mitp

Nathan Marz

mit James Warren

Big Data

Entwicklung und Programmierung von Systemen für große Datenmengen und Einsatz der Lambda-Architektur

Nathan Marz, James Warren

Big Data

Entwicklung und Programmierung von Systemen für große Datenmengen und Einsatz der Lambda-Architektur

Übersetzung aus dem Amerikanischen
von Knut Lorenzen

mitp

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.

ISBN 978-3-95845-176-6
1. Auflage 2016

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

Übersetzung der amerikanischen Originalausgabe:
Nathan Marz, James Warren: Big Data: Principles and best practices of scalable realtime data systems, 9781617290343

Lektorat: Sabine Schulz
Sprachkorrektorat: Maren Feilen
Fachkorrektorat: Werner Keil
Coverbild: © Dreaming Andy / fotolia.com
Satz: III-satz, Husby, www.drei-satz.de

Inhaltsverzeichnis

Vorwort

Als ich zum ersten Mal mit dem Thema Big Data Bekanntschaft machte, hatte ich das Gefühl, in den Wilden Westen der Softwareentwicklung geraten zu sein. Allenthalben wurden relationale Datenbanken und die damit einhergehenden Annehmlichkeiten durch NoSQL-Datenbanken ersetzt, deren Datenmodelle äußerst beschränkt waren, weil sie auf Tausenden von Rechnern betrieben wurden. Die Anzahl dieser NoSQL-Datenbanken, die sich oft kaum voneinander unterschieden, war wirklich überwältigend. Doch dann schickte sich ein neues Projekt namens *Hadoop* an, für Furore zu sorgen – denn hiermit sollte es angeblich möglich sein, enorme Datenmengen verschiedensten eingehenden Analysen zu unterziehen. Aus all diesen neuen Tools schlau zu werden, war allerdings gar nicht so einfach.

Ich kümmerte mich damals in dem Unternehmen, für das ich tätig war, um die Skalierungsprobleme, mit denen wir uns dort konfrontiert sahen. Die zugrundeliegende Architektur war unglaublich komplex – ein Geflecht aus mittels Sharding verknüpften relationalen Datenbanken, Warteschlangen sowie Worker- und Master-/Slave-Prozessen. Es hatten sich bereits defekte Datensätze in die Datenbanken eingeschlichen und so enthielt die Anwendung speziellen zusätzlichen Code, der solche Beschädigungen handhabte. Außerdem hinkten die Slaves ständig hinterher. Ich hielt daher nach alternativen Big-Data-Technologien Ausschau, um ein besseres Design für unsere Datenarchitektur zu finden.

Ein Erlebnis aus dieser Anfangszeit meiner Laufbahn als Softwareentwickler hat meine Ansichten hinsichtlich der Systemarchitektur in besonderer Weise geprägt: Ein Kollege hatte mehrere Wochen damit zugebracht, Daten aus dem Internet zu sammeln und sie auf einem gemeinsam genutzten Dateisystem abzulegen, bis sie in ihrer Gesamtheit für eine geplante Analyse ausreichen würden. Eines Tages passierte es jedoch, dass ich bei routinemäßigen Aufräumarbeiten versehentlich eben diesen gesamten Datenbestand löschte und das Projekt meines Kollegen damit um Wochen zurückwarf.

Mir war klar, dass ich einen großen Fehler begangen hatte, und als Neuling machte ich mir natürlich Sorgen, welche Konsequenzen das für mich haben würde. Würde man mich wegen meiner Unbesonnenheit womöglich sogar feuern? Ich schickte dem Team eine E-Mail, in der ich mich überschwänglich entschuldigte – und zu meiner großen Überraschung zeigten alle sehr viel Verständnis. Ich werde nie vergessen, wie ein Kollege an meinen Schreibtisch herantrat, mir auf die Schulter klopfte und sagte: »Herzlichen Glückwunsch! Jetzt bist du ein richtiger Softwareentwickler.«

In dieser scherzhaften Bemerkung steckte allerdings mehr als nur ein Körnchen Wahrheit über die Realität der Softwareentwicklung: Wir wissen schlicht und einfach nicht, wie man perfekte Software erstellt. Immer wieder finden Bugs Eingang in Produktivsysteme. Wenn eine Anwendung auf eine Datenbank zugreifen kann, können die Bugs das ebenfalls. Diese Erfahrung hat meine Arbeit im Hinblick auf die Neuentwicklung unserer Datenarchitektur stark beeinflusst. Mir war jetzt bewusst, dass die neue Architektur nicht nur skalierbar, fehlertolerant gegenüber Rechnerausfällen und gut verständlich sein sollte, sondern auch mit menschlichen Fehlern zurechtkommen musste.

Meine im Zuge der Neuentwicklung des Systems hinzugewonnenen Erkenntnisse führten letztendlich dazu, dass ich alles, was ich bis zu diesem Zeitpunkt über Datenbanken und Datenhaltung zu wissen geglaubt hatte, infrage stellte. Schließlich entwickelte ich eine Architektur, die auf unveränderlichen Daten und Stapelverarbeitung beruhte – und war erstaunt, um wie vieles einfacher sie im Vergleich zu solchen Architekturen war, die ausschließlich auf inkrementellen Berechnungen basierten. Alles wurde einfacher: die Ausführung der Operationen, die Weiterentwicklung des Systems zwecks Unterstützung neuer Features, das Rückgängigmachen menschlicher Fehlentscheidungen und sogar Geschwindigkeitsoptimierungen. Der Ansatz war so allgemein gehalten, dass er für jedes beliebige Datenhaltungssystem einsetzbar zu sein schien.

Eine Sache verwirrte mich allerdings: Es gab kaum jemanden in der Branche, der vergleichbare Verfahren einsetzte. Stattdessen wurden erschreckend komplizierte Architekturen verwendet, die auf riesigen Clustern inkrementell aktualisierter Datenbanken beruhten. Der von mir entwickelte Ansatz umging dagegen einen Großteil dieser Komplexität bzw. entschärfte sie weitgehend.

In den folgenden Jahren erweiterte und formalisierte ich dieses Konstrukt, das ich *Lambda-Architektur* getauft hatte. Unser fünfköpfiges Team arbeitete bei einem Startup namens BackType an einem Produkt zur Analyse der Daten sozialer Netzwerke, das in Echtzeit verschiedene Kennzahlen aus einem Datenbestand von mehr als 100 TB errechnen konnte. Darüber hinaus war unser kleines Team auch für das Deployment, den Betrieb und das Monitoring des Systems zuständig, das auf einem aus einigen Hundert Maschinen bestehenden Cluster lief. Wann immer wir das System vorführten, waren die Leute verblüfft, dass unser Team lediglich aus fünf Mitarbeitern bestand. Sie fragten dann häufig: »Wie können so wenige Leute so viel erreichen?« Darauf hatte ich eine einfache Antwort parat: »Das liegt nicht an dem, was wir tun, sondern an dem, was wir *nicht* tun.« Durch den Einsatz der Lambda-Architektur vermieden wir die komplexen Verflechtungen, mit denen sich herkömmliche Architekturen herumplagen müssen. Und in der Konsequenz bewirkte die Umgehung dieser Komplexität eine drastische Steigerung unserer Produktivität.

Der Trend zu Big Data hatte das Ausmaß der seit Jahrzehnten in den gebräuchlichen Datenarchitekturen vorhandenen komplexen Strukturen nur noch weiter verstärkt. Jede Architektur, die vornehmlich auf großen, inkrementell aktualisierten Datenbanken beruht, wird von dieser Komplexität in Mitleidenschaft gezogen: Sie führt zu Bugs, beschwerlichen Operationen und hemmt die Produktivität. SQL- und NoSQL-Datenbanken werden oft als gegensätzlich dargestellt, sind auf grundlegender Ebene aber eigentlich gleichartig. Sie begünstigen eine solche Architektur mit ihren zwangsläufig komplexen Strukturen. Auch wenn man es nicht wahrhaben will, bleibt die Komplexität doch ein bösartiges Ungetüm, das einem zu schaffen macht.

Dieses Buch ist meinem Wunsch entsprungen, das Wissen um die Lambda-Architektur zu verbreiten und aufzuzeigen, wie man mit ihrer Hilfe die Komplexität herkömmlicher Architekturen umgehen kann. Ein solches Nachschlagewerk hätte ich mir gewünscht, als ich anfing, mich mit Big Data zu befassen. Ich hoffe, Sie betrachten dieses Buch als eine Reise – eine Reise, auf der Sie Ihr Wissen über Datenhaltungssysteme hinterfragen und entdecken werden, dass die Arbeit mit Big Data nicht nur elegant und einfach sein kann, sondern sogar Spaß macht.

Nathan Marz

Danksagungen

Ohne die Hilfe und Unterstützung vieler Menschen rund um den Globus würde es dieses Buch nicht geben. Zuallererst möchte ich an dieser Stelle meinen Eltern danken, die mich von klein auf dazu anspornten, wissbegierig zu sein und meine Umwelt zu erkunden. Sie haben mich stets bei allen meinen Karrierebemühungen unterstützt.

Ebenso war mein Bruder Iorav schon früh um die Förderung meiner Interessen bemüht. Ich kann mich noch gut daran erinnern, wie er mir bereits in meiner Grundschulzeit Algebra beibrachte. Er war es auch, der mich erstmals mit der Programmierung bekanntmachte – indem er mich in Visual Basic unterwies, als er in der Oberstufe einen entsprechenden Kurs belegte. Diese Lehrstunden weckten meine Leidenschaft für die Programmierung, die zu meinem späteren Werdegang führte.

Zu außerordentlichem Dank verpflichtet bin ich auch Michael Montano und Christopher Golda, den Gründern von BackType. Von dem Moment an, als sie mich als ihren ersten Angestellten begrüßt hatten, gestanden sie mir ein keineswegs selbstverständliches Maß an Entscheidungsfreiheit zu, ohne das ich die Lambda-Architektur nicht ergründen und vollumfänglich hätte ausarbeiten können. Sie stellten den Nutzwert von Open-Source-Software zu keinem Zeitpunkt infrage und gestatteten mir, unsere Technologie unter einer Open-Source-Lizenz zu veröffentlichen. Ich betrachte es als eins der größten Privilegien in meinem Leben, an der Entwicklung dieses Open-Source-Projekts beteiligt zu sein.

Vielen der Professoren, denen ich als Student in Stanford begegnete, gebührt mein besonderer Dank. Tim Roughgarden ist der beste Lehrer, den ich jemals hatte – er sorgte für eine dramatische Verbesserung meiner Fähigkeit, schwierige Probleme gründlich zu analysieren, sie in Teilaufgaben zu zerlegen und schließlich auch erfolgreich zu lösen. Die Entscheidung, so viele seiner Vorlesungen wie nur möglich zu besuchen, gehörte zu den besten, die ich je getroffen habe. Des Weiteren möchte ich auch Monica Lam danken, die mich lehrte, die Eleganz von Datalog zu schätzen. Viele Jahre später kombinierte ich Datalog mit MapReduce zu Cascalog, meinem ersten nennenswerten Open-Source-Projekt.

Chris Wensel zeigte mir als Erster, dass die Verarbeitung großer Datenmengen sowohl elegant als auch schnell möglich ist. Seine Cascading-Bibliothek hat meine Denkweise in Bezug auf das Big-Data-Processing maßgeblich verändert.

Ohne die Pioniere auf dem Gebiet der Big Data wären meine Arbeiten nicht möglich gewesen. Mein besonderer Dank gilt Jeffrey Dean und Sanjay Ghemawat, den Autoren des ursprünglichen MapReduce-Artikels, Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall und Werner Vogels, den Autoren des ursprünglichen Dynamo-Artikels, sowie Michael Cafarella und Doug Cutting, den Gründern des Apache Hadoop-Projekts.

Rich Hickey hat mich im Laufe meiner Karriere als Entwickler stets inspiriert. Clojure ist die beste Programmiersprache, die ich je benutzt habe. Sie zu erlernen, hat mich zu einem besseren Programmierer gemacht. Ich weiß ihre Sachlichkeit und ihre Fokussierung auf Einfachheit sehr zu schätzen. Richs Philosophie hinsichtlich der Zustände und der Komplexität in der Programmierung hat mich stark beeinflusst.

Als ich mit der Arbeit an diesem Buch begann, war ich im Schreiben noch nicht annähernd so geübt, wie ich es jetzt bin. Renae Gregoire, einer meiner Lektorinnen bei Manning, gebührt besonderer Dank für ihre tatkräftige Unterstützung zur Verbesserung meiner Fähigkeiten als Autor. Sie schärfte mir ein, wie wichtig es ist, allgemeine Konzepte durch Beispiele zu veranschaulichen und dank ihr gingen mir in Bezug auf das Verfassen technischer Texte einige Lichter auf. Die Fertigkeiten, die sie mir beibrachte, finden aber nicht nur beim Schreiben technischer Inhalte Anwendung, sondern auch beim Bloggen, beim Halten von Vorträgen und bei der Kommunikation im Allgemeinen. Für die Vermittlung dieser unverzichtbaren Kompetenzen bin ich ihr auf ewig zu Dank verpflichtet.

Dieses Buch hätte ohne die Anstrengungen meines Koautors James Warren nicht annähernd seine jetzige Qualität erreicht. Er hat bei der Integration theoretischer Konzepte phänomenale Arbeit geleistet und sogar immer noch bessere Möglichkeiten gefunden, sie zu präsentieren. Ein Großteil der Klarheit des Buches ist seinen hevorragenden Kommunikationsfähigkeiten zu verdanken.

Die Zusammenarbeit mit dem Manning-Verlag war ein echtes Vergnügen. Man hatte Geduld mit mir und brachte Verständnis dafür auf, dass es Zeit braucht, die richtige Herangehensweise für die literarische Ausarbeitung eines so umfassenden Themas zu finden. Die Verlagsmitarbeiter waren stets freundlich und hilfreich und stellten mir alle zum Schreiben des Buches benötigten Ressourcen bereit. Danke an Marjan Bace und Michael Stephens für all die Unterstützung, und ich danke auch allen anderen Mitarbeitern für ihre Hilfe und Beratung.

Durch das Studium der Bücher anderer Autoren versuche ich, so viel wie möglich über die Kunst des Schreibens zu lernen. Bradford Cross, Clayton Christensen, Paul Graham, Carl Sagan und Derek Sivers waren diesbezüglich besonders einflussreich.

Und schließlich kann ich auch den Hunderten von Leuten, die das Buchmanuskript in der Entstehungsphase lektorierten, kommentierten und uns ihr Feedback lieferten, gar nicht genug danken. Auf dieser Grundlage war es uns möglich, das Buch mehrfach zu überarbeiten, umzuschreiben und neu zu strukturieren, bis wir eine Möglichkeit fanden, seinen Inhalt in angemessener Weise zu präsentieren. Mein besonderer Dank geht an Aaron Colcord, Aaron Crow, Alex Holmes, Arun Jacob, Asif Jan, Ayon Sinha, Bill Graham, Charles Brophy, David Beckwith, Derrick Burns, Douglas Duncan, Hugo Garza, Jason Courcoux, Jonathan Esterhazy, Karl Kuntz, Kevin Martin, Leo Polovets, Mark Fisher, Massimo Ilario, Michael Fogus, Michael G. Noll, Patrick Dennis, Pedro Ferrera Bertran, Philipp Janert, Rodrigo Abreu, Rudy Bonefas, Sam Ritchie, Siva Kalagarla, Soren Macbeth, Timothy Chklovski, Walid Farid und Zhenhua Guo.

NATHAN MARZ

Wenn ich darüber nachdenke, wie viele Menschen in irgendeiner Weise zu diesem Buch beigetragen haben, bin ich wirklich verblüfft. Leider ist es mir an dieser Stelle nicht möglich, sie allesamt einzeln aufzulisten, das ändert aber nichts an meiner Dankbarkeit. Gleichwohl gibt es jedoch einige Personen, denen ich meinen ausdrücklichen Dank aussprechen möchte:

- Meiner Frau Win-Yeng Feng – für deine Zuneigung, Ermutigung und Unterstützung, nicht nur hinsichtlich dieses Buches, sondern in Bezug auf alles, was wir gemeinsam tun.
- Meinen Eltern James und Gretta Warren – für euer unerschütterliches Vertrauen in mich und die Opfer, die ihr gebracht habt, um mir alle Möglichkeiten zu eröffnen.
- Meiner Schwester Julia Warren-Ulanch – du warst mir ein leuchtendes Beispiel, in dessen Fußstapfen ich treten konnte.
- Meinen Professoren und Mentoren Ellen Toby und Sue Geller – für ihre Bereitschaft, mir jede Frage zu beantworten und mir zu demonstrieren, dass es nicht nur Freude macht, Wissen zu erwerben, sondern auch, es zu teilen.
- Chuck Lam – für die vor vielen Jahren an mich gestellte Frage: »He, hast du von dieser neuen Sache namens Hadoop gehört?«
- Meinen Freunden und Kollegen bei RockYou!, Storm8 und Bina – für die gemeinsam gesammelten Erfahrungen und die Gelegenheit, die Theorie in die Praxis umzusetzen.
- Marjan Bace, Michael Stephens, Jennifer Stout, Renae Gregoire und der gesamten Manning-Belegschaft – für eure Unterstützung und Geduld während des Entstehungsprozesses dieses Buches.
- Den Rezensenten und ersten Lesern dieses Buches – für euer Feedback und eure Bewertungen, die uns anspornten, die richtigen Worte zu finden. In letzter Konsequenz hat das Buch in wesentlichem Umfang davon profitiert.

Und schließlich möchte ich auch Nathan meinen Dank dafür aussprechen, dass er mich einlud, ihn auf dieser Reise zu begleiten. Ich war schon vor meiner Beteiligung an diesem Projekt ein großer Bewunderer deiner Arbeit, und die Zusammenarbeit mit dir hat meinen Respekt für deine Ideen und deine Philosophie nur noch verstärkt. Es war mir eine Ehre und ein Vergnügen.

James Warren

Über dieses Buch

Dienste wie soziale Netzwerke, Web Analytics (Traffic-Analyse) und intelligenter E-Commerce haben es oft mit Datenmengen zu tun, die für herkömmliche Datenbanken schlicht zu umfangreich sind. Proportional zur Skalierung und Abfragedichte nimmt auch die Komplexität des Datenaufkommens zu. Für die Handhabung von Big Data reicht es nicht aus, einfach nur die Kapazität des relationalen Datenbankmanagementsystems (*Relational Database Management System*, RDBMS) zu verdoppeln oder irgendeine neue, gerade im Trend liegende Technologie einzusetzen. Glücklicherweise schließen sich Skalierbarkeit und Einfachheit dabei nicht gegenseitig aus – Sie müssen die Sache nur anders angehen. Big-Data-Systeme nutzen zum Speichern und Verarbeiten der Daten viele parallel arbeitende Maschinen gleichzeitig, und das bringt grundlegende Anforderungen mit sich, die den meisten Entwicklern nicht vertraut sind.

Dieses Buch zeigt Ihnen, wie Sie solche Systeme unter Verwendung einer Architektur einrichten können, die sich eine Clusterhardware zunutze macht und zugleich neuartige Tools verwendet, die speziell für die Aufzeichnung und Analyse großer Datenmengen ausgelegt sind. Es beschreibt einen skalierbaren, leicht verständlichen Ansatz für Big-Data-Systeme, die von einem kleinen Team eingerichtet und betrieben werden können. In den folgenden Kapiteln werden die theoretischen Grundlagen der Big-Data-Systeme anhand eines realistischen Beispiels erläutert, in dessen Verlauf auch eine entsprechende praktische Umsetzung erfolgt.

Big Data setzt keine Kenntnisse über Tools zur großmaßstäblichen Datenanalyse oder NoSQL voraus. Sollte Ihnen der Umgang mit herkömmlichen Datenbanken vertraut sein, ist das zwar durchaus hilfreich, aber nicht zwingend erforderlich. Ziel des Buches ist es, Ihnen eine neue Denkweise in Bezug auf Datenhaltungssysteme zu vermitteln und darzustellen, wie Sie schwierige Probleme in leichter lösbare Teilaufgaben unterteilen können. Wir fangen mit den Grundlagen an und leiten daraus dann die notwendigen Eigenschaften der verschiedenen Komponenten einer Architektur ab.

Aufbau des Buches

Es folgt eine Übersicht über die Inhalte der 18 Kapitel dieses Buches.

Kapitel 1 stellt zunächst die Grundlagen eines Datenhaltungssystems vor und bietet Ihnen einen Überblick über die Lambda-Architektur, die einen universellen Ansatz zur Einrichtung eines beliebigen Datenhaltungssystems repräsentiert. In den nachfolgenden Kapiteln 2 bis 17 werden dann die einzelnen Bestandteile dieser Architektur erkundet. Sie sind abwechselnd theoretischer und praktischer Natur: Während die *theoretischen* Kapitel die Konzepte beschreiben, die unabhängig von den eingesetzten Tools anwendbar sind, werden in den *Praxis*-Kapiteln die faktisch vorhandenen Tools benutzt, um die besagten

Konzepte in die Tat umzusetzen. Lassen Sie sich aber nicht von diesen Bezeichnungen beirren – denn tatsächlich enthalten alle Kapitel jede Menge Beispiele.

Die Kapitel 2 bis 9 konzentrieren sich auf den *Batch-Layer* der Lambda-Architektur. Hier erfahren Sie, wie ein Stammdatensatz (engl. *Master Dataset*) modelliert wird, wie Stapelverarbeitungsprozesse dazu verwendet werden, beliebige *Views* (Sichten) der Daten zu erzeugen und welche Kompromisse bei der inkrementellen bzw. stapelweisen Verarbeitung einzugehen sind.

Die Kapitel 10 und 11 beschäftigen sich mit dem *Serving-Layer*, der einen schnellen Zugriff auf die vom Batch-Layer erzeugten Views ermöglicht. Hier werden spezialisierte Datenbanken vorgestellt, denen Daten nur in großen Blöcken hinzugefügt werden. Sie werden feststellen, dass diese im Vergleich zu herkömmlichen Datenbanken sehr viel schlichter daherkommen und daher exzellente Eigenschaften in Bezug auf Geschwindigkeit, Betrieb und Belastbarkeit aufweisen.

Die Kapitel 12 bis 17 haben den *Speed-Layer* zum Thema, der die hohen Latenzzeiten des Batch-Layers kompensiert und aktuelle Ergebnisse für alle *Queries* (Anfragen) liefert. Außerdem werden die NoSQL-Datenbanken, die Streamverarbeitung sowie der Umgang mit der Komplexität inkrementeller Berechnungen eingehender betrachtet.

Kapitel 18 nutzt Ihr neu erworbenes Wissen, um die Lambda-Architektur noch einmal zu rekapitulieren und gegebenenfalls verbleibende Lücken zu schließen. Sie lernen die inkrementelle Stapelverarbeitung sowie einige Varianten der grundlegenden Lambda-Architektur kennen und erfahren, wie Sie Ihre Ressourcen bestmöglich einsetzen können.

Beispielcode und Konventionen

Die Quelltexte des Buches finden Sie unter `https://github.com/Big-Data-Manning`. Dort steht der Code des SuperWebAnalytics.com-Beispiels zum Herunterladen bereit.

Einige Listings sind mit Anmerkungen versehen, die bestimmte Teile des Codes besonders hervorheben oder erläutern sollen. Sie sind in dem jeweiligen Listing durch eine weiße Zahl in einem schwarzen Kreis gekennzeichnet und werden dann unterhalb des Listings aufgeführt. Gelegentlich finden sich auch im Fließtext Java-Codefragmente, die in einer `nicht-proportionalen` Schrift gedruckt sind. Wichtige Abschnitte oder Teile des Codes, auf die im Fließtext verwiesen wird, sind **`fett`** gedruckt.

Kapitel 1

Ein neues Paradigma für Big Data

In diesem Kapitel geht es um folgende Themen:

- Typische Probleme bei der Skalierung herkömmlicher Datenbanken
- NoSQL ist kein Allheilmittel
- Big-Data-Systeme: Grundlagen
- Verfügbare Big-Data-Tools
- Kurz vorgestellt: SuperWebAnalytics.com

Im vergangenen Jahrzehnt ist das allgemeine Datenaufkommen explosionsartig gestiegen. In *jeder einzelnen Sekunde* werden mehr als 30.000 Gigabyte neue Daten generiert – und die Erzeugungsrate nimmt weiter zu.

Und dabei geht es um die unterschiedlichsten Dinge: Anwender erstellen Inhalte wie Blogbeiträge, Tweets, Posts in sozialen Netzwerken oder Fotos. Und alle ihre Aktivitäten werden unaufhörlich von Servern protokolliert. Wissenschaftler nehmen detaillierte Messungen in und an unserer Umwelt vor. Das Internet, letztlich die entscheidende Datenquelle, ist nahezu unvorstellbar groß.

Dieser erstaunliche Anstieg an Datenvolumen hat tiefgreifende Auswirkungen auf die Geschäftswelt. Gängige Datenhaltungssysteme wie relationale Datenbanken sind ausgereizt, sie brechen in zunehmender Zahl unter der Last der »Big Data« zusammen. Die herkömmlichen Systeme und dazugehörigen Verfahren zur Datenhaltung sind ihnen einfach nicht gewachsen.

Um den mit Big Data einhergehenden Herausforderungen begegnen zu können, wurden verschiedene neue Technologien entwickelt. Viele davon sind unter dem Begriff *NoSQL* zusammengefasst. In mancher Hinsicht sind diese Technologien komplexer als herkömmliche Datenbanken, in anderer Hinsicht fallen sie hingegen einfacher aus. Derartige Systeme sind für weitaus größere Datenmengen als üblich geeignet, allerdings sind für die wirklich effiziente Nutzung dieser Technologien auch grundlegend neue Vorgehensweisen erforderlich, denn es handelt sich hierbei nicht um Standardlösungen, die für Datenvolumen jeglicher Größe zu gebrauchen sind.

Bei vielen Big-Data-Systemen hat Google Pionierarbeit geleistet. Dazu gehören etwa verteilte Dateisysteme, das MapReduce-Framework für parallele Berechnungen und Locking Services für verteilte Systeme wie Chubby. Ein weiterer erwähnenswerter Pionier auf diesem Gebiet ist auch Amazon. Dort wurde eine innovative Schlüssel-Werte-Datenbank (engl. *Key/Value Store*) für verteilte Systeme namens Dynamo entwickelt. Die Open-Source-Community brachte daraufhin in den folgenden Jahren Projekte wie Hadoop, HBase, MongoDB, Cassandra, RabbitMQ und viele andere hervor.

In diesem Buch geht es nicht nur um Skalierbarkeit, sondern auch um Komplexität. Um die mit Big Data verbundenen Herausforderungen anzunehmen, müssen wir das Konzept der Datenhaltungssysteme von Grund auf neu überdenken. Sie werden feststellen, dass einige der grundlegenden Methoden, die bei der Verwaltung herkömmlicher Systeme wie RDBMS Anwendung finden, für Big-Data-Systeme zu komplex sind. Ein einfacherer alternativer Ansatz ist das neue Paradigma für Big Data, das Sie in den nachfolgenden Kapiteln ergründen werden. Wir haben diesen Ansatz *Lambda-Architektur* getauft.

In diesem ersten Kapitel erfahren Sie, worum es beim »Big-Data-Problem« eigentlich geht und warum für Big Data ein neues Paradigma erforderlich ist. Sie werden einige der in traditionellen Skalierungsverfahren lauernden Gefahren kennenlernen sowie verschiedene tiefgreifende Schwachstellen bei der Errichtung herkömmlicher Datenhaltungssysteme aufdecken. Ausgehend von den Grundlagen solcher Systeme werden wir gemeinsam eine neue Vorgehensweise für deren Einrichtung erarbeiten, die auf die Komplexität der bislang üblicherweise verwendeten Techniken verzichtet. Sie werden sehen, inwiefern jüngste Technologietrends den Einsatz neuer Systeme begünstigen und schließlich ein Beispiel für ein Big-Data-System betrachten, das wir zur Veranschaulichung der entscheidenden Konzepte im weiteren Verlauf des Buches entwickeln werden.

1.1 Aufbau des Buches

Sie sollten das vorliegende Buch in erster Linie als ein theoretisches Werk betrachten, das sich mit der Methodik zur Erarbeitung von Lösungen für Big-Data-Probleme befasst. Die Grundregeln, die Ihnen hier vermittelt werden, finden unabhängig von den verfügbaren Toolsammlungen Anwendung und gestatten somit eine individuelle Auswahl der für Ihren Anwendungsfall geeigneten Tools.

Dieses Buch ist nicht dazu gedacht, Ihnen Datenbanken, Berechnungsverfahren oder vergleichbare Technologien zu erklären. Sie werden zwar erfahren, wie Sie sich Tools wie Hadoop, Cassandra, Storm und Thrift zunutze machen können, allerdings geht es dabei nicht um deren Einsatz um ihrer selbst willen. Vielmehr dienen diese Tools als Mittel zum Zweck, um Ihnen die grundlegenden Prinzipien für die Einrichtung belastbarer und skalierbarer Datenhaltungssysteme näherzubringen. Mit einer ausführlichen Gegenüberstellung der verschiedenen Tools wäre Ihnen hier nicht geholfen, weil dies nur vom Verständnis der grundlegenden Prinzipien ablenken würde. Bildlich gesprochen könnte man sagen, Sie werden lernen, wie man angelt – und nicht bloß, wie man eine ganz bestimmte Angelrute benutzt.

Wir haben das Buch in *theoretische* und *praktische* Kapitel unterteilt. Die alleinige Lektüre der theoretischen Kapitel sollte ausreichen, um Sie mit umfassendem Wissen in Bezug auf die Errichtung von Big-Data-Systemen auszustatten – allerdings sind wir der Ansicht, dass Ihnen die Zuordnung der Theorie zu bestimmten Tools in den Praxis-Kapiteln ein besseres, differenzierteres Verständnis der präsentierten Sachverhalte ermöglicht.

Lassen Sie sich jedoch nicht von diesen Bezeichnungen täuschen: Auch der theoretische Teil enthält viele praktische Anwendungen. Das kapitelübergreifende Beispiel – SuperWebAnalytics.com – wird sowohl im theoretischen als auch im praktischen Teil dieses Buches verwendet. Die theoretischen Kapitel haben Themen wie Algorithmen, Indizierung und Architektur zum Inhalt und in den Praxis-Kapiteln werden diese Entwürfe dann mittels bestimmter Tools in funktionierenden Code umgesetzt.

1.2 Skalierung mit einer herkömmlichen Datenbank

Fangen wir mit der Erkundung von Big Data an dem Punkt an, an dem auch viele Entwickler zum ersten Mal mit dem Thema in Berührung kommen: Die herkömmliche Datenbanktechnologie stößt an ihre Grenzen.

Stellen Sie sich vor, Ihr Chef bittet Sie, eine kleine Web-Analytics-Anwendung zu programmieren. Sie soll die Anzahl der Pageviews einer beliebigen vom Kunden angegebenen URL aufzeichnen. Bei jedem Pageview übergibt die Webseite des Kunden ihre URL an den Webserver, auf dem die Anwendung läuft. Außerdem soll die Anwendung jederzeit die hundert am häufigsten aufgerufenen URLs zurückliefern – und zwar nach Häufigkeit sortiert.

Sie verwenden für die Pageviews ein übliches relationales Datenbankschema wie in Abbildung 1.1. Ihr Back-End besteht aus einem RDBMS mit einer Tabelle dieses Schemas und einem Webserver. Sobald jemand eine von Ihrer Anwendung überwachte Webseite lädt, übermittelt die Webseite ihre URL an den Webserver und die Anwendung erhöht den Zähler in der zugehörigen Zeile der Datenbank.

Sehen wir uns nun an, welche Schwierigkeiten bei der Weiterentwicklung der Anwendung auftreten. Sie werden gleich feststellen, dass sowohl die Skalierbarkeit als auch die Komplexität Probleme bereiten.

Spaltenname	Typ
id	integer
user_id	integer
url	varchar(255)
pageviews	bigint

Abb. 1.1: Relationales Schema einer einfachen Anwendung für Web Analytics

1.2.1 Skalierung mit einer Warteschlange

Ihre Web-Analytics-Anwendung ist ein voller Erfolg und der darüber abgewickelte Datentraffic nimmt rasant zu. Aus diesem Anlass schmeißt Ihr Unternehmen eine große Party, doch plötzlich kommen noch während der Feier jede Menge E-Mails von Ihrem Monitoring-System herein, die alle dasselbe besagen: »Zeitüberschreitungsfehler beim Einfügen in die Datenbank.«

Also werfen Sie einen Blick in das Protokoll und erkennen auch sogleich die offenkundige Ursache des Problems: Die Datenbank kann nicht mit der Arbeitslast Schritt halten, daher kommt es bei den Requests (Anfragen) zur Erhöhung des Pageview-Zählers zu einer Zeitüberschreitung.

Sie müssen das Problem irgendwie beheben – und zwar schnell. Ihnen fällt auf, dass es unwirtschaftlich ist, den Zähler der Datenbank jeweils nur um eins zu erhöhen. Viel effizienter wäre es, wenn Sie mehrere Erhöhungen sammeln und diese dann in einem einzigen Request zusammenfassen würden. Also passen Sie Ihr Back-End entsprechend an, um dies zu ermöglichen.

Der Webserver richtet seine Requests nun nicht mehr direkt an die Datenbank, sondern an eine Warteschlange zwischen Webserver und Datenbank. Wenn jetzt ein Pageview stattfindet, wird dieses Ereignis in die Warteschlange eingereiht. Danach erstellen Sie einen Worker-Prozess, der jeweils 100 Ereignisse aus der Warteschlange entnimmt und sie zu einer einzelnen Datenbankaktualisierung zusammenfasst. Diese Stapelverarbeitung ist in Abbildung 1.2 illustriert.

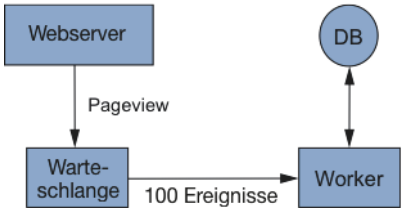

Abb. 1.2: Stapelverarbeitung mit Warteschlange und Worker

1.2.2 Skalierung durch Sharding

Leider stellt sich schon bald heraus, dass die Stapelverarbeitung der Aktualisierungen nur eine Übergangslösung für die Skalierungsprobleme darstellt: Ihre Anwendung wird immer beliebter und die Datenbank ist ruck zuck wieder überlastet. Ihr Worker-Prozess kann nicht mit den Schreibvorgängen Schritt halten, deshalb versuchen Sie weitere Worker hinzuzufügen. Bedauerlicherweise schafft das jedoch keine Abhilfe – der Engpass ist zweifelsohne die Datenbank.

Als Nächstes recherchieren Sie im Internet, wie man relationale Datenbanken skaliert, auf die häufig schreibend zugegriffen wird. Sie finden heraus, dass der beste Ansatz wohl darin besteht, mehrere Datenbankserver zu verwenden und die Tabelle auf alle Server aufzuteilen. Dabei speichert jeder Server eine Untermenge der Daten in der Tabelle. Man bezeichnet dieses Verfahren als *horizontale Partitionierung* oder *Sharding*. Die Arbeitslast beim Schreiben wird hierbei auf mehrere Maschinen verteilt.

Beim Sharding verwenden Sie einen Hashwert, der als Datensatzschlüssel modulo der Anzahl der Datenbanken berechnet wird, um die Daten auf die verschiedenen Server aufzuteilen. Die Datensätze mittels einer Hashfunktion den Datenbanken zuzuordnen, sorgt für eine gleichmäßige Zuweisung der Daten zu den Servern. Dann schreiben Sie ein Skript, das die Datensätze der vorhandenen Datenbankinstanz auf vier neue Datenbanken verteilt. Das dauert eine Weile, also deaktivieren Sie so lange die Worker, damit während der Umverteilung keine Pageviews verloren gehen.

Außerdem müssen Sie den Code Ihrer Anwendung anpassen, damit er anhand des Datensatzschlüssels die zugehörige Datenbank ermitteln kann. Also schreiben Sie eine kleine Library (Bibliothek) als Schnittstelle zu Ihrem Datenbankcode, die die Anzahl der Datenbanken aus einer Konfigurationsdatei einliest, und stellen Ihren Anwendungscode erneut bereit. Die Abfrage der hundert am häufigsten aufgerufenen URLs müssen Sie ebenfalls modifizieren: Die vier Datenbanken müssen einzeln abgefragt und die Ergebnisse zusammengeführt werden, um so die globale Liste der Top-100-URLs zu erhalten.

Während die Anwendung weiterhin an Popularität gewinnt, müssen Sie die Daten ständig auf zusätzliche Datenbanken aufteilen, um mit der steigenden Arbeitslast Schritt halten zu

können. Das Ganze wird mit jedem Mal mühsamer, weil immer mehr Arbeitsschritte koordiniert werden müssen. Sie können auch nicht einfach nur ein Skript ablaufen lassen, das die anfallenden Arbeiten erledigt, weil das viel zu lange dauern würde. Die Neuaufteilung der Daten und die Verwaltung der Worker-Prozesse muss gleichzeitig stattfinden. Und zu allem Überfluss vergessen Sie nun auch noch, die Anzahl der Datenbanken im Code Ihrer Anwendung zu aktualisieren – was zur Folge hat, dass zuhauf Pageview-Erhöhungen in einer falschen Datenbank gespeichert werden. Sie müssen also einmalig ein Skript laufen lassen, das die fehlerhaft platzierten Daten wieder entfernt.

1.2.3 Erste Probleme mit der Fehlertoleranz

Früher oder später sind so viele Datenbanken im Einsatz, dass es bei einem der Datenbankserver regelmäßig zu einem Festplattenausfall kommt. Und während der Rechner nicht verfügbar ist, kann natürlich auch nicht auf den darauf gespeicherten Teil des Datenbestands zugegriffen werden. Sie unternehmen nun folgende Schritte, um sich des Problems anzunehmen:

- Sie aktualisieren Ihr Warteschlangen/Worker-System. Pageview-Zählererhöhungen für nicht verfügbare Datenbanken werden in einer separaten Warteschlange gesammelt, die Sie alle fünf Minuten zu leeren versuchen.
- Sie nutzen die Fähigkeit der Datenbank, Replikationen durchzuführen und fügen zu jeder Datenbank einen Slave hinzu, der einspringen kann, sobald der Master ausfällt. Die Slave-Datenbank ist zwar nicht beschreibbar, aber die Kunden können sich in Ihrer Anwendung wenigstens ihre Statistiken anzeigen lassen.

Mittlerweile denken Sie sich: »Früher war ich damit beschäftigt, neue Features für die Kunden zu entwickeln. Jetzt kümmere ich mich nur noch um Probleme beim Lesen und Schreiben der Daten.«

1.2.4 Probleme mit fehlerhaften Daten

Während Sie am Warteschlangen-/Worker-Code arbeiten, bringen Sie versehentlich einen Bug in die Produktivversion der Anwendung ein, der den Zähler einer URL bei jedem Pageview nicht um eins, sondern um zwei erhöht. Allerdings bemerken Sie dies erst 24 Stunden später – und nun ist das Kind schon in den Brunnen gefallen: Die wöchentlichen Backups sind nutzlos, weil Sie nicht wissen können, welche Datensätze fehlerhaft sind. Nach all den Bemühungen, das System skalierbar und fehlertolerant zu machen, ist es gegen menschliches Versagen völlig ungeschützt. Und eins steht fest: Bugs gelangen zwangsläufig in die Produktivumgebung, auch wenn Sie sich die größte Mühe geben, das zu verhindern.

1.2.5 Was ist schiefgegangen?

Im Verlauf der Weiterentwicklung Ihrer Web-Analytics-Anwendung wurde das System allmählich immer komplexer: Warteschlangen, zusätzliche Datenbanken, Skripte zur Umverteilung der Daten usw. Für die Entwicklung einer Anwendung, die mit Daten arbeitet, ist weitaus mehr erforderlich als nur die Kenntnis des Datenbankschemas. Ihr Code muss ermitteln können, auf welche der Datenbanken er zugreifen muss, und falls Ihnen ein Fehler unterläuft, werden Sie nicht daran gehindert, lesend oder schreibend auf die falsche Datenbank zuzugreifen.

Ein Problem liegt darin begründet, dass der Datenbank nicht »bewusst« ist, lediglich eine Komponente eines verteilten Systems zu sein und sie Ihnen daher keine Unterstützung beim Umgang mit anderen Teildatenbanken, bei der Replikation oder bei verteilten Abfragen bieten kann. Die gesamte zusätzliche Komplexität wird somit Ihnen aufgebürdet, sowohl beim Betrieb der Datenbanken als auch bei der Entwicklung des Codes.

Am schlimmsten ist jedoch, dass solch ein System nicht dafür ausgelegt ist, mit menschlichem Fehlverhalten zurechtzukommen. Ganz im Gegenteil: Weil das System an sich immer komplexer wird, steigt auch die Wahrscheinlichkeit, dass Fehler begangen werden. Abgesehen davon lassen sich Softwarefehler ebenfalls nicht vermeiden – und wenn Sie nicht darauf vorbereitet sind, könnten Sie genauso gut gleich Skripte schreiben, die zufällig ausgewählte Daten zerschießen. Backups allein reichen hier nicht aus, vielmehr muss das System sorgfältig darauf ausgerichtet werden, menschliches Fehlverhalten abzufangen. Die Fehlertoleranz in Bezug auf das menschliche Versagen ist demzufolge keine optionale Angelegenheit, sondern eine unabdingbare Notwendigkeit – insbesondere dann, wenn Big Data die Entwicklung von Anwendungen stark verkomplizieren.

1.2.6 Inwiefern sind Big-Data-Verfahren hilfreich?

Die Big-Data-Verfahren, die Sie im Folgenden kennenlernen werden, lösen die bei der Skalierung auftretenden und von der Komplexität verursachten Probleme auf drastische Weise. Zunächst einmal ist den bei Big Data eingesetzten Datenbanken und Rechensystemen bekannt, dass sie Komponenten eines verteilten Systems sind – um Dinge wie Sharding und Replikation brauchen Sie sich also nicht selbst zu kümmern. Sie werden niemals in eine Situation geraten, in der versehentlich eine falsche Datenbankinstanz abgefragt wird, denn diese Logik ist Bestandteil der Datenbank selbst. Was die Skalierung betrifft, brauchen Sie einfach nur neue Netzknoten hinzuzufügen – das System erledigt die Umverteilung der Daten dann automatisch.

Eine weitere Vorgehensweise, die Sie kennenlernen werden, betrifft das Konzept der *unveränderlichen Daten* (engl. *immutable data*): Statt die Anzahl der Pageviews in einem Datensatz zu speichern, der jedes Mal modifiziert wird, sobald weitere Zugriffe stattfinden, werden die Informationen über den Pageview selbst gespeichert. Diese Informationen werden zu keinem Zeitpunkt geändert – und wenn Ihnen ein Fehler unterläuft, mögen Sie vielleicht fehlerhafte Daten ergänzen, die korrekten Daten bleiben aber dennoch intakt. Diese Art der Fehlertoleranz gegenüber menschlichem Versagen ist erheblich leistungsfähiger als diejenige traditioneller, auf inkrementellen Modifizierungen beruhender Systeme. Bei herkömmlichen Datenbanksystemen hütet man sich davor, unveränderliche Daten zu nutzen, weil die Größe der Datenbank dann sehr schnell ansteigt. Da die Big-Data-Verfahren jedoch leicht mit so umfangreichen Datenmengen umgehen können, eröffnet sich hier die Möglichkeit, die entsprechenden Systeme völlig anders zu konzipieren.

1.3 NoSQL ist kein Allheilmittel

Im vergangenen Jahrzehnt wurden bei skalierbaren Datenhaltungssystemen große Fortschritte erzielt. Dazu gehören etwa großmaßstäbliche Berechnungssysteme wie Hadoop oder auch Datenbanken wie Cassandra und Riak. Systeme dieser Art können enorme Datenmengen handhaben – sie gehen allerdings auch gravierende Kompromisse ein.

Hadoop kann beispielsweise großmaßstäbliche Stapelverarbeitungsberechnungen mit sehr großen Datenmengen ausführen, die jedoch vergleichsweise langsam erfolgen. Damit ist dieses System ungeeignet, wenn Sie auf schnelle Resultate angewiesen sind.

NoSQL-Datenbanken wie Cassandra erzielen ihre Skalierbarkeit dadurch, dass sie ein sehr viel eingeschränkteres Datenmodell bieten, als Sie es beispielsweise von SQL gewohnt sind. Eine Anwendung in derartig begrenzte Datenmodelle einzupferchen, kann ziemlich kompliziert werden. Außerdem können diese Datenbanken modifiziert werden, daher sind sie bei menschlichem Versagen nicht fehlertolerant.

Für sich allein genommen sind diese Tools also kein Allheilmittel. Wenn Sie sie jedoch intelligent miteinander verknüpfen, können Sie skalierbare Systeme minimaler Komplexität für beliebige Inhalte erstellen, die bei menschlichem Versagen fehlertolerant reagieren. Und genau darum geht es bei der Lambda-Architektur, die in diesem Buch vorgestellt wird.

1.4 Grundlagen

Um herauszufinden, wie Datenhaltungssysteme korrekt eingerichtet werden, muss man zunächst die Grundlagen berücksichtigen. Was also muss ein Datenhaltungssystem auf fundamentaler Ebene leisten?

Betrachten wir zunächst eine intuitive Definition: *Ein Datenhaltungssystem liefert anhand in der Vergangenheit bis zum jetzigen Zeitpunkt gesammelter Informationen Antworten auf Fragen.* Das Benutzerprofil eines sozialen Netzwerks beantwortet also Fragen wie beispielsweise »Wie lautet der Name dieser Person?« und »Wie viele Freunde hat diese Person?« Und eine Webseite liefert beim Onlinebanking die Antwort auf Fragen wie »Welche Überweisungen wurden kürzlich ausgeführt?« und »Wie ist mein aktueller Kontostand?«

Datenhaltungssysteme speichern aber nicht nur einfach Informationen und geben sie wieder – sie verknüpfen verschiedene Informationen miteinander, um eine Antwort zu liefern. Der Saldo eines Bankkontos beispielweise wird anhand aller durchgeführten Transaktionen ermittelt.

Auch dass nicht alle Informationen gleichwertig sind, ist eine wichtige Beobachtung. Manche von ihnen lassen sich aus anderen ableiten. Der Saldo eines Bankkontos etwa wird durch den Verlauf der vorgenommenen Transaktionen bestimmt. Die Anzahl der Freunde wird anhand einer Freundesliste ermittelt, die wiederum dadurch entsteht, dass ein Benutzer seinem Profil Freunde hinzufügt (oder daraus entfernt).

Wenn Sie immer weiter zurückverfolgen, woher die Informationen stammen, stoßen Sie früher oder später auf solche, die nicht von anderen abgeleitet wurden. Dies sind die eigentlichen »Rohinformationen«: Informationen, die Sie für wahr halten, schlicht und einfach weil sie vorhanden sind. Diese Informationen bezeichnen wir als *Daten*.

Möglicherweise haben Sie eine andere Vorstellung davon, was der Begriff *Daten* bedeutet. Die Bezeichnungen *Daten* und *Informationen* werden oft synonym gebraucht. Im Rahmen dieses Buches sind mit *Daten* jedoch generell diese speziellen grundlegenden Informationen gemeint, von denen alles andere abgeleitet werden kann.

Wenn ein Datenhaltungssystem Fragen beantworten kann, indem es auf in der Vergangenheit gesammelte Daten zurückgreift, dann wird ein möglichst allgemeingültiges Datenhal-

tungssystem auf die *Gesamtheit* der Daten zurückgreifen. Die allgemeingültigste Definition für ein Datenhaltungssystem lautet daher folgendermaßen:

```
Abfrage = Funktion(Sämtliche Daten)
```

Was immer Sie sich auch vorstellen können, grundsätzlich mit Daten anzufangen, kann als Funktion formuliert werden, die alle verfügbaren Daten als Eingabe entgegennimmt. Merken Sie sich diese Formel, denn sie ist hier der Dreh- und Angelpunkt. Wir werden im weiteren Verlauf des Buches immer wieder auf diese Gleichung verweisen.

Die Lambda-Architektur stellt einen allgemeinen Ansatz zur Implementierung einer beliebigen Funktion dar, die auf beliebige Daten zugreift und mit geringer Latenzzeit Ergebnisse liefert. Das soll allerdings nicht heißen, dass Sie bei jeder Implementierung eines Datenhaltungssystems immer genau dieselben Technologien verwenden werden. Die tatsächlich eingesetzten Technologien hängen selbstverständlich von Ihren Anforderungen ab. Die Lambda-Architektur legt jedoch eine konsistente Vorgehensweise bei der Auswahl der Technologien und deren Verknüpfung miteinander fest, damit sie Ihren Anforderungen gerecht wird.

Werfen wir nun einen Blick auf die Eigenschaften, die ein Datenhaltungssystem aufweisen muss.

1.5 Erwünschte Eigenschaften eines Big-Data-Systems

Bei den angestrebten Eigenschaften von Big-Data-Systemen geht es sowohl um die Komplexität als auch um die Skalierbarkeit. Ein Big-Data-System muss nicht nur schnell und effizient sein, sondern auch leicht verständlich. Sehen wir uns die verschiedenen Eigenschaften einmal der Reihe nach an.

1.5.1 Belastbarkeit und Fehlertoleranz

Angesichts der mit verteilten Systemen einhergehenden Herausforderungen ist es schwierig, Systeme einzurichten, die »das Richtige tun«. Sie müssen sich trotz all der Schwierigkeiten wie gelegentlich ausfallenden Maschinen, der komplizierten Semantik der Konsistenz verteilter Datenbanken, Datendoppelungen, paralleler Ausführung usw. korrekt verhalten. Diese Herausforderungen erschweren auch das Verständnis der Funktionsweise des Systems. Um die Belastbarkeit eines Big-Data-Systems zu erhöhen und es verständlicher zu machen, sollten diese Probleme daher möglichst umgangen werden.

Wie bereits erwähnt, ist die Fehlertoleranz gegenüber menschlichem Versagen unverzichtbar. Hierbei handelt es sich um eine häufig vernachlässigte Systemeigenschaft, die wir in diesem Buch allerdings keinesfalls außer Acht lassen werden. Es ist unvermeidlich, dass irgendwem irgendwann ein Fehler unterläuft, der Einfluss auf das Produktivsystem hat, beispielsweise das Deployment von Code, der zu fehlerhaften Werten in der Datenbank führt. Wenn ein Big-Data-System jedoch auf unveränderlichen Daten und der Neuberechnung der Werte beruht, ist das System von Haus aus vor menschlichem Versagen geschützt, weil es einen übersichtlichen und einfachen Mechanismus zur Wiederherstellung der Daten gibt. Wie das genau funktioniert, ist in den Kapiteln 2 bis 7 ausführlich beschrieben.

1.5.2 Lesen und Aktualisieren mit geringen Latenzzeiten

Bei einem Großteil der Anwendungen sind bei Lesevorgängen geringe Latenzzeiten erforderlich, die typischerweise zwischen einigen Millisekunden und einigen Hundert Millisekunden liegen. Die Anforderungen an die Latenzzeiten beim Aktualisieren variieren je nach Anwendung erheblich: In einigen Fällen müssen Aktualisierungen unverzüglich ausgeführt werden, in anderen reichen Latenzzeiten von mehreren Stunden aus. Dessen ungeachtet müssen Sie in der Lage sein, in Ihrem Big-Data-System geringe Latenzzeiten zu erzielen, *wenn sie benötigt werden*. Und noch wichtiger ist, dass Sie beim Lesen und Aktualisieren geringe Latenzzeiten erreichen müssen, ohne die Belastbarkeit des Systems zu gefährden. Wie Sie geringe Latenzzeiten bei Aktualisierungen implementieren können, erfahren Sie bei der Besprechung des Speed-Layers, die in Kapitel 12 beginnt.

1.5.3 Skalierbarkeit

Die Skalierbarkeit ist die Fähigkeit, die Geschwindigkeit bei wachsenden Datenmengen oder steigender Arbeitslast dadurch aufrechtzuerhalten, dass dem System weitere Ressourcen bereitgestellt werden. Bei der Lambda-Architektur sind alle drei Layer horizontal skalierbar. Die Skalierung wird durch das Hinzufügen weiterer Maschinen erreicht.

1.5.4 Allgemeingültigkeit

Ein allgemein verwendbares System unterstützt ein breites Spektrum von Anwendungen. Tatsächlich wäre dieses Buch nicht besonders nützlich, wenn es die vorgestellten Verfahren nicht verallgemeinern würde! Die Lambda-Architektur beruht auf Funktionen, die von der Gesamtheit der Daten abhängen, daher ist sie ganz allgemein anwendbar – sei es nun für Systeme zur Verwaltung von Finanzen, zur Analyse sozialer Medien und Netzwerke, für wissenschaftliche Anwendungen oder irgendetwas anderes.

1.5.5 Erweiterbarkeit

Sie möchten natürlich nicht jedes Mal das Rad neu erfinden müssen, wenn Sie Ihrem System ein neues Feature hinzufügen oder dessen Funktionsweise ändern wollen. Erweiterbare Systeme gestatten die Ergänzung zusätzlicher Funktionalität bei minimalem Entwicklungsaufwand.

Das Einbringen neuer Features oder auch Änderungen an vorhandenen Funktionen machen es oft erforderlich, alte Daten in ein neues Format zu konvertieren. Bei einem gut erweiterbaren System fällt es leicht, großmaßstäbliche Datenmigrationen vorzunehmen. Die Möglichkeit, solche umfänglichen Migrationen schnell und einfach durchzuführen, ist ein wesentlicher Bestandteil des vorgestellten Ansatzes.

1.5.6 Ad-hoc-Abfragen

Ihre Daten gezielt abfragen zu können, ist äußerst wichtig. Fast jeder größere Datensatz birgt unvermutete Werte. Einen Datensatz auf beliebige Weise durchforsten zu können, eröffnet Möglichkeiten zur Optimierung von Geschäftsvorgängen und kann zu neuen Anwendungen führen. Letzten Endes werden Sie kaum entdecken, welche interessanten Dinge Sie mit Ihren Daten anstellen könnten, wenn Sie nicht in der Lage sind, beliebige

Abfragen zu stellen. Wie Sie Ad-hoc-Abfragen ausführen können, erfahren Sie in den Kapiteln 6 und 7, in denen die Stapelverarbeitung erläutert wird.

1.5.7 Minimaler Wartungsaufwand

Die Wartung, also die Arbeit, die für den reibungslosen Betrieb eines Systems geleistet werden muss, ist für Entwickler eine Last. Sie müssen vorhersehen, wann das System zwecks Skalierung um neue Maschinen erweitert werden muss, dafür Sorge tragen, dass alle Prozesse ordnungsgemäß arbeiten und in der Produktivumgebung auftretende Fehler beheben.

Um den Wartungsaufwand zu minimieren, sollten Komponenten mit möglichst unkomplizierter Implementierung gewählt werden, also Bestandteile, denen einfache Mechanismen zugrunde liegen. Insbesondere die Interna verteilter Datenbanken sind tendenziell äußerst kompliziert. Je komplexer ein System aufgebaut ist, desto höher ist jedoch auch die Wahrscheinlichkeit, dass etwas schiefgeht und umso wichtiger ist es, dass Sie die Arbeitsweise des Systems verstehen, um Fehler zu beheben und Einstellungen vorzunehmen.

Sie können der Komplexität entgegenwirken, indem Sie sich simpler Algorithmen und einfacher Komponenten bedienen. Bei der Lambda-Architektur wird gern der Trick angewendet, die Komplexität aus den Kernkomponenten auszulagern und in Teile des Systems zu verschieben, deren Ausgaben nach einigen Stunden verworfen werden können. Die kompliziertesten eingesetzten Komponenten, wie etwa verteilte Datenbanken, befinden sich in einem solchen Layer, dessen Ausgaben nach einiger Zeit ausgesondert werden. Bei der Besprechung des Speed-Layers in Kapitel 12 werden wir näher auf diese Thematik eingehen.

1.5.8 Fehlerbehebung

Ein Big-Data-System muss die zur Fehlerbehebung erforderlichen Informationen zur Verfügung stellen, falls etwas schiefgeht. Entscheidend ist, dass man bei jedem einzelnen Wert nachverfolgen kann, wie er zustande gekommen ist.

Bei der Lambda-Architektur wird das zum einen durch die funktionale Natur des Batch-Layers erreicht und zum anderen dadurch, dass vorzugsweise Algorithmen zur Neuberechnung der Werte verwendet werden, sofern das möglich ist.

All diese Eigenschaften in einem System zu vereinen, mag auf den ersten Blick wie eine einschüchternde Herausforderung erscheinen. Wenn man jedoch, wie bei der Lambda-Architektur, von den Grundlagen ausgeht, ergeben sich diese Eigenschaften durch das resultierende Systemdesign von ganz allein.

Bevor wir uns ausführlicher mit der Lambda-Architektur befassen, wollen wir noch eine der herkömmlichen Architekturen, die durch Abhängigkeiten oder inkrementelle Berechnungen gekennzeichnet sind, genauer betrachten und erörtern, warum sie vielen der erwünschten Anforderungen nicht gerecht werden.

1.6 Schwierigkeiten vollständig inkrementeller Architekturen

Auf dem obersten Layer sieht eine herkömmliche Architektur wie in Abbildung 1.3 dargestellt aus. Zu den Merkmalen solch einer Architektur gehört es, dass lesend und schreibend

auf eine Datenbank zugegriffen wird und dass die Datenbank inkrementell aktualisiert wird, sobald neue Daten ergänzt werden. Bei einem inkrementellen Ansatz zum Zählen der Pageviews würde beispielsweise ein neu hinzugekommener Pageview dadurch verarbeitet werden, dass der Zähler seiner URL um eins erhöht wird. Dieses Architekturmerkmal ist von viel grundlegenderer Bedeutung als der Unterschied zwischen relationalen und nicht-relationalen Datenbanken – tatsächlich ist der Großteil beider Datenbankvarianten in Form einer vollständig inkrementellen Architektur implementiert. So verhält es sich schon seit Jahrzehnten.

Abb. 1.3: Vollständig inkrementelle Architektur

In diesem Zusammenhang muss betont werden, dass vollständig inkrementelle Architekturen so allgegenwärtig sind, dass vielen Leuten die durchaus bestehende Möglichkeit, deren Problemstellungen durch eine andere Architektur zu umgehen, gar nicht bewusst ist. Hierbei handelt es sich um ein schönes Beispiel für die sogenannte *vertraute Komplexität* – eine Komplexität, die so tief verwurzelt ist, dass man gar nicht darüber nachdenkt, wie man sie umgehen könnte.

Die bei vollständig inkrementellen Architekturen auftretenden Schwierigkeiten sind nicht unerheblich. Wir fangen bei der Erörterung dieses Themas zunächst mit den allgemeinen Komplexitäten an, die mit jeder Architektur dieser Art einhergehen. Anschließend werden wir zwei gegensätzliche Lösungen desselben Problems erkunden: Die eine nutzt die bestmögliche inkrementelle Architektur, die andere beruht auf der Lambda-Architektur. Sie werden sehen, dass die vollständig inkrementelle Version in jeder Hinsicht deutlich unterlegen ist.

1.6.1 Komplexität im Betrieb

Viele der Komplexitäten vollständig inkrementeller Architekturen sorgen beim Betrieb einer Produktivumgebung für Schwierigkeiten. Wir konzentrieren uns hier allerdings nur auf eine davon, nämlich die Notwendigkeit, eine sowohl lesbare als auch beschreibbare Datenbank im laufenden Betrieb zu komprimieren sowie die für die Durchführung dieses Vorgangs erforderlichen Maßnahmen.

Bei einer les- und beschreibbaren Datenbank wird der zugehörige Index beim Hinzufügen und Ändern von Datensätzen ständig modifiziert. Dadurch werden Teile davon nicht mehr verwendet, belegen aber weiterhin Speicherplatz, der früher oder später wieder freigegeben werden muss, um zu verhindern, dass die Festplatte irgendwann voll ist. Diesen Speicherplatz sofort freizugeben, sobald er nicht mehr verwendet wird, wäre zu zeitraubend, daher wird hin und wieder eine sogenannte *Komprimierung* der Datenbank durchgeführt, bei der die belegten Speicherbereiche »in einem Rutsch« freigegeben werden.

Dieser Komprimierungsprozess ist ein aufwendiger Vorgang, der eine beträchtlich höhere Arbeitslast für CPU und Festplatte verursacht. Dadurch sinkt die Geschwindigkeit des Servers während der Komprimierung drastisch. Datenbanken wie HBase oder Cassandra sind dafür bekannt, dass sie äußerst sorgfältig konfiguriert werden müssen, um Probleme oder sogar einen Stillstand des Servers während der Komprimierung zu verhindern. Der bei der Komprimierung auftretende Geschwindigkeitsverlust kann sogar zu einem Dominoeffekt

führen: Wenn zu viele Maschinen gleichzeitig eine Komprimierung durchführen, muss ein Großteil der Arbeitslast dieser Maschinen von anderen Rechnern des Clusters übernommen werden, was wiederum potenziell zu einer Überlastung und zu einem Totalausfall des Clusters führen kann. Wir haben solche Fehlerzustände schon häufig erlebt.

Für eine korrekte Handhabung der Datenbankkomprimierung muss geplant werden, wann welcher Netzknoten eine Komprimierung durchführt, damit nicht zu viele gleichzeitig stattfinden. Dabei müssen Sie abschätzen können, wie lange der Prozess dauert und eventuelle Abweichungen berücksichtigen, um zu verhindern, dass mehr Rechner als beabsichtigt eine Komprimierung vornehmen. Darüber hinaus müssen Sie gewährleisten, dass auf den Netzknoten hinreichend Speicherplatz vorhanden ist, um den Vorgang durchzuführen. Außerdem müssen Sie sicherstellen, dass die Rechenkapazität des Clusters ausreicht, damit es durch die während der Komprimierung fehlenden Ressourcen nicht zu einer Überlastung kommt.

All dies ist mit kompetentem Betriebspersonal durchaus machbar, aber wir behaupten, dass man jegliche Komplexität am besten dadurch handhabt, dass man sie komplett loswird. Je weniger mögliche Fehlerzustände das System besitzt, desto geringer ist die Wahrscheinlichkeit, dass es zu unerwarteten Ausfallzeiten kommt. Die Handhabung der Komprimierung im laufenden Betrieb ist integraler Bestandteil einer vollständig inkrementellen Architektur – bei einer Lambda-Architektur hingegen ist eine Komprimierung der primären Datenbanken überhaupt nicht notwendig.

1.6.2 Extreme Komplexität, um letztendliche Konsistenz zu erzielen

Eine weitere Komplexität inkrementeller Architekturen wird erkennbar, wenn man Hochverfügbarkeit benötigt. Ein hochverfügbares System erlaubt es, Abfragen und Aktualisierungen selbst dann vorzunehmen, wenn Maschinen im Cluster oder Teile des Netzwerks ausgefallen sind.

Wie sich herausstellt, konkurriert die Hochverfügbarkeit unmittelbar mit einer anderen wichtigen Eigenschaft: der *Konsistenz*. Ein konsistentes System liefert stets Ergebnisse, die alle vorangegangenen Schreibvorgänge berücksichtigen. Wie das sogenannte CAP-Theorem zeigt, ist es unmöglich, bei einem Ausfall des Netzwerks sowohl eine Hochverfügbarkeit als auch die Konsistenz eines Systems zu gewährleisten. Daher liefert ein hochverfügbares System bei einem Ausfall des Netzwerks manchmal veraltete Ergebnisse.

Das CAP-Theorem wird in Kapitel 12 ausführlich erläutert – an dieser Stelle geht es zunächst einmal darum, wie sich die Unmöglichkeit, jederzeit völlige Konsistenz und Hochverfügbarkeit zu gewährleisten, auf die Möglichkeiten bei der Einrichtung eines Systems auswirkt. Falls die Hochverfügbarkeit für die geschäftlichen Anforderungen von größerer Bedeutung ist als völlige Konsistenz, müssen Sie sich mit einem ziemlichen Ausmaß an Komplexität herumplagen.

Damit ein hochverfügbares System nach einem Netzwerkausfall wieder konsistente Ergebnisse liefert (im Englischen wird das als *Eventual Consistency* bezeichnet, zu Deutsch etwa »letztendliche Konsistenz«), muss Ihre Anwendung einiges leisten. Betrachten Sie beispielsweise den einfachen Anwendungsfall eines Zählers in einer Datenbank. Dafür ist es naheliegend, eine Zahl in der Datenbank zu speichern, die jedes Mal erhöht wird, wenn ein Ereignis auftritt, das eine Erhöhung des Zählers veranlassen soll. Es wird Sie allerdings

vielleicht überraschen, dass Sie, wenn Sie diesem Ansatz folgen, bei einem Netzwerkausfall mit heftigem Datenverlust rechnen müssen.

Der Grund hierfür ergibt sich daraus, dass verteilte Datenbanken die Hochverfügbarkeit durch mehrere Replikate der gespeicherten Informationen erreichen. Wenn Sie mehrere Kopien derselben Informationen erstellen, bleiben diese auch beim Ausfall eines Rechners oder des Netzwerks noch verfügbar (siehe Abbildung 1.4). Während eines Netzwerkausfalls aktualisiert ein hochverfügbares System diejenigen Replikate, die erreichbar sind. Dadurch weichen die Inhalte der verschiedenen Replikate voneinander ab, weil unterschiedliche Aktualisierungen durchgeführt werden. Erst nach dem Ende des Netzwerkausfalls können die Inhalte der Replikate wieder miteinander abgeglichen werden.

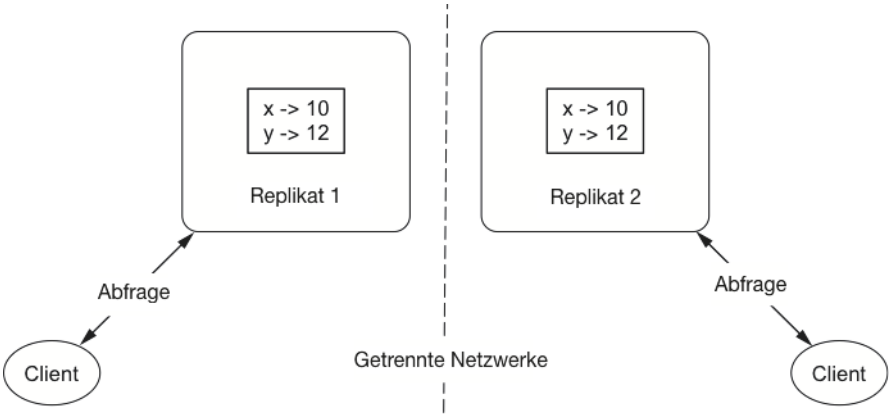

Abb. 1.4: Verwendung von Replikaten zur Erhöhung der Verfügbarkeit

Nehmen wir nun an, es gibt zwei Replikate mit einem Zähler, dessen Wert zu Beginn eines Netzwerkausfalls 10 beträgt. Dann wird der Zähler des ersten Replikats zwei Mal und derjenige des zweiten Replikats ein Mal erhöht. Wenn nun die beiden Replikate mit den Werten 12 und 11 miteinander abgeglichen werden, wie soll dann der neue Wert lauten? Die richtige Antwort lautet 13, aber es ist unmöglich, das anhand der beiden Werte 12 und 11 zu erkennen. Der Netzwerkausfall hätte auch beim Wert 11 (dann wäre der korrekte Wert 12) oder 0 (in diesem Fall wäre der Wert 23) stattfinden können.

Zur Implementierung eines hochverfügbaren Zählers reicht es nicht aus, nur einen Wert zu speichern. Sie benötigen eine Datenstruktur, die bei der Zusammenführung der Replikate zugänglich ist und darüber Auskunft gibt, seit wann die Werte voneinander abweichen. Außerdem müssen Sie Code schreiben, der die Werte korrigiert, sobald der Netzwerkausfall endet – das ist schon eine erstaunliche Komplexität, mit der Sie sich herumplagen müssen, bloß um einen einfachen Zähler zu implementieren.

Die Implementierung letztendlicher Konsistenz in inkrementellen, hochverfügbaren Systemen ist im Allgemeinen nicht intuitiv und fehleranfällig. Diese Komplexität ist integraler Bestandteil hochverfügbarer und vollständig inkrementeller Systeme. Sie werden später noch sehen, inwiefern die andersartige Strukturierung der Lambda-Architektur den Aufwand zum Erzielen eines hochverfügbaren, letztendlich konsistenten Systems erheblich verringert.

1.6.3 Keine Fehlertoleranz gegenüber menschlichem Versagen

Die letzte Schwierigkeit, die wir im Zusammenhang mit vollständig inkrementellen Architekturen aufzeigen möchten, ist das systembedingte Fehlen einer Fehlertoleranz gegenüber menschlichem Versagen. Ein inkrementelles System modifiziert den in der Datenbank gespeicherten Zustand fortwährend, daher werden auch Fehler darin gespeichert. Weil Fehler jedoch unvermeidlich sind, enthält die Datenbank einer vollständig inkrementellen Architektur mit Sicherheit fehlerhafte Daten.

An dieser Stelle ist es wichtig darauf hinzuweisen, dass es sich hierbei um eine der wenigen Komplexitäten vollständig inkrementeller Architekturen handelt, die umgangen werden können, ohne dass die Architektur komplett neu überdacht werden muss. Betrachten Sie einmal die beiden in Abbildung 1.5 dargestellten Architekturen: eine synchrone Architektur, bei der die Anwendung die Datenbank direkt aktualisiert, und eine asynchrone Architektur, bei der die Ereignisse in eine Warteschlange eingereiht werden, bevor die Datenbank im Hintergrund aktualisiert wird. In beiden Fällen werden die ausgeführten Aktionen dauerhaft in einem Ereignisprotokoll gespeichert. Sollten durch menschliches Versagen fehlerhafte Daten verursacht werden, kann anhand des Ereignisprotokolls wieder ein korrekter Zustand der Datenbank rekonstruiert werden. Das Ereignisprotokoll ist unveränderlich und wächst ständig. Und Sie können redundante Überprüfungen vornehmen, beispielsweise die Zugriffsrechte testen, um es höchst unwahrscheinlich zu machen, dass das Ereignisprotokoll durch einen Fehler Schaden nimmt. Diese Vorgehensweise ist ebenfalls Bestandteil der Lambda-Architektur und wird in den Kapiteln 2 und 3 ausführlich erörtert.

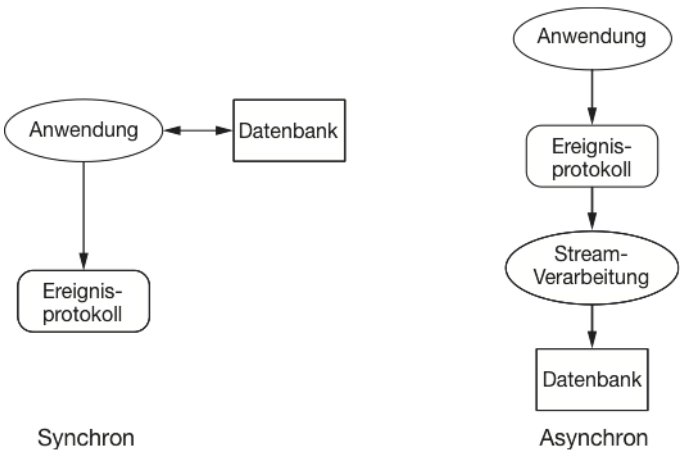

Abb. 1.5: Hinzufügen einer Protokollierung zu einer vollständig inkrementellen Architektur

Vollständig inkrementelle Architekturen können den Mangel der Fehlertoleranz gegenüber menschlichem Versagen zwar durch eine Protokollierung umgehen, auf die anderen aufgeführten Komplexitäten hat sie jedoch keinen Einfluss. Und wie Sie im nächsten Abschnitt sehen werden, haben alle vollständig inkrementellen Architekturen, auch solche, die Protokolle führen, bei der Lösung vieler Aufgaben mit Problemen zu kämpfen.

1.6.4 Vollständig inkrementelle Lösung kontra Lambda-Architektur

Eine der Beispielabfragen, die im gesamten Buch immer wieder Verwendung findet, dient der Gegenüberstellung einer vollständig inkrementellen Architektur und der Lambda-Architektur. Diese Abfrage ist in keinster Weise konstruiert – tatsächlich beruht sie auf einem realen Problem, dem wir in der Praxis schon mehrfach begegnet sind. Sie bezieht sich auf die Berechnung der Pageviews und betrifft zwei verschiedene Arten von eingehenden Daten:

- *Pageviews*, die eine User-ID, eine URL und einen Zeitstempel (engl. *timestamp*) enthalten.
- *Equivs*, die zwei User-IDs enthalten. Ein Equiv bedeutet, dass die beiden User-IDs auf ein und dieselbe Person verweisen. So könnte es beispielsweise ein Equiv für die E-Mail-Adresse *sally@gmail.com* und den Benutzernamen *sally* geben. Falls sich *sally@gmail.com* ein weiteres Mal mit dem Benutzernamen *sally2* registriert, gibt es ein weiteres Equiv für *sally@gmail.com* und *sally2*. Dank der Transitivität ist somit klar, dass die Benutzernamen *sally* und *sally2* auf dieselbe Personen verweisen.

Ziel der Abfrage ist es, die Anzahl der eindeutig unterschiedlichen Besucher einer URL in einem bestimmten Zeitraum zu ermitteln. Die Antwort auf eine Abfrage sollte sämtliche aktuellen Daten berücksichtigen und mit minimaler Latenzzeit (weniger als 100 Millisekunden) geliefert werden. Hier die entsprechende Abfrageschnittstelle:

```
long uniquesOverTime(String url, int startHour, int endHour)
```

Die Equivs machen die Implementierung der Abfrage allerdings etwas knifflig: Falls eine Person dieselbe URL in einem gegebenen Zeitraum mit zwei User-IDs besucht, die über Equivs miteinander verknüpft sind (auch transitiv), soll dies lediglich als ein Besuch gewertet werden. Daher kann ein neu eingehendes Equiv die Ergebnisse aller anderen Abfragen in sämtlichen Zeiträumen ändern.

Wir verzichten hier darauf, die Details der Lösungen zu präsentieren, weil noch zu viele Konzepte erörtert werden müssen, um sie verstehen zu können: Indizierung, verteilte Datenbanken, Stapelverarbeitung, HyperLogLog und vieles mehr. Es wäre kontraproduktiv, Sie schon jetzt mit all diesen Konzepten zu erschlagen. Stattdessen konzentrieren wir uns an dieser Stelle auf die Merkmale und die eklatanten Unterschiede zwischen den beiden Lösungen. Die bestmögliche vollständig inkrementelle Lösung wird in Kapitel 10 ausführlich vorgestellt. Die Lambda-Architektur-Lösung wird in den Kapiteln 8, 9, 14 und 15 allmählich entwickelt.

Besagte Lösungen können anhand dreier Kriterien miteinander verglichen werden: Genauigkeit, Latenz und Datendurchsatz. Die Lambda-Architektur-Lösung ist in jeder Hinsicht deutlich überlegen. In beiden Fällen müssen Näherungen zur Anwendung kommen, allerdings ist die vollständig inkrementelle Version dazu gezwungen, ein weniger genaues Näherungsverfahren mit einer drei bis fünf Mal höheren Fehlerrate zu verwenden. Die Ausführung ist bei der vollständig inkrementellen Version erheblich zeitaufwendiger, wovon sowohl die Latenzzeit als auch der Datendurchsatz betroffen sind. Der auffallendste Unterschied zwischen den beiden Ansätzen ist jedoch, dass die vollständig inkrementelle Variante auf spezielle Hardware zurückgreifen muss, um einen halbwegs akzeptablen Datendurchsatz zu erzielen. Da die vollständig inkrementelle Version bei der Erledigung

von Abfragen viele wahlfreie Zugriffe durchführen muss, ist es praktisch unumgänglich, SSD-Laufwerke zu verwenden, damit Festplattenzugriffe nicht zum Engpass werden.

Dass die Lambda-Architektur in jeder Hinsicht leistungsfähigere Lösungen hervorbringen kann und gleichzeitig die Komplexität umgeht, die vollständig inkrementellen Architekturen zu schaffen macht, weist darauf hin, wie grundsätzlich anders dieser Ansatz ist. Entscheidend ist, die Fesseln vollständig inkrementeller Architekturen abzuwerfen und sich anderen Vorgehensweisen zuzuwenden. Sehen wir uns das genauer an.

1.7 Lambda-Architektur

Beliebige Funktionen mit beliebigen Eingabedaten in Echtzeit zu berechnen, stellt eine schwierige Aufgabe dar. Ein einziges Tool, das hierfür eine Lösung bietet, gibt es nicht. Sie müssen bei der Einrichtung eines Big-Data-Systems vielmehr eine Vielzahl verschiedener Tools und Techniken einsetzen.

Big-Data-Systeme in Form diverser Layer einzurichten, ist eine der grundlegenden Ideen der Lambda-Architektur (siehe Abbildung 1.6). Jeder Layer erfüllt eine Teilmenge der Anforderungen und beruht auf der Funktionalität der darunter befindlichen Layer. Es geht in diesem gesamten Buch um die Details des Designs, die Implementierung und das Deployment der einzelnen Layer, aber die grundsätzlichen Ideen, wie sie zusammenwirken, um das System als Ganzes zu bilden, sind ziemlich einfach zu verstehen.

Ausgangspunkt ist die Gleichung `Abfrage = Funktion(Sämtliche Daten)`. Idealerweise könnte man die Funktion direkt ausführen, um das Ergebnis zu erhalten. Aber selbst wenn das möglich wäre, würde es enorme Ressourcen und einen unangemessenen Aufwand erfordern. Stellen Sie sich beispielsweise vor, Sie müssten bei jeder Abfrage nach dem aktuellen Aufenthaltsort einer Person ein Petabyte Daten einlesen.

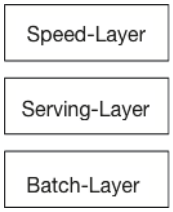

Abb. 1.6: Lambda-Architektur

Der naheliegende alternative Ansatz ist die Vorabberechnung der Abfragefunktion, die wir als *Batch-View* bezeichnen. Statt die Funktion im laufenden Betrieb zu berechnen, entnehmen Sie die Ergebnisse der vorab berechneten Batch-View. Diese ist wiederum indiziert, sodass wahlfrei lesend darauf zugegriffen werden kann. Das System ist nun folgendermaßen aufgebaut:

```
Batch-View = Funktion(Sämtliche Daten)
    Abfrage = Funktion(Batch-View)
```

Bei diesem System führen Sie eine Funktion mit sämtlichen Daten aus, um die Batch-View zu erhalten. Wenn Sie nun das Ergebnis einer Abfrage wissen möchten, führen Sie eine

Funktion mit der Batch-View aus, die die angefragten Werte sehr schnell zur Verfügung stellen kann, ohne erst alle darin enthaltenen Werte durchsuchen müssen.

Dieser Vorgang ist ziemlich abstrakt, betrachten wir also ein Beispiel. Nehmen wir an, Sie programmieren (mal wieder) eine Anwendung für Web Analytics und Sie möchten die Anzahl der URL-Aufrufe innerhalb einer bestimmten Anzahl von Tagen abfragen. Wenn Sie das Ergebnis als Funktion sämtlicher Daten berechnen, müssen Sie den Datensatz nach den Pageviews der URL innerhalb des Zeitraums durchsuchen und die Anzahl der gefundenen Datensätze zurückliefern.

Beim Batch-View-Ansatz wird stattdessen eine Funktion mit sämtlichen Pageviews ausgeführt, um einen Index zu berechnen, der anhand eines Schlüssels `[url, day]` die Anzahl der Pageviews der URL am angegebenen Tag liefert. Zur Beantwortung der Abfrage rufen Sie aus der Batch-View die Werte der Tage im angegebenen Zeitraum auf und summieren sie, um das Ergebnis zu erhalten. In Abbildung 1.7 ist dieser Ansatz dargestellt.

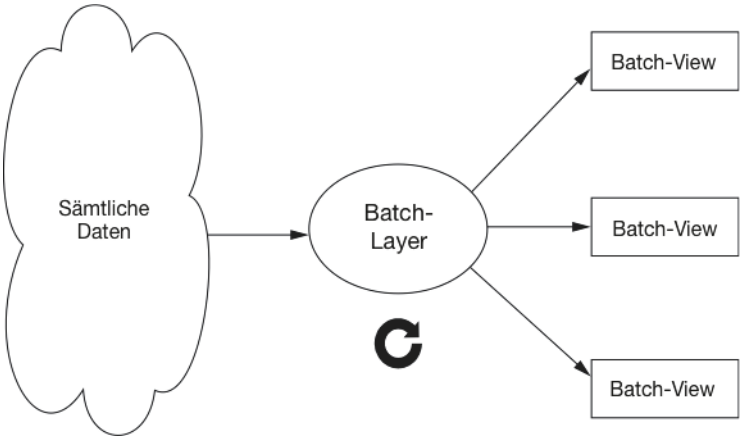

Abb. 1.7: Architektur des Batch-Layers

Es sollte klar sein, dass bei der bisherigen Beschreibung dieses Ansatzes noch etwas fehlt: Die Erzeugung der Batch-View ist zweifelsohne ein Vorgang mit hoher Latenzzeit, weil eine Funktion mit sämtlichen Daten ausgeführt wird. Wenn die Ausführung beendet ist, werden sich zwischenzeitlich neue Daten angesammelt haben, die in den Batch-Views nicht berücksichtigt sind, und damit sind die Ergebnisse schon veraltet. Sie können dieses Problem jedoch fürs Erste ignorieren, denn es lässt sich beheben. Wir gehen also davon aus, dass einige Stunden alte Ergebnisse in Ordnung sind und verfolgen weiterhin die Idee, Batch-Views durch die Ausführung einer Funktion mit sämtlichen Daten vorab zu berechnen.

1.7.1 Batch-Layer

Der Teil der Lambda-Architektur, der die Gleichung `Batch-View = Funktion(Sämtliche Daten)` implementiert, wird als *Batch-Layer* bezeichnet. Dieser Layer speichert die Master-Kopie des Datensatzes und berechnet anhand der darin vorhandenen Daten vorab die Batch-Views (siehe Abbildung 1.8). Sie können sich diesen Stammdatensatz als eine sehr lange Liste von Datensätzen vorstellen.