`-Shell und legen eine Tabelle mit mindestens einer Spaltenfamilie an.
2. In der gleichen SSH-Session starten Sie den Thrift-Service.

```
$ sudo /usr/local/hbase-0.90.3/bin/hbase-daemon.sh start thrift -b 0.0.0.0
```

3. Verwenden Sie Amazons EC2 Web-Interface-Console, um TCP Port 9090 in der Sicherheitsgruppe Ihres Clusters zur öffnen (Network & Security > Security Groups > Inbound > Create a new rule).
4. Passen Sie die von Ihnen entwickelte einfache Thrift-basierte Ruby-Client-Anwendung so an, dass sie den Thrift-fähigen EC2-Knoten anstelle von localhost verwendet. Führen Sie das Programm aus und überprüfen Sie, ob es die richtigen Informationen über die neu angelegte Tabelle zurückliefert.

4.5 Zusammenfassung

HBase ist eine Mischung aus Einfachheit und Komplexität. Das Datenspeicher-Modell ist recht einfach und es gibt ein paar Beschränkungen hinsichtlich des Schemas. Es hilft auch nicht, dass viele Begriffe an die relationale Welt erinnern (zum Beispiel Wörter wie *Tabelle* und *Spalte*). Das HBase-Schema-Design entscheidet in großen Teilen über die Performance-Charakteristika Ihrer Tabellen und Spalten.

HBases Stärken

Zu den bemerkenswerten Features von HBase gehören die robuste, skalierbare Architektur und die integrierte Versionierung und Komprimierung. Die in HBase fest eingebaute Fähigkeit zur Versionierung ist für manche Anwendungsfälle ein unwiderstehliches Feature. Das Vorhalten einer Versions-Historie von Wiki-Seiten ist beispielsweise ein wichtiges Feature für die Überwachung und Pflege. Durch die Wahl von HBase müssen wir keine zusätzlichen Schritte unternehmen, um eine Seitenhistorie zu implementieren – wir bekommen sie umsonst.

Was die Performance angeht, ist HBase skalierbar. Wenn Sie mit großen Datenmengen von mehreren Giga- oder Terabyte arbeiten, könnte HBase etwas für Sie sein. HBase ist Rack-fähig, d. h., es kann Daten schnell in und zwischen 19"-Schränken replizieren, so dass fehlerhafte Knoten schnell und elegant gehandhabt werden können.

Die HBase-Community ist wirklich phantastisch. Es steht nahezu immer jemand im IRC-Kanal[10] oder in den Mailinglisten[11]bereit, der Fragen beantwortet und einem die richtige Richtung weist. Obwohl eine Reihe großer Unternehmen HBase für ihre Projekte nutzen, gibt es keine Firma, die HBase-Services anbietet. Das bedeutet, dass die Leute in der HBase-Community das dem Projekt und der Allgemeinheit zuliebe tun.

HBases Schwächen

Zwar lässt sich HBase vergrößern, aber nicht verkleinern. Die HBase-Community scheint sich einig zu sein, dass fünf Knoten die minimale Größe darstellen. Da es für große Dinge entworfen wurde, ist auch die Administration schwieriger. Bei HBase geht es nicht darum, kleine Probleme zu lösen, und Dokumentation für Nicht-Experten ist nur schwer zu finden, was die Sache noch schwieriger macht.

Darüber hinaus wird HBase so gut wie nie allein eingesetzt. Es ist vielmehr Teil eines Ökosystems skalierbarer Einzelteile, das Hadoop (eine Implementierung von Googles MapReduce), das Hadoop Distributed File System (HDFS) und Zookeeper (ein Service zur Konfiguration verteilter Systeme) umfasst. Dieses Ökosystem hat sowohl Stärken als auch Schwächen. Es bietet eine große architektonische Stabilität, bürdet dem Administrator aber auch die entsprechende Pflege auf.

Eine bemerkenswerte Charakteristik von HBase ist (abgesehen von Zeilenschlüsseln) das Fehlen jeglicher Sortier- und Index-Möglichkeiten. Zeilen werden über ihre Zeilenschlüssel sortiert vorgehalten, aber andere Felder, etwa Spaltennamen und -werte, werden nicht sortiert. Wenn Sie Zeilen also über etwas anderes als den Schlüssel ausfindig machen wollen, müssen Sie die Tabelle scannen oder einen eigenen Index pflegen.

Ein weiteres fehlendes Konzept sind Datentypen. Alle Feldwerte werden bei HBase als uninterpretierte Arrays von Bytes behandelt. Es gibt keine Unterscheidung von Integerwert, String oder Datum. Für HBase ist alles Bytes, d. h., Ihre Anwendung muss diese Bytes selbst interpretieren.

10. irc://irc.freenode.net/#hbase
11. http://hbase.apache.org/mail-lists.html

HBase und CAP

In Bezug auf CAP ist HBase eindeutig CP. HBase macht starke Konsistenz-Garantien. Wenn ein Client einen Wert erfolgreich schreiben kann, empfängt ein anderer Client den aktualisierten Wert beim nächsten Request. Einige Datenbanken wie Riak erlauben die Optimierung der CAP-Gleichung auf Basis einzelner Operationen. Nicht so HBase. Bei ausreichender Partitionierung bleibt HBase verfügbar, wenn ein Knoten ausfällt, und verteilt die Verantwortung auf die anderen Knoten im Cluster. Bleibt (hypothetisch) nur noch ein Knoten übrig, hat HBase keine andere Wahl, als Requests abzulehnen.

Die CAP-Diskussion wird etwas komplizierter, wenn die Cluster-zu-Cluster-Replikation ins Spiel kommt, ein fortgeschrittenes Feature, das wir in diesem Kapitel nicht behandelt haben. Bei einem typischen Multicluster-Setup können Cluster geographisch getrennt sein. In diesem Fall wird ein Cluster für eine Spaltenfamilie das System sein, in dem die Daten eingetragen werden, während die anderen Cluster nur den Zugriff auf die replizierten Daten ermöglichen. Dieses System ist *schlussendlich konsistent*, weil die Replikations-Cluster die aktuellsten Werte zurückliefern, die sie kennen, was aber nicht mit den neuesten Werten auf dem Master-Cluster übereinstimmen muss.

Abschließende Gedanken

Als eine der ersten nichtrelationalen Datenbanken, die wir kennengelernt haben, war HBase eine echte Herausforderung für uns. Die Terminologie ist einem trügerisch vertraut und die Installation und Konfiguration ist nichts für schwache Nerven. Auf der Haben-Seite sind einige HBase-Features wie Versionierung und Komprimierung einzigartig. Diese Aspekte können HBase für die Lösung bestimmter Probleme sehr attraktiv machen. Und natürlich skaliert es über viele Knoten handelsüblicher Hardware sehr gut. Alles in allem ist HBase – wie eine Nagelpistole – ein sehr mächtiges Werkzeug, also achten Sie auf Ihre Finger.

KAPITEL 5

MongoDB

MongoDB ist in vielerlei Hinsicht wie eine Bohrmaschine. Ihre Fähigkeit, eine Aufgabe abzuschließen, hängt größtenteils von den von Ihnen eingespannten Teilen (von den Bohrern bis zum Schleifaufsatz) ab. MongoDBs Stärken sind Flexibilität, Leistungsfähigkeit, einfache Nutzung und die Fähigkeit, große und kleine Jobs erledigen zu können. Obwohl es sich um eine jüngere Erfindung als den Hammer handelt, wird es immer häufiger eingesetzt.

MongoDB wurde 2009 erstmals vorgestellt und ist ein aufsteigender Stern am NoSQL-Himmel. Es wurde als skalierbare Datenbank entworfen – der Name Mongo leitet sich aus dem englischen „hu*mongo*us“, also „riesig“ ab – , mit Performance und einfachem Datenzugriff als Kernziele. Es handelt sich um eine Dokumenten-Datenbank, bei der Daten verschachtelt festgehalten und, noch wichtiger, ab hoc abgefragt werden können. Ein Schema ist (wie bei Riak aber im Gegensatz zu Postgres) nicht notwendig, d. h., Dokumente können optional Felder oder Typen enthalten, die kein anderes Dokument besitzt.

Sie sollten aber nicht glauben, dass MongoDBs Flexibilität es zu einem Spielzeug macht. Es gibt einige große MongoDB-Installationen (häufig einfach *Mongo* genannt) da draußen, etwa bei Foursquare, bit.ly und am CERN (zur Sammlung der Large Hadron Collider-Daten).

5.1 Hu(mongo)us

Mongo bewegt sich zwischen der leistungsfähigen Abfragbarkeit relationaler Datenbanken und der verteilten Natur anderer Datenspeicher wie Riak oder HBase. Der Projektgründer Dwight Merriman bezeichnet MongoDB als die Datenbank, die er bei DoubleClick gerne gehabt hätte, wo er als CTO große Datenmengen speichern und gleichzeitig Ad-hoc-Queries verarbeiten musste.

Mongo ist eine JSON-Dokumenten-Datenbank (auch wenn die Daten technisch in einem als BSON bekannten JSON-Binärformat gespeichert werden). Sie können ein Mongo-Dokument mit einer Zeile einer relationalen Tabelle vergleichen (aber ohne das dazugehörige Schema), deren Werte beliebig tief verschachtelt werden können. In Abbildung 20, *Mongo-Dokument im JSON-Format* erhalten Sie eine Vorstellung davon, wie ein JSON-Dokument aussieht.

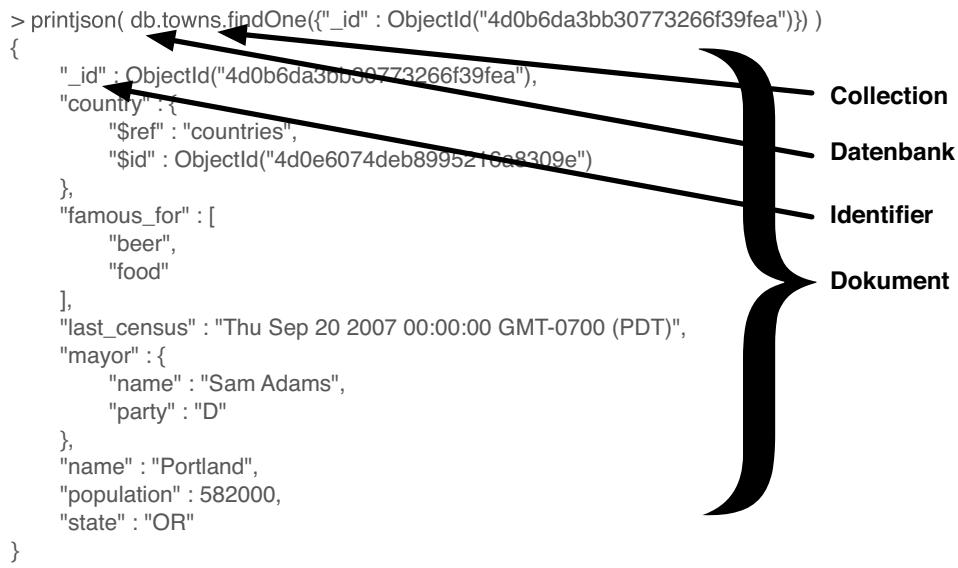


Abbildung 20: Mongo-Dokument im JSON-Format

Mongo ist eine ausgezeichnete Wahl für die ständig wachsende Klasse von Web-Projekten, bei denen große Datenmengen gespeichert werden müssen, das Budget aber keine „Big-Iron"-Hardware zulässt. Da es kein strukturiertes Schema gibt, kann Mongo, zusammen mit dem Datenmodell, wachsen und sich anpassen. Als Web-Startup mit großen Träumen, aber auch wenn Sie bereits groß sind und die Server horizontal skalieren müssen, sollten Sie MongoDB in Erwägung ziehen.

5.2 Tag 1: CRUD und Schachtelung

Heute wollen wir einige CRUD-Operationen durchführen und den Tag mit verschachtelten MongoDB-Queries abschließen. Wie üblich gehen wir die In-

Eric sagt:
Einerseits, andererseits

Ich musste mich für einen Dokumentenspeicher entscheiden, bevor ich den Wechsel in meinem Produktionscode vornehmen konnte. Da ich aus der Welt relationaler Datenbanken komme, war Mongo mit seinen Ad-hoc-Queries für mich ein problemloser Wechsel. Und seine Skalierbarkeit spiegelte meine Träume für die Web-Skalierbarkeit wider. Doch neben der Struktur vertraute ich auch dem Entwicklungsteam. Sie hatten bereits eingestanden, dass Mongo nicht perfekt war, doch ihre klaren Pläne (und das Festhalte an diesen Plänen) basierten auf realen Anwendungsfällen für Web-Infrastrukturen und nicht auf Debatten über Skalierbarkeit und Replikation. Diese pragmatische Konzentration auf Nutzbarkeit sollte durchscheinen, wenn Sie MongoDB nutzen. Ein Nachteil dieses evolutionären Verhaltens ist, dass es verschiedene Möglichkeiten gibt, eine bestimmte Funktion in Mongo durchzuführen.

stallation nicht durch, doch wenn Sie die Mongo-Website besuchen,[1] können Sie einen Build für Ihr Betriebssystem herunterladen oder Anweisungen finden, wie man den Code aus den Quellen selbst kompiliert. Wenn Sie mit OS X arbeiten, empfehlen wir die Installation über Homebrew (`brew install mongodb`). Wenn Sie eine Debian/Ubuntu-Variante nutzen, versuchen Sie es mit Mongodb.orgs eigenem apt-get-Paket.

Um Schreibfehler zu vermeiden, müssen Sie für Mongo zuerst das Verzeichnis anlegen, in dem `mongod` die Daten speichert. Ein typischer Ort ist `/data/db`. Stellen Sie sicher, dass der Benutzer, unter dem der Server läuft, Lese- und Schreibrechte für dieses Verzeichnis hat. Falls er noch nicht läuft, starten Sie den Mongo-Service, indem Sie `mongod` ausführen.

Spaß in der Kommandozeile

Um eine neue Datenbank namens `book` anzulegen, führen Sie zuerst den folgenden Befehl in Ihrem Terminal aus. Er stellt die Verbindung mit dem MySQL-inspirierten Kommandozeilen-Interface her.

```
$ mongo book
```

Die Eingabe von `help` in der Console ist ein guter Anfang. Wir verwenden momentan die `book`-Datenbank, können uns die anderen aber mit `show dbs` zeigen lassen und die Datenbank mit dem `use`-Befehl wechseln.

Um bei Mongo eine Collection (vergleichbar dem *Bucket* der Riak-Nomenklatur) anzulegen, müssen Sie einfach nur einen ersten Datensatz in die Collection einfügen. Da Mongo schemafrei ist, müssen Sie im Vorfeld nichts

1. http://www.mongodb.org/downloads

einrichten. Sie zu verwenden, reicht aus. Tatsächlich existiert unsere book-Datenbank nicht, bis wir erstmalig Werte hinzufügen. Der folgende Code erzeugt eine towns-Collection und fügt etwas hinzu:

```
> db.towns.insert({
  name: "New York",
  population: 22200000,
  last_census: ISODate("2009-07-31"),
  famous_for: [ "statue of liberty", "food" ],
  mayor : {
    name : "Michael Bloomberg",
    party : "I"
  }
})
```

Im obigen Abschnitt haben wir gesagt, dass Dokumente in JSON (AOK, eigentlich BSON) gespeichert werden, d. h., wir fügen neue Dokumente im JSON-Format ein, wobei geschweifte Klammern {…} für ein Objekt (eine Hashtabelle oder eine Map) mit dazugehörigen Werten stehen und eckige Klammern […] ein Array darstellen. Sie können diese Werte beliebig tief verschachteln.

Mit dem Befehl `show collections` können Sie überprüfen, ob die Collection jetzt existiert.

```
> show collections

system.indexes
towns
```

`towns` haben wir gerade erzeugt, während `system.indexes` immer vorhanden ist. Wir können uns den Inhalt einer Collection über `find` ausgeben lassen. Wir haben hier die Ausgabe der Lesbarkeit halber formatiert, bei Ihnen wird einfach nur eine umgebrochene Zeile ausgegeben.

```
> db.towns.find()

{
  "_id" : ObjectId("4d0ad975bb30773266f39fe3"),
  "name" : "New York",
  "population": 22200000,
  "last_census": "Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)",
  "famous_for" : [ "statue of liberty", "food" ],
  "mayor" : { "name" : "Michael Bloomberg", "party" : "I" }
}
```

Im Gegensatz zu relationalen Datenbanken unterstützt Mongo keine serverseitigen Joins. Ein einzelner JavaScript-Aufruf liefert ein Dokument *und* den gesamten verschachtelten Inhalt zurück.

Vielleicht ist Ihnen aufgefallen, dass die JSON-Ausgabe der von uns neu eingefügten Stadt ein `_id`-Feld namens `ObjectId` enthält. Das ähnelt dem `SERIAL`-Inkrement eines numerischen Primärschlüssels bei PostgreSQL. Die `ObjectId` ist immer 12 Bytes lang und besteht aus einem Zeitstempel, der ID des Client-Rechners, der ID des Client-Prozesses und einem drei Byte großen inkrementierten Zähler. Das Byte-Layout ist in Abbildung 21, *Beispiel für ein ObjectId-Layout* zu sehen.

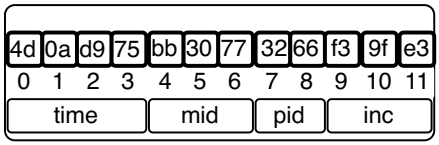


Abbildung 21: Beispiel für ein ObjectId-Layout

Das Gute an diesem Autonummierungsschema ist, dass jeder Prozess auf jedem Rechner seine eigene ID-Generierung handhaben kann, ohne mit anderen `mongod`-Instanzen zu kollidieren. Diese Design-Entscheidung gibt einen Hinweis auf Mongos verteilte Natur.

JavaScript

Mongos Muttersprache ist JavaScript, egal ob komplexe Dinge wie Mapreduce-Queries oder etwas so Einfaches wie um Hilfe zu fragen.

```
> db.help()
> db.towns.help()
```

Diese Befehle führen die Funktionen auf, die es für das gegebene Objekt gibt. `db` ist ein JavaScript-Objekt mit Informationen über die aktuelle Datenbank. `db.x` ist ein JavaScript-Objekt, das eine Collection (namens `x`) repräsentiert. Befehle sind einfach JavaScript-Funktionen.

```
> typeof db
object
> typeof db.towns
object
> typeof db.towns.insert
function
```

Wenn Sie den Quellcode einer Funktion untersuchen wollen, rufen Sie sie einfach ohne Parameter oder Klammern auf (denken Sie mehr an Python als an Ruby).

```
db.towns.insert
function (obj, _allow_dot) {
    if (!obj) {
        throw "no object passed to insert!";
    }
    if (!_allow_dot) {
        this._validateForStorage(obj);
    }
    if (typeof obj._id == "undefined") {
        var tmp = obj;
        obj = {_id:new ObjectId};
        for (var key in tmp) {
            obj[key] = tmp[key];
        }
    }
    this._mongo.insert(this._fullName, obj);
    this._lastID = obj._id;
}
```

Wir wollen weitere Dokumente in unsere `towns`-Collection einfügen und entwickeln dafür unsere eigene JavaScript-Funktion.

mongo/insert_city.js

```
function insertCity(
  name, population, last_census,
  famous_for, mayor_info
) {
  db.towns.insert({
    name:name,
    population:population,
    last_census: ISODate(last_census),
    famous_for:famous_for,
    mayor : mayor_info
  });
}
```

Sie können den Code einfach in die Shell einfügen. Dann rufen Sie sie wie folgt auf.

```
insertCity("Punxsutawney", 6200, '2008-31-01',
  ["phil the groundhog"], { name : "Jim Wehrle" }
)

insertCity("Portland", 582000, '2007-20-09',
  ["beer", "food"], { name : "Sam Adams", party : "D" }
)
```

Wir haben nun drei Städte in unserer Collection, was wir wie vorhin mit `db.towns.find()` überprüfen können.

Lesen: Mehr Spaß mit Mongo

Vorhin haben wir `find` ohne Parameter aufgerufen, um alle Dokumente abzurufen. Um ein bestimmtes Dokument abzurufen, müssen Sie nur eine `_id`-Property setzen. `_id` ist vom Typ `ObjectId`, weshalb Sie für die Query einen String über die Funktion `ObjectId(str)` entsprechend umwandeln müssen.

```
db.towns.find({ "_id" : ObjectId("4d0ada1fbb30773266f39fe4") })

{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney",
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ],
  "mayor" : { "name" : "Jim Wehrle" }
}
```

Die `find()`-Funktion akzeptiert auch einen optionalen zweiten Parameter: ein fields-Objekt, mit dessen Hilfe sich die empfangenen Felder filtern lassen. Wenn Sie nur den Namen der Stadt brauchen (zusammen mit der `_id`), übergeben Sie `name` mit dem Wert `1` (oder `true`).

```
db.towns.find({ _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 1 })

{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney"
}
```

Um alle Felder *außer* `name` abzurufen, setzen Sie `name` auf `0` (oder `false` oder `null`).

```
db.towns.find({ _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 0 })

{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ]
}
```

Wie bei PostgreSQL können Sie mit Mongo Ad-hoc-Queries über Feldwerte, Bereiche oder eine Kombination verschiedener Kriterien erzeugen. Um alle Städte aufzuspüren, die mit dem Buchstaben *P* beginnen und weniger als 10000 Einwohner haben, können Sie einen Perl-kompatiblen regulären Ausdruck (PCRE)[2] und einen Bereichsoperator verwenden.

2. http://www.pcre.org/

```
db.towns.find(
  { name : /^P/, population : { $lt : 10000 } },
  { name : 1, population : 1 }
)
{ "name" : "Punxsutawney", "population" : 6200 }
```

Bedingungsoperatoren haben bei Mongo das Format `feld : { $op : wert }`, wobei `$op` eine Operation wie `$ne` (nicht gleich) ist. Sie könnten sich eine etwas kompaktere Syntax wie `feld < wert` wünschen, doch hier handelt es sich um JavaScript-Code, nicht um eine domänenspezifische Query-Sprache, d. h., die Queries müssen den JavaScript-Syntaxregeln entsprechen. (Später zeigen wir, wie man in bestimmten Fällen eine kürzere Syntax nutzen kann, aber im Moment überspringen wir das.)

Die gute Nachricht in Bezug auf JavaScript als Query-Sprache lautet, dass Sie die Operationen so aufbauen können, als wären sie Objekte. Im folgenden Beispiel bauen wir ein Kriterium auf, bei dem die Einwohnerzahl zwischen 10000 und einer Million liegen soll.

```
var population_range = {}
population_range['$lt'] = 1000000
population_range['$gt'] = 10000
db.towns.find(
  { name : /^P/, population : population_range },
  { name: 1 }
)

{ "_id" : ObjectId("4d0ada87bb30773266f39fe5"), "name" : "Portland" }
```

Wir sind nicht auf Zahlenbereiche beschränkt, sondern können auch Datumsbereiche abrufen. So können wir beispielsweise alle Namen mit einem *last_census* kleiner oder gleich dem 31.1.2008 wie folgt finden:

```
db.towns.find(
  { last_census : { $lte : ISODate('2008-31-01') } },
  { _id : 0, name: 1 }
)

{ "name" : "Punxsutawney" }
{ "name" : "Portland" }
```

Beachten Sie, wie wir das `_id`-Feld in der Ausgabe unterdrücken, indem wir es explizit auf `0` setzen.

Tiefer graben

Mongo liebt verschachtelte Array-Daten. Queries können über exakte Treffer

```
db.towns.find(
  { famous_for : 'food' },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

aber auch über partielle Treffer

```
db.towns.find(
  { famous_for : /statue/ },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
```

über alle passenden Werte

```
db.towns.find(
  { famous_for : { $all : ['food', 'beer'] } },
  { _id : 0, name:1, famous_for:1 }
)

{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

oder über das Fehlen passender Werte erfolgen:

```
db.towns.find(
  { famous_for : { $nin : ['food', 'beer'] } },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "Punxsutawney", "famous_for" : [ "phil the groundhog" ] }
```

Doch seine eigentliche Stärke leitet Mongo aus seiner Fähigkeit ab, tief in ein Dokument eintauchen und Ergebnisse tief verschachtelter Subdokumente zurückliefern zu können. Um ein Subdokument abzufragen, geben Sie den Feldnamen als String ein und trennen die verschachtelten Ebenen durch Punkte voneinander. Zum Beispiel können Sie die Städte mit unabhängigen Bürgermeistern ermitteln

```
db.towns.find(
  { 'mayor.party' : 'I' },
  { _id : 0, name : 1, mayor : 1 }
)
```

```
{
  "name" : "New York",
  "mayor" : {
    "name" : "Michael Bloomberg",
    "party" : "I"
  }
}
```

oder diejenigen mit parteilosen Bürgermeistern:

```
db.towns.find(
  { 'mayor.party' : { $exists : false } },
  { _id : 0, name : 1, mayor : 1 }
)

{ "name" : "Punxsutawney", "mayor" : { "name" : "Jim Wehrle" } }
```

Die obigen Queries sind großartig, wenn man Dokumente mit einem einzigen übereinstimmenden Feld aufspüren möchte, doch was tun, wenn mehrere Felder eines Subdokuments abgefragt werden müssen?

elemMatch

Wir erweitern unsere Suche um die `$elemMatch`-Direktive. Lassen Sie uns eine weitere Collection anlegen, in der wir Länder speichern. Diesmal überschreiben wir jede `_id` mit einem von uns gewählten String.

```
db.countries.insert({
  _id : "us",
  name : "United States",
  exports : {
    foods : [
      { name : "bacon", tasty : true },
      { name : "burgers" }
    ]
  }
})
db.countries.insert({
  _id : "ca",
  name : "Canada",
  exports : {
    foods : [
      { name : "bacon", tasty : false },
      { name : "syrup", tasty : true }
    ]
  }
})
```

```
db.countries.insert({
  _id : "mx",
  name : "Mexico",
  exports : {
    foods : [{
      name : "salsa",
      tasty : true,
      condiment : true
    }]
  }
})
```

Um zu prüfen, ob die Länder eingefügt wurden, führen wir die count-Funktion aus (und erwarten den Wert 3).

```
> print( db.countries.count() )
3
```

Nun wollen wir ein Land finden, das nicht nur Bacon exportiert, sondern leckeren (*tasty*) Bacon.

```
db.countries.find(
  { 'exports.foods.name' : 'bacon',  'exports.foods.tasty' : true },
  { _id : 0, name : 1 }
)

{ "name" : "United States" }
{ "name" : "Canada" }
```

Doch das ist nicht das Ergebnis, das wir suchen. Mongo hat *Canada* zurückgegeben, weil es Bacon und leckeren Sirup exportiert. Hier hilft uns `$elemMatch`. Es legt fest, dass ein Dokument (oder verschachteltes Dokument) als Treffer zählt, wenn *all* unsere Kriterien erfüllt sind.

```
db.countries.find(
  {
    'exports.foods' : {
      $elemMatch : {
        name : 'bacon',
        tasty : true
      }
    }
  },
  { _id : 0, name : 1 }
)

{ "name" : "United States" }
```

`$elemMatch`-Kriterien können auch fortgeschrittene Operatoren nutzen. Sie können jedes Land bestimmen, das leckeres Essen exportiert und bei dem auch die Gewürz-Kennung (condiment) gesetzt ist:

```
db.countries.find(
  {
    'exports.foods' : {
      $elemMatch : {
        tasty : true,
        condiment : { $exists : true }
      }
    }
  },
  { _id : 0, name : 1 }
)

{ "name" : "Mexico" }
```

Mexico ist genau das, was wir gesucht haben.

Boolesche Operatoren

Bisher waren unsere Kriterien implizite *UND*-Operationen. Wenn Sie versuchen, ein Land namens *United States* mit der `_id` *mx* zu finden, gibt Mongo keinen Treffer zurück.

```
db.countries.find(
  { _id : "mx", name : "United States" },
  { _id : 1 }
)
```

Wenn Sie hingegen mit `$or` nach dem einen *ODER* dem anderen suchen, werden zwei Ergebnisse zurückgeliefert. Stellen Sie sich das als *Präfixnotation* vor: `OR A B`.

```
db.countries.find(
  {
    $or : [
      { _id : "mx" },
      { name : "United States" }
    ]
  },
  { _id:1 }
)

{ "_id" : "us" }
{ "_id" : "mx" }
```

Es gibt so viele Operatoren, dass wir sie hier nicht alle behandeln können, doch wir hoffen, Ihnen eine Vorstellung von MongoDBs mächtigen Query-

Fähigkeiten gegeben zu haben. Nachfolgend finden Sie eine nicht ganz vollständige (aber doch recht umfassende) Liste der Befehle.

Befehl	Beschreibung
$regex	Matching eines PCRE-konformen Regulären Ausdrucks (oder verwenden Sie einfach die //-Trennzeichen, wie vorhin gezeigt)
$ne	Nicht gleich
$lt	Kleiner als
$lte	Kleiner oder gleich
$gt	Größer als
$gte	Größer oder gleich
$exists	Prüft, ob ein Feld existiert
$all	Matching aller Elemente im Array
$in	Matching beliebiger Elemente im Array
$nin	Kein Matching beliebiger Element im Array
$elemMatch	Matching aller Felder in einem Array verschachtelter Dokumente
$or	ODER
$nor	NICHT ODER
$size	Matching eines Arrays gegebener Größe
$mod	Modulo
$type	Matching eines Feldes auf angegebenen Datentyp
$not	Negation des angegebenen Operators

Sie finden alle Befehle in der MongoDB-Online-Dokumentation oder Sie können sich eine Kurzreferenz von der Mongo-Website herunterladen. Wir werden uns in den nächsten Tagen noch weiter mit Queries beschäftigen.

Updates

Wir haben ein Problem. New York und Punxsutawney sind eindeutig genug, aber haben wir jetzt das Portland in Oregon oder in Maine (oder Texas oder, oder, oder) eingefügt? Wir wollen unsere `towns`-Collection aktualisieren und um US-Bundesstaaten erweitern.

Die Funktion `update(kriterium,operation)` verlangt zwei Parameter. Der erste ist die Kriteriums-Query – das gleiche Objekt, das Sie auch an `find()` übergeben. Der zweite Parameter ist entweder das Objekt, dessen Felder das/die passende(n) Dokument(e) ersetzt, oder eine Modifikator-Operation. In unserem Fall `$setzt` der Modifikator das Feld `state` auf den String *OR*(egon).

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $set : { "state" : "OR" } }
);
```

Sie fragen sich wahrscheinlich, warum eine `$set`-Operation überhaupt notwendig ist. Mongo denkt nicht in Attributen. Es hat nur aus Optimierungsgründen eine interne, implizite Vorstellung von Attributen. Doch am Interface selbst ist nichts *Attribut*-artiges. Mongo ist *dokumentenorientiert*. Sie werden nur selten Folgendes machen wollen (beachten Sie das Fehlen der `$set`-Operation):

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { state : "OR" }
);
```

Damit würden Sie das *gesamte* Dokument durch das übergebene Dokument ersetzen (`{ state : "OR" }`). Da Sie keinen Befehl wie `$set` angegeben haben, geht Mongo davon aus, dass Sie es einfach ersetzen wollen, also Vorsicht.

Wir können den Erfolg des Updates überprüfen, indem wir das Dokument einfach abrufen (beachten Sie, dass wir `findOne` verwenden, um nur einen Treffer abzurufen).

```
db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })
```

```
{
  "_id" : ObjectId("4d0ada87bb30773266f39fe5"),
  "famous_for" : [
    "beer",
    "food"
  ],
  "last_census" : "Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)",
  "mayor" : {
    "name" : "Sam Adams",
    "party" : "D"
  },
  "name" : "Portland",
  "population" : 582000,
  "state" : "OR"
}
```

Sie können mehr machen, als nur einen Wert zu `$set`zen. `$inc` (eine Zahl inkrementieren) ist auch recht nützlich. Lassen Sie uns Portlands Einwohnerzahl um 1000 erhöhen.

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $inc : { population : 1000} }
)
```

Es gibt viele Direktiven wie diese, etwa den Positionsoperator `$` für Arrays. Neue Operationen werden häufig hinzugefügt und in der Online-Dokumentation ergänzt. Hier die wichtigsten Direktiven:

Befehl	Beschreibung
$set	Setzt das angegebene Feld auf den angegebenen Wert
$unset	Entfernt das Feld
$inc	Addiert den angegebenen Wert zum angegebenen Feld hinzu
$pop	Entfernt das letzte (oder erste) Element aus einem Array
$push	Fügt einen Wert in ein Array ein
$pushAll	Fügt alle Werte in ein Array ein
$addToSet	Wie push, aber ohne doppelte Werte
$pull	Entfernt passende Werte aus einem Array
$pullAll	Entfernt alle passenden Wert aus einem Array

Referenzen

Wie bereits erwähnt, wurde Mongo nicht für Joins entwickelt. Aufgrund seiner verteilten Natur sind Joins recht ineffiziente Operationen. Dennoch ist es für Dokumente manchmal nützlich, sich gegenseitig zu referenzieren. In diesem Fall empfiehlt das Mongo-Entwicklungsteam ein Konstrukt wie `{ $ref : "collection_name", $id : "reference_id" }`. Zum Beispiel können wir unsere `towns`-Collection um eine Referenz auf ein Dokument in `countries` ergänzen.

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $set : { country: { $ref: "countries", $id: "us" } } }
)
```

Nun können Sie Portland aus der `towns`-Collection abrufen.

```
var portland = db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })
```

Um dann das Land abzurufen, können Sie die `countries`-Collection mit der gespeicherten `$id`abfragen.

```
db.countries.findOne({ _id: portland.country.$id })
```

Und mit JavaScript können Sie im town-Dokument nach dem Namen der Collection fragen, die in der Referenz gespeichert ist.

```
db[ portland.country.$ref ].findOne({ _id: portland.country.$id })
```

Die beiden letzten Queries sind gleich, die zweite ist nur etwas mehr datengesteuert.

Vorsicht bei der Schreibweise

Mongo ist bei Schreibfehlern nicht besonders nachsichtig. Wenn Sie dieses Problem noch nicht hatten, werden Sie ihm irgendwann begegnen, also seien Sie gewarnt. Man kann Parallelen zwischen statischen und dynamischen Programmiersprachen ziehen. Bei statischen Sprachen definieren Sie vorab, während dynamische Sprachen Werte akzeptieren, die Sie nicht gewollt haben, selbst so unsinnige Dinge wie `person_name = 5`.

Dokumente sind schemafrei, d. h., Mongo kann nicht wissen, ob Sie `pipulation` zu einer Stadt hinzufügen wollten oder ob Sie tatsächlich den `lust_census` abfragen wollen. Es wird diese Felder ungefragt einfügen bzw. keine Ergebnisse zurückliefern.

Flexibilität hat ihren Preis. *Caveat emptor*.

Löschen

Dokumente aus einer Collection zu löschen, ist einfach. Ersetzen Sie die `find()`-Funktion einfach durch einen Aufruf von `remove()`, und alle passenden Kriterien werden entfernt. Es ist wichtig zu beachten, dass das gesamte Dokument entfernt wird und nicht nur ein passendes Element oder Subdokument.

Wir empfehlen, zuerst `find()` auszuführen, um die Kriterien zu verifizieren, bevor Sie `remove` ausführen. Mongo lässt sich nicht zweimal bitten, bevor es die Operation ausführt. Lassen Sie uns alle Länder löschen, die Bacon exportieren, der nicht lecker ist.

```
var bad_bacon = {
  'exports.foods' : {
    $elemMatch : {
      name : 'bacon',
      tasty : false
    }
  }
}
db.countries.find( bad_bacon )
```

```
{
  "_id" : ObjectId("4d0b7b84bb30773266f39fef"),
  "name" : "Canada",
  "exports" : {
    "foods" : [
      {
        "name" : "bacon",
        "tasty" : false
      },
      {
        "name" : "syrup",
        "tasty" : true
      }
    ]
  }
}
```

Das sieht gut aus. Jetzt löschen wir es.

```
db.countries.remove( bad_bacon )
db.countries.count()
```

```
2
```

Führen Sie nun `count()` aus und verifizieren Sie, dass nur noch zwei Länder übrig sind. Ist das der Fall, war das Löschen erfolgreich!

Lesen mit Code

Wir wollen den Tag mit einer interessanteren Query-Option abschließen: Code. Sie können MongoDB veranlassen, eine Entscheidungsfunktion über Ihre Dokumente auszuführen. Wir haben uns das bis zuletzt aufgehoben, weil es immer der letzte Ausweg sein sollte. Diese Queries sind recht langsam, man kann sie nicht indexieren und Mongo kann sie nicht optimieren. Doch manchmal ist die Leistungsfähigkeit des eigenen Codes nur schwer zu schlagen.

Nehmen wir an, wir suchen Einwohnerzahlen zwischen 6000 und 600000.

```
db.towns.find( function() {
  return this.population > 6000 && this.population < 600000;
} )
```

Mongo besitzt sogar eine Kurzform für einfache Entscheidungsfunktionen.

```
db.towns.find("this.population > 6000 && this.population < 600000")
```

Sie können eigenen Code mit anderen Kriterien über die `$where`-Klausel ausführen. Im folgenden Beispiel filtert die Query auch Städte heraus, die für Murmeltiere berühmt sind.

```
db.towns.find( {
  $where : "this.population > 6000 && this.population < 600000",
  famous_for : /groundhog/
} )
```

Doch seien Sie gewarnt: Mongo führt diese Funktion gnadenlos über jedes Dokument aus und es gibt keine Garantie, dass die angegebenen Felder existieren. Wenn Sie zum Beispiel davon ausgehen, dass ein *population*-Feld existiert und dieses Feld auch nur in einem einzigen Dokument fehlt, dann schlägt die Query fehl, weil JavaScript nicht sauber ausgeführt werden kann. Seien Sie vorsichtig, wenn Sie eigene JavaScript-Funktionen entwickeln – und Sie sollten mit JavaScript vertraut sein, bevor Sie sich an eigenem Code versuchen.

Was wir am ersten Tag gelernt haben

Heute haben wir einen Blick auf unsere erste Dokumenten-Datenbank geworfen: MongoDB. Wir haben gesehen, wie man verschachtelte, strukturierte Daten in JSON-Objekten speichern und diese Daten in beliebiger Tiefe abfragen kann. Sie haben gelernt, dass man sich ein *Dokument* als schemafreie Zeile in einem relationalen Modell vorstellen kann, bei der eine erzeugte `_id` den Schlüssel bildet. Eine Gruppe von Dokumenten wird bei Mongo als *Collection* bezeichnet, ähnlich einer *Tabelle* in PostgreSQL.

Im Gegensatz zu den bisher vorgestellten Datenbanktypen speichert Mongo mit seiner Sammlung von Reihen einfacher Datentypen komplexe, denormalisierte Dokumente, die als Collections beliebiger JSON-Strukturen gespeichert und abgerufen werden. Mongo rundet die flexible Speicherstrategie mit einem mächtigen Query-Mechanismus ab, der nicht durch ein vordefiniertes Schema beschränkt wird.

Die denormalisierte Natur macht einen Dokumenten-Datenspeicher zu einer ausgezeichneten Wahl, um Daten mit unbekannten Eigenschaften zu speichern, während andere Arten (wie relational oder spaltenorientiert) die Dinge im Vorfeld wissen wollen und Schema-Migrationen brauchen, um Felder hinzufügen oder editieren zu können.

Tag 1: Selbststudium

Finden Sie heraus

1. Setzen Sie ein Lesezeichen für die MongoDB-Online-Dokumentation.
2. Schlagen Sie nach, wie man reguläre Ausdrücke in Mongo aufbaut.
3. Machen Sie sich mit `db.help` und `db.collections.help` vertraut.
4. Finden Sie einen Mongo-Treiber für die von Ihnen bevorzugte Programmiersprache (Ruby, Java, PHP und so weiter).

Machen Sie Folgendes

1. Geben Sie ein JSON-Dokument aus, das `{ "Hallo" : "Welt" }` enthält.
2. Wählen Sie eine Stadt über einen die Groß-/Kleinschreibung ignorierenden regulären Ausdruck aus, der das Wort *new* enthält.
3. Finden Sie alle Städte, deren Namen ein *e* enthalten und die für Essen und Bier berühmt sind.
4. Legen Sie eine neue Datenbank namens *blogger* an, die eine Collection benannter Artikel (*articles*) enthält. Fügen Sie neue Artikel ein, die den Namen und die E-Mail des Autors, das Erstellungsdatum und den Text enthalten.
5. Ergänzen Sie die Artikel um ein Array von Kommentaren. Jeder Kommentar besteht aus einem Autor und dem Text.
6. Führen Sie eine Query über eine externe JavaScript-Datei durch.

5.3 Tag 2: Indexierung, Gruppierung, Mapreduce

Die Erhöhung der Query-Performance von MongoDB ist der erste Punkt auf unserem heutigen Zettel. Danach sind einige leistungsfähigere und komplexere, gruppierte Queries dran. Wir runden den Tag mit Datenanalysen per Mapreduce ab (ähnlich dem, was wir mit Riak gemacht haben).

Indexierung: Wenn schnell nicht schnell genug ist

Eines der nützlichen in Mongo fest eingebauten Features ist die Indexierung zur Erhöhung der Query-Performance. Wie wir gesehen haben, ist das etwas, was nicht bei allen NoSQL-Datenbanken zur Verfügung steht. MongoDB stellt mehrere der besten Datenstrukturen zur Indexierung bereit, darunter das klassische B-Tree und weitere wie zweidimensionale und sphärische ("GeoSpatial") Indizes.

Zuerst wollen wir aber ein kleines Experiment durchführen, um die Leistungsfähigkeit von MongoDBs B-Tree-Index kennenzulernen. Dazu erzeugen wir eine Reihe von Telefonnummern mit zufälligen Länderkennungen (Sie können den Code natürlich durch Ihre eigene Länderkennung ersetzen). Geben Sie den folgenden Code in der Console ein. Er generiert 100000 Telefonnummern (was eine Weile dauern kann) zwischen *1-800-555-0000* und *1-800-565-9999*.

mongo/populate_phones.js

```
populatePhones = function(area,start,stop) {
  for(var i=start; i < stop; i++) {
    var country = 1 + ((Math.random() * 8) << 0);
    var num = (country * 1e10) + (area * 1e7) + i;
    db.phones.insert({
      _id: num,
      components: {
        country: country,
        area: area,
        prefix: (i * 1e-4) << 0,
        number: i
      },
      display: "+" + country + " " + area + "-" + i
    });
  }
}
```

Führen Sie die Funktion mit einer dreistelligen Vorwahl (wie 800) und einem Bereich von siebenstelligen Zahlen aus (5550000 bis 5650000 – achten Sie bei der Eingabe auf die Nullen).

```
populatePhones( 800, 5550000, 5650000 )
db.phones.find().limit(2)
```

```
{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800,
  "prefix" : 555, "number" : 5550000 }, "display" : "+1 800-5550000" }
{ "_id" : 88005550001, "components" : { "country" : 8, "area" : 800,
  "prefix" : 555, "number" : 5550001 }, "display" : "+8 800-5550001" }
```

Wenn eine neue Collection erzeugt wird, legt Mongo automatisch einen Index über die `_id` an. Diese Indizes finden Sie in der `system.indexes`-Collection. Die folgende Query zeigt alle Indizes in der Datenbank:

```
db.system.indexes.find()
```

```
{ "name" : "_id_", "ns" : "book.phones", "key" : { "_id" : 1 } }
```

Die meisten Queries umfassen mehr Felder als die `_id`, d. h., wir müssen Indizes auch für diese Felder erzeugen.

Wir wollen einen B-Tree-Index für das `display`-Feld erzeugen. Doch zuerst wollen wir überprüfen, ob der Index die Geschwindigkeit tatsächlich erhöht. Zu diesem Zweck sehen wir uns zuerst eine Query ohne Index an. Die Methode `explain` wird genutzt, um die Details einer Operation auszugeben.

```
db.phones.find({display: "+1 800-5650001"}).explain()
```

```
{
  "cursor" : "BasicCursor",
  "nscanned" : 109999,
  "nscannedObjects" : 109999,
  "n" : 1,
  "millis" : 52,
  "indexBounds" : {
  }
}
```

Ihre Ausgabe wird nicht unserer Ausgabe entsprechen, doch beachten Sie das `millis`-Feld – die zur Verarbeitung der Query benötigte Zeit in Millisekunden – , in dem Sie üblicherweise einen zweistelligen Wert finden.

Wir erzeugen einen Index, indem wir `ensureIndex(felder,optionen)` für die Collection aufrufen. Der Parameter `felder` ist ein Objekt, das die zu indexierenden Felder enthält. Der `optionen`-Parameter beschreibt den Typ des zu erzeugenden Indizes. In unserem Beispiel bauen wir einen eindeutigen Index für `display` auf, bei dem Duplikate einfach aussortiert werden sollen.

```
db.phones.ensureIndex(
  { display : 1 },
  { unique : true, dropDups : true }
)
```

Nun lassen wir `find` erneut laufen und überprüfen mit `explain`, ob sich die Situation verbessert hat.

```
db.phones.find({ display: "+1 800-5650001" }).explain()
```

```
{
  "cursor" : "BtreeCursor display_1",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 0,
  "indexBounds" : {
    "display" : [
      [
        "+1 800-5650001",
        "+1 800-5650001"
      ]
    ]
  }
}
```

Der millis-Wert ist von 52 auf 0 gesunken – die Leistungssteigerung geht also gegen Unendlich (52 / 0)! Spaß beiseite, die Geschwindigkeit ist um eine Größenordnung gestiegen. Beachten Sie auch, dass aus dem Basic-Cursor ein B-Tree-Cursor geworden ist (er wird Cursor genannt, weil er auf die Stelle verweist, an der die Werte gespeichert sind; er enthält sie nicht). Mongo geht die Collection nicht mehr vollständig durch, sondern nutzt den Baum, um den Wert abzurufen. Noch wichtiger ist die Tatsache, dass die Zahl der gescannten Objekte von 109999 auf 1 gesunken ist, da nur ein eindeutiger Lookup notwendig war.

`explain` ist eine nützliche Funktion, doch Sie werden sie nur nutzen, um bestimmte Query-Aufrufe zu untersuchen. Ist ein Profiling in einer normalen Test- oder Produktionsumgebung nötig, benötigen Sie den *Systemprofiler*.

Wir setzen das Profiling-Level auf 2 (Level 2 speichert alle Queries, Level 1 nur langsame Queries mit einer Dauer von über 100 Millisekunden) und führen `find` wie üblich aus.

```
db.setProfilingLevel(2)
db.phones.find({ display : "+1 800-5650001" })
```

Das erzeugt ein neues Objekt in der `system.profile`-Collection, die Sie lesen können wie jede andere Tabelle auch. *ts* ist der Zeitstempel (timestamp), zu dem die Query ausgeführt wurde, *info* ist ein die Operation beschreibender String und *millis* ist die Dauer der Operation.

```
db.system.profile.find()

{
  "ts" : ISODate("2011-12-05T19:26:40.310Z"),
  "op" : "query",
  "ns" : "book.phones",
  "query" : { "display" : "+1 800-5650001" },
  "responseLength" : 146,
  "millis" : 0,
  "client" : "127.0.0.1",
  "user" : ""
}
```

Mongo kann Indizes auch für verschachtelte Werte erzeugen. Wenn Sie einen Index für alle Vorwahlen wünschen, nutzen Sie die Punktnotation für die Feldangabe: `components.area`. Im Produktionsbetrieb sollten Sie Indizes mit der Option `{ background : 1 }` immer im Hintergrund erzeugen.

```
db.phones.ensureIndex({ "components.area": 1 }, { background : 1 })
```

Wenn wir uns mit `find` alle System-Indizes für unsere `phones`-Collection ausgeben lassen, sollte der neue als letzter aufgeführt sein. Der erste Index wird immer automatisch erzeugt, um ein schnelles Lookup über die `_id` zu gewährleisten, und der zweite ist der vorhin von uns angelegte eindeutige Index.

```
db.system.indexes.find({ "ns" : "book.phones" })

{
  "name" : "_id_",
  "ns" : "book.phones",
  "key" : { "_id" : 1 }
}
{
  "_id" : ObjectId("4d2c96d1df18c2494fa3061c"),
  "ns" : "book.phones",
  "key" : { "display" : 1 },
  "name" : "display_1",
  "unique" : true,
  "dropDups" : true
}
{
  "_id" : ObjectId("4d2c982bdf18c2494fa3061d"),
  "ns" : "book.phones",
  "key" : { "components.area" : 1 },
  "name" : "components.area_1"
}
```

Unsere *book.phones*-Indizes haben Wirkung gezeigt.

Wir wollen diesen Abschnitt mit der Bemerkung abschließen, dass der Aufbau von Indizes bei einer großen Collection langsam und Ressourcen-intensiv sein kann. Sie sollten diese Auswirkungen beachten und Indizes außerhalb der Stoßzeiten erzeugen, Indizes immer im Hintergrund generieren und sie nicht automatisch, sondern von Hand anlegen. Sie finden online viele weitere Tipps und Tricks zur Indexierung, aber das sind die grundlegenden, die Sie kennen sollten.

Aggregierte Queries

Die gestern untersuchten Queries eignen sich für die einfache Extraktion von Daten, doch die Nachbearbeitung müssen Sie übernehmen. Wenn wir beispielsweise die Anzahl der Telefonnummern zwischen 559 und 9999 zählen wollen, würden wir es vorziehen, wenn die Datenbank das im Backend für uns erledigt. Wie bei PostgreSQL ist `count` einer der einfachsten Aggregatoren. Er verarbeitet eine Query und gibt eine Zahl (von Treffern) zurück.

```
db.phones.count({'components.number': { $gt : 5599999 } })

50000
```

Veränderung ist gut

Aggregierte Queries geben eine Struktur zurück, die sich von den einzelnen Dokumenten unterscheidet, an die wir gewöhnt sind. `count` aggregiert das Ergebnis in eine Anzahl von Dokumenten, `distinct` in ein Array mit Ergebnissen und `group` gibt Dokumente eigenen Designs zurück. Selbst Mapreduce betreibt einigen Aufwand, um Objekte abzurufen, die Ihren intern gespeicherten Dokumenten ähneln.

Um die Leistungsfähigkeit der folgenden aggregierenden Queries sehen zu können, wollen wir weitere 100000 Telefonnummern in unsere `phones`-Collection einfügen, diesmal aber mit einer anderen Vorwahl.

```
populatePhones( 855, 5550000, 5650000 )
```

Der Befehl distinct() gibt alle passenden Werte zurück (nicht das gesamte Dokument), für die ein oder mehrere Treffer existieren. Die eindeutigen Komponentennummern kleiner als 5550005 können wir wie folgt bestimmen:

```
db.phones.distinct('components.number', {'components.number': { $lt : 5550005 } })

[ 5550000, 5550001, 5550002, 5550003, 5550004 ]
```

Obwohl es 5550000 zweimal gibt (einmal für die Vorwahl 800 und einmal für 855), taucht sie in dieser Liste nur einmal auf.

Die Aggregat-Query `group` ähnelt dem `GROUP BY` von SQL. Sie ist auch die komplexeste der elementaren Mongo-Queries. Wir können alle Telefonnummern über 5599999 in verschiedenen Buckets speichern, gruppiert nach Vorwahlen. `key` ist das Feld, über das wir gruppieren wollen, `cond` (condition, Bedingung) ist der Wertebereich, an dem wir interessiert sind, und `reduce` verlangt eine Funktion, die festlegt, wie die Werte ausgegeben werden.

Erinnern Sie sich an das Mapreduce aus dem Riak-Kapitel? Unsere Daten sind in der vorhandenen Collection von Dokumenten bereits ge*mapped*. Ein Mapping ist also nicht mehr nötig, wir müssen nur noch das Reduce durchführen.

```
db.phones.group({
  initial: { count:0 },
  reduce:  function(phone, output) { output.count++; },
  cond:    { 'components.number': { $gt : 5599999 } },
  key:     { 'components.area' : true }
})

[ { "800" : 50000, "855" : 50000 } ]
```

Die beiden folgenden Beispiele sind zugegebenermaßen an den Haaren herbeigezogen. Sie sollen nur die Flexibilität von `group` verdeutlichen.

Sie können die `count`-Funktion sehr einfach mit dem folgenden `group`-Aufruf nachbauen. Den aggregierenden `key` lassen wir hier weg:

```
db.phones.group({
  initial: { count:0 },
  reduce:  function(phone, output) { output.count++; },
  cond:    { 'components.number': { $gt : 5599999 } }
})

[ { "count" : 100000 } ]
```

Hier haben wir zuerst ein Anfangsobjekt erzeugt, bei dem wir ein Feld namens `count` auf 0 gesetzt haben – hier angelegte Felder erscheinen in der Ausgabe. Als Nächstes beschreiben wir, was mit diesem Feld geschehen soll, indem wir eine Reduce-Funktion deklarieren, die für jedes gefundene Dokument eine 1 hinzuaddiert. Zum Schluss legen wir in einer Bedingung fest, über welche Dokumente der Reduce laufen soll. Unser Ergebnis entspricht `count`, weil die Bedingung identisch war. Wir haben den Schlüssel weggelassen, weil jedes erkannte Dokument zu unserer Liste hinzugefügt werden sollte.

Wir können auch die `distinct`-Funktion nachbauen. Aus Performancegründen erzeugen wir zuerst ein Objekt, das die Zahlen als Felder speichert (tatsächlich erzeugen wir einen Ad-hoc-*set*). In der Reduce-Funktion (die für jedes passende Dokument ausgeführt wird) setzen wir einfach den Wert als Platzhalter auf 1 (das ist das Feld, das wir wollen).

Technisch ist das alles, was wir brauchen. Wenn wir `distinct` aber tatsächlich nachbauen wollen, müssen wir ein Array von Integerwerten zurückgeben. Daher fügen wir eine `finalize(out)`-Methode hinzu, die vor der Rückgabe eines Wertes einmal ausgeführt wird. Sie wandelt das Objekt in ein Array von Feldwerten um. Die Funktion wandelt diese Zahlenstrings dann in Integerwerte um. (Wenn Sie es wirklich ganz genau wissen wollen, dann führen Sie den folgenden Code ohne `finalize` aus).

```
db.phones.group({
  initial: { prefixes : {} },
  reduce:  function(phone, output) {
    output.prefixes[phone.components.prefix] = 1;
  },
  finalize: function(out) {
    var ary = [];
    for(var p in out.prefixes) { ary.push( parseInt( p ) ); }
    out.prefixes = ary;
  }
})[0].prefixes

[ 555, 556, 557, 558, 559, 560, 561, 562, 563, 564 ]
```

Die `group`-Funktion ist, genau wie SQLs `GROUP BY`, mächtig, doch die Mongo-Implementierung hat auch ihre Kehrseite. Zum einen sind Sie auf ein Ergebnis von 10000 Dokumenten beschränkt. Zum anderen funktioniert `group` nicht, wenn Sie Ihre Mongo-Collection aufteilen (Shardinf, was wir morgen machen werden). Außerdem gibt es flexiblere Möglichkeiten, Queries aufzubauen. Aus diesem und anderen Gründen tauchen wir gleich in die MongoDB-Version von Mapreduce ein. Doch vorher wollen wir auf die Grenzen zwischen client- und serverseitigen Befehlen eingehen. Dieser Unterschied hat erhebliche Konsequenzen für Ihre Anwendungen.

Serverseitige Befehle

Wenn Sie die folgende Funktion in der Kommandozeile (oder über einen Treiber) ausführen, würde der Client jede Telefonnummer, also alle 100000, abrufen, und jedes phone-Dokument, eins nach dem anderen, wieder auf dem Server speichern.

mongo/update_area.js

```
update_area = function() {
  db.phones.find().forEach(
    function(phone) {
      phone.components.area++;
      phone.display = "+"+
        phone.components.country+" "+
        phone.components.area+"-"+
        phone.components.number;
      db.phone.update({ _id : phone._id }, phone, false);
    }
  )
}
```

Das Mongo `db`-Objekt bietet aber einen `eval`-Befehl an, der die angegebene Funktion an den Server übergibt. Dadurch wird das Geschnatter zwischen Client und Server drastisch reduziert, weil der Code auf dem Server ausgeführt wird.

```
> db.eval(update_area)
```

Neben der Evaluierung von JavaScript-Funktionen gibt es verschiedene andere eingebaute Mongo-Befehle, die auf dem Server ausgeführt werden, auch wenn einige nur für die admin-Datenbank genutzt werden können (die Sie über `use admin` auswählen).

```
> use admin
> db.runCommand("top")
```

Der `top`-Befehl gibt Details zum Zugriff auf alle Collections des Servers aus.

```
> use book
> db.listCommands()
```

Wenn Sie `listCommands()` ausführen, werden Sie viele der von uns genutzten Befehle sehen. Tatsächlich lassen sich viele gängige Befehle über die `runCommand`-Methode ausführen, etwa das Zählen der Telefonnummern, doch die Ausgabe ist etwas anders.

```
> db.runCommand({ "count" : "phones" })
{ "n" : 100000, "ok" : 1 }
```

Die Zahl (*n*) wird korrekt (100000) zurückgegeben, doch das Objekt enthält ein *ok*-Feld. Das liegt daran, dass `db.phones.count` eine Wrapper-Funktion ist, die unserer Bequemlichkeit zuliebe vom JavaScript-Interface der Shell erzeugt wurde, während `runCommand` den Zähler auf dem Server ausführt. Erinnern Sie sich daran, dass wir Detektiv spielen und uns die Implementierung von `count` ansehen können, indem wir die Klammern weglassen.

```
> db.phones.count
function (x) {
    return this.find(x).count();
}
```

Interessant! `collection.count` ist nur ein Wrapper für den Aufruf von `count` über die Ergebnisse von `find` (das selbst nur ein Wrapper auf ein natives Query-Objekt ist, der einen Cursor zurückgibt, der auf das Ergebnis verweist). Wenn wir uns nun *diese* Query ansehen…

```
> db.phones.find().count
```

sehen wir eine viel größere Funktion (zu groß, um sie hier abzubilden). Schaut man sich den Code an, findet man nach einem ganzen Haufen Setup Zeilen wie diese:

```
var res = this._db.runCommand(cmd);
        if (res && res.n != null) {
            return res.n;
        }
```

Sehr interessant! `count` führt `runCommand` aus und gibt den Wert des `n`-Feldes zurück.

runCommand

Und da wir uns gerade anschauen, wie Methoden funktionieren, wollen wir auch einen Blick auf die `runCommand`-Funktion werfen.

```
> db.runCommand
function (obj) {
    if (typeof obj == "string") {
        var n = {};
        n[obj] = 1;
        obj = n;
    }
    return this.getCollection("$cmd").findOne(obj);
}
```

Wie sich zeigt, ist `runCommand` ebenfalls ein Wrapper für den Aufruf einer Collection namens `$cmd`. Sie können jeden Befehl ausführen, indem Sie diese Collection direkt aufrufen.

```
> db.$cmd.findOne({'count' : 'phones'})
{ "n" : 100000, "ok" : 1 }
```

Wir sind ganz tief unten angekommen und sehen, wie Treiber generell mit dem Mongo-Server kommunizieren.

Umweg

Wir sind diesen Umweg aus zwei Gründen gegangen:

- Um Ihnen zu vermitteln, dass ein Großteil der Magie, den Sie in der `mongo`-Console anstoßen, auf dem Server und nicht dem Client ausgeführt wird. Der Client stellt nur praktische Wrapper-Funktionen zur Verfügung.
- Wir können dieses Konzept der Ausführung serverseitigen Codes auch zu unserem Vorteil nutzen und bei MongoDB etwas aufbauen, was den *Stored Procedures* von PostgreSQL ähnelt.

Jede JavaScript-Funktion kann in einer speziellen Collection namens `system.js` gespeichert werden. Das ist eine ganz normale Collection und Sie speichern eine Funktion, indem Sie den Namen als `_id`, und das Funktionsobjekt als `value` verwenden.

```
> db.system.js.save({
    _id:'getLast',
    value:function(collection){
      return collection.find({}).sort({'_id':1}).limit(1)[0]
    }
})
```

Als Nächstes würden wir den Code direkt auf dem Server ausführen. Die `eval`-Funktion übergibt den String an den Server, evaluiert ihn als JavaScript-Code und gibt das Ergebnis zurück.

```
> db.eval('getLast(db.phones)')
```

Das sollte das gleiche Ergebnis zurückliefern wie der lokale Aufruf von `getLast(collection)`.

```
> db.system.js.findOne({'_id': 'getLast'}).value(db.phones)
```

Beachten Sie aber, dass `eval()` den Mongo-Daemon `mongod` während der Ausführung blockiert, d. h., es eignet sich hauptsächlich für einmalige Dinge oder für Tests, nicht aber für Prozeduren im Produktiveinsatz. Sie können diese Funktion auch innerhalb von `$where` und bei Mapreduce verwenden. Nun haben wir alle Dinge beisammen, um Mapreduce unter MongoDB zu nutzen.

Mapreduce (und Finalize)

Das Mapreduce-Muster von Mongo ähnelt Riak, es gibt aber einige kleine Unterschiede. Statt die `map`-Funktion einen umgewandelten Wert zurückliefern zu lassen, verlangt Mongo zum Mappen den Aufruf einer `emit`-Funktion mit einem Schlüssel. Der Vorteil besteht darin, dass Sie für ein Dokument mehr als einmal „emittieren“ können. Die `reduce`-Funktion verlangt einen einzelnen Schlüssel und eine Liste von Werten, die für diesen Schlüssel emittiert wurden. Mongo kennt schließlich noch einen optionalen dritten Schritt namens `finalize`, der für jeden abgebildeten Wert nur einmal ausgeführt wird, nachdem alle Reducer gelaufen sind. Auf diese Weise können Sie abschließende Berechnungen oder Aufräumarbeiten durchführen, falls das notwendig sein sollte.

Da wir die Grundlagen von Mapreduce bereits kennen, sparen wir uns ein einleitendes Beispiel und steigen gleich richtig ein. Wir wollen einen Bericht erstellen, der alle Telefonnummern zählt, die (nach Land) die gleichen Ziffern enthalten. Zuerst speichern wir eine Hilfsfunktion, die ein Array aller unterschiedlichen Nummern erzeugt. (Wie die Hilfsfunktion funktioniert, ist für das Verständnis der Mapreduce-Gesamtoperation nicht von Bedeutung.)

mongo/distinct_digits.js

```
distinctDigits = function(phone){
  var
    number = phone.components.number + '',
    seen = [],
    result = [],
    i = number.length;
  while(i-) {
    seen[+number[i]] = 1;
  }
  for (i=0; i<10; i++) {
    if (seen[i]) {
      result[result.length] = i;
    }
  }
  return result;
}
db.system.js.save({_id: 'distinctDigits', value: distinctDigits})
```

Laden Sie die Datei in die `mongo`-Kommandozeile. Wenn die Datei im gleichen Verzeichnis liegt, in dem Sie `mongo` gestartet haben, müssen Sie nur den Dateinamen angeben, anderenfalls wird der vollständige Pfad benötigt.

```
> load('distinct_digits.js')
```

Nachdem das erledigt ist, können Sie einen kurzen Test durchführen (wenn es Probleme gibt, sollten Sie sich nicht scheuen, einige `print`-Funktionen einzustreuen).

```
db.eval("distinctDigits(db.phones.findOne({ 'components.number' : 5551213 }))")

[ 1, 2, 3, 5 ]
```

Nun können wir am Mapper arbeiten. Wie bei jeder Mapreduce-Funktion ist die wichtigste Entscheidung, welche Felder abgebildet werden sollen, da das die zurückgegebenen aggregierten Werte bestimmt. Da unser Bericht eindeutige Nummern zurückgibt, ist das Array eindeutiger Werte ein Feld. Doch weil wir auch das Land abfragen müssen, ist dies ein weiteres Feld. Wir verwenden beide Werte als zusammengesetzten Schlüssel: `{digits : X, country : Y}`.

Unser Ziel besteht einfach darin, diese Werte zu zählen, d. h., wir emittieren einfach den Wert 1 (jedes Dokument ist ein zu zählendes Element). Die Aufgabe des Reducers besteht darin, die ganzen Einsen zu addieren.

mongo/map_1.js

```
map = function() {
  var digits = distinctDigits(this);
  emit({digits : digits, country : this.components.country}, {count : 1});
}
```

mongo/reduce_1.js

```
reduce = function(key, values) {
  var total = 0;
  for(var i=0; i<values.length; i++) {
    total += values[i].count;
  }
  return { count : total };
}

results = db.runCommand({
  mapReduce: 'phones',
  map:       map,
  reduce:    reduce,
  out:       'phones.report'
})
```

Da wir den Namen der Collection über den `out`-Parameter festgelegt haben (`out : 'phones.report'`), können wir das Ergebnis abfragen wie jede andere Collection auch. Wir haben einen View entwickelt, der auch in der `show tables`-Liste zu finden ist.

```
> db.phones.report.find({'_id.country' : 8})
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 },
  "value" : { "count" : 19 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 },
  "value" : { "count" : 3 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6 ], "country" : 8 },
  "value" : { "count" : 48 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 7 ], "country" : 8 },
  "value" : { "count" : 12 }
}
has more
```

Geben Sie `it` ein, um die Iteration über die Ergebnisse fortzusetzen. Beachten Sie, dass die eindeutig emittierten Schlüssel unter dem Feld `_id` liegen und dass alle vom Reducer zurückgelieferten Daten unter dem Feld `value` zu finden sind.

Soll der Mapreducer einfach nur das Ergebnis anstelle der Collection ausgeben, können Sie den `out`-Wert auf `{ inline : 1 }` setzen, denken Sie aber daran, dass die Größe der Ergebnismenge begrenzt ist. Bei Mongo 2.0 liegt diese Grenze bei 16MB.

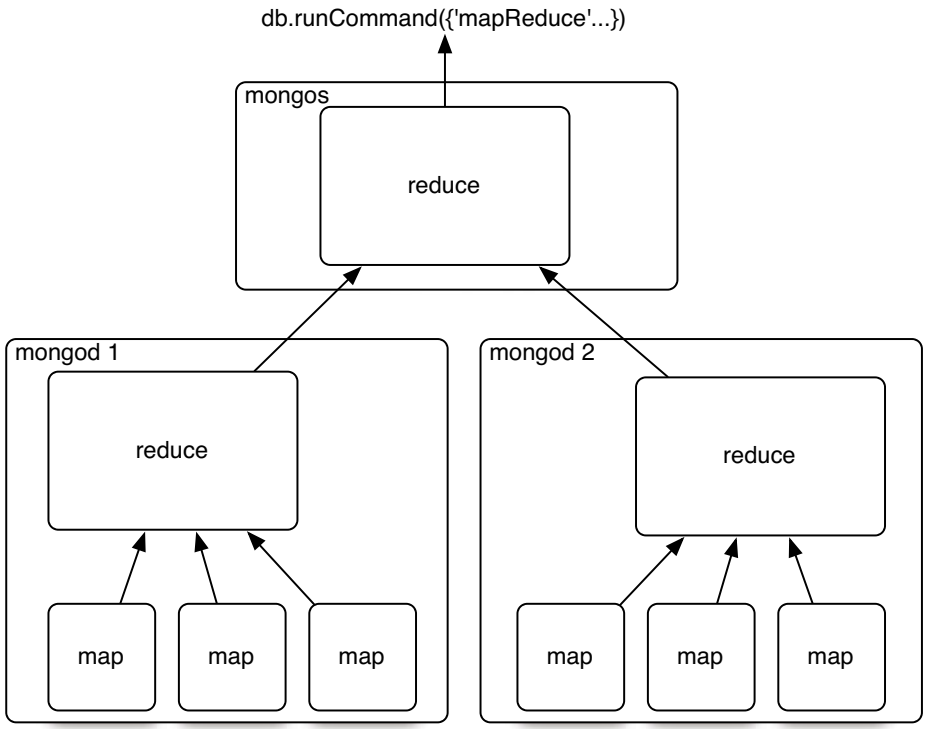


Abbildung 22: Ein Mongo Map/Reduce-Aufruf über zwei Server

Erinnern Sie sich aus dem Riak-Kapitel daran zurück, dass Reducer entweder „gemappte“ (emittierte) Ergebnisse oder die Ergebnisse anderer Reducer als Eingabe nutzen können. Warum würde man die Ausgabe eines Reducers als Eingabe für einen anderen verwenden wollen, wo sie doch an den gleichen Schlüssel gebunden sind? Stellen Sie sich vor, wie das aussehen würde, wenn mehrere Server im Spiel sind, wie in Abbildung 22, *Ein Mongo Map/Reduce-Aufruf über zwei Server* zu sehen.

Jeder Server muss seine eigenen `map`- und `reduce`-Funktionen ausführen und die Ergebnisse dann an den Service zurückgeben, der sie initiiert hat, damit der sie einsammeln kann. Klassisches Teilen und Herrschen. Hätten wir die Ausgabe des Reducers `total` statt `count` genannt, müssten wir in der Schleife beide Fälle verarbeiten:

mongo/reduce_2.js

```
reduce = function(key, values) {
  var total = 0;
  for(var i=0; i<values.length; i++) {
    var data = values[i];
    if('total' in data) {
      total += data.total;
    } else {
      total += data.count;
    }
  }
  return { total : total };
}
```

Allerdings hat Mongo vorausgesehen, dass Sie einige abschließende Änderungen vornehmen, etwa ein Feld umbenennen oder einige andere Berechnungen durchführen müssen. Wenn Sie das Ausgabefeld tatsächlich `total` nennen müssen, können Sie eine `finalize()`-Funktion implementieren, die genauso funktioniert wie die `finalize`-Funktion unter `group()`.

Was wir am zweiten Tag gelernt haben

Am zweiten Tag haben wir unser Query-Arsenal um aggregierte Queries erweitert: `count()`, `distinct()` und schließlich `group()`. Um die Antwortzeiten dieser Queries zu beschleunigen, haben wir die Indexierungsoptionen von MongoDB genutzt. Wenn noch mehr Leistung gefordert ist, steht das allgegenwärtige `mapReduce()` zur Verfügung.

Tag 2: Selbststudium

Finden Sie heraus

1. Einen Kurzbefehl für Admin-Befehle.
2. Die Online-Dokumentation für Queries und Cursor.
3. Die MongoDB-Dokumentation für Mapreduce.
4. Untersuchen Sie über die JavaScript-Schnittstelle den Code für drei Collection-Funktionen: `help()`, `findOne()` und `stats()`.

Machen Sie Folgendes

1. Implementieren Sie eine finalize-Methode, die „count“ als „total“ ausgibt.
2. Installieren Sie einen Mongo-Treiber für eine Sprache Ihrer Wahl und stellen Sie damit die Verbindung zur Datenbank her. Befüllen Sie darüber eine Collection und indexieren Sie eines der Felder.

5.4 Tag 3: Replica-Sets, Sharding, GeoSpatial und GridFS

Mongo hat leistungsfähige Möglichkeiten, Daten auf unterschiedliche Art und Weise zu speichern und abzurufen. Andererseits können das andere Datenbanken auch. Was Dokumenten-Datenbanken so einzigartig macht, ist ihre Fähigkeit, beliebig verschachtelte, schemafreie Datendokumente zu verarbeiten. Was Mongo im Bereich der Dokumentenspeicher zu etwas Besonderem macht, ist seine Fähigkeit, über mehrere Server zu skalieren. Es kann Collections replizieren (auf andere Server kopieren) oder in mehrere Teile zerlegen (Sharding) und Queries parallel ausführen. Das erhöht die Verfügbarkeit.

Replica-Sets

Mongo wurde nicht als Standalone-Lösung entwickelt, sondern um zu wachsen. Datenkonsistenz und Partitionstoleranz sind wichtige Aspekte, doch die Aufteilung von Daten (Sharding) hat ihren Preis: Geht ein Teil einer Collection verloren, wirkt sich das auf alle Daten aus. Wie gut kann die Abfrage einer Länder-Collection sein, wenn sie nur die westliche Hemisphäre enthält? Mongo löst diese implizite Sharding-Schwäche auf ganz einfache Weise: Duplikation. Sie werden in der Produktion keine einzelne Mongo-Instanz einsetzen, sondern die Daten über mehrere Services replizieren.

Heute halten wir uns nicht mit der vorhandenen Datenbank auf, sondern beginnen von vorne und starten einige neue Server. Mongos Standardport ist 27017, weshalb wir jedem Server einen anderen Port geben. Denken Sie daran, dass Sie zuerst die Datenverzeichnisse erzeugen müssen, weshalb wir drei davon anlegen:

```
$ mkdir  ./mongo1 ./mongo2 ./mongo3
```

Als Nächstes starten wir die Mongo-Server. Diesmal setzen wir das `replSet`-Flag für *book* und geben auch den Port mit an. Und da wir gerade dabei sind, setzen wir auch das REST-Flag, damit wir das Web-Interface nutzen können.

```
$ mongod --replSet book --dbpath ./mongo1 --port 27011 --rest
```

Öffnen Sie zwei weitere Terminal-Fenster und führen Sie die folgenden Befehle aus, die weitere Server mit unterschiedlichen Datenverzeichnissen und an anderen Ports starten.

```
$ mongod --replSet book --dbpath ./mongo2 --port 27012 --rest
$ mongod --replSet book --dbpath ./mongo3 --port 27013 --rest
```

Beim Start werden jede Menge Ausgaben erzeugt, darunter auch

```
[startReplSets] replSet can't get local.system.replset config from self \
 or any seed (EMPTYCONFIG)
```

Das ist eine gute Sache; wir haben unseren Replikationssatz (Replica-Set) noch nicht initialisiert und Mongo lässt uns das wissen. Öffnen Sie eine `mongo`-Shell mit einem der Server und führen Sie die Funktion `rs.initiate()` aus.

```
$ mongo localhost:27011
> rs.initiate({
  _id: 'book',
  members: [
    {_id: 1, host: 'localhost:27011'},
    {_id: 2, host: 'localhost:27012'},
    {_id: 3, host: 'localhost:27013'}
  ]
})
> rs.status()
```

Beachten Sie, dass wir ein neues Objekt namens `rs` (Replica-Set) nutzen. Wie andere Objekte auch besitzt es eine `help`-Methode, die Sie aufrufen können. Der `status`-Befehl informiert uns darüber, wann der Replica-Set läuft. Prüfen Sie den Status und warten Sie auf die Fertigstellung, bevor Sie weitermachen. Wenn Sie sich die Ausgabe der drei Server ansehen, sollten Sie auf einem die folgende Ausgabe sehen:

```
[rs Manager] replSet PRIMARY
```

Auf den beiden anderen Servern erscheint die folgende Zeile:

```
[rs_sync] replSet SECONDARY
```

PRIMARY ist der Master-Server. Die Chancen stehen hoch, dass das der Server an Port 27011 ist (da er als erster gestartet wurde). Welcher es auch immer ist, öffnen Sie nun eine Console zum Primary. Fügen Sie etwas über die Kommandozeile ein, und dann führen wir ein Experiment durch.

```
> db.echo.insert({ say : 'HELLO!' })
```

Nach dem Einfügen beenden Sie die Console, und dann wollen wir überprüfen, ob unsere Änderung repliziert wurde, indem wir den Master-Knoten herunterfahren. Dazu reicht es aus, Ctrl+C zu drücken. Wenn Sie sich nun die Logs der beiden verbliebenen Server ansehen, sollten Sie erkennen können, dass einer der beiden zum Master aufgestiegen ist (er gibt die replSet PRIMARY-Zeile aus). Öffnen Sie eine Console mit diesem Rechner (bei uns war es *localhost:27012*), und db.echo.find() sollte den eben eingefügten Wert enthalten.

Wir wollen eine weitere Runde in unserem Console-wechsel-dich-Spielchen spielen. Öffnen Sie eine Console mit dem verbliebenen SECONDARY-Server. Um ganz sicherzugehen, führen Sie die isMaster-Funktion aus. Bei uns sah das wie folgt aus:

```
$ mongo localhost:27013
MongoDB shell version: 1.6.2
connecting to: localhost:27013/test
> db.isMaster()
{
  "setName" : "book",
  "ismaster" : false,
  "secondary" : true,
  "hosts" : [
    "localhost:27013",
    "localhost:27012",
    "localhost:27011"
  ],
  "primary" : "localhost:27012",
  "ok" : 1
}
```

In dieser Shell wollen wir einen weiteren Wert einfügen.

```
> db.echo.insert({ say : 'is this thing on?' })
not master
```

Die Meldung *not master* lässt uns wissen, dass wir nicht an einen sekundären Knoten schreiben können. Und auch ein direktes Lesen von dort ist nicht möglich. Es gibt nur einen Master pro Replica-Set und Sie müssen diesen Master verwenden. Er ist der Wächter über dieses Set.

Bei der Replikation von Daten gibt es ganz eigene Aspekte, die bei einer einfachen Datenbank nicht auftauchen. Beim Mongo-Setup besteht ein Problem darin, zu entscheiden, welcher Server zum Master wird, wenn der Master-Knoten ausfällt. Mongo löst das Problem, indem es bei jedem `mongod` nachfragt und den mit den aktuellsten Daten zum neuen Master befördert. Im Moment sollten noch zwei `mongod`-Instanzen laufen. Gehen Sie nun hin und fahren Sie den aktuellen Master herunter. Denken Sie daran, dass bisher in diesem Fall einfach ein anderer zum neuen Master auserkoren wurde. Doch diesmal passiert etwas anderes. Die Ausgabe des letzten verbliebenen Servers sieht etwa so aus:

```
[ReplSetHealthPollTask] replSet info localhost:27012 is now down (or...
[rs Manager] replSet can't see a majority, will not try to elect self
```

Das hat mit der Mongo-Philosophie der Server-Setups zu tun und ist der Grund, warum Sie immer eine ungerade Zahl von Servern verwenden sollten (drei, fünf und so weiter).

Starten Sie nun wieder die anderen Server und sehen Sie sich die Logs an. Wenn die Knoten wieder laufen, wechseln Sie in den Wiederherstellungsmodus und versuchen, Ihre Daten mit dem neuen Master-Knoten abzugleichen. „Moment mal!?“ (hören wir Sie sagen). „Was passiert, wenn der eigentliche Master Daten besitzt, die noch nicht propagiert wurden?“ Diese Operationen werden verworfen. Eine Schreiboperation in einem Mongo Replica-Set wird nicht als erfolgreich erachtet, bis die meisten Knoten eine Kopie der Daten besitzen.

Das Problem mit der geraden Knotenzahl

Das Konzept der Replikation ist einfach zu verstehen: Sie schreiben etwas an einen MongoDB-Server und diese Daten werden innerhalb des Replika-Sets auf die anderen kopiert. Ist ein Server nicht verfügbar, übernimmt einer der anderen und verarbeitet die Requests. Doch es muss nicht an einem Absturz liegen, dass ein Server nicht verfügbar ist. Manchmal ist die Netzwerkverbindung zwischen den Knoten unterbrochen. In diesem Fall schreibt Mongo vor, dass *eine Majorität von Knoten, die immer noch kommunizieren können, das Netzwerk bilden*.

MongoDB erwartet eine ungerade Anzahl von Knoten im Replika-Set. Betrachten wir zum Beispiel ein aus fünf Knoten bestehendes Netzwerk. Wenn ein Verbindungsausfall es in ein 3-Knoten- und ein 2-Knoten-Fragment teilt, hat das größere Fragment eindeutig die Majorität, es kann einen Master wählen und weiterhin Requests verarbeiten. Ohne Mehrheit ist man nicht beschlussfähig.

Wahlrecht und Arbiter

Sie werden nicht immer eine ungerade Anzahl von Servern haben, die Daten replizieren. In diesem Fall können Sie entweder einen Arbiter starten (was generell empfohlen wird) oder das Wahlrecht erhöhen (was generell nicht empfohlen wird). Bei Mongo ist ein Arbiter ein Server mit Wahlrecht, der aber im fraglichen Set nicht repliziert. Sie starten ihn wie jeden anderen Server, setzen aber in der Konfiguration ein entsprechendes Flag: `{_id: 3, host: 'localhost:27013', arbiterOnly : true}`. Arbiter sind bei Stimmengleichheit nützlich, wie der Vizepräsident im amerikanischen Senat. Standardmäßig hat jede `mongod`-Instanz eine einzige Stimme.

Um zu erkennen, warum eine ungerade Anzahl von Knoten bevorzugt wird, stellen Sie sich vor, was bei einem Replika-Set mit vier Knoten passieren könnte. Nehmen wir an, dass ein Netzwerksegment ausfällt und zwei Server die Verbindung zu den anderen verlieren. Ein Set enthält den ursprünglichen Master, doch da er keine *klare Majorität* des Netzwerks sieht, tritt der Master zurück. Im anderen Set kann ebenfalls kein Master gewählt werden, da ebenfalls nicht mit einer klaren Mehrheit der Knoten kommuniziert werden kann. Beide Sets sind nun nicht in der Lage, Requests zu verarbeiten, d. h., das System ist zusammengebrochen. Bei einer ungeraden Anzahl von Knoten ist dieses spezielle Szenario – ein fragmentiertes Netzwerk, bei dem kein Fragment die klare Mehrheit besitzt – weit weniger wahrscheinlich.

Einige Datenbanken (z. B. CouchDB) erlauben mehrere Master. Das ist bei Mongo nicht der Fall und deshalb ist es auch nicht dafür ausgelegt, einen Datenabgleich zwischen ihnen durchzuführen. MongoDB löst Konflikte zwischen mehreren Mastern einfach dadurch, dass es sie nicht erlaubt.

Im Gegensatz zu (beispielsweise) Riak kennt Mongo immer den aktuellsten Wert. Der Client muss nicht entscheiden. Mongos Anliegen ist eine hohe Konsistenz bei Schreiboperationen, und das Multimaster-Szenario zu vermeiden, ist kein schlechter Ansatz, um das zu erreichen.

Sharding

Einer der Hauptgründe, warum Mongo überhaupt existiert, ist die sichere und schnelle Verarbeitung großer Datenmengen. Die offenkundigste Möglichkeit, das zu erreichen, ist ein horizontales Aufteilen nach Wertebereichen – kurz *Sharding* genannt. Statt alle Werte einer Collection auf einem Server vorzuhalten, werden einige Wertebereiche auf andere Server aufgeteilt. Bei unserer Telefonnummern-Collection könnten wir zu Beispiel alle Telefonnummern kleiner 1-500-000-0000 auf dem Mongo-Server A speichern und

mongos vs. mongoconfig

Sie werden sich fragen, warum Mongo die Einstiegspunkte für `configuration` und `mongos` auf zwei verschiedene Server verteilt. Das liegt daran, dass sie in Produktionsumgebungen generell auf verschiedenen physikalischen Servern liegen werden. Der config-Server (selbst repliziert) verwaltet die Sharding-Informationen für andere Sharding-Server, während `mongos` wahrscheinlich auf Ihrem lokalen Anwendungsserver laufen, an den sich Clients leicht anbinden können (ohne sich darum kümmern zu müssen, mit welchen Shards die Verbindung herzustellen ist).

alle Nummern größer oder gleich 1-500-000-0001 auf Server B. Mongo erleichtert das über ein Autosharding, das die Aufteilung für Sie übernimmt.

Wir wollen eine Reihe (nicht replizierender) `mongod`-Server starten. Wie bei Replika-Sets ist ein spezieller Parameter nötig, um als Sharding-Server zu gelten (was nur bedeutet, dass der Server Sharding beherrscht).

```
$ mkdir ./mongo4 ./mongo5
$ mongod --shardsvr --dbpath ./mongo4 --port 27014
$ mongod --shardsvr --dbpath ./mongo5 --port 27015
```

Nun benötigen wir einen Server, um unsere Schlüssel nachzuhalten. Nehmen wir an, wir haben eine Tabelle mit alphabetisch sortierten Städtenamen angelegt. Wir benötigen nun eine Möglichkeit, herauszufinden, dass (zum Beispiel) die mit A – N beginnenden Städte auf dem Server mongo4 liegen und O – Z auf dem Server mongo5. Bei Mongo legen Sie einen *config-Server* an (ein gewöhnlicher `mongod`), der nachhält, welcher Server (mongo4 oder mongo5) welche Werte enthält.

```
$ mkdir ./mongoconfig
$ mongod --configsvr --dbpath ./mongoconfig --port 27016
```

Abschließend müssen Sie einen vierten Server namens `mongos`starten, der als (einziger) Einstiegspunkt für Ihre Clients fungiert. Die `mongos`-Server stellen die Verbindung mit dem `mongoconfig`.Server her, um die dort gespeicherten Sharding-Informationen nachzuhalten. Wir lassen ihn an Port 27020 laufen und setzen die `chunkSize` auf 1. (Unsere `chunkSize` beträgt 1MB, was den kleinsten erlaubten Wert darstellt. Das ist unserer kleinen Datenmenge geschuldet, damit wir das Sharding in Aktion sehen. Im Produktiveinsatz würden Sie die Voreinstellung oder einen wesentlich größeren Wert verwenden.) Wir verweisen `mongos` über das Flag `-configdb` an den config-Server:Port.

```
$ mongos --configdb localhost:27016 --chunkSize 1 --port 27020
```

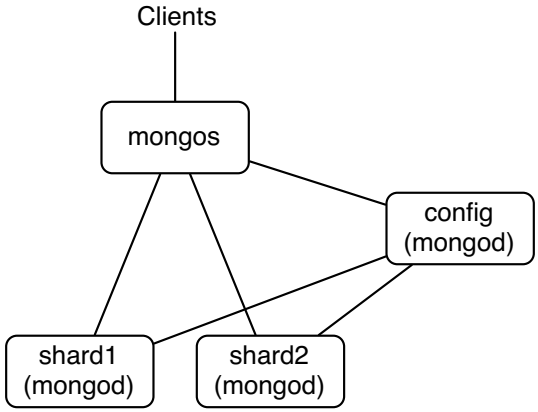


Abbildung 23: Unser kleiner Sharding-Cluster

Eine schöne Sache bei `mongos` ist, dass es sich um einen leichtgewichtigen Klon eines vollständigen `mongod`-Servers handelt. Nahezu alle Befehle, die Sie an `mongod` übergeben können, werden auch von `mongos` verstanden, was es für Clients zu einem perfekten Vermittler zwischen mehreren Sharding-Servern macht. Ein Bild unseres Server-Setups ist hilfreich und in Abbildung 23, *Unser kleiner Sharding-Cluster* zu sehen.

Nun wollen wir in der Console des `mongos`-Servers in die admin-Datenbank wechseln und das Sharding konfigurieren.

```
$ mongo localhost:27020/admin
> db.runCommand( { addshard : "localhost:27014" } )
{ "shardAdded" : "shard0000", "ok" : 1 }
> db.runCommand( { addshard : "localhost:27015" } )
{ "shardAdded" : "shard0001", "ok" : 1 }
```

Nachdem das eingerichtet ist, müssen wir die Datenbank und die Kollektion angeben, bei denen das Sharding genutzt werden soll, sowie das Feld, über das das Sharding erfolgen soll (in unserem Beispiel der Städtename).

```
> db.runCommand( { enablesharding : "test" } )
{ "ok" : 1 }
> db.runCommand( { shardcollection : "test.cities", key : {name : 1} } )
{ "collectionsharded" : "test.cities", "ok" : 1 }
```

Nachdem das Setup abgeschlossen ist, wollen wir einige Daten laden. Wenn Sie den Code zum Buch herunterladen, finden Sie eine 12MB große Datei

namens `mongo_cities1000.json`, die die Daten jeder Stadt auf der Welt mit über 1000 Einwohnern enthält. Laden Sie diese Datei herunter und führen Sie das folgende Skript aus, um die Daten in den `mongos`-Server zu importieren:

```
$ mongoimport -h localhost:27020 -db test --collection cities \
  --type json mongo_cities1000.json
```

In der `mongos`-Console geben Sie dann `use test` ein, um aus der `admin`-Umgebung wieder in die `test`-Umgebung zu wechseln.

Geodaten-Queries

Mongo beherrscht von Haus aus einen netten Trick. Auch wenn wir uns an diesem Tag auf Server-Setups konzentriert haben, wäre kein Tag komplett ohne ein wenig Rambazamba, und das ist Mongos Fähigkeit, Geodaten abfragen zu können. Stellen Sie zuerst die Verbindung mit dem `mongos` Sharding-Server her.

```
$ mongo localhost:27020
```

Den Kern des Geodaten-Geheimnisses bildet die Indexierung. Es handelt sich um eine spezielle Form der Indexierung von Geodaten, die als *Geohash* bezeichnet wird und die nicht nur bestimmte Werte oder Wertebereiche sehr schnell findet, sondern auch in der Nähe liegende Werte. Praktischerweise haben wir am Ende des vorigen Abschnitts auch sehr viele Geodaten installiert. Um sie abfragen zu können, besteht der erste Schritt in der Indexierung der Daten im location-Feld. Der *2te* Index muss für die beiden Werte-Felder gesetzt werden, in unserem Fall ein Hash (z. B. `{ longitude:1.48453, latitude:42.57205 }`), doch es könnte sich ebenso gut um ein Array (zum Beispiel `[1.48453, 42.57205]`) handeln.

```
> db.cities.ensureIndex({ location : "2d" })
```

Wenn wir nicht mit Sharding arbeiten würden, könnten wir Städte einfach an oder in der Nähe eines Ortes abfragen. Bei unserer aktuellen Mongo-Version funktioniert der folgende Befehl aber nur bei nicht verteilten Collections.

```
> db.cities.find({ location : { $near : [45.52, -122.67] } }).limit(5)
```

Das dürfte in zukünftigen Versionen auch mit verteilten Collections funktionieren. Bis dahin müssen Sie die verteilte Collection `cities` über den `geoNear()`-Befehl nach Städten in der Nähe eines bestimmten Ortes abfragen. Hier ein beispielhaftes Ergebnis:

```
> db.runCommand({geoNear : 'cities', near : [45.52, -122.67],
  num : 5, maxDistance : 1})
{
  "ns" : "test.cities",
  "near" : "1000110001000000011100101011100011001001110001111110",
  "results" : [
    {
      "dis" : 0.007105400003747849,
      "obj" : {
        "_id" : ObjectId("4d81c216a5d037634ca98df6"),
        "name" : "Portland",
        ...
      }
    },
    ...
  ],
  "stats" : {
    "time" : 0,
    "btreelocs" : 53,
    "nscanned" : 49,
    "objectsLoaded" : 6,
    "avgDistance" : 0.02166813996454613,
    "maxDistance" : 0.07991909980773926
  },
  "ok" : 1
}
```

`geoNear` hilft auch bei der Fehlersuche von Geodaten-Befehlen. Er liefert eine Vielzahl nützlicher Informationen zurück, etwa die Distanz vom abgefragten Punkt, die durchschnittliche und maximale Distanz vom zurückgelieferten Set sowie Indexinformationen.

GridFS

Ein Nachteil eines verteilten Systems ist das Fehlen eines einzelnen kohärenten Dateisystems. Nehmen wir an, Sie betreiben eine Website, bei der die Benutzer selbst Bilder hochladen können. Wenn Sie mehrere Webserver auf verschiedenen Knoten betreiben, müssen Sie die hochgeladenen Images manuell auf die Festplatten der verschiedenen Webserver kopieren oder ein alternatives zentrales System aufbauen. Mongo handhabt dieses Szenario mit einem eigenen verteilten Dateisystem namens GridFS.

Mongo wird mit einem Kommandozeilen-Tool ausgeliefert, über das man mit GridFS kommunizieren kann. Das Gute ist, dass wir nichts einrichten müssen, um es nutzen zu können. Wenn wir uns die Dateien in den von `mongos` verwalteten Shards über den Befehl `mongofiles` ausgeben lassen, erhalten wir eine leere Liste.

```
$ mongofiles -h localhost:27020 list

connected to: localhost:27020
```

Laden Sie eine beliebige Datei hoch

```
$ mongofiles -h localhost:27020 put my_file.txt

connected to: localhost:27020
added file: { _id: ObjectId('4d81cc96939936015f974859'), filename: "my_file.txt", \
  chunkSize: 262144, uploadDate: new Date(1300352150507), \
  md5: "844ab0d45e3bded0d48c2e77ed4f3b0e", length: 3067 }
done!
```

und *voilà*! Wenn wir uns nun den Inhalt von `mongofiles` ansehen, finden wir auch die hochgeladene Datei.

```
$ mongofiles -h localhost:27020 list

connected to: localhost:27020
my_file.txt  3067
```

Zurück in der `mongo`-Console können wir die Collections sehen, in denen Mongo die Daten speichert.

```
> show collections

cities
fs.chunks
fs.files
system.indexes
```

Da es sich um gute alte Collections handelt, können sie wie alle anderen auch repliziert und abgefragt werden.

Was wir am dritten Tag gelernt haben

Damit schließen wir unsere Untersuchung von MongoDB ab. Heute haben wir uns darauf konzentriert, wie Mongo die Haltbarkeit von Daten über Replika-Sets erhöht und die horizontale Skalierung mittels Sharding unterstützt. Wir haben uns gute Server-Konfigurationen angesehen und gezeigt, wie Mongo den `mongos`-Server als Relais für das Autosharding zwischen mehreren Knoten nutzt. Abschließend haben wir mit einigen in Mongo integrierten Tools wie Geodaten-Queries und GridFS herumgespielt.

Tag 3: Selbststudium

Finden Sie heraus

1. Sehen Sie sich alle Konfigurationsoptionen für Replika-Sets in der Online-Dokumentation an.
2. Finden Sie heraus, wie man einen sphärischen Geodaten-Index erzeugt.

Machen Sie Folgendes

1. Mongo unterstützt begrenzende Umrisse (bounding shapes) (genauer: Quadrate und Kreise). Finden Sie alle Städte innerhalb einer 50-Meilen-Box um das Zentrum von London.[3]
2. Lassen Sie sechs Server laufen: drei Server pro Replika-Set und jedes Replika-Set in einem von zwei Shards. Lassen Sie einen Config-Server und mongos laufen. Lassen Sie GridFS über sie laufen (das ist die Abschlussprüfung).

5.5 Zusammenfassung

Wir hoffen, dass diese Tour durch MongoDB Ihre Fantasie angeregt und verdeutlicht hat, warum sie sich einen Ruf als „gigantische" Datenbank erworben hat. Wir haben in einem einzelnen Kapitel sehr viel besprochen, doch wie immer konnten wir nur an der Oberfläche kratzen.

Mongos Stärken

Mongos primäre Stärke liegt in seiner Fähigkeit, gigantische Datenmengen (und riesige Mengen an Requests) durch Replikation und horizontale Skalierung verarbeiten zu können. Doch zusätzlich hat es den Vorteil eines sehr flexiblen Datenmodells, da Sie nicht an ein Schema gebunden sind und einfach beliebige Werte verschachteln können, die Sie in einem RDBMS mittels SQL-Join zusammenfassen würden.

Zu guter Letzt wurde MongoDB so entworfen, dass es einfach zu verwenden ist. Sie werden die Ähnlichkeit zwischen Mongo-Befehlen und SQL-Datenbank-Konzepten (außer den serverseitigen Joins) bemerkt haben. Das ist kein Zufall und einer der Gründe, warum Mongo soviel Aufmerksamkeit von ehemaligen ORM-Nutzern (objektrelationales Modell) erfährt. Es ist anders genug, um viele Entwickler zu reizen, aber nicht so anders, dass es zu einem furchterregenden Monster wird.

Mongos Schwächen

Wie Mongo die Denormalisierung eines Schemas erzwingt (indem es keines hat), ist vielleicht etwas schwer zu schlucken. Einige Entwickler finden die kalten, harten Beschränkungen relationaler Datenbanken beruhigend. Es kann gefährlich sein, einen alten Wert beliebigen Typs in eine Collection einzufügen. Ein einziger Schreibfehler kann für einige Kopfschmerzen sorgen, wenn Sie Feld- und Collection-Namen nicht als mögliche Fehlerquellen in

3. http://www.mongodb.org/display/DOCS/Geospatial+Indexing

Betracht ziehen. Mongos Flexibilität ist grundsätzlich nicht von Bedeutung, wenn Ihr Datenmodell ausgereift und vollständig ist.

Da sich Mongo auf große Datenmengen konzentriert, funktioniert es am besten in großen Clustern, deren Aufbau und Verwaltung einigen Aufwand erfordern. Im Gegensatz zu Riak, wo das Einfügen neuer Knoten transparent und relativ schmerzfrei erfolgt, verlangt die Einrichtung eines Mongo-Clusters etwas mehr Voraussicht.

Abschließende Gedanken

Mongo ist eine ausgezeichnete Wahl, wenn Sie Ihre Daten aus Gewohnheit mittels ORM in einer relationalen Datenbank speichern. Wir empfehlen es häufig für Rails-, Django- und MVC-Entwickler (Model-View-Controller), weil Validierung und Feld-Verwaltung über die Modelle auf Anwendungsebene erfolgen kann und weil Schema-Migration (größtenteils) der Vergangenheit angehört. Neue Felder in ein Dokument einzufügen ,ist so einfach wie das Einfügen eines neuen Feldes in Ihr Datemodell, und Mongo akzeptiert mit Freude neue Begriffe. Wir halten Mongo für eine wesentlich natürlichere Antwort auf viele gängige Problembereiche anwendungsgesteuerter Datenmengen als relationale Datenbanken.

KAPITEL 6

CouchDB

Ein Ratschenkasten ist ein leichtes und bequemes Werkzeug, das man immer dabei hat, um kleine oder große Jobs zu erledigen. Wie bei Bohrmaschinen kann man Stecknüsse und Schraubendreher verschiedener Formen und Größen aufsetzen. Im Gegensatz zu einer Bohrmaschine, die man an einer Steckdose anschließen muss, passt der Ratschenkasten aber in Ihre Tasche und wird mit Muskelkraft betrieben. Apache CouchDB ist genauso. Es kann wachsen und schrumpfen und passt sich Problembereichen variierender Größe und Komplexität mit Leichtigkeit an.

CouchDB ist die vollkommen JSON- und REST-basierte, dokumentenorientierte Datenbank. CouchDB wurde 2005 veröffentlicht und für das Web mit seinen zahllosen Mängeln, Fehlern, Ausfällen und Pannen entwickelt. Folglich bietet CouchDB eine Stabilität, die von den meisten anderen Datenbanken nicht erreicht wird. Während andere Systeme gelegentliche Netzwerkausfälle tolerieren, funktioniert CouchDB selbst dann noch, wenn eine Verbindung nur selten verfügbar ist.

Ähnlich wie MongoDB speichert CouchDB *Dokumente* – aus Schlüssel/Wert-Paaren bestehende JSON-Objekte, bei denen die Werte unterschiedliche Typen haben können, einschließlich beliebig tief verschachtelter weiterer Objekte. Allerdings gibt es keine Ad-hoc-Queries. Der grundsätzliche Weg, Dokumente zu finden, sind indexierte Views, die durch inkrementelles Mapreduce erzeugt wurden.

6.1 Relaxen auf der Couch

CouchDB macht seinem Motto „relaxen“ alle Ehre. Statt sich ausschließlich auf „Big Iron“-Cluster-Installationen zu konzentrieren, versucht CouchDB, eine Vielzahl von Einsatzszenarien (vom Rechenzentrum bis herunter zum

Smartphone) zu unterstützen. Sie können CouchDB auf Ihrem Android-Telefon, Ihrem MacBook und in Ihrem Rechenzentrum ausführen. In Erlang entwickelt, ist CouchDB kräftig gebaut – die einzige Möglichkeit, es herunterzufahren, besteht darin, den Prozess zu beenden! Durch sein Speichermodell, bei dem Daten nur angehängt werden können, sind Ihre Daten nahezu unzerstörbar und einfach zu replizieren, zu sichern und wiederherzustellen.

CouchDB ist dokumentenorientiert und nutzt JSON als Sprache zur Speicherung und Kommunikation. Wie bei Riak erfolgen alle Aufrufe von CouchDB über dessen REST-Schnittstelle. Die Replikation kann in eine oder beide Richtungen und ad hoc oder kontinuierlich erfolgen. CouchDB bietet eine sehr große Flexibilität bei der Strukturierung, Sicherung und Verteilung Ihrer Daten.

Vergleich zwischen CouchDB und MongoDB

Eine der großen Fragen, die wir in diesem Buch behandeln wollen, lautet „Was ist der Unterschied zwischen CouchDB und MongoDB?“ Oberflächlich betrachtet, scheinen CouchDB und MongoDB – das wir in Kapitel 5, *MongoDB*, auf Seite 147 behandelt haben – einander recht ähnlich zu sein. Beide sind dokumentenorientierte Datenspeicher mit einer Affinität für JavaScript, das JSON für den Datentransport nutzt. Doch es gibt viele Unterschiede, angefangen bei der Projekt-Philosophie bis hin zur Implementierung der Skalierungseigenschaften. Wir werden viele dieser Themen behandeln, wenn wir die wunderschöne Einfachheit von CouchDB erkunden.

Während unserer dreitägigen Tour werden wir viele verlockende Features und Design-Entscheidungen von CouchDB kennenlernen. Wir beginnen wie immer mit einzelnen CRUD-Befehlen und wenden uns dann der Indexierung über Mapreduce-Views zu. Wie bei anderen Datenbanken auch, importieren wir einige strukturierte Daten und nutzen sie dann, um fortgeschrittene Konzepte kennenzulernen. Abschließend entwickeln wir einige einfache eventgesteuerte Client-Anwendungen mit Node.js und lernen, wie CouchDBs Master/Master-Replikationsstrategie mit in Konflikt stehenden Updates umgeht. Gehen wir es an!

6.2 Tag 1: CRUD, Futon und cURL

Heute wollen wir unsere CouchDB-Erkundung mit CouchDBs freundlichem Web-Interface Futon beginnen und grundlegende CRUD-Operationen durchführen. Danach verwenden wir wieder cURL – das wir schon zur Kommunikation mit Riak in Kapitel 3, *Riak*, auf Seite 57 verwendet haben – , um REST-

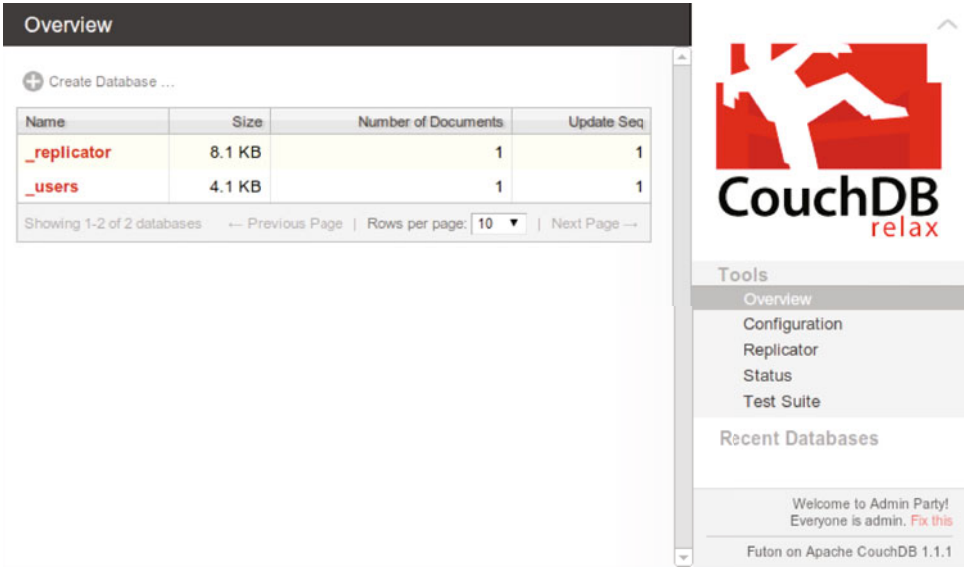


Abbildung 24: CouchDB Futon: Übersichtsseite

Aufrufe durchzuführen. Alle Bibliotheken und Treiber für CouchDB senden hinter den Kulissen REST-Requests. Es ist also sinnvoll, dass wir mit ihnen beginnen, um zu verstehen, wie sie funktionieren.

Es sich mit Futon gemütlich machen

CouchDB besitzt ein nützliches Webinterface namens Futon. Sobald CouchDB installiert ist und läuft, öffnen Sie `http://localhost:5984/_utils/` im Web-Browser. Sie landen dann auf der Übersichtsseite aus Abbildung 24, *CouchDB Futon: Übersichtsseite*.

Bevor wir anfangen können, mit Dokumenten zu arbeiten, müssen wir eine Datenbank anlegen, in der sie gespeichert werden. Wir wollen eine Datenbank mit Musikern anlegen, in der auch Album- und Titeldaten vorgehalten werden. Klicken Sie den Button Create Database... an. Im Popup geben Sie *music* ein und klicken auf Create. Damit werden Sie automatisch auf die Datenbankseite umgeleitet. Von da aus können wir neue Dokumente anlegen oder existierende öffnen.

Auf der Seite der Musikdatenbank klicken Sie auf den New-Document-Button. Sie landen auf einer neuen Seite, die aussieht wie in Abbildung 25, *CouchDB Futon: Ein Dokument anlegen*, auf Seite 196.

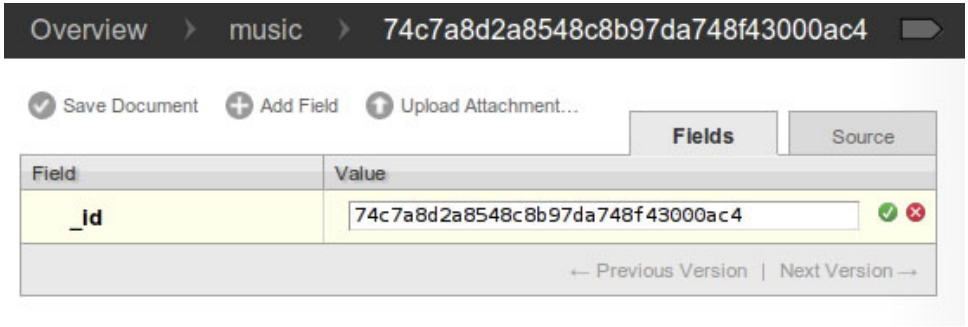


Abbildung 25: CouchDB Futon: Ein Dokument anlegen

Willkommen zur Admin-Party!

Ihnen wird bei Futon vielleicht die Warnung in der unteren rechten Ecke aufgefallen sein, nach der jeder Admin ist. Wäre dies ein Produktionsserver, würden Sie im nächsten Schritt „Fix this“ anklicken und einen Admin-Benutzer anlegen, um zu beschränken, wer was tun kann. In unserem Fall können wir es erst einmal so lassen, da es uns andere Aufgaben erleichtert.

Genau wie bei MongoDB besteht ein Dokument aus einem JSON-Objekt mit Schlüssel/Wert-Paaren (*Felder* genannt). Alle Dokumente in CouchDB besitzen ein `_id`-Feld, das einmalig sein muss und niemals geändert werden kann. Sie können eine `_id` explizit vergeben, doch wenn nicht, erzeugt CouchDB eine für Sie. In unserem Fall reicht die Voreinstellung, weshalb wir die Operation mit einem Klick auf Save Document abschließen.

Unmittelbar nach dem Speichern des Dokuments weist CouchDB ihm ein zusätzliches Feld namens `_rev` zu. Das `_rev`-Feld erhält immer dann einen neuen Wert, wenn das Dokument geändert wird. Das Format des Revision-Strings besteht aus einem Integerwert, gefolgt von einem Strich und einem pseudozufälligen eindeutigen String. Der Integerwert am Anfang gibt die numerische Revision an – in diesem Fall `1`.

Mit einem Unterstrich beginnende Feldnamen haben für CouchDB eine spezielle Bedeutung und `_id` und `_rev` sind besonders wichtig. Um ein existierendes Dokument zu aktualisieren oder zu löschen, müssen Sie *sowohl* eine `_id` als auch eine passende `_rev` angeben. Passen beide nicht zusammen, lehnt

CouchDB die Operation ab. Auf diese Weise werden Konflikte vermieden – indem sichergestellt wird, dass nur die aktuellen Dokumenten-Revisionen modifiziert werden.

Es gibt keine Transaktionen und kein Locking bei CouchDB. Um einen vorhandenen Datensatz zu modifizieren, lesen Sie ihn zuerst ein und notieren `_id` und `_rev`. Dann fordern Sie ein Update an und übergeben dabei das gesamte Dokument samt `_id` und `_rev`. Für alle Operationen gilt: Wer zuerst kommt, mahlt zuerst. Indem es eine passende `_rev` verlangt, stellt CouchDB sicher, dass das Dokument, das Sie ändern wollen, nicht hinter Ihrem Rücken verändert wurde.

In der geöffneten Dokumentenseite klicken Sie auf den Add-Field-Button. In der Field-Spalte geben Sie *name* ein und in der Value-Spalte *The Beatles*. Klicken Sie das grüne Häkchen an und dann den Save-Document-Button. Beachten Sie, dass das `_rev`-Feld jetzt mit `2` beginnt.

CouchDB ist nicht auf das Speichern von Strings beschränkt. Es kann JSON-Strukturen beliebiger Tiefe speichern. Klicken Sie erneut den Add-Field-Button an. Dieses Mal setzen Sie Field auf *albums* und als Value geben Sie die folgende (nicht vollständige Liste) ein:

```
[
  "Help!",
  "Sgt. Pepper's Lonely Hearts Club Band",
  "Abbey Road"
]
```

Nachdem Sie Save Document angeklickt haben, sollte es so aussehen wie in Abbildung 26, *CouchDB Futon: Dokument mit Array-Wert*, auf Seite 198.

Zu einem Album gibt es neben dem Namen noch weitere Informationen festzuhalten, also wollen wir sie hinzufügen. Modifizieren Sie das `albums`-Feld und ersetzen Sie den gerade gesetzten Wert durch Folgendes:

```
[{
  "title": "Help!",
  "year": 1965
},{
  "title": "Sgt. Pepper's Lonely Hearts Club Band",
  "year": 1967
},{
  "title": "Abbey Road",
  "year": 1969
}]
```

Nachdem Sie das Dokument gespeichert haben, sollten Sie im `albums`-Wert die verschachtelten Dokumente sehen können. Das sollte so aussehen wie

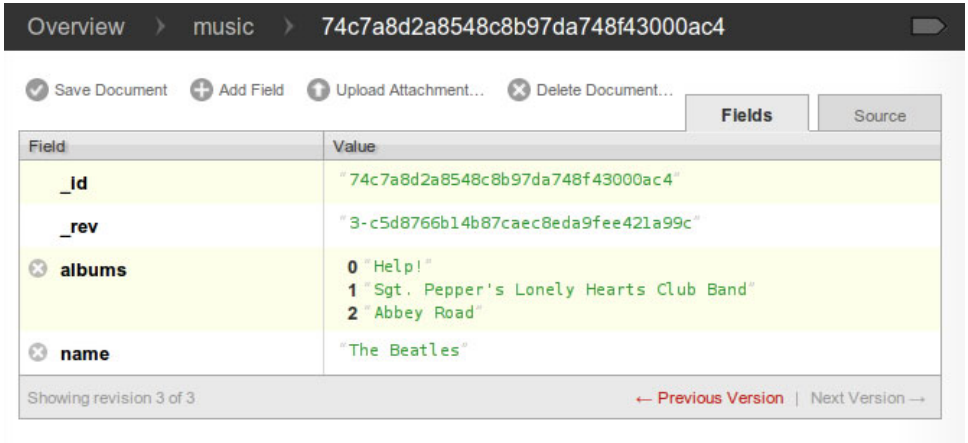


Abbildung 26: CouchDB Futon: Dokument mit Array-Wert

in Abbildung 27, *CouchDB Futon: Dokument mit tief verschachtelten Werten*, auf Seite 199.

Beim Anklicken den Delete-Document-Buttons passiert das, was Sie erwarten: Da Dokument wird aus der `music`-Datenbank gelöscht. Machen Sie das jetzt aber noch nicht. Stattdessen bewegen wir uns jetzt auf die Kommandozeile und sehen uns an, wie man mit CouchDB über REST kommuniziert.

REST-orientierte CRUD-Operationen mit cURL

Die gesamte Kommunikation mit CouchDB ist REST-basiert und das bedeutet, dass alle Befehle über HTTP laufen. CouchDB ist nicht die erste von uns diskutierte Datenbank mit dieser Eigenschaft. Riak – in Kapitel 3, *Riak*, auf Seite 57 diskutiert – nutzt ebenfalls REST für die gesamte Client-Kommunikation. Und genau wie bei Riak können wir mit CouchDB über das Kommandozeilen-Tool cURL kommunizieren.

Bevor wir uns dem Thema Views zuwenden, wollen wir einige einfache CRUD-Operationen durchführen. Als Einstieg öffnen wir eine Kommandozeile und führen den folgenden Befehl aus:

```
$ curl http://localhost:5984/
{"couchdb":"Welcome","version":"1.1.1"}
```

Mit `GET`-Requests (cURLs Standardeinstellung) werden Informationen über das in der URL angegebene „Ding“ zurückgegeben. Beim Zugriff auf die Wurzel (was wir gerade gemacht haben) werden Sie bloß darüber informiert, dass

Abbildung 27: CouchDB Futon: Dokument mit tief verschachtelten Werten

CouchDB läuft und welche Version installiert ist. Nun wollen wir einige Information über die vorhin von uns angelegte `music`-Datenbank abrufen (wir haben die Ausgabe der besseren Lesbarkeit halber formatiert):

```
$ curl http://localhost:5984/music/
{
 "db_name":"music",
 "doc_count":1,
 "doc_del_count":0,
 "update_seq":4,
 "purge_seq":0,
 "compact_running":false,
 "disk_size":16473,
 "instance_start_time":"1326845777510067",
 "disk_format_version":5,
 "committed_update_seq":4
}
```

Das gibt Informationen darüber zurück, wie viele Dokumente in der Datenbank enthalten sind, wie lange der Server schon läuft und wie viele Operationen durchgeführt wurden.

Ein Dokument über GET einlesen

Um ein bestimmtes Dokument abzurufen, hängen Sie seine `_id` an die Datenbank-URL an:

```
$curl http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000ac4
{
  "_id":"74c7a8d2a8548c8b97da748f43000ac4",
  "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
  "name":"The Beatles",
  "albums": [
    {
      "title":"Help!",
      "year":1965
    },{
      "title":"Sgt. Pepper's Lonely Hearts Club Band",
      "year":1967
    },{
      "title":"Abbey Road",
      "year":1969
    }
   ]
}
```

Bei CouchDB sind `GET`-Requests immer sicher. Durch ein `GET` wird bei CouchDB niemals ein Dokument verändert. Um Änderungen vorzunehmen, müssen Sie andere HTTP-Befehle wie `PUT`, `POST` und `DELETE` verwenden.

Ein Dokument mit POST erzeugen

Um ein neues Dokument anzulegen, verwenden Sie `POST`. Dabei müssen Sie einen Content-Type-Header mit dem Wert *application/json* angeben, weil CouchDB den Request sonst ablehnt.

```
$ curl -i -X POST "http://localhost:5984/music/" \
 -H "Content-Type: application/json" \
 -d '{ "name": "Wings" }'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b
Date: Wed, 18 Jan 2012 00:37:51 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
 "ok":true,
 "id":"74c7a8d2a8548c8b97da748f43000f1b",
 "rev":"1-2fe1dd1911153eb9df8460747dfe75a0"
}
```

Der HTTP-Response-Code `201 Created` teilt uns mit, dass das Anlegen erfolgreich war. Der Body der Response enthält ein JSON-Objekt mit nützlichen Informationen wie die `_id`- und `_rev`-Werte.

Ein Dokument mit PUT aktualisieren

Der PUT-Befehl wird genutzt, um ein vorhandenes Dokument zu aktualisieren oder um ein neues Dokument mit einer bestimmten _id anzulegen. Genau wie bei GET besteht die URL für ein PUT aus der Datenbank-URL gefolgt von der _id des Dokuments.

```
$ curl -i -X PUT \
 "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
 -H "Content-Type: application/json" \
 -d '{
 "_id": "74c7a8d2a8548c8b97da748f43000f1b",
 "_rev": "1-2fe1dd1911153eb9df8460747dfe75a0",
 "name": "Wings",
 "albums": ["Wild Life", "Band on the Run", "London Town"]
 }'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b
Etag: "2-17e4ce41cd33d6a38f04a8452d5a860b"
Date: Wed, 18 Jan 2012 00:43:39 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"2-17e4ce41cd33d6a38f04a8452d5a860b"
}
```

Im Gegensatz zu MongoDB, wo Dokumente vor Ort (*in place*) modifiziert werden, überschreiben Sie bei CouchDB immer das gesamte Dokument, das Sie ändern wollen. Im Futon Web-Interface von vorhin sah es vielleicht so aus, als würde man ein einzelnes Feld für sich aktualisieren, doch hinter den Kulissen wurde beim Klick aus Save das gesamte Dokument neu geschrieben.

Wie vorhin erwähnt, müssen sowohl _id als auch _rev genau mit dem zu aktualisierenden Dokument übereinstimmen, sonst schlägt die Operation fehl. Um zu sehen, wie, führen Sie die noch einmal die gleiche PUT-Operation aus.

```
HTTP/1.1 409 Conflict
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Date: Wed, 18 Jan 2012 00:44:12 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 58
Cache-Control: must-revalidate

{"error":"conflict","reason":"Document update conflict."}
```

Sie erhalten als Antwort einen HTTP 409 Conflict zusammen mit einem JSON-Objekt, das das Problem beschreibt. Auf diese Weise stellt CouchDB die Konsistenz sicher.

Ein Dokument mit DELETE löschen

Zum Schluss können wir die DELETE-Operation nutzen, um ein Dokument aus der Datenbank zu entfernen.

```
$ curl -i -X DELETE \
 "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
 -H "If-Match: 2-17e4ce41cd33d6a38f04a8452d5a860b"
HTTP/1.1 200 OK
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Etag: "3-42aafb7411c092614ce7c9f4ab79dc8b"
Date: Wed, 18 Jan 2012 00:45:36 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"3-42aafb7411c092614ce7c9f4ab79dc8b"
 }
```

Die DELETE-Operation gibt eine neue Revisionsnummer zurück, obwohl das Dokument gelöscht wurde. Tatsächlich wurde das Dokument nicht von der Festplatte entfernt, sondern es wurde nur ein neues leeres Dokument angehangen, das das Dokument als gelöscht markiert. Genau wie bei einem Update modifiziert CouchDB Dokumente nicht vor Ort. Doch nach außen hin ist es gelöscht.

Was wir am ersten Tag gelernt haben

Nachdem wir gelernt haben, grundlegende CRUD-Operationen mit Futon und cURL durchzuführen, sind wir bereit, uns fortgeschrittenen Themen zuzuwenden. Am zweiten Tag tauchen wir in den Aufbau indexierter *Views* ein. Mit ihrer Hilfe eröffnen sich uns andere Möglichkeiten, Dokumente abzurufen, und wir sind nicht allein auf deren _id-Werte beschränkt.

Tag 1: Selbststudium

Finden Sie heraus

1. Finden Sie die Online-Dokumentation zur CouchDB HTTP Document API.
2. Wir haben bereits GET, POST, PUT und DELETE verwendet. Welche anderen HTTP-Befehle werden unterstützt?

Machen Sie Folgendes

1. Nutzen Sie cURL, um ein neues Dokument per PUT in die Musikdatenbank einzufügen. Verwenden Sie dazu eine _id Ihrer Wahl.

2. Verwenden Sie `curl`, um eine neue Datenbank mit einem Namen Ihrer Wahl anzulegen. Löschen Sie die Datenbank dann wieder, ebenfalls mit cURL.
3. Nutzen Sie erneut cURL, um ein neues Dokument zu erzeugen, das ein Textdokument als Anhang enthält. Entwickeln Sie dann einen cURL-Request, der nur diesen Anhang abruft.

6.3 Tag 2: Views erzeugen und abfragen

Bei CouchDB ist ein *View* ein Fenster in die in einer Datenbank enthaltenen Dokumente. Views sind der übliche Weg, auf Dokumente zuzugreifen, außer bei ganz trivialen Fällen wie den einzelnen CRUD-Operationen, die wir am ersten Tag gesehen haben. Heute wollen wir zeigen, wie man die Funktionen entwickelt, aus denen sich ein View zusammensetzt. Wir werden auch lernen, wie man mit cURL Ad-hoc-Queries über Views ausführt. Zum Schluss werden wir Musikdaten importieren, um den Views etwas mehr Substanz zu verleihen und um Ihnen zu demonstrieren, wie man couchrest nutzt, eine beliebte Ruby-Bibliothek für die Arbeit mit CouchDB.

Über Views auf Dokumente zugreifen

Ein Mapper besteht aus Mapper- und Reducer-Funktionen, die eine sortierte Liste von Schlüssel/Wert-Paaren generieren. Sowohl Schlüssel als auch Wert können gültiges JSON sein. Der einfachste View heißt `_all_docs`. Er ist standardmäßig für alle Datenbanken verfügbar und enthält einen Eintrag für jedes Dokument der Datenbank, wobei die `_id` den Schlüssel bildet.

Um alle „Dinge“ in der Datenbank abzurufen, stoßen Sie einen `GET`-Request für den `_all_docs`-View an.

```
$ curl http://localhost:5984/music/_all_docs
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    }
  }]
}
```

In der obigen Ausgabe sehen Sie das einzige Dokument, das wir bisher angelegt haben. Die Antwort ist ein JSON-Objekt, das ein Array von `rows` (Zeilen)

enthält. Jede Zeile ist ein Objekt mit drei Feldern:

- `id` ist die `_id` des Dokuments.
- `key` ist der von den Mapreduce-Funktionen erzeugte JSON-Schlüssel.
- `value` ist der dazugehörige JSON-Wert, ebenfalls per Mapreduce erzeugt.

Bei `_all_docs` stimmen die `id`- und `key`-Felder überein, doch bei eigenen Views wird das so gut wie niemals der Fall sein.

Standardmäßig geben Views nicht den Inhalt der Dokumente im `value` zurück. Um alle Felder im Dokument abzurufen, müssen Sie den URL-Parameter `include_docs=true` anhängen.

```
$ curl http://localhost:5984/music/_all_docs?include_docs=true
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    },
    "doc":{
      "_id":"74c7a8d2a8548c8b97da748f43000ac4",
      "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
      "name":"The Beatles",
      "albums":[{
          "title":"Help!",
          "year":1965
        },{
          "title":"Sgt. Pepper's Lonely Hearts Club Band",
          "year":1967
        },{
          "title":"Abbey Road",
          "year":1969
      }]
    }
  }]
}
```

Hier sehen Sie, dass die anderen Properties `name` und `albums` in der Ausgabe in das `value`-Objekt eingefügt wurden. Mit dieser grundlegenden Struktur im Hinterkopf wollen wir eigene Views entwickeln.

Ihr erster View

Da wir nun eine grobe Vorstellung davon haben, wie Views funktionieren, wollen wir unsere eigenen Views entwickeln. Als Einstieg wollen wir das Verhalten des `_all_docs`-Views nachbilden. Danach werden wir immer komplexere Views entwickeln, um tieferliegende Informationen aus unseren Dokumenten für die Indexierung zu extrahieren.

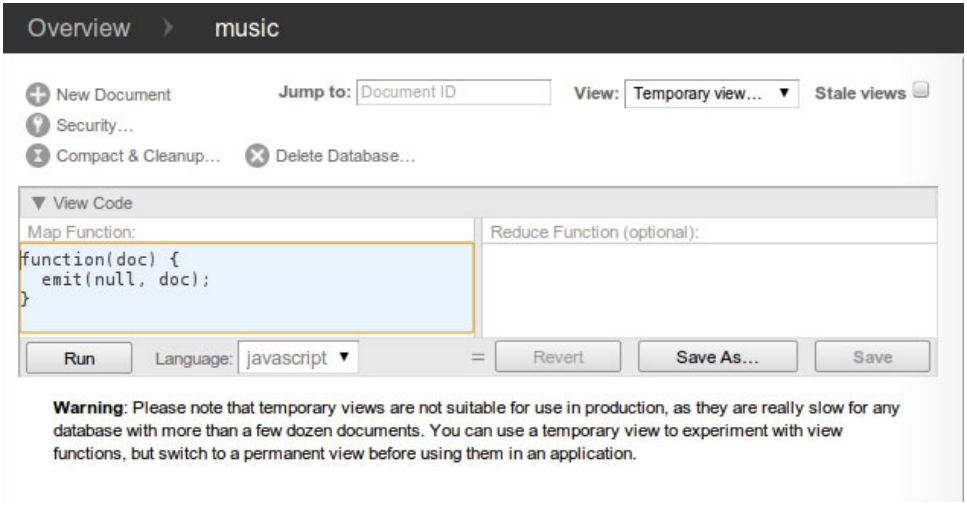


Abbildung 28: CouchDB Futon: temporärer View

Um einen temporären View auszuführen, öffnen Sie Futon in einem Browser[1], so wie wir das am ersten Tag getan haben. Als Nächstes öffnen Sie die `music`-Datenbank, indem Sie den entsprechenden Link anklicken. In der oberen rechten Ecke der Musikdatenbank-Seite wählen Sie „Temporary view…“ aus dem View-Dropdown. Sie sollten eine Seite sehen wie in Abbildung 28, *CouchDB Futon: temporärer View*.

Der Code im linken Map Function-Feld muss wie folgt aussehen:

```
function(doc) {
 emit(null, doc);
}
```

Wenn Sie den Run-Button unter der map-Funktion anklicken, führt CouchDB diese Funktion für jedes Dokument in der Datenbank aus und übergibt dieses Dokument dabei im `doc`-Parameter an diese Funktion. Das erzeugt eine Tabelle mit einer einzigen Ergebnis-Zeile, die wie folgt aussieht:

Schlüssel	Wert
`null`	`{_id: "74c7a8d2a8548c8b97da748f43000ac4", _rev: "4-93a101178ba65f61ed39e60d70c9fd97", name: "The Beatles", albums: [{title: "Help!", year: 1965}, {title: "Sgt. Pepper's Lonely Hearts Club Band", year: 1967}, {title: "Abbey Road", year: 1969}]}`

1. http://localhost:5984/_utils/

Das Geheimnis dieser Ausgabe (und aller Views) ist die `emit`-Funktion (die genau so funktioniert wie die MongoDB-Funktion gleichen Namens). `emit` verlangt zwei Argumente: den Schlüssel und den Wert. Eine gegebene map-Funktion kann emit für ein gegebenes Dokument einmal, mehrmals oder auch gar nicht aufrufen. Im obigen Beispiel emittiert die map-Funktion das Schlüssel/Wert-Paar `null/doc`. Wie in der Ausgabetabelle zu sehen, ist der Schlüssel tatsächlich `null` und der Wert entspricht dem Objekt, das wir am ersten Tag gesehen haben, als wir es direkt über cURL angefordert haben.

Um einen Mapper zu entwickeln, der das Gleiche macht wie `_all_docs`, müssen wir aber etwas anderes emittieren. Erinnern Sie sich daran, dass `_all_docs` das `_id`-Feld des Dokuments als Schlüssel ausgibt und ein einfaches Objekt, das nur das `_rev`-Feld enthält, als Wert. Dementsprechend ändern wir den Code unserer map-Funktion wie folgt ab und klicken auf Run.

```
function(doc) {
 emit(doc._id, { rev: doc._rev });
}
```

Die Ausgabetabelle sollte nun wie folgt aussehen und die gleichen Schlüssel/Wert-Paare enthalten, die wir vorhin bei `_all_docs` gesehen haben:

Schlüssel	Wert
`"74c7a8d2a8548c8b97da748f43000ac4"`	`{rev: "4-93a101178ba65f61ed39e60d70c9fd97"}`

Beachten Sie, dass wir nicht mit Futon arbeiten müssen, um temporäre Views auszuführen. Sie können auch einen `POST`-Request an den `_temp_view`-Handler senden. In diesem Fall übergeben Sie Ihre map-Funktion als JSON-Objekt im Request-Body.

```
$ curl -X POST \
 http://localhost:5984/music/_temp_view \
 -H "Content-Type: application/json" \
 -d '{"map":"function(doc){emit(doc._id,{rev:doc._rev});}"}'
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    }
  }]
}
```

Die Antwort entspricht dem, was wir von `_all_docs` erwarten. Doch was passiert, wenn wir den Parameter `include_docs=true` angeben? Finden wir es heraus!

```
$ curl -X POST \
 http://localhost:5984/music/_temp_view?include_docs=true \
 -H "Content-Type: application/json" \
 -d '{"map":"function(doc){emit(doc._id,{rev:doc._rev});}"}'
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    },
    "doc":{
      "_id":"74c7a8d2a8548c8b97da748f43000ac4",
      "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
      "name":"The Beatles",
      "albums":[...]
    }
  }]
}
```

Statt zusätzliche Felder in das `value`-Objekt zu integrieren, wird diesmal eine separate Property namens `doc` in die Zeile eingefügt, die das gesamte Dokument enthält.

Ein eigener View kann jeden beliebigen Wert emittieren, auch `null`. Die Bereitstellung einer separaten `doc`-Property vermeidet Probleme, die auftreten könnten, wenn man die Zeilenwerte mit dem Dokument kombiniert. Als Nächstes wollen wir uns ansehen, wie man einen View speichert, damit CouchDB die Ergebnisse indexieren kann.

Einen View als Design-Dokument speichern

Führt CouchDB einen temporären View aus, muss es die bereitgestellte map-Funktion für *jedes Dokument* in der Datenbank ausführen. Das ist extrem Ressourcen-intensiv, verbraucht viel Rechenleistung und ist langsam. Sie sollten temporäre Views nur für Entwicklungszwecke nutzen. Für den Produktiveinsatz sollten Sie Ihre Views in sog. *Design-Dokumenten* speichern.

Ein Design-Dokument ist ein reales Dokument innerhalb der Datenbank, genau wie das eben von uns angelegte Beatles-Dokument. Als solches kann es in der üblichen Art und Weise in Views erscheinen und auf andere CouchDB-Server repliziert werden. Um einen temporären View in Futon als Design-Dokument zu speichern, klicken Sie den Save As...-Button an und füllen dann die Design-Document- und View-Name-Felder aus.

Design-Dokumente haben immer IDs, die mit `_design/` beginnen und einen oder mehrere Views enthalten. Der Name des Views unterscheidet ihn von den anderen Views, die im gleichen Dokument enthalten sind. Die Entscheidung, welche Views in welche Design-Dokumente gehören, ist größtenteils anwendungsspezifisch und eine Frage des Geschmacks. Die allgemeine Regel lautet, Views basierend auf dem zu gruppieren, was sie mit den Daten anstellen. Wir werden entsprechende Beispiele sehen, wenn wir interessantere Views entwickeln.

Künstler nach Namen finden

Nachdem wir die Grundlagen der View-Entwicklung kennen, wollen wir einige anwendungsspezifische Views entwickeln. Wie Sie wissen, enthält unsere `music`-Datenbank Informationen über die Künstler, darunter auch ein `name`-Feld mit dem Namen der Band. Über ein normales `GET` oder den `_all_docs`-View können wir auf die Dokumente über ihre `_id`-Werte zugreifen, doch wir wollen Bands über ihren Namen abrufen.

Mit anderen Worten können wir bis jetzt das Dokument mit der `_id` `74c7a8d2a8548c8b97da748f43000ac4` nachschlagen, doch wie finden wir das Dokument, bei dem der `name` den Wert *The Beatles* enthält? Dafür benötigen wir einen View. Wechseln Sie in Futon zurück zur Temporary-View-Seite, geben Sie die folgende map-Funktion ein und klicken Sie auf Run.

couchdb/artists_by_name_mapper.js

```
function(doc) {
  if ('name' in doc) {
    emit(doc.name, doc._id);
  }
}
```

Diese Funktion prüft, ob das aktuelle Dokument ein `name`-Feld enthält und wenn das der Fall ist, emittiert es den Namen und die Dokumenten-`_id` als relevantes Schlüssel/Wert-Paar und erzeugt eine Tabelle wie diese:

Schlüssel	Wert
`"The Beatles"`	`"74c7a8d2a8548c8b97da748f43000ac4"`

Klicken Sie auf den Save-As...-Button und tragen Sie dann für Design Document *artists* und für View Name *by_name* ein. Klicken Sie auf Save, um Ihre Eingabe zu sichern.

Alben nach Namen finden

Einen Künstler über den Namen zu finden, ist nützlich, aber wir können noch mehr. Diesmal wollen wir einen View entwickeln, der uns Alben finden lässt. Das ist das erste Beispiel, bei dem die map-Funktion mehr als ein Ergebnis pro Dokument emittieren kann.

Wechseln Sie wieder auf die Temporary-View-Seite und geben Sie den folgenden Mapper ein:

couchdb/albums_by_name_mapper.js

```
function(doc) {
  if ('name' in doc && 'albums' in doc) {
    doc.albums.forEach(function(album){
      var
        key = album.title || album.name,
        value = { by: doc.name, album: album };
      emit(key, value);
    });
  }
}
```

Die Funktion überprüft, ob das aktuelle Dokument ein `name`- und ein `albums`-Feld enthält. Ist das der Fall, emittiert sie ein Schlüssel/Wert-Paar für jedes Album, bei dem der Schlüssel der Titel oder Name des Albums ist und der Wert ein zusammengesetztes Objekt mit dem Namen des Künstlers und dem ursprünglichen Album-Objekt. Die dabei erzeugte Tabelle sieht etwa so aus:

Schlüssel	Wert
`"Abbey Road"`	`{by: "The Beatles", album: {title: "Abbey Road", year: 1969}}`
`"Help!"`	`{by: "The Beatles", album: {title: "Help!", year: 1965}}`
`"Sgt. Pepper's Lonely Hearts Club Band"`	`{by: "The Beatles", album: {title: "Sgt. Pepper's Lonely Hearts Club Band", year: 1967}}`

Genau wie bei unserem Künstler-nach-Name-View klicken wir den Save-As...-Button. Diesmal nennen wir das Design-Dokument *albums* und den View *by_name*. Wir klicken wieder Save an, um die Änderung zu speichern. Nun wollen wir die Dokumente abfragen.

Abfragen mit unseren selbst entwickelten Artist- und Album-Views

Nachdem wir eine Reihe eigener Design-Dokumente gespeichert haben, kehren wir zur Kommandozeile zurück und fragen sie mit dem `curl`-Befehl ab. Wir beginnen mit dem Künstler-nach-Name-View. In der Kommandozeile führen Sie Folgendes aus:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"The Beatles",
    "value":"74c7a8d2a8548c8b97da748f43000ac4"
  }]
}
```

Um einen View abzufragen, konstruieren Sie einen Pfad der Form `/<datenbank_name>/_design/<design_dok>/_view/ <view_name>`. In unserem Beispiel fragen wir den `by_name`-View im Design-Dokument `artists` der Datenbank `music` ab. Wie zu erwarten, enthält die Ausgabe unser einziges Dokument, mit dem Bandnamen als Schlüssel.

Als Nächstes wollen wir Alben nach Namen finden:

```
$ curl http://localhost:5984/music/_design/albums/_view/by_name
{
  "total_rows":3,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Abbey Road",
    "value":{
      "by":"The Beatles",
      "album":{
        "title":"Abbey Road",
        "year":1969
      }
    }
  },{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Help!",
    "value":{
      "by":"The Beatles",
      "album":{
        "title":"Help!",
        "year":1965
      }
    }
  },{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Sgt. Pepper's Lonely Hearts Club Band",
```

```
    "value":{
      "by":"The Beatles",
      "album":{
        "title":"Sgt. Pepper's Lonely Hearts Club Band",
        "year":1967
      }
    }
  }]
}
```

CouchDB stellt sicher, dass die Datensätze alphabetisch sortiert nach den emittierten Schlüsseln zurückgegeben werden. In der Tat ist das die Indexierung, die CouchDB bietet. Beim Entwurf von Views ist es wichtig, Schlüssel zu emittieren, die sortiert einen Sinn ergeben. Einen View auf diese Weise abzurufen liefert die gesamte Datenmenge zurück, doch was tun, wenn man nur eine Teilmenge will? Eine Möglichkeit ist der URL-Parameter `key`. Wenn Sie einen Schlüssel angeben, werden nur Zeilen mit exakt diesem Schlüssel zurückgegeben.

```
$ curl 'http://localhost:5984/music/_design/albums/_view/by_name?key="Help!"'
{
  "total_rows":3,
  "offset":1,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Help!",
    "value":{
      "by":"The Beatles",
      "album":{"title":"Help!","year":1965}
    }
  }]
}
```

Beachten Sie die `total_rows`- und `offset`-Felder in der Antwort. Das `total_rows`-Feld enthält die Gesamtmenge der Datensätze im View, nicht nur die Anzahl der in diesem Request zurückgegebenen Teilmenge. Das `offset`-Feld sagt uns, an welcher Stelle in der Gesamtmenge der erste zurückgelieferte Datensatz liegt. Basierend auf diesen beiden Zahlen und der Länge der `rows`, können wir berechnen, wie viele weitere Datensätze sich noch zu beiden Seiten des Views befinden.

View-Requests können neben `keys` noch andere Parameter nutzen, doch um sie in Aktion zu sehen, benötigen wir etwas mehr Daten.

Daten per Ruby in CouchDB importieren

Der Import von Daten ist ein Problem, vor dem man immer wieder steht, egal, welche Datenbank man letztlich nutzt. CouchDB ist da keine Ausnahme. In diesem Abschnitt wollen wir Ruby verwenden, um strukturierte Daten in un-

sere `music`-Datenbank zu importieren. Wir zeigen, wie man große Datenmengen in CouchDB importiert und Sie haben einen schönen Datenbestand, mit dem Sie arbeiten können, wenn wir fortgeschrittene Views entwickeln.

Wir nutzen die Musikdaten von Jamendo.com,[2] eine Site, die freie Musik hostet. Jamendo stellt alle Künstler-, Album- und Titel-Daten in einem strukturierten XML-Format zur Verfügung, was für den Import in eine dokumentenorientierte Datenbank wie CouchDB ideal ist.

Wechseln Sie auf Jamendos NewDatabaseDumps-Seite[3] und laden Sie `dbdump_ artistalbumtrack.xml.gz` herunter.[4] Die gezippte Datei ist nur etwa 15MB groß. Zum Parsing von Jamendos XML-Datei nutzen wir das `libxml-ruby`-Gem.

Statt unseren eigenen Ruby-CouchDB-Treiber zu schreiben oder HTTP-Requests direkt anzugeben, nutzen wir ein beliebtes Ruby-Gem namens `couchrest`, das diese Aufrufe in eine komfortable Ruby-API packt. Wir werden nur einige wenige Methoden der API nutzen, doch die Dokumentation ist recht gut, falls Sie den Treiber für eigene Projekte nutzen wollen.[5]

Über die Kommandozeile installieren wir die benötigten Gems:

```
$ gem install libxml-ruby couchrest
```

Genau wie bei den Wikipedia-Daten in Kapitel 4, *HBase*, auf Seite 103 nutzen wir einen SAX-Parser, um Dokumente in die Datenbank einzufügen, die sequentiell verarbeitet werden, während sie über die Standardeingabe hereinkommen. Hier der Code:

code/couchdb/import_from_jamendo.rb

```
① require 'rubygems'
  require 'libxml'
  require 'couchrest'

  include LibXML


② class JamendoCallbacks
    include XML::SaxParser::Callbacks

③   def initialize()
      @db = CouchRest.database!("http://localhost:5984/music")
      @count = 0
      @max = 100 # Wie viele Datensätze sollen importiert werden?
      @stack = []
```

2. http://www.jamendo.com/
3. http://developer.jamendo.com/en/wiki/NewDatabaseDumps
4. http://img.jamendo.com/data/dbdump_artistalbumtrack.xml.gz
5. http://rdoc.info/github/couchrest/couchrest/master/

```
    @artist = nil
    @album = nil
    @track = nil
    @tag = nil
    @buffer = nil
  end

④ def on_start_element(element, attributes)
    case element
    when 'artist'
      @artist = { :albums => [] }
      @stack.push @artist
    when 'album'
      @album = { :tracks => [] }
      @artist[:albums].push @album
      @stack.push @album
    when 'track'
      @track = { :tags => [] }
      @album[:tracks].push @track
      @stack.push @track
    when 'tag'
      @tag = {}
      @track[:tags].push @tag
      @stack.push @tag
    when 'Artists', 'Albums', 'Tracks', 'Tags'
      # ignore
    else
      @buffer = []
    end
  end

⑤ def on_characters(chars)
    @buffer << chars unless @buffer.nil?
  end

⑥ def on_end_element(element)
    case element
    when 'artist'
      @stack.pop
      @artist['_id'] = @artist['id'] # Jamendos artist-id als doc-_id verwenden
      @artist[:random] = rand
      @db.save_doc(@artist, false, true)
      @count += 1
      if !@max.nil? && @count >= @max
        on_end_document
      end
      if @count % 500 == 0
        puts " #{@count} records inserted"
      end
    when 'album', 'track', 'tag'
      top = @stack.pop
      top[:random] = rand
    when 'Artists', 'Albums', 'Tracks', 'Tags'
      # ignorieren
    else
      if @stack[-1] && @buffer
        @stack[-1][element] = @buffer.join.force_encoding('utf-8')
        @buffer = nil
      end
    end
  end
```

```
    def on_end_document()
      puts "TOTAL: #{@count} records inserted"
      exit(1)
    end
  end

⑦ parser = XML::SaxParser.io(ARGF)
  parser.callbacks = JamendoCallbacks.new
  parser.parse
```

① Zum Auftakt laden wir das `rubygems`-Modul und die benötigten Gems.

② Standardmäßig nutzt man LibXML, indem man eine Callback-Klasse definiert. Wir definieren eine `JamendoCallbacks`-Klasse, die unsere SAX-Handler für verschiedene Events kapselt.

③ Während der Initialisierung stellt unsere Klasse mit Hilfe der CourchRest-API zuerst die Verbindung mit unserem lokalen CouchDB-Server her und erzeugt dann die `music`-Datenbank (wenn sie nicht bereits existiert). Danach werden einige Instanzvariablen eingerichtet, die Statusinformationen während des Parsings speichern. Beachten Sie, dass alle Dokumente (und nicht nur die ersten 100) importiert werden, wenn Sie den `@max`-Parameter auf `nil` setzen.

④ Sobald das Parsing begonnen hat, verarbeitet die `on_start_element()`-Methode alle öffnenden Tags. Hier achten wir auf uns besonders interessierende Tags wie `<artist>`, `<album>`, `<track>` und `<tag>`. Wir ignorieren bewusst Container-Elemente – `<Artists>`, `<Albums>`, `<Tracks>` und `<Tags>` – und behandeln alle anderen als Properties, die für die nächstgelegenen Container-Elemente gesetzt werden.

⑤ Sobald der Parser Zeichendaten erkennt, puffern wir sie, um sie dem aktuellen Container-Element (dem Ende des `@stacks`) als Property hinzuzufügen.

⑥ Ein Großteil der interessanten Dinge passiert in der `on_end_element()`-Methode. Hier schließen wir das aktuelle Container-Element, indem wir es vom Stack entfernen. Schließt der Tag ein `artist`-Element, nutzen wir die Gelegenheit, um das Dokument mit der `@db.save_doc()`-Methode in CouchDB abzuspeichern. Jedem Container-Element fügen wir außerdem eine `random`-Property hinzu, die eine frisch generierte Zufallszahl enthält. Wir nutzen diesen Wert später, um einen Titel, ein Album oder einen Künstler zufällig auszuwählen.

⑦ Rubys `ARGF`-Stream kombiniert die Standardeingabe und alle in der Kommandozeile angegebenen Dateien. Wir übergeben das an LibXML und geben eine Instanz unserer `JamendoCallbacks`-Klasse an, um die Tokens – Start-Tags, End-Tags und Zeichendaten – zu verarbeiten.

Wir lassen das Skript laufen, indem wir den entpackten XML-Inhalt über eine Pipe an das Import-Skript übergeben:

```
$ zcat dbdump_artistalbumtrack.xml.gz | ruby import_from_jamendo.rb
TOTAL: 100 records inserted
```

Sobald der Import abgeschlossen ist, wechseln wir wieder in die Kommandozeile und sehen uns unseren View an. Zuerst rufen wir einige Künstler ab. Der URL-Parameter `limit` legt fest, dass wir maximal diese Zahl (oder weniger) von Dokumenten in der Response zurückerhalten wollen.

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?limit=5
{"total_rows":100,"offset":0,"rows":[
{"id":"370255","key":"\"\"ATTIC\"\"","value":"370255"},
{"id":"353262","key":"10centSunday","value":"353262"},
{"id":"367150","key":"abdielyromero","value":"367150"},
{"id":"276","key":"AdHoc","value":"276"},
{"id":"364713","key":"Adversus","value":"364713"}
]}
```

Der obige Request hat beim Anfang der Künstlerliste begonnen. Um mittendrin zu beginnen, können wir den Parameter `startkey` verwenden:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
limit=5\&startkey=%22C%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"364714","key":"Daringer","value":"364714"}
]}
```

Die obige Abfrage beginnt mit Künstlern, deren Namen mit *C* beginnen. Durch Angabe eines `endkey`s können Sie das zurückgelieferte Ergebnis weiter einschränken. Nachfolgend lassen wir uns nur die Künstler zwischen *C* und *D* zurückgeben:

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
startkey=%22C%22\&endkey=%22D%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"}
]}
```

Um die Reihenfolge der Zeilen umzukehren, verwenden wir den URL-Parameter `descending`. Dabei müssen wir aber auch `startkey` und `endkey` umkehren.

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
startkey=%22D%22\&endkey=%22C%22\&descending=true
{"total_rows":100,"offset":16,"rows":[
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"340296","key":"CalexB","value":"340296"}
]}
```

Eine Reihe weiterer URL-Parameter steht zur Modifikation von View-Requests zur Verfügung, doch das sind die gängigsten und diejenigen, die Sie am häufigsten nutzen werden. Einige der URL-Parameter haben mit der Gruppierung zu tun, die vom Reduce-Teil der CouchDB Mapreduce-Views kommt. Wir sehen sie uns morgen an.

Was wir am zweiten Tag gelernt haben

Heute haben wir so einiges gelernt. Wir wissen nun, wie man einfache Views in CouchDB anlegt und in Design-Dokumenten speichert. Wir haben gesehen, wie man Views auf unterschiedliche Art und Weise abfragt, um Teilmengen indexierter Inhalte abzurufen. Mit Ruby und einem beliebten Gem namens `couchrest` haben wir strukturierte Daten importiert und zur Unterstützung unserer Views genutzt. Morgen werden wir auf diesen Ideen aufbauen und fortgeschrittene Views mit Reducern entwickeln, um uns dann anderen APIs zu widmen, die CouchDB unterstützt.

Tag 2: Selbststudium

Finden Sie heraus

1. Wir haben gesehen, dass die `emit()`-Methode Schlüssel ausgeben kann, bei denen es sich um Strings handelt. Welche anderen Typen werden unterstützt? Was passiert, wenn Sie ein Array mit Werten als Schlüssel emittieren?
2. Finden Sie eine Liste der verfügbaren URL-Parameter (wie `limit` und `startkey`), die an View-Requests angehängt werden können. Finden Sie heraus, was sie machen.

Machen Sie Folgendes

1. Unser Import-Skript `import_from_jamendo.rb` weist jedem Künstler eine Zufallszahl zu, indem es eine Property namens `random` einfügt. Entwickeln Sie eine Mapper-Funktion, die Schlüssel/Wert-Paare zurückgibt, bei denen die Zufallszahl der Schlüssel und der Bandname der Wert ist. Speichern Sie diese Funktion in einem neuen Design-Dokument namens `_design/random` und dem View-Namen `artist`.

2. Entwickeln Sie einen cURL-Request, der einen zufälligen Künstler zurückliefert.

   Tipp: Sie müssen den `startkey`-Parameter verwenden und können in der Kommandozeile eine Zufallszahl mit `‘ruby -e 'puts rand'‘ erzeugen`.

3. Das Import-Skript fügt diese `random`-Property auch für jedes Album, jeden Titel und jeden Tag ein. Entwickeln Sie drei zusätzliche Views im Design-Dokument `_design/random` mit den View-Namen `album`, `track` und `tag`, die genauso funktionieren, wie unser `artist`-View.

6.4 Tag 3: Fortgeschrittene Views, Changes-API und Datenreplikation

An den ersten beiden Tagen haben wir gelernt, wie man einfache CRUD-Operationen durchführt und mit Views interagiert, um Daten aufzuspüren. Auf dieser Erfahrung aufbauend wollen wir uns heute Views genauer ansehen und den Reduce-Teil der Mapreduce-Gleichung sezieren. Danach werden wir einige Node.js-Anwendungen in JavaScript entwickeln, um CouchDBs einmalige Changes-API zu unterstützen. Abschließend diskutieren wir die Replikation und wie CouchDB mit Konflikten umgeht.

Fortgeschrittene Views mit Reducern entwickeln

Mapreduce-basierte Views sind das Instrument, mit dem Sie sich CouchDBs Indexierungs- und Aggregations-Fähigkeiten zunutze machen können. Am zweiten Tag haben all unsere Views nur aus Mappern bestanden. Nun wollen wir auch Reducer nutzen, um neue Fähigkeiten zu entwickeln, die wir auf die am zweiten Tag importierten Jamendo-Daten anwenden wollen.

Eine schöne Sache an den Jamendo-Daten ist ihre Tiefe. Künstler haben Alben, auf denen sich Titel befinden. Titel haben wiederum Attribute, einschließlich Tags. Wir wenden unsere Aufmerksamkeit nun diesen Tags zu und wollen einen tiefer grabenden View entwickeln, der diese Tags zählt.

Zuerst kehren wir auf die Temporary-View-Seite zurück und geben die folgende map-Funktion ein:

couchdb/tags_by_name_mapper.js

```
function(doc) {
  (doc.albums || []).forEach(function(album){
    (album.tracks || []).forEach(function(track){
      (track.tags || []).forEach(function(tag){
        emit(tag.idstr, 1);
      });
    });
  });
}
```

Diese Funktion taucht in jedes artist-Dokument ein und arbeitet sich dann durch jedes Album, jeden Track und schließlich durch jeden Tag. Für jedes Tag gibt sie ein Schlüssel/Wert-Paar zurück, das aus der `idstr`-Property des Tags (die String-Darstellung des Tags wie *"rock"*) und der Zahl 1 besteht.

Für diese map-Funktion fügen Sie die folgende Reduce-Funktion hinzu:

couchdb/simple_count_reducer.js

```
function(key, values, rereduce) {
  return sum(values);
}
```

Dieser Code summiert einfach die Zahlen in der `values`-Liste auf – wir gehen darauf genauer ein, sobald wir den View ausgeführt haben. Abschließend klicken wir den Run-Button an. Die Ausgabe sollte der folgenden Tabelle entsprechen:

Schlüssel	Wert
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"acid"	1
"acousticguitar"	1
"acousticguitar"	1
"action"	1
"action"	1

Das sollte nicht zu sehr überraschen. Der Wert ist immer 1, wie wir es im Mapper festgelegt haben, und die Schlüsselfelder wiederholen sich entsprechend der Tracks. Beachten Sie aber die Reduce-Checkbox in der oberen rechten Ecke der Ausgabetabelle. Aktivieren Sie die Checkbox und sehen Sie sich die Tabelle erneut an. Sie sollte nun etwa so aussehen:

Schlüssel	Wert
"17sonsrecords"	5
"acid"	1
"acousticguitar"	2
"action"	2
"adventure"	3
"aksband"	1
"alternativ"	1
"alternativ"	3
"ambient"	28
"autodidacta"	17

Was ist passiert? Kurz gesagt hat der Reducer die Ausgabe *reduziert*, indem er die Mapper-Zeilen entsprechend der Reducer-Funktion zusammengefasst hat. Die Mapreduce-Engine von CouchDB funktioniert konzeptionell genau wie die anderen Mapreducer, die wir bisher kennengelernt haben (Riak 3.3, *Fortgeschrittene Views mit Reducern entwickeln*, auf Seite 71 und MongoDB 5.3, *Fortgeschrittene Views mit Reducern entwickeln*, auf Seite 175). Hier ein Überblick der Schritte, die CouchDB unternimmt, um einen View aufzubauen:

1. Übergabe der Dokumente an die Mapper-Funktion.
2. Einsammeln aller emittierten Werte.
3. Sortieren der emittierten Zeilen nach ihren Schlüsseln.
4. Übergabe von Zeilen-Blöcken mit gleichem Schlüssel an die Reduce-Funktion.
5. Wenn zu viele Daten vorliegen, um alle Reduktionen in einem Aufruf durchzuführen, die Reduce-Funktion mit den bereits vorher reduzierten Werten erneut aufrufen.
6. Die Reduce-Funktion rekursiv aufrufen, bis keine doppelten Schlüssel mehr vorhanden sind.

Reduce-Funktionen arbeiten bei CouchDB mit drei Argumenten: `key`, `values` und `rereduce`. Das erste Argument, `key`, ist ein Array von Tupeln – aus zwei Elementen bestehende Arrays, bestehend aus dem vom Mapper emittierten

Schlüssel und der `_id` des Dokuments, das ihn erzeugt hat. Das zweite Argument, `values`, ist ein Array mit den zu den Schlüsseln gehörenden Werten.

Das dritte Argument, `rereduce`, ist ein Boolescher Wert, der wahr ist, wenn dieser Aufruf eine erneute Reduktion (*rereduction*) ist. Das bedeutet, dass bei diesem Aufruf anstelle der Schlüssel und Werte, die von den Mapper-Aufrufen emittiert wurden, die Produkte vorangegangener Reducer-Aufrufe übergeben werden. In diesem Fall ist der `key`-Parameter `null`.

Reducer-Aufrufe durchlaufen

Lassen Sie uns die Ausgabe einmal beispielhaft durchgehen.

Wir wollen die Dokumente (Künstler) mit Titeln betrachten, die als „ambient" getaggt wurden. Die Mapper gehen die Dokumente durch und emittieren Schlüssel/Wert-Paare der Form „ambient"`/1`.

An irgendeinem Punkt werden so viele emittiert worden sein, dass CouchDB einen Reducer aufruft. Dieser Aufruf könnte wie folgt aussehen:

```
reduce(
  [["ambient", id1], ["ambient", id2], ...],    // Schlüssel sind gleich
  [1, 1, ...],                                  // Alle Werte sind 1
  false                                         // rereduce ist falsch
)
```

Erinnern Sie sich daran, dass wir in unserer Reducer-Funktion die `summe` der `values` berechnen. Da die Werte alle 1 sind, ist die Summe einfach die Länge – also die Anzahl der Tracks, die den „ambient"-Tag enthalten. CouchDB hält diesen Rückgabewert zur späteren Verarbeitung fest. Lassen Sie uns annehmen, dass diese Zahl `10` ist.

Etwas später, nachdem CouchDB diese Aufrufe wiederholt ausgeführt hat, entscheidet es sich, die Zwischenergebnisse der Reducer zu kombinieren, indem es einen Rereduce ausführt:

```
reduce(
  null,               // key-Array ist null
  [10, 10, 8],        // Werte sind die Ergebnisse früherer Reducer-Aufrufe
  true                // rereduce ist wahr
)
```

Unsere Reducer-Funktion berechnet wieder die `sum()` der `values`. Diesmal summieren sich die Werte auf `28`. Rereduce-Aufrufe können rekursiv sein. Sie gehen solange weiter, bis die Reduce-Operation abgeschlossen ist, d. h., bis alle Zwischenwerte zu einem zusammengefasst wurden.

Die meisten Mapreduce-Systeme, einschließlich diejenigen, die von den anderen in diesem Buch behandelten Datenbanken wie Riak und MongoDB

verwendet werden, verwerfen das Ergebnis der Mapper und Reducer, sobald sie mit ihrer Arbeit fertig sind. Diese Systeme betrachten Mapreduce als Mittel zum Zweck – etwas, was man ausführt, wenn die Notwendigkeit besteht, wobei man jedesmal von Neuem beginnt. Nicht so bei CouchDB.

Sobald ein View in einem Design-Dokument vorliegt, hält CouchDB die Zwischenwerte der Mapper und Reducer fest, bis eine Änderung an einem Dokument diese Daten ungültig macht. Zu diesem Zeitpunkt führt CouchDB inkrementell Mapper und Reducer aus, um die aktualisieren Daten zu verarbeiten. Es beginnt dabei nicht von vorne, berechnet also nicht jedes Mal alles neu. Das ist das geniale an CouchDB-Views. CouchDB ist in der Lage, Mapreduce als seinen primären Indexierungsmechanismus zu nutzen, indem es Zwischenergebnisse nicht wegwirft.

Änderungen in CouchDB

CouchDBs inkrementeller Mapreduce-Ansatz ist sicherlich ein innovatives Feature und eines von vielen, das CouchDB von anderen Datenbanken abhebt. Das nächste Feature, das wir untersuchen wollen, ist die Changes-API. Dieses Interface stellt einen Mechanismus zur Verfügung, bei dem Änderungen an einer Datenbank überwacht und Updates sofort vorgenommen werden können.

Die Changes-API macht CouchDB zu einem perfekten Kandidaten für ein Aufzeichnungssystem. Stellen Sie sich ein Multidatenbank-System vor, bei dem Daten aus verschiedenen Richtungen eingehen und bei dem andere Systeme auf dem neuesten Stand gehalten werden müssen (tatsächlich machen wir genau das in Kapitel Abschnitt 8.4, *Tag 3: Zusammenspiel mit anderen Datenbanken*, auf Seite 318). Beispiele sind Suchmaschinen, die über Lucene oder ElasticSeach versorgt werden, oder eine über memcached oder Redis implementierte Caching-Schicht. Sie könnten als Reaktion auf Änderungen auch verschiedene Wartungs-Skripten anstoßen, die Aufgaben wir die Verdichtung der Datenbank oder Backups übernehmen. Kurz gesagt eröffnet diese einfache API eine Vielzahl unterschiedlichster Möglichkeiten. Heute werden Sie lernen, wie man sie nutzt.

Um diese API zu nutzen, wollen wir einige einfache Client-Anwendungen mit Node.js entwickeln.[6] Node.js ist ein serverseitige JavaScript-Plattform, die auf der JavaScript-Engine V8 basiert (die auch in Googles Chrome-Browser verwendet wird). Da Node.js eventgesteuert ist und der Code dafür in JavaScript geschrieben wird, ist es für die Integration mit CouchDB die ideale Lösung. Falls Sie Node.js noch nicht besitzen, wechseln Sie auf die Node.js-Site und installieren Sie die neueste stabile Version (wir nutzen Version 0.6).

6. http://nodejs.org/

Die Changes-API gibt es in drei Varianten: Polling, Long-Polling und kontinuierlich. Wir werden alle drei nacheinander besprechen. Wie immer beginnen wir mit cURL, um dem Kern möglichst nah zu kommen, und lassen darauf einen programmatischen Ansatz folgen.

Changes und cURL

Den ersten und einfachsten Weg, um auf die Changes-API zuzugreifen, bildet das Polling-Interface. Wechseln Sie in die Kommandozeile und geben Sie Folgendes ein (wir haben die Ausgabe etwas gekürzt; bei Ihnen wird das anders aussehen):

```
$ curl http://localhost:5984/music/_changes
{
  "results":[{
      "seq":1,
      "id":"370255",
      "changes":[{"rev":"1-a7b7cc38d4130f0a5f3eae5d2c963d85"}]
    },{
      "seq":2,
      "id":"370254",
      "changes":[{"rev":"1-2c7e0deec3ffca959ba0169b0e8bfcef"}]
    },{
      ... 97 more records ...
    },{
      "seq":100,
      "id":"357995",
      "changes":[{"rev":"1-aa649aa53f2858cb609684320c235aee"}]
  }],
  "last_seq":100
}
```

Wenn Sie einen `GET`-Request ohne Parameter an `_changes` senden, antwortet CouchDB mit allem, was es hat. Genau wie beim Zugriff auf Views können Sie einen `limit`-Parameter angeben, um nur eine Teilmenge der Daten anzufordern, und wenn Sie `include_docs=true` anhängen, werden die vollständigen Dokumente zurückgegeben.

Üblicherweise sind Sie nicht an allen Änderungen seit Anbeginn der Zeit interessiert. Viel wahrscheinlicher ist, dass Sie die Änderungen sehen wollen, die es seit der letzten Prüfung gegeben hat. Dazu nutzen Sie den `since`-Parameter.

```
$ curl http://localhost:5984/music/_changes?since=99
{
 "results":[{
 "seq":100,
 "id":"357995",
 "changes":[{"rev":"1-aa649aa53f2858cb609684320c235aee"}]
 }],
 "last_seq":100
}
```

Geben Sie einen `since`-Wert an, der über der letzten Sequenznummer liegt, bleibt die Antwort leer:

```
$ curl http://localhost:5984/music/_changes?since=9000
{
 "results":[
 ],
 "last_seq":9000
}
```

Mithilfe dieser Methode würde die Client-Anwendung regelmäßig prüfen, ob neue Änderungen vorgenommen wurden und anwendungsspezifisch entsprechend reagieren.

Polling ist eine feine Lösung, wenn Ihre Anwendung zeitliche Verzögerungen zwischen den Updates toleriert. Sind Update relativ selten, könnte das der Fall sein. Wenn Sie beispielsweise Blog-Einträge abrufen, könnte ein Polling alle fünf Minuten reichen.

Wenn Sie schnellere Updates wünschen, ohne den Overhead durch das ständige erneute Öffnen der Verbindung zu erhöhen, dann ist das sog. Long-Polling die bessere Option. Geben Sie den URL-Parameter `feed=longpoll` mit an, hält CouchDB die Verbindung eine gewisse Zeit aufrecht (und wartet auf Änderungen), bevor es den Request abschließt. Versuchen Sie Folgendes:

```
$ curl 'http://localhost:5984/music/_changes?feed=longpoll&since=9000'
{"results":[
```

Sie sollten den Anfang einer JSON-Response sehen, aber sonst nichts. Wenn Sie das Terminal lang genug offen lassen, wird die Verbindung irgendwann von CouchDB geschlossen:

```
],
"last_seq":9000}
```

Aus der Sicht eines Entwicklers ist das Schreiben eines Treibers, der CouchDB auf Änderungen mittels Polling überwacht, identisch mit der Entwicklung eines Longpolling-Treibers. Der Unterschied besteht eigentlich nur darin, wie lange CouchDB gewillt ist, die Verbindung aufrecht zu halten. Wir wollen unsere Aufmerksamkeit nun einer Node.js-Anwendung widmen, die mögliche Änderungen überwacht und die erfassten Daten nutzt.

Änderungen überwachen mit Node.js

Node.js ist ein stark eventgesteuertes System, weshalb auch unser CouchDB-Wächter (Watcher) diesem Prinzip folgt. Unser Treiber überwacht des Chan-

ges-Feed und gibt Änderungs-Events aus, sobald CouchDB geänderte Dokumente meldet. Wir beginnen mit dem groben Entwurf unseres Treibers und reden über seine wichtigsten Punkte. Danach fügen wir die Feed-spezifischen Details hinzu.

Hier also der Entwurf unseres Watcher-Programms und eine kurzen Erklärung, was er macht:

couchdb/watch_changes_skeleton.js

```
var
  http = require('http'),
  events = require('events');

/**
 * Erzeuge CouchDB-Watcher basierend auf Verbindungskriterien;
 * folgt node.js EventEmitter-Muster und emittiert 'change'-Events.
 */

① exports.createWatcher = function(options) {

②   var watcher = new events.EventEmitter();

    watcher.host = options.host || 'localhost';
    watcher.port = options.port || 5984;
    watcher.last_seq = options.last_seq || 0;
    watcher.db = options.db || '_users';

③   watcher.start = function() {
      // ... feed-spezifische Implementation ...
    };

    return watcher;

};

// CouchDB-Überwachung starten, wenn Hauptskript

④ if (!module.parent) {
    exports.createWatcher({
      db: process.argv[2],
      last_seq: process.argv[3]
    })
      .on('change', console.log)
      .on('error', console.error)
      .start();
}
```

① `exports` ist ein Standard-Objekt, das von der CommonJS-Modul-API bereitgestellt wird, die Node.js implementiert. Das Hinzufügen der `createWatcher()`-Methode zu `exports` macht sie für andere Node.js-Skripten verfügbar, die sie als Bibliothek nutzen wollen. Mit dem `options`-Argument kann der Aufrufer festlegen, welche Datenbank überwacht werden soll, und er kann weitere Verbindungseinstellungen überschreiben.

② `createWatcher()` erzeugt ein `EventEmitter`-Objekt, das der Aufrufer nutzen kann, um auf Change-Events zu warten. Die relevanten Fähigkeiten eines `EventEmitter` sind die Möglichkeiten, durch den Aufruf seiner `on()`-Methode auf Events zu warten und durch den Aufruf seiner `emit()`-Methode Events auszulösen.

③ `watcher.start()` ist dafür verantwortlich, HTTP-Requests anzustoßen, um CouchDB auf Änderungen zu überwachen. Treten Änderungen in Dokumenten auf, muss der `watcher` sie als Change-Events emittieren. Alle Feed-spezifischen Details der Implementierung sind hier zu finden.

④ Das letzte Stück Code am Ende legt fest, was das Skript machen soll, wenn es direkt über die Kommandozeile aufgerufen wird. In unserem Fall ruft das Skript die `createWatcher()`-Methode auf und richtet Listener für das zurückgelieferte Objekt ein, die die Ergebnisse über die Standardausgabe ausgeben. Über Kommandozeilen-Argumente kann man festlegen, welche Datenbank verwendet und bei welcher Sequenz-ID begonnen werden soll.

Bisher enthält dieser Code nichts CouchDB-Spezifisches, sondern zeigt nur die Art und Weise, wie Node.js die Dinge angeht. Der Code mag Ihnen fremd vorkommen, insbesondere, wenn Sie noch nicht mit ereignisgesteuerten Server-Techniken gearbeitet haben. Doch wir werden das im weiteren Verlauf des Buches immer mehr nutzen.

Nachdem unser grober Aufbau steht, wollen wir den Code einfügen, der die Verbindung mit CouchDB über Longpolling herstellt und Events emittiert. Nachfolgend nur der Code, der in die `watcher.start()`-Methode gehört. Eingefügt in unseren obigen Entwurf (wo der Kommentar von der *Feed-spezifischen Implementierung* spricht), sollte die neue vollständige Datei `watch_changes_longpolling.js` heißen.

couchdb/watch_changes_longpolling_impl.js

```
    var
①     http_options = {
        host: watcher.host,
        port: watcher.port,
        path:
          '/' + watcher.db + '/_changes' +
          '?feed=longpoll&include_docs=true&since=' + watcher.last_seq
      };

②   http.get(http_options, function(res) {
      var buffer = '';
      res.on('data', function (chunk) {
        buffer += chunk;
      });
      res.on('end', function() {
```

```
③         var output = JSON.parse(buffer);
          if (output.results) {
            watcher.last_seq = output.last_seq;
            output.results.forEach(function(change){
              watcher.emit('change', change);
            });
            watcher.start();
          } else {
            watcher.emit('error', output);
          }
        })
      })
      .on('error', function(err) {
        watcher.emit('error', err);
      });
```

① Als Vorbereitung auf den Request richtet das Skript zuerst ein `http_options`-Konfigurationsobjekt ein. Der `path` verweist auf die gleiche `_changes`-URL, die wir bereits genutzt haben. `feed` wird auf `longpoll` gesetzt und `include_docs=true`.

② Danach ruft das Skript `http.get()` auf, eine Methode der Node.js-Bibliothek, die einen `GET`-Request entsprechend unserer Einstellungen durchführt. Der zweite Parameter an `http.get` ist ein Callback, der die `HTTPResponse` empfängt. Das Response-Objekte sendet `data`-Events, während der Inhalt verarbeitet wird, die wir in den `buffer` einfügen.

③ Wenn das Response-Objekt ein `end`-Event sendet, verarbeiten wir den Puffer (der JSON enthalten sollte). Dem entnehmen wir den neuen `last_seq`-Wert, senden ein `change`-Event und rufen `watcher.start()` erneut auf, um auf die nächste Änderung zu warten.

Um das Skript im Kommandozeilen-Modus auszuführen, starten Sie es wie folgt (die Ausgabe haben wir gekürzt):

```
$ node watch_changes_longpolling.js music
{ seq: 1,
 id: '370255',
 changes: [ { rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85' } ],
 doc:
 { _id: '370255',
 _rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85',
 albums: [ [Object] ],
 id: '370255',
 name: '""ATTIC""',
 url: 'http://www.jamendo.com/artist/ATTIC_(3)',
 mbgid: '',
 random: 0.4121620435325435 } }
{ seq: 2,
 id: '370254',
 changes: [ { rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef' } ],
```

```
doc:
{ _id: '370254',
_rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef',
... 98 more entries ...
```

Hurra, unsere Anwendung funktioniert! Nachdem sie einen Datensatz für jedes Dokument ausgibt, läuft der Prozess weiter und wartet auf weitere Änderungen.

Sie können das Dokument direkt in Futon ändern oder den `@max`-Wert in `import_from_jamendo.rb` ändern und es erneut ausführen. Diese Änderungen spiegeln sich in der Kommandozeile wider. Jetzt legen wir noch einen Zahn zu und nutzen kontinuierliche Feeds, um die Updates noch flotter zu machen.

Änderungen kontinuierlich überwachen

Die Polling- und Longpolling-Feeds des `_changes`-Dienstes erzeugen beide saubere JSON-Ergebnisse. Der *continuous*-Feed geht die Dinge etwas anders an. Statt alle verfügbaren Änderungen in einem `results`-Array zusammenzufassen und danach den Stream zu schließen, sendet er jede Änderung einzeln und hält die Verbindung offen. Auf diese Weise kann er serialisierte JSON-Änderungsmitteilungsobjekte zurückliefern, sobald sie verfügbar sind.

Um zu sehen, wie das funktioniert, probieren Sie Folgendes aus (wir haben die Ausgabe der Lesbarkeit halber gekürzt):

```
$ curl 'http://localhost:5984/music/_changes?since=97&feed=continuous'
{"seq":98,"id":"357999","changes":[{"rev":"1-0329f5c885...87b39beab0"}]}
{"seq":99,"id":"357998","changes":[{"rev":"1-79c3fd2fe6...1e45e4e35f"}]}
{"seq":100,"id":"357995","changes":[{"rev":"1-aa649aa53f...320c235aee"}]}
```

Treten für eine Weile keine Änderungen ein, schließt CouchDB irgendwann die Verbindung, nachdem es eine Zeile wie die folgende ausgegeben hat:

```
{"last_seq":100}
```

Diese Methode hat gegenüber Polling bzw. Longpolling den Vorteil eines reduzierten Overheads, weil die Verbindung offen bleibt. Es geht keine Zeit beim Wiederaufbau der Verbindung verloren. Auf der anderen Seite ist die Ausgabe kein JSON, d. h., das Parsing ist etwas aufwendiger. Auch ist diese Variante keine gute Wahl, wenn der Client ein Web-Browser ist. Ein den Feed asynchron herunterladender Browser empfängt möglicherweise keinerlei Daten, bis die Verbindung abgebaut wird. In diesem Fall sollten Sie besser Longpolling verwenden.

Änderungen filtern

Wie gerade gesehen, bietet die Changes-API einen Einblick in die Vorgänge innerhalb einer CouchDB-Datenbank. Sie liefert alle Änderungen in einem einzigen Stream zurück. Manchmal will man aber gar nicht jede kleine Änderung wissen, sondern ist nur an einer Teilmenge der Änderungen interessiert. Zum Beispiel könnten Sie nur wissen wollen, ob Dokumente gelöscht wurden oder ob Dokumente einer bestimmten Gruppe verändert wurden. An dieser Stelle kommen *Filterfunktionen* ins Spiel.

Ein Filter ist eine Funktion, die ein Dokument (und Request-Informationen) nimmt und entscheidet, ob er das Dokument durchlässt oder nicht. Das ist abhängig vom Rückgabewert. Sehen wir uns an, wie das funktioniert. In unserer `music`-Datenbank besitzen die meisten artist-Dokumente eine `country`-Property, die einen aus drei Buchstaben bestehenden Code enthält. Nehmen wir an, wir wären nur an Bands aus Russland (RUS) interessiert. Unsere Filterfunktion könnte wie folgt aussehen:

```
function(doc) {
  return doc.country === "RUS";
}
```

Fügen wir das in ein Design-Dokument unter dem Schlüssel `filters` ein, können wir sie bei `_changes`-Request mit angeben. Doch bevor wir das tun, wollen wir unser Beispiel erweitern. Statt immer nur russische Bands zu filtern, sollten wir die Eingabe parametrisieren, damit wir das gewünschte Land in der URL angeben können.

Hier ein parametrisierter Länderfilter:

```
function(doc, req) {
  return doc.country === req.query.country;
}
```

Beachten Sie, dass wir die `country`-Property des Dokuments mit einem Parameter gleichen Namens vergleichen, der im Query-String des Requests angegeben wurde. Um uns das in Aktion anzusehen, legen wir ein neues Design-Dokument für geographische Filter an und fügen die Funktion hinzu:

```
$ curl -X PUT \
 http://localhost:5984/music/_design/wherabouts \
 -H "Content-Type: application/json" \
 -d '{"language":"javascript","filters":{"by_country":
 "function(doc,req){return doc.country === req.query.country;}"
 }}'
```

```
{
 "ok":true,
 "id":"_design/wherabouts",
 "rev":"1-c08b557d676ab861957eaeb85b628d74"
}
```

Nun können wir Change-Requests nach Land filtern:

```
$ curl "http://localhost:5984/music/_changes?\
filter=wherabouts/by_country&\
country=RUS"
{"results":[
{"seq":10,"id":"5987","changes":[{"rev":"1-2221be...a3b254"}]},
{"seq":57,"id":"349359","changes":[{"rev":"1-548bde...888a83"}]},
{"seq":73,"id":"364718","changes":[{"rev":"1-158d2e...5a7219"}]},
...
```

Mit Filtern können wir eine Art Pseudosharing einrichten, bei der nur eine Teilmenge der Records zwischen den Knoten repliziert wird. Das ist nicht das Gleiche wie bei echten Sharding-Systemen wie MongoDB oder HBase, bietet aber die Möglichkeit, die Verarbeitung bestimmter Arten von Requests aufzuteilen. Beispielsweise könnte Ihr CouchDB-Hauptserver separate Filter für Benutzer, Bestellungen, Nachrichten und Lager verwenden. Separate CouchDB-Server könnten Änderungen basierend auf diesen Filtern replizieren und jeder könnte einen anderen Aspekt des Unternehmens abdecken.

Da Filterfunktionen beliebigen JavaScript-Code enthalten können, lässt sich auch eine komplexere Logik hineinpacken. Die Prüfung tief verschachtelter Felder wäre mit dem vergleichbar, was wir mit Views gemacht haben. Sie können auch reguläre Ausdrücke zu Testzwecken nutzen oder mathematische Vergleiche anstellen (z. B. nach Datum filtern). Es gibt sogar eine User-Kontext-Property im Request-Objekt (`req.userCtx`), mit der Sie mehr über die Request-Credentials herausfinden können.

Wir werden Node.js und die CouchDB-Changes-API in Kapitel 8, *Redis*, auf Seite 285 wiedersehen, wenn wir eine Multidatenbank-Anwendung entwickeln. Jetzt wollen wir uns aber dem letzten charakteristischen Feature von CouchDB zuwenden, das wir hier behandeln wollen: der Replikation.

Daten replizieren bei CouchDB

Bei CouchDB dreht sich alles um asynchrone Umgebungen und die Langlebigkeit von Daten. Aus Sicht von CouchDB ist es am sichersten, Ihre Daten überall zu speichern und es gibt Ihnen die Werkzeuge an die Hand, mit denen Sie das erreichen können. Einige der Datenbanken, die wir uns angesehen haben, verwenden einen einzelnen Master-Knoten, um die Konsistenz zu gewährleisten. Wieder andere erreichen das über eine Mehrheit überein-

CouchDB oder BigCouch?

Der Ansatz von CouchDB ist für eine Vielzahl von Einsatzgebieten sinnvoll. Es füllt sicherlich eine Nische, die die anderen von uns diskutierten Datenbanken größtenteils nicht abdecken. Andererseits ist es manchmal schön, wenn man Daten selektiv zwischen Knoten replizieren kann, um den verfügbaren Plattenplatz auszunutzen. Anstatt also alle Knoten alle Daten vorhalten zu lassen, wird nur eine bestimmte Anzahl von Kopien vorgehalten. Das ist das *N* im NWR, das wir in Abschnitt *Nodes/Writes/Reads*, auf Seite 82 diskutiert haben.

Das ist kein Feature, das CouchDB direkt von Hause aus unterstützt, doch keine Sorge! BigCouch deckt diesen Bedarf. BigCouch wird von Cloudant entwickelt und gepflegt und bietet eine CouchDB-kompatible Schnittstelle (mit nur wenigen kleinen Unterschieden[a]). Hinter den Kulissen implementiert es aber die Sharding- und Replikationsstrategie einer Dynamo-inspirierten Datenbank wie Riak.

Die Installation von BigCouch macht recht viel Arbeit – und ist wesentlich schwieriger als bei CouchDB – , doch das kann sich lohnen, wenn Ihr Anwendungsszenario ein großes Rechenzentrum vorsieht.

a. http://bigcouch.cloudant.com/api

stimmender Knoten. CouchDB macht beides nicht, sondern unterstützt die sogenannte Multimaster- oder auch Master/Master-Replikation.

Jeder CouchDB-Server kann gleichberechtigt Updates empfangen, auf Requests antworten und Daten löschen, und zwar unabhängig davon, ob er die Verbindung zu anderen Servern herstellen kann oder nicht. Bei diesem Modell werden Änderungen selektiv in einer Richtung repliziert und die Replikation erfolgt für alle Daten gleich. Mit anderen Worten: Es gibt kein Sharding. An der Replikation teilnehmende Server verfügen über alle Daten.

Die Replikation ist das letzte wichtige CouchDB-Thema, das wir behandeln wollen. Zuerst wollen wir uns ansehen, wie man Ad-hoc- und kontinuierliche Replikation zwischen Datenbanken einrichtet. Dann gehen wir die Folgen von Daten-Konflikten durch und zeigen, wie man Anwendungen entwickelt, die mit diesen Fällen elegant umgehen können.

Zum Einstieg klicken Sie den Replicator-Link im Tools-Menü auf der rechten Seite an. Eine Seite wie in Abbildung 29, *CouchDB Futon: Replicator*, auf Seite 231 erscheint. Im „Replicate changes from"-Dialog wählen Sie *music* aus dem linken Dropdown-Menü und tragen *music-repl* im rechten Feld ein. Achten Sie darauf, dass die Continuous-Checkbox deaktiviert ist, und klicken Sie auf Replicate. Klicken Sie auf OK, um die `music-repl`-Datenbank zu erzeugen. Sie

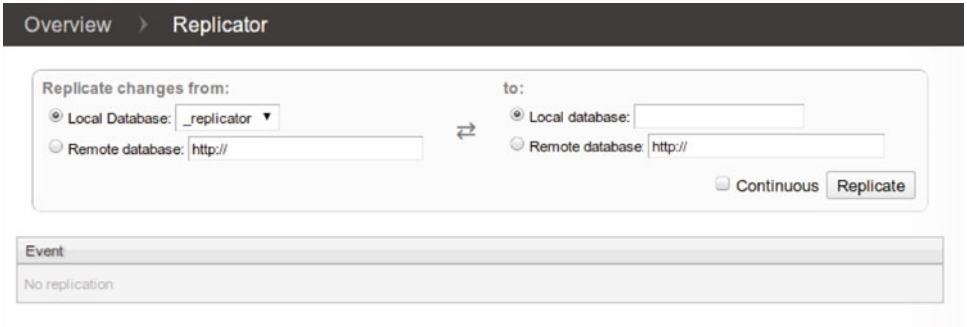


Abbildung 29: CouchDB Futon: Replicator

sollten eine entsprechende Event-Meldung im Event-Log unter dem Formular sehen.

Um sicherzugehen, dass der Replikations-Request funktioniert hat, wechseln Sie zurück in die Futon-Übersicht. Dort sollten Sie nun eine neue Datenbank namens `music-repl` sehen, die die gleiche Anzahl von Dokumenten aufweist wie unsere `music`-Datenbank. Wenn Sie weniger anzeigt, geben Sie ihr noch etwas Zeit und laden Sie dann die Seite neu – CouchDB könnte noch bei der Arbeit sein. Wundern Sie sich nicht, wenn die Update-Seq-Werte nicht übereinstimmen. Das liegt daran, dass die `music`-Originaldatenbank Dokument-Löschungen und -Updates enthält, während es bei `music-repl` nur Einfügungen gibt, um die Dinge zu beschleunigen.

Konflikte erzeugen

Als Nächstes wollen wir einen Konflikt erzeugen und untersuchen, wie man mit ihm umgeht. Halten Sie die Replicator-Seite bereit, da wir die Ad-hoc-Replikation zwischen `music` und `music-repl` häufig anstoßen werden.

Wechseln Sie auf die Kommandozeile und geben Sie Folgendes ein, um ein Dokument in die `music`-Datenbank einzufügen:

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
 -H "Content-Type: application/json" \
 -d '{ "name": "The Conflicts" }'
{
 "ok":true,
 "id":"theconflicts",
 "rev":"1-e007498c59e95d23912be35545049174"
}
```

Auf der Replicator-Seite klicken Sie Replicate an, um eine weitere Synchronisation anzustoßen. Wir können sicherstellen, dass das Dokument erfolgreich repliziert wurde, indem wir es aus der `music-repl`-Datenbank abrufen.

```
$ curl "http://localhost:5984/music-repl/theconflicts"
{
 "_id":"theconflicts",
 "_rev":"1-e007498c59e95d23912be35545049174",
 "name":"The Conflicts"
}
```

Nun fügen wir in die `music-repl`-Datenbank ein Album namens *Conflicts of Interest* ein.

```
$ curl -X PUT "http://localhost:5984/music-repl/theconflicts" \
  -H "Content-Type: application/json" \
  -d '{
    "_id": "theconflicts",
    "_rev": "1-e007498c59e95d23912be35545049174",
    "name": "The Conflicts",
    "albums": ["Conflicts of Interest"]
}'
{
  "ok":true,
  "id":"theconflicts",
  "rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5"
}
```

Und erzeugen einen Update-Konflikt in `music`, indem wir ein anderes Album namens *Conflicting Opinions* einfügen.

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
 -H "Content-Type: application/json" \
 -d '{
   "_id": "theconflicts",
   "_rev": "1-e007498c59e95d23912be35545049174",
   "name": "The Conflicts",
   "albums": ["Conflicting Opinions"]
 }'
{
  "ok":true,
  "id":"theconflicts",
  "rev":"2-cab47bf4444a20d6a2d2204330fdce2a"
}
```

An dieser Stelle besitzt sowohl die `music`- als auch die `music-repl`-Datenbank ein Dokument mit dem `_id`-Wert `theconflicts`. Beide Dokumente liegen in Version 2 vor und wurden aus der gleichen Ur-Revision (`1-e007498c59e95d23912be35545049174`) abgeleitet. Die Frage lautet nun: Was passiert, wenn wir sie replizieren wollen?

Konflikte auflösen

Da unser Dokument in den beiden Datenbanken jetzt einen Konflikt darstellt, wechseln wir wieder auf die Replicator-Seite und stoßen eine weitere Replikation an. Wenn Sie nun einen Fehler erwarten, werden Sie schockiert feststellen, dass die Operation erfolgreich durchgeführt wird. Wie geht CouchDB also mit diesem Widerspruch um?

Wie sich zeigt, wählt CouchDB einfach einen aus und deklariert ihn zum Sieger. Mit Hilfe eines deterministischen Algorithmus wählen alle CouchDB-Knoten den gleichen Sieger aus, wenn ein Konflikt erkannt wird. Doch damit ist die Geschichte noch nicht vorbei. CouchDB speichert auch die nicht gewählten „Verlierer"-Dokumente, so dass sich eine Client-Anwendung zu einem späteren Zeitpunkt die Situation ansehen und auflösen kann.

Um herauszufinden, welche Version unseres Dokuments während der letzten Replikation gewonnen hat, können wir es mit einem normalen `GET`-Request abrufen. Durch Anhängen des URL-Parameters `conflicts=true` liefert CouchDB auch Informationen zu allen in Konflikt stehenden Revisionen zurück.

```
$ curl http://localhost:5984/music-repl/theconflicts?conflicts=true
{
 "_id":"theconflicts",
 "_rev":"2-cab47bf4444a20d6a2d2204330fdce2a",
 "name":"The Conflicts",
 "albums":["Conflicting Opinions"],
 "_conflicts":[
   "2-0c969fbfa76eb7fcdf6412ef219fcac5"
 ]
}
```

Wie wir sehen, hat also das zweite Update gewonnen. Beachten Sie das `_conflicts`-Feld in der Response. Es enthält eine Liste anderer Revisionen, die im Konflikt mit der gewählten Revision stehen. Indem wir einen `rev`-Parameter im `GET`-Request angeben, können wir diese Revisionen abrufen und entscheiden, was wir mit ihnen anfangen wollen.

```
$ curl http://localhost:5984/music-repl/theconflicts?rev=2-0c969f...
{
 "_id":"theconflicts",
 "_rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5",
 "name":"The Conflicts",
 "albums":["Conflicts of Interest"]
}
```

Was wir hier mitnehmen, ist die Tatsache, dass CouchDB nicht versucht, miteinander im Konflikt stehende Änderungen irgendwie intelligent zu verei-

nen. Wie zwei Dokumente zu vereinen sind, ist hochgradig abhängig von der Anwendung, d. h., eine allgemeine Lösung ist nicht praktikabel. In unserem Beispiel ist es sinnvoll, die beiden `albums`-Arrays miteinander zu verknüpfen, doch es gibt auch Szenarien, bei denen die richtige Aktion nicht so offensichtlich ist.

Nehmen wir zum Beispiel an, dass Sie eine Datenbank mit Kalendereinträgen verwalten. Ein Kopie liegt auf Ihrem Smartphone, eine andere auf Ihrem Laptop. Sie erhalten von einem Party-Veranstalter eine Nachricht, die die Location für eine von Ihnen veranstaltete Party angibt. Sie aktualisieren daher Ihre Telefon-Datenbank. Zurück im Büro erhalten Sie eine weitere E-Mail vom Veranstalter, die einen *anderen* Ort angibt. Sie aktualisieren also Ihre Laptop-Datenbank und replizieren die Daten. CouchDB kann nicht wissen, welcher der beiden Orte der richtige ist. Das Beste, was es machen kann, ist, die alten Werte festzuhalten, damit Sie später festlegen können, welcher der Werte denn nun verwendet werden soll. Es liegt an der Anwendung herauszufinden, wie die richtige Benutzerschnittstelle aussehen muss, um diese Situation zu präsentieren und nach einer Lösung zu fragen.

Was wir am dritten Tag gelernt haben

Und so endet unsere Tour durch CouchDB. Wir haben den dritten Tag damit begonnen zu lernen, wie wir unsere Mapreduce-generierten Views erweitern können. Danach sind wir tief in die Changes-API abgetaucht und haben dabei noch eine Abstecher in die Welt der ereignisgesteuerten serverseitigen JavaScript-Entwicklung mit Node.js gemacht. Abschließend haben wir uns kurz angesehen, wie CouchDB seine Master/Master-Replikationsstrategie verfolgt und wie Anwendungen Konflikte erkennen und beheben können.

Tag 3: Selbststudium

Finden Sie heraus

1. Welche nativen Reducer gibt es bei CouchDB? Was sind die Vorteile nativer Reducer gegenüber eigenen JavaScript-Reducern?
2. Wie können Sie von der `_changes`-API eingehende Änderungen serverseitig filtern?
3. Wie alles bei CouchDB wird auch die Initialisierung und der Abbruch der Replikation hinter den Kulissen über HTTP-Befehle abgewickelt. Wir lauten die REST-Befehle zur Einrichtung und dem Entfernen von Replikationsbeziehungen zwischen Servern?
4. Wie können Sie die `_replicator`-Datenbank nutzen, um Replikationsbeziehungen zu erhalten?

Machen Sie Folgendes

1. Entwickeln Sie ein neues Modul namens `watch_changes_continuous.js`, das auf dem Node.js-Grundgerüst basiert, das in Abschnitt *Änderungen überwachen mit Node.js*, auf Seite 223 beschrieben wurde.
2. Implementieren Sie `watcher.start` in der Form, dass es den `_changes`-Feed kontinuierlich überwacht. Stellen Sie sicher, dass es die gleiche Ausgabe erzeugt wie `watch_changes_longpolling.js`.

   Tipp: Wenn Sie nicht weiterkommen, finden Sie eine Beispiel-Implementierung in den Downloads zu diesem Buch.
3. Dokumente mit im Konflikt miteinander stehenden Revisionen haben eine `_conflicts`-Property. Entwickeln Sie einen View, der solche Revisionen ausgibt und mit der dazugehörigen Doc-`_id` verknüpft.

6.5 Zusammenfassung

Im Verlauf dieses Kapitels haben wir gesehen, wie man eine recht breite Palette an Aufgaben mit CouchDB erledigen kann, angefangen bei einfachen CRUD-Operationen bis hin zur Entwicklung von Views aus Mapreduce-Funktionen. Wir haben gesehen, wie man Änderungen überwacht und wir haben die Entwicklung nichtsperrender ereignisgesteuerter Client-Anwendung untersucht. Zum Schluss haben wir gelernt, wie man die Ad-hoc-Replikation zwischen Datenbanken durchführt sowie Konflikte erkennt und aufhebt. Zwar haben wir sehr viele Dinge nicht behandelt, doch jetzt ist es an der Zeit für eine Zusammenfassung, bevor wir uns der nächsten Datenbank zuwenden.

CouchDBs Stärken

CouchDB ist ein robustes und stabiles Mitglied der NoSQL-Community. Basierend auf der Philosophie, dass Netzwerke unzuverlässig und Hardwarefehler unvermeidlich sind, bietet CouchDB einen dezentralisierten Ansatz für die Datenspeicherung. Klein genug, um auf Ihrem Smartphone zu laufen und doch groß genug, um Unternehmen zu unterstützen, bietet sich CouchDB für eine Vielzahl von Einsatzgebieten an.

CouchDB ist ebenso API wie Datenbank. In diesem Kapitel haben wir uns auf das offizielle Apache CouchDB-Projekt konzentriert, doch es gibt eine steigende Anzahl alternativer Implementierungen und CouchDB-Service-Anbieter, die auf hybriden Backends aufbauen. Da CouchDB „aus dem Web für das Web“ entsteht, kann es recht einfach in Web-Technologien – wie Load-Balancer und Caching-Layer – integriert werden.

CouchDBs Schwächen

Natürlich eignet sich CouchDB nicht für jede Aufgabe. CouchDBs Mapreduce-basierte Views sind zwar neuartig, bieten aber nicht all die schicken Möglichkeiten, die man von einer relationalen Datenbank erwartet. Tatsächlich sollten Sie im Produktionsbetrieb *überhaupt keine* Ad-hoc-Queries ausführen. Auch ist CouchDBs Replikationsstrategie nicht immer die richtige Wahl. Die CouchDB-Replikationsstrategie lautet alles oder nichts, d. h., alle replizierten Server haben den gleichen Inhalt. Es gibt kein Sharding, mit dem man die Inhalte im Rechenzentrum verteilen könnte. Der Hauptgrund, neue CouchDB-Knoten hinzuzufügen, besteht nicht darin, die Daten zu verteilen, sondern den Durchsatz von Schreib-Lese-Operationen zu erhöhen.

Abschließende Gedanken

CouchDBs Augenmerk auf Stabilität im Angesicht der Ungewissheit macht es zu einer guten Wahl, wenn Ihr System der rauhen Realität des wilden Internets trotzen muss. Indem es Standard-Webtechniken wie HTTP/REST und JSON nutzt, passt CouchDB dort besonders gut, wo Web-Techniken vorherrschen, also nahezu überall. Innerhalb des durch Mauern gesicherten Gartens eines Rechenzentrums kann CouchDB dennoch sinnvoll sein, wenn Sie sich darauf einlassen, Konflikte zu lösen, wenn sie eintreten oder wenn Sie eine alternative Implementierung wie BigCouch verwenden und kein standardmäßiges Sharding erwarten.

Es gibt eine Reihe weiterer Features, die CouchDB einmalig und besonders machen, auf die wir nicht eingehen konnten. Eine kurze Liste würde die Einfachheit von Backups, binäre Anhänge für Dokumente und CouchApps (ein System zur Entwicklung und dem Einsatz von Web-Anwendungen direkt durch CouchDB ohne weitere Middleware) umfassen. Wir hoffen, Ihnen dennoch einen guten Überblick gegeben zu haben, der Appetit auf mehr macht. Probieren Sie CouchDB bei Ihrer nächsten datengesteuerten Web-Anwendung aus, Sie werden nicht enttäuscht werden!

KAPITEL 7

Neo4J

Ein Gummiband ist vielleicht kein normales Werkzeug für einen Schreiner, ebensowenig wie Neo4j keine Standard-Datenbank ist. Mit einem Gummiband bindet man Dinge fest – ganz egal, wie unmöglich geformt die Objekte auch sein mögen. Wenn Sie einen Tisch auf möglichst natürliche Weise auf einem Lieferwagen befestigen wollen, ist das Gummiband wie für Sie gemacht.

Neo4j ist ein neuer Typ von NoSQL-Datenspeicher, der als *Graph-Datenbank* bezeichnet wird. Wie es der Name andeutet, speichert sie Daten in Form von Graphen (im mathematischen Sinn). Sie ist als „Whiteboard-freundlich" bekannt, d. h., wenn Sie einen Entwurf in Form von Kästchen und Linien auf einem Whiteboard zeichnen können, dann können Sie ihn auch in Neo4j speichern. Neo4j konzentriert sich mehr auf die *Beziehungen zwischen Werten* denn auf die Gemeinsamkeiten *zwischen Wertemengen* (wie Collections von Dokumenten oder Tabellen von Zeilen). Auf diese Weise kann es hochgradig variable Daten auf natürliche und einfache Weise speichern.

Neo4j ist klein genug, um in nahezu jede Anwendung eingebettet werden zu können. Andererseits kann Neo4j Milliarden von Knoten und ebensoviele Kanten speichern. Und die Cluster-Unterstützung mit Master/Slave-Replikation über viele Server macht sie für Probleme nahezu jeder Größenordnung geeignet.

7.1 Neo4J ist Whiteboard-freundlich

Stellen Sie sich vor, Sie müssten eine Engine für Wein-Empfehlungen entwickeln, die unterschiedliche Arten von Weinen, Regionen, Weingütern, Jahrgängen und Bezeichnungen berücksichtigt. Vielleicht müssen Sie Artikel von Autoren nachhalten, die Weine beschreiben. Vielleicht wollen Sie den Benutzer seine Favoriten selbst nachhalten lassen.

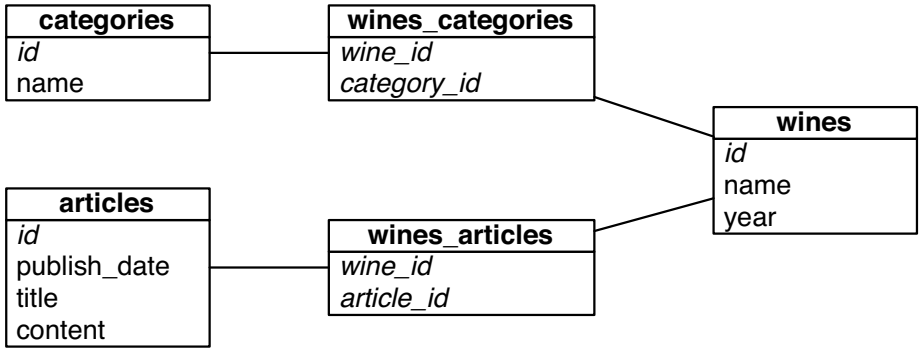


Abbildung 30: Wein-Empfehlungsschema in relationalem UML

Ein relationales Modell könnte eine Kategorie-Tabelle verwenden und eine M-zu-M-Beziehung zwischen den einzelnen Weinen eines Winzers und eine Kombination aus Kategorien und anderen Daten aufbauen. Doch das entspricht nicht ganz dem, wie wir Menschen Daten im Geiste modellieren. Vergleichen Sie die beiden Abbildungen in Abbildung 30, *Wein-Empfehlungsschema in relationalem UML* und Abbildung 31, *Wein-Empfehlung auf dem Whiteboard*, auf Seite 239 miteinander. Ein altes Sprichtwort aus der relationalen Welt besagt: *Auf einer ausreichend langen Zeitachse werden alle Felder optional*. Neo4j handhabt das implizit, indem es Werte und Strukturen nur bei Bedarf zur Verfügung stellt. Wenn es für einen Wein keinen Jahrgang gibt, verwenden Sie stattdessen ein Jahr für die Flasche und lassen den Jahrgang auf den Wein-Knoten zeigen. Es gibt kein Schema, das man anpassen müsste.

In den nächsten drei Tagen werden wir lernen, wie man mit Neo4j über die Console und über REST sowie über Such-Indizes interagiert. Wir werden einige größere Graphen mit Graph-Algorithmen bearbeiten. Abschließend werden wir an Tag 3 einen Blick auf die Tools werfen, die Neo4j für unternehmenskritische Anwendungen bereitstellt. Sie reichen von vollständig ACID-konformen Transaktionen über Hochverfügbarkeits-Cluster bis hin zu inkrementellen Backups.

In diesem Kapitel verwenden wir die Enterprise Edition Neo4j 1.7. Die meisten Aktionen können wir auch mit der GPL Community-Edition durchführen, doch für Tag 3 benötigen wir etwas Enterprise-Funktionalität: Hochverfügbarkeit.

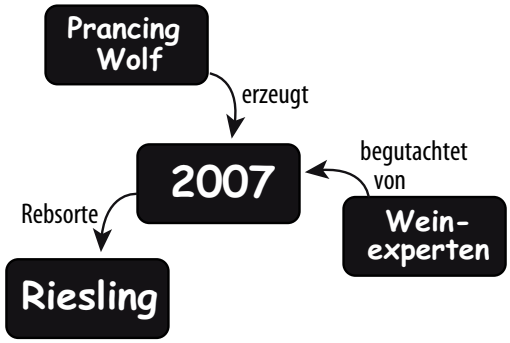


Abbildung 31: Wein-Empfehlung auf dem Whiteboard

7.2 Tag 1: Graphen, Groovy und CRUD

Heute steigen wir gleich richtig ins Geschehen ein. Wir sehen uns das Neo4j Web-Interface an und tauchen tief in die Terminologie von Graph-Datenbanken und CRUD ab. Einen Großteil des Tages verbringen wir damit, zu lernen, wie man eine Graph-Datenbank durch *Traversierung* abfragt. Die hier vorgestellten Konzepte unterscheiden sich grundlegend von den anderen Datenbanken, die wir uns bisher angesehen haben, die größtenteils eine Dokumenten- oder Datensatz-basierte Sicht auf die Welt haben. Bei Neo4j dreht sich hingegen alles um Beziehungen.

Doch bevor wir uns all dem zuwenden, wollen wir mit dem Web-Interface beginnen, um zu sehen, wie Neo4j Daten in Graphen-Form darstellt und wie man sich in diesem Graphen bewegt. Nachdem Sie das Neo4j-Paket heruntergeladen und entpackt haben, wechseln Sie mit `cd` in das entsprechende Verzeichnis und starten den Server wie folgt:

```
$ bin/neo4j start
```

Um sicherzugehen, dass alles läuft, rufen wir mit `curl` die folgende URL ab:

```
$ curl http://localhost:7474/db/data/
```

Wie CouchDB wird auch das Standard-Neo4j-Paket mit einem umfangreichen Web-Administrationstool und Datenbrowser ausgeliefert, mit dem man

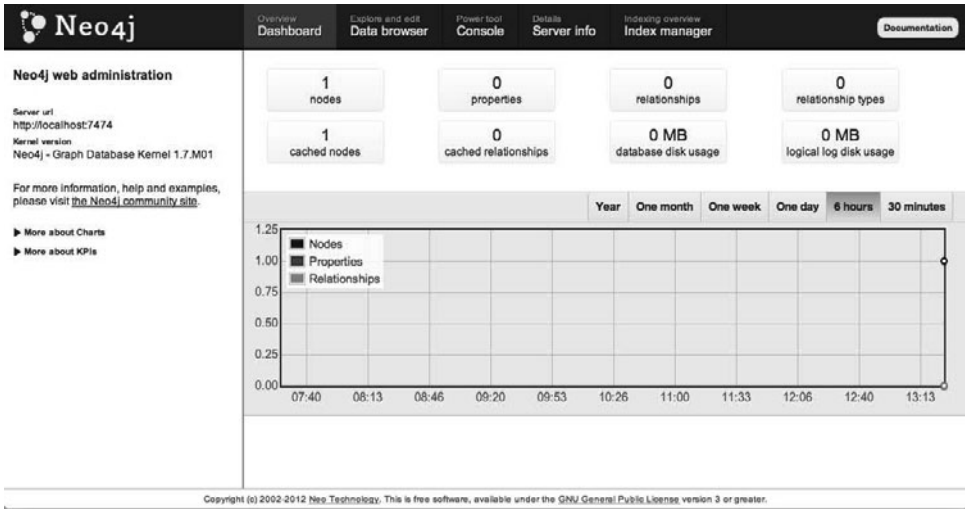


Abbildung 32: Die Web-Administrations-Seite

ausgezeichnet herumspielen kann. Und als ob das noch nicht genug wäre, besitzt es einen der coolsten Graph-Daten-Browser, den wir jemals gesehen haben. Das ist für den Einstieg perfekt, weil sich die Bewegung innerhalb eines Graphen beim ersten Mal recht beschwerlich gestalten kann.

Neo4js Web-Interface

Starten Sie einen Web-Browser und bewegen Sie sich auf die Administrations-Seite

```
http://localhost:7474/webadmin/
```

Sie werden mit einem farbenprächtigen, wenn auch leeren Graphen wie in Abbildung 32, *Die Web-Administrations-Seite* begrüßt. Klicken Sie oben auf Data Browser. Eine neue Neo4j-Installation besitzt einen vorgegebenen Referenzknoten: Knoten 0.

Ein *Knoten* (engl. node) in einer Graph-Datenbank ist den Knoten nicht ganz unähnlich, über die wir uns in den vergangenen Kapiteln unterhalten haben. Wenn wir in der Vergangenheit über einen Knoten gesprochen haben, meinten wir einen physikalischen Server in einem Netzwerk. Betrachtet man das gesamte Netzwerk als großen verbundenen Graphen, ist ein Server-Knoten ein Punkt (auch *vertex*) zwischen Server-*Beziehungen* oder -*kanten*.

Bei Neo4j ist ein Knoten konzeptionell vergleichbar. Er ist ein Punkt (Vertex) zwischen Kanten und kann Daten als Menge von Schlüssel/Wert-Paaren enthalten. Klicken Sie den „+ Property“-Button an und legen Sie den Schlüssel mit *name* und den Wert mit *Prancing Wolf Ice Wine 2007* fest, um einen bestimmten Wein und Jahrgang darzustellen. Klicken Sie dann den unten abgebildeten + Node-Button an:

Zu diesem neuen Knoten fügen Sie die Property (Eigenschaft) `name` mit dem Wert *Wine Expert Monthly* hinzu (abgekürzt formulieren wir das wie folgt: `[name : "Wine Expert Monthly"]`). Die Knotenzahl wird automatisch inkrementiert.

Nun haben wir zwei Knoten definiert, aber es gibt keine Verbindungen zwischen ihnen. Da Wine Expert etwas über den Wein Prancing Wolf geschrieben hat, wollen wir die beiden verknüpfen (eine Beziehung herstellen), indem wir eine Kante erzeugen. Klicken Sie den + Relationship-Button an und definieren Sie die Beziehung von Knoten 1 zu Knoten 0 mit dem Typ `reported_on`.

Sie erhalten eine URL zu dieser Beziehung

```
http://localhost:7474/db/data/relationship/0
```

der *Node 1 reported_on Node 0* zurückliefert.

Genau wie Knoten können auch Beziehungen Properties enthalten. Klicken Sie den „+ Add Property“-Button an und tragen Sie die Property `[rating : 92]` ein, damit wir nachhalten können, wie der Wein bewertet wurde.

Dieser spezielle Eiswein wurde aus einem *Riesling* gewonnen, und auch diese Information wollen wir festhalten. Wir könnten diese Eigenschaft direkt in den Wein-Knoten eintragen, doch Riesling ist eine allgemeine Kategorie, die auch für andere Weine gelten kann, weshalb wir einen weiteren Knoten mit der Property `[name : "riesling"]` anlegen. Nun fügen wir eine weitere Beziehung von Knoten 0 zu Knoten 2 hinzu. Wir nennen sie `grape_type` und geben ihr die Property `[style : "ice wine"]`.

Doch wie sieht unser Graph aus? Wenn Sie den „Ansicht wechseln“-Button (den etwas verschnörkelten Button neben + Relationship) anklicken, sehen Sie so etwas wie in Abbildung 33, *Graph von Knoten, die mit dem aktuellen in Beziehung stehen*, auf Seite 242.

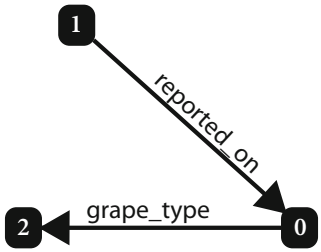


Abbildung 33: Graph von Knoten, die mit dem aktuellen in Beziehung stehen

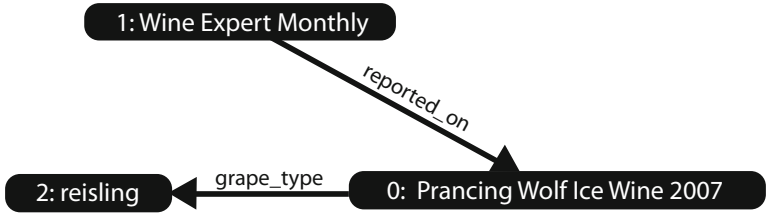


Abbildung 34: Graph mit eigenem Profil

Der Style-Button öffnet ein Menü, in dem Sie das Profil (Profile) wählen können, das zum Rendern des Graphen genutzt wird. Um etwas nützlichere Informationen im Diagramm darzustellen, klicken Sie auf Style und dann auf New Profile. Das bringt Sie auf die „Create new visualization profile"-Seite. Geben Sie oben den Namen *wines* ein und ändern Sie das Label von `{id}` in `{id}: {prop.name}`. Ein Klick auf Save bringt Sie zurück auf die Visualisierungseite. Jetzt können Sie *wines* aus dem Style-Menü wählen und erhalten so etwas wie in Abbildung 34, *Graph mit eigenem Profil*.

Zwar stellt das Web-Interface eine einfache Möglichkeit dar, ein paar Dinge zu bearbeiten, aber für eine produktive Arbeit benötigen wir ein leistungsfähigeres Interface.

Neo4j via Gremlin

Es gibt verschiedene Sprachen, die mit Neo4j zusammenarbeiten: Java-Code, REST, Cypher, Ruby Console und andere. Wir wollen heute eine namens Gremlin verwenden, eine in Groovy geschrieben Sprache zur Traversierung von Graphen. Sie müssen aber Groovy nicht kennen, um Gremlin nutzen zu können, also stellen Sie sich das einfach als eine weitere domänenspezifische Sprache (wie SQL) vor.

Wie andere von uns untersuchte Consolen bietet Gremlin Zugriff auf die Sprach-Infrastruktur, auf der sie basiert. Das bedeutet, dass Sie Groovy-Konstrukte und Java-Bibliotheken in Gremlin nutzen können. Wir haben das als natürlichere Form der Interaktion mit Graphen empfunden als Neo4js nativen Java-Code. Und noch besser: Die Gremlin Console ist auch im Web-Admin verfügbar. Klicken Sie einfach oben den Console-Link an und wählen Sie Gremlin.

Per Konvention ist `g` eine Variable, die ein Graph-Objekt repräsentiert. Graph-*Actions* (also Aktionen) sind Funktionen, die wir auf das Objekt anwenden.

Da Gremlin eine Allzwecksprache zur Graph-Traversierung ist, nutzt sie allgemeine mathematische Graphen-Begriffe. Wenn Neo4j einen Graph-Datenpunkt als *Knoten* bezeichnet, zieht Gremlin den Begriff *Vertex* vor und anstelle einer *Beziehung* spricht Gremlin von einer *Kante* (edge).

Um auf alle Knoten in diesem Graphen zuzugreifen, gibt es eine Property namens `V` für „vertices“

```
gremlin> g.V
==>v[0]
==>v[1]
==>v[2]
```

sowie einer verwandten Property namens `E` ("edges") für die Kanten.

```
gremlin> g.E
==> e[0][1-reported_on->0]
==> e[1][0-grape_type->2]
```

Sie greifen auf einen bestimmten Knoten zu, indem Sie die Knotennummer an die `v`-Methode (kleines v) übergeben.

```
gremlin> g.v(0)
==> v[0]
```

Um sicher zu sein, dass Sie den richtigen Knoten haben, können Sie sich seine Properties über die `map`-Methode ansehen. Beachten Sie, dass Sie bei Groovy/Gremlin Methodenaufrufe verketten können.

```
gremlin> g.v(0).map()
==> name=Prancing Wolf Ice Wine 2007
```

Mit `v(0)` können wir zwar den exakten Knoten abrufen, aber wir können auch alle Knoten nach einem gewünschten Wert filtern. Um zum Beispiel *riesling* über den Namen abzurufen, können wir die Filtersyntax {. . . } nutzen, die bei

Groovy-Code als *Closure* bezeichnet wird. Der gesamte Code zwischen den geschweiften Klammern definiert die Funktion, die, wenn wahr zurückgegeben wird, diesen Knoten durchgeht. Das Schlüsselwort `it` innerhalb der Closure repräsentiert das aktuelle Objekt und wird automatisch befüllt.

```
gremlin> g.V.filter{it.name=='riesling'}
==> v[2]
```

Sobald Sie einen Knoten haben, können Sie die ausgehenden Kanten abrufen, indem Sie `outE` auf ihn anwenden. Eingehende Kanten können entsprechend über `inE` und beide über `bothE()` abgerufen werden.

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE()
==> e[0][1-reported_on->0]
```

Beachten Sie, dass bei Groovy, genau wie bei Ruby, Klammern für Methoden optional sind, d. h., Sie können `outE` auch einfach so aufrufen:

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE
==> e[0][1-reported_on->0]
```

Von den ausgehenden Kanten können Sie sich zu den eingehenden Knoten mit `inV` bewegen, d. h. zu dem Knoten, auf den die Kante zeigt. Die Kante `reported_on` von Wine Expert zeigt auf den Knoten *Prancing Wolf Ice Wine 2007*, was `outE.inV` auch zurückliefert. Die `name`-Property können Sie dann über den Knoten abrufen.

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE.inV.name
==> Prancing Wolf Ice Wine 2007
```

Der Ausdruck `outE.inV` fragt nach allen Knoten, zu denen die Eingangsknoten Kanten besitzen. Die Umkehroperation (alle Knoten, bei denen Kanten *in* die Eingangsknoten laufen) erreichen Sie mit `inE.outV`. Weil diese Operationen so gängig sind, bietet Gremlin für beide Kürzel an. Der Ausdruck `out` ist das Kürzel für `outE.inV` und `in` das Kürzel für `inE.outV`.

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.out.name
==> Prancing Wolf Ice Wine 2007
```

Ein Weingut stellt mehr als einen Wein her, d. h., wenn wir weitere hinzufügen wollen, sollten wir das Weingut als verbindenden Knoten einfügen und eine Kante zum Prancing Wolf herstellen.

```
gremlin> pwolf = g.addVertex([name : 'Prancing Wolf Winery'])
==> v[3]
gremlin> g.addEdge(pwolf, g.v(0), 'produced')
==> e[2][3-produced->0]
```

Nun können wir weitere Rieslinge einfügen: Kabinett und Spätlese.

```
gremlin> kabinett = g.addVertex([name : 'Prancing Wolf Kabinett 2002'])
==> v[4]
gremlin> g.addEdge(pwolf, kabinett, 'produced')
==> e[3][3-produced->4]
gremlin> spatlese = g.addVertex([name : 'Prancing Wolf Spatlese 2007'])
==> v[5]
gremlin> g.addEdge(pwolf, spatlese, 'produced')
==> e[4][3-produced->5]
```

Wir wollen unseren kleinen Graphen erweitern, indem wir einige Kanten vom Riesling-Knoten zu den neu hinzugefügten Knoten herstellen. Wir setzen die riesling-Variable, indem wir den riesling-Knoten filtern. `next()` wird benötigt, um den ersten Knoten aus der Pipeline abzurufen (worauf wir gleich genauer eingehen).

```
gremlin> riesling = g.V.filter{it.name=='riesling'}.next()
==> v[2]
gremlin> g.addEdge([style:'kabinett'], kabinett, riesling, 'grape_type')
==> e[5][4-grape_type->2]
```

Die Spätlese kann in ähnlicher Weise auf den Riesling verweisen, doch der `style` wird auf `spatlese` gesetzt. Haben wir all diese Daten hinzugefügt, sollte Ihr Graph im Visualizer so aussehen wie in Abbildung 35, *Graph nach dem Hinzufügen von Daten mit Gremlin*.

Abbildung 35: Graph nach dem Hinzufügen von Daten mit Gremlin

Die Macht der Pipes

Sie können sich Gremlin-Operationen als eine Folge von Pipes vorstellen. Jede Pipe nimmt eine Collection als Eingabe und liefert eine Collection als Ausgabe. Eine Collection kann ein Element, viele Elemente oder auch gar kein Element enthalten. Die Elemente können Knoten, Kanten oder Property-Werte sein.

Zum Beispiel nimmt die `outE`-Pipe eine Collection von Knoten und gibt eine Collection von Kanten weiter. Diese Folge von Pipes wird als *Pipeline* bezeichnet und drückt das Problem *deklarativ* aus. Vergleichen Sie das mit einem typischen *imperativen* Programmier-Ansatz, bei dem Sie die Schritte beschreiben müssen, die zur Lösung des Problems notwendig sind. Die Verwendung von Pipes ist eine der kompaktesten Möglichkeiten, eine Graph-Datenbank abzufragen.

Im Innersten ist Gremlin eine Sprache, mit der solche Pipes aufgebaut werden können. Genauer gesagt baut sie auf einem Java-Projekt namens Pipes auf. Um das Pipe-Konzept zu untersuchen, wollen wir zu unserem Wein-Graphen zurückkehren. Nehmen wir an, Sie wollen Weine finden, die einem gegebenen Wein entsprechen – d. h. von der gleichen Sorte sind. Wir können einem Eiswein folgen, der ebenfalls eine `grape_type`-Kante zu anderen Knoten enthält (wobei wir den Anfangsknoten ignorieren).

```
ice_wine = g.v(0)
ice_wine.out('grape_type').in('grape_type').filter{ !it.equals(ice_wine) }
```

Wenn Sie bei Smalltalk oder Rails mit Geltungsbereichen gearbeitet haben, wird Ihnen diese Form der Methodenverkettung vertraut vorkommen. Vergleichen Sie nun das obige Beispiel mit der nachfolgenden, normalen Neo4j Java-API, bei der man die Beziehungen der Knoten durchgehen muss, um auf die Sorten-Knoten zugreifen zu können.

```
enum WineRelationshipType implements RelationshipType {
  grape_type
}

import static WineRelationshipType.grape_type;

public static List<Node> same_variety( Node wine ) {
  List<Node> wine_list = new ArrayList<Node>();
  // Alle ausgehenden Kanten dieses Knotens durchgehen
  for( Relationship outE : wine.getRelationships( grape_type ) ) {
    // Alle eingehenden Kanten des ausgehenden Knotens dieser Kante durchgehen
    for( Edge inE : outE.getEndNode().getRelationships( grape_type ) ) {
      // Nur Knoten aufnehmen, die nicht dem angegebenen Knoten entsprechen
      if( !inE.getStartNode().equals( wine ) ) {
        wine_list.add( inE.getStartNode() );
      }
```

```
    }
  }
  return wine_list;
}
```

Anstelle der obigen Verschachtelung und Iteration entwickelte das Pipes-Projekt eine Möglichkeit, ein- und ausgehende Knoten zu deklarieren. Sie bauen eine Folge ein- und ausgehender Pipes, Filter und Request-Werte aus der Pipeline auf. Dann rufen Sie iterativ die `hasNext`-Methode der Pipeline auf, die den nächsten passenden Knoten zurückliefert. Mit anderen Worten: Die Pipeline geht den Baum für Sie durch. Bis die Pipeline angefordert wird, deklarieren Sie nur, wie der Durchlauf erfolgen soll.

Jim sagt: jQuery und Gremlin

Nutzern der beliebten JavaScript-Bibliothek jQuery wird diese Collection-orientierte Traversierung vertraut vorkommen. Betrachten Sie das folgende HTML-Fragment:

```
<ul id="navigation">
  <li>
    <a name="section1">section 1</a>
  </li>
  <li>
    <a name="section2">section 2</a>
  </li>
</ul>
```

Nehmen wir nun an, wir wollen den Text aller Tags mit dem Namen `section1` finden, die Child-Elemente von Listenelementen (`li`) unterhalb des Navigationselements sind (`id=navigation`). Eine mögliche Lösung ist jQuery-Code wie dieser:

```
$('[id=navigation]').children('li').children('[name=section1]').text()
```

Betrachten Sie nun eine Gremlin-Query, die nach einem ähnlichen Datensatz sucht, wobei wir annehmen, dass jeder Parent-Knoten eine Kante besitzt, die auf jedes Child-Element verweist. Ganz schön ähnlich, nicht wahr?

```
g.V.filter{it.id=='navigation'}.out.filter{it.tag=='li'}.
out.filter{it.name=='section1'}.text
```

Um das zu verdeutlichen, hier eine weitere Implementierung der `same_variety`-Methode, die Pipes anstelle von Schleifen nutzt:

```
public static void same_variety( Vertex wine ) {
  List<Vertex> wine_list = new ArrayList<Vertex>();
  Pipe inE       = new InPipe( "grape_type" );
  Pipe outE      = new OutPipe( "grape_type" );
```

```
  Pipe not_wine = new ObjectFilterPipe( wine, true );
  Pipe<Vertex,Vertex> pipeline =
    new Pipeline<Vertex,Vertex>( outE, inE, not_wine );
  pipeline.setStarts( Arrays.asList( wine ) );
  while( pipeline.hasNext() ) {
    wine_list.add( pipeline.next() );
  }
  return wine_list;
}
```

Tief im Inneren ist Gremlin eine `Pipe`-bauende Sprache. Das Durchgehen des Graphen erfolgt immer noch auf dem Neo4j-Server, aber Gremlin vereinfacht den Aufbau von Queries, die Neo4j versteht.

Pipeline vs. Vertex

Um eine Collection abzurufen, die nur eine bestimmte Vertex enthält, können wir sie aus der Liste aller Knoten herausfiltern. Genau das haben wir zum Beispiel bei `g.V.filter{it.name=='reisling'}` getan. Die `V`-Property ist eine Liste aller Knoten, aus der wir eine Subliste abrufen. Doch wenn wir den Knoten selbst wollen, müssen wir `next` aufrufen. Diese Methode ruft die erste Vertex aus der Pipeline ab. Das ähnelt dem Unterschied zwischen einem Array mit einem Element und dem Element selbst.

Eric sagt:
Cypher

Cypher ist eine weitere von Neo4j unterstützte Graphen-Abfragesprache. Sie basiert auf Mustererkennung (pattern matching) und einer SQL-ähnlichen Syntax. Die Klauseln fühlen sich vertraut an, was es einfacher macht, zu verstehen, was vorgeht. Insbesondere die `MATCH`-Klausel ist sehr intuitiv und führt zu Ausdrücken, die an ASCII-Art erinnern.

Zuerst habe ich Cyphers Langatmigkeit nicht gemocht, doch mit der Zeit haben sich meine Augen an das Lesen der Grammatik gewöhnt und ich bin zum Fan geworden.

Sehen wir uns das Cypher-Äquivalent unserer „gleiche Weine"-Query an:

```
START ice_wine=node(0)
MATCH (ice_wine) -[:grape_type]-> () <-[:grape_type]- (similar)
RETURN similar
```

Wir beginnen damit, `ice_wine` an den Knoten 0 zu binden. Die `MATCH`-Klausel verwendet Identifier innerhalb runder Klammern für Knoten und „Pfeile" wie `-[:grape_type]->` für gerichtete Beziehungen. Ich mag dieses Konstrukt, da es einfach ist, die Traversierung der Knoten zu visualisieren.

Doch es kann schnell anspruchsvoll werden. Hier ein etwas praxisgerechteres Beispiel – jeder Teil so leistungsfähig und wortreich wie SQL.

```
START ice_wine=node:wines(name="Prancing Wolf Ice Wine 2007")
MATCH ice_wine -[:grape_type]-> wine_type <-[:grape_type]- similar
WHERE wine_type =~ /(?i)riesl.*/
RETURN wine_type.name, collect(similar) as wines, count(*) as wine_count
ORDER BY wine_count desc
LIMIT 10
```

Während ich mich im Kapitel selbst auf Gremlin konzentriere, ergänzen sich die beiden Sprachen auf natürliche Weise und können freudig nebeneinander koexistieren. In der täglichen Arbeit werden Sie Verwendung für beide finden, je nachdem, wie Sie das jeweilige Problem angehen.

Wenn Sie die Klasse betrachten, die durch den Aufruf der `class`-Property des Filters erzeugt wird, werden Sie sehen, dass sie `GremlinPipeline` zurückgibt.

```
gremlin> g.V.filter{it.name=='Prancing Wolf Winery'}.class
==>class com.tinkerpop.gremlin.pipes.GremlinPipeline
```

Vergleichen Sie das mit den Klasse des nächsten Knotens aus der Pipeline. Hier sehen Sie etwas anderes, nämlich `Neo4jVertex`.

```
gremlin> g.V.filter{it.name=='Prancing Wolf Winery'}.next().class
==>class com.tinkerpop.blueprints.pgm.impls.neo4j.Neo4jVertex
```

Auch wenn die Console die von der Pipeline abgerufenen Knoten ausgibt, bleibt sie doch eine Pipeline, bis Sie etwas aus ihr abrufen.

Schemafrei sozial

Um einen sozialen Aspekt im Graphen zu erzeugen, müssen Sie nur weitere Knoten einfügen. Nehmen wir an, wir wollen drei Leute mit ihren Wein-Vorlieben hinzufügen – zwei die einander kennen und einen Fremden.

Alice ist eine Naschkatze und daher ein großer Fan von Eiswein.

```
alice = g.addVertex([name:'Alice'])
ice_wine = g.V.filter{it.name=='Prancing Wolf Ice Wine 2007'}.next()
g.addEdge(alice, ice_wine, 'likes')
```

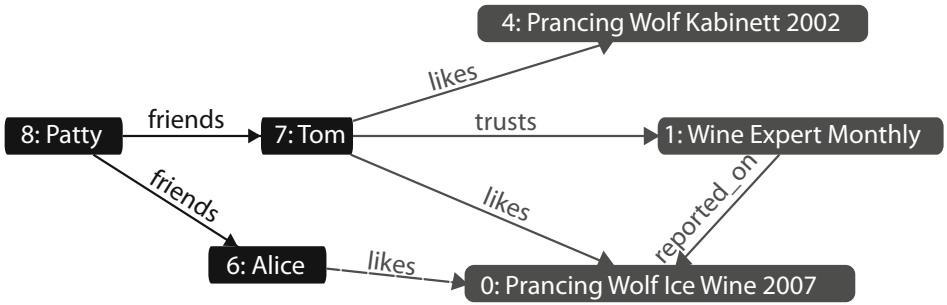
Tom liebt Kabinett und Eiswein und glaubt alles, was im *Wine Expert Monthly* steht.

```
tom = g.addVertex([name:'Tom'])
kabinett = g.V.filter{it.name=='Prancing Wolf Kabinett 2002'}.next()
g.addEdge(tom, kabinett, 'likes')
g.addEdge(tom, ice_wine, 'likes')
g.addEdge(tom, g.V.filter{it.name=='Wine Expert Monthly'}.next(), 'trusts')
```

Patty ist mit Tom und mit Alice befreundet, ist aber erst in jüngster Zeit zum Wein gekommen und muss sich ihre Favoriten noch auswählen.

```
patty = g.addVertex([name:'Patty'])
g.addEdge(patty, tom, 'friends')
g.addEdge(patty, alice, 'friends')
```

Ohne die grundlegende Struktur unseres existierenden Graphen zu verändern, konnten wir ihn um Verhaltensweisen (behavior) ergänzen, was weit über unsere ursprünglichen Absichten hinausgeht. Die neuen Knoten stehen zueinander in Beziehung, wie nachfolgend zu sehen:



Stepping Stones

Wir haben uns einige Gremlin-Kern*schritte*, oder Pipe-Verarbeitungseinheiten, angesehen. Gremlin bietet noch viele mehr. Wir wollen uns noch mehr dieser Bausteine ansehen, die den Graphen nicht nur durchgehen, sondern auch Objekte transformieren, Schritte filtern und Nebeneffekte erzeugen wie das Zählen von Knoten, die nach bestimmten Kriterien gruppiert sind.

Wir haben `inE`, `outE`, `inV` und `outV` kennengelernt, bei denen es sich um *Transformationsschritte* zum Abruf ein- und ausgehender Kanten und Knoten handelt. Zwei weitere Arten sind `bothE()` und `bothV()`, die einfach einer Kante folgen, egal in welcher Richtung sie verläuft.

Nachfolgend rufen wir Alice und all ihre Freunde ab. Wir hängen `name` an das Ende an, um an die `name`-Property jedes Knotens zu gelangen. Da es uns nicht kümmert, in welche Richtung die `friend`-Kante läuft, verwenden wir `bothE` und `bothV`.

```
alice.bothE('friends').bothV.name

==> Alice
==> Patty
```

Wenn uns Alice nicht interessiert, können wir dem `except()`-Filter eine Liste der Knoten übergeben, die wir nicht wünschen.

```
alice.bothE('friends').bothV.except([alice]).name

==> Patty
```

Das Gegenstück zu `except()` ist `retain()`, das (wie Sie sich wohl denken können) nur passende Knoten durchgeht.

Eine andere Option besteht darin, die letzte Vertex stattdessen mit einem Codeblock herauszufiltern, der prüft, dass der aktuelle Schritt nicht der `alice`-Vertex entspricht.

```
alice.bothE('friends').bothV.filter{!it.equals(alice)}.name
```

Und wenn wir nun alle Freunde von Alices Freunden suchen? Dann können wir die Schritte einfach wie folgt wiederholen:

```
alice.bothE('friends').bothV.except([alice]).
bothE('friends').bothV.except([alice])
```

Auf die gleiche Weise könnten wir die Freunde der Freunde von Alices Freunden aufspüren, indem wir weitere `bothE/bothV/except`-Aufrufe hinzufügen. Aber dann muss man sehr viel eintippen und man kann das bei einer variablen Anzahl von Schritten nicht auf diese Weise machen. Hier hilft uns die `loop`-Methode. Sie wiederholt eine Reihe von Schritten, solange die gegebene Closure wahr ist.

Der folgende Code wiederholt die drei vorangegangenen Schritte, indem er die Punkte vom loop-Aufruf zurückzählt. `except` ist also der erste, `bothV` der zweite und `bothE` der dritte Schritt.

```
alice.bothE('friends').bothV.except([alice]).loop(3){
  it.loops >= 2
}.name
```

Nachdem die Schritte durchlaufen wurden, ruft `loop` die angegebene Closure auf – d. h. den Code zwischen den geschweiften Klammern `{...}` In unserem Beispiel hält die `it.loops`-Property nach, wie oft die aktuelle Schleife ausgeführt wurde. Wir prüfen, ob dieser Wert größer oder gleich zwei ist, d. h., die Schleife wird zweimal ausgeführt und dann beendet. Tatsächlich verhält sich die Closure wie die Klausel einer `while`-Schleife bei einer typischen Programmiersprache.

```
==>Tom
==>Patty
==>Patty
```

Die Schleife hat korrekt funktioniert und Tom und Patty gefunden. Doch jetzt taucht Patty zweimal auf. Das liegt daran, dass Patty einmal als Freundin von Alice erkannt wird und ein weiteres Mal als Freundin von Tom. Wir brauchen also eine Möglichkeit, doppelt vorkommende Objekte herauszufiltern, was der Filter `dedup()` ("de-duplicate") macht.

```
alice.bothE('friends').bothV.except([alice]).loop(3){
  it.loops <= 2
}.dedup.name

==>Tom
==>Patty
```

Wenn Sie wissen wollen, welcher Pfad verfolgt wird, um an diese Werte zu gelangen, können Sie den friend->friend-Pfad über die `paths`-Transformation verfolgen.

```
alice.bothE('friends').bothV.except([alice]).loop(3){
  it.loops <= 2
}.dedup.name.paths

==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[8], Tom]
==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[9], Patty]
```

Bei allen Traversierungen haben wir uns bisher nur vorwärts durch einen Graphen bewegt. Manchmal muss man aber zwei Schritte vor gehen und dann wieder zwei Schritte zurück. Beginnend beim Alice-Knoten gehen wir nachfolgend zwei Schritte vor und dann wieder zwei Schritte zurück, was uns zum Alice-Knoten zurückbringt.

```
gremlin> alice.outE.inV.back(2).name
==> Alice
```

Der letzte gängige Schritt, den wir untersuchen wollen, ist `groupCount()`. Er geht die Knoten durch, zählt alle doppelt vorkommenden Werte und hält sie in einer Map fest.

Im folgenden Beispiel werden die `name`-Properties aller Knoten im Graph gesammelt und gezählt:

```
gremlin> name_map = [:]
gremlin> g.V.name.groupCount( name_map )
gremlin> name_map
```

```
==> Prancing Wolf Ice Wine 2007=1
==> Wine Expert Monthly=1
==> riesling=1
==> Prancing Wolf Winery=1
==> Prancing Wolf Kabinett 2002=1
==> Prancing Wolf Spatlese 2007=1
==> Alice=1
==> Tom=1
==> Patty=1
```

Bei Groovy/Gremlin wird eine Map über die Nomenklatur `[:]` angegeben und entspricht dem Ruby/JavaScript-Objektliteral `{}`. Beachten Sie, dass alle Werte 1 sind. Das ist auch genau das, was wir erwarten, weil wir keine Namen wiederholt haben und die `V`-Collection genau eine Kopie jedes Knotens unseres Graphen besitzt.

Nun wollen wir die Weine zählen, die jede Person in unserem System mag. Wir können alle liked-Knoten abrufen und nach Namen zählen.

```
gremlin> wines_count = [:]
gremlin> g.V.outE('likes').outV.name.groupCount( wines_count )
gremlin> wines_count
==> Alice=1
==> Tom=2
```

Wie erwartet, mag Alice einen und Tom zwei Weine.

Groovy

Neben den Gremlin-Schritten steht uns auch das weite Feld der Groovy-Sprachkonstrukte und -Methoden zur Verfügung. Groovy besitzt eine map-Funktion (à la Mapreduce) namens `collect()` und eine Reduce-Funktion namens `inject()`. Mit Hilfe dieser beiden können wir Mapreduce-artige Queries durchführen.

Nehmen wir an, Sie wollen wissen, wie viele Weine noch nicht bewertet wurden. Dazu können wir zuerst eine Liste mit Wahr/Falsch-Werten mappen, die zeigt, ob ein Wein schon bewertet wurde. Dann können wir diese Liste durch einen Reducer laufen lassen, um alle Wahr/Falsch-Werte aufzusummieren. Der Mapping-Teil nutzt `collect` und sieht wie folgt aus:

```
rated_list = g.V.in('grape_type').collect{
  !it.inE('reported_on').toList().isEmpty()
}
```

Im obigen Code gibt der Ausdruck `g.V.in('grape_type')` alle Knoten zurück, die eine eingehende `grape_type`-Beziehung besitzen. Nur Weine besitzen diesen Kantentyp, d. h., unsere Liste enthält alle Weine unseres Systems. Als

Nächstes ermitteln wir in unserer `collect`-Closure, ob der fragliche Wein eingehende `reported_on`-Kanten besitzt. Der `toList`-Aufruf macht aus der Pipeline eine echte Liste. Wir können dann prüfen, ob sie leer ist. Die von diesem Code erzeugte `rated_list` besteht aus Wahr/Falsch-Werten.

Um zu zählen, wie viele Weine nicht bewertet wurden, lassen wir diese Liste über die `inject`-Methode durch einen Reducer laufen.

```
rated_list.inject(0){ count, is_rated ->
  if (is_rated) {
    count
  } else {
    count + 1
  }
}
```

Bei Groovy trennt der Pfeiloperator (`->`) die Eingabeargumente für eine Closure von deren Rumpf. Unser Reducer muss den akkumulierten Zähler nachhalten und entscheiden, ob der aktuelle Wein schon bewertet wurde oder nicht. Das ist der Grund für `count` und `is_rated`. Der `0`-Teil von `inject(0)` initialisiert `count` vor dem ersten Aufruf mit 0. Innerhalb des Bodies der Closure-Funktion liefern wir dann entweder den aktuellen Zähler zurück (wenn der Wein bereits bewertet wurde) oder erhöhen ihn um 1 (wenn er noch nicht bewertet wurde). Das Endergebnis ist die Zahl der Falsch-Werte der Liste (also die Zahl der nicht bewerteten Weine).

```
==> 2
```

Es stellt sich heraus, dass zwei Weine noch nicht bewertet wurden.

Dank dieser vielen zur Verfügung stehenden Tools können Sie mächtige Kombinationen aus Graph-Traversierungen und -Transformationen durchführen. Nehmen wir an, Sie möchten alle Freundes-Paare in Ihrem Graphen finden. Dazu müssen wir zuerst alle Kanten vom Typ `friends` finden und dann über eine `transform`-Operation die Namen der beiden Personen ausgeben, die sich diese Kante teilen.

```
g.V.outE('friends').transform{[it.outV.name.next(), it.inV.name.next()]}

==> [Patty, Tom]
==> [Patty, Alice]
```

Im obigen Code ist der Rückgabewert der `transform`-Closure ein Array-Literal (`[...]`) mit zwei Elementen: den Ein- und Ausgangsknoten der `friend`-Kante.

Um alle Personen und die Weine, die sie mögen, auszugeben, transformieren wir die Ausgabe der Personen (die als Knoten mit Freunden gekennzeichnet sind) in eine Liste mit zwei Elementen: dem Namen der Person und einer Liste der Weine, die sie mag.

```
g.V.both('friends').dedup.transform{
  [ it.name, it.out('likes').name.toList() ]
}

==> [Alice, [Prancing Wolf Ice Wine 2007]]
==> [Patty, []]
==> [Tom, [Prancing Wolf Ice Wine 2007, Prancing Wolf Kabinett 2002]]
```

Ja, an Gremlin muss man sich definitiv erst gewöhnen, insbesondere wenn man vorher noch nie in Groovy programmiert hat. Sobald Sie sich aber daran gewöhnt haben, verfügen Sie über eine ausdrucksstarke und mächtige Möglichkeit, Neo4j-Queries auszuführen.

Domänenspezifische Schritte

Die Traversierung von Graphen ist nett, aber Unternehmen und Organisationen neigen dazu, sich über domänenspezifische Sprachen miteinander zu unterhalten. Wir würden normalerweise zum Beispiel nicht fragen: „Welcher Knoten der eingehenden Kante von grape_type teilt sich die ausgehende Kante dieses Wein-Knotens?“, sondern einfach: „Aus welcher Rebsorte wurde dieser Wein gekeltert?“

Gremlin ist bereits eine domänenspezifische Sprache zur Abfrage von Graph-Datenbanken, doch wie wäre es, wenn wir die Sprache noch spezifischer machen würden? Gremlin erlaubt uns das durch die Entwicklung neuer Schritte, die für die im Graph gespeicherten Daten eine semantische Bedeutung haben.

Wie wollen mit einem neuen Schritt namens `varietal` (Rebsorte) beginnen, der uns die Antwort auf die obige Frage liefert. Wird `varietal` für einen Knoten aufgerufen, sieht sie sich die ausgehenden Kanten vom Typ `grape_type` an und geht die verbundenen Knoten durch.

Wir werden hier ein wenig Groovy-lastig, weshalb wir hier zuerst den Code abbilden und ihn dann Schritt für Schritt erläutern.

neo4j/varietal.groovy

```
Gremlin.defineStep( 'varietal',
  [Vertex, Pipe],
  {_().out('grape_type').dedup}
)
```

Zuerst teilen wir der Gremlin-Engine mit, dass wir einen neuen Schritt namens `varietal` einfügen. Die zweite Zeile weist Gremlin an, die `Vertex`- und `Pipe`-Klassen einzubinden. Die letzte Zeile sorgt für den magischen Moment. Tatsächlich erzeugt sie die Closure, die den Code enthält, die dieser Schritt ausführen soll. Der Unterstrich und die Klammern repräsentieren das aktuelle Pipeline-Objekt. Von diesem Objekt aus gehen wir alle Nachbarknoten durch, die über einen `grape_type`-Knoten miteinander verbunden sind, d. h. die Rebsorten-Knoten. Wir schließen die Sache mit `dedup` ab, um mögliche Duplikate zu entfernen.

Der Aufruf unseres neuen Schrittes erfolgt wie bei jedem anderen Schritt auch. Zum Beispiel liefert die folgende Zeile den Namen der Rebsorte des Eisweins zurück:

```
g.V.filter{it.name=='Prancing Wolf Ice Wine 2007'}.varietal.name

==> riesling
```

Probieren wir etwas anderes. Diesmal wollen wir einen Schritt entwickeln, der eine häufig gestellte Frage beantwortet: Welchen Wein mögen die Freunde am liebsten?

neo4j/friendsuggest.groovy

```
Gremlin.defineStep( 'friendsuggest',
  [Vertex, Pipe],
  {
    _().sideEffect{start = it}.both('friends').
    except([start]).out('likes').dedup
  }
)
```

Genau wie beim letzten Mal übergeben wir Gremlin den Namen des neuen Schritts (`friendsuggest`) und binden es an `Vertex` und `Pipe`. Diesmal filtert unser Code die aktuelle Person heraus. Dazu legen wir die aktuelle Vertex/Pipe mit Hilfe der Funktion `sideEffect{start = it}` in einer Variablen (start) ab. Dann rufen wir alle `friends`-Knoten ab, wobei wir die aktuelle Person ausschließen (Alice soll nicht in der Liste Ihrer eigenen Freunde stehen).

Nun brauen wir uns etwas mit Pipes zusammen! Wir können den neuen Schritt aufrufen, wie jeden anderen auch.

```
g.V.filter{it.name=='Patty'}.friendsuggest.name

==> Prancing Wolf Ice Wine 2007
==> Prancing Wolf Kabinett 2002
```

Weil `varietal` und `friendsuggest` ganz normale `Pipe`-generierende Schritte sind, können wir sie verketten und weitere interessante Queries aufbauen. Die folgende Zeile findet die Rebsorten, die Pattys Freunde am liebsten mögen:

```
g.V.filter{it.name=='Patty'}.friendsuggest.varietal.name

==> riesling
```

Die Nutzung der Groovy-Metaprogrammierung zur Entwicklung neuer Schritte ist ein mächtiges Werkzeug zur Entwicklung domänenspezifischer Sprachen. Doch genau wie bei Gremlin kann es in der Praxis ein wenig dauern, bis man sich daran gewöhnt hat.

Update, Delete, Done

Wir haben Elemente in den Graphen eingefügt und sind ihn durchgegangen, doch wie sieht es mit der Aktualisierung und dem Löschen von Daten aus? Das ist ganz einfach, sobald man den Knoten bzw. die Kante gefunden hat, die man ändern will. Wir wollen eine Gewichtung einfügen, die angibt, wie sehr Alice den Prancing Wolf Ice Wine 2007 mag.

```
gremlin> e=g.V.filter{it.name=='Alice'}.outE('likes').next()
gremlin> e.weight = 95
gremlin> e.save
```

Den Wert zu entfernen, ist ganz einfach.

```
gremlin> e.removeProperty('weight')
gremlin> e.save
```

Bevor wir den Tag beenden und uns dem Selbststudium zuwenden, müssen wir noch darüber reden, wie man eine Datenbank bereinigt.

Führen Sie diese Befehle nicht aus, bevor Sie die heutigen Hausaufgaben erledigt haben!

Das Graph-Objekt kennt Funktionen zum Entfernen von Knoten und Kanten (`removeVertex` bzw. `removeEdge`). Wir können unseren Graphen entfernen, indem wir all seine Knoten und Kanten löschen.

```
gremlin> g.V.each{ g.removeVertex(it) }
gremlin> g.E.each{ g.removeEdge(it) }
```

Sie können überprüfen, ob er wirklich weg ist, indem Sie `g.V` und `g.E` aufrufen. Sie erreichen das Gleiche aber auch mit der sehr gefährlichen `clear`-Methode.

```
gremlin> g.clear()
```

Wenn Sie eine eigene Gremlin-Instanz (außerhalb der Web-Schnittstelle) ausführen, sollten Sie die Graph-Verbindung mit der `shutdown`-Methode sauber herunterfahren.

```
gremlin> g.shutdown()
```

Falls Sie das nicht tun, könnte die Datenbank beschädigt werden. Doch normalerweise wird nur mit Ihnen geschimpft, wenn Sie das nächste Mal die Verbindung herstellen.

Was wir am ersten Tag gelernt haben

Heute haben wir einen ersten Blick auf die Graph-Datenbank Neo4j geworfen – und was für ein andersartiges Biest sie ist. Obwohl wir keine Entwurfsmuster diskutiert haben, brummten unsere Schädel vor lauter Möglichkeiten, als wir damit begannen, mit Neo4j zu arbeiten. Wenn Sie etwas auf ein Whiteboard zeichnen können, können Sie es auch in einer Graph-Datenbank speichern.

Tag 1: Selbststudium

Finden Sie heraus

1. Fügen Sie das Neo4j-Wiki zu Ihren Lesezeichen hinzu.
2. Fügen Sie die Gremlin-Schritte aus dem Wiki oder der API zu Ihren Lesezeichen hinzu.
3. Finden Sie zwei weitere Neo4j-Shells (etwa die Cypher-Shell in der Admin-Console).

Machen Sie Folgendes

1. Fragen Sie die Namen aller Knoten mit einer anderen Shell ab (wie der Cypher-Abfragesprache).
2. Löschen Sie alle Knoten und Kanten aus Ihrer Datenbank.
3. Entwickeln Sie einen neuen Graphen, der Ihre Familie repräsentiert.

7.3 Tag 2: REST, Indizes und Algorithmen

Heute fangen wir mit Neo4js REST-Interface an. Wir werden Knoten und Beziehungen über REST aufbauen und dann REST zur Indexierung und zur Volltextsuche nutzen. Wir sehen uns ein Plugin an, mit dessen Hilfe Gremlin-

Queries über REST auf dem Server ausgeführt werden können, was unseren Code von den Fesseln der Gremlin-Console befreit. Selbst Java lässt sich auf dem Anwendungs-Server oder den Clients ausführen.

REST

Genau wie Riak, HBase, Mongo und CouchDB besitzt auch Neo4j eine REST-Schnittstelle. Ein Grund dafür, dass all diese Datenbanken REST unterstützen, besteht darin, dass es sprachunabhängige Interaktionen über eine Standardschnittstelle erlaubt. Wir können die Verbindung mit Neo4j (das Java braucht, um zu funktionieren) von einer separaten Maschine aus herstellen, ohne in irgendeiner Form an Java gebunden zu sein. Und mit dem Gremlin-Plugin können wir die Leistungsfähigkeit seiner kompakten Query-Syntax über REST nutzen.

Zuerst sollten Sie prüfen, ob der REST-Server läuft, indem Sie mit einem `GET` die Basis-URL abrufen, wodurch der Stammknoten zurückgegeben wird. Er läuft am gleichen Port, den Sie gestern auch für das Webadmin-Tool genutzt haben, unter dem Pfad `/db/data/`. Wir nutzen unseren alten Bekannten `curl`, um die REST-Befehle abzusetzen.

```
$ curl http://localhost:7474/db/data/
{
 "relationship_index" : "http://localhost:7474/db/data/index/relationship",
 "node" : "http://localhost:7474/db/data/node",
 "relationship_types" : "http://localhost:7474/db/data/relationship/types",
 "extensions_info" : "http://localhost:7474/db/data/ext",
 "node_index" : "http://localhost:7474/db/data/index/node",
 "extensions" : {
 }
}
```

Wir erhalten ein hübsches JSON-Objekt zurück, das die URLs anderer Befehle beschreibt, etwa Knoten-Aktionen und Indizes.

Knoten und Beziehungen per REST aufbauen

Knoten und Beziehungen in Neo4j über REST aufzubauen, ist genau so einfach wie bei CouchDB oder Riak. Das Erzeugen eines Knotens ist ein `POST` an den `/db/data/node`-Pfad mit JSON-Daten. Es lohnt sich, jedem Knoten eine `name`-Property zu geben. Das macht das Abrufen der Knotendaten sehr einfach. Sie müssen nur `name` abrufen.

```
$ curl -i -X POST http://localhost:7474/db/data/node \
-H "Content-Type: application/json" \
-d '{"name": "P.G. Wodehouse", "genre": "British Humour"}'
```

Wenn Sie Post nutzen, erscheint der Knotenpfad im Header und im Body stehen Metadaten über den Knoten (die wir hier gekürzt haben). All diese Daten können Sie abrufen, indem Sie `GET` auf die im Header angegebene `Location` anwenden (oder auf die `self`-Property in den Metadaten).

```
HTTP/1.1 201 Created
Location: http://localhost:7474/db/data/node/9
Content-Type: application/json
{
  "outgoing_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/out",
  "data" : {
    "genre" : "British Humour",
    "name" : "P. G. Wodehouse"
  },
  "traverse" : "http://localhost:7474/db/data/node/9/traverse/{returnType}",
  "all_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/9/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/9",
  "properties" : "http://localhost:7474/db/data/node/9/properties",
  "outgoing_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/out/{-list|&|types}",
  "incoming_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/9/relationships",
  "paged_traverse" :
    "http://localhost:7474/db/.../{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/9/relationships/all",
  "incoming_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/in/{-list|&|types}"
}
```

Wenn Sie die Properties der Knoten (nicht die Metadaten) wünschen, können Sie an das `GET` der Knoten-URL ein `/properties` anhängen. Für einzelne Properties hängen Sie zusätzlich noch den Property-Namen an.

```
$ curl http://localhost:7474/db/data/node/9/properties/genre
"British Humour"
```

Ein Knoten allein bringt uns nicht weiter, weshalb wie einen weiteren mit den Properties `["name" : "Jeeves Takes Charge", "style" : "short story"]` anlegen.

Weil P. G. Wodehouse die Kurzgeschichte „Jeeves Takes Charge“ (Jeeves übernimmt das Ruder) geschrieben hat, können wir eine Beziehung zwischen beiden herstellen.

```
$ curl -i -X POST http://localhost:7474/db/data/node/9/relationships \
-H "Content-Type: application/json" \
-d '{"to": "http://localhost:7474/db/data/node/10", "type": "WROTE",
 "data": {"published": "November 28, 1916"}}'
```

Eine schöne Sache am REST-Interface ist, dass es uns tatsächlich schon in den Metadaten (s. o.) über die `create_relationship`-Property gezeigt hat, wie man eine Beziehung herstellt. Auf diese Weise sind REST-Interfaces wechselseitig

Den Pfad finden

Über die REST-Schnittstelle können Sie den Pfad zwischen zwei Knoten ermitteln, indem Sie die Request-Daten an die `/paths`-URL des Startknotens senden. Die Daten des `POST`-Requests müssen als JSON-String vorliegen und den Knoten angeben, dessen Pfad Sie suchen, den Beziehungstyp, den Sie verfolgen wollen und den Algorithmus, der zum Auffinden des Pfades verwendet werden soll.

Im folgenden Beispiel sehen wir uns den Pfad für Beziehungen vom Typ `WROTE` an. Die Suche beginnt bei Knoten 1, verwendet den `shortestPath`-Algorithmus und deckelt die Suche bei einer Tiefe von 10.

```
$ curl -X POST http://localhost:7474/db/data/node/9/paths \
-H "Content-Type: application/json" \
-d '{"to":"http://localhost:7474/db/data/node/10",
 "relationships": {"type" : "WROTE"},
 "algorithm":"shortestPath", "max_depth":10}'
[ {
 "start" : "http://localhost:7474/db/data/node/9",
 "nodes" : [
 "http://localhost:7474/db/data/node/9",
 "http://localhost:7474/db/data/node/10"
 ],
 "length" : 1,
 "relationships" : [ "http://localhost:7474/db/data/relationship/14" ],
 "end" : "http://localhost:7474/db/data/node/10"
} ]
```

Die anderen Pfad-Algorithmen sind `allPaths`, `allSimplePaths` und `dijkstra`. Details zu diesen Algorithmen finden Sie in der Online-Dokumentation,[1] sie hier im Detail zu behandeln würde den Rahmen dieses Buches sprengen.

Indexierung

Wie die anderen Datenbanken, die wir kennengelernt haben, unterstützt auch Neo4j schnelle Daten-Lookups durch den Aufbau von Indizes. Doch es gibt einen Haken. Im Gegensatz zu anderen Datenbank-Indizes, bei de-

1. http://api.neo4j.org/current/org/neo4j/graphalgo/GraphAlgoFactory.html

nen Sie Queries genauso ausführen wie ohne, verwenden Neo4j-Indizes einen anderen Pfad. Das liegt daran, dass die Indexierung ein separater Dienst ist.

Der einfachste Index ist der Schlüssel/Wert-Stil oder auch Hash. Den Schlüssel bilden irgendwelche Daten des Knotens und der Wert ist eine REST-URL, der auf den Knoten im Graphen verweist. Sie können soviele Indizes verwenden, wie Sie wollen, und wir nennen sie „authors“. Das Ende der URLs bildet der Autoren-Name, den wir indexieren wollen, und wir übergeben Knoten 1 als Wert (oder welchen Wert der Wodehouse-Knoten auch hat).

```
$ curl -X POST http://localhost:7474/db/data/index/node/authors \
-H "Content-Type: application/json" \
-d '{ "uri" : "http://localhost:7474/db/data/node/9",
"key" : "name", "value" : "P.G.+Wodehouse"}'
```

Um an den Knoten zu kommen, rufen Sie einfach den Index auf, der (wie Sie bemerken werden) nicht die von uns gesetzte URL zurückgibt, sondern direkt die Knotendaten.

```
$ curl http://localhost:7474/db/data/index/node/authors/name/P.G.+Wodehouse
```

Neben dem Schlüssel/Wert-Index bietet Neo4j auch einen invertierten Index für die Volltextsuche, so dass Sie Queries wie „Gibt mir alle Bücher zurück, deren Namen mit 'Jeeves' beginnen“ durchführen können. Um diesen Index aufzubauen, muss er über die gesamten Daten laufen, und nicht wie vorhin über Einzelelemente. Wie Riak bindet Neo4j Lucene ein, um den invertierten Index aufzubauen.

```
$ curl -X POST http://localhost:7474/db/data/index/node \
-H "Content-Type: application/json" \
-d '{"name":"fulltext", "config":{"type":"fulltext","provider":"lucene"}}'
```

Der `POST` liefert eine JSON-Response zurück, die Informationen über den erzeugten Index enthält.

```
{  "template" : "http://localhost:7474/db/data/index/node/fulltext/{key}/{value}",
  "provider" : "lucene",
  "type" : "fulltext"
}
```

Wenn wir nun Wodehouse in den Volltext-Index einfügen, erhalten wir Folgendes:

```
curl -X POST http://localhost:7474/db/data/index/node/fulltext \
-H "Content-Type: application/json" \
-d '{ "uri" : "http://localhost:7474/db/data/node/9",
"key" : "name", "value" : "P.G.+Wodehouse"}'
```

Die Suche wird dann einfach in Lucene-Query-Syntax an die Index-URL übergeben.

```
$ curl http://localhost:7474/db/data/index/node/fulltext?query=name:P*
```

Indizes können auch für alle Kanten aufgebaut werden. Dazu ersetzen Sie einfach alle *node*-Instanzen in der URL durch *relationship*: `http://localhost:7474/db/data/index/relationship/published/date/1916-11-28`.

REST und Gremlin

Wir haben der ersten Tag mit Gremlin und die erste Hälfte dieses Tages mit dem REST-Interface verbracht. Keine Sorge, wenn Sie nicht wissen, was Sie verwenden sollen. Das Neo4j REST-Interface besitzt ein Gremlin-Plugin (das bei der von uns verwendeten Neo4j-Version standardmäßig verwendet wird).[2] Sie können über REST alle Befehle senden, die Sie auch in der Gremlin-Console verwenden können. Das erlaubt Ihnen, im Produktionsbetrieb die Leistungsfähigkeit und Flexibilität beider Tools zu nutzen. Das ist eine großartige Kombination, weil Gremlin besser für leistungsfähige Queries geeignet ist, während es bei REST mehr um Deployment und Sprachflexibilität geht.

Der folgende Code gibt die Namen aller Knoten zurück. Sie müssen die Daten nur an die Plugin-URL als JSON-String unter dem Feld `script` übergeben.

```
$ curl -X POST \
http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script \
-H "content-type:application/json" \
-d '{"script":"g.V.name"}'

[ "P.G. Wodehouse", "Jeeves Takes Charge" ]
```

Zwar nutzen wir in den Codebeispielen von hier an Gremlin, aber denken Sie immer daran, dass Sie stattdessen auch REST verwenden können.

Big Data

Bisher haben wir nur mit sehr kleinen Datenmengen gearbeitet, darum ist es jetzt an der Zeit, uns anzusehen, was Neo4j mit vielen Daten anstellen kann.

Wir wollen uns einige Film-Daten ansehen, die wir von Freebase.com herunterladen. Wir werden die „performance“-Daten (Tabulator-getrennt) verwenden.[3]Laden Sie die Daten herunter und nutzen Sie dann das folgende Skript. Es geht alle Zeilen durch und erzeugt eine Beziehung zwischen neuen oder existierenden Knoten. Treffer werden im Index über den Namen gefunden.

2. http://docs.neo4j.org/chunked/stable/gremlin-plugin.html
3. http://download.freebase.com/datadumps/latest/browse/film/performance.tsv

Seien Sie gewarnt, die Daten enthalten sehr viele Informationen zu Spielfilmen, von Blockbustern über ausländische Filme bis hin zu, nun ja, Unterhaltung für Erwachsene. Die Ruby-Gems `json` und `faraday` müssen installiert sein, um das Skript auszuführen.

neo4j/importer.rb

```
REST_URL = 'http://localhost:7474/'
HEADER = { 'Content-Type' => 'application/json' }

%w{rubygems json cgi faraday}.each{|r| require r}

# Verbindung mit dem Neo4j REST-Server herstellen
conn = Faraday.new(:url => REST_URL) do |builder|
  builder.adapter :net_http
end

# Existierenden Knoten über den Index abrufen oder neuen anlegen
def get_or_create_node(conn, index, value)
  # Knoten im Index nachschlagen
  r = conn.get("/db/data/index/node/#{index}/name/#{CGI.escape(value)}")
  node = (JSON.parse(r.body).first || {})['self'] if r.status == 200
  unless node
    # Indexierten Knoten nicht gefunden, also neuen anlegen
    r = conn.post("/db/data/node", JSON.unparse({"name" => value}), HEADER)
    node = (JSON.parse(r.body) || {})['self'] if [200, 201].include? r.status
    # Neuen Knoten in den Index einfügen
    node_data = "{\"uri\" : \"#{node}\", \"key\" : \"name\",
      \"value\" : \"#{CGI.escape(value)}\"}"
    conn.post("/db/data/index/node/#{index}", node_data, HEADER)
  end
  node
end

puts "begin processing..."

count = 0
File.open(ARGV[0]).each do |line|
  _, _, actor, movie = line.split("\t")
  next if actor.empty? || movie.empty?

  # Schauspieler- (actor) und Film-Knoten (movie) anlegen
  actor_node = get_or_create_node(conn, 'actors', actor)
  movie_node = get_or_create_node(conn, 'movies', movie)
# Beziehung zwischen Schauspieler und Film herstellen
  conn.post("#{actor_node}/relationships",
    JSON.unparse({ :to => movie_node, :type => 'ACTED_IN' }), HEADER)

  puts " #{count} relationships loaded" if (count += 1) % 100 == 0

end

puts "done!"
```

Sobald alles vorbereitet ist, führen Sie das Skript aus und übergeben den Pfad an die heruntergeladene `performance.tsv`-Datei.

```
$ ruby importer.rb performance.tsv
```

Für die gesamten Daten kann das mehrere Stunden dauern, aber Sie können den Prozess jederzeit abbrechen. Bei Ruby 1.9 verwenden Sie statt `builder.adapter :net_http` besser `builder.adapter :em_synchrony`, das eine nicht-blockierende Verbindung aufbaut.

Raffinierte Algorithmen

Mit unserer großen Spielfilm-Datenbank ist es an der Zeit, das REST-Interface kurz an den Nagel zu hängen und uns wieder Gremlin zuzuwenden.

Kevin Bacon

Wir wollen spaßeshalber einen der berühmtesten existierenden Graph-Algorithmen implementieren: den Kevin-Bacon-Algorithmus. Dieser Algorithmus basiert auf einem Spiel, bei dem es darum geht, die kürzeste Verbindung zwischen einem beliebigen Schauspieler und Kevin Bacon über Spielfilme zu finden, in denen beide mitgespielt haben. Zum Beispiel hat Alec Guinness in *Kafka* mit Theresa Russell gespielt, die in *Wild Things* mit Kevin Bacon aufgetreten ist.

Bevor wir weitermachen, starten Sie die Gremlin-Console und öffnen den Graphen. Dann legen wir einen eigenen Schritt namens `costars` mit dem folgenden Code an. Das ähnelt dem `friendsuggest` von gestern. Er findet die Co-Stars eine actor-Knotens (Schauspieler, die eine Kante mit den Filmen des Ursprungs-Schauspielers teilen).

neo4j/costars.groovy

```
Gremlin.defineStep( 'costars',
  [Vertex, Pipe],
  {
    _().sideEffect{start = it}.outE('ACTED_IN').
    inV.inE('ACTED_IN').outV.filter{
      !start.equals(it)
    }.dedup
  }
)
```

Bei Neo4j geht es nicht so sehr darum, eine Wertemenge „abzufragen“, sondern eher darum, den Graph „durchzugehen“. Das Schöne an diesem Konzept ist, dass der erste Knoten, den Sie „durchgehen“ auch der ist, der dem Startknoten am nächsten ist (gemessen in einfacher Kante/Knoten-Distanz, nicht in gewichteter Distanz). Wir wollen damit beginnen, unsere Start- und End-Knoten zu finden.

```
gremlin> bacon = g.V.filter{it.name=='Kevin Bacon'}.next()
gremlin> elvis = g.V.filter{it.name=='Elvis Presley'}.next()
```

Wir beginnen damit, die Co-Stars der Co-Stars der Co-Stars eines Schauspielers zu finden. Klassischerweise hört man nach sechs Schritten auf, praktisch kann man aber nach vier Schritten aufhören (wenn Sie nichts finden, können Sie die Suche wiederholen). Wir gehen den Graphen viermal durch, was alle Schauspieler 4. Grades findet. Wir nutzen dabei den gerade entwickelten costars-Schritt.

```
elvis.costars.loop(1){it.loops < 4}
```

Nur mit Bacon endende Knoten werden behalten. Alle anderen werden ignoriert.

```
elvis.costars.loop(1){
  it.loops < 4
}.filter{it.equals(bacon)}
```

Nur um sicherzustellen, dass die Schleife beim Kevin Bacon-Knoten nicht ein zweites Mal durchlaufen wird, bricht der Filter sie beim Bacon-Knoten ab. Anders ausgedrückt, wird die Schleife ausgeführt, solange sie nicht viermal durchlaufen wurde und keinen Bacon-Knoten getroffen hat. Dann können wir die Pfade ausgeben, die zum Erreichen jedes Bacon-Knotens genommen wurden.

```
elvis.costars.loop(1){
  it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths
```

Nun müssen wir nur noch den ersten Pfad aus der Liste möglicher Pfade abrufen – der kürzeste Pfad steht an erster Stelle. Das >> nimmt einfach das erste Elemente von der Liste aller Knoten.

```
(elvis.costars.loop(1){
  it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1)
```

Abschließend benennen wir jeden Knoten und filtern Kanten ohne Daten über den Groovy-Befehl grep aus.

```
(elvis.costars.loop(1){
  it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1).name.grep{it}
```

```
==>Elvis Presley
==>Double Trouble
==>Roddy McDowall
==>The Big Picture
==>Kevin Bacon
```

Wir wissen nicht, wer Roddy McDowall ist, doch das ist das Schöne an unserer Graph-Datenbank. Wir müssen das nicht wissen, um eine gute Antwort zu erhalten. Wenn Sie wollen, können Sie noch tiefer in die Groovy-Welt einsteigen, wenn die Antwort nur eine einfache Liste sein soll. Die Daten sind alle da.

Random Walk

Wenn Sie eine gute Teilmenge aus einer großen Datenmenge brauchen, dann ist der sog. „Random Walk" ein guter Trick. Sie fangen mit einem Zufallszahlengenerator an.

```
rand = new Random()
```

Dann wählen Sie eine Zielquote der Gesamtmenge. Wenn nur etwa ein Drittel der etwa 60 Filme von Kevin Bacon zurückgeliefert werden soll, können Sie jede Zufallszahl kleiner 0,33 herausfiltern.

```
bacon.outE.filter{rand.nextDouble() <= 0.33}.inV.name
```

Das sollte so um die zwanzig zufällige Titel aus Bacons Filmografie zurückliefern.

Geht man von Kevin Bacon aus zwei Schritte (zu den Co-Stars seiner Co-Stars), wird die Liste schon recht lang (bei unseren Daten über 300000).

```
bacon.outE.inV.inE.outV.loop(4){
  it.loops < 3
}.count()

==> 316198
```

Wenn Sie aber nur etwa ein Prozent dieser Liste benötigen, fügen Sie einen Filter ein. Beachten Sie, dass der Filter selbst ein Schritt ist, d. h., Sie müssen die loop-Zahl um 1 erhöhen.

```
bacon.outE{
  rand.nextDouble() <= 0.01
}.inV.inE.outV.loop(5){
  it.loops < 3
}.name
```

Unser Ergebnis enthielt Elijah Wood, den wir durch unseren Bacon-Pfad-Algorithmus laufen lassen können, was uns zwei Schritte zurückliefert (Elijah Wood hat in *Deep Impact* zusammen mit Ron Eldard gespielt, der mit Kevin Bacon in *Sleepers* mitgewirkt hat).

Zentralität

Zentralität ist ein Maß für individuelle Knoten in einem Graphen. Wenn Sie zum Beispiel wissen wollen, wie wichtig jeder Knoten in einem Netzwerk gemessen an seinem Abstand zu anderen Knoten ist, dann benötigen Sie einen Zentralitätsalgorithmus.

Der berühmteste Zentralitätsalgorithmus ist wohl Googles PageRank, doch es gibt verschiedene Arten. Wir verwenden eine einfache Version namens *Eigenvektorzentralität*, die einfach nur die Anzahl der ein- und ausgehenden Kanten im Bezug auf einen Knoten zählt. Wir werden jedem Schauspieler einen Wert zuweisen, der angibt, wie viele Rollen er gespielt hat.

Wir benötigen eine Map, um `groupCount()` zu befüllen und einen Zähler `count`, um die maximale Zahl der Schleifen festzulegen.

```
role_count = [:]; count = 0
g.V.in.groupCount(role_count).loop(2){ count++ < 1000 }; ''
```

Den Schlüssel für die `role_count`-Map bilden die Knoten und die Werte bilden die Anzahl der Kanten, die jeder Knoten besitzt. Die einfachste Möglichkeit, die Ausgabe zu lesen, ist das Sortieren der Map.

```
role_count.sort{a,b -> a.value <=> b.value}
```

Der letzte Wert ist der Schauspieler mit den meisten Auftritten. In unseren Daten geht diese Ehre an den legendären Synchronsprecher Mel Blanc mit 424 Credits (die Sie sich mit `g.V.filter{it.name=='Mel Blanc'}.out.name` ausgeben lassen können).

Externe Algorithmen

Die Entwicklung eigener Algorithmen ist ganz schön, aber ein Großteil dieser Arbeit wurde bereits erledigt. Das JUNG-Framework (Java Universal Network/Graph) ist eine Sammlung gängiger Graph-Algorithmen und anderer Tools zur Modellierung und Visualisierung von Graphen. Dank des Gremlin/Blueprint-Projekts ist es einfach, auf JUNG-Algorithmen wie PageRank, HITS, Voltage, Zentralitätsalgorithmen und Graph-als-Matrix-Tools zuzugreifen.

Um JUNG nutzen zu können, müssen wir den Neo4j-Graph in einen neuen JUNG-Graph packen.[4]Um auf den JUNG-Graphen zugreifen zu können, haben Sie zwei Möglichkeiten: Laden Sie alle Blueprint- und JUNG-jars herunter, installieren Sie sie im libs-Verzeichnis des Neo4j-Servers und starten Sie den Server neu, oder laden Sie die vorgepackte Gremlin-Console herunter. Für dieses Projekt empfehlen wir die zweite Möglichkeit, weil es Ihnen die Mühe erspart, verschiedenen Java-Archivdateien (jars) hinterherzujagen.

Sobald Sie die Gremlin-Console heruntergeladen haben, fahren Sie den Neo4j-Server herunter und starten Gremlin. Sie müssen nun das Neo4jGraph-Objekt erzeugen und dabei auf das `data/graph`-Verzeichnis Ihrer Installation verweisen.

```
g = new Neo4jGraph('/users/x/neo4j-enterprise-1.7/data/graph.db')
```

Wir verwenden den Gremlin-Graph `g`. Das Neo4jGraph-Objekt muss in ein GraphJung-Objekt gepackt werden, das wir `j` nennen.

```
j = new GraphJung( g )
```

Ein Grund, warum Kevin Bacon als Zielpfad gewählt wurde, ist seine relative Nähe zu anderen Schauspielern. Er ist in Filmen zusammen mit vielen beliebten Stars aufgetreten. Um wichtig zu sein, musste er selbst nicht viele Rollen spielen, er musste nur mit denen verbunden sein, die gut vernetzt sind.

Da stellt sich die Frage, ob wir einen Schauspieler finden, der (gemessen an der Distanz zu anderen Schauspielern) eine bessere Wahl ist als Kevin Bacon.

JUNG enthält einen Scoring-Algorithmus namens BarycenterScorer, der jeden Knoten basierend auf der Distanz zu allen anderen Knoten bewertet. Wenn Kevin Bacon wirklich die beste Wahl ist, dann müsste sein Wert der kleinste sein, was bedeutet, dass er an allen anderen Schauspielern am „nächsten“ dran ist.

Unser JUNG-Algorithmus soll nur auf Schauspieler angewandt werden, weshalb wir einen *transformer* entwickeln, um nur die actor-Knoten herauszufiltern. Der `EdgeLabelTransformer` erlaubt für den Algorithmus nur die Knoten mit einer `ACTED_IN`-Kante.

```
t = new EdgeLabelTransformer(['ACTED_IN'] as Set, false)
```

4. http://blueprints.tinkerpop.com

Als Nächstes müssen wir den Algorithmus selbst importieren und ihm unseren GraphJung und den Transformer übergeben.

```
import edu.uci.ics.jung.algorithms.scoring.BarycenterScorer
barycenter = new BarycenterScorer<Vertex,Edge>( j, t )
```

Damit erhalten wir die BarycenterScorer-Wertung für jeden Knoten. Sehen wir uns Kevin Bacons Wertung an.

```
bacon = g.V.filter{it.name=='Kevin Bacon'}.next()
bacon_score = barycenter.getVertexScore(bacon)
```

```
~0.0166
```

Sobald wir Kevin Bacons Wertung kennen, können wir jeden Knoten durchgehen und diejenigen festhalten, deren Wertung kleiner ist.

```
connected = [:]
```

Es dauert wirklich lange, die BarycenterScorer-Wertung für jeden Schauspieler in der Datenbank zu berechnen. Darum lassen wir den Algorithmus nur über Kevins Co-Stars laufen. Das dauert, je nach Hardware, ein paar Minuten. BarycenterScorer ist schnell, doch wenn wir ihn über alle Co-Stars laufen lassen, summiert sich das ganz schön.

```
bacon.costars.each{
  score = b.getVertexScore(it);
  if(score < bacon_score) {
    connected[it] = score;
  }
}
```

Alle Schlüssel, die sich in der `connected`-Map befinden, sind eine bessere Wahl als Kevin Bacon. Aber ein bekannter Name ist schöner, deshalb geben wir sie alle aus und wählen uns einen, den wir mögen. Ihre Ausgabe kann variieren, da die Daten der Filmdatenbank ständig ergänzt werden.

```
connected.collect{k,v -> k.name + " => " + v}
```

```
==> Donald Sutherland => 0.00925
==> Clint Eastwood => 0.01488
...
```

Donald Sutherland erscheint in der Liste mit einem respektablen Wert von ~0,00925. Hypothetisch gesehen sind die „Six Degrees“ von Donald Sutherland leichter zu spielen als die traditionellen „Six Degrees“ von Kevin Bacon.

Mit unserem j-Graph können wir nun jeden JUNG-Algorithmus über unsere Daten laufen lassen, z. B. PageRank. Wie bei BarycenterScorer müssen wir zuerst die Klasse importieren.

```
import edu.uci.ics.jung.algorithms.scoring.PageRank
pr = new PageRank<Vertex,Edge>( j, t, 0.25d )
```

Eine vollständige Liste der JUNG-Algorithmen finden Sie in der Online-Javadoc-API. Es werden laufend neue hinzugefügt, weshalb Sie da reinschauen sollten, bevor Sie etwas selbst implementieren.

Was wir am zweiten Tag gelernt haben

Am zweiten Tag haben wir unsere Fähigkeit zur Interaktion mit Neo4j erweitert, indem wir uns das REST-Interface angesehen haben. Wir haben gesehen, wie man über das Gremlin-Plugin Gremlin-Code auf dem Server ausführen und sich vom REST-Interface die Ergebnisse zurückgeben lassen kann. Wir haben mit größeren Datenmengen herumgespielt und zum Schluss eine Handvoll nützlicher Algorithmen kennengelernt, mit denen man in diese Daten eintauchen kann.

Tag 2: Selbststudium

Finden Sie heraus

1. Fügen Sie die Dokumentation der Neo4j REST-API zu Ihren Lesezeichen hinzu.
2. Fügen Sie die API des JUNG-Projekts und der darin implementierten Algorithmen zu Ihren Lesezeichen hinzu.
3. Finden Sie eine Bindung oder ein REST-Interface für Ihre Lieblings-Programmiersprache.

Machen Sie Folgendes

1. Machen Sie aus dem Teil, der beim Kevin-Bacon-Algorithmus den Pfad findet, einen eigenen Schritt. Implementieren Sie dann eine allgemeine Groovy-Funktion (zum Beispiel `def actor_path(g,name1,name2){...}`), die den Graphen und zwei Namen nimmt und die Distanz vergleicht.
2. Wählen Sie einen der vielen JUNG-Algorithmen aus und lassen Sie ihn über einen Knoten (oder über alle Daten, wenn es die API erlaubt) laufen.
3. Installieren Sie einen Treiber Ihrer Wahl und verwenden Sie ihn, um einen Graphen Ihres Unternehmens aufzubauen (welche Leute haben welche

Funktion). Die Kanten beschreiben dabei, wer mit wem wie zusammenarbeitet (berichtet an, arbeitet mit). Bei großen Unternehmen beschränken Sie sich auf die nächstliegenden Abteilungen, Bei kleinen Unternehmen können Sie auch Ihre Kunden mit einbeziehen. Finden Sie die am besten vernetzte Person innerhalb der Organisation, indem Sie die kürzeste Distanz zwischen allen Knoten ermitteln.

7.4 Tag 3: Verteilte Hochverfügbarkeit

Wir wollen unsere Betrachtung von Neo4j damit abschließen, wie wir Neo4j besser auf unternehmenskritische Anwendungen vorbereiten. Sie werden sehen, wie Neo4j Daten über ACID-konforme Transaktionen stabil hält. Wir wollen dann einen Neo4j-Hochverfügbarkeitscluster (high availability, HA) installieren und konfigurieren, um die Verfügbarkeit bei sehr vielen Leseoperationen zu verbessern. Danach sehen wir uns Backup-Strategien an, die die Sicherheit unserer Daten gewährleisten.

Transaktionen

Neo4j ist (genau wie PostgreSQL) eine atomische, konsistente, isolierte, dauerhafte (Atomic, Consistent, Isolated, Durable, kurz ACID) Transaktionsdatenbank. Das macht sie zu einer guten Wahl für wichtige Daten, für die Sie anderenfalls eine relationale Datenbank gewählt hätten. Genau wie bei den bisher gezeigten Transaktionen sind auch Neo4j-Transaktionen Alles-oder-nichts-Operationen. Wenn eine Transaktion beginnt, wird jede nachfolgende Operation als atomische Einheit durchgeführt und ist entweder erfolgreich oder schlägt fehl. Schlägt eine Operation fehl, gelten alle Operationen als fehlgeschlagen.

Die Details der Transaktionverarbeitung finden sich unter Gremlin im zugrundeliegenden Neo4j-Wrapper-Projekt namens Blueprint. Diese Details

können sich dabei von Version zu Version ändern. Wir verwenden Gremlin 1.3, das Blueprints 1.0 nutzt. Wenn Sie bei einer der beiden eine andere Version nutzen, finden Sie die Besonderheiten in den Blueprint API-Javadocs.

Genau wie bei PostgreSQL werden einfache einzeilige Funktionen automatisch in eine implizite Transaktion gepackt. Um mehrzeilige Transaktionen zu demonstrieren, müssen wir den automatischen Transaktionsmodus des Graphen-Objekts deaktivieren. Damit lassen wir Neo4j wissen, dass wir Transaktionen manuell verarbeiten wollen. Sie können den Transaktionsmodus über die `setTransactionMode`-Funktion festlegen.

```
gremlin> g.setTransactionMode(TransactionalGraph.Mode.MANUAL)
```

Wir können eine Transaktion für ein Graphen-Objekt mit `startTransaction` und `stopTransaction(conclusion)` beginnen und beenden. Wenn Sie die Transaktion anhalten, müssen Sie auch angeben, ob die Transaktion erfolgreich war. Wenn nicht, kann Neo4j alle Befehle seit dem Start zurücknehmen (Rollback). Sie sollten die Transaktion in einen `try/catch`-Block packen, um sicherzustellen, dass jede Ausnahme einen Rollback anstößt.

```
g.startTransaction()
try {
  // Mehrere Schritte ausführen...
  g.stopTransaction(TransactionalGraph.Conclusion.SUCCESS)
} catch(e) {
  g.stopTransaction(TransactionalGraph.Conclusion.FAILURE)
}
```

Wenn Sie außerhalb der Grenzen von Gremlin und direkt mit der Neo4j-`EmbeddedGraphDatabase` arbeiten wollen, können Sie die Java API-Syntax für Transaktionen verwenden. Möglicherweise müssen Sie diesen Stil verwenden, wenn Sie Java-Code entwickeln oder eine Sprache nutzen, die hinter den Kulissen Java ist (z. B. JRuby).

```
r = g.getRawGraph()
tx = r.beginTx()
try {
  // Mehrere Schritte ausführen...
  tx.success()
} finally {
  tx.finish()
}
```

Beide Varianten sind vollständig ACID-konform. Selbst bei Systemfehlern wird sichergestellt, dass alle Schreiboperationen zurückgenommen werden, wenn der Server wieder gestartet wird. Wenn Sie Transaktionen nicht selbst

verarbeiten wollen, belassen Sie den Transaktionsmodus besser bei `TransactionalGraph.Mode.AUTOMATIC`.

Hochverfügbarkeit

Der Hochverfügbarkeitsmodus ist Neo4js Antwort auf die Frage, ob eine Graph-Datenbank skalieren kann: Ja, wenn auch mit einigen Vorbehalten. Eine Schreiboperation an einen Slave wird nicht unmittelbar mit allen anderen Slaves synchronisiert, d. h., es besteht die Gefahr, dass die Konsistenz (im Sinne von CAP) für einen Moment verloren geht, auch wenn sie schlussendlich konsistent ist. Bei Hochverfügbarkeit gehen rein ACID-konforme Transaktionen verloren. Das ist der Grund, warum Neo4j-HA größtenteils als Lösung zur Erhöhung der Lesekapazität angepriesen wird.

Wie bei Mongo wählen die Server im Cluster einen Master, der die Hauptkopie der Daten vorhält. Im Gegensatz zu Mongo akzeptieren Slaves aber Schreiboperationen. Slave-Schreiboperationen werden mit dem Master synchronisiert und die Änderungen dann von ihm an die andere Slaves übertragen.

HA-Cluster

Um Neo4j-HA nutzen zu können, müssen Sie zuerst einen Cluster einrichten. Neo4j verwendet einen externen Koordinator-Service namens Zookeeper für den Cluster. Zookeeper ist ein weiteres hervorragendes Projekt, das aus dem Apache Hadoop-Projekt hervorgegangen ist. Es handelt sich um einen Allzweckdienst zur Koordinierung verteilter Anwendungen. Neo4j-HA nutzt es zur Verwaltung seiner Laufzeit-Aktivitäten. Jeder Neo4j-Server hat einen eigenen Koordinator, dessen Aufgabe es ist, seinen Platz im Cluster zu verwalten (siehe Abbildung 36, *Neo4j-Cluster mit drei Servern und deren Koordinatoren*, auf Seite 275).

Glücklicherweise wird Neo4j Enterprise zusammen mit Zookeeper ausgeliefert sowie mit einigen Dateien, die uns bei der Konfiguration eines Clusters helfen. Wir wollen drei Instanzen von Neo4j Enterprise Version 1.7 betreiben. Sie können von der Website eine Kopie für Ihr Betriebssystem herunterladen (achten Sie auch die richtige Edition).[5]Entpacken Sie die Kopie und legen Sie zwei weitere Kopien des Verzeichnisses an. Wir hängen 1, 2 und 3 an die Verzeichnisnamen an und bezeichnen sie auch so.

```
tar fx neo4j-enterprise-1.7-unix.tar
mv neo4j-enterprise-1.7 neo4j-enterprise-1.7-1
cp -R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-2
cp -R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-3
```

5. http://neo4j.org/download/

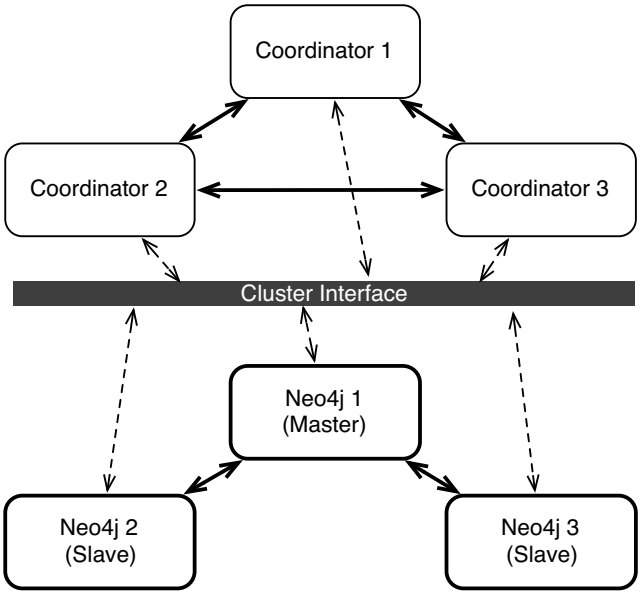


Abbildung 36: Neo4j-Cluster mit drei Servern und deren Koordinatoren

Nun besitzen wir drei identische Kopien unserer Datenbank.

Normalerweise würden Sie eine Kopie pro Server entpacken und den Cluster so konfigurieren, dass er die anderen Server kennt. Da wir sie aber lokal ausführen, verwenden wir stattdessen drei verschiedene Verzeichnisse und nutzen unterschiedliche Ports.

Wir brauchen fünf Schritte, um den Cluster anzulegen. Wir beginnen mit der Konfiguration des Zookeeper Cluster-Koordinators und richten dann die Neo4j-Server ein.

1. Geben Sie jedem Koordinator-Server eine eindeutige ID.
2. Konfigurieren Sie jeden Koordinator-Server so, dass er mit den anderen Servern und seinem Neo4j-Server kommuniziert.
3. Starten Sie alle drei Koordinator-Server.
4. Konfigurieren Sie jeden Neo4j-Server so, dass er im HA-Modus läuft. Geben Sie ihm einen eindeutigen Port und stellen Sie sicher, dass er den Koordinator-Cluster erkennt.
5. Starten Sie alle drei Neo4j-Server.

Zookeeper hält jeden Server über eine im Cluster eindeutige ID nach. Diese Zahl ist der einzige Wert in der Datei `data/coordinator/myid`. Bei Server 1 belassen wir ihn auf der Voreinstellung 1; bei Server 2 setzen wir ihn auf 2 und bei Server 3 auf 3.

```
echo "2" > neo4j-enterprise-1.7-2/data/coordinator/myid
echo "3" > neo4j-enterprise-1.7-3/data/coordinator/myid
```

Wir müssen auch ein paar clusterinterne Kommunikationseinstellungen vornehmen. Jeder Server besitzt eine Datei namens `conf/coord.cfg`. Standardmäßig verwendet die Variable `server.1` den Server `localhost` und es sind zwei Ports gesetzt: der Port für die Quorum-Wahl (Quorum Election Port, `2888`) und der Port für die Master-Wahl (Master Election Port, `3888`).

Den Cluster aufbauen

Ein Zookeeper-Quorum ist eine Gruppe von Servern im Cluster und die Ports, über die sie kommunizieren. (Verwechseln Sie das nicht mit dem Riak-Quorum, bei dem es um eine minimale Majorität geht, die die Konsistenz sicherstellt). Der Port für die Master-Wahl wird genutzt, wenn der Master ausfällt – die verbliebenen Server können über diesen Port einen neuen Master wählen. Wir belassen `server.1` wie es ist und lassen `server.2` und `server.3` aufeinanderfolgende Ports nutzen. Die `coord.cfg`-Dateien auf den Servern 1, 2 und 3 müssen alle die gleichen drei Zeilen enthalten.

```
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

Zum Schluss müssen Sie den öffentlichen Port festlegen, über den Neo4j die Verbindung herstellen kann. Der `clientPort` ist mit 2181 voreingestellt, den wir bei Server 1 auch so belassen. Für Server 2 setzen wir `clientPort=2182` und für Server 3 `clientPort=2183`. Wenn diese Ports bei Ihrem Server belegt sind, wählen Sie einfach andere aus, doch Sie müssen sicherstellen, dass diese Ports bei den folgenden Schritten verwendet werden.

Koordination

Wir starten den Zookeeper-Koordinator mit einem praktischen Skript, das vom Neo4j-Team zur Verfügung gestellt wird. Führen Sie den folgenden Befehl in jedem der drei Server-Verzeichnisse aus:

```
bin/neo4j-coordinator start
Starting Neo4j Coordinator...WARNING: not changing user
  process [36542]... waiting for coordinator to be ready. OK.
```

Der Koordinator läuft jetzt, Neo4j aber noch nicht.

Neo4j einbinden

Nun müssen wir Neo4j so einrichten, dass es im Hochverfügbarkeitsmodus läuft, und dann verbinden wir es mit einem Koordinator-Server. Öffnen Sie `conf/neo4j-server.properties` und tragen Sie für jeden Server die folgende Zeile ein:

```
org.neo4j.server.database.mode=HA
```

Damit versetzen wir Neo4j in den Hochverfügbarkeitsmodus. Bisher haben wir ihn im SINGLE-Modus betrieben. Da wir die Datei gerade geöffnet haben, setzen wir auch den Webserver-Port auf einen eindeutigen Wert. Normalerweise ist der Standardport 7474 in Ordnung, aber da wir drei Neo4j-Instanzen auf einem Rechner betreiben, müssen wir sie für http/https anpassen. Wir wählen die Ports 7471/7481 für Server 1, 7472/7482 für Server 2 und 7473/7483 für Server 3.

```
org.neo4j.server.webserver.port=7471
org.neo4j.server.webserver.https.port=7481
```

Abschließend richten wir jede Neo4j-Instanz so ein, dass Sie sich mit einem der Koordinator-Server verbindet. Wenn Sie die Datei `conf/neo4j.properties` für Server 1 öffnen, sollten Sie einige auskommentierte Zeilen sehen, die mit `ha` beginnen. Das sind die Hochverfügbarkeitseinstellungen, die drei Dinge festlegen: die aktuelle Cluster-Rechnernummer, die Liste der Zookeeper-Server und den Port, den die Neo4j-Server nutzen, um miteinander zu kommunizieren. Für Server 1 fügen Sie die folgenden Felder in `neo4j.properties` ein:

```
ha.server_id=1
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6001
ha.pull_interval=1
```

Die Einstellungen für die beiden anderen Server sind im Wesentlichen gleich, aber wir müssen `ha.server_id=2` für Server 2 und `ha.server_id=3` für Server 3 eintragen und der `ha.server` muss einen anderen Port verwenden (wir nutzen 6002 für Server 2 und 6003 für Server 3). Wie gesagt, müssen Sie die Server-Ports nicht ändern, wenn Sie separate Rechner betreiben. Server 2 enthält Folgendes (und entsprechend geht es mit Server 3 weiter):

```
ha.server_id=2
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6002
ha.pull_interval=1
```

Wir setzen `pull_interval` auf 1, was bedeutet, dass jeder Slave einmal pro Sekunden den Master auf Updates überprüft. Üblicherweise ist der Wert nicht so klein, aber auf diese Weise können wir die Updates für die Beispieldaten sehen, die wir gleich einfügen.

Nachdem wir unsere Neo4j HA-Server konfiguriert haben, ist es an der Zeit, sie zu starten. Genau wie die Koordinator-Server starten wir auch die Neo4j-Server in ihren jeweiligen Installationsverzeichnissen.

```
bin/neo4j start
```

Sie können sich die Ausgaben des Server ansehen, indem Sie sich den Log per tail ausgeben lassen.

```
tail -f data/log/console.log
```

Jeder Server hängt sich an seinen konfigurierten Koordinator an.

Den Cluster-Status prüfen

Der zuerst gestartete Koordinator wird zum Master-Server – wahrscheinlich Server 1. Sie können das überprüfen, indem Sie den Web-Admin der angeschlossenen Neo4j-Instanz öffnen (wir haben eben Server 1 auf Port 7471 gesetzt). Klicken Sie oben den Server Info-Link an und dann im seitlichen Menü auf High Availability.[6]

Die Properties unter High Availability liefern Informationen zu diesem Cluster. Wenn der Server der Master-Server ist, ist die Property wahr. Wenn nicht, können Sie unter InstancesInCluster nachsehen, wer zum Master gewählt wurde. Hier finden Sie alle angeschlossenen Server samt Rechner-ID, welcher der Master ist und weitere Informationen.

Replikation prüfen

Da unser Cluster nun funktioniert, können wir überprüfen, ob die Server korrekt replizieren. Wenn alles nach Plan läuft, sollten alle Schreiboperationen an einen Slave irgendwann auf den Master-Knoten repliziert werden und schlussendlich auch an die anderen Slave-Server. Wenn Sie die Web-Conso-

6. http://localhost:7471/webadmin/#/info/org.neo4j/High%20Availability/

len für alle drei Server öffnen, können Sie die eingebauten Gremlin-Consolen im Web-Admin nutzen. Beachten Sie, dass das Gremlin Graph-Objekt nun in ein HighlyAvailableGraphDatabase-Objekt gepackt wurde.

```
g = neo4jgraph[HighlyAvailableGraphDatabase [/.../neo4j-ent-1.7-2/data/graph.db]]
```

Um unsere Server zu testen, füllen wir einen neuen Graphen mit einigen Knoten auf, die die Namen berühmter Paradoxien enthalten. An einer der Slave-Consolen legen wir den Stammknoten mit Zenons Paradoxien fest.

```
gremlin> root = g.v(0)
gremlin> root.paradox = "Zeno's"
gremlin> root.save
```

Nun wechseln wir auf die Console des Master-Servers und lassen uns die Werte ausgeben.

```
gremlin> g.V.paradox
==> Zeno’s
```

Wechseln Sie nun auf den anderen Slave-Server und fügen Russells Antinomie hinzu, zeigt ein kurzer Blick auf unsere Liste, dass beide Knoten auf dem zweiten Server vorhanden sind, obwohl wir hier nur einen eingegeben haben.

```
gremlin> g.addVertex(["paradox" : "Russell's"])
gremlin> g.V.paradox
==> Zeno's
==> Russell's
```

Wenn die Änderungen noch auf einen der Slave-Server durchgereicht wurden, können Sie noch einmal zur High-Availability-Seite in der Server-Info wechseln. Sehen Sie sich alle Instanzen von `lastCommittedTransactionId` an. Wenn diese Werte gleich sind, sind die Systemdaten konsistent. Je kleiner der Wert, desto älter die Version der Daten auf diesem Server.

Wahl des Masters

Wenn Sie den Master-Server herunterfahren und die Server-Infos für die restlichen Server aktualisieren, werden Sie sehen, dass ein anderer Server als Master gewählt wurde. Wenn Sie den Server wieder starten, wird er wieder in den Cluster eingefügt, aber nun bleibt der alte Master ein Slave (bis ein weiterer Server ausfällt).

Hochverfügbarkeit erlaubt es Systemen mit sehr hohen Leseaktivitäten, einen Graphen über mehrere Server zu replizieren und damit die Last zu ver-

teilen. Zwar ist der Cluster als Ganzes nur schlussendlich konsistent, doch Sie können einige Tricks nutzen, um die Chancen zu minimieren, dass veraltete Daten gelesen werden, beispielsweise die Zuweisung einer Session an einen Server. Mit den richtigen Tools, guter Planung und einem guten Setup können Sie eine Graph-Datenbank aufbauen, die Milliarden von Knoten und Kanten sowie eine nahezu unendliche Zahl von Requests verarbeiten kann. Fügen Sie einfach noch regelmäßige Backups hinzu und schon haben Sie die Lösung für ein solides Produktionssystem.

Backups

Backups sind ein notwendiger Aspekt jeder professionellen Datenbanknutzung. Auch wenn Backups über die Replikation schon integriert sind, helfen nächtliche Backups, die außerhalb gespeichert werden, wenn es zu einer Katastrophe kommen sollte. Dinge wie Feuer im Serverraum oder Erdbeben lassen sich nicht planen.

Neo4j Enterprise bietet ein einfaches Backup-Tool namens neo4j-backup an.

Die leistungsfähigste Methode beim Betrieb eines HA-Servers ist ein Backup-Befehl, der die Datenbank-Datei aus dem Cluster in eine mit einem Zeitstempel verstehende Datei auf einer gemounteten Platte speichert. Eine Kopie von jedem Server stellt sicher, dass sie die aktuellsten Daten haben. Das erzeugte Backup-Verzeichnis ist eine voll funktionsfähige Kopie. Wenn Sie die Daten wiederherstellen müssen, ersetzen Sie einfach das Datenverzeichnis jeder Installation durch das Backup-Verzeichnis und schon sind Sie wieder einsatzbereit.

Sie müssen mit einem vollständigen Backup anfangen. Nachfolgend sichern wir unseren HA-Cluster in einem Verzeichnis, das mit dem aktuellen Datum endet (wir nutzen dazu den *nix-Befehl date).

```
bin/neo4j-backup -full -from ha://localhost:2181,localhost:2182,localhost:2183 \
-to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

Wenn Sie nicht im HA-Modus arbeiten, ändern Sie den Modus im URI einfach in „single“. Sobald Sie ein vollständiges Backup durchgeführt haben, wählen Sie ein inkrementelles Backup, das nur die Änderungen seit dem letzten Backup speichert. Soll auf einem einzelnen Server jeweils um Mitternacht ein vollständiges und danach alle zwei Stunden ein inkrementelles Backup durchgeführt werden, führen Sie den folgenden Befehl aus:

```
bin/neo4j-backup -incremental -from single://localhost \
-to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

Denken Sie daran, dass inkrementelle Backups nur mit vollständig gesicherten Verzeichnissen funktionieren. Stellen Sie also sicher, dass der obige Befehl am gleichen Tag ausgeführt wird.

Was wir am dritten Tag gelernt haben

Heute haben wir einige Zeit damit verbracht, die Neo4j-Daten über ACID-konforme Transaktionen stabil zu halten, haben uns mit Hochverfügbarkeit beschäftigt und mit Backup-Tools.

Es ist wichtig zu beachten, dass alle heute verwendeten Tools die Neo4j Enterprise-Edition verlangen, die eine duale Lizenz verwendet – GPL/AGPL. Wenn Ihr Server Closed Source bleiben soll, sollten Sie zur Community-Edition wechseln oder sich eine OEM-Version von Neo Technology (dem Unternehmen hinter Neo4j) beschaffen. Weitere Informationen erhalten Sie vom Neo4j-Team.

Tag 3: Selbststudium

Finden Sie heraus

1. Finden Sie den Neo4j Licensing Guide.
2. Beantworten Sie folgende Frage: „Welche maximale Anzahl von Knoten wird unterstützt?“ (Tipp: Es steht in den Fragen und Antworten in der Dokumentation.)

Machen Sie Folgendes

1. Replizieren Sie Neo4j über drei physikalische Server.
2. Richten Sie einen Load-Balancer über einen Webserver wie Apache oder Nginx ein und stellen Sie die Verbindung zum Cluster über das REST-Interface her. Führen Sie einen Gremlin-Skript-Befehl aus.

7.5 Zusammenfassung

Neo4j ist eine der besten Open-Source-Implementierungen der (relativ seltenen) Klasse von Graph-Datenbanken. Graph-Datenbanken konzentrieren sich auf die Beziehungen zwischen den Daten, nicht auf die Gemeinsamkeiten zwischen den Werten. Die Modellierung von Graph-Daten ist einfach. Sie können Knoten erzeugen, Beziehungen zwischen ihnen herstellen und optional Schlüssel/Wert-Paare anhängen. Für Abfragen muss man nur festlegen, wie der Graph vom Startknoten aus durchgegangen werden soll.

Neo4js Stärken

Neo4j ist eines der feinsten Beispiele für Open Source Graph-Datenbanken. Graph-Datenbanken eignen sich perfekt für unstrukturierte Daten, in vielerlei Hinsicht sogar besser als Dokumentenspeicher. Neo4j ist nicht nur typ- und schemafrei, sondern legt Ihnen auch keinerlei Beschränkungen auf, in welcher Beziehung die Daten zueinander stehen. Sie ist (im besten Sinne) für alles offen. Momentan unterstützt Neo4j 34,4 Milliarden Knoten und 34,4 Milliarden Beziehungen, was für die meisten Fälle mehr als ausreichend sein dürfte (Neo4j kann 42 Knoten für jeden der 800 Millionen Facebook-Nutzer in einem einzelnen Graphen vorhalten).

Die Neo4j-Distributionen bieten verschiedene Tools für schnelle Lookups mit Lucene und einfach zu nutzende (wenn auch etwas kryptische) Spracherweiterungen wie Gremlin und das REST-Interface. Neben der einfachen Verwendung ist Neo4j schnell. Im Gegensatz zu Join-Operationen in relationalen Datenbanken oder Map/Reduce-Operationen anderer Datenbanken, ist die Zeit bei der Traversierung eines Graphen gleich. Die benötigen Daten sind nur einen Schritt entfernt und müssen nicht im großen Stil über Joins ermittelt und dann herausgefiltert werden – wie bei den meisten anderen Datenbanken, die wir gesehen haben. Es spielt keine Rolle, wie groß der Graph ist. Die Bewegung von Knoten A nach Knoten B ist immer nur ein Schritt, wenn sie eine Beziehung gemeinsam haben. Zum guten Schluss stellt die Enterprise-Edition Neo4j HA für Hochverfügbarkeit und hohes Lesevolumen bereit.

Neo4js Schwächen

Neo4j hat das ein oder andere Manko. Kanten in Neo4j können den Knoten nicht zurück auf sich selbst leiten. Wir haben auch festgestellt, dass die Wahl der Nomenklatur (*Knoten* statt *Vertex* und *Beziehung* statt *Kante*) die Kommunikation erschwert. Obwohl HA bei der Replikation glänzt, kann es nur einen vollständigen Graphen auf andere Server replizieren. Ein Sharding von Subgraphen ist momentan nicht möglich, weil die Größe des Graphen begrenzt ist (auch wenn dieses Limit bei vielen Milliarden liegt). Wenn Sie schließlich eine unternehmensfreundliche Open-Source-Lizenz (wie MIT) suchen, ist Neo4j möglicherweise nichts für Sie. Während die Community-Edition (die wir in den ersten beiden Tagen genutzt haben) unter der GPL steht, müssen Sie möglicherweise eine Lizenz erwerben, wenn Sie eine Produktionsumgebung mit den Enterprise-Tools (die HA und Backups umfassen) betreiben wollen.

Neo4j und CAP

Neo4j HA ist verfügbar und partitionstolerant (AP). Jeder Slave gibt nur das zurück, was er momentan besitzt, was temporär nicht dem entsprechen muss, was der Master-Knoten hat. Zwar können Sie die Update-Latenz reduzieren, indem Sie das Pull-Intervall des Slaves erhöhen, aber technisch ist es trotzdem nur schlussendlich konsistent. Aus diesem Grund wird Neo4j HA empfohlen, wenn es hauptsächlich um das Lesen von Daten geht.

Abschließende Gedanken

Neo4js Einfachheit ist wenig einladend, wenn man die Modellierung von Graph-Daten nicht gewohnt ist. Sie bietet eine leistungsfähige Open-Source-API, die seit Jahren produktiv genutzt wird, und dennoch gibt es nur relativ wenige Nutzer. Wir sehen die Ursache in fehlendem Wissen, da Graph-Datenbanken sich so natürlich in das Schema einfügen, nachdem wir Menschen Daten konzeptionalisieren. Wir stellen unsere Familien als Bäume dar und unsere Freunde als Graphen. Die meisten von uns stellen sich persönliche Beziehungen nicht als selbstreferenzierende Datentypen vor. Für bestimmte Problemklassen wie soziale Netzwerke ist Neo4j die offensichtlich beste Wahl. Doch Sie sollten sie auch bei nicht ganz so offensichtlichen Problemen ernsthaft in Erwägung ziehen – Sie könnten überrascht sein, wie leistungsfähig und einfach sie ist.

KAPITEL 8

Redis

Redis ist wie Schmierfett. Es wird häufig genutzt, um bewegliche Teil einzufetten, um sie geschmeidig zu halten, was die Reibung reduziert und die Gesamtfunktion verbessert. Wie auch immer die Maschinerie Ihres Systems aussieht, sie lässt sich sicher verbessern, wenn man sie ein wenig schmiert. Manchmal ist die Antwort auf Ihr Problem einfach ein gesundes Maß an Redis.

Redis (REmote DIctionary Service) wurde 2009 veröffentlicht und ist ein einfach zu nutzender Schlüssel/Wert-Speicher mit einem hoch entwickelten Befehlssatz. Und wenn es um Geschwindigkeit geht, ist Redis nur schwer zu schlagen. Leseoperationen sind schnell und Schreiboperationen sogar noch schneller. Bei einigen Benchmarks werden über 100000 `SET`-Operationen pro Sekunde gemessen. Der Entwickler von Redis, Salvatore Sanfilippo, bezeichnet sein Projekt als „Datenstrukturserver“, um seine nuancierte Verarbeitung komplexer Datentypen und anderer Features zu beschreiben. Die Untersuchung dieses superschnellen „Mehr-als-nur-Schlüssel/Wert-Speichers“ rundet unsere Betrachtung der modernen Datenbank-Landschaft ab.

8.1 Datenstrukturserver-Speicher

Es ist etwas schwierig zu klassifizieren, was Redis genau *ist*. Im Grunde handelt es sich um einen Schlüssel/Wert-Speicher, aber dieses einfache Etikett wird ihr nicht gerecht. Redis unterstützt fortgeschrittene Datenstrukturen, wenn auch nicht in dem Maß, das dokumentenorientierte Datenbanken bieten. Es unterstützt mengenbasierte Query-Operationen, wenn auch nicht mit der Granularität und Typ-Unterstützung, die man bei relationalen Datenbanken findet. Und natürlich ist es *schnell*, da es Geschwindigkeit gegen Dauerhaftigkeit tauscht.

Zusätzlich zu einem fortgeschrittenen Datenstrukturserver ist Redis eine blockierende Queue (oder Stack) und ein Publish/Subsribe-System. Es bietet konfigurierbare Regeln für Verfallsdaten, Beständigkeitsebenen und Replikationsoptionen. All das macht Redis eher zu einem Toolkit nützlicher Datenstruktur-Algorithmen und -Prozesse denn zu einem Mitglied eines bestimmten Datenbank-Genres.

Redis' ausgedehnte Liste von Client-Bibliotheken macht es zu einer einfachen Option für viele Programmiersprachen. Es ist nicht nur einfach zu nutzen, es ist ein Vergnügen. Wenn eine API ein echtes Nutzererlebnis für einen Programmierer ist, dann sollte Redis neben dem Mac Cube im Museum of Modern Art stehen.

An Tag 1 und 2 werden wir die Features, Konventionen und die Konfiguration von Redis untersuchen. Wir beginnen wie immer mit einfachen CRUD-Operationen und wenden uns dann recht schnell fortgeschrittenen Operationen mit leistungsfähigeren Datenstrukturen zu: Listen, Hashes, Sets und sortierten Sets. Wir entwickeln Transaktionen und manipulieren die Verfalls-Charakteristika der Daten. Wir verwenden Redis, um eine einfache Nachrichten-Queue aufzubauen und untersuchen ihre Publish/Subscribe-Funktionalität. Danach tauchen wir in die Konfigurations- und Replikationsoptionen von Redis ein und lernen, wie man eine der Anwendung angemessene Balance zwischen Dauerhaftigkeit und Geschwindigkeit herstellt.

Datenbanken werden häufig (und in zunehmendem Maße) im Zusammenspiel mit anderen Datenbanken genutzt. Redis wird in diesem Buch als Letztes vorgestellt, damit wir es in eben dieser Weise nutzen können. An Tag 3 bauen wir unser Capstone-System auf, eine Musik-Multidatenbank mit Redis, CouchDB, Neo4J und Postgres, die wir mit Hilfe von Node.js zusammenhalten.

8.2 Tag 1: CRUD und Datentypen

Da das Kommandozeilen-Interface von so primärer Bedeutung für das Redis-Entwicklungsteam ist – und von den Benutzern geliebt wird –, wollen wir den ersten Tag damit verbringen, uns viele der 124 zur Verfügung stehenden Befehle anzusehen. Von besonderer Bedeutung sind die Datentypen und wie man sie für anspruchsvollere Queries als nur „Gib mir den Wert für diesen Schlüssel zurück" nutzt.

Erste Schritte

Redis ist über einige Paket-Bauer wie Homebrew für Mac verfügbar, kann aber auch recht problemlos selbst kompiliert werden.[1] Wir arbeiten mit der Version 2.4. Sobald Sie es installiert haben, starten Sie den Server mit folgendem Aufruf:

```
$ redis-server
```

Standardmäßig läuft er nicht im Hintergrund, aber Sie können dafür sorgen, indem Sie schlicht ein & anhängen, oder öffnen Sie einfach ein anderes Terminal-Fenster. Führen Sie nun das Kommandozeilen-Tool aus, das die Verbindung mit dem Standard-Port 6379 automatisch herstellen sollte.

Nachdem die Verbindung steht, wollen wir den Server „anpingen“.

```
$ redis-cli

redis 127.0.0.1:6379> PING
PONG
```

Wenn keine Verbindung hergestellt werden kann, erhalten Sie eine Fehlermeldung. Geben Sie *help* ein, um sich eine Liste der Hilfsoptionen ausgeben zu lassen. Geben Sie *help* gefolgt von einem Leerzeichen ein und beginnen Sie dann, einen Befehl einzutippen. Wenn Sie die Redis-Befehle nicht kennen, können Sie dann die Tabulator-Taste drücken, um die verfügbaren Optionen durchzugehen.

```
redis 127.0.0.1:6379> help
Type: "help @<group>" to get a list of commands in <group>
      "help <command>" for help on <command>
      "help <tab>" to get a list of possible help topics
      "quit" to exit
```

Heute wollen wir Redis nutzen, um das Backend für einen URL-Shortener wie tinyurl.com oder bit.ly zu entwickeln. Ein URL-Shortener ist ein Service, der eine richtig lange URL nimmt und auf eine kurze Version innerhalb seiner eigenen Domain abbildet, etwa `http://www.myveryververylongdomain.com/somelongpath.php` auf `http://bit.ly/VLD`. Der Besuch dieser kurzen URL leitet die Benutzer auf die lange URL um. Der Benutzer muss keine langen Strings eingeben und der Urheber der URLs erhält einige Statistiken, wie etwa die Anzahl der Besuche.

Bei Redis können wir `SET` nutzen, um einen kurzen Code wie `7wks` mit einem Wert wie `http://www.sevenweeks.org` zu verknüpfen. `SET` verlangt immer zwei

1. http://redis.io

Parameter: einen Schlüssel und einen Wert. Um den Wert des Schlüssels abzurufen, brauchen Sie nur `GET` und den Namen des Schlüssels.

```
redis 127.0.0.1:6379> SET 7wks http://www.sevenweeks.org/
OK
redis 127.0.0.1:6379> GET 7wks
"http://www.sevenweeks.org/"
```

Um den Trafik zu reduzieren, können Sie mit `MSET` beliebig viele Schlüssel/Wert-Paare gleichzeitig setzen. Im folgenden Beispiel bilden wir Google.com auf `gog` und Yahoo.com auf `yah` ab.

```
redis 127.0.0.1:6379> MSET gog http://www.google.com yah http://www.yahoo.com
OK
```

Entsprechend ruft `MGET` mehrere Schlüssel ab und gibt die Werte in Form einer sortierten Liste zurück.

```
redis 127.0.0.1:6379> MGET gog yah
1) "http://www.google.com/"
2) "http://www.yahoo.com/"
```

Redis speichert zwar Strings, erkennt aber auch Integerwerte und stellt einige einfache Operationen für sie zur Verfügung. Wenn Sie laufend festhalten wollen, wie viele Shortener-Schlüssel Sie besitzen, können Sie einen Zähler anlegen und ihn mit dem `INCR`-Befehl inkrementieren.

```
redis 127.0.0.1:6379> SET count 2
OK
redis 127.0.0.1:6379> INCR count
(integer) 3
redis 127.0.0.1:6379> GET count
"3"
```

Zwar gibt `GET count` als String zurück, doch `INCR` erkennt ihn als Integerwert und addiert 1 hinzu. Alle Versuche, einen Nicht-Integerwert zu inkrementieren, scheitern kläglich.

```
redis 127.0.0.1:6379> SET bad_count "a"
OK
redis 127.0.0.1:6379> INCR bad_count
(error) ERR value is not an integer or out of range
```

Kann der Wert nicht in eine ganze Zahl umgewandelt werden, beschwert sich Redis zu Recht. Sie können auch um einen beliebigen Wert inkrementieren (`INCRBY`) oder dekrementieren (`DECR`, `DECRBY`).

Transaktionen

Wir haben Transaktionen bereits bei anderen Datenbanken (Postgres und Neo4j) kennengelernt. Redis' atomische `MULTI`-Blockbefehle verfolgen ein vergleichbares Konzept. Packt man zwei Operationen wie `SET` und `INCR` in einen einzelnen Block, wird er entweder als Ganzes erfolgreich verarbeitet oder gar nicht. Ein solcher Block wird niemals nur partiell verarbeitet.

Wir wollen eine weitere URL samt Kürzel einfügen und auch den Zähler inkrementieren, und das alles in einer einzigen Transaktion. Wir leiten die Transaktion mit dem `MULTI`-Befehl ein und führen sie dann mit `EXEC` aus.

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET prag http://pragprog.com
QUEUED
redis 127.0.0.1:6379> INCR count
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) (integer) 2
```

Wenn Sie `MULTI` nutzen, werden die Befehle (genau wie bei Postgres-Transaktionen) nicht ausgeführt, während Sie sie eingeben, sondern in einer Queue festgehalten und erst zum Schluss in der eingegebenen Reihenfolge verarbeitet.

Ähnlich dem `ROLLBACK` von SQL können Sie eine Transaktion mit dem `DISCARD`-Befehl anhalten. Dieser Befehl löscht die Transaktions-Queue. Im Gegensatz zum `ROLLBACK` wird die Datenbank aber nicht wieder zurückgestellt. Vielmehr werden die Transaktionen einfach nicht ausgeführt. Der Effekt ist der gleiche, auch wenn die zugrundeliegenden Mechanismen unterschiedliche Konzepte nutzen (Rollback oder Stornieren der Transaktion).

Komplexe Datentypen

Bisher haben wir kein besonders komplexes Verhalten gesehen. Das Speichern von String- und Integerwerten unter Schlüsseln – selbst als Transaktion – ist schön und gut, doch die meisten Probleme der Programmierung und Datenspeicherung arbeiten mit vielen verschiedenen Datentypen. Die Möglichkeit, Listen, Hashes, Sets (Mengen) und sortierte Sets zu speichern, erklärt die Beliebtheit von Redis. Und wenn Sie die komplexen Operationen kennengelernt haben, die Sie an ihnen vornehmen können, werden Sie sich dem wahrscheinlich anschließen.

Diese Collection-Datentypen können eine riesige Zahl von Werten (bis zu 2^32 Elemente, also mehr als 4 Milliarden) pro Schlüssel umfassen. Das

reicht aus, um alle Facebook-Accounts als Liste unter einem einzigen Schlüssel vorzuhalten.

Während einem einige Redis-Befehle etwas kryptisch erscheinen mögen, folgen sie doch einem guten Muster. `SET`-Befehle beginnen mit `S`, Hashes mit `H` und sortierte Sets mit `Z`. Listenbefehle fangen entweder mit `L` (für links) oder `R` (rechts an), je nach Richtung der Operation (wie etwa `LPUSH`).

Hash

Hashes sind wie verschachtelte Redis-Objekte, die eine beliebige Anzahl von Schlüssel/Wert-Paaren enthalten können. Wir wollen einen Hash nutzen, um die Nutzer nachzuhalten, die sich für unseren URL-Shortener-Service anmelden.

Hashes sind nett, weil wir mit ihrer Hilfe die Daten ohne künstliche Schlüssel-Präfixe speichern können. (Beachten Sie, dass wir den Doppelpunkt [:] in unserem Schlüssel verwenden. Er ist ein gültiges Zeichen, der einen Schlüssel in logische Segmente aufteilt. Das ist aber nur eine Konvention ohne eine tiefere Bedeutung für Redis.)

```
redis 127.0.0.1:6379> MSET user:eric:name "Eric Redmond" user:eric:password s3cret
OK
redis 127.0.0.1:6379> MGET user:eric:name user:eric:password
1) "Eric Redmond"
2) "s3cret"
```

Anstelle separater Schlüssel können wir einen Hash erzeugen, der seine eigenen Schlüssel/Wert-Paare enthält.

```
redis 127.0.0.1:6379> HMSET user:eric name "Eric Redmond" password s3cret
OK
```

Wir müssen nur einen einzigen Redis-Schlüssel nachhalten, um alle Werte im Hash abzurufen.

```
redis 127.0.0.1:6379> HVALS user:eric
1) "Eric Redmond"
2) "s3cret"
```

Wir können aber auch alle Hash-Schlüssel abrufen.

```
redis 127.0.0.1:6379> HKEYS user:eric
1) "name"
2) "password"
```

Oder wir rufen einen einzelnen Wert ab, indem wir den Redis-Schlüssel gefolgt vom Hash-Schlüssel angeben. Nachfolgend erhalten wir nur das Passwort zurück.

```
redis 127.0.0.1:6379> HGET user:eric password
"s3cret"
```

Im Gegensatz zu den Dokument-Datenspeichern Mongo und CouchDB können Hashes bei Redis nicht verschachtelt werden (was auch für alle anderen komplexen Datentypen, etwa Listen, gilt). Mit anderen Worten: Sie können in Hashes nur Stringwerte speichern.

Mit anderen Befehlen können Sie Hashfelder löschen (`HDEL`) Integerfelder um einen beliebigen Wert erhöhen (`HINCRBY`) oder die Anzahl der Felder in einem Hash bestimmen (`HLEN`).

Listen

Listen enthalten mehrere geordnete Werte und können sowohl als Queues (zuerst rein, zuerst raus) oder als Stacks (zuletzt rein, zuerst raus) fungieren. Es gibt auch raffinierte Aktionen zum Einfügen von Elementen mitten in die Liste, zur Größenbeschränkung der Liste und zum Verschieben von Werten zwischen Listen.

Da unser URL-Shortener nun Benutzer nachhalten kann, sollen sie nun eine Wunschliste mit den URLs speichern können, die sie gerne besuchen wollen. Um eine Liste der gekürzten Websites aufzubauen, verwenden wir den Schlüssel `USERNAME:wishlist` und schieben mit einer Push-Operation eine beliebige Anzahl von Werten rechts in die Liste (also an das Ende).

```
redis 127.0.0.1:6379> RPUSH eric:wishlist 7wks gog prag
(integer) 3
```

Wie bei den meisten Einfüge-Operationen für Collections gibt der Redis-Befehl die Anzahl der abgelegten Werte zurück. Anders ausgedrückt haben wir drei Werte mit Push in der Liste abgelegt, also wird der Wert 3 zurückgegeben. Wir können die Länge der Liste jederzeit mit `LLEN` ermitteln.

Mit Hilfe des Listen-Bereichsbefehls `LRANGE` können wir einen Teil der Liste abrufen, indem wir die erste und die letzte gewünschte Position festlegen. Alle Listenoperationen verwenden bei Redis einen nullbasierten Index. Ein negativer Wert steht für die Position vom Ende der Liste.

```
redis 127.0.0.1:6379> LRANGE eric:wishlist 0 -11) "7wks"
2) "gog"
3) "prag"
```

LREM löscht passende Werte unter dem angegebenen Schlüssel. Sie müssen auch eine Zahl angeben, damit der Befehl weiß, wie viele Treffer er löschen soll. Setzt man den Wert (so wie wir das hier tun) auf 0, werden alle entfernt:

```
redis 127.0.0.1:6379> LREM eric:wishlist 0 gog
```

Hat der Zähler einen Wert größer 0, wird die entsprechende Anzahl von Treffern gelöscht. Ein negativer Wert entfernt die entsprechende Zahl von Treffern vom Ende (der rechten Seite) gelöscht.

Um jeden Wert in der Reihenfolge zu entfernen und abzurufen (wie in einer Queue), können wir sie mit einer Pop-Operation von der linken Seite der Liste (dem Kopf) entfernen.

```
redis 127.0.0.1:6379> LPOP eric:wishlist
"7wks"
```

Soll er als Stack fungieren, nachdem Sie mit `RPUSH` Werte abgelegt haben, würden Sie sie mit `RPOP` wieder vom Ende der Liste entfernen. All diese Operationen werden in einer konstanten Zeit ausgeführt.

Für die obige Kombination von Befehlen können Sie mit `LPUSH` und `RPOP` den gleichen Effekt erzielen (eine Queue) bzw. mit `LPUSH` und `LPOP` einen Stack nachahmen.

Nehmen wir an, wir wollen Werte aus unserer Wunschliste in eine andere Liste besuchter Sites verschieben. Um diese Verschiebung atomisch durchzuführen, könnten wir die Pop- und Push-Operationen in einen Multiblock packen. Bei Ruby würde das etwa wie folgt aussehen (Sie können hier nicht die Kommandozeile nutzen, weil die mit Pop abgerufenen Werte gesichert werden müssen, weshalb wir das `redis-rb`-Gem genutzt haben):

```
redis.multi do
  site = redis.rpop('eric:wishlist')
  redis.lpush('eric:visited', site)
end
```

Doch Redis bietet auch einen einzelnen Befehl, der Werte vom Ende einer Liste nimmt und an den Anfang einer anderen Liste stellt. Dieser Befehl heißt `RPOPLPUSH` (Pop rechts, Push links).

```
redis 127.0.0.1:6379> RPOPLPUSH eric:wishlist eric:visited
"prag"
```

Wenn Sie sich nun die wishlist ansehen, ist `prag` verschwunden und liegt nun in `visited`. Das ist ein nützlicher Mechanismus zum Queuing von Befehlen.

Wenn Sie in der Redis-Dokumentation nach RPOPRPUSH, LPOPLPUSH und LPOPRPUSH suchen, werden Sie konsterniert feststellen, dass es sie nicht gibt. RPOPLPUSH ist Ihre einzige Option, und Sie müssen Ihre Listen entsprechend aufbauen.

Blockierende Listen

Da unser URL-Shortener langsam Fahrt aufnimmt, wollen wir ihn um einige soziale Aktivitäten erweitern, etwa ein Echtzeit-Kommentarsystem, bei dem Leute die von ihnen besuchten Websites kommentieren können.

Wir wollen ein einfaches Messaging-System schreiben, bei dem mehrere Clients Kommentare in eine Queue schieben können und ein anderer Client (der Digester) Nachrichten aus der Queue entnimmt. Der Digester soll nur auf neue Kommentare warten und sie entnehmen, sobald sie eintreffen. Redis stellt für solche Aufgaben einige blockierende Befehle zur Verfügung.

Zuerst öffnen wir ein weiteres Terminal-Fenster und starten einen weiteren redis-cli-Client. Das ist unser Digester. Der Befehl, der solange blockiert, bis ein Wert vorhanden ist, lautet BRPOP. Er verlangt den Schlüssel, aus dem ein Wert über Pop abgerufen werden soll, sowie einen Timeout in Sekunden, den wir mit fünf Minuten festlegen.

```
redis 127.0.0.1:6379> BRPOP comments 300
```

Dann wechseln wir zurück in die erste Console und pushen eine Nachricht in die Kommentare.

```
redis 127.0.0.1:6379> LPUSH comments "Prag is great! I buy all my books there."
```

Wechseln Sie zurück in die Digester-Console. Zwei Zeilen werden zurückgeliefert: der Schlüssel und der mit Pop abgerufene Wert. Es wird auch angegeben, wie lange der Befehl blockiert hat.

```
1) "comments"
2) "Prag is great! I buy all my books there."
(50.22s)
```

Es gibt auch eine blockierende Version eines linken Pops (BLPOP) und eines rechten Pops/linken Pushs (BRPOPLPUSH).

Set

Unser URL-Shortener entwickelt sich gut, doch es wäre schön, wenn wir gängige URLs in irgendeiner Weise gruppieren könnten.

Sets (Mengen) sind ungeordnete Collections ohne Duplikate. Sie sind eine ausgezeichnete Wahl für komplexe Operationen zwischen zwei oder mehr Schlüssel-Werten, z. B. der Bildung von Vereinigungs- und Schnittmengen.

Wenn wir URL-Sets mit einem gemeinsamen Schlüssel kategorisieren wollen, können wir mehrere Werte mit `SADD` einfügen.

```
redis 127.0.0.1:6379> SADD news nytimes.com pragprog.com
(integer) 2
```

Redis hat zwei Werte hinzugefügt. Wir können den vollständigen Set (ohne bestimmte Reihenfolge) über `SMEMBERS` abrufen.

```
redis 127.0.0.1:6379> SMEMBERS news
1) "pragprog.com"
2) "nytimes.com"
```

Nun fügen wir eine weitere Kategorie namens *tech* für technikorientierte Seiten hinzu.

```
redis 127.0.0.1:6379> SADD tech pragprog.com apple.com
(integer) 2
```

Um die Schnittmenge der Websites zu ermitteln, die sowohl nachrichten-, als auch technikorientiert sind, nutzen wir den `SINTER`-Befehl.

```
redis 127.0.0.1:6379> SINTER news tech
1) "pragprog.com"
```

Genauso einfach können wir Werte in einem Set mit Werten aus einem anderen Set aussortieren. Um alle News-Sites aufzuspüren, die keine Tech-Sites sind, verwenden wir `SDIFF`:

```
redis 127.0.0.1:6379> SDIFF news tech
1) "nytimes.com"
```

Wir können auch die Vereinigungsmenge von News- und Tech-Sites bilden. Da es sich um ein Set handelt, werden Duplikate entfernt.

```
redis 127.0.0.1:6379> SUNION news tech
1) "apple.com"
2) "pragprog.com"
3) "nytimes.com"
```

Diese Wertemenge kann auch direkt in einem neuen Set gespeichert werden (`SUNIONSTORE ziel schlüssel [schlüssel ...]`).

```
redis 127.0.0.1:6379> SUNIONSTORE websites news tech
```

Das ist auch nützlich, um die Werte eines Schlüssels unter einen anderen Schlüssel zu kopieren, z. B. `SUNIONSTORE news_copy news`. Vergleichbare Befehle gibt es zum Speichern von Vereinigungs- (`SINTERSTORE`) und Differenzmengen (`SDIFFSTORE`).

Genau wie `RPOPLPUSH` Werte von einer Liste in eine andere verschiebt, bewegt `SMOVE` Werte aus einem Set in einen anderen (nur dass man sich das besser merken kann).

Und wie `LLEN` die Länge einer Liste bestimmt, zählt `SCARD` (Set-Kardinalität) das Set durch (nur dass man sich das schlechter merken kann.)

Da Sets nicht geordnet sind, gibt es keine Links-, Rechts- oder andere Positionsbefehle. Um einen zufälligen Wert per Pop aus einem Set abzurufen, verwenden Sie `SPOP schlüssel` und Sie löschen Werte mit `SREM schlüssel wert [wert ...]`.

Im Gegensatz zu Listen gibt eine keine blockierenden Befehle für Sets.

Sortierte Sets

Während die anderen Redis-Datentypen, die wir bisher betrachtet haben, sich leicht auf gängige Programmiersprachen-Konstrukte abbilden lassen, haben sortierte Sets ein wenig von allen bisher genutzten Datentypen. Sie sind sortiert wie Listen und eindeutig wie Sets. Sie besitzen Schlüssel/Wert-Paare wie Hashes, doch anstelle von Stringfeldern gibt es numerische Gewichtungen, die die Reihenfolge der Werte bestimmen. Sie können sich sortierte Sets wie eine priorisierte Queue mit wahlfreiem Zugriff vorstellen. Diese Fähigkeit hat aber auch ihren Preis. Intern halten sortierte Sets die Werte sortiert vor, d. h., das Einfügen dauert log(N) lang (wobei N die Größe des Sets ist) und nicht die konstante Zeitspanne von Hashes und Listen.

Als Nächstes wollen wir die Beliebtheit einzelner Kurzcodes nachhalten. Jedesmal, wenn jemand eine URL besucht, wird sein Zählerstand erhöht. Genau wie bei einem Hash verlangt das Hinzufügen eines Wertes in einen sortierten Set neben dem Namen des Redis-Schlüssels zwei Werte: den Stand und das fragliche Element.

```
redis 127.0.0.1:6379> ZADD visits 500 7wks 9 gog 9999 prag
(integer) 3
```

Um den Zählerstand zu erhöhen, können wir ihn entweder erneut eintragen (was den Zählerstand einfach nur aktualisiert, ohne einen neuen Wert ein-

zufügen) oder wir inkrementieren ihn um irgendeine Zahl und erhalten den neuen Wert zurück.

```
redis 127.0.0.1:6379> ZINCRBY visits 1 prag
"10000"
```

Sie können auch dekrementieren, indem Sie eine negative Zahl an `ZINCRBY` übergeben.

Wertebereiche

Um Werte aus unserem visits-Set abzurufen, können wir einen Bereichsbefehl, `ZRANGE`, verwenden, der die Werte nach Position sortiert zurückliefert (genau wie der `LRANGE`-Befehl für Listen). Im Falle eines sortierten Sets ergibt sich die Position aus dem aktuellen Zählerstand. Um also die beiden meistbesuchten Sites abzurufen, verwenden Sie Folgendes:

```
redis 127.0.0.1:6379> ZRANGE visits 0 1
1) "gog"
2) "7wks"
```

Beachten Sie, dass die Zählung bei 0 beginnt. Um sich auch die jeweiligen Zählerstände ausgeben zu lassen, hängen Sie `WITHSCORES` an den obigen Code an. Um die Reihenfolge umzukehren, fügen Sie das Wort `REV` ein, etwa `ZREVRANGE`.

```
redis 127.0.0.1:6379> ZREVRANGE visits 0 -1 WITHSCORES
1) "prag"
2) "10000"
3) "7wks"
4) "500"
5) "gog"
6) "9"
```

Doch bei einem sortierten Set werden Sie den Wertebereich nicht für die Position, sondern eher für den Zählerstand festlegen wollen. `ZRANGEBYSCORE` hat eine etwas andere Syntax als `ZRANGE`. Da die unteren und oberen Bereichsangaben standardmäßig *inklusive* sind, können wir sie *ausschließen*, indem wir ihnen eine öffnende runde Klammer voranstellen. Nachfolgend werden also alle Werte zwischen `9 <= score <= 10000` zurückgegeben:

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits 9 9999
1) "gog"
2) "7wks"
```

Doch der folgende Code gibt alles zwischen `9 < score <= 10000` zurück:

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits (9 9999
1) "7wks"
```

Wir können sowohl positive als auch negative Werte für die Bereiche angeben, auch Unendlich. Die folgende Zeile liefert das gesamte Set zurück.

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits -inf inf
```

Auch hier können Sie die Reihenfolge mit `ZREVRANGEBYSCORE` umkehren.

Sie können die Daten nicht nur nach Position (Index) und Zählerstand abrufen, sondern auch mit `ZREMRANGEBYRANK` und `ZREMRANGEBYSCORE` entsprechend löschen.

Vereinigung

Genau wie bei einem normalen Set können wir einen Zielschlüssel erzeugen, der die Vereinigungs- oder Schnittmenge eines oder mehrerer Schlüssel enthält. Das ist einer der komplexeren Redis-Befehle, da er die Schlüssel nicht nur vereinen muss (eine relativ einfache Operation), sondern (möglicherweise) abweichende Positionswerte verschmelzen muss. Die Vereinigungs-Operation (Union) sieht wie folgt aus:

```
ZUNIONSTORE ziel anzahlschlüssel schlüssel [schlüssel ...]
  [WEIGHTS gewichtung [gewichtung ...]] [AGGREGATE SUM|MIN|MAX]
```

`ziel` ist der Schlüssel, in dem gespeichert werden soll, und `schlüssel` steht für einen oder mehrere Schlüssel, die vereinigt werden sollen. `anzahlschlüssel` ist einfach die Anzahl der Schlüssel, die vereinigt werden soll, während die `gewichtung` eine optionale Zahl ist, mit der die Gewichtung jedes relativen Schlüssel multipliziert wird (bei zwei Schlüsseln können Sie zwei Gewichtungen angeben und so weiter). Das abschließende `aggregate` ist eine optionale Regel, die jeden gewichteten Score auflöst und standardmäßig summiert (Sie können aber auch das Minimum oder Maximum vieler Scores bestimmen).

Wir wollen diesen Befehl nutzen, um die Bedeutung eines sortierten Sets mit Kurzcodes zu bestimmen.

Zuerst erzeugen wir einen weiteren Schlüssel, der unsere Kurzcodes nach Stimmen gewichtet. Jeder Besucher einer Seite kann abstimmen, ob er die Site mag oder nicht, und jede Stimme gibt einen Punkt.

```
redis 127.0.0.1:6379> ZADD votes 2 7wks 0 gog 9001 prag
(integer) 3
```

Wir wollen die wichtigsten Websites in unserem System als Kombination von Stimmen und Besuchen ermitteln. Die Abstimmung ist wichtig, doch in etwas kleinerem Maßstab haben auch die Website-Besuche ein gewisses Gewicht (vielleicht sind die Leute von der Website so entzückt, dass sie vergessen, ihre Stimme abzugeben). Wir wollen zwei Arten von Scores addieren, um einen neuen „Wichtigkeits-Score“ zu berechnen. Die abgegebenen Stimmen erhalten dabei die doppelte Wichtung, d. h., sie werden mit 2 multipliziert.

```
ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE SUM
(integer) 3
redis 127.0.0.1:6379> ZRANGEBYSCORE importance -inf inf WITHSCORES
1) "gog"
2) "9"
3) "7wks"
4) "504"
5) "prag"
6) "28002"
```

Dieser Befehl ist auch in anderer Hinsicht mächtig. Wenn Sie zum Beispiel alle Scores in einem Set verdoppeln müssen, können Sie den Schlüssel mit einem Gewicht von 2 vereinen und wieder zurückschreiben.

```
redis 127.0.0.1:6379> ZUNIONSTORE votes 1 votes WEIGHTS 2
(integer) 2
redis 127.0.0.1:6379> ZRANGE votes 0 -1 WITHSCORES
1) "gog"
2) "0"
3) "7wks"
4) "4"
5) "prag"
6) "18002"
```

Sortierte Sets kennen einen ähnlichen Befehl (`ZINTERSTORE`) für Schnittmengen.

Verfallszeit

Ein typischer Anwendungsfall für ein Schlüssel/Wert-System wie Redis ist ein Cache für den schnellen Zugriff auf Daten, bei denen der Preis für den Abruf oder die Berechnung zu hoch ist. Ein Verfallsdatum hilft, zu verhindern, dass die Gesamtzahl der Schlüssel ungehindert anwächst, indem Redis angewiesen wird, ein Schlüssel/Wert-Paar nach einer gewissen Zeit zu löschen.

Sie legen die Verfallszeit mit dem `EXPIRE`-Befehl fest, der einen existierenden Schlüssel und eine Zeitspanne (TTL, time to live) in Sekunden verlangt. Nachfolgend setzen wir einen Schlüssel und legen seine Verfallszeit mit 10 Sekunden fest. Wir können dann mit `EXISTS` prüfen, ob der Schlüssel noch

existiert, und innerhalb dieser 10 Sekunden wird eine 1 (wahr) zurückgegeben. Wenn Sie ein wenig warten, wird schließlich 0 (falsch) zurückgeliefert.

```
redis 127.0.0.1:6379> SET ice "I'm melting..."
OK
redis 127.0.0.1:6379> EXPIRE ice 10
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 0
```

Das Setzen und Verfallenlassen von Schlüsseln ist bei Redis so weit verbreitet, dass es dafür einen eigenen Kurzbefehl namens `SETEX` gibt.

```
redis 127.0.0.1:6379> SETEX ice 10 "I'm melting..."
```

Sie können mit `TTL` feststellen, wie lange ein Schlüssel noch zu leben hat. Wenn Sie die Verfallszeit von `ice` wie oben beschrieben festlegen und dann seine TTL prüfen, erhalten Sie die noch verbliebene Restzeit in Sekunden zurück.

```
redis 127.0.0.1:6379> TTL ice
(integer) 4
```

Bevor der Schlüssel abläuft, können Sie den Timeout noch zu jeder Zeit mit `PERSIST schlüssel` aufheben.

```
redis 127.0.0.1:6379> PERSIST ice
```

Soll ein Schlüssel zu einem bestimmten Zeitpunkt ablaufen, können Sie mit `EXPIREAT` einen *nix-Zeitstempel (Sekunden seit dem 1.1.1970) angeben. Mit anderen Worten ist `EXPIREAT` für absolute Timeouts gedacht, während `EXPIRE` relative Timeouts verwendet.

Ein gängiger Trick, nur aktiv genutzte Schlüssel vorzuhalten, besteht darin, die Ablaufzeit immer dann zu aktualisieren, wenn ein Wert abgerufen wird. Das ist der sog. MRU (moste recently used, zu deutsch etwa zuletzt genutzt)-Caching-Algorithmus, der sicherstellt, dass nur die zuletzt genutzten Schlüssel in Redis verbleiben, während die anderen ganz normal auslaufen.

Namensräume

Bisher haben wir nur mit einem Namensraum gearbeitet. Genau wie bei Riak-Buckets müssen wir Schlüssel manchmal durch Namensräume trennen. Wenn Sie zum Beispiel einen internationalisierten Schlüssel/Wert-Speicher

entwickeln, können Antworten in verschiedenen Sprachen in unterschiedlichen Namensräumen abgelegt werden. Der Schlüssel `greeting` könnte in einem deutschen Namensraum „Guten Tag“ und in einem französischen „Bonjour“ sein. Wenn ein Benutzer seine Sprache wählt, zieht sich die Anwendung einfache alle Werte aus dem zugewiesenen Namensraum.

Die Redis-Nomenklatur bezeichnet einen Namensraum als *Datenbank* und verwendet eine Zahl als Schlüssel. Bisher haben wir mit dem Standard-Namensraum 0 (auch als Datenbank 0 bekannt) gearbeitet. Hier setzen wir `greeting` auf das englische `hello`.

```
redis 127.0.0.1:6379> SET greeting hello
OK
redis 127.0.0.1:6379> GET greeting
"hello"
```

Wechseln wir mit dem `SELECT`-Befehl zu einer anderen Datenbank, ist dieser Schlüssel nicht verfügbar.

```
redis 127.0.0.1:6379> SELECT 1
OK
redis 127.0.0.1:6379[1]> GET greeting
(nil)
```

Und wenn wir in diesem Namensraum einen Wert setzen, hat das keinerlei Auswirkungen auf den Originalwert.

```
redis 127.0.0.1:6379[1]> SET greeting "guten tag"
OK
redis 127.0.0.1:6379[1]> SELECT 0
OK
redis 127.0.0.1:6379> GET greeting
"hello"
```

Da alle Datenbanken in der gleichen Serverinstanz laufen, erlaubt uns Redis, Schlüssel mit dem `MOVE`-Befehl zwischen ihnen zu verschieben. Hier verschieben wir `greeting` nach Datenbank 2:

```
redis 127.0.0.1:6379> MOVE greeting 2
(integer) 2
redis 127.0.0.1:6379> SELECT 2
OK
redis 127.0.0.1:6379[2]> GET greeting
"hello"
```

Das ist nützlich, wenn unterschiedliche Anwendungen einen einzelnen Redis-Server nutzen und gleichzeitig Daten untereinander austauschen wollen.

Und mehr

Redis besitzt viele weitere Befehle für Operationen wie das Umbenennen von Schlüsseln (`RENAME`), die Bestimmung des Typs eines Schlüsselwertes (`TYPE`) und das Löschen eines Schlüssel/Wert-Paares (`DEL`). Es gibt auch einen (unglaublich gefährlichen) `FLUSHDB`-Befehl, der alle Schlüssel aus dieser Redis-Datenbank löscht, sowie dessen apokalyptischen Cousin `FLUSHALL`, der alle Schlüssel aus allen Redis-Datenbanken löscht. Eine vollständige Liste aller Redis-Befehle finden Sie in der Online-Dokumentation.

Was wir am ersten Tag gelernt haben

Die Datentypen von Redis und die komplexen Queries, die es durchführen kann, machen Redis zu weit mehr als einem normalen Schlüssel/Wert-Speicher. Es kann als Stack, Queue oder priorisierte Queue fungieren, ein Objektspeicher sein (über Hashes) und kann sogar komplexe Set-Operationen wie Vereinigungs-, Schnitt- und Differenzmenge durchführen. Es stellt viele atomische Befehle zur Verfügung und dafür stellt es auch einen Transaktionsmechanismus bereit. Die Möglichkeit, Schlüssel nach einer gewissen Zeit verfallen zu lassen, ist fest integriert und für Caches nützlich.

Tag 1: Selbststudium

Finden Sie heraus

Finden Sie die vollständige Dokumentation der Redis-Befehle sowie die Big-O-notierte (O(x)) Zeitkomplexität unter den Befehls-Details.

Machen Sie Folgendes

1. Installieren Sie einen Treiber für Ihre Lieblings-Programmiersprache und stellen Sie die Verbindung mit dem Redis-Server her. Fügen Sie einen Wert ein und erhöhen Sie ihn. Verwenden Sie dazu eine Transaktion.
2. Entwickeln Sie mit einem Treiber Ihrer Wahl ein Programm, das eine blockierende Liste liest und irgendwo ausgibt (Console, Datei, Socket.io und so weiter), und ein weiteres Programm, das an die gleiche Liste schreibt.

8.3 Tag 2: Fortgeschrittene Nutzung, Distribution

An Tag 1 wurde uns Redis als Datenstrukturserver vorgestellt. Heute bauen wir auf dieser Grundlage auf und sehen uns einige fortgeschrittene Funktionen an, die uns Redis bietet, etwa das Pipelining, das Publish/Subscribe-Modell, die Systemkonfiguration und die Replikation. Darüber hinaus sehen wir uns an, wie man ein Redis-Cluster aufbaut und viele Daten schnell speichert. Außerdem nutzen wir eine fortgeschrittene Technik mit Bloomfiltern.

Ein einfaches Interface

Mit 20000 Zeilen Quellcode ist Redis ein vergleichsweise einfaches Projekt. Doch trotz der Codegröße besitzt es ein einfaches Interface, das die gleichen Strings akzeptiert, die wir in der Console eingegeben haben.

Telnet

Wir können ohne Kommandozeilen-Interface arbeiten, indem wir die Befehle per TCP über telnet eingeben und mit Carriage Return/Linefeed (CRLF, oder `\r\n`) abschließen.

redis/telnet.sh

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SET test hello
① +OK
GET test
② $5
hello
SADD stest 1 99
③ :2
SMEMBERS stest
④ *2
$1
```

```
1
$2
99

Ctrl-]
```

Wir sehen, dass unsere Eingabe dem entspricht, was wir in der Console angegeben haben, aber die Console hat die Antworten ein wenig aufgeräumt.

① Redis liefert den Status `OK` zurück, dem ein Pluszeichen vorangeht.

② Bevor es den String *hello* zurückgibt, sendet es `$5`, was bedeutet, dass der nachfolgende String fünf Zeichen lang ist.

③ Nach dem Einfügen von zwei Werten in den Test-Schlüssel wird eine `2` zurückgegeben. Vor ihr steht `:`, was einen Integerwert repräsentiert (zwei Werte wurden erfolgreich hinzugefügt).

④ Abschließend haben wir zwei Elemente abgerufen. Die erste zurückgelieferte Zeile beginnt mit einem Sternchen und der Zahl 2, was bedeutet, dass zwei komplexe Werte zurückgeliefert werden. Die nächsten beiden Zeilen entsprechen der Rückgabe des *hello*-Beispiels, geben aber den String *1* und dann den String *99* zurück.

Pipelining

Wir können unsere eigenen Strings auch einen nach dem anderen über den BSD-Befehl BSD netcat (`nc`) streamen, der auf vielen Unix-Maschinen bereits vorinstalliert ist. Bei netcat müssen wir eine Zeile explizit mit CRLF abschließen (telnet hat das für uns automatisch erledigt). Nachdem der echo-Befehl gesendet wurde, warten wir eine Sekunde, damit der Redis-Server Zeit hat, uns eine Antwort zu liefern. Einige `nc`-Implementierungen kennen die Option `-q`, mit der man herausfinden kann, ob eine Pause notwendig ist. Aber nicht alle Implementierungen unterstützen diese Option, also probieren Sie es aus.

```
$ (echo -en "ECHO hello\r\n"; sleep 1) | nc localhost 6379
$5
hello
```

Wir können diesen Grad der Kontrolle zu unserem Vorteil nutzen, indem wir unsere Befehle durch eine *Pipeline* schicken oder mehrere Befehle in einem einzigen Request senden.

```
$ (echo -en "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
+PONG
+PONG
+PONG
```

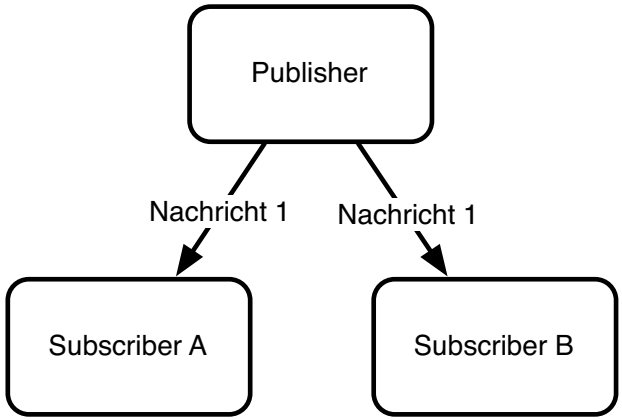


Abbildung 37: Ein Publisher sendet eine Nachricht an alle Subscriber

Das ist wesentlich effizienter, als immer nur einen einzelnen Befehl zu senden und sollte überall da genutzt werden, wo es einen Sinn ergibt – insbesondere bei Transaktionen. Achten Sie nur darauf, jeden Befehl mit `\r\n` abzuschließen, weil das die vom Server erwarteten Trennzeichen sind.

Publish/Subscribe

Gestern konnten wir mit Hilfe einer Liste eine rudimentäre blockierende Queue aufbauen. Die in die Queue geschriebenen Daten konnten durch einen blockierenden Pop-Befehl gelesen werden. Mit Hilfe dieser Queue haben wir ein sehr einfaches Publish/Subscribe-Modell aufgebaut. Eine beliebige Zahl von Nachrichten konnte in die Queue geschoben werden und ein einzelner Queue-Reader hat sie verarbeitet, sobald sie verfügbar waren. Das ist leistungsfähig, aber etwas beschränkt. In vielen Fällen wünscht man sich das umgekehrte Verhalten, d. h., mehrere „Abonnenten“ (Subscriber) wollen die Meldungen eines einzelnen „Herausgebers“ (Publisher) lesen (siehe Abbildung 37, *Ein Publisher sendet eine Nachricht an alle Subscriber*). Redis stellt einige spezialisierte Publish/Subscribe-Befehle zur Verfügung.

Wir wollen den Kommentar-Mechanismus, den wir gestern mit blockierenden Listen aufgebaut haben, verbessern, indem wir es den Benutzern erlauben, einen Kommentar an mehrere Subscriber zu senden (und nicht nur an einen). Wir beginnen mit einigen Subscribern, die sich mit einem Schlüssel verbinden, der in Publisher/Subscriber-Nomenklatur als *Channel* (Kanal) bezeichnet wird. Lassen Sie uns zwei weitere Clients starten und den com-

ments-Channel „abonnieren“. Die Subscribe-Operation blockiert die Kommandozeile.

```
redis 127.0.0.1:6379> SUBSCRIBE comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
```

Mit zwei Subscribern veröffentlichen wir einen beliebigen String, den wir als Nachricht versenden wollen, an den `comments`-Channel. Der `PUBLISH`-Befehl gibt den Integerwert 2 zurück, was bedeutet, dass zwei Subscriber die Nachricht empfangen haben.

```
redis 127.0.0.1:6379> PUBLISH comments "Check out this shortcoded site! 7wks"
(integer) 2
```

Beide Subscriber empfangen eine *Multibulk-Antwort* (eine Liste) mit drei Elementen: den String „message“, den Channel-Namen und die veröffentlichte Nachricht.

```
1) "message"
2) "comments"
3) "Check out this shortcoded site! 7wks"
```

Soll der Client nicht länger Nachrichten empfangen, können Sie den Befehl `UNSUBSCRIBE comments` ausführen, um die Verbindung zum comments-Kanal zu trennen. Sie können auch einfach `UNSUBSCRIBE` (ohne weitere Parameter) angeben, um die Verbindung mit allen Kanälen zu trennen. Unter `redis-cli` müssen Sie allerdings `Ctrl+C` drücken, um die Verbindung zu beenden.

Server-Info

Bevor Sie sich daran machen, die Redis-Systemeinstellungen zu ändern, lohnt sich ein kurzer Blick auf den `INFO`-Befehl, weil die Änderung von Einstellungen auch einige dieser Werte ändert. `INFO` gibt eine Liste mit Serverdaten aus, einschließlich Version, Prozess-ID, genutztem Speicher und Uptime.

```
redis 127.0.0.1:6379> INFO
redis_version:2.4.5
redis_git_sha1:00000000
redis_git_dirty:0
arch_bits:64
multiplexing_api:kqueue
process_id:54046
uptime_in_seconds:4
uptime_in_days:0
lru_clock:1807217
...
```

Sie sollten den Befehl in diesem Kapitel wiederholt nutzen, weil er eine nützliche Momentaufnahme mit globalen Informationen und Einstellungen des Servers liefert. Er liefert sogar Informationen zu Langlebigkeit, Speicherfragmentierung und zum Status des Replikationsservers.

Redis-Konfiguration

Bisher haben wir Redis nur in der Standardkonfiguration genutzt. Für einen Großteil von Redis' Stärke ist aber dessen Konfigurierbarkeit verantwortlich, die es Ihnen erlaubt, die Einstellungen für Ihren Anwendungsfall zu optimieren. Die Datei `redis.conf`, die mit der Distribution mitgeliefert wird (bei *nix-Systemen in `/etc/redis` zu finden), ist größtenteils selbsterklärend, weshalb wir uns nur einen Teil der Datei ansehen und nacheinander einige gängige Einstellungen durchgehen.

```
daemonize no
port 6379
loglevel verbose
logfile stdout
database 16
```

Standardmäßig ist `daemonize` auf *no* gesetzt, weshalb der Server immer im Vordergrund startet. Das ist zu Testzwecken ganz nett, für den Produktiveinsatz aber nicht geeignet. Setzt man diesen Wert auf *yes*, startet der Server im Hintergrund und seine Prozess-ID wird in eine pid-Datei geschrieben.

Die nächste Zeile enthält den Standard-Port für diesen Server (6379). Das ist dann besonders nützlich, wenn mehrere Redis-Server auf einer einzelnen Maschine laufen.

`loglevel` ist mit `verbose` voreingestellt, aber im Produktionsbetrieb sollten Sie es auf `notice` oder `warning` stellen. `logfile` ist mit stdout (standard output, d. h. die Console) voreingestellt, doch im daemonize-Modus müssen Sie hier einen Dateinamen angeben.

`database` legt die Zahl der Datenbanken fest, die Redis zur Verfügung stehen. Wie man zwischen den Datenbanken wechselt, haben wir ja gestern gezeigt. Wenn Sie garantiert nur einen einzigen Datenbank-Namensraum brauchen, können Sie diesen Wert auf 1 setzen.

Dauerhaftigkeit

Redis besitzt einige Persistenz-Optionen. An erster Stelle steht gar keine Persistenz, was bedeutet, dass die Werte einfach im Hauptspeicher gehalten werden. Wenn Sie einen einfachen Caching-Server betreiben, ist das eine akzeptable Wahl, weil Dauerhaftigkeit auch immer die Latenz erhöht.

Eine Sache, die Redis von anderen Caches wie memcached[2] abhebt, ist seine fest integrierte Unterstützung zur Speicherung von Werten auf der Festplatte. Standardmäßig werden die Schlüssel/Wert-Paare nur gelegentlich gesichert. Sie können den `LASTSAVE`-Befehl nutzen, um den *nix-Zeitstempel der letzten Schreiboperation abzufragen. Alternativ können Sie sich das `last_save_time`-Feld der `INFO`-Ausgabe ansehen.

Sie können Dauerhaftigkeit erzwingen, indem Sie den `SAVE`-Befehl ausführen (oder `BGSAVE`, das asynchron im Hintergrund sichert).

```
redis 127.0.0.1:6379> SAVE
```

Wenn Sie sich den Redis-Server-Log ansehen, finden Sie Zeilen wie diese:

```
[46421] 10 Oct 19:11:50 * Background saving started by pid 52123
[52123] 10 Oct 19:11:50 * DB saved on disk
[46421] 10 Oct 19:11:50 * Background saving terminated with success
```

Eine andere Methode für die Dauerhaftigkeit bieten die Snapshot-Einstellungen in der Konfigurationsdatei.

Snapshotting

Wir können das Tempo, in dem Daten auf Platte geschrieben werden, anpassen, indem wir ein save-Feld hinzufügen, löschen oder ändern. Standardmäßig gibt es drei save-Zeilen, die mit dem Schlüsselwort `save` beginnen, auf das eine Zeit in Sekunden und die minimale Anzahl veränderter Schlüssel folgt.

Soll also beispielsweise alle 5 Minuten (300 Sekunden) gesichert werden, wenn sich eine beliebige Anzahl von Schlüsseln geändert hat, dann können Sie Folgendes schreiben:

```
save 300 1
```

Die Standard-Konfiguration enthält einige gute Voreinstellungen. Diese drei Einstellungen legen fest, dass alle 60 Sekunden gesichert wird, wenn sich 10000 Schlüssel geändert haben, bei 10 geänderten Schlüsseln alle 300 Sekunden und bei allen anderen Änderungen alle 900 Sekunden (15 Minuten).

```
save 900 1
save 300 10
save 60 10000
```

2. http://www.memcached.org/

Sie können so viele (oder so wenige) save-Zeilen wie nötig angeben, um die Schwellwerte genau festzulegen.

Append-Only-Datei

Redis ist standardmäßig *schlussendlich dauerhaft*, d. h., es schreibt Werte in durch save-Einstellungen definierten Intervallen asynchron auf die Platte oder die Schreiboperationen werden durch von Clients initiierte Befehle erzwungen. Das ist für einen Second-Level-Caches oder Session-Server akzeptabel, reicht für Daten, die dauerhaft gespeichert werden müssen (z. B. Finanzdaten) aber nicht aus. Stürzt ein Redis-Server ab, werden die Benutzer nicht amüsiert sein, wenn sie Geld verloren haben.

Redis stellt eine „Append-Only-Datei“ (`appendonly.aof`) zur Verfügung, die eine Liste aller Schreiboperationen vorhält. Das ähnelt dem Write-Ahead-Logging in Kapitel 4, *HBase*, auf Seite 103. Stürzt der Server ab, bevor ein Wert gesichert wurde, führt er die Befehle beim Start aus und stellt seinen Zustand wieder her. `appendonly` muss dafür in `redis.conf` auf `yes` gesetzt werden.

```
appendonly yes
```

Dann müssen Sie entscheiden, wie oft ein Befehl an die Datei angehangen wird. Die Einstellung `always` ist dauerhafter, da jeder Befehl gesichert wird. Er ist aber auch langsam, was häufig dem zuwiderläuft, warum die Leute Redis nutzen. Standardmäßig ist `everysec` aktiviert, wodurch die Befehle gesammelt und nur einmal pro Sekunde geschrieben werden. Das ist ein vernünftiger Kompromiss, weil es schnell genug ist und im schlimmsten Fall nur die letzte Sekunde Ihrer Daten verloren geht. Schließlich gibt es noch die Option `no`, die das Flushing einfach dem Betriebssystem überlässt. Das kann recht selten passieren und häufig fahren Sie besser, wenn Sie die Append-Only-Datei ganz weg lassen.

```
# appendfsync always
appendfsync everysec
# appendfsync no
```

Für Append-Only gibt es noch detailliertere Parameter, die Sie sich in der Konfigurationsdatei genauer ansehen sollten, wenn Sie im Produktiveinsatz auf bestimmte Probleme reagieren müssen.

Sicherheit

Redis ist von Haus aus nicht als vollständig sicherer Server ausgelegt, auch wenn Sie in der Dokumentation über die `requirepass`-Einstellung und den

AUTH-Befehl stolpern. Diese können Sie getrost ignorieren, da sie nur Klartext-Passwörter bieten. Da ein Client etwa 100000 Passwörter pro Sekunde ausprobieren kann, ist das bestenfalls Makulatur, ganz abgesehen davon, dass Klartext-Passwörter grundsätzlich unsicher sind.

Interessanterweise bietet Redis auf Befehlsebene Sicherheit durch Verschleierung, indem es Ihnen ermöglicht, Befehle zu verstecken oder zu unterdrücken. Nachfolgend benennen wir den FLUSHALL-Befehl (der alle Schlüssel aus dem System löscht) in den schwer zu erratenden Befehl c283d93ac9528f986023793b411e4ba2 um:

```
rename-command FLUSHALL c283d93ac9528f986023793b411e4ba2
```

Wenn wir nun versuchen, FLUSHALL auszuführen, liefert der Server einen Fehler zurück. Der „geheime“ Befehl funktioniert hingegen.

```
redis 127.0.0.1:6379> FLUSHALL
(error) ERR unknown command 'FLUSHALL'
redis 127.0.0.1:6379> c283d93ac9528f986023793b411e4ba2
OK
```

Oder, noch besser, Sie deaktivieren den Befehl ganz, indem sie ihn auf einen Leerstring setzen.

```
rename-command FLUSHALL ""
```

Sie können eine beliebige Zahl von Befehlen auf diese Weise umbenennen oder löschen und so Ihre Befehlsumgebung an Ihre Bedürfnisse anpassen.

Parameter optimieren

Es gibt verschiedene fortgeschrittene Einstellungsmöglichkeiten, um die Geschwindigkeit langsamer Query-Logs zu erhöhen, Details der Kodierung festzulegen, Latenzen zu optimieren und externe Konfigurationsdateien zu importieren. Wenn Sie in der Dokumentation allerdings auf virtuellen Speicher stoßen, sollten Sie ihn unbedingt meiden. Seit Redis 2.4 gilt er als veraltet und könnte in zukünftigen Versionen ganz wegfallen.

Um Sie beim Test Ihrer Serverkonfiguration zu unterstützen, stellt Redis ein ausgezeichnetes Benchmarking-Tool zur Verfügung. Es stellt standardmäßig die Verbindung mit dem lokalen Port 6379 her und sendet 10000 Requests über 50 parallel laufende Clients. Über das Argument -n können wir 100000 Requests durchführen.

```
$ redis-benchmark -n 100000
====== PING (inline) ======
 100000 requests completed in 3.05 seconds
 50 parallel clients
 3 bytes payload
 keep alive: 1
5.03% <= 1 milliseconds
98.44% <= 2 milliseconds
99.92% <= 3 milliseconds
100.00% <= 3 milliseconds
32808.40 requests per second
...
```

Andere Befehle wie `SADD` und `LRANGE` werden ebenfalls getestet, wobei die komplexeren generell mehr Zeit beanspruchen.

Master/Slave-Replikation

Genau wie andere NoSQL-Datenbanken, die wir betrachtet haben (etwa MongoDB und Neo4j), unterstützt auch Redis die Master/Slave-Replikation. Ein Server ist standardmäßig der Master, wenn Sie ihn nicht als Slave eines anderen Servers festlegen. Die Daten werden auf eine beliebige Zahl von Slave-Servern repliziert.

Slave-Server einzurichten, ist einfach. Zuerst brauchen wir eine Kopie unserer `redis.conf`-Datei.

```
$ cp redis.conf redis-s1.conf
```

Die Datei bleibt bis auf die folgenden Änderungen gleich:

```
port 6380
slaveof 127.0.0.1 6379
```

Wenn alles nach Plan läuft, sollten Sie beim Start des Slave-Servers in etwa Folgendes im Log sehen:

```
$ redis-server redis-s1.conf
```

```
[9003] 16 Oct 23:51:52 * Connecting to MASTER...
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync started
[9003] 16 Oct 23:51:52 * Non blocking connect for SYNC fired the event.
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: receiving 28 bytes from master
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Loading DB in memory
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Finished with success
```

Und im Master-Log sollte der String `1 slaves` auftauchen.

```
redis 127.0.0.1:6379> SADD meetings "StarTrek Pastry Chefs" "LARPers Intl."
```

Wenn wir über die Kommandozeile die Verbindung mit dem Slave herstellen, sollten wir unsere Meeting-Liste vorfinden.

```
redis 127.0.0.1:6380> SMEMBERS meetings
1) "StarTrek Pastry Chefs"
2) "LARPers Intl."
```

Im Produktionseinsatz werden Sie die Replikation generell aus Verfügbarkeits- oder Backup-Gründen verwenden und daher Redis-Slaves auf verschiedenen Rechnern nutzen.

Datendump

Bisher haben wir viel darüber gesprochen, wie schnell Redis ist, doch man kann dafür nur schlecht ein Gefühl entwickeln, ohne mit etwas mehr Daten herumzuspielen.

Lassen Sie uns den Redis-Server mit einer großen Datenmenge befüllen. Wenn Sie wollen, können Sie den Slave laufen lassen, doch ein Laptop oder Desktop läuft mit einem einzelnen Master-Server möglicherweise schneller. Wir wollen uns eine Liste mit über 2,5 Millionen veröffentlichten Büchern von Freebase.com herunterladen. Den Schlüssel bildet dabei die International Standard Book Number (ISBN).[3]

Zuerst benötigen wir das `redis` Ruby-Gem.

```
$ gem install redis
```

Es gibt verschiedene Möglichkeiten, eine große Datenmenge einzufügen, die zunehmend schneller, aber auch komplexer sind.

Die einfachste Möglichkeit besteht darin, die Liste der Daten durchzugehen und über den Standard `redis-rbSET`-Client `SET` für jeden Wert aufzurufen.

redis/isbn.rb

```
LIMIT = 1.0 / 0  # 1.0/0 is Infinity in Ruby
# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
%w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall
count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
  isbn, _, _, title = line.split("\t")
  next if isbn.empty? || title == "\n"
```

3. http://download.freebase.com/datadumps/latest/browse/book/isbn.tsv

```
  $redis.set(isbn, title.strip)

  # Setzen Sie ein LIMIT, wenn nicht alle Daten eingefügt werden sollen
  break if count >= LIMIT
end
puts "#{count} items in #{Time.now - start} seconds"

$ ruby isbn.rb isbn.tsv
2456384 items in 266.690189 seconds
```

Wenn Sie das Einfügen beschleunigen wollen und nicht mit JRuby arbeiten, können Sie optional das `hiredis`-Gem installieren. Dieser C-Treiber ist wesentlich schneller als der native Ruby-Treiber. Entfernen Sie den Kommentar in der `hiredis require`-Zeile, damit der Treiber geladen wird. Bei dieser CPU-gebundenen Operation werden Sie vielleicht keine große Verbesserung sehen, doch wir empfehlen `hiredis` wärmstens für den produktiven Ruby-Einsatz.

Eine große Verbesserung werden Sie beim Pipelining sehen. Nachfolgend verarbeiten wir jeweils 1000 Zeilen und fügen sie dann über eine Pipeline ein. Das hat unsere Einfügezeit um über 300 Prozent reduziert.

redis/isbn_pipelined.rb

```
BATCH_SIZE = 1000
LIMIT = 1.0 / 0  # 1.0/0 ist bei Ruby Unendlich

# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
%w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall

# Daten in einem einzigen Batch-Update einfügen
def flush(batch)
  $redis.pipelined do
    batch.each do |saved_line|
      isbn, _, _, title = line.split("\t")
      next if isbn.empty? || title == "\n"
      $redis.set(isbn, title.strip)
    end
  end
  batch.clear
end

batch = []
count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1

  # Zeilen in Array schieben
  batch << line

  # Schreiben, wenn Array auf BATCH_SIZE anwächst
```

```
  if batch.size == BATCH_SIZE
    flush(batch)
    puts "#{count-1} items"
  end

  # Setzen Sie ein LIMIT, wenn nicht alle Daten eingefügt werden sollen
  break if count >= LIMIT
end
# Verbliebene Werte schreiben
flush(batch)

puts "#{count-1} items in #{Time.now - start} seconds"

$ ruby isbn_pipelined.rb isbn.tsv
2666642 items in 79.312975 seconds
```

Das reduziert die Anzahl der notwendigen Redis-Verbindungen, doch der Aufbau der Pipeline ist selbst mit einem Overhead verbunden. Im Produktiveinsatz sollten Sie mit verschiedenen Batch-Größen experimentieren.

Ein kurzer Hinweis für Ruby-Nutzer: Wenn Ihre Anwendung nicht-blockierend über Event Machine läuft, kann der Ruby-Treiber em-synchrony über `EM::Protocols::Redis.connect` verwenden.

Redis-Cluster

Neben der einfachen Replikation bieten viele Redis-Clients ein Interface zum Aufbau einfacher, verteilter Ad-hoc-Cluster an. Der Ruby-Client redis-rb unterstützt einen über eine konsistente Hash-Funktion verwalteten Cluster. Sie kennen die konsistente Hash-Funktion aus dem Riak-Kapitel, wo Knoten eingefügt und entfernt werden konnten, ohne die meisten Schlüssel verfallen lassen zu müssen. Die Idee ist hier die gleiche, wird aber durch einen Client verwaltet und nicht durch die Server selbst.

Zuerst benötigen wir einen weiteren Server. Im Gegensatz zu unserem Master/Slave-Setup verwenden unsere beiden Server die Master(Standard)-Konfiguration. Wir haben die `redis.conf` kopiert und den Port auf 6380 gesetzt. Mehr ist für die Server nicht nötig.

redis/isbn_cluster.rb

```
LIMIT = 10000
%w{rubygems time redis}.each{|r| require r}
require 'redis/distributed'

$redis = Redis::Distributed.new([
  "redis://localhost:6379/", "redis://localhost:6380/"
])
$redis.flushall

count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
```

```
  count += 1
  next if count == 1
  isbn, _, _, title = line.split("\t")
  next if isbn.empty? || title == "\n"

  $redis.set(isbn, title.strip)

  # Setzen Sie ein LIMIT, wenn nicht alle Daten eingefügt werden sollen
  break if count >= LIMIT
end
puts "#{count} items in #{Time.now - start} seconds"
```

Das Bridging zwischen zwei oder mehr Servern verlangt nur kleinere Änderungen an unserem ISBN-Client. Zuerst müssen wir die Datei `redis/distributed` aus dem Redis-Gem einbinden.

```
require 'redis/distributed'
```

Dann ersetzen wir den Redis-Client durch Redis::Distributed und übergeben ihm ein Array mit Server-URIs. Jeder URI besteht aus dem `redis`-Schema, dem Server (localhost) und dem Port.

```
$redis = Redis::Distributed.new([
  "redis://localhost:6379/",
  "redis://localhost:6380/"
])
```

Die Ausführung des Clients erfolgt wie vorher.

```
$ ruby isbn_cluster.rb isbn.tsv
```

Doch der Client übernimmt nun wesentlich mehr Arbeit, da er die Berechnung übernimmt, welche Schlüssel auf welchen Servern gespeichert werden. Sie können überprüfen, ob die Schlüssel tatsächlich auf separaten Servern gespeichert werden, indem Sie den gleichen ISBN-Schlüssel von jedem Server über die Kommandozeile abrufen. Nur ein Client liefert den Wert mit `GET` zurück. Doch solange Sie Schlüssel abrufen, die über die gleiche Redis::Distributed-Konfiguration gesetzt wurden, ruft der Client die Werte von den richtigen Servern ab.

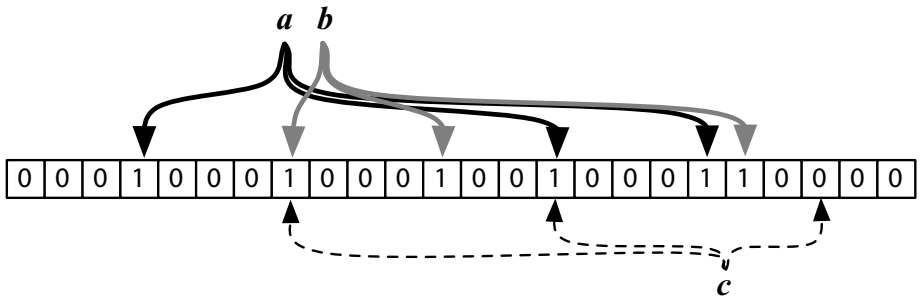
Bloomfilter

Einen eindeutigen Begriff zu verwenden, ist eine ausgezeichnete Strategie, wenn etwas online gefunden werden soll. Wenn Sie ein Buch mit dem Titel *The Jabbyredis* schreiben, können Sie ziemlich sicher sein, dass jede Suchmaschine auf sie verlinkt. Wir wollen ein Skript entwickeln, mit dem wir schnell überprüfen können, ob ein Wort in den Titeln unseres ISBN-Kata-

logs auftaucht. Wir können einen Bloomfilter nutzen, um herauszufinden, ob ein Wort schon verwendet wird.

Ein Bloomfilter ist eine probabilistische Datenstruktur, mit der Sie prüfen können, ob ein Element in einer Menge existiert. Wir haben ihn bereits in 4.3, *Bloomfilter*, auf Seite 121 betrachtet. Zwar kann er falsche Positive zurückliefern, aber keine falschen Negative. Das ist nützlich, wenn Sie schnell herausfinden wollen, ob ein Wert in einem System vorhanden ist oder nicht.

Bloomfilter bestimmen die „Nichtexistenz“ eines Wertes, indem Sie einen Wert in eine dünnbesetzte Folge von Bits umwandeln und mit der Vereinigungsmenge der Bits aller Werte vergleichen. Mit anderen Worten: Es wird ein neuer Wert über ein ODER mit der aktuellen Bloomfilter-Bitsequenz verknüpft. Wenn Sie wissen wollen, ob ein Wert bereits im System vorliegt, führen Sie ein UND über die Bloomfilter-Sequenz aus. Besitzt der Wert irgendwelche Wahr-Bits, die an den entsprechenden Stellen des Bloomfilters nicht wahr sind, dann gibt es den Wert nicht. Nachfolgend eine graphische Darstellung des Konzepts.



Wir wollen ein Programm schreiben, das eine Reihe von ISBN-Buchdaten durchgeht, die Titel der Bücher extrahiert, vereinfacht und in einzelne Wörter aufteilt. Jedes neue Wort läuft dann durch den Bloomfilter. Gibt der Bloomfilter falsch zurück (d. h., das Wort kommt in unserem Bloomfilter noch nicht vor), fügen wir es hinzu. Um das Ganze verfolgen zu können, geben wir jedes neu hinzugefügte Wort aus.

```
$ gem install bloomfilter-rb
```

redis/isbn_bf.rb

```
# LIMIT = 1.0 / 0 # 1.0/0 ist bei Ruby unendlich
LIMIT= 10000
%w{rubygems time bloomfilter-rb}.each{|r| require r}
bloomfilter = BloomFilter::Redis.new(:size => 1000000)

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall
```

```
count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
  _, _, _, title = line.split("\t")
  next if title == "\n"

  words = title.gsub(/[^\w\s]+/, '').downcase
  # Wörter verarbeiten
  words = words.split(' ')
  words.each do |word|
    # Dem Bloomfilter bekannte Wörter überspringen
    next if bloomfilter.include?(word)
    # Neue Wörter ausgeben
    puts word
    # und in Bloomfilter einfügen
    bloomfilter.insert(word)
  end
  # Setzen Sie ein LIMIT, wenn nicht alle Daten eingefügt werden sollen
  break if count >= LIMIT
end
puts "Contains Jabbyredis? #{bloomfilter.include?('jabbyredis')}"
puts "#{count} lines in #{Time.now - start} seconds"
```

Ruby-Wunderkind Ilya Grigorik hat diesen Redis-gestützten Bloomfilter entwickelt, doch das Konzept ist auf jede Sprache übertragbar.

Der Client verwendet die gleiche ISBN-Datei, benötigt aber nur die Buchtitel.

```
$ ruby isbn_bf.rb isbn.tsv
```

Zu Beginn der Ausgabe sollte Sie viele gängige englische Wörter wie *and* und *the* sehen. Gegen Ende der Daten werden die Wörter immer esoterischer, wie *unindustria*.

Der Vorteil dieses Ansatzes besteht darin, doppelte Wörter erkennen zu können. Der Nachteil ist, dass einige falsche Positive durch den Rost fallen – der Bloomfilter könnte ein Wort kennzeichnen, das er noch nicht gesehen hat. Aus diesem Grund würde in einem realen Anwendungsfall eine zweite Prüfung vorgenommen werden, etwa eine langsamere Query über eine entsprechende Datenbank. Das sollte aber nur für einen kleinen Prozentsatz notwendig sein, wenn der Filter groß genug ist (was man berechnen kann).[4]

SETBIT und GETBIT

Wie bereits erwähnt, arbeiten Bloomfilter mit der Umkehr bestimmter Bits in einem dünnbesetzten Binärfeld. Die von uns verwendete Redis Bloomfilter-Implementierung nutzt zwei relativ neue Redis-Befehle, die genau das machen: `SETBIT` und `GETBIT`.

4. http://en.wikipedia.org/wiki/Bloom_filter

Wie alle Redis-Befehle ist `SETBIT` mehr oder weniger selbstbeschreibend. Der Befehl setzt ein einzelnes Bit (auf 0 oder 1) an einer bestimmten Stelle einer Bitfolge, die bei 0 beginnt. Häufiger Anwendungsfall ist das mehrdimensionale Flagging – ein paar Bits lassen sich schneller setzen, als man eine Folge beschreibender Strings schreiben kann.

Wenn wir die Garnierung für einen Hamburger nachhalten wollen, können wir jedem Typ eine Bitposition zuweisen, etwa Ketchup = 0, Senf = 1, Zwiebeln = 2, Salat = 3. Ein Hamburger mit Senf und Zwiebeln lässt sich dann mit 0110 darstellen und in der Kommandozeile wie folgt setzen:

```
redis 127.0.0.1:6379> SETBIT my_burger 1 1
  (integer) 0
redis 127.0.0.1:6379> SETBIT my_burger 2 1
  (integer) 00
```

Später kann ein Prozess überprüfen, ob der Hamburger mit Salat oder Senf sein soll. Wird eine 0 zurückgegeben, lautet die Antwort nein, bei einer eins ja.

```
redis 127.0.0.1:6379> GETBIT my_burger 3
  (integer) 0
redis 127.0.0.1:6379> GETBIT my_burger 1
  (integer) 1
```

Die Bloomfilter-Implementierung nutzt dieses Verhalten zu ihrem Vorteil, indem sie einen Wert als Multibit-Wert hasht. Sie ruft `SETBIT X 1` für jede gesetzte Position in einem `insert` auf (X ist dabei die Bitposition) und prüft die Existenz durch den Aufruf von `GETBIT X` bei `include?` – und gibt falsch zurück, wenn eine `GETBIT`-Position 0 zurückgibt.

Bloomfilter sind ausgezeichnet, wenn man unnötigen Traffic zu einem langsameren, tiefer liegenden System vermeiden will, sei es eine langsamere Datenbank, eine begrenzte Ressource oder ein Netzwerk-Request. Wenn Sie mit einer langsamen Datenbank von IP-Adressen arbeiten und alle neuen Besucher Ihrer Site festhalten wollen, können Sie einen Bloomfilter nutzen, um herauszufinden, ob die IP-Adresse in Ihrem System existiert. Liefert der Bloomfilter ein Nein zurück, wissen Sie, dass die IP-Adresse hinzugefügt werden muss und können entsprechend reagieren. Gibt der Bloomfilter ein Ja zurück, kann die IP-Adresse im Backend vorhanden sein oder auch nicht, und Sie brauchen einen zweiten Lookup, um ganz sicher zu gehen. Darum ist die Berechnung der richtigen Größe wichtig. Gut dimensionierte Bloomfilter können die Fehlerrate und die Wahrscheinlichkeit falscher Positive reduzieren (wenn auch nicht eliminieren).

Was wir am zweiten Tag gelernt haben

Heute haben wir unsere Untersuchung von Redis über einfache Operationen hinaus ausgedehnt und noch das letzte Quäntchen Geschwindigkeit aus einem sehr schnellen System herausgepresst. Wie wir am ersten Tag gesehen haben, bietet Redis die schnelle und flexible Speicherung und Verarbeitung von Datenstrukturen, ist aber mit eingebauten Publish/Subscribe-Funktionen und Bitoperationen auch für komplexere Aufgaben gerüstet. Es ist hochgradig konfigurierbar und kennt viele Einstellungen, mit denen sich Ihre Anforderungen an Dauerhaftigkeit und Replikation erfüllen lassen. Es unterstützt auch Erweiterungen von Drittanbietern, wie etwa Bloomfilter und Clustering.

Damit schließen wir auch die Betrachtung der wichtigsten Operationen des Redis-Datenstruktur-Speichers ab. Morgen werden wir die Dinge etwas anders angehen, indem wir Redis als Grundpfeiler eines polyglotten Persistenz-Setups mit CouchDB und Neo4j nutzen.

Tag 2: Selbststudium

Finden Sie heraus

Finden Sie heraus, was Messaging-Muster sind und wie viele solcher Muster Redis implementieren kann.

Machen Sie Folgendes

1. Führen Sie das Skript zum Befüllen der ISBN-Datenbank aus und deaktivieren Sie dabei Snapshotting und Append-Only-Dateien. Setzen Sie dann `appendfsync` auf `always`, führen Sie es erneut aus und vergleichen Sie den Geschwindigkeitsunterschied.
2. Nutzen Sie ein Web-Framework Ihrer Lieblings-Programmiersprache und versuchen Sie, einen einfachen Redis-gestützten URL-Shortening-Service mit einem Eingabefeld für die URL und einem einfachen Redirect basierend auf der URL zu bauen. Sichern Sie das Ganze über ein Master/Slave-repliziertes Cluster mit mehreren Knoten als Backend ab.

8.4 Tag 3: Zusammenspiel mit anderen Datenbanken

Heute wollen wir unser letztes Datenbank-Kapitel abschließen, indem wir einige der bisher vorgestellten Datenbanken zum Spielen einladen. Redis wird dabei die Hauptrolle übernehmen, da es die Zusammenarbeit mit anderen Datenbanken schneller macht und vereinfacht.

Wir haben im Verlauf des Buches immer wieder gesehen, dass unterschiedliche Datenbanken verschiedene Stärken haben. Moderne System-Designs bewegen sich daher immer mehr in Richtung polyglotter Persistenzmodelle, in denen verschiedene Datenbanken eine bestimmte Rolle innerhalb des Systems übernehmen. Sie werden lernen, wie man ein solches Projekt aufbaut, wobei wir CouchDB als kanonische Datenquelle verwenden, Neo4j zur Verarbeitung der Datenbeziehungen und Redis zur Befüllung mit Daten und als Cache. Betrachten Sie das als Ihre Abschlussprüfung.

Beachten Sie, dass dieses Projekt nicht mit den Vorlieben der Autoren für eine bestimmte Kombination von Datenbanken, Sprachen oder Frameworks zusammenhängt. Es soll nur beispielhaft aufzeigen, wie mehrere Datenbanken zusammenarbeiten und ihre Fähigkeiten zur Erreichung eines übergeordneten Zieles einbringen können.

Ein polyglotter, persistenter Service

Unser polyglotter Persistenz-Service fungiert als Frontend zu einem Band-Informationssystem. Wir wollen eine Liste mit Bandnamen speichern, den Mitgliedern, die in diesen Bands mitwirken und eine beliebige Zahl von Rollen, die jedes Mitglied in dieser Band übernimmt, vom Leadsänger bis zum Keytar-Spieler. Jede der drei Datenbanken – Redis, CouchDB und Neo4j – übernimmt einen anderen Aspekt unseres Band-Managementsystems.

Redis übernimmt in unserem System drei wichtige Rollen: Es unterstützt CouchDB beim Befüllen mit Daten, indem es als Cache für die jüngsten Neo4j-Änderungen fungiert sowie als schneller Lookup für die partielle Suche. Seine Geschwindigkeit und seine Fähigkeit, unterschiedliche Datenformate zu speichern, machen es zu einem guten Kandidaten für die Befüllung und die fest eingebauten Verfallszeit-Regeln eignen sich bestens für die Verarbeitung von Cache-Daten.

CouchDB ist unsere maßgebliche (autoritative) Datenquelle (System of Record, SOR). Die Dokumentenstruktur von CouchDB bietet eine einfache Möglichkeit, Bandinformationen mit verschachtelten Künstler- und Rolleninformationen zu speichern, und wir nutzen die Changes-API von CouchDB, um unsere dritte Datenquelle auf dem neuesten Stand zu halten.

Neo4j ist unser Beziehungsspeicher. Obwohl die direkte Abfrage der CouchDB-Datenquelle völlig in Ordnung wäre, erlaubt uns die Graph-Datenbank eine Einfachheit und Geschwindigkeit bei der Traversierung von Knoten-Beziehungen, die andere Datenbanken nur schwerlich erreichen können. Wir speichern Beziehungen zwischen Bands, Bandmitgliedern und den Rollen, die die Mitglieder spielen.

Das Aufkommen polyglotter Persistenz

Genau wie das Phänomen der polyglotten Programmierung gewinnt auch die polyglotte Persistenz zunehmend an Boden.

Bei der polyglotten Programmierung (falls Sie mit dem Thema nicht vertraut sind) verwendet ein Team mehr als eine Programmiersprache in einem Projekt. Vergleichen Sie das mit der Konvention, eine einzige Allzweck-Programmiersprache für ein Projekt zu nutzen. Das kann aufgrund der unterschiedlichen Stärken der Sprachen nützlich sein. Ein Framework wie Scala kann für serverseitige, zustandsfreie Transaktionen im Web gut geeignet sein, während sich eine Sprache wie Ruby für die Businesslogik besser eignet. Zusammen erzeugen sie eine Synergie. Ein berühmtes polyglottes Sprachsystem wie dieses wurde bei Twitter verwendet.

Einige der vorgestellten Datenbanken unterstützen selbst die polyglotte Programmierung – Riak unterstützt bei der Mapreduce-Entwicklung sowohl JavaScript als auch Erlang und ein einzelner Request kann beide ausführen.

Ähnlich wie der sprachorientierte Cousin nutzt die polyglotte Persistenz die Stärken verschiedener Datenbanken in einem System. Das ist das Gegenteil von der heute üblichen Praxis, eine einzige Datenbank (üblicherweise eine relationale) zu nutzen. Eine einfache Variante ist aber schon heute recht weit verbreitet: die Verwendung eines Schlüssel/Wert-Speichers (wie Redis) als Cache für die vergleichsweise langsamen Queries relationaler Datenbanken (wie PostgreSQL). Wie wir in den vergangenen Kapiteln gesehen haben, ist das relationale Modell für eine wachsende Zahl von Problemen (z. B. der Graphentraversierung) ungeeignet. Doch selbst die neuen Datenbanken können nur als kleine Sterne am Himmel der ganzen Anforderungen scheinen.

Warum dieses plötzliche Interesse am Polyglotten? Martin Fowler bemerkte,[a] das die Verwendung einer einzelnen zentralen Datenbank, in die verschiedene Anwendungen integriert werden können, ein übliches Muster des Software-Designs war. Dieses einstmals beliebte Muster wurde zugunsten einer Middleware-Schicht aufgegeben, bei der mehrere Anwendungen über HTTP mit einer Service-Schicht kommunizieren. Das gibt dem Middleware-Service selbst die Möglichkeit, mit einer beliebigen Zahl von Datenbanken zu arbeiten bzw. im Falle polyglotter Persistenz mit einer beliebigen Zahl von Datenbank-Typen.

a. http://martinfowler.com/bliki/DatabaseThaw.html

Jede Datenbank übernimmt in unserem System eine bestimmte Rolle, sie kommunizieren aber nicht direkt miteinander. Wir verwenden das JavaScript-Framework Node.js, um die Datenbanken zu befüllen und zwischen ihnen zu kommunizieren sowie als einfachen Frontend-Server. Da das Verbinden mehrerer Datenbanken etwas Code verlangt, werden Sie an diesem Tag wesentlich mehr Programme sehen, als bisher.

Befüllen

Unsere erste Aufgabe besteht darin, die Datenspeicher mit den notwendigen Daten zu befüllen. Wir verwenden hier einen zweistufigen Ansatz, d. h., wir befüllen zuerst Redis und dann unsere CouchDB-SOR.

Wie früher laden wir eine Datei von Freebase.com herunter. Wir werden die tabulatorgetrennte `group_membership` verwenden.[5] Diese Datei enthält sehr viele Informationen, doch wir sind nur daran interessiert, den `member`(Künstler)-Namen, den `group`(Band)-Namen sowie die jeweilige `role` (Rolle) zu extrahieren. Diese Rollen liegen in Form einer kommaseparierten Liste vor. *John Cooper* war in der Band *Skillet* z. B. *Leadsänger* und spielte außerdem *akustische Gitarre* und *Bass*.

```
/m/0654bxy  John Cooper Skillet Lead vocalist,Acoustic guitar,Bass  1996
```

Letztlich wollen wir John Cooper und die anderen Mitglieder von Skillet in einem einzelnen CouchDB-Dokument in der folgenden Form ablegen und unter dem URL `http://localhost:5984/bands/Skillet` speichern:

```
{
  "_id": "Skillet",
  "name": "Skillet"
  "artists": [
    {
      "name": "John Cooper",
      "role": [
        "Acoustic guitar",
        "Lead vocalist",
        "Bass"
      ]
    },
    ...
    {
      "name": "Korey Cooper",
      "role": [
        "backing vocals",
        "Synthesizer",
        "Guitar",
        "Keyboard instrument"
      ]
    }
  ]
}
```

Die Datei enthält Daten über mehr als 100000 Bandmitglieder und mehr als 30000 Bands. Das ist nicht besonders viel, für den Aufbau eines eigenen Systems aber ein guter Anfang. Beachten Sie, dass nicht jede Künstler-Rolle

5. http://download.freebase.com/datadumps/latest/browse/music/group_membership.tsv

definiert ist. Die Daten sind also unvollständig, aber darum können wir uns später kümmern.

Phase 1: Datentransformation

Sie wundern sich vielleicht, warum wir uns die Mühe machen, zuerst Redis zu befüllen, statt die Daten direkt in CouchDB einzupflegen. Da es als Vermittler arbeitet, bringt Redis Struktur in die TSV-Rohdaten, was das nachfolgende Einfügen in andere Datenbanken sehr schnell macht. Da unser Plan darin besteht, einen einzelnen Datensatz pro Band anzulegen, erlaubt Redis es uns, die TSV-Datei in einem Durchlauf zu verarbeiten (der Bandname taucht für jedes Mitglied auf und jedes Bandmitglied wird in einer Zeile beschrieben). Jedes einzelne Mitglied direkt in CouchDB einzufügen, kann zu einem Update-Stau führen, wenn für zwei Bandmitglieder zur gleichen Zeit ein Band-Dokument angelegt/aktualisiert wird, weil das System dann einen Datensatz erneut einfügen muss, wenn CouchDBs Versionsprüfung fehlschlägt.

Der Haken an dieser Strategie ist die Beschränkung von Redis auf das Vorhalten des gesamten Datenbestands im RAM – auch wenn wir dieses Limit mit Hilfe des Clusters von Tag 2 lösen könnten.

Neben den Rohdaten benötigen wir ein installiertes Node.js sowie den Node Package Manager (npm). Sobald das erledigt ist, müssen wir drei NPM-Projekte installieren: redis, csv und hiredis (der optionale Redis C-Treiber von gestern, der Redis-Interaktionen stark beschleunigen kann).

```
$ npm install hiredis redis csv
```

Stellen Sie nun sicher, dass der Redis-Server den Standardport 6379 nutzt, oder ändern Sie die `createClient()`-Funktion aller Skripten entsprechend.

Sie können Redis befüllen, indem Sie das folgende Node.js-Skript im gleichen Verzeichnis ausführen, in dem auch Ihre TSV-Datei liegt (deren Namen wir hier mit `group_membership.tsv` annehmen). Alle von uns betrachteten JavaScript-Dateien sind recht umfangreich, weshalb wir sie hier nicht ganz abdrucken. Sie können den gesamten Code über die Pragmatic Website herunterladen. Hier gehen wir nur auf die wichtigsten Teile jeder Datei ein. Laden Sie die folgende Datei herunter und führen Sie sie aus:

```
$ node pre_populate.js
```

Das Skript geht jede Zeile der TSV-Datei durch und extrahiert die Namen der Künstler, der Bands und die Rollen, die die Künstler in der Band übernehmen. Dann fügt es diese Werte in Redis ein (und überspringt Leerzeilen).

Das Format jedes Redis-Bandschlüssels ist `"band:Bandname"`. Das Skript fügt den Namen des Künstlers in ein Set mit Künstlernamen ein. Der Schlüssel `"band:Beatles"` enthält also ein Set mit den Werten `["John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"]`. Die Künstler-Schlüssel enthalten ebenfalls den Bandnamen und ein Set mit den Rollen. `"artist:Beatles:Ringo Starr"` enthält das Set `["Drums"]`.

Der restliche Code hält nur nach, wie viele Zeilen verarbeitet wurden, und gibt die Ergebnisse auf dem Bildschirm aus.

redis/pre_populate.js

```
csv().
fromPath( tsvFileName, { delimiter: '\t', quote: '' }).
on('data', function(data, index) {
  var
    artist = data[2],
    band = data[3],
    roles = buildRoles(data[4]);
  if( band === '' || artist === '' ) {
    trackLineCount();
    return true;
  }
  redis_client.sadd('band:' + band, artist);
  roles.forEach(function(role) {
    redis_client.sadd('artist:' + band + ':' + artist, role);
  });
  trackLineCount();
}).
```

Sie können prüfen, ob der Code Redis wirklich befüllt, indem Sie `redis-cli` starten und `RANDOMKEY` ausführen. Das sollte einen Schlüssel ausgeben, dem `band:` oder `artist:` voransteht. Jeder Wert außer `(nil)` ist gut.

Nachdem Redis nun befüllt ist, machen wir gleich mit dem nächsten Abschnitt weiter. Beim Ausschalten von Redis könnten Daten verloren gehen, wenn Sie keine höhere Dauerhaftigkeit eingestellt oder einen `SAVE`-Befehl angestoßen haben.

Phase 2: Einfügen in CouchDB

CouchDB spielt die Rolle des Haupt-Datenspeichers, des sog. System of Record (SOR). Treten irgendwelche Datenkonflikte zwischen Redis, CouchDB oder Neo4j auf, gewinnt CouchDB. Eine gutes SOR muss alle Daten enthalten, die notwendig sind, um die anderen Datenquellen in seiner Domäne wieder aufbauen zu können.

Stellen Sie sicher, dass CouchDB an seinem Standardport 5984 läuft, oder ändern Sie die Zeile `require('http').createClient(5984, 'localhost')`im

Eric sagt:
Nichtblockierender Code

Bevor wir mit diesem Buch begonnen haben, waren wir mit ereignisgesteuerten, nichtsperrenden Anwendungen nur flüchtig vertraut. *Nichtblockierend* bedeutet genau das: Statt darauf zu warten, dass ein lange laufender Prozess zum Ende kommt, wird der Hauptprozess weiter ausgeführt. Was auch immer als Reaktion auf ein blockierendes Ereignis nötig ist, packen Sie in eine Funktion oder einen Codeblock, der später ausgeführt wird. Das kann das Starten eines separaten Threads bedeuten oder (wie in unserem Fall) die Implementierung eines ereignisgesteuerten Reactor-Musters.

Bei einem blockierende Programm können Sie Code entwickeln, der eine Datenbank abfragt, wartet und dann die Ergebnisse verarbeitet.

```
results = database.some_query()
for value in results
  # Mach etwas mit jedem Wert
end
# Dieser Teil wird erst ausgeführt, wenn alle Ergebnisse verarbeitet wurden...
```

Bei einem ereignisgesteuerten Programm übergeben Sie diese Schleife als Funktion/Codeblock. Während die Datenbank ihr Ding macht, läuft der Rest des Programms weiter. Erst wenn die Datenbank die Ergebnisse zurückgibt, wird die Funktion/der Codeblock ausgeführt.

```
database.some_query do |results|
  for value in results
    # Mach etwas mit jedem Wert
  end
end
# Dieser Teil wird schon ausgeführt, während die Datenbankabfrage läuft...
```

Es dauert seine Zeit, bis man die Vorteile erkennt. Ja, der Rest des Programms kann weiterlaufen, statt untätig auf die Datenbank warten zu müssen, aber ist das üblich? Anscheinend ja, denn als wir begannen, in diesem Stil zu entwickeln, bemerkten wir eine Reduzierung der Latenz um eine ganze Größenordnung.

Wir versuchen, den Code so einfach wie möglich zu halten, doch die nichtblockierende Arbeit mit einer Datenbank ist von Natur aus ein komplexer Prozess. Doch wir haben gelernt, dass es beim Umgang mit Datenbanken generell eine sehr gute Methode ist. Nahezu jede populäre Programmiersprache besitzt irgendeine Art nichtblockierender Bibliothek. Ruby hat EventMachine, Python hat Twisted, Java die NIO-Bibliothek, C# hat Interlace und JavaScript hat natürlich Node.js.

folgenden Code entsprechend. Redis muss nach wie vor laufen. Laden Sie die folgende Datei herunter und führen Sie sie aus:

```
$ node populate_couch.js
```

In der ersten Phase ging es darum, die Daten aus einer TSV-Datei zu ziehen und Redis zu befüllen. In der zweiten Phase geht es nun darum, die Daten aus Redis abzurufen und CouchDB zu befüllen. Wir verwenden für CouchDB keine speziellen Treibern, da es ein einfaches REST-Interface nutzt und Node.js eine einfache HTTP-Bibliothek besitzt.

Im folgenden Code führen wir den Redis-Befehl `KEYS bands:*` aus, um eine Liste aller Bandnamen in unserem System zu erhalten. Bei einer *wirklich* großen Datenmenge könnten wir noch etwas genauer werden (z. B. würde `bands:A*` nur die mit *a* beginnenden Bandnamen zurückliefern, und so weiter). Dann rufen wir für jede Band die Mitglieder ab und extrahieren den Bandnamen aus dem Schlüssel, indem wir den Präfix *bands:* aus dem Schlüsselstring entfernen.

redis/populate_couch.js

```
redisClient.keys('band:*', function(error, bandKeys) {
  totalBands = bandKeys.length;
  var
    readBands = 0,
    bandsBatch = [];

  bandKeys.forEach(function(bandKey) {
    // Substring von 'band:'.length liefert uns den Bandnamen
    var bandName = bandKey.substring(5);
    redisClient.smembers(bandKey, function(error, artists) {
```

Als Nächstes rufen wir die Rollen jedes Künstlers in der Band ab, die Redis als Array von Arrays (jede Rolle hat ihr eigenes Array) zurückgibt. Wir machen das, indem wir die Redis-Befehle `SMEMBERS` zu einem Array namens `roleBatch` zusammenfassen und dann in einem einzigen `MULTI`-Batch verarbeiten. Effektiv wird damit ein einzelner Pipeline-Request ausgeführt:

```
MULTI
 SMEMBERS "artist:Beatles:John Lennon"
 SMEMBERS "artist:Beatles:Ringo Starr"
EXEC
```

Nun packen wir 50 CouchDB-Dokumente zusammen. Wir verwenden einen Stapel von 50 Dokumenten, weil wir den gesamten Satz dann an CouchDBs `/_bulk_docs`-Befehl senden, was ein sehr, sehr schnelles Einfügen ermöglicht.

redis/populate_couch.js

```
redisClient.
  multi(roleBatch).
  exec(function(err, roles)
  {
    var
      i = 0,
      artistDocs = [];

    // Subdokumente für die Künstler aufbauen
    artists.forEach( function(artistName) {
      artistDocs.push({ name: artistName, role : roles[i++] });
    });

    // Neues Band-Dokument auf den Stapel schieben
    bandsBatch.push({
      _id: couchKeyify( bandName ),
      name: bandName,
      artists: artistDocs
    });
```

Mit der Befüllung der Band-Datenbank liegen nun alle von uns benötigten Daten an einem einzigen Ort vor. Wir kennen die Namen vieler Bands, die Künstler, die in diesen Bands gespielt haben und wir kennen deren Rollen innerhalb der Bands.

Jetzt wäre ein guter Zeitpunkt, um eine kleine Pause zu machen, und mit unserem gerade befüllten Band-SOR in CouchDB unter `http://localhost:5984/_utils/database.html?bands` herumzuspielen.

Beziehungsspeicher

Als Nächstes steht unser Neo4j-Service auf dem Plan, mit der wir die Beziehung zwischen den Künstlern und ihren Rollen nachhalten wollen. Natürlich könnten wir CouchDB auch direkt über Views abfragen, doch bei komplexen, auf Beziehungen basierenden, Queries sind wir recht beschränkt. Wenn Wayne Coyne von den Flaming Lips sein Theremin vor einem Konzert verliert, könnten wir Charlie Clouser von Nine Inch Nails frage, der ebenfalls Theremin spielt. Oder wir könnten Künstler mit sich überschneidenden Talenten aufspüren, selbst wenn sie in verschiedenen Bands unterschiedliche Rollen übernehmen – all das mit einer einfachen Traversierung.

Nachdem unsere Ausgangsdaten nun an Ort und Stelle sind, müssen wir Neo4j mit CouchDB synchron halten, sollten sich die Daten der Haupt-Datenquelle jemals ändern. Darum schlagen wir zwei Fliegen mit einer Klappe, indem wir einen Dienst entwickeln, der Neo4j mit allen Änderungen an CouchDB seit Bestehen der Datenbank versorgt.

Wir wollen Redis auch mit den Schlüsseln für die Bands, Künstler und Rollen versorgen, damit wir später schnell auf diese Daten zugreifen können. Glücklicherweise umfasst das alle Daten, die wir bereits in CouchDB eingepflegt haben. Dadurch sparen wir uns separate Befüll-Schritte für Neo4j und Redis.

Stellen Sie sicher, dass Neo4j an Port 7474 läuft, oder ändern Sie die `createClient`-Funktion entsprechend. CouchDB und Redis müssen immer noch laufen. Laden Sie die folgende Datei herunter und führen Sie sie aus. Das Programm läuft so lange weiter, bis Sie es beenden.

```
$ node graph_sync.js
```

Dieser Server nutzt das kontinuierliche Polling aus dem CouchDB-Kapitel, um alle Änderungen an CouchDB nachzuhalten. Wenn eine Änderung erkannt wird, machen wir zwei Dinge: Wir befüllen Redis und wir befüllen Neo4j. Der Code pflegt die Daten über kaskadierende Fallback-Funktionen in Redis ein, zuerst werden die Bands als `"band-name:Bandname"` befüllt. Dieses Muster wird dann auch für den Künstlernamen und die Rollen genutzt.

Auf diese Weise können wir mit Teilstrings suchen. Beispielsweise würde `KEYS band-name:Bea*` die Beach Boys, Beastie Boys, Beatles und so weiter zurückgeben.

redis/graph_sync.js

```
function feedBandToRedis(band) {
  redisClient.set('band-name:' + band.name, 1);
  band.artists.forEach(function(artist) {
    redisClient.set('artist-name:' + artist.name, 1);
    artist.role.forEach(function(role){
      redisClient.set('role-name:' + role, 1);
```

Im nächsten Block befüllen wir Neo4j. Wir haben einen Treiber namens `neo4j_caching_client.js` entwickelt, den Sie als Teil des Codes zu diesem Buch herunterladen können. Es nutzt die HTTP-Bibliothek von Node.js, um die Verbindung zum Neo4j REST-Interface herzustellen. Wir begrenzen darin die Bandbreite ein wenig, damit der Client nicht zu viele Verbindungen auf einmal herstellt. Unser Treiber nutzt auch Redis, um die im Neo4j-Graph vorgenommenen Änderungen nachzuhalten, ohne eine separate Query initiieren z müssen. Das ist die dritte Aufgabe, die wir Redis zukommen lassen (die erste ist der Transformationsschritt zur Befüllung von CouchDB und die zweite, die Sie gerade eben gesehen haben, die schnelle Suche nach Bandnamen)

Der Code erzeugt Band-Knoten (wenn sie erzeugt werden müssen), dann Künstler-Knoten (wenn sie erzeugt werden müssen) und dann die Rollen. Jeder Schritt erzeugt neue Beziehungen, d. h., der The-Beatles-Knoten wird mit den John-, Paul-, George- und Ringo-Knoten verknüpft, die wiederum mit ihren Rollen verknüpft werden.

redis/graph_sync.js

```
function feedBandToNeo4j(band, progress) {
  var
    lookup = neo4jClient.lookupOrCreateNode,
    relate = neo4jClient.createRelationship;

  lookup('bands', 'name', band.name, function(bandNode) {
    progress.emit('progress', 'band');
    band.artists.forEach(function(artist) {
      lookup('artists', 'name', artist.name, function(artistNode){
        progress.emit('progress', 'artist');
        relate(bandNode.self, artistNode.self, 'member', function(){
          progress.emit('progress', 'member');
        });
        artist.role.forEach(function(role){
          lookup('roles', 'role', role, function(roleNode){
            progress.emit('progress', 'role');
            relate(artistNode.self, roleNode.self, 'plays', function(){
              progress.emit('progress', 'plays');
```

Lassen Sie diesen Dienst in einem eigenen Fenster laufen. Jedes Update in CouchDB, das einen neuen Künstler oder eine neue Rolle für einen bestehenden Künstler einfügt, führt zu einer neuen Beziehung in Neo4j und möglicherweise zu neuen Schlüsseln in Redis. Solange dieser Dienst läuft, sollten die Daten immer synchron sein.

Öffnen Sie Ihre CouchDB-Web-Console und öffnen Sie eine Band. Nehmen Sie beliebige Änderungen an den Daten vor: Nehmen Sie ein neues Bandmitglied auf (machen Sie sich selbst zu einem Mitglied der Beatles) oder weisen Sie einem Künstler eine neue Rolle zu. Beachten Sie dabei die graph_sync-Ausgabe. Öffnen Sie dann die Neo4j-Console und suchen Sie nach neuen Verbindungen im Graphen. Wenn Sie ein neues Bandmitglied eingefügt haben, sollten Sie eine Beziehung zum Band-Knoten finden. Die aktuelle Implementierung löscht Beziehungen nicht. Allerdings müsste man den Code auch nicht komplett umschreiben, um die Neo4j `DELETE`-Operation in das Skript zu integrieren.

Der Service

Das ist der Teil, auf den wir hingearbeitet haben. Wir wollen eine einfache Webanwendung entwickeln, die es dem Benutzer erlaubt, nach einer Band zu suchen. Jede Band im System führt ihre Bandmitglieder als Links auf,

und ein Klick auf den Bandmitglied-Link liefert einige Informationen über den Künstler zurück – namentlich die Rolle, die er einnimmt. Zusätzlich wird jeder Künstler im System aufgelistet, der die gleiche Rolle ausfüllt.

Die Suche nach Led Zeppelin liefert uns beispielsweise Jimmy Page, John Paul Jones, John Bonham und Robert Plant zurück. Wenn Sie Jimmy Page anklicken, sehen Sie, dass er Gitarre spielt, und es erscheint eine Liste mit vielen anderen Künstlern, die ebenfalls Gitarre spielen, etwa The Edge von U2.

Um die Entwicklung unserer Webanwendung etwas zu vereinfachen, benötigen wir zwei weitere node-Pakete: bricks (ein einfaches Web-Framework) und mustache (eine Templating-Bibliothek).

```
$ npm install bricks mustache
```

Wie in den vorigen Abschnitten müssen alle Datenbanken laufen, wenn Sie den Server starten. Laden Sie den folgenden Code herunter und führen Sie ihn aus:

```
$ node band.js
```

Der Server läuft an Port 8080, d. h., wenn Sie mit Ihrem Browser `http://localhost:8080/` besuchen, sollten Sie ein einfaches Suchformular sehen.

Sehen wir uns den Code an, der eine Webseite aufbaut, die die Band-Informationen enthält. Jeder URL übernimmt bei unserem kleinen HTTP-Server eine eigene Funktion. Die erste finden Sie auf `http://localhost:8080/band` und Sie erwartet einen Bandnamen als Parameter.

redis/bands.js

```
appServer.addRoute("^/band$", function(req, res) {
  var
    bandName = req.param('name'),
    bandNodePath = '/bands/' + couchUtil.couchKeyify( bandName ),
    membersQuery = 'g.V[[name:"'+bandName+'"]]'
                 + '.out("member").in("member").uniqueObject.name';

  getCouchDoc( bandNodePath, res, function( couchObj ) {
    gremlin( membersQuery, function(graphData) {
      var artists = couchObj && couchObj['artists'];
      var values = { band: bandName, artists: artists, bands: graphData };
      var body = '<h2>{{band}} Band Members</h2>';
      body += '<ul>{{#artists}}';
      body += '<li><a href="/artist?name={{name}}">{{name}}</a></li>';
      body += '{{/artists}}</ul>';
      body += '<h3>You may also like</h3>';
      body += '<ul>{{#bands}}';
      body += '<li><a href="/band?name={{.}}">{{.}}</a></li>';
      body += '{{/bands}}</ul>';
      writeTemplate( res, body, values );
```

Wenn Sie die Band *Nirvana* im Suchformular angeben, lautet der URL-Request `http://localhost:8080/band?name=Nirvana`. Diese Funktion erzeugt (rendert) eine HTML-Seite (das Template liegt in einer externen Datei namens `template.html`). Diese Webseite listet alle Künstler einer Band auf. Die Ergebnisse werden direkt aus CouchDB abgerufen. Sie empfiehlt auch einige Bands, was mit Hilfe einer Gremlin-Query über den Neo4j-Graph geschieht. Die Gremlin-Query sieht für Nirvana wie folgt aus:

```
g.V.filter{it.name=="Nirvana"}.out("member").in("member").dedup.name
```

Mit anderen Worten wird ausgehend vom Nirvana-Knoten nach allen Namen (Duplikate werden unterdrückt) gesucht, deren Mitglieder mit den Nirvana-Mitgliedern verbunden sind. Zum Beispiel hat Dave Grohl sowohl bei Nirvana als auch bei den Foo Fighters gespielt, weshalb Foo Fighters in der Liste zurückgegeben wird.

Die nächste Aktion ist der URL `http://localhost:8080/artist`. Diese Seite liefert Informationen über einen Künstler.

redis/bands.js

```
appServer.addRoute("^/artist$", function(req, res) {
  var
    artistName = req.param('name'),
    rolesQuery = 'g.V[[name:"'+artistName+'"]].out("plays").role.uniqueObject',
    bandsQuery = 'g.V[[name:"'+artistName+'"]].in("member").name.uniqueObject';
  gremlin( rolesQuery, function(roles) {
    gremlin( bandsQuery, function(bands) {
      var values = { artist: artistName, roles: roles, bands: bands };
      var body = '<h3>{{artist}} Performs these Roles</h3>';
      body += '<ul>{{#roles}}';
      body += '<li>{{.}}</li>';
      body += '{{/roles}}</ul>';
      body += '<h3>Play in Bands</h3>';
      body += '<ul>{{#bands}}';
      body += '<li><a href="/band?name={{.}}">{{.}}</a></li>';
      body += '{{/bands}}</ul>';
      writeTemplate( res, body, values );
```

Zwei Gremlin-Queries werden hier ausgeführt. Die erste liefert alle Rollen zurück, die ein Künstler übernimmt, und die zweite ist eine Liste aller Bands, in denen diese Person gespielt hat. Beispielsweise würde Jeff Ward (`http://localhost:8080/artist?name=Jeff%20Ward`) als Drummer in den Bands Nine Inch Nails und Ministry aufgeführt werden.

Ein cooles Feature bei den beiden Seite ist, dass wir Links zwischen diesen Werten erzeugen. Die Künstlerliste in der `/bands`-Seite verlinkt auf die

gewählte /artist-Seite und umgekehrt. Doch wir können die Suche noch etwas einfacher gestalten.

redis/bands.js

```
appServer.addRoute("^/search$", function(req, res) {
  var query = req.param('term');

  redisClient.keys("band-name:"+query+"*", function(error, keys) {
    var bands = [];
    keys.forEach(function(key){
      bands.push(key.replace("band-name:", ''));
    });
    res.write( JSON.stringify(bands) );
    res.end();
```

Hier ziehen wir alle Schlüssel aus Redis, die dem ersten Teil des Strings entsprechen (wie z. B. das vorhin beschriebene `"Bea*"`). Es gibt diese Daten dann im JSON-Format aus. Die `template.html` verlinkt auf den jQuery-Code, der nötig ist, um diese Funktion mit einem Autocomplete-Feature im Suchfeld zu versehen.

Den Service erweitern

Das ist ein recht kleines Skript mit Kernfunktionen, die wir hier implementieren. Sie werden es an vielen Stellen erweitern wollen. Beachten Sie, dass die Empfehlungen nur Bands „erster Ordnung“ berücksichtigen (also nur Bands, in denen die Bandmitglieder mitgespielt haben). Sie erhalten interessante Ergebnisse, wenn Sie eine Query schreiben, die Bands zweiter Ordnung zurückgibt etwa: `g.V.filter{it.name=='Nine Inch Nails'}.out('member').in('member').dedup.loop(3){ it.loops <= 2 }.name`.

Sie werden auch bemerken, dass wir kein Formular anbieten, in dem man Bandinformationen aktualisieren kann. Diese Funktionalität zu ergänzen, ist recht einfach, weil wir den Befüllungs-Code für CouchDB bereits in `populate_couch.js` vorliegen haben und das Befüllen von CouchDB schlussendlich auch Neo4j und Redis konsistent hält, solange der `graph_sync.js`-Service läuft.

Wenn Sie gerne mit dieser Art polyglotter Persistenz herumspielen, können Sie die Sache auch noch weiter treiben. Sie können ein Data-Warehouse in PostgreSQL integrieren,[6] um diese Daten in ein Sternschema zu transformieren – was andere Arten der Analyse erlaubt, etwa das am häufigsten gespielte Instrument oder die durchschnittliche Zahl von Bandmitgliedern gegenüber der Gesamtzahl der Instrumente. Sie können einen Riak-Server einbinden, um Samples der Musik zu speichern, einen HBase-Server nutzen, um ein Messaging-System aufzubauen, bei dem die Benutzer Likes/Dislikes nach-

6. http://en.wikipedia.org/wiki/Data_warehouse

halten, oder eine MongoDB-Erweiterung verwenden, um den Service um ein geografisches Element zu ergänzen.

Oder überarbeiten Sie das Projekt gleich ganz in einer anderen Sprache, einem anderen Web-Framework oder mit anderen Daten. Es gibt so viele Möglichkeiten dieses Projekt zu erweitern, wie es Kombinationen von Datenbanken und Techniken gibt, sie zu entwickeln – ein karthesiches Produkt der gesamten Open Source.

Was Sie am dritten Tag gelernt haben

Heute war ein großer Tag – tatsächlich so groß, dass es uns nicht gewundert hätte, wenn wir mehrere Tage gebraucht hätten, um ihn abzuschließen. Doch das ist ein kleiner Vorgeschmack auf die Zukunft der Datenmanagement – Systeme, weil sich die Welt weg vom Modell *einer großen relationalen Datenbank* hin zum Modell *mehrerer spezialisierter Datenbanken* entwickelt. Wir haben diese Datenbanken über ein wenig nichtblockierenden Code miteinander verbunden, und obwohl das nicht der Schwerpunkt dieses Buches ist, scheint es so, als wäre es auch die Richtung, in die sich die Interaktion mit Datenbanken aus Entwicklersicht hin bewegt.

Die Bedeutung von Redis in diesem Modell dürfen Sie nicht übersehen. Redis bietet sicher keine Funktionalität, die nicht auch eine der anderen Datenbanken bietet, aber sie stellt schnelle Datenstrukturen zur Verfügung. Wir konnten eine einfache Datei in eine Folge sinnvoller Datenstrukturen umwandeln, was integraler Bestandteil der Befüllung und des Transports von Daten ist. Und das geschah auf schnelle und einfach zu nutzende Art und Weise.

Selbst wenn Sie sich dem polyglotten Persistenz-Modell nicht völlig verschrieben haben, sollten Sie Redis für jedes System in Erwägung ziehen.

Tag 3: Selbststudium

Machen Sie Folgendes

1. Ändern Sie den Import-Schritt so ab, dass auch das Einstiegs- und Ausstiegsdatum eines Bandmitglieds festgehalten wird. Halten Sie diese Daten in einem CouchDB-Subdokument für den Künstler vor. Geben Sie diese Information auf der Künstler-Seite aus.
2. Fügen Sie MongoDB in die Mixtur ein, um einige Musik-Samples in GridFS zu speichern, so dass sich die Nutzer ein oder zwei Titel einer Band anhören können. Wenn irgendwelche Titel für eine Band existieren, fügen Sie einen entsprechenden Link in die Web-App ein. Stellen Sie sicher, dass die Riak-Daten und CouchDB aktuell bleiben.

8.5 Zusammenfassung

Der Schlüssel/Wert(Datenstruktur)-Speicher Redis ist leicht und kompakt, mit einer Vielzahl von Einsatzmöglichkeiten. Es ähnelt einem dieser Multitools mit Messer, Flaschenöffner und anderem Krimskrams wie einem Korkenzieher. Redis eignet sich gut für eine Vielzahl ungewöhnlicher Aufgaben. Insgesamt ist Redis so schnell, einfach und dauerhaft, wie Sie es wollen. Zwar wird es nur selten als Standalone-Datenbank genutzt, bildet aber eine perfekte Ergänzung jedes polyglotten Ökosystems, wo es einen immer präsenten Helfer für die Datentransformation, das Caching und die Verwaltung von Nachrichten (über seine blockierenden Befehle) darstellt.

Redis' Stärken

Die offensichtliche Stärke von Redis ist Geschwindigkeit, wie bei vielen anderen Schlüssel/Wert-Speichern seines Schlages. Redis bietet die Möglichkeit, komplexe Werte wie Listen, Hashes und Sets zu speichern und über für die Datentypen spezifische Befehle abzurufen. Über einen Datenstrukturspeicher hinaus können Sie mit den Optionen zur Dauerhaftigkeit Geschwindigkeit gegen Datensicherheit tauschen. Die fest eingebaute Master/Slave-Replikation ist eine weitere nette Möglichkeit, die Dauerhaftigkeit zu erhöhen, ohne das langsame Synchronisieren mit einer Append-Only-Datei bei jeder Operation. Darüber hinaus ist die Replikation eine gute Sache für Systeme mit hohem Leseaufkommen.

Redis' Schwächen

Redis ist größtenteils deshalb so schnell, weil alles im Speicher vorgehalten wird. Manche mögen das sogar für Schummeln halten, weil jede Datenbank schnell ist, die die Platte nicht nutzt. Eine hauptspeicherbasierte Datenbank hat ein angeborenes Problem mit der Dauerhaftigkeit: Wird die Datenbank vor einem Schnappschuss heruntergefahren, gehen Daten verloren. Selbst wenn jede Operation in einer Append-Only-Datei festgehalten wird, laufen Sie Gefahr, verfallene Werte wieder einzuspielen, weil man bei zeitbasierten Events nie darauf bauen kann, dass sie in exakt der gleichen Weise wiedergegeben werden (auch wenn man fairerweise zugeben muss, dass das ein eher hypothetischer denn praktischer Fall ist).

Redis unterstützt auch keine Datenmengen, die größer sind als das verfügbare RAM (Redis entfernt die Unterstützung virtuellen Speichers), d. h., es gibt eine praktische Obergrenze für die Größe. Zwar wird gerade an einem Redis-Cluster gearbeitet, das über das RAM einzelner Maschinen hinauswachsen kann, doch jeder, der sich ein Redis-Cluster wünscht, muss sich mit einem

Client, der das unterstützt (etwa der Ruby-Treiber von Tag 2) einen eigenen Cluster aufbauen.

Abschließende Gedanken

Redis steckt voller Befehle – es gibt mehr als 120 davon. Die meisten Befehle sind einfach genug, um allein durch den Namen verstanden zu werden, sobald man sich an die Idee gewöhnt hat, dass Buchstaben scheinbar willkürlich fehlen (zum Beispiel `INCRBY`) oder dass mathematische Genauigkeit manchmal eher verwirrend als hilfreich ist, zum Beispiel bei `ZCOUNT`, (Anzahl Elemente im sortierten Set) verglichen mit `SCARD` (Set-Kardinalität).

Redis ist bereits integraler Bestandteil vieler Systeme. Viele Open-Source-Projekte basieren auf Redis, von Resque, einer Ruby-basierten, asynchronen Job-Queue, bis hin zum Session-Management im Node.js-Projekt Socket-Stream. Egal welche Datenbank Sie als Haupt-Datenspeicher (SOR) wählen, Sie sollten Redis in die Mischung aufnehmen.

KAPITEL 9

Abschließende Zusammenfassung

Nachdem wir uns durch alle Datenbanken gearbeitet haben, ist mal ein Glückwunsch angesagt!

Wir hoffen, dass Sie ein grundlegendes Verständnis für diese sieben Datenbanken entwickelt haben. Wenn Sie eine davon in einem Projekt nutzen, sind wir glücklich. Und wenn Sie entscheiden, mehrere Datenbanken zu verwenden, so wie wir das am Ende des Redis-Kapitels getan haben, sind wir ekstatisch. Wir glauben, dass die Zukunft des Datenmanagements im polyglotten Persistenz-Modell liegt (bei dem mehr als eine Datenbank in einem Projekt genutzt wird), während man sich langsam von Allzweck-RDBMS entfernt.

Lassen Sie uns die Gelegenheit nutzen und sehen, wo unsere sieben Datenbanken in das große Datenbank-Ökosystem hineinpassen. Bis hierhin haben wir die Details aller Datenbanken behandelt und einige Gemeinsamkeiten und Unterschiede erklärt. Wir werden sehen, wie sie zur weiten und expandierenden Landschaft der Datenspeicher-Optionen beitragen.

9.1 Noch einmal: Gattungen

Wir haben gesehen, dass man die Art und Weise, wie Datenbanken ihre Daten speichern, ganz grob in fünf Gattungen unterteilen kann: relational, Schlüssel/Wert, spaltenorientiert, Dokument und Graph. Wir wollen uns etwas Zeit nehmen und ihre Unterschiede noch einmal rekapitulieren, und sehen, welcher Typ für was gut und nicht so gut ist, d.h., wann Sie sie einsetzen und wann Sie sie meiden sollten.

Relational

Das am weitesten verbreitete, klassische Datenbankmuster. Relationale Datenbank-Management-Systeme (RDBMSs) basieren auf der Mengenlehre und werden in Form zweidimensionaler Tabellen mit Zeilen und Spalten imple-

mentiert. Relationale Datenbanken sind streng typisiert und kennen üblicherweise numerische Werte, Strings, Datumsangaben und nicht interpretierte Blobs, doch wie wir gesehen haben, bietet PostgreSQL Erweiterungen wie Array und Cube.

Gut für:
Aufgrund ihrer strukturierten Natur eignen sich relationale Datenbanken für Projekte, bei denen das Layout der Daten im Voraus bekannt ist, nicht aber die Art und Weise, in der diese Daten später genutzt werden. Mit anderen Worten nehmen Sie die organisatorische Flexibilität im Vorfeld in Kauf, um später flexible Queries nutzen zu können. Viele Unternehmensaufgaben sind passenderweise in dieser Form modelliert, von Bestellungen über den Versand und vom Lager bis zum Online-Shop. Sie wissen im Vorfeld nicht genau, wie Sie die Daten später abfragen werden – wie viele Bestellungen haben wir im Februar verarbeitet? – , doch die Daten sind recht gleichförmig, so dass das Durchsetzen dieser Gleichförmigkeit hilfreich ist.

Nicht so gut für:
Wenn Ihre Daten hochgradig variabel oder stark hierarchisch sind, ist eine relationale Datenbank nicht die beste Wahl. Da Sie vorab ein Schema festlegen müssen, sind Datenprobleme, die ein großes Maß an Abweichungen zwischen den Datensätzen aufweisen, eher schwierig zu lösen. Nehmen wir an, Sie entwickeln eine Datenbank mit allen Lebewesen. Eine vollständige Liste aller Eigenschaften (hatHaare, numBeine, legtEier und so weiter) aufzubauen, wäre eine sperrige Angelegenheit. In einem solchen Fall brauchen Sie eine Datenbank, die Ihnen weniger Beschränkungen auferlegt, was Sie in ihr speichern können.

Schlüssel/Wert

Der Schlüssel/Wert-Speicher (Key-Value, KV), ist das einfachste von uns betrachtete Modell. KVs bilden einfache Schlüssel auf (möglicherweise) komplexere Werte wie riesige Hashtabellen ab. Aufgrund ihrer relativen Schlichtheit hat diese Gattung die größte Flexibilität der Implementierung. Hash-Lookups sind schnell, so dass Geschwindigkeit bei Redis das Hauptanliegen war. Hash-Lookups lassen sich auch leicht verteilen und Riak nutzt diese Tatsache, um sich auf einfach zu verwaltende Cluster zu konzentrieren. Natürlich ist diese Schlichtheit auch ein Nachteil, wenn die Daten eine komplexere Modellierung verlangen.

Gut für:
Ohne (oder nur wenig) Notwendigkeit zur Verwaltung von Indizes sind Schlüssel/Wert-Speicher häufig für die horizontale Skalierbarkeit, extreme Ge-

schwindigkeit oder beides konzipiert. Sie sind für Probleme besonders gut geeignet, bei denen die Daten keine besonders engen Beziehungen zueinander haben. Bei einer Web-Anwendung erfüllen beispielsweise die Session-Daten der Nutzer dieses Kriterium. Die Session-Aktivität der Nutzer unterscheidet sich von den Aktivitäten der anderen Nutzer und sie stehen in keiner Beziehung zueinander.

Nicht so gut für:
Da häufig Möglichkeiten zur Indexierung und dem Scanning fehlen, helfen Ihnen KV-Speicher nicht weiter, wenn Sie die Daten über einfache CRUD-Operationen (Create, Read, Update, Delete) hinaus abfragen müssen.

Spaltenorientiert

Spaltenorientierte Datenbanken haben mit KV- und RDBMS-Speichern vieles gemein. Wie bei Schlüssel/Wert-Datenbanken werden Werte über passende Schlüssel abgefragt. Wie beim relationalen Modell sind die Werte Gruppen von null oder mehr Spalten, wenngleich jede Zeile so viele enthalten kann, wie sie will. Im Gegensatz zu den beiden speichern spaltenorientierte Datenbanken Daten aber in Spalten und nicht in Zeilen. Das Einfügen von Spalten ist billig, die Versionierung trivial und es gibt keine echten Speicherkosten bei unbefüllten Werten. Wir haben HBase als klassische Implementierung dieser Gattung kennengelernt.

Gut für:
Spaltenorientierte Datenbanken wurden traditionell mit horizontaler Skalierbarkeit als primärem Entwurfziel entwickelt. Als solche eignen sie sich besonders gut für „Big Data"-Probleme und sind in Clustern von mehreren zehn, hundert oder gar tausend Knoten zu Hause. Sie tendieren auch zu fest eingebauten Features wie Komprimierung und Versionierung. Das Paradebeispiel für ein gut geeignetes Problem für spaltenorientierte Datenbanken ist die Indexierung von Webseiten. Die Seiten im Web sind textlastig (profitieren also von der Komprimierung), hängen in gewisser Weise zusammen und ändern sich mit der Zeit (profitieren also von der Versionierung).

Nicht so gut für:

Die verschiedenen spaltenorientierten Datenbanken haben unterschiedliche Features und daher auch unterschiedliche Nachteile. Doch eine Sache haben sie alle gemeinsam: Sie sollten Ihr Schema darauf basierend entwerfen, wie Sie die Daten abfragen wollen. Sie sollten also im Voraus eine Vorstellung davon haben, wie Sie die Daten nutzen wollen, und nicht nur, wie die Daten aussehen. Wenn das Nutzungsmuster für die Daten nicht im Vorfeld

definiert werden kann – zum Beispiel schnelles Ad-hoc-Reporting – , ist eine spaltenorientierte Datenbank möglicherweise nicht die beste Wahl.

Dokument

Dokument-Datenbanken erlauben eine beliebige Anzahl von Feldern pro Objekt und s,olche Objekte können als Werte anderer Felder beliebig tief verschachtelt werden. Üblicherweise werden diese Objekte in JavaScript Object Notation (JSON) repräsentiert, woran sowohl MongoDB als auch CouchDB festhalten (obwohl das in keiner Weise eine konzeptionelle Notwendigkeit ist). Da die Dokumente im Gegensatz zu (beispielsweise) relationalen Datenbanken nicht in Beziehung zueinander stehen, lassen sie sich relativ leicht auf mehrere Server verteilen und replizieren, weshalb verteilte Implementierungen gängig sind. MongoDB widmet sich dem Thema (Hoch)Verfügbarkeit, indem es den Aufbau von Rechenzentren unterstützt, die riesige Datenmengen für das Web verwalten. CouchDB konzentriert sich währenddessen darauf, einfach und dauerhaft zu sein und erreicht Verfügbarkeit über die Master/Master-Replikation recht autonomer Knoten. Es gibt sehr viele Überschneidungen bei diesen Projekten.

Gut für:

Dokument-Datenbanken eigenen sich für Probleme hochgradig variabler Domänen. Wenn Sie im Voraus nicht wissen, wie Ihre Daten genau aussehen werden, sind Dokument-Datenbanken eine gute Wahl. Aufgrund der Natur von Dokumenten eignen sie sich häufig auch gut zur Abbildung objektorientierter Programmiermodelle. Das führt zu weniger Problemen, wenn Sie Daten zwischen Datenbank- und Anwendungsmodell verschieben.

Nicht so gut für:

Wenn Sie an ausgefeilte Join-Queries in hochgradig normalisierten Schemata relationaler Datenbanken gewöhnt sind, werden Sie die Fähigkeiten von Dokument-Datenbanken unzureichend finden. Ein Dokument sollte generell alle (oder die meisten) relevanten Informationen enthalten, die für den normalen Einsatz benötigt werden. Während Sie also bei einer relationalen Datenbank Ihre Daten grundsätzlich normalisieren, um die Daten zu reduzieren oder Kopien zu eliminieren, damit die Daten auf dem neuesten Stand bleiben, ist die denormalisierte Form bei Dokument-Datenbanken die Regel.

Graph

Graph-Datenbanken bilden eine neu entstehende Klasse von Datenbanken, die sich mehr auf die wechselseitigen Beziehungen der Daten konzentrieren

als auf die Werte selbst. Neo4j, als unser Open-Source-Beispiel, gewinnt in Anwendungen für soziale Netzwerke immer mehr an Beliebtheit. Im Gegensatz zu anderen Datenbank-Gattungen, die Collections gleichartiger Objekte in gemeinsame Buckets packen, haben Graph-Datenbanken eher eine freie Form – bei den Queries verfolgt man Kanten zwischen zwei Knoten, d.h., es geht um die *Traversierung* von Knoten. Je mehr Projekte sie nutzen, desto mehr weiten sich Graph-Datenbanken über die offensichtlichen „sozialen" Beispiele hinaus aus und besetzen weitere Nischen wie Empfehlungs-Engines, Zugriffskontrolllisten und geografische Daten.

Gut für:

Graph-Datenbanken scheinen für Netzwerk-Anwendungen maßgeschneidert zu sein. Das prototypische Beispiel ist ein soziales Netzwerk, bei dem Knoten die Nutzer repräsentieren, die unterschiedliche Beziehungen zueinander haben. Die Modellierung solcher Daten über die anderen Gattungen ist häufig ein schweres Unterfangen, während eine Graph-Datenbank sie mit Leichtigkeit meistert. Sie eignen sich auch hervorragend für objektorientierte Systeme. Wenn Sie Ihre Daten auf einem Whiteboard modellieren können, lassen sie sich auch in einen Graphen überführen.

Nicht so gut für:

Aufgrund des hohen Grades der Vernetzung zwischen den Knoten eignen sich Graph-Datenbanken generell nicht für die Netzwerk-Partitionierung. Die schnelle Traversierung durch den Graphen erlaubt keine Sprünge zu anderen Netzwerk-Knoten, weshalb Graph-Datenbanken nicht besondere gut skalieren. Wenn Sie eine Graph-Datenbank nutzen, ist sie wahrscheinlich Teil eines größeren Systems, bei dem der Großteil der Daten irgendwo anders gespeichert wird und nur die Beziehungen untereinander im Graph gespeichert werden.

9.2 Eine Wahl treffen

Wie wir es am Anfang bereits gesagt haben, sind Daten das neue Öl. Wir sitzen auf einem riesigen Ozean von Daten, doch solange ihn niemand zu Informationen raffiniert, ist er nutzlos (weniger vornehm ausgedrückt steckt heutzutage jede Menge Geld in den Daten). Wie einfach oder schwierig es ist, diese Daten zu sammeln und letztlich zu speichern, zu verarbeiten und zu veredeln, beginnt mit der Wahl der Datenbank.

Die Entscheidung für eine Datenbank ist häufig komplexer als die bloße Frage, welche Gattung die Daten der gegebenen Domäne am besten abbildet. Auch wenn ein sozialer Graph offensichtlich mit einer Graph-Datenbank

am besten funktioniert, haben Sie als Facebook bei Weitem zu viele Daten, um eine nutzen zu können. Sie werden stattdessen eher eine „Big Data"-Implementierung wie HBase oder Riak wählen. Das zwingt Sie in Richtung eines spaltenorientierten oder eines Schlüssel/Wert-Speichers. In anderen Fällen, wo Sie eine relationale Datenbank als beste Option für Banktransaktionen ansehen, ist es hingegen nützlich zu wissen, dass Neo4j ebenfalls ACID-Transaktionen unterstützt, wodurch sich Ihre Optionen vergrößern.

Diese Beispiele sollen nur hervorheben, dass es bei der Wahl einer (oder mehrerer) Datenbank(en) neben der Gattung noch andere Aspekte zu berücksichtigen gilt, um die Aufgabe bestmöglich lösen zu können. Die generelle Regel lautet, dass mit der Größe der Daten die Fähigkeiten bestimmter Datenbank-Gattungen abnehmen. Spaltenorientierte Datenspeicher-Implementierungen sind häufig so konzipiert, dass sie über Rechenzentren hinweg skalieren und die größten „Big Data"-Datenmengen unterstützen, während Graphen die kleinsten unterstützen. Das ist aber nicht immer der Fall. Riak ist ein groß angelegter Schlüssel/Wert-Speicher, der Daten über hunderte oder tausende von Knoten verteilen kann, während Redis entwickelt wurde, um auf einem Knoten zu laufen, aber die Möglichkeit einiger Master/Slave-Replicas oder durch Clients verwalteter Shards bietet.

Es gibt bei der Wahl einer Datenbank noch sehr viel mehr Aspekte zu beachten, etwa Dauerhaftigkeit, Verfügbarkeit, Konsistenz, Skalierbarkeit und Sicherheit. Sie müssen entscheiden, ob Ad-hoc-Queries wichtig sind oder ob Mapreduce reicht. Bevorzugen Sie ein HTTP/REST-Interface oder sind Sie bereit, einen Treiber mit eigenem Binärprotokoll zu verwenden? Selbst kleinere Erwägungen, etwa die Möglichkeit des Massenimports von Daten, könnten für Sie von Bedeutung sein.

Um den Vergleich zwischen diesen Datenbanken zu vereinfachen, stellen wir in Anhang 1, *Datenbank-Übersichtstabellen*, auf Seite 343 eine entsprechende Tabelle zur Verfügung. Diese Tabelle enthält keine umfassende Liste aller Features. Vielmehr soll sie einen schnellen Vergleich zwischen den von uns betrachteten Datenbanken ermöglichen. Beachten Sie die Versionen der jeweiligen Datenbanken. Die Features ändern sich häufig, weshalb Sie die Werte mit aktuellen Werten abgleichen sollten.

9.3 Wie geht es weiter?

Skalierungsprobleme moderner Anwendungen fallen nun in den Bereich des Datenmanagements. Wir haben in der Evolution von Anwendungen einen Punkt erreicht, wo die Wahl von Programmiersprache, Framework und Betriebssystem – und sogar der Hardware und des Betriebs (dank „Virtual Ma-

chine"-Hosts und „der Cloud") – so günstig und einfach ist, dass sie größtenteils zu einem trivialen Problem geworden ist, das eher durch Vorlieben denn durch Notwendigkeiten bestimmt wird. Wenn Sie heutzutage über die Skalierung Ihrer Anwendung nachdenken, sollten Sie sich eher über die Wahl der Datenbank (oder der Datenbanken) Gedanken machen – sehr wahrscheinlich bilden sie den eigentlichen Flaschenhals. Ihnen bei der richtigen Wahl zu helfen, war das Hauptziel dieses Buches.

Auch wenn Sie am Ende dieses Buches angelangt sind, glauben wir, dass Ihr Interesse an polyglotter Persistenz hoch ist. Die nächsten Schritte bestehen nun darin, sich die Datenbanken genauer anzusehen, die Ihr Interesse geweckt haben, oder weitere Alternativen wie Cassandra, Drizzle oder OrientDB kennenzulernen.

Es ist an der Zeit, sich die Hände schmutzig zu machen.

ANHANG 1

Datenbank-Übersichtstabellen

Dieses Buch enthält viele Informatoinen zu jeder der sieben betrachteten Datenbanken: PostgreSQL, Riak, HBase, MongoDB, CouchDB, Neo4j und Redis. Auf den folgenden Seiten finden Sie ein paar Tabellen, die diese Datenbanken unter bestimmten Aspekten miteinander vergleichen, um Ihnen einen Überblick dessen zu geben, was an anderer Stelle in diesem Buch detaillierter behandelt wird. Zwar sind diese Tabellen kein Ersatz für echtes Verständnis, doch sie bieten Ihnen einen Überblick darüber, was eine Datenbank kann, wo sie Schwächen zeigt und wie sie sich in die moderne Datenbank-Landschaft einfügt.

	Gattung	Version	Datentypen	Datenrelationen
MongoDB	Dokument	2.0	Typisiert	Keine
CouchDB	Dokument	1.1	Typisiert	Keine
Riak	Schlüssel/Wert	1.0	Blob	Ad-hoc (Links)
Redis	Schlüssel/Wert	2.4	Semi-typed	Keine
PostgreSQL	Relational	9.1	Vordefiniert und typisiert	Vordefiniert
Neo4j	Graph	1.7	Untypisiert	Ad-hoc (Kanten)
HBase	Spaltenorientiert	0.90.3	Vordefiniert und typisiert	Keine

	Standard-Objekt	Geschrieben in	Interface-Protokoll	HTTP/REST
MongoDB	JSON	C++	Eigenes über TCP	Simple
CouchDB	JSON	Erlang	HTTP	Ja
Riak	Text	Erlang	HTTP, protobuf	Ja
Redis	String	C/C++	Einfacher Text über TCP	Nein
PostgreSQL	Tabelle	C	Eigenes über TCP	Nein
Neo4j	Hash	Java	HTTP	Ja
HBase	Spalten	Java	Thrift, HTTP	Ja

	Ad-hoc-Query	Mapreduce	Skalierbarkeit	Dauerhaftigkeit
MongoDB	Befehle, Mapreduce	JavaScript	Rechenzentrum	Write-Ahead-Journaling, Safe-Modus
CouchDB	Temporäre Views	JavaScript	Rechenzentrum (über BigCouch)	Absturz
Riak	Schwach, Lucene	JavaScript, Erlang	Rechenzentrum	Dauerhaftes Schreib-Quorum
Redis	Befehle	Nein	Cluster (über Master/Slave)	Append-Only-Log
PostgreSQL	SQL	Nein	Cluster (über Erweiterungen)	ACID
Neo4j	Graph-Traversierung, Cypher, Search	Nein (im Sinne von Verteilung)	Cluster (über HA)	ACID
HBase	Schwach	Hadoop	Rechenzentrum	Write-Ahead-Log

	Sekundärindizes	Versionierung	Massenimport (Bulk Load)	Sehr große Dateien
MongoDB	Ja	Nein	Mongoimport	GridFS
CouchDB	Ja	Ja	Bulk Doc-API	Attachments
Riak	Ja	Ja	Nein	Lewak (veraltet)
Redis	Nein	Nein	Nein	Nein
PostgreSQL	Ja	Nein	COPY-Befehl	BLOBs
Neo4j	Ja (über Lucene)	Nein	Nein	Nein
HBase	Nein	Ja	Nein	Nein

	Verlangt Verdichtung	Replikation	Sharding	Nebenläufigkeit
MongoDB	Nein	Master/Slave (über Replica-Sets)	Ja	Schreib-Locking
CouchDB	Erneutes Schreiben der Datei	Master/Master	Ja (mit Filtern in BigCouch)	Lock-freies MVCC
Riak	Nein	Peer-basiert, Master/Master	Ja	Vektoruhren
Redis	Snapshot	Master/Slave	Erweiterungen (z. B. Client)	Keine
PostgreSQL	Nein	Master/Slave	Erweiterungen (z. B. PL/Proxy)	Schreib-Locking für Tabelle/Zeile
Neo4j	Nein	Master/Slave (bei Enterprise Edition)	Nein	Schreib-Locking
HBase	Nein	Master/Slave	Ja (über HDFS)	Konsistent je Zeile

	Transaktionen	Trigger	Sicherheit	Mandantenfähig (multitenancy)
MongoDB	Nein	Nein	Benutzer	Ja
CouchDB	Nein	Update-Validierung oder Changes-API	Benutzer	Ja
Riak	Nein	Pre-/Post-commits	Keine	Nein
Redis	Queues mit mehreren Operationen	Nein	Passwörter	Nein
PostgreSQL	ACID	Ja	Benutzer/ Gruppen	Ja
Neo4j	ACID	Transaktions-Eventhandler	Keine	Nein
HBase	Ja (wenn aktiviert)	Nein	Kerberos über Hadoop	Nein

	Haupt-Unterscheidungsmerkmal	Schwächen
MongoDB	einfache *Big Data*-Queries	Einbettbarkeit
CouchDB	Dauerhaft und einbettbare Cluster	Abfragbarkeit
Riak	Hochverfügbarkeit	Abfragbarkeit
Redis	Sehr, sehr schnell	Komplexe Daten
PostgreSQL	Bestes OSS RDBMS-Modell	Verteilte Verfügbarkeit
Neo4j	Flexible Graphen	BLOBs oder Terabyte-Größen
HBase	sehr großräumig, Hadoop-Infrastruktur	Flexibles Mitwachsen, Abfragbarkeit

ANHANG 2

Das CAP-Theorem

Die fünf Datenbank-Gattungen zu verstehen, ist ein wichtiges Auswahlkriterium, aber nicht das einzige. Ein immer wiederkehrendes Thema in diesem Buch ist das CAP-Theorem, das eine erschreckende Wahrheit darüber offenbart, wie sich verteilte Datebanksysteme im Angesicht von Netzwerk-Instabilitäten verhalten.

CAP beweist, dass Sie eine verteilte Datenbank entwickeln können, die *konsistent* (Schreiboperationen sind atomisch und alle nachfolgenden Requests liefern den neuen Wert zurück), *verfügbar* (die Datenbank gibt immer einen Wert zurück, solange noch ein Server läuft) oder *partitionstolerant* (das System funktioniert auch dann noch, wenn die Serverkommunikation zeitweise verloren geht, d. h., eine Netzwerkpartition entsteht) ist. Doch Sie können nur zwei dieser Dinge gleichzeitig erreichen.

Mit anderen Worten: Sie können ein verteiltes Datenbanksystem aufbauen, das *konsistent* und *Partitionstolerant* ist, oder *verfügbar* und *partitionstolerant*, oder *konsistent* und *verfügbar* (aber nicht *partitionstolerant* – was im Wesentlichen „nicht verteilt" bedeutet). Doch es ist nicht möglich, eine verteilte Datenbank zu entwickln, die gleichzeitig konsistent, verfügbar und partitionstolerant ist.

Das CAP-Theorem ist relevant, wenn Sie über eine verteilte Datenbank nachdenken, da Sie entscheiden müssen, worauf Sie zu verzichten bereit sind. Die von Ihnen gewählte Datenbank verliert Verfügbarkeit oder Konsistenz. Partitionstoleranz ist eine rein architektonische Entscheidung (ist die Datenbank verteilt oder nicht). Es ist wichtig, das CAP-Theorem zu verstehen, um ihre Möglichkeiten vollständig zu begreifen. Die Kompromisse, die die Datenbank-Implementierungen in diesem Buch machen, werden größtenteils von diesem Theorem beeinflusst.

Ein CAP Abenteuer, Teil I: CAP

Stellen Sie sich die Welt als riesiges verteiltes Datenbanksystem vor. Alles Land der Welt enthält Informationen zu bestimmten Themen, und solange Sie sich irgendwie in der Nähe von Menschen oder Technik aufhalten, erhalten Sie Antworten auf ihre FIagen.

Nehmen wir nun an, Sie sind leidenschaftlicher Beyoncé-Knowles-Fan und es ist der 5. September 2006. Während Sie mit Ihren Freunden auf einer Strandparty die Veröffentlichung von Beyoncés zweitem Studio-Album feiern, werden Sie von einer plötzlichen Flutwelle auf das Meer hinausgespült. Sie können sich auf ein behelfsmäßiges Floß retten und werden Tage später an einer einsamen Insel angespült. Ohne eine Möglichkeit zur Kommunikation sind Sie effektiv vom Rest des Systems (der Welt) abgetrennt (*partitioniert*). Dort warten Sie fünf lange Jahre ...

Eines Morgens im Jahr 2011 erwachen Sie durch Rufe vom Meer. Der Kapitän eines alten Schoners hat Sie entdeckt! Nach all diesen Jahren beugt sich der Kapitän über die Reling und brüllt: „Wie viele Studio-Alben hat Beyoncé gemacht?“

Sie müssen nun eine Entscheidung treffen. Sie können die Frage mit dem letzten Wert beantworten, den Sie haben (und der fünf Jahre alt ist). Oder Sie können eine Antwort verweigern, weil Sie wissen, dass Sie abgeschnitten waren und Ihre Antwort daher möglicherweise mit dem Rest der Welt nicht *konsistent* ist. Der Kapitän bekommt keine Antwort, aber der Zustand der Welt bleibt konsistent (wenn er nach Hause segelt, bekommt er die richtige Antwort). In Ihrer Rolle als abgefragter Knoten können Sie die Daten der Welt *konsistent* halten oder *verfügbar* sein, aber *nicht beides*.

A2.1 Schlussendliche Konsistenz

Verteilte Datenbanken müssen partitionstolerant sein, daher kann die Wahl zwischen Verfügbarkeit und Konsistenz schwierig sein. Wenn Sie sich für Verfügbarkeit entscheiden, ist nach CAP keine echte Konsistenz möglich, doch Sie können dennoch *schlussendliche Konsistenz* bieten.

Die Idee hinter schlussendlicher Konsistenz ist, dass jeder Knoten immer verfügbar ist, um Requests zu beantworten. Als Kompromiss werden Modifikationen an den Daten im Hintergrund an andere Knoten weitergegeben. Das bedeutet, dass das System zu jeder Zeit inkonsistent sein kann, die Daten aber dennoch größtenteils richtig sind.

Der Domain Name Service (DNS) des Internets ist ein erstklassiges Beispiel für ein schlussendlich konsistentes System. Sie registrieren eine Domain und es kann einige Tage dauern, bis sie an alle DNS-Server im Internet weitergegeben wird. Doch zu keiner Zeit ist ein bestimmter DNS-Server nicht erreichbar (solange Sie eine Verbindung mit ihm herstellen können).

Ein CAP-Abenteuer, Teil II: Schlussendliche Konsistenz

Spulen wir zwei Jahre zurück. Sie waren zu diesem Zeitpunkt drei Jahre auf der Insel und Sie entdecken eine Flasche im Sand – wertvoller Kontakt zum Rest der Welt. Sie entkorken die Flasche und frohlocken! Sie haben gerade etwas Wichtiges erfahren ...

Die Zahl von Beyoncés Studio-Alben ist von größter Bedeutung für das aggregierte Wissen dieser Welt. Tatsächlich ist es so wichtig, dass bei jeder Veröffentlich eines neuen Albums jemand das aktuelle Datum und die Zahl auf ein Stück Papier schreibt. Dieses Papier wird in eine Flasche gesteckt, die dann ins Meer geworfen wird. Wenn jemand wie Sie auf einer einsamen Insel vom Rest der Welt abgetrennt ist, besitzt er *schlussendlich* die richtige Antwort.

Zurück in die Gegenwart. Wenn der Schiffskapitän Sie jetzt fragt: „Wie viele Studio-Alben hat Beyoncé?“, bleiben Sie *verfügbar* und antworten „drei“. Sie können zum Rest der Welt *in*konsistent sein, doch Sie sind sich im Bezug auf Ihre Antwort recht sicher, da noch keine weitere Flasche angespült wurde.

Die Geschichte endet damit, dass der Kapitän Sie rettet und Sie bei Ihrer Rückkehr das neue Album vorfinden und glücklich bis ans Ende leben. Solange Sie an Land bleiben, müssen Sie nicht partitionstolerant sein und können bis ans Ende aller Tage konsistent und verfügbar sein.

A2.2 CAP in freier Wildbahn

Bei einigen partitionstoleranten Datenbanken kann man requestbezogen festlegen, ob sie eher konsistent oder eher verfügbar sein soll. Riak arbeitet so und erlaubt den Clients bei einem Request zu entscheiden, welcher Grad an Konsistenz benötigt wird. Die anderen Datenbanken belegen größtenteils die eine oder andere Ecke im CAP-Kompromiss-Dreieck.

Redis, PostgreSQL und Neo4J sind konsistent und verfügbar (CA); sie verteilen keine Daten, weshalb die Partitionierung kein Thema ist (und CAP macht bei nicht verteilten Systemen zugegebenermaßen keinen Sinn). MongoDB und HBase sind generell konsistent und partitionstolerant (CP). Im Falle einer Partitionierung des Netzwerks können sie auf bestimmte Arten von Queries möglicherweise nicht reagieren (zum Beispiel ist in einem Mongo-Replica-Set `slaveok` für Leseoperationen auf „falsch“ gesetzt). In der Praxis werden Hardwarefehler großzügig behandelt – andere noch vernetzte Knoten können den ausgefallenen Server ersetzen – , doch aus Sicht des CAP-Theorems sind sie nicht verfügbar. CouchDB ist schließlich verfügbar und partitionstolerant (AP). Obwohl zwei oder mehr CouchDB-Server Daten untereinander replizieren können, garantiert CouchDB die Konsistenz zwischen irgendwelchen Servern nicht.

Erwähnenswert ist noch, dass Sie bei den meisten dieser Datenbanken den CAP-Typ ändern können (Mongo kann CA sein, CouchDB CP), doch hier haben wir nur ihr Standardverhalten beschrieben.

A2.3 Der Latenz-Kompromiss

Beim Design verteilter Datenbanksysteme geht es aber um mehr als nur um CAP. Beispielsweise sind kurze Latenzzeiten (Geschwindigkeit) ein Hauptanliegen vieler Systemarchitekten. Wenn Sie sich Amazons Dynamo-Papier[1] ansehen, lesen Sie viel über Verfügbarkeit, aber auch über Amazons Anforderunge an die Latenz. Bei bestimmten Klassen von Anwendungen können selbst kleine Änderungen der Latenz zu hohen Kosten führen. Yahoos PNUTS-Datenbank ist berühmt dafür, sowohl die Verfügbarkeit im normalen Betrieb als auch die Konsistenz für Partitionen aufzugeben, nur im das letzte Quäntchen Latenz au dem Design herauszuquetschen.[2]CAP ist beim Umgang mit verteilten Datenbanken ein wichtiger Aspekt, doch es ist auch wichtig zu bedenken, dass die Theorie zu verteilten Datenbanken damit nicht endet.

1. http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
2. http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

Literaturverzeichnis

[TH01] David Thomas und Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison-Wesley, Reading, MA, 2001.

[Tat10] Bruce A. Tate. *Sieben Wochen, sieben Sprachen*. O'Reilly Verlag 2011, ISBN 978-3-89721-322-7

Index

D

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z