HAW HAMBURG

BACHELORTHESIS
Jonathan Ströbele

Analyse und Vergleich eines Polystore mit einer relationalen Datenbank anhand spatio-temporaler Daten

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Jonathan Ströbele

Analyse und Vergleich eines Polystore mit einer relationalen Datenbank anhand spatio-temporaler Daten

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen
Zweitgutachter: Prof. Dr. Marina Tropmann-Frick

Eingereicht am: 29. April 2021

Jonathan Ströbele

Thema der Arbeit

Analyse und Vergleich eines Polystore mit einer relationalen Datenbank anhand spatio-temporaler Daten

Stichworte

Polystore, relationale Datenbank, Performance, spatial, temporal, ETL

Kurzzusammenfassung

Durch immer größer werdende Datenmengen mit heterogenen Datenformaten ist unter dem Stichwort “one size fits all” die Verwendung einer einzelnen Datenbank nicht mehr zeitgemäß. Aus den Anforderungen an Skalierbarkeit haben sich spezialisierte heterogene Systeme entwickelt. In dieser Arbeit wird ein Polystore mit einer relationalen Datenbank im Hinblick auf die Performance verglichen. Als Daten dienen spatio-temporale Fahrplanauskunftanfragen des öffentlichen Nahverkehrs. Der Polystore wurde in Python entwickelt und setzt sich aus PostGIS, TimescaleDB und MongoDB zusammen. Als Vergleich dient die relationale Datenbank MySQL. Verschiedene Anfragen wurden an beide Systeme gestellt und die Laufzeit gemessen. Die Untersuchungen haben gezeigt, dass heterogene Anfragen im Polystore nicht immer schneller sind, obwohl Anfragen auf den spezialisierten Datenbanken effizienter sind als in MySQL.

Jonathan Ströbele

Title of Thesis

Analysis and comparison of a polystore against a relationale database on the basis of spatio-temporal data

Keywords

Polystore, relational database, performance, spatial, temporal, ETL

Abstract

Due to the ever-increasing amounts of data with heterogeneous data formats, the use of a single database is no longer up-to-date under the keyword “one size fits all”. Specialized heterogeneous systems have emerged from the requirements for scalability. In this thesis, a Polystore is compared with a relational database with regard to performance. The data used are spatio-temporal timetable information requests from local public transport. The Polystore was developed in Python and consists of PostGIS, TimescaleDB and MongoDB. It is compared against the relational database MySQL. Various queries were made to both systems and the execution time was measured. The experiments have shown that heterogeneous queries in the Polystore are not always faster, although queries on the specialized databases are more efficient than in MySQL.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

Abkürzungen

CSV Comma-separated values

ETL Extract, transform, load

HVV Hamburger Verkehrsverbund

IoT Internet of things

LSB least significant bit

MARS Multi-Agent Research and Simulation

PoI Point of Interest

UUID Universally unique identifier

WKT Well-known text

ÖPNV Öffentlicher Personennahverkehr

1 Einleitung

1.1 Big Data

Der Begriff *Big Data* wird von Fasel und Meier in [13, S. 6] durch die "3 Vs" beschrieben:

Volume Die Daten liegen in großen Mengen vor, zum Beispiel im Tera- oder Petabytebereich. Wobei die Datenmenge auch immer relativ zur Organisation zu sehen ist, in der sie anfallen [25].

Variance Die Daten liegen in semi- oder unstrukturierter Art und Weise vor. Sie können in verschiedenen Formaten wie Text, Bild oder Audio vorliegen.

Velocity Die Geschwindigkeit, in der die Daten erhoben und verarbeitet werden.

In verteilten Internet of things (IoT) Sensornetzwerken können so zum Beispiel riesige Datenmengen in Echtzeit erhoben werden, für deren Auswertung entsprechend skalierende Systeme entwickelt werden müssen. Aber auch in klassischen Client-Server-Systemen können solche Datenmengen anfallen. Beispielsweise werden bei Amazon täglich mehrere Millionen Anfragen geloggt [10]. Die so erfassten Logdateien sind nicht zwangsläufig strukturiert, da hier zum Beispiel beliebige Parameter in verschachtelten Strukturen mit erfasst werden können. Durch die zunehmende Digitalisierung wird von Reinsel et al. prognostiziert, dass bis 2025 ein globales Datenvolumen von 175 ZB existieren wird [29] (1 ZB entspricht einer Milliarde Terabyte).

1.2 Datenhaltung

Für die so immer größer werdenden Datenmengen sind effiziente Wege der Speicherung und Abfragemöglichkeiten erforderlich. Neben der "bloßen" Speicherung ist aber auch

relevant, dass die Daten in einem für ihre jeweilige Eigenschaft optimierten System abgelegt werden und damit für die jeweilige Eigenschaft spezifische Abfragesprachen verfügbar sind.

Neben den schon seit Anfang der 1970er [8] bekannten relationalen Datenbanken haben sich weitere Datenbankmodelle entwickelt die heute als *NoSQL* bezeichnet werden [22]. Darunter Fallen etwa Key-Value-Stores [10] oder dokumentenorientierte Datenbanken. Zudem gibt es Spezialisierungen von Datenbanksystemen auf verschiedene Datentypen, wie temporale [31], spatiale [14] oder auch Graphendaten [1]. Diese spezialisierten Datenbanken stellen optimierte Abfragesprachen für die jeweiligen Datentypen bereit und erlauben somit eine bessere Abfrage der in ihr gespeicherten Daten.

In vielen Anwendungen tritt nicht nur eine Art von Daten auf, sondern ein Mix aus heterogenen Datentypen [7, 24, 34]. In anderen Szenarien werden heterogene Datenquellen zusammengeführt, um die Daten zu analysieren [6]. Solche Datentypen können zum Beispiel Text, Bild oder strukturierte, temporale oder spatiale Daten sein.

Für diese Zusammenhänge ist ein "one size fits all"-Ansatz, also die Verwendung einer einzigen Datenbank, nicht immer zielführend [33], da eine einzelne relationale Datenbank nicht auf alle Ausprägungen dieser Daten optimiert ist. Somit würden spezifische Optimierungen für Datentypen nicht genutzt werden können. Vorteilhafter sind Systeme, die sich aus verschiedenen spezialisierten Datenbanken zusammensetzen. Solche heterogenen Systeme werden von Stonebraker als *Polystore*-Systeme bezeichnet [32] und werden in Abschnitt 2.2 erläutert. Ein Beispiel für einen Polystore ist BigDAWG [11]. In diesem wurde der heterogenen Datensatz *Multiparameter Intelligent Monitoring in Intensive Care II* (MIMIC-II) [30] verarbeitet. Er stellt u. a. für Patientendaten (relational), ärztliche Berichte (Text) und Zeitserien (temporal) spezialisierte Abfragemöglichkeiten bereit.

1.3 Fahrplanauskunftsanfragen

Fahrplanauskunftsanfragen, wie sie im Öffentlicher Personennahverkehr (ÖPNV) vorkommen, enthalten eine zeitliche und geografische Eigenschaft. Es handelt sich somit um spatio-temporale Daten, die täglich in großen Mengen verarbeitet werden. Der Hamburger Verkehrsverbund (HVV) ist der Verkehrsverbund für die Metropolregion Hamburg. In seinem Zahlenspiegel für 2018 [16] gibt der HVV an, 10 184 Stationen und 763 Linien zu umfassen. In seinem Einzugsgebiet leben ca. 3,511 Millionen Menschen. Pro Tag

nutzen etwa 2,6 Millionen Fahrgäste den HVV. Über das Webportal Geofox[1] bietet der HVV seinen Kunden die Möglichkeit, Anfragen an die Fahrplanauskunft zu stellen und Routen mit dem HVV zwischen einem Start- und Zielpunkt, entweder für den aktuellen oder einen zukünftigen Zeitpunkt, abzufragen. Neben dem Webzugriff gibt es auch weitere Anfragekanäle wie die Android- und iOS-Apps des HVV oder mittels APIs. Der technische Dienstleister HBT des Geofox-Services gibt an, dass pro Monat 20 Millionen Fahrplanauskunftsanfragen gestellt werden [17].

Als zugrundeliegende Datenmenge dieser Arbeit werden die anonymisierten Logs des Geofox-Portals aus dem Jahr 2017 verwendet. Die Logs liegen im Comma-separated values (CSV) Format vor. Sie weisen folgende Eigenschaften auf:

Spatiale Daten Die Anfragen enthalten den gewünschten Start- und Zielort.

Temporale Daten Die Anfragen enthalten den Zeitpunkt, *wann* eine Anfrage gestellt wurde, und *für* wann diese Anfrage gelten soll.

Artefakte Die Anfragen enthalten weitere Anfragedetails, wie z. B. Referrer, Suchparameter oder interne Steuerparameter.

1.4 Motivation

Durch den Einsatz von spezialisierten Datenbanken für die jeweiligen Datentypen in der Polystore-Architektur bietet sich dem Anwender die Mächtigkeit von drei Anfragesprachen. Damit können sehr komplexe Anfragen an das System gestellt werden, was in einer einzelnen Datenbank nicht möglich wäre. Die so verwalteten Daten können für verschiedene Anwendungsfälle verwendet werden. Zum einen sind die Daten für Analysen relevant. Die gesammelten Anfragedaten können zur Untersuchung des Verhaltens der Fahrgäste verwendet werden. Hier sind verschiedene Szenarien von Interesse, wie das Identifizieren hoch frequentierter Stationen, Visualisieren von zeitlichen Anfragevolumen zum Erkennen von Mustern oder die Verteilung verschiedener Anfragekanäle an das System. Dies sind komplexe Anfragen über verschiedene Stores, deren Performance möglichst schnell sein sollte, um Ergebnisse direkt zu erhalten. Mit den Erkenntnissen solcher Analysen können Rückschlüsse auf die Nutzung des Verkehrssystems geschlossen werden und damit Entscheidungen für Optimierungen getroffen werden.

[1] https://www.geofox.de/

Zum Anderen bietet sich die Verwendung der Daten auch im Kontext von Simulationssystemen wie Multi-Agent Research and Simulation (MARS) [23] an. Im, mit MARS realisierten Projekt SmartOpenHamburg[2], wird das Mobilitätsverhalten in der Hamburger Metropolregion simuliert und dient damit als Entscheidungshilfe für "Was-Wäre-Wenn-Szenarien"[3]. In diesem Anwendungsfall wird nicht immer zwingend der gesamte Kontext der Daten benötigt. Für einen GIS-Layer zum Beispiel könnten nur die spatialen Informationen des Polystores relevant sein. Die Polystore-Architektur erlaubt es in diesem Fall ausschließlich auf den jeweils benötigten Daten zu arbeiten. Andere vorhandene Daten müssen nicht berücksichtigt werden und verlangsamen bzw. verkomplizieren die Anfragen somit nicht.

Die Laufzeit der Abfragen an solcher Systeme sollte so gering wie möglich sein. In dieser Arbeit soll die Hypothese untersucht werden, **ob ein Polystore für spatio-temporale Daten niedrigere Abfragegeschwindigkeiten bietet als eine klassische, relationale Datenbank**. Daher wurde ein Polystore entwickelt, der die Datenbanksysteme PostGIS, TimescaleDB und MongoDB enthält. Die Fahrplanauskunftsanfragen wurden semantisch zerlegt und auf diese Systeme aufgeteilt. Für die relationale Datenbank wird das weit verbreitete Datenbanksystem MySQL verwendet. Für beide Architekturen wurden äquivalente Abfragen formuliert und deren Laufzeit gemessen. Um einen Eindruck von den Laufzeiten bei verschieden großen Datenmengen zu erhalten, wurden die Anfragen auf drei verschieden großen Datenmengen durchgeführt.

[2] https://www.smartopenhamburg.de/
[3] https://mars-group.org/projects-using-mars/smartopenhamburg/

2 Grundlagen

In diesem Kapitel werden zuerst die Multi-Datenbanksysteme genauer definiert und anschließend die Logdateien des Geofox-Portals beleuchtet.

2.1 Datenintegration

Um Daten aus verschiedenen Quellen oder Systemen zugreifbar zu machen, unterscheiden Rahm et al. zwischen der *physischen* und *virtuellen Datenintegration* [28, S. 63].
Bei der physischen Integration werden heterogene Datenquellen eingelesen, transformiert und in eine globale Datenbank kopiert (Extract, transform, load (ETL)). Die Daten werden damit in einer zentralen Datenbank mit einem neu definierten globalen Schema bereitgestellt. Die Erstellung dieses Schemas (Schemaintegration) und die Implementierung und Durchführung des Imports (ETL) ist mit großem Aufwand verbunden [27, 36]. Dieser Ansatz kommt in Data-Warehouse-Architekturen zum Einsatz.
Bei der virtuellen Integration verbleiben die Daten in ihren jeweiligen Quellsystemen. Mittels einer Mediator/Wrapper-Architektur wird ein System etabliert, in dem der Mediator Anfragen entgegennimmt und in Teilanfragen an die verschiedenen Datenquellen umwandelt. Die Teilanfragen werden von den Wrappern der jeweiligen Quellen bearbeitet. Dabei entfällt der Aufwand die Daten kopieren zu müssen. Zudem bleibt die Anfragemächtigkeit der Teilsysteme erhalten. Allerdings müssen die Daten durch den Mediator eingesammelt und zusammengefasst werden.

2.2 Architekturen

Von Tan et al. [34] wurde eine Taxonomie zur Einordnung von Multi-Datenbanksysteme vorgeschlagen. Die Systeme werden anhand von zwei Hauptmerkmalen unterschieden.

Erstens anhand der Anzahl der verwendeten Datenbanksysteme, ob ein heterogener Datenspeicher vorliegt, oder ob nur ein einzelner Datenspeicher verwendet wird (homogen). Zweitens wie der Anwender Daten aus dem System abfragen kann, ob eine oder mehrere Anfragesprachen verwendet werden. Daraus ergeben sich die folgenden vier Ausprägungen:

Föderiertes Datenbanksystem

Die Daten liegen in einem homogenen Datenspeicher vor, und werden mit einem globalen Schema beschrieben, gegen das der Benutzer mit einer einzigen Abfragesprache Anfragen formulieren kann. Diese Abfragen müssen dann auf die darunter liegenden Schemas der Datenbanken gemapped werden. Hierin liegt auch eine Schwierigkeit dieser Architektur, da dieses Mapping nicht immer sicher hergestellt werden kann [19].

Polyglot

Das System bietet auf Basis eines homogenen Datenspeichers verschiedene Anfragesprachen an. Dies kann hilfreich sein, um in verschiedenen Kontexten besser Anfragen formulieren zu können. Am Beispiel von Spark SQL [2] sind Anfragen an die Daten zum einen über eine prozedurale API möglich, zum anderen aber auch über SQL-Statements. Je nach Ziel des Anwenders kann so eine passendere Anfragesprache gewählt werden.

Multistore

Im Multistore System wird eine heterogene Datenbanklandschaft verwendet, die mittels einer *globalen* Anfragesprache angefragt werden kann. Hierbei muss allerdings die globale Sprache jeweils auf die darunterliegende Anfragesprache umgewandelt werden. Die Komplexität besteht darin, dass evtl. nicht alle Funktionen oder Möglichkeiten jeder verwendeten Datenbank voll ausgenutzt werden können, da immer ein entsprechendes Element in der globalen Anfragesprache existieren muss. Dem Anwender stehen also nicht zwangsläufig alle Funktionalitäten eines spezialisierten Stores für einen Datentyp zur Verfügung, sondern ist von der Implementierung des Multistores abhängig.

Polystore

Im Polystore-System werden die Ansätze aus Multistore und polygloten Systemen verbunden. Eine heterogene Datenbanklandschaft kann mit ihren jeweiligen nativen Anfragesprachen angesprochen werden. Der Anwender kann die zugrunde liegende Anfragesprache der jeweiligen Datenbank voll ausnutzen. Damit bietet das System dem Anwender eine große Mächtigkeit beim Formulieren von Anfragen. Die Herausforderung besteht darin, die verschiedenen Daten effizient bereitzustellen. Zwischen den unterschiedlichen Datenbanken müssen Wege gefunden werden, die heterogenen Anfragen des Anwenders auszuführen, und die selektierten Daten zurückzugeben.

Ein viel zitiertes Beispiel für eine Polystore Architektur ist BigDAWG [11, 12], in dem ein Polystore für den MIMIC-II Datensatz entwickelt wurde. Mittels *Islands* werden dort verschiedene Datenbank-Technologien (z. B. relational, Array-basiert) abstrahiert, zwischen den Islands können die Daten gecastet und kopiert werden, um die Ergebnismenge zu ermitteln. Ein weiteres Beispiel für ein Polystore-System ist CloudMdsQL [20]. In diesem Fall wurde eine SQL-artige Abfragesprache für heterogene Datenspeicher entwickelt. Die verschiedenen nativen Anfragesprachen werden in der globalen Anfrage als Subqueries eingefügt, die dann von den Systemen ausgeführt werden. Damit steht dem Benutzer das gesamte Spektrum des jeweiligen nativen Datenspeichers zur Verfügung. Im Polystore Myria [35] werden dem Anwender die Daten u. a. mit einer Python API bereitgestellt.

2.3 HVV CSV-Dateien

Grundlage dieser Arbeit sind die Logs der Anfragen an das Geofox-Auskunftssystem des HVV. Die Daten liegen im CSV-Format vor. Die Spalten enthalten verschiedene Informationen wie den Zeitpunkt der Anfrage, den Start- und Zielpunkt, sowie weitere Metainformationen zu dieser Anfrage. Eine Zeile entspricht jeweils einer Anfrage.

Das Geofox-System besteht aus sechs Instanzen, auf die die eingehenden Anfragen gleichmäßig verteilt werden. Jede Instanz schreibt pro Tag eine Logdatei, die man dieser durch eine ID im Dateinamen zuordnen kann. Das *Umschalten* zwischen den Tagen findet nicht zu einem fest definierten Zeitpunkt für alle Instanzen statt. Zusätzlich kann ein Benutzer eine Anfrage auch für ein zukünftiges Datum stellen. Daher ist es für die Auswertung der Daten nicht ausreichend, lediglich einzelne Dateien zu betrachten, es müssen alle Dateien berücksichtigt werden, um ein Gesamtbild der Daten zu erhalten.

An vier Tagen im Jahr 2017 ist der Datenbestand nicht vollständig: Am 26.3. und 5.10. fehlt jeweils eine Datei. Am 19.7. und am 3.8. fehlen jeweils fünf Dateien. Bei der Auswertung dieses Datensatzes ist dies zu berücksichtigen. Insgesamt gibt es damit 2178 ($= 365 \cdot 6 - (2 \cdot 1 + 2 \cdot 5)$) Dateien, welche 72,1 GiB umfassen.

Alle CSV-Dateien folgen dem gleichen Aufbau. Im Groben können damit Anfragezeitpunkt, Ursprung und Ziel, sowie die Verbindung abgeleitet werden. Im Folgenden wird auf die verschiedenen Eigenschaften im Detail eingegangen.

2.3.1 Struktur der Dateien

Die CSV-Dateien haben 28 Spalten, die von `V1` bis `V29` durchnummeriert sind, wobei die Spalte `V15` nicht vorhanden ist, da hier die IP-Adresse des anfragenden Kanals hinterlegt wird, die in den vorliegenden Daten aus Datenschutzgründen entfernt wurde (siehe Abschnitt 2.3.2). In Tabelle 2.1 sind die einzelnen Spalten beschrieben, mit ihrem jeweiligen Datentyp für die weitere Verarbeitung. Bei einigen Spalten ist keine Interpretation möglich, bzw. sie werden durch die Datenbeschreibung des HVV selbst als "intern" gekennzeichnet. Diese Daten haben keine Relevanz für die Auswertung und werden nicht weiter beachtet.

In Listing 2.1 sind exemplarisch zwei Datensätze dargestellt, wie sie in den CSV-Dateien vorkommen.

Tabelle 2.1: Beschreibung der Spalten der CSV-Logdatei

Spalte	Bedeutung	Verarbeitung
`V1`	Anfrage Timestamp	Temporal
`V2-4`	Start	Spatial
`V5-7`	Ziel	Spatial
`V8`	Datum	Temporal
`V9`	Bezugspunkt + Uhrzeit	Temporal
`V10`	*intern*	-
`V11`	via-Station	Artefakt
`V12-13`	*intern*	-

Spalte	Bedeutung	Verarbeitung
V14	Suchbedingungen	Artefakt
V15	*Request IP, nicht enthalten*	-
V16	Ergebnis-Status	Artefakt
V17	Route/Umstiege	Artefakt
V18	*intern*	-
V19	Tarifstufe	Artefakt
V20	Verwendeter Tarif	Artefakt
V21-22	*intern*	-
V23	Bearbeitender Server	Artefakt
V24	Referrer	Artefakt
V25	Isolierte Uhrzeit aus V9	Temporal
V26-29	*intern, unbekannt*	-

Listing 2.1: Zwei beispielhafte Datensätze aus den CSV-Dateien

```
1 V1;V2;V3;V4;V5;V6;V7;V8;V9;V10;V11;V12;V13;V14;V16;V17;V18;V19;
    V20;V21;V22;V23;V24;V25;V26;V27;V28;V29
2 1484546960624;#Osterfeldweg;gk(3564659|5923139|0);;#Knoopstraße/
    Bremer Straße;gk(3565139|5925526|0);;16.1.2017;Ab 7:09;Turbo
    ;1;station-position;intermediate-stops;"DesiredType=fasttrain
    &longdistancebus:10000";5 results;"Osterfeldweg:Knoopstraße/
    Bremer Straße:421:431:HHA-B:145_HHA-B";18;515;HVV;1,6;HVV;
    localhost:8811;gtieos;07:09;;0;0;0
3 1484546961575;#Neumünster;gk(3564191|5994272|0);;#Kaltenkirchen;
    gk(3563682|5967498|0);;16.1.2017;Ab -0:01;10.12.2017;1;
    station-position;;"HandicappedChanging=0|HandicappedWalkin=0|
    DesiredType=fasttrain,longdistancebus:10000";13 results;"Neum
    ünster:Kaltenkirchen:451:495:DB-EFZ:A1_DB-EFZ_Z";60;100004;SH
    ;4,75;HVV;localhost:8802;geofox.hvv.de;;;0;0;0
```

Spatial

Die spatialen Informationen umfassen den Ursprung und das Ziel der Anfrage. Der Ursprung ist in den Spalten `V2` bis `V4` kodiert. Wobei in `V2` der Name einer Station, eines Point of Interest (PoI) oder einer Straße (Adresse) steht, in `V3` eine Koordinate und in `V4` ein Ortsname, sofern es sich um eine Adresse oder PoI handelt.

Enthält der Name das Präfix `#` handelt es sich hierbei um eine Station aus dem Verkehrsnetz, bei dem Präfix `*` handelt es sich um einen PoI. Ist kein Sonderzeichen als Präfix vorangestellt, handelt es sich um eine Adresse.

Die Koordinaten sind in dem in Deutschland üblichen Gauß-Krüger-Koordinatensystem [18, S. 91] kodiert (EPSG:31467[1]), die sich aus einem Rechts- und Hochwert zusammensetzen. In den CSV-Daten sind sie wie folgt kodiert: `gk(3567870|5936661|0)`. Extrahiert man den Rechtswert (`3567870`) und Hochwert (`5936661`) kann die Koordinate in ihre entsprechenden WGS84-Darstellung konvertiert werden (Lat: 53.557051 Lng: 10.023019).

Analog dazu ist in den Spalten `V5` bis `V8` das Ziel der Anfrage kodiert.

Temporal

In der Spalte `V1` ist der Anfragezeitpunkt kodiert, zu dem der Benutzer die Anfrage gestellt hat. Er liegt als Unix-Timestamp mit Millisekunden-Anteil in koordinierter Weltzeit (UTC) vor.

In der Spalte `V8` ist das Datum hinterlegt, für wann die Anfrage gelten soll. In Spalte `V9` ist die dazugehörige Uhrzeit angegeben. Anstelle der Uhrzeit kann auch eine Zeitspanne vorhanden sein. Außerdem ist vermerkt, ob es sich bei dem Zeitpunkt um die Abfahrt oder Ankunft handeln soll.

Artefakte

In den restlichen Spalten liegen verschiedene Metainformationen und interne Steuerparameter vor. Für die vorliegende Analyse wurden nur nachfolgende Informationen berücksichtigt:

[1] https://epsg.org/crs_31467/DHDN-3-degree-Gauss-Kruger-zone-3.html

In Spalte V14 liegen Suchparameter des Benutzers. Damit kann eine Eingrenzung auf bestimmte Verkehrsmittel stattfinden, oder die Art der Verbindung genauer definiert werden (z. B. Schnelligkeit, Barrierefreiheit).

In Spalte V17 ist die Route des ersten Suchergebnisses hinterlegt. Diese setzt sich aus einem oder mehreren Abschnitten zusammen. Ein Abschnitt weist jeweils den Namen der Linie und der Ein- und Aussteigestationen aus, sowie die jeweilige Uhrzeit, wann ein- oder ausgestiegen werden soll.

In Spalte V24 liegt eine Referrer-Information vor. Sie besteht aus dem Namen des technischen Kanals, über den die Anfrage ausgelöst wurde. Damit kann z. B. differenziert werden, ob die Anfrage über das Web/Geofox-Portal, eine Smartphone-App oder eine API-Schnittstelle gestellt wurde.

2.3.2 Anonymisierung

Die der Arbeit zugrundeliegenden Daten wurden anonymisiert bereitgestellt. Das bedeutet, dass keine Request-IP-Informationen enthalten sind. Eine Zuordnung von mehreren Anfragen an einen Benutzer ist damit nicht möglich. Wobei dies selbst beim Vorhandensein der IP nicht möglich wäre, da zum einen nur IPs aus Aufrufen über die Website geloggt werden und zum anderen bei Zugriffen über APIs nicht die IP des Benutzers, sondern die des jeweiligen Systems geloggt wird [15].
Der Dokumentation [15] ist zu entnehmen, dass automatische Testanfragen des Herstellers, durch ihre IP-Adresse identifiziert werden können. Das ist in unserem Datensatz damit nicht möglich.

Außerdem sind Anfragen an konkrete Kombinationen aus Straße und Hausnummer dahingehend anonymisiert, dass die Hausnummer und Koordinaten entfernt wurden. Alle Anfragen an eine bestimmte Hausnummer sind somit nur der Straße zuzuordnen.

2.3.3 Aussagekraft der Daten

Es ist hervorzuheben, dass es sich bei den Daten um die an das Auskunftsportal gestellten Anfragen handelt und *nicht* um reale Fahrten im ÖPNV. Benutzer können Anfragen mehrfach stellen, oder eine Fahrt nicht antreten. Durch die Limitierung, dass Anfragen keiner Session, also einem einzelnen Benutzer, zugewiesen werden können, ist es nicht

möglich mehrfache, identische Anfragen eines Benutzers auf einen Datensatz zu reduzieren, wenn zum Beispiel die Auskunft im Browser neu geladen wird und damit die identische Anfrage erneut gestellt wird. Darüber hinaus kann durch die fehlende IP-Adresse der interne Testtraffic aus dem HVV-Netz nicht herausgefiltert werden, was die Ergebnisse verzerrt, da der Umfang der Testanfragen nicht bekannt ist.

2.3.4 Fachliche Sicht

Zusammengenommen besteht eine einzelne, gültige Anfrage immer aus den folgenden Attributen:

- Ursprung
- Ziel
- Anfragezeitpunkt
- Anfragegültigkeit
- Artefakte (Suchparameter, Route, Referrer)

In Abb. 2.1 ist dies in einem ER-Diagramm dargestellt.

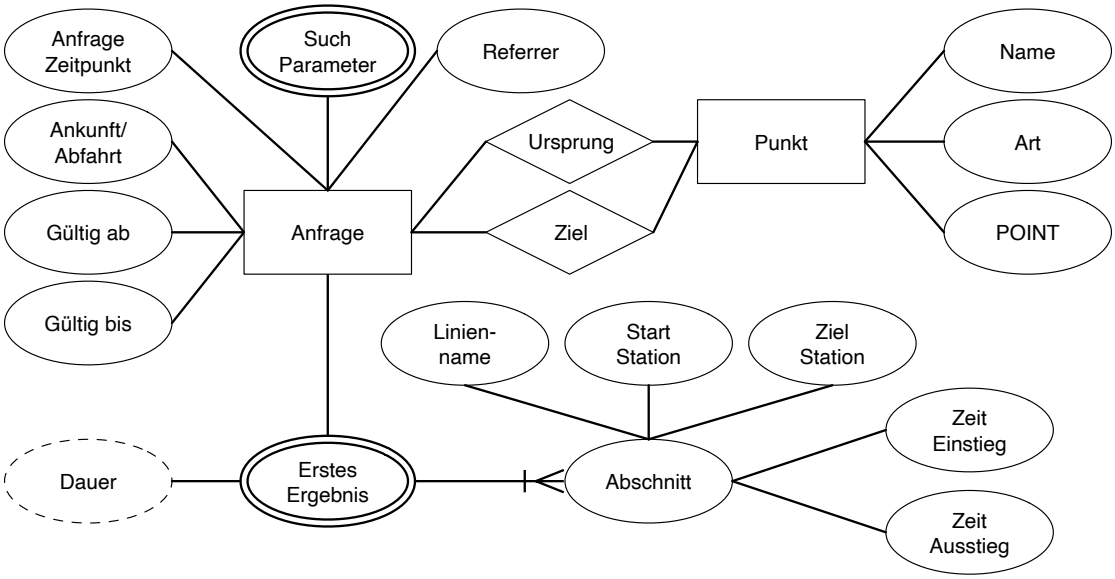

Abbildung 2.1: ER-Diagramm einer Fahrplanauskunftsanfrage

3 Architektur

In diesem Kapitel werden die beiden Architekturen erläutert, die zum Vergleich der Datenhaltung genutzt werden und die Basis für die Untersuchungen bilden.

3.1 Polystore

Der Polystore soll die spatio-temporalen Anfragedaten und die Artefakte umfassen. Dafür wurden in der Industrie etablierte Systeme für die spezialisierten Datenbanken gewählt. Die spatialen Daten werden in PostGIS abgelegt, die temporalen Eigenschaften in TimescaleDB. Die übrigen Artefakte wie Referrer und Parameter werden in einer MongoDB gespeichert. Auf die Auswahl dieser Systeme wird in den folgenden Abschnitten näher eingegangen. In Abb. 3.1 ist das Polystore-System dargestellt. Die drei Datenbanken können durch den Benutzer über eine Python-Schnittstelle angesprochen werden, über die die Daten zurückgeliefert werden. Es besteht auch die Möglichkeit die Daten individuell aus ihrem jeweiligen Store abzufragen. Dies kann für bestimmte Anwendungsfälle, wie zum Beispiel Simulationen, relevant sein, bei denen nur bestimmte Daten von Interesse sind.

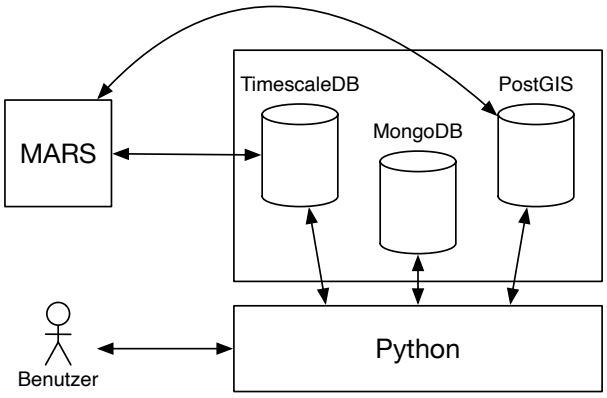

Abbildung 3.1: Kontextsicht Polystore

Um Abfragen durchzuführen wurde mit Python die Klasse `HVVQueryPoly` entwickelt, die es ermöglicht, in der nativen Anfragesprache der jeweiligen Datenbank eine Anfrage zu formulieren. Für jeden Store steht eine Methode bereit, in der mithilfe der nativen Anfragesprache eine Anfrage gestellt werden kann (`select_postgis(...)`, `select_timescale(...)` und `select_mongodb(...)`). Die drei Methoden selektieren die IDs der jeweiligen Ergebnismenge. Sie können beliebig kombiniert werden, wobei immer mindestens eine aufgerufen werden muss. Abschließend wird mit der Methode `results()` die finale Datenmenge abgerufen. Die so entwickelte Klasse folgt der Mediator-Wrapper-Architektur [37]. Sie bietet ein Interface *nach außen* und kapselt die jeweiligen Datenbankadapter.

Für die Formulierung der Teilanfragen muss dem Anwender das Schema der jeweiligen Stores bekannt sein. Ein automatisches Zuordnen der jeweiligen Attribute zu einem Store ist nicht möglich. Mit der Kenntnis des Schemas kann der Anwender aber sehr spezifische Anfragen stellen, unter Ausnutzung der jeweiligen Anfragesprache.

Um eine Anfrage nach dem Aufteilen in die drei Systeme wieder korrekt zusammenführen zu können, muss an jedem Datum in jedem System ein eindeutig identifizierendes Merkmal vorhanden sein. Dafür wurde jede Anfrage mit einem Universally unique identifier (UUID) versehen. Diese UUID dient in allen drei Systemen als Primärschlüssel, somit kann der Datensatz anhand dieser ID wieder zweifelsfrei *zusammengesetzt* werden. Bei der Vergabe wurde die Python-Funktion `uuid.uuid4()`[1] verwendet, die zufällige UUIDs v4 nach RFC 4122 [21] generiert. Durch die Größe von 128-Bit pro ID sind Kollisionen von IDs in der Praxis nahezu ausgeschlossen. Dadurch, dass diese ID in jedem der drei Systeme vorhanden ist, ist es möglich jede Anfrage aus jedem Datensatz kommend wieder zu rekonstruieren. Somit kann die Selektion der Daten in jedem System begonnen werden.

Für die Anfragen an den Polystore ist eine Applikationslogik nötig. Die einzelnen Stores können zwar individuell angefragt werden, allerdings erhält man damit auch nur die jeweiligen Informationen des Stores zurück. Um auch wieder die Daten der anderen Stores zu erhalten, muss zwischen den Stores eine Verbindung hergestellt werden (*join*). Dafür muss auf Applikationsebene eine Logik etabliert werden, die die selektierten IDs aus den verschiedenen Stores verwaltet und die Attribute der selektierten Datensätze zusammenführt.

[1] https://docs.python.org/3/library/uuid.html#uuid.uuid4

Um das Zusammenführen zwischen den drei Datenbanken zu realisieren, wurden verschiedene Ansätze geprüft. Wird in einem Store eine große Menge Daten selektiert, müssen alle IDs aus dieser Ergebnismenge in den weiteren Stores angefragt werden, um die dort vorhandenen Daten zu erhalten. Da dies zwischen den verschiedenen Systemen stattfindet, die keine gemeinsame Schnittstelle haben, muss dies in der Anwendung erfolgen. Um dies zu erreichen ist der einfachste Weg, die im ersten Store identifizierten Primärschlüssel im nächsten Store zu selektieren. Im Falle von SQL ginge dies über eine einfache Abfrage wie z. B. `SELECT * FROM table WHERE id IN(id1, id2, id3)`. Für wenige IDs ist dies unproblematisch, aber bei Anfragen mit vielen 1000 Datensätzen, kann so eine umfangreiche Anfrage nicht mehr verarbeitet werden. Zur Lösung dieses Problems werden temporäre Tabellen angelegt. Temporäre Tabellen sind nur für die jeweilige Datenbankverbindung eines Benutzers sichtbar und werden nach Beendigung der Verbindung automatisch gelöscht. Damit kann es zu keinen Konflikten zwischen mehreren Benutzern kommen. In diesen temporären Tabellen werden die selektierten IDs aus einem ersten Store importiert. Nun kann mittels der temporären Tabelle ein `JOIN` mit den eigentlichen Daten des Stores über die ID stattfinden und so die Daten selektiert werden. Dieses Vorgehen entspricht einem *Semi-Join* [4].

3.1.1 Spatial (PostGIS)

Die spatialen Attribute einer Anfrage, Ursprung und Ziel, werden in PostGIS gespeichert. PostGIS ist eine weitverbreitete spatiale Datenbank, die auf PostgreSQL aufbaut [26]. Sie implementiert den Simple Feature Access Standard des Open Geospatial Consortium und besitzt damit native Unterstützung für spatiale Datentypen wie Punkte und Polygone. Außerdem sind dadurch auch Funktionen für spatiale Daten vorhanden, zum Beispiel um die Distanz zwischen Punkten zu ermitteln oder zu überprüfen ob Punkte in einem bestimmten Gebiet liegen. Damit ist PostGIS als Datenbank für die spatialen Daten geeignet.

Die Daten werden dafür in zwei Tabellen geteilt. Es werden alle eindeutigen Punkte in der Tabelle `points` abgelegt. Damit werden die Ursprungs- und Zielpunkte normalisiert. Jeder Punkt ist damit anhand einer eindeutigen ID identifizierbar und muss nicht mehrfach gespeichert werden. Die Punkte werden mit ihrem jeweiligen Namen, Art und Koordinate gespeichert. Die Tabelle enthält dafür die Spalte `geom` mit dem spatialen Typ `geometry` in dem die Koordinate als Punkt-Geometrie (`POINT()`) abgelegt wird.

Die Anfragen Tabelle `requests` enthält die UUID und die Referenzen auf die Orte für Ursprung und Ziel. Das Schema ist in Abb. 3.2 dargestellt.

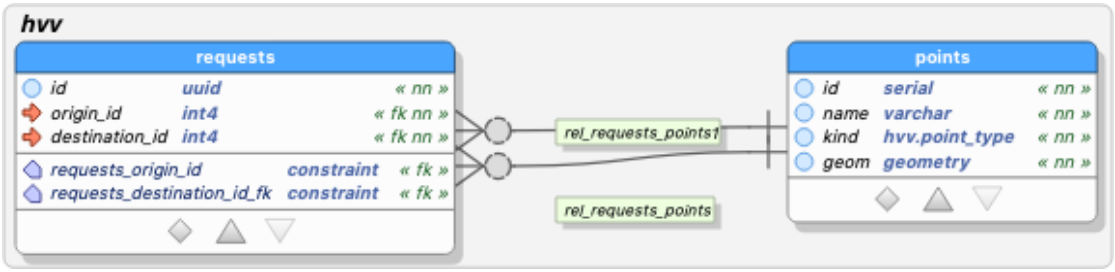

Abbildung 3.2: Schema der PostGIS Datenbank im Polystore

3.1.2 Temporal (TimescaleDB)

Die temporalen Attribute werden in TimescaleDB gespeichert. Dies ist eine temporale Datenbank, die wie PostGIS auf PostgreSQL aufbaut. Sie ermöglicht die Speicherung von temporalen Daten in sogenannten *hypertables*. Diese verhalten sich dem Benutzer gegenüber wie normale Tabellen, dienen allerdings nur als Abstraktion vieler kleinerer Tabellen (*chunks*), die jeweils einen Teil der Daten enthalten. Dieser Prozess wird transparent durch die TimescaleDB verwaltet, sodass der Benutzer wie gewohnt mit den Daten arbeiten kann. Zusätzlich bietet TimescaleDB verschiedene optimierte Funktionen zur Selektion der Daten an und erweitert damit die normale SQL Syntax von PostgreSQL.

Die Anfragedaten werden in einer einzelnen Tabelle `requests` gespeichert, in der Anfragezeitpunkt, Gültigkeitszeitraum, sowie die Information, ob es sich um Ankunft oder Abfahrt handelt, vorliegen.

3.1.3 Artefakte (MongoDB)

Alle weiteren Artefakte einer Anfrage werden in MongoDB abgelegt. Die MongoDB bietet sich als weit verbreitete dokumentenorientierte Datenbank an. Die Artefakte werden als JSON-artiges Dokument abgelegt. Auch verschachtelte Strukturen, wie Suchparameter, lassen sich damit abbilden und durchsuchen. Ein weiterer Vorteil der schemalosen Dokumente ist, dass bei zukünftigen neuen Artefakten keine Änderung an der Datenbank erforderlich ist und diese direkt importiert werden können.

Jedes Dokument in der MongoDB erhält die zuvor für diese Anfrage definierte UUID als Primärschlüssel. Diese wird im Feld `_id` gespeichert und muss zwingend gesetzt werden. Hervorzuheben ist, dass hier vom MongoDB-Standardverhalten abgewichen wird. Normalerweise wird eine von MongoDB definierte `ObjectId` als Primärschlüssel für ein Dokument verwendet. Die Verwendung einer UUID wird aber ausdrücklich unterstützt. Es ist hierbei darauf zu achten, dass im MongoDB Treiber für Anfragen explizit der Support für v4 UUIDs aktiviert wird. In der MongoDB-Standardeinstellung kann es hier zu Abweichungen in der Interpretation[2] der UUID bei verschiedenen Treibern kommen und eine korrekte Interpretation der ID ist dann nicht möglich und der gesuchte Datensatz damit nicht auffindbar.

3.2 Relationales Modell

Zum Vergleich mit dem Polystore wurde das relationale Modell der Datenhaltung in MySQL realisiert. MySQL bietet sich aufgrund seiner hohen Verbreitung als interessantes Vergleichssystem an. Sowohl in der Lehre als auch in der Industrie wird MySQL häufig eingesetzt. Im von DB-Engines durchgeführten weltweiten Ranking nach Popularität von Datenbanksystemen befindet sich MySQL auf dem zweiten Platz [9]. In Abb. 3.3 ist das normalisierte Schema der zuvor bereits vorgestellten HVV-Daten zu sehen. In der Tabelle `requests` finden sich die temporalen Attribute einer Anfrage wieder. Die spatialen Attribute für Ursprung und Ziel einer Anfrage sind über einen Fremdschlüssel aus der `points` Tabelle verbunden. Eine Anfrage ist jeweils mit einem fortlaufenden, numerischen Primärschlüssel versehen. Dieser dient in den weiteren Tabellen für Referrer, Parameter und Route als Fremdschlüssel. Die Parameter der Anfrage sind eindeutig über ihren Namen identifizierbar und werden in einer Verbindungstabelle mit den Anfragen und einem konkreten Parameterwert versehen. Da für einen Parameternamen mehrere Werte vorliegen können, sind mehrfache Eintragungen für Anfrage und Parameter möglich.

[2] https://pymongo.readthedocs.io/en/stable/examples/uuid.html#supported-uuid-representations

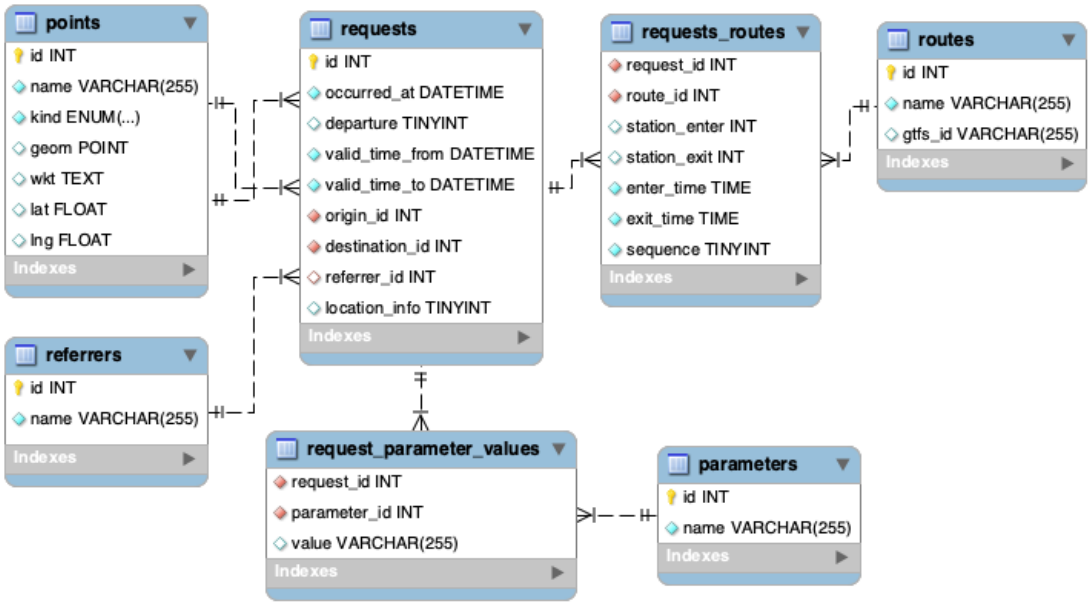

Abbildung 3.3: MySQL Schema Ansicht

Um Anfragen an diese Architektur durchzuführen, wurde eine Wrapper-Klasse `HVVQueryRel` entwickelt. Diese ermöglicht das Durchreichen von SQL-Anfragen an die Datenbank und das Abholen der Ergebnisse.

3.3 Docker-Container

Um eine einfache Installation sowie Reproduzierbarkeit [5] zu gewährleisten sind die beiden Architekturen als Container konzipiert. Mittels Docker können diese Container unabhängig vom zugrunde liegenden System auf verschiedenen Geräten, ohne auf Abhängigkeiten der Software achten zu müssen, installiert werden. Auch eine Installation in der Cloud, zum Beispiel in einem Kubernetes-Cluster, ist damit möglich.

In der `docker-compose.yml`-Konfigurationsdatei wird dabei für jede Datenbank ein konkretes Image mit einer Versionsnummer verwendet. Dadurch ist gesichert, dass immer die getestete Version der Datenbank verwendet wird, und zum Beispiel nach der Veröffentlichung einer neuen Hauptversion einer verwendeten Datenbank nicht etwas ungetestet verwendet wird. In der Tabelle 3.1 sind die für diese Arbeit verwendeten Docker Images und ihre Versionen dokumentiert.

Tabelle 3.1: Verwendete Docker Images

Datenbank	Version	Docker Image
PostGIS	3.0.0 (PostgreSQL 12.2)	kartoza/postgis:12.0[3]
TimescaleDB	1.7.4 (PostgreSQL 12.4)	timescale/timescaledb:latest-pg12[4]
MongoDB	4.4.1	mongo:4[5]
MySQL	8.0.22	mysql:8[6]

Für diese Arbeit bietet die Containerisierung der Architekturen einen weiteren Vorteil, da so durch wenig Aufwand beliebig viele Instanzen der Architekturen parallel betrieben werden können. Es müssen lediglich die von Docker an das Hostsystem gemappten Standardports der jeweiligen Datenbank geändert werden. Die Ports werden mittels einer Konfigurationsdatei gespeichert. Für die späteren Untersuchungen (Kapitel 5) auf verschieden großen Datenmengen ist dies von großem Vorteil, da so nur die jeweilige Konfiguration einer Instanz geladen werden muss. Eine spezielle Behandlung der Zugangsdaten für eine jeweilige Instanz in der Applikation, sowie wiederholtes Stoppen und Starten der Container, ist damit nicht nötig.

3.4 Scaffolding

Zur Automatisierung repetitiver Aufgaben werden häufig Shell-Skripte oder Makefiles erstellt, die diese Aufgaben ausführen. In diesem Projekt wurden in einem `Makefile` Targets entwickelt, die die folgenden Aufgaben durchführen: 1) **Anlegen der Schemas** in den jeweiligen Datenbanken. 2) **Löschen** aller importierter Daten in einer Instanz. 3) **Erstellen der Indizes** nach abgeschlossenen Import. Durch diese Automatisierung wird erreicht, dass die Benutzung des Systems weniger anfällig für Flüchtigkeitsfehler ist. Des Weiteren wird so die Arbeit mit dem System erleichtert, da diese Aufgaben nicht jedes Mal von Hand ausgeführt werden müssen.

[3] https://hub.docker.com/r/kartoza/postgis/
[4] https://hub.docker.com/r/timescale/timescaledb
[5] https://hub.docker.com/_/mongo
[6] https://hub.docker.com/_/mysql

4 Import

In diesem Kapitel wird der Importprozess der CSV-Dateien in die Architekturen beschrieben. Dafür wurde ein ETL-Prozess entwickelt, der die Daten einliest, bereinigt, bzw. in das benötigte Format konvertiert, und abschließend in die jeweiligen Datenbanken schreibt. Daten, die nicht importiert werden konnten, werden mit einem Fehlercode separat für eine anschließende Analyse abgelegt.

4.1 ETL-Prozess

Für den Importprozess der Daten wird ein, in Python entwickeltes, Skript verwendet. Das Skript greift in großem Maße auf die Pandas-Bibliothek[1] zurück, die sich sehr gut für die Verarbeitung von CSV-Dateien eignet [3]. Die CSV-Datei wird zunächst in einen sogenannten DataFrame geladen. Anschließend können auf den einzelnen Spalten der Daten Operationen ausgeführt werden, was sich ideal dafür eignet, die Daten zu konvertieren, bzw. zu bereinigen (wie in Listing 4.1 exemplarisch gezeigt). Die Datei `data.csv` wird in einen DataFrame eingelesen und anschließend die Werte in der Spalte `old` durch die Funktion `transform_data` bearbeitet und das Ergebnis in einer neuen spalte **`new`** abgelegt.

Listing 4.1: Einlesen einer CSV-Datei mit Pandas

```
1 df = pd.read_csv('data.csv', encoding="ISO-8859-1", sep=';',
      quoting=csv.QUOTE_NONE, header=0)
2 df['new'] = df['old'].apply(transform_data)
```

[1] https://pandas.pydata.org/

4.1.1 Escaping in CSV-Spalten

Sollte im Wert einer CSV-Spalte das Spaltentrennzeichen auftauchen, muss dieses escaped, bzw. das Feld eindeutig begrenzt werden, damit der Parser keine zusätzliche Spalte erkennt. Dies ist wichtig, da sonst die Anzahl der eingelesenen Spalten der Zeile nicht mehr zu der erwarteten Anzahl der Spalten passt.

In den vorliegenden Daten werden zwei Spalten durch Anführungszeichen (`"`) begrenzt, die Route- (`V17`) und Parameter-Spalte (`V14`). In beiden Fällen ist das Begrenzen allerdings nicht notwendig, da in den Spaltenwerten das Trennzeichen (`;`) nicht vorkommt. Allerdings kann es in den vorliegenden Daten passieren, dass eine Zeile der CSV-Datei nicht *zu Ende geschrieben* wurde, wobei die Anzahl der Spalten aber erhalten bleibt. Es kann dadurch zu dem Phänomen kommen, dass eine Feldbegrenzung zwar begonnen, aber nicht beendet wurde. Die nachfolgenden Spalten sind aber weiterhin aufgeführt. In Listing 4.2 ist das Problem dargestellt.

Listing 4.2: Fehlendes abschließendes Feldbegrenzerzeichen

```
1 col1;col2;col3
2 val1;"val2";val3
3 val1;"va;;
```

In der zweiten Zeile können die drei Spalten eingelesen werden und ihren Spaltennamen aus der ersten Zeile zugeordnet werden. Die Daten aus der dritten Zeile allerdings sind fehlerhaft. `val1` kann zwar noch `col1` zugeordnet werden, allerdings kann anschließend nach dem öffnenden Anführungszeichen in dieser Zeile kein abschließendes Begrenzungszeichen gefunden werden. Die Spalte `col2` endet nie. Es kommt zu einem Fehler. Zudem ist zu beachten, dass die nachfolgenden Spalten korrekt getrennt werden.

Um dieses Problem in dem vorliegenden Importprozess zu umgehen, werden die Daten ohne die Funktionalität der Begrenzungszeichen importiert (siehe `quoting=csv.QUOTE_NONE` in Listing 4.1). Dadurch enthalten die Werte im Pandas-DataFrame die zusätzlichen Anführungszeichen. Diese wurden mit einer Routine entfernt, die erkennt, ob das schließende Anführungszeichen fehlt. Ist das der Fall wird die Zeile aus den, zu importierenden, Daten entfernt.

4.1.2 Fehlende Daten

Es kann vorkommen, dass einzelne Felder einer Zeile nicht gefüllt sind. Dies kann verschiedene Gründe haben: Zum einen, um das *Nicht-gesetzt-sein* eines Wertes zu signalisieren, zum anderen können aber auch Fehler bei der Anfrage oder beim Erstellen der CSV-Datei aufgetreten sein. Daher muss beim Bearbeiten der einzelnen Werte in den Spalten immer explizit geprüft werden, ob die erwarteten Daten vorhanden sind. Sollten zwingend notwendige Attribute, wie zum Beispiel Ursprung/Ziel oder Zeitstempel fehlen, kann der Datensatz nicht importiert werden und wird verworfen.

4.1.3 Parsing von Zeitstempeln

Anfragezeitpunkt

Der Zeitpunkt der Anfrage durch den Benutzer liegt in Spalte `V1` als Zeitstempel in Millisekunden-Auflösung vor, z. B. `1484546975348`. Er kann durch die in Pandas vorhandene Methode `to_datetime()`[2] in ein Datumsobjekt zur weiteren Verarbeitung umgewandelt werden. Diese Umwandlung liefert für diesen zuvor genannten Zeitstempel das Datum 16.01.2017 06:09:35.
Auffällig ist, dass in den vorliegenden Daten teilweise *kaputte* Zeitstempel vorkommen, die den Zeitstempel in einer wissenschaftlichen Notation auswiesen, z. B. `1,483711e+12`. Durch diese Repräsentation ist es nicht mehr möglich den konkreten Anfragezeitpunkt zu rekonstruieren. Die Pandas Methode `to_datetime()` bietet mit dem Parameter `errors='coerce'` die Möglichkeit auftretende Fehler zu *ignorieren*, bzw. anstelle eines Ergebnisses den Wert `None` als Ergebnis der Umwandlung zurückzugeben. Dadurch können nicht umwandelbare Anfragen ausgefiltert werden.

Anfragegültigkeit

Die Information, für wann eine Anfrage gültig sein soll, wird in den zwei Spalten `V8` und `V9` kodiert. In `V8` ist das Datum in der Form `t.m.jjjj` hinterlegt. In Spalte `V9` ist zuerst der Bezugspunkt der Zeit kodiert, also ob es sich bei der Anfrage um eine Abfahrt zu dieser Uhrzeit (`Ab`) oder eine gewünschte Ankunft zu dieser Uhrzeit (`An`)

[2] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html

handelt. Dann folgt in den meisten Fällen die Uhrzeit als `s:mm`, in wenigen Fällen wird eine Gültigkeitszeitspanne im Format `s:mm-s:mm` angegeben. Zusammengenommen stellt `V9` sich so `<Bezugspunkt> <Uhrzeit>` dar. In Listing 4.3 sind einige Beispiele für diese Daten aufgeführt.

Listing 4.3: Anfrage Gültigkeit Beispiele

```
1 V8;V9
2 16.1.2017;Ab 7:09
3 16.1.2017;An 10:00
4 16.1.2017;An 10:00-12:00
```

Es wurde in einem Filter eine Funktion entwickelt, welche diese beiden Spalten konkateniert und aus Datum und Uhrzeit ein Datumsobjekt bildet, sowie den Bezugspunkt als Boolean zurückgibt (`Ab` entspricht `True` und `An` `False`). Aus dem ersten Beispiel in Listing 4.3 erhalten wir den String `16.1.2017 7:09` als Anfragezeitpunkt. Die Python-Standardbibliothek `datetime` liefert mit `strptime()`[3] eine Methode zum Einlesen eines als String formatierten Datums in einem bestimmten Format. Der so gewonnener Datumsstring kann so mit dem Format `%d.%m.%Y %H:%M` intuitiv eingelesen werden.

Das Einlesen mittels `strptime()` ist zwar unproblematisch, allerdings lässt sich ein deutlicher Geschwindigkeitsvorteil erzielen, wenn das Datumsobjekt manuell konstruiert wird. Dafür werden wie in Listing 4.4 gezeigt, die beiden Strings für Datum und Uhrzeit in ihre Einzelteile getrennt und als Integer gecastet und anschließend das Datumsobjekt aus den Einzelteilen erstellt.

Listing 4.4: Manuelles Datumsparsing

```
1 date_string = '16.1.2017'
2 time_string = '7:09'
3 date_parts  = date_string.split('.')
4 time_parts  = time_string.split(':')
5 date_object = datetime(int(date_parts[2]), int(date_parts[1]),
      int(date_parts[0]), int(time_parts[0]), int(time_parts[1]))
```

Durch den Umstieg auf das manuelle Erstellen der Datumsobjekte konnte die Geschwindigkeit des Importprozesses erhöht werden.

[3] https://docs.python.org/3/library/datetime.html#datetime.datetime.strptime

Fehlerhafte Daten Es lagen vereinzelt invalide Datumsangaben vor, z. B. wurde der Tagesteil um die Ziffer 1 ergänzt (Tag `116` anstelle von `16`) oder die Stundenangabe überschritt die `23`. So gestaltete Daten wurden nicht weiter berücksichtigt und verworfen. Des Weiteren gibt es Uhrzeiten die als `-0:01` angegeben sind, die mit dem vorgestellten Ablauf nicht zu parsen sind. Da die Bedeutung nicht aus der HVV-Dokumentation [15] hervorging, werden so formatierte Daten auch nicht weiter berücksichtigt.

4.1.4 Ursprungs- und Zielpunkt

Ursprung und Ziel sind in Gauß-Krüger kodiert. Um dieses Format in die Datenbanken zu importierten, ist es nötig, sie in ein gebräuchlicheres Format, bestehend aus Längen- und Breitenangabe (Latitude/Longitude), zu konvertieren. Dafür wurde der Python-Wrapper pyproj[4] für die PROJ[5]-Software verwendet, mit der zwischen Koordinaten und Referenzsystemen konvertiert werden kann. In Listing 4.5 ist exemplarisch gezeigt, wie von der Gauß-Krüger-Koordinate zum Well-known text (WKT) konvertiert wurde, welches in die Datenbank importiert werden kann.

[4] https://pyproj4.github.io/pyproj/stable/
[5] https://proj.org/

Listing 4.5: Konvertierten von Gauß-Krüger nach Lat/Lon mittels PROJ

```
from pyproj import Transformer
from shapely.geometry import Point
import re

transformers = {
    "2": Transformer.from_crs("epsg:31466", "epsg:4326"),
    "3": Transformer.from_crs("epsg:31467", "epsg:4326"),
    "4": Transformer.from_crs("epsg:31468", "epsg:4326"),
    "5": Transformer.from_crs("epsg:31469", "epsg:4326")
}

gk = 'gk(3567870|5936661|0)'
m = re.findall(r'\d+', gk)
x_coord = m[0]
y_coord = m[1]
zone = str(x_coord)[:1]

lat, lon = transformers[zone].transform(y_coord, x_coord)
print(lat, lon) # 53.55702780763139 10.023098057764892

wkt = Point(lon, lat).wkt
print(wkt) # POINT (10.02309805776489 53.55702780763139)
```

Neben den Koordinaten ist für einen Punkt auch sein Name vorhanden und durch das Präfix des Namens auch die *Art* des Punktes (Station, PoI oder Adresse). Wenn diese Informationen verbunden werden, wird eine Liste der Punkte erstellt und in die Datenbank importiert. Nach dem Einfügen in die Datenbank werden für eine Anfrage die jeweiligen IDs von Ursprungs- und Zielort ermittelt und dann referenziert.

Um auszuschließen, dass dieselben Punkte mehrfach in die Datenbank eingetragen werden, werden in einem ersten Import-Schritt zuerst ausschließlich alle Punkte importiert, und somit eine für den Datensatz vollständige Liste der vorhandenen Punkte ermittelt. Im anschließenden Import der Anfragen wird diese Liste wie ein Key-Value-Cache der bekannten Punkte verwendet. Anstatt neue Punkte zu erstellen, werden nur solche aus der Liste verwendet. Dadurch ist für das Ermitteln der IDs der Punkte kein Datenbankzugriff nötig, wodurch der Import schneller durchgeführt werden kann.

Da der Datensatz, wie bereits erwähnt, anonymisiert ist, sind bei Anfragen deren Start oder Ziel eine Adresse ist, die Hausnummer sowie Koordinaten entfernt. In diesen Fällen ist es nicht möglich einen eindeutigen Ursprung- oder Zielpunkt zu bestimmen. Aller-

dings ist in der hinterlegten Route einer Anfrage der Name der ersten und letzte Station enthalten. Im Falle von Adressen wurden die Punkte durch einen Abgleich auf den Stationsnamen ermittelt.
Um in diesen Fällen die Information, dass Ursprung und/oder Ziel aus der Route erschlossen wurde, nicht zu verlieren ist den spatialen Daten eine Bitmaske in der Datenbankspalte `location_info` hinzugefügt worden. Die Bits werden beginnend beim Least significant bit (LSB) gezählt (LSB 0). Im Bit 0 ist der Ursprung kodiert und im Bit 1 das Ziel. Ist das Bit an der jeweiligen Stelle 0, wurde der Ort nicht erschlossen, ist es 1 wurde er erschlossen. Damit ergeben sich die in Tabelle 4.1 gezeigten Möglichkeiten. Durch dieses Feld kann für spätere Auswertungen Rücksicht auf die Genauigkeit der Punkte genommen werden.

Tabelle 4.1: Mögliche Werte `location_info` Bitmaske für eine Anfrage

Bitmaske	Numerischer Wert	Bedeutung
`00`	0	Koordinaten waren vorhanden
`01`	1	Nur der Ursprung wurde erschlossen
`10`	2	Nur das Ziel wurde erschlossen
`11`	3	Ziel und Ursprung wurden erschlossen

Anfragen bei denen kein Ursprungs- oder Zielpunkt eindeutig identifiziert werden konnte, wurden nicht importiert.

4.1.5 Parameter

Eine Anfrage kann verschiedene Parameter umfassen. Diese sind als Schlüssel-Wert-Paare (Key-value pairs) kodiert. Diese Paare sind durch Trennzeichen als *flacher* String in einer Spalte der CSV-Datei vorhanden. In Listing 4.6 sind einige Beispiele zu sehen. Mit dem Zeichen `|` werden Paare voneinander getrennt. Innerhalb eines Paares wird der Parametername von seinem Wert mit dem Zeichen `=` getrennt. Zu beachten ist, dass ein Parameter mehrere Werte haben kann. Beim ersten Beispiel hat der Parameter `DesiredType` die beiden Werte `fasttrain` und `longdistancebus:10000`.

Listing 4.6: Beispiele Anfrage Parameter

```
DesiredType=fasttrain&longdistancebus:10000
HandicappedChanging=0|HandicappedWalkin=0|DesiredType=fasttrain,
    longdistancebus:10000
```

In der MongoDB des Polystores werden die Parameter als Objekt an dem Datensatz der Anfrage hinterlegt:

Listing 4.7: Parameter in der MongoDB als Objekt

```
{
    "parameters": {
        "HandicappedChanging": "0",
        "HandicappedWalkin": "0",
        "DesiredType": ["fasttrain", "longdistancebus:10000"]
    }
}
```

Im relationalen Modell ist dafür ein Aufteilen der Parameter in mehrere Tabellen nötig. Hierfür wird eine Zwischentabelle benötigt, welche die Parameter-ID, die Anfrage-ID und den eigentlich Parameterwert verbindet. Für Parameter mit mehreren Werten werden mehrere Einträge angelegt. Das zweite Beispiel aus dem Datensatz zuvor wird in der MySQL-Datenbank wie in Abb. 4.1 zu sehen gespeichert.

name	value
HandicappedChanging	0
HandicappedWalkin	0
DesiredType	fasttrain
DesiredType	longdistancebus:10000

Abbildung 4.1: Struktur der Parameter in MySQL

4.1.6 Route

Der Ergebnis-Reiseweg setzt sich aus beliebig vielen Segmenten zusammen, die jeweils einen Abschnitt von einer Starthaltestelle zu einer Zielhaltestelle darstellen. An jedem Segment ist zusätzlich die Ein- und Aussteigezeit an den Stationen vermerkt, sowie der

Namen der verwendeten Linie und des ausführenden Verkehrsunternehmens. Die Segmente sind mittels `|` voneinander getrennt, die Informationen in einem jeweiligen Segment sind dann weiterhin mit `:` getrennt. Damit lassen sich diese Werte einfach parsen.

4.1.7 Einfügen in Datenbanken

Nach Abschluss der Filterung aller Daten enthält der am Beginn des ETL Skriptes eingelesene DataFrame nur noch die bereinigten und somit validen Daten, die nun in die Datenbank importiert werden können. Aufgrund der großen Anzahl von Daten liegt die Schwierigkeit darin, den Import in einer vertretbaren Zeit durchzuführen.

In einem ersten, intuitiven Ansatz könnte man bei den PostgreSQL-Datenbanken versuchen, zeilenweise über die Daten zu iterieren und diese mit einem `INSERT` in die Datenbanken zu importieren. Dies würde eine Iteration über die Daten erfordern und für jeden Datensatz eine einzelne Datenbankanfrage, was sehr langsam ist. Dieser Ansatz lässt sich so erweitern, dass mehrere Datensätze pro Durchlauf in einem `INSERT`-Statement geschrieben werden – das grundsätzliche Problem bleibt aber bestehen. Zielführender ist es mit `COPY`[6] die Daten *am Stück* zu importieren. Dafür wurden die bereinigten Daten zuerst lokal als CSV-Datei neu gespeichert und dann mittels des `COPY` Befehls in die Datenbank importiert. Dieser Ansatz hat beim Import den schnellsten Durchlauf erzielt.

Im Falle der MongoDB wurde die `insert_many()`[7] Funktion des Python Adapters verwendet, mit der mehrere Dokumente auf einmal importiert werden können. Ein Umweg über eine CSV-Datei die dann mit `mongoimport`[8] importiert würde, wurde nicht weiter evaluiert, da bei einem ersten Versuch die UUID nicht korrekt importiert wurde. Damit wäre der Datensatz in der MongoDB nicht mehr zuzuordnen.

Wichtig für die Geschwindigkeit beim Import ist auch, dass noch keine Indizes auf den angelegten Datenbanktabellen vorhanden sind. Indizes sind zwar wichtig für eine bessere Performance beim Abfragen von Daten, beim Schreiben von vielen Daten in kurzer Zeit bedeutet dies allerdings, dass für jedes neue Datum der Index berechnet werden muss. Daher werden die Indizes erst nach Abschluss des Import-Vorgangs erstellt.

[6] https://www.postgresql.org/docs/12/populate.html#POPULATE-COPY-FROM
[7] https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.insert_many
[8] https://docs.mongodb.com/database-tools/mongoimport/

4.2 Ablauf

Um einen Import der Daten durchzuführen, müssen die folgenden Schritte ausgeführt werden:

1. Zuerst muss die in Kapitel 3 vorgestellte Architektur in Docker gestartet werden, sowie mit dem Make-Target `setup` die Schemas in den jeweiligen Datenbanken angelegt werden.
2. Mit dem Import-Skript für die Punkte müssen die verfügbaren Punkte aus den vorhandenen Daten eingelesen werden.
3. Mit dem Import-Skript für die Anfragen kann nun die gewünschte Anzahl an Anfragen importiert werden.
4. Nach Abschluss des Imports sollten mit dem Make-Target `index` die Indizes der jeweiligen Datenbanken angelegt werden, um eine effiziente Abfrage der Daten zu ermöglichen.

Das Import-Skript wird über die Kommandozeile gestartet. Als Parameter muss ein Ordnerpfad angegeben werden (`-d`), in dem eine beliebige Anzahl der Geofox-CSV-Dateien abgelegt ist. Zusätzlich kann optional die Anzahl der zu importierenden Anfragen mit dem Parameter `-c` angegeben werden. Ein exemplarischer Aufruf des Skriptes sieht wie folgt aus `python3 ./import-requests.py -d ./data/`. Dadurch werden alle CSV-Dateien aus dem Verzeichnis in die Instanz der Default-Konfiguration importiert. Wird dem Aufruf der Export `ENV=<name>` vorangestellt, kann damit die Konfigurationsdatei einer anderen Instanz verwendet werden. Ein Aufruf für die `10k` benannte Instanz würde so aussehen `ENV=10k python3 ./import-requests.py -d ./data/`.

5 Untersuchungen

5.1 Versuchsaufbau

Bei Anfragen an den Polystore werden verschiedene Stores mit individuellen Queries angefragt, bis das Gesamtergebnis vorhanden ist. Evtl. müssen durch die Anwendung auch Daten für eine Selektion der Daten zwischen den Stores kopiert werden. Im relationalen MySQL-Ansatz kann allerdings auch schon eine einzelne Query für das endgültige Ergebnis ausreichend sein.

Aus diesem Grund wird die Laufzeit vom Beginn der Anfrage bis zum Erhalt der Ergebnisse gemessen, um so die beiden Architekturen vergleichen zu können. Die Messungen werden mittels Python 3.9 durchgeführt. Um die benötigte Zeit einer Anfrage zu messen wird vor Beginn der Anfrage mittels des `time` Paketes[1] der aktuelle Timestamp abgefragt und erneut nach Erhalt der Ergebnisse. Dadurch kann mit der Differenz beider Zeitstempel die Dauer der Anfrage bestimmt werden. Jede Messung wird 20 Mal in Folge ausgeführt und dann gemittelt. Damit wird verhindert, dass nebenläufige Prozesse auf dem Testsystem das Messergebnis durch kurzfristige Auslastungen verfälschen.

Um abweichendes Verhalten bei verschiedenen Größen der Datenmengen zu untersuchen, wird jede Anfrage zudem auf je einer Instanz ausgeführt mit 10 000, 100 000 und 1 000 000 importierten Datensätzen. Die Größen wurden so gewählt, um jeweils mit dem Faktor 10 eine deutliche Volumenunterscheidung zu ermöglichen, den Abfrageprozess aber nicht auf mehrere Minuten auszudehnen.

Die Daten der drei Instanzen wurden mit dem Import-Skript importiert, beginnend jeweils mit den CSV-Dateien ab Montag, dem 2.1.2017 (erste Kalenderwoche 2017). Bei ca. 500 000 Anfragen am Tag[2] entspricht dies in der ersten Instanz etwa 2 %, in der zweiten 20 % und in der letzten 200 % der Anfragen eines Tages. Die temporalen Attribute

[1] https://docs.python.org/3/library/time.html#time.time
[2] Tagesdurchschnitt der importierten Daten für Januar 2017 entspricht 521 057 Anfragen.

der importierten Daten liegen damit größtenteils nahe beieinander, da die CSV-Dateien beginnend beim gleichen Zeitpunkt danach zeitlich aufsteigend importiert wurden. Die Verteilung der spatialen Attribute und Artefakte dagegen ist *willkürlich* und nicht vorherzusagen, da sie sich durch das Benutzerverhalten bei der Fahrplanauskunft ergibt.

5.1.1 Testsystem

Messungen werden lokal auf einem MacBook Pro 2015 ausgeführt. Durch die lokale Ausführung wird verhindert, dass nicht kontrollierbare Netzwerklatenzen die Messergebnisse beeinflussen. Das MacBook hat folgende Spezifikationen:

Tabelle 5.1: Testsystem Hard- und Software Merkmale

Merkmal	Ausprägung
Computer	MacBook Pro (Retina, 13-inch, Early 2015)[3]
Betriebssystem	macOS Catalina 10.15.7
Prozessor	3,1 GHz Dual-Core Intel Core i7
Arbeitsspeicher	16 GB 1867 MHz DDR3
Festplatte	512 GB PCIe-based flash storage
Docker	20.10.2, build 2291f61
Python	3.9.0

Tabelle 5.2: Datenbank Versionen

Datenbank	Version
PostGIS	3.0.0 (PostgreSQL 12.2)
TimescaleDB	1.7.4 (PostgreSQL 12.4)
MongoDB	4.4.1
MySQL	8.0.22

[3] https://support.apple.com/kb/SP715?locale=en_US&viewlocale=de_DE

5.1.2 Queries

In den folgenden Abschnitten werden die verschiedenen Queries an die beiden Architekturen vorgestellt. Zuerst wird ein einzelner Datensatz abgefragt. Anschließend wird die jeweilige spezialisierte Datenbank im Polystore mit dem relationalen Modell verglichen. Zuletzt werden heterogene Anfragen über mehrere Stores des Polystores mit der MySQL Datenbank untersucht.

5.2 Einfache Query

Query eines einzelnen Datensatzes mit seinen Spatialen-, Temporalen- und Artefakt-Attributen. Im MySQL-Ansatz (Listing 5.1) ist dafür neben der Query für die Anfrage noch ein komplexerer `JOIN` nötig um die Parameter aus der `1:n`-Beziehung von Anfrage und Parameter auszulesen.

Listing 5.1: Selektieren einer einzelnen Anfrage (MySQL)

```
1  -- Query Datensatz
2  SELECT r.*, ref.`name` as referrer
3  FROM requests r
4  JOIN referrers ref on r.referrer_id = ref.id
5  WHERE r.id = %s LIMIT 1
6
7  -- Query jeweils für Ursprung und Ziel
8  SELECT id, kind, lat, lng, name FROM points WHERE id = %s LIMIT 1
9
10 -- Query für Parameter der anfrage
11 SELECT p.`name`, rpv.value
12
13 FROM requests r
14
15 JOIN request_parameter_values rpv ON rpv.request_id = r.id
16 JOIN parameters p ON p.id = rpv.parameter_id
17
18 WHERE r.id = %s
```

Im Polystore sind je eine SQL-Query an die PostGIS und TimescaleDB und eine Query an die MongoDB nötig, deren Ergebnisse verbunden werden.

```
1 sql = "SELECT * FROM requests WHERE id = %s LIMIT 1" # PostGIS
2 sql = "SELECT * FROM requests WHERE id = %s LIMIT 1" # Timescale
3 mongo_results = self.c_m.find_one({'_id': id}) # MongoDB
```

Das Ergebnis der Messungen ist in Abb. 5.1 dargestellt. Man sieht, dass der Polystore mit dem MongoDB Document Store hier, ohne den umfangreichen JOIN über die Relationship-Tabelle, unabhängig von der Anzahl der Daten ähnliche Laufzeiten hat. Der relationale Ansatz hingegen wird mit zunehmenden Daten langsamer.

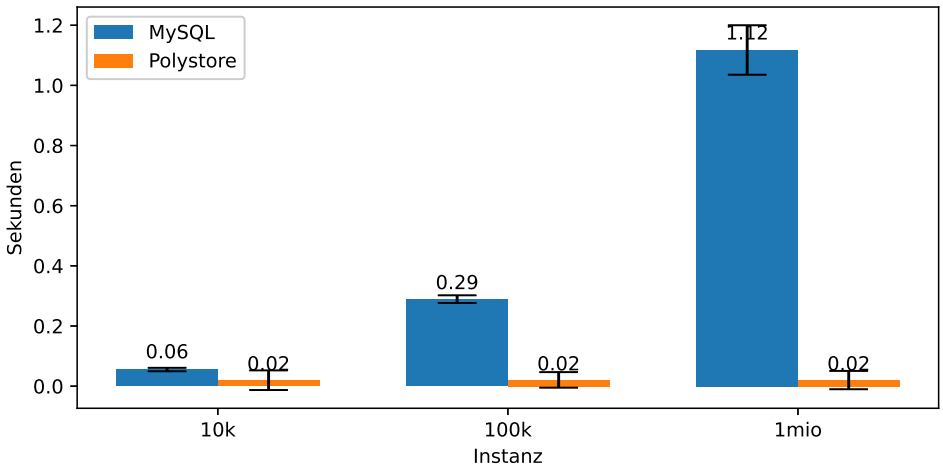

Abbildung 5.1: Anfrage auf eindeutige ID

5.3 Spatiale Abfrage

In dieser Anfrage soll die spatiale Funktion `ST_Within()` verglichen werden. Dazu werden alle Anfragen, deren Ursprünge innerhalb eines gegebenen Polygons liegen, gezählt. Der Test wird zuerst mit einem *einfachen* Polygon ausgeführt, das lediglich ein Rechteck ist. Als Zweites wird ein komplexeres Polygon verwendet, dass die Umrisse des Hamburger Bezirks Altona darstellt. Der Platzhalter `{wkt}` steht in den beiden Queries dabei jeweils für den WKT-String des Polygons.

Listing 5.2: Zählen der Anfragen mit Start in Polygon (MySQL)

```
SELECT COUNT(*) count
FROM requests r
JOIN points p ON r.origin_id = p.id
WHERE ST_Within(p.geom, ST_SwapXY(ST_GeomFromText('{wkt}',4326)))
```

Listing 5.3: Zählen der Anfragen mit Start in Polygon in PostGIS (Polystore)

```
SELECT COUNT(*) count
FROM requests r
JOIN points p ON r.origin_id = p.id
WHERE ST_Within(p.geom, ST_GeomFromText('SRID=4326;{wkt}'))
```

In den Ergebnissen für das einfache Polygon (Rechteck) (Abb. 5.2) ist zu sehen, dass MySQL schneller ist, im Polystore dauert es mit zunehmender Datenmenge länger. Bei Betrachtung eines Polygons *aus der echten Welt* (Abb. 5.3), dass die Umrisse des Bezirks Altona hat, zeigt sich, dass komplexere Anfragen an die PostGIS gegenüber der MySQL eine Verbesserung von 4 bis 5 Sekunden bringen.

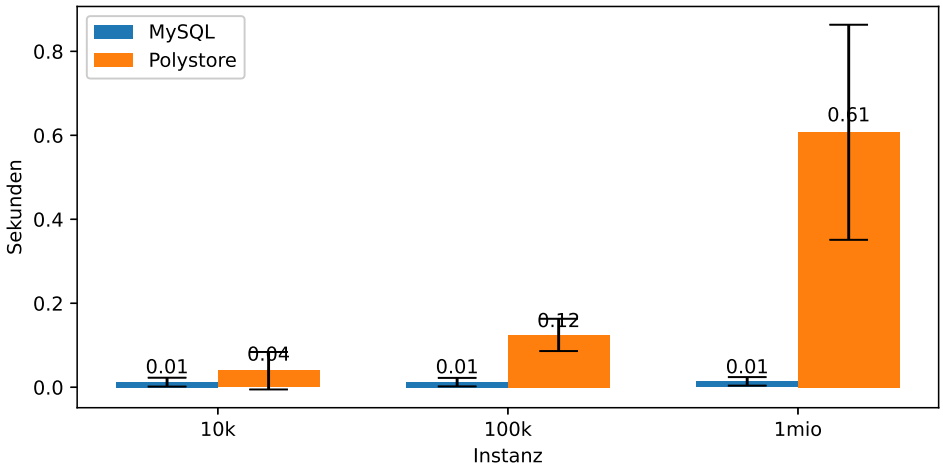

Abbildung 5.2: Anzahl der Anfragen mit Start in einfachem Polygon

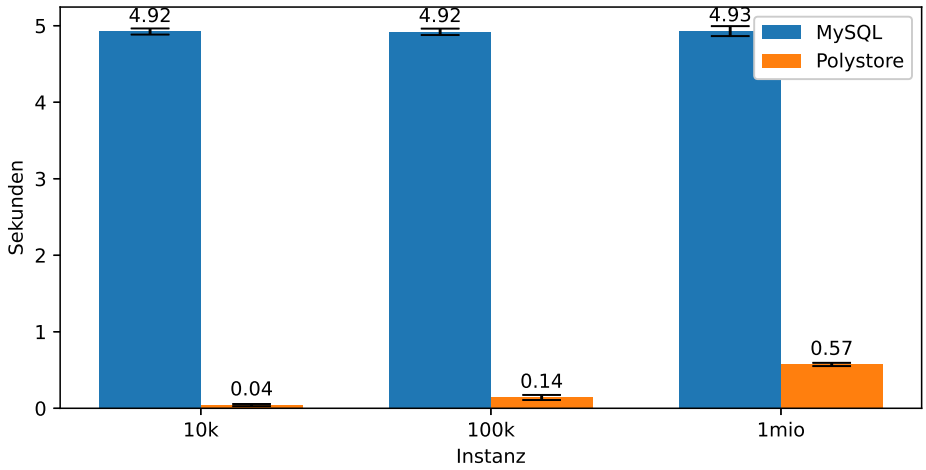

Abbildung 5.3: Anzahl der Anfragen mit Start in komplexem Polygon

5.4 Temporale Abfragen

Die folgenden Anfragen greifen auf temporale Eigenschaften zu. Zuerst wird eine gruppierte Anfrage und dann eine Selektion anhand eines Zeitintervalls durchgeführt.

5.4.1 Gruppieren nach Zeit

In dieser Anfrage soll eine temporale Eigenschaft verglichen werden. Alle vorhanden Anfragen werden anhand ihrer Gültigkeitszeit nach Stunden gruppiert und gezählt. In MySQL ist dafür eine Gruppierung anhand der einzelnen Datumsbestandteile nötig (Listing 5.4). In der TimescaleDB wird dies über die spezifische Funktion `time_bucket()` ermöglicht (siehe Listing 5.5).

Listing 5.4: Gruppieren nach Stunde (MySQL)

```
SELECT
    year(valid_time_from), month(valid_time_from),
    day(valid_time_from), hour(valid_time_from),
    count(*)
FROM requests

GROUP BY year(valid_time_from), month(valid_time_from), day(
    valid_time_from), hour(valid_time_from)
ORDER BY year(valid_time_from), month(valid_time_from), day(
    valid_time_from), hour(valid_time_from)
```

Listing 5.5: Gruppieren nach Stunde in TimescaleDB (Polystore)

```
SELECT
    time_bucket('1 hour', valid_time_from) as date_time,
    count(*) AS count
FROM requests
GROUP BY date_time
ORDER BY date_time
```

In den Ergebnissen in Abb. 5.4 sieht man, dass zwischen TimescaleDB und der MySQL bei den beiden kleineren Instanzen kein eindeutiger Unterschied zu erkennen ist, bei der größten Instanz antwortet die TimescaleDB um 0,72 s schneller.

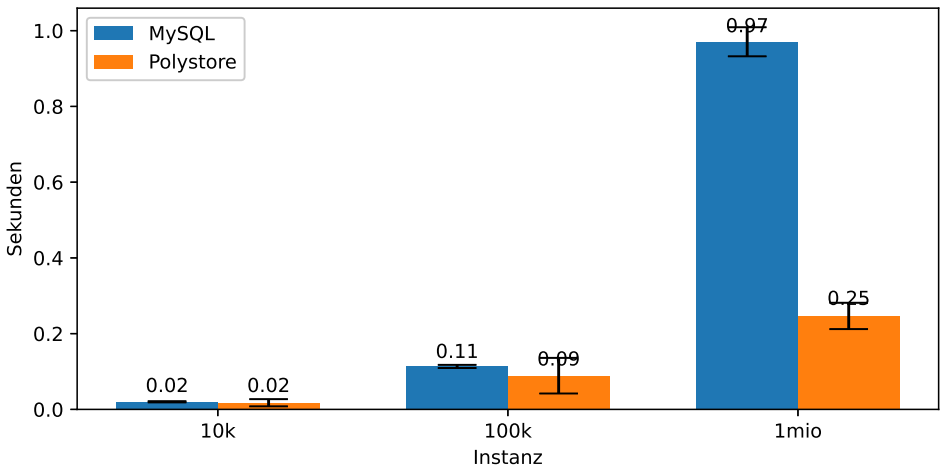

Abbildung 5.4: Anfragen gruppiert nach Gültigkeitszeit

5.4.2 Selektion anhand eines Zeitintervalls

In dieser Query sollen alle Anfragen, deren Anfragegültigkeit in einem bestimmten Zeitintervall beginnt abgefragt werden und die temporalen Eigenschaften zurückgegeben werden.

Listing 5.6: Selektieren nach Zeitintervall (MySQL)

```
1 SELECT r.*
2 FROM requests as r
3 WHERE
4     valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
          < '2017-01-02 09:00:00'
5 ORDER BY occurred_at ASC
```

Listing 5.7: Selektieren nach Zeitintervall (Polystore)

```
1 SELECT r.*
2 FROM requests as r
3 WHERE
4     valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
          < '2017-01-02 09:00:00'
5 ORDER BY occurred_at ASC
```

Die Query ist über alle Instanzen hinweg schneller in der TimescaleDB als in MySQL, wie in Abb. 5.5 zu sehen.

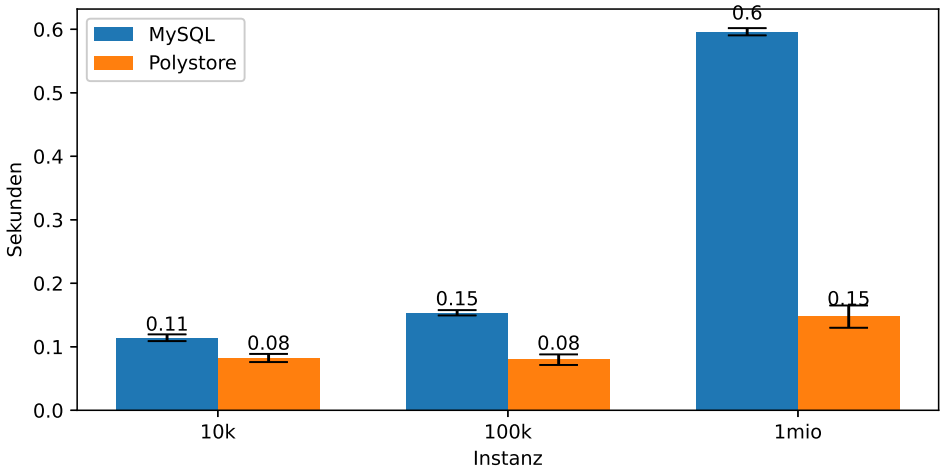

Abbildung 5.5: Anfragen selektiert nach Gültigkeitszeit

5.5 Artefakt-Abfragen

In diesen Queries werden Anfragen anhand von bestimmten Artefakt-Kriterien ausgewählt.

5.5.1 Selektion anhand Referrer

In dieser Query werden Anfragen auf einen spezifischen Referrer durchsucht und ihre temporalen Werte zurückgegeben. Im relationalen Ansatz wird dazu auf der `referrer` Tabelle gejoint (Listing 5.8). Im Polystore werden die entsprechenden Anfragen aus der MongoDB gesucht und anschließend die Ergebnismenge aus der TimescaleDB abgefragt (Listing 5.9).

Listing 5.8: Selektion anhand eines Referrers (MySQL)

```
1 SELECT r.*
2 FROM requests r
3 JOIN referrers re ON re.id = r.referrer_id
4 WHERE re.name = 'geofox.hvv.de'
```

Listing 5.9: Selektion anhand eines Referrers (Polystore)

```
1 q.select_mongodb({'V24': 'geofox.hvv.de'})
2 r = q.results_timescale("""
3     SELECT r.*
4     FROM results rr
5     JOIN requests r ON r.id = rr.id
6 """)
```

Wie in Abb. 5.6 zu sehen ist, ist der relationale Ansatz etwas schneller.

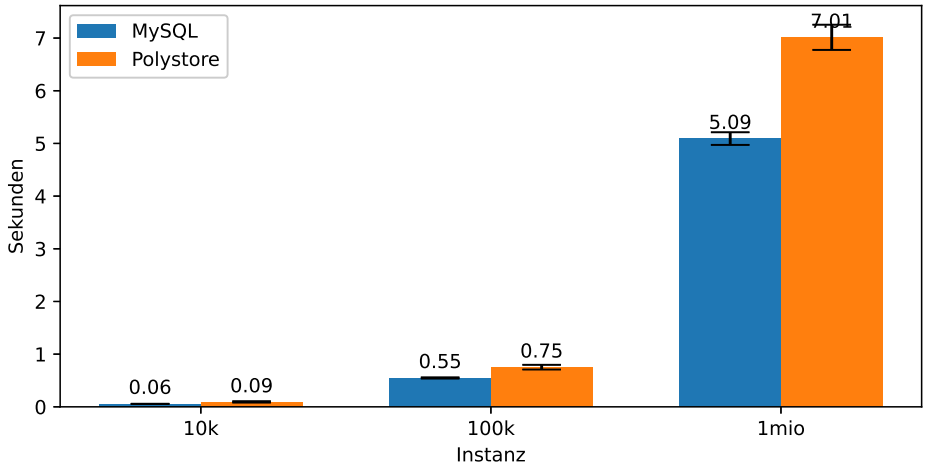

Abbildung 5.6: Selektion anhand Referrer

5.5.2 Selektion anhand Suchparameter

In dieser Query werden Anfragen anhand eines spezifischen Parameterwertes gesucht (`HandicappedChanging=0`) und ihre temporale Eigenschaft selektiert. Im relationalen Ansatz sind dafür mehrere `JOIN`s nötig, wie in Listing 5.10 zu sehen ist, um die `1:n`-Beziehung zwischen Anfrage und Parameterwert aufzulösen. Dieselbe Anfrage wird im Polystore gegen die MongoDB gestellt und anschließend die Ergebnisse aus der TimescaleDB abgefragt, wie in Listing 5.11 zu sehen.

Listing 5.10: Selektion anhand eines Parameterwertes (MySQL)

```sql
SELECT r.*

FROM requests r

JOIN request_parameter_values rpv ON (r.id = rpv.request_id)
JOIN parameters p ON p.id = rpv.parameter_id

WHERE
    p.name = 'HandicappedChanging' AND
    rpv.value = '0'
```

Listing 5.11: Selektion anhand eines Parameterwertes (Polystore)

```
q.select_mongodb({'parameters.HandicappedChanging': '0'})
r = q.results_timescale("""
    SELECT
        r.*
    FROM
        results rr
    JOIN
        requests r ON r.id = rr.id
""")
```

Wie in Abb. 5.7 zu sehen ist, unterscheiden sich MySQL und Polystore auf den beiden kleineren Instanzen nicht nennenswert. Bei der großen Instanz ist der Polystore um 21,5 s schneller.

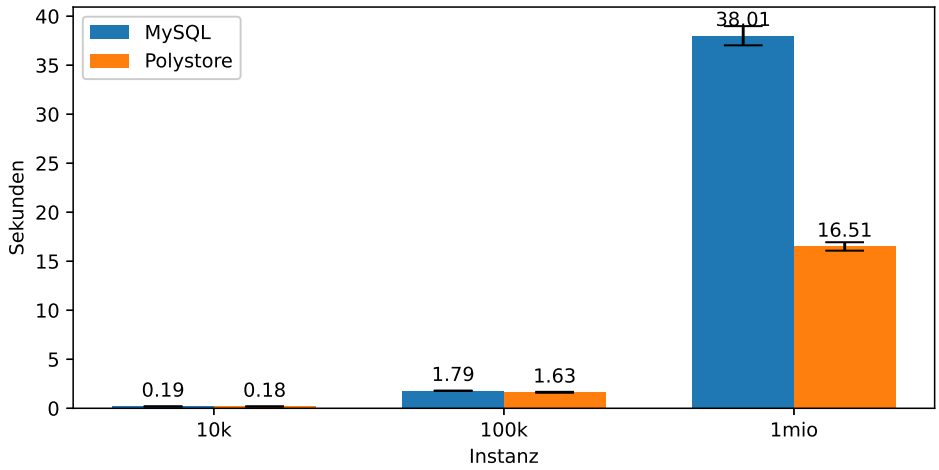

Abbildung 5.7: Selektion anhand Parameter

5.6 Spatio-temporal

Die Query verbindet eine spatiale und eine temporale Bedingung. Es sollen alle Daten selektiert werden, die in einem gegebenen Polygon (jeweils einmal einfach und einmal komplex) starten und deren Gültigkeitszeit in einem gegebenen Intervall liegt.

Listing 5.12: Selektion anhand Zeitspanne und Polygon (MySQL)

```sql
SELECT
    r.*,
    p1.name as origin_name,
    ST_X(p1.geom) as origin_lat,
    ST_Y(p1.geom) as origin_lng,
    p2.name as destination_name,
    ST_X(p2.geom) as destination_lat,
    ST_Y(p2.geom) as destination_lng
FROM requests as r
JOIN points p1 ON p1.id = r.origin_id
JOIN points p2 ON p2.id = r.destination_id

WHERE
    ST_Within(p1.geom, ST_SwapXY(ST_GeomFromText('{wkt}', 4326)))
        and
    valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
       < '2017-01-02 09:00:00'

ORDER BY occurred_at ASC
```

Listing 5.13: Selektion anhand Zeitspanne und Polygon (Polystore)

```
q.select_postgis("""
SELECT r.id
FROM requests r
JOIN points p1 ON p1.id = r.origin_id
WHERE
    ST_Within(p1.geom, ST_GeomFromText('SRID=4326;{wkt}'))
""".format(wkt=horn_simple))

q.select_timescale("""
SELECT
    r.id
FROM requests as r
WHERE
    valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
       < '2017-01-02 09:00:00'
""")
r = q.results()
```

Wie bereits zuvor zeigt sich, dass MySQL bei einem einfachen Polygon (Rechteck) schneller Ergebnisse liefert (Abb. 5.8). Dieselbe Anfrage wurde in Abb. 5.9 mit dem WKT des

Bezirks Altona durchgeführt. Bei einem komplexeren Polygon ist der Polystore schneller.

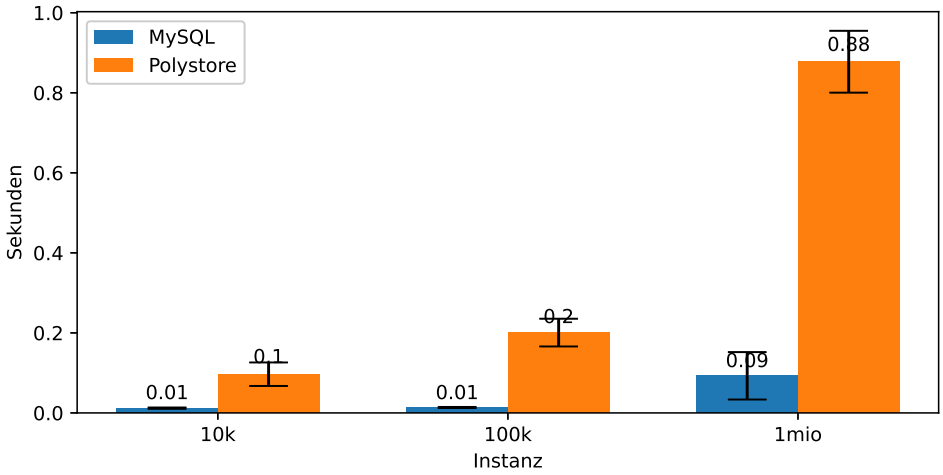

Abbildung 5.8: Selektion anhand Rechteck und Zeitspanne

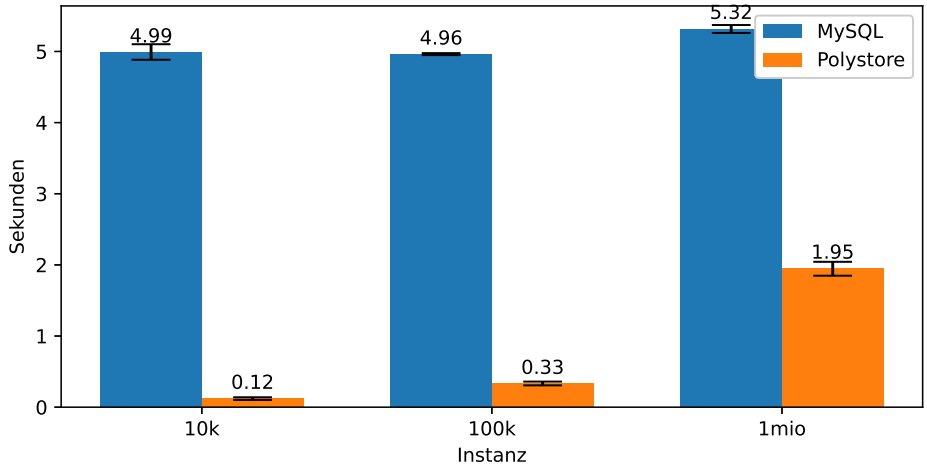

Abbildung 5.9: Selektion anhand Altona und Zeitspanne

5.7 Temporal Artefakt

In der folgenden Query wurde eine temporale Eigenschaft, verbunden mit einem Artefakt, abgefragt. Der gleiche Parameter wie zuvor wird verwendet, jedoch mit einem größeren Zeitfenster, um mehr Ergebnisse zu erhalten.

Listing 5.14: Selektion anhand Zeitspanne und Artefakt (MySQL)

```
SELECT
    r.*,
    p1.name as origin_name,
    ST_X(p1.geom) as origin_lat,
    ST_Y(p1.geom) as origin_lng,
    p2.name as destination_name,
    ST_X(p2.geom) as destination_lat,
    ST_Y(p2.geom) as destination_lng
FROM requests as r
JOIN points p1 ON p1.id = r.origin_id
JOIN points p2 ON p2.id = r.destination_id

JOIN request_parameter_values rpv ON (r.id = rpv.request_id)
JOIN parameters p ON p.id = rpv.parameter_id

WHERE
    (valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
        < '2017-01-02 19:00:00') and
    (p.name = 'HandicappedChanging' AND rpv.value = '0')
```

Listing 5.15: Selektion anhand Zeitspanne und Artefakt (Polystore)

```
q.select_mongodb({'parameters.HandicappedChanging': '0'})

q.select_timescale("""
SELECT
    r.id
FROM requests as r
WHERE
    valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
        < '2017-01-02 19:00:00'
""")

r = q.results(lat_lng=True)
```

Wie in Abb. 5.10 zu sehen, ist bei den beiden kleineren Instanzen die Verarbeitung von MySQL schneller, in der größten Instanz wird die Verarbeitung von MySQL deutlich langsamer als der Polystore.

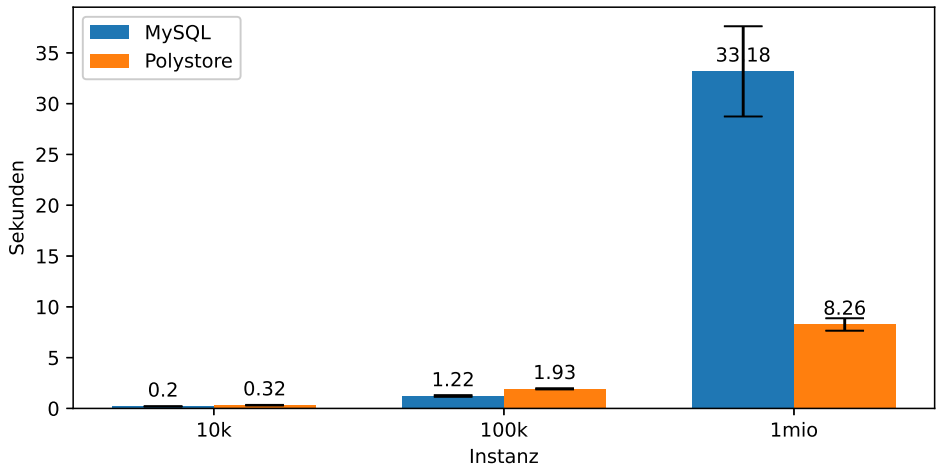

Abbildung 5.10: Selektion anhand Zeitspanne und Artefakt

5.8 Spatio-temporal mit Artefakt

In den folgenden Queries werden alle drei Eigenschaften kombiniert als Suchkriterien verwendet. Im Polystore werden somit alle drei Stores abgefragt.

5.8.1 Polygon, Zeitspanne, Referrer

In der folgenden Query soll auf alle drei Eigenschaften kombiniert zugegriffen werden. Es werden die temporalen und spatialen Informationen selektiert, die einem spatialen, temporalen und Artefakt-Filter genügen. Dafür werden Daten selektiert, die in einem einfachen Rechteck ihren Ursprung haben, deren Gültigkeitszeit in einem bestimmten Intervall beginnt und die einen bestimmten Referrer haben.

Dafür muss im relationalen Ansatz über mehrere Tabellen gejoined werden, wie in Listing 5.16 zu sehen. Im Polystore müssen alle Stores einzeln angefragt werden, wie in Listing 5.17 zu sehen.

Listing 5.16: Selektion anhand Polygon, Zeitspanne und Referrer (MySQL)

```
SELECT
    r.*,
    p1.name as origin_name,
    ST_X(p1.geom) as origin_lat,
    ST_Y(p1.geom) as origin_lng,
    p2.name as destination_name,
    ST_X(p2.geom) as destination_lat,
    ST_Y(p2.geom) as destination_lng

FROM requests as r

JOIN points p1 ON p1.id = r.origin_id
JOIN points p2 ON p2.id = r.destination_id
JOIN referrers re ON re.id = r.referrer_id

WHERE
    ST_WITHIN(p1.geom, ST_SwapXY(ST_GEOMFROMTEXT('{wkt}', 4326))) and
    valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from < '2017-01-02 09:00:00' and
    re.name = 'gtieos'

ORDER BY occurred_at ASC
```

Listing 5.17: Selektion anhand Polygon, Zeitspanne und Referrer (Polystore)

```
q.select_mongodb({'V24': 'gtieos'})

q.select_postgis("""
SELECT r.id

FROM requests r

JOIN points p1 ON p1.id = r.origin_id

WHERE
    ST_WITHIN(p1.geom, ST_GEOMFROMTEXT('SRID=4326;{wkt}'))
""".format(wkt=horn_simple))

q.select_timescale("""
SELECT
    r.id
FROM requests as r
WHERE
    valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
        < '2017-01-02 09:00:00'
""")
```

In Abb. 5.11 ist zu sehen, dass der Polystore über alle drei Instanzen hinweg länger Zeiten für die Abfrage benötigt.

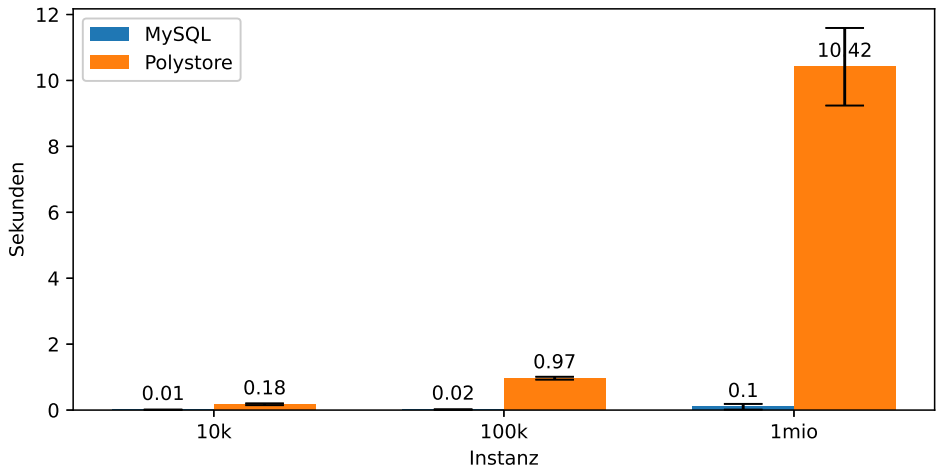

Abbildung 5.11: Selektion anhand Polygon, Zeit und Referrer

5.8.2 Polygon, Zeitspanne, Parameter

In dieser Query wird, wie zuvor, anhand von drei Eigenschaften selektiert, allerdings wird anstelle des Referrers ein Suchparameter verwendet.

Listing 5.18: Selektion anhand Polygon, Zeitspanne und Parameter (MySQL)

```
1  SELECT
2      r.*,
3      p1.name as origin_name,
4      ST_X(p1.geom) as origin_lat,
5      ST_Y(p1.geom) as origin_lng,
6      p2.name as destination_name,
7      ST_X(p2.geom) as destination_lat,
8      ST_Y(p2.geom) as destination_lng
9
10 FROM requests as r
11
12 JOIN points p1 ON p1.id = r.origin_id
13 JOIN points p2 ON p2.id = r.destination_id
14
15 JOIN request_parameter_values rpv ON (r.id = rpv.request_id)
16 JOIN parameters p ON p.id = rpv.parameter_id
17
18 WHERE
19     ST_Within(p1.geom, ST_SwapXY(ST_GeomFromText('{wkt}', 4326)))
           and
20     (valid_time_from >= '2017-01-02 08:00:00' AND valid_time_from
           < '2017-01-02 09:00:00') and
21     (p.name = 'HandicappedChanging' AND rpv.value = '0')
```

Listing 5.19: Selektion anhand Polygon, Zeitspanne und Parameter (Polystore)

```
q.select_mongodb({'parameters.HandicappedChanging': '0'})

q.select_postgis("""
    SELECT r.id
    FROM requests r

    JOIN points p1 ON p1.id = r.origin_id

    WHERE
        ST_Within(p1.geom, ST_GeomFromText('SRID=4326;{wkt}'))
""".format(wkt=horn_simple))

q.select_timescale("""
    SELECT
        r.id
    FROM requests as r
    WHERE
        valid_time_from >= '2017-01-02 08:00:00' AND
           valid_time_from < '2017-01-02 09:00:00'
""")

r = q.results(lat_lng=True)
```

In Abb. 5.12 ist zu sehen, dass trotz der komplexeren Verbindung der Kriterien die Query in MySQL deutlich schneller verarbeitet werden kann. Allerdings liefert diese sehr spezifische Eingrenzung nur eine sehr kleine Ergebnismenge zurück. In Abb. 5.13 wurde die Query mit einem einfachen Polygon in der Größe des Hamburger Stadtgebietes und einer Zeitspanne von einem Tag wiederholt. Trotz der größeren Ergebnismenge ist MySQL in der Verarbeitung schneller.

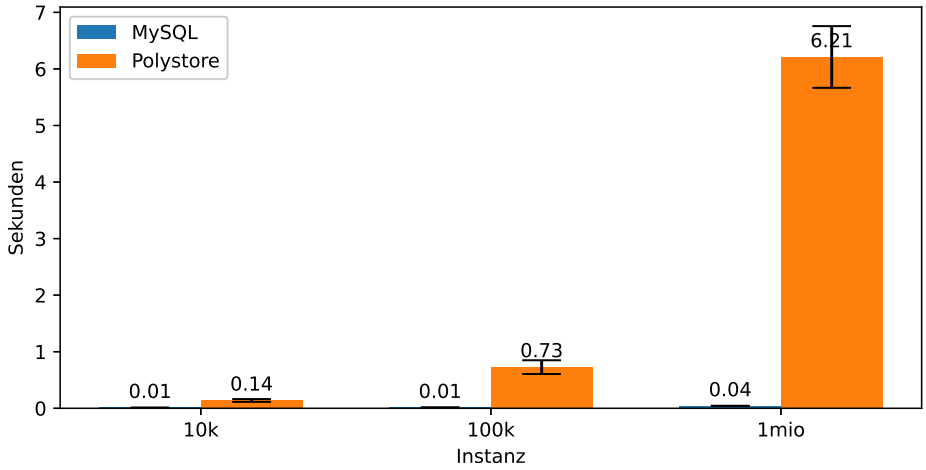

Abbildung 5.12: Selektion anhand Polygon, Zeit und Parameter

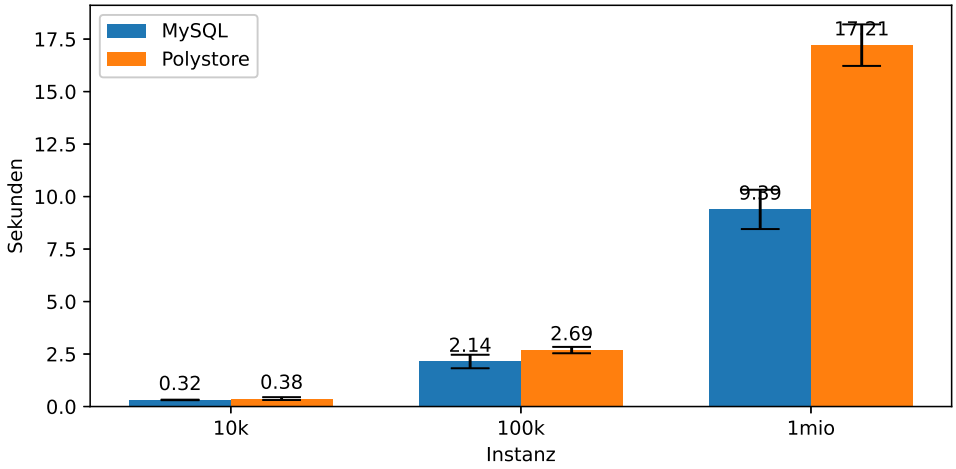

Abbildung 5.13: Selektion anhand Polygon, Zeit und Parameter mit größerem Rechteck und Zeitspanne

6 Diskussion

In Abb. 5.1 wird erkennbar, dass bei der Abfrage eines einzelnen Datensatzes die Performance des Polystores der des MySQL-Ansatzes überlegen ist. Dies ist darauf zurückzuführen, dass hier der Datensatz mit allen Parametern abgefragt wurde. Im MySQL-Ansatz ist hierfür nach dem Abfragen der Attribute eine zusätzliche Abfrage nötig, um die Suchparameter aus der `1:n`-Beziehung abzufragen. Da die Tabelle mit zunehmender Anzahl der Einträge immer größer wird, ist hier zu sehen, dass die Abfrage bei vielen Daten immer länger dauert, da sehr viele Zeilen für den JOIN geprüft werden müssen. Im Polystore ist dies effizienter, da immer mit dem Primärschlüssel der Datensatz in der MongoDB direkt gefunden werden kann und keine aufwendige Suche nötig ist. Man kann also sagen, dass in Szenarien, bei denen die Selektion der Parameter relevant ist, der Polystore-Ansatz bei großen Datenmengen klar überlegen ist.

In den rein spatialen Abfragen mittels der Sucheinschränkung auf ein Polygon ist zu sehen, dass die Suche mit einem einfachen Polygon (Rechteck) im Polystore keinen Performancegewinn geliefert hat (siehe Abb. 5.2). Allerdings hat die PostGIS bei komplexeren Polygonen mit vielen Punkten, wie dem Umriss des Bezirks von Altona, schnellere Ergebnisse geliefert (siehe Abb. 5.3). In MySQL ist die Ausführungszeit in allen drei Instanzen gleichbleibend. Hieraus kann man schließen, dass bei der Anforderung, mit komplexen spatialen Abfragen arbeiten zu müssen, die PostGIS deutliche Vorteile liefert.

Bei rein temporalen Abfragen zeigt sich, dass die auf temporale Daten optimierte TimescaleDB MySQL überlegen ist. Sowohl das Selektieren, als auch das Gruppieren, ist in TimescaleDB deutlich schneller (siehe Abb. 5.4 und 5.5). Betrachtet man die Entwicklung auf den drei Instanzen sieht man, dass beim MySQL-Ansatz die Ausführungszeiten deutlich schneller ansteigen als das bei TimescaleDB der Fall ist. Dies ist auf die optimierte Datenspeicherung der TimescaleDB für temporale Daten im Gegensatz zu MySQL zurückzuführen. Mit den erweiterten Anfragemöglichkeiten der TimescaleDB lässt sich die Anfrage beim Gruppieren auch kompakter formulieren als es bei MySQL der Fall ist.

Bei der Selektion auf Artefakte sind Abfragen auf ein direkt "joinbares" Artefakt wie den Referrer (`1:1` Beziehung zwischen Referrer und Fahrplananfrage) in MySQL etwas schneller als im Polystore. Hier zeigt sich kein nennenswerter Mehrwert des Polystores (Abb. 5.6). Wird allerdings auf komplexere Artefaktstrukturen selektiert, wie die Suchparameter, zeigt sich, dass der Polystore-Ansatz effizienter ist (Abb. 5.7). In der Testinstanz mit einer Million Datensätzen ist der Polystore sogar mehr als doppelt so schnell. Dies ist auf die große Menge der Parameter und die resultierenden JOINs zurückzuführen. Bei komplexeren, arrayartigen Strukturen in den Artefakten stellt sich der Polystore mit der MongoDB als schneller heraus.

Bei den heterogenen Anfragen über mehrere Stores wurde zuerst eine spatio-temporale Anfrage gestellt. Es zeigt sich, wie schon zuvor, dass der MySQL-Ansatz bei einem einfachen Polygon (Rechteck) schneller ist (Abb. 5.8). Die zusätzliche temporale Einschränkung, die als homogene Anfrage im Polystore schneller war, verändert das Gesamtergebnis nicht. Zu erklären ist dies dadurch, dass die einzelnen Abfragen in den Polystores und das Ermitteln der Ergebnismenge in der Anwendungslogik stattfindet. Dieser Teilschritt ist im relationalen Ansatz nicht erforderlich, da dies in der Datenbank selbst geschieht. Wird die Anfrage allerdings wie zuvor mit einem komplexeren Polygon durchgeführt, wiegt die schnellere Anfrage in PostGIS das Gesamtergebnis wieder auf und der Polystore ist trotz des Zwischenschrittes für die Ergebnismenge schneller (Abb. 5.9).
Bei der Temporal-Artefakt-Anfrage zeigt sich, dass der MySQL-Ansatz auf den beiden kleineren Instanzen zwar schneller ist, in der größten Instanz jedoch nicht mehr (Abb. 5.10). Hier zeigt sich, wie schon zuvor in der Artefakt-Anfrage, dass mit zunehmender Anzahl der Daten die `1:n`-Beziehung in der MySQL die Anfragen verlangsamt und die Selektion im Polystore über die MongoDB für Artefakte schneller ist.
In den Anfragen mit allen drei Eigenschaften (spatial, temporal und Artefakte), zeigt sich allerdings, dass der MySQL-Ansatz im Vergleich zum Polystore deutlich schneller ist (Abb. 5.11 und 5.12). Dadurch, dass im Polystore jeder Store zuerst individuell die Ergebnisse anfragen und dann die Schnittmenge ermitteln muss, müssen potenziell viele Daten kopiert werden. Im relationalen MySQL-Ansatz ist dies nicht erforderlich, da dies in der Datenbank durch die JOINs geschieht. Auch die Einschränkung auf Artefakte in der MongoDB, die zuvor schneller war, macht den Polystore nicht besser. Durch die Eingrenzung der Abfragebedingungen kann die MySQL-Datenbank den benötigten JOIN auf die Parameter klein genug halten. Im Polystore hingegen muss wieder die gesamte Teilmenge aus den Stores selektiert werden, um die Ergebnismenge zu bestimmen. Dies macht den Polystore deutlich langsamer. Die kombinierte Anfrage wurde noch einmal

wiederholt mit *größeren* Filtern die mehr Daten selektieren. Allerdings zeigt sich derselbe Effekt wie zuvor. Der MySQL-Ansatz wird zwar langsamer, ist aber trotzdem noch schneller als der Polystore (Abb. 5.13).

Durch das Durchführen der Queries auf den drei verschieden großen Instanzen sieht man bei den heterogenen Anfragen im Polystore, dass mit zunehmender Größe der Polystore schlechter performt als das relationale Modell (Abb. 5.11 und 5.12). Dies ist vor allem darauf zurückzuführen, dass im Polystore die Anfragen an die einzelnen Stores stets größere Ergebnismengen erhalten, die immer vollständig heruntergeladen werden müssen, bevor die endgültige Ergebnismenge aus den drei Mengen ermittelt werden kann.

6.1 Limitierungen

Der hier entwickelte Polystore führt die Anfragen an die jeweiligen Stores sequentiell aus, damit ist zu einem gegebenen Zeitpunkt immer nur eine Datenbank unter Last. Die anderen Datenbanken können erst angesprochen werden, sobald diese Operation abgeschlossen wurde. Dies ist nicht ideal, da das Potenzial der Stores so nicht voll ausgenutzt wird. Eine Optimierung wäre, die jeweiligen Teilanfragen parallel auszuführen. Damit ist eine Anfrage an den Polystore nur noch durch die jeweils langsamste Anfrage limitiert. Zusätzlich werden alle Datenbanken des Polystore auf dem gleichen System ausgeführt, selbst wenn die Anfragen parallelisiert werden, sind sie immer noch dadurch begrenzt, dass die einzelnen Datenbanken auf dem System *um Ressourcen kämpfen*. In diesem Sinne ließe sich der Polystore weiter optimieren, indem jeder Store auf einem individuellen System ausgeführt werden würde.

7 Fazit

Die ursprüngliche Hypothese, dass der Polystore, mit auf die jeweiligen Datentypen spezialisierten Datenbanken effizienter ist, konnte dahingehend bestätigt werden, dass bei homogenen Anfragen auf einer einzelnen Datenbank im Vergleich zum relationalen Modell in MySQL immer bessere Ergebnisse erzielt wurden. Die Datenbanksysteme für die jeweils spezialisierten Datentypen bringen deutliche Vorteile. Allerdings ist bei heterogenen Abfragen über mehrere Datenbanken dieser Effekt nicht zu sehen. Hier bestätigt sich die Hypothese nicht. Durch das Abfragen der verschiedenen Stores müssen viele Daten zwischen den Stores kopiert werden. Dieser Effekt reduziert die Performance des Polystores und wiegt damit auch die Optimierungen der jeweiligen spezifischen Datenbanken gegenüber dem relationalen Ansatz auf. Es bieten sich aber verschiedene Optimierungen an, die die Performance des Polystores deutlich erhöhen könnten, wie in Abschnitt 6.1 dargestellt.

Es ist aber dennoch hervorzuheben, dass durch die semantische Zerlegung der Fahrplanauskunftsanfragen, anhand ihrer Datentypen, dem Anwender ein sehr flexibles System zur Verfügung steht. Mit der Mächtigkeit von drei spezialisierten Anfragesprachen können sehr spezifische Anfragen für Analysen, bzw. die Selektion, von Daten formuliert werden. Hier bietet das Polystore-System deutliche Vorteile. Sofern diese Mächtigkeit allerdings nicht im Fokus der Anwendung steht, bietet sich der *einfachere* relationale Ansatz an, da die Abfragen über drei Eigenschaften hier schneller waren.

7.1 Ausblick

Das Zusammenführen der heterogenen Daten aus dem Polystore ist das kritische Element im Polystore. Das Zusammenführen der Daten wird aber nicht immer zwangsläufig direkt benötigt. Die Abfrage-Logik des Polystores könnte mit *Lazy Loading* Strategien erweitert werden, indem durch die Anfragen zuerst nur kleine Teilmengen identifiziert werden und

erst bei Bedarf der weiteren Attribute, diese aus den anderen Stores abgefragt werden. Wobei hier weiter zu untersuchen wäre, in wieweit dieses Nachladen der Attribute effizient gestaltet werden kann. Das iterative Nachladen von Attributen aus einer Ergebnisliste würde eine Vielzahl einzelner Datenbankanfragen bedeuten. Hier müssten performante Möglichkeiten gefunden werden, dass Nachladen von Attributen zu bündeln.

Literaturverzeichnis

[1] Angles, Renzo ; Gutierrez, Claudio: Survey of Graph Database Models. In: *ACM Comput. Surv.* 40 (2008), Februar, Nr. 1. – URL https://doi.org/10.1145/1322432.1322433. – Place: New York, NY, USA Publisher: Association for Computing Machinery. – ISSN 0360-0300

[2] Armbrust, Michael ; Xin, Reynold S. ; Lian, Cheng ; Huai, Yin ; Liu, Davies ; Bradley, Joseph K. ; Meng, Xiangrui ; Kaftan, Tomer ; Franklin, Michael J. ; Ghodsi, Ali ; Zaharia, Matei: Spark SQL: Relational Data Processing in Spark. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne Victoria Australia : ACM, Mai 2015, S. 1383–1394. – URL https://dl.acm.org/doi/10.1145/2723372.2742797. – ISBN 978-1-4503-2758-9

[3] Bernard, Joey: *Python Recipes Handbook*. Berkeley, CA : Apress, 2016. – URL http://link.springer.com/10.1007/978-1-4842-0241-8. – ISBN 978-1-4842-0241-8

[4] Bernstein, Philip A. ; Chiu, Dah-Ming W.: Using Semi-Joins to Solve Relational Queries. In: *Journal of the ACM* 28 (1981), Januar, Nr. 1, S. 25–40. – URL https://dl.acm.org/doi/10.1145/322234.322238. – ISSN 0004-5411, 1557-735X

[5] Boettiger, Carl: An Introduction to Docker for Reproducible Research. In: *SIGOPS Oper. Syst. Rev.* 49 (2015), Januar, Nr. 1, S. 71–79. – URL https://doi.org/10.1145/2723872.2723882. – Place: New York, NY, USA Publisher: Association for Computing Machinery. – ISSN 0163-5980

[6] Bonaque, R. ; Cao, T. D. ; Cautis, B. ; Goasdoué, F. ; Letelier, J. ; Manolescu, I. ; Mendoza, O. ; Ribeiro, S. ; Tannier, X.: Mixed-instance querying: a lightweight integration architecture for data journalism. In: *Proceedings of the VLDB Endowment* 9 (2016), September, Nr. 13, S. 1513–1516. – URL https://dl.acm.org/doi/10.14778/3007263.3007297. – ISSN 2150-8097

[7] Castano, S. ; Antonellis, V. D.: Global viewing of heterogeneous data sources. In: *IEEE Transactions on Knowledge and Data Engineering* 13 (2001), Nr. 2, S. 277–297

[8] Codd, E. F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Juni, Nr. 6, S. 377–387. – URL https://dl.acm.org/doi/10.1145/362384.362685. – ISSN 0001-0782, 1557-7317

[9] DB-Engines: *DB-Engines Ranking*. 2021. – URL https://db-engines.com/de/ranking. – Zugriffsdatum: 2021-02-01

[10] DeCandia, Giuseppe ; Hastorun, Deniz ; Jampani, Madan ; Kakulapati, Gunavardhan ; Lakshman, Avinash ; Pilchin, Alex ; Sivasubramanian, Swaminathan ; Vosshall, Peter ; Vogels, Werner: Dynamo: Amazon's Highly Available Key-Value Store. In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. New York, NY, USA : Association for Computing Machinery, 2007 (SOSP '07), S. 205–220. – URL https://doi.org/10.1145/1294261.1294281. – event-place: Stevenson, Washington, USA. – ISBN 978-1-59593-591-5

[11] Duggan, Jennie ; Elmore, Aaron J. ; Stonebraker, Michael ; Balazinska, Magda ; Howe, Bill ; Kepner, Jeremy ; Madden, Sam ; Maier, David ; Mattson, Tim ; Zdonik, Stan: The BigDAWG Polystore System. In: *SIGMOD Rec.* 44 (2015), August, Nr. 2, S. 11–16. – URL https://doi.org/10.1145/2814710.2814713. – Place: New York, NY, USA Publisher: Association for Computing Machinery. – ISSN 0163-5808

[12] Elmore, A. ; Duggan, J. ; Stonebraker, M. ; Balazinska, M. ; Cetintemel, U. ; Gadepally, V. ; Heer, J. ; Howe, B. ; Kepner, J. ; Kraska, T. ; Madden, S. ; Maier, D. ; Mattson, T. ; Papadopoulos, S. ; Parkhurst, J. ; Tatbul, N. ; Vartak, M. ; Zdonik, S.: A Demonstration of the BigDAWG Polystore System. In: *Proc. VLDB Endow.* 8 (2015), August, Nr. 12, S. 1908–1911. – URL https://doi.org/10.14778/2824032.2824098. – Publisher: VLDB Endowment. – ISSN 2150-8097

[13] Fasel, Daniel (Hrsg.) ; Meier, Andreas (Hrsg.): *Big Data: Grundlagen, Systeme und Nutzungspotenziale*. Wiesbaden : Springer Fachmedien Wiesbaden, 2016 (Edition HMD). – URL http://link.springer.com/10.1007/978-3-658-11589-0. – ISBN 978-3-658-11588-3 978-3-658-11589-0

[14] Güting, Ralf H.: An Introduction to Spatial Database Systems. In: *The VLDB Journal* 3 (1994), Oktober, Nr. 4, S. 357–399. – Place: Berlin, Heidelberg Publisher: Springer-Verlag. – ISSN 1066-8888

[15] Hamburger Verkehrsverbund: *Geofox Datensatzbeschreibung*. Steindamm 94 20099 Hamburg: . – Von der HVV bereitgestellte Dokumentation

[16] Hamburger Verkehrsverbund: *Der HVV in Zahlen*. 2019. – URL https://www.hvv.de/de/ueber-uns/zahlen-daten-fakten. – Zugriffsdatum: 2020-12-08

[17] HBT: *Hochbahn Hamburg HBT*. – URL https://www.hbt.de/projekt/hochbahn-hamburg/. – Zugriffsdatum: 2020-12-07

[18] Hooijberg, Maarten: *Practical Geodesy*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1997. – URL http://link.springer.com/10.1007/978-3-642-60584-0. – ISBN 978-3-642-64466-5 978-3-642-60584-0

[19] Hull, Richard: Managing Semantic Heterogeneity in Databases: A Theoretical Prospective. In: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. New York, NY, USA : Association for Computing Machinery, 1997 (PODS '97), S. 51–61. – URL https://doi.org/10.1145/263661.263668. – event-place: Tucson, Arizona, USA. – ISBN 0-89791-910-6

[20] Kolev, Boyan ; Bondiombouy, Carlyna ; Valduriez, Patrick ; Jimenez-Peris, Ricardo ; Pau, Raquel ; Pereira, José: The CloudMdsQL Multistore System. In: *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. San Francisco, California, USA : ACM Press, 2016, S. 2113–2116. – URL http://dl.acm.org/citation.cfm?doid=2882903.2899400. – ISBN 978-1-4503-3531-7

[21] Leach, Paul J. ; Mealling, Michael ; Salz, Rich: A Universally Unique IDentifier (UUID) URN Namespace / RFC Editor. RFC Editor, July 2005 (4122). – RFC. – URL http://www.rfc-editor.org/rfc/rfc4122.txt. http://www.rfc-editor.org/rfc/rfc4122.txt. – ISSN 2070-1721

[22] Leavitt, Neal: Will NoSQL Databases Live Up to Their Promise? In: *Computer* 43 (2010), Februar, Nr. 2, S. 12–14. – URL http://ieeexplore.ieee.org/document/5410700/. – ISSN 0018-9162

[23] MARS GROUP: *Multi-Agent Modeling with MARS*. – URL https://mars-group.org/modeling-handbook/. – Version 1.7.2

[24] Mattson, Tim ; Gadepally, Vijay ; She, Zuohao ; Dziedzic, Adam ; Parkhurst, Jeff: Demonstrating the BigDAWG Polystore System for Ocean Metagenomic Analysis. In: *CIDR*, 2017, S. 9

[25] Merv, Adrian: It's going mainstream, and it's your next opportunity. 1 (2011)

[26] Obe, Regina O. ; Hsu, Leo S.: *PostGIS in Action*. 2nd. USA : Manning Publications Co., 2015. – ISBN 978-1-61729-139-5

[27] Rahm, Erhard ; Bernstein, Philip A.: A survey of approaches to automatic schema matching. In: *The VLDB Journal* 10 (2001), Dezember, Nr. 4, S. 334–350. – URL http://link.springer.com/10.1007/s007780100057. – ISSN 10668888

[28] Rahm, Erhard ; Saake, Gunter ; Sattler, Kai-Uwe: *Verteiltes und Paralleles Datenmanagement*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015 (eXamen.press). – URL http://link.springer.com/10.1007/978-3-642-45242-0. – ISBN 978-3-642-45241-3 978-3-642-45242-0

[29] Reinsel, David ; Gantz, John ; Rydning, John: The Digitization of the World from Edge to Core. In: *Framingham: International Data Corporation* (2018), S. 28

[30] Saeed, Mohammed ; Villarroel, Mauricio ; Reisner, Andrew T. ; Clifford, Gari ; Lehman, Li-Wei ; Moody, George ; Heldt, Thomas ; Kyaw, Tin H. ; Moody, Benjamin ; Mark, Roger G.: Multiparameter Intelligent Monitoring in Intensive Care II (MIMIC-II): a public-access intensive care unit database. In: *Critical care medicine* 39 (2011), Nr. 5, S. 952

[31] Snodgrass, Richard T.: Temporal databases. In: *IEEE Computer* 19 (1986), S. 35–42

[32] Stonebraker, Michael: *The Case for Polystores – ACM SIGMOD Blog*. Juli 2015. – URL http://wp.sigmod.org/?p=1629. – Zugriffsdatum: 2020-10-28

[33] Stonebraker, Michael ; Cetintemel, Ugur: "One Size Fits All": An Idea Whose Time Has Come and Gone. In: *Proceedings of the 21st International Conference on Data Engineering*. USA : IEEE Computer Society, 2005 (ICDE '05), S. 2–11. – URL https://doi.org/10.1109/ICDE.2005.1. – ISBN 0-7695-2285-8

[34] Tan, R. ; Chirkova, R. ; Gadepally, V. ; Mattson, T. G.: Enabling query processing across heterogeneous data models: A survey. In: *2017 IEEE International Conference on Big Data (Big Data)*, 2017, S. 3211–3220

[35] Wang, Jingjing ; Baker, Tobin ; Balazinska, Magdalena ; Halperin, Daniel ; Haynes, Brandon ; Howe, Bill ; Hutchison, Dylan ; Jain, Shrainik ; Maas, Ryan ; Mehta, Parmita ; Myers, Brandon ; Ortiz, Jennifer ; Suciu, Dan ; Whitaker, Andrew ; Xu, Shengliang: The Myria Big Data Management and Analytics System and Cloud Service, 2017, S. 11

[36] Widom, Jennifer: Research problems in data warehousing. In: *Proceedings of the fourth international conference on Information and knowledge management - CIKM '95*. Baltimore, Maryland, United States : ACM Press, 1995, S. 25–30. – URL http://portal.acm.org/citation.cfm?doid=221270.221319. – ISBN 978-0-89791-812-1

[37] Wiederhold, G.: Mediators in the architecture of future information systems. In: *Computer* 25 (1992), März, Nr. 3, S. 38–49. – URL http://ieeexplore.ieee.org/document/121508/. – ISSN 0018-9162

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Analyse und Vergleich eines Polystore mit einer relationalen Datenbank anhand spatio-temporaler Daten

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original