HOCHSCHULE HEILBRONN

Bachelorthesis
Angewandte Informatik (SPO1a)

GraphQL und REST im Kontext relationaler und graphbasierter Datenbanken hinsichtlich Latenz bei unterschiedlichen Anfragekomplexitäten

Robin Hefner*

3. Januar 2025

Eingereicht bei Prof. Dr. Fankhauser

*206488, rohefner@stud.hs-heilbronn.de

Inhaltsverzeichnis

Abkürzungsverzeichnis

API: Application Programming Interface

DBMS: Database Management System

DI: Dependency Injection

HTTP: Hypertext Transfer Protocol

RDBMS: relationales Datenbankmanagementsystem

REST: Representational State Transfer

SQL: Structured Query Language

URL: Uniform Resource Locator

Abbildungsverzeichnis

Abstract

This thesis analyses the performance of GraphQL and REST in the context of relational and graph-based databases in terms of latency for different query complexities. The aim of the analysis was to evaluate the advantages and disadvantages of both API technologies and to understand the interactions with the underlying database architectures.
The results show that GraphQL offers clear advantages for complex queries, as it reduces the number of API calls through client-driven data queries. For bulk queries with large amounts of data, REST was able to achieve better response times, while GraphQL was superior for hierarchical and flexible queries. In terms of database technologies, relational databases proved to be efficient with structured data and low branching, while graph databases were particularly convincing with highly networked data and high query complexity thanks to their traversal mechanisms.
The combination of GraphQL and graph databases showed a significant reduction in latency in complex scenarios. For simpler queries, GraphQL was also able to achieve better latencies with a relational database than a comparable REST API. The findings provide practical information for selecting the optimal API and database technology and offer guidance for the development of high-performance and scalable systems.
Keywords: REST, GraphQL, API, relational database, graph database

Zusammenfassung

Die vorliegende Arbeit untersucht die Performance von GraphQL und REST im Kontext von relationalen und graphbasierten Datenbanken hinsichtlich der Latenz bei unterschiedlichen Anfragekomplexitäten. Ziel der Analyse war es, die Vor- und Nachteile beider API-Technologien zu bewerten und die Wechselwirkungen mit den zugrunde liegenden Datenbankarchitekturen zu verstehen.
Die Ergebnisse zeigen, dass GraphQL bei komplexen Abfragen deutliche Vorteile bietet, da es durch clientgesteuerte Datenabfragen die Anzahl der API-Aufrufe reduziert. Bei Bulk-Abfragen mit großen Datenmengen konnte REST bessere Antwortzeiten erzielen, während GraphQL bei hierarchischen und flexiblen Abfragen überlegen war. Hinsichtlich der Datenbanktechnologien erwiesen sich relationale Datenbanken als effizient bei strukturierten Daten und geringen Verzweigungen, während Graphdatenbanken durch ihre Traversal-Mechanismen besonders bei stark vernetzten Daten und hohen Abfragekomplexitäten überzeugten.
Die Kombination aus GraphQL und Graphdatenbanken zeigte eine signifikante Reduzierung der Latenz bei komplexen Szenarien. Bei einfacheren Anfragen konnte GraphQL ebenfalls mit einer relationalen Datenbank bessere LAtenzen erzielen als eine vergleichbare REST API. Die gewonnenen Erkenntnisse liefern praxisrelevante Hinweise für die Auswahl der optimalen API- und Datenbanktechnologie und bieten Orientierungshilfen für die Entwicklung leistungsstarker und skalierbarer Systeme.
Stichwörter: REST, GraphQL, API, relationale Datenbank, Graphdatenbank

1 Einleitung

1.1 Motivation

In der modernen Softwareentwicklung sind APIs (Application Programming Interfaces) zentral für die Verbindung und den Austausch zwischen verschiedenen Diensten und Anwendungen. Traditionell war REST (Representational State Transfer) die bevorzugte Methode, um APIs zu erstellen. Mit der Einführung von GraphQL, einer von Facebook entwickelten Abfragesprache, haben Entwickler jedoch eine weitere Option, die zunehmend an Bedeutung gewinnt.

Die Wahl der passenden API-Architektur hängt eng mit der eingesetzten Datenbanktechnologie zusammen, da diese die Effizienz und Flexibilität der Implementierung stark beeinflusst. Relationale Datenbanken, die auf strukturierten Tabellen basieren, gelten als etabliert und ermöglichen präzise Datenabfragen mit hoher Zuverlässigkeit. In Kombination mit REST bieten sie eine klare und stabile Struktur für den Zugriff auf Daten. GraphQL hingegen bietet die Möglichkeit, gezielt nur die benötigten Daten abzufragen, was besonders bei komplexen Modellen vorteilhaft sein kann.

Für Anwendungen, die stark vernetzte Daten verarbeiten, bieten Graphdatenbanken eine natürliche Grundlage. In Verbindung mit GraphQL lassen sich komplexe Abfragen effizient umsetzen, da die Technologie gut auf die Eigenschaften solcher Datenbanken abgestimmt ist. Auch mit REST können Graphdatenbanken genutzt werden, allerdings ist hier oft zusätzlicher Aufwand nötig, um die Daten in eine geeignete Form zu bringen.

Die Entscheidung zwischen REST und GraphQL sowie der passenden Datenbanktechnologie hat weitreichende Konsequenzen für die Entwicklung und den Betrieb einer Anwendung. Unternehmen sollten sorgfältig prüfen, welche Kombination aus API-Ansatz und Datenbank ihren Anforderungen am besten entspricht – sei es in Bezug auf Leistung, Skalierbarkeit oder Flexibilität.

1.2 Forschungsfragen

Nachfolgend sollen die Forschungsfragen vorgestellt werden, die aus der Motivation abgeleitet wurden. Diese dienen als Grundlage der Forschung für diese Arbeit.

- **FF-1: Wie unterscheiden sich GraphQL und REST hinsichtlich der Latenzzeit bei unterschiedlichen Anfragenkomplexitäten ?** Diese Frage zielt darauf ab, die Performance beider Systeme unter variablen Bedingungen zu vergleichen. Beispielsweise könnte untersucht werden, wie schnell eine API auf eine einfache Datenabfrage reagiert, im Vergleich zu einer komplexeren, die mehrere Abhängigkeiten involviert. Diese Untersuchung könnte Einblicke in die Effizienz der beiden Technologien bieten und somit als Entscheidungshilfe für Entwickler dienen, die die beste Lösung für ihre spezifischen Bedürfnisse auswählen möchten.
- **FF-2: Wie beeinflussen graph- und relationale Datenbanken die Latenz von REST- und GraphQL-APIs ?** Hierbei soll der Einfluss der zugrundeliegenden Datenbanktechnologien auf die Latenzzeiten von API-Anfragen untersucht werden. Dabei wird speziell betrachtet, wie sich die Wahl einer graphbasierten Datenbank im Vergleich zu einer relationalen Datenbank auf die Antwortzeiten der APIs auswirkt. Ziel ist es, herauszufinden, wie verschiedene Datenbankmodelle die Effizienz der API-Interaktionen beeinflussen und welche Datenbanktechnologie die besten Latenzwerte für unterschiedliche Anwendungsfälle bietet.

1.3 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, die Leistungsfähigkeit und Effizienz von REST- und GraphQL-APIs im Zusammenspiel mit relationalen und graphbasierten Datenbanken zu untersuchen und miteinander zu vergleichen. Im Mittelpunkt steht dabei die Analyse, wie sich die Wahl der API-Architektur und der zugrunde liegenden Datenbanktechnologie auf die Latenzzeiten und die Effizienz bei Anfragen unterschiedlicher Komplexität auswirken. Es soll aufgezeigt werden, welche Wechselwirkungen zwischen API-Architektur und Datenbanktechnologie bestehen und wie diese die Leistungsfähigkeit von modernen API-Systemen beeinflussen. Mithilfe der definierten Forschungsfragen wird eine Analyse durchgeführt, die praxisrelevante Einsichten für die Implementierung effizienter API-Systeme liefert.

1.4 Vorgehensweise

Die Untersuchung basiert auf einer Kombination aus theoretischen Analysen und empirischen Experimenten, um die beiden Forschungsfragen zu beantworten. Zunächst erfolgt eine umfassende Literaturrecherche, die bestehende Studien zu den Performance-Unterschieden zwischen GraphQL und REST sowie die Auswirkungen von graph- und relationalen Datenbanken auf die Latenz von API-Anfragen behandelt. Ziel ist es, ein fundiertes Verständnis der Performance-Differenzen beider Technologien zu entwickeln und herauszufinden, wie unterschiedliche Datenbanktechnologien die Antwortzeiten beeinflussen. In den empirischen Experimenten werden APIs sowohl mit REST als auch mit GraphQL unter Verwendung von graph- und relationalen Datenbanken implementiert. Dabei werden Performance-Tests durchgeführt, um die Anfrage- und Antwortzeiten bei verschiedenen Anfragenkomplexitäten zu messen. Ziel der empirischen Untersuchung ist es, herauszufinden, wie die Wahl der Datenbanktechnologie die Latenz der API beeinflusst und welche Kombination aus API-Technologie und Datenbank für unterschiedliche Anwendungsfälle die besten Performance-Werte bietet. Diese Ergebnisse können praxisrelevante Erkentnisse liefern, um die geeignetste Technologie für spezifischen Anforderungen auszuwählen.

2 Grundlagen

Die folgenden Abschnitte sollen die theoretischen Grundlagen vermitteln, die notwendig sind, um das Thema dieser Thesis zu betrachten. Die Konzepte, die hier beschrieben werden sind relationale Algebra, Graphentheorie, APIs, als auch relationale und Graph Datenbanken.

2.1 Relationale Algebra

Die relationale Algebra ist ein mathematisches System, welches 1970 von E.F. Codd entwickelt wurde. Sie wird unter anderem zu Abfrage und Mutation von Daten in relationalen Datenbanken verwendet wird. Durch sie wird eine Menge an Operationen beschrieben, die auf die Relationen angewendet werden können, um neue Relationen zu bilden. [3]

2.1.1 Basisrelation

Die Anwendung relationaler Algebra erfordert die Verwendung von Basisrelationen als fundamentale Bausteine, um mittels Grundoperationen komplexe Ausdrücke zu konstruieren, welche neue Relationen definieren. Eine Basisrelation setzt sich aus drei Komponenten zusammen: Tupel, Attributen und Domänen. Tupel reflektieren die Zeilen einer Tabelle, welche die einzelnen Datensätze repräsentieren. Diese werden durch Attribute in spezifische Spalten eingeteilt, welche die Eigenschaften der Tupel beschreiben. Die Wertebereiche, die für die einzelnen Attribute zulässig sind, werden als Domänen bezeichnet. Demnach ist jede Relation eine Menge von Tupeln mit spezifischen Attributen und deren Domänen. [18]

2.1.2 Grundoperationen

Grundoperationen in der relationalen Algebra sind einfache mengentheoretische Operationen, die auf die Basisrelationen angewandt werden. Insgesamt gibt es sechs Grundoperationen, die nachfolgend erläutert werden.

- Bei der **Selektion** σ werden die einzelnen Tupel basieren auf einer Bedingung gefiltert. Ein Beispiel hierfür wäre σ Alter > 30 (Person) . Hierdurch werden nur Personen mit einem Alter von mehr als 30 Jahren zurückgeliefert.
- Die **Projektion** π ermöglicht es bestimmte Attribute einer Relation auszuwählen oder zu entfernen. Beispielsweise kann durch π Name, Alter (Person) , nur der Name und das Alter einer Person zurückgegeben werden.

- Das **Kartesisches Produkt** $\times$ kombiniert jede Zeile der ersten Relation mit jeder Zeile der zweiten Relation. Somit erzeugt $R \times S$ alle möglichen Kombinationen aus R und S.
- Eine **Vereinigung** $\cup$ verknüpft die Tupel zweier Relationen, die eine gleiche Struktur aufweisen. $R \cup S$ kombiniert somit alle Tupel aus beiden Relationen mit gleicher Struktur, ohne Duplikate zu erzeugen.
- Die **Differenz** $\setminus$ zweier Relationen liefert alle Tupel, die in der ersten Relation vorkommen, aber nicht in der Zweiten. Sinngemäß gibt $R \setminus S$ alle Tupel R aus die nicht in S enthalten sind.
- Der **Schnitt** $\cap$ findet die Tupel, die in beiden Relationen vorhanden sind. Somit gibt $R \cap S$ die Tupel, die sowohl in R als auch in S entahlten sind zurück.

Durch die Verbindung dieser Grundoperationen können andere Operationen, wie beispielhaft ein Theat-Join $\bowtie$, welcher durch eine Kombination aus kartesischem Produkt und Selektion alle Tupel zweier Relationenen aufgrund einer Bedingung miteinander verbindet, erstellt werden. [18]

2.2 Graphentheorie

Ein Graph besteht im allgemeinen aus Knoten und verbindenden Kanten (vgl. Abb 1). In der Informatik bietet diese Datenstruktur einen großen Vorteil gegenüber der relationalen Algebra, wenn es sich um stark verzweigte Daten handelt. [17]

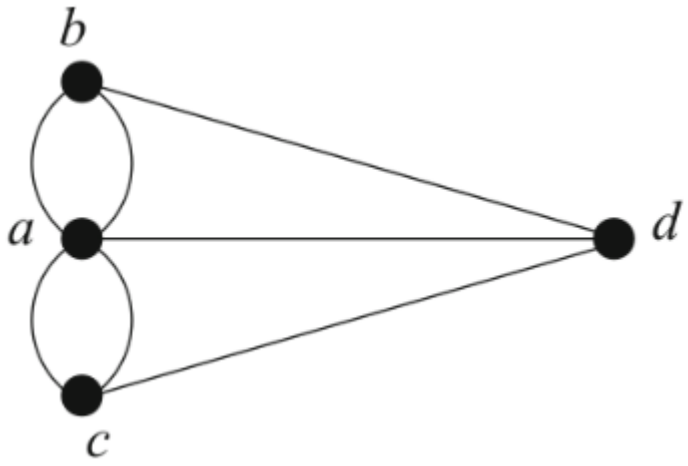

Abbildung 2.1: Modell eines ungerichteten Graphen. [17]

2.2.1 Knoten

Knoten sind Punkte innerhalb eines Graphen, die beispielsweise als Objekte der realen Welt verstanden werden können. Diese Objekte können zum Beispiel geographische Koordinaten sein, um einen Distanz-Graphen zu erstellen oder auch Webseiten, um einen Web-Graphen zu erhalten, der die Verbindung verschiedener Webseiten aufzeigt. Innerhalb der Knoten unterscheidet man verschiedene Typen. Zum einen gibt es Isolierte Knoten. Diese besitzen keine Verbindung zu anderen Knoten des Graphens und können zur Identifizierung nicht-verbundener Teile eines Netzwerks verwendet werden. Außerdem gibt es verbundene Knoten, welche mindestens eine Verbindung zu einem andern Knoten besitzen. Dadurch ergibt sich eine dirkete Verbindung zu einem benachbarten Knoten. Außerdem kann eine Indirekte Verbindung enstehen, indem ein Pfad zwischen Ausgangs und Endknoten existiert, der über mehrere Verbindungen führt. [17] [2]

2.2.2 Kanten

Kanten dienen in einem Graphen dazu, die Knoten miteinander zu Verbinden, um eine Relation zwischen ihnen zu visualisieren. Jede Kante besitzt einen Start- und Endknoten, der auch derselbe Knoten sein kann. Somit würde man in diesem Fall von einer Schleife sprechen. So wie es verschiedene Arten von Knoten gibt, exestieren auch verschiedene Kanten. In Abb. 2.1 sind *ungerichtete Kanten* im Graphen zu sehen. Sie verbinden die Knoten auf die trivialste Art, indem sie ohne Richtung, Gewicht oder andere Beschränkung eine Verbindung herstellen. Durch ungerichtete Kanten ensteht ein ungerichteter Graph. *Gerichtete Kanten* werden in Abb. 2.2 (a) genutzt. Diese werden durch einen Pfeil visualisiert. Dieser gibt an, in welcher Richtung der Graph eine Beziehung zwischen den Knoten herstellt. Es ist ebenfalls möglich, dass zwei Knoten eine beidseitige Beziehung durch zwei gerichtete Kanten, also eine Kante pro Richtung, eingehen. Ein Graph, der durch gerichtete Kanten verbunden ist wird gerichteter Graph genannt. Werden Zahlenwerte zu einer Kante hinzugefügt (vgl. Abb 2 (b)), so spricht man von *gewichteten Kanten*. Diese können verwendet werden, um die Strecke zwischen zwei Kanten darzustellen oder die Kosten für die Nutzung der Kante anzugeben. Wird ein Graph mit gewichteten Kanten verbunden, spricht man von einem gewichteten Graphen.[17] [2]

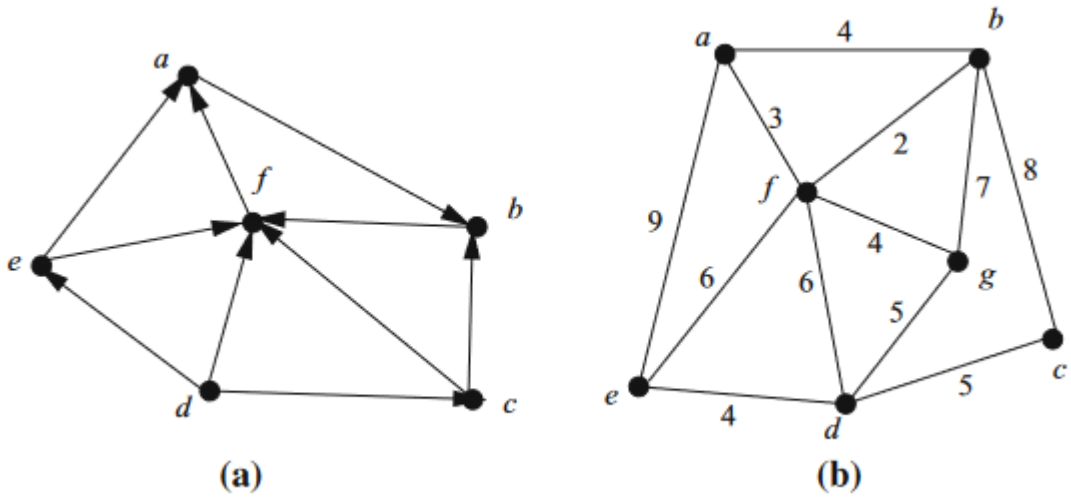

Abbildung 2.2: Modell eines gerichteten (a) und eines gewichtet Graphen (b). [17]

2.2.3 Traversierung

Die Bewegung von einem Knoten oder einer Kante zu einem anderen in einem Graphen nach vorgegebenen Kriterien wird als Traversierung bezeichnet. Graphdatenbanken bieten in der Regel Traversalmechanismen, um Daten von miteinander verbundenen Knoten effizient abzufragen und abzurufen. [15]

2.3 API

Nachfolgend werden die Grundlagen von APIs thematisiert. Hierbei werden die grundlegenden Definitionen sowie die Typen vorgestellt, die für diese Arbeit von Relevanz sind.

2.3.1 Definition API

Der Begriff „API“ steht für „Application Programming Interface“. Eine API bezeichnet eine Schnittstelle, welche Entwicklern den Zugriff auf Daten und Informationen ermöglicht. Bekannte Beispiele für häufig genutzte APIs sind die Twitter- und Facebook-APIs. Diese sind für Entwickler zugänglich und ermöglichen die Interaktion mit der Software von Twitter und Facebook. Zudem ermöglichen APIs die Kommunikation zwischen Anwendungen. Sie bieten den Anwendungen einen Weg, miteinander über das Netzwerk, überwiegend das Internet, in einer gemeinsamen Sprache zu kommunizieren. [11]

2.3.2 REST API

Representational State Transfer (REST) wurde erstmals im Jahr 2000 in einer Dissertation von Roy Fielding beschrieben. Hierbei handelt es sich um einen Software-Architekturstil für APIs. REST basiert auf einer Ressourcenorientierung, bei der jede Entität als Ressource betrachtet und durch eine eindeutige Uniform Resource Locator (URL) identifiziert wird. Die Architektur basiert auf sechs grundlegenden Beschränkungen, darunter die Client-Server-Architektur, bei der Client und Server unabhängig voneinander agieren. Ein wesentlicher Bestandteil von REST ist die Zustandslosigkeit, d. h. jede Anfrage beinhaltet sämtliche für die Verarbeitung erforderlichen Informationen, wodurch die Interaktion zwischen Client und Server vereinfacht wird. Die Umsetzung der CRUD-Operationen (Create, Read, Update, Delete) erfolgt durch die HTTP-Methoden (POST, GET, PUT, DELETE). REST nutzt das in HTTP integrierte Caching, um die Antwortzeiten und die Leistung zu optimieren. Dabei besteht die Möglichkeit, Serverantworten als cachefähig oder nicht cachefähig zu kennzeichnen. Des Weiteren ist eine einheitliche Schnittstelle zu nennen, welche die Interaktionen zwischen unterschiedlichen Geräten und Anwendungen erleichtert. Darüber hinaus erfordert REST ein mehrschichtiges System, bei dem jede Komponente lediglich mit der unmittelbar vorgelagerten Schicht interagiert. RESTful APIs, die diesen Prinzipien folgen, nutzen HTTP-Anfragen, um Ressourcen effizient zu bearbeiten. [6] [20]

2.3.3 GraphQL

GraphQL wurde 2012 von Facebook für den internen Gebrauch entwickelt und 2015 als Open-Source-Projekt veröffentlicht. Das Kernkonzept von GraphQL basiert auf clientgetriebenen Abfragen, bei denen der Client die Struktur der Daten präzise definiert und nur die tatsächlich benötigten Daten anfordert. Diese clientseitige Steuerung reduziert die Menge der übertragenen Daten und führt zu effizienteren Netzwerkaufrufen, da nur die relevanten Informationen übermittelt werden. Im Vergleich zu REST verursacht GraphQL signifikant weniger Overhead, was die Netzwerkperformance optimiert. Die hierarchische Struktur der Abfragen, die die Graph-Struktur widerspiegelt, ermöglicht eine intuitive und flexible Datenmodellierung. Die starke Typisierung in GraphQL wird durch ein Schema definiert, das die Typen der Daten spezifiziert. Dies sorgt für eine verbesserte Validierung der Abfragen und bietet eine klarere Dokumentation. Im Gegensatz zu REST, bei dem für verschiedene Operationen mehrere Endpunkte erforderlich sind, nutzt GraphQL nur einen einzigen Endpunkt für alle API-Abfragen, was die Komplexität auf der Serverseite reduziert und eine vereinfachte API-Verwaltung ermöglicht. [20]

2.4 Datenbank

Im Folgenden werden die Grundlagen von Datenbanken behandelt. Es werden grundlegende Definitionen im Zusammenhang mit Datenbanken und die verschiedenen Arten von Datenbanken vorgestellt.

2.4.1 Definition Datenbank und Datenbank Management System

Eine Datenbank stellt eine Sammlung von Daten und Informationen dar, welche für einen einfachen Zugriff gespeichert und organisiert werden. Dies umfasst sowohl die Verwaltung als auch die Aktualisierung der Daten. Die in der Datenbank gespeicherten Daten können nach Bedarf hinzugefügt, gelöscht oder geändert werden. Die Funktionsweise von Datenbanksystemen basiert auf der Abfrage von Informationen oder Daten, woraufhin entsprechende Anwendungen ausgeführt werden. DBMS bezeichnet eine Systemsoftware, die für die Erstellung und Verwaltung von Datenbanken eingesetzt wird. Zu den Funktionalitäten zählen die Erstellung von Berichten, die Kontrolle von Lese- und Schreibvorgängen sowie die Durchführung einer Nutzungsanalyse. Das DBMS fungiert als Schnittstelle zwischen den Endnutzern und der Datenbank, um die Organisation und Manipulation von Daten zu erleichtern. Die Kernfunktionen des DBMS umfassen die Verwaltung von Daten, des Datenbankschemas, welches die logische Struktur der Datenbank definiert, sowie der Datenbank-Engine, welche das Abrufen, Aktualisieren und Sperren von Daten ermöglicht. Diese drei wesentlichen Elemente dienen der Bereitstellung standardisierter Verwaltungsverfahren, der Gleichzeitigkeit, der Wiederherstellung, der Sicherheit und der Datenintegrität. [9]

2.4.2 Relationale Datenbank

Relationale Datenbanken basieren auf dem relationalen Modell, das von E.F. Codd entwickelt wurde und relationaler Algebra sowie Tuple-Relational-Kalkül zugrunde liegt. Sie speichern Daten in einer hochstrukturierten Tabellenform, wobei jede Tabelle aus Zeilen, den sogenannten Tupeln, und Spalten, den sogenannten Attributen, besteht. Jede Zeile repräsentiert einen Datensatz, während jede Spalte durch einen spezifischen Datentyp definiert ist. Die Struktur ermöglicht eine klare Organisation der Daten und erleichtert deren Verwaltung. Relationale Datenbanken verwenden Primär- und Fremdschlüssel, um Beziehungen zwischen Tabellen herzustellen und referenzielle Integrität zu gewährleisten, wodurch die Datenkonsistenz erhalten bleibt. Aufgrund dieser Eigenschaften werden die Tabellen häufig auch als "Relationen" bezeichnet.

Die bekanntesten RDBMS sind Microsoft SQL Server, Oracle MySQL und IBM DB2. Ein RDBMS organisiert alle Daten in tabellarischer Form und bietet Funktionen wie Primärschlüssel zur eindeutigen Identifikation von Datensätzen sowie Indizes, die die Geschwindigkeit der Datenabfragen erhöhen. Darüber hinaus unterstützen RDBMS die Erstellung virtueller Tabellen, die komplexe Abfragen vereinfachen, sowie einen kontrollierten Mehrbenutzerzugriff mit individueller Rechtevergabe. Die Verwendung einer standardisierten Sprache SQL ermöglicht die Verwaltung, Abfrage und Manipulation von Daten. Die genannten Merkmale machen relationale Datenbanken äußerst flexibel und benutzerfreundlich. Ein großer Vorteil relationaler Datenbanken liegt in ihrer Unterstützung der ACID-Prinzipien (Atomicity, Consistency, Isolation, Durability). Diese gewährleisten Stabilität und Sicherheit bei Transaktionen, was sie für viele Anwendungsbereiche geeignet macht. Darüber hinaus bieten sie eine hohe Datenintegrität, reduzieren Redundanz und ermöglichen die einfache Implementierung von Sicherheitsmaßnahmen. Trotz dieser Vorteile stoßen relationale Datenbanken bei bestimmten Anforderungen an ihre Grenzen. Sie sind oft nicht für hohe Skalierbarkeit geeignet und können mit dem exponentiellen Wachstum von Daten schwer umgehen. Die Einrichtung und Wartung solcher Systeme ist häufig kostspielig, und die Verwaltung unstrukturierter Daten wie Multimedia oder Social-Media-Inhalte stellt eine große Herausforderung dar. Zudem erschwert die tabellarische Struktur komplexe Datenverknüpfungen und die Integration mehrerer Datenbanken. Aufgrund dieser Einschränkungen haben moderne Anwendungen und Big-Data-Anforderungen zur Entwicklung von NoSQL-Datenbanken geführt, die besser auf die Verwaltung unstrukturierter und verteilter Daten ausgelegt sind. Dennoch bleiben relationale Datenbanken aufgrund ihrer Standardisierung, Benutzerfreundlichkeit und breiten Einsatzmöglichkeiten ein wesentlicher Bestandteil der Datenbanktechnologie. [7] [9]

2.4.3 Graphdatenbanken

Graphdatenbanken sind spezialisierte Datenbanksysteme, die das Datenmodell dem User in Form eines Graphen präsentieren. Darüber hinaus enthalten die Graphen Informationen über die Eigenschaften der Knoten und Kanten, wodurch eine flexible und schemafreie Speicherung semistrukturierter Daten ermöglicht wird. Abfragen in Graphdatenbanken werden häufig als Traversalen formuliert, was ihnen gegenüber relationalen Datenbanken erhebliche Geschwindigkeitsvorteile verschafft, insbesondere bei komplexen Beziehungsabfragen. Man unterscheidet zwischen nativen und nicht-nativen Graphdatenbanken. Native Graphdatenbanken sind speziell für das Graphmodell entwickelt und speichern die Daten intern in Graphenform. Nicht-native Graphdatenbanken hingegen verwenden andere Speichertechnologien, wie beispielsweise relationale Datenbanken, und präsentieren die Daten lediglich in Form von Graphen, um CRUD-Zugriffe zu ermöglichen. Beide Typen gehören zur Kategorie der NoSQL-Datenbanken, die durch den Verzicht auf das starre Tabellenschema relationaler Datenbanken deren Schwächen umgehen. Durch ihre besondere Architektur bieten Graph-

datenbanken nicht nur eine effiziente Verarbeitung von Beziehungsdaten, sondern erfüllen auch ACID-Bedingungen und unterstützen Rollbacks, was die Konsistenz der gespeicherten Informationen gewährleistet. Damit stellen sie eine leistungsstarke Alternative zu traditionellen relationalen Datenbanken dar, insbesondere in Anwendungsbereichen mit hochgradig verknüpften Daten. [9] [16]

3 Analyse

In diesem Kapitel sollen die in der Einleitung definierten Forschungsfragen untersucht werden. Um diese Fragen zu beantworten, wird eine Literaturanalyse durchgeführt, die bestehende Studien, Bücher und wissenschaftliche Artikel betrachtet. Ziel ist es, aus der vorhandenen Literatur systematisch Erkenntnisse abzuleiten, die die Performance der Ansätze in verschiedenen Szenarien beleuchten.

3.1 FF-1: Wie unterscheiden sich GraphQL und REST hinsichtlich der Latenzzeit bei unterschiedlichen Anfragenkomplexitäten?

Um zu untersuchen, wie GraphQL und REST sich im Bezug auf Latenz bei verschiedenen Anfragekomplexitäten auswirkt, muss man zunächst darauf eingehen, welche Faktoren oder Spezifikationen, die die APIS nutzen sich auf die Latenz auswirken. Bei der Bereitstellung von Daten setzt REST auf HTTP-Endpunkte. Diese untestützen nativ HTTP-Caching, wodurch Ressourcen in einem Cache zwischengespeichert werden können um unnötige Datenübertragungen und Serveranfragen zu vermeiden und somit die Zugriffszeiten zu verringern. Bei GraphQL wird dies nativ nicht unterstützt. Hierdurch können wiederkehrende Anfragen nicht gecached werden und müssen jeweils immer vom Server bearbeitet werden, wodurch eine höhere Zugriffszeit entsteht. [20]
REST fördert die Zerlegung von Systemen in eine Menge verknüpfter Ressourcen mit einem bestimmten Granularitätsgrad. Dies führt zu schwierigen Abwägungen zwischen Wiederverwendbarkeit und Leistung, die in der allgemeinen Software-Service-Architektur wohlbekannt sind. Weniger granulare und kohäsivere Services werden bevorzugt, da sie lose Kopplung und hohe Wiederverwendbarkeit fördern. Dies kann jedoch zu komplizierten Client-Server-Interaktionen führen, bei denen mehrere aufeinanderfolgende Anfragen notwendig sind, um die benötigten Daten aus dem Ressourcengraphen abzurufen,ein Phänomen, das als „Underfetching“ bekannt ist. Dieses Problem wird auch als n+1-Problem bezeichnet und tritt bei REST auf der Seite des Clients auf. Dieser muss demensprechend weitere Anfragen schicken, bis die benötigte Antwortzeiten führen. [13] [23]
Bei GraphQL kann es ebenso zu einem n+1 Problem kommen. Hierbei tritt dieses jedoch nicht auf der Client Seite auf sondern direkt beim Server bei der Verarbeitung der Anfrage. Der GraphQL Server muss dann mehrere Anfragen an die Datenbank schicken, um die benötigten Daten zu erhalten, um sie dann an den Client auszuliefern. [8]
Hierfür bietet GraphQL einen sogenannten Dataloader, der die Anfragen, die zur bearbeitung

eines Requests benötigt werden bündelt und als eine einzelne, optimierte, Datenbankabfrage ausführt. [14]
Zudem spielt bei der Latenz einer API die Anfragekomplexität eine entscheidende Bedeutung. In Abbildung 3.1 sind vergleichend die Ausführungszeiten von vier unterschiedlichen Api-Abfragen, die sowohl mit REST als auch mit GraphQL durchgeführt wurden, zu sehen. Die Abfragen wurden auf den öffentlich zugänglichen GitHub REST- und GraphQL-APIs durchgeführt, da beide dieselben Ressourcen bereitstellen. Performance wird hierbei in Millisekunden gemessen und gibt Aufschluss darüber, wie sich beide Technologien je nach Art der Anfrage verhalten.

Query	REST	GraphQL
Query 1	171.16	176.96
Query 2	627.00	606.34
Query 3	225.44	144.88
Query 4	338.16	388.46

```
Query 1: GET/user
Query 2: POST /repos/:owner/:repo/issues/:issue_number/comments
Query 3: GET/repos/:owner/:repo/issues/:issue_number
Query 4: GET/repos/:owner/:repo/stargazers
```

Abbildung 3.1: Latenz GraphQL vs. REST [20]

Die erste Abfrage (GET/user) zeigt, dass die Latenz zwischen REST und GraphQL sich nicht signifikant unterscheiden. REST benötigt für diese Anfrage 171,16 Millisekunde, während GraphQL mit 176,96 Millisekunden nur geringfügig langsamer ist. Dies lässt darauf schließen,dass bei einfachen Anfragen, die keine komplexe Datenstruktur beinhalten, sich die Ergebnisse nicht gravierent unterscheiden und als nahezu identisch bezeichnet werden können. Die zweite Abfrage (POST /repos/:owner/:repo/issues/:issue_number/comments), eine Schreiboperation, zeigt ein ähnliches Ergebniss. Hierbei ist GraphQL mit 606,34 Millisekunden leicht schneller als REST mit 627 Millisekunden. Da der Unterschied sehr gering ist, deutet dies daraufhin, dass Schreiboperationen in beiden Technologien ähnlich effizient verarbeitet werden. Eine deutlich Differenz zeigt sich bei der dritten Abfrage (GET/repos/:owner/:repo/issues/:issue_number). REST benötigt hierfür 225,44 Millisekunden, während GraphQL mit nur 144,88 Millisekunden deutlich schneller ist. Dies verdeutlicht die Stärke von GraphQL bei komoplexeren Datenanforderungen. Da GraphQL in der Lage ist, mehrere Datenpunkte in einem einzelnen Aufruf zu bündeln, kann es die Anzahl der API-Aufrufe reduzieren und so effizienter arbeiten. Die vierte Abfrage (GET/repos/:owner/:repo/stargazers) zeigt ein gegensätzliches Bild. Hier ist REST mit 338,16 Millisekunden schneller als GraphQL, welches 388,46 Millisekunden benötigt. Die liegt daran, dass die Abfrage keine komplexe Aggregation von Daten erfodert und der Overhead von GraphQL die Performance negativ beeinflusst. Somit kann man sagen, dass REST in einem Szenario besser abgeschnitten hat

als GraphQL, GraphQL in einem anderen Szenario besscer abschneidet als REST und beide Technologien in zwei Szenarien ähnlich abgeschnitten haben. [20]
Andere Studien zeigen, dass GraphQL bei einfachen Abfragen, die nur einen Endpunkt nutzen, etwa 0,02-mal schneller als REST arbeitet. [23]
Bei komplexeren Anfragen, die vier Endpunkte umfassen und 1000 Ergebnistupel liefern, verarbeitet eine GraphQL-API die Daten bis zu 16-mal schneller als eine REST-API.[12]
Noch anspruchsvollere Anfragen, die fünf oder mehr Endpunkte nutzen, werden von GraphQL sogar bis zu 187-mal schneller bearbeitet.[21]
Bei steigender Komplexität stellt sich jedoch bei einer Menge von 100.000 Ergebnistupeln heraus, dass eine GraphQL API in diesem Fall 0.36 [12] bis 2,5 mal langsamer ist, als eine vergleichbare REST API [1].

3.2 FF-2: Wie beeinflussen graph- und relationale Datenbanken die Latenz von REST- und GraphQL-APIs?

Da APIs nur eine Schnittstelle zwischen einer Datenbank und der Anwendung darstellt, welche die Daten aus der Datenbank konsumieren möchte, ist die Latenz einer API zwingend von der Bearbeitungszeit einer Anfrage innerhalb der Datenbank abhängig. Hierbei nutzen verschiedene Datenbanktypen unterschiedlichste Methoden, um die angefragen Daten aus der Datenbank bereitzustellen. In dieser Arbeit wird ein besonderes Aufgenmerk auf die Verarbeitung anhand der relationalen Algebra, sowie der Graphentheorie gelegt. Diese Methoden bieten bei verschiedenen Anwendungszenarien Vorteile, als auch Nachteile bei der Effizienten und Zeitsparenden bearbeitung von Anfragen. Hierbei zeigt sich bei realtionalen Datenbanken, dass diese bei steigender Anzahl an Daten, welche gespeichert werden zunehmend langsamere Performance bieten. Wie in Abbildung 3.2 jedoch zusehen ist steigt die Menge der Daten, welche im Internet verarbeitet werden, seit 2010 bis heute stetig an und die Prognosen bis ins Jahr 2028 deuten auf einen deutlichen Trend zu erheblich mehr Daten an. Graphdatenbanken können mit solch einem Wachstum deutlich besser umgehen, da durch die Nutzung von Traversal-basierdenden Anfragen, welche für diese Art von Abfragen optimiert sind, die Menge die Menge der Daten innerhalb der Datenbank eine untergordnete Rolle spielen. [9] [19]

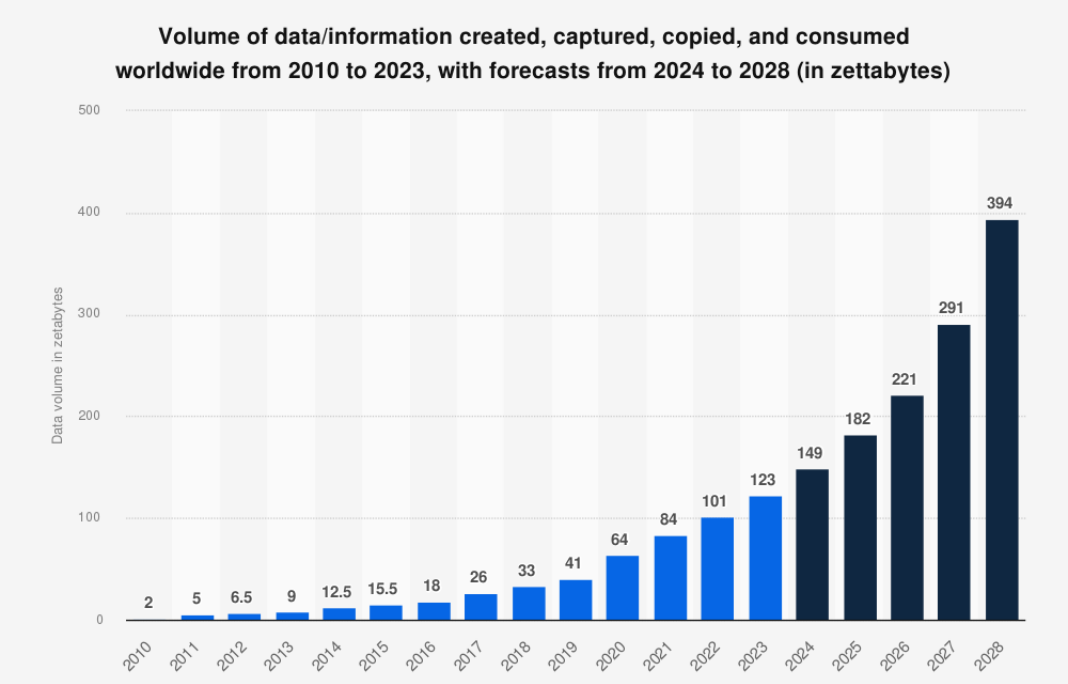

Abbildung 3.2: Datenmenge Weltweit [5]

Ein weiterer Faktor, welcher die Performance einer Datenbank beeinflusst, ist die Verbundenheit, also die Relationen, der Daten. Abbildung 3.3 zeigt, wie sich die Verbundenheit der Daten im Verlauf verändert hat. Hierbei ist deutlich zu sehen, dass Daten stetig eine größere Verzweigung zueienader aufweißen.
Eine Graphdatenbank kann mit diesem hohen Grad an Verzweigung, durch die Verwendung der Graphentheorie und Traversierung performant umgehen. Bei relationalen Datenbanken stellt diese hohe Grad der Relationen eine performance Einbuße dar. Da bei der Verbindung der Daten die Relationen zwischen der Tabellen durch Joins realisiert werden und diese aufwendig zu brechnen sind, steigt mit zuhnemender Anzahl der Relationen auch die Zeit, welche die Datenbank benötigt um mithilfe der Schlüssel und mehrerer Tabellen die Beziehung zwischen den Dateneinträgen zu konstruieren. [9] [16]

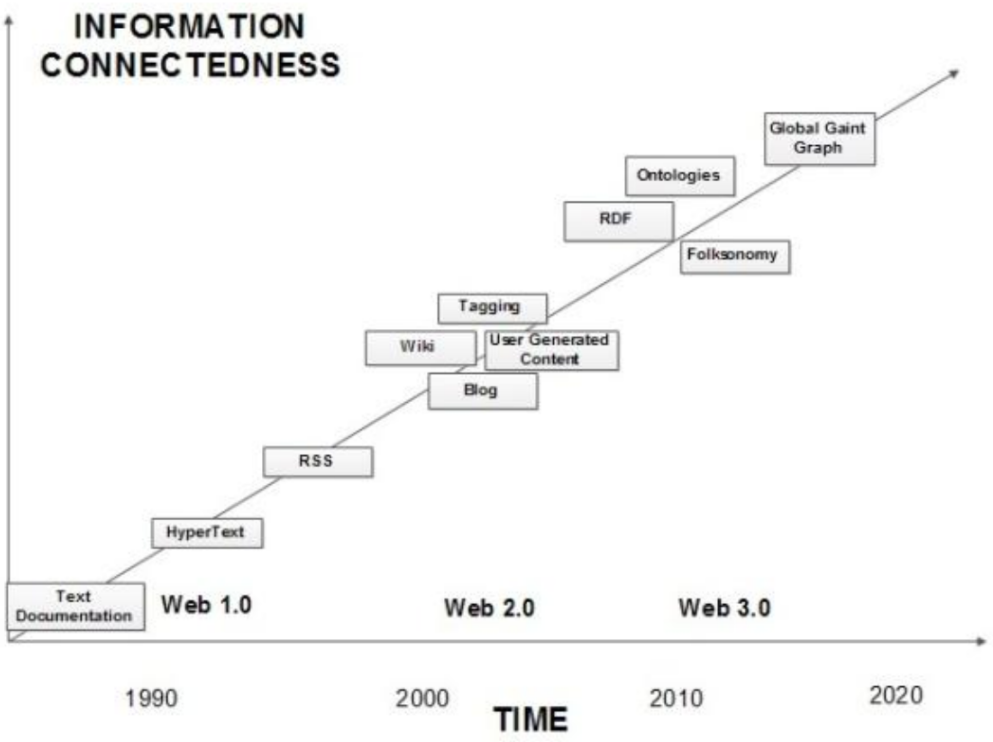

Abbildung 3.3: Informationsverbundenheit [19]

Die Universität von Mississippi führte ein Experiment durch, um die Latenzunterschiede zwischen einer relationalen MySQL-Datenbank und einer Graphdatenbank, konkret Neo4j, zu untersuchen. Dabei wurden sieben unterschiedliche Szenarien simuliert, deren Ergebnisse in den Abbildungen 3.4 und 3.5 dokumentiert sind. In der ersten Spalte der Tabelle wird die Konfiguration der Datenbank dargestellt. Hierbei gibt die Zahl die Anzahl der Nodes oder Tupel in der Datenbank an, gefolgt von einem Datentyp, der beschreibt, welche Art von Daten bei der Erstellung der Datenbank zufällig verwendet wurde. Die Ergebnisse beider Datenbanken sind in Abbildung 3.4 zu sehen, wobei die Abfragen I1 und I2 unterschiedliche Szenarien repräsentieren. Die Abfrage I1 selektiert alle Nodes, deren Payload einem bestimmten Wert entspricht, während I2 die Nodes zählt, deren Payload kleiner ist als ein gegebener Wert. Beide Szenarien nutzen den Datentyp Integer um die Daten anzufordern. Aus den experimentellen Daten wird deutlich, dass die relationale MySQL-Datenbank im Vergleich zur Graphdatenbank Neo4j eine deutlich geringere Latenz in jedem gestesteten Fall aufweist. [22]

Database	Rel I1	Neo I1	Rel I2	Neo I2
1000int	0.3	33.0	0.0	40.6
5000int	0.4	24.8	0.4	27.5
10000int	0.8	33.1	0.6	34.8
100000int	4.6	33.1	7.0	43.9

Abbildung 3.4: Ergebnisse Abfragen durch int in Millisekunden [22]

Abbildung 3.5 zeigt Abfragen, bei denen anhand eines Strings selektiert wurde. Die zweite Zeile gibt dabei an, welche Länge der String hatte, auf dessen Basis die Selektion vorgenommen wurde. Hierbei wird die Menge der Nodes gezählt, die den String enthalten. Die Ergebnisse zeigen ein deutlich anderes Bild im Vergleich zu den vorherigen Resultaten, bei denen mithilfe eines Integers selektiert wurde. Durch die Verwendung eines Strings kann die relationale Datenbank die Graphdatenbank lediglich im ersten Szenario übertreffen. In den komplexeren Fällen spielt die Graphdatenbank ihre Vorteile jedoch klar aus und ist bei anspruchsvolleren Anfragen in der Spitze um den Faktor 167 schneller. [22]

Database	Rel	Neo	Rel	Neo	Rel	Neo	Rel	Neo	Rel	Neo
	d=4	d=4	d=5	d=5	d=6	d=6	d=7	d=7	d=8	d=8
1000char8k	26.6	35.3	15.0	41.6	6.4	41.6	11.1	41.6	15.6	36.3
5000char8k	135.4	41.6	131.6	41.8	112.5	36.5	126.0	33.0	91.9	41.6
10000char8k	301.6	38.4	269.0	41.5	257.8	41.5	263.1	42.6	249.9	41.5
100000char8k	3132.4	41.5	3224.1	41.5	3099.1	42.6	3077.4	41.8	2834.4	36.4
1000char32k	59.5	41.5	41.6	42.6	30.9	41.5	31.9	41.4	31.9	35.4
5000char32k	253.4	42.3	242.9	41.5	229.4	35.3	188.5	38.5	152.0	41.5
10000char32k	458.4	36.3	468.8	41.6	468.3	41.6	382.1	41.5	298.8	36.3
100000char32k	3911.3	41.4	4859.1	33.3	6234.8	37.3	4703.3	41.5	2769.6	41.5

Abbildung 3.5: Ergebnisse Abfragen durch char in Millisekunden [22]

Zusammenfassend lässt sich feststellen, dass relationale Datenbanken bei der Selektion von Daten anhand eines Integers deutlich schnellere Ergebnisse liefern. Bei der Abfrage von Daten anhand eines Strings zeigt sich zunächst, dass relationale Datenbanken bei kleinen Datenmengen besser performen. Mit zunehmender Datenmenge wird jedoch die Graphdatenbank zunehmend effizienter und übertrifft schließlich die Performance der relationalen Datenbank deutlich.

4 Datenmodellierung

Um für das empirische Experiment eine Datenbasis zu schaffen benötigt es ein Datenmodell. Hierbei soll ein möglichst realistisches und flexibles Modell genutzt werden, zudem sollte dieses Verzweigungen aufweisen um verschiedene Abfragekomplexitäten abbilden zu können. In diesem Szenario (vgl. Abb. 3) handelt es sich um ein Projektmanagement-Tool. Es exestieren drei Klassen, welche über drei Beziehungen miteinander verbunden sind. Die Klasse Person modeliert einen Menschen, mit den Attributen Vorname, Nachname und E-Mail. Sie steht mit der Klasse Project in einer n:n-Beziehung, wodurch mehrere Projekte zu einer Person zugeordnet werden können, aber auch mehrere Personen an einem Projekt arbeiten können. In der Klasse Projekt werden nur der Titel und das Datum an welchem das Projekt erstellt wurde gespeichert. Ein Projekt steht in einer 1:n-Beziehung zur Klasse Issue. Dadurch kann einem Issue nur ein Projekt zugeordnet werden, ein Projekt kann aber mehrere Issues beinhalten. Issue speichert Daten wie etwa den Titel, das Erstellngsdatum, den Status und den Grund des Status des Issues. Issue besitzt eine n:n-Beziehung zu Person, wodurch ein Issue von mehreren Personen bearbeitet werden kann und eine Person in mehreren Issues arbeiten kann.

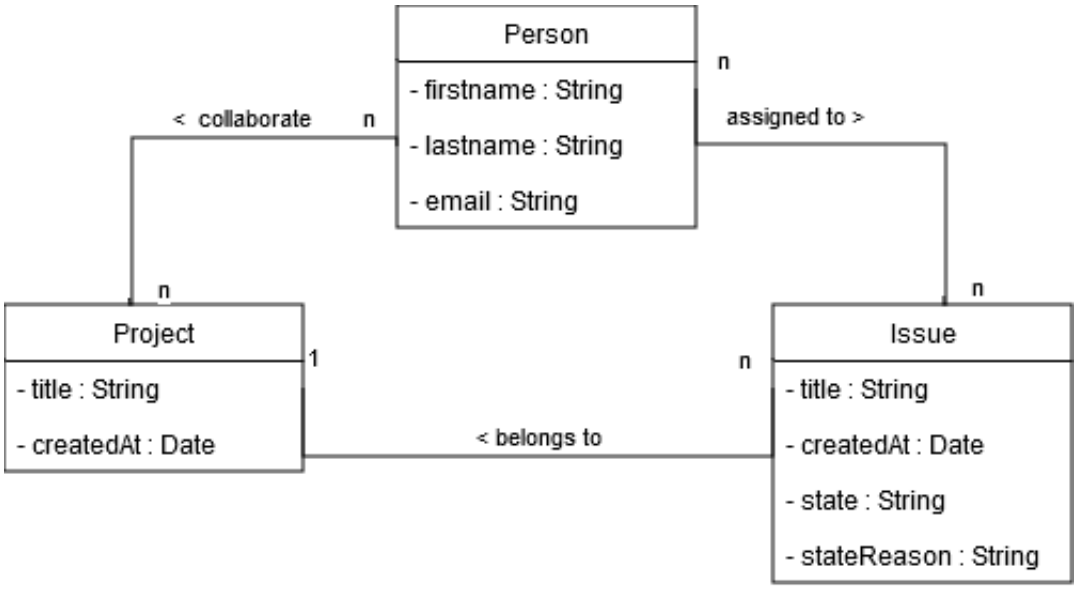

Abbildung 4.1: Klassendiagramm

5 Systemdesign

Innerhalb dieses Abschnitt sollen die konkreten Technologien beschrieben werden, die gewählt wurden um ein System zu entwickeln welches für Latentztests verwendet werden kann.

5.1 Datenbankdesign

Für die Durchführung des Experiments werden zwei Datenbanktypen verwendet, welche nachfolgend mit ihrer konkreten Konfiguration beschrieben werden.

5.1.1 Relationales Datenbankdesign

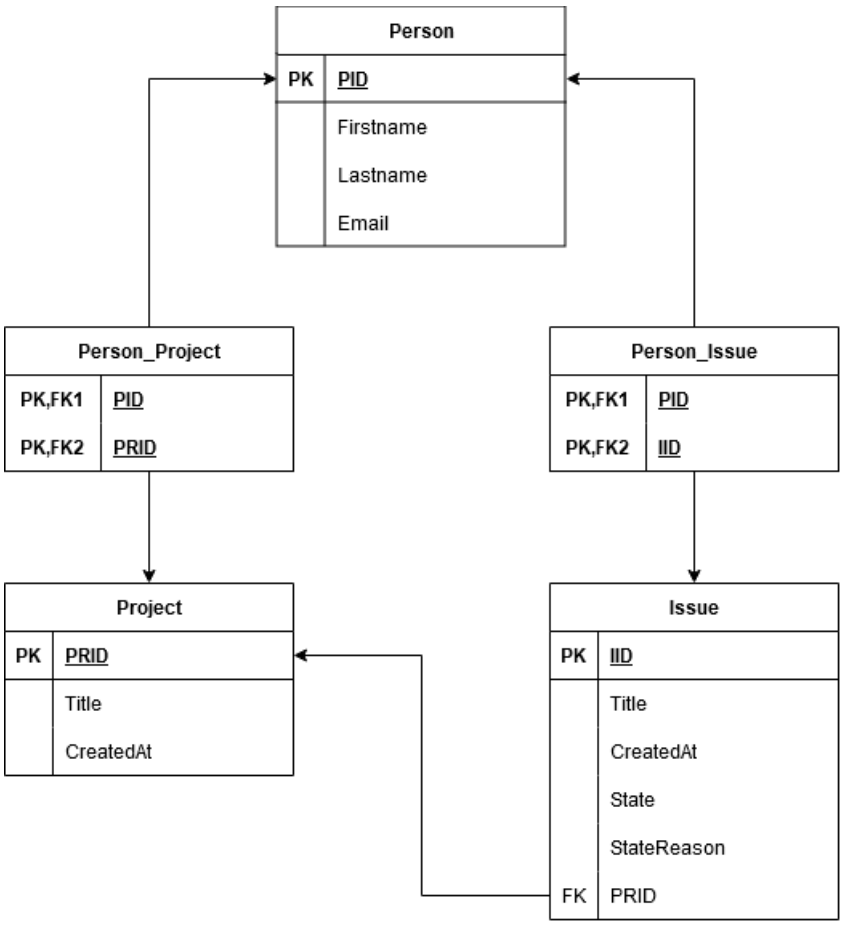

Abbildung 5.1: Tabellen Diagramm

Um eine realtionale Datenbank aus dem gegebenen Datenmodell (Abb. 4.1) zu erstellen muss dieses wie in Abb. 5.1 zu sehen ist angepasst werden, um die Beziehung zwischen Person und Projekt als auch Person und Issue abzubilden. Somit ergeben sich für die relationale Datenbank fünf Tabellen welche in einer PostgreSQL Datenbank realisiert werden. PostgreSQL wird verwendet, da es ein leistungsfähiges, objekt-relationales Datenbanksystem bietet welches Open-Source ist und dadurch kostenfrei zur Verfügung steht. Die Datenbank wurde mit Beispieldaten befüllt, welche in Abbildung 5.2 bis 5.6 beispielhaft zusehen sind. Hierbei wurden 500.000 Personen als auch Projekt und Issue Objekte in die Datenbank eingefügt. Durch die Verwendung von Zwischentabellen, wie person_issue und person_project, enthält die relationale Datenbank 2,5 Mio. Tupel, die einen Speicherbedarf von 215MB besitzen.

pid	firstname	lastname	email
1	Cecilla	Beningfield	cbeningfield9@wp.com

Abbildung 5.2: Tupel der Tabelle Person

pid	iid
1	4894

Abbildung 5.3: Tupel der Tabelle Person_Issue

iid	title	createdat	state	statereason	prid
1	Dabfeed	2023-06-10 00:00:00	Open	Assigned	586

Abbildung 5.4: Tupel der Tabelle Issue

pid	iid
1	714

Abbildung 5.5: Tupel der Tabelle Person_Project

prid	title	createdat
1	Asoka	2022-02-17 00:00:00

Abbildung 5.6: Tupel der Tabelle Project

5.1.2 Graphdatenbankdesign

Um eine Graphdatenbank zu erstellen benötigt man keine definierten Tabellen, da diese die Daten schemafrei speichert. Die Nodes werden bei der Erstellung entsprechend der Objekte benannt, ebenso werden die Beziehungen zwischen den Nodes bei der Erstellung benannt. Als Graphdatenbank wird auf neo4j zurückgegriffen. Es ist eine der am weitesten verbreiteten Graphdatenbanken und bietet eine hohe Benutzerfreundlichkeit. In Abbildung 5.7 ist eine Demonstration einer Beziehung zwischen jeweils einem Node zu sehen. Die Kanten sind als gerichtete Kanten abgebildet, dabei hat Person zwei ausgehende Kanten, Projekt zwei eingehende und Issue jeweils eine eingehende als auch eine ausgehende Kante. Wie in der relationalen Datenbank wurden auch in der Graphdatenbank 500.000 Nodes pro Objekt erstellt. Hierbei werden jedoch keine Zwischentabellen benötigt, um Beziehungen darzustellen, wodurch in der Datenbank 1,5 Mio. Nodes vorhanden sind, die durch 2,01 Mio. Edges verbunden sind. Hieraus ergibt sich eine Gesamtgröße von 197MB.

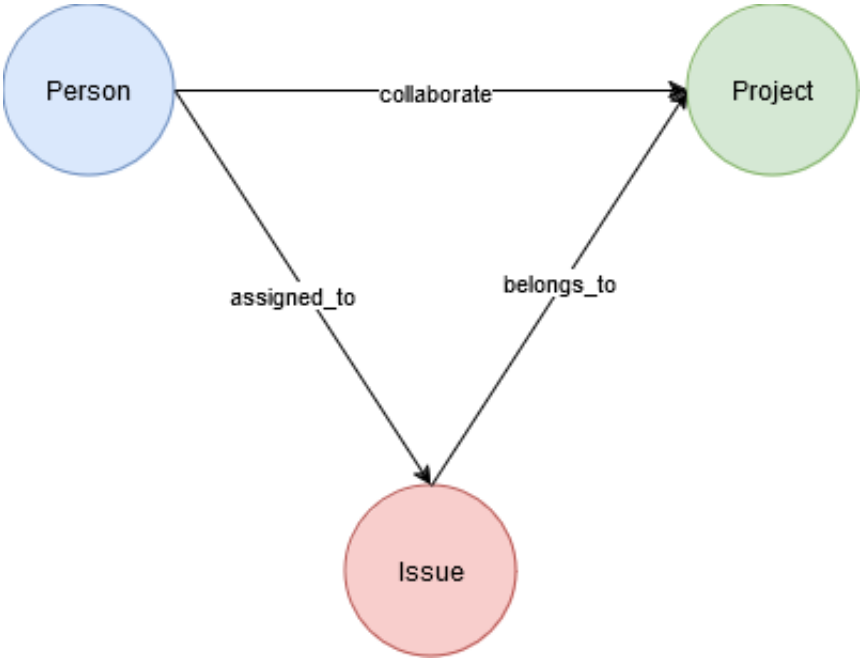

Abbildung 5.7: Graph Diagramm

5.2 Schnittstellendesign

5.2.1 REST

Bei REST wird für jedes Testszenario ein seperater Endpunkt benötigt. Hierbei wurden sechs verschiedene Endpunkte mit unterschiedlichen Komplexität entworfen.

- **HEAD api/resource** wird verwendet, um einen Head-Request durchzuführen um die Latenz der API zu bestimmen.
- **GET api/issues?counter=x&?joins=y**: Hierbei kann die Menge der Ergebnistupel(x) und die Anzahl der Joins(y) die auf der Datenbank durchgeführt werden bei der Anfrage bestimmt werden. Die in Abbildung 5.8 dargestellte Antwort wird hierbei erwwartet.

```
{
    "pid": 10,
    "firstname": "Cecilla",
    "lastname": "Beningfield",
    "email": "cbeningfield9@wp.com"
}
[...]
```

Abbildung 5.8: GET api/issues?counter=x&?joins=y Response

- **GET api/persons/:pid**: Dieser Endpunkt ermöglicht das Abrufen einer bestimmten Person anhand ihrer ID. Die API liefert dabei ein JSON-Objekt zurück, dass die Person mit den Attributen Vorname, Nachname und E-Mail-Adresse beschreibt.(Abb. 5.9)

```
{
    "pid": 10,
    "firstname": "Cecilla",
    "lastname": "Beningfield",
    "email": "cbeningfield9@wp.com"
}
```

Abbildung 5.9: GET api/persons/:pid Response

- Mit dem Endpunkt **GET api/persons** können alle in der Datenbank gespeicherten Personen abgerufen werden. Die Antwort umfasst 5000 Personenobjekte im JSON-Format.(Abb. 5.10)

```
[
    {
        "pid": 1,
        "firstname": "Ruby",
        "lastname": "Burchatt",
        "email": "rburchatt0@msn.com"
    },
        [...]
    {
        "pid": 5000,
        "firstname": "Murdoch",
        "lastname": "Simonitto",
        "email": "msimonittorr@google.ca"
    }
]
```


Abbildung 5.10: GET api/persons Response

- Der Endpunkt **GET api/persons/:pid/projects/issue** erhöht die Komplexität, da hier nicht nur auf ein einzelnes Objekt zugegriffen wird. Stattdessen erfordert die Abfrage mehrere Objekte, die miteinander in Abhängigkeit stehen, um die Anfrage zu bearbeiten. Die Antwort umfasst alle Issues die in Projekten vorhanden sind in der eine Person mitwirkt.(Abb. 5.11)

```
[
   {
        "iid": 1,
        "title": "Pixope",
        "createdAt": "2024-07-23 00:00:00",
        "state": "Closed"
        "stateReason": "Cancelled"
    },
    {
        "iid": 2876,
        "title": "Zoomlounge",
        "createdAt": "2020-08-26 00:00:00",
        "state": "Open"
        "stateReason": "Bug"
    },
]
```


Abbildung 5.11: GET api/persons/:pid/projects/issue Response

- **POST api/persons/:pid/projects/:prid/issues**: Dieser Endpunkt ermöglicht das Erstellen eines neuen Issues in der Datenbank, um nicht nur Abfragen zu testen, sondern auch das Hinzufügen von Daten. Im Body der Anfrage wird ein Issue-Objekt im JSON-Format übergeben.(Abb. 5.12)


```
{
    "title":"test",
    "createdAt":"2023-02-21T00:00:00",
    "state":"Open",
    "stateReason":"Bug"
}
```


Abbildung 5.12: POST api/persons/:pid/projects/:prid/issues Body

Die Antwort enthält das erstellte Issue-Objekt, das eine gültige ID sowie Verknüpfungen zu dem zugehörigen Projekt und der Person beinhaltet.(Abb. 5.13)

```
{
    "iid": 5207,
    "title": "test",
    "createdAt": "2023-02-21T00:00:00",
    "state": "Open",
    "stateReason": "Bug",
    "project": {
        "prid": 12,
        "title": "Y-find",
        "createdAt": "2021-01-10T00:00:00"
    },
    "assignee"{
        "pid": 1,
        "firstname": "Ruby",
        "lastname": "Burchatt",
        "email": "rburchatt0@msn.com"
    },
}
```


Abbildung 5.13: POST api/persons/:pid/projects/:prid/issues Response

5.2.2 GraphQL

GraphQL unterscheidet sich bei der Art der Anfragen sehr stark zu REST. Es wird nur über den Endpunkt POST api/graphql angesprochen. Hierüber werden sowohl Querys als auch Mutations abgedeckt. Die Anfragen werden im Body mithilfe der GraphQL Query Language definiert, welche JSON sehr ähnlich ist. Die in 5.2.1 definierten REST Endpunkte wurden in GraphQL nachgebildet, sodass sie die selbe Antwort liefern. Für einen HEAD Request bietet GraphQL nativ jedoch keine Lösung, weshalb in GraphQL APIs der HEAD Request identisch wie in den REST APIs implementiert wurde. Der parametrisierte Endpunkt aus der REST-API wurde in GraphQL wie in Abb. 5.14 dargestellt umgesetzt. Hierbei sind bei Counter Werte zwischen 0 und 1,5 Mio und bei Joins werte zwischen 0 und 3 möglich.

```
query{
    issuesCount(counter: 10, joins: 1){
        iid
        title
        createdAt
        state
        stateReason
    }
}
```

Abbildung 5.14: GraphQL Query GET api/issues?counter=x&?joins=y

Die in Abbildung 5.15 dargestellte Query ist dem REST Enpunkt GET api/persons/:pid eqivalent. Hierbei wird ebenfalls eine ID übergeben, allerdings können die Felder die in der Antwort enthalten sind explizit gewhlt werden. Hierbei wurden die Personen ID, Vorname, Nachname als auch die Email gewählt, um die selbe Antwort wie der REST Endpunkt zu erhalten.

```
query{
    person(id : 10){
        pid
        firstname
        lastname
        email
    }
}
```

Abbildung 5.15: GraphQL Query equivalent zu GET api/persons/:pid

Der REST Endpunkt GET api/persons wird von der GraphQL Query in Abbildung 5.16 repräsentiert. Hierbei werden die selben Felder selektiert wie in der vorherigen Abfrage, jedoch wird hierbei die Query persons angesprochen, wodurch alle Personen der Datenbank abgerufen werden.

```
query {
    persons {
        pid
        firstname
        lastname
        email
    }
}
```

Abbildung 5.16: GraphQL Query equivalent zu GET api/persons

Abbildung 5.17 zeigt die GraphQL Query, welche dem REST Endpunkt GET api/persons/:pid/projects/issue entspricht. Hierbei werden die im Schema definierten Querys geschachtelt, um eine Abfrage zu erhalten, welche die Abhängigkeiten zwischen den Objekten repräsentiert.

```
query{
    person(id : 10){
        projects{
            issues{
                iid
                title
                createdAt
                state
                stateReason
            }
        }
    }
}
```

Abbildung 5.17: GraphQL Query equivalent zu GET api/persons/:pid/projects/issue

Um den REST Endpunkt POST api/persons/:pid/projects/:prid/issues nachzubilden wurde die in Abb. 5.18 dargestellte Mutation entwickelt. Hierbei wird ein Input Objekt definiert, welches die Attribute beinhaltet, die zur Erstellung des Issues benötigt werden. Danach kann, wie auch in den zuvor beschriebenen Querys, selektiert werden, welche Felder in der Antwort entahlten sind.

```
mutation{
    createIssue(input:{
        title : „Bug in Login"
        createdAt : „2024-12-03T12:30:00
        state : „Open"
        stateReason : „Error in Login"
        prid: 80
        pid: 10
        }){
        iid
        title
        state
        stateReason
        createdAt
    }
}
```

Abbildung 5.18: GraphQL Query equivalent zu POST api/persons/:pid/projects/:prid/issues

5.3 Testumgebung

Zur Ermittlung der Latenzzeiten werden API-Abfragen durchgeführt, bei denen die Antwortzeiten in Millisekunden protokolliert werden. Dafür wird eine Testumgebung mit zwei unterschiedlichen Endgeräten benötigt, um die Last auf mehrere Geräte zu verteilen. Die APIs laufen auf einem Server in Frankfurt, der mit 4 Kernen, 24 GB Arbeitsspeicher, einer 1 Gbit-Internetverbindung und Ubuntu 22.04 als Betriebssystem ausgestattet ist.
Die Abfragen erfolgen von einem PC mit 8 Kernen, 32 GB Arbeitsspeicher, einer 50 Mbit-Internetverbindung und Windows 10 als Betriebssystem. Die durchschnittliche Latenz (Ping) zwischen Server und PC beträgt 24 ms. Da ein Ping jedoch nur auf ISO Schicht 3 aggiert, wird vor jedem Testdurchlauf mithilfe von HEAD-Requests, die in 5.2.1 beschrieben wurden, eine Latenz zwischen den Endgeräten ermittelt die alle Iso-Schichten durchläuft. Um Schwankungen in der Netzwerkauslastung und der Systembelastung zu minimieren, werden pro Testszenario und API jeweils 100 Anfragen ausgeführt. Insgesamt ergibt dies bei 4 APIs und 25 Testszenarien eine Datengrundlage von 10.000 Latenzzeiten.

6 Implementierung

Nachfolgend soll die Implementierung der verschiedenen APIs, sowie die Grundprinzipien während der Implementierung, beschrieben werden. Der Source Code der Implementierung ist unter MIT Lizenz auf Github veröffentlicht. [10]

6.1 Grundprinzipien während der Implementierung

Im Rahmen der Implementierung dieser Anwendung wurden verschiedene Grundprinzipien beachtet, die sowohl die Qualität als auch die Erweiterbarkeit der Anwendungen sicherstellen. Die Wahl der verwendeten Technologien sowie die Anwendung bewährter Praktiken standen dabei im Vordergrund. Als Basis für die Entwicklung wurde Java JDK 17.0.10 Corretto gewählt, da diese Version eine Long-Term-Support (LTS)-Version darstellt und damit eine stabile und sichere Grundlage für die Entwicklung bietet. Amazon Corretto bietet eine optimierte JVM, wodurch alle Softwarebestandteile auf jeder Plattform mit einer zertifizierten JAVA Virtual Machine lauffähig sind. Ergänzt wurde das JDK durch Spring Boot Version 3.3.4, eine weit verbreitete Plattform für die Entwicklung von Webanwendungen und Microservices. Spring Boot ermöglicht eine schnelle und einfache Konfiguration von Anwendungen und vereinfacht den Entwicklungsprozess durch das Automatisieren von häufig auftretenden Aufgaben, wie etwa der Konfiguration von Servern und Datenbankverbindungen. Ein zentrales Konzept während der Implementierung war die Verwendung von Dependency Injection. Diese Technik sorgt dafür, dass die Abhängigkeiten zwischen den einzelnen Komponenten der Anwendung nicht hart kodiert sind, sondern zur Laufzeit durch den DI-Container von Spring injiziert werden. DI fördert die Entkopplung von Komponenten, was zu einer besseren Testbarkeit, Flexibilität und Wartbarkeit des Codes führt. Da der Code durch DI in unabhängige, gut getestete Module unterteilt wird, lässt er sich leicht erweitern und an geänderte Anforderungen anpassen. Zudem trägt DI zur Verbesserung der Lesbarkeit des Codes bei, da Abhängigkeiten nicht explizit im Konstruktor oder an anderen Stellen erstellt werden müssen. Die Implementierung der Anwendung erfolgte unter der Berücksichtigung von Best Practices, die die Qualität des Codes sicherstellen und eine effiziente, langfristige Wartung ermöglichen. Ein wesentlicher Aspekt war hierbei die Modularität der Lösung. Die Anwendung wurde so strukturiert, dass jede Komponente eine klare, abgegrenzte Verantwortung übernimmt. Dies sorgt nicht nur für eine bessere Nachvollziehbarkeit des Codes, sondern erleichtert auch das Testen und die Erweiterung von Funktionalitäten. Bestehende Komponenten können so problemlos durch neue ersetzt oder erweitert werden, ohne die gesamte Anwendung zu beeinträchtigen.

6.2 Post REST

In dieser Arbeit steht "PostREST" für die PostgreSQL REST-API, die eine Schnittstelle für den Zugriff auf eine PostgreSQL-Datenbank über HTTP-Anfragen bietet. Diese API implementiert eine Reihe von Endpunkten, die in Abschnitt 5.2.1 der Arbeit definiert sind. Diese Endpunkte sind dafür verantwortlich, bestimmte Anfragen zu bearbeiten und entsprechende Antworten zurückzugeben. Die zentrale Komponente, die dafür sorgt, dass die Endpunkte korrekt verarbeitet werden, ist die Controller-Klasse(vgl Abb. 6.1). In dieser Klasse sind die Endpunkte integriert, wobei jeder Endpunkt mit seinen spezifischen Pfadvariablen und den Rückgabewerten versehen ist. Die Controller-Klasse übernimmt die Aufgabe, die richtigen Methoden auszuführen, wenn eine Anfrage an einen bestimmten Endpunkt gestellt wird. Der PostrestController ist die konkrete Implementierung des Controllers, der die Geschäftslogik verarbeitet. Er ruft den DBService auf, welcher das Interface IDBService implementiert. Das Interface definiert die Methoden, die notwendig sind, um Daten aus den zugrunde liegenden Datenbanken abzurufen. Diese Methoden kapseln die Logik für den Datenbankzugriff und sind so gestaltet, dass sie von der Controller-Klasse verwendet werden können, um die richtigen Informationen zu erhalten. Die Kommunikation zwischen den verschiedenen Schichten der Anwendung erfolgt durch Dependency Injection. Das bedeutet, dass die verschiedenen Komponenten nicht direkt in der Controller-Klasse erzeugt werden, sondern von außen in die Klasse injiziert werden. In diesem Fall werden die Repositorys der verschiedenen Entitäten in die Controller-Klasse injiziert. Diese Repositorys sind verantwortlich für den direkten Datenbankzugriff und beinhalten die notwendigen Methoden und SQL-Abfragen, die zum Abrufen und Verwalten der Daten in der Datenbank erforderlich sind. Nachdem die Daten erfolgreich aus der Datenbank abgefragt wurden, werden sie durch die Hierarchie der Anwendung weitergegeben. Der PostrestController sorgt dafür, dass die abgerufenen Daten in das gewünschte Format für die API-Antwort umgewandelt werden. Dies ist in diesem Fall das JSON-Format, dass dann dem Nutzer der API als Antwort übermittelt wird. Diese Antwort enthält die angeforderten Informationen, die der Nutzer über die API abgefragt hat.

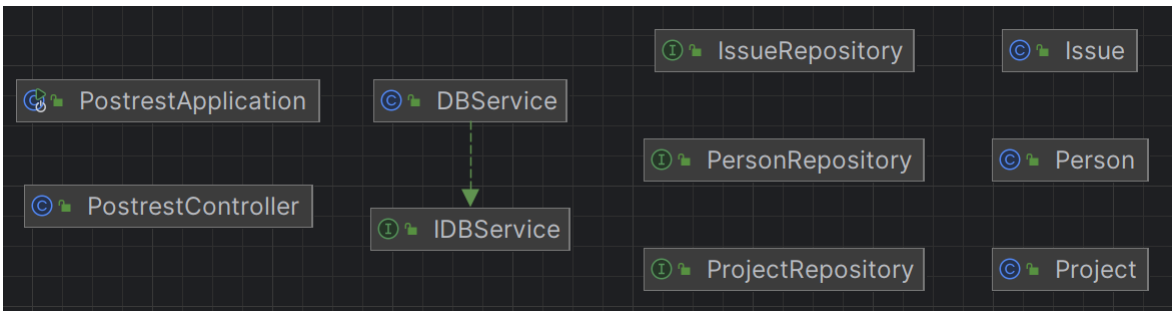

Abbildung 6.1: Java Klassen PostREST

6.3 Post Graph

Hierbei handelt es sich um eine GraphQL-API, die mit einer PostgreSQL-Datenbank verbunden ist. Das Schema, das die Arbeitsweise der API beschreibt, wird – wie bei GraphQL üblich – in der Datei schema.graphqls definiert. In dieser Datei sind die verschiedenen Datentypen sowie die möglichen Queries und Mutationen beschrieben, die zur Abfrage und Bearbeitung der Daten verwendet werden können. In den folgenden Abbildungen sind Beispiele für einen Typ, verschiedene Queries und eine Mutation dargestellt, wie sie in der schema.graphqls-Datei definiert sind:

```
type Issue{
    iid : ID!
    title : String
    createdAt : String
    state : String
    stateReason : String
}
```

Abbildung 6.2: Type aus der Schema.graphqls

```
type Query{
    persons: [Person]
    person(id:ID!): Person
    projects: [Project]
    project(id:ID!) : Project
    issues : [Issue]
    issue(id: String): Issue
}
```

Abbildung 6.3: Queries aus der Schema.graphqls

```
type Mutation {
    createIssue(input: IssueInput): Issue
}
```

Abbildung 6.4: Mutation aus der Schema.graphqls

Die zentrale Komponente für die Verarbeitung von Anfragen ist die Java-Klasse Query(vgl. Abb. 6.5). In dieser Klasse werden Methoden implementiert, die die Bearbeitung der Anfragen gemäß den Strukturen in der schema.graphqls-Datei ermöglichen. Diese Methoden greifen auf die Repositories der Modelklassen zu, um Datenbankoperationen durchzuführen. Die Repositories enthalten die erforderlichen Methoden und Logiken, um Daten in der PostgreSQL-Datenbank zu lesen oder zu ändern. Ein hervorstechendes Merkmal von GraphQL ist der Einsatz von Resolvern. Diese kommen zum Einsatz, um Felder zu verarbeiten, die auf Daten aus anderen Quellen oder Datenbanken basieren. Resolver stellen sicher, dass die relevanten Daten korrekt abgerufen und in der gewünschten Struktur zurückgegeben werden. Die Repositories werden in die benötigten Klassen injiziert, sodass sie nicht direkt innerhalb der Anwendung erstellt werden müssen. Nach erfolgreicher Bearbeitung einer Anfrage liefert die GraphQL-API die Daten im vorgegebenen GraphQL-Format an den Nutzer zurück. Dies gewährleistet eine flexible und leistungsfähige Schnittstelle zur Abfrage und Manipulation der Daten in der zugrunde liegenden PostgreSQL-Datenbank.

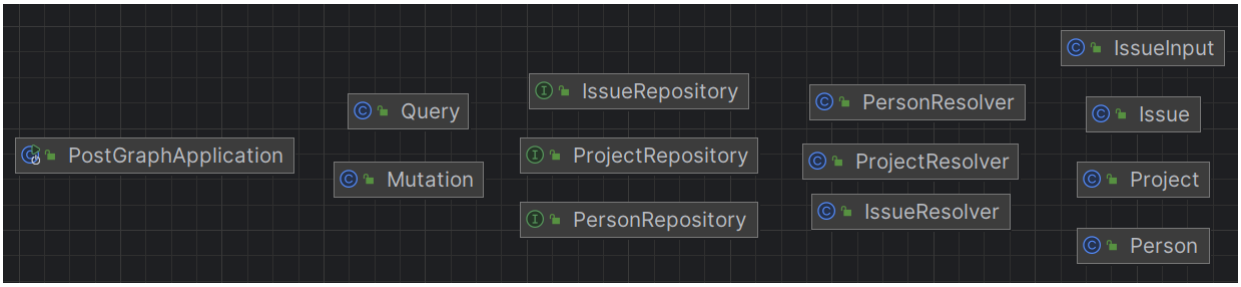

Abbildung 6.5: Java Klassen PostGraph

6.4 Neo4REST

Neo4REST ist eine REST-API, die den Zugriff auf eine Neo4j-Datenbank ermöglicht. Analog zur Struktur von PostREST bietet Neo4REST eine Reihe von Endpunkten, die zur Bearbeitung spezifischer Anfragen und zur Rückgabe der entsprechenden Antworten dienen. Die Details dieser Endpunkte sind in Abschnitt 5.2.2 definiert. Die Verarbeitung der Endpunkte wird durch die zentrale Controller-Klasse sichergestellt (vgl. Abb. 6.7). Wie bei PostREST sind hier die spezifischen Pfadvariablen, Parameter und Rückgabewerte der Endpunkte definiert. Die Controller-Klasse sorgt dafür, dass die passenden Methoden aufgerufen werden, sobald eine Anfrage an einen der Endpunkte gestellt wird. Der Neo4RestController übernimmt dabei die Geschäftslogik und delegiert datenbankbezogene Aufgaben an den DBService, der das Interface IDBService implementiert. Dieses Interface definiert die für den Zugriff auf die Neo4j-Datenbank erforderlichen Methoden. Die Umsetzung der konkreten Abfragen erfolgt über Repositorys, die für die Durchführung der Cypher-Abfragen zuständig sind. Diese Repositorys abstrahieren den Datenbankzugriff und bieten wiederverwendbare Methoden, um auf die in Neo4j gespeicherten Daten zuzugreifen oder sie zu ändern. Die in der Da-

tenbank abgefragten Informationen werden, wie auch bei PostREST, durch die Schichten der Anwendung weitergeleitet und im Controller in das JSON-Format umgewandelt, bevor sie als API-Antwort zurückgegeben werden. So stellt Neo4REST sicher, dass der Nutzer die angeforderten Informationen oder den Status einer Operation im passenden Format erhält.

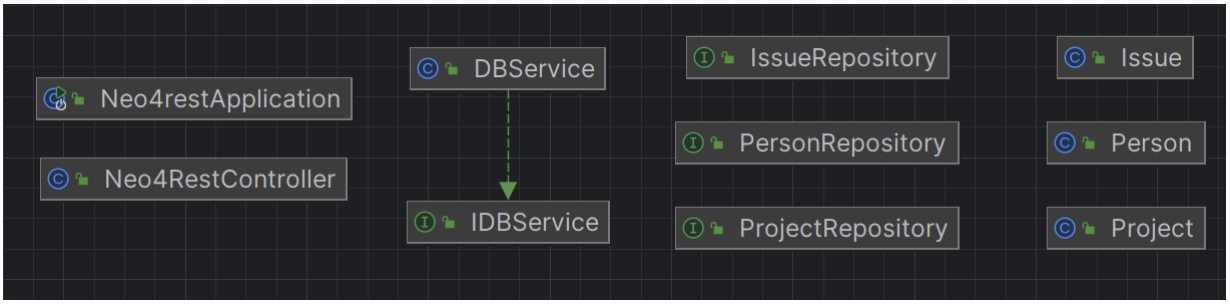

Abbildung 6.6: Java Klassen Neo4REST

6.5 Neo4Graph

Die Neo4Graph-API ist eine GraphQL-API, die mit einer Neo4j-Datenbank verbunden ist. Das Schema, das bereits in der PostGraph-API beschrieben wurde, wird auch in Neo4Graph auf ähnliche Weise implementiert. Wie bei PostGraph kommen auch hier Cypher-Abfragen zum Einsatz, um mit der Neo4j-Datenbank zu interagieren. Die Methoden nutzen die Neo4j-Clientbibliothek, um Knoten und Kanten effizient zu durchsuchen und Datenbankoperationen durchzuführen. Ein weiteres gemeinsames Merkmal beider APIs ist der Einsatz von Resolvern. In Neo4Graph werden diese ebenfalls verwendet, um Felder zu bearbeiten, die auf Beziehungen oder aggregierten Daten im Graphen basieren. Resolver sorgen dafür, dass die benötigten Daten korrekt abgerufen und in der gewünschten Struktur zurückgegeben werden. Nach der Bearbeitung einer Anfrage liefert die GraphQL-API die Daten im standardisierten GraphQL-Format an den Nutzer, was eine konsistente und leistungsfähige Schnittstelle zur Abfrage und Bearbeitung der Daten in der Neo4j-Datenbank sicherstellt

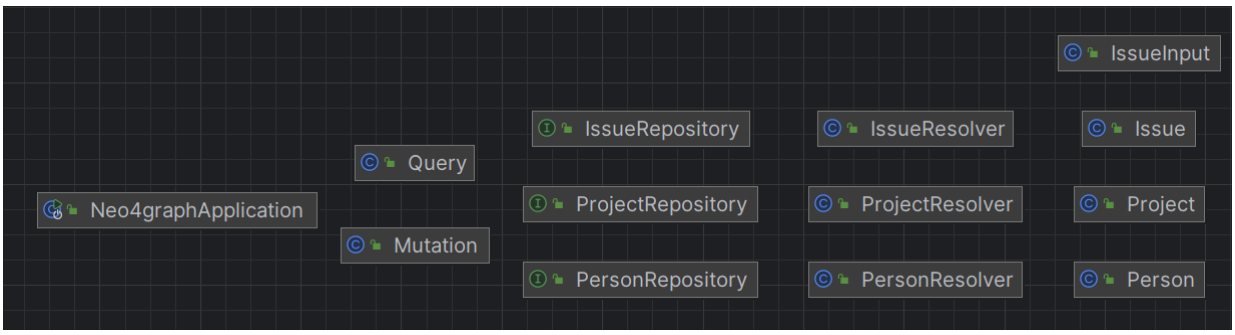

Abbildung 6.7: Java Klassen Neo4Graph

6.6 API-Response Test

Eine Anwendung wurde entwickelt, um API-Anfragen von einem entfernten System durchzuführen. Diese Anwendung enthält für jeden Endpunkt eine Methode(vgl. Abb. 6.8), die jeweils 100 Abfragen in einer Schleife ausführt. Falls für eine Abfrage eine ID erforderlich ist, wird diese mithilfe der Klasse Random aus java.util.Random generiert. Die Abfragen für den parametrisierten Endpunkt werden mit den Joins von 0 bis 3 und den Ergebnistupeln 1, 100, 1000, 10.000 und 100.000 durchgeführt. Die Abfragezeit wird ermittelt, indem die aktuelle Systemzeit in Millisekunden unmittelbar vor der Ausführung der Abfrage erfasst und von der Zeit nach Abschluss der Abfrage abgezogen wird.

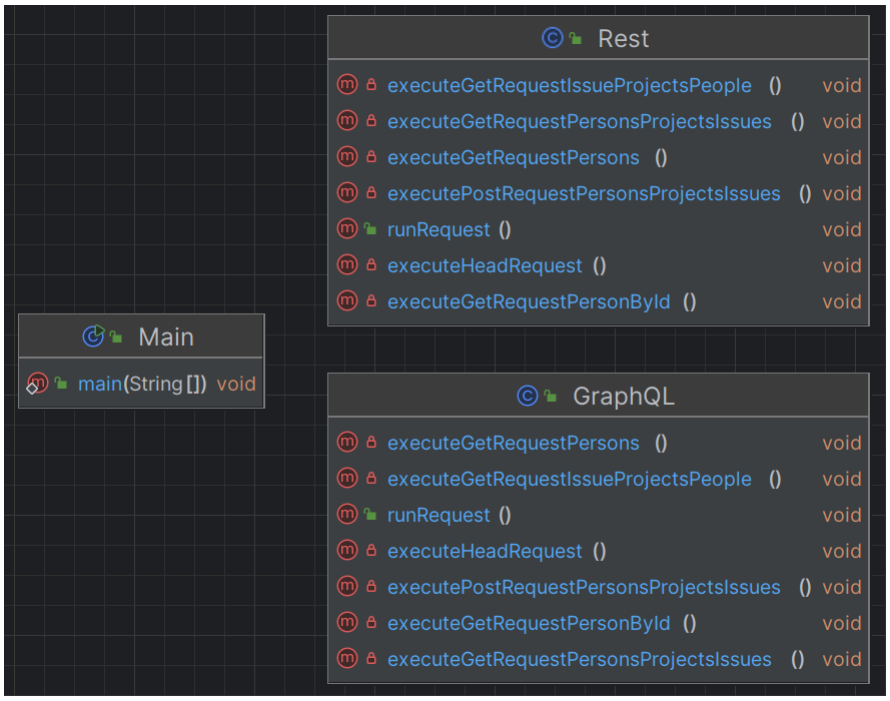

Abbildung 6.8: Java Klassen Neo4Graph

7 Ergebnisse

Im Nachfolgenden werden die Ergebnisse der Latenztests der oben eingeführten APIs dargestellt. Die konkreten Zahlen auf denen die Grafiken beruhen sind im Kapitel Appendix zu finden.
Zunächst wurde, wie in Abbildung 7.1 zusehen, mittels HEAD Request ermittelt, ob die API verfügbar ist und welche Latenz zwischen Client und Server entsteht, ohne eine Datenabfrage auszuführen. Hierbei kann man beobachten, dass Die LAtenz von PostGraph und Neo4Graph sehr dich beieinander liegen. REST basierdende APIs hatten in diesem Experiment eine Latenz die deutlich mehr Variation in der Latenz lieferte.
Daraufhin wurden die Latenzen des parametrisierten API-Endpunktes ermittel. In den Abbildungen 7.2 bis 7.5 wurden diese Ergebnisse in einem Koordinatensystem dargestellt. Die x-Achse repräsentiert die Menge der Tupel, welche angefordert wurden. Aufgrund der hohen Menge der Ergebnistupel wurde diese Achse logarithmisch Dargestellt. Auf der z-Achse ist die Latenz in Millisekunden zu sehen. Die verschiedenen Graphen innerhalb der Abbildungen beschreiben die Menge der Joins, bei relationalen Datenbanken, oder Edges, bei Graphdatenbanken. Rot (Kreis) steht hierbei für null, Blau(Quadrat) steht für eins und Orange (Diamant) steht hierbei für zwei.
Bei PostREST (vgl. Abb. 7.2) steigt somit die Latenz mit der Mange der Ergebnistupeln exponentiell an. Zudem ist zu erkennen, dass bei einer MEnge von 100.000 Ergebnistupeln die Menge der Joins deutlich Einfluss auf die Latenz nehmen. Hierbei ist eine Anfrage die weniger Joins verwendet (Rot) mit 426,5 Millisekunden deutlich perfromanter, als eine Abfrage die mehrere Joins (Orange) verwendet, welche 492,5 Millisekunden benötigt. Bei niedrigerer Tupelzahl wirken sich Joins nicht, bzw. nur in sehr geringen Maß auf die Latenz der Anfrage aus.
PostGraph zeigt in Abbildung 7.3 ein ähnliches Bild, jedoch mit einer deutlich hähren Laten bei einer Anfragengröße von 100.000 Tupel. Die GraphQL API ist hier mit 1831 Millisekunden um einen Faktor von 3,7 langsamer als die REST API, welche auf der selben Datenbank basiert.
Bei Verwendung einer Neo4j Datenbank in Komination mit einer REST-API (vgl. Abb. 7.4) fällt auf, dass die Anfragen mit geringer und hoher Komplexität, sowie geringer Tupelzahl eine sehr ähnliche Latenz aufweißen. Bei höherer Tupelzahl liegt die Anfragedauer jedoch deutlich höher, als mit einer realtionalen Datenbank. Hierbei erkennt man jedoch, dass Anfragen welche mehrere Edges nutzen schneller bearbeitet werden können. So benötigt eine Anfrage, welche ohne das Nutzen einer Kante auskommt 1673 Millisekunden. Wenn man nun die Komplexität erhöht und für die Anfrage zwei Edges verwendet benötigt diese lediglich 1371 Millisekunden.

Behält man nun die Datenbasis bei und nutzt eine GraphQL API (vgl. Abb. 7.5) fällt auf, dass erneut die Anfragedauer bei hoher Tupelzahl deutlich über der einer REST-API liegt. Ebenfalls ist hier zu beobachten, dass komplexere Anfragen schneller bearbeitet werden, als weniger komplexe.
Wie in Abbildung 7.6 zusehen ist, hat die PostgreSQL REST-API eine höhere Latenzzeit als die GraphQL PostgeSQL. Dies deutet darauf hin, dass die GraphQL API bei der Abfrage eines spezifischen Personenobjekts, anhand der Personen-ID in kombination mit einer relationalen Datenbank effizientere Abfragen und eine bessere Performance bei der Verarbeitung bietet. Ähnliches zeigt sich bei einer Neo4j Datenbank. Auch hier hat die Neo4j GraphQL API niedrigere Latenzzeiten als die REST-API. Allerdings sind die Latenzen bei Neo4j minimal höher als bei dem realtionalen Pendant. Das deutet darauf hin, dass Postgres bei dieser Anfrage eine performantere Verarbeitung von Abfragen ermöglicht. Zusammenfassend kann man für diese Anfrage sagen, dass GraphQL sowohl in kombination mit einer relationalen, als auch einer Graphdatenbank eine niedrigere Latenz aufweist. Zudem ist Postgres bei dieser Anfrage insgesamt ein wenig performanter als neo4j.
In Abbildung 7.7 sind die Latenzen für die komplexere Anfrage, die 5000 Personenobjekte aus der Datenbank zurückliefert dargestellt. GraphQL hat hierbei sowohl bei der relationalen Datenbank als auch bei der Graphdatenbank im Median eine niedrigere Latenz. REST ist somit in beiden Fällen die unperformantere API.
Wenn nun die Anfragekomplexität steigt, wodurch sich die Abhängigkeit zwischen den Objekten in der Datenbank erhöht, ist deutlich zu sehen, dass die Streuung bei der Postgres GraphQL API höher ist als bei allen anderen APIs (vgl Abb.7.8). Der Median liegt jedoch deutlich niedriger. Die Neo4j REST-API liegt in diesem Fall deutlich über allen anderen APIs. Gefolgt von der PortgeSQL REST-API.Beide GraphQL APIs weißen erneut eine deutlich niedrigere Latenz auf.
Bei der Speicherung in eine Datenbank ist ein deutlich anderes Bild zu sehen. Wie in Abbildung 7.9 zusehen, ist hierbei die Postgres REST-API diejenige mit der geringsten Latenz. Dicht gefolgt von der Postgres GraphQL API. Eine deutlich höhere Latenz ist bei den neo4j APIs zu erkennen. Hierbei ist allerdings die GraphQL API die performantere.

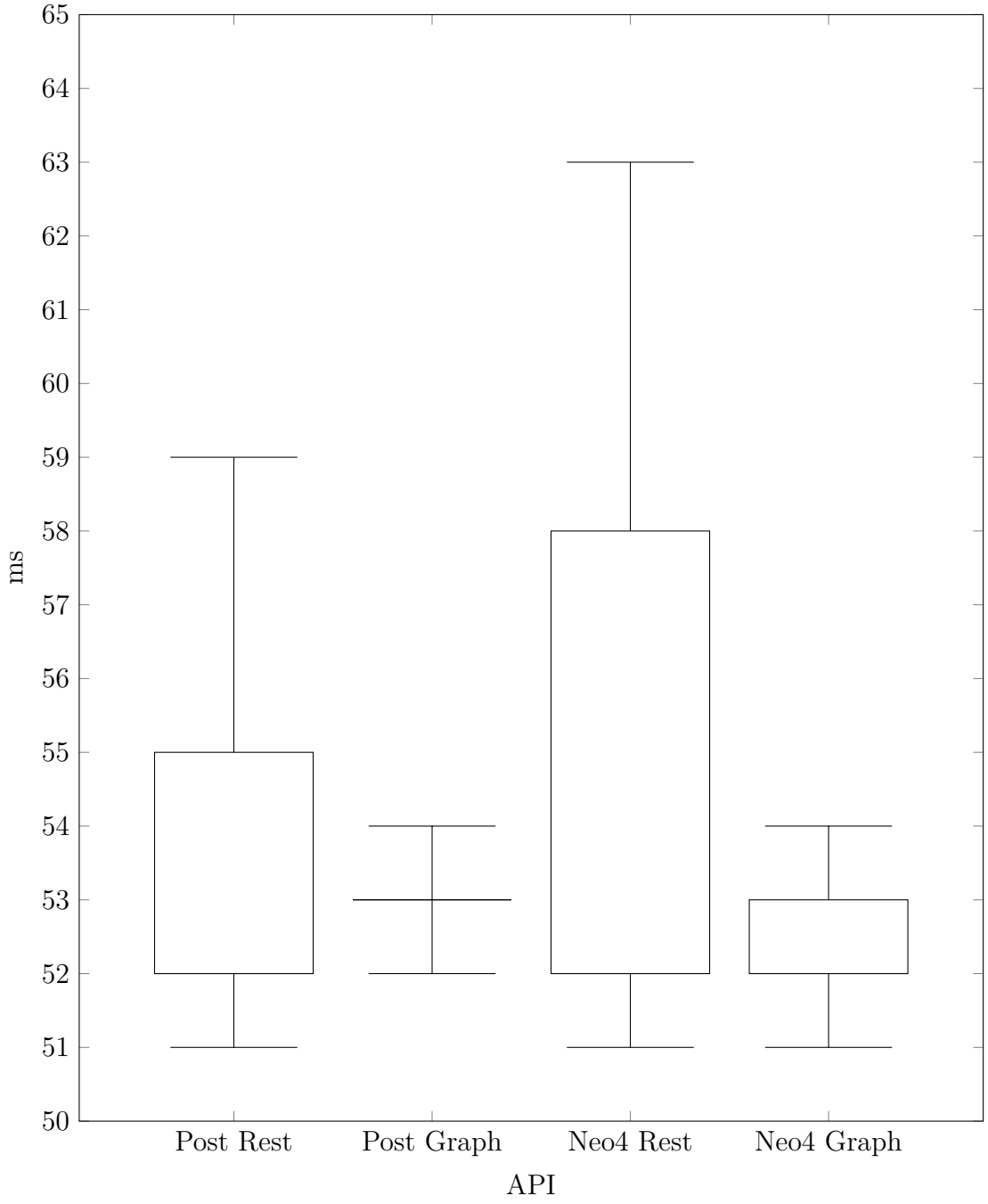

Abbildung 7.1: HEAD /api/resource

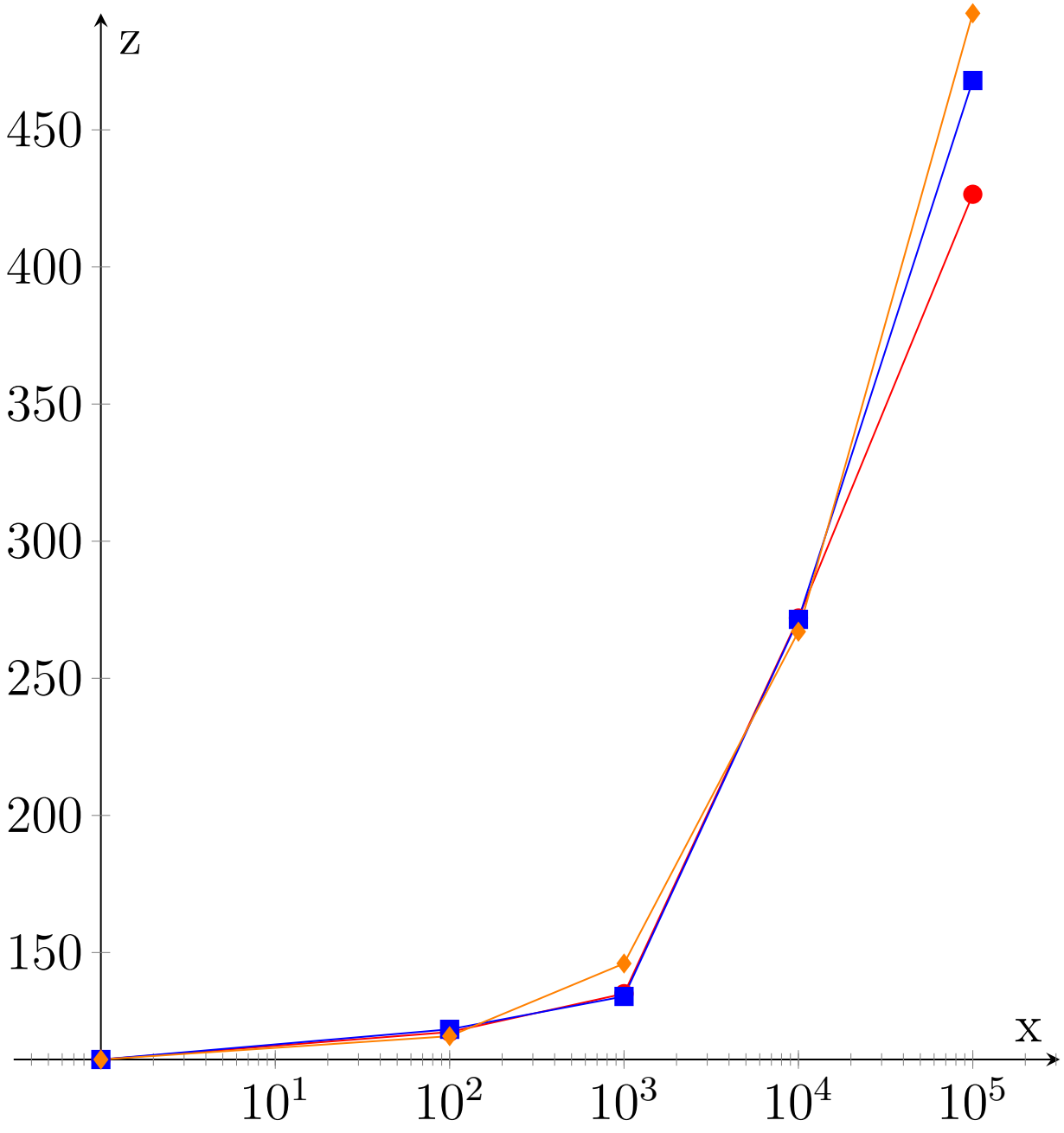

Abbildung 7.2: PostREST parametriesierte Abfragen

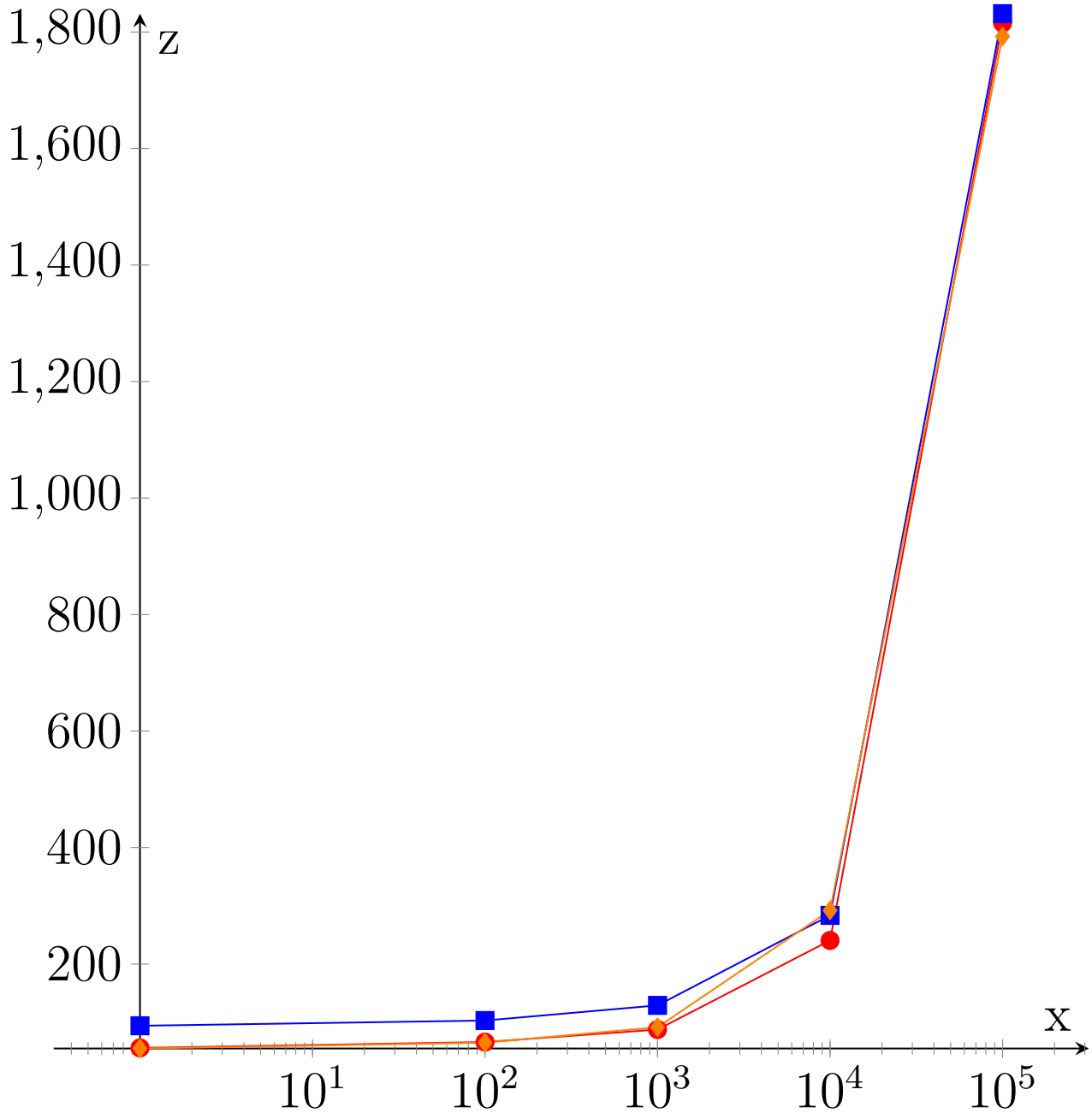

Abbildung 7.3: PostGraph parametriesierte Abfragen

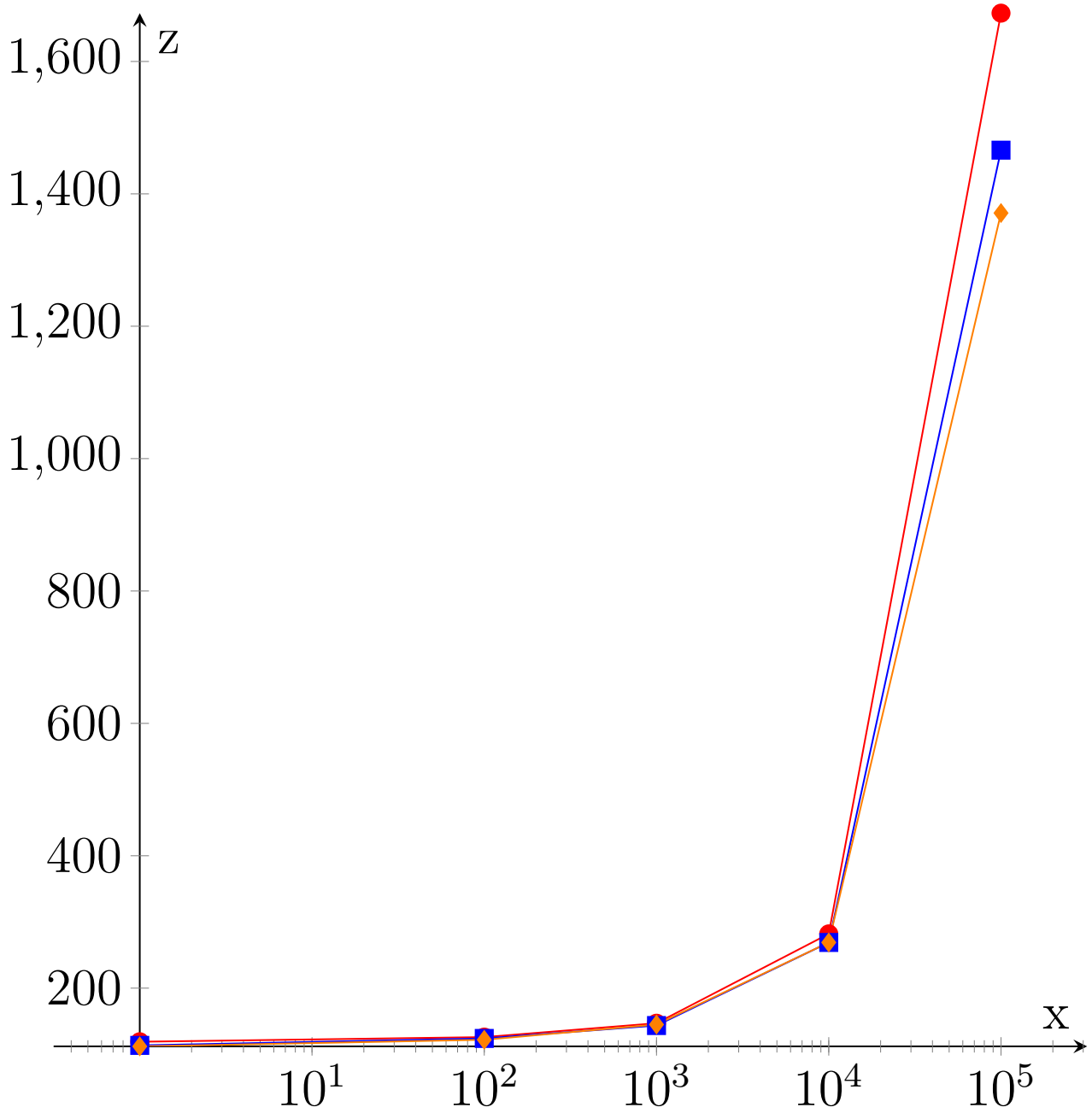

Abbildung 7.4: Neo4REST parametriesierte Abfragen

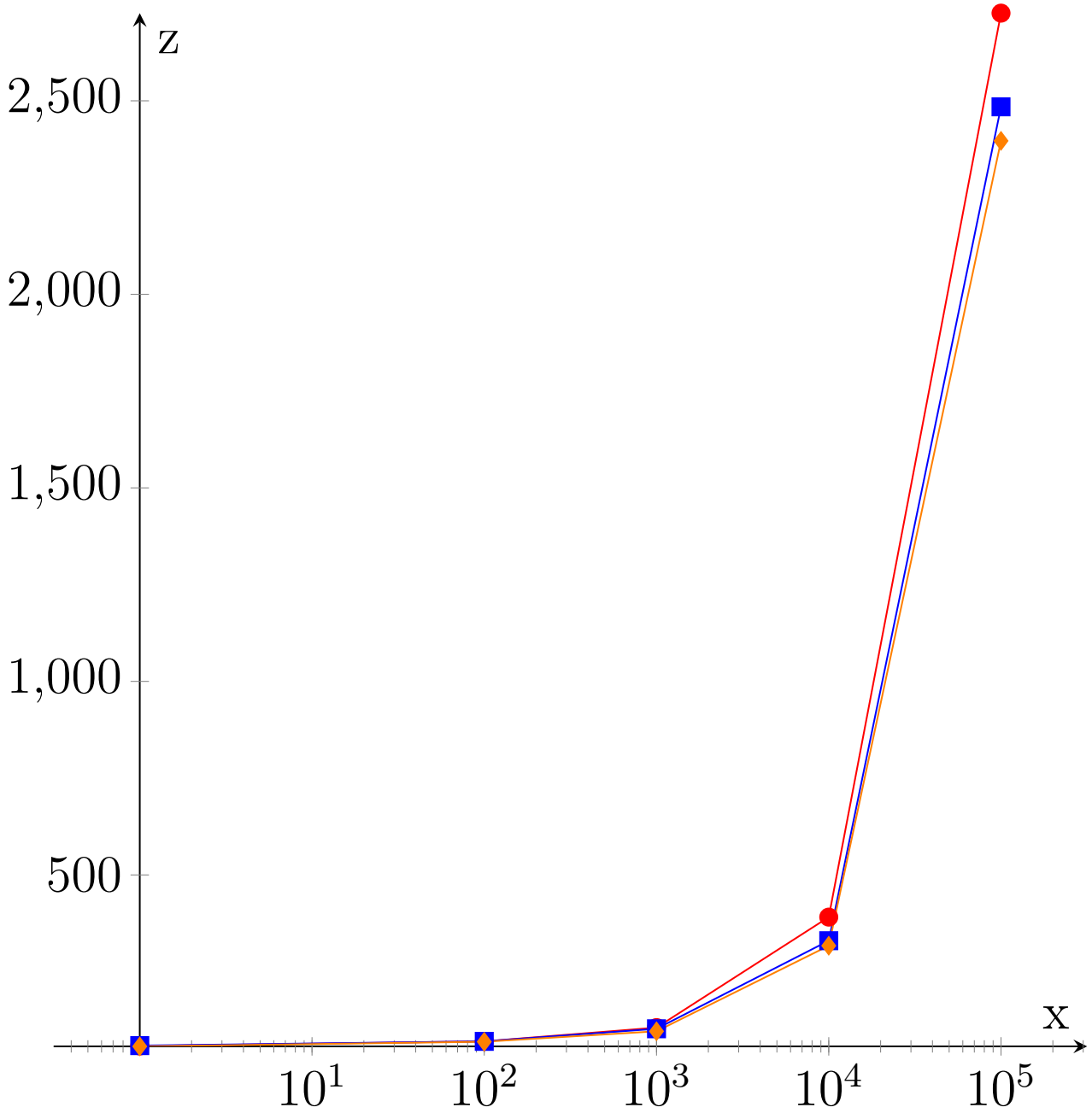

Abbildung 7.5: Neo4Graph parametriesierte Abfragen

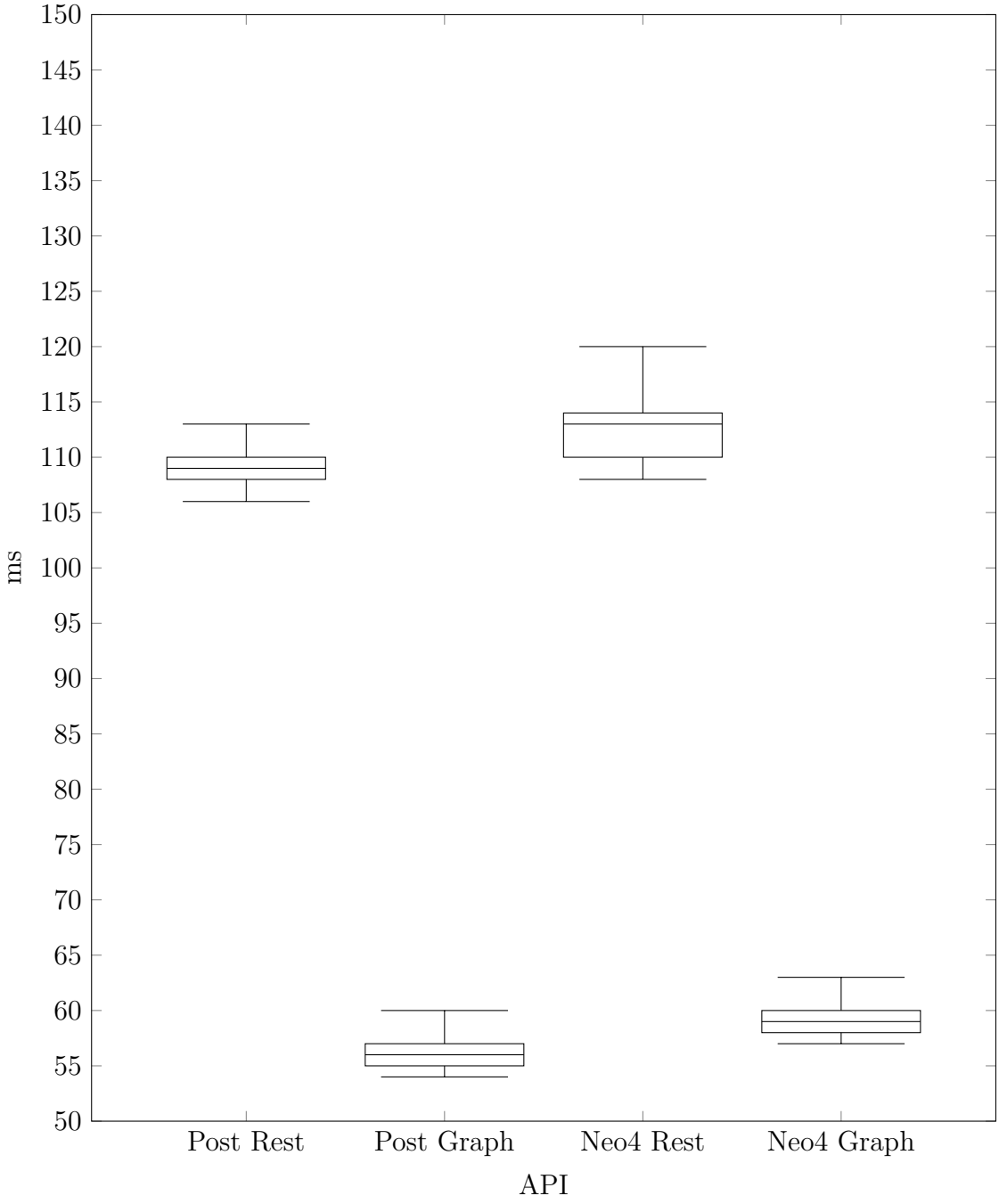

Abbildung 7.6: GET /api/persons/:pid

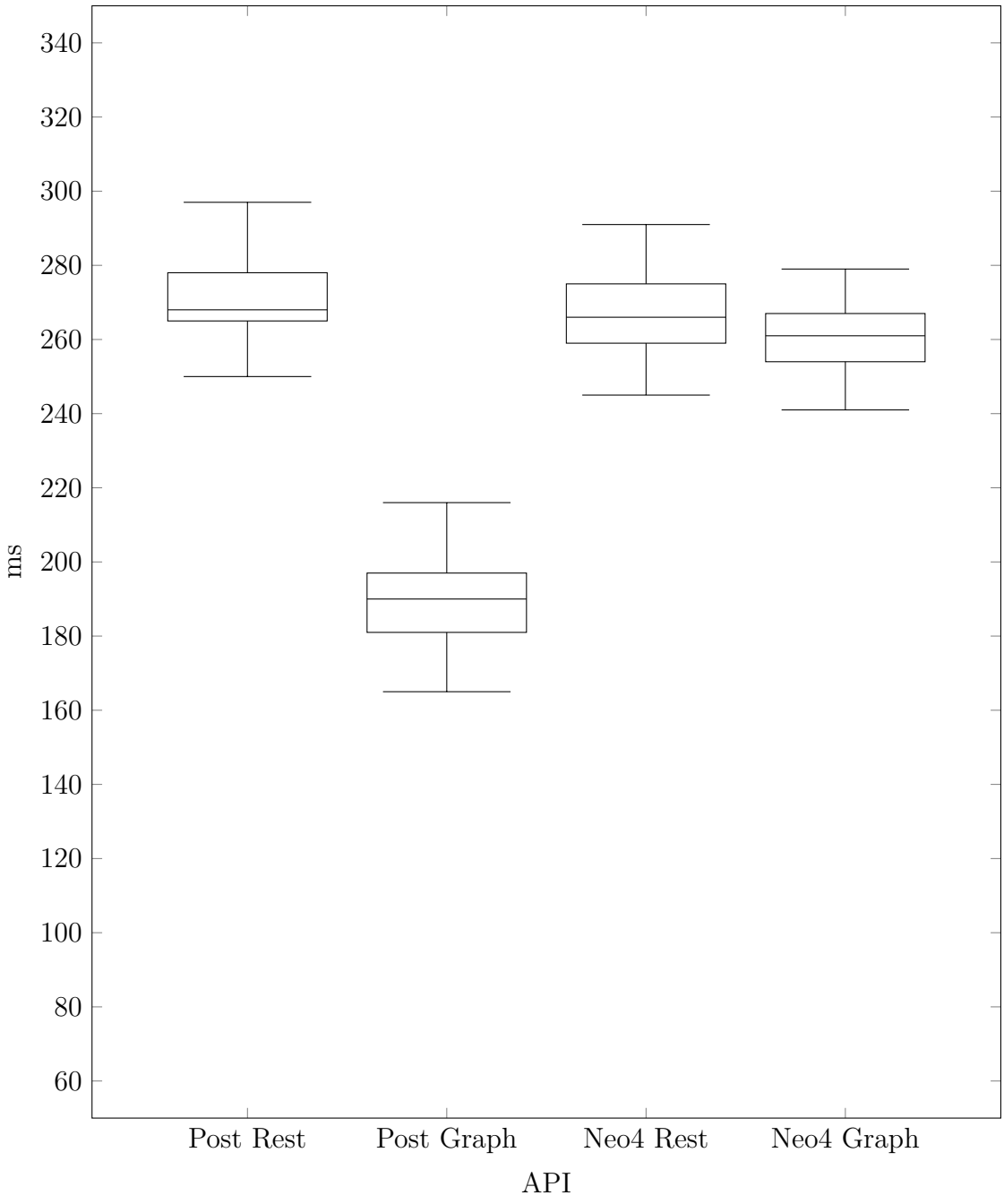

Abbildung 7.7: GET /api/persons

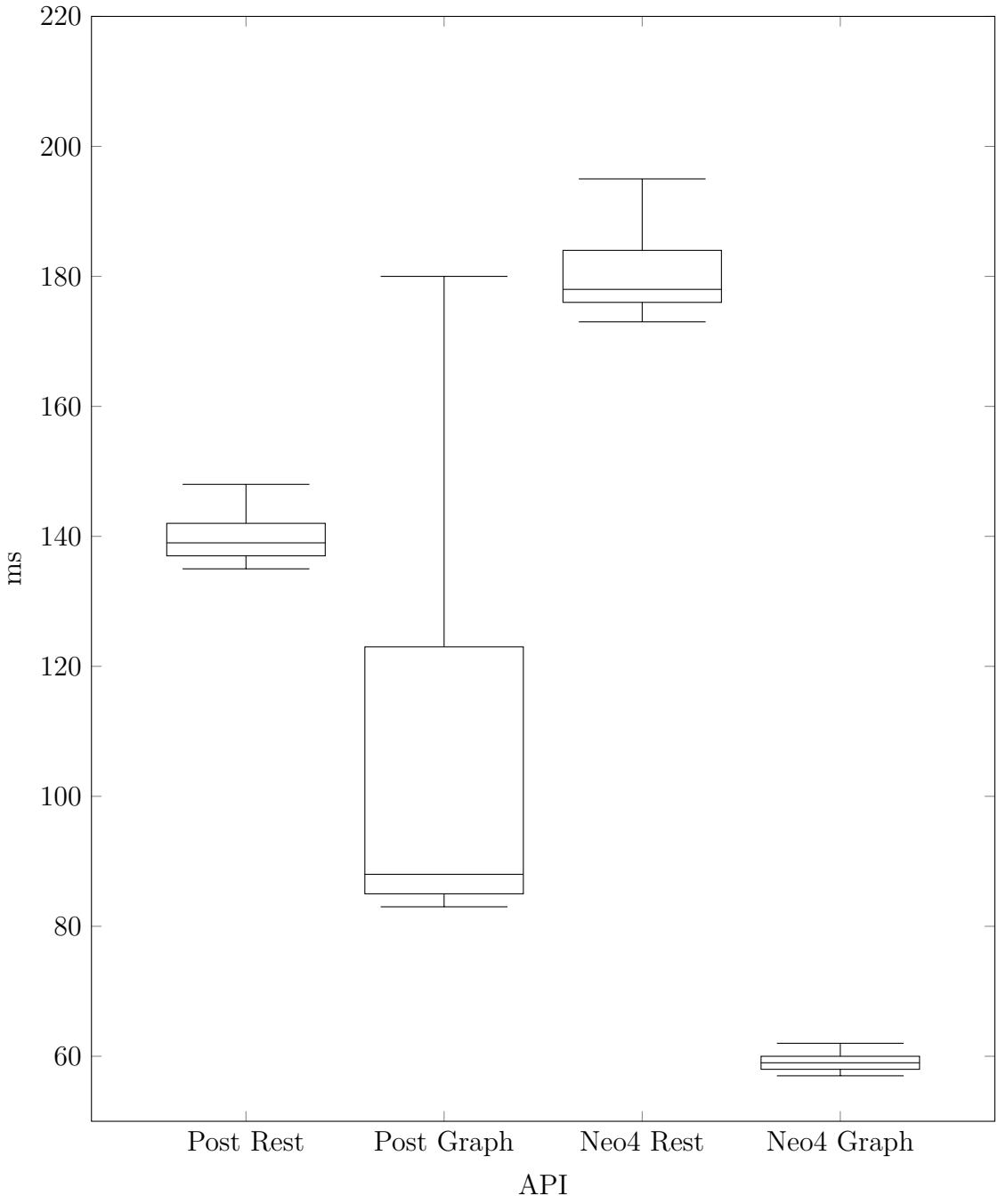

Abbildung 7.8: GET /api/persons/:pid/projects/issues

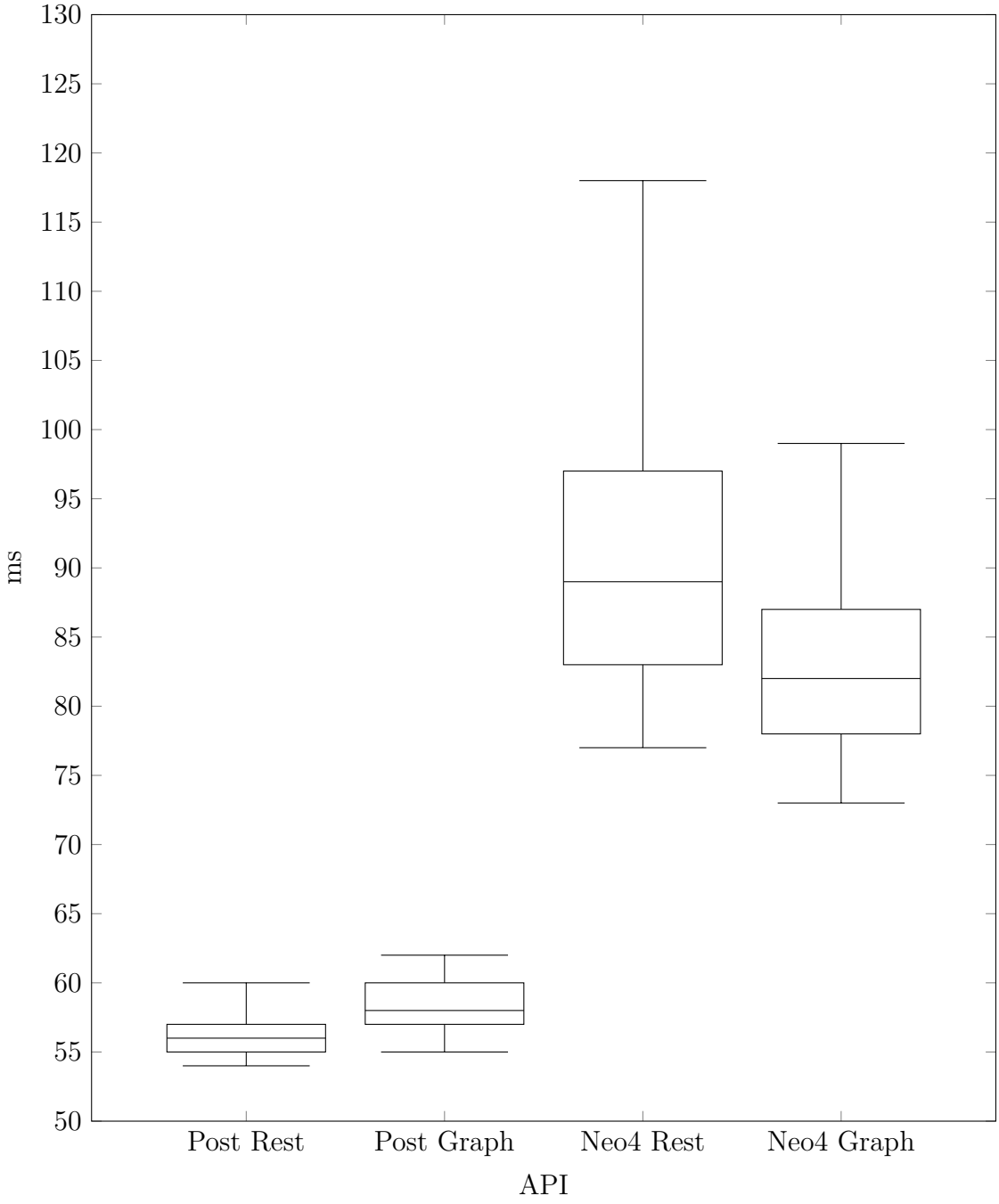

Abbildung 7.9: POST /api/persons/:pid/projects/:prid/issues

8 Diskussion

In diesem Kapitel sollen die aus dem empireischen Experiment gewonnen Erkentnisse mit den Ergebnissen der Literaturanalyse vereinigt werden, um die Ergebnisse des Experiments zu überprüfen.
Im Rahmen des Experiments wurde untersucht, wie sich die Latenzzeiten von REST- und GraphQL-APIs unter verschiedenen Anfragenkomplexitäten unterscheiden. Dabei zeigten die Ergebnisse, dass REST bei Anfragen mit hoher Datenlast (100.000 Ergebnistupel) unabhängig von der zugrunde liegenden Datenbank einen deutlichen Vorteil gegenüber GraphQL hat. Dies deckt sich mit der Literatur, in der berichtet wird, dass GraphQL bei solchen Szenarien etwa 2,5-mal langsamer ist [1].
Für einfache Anfragen wurde in der Literatur ein Geschwindigkeitsfaktor von 0,02 zugunsten von GraphQL genannt [23]. Im Experiment konnte dieser Faktor jedoch in vielen Fällen übertroffen werden. Besonders bei parametrisierten Anfragen, die ohne Joins ausgeführt wurden, zeigte GraphQL eine bis zu 1,9-mal bessere Performance als REST. Dies wurde auch bei einem Endpunkt beobachtet, der realen Anwendungen entspricht (z. B. GET /api/persons/:pid).
Komplexere Anfragen, die mehrere Tabellen oder Nodes innerhalb einer Datenbank abfragen (z. B. GET /api/persons/:pid/projects/issues), zeigten ebenfalls einen deutlichen Vorteil für GraphQL. Diese Überlegenheit wurde bereits in der Literatur beschrieben [20]. GraphQL kann Anfragen, die eine Aggregation oder den Zugriff auf verschachtelte Daten erfordern, effizienter verarbeiten, was seine Stärke bei solchen Szenarien unterstreicht.
Bei der Erstellung von Daten wurden in der Literatur keine signifikanten Unterschiede in der Effizienz zwischen den beiden Ansätzen festgestellt [20]. Diese Beobachtung wurde im Experiment bestätigt: Der Unterschied in der Median-Latenzzeit betrug nicht mehr als 7 Millisekunden.
Ein weiterer Aspekt, welcher Untersucht wurde, war die Auswirkung der Datenbank, konkret einer relationalen oder Graphdatenbank, auf die zuvor genannten APIs. Die parametrisierten Anfragen zeigen, dass die Graphdatenbank und die relationmale Datenbank sich bei der Abfragedauer nicht besonders unterscheiden. Lediglich die Abfrage nach 100.000 Tupeln ruft bei der Graphdatenbank eine enorm Latenz hervor, und dies unabhäng der API. In der Literatur wurde dies so nicht beschrieben [22]. Hier war die Erwartung, dass die Anfragedauer nahezu gleichbleiben sein sollte. Graphdatenbanken sind für Abfragen mit hoher komplexität optimiert und nicht immer für Bulk-Extraktionen. Außerdem haben Graphdatenbanken potenziell mehr Overhead und benötigen mehr Speicher- sowie CPU-Operationen.
Bei einer Abfrage, welche mehr Beziehungsevalutation benötigt (GET /api/persons/:pid /projects/issues), ist zu erkennen, dass die Graphentheorie hier einen deutlich Vorteil auf-

weist. Die GraphQL-API konnte hierbei in Komination mit einer Graphdatenbank die Anfrage 3 mal schneller bearbeiten als eine REST API die mit einer Graphdatenbank operierte. Auf seiten der relationalen Datenbank war ebenfalls ein unterschied zu erkennen. Dieser fiel aber mit einem Faktor von 1,6 deutlich geringer aus.
Bei der Erstellung von Daten auf der Datenbank wurde deutlich Sichtbar, dass eine Graphdatenbank erheblich länger für die Erstellung benötigt. Dies ist dadurch zu begründen, dass bei einer relationalen Datenbank die Tupel lediglich in der Datenbank abgelegt werden, ohne eine beziehung zueinander herzustellen. Eine Graphdatenbank tut dies direkt bei der Erstellung der Daten, wodurch sie mehr Rechenzeit für das Erstellen und die Verbindung der Nodes benötigt. [4]

9 Ausblick

Im Rahmen dieser Arbeit wurden GraphQL und REST in Kombination mit relationalen sowie Graphdatenbanken analysiert. Dabei konnten zahlreiche Ansatzpunkte für weiterführende Untersuchungen identifiziert werden, die sowohl technische als auch konzeptionelle Aspekte betreffen.

Ein erster Ansatzpunkt für zukünftige Arbeiten liegt in der Betrachtung der Implementierung von APIs unter Verwendung verschiedener Programmiersprachen. Die in dieser Arbeit analysierten Implementierungen konzentrierten sich auf spezifische Technologien, was die Frage offen lässt, ob andere Programmiersprachen, insbesondere solche mit unterschiedlichen Paradigmen (z. B. funktional oder objektorientiert), ebenfalls einen Einfluss auf die Latenz oder andere Leistungsparameter der APIs haben könnten. Eine vergleichende Analyse könnte hierbei wertvolle Erkenntnisse liefern.

Im Bereich der Datenbanken wäre es ebenfalls lohnenswert, die Untersuchung auf weitere Datenbanktypen auszudehnen. Besonders dokumentenbasierte NoSQL-Datenbanken wie MongoDB oder Couchbase bieten Potenzial für ergänzende Analysen, da diese aufgrund ihrer strukturellen Unterschiede gegenüber relationalen und graphbasierten Datenbanken interessante Alternativen darstellen könnten. Solche Untersuchungen könnten Aufschluss darüber geben, inwiefern dokumentenbasierte Datenbanken spezifische Vorteile oder Herausforderungen für den Einsatz mit REST oder GraphQL bieten.

Darüber hinaus könnte der Einsatz eines komplexeren Datenmodells spannende Erkenntnisse liefern. In dieser Arbeit wurde ein eher moderates Datenmodell genutzt, das nur eine begrenzte Anzahl an Relationen aufwies. Eine Erweiterung des Modells, beispielsweise durch eine höhere Anzahl von Tabellen oder Knoten sowie komplexere Verknüpfungen zwischen diesen, würde die Belastung auf die Datenbank und die API erhöhen. Solche Szenarien könnten dazu beitragen, die Skalierbarkeit und Effizienz der untersuchten Technologien unter realistischeren Bedingungen zu bewerten.

Zusammenfassend bietet diese Arbeit eine solide Grundlage für weiterführende Analysen und stellt mehrere Ansätze für zukünftige Forschungsarbeiten bereit, die sowohl auf theoretischer als auch auf praktischer Ebene die Relevanz und Einsatzmöglichkeiten von GraphQL und REST vertiefen könnten.

10 Fazit

Die vorliegende Arbeit hat die Performance von REST- und GraphQL-APIs im Zusammenspiel mit relationalen und graphbasierten Datenbanken untersucht und deren Vor- und Nachteile unter verschiedenen Anfragekomplexitäten analysiert. Dabei wurden die beiden zentralen Forschungsfragen beantwortet:

FF-1: Wie unterscheiden sich GraphQL und REST hinsichtlich der Latenzzeit bei unterschiedlichen Anfragenkomplexitäten? GraphQL zeigte bei komplexeren Anfragen deutliche Vorteile, da es in der Lage ist, mehrere Datenpunkte in einer einzigen Abfrage zu bündeln, wodurch die Anzahl der API-Aufrufe reduziert wird. Dies führte in Szenarien mit vielen abhängigen Datenpunkten zu geringeren Latenzzeiten im Vergleich zu REST. REST hingegen war bei einfachen Anfragen effizienter, insbesondere aufgrund der nativen Unterstützung von HTTP-Caching, das wiederholte Anfragen deutlich beschleunigt. Jedoch konnte GraphQL diesem Vorteil ausgleichen und lieferte auch bei einfachen Abfragen schnellere Antwortzeiten. Lediglich bei Anfragen, welche eine große Tupelzahl zurücklieferten konnte REST deutlich bessere Antwortzeiten liefern. Die Ergebnisse zeigen, dass die Wahl der API-Technologie stark von der Komplexität der Anfragen abhängt: Währen REST bei Bulk-Abfragen punktet, ist GraphQL bei einfachen und hierarchischen Abfragen überlegen.

FF-2: Wie beeinflussen graph- und relationale Datenbanken die Latenz von REST- und GraphQL-APIs? Die Wahl der zugrunde liegenden Datenbank hatte ebenfalls einen signifikanten Einfluss auf die Latenz. Relationale Datenbanken erwiesen sich bei der Verarbeitung einfacher und strukturierter Daten als effizient, insbesondere bei Abfragen mit geringen Datenmengen. Mit zunehmender Komplexität und stark vernetzten Daten stießen sie jedoch an ihre Grenzen. Graphdatenbanken, wie Neo4j, zeigten hier ihre Stärke, indem sie Traversal-Mechanismen nutzten, um Beziehungen zwischen Daten effizienter zu verarbeiten. Dies führte bei komplexen Szenarien zu einer spürbaren Reduzierung der Latenz, insbesondere in Kombination mit GraphQL. REST hingegen konnte nicht immer gleichermaßen von graphbasierten Datenbanken profitieren, da zusätzliche Konvertierungsaufwände für die Datenstruktur entstanden

Zusammenfassend zeigt die Arbeit, dass die Kombination aus API-Architektur und Datenbanktechnologie sorgfältig auf die spezifischen Anforderungen eines Anwendungsfalls abgestimmt werden sollte. Während REST und relationale Datenbanken eine bewährte Lösung für klassische Anwendungen darstellen, stellen GraphQL und Graphdatenbanken eine leistungsfähige Alternative für datenintensive und stark vernetzte Anwendungsfälle dar.

Literaturverzeichnis

[1] Tobias Andersson and Håkan Reinholdsson. Rest api vs graphql - a literature and experimental study. 2021. https://researchportal.hkr.se/ws/portalfiles/portal/40132480/FULLTEXT01.pdf.

[2] J. A. Bondy and U. S. R. Murty. *GRAPH THEORY WITH APPLICATIONS*. Elsevier Science Publishing Co., Inc., 52 Vanderbilt Avenue, New York, N.Y. 10017, 1976. ISBN: 0-444-19451-7 https://www.iro.umontreal.ca/~hahn/IFT3545/GTWA.pdf.

[3] E.F. Codd. Relational model of data for large shared data banks. *IBM Research Laboratory, San Jose, California*, 1970. https://doi.org/10.1145/362384.362685.

[4] Calin Constantinov, Mihai L Mocanu, and Cosmin M Poteras. Running complex queries on a graph database: A performance evaluation of neo4j. *Annals of the University of Craiova*, 12(1):38–44, 2015.

[5] IDC; Statista estimates. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028, 2024. https://www.statista.com/statistics/871513/worldwide-data-created/.

[6] Roy T. Fielding. Architectural styles and the design of network-based software architectures. 2000. https://ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[7] Cornelia Győrödi, Alexandra Ștefan, Robert Győrödi, and Livia Bandici. A comparative study of databases with different methods of internal data management. *International Journal of Advanced Computer Science and Applications (IJACSA) Vol. 7, No. 4,*, 2016. https://dx.doi.org/10.14569/IJACSA.2016.070433.

[8] Olaf Hartig and Jorge Pérez. Semantics and complexity of graphql. *WWW 2018, April 23-27, 2018, Lyon, France*, 2018. https://doi.org/10.1145/3178876.3186014.

[9] Mohammad A. Hassan. Relational and nosql databases: The appropriate database model choice. In *2021 22nd International Arab Conference on Information Technology (ACIT)*, pages 1–6, 2021. https://doi.org/10.1109/ACIT53391.2021.9677042.

[10] Robin Hefner. Bachelorthesis. https://github.com/D4rkm4n215/Bachelorthesis.

[11] Daniel Jacobson, Greg Brail, and Dan Woods. *APIs: A Strategy Guide*. O'Reilly Medi, Sebastopol, CA, 2012. ISBN: 978-1-449-30892-6.

[12] Mafalda Isabel Ferreira Landeiro. Analysis of graphql performance: a case study. 2019. https://recipp.ipp.pt/bitstream/10400.22/15946/1/DM_MalfaldaLandeiro_2019_MEI.pdf.

[13] Suresh Kumar Mukhiyaa, Fazle Rabbi, Violet Ka I Pun, Adrian Rutlea, and Yngve Lamo. A graphql approach to healthcare information exchange with hl7 fhir. *The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2019)*, 2019. https://doi.org/10.1016/j.procs.2019.11.082.

[14] Didrik Nordström and Marcus Vilhelmsson. Graphql query performance comparison using mysql and mongodb: By conducting experiments with and without a dataloader, 2022. https://urn.kb.se/resolve?urn=urn%3Anbn%3Ase%3Ahj%3Adiva-58563.

[15] Snehal Eknath Phule. Graph theory applications in database management. *International Journal of Scientific Research in Modern Science and Technology Volume 3*, 2024. https://doi.org/10.59828/ijsrmst.v3i3.190.

[16] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Medi, Sebastopol, CA, 2015. ISBN: 978-1-491-93200-1.

[17] Rahman Saidur. *Basic Graph Theory*. Springer, Cham, Switzerland, 2017. ISBN: 978-3-319-49474-6.

[18] Thomas Studer. *Relationale Datenbanken*. Springer Vieweg, Berlin, Germany, 2019. ISBN: 978-3-662-58976-2.

[19] Clarence J M Tauro, Aravindh S, and Shreeharsha A.B. Comparative study of the new generation, agile, scalable, high performance nosql databases. *International Journal of Computer Applications (0975 – 888) Volume 48– No.20*, 2012. http://dx.doi.org/10.5120/7461-0336.

[20] Sri Lakshmi Vadlamani, Benjamin Emdon, Joshua Arts, and Olga Baysal. Can graphql replace rest? a study of their efficiency and viability. 2021. https://ieeexplore.ieee.org/abstract/document/9474834.

[21] Milena Vesić and Nenad Kojić. N. comparative analysis of web application performance in case of using rest versus graphql. pages 17–24, 2020. https://www.itema-conference.com/wp-content/uploads/2021/03/0_Itema-2020-Conference-Proceedings_Draft.pdf#page=23.

[22] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. 2010. https://doi.org/10.1145/1900008.1900067.

[23] Maximilian Vogel, Sebastian Weber, and Christian Zirpins. Experiences on migrating restful web services to graphql. *Springer Nature 2018*, 2018. https://doi.org/10.1007/978-3-319-91764-1_23.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Publikationen oder Vorträgen anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Heilbronn, den 3. Januar 2025

(Nachname, Vorname)

Appendix

In diesem Anhang befinden sich die Werte der Grafiken, welche im Ergebnis vorgestellt wurden.

HEAD /api/resource

	PostREST	PostGraph	Neo4REST	Neo4Graph
Upper Whisker	59	54	63	54
Upper Quartile	55	53	58	53
Median	52	53	52	52
Lower Quartile	52	53	52	52
Lower Whisker	51	52	51	51

PostREST parametriesierte Abfragen

Tupel	Joins	ms
1	0	111
100	0	121
1000	0	135
10000	0	272
100000	0	426,5

Tupel	Joins	ms
1	1	111
100	1	122
1000	1	134
10000	1	271,5
100000	1	468

Tupel	Joins	ms
1	2	111
100	2	119,5
1000	2	146
10000	2	267
100000	2	492,5

PostGraph parametriesierte Abfragen

Tupel	Joins	ms
1	0	56
100	0	66
1000	0	87,5
10000	0	240,5
100000	0	1815,5

Tupel	Joins	ms
1	1	94
100	1	103
1000	1	129
10000	1	283,5
100000	1	1831,5

Tupel	Joins	ms
1	2	55
100	2	65
1000	2	91,5
10000	2	292
100000	2	1792,5

Neo4REST parametriesierte Abfragen

Tupel	Joins	ms
1	0	119
100	0	126
1000	0	147
10000	0	282
100000	0	1673

Tupel	Joins	ms
1	1	113,5
100	1	124
1000	1	143,5
10000	1	269
100000	1	1466

Tupel	Joins	ms
1	2	112
100	2	122
1000	2	145
10000	2	269
100000	2	1371

Neo4Graph parametriesierte Abfragen

Tupel	Joins	ms
1	0	58
100	0	70
1000	0	105,5
10000	0	391
100000	0	2726,5

Tupel	Joins	ms
1	1	59
100	1	70
1000	1	102,5
10000	1	330
100000	1	2484,5

Tupel	Joins	ms
1	2	57
100	2	69
1000	2	96
10000	2	317
100000	2	2396,5

GET /api/persons/:pid

	PostREST	PostGraph	Neo4REST	Neo4Graph
Upper Whisker	113	60	120	63
Upper Quartile	110	57	114	60
Median	109	56	113	59
Lower Quartile	108	55	110	58
Lower Whisker	106	54	108	57

GET /api/persons

	PostREST	PostGraph	Neo4REST	Neo4Graph
Upper Whisker	297	216	291	279
Upper Quartile	278	197	275	267
Median	268	190	266	261
Lower Quartile	265	181	259	254
Lower Whisker	250	165	245	241

GET /api/persons/:pid/projects/issues

	PostREST	PostGraph	Neo4REST	Neo4Graph
Upper Whisker	148	180	195	62
Upper Quartile	142	123	184	60
Median	139	88	178	59
Lower Quartile	137	85	176	58
Lower Whisker	135	83	173	57

POST /api/persons/:pid/projects/:prid/issues

	PostREST	PostGraph	Neo4REST	Neo4Graph
Upper Whisker	60	62	118	99
Upper Quartile	57	60	97	87
Median	56	58	89	82
Lower Quartile	55	57	83	78
Lower Whisker	54	55	77	73