Ableitung von Methoden zur Steigerung der Effizienz in der Verwendung des Cloud Firestores in bestehenden Anwendungen

David Schütz

7206452 (FH Dortmund) / 11144245 (TH Köln)

MASTERTHESIS

eingereicht am

Verbundstudiengang

Wirtschaftsinformatik

der FH Dortmund / TH Köln

im August 2023

Erstprüfer:in:

Heide Faeskorn-Woyke
Prof. Dr.

Zweitprüfer:in:

Birgit Bertelsmeier
Prof. Dr.

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt. Die vorliegende, gedruckte Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Münster, am 16. August 2023

David Schütz

Inhaltsverzeichnis

Erklärung iii

Abkürzungsverzeichnis vii

Abbildungsverzeichnis ix

Tabellenverzeichnis x

1 Einleitung 1

1.1 Problemstellung . . . 1

1.2 Zielsetzung . . . 2

1.3 Methodik . . . 2

1.4 Aufbau . . . 3

2 Theoretischer Rahmen 4

2.1 Datenbanktypen im Vergleich . . . 4

2.1.1 Relationale Datenbanken . . . 4

2.1.2 Nicht relationale Datenbanken . . . 10

2.2 Google Cloud Firestore . . . 11

2.2.1 Datenmodell . . . 12

2.2.2 Hierarchische Daten . . . 13

2.2.3 Zugriffsoperationen . . . 14

2.2.4 Security . . . 16

2.2.5 Varianten zur Verwendung von Collections 18
2.2.6 Fortgeschrittene Techniken zur Strukturierung von Daten 22
2.3 Aufstellung von Hypothesen 23

3 Methodik 24
3.1 Versuchsaufbau 24
3.1.1 Szenario: Pseudo Relational 26
3.1.2 Szenario: Denormalized 27
3.1.3 Szenario: Shared View 28
3.1.4 Szenario: Dedicated View 28
3.2 Datenerhebung 29
3.2.1 Messinstrumente für Kosten und Performance 29
3.2.2 Messinstrumente für Komplexität und Codemetriken 32
3.2.3 Strukturierung und Durchführung der Messungen 32
3.3 Datenaufbereitung 34
3.3.1 Validierung der Ergebnisse 34
3.3.2 Datageneration 35
3.3.3 Read Tests 36
3.3.4 Write Tests 36
3.3.5 Metriken aus SonarQube 36

4 Ergebnisse 38
4.1 Szenario: Pseudo Relational 38
4.2 Szenario: Denormalized 42
4.3 Szenario: Shared View 48
4.4 Szenario: Dedicated View 54

5 Diskussion 61
5.1 Analyse der Szenarien 61
5.1.1 Einfluss auf Kosten 61

5.1.2 Verständlichkeit und Wartbarkeit des Codes 64
5.1.3 Auswirkungen auf Performance 65
5.2 Beschränkungen des Experiments 68
5.3 Untersuchung der Hypothesen 69
5.4 Ableitung von Handlungsempfehlungen 70

6 Fazit **72**

Quellenverzeichnis **74**

Literatur 74
Online-Quellen 74

Abkürzungsverzeichnis

bspw. beispielsweise

bzgl. bezüglich

bzw. beziehungsweise

ca. circa

DBMS Database Management System

DDL Data Definition Language

etc. et cetera

evtl. eventuell

f. folgende

GCP Google Cloud Platform

H0 Nullhypothese

HTTP Hypertext Transfer Protocol

i. d. R. in der Regel

inkl. inklusive

IT Informationstechnologie

JSON JavaScript Object Notation

LoC Lines of Code

MB Megabyte

o. g. oben genannte

SaaS Software as a Service

SDK Software Development Kit

SPA Single Page Application

SoC System-on-a-Chip

SQL Structured Query Language

u. a. unter anderem

u. U. unter Umständen

URL Uniform Resource Locator

Vgl. vergleiche

Abbildungsverzeichnis

2.1 Erste Normalform ist verletzt 7

2.2 Erste Normalform ist erfüllt 7

2.3 Zweite Normalform ist verletzt 7

2.4 Zweite Normalform ist erfüllt 8

2.5 Dritte Normalform ist verletzt 8

2.6 Dritte Normalform ist erfüllt 9

2.7 Datenmodell des Firestores 12

2.8 Hierarchische Daten im Firestore 13

2.9 Schichtenarchitektur mit Firestore 17

5.1 Vergleich der Datageneration Costs 62

5.2 Vergleich der Read Costs (100 Runs) 63

5.3 Vergleich der Write Costs 64

5.4 Vergleich der Metriken zur Komplexität 65

5.5 Vergleich der Metriken zur Read Performance 66

5.6 Vergleich der Metriken zur Write Performance 67

Tabellenverzeichnis

2.1 Fundamentale Aspekte des relationalen Modells 5
2.2 Häufig genutzte Constraints und ihre Ziele in der Praxis 6

3.1 Versionsnummern der verwendeten Frameworks und Technologien 25
3.2 Übersicht der Perspektiven 25
3.3 Übersicht der Szenarien des Experiments 26
3.4 View Collections für Szenario *Dedicated View* 29
3.5 Initialer Datenbestand 32
3.6 Test Suiten in Phase 2 33
3.7 Anzahl der Performance Events gruppiert nach URL 35
3.8 Metriken aus SonarQube 37

4.1 Metrik: Datageneration (Pseudo Relational) 39
4.2 Metrik: Reads (Pseudo Relational) 40
4.3 Metrik: Writes (Pseudo Relational) 41
4.4 Metrik: SonarQube (Pseudo Relational) 42
4.5 Metrik: Datageneration (Denormalized) 43
4.6 Metrik: Reads (Denormalized) 44
4.7 Metrik: Reservation Writes (Denormalized) 44
4.8 Metrik: Seat Writes (Denormalized) 45
4.9 Metrik: Room Writes (Denormalized) 46
4.10 Metrik: Compound Writes (Denormalized) 47

4.11 Metrik: SonarQube (Denormalized) 48
4.12 Metrik: Datageneration (Shared View) 49
4.13 Metrik: Reads (Shared View) 50
4.14 Metrik: Reservation Writes (Shared View) 50
4.15 Metrik: Seat Writes (Shared View) 51
4.16 Metrik: Room Writes (Shared View) 52
4.17 Metrik: Compound Writes (Shared View) 53
4.18 Metrik: SonarQube (Shared View) 54
4.19 Metrik: Datageneration (Dedicated View) 55
4.20 Metrik: Reads (Dedicated View) 55
4.21 Metrik: Reservation Writes (Dedicated View) 56
4.22 Metrik: Seat Writes (Dedicated View) 57
4.23 Metrik: Room Writes (Dedicated View) 58
4.24 Metrik: Compound Writes (Dedicated View) 59
4.25 Metrik: SonarQube (Dedicated View) 60

Kapitel 1

Einleitung

Das Konzept einer Datenbank ist bei vielen Menschen in der IT untrennbar mit dem relationalen Modell nach Codd und ferner relationalen Datenbanken verbunden. Die gängigen Vertreter dieser Spezies stellen bis heute einen essenziellen Teil im Bereich der Softwareentwicklung dar, wobei sie Marktanteile von über 70 % für sich beanspruchen können. Der Vergleich der Datenbanktypen zeigt jedoch auch, dass das Interesse an relationalen Datenbanken sich über die Jahre stabilisiert hat, während alternative Konzepte wie *Document Store* oder *Key Value Store* deutlich steigende Popularität aufweisen[1]. Daten sind heute ein so fundamental wichtiges und wertvolles Gut, dass im letzten Jahrzent viele neue Produkte in diesem Bereich entwickelt wurden. Diese weisen häufig eine intrinsische Skalierbarkeit für Cloud Computing auf. So ist es bspw. Cockroach Labs mit CockroachDB gelungen, eine uneingeschränkt skalierende Datenbank zu entwickeln, die kompatibel zu PostgreSQL ist[2]. Google bietet mit Firebase eine komplette App Plattform an, welche u. a. mit dem Cloud Firestore einen performanten und vielseitigen *Document Store* beinhaltet[3]. Dies sind lediglich zwei Beispiele auf einer langen Liste voll spannender Innovationen.

1.1 Problemstellung

Der Erfahrungsschatz im Bereich der relationalen Datenbanken ist bedingt durch die lange Historie und dank des breiten Informationsangebots im Internet effektiv unbegrenzt, während Fallstricke für neuere Technologien erst nach und nach bekannt bzw.

[1] Vgl. solidIT consulting & software development GmbH, 2023.
[2] Vgl. Cockroach Labs, 2023.
[3] Vgl. Google, 2022b.

ausgemerzt werden. Der Reifegrad älterer Software ist i. d. R. deutlich höher und die Dokumentation im Zweifel ausführlicher. Die Structured Query Language (SQL) wird aus eigener Erfahrung im Laufe der Ausbildung gelehrt, während alternative Formen von Datenbanken wenn überhaupt meist weniger gründlich behandelt werden. Daher ist es nur natürlich, dass bei der Verwendung nicht relationaler Datenbanken in Ermangelung entsprechender Erfahrung häufig unbewusst versucht wird, diese in das Konzept der relationalen zu zwingen. Auf diese Weise werden klassische Muster angewandt anstatt den innovativen Grundgedanken anzunehmen. So entsteht sowohl im privaten als auch im unternehmerischen Kontext Software, die neue Technologien zwar wohlwollend adaptiert, dabei jedoch technische Schulden aufbaut, Potenziale für eine bessere User Experience ungenutzt lässt und unnötig hohe Kosten verursachen kann. Solche Anwendungen werden häufig zunächst prototypisch von einzelnen Mitarbeitenden oder kleinen Teams implementiert, bevor sie Traktion aufnehmen können. Im Rahmen der produktiven Nutzung fällt später meist auf, dass der Cloud Firestore höhere Kosten als für das Nutzervolumen erwartet erzeugt. Des Weiteren herrscht in Projektteams gerne Unklarheit darüber, ob die Art und Weise, wie die Datenbank eingesetzt wird, nachhaltig und langfristig wartbar ist. Diese Arbeit stellt daher die folgende Forschungsfrage inkl. der anschließenden Unterfragen:

Welche Ansätze sind geeignet, um die Verwendung des Google Cloud Firestores im Rahmen einer solchen Anwendung effizienter zu gestalten?

- Welchen Einfluss haben die gewählten Maßnamen auf die Kosten?
- Wie verhalten sich Verständlichkeit und Wartbarkeit der Anwendung?
- Welche Auswirkungen auf Performance erwirken die Ansätze?

1.2 Zielsetzung

Ziel ist es, am Ende dieser Arbeit Erkenntnisse über die Eigenschaften von Firestore Apps zu gewinnen, die dabei helfen, Refactorings effizient zu steuern und so Kostensenkungen herbeizuführen. Potenzielle Seiteneffekte, die einen Einfluss auf die Komplexität und Qualität der Anwendung sowie des Quellcodes haben, sollen sichtbar gemacht werden, um Fallstricke für zukünftige Projekte zu verhindern.

1.3 Methodik

Die Zielsetzung dieser Masterarbeit soll mithilfe eines Experiments erreicht werden. Dabei wird ein Versuchsaufbau hergestellt, der eine prototypische Implementierung ver-

schiedener Ansätze zur Strukturierung von Daten im Cloud Firestore untersucht. Die Methode betrachtet den Forschungsgegenstand aus diversen Perspektiven und steht im Zentrum der Beantwortung der Forschungsfrage.

1.4 Aufbau

Kapitel 2 beleuchtet zunächst die theoretischen Hintergründe relationaler und nicht relationaler Datenbanken. Hier wird zudem der Google Cloud Firestore behandelt, da dieser in der Forschungsfrage eine zentrale Rolle einnimmt. In Kapitel 3 folgt die Beschreibung des Experiments, welches mithilfe eines vergleichenden Vorgehens die Effekte spezifischer Entwicklungsmaßnahmen untersucht. Kapitel 4 fasst die Ergebnisse aus dem Experiment zusammen. Diese werden in Kapitel 5 diskutiert. In Kapitel 6 folgt ein abschließendes Fazit mit Ausblick auf weitere Potenziale.

Kapitel 2

Theoretischer Rahmen

Hinweis: Teile dieses Kapitels wurden aus der Projektarbeit entnommen.

2.1 Datenbanktypen im Vergleich

Im Bereich Datenbanken gibt es verschiedene Konzepte zur Strukturierung von Daten. Am verbreitetsten sind dabei relationale Systeme, allerdings stellen nicht relationale Datenbanken heute eine ernstzunehmende Alternative für verschiedene Anwendungsfälle dar. Im Vergleich liegen relationale Datenbanken zwar noch weit vorne, jedoch verzeichnen andere Typen wachsende Popularität[1]. In diesem Kapitel werden relationale und nicht relationale Datenbanken gegenübergestellt.

2.1.1 Relationale Datenbanken

Date schreibt, dass die Einführung des relationalen Modells Anfang 1970 das wichtigste Ereignis in der Geschichte der Datenbankenforschung war. Bei dem Modell handelt es sich um ein System, welches Daten stets in Tabellenform darstellt sowie Operatoren zur Limitierung von Zeilen und Spalten bereitstellt. Während relationale Datenbanken lediglich auf Tabellen basieren, bieten alternative Datenbanktypen auch andere Datenstrukturen und Zugriffsoperatoren.[2]

[1] Vgl. solidIT consulting & software development GmbH, 2022.
[2] Vgl. Date, 1981, S.26f.

Das relationale Modell

Das relationale Modell basiert wie in Tabelle 2.1 beschrieben auf drei Säulen: Struktur, Integrität und Manipulation. Die drei wichtigsten Operatoren sind *restrict*, *project* sowie *join*. Der *restrict* Operator schränkt Tabellen auf Zeilenebene ein, während *project* auf Spaltenebene arbeitet. *Join* verbindet Tabellen miteinander, indem gemeinsame Werte verschiedener Spalten kombiniert werden[3]. Da der Output eines Operators stets eine Tabelle darstellt, ist eine Verwendung als Input für einen weiteren Operator möglich. So können Operationen im relationalen System miteinander verkettet werden[4].

Tabelle 2.1: Fundamentale Aspekte des relationalen Modells

Aspekt	*Beschreibung*
Struktur	Die einzige Darstellungsform der Daten ist die Tabelle
Integrität	Constraints stellen die Integrität der Daten sicher
Manipulation	Alle Operatoren des Systems leiten Tabellen aus Tabellen ab

Constraints

Relationale Datenbanken sehen sogenannte Constraints[5] vor, um Integritätsbedingungen zu wahren. Dabei handelt es sich um boolsche Ausdrücke, die zu jeder Zeit den Wert *true* ergeben müssen. Constraints werden in der Data Definition Language (DDL) formal deklariert und zur Laufzeit vom Database Management System (DBMS) forciert[6]. In der Praxis kommen am häufigsten die in Tabelle 2.2 genannten Einschränkungen zum Einsatz. *Primary Keys* werden bspw. zur Identifizierung von Zeilen einer Tabelle sowie zur automatischen Indizierung[7] verwendet, während *Foreign Keys* integre Beziehungen zwischen Tabellen herstellen. Dieser Mechanismus sorgt dafür, dass bspw. ein Datensatz aus einer fiktiven Tabelle *Bestellung* nicht gelöscht werden kann, solange zugehörige Datensätze in einer Tabelle *Bestellposition* existieren[8]. *Unique* ist eine „schwächere“ Form eines *Primary Keys*, da diese Constraint lediglich die Eindeutigkeit eines Werts in einer Spalte sicherstellt, nicht aber für die Identifizierung zuständig ist.

[3] Vgl. Date, 1981, S.60.
[4] Vgl. Date, 1981, S.62.
[5] Englisch für „Einschränkung“
[6] Vgl. Date, 1981, S.254.
[7] Ein Mechanismus des DBMS, der Daten so aufbereitet, dass lesende Zugriffe mit einer Einschränkung auf die indizierte Spalte so effizient wie möglich ablaufen.
[8] Vgl. Date, 1981, S.272.

Tabelle 2.2: Häufig genutzte Constraints und ihre Ziele in der Praxis

Art	*Ziel*
Primary Key	Kombination aus Eindeutigkeit + automatischer Indizierung
Foreign Key	Referenzierung von Zeilen in einer anderen Tabelle
Unique	Eindeutigkeit von Werten in einer Spalte der Tabelle

Über diese Standards hinaus können durch Constraints beliebige fachliche Regeln vom DBMS durchgesetzt werden[9]. In der modernen Softwareentwicklung sieht man in der Regel davon ab, zu viel Fachlogik in Constraints zu verankern, da diese bspw. über verschiedene Tabellen hinweg verstreut sein können und fachliche Zusammenhänge somit schwerer erfass- und wartbar sind. Stattdessen werden Constraints häufig im Programmcode durchgesetzt, da dieser mit heutigen Tools deutlich einfacher zu testen, versionieren und warten ist.

Normalisierung

Maier definiert drei Normalformen für relationale Datenbanken. Normalisierung bezeichnet dabei den Vorgang, das Datenbankschema so anzupassen bzw. zu gestalten, dass bestimmte unerwünschte Eigenschaften auf Schemaebene ausgeschlossen werden[10]. Welche Eigenschaften das konkret sind, beschreiben die Normalformen im Einzelnen.

Erste Normalform

Eine Relation befindet sich dann in der ersten Normalform, wenn alle Attribute atomar vorliegen. Ein Attribut gilt dann als atomar, wenn es sich nicht um zusammengesetzte oder listenartige Werte handelt[11]. Ein zusammengesetzter Wert kann bspw. ein Datumswert sein, wobei diese in der Praxis häufig durch einen entsprechenden Datentyp abgebildet werden. In Abbildung 2.1 ist die erste Normalform verletzt, da die Spalte *Name* listenartige Werte enthält.

[9] Vgl. Date, 1981, S.254.
[10] Vgl. Maier, 1983, S.96.
[11] Vgl. Maier, 1983, S.96.

Alter	Name
25	[Jasmin]
26	[David, Marie, Robin]
29	[Tim]

Abbildung 2.1: Erste Normalform ist verletzt

Abbildung 2.2 löst dieses Problem auf, indem jeder Listenwert in *Name* in eine eigene Zeile verschoben wird. Der Wert *26* kommt nun häufiger in der Spalte *Alter* vor. Die erste Normalform bietet den Vorteil, dass eine Erweiterung des Schemas deutlich einfacher bzw. generell erst sinnvoll möglich ist. In Abbildung 2.2 wurde bspw. die Spalte *Geschlecht* ergänzt. In der Ursprungsform wäre das nicht möglich gewesen, da der Wert *weiblich* nicht zu allen Einträgen in den Listen passt.

Alter	Name	Geschlecht
25	Jasmin	weiblich
26	David	männlich
26	Marie	weiblich
26	Robin	männlich
29	Tim	männlich

Abbildung 2.2: Erste Normalform ist erfüllt

Zweite Normalform

Die zweite Normalform liegt vor, wenn eine Relation die erste Normalform erfüllt und zusätzlich alle Nichtschlüsselattribute funktional vom gesamten Primärschlüssel abhängen[12]. In der Literatur wird sehr häufig das Beispiel einer Bestelldatenbank gegeben, bei der die Adressen zunächst in der Bestellung enthalten sind (siehe Abbildung 2.3). Hier hängen die Attribute *Name*, *Vorname*, *Straße*, *PLZ* und *Ort* nicht vom Primärschlüssel der Bestellung ab, sondern bilden eine eigene fachliche Entität.

Bestellung						
ID	Datum	Name	Vorname	Straße	PLZ	Ort
100	19.04.22	Schütz	David	Musterstraße 42	10101	Musterstadt

Abbildung 2.3: Zweite Normalform ist verletzt

[12] Vgl. Date, 1981, S.361.

In Abbildung 2.4 wird die zweite Normalform erfüllt, da die oben genannten Attribute in eine eigene Tabelle *Kunde* ausgelagert wurden. Die Tabelle Bestellung enthält lediglich einen Fremdschlüssel *Kunde_ID*. So werden Redundanzen sowie Anomalien durch Updates vermieden.

Bestellung					
ID	**Datum**	**Kunde_ID**			
100	19.04.22	22			
Kunde					
ID	**Name**	**Vorname**	**Straße**	**PLZ**	**Ort**
22	Schütz	David	Musterstraße 42	10101	Musterstadt

Abbildung 2.4: Zweite Normalform ist erfüllt

Dritte Normalform

Die dritte Normalform ist erfüllt, wenn sich die Relation in der ersten und zweiten Normalform befindet und zudem kein Nichtschüsselattribut funktional von einem anderen Nichtschüsselattribut abhängt[13]. Auf diese Weise werden transitive Abhängigkeiten zwischen Nichtschlüsselattributen ausgemerzt. In der Literatur wird zur Erklärung der dritten Normalform häufig ein Beispiel mit Postleitzahlen und Orten gegeben. Die Relation *Kunde* in Abbildung 2.5 enthält die Attribute *PLZ* und *Ort*, wobei letzteres von *PLZ* funktional abhängt. Da es sich dabei jedoch nicht um ein Schlüsselattribut handelt, muss das Datenmodell für den Übergang in die dritte Normalform angepasst werden.

Bestellung					
ID	**Datum**	**Kunde_ID**			
100	19.04.22	22			
Kunde					
ID	**Name**	**Vorname**	**Straße**	**PLZ**	**Ort**
22	Schütz	David	Musterstraße 42	10101	Musterstadt

Abbildung 2.5: Dritte Normalform ist verletzt

Abbildung 2.6 erfüllt die dritte Normalform, da die Orte dort in eine eigene Relation

[13] Vgl. Date, 1981, S.358.

ausgelagert und in der Tabelle *Kunde* mithilfe eines Fremdschlüssels referenziert werden. Auf die *Ort_ID* zu verzichten und stattdessen die *PLZ* als Schlüssel auszuwählen, wäre nicht korrekt, da die Beziehung zwischen Postleitzahl und Ortsname in Deutschland nicht in allen Fällen eindeutig ist. Die Verwendung eines technischen Schlüssels ist sowohl generell als auch an dieser konkreten Stelle empfehlenswert.

Bestellung				
ID	**Datum**	**Kunde_ID**		
100	19.04.22	22		
Kunde				
ID	**Name**	**Vorname**	**Straße**	**Ort_ID**
22	Schütz	David	Musterstraße 42	4
Ort				
ID	**PLZ**	**Ort**		
4	10101	Musterstadt		

Abbildung 2.6: Dritte Normalform ist erfüllt

Transaktionen

Eine Transaktion beschreibt eine logische Zusammenfassung von Arbeit, die sich von außen betrachtet als atomarer Prozess darstellt. Ziel einer Transaktion ist die Überführung eines aus fachlicher Sicht sinnvollen Datenzustands in einen neuen validen Zustand. Innerhalb einer Transaktion finden in der Regel jedoch mehrere Operationen statt, sodass temporär invalide Zustände auftreten können, die von außen betrachtet jedoch nicht sichtbar sind. In relationalen Datenbanksystemen übernimmt der so genannte *Transaction Manager* die Aufgabe, Transaktionen zu überwachen, die Ergebnisse bei Erfolg in die Datenbank zu übernehmen und in Fehlersituationen die Operationen zurückzurollen. Auf diese Weise wird die Atomarität einer Transaktion technisch sichergestellt. SQL bietet standardmäßig den Befehl *BEGIN TRANSACTION* zum Starten sowie *COMMIT* und *ROLLBACK* zum Steuern einer Transaktion an[14].

[14] Vgl. Date, 1981, S.446f..

2.1.2 Nicht relationale Datenbanken

Wie in Kapital 2.1.1 erläutert war die Einführung des relationalen Modells durch Codd ein folgenreiches und wichtiges Ereignis in der Geschichte der Datenbankenforschung. Die rapide Entwicklung des Internets und die damit verbundenen steigenden und immer diversifizierteren Datenmengen stellen Unternehmen seit der Jahrtausendwende jedoch vor neue Herausforderungen in Bezug auf die effiziente Verarbeitung von Daten. Nach Gartner handelt es sich bei *Big Data* um hochvolatile, diverse Informationen in großen Mengen, deren Umgang innovative Techniken erfordert und zu verbesserten Entscheidungsgrundlagen und Prozessautomatisierung führen kann[15]. Daten liegen demnach nicht mehr ausschließlich in relationaler Form vor, sondern können auch unstrukturierte Informationen wie Bilder, Audio und Videos enthalten. Darüberhinaus stellen auch Sensoren potenzielle Datenquellen für *Big Data* dar. Die Anforderungen an solche Datenbanken unterscheiden sich je nach Anwendungsfall also von den klassischen relationalen Vertretern, da sie skalierbar und schematisch flexibel, aber gleichzeitig weiterhin sicher und zuverlässig arbeiten müssen. Es gibt verschiedene Arten nicht relationaler Datenbanken, die im Folgenden erläutert werden.

Key Value

Eine Key Value Datenbank ist die einfachste Art nicht relationaler Datenbanken und speichert Daten in simplen Wertepaaren ab. Jedes Tupel kann mit einem Key identifiziert werden und enthält einen Value. Ein Value kann dabei durch einen einzelnen Wert bis hin zu komplexen Datenstrukturen abgebildet werden[16]. Das Sessionmanagement einer Webanwendung basiert häufig auf Key Value Stores. Jede Session wird durch eine ID identifiziert und enthält bspw. persönliche Informationen, den Warenkorb und weitere relevante Informationen. Dieser Datenbankentyp ist hochskalierbar, schemalos und auf massive Schreib- und Leseoperationen ausgelegt[17].

Document

Dokumentenbasierte Datenbanken speichern Daten in Form von *strukturierten* Dokumenten ab. Das Format ist grundsätzlich flexibel, allerdings wird häufig XML oder JSON verwendet. Im Vergleich zu Key Value Datenbanken weisen Dokumente zwar eine Struktur auf, allerdings gibt das System kein Schema wie bei relationalen Datenbanken

[15] Vgl. Gartner, 2022.
[16] Vgl. Meier und Kaufmann, 2019, S.18.
[17] Vgl. Meier und Kaufmann, 2019, S.203f.

vor. Die Inhalte können sich also von Dokument zu Dokument unterscheiden. Ein sehr großer Vorteil dokumentenbasierter Datenbanken ist die horizontale Skalierbarkeit. Das System kann Dokumente über sogenannte *Shards* verteilen, da diese keine physischen Abhängigkeiten zueinander aufweisen. Ein Nachteil ist die fehlende referentielle Integrität, welche nicht von der Datenbank, sondern von der konsumierenden Anwendung sichergestellt werden muss. Die führt auch zu fehlender Normalisierung und potenzieller Duplizierung von Daten. Des Weiteren kann die Replizierung über verschiedene Shards zu temporären Inkonsistenzen führen, was bei der Implementierung der Anwendung bedacht werden muss. Grundsätzlich handelt es sich bei diesem Datenbankentyp um einen Key Value Store, welcher Dokumente als Values verwendet[18].

Graph

Die genannten nicht relationalen Datenbanktypen basieren alle grundsätzlich auf dem Key Value Konzept. Graph Datenbanken unterscheiden sich in dieser Hinsicht stark von den Alternativen, da sie nicht schemalos sind und deren Datenmodell auf Knoten und Kanten basiert. Zwei Knoten eines Graphs können mit Kanten verbunden werden, wobei sowohl die Knoten als auch die Kanten einen Typ aufweisen. Nutzdaten werden dort in Form von Key Value Paaren vorgehalten. Das Schema der Datenbank ergibt sich implizit aus dem bestehenden Graph. Dies hat den Vorteil, dass neue Daten eingefügt werden können, ohne vorher das Schema explizit zu erweitern, während das bestehende Schema durch das System durchgesetzt werden kann. Auch das Konzept der referentiellen Integrität wird so unterstützt[19]. Graph Datenbanken sind besonders nützlich für Anwendungsfälle, in denen die Beziehungen zwischen Datensätzen wichtiger sind als die Daten selbst (bspw. Social Media, Netzwerke, Stammbäume und 3D-Modellierung). Der große Vorteil im Vergleich zu anderen Datenbanken ist die konstante Zugriffszeit auf direkte Nachbarn im Graph, da dafür kein Lookup in einer Beziehungstabelle nötig ist[20].

2.2 Google Cloud Firestore

Der Cloud Firestore ist ein NoSQL Datenbankprodukt von Google. Die Datenbank ist flexibel und skalierbar sowohl für Smartphone Apps als auch Web- und Serverentwicklung einsetzbar. Es handelt sich dabei um ein von Google in der Cloud betriebenes Produkt. Eine Besonderheit ist die native Unterstützung von Echtzeitaktualisierungen sowie

[18]Vgl. Meier und Kaufmann, 2019, S.207f.
[19]Vgl. Meier und Kaufmann, 2019, S.215.
[20]Vgl. Meier und Kaufmann, 2019, S.216.

Offline Support[21]. Google bietet für alle gängigen Programmiersprachen Client SDKs an. Im Folgenden werden die Konzepte und aktuelle Empfehlungen von Google für den Einsatz der Datenbank beleuchtet.

2.2.1 Datenmodell

Der Firestore ist eine dokumentenbasierte Datenbank, in der es zwei strukturierende Elemente gibt: Sammlungen (Collections) und Dokumente. Ein Dokument besteht im Kern aus JSON, kann bis zu 1 MB Speicherplatz belegen und unterstützt einige weitere Datentypen, die von Google hinzugefügt wurden. Collections hingegen enthalten nur Dokumente – hier können keine anderen rohen Daten gespeichert werden. Sie geben der Datenbank eine gewisse Struktur, können jedoch flexibel angelegt werden. Es gibt im Firestore somit kein explizites Schema. Es bietet sich allerdings an, alle Dokumente in einer Collection gleich aufzubauen, um Abfragen einfacher zu gestalten[22].

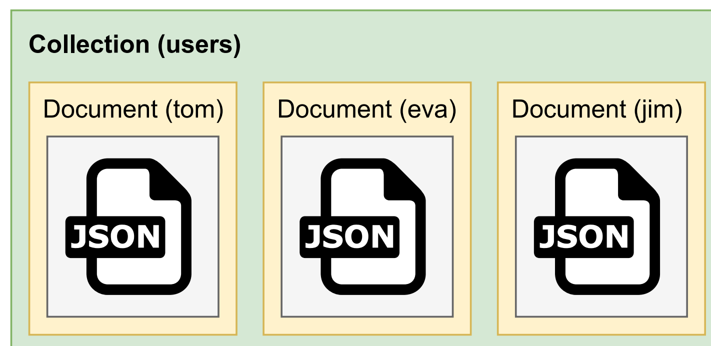


Abbildung 2.7: Datenmodell des Firestores

Jedes Dokument im Firestore kann durch einen eindeutigen Schlüssel referenziert werden. Dieses Prinzip wird in Abbildung 2.7 deutlich. Hier gibt es eine Collection *users*, in der drei Dokumente gespeichert sind: *tom*, *eva* und *jim*. Um nun das Dokument *jim* zu laden, muss die Anwendung den Schlüssel *users/jim* verwenden:

```
1  import { doc } from 'firestore/firestore';
2  const ref = doc(db, 'users/jim');
```

Wichtig dabei ist, dass ein Dokument immer einen Schlüssel bzw. Namen aufweist, dieser aber im Gegensatz zu einer Tabelle in einer SQL Datenbank nicht zwangsläufig im JSON vertreten sein muss. Der Schlüssel wird in der Regel auch nicht durch eine aufsteigende

[21] Vgl. Google, 2022b.
[22] Vgl. Google, 2022a.

Nummer repräsentiert, sondern durch eine beliebige Zeichenkette. Die Anwendung kann diese eigenverantwortlich vergeben oder vom Firestore automatisch generieren lassen[23].

2.2.2 Hierarchische Daten

Der Cloud Firestore unterstützt das Verschachteln von Sammlungen und Dokumenten, um hierarchische Datenstrukturen zu erzeugen. Dokumente können unter sich weitere Sammlungen aufnehmen, in denen wiederum Dokumente gespeichert werden können. Google erlaubt dabei eine Tiefe von 100 Ebenen[24]. So können bspw. persönliche Nutzerdaten in Collections unterhalb eines Nutzerdokuments abgelegt werden, was sowohl die Formulierung von Sicherheitsregeln für Zugriffe als auch Abfragen sehr einfach gestaltet (siehe Abbildung 2.8). Security im Zusammenhang des Firestores wird in Kapitel 2.2.4 erläutert.

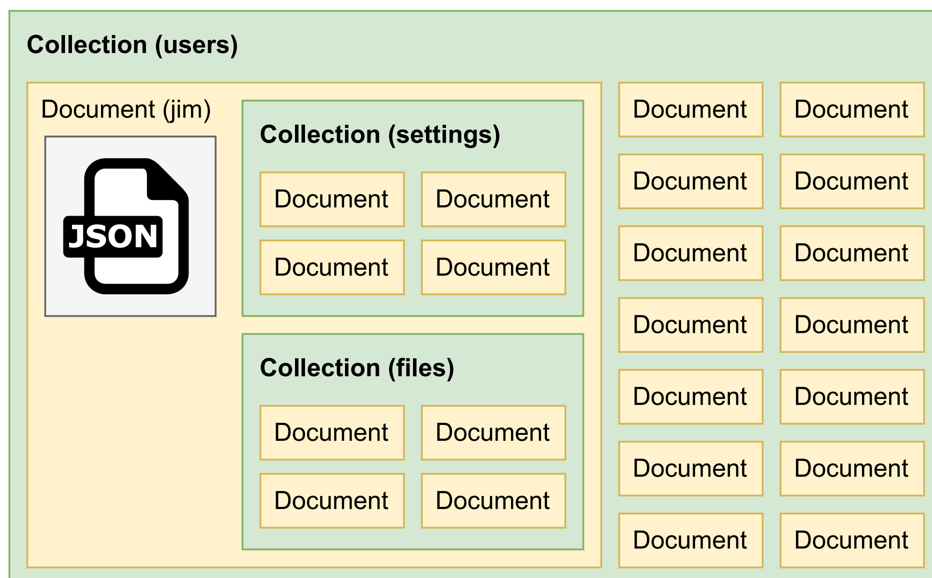


Abbildung 2.8: Hierarchische Daten im Firestore

Um bspw. alle Dateien des angemeldeten Users zu laden, die als Favorit markiert wurden, kann folgendes Codebeispiel dienen:

```
1  import { doc, collection, query, where } from 'firestore/firestore';
2  const userId = ...
3  const ref = collection(db, `users/{userId}/files`);
4  const q = query(ref, where('favorite', '==', true));
```

[23]Vgl. Google, 2022a.
[24]Vgl. Google, 2022a.

2.2.3 Zugriffsoperationen

Beim Cloud Firestore gibt es zwei grundsätzliche Arten von Zugriffsoperationen: Lesende und Schreibende (Reads und Writes). Jede Operation wird von Google in Rechnung gestellt, während das Schreiben[25] mit 0,18 $ pro 100.000 Dokumente teurer ist als das Lesen mit 0,06 $ pro 100.000 Dokumente. Das Löschen von Dokumenten wird hier der Einfachheit halber als schreibende Operation behandelt, in der Abrechnung schlägt diese mit 0,02 $ pro 100.000 Dokumente jedoch deutlich günstiger zu Buche. Seit Oktober 2022 ist es zudem möglich, aggregierende Abfragen wie *count()* zu stellen, wobei hier für jeden angefangenen Block von 1000 Dokumenten nur ein Read abgerechnet wird[26]. Die Client SDKs stellen für jede Operation entsprechende Implementierungen bereit. Präzise Abfragen sind mithilfe von *Queries* möglich, allerdings ist der Firestore nicht für komplexere Anwendungsfälle wie Volltextsuche oder die Eingrenzung von Dokumenten auf Geoinformationen in bestimmten Radien ausgelegt, da entweder die benötigten Abfrageoperatoren fehlen oder die Bildung komplexer Indizes nötig wäre[27].

Lesen in Echtzeit

Abfragen können mithilfe der Client SDKs in Echtzeit formuliert werden. Das bedeutet, dass die Verbindung zur Datenbank so lange geöffnet bleibt, bis die Anwendung diese wieder schließt. Währenddessen wird die Ergebnismenge der Abfrage automatisch mit der Datenbasis im Firestore synchronisiert, sodass Änderungen bspw. eines anderen Clients unverzüglich geladen werden. Dies bietet sich vor allem für Chats oder Liveticker an. Im folgenden Beispiel werden die favorisierten Dateien eines Users in Echtzeit synchronisiert:

```
1  import { doc, collection, query, where, onSnapshot } from 'firestore/firestore';
2  const userId = ...
3  const ref = collection(db, `users/{userId}/files`);
4  const q = query(ref, where('favorite', '==', true));
5  const unsubscribe = onSnapshot(q, (snapshot) => {
6    const files = snapshot.map(doc => doc.data());
7    console.log(files);
8  });
9  ...
10 unsubscribe();
```

[25] Einfügen und Aktualisieren sind synonym.
[26] Vgl. Google, 2023c.
[27] Google verweist hier auf die Verwendung spezialisierter Services.

Schreibende Operationen

Der Firestore unterstützt verschiedene Arten beim Schreiben von Dokumenten. Am einfachsten ist der simple Write, der lediglich ein Dokument in eine Collection schreibt. Folgendes Listing verdeutlicht dieses, indem eine Datei eines Users als Favorit markiert wird:

```
1  import { doc, setDoc } from 'firestore/firestore';
2  const userId = ...
3  const fileId = ...
4  const ref = doc(db, `users/{userId}/files/{fileId}`);
5  await setDoc(ref, { favorite: true }, { merge: true });
```

Beim Aufruf von *setDoc()* muss neben der Referenz auf das Dokument auch der gewünschte Inhalt angegeben werden. Durch den optionalen Schalter *merge: true* ist es möglich, ein evtl. bereits bestehendes Dokument nur zu ergänzen und nicht zu überschreiben. Dies hat den Vorteil, dass ein bestehendes Dokument nicht zunächst geladen, dann angepasst und im Anschluss wieder geschrieben werden muss, da so Kosten und Codezeilen gespart werden. Wenn ein Schlüssel vom Firestore generiert werden soll, kann statt *setDoc addDoc* verwendet werden.

Die zweite Stufe der Writes stellt das sogenannte *Batched Writing* dar. Dabei können schreibende Zugriffe zunächst geplant und dann gesammelt ausgeführt werden.

```
1  import { doc, writeBatch } from 'firestore/firestore';
2  const userId = ...
3  const batch = writeBatch(db);
4  batch.set(doc(db, `users/{userId}/files/46dd7832`), {favorite: false, name: '
     Schreiben.pdf'}, {merge: true});
5  batch.delete(doc(db, `users/{userId}/files/a5e5b0d7`));
6  await batch.commit();
```

Das Listing verdeutlicht dieses Prinzip. Hier wird zunächst eine *Write Batch* erzeugt, diese dann mit geplanten Operationen befüllt und am Ende mit *commit()* ausgeführt. In diesem Beispiel hat dies zur Folge, dass eine Datei umbenannt und der Favoritenstatus entzogen sowie eine weitere Datei gelöscht wird. Der Vorteil liegt in der Atomizität des Vorgangs. Alle Operationen werden zunächst ausgeführt und der Zustand der Datenbank danach angepasst.

Im dritten Schritt bietet der Firestore Transaktionen an. Diese funktionieren vom Prinzip ähnlich wie die Batches, allerdings sind hier auch lesende Zugriffe auf beliebige Dokumente möglich. Des Weiteren führt der Firestore die Operationen auch bei Transaktionen atomar aus. Im folgenden Listing ist das wichtig, denn hier wird eine rudimentäre Geldüberweisung implementiert:

```
import { doc, runTransaction } from 'firestore/firestore';
const orderId = ...;
try {
  runTransaction(db, async (transaction) => {
    const orderRef = doc(db, `order/{orderId}`)
    const sourceAccountRef = doc(db, `account/{order.sourceAccountId}`);
    const targetAccountRef = doc(db, `account/{order.targetAccountId}`);

    const order = (await transaction.get(orderDoc)).data();
    const source = (await transaction.get(sourceAccountRef)).data();
    const target = (await transaction.get(targetAccountRef)).data();

    ... Validation ...

    transaction.set(sourceAccountRef, {
      balance: source.balance - order.amount
    }, {merge: true});

    transaction.set(targetAccountRef, {
      balance: target.balance + order.amount
    }, {merge: true});

    transaction.delete(orderRef);
  });
} catch (e) {
  console.err('Fehler beim Ausführen der Transaktion', e);
}
```

Transaktionen verlangen, dass alle lesenden Zugriffe vor den schreibenden stattfinden, da sonst ein Fehler provoziert wird. Außerdem dürfen gelesene Dokumente sich nicht verändern, bis die Transaktion abgeschlossen ist. Hier kommt allerdings das automatische Wiederholen des Firestores zum Einsatz, sodass der User davon i. d. R. nichts merkt[28].

2.2.4 Security

Durch das breite Angebot von Client SDKs für viele verbreitete Programmiersprachen kann der Cloud Firestore in vielfältigen Umgebungen angebunden werden. Dazu gehören auch clientseitige Anwendungen wie Webseiten oder Fat Clients[29]. Der Service ist global über das Internet erreichbar und erfordert keine Implementierung in ein Mehrschichtenmodell im klassichen Sinne. Stattdessen greifen die Clients in der Regel direkt

[28] Vgl. Google, 2023g.
[29] Ein Programm mit einer grafischen Oberfläche, welches auf einem Desktop installiert wird.

mithilfe eines passenden SDKs auf den Firestore zu (siehe Abbildung 2.9). Um Zugriffe zu autorisieren, bietet Google einen Sicherheitsmechanismus namens *Security Rules* an, der mithilfe einer Konfigurationsdatei gesteuert werden kann. Dies hat den Vorteil, dass kein serverseitiger Code entwickelt werden muss, um den autorisierten Zugriff auf Daten zu gewährleisten[30].

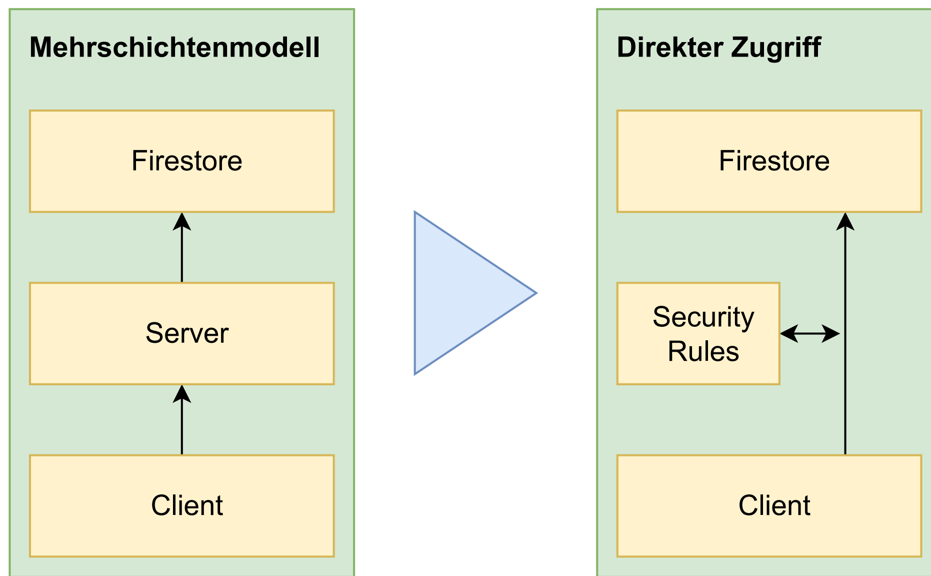


Abbildung 2.9: Schichtenarchitektur mit Firestore

Mithilfe der Sicherheitsregeln kann die Plattform genau bestimmen, ob eine Anfrage an den Firestore erlaubt wird oder nicht. Im Hintergrund wird dabei auch *Firebase Authentication* unterstützt, der Teil der Firebase Plattform, welcher für Authentifizierung zuständig ist. Das folgende Beispiel sichert den Firestore dahingehend ab, dass nur durch *Firebase Authentication* authentifizierte User sowohl lesend als auch schreibend auf die Daten zugreifen dürfen.

```
1  service cloud.firestore {
2    match /databases/{database}/documents {
3      match /{document=**} {
4        allow read, write: if request.auth != null;
5      }
6    }
7  }
```

In der Praxis gestalten sich die Regeln meist deutlich komplexer, sofern die Client SDKs eine breite Verwendung im Projekt finden. Um das Beispiel aus Kapitel 2.2.2 aufzugrei-

[30] Vgl. Google, 2023d.

fen, könnte die Konfiguration in diesem Fall wie folgt aussehen:

```
1  service cloud.firestore {
2    match /databases/{database}/documents {
3      function isUser(userId) {
4        return request.auth != null && request.auth.uid == userId;
5      }
6
7      match /users/{userId} {
8        allow read, write: if isUser(userId);
9
10       match /{document=**} {
11         allow read, write: if isUser(userId);
12       }
13     }
14
15     match /{document=**} {
16       allow read, write: if false;
17     }
18   }
19 }
```

Dieser Code lässt Lese- und Schreibzugriffe nur zu, wenn es sich um Dokumente in der Collection *users* inkl. aller Subcollections handelt und der authentifizierte User der Anfrage der selbe ist, der im Dokumentenpfad hinterlegt ist. Hier zeigt sich auch, dass Codeduplizierung vorgebeugt werden kann, indem gemeinsam genutzte Schnipsel in Funktionen wie *isUser()* ausgelagert werden[31]. Die Engine lässt auch lesende Zugriffe auf Daten beliebiger Dokumente zu, um bspw. Validierung zu implementieren. Dabei muss jedoch beachtet werden, dass durch Sicherheitsregeln ausgelöste Operationen ebenso abgerechnet werden wie sonst.

2.2.5 Varianten zur Verwendung von Collections

Wie beschrieben basiert der Cloud Firestore auf der Vereinigung aus Sammlungen und Dokumenten. Jede lesende und schreibende Operation wird von Google abgerechnet und Dokumente sind sowohl auf 1 MB als auch auf 20.000 Attribute beschränkt. Diese und weitere Einschränkungen spannen einen Handlungsraum auf, in dem sich eine Anwendung bewegen muss, um den Firestore erfolgreich einsetzen zu können. Die technischen Möglichkeiten lassen laut Google verschiedene Ansätze zur Strukturierung von Daten zu, die je nach Anwendungsfall unterschiedlich geeignet sein können und im Folgenden vorgestellt werden.

[31] Vgl. Google, 2023e.

Verschachtelte Daten in einem Dokument

Der einfachste Ansatz, um zueinander in Beziehung stehende Daten zu modelieren, ist die Verwendung eines einzigen Dokuments. Dabei werden neben den Standardattributen auch die abhängigen Daten mithilfe eines Arrays im selben Dokument gehalten[32]. Das folgende Listing zeigt ein beispielhaftes Dokument, welches nach diesem Ansatz strukturiert ist:

```
1 // employees/jim
2 {
3   name: "Jim",
4   department: "Sales",
5   locations: ["Scranton", "Stamford"]
6 }
```

Der Vorteil dabei ist die Simplizität der Methode, da weder zusätzliche Collections noch komplexer Anwendungs- und Abfragecode benötigt wird. Darüberhinaus ist sie äußerst kosteneffizient, da nur ein Read für alle Daten nötig ist. Das Vorgehen bietet sich an, wenn die Arrays kurz und einfach gehalten sind und die Daten mehr der Anzeige denn der Abfrage dienen, denn die Verwendung des *array-contains* bzw. *array-contains-any* Operators wird durch den Firestore streng limitiert. Aufgrund der Beschränkung auf 1 MB pro Dokument skaliert die Methode zudem nur zu einem gewissen Punkt, da so die Performance beim Lesen leiden könnte sowie die relationalen Daten in ihrer Quantität beschränkt werden[33].

Verwendung von Subcollections

Um das Problem der schwachen Skalierung der ersten Methode zu lösen, können Subcollections verwendet werden. Dabei werden wie in Kapitel 2.2.2 beschrieben Dokumente in dedizierten Sammlungen unterhalb eines Elterndokuments abgelegt. Diese Methode bietet sich für Anwendungsfälle an, die eine streng eingeschränkte Sicht auf Dokumente im Kontext einer Elternentität erfordern. Subcollections können vollwertige Dokumente aufnehmen, sodass die Beschränkung auf simple Arrays entfällt. Die Abfragelogik kann hier ähnlich simpel bleiben, bietet bei Bedarf jedoch die Flexibilität für komplexere Filtermöglichkeiten bspw. für die Subcollections. Allerdings verwirkt dieses Vorgehen im Vergleich seine Kosteneffizienz, da für das Lesen aller Daten im Prinzip immer $1 + n$ Reads erforderlich sind. In der Anwendungsentwicklung kann dieser Faktor jedoch durch die Verwendung intelligenter Paginierungsmechaniken limitiert werden. Eine smarte und

[32] Vgl. Google, 2023b.
[33] Vgl. Google, 2023f.

kosteneffiziente Implementierung erfordert somit mehr Aufwand als die erste Methode, bietet jedoch mehr Flexibilität in der Gestaltung relationaler Daten und eignet sich besonders dann, wenn verschachtelte Dokumente eine gewisse Vollwertigkeit aufweisen[34]. Das Beispiel aus der vorherigen Anwendung würde damit so formuliert werden können:

```
1   // locations/scranton
2   {
3     name: "Scranton",
4     address: { ... },
5     employees: 12
6   }
7   // locations/stamford
8   {
9     name: "Stamford",
10    address: { ... },
11    employees: 37
12  }
13  // employees/jim
14  {
15    name: "Jim",
16    department: "Sales",
17  }
18  // employees/jim/employee-locations/scranton
19  {
20    name: "Scranton",
21    workdays: ["Monday", "Tuesday"]
22  }
23  // employees/jim/employee-locations/stamford
24  {
25    name: "Stamford",
26    workdays: ["Wednesday", "Thursday", "Friday"]
27  }
```

In diesem Fall gibt es die Orte Scranton und Stamford in einer eigenen Root Collection, wobei dort verschiedene allgemeine Daten wie die Adresse und die Anzahl der dort tätigen Angestellten gespeichert sind. Das Dokument des Mitarbeiters *jim* wurde um die Locations verschlankt, da diese in eine Subcollection unterhalb des Dokuments ausgelagert wurden. Hier zeigt sich, dass im Firestore häufig mit denormalisierten Daten gearbeitet wird, denn der Name jedes Orts wird sowohl im Stammdatensatz als auch in der Subcollection gespeichert, die anderen Felder jedoch nicht. Für den Anwendungsfall wird angenommen, dass in der Anzeige des Mitarbeiters lediglich der Name der Tätigkeitsstätten relevant ist. Weitere Informationen sind erst dann wichtig, wenn der Ort im Detail geöffnet wird. Die Denormalisierung spart hier einen weiteren Read

[34] Vgl. Google, 2023b.

auf die Stammdaten pro Location, was sich positiv auf Kosten und Performance auswirken kann. Des Weiteren weisen die Dokumente in der Subcollection auch weitere personenbezogene Daten wie Anwesenheitspläne für Arbeitstage auf. Die Zuordnung zwischen Sub- und Root Collection wird in diesem Fall über den Pfad der Dokumente (bspw. *.../scranton*) vorgenommen.

Verwendung von Root Collections

Die dritte Vorgehensweise zur Strukturierung von in Beziehung stehender Daten ist die reine Verwendung von bzw. Fokussierung auf Root Collections. Dies ist besonders nützlich, wenn die Daten aus verschiedenen Perspektiven und Kontexten betrachtet werden sollen, ohne dass komplexe Synchronisierungslogiken zur Denormalisierung notwendig sind[35]. Diese Methode ist der Verwendung einer klassischen relationalen Datenbank am ähnlichsten und gerade für Novizen im Umgang mit dem Firestore am intuitivsten. Der Usecase aus dem vorherigen Beispiel kann damit wie folgt abgebildet werden:

```
1  // locations/scranton
2  {
3    name: "Scranton",
4    address: { ... },
5    employees: 12
6  }
7  // locations/stamford
8  {
9    name: "Stamford",
10   address: { ... },
11   employees: 37
12 }
13 // employees/jim
14 {
15   name: "Jim",
16   department: "Sales",
17 }
18 // employee-locations/jim-scranton
19 {
20   locationId: "scranton",
21   employeeId: "jim",
22   workdays: ["Monday", "Tuesday"]
23 }
24 // employee-locations/jim-stamford
25 {
26   locationId: "stamford",
27   employeeId: "jim",
```

[35] Vgl. Google, 2023b.

```
28      workdays: ["Wednesday", "Thursday", "Friday"]
29    }
```

Hier sind die Dokumente unterhalb des Mitarbeiters in eine Root Collection gewandert und wurden wie bei einer klassischen Fremdschlüsselbeziehung um die Identifier der referenzierten Daten ergänzt. Nun kann die Anwendung sowohl aus Sicht eines Mitarbeitenden als auch einer Location Abfragen formulieren. So können bspw. alle Zuordnungen und damit im Anschluss auch alle Stammdatensätze der entsprechenden Location geladen werden, deren *employeeId = „jim“* lauten. Allerdings sind in diesem Fall mehr Reads notwendig, da die Zuordnungsdokumente keine denormalisierten Daten aufweisen. Das Vorgehen schließt Denormalisierung nicht grundsätzlich aus, allerdings muss bewusst entschieden werden, welche Informationen in der Zuordnung zusätzlich gespeichert werden, da der Datenfluss aus mehreren Richtungen (*Location* und *Employee*) erfolgen kann. Bei unbedachter Anwendung kann dies die Komplexität im Vergleich zur vorherigen Methode deutlich steigern, da hier nicht klar definiert ist, wem die Zuordnung „gehört“ bzw. welche Entity dafür hauptverantwortlich ist.

Einordnung

Jedes Vorgehen weist Vor- und Nachteile auf und dessen Eignung hängt stark vom konkreten Anwendungsfall ab. Laut Google ist die wichtigste Entscheidungsgrundlage, dass sich die Eigenschaften der zu speichernden Daten sowie die Art der Interaktion mit diesen bewusst gemacht werden, da sich aus diesen Erkenntnissen häufig intuitiv die passende Methode ableitet.

2.2.6 Fortgeschrittene Techniken zur Strukturierung von Daten

Die zuvor vorgestellten Varianten in der Verwendung des Firestores können mithilfe fortgeschrittener Techniken zur Strukturierung von Daten in nicht relationalen Datenbanken kombiniert werden. Diese reichen von *Denormalization* über *Aggregates* hin zu *Composite Key*. Bei der Denormalisierung werden abhängige Daten, die über Fremdschlüsselwerte hinausgehen, an Dokumente geschrieben, sodass Daten für die Anzeige nicht aus mehreren Quellen gelesen werden müssen. Dieses Vorgehen beschleunigt Lesevorgänge, bremst Schreiboperationen jedoch aus[36]. Aggregationen sind nützlich, um bspw. komplexe Berechnungen abzuspeichern, um diese effizient und performant anzuzeigen[37]. Ein *Composite Key* erleichtert die Abbildung von Relationen verschiedener

[36]Vgl. Ilya Katsov, 2012, (1).
[37]Vgl. Ilya Katsov, 2012, (2).

Dokumente, indem diese bspw. unter dem Schlüssel *employeeId_departmentId* abgelegt werden[38]. Katsov beschreibt zudem, dass *Application Side Joins* im Bereich der nicht relationalen Datenbanken meist nicht unterstützt werden, sondern auf der Designebene stattfinden sollten[39]. Dies ist auch beim Cloud Firestore der Fall. Überraschenderweise ist die Kombination dieser Techniken mit dem Firestore vergleichsweise unbearbeitet, da lediglich einige Blogartikel und Tutorials im Netz existieren[40]. Darüberhinaus wurde mithilfe von Aggregationen ein Bug einer Crowdfunding Plattform behoben, der innerhalb von 72 Stunden eine Rechnung von über 30.000 $ erzeugt hat[41].

2.3 Aufstellung von Hypothesen

Die Nullhypothese (H0) besagt, dass alle genannten Faktoren unabhängig von der gewählten Art des Umgangs mit der Datenbank sind und somit keine Veränderungen in Bezug auf Kosten, Performance und Komplexität zu erwarten sind. Unter Berücksichtigung der aktuellen Forschung werden hier nun drei Hypothesen aufgestellt, die konträr zu H0 stehen und im Rahmen dieser Arbeit überprüft werden.

1. Wenn eine stark kostenfokussierte Optimierung vorgenommen wird, leiden wahrgenommene Performance und Wartbarkeit der Anwendung darunter.
2. Ein rein auf relationalen Datenbanken basiertes Mindset schadet leseintensiven Anwendungsfällen signifikant.
3. Je stärker der Fokus auf die Optimierung von Reads gelegt wird, desto schwächer ist die Write Performance.

[38] Vgl. Ilya Katsov, 2012, (8), (9).
[39] Vgl. Ilya Katsov, 2012, (3).
[40] Vgl. Jeff Delaney, 2018.
[41] Vgl. Nicolás Contreras V., 2018.

Kapitel 3

Methodik

Wie in Kapitel 2 bereits herausgearbeitet, wird die effiziente Verwendung des Cloud Firestores im Netz zwar punktuell diskutiert, allerdings wurde das Thema bisher nicht in einem methodischen Kontext bearbeitet. Vor dem Hintergrund, dass es sich hier um ein sehr technikzentriertes Thema handelt, bietet sich experimentelle Forschung an. Im Rahmen dieser Arbeit wurde also ein Experiment zur Gewinnung neuer Erkenntnisse durchgeführt. Im Folgenden werden der Versuchsaufbau, die Datenerhebung sowie die Aufbereitung der Daten beschrieben.

3.1 Versuchsaufbau

In Kapitel 1.1 wurde bereits die prototypische Implementierung von Firestore Apps genannt, die zwar auf den Cloud Firestore aufsetzt, diesen allerdings auf eine eher relationale Art und Weise verwendet. Ursprünglich sollte eine solche Anwendung als Testsubjekt für das Experiment fungieren. Diese Idee wurde jedoch verworfen, da die gewonnenen Daten aufgrund für den Cloud Firestore irrelevanten Features und dem damit einhergehenden Overhead sowie unverhältnismäßiger Anpassungsaufwände verwässert worden wären. Daher wurde eine prototypische Software zur Verwaltung von Raumbuchungen implementiert, welche sich auf den Aspekt Cloud Firestore fokussiert, jedoch keinen Wert bspw. auf Design legt. Der große Vorteil des Prototypen gegenüber einer vollwertigen App ist, dass verschiedene Arten, die Daten aufzubereiten und zu strukturieren, exemplarisch sehr einfach implementiert werden konnten. Die Anpassung hätte einen einschneidenden Konzeptwechsel bedeutet, der im Rahmen dieser Arbeit nicht zu leisten gewesen wäre. Technisch basiert der Prototyp auf dem beliebten Web Application Framework *Angular*. Es handelt sich also um eine Single Page Applicati-

on (SPA), welche in einem Webbrowser läuft, moderne Standards implementiert und zahlreiche Funktionen bspw. für die Kommunikation mit Services wie der Google Cloud Platform (GCP) bietet. Zudem nutzt der Prototyp natürlich den Cloud Firestore sowie Google Cloud Functions für serverlose Backendroutinen. Tabelle 3.1 listet die Versionsnummern der relevanten Frameworks und Technologien auf, welche im Rahmen des Experiments zum Einsatz kamen.

Tabelle 3.1: Versionsnummern der verwendeten Frameworks und Technologien

Komponente	*Version*
Angular	15.2
Angular Fire	7.5
Cypress	12.11
Firebase SDK	v8
Firebase Functions	4.3
Typescript	4.9

Das zentrale Datenobjekt des Prototyps ist die Reservierung. Diese bezieht sich immer auf einen User und einen Platz. Ein Platz befindet sich wiederum stets in einem Raum bzw. Büro. Mehrere Büros sind am Ende einem Gebäude zugeordnet. Es gibt also ein vierstufiges Konzept abhängiger Daten. Der Prototyp definiert daher die vier Entities *Reservation*, *Seat*, *Room* und *Compound* sowie den *User*. Datengetriebene Anwendungen weisen typischerweise Listenansichten auf, welche in der Darstellung zwar variieren können, meist jedoch listenbasiert sind. Im Kern werden stets Reservierungen inkl. der abhängigen Daten wie User, Platz, Raum und Gebäude angezeigt, allerdings wechselt gerne die Perspektive, aus der diese betrachtet werden. Die Anwendung definiert drei Anwendungsfälle, die Reservierungen unterschiedlich betrachten. Diese wurden genau so für den Prototypen vorgesehen und sind in Tabelle 3.2 beschrieben.

Tabelle 3.2: Übersicht der Perspektiven

Perspektive	*Beschreibung*
User Based	*Reservations* aus Sicht eines einzelnen *Users*
Time Based	*Reservations* für einen einzelnen Zeitraum (1 Monat)
Location Based	*Reservations* aus Sicht eines einzelnen *Compounds*

Jede Perspektive listet im Kern Reservierungen auf und stellt diese in einer simplen Listenform dar. Der Prototyp sieht zudem Interaktionen mit den Daten vor, bei denen diese aktualisiert werden (bspw. durch eine Namensänderung). Dazu gibt es an jedem Listeneintrag verschiedene Buttons, um die entsprechende Funktion auszulösen. Konkret können sowohl die *Reservation* als auch die zugehörigen Objekte *Seat*, *Room* sowie *Compound* aktualisiert werden.

Um die verschiedenen Techniken aus Kapitel 2.2.5 und 2.2.6 zu repräsentieren, wurden vier Szenarien definiert und die Anwendung entsprechend strukturiert. Eine Übersicht der Szenarien findet sich in Tabelle 3.3.

Tabelle 3.3: Übersicht der Szenarien des Experiments

Szenario	*Beschreibung*
Pseudo Relational	Implementierung auf Basis des relationalen Mindsets
Denormalized	Erweiterung von Pseudo Relational durch denormalisierte Daten
Shared View	Implementierung eines Bucketing Verfahrens
Dedicated View	Vollständige Aggregation der Daten auf Basis der zuvor definierten Perspektiven

Auch wenn es sich bei der Angular App des Prototypen technisch um einen Monolithen handelt und die Cloud Functions in einem einzigen GCP Projekt organisiert werden, sind alle relevanten Codeteile strikt nach den Szenarien getrennt, da nur so die Mess- und Vergleichbarkeit sichergestellt werden kann. In der Praxis bedeutet das, dass bspw. die Masken für das Szenario *Shared View* in der Angular App unter *http://localhost:4300/scenario-shared-view* zu finden sind, während die anderen Szenarien wiederum eigene Routen haben. Jedes Szenario stellt die drei zuvor genannten Perspektiven auf Reservierungen in exakt der gleichen Präsentation bereit. Lediglich die interne Abfrage- und Interaktionslogik mit den Daten unterscheidet sich zwischen den Szenarien.

3.1.1 Szenario: Pseudo Relational

Dieses Szenario kann als Baseline angesehen werden, mit der die anderen Szenarien grundsätzlich verglichen wurden, da es die unoptimierte Form der bestehenden Anwendung mit einem relationalen Mindset darstellt. Jede Entity wird in einer eigenen Root Collection gespeichert und beinhaltet einen entsprechenden Fremdschlüssel auf das jeweils nächstgelegene Datenobjekt in der o. g. Kette. Diese Art der Speicherung erfordert einen *Application Side Join* zur Abfragezeit, um alle benötigten Informationen für eine

Reservierung anzuzeigen. Die Angular App liest die Daten für alle Perspektiven, indem zunächst die Reservierungen und auf Basis der IDs die verknüpften Plätze abgefragt werden. Diese Logik zieht sich einmal durch die gesamte Datenkette von der *Reservation* hin zum *Compound*. Im letzten Schritt werden aus den geladenen Dokumenten für die Anzeige passende Objekte erstellt.

Erwähnenswert ist dabei die Tatsache, dass für die Perspektive *Location Based* keine Filterung via Firestore Query vorgenommen werden kann, da *compoundId* kein Feld einer *Reservation* ist. Daher müssen hier alle Reservierungen ungefiltert aus dem Firestore geladen werden und die Einschränkung auf den gewünschten *Compound* kann erst im Nachhinein zur Anzeige geschehen.

3.1.2 Szenario: Denormalized

Kern dieses Szenarios ist die Denormalisierung der im vorherigen Szenario mit einem *Application Side Join* herangezogenen Daten. Alle Dokumente in der Collection *Reservation* weisen daher die zur Anzeige im Browser benötigten Properties auf. Da die Daten nicht automatisch dort hinzugefügt werden, benötigt das System in diesem Szenario erstmals einige Firestore Trigger in Form von Cloud Functions zur Synchronisierung.

Ein Trigger wird immer dann aufgerufen, wenn ein Dokument in der Collection *Reservation* geschrieben wird. Die Function traversiert die gesamte Datenkette vom *Seat* über dessen *Room* zum finalen *Compound*. Die IDs sowie Namen der Objekte werden daraufhin an der *Reservation* vermerkt und geprüft, ob die denormalisierten Daten sich verändert haben, um einen Cloud Overflow[1] zu verhindern. Wenn die Daten sich tatsächlich geändert haben, wird das Dokument im Anschluss gespeichert.

Das Szenario sieht zudem Trigger für Write Events auf Dokumenten in der o. g. Datenkette vor, die alle zugehörigen Reservierungen laden und etwaige geänderte Properties wie den Namen dort aktualisieren. Je nachdem wie „weit“ das Ojekt von der *Reservation* entfernt ist, verarbeitet die Funktion zur Laufzeit mehr oder weniger Daten. Auf diese Weise werden aktualisierte Daten stets an die benötigten Stellen synchronisiert.

Das Auflisten der Reservierungen im Browser fällt vergleichsweise simpel aus, da nun nur noch die Collection *Reservation* abgefragt werden muss. Der o. g. *Application Side Join* entfällt ersatzlos.

[1] Eine unverhältnismäßig intensive Nutzung bezahlter Ressourcen in der Cloud, die bspw. aus einem simplen Programmfehler entsteht, der eine Endlosschleife entstehen lässt.

3.1.3 Szenario: Shared View

Shared View ist das erste der zwei Szenarien, welche zusätzliche Collections für die Anzeige im Browser einführen. Hier wird zusätzlich zur *Denormalization* das *Bucketing* genutzt, um Reservierungen nach definierten Kriterien zu clustern. Das Verfahren stellt eine Erweiterung zu *Denormalized* dar, allerdings erfolgt zusätzlich die Erzeugung bzw. Aktualisierung von *View* Objekten. Die Denormalisierung wurde beibehalten, um den Daten- und Kontrollfluss der Trigger übersichtlicher zu strukturieren, da auf diese Weise nur eine einzige Function für die View Reconciliation verantwortlich ist.

Da die Anwendung die Reservierungen aus drei Perspektiven betrachten kann (*User Based*, *Time Based* und *Location Based*), kann jede View mithilfe des folgenden Schemas identifiziert werden:

```
1   db.doc(`scenario/shared-view/view/${year}-${month}-${userId}-${compoundId}`)
```

Die Reconciler Function sorgt dafür, dass für jede Kombination der drei unabhängigen Variablen maximal ein Dokument in der Collection existiert, welches die zugehörigen Reservierungen in einem Array sammelt. Die Browseranwendung ist somit in der Lage, je nach Perspektive eine simple Abfrage zu definieren (bspw. alle Views für einen bestimmten Nutzer) und dann eine überschaubare Anzahl an Dokumenten aus dem Firestore zu lesen, während diese jedoch potenziell deutlich mehr Reservierungen beinhalten.

3.1.4 Szenario: Dedicated View

Das vierte Szenario stellt eine Spezialisierung von *Shared View* dar, bei der das *Bucketing* durch *Aggregation* ersetzt wird. Ziel ist, für jede Perspektive passende Views zu erzeugen, sodass der Browser nur das passende Dokument lesen muss, um die relevanten Daten anzuzeigen. Technisch ähnelt die Methode dem Szenario *Shared View*, da auch hier Reconciler Functions die Reservierungen clustern. Der Unterschied liegt darin, dass es pro Perspektive eine dedizierte Collection in der Datenbank gibt (siehe Tabelle 3.4), anstatt alle Views in einer einzigen Sammlung zu organisieren.

Tabelle 3.4: View Collections für Szenario *Dedicated View*

Collection	*Beschreibung*
.../views/user-based/user	Aufbereitung der Reservierungen pro Nutzer nach Monat
.../views/time-based/year-month	Alle Reservierungen pro Monat unabhängig von anderen Faktoren
.../views/location-based/compound	Gruppiert die Reservierungen eines Compounds nach Monat

3.2 Datenerhebung

Im Folgenden wird der Prozess der Datenerhebung beschrieben. Die Arbeit geht dabei zunächst auf die technische Implementierung der verwendeten Messinstrumente ein und beleuchtet im Anschluss die Art und Weise, wie die Daten erhoben wurden.

3.2.1 Messinstrumente für Kosten und Performance

Der Versuchsaufbau stellt eine reale Anwendung dar, die Zugriffe auf Cloud Functions und Cloud Firestore durchführt, während mit ihr interagiert wird. Um diese Zugriffe zu beziffern, wurde das System mit diversen Messinstrumenten ausgestattet. Technisch wurde dafür ein minimaler Logger Server mit Node.js und dem Express Framework implementiert, der entsprechende Endpunkte via HTTP anbietet. Diese schreiben sogenannte Logger Events in eine PostgreSQL Datenbank. Die Browseranwendung sowie die Cloud Functions sprechen den Loggerserver an, wenn zu loggende Events vorliegen. Die Vorgehensweise, mit HTTP Endpunkten zu arbeiten, ist vorteilhaft, da die Pflege der PostgreSQL Datenbank an einer zentralen Stelle erfolgt und nicht auf Client- und Serverkomponenten aufgeteilt wird. Die unterstützten Log Events werden im Folgenden beschrieben.

Logger Event: Firestore Log

Dieses Event beinhaltet Informationen über Zugriffe auf Firestore Collections. Es entspricht folgender Typescript Definition:

```
1  interface FirestoreLog {
2    createdAt: Date;
3    firestoreCollection: string;
```

```
4     op: string;
5     size: number;
6     user: string;
7     source: string;
8     scenario: string;
9   }
```

Das System wurde so programmiert, dass alle lesenden und schreibenden Zugriffe auf den Firestore zentral über den Logger Server dokumentiert werden. Die *size* gibt dabei die Anzahl der Operationen auf der angegebenen Collection an. Das Feld speist sich stets aus der Anzahl der gelesenen bzw. geschriebenen Dokumente. An dieser Stelle muss erwähnt werden, dass Google angibt, erneute Reads bei Realtime Updates für Abfragen zu optimieren, sodass bspw. eine Query, die zehn Dokumente liefert, bei einem Update eines der enthaltenen Dokumente u. U. nur einen weiteren Read statt zehn erzeugt[2]. Da es sich dabei um eine interne Optimierung seitens Google handelt und diese Mechanik technisch nicht von außen sicht- sowie messbar ist, ignoriert der Versuchsaufbau diese Implikationen für Realtime Updates. Auch etwaiger Offline Support sowie Fehlersituationen bei Optimistic Writes wurden nicht berücksichtigt.

Logger Event: Function Call

Das Function Call Event enthält Daten über ausgeführte Cloud Functions. Das zugehörige Typescript Interface lautet wie folgt:

```
1   interface FunctionCall {
2     createdAt: Date;
3     functionName: string;
4     start: Date;
5     end: Date;
6     args: string;
7     user: string;
8     scenario: string;
9   }
```

Jede Cloud Function spricht am Ende ihrer Laufzeit den Logger Server an und übermittelt die entsprechenden Informationen über Start- und Endzeitpunkt. Das Feld *args* entspricht dabei den übergebenen Parametern in Form von JSON.

[2]Vgl. Google, 2023a.

Logger Event: Performance

Immer, wenn die Browseranwendung die Liste der Reservierung der aufgerufenen Perspektive beim Firestore anfragt, merkt sich die Komponente den aktuellen Zeitpunkt. Sobald die Daten erfolgreich geladen wurden und somit zur Anzeige bereit sind, wird ein Event geloggt, welches gemeinsam mit dem nun aktuellen Zeitstempel Auskunft über die Laufzeit des Ladevorgangs gibt. Die Typescript Repräsentation des Events sieht wie folgt aus:

```
1  interface Performance {
2    start: Date;
3    end: Date;
4    scenario: string;
5    user: string;
6    source: string;
7    scenario: string;
8  }
```

Der Vorteil des Events ist, dass so über alle Szenarien einheitliche und somit vergleichbare Zeitintervalle ohne störenden Overhead wie Rendertime gemessen werden.

Logger Event: Suite

Dieses Event stellt die Auflistung einer Menge definierter Messzeiträume dar. Die Informationen dienen in der folgenden Datenauswertung der Abgrenzung bzw. Segmentierung von Firestore Logs, Function Calls und Performance Events. Die Schnittstellenbeschreibung lautet wie folgt:

```
1  interface Test {
2    name: string;
3    start: Date;
4    end: Date;
5    state: 'passed' | 'failed';
6  }
7
8  interface Suite {
9    name: string;
10   start: Date;
11   end: Date;
12   runs: number;
13   passes: number;
14   failures: number;
15   tests: Test[];
16 }
```

3.2.2 Messinstrumente für Komplexität und Codemetriken

Zur Erfassung von Metriken über die Beschaffenheit des Quellcodes und dessen Komplexität wurde der Cloud Service des Open Source Tools *SonarQube* verwendet. Dabei handelt es sich um eine Plattform für statische Codeanalyse, mit dessen Hilfe Informationen bspw. über Codeduplizierung, Testabdeckung und potenzielle Fehler eines Projekts gewonnen werden können. Im Zuge dieser Arbeit wurden die Codebestandteile der vier Szenarien getrennt voneinander analysiert, sodass Aussagen über die Komplexität der Szenarien getroffen werden können. Eigener Implementierungsaufwand war an dieser Stelle nicht notwendig, da der Entwickler SonarSource einen Scanner anbietet, der automatisch die verwendeten Programmiersprachen erkennt und die Ergebnisse in einer hauseigenen SaaS Lösung aufbereitet.

3.2.3 Strukturierung und Durchführung der Messungen

Um die Vergleichbarkeit zwischen den Szenarien zu gewährleisten, wurde mithilfe einer Cloud Function ein initialer Datenbestand erzeugt. Die Intention war eine möglichst realistische Verteilung einer hinreichenden Menge Reservierungen auf die verschiedenen Entities der Kette. Der dafür implementierte Datengenerator erzeugte die in Tabelle 3.5 aufgelisteten Mengen in von den Szenarien abgetrennnten Collections des Firestores, um die Statistiken nicht zu verfälschen.

Tabelle 3.5: Initialer Datenbestand

Anzahl	*Entity*
13.184	Reservation
659	Seat
133	Room
8	Compound
50	User

Die Datenerhebung teilte sich in zwei Phasen auf. Während der ersten Phase wurden die Reservierungen des initialen Datenbestand mit den Faktoren x1, x2, x5 und x10 in die jeweiligen Zieldatenbestände der Szenarien überführt. Ziel der Simulation war die Gewinnung von Erkenntnissen über Aufwand, Kosten und Skalierbarkeit des langfristigen Einsatzes des entsprechenden Szenarios. Die Performance war an dieser Stelle weniger entscheidend, da ein langfristiger Zeitraum berechnet wurde, ohne dass wäh-

renddessen ein etwaiger Benutzer im Browser die Daten betrachtet hätte. Jene Daten wurden erst in der zweiten Phase des Experiments erhoben, während der automatisierte Browsertests auf dem Datenbestand mit dem Faktor x1 mit der Angular App interagierten. Technisch kam dabei das *Cypress* Framework zum Einsatz, welches strukturierte Browsertests in JavaScript definieren kann. Die ausgeführten Suiten bzw. Tests sind in Tabelle 3.6 aufgelistet. Alle dort definierten Suiten wurden für jedes Szenario einmal durchgeführt, sodass insgesamt 1200 lesende sowie 16 schreibende Testergebnisse vorliegen. Grundsätzlich können die Tests in die zwei Gruppen *Reads* und *Writes* unterteilt werden. Während die lesenden Tests sehr simpel und kurzlebig sind, da hier lediglich einmal die entsprechende Perspektive in der Anwendung aufgerufen und gewartet wird, bis alle Daten geladen sind, wurden hier viele Iterationen gewählt. Auf diese Weise werden Skaleneffekte sichtbarer. Die schreibenden Tests wurden je Datenobjekt nur einmal ausgeführt, da hier im Hintergrund in Abhängigkeit des Anwendungsfalls sehr viele Operationen durchgeführt sowie entsprechend lange Laufzeiten generiert und so mehr als ausreichende Datenmengen zum Analysieren erzeugt werden. Die Durchläufe fanden zwischen dem 16.05.2023 und dem 27.05.2023 sowie dem 22.06.2023 und dem 24.06.2023 stets unter den selben Bedingungen hinsichtlich Rechnerleistung, etc. statt. Zur besseren Vergleichbarkeit wurde dabei der Firestore Emulator verwendet, mit dessen Hilfe die Infrastruktur simuliert werden konnte. Das Arbeitsgerät war ein Apple MacBook Pro 14 Zoll von 2021 mit M1 Pro SoC und macOS Ventura 13.3.1. Zusammenfassend ergeben sich also die drei Ergebnistöpfe *Datageneration*, *Reads* und *Writes*.

Tabelle 3.6: Test Suiten in Phase 2

Suite	*Test*	*Beschreibung*	*Iterationen*
reads	user-based	Ruft die Perspektive *user* auf	100
reads	time-based	Ruft die Perspektive *time* auf	100
reads	location-based	Ruft die Perspektive *location* auf	100
writes	write-reservation	Aktualisiert eine *Reservation*	1
writes	write-seat	Aktualisiert einen *Seat*	1
writes	write-room	Aktualisiert einen *Room*	1
writes	write-compound	Aktualisiert einen *Compound*	1

3.3 Datenaufbereitung

Während der Datenerhebung wurden in der Logger Datenbank Zeilen im zweistelligen Millionenbereich erzeugt. Zur Strukturierung und Aufbereitung dieser Datenmenge wurden Postgres *Materialized Views* verwendet, um relevante Informationen auszuarbeiten. Diese bezogen sich stets auf einen der drei in Kapitel 3.2.3 genannten Ergebnistöpfe. Die folgenden Abschnitte listen die einzelnen Views und ihre Zwecke auf.

3.3.1 Validierung der Ergebnisse

Im Folgenden werden die Mechaniken zur Validierung der gemessenen Daten beleuchtet. Technisch kamen dabei Views in der Logger Datenbank zum Einsatz, die verschiedene Bedingungen prüften und somit die Konsistenz des Datenbestands sicherstellten.

failed_state

Diese View lieferte die Anzahl der fehlgeschlagenen Cypress Tests. Das erwartete Ergebnis betrug 0, da fehlerhafte Tests ein Indiz für die inkorrekte Erfassung von Daten darstellen. Das gemessene Ergebnis entsprach dem erwarteten und erfüllte somit die gewünschte Qualität.

functioncalls_without_scenario

Hierbei handelte es sich um eine Auswertung der Cloud Function Calls, denen kein Szenario zugeordnet werden konnte. Ausgenommen davon waren Hilfsfunktionen zum Ausgeben von Statistiken sowie die Generatoren zur Herstellung initialer Datenbestände. Das erwartete Ergebnis von 0 Zeilen wurde erwartungsgemäß erzielt.

ops_with_wrong_scenario

Die View *ops_with_wrong_scenario* wertete alle Firestore Operationen aus, bei denen eine Diskrepanz zwischen der Collection und dem zu diesem Zeitpunkt im Rahmen eines Tests untersuchten Szenarios vorlag. Auch hier wich das tatsächliche Ergebnis nicht von der erwarteten Anzahl 0 ab.

performanceevents_quantities

Diese Check View listete die Anzahl der Performance Events gruppiert nach der URL auf, die im Rahmen der Read Tests registriert wurden. Da für jede Perspektive jeweils 100 Tests durchgeführt wurden und je Test ein Event erzeugt werden sollte, wurden hier stets 100 Events pro URL erwartet. Das gemessene Ergebnis entsprach der erwarteten Menge (siehe Tabelle 3.7).

Tabelle 3.7: Anzahl der Performance Events gruppiert nach URL

Anzahl	*URL*
100	/scenario-dedicated-view/location-based
100	/scenario-dedicated-view/time-based
100	/scenario-dedicated-view/user-based
100	/scenario-pseudo-relational-denormalized/location-based
100	/scenario-pseudo-relational-denormalized/time-based
100	/scenario-pseudo-relational-denormalized/user-based
100	/scenario-pseudo-relational/location-based
100	/scenario-pseudo-relational/time-based
100	/scenario-pseudo-relational/user-based
100	/scenario-shared-view/location-based
100	/scenario-shared-view/time-based
100	/scenario-shared-view/user-based

3.3.2 Datageneration

Für den Ergebnistopf der Datenherstellung wurden die aus Firestore Operationen sowie Function Calls resultierenden Messungen mithilfe der View *datageneration_costs_total* kombiniert. Die Werte wurden dabei nach den in Kapitel 3.2.3 genannten Mengenfaktoren des Datenbestands (x1, x2, x5 und x10) und des Szenarios gruppiert. Auf der View basierend kam ein Microsoft Excel Sheet zum Einsatz, welches die Abweichungen zum Grundfaktor x1 darstellte, indem dort Plan- mit Istkosten verglichen wurden. Plankosten berechneten sich in diesem Fall aus den Istkosten des Faktors x1 multipliziert mit dem gewählten Faktor (bspw. x5). So konnten Trends über die über- oder unterproportionale Entwicklung der Kosten mit wachsenden Datenmengen erkannt werden. Auf diese Weise wurden Skaleneffekte in Abhängigkeit der Anzahl der generierten Reservierungen je Szenario sichtbar. Die in Excel aufbereiteten Daten wurden im Anschluss für

die Darstellung und Analyse in dieser Arbeit verwendet.

3.3.3 Read Tests

Die Aufbereitung der Messungen hinsichtlich der Read Tests basierte auf zwei Views. *tests_reads_costs_firestore* bereitete alle Firestore Operationen auf, die während der Tests registriert wurden und gruppierte diese nach Szenario und Collection. Mithilfe der View konnten die Istkosten der Zugriffe auf die einzelnen Collections für die Szenarien berechnet werden. Darüberhinaus listete *tests_reads_performance* die durchschnittlichen Ladezeiten der Perspektiven auf, die im Rahmen der Tests gemessen wurden. Die Ergebnisse der Views wurden im zweiten Schritt ebenfalls mithilfe einer Excel Tabelle für die Verwendung in dieser Arbeit transformiert.

3.3.4 Write Tests

Für die Analyse der Daten aus den Write Tests wurden ähnlich wie zuvor bei den Read Tests zwei Views für die Aspekte Firestore Costs sowie Performance definiert (*tests_writes_costs_firestore* und *test_writes_performance*). Während die Berechnung der Kosten für die Firestore Zugriffe analog zu den Reads berechnet wurden, unterschied sich die Betrachtung der Performance. Hier wurden die Laufzeiten der im Hintergrund agierenden Cloud Functions als Datengrundlage verwendet. Daraus ergab sich je Test die Dauer der vollständigen Datenaktualisierung vom Trigger bis zum letzten Update. Die dritte Komponente stellte die Betrachtung der Kosten für Cloud Functions dar, welche mithilfe der View *tests_writes_costs_functions* nach Test gruppiert abgebildet wurden. Die finale Transformation der Daten fand analog zu den vorherigen Metriken in einem Excel Sheet statt.

3.3.5 Metriken aus SonarQube

Die Aufbereitung der Daten aus SonarQube erforderte keine speziellen Tools, da diese von SonarSource in der Cloud bereits hinreichend aufbereitet werden. Es mussten lediglich die relevanten Informationen aus dem Tool extrahiert und in eine Excel Tabelle übertragen werden. Für die Auswahl der relevanten Datenpunkte waren sowohl die Erklärungen in SonarQube als auch ein pragmatisches Ausschlussverfahren hilfreich. Dabei wurden Kennzahlen aussortiert, die zwischen den Szenarien keinerlei relevanter Abweichung aufwiesen. Dazu gehörten u. a. Security Ratings, Test Coverage und Duplications, da diese Ergebnisse entweder makellos ausfielen oder in der Betrachtung eines Prototypen weniger relvant erschienen. Die für diese Arbeit zur Analyse ausgewählten

Kennzahlen sind Lines of Code (LoC), Cyclomatic Complexity und Cognitive Complexity, deren Bedeutungen in Tabelle 3.8 erläutert werden. Bei allen Metriken gilt das Mantra, dass kleinere Werte grundsätzlich vorteilhafter sind.

Tabelle 3.8: Metriken aus SonarQube

Kennzahl	*Beschreibung*
Lines of Code	Anzahl der Codezeilen
Cyclomatic Complexity	Benötigte Testfälle für 100% Coverage
Cognitive Complexity	Maßstab wie schwer verständlich der Quellcode ist

Kapitel 4

Ergebnisse

Im Folgenden werden die Ergebnisse des Experiments beschrieben. Dabei erfolgt eine Unterteilung nach den etablierten Szenarien *Pseudo Relational*, *Denormalized*, *Shared View* sowie *Dedicated View*. Innerhalb der Szenarien werden jeweils die drei Ergebnistöpfe *Datageneration*, *Reads* und *Writes* sowie die gesammelten Metriken aus SonarQube behandelt.

4.1 Szenario: Pseudo Relational

Metrik: Datageneration

Die Metrik *Datageneration* zeigte mit wachsender Datenmenge einen strikt proportionalen Anstieg der Kosten. Während beim initialen Datenbestand noch Kosten von 2,37 Cent für 13.184 Writes entstanden, waren es bei Faktor x10 exakt 23,73 Cent für 131.840 Writes. Kosten für Reads und Function Calls fielen gar nicht an, da dieses Szenario keinerlei solcher Mechaniken aufwies. Die Abweichung zum Faktor x1 lag bei allen weiteren Faktoren bei 0 (siehe Tabelle 4.1).

Tabelle 4.1: Metrik: Datageneration (Pseudo Relational)

Faktor	*Position*	*Anzahl*	*Istkosten (ct)*	*Plankosten (ct)*	*Abw. (%)*	*Anteil (%)*
x1	read	0	0,00			0,00
x1	write	13.184	2,37			100,00
x1	firestore	13.184	2,37			100,00
x1	functions	0	0,00			0,00
x1	total		2,37			100,00
x2	read	0	0,00	0,00	0,00	0,00
x2	write	26.368	4,75	4,75	0,00	100,00
x2	firestore	26.368	4,75	4,75	0,00	100,00
x2	functions	0	0,00	0,00	0,00	0,00
x2	total		4,75	4,75	0,00	100,00
x5	read	0	0,00	0,00	0,00	0,00
x5	write	65.920	11,87	11,87	0,00	100,00
x5	firestore	65.920	11,87	11,87	0,00	100,00
x5	functions	0	0,00	0,00	0,00	0,00
x5	total		11,87	11,87	0,00	100,00
x10	read	0	0,00	0,00	0,00	0,00
x10	write	131.840	23,73	23,73	0,00	100,00
x10	firestore	131.840	23,73	23,73	0,00	100,00
x10	functions	0	0,00	0,00	0,00	0,00
x10	total		23,73	23,73	0,00	100,00

Metrik: Reads

Wie in Kapitel 3.2.3 beschrieben, wurden für die Erhebung der *Reads* Metrik die drei Perspektiven *User Based*, *Time Based* und *Location Based* jeweils 100 mal hintereinander aufgerufen. Tabelle 4.2 listet die Ergebnisse nach Firestore Collection gruppiert auf. Die Spalte *Perf. (Ø ms)* gibt die durchschnittliche Zeit in Millisekunden an, die die Anwendung zur vollständigen Anzeige der entsprechenden Daten benötigte. Für den pseudo relationalen Ansatz fiel auf, dass sowohl *time-based* als auch *location-based* die selben Werte aufwiesen. Dies wird in der Tatsache begründet, dass die Anwendung technisch nicht in der Lage war, Reservierungen bereits auf Abfragenebene auf

Compounds einzuschränken, da diese nur indirekt verknüpft waren. Die Filterung fand daher im Browser statt, sodass die Kosten für Reads identisch ausfielen, während die Performance durch den zusätzlichen Schritt ein paar Millisekunden einbüßte. In Summe entstanden für 300 Seitenaufrufe Kosten in Höhe von 21,82 Cent bei 363.700 Reads. Die durchschnittliche Performance lag bei 2836 Millisekunden, wobei die Perspektive *user-based* mit 318 Millisekunden aufgrund der geringeren Datenmenge den kleinsten Teil darstellte.

Tabelle 4.2: Metrik: Reads (Pseudo Relational)

Perspektive	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Perf. (Ø ms)*
user-based	compound	500	0,03	7,69	
user-based	reservation	2000	0,12	30,77	
user-based	room	1900	0,11	29,23	
user-based	seat	2000	0,12	30,77	
user-based	user	100	0,01	1,54	
Summe		6500	0,39	100,00	318,36
time-based	compound	800	0,05	0,45	
time-based	reservation	110.800	6,65	62,04	
time-based	room	12.800	0,77	7,17	
time-based	seat	49.200	2,95	27,55	
time-based	user	5000	0,30	2,80	
Summe		178.600	10,72	100,00	4092,69
location-based	compound	800	0,05	0,45	
location-based	reservation	110.800	6,65	62,04	
location-based	room	12.800	0,77	7,17	
location-based	seat	49.200	2,95	27,55	
location-based	user	5000	0,30	2,80	
Summe		178.600	10,72	100,00	4099,33
Gesamt		363.700	21,82	100,00	2836,79

Metrik: Writes

Kapitel 3.2.3 definiert die Metrik *Writes* als die Kosten, die eine Aktualisierung einer der vier fachlichen Entitäten verursacht und die Zeit, bis alle User des Systems die aktuali-

sierten Daten vollständig sehen können. Tabelle 4.3 enthält die Ergebnisse des Tests für das Szenario *Pseudo Relational*, wobei die Verteilung der Kosten für alle durchgeführten Tests annähernd identisch ausfiel. Grund dafür war die Funktionsweise des Szenarios, da bspw. aufgrund fehlender Denormalisierung für jede Entity lediglich ein Write erforderlich war. Dieser Write wurde durch einen einzigen Function Call ausgelöst, sodass die Zeit, bis alle User die Änderung wahrnehmen konnten, für alle Operationen jeweils ein paar Millisekunden betrug (siehe Tabelle 4.3).

Tabelle 4.3: Metrik: Writes (Pseudo Relational)

Test	*Position*	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer* (s)
reservation	write	reservation	1	0,0002	77,42	
reservation	functions		1	0,0001	22,58	
reservation	total			0,0002	24,98	0,03
seat	write	seat	1	0,0002	77,88	
seat	functions		1	0,0001	22,12	
seat	total			0,0002	24,83	0,02
room	write	room	1	0,0002	76,96	
room	functions		1	0,0001	23,04	
room	total			0,0002	25,12	0,03
compound	write	compound	1	0,0002	77,11	
compound	functions		1	0,0001	22,89	
compound	total			0,0002	25,07	0,03
	total			0,0009	100,00	0,11

Metrik: SonarQube

Die von SonarQube gemessenen Daten zeigten eine vollständige Verteilung von Komplexität und Quellcode auf die Angular App, da hier keine Cloud Functions existierten. Für eine vollständige Testabdeckung der 1321 Codezeilen wären 209 Testfälle notwendig gewesen. Die Cognitive Complexity betrug für dieses Szenario exakt 15 (siehe Tabelle 4.4).

Tabelle 4.4: Metrik: SonarQube (Pseudo Relational)

Metrik	*Modul*	*Wert*	*Anteil (%)*
Lines of Code	Angular	1321	100,00
Lines of Code	Functions	0	0,00
Lines of Code	Gesamt	1321	100,00
Cyclomatic Complexity	Angular	209	100,00
Cyclomatic Complexity	Functions	0	0,00
Cyclomatic Complexity	Gesamt	209	100,00
Cognitive Complexity	Angular	15	100,00
Cognitive Complexity	Functions	0	0,00
Cognitive Complexity	Gesamt	15	100,00

4.2 Szenario: Denormalized

Metrik: Datageneration

Die Denormalisierung der Daten sorgte im zweiten Szenario zum ersten Mal dafür, dass Aufwände im Bereich der Cloud Functions entstanden. Dabei glichen die Anzahl der Calls und die Writes auf dem Firestore sich bei jedem Faktor. Lediglich die Kosten für Cloud Functions stiegen mit jedem Faktor um einige Prozentpunkte an. Während bei Faktor x1 der Kostenanteil für Functions noch bei 11,71 % lag, sorgte der genannte Anstieg für eine Erhöhung des Anteils um ca. 1 % bei Faktor x10. Bei Faktor x2 wichen die Istkosten der Functions jedoch um −1,29 % von den Plankosten ab, was auch deren Anteil an den Gesamtkosten drückte. Die interne Verteilung der Kosten des Firestores auf Reads und Writes fiel für alle Faktoren mit 57,14 % zu 42,86 % identisch aus (siehe Tabelle 4.5).

Tabelle 4.5: Metrik: Datageneration (Denormalized)

Faktor	*Position*	*Anzahl*	*Istkosten (ct)*	*Plankosten (ct)*	*Abw. (%)*	*Anteil (%)*
x1	read	105.472	6,33			57,14
x1	write	26.368	4,75			42,86
x1	firestore	131.840	11,07			88,29
x1	functions	26.368	1,47			11,71
x1	total		12,54			100,00
x2	read	210.944	12,66	12,66	0,00	57,14
x2	write	52.736	9,49	9,49	0,00	42,86
x2	firestore	263.680	22,15	22,15	0,00	88,43
x2	functions	52.736	2,90	2,94	−1,29	11,57
x2	total		25,05	25,09	−0,15	100,00
x5	read	527.360	31,64	31,64	0,00	57,14
x5	write	131.840	23,73	23,73	0,00	42,86
x5	firestore	659.200	55,37	55,37	0,00	87,94
x5	functions	131.840	7,59	7,34	3,43	12,06
x5	total		62,97	62,71	0,40	100,00
x10	read	1.054.720	63,28	63,28	0,00	57,14
x10	write	263.680	47,46	47,46	0,00	42,86
x10	firestore	1.318.400	110,75	110,75	0,00	87,51
x10	functions	263.680	15,81	14,68	7,69	12,49
x10	total		126,56	125,43	0,90	100,00

Metrik: Reads

Das zweite Szenario erwirkte durch Denormalisierung die Beschränkung von Firestore Operationen auf eine einzige Collection für alle Perspektiven (*reservation*), da alle relevanten Daten für die Anzeige dort vorgehalten wurden. So entstanden die meisten Reads und somit auch Kosten in der Perspektive *time-based* (6,65 Cent für 110.800 Operationen). Der Anteil betrug dabei rund 90 %. Die durchschnittliche Ladezeit lag bei 201,94 Millisekunden, wobei der höchtste Wert aufgrund der größeren Datenmenge ebenfalls bei *time-based* gemessen wurde. Die Gesamtkosten für alle Tests betrugen 7,4 Cent für 123.300 Reads (siehe Tabelle 4.6).

Tabelle 4.6: Metrik: Reads (Denormalized)

Perspektive	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Perf. (Ø ms)*
user-based	reservation	2000	0,12	1,62	156,19
time-based	reservation	110.800	6,65	89,86	280,70
location-based	reservation	10.500	0,63	8,52	168,92
Gesamt		123.300	7,40	100,00	201,94

Metrik: Writes

Der Test zur Analyse eines Writes einer Reservierung führte neben der genannten Operation zu jeweils einem Read auf den anderen Collections des Szenarios. Des Weiteren wurden zwei Cloud Functions ausgeführt (eine aktualisierte die Reservierung, die andere denormalisierte die benötigten Daten). Die Function Calls erzeugten in der Einzelbetrachtung die höchsten Kosten, machten in der Gesamtrechnung aber nur einen Anteil von rund 33 % aus. Die Denormalisierung war nach 2,22 Sekunden abgeschlossen und erzeugte Kosten in Höhe von 0,0006 Cent (siehe Tabelle 4.7).

Tabelle 4.7: Metrik: Reservation Writes (Denormalized)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	1	0,0001	14,29	
write	reservation	1	0,0002	42,86	
read	room	1	0,0001	14,29	
read	seat	1	0,0001	14,29	
read	user	1	0,0001	14,29	
firestore		5	0,0004	66,56	
functions		2	0,0002	33,44	
total			0,0006	100,00	2,22

Die Aktualisierung eines *Seats* löste bereits rund 400 Operationen auf dem Firestore aus, wobei der größte Kostenblock mit 39,63 % auf die Reservierungen entfiel. Während des Tests wurden zudem 74 Funktionen ausgeführt, die ihren Dienst nach ca. 12 Sekunden verrichtet hatten. Die Gesamtkosten der Simulation betrugen 0,04 Cent, die sich

zu 83,39 % auf den Firestore und zu 16,61 % auf die Cloud Functions aufteilten (siehe Tabelle 4.8).

Tabelle 4.8: Metrik: Seat Writes (Denormalized)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	73	0,0044	13,39	
read	reservation	36	0,0022	6,61	
write	reservation	72	0,0130	39,63	
read	room	73	0,0044	13,39	
read	seat	72	0,0043	13,21	
write	seat	1	0,0002	0,55	
read	user	72	0,0043	13,21	
firestore		399	0,0327	83,39	
functions		74	0,0065	16,61	
total			0,0392	100,00	11,69

Die Ergebnisse des Tests zur Aktualisierung eines *Rooms* wiesen ein ähnliches Kostenverhältnis zwischen Cloud Functions und Firestore wie der vorherige Run auf (ca. 80 zu 20). Allerdings verteilten sich die Kosten auf vergleichsweise deutlich mehr Operationen. Es wurden insgesamt 1454 Firestore Operationen sowie 346 Function Calls registriert, sodass Gesamtkosten in Höhe von 0,14 Cent entstanden. Die vollständige Aktualisierung war nach rund 41 Sekunden abgeschlossen (siehe Tabelle 4.9).

Tabelle 4.9: Metrik: Room Writes (Denormalized)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	265	0,0159	13,36	
read	reservation	132	0,0079	6,65	
write	reservation	264	0,0475	39,92	
read	room	264	0,0158	13,31	
write	room	1	0,0002	0,15	
read	seat	264	0,0158	13,31	
read	user	264	0,0158	13,31	
firestore		1454	0,1190	80,11	
functions		346	0,0296	19,89	
total			0,1486	100,00	41,49

Der nächste Test zur Aktualisierung eines *Compounds* verzeichnete einen deutlichen Anstieg der gemessenen Firestore Operationen sowie Cloud Functions. Während die Anwendung mit 4186 Function Calls ca. 9 % der Gesamtkosten verursachte, verantworteten 43.693 Firestore Operationen den größten Kostenblock. Der Test war nach rund 7 Minuten abgeschlossen, sodass fortan alle User die aktualisierten Daten an allen Reservierungen wahrnehmen konnten, und erzeugte Kosten in Höhe von 3,92 Cent (siehe Tabelle 4.10).

Tabelle 4.10: Metrik: Compound Writes (Denormalized)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	7944	0,4766	12,15	
write	compound	1	0,0002	0,00	
read	reservation	3972	0,2383	6,08	
write	reservation	7944	1,4299	36,46	
read	room	7944	0,4766	12,15	
read	seat	7944	0,4766	12,15	
read	user	7944	0,4766	12,15	
firestore		43.693	3,5750	91,16	
functions		4186	0,3465	8,84	
total			3,9215	100,00	425,46

Metrik: SonarQube

Durch die Einführung der Denormalisierung teilte sich der Quellcode dieses Szenarios sowohl auf Angular App als auch Cloud Functions auf. Aufgrund der starken Vereinfachung der Abfragelogik im Browser konnten die LoC auf 669 reduziert werden, während 273 Zeilen im Bereich der Functions hinzukamen. Damit ergab sich ein Verhältnis von 71,02 % zu 28,98 %. Die Komplexität teilte sich ebenfalls auf beide Systemteile auf. Die Browserkomponente verantwortete mit 87 Testfällen den größten Teil der Cyclomatic Complexity, während die Cognitive Complexity der Functions mit 7 gegen 4 fast doppelt so hoch ausfiel (siehe Tabelle 4.11).

Tabelle 4.11: Metrik: SonarQube (Denormalized)

Metrik	*Modul*	*Wert*	*Anteil (%)*
Lines of Code	Angular	669	71,02
Lines of Code	Functions	273	28,98
Lines of Code	Gesamt	942	100,00
Cyclomatic Complexity	Angular	87	83,65
Cyclomatic Complexity	Functions	17	16,35
Cyclomatic Complexity	Gesamt	104	100,00
Cognitive Complexity	Angular	4	36,36
Cognitive Complexity	Functions	7	63,64
Cognitive Complexity	Gesamt	11	100,00

4.3 Szenario: Shared View

Metrik: Datageneration

Die Datenerhebung für das Szenario *Shared View* konnte erfolgreich abgeschlossen werden und zeigte eine steigende Kostenentwicklung im Bereich der Cloud Functions von bis zu 27,22 % bei Faktor x10. Auch die Kosten für Reads wichen von den erwarteten Werten ab, sodass die Abweichung der Gesamtkosten mit jedem Faktor um etwas mehr als 1 % stieg. Die Gesamterhöhung fiel geringer aus als die Abweichung der Kosten für Cloud Functions zunächst vermuten ließ, da deren Anteil lediglich bei 10 – 12 % lag. Der Hauptanteil entfiel demnach auf die Kosten für den Firestore, während die interne Verteilung von Reads und Writes sich stets im Bereich 45 % zu 55 % bewegte (siehe Tabelle 4.12).

Tabelle 4.12: Metrik: Datageneration (Shared View)

Faktor	*Position*	*Anzahl*	*Istkosten (ct)*	*Plankosten (ct)*	*Abw. (%)*	*Anteil (%)*
x1	read	128.250	7,70			44,77
x1	write	52.736	9,49			55,23
x1	firestore	180.986	17,19			90,40
x1	functions	26.368	1,83			9,60
x1	total		19,01			100,00
x2	read	259.485	15,57	15,39	1,16	45,06
x2	write	105.472	18,98	18,98	0,00	54,94
x2	firestore	364.957	34,55	34,37	0,52	89,89
x2	functions	52.736	3,89	3,65	6,41	10,11
x2	total		38,44	38,03	1,09	100,00
x5	read	655.610	39,34	38,48	2,24	45,32
x5	write	263.680	47,46	47,46	0,00	54,68
x5	firestore	919.290	86,80	85,94	1,00	89,37
x5	functions	131.840	10,33	9,13	13,09	10,63
x5	total		97,12	95,07	2,16	100,00
x10	read	1.314.810	78,89	76,95	2,52	45,39
x10	write	527.360	94,92	94,92	0,00	54,61
x10	firestore	1.842.170	173,81	171,87	1,13	88,21
x10	functions	263.680	23,23	18,26	27,22	11,79
x10	total		197,05	190,14	3,63	100,00

Metrik: Reads

Das Bucketing im *Shared View* Szenario erlaubte eine durchschnittliche Performance von 84,49 Millisekunden pro Testlauf. Auch hier entfiel der größte Teil auf die Perspektive *time-based* (ca. 85 %). Die Gesamtanzahl an Firestore Operationen betrug exakt 35.000, sodass Kosten in Summe von 2,1 Cent anfielen. Die Reads zielten dabei alle auf die selbe Firestore Collection ab (siehe Tabelle 4.13).

Tabelle 4.13: Metrik: Reads (Shared View)

Perspektive	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Perf. (Ø ms)*
user-based	view	500	0,03	1,43	54,08
time-based	view	29.900	1,79	85,43	133,26
location-based	view	4600	0,28	13,14	66,12
Gesamt		35.000	2,10	100,00	84,49

Metrik: Writes

Die Aktualisierung einer Reservierung erzeugte Gesamtkosten in Höhe von 0,0008 Cent und benötigte rund 2 Sekunden. Die Kosten teilten sich dabei zu ca. 80:20 auf Firestore und Cloud Functions auf, wobei Firestore intern die größten Teile mit jeweils 21,49 % durch Writes auf Reservierung und View anfielen (siehe Tabelle 4.14).

Tabelle 4.14: Metrik: Reservation Writes (Shared View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	1	0,0001	9,09	
write	reservation	1	0,0002	21,49	
read	room	1	0,0001	7,16	
read	seat	1	0,0001	7,16	
read	user	1	0,0001	7,16	
read	view	1	0,0001	7,16	
write	view	1	0,0002	21,49	
firestore		7	0,0007	78,79	
functions		2	0,0002	21,21	
total			0,0008	100,00	2,04

Das Update eines Platzes erforderte jeweils 72 Writes auf *Reservation* und *View*, wobei diese Operationen auch hier den größten Anteil der Firestore Zugriffe ausmachten. Insgesamt generierte der Test 541 Reads und Writes sowie 74 Function Calls. Letztere trugen mit 0,0072 Cent zu 12,68 % der Gesamtkosten bei. Diese betrugen 0,06 Cent. Nach rund 14 Sekunden war die gesamte Aktualisierung der Daten abgeschlossen (siehe

Tabelle 4.15).

Tabelle 4.15: Metrik: Seat Writes (Shared View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	72	0,0043	8,66	
read	reservation	36	0,0022	4,33	
write	reservation	72	0,0130	25,99	
read	room	72	0,0043	8,66	
read	seat	72	0,0043	8,66	
write	seat	1	0,0002	0,36	
read	user	72	0,0043	8,66	
read	view	72	0,0043	8,66	
write	view	72	0,0130	25,99	
firestore		541	0,0499	87,32	
functions		74	0,0072	12,68	
total			0,0571	100,00	13,59

Beim nächsten Test entstanden innerhalb von rund 50 Sekunden Kosten in Höhe von 0,21 Cent, die sich zu 85 % auf Firestore Operationen sowie 15 % auf Cloud Functions aufteilten. Das interne Verhältnis der Firestore Zugriffe war ähnlich beschaffen wie bei den vorherigen Tests, allerdings betrug die Summe 1981 Stück. Des Weiteren registrierte die Statistik 346 Function Calls (siehe Tabelle 4.16).

Tabelle 4.16: Metrik: Room Writes (Shared View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	264	0,0158	8,69	
read	reservation	132	0,0079	4,34	
write	reservation	264	0,0475	26,06	
read	room	264	0,0158	8,69	
write	room	1	0,0002	0,10	
read	seat	264	0,0158	8,69	
read	user	264	0,0158	8,69	
read	view	264	0,0158	8,69	
write	view	264	0,0475	26,06	
firestore		1981	0,1823	85,04	
functions		346	0,0321	14,96	
total			0,2144	100,00	49,43

Der letzte Test des Szenarios aktualisierte einen *Compound*, wobei die Updates für *Reservation* sowie *View* nach wie vor die selbe Größenordnung in der Verteilung der Zugriffe ausmachten. Insgesamt fielen Firestore Kosten in Summe von 5,48 Cent an, welche mit 93,79 % den größten Anteil bildeten. Die restlichen 0,36 Cent wurden durch 4186 Function Calls generiert. Insgesamt benötigte die Aktualisierung aller relevanten Daten ca. 7,5 Minuten (siehe Tabelle 4.17).

Tabelle 4.17: Metrik: Compound Writes (Shared View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	7944	0,4766	8,70	
write	compound	1	0,0002	0,00	
read	reservation	3972	0,2383	4,35	
write	reservation	7944	1,4299	26,09	
read	room	7944	0,4766	8,70	
read	seat	7944	0,4766	8,70	
read	user	7944	0,4766	8,70	
read	view	7944	0,4766	8,70	
write	view	7944	1,4299	26,09	
firestore		59.581	5,4815	93,79	
functions		4186	0,3632	6,21	
total			5,8447	100,00	462,94

Metrik: SonarQube

Das Szenario erforderte 930 Codezeilen, die sich zu ca. 75 % auf die Browseranwendung aufteilten, während der Rest für die Cloud Functions benötigt wurde. Die erforderlichen Testfälle verortete SonarQube ebenfalls zu großen Teilen in der Angular App (ca. 85 %), wobei dieser Teil aufgrund der Cyclomatic Complexity von 4 leichter verständlich war als die Functions (11) (siehe Tabelle 4.18).

Tabelle 4.18: Metrik: SonarQube (Shared View)

Metrik	*Modul*	*Wert*	*Anteil (%)*
Lines of Code	Angular	693	74,52
Lines of Code	Functions	237	25,48
Lines of Code	Gesamt	930	100,00
Cyclomatic Complexity	Angular	97	84,35
Cyclomatic Complexity	Functions	18	15,65
Cyclomatic Complexity	Gesamt	115	100,00
Cognitive Complexity	Angular	4	36,36
Cognitive Complexity	Functions	7	63,64
Cognitive Complexity	Gesamt	11	100,00

4.4 Szenario: Dedicated View

Metrik: Datageneration

Die Ergebnisse der Datageneration für das Szenario *Dedicated View* konnten nicht alle gesammelt werden, da die Anwendung nicht in der Lage war, die Daten für die Faktoren x5 und x10 zu generieren. Grund dafür war die Beschränkung des Firestores auf das in Kapitel 2.2.5 genannte Limit von 1 MB pro Dokument. Daher stellt Tabelle 4.19 lediglich die Werte für die Faktoren x1 und x2 dar.

Faktor x2 zeigte eine geringe Abweichung der Firestore Kosten von 0,10 %, die sich rein aus den Reads ergab. Die Abweichung der Istkosten der Cloud Functions betrug 48,66 % und trug so zu dem um 6,51 % gesteigerten Anteil an den Gesamtkosten bei. Die Verteilung der Kosten für Reads und Writes lag für beide Faktoren bei ca. 42 % zu 58 % und wies so einen deutlich stärkeren Fokus auf Writes auf.

Tabelle 4.19: Metrik: Datageneration (Dedicated View)

Faktor	*Position*	*Anzahl*	*Istkosten (ct)*	*Plankosten (ct)*	*Abw. (%)*	*Anteil (%)*
x1	read	144.316	8,66			42,19
x1	write	65.920	11,87			57,81
x1	firestore	210.236	20,52			82,28
x1	functions	65.920	4,42			17,72
x1	total		24,94			100,00
x2	read	289.340	17,36	17,32	0,25	42,25
x2	write	131.840	23,73	23,73	0,00	57,75
x2	firestore	421.180	41,09	41,05	0,10	75,77
x2	functions	131.840	13,14	8,84	48,66	24,23
x2	total		54,23	49,89	8,71	100,00

Metrik: Reads

Die Tests des Szenarios *Dedicated View* zeichneten sich durch eine gleichmäßige Verteilung der Kosten auf die einzelnen Perspektiven aus (33,33 %), da hier bei jedem Aufruf immer exakt ein Dokument geladen wurde. Die Gesamtkosten beliefen sich daher auf 0,02 Cent für 300 Reads. Lediglich die Performance schlug bei *time-based* aufgrund der höheren Dokumentgröße nach oben aus, lag im Durchschnitt jedoch bei 44,16 Millisekunden pro Aufruf. Die Firestore Zugriffe fanden je Perspektive auf einer dedizierten Collection statt (siehe Tabelle 4.20).

Tabelle 4.20: Metrik: Reads (Dedicated View)

Perspektive	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Perf. (Ø ms)*
user-based	views/user	100	0,01	33,33	31,03
time-based	views/year-month	100	0,01	33,33	68,59
location-based	views/compound	100	0,01	33,33	32,87
Gesamt		300	0,02	100,00	44,16

Metrik: Writes

Das Szenario *Dedicated View* erzeugte strukturell eine ähnliche Verteilung der Reads und Writes wie *Shared View*, allerdings waren hier mehr Collections involviert. So entfiel der größten Kostenanteil von ca. 60 % intern nach wie vor auf die Writes. Insgesamt sorgte der Test für Kosten in Höhe von 0,002 Cent für 11 Firestore Zugriffe sowie 5 Cloud Functions. Der Run wurde nach 3,60 Sekunden erfolgreich abgeschlossen (siehe Tabelle 4.21).

Tabelle 4.21: Metrik: Reservation Writes (Dedicated View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	1	0,0001	5,26	
write	reservation	1	0,0002	15,79	
read	room	1	0,0001	5,26	
read	seat	1	0,0001	5,26	
read	user	1	0,0001	5,26	
read	views/compound	1	0,0001	5,26	
write	views/compound	1	0,0002	15,79	
read	views/year-month	1	0,0001	5,26	
write	views/year-month	1	0,0002	15,79	
read	views/user	1	0,0001	5,26	
write	views/user	1	0,0002	15,79	
firestore		11	0,0011	62,24	
functions		5	0,0007	37,76	
total			0,0018	100,00	3,60

Der Test zur Aktualisierung eines *Seats* erzeugte durch 182 Function Calls und 613 Firestore Operationen Gesamtkosten in Höhe von 0,12 Cent innerhalb von 26,23 Sekunden. Dabei fiel die Verteilung jener Kosten nahezu 50:50 aus. Der größte Kostenanteil der Writes wurde bei den Reservierungen verortet (22,15 %), gefolgt von den View Collections (11,08 %) (siehe Tabelle 4.22).

Tabelle 4.22: Metrik: Seat Writes (Dedicated View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	72	0,0043	7,38	
read	reservation	36	0,0022	3,69	
write	reservation	72	0,0130	22,15	
read	room	72	0,0043	7,38	
read	seat	72	0,0043	7,38	
write	seat	1	0,0002	0,31	
read	user	72	0,0043	7,38	
read	views/compound	36	0,0022	3,69	
write	views/compound	36	0,0065	11,08	
read	views/year-month	36	0,0022	3,69	
write	views/year-month	36	0,0065	11,08	
read	views/user	36	0,0022	3,69	
write	views/user	36	0,0065	11,08	
firestore		613	0,0585	48,84	
functions		182	0,0613	51,16	
total			0,1198	100,00	26,23

Beim nächsten Test zum Update eines Raums fielen Kosten in Höhe von 0,33 Cent an, die zu 34 % aus 862 Function Calls bestanden. Die restlichen 66 % entfielen auf 2245 Firestore Operationen. Diese teilten sich auf 13 Kombinationen aus Collection und Zugriffstyp auf, wobei auch hier ähnliche Verhältnisse wie beim vorherigen Test herrschten. Die Aktualisierung der Daten dauerte insgesamt 46 Sekunden (siehe Tabelle 4.23).

Tabelle 4.23: Metrik: Room Writes (Dedicated View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	264	0,0158	7,40	
read	reservation	132	0,0079	3,70	
write	reservation	264	0,0475	22,20	
read	room	264	0,0158	7,40	
write	room	1	0,0002	0,08	
read	seat	264	0,0158	7,40	
read	user	264	0,0158	7,40	
read	views/compound	132	0,0079	3,70	
write	views/compound	132	0,0238	11,10	
read	views/year-month	132	0,0079	3,70	
write	views/year-month	132	0,0238	11,10	
read	views/user	132	0,0079	3,70	
write	views/user	132	0,0238	11,10	
firestore		2245	0,2140	65,56	
functions		862	0,1124	34,44	
total			0,3264	100,00	46,02

Der letzte Durchlauf zur Aktualisierung eines *Compounds* offenbarte mit rund 88 % einen deutlich höheren Anteil an Firestore Zugriffen als die anderen Tests. Einen erheblichen Teil zu den Gesamtkosten in Höhe von 7,28 Cent trugen die 67.525 Operationen bei. Die interne Verteilung der Aufwände unterschied sich nur marginal zu den vorherigen Runs. Des Weiteren wurden insgesamt 10.462 Function Calls registriert, welche ihren Dienst nach 4 Minuten und 18 Sekunden erfolgreich verrichtet hatten (siehe Tabelle 4.24).

Tabelle 4.24: Metrik: Compound Writes (Dedicated View)

Position	*Collection*	*Anzahl*	*Kosten (ct)*	*Anteil (%)*	*Dauer (s)*
read	compound	7944	0,4766	7,41	
write	compound	1	0,0002	0,00	
read	reservation	3972	0,2383	3,70	
write	reservation	7944	1,4299	22,22	
read	room	7944	0,4766	7,41	
read	seat	7944	0,4766	7,41	
read	user	7944	0,4766	7,41	
read	views/compound	3972	0,2383	3,70	
write	views/compound	3972	0,7150	11,11	
read	views/year-month	3972	0,2383	3,70	
write	views/year-month	3972	0,7150	11,11	
read	views/user	3972	0,2383	3,70	
write	views/user	3972	0,7150	11,11	
firestore		67.525	6,4348	88,43	
functions		10.462	0,8421	11,57	
total			7,2769	100,00	258,23

Metrik: SonarQube

Für die Implementierung des Szenarios *Dedicated View* waren mehr als 1000 Codezeilen notwendig. Aufgrund des Mehraufwands durch die separate Erzeugung der einzelnen Views entfiel ein größerer Anteil auf die Cloud Functions als bei den anderen Szenarien (ca. 35 %). Die Cognitive Complexity verteilte sich fast zu gleichen Teilen auf Web und Functions (7 und 8). Volle Testabdeckung hätte mit mindestens 118 Tests erreicht werden können, wobei der Großteil von 82,20 % auf die Browseranwendung entfiel (siehe Tabelle 4.25).

Tabelle 4.25: Metrik: SonarQube (Dedicated View)

Metrik	*Modul*	*Wert*	*Anteil (%)*
Lines of Code	Angular	691	64,64
Lines of Code	Functions	378	35,36
Lines of Code	Gesamt	1069	100,00
Cyclomatic Complexity	Angular	97	82,20
Cyclomatic Complexity	Functions	21	17,80
Cyclomatic Complexity	Gesamt	118	100,00
Cognitive Complexity	Angular	7	46,67
Cognitive Complexity	Functions	8	53,33
Cognitive Complexity	Gesamt	15	100,00

Kapitel 5

Diskussion

Zur Untersuchung der Frage, welche Ansätze für einen effizienten Einsatz des Cloud Firestores geeignet sind, wurde ein Experiment durchgeführt. Dabei wurden Daten zu Kosten, Performance und Codekomplexität erfasst, die in diesem Kapitel interpretiert und zur Ableitung von Handlungsempfehlungen verwendet werden.

5.1 Analyse der Szenarien

Im Folgenden werden die Forschungsergebnisse vor dem Hintergrund der einzelnen Unterfragen aus Kapitel 1.1 zusammengefasst und analysiert. Dabei wird untersucht, wie die unterschiedlichen Szenarien in jenen Teilbereichen gegeneinander performen.

5.1.1 Einfluss auf Kosten

Welchen Einfluss haben die gewählten Maßnamen auf die Kosten?

Die Kostenbetrachtung fand während des Experiments auf mehreren Ebenen statt: *Datageneration*, *Reads* und *Writes*. Erstere erforschte die Kosten, die für die Herstellung bestimmter Datenmengen anfielen, während Reads und Writes sich auf die entsprechenden Funktionen der laufenden Anwendung fokussierten.

Datageneration Costs

Die Kosten zur Herstellung des Datenbestands erfassten sowohl Informationen zur Nutzung des Firestores als auch der im Hintergrund agierenden Cloud Functions. Abbildung 5.1 fasst die Gesamtkosten der Szenarien für die getesteten Faktoren zusammen.

Dabei zeigt sich der Trend, dass mit fortschreitender Optimierung bzw. Komplexität des Szenarios die Kosten für die Generierung sukzessive steigen. So erzeugt *Pseudo Relational* bei Faktor x10 annähernd so hohe Kosten wie die nächste Optimierungsstufe bei Faktor x2 (*Denormalized*). Die Aufwände von *Denormalized* und *Shared View* schwanken aufgrund der ähnlichen Struktur der Cloud Functions im Vergleich, liegen jedoch stets in einer gleichwertigen Klasse. Auch wenn für *Dedicated View* ab Faktor x5 keine Daten mehr gesammelt wurden, zeigt sich auch dort der genannte Trend. Die Problematik dieses Tests wird in Kapitel 5.2 näher erläutert.

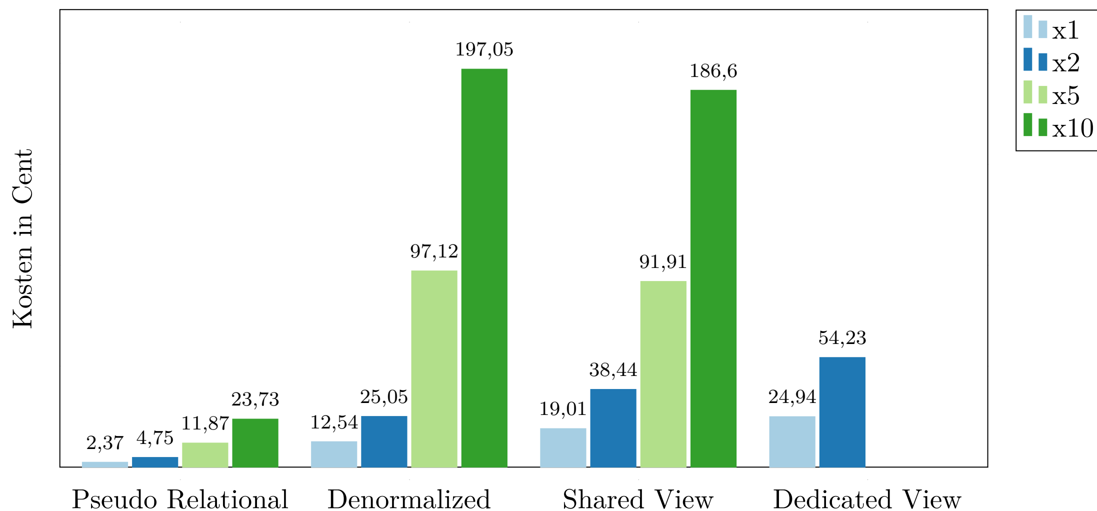


Abbildung 5.1: Vergleich der Datageneration Costs

Aus den Ergebnissen lässt sich schließen, dass die Betriebskosten einer Anwendung, deren Implementierung sich konsequent einer der vier Szenarien verschrieben hat, mit wachsender Datenmenge unterschiedlich stark steigen. Dieser Umstand sollte während der Konzeption und Entwicklung bedacht werden, um nicht im produktiven Betrieb auf unerwartete Risiken zu stoßen.

Read Costs

Die *Read Costs* stellten eine Metrik zur Analyse der Kosten für lesende Zugriffe aus der laufenden Anwendung heraus dar. Abbildung 5.2 zeigt dabei eine deutlich sinkende Tendenz für stärker optimierte Szenarien wie *Shared View* und *Dedicated View*. Da bei Letzterem stets nur ein Read pro Aufruf der Perspektive ausgelöst wurde, betragen die Kosten hier das absolute Minimum.

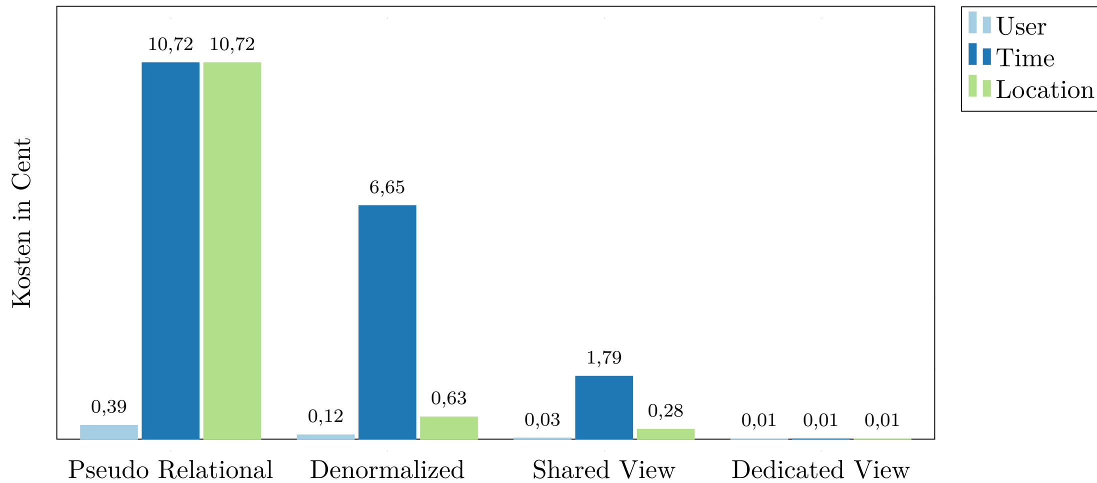


Abbildung 5.2: Vergleich der Read Costs (100 Runs)

Des Weiteren wird bei *Denormalized* sowie *Shared View* deutlich, dass die relative Kostenverteilung zwischen *User Based*, *Time Based* und *Location Based* stets gleichwertig und somit in direkter Abhängigkeit zu der tatsächlich benötigten Datenmenge ausfällt. Lediglich *Pseudo Relational* macht durch einen Ausreißer nach oben auf sich aufmerksam, da die Kosten für die letzteren Perspektiven identisch sind. Dies liegt in der Tatsache begründet, dass für dieses Szenario ein *Application Side Join* notwendig war, da eine effiziente Datenabfrage durch direkte Einschränkung auf den *Compound* unmöglich war. Auch für *User Based* wies diese Methode die höchsten Kosten ihrer Klasse auf, wodurch das Szenario im Ranking den letzten Platz belegte.

Write Costs

Das Experiment untersuchte die Kosten von schreibenden Operationen auf dem Firestore, indem bestimmte Dokumente der Kernentitäten *Reservation*, *Seat*, *Room* und *Compound* aktualisiert wurden. Dabei zeigte sich eine Tendenz zu steigenden Kosten je komplexer der Grundgedanke des betrachteten Verfahrens. Abbildung 5.3 verdeutlicht diesen Trend. Hier wird offensichtlich, dass *Pseudo Relational* im Vergleich annähernd keine Kosten verursachte, da hier keine Denormalisierung von Daten notwendig war und somit keine triggerbasierten Cloud Functions existierten. Diese wurden erst mit *Denormalized* eingeführt, wo sie einen Anteil von rund 10 % der Kosten ausmachten. Die Szenarien *Shared View* und *Dedicated View* erzeugten in der Folge noch höhere Kosten, wobei der Anteil der Cloud Functions beim Letzteren einen deutlich höheren Anteil bedienten, da hier pro View eine zusätzliche Funktion notwendig war.

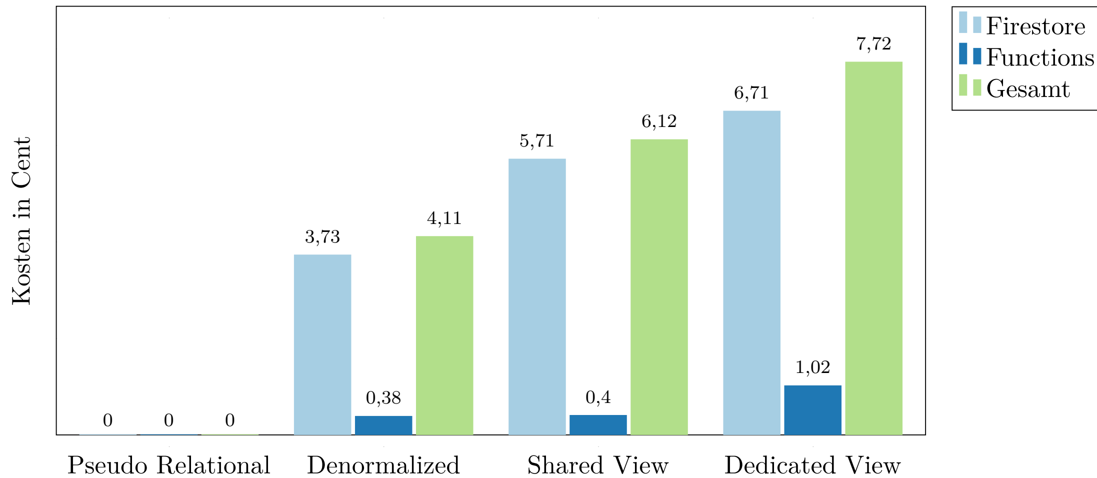


Abbildung 5.3: Vergleich der Write Costs

Aus den Ergebnissen lässt sich schließen, dass die Kosten für Writes höher ausfallen, je stärker die benötigten Daten vom System aufbereitet werden müssen. So sind denormalisierende Techniken bei direkten Zugriffen auf die entsprechenden Collections gar nicht notwendig und erzeugen damit auch keine Kosten.

Gesamtkosten

In der Gesamtbetrachtung zeigt sich eine deutliche antiproportionale Beziehung zwischen *Read Costs* und *Write Costs*. Szenarien, die einen starken Fokus auf kostenoptimierte Reads legen, weisen tendenziell höhere Kosten für schreibende Operationen auf und umgekehrt. Diese Erkenntnis sollte in der Verwendung des Cloud Firestores unbedingt beachtet werden, um keine überraschende Abrechnung von Google zu erhalten.

5.1.2 Verständlichkeit und Wartbarkeit des Codes

Wie verhalten sich Verständlichkeit und Wartbarkeit der Anwendung?

Wenn zugrunde gelegt wird, dass ein kleinerer Wert von höherer Güte ist, zeigten die von SonarQube gemessenen Metriken einen klaren Vorteil bei *Denormalized*, da hier die niedrigsten Komplexitäten errechnet wurden. Die *LoC* lagen mit nur geringfügigem Abstand zu *Shared View* auf dem zweiten Platz. *Dedicated View* wurde im Vergleich dazu etwas komplexer bewertet, während durch die Aufteilung auf mehrere View Collections mehr *LoC* benötigt wurden. Die mit Abstand höchsten Komplexitäten und *LoC* erreichte *Pseudo Relational*. Die Werte werden in Abbildung 5.4 dargestellt, wobei für

jedes Szenario drei Balken für die Metriken *LoC*, *Cyclomatic Complexity* sowie *Cognitive Complexity* gezeichnet werden.

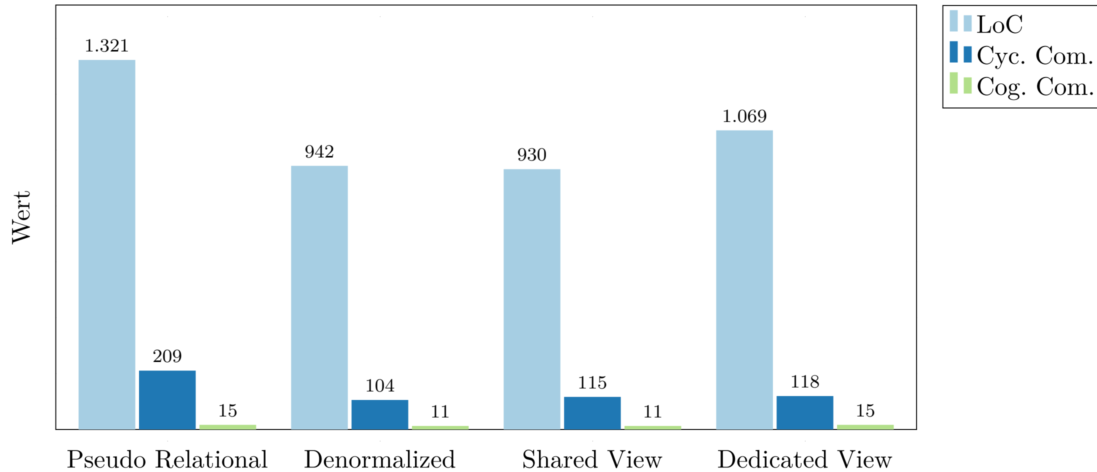


Abbildung 5.4: Vergleich der Metriken zur Komplexität

Aus den vorliegenden Daten lässt sich schlussfolgern, dass auf dem Papier ausgefeilter wirkende Vorgehensweisen wie Bucketing und Aggregation nicht zwangsläufig für verständlicheren und besser wartbaren Code sorgen. Im Gegenteil, die simple Denormalisierung von Collections ist laut SonarQube zu favorisieren, da hier weniger Code notwendig ist und die Komplexität somit tendenziell geringer ausfällt. Grund dafür ist die sehr einfache Abfragelogik im Browser gepaart mit simplen Cloud Functions zur Datenaktualisierung. Die View Szenarien benötigen zumindest auf Seiten der Functions weiteren Code zur Aufbereitung. Außerdem wird deutlich, dass das relationale Mindset zwar keine serverseitige Komplexität erschafft, im Client jedoch zu deutlich mehr Quellcode sowie einer unverhältnismäßig komplexeren Abfragelogik führt.

5.1.3 Auswirkungen auf Performance

Welche Auswirkungen auf Performance erwirken die Ansätze?

Der Aspekt der Performance wurde während des Versuchs aus zwei Perspektiven betrachtet. Zum Einen untersuchte das Experiment die durchschnittliche Ladezeit in lesenden Anwendungsfällen (*User Based*, *Time Based* und *Location Based*), zum Anderen wurden Daten über Laufzeiten von Schreibvorgängen auf den vier Kernentitäten *Reservation*, *Seat*, *Room* sowie *Compound* gesammelt.

Read Performance

Abbildung 5.5 beleuchtet die Ergebnisse der *Read Performance*, indem für jedes Szenario die drei Usecases als Säulen dargestellt werden. Auf der Ordinate wird dabei stets die durchschnittliche Dauer in Millisekunden angegeben.

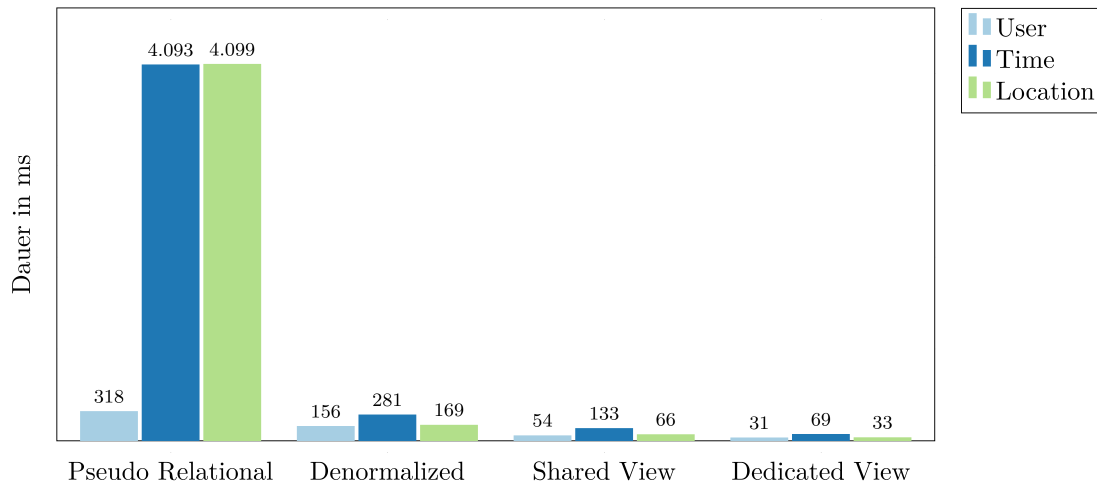


Abbildung 5.5: Vergleich der Metriken zur Read Performance

Die Daten zeigen eine klare Tendenz zu verbesserter Performance je stärker die Reads optimiert werden. So sind die Ladezeiten bei den Szenarien *Shared View* und *Dedicated View* im Grunde genommen zu vernachlässigen, obwohl hier verhältnismäßig hohe Anzahlen an Reservierungen angezeigt wurden. Das liegt darin begründet, dass diese Methoden die eigentlich anzuzeigenden Objekte aggregieren bzw. durch Bucketing sammeln und geschickt für den Client portionieren. Das Szenario *Denormalized* liefert jedoch ebenfalls sehr ansehnliche Werte, vor allem vor dem Hintergrund, dass bspw. in der Perspektive *Time Based* pro Testlauf über 1000 Reservierungen und damit auch Dokumente geladen wurden. Lediglich *Pseudo Relational* sticht mit zwei extremen Negativbeispielen aus der Masse heraus, die sich durch die Tatsache begründen, dass dort ein *Application Side Join* notwendig war. Dieser skalierte gerade bei großen Datenmengen extrem schlecht, da pro Run über 1700 Dokumente geladen wurden und im Browser untereinander zugeordnet werden mussten. Bei kleineren Datenmengen (siehe *User Based*) grenzte die durchschnittliche Ladezeit zwar noch an einen akzeptablen Wert, allerdings schnitten die anderen Szenarien im Vergleich deutlich besser ab.

Write Performance

Die Auswertung der *Write Performance* offenbarte einen interessanten Trend, dessen Implikationen für Entscheidungen hinsichtlich der Datenmodellierung einen gewichtigen Faktor darstellen sollte: Je „weiter" das zu aktualisierende Dokument von dem angezeigten Objekt durch implizite Fremdschlüsselbeziehungen entfernt ist, desto länger dauert bei einem Write die Reconciliation jener Daten. Diese Annahme gilt natürlich nur dann, wenn eine einigermaßen gleichmäßige Aufteilung der Beziehungen in der Datenbank vorliegt. So zeichnet Abbildung 5.6 für alle denormalisierende Szenarien exakt das gleiche Bild: Abgesehen davon, dass die konkreten Werte zwischen diesen variieren, bewegen sich die drei Säulenmengen in der selben Güteklasse. Lediglich *Pseudo Relational* zeigte annähernd konstante Werte in der Performance, da hier aufgrund nicht notwendiger Denormalisierung stets nur eine einzige Write Function getriggert wurde.

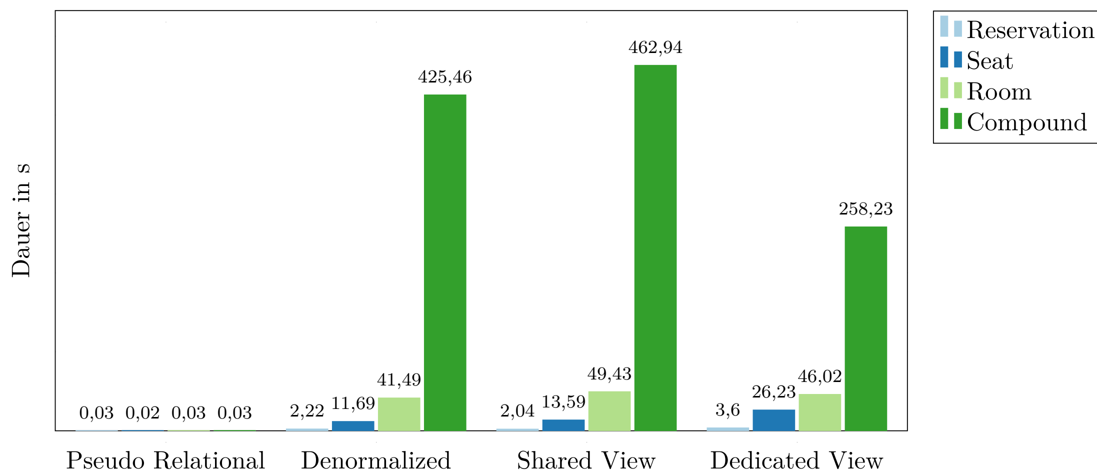


Abbildung 5.6: Vergleich der Metriken zur Write Performance

Wenn die Diagramme zu *Read Performance* und *Write Performance* miteinander verglichen werden, lässt sich daraus schließen, dass die Szenarien mit überdurschnittlich guter Performance in den lesenden Zugriffen zu längeren Laufzeiten in den schreibenden Gegenstücken tendieren. Andersherum sorgt eine optimierte *Write Performance* jedoch für sehr schlechte Werte in den Reads. Diese Erkenntnis stützt die zweite und dritte Hypothese aus Kapitel 2.3. Diese besagen, dass ein pseudo relationales Modell leseintensiven Anwendungsfällen schadet und ein starker Fokus auf die Optimierung von Reads zu schwächerer *Write Performance* führt.

5.2 Beschränkungen des Experiments

Das Experiment ist während seiner Durchführung auf gewisse Hürden und Probleme gestoßen, die unterschiedliche Auswirkungen auf dessen Ausgang hatten. Diese werden im Folgenden erläutert.

Externe Datensammlung statt interner Metriken

Das Firebase SDK bietet leider keine Möglichkeit, die in dieser Arbeit erforderlichen Metriken in ausreichender Detailtiefe zu erheben. Google bietet zwar Tools wie den Key Visualizer an, mit dem die Verteilung der Firestore Operationen auf Collections visualisiert werden können, dieser arbeitet jedoch auf einem sehr aggregierten Level und war daher für dieses Experiment nicht geeignet. Daher musste auf ein selbst implementiertes System für die Erhebung von Metriken zurückgegriffen werden, welches aus Sicht des SDKs von außen agiert. Da bei Zugriffen auf den Firestore stets die Anzahl der geschrieben bzw. gelesenen Dokumente erfasst wurde, fielen etwaige interne Optimierungen wie Offline Support und Caching aus dem Raster. Da alle Szenarien jedoch auf die selbe Weise untersucht und jene Optimierungen bspw. durch komplette Reloads der Browseranwendung vermieden wurden, kann die Datenerfassung als konsistent angesehen werden, auch wenn gemessene Kosten während des Experiments sich von den realen Kosten einer tatsächlichen App unterscheiden können. Da das Experiment somit in einer fixen Umgebung durchgeführt wurde, konnten zudem etwaige Differenzen durch Schwankungen in der Systemleistung, etc. minimiert werden.

Beschränkungen in der Dokumentgröße

Die Kosten zur Herstellung des Datenbestands für das Szenario *Dedicated View* konnten für Faktor x5 und x10 nicht gesammelt werden, da hier die Limitierung des Firestores auf 1 MB pro Dokument zum Tragen kam. Dieser Umstand war für die Auswertung der Ergebnisse jedoch von Vorteil, da so die unterdurchschnittliche Fähigkeit des Verfahrens zu skalieren bzw. die potenzielle Notwendigkeit zur Reevaluierung der View Scopes aufgedeckt wurde. Bei *Shared View* fiel das Problem nicht auf, allerdings ist dieses Szenario aufgrund seiner Funktionsweise bei ausreichend großen Datenmengen auch von der Gefahr betroffen, an dieses Limit zu stoßen. Die Ergebnisse der anderen Tests können jedoch trotzdem als aussagekräftig angesehen werden, da diese von der initialen Herstellung des Datasets entkoppelt waren und auf einem ausreichend kleinen Datenbestand operierten.

5.3 Untersuchung der Hypothesen

Mit den Ergebnissen des Experiments kann die Nullhypothese aus Kapitel 2.3, dass Kosten, Performance und Komplexität unabhängig vom Umgang mit dem Cloud Firestore sind, abgelehnt werden. Die Szenarien erzeugten sehr unterschiedliche Daten für die genannten Faktoren, sodass diese unterschiedlich gut für Firestore Apps geeignet sind.

Wenn eine stark kostenfokussierte Optimierung vorgenommen wird, leiden wahrgenommene Performance und Wartbarkeit der Anwendung darunter.

Das Szenario *Pseudo Relational* kann als stark kostenfokussiert angesehen werden, da hier sowohl *Datageneration Costs* als auch *Write Costs* außerordentlich niedrig ausfielen. Lediglich die *Read Costs* lagen höher, allerdings nur in Situationen mit großen Datenmengen. Das Szenario erzielte bei der Komplexität zudem die unvorteilhaftesten Werte. Darüberhinaus zeigte sich hier die mit Abstand schlechteste *Read Performance*, welche die sehr gute *Write Performance* für leseintensive Anwendungsfälle in den Hintergrund rückte. Die Hypothese kann daher für Usecases angenommen werden, deren Daten eine niedrigere Volatilität aufweisen, sodass der Vorteil bei der *Write Performance* weniger ins Gewicht fällt.

Ein rein auf relationalen Datenbanken basiertes Mindset schadet leseintensiven Anwendungsfällen signifikant.

Diese Hypothese lässt sich auf Seiten der Performance eindeutig beweisen. Das Szenario *Pseudo Relational* weist im Vergleich zu den anderen Verfahren deutlich schlechtere Werte in dieser Kategorie auf. Auch die Kennzahlen zur Komplexität liegen bei dieser Methode auf dem letzten Platz, da hier die meisten *LoC* sowie Softwaretests benötigt werden. Im Bereich der Kosten kann das Szenario lediglich bei den *Write Costs* bestehen, da hier nur verschwindend geringe Beträge zu Buche schlagen. Die Kosten für Reads liegen im Vergleich in einem deutlich höheren Bereich. Da die Hypothese sich auf leseintensive Anwendungsfälle bezieht, kann auch diese angenommen werden.

Je stärker der Fokus auf die Optimierung von Reads gelegt wird, desto schwächer ist die Write Performance.

Diese Hypothese vermutet eine gegensätzliche Wechselwirkung zwischen den Faktoren *Read Performance* bzw. *Read Costs* und *Write Performance*. Die Ergebnisse lassen hier einen klaren Trend erkennen: Die Szenarien mit einer schlechten *Write Performance* weisen gute bis sehr gute Werte bei *Read Costs* und *Read Performance* auf. Allerdings sorgt der positive Ausreißer der *Write Performance* von *Dedicated View* dafür, dass keine

strikt stufenweise Verschlechterung der Kennzahl mit jedem Sprung in der Verbesserung der Reads vorliegt.

5.4 Ableitung von Handlungsempfehlungen

Die Auswahl der passenden Methode zur Strukturierung der Daten im Cloud Firestore ist stark von der Art des zu implementierenden Anwendungsfalls abhängig. Die Variante *Pseudo Relational* diente für diese Arbeit als Referenz, welche zwar exzellente Werte im Bereich der Writes lieferte, sich in allen anderen Bereichen jedoch geschlagen geben musste. Grundsätzlich empfiehlt sich das Szenario *Denormalized* als gesunder Mittelweg, der die Faktoren Kosteneffizienz, Performance und Komplexität in ausgewogenem Maße vereint. Ihre Stärken kann die Variante vor allem in Usecases ausspielen, die das Lesen und Schreiben unaggregierter Basisdaten vorsehen. Für bestehende Anwendungen, die bisher auf dem pseudo relationalen Ansatz aufbauten, gestaltet sich der Migrationspfad zudem im Vergleich zu den auf Views basierten Szenarien deutlich einfacher, da Abfragen hier nicht von Grund auf neu gedacht, sondern lediglich vereinfacht und Funktionen zur Denormalisierung hinzugefügt werden müssen. Schwächen im Vergleich zur Referenz zeigen sich lediglich in den Kennzahlen *Write Performance* und *Write Costs*, allerdings liefert *Denormalized* unter den aufbereitenden Szenarien nach wie vor die besten Ergebnisse. Zudem hängt die wahrgenommene Latenz stark davon ab, welche Basiscollection aktualisiert wird. Anwendungsfälle, die am häufigsten „weiter“ von der angezeigten Entität entfernt operieren (bspw. *Compound*), profitieren daher im Zweifel weniger von der Denormalisierung als solche, die die Root Entity behandeln. Besonders bei mittelmäßig volatilen Daten sowie Interaktionen mit administrativen Dokumenten kann *Denormalized* also seine Stärken in den Vordergrund stellen und die Nachteile hinreichend kaschieren.

Für Anwendungsfälle, die auf einem annähernd statischen Datenbestand aufsetzen, sind die auf *Bucketing* bzw. *Aggregation* aufgebauten Methoden *Shared View* sowie *Dedicated View* besser geeignet. Das können bspw. monatliche Reports als Entscheidungsgrundlage für das Management oder kompliziert berechnete Kennzahlen, die sich aus der Grundmenge der Daten ergeben, sein. Für den effizienten Einsatz der Szenarien ist die Akzeptanz einer Verzögerung der Datenaktualisierung eine Grundvoraussetzung. Dabei sollte das Entwicklungsteam sich Gedanken über die Art des Triggers machen, der die Aufbereitung steuert. Eine von Firestore Writes unabhängige Aktualisierung bspw. mithilfe einer zeitbasierten Schaltung, hat das Potenzial, die Schwächen der Methoden auszumerzen. Die Implementierung des in dieser Arbeit untersuchten Prototyps basiert auf Firestore Triggern und lässt *Write Costs* und *Datageneration Costs* potenziell ins

Unermessliche steigen. Für den operativen Umgang mit Nutzdaten sind die Szenarien also weniger geeignet, da die *Write Performance* noch stärker leidet als bei *Denormalized*, während die *Read Costs* zwar etwas niedriger ausfallen, das die verlorenen Punkte bei den restlichen Kosten sowie Komplexität jedoch nicht ausreichend kompensieren kann. Zudem schwebt die in Kapitel 5.2 beleuchtete Problematik bzgl. der Limitierung des Speicherplatzes von Dokumenten wie ein Damoklesschwert über den Methoden. Problematisch ist an dieser Stelle, dass dieser Umstand erst im laufenden Betrieb bei ausreichend großen Datenmengen offensichtlich wird, sodass Wartung und Weiterentwicklung der Anwendung erschwert werden und im Worst Case eine Neukonzeptionierung erforderlich macht. Zudem kann das ein junges Startup, dessen App durch eine glückliche Fügung plötzlich ernorme Aufmerksamkeit erfährt und hohe Kosten verursacht, in eine existenzbedrohende Lage versetzen. Die Migration der aufbereiteten Daten in neue, kleiner dimensionierte Buckets kann aufwändig und teuer sein und führt zudem nicht in einen nachhaltigen Betrieb, da das Problem so nur auf einen späteren Zeitpunkt verschoben wird. Außerdem sorgen kleinere Aggregate dafür, dass der Vorteil in *Read Costs* und *Read Performance* im Vergleich zu *Denormalized* sukzessive schwindet. Ein Entwicklungsteam sollte sich daher also nicht von den vermeintlich großen Vorteilen im Bereich der Reads blenden lassen und auf Views basierte Szenarien nur einsetzen, wenn der Anwendungsfall es erlaubt.

Kapitel 6

Fazit

Im Rahmen dieser Arbeit wurde erfolgreich ein Experiment durchgeführt, um die Frage, welche Ansätze zur effizienten Verwendung des Google Cloud Firestores für bestehende Anwendungen geeignet sind, zu beantworten. Dabei wurden vier verschiedene Szenarien definiert, die unterschiedliche Methoden in der Strukturierung und Behandlung von Daten aufwiesen. Die Varianten wurden im Rahmen der Methodik auf Faktoren in den Bereichen Kosteneffizienz, Performance sowie Komplexität untersucht und miteinander verglichen. Dabei wurde deutlich, dass ein relationales Mindset die Verwendung des Cloud Firestores unverhältnismäßig teuer gestaltet und vor allem bei leseintensiven Anwendungsfällen für eine unterdurchschnittliche Performance sorgt.

Besser geeignet sind Konzepte, die Daten zumindest denormalisieren, wenn nicht sogar aggregieren bzw. mithilfe von Bucketing aufbereiten. Bei der Analyse der Ergebnisse wurde offensichtlich, dass es keine allgemeingültige, vom Usecase unabhängige Lösung gibt, sondern dass die untersuchten Verfahren ihre Stärken in Abhängigkeit eines konkreten Anwendungsfalls ausspielen können. So eignet sich simple Denormalisierung vor allem für solche Anwendungen, deren operative Nutzdaten häufig gelesen und geschrieben werden, wobei administrative Daten seltener Schreibzugriffe erfahren. In diesem Fall halten sich die analysierten Kennzahlen die Waage, sodass die Anwendung relativ risikoarm betrieben werden kann. Zudem gestaltet sich ein Migrationspfad ausgehend vom pseudo relationalen Vorgehen hin zur Denormalisierung vergleichsweise unkompliziert und weniger aufwändig.

Szenarien, die sich dem Bucketing bzw. der Aggregation bedienen, sollten mit Vorsicht behandelt werden, da der Firestore aufgrund von Limitierungen in der Dokumentgröße nicht unbegrenzt skalieren kann. Diese Techniken sind also vor allem für Reports oder punktuell aggregierte Daten geeignet, vor allem wenn der Datenbestand entweder von

statischer Natur ist oder der Aufbereitungsprozess von den Firestore Write Events getrennt wird. Aggregationen können auch bei sehr volatilen Daten effizient verwendet werden, wenn nicht jeder Write in den Quellcollections eine Cloud Function triggert, sondern die Verdichtung bspw. in festen Zeitintervallen durchgeführt wird. Es ist essenziell, dass Entwicklungsteams sich der Implikationen bzgl. der Limits bewusst sind und diese Techniken tendenziell eher für unkritische Bereiche einer Anwendung bzw. für unabhängig mit der Nutzerbasis wachsende Datenbestände vorsehen. Auf diese Weise wird das Risiko minimiert, im laufenden Betrieb an Grenzen zu stoßen, die die Verwendung der App einschränken.

Die Ergebnisse dieser Arbeit decken sich mit den von Google propagierten Empfehlungen zur Verwendung des Cloud Firestores. Wie in Kapitel 2.2.5 beschrieben rät der Hersteller auch dazu, Daten nicht strikt relational zu strukturieren, sondern sich bekannter Techniken wie Denormalisierung zu bedienen. Dabei wird darauf hingewiesen, dass es essenziell ist, sich der Beschaffenheit der eigenen Daten und den Anforderungen des Anwendungsfalls an diese bewusst zu sein.

Diese Arbeit hat das Konzept der *Pagination*, bei der die Anzahl der abgerufenen Dokumente durch die Anwendung zunächst limitiert und bei Bedarf erhöht wird, nicht betrachtet. Die Denormalisierung kann im Vergleich zu den aggregierenden Varianten sehr einfach durch diese Technik erweitert werden, da hier nach wie vor auf die Root Collection zugegriffen wird und die Anzahl der gelesenen Firestore Dokumente stets der Anzahl der real abgerufenen Entitäten entspricht. Diese Erweiterung hat das natürliche Potenzial, die Kosten für Reads zu minimieren sowie die Performance deutlich zu verbessern und somit den Vorteil der anderen Szenarien aufzuwiegen. Auch für bestehende Apps, die auf dem relationalen Modell aufbauen, kann *Pagination* ein erster Schritt zur Verbesserung der User Experience sein, bevor Techniken wie Denormalisierung implementiert werden, da auch dort stets auf Root Collections zugegriffen wird und die Sparpotenziale enorm sind.

Die Ergebnisse haben gezeigt, dass denormalisierende und aggregierende Techniken naturgemäß mehr Quellcode erfordern als Szenarien, die ohne solche Methoden auskommen. Auch wenn die Abfragelogik in der Browseranwendung dadurch deutlich vereinfacht werden konnte, wäre eine Betrachtung der Frage interessant, wie diese zusätzliche Komplexität auf Seiten des Servers nachhaltig beherrscht werden kann. Erfahrungsgemäß ist es sehr einfach, neue Cloud Functions hinzuzufügen, allerdings kann diese breite Aufteilung der Business Logic schnell zu einer unübersichtlichen Codebasis führen. Dieser Aspekt wurde von dieser Arbeit lediglich durch Kennzahlen im Bereich der Komplexität betrachtet, allerdings findet sich hier sicherlich noch Raum für eine genauere Analyse.

Quellenverzeichnis

Literatur

Date, C. J. (1981). *An Introduction to Database Systems, 3rd Edition*. Addison-Wesley.

Maier, D. (1983). *The theory of relational databases*. Computer Science Press. https://sigmod.org/publications/dblp/db/books/dbtext/Maier83.html

Meier, A., & Kaufmann, M. (2019). *SQL & NoSQL Databases: models, languages, consistency options and architectures for Big Data Management*. Springer Vieweg.

Online-Quellen

Cockroach Labs. (2023, 14. August). *Architecture overview* [CockroachDB docs]. Verfügbar 14. August 2023 unter https://www.cockroachlabs.com/docs/stable/architecture/overview

Gartner. (2022, 27. Mai). *Definition of big data - gartner information technology glossary* [Gartner]. Verfügbar 21. Mai 2022 unter https://www.gartner.com/en/information-technology/glossary/big-data

Google. (2022a, 9. Februar). *Cloud firestore data model | firebase* [Cloud firestore data model]. Verfügbar 4. September 2022 unter https://firebase.google.com/docs/firestore/data-model

Google. (2022b, 7. Juli). *Firestore: NoSQL-Dokumentendatenbank* [Google Cloud]. Verfügbar 7. Juli 2022 unter https://cloud.google.com/firestore?hl=de

Google. (2023a). *Pricing | firestore* [Firestore pricing]. Verfügbar 12. August 2023 unter https://cloud.google.com/firestore/pricing

Google. (2023b, 8. März). *Choose a data structure | firebase* [Choose a data structure]. Verfügbar 7. August 2023 unter https://firebase.google.com/docs/firestore/manage-data/structure-data

Google. (2023c, 8. März). *Count documents with aggregation queries | firebase* [Count documents with aggregation queries]. Verfügbar 7. August 2023 unter https://firebase.google.com/docs/firestore/query-data/aggregation-queries

Google. (2023d, 8. März). *Get started with cloud firestore security rules | firebase* [Get started with cloud firestore security rules]. Verfügbar 7. August 2023 unter https://firebase.google.com/docs/firestore/security/get-started

Google. (2023e, 8. März). *Writing conditions for cloud firestore security rules | firebase* [Writing conditions for cloud firestore security rules]. Verfügbar 7. August 2023 unter https://firebase.google.com/docs/firestore/security/rules-conditions

Google. (2023f, 8. November). *Perform simple and compound queries in cloud firestore | firebase* [Perform simple and compound queries in cloud firestore | firebase]. Verfügbar 14. August 2023 unter https://firebase.google.com/docs/firestore/query-data/queries

Google. (2023g, 8. November). *Transactions and batched writes | firestore* [Transactions and batched writes | firestore]. Verfügbar 14. August 2023 unter https://firebase.google.com/docs/firestore/manage-data/transactions

Ilya Katsov. (2012, 1. März). *NoSQL data modeling techniques* [Highly scalable blog]. Verfügbar 29. Mai 2023 unter https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/

Jeff Delaney. (2018, 2. Mai). *Advanced Data Modeling with Firestore by Example* [Fireship]. Verfügbar 29. Mai 2023 unter https://fireship.io/lessons/advanced-firestore-nosql-data-structure-examples/

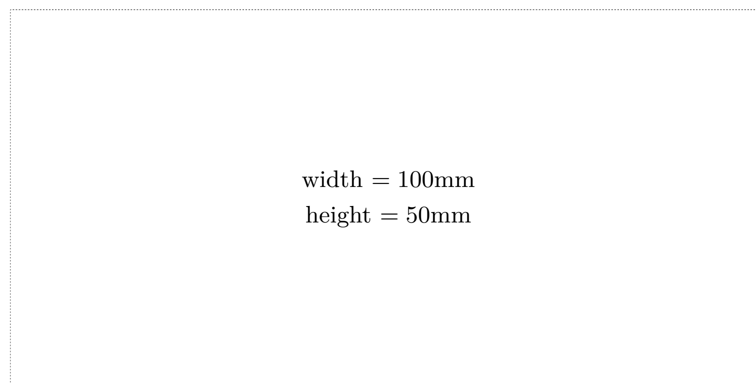
Nicolás Contreras V. (2018, 23. Juli). *How we spent 30k USD in firebase in less than 72 hours | HackerNoon* [Hackernoon]. Verfügbar 29. Mai 2023 unter https://hackernoon.com/how-we-spent-30k-usd-in-firebase-in-less-than-72-hours-307490bd24d

solidIT consulting & software development GmbH. (2022, 23. März). *DB-Engines Ranking - die Rangliste der populärsten Datenbankmanagementsysteme*. Verfügbar 27. März 2022 unter https://web.archive.org/web/20220323025428/https://db-engines.com/de/ranking

solidIT consulting & software development GmbH. (2023, 5. Januar). *DB-Engines Ranking pro Datenbankmodell Kategorie*. Verfügbar 5. Januar 2023 unter https://db-engines.com/de/ranking_categories

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —