



mitp

Nathan  
Marz  
mit  
James  
Warren

# Big Data

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

# Inhaltsverzeichnis

	<b>Vorwort</b> .....	9
	<b>Danksagungen</b> .....	11
	<b>Über dieses Buch</b> .....	15
<b>1</b>	<b>Ein neues Paradigma für Big Data</b> .....	17
1.1	Aufbau des Buches.....	18
1.2	Skalierung mit einer herkömmlichen Datenbank.....	19
1.2.1	Skalierung mit einer Warteschlange.....	19
1.2.2	Skalierung durch Sharding.....	20
1.2.3	Erste Probleme mit der Fehlertoleranz.....	21
1.2.4	Probleme mit fehlerhaften Daten.....	21
1.2.5	Was ist schiefgegangen?.....	21
1.2.6	Inwiefern sind Big-Data-Verfahren hilfreich?.....	22
1.3	NoSQL ist kein Allheilmittel.....	22
1.4	Grundlagen.....	23
1.5	Erwünschte Eigenschaften eines Big-Data-Systems.....	24
1.5.1	Belastbarkeit und Fehlertoleranz.....	24
1.5.2	Lesen und Aktualisieren mit geringen Latenzzeiten.....	25
1.5.3	Skalierbarkeit.....	25
1.5.4	Allgemeingültigkeit.....	25
1.5.5	Erweiterbarkeit.....	25
1.5.6	Ad-hoc-Abfragen.....	25
1.5.7	Minimaler Wartungsaufwand.....	26
1.5.8	Fehlerbehebung.....	26
1.6	Schwierigkeiten vollständig inkrementeller Architekturen.....	26
1.6.1	Komplexität im Betrieb.....	27
1.6.2	Extreme Komplexität, um letztendliche Konsistenz zu erzielen.....	28
1.6.3	Keine Fehlertoleranz gegenüber menschlichem Versagen.....	30
1.6.4	Vollständig inkrementelle Lösung kontra Lambda-Architektur.....	31
1.7	Lambda-Architektur.....	32
1.7.1	Batch-Layer.....	33
1.7.2	Serving-Layer.....	35
1.7.3	Batch- und Serving-Layer erfüllen fast alle Anforderungen.....	35
1.7.4	Speed-Layer.....	36
1.8	Die neuesten Trends.....	39
1.8.1	Prozessoren werden kaum noch schneller.....	39
1.8.2	Elastic Clouds.....	39
1.8.3	Ein lebhaftes Open-Source-Ökosystem für Big Data.....	40
1.9	Beispielanwendung: SuperWebAnalytics.com.....	41
1.10	Zusammenfassung.....	41

<b>Teil I</b>	<b>Batch-Layer</b> .....	43
<b>2</b>	<b>Das Datenmodell für Big Data</b> .....	45
2.1	Die Eigenschaften von Daten .....	46
2.1.1	Daten sind ursprünglich .....	49
2.1.2	Daten sind unveränderlich .....	52
2.1.3	Daten sind beständig korrekt .....	54
2.2	Das faktenbasierte Modell zur Repräsentierung von Daten .....	55
2.2.1	Faktenbeispiele und ihre Eigenschaften .....	56
2.2.2	Vorteile des faktenbasierten Modells .....	58
2.3	Graphenschemata .....	62
2.3.1	Elemente eines Graphenschemas .....	62
2.3.2	Die Notwendigkeit, dem Schema zu gehorchen .....	63
2.4	Ein vollständiges Datenmodell für SuperWebAnalytics.com .....	64
2.5	Zusammenfassung .....	65
<b>3</b>	<b>Das Datenmodell für Big Data: Praxis</b> .....	67
3.1	Wozu ein Serialisierungs-Framework? .....	67
3.2	Apache Thrift .....	68
3.2.1	Knoten .....	69
3.2.2	Kanten .....	69
3.2.3	Eigenschaften .....	70
3.2.4	Alles in Datenobjekten zusammenfassen .....	71
3.2.5	Weiterentwicklung des Schemas .....	71
3.3	Für Serialisierungs-Frameworks geltende Beschränkungen .....	72
3.4	Zusammenfassung .....	73
<b>4</b>	<b>Datenspeicherung im Batch-Layer</b> .....	75
4.1	Speicheranforderungen des Stammdatensatzes .....	76
4.2	Auswahl einer Speicherlösung für den Batch-Layer .....	77
4.2.1	Schlüssel-Werte-Datenbank zum Speichern des Stammdatensatzes verwenden .....	77
4.2.2	Verteilte Dateisysteme .....	78
4.3	Funktionsweise verteilter Dateisysteme .....	79
4.4	Speichern des Stammdatensatzes mit einem verteilten Dateisystem .....	81
4.5	Vertikale Partitionierung .....	83
4.6	Verteilte Dateisysteme sind maschinennah .....	84
4.7	Speichern des SuperWebAnalytics.com-Stammdatensatzes in einem verteiltem Dateisystem .....	85
4.8	Zusammenfassung .....	86
<b>5</b>	<b>Datenspeicherung im Batch-Layer: Praxis</b> .....	87
5.1	Verwendung des Hadoop Distributed File Systems .....	87
5.1.1	Das Problem mit kleinen Dateien .....	89
5.1.2	Eine allgemeinere Abstrahierung .....	89
5.2	Datenspeicherung im Batch-Layer mit Pail .....	91
5.2.1	Grundlegende Pail-Operationen .....	91
5.2.2	Objekte serialisieren und in Pails speichern .....	93

5.2.3	Pail-Operationen . . . . .	95
5.2.4	Vertikale Partitionierung mit Pail . . . . .	96
5.2.5	Pail-Dateiformat und Komprimierung . . . . .	97
5.2.6	Vorteile von Pail zusammengefasst. . . . .	98
5.3	Speichern des Stammdatensatzes für SuperWebAnalytics.com . . . . .	99
5.3.1	Ein strukturiertes Pail für Thrift-Objekte . . . . .	101
5.3.2	Ein einfaches Pail für SuperWebAnalytics.com . . . . .	102
5.3.3	Ein geteiltes Pail zur vertikalen Partitionierung des Datensatzes . . . . .	103
5.4	Zusammenfassung. . . . .	107
<b>6</b>	<b>Batch-Layer . . . . .</b>	<b>109</b>
6.1	Beispiele . . . . .	110
6.1.1	Anzahl der Pageviews innerhalb eines bestimmten Zeitraums . . . . .	110
6.1.2	Vorhersage des Geschlechts. . . . .	110
6.1.3	Einflussreiche Tweets . . . . .	111
6.2	Berechnungen im Batch-Layer . . . . .	112
6.3	Neuberechnungsalgorithmen kontra inkrementelle Algorithmen. . . . .	114
6.3.1	Performance . . . . .	115
6.3.2	Fehlertoleranz gegenüber menschlichem Versagen . . . . .	116
6.3.3	Allgemeine Anwendbarkeit des Algorithmus. . . . .	117
6.3.4	Auswahl eines Algorithmustyps . . . . .	118
6.4	Skalierbarkeit im Batch-Layer . . . . .	118
6.5	MapReduce: Ein Paradigma für Big-Data-Berechnungen. . . . .	120
6.5.1	Skalierbarkeit. . . . .	121
6.5.2	Fehlertoleranz . . . . .	123
6.5.3	Allgemeine Anwendbarkeit von MapReduce . . . . .	123
6.6	Maschinennähe . . . . .	126
6.6.1	Berechnungen in mehreren Schritten sind nicht intuitiv . . . . .	126
6.6.2	Die manuelle Implementierung von Joins ist sehr kompliziert . . . . .	126
6.6.3	Enge Kopplung der logischen und physischen Ausführung. . . . .	128
6.7	Pipe-Diagramme: Eine allgemeinere Auffassung von Stapelverarbeitungsberechnungen . . . . .	129
6.7.1	Konzepte der Pipe-Diagramme . . . . .	130
6.7.2	Ausführen von Pipe-Diagrammen via MapReduce . . . . .	134
6.7.3	Combiner-Aggregator . . . . .	135
6.7.4	Beispiele für Pipe-Diagramme. . . . .	136
6.8	Zusammenfassung. . . . .	138
<b>7</b>	<b>Batch-Layer: Praxis . . . . .</b>	<b>139</b>
7.1	Ein Beispiel zur Veranschaulichung. . . . .	140
7.2	Typische Schwierigkeiten Daten verarbeitender Tools . . . . .	142
7.2.1	Proprietäre Sprachen . . . . .	142
7.2.2	Mangelhaft einbindungsfähige Abstraktionen. . . . .	143
7.3	Einführung in JCascalog . . . . .	144
7.3.1	Das JCascalog-Datenmodell . . . . .	144
7.3.2	Aufbau einer JCascalog-Abfrage . . . . .	146
7.3.3	Abfragen mehrerer Datensätze . . . . .	147
7.3.4	Gruppierung und Aggregatoren . . . . .	150

7.3.5	Schrittweise Abarbeitung einer Abfrage.....	151
7.3.6	Benutzerdefinierte Prädikate.....	154
7.4	Einbindung.....	159
7.4.1	Subqueries kombinieren.....	160
7.4.2	Dynamisch erzeugte Subqueries.....	161
7.4.3	Prädikatmakros.....	164
7.4.4	Dynamisch erzeugte Prädikatmakros.....	167
7.5	Zusammenfassung.....	170
<b>8</b>	<b>Beispiel eines Batch-Layers: Architektur und Algorithmen</b> .....	<b>171</b>
8.1	Design des Batch-Layers für SuperWebAnalytics.com.....	172
8.1.1	Unterstützte Abfragen.....	172
8.1.2	Batch-Views.....	173
8.2	Überblick über den Workflow.....	176
8.3	Aufnahme neuer Daten.....	177
8.4	URL-Normalisierung.....	178
8.5	User-ID-Normalisierung.....	179
8.6	Deduplizierung der Pageviews.....	184
8.7	Berechnung der Batch-Views.....	184
8.7.1	Zeitlicher Verlauf der Pageviews.....	185
8.7.2	Zeitlicher Verlauf der eindeutig unterschiedlichen Besucher.....	186
8.7.3	Analyse der Bounce-Rate.....	187
8.8	Zusammenfassung.....	188
<b>9</b>	<b>Beispiel eines Batch-Layers: Implementierung</b> .....	<b>189</b>
9.1	Ausgangspunkt.....	189
9.2	Vorbereitung des Workflows.....	190
9.3	Aufnahme neuer Daten.....	191
9.4	URL-Normalisierung.....	195
9.5	User-ID-Normalisierung.....	197
9.6	Deduplizierung der Pageviews.....	204
9.7	Berechnung der Batch-Views.....	204
9.7.1	Zeitlicher Verlauf der Pageviews.....	204
9.7.2	Zeitlicher Verlauf der eindeutig unterschiedlichen Besucher.....	207
9.7.3	Berechnung der Bounce-Rate.....	209
9.8	Zusammenfassung.....	212
<b>Teil II Serving-Layer</b> .....		<b>213</b>
<b>10</b>	<b>Serving-Layer</b> .....	<b>215</b>
10.1	Performancekennzahlen des Serving-Layers.....	216
10.2	Lösung des Problems »Normalisierung kontra Denormalisierung« durch den Serving-Layer.....	219
10.3	Anforderungen an eine Datenbank für den Serving-Layer.....	221
10.4	Gestaltung eines Serving-Layers für SuperWebAnalytics.com.....	222
10.4.1	Zeitlicher Verlauf der Pageviews.....	223
10.4.2	Zeitlicher Verlauf eindeutig unterschiedlicher Besucher.....	223

10.4.3	Berechnung der Bounce-Rate. . . . .	224
10.5	Vergleich mit einer vollständig inkrementellen Lösung. . . . .	225
10.5.1	Vollständig inkrementelle Lösung. . . . .	225
10.5.2	Vergleich mit einer auf der Lambda-Architektur beruhenden Lösung. . . . .	231
10.6	Zusammenfassung. . . . .	232
<b>11</b>	<b>Serving-Layer: Praxis</b> . . . . .	<b>233</b>
11.1	ElephantDB: Grundlagen. . . . .	233
11.1.1	Views in ElephantDB erzeugen . . . . .	234
11.1.2	Views in ElephantDB deployen . . . . .	234
11.1.3	ElephantDB verwenden . . . . .	235
11.2	Einrichtung des Serving-Layers für SuperWebAnalytics.com . . . . .	237
11.2.1	Zeitlicher Verlauf der Pageviews . . . . .	237
11.2.2	Zeitlicher Verlauf eindeutig unterschiedlicher Besucher . . . . .	240
11.2.3	Berechnung der Bounce-Rate. . . . .	241
11.3	Zusammenfassung. . . . .	242
<b>Teil III Speed-Layer</b> . . . . .		<b>243</b>
<b>12</b>	<b>Echtzeit-Views</b> . . . . .	<b>245</b>
12.1	Berechnung von Echtzeit-Views . . . . .	246
12.2	Speichern der Echtzeit-Views . . . . .	248
12.2.1	Letztendliche Genauigkeit . . . . .	249
12.2.2	Im Speed-Layer gespeicherter Zustand. . . . .	249
12.3	Schwierigkeiten bei inkrementeller Berechnung. . . . .	250
12.3.1	Gültigkeit des CAP-Theorems . . . . .	251
12.3.2	Das komplexe Zusammenwirken von CAP-Theorem und inkrementellen Algorithmen . . . . .	253
12.4	Asynchrone kontra synchrone Aktualisierungen. . . . .	254
12.5	Echtzeit-Views verwerfen. . . . .	256
12.6	Zusammenfassung. . . . .	258
<b>13</b>	<b>Echtzeit-Views: Praxis</b> . . . . .	<b>259</b>
13.1	Cassandras Datenmodell . . . . .	259
13.2	Cassandra verwenden. . . . .	261
13.2.1	Cassandra für Fortgeschrittene . . . . .	263
13.3	Zusammenfassung. . . . .	264
<b>14</b>	<b>Warteschlangen und Streamverarbeitung</b> . . . . .	<b>265</b>
14.1	Warteschlangen . . . . .	265
14.1.1	Warteschlangen mit nur einem Abnehmer . . . . .	266
14.1.2	Warteschlangen mit mehreren Abnehmern. . . . .	268
14.2	Streamverarbeitung . . . . .	269
14.2.1	Warteschlangen und Worker . . . . .	270
14.2.2	Fallstricke beim Warteschlangen-Worker-Ansatz . . . . .	271

14.3	Streamverarbeitung one-at-a-time auf höherer Ebene .....	272
14.3.1	Storm-Modell .....	272
14.3.2	Gewährleistung der Nachrichtenverarbeitung .....	277
14.4	SuperWebAnalytics.com: Speed-Layer .....	279
14.4.1	Aufbau der Topologie .....	281
14.5	Zusammenfassung .....	282
<b>15</b>	<b>Warteschlangen und Streamverarbeitung: Praxis .....</b>	<b>283</b>
15.1	Definition einer Topologie mit Apache Storm .....	283
15.2	Apache Storm-Cluster und Bereitstellung .....	286
15.3	Gewährleistung der Nachrichtenverarbeitung .....	288
15.4	Implementierung des Speed-Layers .....	291
15.5	Zusammenfassung .....	296
<b>16</b>	<b>Streamverarbeitung kleiner Stapel .....</b>	<b>297</b>
16.1	Genau einmalige Verarbeitung .....	297
16.1.1	Verarbeitung in streng festgelegter Reihenfolge .....	298
16.1.2	Streamverarbeitung kleiner Stapel .....	299
16.1.3	Topologien zur Verarbeitung kleiner Stapel .....	300
16.2	Grundlegende Konzepte der Streamverarbeitung kleiner Stapel .....	302
16.3	Erweiterte Pipe-Diagramme zur Beschreibung der Streamverarbeitung kleiner Stapel .....	304
16.4	Fertigstellung des Speed-Layers für SuperWebAnalytics.com .....	305
16.4.1	Zeitlicher Verlauf der Pageviews .....	305
16.4.2	Berechnung der Bounce-Rate .....	306
16.5	Eine weitere Methode zur Berechnung der Bounce-Rate .....	311
16.6	Zusammenfassung .....	312
<b>17</b>	<b>Streamverarbeitung kleiner Stapel: Praxis .....</b>	<b>313</b>
17.1	Trident verwenden .....	313
17.2	Fertigstellung des Speed-Layers für SuperWebAnalytics.com .....	317
17.2.1	Zeitlicher Verlauf der Pageviews .....	317
17.2.2	Berechnung der Bounce-Rate .....	320
17.3	Fehlertolerante Verarbeitung kleiner Stapel im Arbeitsspeicher .....	326
17.4	Zusammenfassung .....	328
<b>18</b>	<b>Die Lambda-Architektur im Detail .....</b>	<b>329</b>
18.1	Definition von Datenhaltungssystemen .....	329
18.2	Batch- und Serving-Layer .....	331
18.2.1	Inkrementelle Stapelverarbeitung .....	331
18.2.2	Ressourcennutzung des Batch-Layers messen und optimieren .....	338
18.3	Speed-Layer .....	343
18.4	Query-Layer .....	343
18.5	Zusammenfassung .....	345
	<b>Stichwortverzeichnis .....</b>	<b>347</b>



# 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 böses 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 Jambani, 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 hervorragenden 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 Greta 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.

# 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 lauerten 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 – Super-WebAnalytics.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 Datentrain 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.

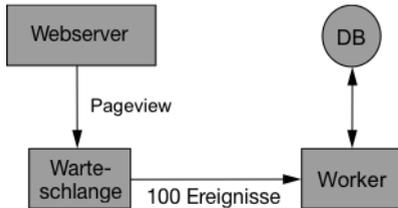


Abb. 1.2: Stapelverarbeitung mit Warteschlange und Worker

## 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 irgendwann irgendwem ein Fehler unterläuft, der Einfluss auf das Produkktivsystem 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.

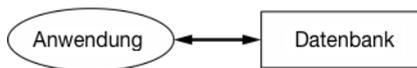
All diese Eigenschaften in einem System zu vereinen, mag auf den ersten Blick wie eine einschüchternde Herausforderung erscheinen. Wenn man jedoch, wie bei der Lambda-Architektur, von den Grundlagen ausgeht, ergeben sich diese Eigenschaften durch das resultierende Systemdesign von ganz allein.

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.

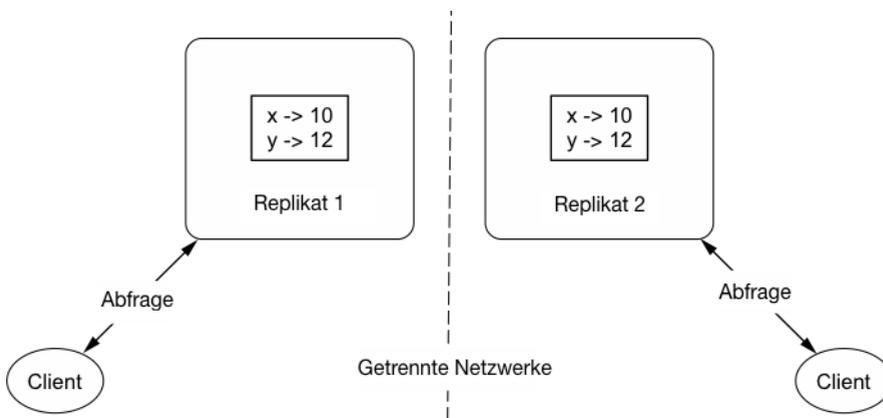
Wie sich herausstellt, konkurriert die Hochverfügbarkeit unmittelbar mit einer anderen wichtigen Eigenschaft: der *Konsistenz*. Ein konsistentes System liefert stets Ergebnisse, die alle vorangegangenen Schreibvorgänge berücksichtigen. Wie das sogenannte CAP-Theorem zeigt, ist es unmöglich, bei einem 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 Netzerkausfall 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 Netzerkausfalls 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 Netzerkausfalls 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 Netzerkausfalls 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 Netzerkausfall 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 Netzerkausfall 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 eingereicht 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.

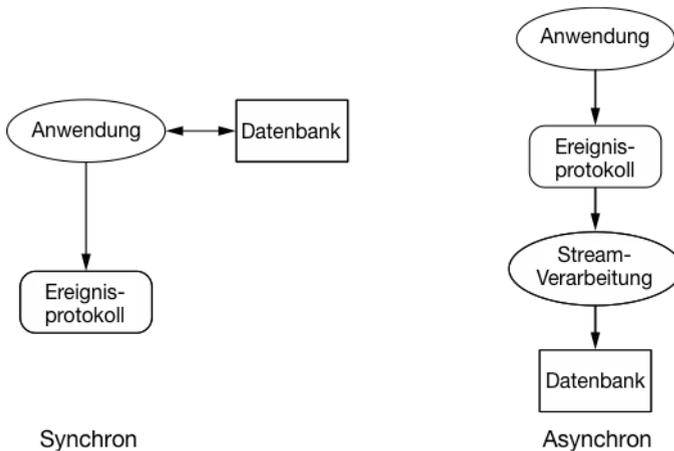


Abb. 1.5: Hinzufügen einer Protokollierung zu einer vollständig inkrementellen Architektur

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  $\text{Abfrage} = \text{Funktion}(\text{Sämtliche Daten})$ . 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.

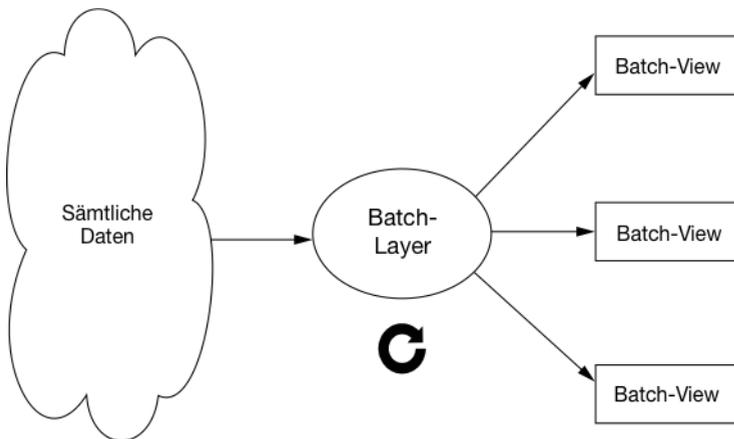


Abb. 1.7: Architektur des Batch-Layers

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  $\text{Batch-View} = \text{Funktion}(\text{Sämtliche Daten})$  implementiert, wird als *Batch-Layer* bezeichnet. Dieser Layer speichert die Masterkopie 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.

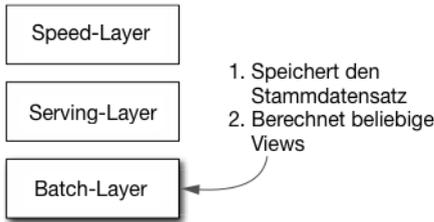


Abb. 1.8: Batch-Layer

Der Batch-Layer muss zwei Dinge erledigen können: den ständig wachsenden, unveränderlichen Stammdatensatz speichern und beliebige Funktionen mit den Daten ausführen. Für Aufgaben dieser Art sind Stapelverarbeitungssysteme am besten geeignet. Hadoop ist ein klassisches Beispiel für solche Systeme und kommt im weiteren Verlauf des Buches zum Einsatz, um die Konzepte des Batch-Layers zu demonstrieren.

Die einfachste Form eines Batch-Layers kann als Pseudocode folgendermaßen dargestellt werden:

```
function runBatchLayer():
  while(true):
    recomputeBatchViews()
```

Der Batch-Layer läuft in einer `while(true)`-Schleife und berechnet die Batch-Views immer wieder neu. In der Praxis ist er etwas komplizierter, aber dazu kommen wir später. Fürs Erste ist diese Betrachtungsweise des Batch-Layers am besten.

Das Schöne an diesem Layer ist seine simple Verwendbarkeit. Die Berechnungen werden wie Programme geschrieben, die in einem einzelnen Thread laufen – für die parallele Ausführung sorgt das System. Daher ist es einfach, stabile, hochskalierbare Berechnungen für den Batch-Layer zu programmieren. Zum Skalieren werden weitere Maschinen hinzugefügt.

Hier ein Beispiel für eine im Batch-Layer ausgeführte Berechnung – Sie brauchen den Code nicht zu verstehen, es geht an dieser Stelle nur darum zu zeigen, wie ein grundsätzlich parallel ausführbares Programm aussieht:

```
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
  new Subquery("?url", "?count")
  .predicate(Api.hfsSeqfile("/data/pageviews"),
    "?url", "?user", "?timestamp")
  .predicate(new Count(), "?count");
```

Dieser Code berechnet die Anzahl der Pageviews für alle URLs einer aus Pageview-Datensätzen bestehenden Eingabemenge. Das Bemerkenswerte daran ist, dass sämtliche Aufgaben der Parallelverarbeitung und des Zusammenführens der Ergebnisse automatisch erledigt werden. Da der Algorithmus auf diese Weise programmiert ist, kann er nach Belieben auf einem MapReduce-Cluster laufen und skaliert dabei, je nach Anzahl der verfügbaren Maschinen, automatisch. Nach Abschluss der Berechnungen enthält das Ausga-

beverzeichnis eine Reihe von Dateien mit den Ergebnissen. Wie man Programme wie dieses schreibt, erfahren Sie in Kapitel 7.

## 1.7.2 Serving-Layer

Der Batch-Layer liefert Batch-Views als Resultat seiner Funktionen. Im nächsten Schritt müssen die Views zugänglich gemacht werden, damit Abfragen vorgenommen werden können. An dieser Stelle kommt der Serving-Layer ins Spiel. Hierbei handelt es sich um eine spezielle verteilte Datenbank, die Batch-Views lädt und wahlfreien Lesezugriff darauf ermöglicht (siehe Abbildung 1.9). Sobald neue Batch-Views bereitstehen, tauscht der Serving-Layer die älteren automatisch gegen die neuen aus, sodass aktuellere Resultate verfügbar sind.

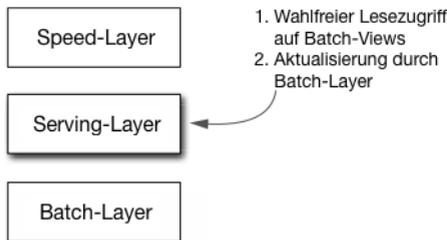


Abb. 1.9: Serving-Layer

Die Datenbank des Serving-Layers unterstützt Aktualisierungen durch den Batch-Layer und wahlfreien Lesezugriff darauf. Wahlfreien *Schreibzugriff* unterstützt sie wohlgermerkt nicht. Das ist ein wichtiger Aspekt, denn der wahlfreie Schreibzugriff leistet den größten Beitrag zur Komplexität von Datenbanken. Da er hier jedoch nicht unterstützt wird, sind diese Datenbanken extrem einfach – und durch eben diese Einfachheit sind sie so stabil, vorhersagbar, leicht zu konfigurieren und zu betreiben. Die Datenbank des Serving-Layers namens ElephantDB, die Sie im weiteren Verlauf dieses Buches noch genauer kennenlernen werden, besteht nur aus ein paar Tausend Zeilen Code.

## 1.7.3 Batch- und Serving-Layer erfüllen fast alle Anforderungen

Der Batch- und der Serving-Layer unterstützen beliebige Abfragen an beliebige Datensätze, weisen jedoch den Nachteil auf, dass die Ergebnisse mehrere Stunden alt sind. Es dauert einige Stunden, bis sich neue Daten durch den Batch-Layer zum Serving-Layer vorgearbeitet haben, wo sie abgefragt werden können. An dieser Stelle ist es wichtig festzuhalten, dass – abgesehen von der hohen Latenzzeit der Aktualisierung – die Batch- und Serving-Layer allen in Abschnitt 1.5 aufgeführten Anforderungen an ein Big-Data-System genügen. Sehen wir sie uns der Reihe nach an:

- **Belastbarkeit und Fehlertoleranz** – Hadoop handhabt den Ausfall von Maschinen automatisch. Der Serving-Layer erstellt im Hintergrund Replikate, um bei Serverausfällen die Verfügbarkeit zu gewährleisten. Batch- und Serving-Layer sind somit fehlertolerant gegenüber menschlichem Versagen, denn falls Fehler begangen werden, können Sie Ihren Algorithmus korrigieren oder fehlerhafte Daten löschen und neu berechnen.

- *Skalierbarkeit* – Sowohl Batch- als auch Serving-Layer sind leicht skalierbar. Beide sind verteilte Systeme und zur Skalierung müssen lediglich weitere Maschinen hinzugefügt werden.
- *Allgemeingültigkeit* – Die beschriebene Architektur ist so allgemein wie möglich gehalten. Sie können beliebige Views beliebiger Datensätze berechnen und aktualisieren.
- *Erweiterbarkeit* – Zum Erstellen einer neuen View muss den Stammdaten lediglich eine neue Funktion hinzugefügt werden. Da der Stammdatensatz beliebige Daten enthalten darf, lassen sich neue Datentypen problemlos ergänzen. Falls Sie eine der Views modifizieren möchten, brauchen Sie sich keine Gedanken über die Unterstützung mehrerer Versionen zu machen, sondern können die gesamte View einfach von Grund auf neu berechnen.
- *Ad-hoc-Abfragen* – Der Batch-Layer unterstützt Ad-hoc-Abfragen von Haus aus. Sämtliche Daten sind bequem von einem Ort aus zugänglich.
- *Minimaler Wartungsaufwand* – Die wichtigste Komponente dieses Systems, die gewartet werden muss, ist Hadoop. Dafür sind gewisse Kenntnisse in der Administration von Computern erforderlich, der eigentliche Betrieb ist jedoch ziemlich einfach. Wie bereits zuvor erwähnt, sind die Datenbanken des Serving-Layers besonders einfach, weil sie keinen wahlfreien Schreibzugriff unterstützen. Da es hier so wenige veränderliche Systembestandteile gibt, kann auch nur sehr viel weniger schiefgehen. Und das hat wiederum zur Folge, dass es sehr unwahrscheinlich ist, dass tatsächlich etwas bei den Serving-Layer-Datenbanken schiefgeht – daher lassen sie sich auch leichter warten.
- *Fehlerbehebung* – Ein- und Ausgaben der Berechnungen finden im Batch-Layer statt. Bei herkömmlichen Datenbanken kann eine Ausgabe die ursprüngliche Eingabe ersetzen – etwa bei der Erhöhung eines Wertes. Bei Batch- und Serving-Layer repräsentieren die Stammdaten die Eingabe und die Ausgabe findet in den Views statt. Auf diese Weise stehen auch die Ein- und Ausgaben der Zwischenschritte zur Verfügung. Diese Ein- und Ausgaben liefern Ihnen sämtliche Informationen, die Sie zur Fehlerbehebung benötigen, falls etwas schiefgeht.

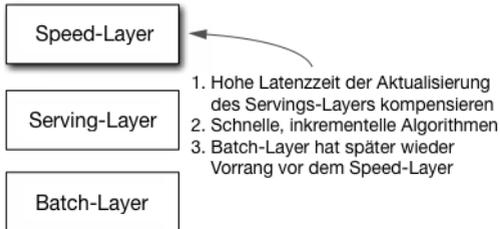
Das Schöne an Batch- und Serving-Layer ist, dass dieser einfache und leicht verständliche Ansatz fast sämtliche erwünschten Eigenschaften besitzt. Man muss sich nicht mit den Problemen der parallelen Ausführung herumplagen und die Skalierung ist trivial. Die einzige Eigenschaft, die fehlt, ist die Aktualisierung mit geringer Latenzzeit. Der letzte der drei Layer, der Speed-Layer, behebt dieses Problem.

### 1.7.4 Speed-Layer

Sobald der Batch-Layer die Berechnung einer Batch-View abgeschlossen hat, wird der Serving-Layer aktualisiert. Das bedeutet, dass die einzigen nicht in der Batch-View vorhandenen Daten diejenigen sind, die während der Berechnung eintrafen. Für ein richtiges Echtzeit-Datenhaltungssystem – das beliebige Funktionen mit beliebigen Eingabedaten in Echtzeit berechnet – fehlen jetzt nur noch die Daten der letzten paar Stunden. Diese Daten bereitzustellen, ist Aufgabe des Speed-Layers. Wie der Name bereits vermuten lässt, soll er gewährleisten, dass neue Daten für Abfragen so schnell verfügbar sind, wie es für das Funktionieren der Anwendung erforderlich ist (siehe Abbildung 1.10).

Der Speed-Layer ähnelt dem Batch-Layer insofern, als dass er anhand eingehender Daten Views erzeugt. Der wesentliche Unterschied besteht darin, dass der Speed-Layer nur die

letzten Daten berücksichtigt, während der Batch-Layer auf sämtliche Daten zurückgreift. Es gibt aber noch einen weiteren großen Unterschied: Um möglichst geringe Latenzzeiten zu erzielen, verarbeitet der Speed-Layer nicht alle neuen Daten gleichzeitig, sondern aktualisiert vielmehr die Echtzeit-Views beim Eingang neuer Daten, statt die Views wie der Batch-Layer von Grund auf neu zu berechnen. Der Speed-Layer führt also inkrementelle Berechnungen durch – keine Neuberechnungen wie der Batch-Layer.



**Abb. 1.10:** Speed-Layer

Der Datenfluss des Speed-Layers kann durch folgende Gleichung formalisiert werden:

Echtzeit-View = Funktion(Echtzeit-View, neue Daten)

Die Aktualisierung einer Echtzeit-View wird anhand der neuen Daten und der bereits vorhandenen View durchgeführt.

Die Lambda-Architektur als Ganzes lässt sich durch die folgenden drei Gleichungen zusammenfassen:

Batch-View = Funktion(Sämtliche Daten)  
 Echtzeit-View = Funktion(Echtzeit-View, neue Daten)  
 Abfrage = Funktion(Batch-View, Echtzeit-View)

Abbildung 1.11 illustriert dieses Konzept. Statt Abfragen nur durch das Ausführen einer Funktion der Batch-View zu beantworten, werden sie sowohl an die Batch- als auch an die Echtzeit-View übermittelt und die Ergebnisse werden zusammengeführt.

Der Speed-Layer verwendet Datenbanken, die wahlfreies Lesen und wahlfreies Schreiben unterstützen. Aufgrund des Supports wahlfreier Schreibzugriffe sind sie sowohl hinsichtlich der Implementierung als auch im Betrieb um Größenordnungen komplexer als die im Serving-Layer eingesetzten Datenbanken.

Das Schöne an der Lambda-Architektur ist: Sobald sich die Daten durch den Batch-Layer bis zum Serving-Layer vorgearbeitet haben, werden die zugehörigen Ergebnisse in den Echtzeit-Views *nicht länger benötigt*. Und das wiederum bedeutet, dass Sie Teile der Echtzeit-Views verwerfen können, wenn sie nicht mehr gebraucht werden. Dies ist ein hervorragendes Ergebnis, denn der Speed-Layer ist erheblich komplexer als die Batch- und Serving-Layer. Wir bezeichnen diese Eigenschaft der Lambda-Architektur als *Komplexitätsisolierung* – damit ist gemeint, dass die Komplexität in einen Layer verlagert wird, in dem Ergebnisse nur temporär vorhanden sind. Falls irgendetwas schiefgeht, können Sie den gesamten Zustand des Speed-Layers verwerfen und nach einigen Stunden ist alles wieder beim Alten.

Fahren wir nun damit fort, das Beispiel einer Web-Analytics-Anwendung weiterzuentwickeln, die die Abfrage der Pageviews innerhalb einer bestimmten Anzahl von Tagen ermöglicht. Der Batch-Layer liefert Views mit der Anzahl der Pageviews anhand eines Schlüssels [url, day].

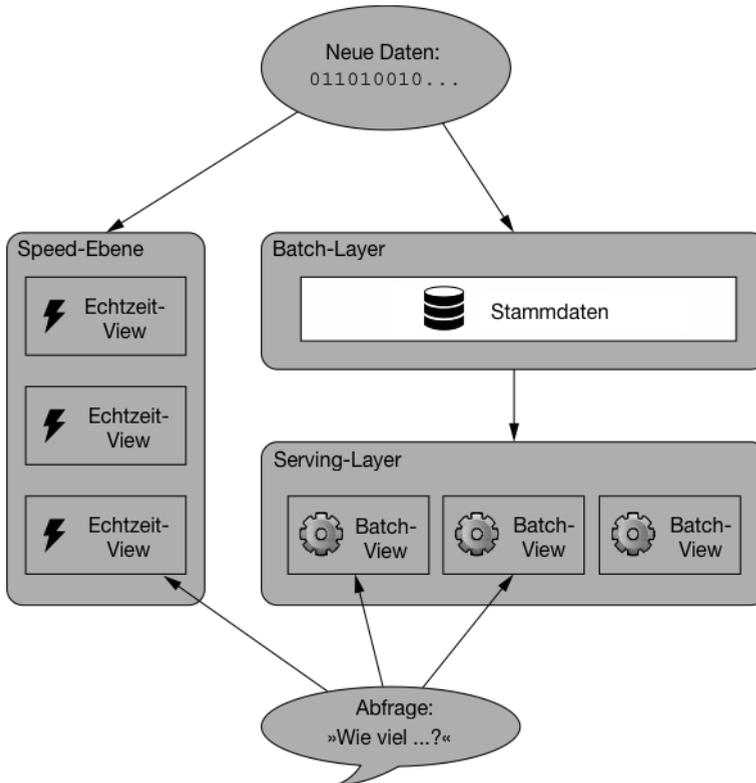


Abb. 1.11: Diagramm der Lambda-Architektur

Der Speed-Layer verfügt über eigene [url, day]-Views. Während der Batch-Layer die Views neu berechnet, indem er die Pageviews buchstäblich einzeln abzählt, aktualisiert der Speed-Layer seine Views durch die Erhöhung eines Zählers, sobald neue Daten eingehen. Zur Beantwortung einer Abfrage fragen Sie Batch- und Speed-Layer ab, um an die Daten über die angegebenen Zeiträume zu gelangen, und summieren die Werte, um das endgültige Ergebnis zu erhalten. Es ist etwas Arbeit erforderlich, um die einzelnen Ergebnisse miteinander abzugleichen, die erledigen wir aber in einem der nachfolgenden Kapitel.

Manche Algorithmen lassen sich inkrementell nur schwer berechnen. Durch die Aufteilung in Batch- und Speed-Layer können Sie im Batch-Layer den exakten Algorithmus und im Speed-Layer eine Näherung verwenden. Der Batch-Layer überschreibt den Speed-Layer immer wieder, sodass die Näherung im Laufe der Zeit korrigiert wird und das System die Eigenschaft einer »letztendlichen Genauigkeit« aufweist. Die Berechnung eindeutig unterschiedlicher Pageviews kann schwierig sein, wenn die Anzahl der Pageviews sehr groß wird. Im Batch-Layer ist die Berechnung einfach, weil auf sämtliche Daten gleichzeitig

zugegriffen werden kann, aber im Speed-Layer müssen Sie womöglich als Näherung auf eine HyperLogLog-Menge zurückgreifen.

Letztlich können Sie auf diese Weise die bestmögliche Performance und Belastbarkeit erzielen. Ein System, das im Batch-Layer exakte Berechnungen und im Speed-Layer Näherungsrechnungen ausführt, wird früher oder später die korrekten Werte liefern, weil der Batch-Layer die im Speed-Layer errechneten Werte nach und nach korrigiert. Zwar finden die Aktualisierungen noch immer mit hoher Latenzzeit statt, aber da die Werte im Speed-Layer nur vorübergehend vorhanden sind, wird die Belastbarkeit der Ergebnisse dadurch nicht beeinflusst. Dank der temporären Natur des Speed-Layers können Sie beim Eingehen von Kompromissen hinsichtlich der Geschwindigkeit sehr aggressiv vorgehen. Und bei Berechnungen, die sich problemlos inkrementell ausführen lassen, liefert das System natürlich immer exakte Resultate.

## 1.8 Die neuesten Trends

Es ist hilfreich, den Hintergrund der in diesem Buch verwendeten Tools ein wenig zu beleuchten. In jüngster Zeit waren einige Trends zu beobachten, die großen Einfluss darauf haben, wie sich die Einrichtung von Big-Data-Systemen gestaltet.

### 1.8.1 Prozessoren werden kaum noch schneller

Allmählich stoßen wir bei der Geschwindigkeit eines einzelnen Prozessorkerns an die physikalischen Grenzen. Falls Sie also größere Datenmengen verarbeiten möchten, müssen Sie Ihre Berechnungen parallel ausführen. Diese Entwicklung hat zum Aufkommen von speziellen, keine gemeinsamen Daten nutzenden parallelen Algorithmen und zugehörigen Systemen wie MapReduce geführt. Statt eine Skalierung durch den Kauf einer schnelleren Maschine zu erzielen (die sogenannte *vertikale Skalierung*), skaliert man die Systeme durch das Hinzufügen weiterer Maschinen (*horizontale Skalierung*).

### 1.8.2 Elastic Clouds

Ein weiterer Trend ist die vermehrte Nutzung von Elastic Clouds, für die auch der Begriff *Infrastructure as a Service* (IaaS) geläufig ist. Amazon Web Services (AWS) dürfte wohl die bekannteste Elastic Cloud sein. Statt vor Ort eigene Hardware zu betreiben, wird sie ganz nach Bedarf angemietet. Auf diese Weise können Sie einen Cluster nahezu augenblicklich vergrößern oder verkleinern. Falls Sie also eine größere Arbeitslast ausführen möchten, können Sie einfach vorübergehend die nötige Hardware mieten.

Elastic Clouds vereinfachen die Systemadministration drastisch. Sie stellen auf Wunsch zusätzlichen Speicherplatz und Rechenleistung zur Verfügung, wodurch sich die Kosten für die Infrastruktur erheblich reduzieren lassen. So bietet etwa Amazon sogenannte Spot-Instances an, für die Sie ein Gebot abgeben können, anstatt einen Festpreis zu zahlen. Werden Sie überboten, erscheint eine Warnmeldung und wenn Sie nicht darauf reagieren, wird Ihre Instanz zwei Minuten später beendet. Da Spot-Instances jederzeit beendet werden können, sind sie erheblich günstiger als normale Instanzen. Für verteilte Systeme wie MapReduce sind sie bestens geeignet, weil die Fehlertoleranz auf Softwareebene gehandhabt wird.

### 1.8.3 Ein lebhaftes Open-Source-Ökosystem für Big Data

Die Open-Source-Community hat in den letzten Jahren eine Fülle von Big-Data-Technologien entwickelt. Alle in diesem Buch vorgestellten Technologien sind quelloffen und können kostenlos genutzt werden.

Sie werden Open-Source-Projekte aus fünf verschiedenen Kategorien kennenlernen. Wie bereits erwähnt, soll dieses Buch keinen Überblick über verfügbare Technologien geben oder einfach nur zeigen, wie sie verwendet werden. Ziel ist es hier, die grundlegenden Prinzipien vorzustellen, damit Sie die Tools selbst bewerten und die für Ihre Anforderungen passenden auswählen können:

- *Stapelverarbeitungssysteme* – Hierbei handelt es sich um Systeme mit hohem Datendurchsatz und hoher Latenzzeit. Stapelverarbeitungssysteme können nahezu beliebige Berechnungen durchführen, benötigen dafür jedoch Stunden oder Tage. Hadoop ist das einzige System aus dieser Kategorie, das wir verwenden. Das Hadoop-Projekt besteht aus den beiden Unterprojekten HDFS (*Hadoop Distributed File System*) und Hadoop MapReduce. HDFS ist ein verteiltes fehlertolerantes Speichersystem, mit dem Daten im Petabyte-Bereich verwaltet werden können. MapReduce ist ein horizontal skalierbares Framework für parallele Berechnungen, das HDFS nutzen kann.
- *Serialisierungs-Frameworks* – Diese Frameworks stellen Tools und Libraries zur Verfügung, um Objekte zwischen unterschiedlichen Sprachen auszutauschen. Objekte werden von irgendeiner Sprache als Bytestream gespeichert, der von einer anderen Sprache deserialisiert wird, die daraus wieder ein Objekt macht. Serialisierungs-Frameworks stellen eine Schema-Definitionssprache bereit, mit der Objekte und deren Datenfelder definiert werden, und bieten Mechanismen zur sicheren Versionierung von Objekten, sodass ein Schema weiterentwickelt werden kann, ohne dass bereits existierende Objekte ungültig werden. Die bekanntesten Serialisierungs-Frameworks sind Thrift, Protocol Buffers und Avro.
- *NoSQL-Datenbanken mit wahlfreiem Zugriff* – In den letzten Jahren wurde eine Vielzahl von NoSQL-Datenbanken entwickelt. Cassandra, HBase, MongoDB, Voldemort, Riak, CouchDB und wie sie alle heißen. Allen diesen Datenbanken ist eins gemeinsam: Sie verzichten auf die von SQL gewohnte Ausdrucksfähigkeit und spezialisieren sich auf bestimmte Operationen. Sie sind *nicht* als allgemeine Datenhaltungssysteme gedacht. In vielerlei Hinsicht ähnelt die Auswahl der zu verwendenden NoSQL-Datenbank dem Auswählen einer Datenstruktur, die in einem Programm eingesetzt werden soll: Hash-tabelle, sortierte Tabelle, verkettete Liste oder Vektor? Sie wissen vorab genau, welche Aufgaben Sie erledigen müssen und wählen dementsprechend aus. Wir werden in der Beispielanwendung, die wir entwickeln, Cassandra verwenden.
- *Benachrichtigungs-/Warteschlangensysteme* – Systeme dieser Art ermöglichen es, asynchron Nachrichten zwischen Prozessen auszutauschen und bieten dabei Fehlertoleranz. Bei der Echtzeitverarbeitung gehört die Nachrichtenwarteschlange zu den entscheidenden Komponenten. Wir werden in diesem Buch Apache Kafka verwenden.
- *Echtzeitberechnungssysteme* – Hierbei handelt es sich um Systeme mit hohem Datendurchsatz und geringer Latenzzeit, die Streamverarbeitung einsetzen. Sie können zwar nicht so umfangreiche Berechnungen wie Stapelverarbeitungssysteme ausführen, verarbeiten Nachrichten jedoch äußerst schnell. Wir verwenden in diesem Buch Storm, denn Storm-Topologien lassen sich leicht erstellen und skalieren gut.

Während diese Open-Source-Programme heranreifen, entstanden im Umfeld einiger Projekte ganze Firmen, die anderen Unternehmen Support leisten. Beispielsweise bieten Cloudera und DataStax Support für Hadoop bzw. Cassandra an. In anderen Fällen werden dagegen Projekte von Firmen beigesteuert. Riak wird beispielsweise von Basho Technologies entwickelt, MongoDB stammt von 10gen und RabbitMQ ist ein Produkt des Unternehmens SpringSource, das zu VMWare gehört.

## 1.9 Beispielanwendung: SuperWebAnalytics.com

Wir werden im weiteren Verlauf des Buches eine Big-Data-Beispielanwendung entwickeln, um die verschiedenen Konzepte zu veranschaulichen. Dabei handelt es sich um eine Anwendung zur Datenverwaltung eines Services, der Ähnlichkeit mit Google Analytics hat. Dieser Service soll in der Lage sein, mehrere Milliarden Pageviews pro Tag nachzuverfolgen.

Er soll in Echtzeit eine Reihe verschiedener Kennzahlen liefern. Diese Kennzahlen können einfache Zähler sein, aber auch komplexe Analysen des Verhaltens der Besucher einer Website umfassen.

Folgende Kennzahlen sollen unterstützt werden:

- *Zeitlicher Verlauf der Anzahl der Pageviews einer URL* – Hiermit können Abfragen wie beispielsweise »Wie viele Pageviews gab es pro Tag im vergangenen Jahr?« oder »Wie viele Pageviews gab es in den letzten 12 Stunden?« beantwortet werden.
- *Zeitlicher Verlauf der Anzahl eindeutig unterschiedlicher Besucher einer URL* – Hiermit können Abfragen wie beispielsweise »Wie viele Leute haben diese Domain im Jahr 2010 besucht?« oder »Wie viele Leute haben diese Domain in den letzten drei Tagen pro Stunde besucht?« beantwortet werden.
- *Berechnung der Bounce-Rate* – »Wie viel Prozent der Besucher haben die Seite besucht, ohne eine andere Seite dieser Website aufzurufen?«
- Wir werden in den folgenden Kapiteln erarbeiten, wie sich Abfragen an die Anwendung speichern, verarbeiten und beantworten lassen.

## 1.10 Zusammenfassung

Sie haben erfahren, was bei der Skalierung relationaler Datenbanksysteme mit herkömmlichen Verfahren wie Sharding alles schiefgehen kann. Die Probleme betreffen nicht nur die Skalierung, denn es wird zunehmend schwieriger, das System zu verwalten, zu erweitern und es überhaupt zu verstehen. Bei der Beschreibung der Einrichtung eines Big-Data-Systems in den folgenden Kapiteln werden wir uns sowohl auf die Belastbarkeit als auch auf die Skalierung konzentrieren. Sie werden sehen, dass beides gleichzeitig erzielbar ist, wenn man die Sache richtig angeht.

Die Vorteile eines gemäß der Lambda-Architektur eingerichteten Systems betreffen nicht nur die Skalierung. Da Ihr System erheblich größere Datenmengen verarbeiten kann, können Sie zusätzliche Daten sammeln und weitere Vorteile aus ihnen ziehen. Mehr Datentypen und ein größeres Datenvolumen eröffnen zusätzliche Möglichkeiten bei der Auswertung der Daten sowie dem Erstellen von Analysen und können zu neuen Anwendungen führen.

Ein weiterer Vorteil der Lambda-Architektur ist die Belastbarkeit Ihrer Anwendungen. Dafür gibt es viele Gründe: Sie können beispielsweise bei Datenmigrationen Berechnungen mit sämtlichen Daten ausführen oder diese korrigieren, falls erforderlich. Sie müssen sich auch nicht mehr mit mehreren verschiedenen zur selben Zeit aktiven Schemata herumplagen. Falls Sie das Schema modifizieren, haben Sie die Möglichkeit, sämtliche Daten zu aktualisieren. Und sollte versehentlich ein falscher Algorithmus auf Produktivdaten angewendet werden, können Sie die fehlerhaften Daten leicht neu berechnen lassen. Sie werden sehen, dass es noch viele weitere Gründe dafür gibt, dass Ihre Big-Data-Anwendungen belastbarer sind.

Auch die Performance ist besser vorhersagbar. Die Lambda-Architektur an sich ist zwar allgemein gehalten und flexibel, aber die einzelnen Komponenten des Systems sind auf bestimmte Aufgaben spezialisiert. Im Gegensatz zu beispielsweise geplanten SQL-Abfragen geschieht nur sehr wenig hinter den Kulissen. Dadurch wird die Performance besser vorhersagbar.

Zerbrechen Sie sich nicht den Kopf, wenn Ihnen einige der gestreiften Themen nicht allzu viel sagen. Wir haben noch ein ziemlich großes Pensum vor uns und die verschiedenen in diesem Kapitel angesprochenen Themenbereiche werden im weiteren Verlauf des Buches noch ausführlicher erläutert. Im nächsten Kapitel erfahren Sie, wie eine Lambda-Architektur eingerichtet wird. Zunächst geht es darum, wie der Stammdatensatz modelliert und schematisiert wird.

# Stichwortverzeichnis

## Symbole

- ! (JCasalog-Datenfeld) 145
- !! (JCasalog-Datenfeld) 145
- ? (JCasalog-Datenfeld) 145

## A

- Abfrage 47
  - schrittweise Ausführung 151
- Abstraktion 143
- ack-Funktion 267
- Ad-hoc-Abfragen 25
- Aggregation 134
- Aggregator 150, 156
  - paralleler 158
- Aggregatorprädikat 146
- Aktualität 330
- Algorithmus
  - allgemeine Anwendbarkeit 117
  - Geburtstagsermittlung 332, 344
  - HyperLogLog 175
  - inkrementeller 114
  - Neuberechnung 114
  - Normalisierung 281
  - Performance 115
  - Punktzahl 111
  - Typ 118
- Allgemeingültigkeit 25
- Amazon Web Services (AWS) 39
- AnalyzeVisits 322
- AnalyzeVisits-Aggregator 187
- Antwortzeit 217
- Apache Thrift 68
- Arbeitslast
  - variierende 255
- Architektur
  - asynchrone 30
  - vollständig inkrementelle 26
- Asynchrone Aktualisierung 254
- atomar 56
- Average 165
- Average-Aggregator 143
- AWS siehe Amazon Web Services

## B

- BaseBasicBolt-Klasse 289
- Batch-Layer 33, 331
  - Anforderungen 35
  - Berechnungen 112
  - Latenzzeit 113
  - mehrere 337
  - Pseudocode 34
  - Ressourcennutzung 338
  - Skalierbarkeit 118
  - Speicherung 75
  - SuperWebAnalytics.com 172
- Batch-View 32, 113, 173
  - Optimierung 209
- Belastbarkeit 24, 336
- Benachrichtigungssystem 40
- Berechnung der Bounce-Rate 209
- BerkeleyDB 234
- Bestätigen 289
- Besucher einer URL 31, 41, 172, 174, 186, 207, 279
  - zeitlicher Verlauf 240
- Bloom-Filter 334
- Bolt 273
- Bounce-Besuche 307
- Bounce-Rate 41, 172, 176, 187, 209, 241, 306, 320
- Buffer 157
- Bytearrays 238

## C

- CAP-Theorem 28, 250, 330
- Cassandra 23, 222, 259
- CassandraState 316
- Chubby 17
- Cluster 79
- combine-Funktion 148, 158
- Combiner-Aggregator 135
- Commit-Log 326
- Composite Columns 263
- Count-Aggregator 150
- CRDT 253

**D**

Dateisystem  
     verteilt 79  
 Daten 47  
     Aktualisierung 52  
     Beständigkeit 48  
     Definition 23  
     Eigenschaften 46  
     Einfachheit 52  
     faktenbasiertes Modell 55  
     fehlerhafte 21  
     Inkonsistenz 60  
     Korrektheit 54  
     löschen 55  
     normalisierte 50  
     Normalisierung 59  
     Roh- 50  
     unstrukturierte 51  
     Unveränderlichkeit 22, 48  
     Ursprünglichkeit 48  
     Vereinigung 148  
     Zusammenführung 251  
 Datenaufbereitung 177  
 Datenaufkommen 17  
 Datenbank  
     Komprimierung 27, 222  
     Replik 29  
 Datenblock 80  
 Dateneingangsrate 342  
 Datenhaltungssystem 109  
     Definition 23, 329  
     erwünschte Eigenschaften 24  
     formale Definition 24  
 Datenknoten 79  
 Datenlokalität 121  
 Datenmenge 17  
 Datenspeicher  
     Anforderungen 76  
 Datenspeicherung  
     Pail 91  
 Datentyp  
     elementarer 68  
 Deduplizierung 177, 184, 204  
 Deltas 337  
 Denormalisierung 219–220  
 Deserialisierung von Objekten 93  
 DistinctCount-Makro 167  
 Domain 210  
 Dublette 57  
 Durchsatz 31, 216  
 Dynamo 17

**E**

Each-Prädikat 168  
 Echtzeitberechnungssystem 40

Echtzeit-Views 37  
     Anforderungen 248  
     Komplexität 249  
 Eigenschaft (Graph) 62  
 Einbindung 159  
 Elastic Cloud 39  
 Eleganz 89  
 ElephantDB 222  
     Cluster 236  
     Grundlagen 233  
     Verwendung 235  
     Views erzeugen 234  
 Equiv 31, 227  
 Equiv-Graph 172, 180  
 Equiv-Kante 179, 279  
 Erweiterbarkeit 25  
 Eventual Consistency 28

**F**

FaceSpace 46  
 fail-Funktion 267  
 Fakt 55  
     Eigenschaften 56  
     Identifizierbarkeit 56  
 Faktenbasiertes Datenmodell 55  
 Fehlerbehebung 26  
 Fehlerhäufigkeit 341  
 Fehlerkorrektur 117  
 Fehlertoleranz 21–22, 24, 30, 52, 116  
 Felder-Gruppierung 275  
 Festplattenausfall 21  
 Filter 130, 134, 154  
 Filterprädikat 146  
 Fixpunkt 180  
 Funktion 154  
 Funktionsprädikat 146

**G**

Garbage Collection 55  
 Geburtstagsermittlung 332, 344  
 Genauigkeit 330  
     letztendliche 38  
 Generatorprädikat 146  
 Geschlecht  
     Vorhersage 110, 124  
 Gesetzliche Vorschriften 75  
 get-Funktion 267  
 getTarget 96  
 Google 17  
 Graphenschema 64  
 groupBy 322  
 Gruppierung 134, 150  
     gemischte 275  
 G-Zähler 253

**H**

Hadoop 9, 23, 79, 87  
 Hash Modding 233  
 Hash Sampling 230  
 Hashfunktion 230  
 Hashwert 20  
 HBase 222  
 HDFS (Hadoop Distributed File System)  
 40, 79, 87  
 Hochverfügbarkeit 28  
 HyperLogLog 117, 175  
 HyperLogLog-Mengen 207

**I**

Idempotenz 278, 302  
 Increment-Operation 169  
 Index 228  
   sortierter 218  
 Information 23, 47  
 init-Funktion 158  
 isKeep-Methode 154  
 isValidTarget 96

**J**

JCascalog 139  
   Datenmodell 144  
   Einführung 144  
   Syntax 146  
 Join 126, 134  
 Join (JCascalog) 147  
 join-Operator 133  
 JSON 63

**K**

Kafka 269  
 Kante (Graph) 62  
 Kennzahlen 41  
 Keyspace 261  
 Key-to-HyperLogLog-Datenbank 226  
 Key-to-Set-Datenbank 225  
 Kleine Dateien 89  
 Knoten (Graph) 62  
 Komplexität 26, 142  
   vertraute 27  
 Komplexitätsisolierung 37  
 Komprimierung 27  
 Konsistenz 28, 250  
   letztendliche 28

**L**

Lambda-Architektur 10, 18, 32, 329  
   formale Definition 37  
 Latenz 216, 329  
 Latenzzeit 25, 119, 247

Laufzeit des Workflows 339  
 Letztendliche Genauigkeit 38, 249  
 Letztendliche Konsistenz 28, 251  
 Limit-Aggregator 163

**M**

Mächtigkeit (Menge) 175  
 Map 103, 259  
 map-Funktion 120  
 Map-Job 334  
 MapReduce 120, 305  
   allgemeine Anwendbarkeit 123  
   Arbeitsweise 125  
   Fehlertoleranz 123  
   Join 126  
   Maschinennähe 126  
   Pageviews 123  
   Skalierbarkeit 121  
 MapState 305  
 Master-Datensammlung  
   SuperWebAnalytics.com 85  
 Memcached 256  
 Menschliches Versagen 30  
 Merge 134  
 Metadaten 315  
 Migration 25  
 Modell für Datenhaltungssysteme 330  
 Modularisierung 143  
 Modulo-Operation 233

**N**

Nachrichtenverarbeitung  
   Gewährleistung 277, 288  
 Namensknoten 79  
 Namenskonflikt 85  
 Nebenläufigkeit 123  
 Neuberechnung  
   teilweise 333  
   vollständige 334  
 Neuberechnungsalgorithmus 114  
 Nimbus 286  
 NoDQL 40  
 Nonce 57  
 Normalisierung 59, 117  
   semantische 50  
 Normalisierung kontra Denormalisierung  
   219  
 NoSQL 17, 22  
 NoSQL-Datenbank 248

**O**

Open-Source-Projekte 40  
 operate-Methode 154  
 Optimierung 209, 220, 295

OrderPreservingPartitioner 263  
out-Methode 146

**P**

Pageviews 31, 41, 57, 110, 124, 172–173,  
185, 204  
  Zeitabschnitte 227  
  zeitlicher Verlauf 223, 237, 305, 317  
  zusammenfassen 218  
Pail 91  
  Dateiformat 97  
  Metadaten 92  
  Operationen 91, 95  
  vertikale Partitionierung 96  
Pail-Library 90  
PailTap 192  
Parallelität 284  
Partial-Makro 169  
partitionBy 322  
Partitionierung  
  horizontale 20  
  vertikale 83  
Partitionstoleranz 250  
Performance 42  
Pig 143  
Pipe-Diagramm 129  
  erweitertes 304  
  Merge 134  
  Wörterzählen 130  
Prädikat  
  benutzerdefiniertes 154  
Prädikat (JCascalog) 146  
Prädikatmakros 164  
  dynamische 167  
Prädikatsoperation 146  
Proprietäre Sprache 142

**Q**

Query-Layer 343

**R**

RandLong-Funktion 164  
RandomPartitioner 263  
RDBMS 15  
reduce-Funktion 120  
Redundanz 219, 258  
Replikant 29, 251  
Replikation 21  
Replikationsfaktor 234  
Rohdaten 50

**S**

Schema 64, 144  
  erzwingbares 68

Schlüssel 259  
Schlüssel-Werte-Datenbank 77  
Schreiben  
  wahlfreies 221  
Schreibzugriff  
  wahlfreier 35  
SequenceFiles 97  
Serialisierung 242  
Serialisierung von Objekten 93  
Serialisierungs-Framework 40, 68  
  Einschränkungen 72  
Serving-Layer 35, 215, 331  
  Anforderungen 35  
  Anforderungen an Datenbank 221  
  Performancekennzahlen 216  
SHA-256 230  
Shard 233  
Sharding 20  
Sharding-Schema 233  
Shredding 192  
Shuffling 122  
Skalierbarkeit 25  
  lineare 119  
Skalierung 22  
  durch Sharding 20  
  horizontale 39  
  mit herkömmlicher Datenbank 19  
  mit Warteschlange 19  
  vertikale 39  
Sloppy Quorums 252  
Snapshot 191  
Sortieren  
  nachrangiges 166  
  Thrift-Datentypen 197  
  zeitlich 310  
Spalte 260  
Spaltenfamilie 259  
Spark 126  
Spark-Streaming 327  
Speed-Layer 36, 343  
  Echtzeit-Views 246  
  formale Definition 37  
  Implementierung 291  
  inkrementelle Berechnung 245  
  SuperWebAnalytics.com 279  
  Zustand 249  
Split-Funktion 130  
Spot-Instance 39  
Spout 273  
  opaker 303  
  transaktionaler 303  
Sprache  
  proprietäre 142  
SQL-Injektion 142

- Stammdatensatz 45
    - Daten hinzufügen 84
    - neue Daten aufnehmen 191
    - Speicherung 81
    - Wachstum 55
  - Stapel
    - eindeutige ID 299
  - Stapelverarbeitung 20
    - inkrementelle 331
  - Stapelverarbeitungssystem 40
  - StateUpdater 304
  - Stichprobe 163
  - Storm-API 283
  - Storm-Cluster 286
  - Storm-Modell 272
  - Störung 251
  - Stream 272
  - Streamgruppierung 275
  - Streamquelle 317
  - Streamverarbeitung 269
    - einzelweise 272
    - in Zeitfenstern 308
    - kleiner Stapel 297, 302
  - struct 68
  - Subqueries
    - dynamische 161
  - Subqueries kombinieren 160
  - Subquery (JCasalog) 146
  - Subquery-Klasse 141
  - Suchvorgang 218
  - Supervisor 287
  - SuperWebAnalytics.com 41
    - Speicherung 99
  - Synchrone Aktualisierung 254
- T**
- Tap 192
  - Tap (JCasalog) 141
  - Task 274
  - Template 165
  - Thrift 68
    - Eigenschaften 70
    - Kante 69
    - Knoten 69
    - Objekt 101
    - Serialisierung 101
  - Tick-Tupel 295
  - Topologie 272, 274, 281
    - Deployment 287
  - TopologyBuilder 283
  - Transaktion 58
  - Trident 313
  - TridentTopology 314
  - Tupel 130, 141, 144, 272
    - eindeutige ID 298
  - Tupel-GAG 278
  - Tupelmenge 133, 144
  - Tweets 111, 125
    - verkettete 162
- U**
- union 68
  - union-Funktion 148
  - URL
    - zeitlicher Verlauf 223
  - URL-Normalisierung 178, 195, 292
  - Ursprünglichkeit 49
  - User-ID-Normalisierung 179, 197
  - UUID 78
- V**
- Verankern 289
  - Verarbeitung
    - genau einmalige 298
    - im Arbeitsspeicher 326
    - Reihenfolge 298
  - Verarbeitungszeit
    - dynamische 339
  - Verfügbarkeit 250
  - Verteiltes Dateisystem 79
    - Funktionsweise 79
  - Verwaltungsaufwand des Workflows 339
  - View 47
  - Vollständig inkrementelle Architektur 225
  - Vorabberechnung 32, 113
- W**
- Wahlfreies Schreiben 221
  - Warteschlange 265
    - Arbeitslast 268
    - zischengeschaltete 277
  - Warteschlangensystem 40
  - Wartungsaufwand 26
  - Wettlaufsituation 191
  - Worker-Prozess 20, 270
  - Workflow 176
    - Vorbereitung 190
  - Wörterzählen 120, 129, 140, 155, 275, 304, 314
- Z**
- Zähler 41, 110
  - Zeitintervall 49
  - Zeitraum 206
  - Zeitstempel 54
  - ZooKeeper 286
  - Zufallszahlengenerator 123
  - Zustand speichern 317