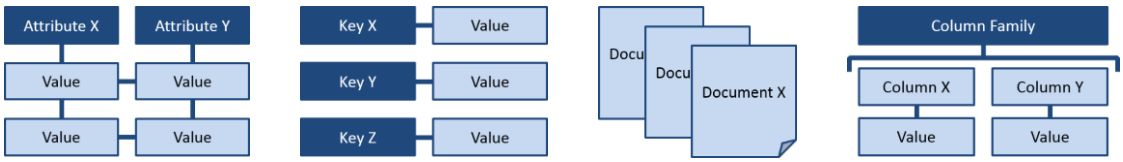


KONZEPTUELLE UND METHODISCHE AUFARBEITUNG VON NOSQL-DATENBANKSYSTEMEN

Robin Hecht

UNIVERSITÄT BAYREUTH

KONZEPTUELLE UND METHODISCHE AUFARBEITUNG VON NOSQL-DATENBANKSYSTEMEN

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Robin Stefan Hecht

aus Berlin

1. Gutachter: Prof. Dr.-Ing. Stefan Jablonski
2. Gutachter: Prof. Dr. Thomas Rauber

Tag der Einreichung: 16. September 2014
Tag des Kolloquiums: 11. Dezember 2014

Für Svenja.

ZUSAMMENFASSUNG

Big Data beschreibt einen derzeitigen Trend in der Informationsverarbeitung, bei dem sehr große Mengen an unterschiedlich strukturierten Daten in einer sehr hohen Geschwindigkeit verarbeitet werden. Die hierbei gestellten Anforderungen überschreiten immer häufiger die Kapazität und die Leistungsfähigkeit von relationalen Datenbanken, die seit Jahrzehnten erfolgreich in der Datenverarbeitung eingesetzt werden. Führende Internetunternehmen sahen sich deshalb zu Beginn dieses Jahrtausends dazu gezwungen, neue Datenbanksysteme zu entwerfen, die auf die spezifischen Anforderungen von Big Data-Anwendungsfällen ausgerichtet sind. Die Konzepte dieser neuen Datenbanksysteme motivierten die Entwicklung einer Vielzahl weiterer, nichtrelationaler Datenbanksysteme, die heute unter dem Namen NoSQL-Datenbanken bekannt sind.

NoSQL-Datenbanken versprechen eine flexible Datenmodellierung, eine hohe Verarbeitungsgeschwindigkeit sowie eine lineare Skalierbarkeit und sind daher Mittelpunkt eines derzeit anhaltenden Hypes um das Thema Big Data. Allerdings existiert kaum neutrale und qualitativ hochwertige Fachliteratur zu dieser Thematik, sodass über die Vor- und Nachteile der einzelnen Datenbanksysteme und deren korrekte Verwendung kaum fachlich fundierten Aussagen vorliegen. Angesichts dieser Problematik ist der Einstieg in die Welt der NoSQL-Datenbanken mit einem unverhältnismäßig hohen Aufwand und einer starken Fehleranfälligkeit verbunden.

Ziel dieser Arbeit ist es, den Einstieg in die komplexe NoSQL-Thematik zu erleichtern und somit die hohen Investitionskosten zu senken, die derzeit mit dem Einsatz eines NoSQL-Datenbanksystems verbunden sind. Durch eine methodische Aufarbeitung der diesen Systemen zugrunde liegenden Konzepte können allgemeingültige Aussagen zu den Vor- und Nachteilen der verschiedenen Datenbanksysteme getroffen und somit das derzeitig vorhandene Fachwissen nicht nur erweitert, sondern vor allem qualitativ aufgewertet werden. Basierend auf den Erkenntnissen der konzeptuellen Aufarbeitung werden darüber hinaus dringend benötigte Modellierungsmuster erstellt, mit deren Hilfe konzeptuelle Datenmodelle systematisch in die logischen Datenmodelle der verschiedenen Datenbanken überführt werden können. Die in dieser Arbeit gewonnenen Erkenntnisse werden abschließend in einem Kriterienkatalog zusammengefasst, mit dem selbst unerfahrene Anwender in die Lage versetzt werden, aus der Vielzahl an derzeit verfügbaren Datenbanksystemen das für einen bestimmten Anwendungsfall ideale System auswählen zu können. Dank der klaren Herausstellung der Vor- und Nachteile der verschiedenen Systeme leistet diese Arbeit einen wertvollen Beitrag zur Versachlichung der NoSQL-Diskussion.

ABSTRACT

Big Data describes a recent trend in information processing whereby a huge amount of differently structured data is processed at a very high velocity. The given requirements this kind of information processing demands, tend to exceed the capacity and performance of relational databases at a more frequent rate, which had successfully been used in data handling for decades. Therefore, at the beginning of the millennium, leading e-commerce businesses felt compelled to develop new database systems, which are aligned to specific requirements of Big Data-use cases. The concept of these new database systems motivated the development of a multitude of non-relational database systems, which are nowadays known as NoSQL databases.

NoSQL databases promise flexible data modeling, high processing speed and linear scalability. Therefore, they are the center of a currently ongoing hype around the topic of Big Data. However, there is hardly any neutral- and specialized literature of high quality for the above-mentioned topic. As a result, there is no qualitative evidence on the advantages and disadvantages of each database system and their correct application. Due to this difficulty, accessing the world of NoSQL databases is associated with a disproportionate effort and a high susceptibility to errors.

The goal of this thesis is to facilitate the access to the complex topic of NoSQL and to reduce the high investment costs, which are currently associated with the implementation of NoSQL databases. Through a methodical reappraisal of the underlying concepts of these systems, universal statements towards advantages and disadvantages of the different database systems can be made and therefore, existing expert knowledge can not only be extended, but especially be enhanced qualitatively. Based on the insights of the conceptual reappraisal, urgently needed modeling patterns are created, whereby conceptual data models can be systematically converted into logical data models for each of the databases. The knowledge, which is gained within this thesis, is summarized conclusively in a set of criteria with which even inexperienced users are enabled to choose the ideal system for a specific use case from the multitude of currently available database systems. Through the clear emphasis on advantages and disadvantages of the different systems, this thesis serves as a valuable contribution towards the objectification of the NoSQL discussion.

DANKSAGUNG

Die hier vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanken und Informationssysteme an der Universität Bayreuth. Ich möchte mich an dieser Stelle ganz herzlich bei allen Personen bedanken, die mich in dieser Zeit bei der Anfertigung dieser Arbeit tatkräftig unterstützt haben.

Mein Dank gilt in erster Linie meinem Doktorvater Prof. Dr.-Ing. Stefan Jablonski, der mir nicht nur die Chance zur Promotion gab, sondern darüber hinaus durch zahlreiche Denkanstöße, Ideen und Ratschläge entscheidend zu dieser Dissertation beigetragen hat. Ganz besonders möchte ich mich für die sehr freundschaftliche Atmosphäre und die hervorragende Unterstützung während meiner Zeit am Lehrstuhl bedanken. Beides ist in dieser Form alles andere als selbstverständlich.

Zudem möchte ich Dr. Olivier Curé für die lehrreichen Diskussionsrunden in Paris und Bayreuth danken, in denen ich erstmalig mit NoSQL-Datenbanken in Berührung kam. Ohne diese konstruktive Zusammenarbeit wäre es nicht zu dem hier vorliegenden Ergebnis gekommen.

Prof. Dr. Thomas Rauber danke ich für die Übernahme des Zweitgutachtens und die kritischen Präsentationsrunden an seinem Lehrstuhl, die mir sehr dabei geholfen haben, die Inhalte und Ergebnisse dieser Dissertation mit wenigen Worten auf den Punkt zu bringen.

Mein besonderer Dank gilt meiner Mutter, die mich immer unterstützte und motivierte und ohne deren Engagement ich nicht in die Lage gekommen wäre, diese Arbeit zu schreiben. Darüber hinaus möchte ich mich ganz herzlich bei Erich für die sehr zeitintensive Korrektur und die zahlreichen konstruktiven Kritikpunkte bedanken, die diese Dissertation ungemein bereichert haben. Nicht zuletzt bedanke ich mich bei Svenja, die nicht nur aufgrund der vielen wertvollen Anregungen aus Blickwinkeln jenseits der Informatik einen großen Anteil an der Fertigstellung dieser Arbeit hat.

INHALTSVERZEICHNIS

Zusammenfassung ..... **i**

Abstract ..... **iii**

Danksagung ..... **v**

Abbildungsverzeichnis ..... **xiii**

Tabellenverzeichnis ..... **xix**

1 Einführung **1**

1.1 NoSQL-Datenbanken 4

1.1.1 Definition 6

1.1.2 Klassifikation 8

1.2 Problemstellung 10

1.2.1 Fehlende Entwicklungsreife 12

1.2.2 Mangelnde Fachliteratur 12

1.3 Ziel der Arbeit 15

1.4 Verwandte Arbeiten 16

1.5 Aufbau der Arbeit 17

2 Relationale Datenbanken **19**

2.1 Einführung 20

2.2 Konzeptuelle Ebene 25

2.2.1 Datenmodell 25

2.2.2 Anfrageverarbeitung 28

2.2.3 Datenmodellierung 32

2.3 Interne Ebene 35

2.3.1 Speicherstrukturen 35

2.3.2 Transaktionen und konkurrierende Zugriffe 39

2.4 Verteilte Datenhaltung 45

2.4.1 Replikation 46

2.4.2 Fragmentierung und Allokation ... 47
2.4.3 Verteilte Anfrageverarbeitung ... 51
2.4.4 Verteilte Transaktionsverwaltung ... 53
2.5 Zusammenfassung ... 56

3 Key Value Stores ... 59
3.1 Einführung ... 60
3.2 Konzeptuelle Ebene ... 61
3.2.1 Datenmodell ... 61
3.2.2 Anfrageverarbeitung ... 63
3.2.3 Datenmodellierung ... 65
3.3 Interne Ebene ... 77
3.3.1 Speicherstrukturen ... 77
3.3.2 Transaktionen und konkurrierende Zugriffe ... 78
3.4 Verteilte Datenhaltung ... 81
3.4.1 Replikation ... 81
3.4.2 Fragmentierung und Allokation ... 81
3.4.3 Verteilte Anfrageverarbeitung ... 84
3.4.4 Verteilte Transaktionsverwaltung ... 86
3.5 Zusammenfassung ... 89

4 Document Stores ... 91
4.1 Einführung ... 92
4.2 Konzeptuelle Ebene ... 93
4.2.1 Datenmodell ... 93
4.2.2 Anfrageverarbeitung ... 96
4.2.3 Datenmodellierung ... 100
4.3 Interne Ebene ... 110
4.3.1 Speicherstrukturen ... 110
4.3.2 Transaktionen und konkurrierende Zugriffe ... 111
4.4 Verteilte Datenhaltung ... 114
4.4.1 Replikation ... 114
4.4.2 Fragmentierung und Allokation ... 115
4.4.3 Verteilte Anfrageverarbeitung ... 116

4.4.4 Verteilte Transaktionsverwaltung ... 118
4.5 Zusammenfassung ... 119

5 Column Family Stores ... 121
5.1 Einführung ... 122
5.2 Konzeptuelle Ebene ... 122
5.2.1 Datenmodell ... 122
5.2.2 Anfrageverarbeitung ... 125
5.2.3 Datenmodellierung ... 127
5.3 Interne Ebene ... 139
5.3.1 Speicherstrukturen ... 139
5.3.2 Transaktionen und konkurrierende Zugriffe ... 142
5.4 Verteilte Datenhaltung ... 144
5.4.1 Replikation ... 146
5.4.2 Fragmentierung und Allokation ... 146
5.4.3 Verteilte Anfrageverarbeitung ... 146
5.4.4 Verteilte Transaktionsverwaltung ... 148
5.5 Zusammenfassung ... 148

6 Kritische Betrachtung von NoSQL ... 151
6.1 Strukturelle Vielfalt ... 152
6.1.1 Relationale Datenbanken ... 154
6.1.2 Key Value Stores ... 159
6.1.3 Document Stores ... 161
6.1.4 Column Family Stores ... 162
6.1.5 Zusammenfassung ... 163
6.2 Hohe Geschwindigkeit ... 166
6.2.1 Benchmarks ... 167
6.2.2 Relationale Datenbanken ... 176
6.2.3 Key Value Stores ... 177
6.2.4 Document Stores ... 178
6.2.5 Column Family Stores ... 179
6.2.6 Zusammenfassung ... 180
6.3 Wachsende Datenmenge ... 183

6.3.1 Benchmarks ... 184
6.3.2 Relationale Datenbanken ... 190
6.3.3 Key Value Stores ... 191
6.3.4 Document Stores ... 192
6.3.5 Column Family Stores ... 194
6.3.6 Zusammenfassung ... 194
6.4 Kritik an NoSQL-Datenbanken ... 197
6.4.1 Fehlende Entwicklungsreife ... 198
6.4.2 Geringer Support ... 199
6.4.3 Geringe Nachhaltigkeit ... 199
6.4.4 Minimale Werkzeugunterstützung ... 199
6.4.5 Mangelndes Fachwissen ... 200
6.4.6 Fehlende Standardisierung ... 200
6.4.7 Zunehmende Komplexität auf Applikationsebene ... 201
6.5 Zusammenfassung ... 202

7 Auswahl eines Datenbanksystems ... 205
7.1 Analyse von Anwendungsfällen ... 206
7.1.1 Key Value Stores ... 206
7.1.2 Document Stores ... 212
7.1.3 Column Family Stores ... 219
7.2 Klassifizierung von Auswahlkriterien ... 226
7.2.1 Konzeptuelle Ebene ... 226
7.2.2 Interne Ebene ... 227
7.2.3 Verteilte Datenhaltung ... 227
7.2.4 Sonstige Kriterien ... 228
7.3 Erstellung eines Kriterienkatalogs ... 229
7.3.1 Identifikation der Kriterien ... 230
7.3.2 Bewertung ... 230
7.3.3 Gewichtung ... 234
7.3.4 Nutzwertberechnung ... 235
7.3.5 Kriterienkatalog ... 235
7.4 Polyglot Persistence ... 237

7.5 Zusammenfassung 239

8 Zusammenfassung, Ausblick und Fazit 241

8.1 Zusammenfassung 241

8.2 Ausblick 243

8.3 Fazit 245

Literaturverzeichnis 247

ABBILDUNGSVERZEICHNIS

1.1: Charakteristische Eigenschaften von Big Data 2
1.2: Klassifikation von NoSQL-Datenbanken 9
1.3: Verlauf des Gartner Hype-Zyklus 10
1.4: Aufbau der Arbeit 18
2.1: Drei-Schichten-Modell nach ANSI/SPARC 20
2.2: Vereinfachte Darstellung eines Prädikatorenschemas 23
2.3: Konzeptuelles Datenmodell eines Online-Shops 24
2.4: Logisches Datenmodell der Entität *Kunde* 26
2.5: Prädikatorenschema der Relation *Kunde* 27
2.6: Operationen der relationalen Algebra 30
2.7: Verarbeitung einer SQL-Anfrage 32
2.8: Logisches Datenmodell eines Online-Shops 34
2.9: Aufbau einer ISAM-Speicherstruktur mit Überlaufseite 36
2.10: Aufbau eines B*-Baums mit der Höhe 3 37
2.11: Konzeptuelles Modell der Entitäten *Warenkorb* und *WarenkorbArtikel* 39
2.12: Aktualisierung des Warenkorbs im Rahmen einer Transaktion 40
2.13: Prinzip des Write Ahead Loggings 41
2.14: Anfrageverarbeitung einer replizierten Master-Slave-Architektur 47
2.15: Anfrageverarbeitung einer funktional partitionierten Architektur 48
2.16: Gegenüberstellung der horizontalen und der vertikalen Fragmentierung. 49
2.17: Reorganisationsaufwand bei statischer Allokation 50
2.18: Vierstufiger Verarbeitungsprozess einer verteilten Leseanfrage 52
2.19: Abbildung einer erfolgreich durchgeführten Transaktion 53
2.20: Abbildung einer fehlgeschlagenen Transaktion 54
3.1: Caching mithilfe von Memcached 60
3.2: Logisches Datenmodell der Entität *Kunde* 62
3.3: Prädikatorenschema des Buckets *Kunde* 63
3.4: Die Operationen *get*, *set* und *delete* 64
3.5: Konzeptuelle Darstellung der Entität *Kunde* 66
3.6: Atomare Abbildung der Entität *Kunde* 67

3.7: Fragmentierte Abbildung der Entität *Kunde* 68
3.8: Konzeptuelles Datenmodell eines Online-Shops 69
3.9: Konzeptuelle Abbildung einer M:N-Beziehung mit zwei 1:N-Beziehungen 70
3.10: Zweistufige Auflösung von referenzierten Adressen 70
3.11: Denormalisierte Abbildung einer Adresse 71
3.12: Konzeptuelle Darstellung der Aggregation *Kunde-Adresse* 72
3.13: Logische Darstellung der Aggregation *Kunde-Adresse* 72
3.14: Identifikation von Teilmengen auf konzeptueller Ebene 74
3.15: Modellierung der identifizierten Teilmengen als Aggregationen 75
3.16: Logische Darstellung der aggregierten Entität *Warenkorb* 76
3.17: Lost Update durch zeitgleiche Aktualisierung 79
3.18: Zeitgleiche Aktualisierung mit Check-and-Set-Funktionen 80
3.19: Allokation mithilfe von Consistent Hashing 82
3.20: Geringer Reallokationsaufwand beim Hinzufügen eines Servers 83
3.21: Verarbeitung von verteilten Leseanfragen 85
3.22: Erfolgreiche Aktualisierung eines Datenobjekts auf zwei Servern 87
3.23: Fehlgeschlagene Aktualisierung eines Datenobjekts auf zwei Servern 87
3.24: Konflikterkennung anhand widersprüchlicher Vektoruhren 88
4.1: Vergleich der Lesbarkeit zwischen JSON und XML 92
4.2: Logisches Datenmodell der Entität *Kunde* 94
4.3: Prädikatorenschema des Dokuments *Kunde* 95
4.4: Formulierung von Restriktions- und Projektionsbedingungen in MongoDB 97
4.5: Formulierung von Restriktions- und Projektionsbedingungen in CouchDB 99
4.6: Konzeptuelle Darstellung der Entität *Kunde* 101
4.7: Abbildung und Anfrage von kompositen Attributen 102
4.8: Konzeptuelles Datenmodell eines Online-Shops 103
4.9: Zweistufige Auflösung von referenzierten Adressen 104
4.10: Konzeptuelle Darstellung der Aggregation *Kunde-Adresse* 105
4.11: Logische Abbildung der Aggregation *Kunde-Adresse* 106
4.12: Identifikation von Teilmengen auf konzeptueller Ebene 108
4.13: Modellierung der identifizierten Teilmengen als Aggregationen 109
4.14: Logische Darstellung der aggregierten Entitäten 110
4.15: Konzeptuelles Modell der Entitäten *Warenkorb* und *WarenkorbArtikel* 111

4.16: Aggregation der Entitäten *Warenkorb* und *WarenkorbArtikel* ... 113
4.17: Abbildung eines MongoDB-Clusters ... 115
4.18: Ablauf einer verteilten MapReduce-Anfrage ... 117
5.1: Logisches Datenmodell der Entität *Kunde* ... 123
5.2: Prädikatorenschema der Column Family *Kunde* ... 124
5.3: Formulierung von Restriktions- und Projektionsbedingungen ... 126
5.4: Konzeptuelle Darstellung der Entität *Kunde* ... 128
5.5: Restriktion mithilfe eines zusammengesetzten Schlüssels ... 129
5.6: Restriktion mithilfe einer Index-Tabelle ... 130
5.7: Konzeptuelles Datenmodell eines Online-Shops ... 131
5.8: Zweistufige Auflösung von referenzierten Adressen ... 132
5.9: Konzeptuelle Darstellung der Aggregation *Kunde-Adresse* ... 133
5.10: Logische Darstellung der Aggregation *Kunde-Adresse* ... 134
5.11: Identifikation von Teilmengen auf konzeptueller Ebene ... 136
5.12: Modellierung der identifizierten Teilmengen als Aggregationen ... 137
5.13: Logisches Datenmodell der aggregierten Entität *Kunde* ... 138
5.14: Aufbau eines SSTables ... 140
5.15: Verarbeitung einer Schreiboperation ... 140
5.16: Verarbeitung einer Leseoperation ... 141
5.17: Konzeptuelles Modell der Entitäten *Warenkorb* und *WarenkorbArtikel* ... 142
5.18: Aggregation der Entitäten *Warenkorb* und *WarenkorbArtikel* ... 143
5.19: Aufbau eines Column Family Store-Clusters ... 144
5.20: Hierarchie der Katalogtabellen ... 147
6.1: Charakteristische Eigenschaften von Big Data ... 151
6.2: Konzeptuelles Datenmodell eines Artikels und seiner Eigenschaften ... 153
6.3: Horizontale Abbildung der Relation *Artikel* ... 154
6.4: Horizontale Abbildung der Entität *Artikel* mit heterogenen Eigenschaften ... 155
6.5: Abbildung der Entitäten *Literatur*, *Artikel* und *E-Book* mittels Vererbung ... 156
6.6: Vertikale Abbildung der Relation *Artikel* ... 157
6.7: Vertikale Abbildung der Relation *Artikel* mit heterogenen Eigenschaften ... 157
6.8: Atomare Abbildung des Buckets *Artikel* ... 159
6.9: Fragmentierte Abbildung des Buckets *Artikel* ... 160
6.10: Darstellung zweier heterogener Artikel ... 162

6.11: Abbildung von heterogenen Artikeln 163
6.12: Aufbau eines YCSB-Datensatzes in einem Key Value Store 169
6.13: Durchsatz und Latenz bei Leseoperationen (Benchmark 1) 170
6.14: Durchsatz und Latenz bei Leseoperationen (Benchmark 2) 171
6.15: Durchsatz und Latenz bei Leseoperationen (Benchmark 3) 171
6.16: Latenzentwicklung bei steigendem Lese-Durchsatz (Benchmark 4) 172
6.17: Latenzentwicklung bei steigendem Lese-Durchsatz (Benchmark 5) 172
6.18: Durchsatz und Latenz bei Schreiboperationen (Benchmark 1) 173
6.19: Durchsatz und Latenz bei Schreiboperationen (Benchmark 2) 173
6.20: Durchsatz und Latenz bei Schreiboperationen (Benchmark 3) 174
6.21: Latenzentwicklung bei steigendem Schreib-Durchsatz (Benchmark 4) 175
6.22: Latenzentwicklung bei steigendem Schreib-Durchsatz (Benchmark 5) 175
6.23: Darstellung der unterschiedlichen Skalierbarkeitsarten 183
6.24: Durchsatz-Scaleup bei Leseoperationen 186
6.25: Antwortzeit-Scaleup bei Leseoperationen 187
6.26: Durchsatz-Scaleup bei Schreiboperationen 188
6.27: Antwortzeit-Scaleup bei Schreiboperationen 189
6.28: Skalierbarkeitseigenschaften von MongoDB (Durchsatz) 192
6.29: Skalierbarkeitseigenschaften von MongoDB (Latenz) 193
6.30: Kritische Fehlermeldung in der CouchDB-Version 1.0 198
7.1: Tweet von Barack Obama zum Wahlsieg 2012 208
7.2: Repräsentation von Benachrichtigungen in Tumblr 209
7.3: Verlauf des wöchentlichen Datenverkehrs bei Gilt 211
7.4: Heterogener Produktkatalog bei CustomInk 213
7.5: Ein typischer Artikel auf der Plattform The Business Insider 214
7.6: Relationales Datenmodell eines Artikels 215
7.7: Datenmodell eines aggregierten Artikels 215
7.8: Datenmodell eines Event-Logging-Systems bei Ebay 220
7.9: Datenmodell zur Speicherung von Benutzerinteraktionen bei Google 221
7.10: Von Facebook Insights bereitgestellte Nutzerstatistiken 222
7.11: Datenmodell von Facebook Insights 223
7.12: Datenmodell von Googles Webcrawler 223
7.13: Datenmodell von Facebook Messages 224

7.14: Die Vor- und Nachteile der unterschiedlichen Datenbankklassen 237

TABELLENVERZEICHNIS

2.1: SQL ist relational vollständig ... 31
2.2: Von SQL unterstützte Operatoren ... 32
2.3: Komponenten des ER-Modells und des relationalen Modells ... 33
2.4: Zugriffskosten der wichtigsten Operationen ... 39
2.5: Isolationslevel des SQL92 Standards und die jeweils vermiedenen Anomalien ... 44
2.6: Funktionsumfang von relationalen Datenbanken ... 56
3.1: Gegenüberstellung der unterschiedlichen Strukturbezeichnungen ... 61
3.2: Unterstützte relationale Operationen und Operatoren ... 65
3.3: Übereinstimmungen zwischen dem ER-Modell und Key Value Stores ... 73
3.4: Performanz bei server- und clientseitiger Koordination ... 85
3.5: Funktionsumfang von Key Value Stores ... 90
4.1: Gegenüberstellung der unterschiedlichen Strukturbezeichnungen ... 93
4.2: Angabe von Datentypen mithilfe von Code-Konventionen ... 94
4.3: Unterstützte relationale Operationen und Operatoren ... 100
4.4: Übereinstimmungen zwischen dem ER-Modell und Document Stores ... 107
4.5: Funktionsumfang von Document Stores ... 120
5.1: Gegenüberstellung der unterschiedlichen Strukturbezeichnungen ... 123
5.2: Unterstützte relationale Operationen und Operatoren ... 127
5.3: Übereinstimmungen zwischen dem ER-Modell und Column Family Stores ... 135
5.4: Funktionsumfang von Column Family Stores ... 149
6.1: Vor- und Nachteile der unterschiedlichen Abbildungsvarianten ... 164
6.2: Konfigurationen der verschiedenen YCS-Benchmarks ... 169
6.3: Gegenüberstellung der Leistungsfähigkeit der Datenbanksysteme ... 181
6.4: Durchsatz-Scaleup bei Leseoperationen ... 186
6.5: Antwortzeit-Scaleup bei Leseoperationen ... 187
6.6: Durchsatz-Scaleup bei Schreiboperationen ... 188
6.7: Antwortzeit-Scaleup bei Schreiboperationen ... 189
6.8: Gegenüberstellung der Skalierbarkeitseigenschaften ... 196
6.9: Vergleich des Funktionsumfangs ... 201
6.10: Eigenschaften der Datenbanksysteme ... 203

7.1: Bewertungsskala 230
7.2: Bewertung der konzeptuellen Ebene 231
7.3: Bewertung der internen Ebene 232
7.4: Bewertung der verteilten Datenhaltung 232
7.5: Bewertung der sonstigen Kriterien 233
7.6: Kriterienkatalog zur Auswahl eines Datenbanksystems 236

1

EINFÜHRUNG

Kapitelinhalt

- Big Data
- NoSQL-Datenbanken
- Problemstellung und Zielsetzung

Der Begriff *Big Data* entwickelte sich in den letzten Jahren zum absoluten Top-Thema in der IT-Welt [29, 66] und ist spätestens seit Bekanntwerden der Überwachungs- und Spionageaffäre der National Security Agency (NSA) auch in den Fokus der öffentlichen Wahrnehmung gerückt. Big Data beschreibt sehr große Datenmengen, die zum Gewinn entscheidungsrelevanter Informationen aus unterschiedlichen Quellen erfasst, gespeichert, analysiert und visualisiert werden können. Diese Informationen können von Unternehmen herangezogen werden, um neue Geschäftsfelder zu identifizieren, Geschäftsprozesse zu optimieren oder um Wettbewerbsvorteile durch eine bessere Kundenbeziehung zu erzielen. In der Forschung können Daten aus Versuchen, Simulationen und Messungen wesentlich umfangreicher als bisher ausgewertet und somit neue Erkenntnisse gewonnen werden. Behörden erhoffen sich dagegen wichtige Informationen zur Kriminalitäts- und Terrorismusbekämpfung. Die Auswertung der in einer zunehmend digitalisierten Welt anfallenden Datenmenge verspricht ein derart großes Gewinnpotenzial, dass um den Begriff Big Data ein regelrechter Hype entstanden ist. Nicht selten ist vom „Öl des Digitalen Zeitalters“ [95] oder dem Ausbruch eines „neuen Goldrausches“ [58] die Rede.

Big Data Analysen unterscheiden sich vom klassischen Data Mining im Business Intelligence in der Datenmenge, der Verarbeitungsgeschwindigkeit und der Datenvielfalt (Abbildung 1.1) [64].

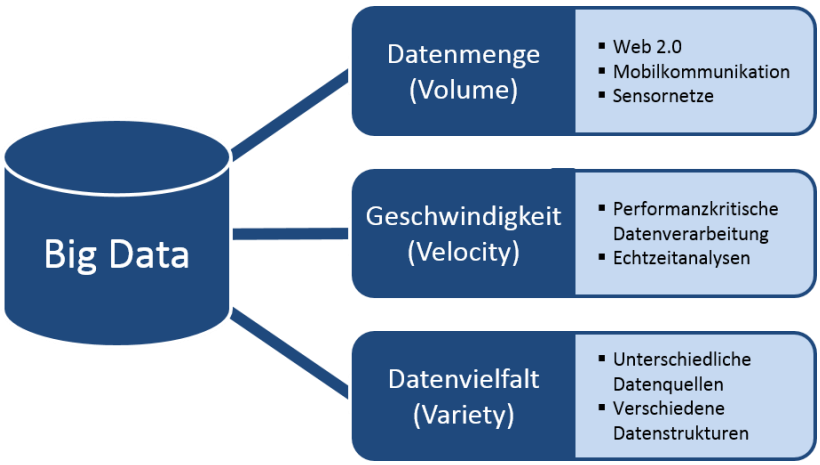


Abbildung 1.1: Charakteristische Eigenschaften von Big Data

Datenmenge

Big Data beschreibt in erster Linie die extrem großen Datenmengen, die in immer mehr Datenbanken von Unternehmen, Instituten und Behörden verarbeitet werden müssen. Durch die rasante Weiterentwicklung und Verbreitung von Informations- und Kommunikationstechnologien ist gegenwärtig ein exponentielles Wachstum [63] dieser Datenmenge zu beobachten. Treibende Faktoren für diese Informationsflut sind unter anderem das Web 2.0, mobile Endgeräte und Sensornetze.

Soziale Netzwerke, Videoplattformen, Onlineversandhäuser, Wikis und ähnliche Web 2.0-Plattformen generieren täglich mehrere Petabyte an Daten. Längst können die Dienste führender Plattformen nur noch unter Zuhilfenahme einer Vielzahl weltweit verteilter Rechenzentren bereitgestellt werden. Die flächendeckende Verbreitung des Internets sowie die Zunahme mobiler Endgeräte ermöglicht einen nahezu permanenten und ortsunabhängigen Zugriff auf diese Plattformen, die aus dem Alltag unserer Informations- und Wissensgesellschaft nicht mehr wegzudenken sind. Darüber hinaus sind zunehmend mehr mobile Endgeräte in der Lage, mithilfe standortbezogener Dienste den realen Alltag mit zusätzlichen Informationen aus dem Internet anzureichern. Die Fortschritte im Bereich der sogenannten *Augmented Reality* (engl. erweiterte Realität) verringern nicht nur die Lücke zwischen realer und virtueller Welt, sie lassen die zu verarbeitende Datenmenge weiter anwachsen. Einen großen Anteil an der gegenwärtigen Datenexplosion hat auch die zunehmende Integration von Sensoren in Alltagsgegenständen, wie beispielsweise Kleidungsstücken oder Haushaltsgeräten. So bieten GPS-Sensoren in Sportschuhen [133] die Möglichkeit, die Laufleistung in Echtzeit zu überwachen und diese Daten anschließend am Computer auszuwerten. Parallel hierzu sorgt die RFID-Technologie gegenwärtig in der industriellen Fertigung [154] und in der Logistikbranche für eine zunehmende Rationalisierung von Geschäftsprozessen. Durch die kontaktlose Übertragung von Paketinformationen können Versandunternehmen immer mehr Schritte im Zustellungsprozess automatisieren und infolgedessen nicht nur Kosten einsparen, sondern auch den Kundenservice dank einer

gestiegenen Transparenz erhöhen. Nach der Vision des *Internet der Dinge* werden Gegenstände in den kommenden Jahren nicht mehr nur passiv, sondern auch vermehrt aktiv mit ihrer Umgebung kommunizieren. Das Potenzial dieser *Smart Objects* ist dabei kaum absehbar und prägt die gegenwärtigen Entwicklungsbereiche Industrie 4.0, Smart Living und vernetzter Autoverkehr [62]. Hinzu kommt die gewaltige Datenmenge, die Sensornetze in wissenschaftlichen Einrichtungen wie CERN oder NASA generieren.

Die beschriebene Entwicklung von Informations- und Kommunikationstechnologien führt zu einem erleichterten Zugang zu Metainformationen, die über einzelne Personen gesammelt werden können. Der sogenannte „Digitale Schatten" [63] eines Menschen wächst durch die Aufzeichnung seines Surfverhaltens im Internet, seiner Bewegungsprofile im Alltag sowie seiner virtuellen und realen Einkäufe. Die mithilfe dieser Daten erstellten personenbezogenen Profile ermöglichen eine Verbesserung des Serviceumfangs von Informations- und Kommunikationssystemen und können von Unternehmen zur Intensivierung ihrer Kundenbeziehungen herangezogen werden. Personenbezogene Profile besitzen jedoch nicht nur im E-Commerce eine steigende Bedeutung, auch Behörden verarbeiten zunehmend mehr personenbezogene Daten, um beispielsweise Straftaten aufklären oder verhindern zu können. Der Informationsgehalt, der durch die Auswertung dieser Daten gewonnen werden kann, ist derart wertvoll, dass immer mehr Metainformationen gesammelt und verarbeitet werden und somit die generierte Datenmenge ständig weiter ansteigt. Der hierbei von Unternehmen und Behörden betriebene Aufwand sowie die hieraus resultierende Datenschutzproblematik rücken verstärkt in den Fokus der öffentlichen Wahrnehmung. Berichte über Google Street View, ACTA oder die NSA zogen nationale und internationale Proteste auf gesellschaftlicher und politscher Ebene nach sich.

Geschwindigkeit

Neben der steigenden Informationsflut, die infolge von Web 2.0 Plattformen, mobilen Endgeräten und Sensornetzen erzeugt wird, ist die hohe Geschwindigkeit, mit der diese Daten gesammelt, verarbeitet und wieder bereitgestellt werden müssen, charakteristisch für Big Data. So müssen führende Internetplattformen wie Twitter und YouTube nicht nur über 347.000 Tweets [97] beziehungsweise über 100 Stunden Videomaterial [185] pro Minute an neuen Daten verarbeiten, sie müssen ihren Nutzern diese Daten anschließend, wie im Beispiel von Twitter, auch „in Echtzeit" zur Verfügung stellen. Fallstudien von Google [22] und Amazon [104] belegen, wie sensibel Benutzer auf geringste Verzögerungen der Webplattformen reagieren. Wenige Millisekunden Latenz reichen demnach aus, um deutliche Rückgänge bei den Benutzerinteraktionen zu registrieren und dadurch einen signifikanten finanziellen Schaden sowie einen Vertrauensverlust bei den Kunden zu verursachen [45]. Ähnlich verhält es sich mit den Ergebnissen, die aus Big Data Analysen gewonnen werden. Wurden im Bereich des Business Intelligence noch vor einiger Zeit Statistiken und Auswertungen von Umsätzen und Benutzerverhalten am Ende eines Tages, Quartals oder Jahres erstellt, wird inzwischen immer häufiger eine Echtzeitanalyse dieser Daten gefordert.

Systeme wie Google Analytics [72] und Facebook Insights [86] ermöglichen es Betreibern einer Webseite, Statistiken über Benutzergruppen, Click-Stream-Analysen und Suchverhalten bereitzustellen, die auf erst wenige Sekunden alten Daten basieren. Hierdurch hofft man frühzeitig Erkenntnisse über Kundenbedürfnisse, Marktentwicklungen und den Erfolg von Werbemaßnahmen zu erhalten, die einen entscheidenden Wettbewerbsvorteil gegenüber der Konkurrenz bedeuten können.

Datenvielfalt

Die dritte typische Eigenschaft von Big Data ist die große Vielfalt der zu verarbeitenden Daten. Neben strukturierten Informationen mit tabellarischem Aufbau, welche die Datenhaltung der vergangenen Jahrzehnte prägten, verarbeiten Informations- und Kommunikationssysteme heutzutage zunehmend mehr sogenannte unstrukturierte beziehungsweise semistrukturierte Daten. Zu diesen zählen benutzergenerierte Inhalte wie Statusmeldungen, Emails und Dokumente, aber auch Logdateien, Sensordaten sowie Bild- und Videomaterialien. So haben sich zum Beispiel die im Bereich des Business Intelligence herangezogenen Daten, mit denen sich im Einzelhandel das Kaufverhalten von Kunden analysieren lässt, mit dem Aufkommen von Onlineshops massiv gewandelt. Wurden früher vorwiegend strukturierte Informationen wie Kundenstammdaten und getätigte Transaktionen erfasst, kombiniert und ausgewertet, werden heutige Kundenprofile mit einer großen Vielfalt von unterschiedlichen, wesentlich persönlicheren Daten angereichert. Im Bereich des Web Analytics werden Log-Dateien, Click-Stream-Daten und Trackinginformationen herangezogen, um zu bestimmen, wie der Kunde auf die Seite gelangt ist, welche Bereiche er angesehen hat und wie er durch Werbung, Bewertungen und Webseitengestaltung beeinflusst wurde [114]. Im Idealfall korrelieren diese Daten mit Kunden-, Surf- und Suchprofilen, die von externen Dienstleistern, wie beispielsweise sozialen Netzwerken, bereitgestellt werden. Die so gewonnenen Daten unterscheiden sich in ihrem Aufbau und ihrer Vielfalt deutlich von den strukturierten Daten, die mithilfe von traditionellen Mitteln, wie personalisierten Kundenkarten und Transaktionsanalysen, gewonnen werden.

1.1 NoSQL-Datenbanken

Der Wandel in der Informations- und Kommunikationsverarbeitung, der durch den Begriff Big Data beschrieben wird, betraf zu Beginn des 21. Jahrhunderts zunächst führende Internetunternehmen wie Google, Amazon und Yahoo sowie soziale Netzwerke wie Facebook, LinkedIn und Twitter. Vor allem die diesen Webseiten zugrunde liegenden Datenbanken, deren Markt bereits seit Jahrzehnten von relationalen Datenbanken dominiert wird, wurden infolge der teilweise extremen Anforderungen an ihre Belastungsgrenzen geführt. Google benötigte beispielsweise ein über mehrere tausend Maschinen verteiltes Datenbanksystem, das in der Lage ist, die gigantische Datenmenge von über sechzig verschiedenen Produkten und Projekten, wie Google Earth,

Google Analytics und dem Google Index, effizient verarbeiten zu können. Da bestehende Datenbanksysteme diesen sehr hohen Ansprüchen nicht gerecht wurden, entwickelte das Unternehmen mit dem Column Family Store *Bigtable* [30] ein eigenes Datenbanksystem, dessen Implementierungskonzept im Jahr 2006 veröffentlicht wurde. Auch Amazon sah sich zu diesem Zeitpunkt mit ähnlichen Herausforderungen konfrontiert. Viele unternehmenskritische Dienste und Daten des Onlineshops, wie Bestsellerlisten, Einkaufswagen, Benutzervorlieben, Verkaufszahlen und Produktkataloge, mussten innerhalb riesiger Cluster verteilt werden. Da auch nur kurzzeitige Ausfälle dieser Dienste massive finanzielle Schäden nach sich ziehen würden, stand vor allem die Hochverfügbarkeit dieser Dienste im Fokus des Interesses von Amazon. Weil sich traditionelle Datenbankenlösungen auch hier als ineffizient erwiesen, entwickelte Amazon den hochverfügbaren Key Value Store *Dynamo* [45] und stellte dessen Konzept im Jahr 2007 vor.

Auch wenn Bigtable und Dynamo bis heute ausschließlich innerhalb ihrer Unternehmen eingesetzt und nicht zur freien Verfügung gestellt wurden, haben die Veröffentlichungen von Google und Amazon für ein hohes Interesse in der Open Source Community gesorgt. Inspiriert von den Bigtable- und Dynamo-Konzepten wurden innerhalb kürzester Zeit neue Datenbanksysteme entwickelt, veröffentlicht und in Unternehmen wie Yahoo [36], Facebook [99], Twitter [172, 174] und LinkedIn [96] eingesetzt. Diese Systeme werden auch als Web-Scale-Datenbanken bezeichnet und haben die Bewältigung der „ungeheuren Datenmengen des Web 2.0-Zeitalters im Terabyte- oder Petabyte-Bereich“ [50] zum Ziel. Aufgrund der steigenden Anforderungen an Informations- und Kommunikationssysteme erhalten diese Datenbanken in zunehmendem Maße auch außerhalb von führenden Internetunternehmen eine erhöhte Aufmerksamkeit, weshalb sich inzwischen der von Johan Oskarsson und Eric Evans im Jahr 2009 eingeführte Begriff „NoSQL“ [153] für diese verteilten, nichtrelationalen Datenbanksysteme etabliert hat. Diese Bezeichnung ist dabei nicht unumstritten und häufig auch irreführend. Einerseits wurde der Begriff bereits 1998 von Carlo Strozzi für eine leichtgewichtige relationale Datenbank ohne SQL-Schnittstelle verwendet [165], andererseits ist nicht die Anfragesprache SQL, sondern das relationale Datenbankmodell Gegenstand der Kritik der NoSQL Bewegung [161]. Auch wenn Strozzi daher die Bezeichnung „NoREL“ für diese neuen nichtrelationalen Datenbanken vorschlug [165], hat sich dennoch die Bezeichnung „NoSQL“ – vermutlich aus absatzwirtschaftlichen Gründen – durchgesetzt.

Innerhalb der NoSQL-Bewegung wird die Verwendung relationaler Datenbanken als Allzweckwerkzeug („One size fits all“ [164]) in der Datenhaltung kritisiert. Ob Apps für Smartphones, Web- und Desktopapplikationen, eingebettete Systeme oder Data Warehouse Systeme, relationale Datenbanken dominieren sämtliche Anwendungsfälle in allen IT-Bereichen der heutigen Zeit. Selbst wenn in bestimmten Anwendungsfällen keine relationalen Daten vorliegen, wird häufig viel Zeit und Energie investiert, um diese Daten mithilfe nicht selten komplexer Transformationen auf das relationale Datenmodell abzubilden. Nach der Philosophie der NoSQL-Bewegung sollten Software-

entwickler in solchen Fällen alternative Datenbankmodelle in Betracht ziehen, die unter Umständen besser zu den Daten und Anforderungen der jeweiligen Anwendungen passen („Use the right tool for the job“) [84]. Dieses Umdenken kann dazu führen, dass eine einzige Anwendung verschiedene Datenbanksysteme einsetzt, die jeweils spezifische Teilanforderungen der Applikation abdecken. In diesem als *Polyglot Persistence* bezeichneten Ansatz ist es durchaus üblich, dass parallel zu NoSQL-Datenbanken auch relationale Systeme eingesetzt werden, weshalb der Begriff „NoSQL“ inzwischen als „Not Only SQL“ interpretiert wird. Diese Interpretation hebt das Potenzial von NoSQL-Datenbanken als Ergänzung zu relationalen Datenbanken in bestimmten Anwendungsfällen hervor und beschreibt somit strenggenommen nicht einzelne Datenbanksysteme, sondern vielmehr die Systemarchitektur, in der diese Datenbanken eingesetzt werden [153].

1.1.1 Definition

NoSQL-Datenbanken fehlt es jedoch nicht nur an einer klaren, unstrittigen Bezeichnung, sondern auch an einer eindeutigen Definition. Da „es weder Gremien noch Organisationen gibt, die sich um eine Begriffsklärung bemüht haben“ [50], ist die Sammlung an Systemen, die sich als NoSQL-Datenbank bezeichnen, entsprechend vielfältig. Diese Datenbanken unterscheiden sich teilweise so stark in ihren Eigenschaften, dass eine klare Definition des Begriffs sowie eine anschließende Klassifikation der Systeme unerlässlich für das weitere Verständnis dieser Arbeit sind.

Kombiniert man die am häufigsten zitierte Definition des NoSQL-Archivs [50] mit der gegenwärtig verfügbaren Fachliteratur [28, 153], ergeben sich in folgenden Kernpunkten Übereinstimmungen:

- Kein relationales Datenmodell,
- Schwache Schemarestriktionen,
- Kein ACID-Transaktionsmodell,
- Einfache Datenreplikation,
- Horizontale Skalierbarkeit.

Diese fünf Eigenschaften zielen auf die gestiegenen Anforderungen von Informations- und Kommunikationssystemen des 21. Jahrhunderts ab [153].

Kein relationales Datenmodell und **schwache Schemarestriktionen:** Die Forderung nach nichtrelationalen Datenmodellen mit schwachen Schemarestriktionen ist auf die steigende strukturelle Vielfalt der zu verarbeitenden Daten zurückzuführen. Dank des Einsatzes von flexibleren Datenmodellen können einerseits heterogene Datensätze, wie sie gerade im Umgang mit semistrukturierten Daten typisch sind [148, S.6], effizienter verwaltet werden und andererseits kann auf strukturelle Änderungen mit einem wesentlich geringeren Aufwand reagiert werden. Diese Vorteile können denen des sauber strukturierten relationalen Datenmodells beispielsweise in der agilen Entwicklung von Webapplikationen überlegen sein, bei der die vergleichsweise häufig auftretenden Schemaänderungen hohe Kosten verursachen können. In solchen Fällen kann ein bewusster Verzicht auf die Vorzüge des relationalen Datenmodells, wie die mächtige Anfragesprache SQL oder das umfangreiche Typsystem, eine durchaus kosteneffizientere Alternative darstellen.

Kein ACID-Transaktionsmodell, **einfache Datenreplikation** und **horizontale Skalierbarkeit:** Der Verzicht auf ein ACID-kompatibles Transaktionsmodell ist den steigenden Geschwindigkeitsanforderungen geschuldet. In Anwendungsfällen, in denen nur sehr einfache Lese- und Schreiboperationen ausgeführt werden und Daten zudem keine hohen Konsistenzanforderungen besitzen, kann eine strikte Umsetzung der ACID-Eigenschaften einen unnötigen Mehraufwand darstellen. Vor allem in verteilten Datenbanksystemen kann in Anbetracht der erhöhten Kommunikationskosten ein Verzicht auf eine strikte Konsistenz sinnvoll sein. Die Forderungen nach einfacher Datenreplikation und horizontaler Skalierbarkeit basiert auf den steigenden Mengen an Anfragen und Daten, die in vielen Anwendungsfällen verarbeitet werden müssen. Im Gegensatz zu relationalen Datenbanken sind NoSQL-Datenbanken von Beginn an für einen Einsatz in verteilten Systemen konzipiert. Diese Systeme sind deswegen in der Lage, ohne den Einsatz zusätzlicher Werkzeuge und ohne Anpassungen auf Applikationsebene, auf mehrere Server verteilt zu werden.

Häufig werden NoSQL-Datenbanken neben den fünf oben genannten Eigenschaften durch weitere Merkmale charakterisiert. So wird die freie Verfügbarkeit dieser Systeme sowie deren einfache, von SQL abweichende Schnittstelle als typisch für NoSQL Systeme angesehen [50]. Da es sich bei Dynamo und Bigtable jedoch um proprietäre Systeme handelt und in den vergangenen Jahren immer mehr NoSQL-Datenbanken mit SQL-ähnlichen Anfragesprachen ausgestattet wurden, sind diese beiden Eigenschaften offensichtlich nicht für alle Systeme bindend. Dennoch fällt auf, dass in NoSQL-Datenbanken einfache, schlüsselbasierte Operationen dominieren [28].

Typisch für NoSQL-Datenbanken ist dagegen deren hoher Spezialisierungsgrad. Angesichts ihrer schlanken Architektur und des geringeren Funktionsumfangs verfügen sie zwar über kein so breites Anwendungspotenzial wie relationale Datenbanken, sie erzielen jedoch wegen ihre Spezialisierung Vorteile bei der Geschwindigkeit und der Verteilung.

1.1.2 Klassifikation

Auch wenn relationale Datenbanken seit Jahrzehnten den Markt dominieren [166], wurden immer wieder alternative Datenbanksysteme entwickelt, die gezielt die Schwächen des relationalen Datenmodells mit neuen Konzepten zu lösen versuchten. Zu den bekanntesten Vertretern gehören Objekt- beziehungsweise objektrelationale Datenbanken, die „seit den 90er Jahren die vielleicht erste ernstzunehmende Alternative“ [50] zu relationalen Datenbanken darstellen, sowie XML- und Graphdatenbanken. Während objektrelationale Datenbanken in der Lage sind, den sogenannten Impedance Mismatch[1] zu beheben, wurden XML-Datenbanken entwickelt, um die primär durch das Internet vermehrt auftretenden semistrukturierten Daten mithilfe von XML-Dokumenten besser verarbeiten zu können. Im Gegensatz hierzu sind Graphdatenbanken auf die Analyse von umfangreichen Beziehungen zwischen Objekten ausgelegt, wie sie bei komplexen Suchanfragen oder Freundschaftsempfehlungen in sozialen Graphen auftreten können.

Bei all diesen Systemen handelt es sich um nichtrelationale Datenbanken mit teilweise schwächeren Schemarestriktionen, welche vorwiegend eine Alternative zum relationalen Datenbankmodell darstellen. Sie genügen jedoch nicht den Leistungsfähigkeitsanforderungen der NoSQL-Definition, da sie größtenteils weder ein von ACID abweichendes Konsistenzmodell noch eine einfache Datenreplikation aufweisen und nicht ohne größeren Aufwand horizontal skalierbar sind [28]. Zusätzlich sind diese Systeme auf eine schnelle und intensive Auflösung von Beziehungen und komplexen Objektverhaltensweisen spezialisiert. Damit unterscheiden sie sich von den überwiegend einfachen, schlüsselbasierten Datensatzanfragen, wie sie für Web-Scale-Datenbanken typisch sind.

Alle gängigen Klassifikationen [28, 50, 74, 84, 101, 111, 170] orientieren sich bei der Gruppierung von NoSQL-Datenbanken an deren Datenmodellen. Auch wenn sich die einzelnen Bezeichnungen sowie die Anzahl der Kategorien in diesen Arbeiten untereinander nicht immer gleichen, kristallisieren sich überwiegend drei Gruppen von NoSQL-Datenbanken heraus, die im Folgenden als *Key Value Stores*, *Document Stores* und *Column Family Stores* bezeichnet werden (Abbildung 1.2).

[1] Der Impedance Mismatch beschreibt ein Problem in der Informatik, das durch den strukturellen Unterschied von Datenobjekten zwischen objektorientierten Programmiersprachen und relationalen Datenbanken entsteht.

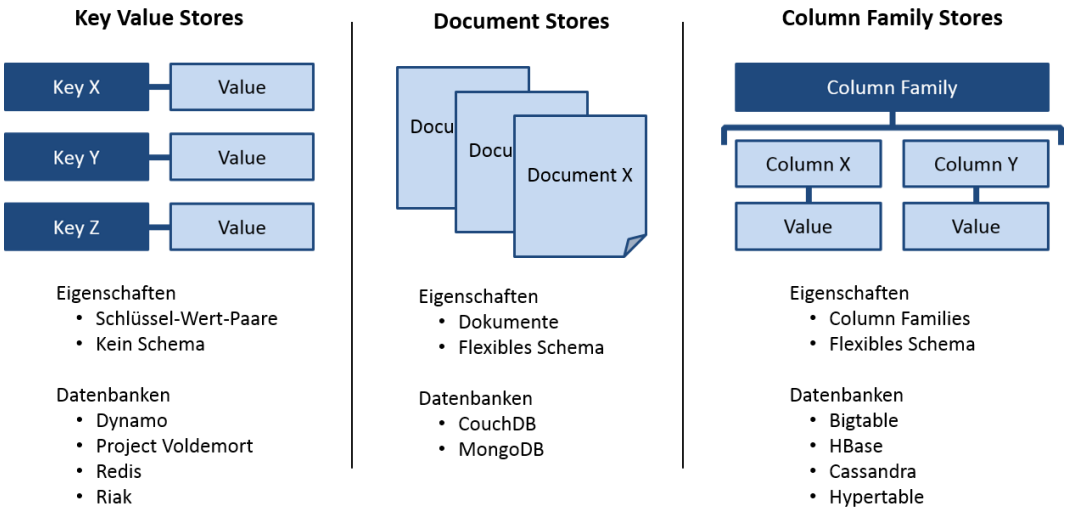


Abbildung 1.2: Klassifikation von NoSQL-Datenbanken

Key Value Stores: Key Value Stores basieren auf einem sehr einfachen Datenmodell, bei dem jeweils ein Schlüssel auf einen einzigen Wert verweist. Da dieser Wert in der Regel beliebige Bytesequenzen annehmen kann, sind Key Value Stores entsprechend vielseitig einsetzbar. Key Value Stores sind in der Lage, einzelne Datensätze sehr effizient und in hoher Geschwindigkeit zu verarbeiten. Der Nachteil dieser Systeme liegt jedoch in ihrem minimalistischen Funktionsumfang. Beispiele für Key Value Stores sind *Dynamo*, *Project Voldemort* und *Redis*.

Document Stores: Diese Systeme verwalten einzelne Datensätze in sogenannten *Dokumenten*. Hierbei handelt es sich um strukturierte Sammlungen von Schlüssel-Wert-Paaren, die überwiegend im ausdrucksstarken JSON-Format abgelegt werden. Aufgrund der daraus resultierenden hohen Flexibilität in der Datenmodellierung und der relativ großen Mächtigkeit der Systeme stellen Document Stores eine Alternative zu relationalen Datenbanken in Anwendungsfällen mit heterogenen Datensätzen dar. Zu den bekanntesten Document Stores gehören *CouchDB* und *MongoDB*.

Column Family Stores: Column Family Stores sind eine Mischung aus Key Value Stores, spaltenorientierten Datenbanken[2] und relationalen Datenbanken [50, S.7]. Ein Datensatz wird durch eine Sammlung von Schlüssel-Wert-Paaren beschrieben. Diese können horizontal angeordnet und daher als Zeile in einer Tabelle dargestellt werden.

[2] Spaltenorientierte Datenbanken speichern ihre Daten physisch nicht zeilenweise, sondern spaltenweise ab. Hierdurch besitzen diese Systeme Vorteile in Anwendungsgebieten, wie beispielsweise Data Warehousing, bei denen häufig nur wenige Attribute von sehr vielen Datensätzen für Berechnungen herangezogen werden.

Schlüssel-Wert-Paare, die in mehreren Datensätzen vorkommen, können dann spaltenweise zu sogenannten *Column Families* gruppiert werden. Column Family Stores wurden für die Verarbeitung sehr großer Datenmengen entworfen und weisen eine hohe Systemkomplexität auf. Die im Laufe dieser Arbeit näher vorgestellten Column Family Stores sind *Bigtable*, *HBase* und *Cassandra*.

1.2 Problemstellung

Das Marktforschungsunternehmen Gartner[3] wählte NoSQL-Datenbanken zu einem der wichtigsten Trends für Informationsinfrastrukturen im Jahr 2013 [66]. Demnach genießen NoSQL-Datenbanken gegenwärtig deren bisher größte mediale Aufmerksamkeit und folgen somit einem fünf Phasen umfassenden Modell [65], das neue Technologien für gewöhnlich durchlaufen (Abbildung 1.3).

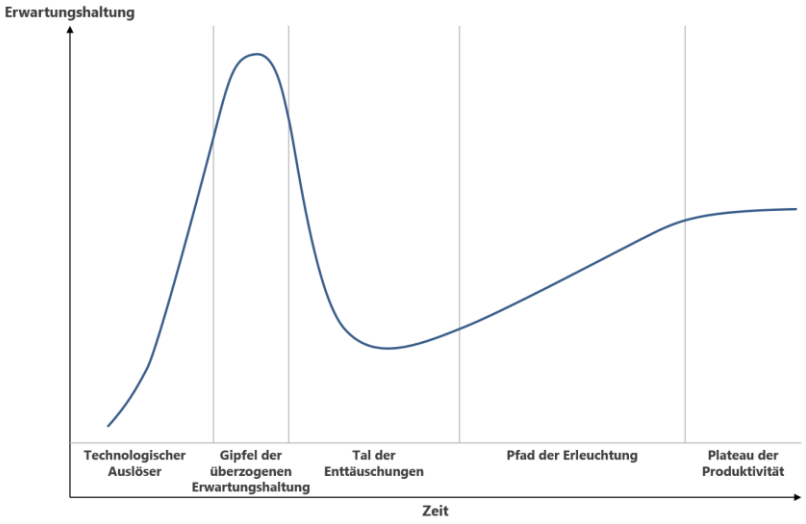


Abbildung 1.3: Verlauf des Gartner Hype-Zyklus

In der ersten Phase („Technologischer Auslöser“) stößt eine erste Veröffentlichung oder ein Projektstart auf ein erhöhtes Interesse in der Fachwelt. Infolge weiterer Veröffentlichungen sowie der Entwicklung anderer, konzeptuell ähnlicher Systeme, rückt die neue Technologie verstärkt in die Wahrnehmung einer breiteren Öffentlichkeit. Eine unsachliche und häufig überstürzte Berichterstattung über erste Erfolgsgeschichten, in denen Fehler und Schwierigkeiten im Umgang mit der neuen Technologie gerne

[3] http://www.gartner.com

verschwiegen werden, sorgen in der zweiten Phase („Gipfel der überzogenen Erwartungen“) für eine übertriebene und unrealistische Erwartungshaltung. Dieser kann die neue Technologie in der dritten Phase („Tal der Enttäuschungen“) nicht gerecht werden. Die hierdurch provozierten Misserfolge führen sehr schnell zu einem starken Rückgang des Interesses und sorgen schließlich dafür, dass die Technologie wieder aus dem Fokus der öffentlichen Wahrnehmung gerät. Nur dank einer konsequenten Weiterentwicklung und Zusammenarbeit mit den nicht vom Hype beeinflussten Nutzern kann die vierte Phase („Pfad der Erleuchtung“) erreicht werden. Hier beginnt sich der wahre kommerzielle Wert der neuen Technologie herauszukristallisieren. Sobald Erfahrungen über die tatsächlichen Vorteile, den Umgang und auch die Grenzen der Technologie bekannt sind und Zweit- und Drittgenerationen der Technik veröffentlicht werden, ist die fünfte Phase („Plateau der Produktivität“) erreicht. Hier entscheidet sich, ob sich die neue Technologie langfristig auf dem Markt durchsetzen oder ein Nischenprodukt bleiben wird [65].

Gartner veröffentlicht jedes Jahr in einem umfassenden Bericht unter anderem eine grafische Darstellung dieser fünf Phasen und kombiniert diese mit einer Einschätzung des gegenwärtigen Status von neuen Technologien („Hype Cycle for Emerging Technologies“). Mithilfe dieses Berichts soll herausgefunden werden, ob es sich bei einer neuen Technologie um einen kurzfristigen Hype handelt oder ob deren Potenzial tatsächlich hoch genug ist, um sich langfristig auf dem Markt zu etablieren.

Auch die NoSQL-Bewegung folgt dieser hier beschriebenen Entwicklung. Die Veröffentlichungen von Google und Amazon in anerkannten Journalen stießen 2006 auf ein großes Interesse in der Fachwelt. Der praktische Einsatz dieser neuen Techniken in allseits bekannten und sehr erfolgreichen Produkten, wie dem Amazon Einkaufswagen oder dem Google Index, animierte die Entwicklung weiterer nicht relationaler Systeme, die seit 2009 unter dem Begriff NoSQL vereint sind (Phase 1). Nachdem in zunehmendem Maße auch andere bekannte Internetunternehmen wie Facebook, Twitter, LinkedIn und Ebay NoSQL-Datenbanken in ihren Infrastrukturen einsetzten und über ihre Erfolge berichteten, stieg das öffentliche Interesse an diesen Systemen sprunghaft an. Inzwischen ist gerade in Kombination mit dem Begriff „Big Data“ ein regelrechter Hype um NoSQL-Datenbanken entstanden, der durch die gegenwärtige Berichterstattung weiter angeheizt wird. Große Datenmengen werden als „Datengold“ [58], „Goldmine“ [31] oder „Öl des Digitalen Zeitalters“ [95] beschrieben und deren Auswertungspotenzial mit dem Ausbruch eines „neuen Goldrausches“ [58] gleichgesetzt.

Laut Gartners Hype-Zyklus können NoSQL-Datenbanken dieser überzogenen Erwartungshaltung nicht gerecht werden. Nach einer Reihe von Misserfolgen wird sich Ernüchterung im Hype-Thema NoSQL-Datenbanken einstellen und das „Tal der Enttäuschungen“ folgen. Diese Einschätzung erscheint durchaus realistisch, da sowohl die enthusiastische Berichterstattung in den Massenmedien als auch die häufig sehr einseitigen Diskussionen in Entwicklerblogs und Konferenzvorträgen zwei gravierende

Problempunkte außer Acht lassen: NoSQL-Datenbanken fehlt es zum einen an Entwicklungsreife und zum anderen an qualitativ hochwertiger Fachliteratur.

1.2.1 Fehlende Entwicklungsreife

NoSQL-Datenbanken weisen gegenüber relationalen Datenbanken einen deutlich geringeren Reifegrad auf. Auch wenn von den meisten NoSQL-Datenbanken inzwischen Produktivversionen existieren, die scheinbar über eine gewisse Systemstabilität verfügen, fällt dennoch die sehr hohe Frequenz auf, in der neue Versionen der verschiedenen Datenbanksysteme veröffentlicht werden. Diese sehr kurzen Entwicklungszyklen sind einerseits auf die mangelnde Anzahl an Funktionalitäten zurückzuführen, die Entwickler für den Einsatz dieser Datenbanken benötigen, und ist andererseits der Instabilität einiger NoSQL-Datenbanken geschuldet, was sich in der relativ hohen Anzahl von Fehlermeldungen in den Mailinglisten der Anbieter widerspiegelt. Die im Vergleich zu relationalen Datenbanken geringere Stabilität und Verlässlichkeit sind selten Bestandteil der gegenwärtigen Berichterstattung und können beim unbedachten Einsatz im produktiven Betrieb sehr schnell zu Misserfolgen führen.

1.2.2 Mangelnde Fachliteratur

Relationale Datenbanken haben sich als Standardlösung in der Datenverarbeitung etabliert. Die Wahl des konkreten Produkts besitzt häufig eine untergeordnete Priorität, da sich diese Systeme nur in wenigen Eigenschaften voneinander unterscheiden. Dank jahrzehntelanger Einsatzerfahrung können Anwender auf umfangreiches Fachwissen zurückgreifen. Dadurch sind sie in der Lage, nahezu jeden Anwendungsfall auf ein relationales Datenmodell abbilden und das System entsprechend seiner Eigenschaften effizient einsetzen zu können. NoSQL-Datenbanken unterscheiden sich in diesen beiden Bereichen von relationalen Datenbanken. Aufgrund fehlender Standards existieren inzwischen über 150 verschiedene Produkte [49], die als NoSQL-Datenbanken bezeichnet werden. Im Gegensatz zu relationalen Datenbanken sind diese Systeme auf spezifische Anwendungsfälle ausgelegt und unterscheiden sich deshalb sehr stark in ihren Eigenschaften. Darüber hinaus können Anwender infolge mangelnder Fachliteratur nicht ansatzweise auf das Wissen zurückgreifen, das über den effizienten Einsatz von relationalen Datenbanken verfügbar ist. Für potenzielle Anwender eines NoSQL-Datenbanksystems ergeben sich deswegen zu Beginn eines Softwareprojekts zwei grundlegende Problemstellungen:

Problem 1 – Die Wahl eines passenden Datenbanksystems

Angesichts der hohen Heterogenität von NoSQL-Datenbanken ist es für Anwender wesentlich schwieriger, das für einen Anwendungsfall passende Produkt auszuwählen. Neben den vielen unterschiedlichen Eigenschaften wird die Wahl auch wegen der hohen Entwicklungsgeschwindigkeit der Systeme erschwert. So kann es vorkommen, dass Artikel über NoSQL-Datenbanken zum Zeitpunkt ihrer Veröffentlichung bereits

wieder veraltet sind. Dieser Umstand wird in einer der ersten umfangreichen NoSQL-Untersuchungen von Cattell [28] deutlich:

> *"Statements in this paper are based on sources and documentation that may not be reliable, and the systems described are „moving targets", so some statements may be incorrect." – Cattell* [28]

Cattel moniert hierbei neben der hohen Entwicklungsgeschwindigkeit vor allem die Qualität der verfügbaren Quellen. Mit den Journal-Publikationen von Google, Amazon und Facebook existieren zwar durchaus anspruchsvolle Artikel, doch stellen diese Veröffentlichungen lediglich zugrunde liegende technische Konzepte einzelner Datenbanksysteme vor. Detaillierte Aussagen über den praktischen Einsatz sowie über die aufgetretenen Schwierigkeiten und daraus resultierende Grenzen werden jedoch weder hier noch in den Dokumentationen der einzelnen Datenbanksysteme getätigt. Die wenigen wissenschaftlichen Veröffentlichungen und die im Handel erhältliche Literatur beschränken sich zum Großteil auf eine Zusammenfassung der aus den Journalen und Dokumentationen verfügbaren Informationen und liefern keine allgemeingültigen Aussagen. Aus diesen Gründen bilden Entwicklerblogs und Erfahrungsberichte häufig die einzige Quelle, um Informationen über den praktischen Einsatz von NoSQL-Datenbanken zu erhalten. Diese Beiträge sind jedoch aus der NoSQL-Bewegung heraus motiviert und lassen neben der gewünschten Qualität auch die nötige Neutralität vermissen. Nicht selten werden relationale Datenbanken mit unsachlichen oder nicht haltbaren Behauptungen belegt, um die vermeintliche Überlegenheit von NoSQL-Datenbanken demonstrieren zu können. Potenzielle Anwender von NoSQL-Datenbanken müssen sich darum nicht nur mit neuen Technologien auseinandersetzen, sondern außerdem in der Lage sein, Aussagen über NoSQL-Datenbanken, die aus dem Hype heraus motiviert sind, von denen zu trennen, die einen fachlich korrekten Informationsgehalt besitzen. Ohne ein tiefgreifendes Verständnis von Datenbanktechnologien ist diese Aufgabe nicht zu bewältigen.

Problem 2 – Der effiziente Einsatz des ausgewählten Datenbanksystems

Für den effizienten Einsatz eines Datenbanksystems sind umfangreiche Kenntnisse über dessen Eigenschaften unabdingbar. Neben den technischen Details des Systems gehören hierzu vor allem die spezifischen Eigenschaften des Datenmodells und die damit verbundenen Anfragemöglichkeiten. Während nahezu jedem Entwickler die relationale Datenmodellierung und die Formulierung von SQL-Statements bestens vertraut sind, ist ein vergleichbares Wissen über den Umgang mit NoSQL-Datenbanken nicht vorhanden. Besonders deutlich wird dieses Problem in der vergleichsweise sehr umfangreichen Dokumentation der beliebtesten [158] NoSQL-Datenbank MongoDB. Zum Thema Datenmodellierung, das für den erfolgreichen Einsatz eines Datenbanksystems essenziell ist, befindet sich trotz mehrjähriger Entwicklungszeit lediglich folgendes Statement:

> *"Document-oriented data modeling is still young. The fact is, many more applications will need to be built on the document model before we can say anything definitive about best practices." – MongoDB* [120]

Wegen des Defizits an allgemeingültigen Abbildungsregeln stehen Entwicklern in der Regel nur die häufig herangezogenen Anwendungsbeispiele von Google [30], Amazon [45], Facebook [99], Twitter [172], Ebay [140, 141], LinkedIn [96] und Digg [135] zur Verfügung, um mögliche Modellierungsmuster identifizieren zu können. Dabei wird jedoch außer Acht gelassen, dass sich aus den spezifischen Lösungen großer Internetunternehmen keine allgemeingültigen Rückschlüsse auf andere, häufig wesentlich kleinere Anwendungsfälle ziehen lassen.

Bedingt durch den hier beschriebenen Mangel an qualitativ hochwertiger Fachliteratur ist der Einstieg in die alles andere als triviale NoSQL-Thematik äußerst schwierig und mit einem sehr hohen Aufwand verbunden. Das gilt insbesondere für Anwender, die keine tiefgreifenden Hintergrundinformationen über die verschiedenen Systeme besitzen und keine umfangreichen praktischen Erfahrungen im Umgang mit NoSQL-Datenbanken vorweisen können. Ihnen ist es nur im Rahmen ausgiebiger Tests möglich, aus der Vielzahl zur Verfügung stehender Datenbanken das für einen bestimmten Anwendungsfall passende System auszuwählen und dieses daraufhin bestmöglich zu verwenden. Entsprechend groß ist die Gefahr, ein NoSQL-Datenbanksystem aus einer falschen Motivation heraus und in einer ineffizienten Art und Weise einzusetzen. Der hier beschriebene Aufwand und das gestiegene Risiko von Fehlentwicklungen sorgen dafür, dass die Investitionskosten, die mit dem Einsatz eines NoSQL-Datenbanksystems verbunden sind, deutlich höher liegen als bei den bewährten relationalen Lösungen. Berücksichtigt man zusätzlich die potenziellen Kosten, welche durch die vergleichsweise geringe Stabilität der Systeme verursacht werden können, erscheint die gegenwärtige Erwartungshaltung an NoSQL-Datenbanken deutlich überzogen. Es wird deswegen nur eine Frage der Zeit sein, bis erste Misserfolge zu einer Ernüchterung des Hypes führen und NoSQL-Datenbanken aus der öffentlichen Wahrnehmung verschwinden werden. Ob sich NoSQL-Datenbanken anschließend auf dem Markt etablieren können, hängt davon ab, wie schnell sie sich zu stabilen Systemen weiterentwickeln und wie schnell sich deren wahrer Nutzen herauskristallisiert.

1.3 Ziel der Arbeit

NoSQL-Datenbanksysteme sind angesichts ihrer fehlenden Entwicklungsreife und der mangelnden Fachliteratur derzeit nicht in der Lage, die hohen Erwartungshaltungen zu erfüllen, die an sie gestellt werden. Auch wenn in einer sehr hohen Entwicklungsgeschwindigkeit neue Datenbankversionen veröffentlicht werden, die einen größeren Funktionsumfang und eine höhere Systemstabilität aufweisen, liegen die Investitionsaufwendungen beim Einsatz von NoSQL-Datenbanken vor allem aufgrund der mangelnden Fachliteratur weiterhin wesentlich höher als bei relationalen Datenbanksystemen. In dieser Problemstellung liegt die Motivation der hier vorliegenden Arbeit.

Ziel dieser Arbeit ist es, den Einstieg in die komplexe Thematik von NoSQL-Datenbanken durch eine methodische Aufarbeitung auf konzeptueller Ebene deutlich zu erleichtern. Dieser neuartige Ansatz bietet die Möglichkeit, bisher verborgene Informationen explizit hervorzuheben und somit das Wissen über NoSQL-Datenbanken nicht nur zu erweitern, sondern auch qualitativ aufzuwerten. Basierend auf den erzielten Ergebnissen sollen allgemeingültige Aussagen über die Vor- und Nachteile der verschiedenen Datenbanksysteme hinsichtlich spezifischer Anforderungen getroffen werden. Außerdem ermöglicht die konzeptuelle und methodische Aufarbeitung die Erstellung von derzeit dringend benötigten Abbildungsregeln, mit deren Hilfe ein bestimmtes NoSQL-Datenbanksystem entsprechend seiner Stärken bestmöglich eingesetzt werden kann. Die hier erzielten Ergebnisse können zu zwei dringend benötigten Hilfsmitteln zusammengefasst werden, die gegenwärtig in der Literatur nicht zu finden sind:

Hilfsmittel 1 – Kriterienkatalog: In dieser Arbeit wird ein Kriterienkatalog entwickelt, der selbst unerfahrene Anwender in die Lage versetzt, aus der Vielzahl zur Verfügung stehender Datenbanksysteme das für einen Anwendungsfall passende System auszuwählen. Die Bewertung für die Eignung eines Datenbanksystems basiert auf einer individuellen Gewichtung verschiedener Auswahlkriterien. Hierdurch können die spezifischen Anforderungen eines jeden Anwendungsfalls feingranular berücksichtigt werden.

Hilfsmittel 2 – Modellierungsmuster: Um einen optimalen Einsatz von NoSQL-Datenbanken zu ermöglichen, werden im Rahmen dieser Arbeit verschiedene Modellierungsmuster erstellt. Mithilfe dieser Modellierungsmuster können Problemstellungen in Anwendungsfällen systematisch auf die spezifischen Datenmodelle der verschiedenen Datenbanksysteme überführt und diese dadurch wesentlich effizienter genutzt werden.

Darüber hinaus leistet diese Arbeit dank der klaren Herausstellung der Vor- und Nachteile der verschiedenen Systeme einen wertvollen Beitrag zur Versachlichung der NoSQL-Diskussion.

1.4 Verwandte Arbeiten

In der oben beschriebenen Herangehensweise unterscheidet sich diese Arbeit grundlegend von anderen Arbeiten, welche die NoSQL-Bewegung thematisieren. Die Veröffentlichungen von Cattell [28] sowie von Hecht und Jablonski [84] zählen zu den ersten Fachartikeln, welche ebenfalls einen Überblick über die Vor- und Nachteile der verschiedenen Datenbanksysteme liefern und unterschiedliche Anwendungsgebiete für NoSQL-Datenbanken vorstellen. Allerdings können diese beiden Arbeiten in Anbetracht des begrenzten Umfangs ausschließlich als Einstiegsliteratur für die NoSQL-Thematik herangezogen werden.

Einen größeren Überblick über die verschiedenen Datenbanksysteme ist in den Fachbüchern von Edlich et al. [50] und Tiwari [170] zu finden. Neben den Vor- und Nachteilen der einzelnen Systeme werden hierin vor allem die technischen Aspekte der Systeme hervorgehoben. Anhand einer Vielzahl an Quelltextauszügen werden Installationsprozesse beschrieben, der Umgang mit den verschiedenen Schnittstellen demonstriert und verschiedene Konfigurationsmöglichkeiten thematisiert. Abgesehen davon, dass diese Beispiele sehr stark von den technischen Eigenschaften bestimmter Datenbankversionen abhängig sind, lassen sich infolge der fehlenden konzeptuellen Arbeit aus diesen Büchern keine allgemeingültigen Rückschlüsse über den Umgang mit NoSQL-Datenbanken ziehen.

Eine Thematisierung von NoSQL-Datenbanken auf konzeptueller Ebene findet sich erstmalig im Fachbuch von Sadalage und Fowler [153]. Anhand eines Anwendungsbeispiels wird hierin ein Modellierungsmuster vorgestellt, mit dessen Hilfe sich Beziehungen zwischen Datensätzen abbilden lassen. Allerdings beschränkt sich die konzeptuelle Arbeit ausschließlich auf eben dieses eine Modellierungsmuster, welches zudem lediglich in einem sehr geringen Umfang behandelt wird. Der komplexe Bereich der Datenmodellierung, der für den erfolgreichen Einsatz eines Datenbanksystems essenziell ist, wird deshalb auch in dieser Arbeit nicht ausreichend abgedeckt.

Neben den hier beschriebenen Arbeiten, welche die NoSQL-Thematik allgemein behandeln, existieren einige Artikel und Fachbücher, die jeweils die spezifischen Eigenschaften eines bestimmten Datenbanksystems exklusiv beschreiben. Hierzu zählen in erster Linie die häufig zitierten Artikel von Google [30], Amazon [45] und Facebook [99] sowie die verfügbaren Fachbücher über die populären NoSQL-Datenbanken CouchDB [9, 103], MongoDB [14, 143] und Cassandra [85], aber auch die vielen offiziellen Dokumentationen zu den einzelnen Systemen. In diesen Quellen werden vor allem die technischen Aspekte der verschiedenen Datenbanksysteme ausführlich beleuchtet. Dabei stehen die bereitgestellten Funktionalitäten, Implementierungsdetails und administrative Informationen im Vordergrund. Auch wenn diese Informationen einen wertvollen Beitrag für die konzeptuelle Aufarbeitung in dieser Arbeit liefern, sind die getätigten Aussagen ausschließlich auf bestimmte Versionen einzelner Datenbanksysteme beschränkt. Allgemeingültige Rückschlüsse und Modellierungsmuster sind darum auch in diesen Arbeiten nicht zu finden. Darüber hinaus ergibt sich bei

diesen Quellen die bereits erwähnte Problematik der fehlenden Neutralität, wodurch vor allem die Wahl nach einem passenden Datenbanksystem sehr stark erschwert wird.

1.5 Aufbau der Arbeit

Diese Arbeit ist in acht Kapitel gegliedert und kann in die zwei Teile konzeptuelle und methodische Aufarbeitung unterteilt werden.

Konzeptuelle Aufarbeitung: Nach einem einführenden Kapitel in die Grundlagen relationaler Datenbanken, die für das Verständnis dieser Arbeit unabdingbar sind, bilden die konzeptuellen Aufarbeitungen von Key Value Stores (Kapitel 3), Document Stores (Kapitel 4) und Column Family Stores (Kapitel 5) den Schwerpunkt dieser Arbeit. In diesen Kapiteln werden die Eigenschaften der verschiedenen Datenbanksysteme methodisch herausgearbeitet, die bei der Wahl und dem Einsatz eines Datenbanksystems von entscheidender Bedeutung sind. Der jeweils erste Teil dieser Kapitel beinhaltet entsprechend der Zielsetzung dieser Arbeit eine konzeptuelle Rekonstruktion der zugrunde liegenden Datenmodelle. Basierend auf diesen Ergebnissen sowie einer Analyse der verschiedenen Anfragemöglichkeiten werden anschließend Abbildungsregeln und Modellierungsmuster erstellt, mit deren Hilfe sich konzeptuelle Datenmodelle auf die verschiedenen logischen Datenmodelle von NoSQL-Datenbanken überführen lassen. Der jeweils zweite Teil dieser Kapitel beschäftigt sich mit den technischen Grundlagen der jeweiligen Systeme. Neben Zugriffspfaden und Transaktionskonzepten liegt der Fokus hierbei auf den verschiedenen Replikations- und Verteilungsstrategien.

Methodische Aufarbeitung: In Kapitel 6 wird das Verhalten von relationalen Datenbanken und NoSQL-Datenbanken in Bezug auf Big Data Anforderungen systematisch untersucht und infolgedessen eine sachliche Bewertung des NoSQL-Hypes ermöglicht. Zu diesem Zweck werden zunächst verschiedene Strategien vorgestellt, mit denen sich heterogene Datensätze (Datenvielfalt) auf die einzelnen Datenmodelle abbilden lassen. Anschließend werden anhand von Benchmarks die Leistungsfähigkeit (Geschwindigkeit) und die Skalierbarkeitseigenschaften (Datenmenge) der wichtigsten Datenbanksysteme untersucht. Das Kapitel schließt mit einer kritischen Auseinandersetzung mit NoSQL-Datenbanken. In Kapitel 7 wird der zur Auswahl eines passenden Datenbanksystems benötigte Kriterienkatalog erstellt. Hierzu werden in einem ersten Schritt entscheidungsrelevante Auswahlkriterien identifiziert und klassifiziert. Anschließend werden die verschiedenen Datenbanksysteme hinsichtlich dieser Kriterien bewertet und tabellarisch gegenübergestellt.

Im abschließenden Kapitel 8 werden die in dieser Arbeit erzielten Ergebnisse zusammengefasst und ein Ausblick auf die zukünftige Entwicklung der NoSQL-Bewegung gewagt.

Ein grafischer Überblick über den Aufbau der Arbeit ist in Abbildung 1.4 dargestellt.

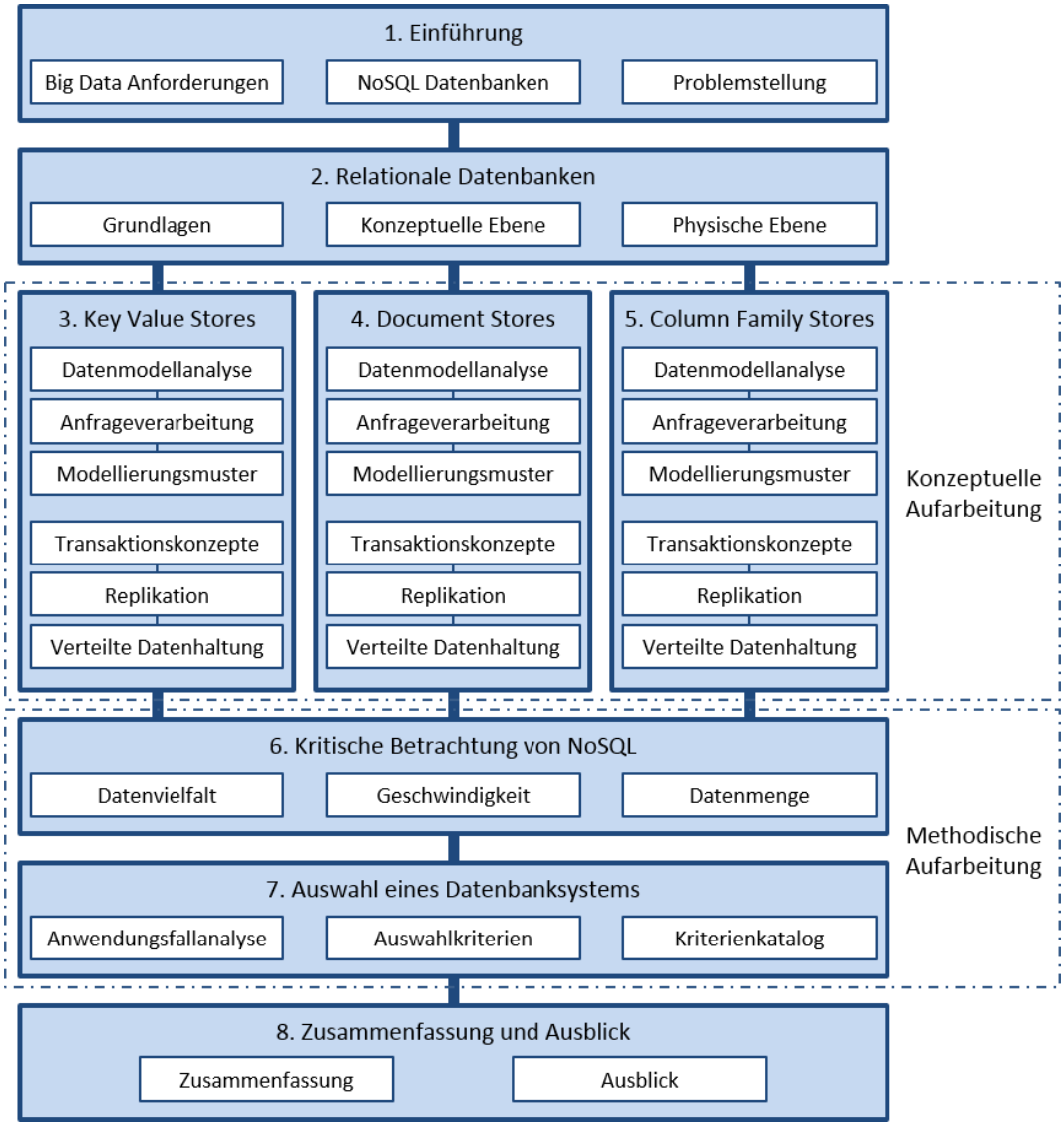


Abbildung 1.4: Aufbau der Arbeit

2

RELATIONALE DATENBANKEN

Kapitelinhalt

- Drei-Schichten-Modell
- Wedekindsches Prädikatorenschema
- Grundlagen von relationalen Datenbanksystemen

Relationale Datenbanken basieren auf dem 1970 von Codd [33] vorgestellten relationalen Datenmodell und haben sich seitdem als de facto Standardlösung im Datenmanagement etabliert. Heute gehören relationale Datenbanken zu den am weitesten verbreiteten und kommerziell am häufigsten eingesetzten Datenbanksystemen [67]. Zu den bekanntesten relationalen Datenbanken zählen *Oracle*[4], *MySQL*[5], *Microsoft SQL Server*[6], *PostgreSQL*[7] und *DB2*[8] [158].

Um die Motivation hinter der NoSQL-Bewegung nachvollziehen zu können, ist eine nähere Betrachtung von relationalen Datenbanken hinsichtlich der im vorherigen Kapitel vorgestellten BigData-Anforderungen unabdingbar. Nach einer Einführung in grundlegende Aspekte der Datenbanktheorie, die für das Verständnis dieser Arbeit zwingend notwendig sind, werden in diesem Kapitel zunächst die Konzepte relationaler Datenbanken vorgestellt, die für die BigData-Eigenschaft *Datenvielfalt* relevant sind. Hierzu zählen das relationale Datenmodell, die Anfragesprache SQL und Konzepte der Datenmodellierung. Der zweite Teil dieses Kapitels setzt sich mit den technischen Eigenschaften von relationalen Datenbanken auseinander, um eine sachliche Diskussion bezüglich der BigData-Eigenschaft *Geschwindigkeit* zu ermöglichen. Bestandteil dieses Abschnitts sind Zugriffspfade und verschiedene Konzepte der Transaktionsverwaltung. Abschließend werden Konzepte verteilter Datenbanksysteme vorgestellt, welche die BigData-Eigenschaft *Datenmenge* betreffen. Neben verschiedenen

[4] http://www.oracle.com/database
[5] http://www.mysql.com
[6] http://www.microsoft.com/sql-server
[7] http://www.postgresql.org
[8] http://www.ibm.com/db2

Replikationstechniken wird dabei auf die Fragmentierung und Allokation von Relationen sowie auf die zugrunde liegenden Konzepte verteilter Anfrageverarbeitung und verteilter Transaktionsverwaltung eingegangen.

2.1 Einführung

Relationale Datenbanken dominieren seit Jahrzehnten den Datenbankmarkt und sind auch heute noch die kommerziell am häufigsten eingesetzten Datenbanksysteme [66]. Die Grundlagen für diesen Erfolg bilden wissenschaftliche Arbeiten aus den 1970er Jahren. Zu den wichtigsten zählen hierbei die Veröffentlichungen zum relationalen Datenmodell von Codd [33, 34]. Parallel entwickelte Bayer die für Zugriffspfade essenziellen B-Bäume [115]. Später folgte das Entity-Relationship-Modell von Chen [142], das noch heute zur konzeptuellen Datenmodellierung verwendet wird. Einen weiteren Meilenstein in der Datenbankgeschichte bildet das von IBM entwickelte System R [53], das als erstes relationales Datenbanksystem gilt und bereits Transaktionskonzepte und die deskriptive Anfragesprache SEQUEL[9] anwendete [55, S.33].

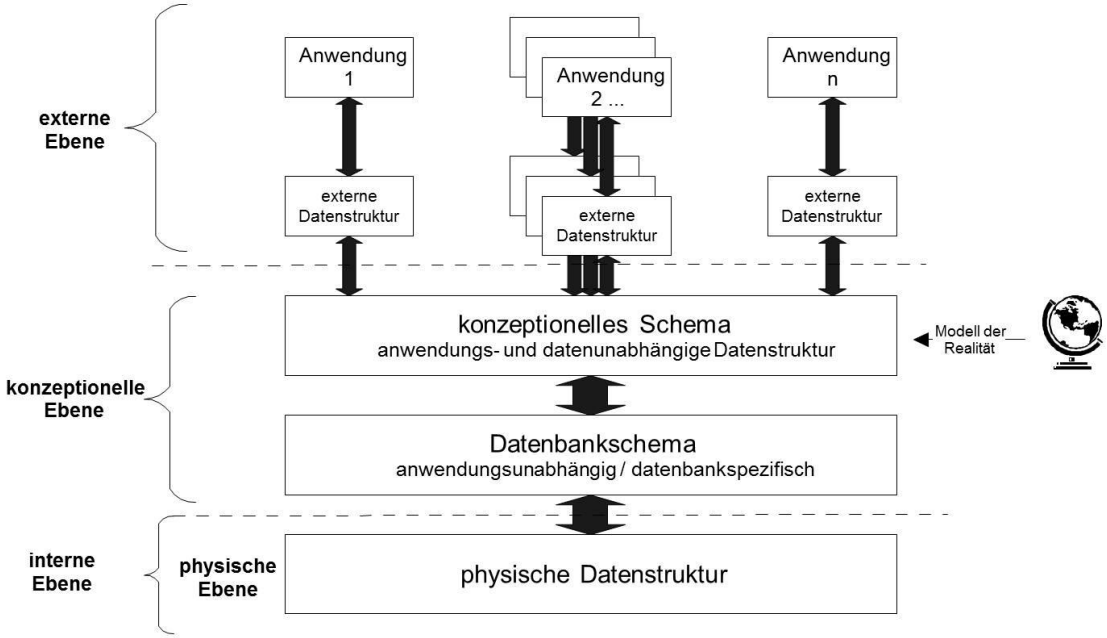


Abbildung 2.1: Drei-Schichten-Modell nach ANSI/SPARC [55]

[9] SEQUEL: Structured English Query Language

Auch wenn bis heute viele verschiedene Datenbanksysteme mit teilweise unterschiedlichen Datenmodellen entwickelt wurden, basieren nahezu alle Systeme auf dem in Abbildung 2.1 dargestellten Drei-Schichten-Modell, das 1975 von einer Arbeitsgruppe des ANSI/SPARC[10] vorgestellt wurde. Das Drei-Schichten-Modell ermöglicht mithilfe verschiedener Beschreibungsebenen unterschiedliche Sichten auf die in einer Datenbank enthaltenen Informationen. Man unterscheidet zwischen externer Ebene, konzeptueller Ebene und interner Ebene. Die einzelnen Beschreibungen innerhalb der Ebenen werden als Schema bezeichnet [76, S.9].

Externe Ebene: Auf externer Ebene werden individuelle Sichten auf die in einer Datenbank verfügbaren Informationen definiert. Hierdurch können exakt die Daten bereitgestellt werden, die eine Benutzergruppe oder Anwendung zur Bearbeitung einer bestimmten Aufgabenstellung benötigt. Neben der Reduzierung der Verarbeitungskomplexität und dem gerade in größeren Systemen essenziellen Aspekt der Rechtevergabe auf ausgewählte Daten kann auf spezifische Anforderungen, wie beispielsweise die Bereitstellung eines speziellen Fachvokabulars, eingegangen werden.

Konzeptuelle Ebene: Zentraler Bestandteil des Drei-Schichten-Modells ist die konzeptuelle Ebene (in Abbildung 2.1 konzeptionelle Ebene genannt). Diese bietet eine Gesamtansicht auf alle in der Datenbank hinterlegten Informationen. Innerhalb der konzeptuellen Ebene wird zwischen dem konzeptuellen und dem logischen Schema unterschieden. „Das konzeptuelle Schema spiegelt die Informationen über Sachverhalte, Beziehungen und Vorgänge der betrachteten Miniwelt wieder, die im Datenbanksystem durch Daten repräsentiert werden sollen“ [76, S.10]. Üblicherweise werden hierfür anschauliche Techniken wie das Entity-Relationship-Modell (ER-Modell) oder die Unified Modeling Language (UML) eingesetzt, welche „von den meisten Menschen intuitiv erschlossen werden“ [67, S.198] können. Das logische Schema entspricht einer Abbildung des plattformunabhängigen konzeptuellen Schemas auf die plattformspezifischen Regeln des eingesetzten Datenbanksystems.

Interne Ebene: Die interne Ebene umfasst Informationen über die „Art und den Aufbau der Datenstrukturen auf dem physikalischen Speicher“ [55, S.44] sowie über Datenzugriffsmechanismen. Auf dieser Ebene können technische Maßnahmen zur Laufzeitverbesserung, Verfügbarkeit und Sicherheit durchgeführt werden. Diese Ebene ist abhängig vom konkreten Datenbanksystem und kann sich daher produktabhängig teilweise erheblich unterscheiden.

Die drei Ebenen sind sowohl physisch als auch logisch voneinander unabhängig, weshalb Änderungen einer Ebene keine oder nur minimale Auswirkungen auf die darüber liegende Ebene nach sich ziehen. So darf das Hinzufügen eines Index auf der internen

[10] ANSI: American National Standards Institute; SPARC: Standards Planning and Requirements Committee

Ebene keinen Einfluss auf das logische Datenbankmodell der konzeptuellen Ebene haben und minimale Änderungen auf konzeptueller Ebene, wie die Umbenennung eines Attributs, sollten keine Änderungen bereits existierender Anwendungen auf der externen Ebene veranlassen [94, S.22–23].

Da ausschließlich die interne Ebene die tatsächlichen Daten enthält, muss ein Datenbanksystem sämtliche von Benutzern und Anwendungen formulierten Anfragen in Operationen der konzeptuellen Ebene und hiernach in Operationen der internen Ebene übersetzen. Anschließend müssen die Daten im Fall von Leseoperationen wieder vom internen Schema bis zum externen Schema durchgereicht und entsprechend aufbereitet werden. „Diese Vorgänge können sehr zeitaufwendig sein, sodass einige DBMS, insbesondere solche, die für die Unterstützung kleiner Datenbanken ausgelegt wurden, keine externen Sichten unterstützen“ [52, S.51].

Im Rahmen des Datenbankentwurfs werden die einzelnen Schemata des Drei-Schichten-Modells mittels einer allgemeingültigen Methodologie systematisch erstellt. Diese umfasst je nach Definition mehrere aufeinander aufbauende Prozessschritte, die sequenziell ausgeführt werden [52, 55, 94]. In ihrem Kern beinhaltet jede Definition vier essenzielle Entwurfsphasen:

Anforderungsanalyse: Der Datenbankentwurf beginnt mit der Beschreibung des Ausschnitts der „realen Welt“, der in der Datenbank abgebildet werden soll [94, S.24]. Hierzu werden sämtliche Informationsanforderungen und Datenverarbeitungsvorgänge der sogenannten Miniwelt in einem Pflichtenheft spezifiziert.

Konzeptueller Datenbankentwurf: Basierend auf den Informationsanforderungen des Pflichtenhefts wird das konzeptuelle Schema entworfen. Hierbei handelt es sich um eine „manuelle / intellektuelle Modellierung“ [94, S.25], bei der „möglichst genau alle Objekte, Gegenstände und Beziehungen“ [55, S.71] der Miniwelt abgebildet werden.

Logischer Datenbankentwurf: Das konzeptuelle Schema wird anschließend in das logische Schema der eingesetzten Datenbank überführt. Dieser Prozess kann mithilfe „halbautomatischer Transformationen“ [94, S.25] erfolgen, bei denen Modellierungswerkzeuge einen ersten Entwurf des logischen Schemas generieren. Dieser wird häufig in weiteren Schritten manuell angepasst, um die im konzeptuellen Schema nicht enthaltenen Informationen, wie beispielsweise die im Pflichtenheft definierten Datenverarbeitungsvorgänge, zu berücksichtigen.

Physischer Datenbankentwurf: Abschließend werden die physischen Speicherstrukturen, die Datensatzanordnung und Zugriffsmechanismen spezifiziert, um eine Effizienzsteigerung zu erzielen. Hierzu ist eine „detaillierte Kenntnis des zugrunde liegenden Datenbanksystems, aber auch der Hard- und Software […] notwendig“ [94, S.33].

Grundlegender Bestandteil jedes Datenbankentwurfs ist die Definition der Informationsanforderungen der betrachteten Miniwelt. Hierzu werden gleichartige Objekte (Entitäten) und Beziehungen der realen Welt zu Mengen zusammengefasst. Eine Menge wird durch bestimmte Eigenschaften (Attribute) charakterisiert. Alle konkreten Entitäten oder Beziehungen einer Menge besitzen spezifische Ausprägungen dieser Eigenschaften (Attributwerte) und können mithilfe von Schlüsselattributen in dieser Menge eindeutig identifiziert werden.

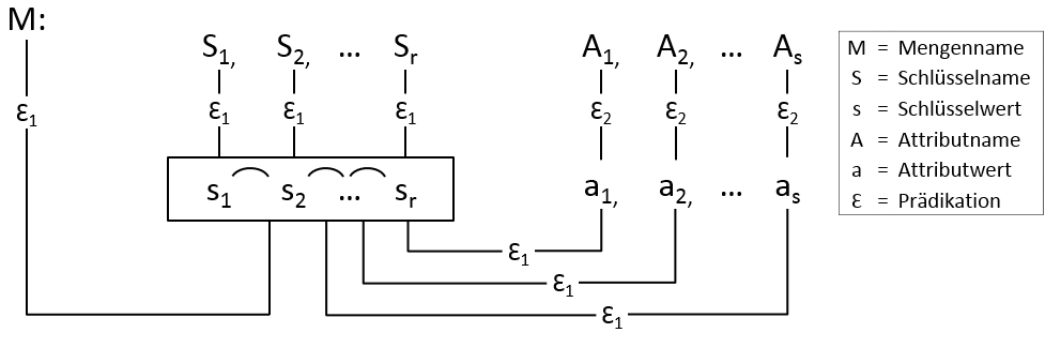


Abbildung 2.2: Vereinfachte Darstellung eines Prädikatorenschemas

Die hier beschriebenen Abhängigkeiten zwischen den einzelnen Begriffen können anhand des in Abbildung 2.2 dargestellten Prädikatorenschemas [181, S.101] veranschaulicht werden. Dabei repräsentiert ein Rechteck eine konkrete Entität der Miniwelt. Dieser Entität können im Rahmen einer Prädikation erster Ordnung (ε_1) verschiedene sogenannte Prädikatoren zugesprochen werden. Zu diesen zählen ein Eigenprädikator (Mengennamen), eine bis mehrere Konnotationen (Schlüsselnamen) sowie mehrere Zusatzprädikatoren (Attributwerte). In einer Prädikation zweiter Ordnung (ε_2) können *optional* Zusatzprädikatoren mittels Apprädikatoren (Attributnamen) näher beschrieben werden. Um die fachliche Diskussion in dieser Arbeit zu vereinfachen, werden im Folgenden die hier verwendeten Termini der Orthosprache [108, S.24] durch die in den jeweiligen Klammern verwendeten Begriffe ersetzt und das Wedekindsche Prädikatorenschema gemäß der Darstellung in Abbildung 2.2 angepasst. Darüber hinaus wird auf eine Unterscheidung zwischen Entitäten und Entitätsmengen sowie Beziehungen und Beziehungsmengen verzichtet. Auch wenn Prädikatorenschemata heutzutage bei der Datenmodellierung kaum noch Beachtung finden, werden sie dennoch im Laufe dieser Arbeit immer wieder aufgegriffen, da sich mit ihnen die Vor- und Nachteile der verschiedenen Datenmodelle sehr gut veranschaulichen lassen.

Sobald die Entitäten, Beziehungen und deren charakteristische Attribute identifiziert wurden, kann das konzeptuelle Datenmodell der abzubildenden Miniwelt modelliert werden. Hierfür werden in der Regel das Entity-Relationship-Modell (ER-Modell) oder

die Unified Modeling Language (UML) herangezogen. Unabhängig von der verwendeten Notationsform werden Entitäten durch Rechtecke und Beziehungen als Linien zwischen den entsprechenden Entitäten dargestellt. Attribute werden innerhalb der Rechtecke notiert. Die so erstellten Modelle enthalten keine Implementierungsdetails und sind darum relativ leicht verständlich. Sie bieten sich deswegen zur Kommunikation mit den in der Regel nicht datenbankversierten Anwendern an.

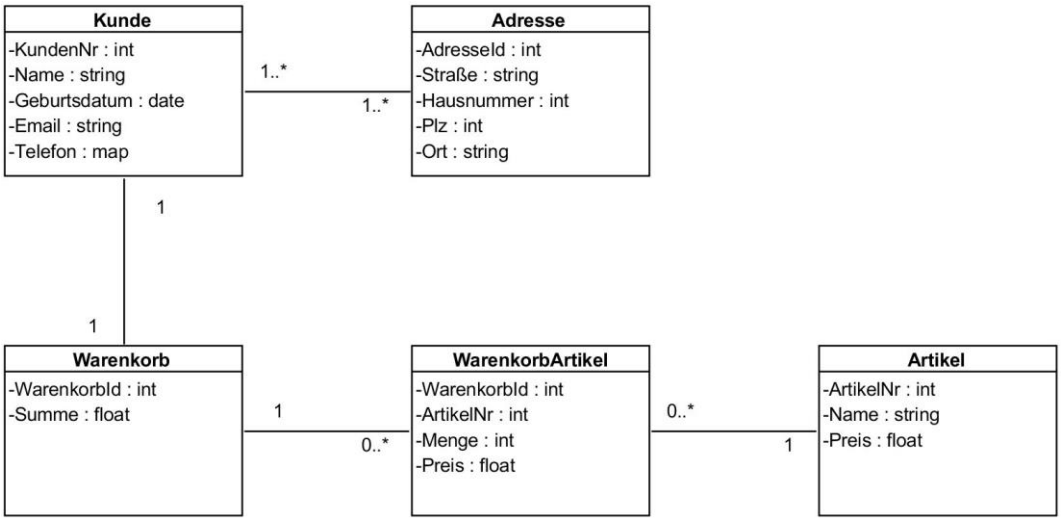


Abbildung 2.3: Konzeptuelles Datenmodell eines Online-Shops

Abbildung 2.3 beschreibt ausschnittsweise das konzeptuelle Datenmodell eines Onlineshops. Hierzu wurden die fünf Entitäten *Kunde*, *Adresse*, *Warenkorb*, *WarenkorbArtikel* und *Artikel* identifiziert. Jede Entität wird durch bestimmte Attribute näher beschrieben. Zwischen den einzelnen Entitäten bestehen verschiedene, binäre Beziehungen, die mithilfe sogenannter Kardinalitäten genauer spezifiziert werden. Man unterscheidet drei typische Arten von Beziehungen:

1:1-Beziehungen: Einer Entität E_1 ist maximal eine Entität E_2 zugeordnet und eine Entität E_2 kann zu maximal einer Entität E_1 eine Beziehung besitzen. Eine 1:1-Beziehung besteht im oben abgebildeten Beispiel zwischen der Entität *Kunde* und der Entität *Warenkorb*.

1:N-Beziehungen: Einer Entität E_1 sind beliebig viele Entitäten E_2 zugeordnet. Gleichzeitig kann eine Entität E_2 zu maximal einer Entität E_1 eine Beziehung besitzen. 1:N-Beziehungen bestehen in Abbildung 2.3 zwischen den Entitäten *Warenkorb* und *WarenkorbArtikel* sowie zwischen *WarenkorbArtikel* und *Artikel*.

M:N-Beziehungen: Einer Entität E_1 sind beliebig viele Entitäten E_2 zugeordnet und eine Entität E_2 kann Beziehungen zu beliebig vielen Entitäten E_1 besitzen. Eine M:N-Beziehung besteht zwischen der Entität *Kunde* und der Entität *Adresse*.

In der in Abbildung 2.3 beschriebenen Miniwelt besitzt ein Kunde genau einen Warenkorb und ein Warenkorb ist genau einem Kunden zugewiesen. Jeder Kunde hat mindestens eine Adresse. Unter einer Adresse muss mindestens ein Kunde erreichbar sein. Einem Warenkorb kann eine beliebige Anzahl von Artikeln zugeordnet werden. Befindet sich ein Artikel in einem Warenkorb, werden die Menge und der Preis für diese Bestellung in der Entität *WarenkorbArtikel* gespeichert. Das hier beschriebene konzeptuelle Datenmodell wird im Laufe dieser Arbeit immer wieder herangezogen, um die vielfältigen Abbildungsmöglichkeiten von konzeptuellen Komponenten auf die verschiedenen logischen Datenmodelle der zu betrachtenden Datenbanksysteme zu erläutern.

2.2 Konzeptuelle Ebene

Das relationale Datenmodell basiert im Wesentlichen auf einfachen Tabellen (Relationen), in denen Zeilen die einzelnen Datenobjekte (Tupel) und Spalten deren jeweiligen Eigenschaften (Attribute) repräsentieren. Die Datenverarbeitung erfolgt mithilfe entsprechender Operatoren ausschließlich mengenorientiert [55, 94].

2.2.1 Datenmodell

Gegeben seien n Wertebereiche (Domänen) $D_1, D_2, \ldots, D_n$, deren Werte ausschließlich atomar sein dürfen. Dann ist eine Relation R als Teilmenge des kartesischen Produkts (Kreuzprodukt) der n Domänen definiert:

$$R \subseteq D_1 \times D_2 \times \ldots \times D_n.$$

Jede Stelle i mit $i \in \{1, \ldots, n\}$ der Relation R bildet ein Attribut, das innerhalb einer Relation über einen eindeutigen Namen angesprochen wird. Ein Schema der Relation R besteht aus dem Namen der Relation sowie den Attribut-Domänen-Paaren, wobei auf die Domänenangabe der Einfachheit halber häufig verzichtet wird:

$$R\,(A_1{:}\,D_1, A_2{:}\,D_2, \ldots, A_n{:}\,D_n).$$

Es wird zwischen einem Schema einer Relation und dessen Elementen (Instanzen) unterschieden [94, S.71]. Ein Element der Menge R wird als Tupel bezeichnet, dessen i-te Komponente dem Attribut A_i entspricht. Die Anzahl aller n Attribute wird als Grad einer Relation bezeichnet, die Anzahl aller Tupel als Kardinalität. Innerhalb einer Relation wird ein Tupel durch den Wert eines Primärschlüssels eindeutig identifiziert. Dieser entspricht im Idealfall einem oder einer möglichst minimalen Menge an Attributen. Primärschlüssel werden in der Relationennotation unterstrichen:

$$R\,(\underline{A_1{:}\,D_1}, A_2{:}\,D_2, \ldots, A_n{:}\,D_n).$$

Üblicherweise werden Relationen als Tabellen dargestellt, in denen das Relationenschema der Spaltenbeschriftung entspricht und einzelne Tupel durch jeweils eine Zeile repräsentiert werden.

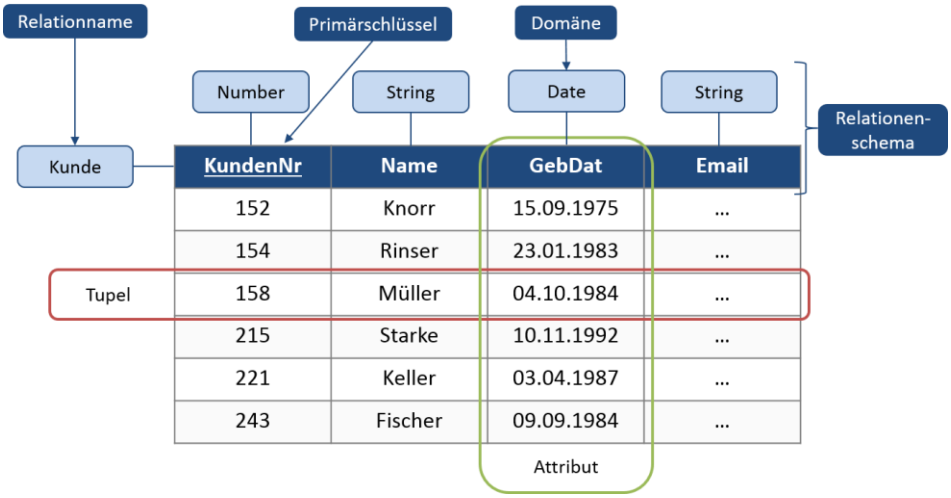


Abbildung 2.4: Logisches Datenmodell der Entität *Kunde*

Abbildung 2.4 beschreibt die Relation *Kunde*, welche der gleichnamigen Entität des in Abbildung 2.3 dargestellten konzeptuellen Datenmodells entspricht. Sie enthält die vier verschiedenen Attribute *KundenNr*, *Name*, *GebDat* und *Adresse*, weshalb sie als Relation vierten Grades bezeichnet wird. Das mehrwertige Attribut *Telefon* wird für gewöhnlich in einer zusätzlichen Tabelle abgebildet. Jedes dieser Attribute besitzt einen im Relationenschema definierten Wertebereich (`Number`, `String`, `Date` und `String`). Das Attribut *Adresse* wird in diesem Beispiel der Einfachheit halber als eine Zeichenkette verarbeitet. Das zugehörige Relationenschema lautet daher:

Kunde (KundenNr: Number, Name: String, GebDat: Date, Adresse: String).

Jedes Tupel in der Relation muss zwingend dieser definierten Struktur folgen, das heißt, jeder Datensatz besteht aus der gleichen Anzahl und Art von Attributen. Jeder Attributwert eines Datensatzes muss dem für das Attribut definierten Wertebereich entsprechen. Besitzt ein Tupel für ein bestimmtes Attribut keine Ausprägung, muss dieser Sachverhalt explizit mit dem Wert `null` gekennzeichnet werden.

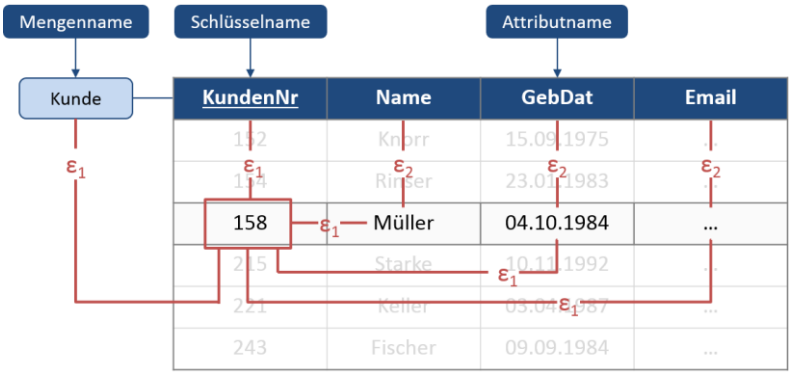


Abbildung 2.5: Prädikatorenschema der Relation *Kunde*

Die hohe Ausdrucksstärke des relationalen Datenmodells wird bei der Betrachtung des entsprechenden Prädikatorenschemas deutlich (Abbildung 2.5). Ein Tupel einer Relation entspricht einem Objekt der abgebildeten Miniwelt. Es wird durch einen Primärschlüssel identifiziert und durch Attributwerte beschrieben (Prädikation erster Ordnung). Attributnamen (Prädikationen zweiter Ordnung) sowie der Name des Primärschlüssels werden im Relationenschema verwaltet. Der Name der Relation entspricht dem Namen der Entitätsmenge. Das relationale Datenmodell ist somit in der Lage, sämtliche Elemente eines Prädikatorenschemas abzubilden.

Um Beziehungen zwischen Objekten darstellen zu können, kann ein Attribut einer Relation R_1 den Schlüsselkandidaten einer Relation R_2 referenzieren. Dieser wird in der Relation R_1 als Fremdschlüssel bezeichnet. „Ein Fremdschlüssel darf `null`-Werte besitzen“ [55, S.148].

Datenintegrität

„Ein relationales Datenbanksystem soll nicht nur die Daten verwalten, sondern auch für ihre Korrektheit sorgen“ [55, S.149]. Diese Korrektheit wird als Integrität bezeichnet und in drei verschiedenen Arten klassifiziert:

Referenzielle Integrität: „Eine Menge von Relationen $R_1, \ldots, R_k$ besitzt die referenzielle Integrität, wenn jeder Wert eines Fremdschlüssels einer Relation R_i Wert eines Primärschlüssels in einer anderen Relation R_j ist“ [55, S.149].

Entity-Integrität: „Eine Menge von Relationen $R_1, \ldots, R_k$ besitzt die Entity-Integrität, wenn jede Relation einen Primärschlüssel besitzt“ [55, S.149].

Semantische Integrität: „Eine Menge von Relationen $R_1, \ldots, R_k$ besitzt die semantische Integrität, wenn die Korrektheit der Eingaben der Benutzer gewährleistet ist“ [55, S.149].

Zur Einhaltung dieser Integritätsarten können verschiedene Bedingungen formuliert werden, die vom Datenbanksystem automatisch sichergestellt werden. Referenzielle

Integrität und Entity-Integrität werden durch statische Integritätsbedingungen formuliert, die bei jedem Zustand der Datenbank erfüllt sein müssen [94, S.159]. Semantische Integritätsbedingungen können mittels Constraints[11] deklarativ beschrieben werden. Häufig werden Constraints herangezogen, um den Wertebereich eines Attributs einzuschränken oder bestimmte Werte auszuschließen. Semantische Integrität kann jedoch auch mit dynamischen Bedingungen unter Zuhilfenahme von Triggern[12] überprüft werden. „Dynamische Integritätsbedingungen beschreiben, welche Bedingungen beim Übergang eines Zustands in einen anderen erfüllt sein müssen“ [55, S.151]. Hierzu zählt beispielsweise, dass sich das Gehalt eines Angestellten ausschließlich erhöhen oder „der Einkaufspreis innerhalb eines Jahres um nicht mehr als 10% steigen darf“ [55, S.151].

2.2.2 Anfrageverarbeitung

Relationale Algebra

Auf dem relationalen Datenmodell lassen sich verschiedene Arten von Operationen ausführen. Man unterscheidet zwischen Anfrageoperationen und Änderungsoperationen [76, S.328]. Änderungsoperationen haben die Modifikation des Datenbestandes zum Ziel. Zu ihnen zählen die Operationen `Insert`, `Update` und `Delete`, mit denen mehrere Tupel in eine Relation eingefügt, aktualisiert oder gelöscht werden können. Änderungsoperationen sind immer auf eine Relation beschränkt und werden tupelweise bearbeitet. Die für die Änderungsoperationen `Update` und `Delete` relevante Tupel-Menge wird durch Anfrageoperationen spezifiziert [52, S.239].

Anfrageoperationen basieren im relationalen Datenmodell auf der relationalen Algebra. Diese stellt eine fundamentale Grundlage relationaler Datenbanken dar. Mithilfe dieser formalen Sprache können Anfragen an ein relationales Datenbankschema formuliert und dadurch Informationen zusammengestellt werden. Die relationale Algebra wird als „abgeschlossen“ bezeichnet, da sie Operationen definiert, die auf Relationen angewendet werden und als Ergebnis erneut Relationen erzeugen. Es ist deswegen möglich, eine Sequenz relationaler Algebra-Operationen zu bilden, die als relationaler Ausdruck bezeichnet wird [52, S.242]. Die einzelnen Operationen eines relationalen Ausdrucks können beliebig ineinander geschachtelt werden [55, S.141].

Die relationale Algebra besteht in ihrem Kern aus sechs Basisoperationen, die im Folgenden kurz vorgestellt werden. Man unterscheidet dabei die Operationen, die speziell

[11] Constraints definieren Bedingungen wie `not null`, `unique` oder `check`.
[12] Trigger sind kleine Programme, die bei Änderungen ausgeführt werden.

für die relationale Algebra definiert wurden (Restriktion und Projektion) von den klassischen Operationen der Mengenlehre (Kartesisches Produkt, Vereinigung und Differenz) [55, S.125].

Restriktion (Selektion)

Bei der Restriktion werden die Tupel einer Relation ausgewählt, die eine gegebene Selektionsbedingung erfüllen. Die Restriktion wird mit σ gekennzeichnet und hat die Selektionsbedingung als Subskript [94, S.86]:

$$R_2 = \sigma_{<Selektionsbedingung>}(R_1).$$

Bei der Selektionsbedingung handelt es sich um einen booleschen Ausdruck, der sich aus Attributnamen aus R_1 oder konstanten Werten, den arithmetischen Vergleichsoperatoren ($=, \neq, <, \leq, >, \geq$) sowie den logischen Operatoren $\wedge$ (und), $\vee$ (oder) und $\neg$ (nicht) zusammensetzt.

Projektion

Im Gegensatz zur Restriktion, bei der einzelne Tupel (Zeilen) einer Relation ausgewählt werden, selektiert die Projektion einzelne Attribute (Spalten). Die Projektion wird durch das Operatorsymbol π dargestellt und hat einen sogenannten Projektor als Subskript [94, S.87]. Hierbei handelt es sich um eine Auflistung von Attributen der entsprechenden Relation R_1:

$$R_2 = \pi_{<Projektor>}(R_1).$$

Kartesisches Produkt (Kreuzprodukt)

Das kartesische Produkt zweier Relationen R und S enthält alle möglichen Paare aus Tupel t_r von R verknüpft mit den Tupel t_s von S:

$$R \times S = \{ t_r * t_s \mid t_r \in R \text{ und } t_s \in S\}.$$

Vereinigung (Union)

Zwei Relationen R und S mit gleichem Schema, das heißt gleicher Stelligkeit und identischen Domänen, können durch die Vereinigung zu einer Relation zusammengefasst werden [94, S.87]:

$$R \cup S = \{t \mid t \in R \text{ oder } t \in S\}.$$

Differenz

„Für zwei Relationen R und S mit gleichem Schema ist die Mengendifferenz […] definiert als Menge der Tupel, die in R aber nicht in S vorkommen“ [94, S.88]:

$$R - S = \{t \mid t \in R \text{ und nicht } t \in S\}.$$

Umbenennung

Eine Relation R als auch deren Attribute $a_1, \ldots, a_n$ können mit der Operation ρ umbenannt werden. Es entsteht eine neue, aber identische Relation S mit den Attributen $b_1, \ldots, b_n$:

$$S = \rho_{S(b1, \ldots, bn)}(R(a_1, \ldots, a_n)).$$

Diese sechs Operationen „sind ausreichend, um die relationale Algebra formal definieren zu können“ [94, S.90]. Die relationale Algebra umfasst noch weitere Operationen, die sich jedoch aus den bisher vorgestellten Basisoperationen ableiten lassen und deshalb nicht näher vorgestellt werden. Auch wenn diese Operationen die Ausdruckskraft nicht weiter erhöhen, werden sie häufig verwendet, da sie die Formulierung relationaler Ausdrücke wesentlich erleichtern. Zu diesen Operationen zählen Verbund (Join), Durchschnitt und Division. Eine ausführliche Erläuterung dieser Operationen befindet sich unter anderem in Kemper und Eickler [94, S.90] und in Elmasri und Navathe [52, S.249]. Abbildung 2.6 veranschaulicht die sechs Basisoperationen der relationalen Algebra.

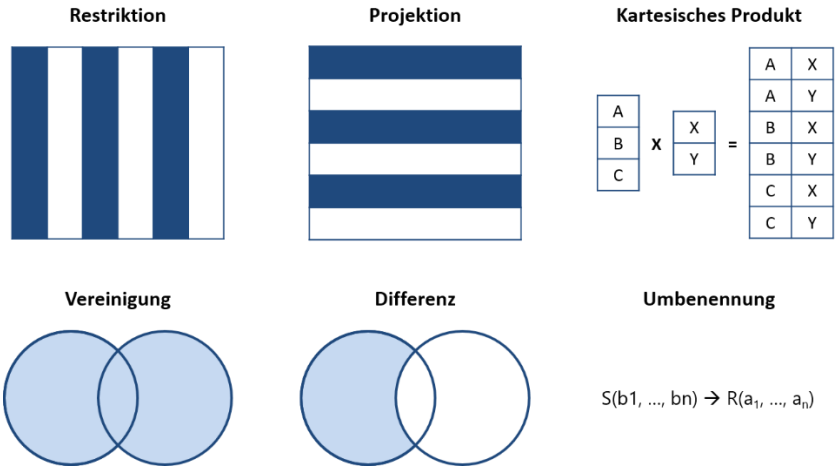


Abbildung 2.6: Operationen der relationalen Algebra

„Codd [34] hat die Ausdruckskraft von relationalen Anfragesprachen definiert. In seiner Terminologie heißt eine Anfragesprache **relational vollständig**, wenn sie mindestens so mächtig ist wie die relationale Algebra“ [94, S.105]. Wie bereits erwähnt, ist die Bereitstellung der sechs relationalen Basisoperationen $(\sigma, \pi, x, \cup, -, \rho)$ hierfür ausreichend.

SQL

Die Structured Query Language (SQL) „ist die Standardsprache[13] zur Kommunikation mit relationalen Datenbanksystemen“ [67, S.208] und wird in leicht abgeänderter Form (Dialekt) „von praktisch allen relationalen Datenbanksystemen zur Verfügung gestellt“ [94, S.111]. SQL „gilt als einer der wichtigsten Gründe für den Erfolg relationaler Datenbanken in der kommerziellen Welt“ [52, S.273].

„SQL beinhaltet einige Merkmale der relationalen Algebra, basiert aber größtenteils auf dem relationalen Tupelkalkül[14], also einer weiteren formalen Anfragesprache für relationale Datenbanken“ [52, S.273-274], die jedoch an dieser Stelle nicht vertieft vorgestellt wird. „Im Unterschied zum theoretischen Modell werden in der Praxis einige Vereinfachungen gemacht, die die Benutzer entlasten und eine effizientere Abarbeitung ermöglichen“ [94, S.111]. Trotz dieser Vereinfachung ist SQL streng relational vollständig, das heißt, jede Operation der relationalen Algebra wird durch genau einen Datenbankoperator der Anfragesprache umgesetzt (Tabelle 2.1).

Relationale Algebra	SQL
Projektion	✓
Restriktion	✓
Kartesisches Produkt	✓
Vereinigung	✓
Differenz	✓
Umbenennung	✓

Tabelle 2.1: SQL ist relational vollständig

Die bisher vorgestellten Befehle stellen dabei nur eine Teilmenge der sogenannten Data Manipulation Language (DML) dar. Neben vielen weiteren Operationen der DML bietet SQL mit der Data Definition Language (DDL) außerdem umfangreiche Möglichkeiten, das relationale Schema zu definieren und zu ändern. Benutzerrechte lassen sich mit der Data Control Language (DCL) verwalten. SQL stellt dementsprechend „alle Funktionen zur Verfügung, die man benötigt, um mit relationalen Datenbanksystemen zu arbeiten“ [67, S.212]. Eine vollständige Übersicht über den Umfang von SQL kann diese Arbeit nicht geben. Es wird daher auf die einschlägige Literatur verwiesen [52, 67, 94]. Tabelle 2.2 gibt eine Übersicht über alle von SQL zur Formulierung von Restriktionsbedingungen zur Verfügung gestellten Operationen.

[13] ISO/IEC 9075:2011

[14] Das relationale Tupelkalkül ist eine formale Anfragesprache, die auf der mathematischen Logik basiert. Sie ist deklarativ und unterscheidet sich somit von der relationalen Algebra, in der eine Suchanfrage durch eine Sequenz von Operationen prozedural formuliert wird [52, S.329].

Operator	SQL
Gleich	✓
Ungleich	✓
Größer, Kleiner	✓
Like	✓
In	✓
Exists	✓
And, Or	✓
Not	✓

Tabelle 2.2: Von SQL unterstützte Operatoren

Bei SQL handelt es sich um eine deskriptive Sprache, die im Gegensatz zu prozeduralen Sprachen nicht definiert, wie eine Aufgabe gelöst werden soll, sondern was als Ergebnis erwartet wird. „Die oft sehr komplexen, zur Festlegung der Auswertung nötigen Entscheidungen werden vom Anfrageoptimierer des Datenbanksystems übernommen“ [94, S.111]. SQL ist deswegen trotz ihrer Mächtigkeit sehr benutzerfreundlich, da Anwender mithilfe der sehr leicht erlernbaren Syntax nur formulieren müssen, „welche Daten sie interessieren, und nicht, wie die Auswertung der Daten vorgenommen wird“ [94, S.111].

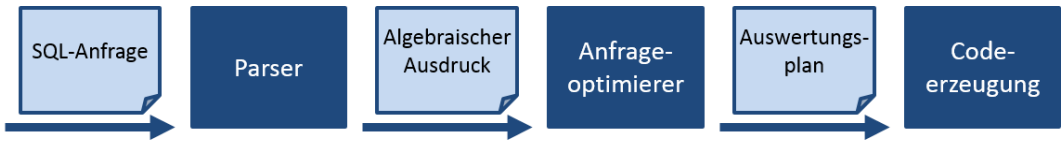


Abbildung 2.7: Verarbeitung einer SQL-Anfrage

SQL-Anfragen werden, wie in Abbildung 2.7 verdeutlicht, in drei Schritten in ausführbaren Code übersetzt. Ein Parser prüft die syntaktische Korrektheit des SQL-Statements und formuliert die Anfrage in einen Ausdruck der relationalen Algebra um. Dieser Ausdruck wird vom Anfrageoptimierer bearbeitet und in einen Auswertungsplan übersetzt. Der Auswertungsplan wird abschließend in ausführbaren Code umgewandelt [55, S.172].

2.2.3 Datenmodellierung

Die relationale Entwurfstheorie beschäftigt sich mit der Transformation des konzeptuellen in das logische Datenbankmodell. Da es viele Möglichkeiten gibt, einen Sachverhalt der Miniwelt in einem relationalen Schema darzustellen, ist es das Ziel, ein möglichst „gutes“ Schema zu entwerfen. Bei der Datenmodellierung wird in der Regel zweistufig vorgegangen. Zunächst wird das konzeptuelle Modell unter Zuhilfenahme

von Algorithmen in das relationale Datenmodell überführt. Dieser Prozess kann in fünf Schritte unterteilt werden:

1. Konvertierung der starken Entitätstypen,
2. Konvertierung der schwachen Entitätstypen,
3. Konvertierung der Beziehungen,
4. Auflösung mehrwertiger Attribute,
5. Konvertierung von is-a-Beziehungen (Spezialisierung, Generalisierung).

Eine detaillierte Beschreibung der einzelnen Abbildungsschritte findet sich unter anderem in Elmasri und Navathe [52, S.320]. Die verschiedenen Konstrukte des ER- und des relationalen Modells sind in Tabelle 2.3 gegenübergestellt. Eine Abbildung von is-a-Beziehungen ist im relationalen Datenmodell nicht vorgesehen. Eine Vorstellung von Modellierungsmustern zur Kompensation dieses Nachteils folgt in Kapitel 6.

ER-Modell	Relationales Modell
Entitätstyp	Relation
Entität	Tupel
Einfaches Attribut	Attribut
Zusammengesetztes Attribut	Menge einfacher Attribute
Mehrwertiges Attribut	Relation und Referenzierung
1:1-Beziehung	Referenzierung
1:N-Beziehung	Referenzierung
M:N-Beziehung	Relation und Referenzierung

Tabelle 2.3: Komponenten des ER-Modells und des relationalen Modells

Die systematische Transformation des konzeptuellen in das logische Datenmodell kann mithilfe von CASE[15]-Werkzeugen automatisch erfolgen. Hierbei wird das konzeptuelle Datenmodell zunächst analysiert. Anschließend werden SQL-DDL-Befehle generiert, mit denen das relationale Schema erzeugt werden kann. Das so generierte grobe Datenbankschema benötigt eine in der zweiten Stufe durchgeführte konzeptuelle Feinabstimmung [94, S.171]. „Die Basis für diesen Feinentwurf bilden funktionale Abhängigkeiten“, mit denen sogenannte Normalformen definiert werden. „Normalformen dienen dazu die „Güte“ eines Relationenschemas zu bewerten“ [94, S.171].

[15] CASE: Computer Aided Software Engineering. Eine Übersicht über verschiedene Werkzeuge bietet unter anderem [52, S.604].

Ein gutes Datenbankschema zeichnet sich neben einer hohen Übersichtlichkeit vor allem durch möglichst wenig redundant gespeicherte Informationen aus. So können einerseits der benötigte Speicherplatz reduziert und andererseits Inkonsistenzen vermieden werden, die durch verschiedene Anomalien verursacht werden [52, 55]. „Wenn für ein Relationenschema diese Normalformen nicht erfüllt sind, kann man es durch Anwendung entsprechender Normalisierungsalgorithmen in mehrere Schemata zerlegen, die dann die entsprechende Normalform erfüllen“ [94]. „Insgesamt existieren fünf Normalisierungsregeln, die aufeinander aufbauen“ [55].

In der Praxis kommen hiervon in der Regel maximal die ersten drei Regeln zum Einsatz, da mit zunehmendem Normalisierungsgrad der Programmieraufwand und die Komplexität des Datenmodells steigen und sich parallel Performanzverluste bei der Anfrage der Daten einstellen können. Dieser Effekt ist vor allem dann zu beobachten, wenn inhaltlich zusammengehörige Daten über mehrere Tabellen verteilt werden und sich dadurch die Anzahl der Operationen erhöht, die benötigt werden, um diese verteilten Informationen bei einer Anfrage wieder zusammenzuführen. Bei der Normalisierung muss darum zwischen Performanz und Konsistenz des Datenmodells abgewogen werden. Häufig wird das Datenmodell in einem ersten Schritt vollständig normalisiert und dann gezielt an den Stellen denormalisiert, an denen Performanzvorteile die Nachteile von Redundanzen überwiegen [55, 67]. Eine Möglichkeit, um die Konsistenz der Datenbank trotz eines geringen Normalisierungsgrads gewährleisten zu können, bietet der Einsatz von Datenbanktriggern.

Kunde

KundenNr	Name	GebDat	Email
152	Knorr	15.09.1975	22
...	...	...	...

Warenkorb

WarenkorbId	KundenNr	Preis
2	152	99,95
...	...	...

KundeAdresse

KundenNr	AdresseId
152	2
...	...

Telefon

KundenNr	Bezeichnung	Nummer
152	Mobil	0176554387
...	...	...

Artikel

ArtikelNr	Name	Preis
34	DBIS1	29.95
...	...	...

Adresse

AdresseId	Straße	Plz	Ort
2	Hauptstr. 3	13593	Berlin
...	...	...	...

WarenkorbArtikel

WarenkorbId	ArtikelNr	Menge	Preis
2	34	4	99.95
...	...	...	...

Abbildung 2.8: Logisches Datenmodell eines Online-Shops

Abbildung 2.8 beschreibt eine logische Abbildung des in Abbildung 2.3 dargestellten konzeptuellen Datenmodells eines Online-Shops. In einem ersten Schritt wird jede konzeptuelle Entität auf eine logische Relation abgebildet. Anschließend muss für jede M:N-Beziehung, wie beispielsweise die zwischen den Entitäten *Kunde* und *Adresse,*

eine sogenannte Join-Tabelle (*KundeAdresse*) angelegt werden. Auch das mehrwertige Attribut *Telefon* in der Entität *Kunde* wird in der Regel durch mindestens eine zusätzliche Tabelle abgebildet, weil relationale Datenbanken den Datentyp `Map` nicht unterstützen. Das am Ende erstellte logische Datenmodell umfasst aus diesen Gründen mehr Entitäten als das zugehörige konzeptuelle Datenmodell.

2.3 Interne Ebene

2.3.1 Speicherstrukturen

In Datenbanksystemen stellen üblicherweise Magnetplatten (Hard Disk Drive – HDD) das bevorzugte Speichermedium dar. Diese bestehen aus mehreren Oberflächen, die wiederum mehrere Spuren besitzen, welche schlussendlich in gleich große Blöcke unterteilt sind. Sowohl bei Magnetplatten als auch bei den immer mehr an Bedeutung gewinnenden Solid State Disks (SSD) bilden Blöcke mit durchschnittlich 4096 Bytes die kleinste Speichereinheit.

Es stehen mehrere verschiedene Speicherstrukturen zur Verfügung, die bestimmen, wie einzelne Datensätze physisch auf die einzelnen Blöcke einer Platte verteilt werden. Ziel ist es, diejenige Speicherstruktur zu wählen, die eine möglichst effiziente Datenverarbeitung für den jeweiligen Anwendungsfall unterstützt. Die wichtigsten und im Folgenden vorgestellten Speicherstrukturen sind Heap, ISAM, B*-Baum und Hash-Funktionen [52, 55].

Neben diesen sogenannten primären Zugriffspfaden können sekundäre Zugriffspfade (Indizes) definiert werden, die keinen Einfluss auf die physische Speicherung der Datensätze haben. Ähnlich einem Schlagwortverzeichnis am Ende eines Buches sind hier zu einem bestimmten Schlüsselwert eines Datensatzes eine oder mehrere zugehörige Speicheradressen hinterlegt, was die Suche nach einem Datensatz erheblich beschleunigen kann. Man unterscheidet dichtbesetzte Indizes, welche die Schlüssel aller Datensätze enthalten und dünnbesetzte Indizes, welche nur auf einige ausgewählte Datensätze verweisen.

Heap

„Die Heap-Struktur (Haufen, sequenzielle Datei) ist die sequenzielle Speicherung der Daten in der Reihenfolge ihrer Eingabe. Die Daten werden in Blöcken nacheinander auf der Platte abgelegt, die als lineare Liste miteinander verkettet sind“ [55, S.469]. Aus diesem Grund können vor allem neue Datensätze äußerst effizient eingefügt werden, da diese lediglich in den letzten Block geschrieben werden müssen.

Allerdings ist das anschließende Auffinden eines Datensatzes vergleichsweise teuer, da die gespeicherten Daten Block für Block nach Treffern abgesucht werden müssen (lineare Suche). Bei b Blöcken müssen somit im Durchschnitt $b/2$ Blöcke in den Hauptspeicher gelesen werden. Deshalb werden Heap-Dateien häufig mit sekundären Zugriffspfaden kombiniert [52, S.162].

Es wird logisches Löschen mithilfe von Löschmarkierungen oder physisches Löschen durch Überschreiben mit Blanks unterstützt. In beiden Fällen wird der alte Speicherplatz nicht mehr zur Verfügung gestellt, weshalb nach einiger Zeit eine Reorganisation notwendig wird. Heap ist daher für kleine Tabellen geeignet, die idealerweise in einen Block passen. Scans auf großen Tabellen sind dagegen äußerst ineffizient [55, S.470].

ISAM

Bei der ISAM-Speicherstruktur (Index Sequential Access Method) werden die Daten über eine Indexspalte sortiert in den Blöcken gespeichert. Darüber hinaus wird ein dünnbesetzter Index angelegt, der jeweils den größten Schlüsselwert eines Blocks als dessen Repräsentant sowie die zugehörige logische Adresse des Blocks speichert (Abbildung 2.9).

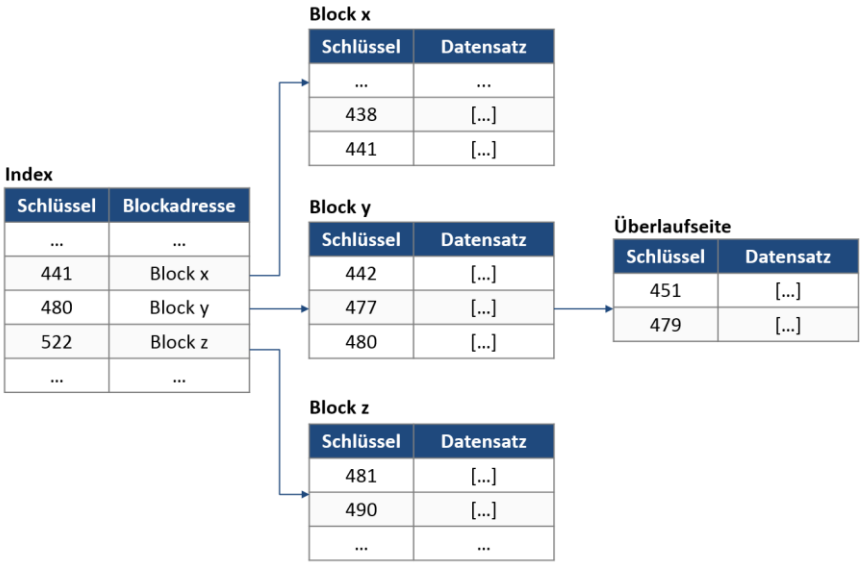


Abbildung 2.9: Aufbau einer ISAM-Speicherstruktur mit Überlaufseite

Beim Einfügen neuer Datensätze wird über den Index die entsprechende Blockadresse ermittelt und anschließend der Datensatz an der richtigen Stelle innerhalb des Blocks eingetragen. Wenn der Block bereits gefüllt ist, wird der Datensatz in einer Überlaufseite (Überlaufblock) gespeichert. Leseoperationen gestalten sich bei wenigen Überlaufseiten sehr effizient und benötigen im Idealfall genau zwei Operationen: Eine Leseoperation auf dem Index und eine innerhalb eines Blocks. „Bei Überlaufseiten sind

es entsprechend mehr Zugriffe, die die Antwortzeit schnell spürbar verlangsamen können“ [55, S.472]. Bei ISAM-Speicherstrukturen werden Datensätze physisch gelöscht und der Speicherplatz wieder für neue Datensätze zur Verfügung gestellt.

Da bei häufigen Datenänderungen schnell Überlaufseiten entstehen, die „eine regelmäßige Dateireorganisation“ nach sich ziehen, ist ISAM eher für „große, statische Datenbestände gut geeignet“. „Gegenüber Heap-Speicherung ist bei Anfragen ein dramatischer Zeitgewinn zu verzeichnen, […] beim Schreiben fällt der Mehraufwand meist nicht ins Gewicht“ [55, S.473]. ISAM-Indizes können bei großen Datenmengen auch mehrstufig gestaltet werden und erhalten dann eine Struktur, die den im Folgenden vorgestellten Bäumen ähnelt.

B*-Bäume

B*-Bäume (auch B⁺-Bäume genannt) stellen eine Erweiterung von B-Bäumen und ISAM dar, bei denen die eigentlichen Daten ausschließlich in linear geordneten Blattknoten des Baumes gespeichert werden. Die Knoten entsprechen einzelnen Blöcken und enthalten ausschließlich Schlüsselwerte. Innere Knoten verweisen mit einem linken Zeiger auf nachfolgende Knoten, die kleinere Schlüsselwerte verwalten und mit einem rechten Zeiger auf Knoten mit gleichen oder größeren Schlüsselwerten (Abbildung 2.10).

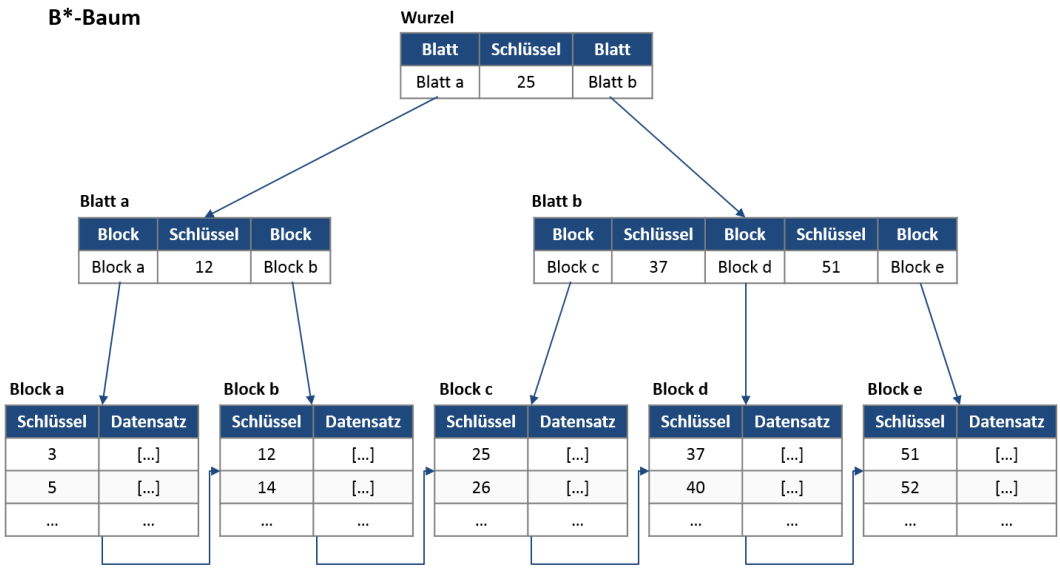


Abbildung 2.10: Aufbau eines B*-Baums mit der Höhe 3

Bäume sind stets balanciert, weshalb sich alle Blätter auf einer gleichen Ebene befinden und sich so für alle Datensätze ein gleicher Suchaufwand ergibt. Die Wurzel eines B*-Baums enthält immer mindestens einen Wert, innere Knoten sind mindestens immer bis zur Hälfte gefüllt.

B*-Bäume bieten eine sehr effiziente Suche in großen Datenbeständen. Ausgehend von der Wurzel entspricht die Anzahl der benötigten Operationen immer der Höhe h des Baumes. Beim Schreiben wird zunächst der Zielblock gesucht und anschließend der Wert in die sortierte Liste eingetragen. Im Fall eines Überlaufs wird der Block gesplittet und der darüber liegende Indexknoten um einen Verweis ergänzt. Es ergeben sich infolgedessen minimal eine und maximal $2h+1$ Operationen. Beim Löschen kann es zum sogenannten Unterlauf kommen, der eine Zusammenlegung der Knoten zur Folge hat. Hier liegen die Kosten bei minimal 1 und maximal $h+1$ [55, S.479].

Durch Splitten, Ausgleichen und Zusammenführen von Knoten bei Einfüge- und Löschoperationen ergibt sich ein „relativ hoher Verwaltungsaufwand“ [55, S.479]. Dieser ist jedoch akzeptabel, solange es sich dabei nicht um Massendatenverarbeitung handelt. Da sich ihre Struktur dynamisch anpassen lässt und infolgedessen kostspielige Reorganisationen vermieden werden können, sind B*-Bäume für veränderliche Datenbestände geeignet.

Hashing

„Die Idee des Hash-Verfahrens besteht darin, die Schlüsselwerte eines Datensatzes mithilfe eines Algorithmus direkt in die physische Adresse umzurechnen und so eine sehr gute Performanz beim Lesen zu erreichen“ [55, S.480]. Die am häufigsten verwendete Hash-Funktion ist die mathematische Funktion Modulo n. Hierfür wird einem Schlüssel zunächst ein ganzzahliger Wert zugeordnet, dieser wird anschließend durch eine große Zahl n geteilt und anhand des entstehenden Rests kann schließlich ein zugehöriger Block ermittelt werden [55, S.480]:

$$h(a) := a \text{ MOD } n.$$

Beim Schreiben weist die Hash-Funktion den errechneten Block zu. Falls dieser bereits voll ist, wird eine Überlaufseite angelegt, welche dann bei einer Suche linear durchlaufen werden muss. Ebenso effizient gestalten sich Löschoperationen, bei denen der Wert physisch entfernt wird. Der Nachteil von Hashing liegt in der nicht vorhandenen Unterstützung von Bereichsanfragen.

Zugriffskosten

Tabelle 2.4 zeigt eine grobe Kostenabschätzung der wichtigsten Zugriffsoperationen für die verschiedenen Speicherstrukturen. Es wird hierbei ein Datenbestand N von 10^8 Datensätzen und eine Clusterbildung von 100 Datensätzen pro Seite angenommen. Während sich die Kosten für Einfüge- und Löschoperationen bei B*-Bäumen angesichts möglicher Ausbalancierungen minimal erhöhen können, verursachen häufige Änderungen bei ISAM teure Reorganisationen des Index. Die Werte der Hash-Funktion berücksichtigen keine Überlaufseiten.

Operation	Direkter Zugriff	Sequenzielle Verarbeitung	Einfügen / Löschen
HEAP	$O(N/2) \approx 5*10^7$	$O(N) \approx 10^8$	$O(1) \approx 2$
ISAM / B*-Baum	$O(\log_k N) \approx 4$	$O(N/100) \approx 10^6$	$O(1) \geq 5$
HASH	$O(1) \approx 1$	$O(N) \approx 10^8$	$O(1) \approx 2$

Tabelle 2.4: Zugriffskosten der wichtigsten Operationen [76, S.225]

Hashverfahren bieten demnach den schnellsten Direktzugriff, unterstützen jedoch ansonst keine Suchzugriffe. Aus diesem Grund haben sich B*-Bäume „wegen ihren ausgewogenen Kosten für alle Operationen als Standardstruktur [für relationale Datenbanken] durchgesetzt“ [76, S.226]. Sekundäre Zugriffspfade können den Zugriff über Nicht-Schlüsselattribute deutlich beschleunigen. Von einem übermäßigen Gebrauch von Indizes sollte jedoch Abstand genommen werden, da die Einführung und Wartung eines Index automatisch mit einem Mehraufwand verbunden ist, der bei Einfüge- und Löschoperationen sowie bei Änderungen auf indizierten Attributen zu spürbaren Performanzverlusten führen kann [52, S.596].

2.3.2 Transaktionen und konkurrierende Zugriffe

„Eine Transaktion ist eine logische Verarbeitungseinheit auf der Datenbank, die eine oder mehrere Datenbankoperationen – Einfügen, Löschen, Modifizieren oder Suchen – beinhaltet“ [52, S.677]. Dieses Prinzip kann anhand des bereits aus Abbildung 2.3 bekannten Datenmodells eines Onlineshops veranschaulicht werden.

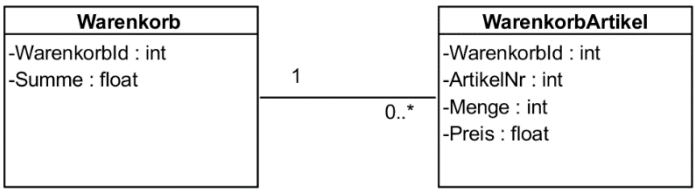


Abbildung 2.11: Konzeptuelles Modell der Entitäten *Warenkorb* und *WarenkorbArtikel*

In Abbildung 2.11 sind die Entitäten *Warenkorb* und *WarenkorbArtikel* abgebildet. Ein Warenkorb wird über einen Primärschlüssel identifiziert und besitzt eine Summe. Ein Warenkorb kann eine beliebige Anzahl an Artikeln enthalten. Diese besitzen neben der Artikelnummer eine Menge und einen Preis. Wird ein Artikel zu einem Warenkorb hinzugefügt, muss die Relation *WarenkorbArtikel* um einen Datensatz erweitert und das Attribut Summe in der Relation *Warenkorb* aktualisiert werden. Um diese einzel-

nen Operationen fehlerfrei und ohne unerwünschte Einflüsse anderer zeitgleich stattfindender Operationen ausführen zu können, werden diese zu einer Transaktion zusammengefasst.

```
Begin of Transaction
   //Schreibe Artikel in WarenkorbArtikel
   //Lies Summe aus Warenkorb
   //Schreibe aktualisierte Summe in Warenkorb
End of Transaction
```

Abbildung 2.12: Aktualisierung des Warenkorbs im Rahmen einer Transaktion

Eine mögliche Transaktion zur Aktualisierung des Warenkorbs ist in Abbildung 2.12 dargestellt. Eine Transaktion startet mit dem Befehl `Begin of Transaction` (BOT). Daraufhin werden die einzelnen Operationen definiert, die innerhalb einer Transaktion ausgeführt werden sollen. In diesem Fall wird zunächst der Artikel in die Relation *WarenkorbArtikel* eingefügt. Anschließend wird die Summe aus der Relation *Warenkorb* ausgelesen, um den entsprechenden Wert erhöht und wieder in die Relation *Warenkorb* zurückgeschrieben. Eine Transaktion endet mit dem Befehl `End of Transaction` (EOT).

Transaktionen weisen gewisse Eigenschaften auf, die unter dem Akronym ACID zusammengefasst werden:

Atomarität (Atomicity): Eine Transaktion wird vollständig oder überhaupt nicht ausgeführt („Alles-oder-Nichts-Prinzip“). Tritt während der Ausführung einer Transaktion ein Fehler auf, werden alle bisher durchgeführten Änderungen dieser Transaktion zurückgesetzt [76, S.393].

Konsistenz (Consistency): Eine Transaktion überführt eine Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand, das heißt, vor und nach der Ausführung einer Transaktion müssen die im Schema definierten Integritätsbedingungen erfüllt sein. „Änderungen, welche zu einer Verletzung der Integritätsbedingungen führen, werden abgewiesen, das heißt, sie führen zum Zurücksetzen der Transaktion“ [76, S.393].

Isolation (Isolation): Parallel ausgeführte Transaktionen laufen isoliert voneinander ab und beeinflussen sich nicht gegenseitig, das heißt, „alle anderen parallel ausgeführten Transaktionen beziehungsweise deren Effekte dürfen [für eine Transaktion] nicht sichtbar sein“ [94, S.289].

Dauerhaftigkeit (Durability): Die von einer Transaktion durchgeführten Änderungen werden dauerhaft (persistent) gespeichert und sind gegenüber zukünftigen Soft- und Hardwarefehlern gesichert.

Aus diesen vier Eigenschaften lassen sich zwei grundlegende Anforderungen an die Transaktionsverwaltung eines Datenbanksystems ableiten. Einerseits müssen konsistente Datenbankzustände nach Fehlern wiederhergestellt werden können (Fehlerbehandlung) und andererseits müssen Transaktionen im Mehrbenutzerbetrieb isoliert voneinander ablaufen können (Nebenläufigkeitskontrolle) [94, S.289]. Entsprechend dieser Anforderungen besteht eine Transaktionsverwaltung aus den Komponenten Recovery (Wiederherstellung) und Synchronisation.

Recovery

In einem Datenbanksystem können verschiedene Fehler zu einem Datenverlust führen. Man unterscheidet lokale Fehler einer Transaktion, Fehler mit Hauptspeicherverlust und Fehler mit Hintergrundspeicherverlust. Je nach aufgetretenem Fehler müssen von der Recovery-Komponente der Transaktionsverwaltung unterschiedliche Maßnahmen eingeleitet werden, um den jüngsten konsistenten Datenbankzustand wiederherstellen zu können [94, S.293]:

Lokale Fehler einer Transaktion: Die am häufigsten auftretenden Fehler sind sogenannte lokale Fehler, bei denen Transaktionen vom Datenbanksystem oder vom Benutzer zurückgesetzt werden. In einem solchen Fehlerfall werden im Rahmen eines „lokalen Undo“ sämtliche Operationen widerrufen, die innerhalb der annullierten Transaktion bereits ausgeführt wurden.

Fehler mit Hauptspeicherverlust: Die zweite Fehlerklasse bilden Fehler mit Hauptspeicherverlust, hervorgerufen beispielsweise durch Stromausfälle, bei denen alle Daten verloren gehen, die noch nicht im Hintergrundspeicher gesichert wurden. Im Rahmen des Recovery wird in diesem Fall zunächst ein „globales Undo“ durchgeführt, um alle nicht erfolgreich abgeschlossenen Transaktionen zurückzusetzen und anschließend ein „Redo“ veranlasst, um die erfolgreichen, aber noch nicht materialisierten Transaktionen erneut durchzuführen. Die hierzu notwendigen Informationen befinden sich in einer im Hintergrundspeicher befindlichen Logdatei (Write Ahead Log – WAL), die vor und nach dem Durchführen einer Transaktion mit Wiederherstellungsinformationen beschrieben wird (Abbildung 2.13).

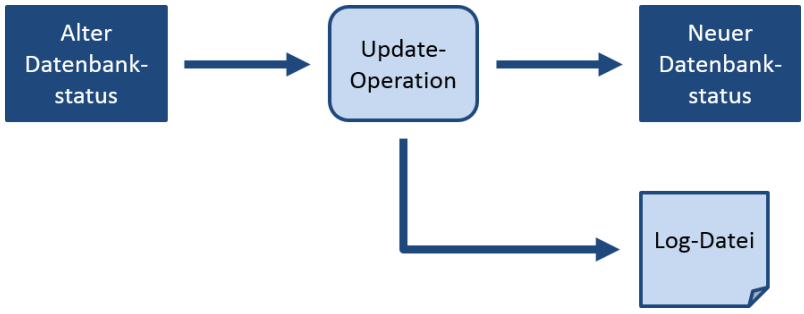


Abbildung 2.13: Prinzip des Write Ahead Loggings

Das Intervall, in der diese Log-Datei persistiert wird, hängt vom eingesetzten Datenbanksystem ab. Das von MySQL verwendete Speichersubsystem InnoDB sichert diese Datei einmal pro Sekunde [128, S.1210].

Fehler mit Hintergrundspeicherverlust: Fehler der dritten und am seltensten auftretenden Fehlerklasse führen zu einem Verlust des Hintergrundspeichers durch Hardwarefehler oder Zerstörungen. Für eine Behebung solcher Fehler ist „eine Archivkopie der materialisierten Datenbasis und ein Log-Archiv mit allen seit Anlegen dieser Datenbasis-Archivkopie vollzogenen Änderungen notwendig“ [94, S.295]. Hierfür bietet sich die später behandelte Replikation an.

Synchronisation

„Eine zentrale Problemstellung beim Mehrbenutzerbetrieb ist der Umgang mit konkurrierenden Zugriffen verschiedener Benutzer auf das gleiche Objekt, zum Beispiel auf eine Tabelle oder einen Datensatz“ [55, S.443]. Infolge eines unkontrollierten Zugriffs können verschiedene Probleme (Anomalien) auftreten:

Lost Update: „Dieses Problem entsteht, wenn sich die Operationen zweier Transaktionen, die auf die gleichen Datenobjekte zugreifen, auf eine Weise zeitlich überschneiden, sodass sich ein falscher Wert eines Datenbankobjekts ergibt“ [52, S.679].

Dirty Read: „Dieses Problem entsteht, wenn eine Transaktion ein Datenbankobjekt aktualisiert und dann aus irgendeinem Grund fehlschlägt. Das aktualisierte Objekt wird von einer anderen Transaktion gelesen, bevor es auf seinen Originalwert zurückgesetzt wird“ [52, S.681].

Unrepeatable Read: Das Problem „entsteht, wenn eine Transaktion […] ein Objekt zweimal liest, während der Wert des Objekts inzwischen von einer anderen Transaktion […] geändert wird“ [52, S.682].

Phantome: „Phantome und nicht wiederholbares Lesen beschreiben im Wesentlichen das gleiche Problem mit dem Unterschied, dass bei den Phantomen [eine andere Transaktion] neue Datensätze eingefügt [hat], während beim nichtwiederholbaren Lesen […] zwischenzeitlich bestehende Daten geändert oder gelöscht“ wurden [55, S.445].

Um Anomalien im Mehrbenutzerbetrieb zu vermeiden und die Konsistenz im Datenbanksystem zu schützen, müssen Datenbankobjekte vor konkurrierenden Zugriffen geschützt werden. Da die sichere Gewährleistung der Konsistenz durch eine strikte serielle Ausführung der Transaktionen „sich jedoch v.a. aus Leistungsgründen“ [147, S.133] verbietet, werden im Folgenden einige Synchronisationsverfahren vorgestellt,

die das Korrektheitskriterium der Serialisierbarkeit[16] erfüllen. Über dieses sehr umfangreiche Thema der Nebenläufigkeitskontrolle (concurrency control) kann an dieser Stelle jedoch nur ein kurzer Überblick gegeben werden[17]. Man unterscheidet pessimistische, versionierte und optimistische Synchronisationsverfahren.

Pessimistische Synchronisationsverfahren: Operationen bei pessimistischen Synchronisationsverfahren erhalten einen exklusiven Zugriff auf Ressourcen mithilfe von Sperren. Das am häufigsten verwendete Verfahren ist das Zwei-Phasen-Sperrprotokoll (Two Phase Locking – 2PL), bei dem zunächst während der sogenannten Wachstumsphase für alle benötigten Objekte Sperren angefordert und diese anschließend während der sogenannten Schrumpfungsphase wieder freigegeben werden. „Die Verwendung von Sperren in Kombination mit dem 2PL-Protokoll garantiert Serialisierbarkeit von Ausführungsplänen. Wenn eine Transaktion ein bereits gesperrtes Objekt benötigt, muss sie unter Umständen warten, bis das Objekt freigegeben wird“ [52, S.720–721]. Abgesehen von der gestiegenen Wahrscheinlichkeit für Deadlocks und verhungernde Transaktionen können sich durch Sperren Leistungseinbußen ergeben, weshalb es wichtig ist, „eine Synchronisation mit möglichst wenig Sperrkonflikten und Rücksetzung zu erreichen“ [147, S.19]. Seit SQL92 kann deswegen der Grad der Isolation eingestellt und demzufolge ein Kompromiss zwischen tolerierbaren Anomalien und benötigter Geschwindigkeit gewählt werden.

Versionierte Synchronisationsverfahren: Mehrversionen-Synchronisationsverfahren (Multiversion Concurrency Control – MVCC) streben eine Reduzierung an Synchronisationskonflikten an, indem für geänderte Objekte zeitweilig mehrere Versionen geführt werden. Dadurch können Lesetransaktionen eine Sicht auf die Datenbank erhalten, die zum Start ihrer Ausführung gültig war. Nachfolgende Änderungen bleiben für die Transaktion unsichtbar. Dieses Konzept beschränkt Konflikte folglich auf Schreiboperationen, welche sowohl unter Zuhilfenahme pessimistischer als auch optimistischer Verfahren gelöst werden können. Gerade in konfliktreichen Anwendungen kann die Leistungsfähigkeit dank MVCC deutlich gesteigert werden [27].

Optimistische Synchronisationsverfahren: Optimistische Synchronisationsverfahren verfolgen die Annahme, dass Konflikte seltene Ereignisse sind und somit präventive Sperren der Datenbankobjekte einen unnötigen Aufwand darstellen. Darum erfolgt „bei optimistischen Nebenläufigkeitskontrolltechniken […] während der Ausführung einer Transaktion keine Überprüfung“ [52, S.726]. Erst nach Abschluss der auf einer lokalen Kopie durchgeführten Operationen werden mögliche Synchronisati-

[16] „Beim Konzept der Serialisierbarkeit werden die Vorzüge der seriellen Ausführung – nämlich Isolation – mit den Vorteilen des Mehrbenutzerbetriebs – nämlich erhöhter Durchsatz – kombiniert“ [94, S.318].

[17] Umfangreiche Literatur zu diesem Thema findet sich unter anderem in Elmasri und Navathe [52] und in Härder und Rahm [76].

onskonflikte mit der Datenbank ermittelt. Im Fall eines Konflikts werden die Änderungen abgewiesen und die Transaktion erneut durchgeführt. Aus diesem Grund eignen sich diese Verfahren nur bei kurzen Transaktionen sowie Operationen mit geringem Konfliktpotenzial.

Transaktionen bilden die „Voraussetzung für die sichere und konsistente Ausführung von DB-Operationen, trotz gleichzeitiger DB-Zugriffe durch zahlreiche Benutzer und möglicher Fehlersituationen wie Rechner- oder Plattenausfälle. Für wesentliche Aufgaben der Transaktionsverwaltung, insbesondere Synchronisation, Logging und Recovery, steht ein Fundus an leistungsfähigen und in der Praxis erprobten Verfahren zur Verfügung“ [76, S.391]. Diese Techniken unterscheiden sich jedoch teilweise erheblich in ihrer Umsetzung der ACID-Eigenschaften. Da eine strikte Umsetzung der ACID-Eigenschaften in Konflikt mit einem möglichst hohen Durchsatz an Transaktionen steht, muss für jeden Anwendungsfall ein geeigneter Kompromiss zwischen diesen beiden Parametern gefunden werden.

Transaktionen sind bei relationalen Datenbanksystemen standardmäßig atomar und werden dank des WAL-Prinzips dauerhaft gespeichert. Zur Synchronisation von nebenläufigen Transaktionen verwenden die meisten relationalen Datenbanken MVCC in Verbindung mit Sperren [128, 136]. Hierdurch werden Leseoperationen nicht blockiert und eine zu einem definierbaren Zeitpunkt konsistente Sicht auf die Daten ermöglicht. Zusätzlich werden häufig die vier im SQL92-Standard definierten Isolationslevel unterstützt. Diese Level bestimmen, wie lange Lese- und Schreibsperren auf einem Objekt gehalten werden und demnach auch, welche Mehrbenutzeranomalien für eine bessere Performanz bewusst in Kauf genommen werden. Tabelle 2.5 gibt einen Überblick über die vier Isolationslevel und die dabei jeweils möglichen Mehrbenutzeranomalien.

Isolationslevel	Lost Update	Dirty Read	Unrepeatable Read	Phantome
Read Uncommitted	✓	✗	✗	✗
Read Committed	✓	✓	✗	✗
Repeatable Read	✓	✓	✓	✗
Serializable	✓	✓	✓	✗

Tabelle 2.5: Isolationslevel des SQL92 Standards und die jeweils vermiedenen Anomalien

2.4 Verteilte Datenhaltung

Steigende Anforderungen können bei einem Datenbanksystem schnell zu Leistungs- und Verfügbarkeitsproblemen führen. Um wachsenden Durchsatz- und Antwortzeitanforderungen gerecht werden zu können, ist es erforderlich, dass sich ein bestehendes Datenbanksystem skalieren lässt. Ein System wird als skalierbar bezeichnet, wenn es in der Lage ist, eine erhöhte Last zu bearbeiten ohne gleichzeitig Leistungseinbußen hinnehmen zu müssen [132]. Last bezieht sich dabei in der Regel auf die zu behandelnde Datenmenge sowie auf die Lese- und Schreibanfragen von Nutzern. Man unterscheidet zwischen vertikaler und horizontaler Skalierung.

Vertikale Skalierung: Die einfachste und nächstliegende Skalierungsmöglichkeit stellt die sogenannte vertikale Skalierung dar, bei der ein vorhandenes Datenbanksystem mit leistungsfähigeren Hardwarekomponenten ausgerüstet wird. Da bei diesem Ansatz weder die Komplexität des Datenbanksystems steigt noch Anpassungen auf Applikationsebene notwendig sind, hat sich dieser Ansatz beim Großteil heutiger Informationssystemen bewährt [155, S.529]. Sobald die Anforderungen jedoch schneller wachsen, als sie mithilfe von Moores Gesetz[18] gelöst werden können, ist diese Lösung zunehmend ungeeigneter. Da die Kosten für solche besseren [Hardwarekomponenten] nicht linear verlaufen und im Hochpreissegment sehr wohl exponentiell steigen können“ [43], ist dieser Ansatz ab einem gewissen Punkt nicht nur unwirtschaftlich, sondern stößt an seine Grenzen, sobald die maximale Aufrüstungskapazität erreicht ist.

Horizontale Skalierung: Spätestens ab dem Moment, an dem die vertikale Skalierung an ihre Grenzen stößt, ist es notwendig das Datenbanksystem auf mehrere Verarbeitungsrechner zu verteilen und es folglich nicht mehr vertikal, sondern horizontal skalieren zu lassen. Bei der horizontalen Skalierung wird die Leistungsfähigkeit des verteilten Datenbanksystems durch die Hinzunahme weiterer Verarbeitungsrechner inkrementell erweitert und steigt dabei idealerweise linear mit der Rechneranzahl [147, S.8]. Neben einer erhöhten Leistungsfähigkeit können verteilte Datenbanksysteme durch redundante Speicherung die Verfügbarkeit des Gesamtsystems erhöhen. Ein weiterer Vorteil ergibt sich angesichts des Einsatzes von Standardhardware, welche wesentlich kosteneffizienter im Vergleich zu Großrechnern ist [147, S.10].

Man unterscheidet viele verschiedene Arten von verteilten Datenbanksystemen [147, S.5], die im Rahmen dieser Arbeit allerdings nicht alle behandelt werden können. Die im Folgenden vorgestellten verteilten Datenbanksysteme besitzen eine sogenannte Shared-Nothing-Architektur, bei der die einzelnen Verarbeitungsrechner (Knoten)

[18] Moores Gesetz: Die heutige Auslegung des Mooreschen Gesetzes besagt, dass sich die Anzahl der Komponenten auf einem Computerchip alle 18 Monate verdoppelt und sich somit die Rechenleistung in diesem Zeitraum entsprechend stark erhöht.

voneinander unabhängig und eigenständig in der Lage sind, ihre Aufgaben zu erledigen. Die verschiedenen Knoten verwenden identische Software, teilen sich aber untereinander keine Hardwareressourcen.

Die einzelnen Knoten eines verteilten Datenbanksystems können sowohl lokal als auch global verteilt werden. Die Vorteile der lokalen Verteilung, beispielsweise innerhalb eines Rechenzentrums, liegen in einer zentralen Administration und den sehr guten Leistungs- und Verfügbarkeitseigenschaften, die sich aus der Kommunikation über lokale Netzwerke (Local Area Network – LAN) ergeben. Knoten von global verteilten Systemen kommunizieren dagegen über wesentlich unzuverlässigere und langsamere Netzwerke (Wide Area Network – WAN), können dafür jedoch Kommunikationskosten einsparen, indem Daten gezielt auf den Rechnern gehalten und verarbeitet werden, in deren Regionen diese Daten am häufigsten verwendet werden (Ausnutzung von Lokalität).

2.4.1 Replikation

Die Replikation bietet die einfachste Möglichkeit, Daten auf mehrere Verarbeitungsrechner zu verteilen. Um die in den meisten Informationssystemen dominierenden Leseoperationen horizontal skalieren zu können, hat sich die redundante Speicherung der für diese Anfragen relevanten Daten auf unterschiedlichen Servern bewährt. Dabei können sowohl Kommunikationskosten durch Weiterleitung der Anfrage an geografisch günstigere Replikate eingespart werden (Ausnutzung von Lokalität), als auch Geschwindigkeitsvorteile erzielt werden, indem die Last auf leistungsstärkere Systeme verteilt wird (Caching). Daneben lassen sich sowohl die Datensicherheit als auch die Verfügbarkeit des Systems durch Replikation auf mehrere Server erhöhen. Da sich der administrative Aufwand vergleichsweise gering gestaltet [155, S.448], ergibt sich in Kombination mit vertikaler Skalierung eine attraktive Lösung, mit der sich die Anforderungen der meisten Informations- und Kommunikationssysteme der heutigen Zeit effizient bewältigen lassen [60, 155].

In Abbildung 2.14 ist die Verteilung von Anfragen auf eine dreifach replizierte Master-Slave-Architektur dargestellt. Da sowohl der Master-Knoten als auch die beiden Slave-Knoten die gleichen Datensätze enthalten, kann die Last von Leseanfragen gleichmäßig auf alle drei Knoten verteilt werden. Im Gegensatz zu Leseanfragen lassen sich Schreibanfragen nicht so einfach auf unterschiedliche Knoten verteilen. Um mögliche Konflikte bei zeitgleichen Änderungen auf unterschiedlichen Knoten zu vermeiden, werden Änderungsoperationen in der Regel nur auf dem Master-Knoten zugelassen. Der Zeitpunkt der anschließenden Synchronisation der Slave-Knoten ist abhängig von den Anforderungen des Anwendungsfalls. Während die synchrone Aktualisierung die sofortige Konsistenz aller Knoten garantiert, ermöglicht die asynchrone Aktualisierung vor allem in geografisch verteilten Systemen eine deutlich schnellere Verarbeitung von Schreibanfragen. Je nach Anwendungsfall muss deshalb ein Kompromiss zwischen Konsistenz und Leistungsfähigkeit gefunden werden. Mittels flexibler Protokolle lassen sich diese Parameter in der Regel sehr feingranular bestimmen. Neben

den erhöhten Änderungskosten sind der erhöhte Speicherplatzbedarf und die gestiegene Systemkomplexität als Nachteile der Replikation zu nennen.

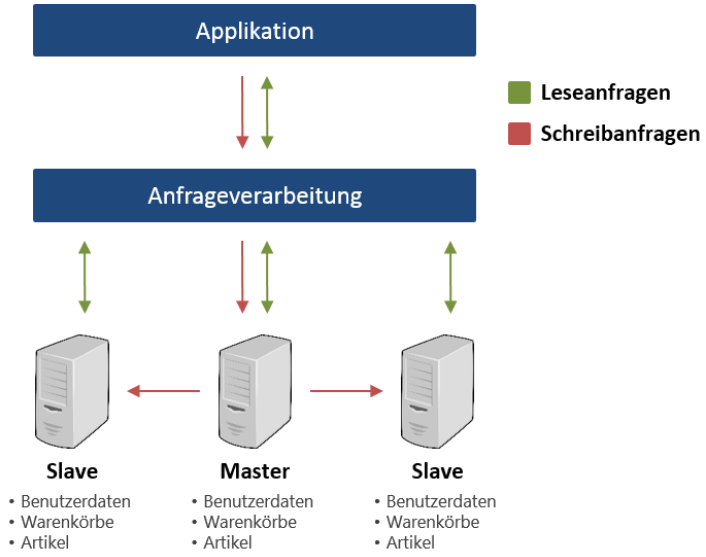


Abbildung 2.14: Anfrageverarbeitung einer replizierten Master-Slave-Architektur

2.4.2 Fragmentierung und Allokation

Mithilfe der Replikation lassen sich Leseanfragen sehr einfach auf mehrere Knoten verteilen. Sobald jedoch die Datenmenge oder Schreiboperationen skaliert werden müssen, ist eine physische Aufteilung des Datenbestands auf mehrere Verarbeitungsrechner unvermeidlich. Bei dem als Top-Down-Ansatz bezeichneten Verfahren [138, 147] wird ein vorhandenes logisches (globales) Schema in mehrere Fragmente unterteilt (Fragmentierungsschema) und diese Fragmente anschließend den einzelnen Verarbeitungsrechnern des verteilten Datenbanksystems zugeordnet (Allokationsschema) [94, S.473].

Fragmentierung

Die physische Verteilung des Datenbestandes auf mehrere Verarbeitungsrechner erfolgt auf Grundlage des Zugriffsverhaltens der zu unterstützenden Anwendungen. Da die Bestimmung der Datenverteilung „von fundamentaler Bedeutung“ für die Leistungsfähigkeit des Gesamtsystems ist und gleichzeitig sehr viele verschiedene, teilweise konträre Anforderungen [147, S.71] berücksichtigt werden müssen, erweist sich die Erstellung von Fragmentierungs- und Allokationsschemata als ein durchaus komplexes Optimierungsproblem, für dessen Lösung umfangreiches Fachwissen sowie umfassende Kenntnisse über die zu unterstützenden Anwendungen notwendig sind.

Eine beliebte Verteilungsmethode stellt die funktionale Partitionierung dar. Bei diesem Verfahren werden die Entitäten eines Datenmodells entsprechend ihrer funktionalen Aufgabe auf verschiedene Knoten verteilt. So lässt sich das in Abbildung 2.3 beschriebene Datenmodell eines Online-Shops beispielsweise in drei funktionale Gruppen aufteilen. Es bieten sich die separate Verwaltung der Benutzerdaten (*Kunde*, *Adresse*), des Warenkorbs (*Warenkorb*, *WarenkorbArtikel*) und der vorhandenen Artikel (*Artikel*) auf verschiedenen Datenbankservern an. Bei dieser in Abbildung 2.15 beschriebenen Aufteilung wird auch der Vorteil der Partitionierung gegenüber der Replikation deutlich, da hier sowohl Lese- als auch Schreibanfragen auf alle Knoten verteilt werden können.

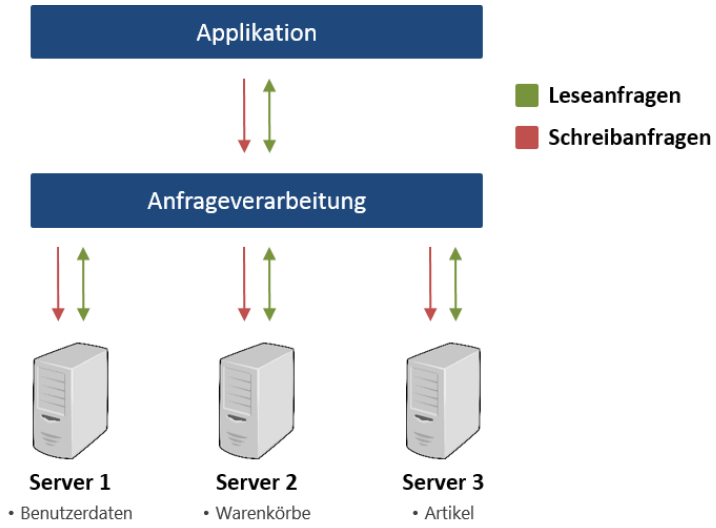


Abbildung 2.15: Anfrageverarbeitung einer funktional partitionierten Architektur

Die funktionale Partitionierung stößt an ihre Grenzen, sobald eine einzige Relation die Kapazitäten eines Servers überschreitet. Für solche Fälle stehen mit der horizontalen und der vertikalen Fragmentierung zwei bewährte Techniken zur Verfügung, mit der sich große Relationen auf mehrere kleinere verteilen lassen. Beide Fragmentierungsvarianten können durch relationale Ausdrücke spezifiziert werden [147, S.61]. Um die Korrektheit der Zerlegungen zu gewährleisten, ist es zwingend erforderlich, dass die Fragmentierung vollständig, das heißt ohne Datenverlust, erfolgt und sich die globale Relation aus den einzelnen Fragmenten vollständig wiederherstellen lässt. Die einzelnen Fragmente sollten darum möglichst überschneidungsfrei (disjunkt) sein.

Horizontale Fragmentierung: Bei der horizontalen Fragmentierung wird eine Relation zeilenweise (horizontal) in strukturell gleiche Fragmente unterteilt. Die Zuweisung der Datensätze auf die zugehörigen Fragmente erfolgt unter Zuhilfenahme eines Selektionsprädikats:

$$R_1 = \sigma_{<\text{Selektionsprädikat}>}(R) \text{ und } R_2 = \sigma_{\neg<\text{Selektionsprädikat}>}(R).$$

Da sich hierbei Vollständigkeit, Rekonstruierbarkeit und Disjunktheit leicht sicherstellen lassen, ist die horizontale Fragmentierung die einfachste und am häufigsten verwendete Fragmentierungsform. In Abbildung 2.16 ist die horizontale Fragmentierung der Benutzertabelle dargestellt. Gemäß ihrer Kundennummer werden die einzelnen Datensätze auf die verschiedenen Fragmente verteilt und daraufhin auf unterschiedlichen Servern verwaltet.

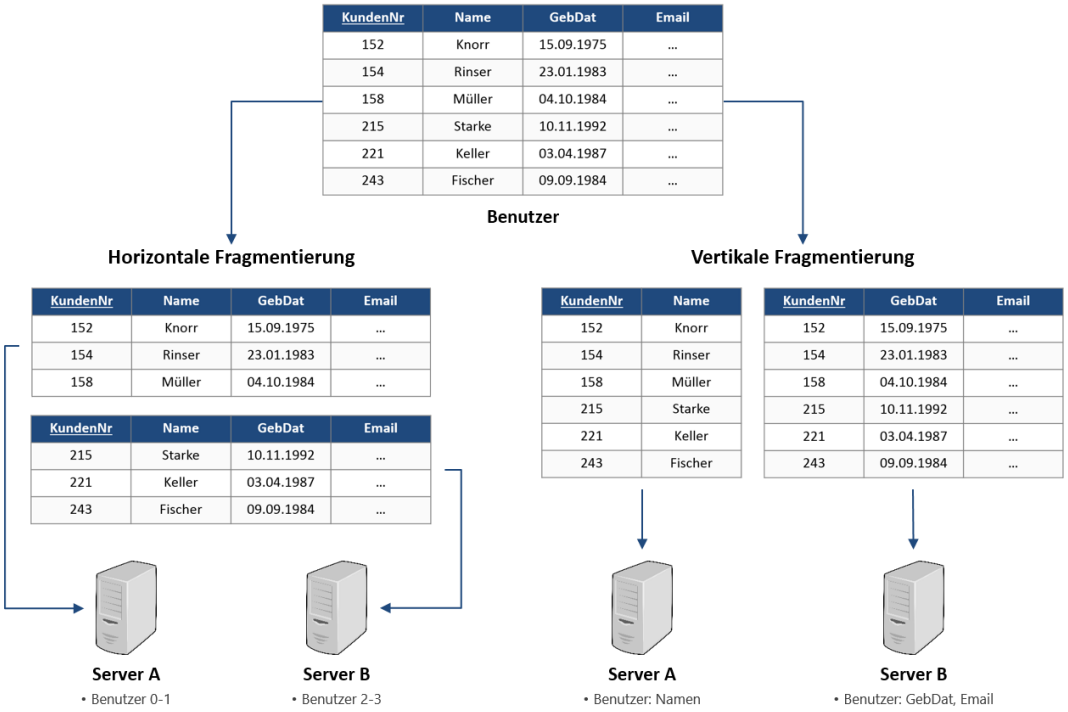


Abbildung 2.16: Gegenüberstellung der horizontalen und der vertikalen Fragmentierung.

Vertikale Fragmentierung: Bei der vertikalen Fragmentierung [138, S.98-112] wird eine Relation spaltenweise (vertikal) zerlegt. Die Aufteilung der Datensätze erfolgt hier mittels Projektion zusammengehöriger Attributgruppen. Einzelne Tupel können durch redundant gespeicherte Primärschlüssel oder Surrogate in allen Fragmenten eindeutig identifiziert und somit vollständig rekonstruiert werden:

$$R_1 = \pi_{<Projektor>}(R) \text{ und } R_2 = \pi_{\neg<Projektor>}(R).$$

Die in Abbildung 2.16 dargestellte vertikale Fragmentierung verteilt die Benutzerdaten entsprechend der Attribute. Um eine vollständige Wiederherstellung zu gewährleisten, müssen die Primärschlüssel der einzelnen Datensätze zwingend in beiden Fragmenten gespeichert werden.

Allokation

Nach der Erstellung des Fragmentierungsschemas stellt die Allokation der einzelnen Datenfragmente auf die verschiedenen Verarbeitungsrechner den zweiten Entwicklungsschritt beim Entwurf verteilter Datenbanken dar. Die Zuweisung der Fragmente wird unter Zuhilfenahme einer Verteilungsfunktion bestimmt, welche einen Schlüssel entgegennimmt und die Adresse des für diesen Schlüssel zuständigen Servers zurückgibt. Man unterscheidet zwischen statischen und dynamischen Allokationsstrategien.

Statische Allokation: Die statische Zuweisung von Fragmenten auf bestimmte Knoten kann durch den Einsatz von Hash- oder Modulo-Funktionen erfolgen:

Servernummer = Schlüssel mod Serveranzahl.

Solche Funktionen ermöglichen eine sehr schnelle und vor allem einfache Datenverteilung. Da die Allokation ausschließlich von diesen Funktionen bestimmt wird, können diese direkt in die Anwendungen integriert werden ohne dabei einen nennenswerten Overhead zu verursachen. Ein Nachteil der statischen Verteilung liegt in den sehr hohen Kosten bei der Änderung der Systemarchitektur. Wird ein Server zu einem Cluster hinzugefügt oder entfernt, kann es passieren, dass ein Großteil der Datenobjekte neu verteilt werden muss. Zusätzlich bieten Hash- oder Modulo-Funktionen eine deutlich schlechtere und nicht kontrollierbare Lastverteilung. Sind beispielsweise Attributwerte nicht gleichverteilt oder besitzen diese eine unterschiedliche Priorität bei den Anfragen, kann es sehr schnell passieren, dass ein Server eine deutlich größere Last zu verarbeiten hat als ein anderer. Statische Allokationen berücksichtigen obendrein nicht die unterschiedlichen Leistungsmerkmale der eingesetzten Rechner.

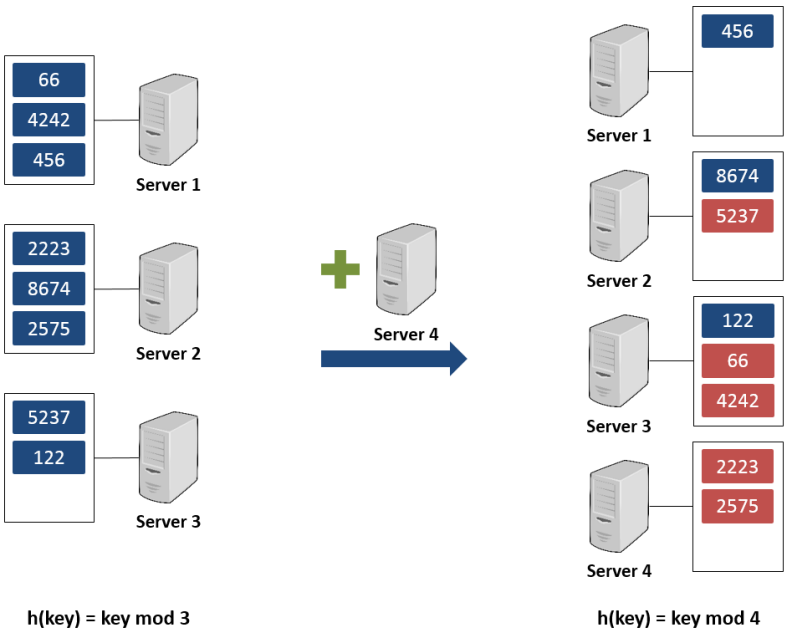


Abbildung 2.17: Reorganisationsaufwand bei statischer Allokation

Dynamische Allokation: Bei der dynamischen Allokation wird der Speicherplatz eines Fragmentes nicht mit einer statischen Funktion berechnet, sondern er kann gezielt bestimmt werden. Die zum Auffinden des Fragments notwendigen Zuweisungsinformationen werden zentral verwaltet. Hierzu bietet sich beispielsweise eine zweispaltige Tabelle an, in der zu jedem Fragment die zugehörige Serveradresse hinterlegt wird. Der Vorteil der dynamischen Allokation liegt in den feingranularen Möglichkeiten, ein Fragment gezielt einem Server zuzuweisen. Hierdurch wird eine wesentlich bessere Lastverteilung im Vergleich zur statischen Allokation ermöglicht. So können Fragmente und unterschiedliche Lasten nicht nur gleichmäßig verteilt, sondern auch unterschiedliche Leistungsstärken der Server berücksichtigt werden. Ein weiterer Vorteil der dynamischen Zuweisung liegt in der besseren Unterstützung von Verbundoperationen, da hier, im Gegensatz zur fixen Allokation, gemeinsam adressierte Daten auf dem gleichen Server abgelegt werden können. Der Nachteil der dynamischen Allokation liegt im größeren Verwaltungsaufwand. Durch die separate Speicherung der Zuweisungsinformationen entsteht ein Overhead, der die Systemkomplexität erhöht. Zudem kann bei sehr vielen Datensätzen die Verteilungstabelle extrem groß und die Suche nach der Serveradresse entsprechend ineffizient werden. Eine Lösung dieses Problems stellt die Kombination von dynamischer und statischer Allokation dar. Datensätze können zunächst mithilfe einer Funktion statischen Wertebereichen zugeordnet werden, die anschließend dynamisch auf bestimmten Servern hinterlegt werden.

Horizontale und vertikale Fragmentierung sowie die Allokation der Fragmente werden in Özsu und Valduriez [138] und in Rahm [147, S.59-77] ausführlich behandelt.

2.4.3 Verteilte Anfrageverarbeitung

Bedingt durch die Fragmentierung des globalen Datenbankschemas in mehrere kleinere, lokale Schemata und die Verteilung dieser Fragmente auf unterschiedliche Verarbeitungsrechner, gestaltet sich die Anfrageverarbeitung in verteilten Datenbanksystemen wesentlich komplexer als in nichtverteilten Datenbanksystemen. Sobald eine Anfrage an das globale Schema entgegengenommen wird, muss diese gemäß der Fragmentierung in mehrere lokale Anfragen aufgeteilt werden. Diese lokalen Anfragen sollten auf den zuständigen Knoten idealerweise parallel ausgeführt werden, um die Kosten der globalen Anfrage nicht zu stark steigen zu lassen. Am Ende müssen die erzielten lokalen Ergebnisse zu einem globalen Ergebnis zusammengeführt werden. Dieser vierstufige Prozess ist in Abbildung 2.18 anhand einer Leseanfrage nach allen Benutzern mit einem Alter über 18 Jahren dargestellt.

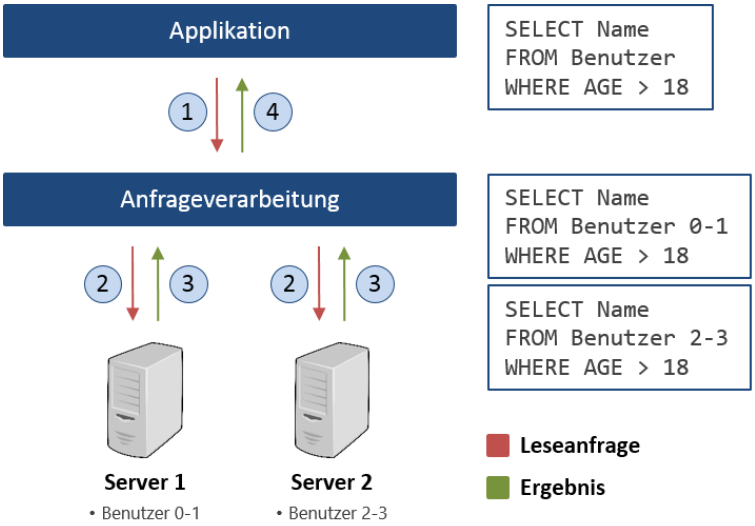


Abbildung 2.18: Vierstufiger Verarbeitungsprozess einer verteilten Leseanfrage

Die parallele Ausführung der lokalen Anfragen ist jedoch nur möglich, wenn diese nicht voneinander abhängig sind. Benötigt, wie im Fall von Verbundoperationen, eine Anfrage die Ergebnisse einer anderen, müssen diese beiden Anfragen sequenziell ausgeführt werden. Aufgrund der hohen Kommunikationskosten in verteilten Datenbanksystemen sollte die hierbei auftretende Kommunikation zwischen den einzelnen Knoten auf ein Minimum beschränkt werden. Verbundoperationen werden deswegen in der Regel nicht wie gewohnt ausgeführt, sondern unter Zuhilfenahme von Semi-Joins realisiert. Bei diesem Ansatz werden in einem ersten Schritt die für die Join-Berechnung benötigten Attributwerte der einen Relation übermittelt und in einem zweiten Schritt die zugehörigen Verbundpartner der anderen Relation zurückgesendet [147, S.104–105]. Da die Ausführungsreihenfolge dieser sequenziellen Semi-Join-Anfragen die Kosten der gesamten Verbundoperation massiv beeinflusst, ist das Optimierungspotenzial in diesem Bereich entsprechend hoch. Neben der Veränderung der Ausführungsreihenfolge sind die Denormalisierung des Datenmodells und die temporär redundante Speicherung von benötigten Daten gängige Methoden, um diese Ausführungskosten zu senken.

Durch Fragmentierung und Allokation sowie der vielfältigen Optimierungsmöglichkeiten zur Reduzierung der Kommunikationskosten besitzt die effiziente Ausführung von verteilten Anfragen eine sehr hohe Komplexität. Auch wenn diese durch eine zusätzliche Abstraktionsebene zumindest teilweise vor den Anwendern verborgen werden kann, sind Anfragen auf verteilte Datenbanken in der Regel wesentlich langsamer als Anfragen auf nichtverteilte Datenbanksysteme.

2.4.4 Verteilte Transaktionsverwaltung

Verteilte Transaktionen beinhalten Operationen, die auf mehreren Knoten ausgeführt werden müssen. Soll beispielsweise ein neuer Benutzer für den Onlineshop registriert werden, müssen die Relationen *Benutzer*, *Anschrift* und *Warenkorb* um neue Einträge erweitert werden (Abbildung 2.3). Durch höhere Kommunikationskosten und potenzielle Fehler wie Nachrichtenverluste, Netzwerkfehler oder Rechnerausfälle gestaltet sich die Integritätssicherung sowie die Verwaltung von verteilten Transaktionen deutlich komplexer als bei nichtverteilten Datenbanksystemen.

Recovery

Zur Gewährleistung der Atomarität müssen sich alle an einer verteilten Transaktion beteiligten Knoten auf ein gemeinsames Commit-Ergebnis einigen können. Entweder wird die Transaktion erfolgreich auf allen Knoten durchgeführt (globales Commit) oder auf allen Knoten zurückgesetzt (globales Abort). Die Anforderungen an ein hierfür notwendiges verteiltes Commit-Protokoll sind ein möglichst geringer Kommunikationsoverhead sowie eine hohe Robustheit gegenüber den oben genannten Fehlern in verteilten Systemen [147, S.114]. Das bekannteste Commit-Protokoll ist das in Abbildung 2.19 und Abbildung 2.20 dargestellte Zwei-Phasen-Commit-Protokoll (2PC-Protokoll).

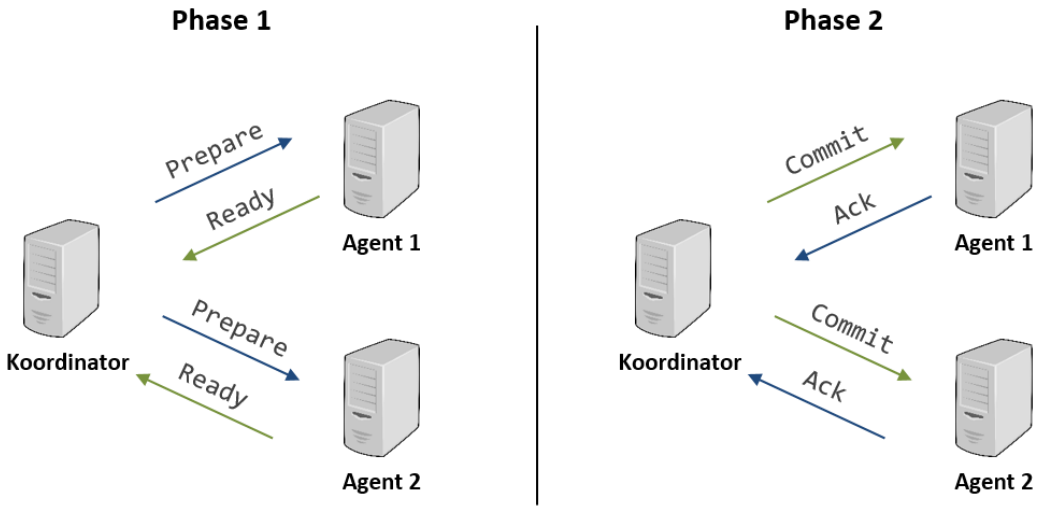


Abbildung 2.19: Abbildung einer erfolgreich durchgeführten Transaktion

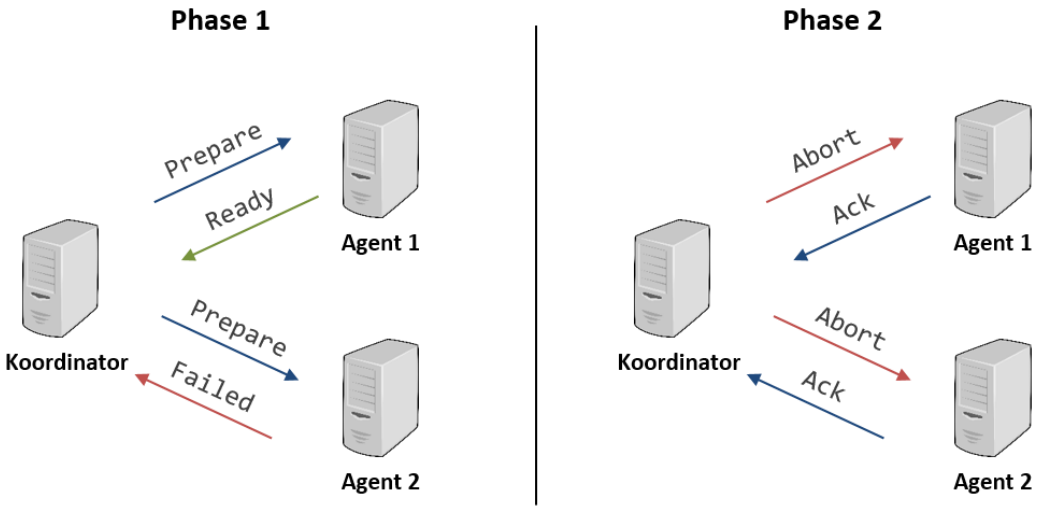


Abbildung 2.20: Abbildung einer fehlgeschlagenen Transaktion

Verteilte Transaktionen werden beim 2PC-Protokoll innerhalb von zwei Phasen von einem bestimmten Knoten koordiniert (Koordinator). Sobald die verteilte Transaktion auf allen Knoten (Agenten) durchgeführt wurde, schickt der Koordinator in der ersten Phase an jeden Agenten eine `PREPARE`-Nachricht. Je nachdem, ob ein Agent in der Lage ist, die Transaktion zu persistieren oder nicht, wird mit den Nachrichten `READY` oder `FAILED` geantwortet. Sobald der Koordinator von allen Agenten die Antwort erhalten hat, sendet er abhängig vom Ergebnis der Antworten in der zweiten Phase entweder die Nachricht `COMMIT` oder `ABORT` an alle Agenten und fordert diese zur Festschreibung oder zum Abbruch der Transaktion auf. Diese bestätigen den Erhalt der Nachricht in einem abschließenden Kommunikationsschritt mit der Nachricht `ACK`.

Mithilfe des 2PC-Protokolls lässt sich die Atomarität von verteilten Transaktionen auch in Fehlerfällen sicherstellen. Der Nachteil des Protokolls liegt in der Abhängigkeit vom Koordinator. Ist dieser wegen eines Fehlers zwischen den beiden Phasen temporär nicht verfügbar, kann es passieren, dass die Agenten in diesem Zeitraum für andere Transaktionen blockiert sind. Aufgrund dieser potenziellen Fehlerquellen wurden das Drei-Phasen-Commit-Protokoll (3PC-Protokoll) sowie verschiedene Varianten des Paxos-Protokolls entworfen, die allerdings wiederum eine deutlich höhere Komplexität aufweisen und einen größeren Kommunikationsaufwand verursachen[19].

[19] Diese Protokolle werden unter anderem in Gray und Lamport [73] ausführlich vorgestellt und deren Vor- und Nachteile miteinander verglichen.

Synchronisation

Um in verteilten Datenbanksystemen Anomalien im Mehrbenutzerbetrieb vermeiden zu können, muss eine systemweite (globale) Serialisierbarkeit von konkurrierenden Zugriffen gewährleistet werden. Da sich die einzelnen Knoten in ihrer Serialisierbarkeitsreihenfolge unterscheiden können, ist die lokale Synchronisation der Zugriffe auf den einzelnen Knoten nicht ausreichend, um eine globale Serialisierbarkeit zu garantieren. Aus diesem Grund muss die Synchronisation global koordiniert werden. Man unterscheidet zwischen zentraler und verteilter Synchronisation.

Zentrale Synchronisation: Bei der zentralen Synchronisation wird die Serialisierbarkeit durch einen zentralen Server garantiert. Jede durchgeführte Transaktion wird von diesem Server synchronisiert. Der Vorteil dieses Ansatzes liegt darin, dass hier die gleichen Synchronisationsverfahren wie in nichtverteilten Datenbanksystemen eingesetzt werden können. Allerdings verursacht dieses Verfahren einen sehr hohen Kommunikationsaufwand. Da zudem das gesamte System von der Leistungsfähigkeit und Verfügbarkeit des zentralen Knotens abhängig ist (Single Point of Failure), wird dieses Verfahren in verteilten Datenbanken in der Regel nicht angewendet [147, S.136].

Verteilte Synchronisation: Bei der verteilten Synchronisation ist jeder Knoten für die lokale Serialisierbarkeit der an ihn gerichteten Transaktionen verantwortlich. Um globale Serialisierbarkeit von verteilten Transaktionen zu gewährleisten, wird die Persistierung einer Transaktion unter Zuhilfenahme verteilter Commit-Protokolle koordiniert. Der Nachteil der verteilten Synchronisation liegt in der durchaus komplexen Behandlung von blockierten und verhungerten Transaktionen [147, S.136]. Die hierbei eingesetzten Verfahren werden in Rahm [147, S.151] ausführlich behandelt.

Zur Synchronisation von verteilten Transaktionen können die gleichen Synchronisationsverfahren wie für nichtverteilte Transaktionen eingesetzt werden. Allerdings hängt die Leistungsfähigkeit eines Verfahrens in verteilten Datenbanken sehr stark von dessen Kommunikationsumfang und von der Häufigkeit von Blockierungen und Rücksetzungen ab. Da diese Parameter bei pessimistischen Synchronisationsverfahren vergleichsweise hoch sind, „ist die für zentralisierte [Datenbanksysteme] festgestellte Überlegenheit von Sperrverfahren gegenüber anderen Synchronisationsverfahren im verteilten Fall nicht notwendigerweise gegeben“ [147, S.160]. Dennoch setzen verteilte relationale Datenbanken in der Regel pessimistische Sperrverfahren ein.

2.5 Zusammenfassung

Relationale Datenbanken dominieren seit Jahrzehnten den Datenbankmarkt und sind auch heute noch die kommerziell am häufigsten eingesetzten Datenbanksysteme. Infolge des enormen Erfahrungsschatzes, der sich über die Jahre im produktiven Umgang mit relationalen Datenbanken aufgebaut hat, handelt es sich bei relationalen Datenbanken um sehr ausgereifte Systeme mit einer hohen Stabilität. Dank eines ausdrucksstarken Datenmodells, einer mächtigen Anfragesprache und eines ACID-kompatiblen Transaktionskonzepts haben sich relationale Datenbanken als quasi Standardlösung im Bereich der Datenhaltung etabliert. Der sehr hohe Funktionsumfang von relationalen Datenbanken ist in Tabelle 2.6 dargestellt.

Funktionalitäten		Relationale Datenbanken
Ausdrucksstarkes Datenmodell	Attributnamen	✓
	Datentypen	✓
Anfragemächtigkeit	Relationale Operatoren	✓
	Verbundoperationen	✓
	Sekundäre Zugriffspfade	✓
ACID-Eigenschaften	Atomarität	✓
	Integritätsbedingungen	✓
	Synchronisationsmechanismen	✓
	Dauerhaftigkeit	✓
Verteilte Datenhaltung	Replikation	✗
	Verteilung	✗

Tabelle 2.6: Funktionsumfang von relationalen Datenbanken

Aufgrund des hohen Funktionsumfangs besitzen relationale Datenbanksysteme allerdings eine sehr hohe Systemkomplexität, welche die Verteilung auf mehrere Verarbeitungsrechner deutlich erschwert. Hinzu kommt die Problematik, dass die in diesem Kapitel vorgestellten Techniken, wie Replikation, Fragmentierung, Allokation und verteilte Transaktionsverwaltung, von den meisten relationalen Datenbanksystemen nicht ausreichend unterstützt werden[20]. Entsprechend hoch ist der zusätzliche Aufwand, der betrieben werden muss, um ein relationales Datenbanksystem auf mehrere Server verteilen zu können. Obendrein ist ein fundiertes Fachwissen in dieser Materie zwingend erforderlich. Deutlich wird diese Problematik beispielsweise bei der Verteilung des beliebtesten freiverfügbaren Datenbanksystems MySQL. Obwohl MySQL im

[20] Eine Übersicht über die unterstützten Funktionalitäten ist unter http://db-engines.com zu finden. Detaillierte Informationen sind den Dokumentationen der Anbieter zu entnehmen.

Vergleich zu anderen Datenbanksystemen einen relativ großen Funktionsumfang für die Verteilung auf mehrere verschiedene Server bietet, müssen dennoch sehr viele der oben genannten Funktionalitäten auf Anwendungsebene realisiert werden. So ist beispielsweise ein MySQL Server in der Lage, mehrere Replikate asynchron zu aktualisieren, jedoch muss die Verteilung der Leselast applikationsseitig erfolgen [129, S.10]. Die von MySQL angebotene horizontale Fragmentierung steht ausschließlich in nichtverteilten Datenbanken zur Verfügung [128, Kap.18]. Demzufolge müssen Allokation, Fragmentierung und Anfrageverarbeitung für verteilte Architekturen auf Anwendungsebene durchgeführt werden [127, S.11]. Außerdem kann sich die Wartung eines verteilten MySQL-Datenbanksystems als sehr kostspielig erweisen, weil die für Überwachung und Fehlerbehandlung benötigten Werkzeuge ausschließlich in der kostenpflichtigen Enterprise-Edition zur Verfügung stehen [130, S.12]. Zusätzlich müssen die mangelnde Unterstützung für das dynamische Hinzufügen und Entfernen von Knoten sowie die daraufhin notwendige, aber bei MySQL fehlende Rebalancierung des Clusters [130, S.24], berücksichtigt werden.

Auch wenn relationale Datenbanken eine umfangreiche Unterstützung für eine Verteilung auf mehrere Server vermissen lassen, müssen nicht sämtliche fehlenden Funktionalitäten applikationsseitig neu entwickelt werden. Inzwischen stehen sehr viele Drittanbieterprodukte wie *Hibernate Shards*[21], *HiveDB*[22], *Shard-Query*[23], *ScaleBase*[24], *ScalArc*[25] und *dbShards*[26] zur Verfügung, die einige dieser Aufgaben übernehmen können [155, S.547]. Daneben ergeben sich mit Aufkommen der NewSQL-Bewegung [160] mehrere interessante Alternativen. Bei diesen Systemen handelt es sich um relationale Datenbanken, die von Beginn an auf eine verteilte Datenhaltung ausgelegt sind.

[21] http://shards.hibernate.org
[22] http://www.hivedb.org
[23] http://code.google.com/p/shard-query
[24] http://www.scalebase.com
[25] http://scalarc.com
[26] http://www.dbshards.com

3

KEY VALUE STORES

Kapitelinhalt

- Grundlagen von Key Value Stores
- Datenmodellierung mit Schlüssel-Wert-Paaren
- Verteilte Datenhaltung mit Consistent Hashing

Key Value Stores (KVS) stellen angesichts ihres simplen Datenmodells und ihres geringen Funktionsumfangs die einfachste Klasse der NoSQL-Datenbanken dar. Diese Systeme sind auf eine möglichst hohe Geschwindigkeit von einfachen Lese- und Schreiboperationen ausgelegt und verfügen in Anbetracht ihres Minimalismus über ein entsprechend kleines Spektrum an potenziellen Einsatzgebieten. Zu den bekanntesten Key Value Stores zählend *Dynamo* [45], *Redis*[27], *Project Voldemort*[28] und *Riak*[29].

In diesem Kapitel werden die Konzepte, welche Key Value Stores zugrunde liegen, systematisch aufgearbeitet. Hierdurch wird ein in der Literatur bisher nicht behandelter Bereich beleuchtet, der für das Verständnis dieser Systeme jedoch unabdingbar ist. Das Kapitel startet mit einer konzeptuellen Rekonstruktion des Datenmodells und einer detaillierten Analyse der Anfragefunktionalitäten. Basierend auf den Ergebnissen werden daraufhin Modellierungsmuster erstellt, mit deren Hilfe sich konzeptuelle Datenmodelle systematisch auf die logischen Datenmodelle von Key Value Stores überführen lassen. Anschließend erfolgt eine nähere Betrachtung der technischen Eigenschaften, wie Speicherstrukturen und Transaktionsverwaltung. Das Kapitel schließt mit der Vorstellung von Replikations- und Verteilungstechniken, die in Key Value Stores zum Einsatz kommen.

[27] http://www.redis.io
[28] http://www.project-voldemort.com
[29] http://www.basho.com/riak

3.1 Einführung

Key Value Stores wie *Berkeley DB*[30] existieren bereits seit den 1990er Jahren. Sie werden aufgrund ihrer sehr guten Performanzeigenschaften bei einfachen, schlüsselbasierten Anfragen häufig als Caching-Systeme in allen möglichen Bereichen der Informationsverarbeitung eingesetzt. Vor allem das hochgradig performante, verteilte Caching-System *Memcached*[31] hat sich in den Architekturen großer Webapplikationen etabliert.

Memcached verbindet den ungenutzten Hauptspeicher der eingesetzten Datenbankserver zu einem großen Key Value Store, in dem vor allem häufig benötigte Datensätze gespeichert werden. Bei anschließenden Leseanfragen wird zunächst in Memcached nachgesehen, ob der gesuchte Datensatz dort hinterlegt ist. Ist dies der Fall, werden die Leseanfragen direkt aus dem Cache bedient und die Leselast auf das (meist relationale) Datenbanksystem deutlich reduziert. Befindet sich der Datensatz nicht in Memcached, wird er aus der Datenbank gelesen und für zukünftige Zugriffe in Memcached hinterlegt (Abbildung 3.1).

```
function getData(dataId) {
    /* get data from memcached */
    data = memcached.get(dataId);
    if (!data) {
         /* if data is not found, request database */
         data = getDataFromDatabase(dataId);
         /* store data in memcached */
         memcached.set(dataId, data);
    }
    return data;
}
```

Abbildung 3.1: Caching mithilfe von Memcached

Memcached erfreut sich durch die effiziente Auslastung des verfügbaren Arbeitsspeichers großer Beliebtheit. Die bereitgestellte Schnittstelle bietet außer den primitiven Methoden `get`, `put` und `remove` keine weiteren Zugriffsmöglichkeiten. Eine Unterstützung für persistente Speicherung, Fehlerbehandlung oder Wiederherstellung ist ebenfalls nicht im Funktionsumfang enthalten. Im Fall eines Hauptspeicherfehlers sind die in Memcached befindlichen Datenobjekte verloren.

Key Value Stores werden jedoch nicht nur wegen ihrer hohen Performanz bei schlüsselbasierten Anfragen geschätzt. Dank ihrer hervorragenden Skalierbarkeitseigenschaften werden sie in den letzten Jahren vermehrt auch als persistente Speichersysteme in sehr großen Datenbankclustern eingesetzt und ermöglichen die zuverlässige

[30] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb
[31] http://www.memcached.org

Verarbeitung extrem großer Datenmengen. Diese Entwicklung ist maßgeblich durch die Veröffentlichung des von Amazon entworfenen Key Value Stores Dynamo geprägt. Dynamo ist ein auf mehrere zehntausend Server verteiltes Datenbanksystem, das die Daten wichtiger Services der weltweit genutzten Onlineplattform bereitstellt. Zu diesen Services zählen unter anderem Bestsellerlisten, Einkaufswagen, Benutzerpräferenzen, Session-Management, Verkaufsplatzierungen und Produktkataloge. Im Fall von Amazon verursachen auch nur kurzfristige Ausfälle dieser Services erhebliche finanzielle Schäden. Aus diesem Grund war neben linearer Skalierbarkeit die nahezu permanente Verfügbarkeit des Systems ein Kernziel bei der Implementierung von Dynamo [45].

Die Veröffentlichung von Dynamo inspirierte die Entwicklung einer Reihe von quelloffenen Systemen, zu deren bekanntesten Vertretern der Key Value Store Riak und das von LinkedIn finanzierte System Project Voldemort zählen. Die diesen Speichersystemen und dem hochgradig performanten Key Value Store Redis zugrunde liegenden Konzepte werden im Folgenden näher vorgestellt.

3.2 Konzeptuelle Ebene

Key Value Stores weisen das einfachste Datenmodell und die geringsten Anfragemöglichkeiten aller in dieser Arbeit vorgestellten Datenbanken auf. Über eine Schnittstelle können Datenobjekte (Values) gespeichert werden. Anschließend sind diese Datenobjekte ausschließlich mittels eines zugehörigen Schlüssels (Key) wieder abrufbar.

3.2.1 Datenmodell

Ein Datensatz in einem Key Value Store besteht aus genau einem Schlüssel und genau einem Wert. Mehrere solcher Schlüssel-Wert-Paare werden in sogenannten Buckets verwaltet. Ein Bucket ähnelt daher einer zweispaltigen relationalen Tabelle. Die Primärschlüssel-Spalte dieser Tabelle enthält die verschiedenen Schlüssel der Datensätze, während die zweite Spalte den jeweiligen Attributwert eines Datensatzes beinhaltet. Die entsprechenden Strukturbezeichnungen sind in Tabelle 3.1 dargestellt.

Relationale Datenbank	Key Value Store
Tabelle	Bucket
Tupel (Datensatz)	Schlüssel-Wert-Paar
Primärschlüssel (ID)	Schlüssel
Attribut	Wert

Tabelle 3.1: Gegenüberstellung der unterschiedlichen Strukturbezeichnungen

Schlüssel und Werte eines Datensatzes werden als einfache Byte-Arrays gespeichert, die vom Datenbanksystem nicht interpretiert werden können. Sie entsprechen deswegen dem relationalen Datentyp `BLOB`[32]. Primitive Datentypen wie Zahlen, boolesche Werte oder Zeichenketten können damit zwar gespeichert werden, eine Auswertung auf Datenbankebene ist jedoch nicht möglich. Es ist deshalb Aufgabe der Applikationslogik, ein Key-Value-Paar nach dem Lesen aus einem Key Value Store in das gewünschte Datenformat zu überführen.

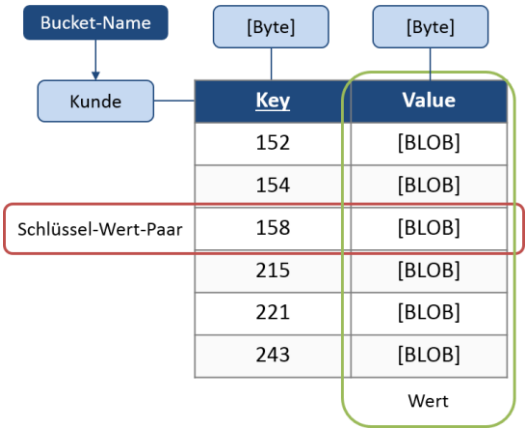


Abbildung 3.2: Logisches Datenmodell der Entität *Kunde*

Abbildung 3.2 beschreibt die Abbildung der aus Kapitel 2 bekannten Entität *Kunde* (Abbildung 2.3) auf das logische Datenmodell eines Key Value Stores. Das Bucket *Kunde* enthält, ähnlich der relationalen Darstellung (Abbildung 2.4), für jeden Datensatz die Kundennummer als Primärschlüssel. Die Attributwerte werden als Byte-Arrays gespeichert und müssen daher auf Applikationsebene interpretiert werden.

Key Value Stores sind aufgrund des minimalistischen Datenmodells nicht in der Lage, Metainformationen der in ihnen gespeicherten Daten zu verwalten. So können weder die Namen des Primärschlüssels noch die Namen der Attribute im Datenmodell gespeichert werden. Auch ist eine Definition von Wertebereichen nicht möglich. Die extreme Ausdrucksschwäche des Datenmodells wird vor allem bei der Betrachtung des zugehörigen Prädikatorenschemas deutlich (Abbildung 3.3).

[32] Binary Large Object

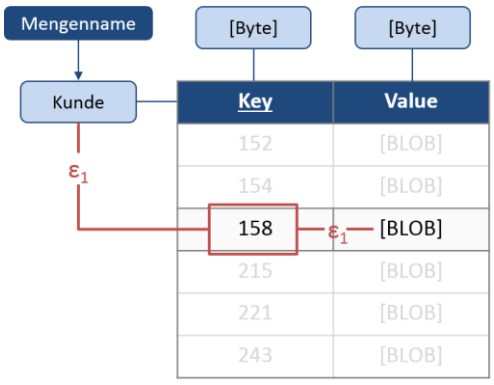


Abbildung 3.3: Prädikatorenschema des Buckets *Kunde*

In einem Key Value Store werden ausschließlich zwei Prädikationen erster Ordnung gespeichert. Hierbei handelt es sich um die Prädikation zwischen dem Namen der Menge und den entsprechenden Schlüsselwerten sowie um die Prädikation zwischen den Schlüsselwerten und den zugehörigen Attributwerten. Weitere Metainformationen wie Schlüsselnamen oder Attributnamen, die für das Verständnis der gespeicherten Daten unabdingbar sind, werden in Key Value Stores nicht hinterlegt. Neben der fehlenden Abbildung grundlegender Metainformationen können in Key Value Stores auch keine Datentypen oder Fremdschlüsselbeziehungen definiert werden. Da Key Value Stores folglich kein Datenbankschema im eigentlichen Sinn aufweisen, werden diese auch als *schemafrei* bezeichnet.

Datenintegrität

Durch die fehlenden Abbildungsmöglichkeiten von Metainformationen und Fremdschlüsselbeziehungen im Datenmodell bieten Key Value Stores folglich weder statische noch dynamische Funktionalitäten zur Sicherung der Datenintegrität. Selbst die Eindeutigkeit der Schlüssel (Entity-Integrität) wird von diesen Systemen bei Schreiboperationen standardmäßig nicht geprüft. Wird ein neues Key-Value-Paar gespeichert, dessen Schlüssel bereits im Bucket vorhanden ist, wird das bestehende Key-Value-Paar in der Regel ohne Vorwarnung überschrieben. Manche APIs ermöglichen daher eine optionale Eindeutigkeitsprüfung bei Einfüge-Operationen

3.2.2 Anfrageverarbeitung

Die von Key Value Stores angebotenen Anfrage- und Änderungsoperationen sind auf ein absolutes Minimum reduziert. Zur Manipulation des Datenbestandes stehen ausschließlich die beiden Änderungsoperationen `set` und `delete` zur Verfügung. Mit diesen Operationen können einzelne Schlüssel-Wert-Paare gespeichert und gelöscht werden. Eine gezielte Änderung eines Schlüssel-Wert-Paares kann nur durch das Einfügen einer neuen Version des Datensatzes erfolgen. Zum Auslesen eines Datensatzes

steht in der Regel nur die Anfrageoperation `get` zur Verfügung. Die drei Basisoperationen von Key Value Stores sind in Abbildung 3.4 dargestellt.

```
> SET <key> <value>

> GET <key>

> DEL <key>
```

Abbildung 3.4: Die Operationen *get*, *set* und *delete*

Die Anfrageverarbeitung von Key Value Stores ist nicht mengenorientiert, sondern satzorientiert. Jede Operation nimmt ausschließlich den Primärschlüssel eines Datensatzes als Parameter entgegen und gibt als Ergebnis maximal ein Schlüssel-Wert-Paar zurück.

Schnittstellen

Zur Kommunikation mit Key Value Stores können Clients sowohl plattformunabhängige Protokolle wie das Representational State Transfer-Protokoll (REST) verwenden, als auch auf eine der vielen spezifische Schnittstellen für nahezu alle gängigen Programmiersprachen zugreifen. Eine deklarative Anfragesprache wie SQL wird nicht angeboten.

Um auf Applikationsebene den Umgang mit einfachen Byte-Arrays zu erleichtern, stellen manche Key Value Stores zusätzliche Klassen bereit, mit denen sich Byte-Arrays in gängige Datenformate wie Strings, Collections (Listen, Maps, Sets) oder Objekte (Java Serialization, Protocol Buffer, Avro, Thrift) umwandeln lassen. Je nach eingesetzter Schnittstelle werden in diesen Systemen die drei Basisoperationen `get`, `set` und `delete` um weitere, datentypspezifische Operationen erweitert. Auf diese Weise können beispielsweise Elemente gezielt in Kollektionen eingefügt und angesprochen werden. Einen überdurchschnittlich großen Umfang an solchen Operationen bietet vor allem der Key Value Store Redis, mit dem sich Kollektionen ähnlich bequem wie auf Applikationsebene verwalten lassen [149].

Trotz des in manchen Systemen vereinfachten Umgangs mit binären Daten bleibt dennoch festzuhalten, dass Key Value Stores nur einen minimalen Umfang an Anfrage- und Änderungsoperationen vorzuweisen haben. Abgesehen von der gezielten Selektion von Datenobjekten in Kollektionen unterstützen Key Value Stores ausschließlich Restriktionen und sind somit nicht relational vollständig. Da zur Formulierung von Restriktionsbedingungen einzig der Gleichheitsoperator zur Verfügung steht, kann immer nur ein Schlüssel-Wert-Paar ausgelesen werden. Key Value Stores sind demzufolge ausschließlich für einfache, schlüsselbasierte Anfragen geeignet. Sobald jedoch komplexere Operationen benötigt werden, kann sich ein erheblicher Mehraufwand einstellen, da fehlende Funktionalitäten auf Anwendungsebene nachimplementiert werden

müssen und die Anzahl der Datenbankaufrufe deutlich steigt. Die von Key Value Stores bereitgestellten Operationen und Operatoren sind in Tabelle 3.2 abgebildet.

Relationale Algebra	Key Value Stores
Projektion	✗
Restriktion	✓
Kartesisches Produkt	✗
Vereinigung	✗
Differenz	✗
Umbenennung	✗
Operator	**Key Value Stores**
Gleich	✓
Ungleich	✗
Größer, Kleiner	✗
Like	✗
In	✗
Exists	✗
And, Or	✗
Not	✗

Tabelle 3.2: Unterstützte relationale Operationen und Operatoren

Datenobjekte in Key Value Stores können mit einem Zeitstempel (Time To Live – TTL) versehen werden, nach dessen Ablauf das Objekt gelöscht wird. Solche Funktionen sind insbesondere für Sitzungsdaten in Web-Applikationen hilfreich. Redis ermöglicht durch die Inkrementierungsoperation `incr` die Erhöhung eines numerischen Wertes und erleichtert so unter anderem die Implementierung eines Counters.

3.2.3 Datenmodellierung

Im Gegensatz zur relationalen Entwurfstheorie existieren bei Key Value Stores keine vergleichbaren Abbildungsregeln und Modellierungsprozesse, mit deren Hilfe sich konzeptuelle Datenmodelle systematisch in die logischen Datenbankmodelle von Key Value Stores überführen lassen. Die fehlenden Abbildungsregeln sind jedoch unerlässlich, um sowohl die Möglichkeiten als auch die Grenzen von Key Value Stores nachvollziehen zu können. Zudem bieten sie zusammen mit Modellierungsprozessen vor allem unerfahrenen Anwendern einen deutlich leichteren Einstieg in den effizienten Umgang mit Key Value Stores. Diese fehlenden Beiträge auf konzeptueller Ebene werden im Folgenden nachgereicht. Basierend auf einer konzeptuellen Aufarbeitung und einer umfangreichen Analyse der unterstützten Operationen und Operatoren werden zunächst allgemeingültige Modellierungsmuster erstellt. Mithilfe der hierdurch ge-

wonnenen Erkenntnisse im Umgang mit Key Value Stores wird ein Modellierungsprozess entworfen, mit dem sich die Elemente des konzeptuellen Datenmodells systematisch auf die Schlüssel-Wert-Paare von Key Value Stores übertragen lassen.

Entitäten

Bedingt durch die Schemafreiheit von Key Value Stores wird die Datenmodellierung bei diesen Systemen, im Gegensatz zur relationalen Datenmodellierung, nicht durch die explizite Abbildung von Metainformationen bestimmt. Folglich besitzen Strukturinformationen wie Attributnamen, Datentypen oder Fremdschlüssel bei der Überführung von Entitäten in einzelne Schlüssel-Wert-Paare eine untergeordnete Rolle. Einen wesentlich größeren Einfluss auf die Datenmodellierung mit Key Value Stores besitzen das minimalistische Datenmodell und vor allem die geringe Anfragemächtigkeit. Da Schlüssel die einzige Möglichkeit darstellen, um mit Datenobjekten in Key Value Stores interagieren zu können, bestimmt deren Modellierung die Performanz und den Umfang der zu unterstützenden Anfragen. Die Entscheidung für oder gegen eine bestimmte Abbildungsvariante von Entitäten hängt daher in erster Linie von den Zugriffsmustern des Anwendungsfalls ab.

Trotz der praktisch nicht vorhandenen Abbildungsregeln, mit denen Entitäten vom konzeptuellen in das logische Datenmodell transformiert werden können, kristallisieren sich bei näherer Betrachtung angesichts des minimalistischen Datenmodells zwei grundlegende Muster bei der Abbildung von Entitäten heraus. Entitäten können entweder durch jeweils ein Schlüssel-Wert-Paar repräsentiert werden oder sie können fragmentiert und auf jeweils mehrere Schlüssel-Wert-Paare abgebildet werden. Diese beiden Muster werden im Folgenden als atomare und fragmentierte Abbildung bezeichnet. Zur Veranschaulichung der verschiedenen Modellierungsmuster wird die in Abbildung 3.5 dargestellten Entität *Kunde* herangezogen.

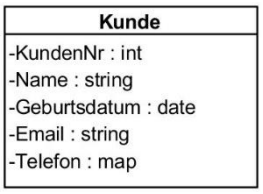


Abbildung 3.5: Konzeptuelle Darstellung der Entität *Kunde*

Atomare Abbildung: Bei der atomaren Abbildung wird eine Entität als ein einzelner Wert gespeichert, der ausschließlich über einen Schlüssel ausgelesen werden kann. In der Regel bietet sich für die Wahl des Schlüssels der Primärschlüssel der Entität an. Aufgrund der Byte-Array-Datenstruktur in Key Value Stores können für die Abbildung der Entität verschiedene Datenformate wie Zeichenketten, Java Serialization oder XML verwendet werden. Allerdings müssen die Datenobjekte hierzu nach dem Auslesen aus der Datenbank durch applikationsseitige Transformationen in die ge-

wünschten Datenformate überführt werden. Eine solche Transformation ist in Abbildung 3.6 exemplarisch für alle weiteren Beispiele in diesem Kapitel dargestellt. Aus Gründen der Übersichtlichkeit wird allerdings in den folgenden Beispielen auf diese explizite Darstellung verzichtet.

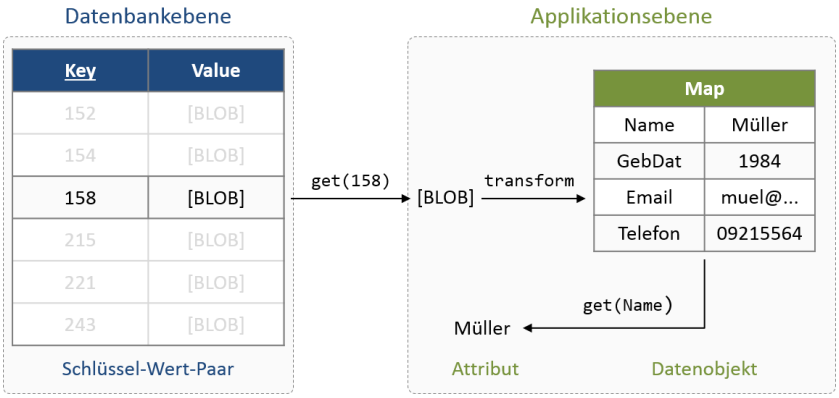


Abbildung 3.6: Atomare Abbildung der Entität *Kunde*

Abbildung 3.6 beschreibt die atomare Abbildung der Entität *Kunde* als ein einfaches Datenobjekt. Über den Schlüssel *158* kann der gesuchte Datensatz direkt ausgelesen und nach einer Transformation in das gewünschte Format umgewandelt werden. Neben dem hierdurch entstehenden Mehraufwand werden in Abbildung 3.6 zwei weitere Nachteile der atomaren Abbildung deutlich. Infolge der ausschließlich schlüsselbasierten Anfragen auf Datenbankseite ist es unmöglich, einen Kunden anhand eines beliebigen Attributes, wie beispielsweise dessen Geburtsdatum, zu finden. Solche Informationen sind im Datenobjekt enthalten und dadurch vor dem Datenbanksystem verborgen. Hieraus ergibt sich der zweite Nachteil dieser Modellierungsvariante. Um ein Attribut gezielt verarbeiten zu können, müssen zunächst das vollständige Datenobjekt aus der Datenbank gelesen und anschließend die benötigten Informationen auf Applikationsebene mit einem entsprechenden Mehraufwand herausgefiltert werden.

Fragmentierte Abbildung: Bei der fragmentierten Abbildung wird eine Entität auf mehrere Schlüssel-Wert-Paare verteilt. Die einzelnen Fragmente lassen sich über einen zusammengesetzten Primärschlüssel direkt ansprechen, wodurch gezielte Anfragen auf bestimmte Fragmente ermöglicht werden. Bei der Modellierung von zusammengesetzten Schlüsseln bietet sich die unten abgebildete Notation an. Dabei wird der Primärschlüssel der Entität mithilfe eines Doppelpunktes mit dem Schlüsselkandidaten des Fragments verknüpft:

`<Primärschlüssel>:<Fragmentschlüssel>`.

Werden Anfragen auf einzelne Attribute in den unterstützten Anwendungsfällen häufiger ausgeführt, stellt die fragmentierte Abbildung eine lohnenswerte Alternative zur

atomaren Abbildung dar. Einzelne Attribute werden in solchen Fällen jeweils auf ein Fragment verteilt und der Attributname als Fragmentschlüssel verwendet.

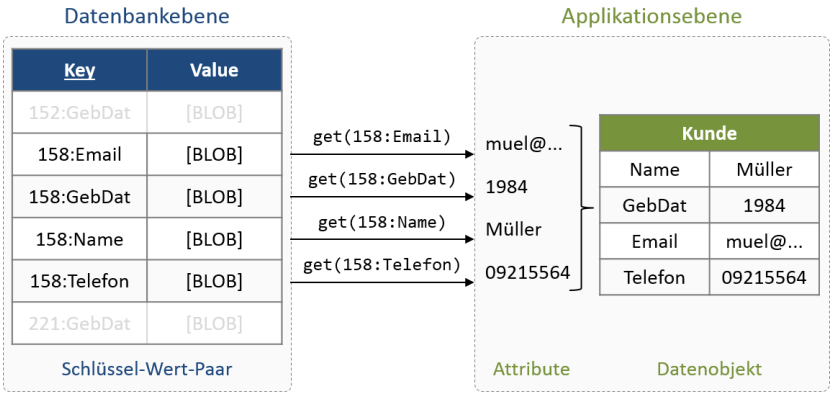


Abbildung 3.7: Fragmentierte Abbildung der Entität *Kunde*

Die fragmentierte Abbildung der Entität *Kunde* ist in Abbildung 3.7 dargestellt. Hier wird deutlich, dass durch dieses Modellierungsmuster ein hoher Mehraufwand entstehen kann, wenn mehrere Attribute einer Entität ausgelesen werden sollen oder die Entität sogar vollständig rekonstruiert werden muss. In diesem Fall muss für jedes Attribut eine zusätzliche Datenbankanfrage ausgeführt und die einzelnen Ergebnisse nach der jeweiligen Transformation auf Anwendungsebene in das gewünschte Datenformat zusammengeführt werden. Zur Rekonstruktion der Entität *Kunde* sind dementsprechend mindestens vier Anfragen notwendig.

Der Nachteil der nicht vorhandenen Typisierung von Werten hat wiederum den Vorteil, dass in einem Bucket sehr heterogene Datenobjekte gespeichert werden können. Die Verarbeitung sowohl von mehrwertigen und kompositen Attributen als auch von generalisierten und spezialisierten Entitäten kann deswegen mit beiden hier vorgestellten Modellierungsmustern problemlos im gleichen Bucket erfolgen.

Beziehungen

Ein Schlüssel-Wert-Paar kann auf den Schlüssel eines anderen Schlüssel-Wert-Paares verweisen und dieses somit referenzieren. Allerdings kann diese Beziehung nicht mithilfe von Operationen datenbankseitig aufgelöst werden (Tabelle 3.2). Die in der relationalen Entwurfstheorie festgestellten Vorteile der Referenzierung gegenüber anderen Modellierungsmustern, wie beispielsweise der Denormalisierung, sind darum bei Key Value Stores nicht zwangsläufig zu erwarten. Aus diesen Gründen ist eine Neubewertung der verschiedenen Modellierungsmuster zur Abbildung von Beziehungen zwingend erforderlich. Neben der Referenzierung und der Denormalisierung wird im Folgenden auch auf die Aggregation eingegangen und deren Vor- und Nachteile unter-

sucht. Zur Veranschaulichung der verschiedenen Modellierungsmuster dient das bereits bekannte und in Abbildung 3.8 erneut dargestellte konzeptuelle Datenmodell eines Online-Shops.

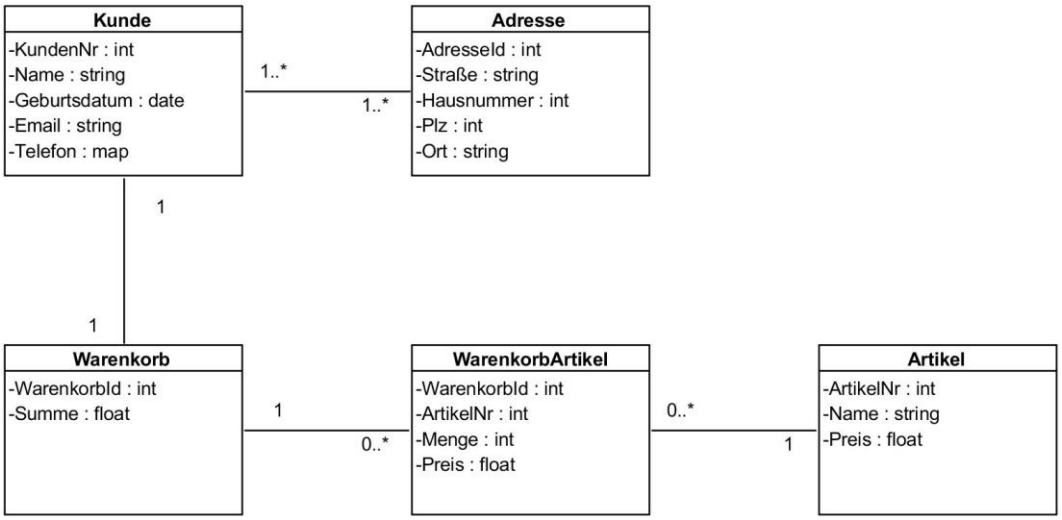


Abbildung 3.8: Konzeptuelles Datenmodell eines Online-Shops

Referenzierung: Bei der Referenzierung werden Beziehungen über Fremdschlüssel dargestellt. Hierbei verweist ein Key-Value-Paar auf den Schlüssel eines anderen. Da Key Value Stores jedoch weder Verbundoperationen unterstützen noch Techniken zur Sicherung der referenziellen Integrität vorweisen können, muss die Auflösung und Wartung von Referenzen manuell erfolgen, das heißt, diese fehlenden Funktionalitäten müssen auf Applikationsebene nachimplementiert werden.

Eine weitere Hürde bei der Verwendung von Referenzen ergibt sich durch die ausschließlich primärschlüsselbasierten Anfragen. Da Key Value Stores nicht über sekundäre Zugriffspfade verfügen, können auch keine Anfragen auf Fremdschlüssel ausgeführt werden. Aus diesem Grund können Beziehungen nur unidirektional von der Seite aus aufgelöst werden, deren Entität die entsprechende Referenz speichert. Soll beispielsweise der Warenkorb zu einem Kunden ermittelt werden, muss die Warenkorb-ID in der Entität *Kunde* gespeichert werden. Enthielte alternativ der Warenkorb die Beziehungsinformation, müssten sämtliche Warenkorb-Datenobjekte vollständig ausgelesen und verarbeitet werden, um die gesuchte Kunden-ID zu ermitteln. Gerade bei größeren Datenmengen ist diese Iteration über sämtliche Warenkorb-Datenobjekte hinweg nicht effizient durchführbar. Ist eine bidirektionale Auflösung einer Beziehung gewünscht, müssen die entsprechenden Schlüssel redundant in beiden Entitäten gespeichert werden. Da jedoch in dem oben aufgeführten Beispiel zwischen einem Kunden und seinem Warenkorb eine 1:1-Beziehung besteht, bietet sich in diesem Fall eine gemeinsame ID für beide Entitäten an. Hierdurch wird eine redundante Abbildung von Kunden-ID und Warenkorb-ID vermieden.

Das Fehlen sekundärer Zugriffspfade wirkt sich vor allem auf die Abbildung von mehrwertigen Beziehungen aus. So erweist sich die aus der relationalen Datenmodellierung gewohnte Abbildung einer 1:N-Beziehung auf der N-Seite der Beziehung als ausgesprochen ineffizient. In diesem Fall müssten sämtliche Datensätze aus der Datenbank ausgelesen und applikationsseitig untersucht werden. Eine wesentlich bessere Alternative stellt die Verwendung von Datentypen wie `Array`, `List` und `Set` dar. Mithilfe dieser Datentypen können mehrere Fremdschlüssel auf der 1-Seite der 1:N-Beziehung gespeichert werden. Die Auflösung der Beziehung umfasst bei dieser Modellierungsvariante nicht mehr alle Datensätze eines Buckets, sondern nur noch die tatsächlich referenzierten Datensätze. Dieser Modellierungsansatz muss auch zur Abbildung von M:N-Beziehungen herangezogen werden. Da diese Beziehungen nicht wie gewohnt mit Join-Tabellen abgebildet werden können, müssen diese in zwei entgegen gerichtete 1:N-Beziehungen umgewandelt werden.

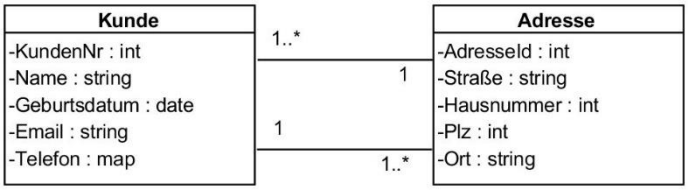


Abbildung 3.9: Konzeptuelle Abbildung einer M:N-Beziehung mit zwei 1:N-Beziehungen

Abbildung 3.9 beschreibt auf konzeptueller Ebene die alternative Darstellung einer M:N-Beziehung durch zwei entgegen gerichtete 1:N-Beziehungen. Der Nachteil dieser Abbildung ist die redundante Abbildung von Beziehungsinformationen, was eine Steigerung des Wartungsaufwandes nach sich zieht.

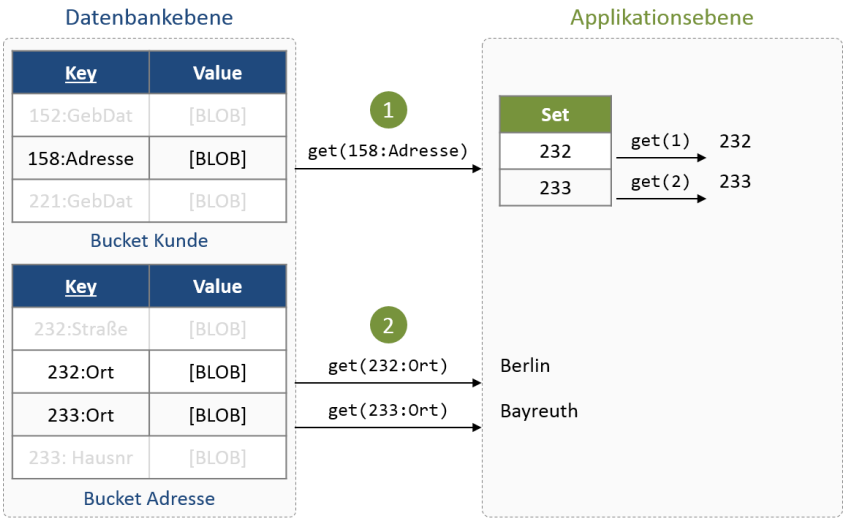


Abbildung 3.10: Zweistufige Auflösung von referenzierten Adressen

Infolge der fehlenden Unterstützung von Key Value Stores zur Auflösung von Referenzen müssen Fremdschlüsselbeziehungen applikationsseitig aufgelöst werden. Die zur Auflösung der mehrwertigen Beziehungen zwischen einem Kunden und seinen Adressen benötigten Schritte sind in Abbildung 3.10 dargestellt. Zunächst wird das Schlüssel-Wert-Paar mit den Adressreferenzen des Kunden ausgelesen und auf Applikationsebene beispielsweise als Datentyp `Set` interpretiert. Für jede Referenz müssen dann in einem zweiten Schritt die zugehörigen Orte ausgelesen werden. Hinsichtlich dieses Mehraufwands bietet sich die Referenzierung nur in Anwendungsfällen an, in denen die Anzahl der an einer Beziehung beteiligten Entitäten gering ist.

Denormalisierung: Bei der Denormalisierung werden bestimmte Beziehungsinformationen redundant in mehreren Entitäten gespeichert. Durch diese Redundanz kann die Anzahl der zur Auflösung einer Beziehung benötigten Datenbankaufrufe deutlich reduziert werden.

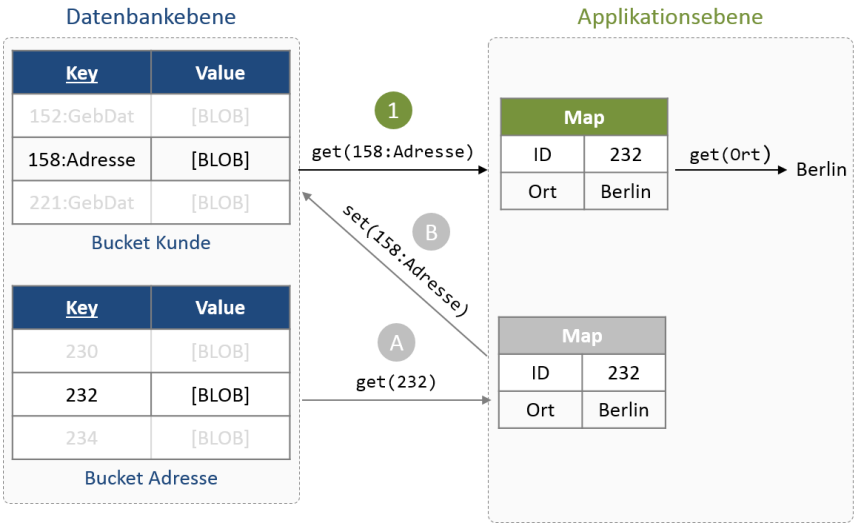


Abbildung 3.11: Denormalisierte Abbildung einer Adresse

Die denormalisierte Abbildung der Beziehung zwischen einem Kunden und seiner Adressen ist in Abbildung 3.11 dargestellt. Zur Auflösung dieser Beziehung ist nur eine Anfrage an das Bucket *Kunde* erforderlich (Schritt 1). Besonders bei mehrwertigen Beziehungen kann sich hierdurch ein großer Vorteil gegenüber der Referenzierung ergeben. Dieser Vorteil wird allerdings auf Kosten einer redundanten Datenhaltung erkauft. Da die Adressinformationen sowohl im Bucket *Kunde* als auch im Bucket *Adresse* hinterlegt werden, drohen im Fall von Adressänderungen Inkonsistenzen im Datenbestand. Um diese Inkonsistenzen zu vermeiden, müssen die redundanten Datensätze in bestimmten Intervallen aktualisiert werden (Schritt A und Schritt B). Durch diese regelmäßigen Aktualisierungen entsteht allerdings erneut ein zusätzli-

cher Aufwand auf Applikationsebene. In Anbetracht dieser Nachteile sollte die Denormalisierung nur bei Entitäten eingesetzt werden, die sich entweder sehr selten ändern oder die niedrigen Konsistenzbestimmungen unterliegen.

Aggregation: Beziehungen zwischen Entitäten können auch mithilfe der Aggregation abgebildet werden. Sind an einer Beziehung schwache Entitäten oder Entitäten beteiligt, die nie direkt angesprochen werden, so können diese Entitäten in eine starke Entität eingebettet und ein sogenanntes Aggregat gebildet werden. Aggregate werden in der Regel bei 1:1- oder 1:N-Beziehungen eingesetzt. M:N-Beziehungen lassen sich mit ihnen nur abbilden, wenn die eingebettete Entität redundant gespeichert wird.

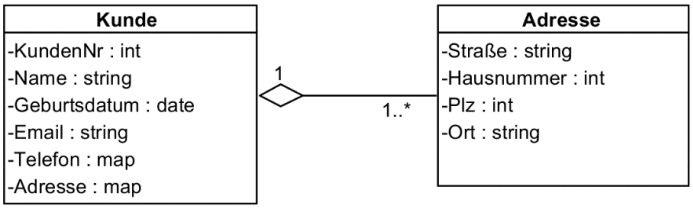


Abbildung 3.12: Konzeptuelle Darstellung der Aggregation *Kunde-Adresse*

Wird beispielsweise eine Adresse nie direkt angesprochen, sondern immer nur über den zugehörigen Kunden, kann die Entität *Adresse* in die Entität *Kunde* eingebettet werden (Abbildung 3.12). Besitzen mehrere Kunden die gleiche Adresse, müssen diese Informationen redundant in jedem Kundendatensatz gespeichert werden.

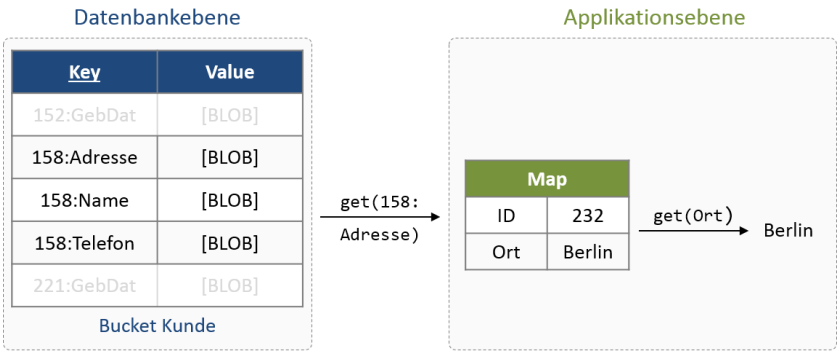


Abbildung 3.13: Logische Darstellung der Aggregation *Kunde-Adresse*

Die Verwendung eines Aggregats ist in Abbildung 3.13 dargestellt. Ähnlich wie bei der Denormalisierung werden die benötigten Adressinformationen als Attribut der Entität *Kunde* abgebildet. Eine Synchronisation der Adressdaten ist aufgrund der fehlenden Redundanz nicht erforderlich. Die Aggregation stellt demnach einen Kompromiss zwischen der Referenzierung und Denormalisierung dar. So weist sie im Vergleich zur Denormalisierung einen geringeren Speicherplatzbedarf und niedrigere Wartungs-

kosten auf, weil deutlich weniger Daten redundant gespeichert werden. Gegenüber der Referenzierung ergeben sich Performanzvorteile bei Lesezugriffen. Da zusammengehörige Daten am selben Speicherplatz verwaltet werden, können teure Beziehungsauflösungen auf Applikationsebene vermieden werden. Diese Vorteile werden jedoch zu Lasten einer schwächeren Anfragemächtigkeit erzielt. Key Value Stores ermöglichen ausschließlich einen primärschlüsselbasierten Datenzugriff und erlauben daher keinen direkten Zugriff auf die eingebetteten Entitäten. So ist beispielsweise bei dem in Abbildung 3.13 dargestellten Datenmodell eine Anfrage aller Orte, an denen Kunden wohnen, nicht möglich. Aus diesem Grund sind eine Identifizierung von starken Entitäten sowie eine gründliche Analyse der Zugriffsmuster absolut notwendig, um Beziehungen effizient mit Aggregationen abbilden zu können.

Modellierungsprozess

Entitäten und Beziehungen eines konzeptuellen Datenmodells können unter Zuhilfenahme der in diesem Kapitel vorgestellten Abbildungsregeln auf das logische Datenmodell eines Key Value Stores abgebildet werden. Die entsprechenden Pendants zwischen diesen beiden Modellen sind in Tabelle 3.3 dargestellt.

ER-Modell	Key Value Stores
Entitätstyp	Bucket
Entität	Schlüssel-Wert-Paar/e
Einfaches Attribut	Attribut / Schlüssel-Wert-Paar
Zusammengesetztes Attribut	Map
Mehrwertiges Attribut	Array, Set, List
1:1-Beziehung	Referenzierung, Denormalisierung, Aggregation
1:N-Beziehung	
M:N-Beziehung	

Tabelle 3.3: Übereinstimmungen zwischen dem ER-Modell und Key Value Stores

Bei der Verwendung dieser Modellierungsmuster muss jedoch zwingend der Mehraufwand auf Applikationsebene berücksichtigt werden, der sich durch das minimalistische Datenmodell und den geringen Funktionsumfang von Key Value Stores ergibt. Je mehr Konstrukte und Funktionalitäten benötigt werden, die über das einfache Lesen und Schreiben einzelner Schlüssel-Wert-Paare hinausgehen, desto schneller steigen die Komplexität und infolgedessen auch die Fehleranfälligkeit und die Entwicklungskosten auf Applikationsebene an. Zudem relativieren sich die erhofften Performanzvorteile gegenüber relationalen Lösungen, wenn fragmentierte Entitäten, Fremdschlüsselbeziehungen oder sekundäre Zugriffspfade applikationsseitig durch eine Vielzahl zusätzlicher Anfragen aufgelöst werden müssen.

In Anbetracht dieser Eigenschaften sind Key Value Stores nicht zur vollständigen Abbildung mehrerer Entitäten und Beziehungen umfassender Miniwelten geeignet. Vielmehr steht hier die performante Bereitstellung einer Entität oder eines minimalen Aggregats im Vordergrund, das aus einer geringen Anzahl an Schlüssel-Wert-Paaren besteht. Diese starke Ausrichtung auf bestimmte Anfragen ist nicht nur der fundamentale Unterschied zu relationalen Datenbanken, sie bestimmt auch sehr stark die Datenmodellierung mit Key Value Stores. Der an dieser Stelle vorgestellte fünfstufige Modellierungsprozess basiert deswegen auf einer Identifikation der zu unterstützenden Anfragen und Ergebnismengen:

1. Bestimmung von Anfragen und Ergebnismengen,
2. Modellierung von Aggregationen,
3. Abbildung starker Entitäten,
4. Abbildung schwacher Entitäten,
5. Abbildung von Beziehungen.

Bestimmung von Anfragen und Ergebnismengen: Die Grundlage der Datenmodellierung mit Key Value Stores bildet die Identifikation der Anfragen eines Anwendungsfalls, auf die das Datenmodell eines Key Value Stores ausgelegt werden soll. Für jede dieser Anfragen gilt es, auf konzeptueller Ebene die zur Bereitstellung der Ergebnismenge benötigten Attribute, Entitäten und Beziehungen zu bestimmen und somit Teilmengen der zu modellierenden Miniwelt zu erzeugen.

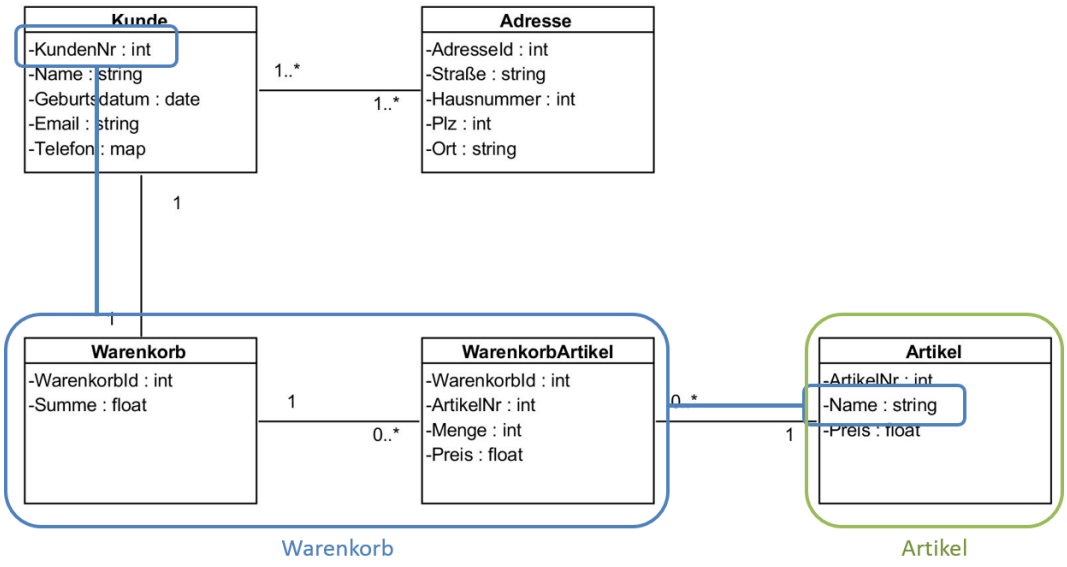


Abbildung 3.14: Identifikation von Teilmengen auf konzeptueller Ebene

In Abbildung 3.14 sind zwei mögliche Teilmengen im konzeptuellen Datenmodell eines Onlineshops hervorgehoben. In der blau gekennzeichneten Teilmenge *Warenkorb* soll über eine Kundennummer der zugehörige Warenkorb mit den Namen aller enthaltenen Artikel bereitgestellt werden. In der grün markierten Teilmenge *Artikel* können alle Artikel über ihre Artikelnummer ausgelesen werden. Das Attribut *Name* in der Entität *Artikel* befindet sich in beiden Teilmengen, weshalb diese nicht disjunkt sind.

Modellierung von Aggregationen: Das konzeptuelle Datenmodell wird so angepasst, dass jede dieser Teilmengen durch eine Aggregation repräsentiert wird. Hierzu wird in einem ersten Schritt für jede Teilmenge eine starke Entität identifiziert, welche die Basis der jeweiligen Aggregation bildet. In einem zweiten Schritt werden die übrigen Entitäten als schwache Entitäten der Aggregation abgebildet. Anschließend gilt es, einen geeigneten Primärschlüssel für die Aggregation zu identifizieren. Ein geeigneter Schlüsselkandidat ist der Primärschlüssel der starken Entität. Da sowohl die Performanz als auch die Mächtigkeit aller zu unterstützenden Anfragen vom Primärschlüssel abhängen, kann es abhängig von der Anfrage notwendig sein, den Primärschlüssel um weitere Attribute zu ergänzen. In einem abschließenden Modellierungsschritt werden die Entitäten und Attribute abgebildet, welche in mehreren Teilmengen vorkommen. Hierbei muss abhängig vom Anwendungsfall entschieden werden, ob die betroffenen Sachverhalte redundant oder als eigenständige, nicht aggregierte Entitäten modelliert werden müssen.

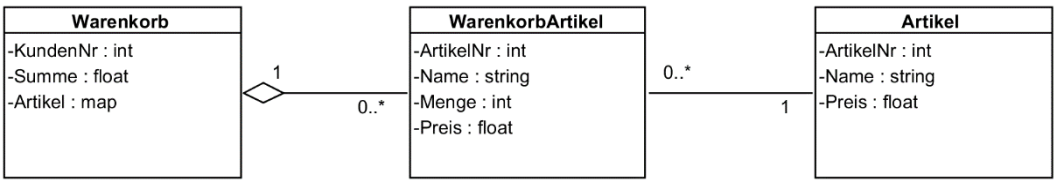


Abbildung 3.15: Modellierung der identifizierten Teilmengen als Aggregationen

In Abbildung 3.15 ist das entsprechend angepasste Datenmodell des Onlineshops abgebildet. Die Teilmenge *Warenkorb* wird durch die starke Entität *Warenkorb* mit dem Primärschlüssel *KundenNr* und der von ihr abhängigen Entität *WarenkorbArtikel* repräsentiert. Die Teilmenge *Artikel* wird mittels der gleichnamigen Entität dargestellt. Der Name eines Artikels, der zur Bildung beider Teilmengen benötigt wird, ist redundant in den beiden Entitäten *WarenkorbArtikel* und *Artikel* abgebildet. Dadurch wird eine Auflösung von Fremdschlüsselbeziehungen auf Applikationsebene vermieden.

Abbildung starker Entitäten: Sobald das konzeptuelle Datenmodell angepasst wurde, können alle starken Entitäten gemäß der vorgestellten Regeln auf das logische Datenmodell abgebildet werden. In diesem Beispiel werden daher die beiden Entitäten *Warenkorb* und *Artikel* durch jeweils ein Bucket repräsentiert.

Abbildung schwacher Entitäten: Anschließend werden alle schwachen Entitäten des konzeptuellen Datenmodells in die starken Entitäten des logischen Datenmodells eingebettet.

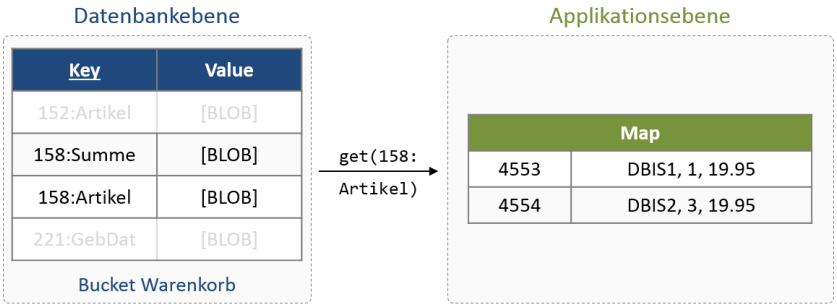


Abbildung 3.16: Logische Darstellung der aggregierten Entität *Warenkorb*

Als eine von vielen in diesem Kapitel vorgestellten Modellierungsmöglichkeiten bietet sich die in Abbildung 3.16 dargestellte Variante an. Hierbei wird die schwache Entität *WarenkorbArtikel* mithilfe des Datentyps `Map` in die Entität *Warenkorb* eingebettet wird. Dieser Datentyp lässt sich über einen zusammengesetzten Primärschlüssel direkt ansprechen und der Inhalt des Warenkorbs applikationsseitig auslesen.

Abbildung von Beziehungen: In einem finalen Schritt werden die verbleibenden Beziehungen in das logische Datenmodell überführt. Da bereits auf konzeptueller Ebene Beziehungen aggregiert und denormalisiert wurden, bleibt an dieser Stelle die Referenzierung als einziges Modellierungsmuster übrig. Angesichts des applikationsseitigen Mehraufwandes zur Auflösung von Fremdschlüsselbeziehungen, muss abhängig vom Anwendungsfall entschieden werden, ob Beziehungen bidirektional oder unidirektional abgebildet werden. In dem Beispiel des Onlineshops verbleiben nach den vorherigen Modellierungsschritten die Beziehungen zwischen den Entitäten *Warenkorb* und *Kunde* sowie zwischen *WarenkorbArtikel* und *Artikel*. Die Beziehung zwischen der Entität *Warenkorb* und *Kunde* kann vernachlässigt werden, da diese wegen des identischen Primärschlüssels implizit abgebildet ist. Die Abbildung der 1:N-Beziehung zwischen *WarenkorbArtikel* und *Artikel* sollte unidirektional auf der Seite von *WarenkorbArtikel* abgebildet werden, da die Anfrage aller Warenkörbe, in der ein bestimmter Artikel liegt, in diesem Anwendungsfall vermutlich nicht ausgeführt werden wird.

Unter Zuhilfenahme der in diesem Kapitel entwickelten Abbildungsregeln und des vorgestellten Modellierungsprozesses können konzeptuelle Datenmodelle vollständig auf das logische Datenmodell von Key Value Stores übertragen werden. So ließen sich beispielsweise auch die beiden Entitäten *Kunde* und *Adresse* auf Schlüssel-Wert-Paare abbilden. Allerdings ist diese vollständige Abbildung größerer Miniwelten auf das minimalistische Datenmodell von Key Value Stores wegen der geringen Anfragemächtig-

keit und des hohen Mehraufwands auf Applikationsebene nicht sinnvoll. Die Abbildung des konzeptuellen Datenmodells auf das logische Datenmodell von Key Value Stores wird vor allem von den Anfragen bestimmt. Da sich dieser Sachverhalt nicht im konzeptuellen Datenmodell abbilden lässt, ist eine automatisierte Transformation des konzeptuellen Datenmodells mithilfe von CASE-Werkzeugen ohne Benutzerunterstützung nicht möglich.

3.3 Interne Ebene

Key Value Stores sind für die hochperformante Verarbeitung von primitiven Lese- und Schreibanfragen konzipiert. Wie sich bereits im vorherigen Kapitel bei der Datenmodellierung andeutete, verzichten Key Value Stores für dieses Ziel auf jeglichen Funktionsumfang, der bei der Verarbeitung von Lese- und Schreibanfragen hinderlich sein kann. Key Value Stores unterscheiden sich darum nicht nur auf logischer, sondern auch auf interner Ebene von relationalen Datenbanken. Die konzeptuelle Aufarbeitung dieser Unterschiede ist Gegenstand dieses Kapitels.

3.3.1 Speicherstrukturen

Aufgrund der höheren Verarbeitungsgeschwindigkeit bevorzugen die meisten Key Value Stores den Arbeitsspeicher gegenüber Festplatten als primäres Speichermedium. Dank des Preisverfalls in diesem Segment können inzwischen auch größere Datenmengen vollständig im Hauptspeicher hinterlegt werden. Festplatten werden überwiegend zur persistenten Speicherung von Sicherungskopien sowie zur Auslagerung von selten benötigten oder sehr großen Datenobjekten verwendet. Der Key Value Store Project Voldemort erlaubt die Einbindung zusätzlicher Speichermedien. Anstelle des Hauptspeichers können dann Schlüssel-Wert-Paare unter anderem auch in MySQL gespeichert werden [145].

Key Value Stores verwenden Hash-Funktionen als primäre Zugriffspfade [45, 145]. Demzufolge weisen diese Systeme bei direkten Suchanfragen sowie simplen Einfüge- und Löschoperationen Zugriffskosten von *O(1)* auf. Neben der hohen Geschwindigkeit ergibt sich hierbei der Vorteil, dass die Zugriffszeiten unabhängig von der Anzahl der hinterlegten Datensätze sind. Key Value Stores besitzen dafür jedoch bei Bereichsanfragen Zugriffskosten von *O(N)* und sind deswegen für solche Anfragen nicht geeignet. Wie bereits in Kapitel 3.2 erwähnt, unterstützen Key Value Stores keine sekundären Zugriffspfade.

3.3.2 Transaktionen und konkurrierende Zugriffe

Transaktionen ermöglichen die sichere und konsistente Ausführung mehrerer Datenbankoperationen trotz Mehrbenutzerzugriffen und eventueller Fehlersituationen. Zur Sicherung der hierfür notwendigen ACID-Eigenschaften ist die Transaktionsverwaltung einer Datenbank zuständig. Deren Kernaufgaben sind die Wiederherstellung eines konsistenten Datenbankzustandes im Fehlerfall (Logging und Recovery) und die Vermeidung von Mehrbenutzeranomalien (Synchronisation).

Wie bereits in Kapitel 2 erläutert, kann eine strikte Umsetzung der ACID-Eigenschaften die Verarbeitungsgeschwindigkeit von Transaktionen deutlich verlangsamen. Da Key Value Stores auf eine höchstmögliche Performanz ausgelegt sind, verzichten diese Systeme nahezu vollständig auf ein ACID-kompatibles Transaktionskonzept.

Recovery

Key Value Stores unterstützen keine Transaktionen. Es kann deshalb immer nur eine Operation auf einem Schlüssel-Wert-Paar ausgeführt werden. Tritt während der Ausführung mehrere zusammenhängender Operationen ein Fehler auf, müssen sämtliche bereits durchgeführten Operationen manuell zurückgesetzt werden. Der dazu benötigte Aufwand kann vor allem bei der fragmentierten Abbildung, bei der eine Entität auf mehrere Schlüssel-Wert-Paare aufgeteilt wird, sehr hoch ausfallen.

Key Value Stores weisen durch die Datenhaltung im Arbeitsspeicher eine geringe Datensicherheit auf. Im Fall eines Stromausfalls oder eines Systemfehlers sind sämtliche Daten im Hauptspeicher verloren. Um diesen Extremfall zu vermeiden, bieten Key Value Stores eine Reihe von Konfigurationsmöglichkeiten, mit denen anwendungsfallspezifische Kompromisse zwischen maximaler Verarbeitungsgeschwindigkeit und ausreichend persistenter Datenhaltung erzielt werden können. So können bei manchen Datenbanken persistente Speichersysteme wie Berkeley Database (DBD) oder MySQL eingebunden werden [145]. Die Sicherung der Hauptspeicherdaten erfolgt dabei mithilfe von Schnappschüssen, welche nach einer definierbaren Anzahl von Operationen oder einem Zeitintervall ausgeführt werden können [93, 151]. Manche Key Value Stores unterstützen die optionale Verwendung einer persistenten Log-Datei, welche alle Operationen zwischen den einzelnen Schnappschüssen speichert (WAL-Prinzip – siehe Kapitel 2.3.2).

Synchronisation

Key Value Stores verzichten vollständig auf den Einsatz von Synchronisationsmechanismen. Alle Operationen werden sequenziell ausgeführt. Hierdurch ergeben sich jedoch Konsistenzprobleme bei einfachen Änderungsoperationen.

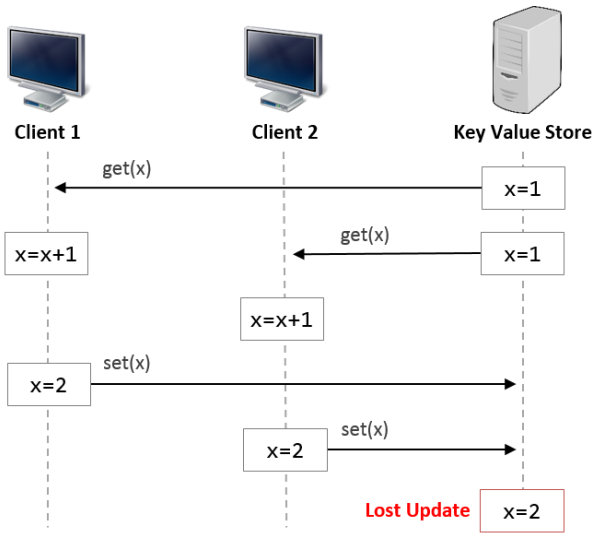


Abbildung 3.17: Lost Update durch zeitgleiche Aktualisierung

In Abbildung 3.17 ist die gleichzeitige Erhöhung eines Zählers *x* durch zwei Clients beschrieben. Client *A* liest den Wert *x = 1* aus der Datenbank aus, erhöht *x* um den Wert *1* und schreibt *x = 2* in die Datenbank zurück. In der Zwischenzeit wurde der Wert *x = 1* von Client *B* gelesen und ebenfalls um den Wert *1* erhöht. Da beide Clients das Ergebnis *x = 2* in die Datenbank zurückschreiben, geht eine Aktualisierung verloren (Lost Update, siehe Kapitel 2.3.2). Anstatt den Wert *3* besitzt *x* am Ende der beiden Erhöhungen den Wert *2*. Solche Inkonsistenzen, die aufgrund des fehlenden Transaktionskonzepts begünstigt werden, können mit Unterstützung von „Check-and-Set"-Funktionen (CAS [151, S.11]) vermieden werden. Hierbei wird vor einer Schreiboperation zunächst geprüft, ob das entsprechende Datenobjekt in der Zwischenzeit verändert wurde. Ist das der Fall, wird die Schreiboperation seitens des Key Value Stores zurückgewiesen und erneut ausgeführt (Abbildung 3.18).

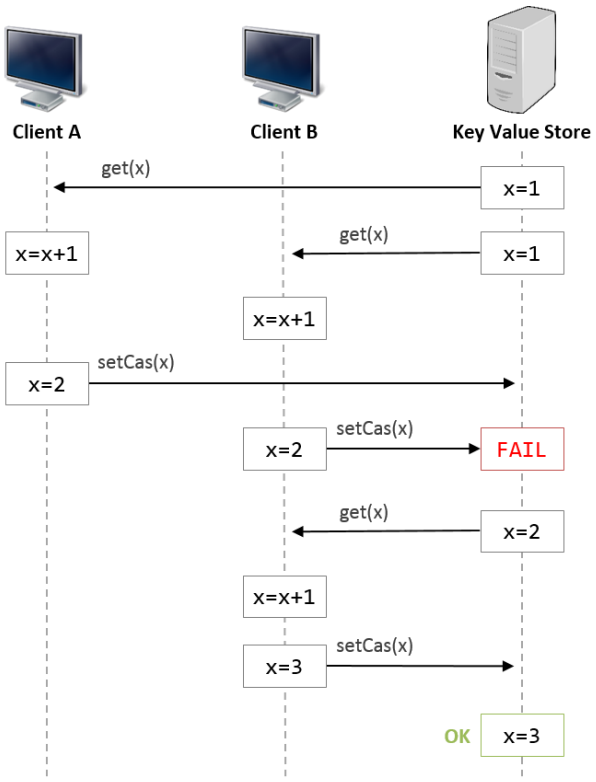


Abbildung 3.18: Zeitgleiche Aktualisierung mit Check-and-Set-Funktionen

CAS-Funktionen sind wegen ihrer hohen Rücksetzungsquote bei schreibintensiven Anwendungsfällen nicht geeignet. Für solche spezifischen Fälle bieten einige Key Value Stores, wie beispielsweise Redis, spezielle Operationen an, mit denen sich Werte direkt auf Datenbankseite inkrementieren lassen. Diese Operationen vermeiden sowohl Inkonsistenzen infolge paralleler Schreiboperationen als auch erhöhte Ausführungskosten, die durch CAS-Funktionen verursacht werden.

Neben Mechanismen zur Isolation von konkurrierenden Zugriffen fehlt es Key Value Stores auch an Techniken zur Sicherung der Konsistenz. Da die Werte der Schlüssel-Wert-Paare nicht von diesen Systemen interpretiert werden können, ist auch eine Überprüfung von Integritätsbedingungen nicht möglich. Aus diesem Grund muss auch die Sicherung der Konsistenz applikationsseitig realisiert werden.

3.4 Verteilte Datenhaltung

Die Veröffentlichung des hochverfügbaren und linear horizontal skalierbaren Key Value Stores Dynamo sorgte im Jahr 2007 nicht zuletzt wegen seines Verteilungskonzepts für großes Aufsehen. Dynamo ermöglicht bei einer vollständig dezentralen Datenhaltung unter minimalem administrativen Aufwand das dynamische Hinzufügen und Entfernen von zusätzlichen Verarbeitungsrechnern [45]. Die hier eingesetzten Techniken wurden anschließend in den Key Value Stores Project Voldemort und Riak übernommen. Der Key Value Store Redis unterstützt keine verteilte Datenhaltung.

3.4.1 Replikation

Die hier beschriebenen Key Value Stores verwenden ein quorumbasiertes Master-Slave-Replikationsverfahren, bei dem eine definierbare Anzahl an Replikaten eine Lese- oder Schreiboperation bestätigen muss. Dadurch kann trotz temporärer Inkonsistenzen einiger Knoten stets eine globale Konsistenz innerhalb des verteilten Systems erzielt werden. Wird bei einem auf n Knoten replizierten System strikte Konsistenz verlangt, so müssen bei Leseoperationen mindestens r Knoten mit demselben Ergebnis antworten und bei Schreiboperationen mindestens w Knoten die Änderung bestätigen, sodass gilt: $r + w > n$. Dynamo verwendet standardmäßig für die Parameter (n, r, w) die Einstellung (*3, 2, 2*) [45, S.215]. Durch die Wahl dieser Parameter kann abhängig vom Anwendungsfall relativ einfach ein Kompromiss zwischen der Bereitstellung von performanten Leseoperationen (*3, 1, 3*) oder Schreiboperationen (*3, 3, 1*) erzielt werden.

Das von Dynamo eingesetzte Sloppy-Quorum-Verfahren unterscheidet sich von klassischen Quorumverfahren dahingehend, dass der Koordinator einer Anfrage zur erfolgreichen Berechnung des Quorums die Servermenge n temporär um weitere Server (Stellvertreterknoten) erweitern kann. Dieses Verfahren ist insbesondere dann von Vorteil, wenn einer der ursprünglichen Knoten der Servermenge n kurzfristig nicht verfügbar ist. Falls ein Koordinator eine Anfrage an einen Stellvertreterknoten übergibt (engl. handoff), speichert er bei sich lokal einen Hinweis (engl. hint), dass der ursprünglich zuständige Knoten aktualisiert werden muss, sobald er wieder erreichbar ist. Der Hinweis besteht aus der Adresse des Knotens, dem Schlüssel des Datensatzes und den zu schreibenden Daten. Ist der ursprünglich zuständige Knoten wieder verfügbar, werden die verpassten Operationen nachträglich mithilfe dieser Informationen durchgeführt. Dieses Verfahren wird als Hinted Handoff bezeichnet [45, S.212].

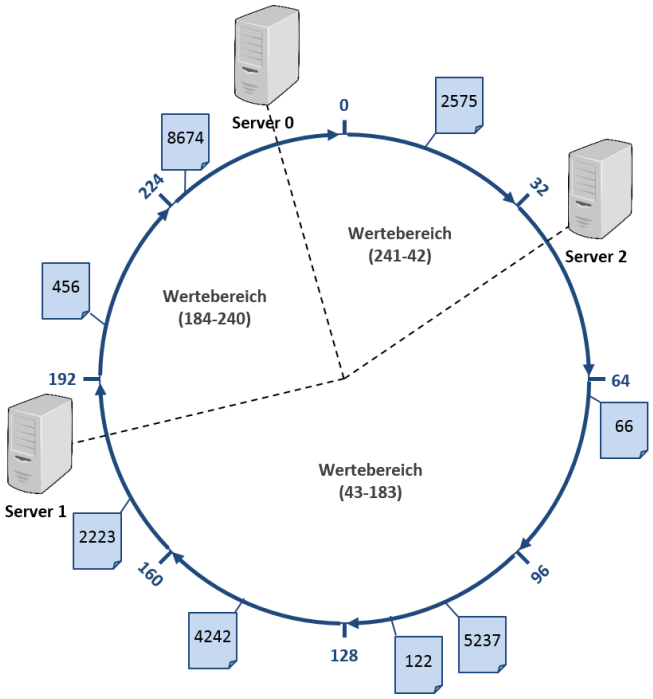
3.4.2 Fragmentierung und Allokation

Schlüssel-Wert-Paare sind aufgrund ihrer minimalen Datenstruktur hervorragend für die feingranulare Verteilung auf mehrere Verarbeitungsrechner geeignet. Eine weitere Unterteilung im Rahmen einer horizontalen oder vertikalen Fragmentierung ist daher nicht notwendig.

In Kapitel 2.4.2 wurden die Vor- und Nachteile von statischen und dynamischen Allokationsverfahren erläutert. Statische Allokationsverfahren ermöglichen eine sehr schnelle und einfache Datenverteilung. Da für die Allokation nicht zwingend ein Koordinator-Knoten benötigt wird, hängt die Verfügbarkeit des Gesamtsystems nicht von der Verfügbarkeit eines einzelnen Knotens ab. Die Nachteile der statischen gegenüber der dynamischen Allokation liegen in der deutlich schlechteren Lastverteilung sowie in dem sehr hohen Reallokationsaufwand, der beim Hinzufügen oder Entfernen von Knoten entstehen kann.

Consistent Hashing

Die sehr hohen Verfügbarkeitsbestimmungen der Amazon Dienste stellten die Hauptanforderung bei der Entwicklung des Key Value Stores Dynamo dar. Damit das verteilte Datenbanksystem nicht von der Verfügbarkeit eines zentralen Verwaltungsdienstes abhängig ist, basiert die Verteilung von Dynamo auf einem statischen Verteilungsverfahren namens Consistent Hashing. Consistent Hashing [92] unterscheidet sich dahingehend von der einfachen Verteilung mittels einer Hashfunktion, dass ein Knoten nicht für einen einzelnen Hashwert, sondern für einen oder mehrere Hashwertebereiche zuständig ist. Deswegen wirken sich die oben genannten Nachteile der statischen Verteilung weniger gravierend auf das verteilte Datenbanksystem aus. Neben Dynamo verwenden die Key Value Stores Project Voldemort und Riak sowie die später vorgestellten NoSQL-Datenbanken CouchDB und Cassandra Consistent Hashing.



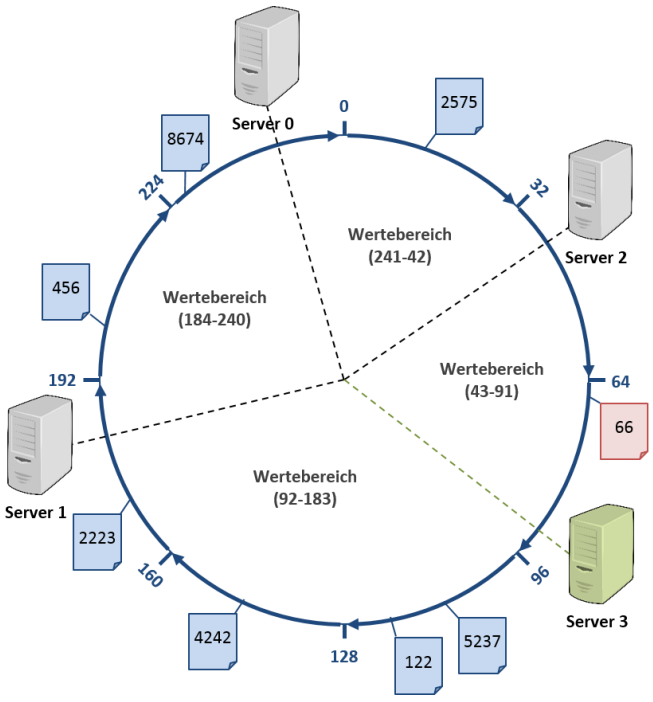
key	h(key)
Server 0	240
Server 1	183
Server 2	42
5237	117
2223	175
66	66
4242	146
8674	226
2575	15
456	200
122	122

h(key) = key mod 256

Abbildung 3.19: Allokation mithilfe von Consistent Hashing

Zur besseren Veranschaulichung des Konzepts wird der Wertebereich einer Hash-Funktion als Ring dargestellt. In Abbildung 3.19 umfasst dieser Wertebereich die Zahlen 0 bis 255. Unter Zuhilfenahme der Hash-Funktion werden alle Server des verteilten Systems auf dem Ring platziert. Als Schlüssel für die Berechnung der Position eines Servers bietet sich unter anderem dessen IP-Adresse an. Zur Bestimmung der Speicheradresse eines Datenobjekts wird mithilfe der Hash-Funktion und dem Schlüssel des Datenobjekts dessen Position auf dem Ring berechnet. Der nächste Server in Uhrzeigerrichtung ist für die Verarbeitung des Schlüssel-Wert-Paares zuständig. Ist ein Replikationsfaktor von $n = 3$ definiert, wird das Key-Value-Paar ausgehend von dessen Ringposition automatisch auf den nächsten drei Servern gespeichert.

Der Vorteil von Consistent Hashing im Vergleich zur klassischen Hashverteilung liegt in den wesentlich günstigeren Reallokationskosten nach dem Hinzufügen und Entfernen von Knoten. In diesen Fällen müssen ausschließlich die Datenobjekte der betroffenen Wertebereiche zum Teil reorganisiert werden. Bei einer Anzahl von s Servern beträgt die Reorganisation – Gleichverteilung angenommen – nur noch $\frac{1}{s+1}$ Datenobjekte [145]. War bei der in Abbildung 2.17 dargestellten einfachen Hash-Verteilung die Reallokation von fünf Datenobjekten notwendig, muss bei der in Abbildung 3.20 dargestellten Variante des Consistent Hashing nur ein Datensatz (rot) auf einen neuen Server verschoben werden.



key	h(key)
Server 0	240
Server 1	183
Server 2	42
5237	117
2223	175
66	66
4242	146
8674	226
2575	15
456	200
122	122
Server 3	91

h(key) = key mod 256

Abbildung 3.20: Geringer Reallokationsaufwand beim Hinzufügen eines Servers

Die bisher vorgestellte Verteilung mithilfe des Consistent Hashings führt zu einer ähnlich schlechten Lastbalancierung wie bei normalen Hashverfahren. So ist beispielsweise in Abbildung 3.20 Server *1* für den mit Abstand größten Wertebereich zuständig. Dynamo-basierte Systeme verwenden darum eine Variante des Consistent Hashings, bei der einem physikalischen Server mittels „virtueller Knoten" mehrere Wertebereiche auf dem Ring zugeteilt werden können. Hierdurch wird eine wesentlich bessere Lastbalancierung erreicht. Zusätzlich ermöglicht dieses Konzept durch die Anzahl der einem Server zugewiesenen Wertebereiche eine effiziente Ausnutzung dessen individueller Leistungsfähigkeit. Handelt es sich bei dem in Abbildung 3.20 hinzugefügten Server 3 um den leistungsstärksten Rechner, kann dieser im Vergleich zu den anderen Servern über eine höhere Anzahl von virtuellen Knoten verfügen.

Aufgrund des dezentralen Verteilungskonzepts sind bei Dynamo-basierten Systemen alle Knoten mit Verwaltungsaufgaben, wie Routing und Fehlerbehandlung, vertraut. Die notwendigen Informationen über den gegenwärtigen Zustand des verteilten Systems werden unter Zuhilfenahme des Peer-to-Peer-Kommunikationsprotokolls Gossip erlangt. Jeder Knoten tauscht hierbei versionierte Statusinformationen über sich und andere Knoten im Cluster aus und ist daher jederzeit über die Topologie des verteilten Systems informiert. Wird ein neuer Rechner einem Cluster hinzugefügt, muss dieser deshalb nur einem einzigen Knoten bekanntgemacht werden.

3.4.3 Verteilte Anfrageverarbeitung

Durch die dezentrale Verwaltungsarchitektur von Dynamo sind alle Knoten in der Lage, Lese- und Schreibanfragen entgegenzunehmen und zu beantworten. Stellt ein Client eine Anfrage an einen Knoten, fungiert dieser für den Zeitraum der Anfrage als Koordinator. Der Koordinator ermittelt die für die Anfrageverarbeitung zuständigen Knoten und führt entsprechend der vom Client übermittelten Konsistenzwerte (n, r, w) die gewünschten Operationen auf diesen Knoten aus.

Abbildung 3.21 beschreibt die Verarbeitung einer Leseanfrage an den Datensatz mit dem Primärschlüssel *2575*. In diesem Beispiel dient Server *5* als Koordinator, als Konsistenzparameter wurden $n = 3$, $w = 2$ und $r = 2$ gewählt. Der Koordinator leitet die Leseanfrage an die für diesen Datensatz zuständigen Server *4*, *2* und *7* weiter. Daraufhin vergleicht er die ersten r erhaltenen Ergebnisse auf Inkonsistenzen und übermittelt schließlich den korrekten Wert an den Client. Im Fall von widersprüchlichen Ergebnissen versucht der Koordinator diese Konflikte zunächst selbst zu beheben (Read Repair). Ist er dazu nicht in der Lage, wird eine Liste mit den verschiedenen Versionen zusammen mit weiteren Informationen zur Konfliktauflösung an den Client übertragen und die Rückübermittlung des korrekten Ergebnisses vom Client erwartet. Schreiboperationen werden ebenfalls an alle zuständigen Knoten weitergeleitet. Sobald die ersten w Knoten die Ausführung der Operationen bestätigt haben, wird eine Bestätigung an den Client gesendet.

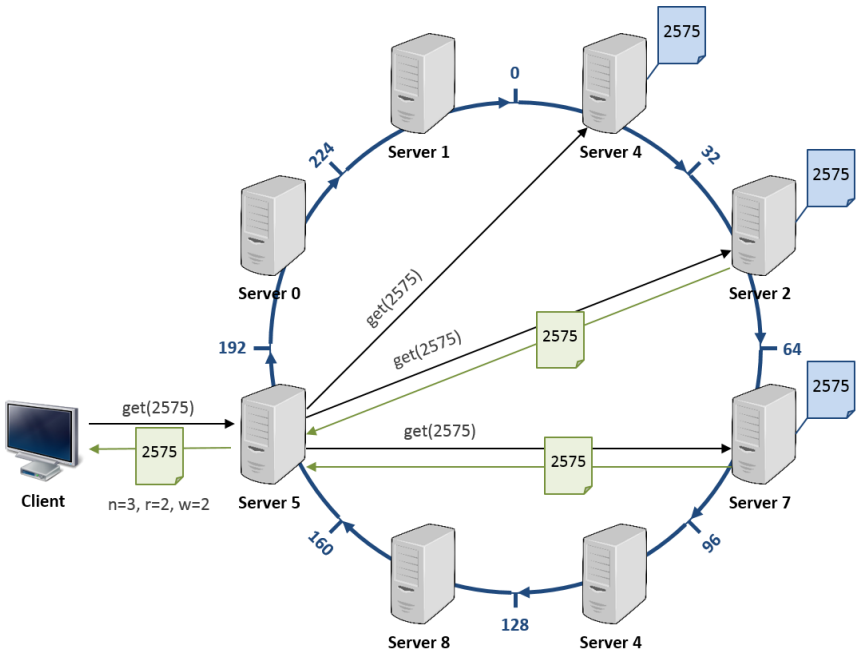


Abbildung 3.21: Verarbeitung von verteilten Leseanfragen

Die vollständige Verwaltung der Anfrageverarbeitung mittels eines Koordinators ermöglicht auf Clientseite eine sehr komfortable und vom verteilten Datenbanksystem unabhängige Interaktion. Es müssen lediglich die Operationen `get` oder `set` an einen Knoten übertragen werden. Der Nachteil liegt jedoch im höheren Verwaltungsaufwand auf Datenbankseite, der vor allem bei hochperformanten Anfragen einen unerwünschten Overhead verursacht. Aus diesem Grund unterstützt Dynamo auch eine Anfragekoordination auf Seiten der Clients. Hierzu werden spezielle Bibliotheken bereitgestellt, welche in definierten Intervallen über die gegenwärtigen Zuständigkeiten innerhalb des Clusters informiert werden. Dank des direkten Sendens der Anfragen an die zuständigen Server können deutliche Performanzgewinne erzielt werden. Nach Angaben von Amazon werden vom Client koordinierte Anfragen in etwa doppelt so schnell verarbeitet wie serverseitig koordinierte (siehe Tabelle 3.4).

Koordination	Lesegeschwindigkeit (Ø)	Schreibgeschwindigkeit (Ø)
Serverseitig	3,9 ms	4,0 ms
Clientseitig	1,55 ms	1,9 ms

Tabelle 3.4: Performanz bei server- und clientseitiger Koordination [45, S.218]

3.4.4 Verteilte Transaktionsverwaltung

Die atomare Ausführung mehrerer zusammenhängender Operationen im Rahmen einer Transaktion wird von Key Value Stores nicht unterstützt. Ist das Einfügen eines neuen Kunden für einen Onlineshop notwendig, müssen die parallelen Einträge in die Buckets *Benutzer*, *Anschrift* und *Warenkorb* applikationsseitig koordiniert und im Fehlerfall gemeinsam zurückgesetzt werden. Wird ein Bucket allerdings repliziert gespeichert, müssen sämtliche Schreiboperation auf allen Replikaten durchgeführt werden. Um verteilte Änderungsoperationen gegenüber Konflikten zu schützen, die infolge von Nachrichtenverlusten, Netzwerkfehlern, Systemausfällen und konkurrierenden Zugriffen verursacht werden können, verfügen Key Value Stores über Recovery- und Synchronisationsmechanismen.

Recovery

Key Value Stores sind auf eine möglichst hohe Verarbeitungsgeschwindigkeit ausgelegt, weshalb Datensätze überwiegend im Hauptspeicher verwaltet werden und eine Sicherung auf persistente Speichermedien nur asynchron durchgeführt wird. Um Änderungen auch gegenüber Hardware- und Katastrophenfehlern zu sichern, müssen die Daten auf andere Rechner repliziert werden. Strikt konsistente Protokolle, wie beispielsweise 2PC, sind in Anbetracht ihrer sehr hohen Kommunikationskosten und ihrer Intoleranz gegenüber kurzzeitigen Fehlern nicht für Key Value Stores geeignet. Längere Verzögerungen oder sogar ein Zurückweisen von Schreiboperationen bei temporären Netzwerk- oder Rechnerfehlern sind in der Regel nicht tolerierbar. Um möglichst hohe Schreibraten zu ermöglichen, werden deswegen Consensus-Protokolle mit einem niedrigen Schreib-Konsistenzparameter bevorzugt. Das hat zur Folge, dass Schreiboperationen gegenüber dem Client bestätigt werden, obwohl die Änderungen noch nicht auf allen Knoten ausgeführt wurden. Durch die asynchrone Verarbeitung der Schreiboperationen können bei anschließenden Leseanfragen inkonsistente Versionen eines Datenobjekts beim Koordinator vorliegen.

Synchronisation

Konflikte können aufgrund von Systemfehlern oder konkurrierenden Schreiboperationen entstehen. Für die Erkennung und Auflösung dieser Fehler ist der für die Ausführung einer Operation verantwortliche Knoten (Koordinator) zuständig. Damit ein Koordinator bei Leseanfragen Konflikte zwischen verschiedenen Versionen eines Datenobjekts erkennen kann, verwenden Key Value Stores das Konzept der sogenannte Vektoruhren (engl. Vector Clocks). Vektoruhren sind eine Weiterentwicklung von Lamport-Uhren [100] und erlauben die eindeutige Identifizierung von nebenläufigen Ereignissen.

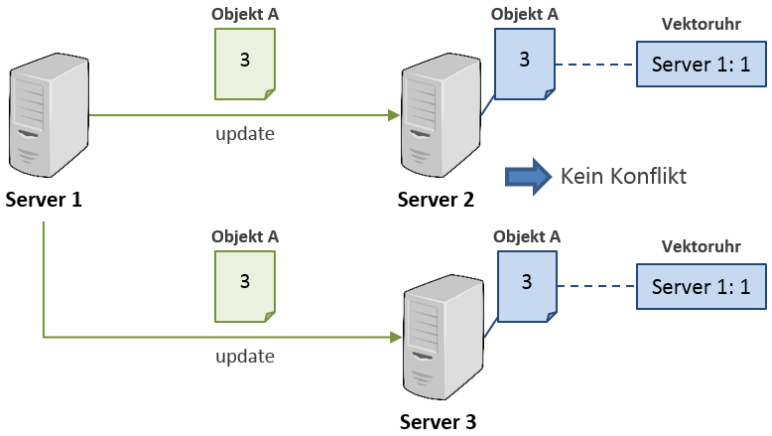


Abbildung 3.22: Erfolgreiche Aktualisierung eines Datenobjekts auf zwei Servern

Jedes Datenobjekt in einem Key Value Store verfügt über einen Zähler (Uhr) für jeden Knoten, der auf dieses Datenobjekt zugreift. Dieser Zähler wird bei jeder Lese- und Schreiboperation eines Knotens inkrementiert. In Abbildung 3.22 wird ein auf den Servern *2* und *3* befindliches Datenobjekt *A* von Server *1* aktualisiert. Daraufhin besitzen sowohl Server *2* und Server *3* identische Vektoruhren bezüglich des Datenobjekts *A*.

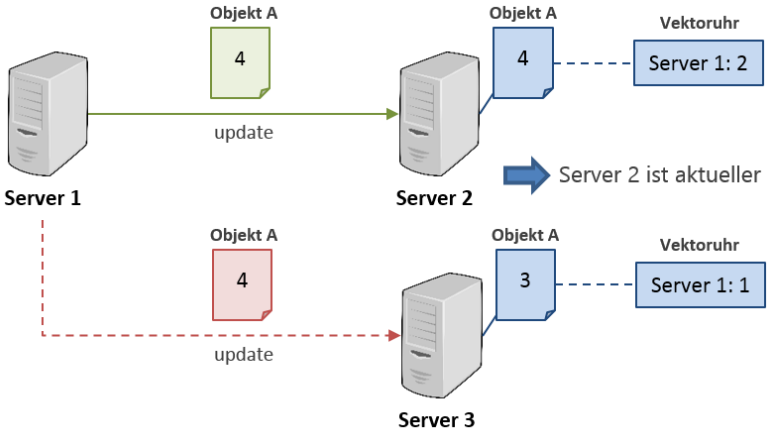


Abbildung 3.23: Fehlgeschlagene Aktualisierung eines Datenobjekts auf zwei Servern

Liegen zwei Versionen eines Datenobjekts A_1 und A_2 vor, lässt sich anhand der jeweiligen Zähler eindeutig feststellen, in welcher Reihenfolge die beiden Datenobjekte bearbeitet wurden. Sind alle Zähler von A_2 größer oder gleich der Zähler von A_1, ist A_2 der legitime Nachfolger von A_1. In Abbildung 3.23 schlug die Aktualisierung des Servers *3* fehl. Server *3* besitzt infolgedessen für Datenobjekt *A* eine kleinere Vektoruhr als Server *2* und kann darum mit dem Wert von Server *2* aktualisiert werden.

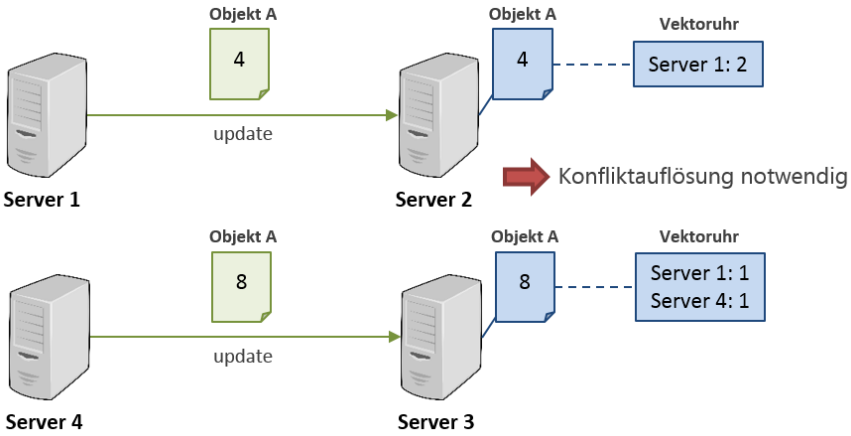


Abbildung 3.24: Konflikterkennung anhand widersprüchlicher Vektoruhren

Wird ein Datenobjekt parallel von zwei verschiedenen Servern verarbeitet, liegen anschließend zwei Versionen mit widersprüchlichen Vektoruhren vor. In Abbildung 3.24 weisen die Server *2* und 3 jeweils unterschiedlich hohe Vektoruhren für die Server *1* und *4* auf. Dank dieser widersprüchlichen Vektoruhren ist das System in der Lage, den Konflikt zu erkennen und kann diesen nun auf verschiedene Art und Weise lösen. Im einfachsten Fall wird bei der nächsten Leseoperation das Datenobjekt mit der neuesten Version an den Client als Ergebnis übermittelt und der Verlust der älteren Aktualisierung in Kauf genommen. Werden jedoch geschäftskritische Daten verarbeitet, bei denen Datenverluste nicht tolerierbar sind, müssen Konflikte durch sensiblere Verfahren aufgelöst werden. Da hierzu häufig zusätzliche Informationen und komplexe Algorithmen notwendig sind, kann die Konfliktauflösung in der Regel nicht datenbankseitig erfolgen. Demzufolge kann der Koordinator je nach Konfiguration die verschiedenen Objektversionen an den Client übergeben und die Konfliktauflösung auf Applikationsebene durchführen lassen.

Ein Beispiel für eine clientseitige Konfliktauflösung stellt Amazons Einkaufswagen dar. Ziel des Services ist es, eine maximale Anzahl von Artikeln in den Einkaufswagen eines Kunden zu platzieren. Ein Zurücksetzen von Schreiboperationen infolge eines Netzwerkfehlers entspricht daher ebenso wenig der Geschäftsphilosophie wie ein Datenverlust aufgrund serverseitiger Konfliktauflösung. Dynamo ist folglich für diesen Anwendungsfall auf minimale Schreibkonsistenz ($w = 1$) konfiguriert. Liegen bei Leseoperationen unterschiedliche Versionen des Einkaufswagens vor, werden diese Versionen auf Applikationsebene zu einer Version zusammengeführt. Hierbei werden niemals Artikel aus dem Einkaufswagen entfernt, sondern im Zweifel bereits vom Benutzer entfernte Artikel erneut angezeigt [45, S.210]. Amazon verzeichnet in nur 0,06% aller Fälle Inkonsistenzen bei Warenkörben. Entsprechend selten muss deshalb ein Read Repair durchgeführt werden. Häufigste Ursache für Konflikte sind konkurrierende Schreiboperationen [45, S.217].

3.5 Zusammenfassung

Key Value Stores sind äußerst minimalistische Systeme, die auf eine sehr hohe Verarbeitungsgeschwindigkeit ausgelegt sind. Da Key Value Stores einzelne Schlüssel-Wert-Paare als Byte-Arrays abspeichern und keine zusätzlichen Metainformationen wie Attributnamen oder Datentypen verwalten, sind diese Systeme nicht in der Lage, die in ihnen gespeicherten Daten zu interpretieren. Sie werden deswegen auch als schemafrei bezeichnet. Eine Ausnahme bildet der Key Value Store Redis, der die Datentypen `String`, `Set`, `List`, `Hash` und `Sorted Set` anbietet. Der Umfang an Anfrage- und Änderungsfunktionalitäten ist bei Key Value Stores ebenfalls auf ein Minimum reduziert und beschränkt sich im Wesentlichen auf die drei Operationen *Lesen*, *Einfügen* und *Löschen*. Da weder sekundäre Zugriffspfade noch Mengenoperationen unterstützt werden, können diese einfachen Operationen ausschließlich auf dem Schlüssel eines einzigen Schlüssel-Wert-Paares ausgeführt werden.

Der Minimalismus von Key Value Stores beschränkt sich nicht nur auf die konzeptuelle Ebene, sondern ist auch auf interner Ebene allgegenwärtig. Key Value Stores unterstützen keine Transaktionen, sodass jede Anfrage- und Änderungsoperation einzeln ausgeführt werden muss. Darüber hinaus werden aufgrund der Schemafreiheit keine Konsistenzprüfungen durchgeführt, konkurrierende Operationen nur optimistisch synchronisiert und die Dauerhaftigkeit – wenn überhaupt – nur asynchron sichergestellt. In der Regel werden die einzelnen Schlüssel-Wert-Paare permanent im Hauptspeicher verwaltet.

Dank des Minimalismus auf konzeptueller und interner Ebene weisen Key Value Stores sehr gute Geschwindigkeits- und Skalierbarkeitseigenschaften auf. Da Anfrage- und Änderungsoperationen durch den Verzicht auf ACID-Eigenschaften praktisch overheadfrei ausgeführt werden, erzielen Key Value Stores bei einfachen Lese- und Schreibanfragen in der Regel sehr hohe Durchsatzraten bei gleichzeitig sehr niedriger Latenz. Des Weiteren lassen sich Schlüssel-Wert-Paare sehr einfach und effizient auf mehrere Verarbeitungsrechner verteilen. Vor allem Systeme, die auf dem Key Value Store Dynamo basieren, verursachen bei der Verteilung einen minimalen administrativen Aufwand. Nicht nur die verteilte Verarbeitung von Anfragen wird von diesen Systemen automatisch übernommen, auch der Lastausgleich, den das Hinzufügen und Entfernen von Datenbankservern bedingt, erfolgt ohne zusätzliche Hilfestellung vollkommen selbstständig. Außerdem verfügen Dynamo-basierte Datenbanksysteme über integrierte Mechanismen zur Erkennung und Behebung von Fehlern und garantieren dadurch eine sehr hohe Verfügbarkeit des Gesamtsystems. Dynamo konnte in den ersten beiden Jahren des Einsatzes bei Amazon 99,9995% aller Anfragen in der vorgegebenen Zeit erfolgreich verarbeiten. Es trat in diesem Zeitraum laut Angaben von Amazon kein einziger Datenverlust auf [45, S.218].

Funktionalitäten		Redis	Voldemort
Ausdrucksstarkes Datenmodell	Attributnamen	✘	✘
	Datentypen	✓	✘
Anfragemächtigkeit	Relationale Operatoren	✘	✘
	Verbundoperationen	✘	✘
	Sekundäre Zugriffspfade	✘	✘
ACID-Eigenschaften	Atomarität	✘	✘
	Integritätsbedingungen	✘	✘
	Synchronisationsmechanismen	✘	✘
	Dauerhaftigkeit	✘	✘
Verteilte Datenhaltung	Replikation	✘	✓
	Verteilung	✘	✓

Tabelle 3.5: Funktionsumfang von Key Value Stores

Das Ausmaß des Minimalismus von Key Value Stores wird bei der Betrachtung von Tabelle 3.5 deutlich, in der die wichtigsten Funktionalitäten von Datenbanksystemen dargestellt sind (vgl. Tabelle 2.6). Dieser Minimalismus ist unproblematisch, solange die fehlenden Funktionalitäten in den entsprechenden Einsatzgebieten nicht benötigt werden. Sobald das jedoch der Fall ist, müssen diese Funktionalitäten auf Anwendungsebene realisiert werden. Der hierdurch entstehende Mehraufwand steigert die Komplexität der Applikation, was wiederum höhere Entwicklungs- und Wartungskosten sowie eine höhere Fehleranfälligkeit zur Folge hat. Im schlechtesten Fall wird der von Key Value Stores aus Performanzgründen vermiedene Overhead vollständig von der Datenbankebene in die Anwendungsebene verlagert, sodass letztendlich keine Geschwindigkeitsvorteile bei der Verarbeitung von Anfragen erzielt werden können.

Key Value Stores können sich in einigen Systemeigenschaften voneinander unterscheiden. So weist beispielsweise Redis ein überdurchschnittlich großes Angebot an Datenstrukturen auf, während Project Voldemort deutlich besser für eine verteilte Datenhaltung ausgelegt ist. Trotz dieser Unterschiede sind die in diesem Kapitel getätigten Aussagen über die grundlegenden Eigenschaften von Key Value Stores allgemeingültig.

4

DOCUMENT STORES

Kapitelinhalt

- Grundlagen von Document Stores
- Datenmodellierung mit JSON
- Verteilte Anfrageverarbeitung mithilfe von MapReduce

Document Stores können als Kompromiss zwischen den einfachen aber sehr schnellen Key Value Stores und den mächtigen relationalen Datenbanken angesehen werden. Diese Systeme verfügen über ein sehr flexibles Datenmodell, mit dem vor allem semistrukturierte Daten sehr effizient verarbeitet werden können. Document Stores lassen sich obendrein sehr leicht replizieren und auf mehrere Server verteilen. Zu den bekanntesten Datenbanken dieser Kategorie zählen *MongoDB*[33] und *CouchDB*[34].

Die Eigenschaften und der Umgang mit Document Stores wurden in der Literatur bisher ausschließlich anhand von Anwendungsbeispielen beschrieben. Durch die fehlende Arbeit auf konzeptueller Ebene existieren derzeit keine allgemeingültigen Aussagen über die Vor- und Nachteile der Datenmodelle und keine allgemeingültigen Abbildungsregeln, mit denen konzeptuelle Datenmodelle auf die logischen Datenmodelle von Document Stores überführt werden können. Diese dringend benötigten Beiträge werden in diesem Kapitel nachgereicht. Basierend auf einer konzeptuellen Aufarbeitung des Datenmodells und einer umfangreichen Analyse der Anfragefunktionalitäten werden allgemeingültige Modellierungsmuster erstellt und somit ein wertvoller Beitrag zum Verständnis dieser Systeme geleistet. Anschließend werden wichtige Systemeigenschaften der internen Ebene von Document Stores behandelt. Neben Speicherstrukturen und Transaktionsverwaltungen stehen hierbei vor allem Konzepte der verteilten Datenhaltung im Fokus dieses Abschnitts.

[33] http://www.mongodb.org
[34] http://couchdb.apache.org

4.1 Einführung

Document Stores verarbeiten Daten in Dokumenten. Sie werden deswegen auch als dokumentorientierte Datenbanken bezeichnet. Ein Dokument ist eine strukturierte Sammlung mehrerer Schlüssel-Wert-Paare und entspricht üblicherweise den Formaten Java Script Object Notation (JSON)[35], YAML Ain't Markup Language (YAML)[36] oder Extensible Markup Language (XML) [78, S.12]. Document Stores sind nicht mit Dokumentmanagementsystemen zu verwechseln, bei denen häufig Dokumente verwaltet werden, die mithilfe von Textverarbeitungsprogrammen erzeugt wurden.

Das Konzept der dokumentorientierten Datenhaltung ist nicht neu. Bereits seit 1984 werden alle Daten in IBMs Groupware-System Lotus Notes[37] in Form von Dokumenten gespeichert. Die wohl bekanntesten Vertreter der Familie der dokumentorientierten Datenbanken stellen jedoch XML-Datenbanken dar. Mit der zunehmenden Verbreitung des Datenaustauschformats XML häuften sich Ende der 1990er Jahre die Forderungen nach einer direkten Verarbeitung von XML-Daten. Insbesondere Anwendungsfälle, bei denen große XML-Dokumente gezielt durchsucht und Ergebnisse erneut in Form von XML-Daten aufbereitet werden mussten, motivierten die Entwicklung von nativen XML-Datenbanken. Diese Systeme verwenden XML-Dokumente als kleinste logische Einheit und zerlegen diese nicht, wie beispielsweise relationale Datenbanken mit XML-Erweiterung, intern in mehrere vertikale Tabellen. XML-Dokumente können von nativen XML-Datenbanken gegen das XML-Schema [177] oder eine Document Type Definition (DTD) [78, S.28] validiert werden. Daneben unterstützen diese Systeme die Anfragesprachen XPath [78, S.162] und XQuery [178] sowie die Transformationssprache XSLT [78, S.144].

JSON

```
{
 "Stadt": "Bayreuth",
 "Land": "Bayern",
 "Einwohner": 70000
}
```

XML

```
<Root type="object">
  <Stadt type="string">Bayreuth</Stadt>
  <Land type="string">Bayern</Land>
  <Einwohner type="number">70000</Einwohner>
</Root>
```

Abbildung 4.1: Vergleich der Lesbarkeit zwischen JSON und XML

In den letzten Jahren ist eine zunehmende Verdrängung von XML durch einfachere Formate wie JSON oder YAML zu beobachten. Der Grund hierfür liegt in der hohen Komplexität von XML. Gerade in einfachen Anwendungsfällen kann die Definition von

[35] http://www.json.org
[36] http://www.yaml.org
[37] http://www.ibm.com/lotus

Namensräumen, XML-Schemata und XSLTs einen zu großen Aufwand darstellen. JSON und YAML sind im Gegensatz zu XML keine Auszeichnungssprachen. Wie anhand des Beispiels in Abbildung 4.1 ersichtlich wird, sind diese Sprachen durch den Verzicht auf sämtliche Auszeichnungselemente (Tags) nicht nur wesentlich kompakter als XML, sondern auch deutlich einfacher für Menschen zu lesen und zu bearbeiten. Die im Rahmen dieser Arbeit vorgestellten Document Stores verwenden das Datenformat JSON.

4.2 Konzeptuelle Ebene

Document Stores verfügen nicht nur über ein mächtiges und zugleich flexibles Datenmodell, sondern ermöglichen darüber hinaus auch das gezielte Verarbeiten der Dokumente über umfangreiche Schnittstellen. Document Stores sind demzufolge in ihrer Mächtigkeit zwischen Key Value Stores und relationalen Datenbanken anzusiedeln.

4.2.1 Datenmodell

Document Stores bilden Datensätze als eigenständige Dokumente ab. Attribute und Attributwerte eines Datensatzes werden mithilfe von Schlüssel-Wert-Paaren im entsprechenden Dokument gespeichert. Dokumente werden innerhalb sogenannter Collections verwaltet und über das Schlüssel-Wert-Paar mit dem Schlüssel *_id* eindeutig identifiziert (Tabelle 4.1).

Relationale Datenbank	Document Stores
Tabelle	Collection
Tupel (Datensatz)	Document
Primärschlüssel (ID)	_id
Attribut	Key-Value-Paar

Tabelle 4.1: Gegenüberstellung der unterschiedlichen Strukturbezeichnungen

Die in dieser Arbeit vorgestellten Document Stores MongoDB und CouchDB verarbeiten Dokumente im Datenformat JSON beziehungsweise Binary JSON (BSON)[38], einer binären Darstellung von JSON. JSON ist ein schlankes, plattformunabhängiges Datenaustauschformat mit JavaScript-Syntax, das sowohl von Menschen als auch von Maschinen interpretiert werden kann.

[38] http://bsonspec.org

Datentyp	Attributwert
Number	123
String	“Test“
Boolean	True
Object	{“Key1“ : 123, “Key2“ : “Test“, …, “ “Keyn“ : 593}
Array	[1, 2, …, n]

Tabelle 4.2: Angabe von Datentypen mithilfe von Code-Konventionen

Die primäre Speichereinheit innerhalb von JSON ist ein sogenanntes Objekt, das mehrere unsortierte Schlüssel-Wert-Paare enthält. Der Wert eines Schlüssel-Wert-Paares kann verschiedenen primitiven Datentypen entsprechen (Tabelle 4.2). Bei diesen handelt es sich um `String`, `Number`, `Boolean` und `Null`. Außerdem kann ein Wert die Datentypen `Object` und `Array` annehmen, wodurch sich beliebig tiefe Verschachtelungen beziehungsweise mehrwertige Attribute abbilden lassen [91]. BSON unterstützt zusätzlich die Datentypen `Date` und `BLOB` [179, S.41]. Der Datentyp eines Attributs ergibt sich in JSON-Dokumenten durch Code-Konventionen. Besitzt ein Attribut beispielsweise den Wert `123`, wird dieser automatisch als `Number` interpretiert.

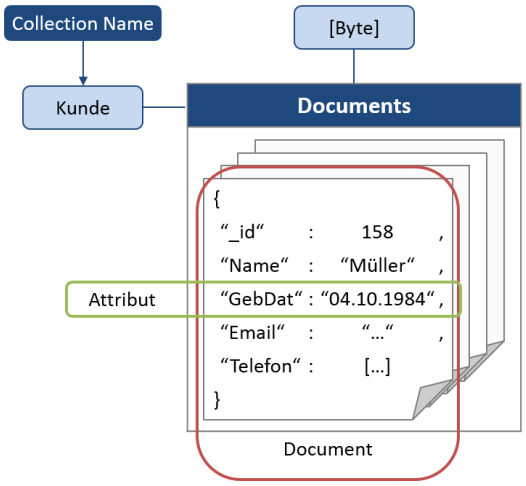


Abbildung 4.2: Logisches Datenmodell der Entität *Kunde*

Abbildung 4.2 beschreibt die Abbildung der bereits aus den vorherigen Kapiteln bekannten Entität *Kunde* auf das Datenmodell eines Document Stores. Die Collection *Kunde* enthält für jeden Datensatz ein Dokument, das über das Schlüssel-Wert-Paar *_id* eindeutig identifiziert wird. Der Wert des Primärschlüssels kann entweder vom Datenbanksystem automatisch generiert oder, wie in diesem Beispiel, manuell durch den Wert der jeweiligen Kundennummer bestimmt werden. Attribute und die zugehörigen Attributwerte werden durch Schlüssel-Wert-Paare repräsentiert.

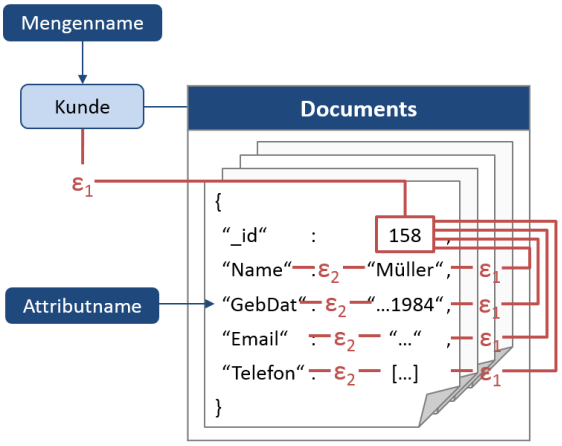


Abbildung 4.3: Prädikatorenschema des Dokuments *Kunde*

Das für Document Stores gültige Prädikatorenschema ist in Abbildung 4.3 dargestellt. Bei der Betrachtung wird deutlich, dass Document Stores außer dem Namen der Collection keine weiteren Metainformationen verwalten. Sie sind daher, ebenso wie Key Value Stores, praktisch *schemafrei*. Trotz des Fehlens eines expliziten Datenbankschemas besitzen Document Stores ein sehr ausdrucksstarkes Datenmodell, in dem sich sämtliche Prädikationen erster und zweiter Ordnung abbilden lassen. Der Grund hierfür liegt darin, dass diese Informationen vollständig in den Dokumenten enthalten sind. Dokumente speichern neben den Werten auch die Namen und die Datentypen der jeweiligen Attribute und werden deshalb auch als *selbstbeschreibend* bezeichnet. Da Document Stores im Gegensatz zu Key Value Stores in der Lage sind, den Inhalt ihrer Daten zu interpretieren, können sie die Strukturinformationen der Dokumente trotz ihrer eigenen Schemafreiheit verarbeiten. Einzig der Name des Primärschlüssels (*Kundenummer*) wird in diesem Beispiel nicht gespeichert, da dieser bei Document Stores standardmäßig *_id* heißen muss. Diese Information ließe sich jedoch bei Bedarf durch ein zusätzliches Schlüssel-Wert-Paar abbilden.

Dokumente können auf die Schlüssel anderer Dokumente verweisen. Diese Referenzen müssen in der Regel jedoch manuell auf Applikationsebene verarbeitet werden. Manche APIs von MongoDB können zwar mithilfe von Code-Konventionen Referenzen auflösen, doch wird von der Verwendung dieser Technik wegen ihrer mangelnden Reife in der Dokumentation mehr oder weniger abgeraten [123, S.153].

Dokumente besitzen eine maximale Größe von 16 MB. Übersteigen Datensätze diese Größe, können sie auf mehrere Dokumente aufgeteilt werden [123, S.130].

Datenintegrität

Da Document Stores über kein festes Datenbankschema verfügen, lässt sich die Struktur der gespeicherten Dokumente auf Datenbankebene nicht zuvor definieren. Aus diesem Grund ist es möglich, dass sich innerhalb einer Collection einzelne Dokumente strukturell vollständig voneinander unterscheiden. Eine Definition und Überprüfung der Anzahl, Namen und Wertebereiche von Attributen sowie von weiteren statischen und dynamischen Integritätsbedingungen müssen demnach auf Applikationsebene erfolgen.

Eine Ausnahme stellt das Integritätskonzept von CouchDB dar. Hier lassen sich optional Integritätsbedingungen durch sogenannte Validationsfunktionen [9, S.67] definieren, die bei jeder Schreiboperation auf dem entsprechenden Dokument ausgeführt werden. Neben inhaltlichen und strukturellen Bedingungen ist auch eine Überprüfung von Benutzerrechten möglich. Wird gegen eine dieser Regeln verstoßen, wird die Schreiboperation abgewiesen.

4.2.2 Anfrageverarbeitung

Die beiden in dieser Arbeit vorgestellten Document Stores unterscheiden sich stark in der Art und Weise ihrer Anfrageverarbeitung. Während MongoDB eine sehr umfangreiche Schnittstelle bereitstellt, mit der eine SQL-ähnliche Interaktion mit dem Datenbanksystem ermöglicht wird, erfolgt die Datenanfrage in CouchDB ausschließlich mithilfe sogenannter Views. Beide Konzepte werden im Folgenden näher betrachtet.

Anfrageverarbeitung mit MongoDB

MongoDB stellt für alle gängigen Programmiersprachen eine Schnittstelle bereit, über die verschiedene Anfrage- und Änderungsoperationen auf der Datenbank ausgeführt werden können. Mit den drei Änderungsoperationen `insert`, `update` und `delete` können Dokumente eingefügt, aktualisiert oder gelöscht werden. Änderungsoperationen werden durch das folgende, dreiteilige Statement formuliert:

```
dbName.collectionName.update( { <Restriktion> }, { <Update> } ).
```

Der erste Teil des Statements ist der Name der Datenbank, der zweite Teil besteht aus dem Namen der Collection und der dritte Teil enthält den Namen der Änderungsoperation sowie einige Parameter. Änderungsoperationen lassen sich deswegen ausschließlich auf einer Datenbank und einer Collection ausführen. Sie sind in der Lage, mehrere Dokumente oder Attribute als Parameter entgegenzunehmen (`insert`, `update`) oder auf mehreren Dokumenten angewendet zu werden (`update`, `delete`) [123, S.109]. Bei den Änderungsoperationen `update` und `delete` muss die Dokumentmenge einer Collection unter Zuhilfenahme von Restriktionsbedingungen eingeschränkt werden.

Restriktionsbedingungen bestehen aus einer Liste mit verschiedenen Schlüssel-Wert-Paaren. Schlüssel repräsentieren beliebige Attribute eines Datensatzes und die jeweils

zugehörigen Werte spezifizieren Bedingungen, welche von den entsprechenden Attributwerten erfüllt werden müssen. MongoDB bietet für die Formulierung dieser Bedingungen eine Fülle von Operatoren, mit denen sich nahezu alle bekannten SQL-Operationen abbilden lassen. Eine Liste aller Operatoren findet sich unter anderem in der Dokumentation von MongoDB [123, S.48].

Leseanfragen werden mittels der Methode `find` formuliert. Diese Methode kann sowohl Restriktions- als auch Projektionsbedingungen entgegen nehmen und diese auf einer Collection ausführen:

```
dbName.collectionName.find( { <Restriktion> }, { <Projektion> } ).
```

Projektionsbedingungen bestehen aus einer Liste mit Attributnamen und einem zusätzlichen Parameter. Die ausgegebene Ergebnismenge umfasst je nach Wert dieses Parameters entweder ausschließlich Attribute dieser Liste (Parameter = 1) oder keines dieser Attribute (Parameter = −1). Sowohl Restriktions- als auch Projektionsbedingungen können reguläre Ausdrücke enthalten und somit mehrere Dokumente beziehungsweise Attribute betreffen. Abbildung 4.4 beschreibt die Ausführung einer Leseanfrage auf der Collection *Kunde*. Durch die hier aufgeführten Restriktions- und Projektionsbedingungen werden alle Namen von Kunden ausgegeben, die im Jahr *1984* geboren sind.

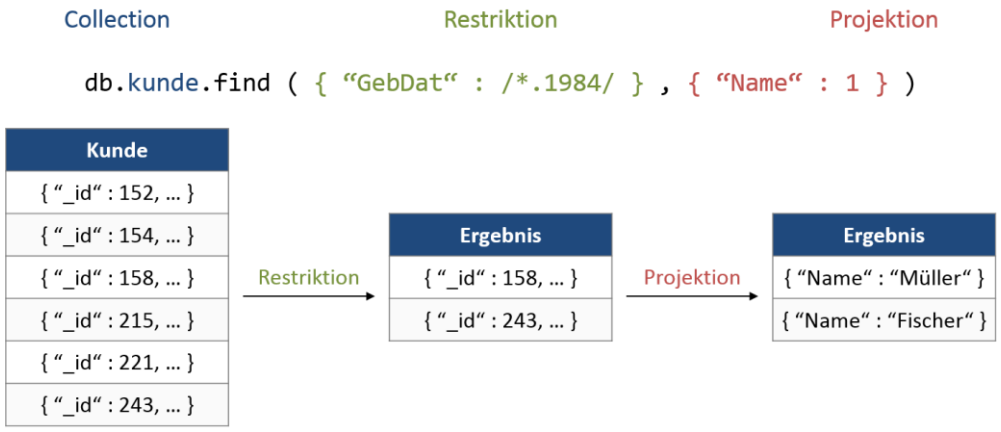


Abbildung 4.4: Formulierung von Restriktions- und Projektionsbedingungen in MongoDB

Die Formulierung dieser einfachen Leseanfragen kann in ihrem Umfang und der Mächtigkeit mit der Definition von SQL-Statements verglichen werden. In beiden Fällen können für beliebige Attribute Restriktionsbedingungen formuliert und die Ausgabe mithilfe der Projektion eingeschränkt werden. MongoDB ist jedoch im Gegensatz zu relationalen Datenbanken nicht in der Lage, auf Datenbankebene Referenzen aufzulösen, weshalb die MongoDB-API kein Pendant zum SQL-Join besitzt.

Mehrwertige und komplexe Attribute von Entitäten können mit den von JSON unterstützten Datentypen `Array` und `Object` abgebildet werden. Um auf einzelne Werte innerhalb dieser Datentypen zugreifen zu können, bietet MongoDB die Punkt-Notation an:

```
dbName.collectionName.find( { "Telefon.Mobil" : "017655512345" } ).
```

Die hier abgebildete Restriktionsbedingung in der Leseoperation `find` bezieht sich auf das komplexe Attribut *Telefon*. Innerhalb dieses Attributs, das durch den Datentyp `Object` repräsentiert wird, befindet sich das einfache Attribut *Mobil*. Mithilfe der Punkt-Notation kann die Restriktionsbedingung direkt auf dieses Attribut bezogen und alle Dokumente mit der Mobilfunknummer *017655512345* direkt gelesen werden.

Anfrageverarbeitung mit CouchDB

Die Anfrageverarbeitung in CouchDB erfolgt ausschließlich über eine REST-Schnittstelle. Entsprechend stehen die HTTP-Methoden `get`, `put` und `delete` als Basisoperationen zur Formulierung von Anfragen zur Verfügung. Um ein JSON-Dokument in eine Collection einfügen zu können, müssen der Operation `put` die URL des CouchDB-Servers, der Name der Collection, die ID des Dokuments und der JSON-Code als Parameter übergeben werden:

```
put http://<couchDB>/<collection>/<id> <json>.
```

Eine gezielte Änderung von bestimmten Attributen in einem Dokument ist auf Datenbankebene nicht möglich. Soll in CouchDB ein Attribut innerhalb eines Dokuments aktualisiert werden, muss das Dokument zunächst ausgelesen, dann auf Applikationsseite modifiziert und anschließend zurückgeschrieben werden. Das aktualisierte Dokument wird als eine neue Version des Dokuments in der Datenbank hinterlegt und die alte Version als ungültig markiert. CouchDB ähnelt insoweit einem Versionsverwaltungssystem wie Subversion. Dokumente können mit der Operation `delete` unter Angabe der Dokument-ID und der Versionsnummer gelöscht werden. Die abgespeicherten JSON-Dokumente lassen sich mit der Methode `get` wieder auslesen:

```
get http://<couchDB>/<collection>/<id>.
```

Mit den Methoden `get`, `put` und `delete` lassen sich einzelne JSON-Dokumente sehr einfach speichern und wieder aufrufen. Um auch komplexere Anfragen formulieren und ausführen zu können, müssen sogenannte Design-Dokumente verwendet werden. Ein Design-Dokument ist wie ein gewöhnliches Dokument in CouchDB hinterlegt und kann über eine *_id* eindeutig angesprochen werden. Design-Dokumente enthalten eine oder mehrere sogenannte Views. Eine View besteht wiederum aus ein oder zwei in JavaScript formulierten Funktionen und kann über einen eindeutigen Namen angesprochen werden:

```
get http://<couchDB>/<collection>/.../<_id>/<viewname>.
```

Wird eine View erstmalig über ihre URL aufgerufen, werden die in ihr definierten Funktionen auf jedem Dokument der Collection ausgeführt. Die wichtigste dieser Funktionen ist die sogenannte Map-Funktion. Diese nimmt als Parameter ein Dokument entgegen und führt anschließend beliebig definierbare Restriktionsbedingungen auf diesem Dokument aus. Als Ergebnis erzeugt die Map-Funktion für jedes Dokument, auf dem sie ausgeführt wurde, ein Schlüssel-Wert-Paar, dessen Wert von den definierten Projektionsbedingungen abhängt. Die Ergebnismenge einer View umfasst infolgedessen eine sortierte Liste von Schlüssel-Wert-Paaren. Abbildung 4.5 beschreibt eine CouchDB-View, welche die gleiche Ergebnismenge wie die in Abbildung 4.4 dargestellte Anfrage in MongoDB erzeugt.

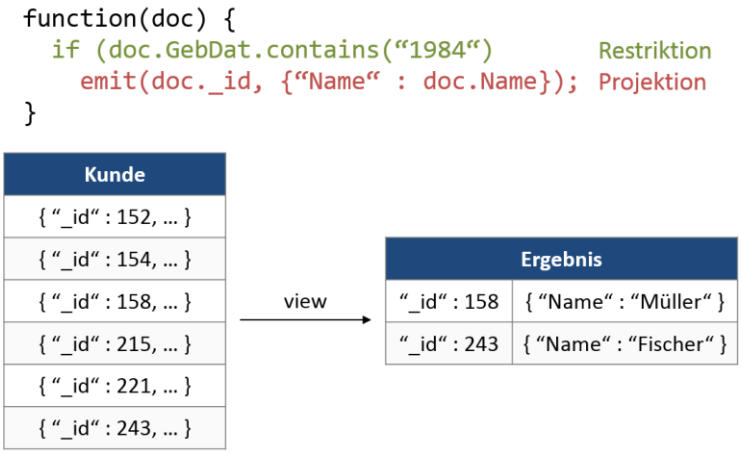


Abbildung 4.5: Formulierung von Restriktions- und Projektionsbedingungen in CouchDB

Die Ergebnismenge einer View lässt sich mithilfe von Parametern weiter reduzieren. Diese können beim Aufruf der View übergeben werden und schränken den Wertebereich der Schlüssel in der Ergebnismenge ein. Der folgende Aufruf gibt die Namen aller Kunden zurück, die *1984* geboren wurden und eine Kundennummer zwischen *150* und *159* besitzen:

```
get http://.../<_id>/<viewname>?starkey="150"&endkey="159".
```

Neben den Map-Funktionen können Views auch sogenannte Reduce-Funktionen enthalten. Diese werden nach der Map-Funktion ausgeführt und dienen in erster Linie der Aggregation sowie der Weiterverarbeitung der Ergebnismenge. So kann eine Reduce-Funktion beispielsweise die Anzahl an männlichen und weiblichen Kunden in der in Abbildung 4.5 dargestellten Ergebnismenge berechnen. Das MapReduce-Konzept wird in Kapitel 4.4.3 ausführlich vorgestellt.

Die Ausführungskosten einer View sind bei ihrem erstmaligen Aufruf sehr hoch, da die in ihr hinterlegten Funktionen auf jedem Dokument einer Collection angewendet werden müssen. Aus diesem Grund sind sogenannte Ad-hoc-Anfragen, bei denen eine

beliebige Anfrage mithilfe einer View formuliert und einmalig ausgeführt wird, in CouchDB sehr ungünstig. Um die Kosten häufig aufgerufener Views deutlich zu reduzieren, werden die erzeugten Ergebnismengen in jeweils einem B*-Baum hinterlegt. Ändert sich ein Dokument oder wird ein neues Dokument hinzugefügt, werden die Map- und Reduce-Funktionen auf diesem Dokument ausgeführt und die Ergebnismenge automatisch aktualisiert.

Relationale Algebra	Document Stores
Projektion	✓
Restriktion	✓
Kartesisches Produkt	✗
Vereinigung	✗
Differenz	✗
Umbenennung	✓
Operator	**Document Stores**
Gleich	✓
Ungleich	✓
Größer, Kleiner	✓
Like	✓
In	✓
Exists	✓
And, Or	✓
Not	✓

Tabelle 4.3: Unterstützte relationale Operationen und Operatoren

In Tabelle 4.3 ist der Funktionsumfang von Document Stores abgebildet. Obwohl sich MongoDB und CouchDB in ihrer Anfrageverarbeitung sehr stark voneinander unterscheiden, gleichen sie sich in ihrer Ausdrucksstärke. Einerseits unterstützen beide Systeme keine Verbundoperationen und können deshalb nicht als relational vollständig bezeichnet werden. Andererseits stellen sie alle gängigen Operationen zur Formulierung von Restriktionsbedingungen zur Verfügung und sind obendrein in der Lage, komplexe Datenstrukturen wie Arrays oder Objekte zu durchsuchen.

4.2.3 Datenmodellierung

Für die Überführung eines konzeptuellen Datenmodells in das logische Datenmodell eines Document Stores gibt es derzeit keine allgemeingültigen Abbildungsregeln. Die Gründe hierfür liegen laut der Dokumentation von MongoDB [120] vor allem an dem vergleichsweise sehr jungen Alter von Document Stores. Aufgrund fehlender praktischer Erfahrungen im Umgang mit diesen Systemen seien derzeit noch keine Aussagen über Modellierungsmuster möglich. Diese sind jedoch zwingend erforderlich, um

die Vorteile und Nachteile eines Datenmodells nachvollziehen zu können. Des Weiteren ermöglichen sie vor allem Anwendern mit einer geringen praktischen Erfahrung im Umgang mit Document Stores einen wesentlich leichteren Einstieg in diese alles andere als triviale Thematik.

Auch wenn es der dokumentorientieren Datenmodellierung an praktischer Erfahrung mangelt, darf dieses Argument nicht für die bisher unzureichende Arbeit auf konzeptueller Ebene herangezogen werden. Basierend auf der in diesem Kapitel bereits erfolgten konzeptuellen Aufarbeitung und der Analyse der bereitgestellten Operationen und Operatoren lassen sich durchaus allgemeingültige Abbildungsregeln und Modellierungsprozesse erstellen, mit denen sich die Strukturen des konzeptuellen Datenmodells auf die Elemente des logischen Datenmodells von Document Stores effizient abbilden lassen. Diese werden im Folgenden näher vorgestellt.

Entitäten

Document Stores verfügen über ein sehr ausdrucksstarkes Datenmodell, mit dem sich nahezu alle Prädikatoren erster und zweiter Ordnung abbilden lassen. Darüber hinaus besitzen diese Datenbanksysteme Schnittstellen mit einem sehr großen Funktionsumfang, sodass die Datenmodellierung deutlich weniger stark von möglichen Zugriffsmustern beeinflusst wird, wie es beispielsweise bei Key Value Stores der Fall ist. Aus diesen Gründen können Entitäten des logischen Datenmodells, wie in der relationalen Datenmodellierung üblich, durch jeweils eine Relation beziehungsweise Collection abgebildet werden. Innerhalb einer Collection werden einzelne Realweltobjekte durch jeweils ein Dokument repräsentiert. Die Eigenschaften des Objekts werden als Schlüssel-Wert-Paare im zugehörigen Dokument gespeichert. Das Schlüsselattribut *_id*, über das ein Dokument eindeutig identifiziert wird, kann entweder manuell oder vom Datenbanksystem einen Wert zugewiesen bekommen. Je nach Modellierungsentscheidung kann eine separate Abbildung des Primärschlüssels als gewöhnliches Attribut notwendig sein. Zur Veranschaulichung dieses Modellierungsmusters wird die in Abbildung 4.6 dargestellte Entität *Kunde* herangezogen.

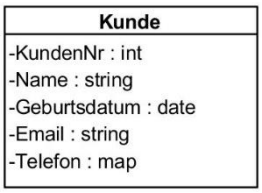


Abbildung 4.6: Konzeptuelle Darstellung der Entität *Kunde*

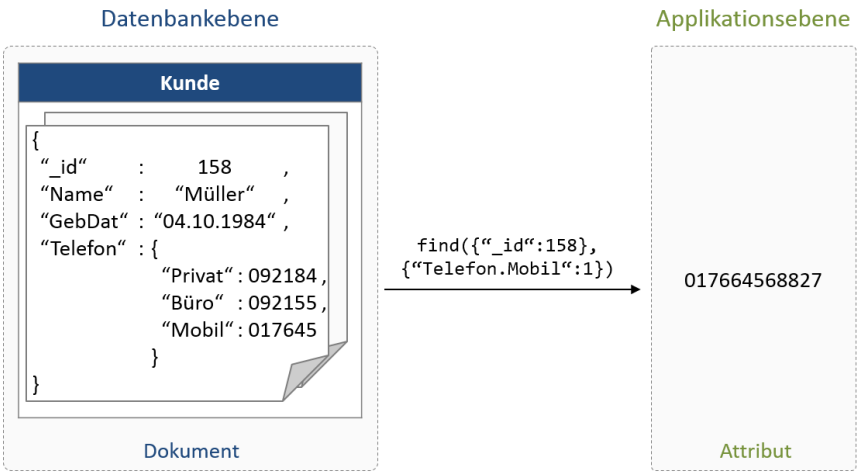


Abbildung 4.7: Abbildung und Anfrage von kompositen Attributen

Eine mögliche logische Abbildung der Entität *Kunde* ist in Abbildung 4.7 dargestellt. Der Datensatz mit der Kundenummer *158* wird gemäß der vorgestellten Abbildungsregel durch ein Dokument repräsentiert und die einfachen Attribute *Name* und *Geburtsdatum* auf primitive Datentypen abgebildet. Zur Darstellung des komplexen Attributs *Telefonnummer* bietet sich der Datentyp `Object` an. Die Verwendung dieses Datentyps ermöglicht eine natürliche Abbildung des Attributs und benötigt daher keine Transformation auf mehrere einfache Attribute, wie sie in der relationalen Datenmodellierung üblich ist. In Abbildung 4.7 wird zudem die Anfragemächtigkeit von Document Stores deutlich. Eine Projektion auf das Attribut *Mobil*, das sich im komplexen Attribut *Telefon* befindet, kann mithilfe der Punkt-Notation vollständig auf der Seite der Datenbank durchgeführt werden. Dabei ergibt sich kein Mehraufwand auf Applikationsebene.

Da Document Stores über kein globales Schema verfügen, sondern die einzelnen Dokumente diese Informationen implizit enthalten, können sich die in einer Collection verwalteten Dokumente in ihrer Struktur voneinander unterscheiden. Demzufolge ist bei der Datenmodellierung eine separate Abbildung von generalisierten und spezialisierten Entitäten in verschiedene Collections nicht notwendig.

Beziehungen

Document Stores sind weder in der Lage Fremdschlüsselbeziehungen aufzulösen noch können sie die Integrität von Referenzen sicherstellen. Dafür stellen sie die komplexen Datentypen `Array` und `Object` bereit und eröffnen so völlig neue Möglichkeiten in der Abbildung von Beziehungen. Diese neuen Möglichkeiten lassen sich entsprechend der im vorherigen Kapitel erfolgten Klassifizierung in Referenzierung, Denormalisierung und Aggregation unterteilen. Die Vor- und Nachteile der verschiedenen Modellierungsmöglichkeiten werden im Folgenden untersucht. Basierend auf den Ergebnissen

dieser Untersuchung werden allgemeingültige Aussagen zur bestmöglichen Abbildung von Beziehungen in Document Stores abgeleitet. Zur Veranschaulichung der verschiedenen Modellierungstechniken wird das bereits bekannte logische Datenmodell eines Onlineshops herangezogen (Abbildung 4.8).

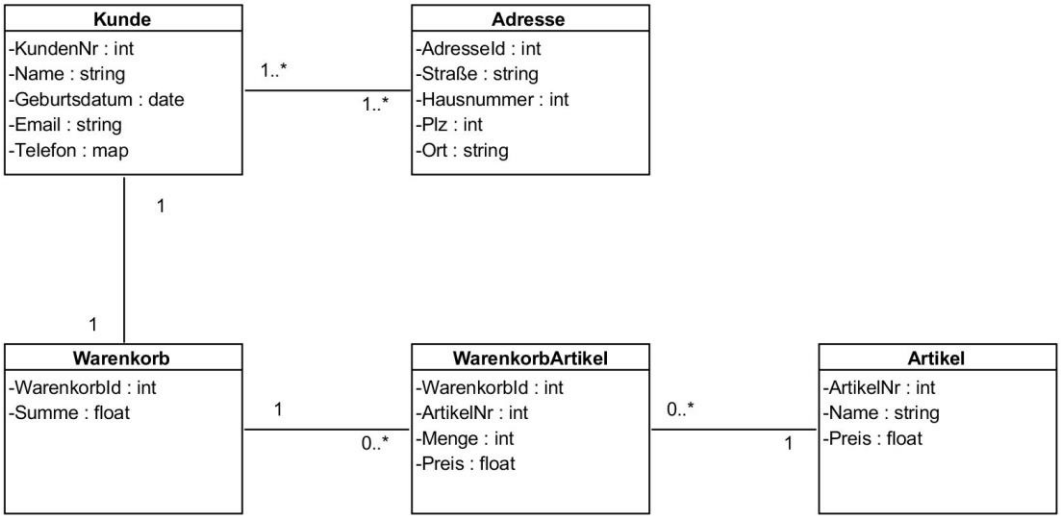


Abbildung 4.8: Konzeptuelles Datenmodell eines Online-Shops

Referenzierung: Ein Attribut kann auf das Schlüsselattribut eines anderen Dokuments verweisen und es somit, ähnlich wie bei relationalen Datenbanken, referenzieren. Da Document Stores jedoch keine Verbundoperationen unterstützen, müssen diese Referenzen manuell auf Applikationsebene aufgelöst werden. Document Stores ermöglichen einen effizienten Zugriff auf Fremdschlüssel mithilfe von sekundären Zugriffspfaden. Folglich müssen Beziehungen nicht wie in Key Value Stores zwingend redundant auf beiden Seiten einer Beziehung gespeichert werden, um diese in beide Richtungen auflösen zu können. Dank der Datentypen `Array` und `Object` sind Dokumente außerdem in der Lage, mit einem Attribut auf mehrere andere Dokumente zu verweisen. Dadurch lassen sich auch 1:N-Beziehungen auf einer beliebigen Seite einer Beziehung abbilden. Die Entscheidung, auf welcher Seite einer Beziehung die Referenzen gespeichert werden sollen, hängt von den zu unterstützenden Anfragen ab. Von einer Modellierung von M:N-Beziehungen durch den Einsatz von Join-Dokumenten beziehungsweise Join-Tabellen, wie sie bei der relationalen Datenmodellierung üblich sind, muss angesichts des hohen Aufwands auf Applikationsebene abgeraten werden. Vielmehr bietet sich hierfür eine Umwandlung der M:N-Beziehung in zwei entgegen gerichtete 1:N-Beziehungen an.

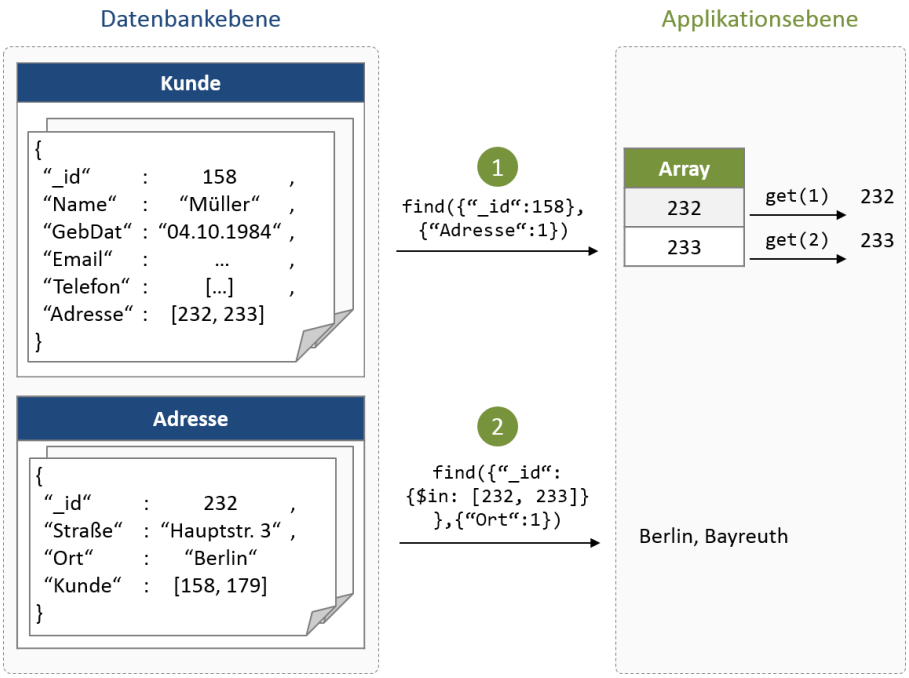


Abbildung 4.9: Zweistufige Auflösung von referenzierten Adressen

In Abbildung 4.9 sind zwei Ausprägungen der beiden Entitäten *Kunde* und *Adresse* als jeweils ein Dokument abgebildet. Die zwischen diesen beiden Entitäten bestehende M:N-Beziehung wurde gemäß dem hier vorgestellten Modellierungsmuster in zwei entgegen gerichtete 1:N-Beziehungen umgewandelt. Das Dokument *158* in der Collection *Kunde* referenziert zwei Dokumente der Collection *Adresse*, das Dokument *232* in der Collection *Adresse* verweist auf zwei Dokumente der Collection *Kunde*. Zur Abbildung der Referenzen wurde in beiden Fällen der Datentyp `Array` verwendet. Der durch die Referenzierung bei Document Stores implizierte Mehraufwand auf Applikationsebene wird ebenfalls in Abbildung 4.9 deutlich. Sollen alle zu einem Kunden zugehörigen Adressen ermittelt werden, muss zunächst das entsprechende `Array` ausgelesen werden (Schritt 1). Anschließend müssen die Referenzen durch zusätzliche Anfragen an die Collection *Adresse* aufgelöst werden (Schritt 2). Die redundante Speicherung der M:N-Beziehung mittels zweier 1:N-Beziehungen erschwert zudem die Wartung, da diese Informationen in zwei Dokumenten in verschiedenen Collections gespeichert sind. Ein weiterer Nachteil in der Verwendung von Referenzen ergibt sich durch die fehlende Sicherung der referenziellen Integrität, was wiederum einen Mehraufwand auf Applikationsebene nach sich ziehen kann.

Um auf Dokumente einer anderen Collection verweisen zu können, ist die Angabe der Schlüsselattribute nicht ausreichend. MongoDB hat deshalb die sogenannten DBRefs [123, S.150] vorgestellt. Hierbei handelt es sich um eine Modellierungskonvention für Referenzen, bei der neben der Dokument-ID auch die Namen der Collection und der Datenbank angegeben werden.

Denormalisierung: Um die Anzahl weiterer Datenbankaufrufe bei der applikationsseitigen Auflösung von Referenzen zu minimieren, können bestimmte Attribute redundant gespeichert werden. So kann beispielsweise die primäre Adresse eines Kunden redundant im Dokument *Kunde* gespeichert und im Fall einer Änderung asynchron aktualisiert werden.

Aggregation: Bei der Aggregation werden zusammengehörige Entitäten in einem Dokument gespeichert. Dabei werden semantisch untergeordnete Entitäten als Unterelemente der semantisch übergeordneten Entitäten abgebildet. Besonders die Beziehungen zwischen starken und schwachen Entitäten lassen sich mithilfe der Datentypen `Object` und `Array` bequem abbilden. Die Aggregation bietet sich zur Abbildung von 1:1- oder 1:N-Beziehungen an. M:N-Beziehung lassen sich dagegen nur zu Aggregaten zusammenfassen, wenn die schwache Entität redundant gespeichert wird.

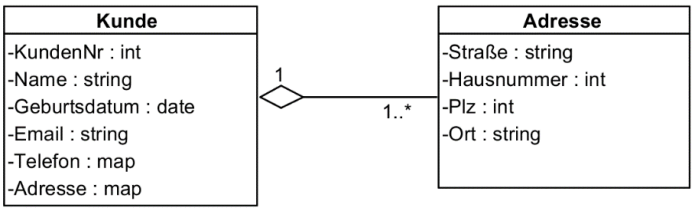


Abbildung 4.10: Konzeptuelle Darstellung der Aggregation *Kunde-Adresse*

Das bereits aus Kapitel 3 bekannte Beispiel, bei dem die Entität *Adresse* in die Entität *Kunde* eingebettet wird, ist in Abbildung 4.10 erneut dargestellt. Document Stores sind aufgrund ihrer Anfragemächtigkeit sehr gut für die Verarbeitung von aggregierten Dokumenten geeignet. Im Gegensatz zu Key Value Stores ermöglichen sie den direkten Zugriff auf eingebettete Entitäten. So kann beispielsweise, wie in Abbildung 4.11 dargestellt, durch die Punkt-Notation direkt auf die Orte der Adressen des Kunden zugegriffen werden, ohne einen zusätzlichen Aufwand auf Applikationsebene zu verursachen.

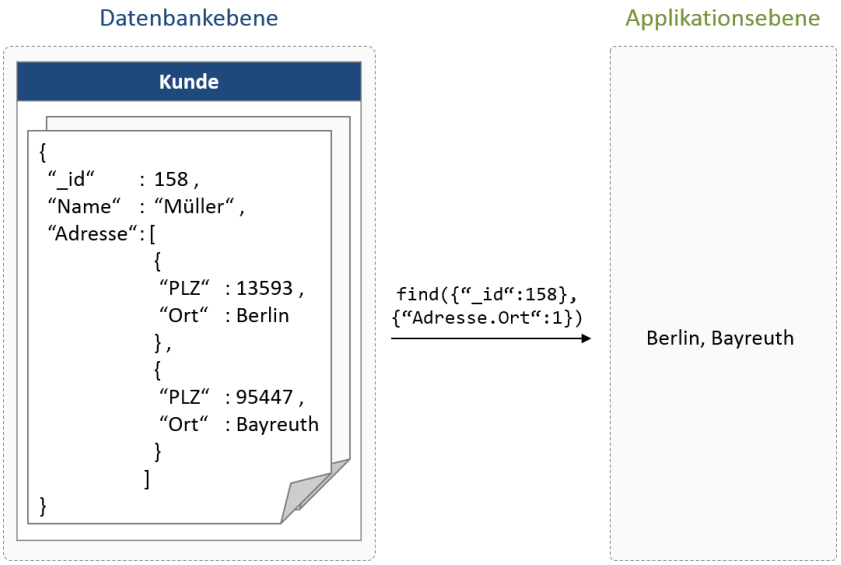


Abbildung 4.11: Logische Abbildung der Aggregation *Kunde-Adresse*

Der Zugriff auf eingebettete Dokumente ist nicht nur der Projektion vorenthalten, sondern kann auch in der Restriktionsbedingung herangezogen werden, um die Ergebnismenge einzuschränken. So kann beispielsweise die Suche nach allen Kunden eines bestimmten Orts datenbankseitig durchgeführt werden. Werden solche Anfragen häufiger ausgeführt, ist das Anlegen eines sekundären Zugriffspfads auf das Attribut *Ort* empfehlenswert.

Aggregate ermöglichen einen eleganten und effizienten Umgang mit Beziehungen auf Applikationsebene. Da zusammenhängende Daten innerhalb eines Dokuments abgebildet werden und demzufolge am selben Speicherplatz liegen, ist für die Auflösung dieser Beziehungen nur ein einziger Datenbankaufruf notwendig. Gerade bei mehrwertigen Beziehungen ergibt sich hierdurch ein hohes Einsparpotenzial an Kommunikationskosten. Deswegen ist das Konzept der Aggregation vor allem in der verteilten Datenhaltung ein wichtiges Instrument, um die Skalierbarkeitseigenschaften eines Datenbanksystems zu verbessern. Darüber hinaus bieten Aggregate eine sehr einfache Form der Transaktionsverwaltung. Da alle Operationen auf einem Dokument atomar sind, werden auch alle zusammenhängenden Operationen auf einem Aggregat entweder vollständig oder gar nicht ausgeführt.

Der Nachteil der Aggregation liegt in der eingeschränkten Anfrageverarbeitung. Aggregate sind auf die performante Ausführung einer bestimmten Anfrageart ausgelegt. Anfragen, die von diesem Zugriffsmuster abweichen, erweisen sich in der Regel als äußerst kostspielig. Soll beispielsweise auf dem oben abgebildeten Datenmodell die Anzahl der Bestellungen eines bestimmten Artikels ermittelt werden, müssen umfangreiche Anfragen auf allen Aggregaten durchgeführt werden.

Aggregate sollten ausschließlich bei 1:1- oder 1:N-Beziehungen eingesetzt werden. Besonders bei der Abbildung von starken und schwachen Entitäten sowie von Baumstrukturen empfiehlt sich eine Einbettung. Da sich M:N-Beziehungen mithilfe der Aggregation nur durch eine redundante Abbildung der eingebetteten Entität realisieren lassen, sollte für solche Beziehungen die Referenzierung vorgezogen werden. Referenzierung bietet sich außerdem in Anwendungsfällen an, in denen eine höhere Flexibilität bei der Anfrageformulierung benötigt wird.

Modellierungsprozess

Document Stores besitzen ein sehr flexibles und zugleich sehr ausdrucksstarkes Datenmodell. Unter Zuhilfenahme der in diesem Kapitel vorgestellten Modellierungsmuster, lassen sich sämtliche Elemente des konzeptuellen Datenmodells systematisch auf Dokumente von Document Stores abbilden. Eine besondere Rolle nehmen dabei die Datentypen `Array` und `Object` ein. Dank dieser Datentypen lassen sich nicht nur zusammengesetzte und mehrwertige Attribute problemlos abbilden, sie ermöglichen auch eine sehr effiziente Abbildung von Beziehungen. Es ergeben sich die in Tabelle 4.4 dargestellten Entsprechungen zwischen den beiden verschiedenen Modellen.

ER-Modell	Document Stores
Entitätstyp	Collection
Entität	Document
Einfaches Attribut	Schlüssel-Wert-Paar
Zusammengesetztes Attribut	Object
Mehrwertiges Attribut	Array
1:1-Beziehung	Referenzierung, Denormalisierung, Aggregation
1:N-Beziehung	
M:N-Beziehung	

Tabelle 4.4: Übereinstimmungen zwischen dem ER-Modell und Document Stores

Neben einem flexiblen und sehr ausdrucksstarken Datenmodell verfügen Document Stores über einen sehr hohen Umfang an Operationen und Operatoren zur Formulierung von Projektions- und Restriktionsbedingungen. Da diese auf jedem Schlüssel-Wert-Paar in einem Dokument angewendet werden können, weist die Datenverarbeitung in Document Stores eine ähnliche Mächtigkeit wie bei relationalen Datenbanken auf. Document Stores sind deshalb, ebenso wie relationale Datenbanken, dafür prädestiniert, mehrere Entitäten und Beziehungen umfassende Miniwelten abzubilden. Der einzige offensichtliche Unterschied in der Datenverarbeitung zwischen diesen beiden Datenbanken wird bei der Abbildung von Beziehungen deutlich. Im Gegensatz zu relationalen Datenbanken unterstützen Document Stores keine Verbundoperationen, weshalb Fremdschlüsselbeziehungen nicht datenbankseitig, sondern applikationsseitig aufgelöst werden müssen. Um den Mehraufwand und damit die Komplexität auf

Anwendungsebene zu reduzieren, werden Beziehungen zwischen Entitäten in Document Stores vorzugsweise mithilfe der Aggregation abgebildet.

Da die Aggregation eine zentrale Stellung in der dokumentorientierten Datenmodellierung einnimmt und sie die Abbildung aller an einer Beziehung beteiligten Entitäten sehr stark beeinflusst, sollte dieses Modellierungsmuster gleich zu Beginn des Datenbankentwurfs berücksichtigt werden. Der im Folgenden vorgestellte Modellierungsprozess besteht aus vier Schritten:

1. Modellierung von Aggregationen,
2. Abbildung starker Entitäten,
3. Abbildung schwacher Entitäten,
4. Abbildung von Beziehungen.

Modellierung von Aggregationen: Um die Anzahl der später abzubildenden Entitäten und Beziehungen zu reduzieren und somit den gesamten Prozess der Datenmodellierung zu vereinfachen, sollten Aggregationen bereits auf konzeptueller Ebene modelliert werden. Hierzu ist zunächst die Identifikation der starken Entitäten notwendig, welche eine zentrale Stellung in der zu modellierenden Miniwelt einnehmen. Im Fall des konzeptuellen Datenmodells eines Onlineshops sind das die drei Entitäten *Kunde*, *Warenkorb* und *Artikel*. Die schwachen Entitäten *Adresse* und *Warenkorb-Artikel* werden nicht direkt angesprochen und können daher als Teile einer Aggregation modelliert werden. Die hieraus resultierenden disjunkten Teilmengen der zu modellierenden Miniwelt sind in Abbildung 4.12 dargestellt.

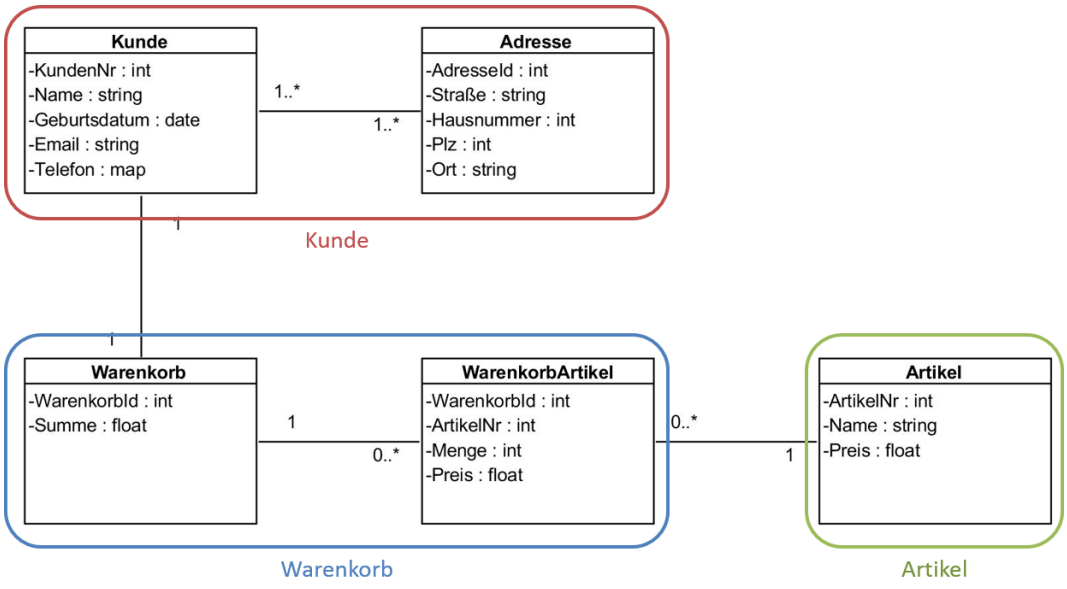


Abbildung 4.12: Identifikation von Teilmengen auf konzeptueller Ebene

Die ermittelten Teilmengen können anschließend zur Modellierung von Aggregationen herangezogen werden. Hierbei wird den schwachen Entitäten der Primärschlüssel entzogen und ein entsprechendes Mengenattribut zur jeweils zugehörigen starken Entität hinzugefügt. Als Ergebnis des ersten Modellierungsschritts ergibt sich das in Abbildung 4.13 dargestellte konzeptuelle Datenmodell.

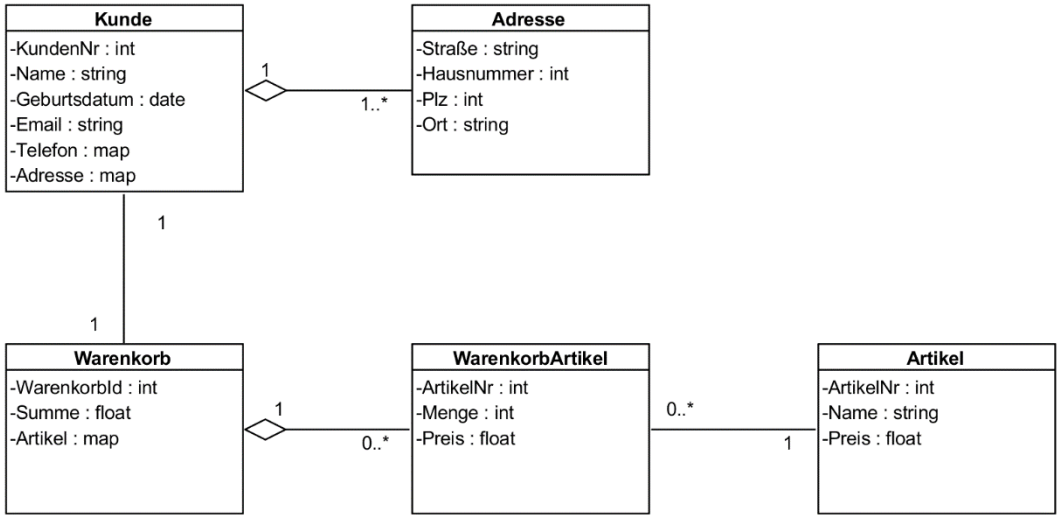


Abbildung 4.13: Modellierung der identifizierten Teilmengen als Aggregationen

Abbildung starker Entitäten: Nachdem das konzeptuelle Datenmodell angepasst wurde, können dessen starke Entitäten in das logische Datenmodell von Document Stores überführt werden. Hierzu können die in diesem Kapitel vorgestellten Abbildungsregeln verwendet werden.

Abbildung schwacher Entitäten: Sobald alle starken Entitäten abgebildet wurden, können die schwachen Entitäten gemäß der modellierten Aggregationen in die Dokumente der zugehörigen starken Entitäten eingebettet werden.

Abbildung von Beziehungen: Abschließend werden die verbleibenden Beziehungen zwischen den einzelnen Entitäten abgebildet. Bei der Verwendung der hier vorgestellten Abbildungsregeln muss der applikationsseitige Mehraufwand berücksichtigt werden, der sich durch die Auflösung von Fremdschlüsselbeziehungen oder die Wartung redundant gespeicherter Informationen ergeben kann. Des Weiteren muss abhängig vom Anwendungsfall entschieden werden, ob eine Beziehung uni- oder bidirektional abgebildet werden soll.

In Abbildung 4.14 ist das entsprechende logische Datenmodell des Onlineshops dargestellt. Die drei starken Entitäten *Kunde*, *Warenkorb* und *Artikel* wurden als Collections abgebildet und die schwachen Entitäten *Adresse* und *WarenkorbArtikel* wurden in die zugehörigen Dokumente eingebettet.

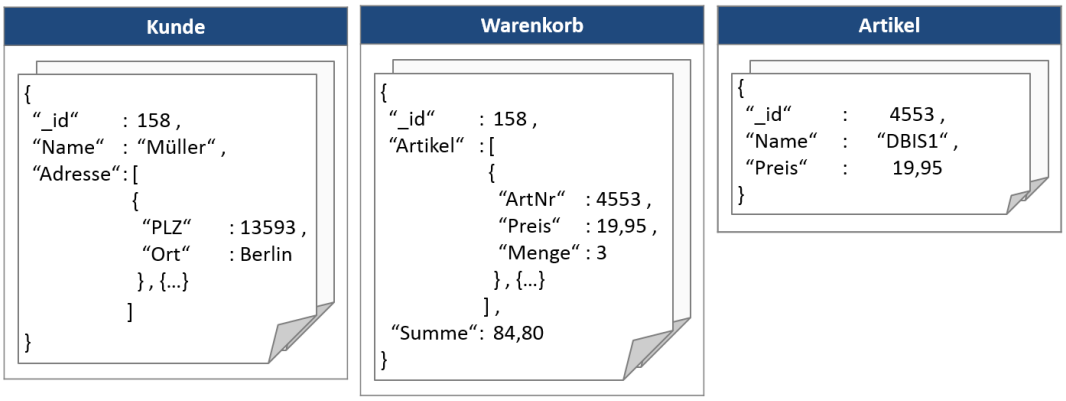


Abbildung 4.14: Logische Darstellung der aggregierten Entitäten

Anhand der in diesem Kapitel entwickelten Abbildungsregeln sowie des hier vorgestellten Modellierungsprozesses wird deutlich, dass die Datenmodellierung bei Document Stores vor allem von der Aggregation bestimmt wird. Da sich Aggregate auf konzeptueller Ebene modellieren lassen, ist eine systemunterstützte Datenmodellierung mithilfe von CASE-Werkzeugen durchaus möglich.

4.3 Interne Ebene

Document Stores sind auf eine effiziente Verarbeitung von aggregierten Daten ausgelegt. Hierzu bieten diese Systeme ein ausdrucksstarkes Datenmodell und eine umfangreiche Anfragefunktionalität. Wie alle NoSQL-Datenbanken ermöglichen Document Stores eine performante Verarbeitung sehr großer Datenmengen. Neben eines von ACID abweichenden Transaktionskonzepts bieten sie eine einfach zu administrierende und horizontal skalierende Verteilung. Die diesen Systemen zugrunde liegenden Konzepte werden in diesem Unterkapitel näher vorgestellt.

4.3.1 Speicherstrukturen

Document Stores verwenden ähnliche Speicherstrukturen wie relationale Datenbanken. Datenbankdateien werden persistent auf Festplatten gespeichert. Benötigte Dokumente werden in den Hauptspeicher geladen und stehen dort als Seiten solange zur Verfügung, bis andere Daten den Speicherplatz benötigen. Als primären Zugriffspfad werden B*-Bäume verwendet.

In MongoDB können für einen schnelleren Zugriff auf bestimmte Attribute Indizes definiert werden. Neben einfachen Attributen, wie beispielsweise dem *_id*-Feld, das als

primärer Zugriffspfad standardmäßig indiziert wird, können auch mehrwertige Attribute und eingebettete Dokumente über einen Index angesprochen werden [123, S.419]. Der für eine Anfrage passende Zugriffspfad wird vom Query Optimizer ausgewählt. Da Indizes bei Schreiboperationen synchron aktualisiert werden, ergibt sich, ähnlich wie bei relationalen Datenbanken, ein Overhead, der für die performante Ausführung von Schreiboperationen hinderlich ist. Indizes können aufsteigend oder absteigend definiert werden. Ein aufsteigender Index auf dem Attribut `Stadt` der Collection `students` besitzt folgende Notation:

```
db.students.ensureIndex( { “Anschrift.Stadt“ : 1 } ).
```

MongoDB enthält eine Reihe von zusätzlichen Indizes. So kann mithilfe des Unique-Index die Eindeutigkeit eines Attributs innerhalb einer Collection gewährleistet werden. Der sogenannte Geospatial-Index ermöglicht das Auffinden eines nahegelegenen Objekts innerhalb eines zweidimensionalen Raums. Hierzu wird ein Array indiziert, dass aus Koordinaten besteht [123, S.432]. MongoDB erlaubt maximal 64 Indizes pro Collection.

Aufgrund des View-Konzepts unterstützt CouchDB keine sekundären Zugriffspfade. Sollen Anfragen auf bestimmte Attribute durchgeführt werden, muss zuvor eine entsprechende View definiert werden. Die mit den dort implementierten Funktionen ermittelten Ergebnisse werden anschließend in einem B*-Baum gespeichert und somit ein effizienter Zugriff auf diese Ergebnismenge ermöglicht [9, S.53].

4.3.2 Transaktionen und konkurrierende Zugriffe

Aufgabe der Transaktionsverwaltung ist die sichere Ausführung von Operationen und die dauerhafte Speicherung der durchgeführten Änderungen trotz potenzieller Fehlerquellen, wie zum Beispiel Mehrbenutzerzugriffe und Systemfehler. Hierzu enhält eine Transaktionsverwaltung die beiden Komponenten Recovery und Synchronisation. Der Umgang von Document Stores mit Transaktionen und konkurrierenden Zugriffen wird im Folgenden anhand des bereits bekannten Datenmodells eines Warenkorbs veranschaulicht (Abbildung 4.15).

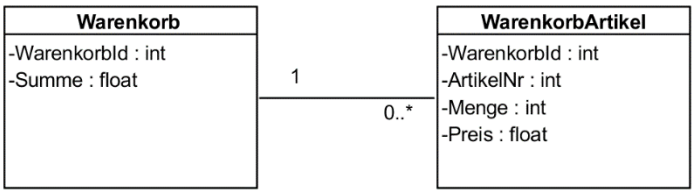


Abbildung 4.15: Konzeptuelles Modell der Entitäten *Warenkorb* und *WarenkorbArtikel*

Recovery

Document Stores unterscheiden sich in der Art und Weise, wie Änderungsoperationen durchgeführt werden. Soll ein Dokument in CouchDB aktualisiert werden, muss das entsprechende Dokument aus der Datenbank ausgelesen werden. Daraufhin kann auf Applikationsebene eine beliebige Anzahl an Operationen auf dem Dokument ausgeführt werden, bevor das modifizierte Dokument mit einer Operation zurück in die Datenbank eingefügt wird. Bei diesem Konzept sind automatisch alle Operationen, die auf einem Dokument ausgeführt werden, atomar. Es können auch mehrere Dokumente gleichzeitig auf Applikationsebene geändert werden, doch die anschließende Sicherung der Dokumente erfolgt jeweils getrennt voneinander. Im Gegensatz dazu erfolgt die Datenmanipulation in MongoDB auf Datenbankebene.

MongoDB unterstützt keine Transaktionen. Aus diesem Grund lassen sich Operationen nicht zu einer atomaren Einheit zusammenfassen, sondern müssen jeweils einzeln ausgeführt werden. Betreffen die Änderungen einer Operation nur ein Dokument, werden diese Änderungen atomar durchgeführt. Umfasst eine Operation jedoch Änderungen auf mehreren Dokumenten, werden diese unabhängig voneinander ausgeführt. Im Falle eines Fehlers ist es Aufgabe der Applikationslogik, die bereits durchgeführten Änderungen zurückzusetzen oder die fehlenden Änderungen erneut einzuspielen [123, S.102]. Trotz dieser unterschiedlichen Verarbeitungskonzepte von Operationen lassen sich sowohl in CouchDB als auch in MongoDB zwei Entitäten nicht mit einer atomaren Operation verändern.

Die Folgen einer mangelnden Transaktionsunterstützung werden bei dem in Abbildung 4.15 dargestellten Beispiel beim Hinzufügen eines Artikels zu einem Warenkorb deutlich. In diesem Fall muss sowohl das Attribut *Summe* im Warenkorb aktualisiert, als auch ein neues Dokument in die Collection *WarenkorbArtikel* eingefügt werden. Diese beiden Operationen müssen bei Document Stores unabhängig voneinander ausgeführt werden und können daher zu Inkonsistenzen führen, sobald eine der beiden Operationen während der Ausführung fehlschlägt. Um solche Inkonsistenzen zu vermeiden, muss die transaktionale Atomarität bei Document Stores auf Anwendungsebene sichergestellt werden. Der hierfür notwendige Aufwand wird unter anderem in der Dokumentation von MongoDB deutlich, in der die manuelle Umsetzung eines Zwei-Phasen-Commit-Protokolls detailliert beschrieben wird [123, S.93].

Die mangelnde Transaktionsunterstützung kann in manchen Fällen wesentlich eleganter und mit einem deutlich geringeren Aufwand durch eine geschickte Datenmodellierung kompensiert werden. Wie im vorherigen Unterkapitel beschrieben, können zusammenhängende Sachverhalte mithilfe der Aggregation zu einem Dokument zusammengefasst werden. Anschließend lassen sich diese Informationen mit einer Operation atomar verändern.

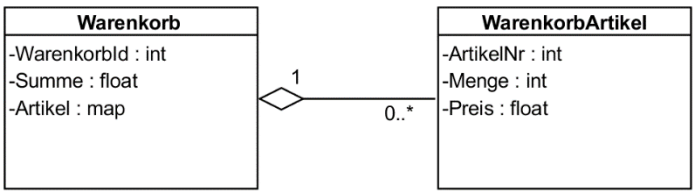


Abbildung 4.16: Aggregation der Entitäten *Warenkorb* und *WarenkorbArtikel*

In Abbildung 4.16 ist die Aggregation der Entitäten *Warenkorb* und *WarenkorbArtikel* dargestellt, bei der jeder Warenkorb mehrere Artikel enthalten kann. Wird ein neuer Artikel zum Warenkorb hinzugefügt, können sowohl der neue Artikel als auch das Attribut *Summe* atomar mit einer Operation verändert werden.

Dauerhaftigkeit wird in MongoDB mittels einer Log-Datei (WAL-Prinzip) sichergestellt, die einmal alle 100 Millisekunden persistiert wird [123, S.207]. Erst wenn eine Änderungsoperation dort eingetragen wurde, wird die Operation gegenüber dem Client bestätigt. Zur Steigerung der Performanz kann das Sicherungsintervall der Log-Datei geändert werden [123, S.208]. Hierdurch steigt jedoch die Gefahr eines Datenverlustes im Fall eines Hauptspeicherfehlers. Schreiboperationen werden in CouchDB auf Applikationsebene durchgeführt. Das geänderte Dokument wird daraufhin als eine neue Version in die Datenbank eingefügt und erst dann gegenüber dem Client bestätigt, wenn das Dokument erfolgreich persistiert wurde. Ein Datenverlust infolge eines Hauptspeicherfehlers ist in CouchDB deswegen nicht möglich [9, S.235].

Synchronisation

Zur Synchronisation von konkurrierenden Operationen verwendet MongoDB globale Lese-Schreib-Sperren. Ist eine solche Sperre gesetzt, können bis zur Aufhebung der Sperre weder Lese- noch Schreiboperationen auf der gesamten Datenbank ausgeführt werden [123, S.688]. Im Gegensatz dazu verwendet CouchDB mit MVCC (siehe Kapitel 2.3.2) ein optimistisches Synchronisationsverfahren, bei dem einzelne Operationen nicht durch Sperren blockiert werden. Leseoperationen erhalten jederzeit eine aktuelle Version des benötigten Datenobjekts, auch wenn zeitgleich eine neue Version des gelesenen Dokuments erstellt wird. Einfüge-Operationen neuer Dokumentversionen werden seriell ausgeführt [9, S.234]. Werden zwei widersprüchliche Versionen eines Dokuments hintereinander eingefügt (Lost-Update), erhalten Leseoperationen zunächst immer die jüngere Version als Ergebnis zurück. Die ältere Version geht dabei nicht verloren, sondern kann bei Bedarf mittels zusätzlicher Funktionalitäten auf Applikationsebene wiederhergestellt werden.

Durch die mangelnde Unterstützung von Transaktionen müssen mögliche Anomalien applikationsseitig verhindert beziehungsweise aufgelöst werden. Darüber hinaus können Integritätsbedingungen nur bei CouchDB und auch nur in einem begrenzten Umfang formuliert und überprüft werden, weshalb die Sicherung der Konsistenz ebenfalls auf Applikationsebene sichergestellt werden muss.

4.4 Verteilte Datenhaltung

MongoDB und CouchDB unterscheiden sich nicht nur in ihrer Anfrageverarbeitung und ihren Synchronisationsmechanismen, auch in der verteilten Datenhaltung verwenden beide Systeme unterschiedliche Konzepte. Diese Unterschiede werden im Folgenden näher vorgestellt.

4.4.1 Replikation

CouchDB verwendet eine Master-Slave-Replikation, bei der ein Master-Server neue, geänderte oder gelöschte Dokumente an einen Slave-Server überträgt. Benutzer können die Replikation über eine HTTP-Anfrage steuern, bei der genau ein Master und genau ein Slave sowie die Dauer der Replikation (einmalig oder permanent) definiert werden. CouchDB ist in der Lage, bidirektionale Replikationen (Master-Master-Replikationen) zu unterstützen, indem zwei unidirektionale Replikationen in beide Richtungen definiert werden [9, S.149]. Auf diese Weise können nicht nur Lese-, sondern auch Schreiboperationen auf allen Knoten durchgeführt werden. CouchDB verwendet zur Synchronisation eine Sequenznummer pro Datenbank sowie eine Änderungshistorie. Durch einen Vergleich der Sequenznummer des Master-Servers und des Slave-Servers kann CouchDB äußerst effizient verpasste Änderungen identifizieren und diese anschließend auf dem Slave-Server einspielen. Des Weiteren ermöglicht dieses Konzept einen effizienten Umgang mit Replikationsfehlern. Wird ein Replikationsvorgang infolge eines Fehlers unterbrochen, wird er dank der Sequenznummern an genau dieser Stelle zu einem späteren Zeitpunkt fortgeführt [9, S.150].

MongoDB verwendet ebenfalls eine unidirektionale Replikation. Hierzu speichern Master-Server alle Schreiboperationen in Log-Dateien und reichen diese je nach Konfiguration synchron oder asynchron an einen oder mehrere Slave-Server weiter. Eine bidirektionale Replikation wie in CouchDB wird nicht unterstützt, weshalb Schreiboperationen ausschließlich auf dem Master-Server ausgeführt werden können. Die Kombination aus einem Master-Knoten und den zugehörigen Slave-Knoten wird als *Replica Set* bezeichnet. Ist ein Master-Knoten aufgrund eines Fehlers für länger als 10 Sekunden nicht erreichbar, wird automatisch ein Slave-Knoten zum neuen Master-Knoten ernannt [123, S.491].

4.4.2 Fragmentierung und Allokation

CouchDB unterstützt nativ keine Verteilung der Dokumente auf unterschiedliche Verarbeitungsrechner. Ist eine Clusterlösung von CouchDB erwünscht, müssen Drittanbieterprodukte wie *BigCouch*[39] oder Erweiterungen wie *Lounge*[40] oder *Pillow*[41] eingesetzt werden. Hierbei handelt es sich um selbstständige Applikationen, welche Dokumente mithilfe von Consistent Hashing Algorithmen auf verschiedene CouchDB Instanzen verteilen [88, 103]. Da Consistent Hashing bereits ausführlich in Kapitel 3.4.2 vorgestellt wurde, wird an dieser Stelle nicht näher auf die Verteilung von CouchDB eingegangen.

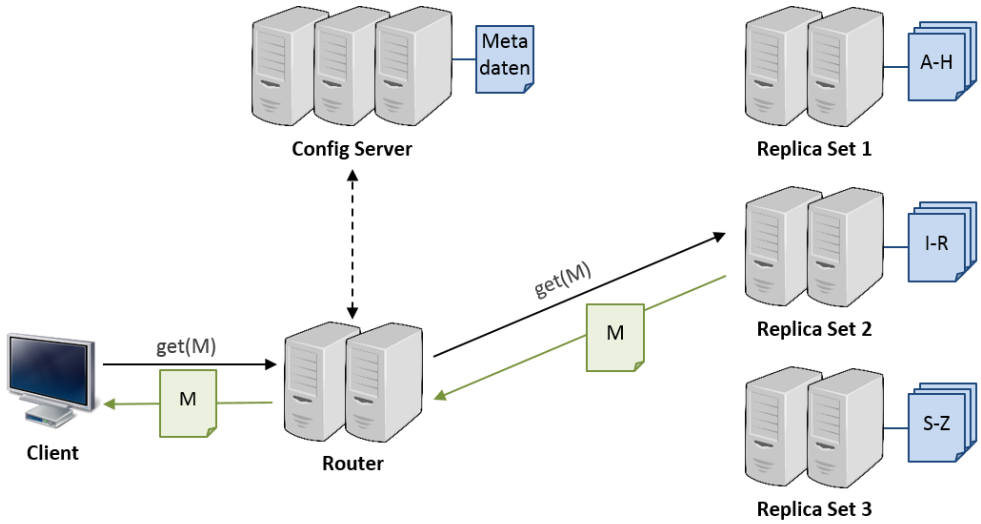


Abbildung 4.17: Abbildung eines MongoDB-Clusters

In Abbildung 4.17 ist die Architektur eines MongoDB-Clusters abgebildet. Dieser wird im Gegensatz zu einem CouchDB-Lounge-Cluster zentral koordiniert. Innerhalb eines MongoDB-Clusters werden drei unterschiedliche Serverarten voneinander unterschieden: Replica Sets, Router und Config Server [123, S.595].

Replica Sets wurden bereits im vorherigen Unterkapitel vorgestellt. Sie bestehen aus einem oder mehreren Knoten, welche die zu verarbeitenden Dokumente speichern. Replica Sets besitzen durch die mehrfache Replikation eine hohe Verfügbarkeit.

Router werden auch als Mongos bezeichnet und bilden die Schnittstelle für clientseitige Anfragen. Sie verarbeiten und leiten Anfragen an die zuständigen Replica Sets weiter und geben schließlich die Ergebnisse an den entsprechenden Client zurück. Ein

[39] http://bigcouch.cloudant.com
[40] http://tilgovi.github.io/couchdb-lounge
[41] https://github.com/khellan/Pillow

MongoDB-Cluster kann aus Gründen der Lastverteilung mehrere Query Router enthalten. Fällt ein Router aus, können andere einspringen. Router laufen meistens auf demselben System wie der Applikationsserver.

Config Server verwalten Metadaten über die Datenverteilung. Ein Produktivsystem besitzt aus Sicherheitsgründen immer drei Config Server [123, S.595]. Fällt ein Config Server aus, ist für diesen Zeitraum keine Verteilungsreorganisation von Dokumenten möglich.

Die Verteilung von Dokumenten auf verschiedene Replica Sets ist von einem sogenannten Verteilungsschlüssel (Shard Key) abhängig. Bei diesem handelt es sich um ein frei wählbares Attribut, das alle Dokumente einer Collection zwangsläufig beinhalten müssen. Abhängig vom Wert dieses Attributs wird ein Dokument entweder per Hashfunktion oder per Wertebereich einem Replica Set zugeordnet. Während die Vorteile der Hashfunktion in der einfachen und gleichmäßigen Verteilung liegen, bietet die Verteilung über Wertebereiche effizientere Bereichsanfragen, da zusammenhängende Datensätze in der Regel im gleichen Replica Set gespeichert werden [123, S.596].

4.4.3 Verteilte Anfrageverarbeitung

Zur Verarbeitung der verteilten Dokumente stellt CouchDB Lounge die Anfragekomponenten Dumbproxy und Smartproxy zur Verfügung [9, S.165]. Der Dumbproxy ist für die Verarbeitung der einfachen REST-Anfragen `get`, `put` und `delete` zuständig. Diese Operationen benötigen außer der zuständigen Serveradresse keine weiteren Informationen und können darum im Rahmen der Anfrageverarbeitung des Consistent Hashings ausgeführt werden.

Der Smartproxy ist für die korrekte Ausführung von View-Anfragen zuständig. Diese verursachen in verteilten Systemen einen wesentlich höheren Aufwand, weil Views auf jedem Dokument im Cluster ausgeführt werden müssen. Views basieren auf dem von Google vorgestellten MapReduce-Konzept [44], bei dem die Datenverarbeitung in eine Map-Phase und eine Reduce-Phase unterteilt werden kann. Der Vorteil dieses Datenverarbeitungs-Paradigmas liegt in seinen sehr guten Parallelisierungseigenschaften. Da die einzelnen Funktionen jeder Phase parallel auf mehreren Servern verarbeitet werden können, wird eine deutliche Reduzierung der Ausführungszeit einer Anfrage ermöglicht.

Abbildung 4.18 beschreibt den Datenfluss einer MapReduce-Anfrage in einem auf vier Servern verteilten Datenbanksystem. Zunächst wird in einem ersten Schritt die Map-Funktion einer View auf jedem Server ausgeführt und von jedem Server eine Ergebnismenge generiert. Diese Ergebnismengen werden in der Reduce-Phase aggregiert. Reduce-Funktionen können mehrmals hintereinander ausgeführt und die Ausführungszeit der Anfrage durch Parallelisierung reduziert werden. Am Ende der Reduce-Phase wird das Ergebnis einer View ausgegeben.

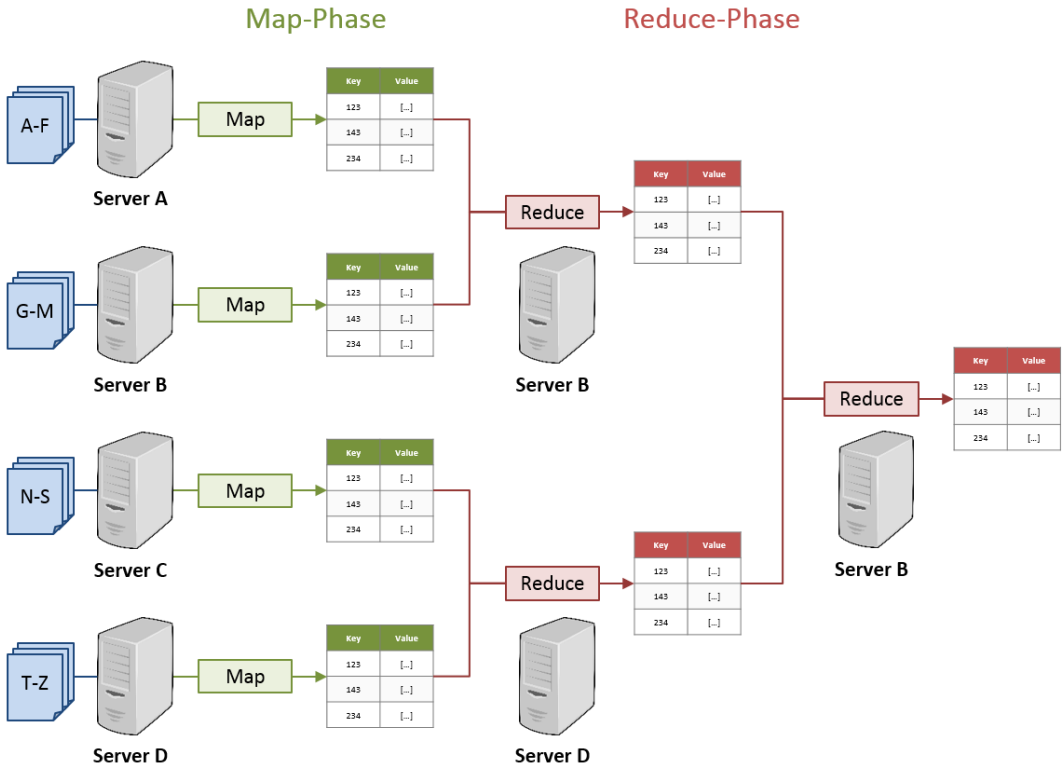


Abbildung 4.18: Ablauf einer verteilten MapReduce-Anfrage

Auch MongoDB ermöglicht die Ausführung von MapReduce-Programmen [123, S.382]. In Anbetracht der vergleichsweise hohen Ausführungskosten von MapReduce-Programmen sowie den sehr ausdrucksstarken Schnittstellen von MongoDB, werden MapReduce-Programme im Gegensatz zu CouchDB nur für die Analyse von sehr großen Datenmengen eingesetzt. Die Verarbeitung von gewöhnlichen Anfragen wird in einem MongoDB-Cluster von Routern übernommen. Entscheidend für die Verarbeitungsgeschwindigkeit einer Anfrage ist der Verteilungsschlüssel. Basiert die Anfrage auf diesem Schlüssel, können die benötigten Dokumente direkt an das zuständige Replica Set adressiert werden. Wird eine Anfrage ohne diesen Schlüssel ausgeführt, müssen sämtliche Replica Sets zeitaufwendig durchsucht werden. Idealerweise handelt es sich bei dem Verteilungsschlüssel um ein häufig adressiertes Attribut, das außerdem von allen performanzkritischen Anfragen verwendet wird [123, S.608].

4.4.4 Verteilte Transaktionsverwaltung

Auch wenn Document Stores keine Transaktionen im eigentlichen Sinn bereitstellen, müssen sie die sichere Ausführung einzelner Operationen gewährleisten. MongoDB und CouchDB setzen verschiedene Techniken ein, um Fehler in verteilten Systemen wie Nachrichtenverluste, Netzwerkfehler oder Rechnerausfälle zu behandeln. Darüber hinaus müssen konkurrierende Schreiboperationen über mehrere Rechner hinweg synchronisiert werden.

Recovery

Da Document Stores keine Transaktionen unterstützen und alle Operationen auf einem Dokument bereits atomar sind, verbleibt für die Recovery-Komponenten lediglich die Sicherung der Dauerhaftigkeit der durchgeführten Operationen. In verteilten Systemen wird hiermit primär der Schutz vor Hardware- und Katastrophenfehlern verstanden, bei denen einzelne Server oder ganze Rechenzentren zerstört werden. Wie bereits in den vorherigen Kapiteln erwähnt, steht die Sicherung der Dauerhaftigkeit in Konflikt mit der Verarbeitungsgeschwindigkeit von Operationen. Aus diesem Grund muss ein Kompromiss zwischen diesen beiden Größen gefunden werden.

CouchDB aktualisiert Replikate asynchron [9, S.149] und weist deswegen bei Schreiboperationen keine zusätzliche Latenz auf. Tritt nach der Ausführung einer Operation ein Hardwarefehler auf und wurden die durchgeführten Änderungen noch nicht repliziert, sind die Daten verloren. MongoDB ermöglicht es Clients, bei jeder Änderungsoperation sogenannte Write Concerns [123, S.67] zu definieren. Diese bestimmen den Grad an Dauerhaftigkeit, der bei der jeweiligen Operation garantiert werden soll. Der Wertebereich der Write Concerns reicht dabei von Änderungen im Hauptspeicher des Master-Knotens bis hin zu persistierten Operationen auf allen Slave-Knoten. So kann abhängig vom Anwendungsfall ein Kompromiss zwischen hoher Verarbeitungsgeschwindigkeit oder hoher Datensicherheit definiert werden.

Synchronisation

Konkurrierende Schreiboperationen werden in MongoDB unter Zuhilfenahme von exklusiven Schreibsperren synchronisiert [123, S.688]. Da Schreiboperationen nur auf dem Master-Knoten eines Replica Sets durchgeführt werden können, ist dieser für die Verwaltung der Sperren zuständig. Es kann immer nur eine Änderungsoperation pro Dokument ausgeführt werden.

Inkonsistenzen durch konkurrierende Schreibzugriffe oder Systemfehler werden in CouchDB mittels Versionierung behoben. Liegen innerhalb von CouchDB widersprüchliche Versionen eines Dokuments vor, werden diese markiert. Anschließend wird standardmäßig immer die jüngste Version des Dokuments zurückgegeben (Last Write Wins). Die widersprüchlichen Versionen können bei Bedarf auf Applikationsebene mit komplexeren Konfliktauflösungsstrategien oder mithilfe von Anwendern zu einer korrekten Version zusammengeführt und in CouchDB zurückgeschrieben werden [9, S.153].

4.5 Zusammenfassung

Document Stores können als Kompromiss zwischen den leichtgewichtigen Key Value Stores und den mächtigen relationalen Datenbanken aufgefasst werden. Obwohl sie keine zusätzlichen Strukturinformationen über die in ihnen gespeicherten Daten verwalten und demzufolge ebenso wie Key Value Stores schemafrei sind, verfügen sie über ein ähnlich ausdrucksstarkes Datenmodell wie relationale Datenbanken. Der Grund hierfür liegt darin, dass einzelne Datensätze als JSON-Dokumente verwaltet werden. Diese Dokumente enthalten neben den Attributwerten auch Metainformationen wie Attributnamen und Datentypen. Da Document Stores in der Lage sind, diese Informationen zu verarbeiten, können sie einen sehr großen Umfang an Operationen und Operatoren zur Datenverarbeitung bereitstellen. Bis auf Verbundoperationen können sämtliche Anfrageoperationen relationaler Datenbanken in Document Stores ausgeführt werden.

Document Stores unterstützen keine Transaktionen. Jede Operation wird einzeln auf einem Dokument ausgeführt. Durch eine geschickte Datenmodellierung mithilfe von Aggregationen können allerdings zusammengehörige Daten in einem Dokument abgebildet und damit atomar geändert werden. Da jeder Datensatz selbstbeschreibend ist und in Document Stores keine globalen Strukturinformationen verwaltet werden, können bei diesen Systemen standardmäßig keine Integritätsbedingungen formuliert und überprüft werden. Konkurrierende Schreiboperationen werden in MongoDB und CouchDB unterschiedlich synchronisiert. Während MongoDB eine pessimistische Synchronisation mittels Sperren vorzieht, verwendet CouchDB mit MVCC ein optimistisches Synchronisationsverfahren. Alle Änderungsoperationen werden in Document Stores dauerhaft gespeichert.

Document Stores lassen sich sehr leicht auf mehrere Rechner verteilen. Dokumente werden nicht fragmentiert, Verbundoperationen nicht unterstützt und Transaktionen nicht angeboten. Hierdurch kann jedes Dokument atomar auf einen Server verteilt und die Anfragen entsprechend weitergeleitet werden. Während MongoDB dank eines sehr großen Funktionsumfangs den Aufwand einer verteilten Datenhaltung vollständig übernehmen kann, lässt sich CouchDB nur unter Zuhilfenahme von Drittanbieterprodukten verteilen.

Funktionalitäten		MongoDB	CouchDB
Ausdrucksstarkes Datenmodell	Attributnamen	✓	✓
	Datentypen	✓	✓
Anfragemächtigkeit	Relationale Operatoren	✓	✓
	Verbundoperationen	✗	✗
	Sekundäre Zugriffspfade	✓	✓
ACID-Eigenschaften	Atomarität	✗	✗
	Integritätsbedingungen	✗	✗
	Synchronisationsmechanismen	✓	✓
	Dauerhaftigkeit	✓	✓
Verteilte Datenhaltung	Replikation	✓	✓
	Verteilung	✓	✗

Tabelle 4.5: Funktionsumfang von Document Stores

Der hohe Funktionsumfang von Document Stores wird in Tabelle 4.5 verdeutlicht. Hierin sind die wichtigsten Funktionalitäten abgebildet, die für die Datenverarbeitung benötigt werden. Angesichts der hier beschriebenen Mächtigkeit von Document Stores beschränkt sich der Mehraufwand auf Applikationsebene auf einige wenige fehlende Funktionalitäten. Hierzu zählen unteren anderem fehlende Verbundoperationen oder das fehlende Transaktionskonzept. Allerdings lassen sich diese beiden Nachteile durch eine geschickte Datenmodellierung mithilfe der Aggregation kompensieren.

Trotz der teilweise sehr großen Unterschiede in den technischen Eigenschaften von MongoDB und CouchDB sind aufgrund des identischen Datenmodells die in dieser Arbeit getätigten Aussagen zur Datenmodellierung allgemeingültig.

5

COLUMN FAMILY STORES

Kapitelinhalt

- Grundlagen von Column Family Stores
- Datenmodellierung mit Column Families
- Verarbeitung von Schreiboperationen in SSTables

Column Family Stores ähneln in ihrem Erscheinungsbild relationalen Datenbanken, da auch hier einzelne Datensätze mithilfe von Tabellen verarbeitet werden. Im Gegensatz zu relationalen Datenbanken sind Column Family Stores jedoch auf eine verteilte Datenhaltung in sehr großen Clustern ausgelegt. Zu den bekanntesten Vertretern von Column Family Stores zählen *Bigtable* [30], *HBase*[42], *Hypertable*[43] und *Cassandra*[44].

In der Literatur werden die Eigenschaften von Column Family Stores und der Umgang mit diesen Systemen ausschließlich anhand von Anwendungsbeispielen erläutert. Infolge der fehlenden Arbeit auf konzeptueller Ebene existieren derzeit keine allgemeingültigen Aussagen über die Vor- und Nachteile der Datenmodelle und keine allgemeingültigen Regeln, mit denen konzeptuelle Datenmodelle auf die logischen Datenmodelle von Column Family Stores überführt werden können. In dieser Problemstellung liegt die Motivation dieses Kapitels. Durch eine methodische Aufarbeitung der Column Family Stores zugrunde liegenden Konzepte soll ein wertvoller Beitrag zum Verständnis dieser Systeme geleistet werden. Basierend auf dieser konzeptuellen Aufarbeitung und einer umfangreichen Analyse der bereitgestellten Anfragefunktionalitäten werden daraufhin allgemeingültige Modellierungsmuster erstellt, mit deren Hilfe auch unerfahrene Anwender Column Family Stores effizient einsetzen können. Anschließend wird die interne Ebene von Column Family Stores analysiert und dabei auf Speicherstrukturen und die Transaktionsverwaltung eingegangen. Das Kapitel schließt mit einer Behandlung der verteilten Datenhaltung in Column Family Stores.

[42] http://hbase.apache.org
[43] http://hypertable.org
[44] http://cassandra.apache.org

5.1 Einführung

Column Family Stores basieren auf dem Konzept von Googles verteiltem Datenbanksystem Bigtable [30]. Bigtable wurde entwickelt, um die bei Google anfallenden sehr großen Datenmengen effizient verarbeiten zu können. Primäre Anforderungen bei der Entwicklung von Bigtable waren neben einer breiten Einsatzmöglichkeit vor allem lineare horizontale Skalierbarkeit, hohe Performanz und permanente Verfügbarkeit.

Bereits bei der Veröffentlichung des Konzepts wurde Bigtable in über sechzig verschiedenen Google Produkten, wie Google Analytics und Google Earth, eingesetzt. Das prominenteste Einsatzgebiet stellt jedoch die Verwaltung des Google Index dar, auf dem die Google Suche basiert. All diese Anwendungen stellen unterschiedliche Anforderungen an die Datenhaltung. Bigtable muss deshalb in der Lage sein, variabel auf widersprüchliche Parameter, wie latenzsensitive Anfragen, hoher OLTP-Durchsatz oder fehlerresistente Persistenz, reagieren zu können. Bigtable kann über mehrere zehntausend Rechner hinweg skaliert werden.

Bei Bigtable handelt es sich um ein proprietäres Datenbanksystem, das ausschließlich von Google eingesetzt wird. Basierend auf der Veröffentlichung des Bigtable-Konzepts entstanden ab 2006 eine Reihe weiterer verteilter Datenbanksysteme, die heute unter den Namen Column Family Stores, Extensible Record Stores oder Wide Column Stores bekannt sind. Während HBase und Hypertable direkte Nachimplementierungen des Bigtable-Konzepts sind, handelt es sich bei Cassandra um ein Hybridsystem, das in vielen Punkten von Bigtable abweicht und vor allem auf interner Ebene sehr Amazons Dynamo ähnelt.

5.2 Konzeptuelle Ebene

Column Family Stores ähneln in ihrem Erscheinungsbild relationalen Datenbanken. Datensätze werden in Tabellen verwaltet und können in manchen Systemen über eine SQL-ähnliche, deklarative Anfragesprache ausgelesen und manipuliert werden. Im Gegensatz zu relationalen Datenbanken sind Column Family Stores jedoch praktisch schemafrei und bieten daher eine sehr große Flexibilität gerade im Umgang mit heterogenen Daten.

5.2.1 Datenmodell

Ähnlich wie in relationalen Datenbanken erfolgt die Datenverarbeitung in Column Family Stores mithilfe von Tabellen. Einzelne Datensätze besitzen einen eindeutigen Primärschlüssel (Row Key) und werden zeilenweise abgebildet. Konkrete Attribute und ihre zugehörigen Werte werden für jeden Datensatz explizit in Form von Schlüssel-

Wert-Paaren dargestellt. Ähnliche oder zusammengehörige Attribute können spaltenweise zu sogenannten *Column Families* gruppiert werden. Column Families sind ein zusätzliches Strukturelement, für das es auf Seiten von relationalen Datenbanken kein Pendant gibt. Ein Datensatz kann für jede Column Family eine beliebige Anzahl von Schlüssel-Wert-Paaren besitzen. Die entsprechenden Strukturbezeichnungen sind in Tabelle 5.1 dargestellt.

Relationale Datenbank	Column Family Stores
Tabelle	Tabelle
Tupel (Datensatz)	Row
Primärschlüssel (ID)	Row Key
-	Column Family
Attribut	Schlüssel-Wert-Paar

Tabelle 5.1: Gegenüberstellung der unterschiedlichen Strukturbezeichnungen

Primärschlüssel, Spaltennamen und Attributnamen werden als Strings verarbeitet und in alphabetischer Reihenfolge im Datenbanksystem abgelegt. Die konkreten Werte eines Attributs werden als Byte-Arrays gespeichert, sodass sich sämtliche primitive Datentypen sowie komplexe Objekte und sogar Bilder in Column Family Stores speichern lassen. Column Family Stores sind, ähnlich wie Key Value Stores, nicht in der Lage, diese Byte-Arrays zu interpretieren. Aus diesem Grund werden Datentypen und auch Fremdschlüsselbeziehungen nicht vom Datenbanksystem unterstützt. Es ist somit Aufgabe der Applikationslogik, Attributwerte bei der Auswertung zu parsen und Referenzen auf Row Keys durch weitere Anfragen aufzulösen.

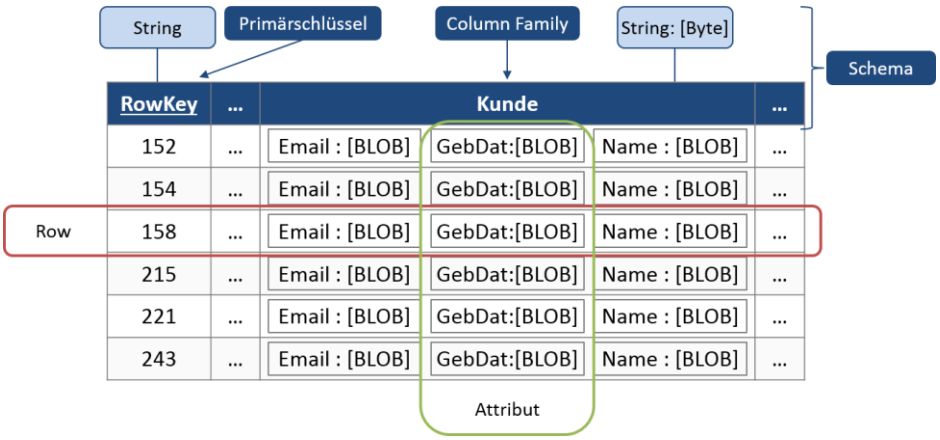


Abbildung 5.1: Logisches Datenmodell der Entität *Kunde*

Abbildung 5.1 veranschaulicht die Abbildung der bereits aus den vorherigen Kapiteln bekannten Entität *Kunde* (siehe Abbildung 3.5) auf das logische Datenmodell eines Column Family Stores. Die abgebildete Tabelle enthält mehrere Column Families, wobei in diesem Beispiel ausschließlich die Column Family *Kunde* explizit dargestellt ist. Diese besitzt für jeden Datensatz (Row) eine alphabetisch sortierte Menge von Schlüssel-Wert-Paaren. Jedes Schlüssel-Wert-Paar repräsentiert ein Attribut und dessen Wert.

Auch wenn die Daten in diesem Beispiel sehr homogen sind, kann theoretisch jeder Datensatz in einer Column Family eine beliebige Anzahl von Schlüssel-Wert-Paaren verwalten. Insgesamt können innerhalb einer Tabelle für jeden Datensatz bis zu zwei Milliarden Schlüssel-Wert-Paare gespeichert werden [39, S.57], weshalb Tabellen in Column Family Stores nicht selten mehr Attribute als Datensätze aufweisen. Column Family Stores werden darum auch als Wide Column Stores bezeichnet und eignen sich vor allem für die Verarbeitung sehr großer Datenmengen an heterogenen Daten. Je nach Datenbanksystem können pro Tabelle drei bis 255 Column Families angelegt werden [11, 89].

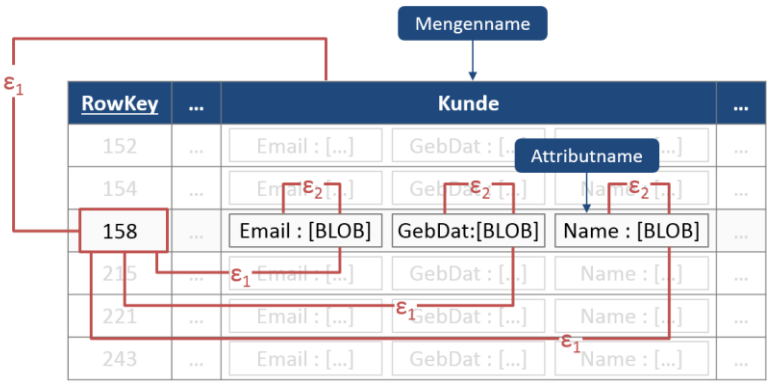


Abbildung 5.2: Prädikatorenschema der Column Family *Kunde*

In Abbildung 5.2 ist das Prädikatorenschema der Column Family *Kunde* dargestellt. Bei näherer Betrachtung wird deutlich, dass außer den Namen der Column Families keine weiteren Schemainformationen verwaltet werden. Column Family Stores sind demzufolge praktisch *schemafrei*. Allerdings muss eine Column Family zuvor angelegt werden, bevor Daten in sie geschrieben werden können. Auch wenn sich die konzeptuellen Datenmodelle deutlich voneinander unterscheiden, ähneln sich die Prädikatorenschemata von Document Stores und Column Family Stores in ihrer Ausdrucksstärke. In beiden Systemen können sämtliche Prädikatoren erster und zweiter Ordnung abgebildet werden, weil innerhalb der Datensätze neben den Attributwerten auch die für die Interpretation notwendigen Attributnamen gespeichert werden. Dank dieser *selbstbeschreibenden* Datensätze besitzen Column Family Stores ebenso wie Document Stores ein lokales Schema. Im Gegensatz zu relationalen Datenbanken bilden Tabellen von Column Family Stores den Namen des Primärschlüssels nicht explizit ab. Ist diese

Prädikation erster Ordnung notwendig, muss diese Beziehung mithilfe eines zusätzlichen Schlüssel-Wert-Paares modelliert werden.

Datenintegrität

Da Column Family Stores außer dem Column Family Namen keine weiteren Metainformationen speichern und nicht in der Lage sind, Attributwerte zu interpretieren, können sie auch keine statischen und dynamischen Regeln zur Sicherung der Datenintegrität überprüfen. Selbst Eindeutigkeitsprüfungen des Primärschlüssels werden bei Einfüge-Operationen neuer Datensätze nicht durchgeführt. Im Fall eines Primärschlüsselkonflikts mit einem bestehenden Datensatz wird der alte Datensatz durch den neuen Datensatz ohne Vorwarnung überschrieben.

Der Column Family Store Cassandra stellt hierbei eine Ausnahme dar. Im Gegensatz zu anderen Systemen können hier für bestimmte Attribute optional Wertebereiche definiert werden. Diese Wertebereiche können bei Schreiboperationen mittels eines in der Schnittstelle befindlichen Validators überprüft werden [40].

5.2.2 Anfrageverarbeitung

Die Interaktion mit Column Family Stores kann über verschiedene Schnittstellen erfolgen. Zu den am häufigsten verwendeten Schnittstellen gehören Konsolenbefehle, Bibliotheken und deskriptive Anfragesprachen. Während Konsolenbefehle nur eine sehr einfache Interaktion basierend auf den drei Basisoperationen `get`, `put` und `delete` ermöglichen, bieten Bibliotheken nicht nur einen größeren Funktionsumfang, sondern auch eine wesentlich angenehmere Formulierung von Anfragen. Hypertable und Cassandra stellen Anfragesprachen mit SQL-ähnlicher Syntax bereit, welche das zugrunde liegende Datenmodell weiter abstrahieren und die Anfrageformulierung hierdurch vereinfachen [42, 89].

Column Family Stores unterstützen nur das Einfügen oder das Löschen von Datensätzen in Tabellen. Soll ein Datensatz aktualisiert werden, wird der alte Datensatz überschrieben beziehungsweise eine neue Version des Datensatzes angelegt. Schreiboperationen basieren auf folgender Syntax:

```
put <TabellenName>, <RowKey>, <ColumnFamily:Column>, <Wert>.
```

Abhängig vom Datenbanksystem und der verwendeten Schnittstelle können pro Operation ein oder mehrere Datensätze verarbeitet werden. Datensätze können zudem mit einem Zeitstempel (Time To Live – TTL) versehen werden. Nach Ablauf der Zeit wird der Datensatz automatisch entfernt [11, Kap.6.7].

Leseanfragen folgen in Column Family Stores immer einem gleichen Muster, egal welche Schnittstelle zur Interaktion verwendet wird. Sie enthalten Restriktions- und Projektionsbedingungen und werden auf einer Tabelle ausgeführt:

```
get <TabellenName>, <Restriktion>, <Projektion>.
```

Die Angabe von Projektionsbedingungen ist optional. Projektionsbedingungen bestehen aus dem Namen der gewünschten Column Family und einem regulären Ausdruck. Dieser schränkt die Ergebnismenge der Attribute (Columns) ein, die in der entsprechenden Column Family enthalten sind:

```
ColumnFamilyName:ColumnName.
```

Die Angabe mindestens einer Restriktionsbedingung ist dagegen vorgeschrieben. Restriktionsbedingungen können mithilfe von regulären Ausdrücken und mehreren Vergleichsoperationen formuliert werden. Diese Bedingungen lassen sich jedoch ausschließlich auf dem Primärschlüssel einer Tabelle anwenden. Eine Restriktion der Ergebnismenge auf das Geburtsjahr *1984* ist daher auf der in Abbildung 5.1 dargestellten Tabelle in dieser Form nicht möglich. Die in den Folgekapiteln vorgestellte Indizierung sowie eine anfragespezifizierte Datenmodellierung können jedoch behilflich sein, um solche Anfragen dennoch zu ermöglichen. Abbildung 5.3 beschreibt die Ausgabe der Kundennamen, deren Schlüssel kleiner oder gleich *158* sind. Hierzu wurde die Syntax der in Cassandra und Hyptertable verfügbaren Anfragesprachen herangezogen.

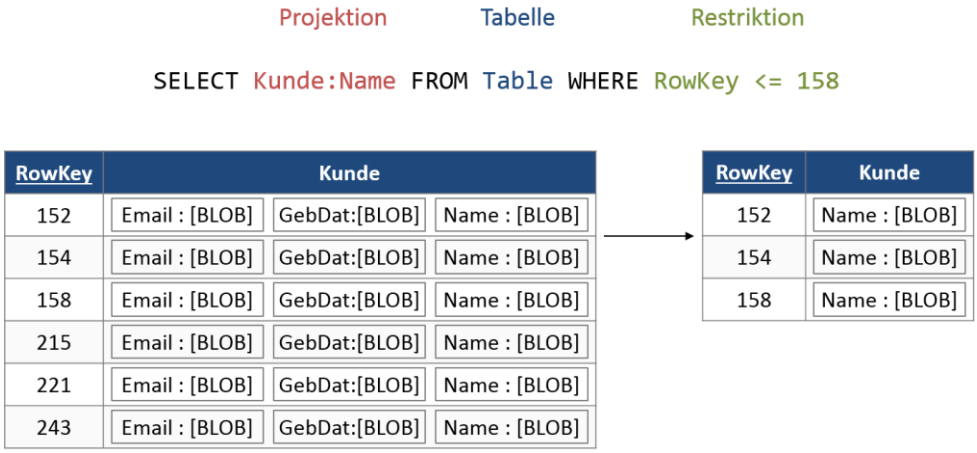


Abbildung 5.3: Formulierung von Restriktions- und Projektionsbedingungen

Column Family Stores unterstützen keine Verbundoperationen und besitzen deswegen auch keine relational vollständige Schnittstelle. Dennoch lassen sich dank der vielen Operationen und der regulären Ausdrücke mächtige Anfragen formulieren (Tabelle 5.2).

Relationale Algebra	Column Family Stores
Projektion	✓
Restriktion	✓
Kartesisches Produkt	✗
Vereinigung	✗
Differenz	✗
Umbenennung	✗
Operator	**Column Family Stores**
Gleich	✓
Ungleich	✗
Größer, Kleiner	✓
Like	✓
In	✓
Exists	✗
And, Or	✓
Not	✗

Tabelle 5.2: Unterstützte relationale Operationen und Operatoren

5.2.3 Datenmodellierung

Column Family Stores fehlt es an allgemeingültigen Abbildungsregeln und Modellierungsprozessen, mit denen sich Elemente eines konzeptuellen Datenmodells systematisch auf die Elemente des logischen Datenmodells überführen lassen. Entwicklern mit einer geringen Erfahrung im Umgang mit Column Family Stores stehen deshalb während des anspruchsvollen Prozesses der Datenmodellierung nur die wenigen bisher veröffentlichten Anwendungsbeispiele aus der Praxis als Hilfestellung zur Verfügung. Das Problem dabei ist, dass sich aus diesen minimalistischen und hochgradig spezialisierten Anwendungsbeispielen keine Rückschlüsse auf Allgemeingültigkeit ziehen lassen. Um fachlich fundierte Aussagen über die Vor- und Nachteile verschiedener Systeme und deren Modellierungsmuster treffen zu können, sind anwendungsfallübergreifende Abbildungsregeln zwingend erforderlich. In dieser Problemstellung liegt die Motivation dieses Kapitels. Basierend auf den Ergebnissen der bereits erfolgten konzeptuellen Aufarbeitung sowie der Analyse der verfügbaren Operationen und Operatoren werden im Folgenden allgemeingültige Abbildungsregeln entwickelt und anschließend bewertet. Abschließend wird ein Modellierungsprozess vorgestellt, mit dessen Hilfe sich die erstellten Abbildungsregeln systematisch ausführen lassen.

Entitäten

Column Family Stores verfügen über ein sehr flexibles und zugleich auch sehr ausdrucksstarkes Datenmodell. Dank der Schemafreiheit dieser Systeme unterliegt die Abbildung einzelner Datensätze in einer Tabelle keinen strukturellen Restriktionen seitens eines globalen Datenbankschemas. Des Weiteren ermöglicht die implizite Speicherung der Schemainformationen in den Datensätzen ein hohes Maß an Ausdrucksstärke, sodass sich alle Prädikatoren erster und zweiter Ordnung abbilden lassen. Entitäten des konzeptuellen Datenmodells werden in Column Family Stores auf jeweils eine Tabelle abgebildet. Die Attribute einer Entität werden als Schlüssel-Wert-Paare hinterlegt. Da diese als Byte-Arrays gespeichert werden, muss die entsprechende Typisierung auf Applikationsebene erfolgen. Column Family Stores weisen durch das Konzept der Column Families eine Modellierungskomponente mehr als relationale Datenbanken auf, wodurch sich zusätzliche Möglichkeiten bei der Datenmodellierung ergeben. So können beispielsweise mehrere zusammengehörige Attribute zu einer Gruppe zusammengefasst und diese dadurch leichter verarbeitet werden.

Der Nachteil von Column Family Stores liegt in deren geringer Anfragemächtigkeit. Zwar stellen diese Systeme viele der von relationalen Datenbanken bekannten Operatoren zur Verfügung, doch lassen sich die formulierten Restriktionsbedingungen ausschließlich auf den Primärschlüsseln einer Tabelle ausführen. Wird die in Abbildung 5.4 dargestellte Entität *Kunde* wie beschrieben als eine Tabelle abgebildet und die Kundennummer als Primärschlüssel herangezogen, ist daraufhin beispielsweise eine Suche nach allen Kunden eines bestimmten Geburtsjahres nicht möglich.

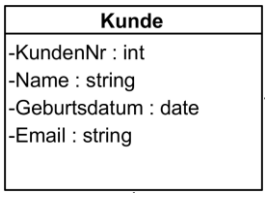


Abbildung 5.4: Konzeptuelle Darstellung der Entität *Kunde*

Aufgrund der hohen Abhängigkeit der Anfragemächtigkeit von der Datenmodellierung, stellen Identifikation, Analyse und Priorisierung aller zu unterstützenden Anfragen die ersten Schritte in der Abbildung einer Entität auf eine Tabelle dar. Basierend hierauf muss anschließend der Primärschlüssel der Tabelle so modelliert werden, dass er alle Attribute beinhaltet, die in den Restriktionsbedingungen der Anfragen benötigt werden. Um die oben erwähnte Restriktion auf das Geburtsjahr ausführen zu können, muss daher dieses Attribut Teil des Primärschlüssels werden:

```
<Kundennummer>:<Geburtsjahr>.
```

Das Entscheidende bei der Modellierung eines zusammengesetzten Schlüssels ist dessen Aufbau. Column Family Stores speichern einzelne Datensätze in alphabetischer Reihenfolge ihrer Primärschlüssel ab. Um Bereichsanfragen auf ein Attribut möglichst effizient unterstützen zu können, muss darum der erste Teil des Primärschlüssels aus dem Namen dieses Attributs bestehen. Um nicht alle Schlüssel einer Tabelle durchlaufen und die gesuchten Datensätze von verschiedenen Speicherplätzen auslesen zu müssen, wäre demzufolge für Anfragen, die das Geburtsjahr in ihren Restriktionsbedingungen enthalten, die folgende Primärschlüssel-Zusammensetzung günstig:

`<Geburtsjahr>:<Kundennummer>`.

Durch die deutlich geringere alphabetische Distanz der Primärschlüssel befinden sich die gesuchten Datensätze mit hoher Wahrscheinlichkeit im selben oder in benachbarten Speicherbereichen und können infolgedessen wesentlich effizienter ausgelesen werden. Für das Auslesen aller Namen der Kunden mit dem Geburtsjahr *1984* ist im Fall dieser Modellierung nur ein einziger Datenbankaufruf notwendig. Abgesehen von der Transformation der Ergebnisse von deren Byte-Array-Datenstruktur in die gewünschten Datenformate, fällt auf Applikationsebene kein weiterer Aufwand an. Die in Abbildung 5.5 noch ausführlich dargestellte Transformation der Datenformate wird in den folgenden Beispielen des Kapitels nicht mehr explizit erwähnt.

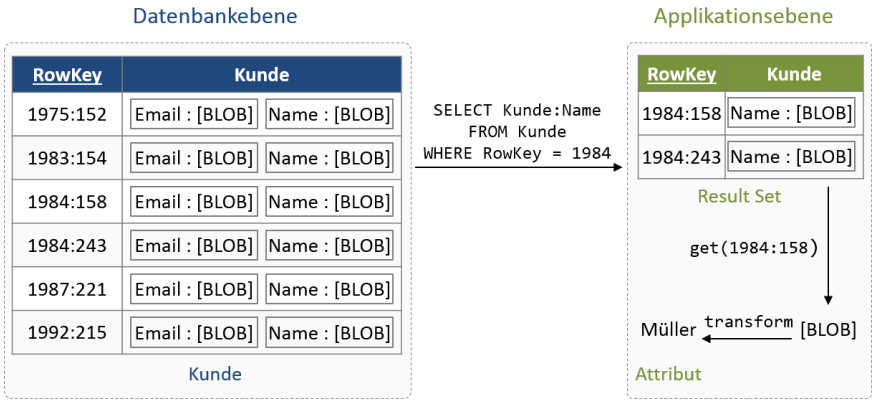


Abbildung 5.5: Restriktion mithilfe eines zusammengesetzten Schlüssels

Mithilfe einer geschickten Modellierung des Primärschlüssels können Tabellen auf die performante Ausführung bestimmter Anfragen ausgerichtet werden. Dennoch müssen häufig auch Anfragen auf Nicht-Schlüsselattribute unterstützt werden. Da die meisten Column Family Stores keine oder erst seit kurzem Indizes anbieten, müssen diese manuell in Form von Index-Tabellen realisiert werden. In Index-Tabellen werden die für eine Anfrage benötigten Nicht-Schlüsselattribute als Primärschlüssel deklariert und die entsprechenden Datensätze der Haupttabelle referenziert.

Um eine Suche auf dem Nicht-Schlüsselattribut *Jahrgang* in der Tabelle *Kunde* zu ermöglichen, bietet sich die in Abbildung 5.6 dargestellte Index-Tabelle an. Diese gibt für einen Jahrgang eine Menge von Schlüssel-Wert-Paaren zurück. Der Schlüssel eines Schlüssel-Wert-Paares enthält den Fremdschlüssel auf die Tabelle *Kunde*. Der Wert eines Schlüssel-Wert-Paares wird nicht benötigt und kann in diesem Fall leer bleiben. Da Column Family Stores keine Verbundoperationen bereitstellen, müssen diese Schlüssel applikationsseitig aus der Ergebnismenge ausgelesen werden. Anschließend muss pro Schlüssel eine weitere Datenbankanfrage gestartet werden, um die zu den Schlüsseln zugehörigen Datensätze zu erhalten.

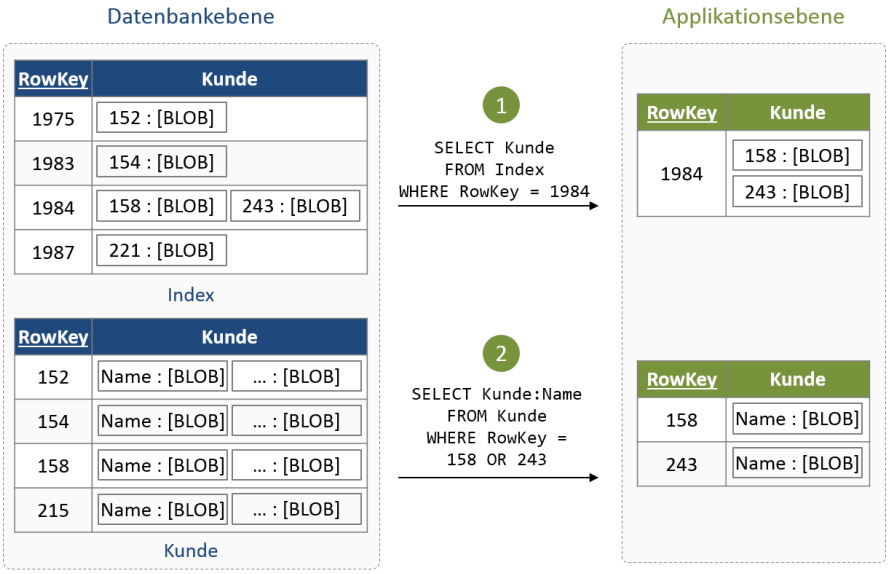


Abbildung 5.6: Restriktion mithilfe einer Index-Tabelle

Weil jedes Schlüssel-Wert-Paar in das gewünschte Format überführt werden muss, entsteht, ähnlich wie bei Key Value Stores, ein Mehraufwand auf Applikationsebene. Allerdings ergibt sich durch die fehlende Typisierung auf Datenbankseite wiederum der Vorteil, dass komposite und mehrwertige Attribute in Maps, Listen oder Arrays gespeichert werden können. Alternativ können diese Attribute auch auf mehrere einzelne Schlüssel-Wert-Paare aufgeteilt und optional in einer Column Family zusammengefasst werden.

Da Column Family Stores kein globales Schema enthalten, sondern selbstbeschreibende Datensätze verwalten, können sich die in einer Tabelle befindlichen Datensätze in ihrer Struktur voneinander unterscheiden. Eine wie in der relationalen Datenmodellierung notwendige Abbildung von generalisierten und spezialisierten Entitäten auf verschiedene Tabellen ist daher nicht notwendig.

Beziehungen

Column Family Stores unterstützen keine Verbundoperationen und sind deshalb nicht in der Lage, Fremdschlüsselbeziehungen datenbankseitig aufzulösen. Allerdings verfügen diese Systeme mit den Column Families über ein zusätzliches Strukturelement in ihrem Datenmodel, mit denen sich nicht nur neue Möglichkeiten bei der Abbildung von Entitäten, sondern auch von Beziehungen ergeben. Diese Modellierungsmöglichkeiten können gemäß der aus den vorherigen Kapiteln bekannten Klassifizierung in Referenzierung, Denormalisierung und Aggregation unterteilt werden. Im Folgenden werden die Vor- und Nachteile der verschiedenen Modellierungsvarianten systematisch herausgestellt und basierend darauf allgemeingültige Aussagen zur bestmöglichen Abbildung von Beziehungen in Column Family Stores abgeleitet. Zur Veranschaulichung der verschiedenen Techniken wird das in Abbildung 5.7 dargestellte logische Datenmodell eines Onlineshops herangezogen.

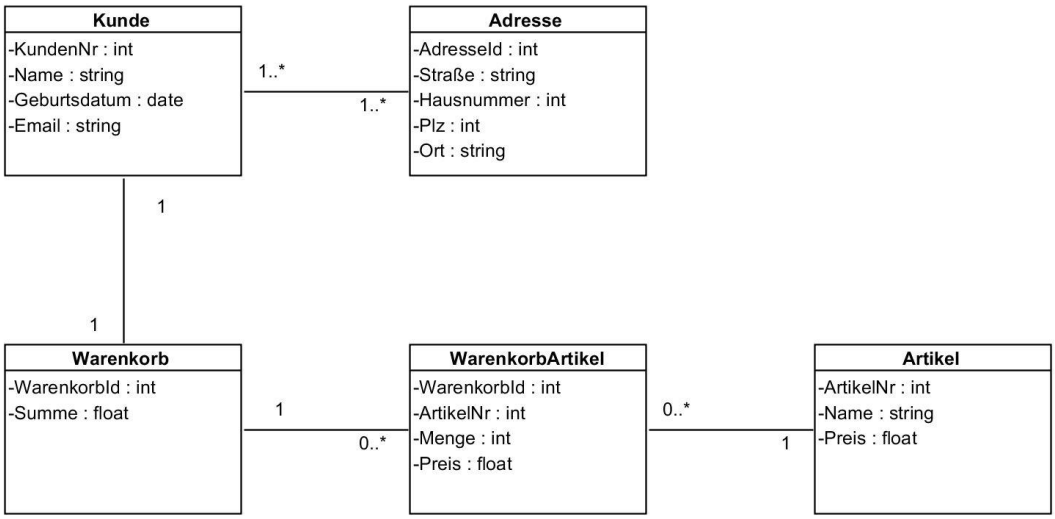


Abbildung 5.7: Konzeptuelles Datenmodell eines Online-Shops

Referenzierung: Ein Schlüssel-Wert-Paar kann auf den RowKey eines anderen Datensatzes verweisen und diesen somit referenzieren. Da Column Family Stores jedoch keine Verbundoperationen und keine Techniken zur Sicherung der referenziellen Integrität bereitstellen, muss die Auflösung und Wartung der Referenzen auf Applikationsebene erfolgen. Hierdurch kann gerade bei einer hohen Anzahl an Referenzen ein erheblicher Mehraufwand entstehen.

Ein weiterer Nachteil der Referenzierung ergibt sich aus der eingeschränkten Anfrageverarbeitung. Da Restriktionsbedingungen ausschließlich den Primärschlüssel enthalten dürfen und sekundäre Zugriffspfade in der Regel nicht angeboten werden, können keine Anfragen auf Fremdschlüssel unterstützt werden. Aus diesem Grund können Referenzen ausschließlich unidirektional von der Entität aus aufgelöst werden, welche die entsprechenden Fremdschlüssel beinhaltet. Während 1:1-Beziehungen

problemlos auf einer beliebigen Seite der Beziehungen verwaltet werden können, gestaltet sich die Abbildung von mehrwertigen Beziehungen aufgrund der Vielzahl an Referenzen deutlich anspruchsvoller. In diesem speziellen Modellierungsfall kann das Konzept der Column Families herangezogen werden. Da jeder Datensatz eine beliebige Anzahl unterschiedlicher Schlüssel-Wert-Paare enthalten kann und sich mehrere zusammengehörige Schlüssel-Wert-Paare zu einer Column Family gruppieren lassen, bietet sich die Abbildung aller Referenzen einer Entität in einer separaten Column Family an. Durch dieses Modellierungsmuster lassen sich auch mehrwertige Beziehungen problemlos auf einer beliebigen Seite der Beziehung verwalten. Ist eine bidirektionale Auflösung einer Fremdschlüsselbeziehung erforderlich, muss diese redundant in beiden beteiligten Entitäten abgebildet werden. Besteht zwischen zwei Entitäten eine M:N-Beziehung, muss diese mithilfe zweier entgegen gerichteter 1:N-Beziehungen modelliert werden.

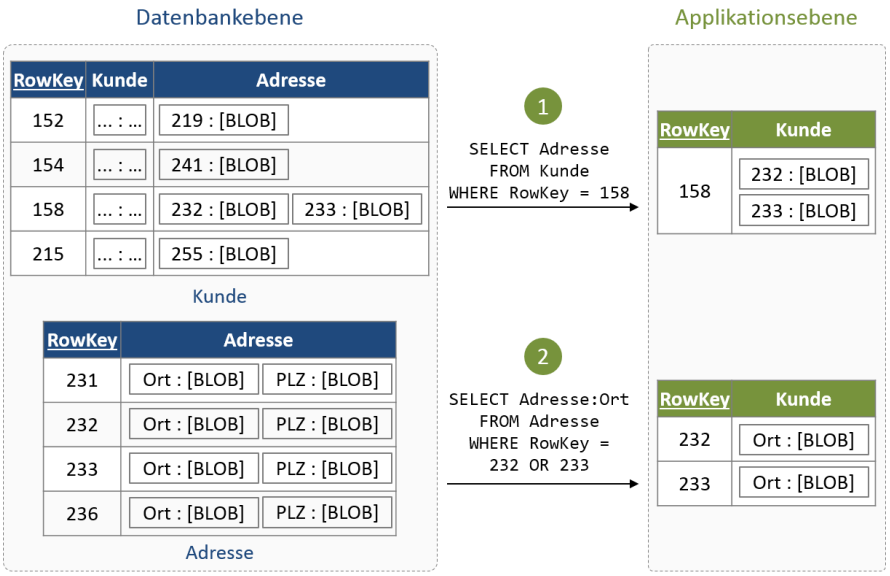


Abbildung 5.8: Zweistufige Auflösung von referenzierten Adressen

Die M:N-Beziehung zwischen den Entitäten *Kunde* und *Adresse* lässt sich wie in Abbildung 5.8 dargestellt abbilden. In diesem Beispiel wird die Tabelle *Kunde* um eine weitere Column Family *Adresse* erweitert, welche die Referenzen jedes Kundendatensatzes auf die zugehörigen Adressdatensätze der Tabelle *Adresse* enthält. Aufgrund fehlender Verbundoperationen sind für die Auflösung dieser Beziehung mindestens zwei Schritte notwendig. In einem ersten Schritt müssen die entsprechenden Referenzen aus der Tabelle *Kunde* gelesen werden, bevor in einem zweiten Schritte die referenzierten Datensätze aus der Tabelle *Adresse* selektiert werden können.

Denormalisierung: Die applikationsseitige Auflösung von Referenzen führt gerade bei mehrwertigen Beziehungen zu einer erheblichen Steigerung von Anfragen. Deswegen können die Daten für häufig benötigte Anfragen redundant in allen Entitäten gespeichert werden, die an einer Beziehung beteiligt sind. So ist beispielsweise die redundante Abbildung der primären Versandadresse eines Kunden in der Column Family *Kunde* eine Möglichkeit, um zusammenhängende Informationen mit einem Datenbankaufruf auslesen zu können. Je nach Konsistenzbestimmung können die redundanten Daten synchron oder asynchron aktualisiert werden.

Aggregation: Die Aggregation stellt die dritte Möglichkeit dar, um Beziehungen in Column Family Stores abbilden zu können. Hierzu werden zunächst auf konzeptueller Ebene semantisch untergeordnete Entitäten als Unterelemente der semantisch übergeordneten Entitäten modelliert. Das so entstehende konzeptuelle Aggregat wird daraufhin als eine Tabelle im logischen Datenmodell abgebildet. In dem konzeptuellen Datenmodell eines Onlineshops bietet sich die Beziehung zwischen den Entitäten *Kunde* und *Adresse* zur Aggregation an. Das entsprechende konzeptuelle Datenmodell ist in Abbildung 5.9 dargestellt.

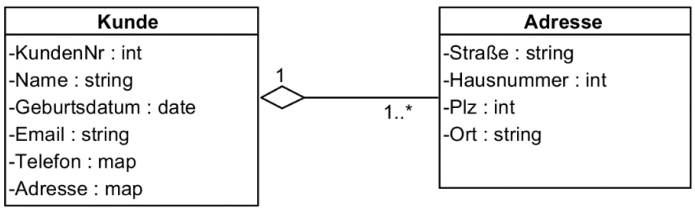


Abbildung 5.9: Konzeptuelle Darstellung der Aggregation *Kunde-Adresse*

Bei der Aggregation sollte die starke Entität in einer eigenen Tabelle abgebildet werden. Der Primärschlüssel der Entität wird als RowKey der Tabelle modelliert und die zugehörigen Schlüssel-Wert-Paare in einer Column Family zusammengefasst. Anschließend werden die eingebetteten Entitätstypen in jeweils einer Column Family abgebildet. Das in Abbildung 5.9 dargestellte Aggregat wird demzufolge auf eine Tabelle *Kunde* mit den beiden Column Families *Kunde* und *Adresse* abgebildet (Abbildung 5.10).

Innerhalb einer Column Family müssen alle Attribute eindeutig benannt sein. Da ein Kunde mehrere Adressen besitzen kann, ist die Eindeutigkeit der ursprünglichen Attributnamen allerdings nicht mehr gewährleistet. Folglich müssen in solchen Fällen die ursprünglichen Attributnamen um eine zusätzliche, für den Datensatz eindeutige Angabe, erweitert werden. In dem in Abbildung 5.10 dargestellten Beispiel wurde hierfür ein einfacher Zähler gewählt. Der Ort der primären Adresse eines Kunden wird dementsprechend über den zusammengesetzten Attributnamen *1:Ort* angesprochen.

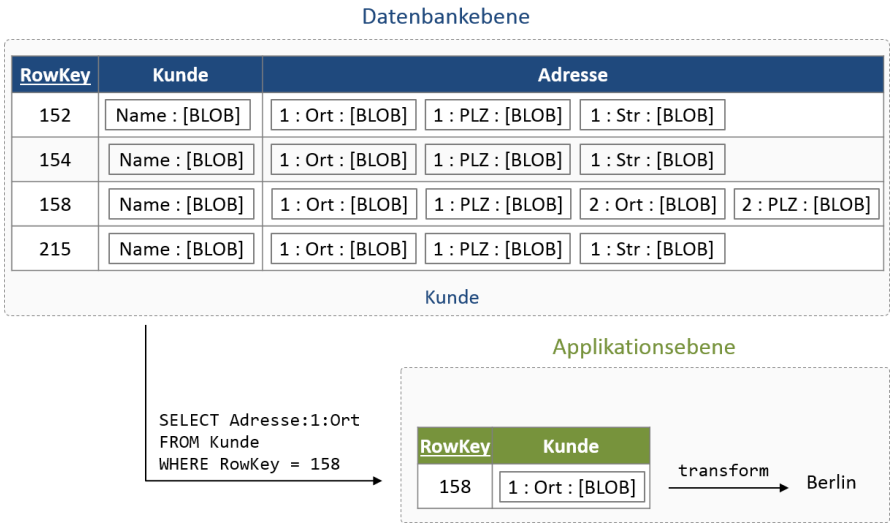


Abbildung 5.10: Logische Darstellung der Aggregation *Kunde-Adresse*

Die Aggregation verbindet die Vorteile der Referenzierung und der Denormalisierung, weil Beziehungen mit einem einzigen Datenbankaufruf aufgelöst werden können und gleichzeitig die referenzierten Entitäten im Fall von 1:1- oder 1:N-Beziehungen nicht redundant gespeichert werden. M:N-Beziehungen können dagegen nur aggregiert werden, wenn die eingebetteten Entitäten jeweils redundant in den starken Entitäten gespeichert werden.

Angesichts der Vorteile der Aggregation im Vergleich zur Referenzierung bietet sich auch bei Column Family Stores das in den vorherigen Kapiteln vorgestellte aggregierte Datenmodell des Online-Shops an. Da ausschließlich die beiden starken Entitäten *Kunde* und *Artikel* in einer eigenen Tabelle abgebildet werden müssen, reduziert sich die Anzahl der benötigten Tabellen deutlich. Die aggregierten Entitäten *Adresse* und *WarenkorbArtikel* werden jeweils in einer Column Family in der Tabelle *Kunde* abgebildet.

Modellierungsprozess

Column Family Stores verfügen über ein flexibles Datenmodell, auf das sich alle Entitäten und Beziehungen des konzeptuellen Datenmodells mithilfe der in diesem Kapitel vorgestellten Abbildungsregeln überführen lassen. Zwischen dem ER-Modell und Column Family Stores ergeben sich die in Tabelle 5.3 dargestellten Übereinstimmungen.

ER-Modell	Column Family Stores
Entitätstyp	Tabelle
Entität	Reihe
Einfaches Attribut	Schlüssel-Wert-Paar
Zusammengesetztes Attribut	Schlüssel-Wert-Paare, Map
Mehrwertiges Attribut	Array, Set, List
1:1-Beziehung	Referenzierung, Denormalisierung, Aggregation
1:N-Beziehung	
M:N-Beziehung	

Tabelle 5.3: Übereinstimmungen zwischen dem ER-Modell und Column Family Stores

Neben einem flexiblen Datenmodell bieten Column Family Stores eine Vielzahl an Operatoren, mit denen sich ausdrucksstarke Restriktionsbedingungen formulieren lassen. Allerdings beziehen sich diese ausschließlich auf die Primärschlüssel einer Tabelle und können nicht auf einfache Schlüssel-Wert-Paare angewendet werden. Zudem stellen Column Family Stores keine Verbundoperationen bereit, sodass Fremdschlüsselbeziehungen nicht datenbankseitig aufgelöst werden können. Zwar lassen sich sekundäre Zugriffspfade und Verbundoperationen auch applikationsseitig realisieren, doch steigt die Komplexität auf Anwendungsebene dann automatisch an, was wiederum eine erhöhte Fehleranfälligkeit und höhere Entwicklungs- und Wartungskosten zur Folge hat.

Aufgrund der hier beschriebenen Eigenschaften sind Column Family Stores nicht für die Abbildung komplexer Miniwelten mit mehreren Entitäten und Beziehungen geeignet. Die Stärken dieser Systeme liegen vielmehr in der Verarbeitung sehr großer Datensätze und der anschließenden Bereitstellung dieser Daten über einen Primärschlüssel. Eine wichtige Rolle nimmt dabei das Konzept der Column Families ein, mit deren Hilfe mehrere Schlüssel-Wert-Paare gruppiert sowie Entitäten und Beziehungen zu Aggregaten zusammengefasst werden können. Diese Aggregate sind, ähnlich wie Aggregate in Key Value Stores, auf bestimmte Anfragen des Anwendungsfalls ausgerichtet. Allerdings umfassen die Aggregate in Column Family Stores eine wesentlich größere Anzahl an Schlüssel-Wert-Paaren und sind daher in der Lage, deutlich größere Teilbereiche einer Miniwelt abzudecken. Die Abbildung dieser sehr großen Aggregate führt häufig zu sehr breiten Tabellen mit mehreren hundert Attributen:

> *„If your data model has no rows with over a hundred columns, you're either doing something wrong or you shouldn't be using Cassandra." – Ed Anuff* [10]

Column Family Stores sind genauso wie Key Value Stores auf bestimmte Anfragen eines Anwendungsfalls ausgelegt. Deswegen sollte der Modellierungsprozess zur systematischen Transformation eines konzeptuellen Datenmodells in das logische Datenmodell ebenfalls auf der Analyse der zu unterstützenden Anfragen basieren:

1. Bestimmung von Anfragen und Ergebnismengen,
2. Modellierung von Aggregationen,
3. Abbildung starker Entitäten,
4. Abbildung schwacher Entitäten,
5. Abbildung von Beziehungen.

Bestimmung von Anfragen und Ergebnismengen: Die Datenmodellierung mit Column Family Stores basiert auf der Identifikation der innerhalb eines Anwendungsfalls zu unterstützenden Anfragen. Für jede Anfrage müssen dann die Attribute, Entitäten und Beziehungen ermittelt werden, welche zur Bereitstellung der jeweiligen Ergebnismenge benötigt werden. Die so entstehenden Teilmengen der abzubildenden Miniwelt sind idealerweise disjunkt.

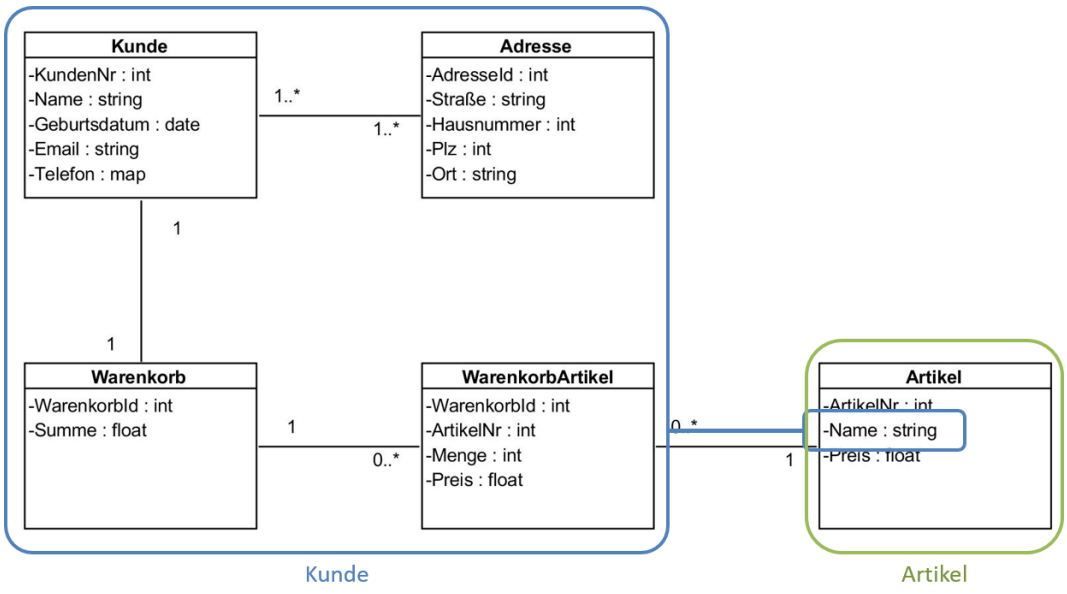


Abbildung 5.11: Identifikation von Teilmengen auf konzeptueller Ebene

Zwei mögliche Teilmengen der Miniwelt eines Onlineshops sind in Abbildung 5.11 dargestellt. Die blau gekennzeichnete Teilmenge *Kunde* umfasst die Entitäten *Kunde*, *Adresse*, *Warenkorb* und *WarenkorbArtikel* sowie das Attribut *Name* der Entität *Artikel*.

Entsprechend dient diese Teilmenge der Bereitstellung von Kunden- und Warenkorbinformationen. Die grün markierte Teilmenge *Artikel* enthält die Entität *Artikel* und wird folglich benötigt, um Informationen über den Produktkatalog zur Verfügung stellen zu können.

Modellierung von Aggregationen: Nach der Identifikation der zu unterstützenden Anfragen und die zur Bereitstellung der Ergebnisse benötigten Teilmengen der Miniwelt muss das konzeptuelle Datenmodell so angepasst werden, dass jede Teilmenge durch genau ein Aggregat repräsentiert wird. Hierzu muss zunächst für jede Teilmenge genau eine starke Entität identifiziert werden, welche als Basis der jeweiligen Aggregation herangezogen werden kann. Anschließend wird in einem zweiten Schritt der Primärschlüssel der Aggregation modelliert. Ein geeigneter Schlüsselkandidat ist der Primärschlüssel der jeweiligen starken Entität. Da dieser maßgeblich die Performanz und die Mächtigkeit der Anfrageverarbeitung beeinflusst, kann es abhängig von den Anfragen notwendig sein, den Schlüsselkandidaten um weitere Attribute zu ergänzen. Die Modellierung der Aggregate endet mit der Abbildung von Attributen, Entitäten und Beziehungen, welche in mehreren Ergebnismengen vorkommen. Abhängig vom Anwendungsfall muss dabei entschieden werden, ob diese redundant in jedem Aggregat oder als eigenständige Entitäten, die nicht Teil eines Aggregats sind, modelliert werden.

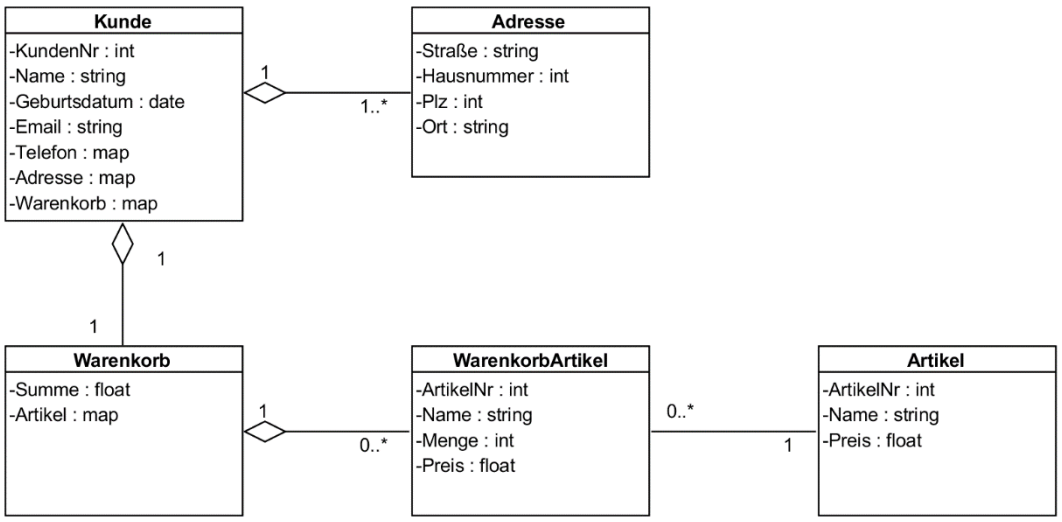


Abbildung 5.12: Modellierung der identifizierten Teilmengen als Aggregationen

In Abbildung 5.12 ist das angepasste konzeptuelle Datenmodell des Onlineshops dargestellt. Die beiden Teilmengen wurden auf die starken Entitäten *Kunde* und *Artikel* abgebildet. Die schwachen Entitäten *Adresse*, *Warenkorb* und *WarenkorbArtikel* wurden in die starke Entität *Kunde* eingebettet. Das Attribut *Name* der Entität *Artikel*,

das zur Bereitstellung beider Teilmengen benötigt wird, wurde redundant in den Entitäten *WarenkorbArtikel* und *Artikel* abgebildet, um eine Auflösung von Fremdschlüsselbeziehungen auf Applikationsebene zu vermeiden.

Abbildung starker Entitäten: Nach der Anpassung des konzeptuellen Datenmodells sollten zunächst die starken Entitäten in das logische Datenmodell von Column Family Stores überführt werden. Unter Berücksichtigung der in diesem Kapitel vorgestellten Abbildungsregeln werden die in Abbildung 5.12 modellierten Aggregate durch die Tabellen *Kunde* und *Artikel* dargestellt und als Primärschlüssel die Schlüsselkandidaten *KundenNr* und *ArtikelNr* herangezogen.

Abbildung schwacher Entitäten: Schwache Entitäten können als Column Families in starke Entitäten eingebettet werden. Hierbei ist zu beachten, dass eingebettete Entitäten in Column Family Stores angesichts fehlender sekundärer Indizes nicht referenziert werden können.

Abbildung von Beziehungen: Abschließend werden die verbleibenden Beziehungen abgebildet. Hinsichtlich der bereits auf konzeptueller Ebene erfolgten Aggregation und Denormalisierung bleibt an dieser Stelle die Referenzierung als einziges verbleibendes Modellierungsmuster übrig. Bei der Abbildung von Fremdschlüsselbeziehungen muss abhängig von den zu unterstützenden Anfragen entschieden werden, in welchen Entitäten Referenzen abgebildet werden müssen.

RowKey	Adresse	Kunde	Warenkorb	WarenkorbArtikel		
152	1:Ort: [...]	Name: [...]	Summe: [...]	35:Menge: [...]	35:Name: [...]	35:Preis: [...]
154	1:Ort: [...]	Name: [...]	Summe: [...]	25:Menge: [...]	25:Name: [...]	35:Preis: [...]
158	1:Ort: [...]	Name: [...]	Summe: [...]	57:Menge: [...]	57:Name: [...]	57:Preis: [...]
215	1:Ort: [...]	Name: [...]	Summe: [...]	33:Menge: [...]	33:Name: [...]	33:Preis: [...]
221	1:Ort: [...]	Name: [...]	Summe: [...]	15:Menge: [...]	15:Name: [...]	15:Preis: [...]
243	1:Ort: [...]	Name: [...]	Summe: [...]	42:Menge: [...]	42:Name: [...]	42:Preis: [...]

Abbildung 5.13: Logisches Datenmodell der aggregierten Entität *Kunde*

Eine mögliche logische Darstellung des konzeptuellen Aggregats *Kunde* ist in Abbildung 5.13 dargestellt. Die abgebildete Tabelle besteht aus den vier Column Families *Adresse*, *Kunde*, *Warenkorb* und *WarenkorbArtikel* und besitzt den Schlüsselkandidaten *KundenNr* als Primärschlüssel. Entsprechend der in diesem Kapitel vorgestellten Abbildungsregeln werden zur Abbildung von 1:N-Beziehungen zusammengesetzte Attributschlüssel verwendet. So bestehen die Schlüssel in der Column Family *WarenkorbArtikel* aus der *AtikelNr* und dem zugehörigen Attributnamen.

Anhand des in diesem Kapitel vorgestellten Modellierungsprozesses und der entwickelten Abbildungsregeln wird deutlich, dass zur systematischen Transformation eines konzeptuellen Datenmodells auf das logische Datenmodell eines Column Family

Stores eine hohe Benutzerinteraktion notwendig ist. Der Grund hierfür liegt darin, dass die Datenmodellierung bei Column Family Stores sehr stark von den zu unterstützenden Anfragen geprägt ist und die Anforderungen dieser Anfragen nicht im konzeptuellen Datenmodell abgebildet werden können. Eine automatisierte Überführung des konzeptuellen Datenmodells in das logische Datenmodell mithilfe von CASE-Werkzeugen kann diesen essenziellen Aspekt der Datenmodellierung nicht berücksichtigen und daher zu keinem zufriedenstellenden Resultat führen.

5.3 Interne Ebene

Column Family Stores wurden für die hochperformante Verarbeitung von Lese- und Schreiboperationen auf sehr großen Datenmengen, wie sie unter anderem bei Google und Facebook auftreten, konzipiert. Da solche Datenmengen nur von verteilten Systemen verarbeitet werden können, sind Column Family Stores von Beginn an auf eine verteilte Datenhaltung ausgelegt. Mit Ausnahme von Cassandra sind für den Betrieb eines Column Family Stores immer mindestens fünf Server notwendig. Die technischen Eigenschaften dieser Systeme sind Gegenstand dieses Unterkapitels.

5.3.1 Speicherstrukturen

Column Family Stores speichern die Inhalte ihrer Tabellen nicht wie relationale Datenbanken zeilenweise, sondern zellenweise ab. Hierzu werden Tabellen vertikal fragmentiert und jede Zelle auf ein Schlüssel-Wert-Paar abgebildet. Ein Schlüssel besteht aus dem Primärschlüssel des Datensatzes, dem Namen der Column Family und dem Namen des entsprechenden Attributes und einem Zeitstempel, der für die Synchronisation konkurrierender Zugriffe herangezogen wird:

```
<Row Key> <Column Family:Column> <Timestamp>.
```

Durch die vertikale Fragmentierung der Tabelle müssen nur die Zellen intern gespeichert werden, für deren Attribute ein Datensatz auch Attributwerte besitzt. Die explizite Speicherung von `null`-Werten ist hier, im Gegensatz zu relationalen Datenbanken, nicht notwendig.

Die einzelnen Schlüssel-Wert-Paare werden in sogenannte SSTables (Sorted String Table) persistent gespeichert [30]. Ein SSTable besteht hauptsächlich aus mehreren Datenblöcken, in denen die Schlüssel-Wert-Paare sortiert nach ihrem Schlüssel abgelegt werden. Zusätzlich enthält ein SSTable einen Blockindex, der den jeweils letzten Schlüssel eines Blocks referenziert. SSTable sind unveränderlich, das heißt, die in ihnen gespeicherten Schlüssel-Wert-Paare werden nicht überschrieben und auch nicht gelöscht. Der Aufbau eines SSTables ist in Abbildung 5.14 dargestellt.

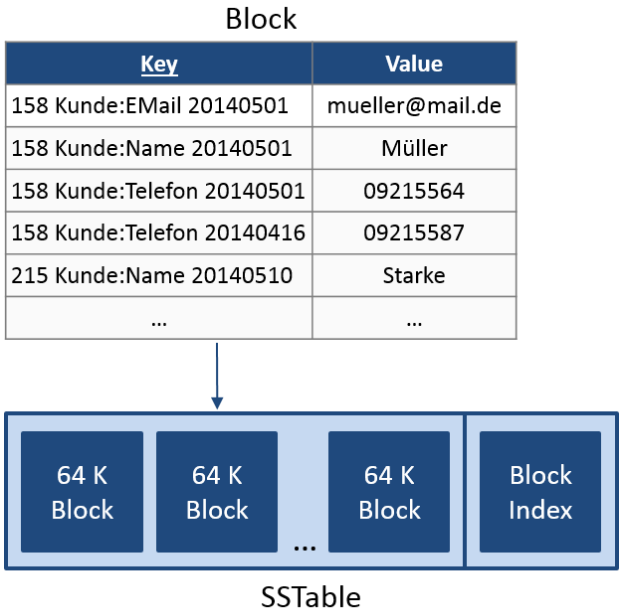


Abbildung 5.14: Aufbau eines SSTables

Schreiboperationen werden in einer persistenten Log-Datei protokolliert, bevor sie bestätigt werden. Die Log-Datei wird abhängig vom eingesetzten Datenbanksystem sofort oder in konfigurierbaren Intervallen persistiert [11, 42]. Parallel hierzu werden die neuen Schlüssel-Wert-Paare im sogenannten Memtable zwischengespeichert. Der Memtable ist ein im Hauptspeicher befindlicher Cache, der solange Daten aufnimmt, bis er eine definierte Größe erreicht. Sobald der Cache gefüllt ist, werden die in ihm befindlichen Schlüssel-Wert-Paare sequenziell in einen neuen SSTable geschrieben. War das Auslagern erfolgreich, werden sowohl Cache als auch Log-Datei geleert. In Abbildung 5.15 ist das Schreiben des Wertes `x=6` exemplarisch dargestellt.

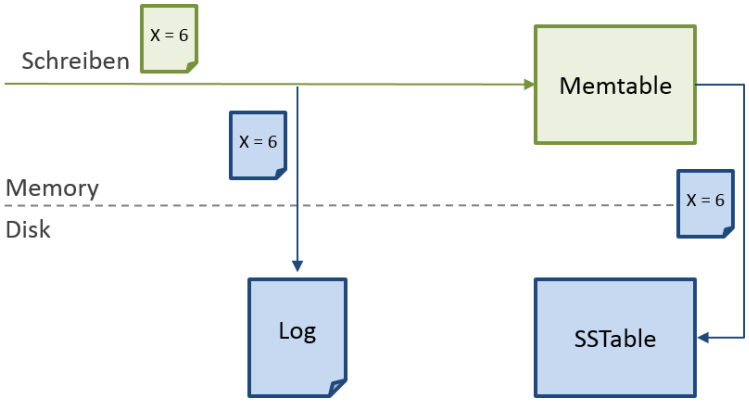


Abbildung 5.15: Verarbeitung einer Schreiboperation

Weil die Daten in einem SSTable nicht überschrieben werden, können vor allem bei häufigen Änderungen sehr viele verschiedene Versionen eines Datensatzes entstehen. Um den Speicherplatzbedarf zu reduzieren und die Lesegeschwindigkeit zu erhöhen, werden verschiedene Komprimierungsschritte in Hintergrundprozessen ausgeführt. Dabei werden verschiedene SSTables zu einem SSTable zusammengefasst. Nicht mehr benötigte Versionen eines Datensatzes oder als gelöscht markierte Daten werden bei der Zusammenführung nicht berücksichtigt, wodurch belegte Ressourcen automatisch wieder freigegeben werden. Dank der alphabetischen Sortierung der Schlüssel-Wert-Paare kann die Komprimierung mit einem relativ geringen Aufwand durchgeführt werden.

Column Family Stores verwalten zu einem Zeitpunkt standardmäßig immer mindestens drei Versionen von jedem Schlüssel-Wert-Paar. Demzufolge müssen trotz Komprimierung bei einer Leseanfrage immer mehrere SSTables durchsucht werden, um die korrekte Version eines Datensatzes zurückgeben zu können (Abbildung 5.16).

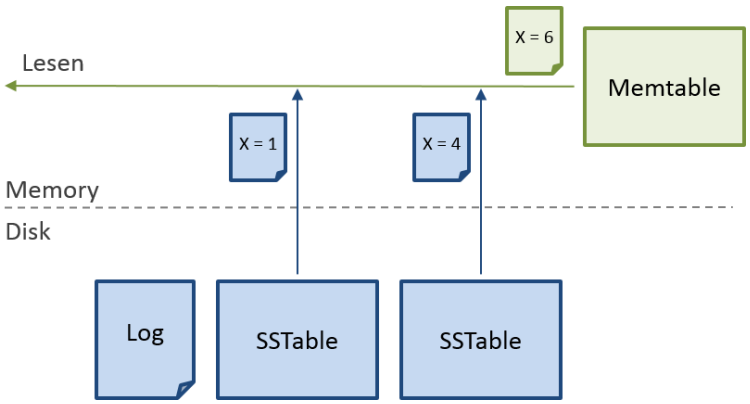


Abbildung 5.16: Verarbeitung einer Leseoperation

Mithilfe von Bloom Filtern [17], die sich im Hauptspeicher befinden, sind Column Family Stores in der Lage, alle SSTables ausschließen zu können, die definitiv das gesuchte Schlüssel-Wert-Paar nicht enthalten. Erst wenn die Möglichkeit besteht, dass sich die gesuchten Informationen in einem bestimmten SSTable befinden, wird dieser durchsucht. Hierbei wird zunächst der Blockindex des SSTables in den Hauptspeicher geladen und mit einer Binärsuche die benötigte Blockadresse ermittelt. Anschließend erfolgt im Rahmen eines zweiten Plattenzugriffs das Lesen des gesuchten Blocks. Die Anzahl der Plattenzugriffe kann beim Column Family Store Cassandra unter Zuhilfenahme des Index-Caches und des Row-Caches reduziert werden [40, S.101]. Idealerweise befinden sich sowohl der Blockindex als auch der Datensatz bereits im Hauptspeicher, sodass kein einziger Plattenzugriff bei Leseanfragen notwendig ist. Sobald alle betroffenen SSTables und der zum Zeitpunkt der Leseanfrage gültige Memtable durchsucht wurden, müssen die verschiedenen Versionen der benötigten Daten zu einer gültigen Version zusammengeführt werden.

Sekundäre Zugriffspfade

In Column Family Stores können Datensätze ausschließlich über ihren Primärschlüssel angesprochen werden. Restriktionen, die auf einem Nicht-Schlüssel-Attribut basieren, werden nicht unterstützt. Aus diesem Grund müssen für solche Anfragen weitere Tabellen erzeugt werden, bei denen das benötigte Attribut als Primärschlüssel modelliert ist. Der Nachteil dieses „manuellen Index" liegt im zusätzlichen Verwaltungsaufwand, der durch Anfrageweiterleitung und Synchronisation entsteht. Dieser Aufwand wird inzwischen allerdings von den Column Family Stores Hypertable und Cassandra datenbankseitig übernommen [39, 90]. Aus Performanzgründen sollte die Wahl eines sekundären Zugriffspfades auf Attribute fallen, deren Werte sich in vielen Datensätzen ähneln [42]. So ist beispielsweise das Attribut *Ort* im Gegensatz zum Attribut *Telefonnummer* ein guter Kandidat für einen Index in der Tabelle *Kunde*.

5.3.2 Transaktionen und konkurrierende Zugriffe

Die Transaktionsverwaltung ist für die sichere Ausführung von Operationen, trotz Mehrbenutzerzugriff oder potenzieller Systemfehler, verantwortlich. Sie besteht aus den beiden Komponenten Recovery und Synchronisation. Das Verhalten dieser beiden Komponenten wird anhand des bereits bekannten Datenmodells eines Warenkorbs veranschaulicht (Abbildung 5.17).

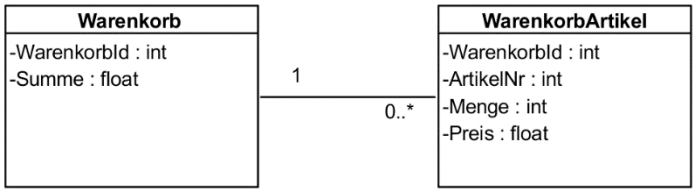


Abbildung 5.17: Konzeptuelles Modell der Entitäten *Warenkorb* und *WarenkorbArtikel*

Recovery

Column Family Stores unterstützen keine Transaktionen. Auf einem Datensatz kann immer nur eine Operation atomar ausgeführt werden. Werden die beiden in Abbildung 5.17 dargestellten Entitäten als eigenständige Tabellen modelliert, müssen im Fall eines Fehlers beim Hinzufügen eines Artikels die an dieser „Transaktion" beteiligten Operationen applikationsseitig zurückgesetzt werden. Dieser hohe Aufwand auf Applikationsebene kann in manchen Fällen mithilfe des Modellierungsmusters der Aggregation reduziert werden.

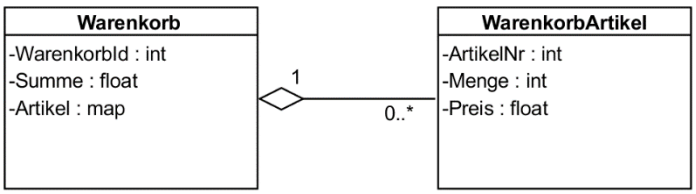


Abbildung 5.18: Aggregation der Entitäten *Warenkorb* und *WarenkorbArtikel*

Werden die beiden Entitäten, wie in Abbildung 5.18 dargestellt, zu einem Aggregat zusammengefasst, befinden sich alle benötigten Informationen innerhalb eines Datensatzes. Durch diese Modellierung lassen sich innerhalb einer atomaren Operation sowohl die Summe aktualisieren als auch neue Artikel zum Warenkorb hinzufügen.

Schreiboperationen werden vor der Bestätigung gegenüber einem Client in einer persistenten Log-Datei gesichert. Tritt vor der Auslagerung des sich im Hauptspeicher befindlichen Memtables in einen persistenten SSTable ein Systemfehler auf, lässt sich mit dieser Log-Datei der alte, konsistente Datenbankzustand wiederherstellen. Änderungen in einem Column Family Store werden deshalb dauerhaft gesichert.

Synchronisation

Konkurrierende Änderungsoperationen werden in Column Family Stores mithilfe des optimistischen Synchronisationsverfahrens MVCC (siehe Kapitel 2.3.2) koordiniert. Wird ein Datensatz geändert, werden die bestehenden Schlüssel-Wert-Paare nicht am selben Speicherplatz überschrieben, sondern es werden neue Versionen dieser Daten erstellt. Hierdurch sind Änderungsoperationen unabhängig von anderen Lese- und Schreiboperationen [40, S.74] [30, S.3].

Allerdings steigt mit MVCC der Aufwand, der auf Applikationsseite betrieben werden muss. Neben den zur Synchronisation notwendigen Zeitstempeln müssen vor allem Mechanismen bereitgestellt werden, welche Anomalien verhindern beziehungsweise auflösen. Obendrein sind Column Family Stores standardmäßig nicht in der Lage, Integritätsprüfungen durchzuführen, weshalb die Überprüfung von statischen und dynamischen Integritätsbedingungen zur Sicherung der Konsistenz applikationsseitig erfolgen muss.

5.4 Verteilte Datenhaltung

Column Family Stores werden häufig in Anwendungsfällen mit sehr großen Datenmengen eingesetzt. Daher handelt es sich bei Column Family Stores fast immer um verteilte Datenbanksysteme. Mit Ausnahme von Cassandra, dessen Verteilung auf Consistent Hashing basiert, wird ein Column Family Store-Cluster zentral koordiniert und besteht deswegen aus mindestens drei Servern. Man unterscheidet die drei Komponenten Tablet-Server, Master-Server und Chubby. Die Namen dieser Komponenten können sich allerdings je nach eingesetztem Datenbanksystem voneinander unterscheiden. Die Architektur eines Column Family-Clusters ist in Abbildung 5.19 dargestellt.

Tablet-Server

Tablet-Server sind für die Verwaltung bestimmter Fragmente (Tablets) einer Tabelle zuständig. Neben der Verarbeitung von Lese- und Schreibanfragen, die direkt von den Clients an sie gesendet werden, gehört die persistente Speicherung der Daten in SSTables zu den Aufgaben eines Tablet-Servers. In der Regel werden die einzelnen SSTables nicht auf dem zuständigen Tablet-Server abgelegt, sondern von dort an ein verteiltes Dateisystem weitergeleitet (siehe 5.4.1). Tablet-Server können dynamisch zum Cluster hinzugefügt oder entfernt werden. Die Anzahl der Tablet-Server kann bei mehreren Tausend liegen.

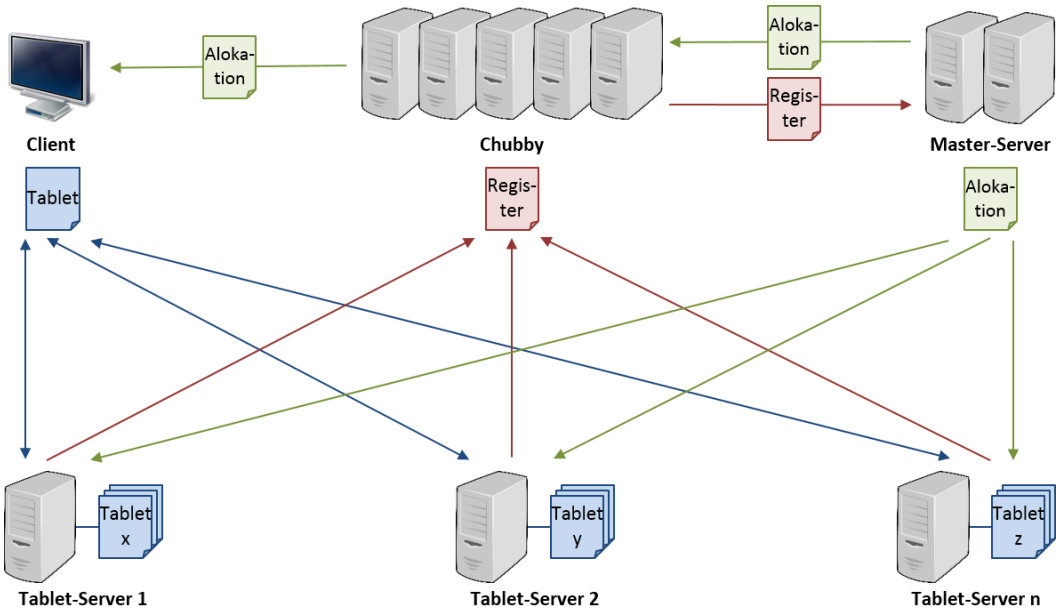


Abbildung 5.19: Aufbau eines Column Family Store-Clusters

Master-Server

Der Master-Server ist für die Verteilung der Daten zuständig. Neben der Allokation von Tablets zu Tablet-Servern gehören die Lastverteilung und die Fehlerbehebung zu seinen Aufgaben. Ist ein Tablet-Server nicht mehr verfügbar oder ist ein Tablet-Server überlastet, weist der Master-Server einzelnen Tablets neue Tablet-Server zu. Daneben ist er für Schemaänderungen, wie die Erstellung und Löschung von Tabellen und Column Families, zuständig.

Clients sind aufgrund spezieller Schnittstellen in der Lage, Lese- und Schreibanfragen direkt an die zuständigen Tablet-Sever zu senden. Für die Lokalisierung eines Tablet-Servers wird der Master-Server dank des Verzeichnisdienstes Chubby nicht benötigt. Demzufolge ist der Master-Server in der Regel nur leicht ausgelastet und kann sogar kurzfristig ausfallen ohne dabei die Verfügbarkeit des Systems zu beeinträchtigen. Um jedoch eine permanente Lastverteilung, Fehlerbehandlung und Schemaänderungen zu ermöglichen, sollte der Master-Server durch mindestens ein Replikat gesichert werden [89, S.1].

Chubby

Chubby ist ein hochverfügbarer, persistenter und verteilter Verzeichnisdienst, der je nach eingesetztem Datenbanksystem auch als Hyperspace (Hypertable) oder ZooKeeper (HBase) bezeichnet wird. Der Verzeichnisdienst ist in Namensräume unterteilt, in denen sich wiederum Ordner und Dateien befinden. Mithilfe von speziellen Bibliotheken können im Rahmen einer Session exklusive Sperren auf Elemente dieser Datenstrukturen reserviert werden. Sobald die Session aktiv oder infolge eines Fehlers beendet wird, werden auch die Sperren automatisch gelöst.

Chubby wird von Column Family Stores für eine Vielzahl verschiedener Aufgaben eingesetzt. Eine davon ist die Verwaltung der Speicheradresse der vom Master-Server erstellten Metadaten-Tabelle, welche die Allokationsinformationen der einzelnen Tablets enthält. Da Clients anhand der Adresse direkt auf die Allokationsinformationen zugreifen können, müssen sie zu keinem Zeitpunkt den Master-Server kontaktieren. Eine weitere Aufgabe von Chubby ist die Verwaltung der Tablet-Server. Jeder Tablet-Server muss sich beim Start in einem speziellen Verzeichnis durch das Anlegen und Sperren einer Datei registrieren. Wird ein Tablet-Server entfernt oder ist dieser wegen eines Fehlers nicht verfügbar, wird automatisch die Session beendet und die Sperre auf die angelegte Datei gelöst. Da das Tablet-Server-Verzeichnis permanent vom Master-Server überwacht wird, ist dieser jederzeit über die gegenwärtige Cluster-Topologie informiert. Chubby wird außerdem für die Speicherung von Schemainformationen und Zugriffskontrolllisten verwendet. Zudem kann mittels einer exklusiven Sperre auf ein Master-Server-Verzeichnis sichergestellt werden, dass zu einem Zeitpunkt nur ein Master-Server aktiv ist.

Da Chubby alle zur Ausführung eines Column Family Stores essenziellen Status- und Konfigurationsinformationen verwaltet, hängt die Verfügbarkeit des gesamten Sys-

tems von der Verfügbarkeit dieses Services ab. Chubby wird deshalb durch vier Replikate gesichert. Anfragen an den Service werden mithilfe des Paxos Konsensusprotokolls synchronisiert, sodass Chubby reibungslos funktioniert, solange mindestens drei dieser Server verfügbar sind [30, S.4].

5.4.1 Replikation

Master-Server von Column Family Stores weisen einzelne Tablets immer nur einem Tablet-Server zu und unterstützen dadurch keine Replikation. Um dennoch eine höchstmögliche Dauerhaftigkeit und Verfügbarkeit der Daten zu gewährleisten, können Tablet-Server die SSTables zusätzlich in verteilten Dateisystemen speichern. Verteilte Dateisysteme wie das *Google File System* [71] und dessen Open Source Implementierung *Hadoop Distributed File System* [157] sind hochverfügbare, mehrfachreplizierte Systeme und werden entsprechend häufig von Column Family Stores zur persistenten Speicherung ihrer Daten herangezogen.

5.4.2 Fragmentierung und Allokation

Sobald eine Tabelle eine Größe von 200 Megabyte erreicht, wird sie horizontal in sogenannte *Tablets* fragmentiert. Jedes Tablet enthält mehrere Datensätze, deren Primärschlüssel in einem bestimmten Wertebereich liegen. Ein Tablet ist immer genau einem Tablet-Server zugewiesen. Diese Zuweisung ist Aufgabe des Master-Servers. Um eine bestmögliche Lastverteilung sowie eine schnelle Umverteilung im Fall eines Tablet-Server-Fehlers ermöglichen zu können, überwacht der Master-Server permanent die Zustände aller Tablet-Server. Zentraler Bestandteil der Allokation ist der Verzeichnisdienst Chubby. Zum einen kann der Master-Server alle dort registrierten Tablet-Server kontaktieren und Statusinformationen abrufen, zum anderen ist in Chubby die Speicheradresse der Metadaten-Tabelle hinterlegt, welche Informationen zur gegenwärtigen Allokation der Tablets beinhaltet.

Die Anzahl an Tablets ändert sich, wenn eine Tabelle erstellt oder gelöscht wird, wenn im Rahmen einer Komprimierung mehrere Tablets zu einem Tablet zusammengefasst werden oder wenn ein Tablet eine Größe von 200 Megabyte überschreitet und deswegen auf zwei Tablets aufgeteilt werden muss. Da diese Änderungen sich auf die Allokation der Tablets auswirken, müssen sie beim Master-Server bekanntgemacht werden. Die ersten drei Änderungen werden allerdings selbst vom Master-Server initiiert, sodass letztendlich nur die von Tablet-Servern durchgeführte Aufteilung eines Tablets beim Master-Server registriert werden muss [30, S.5].

5.4.3 Verteilte Anfrageverarbeitung

Die Suche nach einem für ein Tablet zuständigen Tablet-Server erfolgt mithilfe der in Chubby adressierten Metadaten-Tabelle. Diese ist in ein Root-Tablet und mehrere Metadaten-Tablets unterteilt. Analog zu einem B*-Baum verweist das Root-Tablet auf die

verschiedenen Wertebereiche der Metadaten-Tablets, welche wiederum die Wertebereiche der konkreten Tablet-Server referenzieren (Abbildung 5.20) [30, S.4].

Die in den Clients eingebundenen Bibliotheken können die Tablet-Server-Adressen temporär zwischenspeichern. Erst wenn eine Tablet-Server-Adresse nicht bekannt oder inzwischen ungültig ist, werden diese Informationen aktualisiert. Clients sind somit in der Lage, Lese- und Schreibanfragen an die für einen Wertebereich zuständigen Tablet-Server selbstständig abzuschicken und müssen hierfür nicht den Master-Server kontaktieren.

Column Family Stores besitzen Schnittstellen zu MapReduce-Frameworks wie beispielsweise *Hadoop*[45]. Hierdurch können Column Family Stores sowohl Datenquelle als auch Ziel von MapReduce-Analysen darstellen [30, S.3].

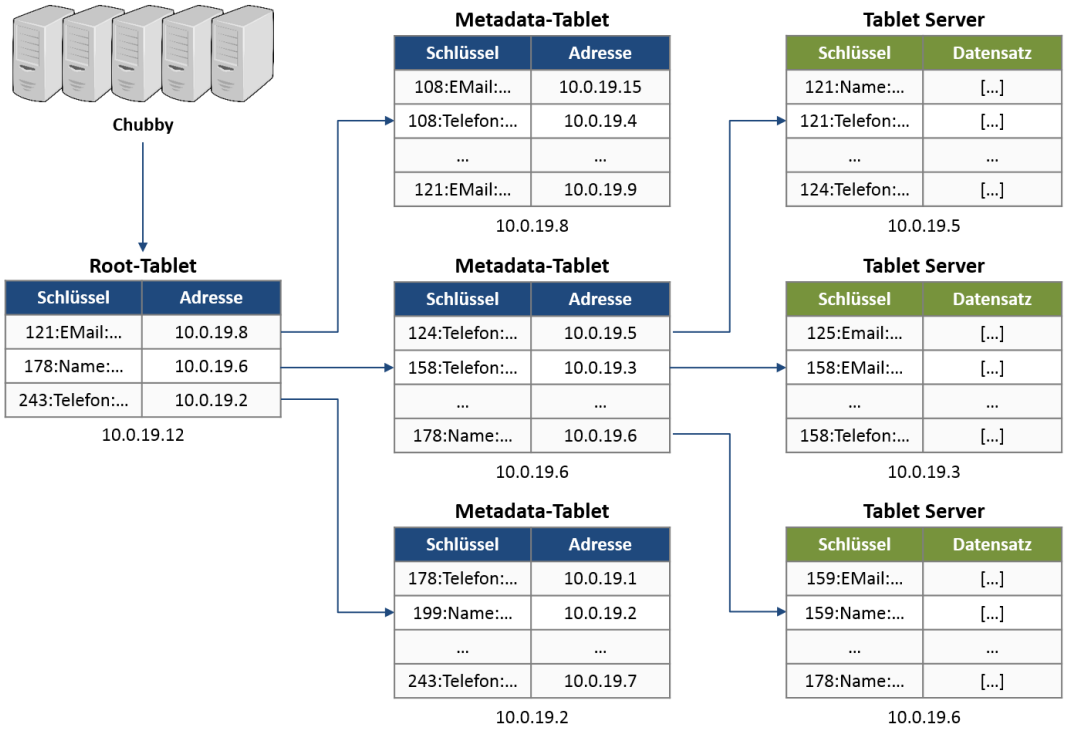


Abbildung 5.20: Hierarchie der Katalogtabellen

[45] http://hadoop.apache.org

5.4.4 Verteilte Transaktionsverwaltung

Wie bereits erwähnt, handelt es sich bei Column Family Stores in der Regel um verteilte Datenbanksysteme. Aus diesem Grund unterscheiden sich die an dieser Stelle erwähnten Techniken zur Gewährleistung der Datenbankkonsistenz in verteilten Datenbanksystemen nicht von denen, die bereits in Kapitel 5.3.2 vorgestellt wurden.

Recovery

Ist ein Tablet-Server infolge eines Fehlers für einen bestimmten Zeitraum nicht verfügbar, wird dieser Ausfall vom Master-Server registriert. Der Master-Server weist daraufhin den Tablets des nicht verfügbaren Tablet-Servers neue Tablet-Server-Adressen zu. Wurde ein verteiltes Dateisystem als Persistenzschicht eingesetzt, sind die dort gespeicherten SSTables und Log-Dateien nicht vom Ausfall des Tablet-Servers betroffen.

Synchronisation

Konkurrierende Lese- und Schreibzugriffe werden mittels MVCC synchronisiert. Jede Zelle einer Tabelle wird standardmäßig dreifach versioniert gespeichert. Bei Leseanfragen wird in der Regel die jüngste Version eines Schlüssel-Wert-Paares zurückgegeben. Alternativ können auch mehrere widersprüchliche Versionen ausgegeben und auf Applikationsebene zusammengeführt werden.

5.5 Zusammenfassung

Column Family Stores sind für die Verarbeitung extrem großer Datenmengen konzipiert und dementsprechend von Beginn an auf eine verteilte Datenhaltung ausgelegt. Das Datenmodell ähnelt rein optisch einer relationalen Tabelle. Allerdings werden hier Attributnamen nicht in einem globalen Schema, sondern lokal im jeweiligen Datensatz gespeichert. Datensätze in Column Family Stores sind darum, ähnlich wie in Document Stores, selbstbeschreibend, weisen jedoch eine geringere Ausdrucksstärke auf. Da Attributwerte als Byte-Arrays gespeichert werden, können diese nicht vom Datenbanksystem interpretiert und demzufolge Funktionalitäten wie Datentypen nicht bereitgestellt werden. Die meisten Column Family Stores verfügen über SQL-ähnliche Anfragesprachen mit einem großen Funktionsumfang. Allerdings können Restriktionsbedingungen nur auf dem Primärschlüssel einer Tabelle formuliert werden. Cassandra bietet deshalb die Möglichkeit, Indizes mithilfe zusätzlicher Tabellen zu realisieren, wodurch indirekt ein Zugriff auf Nichtschlüssel-Attribute ermöglicht wird.

Column Family Stores sind auf eine hohe Schreibgeschwindigkeit von persistenten Daten ausgelegt. Zu Gunsten der Performanz verzichten diese Systeme daher auf viele

ACID-Eigenschaften. So lassen sich weder zusammenhängende Operationen zu Transaktionen zusammenfassen noch Integritätsbedingungen zur Sicherung der Konsistenz formulieren und überprüfen. Außerdem werden konkurrierende Operationen nur durch optimistische Mehrversionsverfahren synchronisiert. Einzig die persistente Sicherung von Änderungsoperationen entspricht vollständig den ACID-Eigenschaften.

Die Verteilung von Column Family Stores wird mit Ausnahme von Cassandra, das Consistent Hashing einsetzt, zentral koordiniert. Die hierfür zuständigen Server sind aus Gründen der Ausfallsicherheit mehrfach repliziert gesichert. Column Family Stores sind für den Einsatz in einem breiten Spektrum an Anwendungsfällen mit unterschiedlichen Anforderungen an Konsistenz und Verarbeitungsgeschwindigkeit konzipiert. Entsprechend flexibel lassen sich die für diese verschiedenen Anforderungen charakteristischen Eigenschaften konfigurieren. Durch die vergleichsweise komplexe Systemarchitektur und die vielfältigen Konfigurationsmöglichkeiten benötigen die auf Bigtable basierenden Column Family Stores bei der erstmaligen Einrichtung einen relativ hohen administrativen Aufwand.

Funktionalitäten		HBase	Cassandra
Ausdrucksstarkes Datenmodell	Attributnamen	✓	✓
	Datentypen	✗	✗
Anfragemächtigkeit	Relationale Operatoren	✓	✓
	Verbundoperationen	✗	✗
	Sekundäre Zugriffspfade	✗	✓
ACID-Eigenschaften	Atomarität	✗	✗
	Integritätsbedingungen	✗	✗
	Synchronisationsmechanismen	✓	✓
	Dauerhaftigkeit	✓	✓
Verteilte Datenhaltung	Replikation	✓	✓
	Verteilung	✓	✓

Tabelle 5.4: Funktionsumfang von Column Family Stores

Die konsequente Ausrichtung von Column Family Stores auf eine hohe Performanz bei persistenten Schreiboperationen spiegelt sich im Funktionsumfang dieser Systeme wider (Tabelle 5.4). Durch den Verzicht auf Verbundoperationen, Transaktionen, Integritätsbedingungen und pessimistischer Synchronisation sind viele Funktionalitäten ausgeschlossen, welche sich negativ auf die Geschwindigkeit von Schreiboperationen auswirken können. Allerdings kann dieser Verzicht zu einem nicht unerheblichen Mehraufwand auf Applikationsebene führen, sobald einige dieser Funktionalitäten in einem Anwendungsfall benötigt werden. Die applikationsseitige Auflösung von Fremdschlüsselbeziehungen oder die manuelle Umsetzung von sekundären Zugriffspfaden kann die erzielten Geschwindigkeits- und Skalierbarkeitsvorteile gegenüber relationalen Datenbanken sehr schnell relativieren.

Die Column Family Stores HBase und Hypertable basieren auf den Konzepten von Googles Bigtable und weisen folglich kaum Unterschiede in ihren Systemeigenschaften auf. Cassandra kombiniert dagegen das Konzept von Bigtable mit dem Konzept von Amazons Dynamo und unterscheidet sich deswegen insbesondere bei der verteilten Datenhaltung von den anderen Column Family Stores. Die erstellten Modellierungsprozesse und Abbildungsregeln sind hiervon nicht betroffen und sind damit systemübergreifend gültig.

6

KRITISCHE BETRACHTUNG VON NOSQL

Kapitelinhalt

- Semistrukturierte Daten
- Verarbeitungsgeschwindigkeit
- Skalierbarkeit

Anwendungen im Bereich Big Data erfassen, speichern und analysieren sehr große Datenmengen aus unterschiedlichen Quellen. Die hieraus gewonnenen Informationen sollen – wenn möglich in Echtzeit – bereitgestellt werden, um die Entscheidungsfindung in Unternehmen, Instituten und Behörden zu unterstützen. Als charakteristische Eigenschaften von Big Data-Anwendungsfällen gelten große Datenmengen, hohe Geschwindigkeitsanforderungen und eine breite Vielfalt der zu verarbeitenden Daten (Abbildung 6.1).

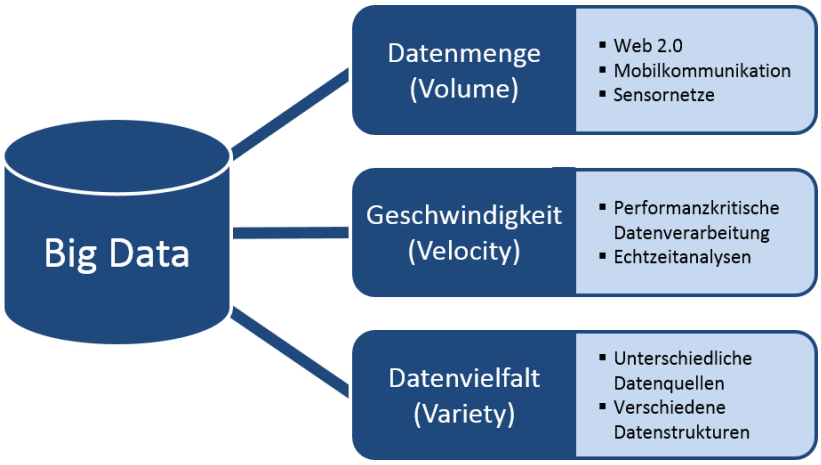


Abbildung 6.1: Charakteristische Eigenschaften von Big Data

Die Eigenschaften von Big Data-Anwendungsfällen unterscheiden sich deutlich von denen klassischer Unternehmensanwendungen, für die relationale Datenbanken konzipiert wurden. Die hohe strukturelle Vielfalt, die wachsende Geschwindigkeit sowie

die gestiegene Menge der zu verarbeitenden Daten stellen derart hohe Anforderungen an heutige Datenbanksysteme, dass diese die Möglichkeiten und Kapazitäten von relationalen Datenbanken übersteigen. Im Gegensatz zu relationalen Datenbanken sind NoSQL-Datenbanken gezielt auf die Anforderungen von Big Data-Anwendungsfällen ausgelegt. Durch Unterschiede sowohl auf konzeptueller als auch auf interner Ebene versprechen diese Systeme ein flexibleres Datenmodell, eine höhere Verarbeitungsgeschwindigkeit und eine bessere Skalierbarkeit im Vergleich zu relationalen Datenbanken. Auch wenn viele große Internetunternehmen in bestimmten Big Data-Anwendungsfällen NoSQL-Datenbanken relationalen Datenbanken vorziehen, gibt es allerdings keine fachlich fundierten Aussagen darüber, ob und wie stark sich die konzeptuellen Unterschiede zwischen NoSQL-Datenbanken und relationalen Datenbanken auf den praktischen Einsatz in Big Data-Anwendungsfällen auswirken.

In dieser Problemstellung liegt die Motivation dieses Kapitels. Ziel ist es, Vor- und Nachteile des praktischen Einsatzes von NoSQL-Datenbanken gegenüber relationalen Datenbanken systematisch herauszuarbeiten und somit einen dringend benötigten Beitrag zur Versachlichung der NoSQL-Diskussion zu leisten. Da die Fokussierung dieser Thematik auf Big Data-Anwendungsfällen liegt, stehen hier insbesondere der Umgang mit einer hohen Datenvielfalt, die Verarbeitungsgeschwindigkeit von Anfragen sowie die Skalierbarkeitseigenschaften dieser Systeme im Vordergrund.

6.1 Strukturelle Vielfalt

Eine zentrale Eigenschaft von Big Data ist die große Vielfalt der zu verarbeitenden Daten. Neben strukturierten Informationen mit einem tabellarischen Aufbau verarbeiten Informations- und Kommunikationssysteme heutzutage zunehmend mehr sogenannte unstrukturierte beziehungsweise semistrukturierte Daten [2, 23, 131]. Hierbei können sich einzelne Datenobjekte in ihrem Aufbau voneinander unterscheiden, obwohl sie gleiche oder ähnliche Realweltobjekte repräsentieren. Semistrukturierte Datensätze können sich in der Anzahl, in der Benennung (Homonyme und Synonyme) und in den Typen ihrer Attribute unterscheiden und unterliegen zudem vergleichsweise häufig strukturellen Veränderungen. Meistens besitzen semistrukturierte Daten kein explizites Schema, sondern enthalten Strukturinformationen implizit in den jeweiligen Datenobjekten, weshalb sie auch als selbstbeschreibend bezeichnet werden [148, 152]. Semistrukturierte Daten traten zunächst vorwiegend bei Web Services und anderen Schnittstellen zwischen heterogenen Systemen auf. Inzwischen sind sie jedoch bei vielen modernen E-Commerce- und Web 2.0-Plattformen anzutreffen, bei denen Produkte beziehungsweise benutzergenerierte Inhalte, wie Beiträge, Bilder oder Videos, mit beliebigen Attributen (Tags) beschrieben werden. So können beispielsweise

Fotos auf der Plattform Flickr mit bis zu 75 verschiedenen, vom Benutzer freiwählbaren Tags [59] annotiert und diese daraufhin zur Auswertung und Suche herangezogen werden [26].

Aufgrund ihrer hohen Heterogenität und ihren häufigen strukturellen Änderungen lassen sich semistrukturierte Daten nur sehr schwer in die vorherbestimmten und unflexiblen Strukturen eines relationalen Datenbankschemas abbilden. NoSQL-Datenbanken sind dagegen schemafrei beziehungsweise speichern Strukturinformationen nicht in einem globalen Schema, sondern verwalten diese Informationen lokal in den jeweiligen Datensätzen. NoSQL-Datenbanken versprechen deshalb einen deutlich angenehmeren Umgang mit semistrukturierten Datensätzen.

Dieses Unterkapitel hat das Ziel, fachlich fundierte Aussagen darüber zu treffen, welche Vor- und Nachteile sich durch die Schemafreiheit von NoSQL-Datenbanken im Vergleich zu relationalen Datenbanken bei der Verarbeitung von semistrukturierten Daten ergeben. Hierzu wird mithilfe der in den vorherigen Kapiteln erzeugten Abbildungsregeln das konzeptuelle Datenmodell einer semistrukturierten Entität auf die logischen Datenmodelle von relationalen Datenbanken und NoSQL-Datenbanken überführt. Anschließend werden die erstellten Abbildungsvarianten analysiert und bewertet.

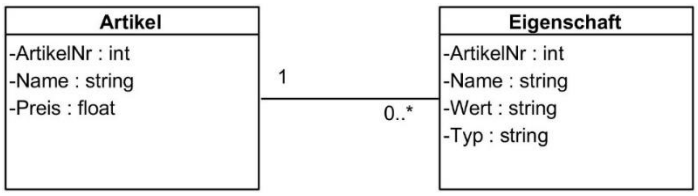


Abbildung 6.2: Konzeptuelles Datenmodell eines Artikels und seiner Eigenschaften

Das zur Veranschaulichung verschiedener Abbildungsmöglichkeiten herangezogene konzeptuelle Datenmodell ist in Abbildung 6.2 dargestellt. Die Entität *Artikel* besteht aus einer festen Anzahl an Attributen (*Name*, *Preis*) und kann optional durch eine variable Anzahl an zusätzlichen Eigenschaften näher beschrieben werden. Eine *Eigenschaft* besitzt einen *Namen*, einen *Wert* und einen *Typ*.

6.1.1 Relationale Datenbanken

Für die Abbildung heterogener Datensätze in relationalen Datenbanken stehen drei verschiedene Optionen zur Verfügung. Im Rahmen dieser Arbeit werden die horizontale Abbildung, die Vererbung und die vertikale Abbildung voneinander unterschieden.

Horizontale Abbildung

Die horizontale Abbildung entspricht dem bereits bekannten Modellierungsmuster, bei dem ein Datensatz durch eine Zeile und die einzelnen Attribute durch jeweils eine Spalte repräsentiert werden (Abbildung 6.3).

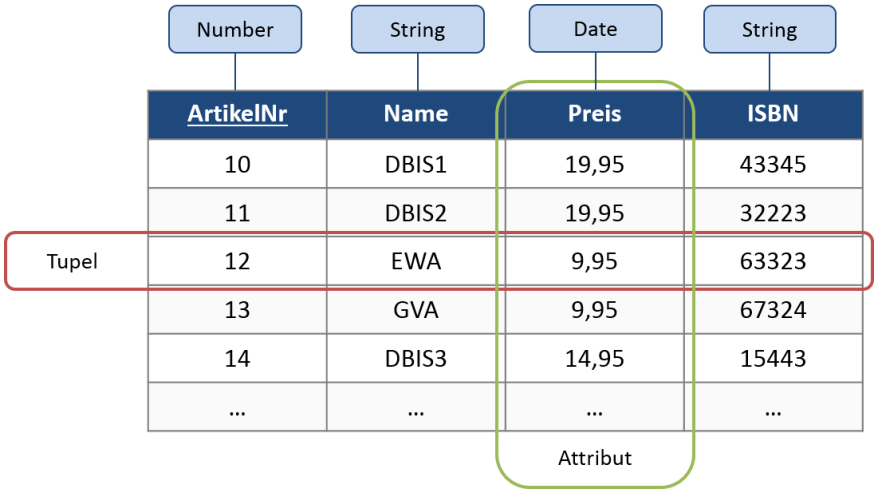


Abbildung 6.3: Horizontale Abbildung der Relation *Artikel*

Diese Abbildungsvariante eignet sich hervorragend für homogene Datensätze, bei denen möglichst viele Datensätze in einer Tabelle die gleichen Eigenschaften besitzen. Werden allerdings heterogene Datensätze, bei denen nur wenige Datensätze gleiche Eigenschaften aufweisen, horizontal abgebildet, kristallisieren sich drei wesentliche Probleme [3] heraus:

Sehr hohe Spaltenanzahl: Semistrukturierte Daten enthalten sehr viele unterschiedliche Attribute, die jeweils in einer eigenen Spalte abgebildet werden müssen. Viele relationale Datenbanksysteme sind jedoch in ihrer maximalen Spaltenanzahl auf circa 1000 Spalten begrenzt [128, 136].

Null-Werte: Jeder Datensatz benötigt nur einen Bruchteil aller verfügbaren Attribute, weshalb viele Attributwerte mit `null` belegt sind. Neben einer Verschwendung von Speicherplatz wird dadurch die Indizierung deutlich erschwert und die Suche verlangsamt.

Schemaevolution: Datensätze mit neuen, zuvor unbekannten Attributen führen zwangsläufig zu Änderungen des Schemas. Da diese Änderungen gerade bei großen Datenmengen mit sehr hohen Kosten verbunden sind, werden sie bei relationalen Datenbanken möglichst vermieden und entsprechend selten durchgeführt.

Abgesehen davon, dass das Schreiben einer SQL-Anfrage durch die extrem hohe Anzahl an Attributen sehr unübersichtlich ist, weisen Anfragen, die auf nur wenige Attribute eines Datensatzes bezogen sind, eine sehr schlechte Performanz auf [4, 184].

ArtikelNr	Name	Preis	ISBN	DOI	ASIN	KB	Volume
10	DBIS1	19,95	43345	*null*	*null*	*null*	*null*
11	DBIS2	19,95	32223	*null*	*null*	*null*	*null*
12	EWA	9,95	63323	*null*	4434	298	*null*
13	GVA	9,95	67324	*null*	*null*	*null*	*null*
14	ACM	14,95	15443	5543	*null*	*null*	44
...	...	...	...	...	...	...	...

Abbildung 6.4: Horizontale Abbildung der Entität *Artikel* mit heterogenen Eigenschaften

Abbildung 6.4 beschreibt eine horizontale Abbildung von heterogenen Datensätzen, die verschiedene Bücher repräsentieren. Neben gedruckten Büchern werden in dieser Tabelle auch digitalisierte Bücher und Konferenzbände verwaltet. Diese verschiedenen Arten von Büchern besitzen unterschiedliche Eigenschaften und führen dadurch zu einer dünnbesetzten Relation mit sehr vielen `null`-Werten. Wird ein Buch eines neuen Typs, wie beispielsweise eine Zeitschrift, eingeführt, muss zwangsläufig das Datenbankschema um neue Spalten erweitert werden.

Vererbung

Angesichts der Nachteile der horizontalen Abbildung wurden in der Vergangenheit einige alternative Konzepte vorgestellt, mit denen sich semistrukturierte Daten besser im relationalen Datenmodell abbilden lassen. Eine in der Praxis häufig angewendete Variante stellt die Vererbung dar. Hierbei wird der homogene Teil der Datensätze in einer eigenen Tabelle gespeichert. Die Attribute mit heterogenen Daten werden klassifiziert und in separaten Tabellen gespeichert. Für das Beispiel in Abbildung 6.4 entstünden die Relationen *Literatur*, *Artikel* und *E-Book* (Abbildung 6.5).

Die Vererbung ermöglicht gegenüber der horizontalen Abbildung eine effizientere Verwaltung von heterogenen Datensätzen, weil die Anzahl der `null`-Werte deutlich reduziert werden kann. Ist jedoch die Heterogenität zu groß, kann ein Datenmodell entstehen, das sehr viele verschiedene Tabellen mit sehr wenigen Datensätzen beinhaltet. Abgesehen davon, dass beim Einfügen eines Datensatzes mit neuen Attributen erneut

eine Schemaänderung oder eine neue Spezialisierung notwendig ist, steigen die Komplexität und die Kosten, die beim Auslesen eines Datensatzes benötigt werden. Der Grund hierfür liegt darin, dass Verbundoperationen vor allem in Verbindung mit Magnetplatten zu den teuersten Operationen gehören, die man in relationalen Datenbanken ausführen kann. Magnetplatten haben die Eigenschaft, dass die Suche nach einem Datensatz deutlich länger dauert als das eigentliche Auslesen der Daten. Bei einer Rotationsgeschwindigkeit von 7.200 Umdrehungen pro Minute liegt die Zugriffszeit durchschnittlich bei 13 Millisekunden. Bei einer Übertragungsrate von 125 Megabyte pro Sekunde werden für das reine Lesen eines 1024 Byte großen Datensatzes dagegen nur 8 Mikrosekunden benötigt. Dieser Wert beträgt weniger als 0,1 Prozent der Zugriffszeit[46]. Soll einer der in Abbildung 6.5 dargestellten Artikel aus dem persistenten Speicher ausgelesen werden, verdoppelt sich demnach die Zugriffszeit im Vergleich zu der Relation in Abbildung 6.4. Wesentlich teurer werden Verbundoperationen, wenn sich die benötigten Daten auf unterschiedlichen Knoten in einem verteilten Datenbanksystem befinden.

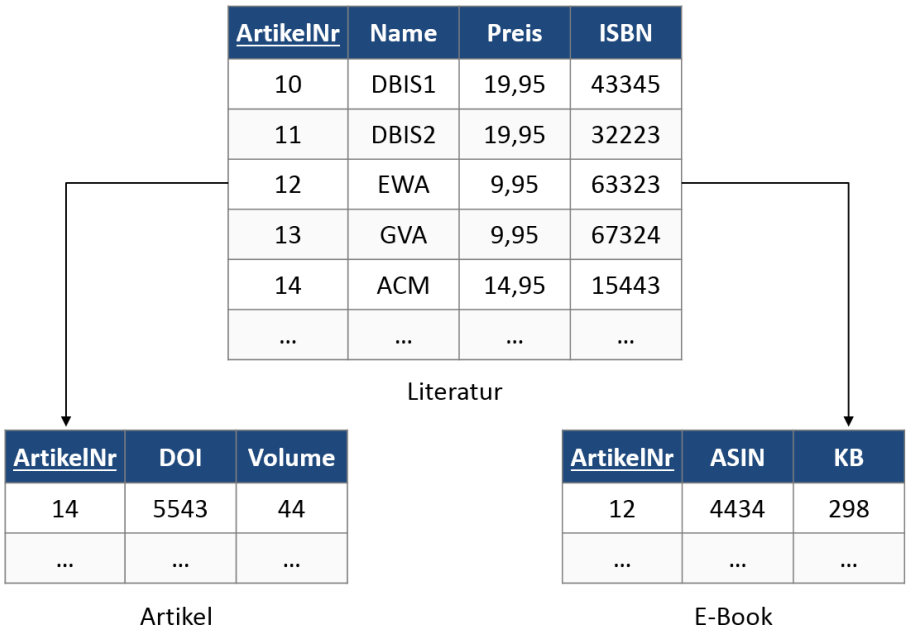


Abbildung 6.5: Abbildung der Entitäten *Literatur*, *Artikel* und *E-Book* mittels Vererbung

[46] http://de.wikipedia.org/wiki/Festplatte#Geschwindigkeit: Seagate Barracuda 7200.12

Vertikale Abbildung

Heterogene Datensätze werden häufig auch mithilfe von Anti-Pattern wie dem EAV-Model (Entity-Attribute-Value) in relationalen Datenbanken gespeichert. Bei diesem Modellierungsmuster werden die einzelnen Datensätze entgegen des bewährten Ansatzes nicht horizontal, sondern vertikal in den Relationen abgebildet.

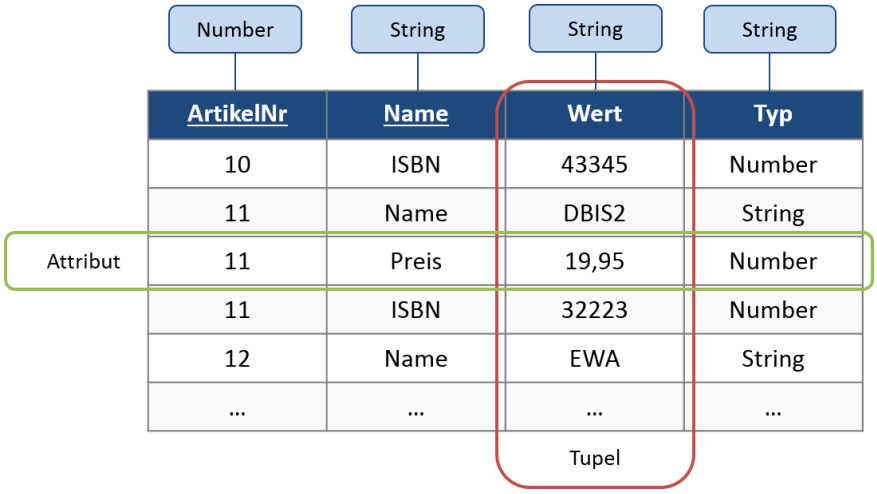


Abbildung 6.6: Vertikale Abbildung der Relation *Artikel*

Bei der vertikalen Abbildung werden Attribute nicht als Spalten, sondern als Datensätze dargestellt. Entsprechend besteht eine Relation lediglich aus drei Spalten, bei denen ein Attributwert durch einen zusammengesetzten Primärschlüssel aus Objekt-ID und Attributname adressiert wird. Da sämtliche Werte in der Spalte *Wert* den gleichen Datentyp besitzen müssen, kann optional, wie in Abbildung 6.6 dargestellt, eine weitere Spalte in die Relation aufgenommen werden, welche den jeweiligen Attributtyp beinhaltet. Die Relation entspricht dann exakt dem konzeptuellen Modell der Entität *Eigenschaften* (Abbildung 6.2).

ArtikelNr	Name	Wert	Typ
12	ASIN	43345	Number
12	KB	DBIS2	Number
13	ISBN	67324	Number
14	DOI	5543	Number
14	Volume	44	Number
15	Ausgabe	04/2013	String
...	.	...	...

Abbildung 6.7: Vertikale Abbildung der Relation *Artikel* mit heterogenen Eigenschaften

Durch die vertikale Abbildung von heterogenen Datensätzen können die oben genannten Nachteile der horizontalen Abbildung vermieden werden. Die drei- beziehungsweise vierspaltige Tabelle enthält keine `null`-Werte und ist demzufolge dichtbesetzt (Abbildung 6.7). Obendrein können neue Attribute, wie beispielsweise die Ausgabenummer einer Zeitschrift, mit einer einfachen Einfüge-Operation zur Laufzeit ohne Schemaänderungen gespeichert werden. Die vertikale Abbildung bringt allerdings eine Reihe von Nachteilen mit sich:

Komplexe und teure Anfragen: Die Verteilung eines Datenobjekts auf jeweils mehrere Tupel verursacht bei dessen Rekonstruktion einen hohen Aufwand. Selbst vergleichsweise einfache Anfragen müssen mithilfe komplexer und fehleranfälliger SQL-Statements mit vielen Verbundoperationen formuliert werden [1, 16]. Die Ausführungskosten zur Rekonstruktion eines Datensatzes liegen dabei wegen der hohen Anzahl an Joins um ein Vielfaches höher als bei der Vererbung.

Keine Datentypen: In der vertikalen Darstellung entsprechen alle Attributwerte dem gleichen Datentyp. Ist eine Unterstützung von verschiedenen Datentypen notwendig, müssen diese über eine weitere Spalte gekennzeichnet und zur Laufzeit auf Applikationsebene aufwendig interpretiert werden. Alternativ kann die Tabelle pro Datentyp eine Spalte besitzen, in welche die zugehörigen Attributwerte eingetragen werden. Nicht benötigte Datentypen müssen dann jedoch mit `null` belegt werden.

Redundantes Schema: Durch die vertikale Abbildung von Datensätzen müssen Metainformationen, wie Attributnamen oder Attributtyp, für jedes Attribut redundant gespeichert werden. Infolgedessen erhöhen sich Speicher- und Wartungskosten.

Die Vor- und Nachteile vertikaler Tabellen wurden bereits in verschiedenen Anwendungsfällen evaluiert. Neben der Speicherung von XML-Daten [61, 169] ist an dieser Stelle vor allem die Verwendung von relationalen Datenbanken als persistente Speicher für RDF-Triple erwähnenswert [21, 32, 81, 183]. Hierbei stellte sich heraus, dass die naheliegende Abbildung von RDF-Triple in einer ternären vertikalen Relation anderen Modellierungsmustern [57, 176] unterlegen ist, da selbst einfache Anfragen sehr schnell zu komplexen und langsamen Verbundoperationen auf der vertikalen Tabelle führen können [1, 168]. Darüber hinaus weist dieser Ansatz deutliche Skalierbarkeitsprobleme auf [182].

Sowohl der horizontale als auch der vertikale Ansatz haben demnach ihre Vor- und Nachteile. Ab wann welcher Ansatz dem anderen vorgezogen werden sollte, hängt von vielen Faktoren ab. Hierzu zählen vor allem der Belegungsgrad mit `null`-Werten, die Anzahl der Spalten sowie die erwartete Ergebnismenge der Projektionen und Restriktionen. Betrachtet man verschiedene Benchmarks [3, 38] lässt sich dennoch die allgemeine Tendenz erkennen, dass bei einer sehr hohen `null`-Wertdichte und einer sehr geringen Anzahl an projizierten Attributen der vertikale dem horizontalen Ansatz vorzuziehen ist. Falls jedoch abschließend ein Ergebnis einer vertikalen Anfrage in der

von vielen Applikationen und Benutzern erwarteten horizontalen Darstellung präsentiert werden muss, entsteht ein zusätzlicher Overhead, der die Performanzgewinne des vertikalen Ansatzes wieder eliminieren kann [16].

Auch wenn sich mit den hier vorgestellten Verfahren semistrukturierte Daten im relationalen Datenmodell abbilden lassen, ist dennoch festzuhalten, dass relationale Datenbanken Schwächen im Umgang mit semistrukturierten Daten offenbaren. Dieser Umstand motivierte zur Jahrtausendwende nicht nur die Entwicklung von XML-Datenbanken, sondern ist auch ein häufig angeführtes Argument für den Einsatz von NoSQL-Datenbanken.

6.1.2 Key Value Stores

In Kapitel 3 wurden zwei verschiedene Modellierungskonzepte für die Abbildung von Entitäten in Key Value Stores vorgestellt. Ebenso wie bei relationalen Datenbanken unterscheiden sich diese beiden Konzepte hinsichtlich der horizontalen oder vertikalen Abbildung.

Atomare Abbildung

Bei der atomaren Abbildung werden Datensätze auf jeweils ein Schlüssel-Wert-Paar abgebildet. Das in Abbildung 6.8 dargestellte Bucket *Artikel* enthält dementsprechend einen Eintrag für jeden Artikel.

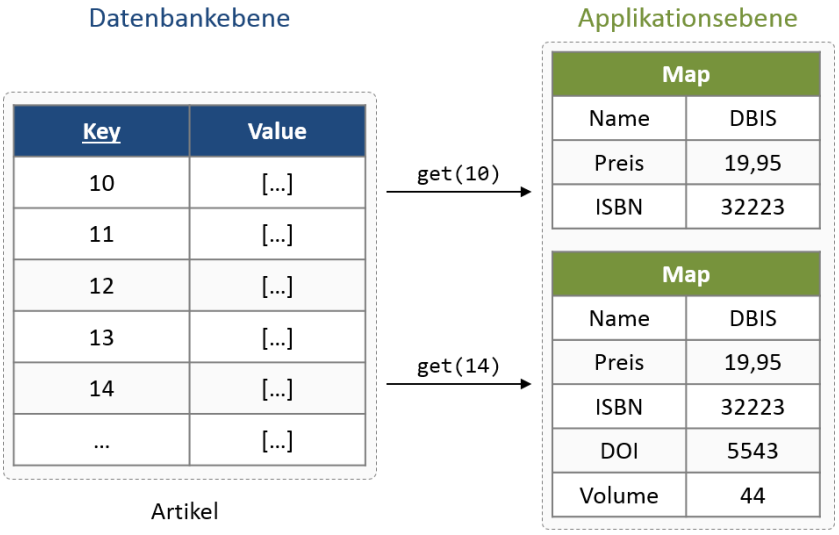


Abbildung 6.8: Atomare Abbildung des Buckets *Artikel*

Key Value Stores besitzen keinerlei Kenntnis über die Inhalte der in ihnen gespeicherten Schlüssel-Wert-Paare und sind auch nicht in der Lage, Restriktionen bezüglich der Modellierung eines Datensatzes zu verarbeiten. Key Value Stores sind deshalb praktisch schemafrei. Aus diesem Grund ist es bei der Abbildung von Datensätzen als atomare Einheiten absolut unerheblich, ob homogene oder heterogene Datensätze verarbeitet werden sollen. Jedes Schlüssel-Wert-Paar kann völlig unterschiedlich strukturierte Datensätze enthalten. Die bei der horizontalen Abbildung in relationalen Datenbanken auftretenden Nachteile sind daher bei Key Value Stores nicht festzustellen.

Der Nachteil der atomaren Abbildung bei Key Value Stores liegt in der eingeschränkten Anfragemächtigkeit. Da Key Value Stores ausschließlich einen primärschlüsselbasierten Datenzugriff ermöglichen, können einzelne Attribute eines Datensatzes bei dieser Modellierungsvariante nur mit einem entsprechenden Mehraufwand in der Applikationslogik ausgelesen und verarbeitet werden.

Fragmentierte Abbildung

Bei der fragmentierten Abbildung wird eine Entität auf mehrere Datensätze verteilt. Ähnlich der vertikalen Abbildung in relationalen Datenbanken wird jedes Attribut eines Datensatzes durch ein Schlüssel-Wert-Paar repräsentiert (Abbildung 6.9).

Key	Value
12:ASIN	43345
12:KB	DBIS2
13:ISBN	67324
14:DOI	5543
14:Volume	44
15:Ausgabe	04/2013
...	...

Artikel

Abbildung 6.9: Fragmentierte Abbildung des Buckets *Artikel*

Die fragmentierte Abbildung in Key Value Stores verfolgt jedoch eine andere Zielsetzung als die vertikale Abbildung in relationalen Datenbanken. Hier steht nicht ein vereinfachter Umgang mit semistrukturierten Daten im Vordergrund, sondern eine höhere Anfragemächtigkeit. Durch die Abbildung einzelner Attribute auf jeweils ein Schlüssel-Wert-Paar können diese im Gegensatz zur atomaren Abbildung direkt angesprochen und verarbeitet werden. Allerdings lassen sich bei dieser Modellierungsvariante komplexere Anfragen, wie die Rekonstruktion eines mehrere Attribute umfassen-

den Datensatzes oder die Beziehungsauflösung zwischen verschiedenen Datenobjekten, nur sehr umständlich und mit einem hohen Aufwand auf Applikationsebene realisieren. Der Grund hierfür liegt vor allem an den nicht vorhandenen Verbundoperationen, welche eine manuelle Verarbeitung von verknüpften Datenobjekten nach sich ziehen. Diese umfangreichen Anfragen sind wegen ihrer Komplexität nicht nur fehleranfälliger und wartungsintensiver, sie verlieren durch die gestiegene Anzahl an Operationen auch den ursprünglichen Performanzvorteil. Außerdem werden bei diesem Ansatz Attributnamen redundant gespeichert, was wiederum den Speicherplatzbedarf und die Wartungskosten erhöht.

Auch wenn sich die beiden hier vorgestellten Abbildungsvarianten in den unterstützten Anfragearten deutlich voneinander unterscheiden, stellen sich angesichts der Schemafreiheit von Key Value Stores keine Unterschiede in der Datenverarbeitung von homogenen und heterogenen Datensätzen heraus.

6.1.3 Document Stores

Document Stores verwalten im Gegensatz zu relationalen Datenbanken kein globales Schema. Da einzelne Datensätze demzufolge keiner vorgegebenen Struktur unterworfen sind, ermöglichen Document Stores eine maximale Flexibilität in der Datenmodellierung. Document Stores sind jedoch im Gegensatz zu Key Value Stores, die ebenfalls eine maximale Flexibilität erlauben, in der Lage, ihre selbstbeschreibenden Datensätze zu interpretieren. Dadurch können Document Stores eine ähnliche Mächtigkeit bei der Anfrageverarbeitung wie relationale Datenbanken aufweisen. Document Stores verbinden somit die Flexibilität von Key Value Stores mit der Modellierungsmächtigkeit von relationalen Datenbanken. Dank des flexiblen und zugleich mächtigen Datenmodells sind diese Systeme wie geschaffen für die Verarbeitung von heterogenen Datensätzen.

Abbildungsregeln zur Überführung des konzeptuellen Datenmodells in das logische Datenmodell eines Document Stores wurden bereits in Kapitel 4 vorgestellt. Entsprechend dieser Regeln wird eine starke Entität meist durch ein eigenes Dokument repräsentiert. Schwache Entitäten werden mithilfe der Aggregation in starke Entitäten eingebettet. Unter Berücksichtigung dieser Abbildungsregeln gibt es für das in Abbildung 6.2 dargestellte Datenmodell eines Artikels und seiner Eigenschaften nur eine mögliche Modellierungsvariante. Bei dieser wird jeder Artikel auf ein Dokument abgebildet und die zugehörigen heterogenen Eigenschaften als Attribute in diesem Dokument hinterlegt.

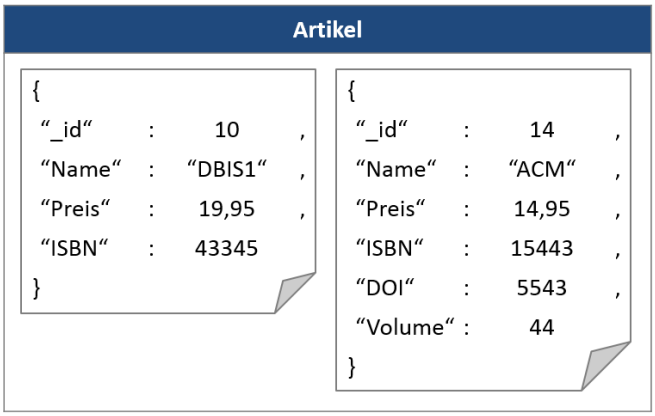


Abbildung 6.10: Darstellung zweier heterogener Artikel

In Abbildung 6.10 ist das logische Datenmodell zweier heterogener Artikel mit den Artikelnummern (*_id*) 10 und 14 dargestellt. Jedes Dokument enthält hierbei nur die Attribute, die für die Beschreibung des Realweltobjekts auch wirklich von Bedeutung sind. Da Dokumente keinem globalen Schema genügen müssen, ist eine Abbildung von `null`-Werten wie bei relationalen Datenbanken nicht notwendig. Neue Datensätze mit neuen Attributen können problemlos eingefügt werden.

Diese Flexibilität hat jedoch den Nachteil, dass für jeden Datensatz explizit das zugehörige Schema gespeichert werden muss. Die redundante Speicherung von Attributnamen führt dabei nicht nur zu einem erhöhten Speicherplatzbedarf, sondern hat auch sehr hohe Änderungskosten für globale Schemaänderungen zur Folge. Ändert sich beispielsweise der Name oder der Typ eines Attributs, das von sehr vielen Dokumenten verwendet wird, müssen diese Änderungen mit einem entsprechend hohen Aufwand in allen Dokumenten durchgeführt werden. Diese Änderungen können jedoch asynchron im Hintergrund erfolgen und beeinflussen im Gegensatz zu Schemaänderungen in relationalen Datenbanken nicht den produktiven Betrieb.

6.1.4 Column Family Stores

Datensätze in Column Family Stores bestehen aus mehreren Schlüssel-Wert-Paaren und werden über einen eindeutigen Schlüssel identifiziert. Wie bereits während der Analyse des Prädikatorenschemas in Kapitel 5.2.1 deutlich wurde, besitzen Datenmodelle von Column Family Stores eine ähnliche Flexibilität und Mächtigkeit wie die von Document Stores verwendeten JSON-Dokumente.

Datensätze in Column Family Stores sind selbstbeschreibend und müssen keinem globalen Schema entsprechen. Strukturelle Unterschiede zwischen den einzelnen in einer Tabelle gespeicherten Datensätzen sind deswegen irrelevant für die Datenverarbeitung. Obwohl sich die in Abbildung 6.11 dargestellten Datensätze mit den Primärschlüsseln 12 und 14 in ihrem Aufbau deutlich von den Datensätzen 10, 11 und 13

unterscheiden sind keine der Nachteile, wie Speicherplatzverschwendung oder Schemaänderung, zu verzeichnen, die bei relationalen Datenbanken auftreten.

RowKey	Artikel				
10	ISBN : 43345	Name : DBIS1	Preis : 19,95		
11	ISBN : 32223	Name : DBIS2	Preis : 19,95		
12	ASIN : 4438	ISBN : 63323	KB : 298	Name : EWA	Preis : 19,95
13	ISBN : 67324	Name : GVA	Preis : 9,95		
14	DOI : 5543	ISBN : 15443	Name : ACM	Preis : 19,95	Volume : 44
...	...				

Abbildung 6.11: Abbildung von heterogenen Artikeln

Column Family Stores eigenen sich deshalb sehr gut für die Verarbeitung von semistrukturierten Daten. Neben den bereits bei Document Stores erwähnten Nachteilen eines lokalen Schemas gegenüber einem globalen Schema müssen bei Column Family Stores vor allem die deutlich geringere Anfragemächtigkeit und die fehlenden Datentypen berücksichtigt werden.

6.1.5 Zusammenfassung

Semistrukturierte Daten lassen sich mithilfe der in dieser Arbeit entwickelten Abbildungsregeln sowie bekannten Modellierungsmustern aus der relationalen Datenbanktheorie auf unterschiedliche Art und Weise auf die Datenmodelle der verschiedenen Datenbanksysteme abbilden. Bei der Analyse der Vor- und Nachteile der einzelnen Abbildungsvarianten kristallisierten sich vier grundlegende Eigenschaften heraus, welche sowohl die Mächtigkeit des Datenbanksystems als auch den Mehraufwand auf Applikationsebene entscheidend beeinflussen:

Lokales Schema: Semistrukturierte Daten weisen eine sehr hohe strukturelle Heterogenität untereinander auf und lassen sich daher nur sehr schwer in ein vorherbestimmtes, globales Schema zwängen. Die lokale Verwaltung von Schemainformationen mittels selbstbeschreibender Datensätze erlaubt dagegen eine wesentlich einfachere Abbildung dieser Daten.

Atomare Datensätze: Semistrukturierte Entitäten werden bei manchen Modellierungsmustern nicht atomar auf jeweils einen Datensatz abgebildet, sondern auf mehrere Datensätze verteilt, um eine höhere Flexibilität und Ausdrucksstärke zu erzielen. Allerdings steigen hierdurch die Kosten zur Rekonstruktion einer Entität. Vor allem bei Datenbanksystemen, die keine Verbundoperationen unterstützen, kann durch die

Fragmentierung der Datensätze ein sehr hoher Aufwand auf Applikationsebene entstehen.

Restriktionen auf Attribute: Manche Abbildungsvarianten ermöglichen eine atomare Abbildung von semistrukturierten Entitäten auf Kosten der Anfragemächtigkeit. In diesen Fällen können Restriktionsbedingungen ausschließlich auf dem Primärschlüssel und nicht auf beliebigen Attributen eines Datensatzes ausgeführt werden.

Datentypen: Die Datenmodelle und vorgestellten Modellierungsmuster unterscheiden sich ferner in ihren Möglichkeiten Datentypen abzubilden. In vielen Fällen werden diese als einfache Strings oder Byte-Arrays verwaltet und müssen anschließend auf Applikationsebene transformiert werden.

Die unterschiedlichen Abbildungsvarianten und die sich hinsichtlich der vier Eigenschaften ergebenden Vor- und Nachteile für relationale Datenbanken (RDBMS), Key Value Stores (KVS), Document Stores (DOC) und Column Family Stores (CFS) sind in Tabelle 6.1 aufgeführt.

Datenbankklasse	RDBMS		KVS		DOC	CFS
Modellierungs-muster	Horizontal	Vertikal	Atomar	Fragmentiert	Dokument	Horizontal
Lokales Schema	✗	✓	✗	✓	✓	✓
Atomare Datensätze	✓	✗	✓	✗	✓	✓
Attributrestriktion	✓	✓	✗	✓	✓	✗
Datentypen	✓	✗	✗	✗	✓	✗

Tabelle 6.1: Vor- und Nachteile der unterschiedlichen Abbildungsvarianten

Relationale Datenbanken verwalten die Strukturen der in ihnen gespeicherten Datensätze zentral in einem globalen Schema. Infolgedessen lassen sich vor allem homogene Datensätze äußerst effizient in diesen Systemen verarbeiten. Heterogene Datensätze mit einer hohen Änderungsrate lassen sich dagegen nur schwer mit dem sehr restriktiven und unflexiblen relationalen Datenmodell in Einklang bringen. Eine hohe Anzahl an `null`-Werten, kostspielige Schemaänderungen und eine auf maximal 1000 Attribute limitierte Heterogenität sind die Folge. Diese Nachteile lassen sich mit einer vertikalen Abbildung der Datensätze und einer damit verbundenen lokalen Speicherung der Schemainformationen vermeiden. Allerdings müssen bei dieser Variante einzelne Datensätze aufwendig durch Verbundoperationen zusammengesetzt und Datentypen applikationsseitig rekonstruiert werden. Die vertikale Abbildung ist demnach nur bei einer sehr hohen Heterogenität und bei einer geringen Anzahl an Attributen pro Datensatz der horizontalen Abbildung vorzuziehen.

Key Value Stores verfügen über ein sehr minimalistisches Datenmodell, bei dem die Datenstruktur der einzelnen Datensätze nicht durch ein globales Schema vorherbestimmt ist. Dieses Datenmodell besitzt allerdings gegenüber einer zweispaltigen relationalen Tabelle keine Vorteile. Berücksichtigt man die minimalistische Anfrageunterstützung von Key Value Stores, sind diese Systeme relationalen Datenbanken bei der Abbildung und Verarbeitung von semistrukturierten Daten auf konzeptueller Ebene sogar deutlich unterlegen. Während bei der atomaren Abbildung auf die Attribute eines Datensatzes nicht zugegriffen werden kann, müssen bei der fragmentierten Abbildung infolge fehlender Verbundoperationen einzelne Datensätze mit einem hohen Aufwand applikationsseitig rekonstruiert werden. Datentypen werden in beiden Abbildungsmustern nicht unterstützt.

Document Stores sind dank ihres flexiblen Datenmodells und ihrer hohen Anfragemächtigkeit sehr gut für die Verarbeitung von heterogenen Datensätzen geeignet. Durch die Verwendung von selbstbeschreibenden Dokumenten unterliegen einzelne Datensätze nicht den Restriktionen eines globalen Schemas. Aus diesem Grund ist es theoretisch möglich, dass sich die Datensätze innerhalb einer Collection völlig voneinander unterscheiden. Neben den gängigen primitiven Datentypen unterstützen Document Stores auch die Datentypen `Array`, `Map` und `Object`, mit denen eine applikationsnahe, natürliche Datenmodellierung ermöglicht wird. Da Document Stores in der Lage sind, die Metainformationen innerhalb der Dokumente zu interpretieren, können auch sehr komplexe Anfragen datenbankseitig verarbeitet werden. Zur Formulierung dieser Anfragen stellen Document Stores einen sehr großen Umfang an Operationen und Operatoren bereit (siehe Kapitel 4.2).

Column Family Stores bilden Entitäten auf jeweils einen Datensatz mit mehreren freiwählbaren Schlüssel-Wert-Paaren ab. Diese unterliegen bis auf einen Column Family Namen keinen weiteren globalen Restriktionen und können sich deswegen zwischen den einzelnen Datensätzen voneinander unterscheiden. Allerdings können Column Family Stores keine Restriktionsbedingungen auf diesen Schlüssel-Wert-Paaren ausführen. Es werden auch keine Datentypen datenbankseitig aufgelöst. Column Family Stores sind darum relationalen Datenbanken nur in der Verarbeitung von heterogenen Datensätzen überlegen, wenn diese ausschließlich über ihren Primärschlüssel angesprochen werden und keine Datentypunterstützung gewünscht ist.

Die Verarbeitung von semistrukturierten Daten ist eine der zentralen Anforderungen, die Anwendungsfälle im Big Data-Umfeld an die zugrunde liegenden Datenbanksysteme stellen. In vielen Fällen wird das relationale Datenmodell als zu restriktiv kritisiert und die Verwendung einer flexibleren NoSQL-Lösung bevorzugt. Diese Systeme verwalten Schemainformationen lokal in jedem Datensatz und versprechen deshalb eine bessere Verarbeitung von semistrukturierten Daten. Betrachtet man allerdings die hier erzielten Ergebnisse und berücksichtigt obendrein die sehr große Anfragemächtigkeit von relationalen Datenbanksystemen, stellen einzig Document Stores eine lohnenswerte Alternative zu relationalen Datenbanken dar. Key Value Stores und Co-

lumn Family Stores sind dagegen auf sehr spezielle Anwendungsfälle, wie die Bereitstellung von spezifischen Datenstrukturen oder die Verarbeitung sehr großer Datenmengen, ausgerichtet und besitzen bei der einfachen Verarbeitung von semistrukturierten Daten keine Vorteile gegenüber relationalen Datenbanken.

6.2 Hohe Geschwindigkeit

Sehr hohe Geschwindigkeitsanforderungen sind ein weiteres charakteristisches Merkmal für Informations- und Kommunikationssysteme in Big Data-Anwendungsfällen. Häufig sollen Anfragen ohne spürbare Verzögerungen verarbeitet und Daten in Echtzeit wieder ausgegeben werden. Typischerweise beinhalten diese Anfragen sehr einfache Operationen, bei denen wenige Attribute umfassende Datensätze abgelegt oder über deren Primärschlüssel abgerufen werden. Komplexe Anfragen über mehrere Datensätze hinweg sind im Bereich des Online Transaction Processings (OLTP) unüblich und eher im Bereich des Online Analytical Processings (OLAP) zu finden.

Entscheidenden Einfluss auf die Verarbeitungsgeschwindigkeit einer Anfrage hat die Leistungsfähigkeit des Datenbanksystems. Je höher diese ist, desto mehr Anfragen können innerhalb eines tolerierbaren Zeitintervalls verarbeitet werden (Durchsatz). Übersteigt jedoch der Durchsatz die Leistungsfähigkeit eines Datenbanksystems, erhöht sich automatisch die Zeit, die für die Verarbeitung einer einzelnen Anfrage benötigt wird (Latenz). Da Anwender bereits auf wenige Millisekunden Verzögerung äußerst sensibel reagieren [22, 104], kann der Zusammenhang zwischen Durchsatz und Latenz große finanzielle Schäden sowie einen Vertrauensverlust bei den Benutzern nach sich ziehen [45]. Aus diesen Gründen ist die Leistungsfähigkeit häufig ein essenzielles Entscheidungskriterium bei der Wahl eines geeigneten Datenbanksystems.

Zwei zentrale Komponenten von relationalen Datenbanken sind das ACID-kompatible Transaktionskonzept und die mächtige Anfragesprache SQL. Der Einsatz dieser Komponenten ermöglicht relationalen Datenbanken eine zuverlässige und in vielen Anwendungsfällen einsetzbare Datenverarbeitung. Ein Nachteil dieser Komponenten ist allerdings, dass sie bei einfachen Lese- und Schreiboperationen, wie sie im OLTP von Big Data-Anwendungsfällen typisch sind, einen Overhead verursachen können, welcher wiederum die Leistungsfähigkeit negativ beeinflusst. Die meisten NoSQL-Datenbanken verzichten auf ein ACID-kompatibles Transaktionskonzept und eine komplexe Anfragesprache, um eine möglichst hohe Verarbeitungsgeschwindigkeit zu erzielen.

Um fachliche fundierte Aussagen über die unterschiedliche Leistungsfähigkeit von relationalen Datenbanken und NoSQL-Datenbanken treffen zu können, werden im Rahmen dieses Unterkapitels die Messergebnisse verschiedener Benchmarks herangezo-

gen. Mithilfe der in dieser Arbeit herausgearbeiteten konzeptuellen Grundlagen werden diese Messergebnisse daraufhin interpretiert und somit die Vor- und Nachteile der verschiedenen Datenbanksysteme in Big Data-Anwendungsbereichen herausgestellt.

6.2.1 Benchmarks

Benchmarks sind ein beliebtes Mittel, um die Leistungsfähigkeit von Computersystemen messen und nach bestimmten Kriterien vergleichen zu können. In der Regel werden hierzu komplexe Problemstellungen formuliert, die das Computersystem lösen muss. Die Formulierung dieser Problemstellungen hat vor allem bei Benchmarks von Datenbanksystemen einen sehr hohen Einfluss auf die erzielten Messergebnisse, da die Leistungsfähigkeit eines Datenbanksystems von sehr vielen anwendungsfallspezifischen Faktoren abhängt. Neben der Menge und der Struktur der Daten bestimmen vor allem die Art und der Umfang der durchgeführten Lese- und Schreiboperationen die Aussagen eines Benchmarks. Zusätzlich beeinflussen die Anzahl der konkurrierenden Zugriffe, unterschiedliche Datenbankkonfigurationen, die Versionen der Datenbanksysteme und vor allem die eingesetzte Hardware die Ergebnisse eines Leistungstests.

Aus diesen Gründen sind Ergebnisse eines Benchmarks nur aussagekräftig, wenn die Problemstellung exakt dem praktischen Anwendungsfall entspricht, für den verschiedene Datenbanksysteme verglichen werden sollen. Ein System, das sehr gute Durchsatzwerte bei einem Benchmark mit sehr vielen einfachen Schreiboperationen erzielt, kann für einen Anwendungsfall, in dem häufig umfangreiche Leseanfragen über Schlüsselwertbereiche durchgeführt werden, völlig unbrauchbar sein. Zudem ist es möglich, dass dieses System die hohen Schreibraten dank abgeschwächter Dauerhaftigkeitsgarantien erzielt hat, was ebenfalls den Anforderungen des Anwendungsfalls widersprechen kann. In Anbetracht dieser Vielfalt an Einflussfaktoren gestaltet sich nicht nur die Suche nach einem passenden Benchmark als sehr schwierig, die Ergebnisse können auch sehr leicht bewusst verfälscht werden. So ist nahezu jeder Datenbankanbieter in der Lage, Benchmarks bereitzustellen, welche die Überlegenheit seiner eigenen Datenbanksysteme „belegen“.

Um eine einheitliche und objektive Bewertung der Leistungsfähigkeit von Datenbanksystemen zu ermöglichen, wurde bereits 1988 das Transaction Processing Performance Council (TPC)[47] gegründet. Hierzu wurden verschiedene, standardisierte Benchmarks erstellt, welche die Anforderungen der am häufigsten auftretenden Anwendungsfälle simulieren. Zu den bekanntesten Benchmarks zählen:

[47] http://www.tpc.org

- TPC-APP: B2B-Geschäftsvorgänge,
- TPC-C: OLTP-Handelsunternehmen,
- TPC-E: OLTP-Börsenhandel,
- TPC-H: OLAP.

Trotz dieser Standards sollten auch TPC-Benchmark-Ergebnisse durchaus kritisch betrachtet werden. Ein Blick auf die Top-Ten-Liste[48] des TPC-C-Benchmarks macht deutlich, wie sehr die Ergebnisse weiterhin von der eingesetzten Hardware und der Konfiguration der Datenbanksysteme abhängen. Neben den verschiedenen Plattformen, auf denen die Datenbanksysteme laufen, können verschiedene Cache-Größen und zuvor optimierte Ausführungspläne die Ergebnisse sehr leicht verfälschen [24] und erschweren so den Vergleich zwischen den einzelnen Datenbanksystemen.

Auch wenn TPC-Benchmarks gerne zur Bewertung von relationalen Datenbanken herangezogen werden, sind sie nicht für die Messung der Leistungsfähigkeit von NoSQL-Datenbanken geeignet. TPC-Benchmarks simulieren verschiedene Anfragen, die typischerweise in einer Domäne auftreten. Hierbei handelt es sich um umfangreiche Lese- und Schreiboperationen auf mehreren Relationen. NoSQL-Datenbanken sind jedoch nicht für die Verarbeitung verschiedenster Anfragen auf mehreren Collections ausgelegt, sondern auf die performante Ausführung einer bestimmten Anfrageart spezialisiert. So sind manche Systeme geeignet, hohe Schreibraten sequenziell zu verarbeiten und andere wiederum können sehr schnell einen beliebigen Datensatz auslesen.

Um eine Aussage darüber treffen zu können, welche Anfrageart sich besser oder schlechter von welchem Datenbanksystem verarbeiten lässt, bietet sich der Yahoo! Cloud Serving Benchmark (YCSB) an. Bei diesem handelt es sich um einen standardisierten OLTP-Benchmark, bei dem sehr einfache Lese- und Schreiboperationen auf den Datensätzen einer einzigen Tabelle beziehungsweise Collection ausgeführt werden. Jeder Datensatz dieser Tabelle besitzt F Attribute. Der Wert eines Attributs besteht aus einer zufälligen ASCII-Zeichenkette der Länge L. In der Standardkonfiguration werden Datensätze mit einer Größe von einem Kilobyte durch die Parameter $F = 10$ und $L = 100$ erzeugt. Abhängig vom eingesetzten Datenbanksystem werden die einzelnen Datensätze als horizontale Tupel (Relationale Datenbank), atomare Werte (Key Value Stores), Dokumente (Document Stores) oder Reihen (Column Family Store) modelliert. Abbildung 6.12 beschreibt den Aufbau eines YCSB-Datensatzes in einem Key Value Store.

[48] http://www.tpc.org/tpcc/results/tpcc_perf_results.asp

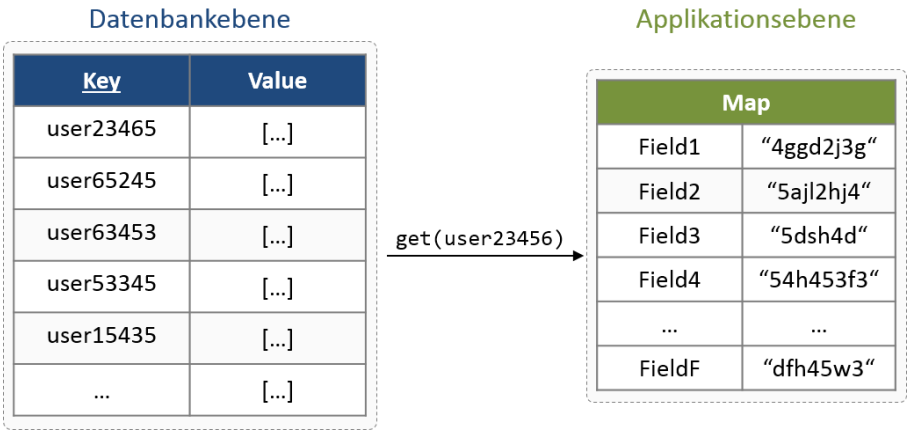


Abbildung 6.12: Aufbau eines YCSB-Datensatzes in einem Key Value Store

Auf diesen Datenmodellen können mehrere sogenannte Workloads ausgeführt werden. Ein Workload besteht aus einer Kombination verschiedener Operationen, die typisch für einen bestimmten Anwendungsfall sind. Innerhalb eines Workloads können neue Datensätze eingefügt, Attributwerte bestehender Datensätze geändert und Attributwerte einzelner Datensätze ausgelesen werden. Der prozentuale Anteil der jeweiligen Operationen innerhalb eines Workloads kann individuell definiert werden.

Eigenschaft	Rabl M Benchmark 1	Rabl D Benchmark 2	Datastax Benchmark 3	Bushik Benchmark 4	Cooper Benchmark 5
Clustergröße	1	8	1	4	6
Prozessoren	8	4	4	8	8
Hauptspeicher	16 GB	4 GB	15 GB	15 GB	8 GB
Tabellengröße	700 MB	10,5 GB	30 GB	25 GB	20 GB
Last	Maximal	Maximal	Maximal	Steigend	Steigend
Cassandra	1.0	1.0	1.1	1.0	0.6 (OP)
HBase	0.9	0.9	0.9	0.9	0.2
MySQL	5.5	-	-	5.5	5.1
MongoDB	-	-	2.2	2.0	-
Redis	2.4	-	-	-	-
Voldemort	0.9	0.9	-	-	-

Tabelle 6.2: Konfigurationen der verschiedenen YCS-Benchmarks

Der YCSB wurde in mehreren Arbeiten [25, 35, 41, 146] auf verschiedenen Datenbanksystemen durchgeführt. Die dabei erzielten Ergebnisse lassen sich jedoch angesichts der hohen Heterogenität im Versuchsaufbau (Tabelle 6.2) nur teilweise miteinander vergleichen. Zwar wurden in allen Benchmarks ähnliche Workloads verwendet, doch

die Anzahl der Datensätze, die eingesetzten Datenbanksysteme und vor allem die zugrunde liegenden Hardwarekomponenten variieren sehr stark zwischen den einzelnen Messungen und sorgen so für unterschiedliche Ergebnisse.

Rabl, Sadoghi, Jacobsen et al. [146] verglichen die Leistungsfähigkeit von Cassandra, HBase, MySQL, Redis und Project Voldemort. Neben dem Verhalten unter maximaler Anfragelast wurde der Einfluss der Cluster- und Hauptspeichergröße im Rahmen verschiedener Hardwarekonfigurationen untersucht. So halten die in Benchmark 1 getesteten Datenbanksysteme die Datenmenge vollständig im Hauptspeicher, während Benchmark 2 eine Persistierung auf sekundäre Speichermedien erzwingt. Benchmark 3 misst ebenso wie Benchmark 1 und 2 die Latenz verschiedener Datenbanksysteme bei vollem Durchsatz. Allerdings sind hierbei vor allem die Ergebnisse von Cassandra kritisch zu betrachten, da diese Studie von DataStax, einem Anbieter einer Cassandra Distribution, in Auftrag gegeben wurde [41]. Die Benchmarks 4 [25] und 5 [35] unterscheiden sich von den Benchmarks 1 bis 3, da hier nicht die Latenz bei maximalem Durchsatz, sondern die Entwicklung der Latenz bei steigendem Durchsatz gemessen wird. Hierbei lässt sich die Leistungsfähigkeit der verschiedenen Datenbanksysteme unter einer ähnlichen Last vergleichen.

Leseoperationen

Um die Leistungsfähigkeit eines Datenbanksystems bei Leseoperationen messen zu können, bietet sich der in YCSB standardmäßig enthaltene Workload B an. Dieser besteht zu 95% aus Lese- und zu 5% aus Schreiboperationen.

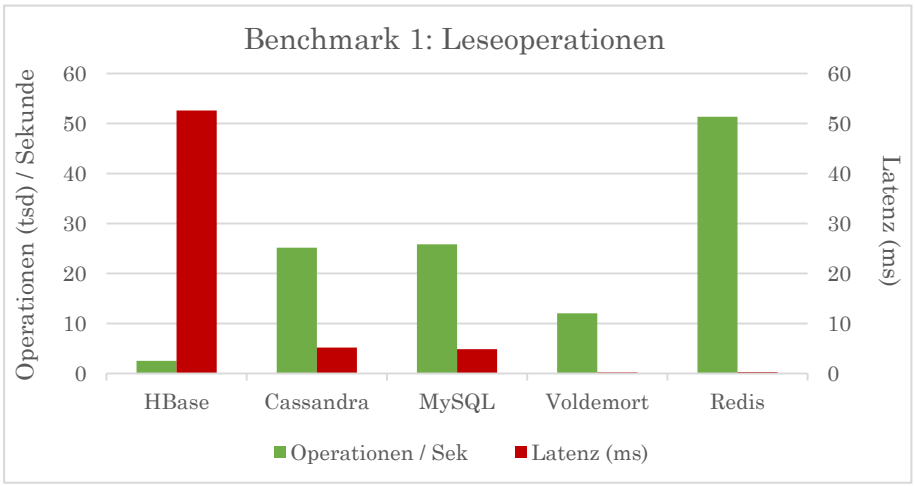


Abbildung 6.13: Durchsatz und Latenz bei Leseoperationen (Benchmark 1)

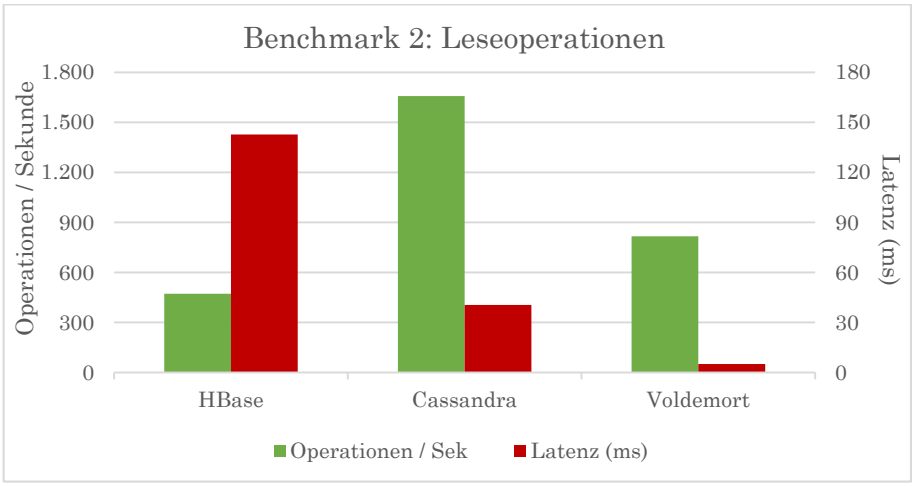


Abbildung 6.14: Durchsatz und Latenz bei Leseoperationen (Benchmark 2)

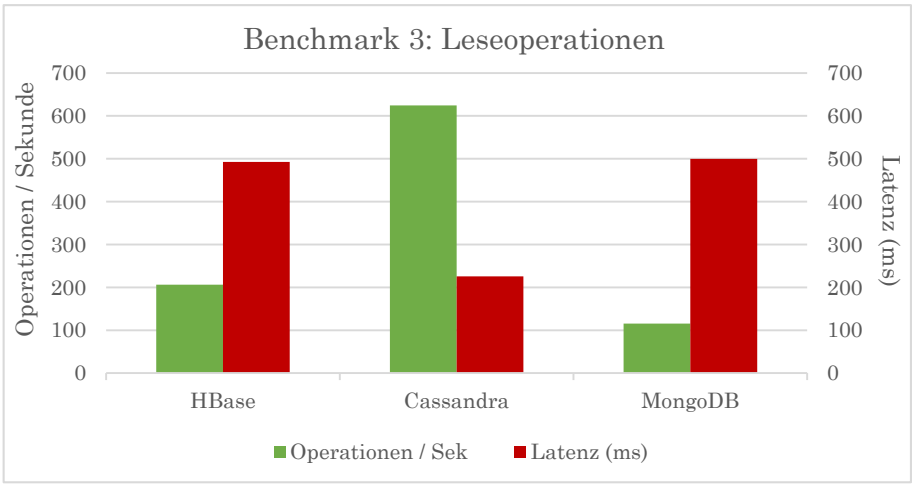


Abbildung 6.15: Durchsatz und Latenz bei Leseoperationen (Benchmark 3)

Die Abbildungen 6.13, 6.14 und 6.15 zeigen die Messergebnisse für Workload B bei den Benchmarks 1 bis 3. Hier werden sowohl der maximale Durchsatz als auch die durchschnittlich auftretende Latenz einer Leseanfrage für jedes Datenbanksystem einzeln gegenübergestellt. Es zeigt sich, dass der Key Value Store Redis den höchsten Durchsatz und gleichzeitig die niedrigste Latenz aufweist. Der Colum Family Store HBase und der Document Store MongoDB besitzen dagegen die schlechteste Leistungsfähigkeit bei Leseoperationen.

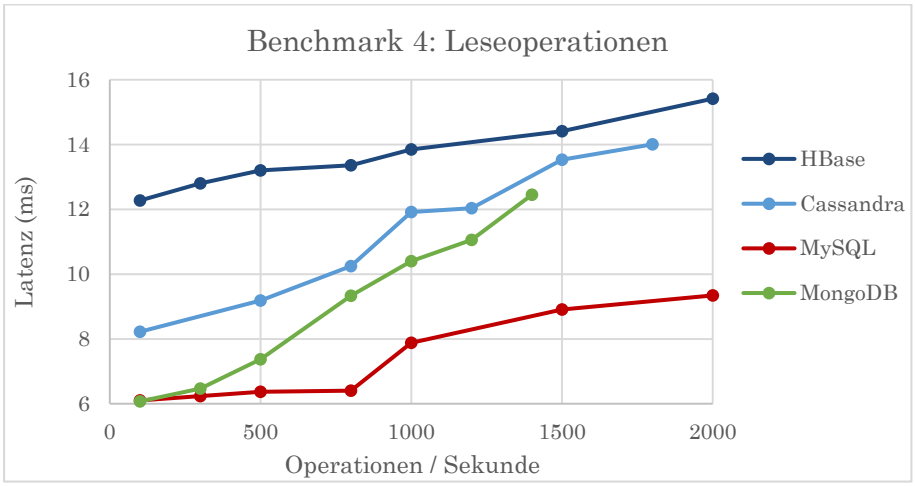


Abbildung 6.16: Latenzentwicklung bei steigendem Lese-Durchsatz (Benchmark 4)

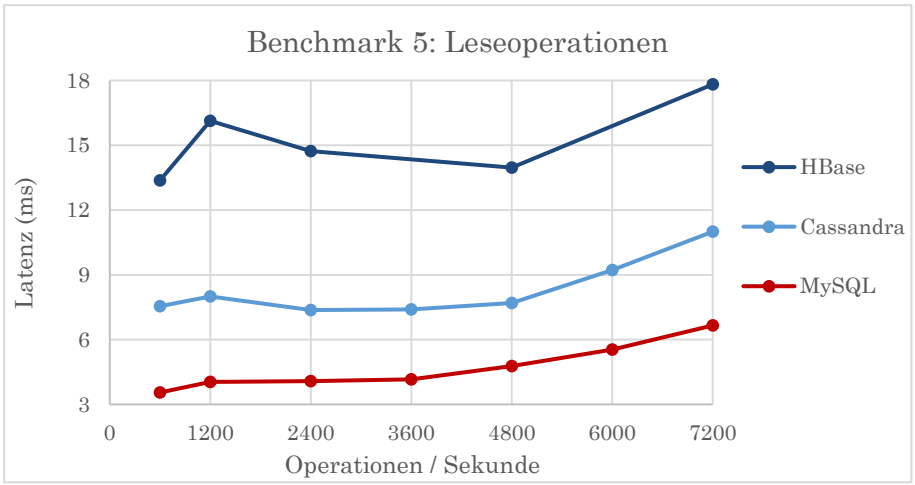


Abbildung 6.17: Latenzentwicklung bei steigendem Lese-Durchsatz (Benchmark 5)

Die Benchmarks 4 [25] (Abbildung 6.16) und 5 [35] (Abbildung 6.17) messen die Latenz der Datenbanksysteme bei unterschiedlichen Lasten. Die hier erzielten Messergebnisse gleichen den Resultaten der Benchmarks 1 bis 3. Erneut weist HBase die höchste Latenz bei Leseanfragen auf. Allerdings nimmt die Latenz bei MongoDB bei steigendem Durchsatz so stark zu, dass ab 1400 Operationen pro Sekunde keine weiteren Messungen durchgeführt werden konnten. MySQL stellt sich bei diesen beiden Messungen als leistungsfähigstes Datenbanksystem heraus.

Schreiboperationen

Für die Messung der Leistungsfähigkeit eines Datenbanksystems in Anwendungsfällen mit sehr hoher Schreiblast bietet sich der YCSB-Workload A an. Dieser besteht zu 50% aus Lese- und zu 50% aus Schreiboperationen.

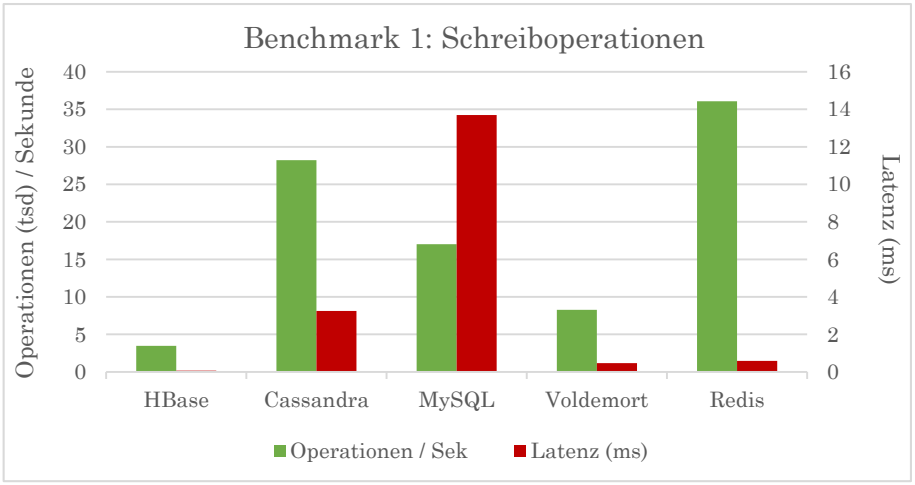


Abbildung 6.18: Durchsatz und Latenz bei Schreiboperationen (Benchmark 1)

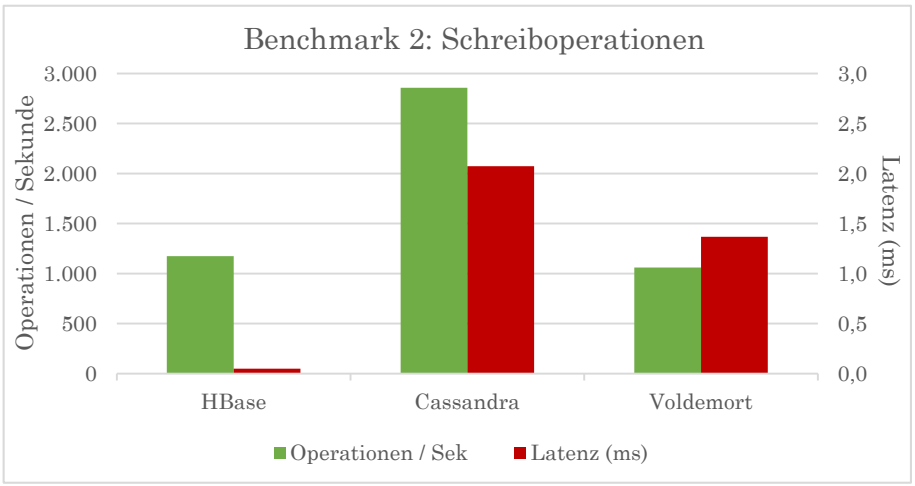


Abbildung 6.19: Durchsatz und Latenz bei Schreiboperationen (Benchmark 2)

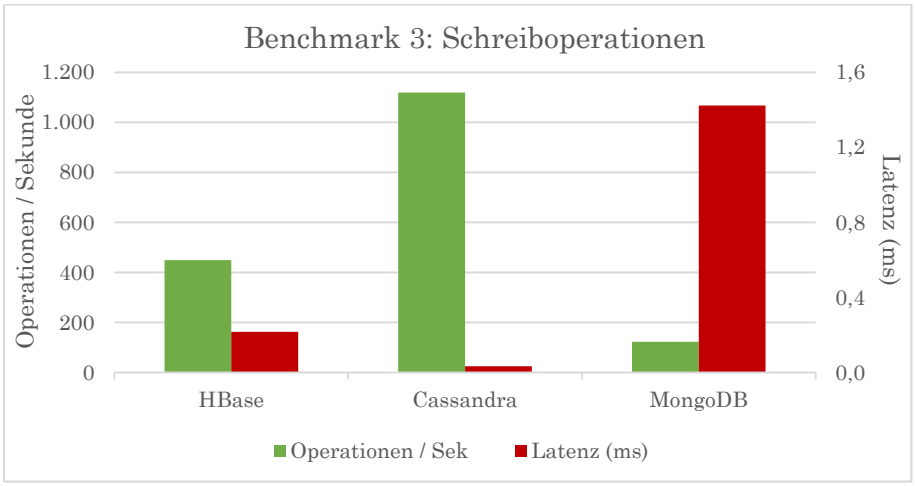


Abbildung 6.20: Durchsatz und Latenz bei Schreiboperationen (Benchmark 3)

Die in den Abbildungen 6.18, 6.19 und 6.20 dargestellten Messergebnisse für Workload A bei den Benchmarks 1 bis 3 unterscheiden sich stark von denen des Workloads B. Zwar kann der Key Value Store Redis weiterhin die meisten Operationen pro Sekunde verarbeiten, allerdings weist der Column Family Store HBase, der die mit Abstand höchste Latenz bei Leseanfragen besitzt, bei Schreibanfragen die niedrigste Latenz auf. MySQL zeigt bei Schreibanfragen die höchste Latenz. Die Latenzmesswerte in Benchmark 3 wurden während mehrerer verschiedener Lasten gemessen. Da speziell die Latenzwerte von Cassandra sehr stark von den Messwerten der Benchmarks 1 und 2 abweichen, sollten diese sehr kritisch betrachtet werden. Darüber hinaus fällt die schlechte Leistungsfähigkeit von MongoDB im Vergleich zu HBase auf. In allen drei Messungen konnte beobachtet werden, dass Column Family Stores eine höhere Leistungsfähigkeit bei Schreiboperationen als bei Leseoperationen besitzen.

Die in den Abbildungen 6.21 und 6.22 dargestellten Messergebnisse der Benchmarks 4 und 5 zeigen ein ähnliches Resultat wie die der Benchmarks 1 bis 3. HBase und Cassandra sind in der Lage, Schreibanfragen wesentlich schneller als MongoDB und MySQL zu bearbeiten. Bei steigendem Durchsatz wächst die Latenz von MySQL so schnell an, dass in Benchmark 4 ab 1500 Operationen pro Sekunde und in Benchmark 5 ab 7000 Operationen pro Sekunde keine weiteren Messerungen durchgeführt werden konnten.

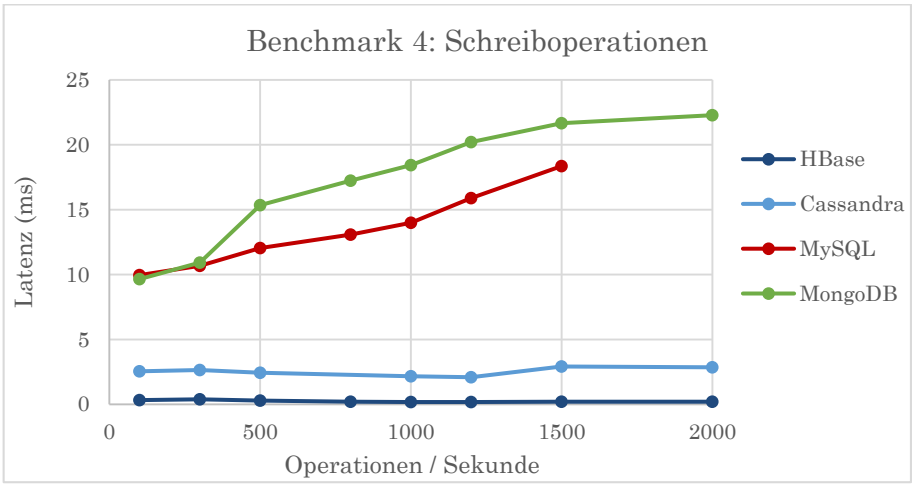


Abbildung 6.21: Latenzentwicklung bei steigendem Schreib-Durchsatz (Benchmark 4)

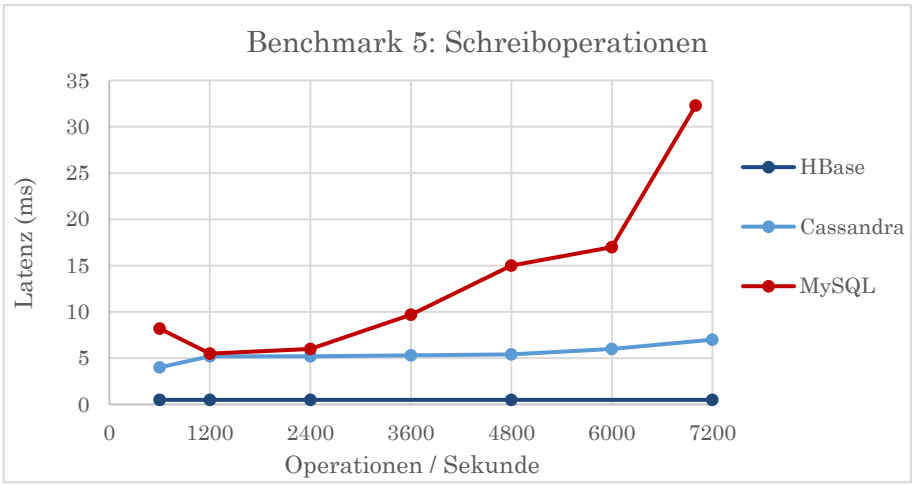


Abbildung 6.22: Latenzentwicklung bei steigendem Schreib-Durchsatz (Benchmark 5)

Auffällig bei der Betrachtung der hier zitierten Benchmarks sind die teilweise sehr großen Unterschiede in den Messergebnissen. Wie bereits zu Beginn dieses Unterkapitels beschrieben, liegen die Gründe hierfür in den sehr unterschiedlichen Versuchsaufbauten. Neben verschiedenen Hard- und Softwarekonfigurationen können, wie im Fall von DataStax, auch Interessenskonflikte die Messergebnisse beeinflussen. Trotz der festgestellten Unterschiede in den Messergebnissen konnten dennoch Tendenzen für Stärken und Schwächen der einzelnen Datenbanksysteme identifiziert werden. Diese werden im Folgenden anhand der technischen Eigenschaften der Datenbanksysteme näher erläutert.

6.2.2 Relationale Datenbanken

Die relationale Datenbank MySQL erwies sich bei Leseoperationen in den Benchmarks 1, 4 und 5 als sehr leistungsstark. Einzig die beiden Key Value Stores Redis und Project Voldemort konnten bessere Latenzwerte und im Fall von Redis auch einen höheren Durchsatz als MySQL aufweisen. Die Schwäche von MySQL liegt dagegen bei der Verarbeitung von Schreiboperationen. In den Benchmarks 1, 4 und 5 wurde die hohe Latenz bei Schreiboperationen nur vom Document Store MongoDB übertroffen. Zwar konnte bei hauptspeicherbasierter Datenverarbeitung ein hoher Durchsatz erzielt werden (Benchmark 1), doch bei festplattenbasierter Persistierung (Benchmarks 4 und 5) fiel dieser am niedrigsten aus.

Die Gründe dafür liegen in der Konzeption relationaler Datenbanken für eine zuverlässige und in vielen Anwendungsfällen einsetzbare Datenverarbeitung. Zu den wichtigsten Komponenten eines relationalen Datenbanksystems zählen deshalb die ACID-kompatible Transaktionsverwaltung sowie die mächtige Anfragesprache SQL. Diese beiden Komponenten können jedoch bei einfachen Lese- und Schreiboperationen, wie sie in den Workloads A und B auftreten, einen Overhead verursachen, der den Transaktionsdurchsatz erheblich verringert.

Der zusätzliche Aufwand, der durch die Prüfung, Optimierung und Übersetzung von SQL-Statements bis hin zu ausführbarem Code erzeugt wird, kann laut Stonebraker und Cattell [163] dank der Fortschritte im Bereich der Anfrageoptimierung in den meisten Fällen vernachlässigt werden. Dagegen hat der Overhead, der aufgrund nicht benötigter Transaktionsgarantien verursacht wird, einen wesentlich höheren Einfluss auf den Durchsatz und kann durchaus einen Flaschenhals in der performanten Verarbeitung von OLTP-Lasten darstellen [161]. Dieser Overhead ist dabei sowohl bei Techniken der Synchronisations- als auch der Recovery-Komponente von Transaktionsverwaltungen zu finden.

Anomalien im Mehrbenutzerbetrieb werden bei relationalen Datenbanken durch die Synchronisation nebenläufiger Transaktionen vermieden. Hierfür werden überwiegend pessimistische Synchronisationsverfahren [128, 136] eingesetzt. Neben dem Setzen und Aufheben von Sperren stellen die Verwaltung einer Lock-Tabelle sowie Techniken zur Erkennung und Vermeidung von Deadlocks und verhungernden Transaktionen einen weiteren Aufwand dar. Ein von Harizopoulos et al. [77] durchgeführter OLTP-Benchmark ermittelte hierbei einen Overhead von ca. 20 Prozent [163]. Hinzu kommen die bei dieser Messung nicht berücksichtigten Kosten zur Prüfung statischer und dynamischer Konsistenzbedingungen.

Einen weiteren Overhead verursachen Datenbank-Operationen, die zur Kontrolle der Integritätsbedingungen erforderlich sind. Diese können „zu intolerabel häufigen Relationen-Scans mit zahlreichen E/A-Vorgängen und hohem Sperrpotential führen“ [76, S.405] und dadurch Schreiboperationen deutlich verlangsamen. Vor allem Trigger können zu erheblichen Leistungsproblemen führen.

Um die Dauerhaftigkeit von Transaktionen zu gewährleisten, werden sämtliche von einer Transaktion durchgeführten Operationen in einer persistenten Log-Datei (WAL-Prinzip) gesichert, bevor die Transaktion als erfolgreich beendet bestätigt wird. Das von MySQL verwendete Speichersubsystem InnoDB sichert diese Datei einmal pro Sekunde im Sekundärspeicher [128, S.1210]. Im Falle eines Hauptspeicherfehlers wären demzufolge im schlimmsten Fall alle Transaktionen verloren, die innerhalb einer Sekunde getätigt wurden. InnoDB bietet weitere Konfigurationsmöglichkeiten, welche die Dauerhaftigkeitsgarantie zugunsten der Performanz lockern. Dennoch entsteht durch die Persistierung der Log-Datei ein Mehraufwand, der laut Stonebraker und Cattell [163] ca. 23 Prozent beträgt.

Ein weiterer Mehraufwand, der sich im Zusammenhang mit persistenter Speicherung von Transaktionen ergibt, entsteht aufgrund des Einsatzes von Festplatten. Der permanente Preisverfall im Hauptspeichersegment ermöglicht es in zunehmendem Maße, OLTP-Datenbanken vollständig im Hauptspeicher zu hinterlegen. Die Verwendung der Pufferverwaltung ist in einem solchen Fall nicht mehr notwendig und verursacht dann einen Overhead von ca. 33 Prozent [163].

Die hier vorgestellten Systemkomponenten können, wenn sie für eine Aufgabenstellung nicht benötigt werden, die OLTP-Performanz vor allem von Schreiboperationen unnötig einschränken. Harizopoulos et al. [77] konnten dank eines Verzichts auf diese Komponenten einen ca. 20-fachen Leistungsgewinn erzielen.

6.2.3 Key Value Stores

Key Value Stores sind auf die hochperformante Verarbeitung einfacher Lese- und Schreibanfragen konzipiert, wie sie in den Workloads A und B vorkommen. Es ist daher nicht überraschend, dass sowohl Redis als auch Project Voldemort die beste Leistungsfähigkeit bei hauptspeicherbasierten Lese- und Schreibanfragen aufweisen. Der Grund für diesen Performanzvorteil liegt in ihrem Minimalismus. Der Verzicht auf jeglichen Funktionsumfang, der nicht zwingend benötigt wird, macht Key Value Stores praktisch overheadfrei, weshalb diese Systeme extrem hohe Durchsatzraten bei sehr niedriger Latenz aufweisen können (siehe Benchmark 1).

Key Value Stores verwenden Hash-Zugriffspfade und greifen darum unabhängig von der Datenmenge mit einem Aufwand von *O(1)* lesend und schreibend auf einzelne Datensätze zu (siehe Kapitel 2.3.1). Allerdings können Key Value Stores genau aus diesen Gründen keine Bereichsanfragen unterstützen und nehmen außer dem Gleichheitsoperator keine weiteren Restriktionsbedingungen bei der Anfrageformulierung entgegen (siehe Kapitel 3.2.2).

Neben dem minimalen Funktionsumfang bei Lese- und Schreiboperationen verzichten Key Value Stores vollständig auf ein ACID-kompatibles Transaktionskonzept. Zusammengehörige Lese- und Schreiboperationen können nicht zu einer atomaren Transaktion zusammengefasst, sondern müssen jeweils einzeln und sequenziell ausgeführt

werden. Hierdurch kann auf kostspielige Verfahren zur Nebenläufigkeitskontrolle von Transaktionen verzichtet werden. Die Schemafreiheit des Datenmodells ermöglicht keine Verletzung von Constraints, Datentypen, Fremdschlüsseln und Triggern, weshalb Schreiboperationen keiner Integritätsprüfung unterzogen werden.

Ein weiteres typisches Merkmal von Key Value Stores ist deren vollständige Datenhaltung im Arbeitsspeicher. Sekundäre Speichermedien werden in der Regel ausschließlich zur Datensicherung verwendet und asynchron aktualisiert. Durch den Verzicht auf eine Sicherung von Redo- und Undo-Informationen in einer persistenten Log-Datei können Schreiboperationen wesentlich schneller verarbeitet werden. Allerdings droht bei diesem Ansatz im Fall eines Systemfehlers ein vollständiger Verlust aller nicht persistierten Daten. Ein weiterer Nachteil kann sich angesichts der begrenzten Größe des Hauptspeichers ergeben. Übersteigt die Datenmenge die Größe des Hauptspeichers (Benchmarks 2 bis 5), können Key Value Stores wie im Fall von Redis gar nicht oder wie im Fall von Project Voldemort nur mit erheblichen Leistungseinbußen vor allem bei Schreiboperationen (siehe Abbildung 6.19) eingesetzt werden.

In Anbetracht des minimalen Funktionsumfangs bei der Anfrageverarbeitung und des Verzichts auf ein ACID-kompatibles Transaktionskonzept reduziert sich das potenzielle Einsatzgebiet von Key Value Stores. Diese Systeme sind ausschließlich für einfache Lese- und Schreiboperationen auf einzelne, konsistenzunkritische Datensätze geeignet. Sobald komplexe Anfragen, eine persistente Speicherung oder Konsistenz im Mehrbenutzerbetrieb erforderlich sind, müssen ein erheblicher Implementierungsaufwand auf Applikationsebene sowie ein Verlust des Geschwindigkeitsvorteils gegenüber anderen Datenbanksystemen in Kauf genommen werden.

6.2.4 Document Stores

Document Stores wurden für die effiziente Verarbeitung von semistrukturierten Daten entworfen. Durch das Datenformat JSON wird ein ausdrucksstarkes und zugleich flexibles Datenmodell bereitgestellt, mit dessen Hilfe sich heterogene Datensätze in ihrer natürlichen Form ohne zusätzliche Hilfsstrukturen abbilden lassen. Dank einer umfangreichen und ausdrucksstarken Anfragebibliothek kann selbst auf verschachtelte Datensätze bequem und ohne weiteren Aufwand auf Applikationsebene zugegriffen werden.

Document Stores verzichten wie die meisten NoSQL-Datenbanken aus Performanzgründen auf eine vollständige ACID-Unterstützung bei der Anfrageverarbeitung. Weder CouchDB noch MongoDB unterstützen Transaktionen und bieten daher keine Möglichkeit, mehrere Operationen zu einer atomaren Einheit zusammenzufassen. CouchDB synchronisiert konkurrierende Anfragen mit optimistischen Mehrversionsverfahren (MVCC). Da Änderungen jedoch auf Applikationsebene erfolgen, muss das gesamte Dokument zwischen Client und Datenbank übertragen werden, was wiederum bei großen Dokumenten zu einem höheren Kommunikationsaufwand führen kann. Im Gegensatz dazu werden Änderungsoperationen bei MongoDB direkt auf den

Dokumenten in der Datenbank durchgeführt und somit der Kommunikationsaufwand reduziert. Allerdings werden Anfragen in MongoDB pessimistisch mithilfe von globalen Lese-Schreib-Sperren synchronisiert. Da diese Sperren einer Schreiboperation einen exklusiven Zugriff auf die gesamte Datenbank und nicht auf einzelne Dokumente sichern, besitzt MongoDB eine sehr schlechte Performanz bei der Verarbeitung von Schreibanfragen (Benchmarks 3 und 4). Ist eine Sperre gesetzt, können außerdem keine Leseanfragen verarbeitet werden. Auch wenn CouchDB die optionale Formulierung und Überprüfung von Integritätsbedingungen durch den Einsatz von Validationsfunktionen ermöglicht, werden Änderungsoperationen in Document Stores standardmäßig nicht auf Korrektheit überprüft, sondern direkt persistent auf Festplatten gespeichert.

Document Stores können ihre Vorteile bei beliebigen Anfragen auf heterogene Datensätze ausspielen, bei denen beispielsweise unter Zuhilfenahme der Aggregation auf teure Verbundoperationen verzichtet werden kann. Bei den Workloads A und B des YCS-Benchmarks werden jedoch einfache Lese- und Schreiboperationen auf den Primärschlüsseln homogener Datensätze ausgeführt. Dieses Anforderungsprofil widerspricht den Daten und Anfragen, für die Document Stores konzipiert wurden. Berücksichtigt man zudem das sehr pessimistische Synchronisationsverfahren von MongoDB, welches im Gegensatz zu anderen Datenbanksystemen eine hohe Konsistenz garantiert, ist das schlechte Abschneiden von MongoDB bei den verschiedenen YCS-Benchmarks nicht überraschend.

6.2.5 Column Family Stores

Column Family Stores besitzen die für Datenbanksysteme ungewöhnliche Eigenschaft, dass sie Schreibanfragen schneller und in einem höheren Durchsatz verarbeiten können als Leseanfragen. Die Gründe hierfür liegen im SSTable-Konzept dieser Systeme. Da Datensätze bei Schreiboperationen nicht direkt geändert werden, sondern stattdessen neue Versionen der Datensätze sequenziell persistiert werden, ziehen Schreibanfragen bei Column Family Stores keine teuren Suchanfragen und Sperrmechanismen nach sich. Dieses Mehrversionskonzept wirkt sich allerdings bei Leseanfragen negativ auf die Performanz aus, da sämtliche Versionen eines Datensatzes gelesen und zusammengeführt werden müssen, um die zu einem Zeitpunkt gültige Version zurückgeben zu können.

HBase bietet in der Standardkonfiguration deutlich höhere Konsistenz- und Dauerhaftigkeitsgarantien als Cassandra. Alle Schreiboperationen werden bei HBase direkt oder optional mit einer Verzögerung von einer Sekunde in einer persistenten Log-Datei protokolliert [11, Kap.14.8] und alle Replikate im verteilten Dateisystem synchron aktualisiert [69, S.337]. Cassandra persistiert diese Log-Datei hingegen nur alle zehn Sekunden [42, S.83] und bestätigt eine Schreibanfrage gegenüber einem Client, sobald die Änderungen bereits auf einem Replikat erfolgreich durchgeführt wurden [42, S.78]. Cassandra besitzt darum eine deutlich höhere Leistungsfähigkeit bei Schreiboperationen als HBase. In den oben aufgeführten Benchmarks sind allerdings Ergebnisse zu

beobachten, die von diesen theoretischen Grundsatzüberlegungen deutlich abweichen. So weist Cassandra zwar erwartungsgemäß einen höheren Durchsatz bei Schreibanfragen auf, dafür verursacht HBase jedoch eine deutlich niedrigere Latenz als Cassandra. Der Grund hierfür liegt darin, dass in dem vom YCS-Benchmark bereitgestellten HBase-Client ein standardmäßig deaktivierter Puffer aktiviert wurde [25, S.11]. Dadurch werden Schreibanfragen zunächst im Hauptspeicher des Clients zwischengespeichert und dann asynchron an das Datenbanksystem übertragen [11, 69]. Die niedrigen Latenzwerte bei einer clientseitigen Zwischenspeicherung sind insofern nachvollziehbar.

Die im Rahmen des YCSBs gemessene Leistungsfähigkeit von Leseanfragen basiert auf einfachen Suchanfragen auf den Primärschlüsseln der verschiedenen Datensätze. Für diese Art von Anfragen ist Cassandra HBase überlegen und kann sowohl einen höheren Durchsatz als auch eine niedrigere Latenz aufweisen. Soll ein Datensatz aus Cassandra gelesen werden, wird der zuständige Tablet-Server mittels einer Hashfunktion und des Primärschlüssels des Datensatzes berechnet. Idealerweise befinden sich der Blockindex und der Datensatz bereits im Cache des Tablet-Servers, sodass für die Suche eines Datensatzes kein einziger Plattenzugriff notwendig ist (siehe Kapitel 5.3.1). Die Tablet-Server-Adresse wird bei HBase hingegen mit einer B*-Baum-Suche ermittelt. Einen Blockindex- und Datensatz-Cache enthalten Tablet-Server in HBase nicht.

Die Hash-Verteilung erweist sich jedoch als gravierender Nachteil, wenn Datensätze anhand eines Wertebereichs ausgelesen werden sollen. Während Cassandra in der Standardkonfiguration nicht in der Lage ist, solche Anfragen zu unterstützen [42, S.19], stellen sich diese in HBase als äußerst effiziente Operationen heraus. Hier werden Datensätze nicht über den Hashwert des Primärschlüssels lokalisiert, sondern sortiert nach ihrem Primärschlüssel abgelegt. Alphabetisch zusammenhängende Datensätze werden infolgedessen am selben Speicherplatz verwaltet, sodass Bereichsanfragen mit einer einzigen Leseoperation beantwortet werden können. Zwar bietet auch Cassandra optional eine Verteilung mithilfe von Wertebereichen und eine sortierte Ablage der Datensätze, doch wird in der Dokumentation von Cassandra von der Verwendung dieser Konfigurationsoption abgeraten [42, S.20].

6.2.6 Zusammenfassung

In diesem Unterkapitel wurde die Leistungsfähigkeit verschiedener Datenbanksysteme unter Zuhilfenahme von Benchmarks gemessen und basierend auf den in den vorherigen Kapiteln herausgearbeiteten Konzepten analysiert und interpretiert. Um die verschiedenen Systeme hinsichtlich ihrer Leistungsfähigkeit vergleichen zu können, gilt es, neben einer Unterscheidung zwischen Lese- und Schreibanfragen weitere Kernkriterien zu berücksichtigen:

Durchsatz: Der Durchsatz beschreibt die Anzahl der Anfragen, die innerhalb eines bestimmten Zeitintervalls von einem Datenbanksystem verarbeitet werden können.

Latenz: Als Latenz wird die Zeit bezeichnet, die ein Datenbanksystem zur Verarbeitung einer einzelnen Anfrage benötigt. Latenz und Durchsatz sind die beiden Kenngrößen zur Bestimmung der Leistungsfähigkeit eines Datenbanksystems. Übersteigt der Durchsatz die Kapazitäten eines Datenbanksystems, steigt automatisch auch die Latenz einer jeden Anfrage.

Bereichsanfragen und Attributrestriktionen: Die Leseanfragen in den Benchmarks basieren auf einfachen Aufrufen bestimmter Datensätze anhand ihrer Primärschlüssel. Einige Datenbanksysteme sind speziell auf diese primitiven Anfragen ausgelegt und weisen deswegen eine sehr hohe Leistungsfähigkeit bei den hier zitierten Messungen auf. Aufgrund fehlender Unterstützung für Bereichsanfragen oder Attributrestriktionen sind diese Leistungsfähigkeitsvorteile allerdings in Anwendungsfällen, in denen mehrere Datensätze anhand eines Wertebereiches ausgelesen oder bei denen Datensätze aufgrund bestimmter Nicht-Schlüsselattributwerte selektiert werden, nicht zu verzeichnen.

Sperren: Einige Datenbanksysteme steigern die Leitungsfähigkeit in der Anfrageverarbeitung indem anstelle von Sperren optimistische Synchronisationsmechanismen eingesetzt werden. Die temporäre Verletzung von Konsistenzgarantien kann jedoch nicht in allen Anwendungsbereichen toleriert werden, weshalb die gemessenen Leistungsfähigkeitsvorteile auch in dieser Hinsicht kritisch betrachtet werden müssen.

Dauerhaftigkeit: Latenz und Durchsatz von Schreibanfragen lassen sich sehr leicht steigern, indem die Dauerhaftigkeitsgarantien der zu speichernden Daten reduziert werden. Hauptspeicherbasierte Datenbanksysteme besitzen bei Schreiboperationen eine wesentlich höhere Leistungsfähigkeit als Systeme, bei denen die geschriebenen Daten zunächst auf Festplatten persistiert werden. Die Dauerhaftigkeit muss folglich bei einem Vergleich der Leistungsfähigkeit zwingend berücksichtigt werden.

Datenbankklasse	RDBMS	KVS		DOC	CFS	
Datenbanksystem	MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Durchsatz: Lesen	++	+++	+	−−−	−−	++
Latenz: Lesen	+	+++	+++	−−	−−−	−
Durchsatz: Schreiben	+	+++	−	−−−	−−	++
Latenz: Schreiben	−−	++	++	−−−	+++	+
Bereichsanfragen	+	−	−	+	+	−
Attributrestriktionen	+	−	−	+	−	−
Sperren	+	−	−	+	−	−
Dauerhaftigkeit	+	−−−	−−	+	+	−

Tabelle 6.3: Gegenüberstellung der Leistungsfähigkeit der Datenbanksysteme

Die unterschiedlichen leistungsbestimmenden Eigenschaften von relationalen Datenbanken (RDBMS), Key Value Stores (KVS), Document Stores (DOC) und Column Family Stores (CFS) sind in Tabelle 6.3 hinsichtlich der hier beschrieben Kriterien gegenübergestellt.

Relationale Datenbanken sind auf eine zuverlässige und in vielen Anwendungsfällen einsetzbare Datenverarbeitung ausgerichtet. Die hierfür notwendigen ACID-Eigenschaften der eingesetzten Transaktionssysteme verursachen jedoch bei Schreiboperationen einen Overhead, der sich negativ auf die Leistungsfähigkeit dieser Operationen auswirkt. Relationale Datenbanken weisen daher im Vergleich zu Key Value Stores und Column Family Stores eine schlechtere Performanz bei Schreibanfragen auf. Leseanfragen können dagegen sehr effizient ausgeführt werden und unterliegen keinen Einschränkungen in Bezug auf Bereichsanfragen.

Key Value Stores sind infolge ihres Minimalismus praktisch overheadfrei und besitzen deswegen bei einfachen Lese- und Schreibanfragen auf einzelne Datensätze die höchste Leistungsfähigkeit. Durch eine eingeschränkte Anfrageunterstützung, eine in der Regel hauptspeicherbasierte Datenhaltung und einen Verzicht auf sämtliche ACID-Garantien können diese Systeme sehr hohe Durchsatzraten bei gleichzeitig niedriger Latenz erzielen. Sobald allerdings zusätzliche Funktionalitäten, wie beispielsweise eine dauerhafte Sicherung der geschriebenen Datensätze, benötigt werden, verlieren Key Value Stores ihre Vorteile gegenüber anderen Datenbanksystemen.

Document Stores wurden für die effiziente Verarbeitung von semistrukturierten, aggregierten Datensätzen entworfen. Für einfache Lese- und Schreiboperationen auf homogenen Daten, wie sie im YCSB durchgeführt werden, sind sie nicht geeignet. MongoDB weist deshalb sowohl bei Lese- als auch bei Schreibanfragen die schlechteste Leistungsfähigkeit auf. Allerdings werden von Document Stores Bereichsanfragen unterstützt und eine dauerhafte Sicherung der Datensätze garantiert.

Column Family Stores sind für hochperformante Schreiboperationen auf sehr großen Datenmengen ausgelegt, zeigen dafür jedoch eine schwächere Leistungsfähigkeit bei der Verarbeitung von Leseoperationen. Aufgrund umfangreicher Konfigurationsmöglichkeiten können Column Family Stores in diesem Spektrum allerdings sehr stark variieren. In seiner Standardkonfiguration ermöglicht HBase beispielsweise eine konsistente und dauerhafte Datenverarbeitung und verzeichnet demzufolge vergleichsweise niedrige Durchsatzraten. Cassandra hingegen erzielt aufgrund minimaler Konsistenzgarantien und einer besseren Cache-Ebene eine deutlich höhere Leistungsfähigkeit sowohl bei Lese- als auch bei Schreibanfragen. Allerdings werden diese Vorteile auf Kosten der Dauerhaftigkeit und einer schlechten Performanz bei Bereichsanfragen erzielt.

Hohe Verarbeitungsgeschwindigkeiten sind eine der Hauptanforderungen von Anwendungen im Big Data-Umfeld. Relationale Datenbanken sind für eine zuverlässige Datenverarbeitung in einem sehr breiten Spektrum an Anwendungsfällen mit unterschiedlichen Anforderungen konzipiert. Sie besitzen darum sowohl bei Lese- als auch

bei Schreibanfragen eine zufriedenstellende Leistungsfähigkeit. Im Gegensatz dazu sind NoSQL-Datenbanken auf die spezifischen Anforderungen bestimmter Anwendungsfälle ausgelegt. Durch einen Verzicht auf ACID-kompatible Transaktionskonzepte und den Einsatz anderer Speicherstrukturen können Key Value Stores und Column Family Stores eine höhere Leistungsfähigkeit bei bestimmten Anfragen aufweisen. Document Stores sind hingegen bei der einfachen Verarbeitung von Lese- und Schreiboperationen auf homogenen Daten relationalen Datenbanken unterlegen.

6.3 Wachsende Datenmenge

Die mit Big Data am häufigsten assoziierte Eigenschaft ist die extrem große Datenmenge, die beispielsweise durch mobile Endgeräte, industrielle Fertigung, Vernetzung von Alltagsgegenständen und dem Web 2.0 generiert wird. Neben der Größe stellt vor allem das sehr schnelle Wachstum dieser Datenmenge sehr hohe Anforderungen an die Skalierbarkeit der zugrunde liegenden Datenhaltung. Wie bereits in Kapitel 2 erwähnt, lassen sich Datenbanksysteme entweder vertikal oder horizontal skalieren. Da die vertikale Skalierung hinsichtlich ihrer physikalischen Grenzen für Big Data Anwendungsbeispiele ungeeignet ist, wird im Folgenden ausschließlich die horizontale Skalierung betrachtet.

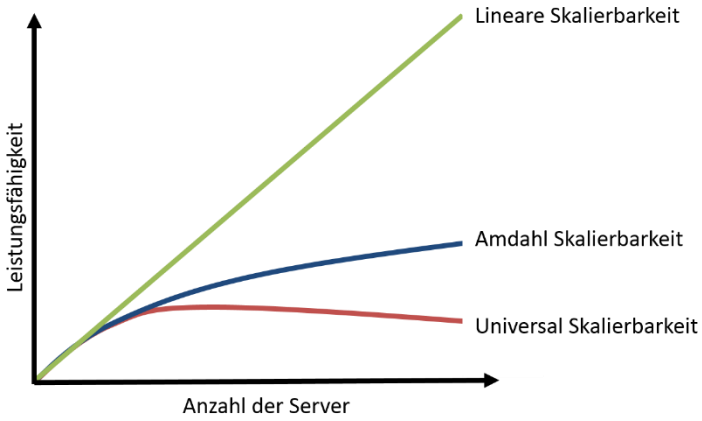


Abbildung 6.23: Darstellung der unterschiedlichen Skalierbarkeitsarten [155, S.526]

Bei der horizontalen Skalierung wird die Leistungsfähigkeit eines Datenbanksystems durch die Hinzunahme weiterer Verarbeitungsrechner inkrementell erweitert. Die Leistungsfähigkeit eines Datenbanksystems steigt dabei idealerweise linear mit der Rechneranzahl [147, S.8]. Viele verteilte Datenbanksysteme weisen jedoch keine lineare, sondern ab einem gewissen Punkt eine leicht negativ abweichende Skalierbarkeit

auf (Abbildung 6.23). Diese leichte Abweichung, welche sich mit zunehmender Serveranzahl verstärkt, liegt nach dem Gesetz von Amdahl [8] daran, dass sich ein Teil der zu verarbeitenden Last nicht parallelisieren lässt und entsprechend sequenziell ausgeführt werden muss. Darüber hinaus sorgen quadratisch mit der Knotenanzahl steigende Kommunikationskosten laut Gunther [75] dafür, dass einige Systeme ab einem gewissen Punkt mit zusätzlichen Verarbeitungsrechnern keine Steigerung, sondern sogar eine Verminderung der Leistungsfähigkeit verzeichnen. Auch wenn es sich bei diesen Gesetzen um theoretische Modelle handelt, ergeben sich dennoch für eine lineare horizontale Skalierbarkeit verteilter Systeme die nachfolgenden Richtlinien:

- Vermeidung von seriellen Prozessen,
- Vermeidung von Kommunikation zwischen den einzelnen Knoten.

Diese Richtlinien haben vor allem auf die Verteilung von relationalen Datenbanken eine sehr große Auswirkung. Um diese Systeme linear skalieren lassen zu können, ist ein Verzicht oder eine Einschränkung mehrerer charakteristischer Eigenschaften von relationalen Datenbanken notwendig. Zu diesen zählen Verbundoperationen, Transaktionen, Sperrprotokolle, Integritätsprüfungen und weitere Techniken, die mehrere Verarbeitungsrechner umfassen können. Da in vielen Anwendungsfällen auf diese Eigenschaften nicht verzichtet werden kann, beziehungsweise der Änderungsaufwand auf Applikationsebene zu hoch ist, werden seitens der NoSQL-Bewegung sehr häufig die schlechten Skalierbarkeitseigenschaften relationaler Datenbanken als Argument gegen einen Einsatz dieser Systeme in Big Data-Anwendungsfällen angeführt. NoSQL-Datenbanken sind im Gegensatz zu relationalen Datenbanken von Beginn an auf eine verteilte Datenhaltung ausgelegt. Durch den Verzicht auf Verbundoperationen, Transaktionen, pessimistische Synchronisationsmechanismen und Integritätsbedingungen sowie dank integrierter Replikations- und Verteilungstechniken sollen NoSQL-Datenbanken deutlich bessere Skalierbarkeitseigenschaften als relationale Datenbanken aufweisen. Die meisten Systeme versprechen eine linear mit der Rechneranzahl steigende Leistungsfähigkeit.

In diesem Unterkapitel werden die Skalierbarkeitseigenschaften der unterschiedlichen relationalen und nichtrelationalen Datenbanksysteme untersucht. Neben einer Analyse des Leistungszuwachses in Abhängigkeit der Knotenanzahl soll auch der für die Verteilung benötigte Aufwand ermittelt werden. Hierzu werden die Messergebnisse verschiedener Benchmarks herangezogen und mithilfe der in dieser Arbeit herausgestellten Verteilungskonzepte analysiert. Die Ergebnisse werden abschließend gegenübergestellt und bewertet.

6.3.1 Benchmarks

Die Skalierbarkeitseigenschaften von verteilten Datenbanksystemen können anhand der Metriken Speedup und Scaleup bewertet werden. Der Speedup bestimmt, wie stark die Leistungsfähigkeit eines Datenbanksystems bei einer konstanten Datenmenge

durch eine Erhöhung der Serveranzahl beeinflusst wird. Üblicherweise wird der Speedup in Bezug auf die Verkürzung der Antwortzeit verwendet [147, S.311]. Der Scaleup bestimmt das Datenbankverhalten, wenn die Datenmenge linear mit der Serveranzahl erhöht wird. Man unterscheidet zwischen Antwortzeit-Scaleup und Durchsatz-Scaleup [147, S.312].

Antwortzeit-Scaleup: „Der Antwortzeit-Scaleup bestimmt die Antwortzeitveränderung bei Einsatz von *n* Rechnern und *n*-facher Datenbankgröße verglichen mit dem 1-Rechner-Fall“ [147, S.312]. Im Optimalfall verändert sich die Antwortzeit dabei nicht (Antwortzeit-Scaleup = 1).

$$\text{Antwortzeit-Scaleup (n)} = \frac{\text{Antwortzeit bei einem Server}}{\text{Antwortzeit bei n Servern}}$$

Durchsatz-Scaleup: Der Durchsatz-Scaleup bestimmt das Verhältnis zwischen der Transaktionsrate auf *n* Rechnern (auf einer *n*-fach großen Datenbank) gegenüber der Transaktionsrate auf einem Rechner“ [147, S.313]. Idealerweise kann hierbei ein lineares Durchsatzwachstum beobachtet werden (Durchsatz-Scaleup = *n*).

$$\text{Durchsatz-Scaleup (n)} = \frac{\text{Durchsatz bei einem Server}}{\text{Durchsatz bei n Servern}}$$

Neben der Leistungsfähigkeit von Ein-Rechner-Datenbanksystemen untersuchten Rabl, Sadoghi, Jacobsen et al. [146] im Rahmen des im vorherigen Kapitel als Benchmark 1 bezeichneten Experiments auch die Skalierbarkeitseigenschaften der verschiedenen Datenbanksysteme. Hierzu wurden Redis, Project Voldemort, HBase, Cassandra und MySQL auf verschiedene Clustergrößen verteilt und die durchschnittlich auftretende Latenz der einzelnen Lese- und Schreibanfragen erneut bei maximalem Durchsatz gemessen. Bei Project Voldemort, Cassandra und HBase handelt es sich um verteilte Datenbanksysteme, die sich ohne applikationsseitigen Zusatzaufwand auf mehrere Rechner verteilen lassen. Redis und MySQL sind dagegen nicht für eine verteilte Datenhaltung konzipiert. Aus diesem Grund müssen sämtliche für die Verteilung notwendigen Funktionalitäten, wie Fragmentierung, Allokation, Lastbalancierung, verteilte Anfrageverarbeitung und Fehlererkennung, applikationsseitig implementiert werden. Neben einer Steigerung des Aufwandes und der Fehleranfälligkeit verursacht die Verlagerung dieser Datenbank-Funktionalitäten in die Applikationsebene eine deutlich höhere Last bei den Clients. Sowohl Redis als auch MySQL benötigten daher bei den Messungen die doppelte Anzahl an Clients [146, S.6]. Außerdem besitzt die applikationsseitige gegenüber der serverseitigen Verteilung den großen Vorteil, dass die Knoten vollkommen isoliert voneinander sind und nicht miteinander kommunizieren können. Hierdurch lassen sich deutlich bessere Scaleup-Werte erzielen.

Leseoperationen

Workload B besteht zu 95 Prozent aus Lese- und zu 5 Prozent aus Schreiboperationen und simuliert die Anfragelast von Applikationen mit einem sehr hohen Leseanteil. In Bezug auf die Leistungsfähigkeit bei der Verarbeitung von Leseanfragen kristallisieren sich deutliche Unterschiede zwischen den serverseitig und den applikationsseitig verteilten Datenbanksystemen bei steigender Serveranzahl heraus.

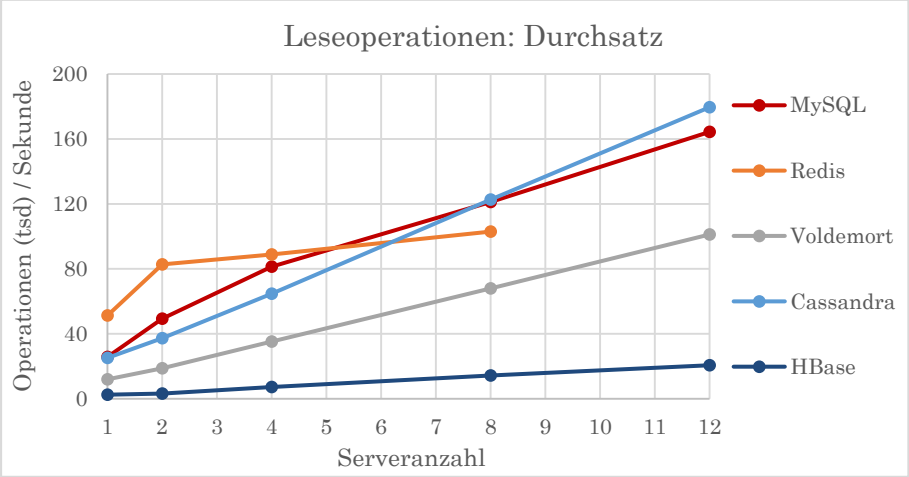


Abbildung 6.24: Durchsatz-Scaleup bei Leseoperationen

Die aus dem vorherigen Kapitel bekannten Messergebnisse für die verschiedenen Datenbanksysteme für Benchmark 1 sind in Abbildung 6.24 erneut zu beobachten. Wird die Serveranzahl von eins auf zwei verdoppelt, ist vor allem bei den beiden applikationsseitig verteilten Datenbanksystemen MySQL und Redis der größte Durchsatzanstieg zu verzeichnen. Wird die Serveranzahl weiter erhöht, weichen jedoch beide Systeme sehr stark vom linearen Leistungszuwachs ab. Der Grund hierfür liegt an der zu hohen Belastung und der schlechten Lastbalancierung auf Clientseite. Die Belastung wurde letztendlich bei Redis so hoch, dass die Messung bei einer Clustergröße von zwölf nicht mehr durchgeführt werden konnte [146, S.6]. Die serverseitig verteilten Datenbanksysteme Project Voldemort, HBase und Cassandra zeigen dagegen unabhängig von der Clustergröße eine lineare Durchsatzsteigerung.

Server #	MySQL	Redis	Voldemort	HBase	Cassandra
1	1,0	1,0	1,0	1,0	1,0
2	1,9	1,6	1,6	1,3	1,5
4	3,1	1,7	2,9	2,9	2,6
8	4,7	2,0	5,6	5,6	4,9
12	6,4		8,4	8,1	7,1

Tabelle 6.4: Durchsatz-Scaleup bei Leseoperationen

Der gemessene Durchsatz-Scaleup ist in Tabelle 6.4 abgebildet. Hier ist zu beobachten, dass die clientseitig verteilten Systeme MySQL und Redis bei niedriger Serveranzahl den höchsten Scaleup aufweisen, aber mit weiter steigender Serveranzahl Project Voldemort und HBase die besten Werte besitzen.

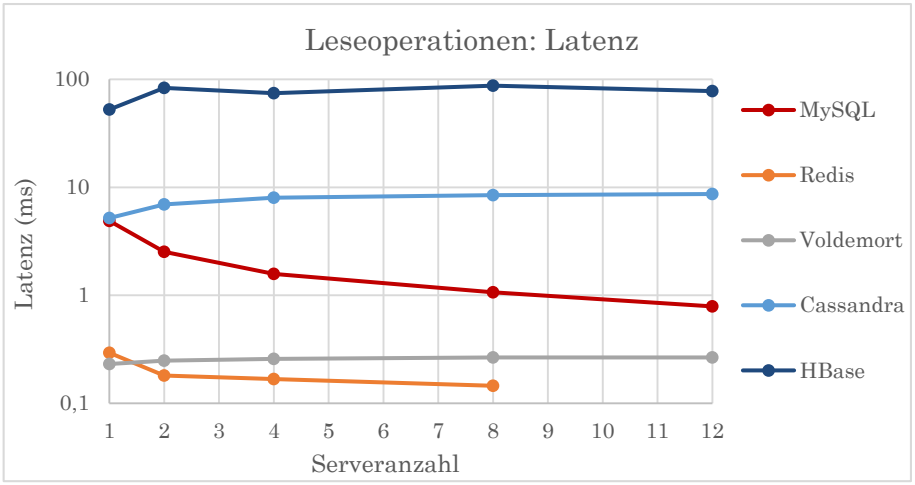


Abbildung 6.25: Antwortzeit-Scaleup bei Leseoperationen

Der bei MySQL und Redis mit steigender Serveranzahl verbundene Mehraufwand auf Clientseite ist auch bei der Messung des Antwortzeit-Scaleups zu beobachten. Da die Clients nicht in der Lage waren, den Durchsatz proportional zur Serveranzahl zu steigern, sank bei diesen Systemen bei jedem hinzugefügten Server die Antwortzeit, sodass am Ende kein Scaleup, sondern ein Speedup gemessen wurde. Project Voldemort und HBase zeigten hingegen eine nahezu konstante Latenz. Bei Cassandra ist eine leichte Zunahme der Antwortzeit zu beobachten (Abbildung 6.25 und Tabelle 6.5).

Server #	MySQL	Redis	Voldemort	HBase	Cassandra
1	1,0	1,0	1,0	1,0	1,0
2	0,5	0,6	1,1	1,6	1,3
4	0,3	0,6	1,1	1,4	1,5
8	0,2	0,5	1,2	1,7	1,6
12	0,2		1,1	1,5	1,7

Tabelle 6.5: Antwortzeit-Scaleup bei Leseoperationen

Berücksichtigt man sowohl die Ergebnisse des Durchsatz-Scaleups, als auch die des Antwortzeit-Scaleups, besitzen Project Voldemort und HBase die beste Skalierbarkeit bei Leseanfragen. Cassandra weicht dagegen bei steigender Serveranzahl etwas stärker als diese beiden Systeme von den Idealwerten ab. Die horizontale Skalierung von

MySQL und Redis muss als gescheitert gewertet werden, da weder ein linearer Durchsatz-Scaleup noch ein konstanter Antwortzeit-Scaleup gemessen werden konnte.

Schreiboperationen

Ein ähnliches Skalierbarkeitsverhalten ist bei der Durchführung des Workloads A zu beobachten, dessen Last zu 50% aus Lese- und zu 50% aus Schreiboperationen besteht. Die applikationsseitig verteilten Datenbanksysteme Redis und MySQL können keinen linearen Durchsatz-Scaleup aufweisen, wohingegen der Durchsatz bei den serverseitig verteilten Systemen Project Voldemort, Cassandra und HBase linear mit der Serveranzahl steigt. Abgesehen vom Messwert von MySQL bei einer Serveranzahl von zwei ähneln die in Abbildung 6.26 dargestellte Kurven denen von Abbildung 6.24.

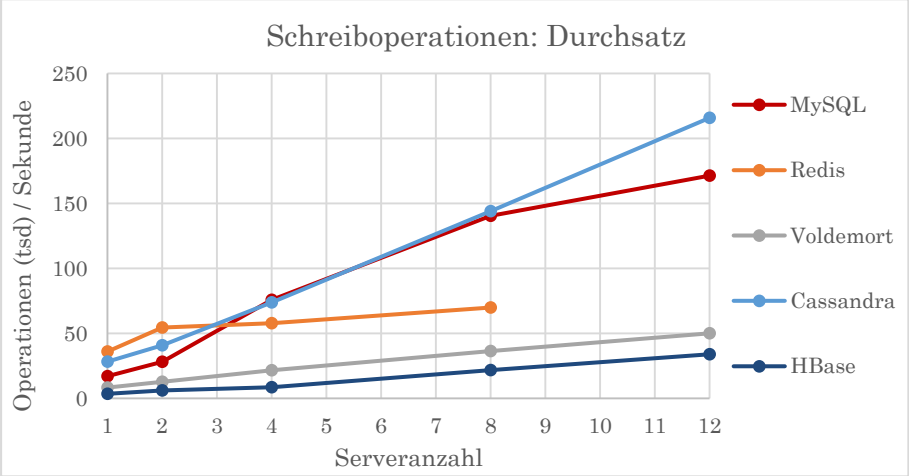


Abbildung 6.26: Durchsatz-Scaleup bei Schreiboperationen

Betrachtet man die Durchsatz-Scaleup-Werte in Tabelle 6.6 fällt auf, dass MySQL trotz des nicht linear verlaufenden Scaleups bei einer Clustergröße von vier und acht Knoten eine n-fache Durchsatzsteigerung aufweisen kann. HBase erweist sich von den serverseitig verteilten Datenbanksystemen bei der Skalierbarkeit als stärkstes System. Project Voldemort besitzt im Vergleich zu Leseanfragen einen schlechteren Durchsatz-Scaleup.

Server #	MySQL	Redis	Voldemort	HBase	Cassandra
1	1,0	1,0	1,0	1,0	1,0
2	1,7	1,5	1,5	1,7	1,4
4	4,5	1,6	2,6	2,5	2,6
8	8,3	1,9	4,4	6,2	5,1
12	10,1		6,1	9,8	7,6

Tabelle 6.6: Durchsatz-Scaleup bei Schreiboperationen

Betrachtet man die von Rabl, Sadoghi, Jacobsen et al. [146] gemessenen Latenzwerte, ist der hohe Durchsatz-Scaleup von MySQL umso erstaunlicher (Abbildung 6.27).

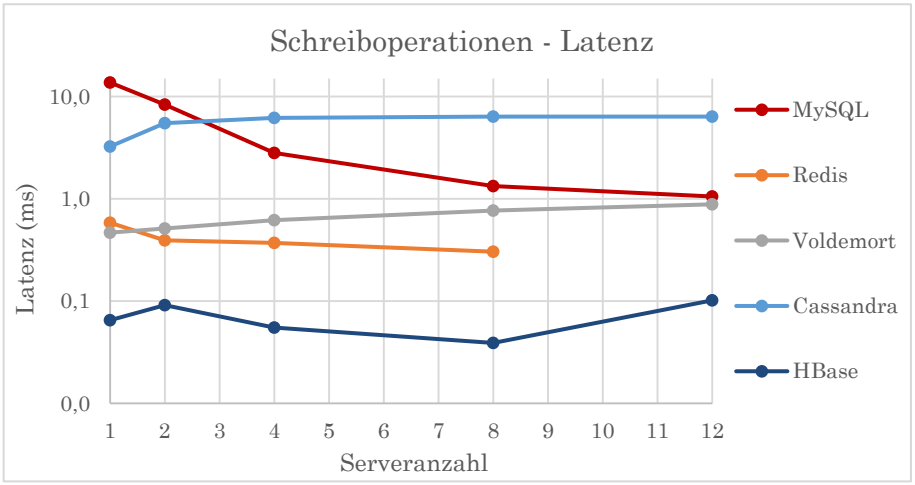


Abbildung 6.27: Antwortzeit-Scaleup bei Schreiboperationen

Wie auch bei Leseoperationen ist bei den applikationsseitig verteilten Datenbanksystemen MySQL und Redis ein Speedup bei der Messung der Antwortzeiten zu beobachten, der besonders bei MySQL sehr niedrig ausfällt. Zudem fallen bei der Betrachtung von Abbildung 6.27 die hohen Schwankungen in der Latenzmessung bei HBase auf. Diese können allerdings an den extrem kurzen Antwortzeiten liegen, die durch das clientseitige Caching verursacht werden. Während Cassandra ab einer Serverzahl von vier Knoten eine konstante Antwortzeit aufweisen kann, steigt diese bei Project Voldemort mit der Anzahl der Server stetig an (Tabelle 6.7).

Server #	MySQL	Redis	Voldemort	HBase	Cassandra
1	1,0	1,0	1,0	1,0	1,0
2	0,6	0,7	1,1	1,4	1,7
4	0,2	0,6	1,3	0,8	1,9
8	0,1	0,5	1,6	0,6	2,0
12	0,1		1,9	1,6	2,0

Tabelle 6.7: Antwortzeit-Scaleup bei Schreiboperationen

Bei Schreiboperationen fallen in erster Linie die sehr guten Skalierbarkeitseigenschaften von MySQL auf. Parallel zum höchsten Durchsatz-Scaleup wurde ein nahezu *n*-facher Speedup gemessen. Bei den serverseitig verteilten Datenbanksystemen zeigt HBase die besten Skalierbarkeitseigenschaften.

6.3.2 Relationale Datenbanken

Die schlechten Skalierbarkeitseigenschaften von relationalen Datenbanksystemen sind seitens der NoSQL-Bewegung der am häufigsten genannte Kritikpunkt, der gegen einen Einsatz eines relationalen Datenbanksystems in Big Data-Anwendungsfällen sprechen soll. Neben den Veröffentlichungen von Google, Amazon, Yahoo und Facebook wird dieses Argument in den Prospekten sämtlicher NoSQL-Datenbanksysteme, auf den Webseiten führender Internetunternehmen sowie in der NoSQL Community immer wieder aufgeführt.

Die Leistungsfähigkeit der in den meisten Anwendungsfällen dominierenden Leseoperationen lässt sich mithilfe von Caches und Replikation horizontal skalieren. Selbst wenn die Datenmenge oder die Performanz von Schreiboperationen eine Verteilung auf mehrere Verarbeitungsrechner erforderlich macht, stehen mit der funktionalen Partitionierung sowie horizontaler und vertikaler Fragmentierung altbewährte Techniken zur Verfügung, um relationale Datenbanken auf mehrere hundert Rechner effizient verteilen zu können. Um bei der Verteilung eine lineare Skalierbarkeit zu erzielen, müssen die oben erwähnten Gesetze von Amdahl [8] und Gunther [75] berücksichtigt werden. Verteilte relationale Datenbanksysteme müssen demnach nicht nur auf serverübergreifende Verbundoperationen verzichten, sondern auch die Anzahl und Länge von nicht parallelisierbaren Transaktionen minimieren. Außerdem müssen pessimistische Sperrprotokolle und serverübergreifende Integritätsprüfungen vermieden werden.

Durch Denormalisierung sowie den Verzicht auf Transaktionen und konsistenzsichernde Maßnahmen verlieren relationale Datenbanken einen Großteil ihrer Vorteile. Darüber hinaus muss die Verteilung applikationsseitig realisiert werden, da relationale Datenbanken in der Regel keine Unterstützung für eine verteilte Datenhaltung besitzen. Die applikationsseitige Verteilung eines relationalen Datenbanksystems ist mit einer hohen Komplexität verbunden. SQL-Statements müssen angepasst, Anfragen an die zuständigen Server weitergeleitet, Ergebnisse aggregiert und Konsistenzprüfungen applikationsseitig durchgeführt werden. Zwar existieren einige Bibliotheken, welche einen Teil dieser Aufgaben übernehmen können, dennoch ist der Aufwand, um ein bestehendes relationales Datenbanksystem horizontal zu skalieren, sehr hoch. Es ist deshalb häufig wesentlich einfacher, eine Applikation zu entwickeln, die von Beginn an auf verteilte Datenhaltung ausgelegt ist, als eine bestehende Applikation dahingehend zu ändern [155, S.533].

Das homogene Datenmodell und die ausschließlich primärschlüsselbasierten Anfragen des YCS-Benchmarks lassen sich hingegen sehr einfach auf ein relationales Datenbanksystem verteilen. Hierzu bieten sich horizontal fragmentierte Tabellen und ein applikationsseitig implementierter Consistent Hashing Algorithmus an [146, S.5]. Da die einzelnen MySQL-Server in diesem Fall völlig unabhängig voneinander sind und Datensätze gezielt auf einer Instanz eingefügt, aktualisiert und gelesen werden, entsteht keine Kommunikation zwischen den einzelnen Servern, welche die Skalierbar-

keit des Systems einschränken könnte. Es ist daher nicht verwunderlich, dass das applikationsseitig verteilte MySQL Datenbanksystem im Benchmark von Rabl, Sadoghi, Jacobsen et al. [146] bei Schreibanfragen sehr gute Skalierbarkeitseigenschaften aufweisen kann. Der Nachteil dieses Ansatzes liegt in der höheren Belastung auf Applikationsebene, was sich vor allem bei steigender Last bemerkbar macht. So waren die Clients nicht in der Lage, den benötigten Durchsatz zu erzeugen, da jeder Thread eine eigene JDBC-Verbindung aufbauen musste. Infolgedessen entstand ein wesentlich höherer Overhead im Vergleich zu den Threads, die mit NoSQL-Datenbanken kommunizierten. Bedingt durch den geringeren Durchsatz sank anschließend die durchschnittliche Latenz einer Anfrage [146].

6.3.3 Key Value Stores

Key Value Stores lassen sich dank ihres Minimalismus sehr einfach und effizient auf mehrere Verarbeitungsrechner verteilen. So ist beispielsweise eine Fragmentierung des Datenmodells überflüssig, da die einzelnen Datensätze in Form von Schlüssel-Wert-Paaren bereits als kleinste Verteilungseinheit vorliegen. Auch ist eine Verletzung der Gesetze von Amdahl [8] und Gunther [75] praktisch ausgeschlossen, weil Key Value Stores keine Funktionen besitzen, die eine Kommunikation mit mehreren Servern benötigen. Es werden keine Verbundoperationen, Transaktionen oder Integritätsprüfungen unterstützt.

Dennoch zeigt der Key Value Store Redis in den Benchmarks von Rabl, Sadoghi, Jacobsen et al. [146] die schlechtesten Skalierbarkeitseigenschaften. Der Grund hierfür liegt darin, dass die Cluster-Version von Redis sich zum Zeitpunkt des Benchmarks noch in der Entwicklung befand und der Key Value Store deswegen applikationsseitig verteilt werden musste. Hierdurch ergab sich ein erheblicher Mehraufwand auf der Clientseite, sodass dieser nicht den benötigten Durchsatz generieren konnte. Zudem sorgten Probleme bei der Lastbalancierung dafür, dass einigen Instanzen mehr Daten zugewiesen wurden, als sie mit ihrer Hauptspeichergröße hätten verarbeiten können. Aufgrund der zu geringen Last bei steigender Serveranzahl ist in den Benchmarks bei Redis eine sinkende Latenz zu beobachten. Durch die Instabilität des Clusters waren außerdem nur Messungen bis zu maximal acht Verarbeitungsrechnern möglich [146, S.6].

Project Voldemort ist eine Nachimplementierung des Amazon Key Value Stores Dynamo. Dieser ist in erster Linie auf eine sehr gute Skalierbarkeit ausgerichtet und ermöglicht ein dynamisches Hinzufügen und Entfernen von Verarbeitungsrechnern zur Laufzeit, ohne dabei Anpassungen auf Applikationsebene durchführen zu müssen. Diese Eigenschaften spiegeln sich in den Ergebnissen von Rabl, Sadoghi, Jacobsen et al. [146] wieder. Hierbei zeigt Project Voldemort sowohl bei Lese- als auch bei Schreibanfragen einen linearen Durchsatz-Scaleup bei einer gleichzeitig konstanten Antwortzeit.

6.3.4 Document Stores

Document Stores werden in den Benchmarks von Rabl, Sadoghi, Jacobsen et al. [146] nicht berücksichtigt. Allerdings werden die Skalierbarkeitseigenschaften von MongoDB, HBase und Cassandra in einer Studie von DataStax [41] miteinander verglichen. Da DataStax ein Anbieter von Cassandra ist und der Studie folglich die nötige Neutralität fehlt, werden im Folgenden die Ergebnisse von Cassandra nicht berücksichtigt.

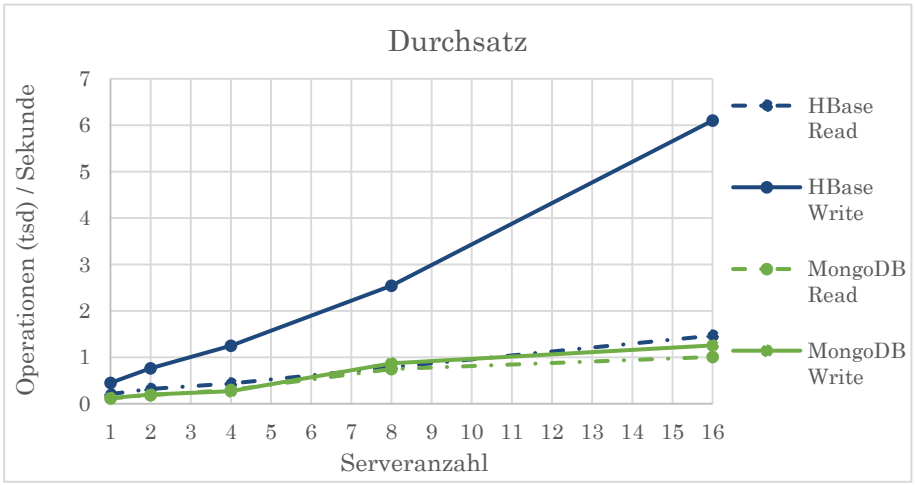


Abbildung 6.28: Skalierbarkeitseigenschaften von MongoDB (Durchsatz)

Bei der Messung des Durchsatzes weist MongoDB sowohl bei Lese- als auch bei Schreibanfragen ein linear mit der Serveranzahl steigendes Wachstum auf (Abbildung 6.28). Der Unterschied zwischen diesen beiden Operationen ist minimal und kann deshalb vernachlässigt werden. Interessant ist der Vergleich zum Durchsatzwachstum im Verhältnis zu HBase. Lese- und Schreiboperationen skalieren in ähnlicher Weise, wie es Leseoperationen bei HBase tun. Schreiboperationen verzeichnen bei HBase allerdings ein wesentlich höheres Wachstum als bei MongoDB.

Die bereits in den vorherigen Kapiteln bei MongoDB festgestellte schwache Leistungsfähigkeit bei Schreiboperationen spiegelt sich nicht nur beim Durchsatz-Scaleup wider. Auch der Antwortzeit-Scaleup weist einen unverhältnismäßig hohen Wert auf, der linear mit der Serveranzahl nahe dem Wert n steigt (Abbildung 6.29). Leseanfragen zeigen dagegen ein ähnliches Verhalten wie Lese- und Schreibanfragen bei HBase und besitzen einen konstanten Antwortzeit-Scaleup im Bereich von eins.

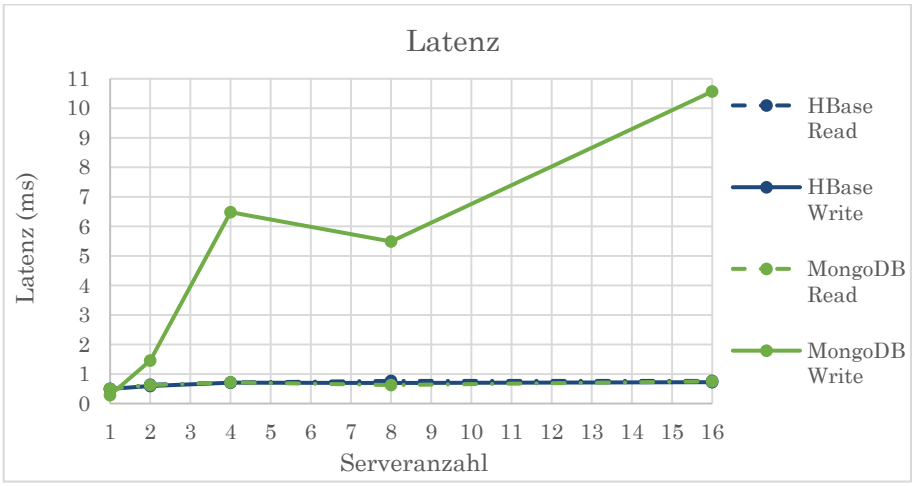


Abbildung 6.29: Skalierbarkeitseigenschaften von MongoDB (Latenz)

Berücksichtigt man die beiden gemessenen Scaleup-Werte, besitzt MongoDB gute Skalierbarkeitseigenschaften bei Leseanfragen. Schreibanfragen skalieren dagegen ausgesprochen schlecht. Einen nicht unerheblichen Anteil an diesem Verhalten dürfte das sehr pessimistische Synchronisationsverfahren haben. Jeder MongoDB-Server in einem Cluster verfügt über eine eigene Lese-Schreib-Sperre, die einer Schreiboperation einen exklusiven Zugriff auf die vollständige Datenbank sichert. Ist eine Sperre gesetzt, können andere Lese- und Schreiboperationen in diesem Zeitraum nicht ausgeführt werden [123, S.691]. Darüber hinaus werden die Replikate eines Replica Sets mithilfe von Consensus Protokollen synchronisiert, wodurch Schreiboperationen erst bestätigt werden, wenn diese auf einer Mehrheit der Server erfolgreich durchgeführt wurden. Durch diese hohen Konsistenzgarantien ist MongoDB in der Verarbeitungsgeschwindigkeit von Schreiboperationen anderen Datenbanken unterlegen. Obendrein verhindert die hierfür notwendige Kommunikation zwischen den einzelnen Servern eine bessere Skalierbarkeit.

Ein MongoDB-Cluster besteht aus mindestens drei Komponenten. Ein Router ist für die Kommunikation zwischen Client und Cluster zuständig, ein Konfigurationsserver koordiniert die Verteilung und ein Datenbankknoten verwaltet die einzelnen Dokumente. Trotz der verschiedenen Komponenten hält sich der administrative Aufwand bei der Einrichtung und Wartung eines MongoDB-Clusters im überschaubaren Rahmen. CouchDB unterstützt nativ keine Verteilung der Dokumente auf unterschiedliche Verarbeitungsrechner. Ist eine Clusterlösung von CouchDB erwünscht, müssen Drittanbieterprodukte wie BigCouch, Lounge oder Pillow eingesetzt werden [88, 103]. Diese verteilen Dokumente mittels Consistent Hashing Algorithmen auf die verschiedenen unabhängigen CouchDB Instanzen.

6.3.5 Column Family Stores

Die beiden Column Family Stores Cassandra und HBase sind auf die Verarbeitung sehr großer Datenmengen ausgelegt. Beide Systeme unterstützen keine Verbundoperationen, bieten keine Transaktionen und prüfen keine Integritätsbedingungen. Entsprechend weisen beide Datenbanken trotz ihrer unterschiedlichen Verteilungsarchitekturen eine nahezu lineare Skalierbarkeit sowohl bei Lese- als auch bei Schreibanfragen auf und benötigen zudem keine Anpassungen auf Applikationsebene.

Cassandra verwendet ebenso wie Amazons Dynamo und Project Voldemort Consistent Hashing, um einzelne Datensätze basierend auf ihrem Primärschlüssel auf unterschiedliche Verarbeitungsrechner zu verteilen. In der Shared-Nothing-Architektur kann dynamisch zur Laufzeit ein Verarbeitungsrechner zum Cluster hinzugefügt oder entfernt werden. Nach jeder Änderung in der Clusterarchitektur werden einzelne Datensätze automatisch umverteilt. Dieser Prozess kann mitunter mehrere Stunden in Anspruch nehmen, wie sich im Elastizitätstest von Copper et al. [35] als auch in praktischen Tests während dieser Arbeit herausstellte.

Die Verteilung der Datensätze erfolgt in HBase über einen zentralen Koordinator. Dieser ermöglicht eine im Vergleich zu Cassandra wesentlich bessere Lastverteilung [35], stellt dafür jedoch einen sogenannten Single-Point-of-Failure dar. Da HBase neben dem Koordinator-Server und einem Verarbeitungsrechner zusätzlich den Verzeichnisdienst Chubby benötigt, besteht ein HBase-Cluster immer aus mindestens drei Komponenten. Dementsprechend höher ist der Einrichtungs- und Wartungsaufwand eines HBase-Clusters gegenüber dem eines Cassandra-Clusters.

6.3.6 Zusammenfassung

In diesem Unterkapitel wurde das Skalierbarkeitsverhalten von relationalen und nichtrelationalen Datenbanken untersucht. Hierzu wurden Messergebnisse verschiedener Benchmarks herangezogen und mithilfe der herausgearbeiteten Verteilungskonzepte analysiert und interpretiert. Dabei stellten sich zwei Hauptkriterien heraus, die für die Bewertung des Verteilungsverhaltens eines Datenbanksystems herangezogen werden können. Ist aufgrund steigender Anforderungen eine Verteilung eines Datenbanksystems auf mehrere Server notwendig, sind neben dem hierfür notwendigen Aufwand die Skalierbarkeitseigenschaften des Datenbanksystems von entscheidender Bedeutung.

Aufwand: Der Aufwand, der zur Verteilung eines Datenbanksystems auf mehrere Verarbeitungsrechner benötigt wird, hängt sehr stark vom Funktionsumfang des Datenbanksystems ab. Im schlechtesten Fall verfügt das System über keine integrierten Verteilungsmechanismen, sodass die Verteilung und Anfrageverarbeitung manuell auf Applikationsebene oder durch den Einsatz von Drittanbieterprodukten realisiert werden muss. Manche Systeme bieten neben integrierten Verteilungsmechanismen auch eine umfangreiche Werkzeugunterstützung, welche die Administration eines Clusters deutlich erleichtert.

Skalierbarkeit: Die Skalierbarkeitseigenschaften eines Datenbanksystems bestimmen, wie gut sich die Last auf mehrere Rechner verteilen lässt. Idealerweise steigt die Leistungsfähigkeit des Systems linear mit der Anzahl der hinzugefügten Rechner. Da das Skalierbarkeitsverhalten stark von der zu verarbeitenden Last abhängig ist, werden Lese- und Schreiboperationen getrennt voneinander betrachtet.

Ein weiteres wichtiges Kriterium eines verteilten Datenbanksystems sind dessen Konsistenzgarantien. Im Umfeld von NoSQL-Datenbanken wird häufig das von Brewer vorgestellte CAP-Theorem herangezogen [20, 70], um die Konsistenzeigenschaften eines Datenbanksystems klassifizieren zu können. Demzufolge kann ein verteiltes Datenbanksystem nur zwei der drei Eigenschaften Konsistenz, Verfügbarkeit oder Partitionstoleranz gleichzeitig erfüllen.

Konsistenz und Partitionstoleranz: Tritt infolge eines Netzwerkfehlers eine Partitionierung des verteilten Datenbanksystems auf, werden anschließend seitens des Datenbankmanagementsystems nicht mehr alle Operationen unterstützt, um die Konsistenz der einzelnen Knoten zu gewährleisten.

Verfügbarkeit und Partitionstoleranz: In Anwendungsfällen, in denen eine dauerhafte Verfügbarkeit aller Operationen trotz des Auftretens von Netzwerkpartitionen die höchste Priorität hat, müssen kurzzeitige Inkonsistenzen, die zu einem späteren Zeitpunkt durch den Einsatz von Synchronisationsmechanismen beseitigt werden, toleriert werden.

Konsistenz und Verfügbarkeit: Da Netzwerkpartitionen überwiegend bei der Kommunikation über unzuverlässige Netze entstehen, können diese bei Datenbanksystemen vernachlässigt werden, die in hochverfügbaren Netzwerken lokal verteilt werden [147, 163].

Zentral verteilte Systeme, wie MongoDB und HBase, ziehen im Fall von Netzwerkpartitionierungen die Konsistenz der vollständigen Verfügbarkeit aller Anfragen vor. Dezentral verteilte Systeme, wie Project Voldemort und Cassandra, werden dagegen als Systeme mit einer „eventuellen Konsistenz“ (engl. „eventual consistency“) charakterisiert, die zu Gunsten der Verfügbarkeit die sofortige Konsistenz der Daten vernachlässigen. Allerdings verlaufen die Grenzen zwischen den von Brewer eingeführten Klassen in der Praxis eher fließend. Sowohl Cassandra als auch Project Voldemort sind abhängig von der Anfrage in der Lage, verschiedene Konfigurationen der konsistenzbestimmenden Parameter entgegen zu nehmen.

Die Bewertung des Aufwandes, der für die Verteilung eines der hier vorgestellten Datenbanksysteme benötigt wird, und die Bewertung der bereitgestellten Konsistenzgarantien basieren auf den herausgearbeiteten Verteilungskonzepten und auf praktischen Erfahrungen, die während des Schreibens dieser Arbeit im Umgang mit den verschiedenen Datenbanksystemen gesammelt werden konnten. Die Skalierbarkeitseigenschaften der Systeme werden mithilfe der in diesem Kapitel vorgestellten Benchmarks bewertet. Die entsprechenden Ergebnisse für relationale Datenbanken

(RDBMS), Key Value Stores (KVS), Document Stores (DOC) und Column Family Stores (CFS) sind in Tabelle 6.8 veranschaulicht. Die Konsistenzgarantien von MySQL und Redis hängen von der applikationsseitig eingesetzten Verteilungsstrategie ab.

Datenbankklasse	RDBMS	KVS		DOC	CFS	
Datenbanksystem	MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Aufwand	−−	−	+++	+++	+	+++
Skalierbarkeit: Lesen	−	−−	+++	+++	+++	++
Skalierbarkeit: Schreiben	−	−−	++	−	+++	+++
Konsistenz	+ −	+ −	−	+	+	−
Verfügbarkeit	+ −	+ −	+	−	−	+

Tabelle 6.8: Gegenüberstellung der Skalierbarkeitseigenschaften

Relationale Datenbanken bieten keine integrierte Unterstützung für eine verteilte Datenhaltung. Aus diesem Grund müssen Aufgaben wie Transaktionsmanagement, Anfrageverarbeitung, Fehlererkennung sowie Allokation und Fragmentierung applikationsseitig beziehungsweise durch Drittanbieterprodukte realisiert werden. Neben dem höheren administrativen Aufwand sind die hierdurch steigenden Anforderungen auf Clientseite als Nachteile von verteilten relationalen Datenbanken zu nennen. Werden leistungsstarke Clients eingesetzt und Verbundoperationen und verteilte Transaktionen nicht benötigt, lassen sich sowohl Lese- als auch Schreiboperationen bei relationalen Datenbanken sehr gut horizontal skalieren. Abgesehen davon, dass bei diesem Ansatz auf viele Vorteile relationaler Datenbanken verzichtet wird, ist zu bemerken, dass sich auf diese Art und Weise jedes Datenbanksystem skalieren lässt.

Die in dieser Arbeit vorgestellten Key Value Stores unterscheiden sich in ihrer Unterstützung für eine verteilte Datenhaltung. Redis ist nicht für einen Einsatz auf verteilten Datenbanken ausgelegt und muss daher manuell verteilt werden. Auch wenn durch den geringen Funktionsumfang von Key Value Stores ein relativ geringer Aufwand bei der Fragmentierung, Allokation und Anfrageverarbeitung anfällt, ist angesichts der hauptspeicherbasierten Datenhaltung eine sensible Lastverteilung essenziell, um Redis auf mehrere Server verteilen zu können. In den Benchmarks von Rabl, Sadoghi, Jacobsen et al. [146, S.6] konnte Redis deswegen nur auf einen Cluster mit maximal acht Knoten zuverlässig verteilt werden. Project Voldemort ist dagegen von Beginn an auf eine verteilte Datenhaltung ausgelegt und daher verhältnismäßig leicht zu administrieren. Zudem weist der Key Value Store sehr gute Skalierbarkeitseigenschaften sowohl bei Lese- als auch bei Schreiboperationen auf.

Ebenso wie bei Key Value Stores unterscheiden sich die beiden in dieser Arbeit vorgestellten Document Stores in ihrer Unterstützung für eine verteilte Datenhaltung.

Während sich CouchDB nur clientseitig beziehungsweise unter Zuhilfenahme von Drittanbieterprodukten verteilen lässt, sind alle für eine Verteilung notwendigen Funktionalitäten in MongoDB integriert. Entsprechend einfach ist die Administration eines MongoDB-Clusters. Leseanfragen können in MongoDB horizontal skaliert werden. Schreibanfragen weisen dagegen wegen des pessimistischen Synchronisationsverfahrens sehr schlechte Skalierbarkeitseigenschaften auf.

Bei Column Family Stores handelt es sich in der Regel um verteilte Datenbanksysteme. Der administrative Aufwand hängt dabei vom eingesetzten System ab. Während sich Cassandra dank der dezentralen Architektur sehr einfach verteilen lässt, ist die Einrichtung eines HBase-Clusters mit einem deutlich höheren Aufwand verbunden. Lese- und Schreibanfragen können sowohl bei Cassandra als auch bei HBase linear horizontal skaliert werden.

Die Verarbeitung sehr großer Datenmengen ist vermutlich die wichtigste Anforderung von Big Data-Anwendungsfällen an die zugrunde liegenden Datenbanksysteme. Diese sind idealerweise in der Lage, ihre Leistungsfähigkeit linear durch das Hinzufügen neuer Server zu erhöhen, sodass sie parallel mit den steigenden Anforderungen wachsen kann. Relationale Datenbanken wurden für solche Anwendungsfälle nicht konzipiert. Darum lassen sie sich nur mit einem sehr hohen Aufwand und mit einem Verzicht auf viele charakteristische Eigenschaften auf mehrere Rechner verteilen und skalieren. Die meisten NoSQL-Datenbanken sind dagegen von Beginn an auf eine verteilte Datenhaltung ausgelegt und lassen sich deshalb mit einem deutlich geringeren Aufwand horizontal skalieren. Besonders hervorzuheben sind Systeme, die auf den Konzepten von Amazons Dynamo und Googles BigTable basieren. Zu diesen gehören Project Voldemort, HBase und Cassandra.

6.4 Kritik an NoSQL-Datenbanken

NoSQL-Datenbanken sind auf spezifische Anforderungen von Big Data-Anwendungsfällen ausgelegt und sind dort in bestimmten Szenarien relationalen Datenbanken überlegen. Um jedoch die Vor- und Nachteile von NoSQL-Datenbanken realistisch einschätzen zu können, müssen neben den drei in diesem Kapitel behandelten Big Data-Eigenschaften weitere Kriterien berücksichtigt werden, welche ausschlaggebend für den erfolgreichen Einsatz eines Datenbanksystems sind.

Mit Aufkommen der ersten NoSQL-Datenbanken startete eine bis heute anhaltende, häufig sehr leidenschaftlich geführte Debatte zwischen Anhängern der verschiedenen Datenbanksysteme [79, 80, 116, 118, 160, 161, 162, 180, 189]. Auch wenn die ausgetauschten Argumente nur selten die gewünschte Neutralität und Qualität aufweisen, kristallisieren sich aus der Vielzahl an Anmerkungen einige grundlegende Kritikpunkte an NoSQL-Datenbanken heraus, die an dieser Stelle aufgegriffen werden.

Hierzu zählen unter anderem die in der Motivation dieser Arbeit genannten Nachteile der fehlenden Entwicklungsreife und des mangelnden Fachwissens. Weitere Kritikpunkte sind der geringe Support, die minimale Werkzeugunterstützung, fehlende Standardisierung, steigende Komplexität auf Applikationsebene und einer hoher Administrationsaufwand.

6.4.1 Fehlende Entwicklungsreife

Relationale Datenbanken existieren seit mehreren Jahrzehnten und besitzen einen entsprechenden Reifegrad. Die Gefahr, beim Einsatz eines relationalen Datenbanksystems auf bisher ungelöste Systemfehler zu stoßen, ist damit sehr gering. NoSQL-Datenbanken sind im Vergleich dazu erst seit wenigen Jahren auf dem Markt und haben in vielen Fällen, wenn überhaupt, gerade erst den Prototypenstatus verlassen. Die mit vielen dieser Systeme verbundene Instabilität und Fehleranfälligkeit spiegelt sich unter anderem in den Fehlermeldungen der Mailinglisten und in der sehr hohen Geschwindigkeit wieder, in der Anbieter neue Versionen bereitstellen.

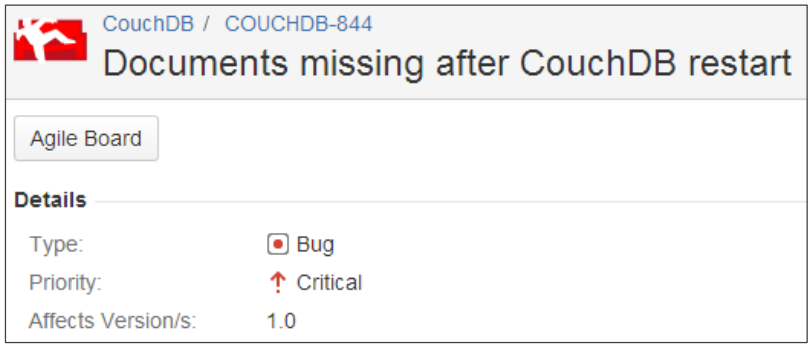


Abbildung 6.30: Kritische Fehlermeldung in der CouchDB-Version 1.0

In Abbildung 6.30 ist eine kritische Fehlermeldung abgebildet, die kurz nach Veröffentlichung der Version 1.0 von CouchDB bekannt wurde. Damals kam es nach einem Server-Neustart zu einem Verlust aller Dokumente. An Beispielen wie diesem wird deutlich, dass beim Einsatz vieler NoSQL-Datenbanken zwingend deren mangelnder Reifegrad berücksichtigt werden muss, um finanzielle Risiken durch Datenverlust, Verfügbarkeitsprobleme oder Sicherheitsmängel zu vermeiden. In einem Whitepaper von Oracle heißt es zu dieser Thematik:

> *"Oracle has been developing database-centric software for thirty years. Building any database is hard, to build one as good as Oracle's takes a long, long time. […] Go for the tried and true path. Don't be risking your data on NoSQL databases." – Alan Downing* [6]

Welche negativen Folgen die mangelnde Systemreife nach sich ziehen kann, wird unter anderem am Systemausfall von Foursquare im Jahr 2010 deutlich. Das ortsbezogene soziale Netzwerk setzte bereits sehr früh MongoDB ein, um die Check-Ins von Benutzern an bestimmten Orten zu verarbeiten. Die hierfür benötigten Daten wurden vollständig im Hauptspeicher zweier Amazon EC2-Instanzen mit 66 GB gespeichert. Infolge einer ungleichmäßigen Lastverteilung wurde 2010 die Kapazität einer dieser beiden Instanzen überschritten, sodass Foursquare für über 11 Stunden nicht mehr zur Verfügung stand [18].

6.4.2 Geringer Support

Auch wenn nach Angaben von Foursquare das MongoDB-Entwicklerteam sehr bemüht war das Problem zu lösen, wird an dieser Stelle ein weiterer Nachteil von NoSQL-Datenbanken deutlich. Diese Systeme bieten im Vergleich zu relationalen Datenbanken einen deutlich geringeren Support. Auch wenn es sich bei den meisten NoSQL-Datenbanken um Open-Source-Projekte handelt, existieren inzwischen einige Firmen, welche einen kommerziellen Support für diese Systeme anbieten. Zu den bekanntesten Unternehmen gehören Datastax[49], Cloudera[50], Teradata[51], Pivotal[52] und MongoDB, Inc. (ehemals 10gen). Allerdings sind diese Firmen allein in Anbetracht ihrer Größe nicht in der Lage, eine ähnlich umfangreiche Unterstützung wie Oracle, Microsoft oder IBM bereitzustellen.

6.4.3 Geringe Nachhaltigkeit

Einhergehend mit dem Support kann auch die Nachhaltigkeit eines Datenbanksystems ein wichtiges Entscheidungskriterium sein. Hierunter fällt zum einen die Entwicklungsdynamik eines Datenbanksystems. Wie bereits erwähnt, können sich einzelne Versionen bestimmter NoSQL-Datenbanken sehr stark voneinander unterscheiden, sodass unter Umständen Anpassungen auf Applikationsebene notwendig sind. Zum anderen fällt unter den Aspekt der Nachhaltigkeit die zukünftige Unterstützung des Projekts. Genauso schnell wie manche NoSQL-Datenbanken auf dem Markt erscheinen, können sie auch wieder von dort verschwinden.

6.4.4 Minimale Werkzeugunterstützung

Neben der ungleichmäßigen Verteilung von MongoDB wurde seitens Foursquare die fehlende Werkzeugunterstützung bemängelt. So gab es damals keine Möglichkeiten

[49] http://www.datastax.com
[50] http://www.cloudera.com
[51] http://www.teradata.de
[52] https://www.pivotal.io

die Lastverteilung zu überwachen oder über Frühwarnsysteme auf den Missstand hingewiesen zu werden [18]. Im Gegensatz dazu bieten Anbieter von relationalen Datenbanken, wie beispielsweise Oracle, eine umfangreiche Unterstützung an zusätzlicher Software. Neben Verwaltungswerkzeugen, Entwicklungsumgebungen und grafischen Benutzeroberflächen wird eine Vielzahl an Produkten von Drittanbietern bereitgestellt, mit denen Anfragen formuliert, Daten verwaltet oder Systeme administriert werden können.

6.4.5 Mangelndes Fachwissen

Relationale Datenbanken besitzen gegenüber NoSQL-Datenbanken den großen Vorteil, dass Anwender auf über 40 Jahre theoretische Grundlagen und praktische Erfahrungen zurückgreifen können. Da Entwickler während ihrer Ausbildung oder ihres Studiums in der Regel ausführlich mit relationalen Datenbanken in Berührung kommen, sind die Vor- und Nachteile dieser Systeme bestens bekannt. Im Gegensatz dazu ist über den praktischen Einsatz von NoSQL-Datenbanken vergleichsweise wenig bekannt. Da nur sehr wenig qualitativ hochwertige Literatur existiert und die Dokumentationen der verschiedenen Systeme einen zu geringen Umfang aufweisen, bilden Erfahrungsberichte in Entwicklerblogs häufig die einzige Informationsquelle für potenzielle Anwender. Die dort veröffentlichten Beiträge lassen allerdings nicht selten die benötigte Qualität und Neutralität vermissen. Durch den Mangel an qualitativ hochwertiger Fachliteratur und fehlende praktische Erfahrungen ist die Wahl eines für einen Anwendungsfall passenden Datenbanksystems mit einem sehr hohen Aufwand verbunden. Derzeit bildet das Erstellen von Prototypen und das ausführliche Testen verschiedener Systeme die einzige Möglichkeit, um verlässliche Informationen zur Eignung bestimmter Datenbanken und deren korrekter Verwendung zu erhalten.

6.4.6 Fehlende Standardisierung

Relationale Datenbanken sind standardisiert und weichen nur in bestimmten Punkten voneinander ab. Auch wenn sich die technischen Eigenschaften zwischen den einzelnen Systemen voneinander unterscheiden können, sind die wichtigsten Komponenten, wie beispielsweise die Anfragesprache SQL, gleich oder zumindest ähnlich. Demzufolge ist ein Wechsel zwischen einzelnen relationalen Datenbanksystemen in der Regel problemlos möglich. Im Vergleich dazu handelt es sich bei NoSQL-Datenbanken um eine Sammlung an sehr heterogenen Systemen. Selbst innerhalb der allgemeingültigen und in dieser Arbeit auch verwendeten Datenbankklassen unterscheiden sich die verschiedenen Systeme erheblich in ihren Eigenschaften. Aufgrund fehlender Standards existieren für NoSQL-Datenbanken keine systemübergreifenden Anfragesprachen oder Werkzeuge. Ein Wechsel von einem NoSQL-System zu einem anderen ist darum mit einem wesentlich höheren Aufwand verbunden, als es bei relationalen Datenbanken der Fall ist.

6.4.7 Zunehmende Komplexität auf Applikationsebene

NoSQL-Datenbanken verzichten auf sehr viele grundlegende Funktionalitäten von relationalen Datenbanken, um gegenüber diesen Systemen eine größere Flexibilität in der Datenmodellierung, eine höhere Performanz bei einfachen Lese- und Schreibanfragen und eine bessere Skalierbarkeit bei der Verarbeitung sehr großer Datenmengen erzielen zu können. Im Rahmen der konzeptuellen Aufarbeitung wurde dieser Verzicht auf Funktionalitäten in dieser Arbeit ausführlich dokumentiert. Er betrifft hauptsächlich die Ausdrucksstärke der Datenmodelle, den Funktionsumfang der Anfrageschnittstellen und die ACID-Garantien der Transaktionsverwaltungen. Dafür bieten NoSQL-Datenbanken Funktionalitäten, welche die Replikation und Verteilung der Daten auf mehrere Rechner deutlich erleichtern. Der aus den verschiedenen Konzepten resultierende Unterschied im Funktionsumfang ist in Tabelle 6.9 veranschaulicht.

Funktionalitäten	Relationale Datenbanken	Key Value Stores	Document Stores	Column Family Stores
Attributnamen	✓	✗	✓	✓
Datentypen	✓	✗	✓	✗
Sekundäre Zugriffspfade	✓	✗	✓	✗
Mengenoperatoren	✓	✗	✓	✓
Verbundoperationen	✓	✗	✗	✗
Transaktionen	✓	✗	✗	✗
Integritätsbedingungen	✓	✗	✗	✗
Synchronisationsmechanismen	✓	✗	✓	✓
Dauerhaftigkeit	✓	✗	✓	✓
Replikation	✗	✓	✓	✓
Verteilung	✗	✓	✓	✓

Tabelle 6.9: Vergleich des Funktionsumfangs

Der Verzicht auf Funktionalitäten zugunsten der Flexibilität, Performanz und Skalierbarkeit ist unproblematisch, solange diese nicht in einem Anwendungsfall benötigt werden. Sollte das allerdings doch der Fall sein, müssen sie durch einen zusätzlichen Implementierungsaufwand oder unter Zuhilfenahme von Drittanbieterprodukten auf Applikationsebene realisiert werden. Dieser Mehraufwand kann die Komplexität der Applikation sehr stark steigern, was wiederum höhere Entwicklungs- und Wartungskosten sowie eine größere Fehleranfälligkeit zur Folge hat. Zudem muss der sich hierdurch ergebende Performanzverlust berücksichtigt werden. So können sich die Geschwindigkeits- und Skalierbarkeitsvorteile von NoSQL-Datenbanken gegenüber relationalen Datenbanken schnell relativieren, wenn Fremdschlüsselbeziehungen oder sekundäre Zugriffspfade applikationsseitig aufgelöst und gewartet werden müssen:

> *„In summary, NoSQL will translate into lots of work for the application.“ – Michael Stonebraker [160]*

6.5 Zusammenfassung

Ziel dieses Kapitels war die systematische Herausarbeitung der Vor- und Nachteile, die sich durch den praktischen Einsatz von NoSQL-Datenbanken im Vergleich zu relationalen Datenbanken ergeben. Die hier getroffenen Aussagen basieren auf den Ergebnissen der in dieser Arbeit durchgeführten konzeptuellen Aufarbeitung sowie auf einer Reihe unabhängiger Benchmarks und liefern dadurch einen fachlich fundierten Beitrag zu der ansonsten überwiegend unsachlich geführten NoSQL-Diskussion. Mithilfe der gewonnenen Erkenntnisse soll vor allem unerfahrenen Anwendern der Einstieg in die alles andere als triviale Thematik der NoSQL-Datenbanken erleichtert werden. Angesichts der Fokussierung der NoSQL-Diskussion auf Big Data-Anwendungsfälle standen in diesem Kapitel besonders der Umgang mit einer hohen Datenvielfalt, die Verarbeitungsgeschwindigkeit von Anfragen sowie die Skalierbarkeitseigenschaften der Datenbanksysteme im Vordergrund. Anschließend wurden weitere Kriterien vorgestellt, die vor dem Einsatz eines konkreten Datenbanksystems individuell berücksichtigt werden müssen. Hierzu zählen Entwicklungsreife, Support, Nachhaltigkeit, Werkzeugunterstützung, Fachwissen, Standardisierung, Administrationsaufwand und Komplexität auf Applikationsebene.

Key Value Stores besitzen ein minimalistisches Datenmodell, das keinen Schemarestriktionen unterworfen ist. Dennoch sind diese Systeme wegen ihrer minimalen Ausdrucksstärke und der geringen Anfragemächtigkeit am wenigsten von allen in dieser Arbeit vorgestellten Datenbanksystemen für die Verarbeitung von heterogenen Datensätzen geeignet. Die Stärke von Key Value Stores liegt in deren Geschwindigkeit. Sowohl Lese- als auch Schreiboperationen können diese Systeme am schnellsten verarbeiten. Allerdings werden diese Vorteile auf Kosten der ACID-Eigenschaften erzielt, sodass in diesen Systemen nur konsistenzunkritische Daten verarbeitet werden sollten. Werden die Schlüssel-Wert-Paare dauerhaft gespeichert, fällt der Performanzvorteil gegenüber anderen Datenbanksystemen deutlich geringer aus. Key Value Stores wie Project Voldemort lassen sich mit minimalem Aufwand auf mehrere Rechner verteilen und dabei linear skalieren. Der Key Value Store Redis verfügt dagegen derzeit noch über keine integrierten Verteilungsfunktionalitäten.

Document Stores sind in der Lage, selbstbeschreibende Datensätze mit einem hohen Funktionsumfang zu verarbeiten. Zwar unterstützen diese Systeme keine Verbundoperationen und keine Transaktionen, doch lassen sich diese fehlenden Funktionalitäten durch eine geschickte Datenmodellierung kompensieren. Document Stores sind daher sehr gut für die persistente Verwaltung heterogener Datensätze geeignet. Bei der Verarbeitung von homogenen Datensätzen weisen diese Systeme allerdings sowohl bei Lese- als auch bei Schreibanfragen die schlechteste Performanz auf. Während sich der Document Store CouchDB nicht ohne Drittanbieterprodukte verteilen lässt, ermöglicht MongoDB eine sehr einfach zu administrierende Verteilung. Allerdings ist bei diesem zentral koordinierten Ansatz ausschließlich bei Leseoperationen eine lineare Skalierbarkeit zu beobachten. Schreibanfragen lassen sich dagegen schlecht skalieren.

Column Family Stores verfügen über ein schemafreies Datenmodell, auf das heterogene Datensätze problemlos abgebildet werden können. Da diese Systeme jedoch ausschließlich Anfragen auf den Primärschlüssel akzeptieren und einen geringeren Funktionsumfang als relationale Datenbanken besitzen, sind Column Family Stores nur in bestimmten Anwendungsfällen für die Verarbeitung von heterogenen Datensätzen heranzuziehen. Einer dieser Anwendungsfälle ist die Verarbeitung sehr großer Datensätze. Eine der großen Stärken von Column Family Stores liegt in der performanten Verarbeitung von persistenten Schreiboperationen. In diesem Bereich sind sie allen anderen Datenbanksystemen überlegen. Dafür zeigen diese Systeme allerdings bei Leseanfragen eine schlechte Performanz. Column Family Stores sind in der Regel auf mehrere Rechner verteilt. Während der Umfang des hierfür notwendigen Aufwandes vom konkreten Datenbanksystem abhängig ist, weisen diese Systeme produktübergreifend eine sehr gute Skalierbarkeit auf.

Relationale Datenbanken verfügen über das mächtigste Datenmodell und die meisten Anfragefunktionalitäten. In Anbetracht der hohen Restriktion, die das globale Schema dieser Systeme impliziert, lassen sich semistrukturierte Datensätze allerdings ab einem gewissen Grad an Heterogenität nicht mehr effizient mit relationalen Datenbanken verarbeiten. Relationale Datenbanken besitzen eine hohe Performanz bei Leseanfragen und eine vergleichsweise schlechte Leistungsfähigkeit bei Schreibanfragen. Zudem sind diese Systeme nicht für eine verteilte Datenhaltung konzipiert. Um die Leistungsfähigkeit von relationalen Datenbanken linear mit der Rechneranzahl skalieren lassen zu können, sind ein Verzicht auf viele Funktionalitäten sowie ein sehr hoher Aufwand auf Applikationsebene notwendig.

Funktionalitäten	Relationale Datenbanken	Key Value Stores	Document Stores	Column Family Stores
Ausdrucksstärke	++	−−	+	−
Anfragemächtigkeit	++	−−	+	−
ACID	++	−−	+	−
Performanz: Lesen	+	++	−−	−
Performanz: Schreiben	−	++	−−	+
Skalierbarkeit: Lesen	−−	+	+	+
Skalierbarkeit: Schreiben	−−	+	−	++

Tabelle 6.10: Eigenschaften der Datenbanksysteme

In Tabelle 6.10 sind die gewonnenen Erkenntnisse unter Berücksichtigung der hier genannten Systemeigenschaften gegenübergestellt. Es fällt auf, dass relationale Datenbanken und Document Stores ihre Stärken auf konzeptueller Ebene haben, während Key Value Stores und Column Family Stores auf interner Ebene den anderen Datenbanksystemen überlegen sind.

7

AUSWAHL EINES DATENBANKSYSTEMS

Kapitelinhalt

- Analyse von Anwendungsfällen
- Bewertung der Auswahlkriterien
- Bereitstellung eines Kriterienkatalogs

Im Laufe dieser Arbeit wurden die Konzepte von NoSQL-Datenbanken systematisch aufgearbeitet und die Vor- und Nachteile einzelner Systeme im Vergleich zu relationalen Datenbanken hervorgehoben. NoSQL-Datenbanken sind demnach hochspezialisierte Systeme, die im Gegensatz zu relationalen Datenbanken kein breites Spektrum an verschiedenen Anwendungsfällen abdecken, sondern auf spezifische Anforderungen bestimmter Anwendungsfälle ausgerichtet sind. Durch diesen hohen Grad an Spezialisierung ist für die Wahl eines für einen bestimmten Anwendungsfall passenden Datenbanksystems ein hohes Maß an Expertise notwendig. Ohne ein tiefes Hintergrundverständnis und umfangreiche praktische Erfahrungen im Umgang mit NoSQL-Datenbanken ist diese komplexe Aufgabe mit einem unverhältnismäßig hohen Aufwand verbunden.

In diesem Kapitel wird ein neuartiger Kriterienkatalog vorgestellt, mit dessen Hilfe auch unerfahrene Anwender in die Lage versetzt werden, mit einem relativ geringen Aufwand aus der Vielzahl an zur Verfügung stehenden Datenbanksystemen das für einen Anwendungsfall ideale System auswählen zu können. Die Auswahlkriterien und Bewertungen, die in diesem Kriterienkatalog herangezogen werden, basieren auf den Erkenntnissen der konzeptuellen Aufarbeitung, den Ergebnissen der im vorherigen Kapitel erfolgten Vergleiche und einer in diesem Kapitel durchgeführten umfangreichen Analyse von Big Data-Anwendungsfällen. Dank einer individuellen Gewichtung der verschiedenen Auswahlkriterien können die spezifischen Anforderungen eines jeden Anwendungsfalls feingranular berücksichtigt werden.

7.1 Analyse von Anwendungsfällen

In Kapitel 6 wurden die verschiedenen relationalen und nichtrelationalen Datenbanksysteme hinsichtlich der spezifischen Anforderungen von Big Data-Anwendungsfällen untersucht und miteinander verglichen. Anschließend wurden im Rahmen der Kritik an NoSQL-Datenbanken weitere Kriterien genannt, die bei der Wahl eines Datenbanksystems berücksichtigt werden müssen. Diese Kriterien basieren zum Großteil auf den Ergebnissen der konzeptuellen Ausarbeitung der unterschiedlichen Datenbanksysteme. Um die praktische Relevanz der Kriterien evaluieren und weitere Auswahlkriterien identifizieren zu können, werden im Folgenden verschiedene Praxisbeispiele analysiert, in denen NoSQL-Datenbanken als Alternative zur relationalen Standardlösung eingesetzt werden. Neben der Veranschaulichung verschiedener Modellierungsvarianten stehen hierbei vor allem die genannten Systemeigenschaften im Vordergrund, auf deren Basis sich die verschiedenen Unternehmen für oder gegen ein bestimmtes Datenbanksystem entschieden haben.

7.1.1 Key Value Stores

Key Value Stores sind minimalistische Systeme, in denen einzelne Datensätze ausschließlich über jeweils einen Schlüssel gelesen und abgelegt werden können. Angesichts der fehlenden ACID-Eigenschaften sollten in Key Value Stores flüchtige Daten mit geringen Konsistenzanforderungen hinterlegt werden. Da die meisten Systeme ihre Daten vollständig im Arbeitsspeicher verwalten, weisen Key Value Stores in der Regel die höchste Leistungsfähigkeit bei einfachen Lese- und Schreibanfragen auf. Key Value Stores, die auf Amazons Dynamo basieren, lassen sich mit einem minimalen Aufwand auf mehrere Server verteilen und dabei horizontal skalieren. Typische Einsatzgebiete von Key Value Stores sind Caching-Systeme, Session Management, Echtzeitanalysen, Activity Feeds, Stapelverarbeitung und Warenkörbe in Onlineshops.

Caching-Systeme

Key Value Stores werden häufig wegen ihrer hohen *Leistungsfähigkeit bei Leseoperationen* in Caching-Systemen eingesetzt. Hier können Leseanfragen deutlich beschleunigt werden, indem die benötigten Ergebnisse aus anderen Datenbanksystemen zuvor berechnet, in Key Value Stores redundant zwischengespeichert und die Anfragen auf diese leistungsstärkeren Systeme umgeleitet werden. Beispiele hierfür sind häufig gelesene Artikel in Onlinemedien, Top-Seller-Listen in E-Commerce-Applikationen oder neueste Beiträge in sozialen Netzwerken. In diesem Bereich hat sich vor allem das verteilte Caching-System Memcached etabliert. Memcached verbindet den ungenutzten Hauptspeicher der zugrunde liegenden Datenbankserver zu einem großen Key Value Store (siehe Kapitel 3.1). Prominente Nutzer von Memcached sind Facebook [134], Zynga [7], Tumblr [113], Wikipedia, YouTube, Flickr und Twitter [117].

Session Management

Key Value Stores können zur Verwaltung von Sitzungsdaten in Webanwendungen herangezogen werden. Da es sich bei HTTP um ein zustandsloses Protokoll handelt, müssen einzelne Client-Anfragen anhand einer ID beim Webserver eindeutig identifiziert werden, um sie einer Sitzung (englisch „session") zuordnen zu können. Der Webserver ist dann in der Lage, bei jedem Seitenaufruf die zugehörigen Sitzungsdaten zu laden und kann so auch komplexe Transaktionen realisieren, die mehrere Seitenzugriffe umfassen. Sitzungsdaten werden ausschließlich über die Session-ID angesprochen und haben hohe Anforderungen an *Lese- und Schreibgeschwindigkeiten*. Darüber hinaus besitzen sie geringe Konsistenzanforderungen, da sie nur für einen begrenzten Zeitraum gültig sind. Als weitere Auswahlkriterien nennt Amazon die hohen *Skalierbarkeits- und Verfügbarkeitsanforderungen*, die eine Verwaltung von mehreren 100.000 aktiven Sitzungen mit sich bringt [45].

Echtzeitanalysen

Business Intelligence Lösungen unterstützen Unternehmen nicht mehr nur bei der strategischen Planung, sondern werden in zunehmendem Maße auch für Entscheidungsfindungen im operativen Geschäft (*Operational Intelligence*) herangezogen. Dabei werden unternehmensrelevante Informationen, wie beispielsweise Verkaufszahlen oder Besucherstatistiken, in Echtzeit generiert und analysiert. Basierend auf den Ergebnissen können dann automatisierte Prozesse angestoßen oder Statistiken auf Managementebene bereitgestellt werden. Im Gegensatz zu Berechnungen in Data Warehouse Lösungen, die häufig mehrere Stunden dauern, besitzen Echtzeitanalysen sehr hohe *Geschwindigkeitsanforderungen an Lese- und Schreibanfragen*. Hier müssen Aggregationen mit minimaler Latenz permanent aktualisiert werden. Aufgrund ihrer hohen Performanz sind Key Value Stores prädestiniert für Echtzeitanalysen. Konsistenzunkritische Daten, wie beispielsweise Aggregationen von Seitenaufrufen, Downloads oder Klicks, können performant im Hauptspeicher generiert werden. Vor allem der Key Value Store Redis bietet sich in Anbetracht seines großen Funktionsumfangs und der *Bereitstellung von komplexen Datenstrukturen* für diese Aufgaben an. So verwendet beispielsweise der Social Bookmark Anbieter Digg[53] die in Redis enthaltene Inkrementierungsfunktion, um die Anzahl der Leser eines Artikels in Echtzeit anzuzeigen [46]. Mithilfe von Redis-Listen lassen sich auch Tracking-Informationen sehr effizient verwalten. Im Vergleich zu Memcached bietet Redis ein deutlich umfangreicheres Datenmodell und ermöglicht zusätzlich eine replizierte und dauerhafte Speicherung von Schlüssel-Wert-Paaren. Zu den prominentesten Anwendern zählen Twitter, Github, Craigslist, Tumblr und Flickr [150].

[53] http://www.digg.com

Activity Feeds

Als Activity Feeds werden Dienste in sozialen Netzwerken bezeichnet, welche die jüngsten Aktivitäten befreundeter oder abonnierter Personen und Organisationen in Echtzeit anzeigen. Da es sich hierbei um sehr einfache, schlüsselbasierte Informationen handelt, die in sehr hoher Geschwindigkeit verarbeitet werden müssen, bieten sich zu diesem Zweck hauptspeicherbasierte Datenbanksysteme wie Redis an.

Twitter: Twitter [54] ist eine Mikroblogging-Plattform auf der Nutzer 140 Zeichen lange Kurznachrichten in Echtzeit veröffentlichen können. Diese sogenannten Tweets können Hashtags, Links, Bilder, Standorte und Verweise auf andere Nutzer enthalten. Jeder Anwender besitzt einen als Timeline bezeichneten Activity Feed. Auf diesem werden alle Tweets der Nutzer angezeigt, deren Nachrichten der Anwender zuvor abonniert hat. Sobald ein Tweet versendet wird, ist dieser nicht nur auf einer öffentlichen Seite für jedermann einsehbar, sondern er wird auch an die Timelines der Abonnenten („Follower“) geschickt. Twitter muss durchschnittlich über 5500 Tweets pro Sekunde [97] verarbeiten. Vor allem bei Prominenten mit mehreren Millionen Followern entstehen hier extreme Anforderungen an die *Verarbeitungsgeschwindigkeit von Lese- und Schreibanfragen*. Twitter verwendet Redis für die Darstellung der Timelines. Hierfür bietet sich der Datentyp `List` an. Sobald ein Tweet veröffentlicht wird, werden die Tweet-ID, die Nutzer-ID und zusätzliche Metainformationen mit einer einfachen Schreiboperation in die Redis-Listen der Follower eingetragen. Da sich die Timelines aller aktiven Twitter-Nutzer im Hauptspeicher befinden, können diese Operationen sehr schnell verarbeitet werden.

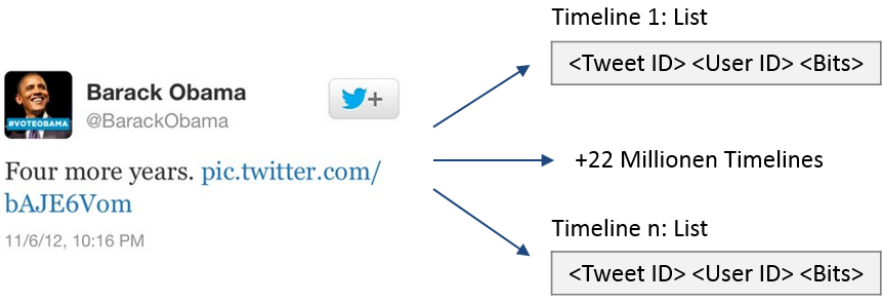


Abbildung 7.1: Tweet von Barack Obama zum Wahlsieg 2012

Abbildung 7.1 beschreibt den Tweet, den Barack Obama unmittelbar nach seinem Wahlsieg 2012 versendet hat. Dieser wurde an die Timelines von über 22 Millionen Followern versendet und über 810.000 Mal weitergeleitet [173]. Eine ähnliche Meldung würde heute, bei über 42 Millionen Obama-Followern [171], eine fast doppelt so

[54] http://www.twitter.com

große Last bei Twitter verursachen. Infolge des begrenzten Hauptspeicherplatzes umfasst eine Timeline maximal 800 Tweets. Die eigentliche Textnachricht wird in einem Memcached-Cluster zwischengespeichert und in MySQL persistiert. Die Beziehungsinformationen zwischen Nutzern und Abonnenten werden in der von Twitter entwickelten Graphdatenbank FlockDB verwaltet [98].

Tumblr: Tumblr[55] ist eine beliebte Plattform, auf der Nutzer Nachrichten, Bilder, Links und Multimediadateien in einem Blog veröffentlichen können. Ähnlich wie bei Twitter wird zwischen dem eigenen Blog und dem sogenannten Dashboard unterschieden. Auf dem Dashboard werden neben aktuellen Beiträgen von abonnierten Blogs auch Benachrichtigungen angezeigt, die erscheinen, sobald ein anderer Nutzer einen eigenen Beitrag mit „gefällt mir" markiert, kommentiert oder weiterleitet. Da diese Benachrichtigungen in Echtzeit angezeigt werden sollen und Tumblr über 7500 solcher Benachrichtigungen pro Sekunde verarbeiten muss [112], stellt dieser Service extrem hohe Anforderungen an die *Lese- und Schreibgeschwindigkeit* des zugrunde liegenden Datenbanksystems.

Abbildung 7.2: Repräsentation von Benachrichtigungen in Tumblr

Seit dem Jahr 2011 setzt Tumblr zur Verarbeitung dieser Benachrichtigungen den Key Value Store Redis ein. Als Schlüssel wird die Benutzer-ID und als Wert die Datenstruktur *Sorted Set* verwendet. Sorted Sets enthalten eindeutige Zeichenketten, die anhand eines Kriteriums sortiert abgelegt werden. Tumblr verwendet hierfür die Zeitstempel der Benachrichtigungen. Hierdurch lassen sich Bereichsanfragen, wie beispielsweise alle Benachrichtigungen des Tages, sehr effizient verarbeiten. In Abbildung 7.2 ist die Verarbeitung einer Follower-Benachrichtigung dargestellt. Für das Dashboard *x* wird entsprechend des Zeitstempels die Nachricht „kevin started following you" hinterlegt.

[55] http://www.tumblr.com

Stapelverarbeitung

Project Voldemort wurde im Jahr 2008 von LinkedIn entwickelt und basiert auf dem Konzept von Amazons proprietärem Key Value Store Dynamo. Project Voldemort unterscheidet sich von Redis und Memcached in der Datenmenge, die verarbeitet werden soll. Dabei steht in erster Linie nicht die performante Verarbeitung von Lese- und Schreibanfragen auf einige wenige Datenobjekte im Fokus, sondern die zuverlässige Bereitstellung sehr großer Datenmengen, die auf aufwendigen Analysen basieren.

LinkedIn: LinkedIn[56] ist ein soziales Netzwerk zur Pflege und zur Neugewinnung von Geschäftskontakten. Mit über 300 Millionen Nutzern [106] ist es das derzeit größte Netzwerk dieser Art weltweit. Wie die meisten sozialen Netzwerke und E-Commerce-Applikationen enthält die Plattform einige Dienste, die spezifisch für jeden Anwender neue Kontakte, bestimmte Produkte oder ausgewählte Nachrichten empfehlen. Diese Dienste basieren auf den Ergebnissen von komplexen Algorithmen, die auf mehreren hundert Terabyte an Daten ausgeführt werden. Aus Performanzgründen werden diese Berechnungen nicht auf dem Live-Datenbanksystem ausgeführt, das für die Transaktionsverwaltung verwendet wird (OLTP), sondern im Rahmen des Online Analytical Processings (OLAP) auf einem Hadoop-Cluster. Für die performante Bereitstellung dieser Ergebnisse verwendet LinkedIn Project Voldemort. Da dieses Datenbanksystem die Daten nicht nur im Hauptspeicher verwaltet, kann es im Gegensatz zu anderen Key Value Stores auch *sehr große Datenmengen* persistent verarbeiten. Beim Einspielen der OLAP-Ergebnisse mithilfe von Stapelverarbeitungen besitzt Project Voldemort gegenüber MySQL deutlich bessere *Performanzeigenschaften* [105, 167].

Onlineshops

Amazon verwendet Dynamo in allen Anwendungsbereichen, in denen neben hoher Performanz vor allem hohe Verfügbarkeit eine zentrale Anforderung ist. Project Voldemort, als quelloffene Implementierung von Dynamo, kann daher in den gleichen Anwendungsgebieten eingesetzt werden.

Gilt: Gilt[57] ist ein Online-Shop, der sich auf den Vertrieb von exklusiven Kleidungsstücken zu Sonderkonditionen spezialisiert hat. In der Regel handelt es sich bei den Produkten um limitierte Restposten bestimmter Marken, die nur innerhalb von 48 Stunden verfügbar sind. Da Kunden vor dem Verkauf nur über den Designer und nicht über die konkreten Artikel informiert werden, ist Gilt zu Beginn einer Verkaufsaktion mit sehr hohen Besucherzahlen konfrontiert, deren Ausmaß sich zuvor nur schwer einschätzen lässt.

[56] http://www.linkedin.com
[57] http://www.gilt.com

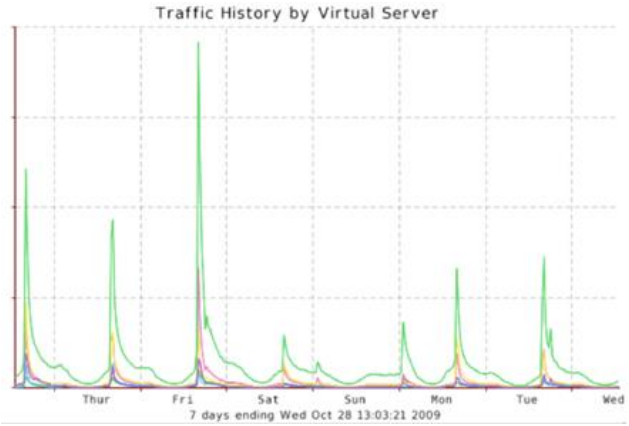


Abbildung 7.3: Verlauf des wöchentlichen Datenverkehrs bei Gilt [110]

In Abbildung 7.3 ist der Verlauf des wöchentlichen Datenverkehrs bei Gilt dargestellt. Jeden Tag um 12:00 Uhr beginnt eine Verkaufsaktion. Zu diesem Zeitpunkt übersteigt der Datenverkehr ein Vielfaches des üblichen Datenaufkommens [110]. Trotz dieser extremen Schwankungen in den Besucherzahlen müssen alle Dienste der Plattform jederzeit wie gewohnt zur Verfügung stehen. Besonders die *permanente Verfügbarkeit* des Warenkorb-Services hat einen massiven Einfluss auf den Erfolg des Geschäftsmodells. Ein kurzfristiger Ausfall einiger Warenkörbe hätte nicht nur den Verlust dieser Einkäufe zur Folge, sondern führt auch zu einem Vertrauensverlust bei den Kunden. Dieser wirkt sich negativ auf zukünftige Besuche dieser Kunden auf Gilt aus. Um diesem finanziellen Schaden vorzubeugen, basiert der Warenkorb-Service bei Gilt auf einem hochverfügbaren Project Voldemort-Cluster. Ein Warenkorb wird durch ein JSON-Objekt repräsentiert und kann über die zugehörige Benutzer-ID eindeutig identifiziert werden. Jeder Warenkorb wird auf drei unterschiedlichen Maschinen gespeichert. Wird ein Artikel zum Warenkorb hinzugefügt, wird die am schnellsten verfügbare Version des Warenkorbs aus der Datenbank ausgelesen, applikationsseitig aktualisiert und anschließend wieder in die Datenbank zurückgeschrieben. Die sich auf den anderen Maschinen befindlichen Versionen des Warenkorbs werden asynchron aktualisiert. Durch den Einsatz unterschiedlicher Clients und das Auftreten von Rechnerausfällen oder Netzwerkfehlern kann es passieren, dass zu einem Zeitpunkt mehrere widersprüchliche Versionen eines Warenkorbs vorliegen. Diese temporären Inkonsistenzen sind während des Einkaufs tolerierbar und werden spätestens bei der abschließenden Bestellung mithilfe von Vektoruhren erkannt und aufgelöst (siehe Kapitel 3.4.4). Unter Umständen ist für die Auflösung der Konflikte die Zuhilfenahme des Kunden erforderlich [45, 110].

Identifizierte Auswahlkriterien

Key Value Stores werden hauptsächlich zur performanten Verarbeitung von Lese- und Schreibanfragen eingesetzt. Die hier genannten Anforderungen gleichen den Auswahlkriterien, die bei der konzeptuellen Aufarbeitung identifiziert wurden:

- Hohe Leistungsfähigkeit bei Leseoperationen (Redis, Project Voldemort),
- Hohe Leistungsfähigkeit bei Schreiboperationen (Redis, Project Voldemort),
- Bereitstellung von komplexen Datenstrukturen (Redis),
- Sehr gute Skalierbarkeitseigenschaften (Project Voldemort),
- Permanente Verfügbarkeit (Project Voldemort).

7.1.2 Document Stores

Document Stores verfügen trotz der hohen Flexibilität des Datenmodells über einen sehr großen Funktionsumfang. Da sie obendrein eine konsistente Datenverarbeitung ermöglichen, können sie auch in Bereichen eingesetzt werden, in denen eine verlässliche Datenverarbeitung erforderlich ist. Document Stores erweisen sich als äußerst effizient im Umgang mit heterogenen Daten, zeigen jedoch bei einfachen Lese- und Schreibanfragen auf homogenen Datensätzen eine geringere Leistungsfähigkeit im Vergleich zu anderen Datenbanksystemen. Document Stores sind demnach für Anwendungsfälle geeignet, in denen ein flexibles, aber zugleich ausdrucksstarkes Datenmodell benötigt wird. Beispiele hierfür sind Produktkataloge, Content Management Systeme, Datenintegration und Anwendungsfälle, in denen häufige Schemaänderungen durchgeführt werden müssen. Da sich diese Systeme noch dazu sehr einfach auf mehrere Rechner verteilen lassen, stellen sie eine elegante Lösung für die Verarbeitung großer Datenmengen dar. Document Stores unterstützen keine Transaktionen.

Produktkataloge

Document Stores eignen sich für Anwendungsfälle, in denen heterogene Datensätze verarbeitet werden sollen. Häufig handelt es sich dabei um Einsatzgebiete, in denen zunächst relationale Datenbanken verwendet wurden, aber das zu strikte Datenmodell ab einem gewissen Zeitpunkt sowohl in der Modellierung als auch in der Ausführung von Anfragen zu Problemen führte.

CustomInk: Der Onlineshop CustomInk[58] steht exemplarisch für ein typisches Abbildungsproblem von heterogenen Artikeln in Produktkatalogen. CustomInk beschränkte sich zu Beginn des Onlinehandels auf den Vertrieb von freigestaltbaren T-Shirts. Diese

[58] http://www.customink.com

homogenen Datensätze ließen sich problemlos in einem relationalen Datenmodell abbilden. Im Laufe der Zeit wurde das Geschäftsmodell jedoch um weitere Artikel erweitert und umfasst inzwischen mehrere hundert verschiedene Produktarten.

StyleName	Categorization1	Categorization2	Extra1	Size1	Size2
BIG Accessories Skull Cap	Head Wear/Tipped Knit Cap	Null	240	One Size	Null
Clear Glass Mug	Drinkware/Mugs	Null	324	Quantity	Null
Jerzees Sleeveless T	Shirts/Sleeveless T-Shirts	Athletic Wear	230	S	M
American Apparel Girly Jeans	Womens Styles/Pants	Null	96	1 (S)	2 (M)
Gildan Ultra Cotton T	Shirts/Short Sleeve T-Shirts	Null	260	YXS	YS

Abbildung 7.4: Heterogener Produktkatalog bei CustomInk [83]

Dieses Wachstum führte zu einem sehr ineffizienten Datenmodell, das in Abbildung 7.4 dargestellt ist. Aufgrund der beschränkten Fähigkeiten relationaler Datenbanken im *Umgang mit heterogenen Datensätzen* (siehe Kapitel 6.1) entschlossen sich die Entwickler bei CustomInk, den Document Store MongoDB zur Abbildung des Produktkatalogs einzusetzen [83].

The National Archives: The National Archives[59] ist das offizielle Archiv des Vereinigten Königreichs. Angefangen mit dem Doomesday Book und der Magna Carta, über detaillierte Dokumente aus den beiden Weltkriegen bis hin zu aktuellen Regierungsberichten enthält das Archiv Dokumente aus über 1000 Jahren Menschheitsgeschichte. Es besitzt nach eigenen Angaben über 11 Millionen Dokumente und ist damit die weltweit größte Sammlung historischer und staatlicher Dokumente. Metainformationen über die hinterlegten Daten werden der Öffentlichkeit über einen Onlinekatalog zur Verfügung gestellt. Mit fortschreitender Digitalisierung der hinterlegten Werke wächst die zu verarbeitende Datenmenge kontinuierlich an. Hinzu kommt die Bereitstellung von Informationen über Werke, die in anderen Archiven hinterlegt sind und in den Onlinekatalog integriert werden müssen. The National Archives verwenden wegen der hohen *Heterogenität der Metadaten* und der *großen Datenmenge* einen MongoDB-Cluster als verteiltes Datenbanksystem. Das ursprünglich über 2000 Tabellen große Datenbankschema konnte hierdurch auf 30 Collections reduziert werden [13, 47].

[59] http://www.nationalarchives.gov.uk

Content Management Systeme

Mithilfe von Content Management Systeme können digitale Informationen erstellt, bearbeitet und veröffentlicht werden. Hierbei werden in der Regel mehrere kooperative Prozesse durchlaufen, an denen Autoren, Verleger, Herausgeber und Administratoren gemeinschaftlich beteiligt sein können. Content Management Systeme sind deswegen in der Lage, Benutzer und Rollen zu verwalten, Verarbeitungsprozesse zu unterstützen, mehrere Versionen von Dokumenten zu verarbeiten und die Veröffentlichung der Inhalte zu ermöglichen. Die Art der Informationen, die über Content Management Systeme zumeist auf Onlinemedien und Blogs veröffentlicht werden, hat sich durch das Web 2.0 in den letzten Jahren deutlich geändert. Neben einfachen Texten werden in zunehmendem Maße Bilder, Videos, benutzergenerierte Inhalte und Informationen aus sozialen Netzwerken in die Veröffentlichungen integriert.

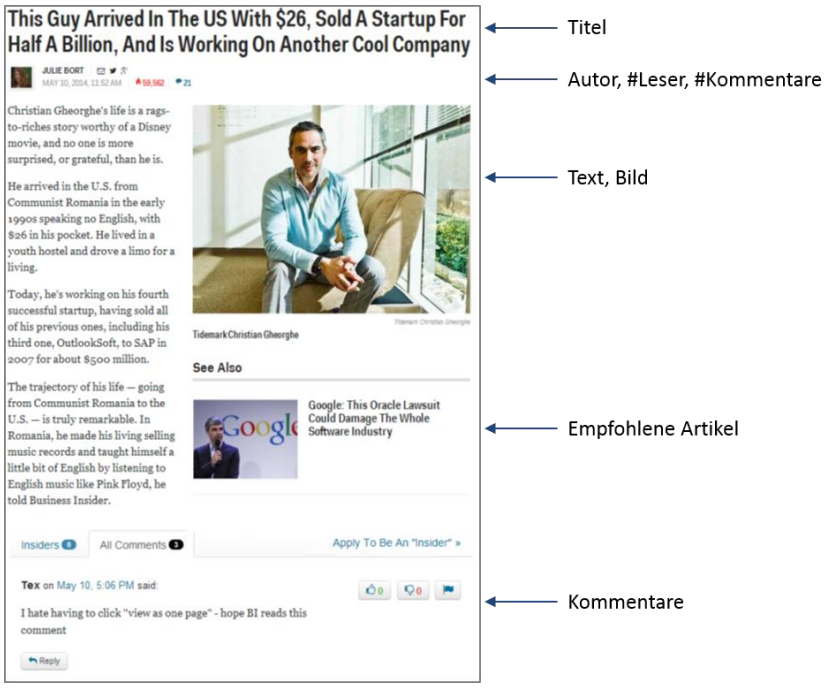


Abbildung 7.5: Ein typischer Artikel auf der Plattform The Business Insider[60].

Ein Artikel besteht üblicherweise aus einem Titel, textuellem Inhalt und Bildern. Daneben werden Autoren, Aufrufstatistiken, empfohlene Artikel und Kommentare angezeigt (Abbildung 7.5). Ein relationales Datenmodell zur Abbildung dieses Sachverhalts ist in Abbildung 7.6 dargestellt.

[60] http://www.businessinsider.com/christian-gheorghe-rags-to-riches-story-2014-5

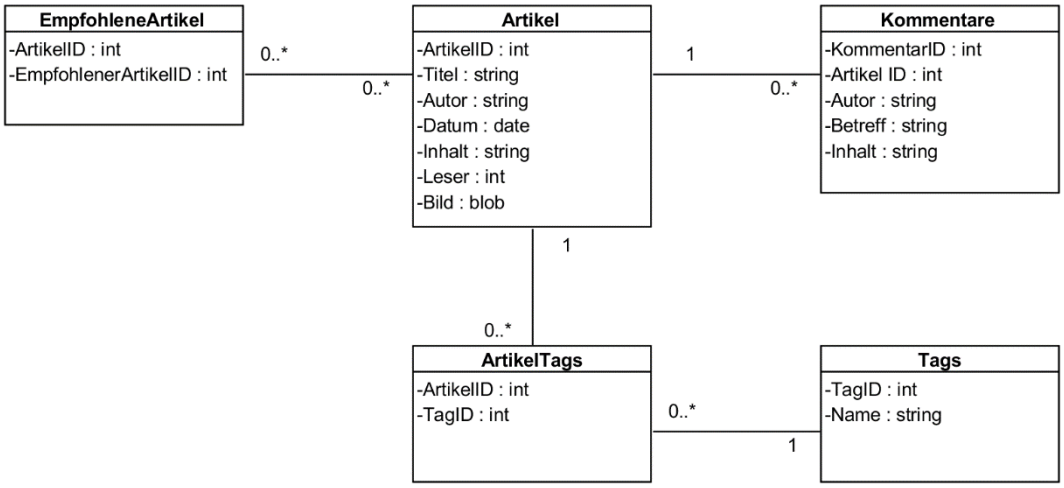


Abbildung 7.6: Relationales Datenmodell eines Artikels

Wird ein Artikel auf der Webseite des Content Management Systems aufgerufen, müssen zur Bereitstellung aller benötigten Informationen bei diesem Datenmodell mehrere Verbundoperationen gleichzeitig ausgeführt werden. Gerade bei gut besuchten Webseiten mit einer hohen Anzahl an Artikeln können hierdurch sehr hohe Ausführungskosten anfallen. Diese hohen Ausführungskosten lassen sich durch Denormalisierung und durch Caching reduzieren. So können beispielsweise der Name und das Bild des Autors redundant in der Relation *Artikel* gespeichert und häufig aufgerufene Artikel mit allen benötigten Informationen zusätzlich in einem Key Value Store hinterlegt werden. Durch diese Maßnahmen steigen allerdings die Komplexität und somit auch die Kosten des Content Management Systems.

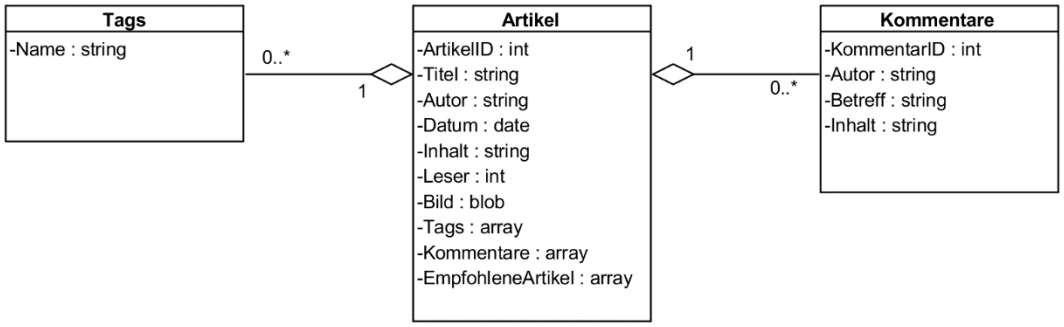


Abbildung 7.7: Datenmodell eines aggregierten Artikels

Document Stores bieten mit der Aggregation eine effiziente und elegante Lösung dieser Problematik. Durch die Abbildung sämtlicher Artikelinformationen in einem einzigen Aggregat beziehungsweise Dokument (Abbildung 7.7), kann die Anzahl der zur Darstellung eines Artikels benötigten Datenbankaufrufe auf genau eine Operation reduziert werden. Dank dieser Modellierungsvariante kann die *Leistungsfähigkeit von*

Lese- und Schreiboperationen auf zusammenhängende Entitäten einer Miniwelt gesteigert werden. Da Document Stores zudem sehr umfangreiche Anfragefunktionalitäten bereitstellen, Replikation automatisch unterstützen und Dokumente sehr einfach auf mehrere Verarbeitungsrechner verteilen, bieten sie sich auch für Content Management Systeme an, die sehr *große Datenmengen* verarbeiten müssen. Aufgrund dieser Vorteile besitzen zunehmend mehr Content Management Systeme aktiv oder über Drittanbieter Schnittstellen zu Document Stores [48, 175, 186]. Bekannte Nutzer von Document Stores in diesen Anwendungsbereichen sind die Onlineausgaben von New York Times, The Guardian, Forbes Magazin, BBC und Business Insider sowie das Multimediaportal von MTV Networks [56, 121].

Datenintegration

Bei der materialisierten Datenintegration werden Informationen aus verschiedenen Datenquellen ausgelesen und in einem zentralen System zusammengeführt. Die Schwierigkeit besteht darin, die in der Regel sehr heterogenen Datenstrukturen verlustfrei und effizient in einem einheitlichen Datenschema abzubilden. Datenintegration wird häufig in Anwendungsfällen benötigt, in denen verschiedene Systeme miteinander verbunden und Daten ausgetauscht werden sollen. Angesichts ihres flexiblen Schemas bieten sich Document Stores für die Verarbeitung verschiedenstrukturierter Datensätze aus unterschiedlichen Quellen an. Außerdem können die immer häufiger im JSON-Format übertragenen Daten direkt in Document Stores verarbeitet werden.

Traackr: Traackr[61] ist eine Influencer-Marketing-Plattform. Beim Influencer-Marketing wird eine werberelevante Zielgruppe nicht über breite Werbemaßnahmen direkt angesprochen, sondern indirekt über ausgewählte Schlüsselfiguren, deren Meinung einen hohen Einfluss auf die gewünschte Zielgruppe hat. Neben Personen mit einem ausgewiesenen Fachwissen, wie Consultants, Ärzten oder Akademikern, sind vor allem Kontakte für das Influencer-Marketing interessant, die ein hohes Ansehen und eine starke Präsenz in sozialen Netzwerken besitzen. Hierzu zählen unter anderem prominente Personen, Journalisten und erfolgreiche Blogger. Unternehmen wie Traackr versuchen anhand von Analysen, Influencer für bestimmte Schlagworte zu finden und ihren Einfluss zu bewerten. Hierzu werden bei Traackr alle möglichen Artikel, Beiträge und Kommentare dieser Personen gespeichert und anschließend ausgewertet. Das Datenmodell bei Traackr besteht im Wesentlichen aus den drei Entitäten *Influencer*, *Site* und *Post*. Jede dieser Entitäten enthält eine *Vielzahl an heterogenen Daten*, die aus sehr vielen verschiedenen und immer wieder neuen Quellen bezogen werden. So werden beispielsweise sowohl 140 Zeichen lange Twitter-Nachrichten als auch mehrere Seiten umfassende Artikel in der Entität *Post* gespeichert. Für jeden Influencer werden neben den personenbezogenen Daten auch Informationen darüber gespeichert, in welcher Häufigkeit er Artikel auf welcher Seite veröffentlicht. Für jede

[61] http://ww.traackr.com

Seite werden spezifische Informationen hinterlegt, die angeben, wie beliebt diese Seite im Netz ist. Indikatoren hierfür sind beispielsweise bei Twitter die Anzahl der Follower oder in Blogs die Anzahl der Verlinkungen auf Google. Für die Abbildung dieser Daten wird MongoDB verwendet. Neben dem flexiblen Schema, das für die Abspeicherung der Beiträge benötigt wird, besitzt MongoDB eine *Anbindung an Hadoop* und ermöglicht damit eine vergleichsweise schnelle Ausführung der eingesetzten Scoring-Algorithmen. Zusätzlich lassen sich in MongoDB sehr leicht *Indizes* definieren, sodass sich auch mehrwertige Beziehungen, wie zwischen Influencer und Seite, sehr schnell und einfach auflösen lassen [159].

Weitere bekannte Beispiele zur Datenintegration mit Document Stores sind die von SunlightLabs[62] entwickelten Applikationen, mit denen Aktivitäten von US-Politikern gegenüber der Bevölkerung transparenter gestaltet werden sollen. Hierzu werden Daten aus verschiedenen Quellen mit *unterschiedlichen Datenstrukturen* aus allen 50 Bundesstaaten in MongoDB gespeichert und mithilfe von Analyse-Werkzeugen Informationen, wie beispielsweise das Abstimmungsverhalten bei bestimmten Gesetzesbeschlüssen, der Bevölkerung zur Verfügung gestellt [124, 125].

Schemaevolution

In Anbetracht ihres flexiblen Schemas sind Document Stores für Anwendungsfälle geeignet, in denen sich das Datenmodell häufig ändert oder dieses zu Beginn der Softwareentwicklung noch nicht vollständig feststeht.

Agile Softwareentwicklung: In der agilen Softwareentwicklung wird versucht, durch einen Abbau an Bürokratie den Softwareentwicklungsprozess deutlich flexibler und schlanker zu gestalten als es bei den klassischen Vorgehensmodellen, wie dem Rational Unified Process (RUP) oder dem V-Modell, der Fall ist. Ziel ist es, die Entwurfsphase auf ein Minimum zu reduzieren und in der folgenden Entwicklungsphase möglichst frühzeitig ausführbare Software zu erzeugen, die dem Kunden zur Abstimmung vorgelegt werden kann. In mehreren, in der Regel sehr kurzen Iterationsschritten, werden diese Prototypen gemeinsam mit dem Kunden verfeinert. Durch die Einbindung des Kunden in den Entwicklungsprozess soll die Gefahr von Fehlentwicklungen reduziert werden. Allerdings muss das Entwicklerteam dann auch in der Lage sein, flexibel auf Kundenwünsche eingehen zu können. Häufige Änderungen sind deshalb typisch für die agile Softwareentwicklung. Ist von diesen Änderungen auch das Datenmodell betroffen, kann dies im Fall von relationalen Datenbanken zu komplexen und ab einer gewissen Datenmenge auch teuren Anpassungen auf Datenbankebene führen. Wird dagegen in einem solchen Projekt ein Document Store eingesetzt, können *Schemaänderungen* problemlos zur Laufzeit auf Applikationsebene durchgeführt werden. Ein weiterer Vorteil von Document Stores ist die im Vergleich zu relationalen Datenbanken sehr viel einfachere Abbildung von objektorientierten Entitäten. JSON-

[62] http://www.sunlightlabs.com

Dokumente unterstützen neben den primitiven Datentypen auch Arrays, Listen und eingebettete Dokumente und sind deswegen in der Lage, *komplexe Datentypen* direkt abzubilden. Beispielsweise lassen sich Objekte des in der agilen Entwicklung von Webapplikationen sehr beliebten Frameworks Ruby on Rails[63] durch einen integrierten Mapper nahezu ohne zusätzlichen Aufwand in MongoDB verwalten [122]. Darüber hinaus können Document Stores dank integrierter Replikations- und Verteilungstechniken ohne weitere Implementierungen auf Applikationsebene und mit einem minimalen administrativen Aufwand horizontal skalieren. Diese Eigenschaften machen MongoDB insbesondere für Startup-Unternehmen sehr attraktiv, die wegen ihres schnellen Wachstums oder ihres Geschäftsmodells sehr große Datenmengen verarbeiten müssen, gleichzeitig jedoch hinsichtlich ihres in der Regel sehr limitierten Budgets keine teuren oder komplexen Datenbanksysteme einsetzen können.

Craigslist: Schemaänderungen betreffen nicht nur die operativen Datenbanksysteme, sondern wirken sich auch auf die Archivierung aus. Craigslist[64] ist eine vor allem im englischsprachigen Raum sehr beliebte Internetplattform, auf der Kleinanzeigen jeglicher Art online gestellt werden können. Seit dem Jahr 1999 verzeichnet die Webseite ein sehr starkes Wachstum und ist heute in über 700 Städten in mehr als 50 Ländern weltweit verfügbar. Craigslist archiviert von Beginn an jede jemals geschaltete Kleinanzeige und stellt diese ihren Nutzern in einem Archiv zur Verfügung. Inzwischen müssen täglich über 1,75 Millionen Kleinanzeigen archiviert werden. Das Datenbankschema des Onlinedienstes hat sich im Laufe der Jahre mehrfach geändert. Aufgrund der hier beschriebenen Systemarchitektur muss jede dieser Änderungen nicht nur in der Live-Datenbank, sondern auch im Archiv, das jahrelang auf einem relationalen Datenbanksystem basierte, durchgeführt werden. Wegen der enormen Datenmenge, die im Archiv hinterlegt ist, nahmen *Änderungen am Datenbankschema* nicht selten mehrere Monate in Anspruch. Da bei jeder Schemaänderung das Archiv nicht zur Verfügung stand, konnten in diesen Zeiträumen keine Anzeigen aus dem operativen Datenbanksystem in das Archiv ausgelagert werden. Hierdurch kam es zu deutlichen Verzögerungen im Frontend. Craigslist verwendet aus diesen Gründen seit dem Jahr 2011 einen MongoDB-Cluster als Datenbanksystem für das Archiv. Neben dem flexiblen Schema wurden die *integrierte Replikation* und die *automatische Fehlerbehandlung* als Gründe für den Umzug genannt [119, 187, 188].

[63] http://www.rubyonrails.org
[64] http://www.craigslist.org

Identifizierte Auswahlkriterien

Document Stores sind ähnlich wie relationale Datenbanken in der Lage, in einem breiten Spektrum an Anwendungsfällen eingesetzt zu werden. Die im Folgenden aufgeführten Entscheidungskriterien sind typisch für einen Einsatz dieser Systeme:

- Abbildung größerer Miniwelten mit Aggregationen,
- Effiziente Verarbeitung von heterogenen Datensätzen,
- Einfache Schemaänderungen,
- Unterstützung von komplexen Datentypen,
- Geringer Administrationsaufwand bei verteilter Datenhaltung,
- Anbindung an MapReduce-Frameworks.

Bei der konzeptuellen Aufarbeitung kristallisierte sich neben den hier genannten Kriterien eine Reihe weiterer Eigenschaften von Document Stores heraus, die bei der Wahl eines Datenbanksystems von entscheidender Bedeutung sein können:

- Große Anfragemächtigkeit,
- Synchronisation von konkurrierenden Zugriffen,
- Persistente Speicherung.

7.1.3 Column Family Stores

Column Family Stores weisen die geringste Flexibilität unter den NoSQL-Systemen auf. Abgesehen davon, dass Column Families, bevor sie genutzt werden können, in einem globalen Schema definiert werden müssen, sind Anfragen auf Datensätze ausschließlich über den Primärschlüssel möglich. Innerhalb der Column Families können beliebig strukturierte Datensätze verwaltet werden, weshalb Column Family Stores für die Verarbeitung von homogenen Datensätzen geeignet sind. Einer der beiden Hauptgründe für den Einsatz eines Column Family Stores ist die hohe Performanz, mit der Schreibanfragen persistent verarbeitet werden können. Der andere große Vorteil von Column Family Stores ist deren Skalierbarkeit. Aus diesen Gründen bieten sich Column Family Stores für Anwendungsfälle an, in denen sehr große Mengen an heterogenen Daten mit einer hohen Schreibgeschwindigkeit verarbeitet werden müssen. Nicht selten handelt es sich hierbei um Anwendungsfälle, wie beispielsweise Logging und Echtzeitanalysen, in denen Daten mehrheitlich geschrieben und nur vergleichsweise selten gelesen werden. Column Family Stores sind angesichts der beschränkten Anfragemächtigkeit und des erhöhten administrativen Aufwands nicht für alltägliche Speicherproblematiken der Informatik geeignet. Wie alle NoSQL-Datenbanken unterstützen sie keine Transaktionen.

Logging

Im Rahmen des Loggings werden Ereignisse kontinuierlich protokolliert und typischerweise in Textdateien abgelegt. So speichern beispielsweise Webserver Informationen über Anfragen, Benutzerverhalten, Warnungen und Fehlermeldungen. Um diese in der Regel *sehr großen Datenmengen* effizient analysieren zu können, sollten Ereignisse nicht in Textdateien, sondern als eigenständige Datensätze in einer Datenbank gespeichert werden. Logging-Informationen werden kontinuierlich in einer *sehr hohen Geschwindigkeit geschrieben*, niemals aktualisiert und verhältnismäßig selten gelesen. In Anbetracht dieser Eigenschaften bieten sich Column Family Stores zur Verarbeitung von Logging-Informationen an. Ein weiterer Vorteil von Column Family Stores in diesem Anwendungsgebiet ist deren Möglichkeit, Datensätze mit einem Verfallsdatum zu versehen (Time-To-Live – TTL). Nach Ablauf dieses Datums werden die Datensätze automatisch aus dem System entfernt. Die Analyse der sehr großen Datenmengen erfolgt mithilfe von *MapReduce-Frameworks*, wie beispielsweise Hadoop.

Ebay: Angefangen von der Bestellung, über die Bezahlung und den Versand bis hin zu Rückgaben werden beim Onlineauktionsportal Ebay sämtliche Ereignisse protokolliert, die während und nach einem Geschäft von Benutzern durchgeführt werden. Diese Informationen werden unter anderem dazu verwendet, um Betrugsfälle aufzudecken beziehungsweise zu vermeiden, Vermittlergebühren qualitätsgerecht abzurechnen (Quality Click Pricing) und um Analysen über Bestell- und Versandprozesse bereitstellen zu können.

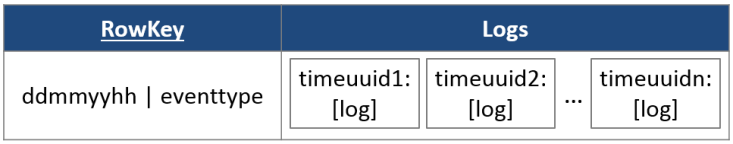


Abbildung 7.8: Datenmodell eines Event-Logging-Systems bei Ebay

Aufgrund der hohen Schreibgeschwindigkeit und der sehr großen Datensätze verwendet Ebay den Column Family Store Cassandra zur Verwaltung der Ereignisse. Für jeden Ereignistyp und für jede Stunde wird ein neuer Datensatz angelegt. Dieser enthält mehrere Schlüssel-Wert-Paare. Als Schlüssel wird der Zeitstempel des Ereignisses herangezogen, als Wert die Logging-Information (Abbildung 7.8). Diese Daten werden dann mit MapReduce-Funktionen analysiert und die Ergebnisse verschiedenen Services bereitgestellt [139, 141].

Google: Google zeichnet das Benutzerverhalten bei der Verwendung von Google-Produkten auf und speichert diese Daten in Bigtable. Unter Zuhilfenahme von MapReduce-Funktionen werden anschließend Daten für Benutzerprofile, Statistiken und andere Dienste extrahiert.

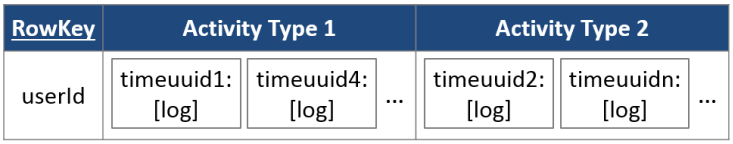


Abbildung 7.9: Datenmodell zur Speicherung von Benutzerinteraktionen bei Google

Die Speicherung dieser Daten erfolgt mithilfe des in Abbildung 7.9 dargestellten Datenmodells. Für jeden Nutzer wird hierzu in Bigtable genau ein Datensatz angelegt. Für jeden Interaktionstyp, wie beispielsweise Suchanfragen, existiert eine Column Family. Ähnlich wie bei Ebay werden Ereignisse in Form von Schlüssel-Wert-Paaren gespeichert, die aus dem Zeitstempel und dem Ereigniswert bestehen [30]. Führt ein Benutzer beispielsweise eine Suchanfrage aus, wird ein Schlüssel-Wert-Paar bestehend aus Zeitstempel und Suchtext in der entsprechenden Column Family in den Benutzerdatensatz eingefügt. Da Google für jeden Nutzer einen Datensatz anlegt und innerhalb eines Datensatzes alle Interaktionen des jeweiligen Nutzers speichert, entstehen hier *sehr große Tabellen* mit einer Vielzahl an Datensätzen, die jeweils eine sehr hohe Anzahl verschiedener Schlüssel-Wert-Paare beinhalten. Bigtable wird auch von Google Analytics verwendet. Dieser Service stellt Webseitenbetreibern Statistiken über Besucher- und Einkaufszahlen bereit. Hierzu müssen kleine JavaScript-Programme in die Webseiten eingebunden werden, die beim Aufruf der Seite Daten an Google senden. Für jede Session wird dort ein neuer Datensatz in Bigtable erzeugt und darin die Benutzerinteraktionen während der Session protokolliert. Durch den Einsatz von MapReduce-Funktionen werden dann die angebotenen Statistiken generiert.

Echtzeitanalysen

Statistiken, die mit MapReduce-Programmen erzeugt werden, sind in der Regel zeitintensiv und demzufolge nicht für Echtzeitanalysen geeignet. Aus diesem Grund basieren Echtzeitanalysen auf Daten, die bereits beim Eintreffen eines Ereignisses aggregiert werden. Hauptspeicherbasierte Datenbanksysteme wie Key Value Stores können wegen ihres begrenzten Speicherplatzes nicht in Anwendungsgebieten, wie beispielsweise Google Analytics, eingesetzt werden, weil die dort generierten Datenmengen zu groß sind. Des Weiteren handelt es sich bei diesen Informationen um geschäftskritische Daten, die *hohe Persistenzanforderungen* besitzen. In solchen Anwendungsfällen bieten sich Column Family Stores zur Realisierung von Echtzeitanalysen an.

Facebook Insights: Ähnlich wie Google Analytics stellt Facebook Insights[65] Webseitenbetreibern umfangreiche Informationen über ihre Besucher und die Art und Weise, wie diese mit der Webseite interagieren, bereit. Als Datenquellen dienen neben der Facebook-Plattform und Diensten wie Facebook Ads auch die sogenannten Social

[65] http://www.facebook.com/insights

Plugins (Like Buttons, Kommentar-Boxen), die auf den Webseiten eingebunden werden können. Diese Plugins senden Benutzerinformationen an Facebook und ermöglichen dadurch die Erstellung von detaillierten Statistiken, die unter anderem darüber informieren, welcher Nutzer zu welchem Zeitpunkt und in welcher Art und Weise die Webseite besucht hat (Abbildung 7.10).

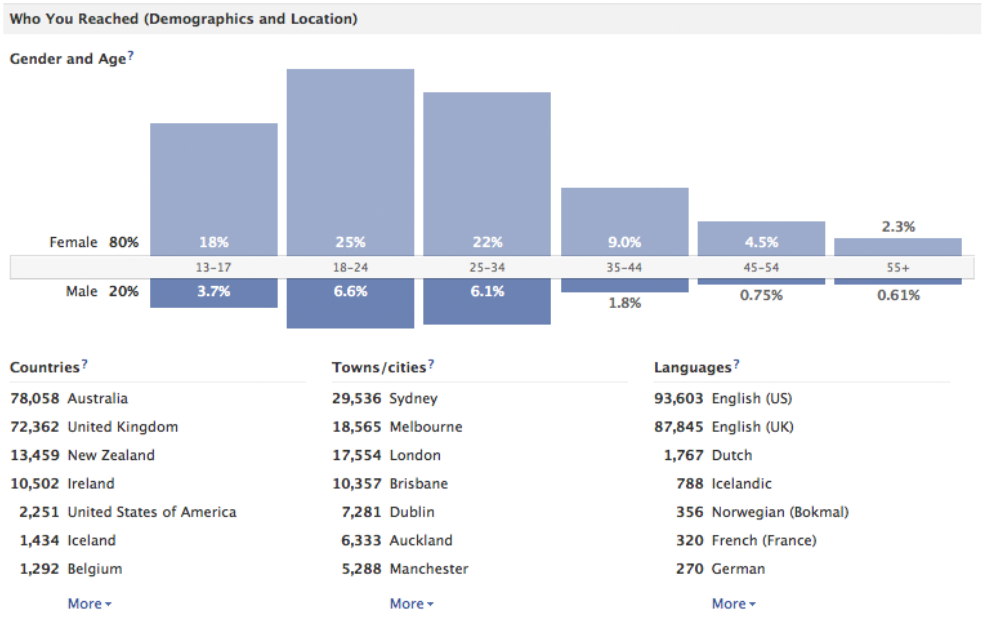


Abbildung 7.10: Von Facebook Insights bereitgestellte Nutzerstatistiken[66]

Bei Facebook Insights werden Statistiken den Webseitenbetreibern in Echtzeit, das heißt innerhalb von 30 Sekunden, bereitgestellt. Hierzu muss dieser Dienst in der Lage sein, über 200.000 Ereignisse pro Sekunde verarbeiten zu können. Da bei jedem Ereignis gleich mehrere Zähler aktualisiert werden müssen, besitzt Facebook Insights *sehr hohe Anforderungen an die Schreibgeschwindigkeit* der zugrunde liegenden Datenbank. Facebook Insights basiert aus diesem Grund auf einem HBase-Cluster.

Das Datenmodell von Facebook Insights ähnelt dem von Google Analytics (Abbildung 7.9). Für jede Webseite wird ein Datensatz verwaltet. Dieser besteht aus mehreren Zählern, die wiederum über Column Families gruppiert werden. Besucht beispielsweise ein männlicher Nutzer eine bestimmte Webseite um 18:00 Uhr, werden in dem in Abbildung 7.11 dargestellten Beispiel mindestens drei Zähler pro Column Family aktualisiert [19, 68, 86].

[66] Quelle: http://www.whiteecho.com/wp-content/uploads/demographics_large.png

RowKey	Hourly Counters				Daily Counters				Lifetime Counters			
url	6pm Total: 100	6pm Male: 50	6pm US: 92	...	1/1 Total: 1000	1/1 Male: 320	1/1 US: 670	...	Total: 9000	Male: 6780	US: 5800	...

Abbildung 7.11: Datenmodell von Facebook Insights [68]

Große Datenmengen

Neben der hohen Performanz bei Schreiboperationen ist vor allem die Speicherung von sehr großen Datenmengen der große Vorteil von Column Family Stores. Diese Datenbanken sind von Beginn an als verteilte Systeme konstruiert und in der Lage, auch über mehrere zehntausend Rechner hinweg zu skalieren.

Google: Die Google Suche und viele weitere Dienste basieren auf den Daten, die der Google Webcrawler generiert. Um die *extrem großen und stetig weiter wachsenden Daten* effizient verarbeiten zu können, hat Google den Column Family Store Bigtable entworfen.

RowKey	Contents	Anchors		
com.cnn.www	content: <html>...<html>	cnnsi.com: CNN	my.look.ca: CNN.com	...

Abbildung 7.12: Datenmodell von Googles Webcrawler [30]

Bei diesem System werden Webseiten, wie beispielsweise *www.cnn.com*, als ein Datensatz in einer großen Tabelle gespeichert (siehe Abbildung 7.12). Jede Webseite wird über ihre invers notierte URL eindeutig identifiziert. Da alle Datensätze in Bigtable in alphabetischer Reihenfolge ihres Schlüssels abgelegt werden, liegen, bedingt durch diese inverse Notation, alle Webseiten einer Domäne physikalisch nahe beieinander. Analysen auf diesen Daten werden hierdurch deutlich effizienter. Die Tabelle besteht aus den beiden Column Families *Contents* und *Anchor*. Die Column Family *Contents* enthält den vollständigen HTML-Inhalt einer Webseite. Die Column Family *Anchor* umfasst dagegen die Namen der Seiten und die zugehörigen Link-Texte, die auf die Webseite verweisen. In dem in Abbildung 7.12 dargestellten Beispiel verweisen die Webseiten *cnnsi.com* und *my.look.ca* auf die Webseite *www.cnn.com* [30].

Neben Google Analytics und der hier beschriebenen Speicherung der Daten, die der Webcrawler generiert, wird Bigtable für eine Reihe weiterer Dienste bei Google eingesetzt. So wird beispielsweise das gesamte Bildmaterial von Google Earth in Bigtable verwaltet. Jeder Datensatz entspricht hier einem geografischen Segment und enthält die zugehörigen Bilder. Über eine zusätzliche Index-Tabelle werden Anfragen auf dieses Kartenmaterial beantwortet [30].

Facebook Messages: Im Jahr 2011 führte Facebook eine neue Version seines Nachrichtendienstes *Messages* ein. Statt Anwendern wie bisher alle Nachrichten einzeln und in der Reihenfolge ihres Eintreffens in einem Posteingang anzuzeigen, werden die Nachrichten seitdem zu Konversationen zusammengefasst. Eine Konversation kann zwischen zwei oder mehreren Anwendern bestehen. Facebook unterscheidet dabei nicht zwischen den unterschiedlichen Arten der elektronischen Textkommunikation. Anwender können sowohl per Email als auch über Facebook-, Chat- und SMS-Nachrichten an einer Konversation teilnehmen. Bereits im Jahr 2010 wurden über 135 Milliarden Nachrichten pro Monat versendet [126]. Angesichts des starken Wachstums von Facebook dürfte diese Zahl inzwischen um ein Vielfaches gestiegen sein. Typisch für Konversationen ist der mit 45 Prozent relativ hohe Anteil an Schreiboperationen, da Anwender eine empfangene Nachricht in der Regel nur einmal lesen. Darüber hinaus wächst die Datenmenge kontinuierlich an, weil Facebook Konversationen und die zugehörigen Nachrichten nicht löscht. Wegen dieser *sehr hohen Schreiblast* und der *sehr großen Datenmenge* basiert Facebook Messages auf HBase. Die Entscheidung für HBase und gegen den von Facebook selbst entwickelten Column Family Store Cassandra fiel aufgrund der besseren *Konsistenz- und Skalierbarkeitseigenschaften* von HBase [19, 126, 156].

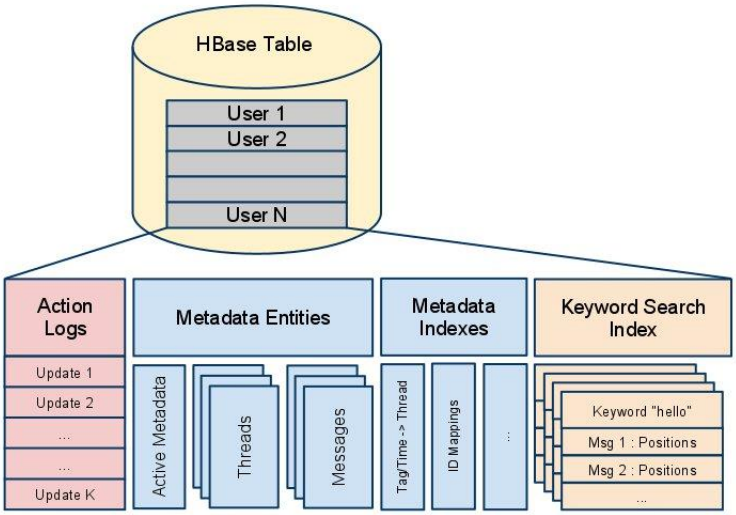


Abbildung 7.13: Datenmodell von Facebook Messages [107]

Die für Facebook Messages benötigten Daten werden in einer Tabelle mit insgesamt acht Column Families verwaltet, die sich wiederum in die drei Gruppen *Action Log*, *Metadata Entities and Indizies* und *Keyword Search Index* unterteilen lassen (Abbildung 7.13). Für jeden Benutzer – und demzufolge auch für jedes Postfach – existiert genau ein Datensatz, der über die Benutzer-ID eindeutig identifiziert wird [82, 107]. In der Column Family *Action Log* werden die eigentlichen Nachrichten gespeichert.

Überschreiten diese oder die Anhänge eine gewisse Größe, werden sie in den persistenten Key Value Store Haystack [15] ausgelagert. Neben den Nachrichten werden in dieser Column Family sämtliche Benutzeraktionen verwaltet, die sich auf die Metadaten der Nachrichten auswirken. Die Metadaten selbst, die beispielsweise angeben, ob eine Nachricht gelesen wurde, zu welchem Thread (Konversation) die Nachricht gehört und welche Nutzer an diesem Thread beteiligt sind, werden in verschiedenen Metadaten-Column Families gespeichert. Hier werden auch Sicherheitseinstellungen, Ordnerstrukturen, Kontakte und Statistiken verwaltet. Die für den effizienten Zugriff auf diese Daten benötigten Indizes werden durch den Einsatz zusätzlicher Column Families realisiert, die ebenfalls zu dieser Gruppierung gehören. Die Column Family *Keyword Search Index* ermöglicht eine Suche nach Nachrichten, die ein bestimmtes Schlüsselwort enthalten. Hierzu werden eingehende Nachrichten mithilfe der Volltextsuchmaschine *Lucene*[67] untersucht und für jedes Wort ein Schlüssel-Wert-Paar erzeugt. Diese Schlüssel-Wert-Paare enthalten das Wort als Schlüssel und als Wert eine Map. Diese beinhaltet sowohl die entsprechenden Nachrichten-IDs als auch die Positionsangaben des Wortes innerhalb der jeweiligen Nachrichten.

Identifizierte Auswahlkriterien

Bei der Analyse typischer Einsatzgebiete von Column Family Stores fällt auf, dass diese Systeme vorwiegend in Anwendungsbereichen mit teilweise extremen Anforderungen an die Datenhaltung eingesetzt werden. Diese Anforderungen umfassen:

- Sehr gute Skalierbarkeitseigenschaften,
- Hohe Leistungsfähigkeit bei Schreiboperationen,
- Persistente Speicherung,
- Anbindung an MapReduce-Frameworks.

Neben diesen vier Auswahlkriterien konnten im Rahmen der konzeptuellen Aufarbeitung weitere Anforderungen identifiziert werden, die zum Einsatz eines Column Family Stores führen können:

- Einfache Administration (Cassandra),
- Permanente Verfügbarkeit (Cassandra),
- Konsistenz in verteilten Systemen (HBase).

[67] http://lucene.apache.org

7.2 Klassifizierung von Auswahlkriterien

Während der konzeptuellen Aufarbeitung und bei der Analyse der praktischen Anwendungsfälle konnten verschiedene Auswahlkriterien identifiziert werden, welche die Wahl eines Datenbanksystems maßgeblich beeinflussen. Diese Kriterien betreffen verschiedene Bereiche eines Datenbanksystems und können entsprechend dieser Bereiche klassifiziert werden. Im Folgenden wird zwischen Kriterien der konzeptuellen Ebene, der internen Ebene, der verteilten Datenhaltung und sonstigen Kriterien unterschieden.

7.2.1 Konzeptuelle Ebene

Kriterien, welche die konzeptuelle Ebene eines Datenbanksystems betreffen, sind die Ausdrucksstärke und die Flexibilität des Datenmodells sowie die Mächtigkeit der bereitgestellten Anfrageschnittstelle.

Ausdrucksstärke des Datenmodells

Der größte Unterschied zwischen den verschiedenen Datenbanksystemen ist der Grad an Ausdrucksstärke, der von den unterschiedlichen Datenmodellen ermöglicht wird. Je mehr Modellierungsmöglichkeiten von einem Datenmodell bereitgestellt werden, desto umfangreicher und detaillierter kann eine Miniwelt auf das jeweilige Datenmodell abgebildet werden. Wie sich bei der konzeptuellen Aufarbeitung herausstellte, unterscheiden sich die einzelnen Datenmodelle vor allem in ihren Fähigkeiten, Attributnamen sowie einfache und komplexe Datentypen abbilden und interpretieren zu können. Manche Datenmodelle sind zudem in der Lage, mehrere Entitäten ineinander einzubetten und können dadurch Beziehungen sehr effizient abbilden.

Flexibilität des Datenmodells

Je größer die Flexibilität eines Datenmodells ist, desto leichter lassen sich Datensätze mit einer heterogenen Struktur darin abbilden. Der Umgang der verschiedenen Datenbanksysteme mit heterogenen Datensätzen wurde in Kapitel 6.1 ausführlich behandelt.

Mächtigkeit der Anfrageschnittstelle

Die verschiedenen Datenbanksysteme unterscheiden sich sehr stark im Umfang der bereitgestellten Anfragefunktionalitäten. Je nach eingesetztem Datenbanksystem stehen Anfragesprachen, sekundäre Zugriffspfade und unterschiedlich viele Operationen zur Verfügung. Die Bereitstellung von Verbundoperationen muss an dieser Stelle hervorgehoben werden, da diese die Anfragemächtigkeit eines Datenbanksystems besonders stark beeinflussen.

7.2.2 Interne Ebene

Die Forderungen nach ACID-Eigenschaften sowie nach einer hohen Leistungsfähigkeit von Lese- und Schreiboperationen betreffen die interne Ebene eines Datenbanksystems. Da eine vollständige Unterstützung der ACID-Eigenschaften in Konflikt mit der Forderung nach einer maximalen Leistungsfähigkeit steht, muss ein anwendungsfallspezifischer Kompromiss zwischen diesen beiden Anforderungen gefunden werden.

ACID-Eigenschaften

Viele NoSQL-Datenbanken verzichten auf eine vollständige Unterstützung der ACID-Eigenschaften, um bei Lese- und Schreibanfragen eine höhere Performanz als relationale Datenbanken zu erzielen. Durch diesen Verzicht kann allerdings der Aufwand auf Applikationsebene erheblich ansteigen, sollten atomare Transaktionen, Konsistenzprüfungen, Synchronisation von konkurrierenden Zugriffen und eine dauerhafte Speicherung von Änderungsoperationen benötigt werden. Eine Analyse der jeweils bereitgestellten ACID-Eigenschaften erfolgte im Rahmen der konzeptuellen Aufarbeitung.

Leistungsfähigkeit von Lese- und Schreiboperationen

Wie in den Benchmarks in Kapitel 6.2 deutlich wurde, weisen die einzelnen Datenbanksysteme zum Teil erhebliche Unterschiede in ihrer Leistungsfähigkeit auf. Manche Systeme sind auf die performante Verarbeitung von Leseoperationen ausgelegt, während andere Systeme wiederum für eine möglichst hohe Leistungsfähigkeit bei Schreiboperationen konzipiert wurden.

7.2.3 Verteilte Datenhaltung

Kriterien wie Administrationsaufwand, Skalierbarkeit von Lese- und Schreiboperationen, permanente Verfügbarkeit und Konsistenz von Replikaten betreffen die verteilte Datenhaltung von Datenbanksystemen.

Administrationsaufwand

Die meisten NoSQL-Datenbanken sind von Beginn an auf eine verteilte Datenhaltung ausgelegt und beinhalten daher Funktionalitäten, mit denen sich unter anderem Fragmentierung, Allokation, Anfrageverarbeitung und Fehlerbehandlung mit einem geringeren Aufwand realisieren lassen.

Skalierbarkeit

Eine häufig in Big-Data-Anwendungsfällen gestellte Anforderung an Datenbanksysteme ist die Skalierbarkeit von Lese- und Schreiboperationen. Das Skalierbarkeitsverhalten der unterschiedlichen Datenbanksysteme wurde im Rahmen verschiedener Benchmarks in Kapitel 6.3 untersucht.

Verfügbarkeit

Die Verfügbarkeit eines verteilten Datenbanksystems hängt von der eingesetzten Verteilungsstrategie ab. In der Regel besitzen dezentral verteilte Systeme eine höhere Verfügbarkeit als zentral verteilte Systeme.

Konsistenz von Replikaten

Die Konsistenzsicherung von Daten, die auf mehrere Server repliziert gespeichert wurden, steht sowohl mit der Leistungsfähigkeit als auch mit der Verfügbarkeit von verteilten Datenbanksystemen in Konflikt. Die hier vorgestellten Datenbanksysteme unterscheiden sich in diesem Kriterium voneinander.

7.2.4 Sonstige Kriterien

Die Wahl eines Datenbanksystems wird zusätzlich von einer Reihe weiterer Kriterien bestimmt, die sich allerdings nicht quantifizieren, sondern nur abschätzen lassen. Zu diesen „weichen" Kriterien zählen Entwicklungsreife, Werkzeugunterstützung, Standardisierung und Fachwissen.

Entwicklungsreife

Relationale Datenbanken weisen dank ihres höheren Alters und der größeren praktischen Erfahrung eine höhere Entwicklungsreife und folglich eine geringere Fehleranfälligkeit als NoSQL-Datenbanken auf. Als Indizien für die Entwicklungsreife können das Alter eines Datenbanksystems sowie die Anzahl und die Geschwindigkeit herangezogen werden, in der neue Versionen des Datenbanksystems veröffentlicht werden.

Werkzeugunterstützung

Viele Systeme weichen im Umfang und in der Qualität der bereitgestellten Werkzeuge zum Teil sehr deutlich voneinander ab. Hierdurch kann sich vor allem der Aufwand, der für die Formulierung von Anfragen und für die Administration des Datenbanksystems benötigt wird, zwischen den verschiedenen Datenbankprodukten sehr stark unterscheiden.

Standardisierung

Ein Kritikpunkt an NoSQL-Datenbanken ist deren fehlende Standardisierung, welche vor allem den Austausch eines Datenbanksystems deutlich erschweren kann.

Fachwissen

Die Menge an verfügbarem Fachwissen kann ebenfalls ein entscheidendes Auswahlkriterium darstellen, das für oder gegen den Einsatz eines bestimmten Datenbanksystems sprechen kann. Auch wenn sich die Qualität des Fachwissens nicht quantifizieren

lässt, kann der Umfang an Dokumentationen, Fachliteratur und Anwendungsbeispielen sowie die Popularität in der Entwicklergemeinde[68] als Indiz für diese Größe herangezogen werden.

7.3 Erstellung eines Kriterienkatalogs

Das Ziel dieses Kapitels ist die Entwicklung eines Kriterienkatalogs, mit dessen Hilfe auch unerfahrene Anwender das Datenbanksystem auswählen können, das am besten zu den Anforderungen eines bestimmten Anwendungsfalls passt. Die Grundlage für diesen Kriterienkatalog bilden die aus der konzeptuellen Aufarbeitung gewonnenen Erkenntnisse, die in den Benchmarks erzielten Ergebnisse und Einschätzungen, die auf praktischen Erfahrungen im Umgang mit diesen Systemen beruhen. Bei der Betrachtung der entscheidungsrelevanten Kriterien fällt auf, dass die Wahl eines Datenbanksystems weniger durch quantitative als vielmehr durch qualitative Systemeigenschaften beeinflusst wird, die sich nicht in Zahlen darstellen lassen. Um dennoch eine systematische und für einen Anwender nachvollziehbare Bewertung der unterschiedlichen Datenbanksysteme hinsichtlich dieser Kriterien ermöglichen zu können, bietet sich der Einsatz von Nutzwertanalysen [51, 87] an.

Nutzwertanalysen sind Bewertungsverfahren der Entscheidungstheorie, die herangezogen werden können, wenn nicht ausschließlich quantitative Einflussgrößen in die Entscheidungsfindung zwischen verschiedenen Optionen einbezogen werden sollen. Obwohl sich die verschiedenen Definitionen in der Anzahl der Entwicklungsschritte beim Aufstellen einer Nutzwertanalyse voneinander unterscheiden, können sie auf die folgenden vier Prozessschritte reduziert werden:

- Identifikation der Kriterien,
- Bewertung,
- Gewichtung,
- Nutzwertberechnung.

Entsprechend dieser vier Entwicklungsschritte soll nun der Kriterienkatalog erstellt werden, der zur Auswahl eines Datenbanksystems benötigt wird.

[68] http://db-engines.com/de/ranking

7.3.1 Identifikation der Kriterien

In einem ersten Schritt müssen die für die Entscheidungsfindung relevanten Kriterien identifiziert werden. Da eine überschaubare Anzahl an Entscheidungsalternativen eine Voraussetzung für eine sinnvolle und ökonomische Anwendung der Nutzwertanalyse darstellt, sollten gleichartige Kriterien zu Gruppen zusammengefasst werden. Diese Gruppen können anschließend erneut unterteilt werden, sodass Baumstrukturen entstehen, die mehrere Ebenen umfassen können. Neben der besseren Übersichtlichkeit und dem reduzierten Aufwand, soll hiermit vor allem vermieden werden, dass die Entscheidung durch die reine Anzahl der Kriterien mit gleicher Intention beeinflusst wird [87, S.282]. Die für die Auswahl eines Datenbanksystems entscheidungsrelevanten Kriterien wurden in den beiden vorherigen Unterkapiteln identifiziert und klassifiziert. Auf eine erneute Auflistung dieser Kriterien wird an dieser Stelle verzichtet.

7.3.2 Bewertung

Um die einzelnen Alternativen bezüglich der identifizierten Auswahlkriterien vergleichen zu können, müssen diese zunächst einheitlich bewertet werden. In der Regel wird hierzu eine Skala von 0 bis 10 Punkten verwendet, bei der 0 Punkte die schlechteste und 10 Punkte die beste Bewertung darstellt. Ziel dieser Bewertung ist es, „aus qualitativen Ergebnisgrößen mit verbaler Rangordnung quantitative Ergebnisgrößen mit messbarer oder abzählbarer Rangordnung zu erhalten“ [87, S.288] (Tabelle 7.1).

Qualitative Bewertung	Quantitative Bewertung
Sehr gut	10
	9
Gut	8
	7
Befriedigend	6
	5
	4
Ausreichend	3
	2
Mangelhaft	1
	0

Tabelle 7.1: Bewertungsskala

Aufgrund des großen Ermessensspielraums bei der Bewertung der qualitativen Eigenschaften muss zwingend darauf geachtet werden, dass bei der Beurteilung der verschiedenen Alternativen keine Verzerrungen auftreten.

Die Bewertung der in dieser Arbeit vorgestellten Datenbanksysteme hinsichtlich der identifizierten Auswahlkriterien basiert auf den Erkenntnissen der konzeptuellen Aufarbeitung, auf den Messergebnissen der Benchmarks und auf praktischen Erfahrungswerten, die im Rahmen mehrerer Versuche mit diesen Systemen gewonnen werden konnten. Anhand der vergebenen Punkte wird deutlich, dass MySQL und MongoDB die Kriterien der konzeptuellen Ebene am besten erfüllen. Dank ihres ausdrucksstarken Datenmodells und ihrer hohen Anfragemächtigkeit sind diese Systeme am besten zur Abbildung vollständiger Miniwelten geeignet. Column Family Stores und vor allem Key Value Stores können diese Kriterien dagegen nur unzureichend erfüllen (Tabelle 7.2).

Datenbankklasse		RDBMS	KVS		DOC	CFS	
Kriterien		MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Konzeptuelle Ebene	Attributnamen	10	0	0	10	10	10
	Einfache Datentypen	10	1	0	7	0	0
	Komplexe Datentypen	0	10	0	10	0	0
	Abbildung heterogener Datensätze	4	3	3	10	7	7
	Aggregationen	0	0	0	10	5	5
	Verbundoperationen	10	0	0	2	0	0
	Umfang an Operatoren	10	1	1	7	5	5
	Anfragesprache	10	0	0	0	0	7
	Sekundäre Zugriffspfade	10	0	0	10	0	7

Tabelle 7.2: Bewertung der konzeptuellen Ebene

Auf interner Ebene verfügen relationale Datenbanken über ein ACID-kompatibles Transaktionskonzept, das Änderungsoperationen einmal pro Sekunde dauerhaft abspeichert. MongoDB verwendet zur Synchronisation von konkurrierenden Schreibzugriffen eine globale Sperre und speichert Änderungsoperationen zehnmal pro Sekunde ab. Column Family Stores synchronisieren konkurrierende Schreibzugriffe nur optimistisch und können deshalb Inkonsistenzen aufweisen. Änderungen werden einmal pro Sekunde (HBase) beziehungsweise einmal alle zehn Sekunden (Cassandra) persistiert. Der Leistungsvorteil, der durch die Lockerung der ACID-Eigenschaften gewonnen werden kann, wurde in den Benchmarks in Kapitel 6.2 deutlich. Key Value Stores und Column Family Stores konnten hierdurch vor allem bei Schreiboperationen große Geschwindigkeitsvorteile erzielen. Der Document Store MongoDB ist für die performante Verarbeitung von einfachen Lese- und Schreiboperationen auf homogenen Datensätzen nicht geeignet (Tabelle 7.3).

Datenbankklasse		RDBMS	KVS		DOC	CFS	
Kriterien		MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Interne Ebene	Atomarität	10	0	0	0	0	0
	Konsistenzsicherung	10	0	0	0	0	3
	Isolation (Locks)	10	0	0	10	0	0
	Dauerhaftigkeit	8	1	2	10	8	4
	Performanz Leseoperationen	5	10	8	1	1	4
	Performanz Schreiboperationen	2	10	8	1	7	8

Tabelle 7.3: Bewertung der internen Ebene

Relationale Datenbanken und der Key Value Store Redis sind im Gegensatz zu den meisten NoSQL-Datenbanken nicht für eine verteilte Datenhaltung konzipiert. Entsprechend hoch ist der zusätzliche Aufwand, der bei diesen Systemen für eine Verteilung betrieben werden muss. Bei den Benchmarks in Kapitel 6.3 zeigte sich, dass vor allem die beiden Column Family Stores und Project Voldemort die besten Skalierbarkeitseigenschaften aufweisen. Schreibanfragen werden in MongoDB und HBase zentral koordiniert, weshalb die Replikate eine hohe Konsistenz aufweisen. Cassandra und Project Voldemort koordinieren Schreibzugriffe dagegen dezentral und besitzen daher eine hohe Verfügbarkeit. Die Konsistenz- und Verfügbarkeitseigenschaften von MySQL und Redis hängen von der gewählten Verteilungsstrategie auf Applikationsebene ab. MongoDB, HBase und Cassandra ermöglichen die Ausführung von MapReduce-Programmen (Tabelle 7.4).

Datenbankklasse		RDBMS	KVS		DOC	CFS	
Kriterien		MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Verteilte Datenhaltung	Aufwand	1	3	10	9	6	10
	Skalierbarkeit: Leseoperationen	2	1	10	10	10	8
	Skalierbarkeit: Schreiboperationen	2	1	8	2	10	10
	Verfügbarkeit	5	5	8	2	2	8
	Konsistenz von Replikaten	5	5	2	8	8	2
	MapReduce-Unterstützung	0	0	0	10	10	10

Tabelle 7.4: Bewertung der verteilten Datenhaltung

Die Bewertung der sonstigen Kriterien basiert zum einen auf Erfahrungswerten, die im praktischen Umgang mit den verschiedenen Datenbanksystemen gewonnen werden konnten, und zum anderen auf den verfügbaren Informationen, die auf den Webseiten der einzelnen Anbieter bereitgestellt werden. Bei diesen Werten handelt es sich um subjektive Einschätzungen, die keine „harten" Daten oder Fakten zur Grundlage haben. Die Einschätzung der Entwicklungsreife basiert auf den Veröffentlichungszeitpunkten und auf der Aktivität der Entwicklergemeinde. Die meisten NoSQL-Datenbanken wurden zwischen den Jahren 2007 und 2009 entwickelt und besitzen somit einen hohen Entwicklungsrückstand auf das relationale Datenbanksystem MySQL (1994). MongoDB, Cassandra und HBase werden sehr aktiv weiterentwickelt. Neue Versionen dieser Datenbanksysteme erscheinen mehrfach pro Monat. Die beiden Key Value Stores werden dagegen deutlich langsamer weiterentwickelt. Während neue Redis-Versionen mehrmals jährlich veröffentlicht werden, ruht die Entwicklung von Project Voldemort seit September 2012[69]. Für viele NoSQL-Datenbanken wurden in den letzten Jahren Werkzeuge entwickelt, mit denen die Administration und die Entwicklung mit diesen Systemen deutlich erleichtert werden. Vor allem für den Document Store MongoDB wurden sehr viele Werkzeuge entworfen[70]. Dennoch sind die Quantität und Qualität an Werkzeugunterstützung weiterhin wesentlich geringer als bei relationalen Datenbanken. Das mangelnde Fachwissen über NoSQL-Datenbanken ist ein Motivationspunkt dieser Arbeit. Auch wenn der enorme Wissensrückstand auf relationale Datenbanken nicht in einer Skala von 0 bis 10 ausgedrückt werden kann, muss an dieser Stelle zumindest der Vorteil von MongoDB gegenüber anderen NoSQL-Datenbanken in diesem Bereich hervorgehoben (Tabelle 7.5).

Datenbankklasse		RDBMS	KVS		DOC	CFS	
Kriterien		MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Sonstige Kriterien	Entwicklungsreife	10	3	0	6	6	6
	Werkzeugunterstützung	10	1	0	6	3	2
	Fachwissen	10	1	0	3	2	1
	Standardisierung	10	0	0	0	0	0

Tabelle 7.5: Bewertung der sonstigen Kriterien

[69] https://github.com/voldemort/voldemort/downloads
[70] Eine umfangreiche Auswahl findet sich unter http://docs.mongodb.org/ecosystem/tools/administration-interfaces

7.3.3 Gewichtung

Kriterien können mit Gewichtungsfaktoren multipliziert werden, um die unterschiedlichen Prioritäten dieser Kriterien bei der Entscheidungsfindung zu berücksichtigen. Die Gewichtung ist von rein subjektiver Natur und hängt von den Präferenzen des Entscheidungsträgers ab. Gewichtungsfaktoren werden prozentual angegeben, weshalb die Summe aller Einzelfaktoren 100 Prozent ergeben muss.

Die zur Auswahl eines Datenbanksystems identifizierten Kriterien wurden in die vier Gruppen konzeptuelle Ebene, interne Ebene, verteilte Datenhaltung und sonstige Kriterien klassifiziert. Um eine einheitliche und strukturierte Bewertung aller Kriterien zu ermöglichen, sollte das Gesamtgewicht von 100 Prozent zunächst auf diese vier Gruppen verteilt werden. Hierdurch werden die Anforderungsschwerpunkte eines Anwendungsfalls bereits grob umrissen. Die zentrale Fragestellung bei der Gewichtung der Kriterienklassen lautet, ob das eingesetzte Datenbanksystem eine Miniwelt möglichst vollständig abbilden soll oder ob das System vor allem aus leistungsspezifischen Gesichtspunkten eingesetzt wird. Unter Berücksichtigung der Anforderungen können dann die Schwerpunkte auf die konzeptuelle Ebene oder auf die interne Ebene und verteilte Datenhaltung gelegt werden.

Anschließend können die in den Gruppen enthaltenen Kriterien durch weitere Gewichtungen feingranular an die spezifischen Anforderungen eines Anwendungsfalls angepasst werden. Auf konzeptueller Ebene gilt es, einen Kompromiss zwischen der Ausdrucksstärke (Mächtigkeit) und der Flexibilität eines Datenmodells zu finden. Des Weiteren muss entschieden werden, ob Beziehungen mithilfe von Referenzen oder Einbettungen abgebildet werden sollen. Entsprechend der Auswahl muss dann ein Schwerpunkt bei Verbundoperationen oder Aggregationen gesetzt werden. Auf interner Ebene muss ein Kompromiss zwischen der Verlässlichkeit und der Leistungsfähigkeit der Datenverarbeitung gefunden werden. Je nach Anwendungsfall kann durch einen Verzicht auf bestimmte ACID-Eigenschaften ein Leistungsvorteil bei der Verarbeitung von Lese- und Schreiboperationen erzielt werden. Ist eine verteilte Datenhaltung notwendig, ist sicherlich der hierfür zu betreibende Aufwand ein entscheidendes Auswahlkriterium. Der Entscheidung für eine hohe Verfügbarkeit oder eine hohe Konsistenz sollte eine geringere Priorität beigemessen werden, da jedes Datenbanksystem sehr viele Konfigurationsmöglichkeiten bietet und sich demzufolge individuell auf die Anforderungen eines Anwendungsfalls anpassen lässt. Bei der Auswahl eines Datenbanksystems können weitere Kriterien eine entscheidende Rolle spielen. Je nach Anwendungsfall müssen deshalb gerade auch die „sonstigen Kriterien“ berücksichtigt werden.

7.3.4 Nutzwertberechnung

Nach der Identifizierung und Klassifizierung der Auswahlkriterien, der Bewertung der Eigenschaften und der Gewichtung der Kriterien kann in einem abschließenden Schritt der Nutzwert einer Alternative berechnet werden. Der Nutzwert einer Alternative a_i berechnet sich aus der Summe der Teilnutzwerte, die für jedes Kriterium k berechnet werden. Ein Teilnutzwert eines Kriteriums ist das Produkt aus dessen Gewichtung g_k und der für jede Alternative spezifischen Bewertung $b_k(a_i)$.

$$a_i = \sum_{k=1}^{K} g_k \cdot b_k(a_i)$$

Bei der Entscheidungsfindung wird dann die Alternative bestimmt, die von allen Alternativen den höchsten Nutzwert besitzt. Der Nutzwert wird in dieser Arbeit aus Gründen der Nachvollziehbarkeit prozentual zur maximal möglichen Punktzahl angegeben.

7.3.5 Kriterienkatalog

Die Klassifizierung, Bewertung und Gewichtung der Kriterien sowie die Berechnung des Nutzwertes kann tabellarisch in Form eines Kriterienkatalogs dargestellt werden (Tabelle 7.6). Für jede der vier Kriterienklassen *Konzeptuelle Ebene*, *Interne Ebene*, *Verteilte Datenhaltung* und *Sonstige Kriterien* kann von einem Entscheidungsträger eine Gewichtung vorgenommen werden. Darüber hinaus wird eine feingranulare Abstufung der einzelnen Kriterien innerhalb einer Klasse ermöglicht. Für jedes Datenbanksystem wird der prozentuale Teilnutzwert innerhalb einer Kriterienklasse berechnet. Die Summe aller Teilnutzwerte unter Berücksichtigung der Gewichtung der Kriterienklassen ergibt den Nutzwert eines Datenbanksystems.

In dem in Tabelle 7.6 dargestellten Kriterienkatalog wurde eine relativ ausgeglichene Gewichtung der Kriterien gewählt, bei der jedes Kriterium in etwa den gleichen Einfluss auf den Nutzwert eines Datenbanksystems ausübt. Auch wenn diese Gewichtung nicht den Anforderungen eines spezifischen Anwendungsfalls gerecht wird, kann hierdurch jedoch ein neutraler Überblick über den bereitgestellten Funktionsumfang der Datenbanksysteme gegeben werden.

Nutzwertanalysen ermöglichen eine transparente und nachvollziehbare Bewertung unterschiedlicher Alternativen unter Berücksichtigung von quantitativen und qualitativen Auswahlkriterien. Allerdings hängen die ermittelten Nutzwerte sehr stark von vielen subjektiven Parametern ab. Vor allem die Bewertung von Eigenschaften und die Gewichtung von Auswahlkriterien bieten einen sehr großen Spielraum für die Beeinflussung der berechneten Ergebnisse. Aufgrund dieser Tatsache kann der hier entwickelte Kriterienkatalog ausschließlich einen Indikator für den Nutzen eines Datenbanksystems in einem bestimmten Anwendungsfall bereitstellen. Der Kriterienkatalog sollte ebendarum als Hilfestellung und nicht als Ergebnis eines komplexen Entscheidungsprozesses interpretiert werden.

Datenbankklasse				RDBMS	KVS		DOC	CFS	
Kriterien		Gewichtung		MySQL	Redis	Voldemort	MongoDB	HBase	Cassandra
Konzeptuelle Ebene	Attributnamen	36%	10%	10	0	0	10	10	10
	Einfache Datentypen		10%	10	1	0	7	0	0
	Komplexe Datentypen		10%	0	10	0	10	0	0
	Abbildung heterogener Datensätze		10%	4	3	3	10	7	7
	Aggregationen		10%	0	0	0	10	5	5
	Verbundoperationen		10%	10	0	0	2	0	0
	Umfang an Operatoren		20%	10	1	1	7	5	5
	Anfragesprache		10%	10	0	0	0	0	7
	Sekundäre Zugriffspfade		10%	10	0	0	10	0	7
	Teilnutzwert		100%	74%	16%	5%	73%	32%	46%
Interne Ebene	Atomarität	24%	13%	10	0	0	0	0	0
	Konsistenzsicherung		13%	10	0	0	0	0	3
	Isolation (Locks)		13%	10	0	0	10	0	0
	Dauerhaftigkeit		13%	8	1	2	10	8	4
	Performanz Leseoperationen		25%	5	10	8	1	1	4
	Performanz Schreiboperationen		25%	2	10	8	1	7	8
	Teilnutzwert		100%	65%	51%	43%	30%	30%	39%
Verteilte Datenhaltung	Aufwand	24%	25%	1	3	10	9	6	10
	Skalierbarkeit: Leseoperationen		15%	2	1	10	10	10	8
	Skalierbarkeit: Schreiboperationen		15%	2	1	8	2	10	10
	Verfügbarkeit		15%	5	5	8	2	2	8
	Konsistenz von Replikaten		15%	5	5	2	8	8	2
	MapReduce-Unterstützung		15%	0	0	0	10	10	10
	Teilnutzwert		100%	24%	26%	67%	71%	75%	82%
Sonstige Kriterien	Entwicklungsreife	16%	25%	10	3	0	6	6	6
	Werkzeugunterstützung		25%	10	1	0	6	2	3
	Fachwissen		25%	10	1	0	3	1	2
	Standardisierung		25%	10	0	0	0	0	0
	Teilnutzwert		100%	100%	13%	0%	38%	23%	28%
Nutzwert		**100%**		**64%**	**26%**	**28%**	**56%**	**40%**	**50%**

Tabelle 7.6: Kriterienkatalog zur Auswahl eines Datenbanksystems

7.4 Polyglot Persistence

Bei der Gewichtung der im Kriterienkatalog aufgeführten Auswahlkriterien fällt auf, dass auf konzeptueller Ebene ein Kompromiss zwischen der Mächtigkeit und Flexibilität des Datenmodells und auf interner Ebene ein Kompromiss zwischen vollständiger ACID-Unterstützung und Leistungsfähigkeit gefunden werden muss. Berücksichtigt man zudem die Kriterien der verteilten Datenhaltung und die sonstigen Kriterien, entsteht ein sechsdimensionaler Raum, innerhalb dessen die Vor- und Nachteile der verschiedenen Datenbanksysteme veranschaulicht werden können (Abbildung 7.14).

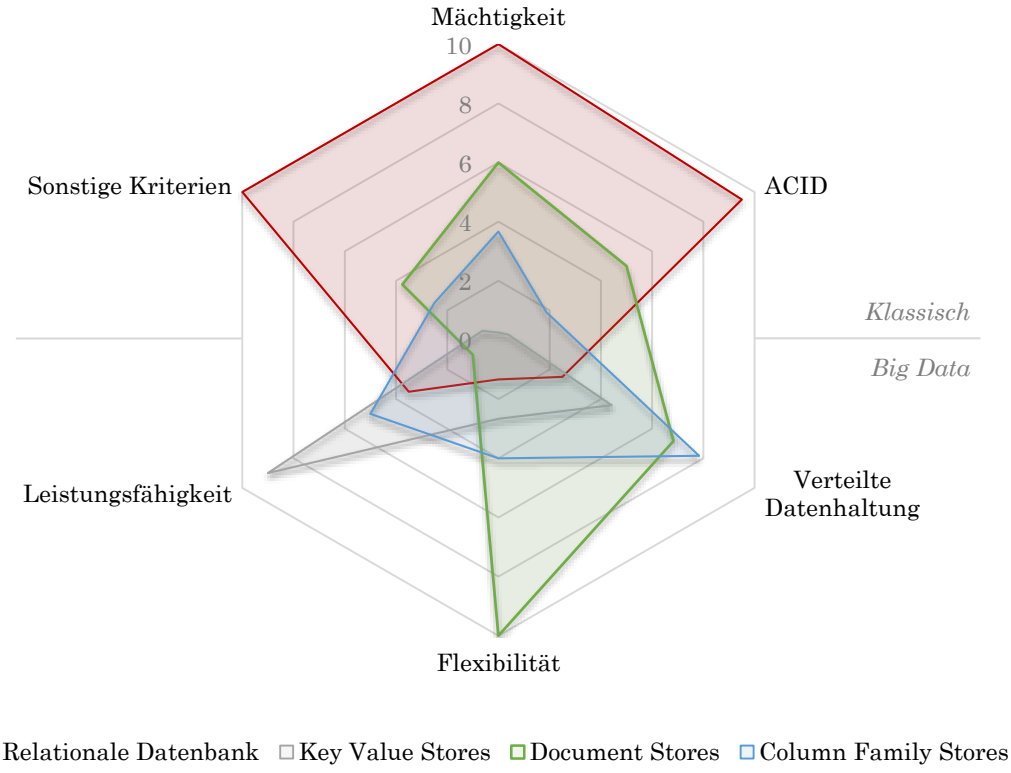


Abbildung 7.14: Die Vor- und Nachteile der unterschiedlichen Datenbankklassen

Der in Abbildung 7.14 dargestellte Vergleich zwischen den verschiedenen Datenbankklassen beruht auf den durchschnittlichen Bewertungspunkten der einzelnen Systeme. Je größer die Fläche einer Datenbankklasse in dieser Grafik ist, desto breiter ist das Spektrum an möglichen Einsatzgebieten, das von diesen Datenbanken abgedeckt wird. Aus dieser Abbildung wird ersichtlich, dass relationale Datenbanken und Docu-

ment Stores angesichts ihres hohen Funktionsumfangs in nahezu jedem Anwendungsfall eingesetzt werden können, während Column Family Stores und vor allem Key Value Stores auf deutlich kleinere Bereiche begrenzt sind.

Zudem wird in Abbildung 7.14 der fundamentale Unterschied zwischen relationalen Datenbanken und NoSQL-Datenbanken deutlich. Relationale Datenbanken sind auf die Anforderungen der „klassischen" Datenhaltung ausgelegt, in denen eine hohe Mächtigkeit, ACID-Unterstützung und viele „sonstige" Kriterien gefordert werden. Dieses Spektrum wird von relationalen Datenbanken nahezu konkurrenzlos dominiert, weshalb diese Systeme als „Allzweckwaffe" in fast jedem Anwendungsfall eingesetzt werden können. NoSQL-Datenbanken sind dagegen gezielt auf die Kriterien von Big Data-Anwendungsfällen ausgelegt, in denen relationale Datenbanken deutliche Schwächen offenbaren. Beispiele hierfür sind eine zu geringe Leistungsfähigkeit bei Anfragen (Key Value Stores), mangelnde Flexibilität bei der Speicherung von heterogenen Datensätzen (Document Stores) oder die Verarbeitung von sehr großen Datenmengen (Column Family Stores).

Aufgrund ihres hohen Spezialisierungsgrades können NoSQL-Datenbanken in der Regel nur einen Teil der Anforderungen eines Anwendungsfalls sinnvoll abdecken. In den meisten Anwendungsfällen werden diese Systeme darum in Kombination mit relationalen Datenbanken eingesetzt. Die Philosophie, NoSQL-Datenbanken als Ergänzung zu relationalen Lösungen einzusetzen, wird durch Begriffe wie „Not Only-SQL" oder „Polyglot Persistence" [102] zum Ausdruck gebracht. Ähnlich wie bei der Wahl einer passenden Programmiersprache sollen Entwickler anwendungsfallabhängig eine oder mehrere Datenbanken aussuchen, welche in ihrem Zusammenspiel die Anforderungen bestmöglich erfüllen. Der Einsatz mehrerer Datenbanksysteme erhöht jedoch den Aufwand, der zur Administration des Gesamtsystems benötigt wird. Neben weiteren Servern muss ein Unternehmen mit mehreren verschiedenen Datenbanksystemen auch in zusätzliches Fachwissen (neues Personal, Fortbildungen) investieren.

Key Value Stores und relationale Datenbanken

Key Value Stores werden häufig mit relationalen Datenbanken kombiniert, um als Caching-Systeme bestimmte Leseanfragen zu beschleunigen. Zwar lässt sich die Leistungsfähigkeit von relationalen Datenbanken mittels einer Erweiterung des Arbeitsspeichers ebenfalls erhöhen, allerdings bleibt dennoch ein gewisser Overhead bestehen, der durch Parsen von SQL-Anfragen, Öffnen und Schließen von Tabellen sowie durch Synchronisationsmechanismen entsteht. Ein Beispiel für intensives Caching bietet Facebook. Hier werden sämtliche Benutzerinteraktionen wie Statusmeldungen, Likes und Shares aber auch Facebooks Timeline persistent in einem der weltweit größten MySQL-Cluster persistent gespeichert [109]. Der Großteil der Leselast wird jedoch von Memcached bewältigt. Nach eigenen Angaben ist der Memcached-Cluster bei Facebook in der Lage, über eine Milliarde Anfragen pro Sekunde zu verarbeiten [54, 134].

Document Stores und relationale Datenbanken

Document Stores eignen sich zur Verarbeitung heterogener Datensätze, sind aber bei homogenen Datensätzen relationalen Datenbanken unterlegen. Infolgedessen werden Document Stores häufig gemeinsam mit relationalen Datenbanken eingesetzt. So wird beispielsweise in dem bereits vorgestellten Anwendungsfall von CustomInk der Produktkatalog mit MongoDB betrieben, aber andere geschäftskritische Daten weiterhin in Oracle gespeichert. Ähnlich verhält es sich mit Content Management Systemen, die zwar die Inhalte einer Webseite in einem Dokument speichern können, aber beispielsweise Benutzerinformationen weiterhin in einem relationalen Datenbanksystem verwalten.

Column Family Stores und relationale Datenbanken

Column Family Stores werden in Anwendungsfällen eingesetzt, in denen großen Datenmengen mit einer sehr hohen Schreibgeschwindigkeit verarbeitet werden sollen. Wie bereits erwähnt, werden sämtliche Benutzerdaten und Interaktionen bei Facebook in MySQL gespeichert und Leseanfragen durch den Einsatz von Memcached beschleunigt. Die Nachrichten, die zwischen den Anwendern ausgetauscht werden, befinden sich dagegen in einem HBase-Cluster, da dieser deutlich besser für die Verarbeitung sehr großer Datenmengen und hoher Schreiblasten geeignet ist.

7.5 Zusammenfassung

Im Gegensatz zu relationalen Datenbanken, die quasi als „Allzweckwaffe“ in einem sehr breiten Spektrum an Anwendungsfällen eingesetzt werden können, sind NoSQL-Datenbanken hochgradig spezialisierte Systeme, welche auf die charakteristischen Anforderungen von Anwendungsfällen im Big Data-Bereich ausgelegt sind. Aufgrund dieses hohen Spezialisierungsgrades ist gegenwärtig ein sehr hohes Maß an Expertise notwendig, um das für einen Anwendungsfall ideale Datenbanksystem auswählen zu können. Ohne umfangreiche Erfahrungen im Umgang mit den einzelnen Datenbanken und daraus resultierenden Erkenntnissen über die Vor- und Nachteile der verschiedenen Systeme ist die Datenbankwahl nur mit einem unverhältnismäßig hohen Aufwand durchführbar. Aufgrund dieser Problematik wurde in diesem Kapitel ein Kriterienkatalog erstellt, mit dessen Hilfe der komplexe Auswahlprozess eines passenden Datenbanksystems deutlich erleichtert wird.

Der Kriterienkatalog basiert auf einer umfangreichen Identifikation relevanter Auswahlkriterien. Neben den Kriterien, die sich aus den Big Data-Anforderungen und den spezifischen Eigenschaften der unterschiedlichen Datenbankkonzepte ableiten lassen, flossen Kriterien in den Katalog mit ein, die im Rahmen einer umfangreichen Analyse praktischer Anwendungsfälle identifiziert werden konnten. Die Auswahlkriterien

wurden in die vier Klassen *Konzeptuelle Ebene*, *Interne Ebene*, *Verteilte Datenhaltung* und *Sonstige Kriterien* gegliedert.

Die Analyse der Kriterien ergab, dass die Wahl eines passenden Datenbanksystems sowohl von quantitativen aber vor allem auch von qualitativen Kriterien abhängig ist. Um diese unterschiedlichen Kriterien einheitlich vergleichen zu können, wurde auf das Konzept der Nutzwertanalysen zurückgegriffen. Hierbei werden verschiedene Alternativen (Datenbanksysteme) für jedes Kriterium in einer Skala von 0 bis 10 Punkten bewertet. Da jedes Kriterium mithilfe von Prozentzahlen verschieden gewichtet werden kann, ist die Berücksichtigung anwendungsfallspezifischer Anforderungen problemlos möglich. Der Nutzwert eines Datenbanksystems berechnet sich aus der gewichteten Summe der einzelnen Punktwerte.

Der Kriterienkatalog bietet eine hilfreiche Unterstützung bei der Wahl eines passenden Datenbanksystems. Dank der transparenten Bewertung und Gewichtung der einzelnen Kriterien kann die Entscheidung für oder gegen ein bestimmtes Datenbanksystem sehr gut nachvollzogen werden. Allerdings sollte der Kriterienkatalog ausschließlich als Hilfsmittel und nicht als Ergebnis des komplexen Auswahlprozesses herangezogen werden. In Anbetracht der äußerst subjektiven Bewertung und Gewichtung der Kriterien, kann der Kriterienkatalog nur einen Indikator und keine fundierte Aussage für die Eignung eines Datenbanksystems zur Bearbeitung bestimmter Anforderungen bereitstellen.

8

ZUSAMMENFASSUNG, AUSBLICK UND FAZIT

Kapitelinhalt

- Zusammenfassung der Ergebnisse
- Ausblick auf die weitere Entwicklung
- Fazit

Dieses Kapitel fasst die in dieser Arbeit durchgeführten Analysen zusammen, gibt einen Ausblick auf die zukünftige Entwicklung von NoSQL-Datenbanken und endet mit einem Fazit zu dieser Thematik.

8.1 Zusammenfassung

Big Data beschreibt einen derzeitigen Trend in der Informations- und Kommunikationsverarbeitung, bei dem extrem große Mengen verschieden strukturierter Daten in hoher Geschwindigkeit verarbeitet werden. Die hierbei gestellten Anforderungen überschreiten immer häufiger die Kapazität und die Leistungsfähigkeit der bisher eingesetzten relationalen Datenbanksysteme. Infolge dieser Entwicklung wurden seit 2006 verschiedene Datenbanksysteme entworfen, die für den Einsatz in Big Data Anwendungsfällen konzipiert sind und als NoSQL-Datenbanken bezeichnet werden. Diese Systeme verfügen über ein flexibles Datenmodell sowie eine hohe Leistungsfähigkeit und sind von Beginn an auf Skalierbarkeit ausgerichtet. Allerdings weisen diese Systeme im Vergleich zu relationalen Datenbanken, die seit mehreren Jahrzehnten erfolgreich eingesetzt und kontinuierlich weiterentwickelt werden, die deutlich geringere Entwicklungsreife auf. Darüber hinaus existiert, bedingt durch die hohe Heterogenität der Systeme und die mangelnde Erfahrung im Umgang mit NoSQL-Datenbanken, kaum verlässliche und qualitativ hochwertige Fachliteratur zu dieser Thematik. Anwender von NoSQL-Datenbanken stehen deshalb zu Beginn eines Softwareprojekts vor zwei zentralen Problemstellungen:

1. Die Wahl eines passenden Datenbanksystems,
2. Der effiziente Einsatz des ausgewählten Datenbanksystems.

Diese beiden Fragestellungen lassen sich aufgrund des Mangels an Fachwissen derzeit nur mithilfe umfangreicher praktischer Versuche lösen. Der hierfür benötigte Aufwand und die mit den Systemen steigende Fehleranfälligkeit sorgen für deutlich höhere Investitionskosten beim Einsatz eines NoSQL-Datenbanksystems im Vergleich zu relationalen Lösungen. Um den Einsatz von NoSQL-Datenbanken zu erleichtern und somit die Kosten zu senken, wurden im Rahmen dieser Arbeit zwei derzeit dringend benötigte Hilfsmittel entwickelt:

1. Ein Kriterienkatalog zur Auswahl eines passenden Datenbanksystems,
2. Allgemeingültige Abbildungsregeln zum effizienten Einsatz von NoSQL-Datenbanken.

Die Basis für diese beiden Hilfsmittel bildet eine methodische Aufarbeitung der Konzepte, die den verschiedenen NoSQL-Datenbanken zugrunde liegen und in der Literatur bisher kaum behandelt wurden. Durch diese neuartige Herangehensweise konnten erstmalig allgemeingültige und fachlich fundierte Aussagen über die Vor- und Nachteile der unterschiedlichen Datenbanksysteme getroffen werden. Diese Erkenntnisse flossen zusammen mit den Ergebnissen eines umfassenden Vergleichs auf interner Ebene in die Bewertung verschiedener Datenbanken ein. Die Bewertung der Systeme wurde anschließend in einem Kriterienkatalog bereitgestellt, der es durch eine individuelle Gewichtung verschiedener Auswahlkriterien ermöglicht, die spezifischen Anforderungen eines jeden Anwendungsfalls feingranular abzubilden. Da selbst unerfahrene Anwender die Eignung eines Datenbanksystems anhand einer Punkteskala sehr einfach ablesen können, ist der hier erstellte Kriterienkatalog ein wertvolles Hilfsmittel bei der Auswahl eines Datenbanksystems.

Im Rahmen der konzeptuellen Aufarbeitung wurden die Vor- und Nachteile der unterschiedlichen Datenmodelle herausgestellt und die Anfragemächtigkeit der verschiedenen Datenbanksysteme detailliert untersucht. Basierend auf diesen Ergebnissen wurden allgemeingültige und umfangreiche Abbildungsregeln erstellt, mit deren Hilfe konzeptuelle Datenmodelle systematisch in die logischen Datenmodelle der verschiedenen NoSQL-Datenbanken überführt werden können. Dank dieser Modellierungsmuster wird einerseits die Einarbeitungszeit verkürzt und anderseits die Wahrscheinlichkeit eines ineffizienten Einsatzes eines NoSQL-Datenbanksystems sehr stark reduziert.

8.2 Ausblick

Durch die geringe Entwicklungsreife und den Verzicht auf viele bewährte Techniken, wie deklarative Anfragesprachen oder ACID-kompatible Transaktionsverwaltungen, werden NoSQL-Datenbanken von Beginn an seitens der Anhänger von relationalen Datenbanken sehr kritisch betrachtet. Häufig werden NoSQL-Datenbanken als Hype mit sehr kurzer Lebensdauer bezeichnet. So heißt es in einem im Jahr 2011 von Oracle zu diesem Thema herausgebrachten Whitepaper:

> *"NoSQL databases are beginning to feel like an ice cream store that entices you with a new flavor of the month. However, you shouldn't get too attached to any of the flavors, because it may not be around for that long." – Oracle* [6]

Diese Einschätzung wurde jedoch bereits ein halbes Jahr später indirekt revidiert, als Oracle eine eigene, proprietäre NoSQL-Datenbank auf den Markt brachte [5, 137] und das oben aufgeführte Whitepaper zeitgleich wieder von der offiziellen Webseite entfernte. Dass die seitens der NoSQL-Bewegung geäußerten Kritikpunkte an relationalen Datenbanken durchaus ihre Berechtigung zu haben scheinen, zeigt nicht nur das Beispiel von Oracle, sondern auch die allgemeine Entwicklung von relationalen Datenbanken in den letzten drei Jahren. Unter dem Begriff *NewSQL* [12] gruppieren sich zunehmend mehr relationale Datenbanksysteme, die auch Konzepte von NoSQL-Datenbanken beinhalten und hierdurch versuchen, die Vorteile dieser beiden Datenbankwelten zu vereinen. NewSQL-Datenbanken sind von Beginn an auf eine verteilte Datenhaltung ausgelegt, verwenden optimistische Synchronisationsmechanismen und setzen teilweise den Hauptspeicher als primäres Speichermedium ein. Gleichzeitig besitzen sie ein relationales Datenmodell, stellen eine SQL-Schnittstelle bereit und verfügen über ein ACID-kompatibles Transaktionskonzept. Durch die Unterstützung von Verbundoperationen und Transaktionen können NewSQL-Datenbanken durchaus schlechtere Skalierbarkeitseigenschaften als NoSQL-Datenbanksysteme aufweisen. Allerdings liegt die Entscheidung beim Anwender, ob und in welchem Umfang er auf diese Techniken zurückgreift und wie stark er dadurch die Skalierbarkeit beeinflusst. Die Forderung nach Transaktionsunterstützung scheint derart stark zu wiegen, dass die daraus resultierenden Performanznachteile in vielen Anwendungsfällen tolerierbar sind. In der Veröffentlichung von Googles neuem NewSQL-Datenbanksystem *Spanner* [37] ist folgendes Statement zu lesen:

> *"We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions." – Google* [37]

Diese Aussage ist insofern erstaunlich, weil Google zuvor bei Bigtable bewusst auf Transaktionen verzichtete, da zum damaligen Zeitpunkt die meisten Anwendungen bei Google diese Funktionalität nicht benötigten [30]. Neben Googles Spanner gehören

VoltDB[71], *Clustrix*[72] und *NuoDB*[73] zu den bekanntesten NewSQL-Datenbanksystemen. Diese Datenbanken wurden von Grund auf neu entwickelt und berücksichtigen viele Konzepte und Anregungen der NoSQL-Bewegung. Zusätzlich werden mehrere bestehende relationale Datenbanken zur Gruppe von NewSQL-Datenbanken hinzugezählt, welche mithilfe von Erweiterungen eine verteilte Datenhaltung unterstützen. Zu diesen Systemen zählen unter anderem *ScaleBase*[74], *dbShards*[75] und *MySQL Cluster*[76].

NewSQL-Datenbanken verfolgen einen sehr interessanten und vielversprechenden Ansatz, indem sie versuchen, die Vorteile von relationalen Datenbanken und NoSQL-Datenbanken zu kombinieren. Zum aktuellen Zeitpunkt existieren jedoch kaum Erfahrungswerte im Umgang mit diesen Systemen, weshalb eine umfangreiche Bewertung und ein aussagekräftiger Vergleich mit NoSQL-Datenbanken gegenwärtig nicht möglich sind. Allerdings lassen sich anhand der NewSQL-Bewegung Tendenzen erkennen, welche Aussagen über die zukünftige Entwicklung von NoSQL-Datenbanken und relationalen Datenbanken ermöglichen. So ist es sehr wahrscheinlich, dass Ansätze und Konzepte von NoSQL-Datenbanken weiterhin in die zukünftige Entwicklung der etablierten relationalen Datenbanksysteme mit einfließen werden. Als Beispiele hierfür können neben der JSON-Unterstützung von PostgreSQL [144, S.253] auch die Anbindung vieler relationaler Data Warehouse Systeme an das MapReduce-Framework Hadoop angeführt werden. Parallel zur Adaption von NoSQL-Konzepten in relationalen Datenbanken werden NoSQL-Datenbanken in einem sehr hohen Tempo weiterentwickelt und mit Funktionalitäten ausgestattet, die aus relationalen Datenbanken bestens bekannt sind. Hierzu zählen unter anderem deklarative Anfragesprachen, eine steigende Schemamächtigkeit und eine zunehmende Unterstützung von Transaktionen. Hält dieser Trend weiterhin an, wird es zunehmend schwieriger werden, eine klare Grenze zwischen den verschiedenen Datenbankarten ziehen zu können. In ein paar Jahren könnte dann ein sehr breiter Markt an Datenbanken existieren, bei dem nur noch wenige Experten die Unterschiede zwischen den einzelnen Systemen aufzählen können.

[71] http://www.voltdb.com
[72] http://www.clustrix.com
[73] http://www.nuodb.com
[74] http://www.scalebase.com
[75] http://www.dbshards.com
[76] http://www.mysql.com/products/cluster

8.3 Fazit

NoSQL-Datenbanken sind hochgradig spezialisierte Systeme, die auf die spezifischen Anforderungen von Big Data-Anwendungsfällen ausgelegt sind. Diese Datenbanken können in Bereichen, in denen ein flexibles Datenmodell, eine hohe Leistungsfähigkeit und lineare Skalierbarkeit gefordert werden, eine interessante Alternative zu relationalen Standardlösungen bieten. Derzeit existieren über 150 verschiedene Datenbanksysteme [49], die zur Familie der NoSQL-Datenbanken gezählt werden können. Diese Systeme lassen sich grob in die drei Klassen *Key Value Stores*, *Document Stores* und *Column Family Stores* unterteilen.

Key Value Stores: Key Value Stores sind minimalistische Systeme, die auf eine maximale Performanz von einfachen Lese- und Schreiboperationen ausgelegt sind. Diese Systeme eignen sich deshalb für Anwendungsbereiche wie Caching, Session-Management, Echtzeitanalysen und Activity Feeds in sozialen Netzwerken. Sobald weitere Funktionalitäten als das einfache Ablegen und Aufrufen von Schlüssel-Wert-Paaren benötigt werden, muss angesichts des sehr schnell steigenden Mehraufwands auf Applikationsebene von der Verwendung von Key Value Stores abgeraten werden.

Document Stores: Document Stores verfügen über ein mächtiges und zugleich flexibles Datenmodell und können daher mehrere Entitäten umfassende Miniwelten mit variablen Datensätzen sehr gut abbilden. Typische Einsatzgebiete sind heterogene Produktkataloge, Archive, Content Management Systeme und Datenintegrationen. Document Stores sind relationalen Datenbanken bei der Verarbeitung von homogenen Datensätzen unterlegen und weisen eine geringere Leistungsfähigkeit als Key Value Stores und Column Family Stores auf.

Column Family Stores: Column Family Stores zeichnen sich durch eine hohe Leistungsfähigkeit bei Schreiboperationen sowie sehr gute Skalierbarkeitseigenschaften aus. Diese Systeme sind daher für eine performante Verarbeitung sehr großer Datenmengen geeignet, die persistent für umfangreiche Analysen gespeichert werden müssen. Typische Einsatzgebiete sind vor allem Logging- und Echtzeitanalyseaufgaben. Column Family Stores sind nicht für Anwendungsfälle geeignet, in denen ein mächtiges Datenmodell und umfangreiche Anfragefunktionalitäten benötigt werden.

Neben den Vorteilen, die NoSQL-Datenbanken gegenüber relationalen Datenbanken in bestimmten Anwendungsfällen aufweisen können, besitzen NoSQL-Datenbanken auch mehrere Nachteile, die seitens der NoSQL-Bewegung eher selten thematisiert werden. Zu diesen weniger offensichtlichen Nachteilen zählen unter anderem fehlende Entwicklungsreife, geringerer Support, geringere Nachhaltigkeit, minimale Werkzeugunterstützung, mangelndes Fachwissen, fehlende Standardisierung und eine höhere Komplexität auf Anwendungsebene. Diese Nachteile sollten vor dem Einsatz eines NoSQL-Datenbanksystems dringend berücksichtigt werden, um teure Fehlinvestitionen in ein unpassendes Datenbanksystem zu vermeiden.

Durch die hohe Vielfalt an NoSQL-Datenbanksystemen und das zunehmende Aufkommen der NewSQL-Bewegung existieren derzeit mehrere vielversprechende Alternativen zu den relationalen Standardlösungen. Es wird interessant zu verfolgen sein, wie die etablierten Anbieter von relationalen Datenbanken zukünftig auf diese neuen Ansätze reagieren werden.

LITERATURVERZEICHNIS

[1] D. J. Abadi, A. Marcus, S. R. Madden und K. Hollenbach, “SW-Store: a vertically partitioned DBMS for Semantic Web data management,” in *The International Journal on Very Large Data Bases*, Bd. 18, Nr. 2, S. 385–406, 2009.

[2] S. Abiteboul, “Querying Semi-Structured Data,” in *Proceedings of the 6th International Conference on Database Theory (ICDT ’97)*, 1997, S. 1–18.

[3] R. Agrawal, A. Somani und Y. Xu, “Storage and Querying of E-Commerce Data,” in *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB ’01)*, 2001, S. 149–158.

[4] R. Agrawal, R. Srikant und Y. Xu, “Database Technologies for Electronic Commerce,” in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB ’02)*, 2002, S. 1055–1058.

[5] M. Alam, A. Muley, C. Kadaru und A. Joshi, *Oracle NoSQL Database: Real-Time Big Data Management for the Enterprise*. New York, NY, USA, McGraw-Hill Companies, 2014.

[6] Alan Downing, “Debunking the NoSQL Hype,” Oracle, Redwood Shores, CA, USA, Technischer Report, 2011.

[7] Amar Arsikere, “Building a scalable game server,” 2011. Verfügbar: http://code.zynga.com/2011/07/building-a-scalable-game-server. Zugriffsdatum: 16.09.2014.

[8] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the AFIPS ’67 Spring Joint Computer Conference*, 1967, S. 483–485.

[9] J. C. Anderson, J. Lehnardt und N. Slater, *CouchDB: The Definitive Guide*. Sebastopol, CA, USA: O’Reilly Media, 2010.

[10] E. Anuff, “Indexing in Cassandra,” 2011. Verfügbar: http://www.anuff.com/2011/02/indexing-in-cassandra.html. Zugriffsdatum: 16.09.2014.

[11] Apache Software Foundation, “The Apache HBase Reference Guide,” 2014. Verfügbar: http://hbase.apache.org/book.html. Zugriffsdatum: 16.09.2014.

[12] M. Aslett, “How will the database incumbents respond to NoSQL and NewSQL?,” The 451 Group, New York, NY, USA, Technischer Report, 2011.

[13] M. Ballard, “National Archives releases public application programming interface for 11m records,” 2011. Verfügbar: http://www.computerweekly.com/news/2240105501/National-Archives-releases-public-application-programming-interface-for-11m-records. Zugriffsdatum: 28.08.2014.

[14] K. Banker, *MongoDB in Action*. Shetter Island, NY, USA: Manning Publications, 2011.

[15] D. Beaver, S. Kumar, H. C. Li, J. Sobel und P. Vajgel, “Finding a needle in Haystack: Facebook’s photo storage,” in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI’10)*, 2010, S. 47–60.

[16] J. L. Beckmann, A. Halverson, R. Krishnamurthy und J. F. Naughton, “Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format,” in *Proceedings of the 22nd International Conference on Data Engineering (ICDE ’06)*, 2006, S. 58.

[17] B. H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors,” in *Communications of the ACM*, Bd. 13, Nr. 7, S. 422–426, Juli 1970.

[18] R. Bodkin, “Foursquare’s MongoDB Outage,” 2010. Verfügbar: http://www.infoq.com/news/2010/10/4square_mongodb_outage. Zugriffsdatum: 16.09.2014.

[19] D. Borthakur, S. Rash, R. Schmidt, A. Aiyer, J. Gray, J. Sen Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov und A. Menon, “Apache Hadoop Goes Realtime at Facebook,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD’ 11)*, 2011, S. 1071–1080.

[20] E. A. Brewer, “Towards Robust Distributed Systems,” in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC ’00)*, 2000, S. 7.

[21] J. Broekstra, A. Kampman und F. Van Harmelen, “Sesame: A Generic Architecture for Storing and Querying RDFand RDFSchem,” in *Proceedings of the 1st International Semantic Web Conference on The Semantic Web (ISWC ’02)*, 2002, S. 54–68.

[22] J. Brutlag, “Speed Matters for Google Web Search,” 2009. Verfügbar: http://services.google.com/fh/files/blogs/google_delayexp.pdf. Zugriffsdatum: 16.09.2014.

[23] P. Buneman, “Semistructured Data,” in *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS ’97)*, 1997, S. 117–121.

[24] D. Burleson, “Database Benchmark Wars: What You Need To Know,” 2002. Verfügbar: http://www.zdnet.com/news/database-benchmark-wars-what-you-need-to-know/298320. Zugriffsdatum: 16.09.2014.

[25] S. Bushik, “A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak,” 2012. Verfügbar: http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html. Zugriffsdatum: 16.09.2014.

[26] L. Cagliero, A. Fiori und L. Grimaudo, “A Rule-Based Flickr Tag Recommendation System,” in *Social Media Retrieval*, N. Ramzan, R. Zwol, J.-S. Lee, K. Clüver und X.-S. Hua, (Hg.). London, UK: Springer-Verlag, 2013, S. 169–189.

[27] M. J. Carey und W. A. Muhanna, “The Performance of Multiversion Concurrency Control Algorithms,” in *ACM Transactions on Computer Systems (TOCS)*, Bd. 4, Nr. 4, S. 338–378, 1986.

[28] R. Cattell, “Scalable SQL and NoSQL Data Stores,” in *ACM SIGMOD Record*, Bd. 39, Nr. 4, S. 12–27, 2010.

[29] CeBIT, “Big Data - Datability,” 2014. Verfügbar: http://www.cebit.de/de/news-trends/trends/big-data-datability/big-data-datability.xhtml. Zugriffsdatum: 16.09.2014.

[30] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Ch, A. Fikes und R. Gruber, “Bigtable: A Distributed Storage System for Structured Data,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI ’06)*, 2006, S. 205–218.

[31] H. Chen, R. H. K. Chiang und V. C. Storey, “Business Intelligence and Analytics: From Big Data to Big Impact,” in *MIS Quarterly*, Bd. 36, Nr. 4, S. 1165–1188, 2012.

[32] E. I. Chong, S. Das, G. Eadon und J. Srinivasan, “An Efficient SQL-based RDF Querying Scheme,” in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB ’05)*, 2005, S. 1216–1227.

[33] E. F. Codd, “A relational model of data for large shared data banks,” in *Communications of the ACM*, Bd. 13, Nr. 6, S. 377–387, 1970.

[34] E. F. Codd, “Relational Completeness of Data Base Sublanguages,” in *Database Systems, Prentice Hall and IBM Research Report RJ 987*, San Jose, CA, USA, S. 65–98, 1972.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan und R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC ’10)*, 2010, S. 143–154.

[36] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, S. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver und R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform," in *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB '08)*, 2008, S. 1277–1288.

[37] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, S. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang und D. Woodford, "Spanner: Google's Globally-Distributed Database," in *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI'12)*, 2012, S. 251–264.

[38] B. C. Bin Cui, J. Z. J. Zhao und D. Y. D. Yang, "Exploring Correlated Subspaces for Efficient Query Processing in Sparse Databases," in *IEEE Transactions on Knowledge and Data Engineering*, Bd. 22, Nr. 2, S. 219–233, 2010.

[39] DataStax, "Apache Cassandra 1.1 Documentation," 2014. Verfügbar: http://www.datastax.com/doc-source/pdf/cassandra11.pdf. Zugriffsdatum: 16.09.2014.

[40] DataStax, "Apache Cassandra 2.0 Documentation," 2014. Verfügbar: http://www.datastax.com/documentation/cassandra/2.0/pdf/cassandra20.pdf. Zugriffsdatum: 16.09.2014.

[41] DataStax, "Benchmarking Top NoSQL Databases," 2013. Verfügbar: http://www.datastax.com/resources/whitepapers/benchmarking-top-nosql-databases. Zugriffsdatum: 16.09.2014.

[42] DataStax, "CQL for Cassandra 2.0 Documentation," 2014. Verfügbar: http://www.datastax.com/documentation/cql/3.1/pdf/cql31.pdf. Zugriffsdatum: 16.09.2014.

[43] Datenbanken Online Lexikon, "Skalierbarkeit," 2012. Verfügbar: http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/Skalierbarkeit. Zugriffsdatum: 16.09.2014.

[44] J. Dean und S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Communications of the ACM*, Bd. 51, Nr. 1, S. 107–113, Januar 2008.

[45] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, S. Vosshall und W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, 2007, S. 205–220.

[46] Digg, “Redis at Digg: Story View Counts,” 2011. Verfügbar: http://nosql.mypopescu.com/post/3342598062/redis-at-digg-story-view-counts. Zugriffsdatum: 16.09.2014.

[47] A. Drozdov, “From SQL Server to MongoDB,” 2011. Verfügbar: http://www.mongodb.com/presentations/mongouk-2011/from-sql-server-to-mongodb. Zugriffsdatum: 25.08. 2014.

[48] Drupal, “MongoDB,” 2014. Verfügbar: https://drupal.org/project/mongodb. Zugriffsdatum: 16.09.2014.

[49] S. Edlich, “NoSQL Archive,” 2011. Verfügbar: http://nosql-database.org. Zugriffsdatum: 16.09.2014.

[50] S. Edlich, A. Friedland, J. Hampe und B. Brauer, *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. München: Carl Hanser Verlag, 2010.

[51] F. Eisenführ, T. Langer und M. Weber, *Rationales Entscheiden*, 5. Aufl. Heidelberg: Springer-Verlag, 2010.

[52] R. Elmasri und S. B. Navathe, *Grundlagen von Datenbanksysteme*, 3. Aufl. München: Pearson Studium, 2002.

[53] K. P. Eswaran, J. N. Gray, R. A. Lorie und I. L. Traiger, “The Notions of Consistency and Predicate Locks in a Database System,” in *Communications of the ACM*, 1976, Bd. 19, Nr. 11, S. 624–633.

[54] Facebook Engineering, “Facebook Tech Talk - MySQL and HBase at Scale,” 2011. Verfügbar: https://www.facebook.com/events/232436933489216. Zugriffsdatum: 16.09.2014.

[55] H. Faeskorn-Woyke, B. Bertelsmeier, S. Riemer und E. Bauer, *Datenbanksysteme: Theorie und Praxis mit SQL3, Oracle und MySQL*. München: Pearson Studium, 2007.

[56] E. Farrell, “Auntie on the Couch,” 2010. Verfügbar: http://www.infoq.com/presentations/Auntie-on-the-Couch. Zugriffsdatum: 16.09.2014.

[57] D. C. Faye, O. Curé und G. Blin, “A survey of RDF storage approaches,” in *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, Bd. 15, Nr. 2012, S. 11–35, 2012.

[58] T. Fischermann und G. Hamann, “Big Data: Wer hebt das Datengold?,” 2013. Verfügbar: http://www.zeit.de/2013/02/Big-Data. Zugriffsdatum: 16.09.2014.

[59] Flickr, “Tags,” 2013. Verfügbar: http://www.flickr.com/help/tags. Zugriffsdatum: 16.09.2014.

[60] E. Florenzano, “Nonrelational Databases,” in *Web Operations: Keeping the Data On Time*, J. Allspaw und J. Robbins, (Hg.). Sebastopol, CA, USA: O’Reilly Media, 2010, S. 247–262.

[61] D. Florescu und D. Kossmann, “A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database,” INRIA, Rocquencourt, France, Technical Report, 1999.

[62] P. Gabriel, K. Gaßner und S. Lange, *Das Internet der Dinge - Basis für die IKT-Infrastruktur von morgen*. Berlin: VDI Verlag, 2010.

[63] B. J. Gantz und D. Reinsel, “Extracting Value from Chaos,” International Data Corporation (IDC), Framingham, MA, USA, Technical Report, 2011.

[64] Gartner Inc., “Gartner Says Solving ‘Big Data’ Challenge Involves More Than Just Managing Volumes of Data,” 2011. Verfügbar: http://www.gartner.com/newsroom/id/1731916. Zugriffsdatum: 16.09.2014.

[65] Gartner Inc., “Hype Cycles,” 2013. Verfügbar: http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp. Zugriffsdatum: 16.09.2014.

[66] Gartner Inc., “Top Technology Trends Impacting Information Infrastructure in 2013,” 2013. Verfügbar: http://www.gartner.com/newsroom/id/2359715. Zugriffsdatum: 16.09.2014.

[67] F. Geisler, *Datenbanken - Grundlagen und Design*. Bonn: MITP-Verlag, 2005.

[68] L. George, “HBase Advanced,” 2012. Verfügbar: http://de.slideshare.net/jaxlondon2012/hbase-advanced-lars-george. Zugriffsdatum: 16.09.2014.

[69] L. George, *HBase The Definitive Guide*. Sebastopol, CA, USA: O’Reilly Media, 2011.

[70] S. Gilbert und N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,” in *ACM SIGACT News*, Bd. 33, Nr. 2, S. 51–59, 2002.

[71] S. Ghemawat, H. Gobioff und S. Leung, “The Google File System,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP ’03)*, 2003, S. 29–43.

[72] Google, “Google Analytics,” 2014. Verfügbar: http://www.google.de/intl/de/analytics. Zugriffsdatum: 16.09.2014.

[73] J. Gray und L. Lamport, “Consensus on Transaction Commit,” in *ACM Transactions on Database Systems*, Bd. 31, Nr. 1, S. 133–160, 2006.

[74] C. Gull, *Web-Applikationen entwickeln mit NoSQL*. Haar bei München: Franzis Verlag GmbH, 2011.

[75] N. J. Gunther, *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Berlin: Springer-Verlag, 2007.

[76] T. Härder und E. Rahm, *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Berlin: Springer-Verlag, 1999.

[77] S. Harizopoulos, D. J. Abadi, S. Madden und M. Stonebraker, "OLTP Through the Looking Glass, and What We Found There," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, 2008, S. 981–992.

[78] E. R. Harold und W. S. Means, *XML in a Nutshell*, 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, 2004.

[79] D. Harris, "Facebook trapped in MySQL 'fate worse than death,'" 2011. Verfügbar: http://gigaom.com/2011/07/07/facebook-trapped-in-mysql-fate-worse-than-death. Zugriffsdatum: 16.09.2014.

[80] D. Harris, "Is Stonebraker right? Why SQL isn't the choice du jour for many apps," 2011. Verfügbar: http://gigaom.com/2011/07/21/is-stonebraker-right-why-sql-isnt-the-choice-du-jour-for-many-apps. Zugriffsdatum: 16.09.2014.

[81] S. Harris und N. Gibbins, "3store: Efficient Bulk RDF Storage," in *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03)*, 2003, S. 1–20.

[82] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau und R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: A Facebook Messages Case Study," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, 2014, S. 199–212.

[83] N. Harvey, "MongoSV," 2011. Verfügbar: http://www.nathenharvey.com/blog/2011/12/29/mongosv. Zugriffsdatum: 16.09.2014.

[84] R. Hecht und S. Jablonski, "NoSQL Evaluation: A Use Case Oriented Survey," in *Proceedings of the 2011 International Conference on Cloud and Service Computing (CSC '11)*, 2011, S. 336–341.

[85] E. Hewiit, *Cassandra: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[86] A. Himel, "Building Realtime Insights," 2011. Verfügbar: https://www.facebook.com/note.php?note_id=10150103900258920. Zugriffsdatum: 16.09.2014.

[87] W. Hoffmeister, *Investitionsrechnung und Nutzwertanalyse*, 2. Aufl. Berlin: Berliner Wissenschafts-Verlag, 2008.

[88] B. Holt, *Scaling CouchDB*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[89] Hypertable, "Documentation," 2014. Verfügbar: http://hypertable.com/documentation. Zugriffsdatum: 16.09.2014.

[90] Hypertable, "Secondary Indices Have Arrived," 2012. Verfügbar: http://hypertable.com/blog/secondary_indices_have_arrived. Zugriffsdatum: 16.09.2014.

[91] JSON, "Introducing JSON," 2014. Verfügbar: http://json.org. Zugriffsdatum: 16.09.2014.

[92] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine und D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, 1997, S. 654–663.

[93] Karl Seguin, "The Little Redis Book," 2012. Verfügbar: http://openmymind.net/redis.pdf. Zugriffsdatum: 16.09.2014.

[94] A. Kemper und A. Eickler, *Datenbanksysteme*, 8. Aufl. München: Oldenbourg Wissenschaftsverlag, 2011.

[95] D. Klein, S. Tran-Gia und M. Hartmann, "Big Data," in *Informatik-Spektrum*, Bd. 36, Nr. 3, S. 319–323, Apr. 2013.

[96] J. Krebs, "Project Voldemort: Scaling Simple Storage at LinkedIn," 2009. Verfügbar: http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin. Zugriffsdatum: 16.09.2014.

[97] R. Krikorian, "New Tweets per second record, and how!," 2013. Verfügbar: https://blog.twitter.com/2013/new-tweets-per-second-record-and-how. Zugriffsdatum: 16.09.2014.

[98] R. Krikorian, "Timelines at Scale," 2012. Verfügbar: http://www.infoq.com/presentations/Twitter-Timeline-Scalability. Zugriffsdatum: 16.09.2014.

[99] A. Lakshman und P. Malik, "Cassandra - A Decentralized Structured Storage System," in *ACM SIGOPS Operating Systems Review*, Bd. 44, Nr. 2, S. 35–40, 2010.

[100] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," in *Communication of the ACM*, Bd. 21, Nr. 7, S. 558–565, 1978.

[101] N. Leavitt, “Will NoSQL Databases Live Up to Their Promise?,” in *Computer*, Bd. 43, Nr. 2, S. 12–14, 2010.

[102] S. Leberknight, “Polyglot Persistence,” 2008. Verfügbar: https://www.altamiracorp.com/blog/employee-posts/polyglot-persistence. Zugriffsdatum: 16.09.2014.

[103] J. Lennon, *Beginning CouchDB*. New York, NY, USA: Apress, 2009.

[104] G. Linden, “Make Data Useful,” 2006. Verfügbar: http://de.scribd.com/doc/4970486/Make-Data-Useful-by-Greg-Linden-Amazoncom. Zugriffsdatum: 16.09.2014.

[105] LinkedIn, “Building a terabyte-scale data cycle at LinkedIn with Hadoop and Project Voldemort,” 2009. Verfügbar: http://data.linkedin.com/blog/2009/06/building-a-terabyte-scale-data-cycle-at-linkedin-with-hadoop-and-project-voldemort. Zugriffsdatum: 16.09.2014.

[106] LinkedIn Corporation, “About LinkedIn,” 2014. Verfügbar: http://press.linkedin.com/about. Zugriffsdatum: 16.09.2014.

[107] J. Liu, “Inside Facebook Messages’ Application Server,” 2011. Verfügbar: https://www.facebook.com/notes/facebook-engineering/inside-facebook-messages-application-server/10150162742108920. Zugriffsdatum: 16.09.2014.

[108] P. Lorenzen und O. Schwemmer, *Konstruktive Logik, Ethik und Wissenschaftstheorie*, 2. Aufl. Mannheim: Bibliographisches Institut Mannheim, 1975.

[109] R. Mack, “Building Timeline: Scaling up to hold your life story,” 2012. Verfügbar: https://www.facebook.com/note.php?note_id=10150468255628920. Zugriffsdatum: 16.09.2014.

[110] G. Magnusson, “Project Voldemort at Gilt Groupe: When Failure Isn’t an Option,” 2010. Verfügbar: http://www.infoq.com/presentations/Project-Voldemort-at-Gilt-Groupe. Zugriffsdatum: 16.09.2014.

[111] A. Marcus, “The NoSQL Ecosystem,” in *The Architecture Of Open Source Applications*, A. Brown und G. Wilson, (Hg.). lulu.com, 2012, S. 181–200.

[112] B. Matheny, “Staircar: Redis-powered notifications,” 2011. Verfügbar: http://engineering.tumblr.com/post/7819252942/staircar-redis-powered-notifications. Zugriffsdatum: 16.09.2014.

[113] B. Matheny, “Tumblr Engineering - Open Source - Memcache Top,” 2013. Verfügbar: http://engineering.tumblr.com/post/48701285213/open-source-memcache-top. Zugriffsdatum: 16.09.2014.

[114] A. McAfee und E. Brynjolfsson, “Big Data: The Management Revolution,” in *Harvard Business Review*, Bd. 90, Nr. 10, S. 60–68, 2012.

[115] E. M. McCreight und R. Bayer, “Organization and maintenance of large ordered indexes,” in *Acta Informatica*, Bd. 1, Nr. 3, S. 173–189, 1972.

[116] H. Mehling, “10 things you Need to Know About NoSQL Databases,” 2010. Verfügbar: http://www.databasejournal.com/features/article.php/3905531/10-things-you-Need-to-Know-About-NoSQL-Databases. Zugriffsdatum: 16.09.2014.

[117] Memcached, “memcached - a distributed memory object caching system,” 2014. Verfügbar: http://memcached.org. Zugriffsdatum: 16.09.2014.

[118] D. Mituzas, “Stonebraker trapped in Stonebraker ‘fate worse than death,’” 2011. Verfügbar: http://dom.as/2011/07/08/stonebraker-trapped. Zugriffsdatum: 16.09.2014.

[119] MongoDB, “Craigslist Customer Success with MongoDB,” 2014. Verfügbar: http://www.mongodb.com/customers/craigslist. Zugriffsdatum: 16.09.2014.

[120] MongoDB, “MongoDB Data Modeling and Rails,” 2014. Verfügbar: http://www.mongodb.org/display/DOCS/MongoDB+Data+Modeling+and+Rails. Zugriffsdatum: 16.09.2014.

[121] MongoDB, “Production Deployments,” 2014. Verfügbar: http://www.mongodb.org/about/production-deployments. Zugriffsdatum: 16.09.2014.

[122] MongoDB, “Ruby MongoDB Driver,” 2014. Verfügbar: http://docs.mongodb.org/ecosystem/drivers/ruby. Zugriffsdatum: 16.09.2014.

[123] MongoDB, “The MongoDB 2.6.4 Manual,” 2014. Verfügbar: http://docs.mongodb.org/master/MongoDB-manual.pdf. Zugriffsdatum: 16.09.2014.

[124] L. Montanez, “How We Use MongoDB at Sunlight,” 2010. Verfügbar: http://sunlightfoundation.com/blog/2010/05/28/how-we-use-mongodb-sunlight. Zugriffsdatum: 16.09.2014.

[125] L. Montanez, “Sunlight Labs & MongoDB,” 2010. Verfügbar: http://de.slideshare.net/luigimontanez/sunlight-labs-mongodb-mongodc. Zugriffsdatum: 16.09.2014.

[126] K. Muthukkaruppan, “The Underlying Technology of Messages,” 2010. Verfügbar: https://www.facebook.com/note.php?note_id=454991608919. Zugriffsdatum: 16.09.2014.

[127] MySQL, “Leitfaden zu NoSQL mit MySQL - Das Beste aus beiden Welten nutzen,” 2012. Verfügbar: http://www.mysql.de/why-mysql/white-papers/leitfaden-zu-nosql-mit-mysql. Zugriffsdatum: 16.09.2014.

[128] MySQL, “MySQL 5.7 Reference Manual,” 2014. Verfügbar: http://dev.mysql.com/doc/refman/5.7/en. Zugriffsdatum: 16.09.2014.

[129] MySQL, “MySQL Replikation - Höhere Skalierbarkeit und Verfügbarkeit mit MySQL 5.5,” 2010. Verfügbar: http://www.mysql.de/why-mysql/white-papers/mysql-replikation-mit-mysql-5-5. Zugriffsdatum: 16.09.2014.

[130] MySQL, “MySQL: Leitfaden zu Hochverfügbarkeitslösungen,” 2011. Verfügbar: http://www.mysql.de/why-mysql/white-papers/mysql-leitfaden-zu-hochverfugbarkeitslosungen. Zugriffsdatum: 16.09.2014.

[131] S. Nestorov, S. Abiteboul und R. Motwani, “Inferring Structure in Semistructured Data,” in *ACM SIGMOD Record*, Bd. 26, Nr. 4, S. 39–43, Dezember 1997.

[132] B. C. Neuman, “Scale in Distributed Systems,” in *Readings in Distributed Computing Systems*, M. Singhal und T. L. Casavant, (Hg.). Marina del Rey, CA, USA: IEEE Computer Society, 1994, S. 463–489.

[133] Nike, “Nike+,” 2014. Verfügbar: http://nikeplus.nike.com/plus. Zugriffsdatum: 16.09.2014.

[134] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, S. Saab, D. Stafford, T. Tung und V. Venkataramani, “Scaling Memcache at Facebook,” in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI ’13)*, 2013, S. 385–398.

[135] D. Obasanjo, “Building Scalable Databases: Denormalization, the NoSQL Movement and Digg,” 2009. Verfügbar: http://www.25hoursaday.com/weblog/2009/09/10/BuildingScalableDatabasesDenormalizationTheNoSQLMovementAndDigg.aspx. Zugriffsdatum: 16.09.2014.

[136] Oracle, “Oracle Database Concepts 12c,” 2014. Verfügbar: http://docs.oracle.com/cd/E16655_01/server.121/e17633.pdf. Zugriffsdatum: 16.09.2014.

[137] Oracle, “Oracle NoSQL Database,” 2011. Verfügbar: http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf. Zugriffsdatum: 16.09.2014.

[138] M. T. Özsu und P. Valduriez, *Principles of Distributed Database Systems*, 3. Aufl. New York, NY, USA: Springer-Verlag, 2011.

[139] J. Patel, “Buy It Now! Cassandra at eBay,” 2013. Verfügbar: http://de.slideshare.net/planetcassandra/c-summit-2013-buy-it-now-cassandra-at-ebay-by-jay-patel. Zugriffsdatum: 16.09.2014.

[140] J. Patel, “Cassandra Data Modeling Best Practices, Part 1,” 2012. Verfügbar: http://www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1. Zugriffsdatum: 16.09.2014.

[141] J. Patel, “Cassandra Data Modeling Best Practices, Part 2,” 2012. Verfügbar: http://www.ebaytechblog.com/2012/08/14/cassandra-data-modeling-best-practices-part-2. Zugriffsdatum: 16.09.2014.

[142] P. Pin und S. Chen, “The Entity-Relationship Model - Toward a Unified View of Data,” in *ACM Transactions on Database Systems (TODS)*, Bd. 1, Nr. 1, S. 9–36, 1976.

[143] E. Plugge, T. Hawkins und P. Membrey, *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. New York, NY, USA: Apress, 2010.

[144] PostgreSQL, “PostgreSQL 9.3.5 Documentation,” 2014. Verfügbar: http://www.postgresql.org/docs/9.3/static/index.html. Zugriffsdatum: 16.09.2014.

[145] Project Voldemort, “Project Voldemort - Design,” 2013. Verfügbar: http://www.project-voldemort.com/voldemort/design.html. Zugriffsdatum: 16.09.2014.

[146] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gomez-Villamor, V. Muntes-Mulero und S. Mankovskii, “Solving Big Data Challenges for Enterprise Application Performance Management,” in *Proceedings of the 38th Conference on Very Large Databases (VLDB ’12)*, 2012, S. 1724–1735.

[147] E. Rahm, *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Bonn: Addison-Wesley, 1994.

[148] E. Rahm und G. Vossen, *Web & Datenbanken: Konzepte, Architekturen, Anwendungen*. Heidelberg: dpunkt.verlag, 2003.

[149] Redis, “Command Reference,” 2014. Verfügbar: http://redis.io/commands. Zugriffsdatum: 16.09.2014.

[150] Redis, “Who’s using Redis?,” 2014. Verfügbar: http://redis.io/topics/whos-using-redis. Zugriffsdatum: 16.09.2014.

[151] M. J. Russo, “Redis, from the Ground Up,” 2010. Verfügbar: http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up. Zugriffsdatum: 16.09.2014.

[152] G. Saake und A. Heuer, *Datenbanken: Implementierungstechniken.* Bonn: MITP-Verlag, 1999.

[153] P. J. Sadalage und M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* Crawfordsville, IN, USA: Addison-Wesley, 2012.

[154] L. Schnitzler und M. Hohensee, "Hannover Messe: Wie das Internet der Dinge den Maschinenbau revolutioniert," 2013. Verfügbar: http://www.wiwo.de/unternehmen/industrie/hannover-messe-wie-das-internet-der-dinge-den-maschinenbau-revolutioniert/8023054.html. Zugriffsdatum: 16.09.2014.

[155] B. Schwartz, S. Zaitsev und V. Tkachenko, *High Performance MySQL*, 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, 2012.

[156] J. Seligstein, "See the Messages that Matter," 2011. Verfügbar: https://www.facebook.com/notes/facebook/see-the-messages-that-matter/452288242130. Zugriffsdatum: 16.09.2014.

[157] K. Shvachko, H. Kuang, S. Radia und R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 26st Symposium on Mass Storage Systems and Technologies (MSST '10)*, 2010, S. 1–10.

[158] solid IT, "DB-Engines Ranking," 2014. Verfügbar: http://db-engines.com/en/ranking. Zugriffsdatum: 16.09.2014.

[159] G. Stathis, "Traackr's migration from HBase to MongoDB," 2012. Verfügbar: http://traackr.com/blog/2012/02/traackrs-migration-from-hbase-to-mongodb. Zugriffsdatum: 16.09.2014.

[160] M. Stonebraker, "New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps," 2011. Verfügbar: http://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext. Zugriffsdatum: 16.09.2014.

[161] M. Stonebraker, "The 'NoSQL' Discussion has Nothing to Do With SQL," 2009. Verfügbar: http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext. Zugriffsdatum: 16.09.2014.

[162] M. Stonebraker, "Why Enterprises Are Uninterested in NoSQL," 2010. Verfügbar: http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext. Zugriffsdatum: 16.09.2014.

[163] M. Stonebraker und R. Cattell, "10 Rules for Scalable Performance in Simple Operation' Datastores," in *Communications of the ACM*, Bd. 54, Nr. 6, S. 72–80, Juni 2011.

[164] M. Stonebraker und U. Cetintemel, "'One Size Fits All': An Idea Whose Time Has Come and Gone," in *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, 2005, S. 2–11.

[165] C. Strozzi, "NoSQL Relational Database Management System," 2012. Verfügbar: http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql. Zugriffsdatum: 16.09.2014.

[166] S. Sumanthi und S. Esakkirajan, *Fundamentals of Relational Database Management Systems*. Berlin: Springer-Verlag, 2007.

[167] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman und S. Shah, "Serving Large-scale Batch Computed Data with Project Voldemort," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, 2012, S. 223–236.

[168] Y. Theoharis, V. Christophides und G. Karvounarakis, "Benchmarking Database Representations of RDF/S Stores," in *Proceedings of the 4th International Semantic Web Conference (ISWC '05)*, 2005, S. 685–701.

[169] F. Tian, D. J. DeWitt, J. Chen und C. Zhang, "The Design and Performance Evaluation of Alternative XML Storage Strategies," in *ACM SIGMOD Record*, Bd. 31, Nr. 1, S. 5–10, Mar. 2002.

[170] S. Tiwari, *Professional NoSQL*. Indianapolis, IN, USA: John Wiley & Sons, 2011.

[171] Twitter, "Barack Obama," 2014. Verfügbar: https://twitter.com/BarackObama. Zugriffsdatum: 16.09.2014.

[172] Twitter, "Cassandra at Twitter Today," 2010. Verfügbar: https://blog.twitter.com/2010/cassandra-twitter-today. Zugriffsdatum: 16.09.2014.

[173] Twitter, "Golden Tweets 2012," 2012. Verfügbar: https://2012.twitter.com/en/golden-tweets.html. Zugriffsdatum: 16.09.2014.

[174] Twitter, "Introducing FlockDB," 2010. Verfügbar: https://blog.twitter.com/2010/introducing-flockdb. Zugriffsdatum: 16.09.2014.

[175] TYPO3, "Packages - Overview," 2014. Verfügbar: http://forge.typo3.org/projects/packages. Zugriffsdatum: 16.09.2014.

[176] R. de Virgilio, F. Giunchiglia und L. Tanca, *Semantic Web Information Management: A Model-Based Perspective*. Heidelberg: Springer-Verlag, 2010.

[177] E. van der Vlist, *XML Schema*. Sebastopol, CA, USA: O'Reilly Media, 2002.

[178] P. Walmsley, *XQuery*. Sebastopol, CA, USA: O'Reilly Media, 2007.

[179] P. Warden, *Big Data Glossary*. Sebastopol, CA, USA: O'Reilly Media, 2011.

[180] B. Warfield, "NoSQL is a Premature Optimization," 2011. Verfügbar: http://smoothspan.wordpress.com/2011/07/22/nosql-is-a-premature-optimization. Zugriffsdatum: 16.09.2014.

[181] Wedekind, *Datenbanksysteme Bd. 1*, 3. Aufl. Mannheim: Bibliographisches Institut Mannheim, 1991.

[182] C. Weiss, S. Karras und A. Bernstein, "Hexastore: Sextuple Indexing for Semantic Web Data Management," in *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB '08)*, 2008, S. 1008–1019.

[183] K. Wilkinson, C. Sayers, H. Kuno und D. Reynolds, "Efficient RDF Storage and Retrieval in Jena2," in *Proceedings of the 1st International Workshop on Semantic Web and Databases (SWDB '03)*, 2003, S. 131–150.

[184] B. Yang, W. Qian und A. Zhou, "Using Wide Table to manage web data: a survey," in *Frontiers of Computer Science in China*, Bd. 2, Nr. 3, S. 211–223, August 2008.

[185] YouTube, "YouTube Blog: Here's to eight great years," 2013. Verfügbar: http://youtube-global.blogspot.de/2013/05/heres-to-eight-great-years.html. Zugriffsdatum: 16.09.2014.

[186] Zapier, "MongoDB Integrations," 2014. Verfügbar: https://zapier.com/zapbook/mongodb. Zugriffsdatum: 16.09.2014.

[187] J. Zawodny, "Lessons Learned Migrating 2+ Billion Documents at Craigslist," 2011. Verfügbar: http://de.slideshare.net/jzawodn/lessons-learned-migrating-2-billion-documents-at-craigslist. Zugriffsdatum: 16.09.2014.

[188] J. Zawodny, "Living with SQL and NoSQL at craigslist, a Pragmatic Approach," 2012. Verfügbar: http://de.slideshare.net/jzawodn/living-with-sql-and-nosql-at-craigslist-a-pragmatic-approach. Zugriffsdatum: 16.09.2014.

[189] J. Zawodny, "NoSQL is What?," 2011. Verfügbar: http://blog.zawodny.com/2011/07/23/nosql-is-what. Zugriffsdatum: 16.09.2014.