Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Masterarbeit

Benchmarking von Real-Time Datenbanken

Marian Andre Schaub

Marian.Andre.Schaub@studium.uni-hamburg.de
Studiengang IT-Management und -Consulting
Matr.-Nr. 6592515
Fachsemester 11

Erstgutachter: Prof. Dr. -Ing Norbert Ritter
Zweitgutachter: Wolfram Wingerath

Abgabe: 11.2018

Inhaltsverzeichnis

1 Einleitung

Moderne Webanwendungen vereinen zwei Eigenschaften. Sie sind zum einen meist hochgradig interaktiv. Eine Vielzahl an Nutzern greift auf die bereitgestellten Informationen zu, modifiziert diese oder generiert gänzlich neue Daten. Dieses Verhalten führt zu der zweiten Eigenschaft: die großen Masse an Daten, welche von den Anwendungen gehandhabt werden muss. Während klassische Webseiten nur statische Inhalte angezeigt haben, ist der Nutzer heute an Anwendungen gewöhnt, die Echtzeitinteraktion mit anderen Nutzern und Daten ermöglichen [LL13].

Reaktive Anwendungen sind für diese Fälle prädestiniert. Frontend-Frameworks wie *React*[1] oder *Angular*[2] gewinnen stark an Popularität, da sie den Entwickler bei der Handhabung der Interaktivität unterstützen. Die Konzepte der reaktiven Programmierung sind nicht neu, finden dank dieser Aufmerksamkeit ihren Weg jedoch in immer mehr Programmiersprachen und Frameworks. Nicht nur für die Interaktion zwischen Nutzer und Anwendung werden sinnvolle Abstraktionen geboten, auch Ereignisse zwischen der Clientanwendung und dem Server können dank der gebotenen Konstrukte einfacher verarbeitet werden [Me18c][Lo16]. Gerade, weil diese Art von Applikationen stark auf Daten basieren, spielen Datenbanken für diesen Kontext eine kritische Rolle. Klassische Datenbank Mangement System sind nicht auf Grundlage der Anforderungen heutiger Webapplikationen entstanden. Sie bieten ein *pull*-basiertes Zugriffsmodell, welches für reaktive Anwendungen einen Bruch bedeutet. Um diese Problematik zu lösen, wurden verschiedene Ansätze entwickelt. Während viele Lösungen ineffizient sind und die Datenbank zusätzlich belasten, haben einige Systeme andere Wege gefunden [WGF+17]. Diese werden in dieser Arbeit als *Real-Time Datenbanken* verstanden. Sie bieten dem Nutzer die Möglichkeit eine Abfrage zu registrieren. Bei Änderungen in den Datensätzen erfolgt automatisch eine Benachrichtigung mit den neuen Daten.

Populäre Vertreter sind beispielsweise *Firebase*, *Meteor* und *RethinkDB*. Auch, wenn diese Systeme in ihrer Funktionalität Parallelen aufweisen, ist eine eindeutige Kategorisierung schwierig. *Meteor* ist ein Framework für die Entwicklung von Webapplikationen und integriert die Entwicklung von Frontend sowie Backend und Datenhaltung über *MongoDB*. *RethinkDB* ist im Gegensatz dazu eine reines Datenbank Management System. *Firebase* ist als *Backend-as-a-Service*-Anbieter zwischen diesen Extremen anzusiedeln [Wi17a]. Die Heterogenität dieser Systeme erschwert es, eine Grundlage für den qualitativen Vergleich der Anbieter zu finden.

Dieser Aspekt stellt den Forschungsauftrag dieser Arbeit dar. Zunächst muss eine Definition gefunden werden, welche die Systeme auf einer gemeinsamen Abstraktionsebene er-

[1] https://reactjs.org/
[2] https://angular.io/

fasst. Auf dieser Grundlage muss erörtert werden, anhand von welchen Dimensionen die Qualität solcher Systeme bewertet werden kann. Dies beinhaltet, dass analysiert werden muss, wie Qualität für die identifizierten Dimensionen messbar gemacht werden kann in Form von Qualitätsmetriken.
Das Benchmarking stellt eine populäre Methode für den Vergleich von Softwaresystemen dar. Mit Hilfe der definierten Qualitätsmetriken wird ein Szenario entwickelt, welches den Rahmen einer Benachmarkingapplikation bildet. Anschließend wird diese für zwei Systeme implementiert, um Vergleichsergebnisse für die beiden Vertreter zu generieren.

Zusammenfassend wird dieses Forschungsvorhaben in der folgenden Forschungsfrage beschrieben:

> *„Was sind aussagekräftige Qualitätsdimensionen für die Bewertung von Real-Time Datenbanken und wie können diese im Rahmen einer Benchmarkingapplikation messbar gemacht werden?“*

1.1 Aufbau der Arbeit

Kapitel 2 erläutert die Grundlagen im Kontext von *Real-Time Datenbanken*. Neben einer Abgrenzung zu verwandten Technologien, wird eine Definition des Begriffs vorgestellt und eine Auswahl bekannter Vertreter wird beschrieben.
Kapitel 3 führt Konzepte und Anforderung von Benchmarks ein. Es bietet einen grundlegenden Überblick über die Methodik im Allgemeinen und im Hinblick auf die betrachteten Systeme.
Kapitel 4 legt die konzeptionelle Grundlage für den Benchmark. Der Qualitätsbegriff wird erläutert und relevante Qualitätsdimensionen werden identifiziert. Auf dieser Grundlage werden Metriken entwickelt, mit Hilfe derer die Dimensionen messbar gemacht werden.
Kapitel 5 beschreibt das Szenario, welches den Rahmen für die Benchmarkanwendung darstellt. Hieraus ergeben sich das Datenmodell und die Belastungsszenarien für den Benchmark.
Kapitel 6 beschreibt die eigentliche Benchmarkapplikation. Die einzelnen Komponenten werden vorgestellt, ein besonderer Schwerpunkt liegt hierbei auf der Datenbankschnittstelle und ihrer Implementierung durch die ausgewählten Vertreter Firebase und Baqend.
Kapitel 7 stellt die Ergebnisse eines mit der Benchmarkapplikation durchgeführten Tests vor.
Kapitel 8 fasst die Beiträge der Arbeit im Kontext der Forschungsfrage zusammen und gibt einen Ausblick über mögliche zukünftige Forschungsvorhaben.

2 Grundlagen von Real-Time Datenbanken

In diesem Kapitel wird dargestellt, mit welcher Motivation *Real-Time Datenbanken* entwickelt wurden. Hierfür ist es zunächst wichtig zu verstehen, welche technologischen Fortschritte einen konkreten Bedarf für solche Systeme geschaffen haben. Daraufhin wird eine Definition konstruiert um das Verständnis des Begriffs *Real-Time Datenbank* im Kontext dieser Arbeit klarzustellen. Abschließend werden Systeme vorgestellt, welche nach diesem Verständnis der Gruppe der *Real-Time Datenbanken* zuzuordnen sind.

2.1 Technologische Wegbereiter

Der Bedarf für ein Konzept wie *Real-Time Datenbanken* hängt mit verwandten Ideen und Technologien zusammen. In diesem Abschnitt werden die Idee der Reaktiven Programmierung und Protokolle für Server-Client-Kommunikation näher beleuchtet.

2.1.1 Reaktive Programmierung

Reaktive Programmierung beschreibt ein Programmierparadigma, welches vor allem im Kontext hochinteraktiver Applikationen Anwendung findet. Bei der *Imperativen Programmierung* ist der Kerngedanke sequentiell Programmschritte abzuarbeiten, welche zu der Lösung des Problems führen. Es wird davon ausgegangen, dass die einzelnen Schritte zeitlich in der vorgegebenen Reihenfolge durchgeführt werden. Eine Herausforderung stellen hierbei Applikationen mit hoher Interaktivität dar. Es ist nicht vorherzusehen, zu welchem Zeitpunkt ein Nutzer eine bestimmte Handlung ausführt oder Benachrichtigungen von einem Server eintreffen. Für solche Anwendungsfälle bietet die Reaktive Programmierung passende Konstrukte [Me18c].

Mezzalira definiert Reaktive Programmierung als „[…] paradigm based on asynchronous data streams that propagate changes during the application life cycle“ [Me18c, S. 11].

Für den Umgang mit Datenströmen werden zwei grundlegende Konstrukte vorgeschlagen [WH00]:

- **Behaviours**: Werte, welche sich zeitabhängig verändern, beispielsweise ein Wert, der die Sekunden der aktuellen Minute repräsentiert.
- **Events**: Eine zeitlich geordnete Abfolge von Ereignissen, welche jeweils einen bestimmten Wert tragen, beispielsweise Mausklick-Interaktionen mit einem bestimmten Element.

Während die Konzepte bereits geraume Zeit existieren, hat vor allem die erste Veröffentlichung des *Reaktiven Manifests*[3] im Jahr 2013 zu einer gesteigerten Popularität des Paradig-

[3] https://www.reactivemanifesto.org/de

mas beigetragen. Mittlerweile haben sich eine Vielzahl von Frameworks entwickelt und viele Sprachen implementieren Elemente der reaktiven Programmierung. Dies führt zu einer weiten Verbreitung von reaktiven Anwendungen [Lo16]. Vor allem im Bereich von *Real-Time Webapplikationen* spielen die Konzepte eine wichtige Rolle. Hier treten sowohl viele Ereignisse von Seiten des Nutzers, wie beispielsweise Mausklicks, als auch viele Ereignisse von Serverseite in Form von veränderten Daten auf [Me18c].

2.1.2 Neuerungen auf Protokollebene

Klassische Client-Server Webapplikationen basieren auf Protokollebene auf dem *Hyper Text Transfer Protocol* (HTTP). Dieses Protokoll bietet die Möglichkeit Anfragen vom Client an den Server zu übermitteln und eine entsprechende Antwort zu erhalten. Es ist jedoch nicht vorgesehen, dass von Seiten des Servers eine Verbindung zum Client aufgebaut wird, um aktuelle Daten zu liefern. Dies erfordert eine explizite vorherige Anfrage. Über *Polling* (periodisches Senden von Anfragen nach Neuigkeiten) und *Long-Polling* (den Kanals so lange offen halten, bis ein Update kommt) kann diese Funktionalität umgesetzt werden. Bei datenintensiven Applikationen ist dies jedoch keine effiziente Methode, weshalb mit *Server Sent Events* und *Websockets* Alternativen entwickelt wurden [Gr13].

Server Sent Events baut als Konzept auf dem vorher genannten *Long-Polling* auf und hält eine *HTTP*-Verbindung dauerhaft offen. Dem Nutzer werden das Verwalten der Verbindung und Nachrichtenverarbeitung abgenommen, wodurch sich *Server-Sent Events* als effiziente und einfach zu handhabende Lösung für Server-Client-Übertragung von text-basierten Daten präsentiert [Gr13].

Websocket ist im Gegensatz zu *Server Sent Events* ein eigenständiges Protokoll und nicht zwingend abhängig von *HTTP*. Dieses wird nur für den initialen Handshake genutzt, kann außerhalb des Browserkontextes bei Bedarf aber auch anderweitig ersetzt werden. Es eignet sich für die bidirektionale Übertragung von sowohl text-basierten Daten, als auch Binärdaten [Gr13].

2.2 Motivation für Real-Time Datenbanken

Auf der Grundlage von *Reaktiver Programmierung* und Protokollen wie beispielsweise *Websockets* können interaktive und datenintensive Applikationen effizient und leistungsstark realisiert werden. Ein Bruch entsteht jedoch bei der Datenhaltung mit klassischen Datenbanksystemen. Ähnlich, wie bereits bei der Client-Server-Kommunikation mit HTTP erläutert, funktionieren diese nach dem Prinzip, dass zunächst eine Anfrage gestellt wird. Die Datenbank liefert daraufhin das für den Zeitpunkt aktuelle Ergebnis zurück. Um zukünftige Veränderungen, welche für die gestellte Anfrage relevant wären, zu erhalten muss nach dem Prinzip des *Polling* die Anfrage periodisch wiederholt werden. Diese Systeme werden auch als *pull-basiert* bezeichnet. Unter Umständen wird bei diesem Ansatz, je

nach Wahl des Aktualisierungsintervalls, die Datenbank unnötigerweise dauerhaft belastet oder Änderungen erreichen den Nutzer nur mit Verzögerung [WGF+17].
Als Reaktion wurden verschiedene Lösungen und Systeme entwickelt. Ein Teil dieser Systeme werden im Folgenden unter dem Begriff *Real-Time Datenbank* zusammengefasst und sind der zentrale Betrachtungspunkt dieser Arbeit. In Abschnitt 2.5 wird kurz auf verwandte Technologien eingegangen, die nicht Kern dieser Betrachtung sind.

2.3 Definition

Als *Real-Time Datenbanken* werden in dieser Arbeit Systeme bezeichnet, welche die Daten auf dem Client automatisch und in Echtzeit mit den aktuellen Daten aus der Datenbank synchronisieren [Wi17a]. Der Echtzeit-Begriff bezieht sich in diesem Kontext jedoch nicht auf eine konkrete Vorgabe von maximalen Antwortzeiten. Dieses Verständnis von dem Begriff *Real-Time Datenbank* ist in der Literatur weit verbreitet [HLC91][KG94][BLS97], in dieser Arbeit jedoch nicht der Fokus.

2.3.1 Öffentliche Schnittstelle

Grundlegend muss jedes System der Kategorie *Real-Time Datenbank* zwei Rollen einnehmen, um die geforderte Funktionalität zu liefern:

- **Persistenter Speicher:** Als Datenbank Management System muss eine Schnittstelle für die Basisoperationen (*CRUD*[4]) bereitgestellt werden. Aus dem Kontext *Relationaler Datenbank Management Systeme* ist die *Structured Query Language (SQL)* als standardisierte Zugriffssprache bekannt. *SQL* bietet Funktionalität für die Datendefinition, Datenmanipulation und Datenabfrage. Systeme, welche nicht dem relationalen Datenmodell folgen, werden häufig unter dem Begriff *NOSQL*[5] zusammengefasst [Wi15]. Das Bereitstellen einer Sprache für Datenmanipulation und Datenabfrage wird vorausgesetzt.

[4] Create, Read, Update und Delete
[5] Ursprünglich als *NoSQL* ein Sammelbegriff für Systeme welche eine andere Zugriffssprache als SQL nutzen. Mittlerweile häufig unter dem Akronym *NOSQL* als *Not only SQL* verstanden, Systeme können als auch SQL neben anderen Zugriffssprachen unterstützen.

- **Subjekt im Beobachter-Muster:** In Abbildung 1 ist das Beobachter-Muster dargestellt. Über eine Schnittstelle muss für Beobachter die Möglichkeit gegeben werden sich bei dem Subjekt zu registrieren (*subscribe*), um über Änderungen in den Datensätzen benachrichtigt zu werden. Zudem muss auch eine Abmeldung (*unsubscribe*) möglich sein. Eine *notify*-Methode, um Beobachter zu benachrichtigen muss nicht Teil der öffentlich zugänglichen Schnittstelle sein.

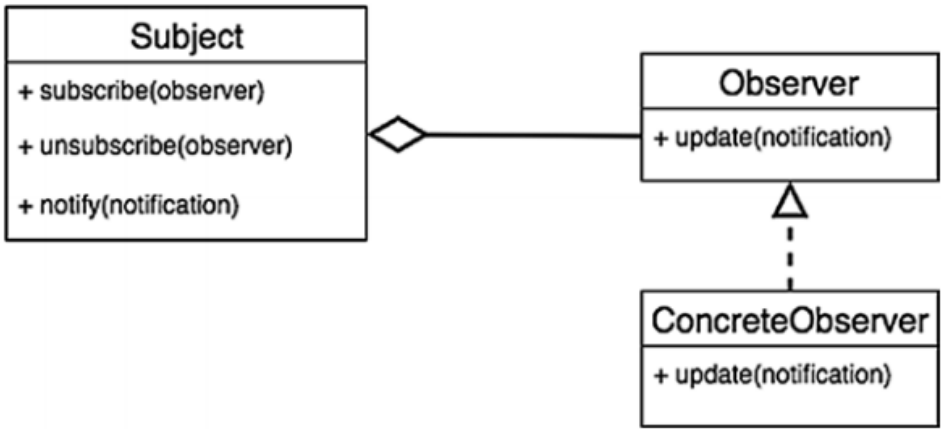

Abbildung 1: Beobachter-Muster in UML (Quelle: [Me18c, S. 69])

Für den Nutzer, bzw. eine Client-Applikation, bietet sich in jedem Fall eine Schnittstelle wie sie in Abbildung 2 beschrieben wird.

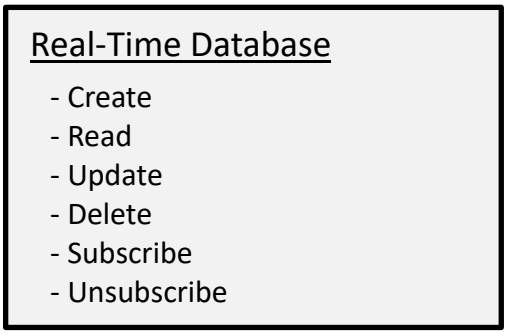

Abbildung 2: Schnittstellen für Real-Time Datenbanken (*Quelle: eigene*)

Read und *Subscribe* sind hierbei die lesenden Methoden. *Read* liefert den aktuellen Zustand der Datenbank zu dem entsprechenden Zeitpunkt. Hierzu muss über eine Abfrage gegebenenfalls eingeschränkt werden, welche Daten von Relevanz sind. *Subscribe* liefert als ersten Rückgabewert das gleiche Ergebnis wie die *Read*-Methode. Dies betrifft alle Daten, welche zum Zeitpunkt der Abfrage bereits in der Datenbank vorhanden sind. Im Folgenden werden diese Daten als *historische Daten* bezeichnet. Wenn neue Daten in der Datenbank landen, welche für die initial gesendete Anfrage relevant sind, sendet die *Real-Time Datenbank* eine Benachrichtigung. In dieser sind die aktualisierten Datensätze, sowie Metainformationen enthalten.

2.4 Bekannte Vertreter

Nach dem definierten Verständnis einer *Real-Time Datenbank* sind diverse Systeme Teil dieser Kategorie. Sie alle bieten die geforderte Funktionalität, unterscheiden sich jedoch hinsichtlich bestimmter Eigenschaften. Wie die Real-Time Funktionalität intern von den einzelnen Systemen realisiert wird, stellt Wingerath [Wi17a] im Detail dar und wird in dieser Arbeit nicht näher behandelt.

In Abbildung 3 ist vereinfacht der Aufbau einer *Real-Time Datenbank* dargestellt. Mit Hilfe dieser Darstellung sollen die Unterschiede visualisiert werden.

Die Unterschiede können wie folgt kategorisiert werden:

- **Transparenz**: Einige Systeme sind als Open Source Software (mit grünem Rahmen um die Komponente *Real-Time Datenbank* dargestellt) frei zugänglich und die inneren Mechanismen sind einsehbar. Andere lassen als proprietäre Software (mit rotem Rahmen um die Komponente *Real-Time Datenbank* dargestellt) keine Rückschlüsse darüber zu, wie Funktionalitäten umgesetzt werden.

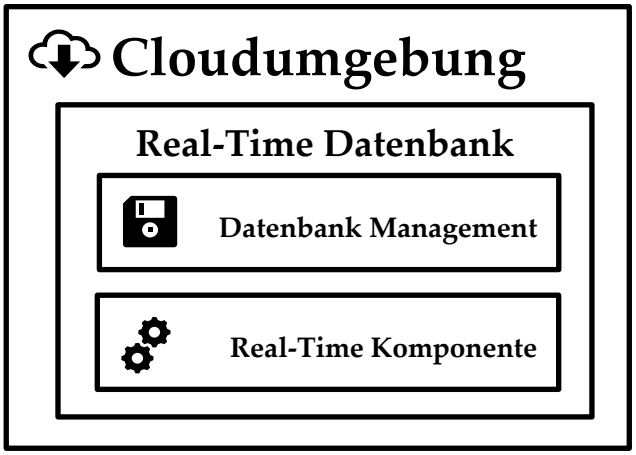

Abbildung 3: Komponenten von Real-Time Datenbanken (Quelle: *eigene*)

- **Umfang**: Während einige Anbieter die Real-Time-Funktionalität als Komponente zu einer bestehenden externen Datenbanklösung bereitstellen, ist in anderen Fällen das gesamte System eine Eigenentwicklung. Externe Komponenten werden mit gelbem Hintergrund dargestellt.
- **Bereitstellung**: Die meisten Systeme können optional als Hosted Service vom Hersteller oder externen Anbietern bereitgestellt werden (Cloudumgebung mit grauem Hintergrund), manche werden jedoch auch ausschließlich als Cloudanwendung angeboten (Cloudumgebung mit blauem Hintergrund).

Firebase Realtime Database

Die *Firebase Realtime Database* ist Teil der Firebase Plattform von Google, welche eine Reihe weiterer Cloud-Dienste zur Verfügung stellt. Sie wurde 2012 auf Grundlage der Chat Applikation *Envolve* entwickelt [Le12]. Im Jahr 2014 wurde diese von Google aufgekauft und durch die Ergänzung weiterer Dienste zu der heutigen Backend-as-a-Service-Plattform aufgebaut [Ta14]. In Abbildung 4 wird die Einordnung hinsichtlich der genannten Unterschiede vorgenommen. Bei *Firebase* handelt es sich um eine **proprietäre Software**. Für die Datenhaltung wird intern eine *Dokumentenorientierte Datenbank* verwendet, welche Daten in Form eines einzelnen JSON-Dokuments speichert. Hierbei handelt es sich um eine **Eigenentwicklung**. Die Bereitstellung über die *Firebase* Cloudplattform ist **nicht optional**, bietet somit die einzige Möglichkeit der Nutzung [Go18d][DB18].

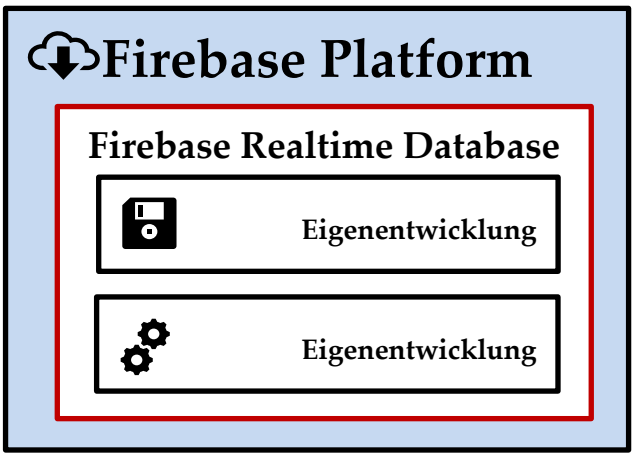

Abbildung 4: Komponenten der *Firebase Realtime Database* (Quelle: *eigene*)

Meteor

Meteor ist ein auf JavaScript basierendes Web-Framework. Es bietet die Möglichkeit Webanwendungen sowohl client-, als auch serverseitig mit JavaScript zu entwickeln. Abbildung 5 ordnet das System wie folgt ein: *Meteor* ist **Open Source** und wird seit 2012 von der *Meteor Development Group* entwickelt [Sc12]. Für die eigentliche Datenhaltung wird mit **MongoDB** eine *dokumentenorientierte Datenbank* verwendet. Während das eigentliche Framework mit jeder Infrastruktur frei verwendet werden kann, bietet *Meteor* mit *Galaxy* seit 2016 **optional** ein eigenes Cloud-Hosting für *Meteor*-Applikationen an [Ba16].

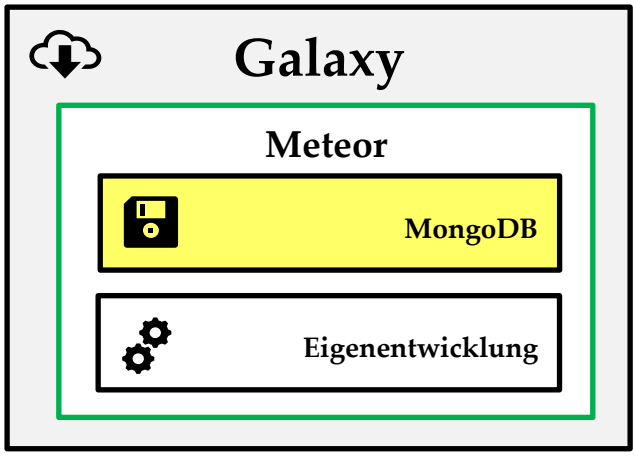

Abbildung 5: Komponenten von *Meteor* (Quelle: *eigene*)

RethinkDB

In Abbildung 6 wird mit *RethinkDB* eine seit 2009 **eigenentwickelte** dokumentenorientierte Datenbank mit nativer Unterstützung für Real-Time-Funktionalität dargestellt [Ak09]. Seit 2012 ist die Software **Open Source** und 2016 stellte das Unternehmen hinter *RethinkDB* den Betrieb ein [Re12][Ak16]. Seitdem wird das Projekt von der *Linux Foundation* verwaltet [Gl17]. Neben *RethinkDB* wurde mit *Horizon*[6] ein Open-Source-Backend entwickelt, um das Erstellen von Real-Time-Webanwendungen auf Basis von *RethinkDB* zu vereinfachen. Als Open-Source-Projekt bietet *RethinkDB* dem Nutzer frei Wahl für die Infrastruktur. **Optional** gibt es einige Cloud-Anbieter, welche zugeschnittenes Cloud-Hosting für *RethinkDB* anbieten[7].

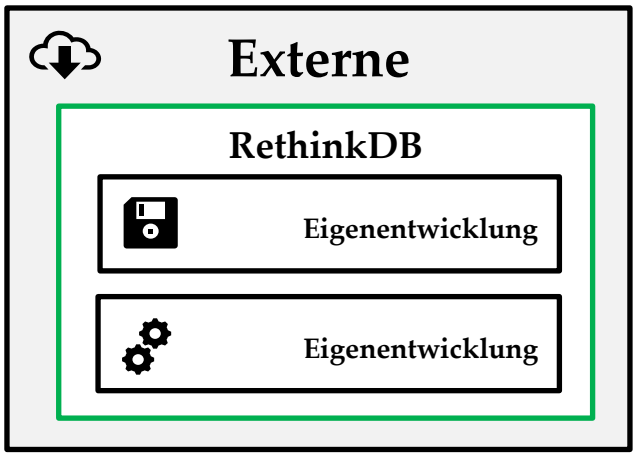

Abbildung 6: Komponenten von *RethinkDB* (Quelle: *eigene*)

Parse

Abbildung 7 beschreibt die Komponenten von *Parse*. Das System wurde 2011 als Backend-as-a-Service veröffentlich [Ki11]. Im Jahr 2013 hat Facebook den Dienst gekauft und weitergeführt [Pu13]. Drei Jahre später wurde von Facebook angekündigt, das Hosting von *Parse* einzustellen. Anfang 2017 wurde dieser Schritt vollzogen und das System als **Open-Source-Projekt** weitergeführt [La17]. Als Datenbank für den *Parse-Server* kann **MongoDB oder PostgreSQL** verwendet werden, wobei Ersteres von den Entwicklern empfohlen wird [Pa18b]. Auch wenn der eigentliche Hosted-Service nicht mehr angeboten wird, gibt es **optional** verschiedene externe Anbieter, wie Microsoft Azure[8] oder Back4App[9].

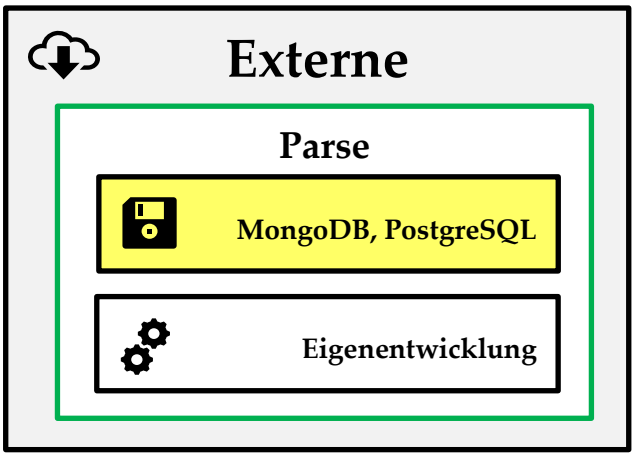

Abbildung 7: Komponenten von Parse (Quelle: *eigene*)

[6] https://github.com/rethinkdb/horizon
[7] Z.B. https://www.compose.com/databases/rethinkdb
[8] https://azuremarketplace.microsoft.com/en-us/marketplace/apps/Microsoft.ParseServer
[9] https://www.back4app.com/

Baqend/InvaliDB

Baqend ist ein Backend-as-a-Service-Anbieter und ist seit 2016 öffentlich verfügbar [Ge16]. Die Real-Time Funktionalität ist seit 2017 verfügbar [Wi17b]. Genutzt wird hierfür die eigenständige Komponente *InvaliDB* [Ge17]. Wie aus Abbildung 8 ersichtlich, findet die Datenhaltung in einer **MongoDB**-Instanz statt [Ba18b]. Der Kern der Baqend Plattform ist **proprietäre Software**. Neben dem Clouddienst existiert eine eingeschränkte *Community Edition*, welche auf eigener Infrastruktur aufgesetzt werden kann. Zudem kann die *Enterprise Edition* mit vollem Funktionsumfang ebenfalls auf eigener Infrastruktur aufgesetzt werden, eine Nutzung der gegebenen Cloudplattform ist somit **optional** [Ba18c].

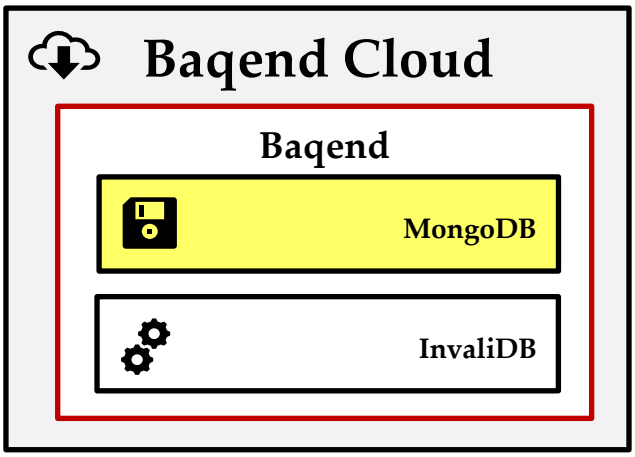

Abbildung 8: Komponenten von Baqend (Quelle: *eigene*)

Bei dem Vergleich der verschiedenen Systeme in Abbildung 4, Abbildung 5, Abbildung 6, Abbildung 7 und Abbildung 8 wird deutlich, dass sie sich hinsichtlich der Abstraktionsebene deutlich unterscheiden. Während *Firebase* als proprietärer Cloudservice für den Nutzer ausschließlich über das Internet als geschlossenes System zugänglich ist, kann *RethinkDB* als Open Source Datenbank in kontrollierter, lokaler Umgebung als offenes System bereitgestellt werden. Gemeinsamkeiten lassen sich in der Datenhaltung erkennen, da alle Systeme eine Form von *dokumentenorientierter Datenbank* nutzen. *Parse* bietet zusätzlich Unterstützung für ein *relationales Datenbank Management System* in *PostgreSQL*.

2.5 Abgrenzung

Neben den beschriebenen Systemen existieren weitere Ansätze um mehr *push-basiertes* Verhalten in der Datenhaltung zu ermöglichen. In Abbildung 9 findet eine Einordnung des Verständnisses von *Real-Time Datenbanken* im Kontext verwandter Technologien statt.

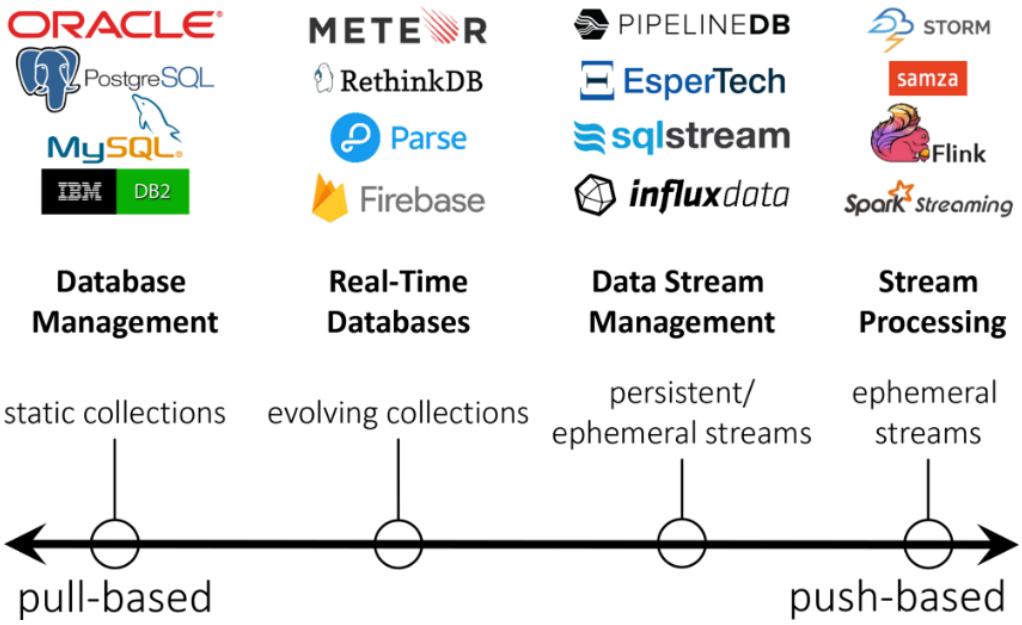

Abbildung 9: Kategorien datenverarbeitender Informationssysteme und das zugrunde liegende Datenverständnis (Quelle: [Wi17a])

Das klassische *Datenbank Management* ist entstanden mit dem Ziel statische Sammlungen von Daten abbilden zu können. Unter dem Begriff *Active Database Systems* werden Systeme zusammengefasst, welche das klassische Verständnis um reaktive Elemente erweitern [PD99]. Konzepte wie *Trigger* und *Event-Condition-Action* Regeln ermöglichen, dass die Datenbanken auf interne und externe Ereignisse reagieren können. Dies vereinfacht die Umsetzung von Systemen, welche Überschneidungen zu den in dieser Arbeit betrachteten Systemen aufweisen. Sie bieten jedoch nicht nativ die geforderten Schnittstellen. Zudem belasten diese Mechanismen die eigentliche Datenbank zusätzlich, welche ohnehin nur eingeschränkte Skalierbarkeit ermöglichen. Einige Datenbank Management Systeme bringen Funktionalität wie *Change Notifications* mit. Der Nutzer wird über Veränderungen benachrichtigt, muss jedoch die veränderten Daten daraufhin erneut abfragen [Wi17a].
Data Stream Management und *Stream Processing* fokussieren sich auf die Verarbeitung und Filterung von Datenströmen. Nur teilweise werden Daten hierbei persistiert. Zudem bieten diese Systeme nativ keinen Zugriff auf *historische Daten*, nur auf eintreffende Daten ab dem Zeitpunkt der Registrierung. Dieses Problem besteht auch bei einigen Systemen, die grundsätzlich Datenbank-Funktionalität mitbringen, wie beispielsweise *OrientDB* und *Graphcool* [Wi17a].

In Abschnitt 2.1.1 wurden mit *Behaviours* und *Events* die zwei unterschiedlichen Konstrukte für die Repräsentation von Datenströmen in der *reaktiven Programmierung* vorgestellt. *Events* lassen sich mit den beschriebenen Systemen des *Data Stream Management* und *Stream Processing* als Abfolge von Ereignissen angemessen repräsentieren. *Behaviours* sind demgegenüber dem Verständnis nach zeitlich veränderliche Werte. Diese existieren nicht erst ab dem Zeitpunkt der ersten Veränderung. *Real-Time Datenbanken* sind Sammlungen von veränderlichen Daten und bieten dank den *historischen Daten* die Möglichkeit einen initialen Wert zu liefern. Eine detaillierte Betrachtung der Abgrenzung und Systeme bietet Wingerath [Wi17a].

3 Vorgehen und Methodik

Nachdem in Kapitel 2 definiert wurde mit welchen Systemen sich diese Arbeit beschäftigt, soll in diesem Kapitel die Methodik vorgestellt werden um die Systeme näher zu untersuchen. Hierzu wird das allgemeine Verständnis von Benchmarking erläutert und herausgearbeitet, welche besonderen Anforderungen sich im Kontext der beschriebenen Systeme ergeben.

3.1 Benchmark als Werkzeug

Der Begriff Benchmark ist weit verbreitet und findet in einer Vielzahl unterschiedlicher Bereiche Verwendung. Während er seinen Ursprung im Bereich des Bau- und Vermessungswesens hat, ist er vor allem ein beliebtes Instrument im Kontext der Betriebswirtschaftslehre und Unternehmensführung [Go06, S. 16]. Als Begründer der Methodik gilt Robert C. Camp, welcher für sein Buch „Benchmarking" als weiterführende Beschreibung folgenden Untertitel wählte: „the search for industry best practices that lead to superior performance" [Ca89]. „Warum und wie machen es Andere besser und was können wir daraus lernen?" [MA09, S. 20] fassen Mertins und Kohl den Kerngedanken in Form einer zentralen Fragestellung zusammen. Spendolini zeigt jedoch auf Grundlage der Untersuchung von 49 gängigen Definitionen auf, dass in vielen Aspekten zwischen den verschiedenen Definitionen keine Einigkeit herrscht, beispielsweise hinsichtlich der Laufzeit eines Benchmarks, der Methodik oder dem Objekt [Spendolini 1992 nach Ul98, 14f].

3.2 Benchmarking in der IT

Unter *IT-Benchmarking* wird im betriebswirtschaftlichen Kontext üblicherweise die Gegenüberstellung von Kosten und Leistung der IT-Abteilung im Vergleich zu konkurrierenden Unternehmen verstanden [MA09]. Aus technischer Sichtweise beschränkt sich die Betrachtung auf einzelne Computersysteme. Klassischerweise wurde hierbei Leistung in einer Form von Durchsatz gemessen, beispielsweise Transaktionen pro Sekunde. Kosten wurden in Form der Gesamtkosten des Betriebs über eine festgelegte Zeitspanne gemessen. Dies beinhaltet Kosten für Anschaffung von Hard- und Software, Anpassung, Betrieb und Wartung. Die so stattfindende Bewertung anhand einzelner Leistungskriterien birgt jedoch das Problem, dass konkrete Anwendungsfälle unterschiedliche Anforderungen mit sich bringen, welche von solchen generisch gehaltenen Benchmarks unzureichend abgedeckt werden. Um diesen Bedarf zu decken wurden *domänenspezifische Benchmarks* entwickelt, welche den Fokus auf bestimmte Anwendungsfelder legen [Gr93].

In dem Bereich Datenbanken und Transaktionsverarbeitung haben viele Datenbankhersteller eigene Benchmarks entworfen, welche gezielt die Stärken ihrer Produkte betonten und keinen objektiven Vergleich bieten konnten. Als Reaktion haben sich 1988 34 Unternehmen

zusammengeschlossen und die *Transaction Processing Performance Council* gegründet. Ziel war es einheitliche domänenspezifische Benchmarks zu entwickeln, welche als verlässlicher Standard fungieren können. Aus den ursprünglich zwei zur Verfügung gestellten Benchmarks haben sich bis zum jetzigen Zeitpunkt elf domänenspezifische Benchmarks entwickelt. Schwerpunkte sind unter anderem Themen wie *Datenintegration*, *Entscheidungsunterstützungssysteme* und *Big Data* [TP18].

3.2.1 Anforderungen an Benchmarks

Nach Gray [Gr93] bestehen für *domänenspezifische Benchmarks* vier zentrale Anforderungen:

- **Relevanz**: Die Leistung des Systems muss unter für das Anwendungsfeld typischen Bedingungen und Belastungen gemessen werden.
- **Portierbarkeit**: Der Benchmark sollte auf unterschiedlichen Systemen und Architekturen implementiert werden können.
- **Skalierbarkeit**: Der Benchmark sollte an die Leistungsfähigkeit des Computersystems angepasst werden können.
- **Einfachheit**: Der Benchmark muss verständlich gehalten sein.

Huppler [Hu09] stellt in seinem Paper „The Art of Building a Good Benchmark“ neben der bereits vorher aufgeführten **Relevanz** die folgenden vier Anforderungen in den Vordergrund:

- **Wiederholbarkeit**: Der Benchmark kann mehrfach durchgeführt werden und führt zu dem gleichen Ergebnis.
- **Fairness**: Die zu vergleichenden Systeme bringen die nötigen Voraussetzungen mit, um gleichberechtigt teilzunehmen (siehe **Portierbarkeit** nach Gray).
- **Verifizierbarkeit**: Es kann mit begründetem Vertrauen davon ausgegangen werden, dass die Ergebnisse korrekt sind.
- **Wirtschaftlichkeit**: Der Benchmark kann durch den entsprechenden Geldgeber finanziert werden.

Ergänzend wird darauf hingewiesen, dass nicht jeder Benchmark alle Aspekte in vollem Umfang berücksichtigen muss. Sie sollen jedoch als Richtlinien gesehen werden, für die kontinuierliche Neu- und Weiterentwicklung von Benchmarks.

3.2.2 Benchmark von Cloudanwendungen

In Kapitel 2 wird das dieser Arbeit zu Grunde liegende Verständnis von *Real-Time Datenbanken* erläutert. Die Menge an Systemen, welche durch diese Definition beschrieben werden, ist wie dargestellt heterogen. Die *Firebase Realtime Database* ist beispielsweise Teil einer cloudbasierten Entwicklungsplattform.

Unter dem Aspekt der vorhergehend genannten Fairness sollen auch diese Systeme in die Betrachtung einbezogen werden. Dadurch ergeben sich bedeutende Implikationen für die Gestaltung eines Benchmarks.

Bermbach, Wittern et al. [BWT17] sehen folgende zentrale Unterschiede:

- Der **Zugriff** auf das zu testende System: Im traditionellen Benchmarking besteht oft eine direkte Kontrolle über die Systeme und genaue Kenntnis über die internen Abläufe und Implementierungsdetails. Bei Anwendungen in der Cloud muss davon ausgegangen werden, dass kaum oder eingeschränkter Zugriff auf die Systeme selbst besteht und die Kommunikation nur über definierte Schnittstellen erfolgt. Das System bleibt aus Sicht des Benchmarks eine *Black Box*.
- Der Fokus auf **Leistung**: Traditionelle Benchmarks beschränken die untersuchten Qualitätsmerkmale oft ausschließlich auf bestimmte Durchsatzmetriken unter starker Belastung. Zurückgreifend auf den vorhergehenden Punkt ist hierzu festzustellen, dass Clouddienste als Reaktion auf Belastung dynamisch skalieren, der Nutzer hiervon jedoch in der Regel nichts mitbekommt. Dies erschwert den Vergleich solcher Metriken. Ein weiterer wichtiger Aspekt in diesem Kontext sind rechtliche Regelungen der Anbieter. *Firebase* beispielsweise verbietet im Rahmen seiner *Acceptable Use Policy* explizit „den vorsätzlichen Versuch ein System zu überlasten“ [Fi18], was viele traditionelle Benchmarks im Kern beabsichtigen oder in Kauf nehmen.

3.3 Aufbau eines Benchmarks

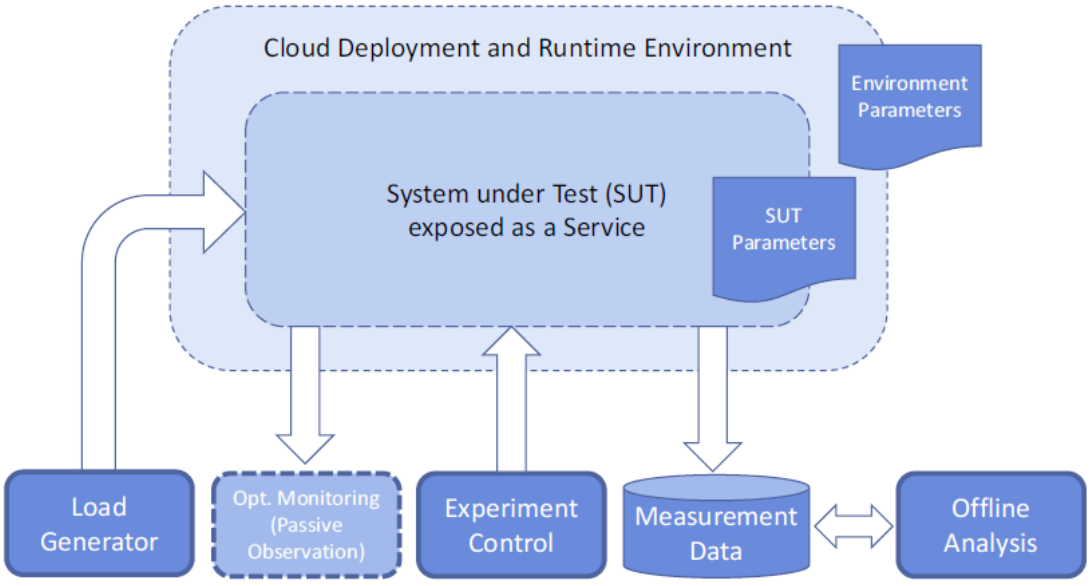

Abbildung 10: Komponenten für Benchmarking von Cloudanwendungen [BWT17, S. 15]

Wie bereits im vorhergehenden Kapitel beschrieben, sind ein Teil der in dieser Arbeit betrachteten Systeme Cloudanwendungen. Wie dargestellt bringt diese Kategorie besondere Einschränkungen in der Konzeption eines Benchmarks mit sich. Aus Gründen der Fairness wird in der weiteren Betrachtung diese Einschränkungen für alle zu testenden Systeme angenommen.

Bermbach, Wittern et al. beschreiben den Aufbau und die Komponenten eines Benchmarks für Cloudanwendungen wie in Abbildung 10 dargestellt [BWT17]:

- **System under Test**: Das zu testende System.
- **Lastgenerator**: Komponente, welche die Belastung für das zu testende System erzeugt.
- **Experimentsteuerung**: Übernimmt die Steuerung der Versuchsdurchläufe und Koordinierung der übrigen Komponenten.
- **Monitoring**: Optional kann ein externes Monitoring-Tool zum Einsatz kommen um weitere Messwerte zu erfassen, ohne die Leistung des Benchmarks zu beeinflussen.
- **Messdaten**: Die definierten Qualitätsmerkmale müssen in Form ihrer zugeordneten Metriken feingranular erfasst werden.
- **Offline Analyse**: Auswertung der Messdaten. Findet nach dem Benchmarkdurchlauf statt, damit die Ressourcen des Benchmark nicht zusätzlich belastet werden. Ansonsten können die Ergebnisse verfälscht werden.

4 Konzeption des Benchmarks

Für die Gestaltung des Benchmarks werden in diesem Kapitel Anforderungen gesammelt. Zunächst wird auf Grundlage der vorhergehenden Kapitel zusammengefasst, welche allgemeinen Anforderungen erfüllt werden müssen. Diese werden ergänzt durch Anforderungen, welche sich aus dem Anwendungsfall bzw. der Motivation, hinter dem Benchmark ergeben. Abschließend wird untersucht, welche Möglichkeiten bestehen, die Qualität einer *Real-Time Datenbank* zu bewerten und anhand von welchen Metriken diese Qualität messbar gemacht wird.

4.1 Anforderungsanalyse

Aus den Grundlagen von *Real-Time Datenbanken* und Benchmarks aus Kapitel 2 und 3 ergeben sich folgende Anforderungen:

- Aus dem Grundsatz der Fairness (siehe 3.2.1) ergibt sich, dass ein Zugriff auf das zu testende System nur über eine Schnittstelle erfolgen darf, welche auch das System mit den höchsten Zugriffsrestriktionen anbietet. Aufgrund der proprietären Cloudanbieter kann der Zugriff somit nur von einem Webclient über eine spezifizierte Webschnittstelle erfolgen.
- Der Lastgenerator muss parametrisierbar sein, um zu gewährleisten, dass der Benchmark skalierbar ist.
- Die Belastung des Systems soll realistische Interaktion simulieren und aus rechtlichen Gründen (*Acceptable Use Policy*) keine mutwillige Überlastung der Systeme hervorrufen (siehe 3.2.2).
- Das System muss während der Testläufe Messwerte protokollieren und nach der Ausführung für eine weitergehende Analyse in geeignetem Format zur Verfügung stellen. Hierdurch kann auch die Verifizierbarkeit sichergestellt werden, da alle Aktionen und ihre Konsequenzen rekonstruiert werden können.
- Bei Auswahl der Kandidaten für die Testläufe ist der Aspekt der Finanzierbarkeit, im Hinblick auf die Kostenstruktur der Cloudanbieter, zu berücksichtigen.

4.2 Motivation

Die Zielsetzung eines Benchmarks beeinflusst entscheidend seine Gestaltung. Wie in Kapitel 3 beschrieben ergibt sich die Motivation für Benchmarking häufig aus einem betriebswirtschaftlichen Interesse. Dies impliziert eine Form von Kosten-/Leistungsvergleich. Abschnitt 3.2.2 erläutert Gründe, warum eine reine Leistungsbewertung von Cloudanwendungen eine Herausforderung darstellen kann. Der nachfolgende Abschnitt 4.3 befasst sich damit, wie die Qualität in Form von Leistung oder auch anderer Merkmale beschrieben werden kann.

Entscheidenden Einfluss auf die konkrete Gestaltung hat vor allem die Perspektive, aus welcher die Systeme bewertet werden sollen. Die unterschiedlichen Zielgruppen umfassen unter anderem Marketingabteilungen, Vertrieb, Zertifizierungsstellen, Wissenschaftler und Software Entwickler. Jede Gruppe folgt unterschiedlichen Interessen und daraus ergeben sich unterschiedliche Anforderungen. In Kapitel 2.1 wurde erläutert, dass *Real-Time Datenbanken* im Kontext der *reaktiven Programmierung* neue Möglichkeiten bieten. Reaktive Systeme können hierdurch vom Front-End bis zu der Datenhaltung nach einheitlichen Prinzipien gestaltet werden. Aus diesem Grund wird in dieser Arbeit die Perspektive der Software Entwicklung eingenommen. Zielsetzung können unter anderem das Erkennen von Flaschenhälsen in der Software und Optimierung des Entwicklungsprozesses sein [Da95].
Andere Motivationen für eine Untersuchung der Systeme können das Erproben von neuen Technologien sein oder die Beobachtung des Verhaltens in verschiedenen Situationen [BWT17, S. 31]. Der Benchmark soll auch eine explorative Auseinandersetzung mit den unterschiedlichen Systemen und ihren Fähigkeiten ermöglichen.

4.3 Identifikation von Qualitätsdimensionen

ISO/IEC 25010 definiert Qualität von Software Systemen folgendermaßen: "... the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value. These stated and implied needs are represented ... by quality models that categorize product quality into characteristics, which in some cases are further subdivided into sub-characteristics." [In11] nach [WVB14, S. 53].
Im Rahmen des *Product Quality Model* werden acht Charakteristiken unterschieden:

- **Funktionale Eignung**: Der Grad, mit dem ein System den gestellten und implizierten Anforderungen gerecht wird. Unterteilt in Funktionale Vollständigkeit, Funktionale Korrektheit und Funktionale Angemessenheit
- **Leistungsfähigkeit**: Leistung in Relation zu den eingesetzten Ressourcen.
- **Kompatibilität**: Der Grad, mit dem ein System mit anderen Systemen interagieren kann, um seine Aufgaben zu lösen.
- **Usability**: Der Grad, mit dem ein System von Nutzern mit angemessener Effizienz und Effektivität bedient werden kann.
- **Zuverlässigkeit**: Der Grad, mit dem ein System seine spezifizierte Leistung über eine festgelegte Zeitspanne erbringt.
- **Sicherheit**: Der Grad, mit dem ein System Informationen und Daten schützt, um ausschließlich autorisierten Zugriff zuzulassen.
- **Wartbarkeit**: Der Grad an Effektivität und Effizienz, mit dem ein System zu Wartungszwecken modifiziert werden kann.
- **Portabilität**: Der Grad an Effektivität und Effizienz, mit dem ein System in eine andere Software- oder Hardwareumgebung überführt werden kann.

Nicht alle dieser Charakteristiken lassen sich mit Hilfe eines Benchmarks messen. Gleichzeitig sollte ein Benchmark nach Möglichkeit nicht auf einen Aspekt reduziert werden,

sondern mehrere Qualitätsdimensionen abbilden können [BWT17]. Viele der Charakteristika setzen für eine Analyse umfassenden Zugriff auf das zu testende System voraus. In diesem Kontext wird der Fokus auf die nachfolgenden Charakteristika und ihre Subcharakteristika gelegt.

4.3.1 Funktionale Eignung

Unter dem Aspekt der Funktionalen Eignung wird nach ISO/IEC 25010 die folgenden drei Subcharakteristika zusammengefasst [In18]:

- **Funktionale Vollständigkeit:** Der Grad, mit dem die von dem System bereitgestellte Funktionalität ermöglicht, dass die spezifizierten Anforderungen erfüllt werden.
- **Funktionale Korrektheit:** Der Grad, mit dem das System korrekte Ergebnisse zur Verfügung stellt.
- **Funktionale Angemessenheit:** Der Grad, mit dem die bereitgestellte Funktionalität die Bewältigung von spezifizierten Aufgaben vereinfacht.

Die funktionalen Anforderungen, um der präsentierten Definition einer *Real-Time Datenbank* zu entsprechen, reduzieren sich im Kern auf die Bereitstellung der geforderten Schnittstelle. Nach diesem Verständnis wird ein System als funktional vollständig betrachtet, wenn diese Schnittstelle existiert. Funktionalität, welche über diese grundlegende Schnittstelle hinausgeht, wird aus diesem Grund in der Kategorie der funktionalen Angemessenheit betrachtet. Da somit alle Systeme, welche Fokus dieser Arbeit sind, als funktional vollständig gelten, wird dieses Charakteristikum nicht weiter betrachtet. Die Aspekte der funktionalen Korrektheit und funktionalen Angemessenheit werden in den Abschnitten 4.4.2 und 4.4.3 für den Kontext dieser Arbeit untersucht, um geeignete Metriken zu identifizieren.

4.3.2 Leistungsfähigkeit

Der Aspekt der Leistungsfähigkeit wird nach ISO/IEC 25010 ebenfalls in drei Subcharakteristika unterteilt [In18]:

- **Zeitverhalten:** Der Grad, in dem Antwort- und Verarbeitungszeit, sowie Durchsatzraten bei der Erbringung der Funktionalität den Anforderungen entsprechen.
- **Ressourcenauslastung**: Der Grad, mit dem Mengen und Arten von Ressourcen entsprechend der Anforderungen genutzt werden.
- **Kapazität**: Der Grad, mit dem die Limitierungen des Systems den Anforderungen entsprechen.

Die Ressourcenauslastung und Kapazität eines Systems lassen sich ohne direkten Zugriff nicht untersuchen. Dies gilt ebenso für die Verarbeitungszeit, da nur die Antwortzeit aus Perspektive des Nutzers gemessen werden kann.

4.4 Identifikation von Qualitätsmetriken

Eine Qualitätsmetrik beschreibt eine Funktion, welche einer messbaren Qualitätsdimension einen Wert zuweist. Sie drückt dadurch die Unterschiede innerhalb der Qualitätsdimension aus [BWT17]. Es kann unterschieden werden, zwischen objektiven und subjektiven Metriken. Objektive Metriken umfassen absolute Messwerte. Subjektive Metriken werden genutzt, wenn keine eindeutige Messung vorgenommen werden kann. Sie werden oft in Relation zu anderen Bewertungen betrachtet [Ba92].

4.4.1 Zeitverhalten

Eine mögliche Metrik um das Zeitverhalten des Systems zu messen, ist die Antwortzeit. Im Kontext traditioneller Datenbank Management Systeme wird dies oft als Latenz gemessen. Für *Real-Time Datenbanken* ist dieser Aspekt von besonderer Bedeutung. Die grundlegende Motivation für die Entwicklung dieser Systeme, besteht in dem effizienten Synchronisieren von Daten zwischen der Datenbank zu dem Nutzer. Die Latenz ist ein kritischer Faktor, denn sie bestimmt, wie lange die Daten eines Nutzers nicht synchron zu den Daten in der Datenbank sind. Für traditionelle Datenbanken beschreibt Latenz die Zeitspanne zwischen dem Zeitpunkt der Abfrage und dem Zeitpunkt, an dem der Nutzer das Ergebnis erhält.
Eine breitere Definition beschreibt Latenz als Zeit, die ein Paket zwischen seinem Ursprung, der es verschickt, zum Eintreffen an seinem Ziel benötigt [Gr13, S. 3]. Dieses Verständnis ist für den Anwendungsfall von *Real-Time Datenbanken* passender. Der Nutzer stellt initial eine Abfrage an das System und erhält ein initiales Ergebnis geliefert. Für diesen ersten Schritt würde die Messung von Latenz zwischen Abfrage und Ergebnis eine sinnvolle Messung ergeben. Für alle weiteren Ergebnisse, welche bei dem Nutzer eintreffen, macht es jedoch keinen Sinn, den Zeitpunkt der Abfrage als Ausgangspunkt zu nehmen. Vielmehr ist entscheidend, zu welchem Zeitpunkt die Aktion stattgefunden hat, durch welche das Element Teil der Ergebnismenge wurde. Abbildung 11 verdeutlicht das Problem. Die relevante Zeitspanne für das Element *x* liegt nicht zwischen dem Zeitpunkt *t1* und *t4*, sondern zwischen dem Zeitpunkt *t3* und *t4*.

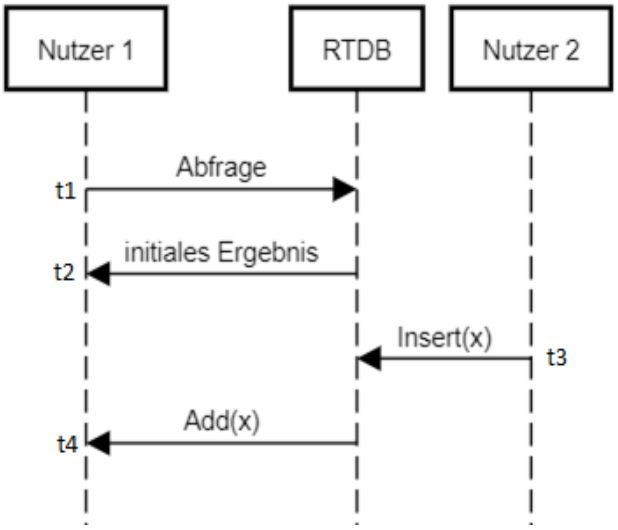

Abbildung 11: Latenzmessung für Real-Time Datenbanken (Quelle: *eigene*)

Die Annahme, dass die korrekte Zeitspanne in jedem Fall zwischen der Einfügeoperation des Elements von *Nutzer 2* und dem Empfang des Elements von *Nutzer 1* liegt, ist jedoch unzutreffend.

Dies tritt nur ein, wenn die Abfrage von *Nutzer 1* alle Elemente erhalten möchte, die seine Bedingungen erfüllen. Wenn ein Element zu dem Zeitpunkt des Einfügens die Bedingung erfüllt, wird es an *Nutzer 1* versendet.

Abbildung 12 zeigt einen Fall, in dem diese Annahme nicht zutrifft. *Nutzer 1* stellt eine sortierte Abfrage und begrenzt das Ergebnis auf eine bestimmte Anzahl an Elementen. *Nutzer 2* fügt das Element x in die *Real-Time Datenbank* ein. Es erfüllt die Bedingung von der Abfrage von *Nutzer 1*, wird jedoch außerhalb der Begrenzung einsortiert. *Nutzer 2* verändert daraufhin das Element *y*, welches vorher Teil des initialen Ergebnisses für Abfrage war. Durch die Veränderung erfüllt Element y die Bedingung nicht mehr und *Nutzer 1* wird benachrichtigt, dass es aus dem Ergebnis entfernt wird. Daraufhin wird ein neues Element Teil des begrenzten Ergebnisses, in diesem Fall Element *x*. Der relevante Zeitraum liegt in diesem Fall jedoch nicht zwischen dem Einfügen von Element *x* (*t3*) und dem Empfang bei *Nutzer 1* (*t6*), sondern beginnt mit der Aktion, welche dazu geführt hat, dass *x* Teil des Ergebnisses wird. Diese Aktion ist die Veränderung von Element *y* zum Zeitpunkt *t4*.

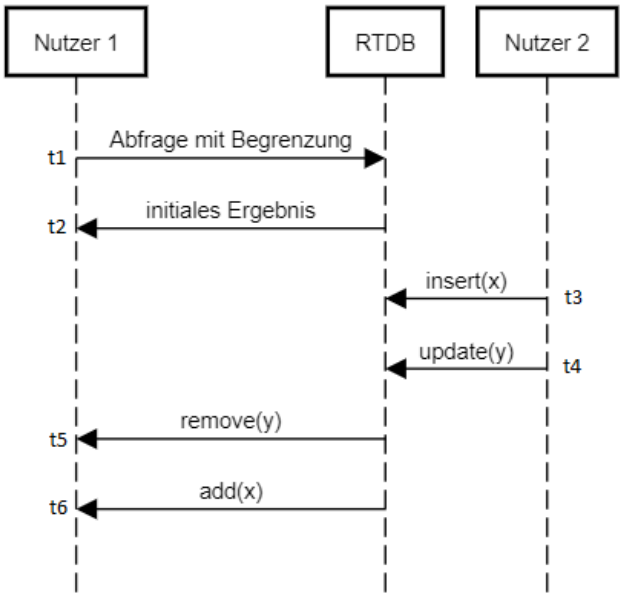

Abbildung 12: Latenzmessung für Abfragen mit Begrenzung (Quelle: *eigene*)

Als Resultat bezieht sich Latenz für diesen Anwendungsfall auf den Zeitraum zwischen dem Zeitpunkt der Aktion, welche ein Element zum Teil der Ergebnismenge werden lässt, und dem Zeitpunkt des Empfangs bei dem Nutzer. Dieses Verständnis wird unter dem Begriff *Aktion-zu-Empfang-Latenz* in der weiteren Arbeit verwendet. Die *Aktion-zu-Empfang-Latenz* soll als Metrik für das Zeitverhalten verwendet werden, gemessen in Form von Millisekunden.

In einem realen Szenario muss davon ausgegangen werden, dass sich *Nutzer 1* und *Nutzer 2* aus dem vorhergehenden Beispiel nicht auf demselben System arbeiten, sondern auf unabhängigen Computern. Uhren auf unterschiedlichen Systemen sind aufgrund von Taktversatz nie gleich, was für Verteilte Systeme ein großes Problem darstellt. Die Synchronisation der Systemuhren ist ein sehr komplexes Aufgabenfeld [La78].

Um diese Problematik für den Benchmark zu vermeiden, wird davon ausgegangen, dass sowohl das Einfügen, als auch der Empfang der Daten auf demselben System stattfinden.

4.4.2 Funktionale Korrektheit

In dem gegebenen Kontext arbeiten die Systeme dann korrekt, wenn jeder Nutzer alle Elemente erhält, die seiner Abfrage entsprechen. Auf Grundlage der Messdaten muss es möglich sein, zu ermitteln, welche Benachrichtigungen der Nutzer erhalten sollte und welche tatsächlich eintreffen. Ebenso muss überprüft werden, ob die korrekten Informationen in der Benachrichtigung übermittelt wurden. Bei nicht korrektem Verhalten wird eine Diskrepanz zwischen diesen beiden Mengen auftreten.

4.4.3 Funktionale Angemessenheit

In Abschnitt 4.2 wird festgelegt, dass der Benchmark aus Perspektive der Software Entwicklung gestaltet werden soll. Wie bereits bezüglich der Funktionalen Vollständigkeit festgestellt wurde, bieten alle Systeme die geforderte Funktionalität in Form der definierten Schnittstelle. Die Funktionale Angemessenheit soll bewerten, in welchem Grad die Funktionalität für den Nutzer vereinfacht zur Verfügung gestellt wird. Diese Bewertung ist nur in Form einer subjektiven Metrik möglich.
Einfach wird in diesem Kontext wie folgt verstanden: je weniger eigener Entwicklungsaufwand für die Lösung einer gegebenen Problemstellung notwendig ist, desto einfacher ist die Handhabung.
Im Kontext von Datenbanken kann eine einfache Handhabung nach diesem Verständnis mit der Ausdruckskraft der Abfragesprache assoziiert werden. Wichtig für die Bewertung der Ausdruckskraft der Abfragesprache ist die Zielsetzung. *SQL* bietet eine deutlich höhere Ausdruckskraft als die meisten Abfragesprachen von *NO-SQL-Datenbanken*. Für den Einsatz in sehr datenintensiven und verteilten Anwendungen eignen sich diese jedoch besser [Wi15]. Wie bereits in Abschnitt 2.4 festgestellt, sind die unterschiedlichen Vertreter in der Tatsache geeint, dass für die Datenhaltung eine Form von dokumentenorientierter Datenbank zum Einsatz kommt. Aus diesem Grund wird die Abfragesprache von *dokumentenorientierten Datenbanken* für diesen Kontext als sinnvolle Referenz für funktionale Angemessenheit angesehen. Die laut *DB-Engines*[10] populärste dokumentenorientierte Datenbank ist *MongoDB*. Sie bietet dem Nutzer über die Schnittstelle verschiedene Möglichkeiten, um diverse Problemstellungen einfacher zu bewältigen. Betrachtet werden in diesem Rahmen die von MongoDB zur Verfügung gestellten Abfrageoperatoren, sowie Funktionen, um Abfragen weitergehend zu vereinfachen.

Abfrageoperatoren

MongoDB bietet Operatoren für die Abfragegestaltung unter anderem in den folgenden Kategorien [Mo18a]:

[10] https://db-engines.com/de/ranking (Stand 26.10.2018)

- **Vergleichende Abfrageoperatoren**: Operatoren, welche die Möglichkeit geben, eine Bedingung im Vergleich zu einem spezifizierten Wert zu formulieren. Beispielsweise kann hiermit geprüft werden, ob Elemente gleich (**$eq**) oder ungleich (**$ne**) einem bestimmten Wert sind, oder ob sie größer (**$gt**) oder kleiner (**$lt**) sind.
- **Logische Abfrageoperatoren:** Operatoren, welche eine Bewertung anhand von Aussagenlogik ermöglichen, beispielsweise indem zwei Bedingungen in Form von Konjunktion (**$and**) oder Disjunktion (**$or**) logisch verknüpfen können.
- **Elementbezogene Abfrageoperatoren:** Operatoren, welche Eigenschaften der betrachteten Elemente bewerten können, wie beispielsweise das Vorhandensein eines bestimmten Feldes (**$exists**).
- **Evaluationsbezogene Abfrageoperatoren:** Operatoren, welche eine Evaluation anhand von komplexen Bedingungen ermöglichen. Beispielsweise in Form eines Vergleichs mit vorgegebenem JSON-Schema (**$jsonSchema**), mit regulären Ausdrucken (**$regex**) oder in Form einer Textsuche (**$text**).
- **Raumbezogene Abfrageoperatoren:** Operatoren, welche eine Bewertung von geometrischen Daten ermöglichen, beispielsweise die Nähe zu einem bestimmten Punkt (**$near**) oder die Lage in einem bestimmten Bereich (**$geoWithin**).
- **Array Abfrageoperatoren**: Operatoren, welche auf Listen von Daten angewandt werden können. Beispielsweise kann ermittelt werden ob eine Menge an spezifizierten Daten Teil der betrachteten Liste ist (**$all**) oder ob die Größe der Liste einem bestimmten Wert entspricht (**$size**).

Zusätzliche Abfragefunktionalität

Neben den Abfrageoperatoren bietet MongoDB eine Reihe von zusätzlicher Funktionen, um Abfrageergebnisse zu modifizieren, unter anderem umfassen diese:

- **Sortierung**: Mit Hilfe der **sort**-Methode kann das Ergebnis einer Abfrage anhand von spezifizierten Feldern aufsteigend oder absteigend sortiert werden [Mo18b].
- **Begrenzung**: Die Funktion **limit** bietet die Möglichkeit, das Ergebnis einer Abfrage auf eine spezifizierte Anzahl zu reduzieren [Mo18d].
- **Versatz:** Die **skip**-Methode ermöglicht das Überspringen von einer spezifizierten Anzahl an Elementen des Ergebnisses [Mo18c].

Die dargestellten Funktionen von MongoDB stellen lediglich einen Auszug des breiten Spektrums an unterstützter Funktionalität dar. In dieser Arbeit wird nur ein Teil der verfügbaren Funktionalität im Detail betrachtet. Dies liegt zum einen daran, dass viele der betrachteten Systeme nur eine kleine Teilmenge der Funktionalität von MongoDB unterstützen. Es wäre wenig sinnvoll Szenarien zu konzipieren, welche kaum ein Anbieter unterstützt. Zum anderen sollen die einzelnen Funktionen nicht isoliert betrachtet werden, sondern auch in Kombination mit anderen. Dadurch ergeben sich bereits aus einer geringen Menge an betrachteten Operatoren eine große Anzahl möglicher Testszenarien. Die

gewählte Referenzmenge an Operatoren wird in dem folgenden Abschnitt bestimmt. Als Metrik gilt der Grad mit der die zu testenden Systeme gleichwertige Funktionalität bieten.

4.5 Analyse der funktionalen Angemessenheit

Wie zuvor beschrieben muss eine Menge an Operatoren von *MongoDB* als Referenz bestimmt werden. Für diese Arbeit wird der Fokus auf die Vergleichsoperatoren und die Funktionen zu Sortierung, Begrenzung und Versatz gelegt. Zunächst werden diese im Detail für *MongoDB* beschrieben. Darauffolgend werden die einzelnen Systeme hinsichtlich ihrer Unterstützung analysiert. Zum Abschluss wird ein Vergleich zwischen den Systemen und der Referenzmenge vorgenommen.

4.5.1 MongoDB

Die Funktionen für Begrenzung und Versatz wurden bereits im vorhergehenden Abschnitt beschrieben. Die grundlegende Funktionalität der Sortierung wurde ebenfalls beschrieben. Es wird sowohl aufsteigende, als auch absteigende Sortierreihenfolge unterstützt. Ein wichtiger Aspekt ist zudem, dass *MongoDB* auf eine Abfrage eine sortierte Ergebnismenge liefert. Im Fall von *Real-Time Datenbanken* wird diese Ergebnismenge fortgehend erweitert um einzelne Elemente. Um als funktional gleichwertig bewertet zu werden, muss das System geeignete Informationen liefern, welche dem Nutzer die korrekte Einordnung in die vorhandene Ergebnismenge ermöglichen.

Vergleichsoperatoren

MongoDB bietet dem Nutzer die folgenden Vergleichsoperatoren [Mo18a]:

- **$eq:** Bewertet, ob ein Wert dem spezifizierten Parameter entspricht.
- **$gt:** Bewertet, ob ein Wert größer als ein spezifizierter Parameter ist.
- **$gte:** Bewertet, ob ein Wert größer oder gleich einem spezifizierten Parameter ist.
- **$in:** Bewertet, ob ein Wert einem der Werte aus einer spezifizierten Liste entspricht.
- **$lt:** Bewertet, ob ein Wert kleiner als ein spezifizierter Parameter ist.
- **$lte:** Bewertet, ob ein Wert kleiner oder gleich einem spezifizierten Parameter ist.
- **$ne:** Bewertet, ob ein Wert dem spezifizierten Parameter nicht entspricht.
- **$nin:** Bewertet, ob ein Wert keinem der Werte in einer spezifizierten Liste entspricht.

4.5.2 Firebase

Die *Firebase Realtime Database* bietet die nachfolgenden Operatoren und Funktionen. Im Gegensatz zu den anderen hier beschriebenen Systemen, basiert Firebase auf einem simpleren Datenmodell. Jede Datenbank wird durch ein einzelnes Dokument repräsentiert und nicht als Sammlung von Dokumenten, wie in den anderen Fällen. Die Funktionen und Operatoren basieren auf der Navigation in der Baumstruktur des Dokuments [Go18a].

Sortierung

Firebase liefert drei verschiedene Methoden um Daten zu sortieren [Go18c]:

- **OrderByChild:** Die Ergebnisse werden anhand des Wertes eines festgelegten Kindknotens sortiert. Es können auch tiefer liegende Kindknoten spezifiziert werden.
- **OrderByKey:** Die Ergebnisse werden anhand der Schlüssel der Kindknoten sortiert.
- **OrderByValue:** Die Ergebnisse werden anhand der Werte der Kindknoten sortiert.

Pro Abfrage kann ausschließlich eine Sortierungsmethode verwendet werden. Eine Sortierungsreihenfolge kann nicht spezifiziert werden, standardmäßig wird aufsteigend sortiert. In den Benachrichtigungen wird die Sortierung der einzelnen Elemente über den Schlüssel des vorhergehenden Elements angegeben.

Begrenzung

Für die Begrenzung der Ergebnismenge werden zwei Methoden geboten [Go18c]:

- **LimitToLast:** Begrenzt die Ergebnismenge auf die letzten x Ergebnisse.
- **LimitToFirst:** Begrenzt die Ergebnismenge auf die ersten x Ergebnisse.

Die Begrenzungsmethoden können ausschließlich auf sortierte Abfragen angewendet werden.

Versatz

Nach dem klassischen Verständnis einer Versatzmethode, bietet *Firebase* keine solche Funktionalität. Es ist nicht universell möglich beispielsweise die ersten zehn Ergebnisse zu überspringen.

Vergleichsoperatoren

Um die Ergebnismenge nach bestimmten Kriterien zu filtern, werden drei Methoden zur Verfügung gestellt [Go18c]:

- **equalTo**: Überprüft den durch die Sortierung spezifizierten Wert oder Schlüssel auf Gleichheit zu dem gegebenen Parameter.
- **startAt**: Überprüft, ob der durch die Sortierung spezifizierte Wert oder Schlüssel größer oder gleich dem gegebenen Parameter ist.
- **endAt**: Überprüft, ob der durch die Sortierung spezifizierte Wert oder Schlüssel kleiner oder gleich dem gegebenen Parameter ist.

Alle Vergleichsoperatoren können nur auf sortierte Abfragen angewendet werden und sind durch die Einschränkungen der Sortierungsmethoden ausschließlich auf einen Wert oder Schlüssel anwendbar. Mit Hilfe der Datenmodellierung können einige Werte als Schlüssel von Elternelementen ausgelagert werden. Dadurch kann theoretisch die Funktionalität von **equalTo** auf mehrere Werte ausgedehnt werden, da diese Schlüsselwerte über

den Zugriffspfad vorausgesetzt werden[11]. Dies führt jedoch zu tieferer Verschachtelung, welche aus Leistungsgründen nicht empfohlen wird [Go18a].

4.5.3 Meteor

Wie in Abschnitt 2.4 beschrieben, verwendet *Meteor* für die Datenhaltung *MongoDB*. Die von *Meteor* bereitgestellte Schnittstelle ist kompatibel mit der Schnittstelle von *MongoDB*. Aus diesem Grund sind Funktionen und Operatoren identisch. Mit Verweis auf Abschnitt 4.5.1 werden diese hier nicht erneut erläutert. Wenn Operatoren außerhalb der festgelegten Referenzmenge betrachtet werden, bietet *Meteor* als einziges System vollständige, funktionale Gleichwertigkeit zu *MongoDB* [Me18a].

Sortierung

Meteor bietet dem Nutzer standardmäßig eine Abstraktion von einzelnen Benachrichtigungen an. Wenn der Nutzer eine Abfrage registriert, werden Benachrichtigungen so lange gesammelt, bis eine lokale *Minimongo* Sammlung anlegt wird. Von da an wird diese automatisch mit Hilfe der Benachrichtigungen synchronisiert. Dies beinhaltet auch die Sortierung. Die einzelnen Benachrichtigungen enthalten keine zusätzlichen Informationen über die Position eines Elements [Me18b]. Das genutzte Protokoll *DDP* bietet für diesen Anwendungsfall mit den Benachrichtigungstypen *addedBefore* und *movedBefore* Möglichkeiten, diese kommen bei *Meteor* jedoch nicht zum Einsatz [SWO16].

4.5.4 RethinkDB

RethinkDB bündelt die Real-Time-Funktionalität unter dem Begriff *Changefeeds*. Über die Funktion *changes* kann ein Großteil der Funktionen und Operatoren aus dem Kontext der Abfragesprache *ReQL* verwendet werden. Dies umfasst zusätzlich zu den hier betrachteten Vergleichsoperatoren unter anderem logische Abfrageoperatoren, evaluationsbezogene Abfrageoperatoren und raumbezogene Abfrageoperatoren [Re18d][Re18a][da16].

Sortierung

Die Methode **order_by** kann für Changefeeds genutzt werden. Sowohl aufsteigende, als auch absteigende Sortierreihenfolge wird unterstützt. Die Funktionalität kann jedoch nur auf den Primär- oder Sekundärindex angewendet werden. Zudem muss zwingend zusätzlich **limit** verwendet werden [Re18d]. Damit der Nutzer den Index des Ergebnisses erhält, muss bei der Abfrage der optionale Parameter *include_offsets* mit *true* angegeben werden [Re18b].

[11] Beispielsweise könnte die Farbeigenschaft eines Elementes als Schlüssel des Elternelementes ausgelagert werden und ein Zugriff über den Pfad („*Elemente/rot/*“) erfolgen. Alle roten Elemente müssten dazu Kindknoten des Elementes *rot* sein.

Begrenzung

Zu der Begrenzung der Ergebnismenge wird die **limit** Methode verwendet. Die Verwendung von **limit** kann nur in Kombination mit der **oder_by** Methode erfolgen [Re18d].

Versatz

Eine Versatzfunktion wird von *RethinkDB* in Kombination mit *Changefeeds* nicht angeboten.

Vergleichsoperatoren

Im Rahmen der **filter**-Methode können eine Reihe von Operatoren für den Vergleich zum Einsatz kommen: **eq** (**$eq**), **ne** (**$ne**), **gt** (**$gt**), **ge** (**$gte**), **lt** (**$lt**) und **le** (**$lte**) sind als funktional gleichwertig zu ihrem jeweiligen Äquivalent in MongoDB zu betrachten. Die Funktionalität von **$in**, bzw. **$nin** kann mit Hilfe der Operatoren **contains**, bzw. **not** und **contains** umgesetzt werden [Re18d][Re18a].

4.5.5 Parse

Die Real-Time-Funktionalität wird bei *Parse* als *LiveQuery* bezeichnet. Neben den Vergleichsoperatoren werden für *LiveQuery* eine begrenzte Anzahl weiterer Abfrageoperatoren unterstützt, unter anderem aus dem Bereich der evaluationsbezogenen Operatoren, raumbezogenen Operatoren und elementbezogenen Operatoren. Die Funktionen für Begrenzung (**limit**) und Versatz (**skip**) sind funktional gleichwertig zu ihren Äquivalenten von *MongoDB* [Pa18a][Wa16].

Sortierung

Sortierung wird in aufsteigender, bzw. absteigender Reihenfolge in Form der Methoden **ascending** und **descending**. Zudem ist es in Form von **addAscending**, bzw. **addDescending** möglich anhand mehrere Werte zu sortieren. Es werden im Rahmen der Benachrichtigungen für neue oder veränderte Ergebnisse jedoch keine Informationen für die Einordnung in die sortierte Ergebnismenge ausgewiesen [Pa18a][Wa16].

Vergleichsoperatoren

Im Rahmen des *Live Query Protocols* bietet *Parse* Unterstützung für alle von *MongoDB* angebotenen Vergleichsoperatoren: **$lt**, **$lte**, **$gt**, **$gte**, **$eq**, **$ne**, **$in** und **$nin** [Wa16].

4.5.6 Baqend

Baqend stellt Real-Time-Funktionalität in Form der *Real-Time Queries* zur Verfügung. Grundsätzlich bestehen zwei Möglichkeiten, um diese zu nutzen [Ba18a]:

- **ResultStream:** Liefert initial die gesamte Ergebnismenge und bei jeder Veränderung die modifizierte gesamte Ergebnismenge

- **EventStream:** Liefert die initiale Ergebnismenge und in feingranularen Benachrichtigungen einzelne Veränderungen an der Ergebnismenge.

Sortierung, Begrenzung und Versatz

Sortierung wird in Form der Methoden **ascending** (aufsteigende Reihenfolge) und **descending** (absteigende Reihenfolge) unterstützt. Für Begrenzung und Versatz werden mit **limit offset** Funktionen zur Verfügung gestellt, welche funktional gleichwertig zu den Äquivalenten von *MongoDB* sind [Ba18a]. Eine Limitierung stellt der Maximalwert für Begrenzung und Versatz von 500 Elementen dar. Diese wird jedoch im Folgenden als technische Einschränkung und nicht als funktionale Einschränkung betrachtet. Aus diesem Grund fällt sie bei der Bewertung im folgenden Abschnitt nicht ins Gewicht.

Vergleichsoperatoren

Baqend bietet mit den folgenden Operatoren funktional gleichwertige Äquivalente zu *MongoDB*: **equal** (**$eq**), **notEqual** (**$ne**), **greaterThan** (**$gt**), **greaterThanOrEqualTo** (**$gte**), **lessThan** (**$lt**), **lessThanOrEqualTo** (**$lte**), **in** (**$in**) und **notIn** (**$nin**) [Ba18d].

4.5.7 Vergleich der Systeme

Auf Grundlage der beschriebenen Schnittstellen wird in Tabelle 1 und

Tabelle 2 gezeigt, wie viel Unterstützung die Systeme für die Operatoren und weiteren Funktionen von *MongoDB* bieten. Die farbliche Kennzeichnung visualisiert hierbei **keine Unterstützung** (rot), **eingeschränkte Unterstützung** (gelb) und **funktional gleichwertige Unterstützung** (grün). Die einzelnen vorhergehenden Abschnitte beschreiben eventuelle Einschränkungen im Detail. Die Ergebnisse des detaillierten Vergleichs zeigen, dass alle Systeme zumindest mit Einschränkungen Teilen Unterstützung für die gewählte Referenzmenge bieten. Das folgende Kapitel stellt das Szenario für die Benachmarking Anwendung vor. Im Rahmen des Szenarios werden konkrete Abfragen definiert, welche die Operatoren und Funktionen in realitätsnahem Kontext einsetzen und kombinieren.

Tabelle 1: Übersicht der Unterstützung für Vergleichsoperatoren

	$eq	$ne	$gt	$gte	$lt	$lte	$in	$nin
Firebase								
Meteor								
RethinkDB								
Parse								
Baqend								

Tabelle 2: Übersicht der Unterstützung für sort, limit und skip

	sort	limit	skip
Firebase			
Meteor			
RethinkDB			
Parse			
Baqend			

5 Konzeption des Szenarios

Dieses Kapitel beschreibt den Kontext, in welchem die Benchmarkapplikation konzipiert wird. Aus dem Szenario ergeben sich die grundlegende Datenstruktur, die möglichen Belastungsszenarien und die Arten von Abfragen, welche zum Test der funktionalen Angemessenheit eingesetzt werden. Das Szenario legt den gemeinsamen Rahmen fest, in dem die identifizierten Qualitätsmerkmale beobachtet werden. Dadurch können die Merkmale nicht nur in Isolation betrachtet werden, sondern auch relativ zu den Ausprägungen der anderen Qualitätsmerkmale.

5.1 Szenariobeschreibung

Auf Grundlage der erhobenen Anforderungen und mit Betrachtung der spezifizierten Zielgruppe soll dieser Benchmark eine hohe Applikationsnähe aufweisen. Der Begriff *Application Benchmark* beschreibt den Ansatz, Benchmarkszenarien zu gestalten, welche einen Großteil der Komponenten realer Applikation repräsentieren und beanspruchen können [BMX16]. Im Kontrast dazu legen *Microbenchmarks* den Fokus auf einzelne, isolierte Aspekte im Hinblick auf spezifische Metriken [BMX09]. Da der Benchmark eine explorative Auseinandersetzung mit den einzelnen Technologien unterstützen soll, ist starker Anwendungsbezug sinnvoll. Um eine reale Applikation möglichst realitätsnah zu simulieren, wurde als Szenario eine datenintensiven Applikation, welche Möglichkeiten für Skalierung und Interaktion bietet, entworfen.

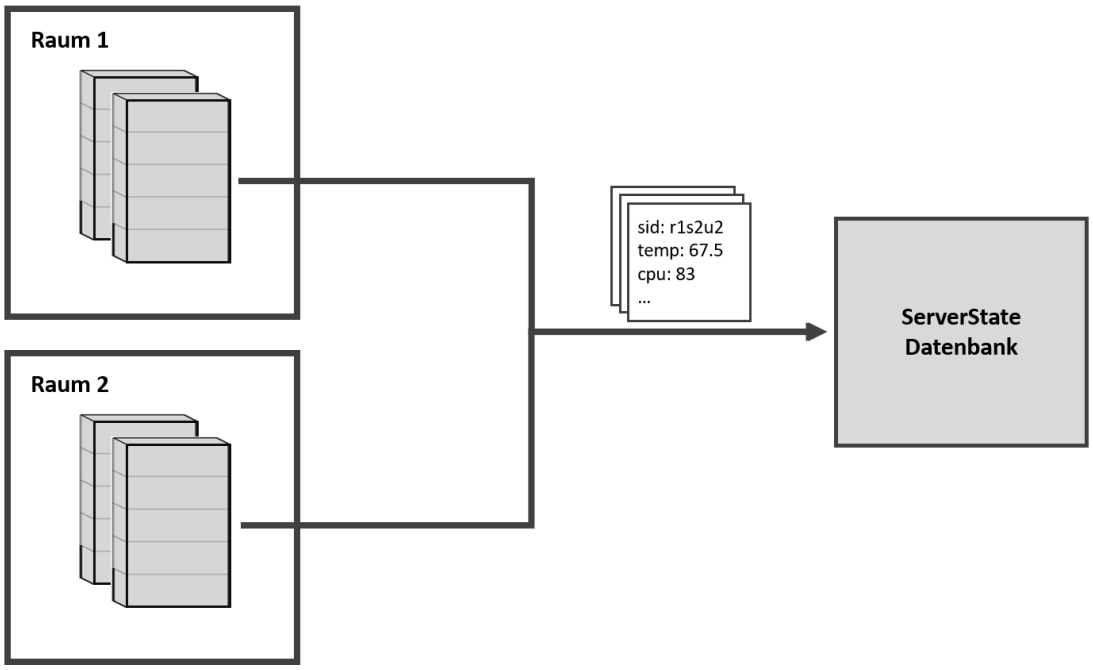

Abbildung 13: Szenario der Datengenerierung (Quelle: *eigene*)

In Abbildung 13 wird verdeutlicht, auf Grundlage welcher Daten der Benchmark konzipiert werden soll. Im Kern beschreibt das Szenario die Echtzeitanalyse und Überwachung von Zustandsmeldungen eines Rechenzentrums, bzw. der Serverräume eines Unternehmens. In dem Modell werden eine Anzahl an Serverschränken, aufgeteilt auf mehrere Räume simuliert. In jedem Serverschrank sind eine spezifizierte Anzahl an Servern. Diese liefern periodisch Meldungen über ihren aktuellen Zustand an eine zentrale Datenbank. Diese Werte können beispielsweise CPU-Auslastung und -Temperatur beinhalten. Auf Grundlage dieser Daten soll eine Monitoring Applikation bestimmte Analysen auf den Echtzeitdaten durchführen. Die Rolle der Datenbankkomponente wird von der jeweiligen *Real-Time Datenbank* eingenommen werden. Die eigentliche Testapplikation ist ein webbasiertes Monitoring Tool, welches unterschiedliche Ansichten auf den Zustand der Server bietet. Über diese Ansichten können unterschiedliche Abfragen und Operatoren in einem realistischen Kontext umgesetzt werden.
Die Anzahl an Räumen, Serverschränken und einzelnen Servereinheiten bestimmt die Last, welche auf die Datenbank ausgeübt wird. Für die Testzwecke wird davon ausgegangen, dass in zwei Serverräumen jeweils vier Serverschränke stehen, welche mit jeweils fünf Servereinheiten bestückt sind. Unter der Annahme, dass jeder Server seinen Zustand einmal pro Sekunde mitteilt, würde dies beispielsweise eine Belastung von 40 Schreiboperationen pro Sekunde für die Datenbank bedeuten. Die Anzahl an Instanzen der Webapplikation steuert die Anzahl der Registrierungen bei der Datenbank.

5.2 Datenmodell

Für die Zustandsdaten der Server sind zwei Datensammlungen vorgesehen. Zum einen soll der aktuelle Zustand des Servers in einer Sammlung *ServerState* festgehalten werden. Diese Sammlung enthält für jeden Server nur die aktuellen Werte. Zum anderen soll jede Zustandsmessung parallel in einer Sammlung *ServerData* festgehalten werden, welche die Daten zu allen vergangenen Zuständen der Server beinhaltet. Eine einzelne Zustandsmessung soll hierbei mindestens die folgenden Felder beinhalten:

- **mid:** Eindeutiger Kennzeichner der aktuellen Messung
- **sid:** Eindeutiger Kennzeichner des Servers. Zusammengesetzt aus Raumnummer (**serverroom**), Schranknummer (**rack**) und Einheitsnummer (**unit**).
- **serverroom:** Nummer des Raumes, in dem sich der Server befindet.
- **rack:** Nummer des Serverschrankes, in dem sich der Server befindet.
- **unit:** Nummer der Servereinheit im Serverschrank.
- **temp:** Messwert für die Prozessortemperatur
- **cpu:** Messwert der prozentualen Prozessorauslastung
- **ts:** Zeitstempel der Messung

5.3 Auswahl der Abfragen

Das Ziel der Auswahl der unterschiedlichen Abfragen besteht in der Abbildung von möglichst vielen realitätsnahen Anwendungsfällen. Zudem steht der Gebrauch und die Kombination von möglichst vielen Operatoren und Funktionen aus der gewählten Referenzmenge im Vordergrund der Betrachtung. Die Abfragen generieren die Datengrundlage für vier verschiedene Ansichten. Jede Ansicht repräsentiert eine grundlegende Abfrage. Mit Hilfe von unterschiedlichen Operatoren und Funktionen kann die Abfrage ergänzt werden. Für ein allgemeines Verständnis werden die Abfragen in *SQL* formuliert.

5.3.1 Übersicht aller Server

Für die erste Ansicht sollen alle aktuellen Serverdaten in einer Übersicht gemeinsam dargestellt werden. Die Standardansicht stellt also eine Abfrage aller Daten aus der *ServerState* Datenbank dar. Verfeinert werden kann diese Abfrage mit Hilfe von Bereichsfiltern auf einzelnen Eigenschaften. Eine weitere Stufe der Verfeinerung stellt eine Kombination aus mehreren Bereichsfiltern dar. Hieraus ergeben sich die ersten drei Abfragetypen:

```
A1: SELECT * FROM ServerState
A2: SELECT * FROM ServerState WHERE CPU >= a AND CPU <= b
A3: SELECT * FROM ServerState WHERE CPU >= a AND CPU <= b
    AND TEMP >= c AND TEMP <= d
```

A2 stellt also beispielsweise eine Abfrage mit Bereichsfilter auf dem Attribut *cpu* dar, wobei a und b für die untere bzw. obere Grenze des Bereichs stehen. **A3** ergänzt diese Abfrage um einen weiteren Bereichsfilter auf dem Attribut *temp*. Verwendete Operatoren sind **$gte** und **$lte**, angewandt auf einem bzw. mehreren Attributen in Kombination.

5.3.2 Übersicht der heißesten Server

Diese Ansicht soll eine Übersicht über eine definierte Anzahl an aktuell heißtesten Servern liefern. In der Standardansicht werden die x Server mit den aktuell höchsten *temp* Werten dargestellt. Mit Hilfe einer Navigation sollen die nächsten x Server angezeigt werden können. Zudem soll ebenfalls der Einsatz eines Bereichsfilters ermöglicht werden. Die begrenzenden Werte sind in diesem Fall nicht inbegriffen. Daraus ergeben sich die folgenden Abfragetypen

```
A4: SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT x
A5: SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT x OFFSET x
A6: SELECT * FROM ServerState WHERE CPU > a AND CPU < b
    ORDER BY TEMP DESC LIMIT x OFFSET x
```

A4 ist eine Abfrage mit absteigender Sortierung und Begrenzung, **A5** verwendet zusätzlich einen Versatz und **A6** ergänzt dies um einen Bereichsfilter auf dem Attribut *temp*. Verwendet werden die Operatoren **$gt** und **$lt**, sowie die Funktionen **sort**, **skip** und **limit**.

5.3.3 Übersicht eines Serverraumes

Die Übersichtsansicht für Serverräume soll den aktuellen Zustand aller Server in dem jeweiligen Raum anzeigen. Dies beinhaltet nur eine Abfrage:

A7: `SELECT * FROM ServerState WHERE serverroom = r`

r steht für die Nummer des jeweiligen Raumes. Durch diese Abfrage wird der Operator **$eq** abgedeckt.

5.3.4 Detailansicht eines Servers

Diese Detailansicht soll den Verlauf der letzten x Messwerte eines Servers darstellen. Hierfür muss auf die Daten aus der Sammlung *ServerData* zurückgegriffen werden. Zusätzlich soll es möglich sein, über eine Navigation um x Werte weiter zu älteren Messwerten zu springen. Dies beinhaltet zwei verschiedene Abfragetypen:

A8: `SELECT * FROM ServerData WHERE sid = s ORDER BY TS DESC LIMIT x`

A9:
```
SELECT * FROM ServerData WHERE sid = s ORDER BY TS DESC LIMIT x
    OFFSET x
```

Bei den Abfragen steht die Variable s für die jeweilige Server-ID und x für die Anzahl, auf welche die Ergebnismenge begrenzt werden soll. Verwendet wird in beiden Abfragen der **$eq** Operator, sowie die Funktionen **sort** und **limit**. Im Fall von **A9** wird zusätzlich **skip** abgedeckt.

6 Komponenten der Anwendung

Den Kern der Benchmarkanwendung bilden wie in Abbildung 14 dargestellt die folgenden Komponenten:

- **Lastgenerator:** Wird umgesetzt durch die Klasse **Producer.**
- **Datenbankschnittstelle:** Wird umgesetzt durch die Klasse **DbInterface.**
- **Ansichten:** Die vier zuvor identifizierten Ansichten werden durch die Klassen **OverviewListClient** (Liste der heißesten Server), **OverviewClient** (Übersicht aller Server), **RoomClient** (Übersicht eines Serverraumes) und **ServerClient** (Detailansicht eines Servers) realisiert.

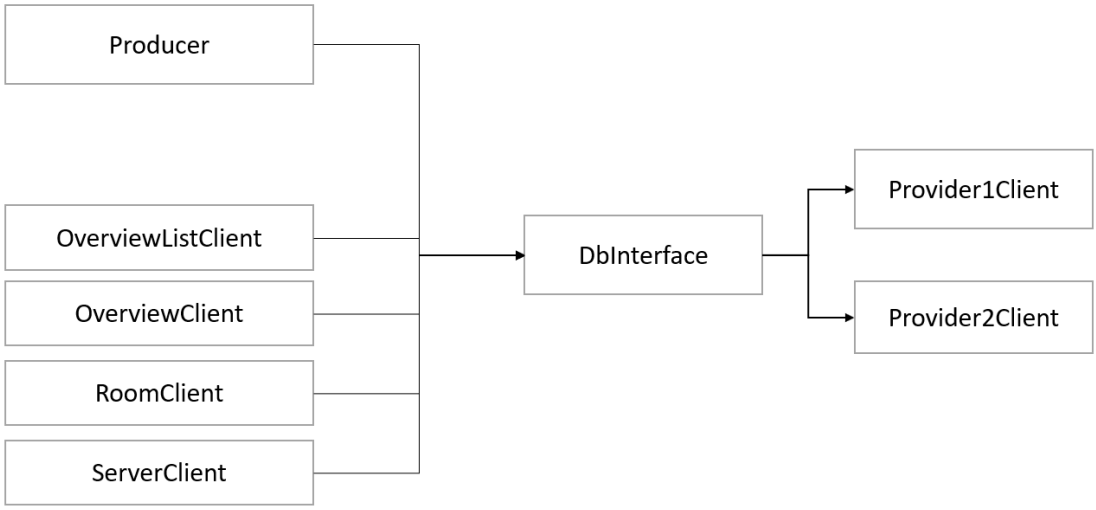

Abbildung 14: Übersicht der wichtigsten Klassen (Quelle: *eigene*)

Zusätzlich ergänzen Komponenten zu **Steuerung und Messung** des Benchmarks die Applikation. In den folgenden Abschnitten werden die einzelnen Komponenten näher beschrieben.

6.1 Lastgenerator

Der Lastgenerator wird durch die Klasse **Producer** implementiert. Zunächst erstellt diese für das vorgegebene Szenario Basisdaten für jeden Server mit einem initialen Wert für *cpu* und *temp*, welche auf einem Zufallswert in realistischem Wertebereich basiert. Danach wird über die Liste an Servern iteriert und der Lastgenerator sendet den jeweiligen Datensatz an beide ausgewählten Datenbanken. Nach Durchlaufen der Liste wird erneut mit dem ersten Server gestartet. Die Werte *cpu* und *temp* werden anhand von Zufallswerten in eine bestimmte Richtung angepasst. Die Temperaturwerte orientieren sich hierbei an den CPU-Werten der vorherigen Messung, um realistische Verlaufskurven widerzuspiegeln. Der Lastgenerator wird einmal pro Sekunde aufgerufen. Je nach Parameter werden eine

bestimmte Anzahl an Serverwerten versendet, wobei 1 das Minimum darstellt und 40 in dem gewählten Szenario das Maximum. Da alle Werte dynamisch erzeugt werden, kann das Maximum jedoch theoretisch unbegrenzt angehoben werden, indem mehr Räume, Serverschränke oder Einheiten festgelegt werden.

6.2 Datenbankschnittstelle

Über die Klasse **DbConnector** werden die Verbindungen zu den gewählten *Real-Time Datenbanken* initialisiert.
Die Klasse **DbInterface** stellt die Schnittstelle für den einheitlichen Zugriff auf die *Real-Time Datenbanken* dar. Alle Methoden, welche sie bereitstellt, müssen ebenfalls von den konkreten Implementierungen der Klassen für die jeweiligen Anbieter bereitgestellt werden. Bei ihrer Instanziierung wird eine Kennung für den jeweiligen Anbieter übergeben. Daraufhin instanziiert sie die jeweilige Anbieterklasse und delegiert alle folgenden Methodenaufrufe an diese weiter. Die folgenden Methoden müssen von Anbieterklassen bereitgestellt werden:

- **doQuery:** Der per Konstruktor überlieferte Abfragetyp wird mit den gegebenen Parametern ausgeführt. Der Nutzer erhält als Rückgabewert ein *Observable*, an dem er sich registrieren kann.
- **updateQuery:** Der gegebene Abfragetyp wird anhand der neuen Parameter neu generiert und danach dem Nutzer analog zu der **doQuery** Methode zurückgeliefert.
- **saveData:** Der übergebene Datensatz des Lastgenerators wird in *ServerState* und *ServerData* gespeichert, bzw. die Daten angepasst.

6.2.1 Einheitliche Benachrichtigungen

Ein Herausforderung für die Gestaltung einer einheitlichen Datenbankschnittstelle stellt die Struktur der Benachrichtigungen dar. Es wurden universelle Methoden entworfen, welche einen Zugriff unabhängig von der eigentlichen *Real-Time Datenbank* ermöglichen. Die Auswahl und Art der Informationen, welche über die Benachrichtigungen mitgeteilt werden, sind für die einzelnen Systeme teilweise sehr individuell. In den folgenden Abschnitten werden diese beschrieben und abschließend eine einheitliche Struktur für diesen Anwendungsfall abgeleitet.

Firebase

Die Firebase Realtime Database bietet die Möglichkeit, für fünf verschiedene Arten von Benachrichtigungen eine Registrierung vorzunehmen [Go18b]:

- **value:** Wird erstmalig versendet mit den initialen Daten an dem spezifizierten Zugriffspfad. Danach wird bei jeder Veränderung der Daten eine Benachrichtigung mit den aktualisierten Daten versendet.

- **child_added:** Für jeden initial vorhandenen und im späteren Verlauf der Registrierung hinzukommenden Kindknoten wird eine Benachrichtigung vom System versendet.
- **child_removed:** Das System versendet eine Benachrichtigung, wenn ein Kindknoten entfernt wird oder nicht mehr Teil der Ergebnismenge der spezifizierten Abfrage ist.
- **child_changed:** Wenn die Daten eines Kindknotens verändert werden, sendet das System eine Benachrichtigung mit den aktualisierten Daten.
- **child_moved:** Wenn sich die Position eines Kindknotens auf Grund der Sortierung verändert, versendet das System eine Benachrichtigung.

Bei sortierten Abfragen wird für die Benachrichtigungstypen *child_added*, *child_changed* und *child_moved* neben den Daten des betreffenden Kindknotens zusätzlich der Schlüssel des vorhergehenden Ergebnisses geliefert. Handelt es sich um das erste Element wird für diesen Wert *null* zurückgeliefert. Da *Firebase* standardmäßig nur aufsteigende Sortierungsreihenfolge unterstützt, müssen die Begrifflichkeiten bezüglich der Reihenfolge auch in diesem Kontext verstanden werden. Im Unterschied zu den anderen Systemen muss der Nutzer bei Firebase eine explizite Registrierung für jeden einzelnen Benachrichtigungstyp vornehmen.

RethinkDB

Benachrichtigungen können bei RethinkDB um diverse optionale Metainformationen erweitert werden. Ohne weitere Angaben enthalten Benachrichtigungen zwei Felder [Re18c]:

- **old_val:** Zustand vor der Veränderung
- **new_val:** Zustand nach der Veränderung

Enthalten beide Felder einen Wert, so kann von dem Benachrichtigungstyp **change** ausgegangen werden. Ist der Wert **old_val** auf *nil* gesetzt, ist es ein neu eingefügtes Element. Wenn der **new_val** auf *nil* gesetzt ist, wurde das Element gelöscht. Optional kann ein explizites Feld für den Benachrichtigungstyp eingefügt werden [Re18d]:

- **add:** Ein Element wurde der Ergebnismenge hinzugefügt.
- **remove:** Das Element wurde aus der Ergebnismenge entfernt.
- **change:** Das Element der Ergebnismenge wurde verändert.
- **initial:** Das Element ist Teil des initialen Ergebnisses auf die Abfrage.
- **uninitial:** Ein als initial gekennzeichnetes Element wurde bereits versendet, ist jedoch durch zwischenzeitliche Veränderung Teil des noch nicht versendeten Rests der initialen Ergebnismenge geworden.
- **state:** Es handelt sich um eine optionale Statusbenachrichtigung, welche beispielsweise anzeigen kann, dass alle initialen Ergebnisse versendet wurden.

Optional kann für Abfragen mit Sortierung der alte und neue Index des Elements in Form von **old_offset** und **new_offset** in der Benachrichtigung erfasst werden.

Meteor

Meteor verwendet für die Registrierung und Benachrichtigung von Nutzern das für diesen Anwendungsfall konzipierte *Distributed Data Protcol*. Dieses stellt die folgenden Benachrichtigungsarten zur Verfügung [SWO16]:

- **added:** Ein Element wurde der Ergebnismenge hinzugefügt.
- **changed:** Das Element der Ergebnismenge wurde verändert.
- **removed:** Das Element wurde aus der Ergebnismenge entfernt.
- **ready:** Benachrichtigung, dass alle initialen Ergebnisse für eine Abfrage versendet wurden.
- **addedBefore (nicht verwendet):** Analog zu *added*, enthält zusätzlich für sortierte Abfragen den Schlüssel des nachfolgenden Elements in der Ergebnismenge.
- **movedBefore (nicht verwendet):** Analog zu *changed*, enthält zusätzlich für sortierte Abfragen den Schlüssel des nachfolgenden Elements in der Ergebnismenge.

Nur die ersten vier Typen werden von *Meteor* verwendet, die Sortierung erfolgt zu dem jetzigen Zeitpunkt clientseitig. Benachrichtigungen der Typen **added** und **changed** enthalten zudem den neuen Datensatz, **removed** enthält den Datensatz des zu löschenden Elements.

Parse

Der Parse Live Query Server verwendet fünf verschiedene Benachrichtigungstypen um den Nutzer über Veränderungen in Kenntnis zu setzen [Wa16]:

- **create**: Ein Element wurde neu erstellt und ist Teil der Ergebnismenge.
- **enter**: Ein bereits vorhandenes Element, welches zuvor nicht Teil der Ergebnismenge war, ist neu in der Ergebnismenge.
- **update**: Das Element der Ergebnismenge wurde verändert.
- **leave**: Ein Element der Ergebnismenge wurde verändert und ist nicht mehr Teil der Ergebnismenge
- **delete**: Ein Element der Ergebnismenge wurde gelöscht.

Analog zu den anderen Systemen sind ebenfalls die Daten der veränderten oder gelöschten Elemente enthalten. Wie bereits in Abschnitt 4.5.5 beschrieben wird kein Index für die sortierten Abfragen zur Verfügung gestellt.

Baqend

Die Benachrichtigungen von Baqend stellen dem Nutzer eine Reihe von Informationen zur Verfügung. Die folgenden Typen von Benachrichtigungen werden unterschieden:

- **add:** Ein Element wurde der Ergebnismenge hinzugefügt.
- **change:** Ein Element der Ergebnismenge wurde verändert und bleibt Teil der Ergebnismenge.

- **changeIndex:** Wird für sortierte Abfragen versendet, wenn ein Element nach Veränderung Teil der Ergebnismenge bleibt, seine Position in der Menge jedoch verändert wird.
- **remove:** Das Element ist nicht länger Teil der Ergebnismenge.
- **match:** Optionale Benachrichtigung, welche die Benachrichtigungstypen *add*, *change* und *changeIndex* vereint.

Benachrichtigung enthalten zudem unter anderem Felder, um mitzuteilen, ob das Element Teil der initialen Ergebnismenge ist (**initial**), an welcher Position es im Fall einer sortierten Abfrage steht (**index**) und durch welche Operation die Benachrichtigung ausgelöst wurde (**operation**).

Einheitliche Struktur

Auf Grundlage der vorhandenen Strukturen wird deutlich, dass ein Großteil der Anbieter auf die Bündelung der Benachrichtigungen in einer Registrierung setzt. Dies wird auch für die entwickelte Anwendung vorausgesetzt.
Die folgenden Felder werden für die einheitlichen Benachrichtigungen vorausgesetzt:

- **matchType:** Der Typ der Benachrichtigung. Aus den verschiedenen Lösungen der einzelnen Anbieter kann eine gemeinsame Menge an Benachrichtigungstypen in folgendem Format abgeleitet werden:
  - **add:** Ein Element ist Teil der Ergebnismenge geworden
  - **change:** Ein Element, welches Teil der Ergebnismenge war wurde verändert und bleibt Teil der Ergebnismenge
  - **move:** Für sortierte Abfragen verwendet, wenn die Position eines Elements in der Ergebnis Menge im Zuge einer Veränderung angepasst wird.
  - **remove:** Das Element ist nicht mehr Teil der Lösungsmenge
- **index:** Für den Fall einer sortierten Abfrage wird der Index des Elements in Form der absoluten Position in der Ergebnismenge angezeigt.
- **data:** Die Daten des veränderten Elements.

Durch diese Benachrichtigungsformat wird für den gegebenen Anwendungsfall die nötige Funktionalität abgedeckt. Das nachfolgende Kapitel beschreibt Details über die Implementierung der Schnittstellen für die ausgewählten Systeme. Unter anderem wird die Überführung der Benachrichtigungen in das einheitliche Format genauer betrachtet.

6.2.2 Auswahl der Systeme

Um auf der Basis des entwickelten Benchmarks Testläufe durchzuführen, müssen zwei Systeme ausgewählt werden, für welche die Datenbankschnittstelle implementiert wird. Gewählt wurden die Systeme *Firebase* und *Baqend*. Entscheidungsgrundlage sind hierfür vor allem zwei Faktoren:

- **Relevanz:** Abschnitt 4.5.7 zeigt, dass die beiden Systeme die Extreme der Bewertungsskala für funktionale Angemessenheit repräsentieren. Während Firebase star-

ke Einschränkungen in allen Aspekten aufweist, bietet Baqend in den betrachteten Aspekten funktionale Gleichwertigkeit zu *MongoDB*. Aus diesem Grund ist die Untersuchung der Auswirkungen auf Leistung und die Korrektheit interessant. Ein weiterer Punkt ist die Beliebtheit der Systeme. Hierfür gibt es keine eindeutige Metrik, da die verschiedenen Systeme selten in derselben Kategorie betrachtet werden. Ein möglicher Indikator sind die Downloadzahlen der *npm*-Pakete der einzelnen Anbieter. Mit ca. 345.000 wöchentlichen Downloads ist *Firebase* hier mit Abstand führend[12]. Es erscheint sinnvoll das beliebteste System in den Test einzubeziehen.

- **Finanzierbarkeit:** Im Rahmen dieser Arbeit ist das Aufbringen finanzieller Mittel für die Testdurchführung nicht möglich. Sowohl Firebase, als auch Baqend konnten hierfür Modelle zur Verfügung stellen.

6.2.3 Implementierungsdetails Firebase

Die Datenbankschnittstelle für *Firebase* ist in der Klasse **FirebaseClient** realisiert. Nachfolgend werden die einzelnen repräsentierten Abfragen mit der jeweiligen Repräsentation in der Abfragesprache von Firebase gezeigt.

Repräsentation der Abfragen

A1: `SELECT * FROM ServerState`

```
Firebase.database().ref('serverState/')
```

A2: `SELECT * FROM ServerState WHERE CPU >= a AND CPU <= b`

```
Firebase.database().ref('serverState/').orderByChild('cpu')
      .startAt(a).endAt(b)
```

A3: `SELECT * FROM ServerState WHERE CPU >= a AND CPU <= b AND TEMP >= c AND TEMP <= d`

Kann mit Firebase nicht realisiert werden, da nur nach einem Attribut sortiert werden kann.

A4: `SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT x`

```
Firebase.database().ref('serverState/').orderByChild('temp')
      .limitToLast(x)
```

Während die Abfrage die korrekten Werte liefert, ist die Reihenfolge jedoch falsch, da standardmäßig aufsteigend, anstatt absteigend wie hier gefordert sortiert wird.

A5: `SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT x OFFSET x`

Kann mit Firebase nicht realisiert werden, da es keine Versatzfunktion gibt.

[12] https://www.npmjs.com/package/firebase (Stand: 31.10.2018)

```
A6: SELECT * FROM ServerState WHERE CPU > a AND CPU < b
    ORDER BY TEMP DESC LIMIT x OFFSET x
```

Kann mit Firebase nicht realisiert werden, da es keine Versatzfunktion gibt. Zudem werden die Operatoren $gt und $lt nicht unterstützt.

```
A7: SELECT * FROM ServerState WHERE serverroom = r
      Firebase.database().ref('serverState/')
            .orderByChild('serverroom').equalTo(r)
A8: SELECT * FROM ServerData WHERE sid = s ORDER BY TS DESC LIMIT x
      Firebase.database().ref('serverData/'+s)
            .orderByChild('ts').limitToLast(x)[13]
A9: SELECT * FROM ServerData WHERE sid = s ORDER BY TS DESC LIMIT x
    OFFSET x
```

Kann mit Firebase nicht realisiert werden, da es keine Versatzfunktion gibt.

Überführung der Benachrichtigungsstruktur

Um die einheitliche Benachrichtigungsstruktur zu erfüllen, müssen zwei Probleme bereinigt werden. Zum einen wird bei Firebase standardmäßig für jeden Benachrichtigungstyp eine eigene Registrierung vorgenommen. Dies wird gehandhabt, indem diese zunächst einzeln gesammelt werden und daraufhin in einem gemeinsamen *Observable* gebündelt werden. Das zweite Problem ist der Sortierungsindex. Da kein absoluter Wert angegeben wird, sondern der Schlüssel des vorhergehenden Elements, muss hierfür eine extra Repräsentation der Ergebnismenge vorgehalten werden. Aus dieser lässt sich der absolute Index ableiten. Es muss beachtet werden, dass sich der Begriff *vorhergehend* hierbei auf eine aufsteigende Sortierreihenfolge bezieht. Wenn absteigend sortiert wird, bezieht sich der Schlüssel in Wirklichkeit auf das nachfolgende Element.

6.2.4 Implementierungsdetails Baqend

Die Datenbankschnittstelle für Baqend ist in der Klasse *BaqendClient* realisiert. Auch hierfür werden nachfolgend die Abfragetypen mit ihren Repräsentationen in der Abfragesprache von Baqend dargestellt.

Repräsentation der Abfragen

```
A1: SELECT * FROM ServerState
      db.ServerState.find()
```

[13] In diesem Fall wurde auf die in Abschnitt 4.5.2 beschriebene Methode zurückgegriffen, das Attribut sid als Elternknoten auszulagern. Dadurch kann über den Zugriffspfad die Gleichheit von sid mit dem Parameter s sichergestellt werden und trotzdem ein weiterer Vergleichsoperator genutzt werden.

```
A2: SELECT * FROM ServerState WHERE CPU >= a AND CPU <= b
      db.ServerState.find().between(a,b)
A3: SELECT * FROM ServerState WHERE CPU >= a AND CPU <= b
    AND TEMP >= c AND TEMP <= d
      db.ServerState.find().between('cpu',a,b).between('temp',c,d)[14]
A4: SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT x
      db.ServerState.find().descending('temp').limit(x)
A5: SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT x OFFSET x
      db.ServerState.find().descending('temp').offset(x).limit(x)
A6: SELECT * FROM ServerState WHERE CPU > a AND CPU < b
    ORDER BY TEMP DESC LIMIT x OFFSET x
      db.ServerState.find().descending('temp').gt('cpu',a).lt('cpu',b)
            .offset(x).limit(x)
A7: SELECT * FROM ServerState WHERE serverroom = r
      db.ServerState.find().equal('serverroom', r)
A8: SELECT * FROM ServerData WHERE sid = s ORDER BY TS DESC LIMIT x
      db.ServerData.find().equal('sid', s).descending('ts').limit(x)
A9: SELECT * FROM ServerData WHERE sid = s ORDER BY TS DESC LIMIT x
    OFFSET x
      db.ServerData.find().equal('sid', s).descending('ts')
            .offset(x).limit(x)
```

Überführung der Benachrichtigungsstruktur

Die Registrierung einer Abfrage liefert bei Baqend nativ ein *Observable* zurück. Mit Hilfe der *map*-Methode können die einzelnen Benachrichtigungen angepasst werden. Hierzu müssen die Benachrichtigungstypen auf die gewählten Begrifflichkeiten geändert und die relevanten Informationen herausgefiltert werden.

6.3 Ansichten

Die unterschiedlichen Ansichten repräsentieren jeweils eine Benutzerschnittstelle des entwickelten Monitoring Tools. Sie vermitteln dem Nutzer die Interaktion mit einer vollwertigen Applikation. Hierzu wurde Wert daraufgelegt, dass realitätsnahe Funktionalität gewährleistet wird.

Jede Ansicht arbeitet mit einer Instanz der Datenbankschnittstelle. In dieser wird die jeweilig relevante Abfrage hinterlegt (A1-A9). Sie registrieren sich auf diese Abfrage und verarbeiten die Benachrichtigungen. Für jede der ausgewählten *Real-Time Datenbanken*, wird jeweils eine Instanz der einzelnen Ansichten generiert. So kann der Nutzer die Ergebnisse

[14] Der Operator **between** ist die Kombination aus **$gte** und **$lte**

Seite an Seite betrachten. Wird eine Aktion durchgeführt, welche eine bestimmte Ansicht betrifft, empfangen beide Instanzen die Benachrichtigung und reagieren darauf. Dies ist beispielsweise der Fall, wenn ein Bereichsfilter auf der Übersichtsansicht ausgeführt wird. In jeder Ansicht wird der Anbieter angezeigt. Zudem werden grobe Latenzwerte angezeigt, wenn es sinnvoll ist. Hierbei handelt es sich nicht um die abschließend berechnete *Aktion-zu-Empfang-Latenz*. Diese wird erst im Nachgang während der Analyse erstellt. Die hier dargestellten Werte sollen lediglich als Anhaltspunkte während der Nutzung dienen. So können auch Nutzer, welche keine detaillierte Analyse durchführen die Latenz beurteilen. Sie basiert nur auf dem Zeitpunkt des Sendens und dem Zeitpunkt des Empfangs eines Elementes und ist aus diesem Grund nicht für die Listenansicht der heißesten Server geeignet (siehe Abschnitt 4.4.1). Dort wird sie nicht angezeigt.

6.3.1 Übersicht aller Server

In der Übersichtsansicht werden alle aktuellen Servermesswerte visualisiert. Hierzu werden sie in einem Koordinatensystem dargestellt. Wie in Abbildung x ersichtlich, wird jeder Messwert mit einem orangefarbenen Kreis dargestellt. Die x-Achse repräsentiert die CPU-Auslastung und die y-Achse die Prozessortemperatur. Das Koordinatensystem wird dynamisch an die ausgewählten Bereichsfilter angepasst. Wenn der Nutzer die Maus über einen der Kreise führt werden Details zu diesem Server angezeigt. Wird der Kreis angeklickt, erscheint der jeweilige Server in der Detailansicht (siehe 6.3.4).

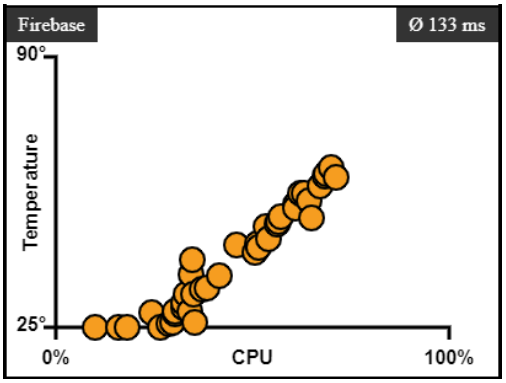

Abbildung 15: Ansicht für die Übersicht aller Server (Quelle: *eigene*)

6.3.2 Liste der heißesten Server

In dieser Ansicht werden die grundlegenden Informationen zu den aktuell heißesten Servern angezeigt. Abbildung 16: Ansicht der heißesten Server (Quelle: eigene) zeigt die gewählte Darstellung. Für jeden Server werden die Server-ID, die aktuelle Prozessortemperatur und die CPU-Auslastung angezeigt. Über die Pfeiltasten lässt sich zu den nächstheißesten Servern navigieren. Wird ein Bereichsfilter ausgewählt, wird die Liste angepasst und beinhaltet nur die entsprechenden Servermesswerte. Wird ein Listenelement von dem Nutzer angeklickt, erscheint der Server in der Detailansicht.

Firebase	◄		►
1	r1r3u4	75.41°C	82.73%
2	r1r0u3	71.37°C	78.17%
3	r1r3u3	60.82°C	57.68%
4	r2r0u1	59.34°C	62.05%
5	r1r0u0	58.27°C	65.53%
6	r2r2u0	58.02°C	62.00%
7	r1r3u1	57.25°C	66.52%
8	r1r3u0	55.84°C	62.68%
9	r2r1u0	55.29°C	62.94%
10	r2r0u2	54.62°C	63.66%
11	r2r2u1	53.53°C	58.74%
12	r2r0u0	50.61°C	48.49%

Abbildung 16: Ansicht der heißesten Server (Quelle: eigene)

6.3.3 Übersicht eines Serverraumes

Die Raumübersicht zeigt eine visuelle Repräsentation des Serverraumes. Hierzu werden die einzelnen Serverschränke und ihre Servereinheiten dargestellt. Für jede Servereinheit wird farblich kodiert die Prozessortemperatur und CPU-Auslastung angezeigt. Das Spektrum reicht hierbei von grün (geringe Temperatur/Auslastung) bis rot (hohe Temperatur/Auslastung). Wenn der Nutzer die Maus über eine Servereinheit führt, werden zusätzliche Informationen für diesen Server angezeigt. Wird ein Server angeklickt, erscheint er in der Serverdetailansicht.

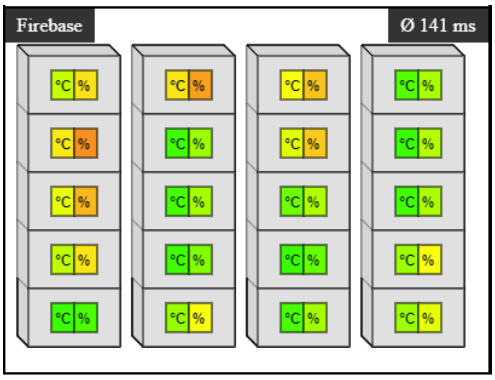

Abbildung 17: Ansicht der Raumübersicht (Quelle: *eigene*)

6.3.4 Detailansicht eines Servers

In der Serverdetailansicht wird der zeitliche Verlauf von Prozessortemperatur und CPU-Auslastung eines spezifizierten Servers dargestellt. Diese Ansicht wird repräsentiert durch die Klasse **ServerClient**. Abbildung 18 zeigt die Verlaufskurven der Messwerte über die letzten zehn Messungen. Die Darstellung wird automatisch an die gewählte Begrenzung angepasst. Über die Pfeiltasten links und rechts kann der Nutzer im Zeitverlauf navigieren.

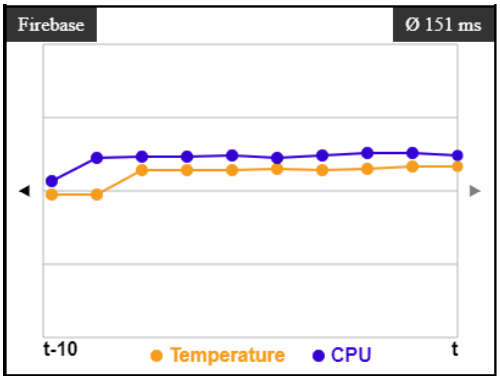

Abbildung 18: Serverdetailansicht (Quelle: *eigene*)

6.4 Steuerung

Für die Steuerung der Benchmarkapplikation sind im Kern zwei Komponenten verantwortlich. Die zentrale Komponente ist index.js. In dieser Datei werden alle anderen Komponenten instanziiert. Über den **DbConnector** werden zunächst die Verbindungen zu den Datenbanken aufgebaut. Daraufhin wird für jede Ansicht eine Instanz pro *Real-Time Datenbank* erzeugt und der Producer wird instanziiert.

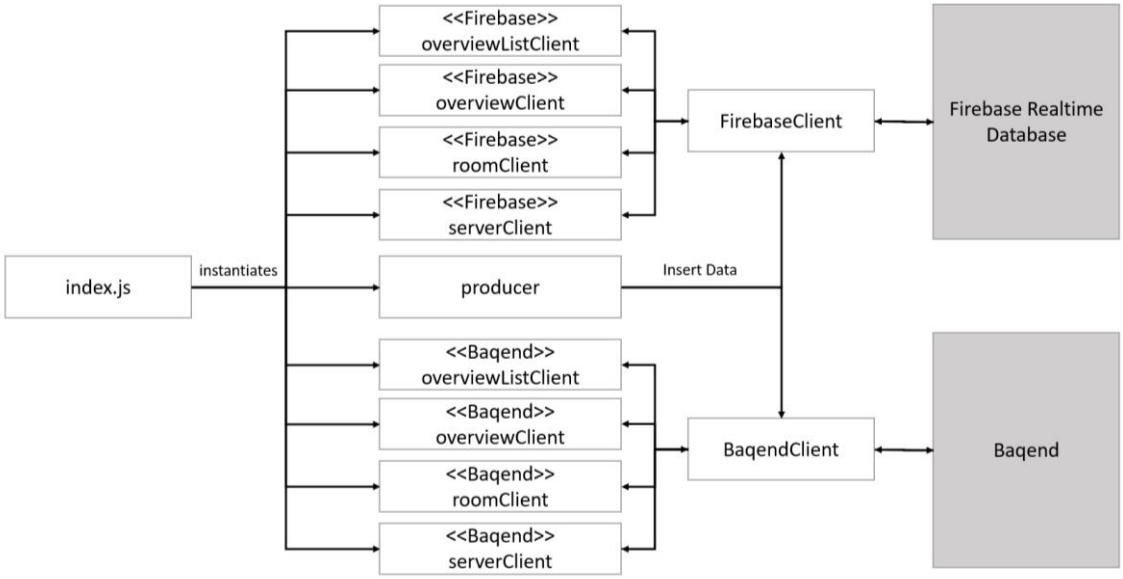

Abbildung 19: Benchmarksteuerung (Quelle: eigene)

In Abbildung 19 wird dieser Ablauf dargestellt. Wenn der Producer gestartet wird, sendet er die Datensätze simultan an die beiden Datenbankschnittstellen. Die Ansichten registrieren ihre Abfragen ebenfalls bei der jeweiligen Schnittstelle und erhalten die Benachrichtigungen der *Real-Time Datenbank*.

Abbildung 20: Kontrollleiste (Quelle: *eigene*)

Mit Hilfe der in Abbildung 20 dargestellten Kontrollleiste kann der Nutzer Einfluss auf den Benchmark nehmen. Hier kann der Lastgenerator gestartet und angehalten werden. Zudem können für die einzelnen Ansichten Parameter gesetzt werden. Für die Listenansicht der heißesten Server und die Übersichtsansicht können links Bereichsfilter gesetzt werden. Für die Raumansicht kann mittig der Raum ausgewählt werden. Für die Serverdetailansicht kann rechts ausgewählt werden, dass automatisch der heißeste Server angezeigt wird. Zudem kann die Anzahl an vergangenen Messwerten, welche in der Ansicht dargestellt werden sollen, bestimmt werden. Die Klasse Controls ist zuständig für Verteilung der ausgewählten Parameter und aller Aktionen, welche auf den Ansichten ausgeführt werden. Über diesen Weg ist sichergestellt, dass jede Aktion automatisch für alle betroffenen Ansichten durchgeführt werden und die Darstellung vergleichbar bleibt.

6.5 Messung

Die Erfassung von Messwerten findet sowohl im Lastgenerator, als auch in den Datenbankschnittstellen statt. Der Lastgenerator protokolliert alle gesendeten Elemente. Die Da-

tenbankschnittstellen protokollieren alle eingehenden Ereignisse. Über die in Abbildung 20 sichtbare Schaltfläche *export*, werden diese Komponenten benachrichtigt und exportieren die protokollierten Daten als JSON-Dateien, welche für den Nutzer heruntergeladen werden.

7 Ergebnisse

Dieses Kapitel beschreibt die Durchführung von Versuchsdurchläufen mit der entwickelten Benchmarkapplikation und stellt abschließend die Ergebnisse vor.

7.1 Versuchsaufbau

- **Zeitraum:** 600 Sekunden
- **Instanzen**: Es wird eine Instanz als Produzent und eine Instanz als Empfänger verwendet. Die Auslagerung des Produzenten in eine eigene Instanz ist erforderlich, um Aussagekräftige Messwerte zu erhalten. Dies liegt an der Eigenschaft einiger Systeme, eine clientseitige Kopie der relevanten Daten zu erfassen und dem Nutzer direkt zu liefern. Eine Kommunikation mit dem Server findet erst im Nachgang statt. Hierdurch entstehen beispielsweise bei Firebase unrealistische Latenzwerte von unter 10ms. Das Auslagern beugt diesem Verhalten vor.
- **Lastparameter**: Der Lastgenerator produziert eine Schreibaktion, in Form von Insert bzw. Update eines Messwerts, pro Sekunde.

7.2 Einschränkungen

Die Ergebnisse der präsentierten Testläufe unterliegen bestimmten Restriktionen. Baqend besitzt ein *Rate Limiting* von 5000 Anfragen pro Zeitspanne von 5 Minuten um Missbrauch vorzubeugen. Dies bedingt, dass der Lastgenerator nur mit dem Minimalwert von einer Schreibaktion pro Sekunde genutzt werden kann.

7.3 Analyse der Messwerte

Auf den protokollierten Daten werden zwei Analysen ausgeführt. Grundlage für beide ist, dass zunächst die Menge an erwarteten Benachrichtigungen generiert werden muss. Hierfür werden die Daten des Lastgenerators schrittweise durchlaufen. Je nach Abfrage wird nach jedem Schritt die Ergebnismenge berechnet. Der Vergleich mit der Ergebnismenge aus dem vorherigen Schritt legt fest, welche Art von Benachrichtigung erzeugt werden muss. Hierbei wird der Zeitstempel der Aktion festgehalten, welche die Benachrichtigung ausgelöst hat. Die berechnete Menge an Benachrichtigungen wird, sortiert nach Benachrichtigungstyp, mit der Menge an gemessenen Benachrichtigungen für diese Abfrage abgeglichen. Stimmen Server-ID (bzw. der Primärindex) der Benachrichtigungen an derselben Position in der jeweiligen Menge überein, wird davon ausgegangen, dass die Benachrichtigung korrekt generiert wurde. Für sortierte Abfragen wird zusätzlich der Sortierungsindex verglichen. Für die Messung der Latenzwerte werden der in der Erwartungsmenge protokollierte Zeitstempel mit dem Zeitstempel des Empfangs der gemessen Benachrichtigung verrechnet. Hierdurch wird die Aktion-zu-Empfangs-Latenz für jede Benachrichtigung berechnet. Der Analyseprozess ist unterschiedlich für die verschiedenen Anbieter, da

die Benachrichtigungen unterschiedlich generiert werden. Die Erwartungsmenge wird aus diesem Grund individuell für die beiden Anbieter erstellt.

7.4 Messungen

Die folgenden Ergebnisse entstanden während des Testdurchlaufs für die verschiedenen Abfragetypen:

`A1: SELECT * FROM ServerState`

Tabelle 3: Ergebnisse für den Baqend-Client *Abfrage A1*

	Erwartungswert	Messwert
Add Event	40	40
Change Event	560	560
Move Event	0	0
Remove Event	0	0

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **127ms**.

Tabelle 4: Ergebnisse für den Firebase-Client *Abfrage A1*

	Erwartungswert	Messwert
Add Event	40	40
Change Event	560	560
Move Event	0	0
Remove Event	0	0

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **168ms**.

`A2: SELECT * FROM ServerState WHERE CPU >= 40 AND CPU <= 70`

Tabelle 5: Ergebnisse für den Baqend-Client *Abfrage A2*

	Erwartungswert	Messwert
Add Event	31	31
Change Event	267	267
Move Event	0	0
Remove Event	12	12

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **132ms**.

Tabelle 6: Ergebnisse für den Firebase-Client *Abfrage A2*

	Erwartungswert	Messwert
Add Event	31	31
Change Event	267	267
Move Event	267	267
Remove Event	12	12

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **165ms**.

A3:
```
SELECT * FROM ServerState WHERE CPU >= 30 AND CPU <= 75
AND TEMP >= 40 AND TEMP <= 65
```

Tabelle 7: Ergebnisse für den Baqend-Client *Abfrage A3*

	Erwartungswert	Messwert
Add Event	35	35
Change Event	325	325
Move Event	0	0
Remove Event	11	11

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **105ms**.

- *A3 wird von Firebase nicht unterstützt*

A4:
```
SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT 18
```

Tabelle 8: Ergebnisse für den Baqend-Client *Abfrage A4*

	Erwartungswert	Messwert
Add Event	49	49
Change Event	144	144
Move Event	100	100
Remove Event	31	31

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **134ms**.

Tabelle 9: Ergebnisse für den Firebase-Client *Abfrage A4*

	Erwartungswert	Messwert
Add Event	49	49
Change Event	244	244
Move Event	244	244
Remove Event	31	31

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **169ms**.

A5: `SELECT * FROM ServerState ORDER BY TEMP DESC LIMIT 10 OFFSET 10`

Tabelle 10: Ergebnisse für den Baqend-Client *Abfrage A5*

	Erwartungswert	Messwert
Add Event	52	52
Change Event	82	82
Move Event	42	42
Remove Event	42	42

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **124ms**.

- *A5 wird von Firebase nicht unterstützt*

A6:
```
SELECT * FROM ServerState WHERE CPU > 30 AND CPU < 75
ORDER BY TEMP DESC LIMIT 10 OFFSET 10
```

Tabelle 11: Ergebnisse für den Baqend-Client *Abfrage A6*

	Erwartungswert	Messwert
Add Event	60	60
Change Event	83	83
Move Event	44	44
Remove Event	50	50

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **98ms**.

- *A6 wird von Firebase nicht unterstützt*

A7: `SELECT * FROM ServerState WHERE serverroom = 1`

Tabelle 12: Ergebnisse für den Baqend-Client *Abfrage A7*

	Erwartungswert	Messwert
Add Event	20	20
Change Event	280	280
Move Event	0	0
Remove Event	0	0

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **130ms**.

Tabelle 13: Ergebnisse für den Firebase-Client *Abfrage A7*

	Erwartungswert	Messwert
Add Event	20	20
Change Event	280	280
Move Event	0	0
Remove Event	0	0

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **152ms**.

A8:
```
SELECT * FROM ServerData WHERE sid = 'r2r2u0'
ORDER BY TS DESC LIMIT 15
```

Tabelle 14: Ergebnisse für den Baqend-Client *Abfrage A8*

	Erwartungswert	Messwert
Add Event	15	15
Change Event	0	0
Move Event	0	0
Remove Event	0	0

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **97ms**.

Tabelle 15: Ergebnisse für den Firebase-Client *Abfrage A8*

	Erwartungswert	Messwert
Add Event	15	15
Change Event	0	0
Move Event	0	0
Remove Event	0	0

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **143ms**.

```
A9: SELECT * FROM ServerData WHERE sid = 'r2r2u0'
    ORDER BY TS DESC LIMIT 3 OFFSET 3
```

Tabelle 16: Ergebnisse für den Baqend-Client *Abfrage A9*

	Erwartungswert	Messwert
Add Event	12	12
Change Event	0	0
Move Event	0	0
Remove Event	9	9

- Es wurden **keine Abweichungen** zwischen Erwartungs- und Messwert festgestellt.
- Die Analyse der *Aktion-zu-Empfang-Latenz* ergibt durchschnittlich **135ms**.

- *A9 wird von Firebase nicht unterstützt*

7.5 Zusammenfassung der Ergebnisse

Tabelle 17: Übersicht der Testergebnisse

	Baqend	Firebase
A1	127ms	168ms
A2	132ms	165ms
A3	105ms	-
A4	134ms	169ms
A5	124ms	-
A6	98ms	-
A7	130ms	152ms
A8	97ms	143ms
A9	135ms	-
Ø	120ms	159ms

Die Gesamtergebnisse in Tabelle 17 zeigen, dass Baqend im Vergleich hinsichtlich der drei ausgewählten Qualitätskriterien besser abschneidet:

- **Funktionale Korrektheit:** In keinem Test sind Auffälligkeiten oder inhaltliche Abweichungen aufgetreten. Beide Kandidaten entsprachen den Erwartungswerten.
- **Funktionale Angemessenheit:** Wie aus der Analyse der Systeme bereits hervorging, deckt Baqend die festgelegte Referenzmenge an Operatoren und Funktionen ab. Firebase schränkt den Nutzer in diesem Punkt stärker ein. Trotzdem werden fünf von neun Abfragetypen abgedeckt, darunter die grundlegenden Abfragen zu allen vier Ansichten.

- **Zeitverhalten:** Die berechneten Aktion-zu-Empfang-Latenzen liegen für beide Anbieter in einem niedrigen Bereich. Baqend bietet hier ebenfalls über alle Abfragen hinweg die besseren Werte.

8 Fazit

Zum Abschluss soll auf die einleitende Forschungsfrage dieser Arbeit zurückgeblickt werden:

> *„Was sind aussagekräftige Qualitätsdimensionen für die Bewertung von Real-Time Datenbanken und wie können diese im Rahmen einer Benchmarkingapplikation messbar gemacht werden?“*

Um diese Frage zu beantworten wurden die folgenden Beiträge geleistet:

- Eine Definition des Begriffs *Real-Time Datenbank*, welcher eine einheitliche Schnittstelle definiert und von interner Funktionalität abstrahiert. Dies ermöglicht eine Menge an heterogenen Systemen aus einer einheitlichen Perspektive zu bewerten (siehe Kapitel 2).
- Die Identifikation von relevanten Qualitätsdimensionen für *Real-Time Datenbanken*, in Form von Zeitverhalten, funktionaler Korrektheit und funktionaler Angemessenheit (siehe Kapitel 4.3).
- Die Identifikation von Metriken, um die Qualitätsmerkmale messbar zu machen. Für das Zeitverhalten wurde die Aktion-zu-Empfang-Latenz als zentrale Metrik identifiziert um Latenzmessungen für *Real-Time Datenbanken* durchzuführen. Die funktionale Korrektheit wird in Form der Abweichung zwischen einer berechneten Benachrichtigungsmenge und der gemessenen Benachrichtigungsmenge gemessen. Für die funktionale Angemessenheit wurde die Ausdruckskraft der Abfragesprache als geeignetes Mittel der Bewertung identifiziert. Diese wird in Form einer Referenzmenge an Operatoren und Funktionen aus dem Funktionsumfang der Datenbank *MongoDB* repräsentiert, welche mit dem Funktionsumfang der *Real-Time Datenbanken* abgeglichen wird. Hierfür wurden neun Abfragetypen entworfen, welche von den Vergleichskandidaten implementiert werden müssen (siehe Kapitel 4.4 und 4.5).
- Konzeption eines realitätsnahen Szenarios als Rahmen für eine applikationsgetriebene Benchmarkinganwendung. Neben der Messung der identifizierten Qualitätsmetriken bietet sie die Möglichkeit die Systeme in dem gegebenen Szenario kennenzulernen und explorativ zu untersuchen. Diese Eigenschaften waren Teil der gesammelten Anforderungen für die Zielgruppe der Softwareentwicklung. (siehe Kapitel 5 und 6).
- Konzeption und Implementierung einer einheitlichen Datenbankschnittstelle für *Real-Time Datenbanken* und zusätzliche Implementierung dieser Schnittstelle für die Anbieter *Baqend* und *Firebase* (siehe Kapitel 6.2).
- Umsetzung erster Testläufe für die ausgewählten Systeme *Baqend* und *Firebase*, sowie Analyse der Messwerte (siehe Kapitel 7).

8.1 Ausblick

Neben den Einschränkungen des Versuchsaufbaus, welche in Abschnitt 7.2 beschrieben werden, existieren einige Aspekte, welche aufgrund des Zeitrahmens in dieser Arbeit keine Berücksichtigung gefunden haben:

- Die Erweiterung der Referenzmenge auf weitere Operatoren und Funktionen sollte in Betracht gezogen werden. Vor allem im Hinblick darauf, dass zukünftig eine breitere Unterstützung für weitere Funktionalität durch *Real-Time Datenbanken* zu erwarten ist.
- Die Umsetzung des Benchmarks für die weiteren beschriebenen Systeme. Mit geringerer Restriktion wären auch stärkere Belastungsszenarien interessant.
- Die Erweiterung der Betrachtung auf neue Systeme und Anbieter. Beispielsweise wird von Google derzeit der *Cloud-Firestore* in einer Beta-Phase getestet. Dieser ist ein funktional umfangreicherer Nachfolger für die *Firebase Realtime Database*[15]
- Eine wichtige funktionale Erweiterung des Benchmarks wäre die Ausgliederung des Lastgenerators in ein eigenständiges, nicht browsergebundenes Skript. Hierdurch könnte zusätzlich die Belastung skaliert werden und auch verteilte Szenarien wären realisierbar. Aufgrund der Problematik von Latenzmessung in verteilten Szenarien, welche in Abschnitt 4.4.1 erläutert wird, würde dieses Vorgehen jedoch zu Lasten der Aussagekraft der Messwerte gehen.
- Die dynamisch generierte Belastung basiert zu einem Teil auf Zufallswerten. Diese Tatsache schränkt einen wichtigen Aspekt der Benchmarkgestaltung, die Wiederholbarkeit, bedeutend ein. Im Rahmen der Messung werden alle Schreiboperationen des Lastgenerators erfasst und als *DataWriteLog* in Form einer *JSON-Datei* exportiert. Diese Datei könnte durch den Lastgenerator eingelesen werden, um ein exakt vergleichbares Belastungsszenario mehrfach durchzuführen.
- Ein weiterer Anwendungsfall für den Benchmark wäre der Vergleich des gleichen Systems in unterschiedlichen Hosting-Umgebungen. Das Beispiel einer *Meteor* Applikation zeigt, dass dieselbe Anwendung, bei dem Anbieter *Galaxy* gehosted und zum Vergleich bei *Amazon Web Services* gehosted, Latenzunterschiede um den Faktor 10 zu Lasten von *Galaxy* aufweisen kann [Ha18]. Dadurch könnten die Cloudanbieter zum Testschwerpunkt werden.

[15] https://firebase.google.com/docs/firestore/ (Stand: 01.11.2018)

A Anhang

A.1 Struktur des Datenträgers

Der beiliegende Datenträger enthält folgende Ordnerstruktur:

- **Dokumente**: In diesem Ordner befindet sich die digitale Fassung diese Arbeit.
- **Quelltext**: Dieser Ordner beinhaltet die Quelltexte für die entwickelte Benchmarking Applikation. Gestartet werden kann das Programm mit Aufruf der index.html Datei in einem handelsüblichen Browser. Die Quelltexte sind in dem Ordner src enthalten. Um Veränderungen des Codes durchzuführen müssen die Abhängigkeiten mit Hilfe des Node Package Managers und dem Befehl npm install installiert werden. Anschließend kann ein Entwicklungsserver über npm start gestartet werden, welcher über den localhost die geänderte Fassung transpiliert. Über den Befehl npm run build kann die geänderte Fassung in eine standardmäßig lauffähige Javascript Anwendung für den Browser überführt werden. Im Unterordner doc befindet sich eine Dokumentation des Quellcodes.
- **Ergebnisse**: In einzelnen Ordnern sind die detaillierten Ergebnisse der Messungen aus Kapitel 7 enthalten. Zusätzlich hinterliegen Skripte für die Analyse der Datensätze. Die jeweiligen Skripte lassen sich über die Kommandozeile in der Form *node skript.js -p „ba“* für Baqend oder *node skript.js -p „fb“* für Firebase ausführt.

Literaturverzeichnis

[Ak09] Akhmechet, S.: RethinkDB: a new kind of database. https://www.rethinkdb.com/blog/rethinkdb-a-new-kind-of-database/, 21.10.2018.

[Ak16] Akhmechet, S.: RethinkDB is shutting down. https://www.rethinkdb.com/blog/rethinkdb-shutdown/.

[Ba16] Bakhshi, R.: One Galaxy for Everyone. https://blog.meteor.com/one-galaxy-for-everyone-e1243b10d252, 21.10.2018.

[Ba18a] Baqend: Real-Time Queries. https://www.baqend.com/guide/topics/realtime/, 28.10.2018.

[Ba18b] Baqend: Schema and Types. https://www.baqend.com/guide/topics/schema/, 21.10.2018.

[Ba18c] Baqend: Frequently Asked Questions. https://www.baqend.com/guide/topics/faq/, 21.10.2018.

[Ba18d] Baqend: Queries. https://www.baqend.com/guide/topics/queries/, 28.10.2018.

[Ba92] Basili, V. R.: Software modeling and measurement: the Goal/Question/Metric paradigm, 1992.

[BLS97] Bestavros, A.; Lin, K.-J.; Son, S.H. Hrsg.: Real-Time Database Systems. Issues and Applications. Springer, Boston, MA, 1997.

[BMX09] Barbosa, D.; Manolescu, I.; Xu Yu, J.: Microbenchmark. In (LIU, L.; ÖZSU, M. T. Hrsg.): Encyclopedia of Database Systems. Springer US, Boston, MA, 2009.

[BMX16] Barbosa, D.; Manolescu, I.; Xu Yu, J.: Application Benchmark. In (LIU, L.; ÖZSU, M. T. Hrsg.): Encyclopedia of Database Systems. Springer New York, New York, NY, 2016.

[BWT17] Bermbach, D.; Wittern, E.; Tai, S.: Cloud Service Benchmarking. Springer International Publishing, Cham, 2017.

[Ca89] Camp, R. C.: Benchmarking: the search for industry best practices that lead to superior performance, 1989.

[da16] danielmewes: Changefeeds on getIntersecting queries. https://github.com/rethinkdb/rethinkdb-legacy/commit/9e3bba095cbddfe092ab47c10380dded0b8c78b9, 28.10.2018.

[Da95] Daneva, M.: Software benchmark design and use. In (Browne, J.; O'Sullivan, D. Hrsg.): Re-engineering the Enterprise. Proceedings of the IFIP TC5/WG5.7

Working Conference on Re-engineering the Enterprise, Galway, Ireland, 1995. Springer, Boston, MA, 1995.

[DB18] DB-Engines: Firebase Realtime Database Systemeigenschaften. https://db-engines.com/de/system/Firebase+Realtime+Database, 20.10.2018.

[Fi18] Firebase Inc.: Acceptable Use Policy. https://www.firebase.com/terms/acceptable-usage-policy.html, 11.10.2018.

[Ge16] Gessert, F.: Baqend Cloud now publicly available for your applications. https://medium.baqend.com/baqend-cloud-now-publicly-available-for-your-applications-759dec35c751, 21.10.2018.

[Ge17] Gessert, F.: Lessons Learned Building a Backend-as-a-Service: A Technical Deep Dive. https://medium.baqend.com/how-to-develop-a-backend-as-a-service-from-scratch-lessons-learned-a9fac618c2ce, 21.10.2018.

[Gl17] Glukhovsky, M.: RethinkDB joins The Linux Foundation. https://rethinkdb.com/blog/rethinkdb-joins-linux-foundation/, 21.10.2018.

[Go06] Goebel, H.: Entwicklung einer Benchmarking-Methode für die Bewertung der Verbesserung von Gewässerstrukturen an Fließgewässern. Oldenbourg Industrieverlag, 2006.

[Go18a] Google: Structure Your Database. https://firebase.google.com/docs/database/ios/structure-data, 28.10.2018.

[Go18b] Google: firebase.database.Reference. https://firebase.google.com/docs/reference/js/firebase.database.Reference, 28.10.2018.

[Go18c] Google: Work with Lists of Data on the Web. https://firebase.google.com/docs/database/web/lists-of-data, 28.10.2018.

[Go18d] Google: Firebase Realtime Database. https://firebase.google.com/docs/database/, 20.10.2018.

[Gr13] Grigorik, I.: High performance browser networking. O'Reilly, North Sebastopol, California, 2013.

[Gr93] Gray, J. Hrsg.: The Benchmark handbook. For database and transaction processing systems. Kaufmann, San Mateo, Cal., 1993.

[Ha18] Hartzog, E.: AWS vs Galaxy for Meteor Hosting. https://www.erichartzog.com/blog/aws-vs-galaxy-for-meteor-hosting, 31.10.2018.

[HLC91] Haritsa, J. R.; Livny, M.; Carey, M. J.: Earliest deadline scheduling for real-time database systems. In (IEEE Hrsg.): Real-Time Systems Symposium, 1991. Proceedings., Twelfth, 1991; S. 232–242.

[Hu09] Huppler, K.: The Art of Building a Good Benchmark. In (Nambiar, R.; Poess, M. Hrsg.): Performance evaluation and benchmarking. First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24 - 28, 2009 ; revised selected papers. Springer, Berlin, Heidelberg, 2009; S. 18–30.

[In11] International Organization for Standardization (ISO): ISO/IEC 25010:2011 - Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models. ISO/IEC, 2011.

[In18] International Organization for Standardization (ISO): ISO/IEC 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 26.10.2018.

[KG94] Kao, B.; Garcia-Molina, H.: An overview of real-time database systems. In (Stoyenko, A. D. Hrsg.): Real Time Computing. Springer, 1994; S. 261–282.

[Ki11] Kincaid, J.: YC-Funded Parse: A Heroku For Mobile Apps. https://techcrunch.com/2011/08/04/yc-funded-parse-a-heroku-for-mobile-apps/, 21.10.2018.

[La17] Lacker, K.: A Parse Shutdown Reminder. https://blog.parseplatform.org/announcements/a-parse-shutdown-reminder/, 21.10.2018.

[La78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In Communications of the ACM, 1978, 21; S. 558–565.

[Le12] Lehenbauer, M.: Developers, meet Firebase! https://firebase.googleblog.com/2012/04/developers-meet-firebase.html, 20.10.2018.

[LL13] Lengstorf, J.; Leggetter, P.: Realtime Web Apps. Apress, Berkeley, CA, 2013.

[Lo16] Lohmüller, J. C.: Reactive Programming. Mehr als nur Streams und Lambdas. https://www.informatik-aktuell.de/entwicklung/programmiersprachen/reactive-programming-mehr-als-nur-streams-und-lambdas.html, 27.02.2018.

[MA09] Mertins, K.; Anderes, D. Hrsg.: Benchmarking. Leitfaden für den Vergleich mit den Besten. Symposion-Publ, Düsseldorf, 2009.

[Me18a] Meteor: Collections. https://docs.meteor.com/api/collections.html, 28.10.2018.

[Me18b] Meteor: Publish and subscribe. https://docs.meteor.com/api/pubsub.html, 28.10.2018.

[Me18c] Mezzalira, L.: Front-End Reactive Architectures. Explore the Future of the Front-End using Reactive JavaScript Frameworks and Libraries. Apress, Berkeley, CA, 2018.

[Mo18a] MongoDB, I.: Query and Projection Operators. https://docs.mongodb.com/manual/reference/operator/query/, 27.10.2018.

[Mo18b] MongoDB, I.: cursor.sort. https://docs.mongodb.com/manual/reference/method/cursor.sort/index.html, 27.10.2018.

[Mo18c] MongoDB, I.: cursor.skip. https://docs.mongodb.com/manual/reference/method/cursor.skip/index.html, 27.10.2018.

[Mo18d] MongoDB, I.: cursor.limit. https://docs.mongodb.com/manual/reference/method/cursor.limit/index.html, 27.10.2018.

[Pa18a] Parse: Queries. https://docs.parseplatform.org/js/guide/#queries, 28.10.2018.

[Pa18b] Parse: Database. https://docs.parseplatform.org/parse-server/guide/#database, 21.10.2018.

[PD99] Paton, N. W.; Diaz, O.: Active database systems. In ACM Computing Surveys (CSUR), 1999, 31; S. 63–103.

[Pu13] Purdy, D.: Welcoming Parse to Facebook. https://developers.facebook.com/blog/post/2013/04/25/welcoming-parse-to-facebook/, 21.10.2018.

[Re12] RethinkDB Team: RethinkDB is out: an open-source distributed database. https://www.rethinkdb.com/blog/rethinkdb-12-release/, 21.10.2018.

[Re18a] RethinkDB: JavaScript ReQL command reference. https://rethinkdb.com/api/javascript/, 28.10.2018.

[Re18b] RethinkDB: ReQL command: changes. https://rethinkdb.com/api/ruby/changes/, 28.10.2018.

[Re18c] RethinkDB: ReQL command: changes. https://rethinkdb.com/api/ruby/changes/, 30.10.2018.

[Re18d] RethinkDB: Changefeeds in RethinkDB. https://rethinkdb.com/docs/changefeeds/ruby/, 28.10.2018.

[Sc12] Schmidt, G.: Meteor's new $11.2 million development budget. https://blog.meteor.com/meteors-new-11-2-million-development-budget-7370586949e7, 21.10.2018.

[SWO16] Stubailo, S.; Willson, H.; Oliver, A.: DDP Specification. https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md, 28.12.2018.

[Ta14] Tamplin, J.: Firebase is Joining Google! https://firebase.googleblog.com/2014/10/firebase-is-joining-google.html, 20.10.2018.

[TP18] TPC: Active TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp, 11.10.2018.

[Ul98] Ulrich, P.: Organisationales Lernen durch Benchmarking. Deutscher Universitätsverlag, Wiesbaden, 1998.

[Wa16] Wang, M.: Parse LiveQuery Protocol Specification. https://github.com/parse-community/parse-server/wiki/Parse-LiveQuery-Protocol-Specification, 28.10.2018.

[WGF+17] Wingerath, W.; Gessert, F.; Friedrich, S.; Witt, E.; Ritter, N.: The Case For Change Notifications in Pull-Based Databases. In Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs ‚Datenbanken und Informationssysteme" (DBIS), 6.-10., 2017.

[WH00] Wan, Z.; Hudak, P.: Functional reactive programming from first principles. In (Lam, M. Hrsg.): Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. ACM, New York, NY, 2000; S. 242–252.

[Wi15] Wiese, L.: Advanced Data Management. De Gruyter, s.l., 2015.

[Wi17a] Wingerath, W.: 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, 26.02.2018.

[Wi17b] Wingerath, W.: Going Real-Time Has Just Become Easy: Baqend Real-Time Queries Hit Public Beta. https://medium.baqend.com/going-real-time-has-just-become-easy-baqend-real-time-queries-hit-public-beta-3a44a13fde86, 21.10.2018.

[WVB14] Wieczorek, M.; Vos, D.; Bons, H.: Systems and Software Quality. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

Abbildungsverzeichnis

Tabellenverzeichnis

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient 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.

Ich bin mit einer Einstellung meiner Abschlussarbeit in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den

[Name des Verfassers]