FH JOANNEUM Gesellschaft mbH

NoSQL Datenbanken

Bachelorarbeit 1

eingereicht am 29.04.2015

Fachhochschul-Studiengang Software Design

Betreuer: DI Johannes Feiner

eingereicht von: Daniel Frech

Personenkennzahl: 1210418058

April 2015

Zusammenfassung

NoSQL Datenbanken bieten neue und alternative Wege zur Speicherung von Daten. Was zeichnet sie aus und welche Vorteile bieten sie? Dieser Frage geht diese Arbeit nach. Der erste Teil gibt dabei einen Überblick über die Grundlagen der Welt der NoSQL Datenbanken. Im zweiten Teil der Arbeit wird in einer Fallstudie eine relationale Datenbank, MySQL, mit einer NoSQL Datenbank, CouchDB, verglichen. Die zentralen Punkte des Vergleichs sind die Fragen, wie Daten erzeugt und geändert werden können, wie sich die Daten auswerten lassen und welche Auswirkung die Datenbank auf die Datenmodellierung hat. Dabei zeigt sich, dass die richtige Wahl der Datenbank keine Leichte ist, denn sowohl relationale, als auch No-SQL Datenbanken bieten zahlreiche Vor-, aber auch Nachteile, die es zu bedenken gilt.

Abstract

NoSQL databases offer new possibilities for persisting data. But what sets them apart? This question is in the center of this thesis. The first part gives an overview of the world of NoSQL databases. The second part consists of a cases study, in which a relationale database, MySQL, is compared with a NoSQL database, CouchDB. The central aspects of this comparison are, how data can be created and changed, how data can be queried and analyzed and what effects the databases have on data modeling. The result is, that the right choice of a database is not an easy one. Relational databases, as well as NoSQL databases, have several advantages, but also disadvantages, that need to be considered.

Inhaltsverzeichnis

Kapitel 1

Einleitung

> The Digital Universe Is Huge – And Growing Exponentially.
> (Turner et al. (2014))

Immer mehr Daten werden heute von Applikationen gespeichert und verwaltet. Das digitale Universum wächst dabei nach Angaben der IDC jährlich um den Faktor 8 (vgl. Gantz & Reinsel (2011), Seite 1). Mit diesem Wachstum ergeben sich neue Anforderungen an die Datenbanken. NoSQL Datenbanken stellen hier eine Alternative zu den bekannten relationalen Datenbanken dar und bieten neue Möglichkeiten Daten zu speichern.

Aber nicht nur das Wachstum der Datenmengen, auch der Erfolg von objektorientierten Programmiersprachen bei Webanwendungen stellen geänderte Anforderungen an die Datenhaltung. Auch hier bieten NoSQL Datenbanken neue Möglichkeiten der Datenhaltung an.

Diese Arbeit beleuchtet die Eigenschaften von NoSQL Datenbanken und untersucht die Unterschiede zu relationalen Datenbanken. Im ersten Teil der Arbeit werden die Grundlagen, die zur Entstehung von NoSQL Datenbanken, sowie die unterschiedlichen Arten von Datenbanken beleuchtet. Zudem werden grundlegende Funktionen, wie die Datenabfrage mittels MapReduce, beschrieben.

Im zweiten Abschnitt werden zwei konkrete Datenbanken, eine relationale und eine NoSQL Datenbank, miteinander verglichen. Drei Fragen stehen dabei im Mittelpunkt des Vergleichs:

1.) Welche Auswirkung hat die Wahl der Datenbank auf die Datenmodellierung für eine Applikation?

2.) Wie können Daten erzeugt und geändert werden?

3.) Auf welche Art und Weise lassen sich gespeicherte Daten abfragen und auswerten?

Ziel ist es, einen grundlegenden Einblick in die Welt der NoSQL Datenbanken zu schaffen und eine Entscheidungshilfe für die Wahl der richtigen Datenbank für zukünftige Projekte zur Verfügung zu stellen.

Kapitel 2

Grundlagen

> In 2013, there were almost as many bits in the Digital Universe as stars in the physical universe.
> (Gantz & Reinsel (2011))

Dieser Teil der Arbeit befasst sich mit den Grundlagen von NoSQL Datenbanken. Beschrieben wird, aus welchen Gründen NoSQL Datenbanken entstanden sind, welche Arten von NoSQL Datenbanken es gibt und mit welchen Grundprinzipien sie funktionieren.

2.1 Relationale Datenbanken

Relationale Datenbanken bilden seit Jahrzehnten das Rückgrat vieler Software Programme. Das aus gutem Grund, bieten sie doch eine ausgereifte und erprobte Technologie zur Datenhaltung an. Gründe für die Popularität relationaler Datenbanken gibt es zahlreiche (vgl. Sadalage & Fowler (2012), Seite 4).

2.1.1 Nebenläufigkeit und Transaktionen

Datenbanken bilden die Grundlage vieler Programme. Mit wachsender Größe steigt damit auch die Anzahl der Zugriff auf die Daten die in den Datenbanken gespeichert werden. Dabei kann nicht verhindert werden, dass Benutzer oder Prozesse zur gleichen Zeit auf die gleichen Daten zugreifen, ob in lesender oder schreibender Form. Besonders im Enterprise Bereich ist dabei die Konsistenz der Daten von zentraler Bedeutung.

Relationale Datenbanken behandeln alle Zugriffe auf Daten durch Transaktionen. Diese bilden eine bewährte Methode, um die Komplexität, die mit Nebenläufigkeiten verbunden ist, einzudämmen. Kapitel 2.4.1 auf Seite 8 geht näher auf die Konsitenz von Daten in relationalen Datenbanken durch Unterstützung des ACID Prinzips ein.

2.1.2 Integrierbarkeit

Enterprise Systeme bestehen in den meisten Fällen aus einer Vielzahl an unterschiedlichen Applikationen, die von unterschiedlichen Teams geschrieben und gewartet werden. Diese Applikationen müssen auch auf die gleichen Daten zugreifen und Updates aus einer Applikation heraus sollen in den anderen Applikationen sichtbar sein. Daraus resultiert in vielen Fällen eine gemeinsame, geteilte Datenbank, in der alle Daten zusammengefasst werden. Die Aufgabe die Daten bei zeitgleichen Zugriffen in einen konsistenten Zustand zu halten wird dabei von der Datenbank übernommen. Relationale Datenbanken eignen sich im Besonderen durch die später beschriebenen Eigenschaften der Transaktionskontrolle und Konsistenz (siehe Kapitel 2.4, Seite 8) für diese Art von geteilten Datenbanken.

2.1.3 Einheitliche Umsetzung

Ein wichtiger Grund für die Popularität relationaler Datenbanken ist die einheitliche Umsetzung über die unterschiedlichen Datenbanken hinweg. Obwohl es natürlich Unterscheide zwischen den einzelnen Datenbanken gibt, so bleibt der Kern doch gleich. So können Entwickler und Datenbankadministratoren, die einmal gelernt haben das relationale Model anzuwenden, dieses Wissen auf einfache Art auf andere Projekte mit anderen relationalen Datenbanken anwenden.

Aber wie bei allem, so sind auch relationale Datenbanken bei weitem nicht perfekt. Mit der Entwicklung objektorientierter Programmiersprachen und der Möglichkeit, heute Daten in zuvor ungeahntem Ausmaß zu erfassen, speichern und zu verarbeiten, treten vermehrt auch die Schwächen relationaler Datenbanken zutage (vgl. Sadalage & Fowler (2012), Seite 5 bis 9).

2.1.4 Objekt-relationale Unverträglichkeit

Die Art und Weise, in der Daten im Speicher während der Programmausführung vorliegen, unterscheidet sich besonders bei objektorientierten Programmiersprachen deutlich von der, wie sie in relationalen Datenbanken gespeichert werden. So sehen relationale Datenbanken die Möglichkeit der Speicherung von verschachtelten Datensätzen oder Listen grundsätzlich nicht vor. Auch wenn Datenbanken wie Oracle eine Möglichkeit zur Speicherung mittels „Nested Tables“ anbieten, so wird selbst von Tom Kyte, einem der bekanntesten Oracle Evangelisten, von der Verwendung abgeraten:

> Me – I'll never use a nested table in a CREATE TABLE statement. You spend all of your time UN-NESTING them to make them useful again! (siehe Kyte (abgerufen am 22.03.2015))

Aus diesem Grund heraus entstanden in den 1990er Jahren objektorientierte Datenbanken. Die damals vorherrschende Meinung, dass diese Art der Datenbanken

relationalen Datenbanken verdrängen würden, trat allerdings nicht ein. Obwohl sich die objektorientierte Programmierung durchsetzte, konnte an der Vormachtstellung der relationalen Datenbanken nicht gerüttelt werden. Geblieben ist trotz alle dem die Schwierigkeit, das Abbild der Daten im Speicher in das relationale Modell zu übertragen.

2.1.5 Verteilbarkeit der Daten

Im Laufe der letzten Jahre wuchs die Datenmenge die Applikationen verarbeiteten und speicherten weiterhin an. Dies führte dazu, dass einzelne Maschinen nicht mehr ausreichten um mit diesen enormen Datenmengen umzugehen. Stattdessen wurde versucht, einen Verbund an Servern als Cluster zusammenzufassen. Die Speicherung und die Verarbeitung der Daten sollte dabei möglichst gleichmäßig auf die Server verteilt werden. Relationale Datenbanken wurden aber nicht im Hinblick auf den Einsatz auf Clustern entwickelt. Einige Anbieter relationaler Datenbanken wie beispielsweise Oracle versuchten, eine entsprechende Funktionalität für relationale Datenbanken anzubieten. Dies gelang aber nur zu Teilen und unter der Voraussetzung, andere zentrale Funktionalitäten der Datenbank auszuhebeln. So können zum Beispiel Daten die in Oracle in Partitionen auf verschiedenen Servern gespeichert werden, nicht in einer SQL Abfrage zusammen abgefragt werden. Kapitel 2.4.2 auf Seite 9 geht auf die Ursachen und Gründe hierfür im Detail ein.

2.2 Der NoSQL Ansatz

> It's better to think of NoSQL as a movement rather than a technology.
> (Sadalage & Fowler (2012), Seite 11)

Tatsächlich sind unter dem Begriff NoSQL zahlreiche, zum Teil sehr unterschiedliche Datenbanken zu finden. Kapitel 2.6 auf Seite 16 beschreibt die wichtigsten,

unterschiedlichen Ansätze.

Aber auch der Begriff NoSQL selbst ist mitunter irreführend. Neben der Diskussion, ob dieser nun tatsächlich für „No“ oder doch „Not only“ SQL steht, ist die Abwendung von Abfragen mittels SQL nicht der zentrale Punkt der NoSQL Datenbanken. Einige Datenbanken wie Cassandra oder MongoDB unterstützen sogar Zugriffe über SQL ähnliche Abfragen. Vielmehr wird versucht, die Schwächen der relationalen Datenbanken auszunützen und eine Alternative zu diesen anzubieten. Es geht also nicht darum, relationale Datenbanken abzulösen und zu ersetzen. In vielen Punkten bieten diese enorme Vorteile und nicht umsonst werden sie seit Jahrzehnten in der Mehrheit der Projekte eingesetzt. Aber es gibt heutzutage auch Anwendungsfälle, in denen ein Einsatz von relationalen Datenbanken auch gravierende Nachteile mit sich bringt. NoSQL soll in seinen unterschiedlichen Ausprägungen hier ein alternatives Werkzeug darstellen, das geänderte Aufgabenstellungen anders und besser lösen kann.

Diese Arbeit betrachtet die Unterschiede zwischen den beiden Arten von Datenbanken und in welchem Fall ein Einsatz von NoSQL Datenbank sinnvoll und lohnenswert sein kann.

2.3 Datenmodelle

Die Datenmodellierung in relationalen Datenbanken kann stark von der in NoSQL Datenbanken abweichen. Wo in relationalen Datenbanken Daten überwiegend in normalisierter Form abgespeichert werden, erlauben NoSQL Datenbanken es zum Beispiel Daten auch in aggregierter Form in sogenannten Dokumenten abzuspeichern (siehe Kaptiel 2.6.2 auf Seite 17). Damit kann ein Dokument einer NoSQL Datenbank mitunter die interne Repräsentation eines Objekts im Speicher widerspiegeln. Ein entsprechendes Umwandeln in normalisierte Form, wie das bei relationalen Datenbanken der Fall ist, entfällt damit.

Welche Auswirkungen das im Detail auf die Datenmodellierung und Datenbankabfragen und Erweiterungsmöglichkeit hat, steht im Zentrum der Fallstudie in Kapitel 3 auf Seite 24.

2.4 Konsistenz

Die Konsistenz von Daten muss bei der Programmierung von Applikationen immer beachtet werden. Sollen doch die in Datenbanken gespeicherten Daten auch bei zahlreichen Zugriffen konsistent bleiben. Allerdings legen nicht alle Applikationen den gleichen Wert auf Konsistenz. Bei der Software für einen Geldautomaten kannen nicht konsistente Daten beispielsweise zu falschen Kontoständen führen. Dieser Zustand muss daher um jeden Preis verhindert werden. Eine Messaging Applikation wie beispielsweise Twitter unterliegt hingegen vermutlich nicht den gleich hohen Ansprüchen in Hinsicht der Datenkonsistenz. Dieses Kapitel beleuchtet die Unterschiede, die es im Hinblick auf die Konsistenz von Daten gibt und wie Datenbanken diesen Anspruch erfüllen können.

2.4.1 ACID

ACID ist ein Acronym, das für „atomicity, consistency, isloation and durability“ steht. Es ist eng mit der Welt der relationalen Datenbanken verbunden und erlaubt die Manipulierung von Datensätzen verschiedener Tabellen in einer einzigen Transaktion (vgl. Sadalage & Fowler (2012), Seite 19 und (vgl. (Tiwari 2011), Seite 113/114).

Atomicity beschreibt, dass Transaktionen atomar stattfinden. Das bedeutet, das entweder eine gesamte Transaktion erfolgreich ist, oder falls nicht, diese rückgängig gemacht wird. Eine Transaktion kann dabei mehrere Befehle und Abfragen beinhalten.

Consistency sagt aus, dass Daten immer in konsistenter Form vorhanden sein müssen, bevor eine weitere Aktion durchgeführt werden kann.

Isloation beschreibt die Sicherheit, dass kein anderer Prozess Daten ändern kann, die gerade in einer Transaktion bearbeitet werden.

Durability bezeichnet den Zustand, dass sämtliche gespeicherten (commited) Daten bei jeglicher Art eines Systemausfalls wieder hergestellt werden können.

Während in Relationalen Datenbanken nach dem ACID Prinzip einen zentralen Bestandteil bilden, ist das bei NoSQL Datenbanken nicht immer der Fall. Gründe dafür sind im CAP Theorem zu finden.

2.4.2 Das CAP-Theorem

Das von Eric Brewer im Jar 2000 erstmals vorgestellte Thoerem besagt, dass es in einem großen, verteilten Datensystem drei Kriterien gibt, die voneinander abhängig sind (vgl. Hewitt (2010), Seite 19 bis 21):

Consistency - Alle Datenbank Benutzer lesen den selben Wert bei der gleichen Abfrage, auch bei parallelen Änderungen.

Availibilty - Alle Datenbank Benutzer können zu jeder Zeit Daten lesen und schreiben.

Partition Tolerance - Die Datenbank kann auf mehrere Maschinen verteilt werden und sie funktioniert auch bei teilweisen Netzwerkausfällen.

Das Theorem besagt weiters, dass in jedem System nur zwei der drei Kriterien mehrheitlich unterstützt werden können. Dies geht einher mit einer bekannten Aussage in der Software Entwicklung: "You can have it good, you can have it fast, you can have it cheap: pick two."(vgl. Hewitt (2010), Seite 20). Abbildung 1 bildet das CAP-Theorem visuell abund zeigt, dass es keine Schnittmenge der drei Bereiche gibt.

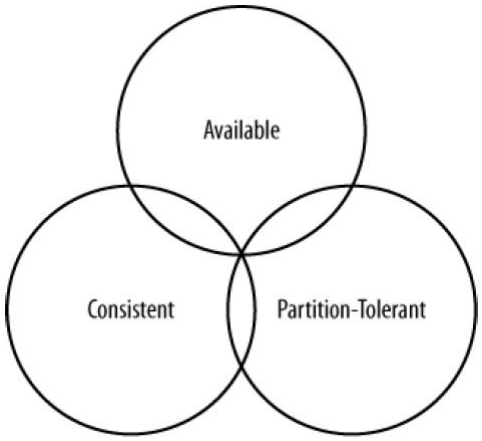

Abbildung 1: Kriterien im CAP-Theorem (Hewitt (2010), Seite 21)

Für Datenbanken hat das umgfangreiche Auswirkungen. Soll das Augenmerk auf der Konsitenz der Daten und deren Verfügbarkeit liegen, müssen Kompromisse bei der Partitionierbarkeit gemacht werden. Das ist genau der Weg, den relationale Datenbanken beschreiten. NoSQL Datenbanken hingegen verändern die Gewichtung. Indem die Konsistenz etwas gelockert wird, ermöglicht man den Datenbanken partitionierbar zu sein. Abbildung 2 zeigt eine Übersicht, wie einige Datenbanken anhand des CAP-Theorems eingeordnet werden können. Die unterschiedlichen NoSQL Datenbanken werden in Kapitel 2.6 ab Seite 16 näher beleuchtet.

Wichtig ist dabei zu beachten, dass die Übergänge fließend sind. In vielen Fällen bieten Datenbanken Einstellungsmöglichkeiten, um selbst entscheiden zu können, welches Kriterium wie stark gewichtet werden soll.

2.5 Verteilte Daten

Bei steigendem Datenaufkommen oder Abfragen an eine Datenbank, gibt es zwei Möglichkeiten der Skalierung: Vertikal oder Horizontal. Vertikal zu skalieren bedeutet, in erster Linie bestehende Hardware durch größere, leistungsstärkere auszutauschen. Irgendwann ist jedoch der Punkt erreicht, an dem dies nicht mehr

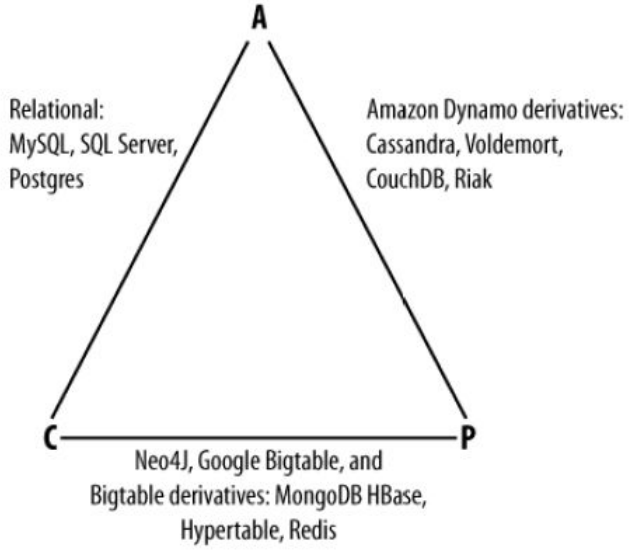

Abbildung 2: Einordnung einiger Datenbanken nach dem CAP-Theorem (Hewitt (2010), Seite 22)

möglich oder einfach zu kostspielig ist. Horizontales Skalieren bedeutet hingegen, mehrere kostengünstigere Server in einem Cluster zu verbinden. Grace Hopper hat diesbezüglich einen treffenden Vergleich beschrieben:

> In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.
> — Grace Hopper, siehe White (2012), Seite 1

Mit dem Aufkommen gestiegener Datenmengen wurde allerdings erkannt, dass sich relationale Datenbanken nicht wirklich für den Einsatz auf Clustern eignen (vgl. Sadalage & Fowler (2012), Seite 8). Obwohl Datenbanken wie Oracle mit Oracle RAC eine entsprechende Funktionalität bieten, wurden relationale Datenbanken nicht im Hinblick darauf entwickelt.

Viele NoSQL Datenbanken hingegen stellen den Aspekt der Partitionierbarkeit der Daten in das Zentrum ihrer Überlegungen. In den Anfängen von den großen der Branche wie Google oder Facebook getrieben, kommen heutzutage mehr und

mehr Firmen an die Kapazitätsgrenzen der Datenmengen, die mit einem einzigen Server verarbeitet werden können. Sie stehen heute in zunehmenden Maße vor den gleichen Problemen wie beispielsweise Google damals. Mit dem entscheidenden Unterschied, dass sie von den Erfahrungen die damals gemacht wurden und aus denen NoSQL hervorging, profitieren können. Im Folgenden werden die unterschiedlichen Möglichkeiten der Verteilung von Daten vorgestellt.

2.5.1 Einzel-Server Lösungen

Die einfachste Form der Verteilung von Daten ist sie erst gar nicht zu verteilen. Alle Lese- und Schreibanfragen an eine Datenbank werden damit von einem Server bearbeitet. Wo immer es möglich ist, ist dieser Weg zu präferieren, werden damit sämtliche Probleme, die mit der Verteilung der Daten entstehen, umgangen. Dieser Weg sollte selbst für NoSQL Datenbanken, die auf die Verteilung hin optimiert wurden, der bevorzugte Weg sein, sofern die Datenmengen und das Applikationsmodell dies zulassen (vgl. Sadalage & Fowler (2012), Seite 38).

2.5.2 Sharding

Sharding bezeichnet den Prozess, Daten auf mehrere Server zu verteilen. Indem nur ein Teil der Daten auf jedem Server gespeichert wird, kann eine größere Datenmenge gespeichert und verarbeitet werden, ohne dafür einen leistungsstärkeren Server zu beschaffen. Lediglich mehrere, leistungsschwächere Maschinen werden benötigt (vgl. Chodorow (2013), Seite 231). Abbildung 3 zeigt die Verteilung von Daten mittels Sharding.

Ein Beispiel für Sharding wäre, Kundendaten manuell auf unterschiedlichen Servern zu verteilen. Dabei könnten Daten von Kunden aus Europa auf einem Server in Europa und Daten von Kunden in Asien auf einem Server in Asien gespeichert werden. Diese Form des Shardings ist mit fast allen Datenbanken möglich, da die Verantwortung für die Verteilung der Daten nicht in der Datenbank, sondern der

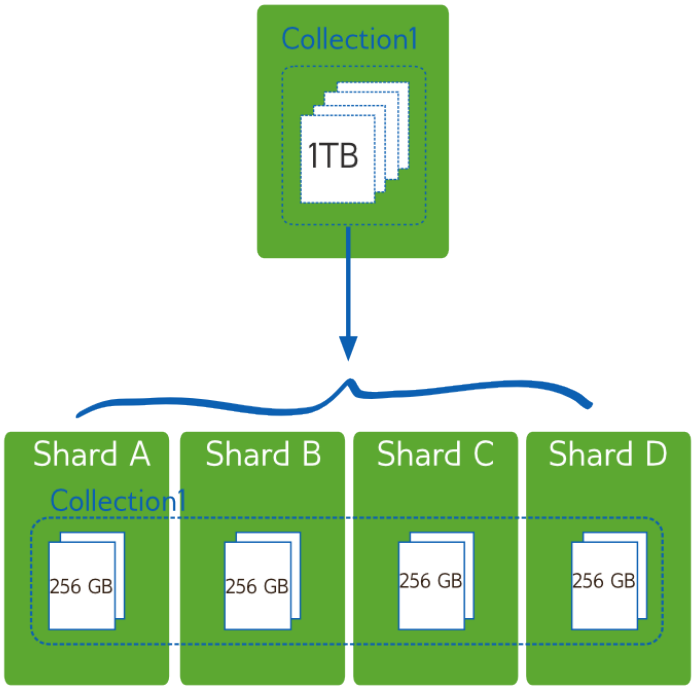

Abbildung 3: Sharding (MongoDB (abgerufen am 02.04.2015))

aufrufenden Applikation liegt. Dieser Vorgang erschwert jedoch das Programmiermodel, da auch bei Abfragen die unterschiedlichen Shards berücksichtigt werden müssen. Hinzu kommt, dass eine Änderung des Shardings auch eine Änderung in der Programmlogik in der Applikation nach sich zieht (vgl. Sadalage & Fowler (2012), Seite 39).

NoSQL Datenbanken unterstützen aber zudem in vielen Fällen das Automatisierte Sharding. Die Architektur in der die Daten gehalten werden, ist damit von der Applikationslogik getrennt und das Sharding kann unabhängig von der Programmlogik geändert werden. Die Abbildungen 5 und 4 zeigen, wie dieser Prozess in MongoDB erfolgt. Die Applikation ruft die Datenbank immer an einer zentralen Stelle auf. Die Verteilung der Daten auf den einzelnen Shards erfolgt automatisch, anhand der in der Datenbank konfigurierbaren Regeln.

Während Sharding eine positive Auswirkung auf die Lese- und Schreibgeschwindigkeit hat, bietet es keinen allzu großen Nutzen im Hinblick auf die Ausfallsicherheit. Würde ein Node eines Clusters ausfallen, stünden die Daten der anderen

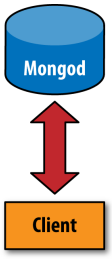

Abbildung 4: MongoDB ohne Sharding (Chodorow (2013), Seite 233)

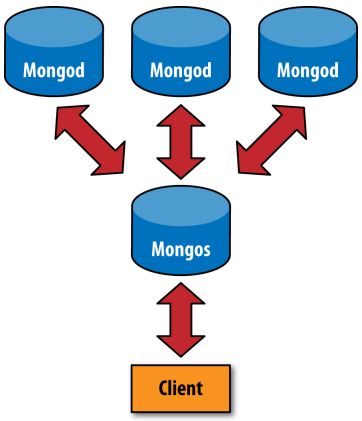

Abbildung 5: MongoDB mit Sharding (Chodorow (2013), Seite 233)

Nodes zwar noch zu Verfügung. Die Daten des aufgefallenen Nodes würden jedoch trotzdem fehlen. Anstatt eines kompletten Ausfalls, würden also lediglich bestimmte Teile ausfallen, bzw. fehlen. Um dieses Problem zu umgehen, bieten sich weitere Formen der Verteilung an.

2.5.3 Master-Slave Replikation

Bei der Master-Slave Replikation werden alle Daten auf mehrere Server verteilt. Ein Server wird dabei als Master festgesetzt. Alle Schreibvorgänge müssen in Folge über diesen Master Server abgewickelt werden. Lesevorgänge können von allen Server beantwortet werden.

Diese Art der Verteilung bietet sich vor allem bei Daten an, auf die überwiegend in lesender Form zugegriffen wird. Überwiegen jedoch schreibende Zugriffe, ist der Vorteil bei weitem nicht so hoch, müssen doch alle Zugriffe weiterhin über

den Master Server laufen.

Ein weiterer Vorteil dieser Form der Verteilung ist, dass auch bei einem Ausfall einer oder mehrere Server auf sämtliche Daten zugegriffen werden kann, sofern noch zumindest ein Server verfügbar ist. Zudem kann im Falle des Ausfalls des Master Servers, ein anderer Server die Aufgaben des Master Servers übernehmen. Die Slave-Server können somit auch lediglich als eine Art Hot-Backup fungieren, womit die Vorteile der Single-Server Datenhaltung erhalten blieben, bei gleichzeitiger erhöhter Ausfallssicherheit (vgl. Sadalage & Fowler (2012), Seite 40,41).

Der Nachteil dieser Art der Verteilung liegt klar bei der Konsistenz der Daten. Nach einem Schreibprozess am Master Server dauert es eine bestimmte Zeitspanne, bis wieder alle Server im Cluster den gleichen Stand haben und konsistent sind. Um diesem Problem entgegen zu treten bieten unterschiedliche Datenbanken unterschiedliche Lösungen an.

2.5.4 Peer-to-Peer Replikation

Während in der zuvor beschriebenen Master-Slave Replikation die Fähigkeit, Lesezugriffe zu verarbeiten erhöht wird, bietet es nur eine geringe Verbesserung bei Schreibzugriffen. Der Master-Server stellt damit weiterhin einen Flaschenhals und Single-Point-of-Failure dar (vgl. Sadalage & Fowler (2012), Seite 42). Im Fall von Daten, auf die oft in schreibender Form zugegriffen wird, bietet sich daher die Peer-to-Peer Replikation an.

Ähnlich wie zuvor werden auch in diesem Fall alle Daten auf alle Server verteilt. Allerdings kommunizieren die einzelnen Nodes Änderungen untereinander und erlauben so auch schreibende Zugriffe über alle Nodes. Damit ist es auch möglich, zusätzliche Server schnell dem Cluster hinzuzufügen.

Die Vorteile, die diese Art der Verteilung ganz offensichtlich bietet, haben allerdings auch eine Schattenseite. Erneut geht die erhöhte Ausfallsicherheit und

Performance auf Kosten der Konsistenz der Daten. Die Handhabung inkonsistenter Daten ist damit bei der Architektur der Applikation mit zu berücksichtigen, wird doch die Verfügbarkeit der Daten gegen deren Konsistenz eingetauscht.

2.6 Kategorien von NoSQL Datenbanken

Der Begriff NoSQL vereint zahlreiche Datenbanken, die sich untereinander zum Teil sehr deutlich unterscheiden. Die üblichsten Arten von NoSQL Datenbanken werden im Folgenden vorgestellt. Allerdings lassen sich viele NoSQL Datenbanken nicht eindeutig in eine der vorgestellten Gruppen einordnen. So ist beispielsweise OrientDB eine Graphdatenbank, die zugleich auch Dokumente speichert.

2.6.1 Key-Value Datenbanken

Key-Value Datenbanken bilden die einfachste Form der NoSQL Datenbanken. Sie speichern jeweils Key-Value Paare und sind auf die Abfragen der gespeicherten Schlüssel optimiert. Sie entsprechen damit in etwa den in vielen Programmiersprachen bekannten Hast-Tabellen. Manche Key-Value Datenbanken wie Redis bieten die Möglichkeit, komplexere Datentypen, wie zum Beispiel Listen, als Wert zu speichern. Dies ist allerdings keine Voraussetzung für Key-Value Datenbanken (vgl. Redmond & Wilson (2012), Seite 4). Bekannte Key-Value Datenbanken sind Redis, Riak, Memchached DB oder Amazon DynamoDB.

Der Vorteil der Key-Value Datenbanken liegt in ihrer Einfachheit und der damit verbundenen Geschwindigkeit. Damit eignen sie sich hervorragend für in-Memory Caching. Als Nachteil ist zu sehen, das Key-Value Datenbanken keine Transaktionen unterstützen, wie es relationale Datenbanken tun. Lediglich das Schreiben eines Key-Value Paares ist als Atomare Transaktion zu sehen. Bei in-Memory Datenbanken wie Memcached DB gehen bei einem Ausfalls des Servers zudem die Daten verloren. Dies ist bei einer Implementierung dieser Datenbanken zu

beachten.

Daraus abgeleitet ergeben sich als mögliche Anwendungsfälle für Key-Value Datenbanken zum Beispiel das Speichern von Session Informationen bei Webanwendungen. Aber auch bei e-Commerce Anwendungen kann es sich anbieten, den Inhalt des Einkaufswagens von Kunden in Key-Value Datenbanken zu speichern. In beiden Fällen ist auch der Verlust der Daten im Falle eines Server-Ausfalls zu verkraften und wiegt die Vorteile bei weitem auf.

2.6.2 Dokumentenorientierte Datenbanken

Dokumentenorientiere Datebanken speichern Dokumente die aus einer Menge aus Key-Value Paaren bestehen. Dokumente können dabei verschachtelte Strukturen aufweisen und somit zum Beispiel auch weitere Dokumente beinhalten. Sie bestechen mit einer großen Flexibilität und bieten eine Vielzahl an Anwendungsmöglichkeiten (vgl. Redmond & Wilson (2012), Seite 6).

Im Unterschied zu relationalen Datenbanken erlauben dokumentenorientierte Datenbanken, bereits aggregierte Daten zu speichern. Ein Dokument kann damit der Struktur von Daten im Speicher entsprechen und ermöglicht so unter Umständen den Zugriff auf die gesamten Daten, beispielsweise eine Benutzers, mit einer einzigen Abfrage. Während in Relationalen Datenbanken in diesem Fall mehrere Tabellen miteinander verbunden werden müssen um das Ergebnis zu erreichen, liegen in dokumentenorientierten Datenbanken diese bereits aggregiert vor. Dies hat allerdings umfangreiche Auswirkungen auf die Datenmodellierung, wie die Fallstudie dieser Arbeit zeigen wird.

Bekannte dokumentenorientierte Datenbanken sind unter anderem MongoDB, CouchDB, Couchbase, oder aber auch Lotus Notes. Dies zeigt, dass das Konzept der dokumentenorientierten Datenbanken eigentlich nicht neu ist, sondern bereits früher eingesetzt wurde.

2.6.3 Spaltenorientierte Datenbanken

Relationale Datenbanken sind zeilenorientiert, das bedeutet, dass alle Informationen eines Datensatzes in einer Zeile gespeichert werden, in diesem Fall in einer Tabelle. Spaltenorientiere Datenbanken hingegen speichern die Werte von Spalten gemeinsam, in sogenannten Spaltenfamilien (Column-Families) (vgl. Redmond & Wilson (2012), Seite 5). Während einzelne Spaltenfamilien mit Tabellen aus Relationalen Datenbanken verglichen werden können, unterscheiden sie sich darin, dass Datensätze (oder Zeilen) in Spaltenorientierten Datenbanken nicht über die selbe Anzahl an Spalten verfügen müssen (vgl. Sadalage & Fowler (2012), Seite 101). Abbildung 6 zeigt ein Beispiel einer Spaltenfamilie.

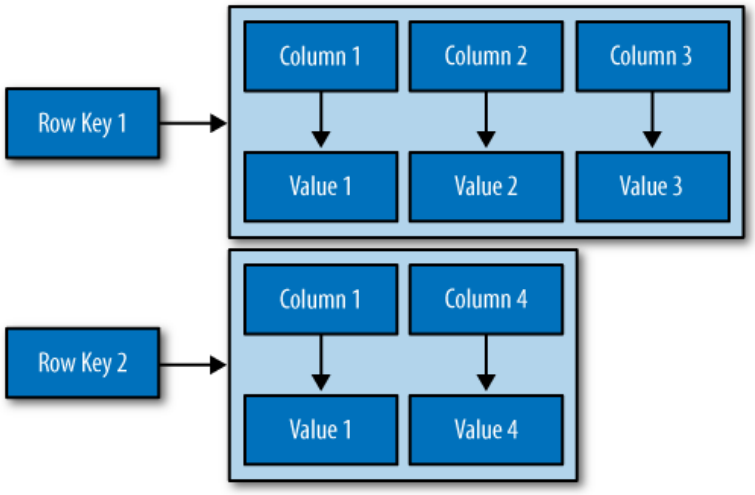

Abbildung 6: Eine Spaltenfamilie (Redmond & Wilson (2012), Seite 44)

Der Vorteil der sich daraus ergibt ist eine höhere Flexibilität bei der Modellierung der Daten. Zusätzliche Spalten können jederzeit hinzugefügt werden, ohne bestehende Daten ändern zu müssen. Zudem bieten Spaltenorientierte Datenbanken einen großen Vorteil, wenn Teilmengen von Datensätzen einer großen Datenmenge verarbeitet werden, wie das beispielsweise in Data-Warehouse Anwendungen der Fall ist (vgl. Vaish (2013), seite 27).

Spaltenorientierte Datenbanken sind eng mit dem Begriff Big Table verbunden und wurden von den großen Anbietern wie Google oder Amazon entwickelt. Zu

den bekanntesten Vertretern dieser Kategorie zählen HBase, Cassandra oder Hypertable.

2.6.4 Graphdatenbanken

Bei Graphdatenbanken stechen bei der Speicherung von stark vernetzten Daten hervor. Eine Datenbank besteht dabei aus Knoten (Nodes) und den Beziehungen zwischen den Knoten. Sowohl die Knoten, als auch die Beziehungen können Key-Value Paare zur Beschreibung speichern. Abbildung 7 zeigt in der Datenbank Neo4j gespeicherte Daten in einem Graphen.

Abbildung 7: Beispiel Graph in Neo4j

Der große Vorteil von Graphdatenbanken kommt bei großen Datenmengen zum Tragen. Während bei Relationale Datenbanken die Geschwindigkeit bei join-intensiven Abfragen mit Zunahme der Datenmenge sinkt, bleibt sie bei Graphdatenbanken relativ konstant. Das ist darauf zurückzuführen, das Abfragen immer auf eine bestimmte Region des Graphen begrenzt sind. Die Geschwindigkeit der Abfrage ist damit nur von der Größe des Teils des Graphen abhängig der abge-

fragt wird und nicht vom gesamten Graphen (vgl. Robinson et al. (2013), Seite 8).

Die verbreitetsten Graphdatenbanken sind Neo4j und OrientDB.

2.7 Abfragen mit MapReduce

Während relationale Datenbanken ad-hoc Abfragen mittels SQL unterstützen, bieten viele NoSQL Datenbanken diese Möglichkeit nicht. Anstatt SQL wird in diesen Fällen MapReduce eingesetzt, um vordefinierte Abfragen abzusetzen und Daten zu analysieren. Vorweg ist festzustellen, dass MapReduce nicht in allen Datenbanken gleich implementiert ist. Die folgende Beschreibung richtet sich im wesentlichen an die Beschreibung von MapReduce von Google und Hadoop.

MapReduce ist ein Programmiermodel, dass es erlaubt, große Datenmengen zu verarbeiten und zu analysieren. Abfragen werden dabei in eine *map* und eine *reduce* Funktion aufgeteilt, die vom darunterliegenden System automatisch auf mehrere Server verteilt und parallel verarbeitet wird. Treten dabei Hardwarefehler oder Netzwerkprobleme bei der Kommunikation unter den Servern auf, wird die gesamte Verarbeitung dadurch nicht unterbrochen. Das Programmiermodell ist damit leicht einzusetzen und abstrahiert die Komplexität die bei Abfragen auf verteilten Systemen auftreten (vgl. Dean & Ghemawat (2008)).

Neben der Verteilung der Abfragen auf mehrere Server ist der Vorteil von MapReduce auch in den Zugriffszeiten von Festplatten begründet. Diese sind im Laufe der Jahre nicht so schnell gestiegen wie die Übertragungsraten von Daten. Benötigt eine Abfrage Zugriff auf verteilt gespeicherte Daten - wie das in relationalen Datenbanken bei Abfragen mit Joins der Fall ist - kann es bei großen Datenmengen vorteilhafter sein, alle Daten sequentiell zu lesen. Die Abfragegeschwindigkeit würde sich damit an der Übertragungsrate orientieren (vgl. White (2012), Seite 5).

Abbildung 8 zeigt die wesentlichen Unterschiede zwischen relationalen Datenbanken und Abfragen mit MapReduce.

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Abbildung 8: MapReduce im Vergleich zu Relationalen Datebanken (White (2012), Seite 5)

2.7.1 Ablauf von Abfragen

Der Ablauf wird anhand des Artikels zu MapReduce Abfragen von Jeff Dean und Sanjay Ghemawat erläutert (vgl.Dean & Ghemawat (2008)). Aufgabe ist es dabei, die Häufigkeit von Wörtern in einer Sammlung von Dokumenten zu finden. Wie vorhin erwähnt, wird die Abfrage in zwei Funktionen aufgeteilt. Die *map* Funktion würde dabei im Pseudocode wie folgt aussehen:

```
1 map(String key, String value):
2   // key: document name
3   // value: document contents
4
5   for each word w in value:
6     EmitIntermediate(w, 1);
```

Diese Funktion wird zur Ausführung auf mehrere Server verteilt. Sämtliche Dokumente werden durchlaufen und jedes Wort liefert als Ausgabe ein Key-Value Paar, das aus dem Wort als Key und der Zahl 1 als Value besteht. Das daraus resultierende Ergebnis könnte wie wie folgt aussehen (hier im JSON Format dargestellt):

```
{"Map", 1
 "Reduce, 1,
 "anhand", 1,
 "eines", 1,
 "Beispiels", 1,
 "Map", 1,
 "Reduce", 1,
 ...
}
```

Basierend auf diesen Ergebnissen, würde darauf folgende die Reduce Funktion ausgeführt werden.

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts

  int result = 0;

  for each value in values:
    result += value;

  Emit(result);
```

Diese kombiniert die Ergebnisse und liefert aggregierte Daten, in diesem Fall die Anzahl der Wörter als Endergebnis aus:

```
{"Map", 2
 "Reduce", 2,
 "anhand", 1,
 "eines", 1,
 ...
}
```

Die Reduce Funktion kann erneut auf mehrere Server verteilt werden. Sie ist allerdings nicht zwingend vorgeschrieben. Wird keine Reduce Funktion angegeben, bilden die Ergebnisse der Map Funktion das Endergebnis. Abbildung 9 zeigt einen

schemtischen Überblick der Abläufe beim Aufruf einer MapReduce Funtkion.

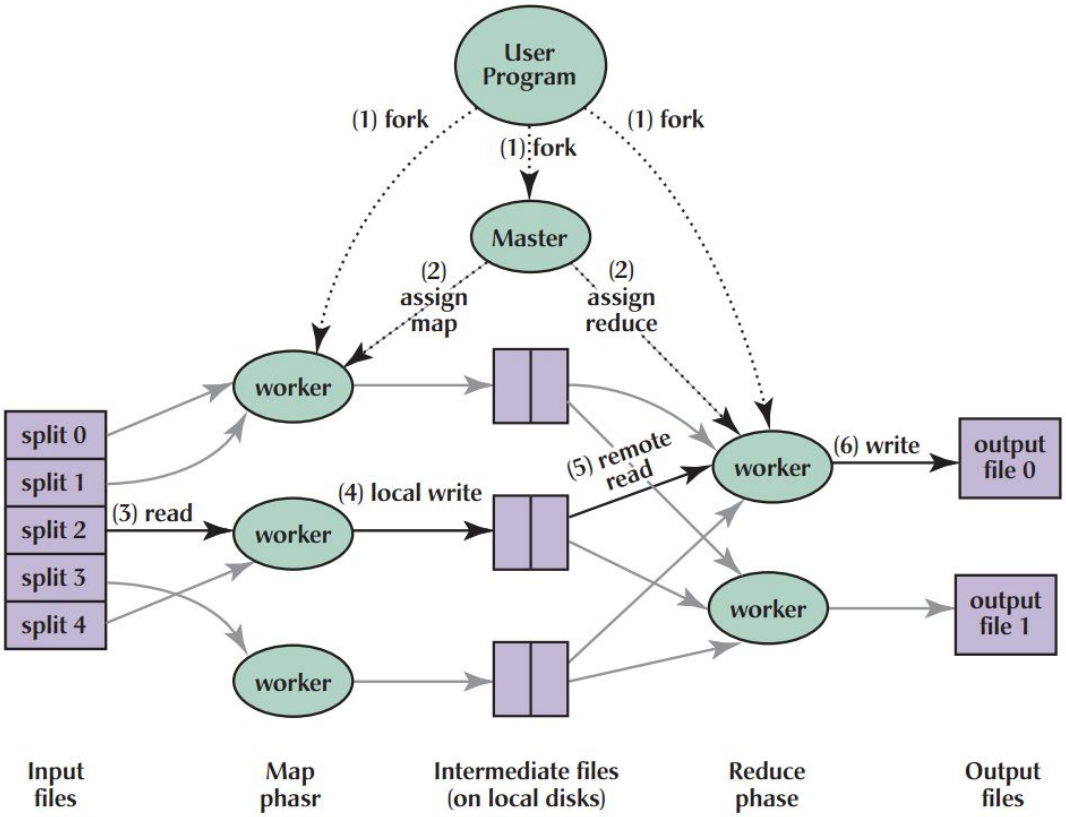

Abbildung 9: Ablauf bei MapReduce Abfragen (Dean & Ghemawat (2008)

2.8 Zusammenfassung

Dieser Teil der Arbeit hat die Grundlagen von NoSQL Datenbanken beleuchtet. Dabei ist deutlich geworden, dass sich NoSQL Datenbanken in vielen Punkten von relationalen Datenbanken unterscheiden. Aber auch, dass der Begriff NoSQL für eine Vielzahl, zum Teil sehr unterschiedliche Datenbanken steht. Im folgenden Abschnitt wird in einer Fallstudie ein Vertreter der relationalen Datenbanken, MySQL, mit einem aus der Welt der NoSQL Datenbanken, CouchDB, verglichen.

Kapitel 3

Fallstudie

3.1 Einleitung

Inwiefern kann eine NoSQL Datenbank einen Mehrwert und sinnvolle Alternative zu einer relationalen Datenbank darstellen? Diese Frage steht im Mittelpunkt der folgenden Fallstudie.

Wie in den letzten Kapiteln gezeigt, stehen unter dem Begriff NoSQL eine Vielzahl an zum Teil sehr unterschiedlichen Datenbanken zur Verfügung. Eine abschließende Beantwortung dieser Frage ist nicht nur im Rahmen dieser Arbeit nicht möglich, sondern je nach Anwendungsfall und Datenbank erneut zu beantworten. Dennoch sollen anhand der Beispiele die Unterschiede, Stärken und Schwächen zu relationalen Datenbanken gezeigt werden. Ziel ist es, damit einen konkreten Ansatz zur Bewertung und den Einsatzmöglichkeiten von NoSQL Datenbanken zu erhalten und folgend zentrale Fragen zu beantwortet und zu verglichen:

1.) Welche Auswirkung hat die Wahl der Datenbank auf die Datenmodellierung für eine Applikation?

2.) Wie können Daten erzeugt und geändert werden?

3.) Auf welche Art und Weise lassen sich gespeicherte Daten abfragen und auswerten?

Für diesen Anwendungsfall wird eine Applikation herangezogen, die Skigebiete speichern und anzeigen soll. Skigebiete weltweit sollen in einer Datenbank erfasst werden und Details zum Land und der Region in der sie beheimatet sind beinhalten. Zusätzlich sollen Skilifte und Übersichtskarten der Skigebiete erfasst werden und Benutzer sollen die Möglichkeit haben, Kommentare für Skigebiete hinterlegen zu können.

Als relationale Datenbank wird dabei MySQL in der Version 5.6.16 eingesetzt, für die NoSQL Datenbanken kommt CouchDB als dokumentenorientierte Datenbank in der Version 1.6.1 zum Einsatz. Der folgende Abschnitt gibt einen kurzen Überblick über CouchDB.

3.2 Apache CouchDB

Startet man den CouchDB Server, wird man mit der Nachricht „Apache CouchDB has started. Time to relax.“ begrüßt. Und tatsächlich versucht man mit CouchDB einen einfachen und klar nachvollziehbaren Weg zu gehen, bei dem die Produktivität beim Entwickeln im Mittelpunkt steht (vgl. Anderson et al. (2010), Seite 3). Zudem ist die in Erlang implementierte und 2005 erstmals veröffentlichte Datenbank auf Anwendungen im Web ausgerichtet. Bekannte Schwierigkeiten bei Web-Anwendungen, wie zum Beispiel Netzwerkschwankungen, werden dabei gezielt abgefangen. CouchDB bietet dabei eine Robustheit, die seinesgleichen sucht (vgl. Redmond & Wilson (2012), Seite 177).

Datenhaltung

CouchDB zählt zur Gruppe der dokumentenorientierten NoSQL Datenbanken. Die Daten werden in Form von Dokumenten im JSON Format gespeichert. Je-

des Dokument besteht aus einer beliebigen Anzahl von Paaren, die jeweils aus einem Schlüssel und einem Wert bestehen. Ein einzelnes Paar wird dabei als Feld bezeichnet. Jeder Wert-Teil eines Feldes kann beliebige Daten wie Arrays oder weitere Dictionaries enthalten. Ein Dokument kann in CouchDB zum Beispiel wie folgt aussehen:

```
{
    "_id": "00a354787f89c0ef2e10e88a0c0001g2",
    "_rev": "1-2628a55ac8c3abfffc1f6e30c9949fd3",
    "doc_type": "student",
    "name": {
        "first_name": "Daniel",
        "last_name": "Frech"
    },
    "birth_place": "Graz",
    "interests": [
        "skiing",
        "biking"
    ]
}
```

Im Gegensatz zu anderen dokumentenorientierten NoSQL Datenbanken, bietet CouchDB keine Möglichkeit Dokumente gleicher Art zusammenzufassen, wie es beispielsweise MongoDB mit Collections unterstützt. Im Rahmen dieser Arbeit wird daher jedem Dokument ein Feld mit dem Schlüssel „doc_type“ und der Dokumentenart als Wert hinzugefügt. Somit können in Folge Dokumente unterschieden und gruppiert werden.

Felder die mit einem Unterstrich beginnen, haben eine besondere Bedeutung in CouchDB. Der Wert für das _id Feld wird bei der Anlage von CouchDB vergeben, sofern dieser nicht angegeben wird.

Das Feld mit dem Schlüssel _rev wird immer von CouchDB vergeben und steht für die Revisionsnummer des Dokuments. Wird ein Dokument in CouchDB überarbeitet, wird das bestehende Dokument in der Datenbank nicht verändert, son-

dern ein neues Dokument mit der fortlaufenden Revisionsnummer angelegt. Auch beim Löschen eines Dokuments wird ein neues, leeres Dokument mit der nächsten Revisionsnummer angelegt. Entsprechend sind bei jeder UPDATE oder DELETE Operation immer die Werte des _id und _rev Feldes des Dokuments anzugeben. Ohne diese Information ist eine Änderung nicht möglich (vgl.Redmond & Wilson (2012), Seite 180).

Zugriff auf die Datenbank

Alle Zugriffe auf die Datenbank erfolgen über REST Aufrufe. Damit werden keinerlei spezifische Datenbanktreiber benötigt. Für die in der Fallstudie angeführten Zugriffe wird das Tool curl verwendet, mit dem aus der Kommandozeile direkt auf die API von CouchDB zugegriffen werden kann. Mit folgendem Befehl wird in CouchDB zum Beispiel eine neue Datenbank mit dem Namen „skiing“ erzeugt:

```
curl –X PUT http://127.0.0.1:5984/skiing
```

Sofern die Datenbank noch nicht existiert legt CouchDB diese an und liefert danach folgende Antwort:

```
{"ok":true}
```

Abfragen

Daten können in CouchDB nur über Views abgefragt werden, ein direkter Zugriff auf die Dokumente und ad-hoc Abfragen werden nicht unterstützt. Views sind, im Gegensatz zu relationalen Datenbanken, keine SQL sondern Map-Reduce Abfragen, die ebenfalls in Form von Dokumenten in der Datenbank gespeichert werden. Die Funktionsweise der Abfragen mittels Views wird in der Fallstudie im Detail erläutert.

Datenverteilung und Konsistenz

CouchDB unterstützt keine Transaktionen oder Sperren von Dokumenten zum Schreiben. Anhand der übergebenen Revisionsnummer des Dokuments wird überprüft, ob der Änderungsvorgang auch tatsächlich am aktuellen Dokument geschieht. Ansonsten wird je Schreibvorgang wie in Kapitel 3.2 auf Seite 25 beschrieben ein neues Dokument mit neuer Revisionsnummer angelegt.

Diese Tatsache ist vor allem bei der Nutzung der Option zum Replizieren der Daten zu beachten. CouchDB unterstützt die Verteilung der Daten in Form der auf Seite 15 beschriebenen Peer-to-Peer Replizierung. Dabei verfügen alle eingebundenen Server über alle Daten und stehen uneingeschränkt für Lese- und Schreibzugriffe zur Verfügung (vgl. Redmond & Wilson (2012), Seite 212). Daten in CouchDB sind damit hoch verfügbar und partitionierbar, allerdings müssen, dem CAP Theorem auf Seite 9 folgend, Einschränkungen im Hinblick auf die Konsistenz gemacht werden. Eine detaillierte Beschreibung des Konsistenzverhaltens und der Handhabung von Konflikten in CouchDB ist im Buch Anderson et al. (2010) ab Seite 11 zu finden.

Sonstiges

CouchDB bietet eine Admin-Oberfläche mit dem Namen „Futon“ an, über die Datenbanken konfiguriert und deren Dokumente angezeigt werden können. Diese ist nach dem Start des CouchDB Servers unter der Adresse `http://server_address:5984` erreichbar. Abbildung 10 zeigt das Beispieldokument in der Admin Oberfläche an.

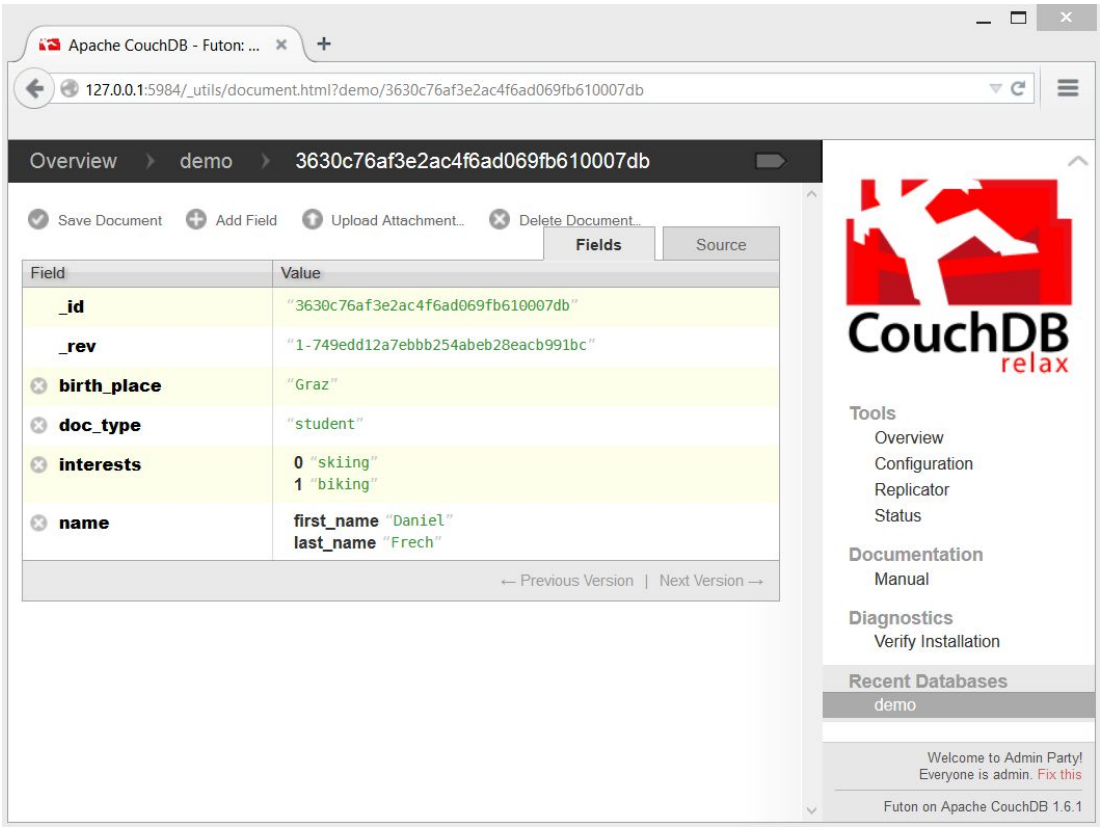

Abbildung 10: Admin-Oberfläche Futon

3.3 Datenmodellierung

Die Überlegung, auf welche Art und Weise Daten einer Applikation in einer Datenbank gespeichert werden, ist von großer Bedeutung für den späteren Erfolg einer Anwendung. NoSQL Datenbanken unterscheiden sich in grundlegenden Eigenschaften von relationalen Datenbanken. Dieses Kapitel untersucht, inwiefern sich diese Unterschiede auf die Datenmodellierung auswirken.

3.3.1 MySQL

Um die Daten für die Applikation in diesem Anwendungsfall zu speichern, gibt es unterschiedliche Möglichkeiten. Abbildung 11 zeigt eine davon.

Aus dem Diagramm können folgende Schlussfolgerungen gezogen werden:

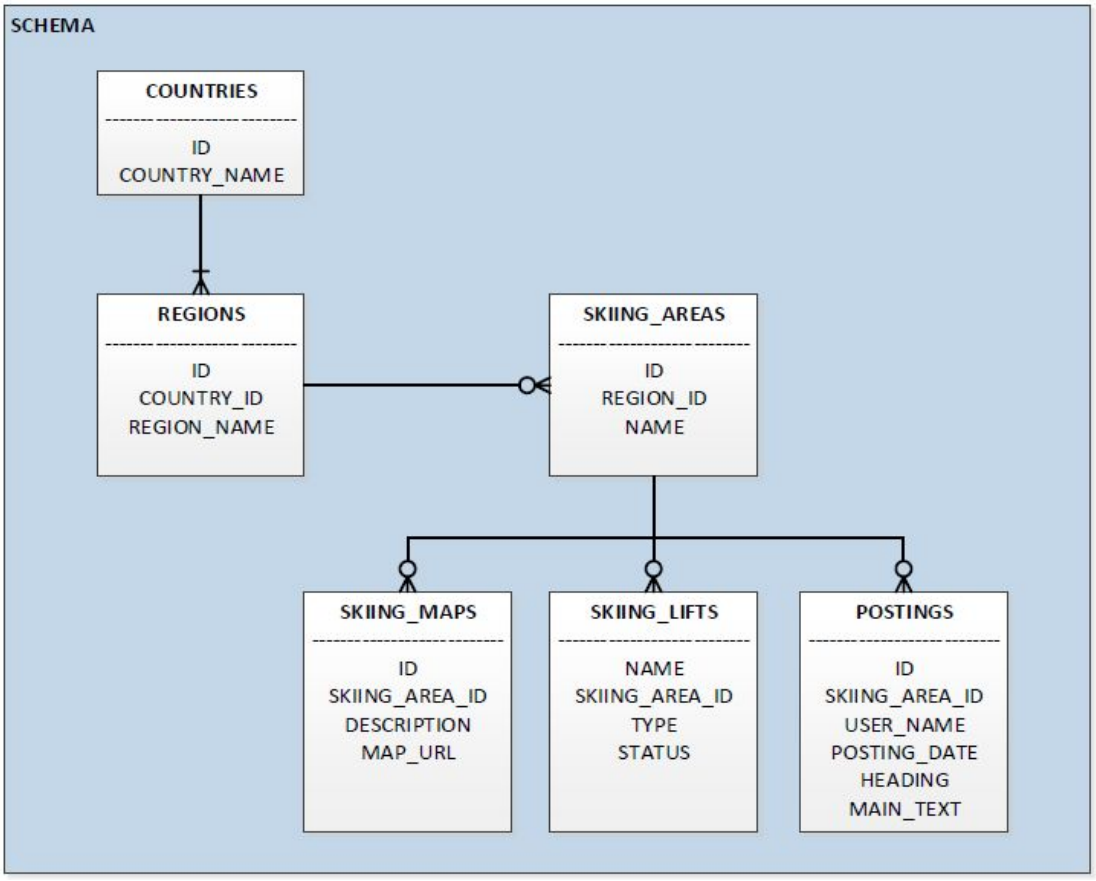

Abbildung 11: MySQL - Relationales Modell

- Jedes Land kann eine oder mehrere Regionen beinhalten.

- Skigebiete sind nicht mit Ländern, sondern Regionen verknüpft.

- Skigebiete können, müssen aber nicht, Informationen über die Lifte des Skibgebiets, Übersichskarten und Kommentare von Benutzern speichern

Die Beziehungen unter den Entitäten werden in relationalen Datenbanken in Form von Relationships festgelegt. Mit dem hier gezeigten Model ist es daher nicht möglich, ein Skigebiet direkt mit einem Land zu verknüpfen. Auch die Spalten jeder Tabelle und die Bedingungen für jede Spalte werden explizit bei Anlage der Tabelle mit Hilfe von Constraints angegeben. Diese überwachen in weiterer Folge die Richtigkeit der gespeicherten Daten für diese Bedienungen. Zudem werden durch die Aufteilung der Daten in einzelne Tabellen Redundanzen vermieden.

3.3.2 CouchDB

Mit der Möglichkeit in CouchDB Daten in Dokumenten auch verschachtelt zu speichern, ergeben sich auch andere Möglichkeiten zur Datenmodellierung. Diese werden im Folgenden näher untersucht.

Variante 1

In der Ersten Art der Speicherung der Daten, werden alle Daten eines Skigebiets in einem Dokument gespeichert. Abbildung 12 zeigt dieses Modell.

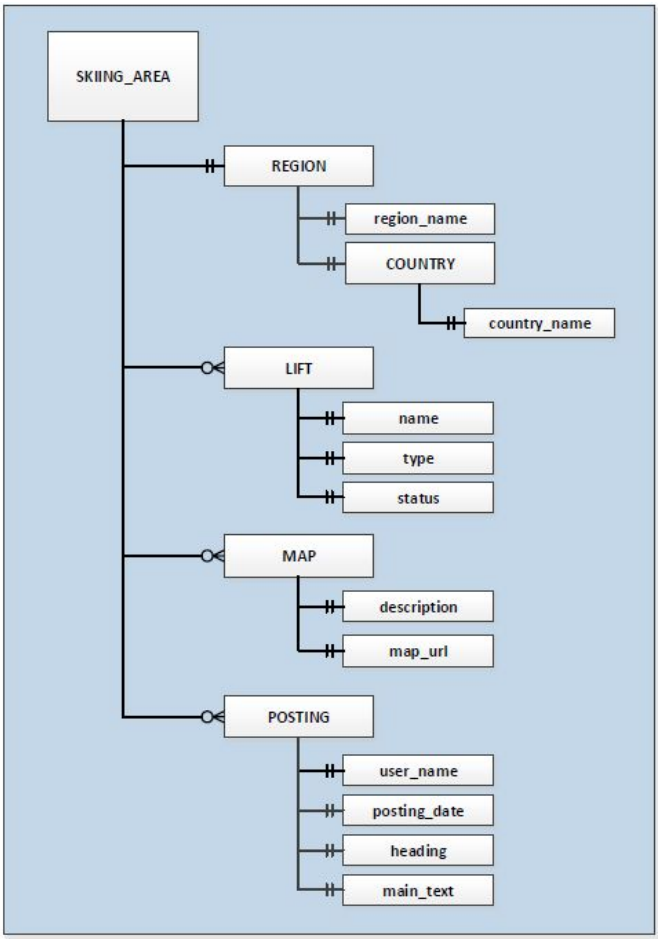

Abbildung 12: CouchDB - Variante 1

Aus dem Diagramm können folgende Schlussfolgerungen gezogen werden:

- Die Lage des Skigebiets beinhaltet die Information über das Land, sowie die

Region in dem das Skigebiet liegt.

- Skilifte, Karten und Kommentare von Benutzern werden in Form von Listen im Dokument gespeichert.

Ein Dokument würde diesem Modell nach wie folgt aussehen:

```
{
    "_id": "00z353737f89c0eg2e10e88a0c0002h3",
    "_rev": "1-1635g55gc8c3abfddc1f6r50c9942ft1",
    "doc_type": "skiing_area",
    "name": "Schladming (Planai)",
    "location": {
        "country_name": "Austria",
        "region_name": "Styria"
    },
    "lifts": [
        {
            "name": "Burgstallalmlift",
            "type": "Chairlift",
            "status": "open"
        },
        {
            "name": "Planai 1. Teilstrecke",
            "type": "ropeway",
            "status": "open"
        },
        {
            "name": "Planai 2. Teilstrecke",
            "type": "ropeway",
            "status": "closed"
        }
    ],
    "maps": [
        {
            "description": "Panorama overview",
            "map_url": "maps/planai.jpg"
        }
```

```
32      ],
33      "postings": []
34  }
```

Zu beachten ist, das CouchDB keine Möglichkeit der Einhaltung der im Diagramm eingezeichneten Struktur bietet. So könnte ein Dokument über Skilifte verfügen, aber keine Karten und Kommentare beinhalten. CouchDB würde auch andere Bezeichnungen und komplett andere Inhalte im Dokument zulassen. Die Einhaltung der eingezeichneten Struktur liegt damit alleine bei den aufrufenden Programmen.

Um auch Länder und Regionen in der Datenbank zu speichern, können diese in einem eigenen Dokument gespeichert werden, wie Abbildung 13 zeigt.

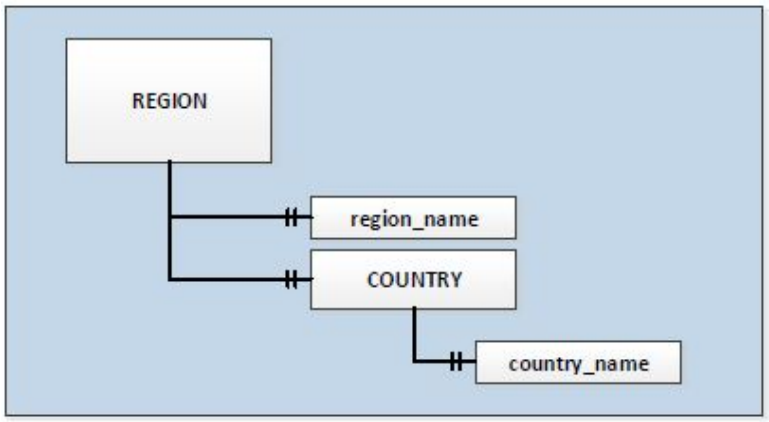

Abbildung 13: CouchDB - Speicherung der Länder und Regionen

Damit können Programme auch Länder und Regionen abfragen, bevor diese im Skigebiet-Dokument gespeichert werden. Allerdings besteht keinerlei Beziehung zwischen den beiden Dokumenten. Erneut ist die aufrufende Applikation für die Einhaltung der richtigen Länder und Regionen zuständig. Zudem müssen bei einer Änderung einer Länderbezeichnung alle Skigebiete, die dieses Land beinhalten, manuell geändert werden.

Variante 2

Die in Variante 1 gezeigte Speicherung der Region im Dokument des Skigebiets hat zur Folge, dass der Ländername redundant in jedem Dokument gespeichert wird. Aber auch in CouchDB kann stattdessen eine Referenz auf die ID des Regionsdokuments gespeichert werden. Abbildung 14 zeigt das Diagramm für diese Variante.

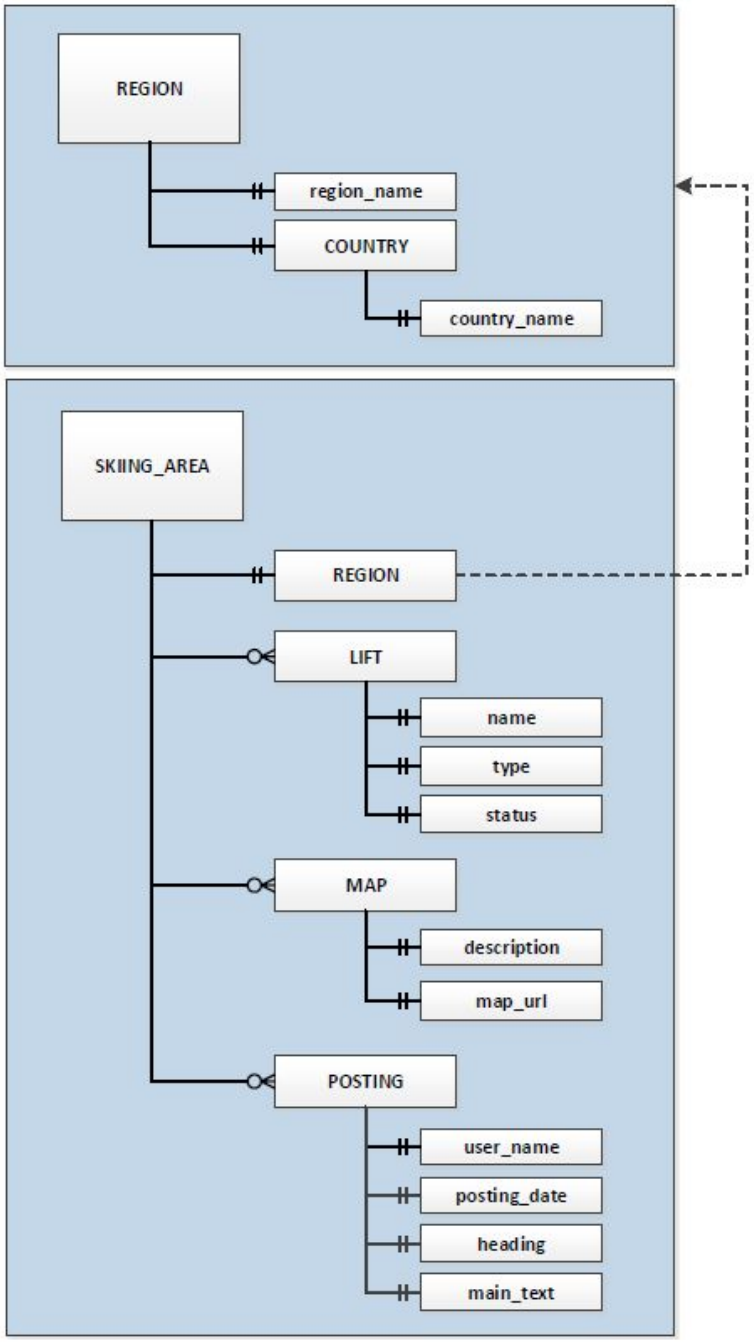

Abbildung 14: CouchDB - Variante 2

Damit würden zwei Dokumente benötigt werden, um ein Skigebiet zu beschreiben:

```
{
    "_id": "01z3347355z9c0eg2e15r34a4d6566wr7",
    "_rev": "1-3345g5r3c343abfddc1f6r50c994f2e4",
    "doc_type": "region",
```

```
    "region_name": "Styria",
  "country": "Austria"
}
```

```
{
    "_id": "00z353737f89c0eg2e10e88a0c0002h3",
    "_rev": "1-1635g55gc8c3abfddc1f6r50c9942ft1",
    "doc_type": "skiing_area",
    "name": "Schladming (Planai)",
    "region": "01z3347355z9c0eg2e15r34a4d6566wr7",
    "lifts": [
        {
            "name": "Burgstallalmlift",
            "type": "Chairlift",
            "status": "open"
        },
        {
            "name": "Planai 1. Teilstrecke",
            "type": "ropeway",
            "status": "open"
        },
        {
            "name": "Planai 2. Teilstrecke",
            "type": "ropeway",
            "status": "closed"
        }
    ],
    "maps": [
        {
            "description": "Panorama overview",
            "map_url": "maps/planai.jpg"
        }
    ],
    "postings": []
}
```

Während diese Variante der Speicherung in relationalen Datenbanken näher kommt, gibt es doch entscheidende Unterschiede: Dokumentenorientierte NoSQL Daten-

banken wie CouchDB haben keinen festgelegten Schema und erzwingen keine Einhaltung dieser Verbindungen. Außerdem unterstützen sie keine Validierung der Beziehungen. Weiters unterstützt CouchDB keine Abfragen in der Art, wie es in relationalen Datenbanken und SQL mit Joins der Fall ist. Es müssen daher zwei getrennte Abfragen an die Datenbank zur Ermittlung der Region und des Skigebiets abgesetzt werden (die Abfrage von Daten wird im Detail in Kapitel 3.5 beleuchtet). Daher ist in diesem Fall die Speicherung des Landes und der Region innerhalb des Skigebiet Dokuments die bessere Wahl.

Variante 3

Eine weitere Überlegung gilt der Speicherung von Informationen, wie den Benutzerkommentaren, im Dokument der Skigebiete. Wird hier eine große Zahl an Kommentaren erwartet, würde die Speicherung direkt im Dokument der Skigebiete zu sehr großen Dokumenten führen. Dieses müsste zudem bei jedem neuen Posting erneut als ganzes gespeichert werden. Daher kann auch in diesem Fall die Speicherung der Postings in ein eigenes Dokument ausgelagert werden. Abbildung 15 zeigt diese Variante.

Es gelten auch hier die gleichen Merkmale beim Verlinken der Dokumente wie in Variante 2. Dennoch könnte in diesem Fall die Auslagerung der Kommentare in ein eigenes Dokument durchaus sinnvoll sein. Dies würde die Größe des Skigebiete Dokuments verringern und dadurch zu schnelleren Ladezeiten führen. Das Laden der Benutzerkommentare könnte dann in einem asynchronen, zweiten Aufruf erfolgen.

Ergebnis

Die Vergleiche in diesem kleinen Beispiel zeigen bereits, das mit NoSQL Daten in anderer Form gespeichert werden können, als dies in relationalen Datenbanken der Fall ist. Die Speicherung der Daten ohne fixes Schema scheint dabei verlo-

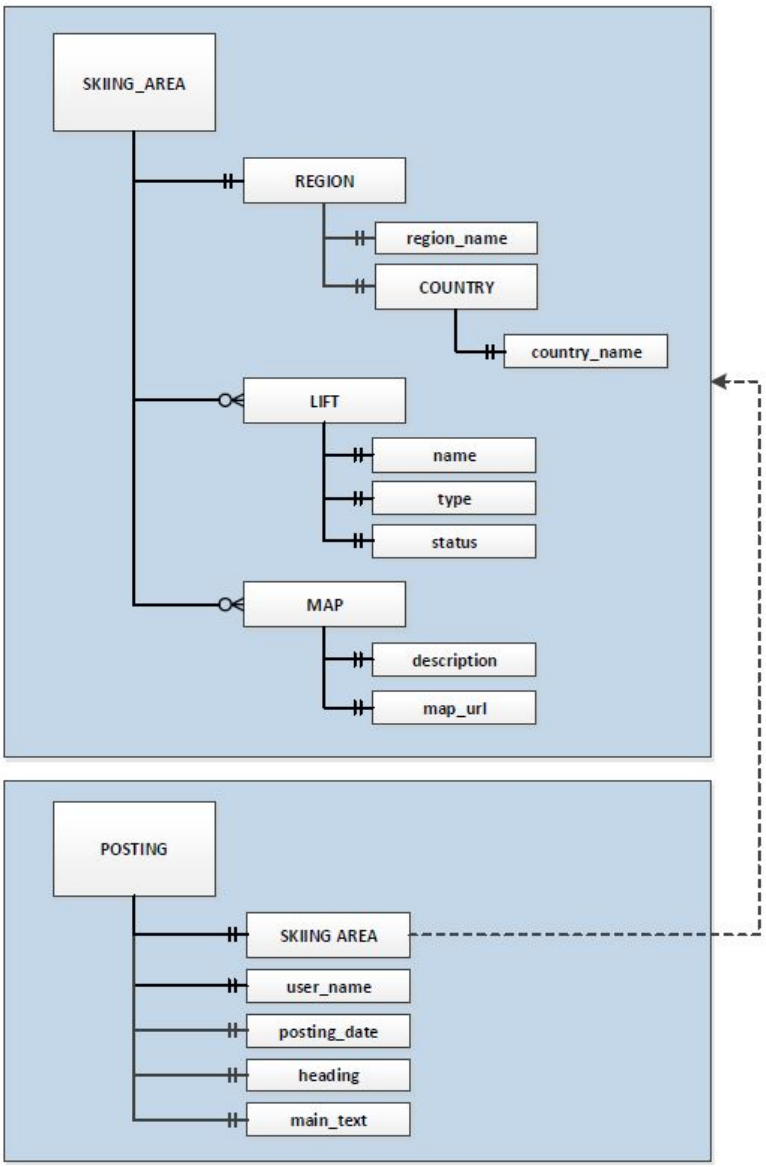

Abbildung 15: CouchDB - Variante 3

ckend, bricht sie doch die starren Verbindungen und Speicherung in normalisierter Form der relationalen Datenbanken auf. Allerdings sind die damit einhergehenden Auswirkungen nicht zu unterschätzen. Die Applikation muss in diesem Fall einen Großteil der Aufgaben übernehmen, die ansonsten die Datenbank übernimmt. Eine sorgfältige Datenmodellierung ist daher mindestens genauso wichtig, wie das bei relationalen Datenbanken der Fall ist.

Festzustellen ist auch, dass die Modellierung in dokumentenorientierten NoSQL Datenbanken wie CouchDB voraussetzt, dass man von der Vorgehensweise bei

relationalen Datenbanken abweicht. Ein reines kopieren der Modellierung einer relationalen Datenbank ist nicht sinnvoll. Um die Vorteile, die NoSQL bietet, auch nutzen zu können, muss man sich der Unterschiede bewusst sein und die gesamte Applikation darauf abstimmen.

3.4 Datenerstellung

In diesem Kapitel wird untersucht, wie Daten in den beiden Datenbanken erzeugt und gespeichert werden können.

3.4.1 MySQL

SQL bietet umfangreiche Möglichkeiten, um Daten in MySQL zu erzeugen und zu ändern. Ein neuer Eintrag in der Ländertabelle kann beispielsweise wie folgt erzeugt werden:

```
1 INSERT
2   INTO skiing.countries (id, iso_code, country_name)
3 VALUES (512, 'AT', 'AUSTRIA');
4
5 COMMIT;
```

Eine wichtige Grundlage relationaler Datenbanken bilden dabei Transaktionen. Die Daten werden erst dann tatsächlich in der Datenbank festgeschrieben, wenn der Insert Befehl mit einem Commit Befehl abgeschlossen wird. Dieser kann auch erst nach mehreren Insert Befehlen erfolgen. Erfolgt kein Commit, oder wird ein expliziter Rollback Befehl ausgeführt, werden alle bis dahin nicht durch einen Commit abgeschlossenen Eingaben rückgängig gemacht. Das gleiche gilt auch für Befehle die Daten ändern:

```
1 UPDATE skiing.countries
2    SET country_name = 'Austria'
3  WHERE iso_code = 'AT';
```

```
4
5 COMMIT;
```

Mit dem Commit werden die bestehenden Daten, in diesem Fall der Ländername „Austria“, mit den neuen Daten, hier „Oesterreich“, überschrieben. Der Ländername „Austria“ ist danach in der Datenbank nicht mehr vorhanden. Eine Ausnahme bilden dabei Funktionen wie es Oracle mit Flashback Queries bietet. Dabei können auch Daten gelesen oder zurück geholt werden, die vor einem Commit bestanden haben. Grundsätzlich werden Daten aber in relationalen Datenbanken bei einem Update mit eine Commit endgültig überschrieben.

SQL bietet auch eine große Flexibilität beim Ändern der Daten. Im Beispiel wurde der Ländername anhand des ISO Codes abgeändert. Ebenso könnte das Update aber anhand der ID erfolgen oder es könnten auch mehrere Dateneinträge mit einem Update geändert werden. Funktionen wie Subqueries erlauben es zudem, über ganze Tabellen zu iterieren und jede Zeile anhand anderer Werte zu ändern, wie folgendes Beispiel zeigt.

```
1 UPDATE skiing.countries cty
2    SET cty.regions = ( SELECT count(1)
3                          FROM skiing.regions
4                         WHERE country_id = cty.id
5                       );
```

Dieses Statement setzt eine Spalte mit dem Namen „REGIONS“ in der Ländertabelle voraus. Eine entsprechende Speicherung der Anzahl der Regionen in der Ländertabelle ist nicht sinnvoll, dieses Beispiel soll lediglich die Möglichkeiten aufzeigen, die SQL bietet.

Um die Konsistenz der Daten zu gewährleisten, arbeiten relationalen Datenbanken mit Locks. Die zeitgleiche Änderung des selben Datensatzes ist damit nicht möglich. Sobald eine Transaktion aber mit einem Commit abgeschlossen wurde, kann das nächste Update die Daten erneut überschreiben. Mit Hilfe von Transaktionen können in SQL aber auch mehrere Befehle zusammengefasst werde. So

kann eine Vielzahl an schreibenden Operationen hintereinander erfolgen und diese erst am Ende mit einem Commit oder Rollback bestätigt oder zurück genommen werden.

3.4.2 CouchDB

In CouchDB werden Daten über HTTP Requests erzeugt und geändert. Mit folgendem Befehl wird der gleiche Eintrag wie zuvor erzeugt, diesmal allerdings als Dokument:

```
curl -X PUT
http://127.0.0.1:5984/skiing/6e1295ed6c29495e54cc05947f18c8af
-d ' { "doc_type": "country",
       "iso_code": "AT",
       "country_name":"Austria"
     }'
```

Im Gegensatz zu MySQL wird das Dokument mit dem übersendeten Inhalt sofort erstellt. Ein Commit ist hier nicht notwendig. CouchDB bestätigt das Anlegen des Dokuments mit folgender Antwort:

```
{
  "ok":true,
  "id":"6e1295ed6c29495e54cc05947f18c8af",
  "rev":"1-2902191555"
}
```

Die Antwort besteht aus dem Universal Unique Identifier, kurz UUID, sowie der Revisionsnummer. Die UUID wurde dabei bereits bei der Anlage mit übergeben. Um eine neue, gültige UUID zu erlangen, kann folgender Befehl abgesetzt werden:

```
curl -X GET http://127.0.0.1:5984/_uuids
```

CouchDB liefert daraufhin folgende Antwort:

```
{
  "ok":true,
```

```
  "id":"5e8564ed6c25696e54jh05947p74c8mk",
  "rev":"1-1582191355"
}
```

Um mehrere UUIDs auf einmal abzurufen, kann dem Request optional noch ein Parameter hinzugefügt werden:

```
curl -X GET http://127.0.0.1:5984/_uuids?count=5
```

Die Revisionsnummer des Dokuments steht für einen wichtigen Mechanismus in CouchDB. Im Gegensatz zu MySQL wird in CouchDB bei einer Änderung des Dokuments ein neues Dokument mit einer neuen Revisionsnummer erstellt. Die Daten werden also nicht überschrieben, sondern bestehen nebeneinander. Um ein Dokument überhaupt abändern zu können, ist dementsprechend eine Revisionsnummer notwendig.

```
curl -X PUT
http://127.0.0.1:5984/skiing/6e1295ed6c29495e54cc05947f18c8af
-d ' { "rev":"1-1582191355"
       "doc_type": "country",
       "iso_code": "AT",
       "country_name":"Oesterreich"
     }'
```

Mit der Änderung erzeugt CouchDB ein neues Dokument mit einer neue Revisionsnummer. Wird das selbe Dokument mit der gleichen Revisionsnummer zeitgleich geändert, wird lediglich jener Befehl ausgeführt, der zuerst den Server erreicht. Die zweite Änderung des selben Dokuments würde folgende Antwort erzeugen:

```
{"error":"conflict","reason":"Document update conflict."}
```

Die gleiche Antwort würde auch erzeugt werden, wenn dem zu ändernden Dokument keine Revisionsnummer mitgegeben wird. Um Daten in CouchDB zu ändern ist somit die aktuelle Revisionsnummer des Dokuments immer zwingend notwendig.

Um veraltete, nicht mehr benötigte Revisionen von Dokumenten nicht in der Datenbank aufheben zu müssen, bietet CouchDB die Funktion Compaction. Damit werden diese Revisionen in der Datenbank gelöscht.

```
curl -H "Content-Type: application/json" -X POST http://localhost
    :5984/skiing/_compact
```

Im Gegensatz zu MySQL bietet CouchDB keine erweiterte Transaktionskontrolle an. Jede Änderung eines Dokuments erfolgt atomar in einem Vorgang und wird sofort bestätigt. Mehrere Operationen können aber nicht zusammengefasst bestätigt oder zurück genommen werden. Sollen eine oder mehrere Operationen zurück genommen werden, muss das Dokument mit der Revisionsnummer vor der Änderung gespeichert werden.

Ergebnis

Auch dieser Vergleich zeigt deutliche Unterschiede zwischen den beiden Datenbanken. Während in MySQL mittels SQL Daten gezielt und flexibel bearbeitet werden, wird in CouchDB immer das komplette Dokument in der aktuellsten Revision benötigt. Zudem können in NoSQL nicht mehrere Operationen in einer Transaktion zusammengefasst werden und gemeinsam bestätigt oder zurück genommen werden. Solange Daten in einem Dokument gemeinsam gespeichert und geändert werden, fällt dieser Unterschied vielleicht nicht so sehr ins Gewicht. Werden sie aber über mehrere Dokumente hinweg in einer Operation bearbeitet, muss erneut die Applikation die Aufgabe der Transaktionskontrolle übernehmen und die Änderungen gegebenenfalls rückgängig machen. Dieser Unterschied sollte daher bereits bei der Planung der Applikation und Datenmodellierung entsprechend berücksichtigt werden.

3.5 Abfragen

Dieses Kapitel widmet sich den Abfragemöglichkeiten der beiden Datenbanken. Es wird gezeigt, inwiefern die Datenbanken die zuvor erzeugten Daten aus der Datenbank wieder ausgelesen können und auf welche Art und Weise die Applikationen auf die Daten zugreifen können.

3.5.1 MySQL

Relationale Datenbanken bieten mit SQL eine zum Großteil standardisierte Sprache zur Datenabfrage an. Mit Hilfe von SQL können umfangreiche Abfragen abgesetzt werden, welche die Daten gefiltert, zusammengefasst oder gruppiert zurückgeben. Ein großer Vorteil der Abfragen mit SQL ist, dass diese auch ad-hoc formuliert und abgesetzt werden können und nicht schon vorab definiert werden müssen. Ähnlich wie im Kapitel zuvor, bietet SQL damit einen sehr flexiblen Zugriff auf die Daten an.

Allerdings gibt die Form der Datenspeicherung in relationalen Datenbanken meist nicht die interne Repräsentation der Daten in der Applikation wieder. Dort könnten Skigebiete zum Beispiel mitsamt der Lifte und Karten des Gebiets, oder zumindest dem Verweis darauf, gespeichert sein. Um die Daten entsprechend zu konvertieren, müssen diese mit Hilfe von Joins vorher jedes Mal zusammengefasst werden. Objektrelationale Abbildungen (ORM - Object-relational mapping) wie zum Beispiel mittels OBatis oder Hibernate helfen bei dieser Konvertierung. Dennoch müssen die Daten intern bei jeder Abfrage entsprechend konvertiert werden.

Um einen Vergleich der Abfragemöglichkeiten von relationalen Datenbanken zu NoSQL Datenbanken zu erhalten, werden im folgenden einige beispielhafte Abfragen anhand des vorgestellten Datenmodells formuliert. Die selben Abfragen werden im Anschluss auch mit CouchDB umgesetzt. Als erstes sollen Skigebiete anhand des Namens gesucht werden. SQL bietet hierfür zahlreiche Möglichkeiten,

die folgende Abfrage zeigt eine Möglichkeit alle Skigebiete die das Wort „Schladming“ enthalten zurück zu geben.

```
SELECT sg.name, rg.region_name, ct.country_name
  FROM skiing.skigebiete sg
 INNER JOIN skiing.regions rg   ON sg.region_id  = sg.id
 INNER JOIN skiing.countries ct ON rg.country_in = ct.id
 WHERE sg.name LIKE '%Schladming%';
```

Genauso einfach könnten alle Skigebiete in Österreich wieder gegeben werden:

```
SELECT sg.name, rg.region_name, ct.country_name
  FROM skiing.skigebiete sg
 INNER JOIN skiing.regions rg   ON sg.region_id  = sg.id
 INNER JOIN skiing.countries ct ON rg.country_in = ct.id
 WHERE ct.iso_code = 'AT';
```

Oder die Anzahl an Skigebieten pro Land:

```
  SELECT ct.country_name, count(1) skiing_areas
    FROM skiing.skigebiete sg
   INNER JOIN skiing.regions rg   ON sg.region_id  = sg.id
   INNER JOIN skiing.countries ct ON rg.country_in = ct.id
GROUP BY ct.iso_code;
```

Oder es wird die Anzahl der Liften in die Abfrage inkludiert:

```
SELECT sg.name, rg.region_name, ct.country_name,
      ( SELECT count(1)
          FROM skiing_lifts
         WHERE skiing_area_id = sg.id
      ) AS lifts
  FROM skiing.skigebiete sg
 INNER JOIN skiing.regions rg   ON sg.region_id  = sg.id
 INNER JOIN skiing.countries ct ON rg.country_in = ct.id
 WHERE ct.iso_code = 'AT';
```

Die Abfragen zeigen die flexiblen Zugriffsmöglichkeiten, die SQL bietet. Dies ist nicht nur für die Applikation selbst von Vorteil, sondern kann auch später, wenn

ad-hoc Auswertungen über bestimmte Bereiche benötigt werden, von großem Nutzen sein.

Will man allerdings alle Skigebiete mitsamt seiner Lifte zurückgeben, wird die Abfrage etwas komplizierter. Natürlich könnte man auch hier die Tabellen wie vorhin mit Joins verknüpfen, aber dann würde die Information über das Skigebiet für jeden Lift zurück gegeben werden:

```
SELECT sg.name, rg.region_name, ct.country_name,
      ( SELECT count(1)
          FROM skiing_lifts
         WHERE skiing_area_id = sg.id
      ) AS lifts
  FROM skiing.skigebiete sg
 INNER JOIN skiing.regions rg   ON sg.id = sg.region_id
 INNER JOIN skiing.countries ct ON ct.id = rg.country_in
  LEFT JOIN skiing.lifts li     ON sg.id = lf.skiing_area_id
 WHERE ct.iso_code = 'AT';
```

Selbst dieses kleine Beispiel zeigt, dass die Anzahl der Joins schnell steigt und dadurch mitunter schwerer les- und wartbar wird. Entsprechend werden die Abfragen aufgeteilt und zuerst die Daten für das Skigebiet geladen und erst danach die Information über die dazugehörigen Lifte:

```
SELECT sg.name, rg.region_name, ct.country_name
  FROM skiing.skigebiete sg
 INNER JOIN skiing.regions rg   ON sg.region_id  = sg.id
 INNER JOIN skiing.countries ct ON rg.country_in = ct.id
 WHERE sg.id = 1;

 SELECT li.name, li.type, li.status
   FROM lifts li
  WHERE li.skiing_area_id;
```

Diese Aufteilung ist in vielen Webapplikationen wiederzufinden, wo Daten in Master-Detail Forms angezeigt werden. Hier zeigt sich deutlich die Diskrepanz zwischen der internen Repräsentation der Daten in der Applikation und der Da-

tenspeicherung in der relationalen Datenbank. Um die gespeicherten Daten in der Form zu liefern, die die Programmlogik benötigt sind zahlreiche Umwandlungen notwendig.

3.5.2 CouchDB

CouchDB bietet keine ad-hoc Abfragemöglichkeiten, wie es mit SQL der Fall ist. Entsprechend müssen Abfragen vorab überlegt werden und Views dafür angelegt werden. Views sind in der Datenbank gespeicherte Dokumente, die mittels Map-Reduce die entsprechenden Daten ermitteln. Views werden ebenfalls über HTTP Requests in der Datenbank angelegt und erstellen einen Index, auf den in Folge zugegriffen werden kann. Mit folgendem Befehl kann eine View zum Ermitteln aller Skigebiete in CouchDB angelegt werden:

```
curl -H 'Content-Type:application/json' -X
POST http://admin:admin@127.0.0.1:5984/skiing/ -d
'{ "_id": "_design/all_areas_by_name",
   "language": "javascript",
   "views": { "all_areas_by_name":
                { "map": "function(doc) {
                            if(doc.doc_type == ""skiing_area"") {
                              emit( doc.skiing_area_name, doc);
                            };
                         }"
                }
           }
}'
```

Diese View bedient sich dabei lediglich des Map Teils der MapReduce Abfrage. Zuerst wird dabei überprüft, ob der Dokumenten-Typ ein Skigebiet ist. Trifft dies zu, wird die UUID des Dokuments, sowie ein Key-Value Paar zurück gegeben. Dieses Key-Value Paar wird in der emit() Funktion angegeben und ist entscheidend für mögliche Abfragen auf die View. In diesem Fall wird der Name des Skigebiets als Schlüssel und das gesamte Dokument als Wert zurück gegeben.

Darauf aufbauend, können nun alle Skigebiete abgefragt werden:

```
curl -X GET http://127.0.0.1:5984/skiing/_design/all_areas_by_name/
    _view/all_areas_by_name
```

Diese Abfrage würde alle Skigebiete, ohne Einschränkung zurück liefern. Jedes Skigebiet würde damit das Dokument mit allen beinhaltet Details, also beispielsweise auch den Liften und Karten, beinhalten. Ein einzelnes Skigebiet kann wie folgt abgefragt werden:

```
http://127.0.0.1:5984/skiing/_design/all_areas_by_name/_view/
    all_areas_by_name?key="Hochkar"
```

Dies setzt allerdings voraus, dass der Schlüssel wie zuvor beschrieben aus dem Namen des Skigebiets besteht. Auch die zuvor in MySQL formulierte Abfrage nach Skigebieten die das Wort Schladming beinhalten, ist nun (beinahe) umsetzbar:

```
curl -X GET http://127.0.0.1:5984/skiing/_design/all_areas_by_name/
    _view/all_areas_by_name?startkey="Schladming"&endkey="SchladmingZ
    "
```

Im Unterschied zur Abfrage in MySQL, muss das Skigebiet allerdings dem dem Wort „Schladming“ beginnen. Views in CouchDB werden vorab berechnet und unterstützen nur die Abfrage über einen bestimmten Schlüssel. Die in MySQL formulierte Abfrage mit LIKE ist nur dann möglich, wenn der Beginn des Suchbegriffes bekannt ist.

Eine Möglichkeit, diese Einschränkung zu umgehen, wäre es, vom Programm eine Temporäre View anzulegen. Diese wird erstellt und liefert sofort das Ergebnis zurück:

```
curl -H 'Content-Type:application/json' -X
POST http://admin:admin@127.0.0.1:5984/skiing/_temp_view -d
'{
  {"map": "function (doc) {
            if (doc.name && doc.name.indexOf(""Schladming"") !== -1)
    {
               emit( doc.skiing_area_name, doc);
```

```
            }
          }"
  }
}'
```

Bei großen Datenmengen ist dieser Weg allerdings nicht ratsam, da das Ergebnis zur Laufzeit ermittelt wird, indem alle Dokumente durchsucht werden. Als Alternative kann mit Plugins wie couchdb-lucen oder ElasticSearch ein textbasierter Index erstellt werden, über den danach gesucht werden kann.

Die bisher verwendetet View kann auch die Frage nach der Anzahl an Skigebieten pro Land nicht beantworten. Für diese Abfrage wird eine zusätzliche View benötigt, die folgende Funktion beinhaltet:

```
{ "map": "function(doc) {
  if(doc.doc_type == "skiing_area") {
    emit([doc.location.country_name, doc.location.region_name], 1);
    };
  }",
  "reduce": "function(keys, values, rereduce) {
     return sum(values);
  }"
}
```

In diesem Fall kommt auch der Reduce Teil der Abfrage zum Einsatz. Im ersten Schritt werden in der Map Abfrage für jedes Skigebiet der Ländername als Schlüssel und die Zahl eins als Wert zurück geliefert. Die Reduce Abfrage fasst im Anschluss die Werte pro Ländername zusammen und liefert somit die Summe der Skigebiete pro Land zurück. Da in diesem Fall das Dokument nicht zurück gegeben wird, sondern nur der Ländername, ist auch eine Ermittlung des Ländercodes nicht möglich. Für diesen Fall müsste die View geändert werden, oder eine zweite Abfrage an eine andere View gestellt werden.

Ergebnis

Dieser Abschnitt zeigt die vielleicht gravierendsten Unterschiede zwischen der MySQL und der CouchDB Datenbank. CouchDB kann komplette Dokumente mit allen benötigen Daten mit nur einer Abfrage zurück liefern. Dazu wird nur das Dokument abgefragt und es werden keinerlei aufwendigen Joins, wie das in SQL oft der Fall ist, benötigt. MySQL hingegen bietet mit SQL die Möglichkeit, auch umfangreichere Abfragen und Datenanalysen einfach erstellen zu können.

Hinzugefügt werden muss an dieser Stelle, dass andere NoSQL Datenbanken wie beispielsweise MongoDB, mitunter umfangreichere Abfragemöglichkeiten als CouchDB bieten. Dennoch wird die Flexibilität der relationalen Datenbanken nicht erreicht. Auch hier muss die Applikation, wo notwendig und möglich, die weitere Datenmanipulation übernehmen.

Kapitel 4

Zusammenfassung

Der Vergleich der relationalen Datenbank MySQL und der NoSQL Datenbank CouchDB hat gezeigt, dass relationale Datenbanken keineswegs zu den alten Eisen gehören. Im Gegenteil. Mit der Transaktionskontrolle für mehrere Operationen und den flexiblen Abfragemöglichkeiten mit Hilfe von SQL bieten sie wertvolle Instrumente. Zudem können sie auch zahlreiche Aufgaben zur Kontrolle der gespeicherten Daten übernehmen. Zusammenfassend kann aus dem Vergleich der Schluss gezogen werden, dass relationale Datenbanken dann die richtige Wahl sind, wenn ein hoher Wert auf die Konsistenz der Daten gelegt wird und - oder - die Daten oft auf unterschiedliche Arten abgefragt werden sollen. Für die Entwicklung einer Buchhaltungssoftware, wo verbuchte Daten flexibel abgefragt und manipuliert werden müssen, wäre entsprechend die Verwendung einer relationale Datenbank zu empfehlen.

Aber nicht immer stehen die zuvor beschriebenen Punkte im Mittelpunkt der Überlegungen. Und der Vergleich hat gezeigt, dass auch NoSQL Datenbanken wie CouchDB durchaus Vorteile zu bieten haben. Zum Beispiel die Speicherung komplexer Datenstrukturen in einem Dokument und der damit verbundenen Möglichkeit die Daten mit nur einer Abfrage zu ermitteln. Oder der Möglichkeit, Daten auf einfache Art und Weise auf mehrere Server zu verteilen. Oder der Möglichkeit, Dokumente flexibel zu ändern oder zuerweitern, ohne dabei Anpassungen an der

Struktur vornehmen zu müssen, wie das bei Tabellen in relationalen Datenbanken der Fall ist.

Wie in Kapitel 2.6 beschrieben, steht der Begriff NoSQL aber auch für zum Teil sehr unterschiedliche Datenbanken. Jede Art von NoSQL Datenbank bietet damit auch unterschiedliche Vorteile. NoSQL Datenbanken sind dann zu empfehlen, wenn die Eigenschaften der Datenbanken klar zu den Aufgaben der Applikation passen. Im Fall von CouchDB wäre beispielsweise eine Wiki Anwendung ein möglicher Anwendungsfall. Die Daten einer Wiki Seite könnten in einem Dokument gespeichert werden und dabei beliebige Datenstrukturen enthalten. Analytische ad-hoc Abfragen wären bei einer solchen Applikation die Ausnahme und die Tatsache, dass einzelne Seiten des Wikis, und damit die Dokumente, über eine unterschiedliche Struktur verfügen können, wäre ein großer Vorteil.

Zusammenfassend hat sich gezeigt, dass die Wahl der richtigen Datenbank keine einfache ist. Mit der richtigen Entscheidung kann der Grundstein für eine erfolgreiche Applikation gelegt werden und NoSQL Datenbanken können eine sinnvolle Alternative zu relationalen Datenbanken darstellen. Vielleicht ist aber der richtige Weg gar nicht die Entscheidung für oder gegen eine Datenbank, sondern vielmehr die Nutzung mehrerer Datenbanken für eine Applikation. Ein Key-Value Store für schnelle Zugriffe für Session Daten, eine relationale Datenbank für sensible Finanzdaten und eine Dokumentenorientierte Datenbank für die Dokumentations- und Hilfeseiten wäre eine Möglichkeit, unterschiedliche Datenbanken in einer Applikation zu vereinen. Das Ergebnis des Vergleichs dieser Arbeit liefert damit keinen Gewinner und Verlierer, sondern zwei Gewinner mit zahlreichen Möglichkeiten und Funktionen.

Abbildungsverzeichnis

Literaturverzeichnis

Anderson, J. C., Lehnardt, J. & Slater, N. (2010), *CouchDB: The Definitive Guide*, O'Reilly Media, Inc., Sebastopol.

Chodorow, K. (2013), *MongoDB: The Definitive Guide*, 2. edn, O'Reilly Media, Inc., Sebastopol.

Dean, J. & Ghemawat, S. (2008), 'Mapreduce: simplified data processing on large clusters', *Communications of the ACM* **51**(1), 107–113.

Gantz, J. & Reinsel, D. (2011), 'Extracting value from chaos', *IDC iview* (1142), 9–10.

Hewitt, E. (2010), *Cassandra: The Definitive Guide*, O'Reilly Media, Inc., Sebastopol.

Kyte, T. (abgerufen am 22.03.2015), 'Use of nested tables', `https://asktom.oracle.com/pls/asktom/f?p=100:11:0%3A%3A%3AP11_QUESTION_ID:8135488196597`.

MongoDB (abgerufen am 02.04.2015), 'Sharding introduction', `http://docs.mongodb.org/manual/core/sharding-introduction/`.

Redmond, E. & Wilson, J. R. (2012), *Seven Databases in Seven Weeks - A Guide to Modern Databases and the NoSQL Movement*, 1. edn, Pragmatic Bookshelf, Raleigh, Noth Caolina and Dallas, Texas.

Robinson, I., Webber, J. & Eifrem, E. (2013), *Graph Databases*, O'Reilly Media, Inc., Sebastopol.

Sadalage, P. J. & Fowler, M. (2012), *NoSQL Distilled - A Brief Guide to the Emerging World of Polyglot Persistence*, 1. edn, Addison-Wesley, Amsterdam.

Tiwari, S. (2011), *Professional NoSQL*, John Wiley & Sons, New York.

Turner, V., Gantz, J. F., Reinsel, D. & Minton, S. (2014), 'The digital universe of opportunities: Rich data and the increasing value of the internet of things', *International Data Corporation, White Paper, IDC_1672* .

Vaish, G. (2013), *Getting Started with Nosql*, Packt Publishing Ltd, Birmingham.

White, T. (2012), *Hadoop: The Definitive Guide*, 3. edn, O'Reilly Media, Inc., Sebastopol.