Verkehrsdaten in Graphdatenbanken

STUDIENARBEIT (Modul T3_3101)

Studiengang Informatik

Dualen Hochschule Baden-Württemberg Stuttgart Campus Horb

von

Dennis Heine

Juni 2024

Matrikelnummer, Kurs	6193616, HOR-TINF2021
Betreuer/Gutachter	Prof. Dr.-Ing. Olaf Herden

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: „Verkehrsdaten in Graphdatenbanken“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hechingen, den 02.06.24	D. Heine
Ort *Datum*	*Unterschrift*

ABSTRACT

Die vorliegende Arbeit untersucht die Integration und Nutzung von Verkehrsdaten in Graphdatenbanken, mit besonderem Fokus auf das General Transit Feed Specification (GTFS)-Format und dessen Anwendung in der Graphdatenbank Neo4j. Angesichts der zunehmenden Urbanisierung und steigenden Mobilitätsbedürfnisse sind präzise Verkehrsdaten unerlässlich für die Optimierung von Verkehrssystemen. Graphdatenbanken wie Neo4j bieten Vorteile gegenüber relationalen Datenbanken, da sie die effiziente Speicherung und Abfrage komplexer Beziehungen ermöglichen. In der Arbeit werden die theoretischen Grundlagen von Verkehrsdaten und Graphdatenbanken erläutert, einschließlich einer umfassenden Definition und Klassifikation von Verkehrsdaten. Ein besonderer Schwerpunkt liegt auf dem GTFS-Format, dessen Struktur und Anwendungsmöglichkeiten detailliert betrachtet werden. Die gesammelten Verkehrsdaten aus verschiedenen Quellen werden in ein einheitliches Format gebracht und durch spezifische Prüf- und Bereinigungsmaßnahmen aufbereitet.
Die Modellierung und Implementierung der aufbereiteten Daten in Neo4j erfolgt durch die Definition von Knoten, Kanten und Eigenschaften, um die Beziehungen zwischen den Datenpunkten abzubilden. Die Implementierung wird schrittweise mit Python-Skripten durchgeführt. Die Arbeit zeigt, dass Graphdatenbanken wie Neo4j hervorragend für die Speicherung und Verarbeitung von Verkehrsdaten im GTFS-Format geeignet sind. Die Ergebnisse bieten wertvolle Ansätze zur kontinuierlichen Verbesserung der öffentlichen Verkehrsinfrastruktur und eröffnen vielfältige Möglichkeiten für zukünftige Entwicklungen im Bereich der Verkehrsdaten und Graphdatenbanken.

Inhaltsverzeichnis

1. Abbildungsverzeichnis

2. Einleitung

2.1. Motivation

Die Analyse und Optimierung von Verkehrssystemen stellen eine zentrale Herausforderung in den modernen Gesellschaften dar. Mit zunehmender Urbanisierung und steigender Mobilitätsnachfrage wird es immer wichtiger, effiziente und nachhaltige Lösungen zur Verbesserung des öffentlichen Nahverkehrs zu entwickeln. Verkehrsdaten spielen dabei eine entscheidende Rolle, da sie die Grundlage für die Analyse von Fahrgastströmen, die Optimierung von Routen und Fahrplänen sowie die Verbesserung der Servicequalität bilden. In einer Zeit, in der Städte weltweit mit Verkehrsüberlastung und Umweltproblemen kämpfen, bieten präzise Verkehrsdaten die Basis für evidenzbasierte Entscheidungen im Verkehrswesen.[1]

Graphdatenbanken, insbesondere Neo4j, bieten erhebliche Vorteile gegenüber traditionellen relationalen Datenbanksystemen, da sie die effiziente Speicherung und Abfrage von Beziehungen zwischen Haltestellen, Routen und Fahrplänen ermöglichen. Diese Technologie hat sich in zahlreichen Anwendungsbereichen bewährt und bietet leistungsstarke Werkzeuge zur Analyse komplexer Netzwerke.[2]

Das GTFS-Format hat sich als De-facto-Standard für die Veröffentlichung von Verkehrsdaten etabliert. Es ermöglicht Verkehrsbetrieben, ihre Daten in einer strukturierten und maschinenlesbaren Form zu veröffentlichen, was die Integration in verschiedene Anwendungen erleichtert. Diese breite Akzeptanz und Nutzung des GTFS-Formats zeigt das große Potenzial, das in der Kombination von Verkehrsdaten und modernen Datenbanktechnologien liegt. Angesichts dieser Entwicklungen ist es von großer Bedeutung, die Möglichkeiten und Herausforderungen der Integration von GTFS-

[1] (Juan de Dios Ortúzar, 2011)
[2] (Ian Robinson, 2015)

Daten in Graphdatenbanken zu untersuchen und deren Potenziale für die Optimierung des öffentlichen Nah- und Fernverkehrs aufzuzeigen.[3]

2.2. Zielsetzung

Das übergeordnete Ziel dieser Arbeit besteht darin, die Integration und Nutzung von Verkehrsdaten in Graphdatenbanken zu untersuchen und deren Potenziale aufzuzeigen. Zunächst sollen die theoretischen Grundlagen von Verkehrsdaten und Graphdatenbanken detailliert betrachtet werden. Dies umfasst eine Definition und Klassifikation von Verkehrsdaten sowie eine Einführung in die Funktionsweise und Vorteile von Graphdatenbanken. Ein besonderer Schwerpunkt liegt dabei auf dem GTFS-Format, dessen Struktur und Anwendungsmöglichkeiten ausführlich erläutert werden. Durch die Sammlung und Aufbereitung von Verkehrsdaten aus verschiedenen Quellen, einschließlich des öffentlichen Personennahverkehrs (ÖPNV) und der Deutschen Bahn, sollen die Daten in ein einheitliches Format gebracht und ihre Qualität durch spezifische Prüf- und Bereinigungsmaßnahmen sichergestellt werden.

Ein weiterer wesentlicher Teil der Arbeit besteht in der Modellierung und Implementierung der aufbereiteten Verkehrsdaten in der Graphdatenbank Neo4j. Hierbei werden Knoten, Kanten und Eigenschaften definiert, um die komplexen Beziehungen zwischen den verschiedenen Datenpunkten abzubilden. Besonders die Integration von GTFS-Daten und deren Nutzung in der Graphdatenbank stehen im Fokus. Nach der Implementierung erfolgt eine umfassende Analyse der Verkehrsdaten in der Graphdatenbank, wobei verschiedene Abfragen und Analysen durchgeführt werden, um die Potenziale und Vorteile der Nutzung von Graphdatenbanken im Verkehrssektor aufzuzeigen. Abschließend werden die praktischen Relevanzen der Ergebnisse diskutiert und Handlungsempfehlungen für die Praxis abgeleitet. Zudem wird ein Ausblick auf zukünftige Forschungs- und Anwendungsbereiche gegeben, um mögliche

[3] (Byrd, GTFS Startseite, 2024)

Weiterentwicklungen der Methodik sowie neue Forschungsthemen im Bereich der Verkehrsdaten und Graphdatenbanken aufzuzeigen.

2.3. Relevanz der Arbeit

Die Relevanz der vorliegenden Arbeit ergibt sich aus der zentralen Bedeutung von Verkehrsdaten für die Planung und Optimierung von Transportsystemen. In einer zunehmend urbanisierten Welt ist die Effizienz des öffentlichen Nahverkehrs (ÖPNV) entscheidend. Verkehrsdaten ermöglichen es, Verkehrsströme zu verstehen, Engpässe zu identifizieren und gezielte Verbesserungen umzusetzen. Die Integration dieser Daten in Graphdatenbanken wie Neo4j bietet innovative Möglichkeiten, um diese Herausforderungen anzugehen. Graphdatenbanken sind besonders effizient bei der Abbildung und Analyse komplexer Netzwerke, was detaillierte Untersuchungen und Optimierungen von Transitrouten und Fahrplänen ermöglicht.[4]

Durch die Nutzung moderner Datenbanktechnologien und offener Datenformate wie dem General Transit Feed Specification (GTFS)-Format, das von zahlreichen Anwendungen weltweit verwendet wird, können erhebliche Verbesserungen in der Servicequalität und Effizienz des ÖPNV erzielt werden. Diese Arbeit leistet einen wichtigen Beitrag, indem sie methodische Ansätze zur Integration und Analyse von Verkehrsdaten in Graphdatenbanken darstellt und deren praktische Anwendungen im deutschen Verkehrswesen untersucht. Die Implementierung von Verkehrsdaten in Graphdatenbanken kann die Echtzeitüberwachung und -steuerung von Verkehrsflüssen verbessern, was die Pünktlichkeit und Zuverlässigkeit des ÖPNV steigert und somit die Attraktivität des öffentlichen Verkehrs erhöht.[5]

[4] (Ian Robinson, 2015)
[5] (Bibri, 2019)

3. Grundlagen

3.1. Verkehrsdaten

Das folgende Kapitel schafft ein Grundverständnis über verschiedene Typen von Verkehrsdaten und die Quellen, aus denen man diese beziehen kann.

3.1.1. Definition und Typen von Verkehrsdaten

Verkehrsdaten umfassen eine Vielzahl von Informationen, die zur Analyse, Planung und Verwaltung von Verkehrssystemen verwendet werden. Diese Daten können in verschiedene Typen unterteilt werden, die jeweils unterschiedliche Aspekte des Verkehrs abdecken. Als erste Kategorie gibt es die Historischen Verkehrsdaten. Sie sind eine wichtige Kategorie und umfassen die Sammlung und Speicherung von Informationen über Verkehrsmuster über längere Zeiträume. Solche Daten sind wesentlich für die Analyse von Trends, die Bewertung der Effizienz bestehender Verkehrsinfrastrukturen und die Planung zukünftiger Verkehrsprojekte. Historische Daten bieten Einblicke in langfristige Verkehrsverhalten und ermöglichen es, fundierte Entscheidungen zu treffen, um die Verkehrsinfrastruktur zu verbessern und die Mobilität der Bevölkerung zu optimieren.[6]

Statischen Verkehrsdaten stellen eine weitere bedeutende Kategorie dar. Diese Daten umfassen aggregierte Informationen, die aus verschiedenen Quellen gesammelt und analysiert werden, um Verkehrsmodelle zu erstellen, Verkehrsflüsse zu simulieren und die Auswirkungen von Verkehrsmaßnahmen zu bewerten. Sie dienen als quantitative Grundlage für die Beurteilung der Leistung von Verkehrssystemen und sind entscheidend für die Entwicklung von Strategien zur Verbesserung der Verkehrseffizienz.

[6] (Kitchin, 2013)

Beispielsweise können statistische Daten dazu beitragen, Engpässe im Verkehrsnetz zu identifizieren und Lösungen zur Entlastung dieser Bereiche zu entwickeln.[7]

Wie eben erwähnt sind historische und statistische Verkehrsdaten besonders wichtig, da sie eine Grundlage für die Planung und Verwaltung von Verkehrssystemen bieten. Historische Daten helfen, langfristige Trends und Veränderungen im Verkehrsverhalten zu verstehen, während statistische Daten eine detaillierte Analyse und Modellierung von Verkehrsflüssen ermöglichen. Beide Datenarten tragen entscheidend dazu bei, die Effizienz und Sicherheit von Verkehrssystemen zu erhöhen und die Lebensqualität in urbanen Gebieten zu verbessern.[67]

3.1.2. Quellen für Verkehrsdaten (ÖPNV, Bahn, etc.)

Verkehrsdaten im Bereich des öffentlichen Personennahverkehrs (ÖPNV) in Deutschland stammen aus verschiedenen Quellen, die eine umfassende Abdeckung der Verkehrsinformationen gewährleisten. Zu den primären Quellen gehören die Verkehrsbetriebe selbst, die detaillierte Fahrplandaten, Routeninformationen und Haltestellendaten bereitstellen. Diese Daten werden häufig im GTFS-Format veröffentlicht und stehen der Öffentlichkeit zur Verfügung. Die Deutsche Bahn AG beispielsweise bietet umfangreiche Daten zu Zugverbindungen, Fahrplänen und Verspätungen über ihre offizielle Website, mobile Apps und offene Datenportale an. Auch regionale Verkehrsbetriebe wie die Berliner Verkehrsbetriebe (BVG) und der Münchner Verkehrs- und Tarifverbund (MVV) veröffentlichen Fahrplandaten und Echtzeitinformationen über eigene Plattformen und APIs. Diese Daten sind essenziell für die Planung und den Betrieb des ÖPNV und unterstützen sowohl Fahrgäste als auch Entwickler von Mobilitätsanwendungen.[8]

[7] (Athanasios Maimaris, 2017)
[8] (AG, 2024)

Staatliche und kommunale Institutionen spielen ebenfalls eine wichtige Rolle bei der Bereitstellung von ÖPNV-Daten. Das Bundesministerium für Digitales und Verkehr (BMDV) stellt über das MDM (Mobilitäts Daten Marktplatz) eine zentrale Plattform zur Verfügung, über die Verkehrsdaten gesammelt, standardisiert und zugänglich gemacht werden. Der MDM ermöglicht die Integration und Nutzung von Daten aus verschiedenen Quellen, einschließlich Echtzeitdaten von Verkehrsleitsystemen und statischen Daten zu ÖPNV-Infrastrukturen. Kommunale Verkehrsleitsysteme liefern zudem Daten über Verkehrsfluss, Fahrgastzahlen und Verspätungen, die in Echtzeit gesammelt und verarbeitet werden, um die Verkehrssteuerung und -planung zu optimieren. Diese Datenquellen sind unverzichtbar für die Verkehrsüberwachung und -steuerung im städtischen Nahverkehr, da sie eine effiziente Verkehrslenkung und -planung ermöglichen.[9]

[9] (Verkehr, 2024)

3.2. Graphendatenbanken

In diesem Abschnitt soll es um die Definition und Grundlagen von Datenbanken, sowie um die Vorteile gegenüber anderen Datenbanktypen gehen. Zudem wird ein genauerer Blick auf die Graphdatenbank Neo4j geworfen und dessen Merkmale.

3.2.1. Definition und Grundlagen

Graphdatenbanken sind spezialisierte Datenbankmanagementsysteme, die darauf ausgelegt sind, Daten in Form von Knoten (Nodes) und Kanten (Edges) zu speichern und zu verwalten. Diese Struktur ermöglicht es, komplexe Beziehungen zwischen den Daten effizient darzustellen und zu analysieren. Anders als relationale Datenbanken, die Daten in Tabellen organisieren, verwenden Graphdatenbanken eine grafische Struktur, die besonders gut für die Modellierung von vernetzten Daten geeignet ist.[10]

Ein Knoten in einer Graphdatenbank repräsentiert eine Entität, wie z.B. eine Person, einen Ort oder ein Objekt. Kanten definieren die Beziehungen zwischen diesen Entitäten und können zusätzliche Eigenschaften enthalten, die die Natur der Verbindung genauer beschreiben. Diese Modellierungsmethode erlaubt eine flexible und intuitive Darstellung von Daten, die in vielen realen Anwendungsfällen, wie sozialen Netzwerken, Empfehlungssystemen oder Verkehrsnetzen, von Vorteil ist. Die Fähigkeit von Graphdatenbanken, Beziehungen zwischen Datenpunkten direkt darzustellen, führt zu einer erheblich verbesserten Abfrageleistung bei der Analyse komplexer Netzwerke im Vergleich zu traditionellen relationalen Datenbanken.[11]

Graphdatenbanken nutzen spezielle Abfragesprachen, um die Beziehungen zwischen den Knoten effizient zu navigieren. Eine der bekanntesten ist Cypher, die von der Graphdatenbank Neo4j verwendet wird. Cypher ermöglicht es, auf intuitive Weise

[10] (Ian Robinson, 2015)
[11] (Renzo ANgles, 2008)

komplexe Abfragen zu formulieren, die tief in die Struktur des Graphen eindringen können. Diese Leistungsfähigkeit ist besonders nützlich in Anwendungsbereichen, in denen die Beziehungen zwischen Datenpunkten von zentraler Bedeutung sind und schnell analysiert werden müssen. Insgesamt bieten Graphdatenbanken durch ihre spezielle Struktur und Abfragesprachen eine leistungsstarke Alternative zu traditionellen Datenbankmanagementsystemen, insbesondere für Anwendungen, die intensive Vernetzungs- und Beziehungsauswertungen erfordern.[10]

3.2.2. Vorteile und Anwendungen

Graphdatenbanken bieten zahlreiche Vorteile gegenüber traditionellen relationalen Datenbanken, insbesondere wenn es um die Speicherung und Abfrage komplexer und stark vernetzter Daten geht. Einer der bedeutendsten Vorteile ist die hohe Abfrageleistung bei der Verarbeitung von Beziehungen zwischen Datenpunkten. Da Graphdatenbanken speziell darauf ausgelegt sind, Knoten und Kanten direkt zu verknüpfen, ermöglichen sie schnelle und effiziente Abfragen auch in großen und komplexen Datennetzen. Dies steht im Gegensatz zu relationalen Datenbanken, bei denen relationale JOIN-Abfragen oft zu erheblichen Leistungseinbußen führen können.[12]

Ein weiterer Vorteil von Graphdatenbanken ist ihre Flexibilität und Erweiterbarkeit. Sie ermöglichen eine schemalose Datenmodellierung, was bedeutet, dass das Datenmodell dynamisch angepasst werden kann, ohne dass eine aufwendige Neustrukturierung der Datenbank erforderlich ist. Dies ist besonders vorteilhaft in Anwendungsbereichen, in denen sich die Struktur der Daten häufig ändert oder erweitert werden muss, wie z.B. in der Entwicklung von sozialen Netzwerken, Empfehlungssystemen und Content-Management-Systemen. Durch die Möglichkeit, neue Knoten und Kanten sowie deren Eigenschaften flexibel hinzuzufügen, können Graphdatenbanken schnell an neue

[12] (Domingos, 2015)

Anforderungen angepasst werden, was die Entwicklung und Wartung von Anwendungen erheblich vereinfacht.[12]

Graphdatenbanken finden vielfältige Anwendungen in verschiedenen Bereichen. Ein prominentes Anwendungsbeispiel ist das Management von sozialen Netzwerken. In Plattformen wie Facebook oder LinkedIn werden die Beziehungen zwischen Nutzern, Gruppen und Inhalten als Graphen modelliert, was eine effiziente Navigation und Abfrage von Verbindungen ermöglicht. Ein weiteres Anwendungsfeld ist die Empfehlungstechnologie, wie sie von Unternehmen wie Amazon oder Netflix genutzt wird. Hierbei werden Beziehungen zwischen Produkten und Nutzern analysiert, um personalisierte Empfehlungen zu generieren. Darüber hinaus werden Graphdatenbanken in der Verkehrsplanung und -optimierung eingesetzt, um komplexe Netzwerke von Verkehrswegen und -knoten zu modellieren und zu analysieren. Dies ermöglicht eine effiziente Routenplanung, die Erkennung von Engpässen und die Optimierung des Verkehrsflusses.[13]

In der Bioinformatik werden Graphdatenbanken verwendet, um komplexe biologische Netzwerke darzustellen, wie z.B. Protein-Protein-Interaktionen oder Stoffwechselwege. Dies ermöglicht Forschern, komplexe biologische Prozesse zu modellieren und zu analysieren, was zu neuen Erkenntnissen und Fortschritten in der medizinischen Forschung beitragen kann. Insgesamt bieten Graphdatenbanken durch ihre einzigartige Struktur und Leistungsfähigkeit eine leistungsstarke Lösung für eine Vielzahl von Anwendungen, die auf die Analyse und Verarbeitung komplexer Netzwerke angewiesen sind.[13]

[13] (Pramod J. Sadalage, 2012)

3.2.3. Einführung und Merkmale

Neo4j ist eine führende Graphdatenbank, die speziell entwickelt wurde, um komplexe und stark vernetzte Daten effizient zu speichern und zu analysieren. Im Gegensatz zu traditionellen relationalen Datenbanken, die Daten in Tabellen mit starren Schemata organisieren, verwendet Neo4j eine flexible Struktur aus Knoten (Nodes) und Kanten (Edges) wie für Graphdatenbanken typisch. Ein herausragendes Merkmal von Neo4j ist seine native Graphspeicherung und -verarbeitung. Dies bedeutet, dass Daten direkt als Graphen gespeichert und abgerufen werden können, ohne dass eine Umwandlung in eine andere Datenstruktur erforderlich ist. Diese native Unterstützung führt zu einer erheblich besseren Abfrageleistung bei der Navigation durch komplexe Netzwerke. Neo4j ist darauf ausgelegt, Millionen von Knoten und Kanten effizient zu verarbeiten, was es besonders geeignet für Anwendungen macht, die intensive Vernetzungs- und Beziehungsauswertungen erfordern.[14]

Die Abfragesprache Cypher ist ein weiteres zentrales Merkmal von Neo4j. Cypher wurde speziell für die Arbeit mit Graphdaten entwickelt und ermöglicht es, komplexe Abfragen auf einfache und lesbare Weise zu formulieren. Mit Cypher können Entwickler intuitive und effiziente Abfragen schreiben, die tief in die Struktur des Graphen eindringen, um Beziehungen zu analysieren und Muster zu erkennen. Diese Sprache trägt wesentlich zur Benutzerfreundlichkeit und Flexibilität von Neo4j bei, da sie die Interaktion mit den Daten vereinfacht und beschleunigt.[14]

Neo4j bietet auch umfassende Unterstützung für verschiedene Programmiersprachen und Frameworks, was die Integration in bestehende Systeme und Anwendungen erleichtert. Darüber hinaus verfügt Neo4j über eingebaute Transaktionssicherheit und ACID-Konformität (Atomicity, Consistency, Isolation, Durability), die für die Gewährleistung der Datenintegrität und -konsistenz in anspruchsvollen Anwendungen unerlässlich sind. Diese Merkmale machen Neo4j zu einer robusten und zuverlässigen

[14] (Kitchin, 2013)

Plattform für die Entwicklung von Anwendungen, die auf die Analyse und Verarbeitung komplexer Netzwerke angewiesen sind. Ein weiterer Vorteil von Neo4j ist die Möglichkeit der Skalierung und der effizienten Verarbeitung großer Datenmengen. Neo4j unterstützt verteilte Datenbankarchitekturen und ermöglicht es, Daten über mehrere Knoten und Rechenzentren hinweg zu verteilen. Dies stellt sicher, dass auch bei zunehmendem Datenvolumen und steigenden Anforderungen an die Abfrageleistung eine hohe Performance und Verfügbarkeit gewährleistet ist.[14]

Zusammenfassend lässt sich sagen, dass Neo4j durch seine native Graphenspeicherung, die leistungsfähige Abfragesprache Cypher, die umfassende Unterstützung für verschiedene Entwicklungsumgebungen und die robusten Sicherheits- und Skalierungsfunktionen eine ideale Lösung für die Speicherung und Analyse vernetzter Daten darstellt. Diese Merkmale machen Neo4j besonders geeignet für Anwendungen in Bereichen wie sozialen Netzwerken, Empfehlungsdiensten, Verkehrsplanung und Bioinformatik, wo die Analyse von Beziehungen zwischen Datenpunkten von zentraler Bedeutung ist.[14]

3.3. GTFS (General Transit Feed Specification)

Das GTFS-Dateiformat wurde entwickelt, um öffentliche Verkehrsdaten universell zugänglich zu machen. GTFS steht für „General Transit Feed Specification“, was übersetzt „Allgemeine Spezifikation für den Transitverkehr“ bedeutet. Das Dateiformat ist ein offener Standard, um Fahrgästen relevante Informationen über Verkehrssysteme bereitzustellen. Dadurch können öffentliche Verkehrsbetriebe ihre Verkehrsdaten für eine Vielzahl von Softwareanwendungen bereitstellen, was von Tausenden öffentlicher Verkehrsbetriebe weltweit genutzt wird.[15]

GTFS kann in zwei Hauptbestandteile unterteilt werden: GTFS Realtime und GTFS Schedule. GTFS Realtime umfasst aktualisierte Fahrten, Fahrzeugpositionen und Servicewarnungen, während GTFS Schedule Informationen zu Routen, Fahrplänen, Tarifen und geografischen Transitdetails enthält. GTFS Schedule wird in einfachen Textdateien dargestellt, um den Umgang mit den Daten ohne komplexe Software zu ermöglichen. GTFS-Realtime verwendet einen sprach- und plattformneutralen Mechanismus zur Serialisierung strukturierter Daten, der auf Protokollpuffern basiert. Durch die weltweite Verwendung hat GTFS eine große Bedeutung und einen umfassenden Einfluss auf den Umgang mit Routen, Fahrplänen, Haltestellenstandorten und weiteren Informationen.[15]

[15] (Byrd, GTFS Startseite, 2024)

3.3.1. Geschichte

Die Entwicklung des General Transit Feed Specification (GTFS)-Standards begann mit einer Kooperation zwischen TriMet und Google. Trimet ist ein Unternehmen aus Portland, Oregon, welches Bus-, Stadtbahn- und Pendlerbahndienste in der Region anbietet.[16] Diese Zusammenarbeit zielte darauf ab, das Problem der Online-Fahrplanauskunft durch die Nutzung offener Datensätze zu lösen und diese der Öffentlichkeit zugänglich zu machen. TriMet und Google erstellten ein Format für Verkehrsdaten, das einfach zu pflegen und in Google Maps zu importieren war. Ursprünglich wurde dieses Format als “Google Transit Feed Specification” (GTFS) bekannt.[17]

Heutzutage werden GTFS-Daten von einer Vielzahl von Drittanbieter-Softwareanwendungen für unterschiedliche Zwecke verwendet, darunter Reiseplanung, Fahrplanerstellung, mobile Datennutzung, Datenvisualisierung, Barrierefreiheit, Planungstools und Echtzeitinformationssysteme. Im Jahr 2010 wurde der Name in "General Transit Feed Specification" geändert, um die breitere Nutzung über Google-Produkte hinaus zu reflektieren.[17]

GTFS zeichnet sich unter den Datenformaten für den öffentlichen Verkehr dadurch aus, dass es für spezifische praktische Anforderungen bei der Übermittlung von Serviceinformationen an Fahrgäste konzipiert wurde. Es dient nicht als umfassendes Vokabular zur Verwaltung betrieblicher Details. Das Format ist so gestaltet, dass es sowohl für Menschen als auch für Maschinen leicht zu erstellen und zu lesen ist. Selbst Organisationen, die detaillierte interne Daten mit Standards wie NeTEx verwalten, nutzen GTFS, um Daten für Verbraucheranwendungen zugänglicher zu machen.[17]

[16] (TriMet, 2024)
[17] (Byrd, GTFS Hintergrund, 2024)

3.3.2. Definition und Struktur von GTFS

Die GTFS-Daten setzen sich aus mehreren Textdateien zusammen, welche zusammen zu einem Ordner gezipt werden und so dem Nutzer bereitgestellt werden. Dabei hat jede Textdatei einen vordefinierten Namen und einen einheitlichen Inhalt. Um dem GTFS-Standard gerecht zu werden gibt es grundlegende Dateien, welche in jeden GTFS Daten vorhanden sein müssen und optionale Dateien welche je nach Informationsmenge den GTFS Daten hinzugefügt werden können.[18]

Eine der grundlegenden Dateien ist **agency.txt,** die Informationen über die Verkehrsunternehmen enthält, die in dem GTFS-Datensatz vertreten sind. Diese Datei gibt Einblicke in die Organisationen, die den öffentlichen Nahverkehr betreiben, und ermöglicht es, die Dienste entsprechend zuzuordnen und zu kennzeichnen.[18]

Eine weitere wichtige Datei ist **stops.txt**, die die Haltestellen definiert, an denen Fahrgäste ein- und aussteigen können. Diese Datei ist von entscheidender Bedeutung für die Reiseplanung und die Navigation im öffentlichen Nahverkehrssystem.[18]

Die **routes.txt**-Datei beschreibt die verschiedenen Transitstrecken, die von den Verkehrsbetrieben bedient werden. Sie gibt Informationen über die Routennummern, Namen und andere relevante Details, die für die Identifizierung und Unterscheidung der Strecken erforderlich sind.[17]

Die **trips.txt**-Datei enthält Informationen über die einzelnen Fahrten auf den Transitstrecken. Sie zeigt die Abfolge von Haltestellen an, die während einer Fahrt bedient werden, sowie weitere Details wie Fahrzeugtypen und Betriebszeiten.[18]

[18] (Byrd, GTFS Struktur des Dokuments, 2024)

Eine weitere wichtige Datei ist **stop_times.txt,** die die Zeiten angibt, zu denen Fahrzeuge an den Haltestellen ankommen und abfahren. Diese Informationen sind entscheidend für die Fahrplanung und ermöglichen es den Fahrgästen, ihre Reisen genau zu planen.[18]

Zusätzlich zu diesen grundlegenden Dateien gibt es optionale Dateien wie **calendar.txt**, **fare_attributes.txt** und **frequencies.txt**, die weitere spezifische Informationen über den Fahrplan, die Fahrpreise und die Fahrzeiten enthalten können. Durch die Einhaltung des GTFS-Standards können Verkehrsbetriebe sicherstellen, dass ihre Dienste in einer einheitlichen Form präsentiert und von Fahrgästen leicht zugänglich sind.[17]

3.3.3. Anwendung und Relevanz im Verkehrswesen

Das GTFS-Format wird weltweit von über 10.000 Verkehrsbetrieben in über 100 Ländern genutzt und hat sich schnell zu einem Branchenstandard entwickelt. Einige Verkehrsbetriebe erstellen und pflegen diese Daten selbst, während andere Dienstleister dafür beauftragen. Der einfache, textbasierte und offene Standard ermöglicht es vielen Verkehrstechnologieanbietern, GTFS-Daten zu verarbeiten und zu integrieren. Ein fundiertes Verständnis von GTFS erlaubt es Verkehrsunternehmen, bessere Entscheidungen bezüglich der Datenpflege und -verteilung zu treffen, was die Servicequalität maßgeblich beeinflussen kann.[19]

Da GTFS ein offener Standard ist, können Verkehrsbetriebe Informationen mit zahlreichen unterstützenden Tools bereitstellen. Dies umfasst einfache Textbearbeitungsprogramme bis hin zu komplexeren Tabellenkalkulationsprogrammen. GTFS-Daten werden in Form von Textdateien erstellt, die einen Dienst beschreiben und online unter einem öffentlich zugänglichen Permalink gehostet werden. Diese Daten können dann von verschiedenen Anwendungen wie Google Maps, Apple Maps, Transit App oder Open Trip Planner genutzt werden. Fahrgäste können somit zwischen

[19] (Byrd, GTFS Warum GTFS verwenden?, 2024)

verschiedenen Reiseplanungs-Apps wählen, je nach ihren individuellen Bedürfnissen und Präferenzen.[19]

Zusätzlich zu den grundlegenden Fahrplaninformationen bietet GTFS optionale Funktionen wie Fahrpreisanzeigen, Flex-Dienste für bedarfsgesteuerte Transportoptionen und Pathways zur Darstellung von Barrierefreiheitsinformationen. GTFS-Realtime ergänzt diese Funktionen durch Echtzeit-Updates zu Fahrzeugpositionen. Dies ermöglicht eine umfassende und aktuelle Informationsbereitstellung, die sowohl für Fahrgäste als auch für Verkehrsplaner und Entscheidungsträger wertvoll ist.[19]

Aktuelle und qualitativ hochwertige GTFS-Daten sind essenziell für die Planung und Analyse im öffentlichen Nahverkehr. Ab 2023 verlangt die US-amerikanische Verkehrsbehörde Federal Transit Administration von den Verkehrsbetrieben, dass sie GTFS-Daten zusammen mit ihrem jährlichen Bericht über die National Transit Database einreichen, was die Bedeutung und Relevanz von GTFS weiter unterstreicht.[19]

4. Datenbeschaffung und -aufbereitung

4.1. Beschaffung der Verkehrsdaten

Im ersten Schritt sollen öffentlich zugängliche Verkehrsdaten für die Arbeit gesucht und strukturiert abgespeichert werden.

4.1.1. Datenquellen und Zugänglichkeit

Sucht man öffentlich zugängliche Quellen für Verkehrsdaten des ÖPNV in Deutschland, stößt man schnell auf die Seite „OpenData ÖPNV“. Dieses Portal wird von der Initiative „OpenData ÖPNV“ betrieben und veröffentlicht einen wachsenden Bestand an Mobilitätsdaten. Es stellt Daten von zahlreichen Tarif- und Verkehrsverbünden im jeweiligen Verbundraum zur Verfügung, orientiert an der VDV-Mitteilung 7030, die OpenData und OpenService behandelt.[20]

Das Portal zielt darauf ab, Daten in einem offenen, maschinenlesbaren Format bereitzustellen, um Transparenz, Vertrauen und Verständnis zu schaffen. Es fördert die Generierung neuer Ideen zur Verbesserung der Mobilität und den Austausch zwischen ÖPNV-Nutzern, Entwicklern, Unternehmen und Partnerorganisationen. Betriebsinterne und personenbezogene Daten werden jedoch nicht veröffentlicht.[20]

Dort wird unter anderem auch der DELFI Datensatz bereitgestellt, welcher eine enorme Größe hat und Verkehrsdaten aus ganz Deutschland vereint. DELFI, die Durchgängige Elektronische Fahrgast Information, bietet den technologischen und organisatorischen Rahmen für eine einheitliche Routenberechnung im öffentlichen Personenverkehr in Deutschland. Die Produkte von DELFI, insbesondere der deutschlandweite DELFI-Datensatz und das Zentrale Haltestellenverzeichnis (zHV), bilden die Grundlage für

[20] (OpenData, ÖPNV-Initiative für Deutschland , 2024)

zukunftsfähige Informationsdienste. Diese Dienste sind zuverlässig, transparent und hochaktuell.[21]

DELFI ist ein Kooperationsnetzwerk, das alle Bundesländer, den Bund sowie weitere Partner umfasst. Es schafft die technischen Voraussetzungen für die Auskunft über bundeslandübergreifende Reiseketten. Der DELFI e.V. fungiert als organisatorische Schaltstelle zwischen den Interessen der Kooperationspartner und fördert technologische und fachliche Innovationen.[21]

Auf der Seite von „OpenData ÖPNV“ konnten weitere Datensätze zu lokalen Verkehrsverbunden gefunden werden. Beispielsweise zu Verkehrsverbunden wie RMV (Rhein-Main Verkehrsverbund), VVS (Verkehrs- und Tarifverbund Stuttgart), NWL (Nahverkehrssystem Westfalen-Lippe) und vielen weiteren.

Zusätzlich wurden viele weitere Verkehrsdaten für regionale Verkehrsverbunde auf den jeweiligen Webseiten heruntergeladen. Beispiele sind der VMT (Verkehrsverbund Mittelthüringen), VBB (Verkehrsverbund Berlin-Brandenburg) und NVBW (Nahverkehrsgesellschaft Baden-Württemberg)

Eine weitere große Anlaufstelle ist die Seite GTFS.de. Seit der Verordnung 2017/1926 der EU-Kommission werden seit dem 1 Januar 2020 alle nationalen statischen Fahrplandaten für Deutschland über einen nationalen Zugangspunkt im NeTEx-Format bereitgestellt. Basierend auf diesen Daten bietet die Seite GTFS.de täglich generierte Fahrplandatensätze im GTFS-Format. Diese Datensätze beinhalten das gesamten Fern- und Regionalbahnnetz der Deutschen Bahn, sowie die lokalen und städtischen Verkehrsdaten aller Verkehrsunternehmen.[22]

[21] (OpenData, Delfi, 2024)
[22] (Brosi, 2024)

4.1.2. Managment der Datenquellen

Die Daten wurden aus den unterschiedlichen Datenquellen meist manuell über die entsprechende Website heruntergeladen. Dafür wurde eine Ordner-Struktur gewählt bei dem es ein Hauptordner „Daten“ gibt, in welchem sich dann die jeweiligen Unterordner mit den GTFS-Daten befinden. Zum Management der Daten wurde eine Excel Tabelle angelegt in welcher Unteranderem der Verkehrsbetrieb, Typ, Format, Größe, Website, Zuletzt Aktualisiert, Beschreibung, Name und Anmerkung beinhaltet.

Verkehrsbetrieb	Typ	Format	Größe	Website	Zuletzt Aktuallisiert	Beschreibung	Name	Anmerkungen
Delfi	Soll-Fahrplandaten	GTFS	2,77GB	https://www.opendata-oepnv.de/ht/de/organisation/delfi	30.10.23	DELFI, die Durchgängige ELektronische FahrgastInformation, setzt sowohl den technologischen als auch den c	20231030_fahrplaene_gesamtdeutschland_gtfs	
Delfi	Haltestellen	Csv	134MB	https://www.opendata-oepnv.de/ht/de/organisation/delfi	06.11.23		20231106_zHV_gesamt	
vvs	Liniendaten	Csv	51KB	https://www.opendata-oepnv.de/ht/de/organisation/verk	02.10.23	Für die Menschen in der Region Stuttgart bietet der Verkehrs- und Tarifverbund Stuttgart (VVS) eine integrier	vvs_linien_j23	
vvs	Haltestellen	Csv	755KB	https://www.opendata-oepnv.de/ht/de/organisation/verk	02.10.23		vvs_haltestelle_j23	
rmv	Haltestellen	Csv	3MB	https://www.opendata-oepnv.de/ht/de/organisation/verk	02.09.21	Der Rhein-Main-Verkehrsverbund (RMV) ist einer der größten deutschen Verkehrsverbünde. Er koordiniert ur	RMV_Haltestellen	
NWL	Soll-Fahrplandaten	GTFS	200MB	https://www.opendata-oepnv.de/ht/de/organisation/verk	17.10.23	Ein zuverlässiges Nahverkehrssystem in Westfalen-Lippe sowie über die Region hinaus erhalten und ausbauen	gtfs-nwl-20231017	
nvv	Soll-Fahrplandaten	GTFS	109MB	https://www.opendata-oepnv.de/ht/de/organisation/verk	21.01.21	Der NVV ist die gemeinsame Dachorganisation des Bus-, Tram und Eisenbahnverkehres der fünf nordhessische	2021-01-nvv-rt-tram-bus	
nvv	Haltestellen	Csv	2MB	https://www.opendata-oepnv.de/ht/de/organisation/verk	15.05.19		190515-nvv-haltepositionen	
VRS	Soll-Fahrplandaten	GTFS	361MB	https://offenedaten-koeln.de/dataset/vrs-verkehrsdaten-g	03.04.23	Die VRS GmbH stellt Fahrplandatendaten in diesem Format zur Verfügung.	GTFS_VRS_mit_SPNV	
GTFS	Soll-Fahrplandaten	GTFS	1,19GB	https://gtfs.de/de/feeds/	unbekannt	Es handelt sich mit über 20.000 Linien, mehr als 500.000 Haltepunkten und fast 2 Millionen regelmäßig verkel	GTFS	
VMT	Soll-Fahrplandaten	GTFS	119MB	https://www.vmt-thueringen.de/auskunft/open-data/	unbekannt	Der Verkehrsverbund Mittelthüringen stellt die Fahrplandaten des Verbundgebietes und Gesamt Thüringen ir	VMT_GTFS	
VBB	Haltestellen	GTFS	610MB	https://www.vbb.de/vbb-services/api-open-data/datensae	01.01.21	Der Verkehrsverbund Berlin-Brandenburg (VBB) stellt nicht nur Bus- und Bahn-Fahrplandaten aus Berlin und E	GTFS-2	
NVBW	Soll-Fahrplandaten	GTFS	12,3MB	https://www.nvbw.de/open-data/fahrplandaten/fahrplanc	jeden 1 und 3 Diensta	Nahverkehrsgesellschaft Baden Württemberg	bodo, sind nicht alle....	gibt noch mehr

Abbildung 1: CSV-Datei zum Datenmanagmant der Verkehrsdaten

Um Daraufhin einen Überblick über die Anzahl der Daten in den jeweiligen Datensätzen und der insgesamt Gefundenen Daten zu bekommen, wurde ein Python Skript mit dem Namen Count.py geschrieben. Dieses zählt die Zeile in jeder Datei und gibt die Anzahl der insgesamt gefunden Datenpunkte über alle gleichen Dateien zurück. Die Ergebnisse werden in eine Datei result_count.txt geschrieben, um diese anschaulich festzuhalten.

```
Einzelne Zählungen:
--------------------
agency.txt
--------------------
gtfs-nwl-20231017_agency.txt: 45 Zeilen
GTFS_VRS_mit_SPNV_agency.txt: 27 Zeilen
GTFS-2_agency.txt: 34 Zeilen
GTFS_agency.txt: 420 Zeilen
20231030_fahrplaene_gesamtdeutschland_gtfs_agency.txt: 1107 Zeilen
VMT_GTFS_agency.txt: 37 Zeilen
bodo_agency.txt: 19 Zeilen
2021-01-nvv-rt-tram-bus_agency.txt: 3 Zeilen
--------------------
stop_times.txt
--------------------
gtfs-nwl-20231017_stop_times.txt: 1824734 Zeilen
GTFS_VRS_mit_SPNV_stop_times.txt: 2908699 Zeilen
GTFS-2_stop_times.txt: 5881736 Zeilen
GTFS_stop_times.txt: 30592069 Zeilen
20231030_fahrplaene_gesamtdeutschland_gtfs_stop_times.txt: 34923504 Zeilen
VMT_GTFS_stop_times.txt: 1671908 Zeilen
bodo_stop_times.txt: 133554 Zeilen
2021-01-nvv-rt-tram-bus_stop_times.txt: 679688 Zeilen
--------------------
trips.txt
--------------------
gtfs-nwl-20231017_trips.txt: 79515 Zeilen
GTFS_VRS_mit_SPNV_trips.txt: 130306 Zeilen
GTFS-2_trips.txt: 250017 Zeilen
GTFS_trips.txt: 1486980 Zeilen
20231030_fahrplaene_gesamtdeutschland_gtfs_trips.txt: 1728930 Zeilen
VMT_GTFS_trips.txt: 122176 Zeilen
bodo_trips.txt: 6301 Zeilen
2021-01-nvv-rt-tram-bus_trips.txt: 37058 Zeilen
--------------------
stops.txt
--------------------
gtfs-nwl-20231017_stops.txt: 40859 Zeilen
GTFS_VRS_mit_SPNV_stops.txt: 7010 Zeilen
GTFS-2_stops.txt: 41367 Zeilen
GTFS_stops.txt: 668337 Zeilen
20231030_fahrplaene_gesamtdeutschland_gtfs_stops.txt: 529158 Zeilen
VMT_GTFS_stops.txt: 11789 Zeilen
bodo_stops.txt: 2104 Zeilen
2021-01-nvv-rt-tram-bus_stops.txt: 7616 Zeilen
--------------------
routes.txt
--------------------
gtfs-nwl-20231017_routes.txt: 2152 Zeilen
GTFS_VRS_mit_SPNV_routes.txt: 625 Zeilen
GTFS-2_routes.txt: 1274 Zeilen
GTFS_routes.txt: 24268 Zeilen
20231030_fahrplaene_gesamtdeutschland_gtfs_routes.txt: 27382 Zeilen
VMT_GTFS_routes.txt: 860 Zeilen
bodo_routes.txt: 93 Zeilen
2021-01-nvv-rt-tram-bus_routes.txt: 404 Zeilen

Gesamte Zählungen:
agency.txt: 1692 Zeilen
stop_times.txt: 78615892 Zeilen
trips.txt: 3841283 Zeilen
stops.txt: 1308240 Zeilen
routes.txt: 57058 Zeilen
```

Abbildung 2: Inhalt der result_count.txt

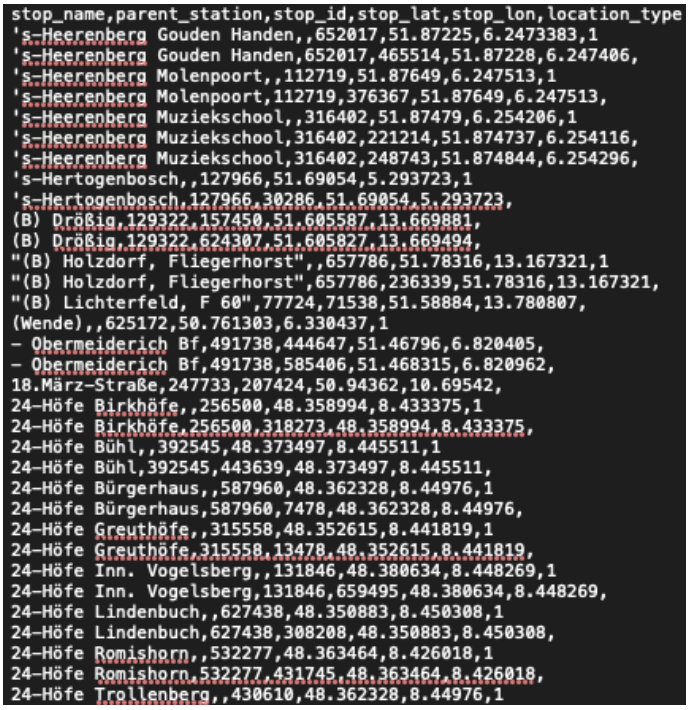
Diese Informationen helfen mit den Daten in einem Effizienten weg umzugehen und schaffen einen groben Überblick über die Daten. Insgesamt beinhalten alle gefunden Daten 1.692 agencys (Unternehmen), 78.615.892 stop_times (Stoppzeiten), 3.841.283 trips (Reisen), 1.308.240 stops (Haltestellen) und 57.058 routes (Routen).

4.2. Datenaufbereitung

Die gefundenen Daten sollen in ein einheitliches Format gebracht werden, um den Datenimport in die Datenbank später zu erleichtern. Zudem soll die Qualität der Daten überprüft werden.

4.2.1. Formatierung und Transformation

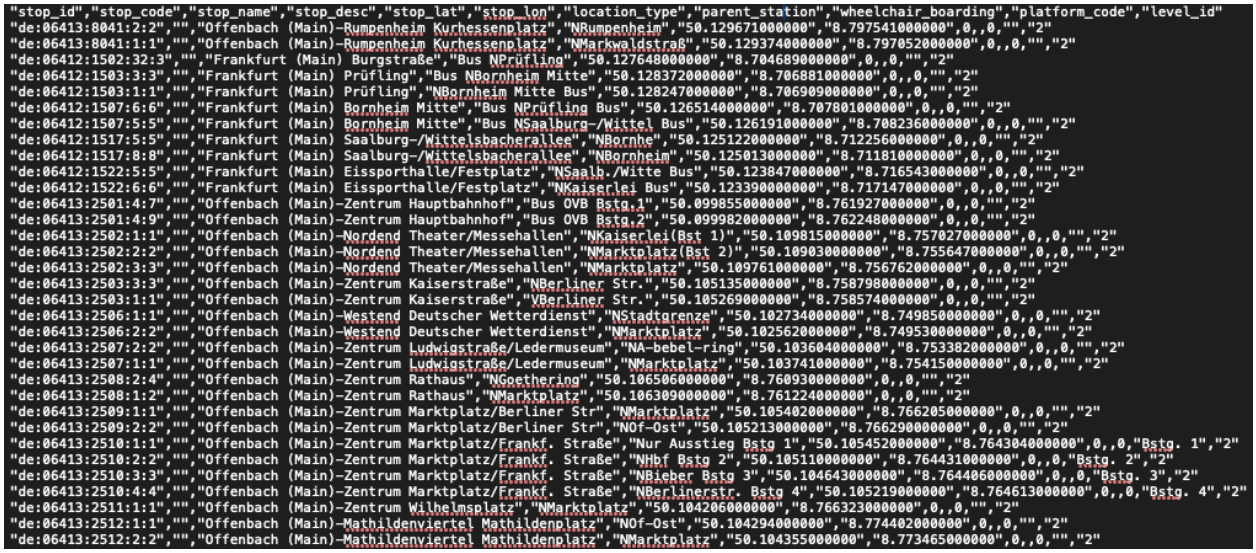
Nach genauerem betrachten der Daten fällt schnell auf das unterschiedliche Dateien aus unterschiedlichen Datensätzen andere Parameter enthalten und diese oft auch in einer anderen Reihenfolge in den jeweiligen Dateien gespeichert sind. Beispielsweise beinhaltet die Datei stops.txt aus dem Datensatz der NVBW (Nahverkehrsgesellschaft Baden-Württemberg) lediglich die Parameter stop_name, parent_station, stop_id, stop_lat, stop_lon und location_type.



```
stop_name,parent_station,stop_id,stop_lat,stop_lon,location_type
's-Heerenberg Gouden Handen,,652017,51.87225,6.2473383,1
's-Heerenberg Gouden Handen,652017,465514,51.87228,6.247406,
's-Heerenberg Molenpoort,,112719,51.87649,6.247513,1
's-Heerenberg Molenpoort,112719,376367,51.87649,6.247513,
's-Heerenberg Muziekschool,,316402,51.87479,6.254206,1
's-Heerenberg Muziekschool,316402,221214,51.874737,6.254116,
's-Heerenberg Muziekschool,316402,248743,51.874844,6.254296,
's-Hertogenbosch,,127966,51.69054,5.293723,1
's-Hertogenbosch,127966,30286,51.69054,5.293723,
(B) Dröβig,129322,157450,51.605587,13.669881,
(B) Dröβig,129322,624307,51.605827,13.669494,
"(B) Holzdorf, Fliegerhorst",,657786,51.78316,13.167321,1
"(B) Holzdorf, Fliegerhorst",657786,236339,51.78316,13.167321,
"(B) Lichterfeld, F 60",77724,71538,51.58884,13.780807,
(Wende),,625172,50.761303,6.330437,1
- Obermeiderich Bf,491738,444647,51.46796,6.820405,
- Obermeiderich Bf,491738,585406,51.468315,6.820962,
18.März-Straße,247733,207424,50.94362,10.69542,
24-Höfe Birkhöfe,,256500,48.358994,8.433375,1
24-Höfe Birkhöfe,256500,318273,48.358994,8.433375,
24-Höfe Bühl,,392545,48.373497,8.445511,1
24-Höfe Bühl,392545,443639,48.373497,8.445511,
24-Höfe Bürgerhaus,,587960,48.362328,8.44976,1
24-Höfe Bürgerhaus,587960,7478,48.362328,8.44976,
24-Höfe Greuthöfe,,315558,48.352615,8.441819,1
24-Höfe Greuthöfe,315558,13478,48.352615,8.441819,
24-Höfe Inn. Vogelsberg,,131846,48.380634,8.448269,1
24-Höfe Inn. Vogelsberg,131846,659495,48.380634,8.448269,
24-Höfe Lindenbuch,,627438,48.350883,8.450308,1
24-Höfe Lindenbuch,627438,308208,48.350883,8.450308,
24-Höfe Romishorn,,532277,48.363464,8.426018,1
24-Höfe Romishorn,532277,431745,48.363464,8.426018,
24-Höfe Trollenberg,,430610,48.362328,8.44976,1
```

Abbildung 3: stops.txt aus NVBW Datensatz

Die Datei stops.txt aus dem DELFI Datensatz hingegen beinhaltet die Parameter stop_id, stop_code, stop_name, stop_desc, stop_lat, stop_lon, location_type, parent_station, wheelchair_boarding, platform_code und level_id.



```
"stop_id","stop_code","stop_name","stop_desc","stop_lat","stop_lon","location_type","parent_station","wheelchair_boarding","platform_code","level_id"
"de:06413:8041:2:2","","Offenbach (Main)-Rumpenheim Kurhessenplatz","NRumpenheim","50.129671000000","8.797541000000",0,,0,"","2"
"de:06413:8041:1:1","","Offenbach (Main)-Rumpenheim Kurhessenplatz","NMarkwaldstraß","50.129374000000","8.797052000000",0,,0,"","2"
"de:06412:1502:32:3","","Frankfurt (Main) Burgstraße","Bus NPrüfling","50.127648000000","8.704689000000",0,,0,"","2"
"de:06412:1503:3:3","","Frankfurt (Main) Prüfling","Bus NBornheim Mitte","50.128372000000","8.706881000000",0,,0,"","2"
"de:06412:1503:1:1","","Frankfurt (Main) Prüfling","NBornheim Mitte Bus","50.128247000000","8.706909000000",0,,0,"","2"
"de:06412:1507:6:6","","Frankfurt (Main) Bornheim Mitte","Bus NPrüfling Bus","50.126514000000","8.707801000000",0,,0,"","2"
"de:06412:1507:5:5","","Frankfurt (Main) Bornheim Mitte","Bus NSaalburg-/Wittel Bus","50.126191000000","8.708236000000",0,,0,"","2"
"de:06412:1517:5:5","","Frankfurt (Main) Saalburg-/Wittelsbacherallee","NBornhe","50.125122000000","8.712256000000",0,,0,"","2"
"de:06412:1517:8:8","","Frankfurt (Main) Saalburg-/Wittelsbacherallee","NBornheim","50.125013000000","8.711810000000",0,,0,"","2"
"de:06412:1522:5:5","","Frankfurt (Main) Eissporthalle/Festplatz","NSaalb./Witte Bus","50.123847000000","8.716543000000",0,,0,"","2"
"de:06412:1522:6:6","","Frankfurt (Main) Eissporthalle/Festplatz","NKaiserlei Bus","50.123390000000","8.717147000000",0,,0,"","2"
"de:06413:2501:4:7","","Offenbach (Main)-Zentrum Hauptbahnhof","Bus OVB Bstg.1","50.099855000000","8.761927000000",0,,0,"","2"
"de:06413:2501:4:9","","Offenbach (Main)-Zentrum Hauptbahnhof","Bus OVB Bstg.2","50.099982000000","8.762248000000",0,,0,"","2"
"de:06413:2502:1:1","","Offenbach (Main)-Nordend Theater/Messehallen","NKaiserlei(Bst 1)","50.109815000000","8.757027000000",0,,0,"","2"
"de:06413:2502:2:2","","Offenbach (Main)-Nordend Theater/Messehallen","NMarktplatz(Bst 2)","50.109030000000","8.755647000000",0,,0,"","2"
"de:06413:2502:3:3","","Offenbach (Main)-Nordend Theater/Messehallen","NMarktplatz","50.109761000000","8.756762000000",0,,0,"","2"
"de:06413:2503:3:3","","Offenbach (Main)-Zentrum Kaiserstraße","NBerliner Str.","50.105135000000","8.758798000000",0,,0,"","2"
"de:06413:2503:1:1","","Offenbach (Main)-Zentrum Kaiserstraße","VBerliner Str.","50.105269000000","8.758574000000",0,,0,"","2"
"de:06413:2506:1:1","","Offenbach (Main)-Westend Deutscher Wetterdienst","NStadtgrenze","50.102734000000","8.749850000000",0,,0,"","2"
"de:06413:2506:2:2","","Offenbach (Main)-Westend Deutscher Wetterdienst","NMarktplatz","50.102562000000","8.749530000000",0,,0,"","2"
"de:06413:2507:2:2","","Offenbach (Main)-Zentrum Ludwigstraße/Ledermuseum","NA-bebel-ring","50.103604000000","8.753382000000",0,,0,"","2"
"de:06413:2507:1:1","","Offenbach (Main)-Zentrum Ludwigstraße/Ledermuseum","NMarktplatz","50.103741000000","8.754150000000",0,,0,"","2"
"de:06413:2508:2:4","","Offenbach (Main)-Zentrum Rathaus","NGoethering","50.106506000000","8.760930000000",0,,0,"","2"
"de:06413:2508:1:2","","Offenbach (Main)-Zentrum Rathaus","NMarktplatz","50.106309000000","8.761224000000",0,,0,"","2"
"de:06413:2509:1:1","","Offenbach (Main)-Zentrum Marktplatz/Berliner Str","NMarktplatz","50.105402000000","8.766205000000",0,,0,"","2"
"de:06413:2509:2:2","","Offenbach (Main)-Zentrum Marktplatz/Berliner Str","NOf-Ost","50.105213000000","8.766290000000",0,,0,"","2"
"de:06413:2510:1:1","","Offenbach (Main)-Zentrum Marktplatz/Frankf. Straße","Nur Ausstieg Bstg 1","50.105452000000","8.764304000000",0,,0,"Bstg. 1","2"
"de:06413:2510:2:2","","Offenbach (Main)-Zentrum Marktplatz/Frankf. Straße","NHbf Bstg 2","50.105110000000","8.764431000000",0,,0,"Bstg. 2","2"
"de:06413:2510:3:3","","Offenbach (Main)-Zentrum Marktplatz/Frankf. Straße","NBieber Bstg 3","50.104643000000","8.764406000000",0,,0,"Bstg. 3","2"
"de:06413:2510:4:4","","Offenbach (Main)-Zentrum Marktplatz/Frankf. Straße","NBerlinerstr. Bstg 4","50.105219000000","8.764613000000",0,,0,"Bstg. 4","2"
"de:06413:2511:1:1","","Offenbach (Main)-Zentrum Wilhelmsplatz","NMarktplatz","50.104206000000","8.766323000000",0,,0,"","2"
"de:06413:2512:1:1","","Offenbach (Main)-Mathildenviertel Mathildenplatz","NOf-Ost","50.104294000000","8.774402000000",0,,0,"","2"
"de:06413:2512:2:2","","Offenbach (Main)-Mathildenviertel Mathildenplatz","NMarktplatz","50.104355000000","8.773465000000",0,,0,"","2"
```

Abbildung 4: stops.txt aus DELFI Datensatz

Wie zu sehen, beinhalten die Daten nicht nur eine Inkonsistenz der vorhandenen Parameter, sondern auch eine Inkonsistenz im Format (der Anordnung) der jeweiligen Parameter. Dies ist bei mehreren Dateien zu beobachten.

Eine solche Inkonsistenz der Daten ist für einen einheitlichen Import in eine Datenbank unakzeptabel. Daher musste sich ein Weg überlegt werden, mit welchem man alle Dateien in einen konsistenten Zustand bringt. Jede Datei sollte die gleichen Parameter in der gleichen Reihenfolge beinhalten.

Um dies zu gewährleisten, wurde das GTFS-Datenformat genauer analysiert und es wurde sich ausschließlich auf die wichtigsten Parameter geeinigt. So wurde für den Import in die Datenbank bei jeder Datei nur eine Hand voll Parameter berücksichtigt. Zudem wurden bei jedem GTFS-Datensatz nur die „grundlegenden Dateien“ (siehe: Definition und Struktur von GTFS) berücksichtigt, da diese garantiert in jedem GTFS-Datensatz vorhanden sein müssen.

Im Einzelnen wurde sich bei folgenden Dateien auf folgende Parameter geeinigt:

- agency.txt: agency_id, agency_name, agency_url, agency_timezone
- routes.txt: route_id, agency_id, route_short_name, route_long_name
- trips.txt: route_id, service_id, trip_id, trip_headsign, direction_id, trip_short_name
- stop_times.txt: trip_id, arrival_time, depature_time, stop_id, stop_sequence
- stops.txt: stop_id, stop_name, stop_lat, stop_lon

In einer späteren Iteration könnte man sich überlegen mehr Parameter der GTFS-Daten zu berücksichtigen. Zudem könnte man auch „optionale Dateien“ (siehe: Definition und Struktur von GTFS) für den Import berücksichtigen.

Um die Dateien alle in das oben aufgezeigte Format mit den jeweiligen Parametern zu bringen, wurde ein Python Skript geschrieben mit dem Namen transfer.py.

Dieses funktioniert folgendermaßen:

Über eine Methode wird ein Input und ein Output Ordner für die Daten angegeben, sowie die Reihenfolge wie die Daten in der neuen Anordnung stehen sollen (file_order_mapping). Daraufhin wird der Output Ordner falls noch nicht vorhanden angelegt und es wird für jede Datei im file_order_mapping die entsprechende Datei im Output Ordner angelegt.

```
def process_directory(input_directory, output_directory, file_order_mapping):
    if not os.path.exists(output_directory):
        os.makedirs(output_directory)

    for filename, desired_order in file_order_mapping.items():
        input_file = os.path.join(input_directory, filename)
        output_file = os.path.join(output_directory, filename)
        reorder_columns(input_file, output_file, desired_order)
```

Abbildung 5: Methode process_directory aus transfer.py

Die Methode reorder_colums() ist dann für die Restrukturierung der Daten in den neuen Output Ordner zuständig. Sie liest die entsprechende Datei aus dem Input Folder ein und kopiert die benötigten Parameter Zeile für Zeile in ein Array namens „fieldnames“. Danach wird das Array „fieldnames“ Zeile für Zeile, in richtiger Reihenfolge, in die neue Datei im Output Folder geschrieben.

```python
def reorder_columns(input_file, output_file, desired_order):
    with open(input_file, 'r', encoding='utf-8-sig') as f:
        reader = csv.DictReader(f)

        # Get all unique fieldnames from the input file
        fieldnames = reader.fieldnames

        # Add missing columns to the desired order
        for field in desired_order:
            if field not in fieldnames:
                fieldnames.append(field)

        # Write the output file with the new column order
        with open(output_file, 'w', newline='', encoding='utf-8') as fw:
            writer = csv.DictWriter(fw, fieldnames=desired_order)
            writer.writeheader()

            for row in reader:
                # Ensure missing keys have an empty string value
                ordered_row = {key: row.get(key, '') for key in desired_order}
                writer.writerow(ordered_row)
```

Abbildung 6: Methode reorder_columns aus transfer.py

Wurden diese beiden Methoden definiert, können sie nacheinander mit der richtigen Angabe des Input und Output Folder und der Angabe für das Ordnen der Dateien im „file_order_mapping“, aufgerufen werden.

```python
if __name__ == "__main__":
    input_directory = 'Transfer'
    output_directory = 'Output'

    file_order_mapping = {
        'stops.txt': ['stop_id', 'stop_name', 'stop_lat', 'stop_lon'],
        'agency.txt': ['agency_id', 'agency_name', 'agency_url', 'agency_timezone'],
        'routes.txt': ['route_id', 'agency_id', 'route_short_name', 'route_long_name'],
        'trips.txt': ['route_id', 'service_id', 'trip_id', 'trip_headsign', 'direction_id', 'trip_short_name'],
        'stop_times.txt': ['trip_id', 'arrival_time', 'departure_time', 'stop_id', 'stop_sequence']
    }

    process_directory(input_directory, output_directory, file_order_mapping)
```

Abbildung 7: Main() Methode aus transfer.py

Nach der Ausführung dieses Skriptes auf die gewünschten Daten, kann sichergestellt werden, dass sie in einem einheitlichen Format vorliegen.

4.2.2. Qualitätsprüfung und Datenbereinigung

Nachdem wir die Daten ins gleiche Format gebracht haben, müssen wir noch dafür sorgen, dass sie konsistent sind. Dafür müssen wir den Aufbau der Dateien genauer verstehen.

Jede Datei verweist auf die jeweilig nächste Datei bzw. auf den nächsten Datensatz mithilfe einer ID. Jede Route hat beispielsweise eine agency_id, womit die routen zu den agencys (Unternehmen) eindeutig zugeordnet werden können. Dieses Prinzip zieht sich durch alle Datensätze. Stop_times hat die Parameter trip_id und stop_id mit der man eine Stop_time einer Haltestelle und einer Reise zuordnen kann. Die Verbindung aller Datensätze ist rechts dargestellt. Wenn wir die Daten später in der Datenbank mit Beziehungen zueinander bringen wollen, können wir das ausschließlich über die jeweiligen IDs erreichen. Dafür muss aber garantiert werden das auch jeder Datensatz in jeder Datei eine ID besitzt.

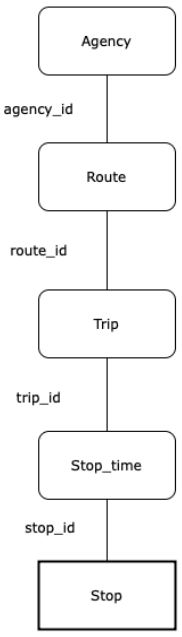


Abbildung 8: Verbindungen über Ids

Um zu garantieren das jeder Datensatz diese Bedingung erfüllt und diese ID auch im jeweiligen Datensatz der verbundenen Komponente vorhanden ist, wurde erneut ein Python Skript namens quality.py geschrieben.

Dieses geht mit einer for-schleife jedes ID-Feld jeder Datei durch. Im ersten Schritt wird überprüft, ob dieses Feld nicht leer ist. Danach wird in der jeweiligen verbundenen Datei nach der ID im entsprechenden Feld gesucht. Falls dieser Wert gefunden wurde, wird „found“ auf True gesetzt und es wird aus der schleife ausgebrochen. Danach wird der nächste Parameter geprüft. Sollte eine ID in der verbundenen Datei nicht gefunden werden, wird diese ausgegeben. Zusätzlich wird vor und nach der Schleife noch die Zeit

gemessen für die Überprüfung und am Ende der Überprüfung ausgegeben. In der folgenden Abbildung ist der Code für die Überprüfung der route_id dargestellt. Bei dem anderen Parameter sieht der Code entsprechend gleich aus.

```
start_time = time.time()
# Überprüfung der Agency ID in Routes
for index, row in df1_routes.iterrows():

    if row.values[1] != None:

        agency_id = row.values[1]

        found = False

        for index, row in df1.iterrows():

            if agency_id == row.values[0]:

                found = True
                break
            else:
                continue

        if found == False:
            print("Agency ID not found: " + agency_id)
    else:
        print(f"Agency ID in Line {index} is None")

end_time = time.time()

print("Finished Agency ID Time elapsed: " + str(end_time - start_time))
```

Abbildung 9: Überprüfung Agency_id in quality.py

Nach dieser Überprüfung kann sichergestellt werden, dass es sich um konsistente Daten im gleichen Format handelt und das jeder Datenknoten eine Verbindung zu einem anderen Datenknoten hat.

5. Modellierung in Neo4j

5.1. Datenmodellierung für Verkehrsdaten

Im Folgenden soll es um das Datenmodell für die Abspeicherung der Daten in der Neo4j Datenbank gehen. Es wird beschrieben wie die GTFS-Daten in Nodes, Edges und Properties gebracht werden und wie diese zu einem Graphen sich zusammenstellen lassen.

5.1.1. Nodes, Edges und Properties in Neo4j

Um unsere GTFS-Datensätzen in eine Neo4j-Graphdatenbank zu laden und die komplexen Beziehungen zwischen den verschiedenen Datenpunkten zu modellieren, wurden verschiedene Knoten (Nodes), Kanten (Edges) und Eigenschaften (Properties) definiert, um eine umfassende und effiziente Darstellung der Verkehrsdaten zu gewährleisten. Die Struktur der Datenbank basiert auf der Notwendigkeit, zentrale Verkehrselemente wie Haltestellen, Routen, Fahrten und Haltezeiten abzubilden und ihre Beziehungen zueinander darzustellen.

Ein Knoten in einer Graphdatenbank repräsentiert eine Entität. Im Fall der GTFS-Daten wurden Knoten wie Stop (Haltestelle), Agency (Unternhemen), Route (Route), Trip (Fahrt) und StopTime (Haltezeit) definiert. Jeder dieser Knoten besitzt spezifische Eigenschaften, die relevante Informationen enthalten. So enthält der Stop-Knoten Eigenschaften wie die eindeutige Kennung der Haltestelle (stop_id), den Namen der Haltestelle (stop_name) und die geografischen Koordinaten (location). Der Agency-Knoten umfasst Eigenschaften wie die eindeutige Kennung des Verkehrsunternehmens (agency_id), den Namen des Unternehmens (agency_name), die URL der Unternehmenswebsite (agency_url) und die Zeitzone des Unternehmens (agency_timezone).

Abbildung 10: Knoten (Nodes)

Der Route-Knoten enthält unter anderem die Eigenschaften route_id, agency_id, route_short_name und route_long_name, während der Trip-Knoten Eigenschaften wie route_id, service_id, trip_id, trip_headsign, direction_id und trip_short_name besitzt. Schließlich enthält der StopTime-Knoten die Eigenschaften trip_id, arrival_time, departure_time, stop_id und stop_sequence, die die Haltezeiten einer Fahrt an den verschiedenen Haltestellen darstellen.

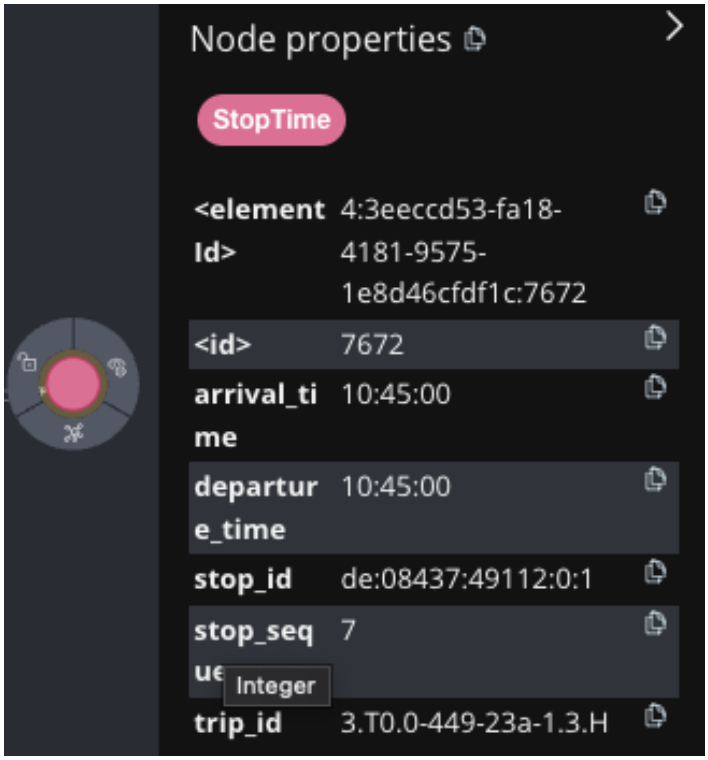


Abbildung 11: Eigenschaften (Properties)

Die Kanten in der Graphdatenbank beschreiben die Beziehungen zwischen den Knoten. In diesem Schema verbindet die Kante OPERATES ein Verkehrsunternehmen (Agency) mit den Routen (Route), die es betreibt. Die Kante USES verbindet eine Route mit den Fahrten (Trip), die auf dieser Route stattfinden. Die Kante BELONGS_TO verbindet eine Haltezeit (StopTime) mit der zugehörigen Fahrt (Trip), während die Kante STOPS_AT eine Haltezeit mit der zugehörigen Haltestelle (Stop) verbindet. Schließlich stellt die Kante NEXT_STOP die Reihenfolge der Haltestellen auf einer Fahrt dar, indem sie aufeinanderfolgende StopTime-Knoten miteinander verbindet.

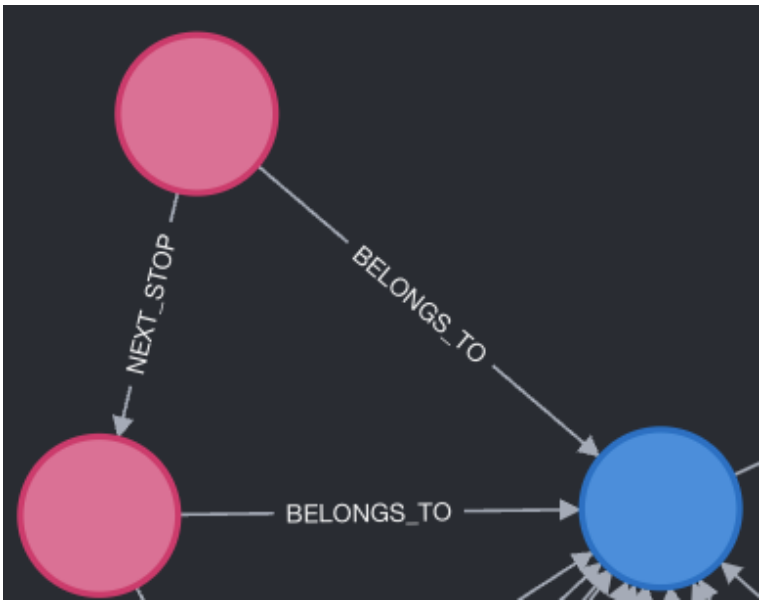


Abbildung 12: Kanten (Edges)

Insgesamt ermöglicht dieses Datenbankschema eine intuitive und leistungsstarke Modellierung und Analyse der Verkehrsdaten, was die Planung und Optimierung des öffentlichen Nah- und Fernverkehrs erheblich erleichtert.

Bei der Datenmodellierung und der Struktur der GTFS-Verkehrsdaten in Neo4j wurde sich an folgendem Artikel orientiert. (Graser, 2023)

5.1.2. Abbildung der Verkehrsdaten als Graph

Durch die im vorherigen Abschnitt beschrieben Darstellung der Verkehrsdaten entsteht ein einheitliches Graphenbild. In diesem kann man sich entweder von der obersten Einheit, der Agency, bis zur untersten Einheit, dem Stop, arbeiten oder andersherum. Mithilfe des Graphen lassen sich übersichtlich alle Daten einer oder mehrere Agenturen darstellen. In folgender Abbildung ein kleiner Ausschnitt eines Unternehmens und der damit verbundenen Daten.

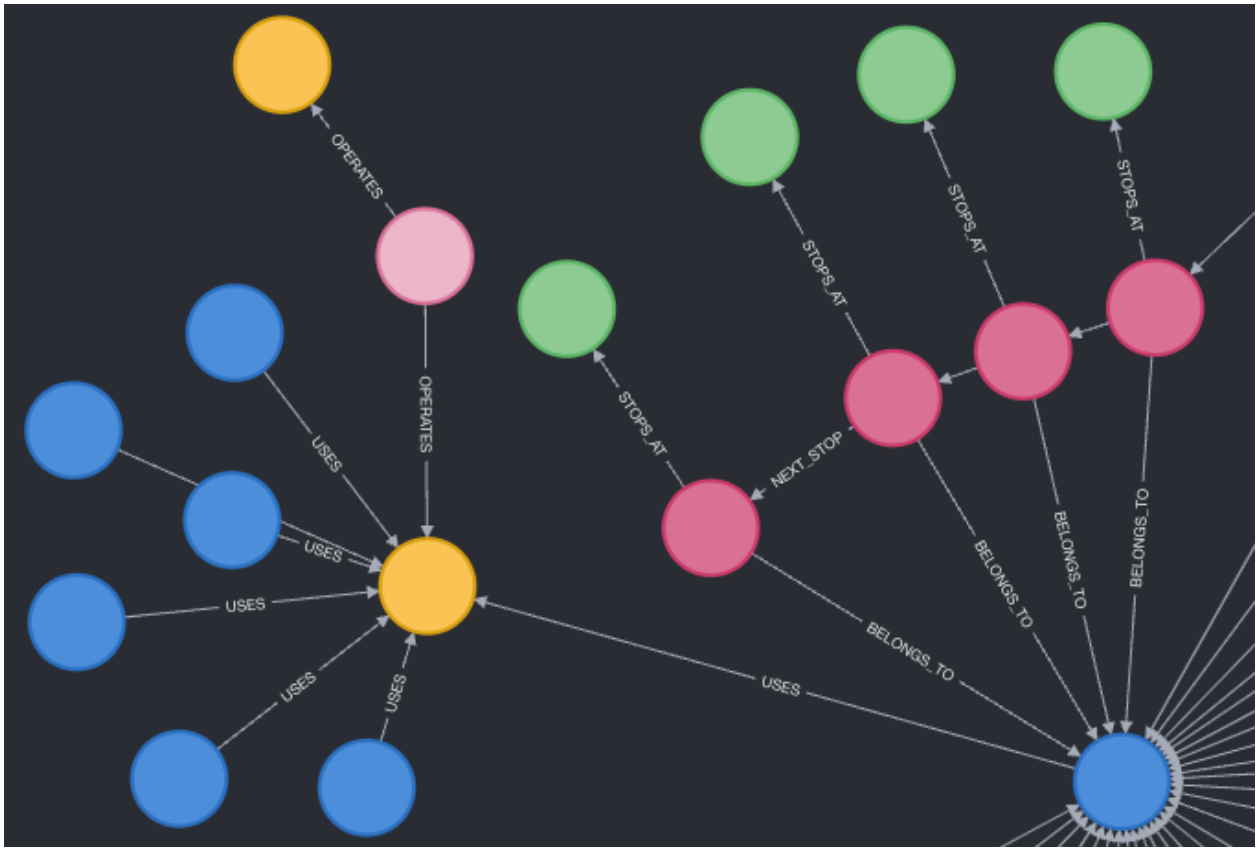


Abbildung 13: Graph Verkehrsdaten

5.2. Implementierung

5.2.1. Aufsetzen einer Datenbank in Neo4j

Nach der Installation von Neo4j und der Installation von Neo4j Desktop konnte mit dem erstellen und der Konfiguration einer Datenbank für das eigene Projekt begonnen werden. Dazu wurde in Neo4j Desktop ein neues Projekt angelegt mit dem Entsprechenden Namen der Studienarbeit. In diesem konnte dann eine DBMS (Datenbankmanagmentsystem) erstellt werden. Dieses wurde über die .config File für den Einsatz dieses Projektes angepasst. Es wurden die Parameter dbms.memory.heap.max_size und dbms.memory.pagecache.size erhöht um den Import und das lesen von einer großen Datenmenge zu beschleunigen.

Zusätzlich wurde das APOC-Plugin installiert. Das APOC (Awesome Procedures on Cypher) Plugin in Neo4j erweitert die Funktionalität der Cypher-Abfragesprache erheblich, indem es eine umfangreiche Sammlung von nützlichen und leistungsstarken Funktionen bereitstellt. Diese umfassen unter anderem die Batch-Verarbeitung großer Datenmengen, die Integration externer Datenquellen, die Manipulation von Graphstrukturen und die Durchführung komplexer Algorithmen. APOC erleichtert somit die Arbeit mit Neo4j, insbesondere bei der Verarbeitung und Analyse großer und komplexer Datensätze, und trägt wesentlich zur Effizienz und Vielseitigkeit von Graphanwendungen bei.[23]

[23] (Neo4j, 2024)

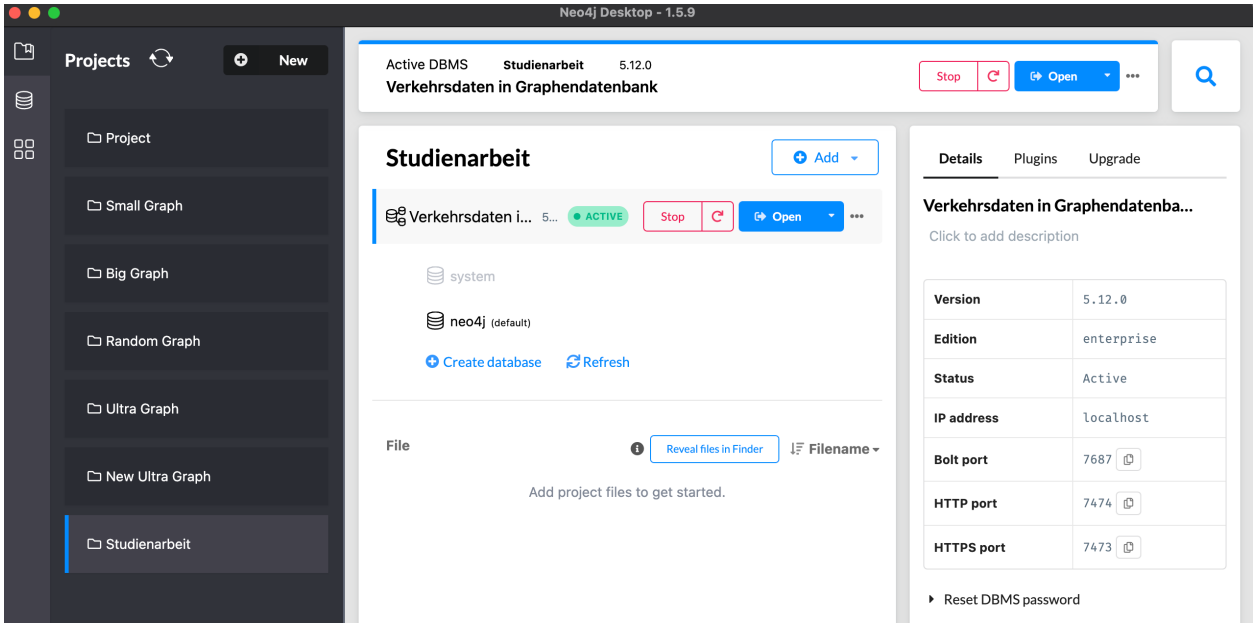


Abbildung 14: Neo4j Desktop

Nach der Installation des Plugins kann die Datenbank das erste mal gestartet werden. Nach dem Starten ist die Datenbank standartmäßig unter localhost auf Port 7687 erreichbar. Um mit der Datenbank leichter agieren zu können, gibt es den Neo4j Browser, welcher direkt mit Neo4j Desktop mitinstalliert wurde. Dieser verbindet sich automatisch mit der aktiven Datenbank und in ihm kann man dann die gewünschten Cypher Befehle ausführen.

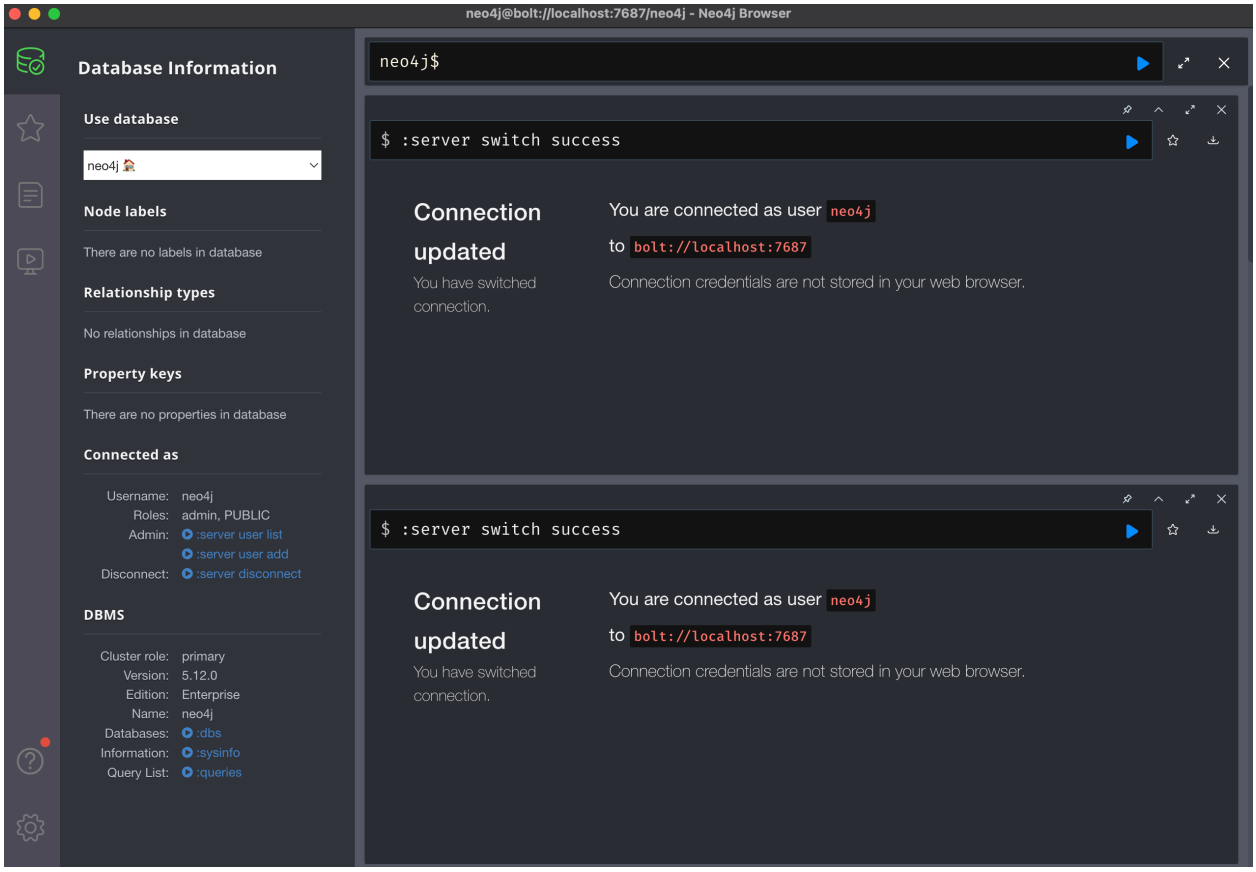


Abbildung 15: Neo4j Browser

Nun ist die Datenbank bereit für den Import der vorbearbeiteten GTFS-Datensätze.

5.2.2. Implementierungsschritte mit Python Skript

Um die GTFS-Daten im finalen Schritt in die Datenbank zu importieren, musste sich im Vorhinein mit der Abfragesprache Cypher intensiver auseinandergesetzt werden. Zu Beginn wurde sich der erste GTFS-Datensatz gepackt, mit den Skripten transfer.py und quality.py in die gewünschte Form gebracht und in das Import Verzeichnis der Datenbank geschoben. Dieses lässt sich leicht über die Anwendung von Neo4j Desktop öffnen. Mit den Dateien, welche sich im Importverzeichnis befinden, kann man dann im folgenden Schritt direkt über die Cypher-Eingabe im Neo4j Browser interagieren. So wurden mit

folgendem Befehl alle agencys in die Datenbank, mit den im vorhinein definierten Parameter (siehe: Formatierung und Transformation), eingefügt:

```
'LOAD CSV WITH HEADERS FROM "file:///agency.txt" AS row RETURN row',
  'CREATE (:Agency {
     agency_id: row.agency_id,
     agency_name: row.agency_name,
     agency_url: row.agency_url,
     agency_timezone: row.agency_timezone
  })'
```

Der Befehl lädt die entsprechende .txt Datei aus dem Importverzeichniss und geht diese dann Zeile für Zeile durch. Für jede Zeile wird ein Agency Knoten erstellt mit den Parametern aus der jeweiligen Zeile als Eigenschaft.

Als nächsten Schritt wurden die Routen aus der Route.txt File mit folgendem Befehl importiert:

```
'LOAD CSV WITH HEADERS FROM "file:///routes.txt" AS row RETURN row',
  'MATCH (a:Agency {agency_id: row.agency_id})
   CREATE (a)-[:OPERATES]->(:Route {
     route_id: row.route_id,
     agency_id: row.agency_id,
     route_short_name: row.route_short_name,
     route_long_name: row.route_long_name
  })'
```

Die Knoten und Eigenschaften der Route wurden äquivalent zum vorherigen Befehl erstellt. Mit dem Befehl MATCH wurde dabei die zur Route passende Agency über die agency.id gesucht und über den Befehl `(a)-[:OPERATES]->(:Route)` wurde eine Kante mit dem Namen OPERATES zu dieser Agency erstellt.

Dieselbe Befehlsabfolge wurde für die Trips gemacht:

```
'LOAD CSV WITH HEADERS FROM "file:///trips.txt" AS row RETURN row',
  'MATCH (r:Route {route_id: row.route_id})
   CREATE (r)<-[:USES]-(:Trip {
     route_id: row.route_id,
     service_id: row.service_id,
     trip_id: row.trip_id,
     trip_headsign: row.trip_headsign,
     direction_id: toInteger(row.direction_id),
     trip_short_name: row.trip_short_name
  })'
```

Die Trips wurden über die route_id mit einer Kante namens USES mit der jeweiligen Route verbunden.

Um die Stop_times einzufügen mussten zuerst die Stops hinzugefügt werden. Dies liegt daran, dass die Stop_times über die trip_id mit den Trips und über die stop_id mit den Stops verbunden sind. Somit haben die Stop_times keinen eindeutigen Parameter, mit dem sie identifiziert werden können und sind von den Verbindungen zu Trips und Stops abhängig.

Die Stops wurden mit den gleichen Befehlen wie die Agencys eingefügt.

```
'LOAD CSV WITH HEADERS FROM "file:///stops.txt" AS row RETURN row',
  'CREATE (:Stop {
     stop_id: row.stop_id,
     stop_name: row.stop_name,
     location: point({
        latitude: toFloat(row.stop_lat),
        longitude: toFloat(row.stop_lon)
     })
  })'
```

Besonders bei den Stops ist, dass die Latitude und Longitude, also die Höhen und Breiten Koordinaten, in eine Pointer Variable gepackt wurden, welche auf den namen „location“

hört. Dies ist für die Darstellung der Stops auf einer Karte empfehlenswert und dadurch kann eine effizientere Speicherung garantiert werden.

Zuletzt konnten dann die Stop_times eingefügt werden. Diese wurden mit folgendem Befehl in die Datenbank geladen:

```
'LOAD CSV WITH HEADERS FROM "file:///stop_times.txt" AS row RETURN row',
  'MATCH (t:Trip {trip_id: row.trip_id}), (s:Stop {stop_id: row.stop_id})
   CREATE (t)<-[:BELONGS_TO]-(:StopTime {
     trip_id: row.trip_id,
     arrival_time: row.arrival_time,
     departure_time: row.departure_time,
     stop_id: row.stop_id,
     stop_sequence: toInteger(row.stop_sequence)
   })-[:STOPS_AT]->(s)'
```

Dieser Befehl erstellt den Knoten StopTime mit seinen Eigenschaften und erstellt zusätzlich 2 Kanten. Die eine Kante ist über die trip_id mit den Trips verbunden und hört auf den Namen BELONGS_TO und die andere Kante verbindet die StopTime mit dem entsprechenden Stop über die stop_id mit dem Namen STOPS_AT.

Abschließend sollen die StopTimes über eine Kante namens NEXT_STOP miteinander verbunden werden, um nachverfolgen zu können, wie das jeweilige Fahrzeug die Stops nacheinander abfährt. Dafür werden sich alle StopTimes eines Trips angeschaut und nach der Reihenfolge des stop_sequence Parameter mit der jeweils nächsten StopTime verbunden. Dies geschieht mit dem nachfolgenden Befehl.

```
MATCH (t:Trip) RETURN t',
  'MATCH (t)<-[:BELONGS_TO]-(st) WITH st ORDER BY st.stop_sequence ASC
   WITH collect(st) AS stops
   UNWIND range(0, size(stops)-2) AS i
   WITH stops[i] AS curr, stops[i+1] AS next
   MERGE (curr)-[:NEXT_STOP]->(next)'
```

Da das Hochladen von so großen Datensätzen eine Menge Zeit benötigt, wurden die Anfragen mit einem Befehl aus dem APOC-Plugin optimiert. Jede Anfrage wurde mit folgendem Aufruf umrundet:

```
CALL apoc.periodic.iterate( BEFEHL , {batchSize: 5000, parallel: true} )
```

Dieser sorgt dafür die Daten in einem Block von 5000 Datensätze in die Datenbank geschrieben wird und das dieses auch parallel bei genügend Rechenleistung ausgeführt werden kann.

Damit diese Befehle nicht manuell hintereinander im Neo4j Browser eingegeben werden müssen, wurde das Hochladen mit einem Python-Skript namens upload_gtfs_in_db.py automatisiert. Dabei wird die neo4j Bibliothek benutzt und mithilfe dieser eine Verbindung zur lokal laufenden Datenbank hergestellt.

```python
from neo4j import GraphDatabase
import time

# Neo4j-Datenbankverbindung herstellen
uri = "bolt://localhost:7687"
username = "neo4j"
password = "password"
driver = GraphDatabase.driver(uri, auth=(username, password))
```

Abbildung 16: Neo4j Datenbankverbindung

Die Library „time“ wurde zusätzlich eingefügt, um die Zeit zu messen wie lange das Upload Skript für das Hochladen der Daten in die Datenbank braucht. Zu Beginn wird dann die Datenbank sicherheitshalber mit folgendem Befehl geleert.

```
MATCH (n)
DETACH DELETE n
```

Daraufhin wurden die eben beschrieben Befehle nach dem folgenden Schema ausgeführt. Beispielsweise wird dies anhand des Agency uploade gezeigt.

```
# Cypher-Abfrage zum Laden der agency
load_agency = """
CALL apoc.periodic.iterate(
  'LOAD CSV WITH HEADERS FROM "file:///agency.txt" AS row RETURN row',
  'CREATE (:Agency {
      agency_id: row.agency_id,
      agency_name: row.agency_name,
      agency_url: row.agency_url,
      agency_timezone: row.agency_timezone
  })',
  {batchSize: 5000, parallel: true}
)
"""

with driver.session() as session:
    session.run(load_agency)

print("Finished agency")
```

Abbildung 17: Ausführung Cypherbefehl in Python

Auf diese Weise konnten auch große GTFS-Datensätze mit mehreren GB(Gigabyte) über eine größere Zeitspanne automatisiert hochgeladen werden.

5.2.3. Einfügen neuer Datensätze

Um jetzt neue Datensätze in die Datenbank einzufügen muss garantiert werden das im einzufügenden Datensatz sich keine Daten befinden, die bereits in der Datenbank sind und sich somit keine doppelten Daten in der Datenbank befinden. Dafür wurde sich eine Strategie überlegt, wie neue Daten in die Datenbank eingefügt werden können ohne doppelte Daten in der Datenbank zu schreiben.

Diese Strategie sieht folgendermaßen aus:

Zu Beginn werden die Agencys des neu einzufügenden Datensatz mit den Agnecys, welche bereits in der Datenbank sind verglichen. Ist eine Agency bisher nicht in der Datenbank vorhanden, kann davon ausgegangen werden, dass alle Routes, Trips und StopTimes dieser Agnecy bisher noch nicht in der Datenbank vorhanden sind. Also wird für jede Agnecy, welche neu eingefügt wird, jede dazugehörige Route und Trip aus dem neu einzufügenden Datensatz gesucht und in die Datenbank eingefügt. Da die StopTimes mit den Stops verbunden sind, müssen erst die Stops genauer betrachtet werden. Bei den Stops muss besonders drauf geachtet werden, ob dieser sich bereits in der Datenbank befindet. Es kann sein, dass mehrere Unternehmen die gleichen Stops anfahren. Also wird im ersten Schritt geprüft, ob der jeweilige Stop sich in der Datenbank befindet und falls dies nicht der Fall ist, wird ein neuer Stop eingefügt. Daraufhin kann dann als letzter Schritt, nach der Validierung der Stops, die StopTime eingefügt werden. Ist ein Stop bereits vorhanden, wird dieser aus dem einzufügenden Datensatz ignoriert und die entsprechend neu einzufügende StopTime wird mit der Stop_id des bereits vorhanden Stops verbunden. Ob ein Stop bereits vorhanden ist, wird an den Latitude und Longitude Daten des Stops festgestellt, welche sich in der Location Variable als Point befinden.

Um diese Strategie umzusetzen, wurde ein weiteres Python Skript entwickelt das auf den Namen „upload_gtfs_in_db“ hört. Für den eben beschriebenen Vorgang des einfügen von bisher nicht vorhandenen Agencys ist nachfolgender Code verantwortlich:

```
if agency_found == False:
    print(f"{df_row.values[1]} not found")
    create_agency(df_row.values[0], df_row.values[1], df_row.values[2], df_row.values[3])
    double_agency = df_row.values[0]

    for index, df_row_routes in df1_routes.iterrows():
        if df_row_routes.values[1] == double_agency:
            # print(f"Routes ID: {df_row_routes.values[0]}")

            create_route(df_row_routes.values[0], df_row_routes.values[1], df_row_routes.values[2], df_row_routes.values[3])
            double_routes = df_row_routes.values[0]

            for index, df_row_trips in df1_trips.iterrows():
                if df_row_trips.values[0] == double_routes:
                    # print(f"Trip ID: {df_row_trips.values[2]}")

                    create_trip(df_row_trips.values[0], df_row_trips.values[1], df_row_trips.values[2], df_row_trips.values[3],
                    double_trip_id = df_row_trips.values[2]

                    for index, df_row_stop_times in df1_stop_times.iterrows():
                        if df_row_stop_times.values[0] == double_trip_id:
```

Abbildung 18: Einfügen der neuen Agency in Datenbank (1)

```
create_trip(df_row_trips.values[0], df_row_trips.values[1], df_row_trips.values[2], df_row_trips.values[3], df_row_trips.values[4], df_row_trips.values[5])
double_trip_id = df_row_trips.values[2]

for index, df_row_stop_times in df1_stop_times.iterrows():
    if df_row_stop_times.values[0] == double_trip_id:
        # print(f"Processing stop times for Stop Time ID: {df_row_stop_times.values[3]} and Trip ID: {df_row_trips.values[2]}")

        # create_stop_time hier noch nicht da erst stops erstellen

        double_stop_id = df_row_stop_times.values[3]

        for index, df_row_stops in df1_stops.iterrows():
            if df_row_stops.values[0] == double_stop_id:

                # print(f"Stop ID with {df_row_stops.values[1]} found and location is {df_row_stops.values[3]}")

                stop_db = search_stop_by_coordinates(df_row_stops.values[2], df_row_stops.values[3])

                temp = stop_db["location"]
                long = temp[0]
                lat = temp[1]

                if lat == df_row_stops.values[2] and long == df_row_stops.values[3]:
                    pass      You, vorgestern • finished merge skript
                else:
                    create_stop(new_id(df_row_stops.values[0]), df_row_stops.values[1], df_row_stops.values[2], df_row_stops.values[3])
                    create_stop_time(new_id(df_row_stop_times.values[0]), df_row_stop_times.values[1], df_row_stop_times.values[2], new_id(df_row_stop_times.values[3]),
                    break
                create_stop_time(new_id(df_row_stop_times.values[0]), df_row_stop_times.values[1], df_row_stop_times.values[2], stop_db["stop_id"], df_row_stop_times.va
```

Abbildung 19: Einfügen der neuen Agency in Datenbank (2)

Die Methoden create_agency, create_route, create_trip, create_stop_time und create_stop funktionieren nach dem Gleichen Prinzip wie beim Upload-Skript aus vorherigem Kapitel (siehe: Implementierungsschritte mit Python Skript).

Jedoch ist bei GTFS-Daten nicht garantiert, dass jedes Unternehmen das GTFS-Daten bereit stellt, global eindeutige IDS verwendet. So kann es durchaus vorkommen das für

zwei unterschiedliche Routen in zwei unterschiedlichen Datensätzen die gleiche ID verwendet wird. Um dieses Problem zu umgehen wurde sich darauf geeinigt für jeden im Nachhinein neu eingefügten Datensatz vor die jeweilige ID das Pre-Suffix „n_“ anzuhängen. Dies übernimmt die folgende Methode:

Abbildung 20: Einfügen neuer ID

Ist die Agency bereits in der Datenbank vorhanden, muss sichergestellt werden, dass falls es eine neue Route, einen neuen Trip, eine neue StopTime oder einen neuen Stop gibt, diese in die Datenbank eingefügt werden, ohne dass diese sich mit bereits vorhandenen Daten in der Datenbank doppeln. Um dies zu garantieren, wurde sich folgende Strategie ausgedacht:

Ist eine Agency bereits vorhanden wird im einzufügenden Datensatz jede dazugehörige Route, jeder Trip und jede StopTime durchgegangen bis zur untersten Einheit, was in diesem Falle der einzelne Stop ist. Sollte dieser bereits in der Datenbank vorhanden sein wird er ignoriert und die Überprüfung findet für den Stop der nächsten StopTime statt. Sollte jedoch ein Stop noch nicht in der Datenbank vorhanden sein wird dieser in die Datenbank eingefügt. Dieser wird ebenfalls wie oben bereits gezeigt mit der Methoe new_id und der damit generierten neuen stop_id eingefügt. Sollte ein neuer Stop eingefügt worden sein, wird daraufhin die zugehörige neue StopTime, der zugehörige neue Trip und die zugehörige neue Route in die Datenbank eingefügt und an die bereits vorhandene Agency gehängt. Da der Einfügeprozess bei dieser Strategie vom Stop ausgehend (vom Ende des Baumes) ist, müssen neue Methoden generiert werden für das Einfügen der jeweiligen Knoten und Kanten. Zuerst wird daraufhin die dazugehörige StopTime eingefügt und mit dem jeweiligen Stop über die Kante STOPS_AT verbunden. Anschließend wird, falls eine neue StopTime eingefügt wurde, der entsprechend neue Trip als Knoten eingefügt und mit den entsprechenden StopTimes über die Kante BELONGS_TO verbunden. Falls ein neuer Trip eingefügt wurde, wird die entsprechende neue Route als Knoten eingefügt und einerseits mit dem dazugehörigen Trip über die

Kante USES verbunden und andererseits mit der entsprechend in der Datenbank bereits vorhandenen Agency über die Kante OPERATES verbunden. Diese neuen Methoden fügen die Daten ebenfalls alle mit der neuen ID ein und werden im nachfolgendem am Beispiel der StopTimes einfügung dargestellt:

```
def create_stop_time_without_trip(trip_id, arrival_time, departure_time, stop_id, stop_sequence):
    trip_id = new_id(trip_id)
    stop_id = new_id(stop_id)
    with driver.session() as session:
        result = session.run("""
            MATCH (s:Stop {stop_id: $stop_id})
            CREATE (st:StopTime {
                trip_id: $trip_id,
                arrival_time: $arrival_time,
                departure_time: $departure_time,
                stop_id: $stop_id,
                stop_sequence: toInteger($stop_sequence)
            })-[:STOPS_AT]->(s)
        """, trip_id=trip_id, arrival_time=arrival_time, departure_time=departure_time, stop_id=stop_id, stop_sequence=stop_sequence)
    global added_stop_times
    added_stop_times += 1
    driver.close()
```

Abbildung 21: Einfügen StopTimes mit Kante zu Stops

Die globale Variable added_stop_times zählt dabei an dieser Stelle, genauso wie bei der Strategie des einfügen einer neuen Agency, wie viele neue StopTimes beim Durchlaufen des Skripts eingefügt wurden. Dies wird später in der Kommandozeile dem Nutzer als Feedback zurückgegeben.

Der Code für die eben beschriebene Strategie des einfügen bei einer bereits vorhandenen Agency sieht folgendermaßen aus:

```
if df_row.values[1] == agency['agency_name']:

    found_doubles += 1
    agency_found = True
    double_agency = agency['agency_id']
    double_agency_id = df_row.values[0]
    # print(f"Agency ID: {double_agency}")

    for index, df_row_routes in df1_routes.iterrows():

        if df_row_routes.value[1] == double_agency_id:

            neu = False
            neu_trip = False
            neu_stop_time = False
            neu_stop = False

            double_routes_id = df_row_routes.values[0]

            for index, df_row_trips in df1_trips.iterrows():

                if df_row_trips.value[0] == double_routes_id:

                    double_trip_id = df_row_trips.values[2]

                    for index, df_row_stop_times in df1_stop_times.iterrows():

                        if df_row_stop_times.values[0] == double_trip_id:

                            double_stop_id = df_row_stop_times.values[3]

                            for index, df_row_stops in df1_stops.iterrows():

                                if df_row_stops.values[0] == double_stop_id:
```

Abbildung 22: Einfügen Daten vorhandener Agencys in Datenbank (1)

```
            for index, df_row_stops in df1_stops.iterrows():

                if df_row_stops.values[0] == double_stop_id:

                    stop_db = search_stop_by_coordinates(df_row_stops.values[2], df_row_stops.values[3])
                    temp = stop_db["location"]
                    long = temp[0]
                    lat = temp[1]

                    if lat == df_row_stops.values[2] and long == df_row_stops.values[3]:
                        pass
                    else:
                        new_stop_id = new_id(df_row_stops.values[0])
                        create_stop(new_stop_id, df_row_stops.values[1], df_row_stops.values[2], df_row_stops.values[3])
                        neu_stop = True
                        break
            if neu_stop == True:
                create_stop_time_without_trip(df_row_stop_times.values[0], df_row_stop_times.values[1], df_row_stop_times.values[2], new_stop_id, df_row_stop_tim
                neu_stop_time = True
                neu = False

        if neu_stop_time == True:
            create_trip_without_route(df_row_trips.values[0], df_row_trips.values[1], df_row_trips.values[2], df_row_trips.values[3], df_row_trips.values[4], df_row_
            neu_trip = True
            neu_stop_time = False
if neu_trip == True:
    create_route_without_agency(df_row_routes.values[0], df_row_routes.values[1], df_row_routes.values[2], df_row_routes.values[3])
```

Abbildung 23: Einfügen Daten vorhandener Agencys in Datenbank (2)

Um den Code des Skripts vollständig zu zeigen, wird in nachfolgender Abbildung dargestellt, wie sich mit der Datenbank verbunden wird, wie die entsprechend neu einzufügenden Daten mithilfe der Bibliothek Pandas eingelesen werden und wie die Daten der neu einzufügenden Agencys, sowie der Agencys die bereits vorhanden sind, mit einer for-Schleife durchgegangen werden.

```
# Verbindung zur Neo4j-Datenbank herstellen
uri = "bolt://localhost:7687"  # URI der Neo4j-Datenbank
username = "neo4j"  # Benutzername für die Authentifizierung
password = "password"  # Passwort für die Authentifizierung
driver = GraphDatabase.driver(uri, auth=(username, password))

main_Folder = "Main-Data"

file_agency = os.path.join(main_Folder, 'agency.txt')
file_routes = os.path.join(main_Folder, 'routes.txt')
file_trips = os.path.join(main_Folder, 'trips.txt')
file_stop_times = os.path.join(main_Folder, 'stop_times.txt')
file_stops = os.path.join(main_Folder, 'stops.txt')

df1 = pd.read_csv(file_agency)
df1_routes = pd.read_csv(file_routes)
df1_trips = pd.read_csv(file_trips)
df1_stop_times = pd.read_csv(file_stop_times)
df1_stops = pd.read_csv(file_stops)

agency_found = False

found_doubles = 0

# Durch jede Zeile des DataFrames iterieren
for index, df_row in df1.iterrows():

    # Überprüfen, ob der Wert in der zweiten Spalte gleich einem bestimmten Wert ist
    agency_list = search_all_agencys()

    for agency in agency_list:
```

Abbildung 24: Einlesen der Einzufügenden Dateien in die Datenbank

6. Fazit

6.1. Zusammenfassung

Wissenschaftlich wurde in dieser Arbeit festgestellt, dass Verkehrsdaten, insbesondere im GTFS-Format, eine zentrale Rolle bei der Analyse und Optimierung von Verkehrssystemen spielen. Diese Daten bieten eine wertvolle Grundlage für die Planung und Verbesserung des öffentlichen Nahverkehrs, indem sie detaillierte Informationen zu Fahrgastströmen, Routen und Fahrplänen liefern.

Methodisch wurde ein umfassender Ansatz zur Datenbeschaffung, -aufbereitung und -integration verfolgt. Die Daten wurden aus verschiedenen öffentlich zugänglichen Quellen gesammelt und mittels Python-Skripten einheitlich formatiert und bereinigt. Anschließend wurden die aufbereiteten GTFS-Daten in die Graphdatenbank Neo4j geladen, wobei die spezifische Struktur von Nodes, Edges und Properties genutzt wurde, um die komplexen Beziehungen zwischen den Verkehrselementen darzustellen.

Die praktische Anwendung dieser Arbeit zeigt, dass durch die Nutzung von Graphdatenbanken wie Neo4j eine effiziente Speicherung und Darstellung von Verkehrsdaten möglich ist. Die in dieser Arbeit entwickelte Datenbankstruktur ermöglicht es, die Daten auf intuitive Weise zu modellieren und komplexe Abfragen effizient durchzuführen.

Zusammengefasst zeigt diese Arbeit, dass Graphdatenbanken, insbesondere Neo4j, aufgrund ihrer Fähigkeit zur Darstellung und Analyse komplexer Netzwerke, hervorragend zur Speicherung und Verarbeitung von Verkehrsdaten im GTFS-Format geeignet sind. Die Ergebnisse dieser Arbeit unterstreichen die Relevanz und das Potenzial von Graphdatenbanken für die effektive Verwaltung und Nutzung von Verkehrsdaten, was zu einer verbesserten Effizienz und Servicequalität im öffentlichen Nahverkehr führt.

6.2. Ausblick

Die vorliegende Arbeit hat die Integration und Nutzung von Verkehrsdaten in Graphdatenbanken untersucht und dabei gezeigt, wie diese Technologie zur Verbesserung des öffentlichen Nahverkehrs beitragen kann. Ein wichtiger nächster Schritt in der Weiterentwicklung besteht darin, die Datenbank, um zusätzliche Datensätze und Funktionen zu erweitern. Wie bereits in der Arbeit erwähnt, könnten weitere optionale Dateien aus den GTFS-Datensätzen in die Datenbank mit einbezogen werden und nicht wie bisher, ausschließlich die grundlegenden Dateien. Dies würde den Informationsgehalt der Verkehrsdaten in der Graphdatenbank weiter steigern und eine Bereicherung für die Verkehrsunternehmen darstellen. Zudem könnte die Einbindung von Echtzeitdaten aus dem GTFS-Realtime-Format dazu beitragen, aktuelle Verkehrsinformationen noch besser abzubilden und die Grundlage für dynamische Routenplanungen und Echtzeitanalysen zu schaffen. Durch die kontinuierliche Aktualisierung der Daten und die Implementierung neuer Abfragefunktionen könnten präzisere und aktuellere Einblicke in das Verkehrsgeschehen gewonnen werden.

Zukünftige Arbeiten könnten sich darauf konzentrieren, die Benutzerfreundlichkeit und die Performance der Upload Skripte sowie der Graphdatenbanken weiter zu verbessern. Beim Ausführen der in dieser Arbeit dargestellten Methoden auf große Datensätze wurde ein enormer Einbruch der Leistung herausgefunden. Trotz Optimierung der Zugriffe auf die Datenbank mit dem APOC-Plugin wurden bei großen Datensätzen enorme Upload Zeiten gemessen. Insbesondere die Skalierbarkeit der Datenbank bei wachsenden Datenmengen und die Effizienz bei der Ausführung komplexer Abfragen sind wichtige Aspekte, die weiter optimiert werden können. Zudem wäre es wertvoll, die Interoperabilität zwischen verschiedenen Datenbanksystemen zu untersuchen und Möglichkeiten zu entwickeln, wie Verkehrsdaten nahtlos zwischen verschiedenen Plattformen ausgetauscht und integriert werden können.

Zudem könnte man sich ein Weg überlegen die Daten, welche sich in der Datenbank befinden nutzerfreundlicher dem Endkonsumenten anzeigen zu lassen. Wie in diesem Artikel (Graser, 2023), welcher zur Inspiration der Datenstruktur in dieser Arbeit diente, erwähnt, gibt es eine Erweiterung Namens „Neomap“, welche Koordinatendaten eines Knoten auf einer geographischen Karte anzeigen kann. Leider ist dieses Plugin aktuell nicht mit der neusten Version der Neo4j Datenbank kompatibel und wurde daher in dieser Arbeit nicht weiter thematisiert.

Abschließend lässt sich festhalten, dass die vorliegende Arbeit die Grundlage für viele weitere Untersuchungen und Entwicklungen im Bereich der Verkehrsdaten und Graphdatenbanken gelegt hat. Die Erkenntnisse und Ergebnisse bieten wertvolle Ansätze für die kontinuierliche Verbesserung der öffentlichen Verkehrsinfrastruktur und eröffnen vielfältige Möglichkeiten für zukünftige Forschung und Entwicklung in diesem spannenden und wichtigen Bereich.

7. Literaturverzeichnis

AG, D. B. (05. 01 2024). *Digitalisierung*. Von https://www.deutschebahn.com/de/Digitalisierung abgerufen

Athanasios Maimaris, G. P. (2017). A Review of Intelligent Transportation Systems from a Communications Technology Perspective., (S. 7). Nicosia, Cyprus.

Bibri, S. E. (2019). On the sustainability of smart and smarter cities in the era of big data: an interdisciplinary and transdisciplinary literature review. *Journal of Big Data*, 6.

Brosi, D. P. (29. 05 2024). *GTFS*. Von https://gtfs.de abgerufen

Byrd, A. (20. 05 2024). *GTFS Hintergrund*. Von GTFS: https://gtfs.org/de/background/ abgerufen

Byrd, A. (20. 05 2024). *GTFS Startseite*. Von GTFS: https://gtfs.org/de/ abgerufen

Byrd, A. (20. 05 2024). *GTFS Struktur des Dokuments*. Von GTFS: https://gtfs.org/de/schedule/best-practices/#struktur-des-dokuments abgerufen

Byrd, A. (20. 05 2024). *GTFS Warum GTFS verwenden?* Von GTFS: https://gtfs.org/de/#warum-gtfs-verwenden abgerufen

Corfield, H. (03. 02 2022). *Telstra's Graphie Award for Excellence in Data Discovery: Why Neo4j?* Von Neo4j: https://neo4j.com/blog/telstras-graphie-award-for-excellence-in-data-discovery-why-neo4j/ abgerufen

Domingos, P. (2015). *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World.* United States: Basic Books.

Graser, A. (27. 11 2023). *anitagraser.com*. Von Setting up a graph db using GTFS data & Neo4J: https://anitagraser.com/2023/11/27/setting-up-a-graph-db-using-gtfs-data-neo4j/ abgerufen

Ian Robinson, J. W. (2015). *Graph Databases New opportunities for connected Data.* Sebastopol, United States of America: O'Reilly Media.

Juan de Dios Ortúzar, L. G. (2011). *Modelling Transport.* John Wiley & Sons, Ltd.

Kitchin, R. (2013). The real-time city? Big data and smart urbanism. *GeoJournal*, 14.

Neo4j. (31. 05 2024). *APOC*. Von Neo4j: https://neo4j.com/docs/apoc/current/overview/apoc/ abgerufen

OpenData. (29. 05 2024). *ÖPNV-Initiative für Deutschland* . Von https://www.opendata-oepnv.de/ht/de/ueber abgerufen

OpenData. (29. 05 2024). *Delfi*. Von https://www.opendata-oepnv.de/ht/de/organisation/delfi/startseite abgerufen

Pramod J. Sadalage, M. F. (2012). *NoSQL Distilled A Brief Guide to the Emerging World of Polyglot Persistence.* United States: Addison-Wesley Professional.

Renzo ANgles, C. G. (2008). Survey of graph database models. *ACM Computing Survey*, 40.

TriMet. (31. 05 2024). *About Trimet*. Von TriMet: https://trimet.org/about/index.htm#:~:text=TriMet%20provides%20bus%2C%20light%20rail,a%20better%20place%20to%20live. abgerufen

Verkehr, B. f. (05. 01 2024). *Deutschlands Plattform für Daten, die etwas bewegen*. Von https://mobilithek.info/about abgerufen

8. Anhang

Alle in der Arbeit erwähnten Dateien (ausschließlich der GTFS-Datensätze) und Skripte lassen sich unter folgendem öffentlichen Github Repository einsehen:
https://github.com/DennisHeine02/Verkehrsdaten-in-Graphdatenbanken