Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Master Arbeit:

Inkrementelle Auswertung geobasierter MongoDB-Anfragen

Marcel Patzwahl

marcel.patzwahl@informatik.uni-hamburg.de
Studiengang Informatik Msc.
Matr.-Nr. 6208642

Gutachter:	Prof. Dr.-Ing. Norbert Ritter
Gutachter:	Dipl.-Inform. Fabian Panse
Betreuer:	Wolfram Wingerath

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit geht es um die Zusammenführung zweier, für sich etablierte Themen aus der Informatik. Auf der einen Seite steht die Geografie (bzw. genauer die Geodäsie). Dabei handelt es sich um „die Wissenschaft der Ausmessung und Abbildung der Erdoberfläche“ [Tor02] und wurde durch Friedrich Robert Helmert definiert. Dieser lange für tot geglaubte Zweig der Mathematik hat in jüngster Zeit wieder an Relevanz gewonnen [Bru12]. Nicht zuletzt durch die Einführung von GPS Systemen und den darauf aufbauenden Navigationssystemen. Auch die Einführung von GPS auf dem Smartphone hat diesen Siegeszug weiter vorangetrieben [Bru12]. Heute gibt es diverse Bibliotheken für geodätische Berechnungen für verschiedenste Programmiersprachen. Geo[1] für C#, Spatial4j[2] für Java oder GeoPHP[3] für PHP sind nur einige Beispiele eines großen Spektrums von Bibliotheken. Die Aufgabe dieser Bibliotheken ist es, die meist komplexe Mathematik hinter der Berechnung von Distanzen oder ähnlichen, geometrischen Eigenschaften zu verstecken und sie durch eine einfach zu nutzende Schnittstelle einsetzen zu können. Eines der am etabliertesten in diesem Bereich ist die S2 Bibliothek. Sie wird von Google selbst als Open Source Projekt freigegeben [Goo17d] und führt u.a. die Berechnungen im Hintergrund von Google Maps durch [Per17].
Zudem gibt es SQL wie auch NoSQL Datenbanken zur Anfrage von räumlichen Daten (o.a. *Spatial Data*). Diese Daten werden nach einer spezifischen Indexierungsstrategie gespeichert, die die Abfrage dieser Daten schneller macht (Siehe Kapitel 3).
Das zweite Thema dieser Arbeit sind Echtzeitanfragen an Datenbanken. Anders als bei klassischen Datenbankanfragen, wird bei Echtzeitanfragen eine Socket-Verbindung zum Server aufrecht erhalten, sodass der Klient mit den aktuellsten Daten versorgt werden kann. Dieses Prinzip wird tiefgreifend in Kapitel 2 dargestellt. Als konkreter Anwendungsfall wird InvaliDB [Winht] der Universität Hamburg dienen.
Die folgende Arbeit wird sich damit beschäftigen, inwieweit es möglich ist, geodätische Anfragen auch in Echtzeitdatenbanksystemen umzusetzen.

1.1 Motivation

Seit der Entstehung des World Wide Web (WWW), gibt es den Wunsch Daten immer schneller an den Endnutzer auszuliefern. Angefangen hat das WWW mit der Auslieferung statischer Webseiten. Dabei war das Laden einer Internetpräsenz eine Prozedur nach dem klassischen Client-Server Modell, wie es in Abbildung 1.1 zu sehen ist. Der Client ruft

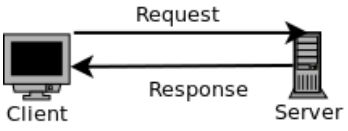
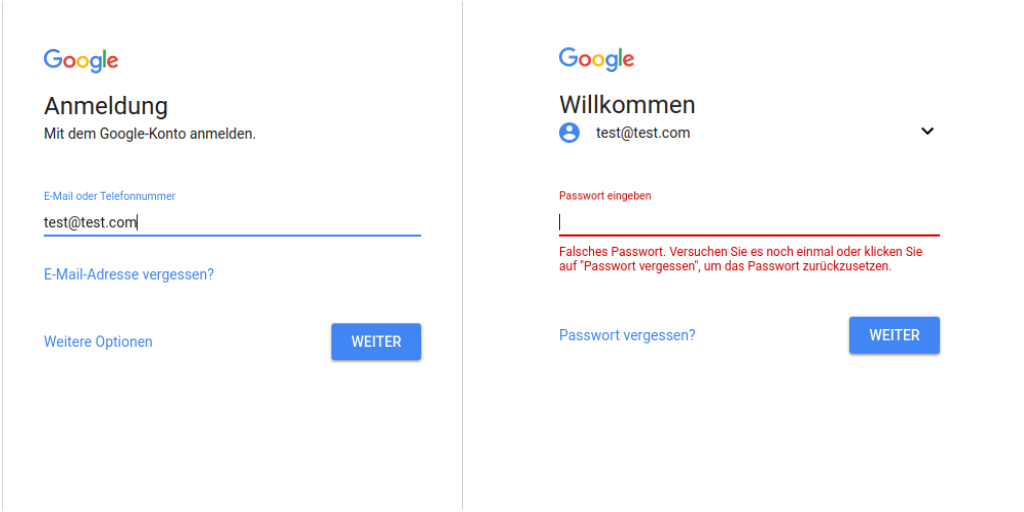


Abbildung 1.1: Client-Server Kommunikation

[1] https://github.com/sibartlett/Geo
[2] https://github.com/locationtech/spatial4j
[3] https://geophp.net/

mithilfe eines Browsers eine Internetseite auf und stellt damit eine Anfrage (o.a. *Request*) an einen Server. Dieser gibt eine Antwort (o.a. *Response*) zurück, dem sogenannten DOM-Baum (kurz: DOM). Dieser ist jedoch nichts anderes als die Anordnung von *HTML Tags* in einer Baumstruktur. Das Problem an dieser Art der Kommunikation ist, dass beliebige Anfragen immer einen vollständigen DOM als Antwort zur Folge haben. Dies ist sehr ineffizient bei minimalen Änderungen am DOM.



(a) Google Login [Goo18c] (b) Google Login fehlgeschlagen [Goo18c]

Abbildung 1.2: Beispiel Google Login: Minimale Veränderung im DOM

Ein Beispiel für eine minimale Änderung wäre, dass bei Eingabe eines falschen Logins, ein Informationstext erscheint, um den Nutzer über die fehlerhafte Eingabe zu informieren (Siehe Abbildung 1.2b). Die DOMs würden sich minimal darin unterscheiden, dass der Login mit dem Informationstext über ein modifiziertes *div* Element verfügt. Für diese Änderung war zu Beginn des Internets das Nachladen des vollständigen DOMs nötig. Der Grund hierfür war, dass es auf Seite des Klienten keine Möglichkeit gab, die Daten auszuwerten. Aus diesem Grund wurden die Daten an den HTTP-Server geleitet, der wiederum eine vollständige Seite ausliefert.
Mit **Javascript** kam 1995 [Fla07] eine Sprache zu HTML und CSS hinzu, die es erlaubte auf Seite des Klienten Auswertungen vorzunehmen und Inhalte zu verändern oder neu zu laden. Diese Prozeduren wurden im Browser des Klienten ausgeführt, ohne dass der Server kontaktiert werden muss.
Aufbauend auf Javascript kam **AJAX** im Jahre 2005 [Gar18]. Diese Technologie erlaubte es erstmals, ohne weitere Browser-Plugins oder Einsetzen eines Java-Applets, Teilbereiche einer Seite dynamisch nachzuladen. Somit konnte bei vielen Operationen die Kommunikation mit dem HTTP-Server umgangen werden. Das Neu-Laden einer gesamten HTML Seite war damit nicht mehr zwingend notwendig.
Diese Technologien sollten aber nur den Anfang bilden. Denn schnell erkannten auch prägende Firmen wie Google und Facebook das Potenzial und veröffentlichten neue Bi-

bliotheken auf Basis von Javascript namens **AngularJS**[4] und **React**[5]. Mit diesen Technologien ist es möglich, lediglich eine feste HTML Seite auszuliefern und bei Aktionen des Nutzers Bereiche auszutauschen. Im Gegensatz zu AJAX wurden nun nicht nur Teile in einer Menge von HTML Seiten ausgetauscht, sondern der gesamte DOM-Baum wird dynamisch erzeugt und verändert. Aus diesem Grund sind diese Art von Internetseiten auch als **Single Page Applications** bekannt [MP13]. Auch neben den Frontend Technologien gibt es immer mehr Vorstöße das Backend mehr in Richtung Echtzeitauswertung zu bringen [KGM94].
Der Flaschenhals war nun die Datenbank. Sie basiert auf einem *pull*-basierten Prinzip. Das heißt, dass bei einer Anfrage des Nutzers die relevanten Daten aus der Datenbank angefragt und dem Benutzer präsentiert werden. Jedoch gibt es längst viele Beispiele, in denen diese Art der Datenanfrage unzureichend geworden ist. Jede Applikation, in der etwas in Echtzeit wiederholt aktualisiert werden muss, wie z.B. der Live-Ticker eines Fußballspiels, ein Chat oder ein kollaboratives Textverarbeitungsprogramm hat somit einen sehr großen Mehraufwand, Daten aktuell zu halten.
In einem Szenario, in dem *pull*-basierte Systeme im Hintergrund arbeiten, muss die Datenbank wiederholt angefragt werden. Auf den Großteil der Anfragen wird jedoch mit den gleichen Antworten reagiert, weil es noch keine aktuelleren Daten gibt. Das Stellen und die Abarbeitung dieser Anfragen sind verschwendete Ressourcen und stellen das *pull*-basierte Paradigma für diese Art von Anwendungen infrage. Aus diesem Grund gibt es in jüngster Zeit mehrere Versuche *push*-basierte Datenbanksysteme zu entwickeln [Win17a]. Das grobe Ziel ist es, dass die Datenbank proaktiv aktualisierte Daten an den Klienten schickt, ohne dass diese zuerst angefragt werden müssen. Denn die Datenbank weiß zuerst, wann Änderungen vorliegen. Das *push*-basierte Paradigma ist somit besser geeignet für Echtzeitanwendungen. Damit sind Ressourcen frei für andere Aufgaben.
Zu Beginn wurden einige Beispiele mit Chatrooms und Live-Tickern gegeben. Neben diesen klassischen Beispielen sollen im folgenden auch einige Beispiele motiviert werden, in denen es von Vorteil ist, geodätische Echtzeitanfragen zu nutzen. Dies sind unter anderem Szenarien, in denen der Nutzer bewegliche Objekte auf einer Karte verfolgen möchte. Ein konkreter Anwendungsfall könnte beispielsweise das Rufen eines Taxis sein. Mit einer Anfrage in einer App, dass der Kunde in 5 Minuten ein Taxi benötigt, könnte geprüft werden welches der Taxen im nahen Umfeld derzeit frei ist. Ein anderer Fall könnte eine Applikation sein, die Menschen miteinander vermittelt, die spontan ein gemeinsame Aktivität ausführen möchten und sich somit in der näheren Umgebung zueinander befinden müssen. Wie zu sehen ist, besteht nicht nur ein theoretisches Interesse an der Umsetzbarkeit von geodätischen Datenanfragen in Echtzeit. Es gibt auch mehrere konkrete Umsetzungskonzepte, in denen es durchaus Sinn für den Entwickler macht, ein *push*-basiertes System im Hintergrund zu nutzen.

[4] https://angularjs.org/
[5] https://reactjs.org/

1.2 Struktur der Arbeit

Im Folgenden wird nach der Problemstellung eine Einführung in den Kontext gegeben, in der diese Arbeit umgesetzt wird. Dabei soll zuerst auf die Theorie hinter Real-Time-Queries eingegangen werden. Anschließend wird am konkreten Beispiel der Firma Baqend, deren Gesamtarchitektur mit ihren wichtigsten Komponenten für diese Arbeit vorgestellt. Diese sind Orestes und InvaliDB. Anschließend werden einige Grundlagen besprochen, auf die diese Arbeit aufbaut. Hier wird es zuerst um die von MongoDB genutzte S2 Bibliothek gehen. Darauf hin wird der Quasistandard für austauschbare geografische Daten namens *GeoJSON* vorgestellt. Aufbauend wird die Implementation von MongoDB begutachtet. Mit dem Wissen um den Inhalt den MongoDB im Bereich Geo-Queries bietet, wird eine Real-Time Version der Anfrageauswertung in Orestes implementiert, die das gleiche Ein- und Ausgabeverhalten bietet. Die Anfrageauswertung wird im darauf folgenden Kapitel an einem konkreten Beispiel mithilfe einer Baqend Anwendung gezeigt und evaluiert. Zum Schluss folgt ein Ausblick und das Fazit. Verzeichnisse für Grafiken und Literatur sind in den Anhängen zu finden.

1.3 Problemstellung

Geo-Queries finden ihren Einsatz bereits in einer Vielzahl von Datenbanken, wie MySQL [Ora17] oder MongoDB [Inc17x]. Dabei können sie auch als Fließkommazahl in den Tabellen oder Dokumenten gehalten werden und die Berechnungen von der Applikation durchführen lassen. Dies ist aus folgenden Gründen nicht performant: Als Beispiel soll die Distanzberechnung eines Punktes zu bestimmten Datensätzen dienen. Eine Applikation fragt eine Menge von Datensätzen an und filtert die, die näher als eine bestimmte Distanz zum gegebenen Punkt liegen. Jeden Datensatz anzufragen und anschließend nur die behalten, die das Distanzkriterium erfüllen, ist keine optimale Lösung. Es wird eine sehr große Datenmenge übertragen, wovon nur eine kleine Menge von Relevanz ist. Aus diesem Grund ist es von Vorteil, die Filterung bereits auf Datenbankebene vorzunehmen, sodass nur die relevanten Ergebnisse an die Applikation übertragen werden. Ein weiterer Faktor ist die Berechnungsart der Distanz. Je nachdem, wie diese implementiert ist, kann sie bereits in frühen Berechnungsschritten einen Großteil der Datensätze ausschließen, bevor teurere Operationen ausgeführt werden. Dieses Problem wird im späteren Verlauf der Arbeit genauer beschrieben. Somit sollte ähnlich wie in der Kryptografie [Zim99] auf bestehende Lösungen, wie der Implementationen in der Datenbank zurückgegriffen werden, anstatt diese selbst zu entwickeln.
Die bestehenden Lösungen konzentrieren sich allerdings darauf, geodätische Daten einer Anfrage gegen eine Menge von Datensätzen zu testen (*pull*-basiert). Diese Arbeit wird die entgegengesetzte Richtung beleuchten, in der eine Anfrage dauerhaft gehalten wird. Hierfür muss der Zustand des Queries zunächst ausgewertet werden. Dies gibt eine erste Ergebnismenge von Objekten zurück. Ab diesem Zeitpunkt muss bei jeder Schreib-

operation auf das System erneut ausgewertet werden, ob die hinzugefügten, gelöschten oder modifizierten Objekte auf die Anfrage zutreffen. In der folgenden Arbeit wird das Verhalten, dass ein Objekt auf die Filterkriterien einer Anfrage (o.a. **Query**) zutrifft, als **Match** bezeichnet. Entsprechend den matchenden Objekten, muss die Ergebnismenge des Queries angepasst werden. Somit ist die Herausforderung, bereits bestehende *pull*-basierte Geo-Queries inkrementell auszuwerten. Diese Art der Verarbeitung soll möglichst performant sein. Performant bedeutet in diesem Kontext, dass sich das System für den Benutzer reaktiv anfühlen soll. Er darf somit nicht das Gefühl bekommen, dass er auf Resultate seitens der Applikation wartet.

1.4 Begriffsklärung und verwendete Notation

In diesem Kapitel sollen grundlegende Begrifflichkeiten geklärt werden, die als Basis für die folgende Arbeit angenommen werden.

Latitude

Die Latitude ist der englische Begriff für die geografische Breite (o.a. Breitengrad). Dieser wird in Grad angegeben und beschreibt die Entfernung vom Äquator. Die beiden vom Äquator am weitesten entfernten Punkte sind der Nord- und Südpol. Der Nordpol liegt bei $+90°$ und der Südpol bei $-90°$. Somit kann sich die Latitude nur zwischen diesen beiden Werten bewegen. Der Begriff Latitude wird dem ansonsten im deutschen weiter verbreiteten Begriff Breitengrad vorgezogen, da sich diese Arbeit hauptsächlich mit Englischer Literatur und Implementationen auseinander setzt und selbst darauf aufbaut.

Longitude

Komplementär zur Latitude beschreibt die Longitude den Längengrad. Ausgehend vom Nullmeridian wird der Längengrad nach Osten und Westen ebenfalls in Grad angegeben. Dabei kann bis zu $\pm 180°$ in die beiden Himmelsrichtungen gemessen werden. Die Wahl der Bezeichnung fällt aus den gleichen Gründen auf Longitude, anstatt Längengrad, wie zuvor bei der Begriffsklärung zur Latitude erläutert.

Verwendete Notation

Besondere Begriffe werden in der folgenden Arbeit *kursiv* hervorgehoben, um sie vom Text abzuheben. Bei diesen Schlagworten handelt es sich um spezielle Begriffe, wie sie bspw. auch im beiliegenden Quelltext vorzufinden sind. Dabei sind es zumeist Klassen, Methoden/Funktionen, Indices, Query Selektoren, Query Operatoren o.ä.
Viele der *kursiv* verwendeten Begriffe sind ebenfalls in **Bold** vorzufinden. Dies hebt den Teil des Textes hervor, in dem die Begriffe eingeführt und erklärt werden.

2 Kontext

In diesem Kapitel soll es um den Kontext gehen, in dem die Anfrageauswertung für geografische Queries umgesetzt wird. Dabei wird zuerst das theoretische Konzept der sogenannten Real-Time-Queries beleuchtet. Anschließend wird die konkrete Architektur von Baqend vorgestellt, welche Real-Time-Queries bereits in einer Beta Version unterstützen [Baq17b]. Dabei werden die beiden Komponenten InvaliDB und Orestes der Quaestor Architektur [GSW+17] detailliert vorgestellt, da sich die Implementation dieser Arbeit ebenfalls in Orestes eingliedert.

2.1 Real-Time Queries

Echtzeit-Anfragen o.a. Real-Time Queries sind aus einem Bedürfnis entstanden, dass Nutzer nach einer Aktion umgehend den Effekt eben dieser sehen wollen [Win17b]. Diese Eigenschaft wird weithin als reaktiv [KBD13] verstanden. Während die reaktive Programmierung bereits in aktuellen Sprachen und Frameworks angekommen ist [Rea17], arbeiten Datenbanken weithin mit einem, für dieses Szenario nicht optimalen, *pull*-basierten System.
Pull-basiert bedeutet, wenn ein Klient Informationen erhalten will, muss er die Datenbank für die Daten anfragen. Das ist sehr ineffizient in reaktiven Anwendungen, da so kontinuierlich Datenbankanfragen gestellt werden müssen, um die aktuellsten Daten anzeigen zu können. Ansonsten könnte der Klient veraltete Daten sehen.
An dieser Stelle greifen Real-Time Queries ein. Sie basieren auf einem *push*-basierten Prinzip. Das heißt, Klienten bekommen proaktiv von der Datenbank die aktuellsten Daten zugeschickt. Da die Datenbank immer zuerst weiß, wann Daten verändert wurden, kann sie auch am schnellsten über Aktualisierungen informieren. So ist es für bestimmte Kontexte ein effektiverer Ansatz, als der *pull*-basierte. Durch dieses Konzept spart sich der Klient das ununterbrochene Anfragen und somit auch Ressourcen.
Während *pull*-basierte Systeme auf dem Client-Server-Pattern beruhen, sind *push*-basierte Systeme meist mithilfe des **Observer-Patterns** realisiert [app18a][Baq17b][Wen18]. Das *Observer-Pattern* funktioniert dabei wie folgt: Der Klient kann sich als **Observer** an einem **Observable** anmelden [Sch09]. Im Fall von Real-Time-Queries dem Query. Bei einem Update, die den Query betreffen, werden alle *Observer* mit einer **Subscription** an diesem *Query* benachrichtigt. In webbasierten Anwendungen wird beispielsweise bei der ersten Anfrage eine Socket-Verbindung aufgebaut [app18a]. Über diesen Kanal ist die Klient-Applikation dauerhaft mit dem Server verbunden (*Subscription*). Über diesen offenen Kanal ist der Server weiterhin in der Lage *Updates* zu schicken, wenn diese vorliegen. Das Prinzip, dass über einen solchen Kanal dauerhaft Daten transferiert werden können, wird auch als **Streaming** bezeichnet [Ran14].
Im Folgenden soll gezeigt werden, wie Real-Time Queries bereits praktisch bei Baqend

umgesetzt werden und wie eine Javascript-Schnittstelle für diesen Anwendungsfall konstruiert ist.

2.2 InvaliDB

InvaliDB ist eine skalierbare Systemarchitektur. Es ermöglicht *push*-basierte Real-Time Queries aufbauend auf einer klassischem *pull*-basierten Datenbank [Winht]. Dabei ist auch die Query-Engine selbst austauschbar, um eine vollständige Unabhängigkeit zur Datenbank zu gewährleisten [Win17b]. Dieses System wird derzeit am Fachbereich Informationssysteme (VSYS) entwickelt und vom Startup Baqend in ihrer **Database-as-a-Service** Plattform genutzt, um Real-Time Queries zu ermöglichen.

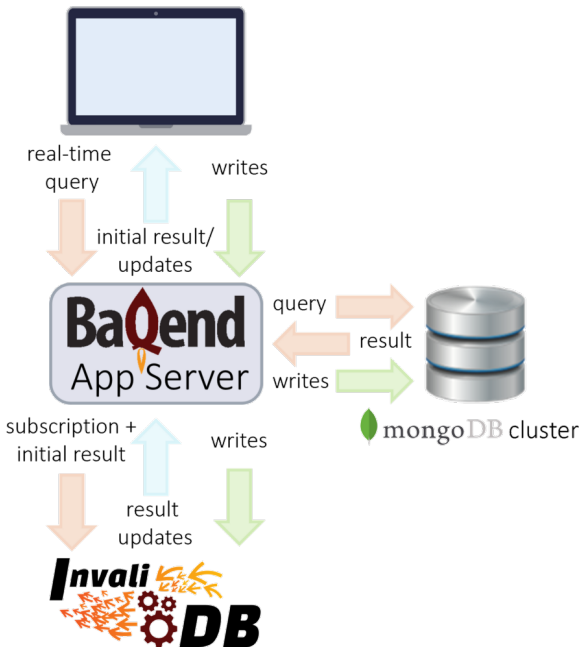


Abbildung 2.1: Schematische Darstellung der Baqend Architektur [Win17b]

Wie in Abbildung 2.1 zu sehen ist, kommuniziert InvaliDB direkt mit den Baqend App Servern. Diese senden Daten über eine in Redis realisierte Message Queue [Suc]. Dabei können drei Szenarien auftreten, in denen ein App Server mit InvaliDB kommuniziert [Win17b]. Wenn ein Real-Time Query *subscribed* wird, sendet der App Server den Query und das initiale Ergebnis, welches der App Server vom MongoDB Cluster bekommt, an InvaliDB (Analog muss auch eine *Subscription* in InvaliDB gekündigt werden). Kommen Schreiboperationen vom Klienten, wird dies im MongoDB Cluster ausgeführt und das resultierende Objekt (**After Image** genannt) wird an InvaliDB gegeben. InvaliDB prüft anschließend für die gehaltenen Real-Time Queries den Matching Status. Dabei gibt es drei Szenarien, die eine Ergebnismenge beeinflussen können. In Abbildung 2.2 ist dies in einer Baumstruktur aufgezeigt. Die beiden wichtigen Fragen, die jedem Datensatz gestellt werden, sind:

1. Waren sie vor dem After-Image ein Match?
2. Sind sie nach dem After-Image noch ein Match?

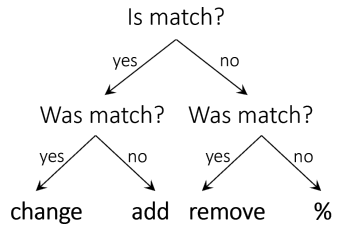


Abbildung 2.2: Real-Time-Queries Matching Baum [Win17b]

Trifft auf beides „Nein“ zu, kann der Datensatz ignoriert werden. Alle anderen Szenarien haben einen Einfluss, indem sie als Match in die Ergebnismenge aufgenommen, entfernt oder geändert werden müssen.
Die Resultate der Auswertung werden zurück an die App Server gegeben, die die betroffenen Real-Time Queries *subscribed* haben.

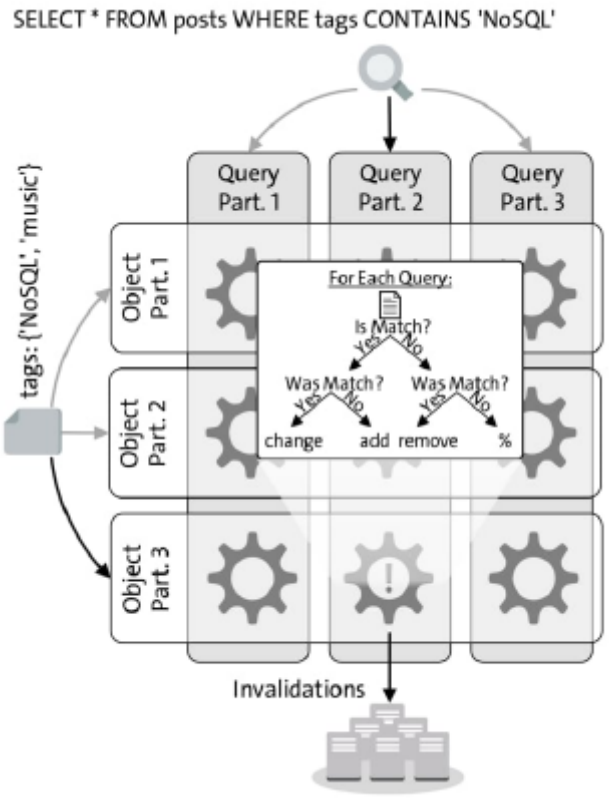


Abbildung 2.3: InvaliDB Partitionierung [Winht]

Intern basiert InvaliDB auf einem zweidimensionalen Partitionsschema (Siehe Abbildung 2.3). InvaliDB erlaubt es so in zwei Dimensionen zu skalieren. Steigt die Zahl an *Query Subscriptions*, können weitere Query Partitionen hinzugefügt werden. Gleichzeitig können weitere Objektpartitionen hinzugefügt werden, um den Durchsatz der Schreiboperationen zu erhöhen. Zudem ist in [Winht] nachgewiesen worden, dass InvaliDB in

beide Dimensionen linear mit der Anzahl der Objekte/Queries skaliert.
Objekte und Queries werden anhand eines berechneten Hashes in einer Partition abgelegt, sodass die Zahl an Objekten und Queries gleichmäßig über das gesamte Cluster verteilt werden. Die Implementation von InvaliDB basiert dabei auf Apache Storm [Winht].

2.3 Orestes

Die letzte Komponente, die in diesem Verbund erklärt werden soll, ist Orestes. Nach Abbildung 2.1 ist Orestes innerhalb des Baqend App Servers einzuordnen. Auf der Website vom Fachbereich VSYS der Universität Hamburg wird Orestes wie folgt eingeleitet:

> „Orestes ist eine skalierbares Database- und Backend-as-a-Service System. Es reichert NoSQL-Datenbanksysteme mit funktionalen und nicht-funktionalen Eigenschaften an. Durch automatisierte polyglotte Persistenz werden Anfragen an ein zu der Anwendung passendes Datenbanksystem weitergeleitet. Durch kohärentes Web-Caching mit Cache Sketches werden Datenbank-Objekte automatisch gecacht und durch ein Bloom filter-basiertes Cache-Kohärenz-Protokoll aktuell gehalten, um die Vorteile von geringer Latenz und hoher Skalierbarkeit mit definierbaren Konsistenz-Kriterien zu vereinen. Scalable Cache-Aware Optimistic Transactions bieten skalierbare Transaktionen für unveränderte NoSQL-Systeme. Des weiteren werden systemübergreifende Backend- und Datenbank Abstraktionen geschaffen, u.a. feingranulare Zugriffskontrolle durch ein Authentifizierungs- und Autorisierungenssystem, ein reichhaltiges objektorientiertes Schema-System, eine Infrastruktur für JavaScript-basierte serverseitige Stored Procedures und Trigger, sowie Autoskalierungsstrategien für Caching, Replikation und Sharding.“ [VSI17]

Orestes ist somit eine weitere Schicht, die ebenfalls mit mehreren Servern verteilt und unabhängig skalierbar ist. Orestes kann vor verschiedene Datenbanken geschaltet werden. In Abbildung 2.1 bildet es den Applikationsserver zwischen MongoDB und InvaliDB. Das Caching ist zwar eines der Hauptaufgaben von Orestes, soll in dieser Arbeit aber nicht weiter im Detail betrachtet werden. Im Fokus soll nun die Verarbeitung der Real Time Queries stehen.
Wie bereits im vorherigen Abschnitt erwähnt, gibt es drei Szenarien in denen Orestes mit InvaliDB kommuniziert. Um als Vermittler zwischen MongoDB und InvaliDB agieren zu können, muss Orestes in der Lage sein MongoDB Queries zu verstehen und auszuwerten. In diesem Bereich setzt diese Arbeit ein, indem sie die bestehende Auswertung mit der Auswertung von geodätischen Anfragen anreichert.

2.4 Die Baqend Platform

Die Firma Baqend bietet zum Zeitpunkt dieser Arbeit zwei Dienste an: das **Speed Kit**[1] und die **Platform**[2]. Das *Speed Kit* versucht bestehende Webseiten durch verschiedene neuartige Technologien (wie CDNs) schneller zu machen. Die *Platform* ist eine sogenannte *Backend-as-a-Service* Plattform. Entwickler können sich auf der Webseite von Baqend einen Account anlegen und eine oder mehrere Apps erstellen. Für jede App wird eine virtuelle Umgebung bereitgestellt, die der zuvor dargestellten Architektur aus InvaliDB, MongoDB und Orestes entspricht. Dies soll aber irrelevant für den Entwickler sein, der auf der Plattform eine Applikation entwickelt. Ziel von *Backend-as-a-Service* soll sein, dass das gesamte Backend einer Applikation wie ein Service behandelt wird. Das heißt, es gibt eine Schnittstelle über die der App-Entwickler mit dem Backend kommunizieren kann. Allerdings hat der App-Entwickler ansonsten keine Kontrolle über das Backend. Auch die Hoheit über die Daten ist nicht gegeben, da diese beim Dienstleister in den Rechenzentren liegen. Dieser Ansatz hat Vor- und Nachteile (Einfachheit gegen Datenhoheit) [BF18]. Der Entwickler nimmt aber die Nachteile in Kauf, um schneller ein funktionierendes Produkt an den Markt bringen zu können. Das Aufsetzen, Warten und Erweitern eines funktionierenden und performanten Backends fällt weg, da diese Aufgabe an den Dienstleister des *Backend-as-a-Service* Dienstes abgegeben wird. *Backend-as-a-Service* ist ein besonders lukrativer Ansatz für Firmen, die nur oder hauptsächlich Frontend-Entwickler beschäftigen. Somit müssen sie sich nicht um das Backend kümmern, da es automatisch mit der Applikation skaliert.
Mithilfe einer Schnittstelle kann im Beispiel von Baqend in Javascript mit dem Backend kommuniziert werden [Baq18a]. Baqend bietet einige Starter-Kits an, die den Umgang mit einigen bekannten Frameworks, wie AngularJS, React oder auch Bootstrap erleichtern sollen [Baq18b]. *Backend-as-a-Service* Anbieter finanzieren sich, indem Nutzer der Plattform Pakete buchen, in denen sie mehr Speicher oder mehr Requests pro Monat bekommen [Lan15]. Die Baqend Schnittstelle gibt es derzeit in zwei verschiedenen Varianten. Einmal die klassische, stabile Schnittstelle und zum anderen die Real-Time Version der Schnittstelle. Zweiteres ist derzeit separat verfügbar, da es sich hier zum Zeitpunkt des Schreibens der Arbeit (Jan. 2018) noch um ein Beta-Feature handelt [Baq17b].
Die Real-Time Schnittstelle bietet zwei Arten mit Real-Time Queries zu arbeiten. Die erste Möglichkeit sind **Self-Maintaining Queries** [Baq17b], die immer die aktuellsten Daten zu einem Query enthalten. Nachdem eine Anfrage gestellt wurde, gibt Baqend eine Ergebnismenge zurück. Diese wurde durch eine initiale Anfrage an MongoDB ermittelt. Gleichzeitig wird auch der Query in InvaliDB eingepflegt und der Klient meldet sich an diesem Query an [Win17b]. Dies funktioniert analog zum zuvor erwähnten *Observer-Pattern*. Wird nun eine Schreibanfrage gestellt, die auf die Ergebnismenge von den in InvaliDB gehaltenen Queries zutrifft, werden diese anschließend erneut zu den betroffe-

[1] http://www.baqend.com/speedkit.html
[2] http://www.baqend.com/platform.html

nen Klienten geschickt. Die von Baqend bereitgestellte Javascript-Schnittstelle ist in Listing 2.1 gezeigt. In diesem Listing wird die grundlegende Nutzung der *Self-Maintaining Queries* vorgeführt.

```
1  // Einen Query erzeugen, der das 'name' Feld der Todo Kollektion nach dem
2  // Wort kochen durchsucht und auf die ersten 10 Ergebnisse limitert ist.
3  var query = DB.Todo.find()
4    .matches('name', /^kochen/)
5    .limit(10);
6
7  // Callback Funktionen, die bei bestimmten Ereignissen aufgerufen werden
8  // Wenn eine Aktualisierung der Query Matches geschah. Result
9  // entspricht der aktuellen Ergebnissmenge, die zutrifft.
10 var onNext = result => console.log(result);
11 // Wenn ein Fehler aufgetreten ist
12 var onError = err => console.log(err); // optional
13 // Wenn es zu einem Verbindungsabbruch kam
14 var onComplete = () => console.log('I am offline!'); // optional
15
16 // Der Klient meldet sich am Query an, um über Aktualisierungen informiert
17 // zu werden
18 var subscription = query.resultStream(onNext, onError, onComplete);
19 // Subscription wird aufgelöst
20 subscription.unsubscribe();
```

Listing 2.1: Baqend Javascript-Schnittstelle für *Self-Maintaining Queries* [Baq17b]

Die zweite Anfrage-Art sind die **Event Stream Queries**. Hier wird anders als beim *Self-Maintaining Query*, nicht die Ergebnismenge aktualisiert, sondern ein Ereignis (o.a. **Event**) mit Informationen darüber verschickt, welche der Daten aktualisiert wurden [Baq17b]. Durch diese Art von Query-Subscription müssen insgesamt weniger Daten übertragen werden. Zudem ermöglicht es dem Klienten selbst zu entscheiden, wie er auf das jeweilige *Event* reagiert. In Listing 2.2 ist ein Beispiel für *Event Stream Queries* zu sehen. Dabei wurde das Beispiel nur an den relevanten Stellen von Listing 2.1 angepasst. Der Query ist in diesem Beispiel der gleiche wie zuvor. Es wurde lediglich die *onNext* Methode und der Subscription Aufruf angepasst.

```
1  // Das Result kommt gekapselt in einem Event, welches einiges zusätzliche
2  // Informationen neben den matchenden Daten beinhaltet
3  var onNext = event =>
4    console.log(event.matchType + '/'
5      + event.operation + ': '
6      + event.data.name + ' is now at index '
7      + event.index);
8
9  var subscription = query.eventStream(onNext, onError, onComplete);
```

Listing 2.2: Baqend Javascript-Schnittstelle für *Event Stream Queries* [Baq17b]

In Kapitel 5 wird mithilfe einer Baqend App und der hier beschriebenen API eine *Proof-of-Concept* Applikation vorgestellt, die die Geo-Query Auswertung dieser Arbeit in InvaliDB nutzt. An diesem Beispiel soll evaluiert werden, wie sich die Geo-Queries im Kontext von Echtzeit Anfragen verhalten. In dieser Applikation soll gezeigt werden, ob es möglich ist Real-Time Geo-Queries umzusetzen und wie performant sich diese verhalten.

3 Grundlagen

3.1 Googles S2 Bibliothek

Das Kernstück von MongoDBs geodätischen Anfragen stellt Googles S2 Bibliothek [Per17] dar. MongoDB verfügt dabei über zwei grundlegend verschiedene Berechnungsarten. Welche genutzt wird, entscheidet der Geo-Index, der auf einer MongoDB *Collection* liegt und welcher *Query Selektor* genutzt wird (Siehe hierzu Abschnitt 3.3). Im Gegensatz zum planaren Geo-Index werden Berechnungen für einen sphärischen Index nicht intern von MongoDB berechnet, sondern an die S2 Bibliothek delegiert.
Googles S2 Bibliothek wurde am 13. Januar 2011 unter der Apache Licence 2.0 veröffentlicht [Goo17d]. Entworfen und entwickelt wurde S2 von Eric Veach [Pro17]. Der Name stammt von der mathematischen Notation zur Einheitssphäre S^2 [S2G18b].
S2's Kernaufgabe ist dabei Punkte auf der Erde in eine mathematische Sphäre zu übersetzen. Da die Erde jedoch keine richtige Sphäre ist, kommt es bei Berechnungen zu Ungenauigkeiten von bis zu 0,56% [S2G18b]. Berechnungen auf der Erde wären genauer, wenn ein Ellipsoid anstatt einer Sphäre herangezogen wird. Dies steht aber mit zwei der drei Hauptziele von S2 in Konflikt: Performanz und Robustheit. Das dritte Ziel ist die Flexibilität. Berechnungen auf dem Ellipsoid gelten als einige Größenordnungen langsamer als auf der Sphäre. Die Entwicklung robuster geometrischer Algorithmen erfordert die exakte Implementierung von geometrischen Prädikaten, welche keine numerischen Fehler aufweisen. Dies ist bisher nicht für Ellipsoide möglich [S2G18b]. Robustheit besagt, dass es mathematische Garantien gibt, die für jede valide Eingabe gelten. Dies ist besonders wichtig, wenn Algorithmen in einer höheren Ebene implementiert sind und sie von Operationen auf einer niedrigen Ebene abhängig sind (Bspw. Fließkomma-Arithmetik). Flexibilität heißt, dass die Schnittstelle zu S2 groß ist und dem Benutzer soviel Kontrolle wie möglich zu geben. Performanz wird durch die Kantenindex Struktur namens *S2ShapeIndex* gewährleistet. Dies ist ein Index, der im Speicher gehalten wird und die meisten Operationen beschleunigt. Ein Beispiel wurde in [S2G18b] gegeben, dass es nur wenige 100 Nanosekunden benötigt, das Polygon aus über einer Millionen zu finden, welches einen gesuchten Punkt beinhaltet.
S2 wird unter anderem von Google selbst innerhalb von Google Maps genutzt [Per17]. Weitere bekannte Firmen, die S2 nutzen, sind Foursquare [Dis17] und MongoDB Inc. [Inc17x].
Als Grundlage der folgenden Erklärungen und der Umsetzung der inkrementellen Geo-Query Auswertung soll der Quelltext der Java Portierung[2] dienen. In den folgenden Unterkapiteln werden die wichtigsten Strukturen der S2 Bibliothek vorgestellt. Diese werden später als Grundlage zur Berechnung der Distanzen und der Beziehungen zwischen verschiedenen Formen in MongoDB und dieser Arbeit benötigt.

[2] https://github.com/google/s2-geometry-library-java/tree/master/src/com/google/common/geometry

3.1.1 S2-Cell

Die S2 Grundstruktur sind sogenannte Zellen. Sie bilden eine hierarchische Dekomposition der Sphäre zu kompakten Repräsentationen eines Punktes oder einer Region [Pro17]. Sie sind kompakt, weil eine Zelle durch einen 64 Bit Integer Wert eindeutig identifiziert und beschrieben werden kann. Wie im Laufe dieses Abschnitts noch zu sehen sein wird, ist es durch die Anzahl der Zellen möglich, Positionen auf den cm^2 genau zu bestimmen. Im Rahmen dieser Arbeit wird primär von der Erde als konkreter Fall der Sphäre ausgegangen. Die Mathematik kann allerdings auf beliebige andere Sphären übertragen werden [Per17].
Begonnen wird mit einem Punkt auf der Sphäre. Ein Punkt ist definiert durch eine eindeutige zweidimensionale Position. Die beiden Dimensionen werden als Längen (*Longitude*)- und Breitengrade (*Latitude*) bezeichnet. Der Längengrad beschreibt die x Koordinate auf der Erde und der Breitengrad die y Koordinate. Dabei nutzt S2 ein Tupel der Form $(Latitude, Longitude)$ $((y, x)$ im kartesischen Koordinatensystem), als Identifikation eines Punktes. Dies ist die zumeist anzufindende Anordnung und gilt als Standard [Mic18]. Wie noch zu sehen sein wird, gibt es auch Fälle in denen die umgekehrte Reihenfolge genutzt wird.
Der S2 Punkt wird zuerst in einen 3-dimensionalen n-vector überführt [Goo17d].

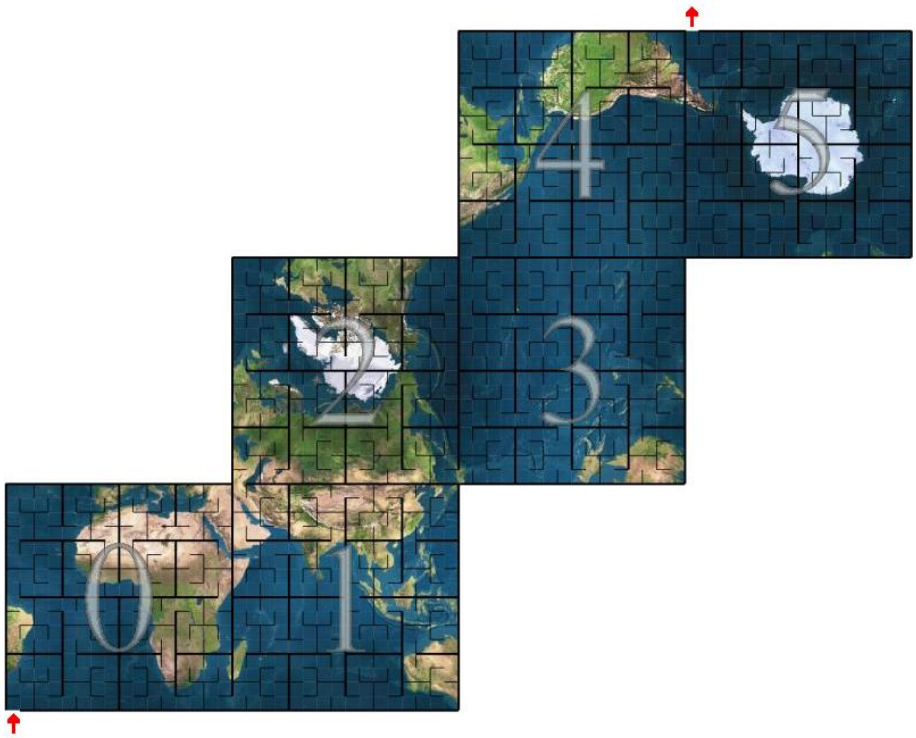


Abbildung 3.1: S2 Erdwürfel [S2G18a]

Die Sphäre wird in einen Würfel projiziert (Siehe Abbildung 3.1). Dadurch entstehen sechs Seiten, die jeweils einen zweidimensionalen Teilausschnitt der Erdkugel repräsen-

tieren. Jede dieser Seiten ist wiederum ein Quadtree, worin der Punkt projiziert wird (Siehe Abschnitt 3.1.1).

Quadtree

Der Quadtree ist eine Baumstruktur, in dem jeder innere Knoten genau vier Kindknoten hat [Sam84]. Er dient der Unterteilung eines zweidimensionalen Raumes. Google hat sich für den Quadtree aus folgenden Gründen entschieden [Pro17]:

1. Eine Auflösung, die gut genug ist, um geografische Merkmale zu indexieren
2. Kompakte Repräsentation einer Zelle
3. Schnelle Methode, um beliebige Bereiche der Erde abzufragen
4. Alle Zellen auf einer Ebene sollten die gleiche Fläche auf der Erde bedecken

Jedoch hat Google keine Informationen genannt, warum sie den Quadtree über andere Methoden bevorzugt haben [Pro17]. Dabei gibt es eine große Vielzahl an Möglichkeiten, räumliche Informationen zu indexieren. Einige Beispiele sind R-Trees [HMTT08], Octrees oder BSP-Trees (Binary Space Partitioning) [Jam03]. Eine mögliche Begründung ist in [KRA02] zu finden. Hierin wurden Quadtrees und R-Trees gegenübergestellt. In diesem Paper sind die Forscher von Oracle zu dem Schluss gekommen, dass Quadtrees in Bereichen eingesetzt werden sollten, in denen viele Update-Operationen auf Basis von grundlegenden Polygon Berechnungen oder eine hohe Zahl an parallelen Update-Operationen auf eine Datenbank ausgeführt werden [KRA02]. Dies sind auch Punkte, die für Google Maps wichtig sind. Daher ist auf Basis dieses Papers, eine Entscheidung Googles für Quadtrees nachvollziehbar.
Microsoft geht bei Bing Maps ähnlich vor wie Google. Allerdings übertragen sie die Erde nicht auf einen Würfel, sondern nehmen die Erde als zweidimensionale Fläche an und unterteilen diese in einen Quadtree [Sch17].

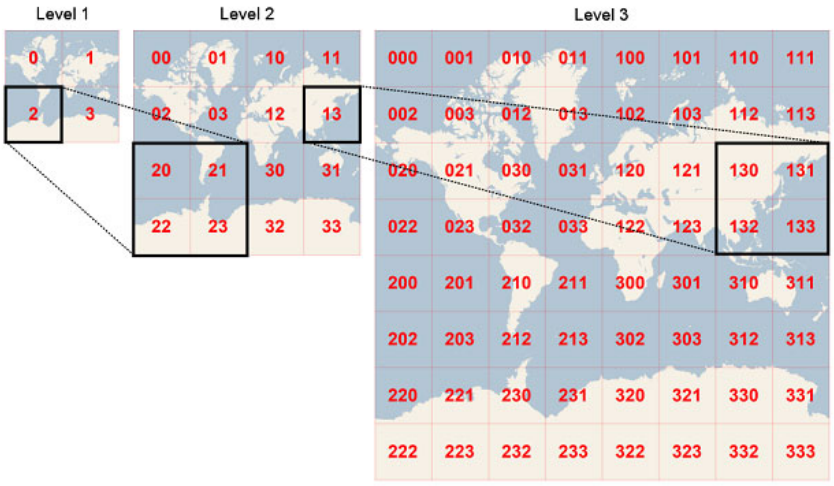


Abbildung 3.2: Bing Maps Quadtree Unterteilung [Sch17]

In dem zuvor referenzierten Beitrag von Joe Schwartz ist Abbildung 3.2 enthalten. Diese zeigt sehr anschaulich, wie die Erde in einen Quadtree projiziert werden kann. In Schritt eins wird die Erde in vier Quadranten geteilt und mit Zahlen von null bis drei indexiert. Jeder dieser vier Quadranten wird im nächsten Schritt durch vier neue Quadranten unterteilt und mit Zahlen von 00 bis 33 indexiert. Dabei repräsentiert die erste der beiden Zahlen weiterhin die gröbere Indexierung aus Schritt 1.

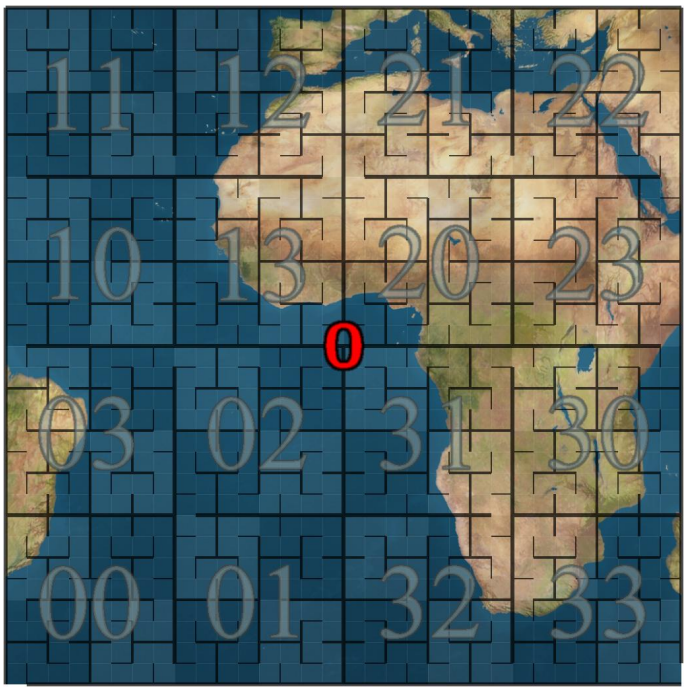


Abbildung 3.3: S2 Gesichtsseite 0 [S2G18a]

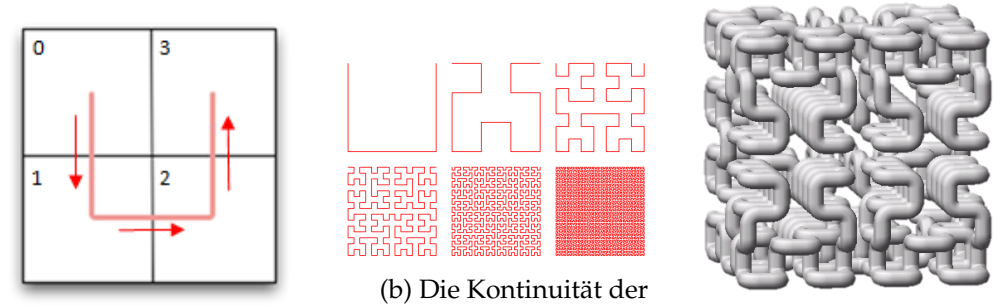
Analog geht auch Google in S2 vor. Abbildung 3.3 zeigt beispielhaft für die erste Gesichtsseite des Würfels, wie dieser in einen Quadtree unterteilt wird. Die Anordnung der Quadranten unterscheidet sich dabei von dem Ansatz, den Microsoft in Bing geht. Die Gründe hierfür werden im späteren Abschnitt zu Hilbert Kurven und 3.3 erläutert.

Nicht lineare Transformation

Es gab allerdings ein Problem mit der Übersetzung in einen Würfel. Zellen, die die gleiche Fläche auf dem Würfel einnahmen, hatten nicht die gleiche Größe auf der Sphäre. Dabei haben sich Zellen bis zu einem Faktor von 5,2 unterschieden [Goo17d]. Aus diesem Grund muss eine nicht-lineare Transformation durchgeführt werden, um den Größenunterschied zu eliminieren. Dabei gibt es drei Möglichkeiten: die lineare Transformation, die tangente Transformation und die quadratische Transformation. Die lineare Transformation hatte keinen Effekt und die Größen der Zellen unterschieden sich weiterhin. Eine tangente Transformation machte die Größen einheitlich, allerdings war diese Methode zu langsam. Es wurde sich für die quadratische Transformation entschieden, da diese fast so gute Ergebnisse wie die tangente Transformation lieferte, allerdings um einiges

schneller ist [Pro17]. Nun fehlt lediglich der letzte Schritt, um eine Koordinate einer Zelle in eine eindeutige Zahl umzuwandeln. Genauer gesagt in eine 64 Bit Ganzzahl. Dies wurde durch die Hilbert Kurve gelöst.

Hilbert Kurven



(a) Hilbert Kurve auf einem 2x2 Gitter [Dat17]

(b) Die Kontinuität der Hilbert Kurve [Dat17]

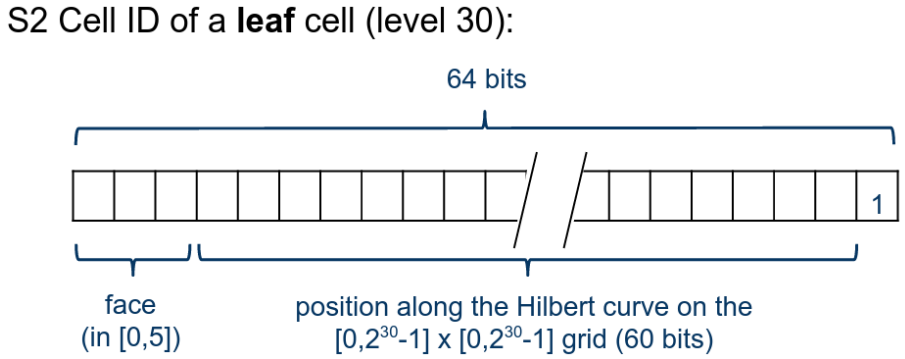
(c) Die Hilbert Kurve im 3-Dimensionalen Raum [Dat17]

Abbildung 3.4: Grafische Darstellung der Hilbert Kurve

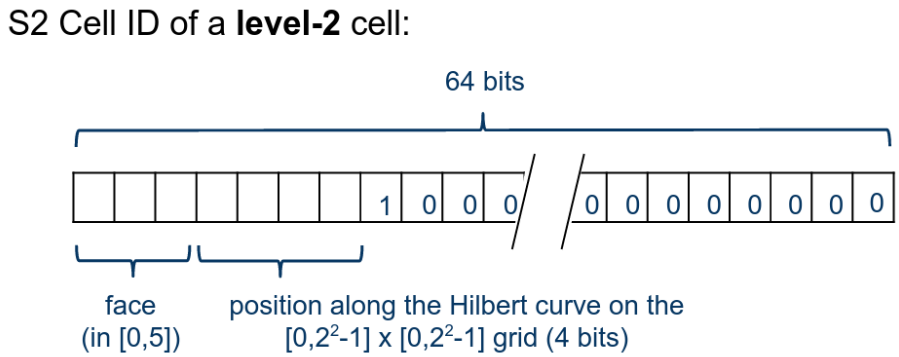
Die Hilbert Kurve ist eine kontinuierliche Linie, die einen gegebenen Raum ausfüllt [Hil35]. Bildlich ist die Hilbert Kurve so zu verstehen, dass ein Faden so auf einem Schachbrett ausgelegt wird, sodass der Faden jedes Feld einmal besucht. Als erster hat David Hilbert diese Kurve 1891 beschrieben [Hil91]. Somit wurde sie nach ihm benannt. Charakteristisch für die Hilbert Kurve ist ihre U-Form (Siehe Abbildung 3.4a). Die U-Form wird rekursiv fortgeführt, um einen Raum auszufüllen. Diese Eigenschaft ist in Abbildung 3.4b auf unsichtbaren Gittern verschiedener Größen zu beobachten. Folgende Eigenschaft ist jedoch von großer Bedeutung für die S2 Bibliothek: Sie kann eine beliebige n-Dimension auf eine Dimension reduzieren. Ausgehend vom Eingangsbeispiel dieses Abschnitts, kann auch ein Maßband in der U-Form über ein Gitter ausgelegt werden. Dabei sei d die angegebene Distanz des Maßbands. Werden nun zwei naheliegende Punkte (x, y) auf dem Gitter gewählt, dann ist auch die Distanz d zwischen den beiden Punkten auf dem Maßband sehr gering. Diese Eigenschaft ist der U-Form der Hilbert Kurve zu verdanken. Da Hilbert Kurven nicht nur auf der zwei-dimensionalen Fläche konstruiert werden können, sondern in jedem n-dimensionalen Raum (Siehe Abbildung 3.4c für ein Beispiel im drei-dimensionalen Raum). So können Koordinaten in jedem n-dimensionalen Raum auf einen 1-dimensionalen Wert reduziert werden und die Distanz zwischen diesen Koordinaten bleibt identisch.
Die Eigenschaft, dass zwei auf der Fläche nah zueinander liegende Punkte auch nah auf dem bildlichen Maßband sind, wird von der S2 Bibliothek für die Zellen ausgenutzt. Zellen werden durch einen 64-Bit Integer dargestellt. Sind dessen Werte nah beieinander, so ist auch die Distanz auf der Sphäre zwischen diesen beiden Punkten gering. Grob gesagt: Die 64-Bit Integer zur Beschreibung der S2 Zellen entspricht dem Wert d auf der Hilbert

Kurve. Dies ist jedoch nicht die einzige Information, die in die Werte kodiert wird.

S2 Zellenstruktur



(a) S2Cell Level 30 Integer Struktur [Pro17]



(b) S2Cell Level 2 Integer Struktur [Pro17]

Level	Min Area	Max Area
0	85,011,012 km^2	85,011,012 km^2
1	21,252,753 km^2	21,252,753 km^2
12	3.31 km^2	6.38 km^2
30	0.48 cm^2	0.93 cm^2

(c) Abgedeckte Fläche einer S2Cell [Pro17]

Abbildung 3.5: S2Cell Level Informationen

Wie eingangs erwähnt wurde, sind Zellen hierarchisch nach ihrer Größe geordnet. Wie in Abbildung 3.5c zu sehen ist, gibt es dabei insgesamt 31 Ebenen (o.a. Level). Diese Level entsprechen denen des Quadtrees aus Abbildung 3.2. Level 0 stellt dabei die größten Flächen dar und Level 30 die Kleinsten. Ebenfalls zu sehen ist, dass die kleinsten Zellen, Punkte mit einer Genauigkeit von weniger als einen cm^2 beschreiben können. Dargestellt ist jede Zelle eindeutig durch einen 64-Bit Integer. Dabei sind mehrere Informationen in den Wert kodiert. Die ersten drei Bits sagen aus, auf welcher Seite des Würfels die Zelle projiziert wurde. Der mittlere Teil ist der Wert d der Hilbert Kurve, um den Punkt zu lokalisieren. Der letzte Teil sagt aus, auf welchem Level sich die Zelle befindet. In Ab-

bildung 3.5a ist zu sehen, dass an letzter Position eine eins steht. Je weiter links die eins rückt, desto kleiner das Level. Da die niedrigsten Level die Oberfläche mit den wenigsten Zellen beschreiben, benötigen diese auch die geringste Distanz auf der Hilbert Kurve, während die Level 30 Zellen die meisten Bits benötigen.
Durch diese kompakte Darstellung als 64-Bit Ganzzahl ist der Speicherbedarf sehr gering. So ist es möglich, mehrere Informationen in einen primitiven Datentyp zu speichern, anstatt in einer Struktur oder Klasse, welche deutlich mehr Speicher benötigt. Auch Berechnungen sind sehr performant. Wird beispielsweise geprüft, ob eine Zelle eine andere schneidet, kann der Vergleich bereits scheitern, wenn die beiden Zellen auf verschiedenen Gesichtsseiten des Würfels liegen. Hierfür müssen lediglich die ersten drei Bits verglichen werden. Auf dem nächsten Level können ebenfalls viele Fälle frühzeitig aussondiert werden. Wenn die beiden Punkte das gleiche Level haben oder die zu beinhaltende Zelle kleiner ist als die, die enthalten sein soll. Wenn diese Überprüfungen erfolgreich verlaufen, muss lediglich geprüft werden, ob der Wert der Hilbert Kurve des enthaltenden Punktes in dem des Enthaltenden ist.
In den folgenden Abschnitten werden S2 Strukturen vorgestellt, die auf den Zellen aufbauen und von großer Bedeutung für die Implementation von MongoDB und damit auch für diese Arbeit sind.

3.1.2 S2Point

In S2 sind Punkte nicht, wie zunächst erwartet, in Latitude/Longitude Angaben gespeichert. Intern werden diese Punkte in 3D-Vektoren auf der Einheitssphäre übersetzt und Berechnungen werden in dieser Darstellung durchgeführt [Goo17e].
Der Grund für diesen Schritt ist, dass es Probleme bei der korrekten Berechnung in der Latitude/Longitude im Bereich der Pole gibt. Folgende Eigenschaften erheben diese Probleme:

- Die Longitude hat eine Singularität an den Polen [Gad10].
- Die Longitude hat eine Diskontinuität am $\pm 180^\circ$ Meridian [Gad10].
- Wird das ellipsoide Erdmodel genutzt, können Berechnungen mit der Latitude/Longitude komplex und approximativ an den Polen werden, da die Erde an den Polen stärker gekrümmt ist, als ein Ellipsoid [Gad10].

Die zuvor genannten Eigenschaften können zu Fehlern in Algorithmen führen, wenn diese nicht korrekt abgefangen und verarbeitet werden [Gad10]. Es gibt einige Berechnungsalternativen, wie UTM und die kartesische Flacherde. Diese geben jedoch ebenfalls approximative Antworten und sind komplex bei großen Distanzen [Gad10].
Aus den zuvor genannten Gründen führte Kenneth Gade die normalen Vektoren ein (Auch genannt n-Vektor), welche eine nicht singulare Repräsentation ist, die sich als sehr performant und präzise im praktischen Gebrauch erweist [Gad10]. In dem Paper wird

angeführt, dass die Nutzung von n-Vektoren einfach ist und exakte Antworten für alle globalen Positionen und Distanzen in der ellipsoiden und des sphärischen Erdmodels zurückgibt.
Da S2 Angaben in $(Latitude, Longitude)$ annimmt, übersetzt S2 diese zuerst in einen 3D-Vektor. Dies wird in S2 wie folgt berechnet. Es sei φ die Latitude und θ die Longitude. Dann ist ein äquivalenter 3D-Vektor wie folgt definiert [Goo17g]:

$$v = \begin{pmatrix} \cos\theta \cdot \cos\varphi \\ \sin\theta \cdot \cos\varphi \\ \sin\varphi \end{pmatrix}$$

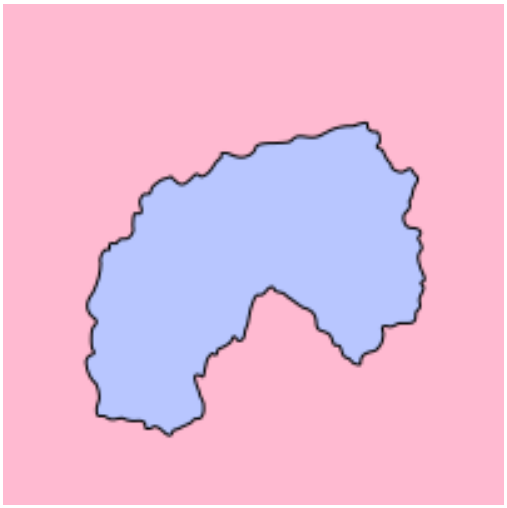
Sind die Latitude/Longitude Angaben erfolgreich in n-Vektoren überführt, ist die Berechnung der Distanz zwischen diesen ebenfalls sehr intuitiv. Nach folgender Formel wird die Distanz berechnet: $atan2(|a \times b|, a \cdot b)$. Diese Formel berechnet den Winkel zwischen zwei Punkten, welche der Distanz zweier Punkte auf der Einheitssphäre entspricht [Goo17f]. S2 gibt dies als finales Ergebnis zurück. Es liegt am Benutzer von S2 diese Distanz in einen realen Wert, wie z.B. Meter auf der Erde, zu übersetzen.

3.1.3 S2Loop

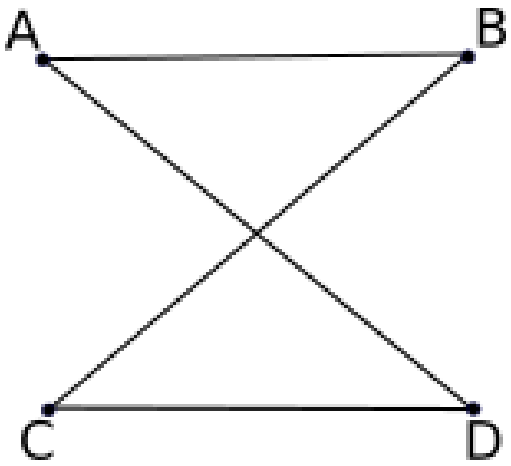
Bei *S2Loop* handelt es sich um die Grundform eines Polygons. Der Name *Loop* (dt. Schleife) ist dabei bereits sehr passend gewählt. Denn es handelt sich in der Tat um eine beliebige Anzahl von Punkten, die eine geschlossene Schleife bilden. Der eingeschlossene Raum ist bei S2 immer Links gesehen von den einzelnen Knoten [Goo17h]. Dabei wird das *S2Loop* entgegen dem Uhrzeigersinn konstruiert. Umgekehrt heißt es, wenn im Uhrzeigersinn eine kleine Fläche gezogen wird, ist der eingeschlossene Raum entgegen dem Uhrzeigersinn sehr groß [Goo17h]. Damit entspricht die *S2Loop* dem Jordanschen Kurvensatz (o.a. „jordan curve theorem“ [Mun99]). Der Satz besagt, dass eine kontinuierliche, sich nicht selbst schneidende Schleife einen Bereich in einen inneren und einen äußeren Bereich trennt.
In Abbildung 3.6a ist eine solche Schleife in Form der schwarzen Linie zu erkennen. Sie schneidet sich nicht selbst und ist in sich geschlossen. Somit trennt sie einen inneren Bereich (gekennzeichnet in blau) und einen äußeren Bereich (gekennzeichnet in rosa). In *S2Loop* wird die Linie entgegen dem Uhrzeigersinn gezogen. Damit ist in diesem Beispiel die blaue Fläche, die die durch *S2Loop* definiert wird.
Abgesehen von dieser Eigenschaft darf kein Knoten in einem *S2Loop* doppelt vorkommen, weder adjazent, noch nicht adjazent. In den folgenden Listings 3.1 und 3.2 sind jeweils ein Beispiel zu diesen beiden Fällen zu sehen.

```
1 [[1,1], [2,2], [2,2], [3,3]]
```

Listing 3.1: Beispiel einer invaliden *S2Loop* mit gleichen adjazenten Knoten



(a) Jordan Curve Theorem [Wik17]



(b) Invalides *S2Loop* mit sich schneidenden nicht adjazenten Kanten

Abbildung 3.6: Jordan Curve Theorem

```
1 [[1,1], [2,2], [3,3], [1,1]]
```

Listing 3.2: Beispiel einer invaliden *S2Loop* mit gleichen nicht adjazenten Knoten

Zudem dürfen sich nicht adjazente Kanten nicht überschneiden. Ein Beispiel für ein solches invalides *S2Loop* ist in Abbildung 3.6b zu finden. Hier ist sehr anschaulich zu sehen, warum es nicht erlaubt sein darf, dass sich Kanten schneiden. So ist der Satz, dass eine innere und eine äußere Fläche getrennt werden, nicht mehr gegeben. Durch das schneiden entstehen immer zwei voneinander getrennte, innere Flächen.
Eine sehr intuitive Regel ist, dass *S2Loops* aus mindestens drei Knoten gebildet werden [Goo17h]. Der Grund ist, dass ein *S2Loop* mit lediglich zwei Knoten nicht geschlossen werden kann. Diese Eigenschaft und die der Knoten-Duplikate werden jedoch nicht erzwungen. Es werden lediglich Warnungen ausgegeben, dass ein *S2Loop* diese Eigenschaften besitzt. Handelt der Benutzer entgegen dieser Richtlinien, können fehlerhafte Werte berechnet werden [Goo17h]. Es liegt somit in der Verantwortung des Nutzers der Bibliothek, dass korrekte Werte und Strukturen an die S2 Bibliothek gegeben werden.
S2Loops werden normalisiert. Das heißt, dass sie anschließend eine Größe von $2 \cdot \pi$ nicht überschreiten und damit keine größere Fläche als die Hälfte der Erdsphäre haben. Überschreitet eine *S2Loop* diese Größe, wird ihre Fläche durch ihr Inverses beschrieben.
In den folgenden Kapiteln werden viele Berechnungen zum Enthaltensein vom *S2Loop* abhängig sein. Aus diesem Grund soll bereits an dieser Stelle geklärt werden, wie ein *S2Loop* andere, für diese Arbeit relevante, S2 Formen beinhalten kann. Bevor S2 die Formen selbst auf Inhalt prüft, wird zuerst eine sogenannte **BoundingBox** auf Inhalt geprüft. Die meisten komplexeren S2 Formen (*S2Point* ausgeschlossen) besitzen zusätzlich zur eigentlichen Form eine *BoundingBox*. Diese ist in allen Fällen ein Objekt der Klasse *S2LatLngRect* und definiert ein Rechteck mithilfe von Latitude und Longitude. Dabei definieren zwei Intervalle die Größe des Rechtecks. Ein Intervall für die x-Richtung und

ein Intervall für die y-Richtung. Die Größe dieses Rechtecks variiert bei den unterschiedlichen Formen und auch durch welchen Konstruktor sie erzeugt wurden. Ein Beispiel, auf welche Weise *S2Loops* in den folgenden Kapiteln besonders oft konstruiert werden, ist über eine Menge von *S2Points*. In diesem Fall wird eine *BoundingBox* mit maximaler Größe erzeugt ([[-90, 90], [-180, 180]]).
Im Folgenden sollen die zwei Enthaltensein-Beziehungen „Punkt in *S2Loop*“ und „*S2Loop* in *S2Loop*“ im Detail erläutert werden, da komplexere Enthaltensein-Beziehungen auf diese Fälle zurückzuführen sind. In S2 wird zuerst geprüft, ob die Form in der *BoundingBox* enthalten ist. Dies ist ein grober und schneller Vergleich, um viele Negativ-Fälle frühzeitig auszuschließen. Für die „Punkt in *S2Loop*“ Berechnung wird sich auf die Kreuzungszahl (o.a. Crossing Number) berufen [Hac62]. Hierfür wird ein Strahl von einem Punkt in eine beliebige Richtung gezogen. Die Zahl der gekreuzten Kanten des *S2Loop*s durch den Strahl bestimmt, ob der Punkt im *S2Loop* ist.
Ist die Zahl der gekreuzten *S2Loop*-Kanten gerade, befindet sich der Punkt außerhalb des *S2Loop*s. Ist die Zahl der gekreuzten *S2Loop*-Kanten ungerade, ist der Punkt im *S2Loop*. Ein Beispiel ist in Abbildung 3.7 zu sehen. Ein Punkt $P1$ ist innerhalb des *Loop*s und ein Punkt $P2$ ist außerhalb des *S2Loop*s. Es werden willkürlich einige Linien von den Punkten aus gezogen. Wie zu sehen ist, gilt für alle Strahlen von $P1$, dass sie eine ungerade Kantenzahl kreuzen. Die von $P2$ aus gezogenen Strahlen kreuzen immer eine gerade Zahl an Kanten. S2 berechnet die Kreuzungen dabei wie folgt: Es nimmt einen willkürlichen Startpunkt bei $(0, 1, 0)$ und zieht einen Strahl S zum Punkt P, der geprüft wird, ob er enthalten ist. Anschließend wird über das *S2Loop* iteriert. Die Kanten werden aus dem nullten und letzten Knoten gebildet und anschließend wird geprüft, ob S von P geschnitten wird. Dies wird dem ersten und nullten Knoten wiederholt etc..

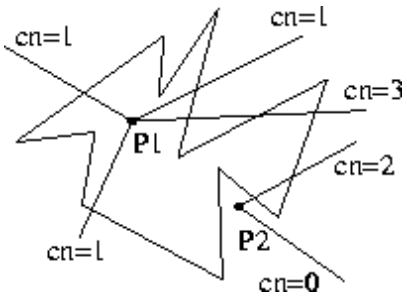


Abbildung 3.7: Beispiel: Kreuszungszahl [Vor10]

Zudem werden einige Sonderfälle geprüft. Zum Beispiel, ob ein Knoten direkt von S geschnitten wird. Die Anzahl der Schnitte wird aufsummiert, um die Kreuzungszahl zu berechnen. Anhand dieser wird schließlich bestimmt, ob der Punkt in der *S2Loop* enthalten ist [gol17b].
Zur Prüfung einer *S2Loop* B in einer *S2Loop* A prüft S2 in der gegebenen Reihenfolge, ob folgende Punkte erfüllt sind [Goo17i]:

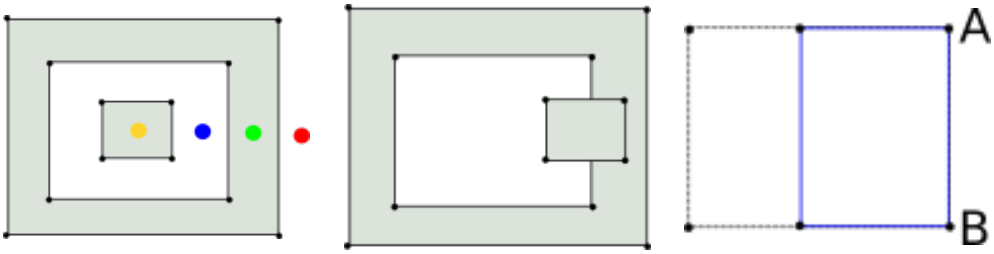
1. Die *BoundingBox* von B ist in der *BoundingBox* von A enthalten.
2. Es gibt keine Schnitte zwischen den Kanten von A und den Kanten von B.
3. Enthalten A und B identische Knoten, muss sichergestellt werden, dass die Fläche die B einschließt in der Fläche die A einschließt, enthalten ist

4. Füllt die Vereinigung der *BoundingBoxes* von A und B die gesamte Sphäre, dürfen keine Knoten von A in B enthalten sein, wenn sie identisch sind

3.1.4 S2Polygon

Das *S2Polygon* ist eine Menge von *S2Loops*, für die einige zusätzliche Regeln gelten. Diese sind im Folgenden aufgeführt [Goo17j]:

1. Ein *S2Polygon* kann null oder mehr *S2Loops* enthalten.
2. Ein Punkt ist im *S2Polygon* enthalten, wenn es in einer ungeraden Zahl von *S2Loops* enthalten ist. Siehe hierzu Abbildung 3.8a. Der rote Punkt ist in null *S2Loops* enthalten und somit in keiner ungeraden Anzahl. Das heißt, dass es auch nicht im *S2Polygon* enthalten ist. Anders beim grünen Punkt. Dieser ist in einem *S2Loop* enthalten (ungerade) und somit im *S2Polygon*. Der blaue Punkt ist wieder in einer geraden Anzahl von *S2Loops* und ist somit in einem Loch. Das heißt der Punkt liegt nicht im *S2Polygon*.
3. Einzelne *S2Loop*s dürfen sich nicht überschneiden. Das heißt die Grenze eines *S2Loops* darf nicht sowohl das Innere eines *S2Loops*, als auch das Äußere enthalten. Ein Beispiel für ein solches invalides *S2Polygon* ist in Abbildung 3.8b zu sehen.
4. *S2Loops* dürfen keine Kanten teilen. Das heißt, wenn eine Kante von Knoten A nach Knoten B in einem *S2Loop* enthalten ist, dann darf weder eine weitere Kante A nach B noch eine Kante von B nach A in einem anderen *S2Loop* enthalten sein. Ein Beispiel für ein solches invalides *S2Polygon* ist in Abbildung 3.8c zu sehen.
5. Keine *S2Loop* darf mehr als die Hälfte der Fläche der Gesamtsphäre enthalten. Dies verhindert, dass keine *S2Loop* das Komplement einer anderen *S2Loop* enthält.
6. Verschiedene *S2Loops* dürfen die gleichen Knoten enthalten.



(a) Darstellung zu Regel 2 (b) Darstellung zu Regel 3 (c) Darstellung zu Regel 4

Abbildung 3.8: Darstellungen zu den Polygon-Regeln

3.1.5 S2Cap

Eine Struktur, die speziell für den *$circleSphere* Query Selektor in MongoDB benötigt wird, ist das *S2Cap*. Diese beschreibt einen sphärischen Deckel. Dabei wird ein Teilausschnitt der Sphäre genommen und an einem bestimmten Punkt durch eine Fläche abgeschnitten. Definiert wird das *S2Cap* durch einen *S2Point* und einer Höhe. Der *S2Point* ist das Zentrum auf der Oberfläche der Einheitssphäre. Daher muss der Punkt selbst auch in Einheitslänge vorliegen. Die Höhe ist die Distanz von dem Punkt zur Fläche, die abgeschnitten wird. Die abgeschnittene Fläche entspricht wiederum einem Kreis. Das *S2Cap* kann somit wie ein Kreis in der planaren Ebene genutzt werden. Der Radius verläuft entlang der Sphäre und ist somit keine gerade Linie [gol17a].

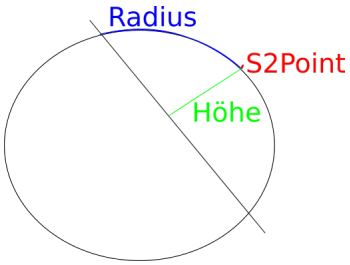


Abbildung 3.9: *S2Cap* Konzept

3.2 GeoJSON

Das Quasistandard-Format zur Angabe von geodätischen Daten im *2dsphere* Index von MongoDB ist das GeoJSON Format [Inc17x]. 2015 formte die Internet Engineering Task Force (IETF) in Zusammenarbeit mit den ursprünglichen Spezifikationsauthoren von GeoJSON eine Arbeitsgruppe, um GeoJSON zu standardisieren. Dabei entstand im August 2016 der RFC 7946 [For17]. GeoJSON ist, wie der Name bereits suggeriert, ein Format basierend auf JSON, um geodätische Informationen in heterogenen Systemen austauschbar zu machen. Dabei können folgende geometrische Objekte dargestellt werden: Punkte, Linien, Polygone, Multipunkte, Multilinien, Multipolygone und Geometrie Kollektionen. Die Grundstruktur jeder Form ist eine zweidimensionale Koordinate. Die Reihenfolge ist mit $(Longitude, Latidude)$ definiert und entspricht somit dem Gegenteil der Anordnung in S2 [For17]. Im Folgenden sollen die Objekttypen kurz mit ihrer dazugehörigen Syntax vorgestellt werden und wie GeoJSON in MongoDB integriert ist.
In MongoDB wird ein GeoJSON Objekt mit dem Operator **$geometry** eingeleitet. *$geoWithin*, *$geoIntersects*, *$near* und *$nearSphere* können diesen Operator enthalten [Inc17r]. Die Standardsyntax für den Inhalt des *$geometry* Operators ist Listing 3.3 zu entnehmen.

```
1 $geometry: {
2    type: "<GeoJSON object type>",
3    coordinates: [ <coordinates> ]
4 }
```

Listing 3.3: Standardstruktur des *$geometry* Operators [Inc17r]

3.2.1 Point

```
1 {type: "Point", coordinates: [179,89]}
```

Listing 3.4: GeoJSON *Point* [Inc17r]

angegeben. Zu beachten ist, dass in MongoDB die Reihenfolge von $(Longitude, Latidude)$ genutzt wird, um konform zur GeoJSON Definition zu sein. Als valide Werte für die Longitude können Fließkommazahlen im Bereich von -180 bis 180 übergeben werden. Für Latitude gilt ein Bereich von -90 bis 90.

3.2.2 LineString

```
1 {type: "LineString", coordinates: [[179, 89], [-179,-89]]}
```

Listing 3.5: GeoJSON *Linestring* [Inc17r]

Ein *LineString* entspricht einer Linie. Die Darstellung ist ein Array mit mindestens zwei Punkten. Dabei gibt es, anders als beim Polygon, keine weiteren Regeln, die eingehalten werden müssen. Besteht ein LineString aus mindestens vier Punkten und ist der erste identisch zum letzten Punkt, wird von einem **LinearRing** [For17] gesprochen. Der *LinearRing* ist nicht explizit im GeoJSON Standard definiert, allerdings bildet es die Substruktur des Polygons. Damit ist es gleichzusetzen mit der *S2Loop* in Bezug zum *S2Polygon*. Der Unterschied ist, dass beim *LinearRing* das erste und letzte Element gleich sein müssen. Somit muss der Ring explizit geschlossen werden, während ein *S2Loop* implizit geschlossen wurde.

3.2.3 Polygon

```
1 {type: "Polygon", coordinates: [[0, 0], [3, 6], [6, 1], [0, 0]]}
```

Listing 3.6: GeoJSON *Polygon* mit einem Ring [Inc17r]

Wie in Listing 3.6 zu sehen ist, handelt es sich bei den internen Arrays eines *Polygon*s um *LinearRing*s. Des Weiteren ist die Struktur des *Polygon*s sehr eng verflochten mit den internen S2 Strukturen *S2Loop* und *S2Polygon*. Intern werden die Knoten eines *LinearRing*s in ein *S2Loop* übersetzt. Wie im vorherigen Abschnitt erwähnt, erwarten *S2Loops* implizit das schließende Glied einer Schleife. Somit musste dieses Element nicht explizit als Teil des Arrays definiert sein. Das heißt, die letzte Koordinate des Arrays musste ungleich der ersten Koordinate sein. In MongoDB Queries wird explizit dieses schließende Glied erwartet, um mit der GeoJSON Definition des *LinearRing*s konform zu sein [Inc17g]. Dies wird vor der eigentlichen Berechnung geprüft und kann zu einem Fehler während der Auswertung führen [Inc17g]. Ebenfalls erlaubt sind gleiche Knoten, die adjazent zueinander sind. Diese werden intern gelöscht (genauso wie der letzte Knoten) bevor sie an die S2 Bibliothek weiter gegeben werden. In S2 mussten mindestens drei Knoten in einem Ar-

ray vorhanden sein. Analog müssen es in MongoDB vier sein, aufgrund des schließenden Endglieds.

```
{
    type : "Polygon",
    coordinates: [[[0, 0], [3, 6], [6, 1], [0, 0]],
                  [ [2, 2], [3, 3], [4, 2], [2, 2]]]
}
```

Listing 3.7: GeoJSON *Polygon* mit mehreren Ringen [Inc17r]

In Listing 3.7 ist ein komplexeres *Polygon* mit einem äußeren und einem inneren Ring gegeben. Der erste definierte Ring ist dabei immer der äußere Ring. Diese Reihenfolge wird in S2 selbst nicht erzwungen [Goo17j]. Die Reihenfolge ist dort willkürlich zu wählen. Bei allen weiteren Ringen ist die Reihenfolge in MongoDB ebenfalls willkürlich. Sie dürfen nur nicht den ersten Ring enthalten. In MongoDB ist zudem nur ein äußerer Ring erlaubt [Inc17q]. Ebenfalls muss mindestens ein Ring im *Polygon* enthalten sein [Inc17q]. *S2Polygon*e durften auch null *S2Loops* besitzen. Für den äußeren Ring gilt, wie auch in S2, dass keine Linie eine andere Linie des Rings schneiden darf [Inc17q]. Anschließend folgen beliebig viele innere Ringe. Die inneren Ringe müssen vollständig im äußeren Ring enthalten sein [Inc17q]. Allerdings dürfen sie eine Kante mit dem äußeren Ring gemein haben [Inc17q]. Schließlich dürfen sich keine internen Ringe schneiden oder überlappen [Inc17q].

3.2.4 Multi-Objekte

Zu den gegebenen Objekten, gibt es jeweils ein Multi-Objekt. Damit sind es drei weitere Objekte: *MultiPoint*, *MultiLineString* und *MultiPolygon*. Der *coordinates* Wert wird bei allen Multi-Objekten aus einem Array der zuvor erläuterten Strukturen erzeugt.

3.2.5 GeometryCollection

Schließlich gibt es noch eine **GeometryCollection**, die ein Array aus beliebigen GeoJSON Objekten ist. In Listing 3.8 ist eine GeometryCollection eines *MutliPoint*s und *Polygon*s definiert.

```
{
  type: "GeometryCollection",
  geometries: [
     {
       type: "MultiPoint",
       coordinates: [
          [ -73.9580, 40.8003 ],
          [ -73.9498, 40.7968 ],
          [ -73.9737, 40.7648 ]
       ]
     },
```

```
12      {
13         type : "Polygon",
14         coordinates : [
15             [[0, 0], [3, 6], [6, 1], [0, 0]],
16             [[2, 2], [3, 3], [4, 2], [2, 2]]
17         ]
18      }
19   ]
20 }
```

Listing 3.8: *GeometryCollection* mit *MultiPoint* und *Polygon* [Inc17r]

3.3 Geografische Anfragen in MongoDB

In diesem Abschnitt sollen die Indices und Operatoren näher vorgestellt werden, die MongoDB im Bereich geodätischer Anfragen bereit stellt. Dabei ermöglicht MongoDB durch seine Indices zwei verschiedene Berechnungsarten. Der *2d* Index berechnet auf dem kartesischen Koordinatensystem und der *2dsphere* Index im geodätischen System. Da die Implementation dieser Arbeit ausschließlich zweiteres unterstützt, liegt der Fokus auf der Beschreibung dieses Indexes und die dafür unterstützten Operatoren. Nichtsdestotrotz werden auch Operatoren, die ausschließlich mit dem kartesischen Index arbeiten, erwähnt. Diese haben jedoch für den weiteren Verlauf der Arbeit keine größere Bedeutung und die Vorstellung dient lediglich der Vollständigkeit.
Es soll ebenfalls auf die Interaktion mit der S2 Bibliothek eingegangen werden und wie die Operatoren die Ergebnisse filtern. Grundlage der Evaluation ist MongoDB in der Version 3.4.

3.3.1 2d Index

Der **2d** Index führt Berechnungen auf einem kartesischen Koordinatensystem durch. Die Punkte werden in klassischen (x, y) Tupeln gespeichert. In MongoDB werden diese auch als *Legacy Coordinate Pairs* bezeichnet [Inc17h]. Die Speicherung als GeoJSON Objekte ist nicht unterstützt [Inc17j]. *2d* Indices unterstützen zudem nur distanzbasierte Berechnungen [Inc17i]. Das heißt, dass der später vorgestellte Query Selektor *$geoWithin* nicht mit dem *2d* Index funktioniert.
Es ist auch nicht möglich den *2d* Index als *Shard Key* zu nutzen [Inc17j]. Sharding ist ein Konzept von MongoDB, welches erlaubt eine *Collection* auf mehrere Maschinen aufzuteilen [Inc18d]. Ziel dieses Konzepts ist es die Leselast pro Maschine zu senken. Dabei werden die Daten anhand ihres *Shard Keys* auf die verschiedenen Maschinen verteilt.
Abschließend ist zu sagen, dass die Berechnung im *2d* Index lediglich eine Approximation ist. Werden Berechnungen im Bereich des Medians oder der Pole ausgeführt, sollte auf den *2dsphere* Index zurückgegriffen werden [Inc17x].

Intern berechnet MongoDB für diesen Index Geohashes, nach denen die Punkte indexiert werden [Inc17a]. Geohashes funktionieren in MongoDB ähnlich zu der Aufteilung in Quadranten von S2 oder Bing mittels Quadtree (Siehe Abbildung 3.2). Dabei beachtet MongoDB die sogenannte *Location Range* [Inc17b]. Diese gibt an, auf welchem Ausschnitt der Erde Punkte eingefügt werden dürfen. Standardmäßig wird die Reichweite der Longitude (-180 inklusive bis 180 exklusive) verwendet. Wird dies nicht eingehalten, wirft MongoDB einen Fehler. Für die Latitude allerdings ist es erlaubt, Punkte außerhalb der erlaubten Reichweite einzufügen. Das Verhalten für diese Punkte ist allerdings undefiniert. Die Reichweite kann nur eindimensional verändert werden (Siehe Listing 3.9).

```
1 db.collection.createIndex( { <location field> : "2d" } ,
2                            { min : <lower bound> , max : <upper bound> } )
```

Listing 3.9: Angabe einer Reichweite des 2d Indexes [Inc17b]

Das erklärt, warum eine Reichweite invalide Werte erlaubt.
Unter Beachtung der Reichweite des Indices, wird die Karte in die besagten vier Quadranten aufgeteilt. Anders als in der Abbildung 3.2 zu sehen, werden die Quadranten mit einem zwei Bit Wert kodiert.
Dabei steht 00 für den unteren linken Bereich, 01 für den oberen rechten, 10 für den unteren rechten und 11 für den oberen rechten Quadranten (Siehe Abbildung 3.10). Ebenfalls identisch zur Grafik, ist die Einführung der Subquadranten durch weiteres Anhängen von zwei Bit Werten um eine höhere Präzision zu erlangen. So werden alle der vier Quadranten erneut in vier Subquadranten geteilt, was zu einer Notation von $0000 - 0011$ für den unteren linken Quadranten führen würde. Dieser Geohash wird zu jedem Dokument gespeichert und hilft beim schnelleren Aussortieren von uninteressanten Punkten. Dabei erfüllt der Geohash im *2d* Index die gleiche Aufgabe, wie die Hilbert Kurve in der S2 Bibliothek.

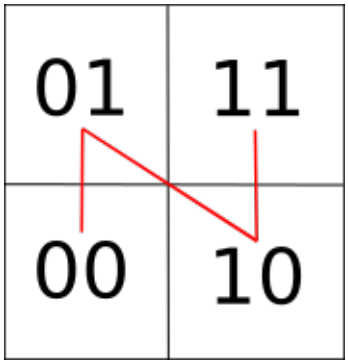


Abbildung 3.10: Geohash Aufteilung in Quadranten

Ein Nachteil von Geohashes ist jedoch, dass die Quadranten nach einer Z-Form abgelaufen werden, während bei der Hilbert Kurve eine U-Form verwendet wird. Siehe hierzu auch Abbildung 3.4a und 3.10. Während wir eine Diskontinuität zwischen 01 und 10 beim Geohash haben, verhindert die U-Form der Hilbert Kurve diese. Der Nachteil wird sichtbar, wenn Distanzen zwischen zwei Punkten berechnet werden. Dadurch, dass die Linie beim Geohash nicht kontinuierlich ist, müssen einige Reichweiten aufgeteilt werden, was in zusätzlichen Abfragen an die Datenbank resultiert. Daher eignet sich die U-Form der Hilbert Kurve besser, da sie keine dieser Diskontinuitäten einbringt. Siehe auch [Joh17] für eine detaillierte Darstellung zu diesem Thema.

3.3.2 2dsphere Index

Der **2dsphere** Index berechnet die Geometrien auf einer erdähnlichen Sphäre. Daten können dabei als *Legacy Coordinate Pairs* oder als GeoJSON Objekte gespeichert werden. Ersteres führt zu einer Konvertierung in ein äquivalentes GeoJSON Objekt [Inc17k]. Bei Distanzberechnungen mit dem *geoNear* Kommando oder der *$geoNear* Pipeline Stage ist es nicht erlaubt, mehr als einen *2dsphere* Index auf der *Collection* gesetzt zu haben [Inc17l]. Diese Restriktion gilt allerdings nicht für die Query Selektoren (*$geoIntersects*, *$geoWithin*, *$near* und *$nearSphere*). Den *2dsphere* Index als Shard-Key verwenden ist ebenfalls nicht möglich [Inc17m].
Werden Angaben in Latitude/Longitude gemacht, müssen diese in umgekehrter Reihenfolge an MongoDB gegeben werden [Inc17s]. Das heißt, die erste Koordinate ist die Longitude und die zweite die Latitude. Grund hierfür ist, dass die Formen intern gleich behandelt werden, unabhängig davon, ob sie über den *2d* oder den *2dsphere* Index kommen. Da die *Legacy Coordinate Pairs* die Koordinaten im klassischen Schema (x, y) entgegen nehmen, muss auch bei Latitude/Longitude zuerst der Wert für die Länge und anschließend der für die Breite übergeben werden. Dies rührt daher, wenn die Erde auf ein kartesisches Koordinatensystem gelegt wird, dann beschreibt die Longitude die x-Koordinate und die Latitude die y-Koordinate. Ein weiterer Vorteil ist, dass diese Anordnung bereits konform dem GeoJSON Standard ist und es somit keiner Umordnung der Koordinaten bedarf (Siehe Abschnitt 3.2).

3.3.3 Query Selektoren

MongoDB unterstützt drei Operatoren zum Anfragen von Datensätzen:

1. *$geoWithin*
2. *$geoIntersects*
3. *$near/ $nearSphere*

Die Datensätze werden im Folgenden kurz als Dokumente bezeichnet, da jeder Datensatz in MongoDB einem JSON Dokument entspricht. MongoDB nennt diese speziellen Zugriffsoperatoren **Query Selektoren** [Inc17t]. Im Rahmen dieser Arbeit wird diese Bezeichnung übernommen.
$geoWithin überprüft, welche Dokumente in einem bestimmten Bereich liegen. *$geoIntersects* prüft, welche Dokumente von der in der Anfrage gegebenen Form geschnitten werden. *$near/ $nearSphere* prüfen, welche Dokumente in einer bestimmten Distanz zu einem gegebenen Punkt liegen. In den folgenden Abschnitten werden diese Query Selektoren detailliert dargestellt, die Syntax zur Anfrage an MongoDB untersucht und es wird beschrieben, wie die interne Implementation dieser Operatoren aufgebaut ist.

$geoWithin

$geoWithin prüft, ob Dokumente vollständig in einer bestimmten Form liegen. Dabei zählt MongoDB den Rand der Form ebenfalls zur Inklusion hinzu [Inc17u]. Formen die andere Formen enthalten können, sind je nach Index verschieden. Im *2dsphere* Index sind *Polygon*e, *MultiPolygon*e und Kreise [Inc17u] unterstützt. Ebenfalls hierzu zählen *GeometryCollection*s, die *Polygon*e oder *MultiPolygon*e enthalten. In der Dokumentation ist dies nicht erwähnt. Jedoch ist die Logik für diese Fälle implementiert [Inc17e]. *Polygon*, *MultiPolygon* und *GeometryCollection* sind dabei Teil der GeoJSON Schnittstelle (*$geometry*), während Kreise separat über den *$centerSphere* Operator bereitgestellt werden [Inc17u]. Im *2d* Index sind *$box*, *$polygon*, *$center* und ebenfalls *$centerSphere* unterstützt [Inc17u]. Auf der anderen Seite können alle Formen in den zuvor erwähnten Formen enthalten sein. Aber auch hier gilt die strikte Trennung zwischen *Legacy Coordinate Pairs* und GeoJSON. Nur Kreise, die mithilfe des *$centerSphere* Operators erzeugt wurden, sind in beiden Indices enthalten [Inc17p].
Das Setzen eines Index ist für die Nutzung von *$geoWithin* Queries nicht notwendig, sie verbessern allerdings die Performance [Inc17u]. Ist kein *2d* Index vorhanden, wird lediglich die für die übergebene Form vorgesehene Berechnung ausgeführt.
Seit Version 2.4 existiert *$geoWithin* als Ersatz für *$within*, welches mit der Version als veraltet eingestuft wurde [Inc17u]. Die grundlegende Syntax von *$geoWithin* ist dabei Listing 3.10 zu entnehmen.

```
{
   <location field>: {
      $geoWithin: {
         $geometry: {
            type: <"Polygon" or "MultiPolygon"> ,
            coordinates: [ <coordinates> ]
         }
      }
   }
}
```

Listing 3.10: Standardstruktur des *$geoWithin* Operators [Inc17u]

Die Koordinaten sind dabei für *Polygon*e und *MultiPolygon*e so anzugeben, wie in Abschnitt 3.2 dargestellt.
$geoWithin ist performanter als die *$near* / *$nearSphere* Operatoren, da keine anschließende Sortierung der Suchergebnisse stattfindet.
*Polygon*e, die einen Ring besitzen, dürfen nicht größer als eine Hemisphäre sein. Sind sie größer als eine Hemisphäre, sucht MongoDB intern nach Dokumenten, die im Komplement der angegebenen Fläche liegen. Dies wird nicht immer gewollt sein. Um *Polygon*e mit einer Fläche größer als einer Hemisphäre richtig nutzen zu können, gibt es die sogenannten **BigPolygon**s. Diese sind eine eigens für MongoDB entwickelte Erweiterung, um den zuvor genannten Spezialfall richtig behandeln zu können [Inc17u]. *BigPolygon*s wer-

den verwendet, wenn ein anderes **Coordinate Reference System** (**CRS**) in der Anfrage angegeben wird. CRS sind eine Sammlung von Regeln, wie Koordinaten einheitlich beschrieben werden können. Dabei gibt es CRS für lokale Spektren, als auch für das globale Spektrum der Erde [BSSL17].

```
{
   <location field>: {
      $geoWithin: {
         $geometry: {
           type: "Polygon" ,
           coordinates: [ <coordinates> ],
           crs: {
              type: "name",
              properties: { name: <crs type> }
           }
         }
      }
   }
}
```

Listing 3.11: Standardstruktur des $geoWithin Operators mit CRS Angabe [Inc17u]

Das in Listing 3.11 angegebene CRS Objekt besitzt ein festes Schlüssel-/ Wert-Paar und einen weiteren Schlüssel *properties*. Dieser enthält einen weiteren Namen, der sich zwischen drei Werten unterscheiden kann.

1. urn:x-mongodb:crs:strictwinding:EPSG:4326
2. urn:ogc:def:crs:OGC:1.3:CRS84
3. EPSG:4326

Das erste aufgeführte CRS ist das von MongoDB eigens spezialisierte CRS für *BigPolygon*s. Dieser benutzt ein entgegen des Uhrzeigersinns gezeichnetes Polygon konform dem Jordanschen Kurvensatz. Bei *Polygon*en, die kleiner sind als eine Hemisphäre, verhält sich dieses CRS exakt, als wenn kein CRS gegeben ist [Inc17u]. Das heißt, intern wird weiterhin die S2 Bibliothek für die Berechnung genutzt. Erst bei Angabe des CRS und *Polygon*en größer einer Hemisphäre wird die Implementation von MongoDB selbst verwendet. S2 würde an dieser Stelle das Inverse des *Polygon*s verwenden (Die Fläche, die durch Ziehen der Linie im Uhrzeigersinn links der Linie entsteht). *BigPolygon* allerdings akzeptiert *Polygon*e dieser Größe und nimmt die Fläche, die links der Linie, die entgegen des Uhrzeigersinns gezeichnet wurde [Inc17c]. Eine weitere Einschränkung von *BigPolygon* ist, dass es nur aus einer Loop bestehen darf [Inc17u]. Ansonsten gelten die gleichen Regeln wie bei einem *S2Polygon*.
CRS Nummer zwei und drei sind ebenfalls im Quelltext angegeben [Inc17f], allerdings existieren noch keine konkreten Implementationen für diese CRS. Das heißt, dass es zum Stand von Version 3.4 noch keinen Unterschied macht, ob „urn:ogc:def:crs:OGC:1.3:CRS84“

oder „EPSG:4326“ als CRS verwendet wird. Die angegebene Form wird in beiden Fällen so behandelt, als wenn kein CRS gegeben ist. Es wird auch kein Fehler geworfen für ein unbekanntes CRS, da diese als Definitionen existieren.
Neben dem *$geometry* Operator gibt es einige weitere **Shape Operator**en, die eine andere Syntax erwarten. Die Bezeichnung *Shape Operator* ist dabei der MongoDB Dokumentation entnommen und beschreibt die folgenden Operatoren innerhalb eines *$geoWithin* Queries [Inc17u]:

- *$box*
- *$polygon*
- *$center*
- *$centerSphere*
- *$geometry*

Diese Bezeichnung wird im weiteren Verlauf der Arbeit für die gleichen Operatoren genutzt. Bis auf *$centerSphere* und *$geometry* handelt es sich um *Legacy Coordinate Pairs*. Aufgrund dessen sind *$box*, *$polygon* und *$center* nicht in *Collections* mit einem *2dsphere* Index erlaubt [Inc17o] [Inc17w] [Inc17n]. Da Orestes lediglich einen Geo-Index anbietet [Baq17a] (Den *2dsphere* Index) und alle Berechnungen auf der Sphäre ausführt, werden diese *Shape Operator*en nicht weiter im Detail beschrieben.

```
1 {
2    <location field>: {
3       $geoWithin: { $centerSphere: [ [ <x>, <y> ], <radius> ] }
4    }
5 }
```

Listing 3.12: Standardstruktur des $geoWithin Operators mit $centerSphere Shape Operator [Inc17p]

$centerSphere allerdings weist zwar die gleiche Syntax auf wie die *Legacy Shape*s, ist aber im *2dsphere* Index nutzbar. Da GeoJSON keine Kreise als Formen anbietet, existiert dieser *Shape Operator* gesondert. Er stellt einen Kreis, bestehend aus einem Punkt und einem Radius dar. Wie in Listing 3.12 zu sehen ist, besteht die *$centerSphere* Struktur, anders als bei *$geometry*, aus einem Array. Das interne Array beschreibt den Punkt. Es folgt eine einzelne positive Fließkommazahl für den Radius. Intern wird der Kreis mithilfe eines *S2Cap* aus Abschnitt 3.1.5 beschrieben. Werden weitere Elemente in diesem Array angegeben, führt dies zu einem Fehler.

$nearSphere

$nearSphere berechnet die Distanz eines gegebenen Punktes zu den in der Datenbank gespeicherten Dokumenten und gibt diese sortiert aufsteigend nach ihrer Distanz zum

Punkt zurück. Dabei können mit *$maxDistance* und *$minDistance* zwei Filter spezifiziert werden, um Dokumente über und unter spezifischen Grenzwerten von der Ergebnismenge auszuschließen. Neben *$nearSphere* gibt es in MongoDB einen weiteren Operator *$near*, der sehr ähnlich in seiner Funktionsweise ist. Während die Berechnung von *$near* vom unterliegenden Index abhängt, ist *$nearSphere* unabhängig. Im Detail heißt das, dass *$nearSphere* unabhängig vom Index die Distanzen geodätisch berechnet, während *$near* je nach unterliegendem Index die dazugehörige Berechnung durchführt. Für Orestes wären *$near* und *$nearSphere* somit identisch, da alle Berechnungen geodätisch durchgeführt werden. Aus diesem Grund wird der *$near* Operator nicht weiter im Detail betrachtet, sondern es wird sich nur auf den *$nearSphere* Operator konzentriert.

```
{
  $nearSphere: {
     $geometry: {
        type : "Point",
        coordinates : [ <longitude>, <latitude> ]
     },
     $minDistance: <distance in meters>,
     $maxDistance: <distance in meters>
  }
}
```

Listing 3.13: Standardstruktur des *$nearSphere* Operators mit GeoJSON Punkt [Inc17v]

Die grundlegende Syntax einer *$nearSphere* Anfrage ist Listing 3.13 zu entnehmen. Die Anfrage ist dabei auf einen Punkt beschränkt. Es sind somit keine anderen Formen erlaubt.

```
{
  $nearSphere: [ <x>, <y> ],
  $minDistance: <distance in radians>,
  $maxDistance: <distance in radians>
}
```

Listing 3.14: Standardstruktur des *$nearSphere* Operators mit *Legacy Coordinate Pair* Punkt [Inc17v]

Alternativ zu einer GeoJSON Struktur sind auch Punkte mit *Legacy Coordinate Pairs* erlaubt (Siehe Listing 3.14).
Eine weitere wichtige Information ist, dass *$maxDistance* immer als Beschränkung eingesetzt werden darf, *$minDistance* allerdings nur, wenn es sich beim darunterliegenden Index um *2dsphere* handelt [Inc18a].
Anders als bei *$geoWithin* wird für *$nearSphere* ein Index benötigt, ohne den der Operator nicht angewendet werden kann [Inc17v]. Als weitere Einschränkung gilt, da der Index gesetzt sein muss, kann nach keinem weiteren Operator gesucht werden, der ebenfalls einen Index benötigt. So können nicht mehrere *$nearSphere* und *$near* Queries kombiniert

werden, aber auch eine Kombination mit dem *$text* Operator für die Volltextsuche ist beispielsweise nicht möglich [Inc17v].
Sind keine weiteren Kriterien zur Suche mit angegeben, sortiert *$nearSphere* automatisch nach der kürzesten Distanz zum gegebenen Punkt aufsteigend. Ist jedoch ein explizites Sortierkriterium angegeben, wird die implizite Sortierung von *$nearSphere* überschrieben. Somit wird es nicht als zweite Sortieroperation mit niedrigerer Priorität genutzt, sondern wird vollständig abgeschaltet [Inc17v].
Es folgt ein Beispielszenario zur Erläuterung der Eigenschaft. In einer Restaurantsuche kann nach der Anzahl der Sterne sortiert werden. Nun gibt es zwei Restaurants mit fünf Sternen in den Ergebnissen. In diesem Szenario wird nicht in jedem Fall das Restaurant höher eingestuft, welches näher am Anfragepunkt liegt.

4 Anforderungen und Umsetzung der Geo-Query Auswertung

In diesem Kapitel wird die konkrete Umsetzung der Geo-Query Auswertung in Orestes beschrieben. Zuerst sollen hierzu die Anforderungen aufgezeigt werden. Anschließend werden die wesentlichen Entscheidungen beschrieben, die bei der Umsetzung aufgetreten sind. Anhand der getroffenen Entscheidungen wird ein detailliertes Bild der endgültigen Implementation gegeben und der dazugehörigen Test-Suite.
Schließlich wird die Logik behandelt, wie das Matching eingehender JSON Objekte gegen Queries vollzogen wird. Dabei werden die beiden Kernaufgaben der Übersetzung von JSON und GeoJSON zu Java Klassen und die Berechnung der geometrischen Beziehungen im Detail beschrieben.

4.1 Anforderungen

Angemeldete Real-Time-Queries werden ebenfalls an MongoDB gegeben (Siehe Abschnitt 2.1). Das erste Ergebnis einer Query-Subscription kommt von MongoDB. Die weiteren Ergebnisse kommen von InvaliDB. Hierzu muss InvaliDB die vollständige Anfragesprache in Bezug auf Geo-Queries verstehen, die Orestes bereits bei regulären Anfragen an MongoDB erlaubt hat. Somit muss die Geo-Komponente nicht das gesamte Verhalten von MongoDB unterstützen, da nur Query Selektoren und Operatoren des *2dsphere* Index erlaubt sind. Allerdings muss die vollständige Anfragesprache verstanden werden, sodass die Komponent mit einer korrekten Fehlerbehandlung reagieren kann, wenn etwas in Bezug auf den *2d* Index angefragt wird. Dadurch können die Operatoren auf den *2d* Index von MongoDB ausgeschlossen werden. Die Implementation konzentriert sich somit komplett auf die sphärischen Berechnungen. Zudem muss InvaliDB sowohl in der Lage sein, Formen als *Legacy Coordinate Pairs* zu analysieren, als auch GeoJSON. Der Grund ist, dass der *2dsphere* Index auch *Legacy Coordinate Pairs* verarbeiten kann, indem er diese zuerst in ein äquivalentes GeoJSON überführt [Inc17k]. Somit muss InvaliDB insgesamt das gleiche Ein- und Ausgabeverhalten besitzen, wie MongoDB. Um dies sicher zu stellen, wird eine umfangreiche Test-Suite bereitgestellt. Der Aufbau der Test-Suite wird in Abschnitt 4.3.3 beschrieben.
Da bis dato nur Operatoren mit Real-Time-Queries möglich sind, die keine sortierte Ergebnismenge benötigen, muss eine Sortierung von Ergebnismengen eingeführt werden. Diese ist essentiell für den *$nearSphere* Query Selektor, da dieser Dokumente aufsteigend nach der Distanz zum Zielpunkt zurückgibt.
Da die Query-Auswertung von Orestes durch die Geo-Queries erweitert wird, sollte auch die Programmiersprache dieser Komponente der von Orestes entsprechen. Dabei handelt es sich um Java. Somit wird auch die Geo-Query Auswertung in Java implementiert, um

keine ungewollte Heterogenität hervorzurufen.
MongoDB gibt einen Großteil seiner sphärischen Berechnungen an S2 weiter. Daher muss evaluiert werden, ob es geeignete Kandidaten in Java gibt. Hierzu wird im folgenden Abschnitt eine Gegenüberstellung zweier möglicher Kandidaten gegeben.
Schließlich muss sowohl die Funktionalität der Geo-Komponente, als auch ihre Performance getestet werden. Die Funktionalität wird anhand einer Proof-of-Concept Applikation geprüft. Diese nutzt die Javascript-Schnittstelle der Baqend-Platform. An ihr wird zum einen die Korrektheit der Berechnungen (wird auch durch die Tests in Orestes sichergestellt) gezeigt, als auch, dass die richtigen *Events* bei einer *Query Subscription* ausgelöst werden. Zur Überprüfung der Performance muss gezeigt werden, dass die Geo-Komponente keinen Flaschenhals besitzt. Dies wird anhand einer Benchmark Applikation geprüft.

4.2 Gegenüberstellung möglicher Geo-Bibliotheken

MongoDB nutzt die in C++ geschriebene S2 Bibliothek für die sphärischen Berechnungen [Inc17x]. Daher muss sich im Umfeld des Java Ökosystems nach einer passenden Alternative umgeschaut werden. Eine Gegenüberstellung fand zwischen zwei Bibliotheken statt. Der erste Kandidat Spatial4j war vielversprechend aufgrund seines Umfangs und der aktiven Weiterentwicklung (Letztes Release: 28. Dezember 2017, Stand Februar 2018) [Loc17]. Wird eine Bibliothek von vielen Menschen genutzt und weiter entwickelt, kann davon ausgegangen werden, dass sie eine gewisse Reife und Stabilität besitzt. Der zweite Kandidat war die Portierung der S2 Bibliothek in Java, die ebenfalls von Google entwickelt wird. Im Folgenden sollen beide Bibliotheken kurz vorgestellt werden, sowie die Punkte, die zur Entscheidung zugunsten von S2 geführt haben.

4.2.1 Spatial4j

Eine der derzeit bekanntesten Geo-Bibliotheken ist Spatial4j mit 114 Forks und 426 Stars auf Github (Stand Oktober 2017) [Loc17]. Zu den Features von Spatial4j zählen neben sphärischen auch euklidische und zylindrische Berechnungsarten. Es kennt alle Formen, die im GeoJSON Standard definiert sind. Zusätzlich werden Kreise und sogenannte *Buffered-Line-Strings* unterstützt. Jedoch kennt es nicht die Multi-Versionen der jeweiligen Formen des GeoJSON Standards. Zudem ist Spatial4j in der Lage, GeoJSON sowohl zu importieren, als auch zu exportieren. Neben GeoJSON sind auch WKT (Well-known-Text) und Polyshape als Formate unterstützt. Mit der Bibliothek ist es möglich, Distanzen und bestimmte weitere Beziehungen zwischen den zuvor genannten Formen zu berechnen. Dazu zählen die *contains*, *within*, *disjoint* und *intersect* Beziehung.
Somit ist die Anzahl der Features, die Spatial4j mitbringt, sehr hoch. Allerdings wird nur ein Teil der Features benötigt. Dazu zählt das Importieren und Exportieren von GeoJSON und die sphärischen Berechnungen auf den GeoJSON Formen. Ein großer Nachteil

ist, dass sphärische Berechnungen auf LineStrings und Polygone bis zu diesem Zeitpunkt nicht unterstützt werden.

4.2.2 S2 Java Portierung

Da von MongoDB als Referenzimplementation ausgegangen wird, ist eine Java Portierung von S2 eine ebenfalls geeignete Wahl. Mit S2 wäre die Implementation sehr nah an der von MongoDB. Somit ist das Risiko geringer, Differenzen im Verhalten zu MongoDB hervorzurufen. Allerdings bietet S2 keine Unterstützung für das Importieren und Exportieren von GeoJSON. Dies wird allerdings auch von MongoDB selbst übernommen. Somit ist es kein Nachteil, wenn dies nicht von der gewählten Bibliothek übernommen werden kann. Der Vorteil, GeoJSON selbst zu übersetzen, ist die erhöhte Kontrolle über den Quelltext. Dadurch ist es möglich sich besonders nah an der Implementierung von MongoDB zu halten und bspw. Fehler bereits während der Übersetzung von JSON zu den Strukturen in Java zu generieren.
Ein wirklicher Nachteil der Portierung ist jedoch, dass dieser nicht vollständig gegenüber der C++ Version ist. Es fehlen einige Funktionen auf LineStrings (Siehe auch Kapitel 7.2). Trotzdem ist der Feature-Umfang in diesem Bereich höher als bei Spatial4j, da Polygone vollständig unterstützt werden.

4.2.3 Entscheidung

Spatial4j würde etwas Programmieraufwand abnehmen, indem es bereits einen fertigen *Importer* und *Exporter* für GeoJSON anbietet. Hierfür wird allerdings Kontrolle über den Quelltext aufgegeben. Somit ist nicht gewährleistet, dass die Implementation exakt das gleiche Verhalten aufweist, wie MongoDB. Zudem ist die Umsetzung der sphärischen Berechnungen noch nicht so weit fortgeschritten, wie es bei S2 der Fall ist.
Weitere Punkte, die für S2 sprechen, sind zum einen die Übertragbarkeit auf den MongoDB Quelltext und dass Google die Portierung ebenfalls selbst entwickelt.
Aufgrund der zuvor genannten Punkte, fiel die Entscheidung auf S2.

4.3 Architektur der Geo-Komponente

Im folgenden Unterkapitel wird ein detaillierter Einblick in die Architektur der Geo-Komponente gegeben. Dabei wird zuerst die Schnittstelle beschrieben, an der die Geo-Komponente ansetzt. Anschließend wird die Komponente selbst beschrieben. Zum Schluss wird die dazugehörige Test-Suite und was sie abdeckt, vorgestellt. Auf der beiliegenden CD befindet sich der vollständige Quelltext der Geo-Komponente und die dazugehörigen Tests als Referenz zu diesem Kapitel.

4.3.1 Die Schnittstelle zum Orestes Projekt

Wie bereits in Abschnitt 2.3 erläutert, wird der Quelltext der Geo-Komponente Teil der Auswertung in Orestes. Diese sind im Paket *orestes-events* zu finden. Die Schnittstelle ist die Klasse *PredicateCache*. Hier wird unterschieden, um welchen Operator (o.a. in diesem Kontext Prädikat) es sich handelt. Einige Beispiele für existierende Prädikate sind: *Exists*, *Regex* oder *GreaterThan*. An dieser Stelle sind jeweils ein Fall für den *$nearSphere* und für den *$geoWithin* Operator integriert. Beide Operatoren werden von jeweils einer Klasse repräsentiert, die von einer abstrakten *Predicate* Klasse erbt. Im Folgenden soll im Kontext der Klasse von Prädikaten gesprochen werden. Wenn es sich um die jeweiligen Syntaxelemente *$nearSphere* und *$geoWithin* als Teil einer Anfrage handelt, sind diese als Query Selektoren bezeichnet.
Beim Aufruf einer der Prädikat-Klassen wird in dessen Konstruktor der übergebene Query im JSON Format in ein *Expression* Objekt für den jeweiligen Query Selektor extrahiert. In diesem Schritt wird der Query analysiert und die wichtigen Informationen für die Auswertung extrahiert. Der Prozess wird detailliert in Abschnitt 4.4 beschrieben. Das *Expression* Objekt wird fortan als Feld der jeweiligen Prädikat-Klasse gehalten. Jede Klasse, die von *Predicate* erbt, implementiert eine *computeMatchingStatus* Methode. In dieser kommt ein in der Datenbank gespeichertes Objekt herein und wird gegen den gehaltenen Query evaluiert. Dabei handelt es sich jedoch nicht um das vollständige Objekt, sondern bereits den Wert, der unter dem Schlüssel liegt, worauf der Geo-Index gesetzt ist. In der *computeMatchingStatus* Methode wird zuerst nur ein boolescher Wert zurückgegeben, ob das Objekt auf den Query zutrifft. Die Sortierung der gefilterten Objekte findet in einem späteren Schritt statt und ist nicht Teil der Geo-Komponente.
Alle Aufrufe aus den beiden Prädikat-Klassen werden in die Geo-Komponente weitergeleitet. In diesem Paket sind die Geo-spezifischen Strukturen und die sphärischen Berechnungen definiert. Bei der Umsetzung des Pakets wurde sehr darauf geachtet, nahe der Implementation von MongoDB zu sein. Dies hat den Vorteil, dass bei möglichem Fehlverhalten ein Vergleich mit der MongoDB Implementation stattfinden kann, um so schneller an die Quelle des Fehlers zu gelangen. Die Architektur dieses Pakets wird im folgenden Kapitel beschrieben.

4.3.2 Die Architektur der Geo-Komponente

In Abbildung 4.1 ist die Paketstruktur der obersten zwei Ebenen der Geo-Komponente dargestellt. In diesem Kapitel werden die Aufgaben der Klassen dieses Pakets beschrieben. Im Ordner *expression* befinden sich die zuvor erwähnten *Expression*-Klassen. Diese werden durch Extraktion aus dem JSON der Queries erzeugt und enthalten die relevanten Informationen für die späteren Berechnungen. Dabei wird die *GeoNearExpression* für die *$nearSphere* Queries benötigt und die *GeoExpression* für die *$geoWithin* und *$geoIntersect* Queries. Beispielsweise extrahiert der *$nearSphere* Query die minimale und maximale

Distanz, als auch den Punkt selbst, gegen den die Distanz berechnet wird.

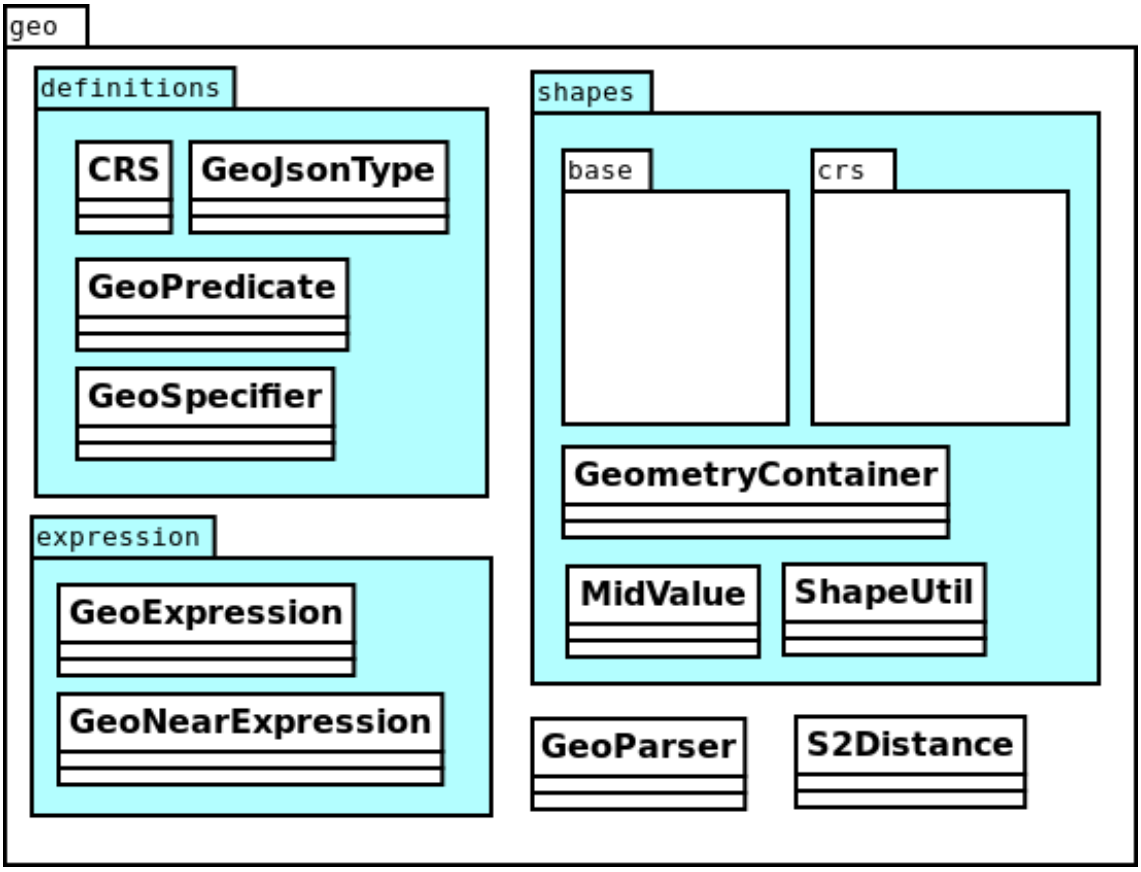


Abbildung 4.1: Geo-Komponente: UML Diagramm

Diese Klassen nutzen unter anderem den *GeoParser* und den *GeometryContainer* des *shapes* Paket. Der *GeoParser* ist eine Klasse, die das Übersetzen spezieller Teilbereiche des JSON übernimmt und eine dazugehörige Objektrepräsentation zurückgibt. Der *GeoParser* selbst hat dabei jedoch keinerlei Kontextwahrnehmung und ist nicht in der Lage, ein vollständiges JSON Dokument in verschiedene Objekte zu übersetzen. Er erwartet die für den Kontext benötigten Substrukturen. Beispielsweise erwartet die *parseGeoJsonPoint* Methode bereits den Inhalt des *$geometry* Operators. Zudem muss vorher bereits bestätigt worden sein, dass es sich beim Typ um einen Punkt handelt, ansonsten würde die Übersetzung in einem Fehler resultieren. Ein Beispiel für ein solches JSON wurde in Kapitel 3.3 gezeigt. Auf dieser Syntax bauen die Orestes Queries ebenfalls auf. Der *GeoParser* ist somit ein Dienstleister, der von diversen anderen Klassen in Anspruch genommen wird, um Teile des Queries in die jeweiligen Objekte zu übersetzen, mit denen diese Klassen wiederum weiter arbeiten können.
Die *GeometryContainer* Klasse ist, wie der Name bereits suggeriert, ein Container für alle Formen, die in MongoDB existieren. Das heißt, dass die Container-Klasse eine beliebige Form repräsentieren kann. Im Kontext von Orestes sind es alle, die kompatibel mit geodätischen Berechnungen sind. Der *GeometryContainer* ist auch eine der Klassen, die

den Kontext herausfinden, in dem sie agieren. Beispielsweise besitzt der *GeometryContainer* eine *parseFromQuery* Methode, die den gesamten Query annimmt. Hier wird zuerst der *GeoParser* genutzt, um den Typ des GeoJSON Objektes herauszufinden. Anhand des extrahierten Typs unterscheidet der *GeometryContainer* anschließend, wie die eigentliche Struktur zu übersetzen ist und ruft für den Prozess wieder den *GeoParser* auf. Das resultierende Objekt wird intern vom *GeometryContainer* gehalten. Ebenfalls kann der *GeometryContainer* Beziehungen zu anderen Containern berechnen. Beispielsweise, ob ein Container in einem anderen enthalten ist. Da bestimmte Formen verschiedene andere Formen enthalten können und bestimmte andere Formen nicht, muss der Vergleich auf dieser Abstraktionsebene stattfinden. Auf die Möglichkeiten wird im späteren Verlauf der konkreten Umsetzung der Operatoren zurückgekommen.
Im Paket *definitions* sind einige Konstanten definiert, die dabei helfen sollen, valide Prädikate, Formen, CRS Definitionen und GeoJSON Typen zu identifizieren. Beispielsweise definiert die Enumeration *GeoJsonType* alle validen GeoJSON Typen. Ist ein Wert keiner dieser Konstanten zuzuordnen, handelt es sich um einen Typ, der nicht verarbeitet werden kann.
Im *shapes* Paket gibt es neben dem Container und den beiden Helferklassen *MidValue* und *ShapeUtil* noch zwei weitere Sub-Pakete. Im Paket *base* befinden sich alle Basis-Formen. Mit diesen kann in der euklidischen Ebene gerechnet werden. Im *crs* Paket liegen Versionen, die eine Basis-Form als Feld besitzen. Zudem haben sie eine CRS Angabe, um zu entscheiden, wie mit ihnen gerechnet wird. Zusätzlich enthalten sie auch eine S2 Version der jeweiligen Form. Mit den geometrischen Formen aus S2 werden die geodätischen Berechnungen durchgeführt.

4.3.3 Die Test-Suite

Die Test-Suite wurde nach einem Bottom-Up Ansatz parallel zur Query-Auswertung entwickelt. Neben der Entwicklung der Datenstrukturen wurden kontinuierlich Unit-Tests nachgetragen. Unit-Tests existieren für alle Klassen, die selbst eine Logik ausführen. Hier soll verifiziert werden, dass die jeweilige Logik korrekt ist. Tests für Klassen, die bestehende S2 Strukturen ummanteln oder ausschließlich Aufrufe an S2 delegieren, wurden nicht geschrieben. Die Korrektheit dieser Komponenten muss durch das S2 Paket selbst verifiziert sein. Die Unit-Tests sind mit dem IntelliJ Coverage Feature entwickelt worden, um sicherzustellen, dass eine möglichst hohe Testabdeckung gewährleistet ist. Diese Abdeckung zeigt an, welche Instruktionspfade durch Tests abgedeckt sind (hervorgehoben in grün) und welche Pfade nicht abgedeckt sind (hervorgehoben in rot). Ein Ausschnitt der Abdeckung ist in Abbildung 4.2 zu sehen. Die Abdeckung wird prozentual zum Quelltext, der durchlaufen wird, gemessen. Je weiter die Abdeckung gegen 100% geht, desto sicherer kann davon ausgegangen werden, dass zumindest jeder mögliche Instruktionspfad durch mindestens einen Test abgedeckt ist. Dies ist allerdings kein Nachweis für Korrektheit der ausgeführten Methoden. Ziel war es trotzdem möglichst viele Instruktionspfade in den Unit-Tests abgedeckt zu haben.

```
public static GeoSpecifier parseGeoSpecifier(JsonElement elem) {
    if (!elem.isJsonObject()) {
        return GeoSpecifier.UNKNOWN;
    }
    JsonObject obj = elem.getAsJsonObject();
    if (obj.has( memberName: "$box"))
        return GeoSpecifier.BOX;
    if (obj.has( memberName: "$center"))
        return GeoSpecifier.CENTER;
    if (obj.has( memberName: "$polygon"))
        return GeoSpecifier.POLYGON;
    if (obj.has( memberName: "$centerSphere"))
        return GeoSpecifier.CENTER_SPHERE;
    if (obj.has( memberName: "$geometry"))
        return GeoSpecifier.GEOMETRY;

    return GeoSpecifier.UNKNOWN;
}
```

Abbildung 4.2: IntelliJ Test-Coverage Feature

Nach Fertigstellung eines Operators wurden weitere Tests hinzugefügt, die einen vollständigen Query mit diesem Operator testen. Anschließend wurde verglichen, ob bestimmte Objekte gegen den Query matchen. Diese Tests sind in den Klassen *GeoWithinQueryTest* und *NearSphereQueryTest* des *orestes-callback* Pakets zu finden. Sie decken simple positiv- und negativ-Tests zum Matching von Queries gegen Objekte ab. Zudem werden auch Grenzfälle getestet, die sich genau auf den äußeren Linien der Formen befinden oder minimal außerhalb der Form.

Schließlich gibt es eine Klasse *EventGeoTest* im *orestes-tests* Paket. Hier wird noch einmal der vollständige Query gegen verschiedene Objekte getestet. Dies jedoch unter realeren Bedingungen. In diesen Tests wird, wie im realen Einsatz auch, eine Subscription für den jeweiligen Query angelegt. Vor, während und nach der Existenz der Subscription werden Objekte zur Datenbank hinzugefügt, entfernt oder modifiziert. Anschließend werden die ausgelösten Ereignisse auf Validität geprüft. In diesem Kontext wurde auch die Sortierung der Ergebnisse beim Einsatz des *$nearSphere* Query getestet. Neben der impliziten Sortierung wird auch das Verhalten beim Hinzufügen einer expliziten Sortierung oder das Zusammenspiel mit dem Einsatz von Limits und Offsets getestet. Auch komplexe

Kompositionen mit verschiedenen Operatoren werden hier geprüft. Beispielsweise wie sich der *$nearSphere* Operator in Kombination mit einem *$regex* Operator verhält.
Wie zu sehen ist, wird von unten nach oben getestet. Zuerst testen Unit-Tests die interne Logik der einzelnen Klassen und deren Zusammenspiel. Anschließend wird auf der nächsten Ebene überprüft, wie das Gesamtverhalten eines Operators ist, indem Queries gegen existierende Objekte auf einen Matching Status geprüft werden. Schließlich wird das Real-Time Verhalten des Geo-Pakets getestet, indem Query-Subscriptions initiiert werden, die einen oder mehrere Geo-Operatoren enthalten. Somit kann in verschiedener Granularität nachgeprüft werden, wenn in der Geo-Komponente etwas nicht funktioniert oder wo es sich nicht wie das MongoDB Pendant verhält. Sollte es eine Stelle geben, die nicht das erwartete Verhalten liefert, kann mithilfe des Bottom-Up Ansatzes schnell nachvollzogen werden, welche Komponente, Klasse oder Methode für das Fehlverhalten verantwortlich ist.

4.4 Übersetzen der relevanten Daten

In Abschnitt 4.3.2 wurde bereits erläutert, wo die Übersetzung des JSON in die Objekte statt findet. In diesem Abschnitt wird detailliert darauf eingegangen, welche Informationen wie übersetzt werden müssen.
Beim Übersetzen wird zwischen zwei Strukturen unterschieden. Den Queries, mit denen eine Prädikatklasse instantiiert wird und die JSON Dokumente, die in der Datenbank gespeichert sind. Die Dokumente werden an die *computeMatchingStatus* Methode gegeben und gegen den Query verglichen. Hierfür müssen sie ebenfalls zuerst in eine Java Klasse übersetzt werden. Anschließend kann der Matching Status berechnet werden. Dies wird Thema der darauffolgenden Abschnitte.

4.4.1 $nearSphere Query

Der JSON Inhalt eines *$nearSphere* Queries wird in der Klasse *GeoNearExpression* übersetzt. In Abbildung 3.13 und 3.14 sind zwei Beispiele für die eingehende JSON Struktur zu sehen. Das innere JSON Objekt unter dem Schlüssel *$nearSphere* wird von der Expression-Klasse weiter verarbeitet. *$nearSphere* ist in der Lage, unterschiedliche Strukturen für *Legacy Coordinate Pairs* und GeoJSON Formen zu verarbeiten. Zwar wurde entschieden, sich vollkommen auf den *2dsphere* Index zu konzentrieren, trotzdem ist dieser auch in der Lage, *Legacy Coordinate Pairs* einzulesen und diese in GeoJSON Punkte zu übersetzen [Inc17k]. Der erste Schritt ist somit herauszufinden, ob es sich um *Legacy Coordinate Pairs* oder eine GeoJSON Struktur handelt. Die minimale und maximale Distanz können einheitlich extrahiert werden, ungeachtet des Formats des eigentlichen Punktes. Der Punkt selbst wird einheitlich übersetzt. Lediglich der Ort, an dem die Koordinaten liegen, unterscheiden sich aufgrund der Strukturen. Der JSON Punkt wird in eine *PointWithCRS* Klasse übersetzt, welcher als **Centroid** bezeichnet wird. Dieser enthält

ein Feld *oldPoint* für den Punkt in der euklidischen Ebene. Hierin wird die Longitude als Wert für x und die Latidude als Wert für y gesetzt. Das ebenfalls enthaltene CRS wird auf *FLAT* für einen Punkt in der euklidischen Ebene oder auf *SPHERE* für die Sphäre gesetzt. Zudem sind ein S2Point und eine S2Cell enthalten, die bei *Legacy Coordinate Pairs* Koordinaten zuerst unbelegt bleiben. An dieser Stelle wird auch geprüft, ob es sich um valide Werte für die Latitude und Longitude handelt. Ist die Latitude oder Longitude invalide oder enthält das JSON andere Elemente als die Folgenden:

- *$nearSphere* (oder Synonyme: *$near* und *$geoNear*)
- *$geometry*
- *$minDistance*
- *$maxDistance*

dann wird die Übersetzung mit einem Fehler abgebrochen.
Nachdem verifiziert wurde, dass die Latitude und Longitude valide sind, ist es anschließend möglich, euklidische Punkte in die Sphäre zu übersetzen. Hierbei wird zuerst das CRS im Centroid auf *SPHERE* gesetzt. Anschließend werden die Latitude und Longitude wie folgt von der Grad Angabe in Radianten übersetzt: $Grad \times \frac{180}{\pi}$ [Goo17c]. Nach der Übersetzung findet eine Normalisierung statt, damit die Punkte weiterhin in einer validen Latitude und Longitude liegen. Die Normalisierung wird wie folgt in S2 durchgeführt [Goo17o]:
$Latitude = max(\frac{\pi}{2},\ min(\frac{\pi}{2},\ Latidude))$
$Longitude = Longitude\ \%\ (2 \times \pi)$
Anschließend wird die resultierende Latitude/Longitude in einen n-Vektor übersetzt. Die Übersetzungsprozedur ist in Abschnitt 3.1.2 erläutert. Als letzter Schritt wird der Punkt in eine *S2Cell* projiziert. Die Projektion ist in Abschnitt 3.1.1 erläutert. Der resultierende *S2Point* und die *S2Cell* werden in den *PointWithCRS* gespeichert, welche für die spätere Distanzberechnung in Abschnitt 4.5 benötigt werden.

4.4.2 $geoWithin Query

Die *$geoWithin* und *$geoIntersect* Query Selektoren werden mithilfe der *GeoExpression* Klasse übersetzt. Wobei *$geoIntersect* selbst noch nicht weiter verarbeitet wird (Siehe Abschnitt 7.1). Der Grund dafür, dass mit der *GeoExpression* Klasse zwei unterschiedliche Query Selektoren übersetzt werden können ist, dass die Strukturen des übergebenen JSON identisch sind. Beide Query-Selektoren enthalten einen *$geometry* Shape Operator oder Formen mit Legacy Coordinate Pairs (*$box*, *$polygon*, *$center*, *$centerSphere*). Hier wird zuerst unterschieden, um welchen Shape Operator es sich handelt. Anschließend wird die Form entsprechend in das dafür vorgesehene Objekt aus dem *shapes* Paket übersetzt. Während für die Shape Operatoren *$geometry* und *$centerSphere* bereits die dazu-

gehörigen S2 Klassen befüllt werden, wird für die restlichen Legacy Operatoren das dazugehörige Legacy Objekt belegt. Auf die Legacy Formen soll im Folgenden nicht weiter eingegangen werden, da hier die Formen nach simpler Geometrie mit den Werten befüllt werden.
Da GeoJSON keine Form für Kreise anbietet, gibt es den *$centerSphere* Shape Operator. Ausgehend von der Syntax, sieht der *$centerSphere* Query aus wie ein Query, der intern euklidische Berechnungen durchführt. Intern wird das Enthaltensein jedoch mithilfe von S2 berechnet. Im Folgenden wird zuerst die Übersetzung des *$centerSphere* Shape Operators beschrieben. Anschließend wird die Übersetzung GeoJSON Typen erläutert. Nach diesen Schritten muss geprüft werden, ob in der Form andere Formen enthalten sein können. Die Formen die zu dieser Gruppe zählen sind:

- *Polygon*
- *MultiPolygon*
- *Cap*
- *GeometryCollection* mit *Polygon* oder *MultiPolygon*

Erfüllt die Form diese Bedingung ist die Übersetzung erfolgreich. Ansonsten wird ein Fehler während der Verarbeitung geworfen.

$centerSphere

Ein grober Überblick über den MongoDB Shape Operator wurde bereits in Abschnitt 3.3.3 gegeben. Syntaktisch ist dieser Operator identisch zum *$center* Shape Operator. Dementsprechend werden beide identisch übersetzt. Die *CapWithCRS* Klasse ist die Zielstruktur, in die die Informationen übersetzt werden. Sie besteht aus einem euklidischen Kreis und einem *S2Cap*. Der euklidische Kreis wird mit den Werten befüllt, wie sie in der Anfrage übergeben wurden. Für die Übersetzung in ein *S2Cap*, muss der Centroid des euklidischen Kreises zuerst in einen *S2Point* übersetzt werden. Dies geschieht über die n-Vektor Transformation. Der Radius des euklidischen Kreises muss in Radianten übersetzt werden. Anschließend wird aus dem *S2Point* und dem Radius das *S2Cap* wie folgt berechnet. Die Achse entspricht dem *S2Point* und die Höhe wird nach der Formel: $2 \times sin(\frac{Winkel}{2}^2)$ berechnet. Diese Berechnungsart hat eine höhere Genauigkeit, als $1 - cos(Winkel)$. Da $cos(Winkel)$ bei Winkeln gegen null immer eins ergibt [Goo17n].

$geometry

Die zweite Möglichkeit ist, dass ein GeoJSON Objekt im Query enthalten ist. In diesem Fall wird zuerst der Typ und das CRS (wenn gegeben) extrahiert. Ist kein CRS vorhanden, wird es auf *SPHERE* gesetzt. Anhand dieser Information werden die Koordinaten unter

Beachtung der in Abschnitt 3.2 definierten Regeln in eines der dazugehörigen CRS Objekte übersetzt. Dabei kann jede Form auf mehrere Punkte reduziert werden. Die Übersetzung der Latitude und Longitude dieser Punkte findet anschließend analog zur Beschreibung aus Abschnitt 4.4.1 statt. Ist das strikte CRS
„urn:x-mongodb:crs:strictwinding:EPSG:4326“ gegeben, wird das enthaltene Polygon mit den strikteren Regeln aus Abschnitt 3.3.3 zu einem *BigPolygon* übersetzt.

4.4.3 Datenbank Objekte

Bei Objekten aus der Datenbank kann es sich um drei verschiedene Arten handeln. Formen mit *Legacy Coordinate Pairs*, GeoJSON oder ein von Orestes eigens bereitgestellter Typ namens *GeoPoint*. Alle Typen kommen als JSON in die *computeMatchingStatus* Methode und müssen ebenfalls zuerst in einen *GeometryContainer* übersetzt werden, bevor das Prädikat den Matching Status berechnen kann. Die Übersetzung der ersten beiden Fälle ist analog zu den Verfahren, die in den vorherigen Abschnitten dargestellt wurden. Der *GeoPoint* besitzt zwei Felder namens *latitude* und *longitude*. Beide Werte werden extrahiert und anschließend in den *oldPoint* des *PointWithCRS* gespeichert. Als CRS wird der Wert *FLAT* gesetzt. Anschließend muss der Punkt ebenfalls in die Sphäre überführt werden. Das Verfahren ist in Abschnitt 4.4.1 beschrieben worden.

4.5 Berechnung des Matchings für den $nearSphere Query Selektor

Für den *$nearSphere* Query Selektor wird die Distanz nach der in Abschnitt 3.1.2 vorgestellten Formel berechnet. Anschließend wird das Ergebnis mit dem Wert $6378,1 * 1000$ multipliziert. Da n-Vektoren intern auf Einheitslänge basieren [Gad10], muss das errechnete Ergebnis auf die Größe der Erde skaliert werden. MongoDB benutzt hierfür den äquatorialen Radius von 6378,1 [Alm17]. Anschließend wird das Ergebnis mit 1000 multipliziert, um die Distanz in Metern zu überführen. Die *$minDistance* und *$maxDistance* sind ebenfalls in Metern anzugeben. Der Matching Status liegt vor, wenn der errechnete Wert zwischen oder gleich der unteren (*$minDistance*) und oberen (*$maxDistance*) Schranke ist.
Beim Punkt muss lediglich die Distanz berechnet werden. Wird die Distanz zwischen einer anderen Form und dem Controid des Queries berechnet, wird bei der *Polyline* und dem *Polygon* die minimale Distanz verwendet. Hierfür wird über jeden Knoten iteriert und geprüft, welcher die minimale Distanz zum Centroid des Queries besitzt. Beim *$centerSphere* errechnet sich die Distanz aus der Differenz zweier Winkel. Der erste beschreibt die Distanz zwischen der S2Cap Achse und dem Centroid. Der zweite ist der Öffnungswinkel der aufgespannten Kuppel. Dieser wird wie folgt berechnet: $2 \times asin(\sqrt{0.5 \times \mathit{Höhe}})$ [Goo17k]. Die Differenz dieser beiden Winkel entspricht der Distanz zwischen dem Centroid und dem Rand des Kreises. Somit wird bei allen Formen die minimale Distanz

zwischen der Form und dem Centroid gesucht. Diese ist Gegenstand der zukünftigen Vergleiche mit *$minDistance* und *$maxDistance*.
Bei den Multi-Formen und der *GeometryCollection* wird ähnlich vorgegangen. Hier wird durch jedes Element iteriert und die kürzeste Distanz über alle Subformen wird als Distanz zum Centroid verwendet. Auch bei der *GeometryCollection* wird über jede der unterschiedlichen Formen iteriert und geprüft, welche die niedrigste Distanz aufweist.

4.6 Berechnung des Matchings für den $geoWithin Query Selektor

Für *$geoWithin* müssen eine viel höhere Zahl an Fällen abgedeckt werden. Während es nur eine geringe Zahl an Formen gibt, die andere Formen enthalten können, kann jede Form enthalten sein. Zu denen, die andere enthalten können, zählen *Polygon*e, *BigPolygon*e, *MultiPolygon*e und *GeometryCollection*s. Dabei überprüft die *contains* Methode des *GeometryContainers* der Reihe nach, welche Form im übergebenen *GeometryContainer* enthalten ist und führt dementsprechend die Berechnung durch. Da Formen, wie *MultiPolygone* und *GeometryCollection*s Methoden wiederverwenden, die bereits auch im *Polygon* zur Bestimmung auf Inhalt verwendet wurden, beschreibt diese Arbeit diese nicht detaillierter in den folgenden Unterkapiteln.
Für Kreise (*$centerSphere*) [Inc17p], ist es lediglich möglich Punkte in diesen zu finden [Ret17]. Für alle weiteren Formen liegt in Version 3.4 von MongoDB keine Logik vor und die Abfrage liefert somit keine Ergebnisse, auch wenn die Form im Kreis enthalten wäre. Der Fehler wurde inzwischen behoben und erhält mit Version 3.6 Einzug in MongoDB [Ret17]. Diese Arbeit baut jedoch auf die Implementation von 3.4 auf und das Ein- und Ausgabeverhalten soll dahingehend analog zu dem von dieser Version sein. Aus diesem Grund wurde sich dafür entschieden ebenfalls nur das Enthaltensein von Punkten zu berücksichtigen.
In der folgenden Tabelle wird ein Überblick geschaffen, welche Formen, welche anderen Formen enthalten können. Auf umfangreichere Beschreibungen wird durch Referenzen auf das entsprechende Unterkapitel verwiesen.

Formen / Enthaltene Formen	*Polygon*	*BigPolygon*	*MultiPolygon*	*Geometry Collection*	*centerSphere*
Punkt	Siehe Abschnitt 4.6.1	Siehe Abschnitt 4.6.1	In einem der *Polygon*e enthalten	In einem der Formen enthalten	Siehe Abschnitt 4.6.1

LineString	Nicht von S2 implementiert	Nicht von S2 implementiert	Muss in einem der *Polygon*e enthalten sein	Muss in einem der Formen enthalten sein	Nicht möglich bis MongoDB 3.6
Polygon	Siehe 4.6.2	Siehe 4.6.2	Vergleich aus 4.6.2 bis das erste *Polygon* zutrifft	Vergleich aus 4.6.2 bis die erste Form zutrifft	Nicht möglich bis MongoDB 3.6
MultiPoint	Jeder Punkt muss im *Polygon* enthalten sein	Jeder Punkt muss im *BigPolygon* enthalten sein	Jeder Punkt muss in einem der *Polygon*e enthalten sein	Jeder Punkt muss in einer der Formen enthalten sein	Nicht möglich bis MongoDB 3.6
MultiLine String	Jedes *LineString* muss im *Polygon* enthalten sein	Jedes *LineString* muss im *BigPolygon* enthalten sein	Jedes *LineString* muss in einem der *Polygon*e enthalten sein	Jedes *LineString* muss in einem der Formen enthalten sein	Nicht möglich bis MongoDB 3.6
MultiPolygon	Jedes der *Polygon*e des *MultiPolygon*s muss im *Polygon* enthalten sein	Jedes der *Polygon*e des *MultiPolygon*s muss im *BigPolygon* enthalten sein	Jedes *Polygon* des *MultiPolygon*s muss in einem der *Polygon*e des Query *MultiPolygons* enthalten sein	Jedes *Polygon* des *MultiPolygons* muss in einem der Formen der *GeometryCollection* enthalten sein	Nicht möglich bis MongoDB 3.6

Geometry Collection	Jede Form der *GeometryCollection* muss im Polygon enthalten sein	Jede Form der *GeometryCollection* muss im *BigPolygon* enthalten sein	Jede Form der *GeometryCollection* muss in einem der *Polygon*e des *MultiPolygons* enthalten sein	Jede Form der *GeometryCollection* muss in einer Form der Query *GeometryCollection* enthalten sein	Nicht möglich bis MongoDB 3.6

4.6.1 Punkt

Ein Punkt kann auf mehrere Arten in einem Polygon enthalten sein. Um in einem Polygon enthalten zu sein, muss der Punkt selbst im *S2Loop* vom *S2Polygon* enthalten sein. Besteht das Polygon aus mehreren *S2Loop*s, muss sich der Punkt in einem der ungeraden *S2Loop*s befinden, da es sich ansonsten um ein Loch handelt, in dem sich der Punkt befindet. Eine Vorprüfung, ob der Punkt überhaupt im *BoundingBox* des Polygons ist, kann klare Fälle bereits frühzeitig ausschließen. Eine zweite Prüfung findet mit der *S2Cell* des Punktes statt, schneidet die *S2Cell* das Polygon, ist der Punkt ebenfalls enthalten. Diese Prüfung ist zwar langsamer, als die Prüfung des Enthaltensein des Punktes selbst [Inc17d], allerdings werden hier auch die Knoten und Kanten des Polygons berücksichtigt.
Beim *BigPolygon* findet eine vereinfachte Prüfung statt. Da das *BigPolygon* aus maximal einem *S2Loop* bestehen darf, wird auch lediglich geprüft, ob der Punkt in diesem enthalten ist. Zur Wiederholung ist das Enthaltensein eines Punktes in einem *S2Loop* in Abschnitt 3.1.3 beschrieben.
Ein Punkt ist in einem *$centerSphere* (oder genauer einem *S2Cap*) enthalten, wenn die *S2Cell* des Punktes im *S2Cap* enthalten ist oder sie schneidet. Hierzu werden die vier Knoten der *S2Cell* des Punktes genommen und für jeden wird geprüft, ob dieser im *S2Cap* enthalten ist. Das Enthaltensein von einem dieser Punkte wird durch folgende Formel ermittelt [Goo17m]: $(Achse - Knoten\ der\ S2Cell)^2 <= 2 * Höhe\ des\ S2Cap$
Trifft dies für keinen der Punkte zu, wird geprüft, ob einer der Punkte das *S2Cap* schneidet. Hier wird zuerst getestet, ob das *S2Cap* eine Hemisphäre oder größer ist (Höhe >= 1). Ist dies der Fall, sind die *S2Cell* und das Komplement des *S2Cap* konvex. Dadurch ist keiner der Knotenpunkte enthalten und somit auch kein anderer Punkt der *S2Cell* [Goo17l]. In diesem Fall wäre die *S2Cell* nicht im *S2Cap* enthalten. Ist dies nicht der Fall, wird nun geprüft, ob die *S2Cell* die Achse enthält. Ist dies ebenfalls nicht der Fall, kann nur noch ein Innerer Punkt der *S2Cell* das *S2Cap* schneiden. Trifft all das nicht zu, ist der Punkt nicht im *S2Cap* enthalten.

4.6.2 Polygon

Damit ein Polygon ein anderes Polygon enthält, müssen alle Punkte die Polygon A enthält auch in Polygon B enthalten sein. Der einfachste und zugleich am meisten in der Praxis auftretende Fall [Goo17j] ist, dass beide Polygone eine *S2Loop* besitzen. Hier wird lediglich geprüft, ob *S2Loop* B in *S2Loop* A enthalten ist. Zur Wiederholung der Bestimmung, ob ein *S2Loop* in einem anderen enthalten ist, siehe Abschnitt 3.1.3.
Anschließend findet die *BoundingBox* Prüfung statt. Ist die *BoundingBox* von Polygon B nicht in A enthalten, kann es trotzdem noch einen speziellen Fall geben, dass die Polygone ineinander enthalten sind. Hierfür muss die Vereinigung der beiden *BoundingBox*en die gesamte Longitude ausfüllen und eines der Polygone muss aus mehreren Hüllen bestehen [Goo17j]. Der anschließend einfachste Fall ist, dass beide Polygone keine Löcher besitzen. In diesem Fall muss jedes *S2Loop* von B in einem *S2Loop* von A enthalten sein. Der komplexeste Fall ist, wenn beide Polygone Löcher enthalten können. In diesem Fall wird geprüft, ob Polygon A alle Hüllen von B enthält und A keine Löcher von B schneidet.
Beim *BigPolygon* machen sich die MongoDB Entwickler eine Eigenschaft von S2 zunutze, um die Berechnung auf einem Polygon auszuführen, welches größer als eine Hemisphäre ist. Ist ein *S2Loop* größer als eine Hemisphäre, wird es invertiert. MongoDB fragt ab, ob das *S2Loop* invertiert wurde und weiß damit, ob das originale *S2Loop* größer als eine Hemisphäre war. Ist dies der Fall, wird abgefragt, ob das resultierende *S2Loop* das übergebene Polygon schneidet (Zwei Polygone schneiden sich, wenn ein Punkt von Polygon A in Polygon B enthalten ist). Ist dies nicht der Fall, ist es gleichbedeutend damit, dass es im großen Polygon enthalten ist. Wurde das *S2Loop* nicht invertiert, kann die Implementation für ein *S2Polygon* angewendet werden.

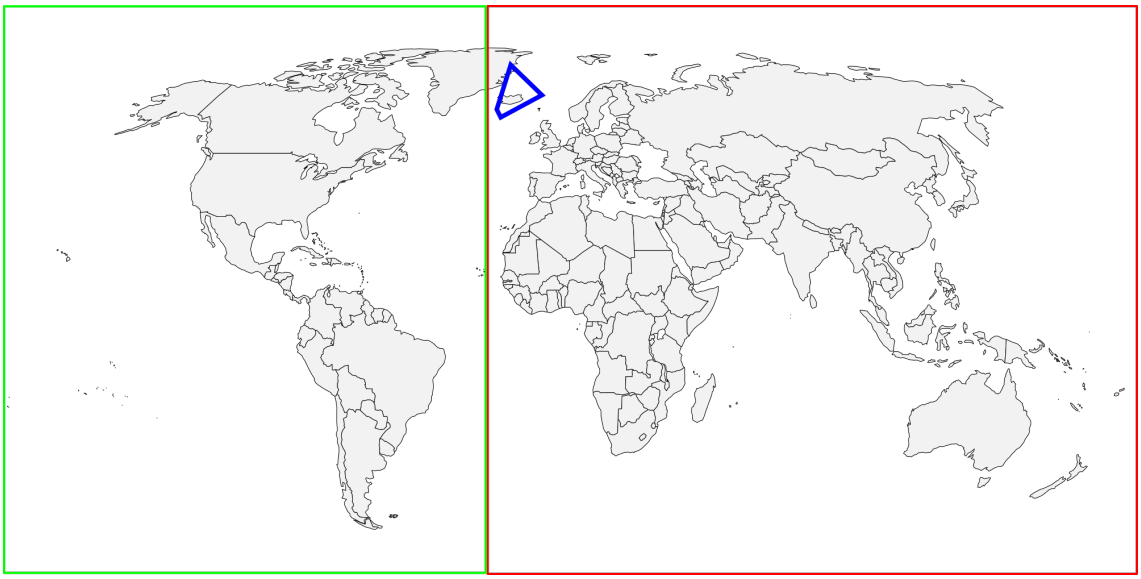


Abbildung 4.3: BigPolygon Enthaltensein (Karte entnommen von [Par17]).

In Abbildung 4.3 ist ein Beispiel für den Spezialfall des *BigPolygon*s gegeben. Es wird abgefragt, ob das blaue Polygon im roten *BigPolygon* enthalten ist. Das Rote wird durch S2 zum grünen Polygon invertiert. Nun wird abgefragt, ob das blaue Polygon mindestens einen Punkt im grünen hat (Sie sich schneiden). Da dies nicht der Fall ist, ist das blaue Polygon im Roten enthalten und der *$geoWithin* Query würde somit das blaue Polygon in der Ergebnismenge zurückgeben.

5 Evaluation der Geo Query Komponente anhand einer Baqend Anwendung

In diesem Kapitel wird mithilfe der von Baqend angebotenen Javascript Schnittstelle (Siehe 2.4) eine Applikation erstellt, die Geo-Real-Time-Queries in einem realen Kontext verwendet. Hieran soll zum einen gezeigt werden, dass der in dieser Arbeit entwickelte Ansatz für Geo-Real-Time-Queries funktioniert. Zum anderen soll auch die Performanz und Skalierbarkeit der entwickelten Komponente untersucht werden, soweit es im Rahmen dieser Arbeit möglich ist.

5.1 Problemstellung

Es wird eine Applikation für die Baqend Plattform (Siehe Abschnitt 2.4) mithilfe der Real-Time API entwickelt. Die *Proof-of-Concept* Applikation wird als Lösung für ein realistisches (o.a. *real world*) Problem entwickelt.
Proof-of-Concept soll in diesem Kontext bedeuten, dass keine vollständige Applikation erzeugt wird, die bereit ist, produktiv genutzt zu werden. Sie soll lediglich als ein Prototyp für den Anwendungszweck von Geo-Real-Time-Queries dienen. Die Features beschränken sich ebenfalls auf den Schwerpunkt, dass die Applikation in Hinblick auf Geo-Real-Time-Queries evaluiert werden kann. Die Applikation ist fester Bestandteil dieser Arbeit. Der dafür verwendete Quelltext ist zur Referenz ebenfalls vollständig auf der beiliegenden CD enthalten.
Als Problemstellung wurde folgendes Szenario definiert: Als Benutzer eines Handys, möchte dieser mittels einer App ein Taxi rufen. Er setzt seinen derzeitigen Standort und definiert einen Bereich (bspw. in Kilometern). In diesem Bereich werden alle Taxen angezeigt und ob diese frei oder belegt sind. Verlässt ein Taxi den vom Nutzer definierten Bereich, wird es nicht mehr auf der Karte angezeigt und kann somit nicht gerufen werden. Kommt ein Taxi in den definierten Bereich, wird es durch den Nutzer auswählbar und kann gerufen werden - angenommen das Taxi ist noch nicht reserviert. Taxen senden ihre derzeitige Position an die Datenbank. Dies kann unter anderem durch ein GPS umgesetzt werden.
Als Lösung für dieses Problem wird eine Webapplikation entwickelt, die Zugriff auf die Geo-Real-Time-Queries via der Baqend Plattform Schnittstelle hat. Dabei werden einige Aspekte emuliert. Zum einen wurden keine öffentlichen APIs zu GPS-Daten von Taxen gefunden. Dieser Umstand wird umgangen, indem einige Routen als Arrays von $(Latitude, Longitude)$ Angaben gespeichert werden. In einer Oberfläche kann für jedes Taxi eine Route gewählt werden. Diese wird anschließend vom Taxi abgefahren. Ebenfalls emuliert wird das Setzen der Position des Nutzers. In einer realen Applikation wäre es intuitiver, den derzeitigen Standort des Nutzers abzufragen und als Position in der

App zu setzen. Zu Testzwecken wird die Position allerdings per Klick auf die Karte definiert.

5.2 Umsetzung der Proof-of-Concept Applikation

Im vorangegangenen Abschnitt wurde bereits grob erläutert, dass die Applikation zwei Oberflächen besitzt. Eine Oberfläche zum Steuern der Routen für die Taxen und dem Aktivieren, dass sie diese Routen fahren. Zum anderen eine Oberfläche für den Klienten, auf dem er seine Position setzt und den Bereich in dem er nach Taxen suchen möchte. Sowohl der *$geoWithin* als auch *$nearSphere* Query Selektor sind für dieses Szenario geeignet. *$nearSphere* ist dabei leichter umzusetzen, da es bereits eine maximale Distanz in Metern entgegen nimmt. Allerdings ist die zusätzliche Sortieroperation nach der Distanz zum Ziel irrelevant. Um trotzdem ihre Funktionsfähigkeit nachzuweisen, sollen die Taxen in verschiedenen Farben gekennzeichnet werden. Um einen *$geoWithin* Query Selektor einsetzen zu können, muss die angegebene Distanz in eine Angabe von $(Latidude, Longitude)$ in die verschiedenen Richtungen umgerechnet werden. Hier soll zu Demonstrationszwecken, anders als beim *$nearSphere* Query-Selektor, kein Kreis um das Ziel gelegt werden, sondern ein Polygon in Form eines Quadrates. Es resultieren somit zwei Oberflächen für den Nutzer, die jeweils eine der beiden Queries abonnieren.

5.2.1 Die Fahrer-Oberfläche

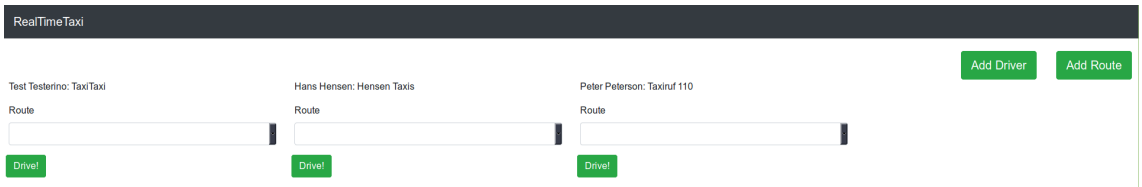


Abbildung 5.1: PoC-App: Fahrer-Oberfläche

In der Fahrer-Oberfläche kann für jeden bestehenden Fahrer eine der existierenden Routen gewählt werden (siehe Abbildung 5.1). Durch einen Klick auf den *Drive!* Button beginnt das Taxi die gewählte Route entlang zu fahren. Dabei wird eine asynchrone Funktion ausgeführt, die sich bei erfolgreicher Ausführung rekursiv selbst aufruft. In dieser Funktion wird die Position des Taxis an den nächsten Index des Routen-Arrays gesetzt. Ist das Taxi die Route vollständig abgefahren, fängt es von vorne an. Das Fahren ist theoretisch unendlich lange möglich, was eine Warnung des Browsers nach sich zieht, dass das Skript die Ausführung der Seite blockiert. Daher muss die Funktion asynchron aufgerufen werden, da die Oberfläche ansonsten nicht mehr reagiert, nachdem das erste Taxi auf den Weg geschickt wurde.

```
1 async function startDrive(driverId, routeId) {
2     DB.Taxi
```

```
3         .find()
4         .equal('id', driverId)
5         .singleResult((taxi) => {
6             DB.Route.find().equal('id', routeId).singleResult((route) => {
7                 updatePos(taxi, route.route, 0);
8             });
9         });
10 }
11
12 function updatePos(taxi, route, index) {
13     if (!globalDriving[taxi.id]) {
14         return;
15     }
16     if (route.length <= index) {
17         index = 0;
18     }
19     taxi.pos = route[index];
20     setTimeout(function(){
21         taxi.update().then((success) => {
22             return updatePos(taxi, route, index+1);
23         });
24     }, 50);
25 }
```

Listing 5.1: Funktion zum Abfahren der Route

In Listing 5.1 ist der Quelltext für diese Logik zu sehen. Die **startDrive** Methode wird asynchron aufgerufen, wenn der *Drive!* Button gedrückt wird. Dort wird nach dem Fahrer und der Route mit den Identifiern aus den Parametern gesucht. Existieren diese, wird mit dem Taxi und der Route die **updatePos** Methode aufgerufen. Diese aktualisiert die Position des Taxis mit dem in den Parametern übergebenen Index im Route-Array. Ist die Aktualisierung erfolgreich, ruft diese Funktion sich selbst mit dem darauffolgenden Index auf. Die **setTimeout** Methode bestimmt, wie lange gewartet wird (in Millisekunden), bis der Quelltext ausgeführt wird. Die Zeit kann angepasst werden, um die Taxen schneller bzw. langsamer fahren zu lassen. Im Quelltext befindet sich eine globale **Map (o.a. Dictionary)**. Diese führt über alle Taxen buch, ob sie gerade fahren. Durch das Betätigen des *Drive!* Buttons wird der Wert für das Taxi invertiert. In Zeile 13 ist zu sehen, wie dies als Abbruchbedingung für das Fahren der Route genutzt wird.
Neben den Formularen für die bestehenden Taxen gibt es zwei Buttons am oberen rechten Rand. Über **Add Driver** kann ein Fahrer mithilfe eines Formulars hinzugefügt werden.
Unter **Add Route** kann eine neue Route hinzugefügt werden (Siehe Abbildung 5.2). Hier wird eine Karte mithilfe von Google Maps geladen. Anschließend können Marker gesetzt werden, um eine Route zu definieren. Hat der Nutzer alle Marker für seine Route gesetzt, kann er diese speichern. Durch einen Klick auf **Reset Route**, kann der Nutzer eine neue Route erzeugen oder bei einem Fehler seinen Fortschritt zurücksetzen.

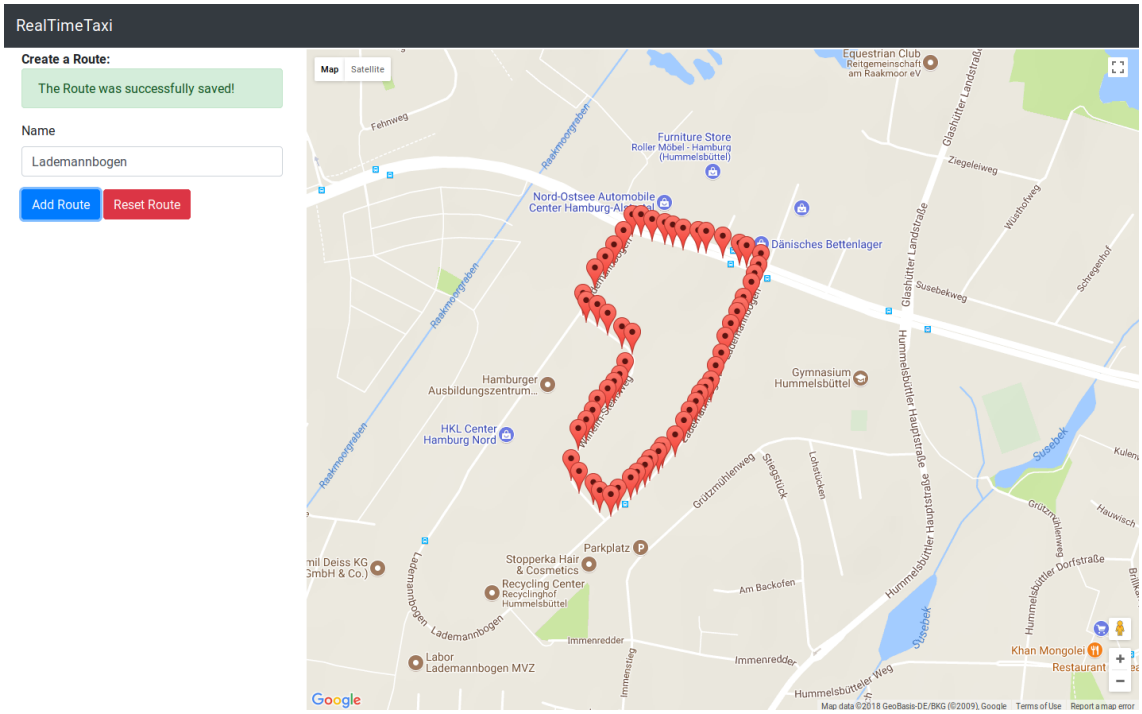


Abbildung 5.2: PoC-App: Route hinzufügen

5.2.2 Die Klienten-Oberfläche

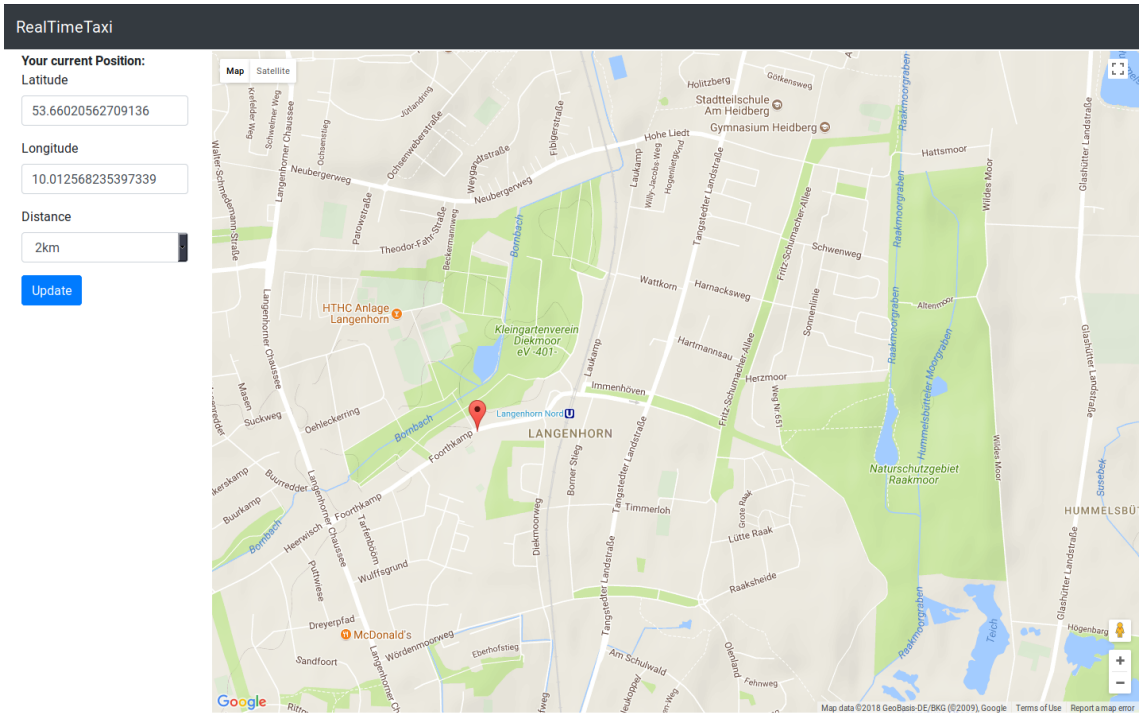


Abbildung 5.3: PoC-App: Position des Klienten setzen

Wie zuvor erwähnt, werden zwei Oberflächen bereitgestellt, um sowohl den *$nearSphere* als auch den *$geoWithin* Query-Selektor in einem ähnlichen Szenario zu testen. Beide

Oberflächen sehen sehr ähnlich aus und sind ebenfalls identisch bedienbar. Wie in Abbildung 5.3 zu sehen ist, kann auf eine beliebige Stelle der Karte geklickt werden, um die eigene Position zu definieren. In der linken Spalte stehen die Latidude und Longitude Angabe zur Information. Es kann ein Radius angegeben werden, in dem Taxen angezeigt werden. Durch einen Klick auf den **Update** Button wird eine *Query-Subscription* erzeugt. Dieser ist in beiden Oberflächen visuell sichtbar. In der *$nearSphere* Oberfläche wird ein roter Kreis gezeigt. Ab diesem Zeitpunkt werden auch alle Taxen im definierten Query gezeigt. Hier unterscheiden sich die *$nearSphere* und *$geoWithin* Oberflächen. In der *$nearSphere* Oberfläche werden Ergebnisse sortiert aufsteigend nach der Distanz zum Ziel angezeigt. Dies soll auch in der Applikation ersichtlich sein. Aus diesem Grund sind die Taxen farbig markiert, je nachdem welches am nächsten zum Ziel ist.

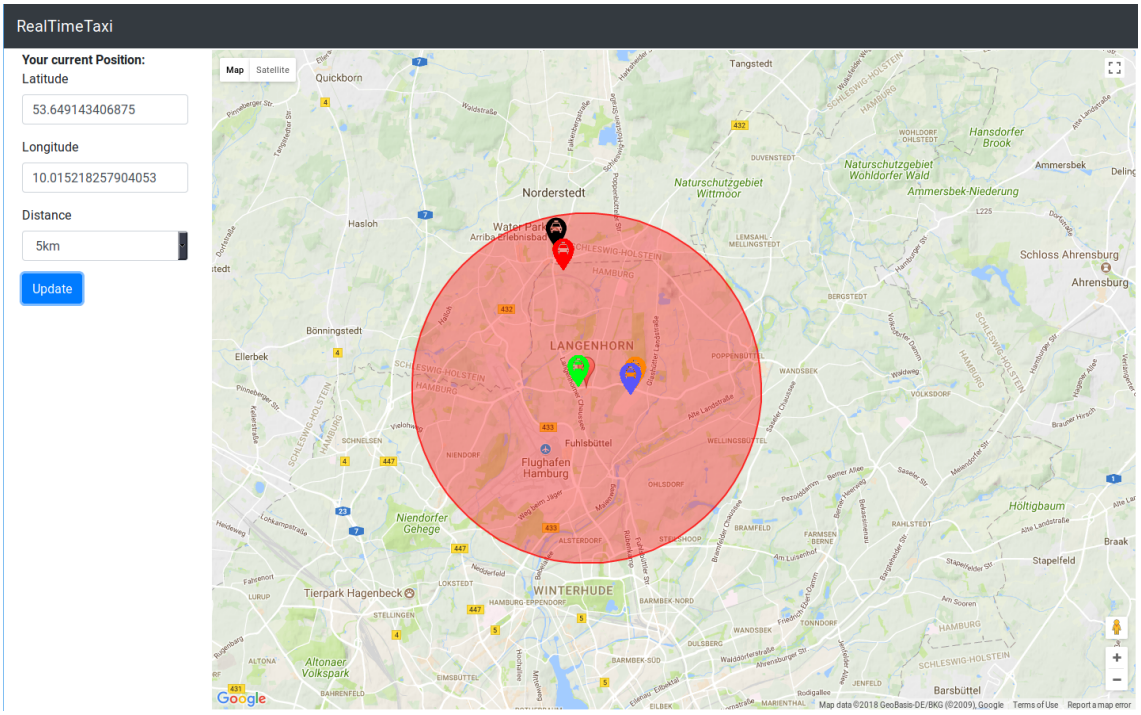


Abbildung 5.4: PoC-App: Sortierung des *$nearSphere* Query

Die farbliche Rangfolge ist dabei wie folgt definiert: grün, blau, orange, rot. Alle weiteren Taxen werden in schwarz abgebildet (Siehe Abbildung 5.4).

```
1  var query = DB.Taxi
2      .find()
3      .near('pos', clientPosition, range);
4
5  var onNext = event => {
6      if (event.matchType === 'add') {
7          createTaxiMarker(event.data, counter);
8      } if (event.matchType === 'change') {
9          updateTaxiPos(event.data.id, event.data.pos);
10     } if (event.matchType === 'changeIndex') {
11         updateTaxiColor(event.data.id, event.index);
```

```
12      } if (event.matchType === 'remove') {
13          taxis[event.data.id].setMap(null);
14      }
15  }
16
17  var stream = query.eventStream();
18  subscription = stream.subscribe(onNext);
```

Listing 5.2: Baqend Real-Time-Query Subscription *$nearSphere*

In Listing 5.2 ist die *Query Subscription* für den *$nearSphere* Query Selektor, in Form eines **eventStream-Queries**, zu sehen. Zuerst wird in Zeile 1-3 ein *near* Query erzeugt (Dies ist eine Funktion der Baqend Schnittstelle für den MongoDB *$nearSphere* Query Selektor). Dieser wird an der Taxi Kollektion aufgerufen und soll das *pos* Feld der Taxen gegen die Position des Klienten vergleichen. Die maximale Distanz ist durch den *range* Parameter definiert. Anschließend wird eine Funktion definiert, die ein Ereignis entgegen nimmt und je nach ihrem *match type* Änderungen durchführt. In diesem Fall sind die relevanten Ereignisse:

- *add*: Ein neues Taxi muss auf die Karte gesetzt werden
- *change*: Die Position eines bestehenden Taxis muss aktualisiert werden
- *changeIndex*: Die Farbe eines bestehenden Taxis muss geändert werden
- *remove*: Das Taxi muss von der Karte entfernt werden.

Diese Funktion wird bei der *Subscription* des *eventStreams* gesetzt. Sie wird jedes mal aufgerufen, wenn es eine Änderung innerhalb des *near* Queries gibt. Anhand des Event-Typs, kann ermittelt werden, was im Bereich des Queries passiert ist. In den Fällen kann anschließend entsprechend in der Applikation reagiert werden. Neben dem sogenannten *onNext Callback*, welches bei jedem Event aufgerufen wird, gibt es noch zwei weitere, optionale Callbacks, die an die *subscribe* Methode gegeben werden können. Als zweiter Parameter kann ein *onError* Callback übergeben werden, um auftretende Fehler zu behandeln [Baq17b]. Als dritter Parameter kann ein *onComplete* Callback übergeben werden, welcher aufgerufen wird, wenn Verbindungsprobleme zur Baqend Plattform auftreten [Baq17b]. Die Klienten-Oberfläche für den *withinPolygon* Query (Baqend API Funktion für MongoDBs *$geoWithin* Query) ist in Abbildung 5.5 zu sehen. Anstatt einer Distanz, wird bei diesem Query Selektor geprüft, ob Objekte im gegebenen Polygon liegen. Als Polygon wird eine quadratische Fläche definiert. Die Grenzen des Polygons entsprechen näherungsweise der angegeben Distanz in alle Richtungen.

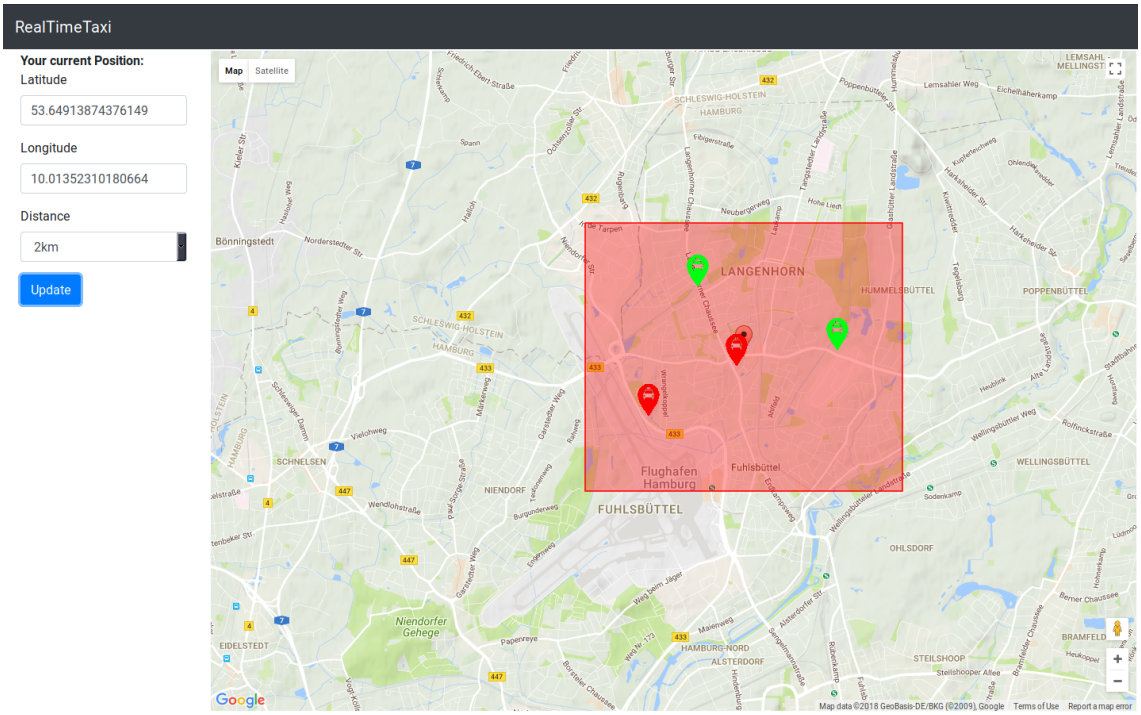


Abbildung 5.5: PoC-App: withinPolygon Klienten-Oberfläche

Wie in Listing 5.3 zu sehen, kann in diesem Fall keine Distanz direkt an den Query übergeben werden. Zu Testzwecken wird daher ein Polygon definiert, welches in alle Richtungen die vom Klienten gewählte Distanz besitzt. Die Distanz ist lediglich näherungsweise definiert und entspricht nicht der exakten Berechnung. Das generierte Polygon wird anschließend an den *withinPolygon* Query gegeben. Ansonsten ist die Query-Subscription analog zu der des *near* Queries.

```
1  function createPolyFromPos(pos, range) {
2      var dist = range * 0.00000860000;
3      return [
4          new DB.db.GeoPoint(pos.latitude - dist, pos.longitude - dist * 2),
5          new DB.db.GeoPoint(pos.latitude + dist, pos.longitude - dist * 2),
6          new DB.db.GeoPoint(pos.latitude + dist, pos.longitude + dist * 2),
7          new DB.db.GeoPoint(pos.latitude - dist, pos.longitude + dist * 2),
8          new DB.db.GeoPoint(pos.latitude - dist, pos.longitude - dist * 2),
9      ];
10 }
11
12 var queryPoly = createPolyFromPos(clientPosition, range);
13 var query = DB.Taxi
14     .find()
15     .withinPolygon('pos', queryPoly);
```

Listing 5.3: Baqend Real-Time-Query Subscription: withinPolygon

5.3 Evaluation der Geo-Real-Time-Queries

In diesem Kaptitel wird die zuvor vorgestellte PoC-App evaluiert. Zuerst soll anhand der PoC-App in ihrem ist-Zustand die Funktionalität und Korrektheit geprüft werden. Anschließend wird anhand einiger Umstrukturierungen versucht etwas über die Performance und Skalierbarkeit der Geo-Real-Time-Queries zu sagen. Dabei wird eine Aussage darüber getroffen, ob mit der Geo-Komponente ein Flaschenhals eingeführt wurde.

5.3.1 Funktionalität der Geo-Real-Time-Queries

Den Kern der Evaluation der Funktionalität bildet Abbildung 2.2 aus Kapitel 2. Zur Wiederholung: dort ist ein Baum zu sehen, welcher die interessanten *Change Events* einer *Query Subscription* zeigt. Jedes Blatt dieses Baumes kann auf einfache Art und Weise durch die Funktionalität der PoC-App überprüft werden, da für jede der drei *match types* ein Callback registriert ist, die ein visuelles Resultat des Matching Status widerspiegelt. Der linke Pfad ist dadurch erfüllt, dass sich die Fahrzeuge im definierten Bereich bewegen. Sie waren vorher ein Match und es gab eine Aktualisierung ihrer Position innerhalb des definierten Bereichs. Dies spiegelt sich in einer Bewegung des Taxis wieder.
Der zweite Pfad besagt, dass es sich vorher um kein Match handelte, sich aber mit der letzten Aktualisierung der Position in den Bereich des Queries bewegt hat. Dies ist in der PoC-App an den Grenzen der Form zu erkennen. Taxen außerhalb des rot gekennzeichneten Bereichs werden nicht dargestellt. Kommen sie in den Bereich hinein erscheint auch der Marker, welcher ihre Position visualisiert.
Ähnlich ist die Funktionalität im dritten Pfad zu beobachten: Bewegt sich ein Taxi außerhalb der rot gekennzeichneten Fläche, ist dessen Marker nicht mehr auf der Karte zu sehen und wird somit nicht weiter verfolgt, bis es erneut in den definierten Bereich des Queries eintritt.
Eine weitere Funktionalität, die mit den Geo-Queries Einzug erhält, ist die Sortierung der Ergebnisse. Dies trifft auf den *$nearSphere* Query zu und ist nicht direkt in Abbildung 2.2 ersichtlich, da es sich um einen Spezialfall des ersten Pfades handelt. Zur Wiederholung: ein *$nearSphere* Query gibt die Resultate sortiert zurück. Das heißt für einen Real-Time-Query handelt es sich um ein *change event*, da sich der Index des Objektes verändert hat. Dies ist, wie im vorherigen Abschnitt vorgestellt, ersichtlich durch die verschiedene Kolorierung der Taxen. Bei einer Veränderung der Position eines Taxis, kann es auch ein *indexChange event* eines anderen Taxis hervorrufen, da dieses nach der Aktualisierung näher am Ziel ist, als das bisherige.
Im Quelltext der PoC-App ist zu sehen, dass verschiedene Datentypen für die Google Maps API und die Baqend API genutzt werden mussten. Zwar arbeiten beide mit Angaben in $(Latitude, Longitude)$, allerdings werden beispielsweise die grafische Darstellung des Kreises oder des Quadrates und der (nicht sichtbare) Query Bereich getrennt voneinander berechnet. Die visuelle Form wird von der Google Maps API berechnet, während

die nicht sichtbare Form von der in dieser Arbeit entwickelten Geo-Komponente berechnet wird. Wenn ein Taxi genau an der Stelle von der Karte entfernt wird, wenn es den rot gekennzeichneten Bereich verlässt, ist es somit auch ein Beweis dafür, dass die geometrischen Formen identisch definiert sind. Da die Google Maps API öffentlich und kommerziell genutzt wird, kann hier davon ausgegangen werden, dass die Berechnungen dieser API korrekt sind. Durch den Ein- bzw. Austritt eines Taxis in den rot gekennzeichneten Bereich ist zu sehen, dass die Abweichungen nicht vorhanden bis minimal sind. Somit ist ersichtlich, dass auch die, in dieser Arbeit entwickelte, Geo-Komponente Positionen, geometrische Formen und ihre Beziehungen auf der Erde mit hoher Genauigkeit berechnet.

5.3.2 Performance und Skalierbarkeit

Einer der wichtigsten Punkte, die für InvaliDB sprechen, ist dessen horizontale Skalierbarkeit. Genauer heißt dies, dass der Durchsatz von *Query Subscriptions* und Schreiboperationen linear mit der Anzahl der Maschinen im Cluster skaliert [Winht]. Diese Eigenschaft darf durch die Geo-Komponente nicht gebrochen werden. Um diese Behauptung zu zeigen, müsste ein ähnlicher Versuchsaufbau, wie in [Winht] stattfinden. In diesem würde geprüft werden, dass auch unter Verwendung der Geo-Queries der Durchsatz mit der Anzahl der Maschinen skaliert. Da dieser Aufbau weder aus Zeit noch aus Kostengründen in dieser Arbeit durchgeführt werden konnte, wird trotzdem ein möglicher Versuchsaufbau skizziert.
Im Anschluss daran wird mit den gegebenen Mitteln ein Test durchgeführt, dass mit der Geo-Komponente selbst kein Flaschenhals in der Berechnung des *Matching Status* eingeführt wurde. Ein Flaschenhals bezeichnet ein Stück Software, welches viel langsamer als der Rest der Software agiert. Er bildet dadurch einen metaphorischen Flaschenhals, da Daten, die durch eine Software verarbeitet und durchgereicht werden, sich an der Komponente aufstauen [Gra82]. Im Ergebnis wird versucht einen Bezug zum Experiment aus [Winht] herzustellen und so eine Aussage treffen zu können, ob die Geo-Komponente einen Flaschenhals bildet.

Versuchsskizze zum Nachweis der linearen Skalierbarkeit

In diesem Abschnitt soll ein Versuchsaufbau skizziert werden, um die lineare Skalierbarkeit von InvaliDB mit Geo-Queries nachzuprüfen. Der Aufbau ist dabei sehr nah am ursprünglichen Test aus [Winht] angelehnt.
Es wird eine Umgebung benötigt, welche aus mehreren Maschinen mit identischer Hardware besteht, welche über LAN miteinander verbunden sind. Auf einer der Maschinen sind virtuelle Server für zwei InvaliDB Klienten installiert. Eine wäre für das Einsetzen und Aktualisieren der Daten zuständig. Der zweite meldet die Queries an. Ein Redis Server zur Kommunikation zwischen dem InvaliDB Klienten und einem Storm-Cluster, in dem InvaliDB läuft wird ebenfalls benötigt.

Ein Versuch besteht aus zwei Phasen: Der Präparierungs- und Ausführungsphase. In der Präperierungsphase wird der erste InvaliDB Klient eine festgelegte Zahl von *Query-Subscriptions* ausführen. Diese müssen sich in ihrer Position (oder anderen Merkmalen) unterscheiden. Ansonsten wird nicht explizit ein neuer Query angemeldet, sondern die Anmeldung findet auf einen bereits bestehenden Query statt [Winht]. Nach der Anmeldung der Queries beginnt die Ausführungsphase. Der zweite InvaliDB Klient spielt in einem geregelten Maß Objekte mit einer Positionseigenschaft ein. Dabei muss darauf geachtet werden, dass die Objekte zu bestimmten Queries eine Übereinstimmung bilden. Die Zahl der Übereinstimmungen sollte jedoch niedrig gewählt werden. Bei einer hohen Zahl an Übereinstimmungen, kann das Ergebnis des Versuchs verfälscht werden, da viele Ressourcen für die Serialisierung und Deserialisierung der übereinstimmenden Objekte benötigt werden und dieser in sich bereits einen Flaschenhals bildet [Winht]. Bei jeder Einfüge-Operation wird die Latenz gemessen von dem Moment in der ein Update geschrieben wird, bis es eine Benachrichtigung von InvaliDB für das erfolgreiche Einfügen gibt. In dieser Umgebung muss nun geprüft werden, wie das Cluster mit der Anzahl der eingefügten Objekte und mit der Anzahl der *Query Subscriptions* skaliert. Zuerst wird die Leistung einer Query- und Objektpartition geprüft. Nun wird eine feste Zahl an weiteren Maschinen für Objekt- und Query Partitionen hinzugefügt. Anschließend wird die Zahl der *Query-Subscriptions* und Schreiboperationen linear mit der Anzahl der Maschinen erhöht. Nun muss geprüft werden, ob die Latenz im Cluster unter der erhöhten Last gleichbleibend ist.
Zudem wurde im vorhergehenden Abschnitt gezeigt, wie der *$nearSphere* und der *$geoWithin* Query Selektor in dem hier dargestellten Szenario identisch genutzt werden können. Somit kann durch den hier dargestellten Versuchsaufbau auch der Overhead der Sortierung im *$nearSphere* Selektor gegenüber einem unsortierten Query Selektor getestet werden.
Leider ist es weder zeitlich, noch von den Ressourcen möglich einen solchen Versuchsaufbau im Zuge dieser Arbeit zu initiieren und durchzuführen. Somit kann die Bedingung, dass Maschinen mit der Höhe des Durchsatz skalieren, nicht überprüft werden. Allerdings soll getestet werden, ob mit der Geo-Komponente ein Flaschenhals eingebaut wurde. Der Versuchsaufbau für diesen Test ist im folgenden Abschnitt geschildert.

Versuchsaufbau

In diesem Versuch wird anhand des privaten Computers des Autors ein Test ausgeführt, der zeigen soll, dass mit der Geo-Komponente kein Flaschenhals eingeführt wurde. Da bereits in [Winht] gezeigt wurde, dass InvaliDB linear skaliert, soll hier lediglich geprüft werden, ob der Durchsatz bei Geo-Queries ähnlich hoch ist, wie beim Test in [Winht] mit einer Objekt- und Query-Partition.
Als Hardware für den Versuch kam ein Thinkpad T450s mit einem Intel i7-5600U und 12 GB RAM zum Einsatz. Auf der Maschine laufen ein InvaliDB Server mit Storm, ein Redis

Server und eine MongoDB Instanz. Zudem wird der Chrome Browser in Version 64 zum Ausführen der Javascript Datei genutzt (Ebenfalls zur Referenz in der beiliegenden CD enthalten).
Zum Testen kam ein Skript zum Einsatz, über welches verschiedene Operationen gestartet werden können. Die erste Aktion meldet eine Zahl an *Query Subscriptions* an und gibt jede Sekunde die Anzahl der aufgetretenen *Change Events* aus. Die zweite Aktion fügt jede Sekunde 100 Objekte ein. Das Objekt besteht dabei aus einem GeoPoint Feld.
Der Versuchsaufbau wurde versucht so nah wie möglich am Experiment von [Winht] zu halten, um das abschließende Ergebnis übertragbar zu machen. So gibt es auch hier eine Präperierungs- und Ausführungsphase. In der Präparierungsphase wird die Kollektion, in der die Objekte eingefügt wurden, geleert. Anschließend werden alle bestehenden *Query Subscriptions* gekündigt und eine fixe Zahl an neuen *$geoWithin* Queries angemeldet. Das Skript erlaubt ebenfalls die Anmeldung von *$nearSphere* Queries. Da der Versuchsaufbau möglichst identisch zu [Winht] sein soll und dort nicht sortierende Queries genutzt wurden, soll dies auch in diesem Versuch der Fall sein. Jeder Query definiert ein quadratisches Polygon über eine kleine Fläche. Die Polgone werden beginnend am Nordpol entlang der Latitude angemeldet. Die Longitude ist konstant gleichbleibend. In der Ausführungsphase werden 100 neue Objekte mit einem *GeoPoint* Element pro Sekunde eingefügt. Die Dauer der Ausführungsphase ist dabei auf 60 Sekunden festgelegt. Es ist eine feste Zahl von Übereinstimmungen pro Versuch festgelegt. Diese Zahl wurde bewusst niedrig gewählt, damit nicht zuviele Ressourcen für die Serialisierung und Deserialisierung von Objekten aufgewendet wurde. Ziel war es die reine Performanz der Berechnung der *Matching*s zu messen. In [Winht] wurde sich für 1000 Übereinstimmungen entschieden. Ein Experiment dauerte ebenfalls 60 Sekunden. Somit handelte es sich um:
$\frac{1000\ \ddot{U}bereinstimmungen}{60\ Sekunden} = 16,\overline{66} \approx 17\ \ddot{U}bereinstimmungen\ pro\ Sekunde.$
Es gab im Test zur Lese-Skalierbarkeit 1000 op/s. Somit wurden 60000 Objekte während des Experiments eingefügt. Bei 60000 Gesamteinträgen waren $\frac{1}{60}$ Übereinstimmungen. Analog hierzu wurde sich in diesem Experiment für:
$\frac{100\ \ddot{U}bereinstimmungen}{60\ Sekunden} = 1,\overline{66} \approx 2\ \ddot{U}bereinstimmungen\ pro\ Sekunde$
entschieden. Die Zahl der Übereinstimmungen pro Sekunde wird überwacht. Schwankt die Zahl von Sekunde zu Sekunde, sagt dies aus, dass InvaliDB nicht in der Lage ist, die anfallenden Objekte auf ihren Übereinstimmungsstatus zu überprüfen (Siehe Abbildung 5.6b). Dies gilt als Abbruchbedingung für den Versuch. Es wird anschließend ein neuer Versuch mit einer niedrigeren Zahl an *Query Subscriptions* gestartet. Verlief der Versuch erfolgreich, wird die Zahl der *Query Subscriptions* erhöht. So wird sich in 200er Schritten der Zahl der möglichen *Query Subscriptions* genähert.

(a) Stabile Verarbeitung bei 6800 Subscriptions

(b) Instabile Verarbeitung bei 7000 Subscriptions

Abbildung 5.6: Ergebnis der möglichen, aktiven *Query Subscriptions*

Ergebnis

Ohne Ausführung des Skripts lag die CPU-Last bei 10%. Die Testmaschine war in der Lage 200 Schreiboperationen pro Sekunde auszuführen, bevor alle vier CPUs zu 100% ausgelastet waren. Um Ressourcen für die *Query Subscriptions* übrig zu haben, wurde entschieden 100 Schreiboperationen pro Sekunde durchzuführen. Mit den Schreiboperationen liegt die Auslastung der Test-Maschine bei rund 50%. Somit bleiben 50% CPU Ressourcen für die *Query Subscriptions*.
Bis 6800 *Subscriptions* sind konstante *Change Events* gemessen worden (Siehe Abbildung 5.6a). Bei einer höheren Anzahl, kam es zu vielen Unregelmäßigkeiten. Dies entspricht:
$100\ op/s\ *\ 6800\ Query\ Subscriptions = 680.000$ *Matching* Operationen pro Sekunde.
Zum Vergleich sind im Lese-Skalierbarkeit Test von [Winht]:
$1000\ op/s\ *\ 1500\ Query\ Subscriptions = 1.500.000$ *Matching* Operationen pro Sekunde erreicht worden. Dies entspricht $\frac{1}{3}$ der *Matching* Operationen. In [Winht] wurden Intel Xeon E5-2620 v2 mit sechs Kernen und einer signifikant höheren Leistung als ein i7-5600U [CP18] verwendet. Die Maschine wurde ausschließlich für die InvaliDB Instanz und somit für die *Query Subscriptions* genutzt. Schreiboperationen wurden von einer weiteren Maschine übernommen. Die Maschine wurde bei dem Versuch zu rund 80% ausgelastet.
Durch die Ungenauigkeiten und Störungen, die durch die zusätzlichen Aufgaben neben der *Matching* Berechnungen in diesem Experiment hervorgerufen wurden, sind die Ergebnisse nicht zweifelsfrei auf [Winht] übertragbar. Die Störungen würden jedoch zum Negativen dieses Ergebnisses beitragen. Somit kann bei den Leistungsunterschieden der Maschinen und der erhöhten Arbeitslast der Testmaschine ein Ergebnis von $\frac{1}{3}$ der erreichten *Matching* Operationen als sehr positiv gesehen werden. Es kann somit gesagt werden, dass mit der Implementation der Geo-Queries kein Flaschenhals eingeführt wurde.
Ebenfalls überprüft wurde der *$nearSphere* Query. Dieser hat signifikant bessere Ergebnisse (> 9600 Subscriptions) erzielt als der *$geoWithin* Query. Dies rührt aber daher, dass

bei einem Objekt pro *Query Subscription* keine Sortierung stattfand und im *$nearSphere* weniger Fälle beachtet werden müssen. Die Berechnung ist somit weniger komplex. Für das Ergebnis heißt es aber, dass auch der *$nearSphere* Query Selektor selbst keinen Flaschenhals bildet, da auch er mehr als $\frac{1}{3}$ der Matching Berechnungen aus [Winht] erreicht hat.

6 Verwandte Arbeiten

Es gibt bereits einige Datenbanken die Real-Time-Queries anbieten. Einige dieser Projekte unterstützen inzwischen auch Geo-Queries. In diesem Kapitel werden einige dieser Projekte näher vorgestellt. Der Fokus liegt dabei auf den geodätischen Anfragen. Da die Anfragesprache (o.a. *Query DSL*) dieser Arbeit und MongoDB identisch ist, wird der Bezug zu dieser Arbeit auch immer den Bezug zu MongoDB implizieren.

6.1 Firebase

Firebase ist eine Plattform die von Google entwickelt wird. Sie soll dem Nutzer helfen, mit wenig Aufwand, „gute“ mobile Applikationen zu entwickeln [Goo17a]. Eine Komponente der Firebase Plattform ist dessen Real-Time Datenbank. Dies ist eine in der Cloud gehostete NoSQL Datenbank, die es erlaubt Daten zu speichern und diese in Echtzeit zwischen den Nutzern zu synchronisieren [Goo17b]. Anders als bei vielen anderen Document-Stores, arbeitet Firebase nicht auf vielen JSON-Dokumenten, sondern auf einem einzigen Dokument. Um an Daten zu kommen, muss durch die Hierarchie des JSON Dokuments navigiert werden, bis der gesuchte Kind-Knoten gefunden wurde [Win17a].

6.1.1 Geo-Queries

Firebase hat die Möglichkeit geodätischen Anfragen auszuwerten. Dies ist durch eine Open-Source Bibliothek namens GeoFire möglich. GeoFire gibt es derzeit für drei verschiedene Sprachen [Goo18a]:

1. Javascript
2. Objectiv-C/Swift
3. Java

GeoFire ist sehr minimal. Es erlaubt Positionsdaten in Form von $(Latitude, Longitude)$ zu Datensätzen zu speichern [Wen18].

```
1 var geoQuery = geoFire.query({
2   center: [37.4, -122.6],
3   radius: 1.609 //kilometers
4 });
```

Listing 6.1: Beispiel eines GeoFire Queries [Wen18]

Zudem können Real-Time-Geo-Queries auf eine bestimmte Position mit einem gewissen Radius gesetzt werden [Wen18] (Siehe Listing 6.1).

```
1 geoQuery.on("key_entered", function(key, location, distance) {
2   console.log("Bicycle shop " + key + " found at " + location + " (" + distance
        + " km away)");
3 });
```

Listing 6.2: Beispiel eines GeoFire Callbacks [Wen18]

GeoFire bietet zwei Events für Geo-Queries: *key_entered* und *key_exited*. Diese reagieren, wenn ein Objekt den angegebenen Radius des Queries betritt oder verlässt [Wen18]. Ein Beispiel ist in Listing 6.2 zu sehen. Hier wird mittels der *on* Methode ein Callback für das gewünschte Event angemeldet.
Die Positionsdaten werden zum Key eines Datensatzes gespeichert. Hierdurch ist es GeoFire möglich, nicht alle Daten in den Speicher zu laden, sondern nur die, die sich in der Nähe zum Query befinden [Goo18a]. Hierdurch soll die Applikation auch bei großen Datenmengen leichtgewichtig und responsiv bleiben. Neben diesen Features erlaubt GeoFire noch folgendes [Wen18]:

1. Positionsdaten löschen
2. Query-Kriterien ändern
3. Query-Subscriptions abbrechen
4. Behandeln von Daten, die sich innerhalb des Queries bewegen
5. Alarmieren, wenn die initialen Daten verarbeitet wurden

GeoFire zeigt sehr anschaulich, wie es mit sehr einfachen Mitteln möglich ist, Applikationen die mit geodätischen Daten arbeiten, zu implementieren. Mit den Events für den Ein- und Austritt von Objekten aus dem Query wird auch der Vorteil von Real-Time-Queries hervorgehoben. GeoFire ist jedoch wie bereits erwähnt sehr minimal. Es erlaubt lediglich das Speichern von Punkten zu Datensätzen. Der Geo-Query ähnelt dem *$geoWithin* Query Selektor mit einem *$centerSphere* Shape Operator aus dieser Arbeit. Neben diesem speziellen Query und dem Speichern von Punkten bestehen allerdings keinerlei Möglichkeiten andere Formen zu speichern oder Queries mit diesen zu konstruieren. Firebase bleibt somit weit hinter den Möglichkeiten, die diese Arbeit für den Nutzer bietet.

6.2 RethinkDB

RethinkDB ist eine skalierbare Open-Source Datenbank, welche es erlaubt Real-Time Applikationen zu schreiben [Ret18a]. Intern arbeitet RethinkDB ebenfalls mit einem JSON Dokumentspeicher. RethinkDB ist vom Grundkonzept ausgehend sehr ähnlich zu MongoDB [Win17a]. Real-Time-Features sind durch sogenannte *Changefeeds* möglich. Hierbei wird jeder Applikationsserver mit einem speziellen RethinkDB Prozess (RethinkDB

Proxy) gekoppelt. Am 25. September 2014 hat Josh Kuhn den Einzug geodätischen Anfragen in RethinkDB 1.15 angekündigt [Kuh18]. Der hauptverantwortliche Entwickler für die geobasierten Anfragen war Daniel Mewes [Kuh18].
Zu den geobasierten Features in RethinkDB zählen Kommandos zum Im- und Exportieren des GeoJSON Formats, Anfragen der gespeicherten Objekte, Indices und zusätzliche Funktionen, die auf den Geometry Objekten arbeiten [Kuh18].

6.2.1 Im- und Export von GeoJSON

GeoJSON Dokumente werden in oder aus *ReQL Geometry Objects* generiert. **ReQL Geometry Objects** sind dabei die intern genutzten Objekte für die Anfrage-Sprache von RethinkDB.

```
1 var geoJson = {
2     'type': 'Point',
3     'coordinates': [ -122.423246, 37.779388 ]
4 };
5 r.table('geo').insert({
6     id: 'sfo',
7     name: 'San Francisco',
8     location: r.geojson(geoJson)
9 }).run(conn, callback);
```

Listing 6.3: Import von GeoJSON in RethinkDB [Ret18b]

Anders als in der Implementierung dieser Arbeit, liegt es in der Verantwortung des Entwicklers vorher ein GeoJSON in ein *ReQL Geometry Object* umzuwandeln, bevor es bspw. in die Datenbank eingefügt werden kann. Einträge werden somit, wie in Listing 6.3 gezeigt, eingefügt. Hervorzuheben ist dabei Zeile 8, in der mithilfe der Funktion *r.geojson* das GeoJSON Objekt in ein *ReQL Geometry Object* überführt wird. In dieser Arbeit wird dies intern von der Anfrageauswertung gemacht, wenn sich das GeoJSON unter einem *$geometry* Shape Operator befindet. Da *ReQL Geometry Objects* nicht so mächtig sind wie GeoJSON Dokumente, ist auch nur eine Untermenge aller GeoJSON Formen zur Nutzung in RethinkDB erlaubt. Hierunter fallen der *Point*, *LineString* und das *Polygon*. Somit sind weder Multi-Objekte erlaubt, noch die *GeometryCollection*. Allerdings können die Objekte in ein Array verpackt werden. Anschließend müssen sie mit einem geospatialen Multi-Index ausgestattet werden. Dies erreicht den gleichen Effekt, wie ein Multi-Objekt [Ret18b]. Neben den bekannten GeoJSON Formaten unterstützt RethinkDB auch Kreise als zusätzliche Form.
Koordinaten werden wie bei dieser Arbeit auch in der Form $(Longitude, Latitude)$ entgegen genommen [Ret18b]. Andere Formate werden von RethinkDB abgelehnt.

6.2.2 Geodätische Anfragen

RethinkDB unterstützt zwei Arten von Anfragen: *get_intersecting* gibt alle Objekte zurück, die das gesuchte Objekt schneiden [Ret18c] (analog zu *$geoIntersects*). *get_nearest*

findet das nächste Objekt zum gesuchten Punkt [Kuh18]. Es gibt ebenfalls eine Liste mit aufsteigender Entfernung zurück [Ret18d]. *get_nearest* bietet dabei einige weitere Parameter an. Es lässt sich die maximale Länge der zurückgegebenen Liste spezifizieren. Dies ist in dieser Arbeit durch Setzen des *limit* Operators möglich. Es kann eine maximale Distanz angegebenen werden, ab der Ergebnisse gefiltert werden [Ret18d]. Eine minimale Distanzangabe fehlt allerdings. Anders als in dieser Arbeit, lässt sich die Einheit bestimmen, in der die Distanz gemessen wird. Diese ist in dieser Arbeit fest als Meter spezifiziert. RethinkDB erlaubt: Meter, Kilometer, Meilen, Nautische Meilen und Fuß [Ret18d]. Zudem erlaubt der Query zwei Geo-Systeme: WGS84, welches ebenfalls der Standard in dieser Arbeit ist und eine Einheitssphäre, welche sich näher am 1 Meter Einheitskreis verhält [Ret18d]. Eine Anfrage zum Enthaltensein existiert bis zum Schreiben dieser Arbeit nicht.

6.2.3 Funktionen

Neben Anfragen bietet RethinkDB auch Funktionen an, die auf einzelne oder mehrere Objekte angewendet werden können. Bei mehreren Objekten verhält sich die Funktion wie ein Filter [Ret18f]. Auch hier gibt es eine Funktion für die Distanzberechnung zwischen zwei Objekten (*r.distance*) und der Schnitt-Beziehung (*r.intersects*) [Kuh18]. Der Schnitt kann an einer Menge von Objekten ausgeführt werden und filtert alle Objekte, die das gesuchte Objekt schneiden [Ret18f].

```
1 var circle1 = r.circle([-117.220406,32.719464], 10, {unit: 'mi'});
2 r.table('objects')('position').includes(circle1).run(conn, callback);
```

Listing 6.4: RethinkDB Funktionen:Include als Filter [Ret18e]

In den Funktionen findet sich auch eine *r.includes* Funktion. Diese agiert auch als Filter auf einer Menge [Ret18e] und ist somit das, was dem *$geoWithin* Query in dieser Arbeit am nächsten kommt. Ein Beispiel ist in Listing 6.4 zu sehen. Es wird zuerst ein *Geometry Object* der Form Kreis erzeugt. Anschließend werden alle Einträge der Tabelle *objects* mit ihrem Feld *position* überprüft, ob sie im vorher definierten Kreis enthalten sind. Die Rückgabe dieses Beispiels entspricht somit der einer *$geoWithin* Anfrage in dieser Arbeit. Die Performance wird allerdings nicht vergleichbar gut sein, wie bei einem Query-Selektor, da zuerst eine Menge an Objekten in den Speicher geladen werden muss, bevor diese gefiltert werden kann.
Schließlich bietet RethinkDB zwei weitere Funktionen an, die nicht in dieser Arbeit existieren. *r.fill* konvertiert einen *LineString* in ein *Polygon* und *r.polygon.sub* subtrahiert zwei Polygone voneinander [Kuh18]. Es wird also die Differenz von Polygon A zu Polygon B berechnet.

6.3 Appbase.io

Appbase.io ist eine Firma, die von Siddharth Kothari gegründet wurde und sich als Ziel gesetzt hat, bessere Frontend-Entwickler Werkzeuge bereitzustellen [Hal18]. Appbase.io heißt auch das Hauptprodukt der Firma und ist ein gehosteter Elasticsearch Service [Hal18]. Im Hintergrund von Appbase.io läuft Elasticsearch im *Streaming Mode*, welches das *Streamen* von Updates und Query Ergebnissen ermöglicht. Ein *Stream* heißt in diesem Kontext das gleiche, wie eine *Query Subscription* im Kontext von Baqend. Es wird nach dem ersten Request eine dauerhafte Verbindung (via Socket) offen gehalten [app18a]. Über diesen Socket hat der Server die Möglichkeit, bei Änderungen in den Daten, Events an die Klienten zu schicken. Appbase.io unterstützt dabei jede Art von Query, die auch Elasticsearch in ihrer Query DSL definiert hat [Hal18].
Als eines ihrer Haupt-Use-Cases bewerben sie die dadurch möglich werdenden *Real-Time Geolocation Queries* [app18b]. Da die Anwendungszwecke nur grob zeigen, welche Art von Applikationen mit Appbase.io möglich sind und auch die Dokumentation sehr minimal ist, wird sich in diesem Kapitel ausschließlich auf die *Query DSL* für Geodaten von Elasticsearch konzentriert. Diese ist vollständig über die Appbase.io Schnittstelle zugreifbar und kann somit uneingeschränkt genutzt werden.

6.3.1 Elasticsearch Geo Query DSL

Elasticsearch unterstützt zwei Typen von geodätischen Feldern. Der *geo_point*, welcher Punkte darstellt und durch (lat, lon) Paare definiert wird [ela18a]. Zum anderen *geo_shape*, welches Punkte, Linien, Kreise, Polygone, Multi-Polygone, Multi-Punkte, Multi-Linien, Geometrie Kollektionen und sogenannte *Envelope*s unterstützt [ela18a][ela18b]. *Envelope* ist dabei ein spezieller Typ, welches ein Rechteck durch einen oberen linken und einen unteren rechten Punkt definiert. Neben den in Elasticsearch eingebauten Typen, sind auch alle Typen, die in GeoJSON definiert sind, nutzbar. Am *geo_shape* können eine Vielzahl von Optionen gesetzt werden. Dabei übersteigen die Konfigurationsmöglichkeiten, die der anderen Kandidaten in diesem Kapitel, als auch dieser Arbeit. Beispielsweise kann die Indexierungsmethode über die Option *tree* so eingestellt werden, dass entweder Geohashes oder Quadtrees verwendet werden [ela18b]. Als Längeneinheiten kann zwischen sieben verschiedenen gewählt werden (Yards, Meilen, Kilometer, Meter, Zentimeter, Millimeter) [ela18b]. Mit der *orientation* Option kann vorgegeben werden, in welcher Richtung Polygone interpretiert werden (im oder gegen den Uhrzeigersinn) [ela18b]. Es sollen jedoch nicht alle Optionen an dieser Stelle aufgezählt werden. [ela18b] stellt eine vollständige Liste über alle Optionen bereit.
Neben den Formen unterstützt die Elasticsearch Geo Query DSL auch vier verschiedene Queries. Der gleichnamige Query *geo_shape* findet alle Formen, die in einer bestimmten Beziehung zu einer übergebenen Form stehen [ela18a]. Folgende Beziehungen werden dabei unterstützt

- Schnitt
- Disjunktheit
- Enthaltensein
- Enthält

Der zweite Query ist *geo_bounding_box* und findet alle Formen die innerhalb des definierten Rechtecks fallen [ela18a]. *geo_distance* findet Formen innerhalb einer gegebenen Distanz [ela18a]. Zuletzt gibt es noch *geo_polygon*, welches alle Formen innerhalb des übergebenen Polygons findet [ela18a].
Somit besitzt Elasticsearch die insgesamt ausgeprägteste *Geo-Query DSL*

6.4 Meteor

Zuletzt soll ein Javascript App Framework vorgestellt werden, welches speziell für reaktive Anwendungen und Webseiten entwickelt wurde [Win17b]. Meteor nutzt im Hintergrund MongoDB und kann über eine Schnittstelle angefragt werden. Dies funktioniert ähnlich zur Baqend Schnittstelle. Baqend hat jedoch zusätzliche Methoden, um Daten anzufragen. Möchte der Nutzer trotzdem MongoDB Queries nutzen, kann er hierfür eine gesonderte Methode namens **where** verwenden [Baq17a] (Siehe Listing 6.5).

```
1  Restaurants.find().where({
2      location: {
3          $geoWithin: {
4              $geometry: {
5                  type: "Polygon"
6                  coordinates: [[min.lng, min.lat], [max.lng, max.lat]
7              }
8          }
9      }
10 });
```

Listing 6.5: Beispiel eines MongoDB Queries in Baqend

Meteor auf der anderen Seite kapselt MongoDB Queries nicht in eine gesonderte Methode, sondern macht sie zu ihrer eigenen Schnittstelle. Eine äquivalente Meteor-Anfrage aus Listing 6.5 resultiert somit in eine Anfrage, wie es in Listing 6.6 zu sehen ist.

```
1  Restaurants.find({
2      location: {
3          $geoWithin: {
4              $geometry: {
5                  type: "Polygon"
6                  coordinates: [[min.lng, min.lat], [max.lng, max.lat]
7              }
8          }
9      }
```

```
10 });
```

Listing 6.6: Beispiel eines MongoDB Queries in Meteor

Meteor fokussiert sich auf reaktive Anwendungen. Das heißt, es erlaubt ebenfalls *Self-Maintaining Queries* und *Event Stream Queries*, indem es zwei unterschiedliche Konzepte in der Schnittstelle realisiert. Das erste Konzept heißt **Change Monitoring + Poll and Diff**. *Change Monitoring* überwacht einen Meteor App Server auf eingehende Schreiboperationen. Bei Schreiboperationen die aktive *Real Time Queries* betreffen, wird eine Aktualisierung an die *Subscriber* des Queries geschickt [Win17b]. Dies ist jedoch bei mehreren App Servern problematisch. Kommen Schreiboperationen über einen Meteor Server B in die MongoDB, werden andere Meteor Server A nicht informiert. Dadurch halten die Klienten, die *Real Time Queries* auf *A subscribed* haben, veraltete Daten. Aus diesem Grund wird das *Change Monitoring* mit *Poll and Diff* gepaart. *Poll and Diff* stellt klassische *pull*-basierte Queries in regelmäßigen Abständen und gibt nur die relevanten Veränderungen an die Klienten weiter [Win17b]. Dies ist auf zwei Arten ineffizient. Zum einen können Klienten über einen Zeitraum über veraltete Daten verfügen und zum anderen werden unnötig viele Queries gestellt, was in einer höheren Ressourcennutzung resultiert (Siehe Kapitel 1.1).
Aus diesem Grund wurde ein zweites Konzept eingeführt, dass das *Change Monitoring + Poll and Diff* ablösen soll. Bei diesem Konzept handelt es sich, um das **Oplog Tailing**. Dabei hängen sich Meteor Server in den Oplogs der MongoDB Primary Server [Win17b]. Replikation in MongoDB funktioniert dabei wie folgt: Ein verteiltes MongoDB System (ein sogenanntes *Replica Set*) hat einen primären und mehrere sekundäre MongoDB Instanzen [Inc18c]. Nur die primäre Instanz erhält Schreiboperationen. Diese Schreiboperationen werden in einem **Oplog** repliziert [Inc18b]. Die sekundären Instanzen lesen dieses *Oplog* und führen die Schreiboperationen auf ihren replizierten Kollektionen aus. Die Meteor Server machen in diesem Konzept nichts anderes, als sich ebenfalls an jedes *Oplog* von den primären MongoDB Instanz zu hängen und die Schreiboperationen gegen die aktiven *Real Time Queries* zu prüfen und gegebenenfalls Änderungen an den Klienten weiter zu geben [Win17b].
Der Vorteil von Meteor ist, dass es die vollständige *Query DSL* von MongoDB versteht. Somit wird auch die vollständige *Geo Query DSL* unterstützt. Der Nachteil von Meteor ist allerdings, dass es neben Queries mit der *skip* Option, auch Geo-Queries nur mit dem alten *Change Monitoring + Poll and Diff* unterstützt. Dies wurde bei der Fertigstellung des *Oplogs* mit Version 0.7.2 angekündigt [DeB18]. Dieses Problem wurde bis zum Schreiben dieser Arbeit (Februar 2018) nicht gelöst [Col18]. Somit sind Geo-Real-Time-Queries in Meteor nur mit den zuvor genannten Nachteilen des *Change Monitoring + Poll and Diff* nutzbar und nicht mit dem effizienteren *Oplog Tailing*.

7 Ausblick

Dieses Kapitel enthält einige Erweiterungen, die es nicht in die Geo-Komponente geschafft haben. Die Priorität für diese Punkte war zu gering, um sie zeitlich, während dieser Arbeit einzubringen. Allerdings tragen sie dazu bei, dass die geodätische Auswertung von Orestes vollständiger gegenüber der von MongoDB wird.

7.1 $geoIntersects Anfragen

Der *$geoIntersects* Query Selektor wurde nicht von MongoDB übernommen. Der Grund hierfür ist, dass auch die reguläre Query API von Baqend derzeit keine Möglichkeit bietet *$geoIntersects* Anfragen zu nutzen [Baq17a] (nur gekapselt in einer *where* Anfrage). Aus diesem Grund war die Priorität sehr gering diesen Operator neben *$geoWithin* und *$nearSphere* mit einzubringen. Allerdings ist der Implementationsaufwand für diesen Operator sehr gering, aufgrund der bereits bestehenden Strukturen. Die *GeoExpression* Klasse ist bereits in der Lage die Elemente einer *$geoIntersect* Anfrage zu übersetzen. Dabei wird das Prädikat (*Intersect*) extrahiert und die Formen werden äquivalent zu denen der *$geoWithin* Anfragen übersetzt. Somit muss lediglich eine weitere Prädikatklasse für den *$geoIntersect* Operator eingeführt werden. Der Konstruktor dieser Klasse ist analog zu dem vom *$geoWithin* Operator zu strukturieren. Lediglich das Prädikat *Intersect* muss an die *GeoExpression* Klasse gegeben werden, anstatt des *Within* Prädikats. Auch der Ablauf der *computeMatchingStatus* Methode ist analog zu der von *$geoWithin*. Es muss anstatt der *contains* Beziehung eine weitere *intersects* Beziehung am *GeometryContainer* implementiert werden. Diese Methode muss die korrekte Berechnung für die *intersect* Beziehung mithilfe von S2 für alle möglichen Formen berechnen. Ist dies erfüllt, kann diese in der *computeMatchingStatus* Methode aufgerufen werden und der Operator ist einsatzfähig.

7.2 $geoWithin Anfragen auf LineStrings

Derzeit ist die *$geoWithin* Implementation dieser Arbeit vollständig gegenüber der von MongoDB, bis auf das Enthaltensein von *LineStrings*. Der Grund hierfür ist, dass die Methode *intersectsWithPolyline* der *S2Polygon* Klasse nicht in der Java Implementation von S2 existiert. Da auch der *MultiLineString* und die *GeometryCollection* auf diese Methode zurückgreifen würden, können für diese Formen derzeit ebenfalls nicht die Inhaltsbeziehung berechnet werden. Aufgrund der Komplexität und dass einige Substrukturen fehlen, die für die Umsetzung der Methode nötig sind, wurde das Problem nach hinten gestellt. Dies geschah zugunsten von anderen Features, wie der *BigPolygon* Implementation. Dabei haben zwei Gründe dazu geführt, dass dieses Feature gegenüber anderen nach hinten gestellt wurde: die Umsetzung dieser Logik ist nicht Teil der Arbeit, sondern

sollte Teil der S2 Bibliothek sein, da sie vollumfänglich für die sphärischen Berechnungen eingesetzt wird. Ob die gewünschte Funktionalität noch von Google nachgereicht wird ist nicht auszuschließen, da am 19. April 2016 bestätigt wurde, dass ein Update in Arbeit ist [Goo18b]. Am 21. April 2017 wurde dies noch einmal bestätigt [Goo18b]. Eine Veröffentlichung dieses Updates geschah nicht mehr im Rahmen dieser Arbeit.
Der zweite Grund ist, dass die Query Schnittstelle von Baqend ebenfalls ausschließlich Punkte in Polygonen prüft [Baq17a] (Ausnahme: *where* Bedingung). Somit bleibt die Implementation dieses Features offen für die Zukunft.

7.3 Einführung eines GeoJSON Attribut Typs in der Baqend-Platform

Zwar ist die geobasierte Auswertung in Orestes, bis auf die zuvor erwähnten Punkte, vollständig. Allerdings ist diese nicht vollständig in der Baqend-Platform nutzbar. Nutzt der Klient der Orestes Schnittstelle den *$nearSphere* Query Selektor, kann er lediglich nach *GeoPoint*s suchen, nicht aber nach anderen GeoJSON Formen. Ein Problem ist, dass GeoJSON nicht als Attribut Typ bei der Schema Spezifikation zur Verfügung steht. Um dieses Problem zu umgehen, kann jedoch der Attribut Typ *JSONObject* genutzt werden. So ist es zumindest möglich GeoJSON Objekte zu speichern. Es ist allerdings nicht von der Oberfläche aus möglich, auf dieses *JSONObject* Attribut einen *2dsphere* Index zu setzen. Bei *$geoWithin* Anfragen bildet dies kein großes Problem. Wie in Abschnitt 3.3.3 beschrieben, ist der *$geoWithin* Query Selektor auch uneingeschränkt ohne einen Index nutzbar. Anfragen sind allerdings langsamer auf Kollektionen ohne Index.
Das Kernproblem, worauf in diesem Abschnitt allerdings aufmerksam gemacht werden soll, ist die Nutzung des *$nearSphere* Query Selektors. Dieser benötigt einen Index, um angewendet werden zu können (s.a. Abschnitt 3.3.3). Würde *$nearSphere* nun zur Anfrage auf das unindexierte *JSONObject* Attribut angewendet, resultiert die *Query Subscription* in einer Fehlermeldung.
Eine einfache und schnelle Lösung wäre es, dem Nutzer im Baqend Dashboard zu ermöglichen, dem Attribut Typ *JSONObject* einen *2dsphere* Index zuzuweisen. Dies würde den Nutzer allerdings nicht daran hindern auch Daten in das Attribut zu speichern, die kein GeoJSON Objekt sind. Eine zweite umfassendere Lösung wäre es, einen GeoJSON Attribut Typ zur Auswahl zu stellen. Dieser ist intern ebenfalls als *JSONObject* modelliert und es kann ein *2dsphere* Index auf dieses Attribut gesetzt werden. Dies ist gleichzeitig der einzig mögliche Index und wäre somit mit der bestehenden Oberfläche des Baqend Dashboards konformer, als Lösung eins. Ebenfalls kann bei dieser Lösung bereits beim Einfügen von Datensätzen geprüft werden, ob es sich um ein valides GeoJSON Objekt handelt.

8 Fazit

In dieser Arbeit wurde gezeigt, wie geodätische Anfragen für Real-Time-Datenbanken realisiert werden können. Dabei wurde in Kapitel 1 eine Einführung in das Thema der geodätischen Berechnungen gegeben und was sich für ein Problem im Kontext zu inkrementeller Auswertung ergibt.
Daraufhin wurde in Kapitel 2 der Kontext der Real-Time fähigen Datenbanken und deren theoretisches Konzept im Detail beleuchtet. Dabei wurde die Systemarchitektur von Baqend als konretes Beispiel herangezogen.
In Kapitel 3 wurde ein Einblick in die relevanten Themen gegeben. Zuerst wurde dabei auf die Internas der S2 Bibliothek eingegangen. In dessen Verlauf wurde gezeigt, warum diese Bibliothek besonders performant ist. S2 bildet die Grundlage der geodätischen Berechnungen von MongoDB und auch dieser Arbeit. Anschließend wurde der GeoJSON Standard vorgestellt, welcher einen Großteil von MongoDBs Geo-Query-Sprache ausmacht. Zum Schluss der Grundlagen wurden weitere Inhalte zu MongoDBs Geo-Queries und dessen Internas geliefert.
In Kapitel 4 wurde auf Basis der zuvor dargestellten Grundlagen eine Komponente in Orestes programmiert, die in der Lage ist, Geo-Queries inkrementell auszuwerten. Dabei wurde ein besonderes Augenmerk darauf gelegt, dass die Auswertung dieser Queries das gleiche Ein- und Ausgabeverhalten von MongoDB in der Version 3.4 besitzen. Dies wurde versucht durch die Testabdeckung nachzuweisen. MongoDB war gleichzeitig der erste konkrete Anwendungsfall für den produktiven Einsatz von InvaliDB beim Startup Baqend. Dank ihrer Infrastruktur und bereitgestellten Schnittstellen, war es problemlos möglich die entwickelte Geo-Komponente in einem realen Kontext zu testen. Dies geschah mithilfe einer sogenannten Proof-of-Concept Applikation. Hier konnte die Funktionalität der Komponente und ihre Real-Time Fähigkeit nachgewiesen werden. Es konnte ebenfalls gezeigt werden, dass die Geo-Komponente keinen Flaschenhals besitzt. Allerdings war es nicht mehr möglich die lineare Skalierbarkeit von InvaliDB mit Geo-Queries zu zeigen, da hierfür nicht die Zeit und Ressourcen für einen geeigneten Testaufbau bereit standen.
In Kapitel 6 wurden einige weitere Real-Time fähige Datenbanken mit ihrer Geo-Query DSL vorgestellt. Dabei wurde jeder Kandidat gegen diese Arbeit verglichen.
Schließlich wurde in Kapitel 7 dargestellt, welche weiteren Features für die Geo-Komponente vorgesehen waren. Diese haben es jedoch aus zeitlichen Gründen nicht in diese Arbeit geschafft.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[Alm17] ALMANAC, Astronomical: *Selected Astronomical Constants, 2011*. https://web.archive.org/web/20130826043456/http://asa.usno.navy.mil/SecK/2011/Astronomical_Constants_2011.txt. Version: November 2017

[app18a] APPBASE.IO: *Appbase.io Docs: GETing or Streaming Data*. http://docs.appbase.io/rest-quickstart.html#geting-or-streaming-data. Version: Februar 2018

[app18b] APPBASE.IO: *Appbase.io UseCases: Realtime Geolocation Queries*. https://appbase.io/usecases/geo-queries. Version: Januar 2018

[Baq17a] BAQEND: *Baqend: Queries*. http://www.baqend.com/guide/topics/queries/. Version: Oktober 2017

[Baq17b] BAQEND: *Baqend Real-Time Queries*. https://www.baqend.com/guide/topics/realtime/. Version: Mai 2017

[Baq18a] BAQEND: Baqend Getting Started: Install Baqend. (2018), Januar. https://www.baqend.com/guide/gettingstarted/#install-baqend

[Baq18b] BAQEND: *Baqend Platform Starter Kits*. https://www.baqend.com/guide/starter-kits/. Version: Januar 2018

[BF18] BRAUBACH, Prof. Dr. L. ; FEDERRATH, Prof. Dr. H.: *VIS Folien 12 Cloud Computing*, Januar 2018

[Bru12] BRUMMELEN, P. van: *Heavenly Mathematics*. 1. Princeton University Press, 2012. – 208 S. – ISBN 978–0691148922

[BSSL17] BUTLER, Howard ; SCHMIDT, Christopher ; SPRINGMEYER, Dane ; LIVNI, Josh: *About SpatialReference*. http://spatialreference.org/about/. Version: Dezember 2017

[Col18] COLEMAN, Tom: *GitHub Meteor Docs Oplog Observe Driver*. https://github.com/meteor/docs/blob/version-NEXT/long-form/oplog-observe-driver.md. Version: Februar 2018

[CP18] CUTRESS, Ian ; PCMAG: *Intel Xeon E5 2620 v2 vs Core i7 6700K*. http://cpuboss.com/cpus/Intel-Xeon-E5-2620-v2-vs-Intel-Core-i7-6700K. Version: Januar 2018

[Dat17] DATAGENETICS: *Hilbert Curves*. http://datagenetics.com/blog/march22013/. Version: Dezember 2017

[DeB18] DEBERGALIS, Matt: *Meteor 0.7.2: completing our work scaling realtime MongoDB queries*. https://blog.meteor.com/meteor-0-7-2-completing-our-work-scaling-realtime-mongodb-queries-76f231a69467. Version: Februar 2018

[Dis17] DISHMAN, Lydia: *How Foursquare is Building A "Humane"Map Framework To Rival Google's*. https://www.fastcompany.com/3007394/how-foursquare-building-humane-map-framework-rival-googles. Version: Juli 2017

[ela18a] ELASTIC: *Elasticsearch: Geo Queries*. https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-queries.html. Version: Januar 2018

[ela18b] ELASTIC: *Elasticsearch: Geo Shape*. https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-shape.html. Version: Januar 2018

[Fla07] FLANAGAN, David: *JavaScript, das umfassende Refernzwerk*. Bejing : O'Reilly, 2007. – ISBN 978–3–89721–491–0

[For17] FORCE, Internet Engeneering T.: *RFC 7946*. https://tools.ietf.org/html/rfc7946. Version: Dezember 2017

[Gad10] GADE, Kenneth: A non-singular horizontal position representation. In: *The journal of navigation* 63 (2010), Nr. 3, S. 395–417

[Gar18] GARRETT, Jesse J.: *Ajax: A New Approach to Web Applications*. http://www.adaptivepath.org/ideas/ajax-new-approach-web-applications/. Version: Januar 2018

[gol17a] GOLANG: *Go S2 Package Documentation: S2Cap*. https://godoc.org/github.com/golang/geo/s2. Version: November 2017

[gol17b] GOLANG: *GoDoc: EdgeOrVertexCrossing*. https://godoc.org/github.com/golang/geo/s2#EdgeOrVertexCrossing. Version: Dezember 2017

[Goo17a] GOOGLE: *Firebase*. https://firebase.google.com. Version: Dezember 2017

[Goo17b] GOOGLE: *Firebase Realtime Database*. https://firebase.google.com/products/realtime-database/. Version: Dezember 2017

[Goo17c] GOOGLE: *S1Angle Degrees*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S1Angle.java#L105. Version: November 2017

[Goo17d] GOOGLE: *s2-geometry-library*. https://code.google.com/archive/p/s2-geometry-library/. Version: Juli 2017

[Goo17e] GOOGLE: *s2-geometry-library-java*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Point.java. Version: Juli 2017

[Goo17f] GOOGLE: *S2 Geometry Library Java: S1Angle*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S1Angle.java#L58. Version: September 2017

[Goo17g] GOOGLE: *S2 Geometry Library Java: S2latLng*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2LatLng.java#L169. Version: September 2017

[Goo17h] GOOGLE: *S2 Geometry Library Java: S2Loop*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Loop.java#L31. Version: September 2017

[Goo17i] GOOGLE: *S2 Geometry Library Java: S2Loop contains*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Loop.java#L388. Version: Dezember 2017

[Goo17j] GOOGLE: *S2 Geometry Library Java: S2Polygon*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Polygon.java#L34. Version: September 2017

[Goo17k] GOOGLE: *S2Cap Berechnung des Öffnungswinkels*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Cap.java#L121. Version: November 2017

[Goo17l] GOOGLE: S2Cap Intersects S2Cell. (2017), November. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Cap.java#L331

[Goo17m] GOOGLE: *S2Cap Punkt in S2Cap enthalten*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Cap.java#L387. Version: November 2017

[Goo17n] GOOGLE: *S2Cap Winkelberechnung*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2Cap.java#L73. Version: November 2017

[Goo17o] GOOGLE: *S2LatLng Normalisierung*. https://github.com/google/s2-geometry-library-java/blob/master/src/com/google/common/geometry/S2LatLng.java#L148. Version: November 2017

[Goo18a] GOOGLE: *Geofire*. https://github.com/firebase/geofire. Version: Januar 2018

[Goo18b] GOOGLE: *GitHub S2 Geometry Library*. https://github.com/google/s2-geometry-library-java/issues/3. Version: Januar 2018

[Goo18c] GOOGLE: *Google SignIn*. https://accounts.google.com/signin. Version: Januar 2018

[Gra82] GRAHAM, A. K.: Computer software: Software design: Breaking the bottleneck: A visionary approach is recommended in place of the piecemeal policy now in effect. In: *IEEE Spectrum* 19 (1982), March, Nr. 3, S. 43–50. http://dx.doi.org/10.1109/MSPEC.1982.6366826. – DOI 10.1109/MSPEC.1982.6366826. – ISSN 0018–9235

[GSW+17] GESSERT, Felix ; SCHAARSCHMIDT, Michael ; WINGERATH, Wolfram ; WITT, Erik ; YONEKI, Eiko ; RITTER, Norbert: Quaestor: query web caching for database-as-a-service providers. In: *Proceedings of the VLDB Endowment* 10 (2017), Nr. 12, S. 1670–1681

[Hac62] HACKER, Richard: Certification of Algorithm 112: Position of point relative to polygon. In: *Commun. ACM* 5 (1962), Nr. 12, 606. http://dblp.uni-trier.de/db/journals/cacm/cacm5.html#Hacker62

[Hal18] HALL, Susan: *AppBase.io Offers a Streaming NoSQL Service for Real-Time Apps*. https://thenewstack.io/appbase-io-offers-streaming-nosql-service-real-time-apps/. Version: Januar 2018

[Hil91] HILBERT, David: Ueber die stetige Abbildung einer Linie auf ein Flächenstück. In: *Mathematische Annalen* 38 (1891), Nr. 3, S. 459–460

[Hil35] In: HILBERT, David: *Über die stetige Abbildung einer Linie auf ein Flächenstück*. Springer, 1935. – ISBN 978–3–662–38452–7

[HMTT08] In: HADJIELEFTHERIOU, Marios ; MANOLOPOULOS, Yannis ; THEODORIDIS, Yannis ; TSOTRAS, Vassilis J.: *R-Trees – A Dynamic Index Structure for Spatial Searching*. Springer, 2008. – ISBN 978–0–387–35973–1

[Inc17a] INC., MongoDB: *2d Index Internals*. https://docs.mongodb.com/v3.4/core/geospatial-indexes/. Version: Dezember 2017

[Inc17b] INC, MongoDB: *Define Location Range for a 2d Index*. https://docs.mongodb.com/v3.4/tutorial/build-a-2d-index/#geospatial-indexes-range. Version: Dezember 2017

[Inc17c] INC., MongoDB: *MongoDB 3.0 features: Big Polygon*. https://www.mongodb.com/blog/post/mongodb-30-features-big-polygon. Version: Dezember 2017

[Inc17d] INC, MongoDB: *MongoDB Code: GeometryContainer containsPoint Methode*. https://github.com/mongodb/mongo/blob/r3.4.8/src/mongo/db/geo/geometry_container.cpp#L365. Version: Dezember 2017

[Inc17e] INC., MongoDB: *MongoDB Code: GeometryContainer Implementation*. https://github.com/mongodb/mongo/blob/r3.4.8/src/mongo/db/geo/geometry_container.cpp. Version: Dezember 2017

[Inc17f] INC., MongoDB: *MongoDB Code: GeoParser CRS Definitionen*. https://github.com/mongodb/mongo/blob/r3.4.8/src/mongo/db/geo/geoparser.cpp#L71. Version: Dezember 2017

[Inc17g] INC., MongoDB: *MongoDB Code: GeoParser.parseGeoJSONPolygonCoordinates*. https://github.com/mongodb/mongo/blob/r3.4.8/src/mongo/db/geo/geoparser.cpp#L176. Version: November 2017

[Inc17h] INC, MongoDB: *MongoDB Documentation 2D Indexes*. https://docs.mongodb.com/v3.4/core/2d/. Version: August 2017

[Inc17i] INC, MongoDB: *MongoDB Documentation 2D Indexes: Behavior*. https://docs.mongodb.com/v3.4/core/2d/#behavior. Version: August 2017

[Inc17j] INC, MongoDB: *MongoDB Documentation 2D Indexes: Considerations*. https://docs.mongodb.com/v3.4/core/2d/#considerations. Version: August 2017

[Inc17k] INC, MongoDB: *MongoDB Documentation 2dsphere Indexes*. https://docs.mongodb.com/v3.4/core/2dsphere/. Version: August 2017

[Inc17l] INC, MongoDB: *MongoDB Documentation 2dsphere Indexes: Considerations*. https://docs.mongodb.com/v3.4/core/2dsphere/#considerations. Version: August 2017

[Inc17m] INC, MongoDB: *MongoDB Documentation 2dsphere Indexes: Shard Key Restrictions*. https://docs.mongodb.com/v3.4/core/2dsphere/#shard-key-restrictions. Version: August 2017

[Inc17n] INC, MongoDB: *MongoDB Documentation $box*. https://docs.mongodb.com/v3.4/reference/operator/query/box/#op._S_box. Version: November 2017

[Inc17o] INC, MongoDB: *MongoDB Documentation $center*. https://docs.mongodb.com/v3.4/reference/operator/query/center/#op._S_center. Version: November 2017

[Inc17p] INC, MongoDB: *MongoDB Documentation $centerSphere*. https://docs.mongodb.com/v3.4/reference/operator/query/centerSphere/#op._S_centerSphere. Version: November 2017

[Inc17q] INC, MongoDB: *MongoDB Documentation GeoJSON: Polygon*. https://docs.mongodb.com/v3.4/reference/geojson/#polygon. Version: September 2017

[Inc17r] INC, MongoDB: *MongoDB Documentation $geometry*. https://docs.mongodb.com/v3.4/reference/operator/query/geometry/. Version: August 2017

[Inc17s] INC, MongoDB: *MongoDB Documentation Geospatial Queries*. https://docs.mongodb.com/v3.4/geospatial-queries/. Version: November 2017

[Inc17t] INC, MongoDB: *MongoDB Documentation Geospatial Query Operators*. https://docs.mongodb.com/v3.4/reference/operator/query-geospatial/. Version: Dezember 2017

[Inc17u] INC, MongoDB: *MongoDB Documentation $geoWithin*. https://docs.mongodb.com/v3.4/reference/operator/query/geoWithin/. Version: August 2017

[Inc17v] INC, MongoDB: *MongoDB Documentation $nearSphere*. https://docs.mongodb.com/v3.4/reference/operator/query/nearSphere/. Version: September 2017

[Inc17w] INC, MongoDB: *MongoDB Documentation $polygon*. https://docs.mongodb.com/v3.4/reference/operator/query/polygon/#op._S_polygon. Version: November 2017

[Inc17x] INC., MongoDB: *New Geo Features in Mongo 2.4*. https://www.mongodb.com/blog/post/new-geo-features-in-mongodb-24. Version: Juli 2017

[Inc18a] INC., MongoDB: *MongoDB Documentation $minDistance. Februar$*2018

[Inc18b] INC, MongoDB: *MongoDB Documentation Replica Set Oplog*. https://docs.mongodb.com/v3.4/core/replica-set-oplog/. Version: Januar 2018

[Inc18c] INC, MongoDB: *MongoDB Documentation Replication*. https://docs.mongodb.com/v3.4/replication/. Version: Februar 2018

[Inc18d] INC, MongoDB: *MongoDB Documentation Sharding*. https://docs.mongodb.com/v3.4/sharding/. Version: Februar 2018

[Jam03] JAMES, Doug: *Spatial Data Structures*. November 2003

[Joh17] JOHNSON, Nick: *Damn Cool Algorithms: Spatial indexing with Quadtrees and Hilbert Curves*. http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves. Version: Dezember 2017

[KBD13] KAMBONA, Kennedy ; BOIX, Elisa G. ; DE MEUTER, Wolfgang: An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. New York, NY, USA : ACM, 2013 (DYLA '13). – ISBN 978–1–4503–2041–2, 3:1–3:9

[KGM94] In: KAO, Ben ; GARCIA-MOLINA, Hector: *An Overview of Real-Time Database Systems*. Springer, 1994. – ISBN 978–3–642–88049–0

[KRA02] KANTH, Kothuri Venkata R. ; RAVADA, Siva ; ABUGOV, Daniel: Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In: FRANKLIN, Michael J. (Hrsg.) ; MOON, Bongki (Hrsg.) ; AILAMAKI, Anastassia (Hrsg.): *SIGMOD Conference*, ACM, 2002. – ISBN 1–58113–497–5, 546–557

[Kuh18] KUHN, Josh: *RethinkDB 1.15: Geospatial queries*. https://rethinkdb.com/blog/1.15-release/. Version: Januar 2018

[Lan15] LANE, Kin: *Overview of the backend as a service (BaaS) space*. 2015

[Loc17] LOCATIONTECH: *Spatial4j*. https://github.com/locationtech/spatial4j. Version: Oktober 2017

[Mic18] MICROSOFT: *About latitude and longitude*. https://msdn.microsoft.com/en-us/library/aa578799.aspx. Version: Januar 2018

[MP13] MIKOWSKI, Michael S. ; POWELL, Josh C.: *Single Page Web Applications*. Manning, 2013. – 432 S. – ISBN 9781617290756

[Mun99] MUNSHI, Ritabrata: The Jordan curve theorem. In: *Resonance* 4 (1999), Nr. 9, 32–37. http://dx.doi.org/10.1007/BF02834230. – DOI 10.1007/BF02834230. – ISSN 0973–712X

[Ora17] ORACLE: *MySQL Spatial Analysis Functions*. https://dev.mysql.com/doc/refman/5.7/en/spatial-analysis-functions.html. Version: November 2017

[Par17] PARETO SOFTWARE, LLC: *SimpleMap Worldmap*. https://simplemaps.com/static/demos/resources/svg-library/svgs/world.svg. Version: Dezember 2017

[Per17] PERONE, Christian S.: *Google's S2, geometry on the sphere, cells and Hilbert curve*. http://blog.christianperone.com/2015/08/googles-s2-geometry-on-the-sphere-cells-and-hilbert-curve/. Version: Oktober 2017

[Pro17] PROCOPIUC, Octavian: *Geometry on the Sphere: Google's S2 Library*. https://docs.google.com/presentation/d/1Hl4KapfAENAOf4gv-pSngKwvS_jwNVHRPZTTDzXXn6Q/view?pli=1#slide=id.i0. Version: Juli 2017

[Ran14] RANJAN, Rajiv: Streaming Big Data Processing in Datacenter Clouds. In: *IEEE Cloud Computing* 1 (2014), Nr. 1, 78–83. http://dblp.uni-trier.de/db/journals/cloudcomp/cloudcomp1.html#Ranjan14

[Rea17] REACTIVEX: *ReactiveX*. http://reactivex.io/. Version: Oktober 2017

[Ret17] RETHANS, Derick: $geoWithin with $centerSphere does not find GeoJSON documents other than Points. (2017), November. https://jira.mongodb.org/browse/SERVER-27968

[Ret18a] RETHINKDB: RethinkDB. (2018), Januar. https://www.rethinkdb.com/

[Ret18b] RETHINKDB: *RethinkDB Api: ReQL command: geojson*. https://rethinkdb.com/api/javascript/geojson/. Version: Januar 2018

[Ret18c] RETHINKDB: *RethinkDB Api: ReQL command: getIntersecting*. https://rethinkdb.com/api/javascript/get_intersecting/. Version: Januar 2018

[Ret18d] RETHINKDB: *RethinkDB Api: ReQL command: getNearest*. https://rethinkdb.com/api/javascript/get_nearest/. Version: Januar 2018

[Ret18e] RETHINKDB: *RethinkDB Api: ReQL command: includes*. https://rethinkdb.com/api/javascript/includes/. Version: Januar 2018

[Ret18f] RETHINKDB: *RethinkDB Api: ReQL command: intersects*. https://rethinkdb.com/api/javascript/intersects/. Version: Januar 2018

[S2G18a] S2GEOMETRY: *S2Geometry: Earth Cube*. https://s2geometry.io/resources/earthcube. Version: Januar 2018

[S2G18b] S2GEOMETRY: *S2Geometry Overview*. https://s2geometry.io/about/overview. Version: Februar 2018

[Sam84] SAMET, Hanan: The quadtree and related hierarchical data structures. In: *ACM Computing Surveys (CSUR)* 16 (1984), Nr. 2, S. 187–260

[Sch09] SCHMIDT, Stephan: *PHP Design Patterns*. O'Reilly, 2009

[Sch17] SCHWARTZ, Joe: *Bing Maps Tile System*. https://msdn.microsoft.com/en-us/library/bb259689.aspx. Version: September 2017

[Suc] SUCCO, Stephan: *Skalierbare, echtzeitnahe Kommunikation von Änderungen an Datenbankobjekten und Abfrageresultaten innerhalb einer Backend-as-a-Service-Architektur*, Diplomarbeit

[Tor02] TORGE, Wolfgang: *Geodäsie*. De Gruyter, 2002 (2). – 380 S. – ISBN 978–3110175455

[Vor10] VORNBERGER, Prof. Dr. O.: *Computergrafik*. Vorlesungsskriptkript, März 2010

[VSI17] VSIS, Universität H.: *Orestes*. https://vsis-www.informatik.uni-hamburg.de/vsis/research/lookproject/55. Version: oct 2017

[Wen18] WENGER, Jacob: *GeoFire 2.0*. https://firebase.googleblog.com/2014/06/geofire-20.html. Version: Januar 2018

[Wik17] WIKIPEDIA: Jordan Curve Theorem. (2017), Dezember. https://en.wikipedia.org/wiki/Jordan_curve_theorem

[Win17a] WINGERATH, Wolfram: *A Real-Time Database Survey: The Architecture of Meteor, RethinkDB, Parse & Firebase*. https://medium.baqend.com/real-time-databases-explained-why-meteor-rethinkdb-parse-and-firebase-dont-scale-822ff87d2f87. Version: Dezember 2017

[Win17b] WINGERATH, Wolfram: *Real-Time Databases Explained: Why Meteor, RethinkDB, Parse and Firebase Don't Scale*. https://medium.com/@Wolle/822ff87d2f87. Version: Mai 2017

[Winht] WINGERATH, Wolfram: *Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases*, Diss., Noch nicht veröffentlicht

[Zim99] ZIMMERMANN, Phil: *Introduction to Cryptography*. 1999. – 86 S.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den _____ Unterschrift: _____