

Vergleich zweier Datenbanksysteme am Beispiel einer Social-Media-Plattform

vorgelegt von

Max Harnisch

Matrikelnummer: 000 000

am Fachbereich VI – Informatik und Medien
der Berliner Hochschule für Technik

Hausarbeit

als Prüfungsleistung im Modul

Datenbank-Technologien (FHBSWF MIM 20 W24)

im Studiengang

Medieninformatik Online

Tag der Abgabe XX. Monat 20XX



Betreuer

Prof. Dr.-Ing. Nils Jensen
Nikolai Alex

Kurzfassung

Platzhalter

Abstract

Platzhalter

Inhaltsverzeichnis

1. Einleitung	1
1.1 Problemstellung	1
1.2 Ziel	1
1.3 Methodik.....	1
2. Grundlagen.....	1
2.1 Relationale Datenbanken	2
2.2 Graph-Datenbanken	2
3. Anforderungen eines Sozial-Media-Netzwerks	2
4. Datenmodellierung	3
5. Implementierung	4
5.1 Flask-Webanwendung	5
5.2 Datengenerierung	5
5.3 PostgreSQL - Datenbankerstellung.....	6
5.4 Neo4J - Datenbankerstellung	7
6. Abfragebeispiele	8
6.1 Aktuelle 5 Beiträge von Freunden	8
6.2 Eigene Beiträge	11
6.3 Neuen Beitrag hinzufügen	15
7. Leistungsmessung	16
7.1 Cold-Start-Latenz	16
7.2 Connection-Pooling	16
8. Vor- und Nachteile der Datenbanksysteme	17
9. Fazit	17
Literaturverzeichnis	18
Abbildungsverzeichnis	18

1. Einleitung

1.1 Problemstellung

Soziale Medien ermöglichen es ihren Nutzern sich miteinander zu vernetzen. Inhalte können jederzeit in unlimitierter Menge erstellt und verbreitet werden. Daraus resultieren einige Anforderungen, an für solche Plattformen verwendete Datenbanksysteme, sie müssen großen Datenmengen wie Benutzerinformationen, Freundschaften, Likes und Kommentare, effizient speichern und abfragen können.

Meist ist das Ziel einer solchen Plattform zu wachsen. Für steigende Nutzerzahlen sind dabei die Faktoren Skalierbarkeit und Performance essenziell, da komplexe Abfragen, zu stark vernetzten Daten, nicht von allen Datenbanksystemen gleich gut bearbeitet werden können. Es gilt herauszufinden, welche Datenbanksysteme hinsichtlich der Faktoren Skalierbarkeit und Performance am besten zur Verwendung in solchen Anwendungsfällen geeignet sind.

1.2 Ziel

Ziel dieser Arbeit ist es, die Eignung dreier verschiedener Datenbankmodelle, einer relationalen und einer Graph- Datenbank, zum Einsatz bei Social-Media-Plattformen zu untersuchen. Durch einen Performancevergleich bei typischen Abfragen, soll ermittelt werden, welches der Systeme sich am besten für beziehungsintensive und komplexe Datenstrukturen eignet. Es sollen Erkenntnisse darüber gewonnen werden, wie sich die Wahl des Datenbanksystems auf die Skalierbarkeit und die Reaktionsgeschwindigkeit auswirkt. Diese Erkenntnisse sollen durch eine Webseite interaktiv visualisiert werden.

1.3 Methodik

Um die Performance der Datenbanksysteme zu bewerten, wird ein Social-Media-Datenmodell erstellt, welches Benutzerprofile, Freundschaften, Beiträge, Likes und Kommentare umfasst. Dieses Datenmodell wird auf die drei Datenbanktechnologien, relationale Datenbank (PostgreSQL), Graph-Datenbank (Neo4j) und dokumentenorientierte Datenbank (MongoDB) übertragen. Es folgt die Generierung von Testdaten, mit denen die Interaktionen typischer Nutzer über die Social-Media-Plattform simuliert wird. Anschließend werden diese Daten für Nutzerabfragen genutzt, wie z.B. die Suche nach anderen Nutzern oder die Anzeige relevanter Beiträge anderer Nutzer. Diese Abfragen werden auf den drei Datenbanken durchgeführt, wobei die Laufzeiten gemessen werden. Abschließend werden die Ergebnisse analysiert und die jeweiligen Stärken und Schwächen der Datenbanken herausgearbeitet.

Zur interaktiven Visualisierung der Projekthinhalte wird eine Webanwendung erstellt, mit der ein Nutzer eine Beispiel-Social-Media-Plattform nutzen kann und live vergleichende Performancedaten zu seinen ausgeführten Aktionen erhält.

2. Grundlagen

In diesem Abschnitt werden die Grundlagen der beiden Datenbankarten, relational und graphbasiert, sowie deren wesentliche Eigenschaften und Anwendungsbereiche behandelt. Relationale Datenbanken werden traditionell für strukturierte und stark normalisierte Daten eingesetzt, während Graph-Datenbanken zunehmend bei der Verarbeitung von stark vernetzten Daten Anwendung finden.

2.1 Relationale Datenbanken

Relationale Datenbanken basieren auf dem von Edgar F. Codd entwickelten relationalen Modell. Daten werden in Tabellen organisiert, wobei jede Tabelle eine Entität repräsentiert. Die Zeilen einer Tabelle stehen für Datensätze, während die Spalten die Eigenschaften dieser Entität beschreiben. Beziehungen zwischen den Tabellen werden durch Primär- und Fremdschlüssel abgebildet, was es ermöglicht, Daten in einer stark normalisierten Form zu speichern.

Ein wesentliches Merkmal relationaler Datenbanken ist die Unterstützung der **ACID-Eigenschaften** (Atomicity, Consistency, Isolation, Durability), die Transaktionen sicher und zuverlässig machen. Datenabfragen und -manipulationen werden über die standardisierte Abfragesprache SQL (Structured Query Language) durchgeführt, die auf relationaler Algebra basiert.

Relationale Datenbanken sind besonders geeignet für strukturierte Daten, bei denen feste Schemas und komplexe Abfragen benötigt werden. Ihre Einschränkungen liegen jedoch bei der Verarbeitung von stark vernetzten Daten oder hochdimensionalen Abfragen, die durch die Notwendigkeit mehrerer Joins oft ineffizient werden.

2.2 Graph-Datenbanken

Graph-Datenbanken sind darauf ausgelegt, hochvernetzte Daten effizient zu speichern und zu verarbeiten. Sie verwenden ein flexibles Modell, das aus Knoten (Entitäten), Kanten (Beziehungen) und Eigenschaften besteht. Dieses Modell ermöglicht die intuitive Darstellung und Abfrage von Beziehungen zwischen Entitäten, ohne dass komplexe Joins wie in relationalen Datenbanken erforderlich sind.

Ein Hauptvorteil von Graph-Datenbanken ist die Effizienz bei der Traversierung von Beziehungsstrukturen, wie sie häufig in sozialen Netzwerken, Empfehlungsdiensten oder betrugsbezogenen Analysen vorkommen. Die Abfragesprachen, wie Cypher für Neo4j, ermöglichen es, selbst komplexe Abfragen mit einer leicht verständlichen und deklarativen Syntax zu formulieren.

„Cypher entstand im Jahr 2011 als ein Projekt des Unternehmens Neo4j und stellt eine Abfragesprache für die Graphdatenbank Neo4j zur Verfügung. Im Jahr 2015 machte Neo4j die Abfragesprache mit openCypher zu einem Open-Source-Projekt. Es handelt sich bei Cypher um eine deklarative Abfragesprache, die für Property-Graphdatenbanken vorgesehen ist. Mit der Sprache lassen sich dank der übersichtlichen Syntax selbst komplexe Abfragen oder Datenaktualisierungen in Graphdatenbanken schnell und einfach formulieren.“ (Vgl. Luber, Litzel, 2020)

Graph-Datenbanken bieten Flexibilität in der Schemaentwicklung und sind besonders geeignet für Daten mit dynamisch wachsenden Strukturen oder komplexen Beziehungsmodellen. Sie können relationalen Datenbanken jedoch in Anwendungsfällen unterlegen sein, bei denen strukturierte Daten mit wenigen Beziehungen verarbeitet werden.

3. Anforderungen eines Sozial-Media-Netzwerks


Social-Media-Netzwerke stellen komplexe Anforderungen an die zugrundeliegenden Datenbanksysteme, da sie eine Vielzahl von unterschiedlichen Datenarten und Beziehungen speichern und verarbeiten müssen. Zentral ist dabei die Verwaltung der Benutzerprofile, die neben grundlegenden Informationen auch dynamische Daten umfassen. Das Datenbanksystem muss nicht nur große Mengen an Benutzerdaten effizient speichern, sondern auch in der Lage sein, Beziehungen zwischen diesen Daten schnell analysieren zu können. Besonders bei der Darstellung und Abfrage von

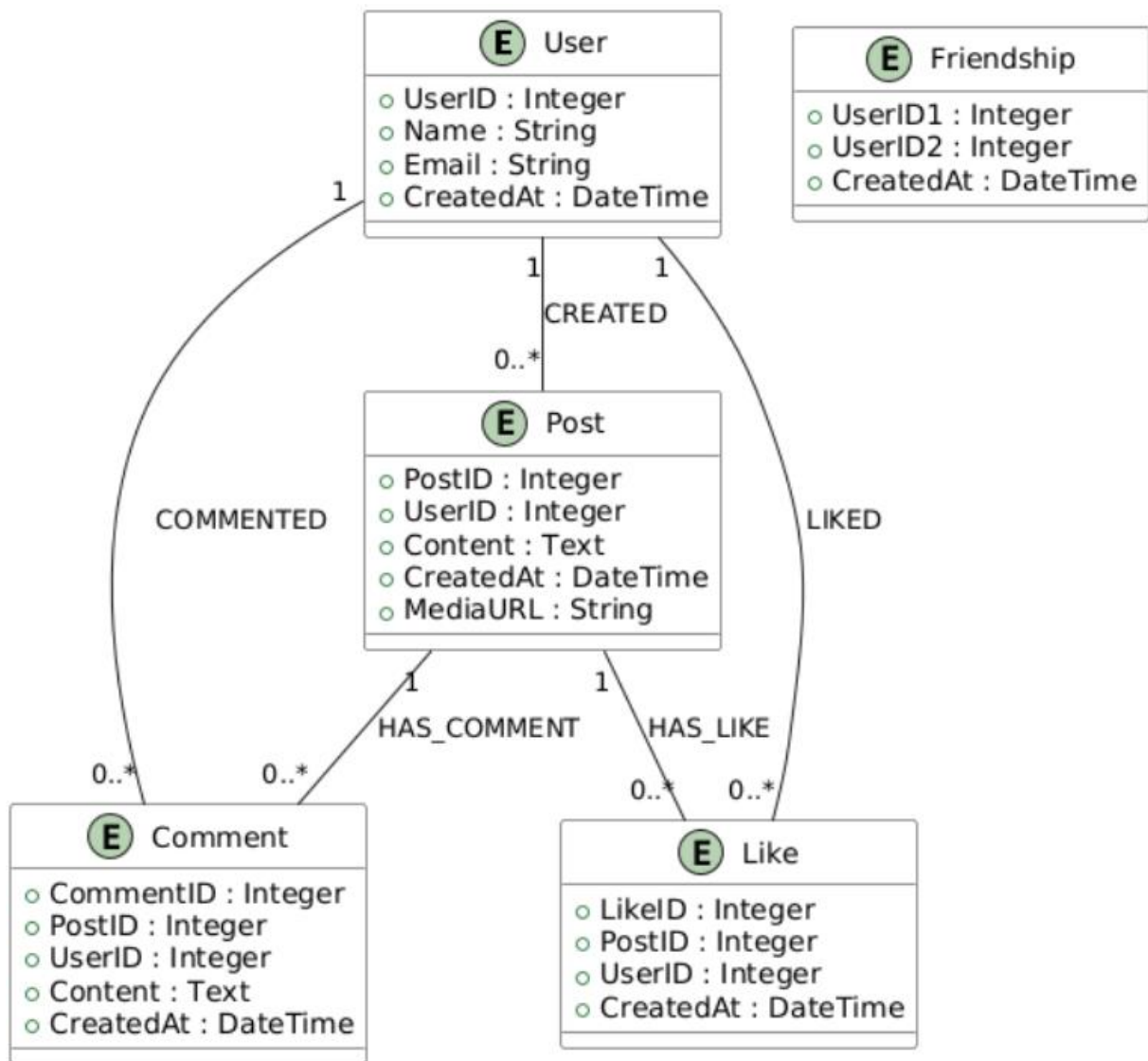
Verbindungen, z.B. der Abfrage nach Freuden von Freunden oder der Identifikation von gemeinsamen Interessen, stoßen traditionelle Datenbanksysteme oft an ihre Grenzen. (Vgl. Robinson, Webber & Eifrem, 2015, S.105-122)

Zusätzlich ist die Skalierbarkeit ist besonders wichtig, da die Leistung der Plattform auch bei Millionen von Nutzern und Milliarden von Interaktionen stabil sein muss. Darüber hinaus müssen Social-Media-Datenbanken Echtzeitanforderungen erfüllen, um Nutzern eine reibungslose und responsive Benutzererfahrung zu ermöglichen. Daten wie Kommentare oder Likes sollten ohne Verzögerung sichtbar sein, und Empfehlungen für Freundschaften oder Inhalte müssen schnell abgerufen werden können. (Vgl. Robinson, Webber & Eifrem, 2015, S.166 f.)

Insgesamt muss ein Datenbanksystem für ein Social-Media-Netzwerk also eine hohe Effizienz und Skalierbarkeit bei der Verarbeitung von stark vernetzten und dynamischen Daten bieten. (Vgl. Robinson, Webber & Eifrem, 2015, S.105-122)

4. Datenmodellierung

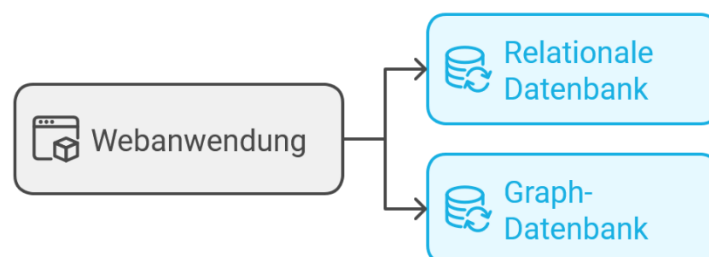
Im Rahmen dieser Untersuchung soll eine Sozial-Media-Plattform mit Ähnlichkeiten zu Instagram erstellt werden. Es sollen Beiträge erstellt werden können, die sich aus einem Textbeitrag und einem Bild zusammensetzen. Ein Nutzer muss zur Nutzung der Plattform ein Nutzerprofil anlegen und kann dann Freunde hinzufügen, Beiträge erstellen und Beiträge anderer Nutzer sehen, diese Liken und Kommentieren. Um dies zu realisieren werden die fünf in der Abbildung  Abgebildeten Entitäten benötigt.



Sie finden den zur Modellgenerierung genutzten Code im in Kapitel [X](#) verlinkten GitHub Repo.

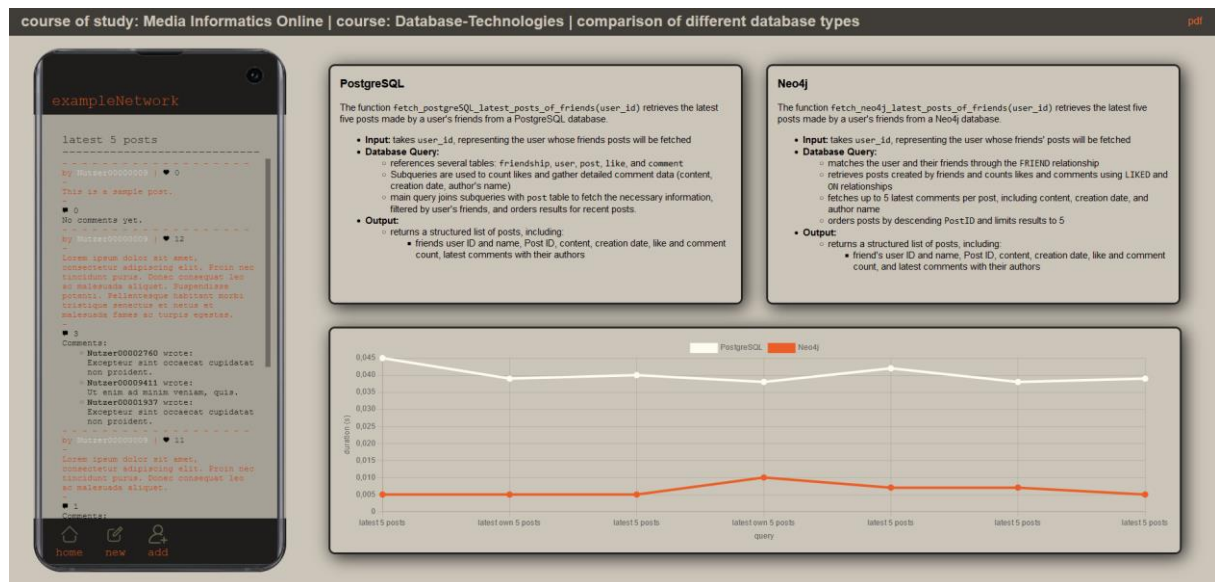
5. Implementierung

Zur Umsetzung des Vergleichs von Relationalen und Graph-Datenbanken wurde im Rahmen dieses Projektes ein Webanwendung entwickelt, die es durch die Darstellung eines Handys ermöglicht, eine Sozial-Media-App zu bedienen. Sozial-Media-Apps benötigen eine Vielzahl von Daten, welche bereitgestellt werden müssen. Die Webanwendung visualisiert interaktiv welche Datenbankoperationen beim Nutzen der App benötigt werden und erfasst die Zeit, die die



Datenbanken zum Beantworten der Anfragen benötigen. Die dadurch erhobenen, vergleichbaren Laufzeiten werden visualisiert.

Damit fungiert die Webanwendung als Nutzerfreundliches Abfragetool und ermöglicht eine realitätsnahe, vergleichbare Darstellung eines datengetriebenen Sozial-Media-Netzwerkes. Die Webanwendung kommuniziert direkt mit den lokalen Datenbanken und hält damit den verwendeten Technologieumfang möglichst gering.



5.1 Flask-Webanwendung

Bei der erstellten Webanwendung handelt es sich um eine mit dem Framework Flask erstellte Webanwendung, die ausschließlich für den Zweck als Visualisierungstool erstellt wurde. Die Anwendung verfügt nicht über zur Veröffentlichung nötige Sicherheitsmechanismen.

Das Mikro-Webframework Flask wurde von dem österreichischen Softwareentwickler Armin Ronacher entworfen und zählt heute zu den beliebtesten Webframeworks für Python. Der Funktionsumfang ist einfach gehalten, Funktionen können aus bereits bestehenden Bibliotheken eingebunden werden. (Vgl. Luber, Litzel, 2021)

Die Kommunikation zum Webserver erfolgt in Flask über die Schnittstelle Python Web Server Gateway Interfaces (WSGI). Lizenziert ist Flask unter freier BSD-Lizenz. (Vgl. Luber, Litzel, 2021)

„Die BSD-Lizenz steht für eine Reihe freier Lizenzen, die nur minimale Beschränkungen hinsichtlich der Nutzung und Weiterverbreitung der lizenzierten Software vorsieht.“ (Kern, 2017)

5.2 Datengenerierung

Zum Vergleich der Datenbanksysteme am Beispiel eines Sozial-Media-Netzwerkes werden Nutzerdaten benötigt. Der Inhalt dieser Nutzerdaten ist zum Leistungsvergleich der Datenbanken nicht relevant. Daher wurden die Daten mithilfe von zu diesem Zweck erstellten Skripten generiert und im JSON-Format zwischengespeichert.

Da die Inhalte nicht relevant sind, erhalten Nutzer einen einzigartigen Benutzernamen und eine einzigartige E-Mailadresse (z. B. Benutzername = Nutzer00000001; E-Mail = 00000001@example.com).

Hier beispielhaft das Skript zum Erstellen von Nutzern (`gen_users.py`).


```

1. import json
2. from datetime import datetime
3.
4. def generate_user_data(filename, num_users):
5.     try:
6.         # Load existing users if file exists
7.         with open(filename, "r", encoding="utf-8") as f:
8.             users = json.load(f)
9.     except FileNotFoundError:
10.        # If file doesn't exist, start with an empty list
11.        users = []
12.
13.    # Determine starting point based on the last UserID in the file
14.    start_id = users[-1]["UserID"] + 1 if users else 1
15.
16.    for i in range(num_users):
17.        user_id = start_id + i
18.        user = {
19.            "UserID": user_id,
20.            "Name": f"Nutzer{str(user_id).zfill(8)}",
21.            "Email": f"{str(user_id).zfill(8)}@example.com",
22.            "CreatedAt": datetime.utcnow().isoformat(),
23.        }
24.        users.append(user)
25.
26.    with open(filename, "w", encoding="utf-8") as f:
27.        json.dump(users, f, indent=4, ensure_ascii=False)
28.
29.    print(f"{num_users} users have been added to {filename}")
30.
31. # Example usage with repetition
32. repetitions = 10 # Number of times to repeat the process
33. users_per_iteration = 1000
34.
35. for _ in range(repetitions):
36.     try:
37.         generate_user_data("users.json", users_per_iteration)
38.     except Exception as e:
39.         print(f"An error occurred during this iteration: {e}")
40.

```

Sie finden die weiteren Skripte im in Kapitel [X](#) verlinkten GitHub Repo.

Anschließend wurden die Daten den Datenbanken hinzugefügt. So konnte sichergestellt werden, dass die Daten in den Beiden Datenbanken identisch sind.

5.3 PostgreSQL - Datenbankerstellung

Die PostgreSQL-Datenbank wurde mithilfe von SQLAlchemy erstellt und befüllt. Dabei wurde in zwei Schritten vorgegangen, es wurde das Datenbankschema erstellt und die Datenbank mit JSON-Daten befüllt.

Zur Erstellung des Datenbankschemas wurden die Relationen und deren Beziehungen modelliert. Die Datenbank hat die fünf Haupttabellen user, post ,comment, like.

Hier ein Auszug aus dem Skript zum Erstellen der Relation „post“ (genDB1_postgreSQL.py).

```

1. class Post(Base):
2.     __tablename__ = 'post'
3.
4.     PostID = Column(Integer, primary_key=True, autoincrement=True)
5.     UserID = Column(Integer, ForeignKey('user.UserID'), nullable=False)
6.     Content = Column(Text, nullable=True)
7.     CreatedAt = Column(DateTime, default=datetime.utcnow)
8.
9.     user = relationship("User", back_populates="posts")

```

```
10.     comments = relationship("Comment", back_populates="post")
11.     likes = relationship("Like", back_populates="post")
12.
```

Sie finden die vollständige Datei im in Kapitel **X** verlinkten GitHub Repo.

Die Tabellen sind wie folgt definiert:

- User-Tabelle: Enthält Benutzerinformationen wie UserID, Name, Email und CreatedAt. Dabei wird sichergestellt, dass die E-Mail-Adresse eindeutig ist.
- Post-Tabelle: Speichert die Inhalte der Posts und verknüpft sie mit den Nutzern über UserID.
- Comment-Tabelle: Verknüpft Kommentare mit Posts und Nutzern.
- Like-Tabelle: Verknüpft Likes mit Posts und Nutzern.
- Friendship-Tabelle: Modelliert Freundschaften zwischen Nutzern. Eine eindeutige Einschränkung (UNIQUE CONSTRAINT) sorgt dafür, dass keine doppelten Freundschaftseinträge möglich sind.

Nach Erstellung der Datenbank, nachdem alle Relationen generiert wurden, mussten diese mit Daten befüllt werden. Dazu wurden die Relationen in der zweiten Phase, mit den Daten aus JSON-Dateien befüllt. Dafür wurde SQLAlchemy Core verwendet.

Dabei wurde zuerst das bestehende Schema der Datenbank mit `metadata.reflect()` reflektiert, anschließend wurden die Daten eingelesen und dann mit der `insert()`-Methode in die Datenbank eingefügt.

Hier ein Beispiel für das Einfügen von Benutzerdaten:

```
1. # Insert Users
2. with open('data/users.json', 'r', encoding='utf-8') as f:
3.     users_table = metadata.tables['user']
4.     users = json.load(f)
5.     conn.execute(insert(users_table), users)
```

5.4 Neo4J - Datenbankerstellung

Die Neo4j Graph-Datenbank wurde in mehreren Schritten, mithilfe des neo4j-Python-Treibers aufgebaut. Das Vorgehen kann in vier Phasen unterteilt werden.

In der ersten Phase, dem Verbindungsaufbau und der Konfiguration wurde eine globale Driver-Instanz erstellt, wozu die URI, der Benutzername und das Passwort zentral definiert wurden.

```
1. # Database 2 | neo4j
2.
3. NEO4J_URI = "bolt://localhost:7687"
4. NEO4J_USER = "neo4j"
5. NEO4J_PASSWORD = "db2_neo4j"
6.
7. # Erstellen einer globalen Driver-Instanz
8. driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD))
```

Die zweite Phase stellt die Eindeutigkeit der Daten sicher, indem mit der Methode `ensure_constraints()`, vor dem hinzufügen von Daten, sichergestellt wird, dass eindeutige Einschränkungen (CONSTRAINTS) für jeden Knotentyp gesetzt werden.

```
1. CREATE CONSTRAINT IF NOT EXISTS FOR (u:User) REQUIRE u.UserID IS UNIQUE
```

Diese eindeutigen Einschränkungen sind mit dem PRIMARY KEY oder mit UNIQUE-Constraints aus relationalen Datenbanken wie PostgreSQL vergleichbar.

Der Datenimport, die dritte Phase, liest die Daten aus den entsprechenden JSON-Dateien aus und fügt diese mittels MERGE in die Datenbank ein. MERGE stellt dabei sicher, dass die Knoten nur erstellt werden, wenn diese noch nicht existieren. So werden Duplikate vermieden.

```
1. MERGE (u:User {UserID: $UserID})
2. SET u.Name = $Name, u.Email = $Email, u.CreatedAt = datetime($CreatedAt)
```

In der vierten Phase wurden die Beziehungen erstellt. Dazu müssen die Beziehungen zwischen den Knoten explizit definiert werden. Hier am Beispiel der Beziehung CREATED zwischen User und Post.

```
MATCH (u:User {UserID: $UserID}), (p:Post {PostID: $PostID})
```

```
MERGE (u)-[:CREATED]->(p)
```

Diese simplen Anfragen ermöglichen es auch komplexere Beziehungsnetzwerke abzubilden. Den vollständigen Code zum erstellen und befüllen der Neo4j-Datenbank finden Sie im in Kapitel [X](#) verlinkten GitHub Repo.

6. Abfragebeispiele

Um die unterschiedlichen Datenbanken vergleichbar testen zu können wurde drei für eine Sozial-Media-Plattform typische Datenbank abfragen ausgewählt und für sowohl für PostgreSQL als auch für neo4j modelliert.

6.1 Aktuelle 5 Beiträge von Freunden

Bei erfassen der Anforderungen eines Sozial-Media-Netzwerk wurde als ein Grundbestandteil, das erhalten von Beiträgen vernetzter Personen erkannt. Als Nutzer möchte an die Posts der mit einem befreundeten Nutzer sehen. Dazu müssen die Beiträge der Freunde geladen werden, es muss ermittelt werden wie oft die Beiträge mit gefällt mir markiert wurden und ggf. Kommentare geladen werden.

6.1.1 PostgreSQL:

Um diese Anforderung umzusetzen wurde für die PostgreSQL-Datenbank die Funktion `fetch_postgreSQL_latest_posts_of_friends()` erstellt. Die Funktion ruft nicht die Posts der mit dem User befreundeten Personen ab, aber auch Metadaten wie die Anzahl der Likes, die Anzahl der Kommentare und die letzten 5 Kommentare. Hier wurde angenommen, dass die letzten fünf Kommentare standartmäßig geladen werden und weitere, sofern nötig, über eine gesonderte Anfrage nachgeladen werden.

Die Funktion beginnt mit der Initialisierung einer SQLAlchemy-Session, welche für die Kommunikation mit der Datenbank zuständig ist. Bevor die eigentlichen Daten abgerufen werden, wird eine sogenannte „Warm-Up“-Funktion namens `warm_up_postgresql()` aufgerufen, die sicherstellt, dass die Datenbankverbindung aktiv ist und mögliche Performance-Optimierungen wie Caching durchgeführt werden.

Um die benötigten Metadaten abzufragen, wurden zwei Subqueries erstellt:

- Like-Subquery: Die Abfrage gruppiert die Likes nach PostID und zählt die Gesamtanzahl der Likes pro Post.

```
1. like_count_subquery = (
```

```

2.     session.query(
3.         like_table.c.PostID,
4.         func.count(like_table.c.LikeID).label("Likes")
5.     )
6.     .group_by(like_table.c.PostID)
7.     .subquery()

```

- **Kommentare-Subquery:** Die Abfrage gruppiert die Kommentare nach PostID, zählt die Anzahl der Kommentare und aggregiert die aktuellsten fünf. Es werden Details wie der Kommentarinhalt, der Erstellungszeitpunkt und der Name des Autors angefragt.

```

1. comment_count_subquery = (
2.     session.query(
3.         comment_table.c.PostID,
4.         func.count(comment_table.c.CommentID).label("CommentCount"),
5.         func.array_agg(
6.             func.json_build_object(
7.                 "CommentID", comment_table.c.CommentID,
8.                 "Content", comment_table.c.Content,
9.                 "CreatedAt", comment_table.c.CreatedAt,
10.                "AuthorName", user_table.c.Name
11.            )
12.        ).label("LatestComments")
13.    )
14.    .join(user_table, user_table.c.UserID == comment_table.c.UserID)
15.    .group_by(comment_table.c.PostID)
16.    .subquery()
17. )

```

Die Hauptabfrage kombiniert die Subqueries mit den Relationen post, user und friendship. Die friendship-Tabelle wird genutzt, um die Freunde eines Nutzers anhand der UserID1 (Nutzer) und UserID2 (Freund) zu identifizieren. Danach werden die post- und user-Tabellen eingebunden, um die Beiträge und ihre Autoren zu referenzieren. Die Subqueries für Likes und Kommentare werden mittels outerjoin eingebunden, sodass auch Posts ohne Likes oder Kommentare berücksichtigt werden können. Es werden nur Beiträge der Freunde des angegebenen Nutzers (UserID1) abgerufen, wobei die neuesten Beiträge (PostID.desc()) priorisiert werden. Die Anzahl der Ergebnisse ist auf fünf beschränkt.

```

1. query = (
2.     session.query(
3.         post_table.c.PostID,
4.         post_table.c.Content,
5.         post_table.c.CreatedAt,
6.         user_table.c.UserID.label("FriendID"),
7.         user_table.c.Name.label("FriendName"),
8.         func.coalesce(like_count_subquery.c.Likes, 0).label("Likes"),
9.         func.coalesce(comment_count_subquery.c.CommentCount, 0).label("CommentCount"),
10.        func.coalesce(comment_count_subquery.c.LatestComments, []).label("LatestComments"),
11.    )
12.    .join(friendship_table, friendship_table.c.UserID2 == post_table.c.UserID)
13.    .join(user_table, user_table.c.UserID == post_table.c.UserID)
14.    .outerjoin(like_count_subquery, like_count_subquery.c.PostID == post_table.c.PostID)
15.    .outerjoin(comment_count_subquery, comment_count_subquery.c.PostID ==
post_table.c.PostID)
16.    .filter(friendship_table.c.UserID1 == user_id)
17.    .order_by(post_table.c.PostID.desc())
18.    .limit(5)
19. ).all()

```

Die Ergebnisse dieser Abfrage werden in einer strukturierten Form aufbereitet. Jeder Beitrag wird als ein Python-Dictionary dargestellt, welches Informationen wie die Post-ID, den Inhalt, den Erstellungszeitpunkt, die Anzahl der Likes, die Anzahl der Kommentare und eine Liste der neuesten

Kommentare enthält. Diese Struktur erleichtert die Verarbeitung der Daten durch das aufrufende System, beispielsweise für die Anzeige in einer Benutzeroberfläche.

6.1.2 Neo4j:

Zur Umsetzung der Abfrage mit Neo4j, wurde die Funktion `fetch_neo4j_latest_posts_of_friends` entwickelt. Die Funktion fragt die neuesten Beiträge der Freunde eines Nutzers aus einer Neo4j-Datenbank ab. Dabei werden nicht nur die Beiträge, sondern auch ergänzende Informationen wie die Anzahl der Likes, die Anzahl der Kommentare und die letzten fünf Kommentare eines jeden Beitrags mit den zugehörigen Autoren bereitgestellt. Diese Funktion kombiniert die Stärke von Graph-Datenbanken mit einer optimierten Abfragestruktur, um relevante Daten schnell und übersichtlich bereitzustellen.

Zu diesem Zweck wird eine Neo4j-Sitzung, die über den globalen Treiber (driver) initiiert wird, verwendet.

```
1. # Database 2 | neo4j
2.
3. NEO4J_URI = "bolt://localhost:7687"
4. NEO4J_USER = "neo4j"
5. NEO4J_PASSWORD = "db2_neo4j"
6.
7. # Erstellen einer globalen Driver-Instanz
8. driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD))
9.
10. # Schließen des Drivers beim Beenden des Programms
11. atexit.register(driver.close)
```

Da zum Performancevergleich der Datenbanken die Laufzeiten der Abfragen gemessen werden, wird vor der Abfrage eine simple Abfrage zum aufwecken der Datenbank durchgeführt. Diese Abfrage sorgt dafür, dass die Verbindung zur Neo4j-Datenbank aktiv ist und potenzielle Verzögerungen zum initialisieren der Verbindung, keine Auswirkungen auf die Messergebnisse haben.

Die Hauptabfrage ist in Cypher geschrieben, sie kombiniert mehrere Schritte miteinander. Die folgende Abfrage identifiziert alle Freunde (friend) des Nutzers (u) und deren Beiträge (post).

```
1. MATCH (u:User {UserID: $UserID})-[:FRIEND]->(friend:User)-[:CREATED]->(post:Post)
```

Es wird die Anzahl der Nutzer ermittelt, die einen Beitrag mit gefällt mir markiert haben. Da aber auch Beiträge ohne diese Markierung abgefragt werden sollen (nicht jeder Post wird geliked, bzw. jeder neue Post ist nicht geliked), wird OPTIONAL MATCH verwendet.

```
1. OPTIONAL MATCH (post)<-[:LIKED]-(like:User)
```

Die folgenden Abfragen sammeln die Kommentare zu einem Beitrag und ordnen jedem Kommentar den Verfasser zu.

```
1. OPTIONAL MATCH (post)<-[:ON]-(comment:Comment)
2. OPTIONAL MATCH (comment)<-[:WROTE]-(commentAuthor:User)
```

Anschließend erfolgt die Aggregation der Daten, es werden die Likes und Kommentare gezählt und eine Liste der aktuellsten fünf Kommentare pro Beitrag, mit den Erstellungsdaten und dem Nutzernamen des Verfassers zusammengefasst.

```
1. WITH DISTINCT friend, post, count(DISTINCT like) AS likeCount, count(DISTINCT comment) AS
   commentCount,
2.      collect(DISTINCT {
```

```

3.         CommentID: comment.CommentID,
4.         Content: comment.Content,
5.         CreatedAt: comment.CreatedAt,
6.         AuthorName: commentAuthor.Name
7.     })[0..5] AS latestComments

```

Anschließend werden die Ergebnisse nach der neusten PostID sortiert und auf die fünf neusten Beiträge begrenzt.

Die Ergebnisse der Abfrage werden in Python weiterverarbeitet, um eine benutzerfreundliche Struktur zu erzeugen. Jeder Beitrag wird als ein Dictionary dargestellt.

```

1. {
2.     "posted_by_UserID": record["FriendID"],
3.     "FriendName": record["Name"],
4.     "PostID": record["PostID"],
5.     "Content": record["Content"],
6.     "CreatedAt": str(record["CreatedAt"]),
7.     "Likes": record["likeCount"],
8.     "CommentCount": record["commentCount"],
9.     "LatestComments": [
10.         {
11.             "CommentID": comment["CommentID"],
12.             "Content": comment["Content"],
13.             "CreatedAt": str(comment["CreatedAt"]),
14.             "AuthorName": comment["AuthorName"]
15.         }
16.         for comment in record["latestComments"]
17.     ]
18. }

```

Die Funktion kombiniert die Vorteile einer Graph-Datenbank mit der Effizienz von Cypher-Abfragen, um komplexe Beziehungen und Metadaten schnell und präzise abzurufen. Die strukturierte Rückgabe der Daten erleichtert die Integration ins Frontend und ermöglicht einen leichten Vergleich der Abfrageergebnisse, zu den der PostgreSQL-Abfrage.

6.1.3 Bezug zur Webanwendung

Die Beschriebenen Funktionen werden in der erarbeiteten Webanwendung zum laden der 5 aktuellsten Beiträge von Nutzern, denen der Nutzer folgt verwendet. Diese Abfrage wird beim starten der Anwendung automatisch ausgeführt, da diese Inhalte auf dem Homescreen der App dargestellt werden. Sobald ein Nutzer während der Nutzung der App erneut den Homescreen aufruft, wird die Abfrage erneut ausgeführt.

6.2 Eigene Beiträge

Wenn ein Nutzer einen Beitrag verfasst und diesen teilt, dann muss dieser Nutzer weiter auf den eigenen Beitrag zugreifen können, die eigenen Beiträge laden können. Zu diesem Zweck wurden die folgenden zwei Funktionen zum Laden der eigenen Beiträge erstellt.

6.2.1 PostgreSQL:

Die Funktion `fetch_postgresql_own_posts` wurde entwickelt, um die fünf neuesten Beiträge eines Nutzers aus einer PostgreSQL-Datenbank abzurufen. Dabei werden zusätzlich Metadaten wie die Anzahl der Likes, die Anzahl der Kommentare sowie die letzten fünf Kommentare eines jeden Beitrags mitsamt den Autoreninformationen bereitgestellt. Diese Funktion nutzt SQLAlchemy, um die Datenbankabfragen effizient zu gestalten und die Ergebnisse in einer strukturierten Form zurückzugeben.

Ablauf der Funktion

Nach dem Aufbau einer Datenbank-Session und einem optionalen Warm-Up zur Initialisierung der Verbindung, werden die notwendigen Tabellen referenziert. Anschließend folgen die Subqueries und die Hauptabfrage.

Erstellung von Subqueries

Zwei Subqueries werden verwendet, um die Likes und Kommentare der Beiträge effizient zu aggregieren.

Likes-Subquery

Die erste Subquery zählt die Anzahl der Likes für jeden Beitrag. Dies geschieht durch eine Gruppierung nach PostID:

```
1. like_count_subquery = (  
2.     session.query(  
3.         like_table.c.PostID,  
4.         func.count(like_table.c.LikeID).label("Likes")  
5.     )  
6.     .group_by(like_table.c.PostID)  
7.     .subquery()  
8. )
```

Das Ergebnis dieser Subquery liefert eine Zuordnung von PostID zu der Anzahl der Likes.

Kommentare-Subquery

Die zweite Subquery zählt die Anzahl der Kommentare und sammelt Details zu den letzten fünf Kommentaren eines jeden Beitrags, einschließlich der Informationen über den Verfasser:

```
1. comment_count_subquery = (  
2.     session.query(  
3.         comment_table.c.PostID,  
4.         func.count(comment_table.c.CommentID).label("CommentCount"),  
5.         func.array_agg(  
6.             func.json_build_object(  
7.                 "CommentID", comment_table.c.CommentID,  
8.                 "Content", comment_table.c.Content,  
9.                 "CreatedAt", comment_table.c.CreatedAt,  
10.                "AuthorName", user_table.c.Name  
11.             )  
12.         )[:5].label("LatestComments")  
13.     )  
14.     .join(user_table, user_table.c.UserID == comment_table.c.UserID)  
15.     .group_by(comment_table.c.PostID)  
16.     .subquery()  
17. )
```

Diese Subquery sammelt die Details der letzten fünf Kommentare in einer Liste mit JSON-Objekten.

Hauptabfrage

Die Hauptabfrage kombiniert die Ergebnisse der Subqueries mit den Primärtabellen und wendet Filter und Sortierungen an:

```
1. query = (  
2.     session.query(  
3.         post_table.c.PostID,  
4.         post_table.c.Content,  
5.         post_table.c.CreatedAt,  
6.         func.coalesce(like_count_subquery.c.Likes, 0).label("Likes"),
```

```

7.         func.coalesce(comment_count_subquery.c.CommentCount, 0).label("CommentCount"),
8.         func.coalesce(comment_count_subquery.c.LatestComments, []).label("LatestComments"),
9.     )
10.    .filter(post_table.c.UserID == user_id) # Filter für die eigenen Beiträge
11.    .outerjoin(like_count_subquery, like_count_subquery.c.PostID == post_table.c.PostID)
12.    .outerjoin(comment_count_subquery, comment_count_subquery.c.PostID ==
post_table.c.PostID)
13.    .order_by(post_table.c.PostID.desc()) # Sortierung nach neuester PostID
14.    .limit(5) # Begrenzung auf 5 Ergebnisse
15. ).all()

```

Die Abfrage filtert ausschließlich Beiträge des angegebenen Nutzers (UserID) und fügt die Likes- und Kommentare-Subqueries über LEFT JOIN ein. So werden auch Beiträge ohne Likes oder Kommentare korrekt behandelt. Die Ergebnisse werden nach der neuesten PostID sortiert und auf fünf begrenzt.

Rückgabe der Ergebnisse

Nach Abschluss der Abfrage werden die Ergebnisse in ein benutzerfreundliches Format umgewandelt. Jeder Beitrag wird als Dictionary dargestellt:

```

1. own_posts = [
2.     {
3.         "PostID": row.PostID,
4.         "Content": row.Content,
5.         "CreatedAt": row.CreatedAt.strftime('%Y-%m-%d %H:%M:%S'),
6.         "Likes": row.Likes,
7.         "CommentCount": row.CommentCount,
8.         "LatestComments": row.LatestComments
9.     }
10.    for row in query
11. ]

```

Die strukturierte Rückgabe enthält für jeden Beitrag folgende Informationen:

- **PostID:** Die eindeutige ID des Beitrags.
- **Content:** Der Inhalt des Beitrags.
- **CreatedAt:** Der Zeitpunkt der Erstellung, formatiert als lesbare Zeichenkette.
- **Likes:** Die Anzahl der Likes des Beitrags.
- **CommentCount:** Die Anzahl der Kommentare.
- **LatestComments:** Eine Liste der letzten fünf Kommentare mit Details wie CommentID, Content, CreatedAt und dem Namen des Verfassers.

Die Laufzeit der Abfrage wird gemessen und zusammen mit den Beitragsdaten zurückgegeben:

```

1. return {
2.     "postgresql_own_posts": own_posts,
3.     "fetch_postgresql_own_posts_runtime": runtime
4. }

```

Die Funktion kombiniert effiziente SQLAlchemy-Abfragen mit Subqueries, um sowohl die Hauptdaten als auch die zugehörigen Metadaten eines Beitrags bereitzustellen. Dank der klar strukturierten Rückgabe kann das Ergebnis direkt in einer Benutzeroberfläche angezeigt werden, was diese Funktion ideal für soziale Netzwerke oder Content-Plattformen macht.

6.2.2 Neo4J:

Die Funktion `fetch_neo4j_own_posts` ermöglicht es, die fünf neuesten Beiträge eines Nutzers aus einer Neo4j-Datenbank abzurufen. Neben den Beiträgen selbst werden zusätzliche Informationen wie die Anzahl der Likes, die Anzahl der Kommentare sowie die letzten fünf Kommentare eines jeden Beitrags inklusive der Autoren bereitgestellt. Diese Daten werden mithilfe einer zentralen Cypher-Abfrage ermittelt, die sowohl Beiträge als auch zugehörige Metadaten in einem einzigen Schritt aggregiert.

Die Abfrage startet mit dem Abruf der Beiträge des Nutzers. Hierfür wird die Beziehung `[:CREATED]` zwischen dem Nutzerknoten (User) und den zugehörigen Beitragknoten (Post) genutzt:

```
1. MATCH (u:User {UserID: $UserID})-[:CREATED]->(post:Post)
```

Anschließend werden die Likes zu den Beiträgen gezählt. Dazu wird die Beziehung `[:LIKED]` zwischen dem Beitrag und den Nutzerknoten herangezogen. Damit auch Beiträge ohne Likes berücksichtigt werden, wird ein `OPTIONAL MATCH` verwendet:

```
1. OPTIONAL MATCH (post)<-[:LIKED]-(like:User)
```

Um die Kommentare zu einem Beitrag zu erfassen, wird eine weitere `OPTIONAL MATCH`-Abfrage verwendet, die die Beziehung `[:ON]` zwischen einem Kommentar (Comment) und einem Beitrag (Post) abbildet:

```
1. OPTIONAL MATCH (post)<-[:ON]-(comment:Comment)
```

Die Verfasser der Kommentare werden über die Beziehung `[:WROTE]` mit den Kommentar-Knoten verknüpft. Auch hier wird `OPTIONAL MATCH` eingesetzt, um sicherzustellen, dass Beiträge ohne Kommentare nicht ausgeschlossen werden:

```
1. OPTIONAL MATCH (comment)<-[:WROTE]-(commentAuthor:User)
```

Sobald alle relevanten Daten erfasst sind, wird mithilfe von Aggregatfunktionen eine strukturierte Ausgabe vorbereitet. Die Anzahl der Likes und Kommentare wird gezählt, und die letzten fünf Kommentare eines jeden Beitrags werden gesammelt:

```
1. WITH DISTINCT post, count(DISTINCT like) AS likeCount, count(DISTINCT comment) AS
   commentCount,
2.     collect(DISTINCT {
3.         CommentID: comment.CommentID,
4.         Content: comment.Content,
5.         CreatedAt: comment.CreatedAt,
6.         AuthorName: commentAuthor.Name
7.     })[0..5] AS latestComments
```

Schließlich werden die gesammelten Informationen zurückgegeben. Die Ergebnisse werden nach der neuesten PostID sortiert und auf fünf Beiträge begrenzt:

```
1. RETURN post.PostID AS PostID, post.Content AS Content, post.CreatedAt AS CreatedAt,
2.        likeCount, commentCount, latestComments
3. ORDER BY post.PostID DESC
4. LIMIT 5
```

Die Ergebnisse der Abfrage werden in Python weiterverarbeitet, um sie in ein benutzerfreundliches Format zu bringen. Dabei wird jeder Beitrag als ein Dictionary dargestellt, das alle relevanten Informationen enthält:

```
1. own_posts = [
2.     {
3.         "PostID": record["PostID"],
```

```

4.         "Content": record["Content"],
5.         "CreatedAt": str(record["CreatedAt"]),
6.         "Likes": record["likeCount"],
7.         "CommentCount": record["commentCount"],
8.         "LatestComments": None if record["latestComments"] is None else [
9.             {
10.                 "CommentID": comment["CommentID"],
11.                 "Content": comment["Content"],
12.                 "CreatedAt": str(comment["CreatedAt"]),
13.                 "AuthorName": comment["AuthorName"]
14.             }
15.             for comment in record["latestComments"]
16.         ]
17.     }
18.     for record in result
19. ]

```

Zusätzlich wird die Laufzeit der Abfrage gemessen, um die Performance der Funktion zu überwachen:

```

1. start_time = perf_counter()
2. # Abfrage ausführen
3. end_time = perf_counter()
4. runtime = end_time - start_time

```

Die finale Rückgabe der Funktion enthält die Liste der eigenen Beiträge des Nutzers sowie die Laufzeit der Abfrage:

```

1. return {
2.     "neo4j_own_posts": own_posts,
3.     "fetch_neo4j_own_posts_runtime": runtime
4. }

```

Diese Funktion kombiniert die Stärken der Graph-Datenbank Neo4j mit einer effizienten Cypher-Abfrage, um eine Vielzahl von Informationen in einem einzigen Schritt abzurufen. Die klare Struktur der Ergebnisse ermöglicht eine einfache Integration in Frontend-Systeme und bietet eine leistungsfähige Möglichkeit, eigene Beiträge eines Nutzers mitsamt aller relevanten Metadaten darzustellen.

6.2.3 Bezug zur Webanwendung

Platzhalter

6.3 Neuen Beitrag hinzufügen

Ein weiter essenzieller Bestandteil zur Nutzung eines Sozial-Media-Netzwerks ist das Verfassen von eigenen Beiträgen (Posts).

6.3.1 PostgreSQL:

Die Funktion `create_post_postgresql` ermöglicht es, einen neuen Beitrag für einen bestimmten Nutzer in einer PostgreSQL-Datenbank zu erstellen. Dabei wird die nächste verfügbare PostID (höchste vorhandene PostID plus eins) ermittelt und der neue Beitrag mit den zugehörigen Daten in die Datenbank eingefügt. Da sich die Struktur der PostgreSQL-Funktionen ähneln, wird hier nur auf die relevante Datenbankinteraktion eingegangen, alles weitere finden Sie im vorherigen Teil des Kapitels.

Die zentrale Funktion der Datenbankinteraktion besteht aus zwei Teilen. Im Ersten, wird die Relation `Post` nach der höchsten vorhandenen PostID durchsucht.

```

1. SELECT COALESCE(MAX("PostID"), 0) AS MaxPostID FROM post

```

Im zweiten Teil wird der neue Beitrag hinzugefügt, dazu wird einen neuen Eintrag in die Tabelle Post eingefügt und diesem die ermittelte PostID, die UserID des Verfassers und der Inhalt des Beitrags mit aktuellem Zeitstempel hinzugefügt.

```
1. INSERT INTO post ("PostID", "UserID", "Content", "CreatedAt")
2. VALUES (:PostID, :UserID, :Content, :CreatedAt)
```

6.3.2 Neo4J:

Die Funktion `create_post_neo4j` fügt einen neuen Beitrag der Neo4j-Datenbank hinzu, wobei die nächste verfügbare PostID berechnet wird und der Beitrag mit dem Nutzer verknüpft wird. Der gesamte Prozess wird durch eine zentrale Cypher-Abfrage umgesetzt. Da sich die Struktur der Neo4j-Funktionen ähneln, wird hier nur auf die relevante Datenbankinteraktion eingegangen, alles weitere finden Sie im vorherigen Teil des Kapitels.

```
1. MATCH (u:User {UserID: $UserID})
2. WITH u, coalesce(MAX((:Post).PostID), 0) + 1 AS newPostID
3. CREATE (p:Post {PostID: newPostID, Content: $Content, CreatedAt: datetime()})
4. MERGE (u)-[:CREATED]->(p)
5. RETURN newPostID
```

Die Funktion `coalesce(MAX((:Post).PostID), 0)` sucht die höchste existierende PostID und setzt sie auf 0, falls keine Posts existieren. Ein neuer Post-Knoten wird mit den Attributen PostID, Content, und CreatedAt erzeugt. Der neue Post wird über die CREATED-Beziehung mit dem Benutzer verbunden. Die Abfrage gibt die neu erzeugte PostID zurück, um sie weiterzuverwenden.

Diese Abfrage ist kompakt und nutzt die Stärke von Neo4j, um mit minimalen Schritten sowohl Daten zu erstellen als auch Beziehungen zu pflegen.

6.3.3 Bezug zur Webanwendung

Die beschriebenen Funktionen fügen neue Beiträge zur Datenbank hinzu. Dies kann in der Darstellung der App, in der Webanwendung ausprobiert werden. Wenn der Nutzer einen neuen Beitrag erstellt wird dann zum einen der Prozess des Erstellens eines neuen Beitrags ausgeführt und zum anderen die Abfrage nach ebenfalls vom Nutzer erstellen Posts ausgelöst. Auf die Entwicklung eines echten Sozial-Media-Netzwerks bezogen, könnte und müsste man anders vorgehen. Bei einer tatsächlichen Anwendungsentwicklung sollte auf das sofortige neu laden der eigenen Posts, je nach Internetverbindung verzichtet werden. Solche Überlegungen würden jedoch den Rahmen dieser Ausarbeitung sprengen.

7. Leistungsmessung

Ziel dieser

7.1 Cold-Start-Latenz

Da die erfassten Laufzeiten vergleichbar sein müssen, gilt es mögliche Latenzen auszuschließen. Die Cold-Start-Latenz ist...

7.2 Connection-Pooling

Platzhalter

8. Vor- und Nachteile der Datenbanksysteme

Platzhalter

9. Fazit

Platzhalter

Literaturverzeichnis

Kern, B. (1. Dezember 2017). Open Source-Lizenzen - Die BSD-Lizenz. Abgerufen am 3. Februar 2023 von <https://wss-redpoint.com/open-source-lizenzen-die-bsd-lizenz>

Luber, S., Litzel, N. (03. März 2020). Was ist Cypher? Abgerufen am 19. Dezember 2024 von <https://www.bigdata-insider.de/was-ist-cypher-a-912813/>

Luber, S., Litzel, N. (28. Januar 2021). Was ist Flask? Abgerufen am 11. Dezember 2024 von <https://www.bigdata-insider.de/was-ist-flask-a-994166/>

Robinson, I., Webber, J., & Eifrem, E. (2015). Graph Databases: New Opportunities for Connected Data. O'Reilly Media.

Links

<https://www.python.org/>

<https://flask.palletsprojects.com/en>

<https://www.sqlalchemy.org/>

<https://pypi.org/project/neo4j/>

platzhalter

Abbildungsverzeichnis

Platzhalter