HOCHSCHULE HEILBRONN

Bachelorthesis
Angewandte Informatik (SPO1a)

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

Robin Hefner*

22. Dezember 2024

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

REST: Representational State Transfer

SQL: Structured Query Language

URL: Uniform Resource Locator

Abbildungsverzeichnis

Abstract

Zusammenfassung

1 Einleitung

1.1 Motivation

In der modernen Softwareentwicklung spielen APIs (Application Programming Interfaces) eine entscheidende Rolle bei der Integration und Kommunikation zwischen verschiedenen Diensten und Anwendungen. Traditionell wurde REST (Representational State Transfer) als Standard für die Erstellung und Nutzung von APIs verwendet. Mit der Einführung und zunehmenden Verbreitung von GraphQL, einer Abfragesprache für APIs, die von Facebook entwickelt wurde, stehen Entwickler nun vor der Wahl zwischen diesen beiden Ansätzen. Zusätzlich gewinnt die Wahl der zugrunde liegenden Datenbanktechnologie an Bedeutung, da sie maßgeblich beeinflusst, wie effektiv REST und GraphQL implementiert werden können. Relationale Datenbanken, die auf strukturierten Tabellen und SQL basieren, bieten bewährte Mechanismen für komplexe Abfragen und garantieren hohe Datenintegrität. In Kombination mit REST ermöglichen sie eine klare Strukturierung von Endpunkten und eine stabile, vorhersehbare Datenabfrage. In GraphQL hingegen können relationale Datenbanken durch Resolver genutzt werden, um gezielt nur die angeforderten Daten bereitzustellen, was die Abfrageleistung bei komplexen Datenmodellen verbessern kann. Graphdatenbanken bieten hingegen eine natürliche Integration für stark vernetzte Daten. Sie zeigen ihre Stärken besonders in Kombination mit GraphQL, da die flexible Abfragesprache direkt auf die Eigenschaften von Graphdatenbanken abgestimmt ist und tiefe, verknüpfte Abfragen effizient ermöglicht. Im Kontext von REST hingegen können Graphdatenbanken ebenfalls verwendet werden, erfordern jedoch oft eine zusätzliche Ebene der Verarbeitung, um die Netzwerkstruktur in flache, hierarchische API-Endpunkte zu überführen. Die Wahl zwischen REST und GraphQL hat signifikante Auswirkungen auf die Entwicklung und den Betrieb der Anwendung, insbesondere im Zusammenspiel mit der zugrunde liegenden Datenbanktechnologie. Unternehmen müssen eine fundierte Entscheidung treffen, welche Kombination aus API-Architektur und Datenbank besser zu ihren Anforderungen im Hinblick auf Leistungsfähigkeit, Skalierbarkeit und Anpassungsfähigkeit passt.

1.2 Forschungsfragen

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

- **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?** Diese Frage zielt darauf ab, den Einfluss der zugrundeliegenden Datenbanktechnologien auf die Latenzzeiten von API-Anfragen zu untersuchen. 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 analysieren und vergleichend zu bewerten. Im Fokus steht dabei, wie sich die Wahl der API-Architektur und der zugrunde liegenden Datenbanktechnologie auf die Latenzzeiten und die Effizienz bei Anfragen mit unterschiedlicher Komplexität auswirkt. Ziel ist es, fundierte Erkenntnisse und Handlungsempfehlungen für Entwickler und Unternehmen bereitzustellen, um die optimale Kombination aus API-Architektur und Datenbanktechnologie für spezifische Anforderungen hinsichtlich Leistungsfähigkeit auszuwählen. Hierfür soll mithilfe der definierten Forschungsfragen eine Analyse durchgeführt werden, durch welche die Arbeit praxisrelevante Einsichten für die effiziente Implementierung moderner API-Systeme liefern soll.

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. Ein besonderes Augenmerk liegt auf der Analyse der Latenzzeiten, sowohl bei einfachen als auch bei komplexeren Abfragen, die mehrere Abhängigkeiten beinhalten. 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 Entwicklern als Entscheidungshilfe dienen, um die geeignetste Technologie für ihre 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. [6]

2.1.1 Basisrelation

Um relationale Algebra anzuwenden werden Basisrelationen benötigt. Diese bilden den Grundbaustein, um mit Grundoperationen komplexe Ausdrücke aufzubauen, die neue Relationen definieren. Basisrelationen bestehen aus drei Bausteinen, nämlich Tupel, Attributen und Domänen. Tupel spiegeln die Zeilen der Tabellen woeder, die einzelne Datensätze repräsentieren. Diese werden durch Attribute in einzelne Spalten eingeteilt, welche die Eingenschaften der Tupel beschreiben. Die Wertebereiche, die für die einzelnen Attribute zulässig sind nennt man Domänen. Somit ist jede Relation eine Menge von Tupeln mit spezifischen Attributen und deren Domänen. [6]

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 auszuzwählen 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. [6]

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. [5]

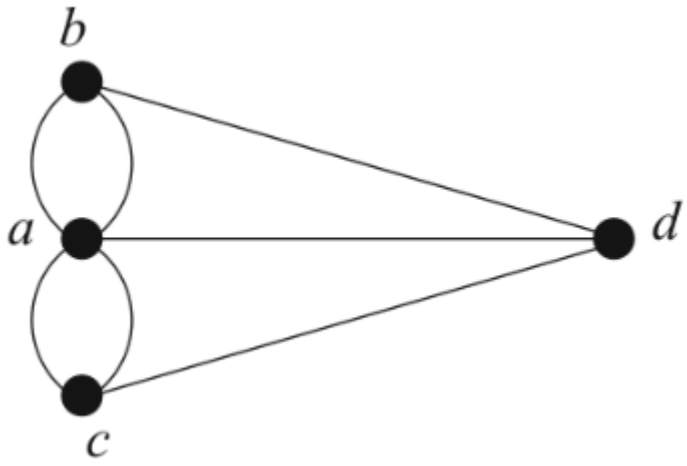

Abbildung 2.1: Modell eines ungerichteten Graphen. [5]

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. [5]

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. 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 (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.[5]

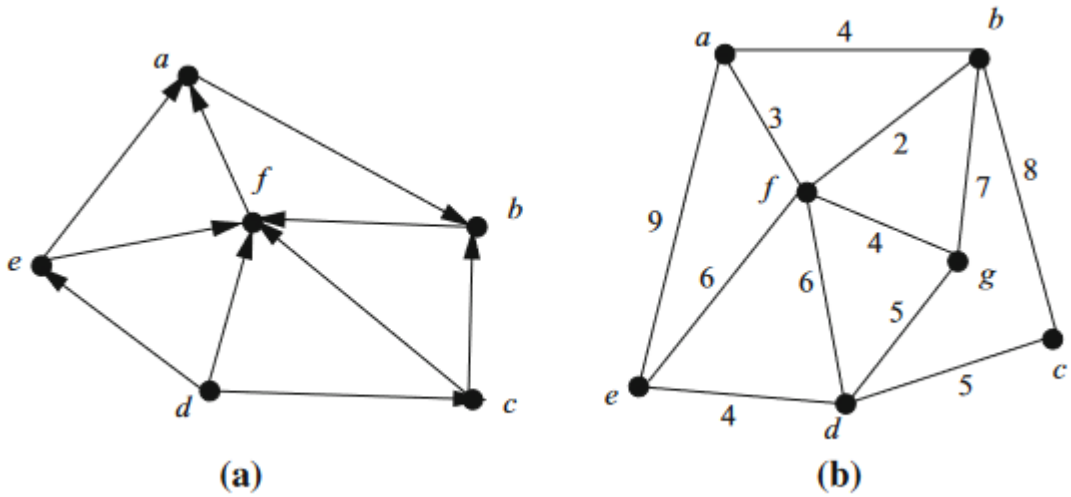

Abbildung 2.2: Modell eines gerichteten (a) und eines gewichtet Graphen (b). [5]

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. [4]

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. [1] [7]

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. Eine wichtige Erkenntnis aus der Untersuchung ist, dass GraphQL bei großen und komplexen Datenanforderungen eine deutlich bessere Effizienz als REST aufweist, da es die Anzahl der erforderlichen Serveraufrufe erheblich verringert. [7]

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. [3]

2.4.2 Relationale Datenbank

Relationale Datenbanken basieren auf dem von E. F. Codd eingeführten relationalen Modell. Es verwendet relationale Algebra und Tupel-Relationen und speichert Daten in tabellarischer Form, wobei Zeilen als Tupel und Spalten als Attribute bezeichent werden. Dies hat zur Folge, dass die Struktur, in der die Daten gespeichert werden sollen vor der Speicherung in der Datenbank bekannt sein müssen. Falls Werte nicht vorkammen werden diese auf null gesetzt. Die Tabellen sind durch Primär- und Fremdschlüssel miteinander verknüpft. Diese Art von Datenbanken sind einfach zu entwerfen und umzusetzen und zeichnen sich durch Benutzerfreundlichkeit, Konsistenz und Flexibilität aus. Diese Datenbanken eignen sich besonders für normalisierte Daten und solche, die Transaktionsintegrität erfordern. [2] [3]

2.4.3 Graphdatenbanken

Graphdatenbanken basieren auf dem Konzept der Graphentheorie, die Daten in Form von Graphen schemafreien Weise speichern. Eine Graphdatenbank ist eine Sammlung von Knoten und Kanten, wobei die Knoten Entitäten oder Objekte darstellen und die Kanten die Beziehungen zwischen den Knoten darstellen. Der Graph enthält auch Informationen über die Eigenschaften der mit den Knoten verbundenen Objekte. Graphdatenbanken bieten eine effiziente Datenspeicherung speziell für semistrukturierte Daten. Die Formulierung von Abfragen als Traversale in Graphen-Datenbanken sind sie schneller als relationale Datenbanken.

Graphdatenbanken befolgen ACID-Bedingungen und bieten Rollback-Unterstützung, die die Konsistenz der Informationen Informationen garantiert. [3]

3 Analyse

Die zunehmende Verbreitung von Web-APIs hat die Diskussion über deren Effizienz und Performanz in der Softwareentwicklung intensiviert. Während REST aufgrund seiner Einfachheit und Standardisierung lange Zeit als de-facto Standard galt, bietet GraphQL durch flexible Abfragen ein vielversprechendes Alternativmodell.

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.

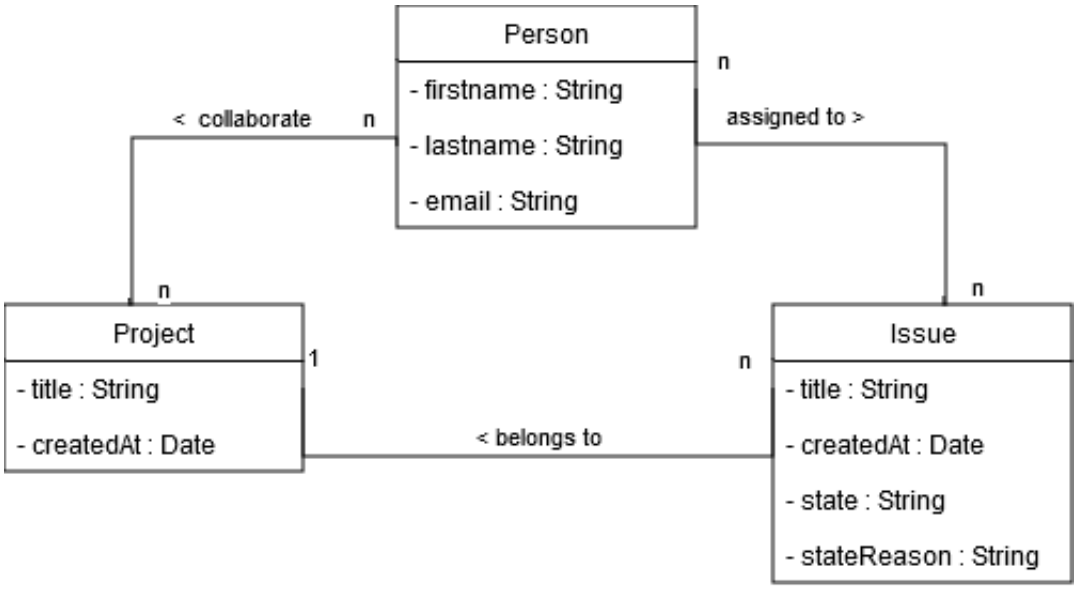

Abbildung 4.1: Klassendiagramm

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

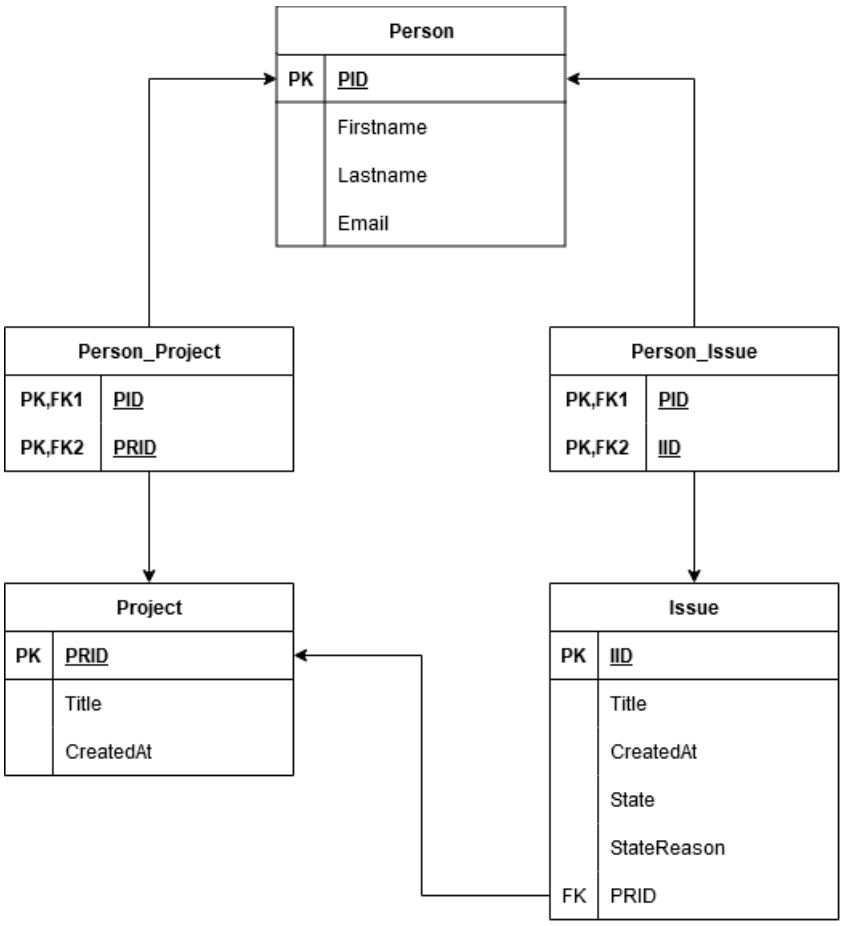

Abbildung 5.1: Tabellen Diagramm

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.

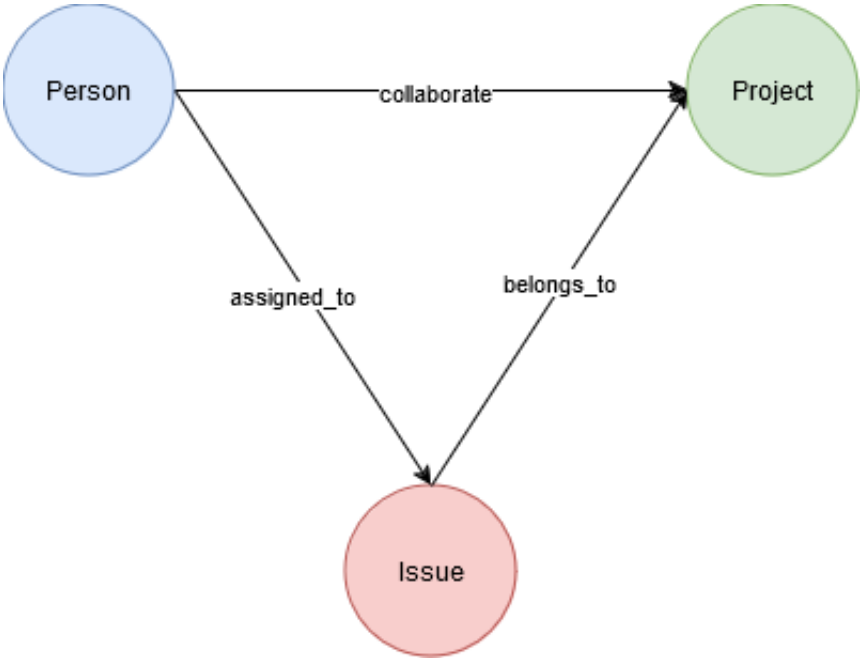

Abbildung 5.7: Graph Diagramm

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.

```
{
    "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. 5.3)

```
{
    "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. 5.4)

```
[
    {
        "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. 5.5)

```
[
   {
        "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. 5.6)


```
{
    "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. 5.7)

```
{
    "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. Die in Abbildung 5.8 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 5.14: GraphQL Query equivalent zu GET api/persons/:pid

Der REST Endpunkt GET api/persons wird von der GraphQL Query in Abbildung 5.9 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 5.15: GraphQL Query equivalent zu GET api/persons

Abbildung 5.10 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 5.16: 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. 5.11 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 5.17: 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. 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 4 Testszenarien eine Datengrundlage von 1600 Latenzzeiten.

6 Implementierung

Nachfolgend soll die Implementierung der verschiedenen APIs, sowie die Grundprinzipien während der Implementierung, beschrieben werden.

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. 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.

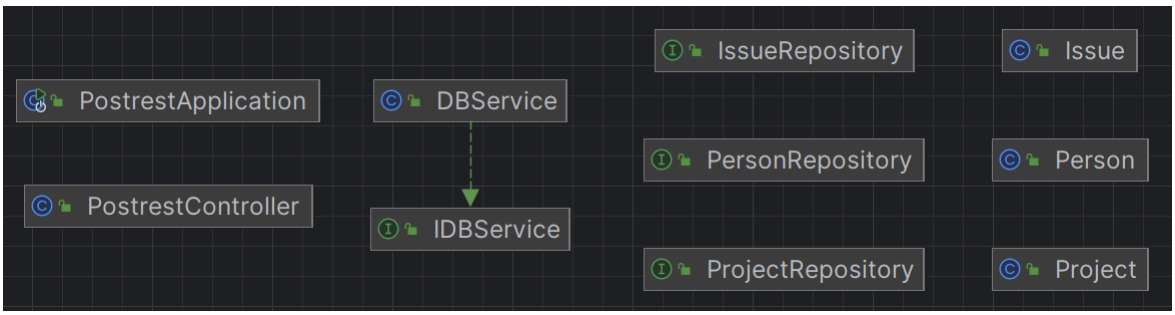

Abbildung 6.1: Java Klassen PostREST

6.3 Post Graph

Hierbei handelt es sich um eine GraphQL API mit PostgreSQL Datenbank.

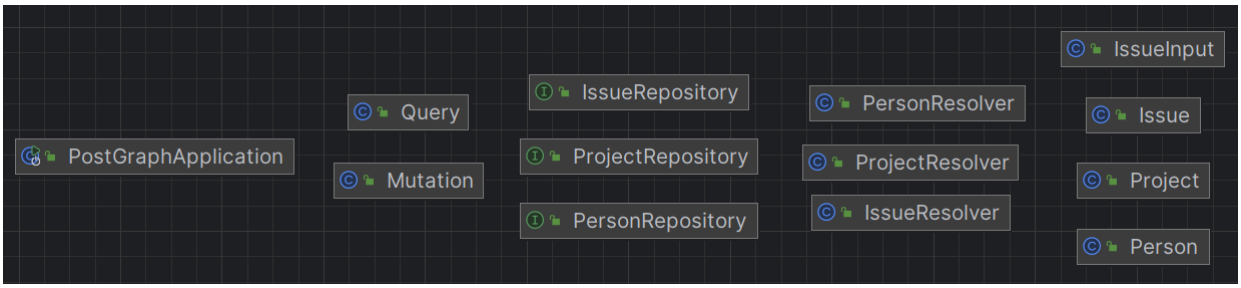

Abbildung 6.2: Java Klassen PostGraph

6.4 Neo4REST

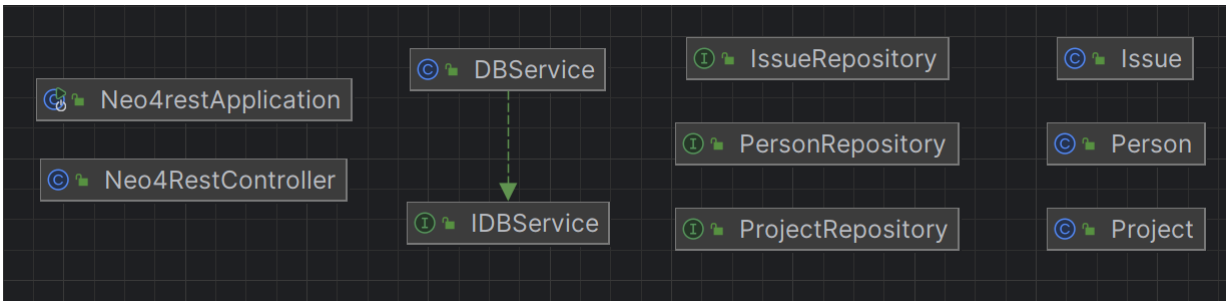

Abbildung 6.3: Java Klassen Neo4REST

6.5 Neo4Graph

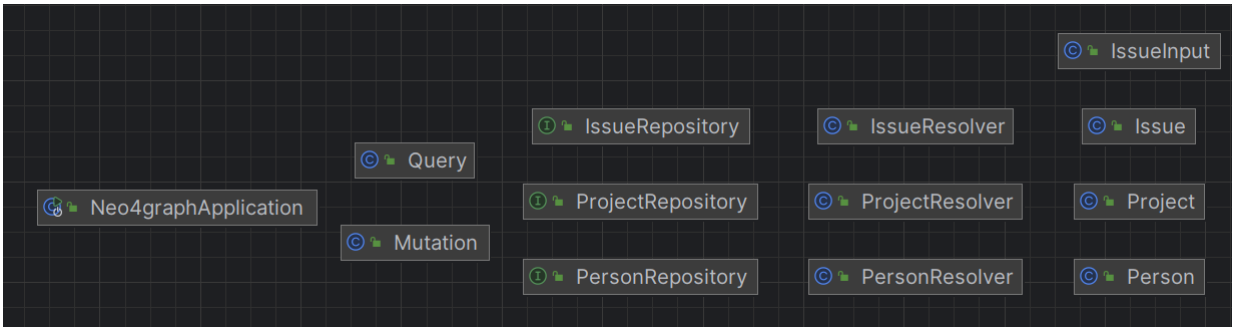

Abbildung 6.4: Java Klassen Neo4Graph

7 Ergebnisse

Im Nachfolgenden werden die Ergebnisse der Latenztests der oben eingeführten APIs dargestellt.
Wie in Abbildung 7.1 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 7.2 sind die Latenzen für die komplexere Anfrge, die alle Personenobjekte aus der Datenbank zurückliefert dargestellt. Bei den APIs die mit Postgres implementiert sind ist eine deutlich niedrigere Latenz zu sehen, als bei den mit neo4j implementierten. 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 REST-API höher ist als bei allen anderen APIs (vgl Abb.7.3). Zudem ist diese API diejenige mit der höchsten Latenz. Allerdings liegt die Neo4j REST-API im Median über den Postgres 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 7.4 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.

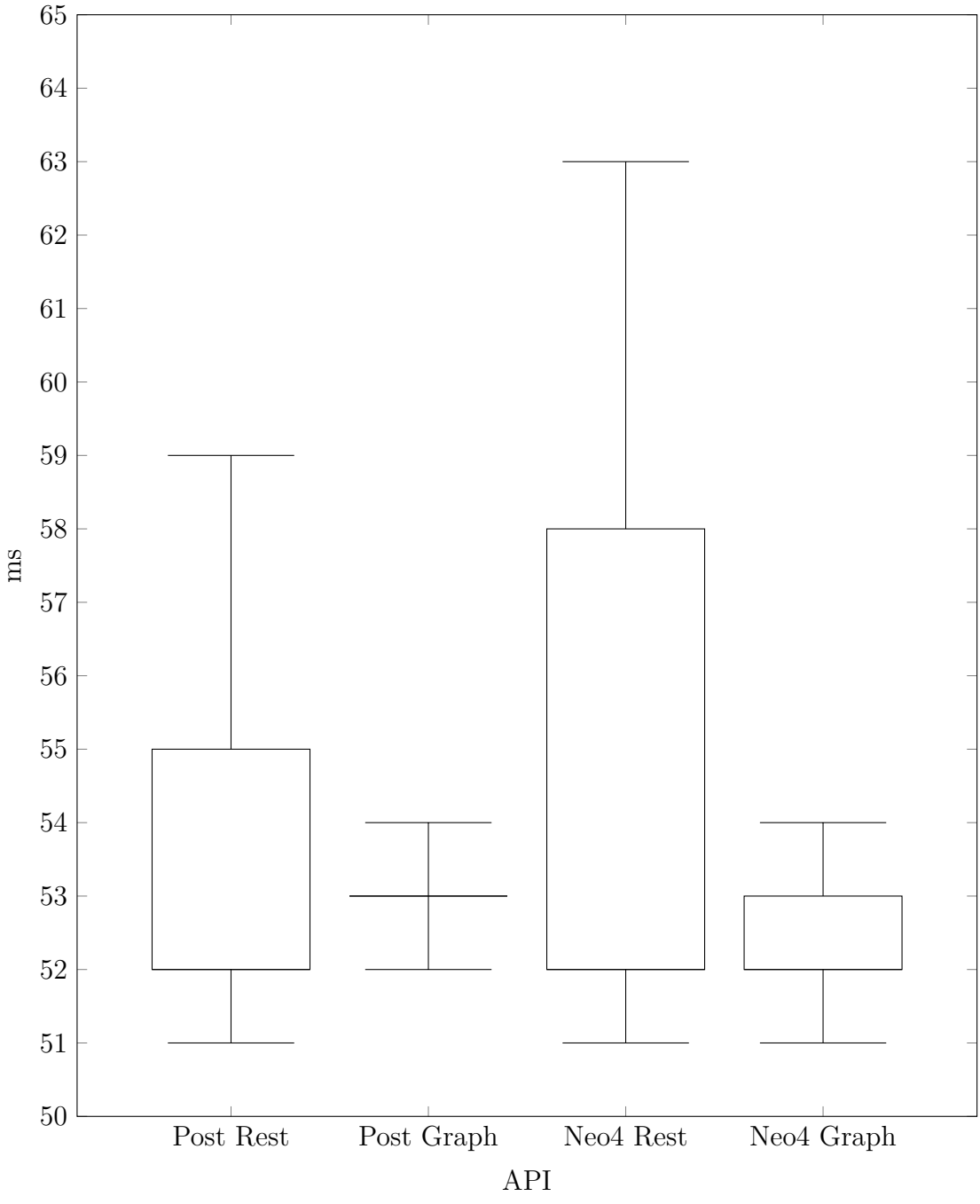

Abbildung 7.1: HEAD /api/resource

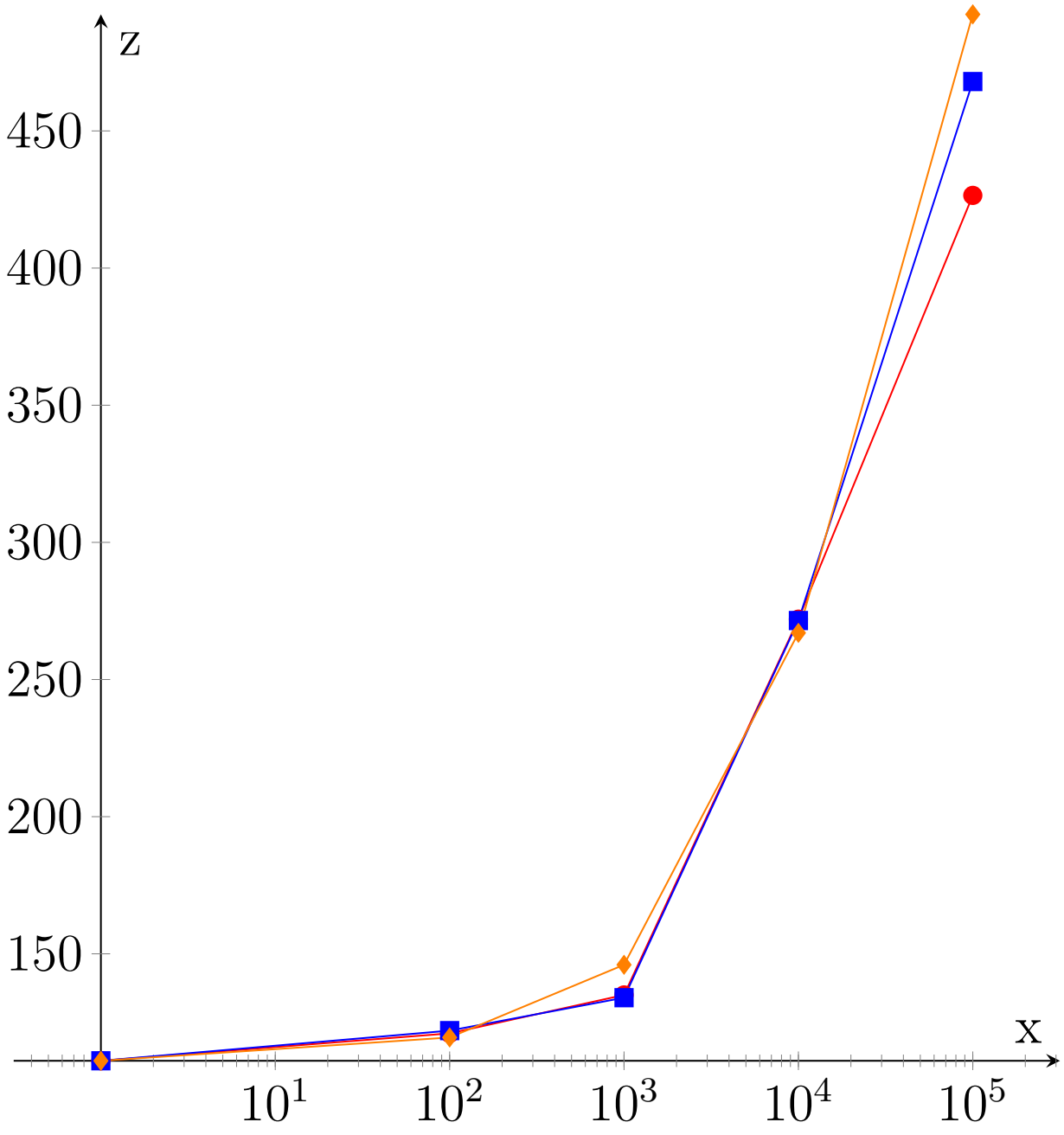

Abbildung 7.2: PostREST parametriesierte Abfragen

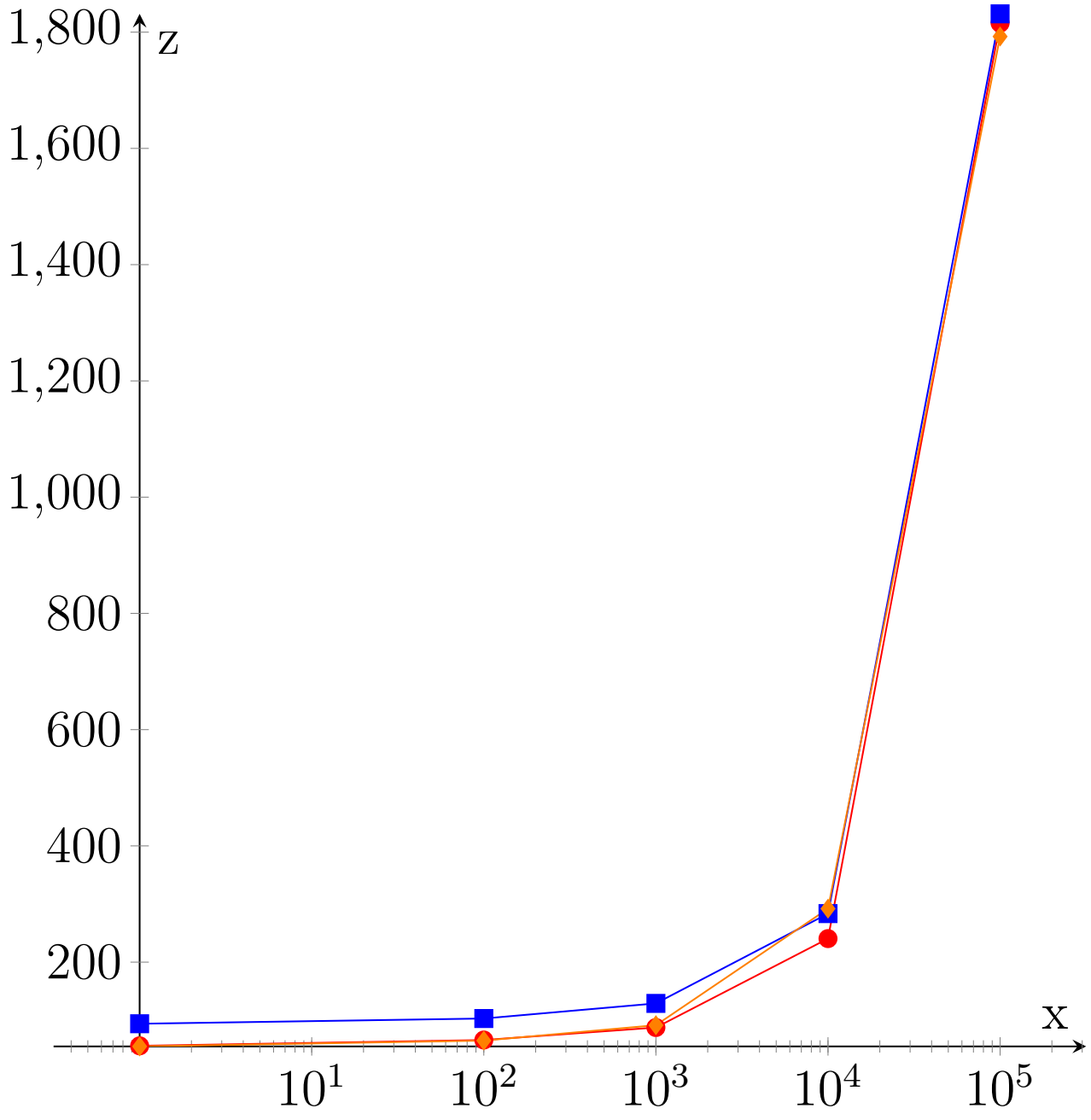

Abbildung 7.3: PostGraph parametriesierte Abfragen

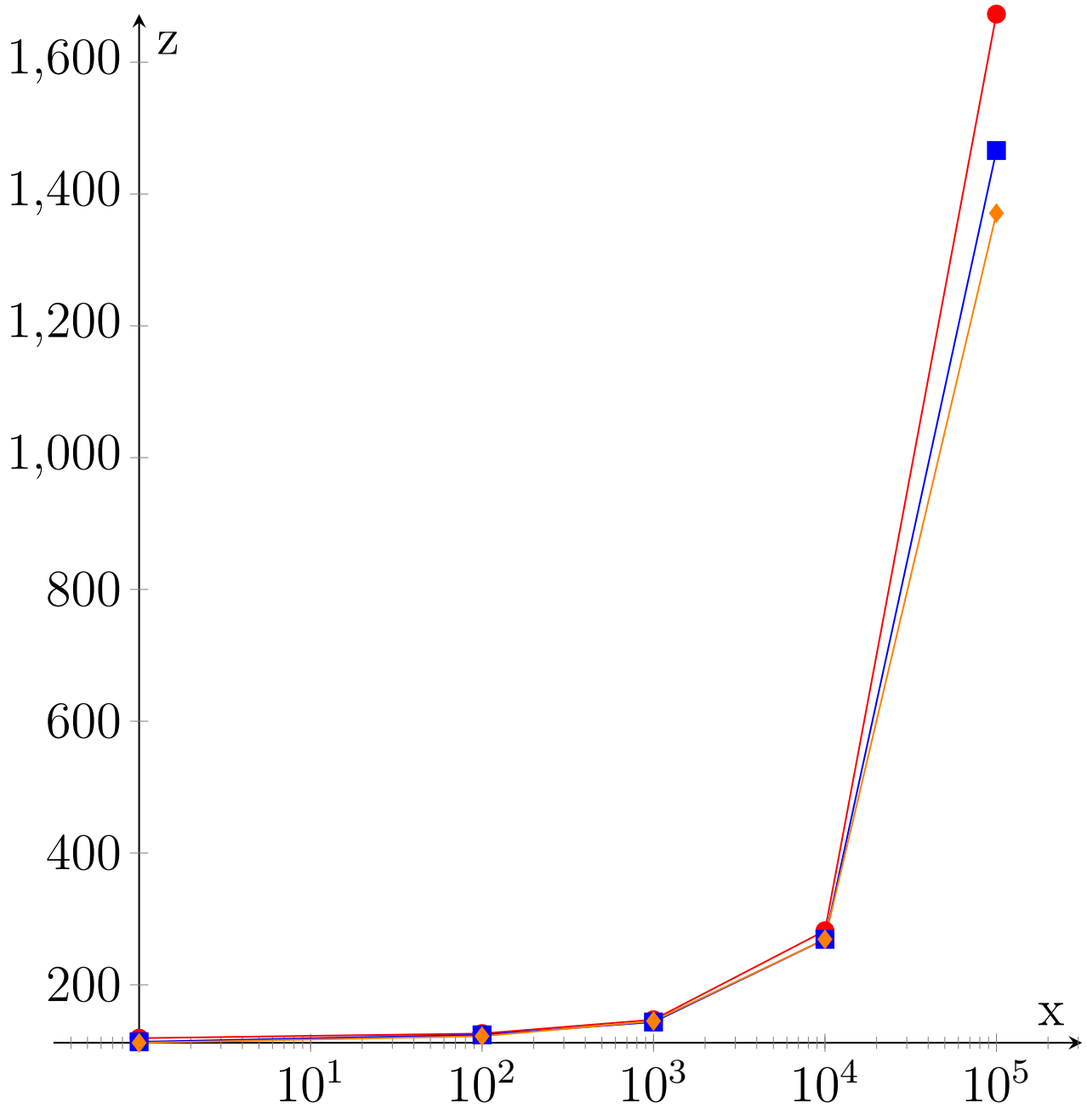

Abbildung 7.4: Neo4REST parametriesierte Abfragen

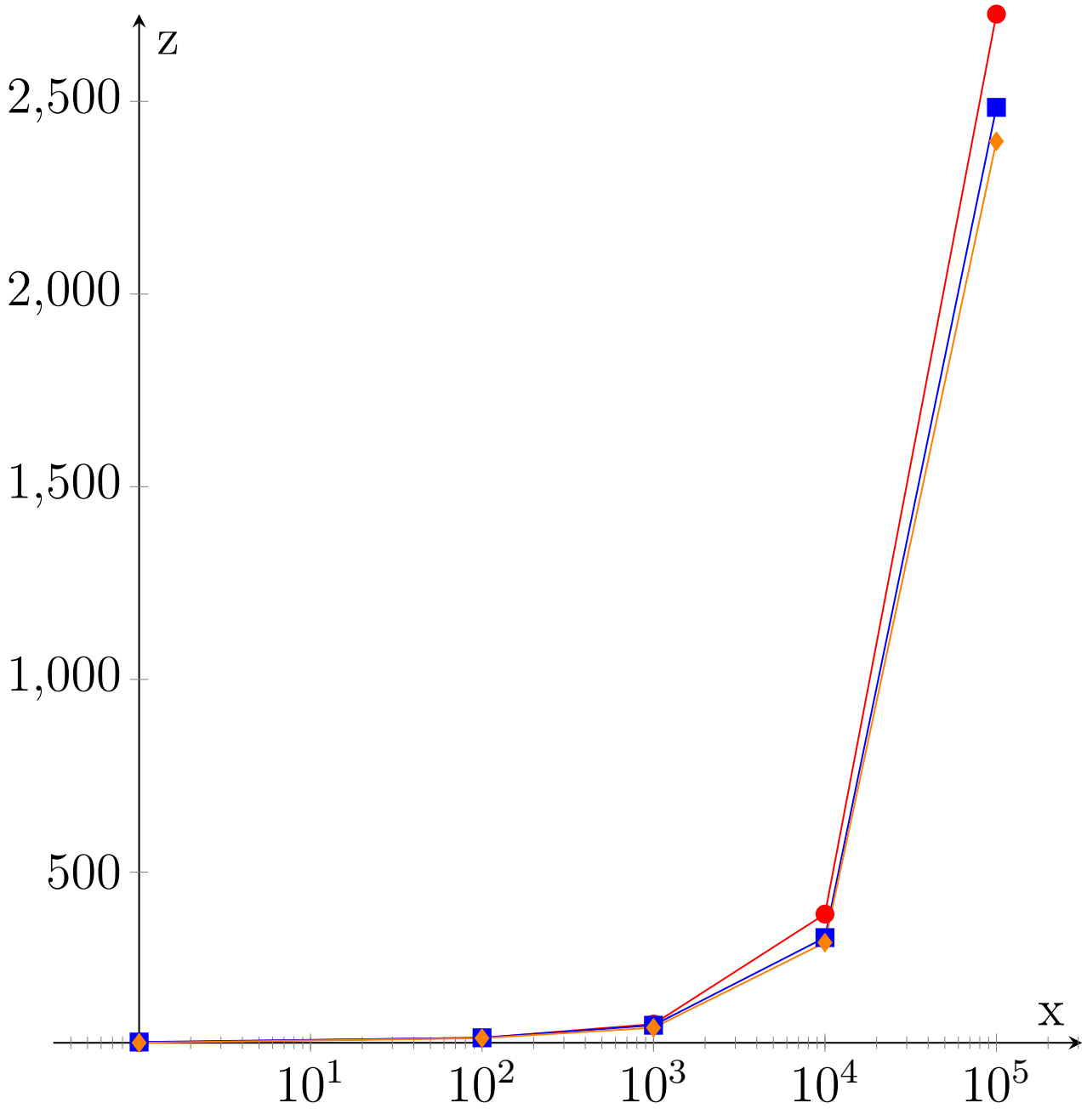

Abbildung 7.5: Neo4Graph parametriesierte Abfragen

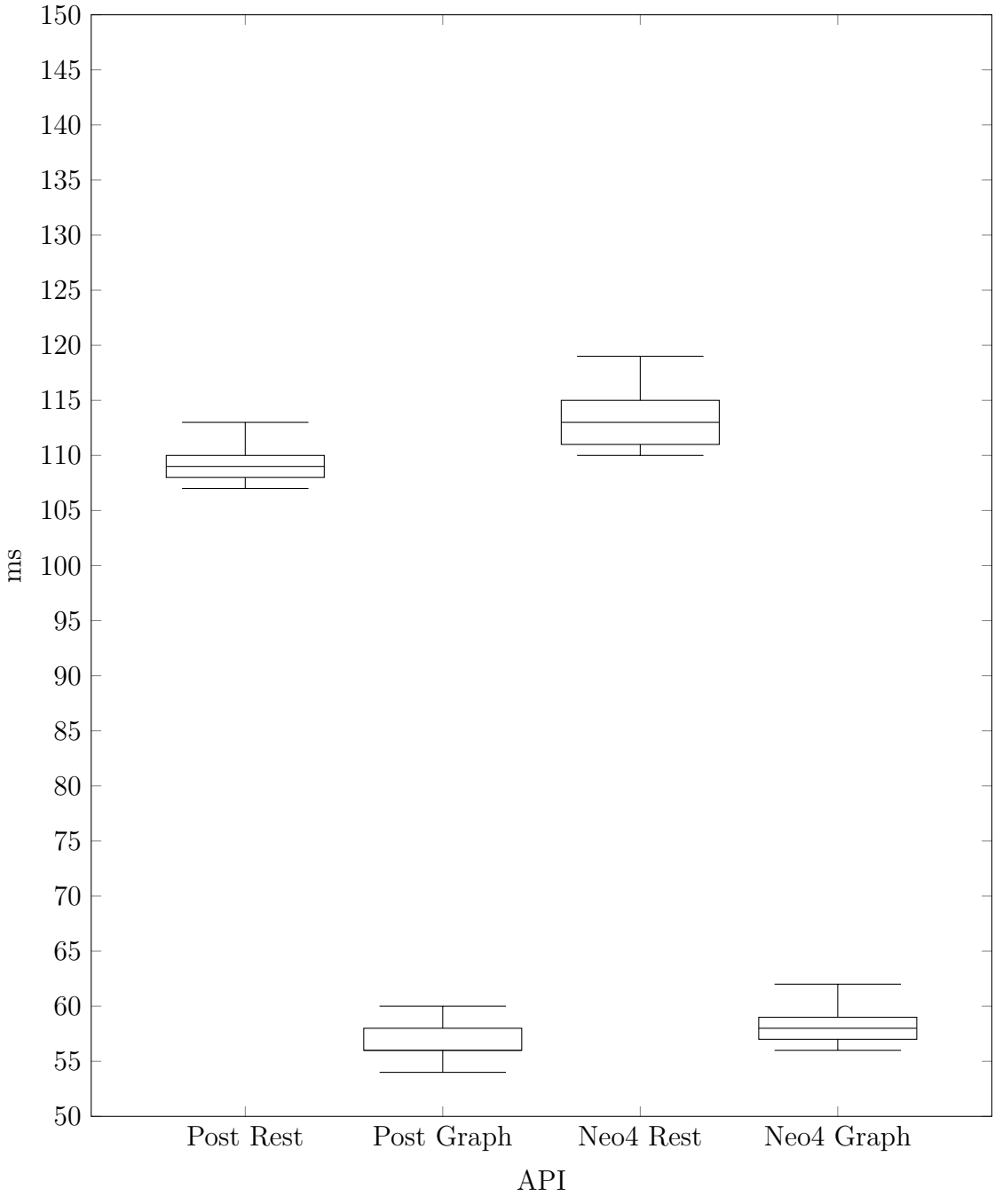

Abbildung 7.6: GET /api/persons/:pid

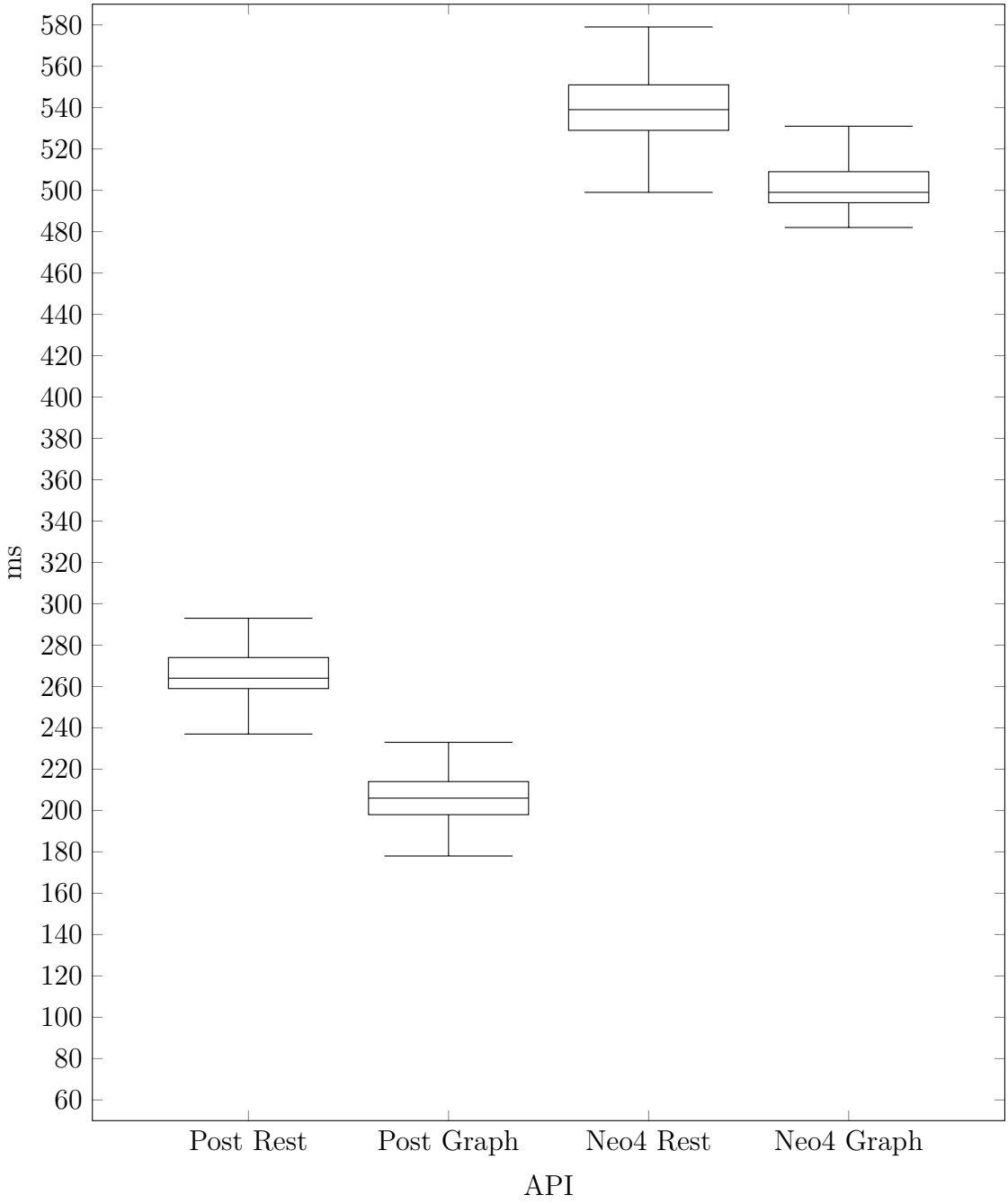

Abbildung 7.7: GET /api/persons

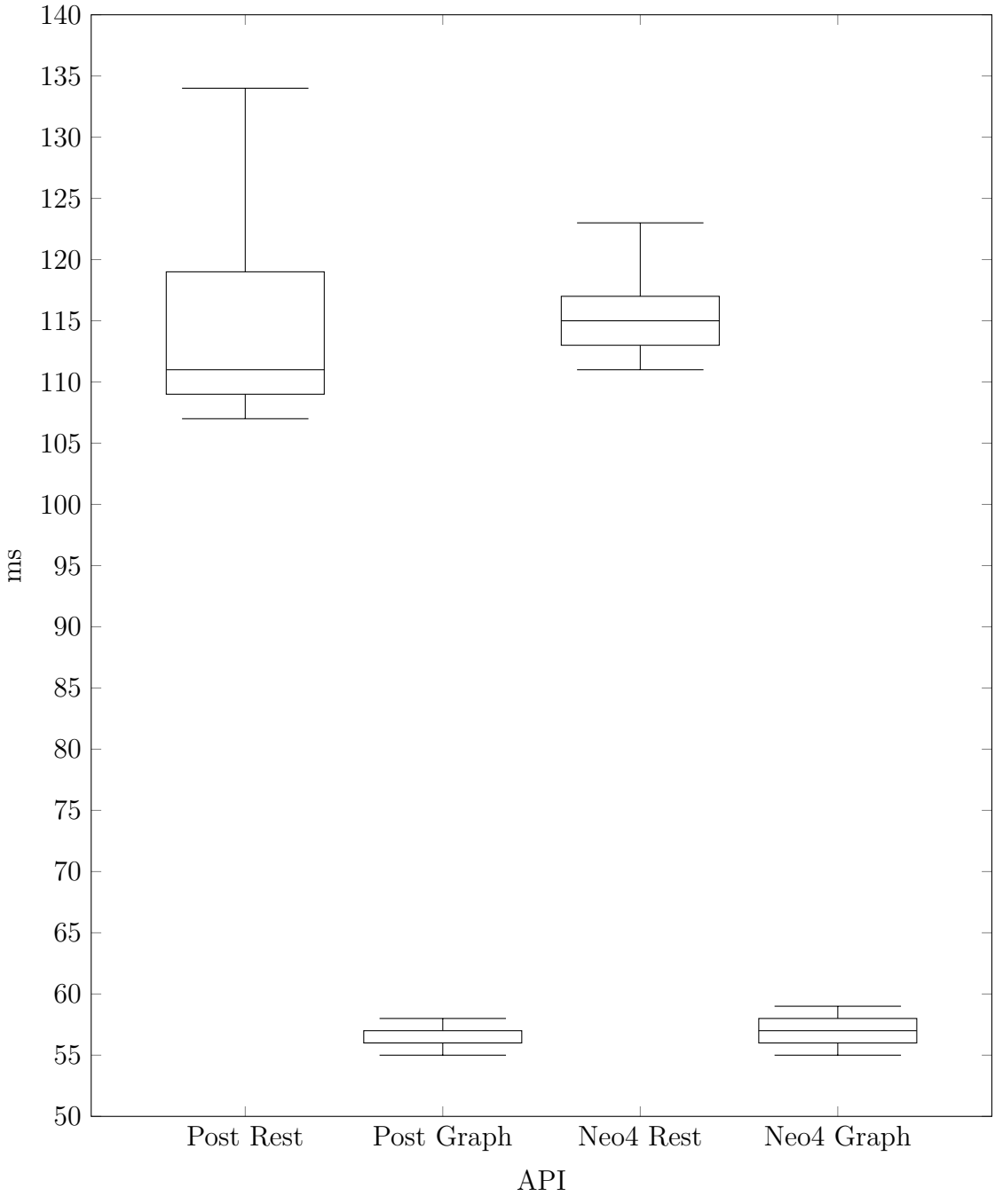

Abbildung 7.8: GET /api/persons/:pid/projects/issues

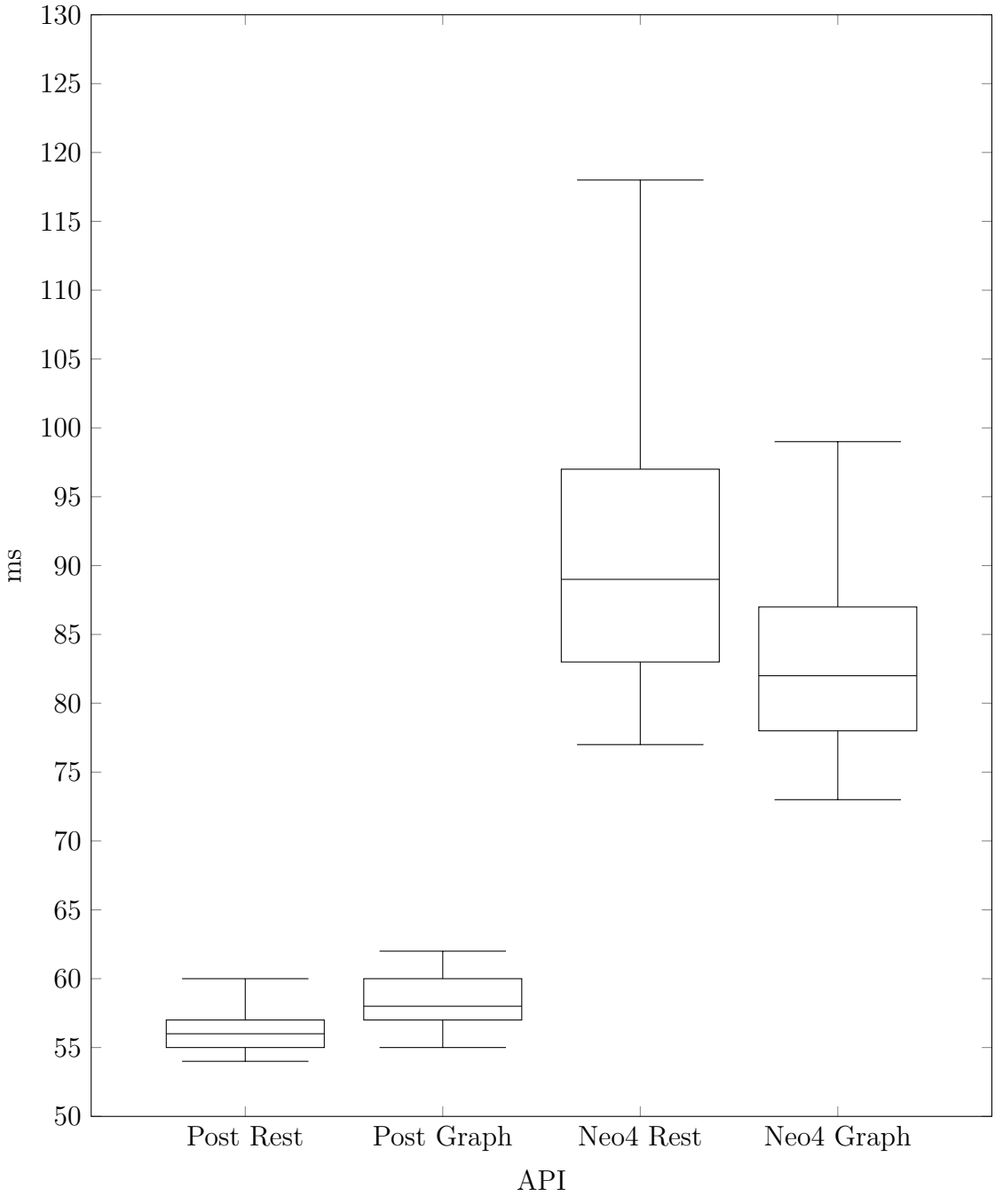

Abbildung 7.9: POST /api/persons/:pid/projects/:prid/issues

8 Diskussion

...

9 Fazit

...

Literaturverzeichnis

[1] Roy T. Fielding. Architectural styles and the design of network-based software architectures. 2000.

[2] 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.

[3] 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.

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

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

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

[7] Sri Lakshmi Vadlamani, Benjamin Emdon, Joshua Arts, and Olga Baysal. Can graphql replace rest? a study of their efficiency and viability. 2021.

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 22. Dezember 2024

(Nachname, Vorname)