Institut für Softwaretechnik

UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

JGraLab: Persistenz in Datenbanken
Diplomarbeit

vorgelegt von:

Nicolas Vika

Matrikelnummer 119920121

am 30. September 2010

Betreuer: Prof. Dr. Jürgen Ebert, Dr. Volker Riediger, Daniel Bildhauer

{ebert | riediger | dbildh}@uni-koblenz.de

Erklärung

Hiermit erkläre ich, wie in §10 Abschnitt 6.2 der Diplomprüfungsordnung für Studierende der Informatik an der Universität Koblenz-Landau gefordert, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

Koblenz, den 30.10.2010

Zusammenfassung

Im Rahmen dieser Diplomarbeit wird eine Datenbank-Persistenzlösung für die TGraphenbibliothek *JGraLab* entwickelt. Diese erlaubt das Persistieren von TGraphen in Datenbanken ausgewählter Technologien und sorgt gleichzeitig dafür, dass die Datenbankpersistenz eines TGraphen für den Benutzer transparent bleibt.

Nach der Anforderungserhebung und der Recherche zur Bewertung der Tauglichkeit von bereits bestehenden Werkzeugen für dieses Projekt, wird die Anwendungsdomäne erläutert. Anschließend wird dargelegt wie die Persistierung von TGraphen mit allen ihren Eigenschaften in Datenbanken ermöglicht werden kann.
Dem schließt sich der konzeptuelle Entwurf an, in dem die Details der Lösung beschrieben werden. Als nächstes wird der objektorientierte Feinentwurf zur Integration der Lösung in die TGraphenbibliothek *JGraLab* entwickelt, der die Grundlage der programmatischen Umsetzung bildet.

Eine Anleitung zur Verwendung der Lösung und eine Bewertung des Laufzeitverhaltens der umgesetzten Implementation schließen die Arbeit ab.

Inhaltsverzeichnis

1. Einführung

1.1. Hintergrund

Ein Forschungsschwerpunkt der Arbeitsgruppe Ebert am Institut für Softwaretechnik liegt auf dem Gebiet der Graphentechnologie als Herangehensweise zur Umsetzung von Anwendungssystemen mittels Graphen sowie graphentheoretischer Instrumente und Verfahren. Graphen kommen dabei zur Repräsentation von strukturierten Informationen zum Einsatz [Ebe10].
Ein Graph besteht aus einer endlichen Menge von Knoten und Kanten. Knoten werden durch die Kanten verbunden. Grafisch können Knoten als Punkte und Kanten als Linien, welche die Punkte verbinden, veranschaulicht werden [EGSW98]. Zum Beispiel kann eine Straßenkarte durch einen Graphen repräsentiert werden. Dabei kann ein Knoten eine Kreuzung und eine Kante eine Straße darstellen. Straßen bzw. Kanten verbinden dann Kreuzungen bzw. Knoten.
In der Informatik bilden Graphen eine grundlegende Datenstruktur, die sich zur anschaulichen und strukturierten Darstellung vieler Sachverhalte und Probleme eignet. Unter anderem dienen Graphen dabei zur Modellierung des jeweils betrachteten Realitätsausschnitt. Darüber hinaus können Graphen mit formalen Methoden analysiert werden [Bra94].

In der Arbeitsgruppe Ebert kommen die eigens entwickelten *TGraphen* zum Einsatz. Dabei handelt es sich um typisierte, attributierte, gerichtete und angeordnete Graphen. Die Eigenschaften von TGraphen gestatten es, unterschiedlichste Arten von Graphen mit unterschiedlicher Mächtigkeit darzustellen [EGSW98]
Die möglichen Ausprägungen eines TGraphen werden durch ein Schema beschrieben. Ein Schema legt Typisierung und Attributierung der Knoten und Kanten sowie die Richtung der Kanten im TGraph fest. Ein Schema ist in der Regel durch eine Perspektive auf die abzubildenden Sachverhalte bestimmt und abstrahiert dazu über bestimmte Merkmale, die für die Perspektive nicht von Belang sind.
So kann je nach Schema eine Graphrepräsentation eine sehr vereinfachte bis sehr detaillierte Abbildung eines Sachverhaltes darstellen. Im weiter oben angeführten Beispiel der Repräsentation einer Straßenkarte durch einen Graphen wurde nicht festgelegt, dass verschiedene Typen von Straßen (z. B. Spiel-, Land- oder Bundesstraße) im Graph unterschieden werden. Es wurde über die Arten der Straßen abstrahiert. Mit einem Schema kann man dies festlegen.

Alle nötigen Funktionalitäten zum Traversieren[1] und Manipulieren von TGraphen stellt eine in Java implementierte Klassenbibliothek namens *JGraLab* [Kah06] zur Verfügung. Zu Analysezwecken von TGraphen wurde zusätzlich die Anfragesprache *GreQL* entwickelt [Mar06].
TGraphen kommen unter anderem im Projekt GUPRO [EGSW98] in einem konkreten Anwendungsgebiet zum Einsatz. Das Projekt beschäftigt sich mit Möglichkeiten der Analyse von Software-Altsystemen, um Erkenntnisse zur Wartung dieser Systeme zu gewinnen. Entwickelte Verfahren und Werkzeuge arbeiten dazu auf TGraphrepräsentationen der untersuchten Software-Systeme. Sind die Quelltexte der Software verfügbar, können *Faktenextraktoren* [BV08] diese parsen und daraus TGraphen extrahieren. Zur Auswertung der Software-Systeme müssen erzeugte TGraphen traversiert werden. JGralab ist deshalb für eine schnelle Traversierung optimiert.

1.2. Problemstellung und Motivation

TGraphen können aus beliebig vielen Knoten und Kanten bestehen und somit sehr groß werden. Ihre Größe wird in der Regel von der Granularität des Graphschemas und dem Umfang der repräsentierten Daten bestimmt. Für das Beispiel aus dem letzten Abschnitt der Straßenkarte als Graph bedeutet dies, dass die Granularität des Graphschemas den Detailgrad der resultierenden Karte festlegt. Je mehr Details in der Karte repräsentiert werden sollen, desto größer kann der Graph mit der Anzahl der dargestellten Kreuzungen und Straßen werden. Betrachtet man ein Software-System, so hängt die Größe des Graphen ebenfalls von der Granularität des Schemas und der Anzahl zu parsender Quelltextzeilen ab. Mit der Größe eines Graphen steigt ebenso sein Speicherbedarf.
Der Speicherbedarf eines Graphen kann problematisch werden, sobald er die Größe des verfügbaren physikalischen Hauptspeichers übersteigt. So wird bisher ein TGraph aus einer Datei komplett in den Hauptspeicher geladen. Ist der Speicherbedarf des Graph größer, als Hauptspeicher verfügbar ist, so lagert die Speicherverwaltung des Betriebssystems einen Teil des Graphen in die Auslagerungsdatei aus. Diese liegt in der Regel auf einer Festplatte. Dadurch ist die Traversierung des Teils des TGraphen, der ausgelagert wurde, langsamer als die Traversierung des Teils im Hauptspeicher, da der ausgelagerte Teil zuvor wieder in den Hauptspeicher eingelagert werden muss. Die Traversierung kann dadurch um mehrere Größenordnungen langsamer ausfallen.

Eine langsamere Traversierung ist nicht unbedingt problematisch, kommt es jedoch zum *Seitenflattern* (auch Page-Thrashing genannt), wird das ganze System unbenutzbar. Dieses Phänomen kann auftreten, wenn sich mindestens ein Prozess im Speicher nicht lokal verhält und sein Verhalten der Auslagerungsstrategie entgegenläuft. Nicht lokal bedeutet, dass der Prozess nicht

[1] Unter Traversierung werden Verfahren zum Durchlaufen eines Graphen, um alle Elemente einmal zu besuchen, zusammengefasst.

auf gleichen und benachbarten Daten arbeitet - er springt im Speicher hin und her. Fordert der Prozess dabei andauernd ausgelagerte Seiten[2] an, müssen diese erst wieder eingelagert werden. Lagert das Betriebssystem dabei auch noch Seiten aus, die von einem Prozess (i. d. R. gibt es mehr als einen) als nächstes (ebenfalls andauernd) wieder angefordert werden, muss es diese erst gerade ausgelagerten Seiten wieder einlagern - man spricht vom Seitenflattern.

Ist der verfügbare Speicher erschöpft, erhöht sich die Wahrscheinlichkeit, dass ein Seitenflattern auftritt. Denn sollen dann ausgelagerte Seiten eingelagert werden, so müssen zuvor dazu Seiten aus dem Hauptspeicher ausgelagert werden, um Platz für die Einlagerung der noch ausgelagerten Seiten zu schaffen. Geschieht dies zu oft, entsteht ebenfalls ein Flattern von Seiten zwischen Hauptspeicher und Festplatte. Dabei verbringt das System erheblich mehr Zeit mit dem Ein- und Auslagern von Daten als mit der eigentlichen Aufgabe.

Diesem Problem kann begegnet werden, indem der Graph nicht komplett in den Hauptspeicher geladen wird, sondern nur die jeweils nötigen Teile des Graphen dynamisch geladen werden. Mit dem TG-Dateiformat von JGraLab ist dies nicht möglich, da es nur begrenzt einen wahlfreien Zugriff erlaubt. Einen Ansatz, diesem Problem zu begegnen, bieten Datenbanken. So könnte ein TGraph statt in einer Datei in einer Datenbank persistiert werden und nur die jeweils nötigen Teile dynamisch in den Hauptspeicher geladen werden.

1.3. Ziel

Ziel der Diplomarbeit ist es, eine Möglichkeit zu erarbeiten, TGraphen in Datenbanken zu speichern und dabei eine hinreichend schnelle Traversierung zu gewährleisten, selbst wenn der Speicherbedarf des TGraphen größer als der verfügbare physikalische Hauptspeicher ausfällt. Darüber hinaus müssen alle Eigenschaften von TGraphen auch bei Speicherung in einer Datenbank erhalten bleiben. Für den Benutzer soll die Ablage des Graphen in einer Datenbank dabei möglichst transparent bleiben.

1.4. Gliederung

Die Gliederung der Diplomarbeit orientiert sich in Grundzügen am Software-Lebenslauf. Sie beginnt mit einem Kapitel zur Anforderungserhebung. Danach wird jeweils in separaten Kapiteln die durchgeführte Recherche, die Anwendungsdomäne und der Entwurf zugrundeliegender relationaler Datenmodelle behandelt. Dem folgt ein Kapitel zum Entwurf der Gesamtlösung und ein Kapitel zum objektorientierten Feinentwurf der Implementation.

[2] Der Speicher eines Rechners wird durch das Betriebssystem in Seiten organisiert.

Anschließend wird die Verwendung der realisierten Implementation erklärt und die Implementation einer Leistungsbewertung unterzogen. Die Arbeit schließt mit Fazit und Ausblick ab.

Der Arbeit sind im Anhang alle Ausführungen beigefügt, die den Rahmen der Abschnitte sprengen würden, in denen sich auf sie bezogen wird.

1.5. Zielgruppe

Die Diplomarbeit richtet sich an Leser, die mindestens über den Wissensstand eines Informatikstudenten mit *abgeschlossenem Vordiplom oder ähnlichem Bildungsgrad* verfügen. Der Leser sollte ebenfalls Grundkenntnisse in der *relationalen Algebra* von *Edgar F. Codd* [Cod83] mitbringen.

2. Anforderungserhebung

In diesem Kapitel werden alle Anforderungen an die Lösung beschrieben. Zunächst wird auf den Sinn der Anforderungserhebung und den Aufbau der Anforderungsliste eingegangen. Im Anschluss werden die gesammelten Anforderungen aufgeführt.

2.1. Sinn der Erhebung

In der Anforderungserhebung werden die Anforderungen an die zu entwickelnde Lösung gesammelt. Ergebnis dieser Erhebung ist die Anforderungsliste.
Die erhaltene Anforderungsliste dient im Projektverlauf mehreren Zwecken. Zunächst soll die Liste als gemeinsame Grundlage aller Beteiligten für die geforderten Eigenschaften der zu entwickelnden Lösung dienen. So kann bei den Beteiligten eine weitgehend deckungsgleiche Vorstellung von der Lösung entstehen. In diesem Prozess dient die Anforderungsliste auch der Entscheidung, welche Aspekte realisiert werden. Im Entwicklungsverlauf dient sie schließlich als Prüfliste, welche Aspekte bereits realisiert wurden.

2.2. Aufbau der Anforderungsliste

Eine Anforderungsliste besteht aus einer Reihe von Anforderungen und sollte vollständig, redundanz- und widerspruchsfrei sein. Jede Anforderung beschreibt genau einen Aspekt der Lösung und sollte in sich abgeschlossen, realisierbar und überprüfbar sein. Eine Anforderung setzt sich zusammen aus einer Beschreibung, ihrer Umsetzungsverbindlichkeit und ihrem Erfüllungsgrad.[1]

Umsetzungsverbindlichkeit. Jede Anforderung ist mit einer Umsetzungsverbindlichkeit (abgekürzt UV) versehen. Sie legt fest, wie sehr die Pflicht besteht, die Anforderung umzusetzen.

- MUSS- Die Anforderung muss auf jeden Fall umgesetzt werden.
- SOLL- Die Anforderung sollte umgesetzt werden, wenn es möglich ist.
- OPTIONAL- Die Anforderung kann umgesetzt werden, wenn noch Zeit dazu bleibt.

[1] Eine Anforderung besteht aus weiteren Elementen, die ausgeblendet sind, da sie an dieser Stelle nicht von Belang sind.

Beschreibung. Die Beschreibung ist ein kurzer, vollständiger, klarer und natürlichsprachlicher Satz, der genau eine Aussage macht. Er kann durch eine zusätzliche Begründung ergänzt sein.

Erfüllungsgrad. Mit der Angabe eines Erfüllungsgrads wird die Umsetzungshöhe bewertet. Der Erfüllungsgrad ist unter der Beschreibung vermerkt und *hervorgehoben*.

2.3. Gesammelte Anforderungen

Die aufgeführten Anforderungen wurden in den gemeinsamen Besprechungen von den Betreuern und dem Autor der vorliegenden Arbeit erarbeitet. Die Anforderungen wurden dann aus den Protokollen der Besprechungen abgeleitet, in einer Liste erfasst und mehrmals aktualisiert. Diesem Kapitel liegt die endgültige Fassung der Anforderungsliste zugrunde.

2.3.1. Anforderungen an Recherche

Zu Beginn der Arbeit soll nach Informationen und bestehenden Datenbank-Technologien und Werkzeugen, welche für die Lösung verwendet werden können, recherchiert werden.
An die Recherche werden folgende Anforderungen gestellt:

- MUSS - Es müssen mögliche Datenbankalternativen hinsichtlich ihrer Eignung zur Erfüllung der Anforderungen untersucht werden.
  Erfüllt. Betrachtet wurde eine Reihe von Datenbank-Technologien.
- MUSS - Bestehende Werkzeuge sollen hinsichtlich ihrer Tauglichkeit für dieses Projekt bewertet werden.
  Erfüllt.
- MUSS - Es soll mindestens eine Datenbank ausgewählt werden.
  Erfüllt. Es wurden drei verschiedene Lösungen ausgewählt.
- SOLL - Die gewählte Datenbank soll den eigenen Entwicklungsaufwand soweit wie möglich reduzieren.
  Erfüllt.
- SOLL - Recherchierte Datenbanken und Werkzeuge sollten, selbst im Falle der Unbrauchbarkeit für dieses Projekt, kurz beschrieben und abgehandelt werden.
  Erfüllt.
- MUSS - Es sollen bestehende Persistenz-Lösungen für Java untersucht werden.
  Erfüllt.

2.3.2. Anforderungen an untersuchte Datenbank-Technologien

Folgende Anforderungen werden an die gewählten Datenbank-Technologie, die aus Datenbank und Datenbankverwaltungssystem bestehen, gestellt:

- MUSS - Das Datenbankverwaltungssystem soll eine Zugriffsschnittstelle für Java bieten, damit direkt aus *JGraLab* auf sie zugegriffen werden kann.
  Erfüllt. Gewählte Systeme können über JDBC[2] *angesprochen werden.*
- SOLL - Das Datenbankverwaltungssystem soll unter einem Open-Source-Lizenzmodell verfügbar sein.
  Erfüllt. Gewählte Systeme stehen frei zur Verfügung.
- SOLL - Das Datenbankverwaltungssystem soll dem Benutzer die Möglichkeit geben, Einfluss auf serverseitige Caching- und Zugriffsstrategien zu nehmen.
  Erfüllt. Gewählte Systeme bieten dies an.
- MUSS - Die Datenbank soll die Persistierung von TGraphen mit allen ihren Eigenschaften ermöglichen.
  Erfüllt.

2.3.3. Funktionale Anforderungen

Die funktionalen Anforderungen legen fest, was die Lösung leisten soll. Folgende funktionalen Anforderungen werden an die entwickelte Gesamtlösung gestellt:

- MUSS - Die Lösung soll das Persistieren von TGraphen in Datenbanken ermöglichen.
  Erfüllt.
- MUSS - Die Ablage der TGraphen in Datenbanken soll für den Benutzer transparent bleiben.
  Erfüllt.
- MUSS - Die Lösung soll durch Erweiterung von *JGraLab* realisiert werden.
  Erfüllt.
- SOLL - Die Programmierschnittstelle von *JGraLab* soll sich nicht ändern.
  Erfüllt.
- MUSS - Alle Eigenschaften der TGraphen sollen auch in der Datenbank erhalten bleiben.
  Erfüllt.
- MUSS - Die Lösung soll strukturierte Performanzmessungen unterstützen.
  Erfüllt.

[2] Kurz für *Java Database Connectivity* [Net10a].

- SOLL - Sollten Caching- und Zugriffsstrategien der gewählten Datenbank nicht ausreichend für eine hinreichend schnelle Traversierung sein, so sollen eigene Strategien entwickelt werden.
  Erfüllt.
- SOLL - Die Lösung soll für *DHH-TGraphen* vorbereitet werden.
  Erfüllt. Es wird gezeigt, wie die Lösung zu erweitern ist, um DHH-TGraphen ebenfalls zu unterstützen.
- MUSS - Es soll ein generisches Datenbankschema entwickelt werden.
  Erfüllt.
- OPTIONAL - Es sollen mehrere Datenbankschemata entwickelt werden.
  Partiell erfüllt. Es wurde das generische Datenbankschema umgesetzt und eine mögliche Alternative skizziert.
- MUSS - Sollte der Hauptspeicherbedarf des Graphen größer sein, als Hauptspeicher verfügbar ist, so soll der Graph nicht komplett geladen werden.
  Erfüllt.
- MUSS - Die Lösung soll nur jeweils benötigte Teile des Graphen in den Hauptspeicher laden.
  Erfüllt.
- MUSS - Die Lösung soll den verfügbaren Hauptspeicher niemals komplett ausschöpfen.
  Erfüllt.
- MUSS - Die Lösung soll das clientseitige Cachen von Teilen des Graphen erlauben.
  Erfüllt.
- SOLL - In einer Datenbank sollen beliebig viele TGraphen gespeichert werden können.
  Erfüllt.
- SOLL - Änderungen am Graphmetaschema sollen möglichst wenig Änderungen am generischen Datenbankschema nach sich ziehen.
  Erfüllt. Es werden nur wenige Teile des Graphmetaschemas im generischen Datenbankschema referenziert.

2.3.4. Anforderungen an den Entwurfs- und Entwicklungsprozess

Diese Anforderungen legen Vorgehen bei Entwurf, Spezifikation und Implementierung sowie Dokumentation und Programmierung fest.

- MUSS - Die Lösung soll in Java realisiert werden.
  Erfüllt.

- MUSS - Es sollen möglichst wenige Teile von *JGraLab* neu implementiert werden.
  Erfüllt. Es mussten alle Klassen neu implementiert werden, die Graph, Knoten, Kante und Inzidenz repräsentieren. Damit JGraLab die neue Implementation benutzen kann, mussten nur wenige zusätzliche Klassen angepasst werden.
- MUSS - Die Quelltexte sollen *Javadoc-kompatibel* kommentiert werden.
  Erfüllt.
- MUSS - Die Quelltexte sollen mit *Unit-Testing* [Hun05] überprüft werden.
  Erfüllt.
- MUSS - Quelltexte sollen nach den Prinzipien, Methoden und Techniken aus [Mar09] erstellt werden.
  Erfüllt.
- SOLL - Es sollen Entwurfsmuster [GHJV95] verwendet werden.
  Erfüllt.
- MUSS - Die Architektur der Lösung soll bereits im Entwurf in UML 2 [RQZ05] visualisiert werden.
  Erfüllt.

2.3.5. Qualitätsanforderungen

An die Qualitätseigenschaften der Lösung wird nur eine Anforderung gestellt. Diese ist von zentraler Bedeutung und lautet:

- MUSS - Eine Traversierung des TGraph soll weiterhin hinreichend schnell sein, d. h. die Traversierung soll eine vertretbare Laufzeit aufweisen.
  Nicht erfüllt. Die Traversierung ist zu langsam.

Das nächste Kapitel handelt von der Recherche, die parallel zur Anforderungserhebung stattfand.

3. Recherche

Datenbanken werden schon lange in vielen Anwendungsgebieten zur Speicherung von Informationen verwendet. Aus diesem Grund stehen eine Vielzahl von Datenbank-Technologien und begleitender Werkzeuge zur Verfügung. So wurde zunächst eine Literatur- und Online-Recherche vorgenommen, um bereits bestehende Lösungen hinsichtlich ihrer Tauglichkeit für dieses Projekt zu bewerten. Ziel der Recherche war es:

- durch gefundene Informationen und Werkzeuge den Entwicklungsaufwand zu reduzieren,
- fundierte Erkenntnisse zum Treffen von Entwurfsentscheidungen zu erhalten, und
- die einzusetzende Datenbank-Technologie auszuwählen.

Zunächst werden die zugrundeliegenden Bewertungskriterien für alle untersuchten Lösungen erläutert.

3.1. Allgemeine Bewertungskriterien

Aufgrund der Fülle an verfügbaren Datenbank-Technologien konnte im verfügbaren Zeitrahmen nicht jede gefundene Software detailliert betrachtet werden. Deshalb wurden eine Reihe von Kriterien definiert, um die Menge der zu betrachtenden Lösungen einzuschränken. Jede betrachtete Lösung wurde zunächst auf eine Reihe von allgemeinen Eigenschaften hin untersucht. Alle Lösungen, die diese Kriterien nicht erfüllen, wurden nicht näher betrachtet.

Lizenz. Damit eine gefundene Lösung verwendet werden kann, soll sie frei verfügbar sein. Sollte es sich nicht um ein Open-Source-Produkt handeln, so sollte das Produkt Lizenzen für Forschung und Lehre bieten.

Dokumentationsgrad. Neben der Lizenzsituation ist der Dokumentationsgrad einer Lösung entscheidend für ihre Verwendung. Es sollte mindestens eine umfangreiche Referenzdokumentation durch den Hersteller angeboten werden.

Zukunftssicherheit. Eine Lösung sollte mindestens noch fünf Jahre verfügbar sein. Da dies nur selten explizit vom Hersteller zugesichert wird, sollte die Lösung mindestens aktiv weiterentwickelt, gewartet oder betreut werden. Eine breite Nutzerbasis und aktive Community wirkt sich in der Beurteilung der Zukunftssicherheit ebenfalls positiv aus, da dann ein Interesse von vielen Beteiligten an einer weiteren Existenz der Lösung besteht.

3.2. Bewertungskriterien für Datenbanken-Technologien

Die betrachteten Datenbank-Technologien wurden zusätzlich auf folgende Kriterien hin untersucht. Alle Datenbanken, die diese Kriterien nicht erfüllen, wurden ebenfalls nicht näher betrachtet.

Speicherverhalten. Das Speicherverhalten des Datenbankverwaltungssystems muss beeinflussbar sein. Durch das Ablegen von TGraphen in Datenbanken soll u. a. ein clientseitig auftretendes Seitenflattern vermieden werden. Tritt jedoch ein serverseitiges Seitenflattern auf, so ist das Problem nur verlagert worden. Laufen Client und Server auf der gleichen Maschine, so verschlimmern sich die Auswirkungen des Seitenflatterns noch, denn dadurch werden beide unbenutzbar.
In den Speicherbereichen, die durch ein modernes Datenbankverwaltungssystem[1] verwaltet werden, sollte ein Seitenflattern eigentlich ausgeschlossen werden können. Diese Systeme sind sehr stark für einen schnellen Zugriff auf die verwalteten Daten und dabei auch für ein sicheres konfliktfreies Persistieren dieser optimiert [KE09].
Der Speicher des Datenbankservers wird ebenfalls vom installierten Betriebssystem verwaltet und es laufen diverse weitere Prozesse. So kann es immer noch seitens des Betriebssystems zu Problemen kommen.

Serverseitige Caching- und Zugriffsstrategien. Die Traversierung von Graphen, die in einer Datenbank persistiert wurde, soll weiterhin hinreichend schnell sein. Da der Zugriff auf Elemente in einer Datenbank sehr langsam sein kann, sollte die Datenbank dem Benutzer die Möglichkeit geben, Einfluss auf serverseitige Caching- und Zugriffsstrategien zu nehmen.

Zugriffsschnittstelle. Das Datenbankverwaltungssystem der Datenbank sollte eine Zugriffsschnittstelle für Java bieten, damit direkt aus *JGraLab* auf sie zugegriffen werden kann.

[1] Auch *Database Management System*, (kurz DBMS) genannt.

Speicherung von TGraphen. Die Datenbank muss die Persistierung von TGraphen mit allen ihren Eigenschaften ermöglichen - unabhängig davon, zu welchem Graphschema der TGraph gehört.

3.3. Untersuchte Lösungen

Im Laufe der Recherche wurden eine Reihe von Lösungen untersucht. Betrachtet wurden Vertreter der *relationalen*, *objektrelationalen* und *objektorientierten Datenbankkonzepte* sowie einige *Persistenz-Frameworks*. Schließlich wurden drei Datenbank-Technologien ausgewählt.

3.3.1. Relationale Datenbanken

Das relationale Modell wurde Anfang der 70er konzipiert und zeichnet sich durch eine *mengenorientierte* Datenverarbeitung aus. Es besteht im wesentlich aus flachen Tabellen (Relationen), in denen die Zeilen den Datenobjekten entsprechen. In dieser einfachen Struktur liegt wahrscheinlich der Erfolg des relationalen Modells begründet. Datenbanktechnologien, die dieses Modell implementieren, dominieren bis heute den Markt [KE09]. Das relationale Datenbankmodell wurde im *SQL:92-Standard* standardisiert.
Das relationale Datenbankmodell wird in der Literatur umfangreich behandelt und ist sehr gut dokumentiert. Viele Algorithmen und Konzepte, die dort zum Einsatz kommen, sind in Standardwerken nachlesbar, z. B. [KE09].
Durch die Implementation werden zwei Vertreter des relationalen Modells unterstützt.

3.3.2. Objektrelationale Datenbanken

Der objektrelationale Ansatz ging aus dem relationalen hervor. Im *SQL:99-Standard* wurde das objektrelationale Datenbankenmodell standardisiert. Das objektrelationale Konzept erlaubt in *SQL* die Definition von Objekttypen mit Struktur und Verhalten, die Anordnung dieser Typen in Typhierarchien sowie die Speicherung von Instanzen dieser Typen in Tabellen. Die persistenten Daten müssen, wie auch im relationalen Konzept, per *SQL* abgefragt werden.
Es wird damit versucht, die *semantische Lücke* zwischen der relationalen Datenbankwelt und den objektorientierten Entwurfsmethoden und Programmiersprachen zu schmälern.
Beansprucht ein Datenbankhersteller, dass sein System objektrelational sei, so bedeutet dies nicht zwingend, dass alle Konzepte des *SQL:99-Standards* angeboten werden. So gibt es zwischen den Anbietern zwar eine gemeinsame Basis, aber auch Unterschiede hinsichtlich des Umfangs, der Ausprägung und der Syntax der objektrelationalen Erweiterungen der angebotenen Datenbanksysteme.
Seit der Veröffentlichung von *SQL:99* sind weitere Standards[2] erschienen, die hauptsächlich

[2] *SQL:2003, SQL:2006 und SQL:2008.*

Funktionalitäten im Zusammenhang mit *XML* spezifizieren.
Auch das objektrelationale Datenbankmodell wird in der Literatur umfangreich behandelt und ist sehr gut dokumentiert.
Durch die Implementation wird ein Vertreter des objektrelationalen Konzepts unterstützt.

3.3.3. Objektorientierte Datenbanken

Objektorientierte Datenbanken erlauben das Persistieren von Objekten mit Struktur und Zustand in einer Datenbasis. Sie schließen die *semantische Lücke* zwischen der Datenbankwelt und den objektorientierten Entwurfsmethoden und Programmiersprachen.
Wie es auch bei den objektrelationalen Datenbanken der Fall ist, existierte bereits eine Reihe von herstellerspezifischen Lösungen, bevor ein Standard verabschiedet wurde. Die *Object Database Management Group* erarbeitete den *ODMG-Standard*, der seit 2001 in Version 3.0 vorliegt [BEJ+00]. Die Arbeiten an einer vierten Generation sind aktuell im Gange, es ist aber noch nicht abzusehen, wann diese abgeschlossen sein werden. Auch dieser Standard wird nicht immer vollständig von den Herstellern objektorientierter Datenbank-Systemen implementiert.
Objektorientierte Datenbanken sind nah am Konzept und der Denkweise bei objektorientiertem Entwurf und Entwicklung. Algorithmen und Konzepte, die dort zum Einsatz kommen, sind jedoch wenig bis gar nicht dokumentiert. Auch die Standardwerke schweigen sich dazu aus. So könnte man voraussetzen, dass ähnliche Verfahren wie im relationalen Modell zum Einsatz kommen - mit Sicherheit lässt sich das aber nicht behaupten. Somit wird auf eine Unterstützung von Vertretern des objektorientierten Datenbankmodells durch die Implementation verzichtet.

3.4. Persistenz-Frameworks

Ein Persistenz-Framework soll dem Entwickler möglichst viele Aufgaben, die bei der Persistierung von Daten anfallen, abnehmen und eine Anwendung von der Art der Datenspeicherung entkoppeln. Es bietet dazu Operationen zum Laden und Speichern von Daten aus Datenquellen.
Bereits 2002 wurde von *Sun Microsystems* ein Modell für ein herstellerunabhängiges Framework zur Persistierung von Java-Objekten spezifiziert, die sogenannten *Java Data Objects* (kurz *JDO*) [ASF10a]. Mittlerweile ist der Standard in Version 2.3 verfügbar und wird im Rahmen des *Apache DB Projects* vorangetrieben.
Eine Untermenge der Persistenz-Frameworks stellen die *ORM-Frameworks* dar. *ORM* steht für *Object-Relational Mapping* und ist ein Konzept zur Abbildung von Objekten in relationale Datenbanken.
Für ORM-Frameworks erarbeitete *Sun Microsystems* ebenfalls eine Spezifikation.[3] Die *Java Per-*

[3] *JDO* wurde zu Gunsten von *JPA* aufgegeben und an die *Apache Software Foundation* übergeben

sistence API (kurz *JPA*) wurde 2006 erstmalig veröffentlicht und ist mittlerweile in Version 2.0 verfügbar [Mic10].
Durch ein Persistenz-Framework kann eine Lösung von der eingesetzten Datenquelle weitgehend unabhängig bleiben. Dies gilt auch für den Einsatz von Datenbank-Technologien als Datenquellen, die eine abweichende Implementierung eines Standards bieten. Durch *JDO* und *JPA* werden selbst die Persistenz-Frameworks austauschbar.
Da die Austauschbarkeit von Komponenten kein Ziel der Diplomarbeit ist, wurde auch kein Persistenz-Framework verwendet.

3.5. Gewählte Datenbank-Technologien

In diesem Abschnitt werden die gewählten Lösungen jeweils mit einer kurzen Charakterisierung, ihrer zusammengefassten Historie und den bewerteten Kriterien vorgestellt.
Die Betrachtung der nicht gewählten Lösungen ist in Anhang A angefügt.

3.5.1. MySQL

Produkt	MySQL
Hersteller	Oracle
Homepage	`http://www.mysql.com/`
Aktuelle Version	5.1.50
Unterstützte Plattformen	Linux, diverse Unix-Derivate, Windows, Mac OS X, i5/OS, OpenVMS etc.
Implementierungssprache	C, C++
Maximale Datenbankgröße	unbegrenzt
Maximale Tabellengröße	64 bis 65.536 TByte[4]
Zielarchitektur	Client-Server, Embedded
Datenabfrage über	SQL
Lizenz	GPL, kommerziell
Status	aktiv

Charakterisierung. *MySQL* ist ein Datenbankverwaltungssystem für relationale Datenbanken und gilt als die populärste Lösung unter den Open-Source-Vertretern. Es ist sowohl als Hochverfügbarkeitslösung als auch für Embedded-Szenarien geeignet. So kann eine reduzierte Version kompiliert werden, die für mobile oder integrierte Geräte geeignet ist.
Im Gegensatz zu anderen Datenbankverwaltungssystemen unterstützt *MySQL* verschiedene *Speicher-Engines*. Diese sind jeweils für spezielle Einsatzszenarien vorgesehen. So existieren

[4] Abhängig von der Speicher-Engine.

Speicher-Engines für transaktionssichere als auch für nicht-transaktionssichere Tabellen.
MySQL implementiert zum größten Teil den *SQL:92-Standard*, Teile des *SQL:99-Standards*[5] und bietet darüber hinaus eine Reihe von proprietären Erweiterungen.

Historie. Die Ursprünge von *MySQL* lassen sich bis ins Jahr 1979 zurückverfolgen. In diesem Jahr begann die Entwicklung des Datenbanksystems *UNIREG*. 1986 wurde es in der Programmiersprache C umgesetzt, um auch auf Unix-Systemen zum Einsatz kommen zu können. Um Daten aus UNIREG-Datenbanken in Zukunft auch für Web-Anwendungen zur Verfügung zu stellen, begann 1994 die Arbeit an MySQL. Ein Jahr später erschien die erste interne Version und 2000 schließlich die erste öffentliche Vorabfassung. Diese trug bereits die Versionsnummer 3.21, ist seitdem open-source, wird fortlaufend weiterentwickelt und ist für mehrere Betriebssysteme verfügbar.
Der Hersteller *MySQL AB* wurde 2008 von der Firma *Sun Microsystems* aufgekauft, welche 2009 von *Oracle* übernommen wurde.

Zukunftssicherheit. Aufgrund seines hohen Verbreitungsgrades durch Einsatz in unzähligen Projekten kann die Zukunft von *MySQL* als sicher angesehen werden.
Eine Abschätzung der *Zukunftssicherheit des Lizenzmodells* von *MySQL* ist zum jetzigen Zeitpunkt jedoch nicht möglich. Zwar wird *MySQL* in sehr vielen freien sowie kommerziellen Lösungen eingesetzt und befindet sich aktiv in der Weiterentwicklung zur Version 5.5, jedoch steht *MySQL in Konkurrenz zu oracle-eigenen Datenbanklösungen*. *Oracle* konnte die Übernahme im Januar 2010 vollständig abschließen und hat noch keine Zukunftspläne für die weitere Vermarktung von *MySQL* bekanntgegeben.

Lizenz. *MySQL* verfolgt ein duales Lizenzsystem. Es ist ein Open-Source-Projekt, d. h. der Quelltext ist frei verfügbar und steht unter der *GNU General Public License*. Gleichzeitig steht eine kommerzielle Variante zur Verfügung, die für Closed-Source-Projekte geeignet ist und professionellen Support beinhaltet.

Dokumentationsgrad. Der Dokumentationsgrad zu *MySQL* kann als *sehr gut* bewertet werden. Über die Homepage ist das *Referenzhandbuch in mehreren Sprachen* verfügbar. *Tutorials*, *technische Artikel* und eine kurze *Literaturliste* von erschienen Büchern runden das Bild ab. Leider ist keines der Bücher komplett online verfügbar. Fragen, die über diese Dokumentation hinausgehen, können in der großen und aktiven Community rund um *MySQL* in *Foren*, einer *Mailingliste* und *Chats* gestellt werden.

[5] Dennoch bezeichnet der Hersteller *MySQL* nicht als objekt-relational.

Zugriffsschnittstelle. Mit Java kann über *JDBC* auf eine *MySQL*-Datenbank zugegriffen werden. Ferner bietet es *ODBC*[6] und native Schnittstellen für eine Reihe von weiteren Programmiersprachen.

Speicherverhalten. Je nach Speicher-Engine lassen sich die Größe von diversen Puffern konfigurieren.

Serverseitige Caching- und Zugriffsstrategien. *MySQL* bietet drei Indexarten an (R-/R+-Baum, Hash). Die mögliche Definition eines Index auf eine Tabelle hängt von der verwendeten Speicher-Engine ab. Daten können auch geclustered werden, dies hängt ebenfalls von der verwendeten Speicher-Engine ab.
Das Tuning von Datenbanken wird in einem Kapitel des Referenzhandbuchs [MDT10] behandelt.

Da der Quelltext von *MySQL* frei verfügbar ist, kann generell auf o. a. Eigenschaften Einfluss genommen werden. So können modifizierte Varianten kompiliert und eingesetzt werden.

3.5.2. JavaDB/Apache Derby

Produkt	JavaDB
Hersteller	Oracle
Homepage	`www.oracle.com/technetwork/java/javadb/`
Aktuelle Version	10.5.3.0
Unterstützte Plattformen	durch Java alle
Implementierungssprache	Java
Maximale Datenbankgröße	2,306 EB
Maximale Tabellengröße	abhängig von maximaler Dateigröße des Betriebssystems
Zielarchitektur	Client-Server, Embedded
Datenabfrage über	SQL
Lizenz	Apache License 2.0
Status	aktiv

Charakterisierung. *JavaDB* ist ein Datenbankverwaltungssystem für relationale Datenbanken. Bei *JavaDB* handelt es sich um eine durch *Oracle* unterstützte Distribution von *Apache Derby*. Als leichtgewichtiges System ist es vor allem für den eingebetteten Einsatz bestimmt. Damit ist nicht der Einsatz auf (mobilen) Kleingeräten gemeint, sondern dass die Datenbank als Teil der Anwendung fungiert und nur von ihr darauf zugegriffen wird. *JavaDB* kann darüber hinaus aber auch im klassischen *Client-Server-Szenario* betrieben werden.

[6] Kurz für *Open Database Connectivity*.

JavaDB implementiert den *größten Teil des SQL:92 Standards und Teile des SQL:99- und SQL:2003-Standards*.
Im Gegensatz zu anderen Datenbankverwaltungssystemen ist das Format einer Datenbank betriebssystemunabhängig. Einmal angelegte Datenbanken können ohne weiteres auf Rechner mit anderen Betriebssystemen übernommen werden.

Historie. Die Ursprungsversion *JBMS* wurde 1997 von dem Unternehmen *Cloudscape Inc* veröffentlicht und später in *Cloudscape* umbenannt. Der Hersteller wurde 1999 von *Informix Software* aufgekauft. 2001 kaufte *IBM* wiederum die Datenbanksparte von *Informix Software* auf und führte die Entwicklung von *Cloudscape* fort, um es primär in seinen eigenen java-basierten Produkten einzubetten.
2004 übergab *IBM* den Quelltext von *Cloudscape* der *Apache Software Foundation*. Seitdem ist die Lösung als freie Software unter dem Namen *Derby* bekannt. Ab 2005 beteiligte sich *Sun Microsystems* an der Weiterentwicklung von *Derby* und integriert es seit 2006 unter dem Namen *JavaDB* in das *JDK 6*.
2009 wurde *Sun Microsystems* schließlich von *Oracle* aufgekauft.

Zukunftssicherheit. Die zukünftige Verfügbarkeit von *JavaDB* und *Apache Derby* kann als *sicher* bewertet werden. Es befindet sich in der aktiven Weiterentwicklung und da hinter dem Projekt keine kommerzielle Unternehmung steht, kann es auch nicht aufgekauft werden.

Lizenz. *JavaDB* steht unter der *Apache License 2.0*. Software, die unter dieser Lizenz steht, darf in jedem Umfeld frei verwendet, modifiziert und verteilt werden. Programme, die unter Apache-Lizenz stehende Quelltexte verwenden, brauchen selbst nicht unter dieser Lizenz zu stehen.

Dokumentationsgrad. Der Dokumentationsgrad zu JavaDB kann als *gut* bewertet werden. Über die Homepage von Apache Derby [ASF10b] sind das *Referenzhandbuch, eine technische Dokumentation, mehrere Artikel, ein FAQ und die API-Beschreibung in Englisch* verfügbar.
Fragen, die über diese Dokumentation hinausgehen, können in der großen und aktiven Community über mehrere *Mailinglisten* und einem *Chat* gestellt werden.

Zugriffsschnittstelle. Auf eine *JavaDB*-Datenbank kann über *JDBC* zugegriffen werden. Zusätzlich kann eine Instanz von *JavaDB* im gleichen Prozess der *Java-VM* gestartet werden in der auch das zugreifende Programm läuft. Für andere Sprachen wird *ODBC* unterstützt.

Speicherverhalten. Es lassen sich die Größe von diversen Puffern und das Schreibverhalten auf den Hintergrundspeicher konfigurieren.

Serverseitige Caching- und Zugriffsstrategien. *JavaDB* bietet für die Beschleunigung des Zugriffs auf Daten drei Index-Implementationen an (balancierter B+-Baum, Heap und Hashtabelle). Ferner können Daten geclustered werden.

3.5.3. PostgreSQL

Produkt	PostgreSQL
Hersteller	PostgreSQL-Team
Homepage	`http://www.postgresql.org/`
Aktuelle Version	8.4.2
Unterstützte Plattformen	Linux, diverse Unix-Derivate, Windows
Implementierungssprache	C
Maximale Datenbankgröße	unbegrenzt
Maximale Tabellengröße	32 TByte
Zielarchitektur	Client-Server
Datenabfrage über	SQL
Lizenz	BSD
Status	aktiv

Charakterisierung. *PostgreSQL* ist ein freies, objektrelationales Datenbankverwaltungssystem, welches eine verteilte Hochverfügbarkeitslösung zur Verfügung stellen kann. Es ist neben *MySQL* das zweite weithin bekannte große Open-Source-DBMS. Es implementiert den *SQL:92-Standard* vollständig, Teile des *SQL:99-* und Teile des *SQL:2008-Standards*. Zudem unterstützt es Transaktionen nach dem *ACID-Prinzip*. Das komplette Produkt kann in Form von Installationspaketen oder Images von vorinstallierten Varianten auf Live-CDs von der Homepage heruntergeladen werden[7].

Historie. Ebenso wie *MySQL* hat *PostgreSQL* eine lange Geschichte hinter sich. Die Entwicklung von *PostgreSQL* begann Anfang der 1980er Jahre. Es stammt ursprünglich aus dem *Ingres*-Projekt, einer Datenbankentwicklung der *University of California in Berkeley*. Aus den Erkenntnissen des Projekts ging 1985 das *Post-Ingres*-Projekt hervor, welches 1989 als *Postgres* fertiggestellt wurde. 1994 wurde *Postgres* um einen SQL-Interpreter erweitert und die Software als Open-Source unter dem Namen *Postgres95* veröffentlicht. Die Entwicklung von *PostgreSQL* wurde 1996, zusammen mit dem Wechsel auf den heutigen Namen, begonnen. Seitdem wird es fortlaufend weiterentwickelt und seit 1997 von einer Open-Source-Community betreut.
Mittlerweile hat sich um *PostgreSQL* ein Markt gebildet, auf dem diverse kommerziell ausgerichtete Unternehmen ihre Dienstleistungen anbieten.

[7] Für eine vollständige Liste aller Eigenschaften siehe `http://www.postgresql.org/about/featurematrix/`

Zukunftssicherheit. Die Zukunft von *PostgreSQL* kann zum jetzigen Zeitpunkt als *sicher* betrachtet werden. *PostgreSQL* befindet sich aktiv in der Weiterentwicklung und wird in vielen freien und kommerziellen Lösungen eingesetzt. Da hinter dem Projekt keine kommerzielle Unternehmung steht, kann *PostgreSQL* nicht aufgekauft werden.

Dokumentationsgrad. Der Dokumentationsgrad zu PostgreSQL kann als *ausgezeichnet* bewertet werden. Über die Homepage des Produkts sind das *Referenzhandbuch und eine technische Dokumentationen in mehreren Sprachen* verfügbar. Abgerundet wird das Bild über eine lange *Literaturliste* von erschienen Büchern für den Praxiseinsatz. Eines davon ist komplett online verfügbar [Mom01]. Fragen, die über diese Dokumentation hinausgehen, können in der großen und aktiven Community rund um PostgreSQL in diversen *Mailinglisten* und *Chats* gestellt werden.

Zugriffsschnittstelle. Mit Java kann über *JDBC* auf eine *PostgreSQL*-Datenbank zugegriffen werden. Ferner bietet es *ODBC* und Schnittstellen für eine Reihe von weiteren Programmiersprachen.

Lizenz. *PostgreSQL* ist unter der *BSD-Lizenz* verfügbar. Software unter dieser Lizenz darf frei verwendet werden, und es ist erlaubt, sie zu kopieren, zu verändern und zu verbreiten. Einzige Bedingung der Lizenz ist, dass der Copyright-Vermerk des ursprünglichen Programms nicht entfernt werden darf. Es eignet sich somit auch als Vorlage für kommerzielle Produkte, die als Closed-Source-Lösungen angeboten werden.

Speicherverhalten. Es lassen sich die Größe von diversen Puffern und das Schreibverhalten auf den Hintergrundspeicher konfigurieren.

Serverseitige Caching- und Zugriffsstrategien. Die Performanz von Anfragen kann serverseitig durch das Definieren von Indizes verbessert werden. Unterstützt werden unique-, partielle und funktionale Indizes. Mehrere Indizes können zu einem multidimesionalen Index verknüpft werden. Zudem unterstützt *PostgreSQL GiST-Indizes (Generalized Search Tree)*. Es handelt sich dabei um eine baumbasierte Zugriffsmethode, die als Schablone für beliebige Indexmodelle dient. Z. B. können B- und R-Bäume mit einem GiST-Index implementiert werden. Somit können zu spezifischen Daten der Datenbank auch passende Zugriffsmethoden zur Verfügung gestellt werden. Zudem können beliebig viele Indizes pro Tabelle definiert werden.
Neben dem Definieren von Indizes unterstützt *PostgreSQL* auch das *Clustering von Daten*.
Schließlich wird das Tuning der Datenbank in mehreren Artikeln der technischen Dokumentation besprochen.

3.5.4. Begründung der Auswahlentscheidung

Es wurde sich für die angeführten Datenbank-Technologien entschieden, da:

- *MySQL* sich durch einen sehr hohen Verbreitungsgrad auszeichnet,
- *PostgreSQL* ebenfalls weit verbreitet ist - wenn auch nicht so weit wie *MySQL*. Im Gegenzug ist es besser dokumentiert.
- *JavaDB* im Embedded-Modus betrieben werden kann und zusammen mit Java ausgeliefert wird.

Da die Quelltexte der Lösungen frei verfügbar sind, können modifizierte Varianten kompiliert und eingesetzt werden.

Für die gewählten Datenbanksysteme wird zunächst nur angenommen, dass sie die Persistierung von TGraphen mit allen ihren Eigenschaften ermöglichen können. Deshalb wird in den folgenden Kapiteln gezeigt, dass dies zumindest auf konzeptueller Ebene der Fall ist. Der endgültige Beweis wird mit der Implementierung erbracht.

4. Anwendungsdomäne

Nach der Anforderungserhebung und dem Abschluss der Recherche muss eine Lösung konzipiert werden. Als erster Schritt wird dazu die Anwendungsdomäne betrachtet. Die Anwendungsdomäne grenzt das Problemfeld und den Einsatzbereich der zu entwickelnden Lösung ein.
Die Domäne wird durch zwei objektorientierte Datenmodelle beschrieben. Es handelt sich dabei um das Modell des Graphen und das Modell des Graphmetaschemas für *JGraLab*.

4.1. Graph

Das Klassendiagramm in Abbildung 4.1 beschreibt konkrete Graphen. Es wurde aus der Implementation von *JGraLab*[1] abgeleitet.
Im Klassendiagramm wird ein Graph durch die Klasse `Graph` repräsentiert. Er ist über eine global eindeutige Kennung (Eigenschaft `id`) identifizierbar und kennt sein Graphschema (Klasse `Schema`).
Ein Graph besteht aus beliebig vielen Knoten und Kanten, repräsentiert durch die Klassen `Vertex` und `Edge`. Ihre gemeinsamen Eigenschaften werden durch die abstrakte Superklasse `GraphElement` zusammengefasst. Jedes Element eines Graphen trägt eine im Graph eindeutige Kennung, repräsentiert durch das Attribut `id`. Zudem ist jedem Element sein Graph bekannt. Ausgedrückt wird dies durch den Rollennamen `graph` am linken Ende der Assoziation zwischen `Graph` und `GraphElement`.
Graphen, Knoten und Kanten können attributiert werden. Sie haben deshalb die gemeinsame abstrakte Superklasse `AttributedElement`.
Ein Knoten ist Teil der globalen Knotensequenz *Vseq* und eine Kante ist Teil der globalen Kantensequenz *Eseq*. Modelliert werden die Sequenzen durch eine Verkettung ihrer Elemente. Knoten und Kanten können jeweils einen Vorgänger und Nachfolger in den globalen Sequenzen des Graphen haben, ausgedrückt durch entsprechende Rollennamen (`prev` und `next`) an den Enden der reflexiven Assoziationen. Der Graph referenziert jeweils nur das erste und letzte Element einer globalen Liste. Diese Assoziationen sind im Diagramm der Übersichtlichkeit halber ausgeblendet.
Knoten und Kanten sind durch Inzidenzen (Klasse `Incidence`) verbunden. Ein Knoten kann

[1] *JGraLab Carnotaurus Release 1709/67* vom 11. September 2009

mit einem anderen Knoten durch beliebig viele eingehende und ausgehende Kanten verbunden sein. Dazu kennt er die verbindende Inzidenz für die erste und letzte verbundene Kante (Assoziationen zu `Incidence` mit Rollennamen `first` und `last`). Da eine Inzidenz ihren optionalen lokalen Vorgänger und Nachfolger an einem Knoten kennt, sind dem Knoten über diese Verkettung alle weiteren verbundenen Inzidenzen und somit Kanten bekannt. Die Verkettung wird analog zu Knoten und Kanten mit einer Assoziation ausgedrückt, die in der Klasse `Incidence` beginnt und endet und deren Enden mit den Rollennamen `prev` und `next` versehen sind.
Eine Kante ist immer über zwei Inzidenzen an Anfang und Ende mit genau einem Knoten verbunden[2], repräsentiert durch zwei Assoziationen mit Rollennamen `from` und `to` an ihrem Ende.
Graphen, Knoten und Kanten sind typisiert und sind deshalb Instanzen von `GraphClass`, `VertexClass` und `EdgeClass` aus dem Graphmetaschema, welches im nächsten Abschnitt vorgestellt wird.

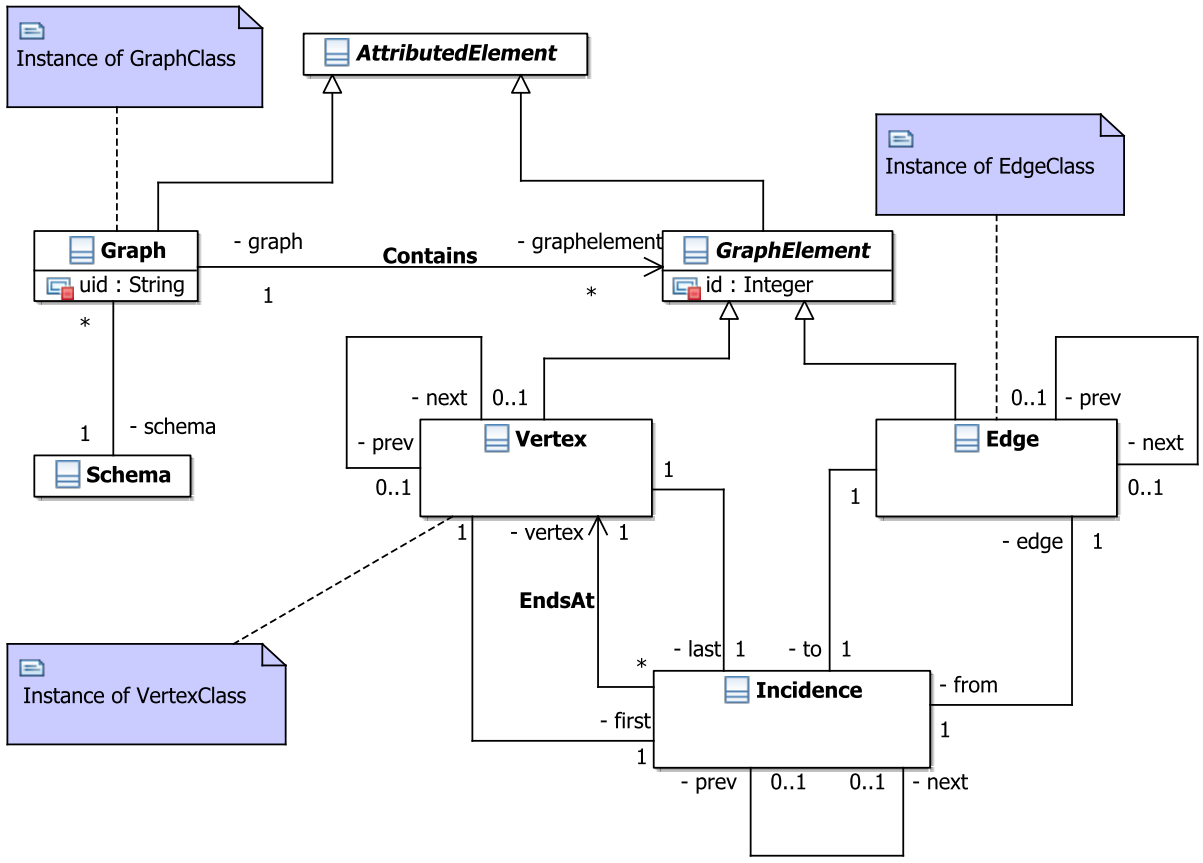

Abbildung 4.1. – Vereinfachte Struktur der Graphebene, hergeleitet aus der Implementation von *JGraLab*. Pfeilspitzen repräsentieren keine Navigierbarkeit, sondern die Leserichtung des Assoziationsnamens.

[2]Beginnt und endet eine Kante am selben Knoten, so liegt eine Schlinge vor.

4.2. Graphmetaschema

Das Graphmetaschema (siehe Abbildung 4.2) beschreibt den Aufbau der möglichen Graphschemata für TGraphen. Ein Graphschema definiert Typen für Graphen, Knoten und Kanten, die Attributierung dieser Typen *sowie die mögliche Anordnung von Knoten und Kanten im Graph*. Im Graphmetaschema wird das Graphschema ebenfalls durch die Klasse `Schema` vertreten.

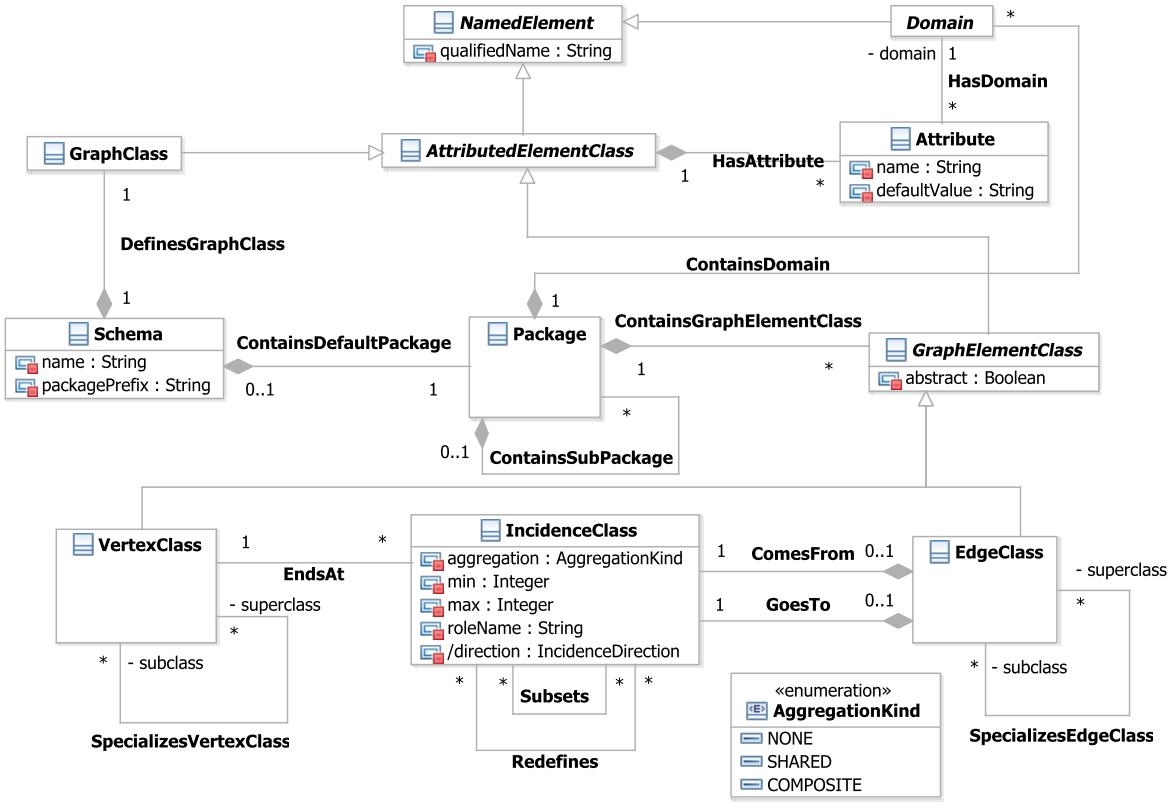

Abbildung 4.2. – Vereinfachte Struktur des Graphmetaschemas aus [BER+08]

Jedes Graphschema ist über seinen qualifizierten Namen, welcher sich aus den Eigenschaften `name` und `packagePrefix` zusammensetzt, identifizierbar. Jedes Graphschema definiert genau eine Graphklasse (Klasse `GraphClass`) und beliebig viele Typen für Knoten und Kanten, repräsentiert durch die Klassen `VertexClass` und `EdgeClass`.
Jeder Knoten- oder Kantentyp kann abstrakt sein, ausgedrückt durch das Attribut `abstract` in der gemeinsamen abstrakten Superklasse `AttributedElementClass`, und kann beliebig viele Attribute besitzen.
Jedes Attribut, repräsentiert durch die Klasse `Attribute`, besitzt einen im Zusammenhang mit dem attributierten Typ eindeutigen Namen und eine Standardbelegung (durch die Eigenschaften `name` und `defaultValue`). Zudem hat jedes Attribut eine Wertdomäne. In Abbildung 4.2 ist jedoch keine Entsprechung vertreten.
Jeder Knoten- oder Kantentyp ist ebenfalls über einen qualifizierten Namen eindeutig identi-

fizierbar, repräsentiert durch die Eigenschaft `qualifiedName` aus der abstrakten Superklasse `NamedElement`.
Knoten- und Kantentypen gehören jeweils einer separaten Typhierarchie an und können mehrere Super- und Subklassen haben. Ausgedrückt wird dies durch die Assoziationen mit Beschriftung `SpecializeVertexClass` und `SpecializesEdgeClass` an `VertexClass` und `EdgeClass`.
Zudem werden definierte Knoten- und Kantentypen in Paketen strukturiert (Klasse `Package`). Ein Paket kann weitere Pakete enthalten.
Zusätzlich werden im Graphschema die möglichen Ausprägungen der Verbindungspunkte zwischen Knoten und Kanten festgelegt. Die Klasse `IncidenceClass` beschreibt im Zusammenhang mit den Assoziationen `ComesFrom` und `GoesTo` zur Klasse `EdgeClass` und der Assoziation `EndsAt` zur Klasse `VertexClass`, dass im Graphschema definiert wird, welche Kanten eines bestimmten Kantentyps zwischen welchen Knoten bestimmter Knotentypen existieren dürfen.
Ferner wird durch die Eigenschaften der Klasse `IncidenceClass` ausgedrückt, dass gleichzeitig die möglichen Ausprägungen der Enden von Kanten festgelegt werden. Die Klasse legt mit Attribut `roleName` fest, welche Rolle der verbundene Knoten inne hat. Attribut `direction` legt fest ob es sich um eine *eingehende* oder *ausgehende* Kante handelt. Die Attribute `min` und `max` definieren Unter- und Obergrenze für die Anzahl der inzidenten Kanten eines bestimmten Typs. Mit `aggregation` wird festgelegt ob das Kantenende eine *einfache Assoziation*, *Aggregation* oder *Komposition* bildet.

Im nächsten Kapitel werden aus den objektorientierten Modellen entsprechende relationale Schemata abgeleitet.

5. Ableitung relationaler Schemata

Relationale Schemata können als Grundlage für physikalische Datenbankschemata dienen. Deshalb werden in diesem Kapitel aus den Modellen der Anwendungsdomäne relationale Schemata, welche die Grundlage für physikalische Datenbankschemata bilden, abgeleitet. Diese sollen wiederum das Persistieren von Graphen in relationalen und objektrelationalen Datenbanken ermöglichen.
Zunächst werden die relevanten Anforderungen präzisiert.

5.1. Relevante Anforderungen

Neben der primären Anforderung, dass die Lösung das Persistieren von TGraphen in Datenbanken ermöglichen soll, sind für die aktuelle Betrachtung weitere Anforderungen von direkter und indirekter Relevanz.
Bei der Persistierung sollen zusätzlich die Eigenschaften von TGraphen erhalten bleiben. Aus der Definition von TGraphen gehen die Implikationen dieser Anforderung hervor. TGraphen sind attributierte, gerichtete, angeordnete und typisierte Graphen. Diese Eigenschaften müssen erhalten bleiben.
Zusätzlich muss zu jedem TGraph auch sein Graphschema bekannt sein, denn *ein TGraph gehört immer zu genau einem Graphschema*. Dazu soll so viel wie nötig und so wenig wie möglich an Graphschemainformationen in der Datenbank persistiert werden.
Eine weitere Anforderung impliziert, dass die Lösung die *spätere Persistierung von DHH-TGraphen nicht behindern* oder gar unmöglich machen soll. *DHH-TGraphen* sind *verteilte hierarchische Hyper-TGraphen*. Diese Graphen sollen neben *Hyperkanten* [Ber73] und einem *Hierarchiekonzept* zwischen den Graphelementen [Lit92] auch die mögliche Verteilung des Graphen an verschiedenen Orten unterstützen.
Ferner sollen beliebig viele Graphen in einer Datenbank persistiert werden können.
Alle angeführten Anforderungen und Präzisierungen fließen in den nächsten Abschnitten mit ein in die Entwicklung der Relationen.

5.2. Relationen

In den folgenden Abschnitten werden die bereits vorgestellten objektorientierten Modelle auf eine Menge von Relationen abgebildet. Die erhaltenen Relationen bilden dann ein relationales

Modell der Anwendungsdomäne.
Allgemein beschreibt eine Relation eine Beziehung, die zwischen Dingen bestehen kann. Formal ausgedrückt beschreibt eine Relation eine Menge von Tupeln. Ein Tupel repräsentiert eine konkrete Ausprägung einer Relation [KE09]. Aufgeschrieben besteht eine Relation aus einem Namen und einer Menge von Attributen, die jeweils einen Typ haben. Da für die Notation von Relationen unterschiedliche Schreibweisen existieren, wird die Schreibweise für diese Arbeit auf folgende Art[1] festgelegt:

```
Relation := <Relationsname> '(' <Attribute> ')'

Attribute :=
   <Schlüsselattribut> {',' <Schlüsselattribut>} {',' <Attribut>}

Schlüsselattribut := <Attributname> : <Typ>

Attribut := <Attributname> : <Typ>

Typ := 'Integer' | 'Long' | 'String' | 'Direction'
```

Eine Relation beginnt mit einem Namen. Darauf folgend werden in Klammern die Attribute der Relation aufgeführt. Zwei Attribute sind voneinander durch ein Komma getrennt.
Zunächst wird mindestens ein Schlüsselattribut angeführt, dann können weitere Schlüsselattribute folgen. Anschließend folgen beliebig viele Nichtschlüsselattribute.
Jedes Attribut und Schlüsselattribut besteht aus einem Attributnamen und einem Typ. Beide werden durch einen Doppelpunkt voneinander getrennt. Der Name eines Schlüsselattributs wird im Gegensatz zum Namen eines herkömmlichen Attributs unterstrichen. Namen sind frei wählbar. Relationennamen müssen in einem relationalen Schema eindeutig sein. Attributnamen müssen in einer Relation eindeutig sein.
Als mögliche Typen der Attribute werden bewusst nur jene vier aufgeführt, die im Rahmen dieser Arbeit vorkommen.
Aus den Relationen resultieren die Tabellen der Datenbank und aus den Attributen die Spalten der Tabellen. Ein Tupel stellt dann eine Zeile der Tabelle dar, in der die Spalten mit konkreten Werten belegt sind. Die Zeile ist über die Belegung der Spalten, die aus den Schlüsselattributen resultieren, eindeutig identifizierbar.

5.3. Das generische relationale Schema

In diesem Abschnitt wird die Entwicklung eines generischen relationalen Schemas für die Persistierung von TGraphen in Datenbanken beschrieben. Generisch bedeutet in diesem Zusam-

[1] in EBNF

menhang, dass in einer Datenbank, die diesem Schema entspricht, alle Ausprägungen von beliebigen TGraphen persistiert werden können, unabhängig von deren Graphschema.
Dazu werden die objektorientierten Modelle der Anwendungsdomäne durch Anwendung von Abbildungsvorschriften auf Relationen abgebildet. Es wird jedoch ein Anteil der Relationen durch die bisherige konkrete Implementation von *JGraLab* bestimmt.

5.3.1. Abbildungsvorschriften

Durch die Abbildung von Elementen aus den objektorientierten Modellen der Anwendungsdomäne auf Relationen ist es das Ziel, ein relationales Schema zu erhalten, welches:

- nur wenige (redundante) Informationen in der Datenbank zulässt, da TGraphen sehr groß werden können, und
- nur wenige *relationale Verbundoperationen* [2] in der Datenbank hervorruft, da in der Regel die Geschwindigkeit einer Lösung unter zu vielen Verbundoperationen leidet.

Eine Verbundoperation entsteht, wenn mehr als nur eine Relation angefasst werden muss, um alle benötigten Daten abzurufen. Im generischen relationalen Schema wird eine Verbundoperation in Kauf genommen, wenn dadurch mehrere Redundanzen über die Menge der Tupel vermieden werden kann.
Um dies zu erreichen, wird nach folgenden Vorschriften bei der Abbildung des *objektorientierten Modells des Graphen* auf eine Menge Relationen vorgegangen:

- jede nicht abstrakte Klasse wird auf eine Relation abgebildet,
- die Attribute einer Klasse werden auf entsprechende Attribute der Relation abgebildet,
- es werden keine Generalisierungen abgebildet,
- ein Rollenname am Ende einer Assoziation wird auf ein Attribut der Relation abgebildet, die aus der Klasse resultiert, die am gegenüberliegenden Ende der Assoziation steht,
- pro Assoziation wird maximal ein Rollenname abgebildet,

Aus dem Graphmetaschema werden nur jene Elemente abgebildet, auf die sich im Modell des Graphen bezogen wird.
Die Abbildung von Generalisierungsbeziehungen würde im relationalen Modell zu weniger Attributen und kürzeren Relationen führen. Dadurch ließen sich jedoch keine weiteren Informationsminimierungen in der Datenbank umsetzen. Es würden nur mehr (vermeidbare) Verbundoperationen nötig werden.
Grundsätzlich finden sich durch Anwendung der Abbildungsvorschriften alle Eigenschaften von TGraphen im relationalen Datenmodell wieder. Die Anordnung und Reihenfolge von Graphelementen wird dennoch auf eine andere Weise abgebildet.

[2] Auch *Joins* genannt.

5.3.2. Anordnung der Elemente im Graph

Die Anordnung der Elemente im Graph wird durch die Reihenfolge der Inzidenzen in den Inzidenzlisten der Knoten realisiert. Zusätzlich haben Knoten und Kanten eine Reihenfolge in den globalen Knoten- und Kantensequenzen des Graphen. Im objektorientierten Modell des Graphen werden die Reihenfolgen in den Listen durch die Rollenname `prev` und `next` an der Klasse `Incidence`, bzw. `Vertex` und `Edge` ausgedrückt.
JGraLab implementiert diese Struktur bisher als doppelt verkettete Liste, da diese Lösung sehr effizient beim Traversieren von Graphen ist, die im physischen Hauptspeicher eines Rechners vorliegen [Kah06].
Sollen jedoch nur die jeweils gerade relevanten Teile eines Graphen im Hauptspeicher vorliegen, so kann diese Implementation nicht verwendet werden, da sie den Graph immer *komplett* in den Hauptspeicher lädt. Zudem erweist sich die Nachbildung der verketteten Liste in der Datenbank bei genauer Betrachtung nicht als die günstigste Lösung.
Ein günstigeres Verhalten wird durch eine Nummerierung der Elemente einer Liste erzielt. Jedes Element bekommt eine eindeutige Sequenznummer, die die Reihenfolge des Elements in der Liste abbildet.[3]
In den folgenden Abschnitten ersetzt deshalb in den aufgeführten Relationen das Attribut `sequenceNumber` die entsprechenden Attribute für die Rollennamen `prev` und `next`.

5.3.3. Graphschema

Jeder Graph gehört zu einem Graphschema. Das Graphschema muss komplett verfügbar sein, damit ein Graph geladen werden kann. *JGraLab* bringt bereits Funktionalitäten zum Lesen eines serialisierten Graphschemas (in TG-Notation[4]) aus einem Eingabestrom mit. Es genügt deshalb, das Graphschema in seiner serialisierten Form in der Datenbank zu speichern. Dadurch entfällt die Notwendigkeit für eine Reihe von Relationen, die zur feingranularen Aufnahme eines Graphschemas nötig wären.

Somit wird nur die Klasse `Schema` des Graphmetaschemas auf die entsprechende Relation abgebildet:

```
Schema(
   schemaId: Integer,
   prefix: String,
   name: String,
   definition: String
)
```

[3] Für eine detaillierte Betrachtung siehe Anhang B. Diese setzt jedoch die genaue Kenntnis des generischen relationalen Schemas voraus.
[4] Sprache von *JGraLab* zum Beschreiben von Graphen und ihren Schemata.

Die Attribute der Relation enthalten:

- `schemaId`: den Primärschlüssel des Graphschemas,
- `prefix` und `name`: den Namenspräfix entsprechend der *Package-Deklaration* und den Namen des Graphschemas sowie
- `serializedDefinition`: die serialisierte Definition des Graphschemas in TG-Notation.

Als Primärschlüssel eines Graphschemas können auch Präfix und Name dienen. Um jedoch bei der Referenzierung von Graphschemata in anderen Relationen Redundanzen zu minimieren, wurde eine numerische Repräsentation gewählt.
Enthält eine Datenbank nur einen Graph oder mehrere Graphen, die zum gleichen Schema gehören, so enthält die aus der Relation resultierende Tabelle nur einen Eintrag, nämlich das einzige Graphschema.
Um mit zugehörigen TGraphen arbeiten zu können, *muss zuvor aus der serialisierten Definition das Graphschema geladen werden*. Um Informationen über ein Element aus dem Graphschema abrufen zu können, muss der Typ des Elements bekannt sein.
Im nächsten Abschnitt wird eine Relation vorgestellt, die die verwendeten Typen aufnehmen kann.

5.3.4. Typen

Jedes typisierte Element im Graph muss seinen Typ kennen, um weitere nötige Informationen, die für den Typ im Graphschema definiert wurden (z. B. Attributierung) abzurufen. Dazu müssen jene Typen gespeichert werden, die auch Verwendung im Graph finden. Folgende Relation kann die verwendeten Typen aufnehmen, die die Klasse `AttributedElementClass` aus dem Graphmetaschema spezialisieren:

```
Type(
  typeId: Integer,
  qualifiedName: String,
  schemaId: Integer
)
```

Die Attribute der Relation enthalten:

- `typeId`: den Primärschlüssel des Typs,
- `qualifiedName`: den Namen des Typs und
- `schemaId`: den Fremdschlüssel des Graphschemas aus Relation `Schema`, zu dem der Typ gehört.

Um Redundanzen in referenzierenden Relationen zu minimieren, wird statt der Kombination der Attribute `qualifiedName` und `schemaId`, die kürzere nummerische `typeId` als Primärschlüssel eingeführt.
Alle nötigen Informationen zu einem Typ können über seinen qualifizierten Namen beim Graphschema abgerufen werden.

5.3.5. Graph

Ein konkreter TGraph als Instanz eines Graphschemas kann durch folgende Relation aufgenommen werden:

```
Graph(
  gId: Integer,
  id: String,
  version: Long,
  vSeqVersion: Long,
  eSeqVersion: Long,
  typeId: Integer
)
```

Die Attribute der Relation enthalten:

- `gId`: den Primärschlüssel des Graphen und
- `id`: die *id des Graphen*, welche durch den Benutzer vorgegeben wird. Nach Außen hin ist der Graph nicht unter seinem Primärschlüssel, sondern unter dieser *id* bekannt.
- `version`: die Version des Graphen,
- `vSeqVersion`: Version der globalen Knotensequenz *Vseq*,
- `eSeqVersion`: Version der globalen Kantensequenz *Eseq*,
- `typeId`: den Fremdschlüssel einer Graphklasse aus Relation `Type`.

Um Redundanzen in referenzierenden Relationen zu minimieren, wird statt der `id`, die als String sehr lang werden darf, die kürzere nummerische `gId` als Primärschlüssel eingeführt.
Die Versionen der globalen Knoten- und Kantensequenz müssen aufgrund der Implementation von *Failfast-Iteratoren* in *JGraLab* persistiert werden.
Im nächsten Abschnitt werden Relationen zum Speichern von Knoten und Kanten eines Graphen vorgestellt.

5.3.6. Knoten und Kanten

Die Knoten eines Graphen (im Modell `Vertex`) werden durch folgende Relation aufgenommen:

```
Vertex(
  vId: Integer, gId: Integer,
  typeId: Integer,
  sequenceNumber: Long,
  lambdaSeqVersion: Long
)
```

Die Attribute der Relation enthalten:

- `vId`: die Id des Knotens im Graph,
- `gId`: den Fremdschlüssel des Graphen aus Relation `Graph`, zu dem der Knoten gehört,
- `typeId`: den Fremdschlüssel des Kontentyps aus Relation `ReferencedType`,
- `sequenceNumber`: die Sequenznummer des Knotens, die seine Position in der globalen Knotenliste des Graphen abbildet, und
- `lambdaSeqVersion`: die Version der Inzidenzliste Λ_{seq}. Diese muss ebenfalls aufgrund der Implementation von *Failfast-Iteratoren* in *JGraLab* persistiert werden.

Die Kanten des Graphen (im Modell `Edge`) werden durch folgende Relation aufgenommen:

```
Edge(
  eId: Integer, gId: Integer,
  typeId: Integer,
  sequenceNumber: Long
)
```

Die Attribute der Relation enthalten:

- `eId`: die Id der Kante im Graph,
- `gId`: den Fremdschlüssel des Graphen aus Relation `Graph`, zu dem die Kante gehört,
- `typeId`: den Fremdschlüssel eines Kantentyps aus Relation `ReferencedType`,
- `sequenceNumber`: die Sequenznummer der Kante, die ihre Position in der globalen Kantenliste des Graphen abbildet.

Da beliebig viele Graphen persistiert werden sollen, kann nicht garantiert werden, dass `vId` und `eId` immer eindeutig sind. Somit ergibt sich der natürliche Schlüssel eines Knotens aus

der Kombination von `vId` und `gId` und der natürliche Schlüssel einer Kante aus `eId` und `gId`. Da die `vId` von Knoten und die `eId` von Kanten nicht disjunkt sind, wird das Attribut nicht in eine gemeinsame Relation herausfaktorisiert. Auch die Attribute `sequenceNumber` werden nicht in eine gemeinsame Relation herausfaktorisiert, da es sich dabei um Positionen der Elemente in unterschiedlichen Listen handelt.
Die Attribute `typeId` und `gId` könnten in eine gemeinsame Relation ausgelagert werden. Dies führt jedoch zu keiner Reduzierung von Redundanzen und kostet eine zusätzliche Verbundoperation.
Im nächsten Abschnitt wird eine Relation für Inzidenzen vorgestellt.

5.3.7. Inzidenzen

Knoten und Kanten sind durch Inzidenzen miteinander verbunden. Diese werden im objektorientierten Modell durch die Klasse `Incidence` repräsentiert. Folgende Relation nimmt diese auf:

```
Incidence(
  eId: Integer, gId: Integer, direction: Direction,
  vId: Integer,
  sequenceNumber: Long
)
```

Die Attribute der Relation enthalten:

- `eId`: den Fremdschlüssel der Kante aus Relation `Edge`, die durch die Inzidenz mit einem Knoten verbunden ist,
- `gId`: den Fremdschlüssel des Graphen aus Relation `Graph`, zu dem die Inzidenz gehört,
- `direction`: die Richtung der inzidenten Kante des Knotens (IN oder OUT), und
- `vId`: den Fremdschlüssel des Knotens aus Relation `Vertex`, der durch die Inzidenz mit einer Kante verbunden ist,
- `sequenceNumber`: die Sequenznummer der Inzidenz, die ihre Position in der Inzidenzliste des verbundenen Knotens abbildet.

Der Primärschlüssel einer Inzidenz ergibt sich aus der Kombination von `eId`, `gId` und `direction`. Zuletzt müssen die Belegungen von Attributen abgebildet werden.

5.3.8. Attribute

JGraLab bringt bereits Funktionalitäten zum Lesen von serialisierten Attributbelegungen mit. Es genügt deshalb, Attributbelegungen in ihrer serialisierten Form in der Datenbank zu spei-

chern. Die Belegung der Attribute von Graphen, Knoten und Kanten werden durch folgende Relationen aufgenommen:

```
GraphAttributeValue(
  gId: Integer, attributeId: Integer,
  value: String
)

VertexAttributeValue(
  vId: Integer, gId: Integer, attributeId: Integer,
  value: String
)

EdgeAttributeValue(
  eId: Integer, gId: Integer, attributeId: Integer,
  value: String
)
```

Die Attribute der Relationen enthalten:

- `gId, vId, eId`: den Fremdschlüssel des attributierten Elements, welches das belegte Attribut enthält,
- `attributeId`: den Fremdschlüssel des Attributs aus Relation `Attribute` (die als nächstes definiert wird), und
- `value`: den serialisierten Wert, mit dem das Attribut belegt ist.

Da die Primärschlüssel von Graph, Knoten und Kanten nicht disjunkt sind, können die Attributbelegungen nicht in einer gemeinsamen Relation gespeichert werden.
Über das attributierte Element und den Namen des Attributs kann das Attribut selbst und somit die Attributdomäne aus dem Graphschema ermittelt werden. Ist die Attributdomäne bekannt, so kann die serialisierte Belegung übersetzt werden.
Um Redundanzen durch die Namen der Attribute zu minimieren, werden diese in folgende Relation ausgelagert:

```
Attribute(
  attributeId: Integer,
  name: String
 )
```

Die Attribute der Relation enthalten:

- `attributeId`: den Primärschlüssel des Attributs und

- `name`: den Namen des Attributs.

Die Redundanzminimierung wird über die kürzere nummerische `attributeId` als Primärschlüssel erreicht.

5.3.9. Aufwandsbetrachtung

Aus dem generischen relationalen Modell wird in der Implementierungsphase ein physikalisches Datenbankschema abgeleitet. In einer Datenbank, die diesem Schema entspricht, können TGraphen persistiert werden.
An dieser Stelle kann bereits eine vorläufige Aufwandsbetrachtung vorgenommen werden. Um ein komplettes Element aus einer Datenbank abzurufen, müssen mehrere Relationen angefasst werden. Tabelle 5.1 fasst zusammen, welche Relationen jeweils für das Abrufen von Graph, Knoten und Kante relevant sind. Diese resultieren zum Teil in entsprechenden Verbundoperationen, wenn konkrete Elemente aus einer Datenbank abgerufen werden.

Graph	**Knoten**	**Kante**
`Graph`, `Type`, `GraphAttributeValue`, `Attribute`, `Vertex`, `Edge`	`Vertex`, `Type`, `VertexAttributeValue`, `Attribute`, `Incidence`	`Edge`, `Type`, `EdgeAttributeValue`, `Attribute`, `Incidence`
Summe = 6	Summe = 5	Summe = 5

Tabelle 5.1. – Relationen, die zum Abrufen des jeweiligen Elements anzufassen sind.

Durch ein graphspezifisches relationales Schema können die anzufassenden Relationen pro abzurufendes Element reduziert werden.

5.4. Graphschemaspezifische relationale Schemata

In diesem Abschnitt wird eine allgemeine Vorgehensweise zur Herleitung von graphschemaspezifischen relationalen Modellen aus gegebenen Graphschemata skizziert. Eine Unterstützung von graphschemaspezifischen relationalen Schemata durch die Implementation wurde im Rahmen der Arbeit jedoch nicht umgesetzt.
Das generische Schema ist für alle Ausprägungen von Graphschemata und Graphen konzipiert. Graphschemaspezifisch bedeutet hingegen, dass in einer Datenbank, die diesem Modell entspricht, nur alle Ausprägungen von TGraphen, die einem *gemeinsamen* Graphschema entsprechen, persistiert werden können. Es wird sich somit auf ein Graphschema pro Datenbank

beschränkt.
Im Vergleich zum generischen relationalen Schema soll der Zugriff auf Elemente eines gewissen Typs schneller erfolgen.
Ausgehend vom generischen Schema sind eine Reihe von Optimierungen für graphschemaspezifische relationale Schemata möglich.

5.4.1. Gemeinsame Relationen

Aus dem generische relationalen Schema finden sich die Relationen `Schema` und `Type` fast unverändert auch in den graphschemaspezifischen Fassungen wieder. Die Relation `Incidence` wird jeweils unverändert übernommen. Die graphschemaspezifischen Schemata können diese drei Relationen enthalten.

Da ein graphschemaspezifisches relationales Schema nur für Graphen eines Graphschemas ausgelegt sein muss, fällt die Relation *Schema* schlanker als im generischen relationalen Schema aus. Zunächst kann das Graphschema auf die gleiche Weise wie im generischen Datenbankschema behandelt werden, da sich durch die Speicherung des Graphschemas in serialisierter Form die Komplexität des resultierenden Graphschemas erheblich reduziert. So entspricht die Relation fast jener aus dem generischen Datenbankschema:

```
Schema(
   serializedDefinition: String
)
```

Wie auch beim generischen Datenbankschema muss die serialisierte Definition des Graphschemas geladen werden, bevor mit den zugehörigen TGraphen gearbeitet werden kann.
Da die Relation nur ein einziges Graphschema enthalten wird, benötigt es kein künstliches Schlüsselattribut, da es in anderen Relationen nicht referenziert wird und nur einmal abgerufen werden muss[5].
Durch eine schlanke Relation für das einzige Graphschema ergibt sich gegenüber dem generischen relationalen Schema noch keine ausschlaggebenden Vorteile. Diese ergeben sich vielmehr aus einem anderen Umgang mit Typen, Attributen und Attributbelegungen.

Jedes typisierte Element im Graph muss weiterhin seinen Typ kennen. Im folgenden Abschnitt wird dies durch die Anwendung von verschiedenen Abbildungstechniken realisiert. Zwei der aufgeführten Techniken benötigen weiterhin eine separate Relation zur Persistierung der Typnamen.
Folgende Relation nimmt die Bezeichner auf:

[5] Beim ersten Laden eines Graphen (um einer möglichen Implementierung bereits vorweg zu greifen)

```
Type(
  typeId: Integer,
  qualifiedName: String
)
```

Die Attribute der Relation enthalten:

- `typeId`: den Primärschlüssel des Typs und
- `qualifiedName`: den Namen des Typs.

Um ebenfalls Redundanzen in referenzierenden Relationen zu minimieren, wird die kürzere numerische `typeId` als Primärschlüssel eingeführt.
Im Vergleich zum generischen Datenbankschema entfällt eine Referenz auf das Graphschema, welches den Typ definiert.
Alle nötigen Informationen zu einem Typ können nun über seinen qualifizierten Namen beim Graphschema abgerufen werden.
Konkret brauchen in dieser Relation nur verwendete Knoten- und Kantentypen gespeichert werden, da der Typ der Graphen bereits (implizit) bekannt ist.
In den folgenden Abschnitten wird skizziert wie die Relationen für Graphen, Knoten, Kanten und Attribute aus dem Graphschema abgeleitet werden können.

5.4.2. Abbildungstechniken für Typen und Attribute

In einem Graphschema werden Typen für Graphen, Knoten und Kanten definiert. Zusätzlich können die definierten Typen attributiert werden.
Für die spezifische Abbildung von Typen auf Relationen können drei verschiedene Verfahren angewendet werden.

Relation pro Typhierarchie. Mit dem Verfahren *Relation pro Vererbungshierarchie* werden alle Klassen, die der gleichen Vererbungshierarchie angehören auf *eine* Relation abgebildet. Die Relation bekommt den Namen der Wurzelklasse der Vererbungshierarchie.
Zusätzlich können alle Attribute der Klassen in der Vererbungshierarchie auf entsprechende Attribute in der Relation abgebildet werden. Somit können alle Attributbelegungen in einem Tupel abgelegt werden.
Mit dieser Technik lässt sich der Zugriff auf die Werte der Attribute einer Instanz effizienter gestalten. Es können jedoch viele Tupel mit unbelegten Attributen in der Relation auftreten.
Durch Anwendung dieser Technik auf konkrete Graphschemata erhält man immer drei Relationen `Graph`, `Vertex` und `Edge`. Die Definition von Typen für Knoten und Kanten resultiert in zwei Vererbungshierarchien. Aus diesen Hierarchien werden `Vertex` und `Edge` abgeleitet.

Aus der Definition der einzigen Graphklasse in einem Graphschema wird die Relation `Graph` abgeleitet.
Auf zusätzliche Relationen, die die Attributbelegungen aufnehmen, kann verzichtet werden, da die drei Relationen jeweils alle Attributbelegungen in entsprechenden Relationenattributen enthalten.
In den Relationen muss jedoch immer noch zwischen den Typen der enthaltenen Elemente unterschieden werden. Dazu ist ein weiteres Attribut nötig, welches den konkreten Typ eines Elements anzeigt. Es wird somit weiterhin eine Relation `Type` benötigt. In dieser muss das definierende Graphschema nicht explizit referenziert werden, da davon ausgegangen werden kann, dass sich auf ein einziges bezogen wird.

Relation pro Typ. Mit dem Verfahren *Relation pro Typ* wird *jeder im Graphschema definierte Typ auf eine Relation abgebildet*. Eine Relation wird entsprechend dem abgebildeten Typ benannt. Dabei werden nur Attribute, die unmittelbar in der Klasse definiert sind, auf entsprechende Attribute in der Relation abgebildet. Im Gegensatz zum Verfahren *Tabelle pro Vererbungshierarchie* fließen von Basisklassen geerbte Attribute nicht in die Relation der spezialisierenden Klasse, sondern in die Relationen der Basisklassen mit ein. Somit ist eine Instanz einer Klasse nicht nur mit einem Eintrag in der entsprechenden Relation, sondern auch in den Relationen der Basisklassen vertreten.
Mit dieser Technik lässt sich das Auftreten von vielen unbelegten Attributen vermeiden. Um jedoch die Werte aller Attribute einer Instanz einer abgebildeten Klasse zu erhalten, müssen *alle* aus den Basisklassen resultierenden Relationen angefasst werden. Dies führt wiederum zu entsprechenden Verbundoperationen.
Durch Anwendung dieser Technik auf konkrete Graphschemata erhält man immer so viele Relationen wie Typen im Graphschema definiert werden. Auf zusätzliche Relationen, welche die Attributbelegungen aufnehmen, kann ebenfalls verzichtet werden, da die erhaltenen Relationen jeweils alle Attributbelegungen in entsprechenden Relationenattributen enthalten.
Eine zusätzliche Relation `Type` zur Unterscheidung der Typen wird nicht mehr benötigt.

Relation pro abstraktem Typ. Das Verfahren *Relation pro abstraktem Typ* stellt einen Kompromiss zwischen den bereits vorgestellten Techniken dar. Es werden alle Typen, die von einem *gemeinsamen abstrakten Typen* erben auf eine Relation abgebildet. Die Relation wird entsprechend dem abstrakten Typ benannt. Die Attribute aller Subtypen werden auf entsprechende Attribute der Relation abgebildet.
Vorausgesetzt wird das mindestens die Wurzelelemente der Typhierarchien abstrakt sind. Mit diesem Verfahren sollen weniger unbelegte Attribute als mit dem Verfahren *Relation pro Typhierarchie* auftreten und weniger Verbundoperationen als mit dem Verfahren *Relation pro Typ* notwendig sein.
Durch Anwendung dieser Technik auf konkrete Graphschemata erhält man immer so viele Re-

lationen wie abstrakte Typen im Graphschema definiert werden. Sind im ungünstigsten Fall keine abstrakten Typen definiert oder nur die Wurzelelemente der Typhierarchien abstrakt, so verhält es sich wie das Verfahren *Relation pro Typhierarchie*.
Auf eine Relationen, die die Attributbelegungen aufnehmen, kann ebenfalls verzichtet werden. Zur Unterscheidung der Typen ist jedoch zusätzlich die Relation `Type` nötig.

Alle vorgestellten Verfahren *unterstützen das Konzept der Mehrfachvererbung*, welches in TGraphschemata erlaubt ist.

5.4.3. Native Wertdomänen

Eine höhere Performanz kann in der Regel durch Abbildung der Attributdomänen auf entsprechende Datentypen und Konzepte der Datenbank erzielt werden. Somit kann für einen Teil der Attribute die Serialisierung ihrer Belegung in einen String entfallen.
Die verwendete Datenbank-Technologien bieten Datentypen entsprechend dem SQL:99-Standard an. Somit können folgende Attributdomänen, die *primitive Typen* repräsentieren, auf entsprechende Datentypen der Datenbank abgebildet werden:

- `BooleanDomain` auf `BOOLEAN` oder `BIT`,
- `IntegerDomain` auf `INT`,
- `LongDomain` auf `BIGINT`,
- `DoubleDomain` auf `DOUBLE PRECISION`,
- `StringDomain` auf `NVARCHAR` oder `TEXT`, und
- `EnumDomain` auf `ENUM`.

Folgende Attributdomänen können nicht unmittelbar auf entsprechende Datentypen abgebildet werden, da sie zusammengesetzte Attributdomänen repräsentieren. Sie können jedoch auf separate Relationen abgebildet werden, sofern sie sich ausschließlich aus Attributdomänen zusammensetzen, sie primitive Typen repräsentieren.

- `RecordDomain`
- `ListDomain`
- `SetDomain`
- `MapDomain`

Setzt sich eine Attributdomäne nicht ausschließlich aus primitiven Attributdomänen zusammen, so müssen ihre Ausprägungen weiterhin *in einen String serialisiert werden*.

5.4.4. Aufwandsbetrachtung

Grundsätzlich kann der Aufwand zum Abrufen eines Elements geringer ausfallen als es beim generischen relationalen Schema der Fall ist (siehe Tabelle 5.2). Dort sind jeweils 6 Relationen

für den Graphen und jeweils 5 Relationen pro abgerufenem Knoten bzw. Kante anzufassen. Bei der Anwendung aller angeführter Verfahren entfällt die Notwendigkeit separater Relationen für die Aufnahme der Attribute und ihrer Belegung.

Durch Anwendung des Verfahrens *Relation pro Typhierarchie* entstehen graphschemaspezifische relationale Schemata, die dem generischen relationalen Schema noch sehr ähnlich sind. Sie führen ähnliche Relationen für Graphschema, Graphen, Knoten, Kanten, Inzidenzen und Typen, kommen insgesamt aber mit weniger Relationen aus. Es müssen auch weniger Relationen für das Abrufen eines Elements angefasst werden. Der Aufwand pro Element ist konstant.

Durch die Anwendung des Verfahrens *Relation pro Typ* entstehen graphschemaspezifische relationale Schemata die für jeden im Graphschema definierten Typ eine Relation enthalten. Der Aufwand für den Abruf eines Elements hängt somit von der Tiefe m des Elementtyps in seiner Vererbungshierarchie ab. Die Vererbungshierarchie muss vom Typ aus bis zum Wurzeltyp hinaufgestiegen werden, um alle Attribute und Belegungen des abzurufenden Elements zu sammeln. Beträgt die Tiefe mehr als 4 Ebenen für Knoten bzw. Kanten, so fällt der Aufwand höher als beim generischen relationalen Schema aus.
Für das Verfahren *Relation pro abstraktem Typ* verhält es sich grundsätzlich analog. Jedoch fällt der Aufwand immer um eine Relation höher aus als beim Verfahren *Relation pro Typ*.

Verfahren	**Graph**	**Knoten**	**Kante**
Relation pro Typhierarchie	`Graph`, `Vertex`, `Edge`	`Vertex`, `Type`, `Incidence`	`Edge`, `Type`, `Incidence`
Summe	3	3	3
Relation pro Typ	`Graph`, `Vertex`, `Edge`	m Relationen für `Vertex`, `Incidence`	m Relationen für `Edge`, `Incidence`
Summe	3	$m + 1$	$m + 1$
Relation pro abstraktem Typ	`Graph`, `Vertex`, `Edge`	m Relationen für `Vertex`, `Type`, `Incidence`	m Relationen für `Edge`, `Type`, `Incidence`
Summe	3	$m + 2$	$m + 2$

Tabelle 5.2. – Relationen nach angewendeter Abbildungstechnik, die zum Abrufen des jeweiligen Elements anzufassen sind.

Kombiniert man ein Verfahren mit der Abbildung der Attributdomänen auf entsprechende Datentypen und Konzepte der Datenbank, so sollte die Implementation performanter als die Implementation des generischen relationalen Schemas sein.

5.5. Anpassung für DHH-TGraphen

Zur Unterstützung *verteilter hierarchischer Hyper-TGraphen* müssen die relationalen Modelle angepasst werden. Die dargelegte Erweiterung wird jedoch nicht in der Implementierungsphase umgesetzt. Es soll nur gezeigt werden, dass die Lösung für die Unterstützung von DHH-TGraphen erweitert werden kann.
Die Eigenschaft der *Verteiltheit* schlägt sich nicht in Form eines konkreten Elements im Graph nieder. Folglich müssen die relationalen Schemata dafür nicht angepasst werden.

Die Eigenschaft der *Hierarchie* wird in DHH-TGraphen über eine Attributierung der Graphelemente abgebildet. Jedes Graphelement referenziert sein *Elternelement σ durch ein entsprechendes Attribut*. Für diese Eigenschaft muss das generische relationale Modell ebenfalls nicht angepasst werden, denn es lässt bereits die Aufnahme jeglicher Attributierungen von Graphelementen zu. Voraussetzung ist, dass die Wertbelegung durch *JGraLab* serialisiert werden kann.

Eine *Hyperkante* kann im Gegensatz zur konventionellen Kante mehr als zwei Knoten verbinden. Sie kann somit an mehr als zwei Inzidenzen beteiligt sein. Realisiert werden Hyperkanten über eine *Erweiterung des herkömmlichen Kantenkonzepts um eine Inzidenzliste*.

Knoten am Anfang oder am Ende einer Kante können eine Rolle einnehmen. Diese Rollen werden mit Rollennamen beschrieben und im Graphschema festgelegt. Bisher hängt die Rolle eines Knotens davon ab, ob er am Anfang (ausgehende Kantenrichtung) oder am Ende der Kante (eingehende Kantenrichtung) liegt. Mit der Kantenrichtung kann über das Graphschema die Rolle des Knotens bestimmt werden.

Eine Hyperkante kann mehr als nur jeweils einen Anfangs- und Endknoten haben. Zusätzlich können die Knoten am Anfang bzw. dem Ende einer Hyperkante verschiedene Rollen einnehmen. Die Rolle eines Knotens kann somit nicht mehr ausschließlich von der Richtung der Kante abgeleitet werden. Die Rolle eines Knotens wird im *DHH-TGraphen* mitgeführt.
Um die Inzidenzliste einer Hyperkante und die Rollen der Knoten an Anfang und Ende einer Hyperkante zu unterstützen, müssen die relationalen Schemata angepasst werden.

Dazu muss die Relation `Incidence` erweitert werden. Sie lautet dann:

```
Incidence(
  eId:Integer, vId:Integer, gId:Integer, direction:Direction,
  sequenceNumberInLambdaSeqOfEdge:Long,
  sequenceNumberInLambdaSeqOfVertex:Long,
  roleId:Integer
)
```

Die Attribute der Relation enthalten:

- `eId`: den Fremdschlüssel der Kante aus Relation `Edge`, die durch die Inzidenz mit einem Knoten verbunden ist,
- `vId`: den Fremdschlüssel des Knotens aus Relation `Vertex`, der durch die Inzidenz mit einer Kante verbunden ist,
- `gId`: den Fremdschlüssel des Graphen aus Relation `Graph`, zu dem die Inzidenz gehört,
- `direction`: die Richtung der Kante (`IN` oder `OUT`),
- `sequenceNumberInLambdaSeqOfEdge`: die Sequenznummer, welche die Position der Inzidenz in der Inzidenzliste der verbundenen Kante abbildet,
- `sequenceNumberInLambdaSeqOfVertex`: die Sequenznummer, welche die Position der Inzidenz in der Inzidenzliste des verbundenen Knotens abbildet, und
- `roleId`: den Fremdschlüssel eines Rollennamens der Inzidenz aus Relation `Role`.

Analog zu den Attributen können Redundanzen durch die Namen der Rollen minimiert werden. Dazu werden die Rollennamen in folgende Relation ausgelagert:

```
Role(
  roleId:Integer, name:String
)
```

Die Attribute der Relation enthalten:

- `roleId`: den Primärschlüssel des Rollennamens und
- `name`: den Namen der Rolle.

Auch hier wird die Redundanzminimierung über die kürzere numerische `roleId` als Primärschlüssel erreicht.

Im nächsten Kapitel wird der Entwurf der Lösung behandelt.

6. Konzeptueller Entwurf der Lösung

In diesem Kapitel wird der konzeptuelle Entwurf der entwickelten Lösung behandelt. Im Entwurfsprozess wird die Struktur und das Verhalten der Software geplant. Die so erhaltene Architekturbeschreibung dient als Grundlage der Implementation.
Eine Architekturbeschreibung setzt sich aus den Beschreibungen mehrerer Sichten auf das System zusammen [Reu06]. Ausgehend von der groben Sicht auf das System, wird in diesem Kapitel das dynamische Laden von Graphelementen aus der Datenbank und anschließend ein einfaches Caching-Verfahren vorgestellt.
Das Kapitel schließt mit einer detaillierten Betrachtung der Verwaltung der Sequenznummern. Zunächst wird jedoch eine grundlegende Entwurfsentscheidung dargelegt.

6.1. Exklusiver Zugriff

Der Zugriff auf einen Graphen in einer Datenbank erfolgt exklusiv. Solange ein Benutzer den Graph aus der Datenbank geöffnet hat, soll kein weiterer Benutzer Zugriff auf den Graphen erhalten.
Ziel der Arbeit ist es, eine Lösung zu entwickeln, die es erlaubt, mit Graphen zu arbeiten, deren Speicherbedarf die Größe des physikalischen Hauptspeichers um ein Vielfaches übersteigen. Gleichzeitig soll eine Traversierung dieser Graphen hinreichend schnell möglich sein. Der nebenläufige Zugriff ist entsprechend der Aufgabenstellung nicht Bestandteil dieser Arbeit.
Zudem erhöht die Möglichkeit eines nebenläufigen Zugriffs die Komplexität und den Umfang der Problemstellung erheblich, da zusätzlich Funktionalitäten zur Synchronisierung der Zugriffe auf den Graphen entworfen und implementiert werden müssen.
Für die folgenden Betrachtungen wird somit ein exklusiver Zugriff auf den Graphen angenommen.

6.2. Grobe Architektur

Aus der Problemstellung lassen sich zwei unabhängige Problembereiche identifizieren. Dies sind die *Persistierung von TGraphen in Datenbanken* und das *dynamische Laden eines TGraphs aus einem Hintergundspeicher*. Auf höchster Betrachtungsebene ergeben sich daraus zwei Zuständigkeiten. Diese können auf entsprechende Teile der Lösung abgebildet werden und ergeben

zusammen mit einer Datenbank als Hintergrundspeicher die grobe Sicht auf die Architektur der Lösung.
Die grobe Lösungsarchitektur teilt sich in Abbildung 6.1 in Schichten auf. Jede Schicht nutzt nur ihre unmittelbar darunter liegende Schicht, um ihre Dienste der darüberliegenden Schicht zu erbringen.
Auf oberster Ebene befindet sich die *Graphschicht*. Diese stellt den Dienst des *dynamischen Ladens eines TGraphs aus einem Hintergundspeicher* zur Verfügung. In dieser Schicht wird gesteuert *wann* geladen und persistiert wird.
Darunter befindet sich die *Datenbankabstraktionsschicht*. Diese stellt die Dienste des *Persistierens von TGraphen in Datenbanken* und des *Ladens von TGraphen aus Datenbanken* zur Verfügung. Denn ein Graph soll einmal persistiert auch wieder abgerufen werden können. In dieser Schicht wird gesteuert *wie* geladen und persistiert wird.
Auf der unterste Ebene befindet sich die Datenbank. Diese Schicht wird durch die gewählten Datenbank-Technologien realisiert. Der Entwurf dieser Schicht wurde bereits in den vorherigen Kapiteln behandelt. So wird im weiteren Verlauf des Kapitels nicht weiter darauf eingegangen, sondern das Zusammenspiel der obersten beiden Schichten behandelt.

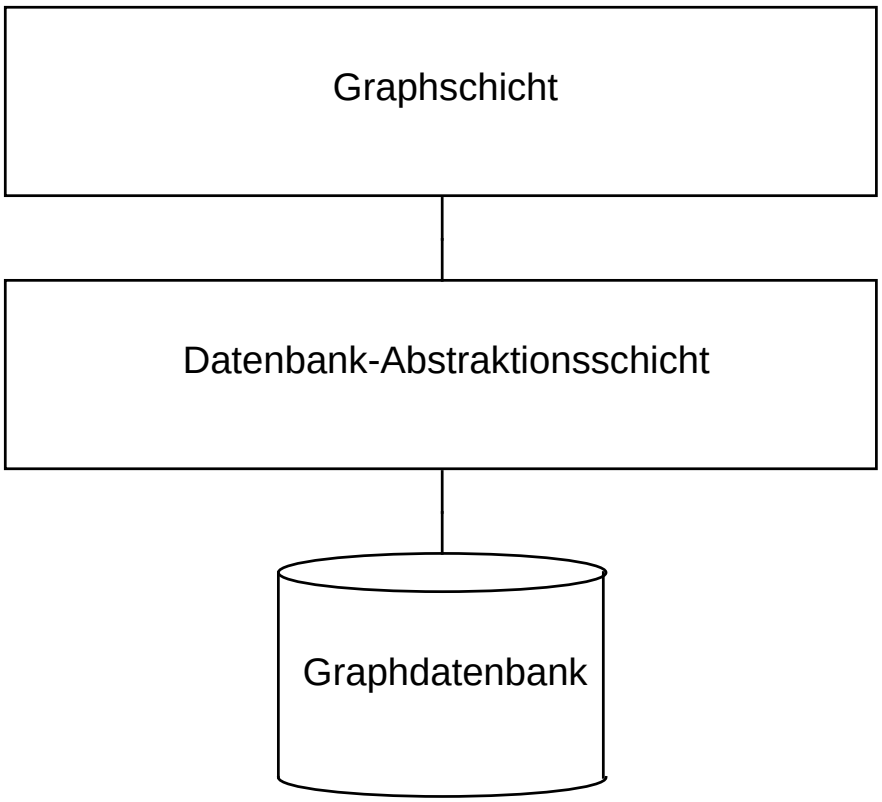

Abbildung 6.1. – Grobe Sicht auf die Architektur der Lösung in Schichten aufgeteilt.

6.3. Dynamisches Laden

Beweggrund für das Persistieren von Graphen in Datenbanken ist ihr möglicher Speicherplatzverbrauch, der die Größe des Arbeitsspeichers eines herkömmlichen Rechners um das Vielfache übersteigen kann. Solche Graphen passen nicht komplett in den lokalen Arbeitsspeicher und sollen deshalb nur partiell geladen werden.
Grundsätzlich gibt es zwischen der klassischen Variante des kompletten Ladens eines Graphen mit all seinen Elementen und der Variante des dynamischen Ladens von Graphelementen einen fundamentalen strukturellen Unterschied. Während in der klassischen Variante ein Graph all seine Elemente immer im lokalen Speicher vorliegen hat und sie über explizite Referenzen erreichen kann, hat der Graph beim dynamischen Laden zunächst *keine* Elemente im lokalen Arbeitsspeicher vorliegen. Er kennt für jedes Element nur die minimal nötigen Informationen zum dynamischen Nachladen dieser Elemente.
Abbildung 6.2 zeigt die vereinfachte Struktur des Graphen. In dieser Fassung ist der Graph mit seinen Knoten und Kanten nicht mehr über Referenzen verbunden. Auch Knoten und Kanten referenzieren ihre Vorgänger und Nachfolger in Knoten-, Kanten- und Inzidenzlisten nicht mehr unmittelbar. Alle Elemente sind nur über einen *Identifier* bekannt. Dieser dient zum Verwalten der Elemente im lokalen Arbeitsspeicher und zum Laden von Elementen aus der Graphdatenbank. Die Klasse `GraphDatabase` realisiert die Dienste der Datenbankabstraktionschicht.
Dazu hat ein Graph den Identifier `gId`, der zudem als Primärschlüssel für den Graph in der Datenbank fungiert. Die Identifier von Knoten (`vId`) und Kanten (`eId`) dienen im Verbund mit dem Identifier des Graphen als Primärschlüssel.
Ein Graph und all seine Elemente sind in genau einer Graphdatenbank enthalten und jede Graphdatenbank kann beliebig viele Graphen aufnehmen.
In der Abbildung aufgeführte Listen verwalten die Reihenfolge der jeweiligen Elemente in der repräsentierten Sequenz. `VertexList` repräsentiert und verwaltet $V\!seq$, `EdgeList` repräsentiert und verwaltet $E\!seq$ und `IncidenceList` repräsentiert und verwaltet Λseq.
Zum Beispiel kann der Graph über `VertexList` den Identifier eines Knotens erfahren, oder ein Knoten kann die Identifier seines Vorgängers und Nachfolgers bestimmen. Eine Kante kennt zusätzlich immer ihren Anfangs- und Endknoten über die Identifier `alphaVId` und `omegaVId`.
Wird nun ein konkretes Element angefordert, so kann der Graph über den Primärschlüssel des angeforderten Elements dieses aus der Datenbank laden.
Zusätzlich referenziert ein Graph genau einen Cache. In diesem werden zuvor geladene Elemente vorgehalten. Der Cache wird in Abschnitt 6.5 behandelt.

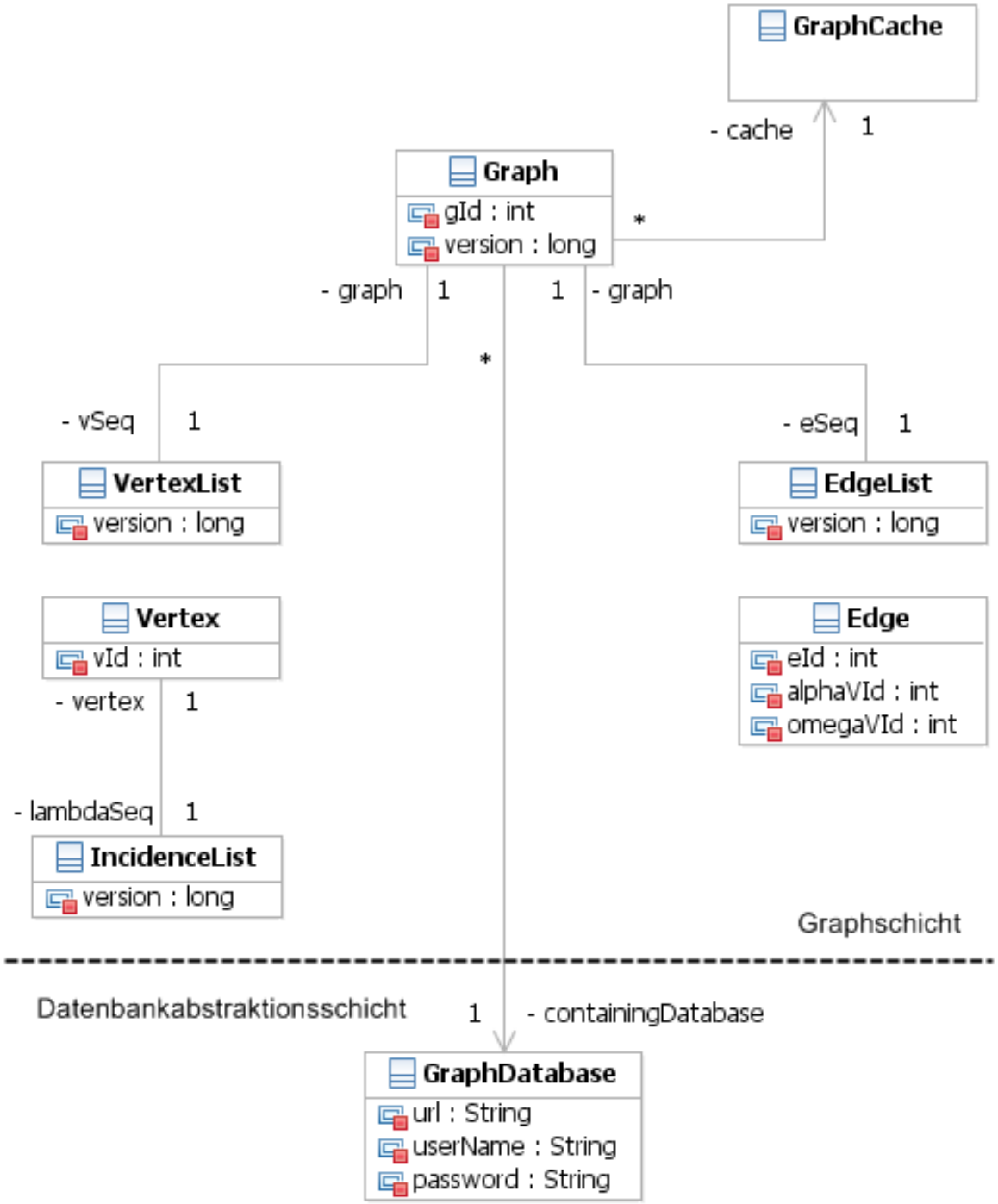

Abbildung 6.2. – Klassendiagramm der vereinfachten Struktur des Graphen zum dynamischen Laden von Elementen.

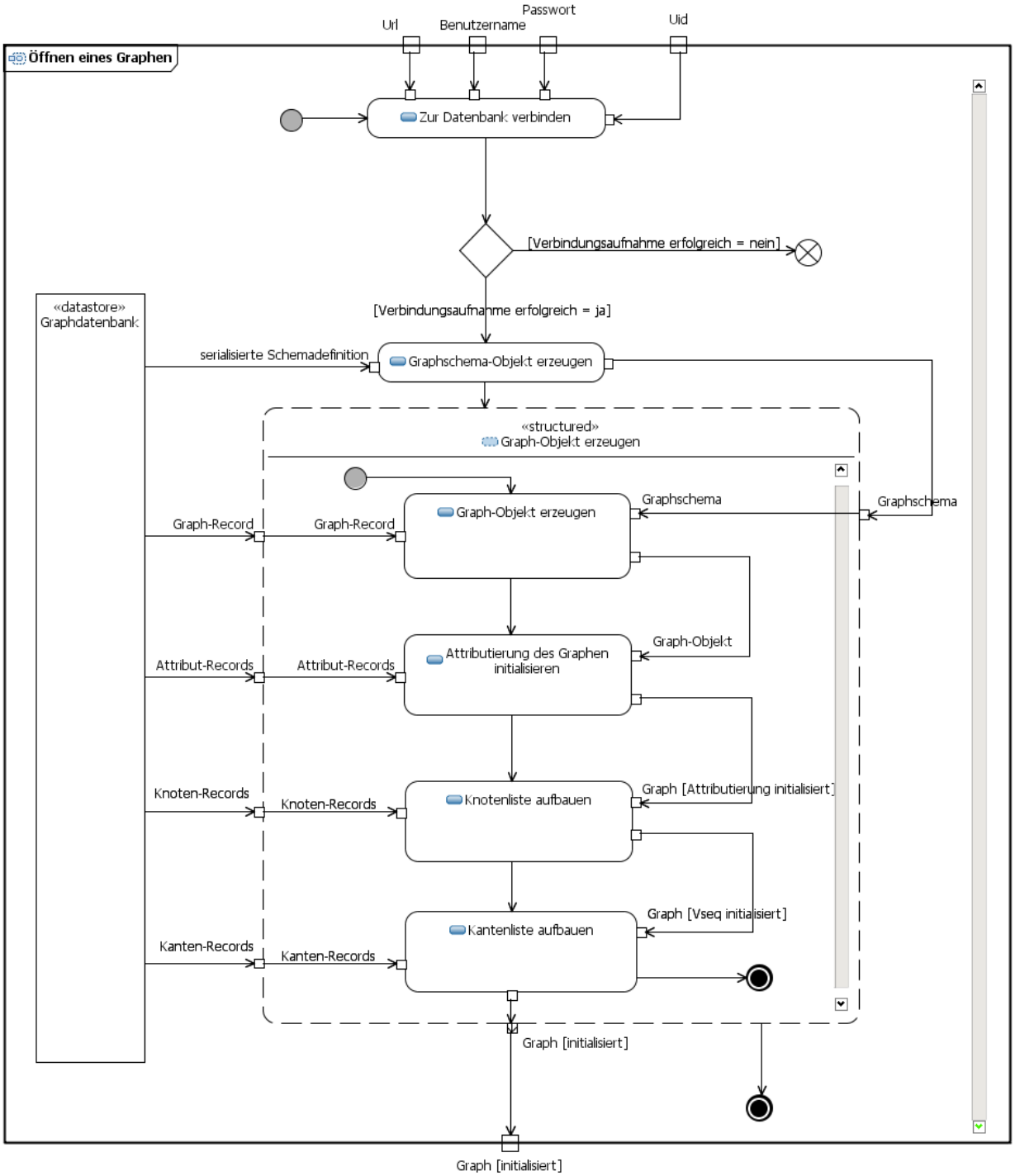

Abbildung 6.3. – Aktivitätsdiagramm - Öffnen eines minimalen Graph aus einer Datenbank.

6.4. Öffnen eines Graphen

Wird ein Graph aus einer Datenbank geöffnet, so wird zunächst nur ein *minimaler* Graph geladen. Ein minimaler Graph besteht aus `Graph`, `VertexList`, `EdgeList` und dem `GraphCache`. Die Graphattributierung ist initialisiert, der Graph kennt sein Graphschema und er kennt die Reihenfolge seiner Elemente in den globalen Sequenzen *Vseq* und *Eseq*. Es werden vorerst keine Knoten und Kanten im lokalen Speicher angelegt.
Abbildung 6.3 fasst den gesamten Vorgang in einem Aktivitätsdiagramm zusammen. Nach der erfolgreichen Verbindungsaufnahme zur Datenbank wird die serialisierte Definition des Graphschemas abgerufen und daraus das Graphschema im lokalen Arbeitsspeicher erzeugt. Dazu müssen die benötigten Klassen zuvor per *Commit* erzeugt worden sein.
Dann wird der Graph im lokalen Arbeitsspeicher angelegt und initialisiert. Dazu wird ihm das Graphschema mitgegeben und sein interner Zustand aus der Datenbank wiederhergestellt. Der Zustand setzt sich zusammen aus:

- der Belegung der Instanzvariablen des Graph-Objekts,
- der Belegung der Attribute des Graphen,
- der Reihenfolge der Knoten in der globalen Knotenliste, und
- der Reihenfolge der Kanten in der globalen Kantenliste.

Diese werden aus den entsprechenden Tabellen der Datenbank geladen. Die Tabellen resultieren aus den Relationen des generischen relationalen Schemas. Folgende Relationen müssen zum Laden eines minimalen Graphen angefasst werden:

- `Graph` und `Type` zum Abruf der Belegung der Instanzvariablen des Graph-Objekts,
- `GraphAttribute` und `Attribute` zum Abruf der Belegung der Attribute des Graphen,
- `Vertex` zum Abruf der Reihenfolge der Knoten in der globalen Knotenliste, und
- `Edge` zum Abruf der Reihenfolge der Kanten in der globalen Kantenliste.

Aus den Relationen werden die entsprechenden Tupel abgerufen. Diese sind als *Records* im Diagramm aufgeführt.
Mit der Erzeugung eines Caches wird die Initialisierung des Graph-Objekts abgeschlossen. Nachdem der minimale Graph erzeugt wurde, *steht er immer im lokalen Arbeitsspeicher zur Verfügung*. Knoten und Kanten können nun dynamisch geladen werden.

6.5. Caching

Caching bezeichnet das Zwischenspeichern von Daten. Ein Cache ist ein Puffer, der Daten zwischenspeichert. Bei erneuter Verwendung der Daten im Cache können diese schneller aus dem

Cache zur Verfügung gestellt werden, als sie erneut aus ihrem Ursprungsspeicher zu laden. Der Geschwindigkeitsvorsprung wird dabei realisiert durch eine Kombination von:

- schnellerem Speicher,
- schnellerer Anbindung des Caches an den Datenanforderer und
- geringerer Entfernung des Caches zum Datenanforderer.

Maßgeblich für das Ausschöpfen des Potentials eines Caches ist das eingesetzte Caching- Verfahren. Da ein Cache in der Regel erheblich kleiner als der langsamere Ursprungsspeicher ist, erschöpft sich seine Speicherkapazität auch wesentlich schneller. Das Caching-Verfahren soll dafür sorgen, dass nicht mehr benötigte Daten im Cache freigegeben werden, um wieder Platz für benötigte Daten zu schaffen [TG01].
Das dynamische Laden von Graphelementen hat gegenüber dem klassischen Ansatz einen entscheidenden Nachteil. Wird ein Graphelement angefordert, so muss es erst aus der Datenbank geladen werden. Nach seiner Verwendung wird das Graphelement aus dem lokalen Arbeitsspeicher gelöscht. Bei der nächsten Verwendung muss dieses Graphelement abermals aus der Datenbank geladen werden. Um diesen Nachteil abzuschwächen *werden bereits geladene Graphelemente in einem Cache zwischengespeichert*.
Im nächsten Abschnitt wird ein einfaches Verfahren zum Zwischenspeichern von Graphelementen beschrieben.

6.6. Graph-Caching-Verfahren

Der *Graph-Cache* befindet sich im lokalen Arbeitsspeicher des Client. *Jedes geladene Graphelement* soll in diesem Cache gesammelt werden, um somit nicht unmittelbar nach seiner Verwendung wieder aus dem lokalen Arbeitsspeicher gelöscht zu werden. Dabei wird ein Graphelement grundsätzlich mit seiner komplett initialisierten Attributierung in den Cache aufgenommen.
Knoten werden zusätzlich mit ihrer Inzidenzliste gecacht, da diese in der Regel komplett durchlaufen wird. Konkret handelt es sich dabei um Objekte vom Typ `Vertex` mit jeweils dazu gehörender `emph`IncidenceList.
Kanten setzen sich aus zwei Objekten zusammen, die jeweils eine Richtung der Kante mit der zugehörigen Inzidenz repräsentieren. Konkret handelt es sich dabei um Objekte vom Typ `emph`Edge mit jeweils dazu gehörender `ReversedEdge`. `Edge` repräsentiert die ausgehende Richtung einer Kante aus einem Startknoten α, mit der verbindenden Inzidenz. *ReversedEdge* repräsentiert die eingehende Richtung einer Kante in einen Endknoten ω ebenfalls mit der verbindenden Inzidenz. Diese Struktur wird durch die Programmierung von *JGraLab* vorgegeben und soll möglichst nicht geändert werden.
Beim nächsten Zugriff auf ein Element kann es direkt lokal aus dem Cache zur Verfügung gestellt werden, vorausgesetzt es befindet sich noch in diesem. Dieser Fall wird *Cache-Hit* genannt. Befindet sich ein angefragtes Element nicht im Cache, so liegt ein *Cache-Miss* vor.

Solange noch Kapazitäten im Cache frei sind, werden weitere Graphelemente in ihm aufgenommen.
Aus dem Cache werden Elemente entfernt, sobald der Cache voll und ein angefordertes Element nicht bereits im Cache ist. In diesem Fall muss für das nachgeladene Element Speicher im Cache freigegeben werden. Dazu müssen Elemente aus dem Cache entfernt werden.
Im einfachsten Ansatz wird das älteste Element aus dem Cache entfernt. Das neue Element nimmt somit den Platz des ältesten ein. Dadurch werden jedoch auch Elemente entfernt, die zwar alt sind, jedoch andauernd angefragt werden und deshalb besser im Cache verbleiben sollten. Besser ist es das am längsten nicht verwendete Element aus dem Cache zu entfernen.
Mit beiden Ansätzen kommt es jedoch zu einem unnötig hohen Aufwand, da nun bei jedem *Cache-Miss* ein Element aus dem Cache entfernt werden muss, um Platz für das neue zu schaffen - denn der Cache ist voll.
Dieser Effekt kann durch das gleichzeitige Entfernen von mehreren Elementen abgeschwächt werden.
Dabei gilt es zu beachten, dass ein Graphelement immer solange verwendbar bleiben muss bis es vom Benutzer freigegeben wird. Selbst dann noch, wenn es während der Benutzung aus dem Cache entfernt wird. Wird ein Graphelement aus dem Cache entfernt so bedeutet dies nicht immer, dass es aus dem lokalen Arbeitsspeicher gelöscht werden darf.
Grundsätzlich wird ein globaler Cache angelegt, der allen geladenen oder erzeugten Graphen zur Verfügung steht. Ein Teil des durch die Lösung reservierten Speichers wird von den minimalen Graphen belegt. Der verbleibende Speicher steht komplett dem Graph-Cache zur Verfügung. Abbildung 6.4 verdeutlicht dies nochmal.

6.7. Laden von Knoten und Kanten

Konnte ein Graph erfolgreich geöffnet werden, so steht ihm ein leerer Cache zur Verfügung. Dieser kann nun mit Knoten und Kanten gefüllt werden.
Knoten und Kanten werden, wenn sie benötigt werden, über ihren Identifier (`vId` und `eId`) zur Verfügung gestellt. Zusammen mit dem Identifier des Graphen (`gId`) ergibt sich ihr Primärschlüssel über den sie in der Datenbank identifiziert werden können.
Abbildung 6.5 stellt den Vorgang *Zurückgeben eines Elements* in einem Aktivitätsdiagramm dar. Das Diagramm abstrahiert über Knoten und Kanten und führt stattdessen `Element` an, da sich der Vorgang für Knoten und Kanten analog verhält.
Zunächst wird der Cache befragt, ob er ein Element mit gegebenen Identifier vorhält. Ist dies der Fall, so wird dieses Element aus dem Cache zurückgegeben. Enthält der Cache kein Element mit diesem Identifier, so wird es über den Primärschlüssel aus der Graphdatenbank geladen.
Nach dem Laden des angeforderten Elements wird das Element in den Cache aufgenommen und zur Verfügung gestellt.

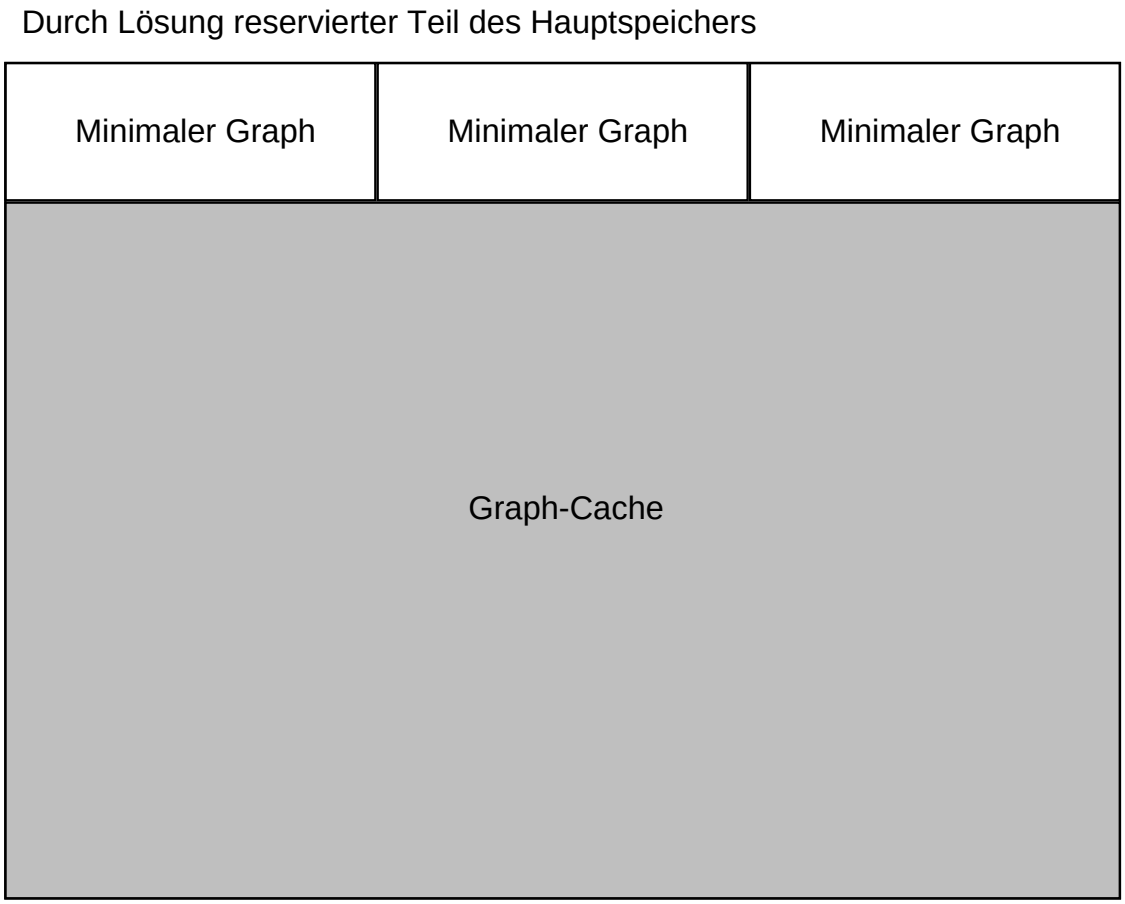

Abbildung 6.4. – Zusammensetzung des durch die Lösung reservierten Hauptspeichers mit drei geladenen Graphen.

6.8. Schreiben von Änderungen

Alle Änderungen am Graphen auf Client-Seite müssen irgendwann in die Datenbank geschrieben werden. Zum Zurückschreiben von Änderungen in den Hintergrundspeicher existieren verschiedene Ansätze [TG01]. Für den Zeitpunkt des Zurückschreibens im Zusammenspiel mit Caches werden zwei Verfahren unterschieden:

- *Write-Through*: Änderungen an Daten werden *direkt* (so früh wie möglich) in den Hintergrundspeicher zurückgeschrieben, und
- *Write-Back*: Änderungen werden *nicht direkt* zurückgeschrieben, sondern erst zu einem späteren Zeitpunkt. Dann werden *alle aufgelaufenen Änderungen* in einem Durchgang zurückgeschrieben.

Da ein Benutzer exklusiven Zugriff auf einen Graph in der Datenbank erhält, können grundsätzlich alle Änderungen am Graph per *Write-Through* direkt in die Datenbank geschrieben

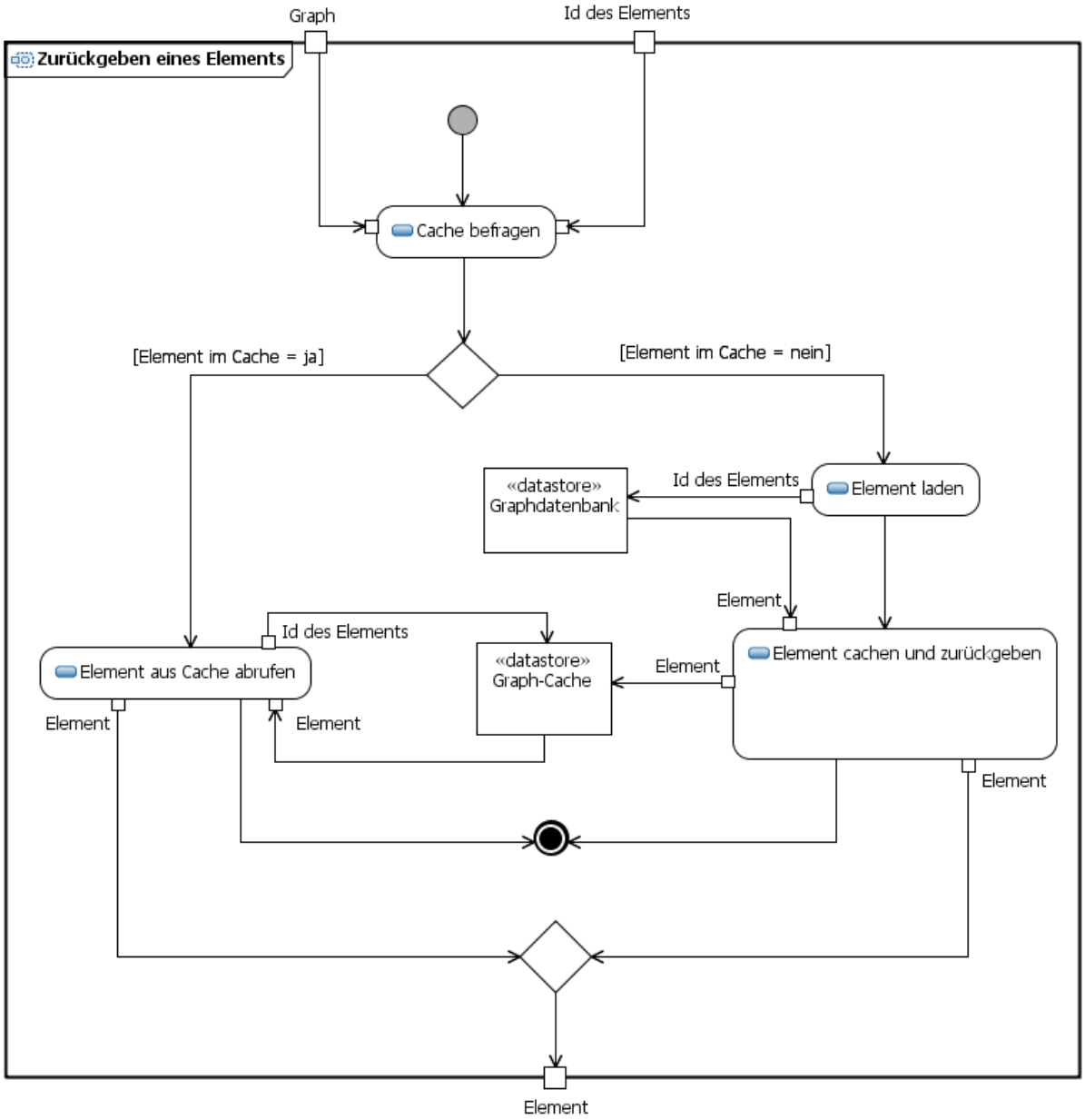

Abbildung 6.5. – Aktivitätsdiagramm - Zurückgeben eines angeforderten Elements.

werden. Bei nebenläufigen Zugriff könnte sonst das Auftreten von Inkonsistenzen zwischen dem Graphen auf Client-Seite und in der Datenbank nicht ausgeschlossen werden.
Um eine höhere Geschwindigkeit bei der Manipulation von Graphen zu erzielen werden nicht alle Änderungen direkt persistiert. Die Version des Graphen und die Versionen von Knoten, Kanten und Inzidenzliste müssen nicht unmittelbar nach einer Änderung persistiert werden. Es genügt diese kurz vor dem Schließen der Graphdatenbank zurückzuschreiben, da ihre aktuellen Wert nur clientseitig für die ordnungsgemäße Funktionsweise der entsprechenden *Failfast-Iteratoren* zur Verfügung stehen muss. Beim Anlegen eines neuen Graphelements wird es erst komplett mit seiner Attributierung initialisiert, bevor es persistiert wird.

6.9. Sequenznummern

Im generischen Datenbankschema wird die *Reihenfolge der Elemente* in den Knoten- und Kantensequenzen sowie in den Inzidenzlisten *auf eine Nummerierung der Elemente abgebildet*. Dabei wird das erste Element einer Sequenz mit der kleinsten und das letzte Element mit der größten Nummer versehen.
Grundsätzlich hat dieser Ansatz den Nachteil, dass ein hoher Aufwand beim Einfügen und Verschieben von Elementen in einer Sequenz entsteht, da die Nummern aller nachfolgenden Elemente entsprechend angepasst werden müssen, damit die Reihenfolge erhalten bleibt.
Durch *Lücken in der Nummerierung* kann verhindert werden, dass dies bei jedem Einfügen und Verschieben geschehen muss.
Durch Vergabe von *Sequenznummern, die durch* 2^k *teilbar* sind, können zwischen zwei Elementen $2^k - 1$ weitere Elemente Platz finden und die Verwaltung der Nummerierung bleibt handhabbar, da die verfügbaren numerischen Typen in Java und den Datenbanken immer eine Kapazität von 2^n aufweisen.
Einer Sequenz können folglich maximal $2^{\mathrm{n-k}}$ Elemente angehören. Dann ist der Zahlenraum *ausgeschöpft*, da kein weiteres Element aufgenommen werden kann, welches einen Abstand zum Vorgänger und Nachfolger von jeweils 2^k einhalten kann.
Zwischen zwei Elemente passen $2^k - 1$ Elemente, jedoch teilt jedes eingefügte Element die Lücke in zwei gleich große Teile, so dass im Mittel weit weniger Einfügungen vorgenommen werden können, bevor eine neue Lücke geschaffen werden muss. Im ungünstigsten Fall können genau k Einfügungen vorgenommen werden.
Im folgenden Abschnitt wird erläutert, wie die Sequenznummern vergeben werden.

6.9.1. Vergabe

Bei Erstellung eines Graphen muss zunächst der Exponenten k gewählt werden, um die Distanz 2^k zwischen den Elementen aller Sequenzen in einem Graphen festzulegen.

Erzeugen eines Elements. Jedes neue Element, welches nun im Graphen erzeugt wird, wird an seine Sequenz hinten angehängt und bekommt dazu eine durch 2^k teilbare und fortlaufende Sequenznummer. Das erste Element jeder Sequenz bekommt als Sequenznummer den Wert 0, das zweite Element den Wert $1 \cdot 2^k$ und das n-te Element den Wert $(n-1) \cdot 2^k$ zugewiesen.
Dabei stellt die 0 die Mitte des Zahlenraums dar, denn Sequenznummern können auch negative Werte annehmen. So können Elemente einer Sequenz genauso einfach voran- wie hintenangestellt werden.
Nach dem ersten Anlegen aller Elemente liegen Lücken der Größe 2^{k-1} zwischen den Nummern einer Sequenz. Die Sequenznummern aller Elemente haben den gleichen Abstand und die Reihenfolge der Elemente in einer Sequenz entspricht zunächst der Reihenfolge ihrer Erzeugung.

Voranstellen eines Elements. Soll ein Element einer Sequenz vorangestellt werden, so wird dem Element die Sequenznummer des ersten Elements abzüglich 2^k zugewiesen. Somit wird es das neue erste Element einer Sequenz, da es die kleinste Nummer in dieser Sequenz aufweist.

Anhängen eines Elements. Soll ein Element einer Sequenz angefügt werden, so wird dem Element die Sequenznummer des letzten Elementes zuzüglich 2^k zugewiesen. Somit wird es das neue letzte Element einer Sequenz, da es die größte Nummer in dieser Sequenz aufweist.

Verschieben eines Elements. Soll die Reihenfolge einer Sequenz geändert, d. h. mindestens ein Element verschoben werden und es nicht gerade das erste oder letzte Element der Sequenz werden soll, so gestaltet sich die Ermittlung der neuen Sequenznummer als aufwändiger.
In *JGraLab* wird ein Element durch die Methoden `putBefore` oder `putAfter` vor oder nach einem gegebenen Element eingefügt[1]. So muss überprüft werden, ob unmittelbar vor oder nach der Sequenznummer des gegebenen Elements noch eine Lücke in den Sequenznummern besteht. Dazu wird die Sequenznummer des direkten Vorgängers bzw. Nachfolgers von der Sequenznummer des gegebenen Elements abgezogen. Der Betrag der Differenz abzüglich Eins ergibt die Größe der verfügbaren Lücke.
Formal dargelegt:

$$g = |s_{\text{given}} - s_{\text{prevOrNext}}| - 1$$

mit
g: der Lückenbreite,
s_{given}: Sequenznummer des gegebenen Elements und

[1] Vorausgesetzt es befindet sich nicht bereits an dieser Stelle.

$s_{\text{prevOrNext}}$: Sequenznummer des Vorgängers oder Nachfolgers des gegebenen Elements.

Beträgt die Lückenbreite $g = 1$, dann bekommt das zu verschiebende Element die Sequenznummer des gegebenen Elements abzüglich bzw. zuzüglich 1 zugewiesen, je nach dem ob es der neue Vorgänger oder Nachfolger des gegebenen Elements werden soll.
Ist die Lückenbreite $g > 1$, so wird die Lückenbreite ohne Rest durch 2 dividiert, um die Mitte der freien Lücke zu ermitteln. Das zu verschiebende Element wird dann in der Mitte der Lücke positioniert. Soll es vor dem gegebenen Element eingefügt werden, so wird das Ergebnis der Division von der Sequenznummer des gegebenen Elements abgezogen und dem zu verschiebenden Element als neue Sequenznummer zugewiesen. Soll es nach dem gegebenen Element eingefügt werden, so wird das Ergebnis der Division addiert.
Nun kann es auch vorkommen, dass die Lückenbreite $g = 0$ ist. Dann muss eine Lücke geschaffen werden. Mögliche Ansätze werden im nächsten Abschnitt beschrieben.

6.9.2. Lücken erzeugen.

Die benötigte Lücke kann durch drei verschiedene Verfahren erzeugt werden:

Komplette Reorganisation. Eine komplette Reorganisation geschieht durch Aktualisieren der Nummern *aller* Elemente einer betroffenen Sequenz mit Werten, die durch 2^k ohne Rest dividierbar sind. Danach haben alle vergebenen Nummern in der Sequenz wieder den gleichen Abstand.
Der Aufwand für *eine* komplette Reorganisation beträgt immer m, mit m als Anzahl der Elemente in einer Sequenz.
Im zeitlichen Verlauf betrachtet entsteht ein Gesamtaufwand von $r_c \cdot m$, mit r_c als Anzahl der kompletten Reorganisationen. Verteilt man den Gesamtaufwand zeitlich über die Anzahl der vorgenommenen Einfügungen e, so erhält man den mittleren Aufwand über die vorgenommenen Einfügungen $(r_c \cdot m)/e$.
Nimmt man nun an, dass die Anzahl der Elemente in der Sequenz gleich bleibt und beim Einfügen von Elementen immer der ungünstigste Fall vorliegt, also alle $k + 1$ Einfügungen keine Lücke mehr vorhanden ist, so beträgt der mittlere Aufwand über die vorgenommenen Einfügungen $m/(k + 1)$. In der Regel sollte die Anzahl der Einfügungen $e > k + 1$ sein, da Elemente nicht immer in den gleichen Zielbereich verschoben werden bevor eine Reorganisation nötig wird.
Eine komplette Reorganisation kann nur vorgenommen werden, wenn der Zahlenraum noch nicht erschöpft ist[2].

[2] In diesem Fall muss k verkleinert werden.

Partielle Reorganisation. Eine partielle Reorganisation geschieht durch Aktualisieren der Nummern jener Elemente einer betroffenen Sequenz *mit kleinerer bzw. größerer Sequenznummer*, als der Sequenznummer des gegebenen Elements. Ebenfalls mit Werten, die durch 2^k ohne Rest dividierbar sind. Anschließend haben alle vergebenen Nummern einer Sequenz vor bzw. nach dem gegebenen Element den gleichen Abstand. In einer Sequenz wird rechts oder links neben dem gegebenen Element reorganisiert. Es wird jene Seite reorganisiert, die weniger Elemente hat.
Der Aufwand für *eine* partielle Reorganisation beträgt f, mit f als Anzahl der reorganisierten Elemente. Im besten Fall ist $f = 1$ und im ungünstigsten Fall $f = m/2$, mit m als Anzahl der Elemente in einer Sequenz.
Über den zeitlichen Verlauf entsteht ein Gesamtaufwand von $\sum_{i=0}^{r_p} f_i$ mit r_p als Anzahl vorgenommener partieller Reorganisationen und f_i als Anzahl der reorganisierten Elemente zur Reorganisation i. Verteilt man den Gesamtaufwand zeitlich über die Anzahl der vorgenommenen Einfügungen e, so erhält man den mittleren Aufwand über die vorgenommenen Einfügungen $(\sum_{i=0}^{r_p} f_i)/e$.
Auch eine partielle Reorganisation kann nur vorgenommen werden, wenn der Zahlenraum noch nicht erschöpft ist.

Lokale Reorganisation. Eine lokale Reorganisation kann durch Aktualisieren der Nummern von Elementen im Zielgebiet einer betroffenen Sequenz vorgenommen werden. Lokal bedeutet, dass *maximal die* $2^k - 1$ *Vorgänger- oder Nachfolger* des gegebenen Elements betroffen sind. Im Gegensatz zu den zuvor vorgestellten Verfahren wird nicht sofort reorganisiert, sondern erst in der Nachbarschaft des gegebenen Elements nach einer freien Sequenznummer gesucht.
Wird eine freie Sequenznummer in diesem Bereich gefunden, so kann eine Lücke durch lokale Reorganisation geschaffen werden. Dazu müssen die Sequenznummern jener Elemente aktualisiert werden, die zwischen dem Zielgebiet und der freien Sequenznummer liegen. Durch *Verschieben* der Elemente im Zahlenraum um 1, wird eine Lücke der Größe $g = 1$ im Zielgebiet geschaffen. Dort wird das zu verschiebende Element dann eingefügt.
Der Aufwand für *eine* partielle Reorganisation beträgt ebenfalls f. Im besten Fall ist $f = 1$ und im ungünstigsten $f = 2^k - 1$ (da das Verfahren darauf begrenzt ist). Der zeitlich entstehende Gesamtaufwand und der mittlere Aufwand über die vorgenommenen Einfügungen entsprechen denen der partiellen Reorganisation.
Eine lokale Reorganisation kann selbst wenn der Zahlenraum erschöpft ist vorgenommen werden.

6.9.3. Festlegungen

Der Exponent *k wird auf 32 festgelegt*. Als Datentyp für die Sequenznummern dient der Java-Typ `long`, der eine Zahlenraum der Breite 2^{64} aufspannt und dem Datenbanktyp `BIGINT` ent-

spricht. Es können somit maximal 2^{32} Elemente in einer Sequenz Platz finden.
Da in einem Graphen maximal 2^{31} Knoten und ebenso viele Kanten enthalten sein können, kann der Zahlenraum nicht ausgeschöpft werden.
Zum Erzeugen von benötigten Lücken in den Sequenzen wird deshalb nur die *komplette Reorganisation* implementiert.

Im nächsten Kapitel wird die programmatische Umsetzung der entworfenen Lösung vorgestellt.

7. Objektorientierter Feinentwurf

In diesem Kapitel wird der objektorientierte Feinentwurf zur Integration der Lösung in *JGraLab* beschrieben. Alle behandelten Schnittstellen und Klassen gehören, wenn nicht anders genannt, zum Package `de.uni_koblenz.jgralab.impl.db`.
Zunächst wird die Graphschicht behandelt und dann die Datenbankabstraktionsschicht beschrieben.

7.1. Graphschicht

In der Graphschicht wird das *dynamische Laden von Graphelementen* realisiert. Diese werden in den folgenden Abschnitten beschrieben.

7.1.1. Schnittstellen von Graph, Knoten und Kante

In der Graphdatenbank werden die internen Zustände von Graphen, Knoten und Kanten persistiert. Durch die Implementierungsklassen im Package `de.uni_koblenz.jgralab.impl` sind nicht die kompletten internen Zustände der Objekte zugänglich. So werden zur Ergänzung entsprechende Interfaces definiert, die einen Zugriff auf die internen Zustände fordern. Zusätzlich werden spezifische Methoden für das dynamische Laden von Graphelementen eingeführt. Einen Überblick verschafft das Klassendiagramm in Abbildung 7.1.

`DatabasePersistable`. Das Interface `DatabasePersistable` definiert Zugriffsmethoden auf den Persistenz- und Initialisierungszustand eines Objekts. Somit kann festgestellt werden, ob der interne Zustand eines Objekts bereits persistiert und/oder wiederhergestellt wurde. Zudem kann der Primärschlüssel des Graphen abgefragt werden.

`DatabasePersistableGraph`. Das Interface `DatabasePersistableGraph` definiert Methoden zum Setzen des Primärschlüssels eines Graphen sowie zum Füllen der Knoten- und Kantenliste, um einen bereits existierenden Graphen wiederherzustellen. Zusätzlich definiert es Methoden zum Füllen der globalen Knoten- und Kantenlisten, um einen bereits existierenden Graphen wiederherzustellen.

DatabasePersistableVertex. Das Interface `DatabasePersistableVertex` definiert Zugriffsmethoden auf die Sequenznummer eines Knotens in der globalen Knotensequenz *Vseq* und die Version der Inzidenzliste des Knotens. Zusätzlich definiert es eine Methode zum Füllen der Inzidenzliste, um einen bereits existierenden Knoten wiederherzustellen.

DatabasePersistableEdge. Das Interface `DatabasePersistableEdge` definiert Zugriffsmethoden auf die Sequenznummer in der globalen Kantensequenz *Eseq*.

DatabasePersistableIncidence. Das Interface `DatabasePersistableIncidence` definiert Zugriffsmethoden auf die Sequenznummer in der lokalen Inzidenzliste Λ*seq* des inzidenten Knotens einer Kante. Zusätzlich definiert das Interface Zugriffsmethoden auf den Identifier des inzidenten Knotens und der inzidenten Kante.

7.1.2. Implementierung von Graph, Knoten und Kante

Die definierten Schnittstellen werden durch Klassen implementiert, die zudem von ihren Pendants aus dem Paket `de.uni_koblenz.jgralab.impl` erben. Die neuen Implementierungsklassen überschreiben eine Reihe von Methoden ihrer Basisklassen, um die Standardfunktionalität durch Datenbankpersistenz und dynamisches Laden zu ersetzen.

GraphImpl. Die Klasse `GraphImpl` repräsentiert einen Graph, der in einer Graphdatenbank persistiert werden kann. Sie implementiert die Schnittstelle `DatabasePersistableGraph` und bietet Methoden zum Anlegen, Ändern und Lesen von Elementen im Graph und somit in der Datenbank an. Sie hat Zugriff auf den `GraphCache` und die `GraphDatabase` und lädt ggf. Graphelemente nach.
Zur Verwaltung der Graphelemente referenziert der Graph jeweils eine `VertexList` und eine `EdgeList`.

VertexImpl. Die Klasse `VertexImpl` repräsentiert einen Knoten, der in einer Graphdatenbank persistiert werden kann. Die Schnittstelle `DatabasePersistableVertex` wird von der Klasse `VertexImpl` implementiert. Zur Verwaltung der inzidenten Kanten des repräsentierten Knotens, referenziert die Klasse eine `IncidenceList`.

EdgeImpl. Die Klasse `EdgeImpl` repräsentiert eine Kante. Die Klasse selbst enthält jedoch nur die Inzidenz am Anfangsknoten α einer Kante, die in einer Graphdatenbank persistiert werden kann und implementiert die Schnittstelle `DatabasePersistableIncidence`.
Der Zugriff auf die Inzidenz am Endknoten ω der Kante wird über Klasse `ReversedEdgeImpl` realisiert, die von `EdgeImpl` komponiert wird.

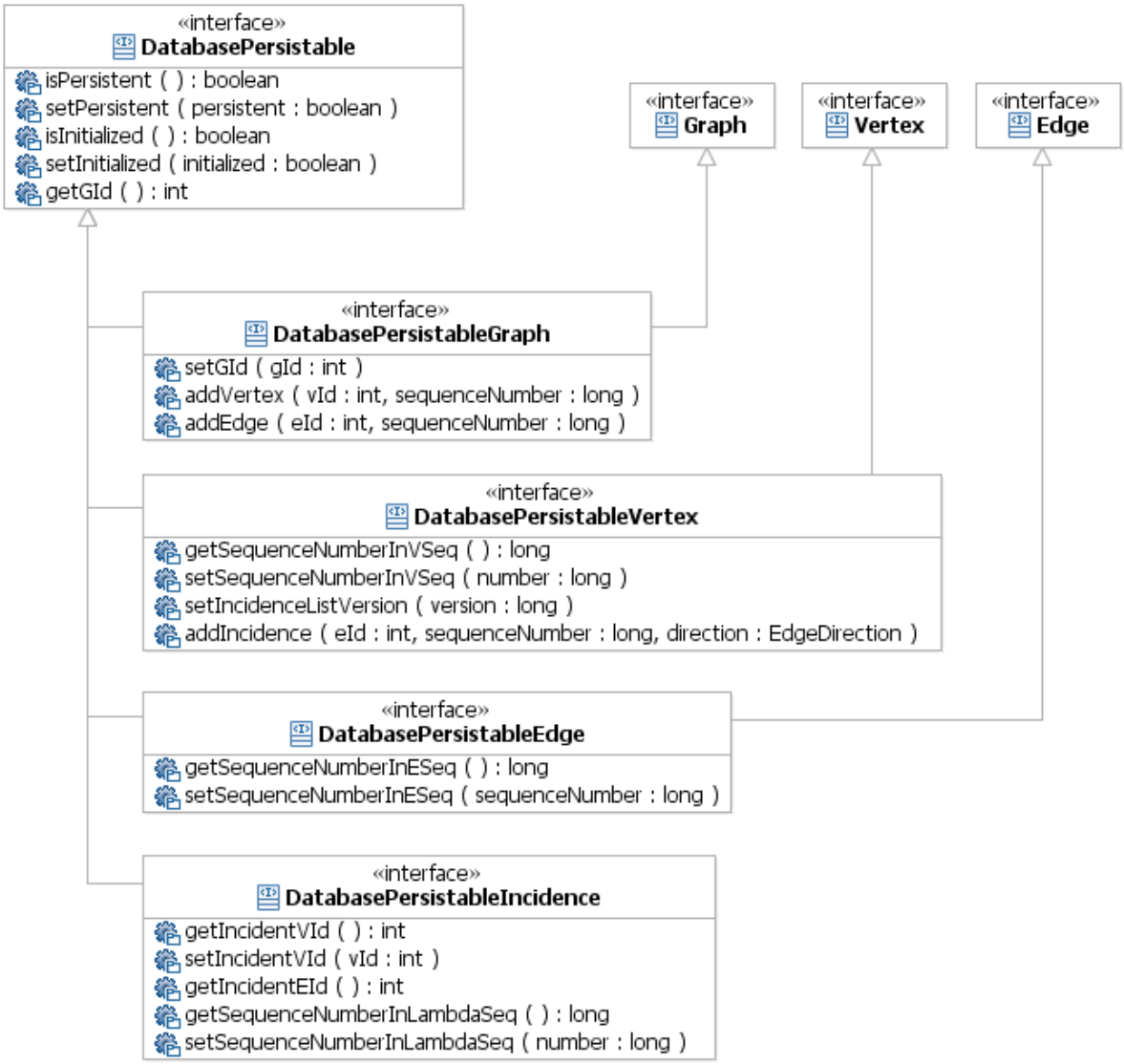

Abbildung 7.1. – Klassendiagramm der Interfaces.

`ReversedEdgeImpl`. Die Klasse `ReversedEdgeImpl` repräsentiert die eingehende Richtung und Inzidenz einer Kante in den Endknoten ω. Eine Kante wird somit immer durch zwei Klassen repräsentiert.
Die Klasse implementiert ebenfalls die Schnittstelle `DatabasePersistableIncidence`.

Soll nun ein Graph, ein Knoten oder eine Kante persistiert werden, so übergibt die Klasse `GraphImpl` das entsprechende Objekt an `GraphDatabase`, welche für die konsistente Persistierung in der gewählten relationalen Datenbank sorgt.

7.1.3. Sequenzen im Graph

Zur Verbesserung der Wartbarkeit der Lösung wird jede Sequenz durch eine eigene Klasse implementiert. Abbildung 7.2 führt diese in einem Klassendiagramm auf.
Eine Sequenz wird nicht explizit an `GraphDatabase` zur Persistierung übergeben, sondern wird implizit über die in ihr enthaltenen Elemente persistiert.

`GraphElementList<T>`. Die abstrakte Klasse `GraphElementList` fasst gemeinsame Eigenschaften und Funktionalitäten der Listen im Graph zusammen. Sie implementiert Funktionalitäten zur Versionierung einer Liste und zur Berechnung von freien Sequenznummern, die zur Abbildung der Reihenfolge von Elementen einer Liste dienen. Der Typ der enthaltenen Elemente wird über den Typparameter `T` festgelegt. Klassen, die von `GraphElementList` erben, müssen ihre verwalteten Elemente nach Sequenznummern sortieren.
Zusätzlich definiert die Klasse abstrakte Methoden zum Lesen, Ändern und Reorganisieren einer Liste von Graphelementen. Diese Methoden werden in den folgenden Klassen implementiert

`VertexList`. Die Klasse repräsentiert die globale Knotenliste *Vseq* des Graphen und erbt einen Teil ihrer Funktionalität von `GraphElementList`. Sie aggregiert eine Sammlung von Knoten-Ids, die nach Sequenznummern sortiert sind und implementiert die abstrakten Methoden aus `GraphElementList` entsprechend. Über `VertexList` kann der erste und letzte Knoten in *Vseq* sowie der direkte Vorgänger und Nachfolger eines gegebenen Knotens abgefragt werden. Zusätzlich kann die Länge der Knotenliste ermittelt werden. Außerdem können Knoten der globalen Knotenliste vorangestellt oder angehängt sowie ihre Reihenfolge geändert werden.
`VertexImpl` und `GraphImpl` bieten ebenfalls Methoden zum Zugreifen und Manipulieren von *Vseq* an. Diese delegieren ihre Aufgabe jedoch an `VertexList`.
Zusätzlich definiert `VertexList` eine Methode zum Wiederherstellen der globalen Knotenliste eines bereits in der Datenbank persistierten Graphen.

`EdgeList`. Die Klasse `EdgeList` repräsentiert die globale Kantenliste *Eseq* des Graphen und erbt einen Teil ihrer Funktionalität von der abstrakten Klasse `GraphElementList`. Ihre Funktionalität implementiert sie analog zu `Vertexlist`.
Methoden aus `EdgeImpl` und `GraphImpl` zum Zugriff auf und zur Manipulation von *Eseq* delegieren ihre Aufgabe an `EdgeList`.

`IncidenceList`. Die Klasse `IncidenceList` repräsentiert die lokale Inzidenzliste Λseq eines Knotens und erbt ebenfalls einen Teil ihrer Funktionalität von der abstrakten Klasse `List`. Ihre Funktionalität implementiert sie ebenso analog zu `Vertexlist`.

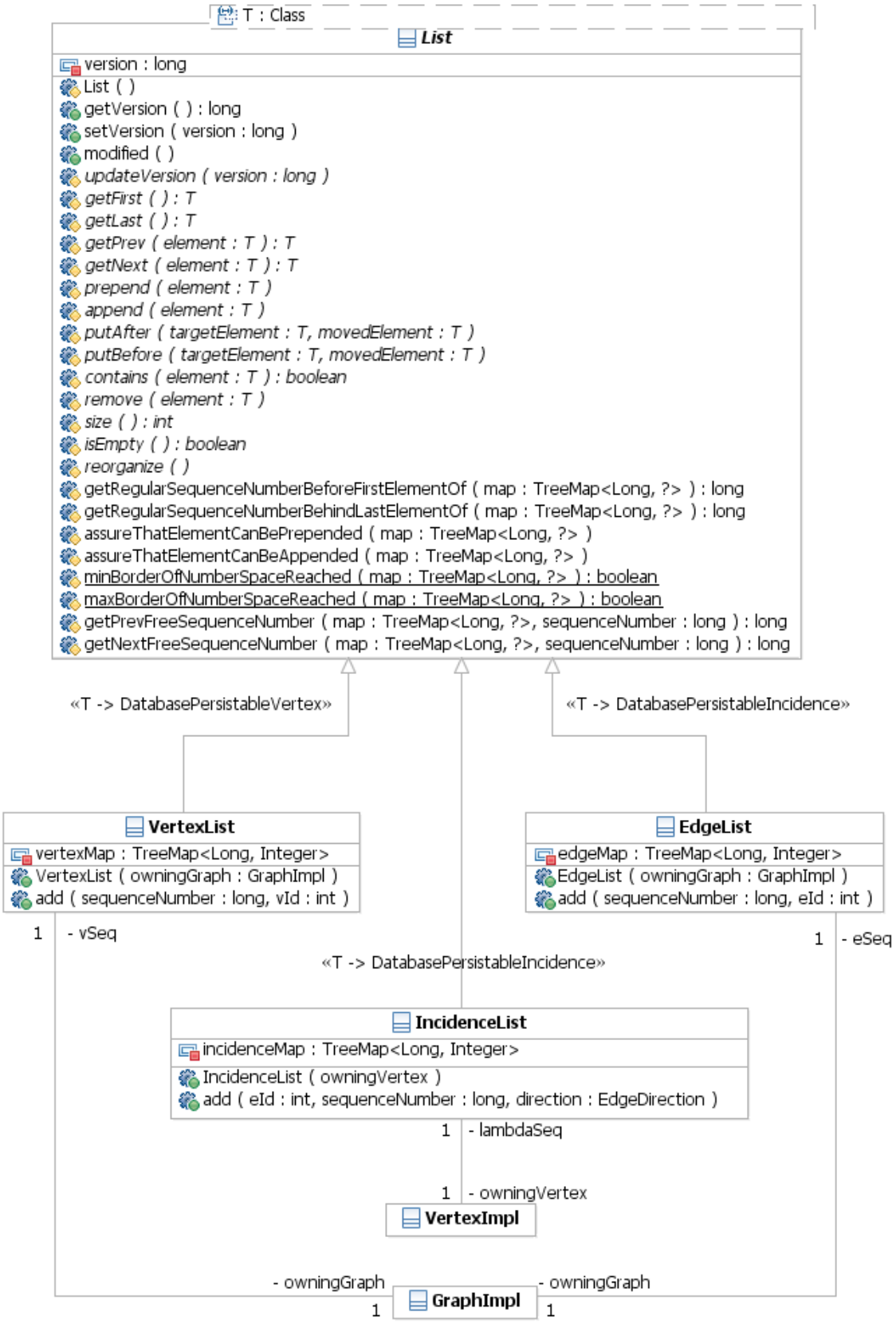

Abbildung 7.2. – Klassendiagramm der Listen im Graph. `VertexList`, `EdgeList` und `IncidenceList` implementieren jeweils alle abstrakte Methoden aus `GraphElementList`.

Methoden aus `EdgeImpl` und `VertexImpl` zum Zugriff auf und zur Manipulation von Λ_{seq} delegieren ihre Aufgabe an `IncidenceList`.

Wird ein Knoten per Aufruf von `getNextVertex()` aufgefordert, seinen Nachfolger aus *Vseq* zurückzugeben, so delegiert er diese Aufgabe an `GraphImpl`, welche diese wiederum an die Klasse `VertexList` delegiert. `VertexList` ermittelt nun den Identifier des Nachfolgerknotens und ruft diesen bei `GraphImpl` über `getVertex(int)` ab.
`GraphImpl` befragt zunächst `GraphCache`. Ist der angeforderte Knoten nicht im Cache, so wird dieser von der Klasse `GraphDatabase` geladen.

7.1.4. Caching

Die Klasse `GraphCache` implementiert den Cache (siehe Abbildung 7.3). Knoten und Kanten werden über ihren Identifier verwaltet. Sie können über diesen abgerufen, aus dem Cache entfernt oder auf ihre Mitgliedschaft im Cache geprüft werden. Ferner kann der gesamte Cache geleert werden.
Die Klasse `GraphCache` überlässt dem *Garbage-Collector* das Entfernen von Elementen aus dem Cache. Jedes aufgenommene Objekt ist dazu nur über eine `WeakReference` erreichbar. Eine `WeakReference` gibt ein referenziertes Objekt zum Löschen durch den *Garbage Collector* frei. Solange das Objekt noch nicht gelöscht wurde, bleibt es über die `WeakReference` erreichbar [Nic10].
Jedes noch verwendete Graphelement, also in Java noch durch eine *harte* Referenz erreichbar, verbleibt weiterhin im Cache. Graphelemente bleiben somit während ihrer gesamten Inanspruchnahme immer vollständig benutzbar[1]. Abbildung 7.4 stellt diese Situation dar.
Ein Graphelement wird nur explizit per `remove()` entfernt, wenn es aus dem Graph gelöscht wird.

7.2. Datenbankabstraktionsschicht

Im Klassendiagramm aus Abbildung 7.5 sind alle Klassen aufgeführt, welche die *Datenbankabstraktionsschicht* realisieren und den Zugriff auf die physische relationale Datenbank kapseln.
Die Graphdatenbank wird auf einem relationalen Datenbankverwaltungssystem gehostet. Unterstützt werden *MySQL 5.1.5*, *PostgreSql 8.4* und *JavaDB 10.23*.

[1] Die Prüfung auf Gleichheit über den Gleichheitsoperator (==) bleibt weiterhin möglich.

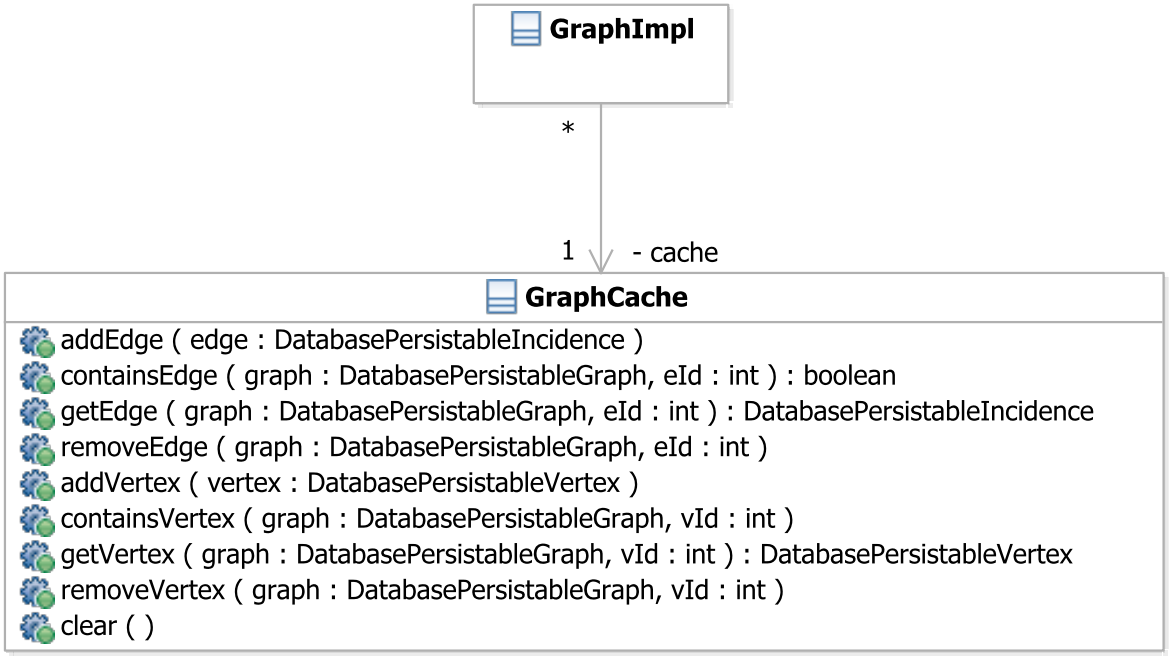

Abbildung 7.3. – Klassendiagramm des Caches.

7.2.1. Datenbank

Die Klasse `GraphDatabase` repräsentiert auf Clientseite die physische Graphdatenbank und bietet Methoden zum Lesen und Ändern von Graphen, Knoten, Kanten und ihren Attributen an. Graphschemata können zwar angelegt, geladen und gelöscht werden, nach dem Anlegen aber nicht in Details verändert, sondern nur komplett überschrieben werden.
Damit keine `GraphDatabase` instanziiert werden kann, die keine Verbindung zu ihrem physischen Pendant hat, kann eine `GraphDatabase` nur über eine *Factory-Methode* (Entwurfsmuster aus [GHJV95]) erzeugt werden. Dieser Methode werden die *URL zur physischen Datenbank* sowie *Benutzername* und *Passwort* einer Benutzerrolle mit Schreib- und Leserechten auf diese Datenbank übergeben.
Anhand der URL stellt die Klasse fest, ob es sich um eine *MySQL-*, *PostgreSql-* oder einer *JavaDB-Datenbank* bzw. *Derby-Datenbank* handelt und erzeugt für den internen Gebrauch die entsprechende Subklasse von `SqlStatementList`. Die Verbindung zur physischen Datenbank wird über *JDBC* hergestellt. Die Klasse `GraphDatabase` definiert dazu eine Referenz auf die Klasse `Connection` aus dem Java-Paket `java.sql`. Wurde zuvor bereits eine Verbindung zur Datenbank hergestellt und noch nicht geschlossen, so wird diese wiederverwendet.

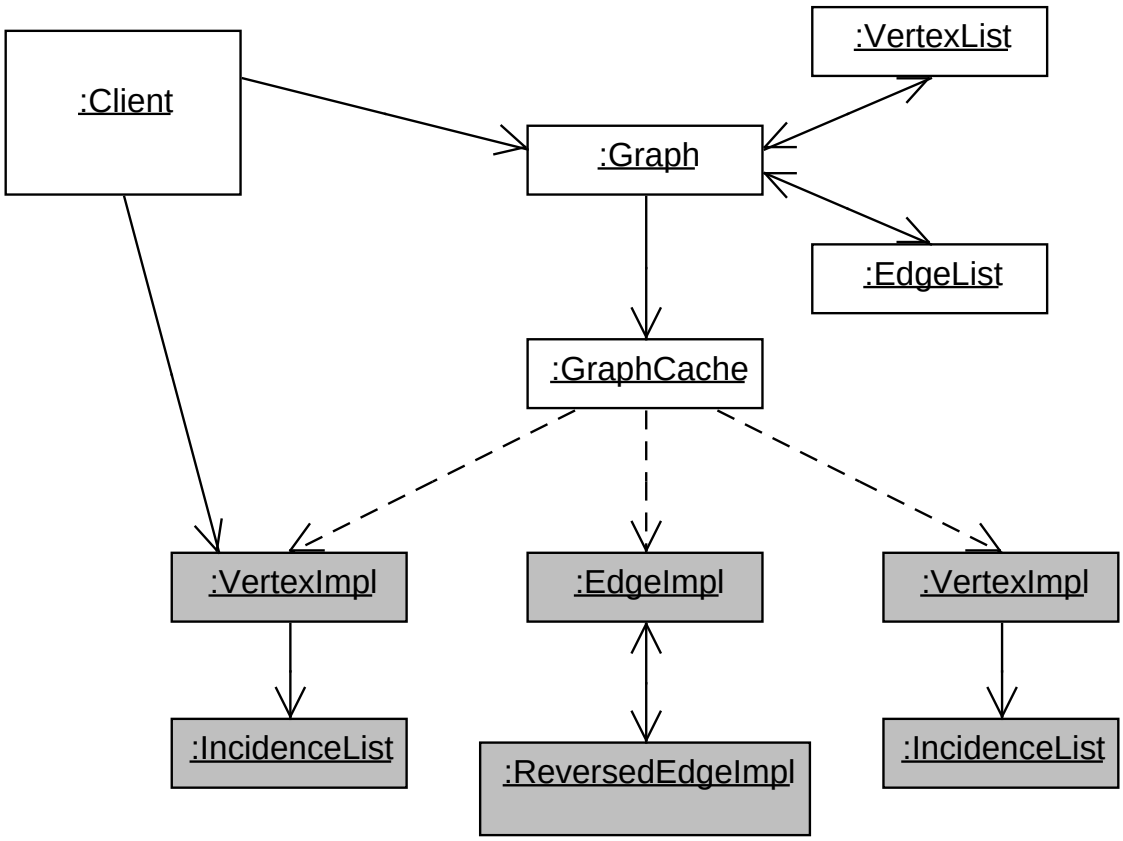

Abbildung 7.4. – Objektdiagramm zur Laufzeit, welches Elemente im Cache zeigt (grau hinterlegt). Eine gestrichelte Linie repräsentiert eine `WeakReference`.

7.2.2. Generisches relationales Schema

Die Klassen, die in diesem Abschnitt behandelt werden, implementieren zusammen das *generische relationale Schema*. Soll ein anderes Schema zum Einsatz kommen, so müssen die Klassen entsprechend neu implementiert werden.

`SqlStatementList`. Die abstrakte Klasse `SqlStatementList` definiert zu implementierende Methoden, die ausführbare SQL-Anweisungen zurückgeben, die zur Arbeit mit Graphen in einer Datenbank nötig sind. Dies sind Operationen zum Lesen, Schreiben und Löschen von Graphschemata, Graphen, Knoten, Kanten, Inzidenzen und Attributen sowie zum Anwenden des physikalischen Datenbankschemas, welches dem generischen relationalen Schema entspricht. Dazu implementiert die Klasse das Entwurfsmuster *AbstractFactory* (siehe [GHJV95]). Zur serverseitigen Reorganisation der Sequenzen enthält sie zudem entsprechende *Stored Pro-*

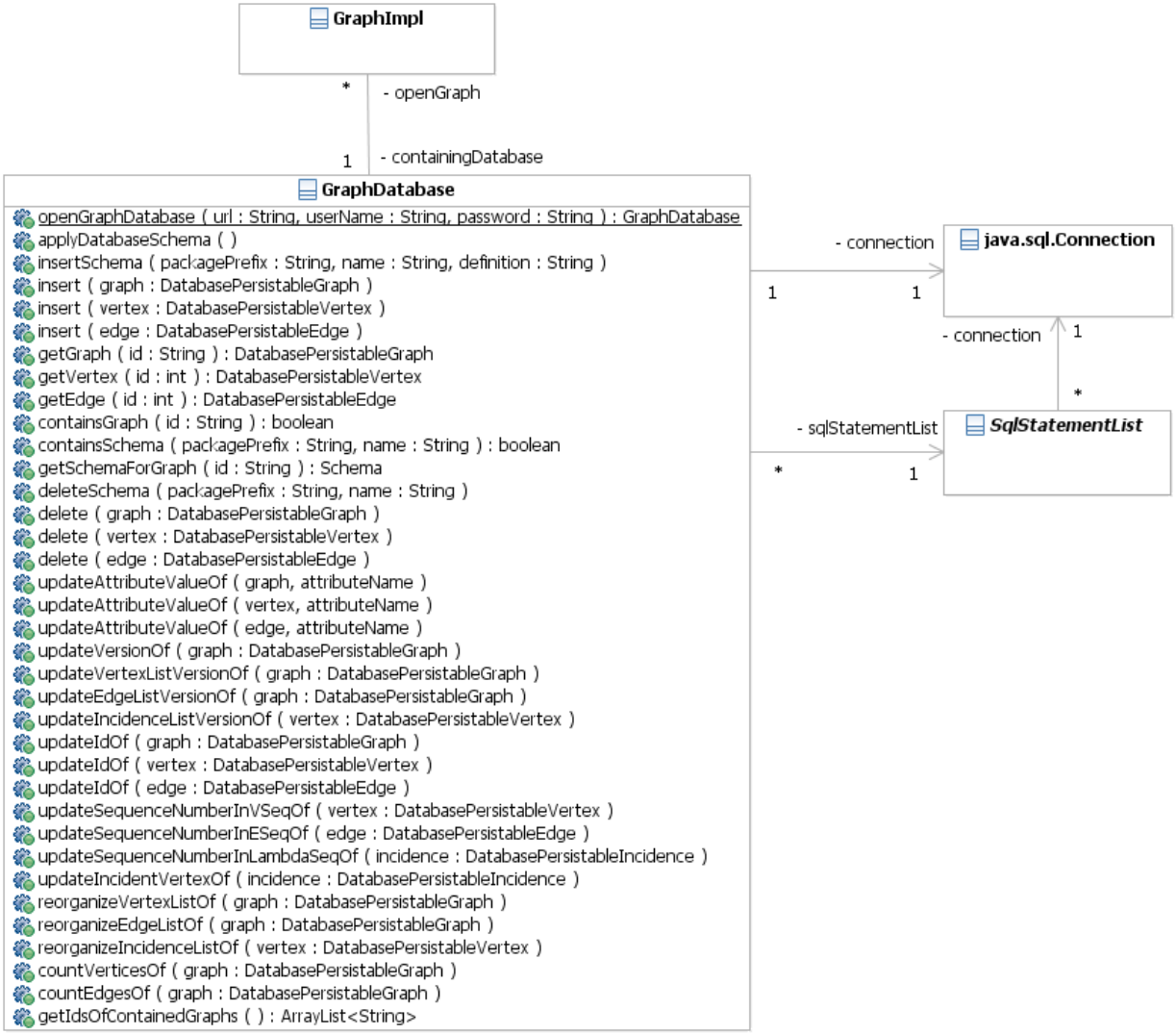

Abbildung 7.5. – Klassendiagramm der Graphdatenbank. Der Übersichtlichkeit halber sind die Inhalte von `SqlStatementList` ausgeblendet.

cedures[2].

`MySqlStatementList`. Die Klasse `MySqlStatementList` implementiert alle abstrakten Methoden aus `SqlStatementList`, so dass SQL-Anweisungen, die *im MySQL-spezifischen SQL-Dialekt* verfasst sind, zurückgegegben werden.

`PostgreSqlStatementList`. Die Methoden der Klasse `PostgreSqlStatementList` geben analog zur Klasse `MySqlStatementList` Anweisungen zurück, die *im PostgreSql-spezifischen SQL-Dialekt* verfasst sind.

`DerbyStatementList`. Die Methoden der Klasse `DerbyStatementList` geben Anweisungen zurück, die *im JavaDB-spezifischen SQL-Dialekt* verfasst sind.

7.3. Übersicht

Das Klassendiagramm in Abbildung 7.6 bietet eine Übersicht der entworfenen Klassen und Schnittstellen. Der Übersichtlichkeit halber sind alle internen Klassen ausgeblendet.

[2]Operation, die auf dem Datenbankverwaltungssystem ausgeführt wird.

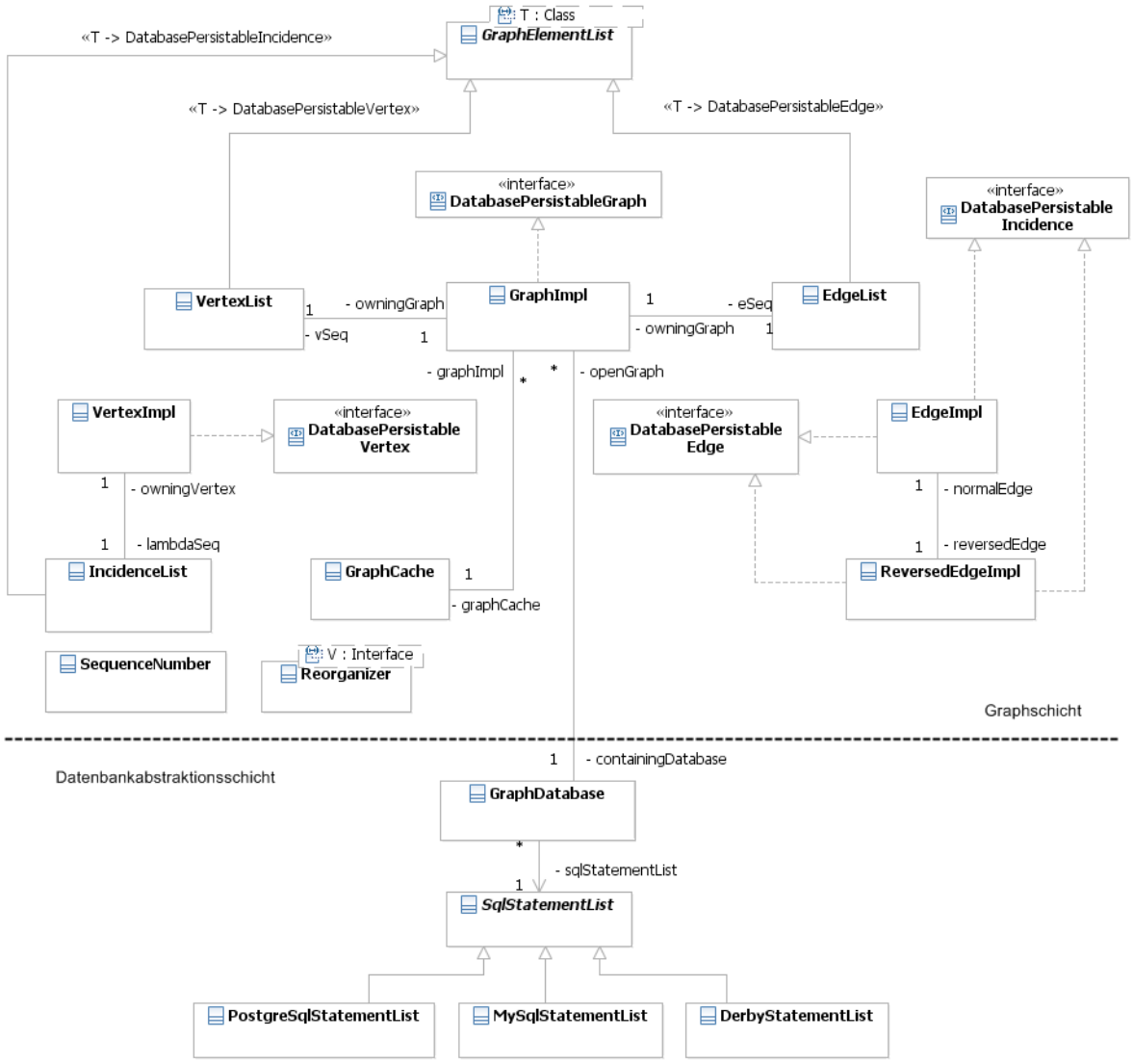

Abbildung 7.6. – Übersichts-Klassendiagramm des Pakets `de.uni_koblenz.jgralab.impl.db`

8. Verwendung der Implementation

In diesem Kapitel wird die Verwendung der Implementation an einfachen Beispielen erklärt.

8.1. Einsatzszenarien

Die *Datenbank-Persistenz-Implementation* ist für die folgenden zwei Einsatzszenarien ausgelegt:

- Client und Server laufen auf der *gleichen Maschine*. Die Datenbank wird über einen Dienst oder Server im Adressraum des Clients zur Verfügung gestellt.
- Client und Server laufen auf *verschiedenen Maschinen* und die Kommunikation zwischen beiden geschieht über ein Netzwerk. Die Datenbank wird über einen Datenbankserver zur Verfügung gestellt, der sich in der Regel nicht im gleichen Adressraum befindet.

Bevor die Implementation verwendet werden kann müssen gewisse Voraussetzungen erfüllt sein.

8.2. Voraussetzungen

Die *Datenbank-Persistenz-Implementation* ist ein Teil von *JGraLab* und nicht alleine lauffähig. Zur Verwendung dieser benötigt ein Benutzer neben *JGraLab* noch eine Datenbank. Unterstützt werden die Datenbank-Technologien *MySQL*, *PostgreSQL* und *JavaDB* bzw. *Apache Derby*.
Grundsätzlich muss der Benutzer über Schreib- und Leserechte auf die verwendete Datenbank verfügen.

8.3. Verbindungsaufnahme zur Datenbank

Die Verbindungsaufnahme zur Datenbank erfolgt über *JDBC*. Dazu muss neben einem *Benutzernamen* und dazu passendem *Passwort* auch die *Url* zur Datenbank bekannt sein. Die *Url* setzt sich zusammmen aus:

```
<Präfix>://<Domänenname des Servers>:<Port>/<Datenbankname>
```

Der Domänenname des Servers und der Name der Datenbank können beliebig lauten. Der Präfix ist jedoch abhängig von der eingesetzten Datenbank-Technologie und deshalb nicht frei wählbar. Anhand des Präfix unterscheidet die Implementation die verwendete Datenbanktechnologie.
Die möglichen Präfixe lauten:

- `mysql` für *MySQL*,
- `postgresql` für *PostgreSQL*, und
- `derby` für *JavaDB* bzw. *Apache Derby*.

Der *TCP-Port* unter dem ein Dienst erreichbar ist, kann umkonfiguriert werden. Die Datenbanken sind jedoch in der Regel unter folgenden Standard-Ports erreichbar:

- `3306` für *MySQL*,
- `5832` für *PostgreSQL*, und
- `1527` für *JavaDB* bzw. *Apache Derby*.

Lautet zum Beispiel der Domänenname des Servers `dbserver.uni-koblenz.de` und der Name der Datenbank `chuck`, dann sind folgende *Urls* möglich:

```
mysql://dbserver.uni-koblenz.de:3306/chuck

postgresql://dbserver.uni-koblenz.de:5832/chuck

derby://dbserver.uni-koblenz.de:1527/chuck
```

Die Verbindung zur Datenbank wird in der Klasse `GraphDatabase` verwaltet. Sie befindet ich im Java-Paket `de.uni_koblenz.jgralab.impl.db`. Zur Aufnahme der Verbindung zur Datenbank stellt sie folgende Methode zur Verfügung:

```
openGraphDatabase(String url, String user, String password)
```

Liegt zum Beispiel eine *MySQL-Datenbank* auf dem Server `dbserver.uni-koblenz.de`, der Name der Datenbank lautet `chuck`, der Benutzername `marvin` und das Passwort `towel`, dann gestaltet sich die Verbindungsaufnahme wie folgt:

```
try{
  String url = "mysql://dbserver.uni-koblenz.de:3306/chuck";
  String user = "marvin";
  String password = "towel";
```

```
  GraphDatabase graphDatabase =
    GraphDatabase.openGraphDatabase(url, user, password);
  // nun kann graphDatabase benutzt werden
}
catch(GraphDatabaseException exception){
  exception.printStackTrace();
}
```

Schlägt die Verbindungsaufnahme fehl, wird dies durch eine entsprechende *Ausnahme* angezeigt.
Bevor eine Datenbank jedoch verwendet werden kann, muss sie zur Aufnahme von Graphen vorbereitet werden.

8.4. Datenbank vorbereiten

Handelt es sich um eine leere Datenbank, die noch nicht für die Verwendung mit der *Datenbank-Persistenz-Implementation* vorbereitet wurde, müssen zunächst alle Tabellen in der Datenbank angelegt werden, die nötig sind um Graphen zu speichern. Die Klasse `GraphDatabase` stellt dazu folgende Methode zur Verfügung:

```
applyPhysicalSchema()
```

Wird das oben angeführte Beispiel weitergeführt, dann gestaltet sich das Erzeugen der Tabellen wie folgt:

```
try{
  String url = "mysql://dbserver.uni-koblenz.de:3306/chuck";
  String user = "marvin";
  String password = "towel";
  GraphDatabase graphDatabase =
    GraphDatabase.openGraphDatabase(url, user, password);
  graphDatabase.applyPhysicalSchema();
}
catch(GraphDatabaseException exception){
  exception.printStackTrace();
}
```

Schlägt die Operation fehl, so wird dies durch eine entsprechende *Ausnahme* angezeigt. Da die Datenbank noch leer ist und mit Graphen gefüllt werden soll, ist die Datenbank *zunächst für*

das schnelle Ausführen von Schreiboperationen optimiert.
Bevor ein Graph erzeugt werden kann, muss sein Schema in die Datenbank geladen werden.

8.5. Graphschemata laden

Zu einem persistierten Graphen muss immer das entsprechende Graphschema in der Datenbank verfügbar sein. Jedes Graphschema, dass verwendet werden soll, muss deshalb zunächst in die Datenbank geladen werden.
Die Klasse `GraphIO` aus dem Java-Paket `de.uni_koblenz.jgralab` stellt dazu eine statische Methode zur Verfügung:

```
loadSchemaIntoGraphDatabase(String file, GraphDatabase graphDatabase)
```

Soll zum Beispiel das Graphschema aus Datei `citymapschema.tg` in eine Datenbank geladen werden und besteht bereits eine Verbindung über `graphDatabase`, so gestaltet sich der Vorgang wie folgt:

```
try{
  GraphIO.loadSchemaIntoGraphDatabase("citymapschema.tg", graphDatabase);
}
catch(Exception exception){
  exception.printStackTrace();
}
```

Schlägt die Operation fehl, so wird dies durch eine entsprechende *Ausnahme* angezeigt.

Bevor ein Graph in der Datenbank erzeugt werden kann, muss das Schema *committed* werden. Bei einem *Commit* werden Java-Klassen aus den definierten Typen im Schema und für das Schema selbst erzeugt. Dies kann entweder mit dem Schema aus einer TG-Datei geschehen (siehe dazu das JGraLab-Tutorial [fST10]) oder durch laden und committen des Schemas aus der Datenbank heraus. `GraphIO` stellt dazu folgende Methode bereit:

```
loadAndCommitSchemaFromDatabase(
  GraphDatabase graphDatabase,
  String packagePrefix,
  String name,
  String path
)
```

Sollen die Java-Klassen für das Graphschema `citymap.schema.CityMapSchema` im Ordner `classes` erzeugt werden und besteht bereits eine Verbindung über `graphDatabase` zur Datenbank, so gestaltet sich der Aufruf auf folgende Weise:

```
try{
  GraphIO.loadAndCommitSchemaFromDatabase(
    graphDatabase, "citymap.schema", "CityMapSchema", "classes"
  );
}
catch(Exception exception){
  exception.printStackTrace();
}
```

8.6. Erzeugen von Graphen

Sobald die Datenbank vorbereitet, ein passendes Graphschema in ihr persistiert und Schema *committed* ist, kann ein *datenbankpersistenter Graph* erzeugt werden. Auf folgende Weise wird ein Graph `Koblenz` vom Typ `CityMap` in der Datenbank angelegt:

```
try{
  CityMap cityMap = CityMapSchema.instance()
    .createCityMapWithDatabaseSupport("Koblenz", graphDatabase);
  // nun können Knoten und Kanten erzeugt werden
}
catch(Exception exception){
  exception.printStackTrace();
}
```

Knoten und Kanten werden analog zur klassischen *In-Memory-Implementation* erzeugt. Eine genaue Betrachtung kann dem *JGraLab Tutorial* [fST10] entnommen werden.

8.7. Laden bereits existierender Graphen

Sind in der Datenbank bereits Graphen enthalten, so kann der Benutzer diese laden. Eine Liste der in einer Datenbank enthaltenen Graphen kann über `getIdsOfContainedGraphs()` aus der Klasse `GraphDatabase` abgefragt werden.
Zum Laden eines Graphen aus einer Datenbank stellt die Klasse `GraphIO` eine statische Methode bereit:

```
loadGraphFromDatabase(String id, GraphDatabase graphDatabase)
```

Soll zum Beispiel der Graph `Koblenz` aus einer Datenbank geladen werden und besteht bereits eine Verbindung über `graphDatabase`, so gestaltet sich der Vorgang wie folgt:

```
try{
  Graph graph =
    GraphIO.loadGraphFromDatabase("Koblenz", graphDatabase);
  // nun kann der Graph verwendet werden
}
catch(Exception exception){
  exception.printStackTrace();
}
```

Schlägt die Operation fehl, so wird dies ebenfalls durch eine entsprechende *Ausnahme* angezeigt.

8.8. Leistungsverbessernde Maßnahmen

Optimierung für Schreiboperationen. Soll ein Graph erzeugt werden, so empfiehlt es sich die Datenbank zuvor für Schreiboperationen zu optimieren. Dazu bietet die Klasse `GraphDatabase` folgende Methode an:

```
optimizeForWrite();
```

Durch Aufruf der Methode werden *Primär- und Fremdschlüsselbeschränkungen der Tabellen für Knoten, Kanten, Inzidenzen und Attributbelegungen aufgehoben*. Zusätzlich werden *alle definierten Indizes auf diese Tabellen gelöscht*. Damit entfallen beim Einfügen von Daten Einschränkungsüberprüfungen und die Aktualisierung der Indizes. Je nach verwendeter Datenbank-Technologie wird zusätzlich der *Auto-Commit-Modus von JDBC deaktiviert*. Somit muss der Client nicht darauf warten bis eine Datenbank die eingefügten Daten auch tatsächlich physikalisch persistiert hat. Insgesamt können Graphelemente auf diese Weise schneller in der Datenbank erzeugt werden.

Optimierung für Leseoperationen. Soll ein Graph traversiert werden, so empfiehlt es sich die Datenbank zuvor für Leseoperationen zu optimieren. Dazu bietet die Klasse `GraphDatabase` folgende Methode an:

```
optimizeForRead();
```

Durch Aufruf der Methode werden *Primär- und Fremdschlüsselbeschränkungen der Tabellen für Knoten, Kanten, Inzidenzen und Attributbelegungen angelegt*. Zusätzlich werden *Indizes auf diese Tabellen angelegt*. Zusätzlich werden je nach verwendeter Datenbank-Technologie die Datensätze in den Tabellen *geclustert*. Unter *Clustering* versteht man die Ballung logisch verwandter Datensätze zur Zugriffsbeschleunigung. Dabei werden Daten, die häufig zusammen benötigt werden, dicht beieinander auf dem Hintergrundspeicher abgelegt. Im Idealfall liegen sie dann auf der gleichen Seite [KE09]. Insgesamt können Graphen auf diese Weise schneller traversiert werden.

8.9. Speicherreservierung durch Java-VM und Datenbankverwaltungssystem

Graphen in Datenbanken können im Gegensatz zu Graphen aus TG-Dateien fast beliebig groß werden, ohne das der physikalische Hauptspeicher überläuft, da nur die jeweils gerade relevanten Teile im Hauptspeicher gehalten werden. Idealerweise sollte sich der Graph immer im schnellen physischen Arbeitsspeicher des Rechners befinden und nicht durch das Betriebssystem ausgelagert werden. Da jedoch kein direkter Einfluss auf die Entscheidung zur Auslagerung durch das Betriebssystem genommen werden kann, darf die Caching-Strategie der Auslagerungsstrategie des Betriebssystems nicht zuwiderlaufen.
Damit dies nicht bereits durch eine ungünstige Startkonfiguration der Java-VM geschieht, muss der Benutzer folgende Regeln beachten:

1. Der Benutzer kann beim Start einer Java-VM den verfügbaren Speicherplatz auf dem Heap für diese festlegen. Dabei sollte er *nicht mehr reservieren als schneller physikalischer Speicher im Rechner installiert ist*, denn sonst werden auf jeden Fall Teile des Graphen ausgelagert.
2. Auf einem Rechner laufen in der Regel mehrere Prozesse, die ebenfalls Arbeitsspeicher benötigen. Es muss deshalb immer weit weniger als die zuvor angeführte Obergrenze reserviert werden, da anderen Prozessen auch schneller physikalischer Arbeitsspeicher gelassen werden sollte. Müssen ständig Daten anderer Prozesse ein und ausgelagert werden, wird die Arbeit mit dem Graphen ebenfalls langsam bis sogar unmöglich.

Laufen Clientanwendung und Datenbankserver auf dem selben Rechner, so muss zusätzlich die Speicherauslastung durch das verwendete Datenbankverwaltungssystem beachtet und ggf. konfiguriert werden. Da die Betrachtung den Rahmen der Arbeit sprengt, wird an dieser Stelle nur auf adäquate Literatur verwiesen. Neben entsprechenden Kapiteln in den Handbüchern

der unterstützten Datenbanktechnologien ([MDT10], [Mom01] und [ADP10]) können [Köh10] für *MySQL*, [Wil10] für *PostgreSQL* und [San10] für *JavaDB* bzw. *Apache Derby* als Ausgangspunkt für die Betrachtung des Performanzverhaltens herangezogen werden.

Im nächsten Kapitel wir die die Implementation bewertet.

9. Bewertung der Implementation

In diesem Kapitel wird die umgesetzte Lösung bewertet. Konkret wird das Laufzeitverhalten durch Zeitmessungen für das Erzeugen und Traversieren von Graphen bewertet.
Die Zeitmessungen werden an der klassischen *In-Memory-Implementation* aus dem Java-Paket `de.uni_koblenz.jgralab.impl.std` und der neu implementierten *Datenbank-Persistenz-Implementation* aus `de.uni_koblenz.jgralab.impl.db` vorgenommen. Die ermittelten Ergebnisse werden gesammelt und miteinander verglichen.

9.1. Konfiguration des Testsystems

Das Testsystem besitzt die folgende Hardware-Ausstattung:

- Prozessor: *Athlon64 X2 5200+*
- Hauptspeicher: 2GB DDR2-RAM
- Festplattengröße: 250GB
- Betriebssystem: *Windows XP SP3*
- *JVM 1.6.0_21* mit Standardeinstellung für initiale Heap-Größe

Gemessen wird das Einsatzszenario *Client und Server laufen auf der gleichen Maschine*. *JGraLab* und das Datenbankverwaltungssystem laufen auf dem selben Rechner.

Auf die Messung des Einsatzszenarios *Client und Server laufen auf verschiedenen Maschinen* wird verzichtet. Probemessungen ergeben, dass die Zeitaufwände aufgrund der Latenzen über das Netzwerk erwartungsgemäß höher ausfallen, die Gesamtentwicklung der Messergebnisse sich analog zum Einsatzszenario *Client und Server laufen auf der gleichen Maschine* verhält.

Zur Bestätigung der erhaltenen Ergebnisse werden zusätzlich Probemessungen auf Rechnern mit anderen Hardware-Konfigurationen vorgenommmen. Die erhaltenen Messergebnisse verhalten sich analog zu den Messergebnissen, die im nächsten Abschnitt angeführt werden.

9.2. Verwendete Graphen

Die verwendeten Graphen werden durch den *Java-Faktenextraktor für Gupro* [BV08] erzeugt. Dieser extrahiert Graphen aus Java-Quelltexten. Die erzeugten Graphen repräsentieren die feingranulare Syntax der geparsten Java-Quelltexte. Der *Java-Faktenextraktor* verwendet das Graphschema `de.uni_koblenz.jgralab.grabaja.java5schema.Java5Schema`.
Die Anzahl der Knoten und Kanten in einem solchen Graph steigt erfahrungsgemäß mit der Anzahl der geparsten Quelltextzeilen. Erzeugte Graphen weisen in der Regel eine baumartige Struktur auf.
Es wurden drei verschiedene Graphen erzeugt und verwendet:

- ein kleiner Graph mit 312 Knoten und 400 Kanten extrahiert aus einer Klasse des Java-Extraktors:
  `de.uni_koblenz.jgralab.grabaja.extractor.Utilities.java`
- ein mittlerer Graph mit 5590 Knoten und 7317 Kanten extrahiert aus einer Klasse der Datenbank-Persistenz-Implementation:
  `de.uni_koblenz.jgralab.impl.db.GraphDatabase.java`
- ein großer Graph mit 143783 Knoten und 217278 Kanten extrahiert aus den Quelltexten des Parser-Generators *ANTLR 2.7.6*

9.3. Messergebnisse

Erzeugung. Zunächst werden Graphen verschiedener Größe *erzeugt und persistiert*. Der Vorgang wird dreimal durchgeführt und die gemittelten Messergebnisse sind in Tabelle 9.1 festgehalten. Aufgeführt sind der mittlere Zeitaufwand der verwendeten Implementationen über die Zahl der Durchläufe in ms. Die Ergebnisse der klassischen *In-Memory-Implementation* sind in der Spalte `impl.db` aufgeführt. Die Ergebnisse der *Datenbank-Persistenz-Implementation* sind in drei Spalten aufgeführt. Die Spalten tragen jeweils die Namen der verwendeten Datenbank-Technologien.

Graphgröße	**`impl.std`**	**MySQL**	**PostgreSql**	**JavaDB**
klein	49 ms	14.305 ms	3.923 ms	6257 ms
mittel	333 ms	2.505.274 ms	91.576 ms	290.672 ms
groß	6.988 ms	n. a.	n. a.	n. a.

Tabelle 9.1. – Zeitaufwände für das Erzeugen und Persistieren von Graphen verschiedener Größe.

Die Messergebnisse zeigen, dass der Zeitaufwand von Schreib-Operationen auf dem Graph mit der *Datenbank-Persistenz-Implementation* grundsätzlich um ein Vielfaches höher liegt, als mit der *In-Memory-Implementation*. Zudem schwankt der Zeitaufwand stark, abhängig von der einge-

setzten Datenbank-Technologie.
Der große Graph konnte nicht mit der *Datenbank-Persistenz-Implementation* erzeugt werden, da der Vorgang mit jeder der eingesetzten Datebanken nach mehreren Stunden noch nicht abgeschlossen war und deshalb abgebrochen wurde. Werden die bereits ermittelten Zeitaufwände für die Erzeugung des kleinen und mittleren Graphen anhand ihrer Knoten- und Kantenanzahl hochgerechnet, so sollte die Erzeugung des großen Graphen mit *MySQL* etwa 19 Stunden, mit *PostgreSql* etwa 45 Minuten und mit *JavaDB* etwa 3 Stunden dauern.
Am schnellsten kann ein Graph mit *PostgreSql* erzeugt werden.

Traversierung. Als nächstes werden die zuvor erzeugten Graphen *komplett traversiert*. Komplett bedeutet, dass jeder Knoten und jede Kante mindestens einmal besucht wird. Die Traversierung wird jeweils mit einer *Tiefensuche* (in Tabelle *dfs*) und einer *Breitensuche* (in Tabelle *bfs*) durchgeführt. Diese Algorithmen traversieren den Graph komplett, da das gesuchte Element nicht in ihm vorhanden ist[1].
Der Vorgang wird jeweils dreimal durchgeführt und die gemittelten Messergebnisse sind in Tabelle 9.2 festgehalten. Die Traversierung eines großen Graphen entfällt für die *Datenbank-Persistenz-Implementation*.

Graphgröße	**`impl.std`**	**MySQL**	**PostgreSql**	**JavaDB**
klein	dfs 1,66 ms bfs 0,79 ms	dfs 2325,16 ms bfs 2258,39 ms	dfs 324,70 ms bfs 321,50 ms	dfs 278,25 ms bfs 247,48 ms
mittel	dfs 5,92 ms bfs 8,04 ms	n. a.	dfs 9317,94 ms bfs 11058,82 ms	dfs 7996,44 ms bfs 9639,03 ms
groß	dfs 137,65 ms bfs 672,92 ms	n. a.	n. a.	n. a.

Tabelle 9.2. – Messergebnisse für das *komplette* Traversieren von Graphen verschiedener Größe mit Tiefen- und Breitensuche.

Auch diese Messungen zeigen, dass der Zeitaufwand von Lese-Operationen auf dem Graph mit der *Datenbank-Persistenz-Implementation* grundsätzlich um ein Vielfaches höher liegt, als mit der *In-Memory-Implementation*.
Als nächstes werden die zuvor erzeugten Graphen *teilweise traversiert*. Auch hier kommen *Tiefensuche* und *Breitensuche* zum Einsatz. Diesmal finden die Algorithmen jedoch das gesuchte Element.

Der Vorgang wird jeweils dreimal durchgeführt und die gemittelten Messergebnisse sind in Tabelle 9.3 festgehalten.

[1] Für die genaue Implementation der Suchalgorithmen siehe Anhang C

Graphgröße	`impl.std`	MySQL	PostgreSql	JavaDB
klein	dfs 0,95 ms bfs 0,26 ms	dfs 590,40 md bfs 720,80 ms	dfs 88,13 ms bfs 99,90 ms	dfs 195,88 ms bfs 221,17 ms
mittel	dfs 4,85 ms bfs 1,29 ms	n. a.	dfs 5659,65 ms bfs 2512,24 ms	dfs 4730,74 ms bfs 2016,47 ms
groß	dfs 41,81 ms bfs 30,00 ms	n. a.	n. a.	n. a.

Tabelle 9.3. – Messergebnisse für das *teilweise* Traversieren von Graphen verschiedener Größe mit Tiefen- und Breitensuche.

Die Messergebnisse verhalten sich analog zu den Messergebnissen der kompletten Traversierung.

Die Traversierung des mittelgroßen Graphen in der *MySQL-Datenbank* fand auch nach mehreren Stunden keinen Abschluss und wurde deshalb abgebrochen. Da zuvor kein großer Graph in der *JavaDB* und der *MySQL-Datenbank* persistiert werden konnten, entfiel die Traversierung. Am schnellsten kann ein Graph in einer *PostgreSql-Datenbank* traversiert werden.

Zur Bestätigung der erhaltenen Ergebnisse wurde die Messungen auch auf Rechnern mit anderen Hardware-Konfigurationen vorgenommmen. Die erhaltenen Messergebnisse verhalten sich analog zu den bereits angeführten und werden deshalb nicht aufgelistet.

9.4. Einzelmessungen

Um festzustellen wie hoch der Zeitaufwand für einzelne Vorgänge mit der *Datenbank-Persistenz-Implementation* ausfällt, werden zusätzlich Messungen der Einzeloperationen vorgenommen. Jede Operation wurde 10000 mal ausgeführt und gemessen. Die gemittelten Messergebnisse sind in Tabelle 9.4 zusamengefasst.

Operation	MySQL	PostgreSql	JavaDB
Leeren Java5-Graph erzeugen	127,92 ms	149,36 ms	140,42 ms
Knoten ohne Attribute erzeugen	0,11 ms	0,39 ms	0,28 ms
Attribut erzeugen	0,16 ms	0,15 ms	0,09 ms
Kante mit Inzidenzen und ohne Attribute erzeugen	7,2 ms	2,70 ms	9,73 ms
Graph laden mit 2 Knoten und 10000 Kanten	29,06 ms	226,42 ms	23,65 ms
Knoten aus $Vseq$ abrufen	1,62 ms	0,92 ms	1,44 ms
Kante aus $Eseq$ abrufen	0,85 ms	3,81 ms	17,6 ms
Kante aus Λseq abrufen	0,51 ms	1,81 ms	17,2 ms

Tabelle 9.4. – Messergebnisse für Einzeloperationen auf dem Graph.

In den Einzelmessungen liegen die Ergebnisse der verschiedenen Datenbank-Technologien nicht so weit auseinander, wie es noch bei der Erzeugen und Traversierung eines kompletten Graphen der Fall ist.
Die Einzeloperationen konnten unter idealen Bedingungen durchgeführt werden. Vor Schreiboperationen wird die Datenbank jeweils mit `optimizeForWrite()` dafür vorbereitet - analog geschieht dies für Leseoperationen mit `optimizeForRead()`. Bei der Erzeugung eines kompletten Graphen müssen jedoch neben Schreib- auch Leseoperationen durchgeführt werden. Die Ergebnisse der Einzelmessung sind somit nicht unmittelbar auf komplexere Graphen anwendbar.

9.5. Zusammenfassung der Messergebnisse

Erwartungsgemäß fällt das Erzeugen und Traversieren von Graphen mit der *Datenbank-Persistenz-Implementation* langsamer aus, als mit der klassischen *In-Memory-Implementation*.
Jedoch fallen Erzeugung und Traversierung im Vergleich sehr langsam aus. So konnten keine Graphen erzeugt werden, die nicht komplett in den Hauptspeicher eines handelsüblichen Rechners passen. Auch wurde auf das Erzeugen von Graphen mit anderen Graphschemata verzichtet.
Insgesamt schlägt sich das Datenbanksystem *PostgreSql* am besten. Für Erzeugung und Traversierung von kompletten Graphen ist es das schnellste der unterstützten Systeme und verhält sich zuverlässiger, als *JavaDB* und *MySQL*. Die Messergebnisse schwanken zwischen den Läufen weit weniger als es mit *JavaDB* und *MySQL* der Fall ist. Zudem steigt der Zeitaufwand mit *PostgreSql* mit der Größe der Graphen weniger stark. Das mag daran liegen, dass *PostgreSql* selbst bei reiner Optimierung für Schreiboperationen dennoch eine akzeptable Leseleistung bietet.

10. Fazit

Dieses Kapitel widmet sich der abschließenden Betrachtung der Entwicklung der Gesamtlösung. Zunächst werden die Ergebnisse der Diplomarbeit zusammengefasst. Danach werden die erfüllten Anforderungen bewertet und anschließend mögliche Weiterentwicklungen aufgezeigt.

10.1. Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde eine Datenbank-Persistenzlösung für die TGraphenbibliothek *JGraLab* entwickelt. Die Lösung erlaubt das Persistieren von TGraphen in Datenbanken ausgewählter Technologien und sorgt gleichzeitig dafür, dass die Datenbankpersistenz eines TGraphen für den Benutzer transparent bleibt.

Nach der Anforderungserhebung und der Recherche zur Bewertung der Tauglichkeit von bereits bestehenden Werkzeugen für dieses Projekt wurden drei Datenbank-Technologien zur Unterstützung ausgewählt. Anschließend wurde dargelegt wie die Persistierung von TGraphen mit allen ihren Eigenschaften in Datenbanken ermöglicht werden kann.
Anschließend wurde der konzeptuelle Entwurf vorgenommen und die Details der Lösung beschrieben. Dann wurde der objektorientierte Feinentwurf zur Integration der Lösung in die TGraphenbibliothek *JGraLab* entwickelt.
Der objektorientierte Feinentwurf wurde implementiert und die finale Implementation einer Leistungsbewertung unterzogen.

10.2. Umgesetzte Anforderungen

Die meisten Anforderungen an die Gesamtlösung konnten vollständig erfüllt werden. Eine optionale Anforderung konnte nur *partiell erfüllt* werden. Diese fordert, dass mehrere Datenbankschemata entwickelt werden sollen. Es wurde das generische relationale Schema entwickelt und umgesetzt. Zusätzlich wurde die Entwicklung weiterer mögliche Schemata skizziert.

Nur eine Anforderung konnte *nicht erfüllt* werden. Diese ist jedoch eine der Hauptanforderungen an die Lösung. Sie fordert, dass die Traversierung eines Graphen, der in einer Datenbank

persistiert wurde *hinreichend* schnell sein soll.
Zunächst dauert bereits das Erzeugen von großen Graphen, die nicht komplett in den Hauptspeicher eines handelsüblichen Rechners passen, mit der neuen Implementation noch zu lange. Aus diesem Grund wurden zur Leistungsbewertung keine solchen Graphen erzeugt. Stattdessen wurden vergleichsweise kleine Graphen zur Leistungsbewertung herangezogen.
Die Zeitmessungen ergeben, dass der Zeitaufwand für eine Traversierung mindestens um den Faktor 200 höher ausfällt, als es mit der klassischen *In-Memory-Implementation* der Fall ist.

10.3. Ausblick

Erwartungsgemäß ist die *Datenbank-Persistenz-Implementation* langsamer als die klassische *In-Memory-Implementation*. Für die Arbeit mit großen Graphen ist sie jedoch *noch* zu langsam. Die Weiterentwicklung der Implementation sollte deshalb ihren Fokus auf die Beschleunigung von Erzeugung und Traversierung der Graphen in Datenbanken legen.

Folgende Ansätze sind denkbar:

- *Graphschemaspezifische relationale Schemata* - dieser Ansatz zur wird bereits in Kapitel 5.4 behandelt. Da der Aufwand für das Persistieren und Abrufen von Graphelementen dieser Schemata geringer ist, als mit dem generischen relationalen Schema, sollte dieser Ansatz in eine Weiterentwicklung mit einfließen.
- *Einsatz von SSDs* - durch den Einsatz von *Solid State Disks* kann die Zugriffsgeschwindigkeit im Vergleich zu herkömmlichen Festplatten um eine Größenordnung höher ausfallen.
- *Weitere Datenbank-Technologien* - es existieren eine Vielzahl von weiteren Datenbanken, die im Rahmen der Arbeit nicht betrachtet werden konnten. Neben relationalen, objektrelationalen und objektorientierten Datenbanken existieren weitere Datenbank-Konzepte, die den Anspruch haben, effizienter als die drei genannten zu sein [End10].

Literaturverzeichnis

[Ach06] ACHILLES, ALBRECHT: *Betriebssysteme: Eine kompakte Einführung mit Linux*. Springer, Berlin, 2006.

[ADP10] APACHE DB PROJECT, THE: *Apache Derby: Documentation*. Abruf am 24. September 2010. `http://db.apache.org/derby/manuals/index.html`.

[ASF10a] APACHE SOFTWARE FOUNDATION, THE: *Java Data Objects*. Abruf am 3. Februar 2010. `http://db.apache.org/jdo/`.

[ASF10b] APACHE SOFTWARE FOUNDATION, THE: *Derby*. Abruf am 5. September 2010. `http://db.apache.org/derby/`.

[Bar09] BARTIG, WILLIAM: *The Open Source Database Benchmark*. Abruf am 22. Dezember 2009. `http://osdb.sourceforge.net/`.

[Bee06] BEEGER, ROBERT F.: *Persistenz in Java-Systemen mit Hibernate 3*. dpunkt-Verlag, Persistenz in Java-Systemen mit Hibernate 3, 2006.

[BEJ+00] BERLER, MARK, JEFF EASTMAN, DAVID JORDAN, CRAIG RUSSELL, OLAF SCHADOW, TORSTEN STANIENDA und FERNANDO VELEZ: *The object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[Ber73] BERGE, CLAUDE. North Holland Publ. Comp., Amsterdam, 1973.

[BER+08] BILDHAUER, DANIEL, JÜRGEN EBERT, VOLKER RIEDIGER, HANNES SCHWARZ und SASCHA STRAUSS: *grUML - Eine UML-basierte Modellierungssprache für TGraphen (Version 0.92 vom 24.01.08)*. Institut für Softwaretechnik, 2008.

[Bra94] BRANDSTÄDT, ANDREAS: *Graphen und Algorithmen*. Teubner, Stuttgart, 1994.

[BV08] BALDAUF, ARNE und NICOLAS VIKA: *Java-Faktenextraktor für Gupro*. Universität Koblenz-Landau, Koblenz, 2008. Studienarbeit.

[CDN93] CAREY, MICHAEL J., DAVID J. DEWITT und JEFFREY F. NAUGHTON: *The 007 Benchmark*. SIGMOD Rec., 22(2):12–21, 1993.

[Cod83] CODD, E. F.: *A relational model of data for large shared data banks*. Commun. ACM, 26(1):64–69, 1983.

[DW98] DAHM, PETER und FRIEDBERT WIDMANN: *Das Graphenlabor*. Koblenz, 1998.

[Ebe10] EBERT, AG: *Homepage der Arbeitsgruppe Ebert*. Abgerufen am 15. September 2010. http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/ag_ebert_home/.

[EGSW98] EBERT, JÜRGEN, RAINER GIMNICH, HANS H. STASCH und ANDREAS WINTER: *Gupro: Generische Umgebung zum Programmverstehen*. Fölbach, Koblenz, 1998.

[End10] ENDLICH, STEFAN: *Aufstand der NoSQL-DBs*. database pro, 2010(1):31–33, 2010.

[FSF10] FREE SOFTWARE FOUNDATION, THE: *GNU GENERAL PUBLIC LICENSE - Version 3, 29 June 2007*. Abgerufen am 5. Januar 2010. http://www.gnu.org/licenses/gpl-3.0.txt.

[fST10] SOFTWARE-TECHNIK, INSTITUT FÜR: *JGraLab Tutorial*. Abgerufen am 26. September 2010. http://userpages.uni-koblenz.de/~ist/JGraLab_Tutorial.

[Gep02] GEPPERT, ANDREAS: *Objektrelationale und objektorientierte Datenbankkonzepte und -systeme*. dpunkt.verlag GmbH, Heidelberg, 2002.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Har01] HARTWIG, JENS: *PostgreSQL Professionell und praxisnah*. Addison-Wesley Verlag, München, 2001.

[Hun05] HUNT, ANDREW; THOMAS, DAVE: *Unit-Tests mit JUnit*. Hanser Fachbuchverlag, München, Wien, 2005.

[JS10] JAVA-SOURCE.NET: *Open Source Persistence Frameworks in Java*. Abruf am 20. Januar 2010. http://java-source.net/open-source/persistence.

[Kah06] KAHLE, STEFFEN: *JGraLab: Konzeption, Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*. Universität Koblenz-Landau, Koblenz, 2006. Diplomarbeit.

[KE09] KEMPER, ALFONS und ANDRE EICKLER: *Datenbanksysteme - Eine Einführung, 7. aktualisierte und erweiterte Auflage*. Oldenbourg Wissenschaftsverlag GmbH, München, 2009.

[Köh10] KÖHNTOPP, KRISTIAN: *MySQL Performance Tuning*. MySQL GmbH, Abruf am 22. September 2010. http://blog.koehntopp.de/uploads/PT_isotopp_deutsch.pdf.

[KS09] KRÜGER, GUIDO und THOMAS STARK: *Handbuch der Java-Programmierung, Standard Edition Version 6, 5. Auflage*. Addison-Wesley Verlag, München, 2009.

[LD04] LOCKEMANN, PETER C. und KLAUS R. DITTRICH: *Architektur von Datenbanksystemen*. dpunkt.verlag GmbH, Heidelberg, 2004.

[Lit92] LITAUER, CHRISTOPH. Universität Koblenz-Landau, 1992.

[Län98] LÄNGER, MARTIN: *Entwicklung, Implementierung und Bewertung einer objektorientierten, datenbankgestützten Java-Klassenbibliothek für Graphen*. Universität Koblenz-Landau, Koblenz, 1998. Diplomarbeit.

[Mar06] MARCHEWKA, KATRIN: *Entwurf und Definition der Graphanfragesprache GReQL 2*. Universität Koblenz-Landau, Koblenz, 2006. Diplomarbeit.

[Mar09] MARTIN, ROBERT C.: *Clean Code - Refactorings, Patterns, Testen und Techniken für sauberen Code, 1. Auflage*. mitp-Verlag, Heidelberg, München, Landsberg, Frechen, Hamburg, 2009.

[MB09] MONTE BARETTO, JOSÉ ANGEL: *Transaktionskonzept für die TGraphenbibliothek JGraLab*. Universität Koblenz-Landau, Koblenz, 2009. Diplomarbeit.

[MDT10] MYSQL DOCUMENTATION TEAM, THE: *MySQL 5.1 Referenzhandbuch*. Sun Microsystems, 2010. http://downloads.mysql.com/docs/refman-5.1-de.a4.pdf.

[Mic10] MICROSYSTEMS, SUN: *Java Persistence API*. Abruf am 3. Februar 2010. http://java.sun.com/javaee/technologies/persistence.jsp?intcmp=3282.

[Mom01] MOMJIAN, BRUCE: *PostgreSQL: introduction and concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. http://www.postgresql.org/files/documentation/books/aw_pgsql/index.html.

[Net10a] NETWORK, SUN DEVELOPER: *JDBC API Specifications*. Oracle, Abruf am 10. September 2010. http://java.sun.com/products/jdbc/download.html#corespec40.

[net10b] NETZZEITUNG.DE: *EU stellt sich gegen Sun-Oracle-Deal*. Abruf am 20. Januar 2010. http://www.netzeitung.de/nachrichten_archiv/1512602_51_EU_stellt_sich_gegen_Sun-Oracle-Deal.html.

[Nic10] NICHOLAS, ETHAN: *Understanding Weak References*. Abgerufen am 10. Juni 2010. http://weblogs.java.net/blog/2006/05/04/understanding-weak-references.

[Pie07] PIENTKA, FRANK: *Einführung in das Java-Datenbanksystem Derby*. Datenbank-Spektrum, 2007(21):43–50, 2007. http://www.datenbank-spektrum.de/pdf/dbs-21-42.pdf.

[PT10] POSTGRESQL-TEAM, THE: *Homepage von PostgreSQL*. Abruf am 7. Januar 2010. http://www.postgresql.org/.

[Ras05] RASCHKE, TANJA: *Vergleichende Betrachtung von Verfahren zur Entwicklung Datenbank-gestützter objektorientierter Informationssysteme*. Universität Koblenz-Landau, Koblenz, 2005. Studienarbeit.

[Reu06] REUSSNER, RALF;HASSELBRING, WILHELM: *Handbuch der Software-Architektur*. dpunkt-Verlag, Heidelberg, 2006.

[Rod10] RODEN, GOLO: *Frei und Business-tauglich*. database pro, 2010(1):68–73, 2010.

[RQZ05] RUPP, CHRIS, STEFAN QUEINS und BARBARA ZENGLER: *UML 2 glasklar*. Hanser Fachbuchverlag, München, 2005.

[San10] SANDSTÅ, OLAV: *Configuring Apache Derby for Performance and Durability*. Sun Microsystems, Database Technology Group, Abruf am 10. September 2010. `http://db.apache.org/derby/binaries/DerbyPerfDurability-2006.pdf`.

[SM99] STONEBRAKER, MICHAEL und DOROTHY MOORE: *Objektrelationale Datenbanken - Die nächste große Welle*. Hanser Verlag, München; Wien, 1999.

[TG01] TANENBAUM, ANDREW S. und JAMES GOODMAN: *Computerarchitektur. Strukturen, Konzepte, Grundlagen - 4. Auflage*. Pearson Studium, München, 2001. `http://www.gupro.de/~ist/JGraLab_Tutorial`.

[TPPC09] TRANSACTION PROCESSING PERFORMANCE COUNCIL, THE: *Transaction Processing Performance Council*. Abruf am 23. Dezember 2009. `http://www.tpc.org/tpcc/default.asp`.

[VER10] VERSANT: *Homepage von db4objects*. Abgerufen am 4. Januar 2010. `http://www.db4o.com/`.

[Wil10] WILES, FRANK: *Performance Tuning PostgreSQL*. Revolution Systems, LLC, Abruf am 22. September 2010. `http://www.revsys.com/writings/postgresql-performance.html`.

A. Recherchierte nicht unterstützte Persistenz-Lösungen

In diesem Anhang werden alle näher untersuchten Lösungen betrachtet, die in die engere Auswahl kamen, aber letztlich doch nicht für eine Unterstützung durch die Implementation in Frage kamen.

A.1. db4o

Produkt	db4o
Hersteller	VERSANT
Homepage	`http://www.db4o.com/`
Aktuelle Version	7.12
Unterstützte Plattformen	durch Java alle, mit .NET nur Windows
Implementierungssprache	Java, .NET
Maximale Datenbankgröße	256 GByte
Maximale Tabellengröße	entfällt
Zielarchitektur	Client-Server, Embedded
Datenabfrage über	QBE, Criteria Query, nativ
Lizenz	GPL, kommerziell
Status	aktiv

Charakterisierung. Bei *db4o* (kurz für *database for objects*) handelt es sich um eine Objektdatenbank. Sie zeichnet sich durch einen geringen Speicherbedarf aus und zielt besonders auf eine möglichst einfache Verwendung in Programmen und der Einbettung in Umgebungen mit beschränkten Ressourcen, wie mobilen Endgeräten ab. db4o implementiert nicht den ODMG3.0-Standard.
Zur Abfrage von Objekten bietet es *Query by example (QBE)*, *Criteria Queries* und native Abfragen in *Java* und *.NET*.

Historie. Die Entwicklung von *db4o* begann im Jahr 2000 und wurde ab 2004 von dem Unternehmen *db4objects, Inc.* übernommen. 2008 trennte sich *db4objects Inc.* von seiner Datenbanksparte und verkaufte diese an die Firma *VERSANT*.

Zukunftssicherheit. Die Zukunft von *db4o* erscheint als *relativ sicher*. Es befindet sich aktiv in der Weiterentwicklung und wird in vielen kommerziellen Lösungen eingesetzt. Zudem existiert eine große Entwicklergemeinde um das Projekt.

Dokumentationsgrad. Der Dokumentationsgrad zu *db4o* kann als *gut* bewertet werden. Über die Homepage des Projekts sind neben dem *Referenzhandbuch* auch *Tutorials*, *Artikel*, *Anleitungen* und Informationen aus einer *Knowledge Base* abrufbar. Zudem existieren eine handvoll Bücher zum Thema *db4o*. Keines davon ist komplett online verfügbar.
Wie bereits oben angeführt existiert eine große Community um das Projekt.

Lizenz. Je nach Einsatz sieht der Hersteller verschiedene Lizenzen vor. Eine kommerzielle Lizenz ist zu erwerben, wenn *db4o* in einem kommerziellen Produkt eingebettet werden soll. Entgeltfrei unter der *GPL-Lizenz* [FSF10] ist es verwendbar, wenn db4o in ein Produkt, welches ebenfalls unter dieser Lizenz steht, eingebettet wird.

Speicherung von TGraphen. Da ganze Objekte gespeichert werden können, sollten auch komplette TGraphen, die schließlich aus Objekten bestehen, persistiert werden können. Die entsprechenden Klassen müssen dazu entweder mit einer speziellen *Annotation* versehen oder in einer Konfigurationsdatei registriert werden. Alle referenzierten Objekte werden ebenfalls persistiert, wenn ihre Klasse ebenfalls annotiert oder registriert sind[1]. Somit ist es im Gegensatz zu relationalen Datenbanken nicht notwendig ein vollständiges Datenmodell mit Constraints zu definieren.

Speicherverhalten. Zu diesem Kriterium konnten keine Informationen gefunden werden.

Serverseitige Caching- und Zugriffsstrategien. *db4o* erlaubt das Definieren von Indizes. Es werden jedoch nur eindimensionale Indizes (d. h. über ein Attribut eines Objekts) unterstützt. Heuristiken zum Tuning der Datenbank werden im Referenzhandbuch erläutert.
db4o sieht einen clientseitigen Objekt-Cache vor. Dieser kann deaktiviert werden, um eine eigene Implementierung zu verwenden.

[1] *Persistence by Reachability.*

Da der Quelltext von *db4o* frei verfügbar ist, kann generell auf oben angeführte Eigenschaften Einfluss genommen werden. So können modifizierte Varianten kompiliert und eingesetzt werden.
Ein Nachteil ist hingegen, dass die *maximale Datenbankgröße auf 256 GByte beschränkt* ist. So können keine größeren Graphen gespeichert werden.

A.2. Persistenz-Frameworks

A.2.1. Bewertungskriterien

Betrachtete Persistenz-Frameworks wurden ebenfalls hinsichtlich ihrer Eignung zur Erfüllung der Anforderungen untersucht. Folgende Kriterien werden deshalb zur Bewertung herangezogen:

Clientseitiges Caching. In der Regel bringen Persistenz-Frameworks eigene Caching-Funktionen mit. *JGraLab* sollte dennoch die Möglichkeit haben, Einfluss auf diese zu nehmen.

Implementierungssprache Um das Framework direkt in *JGraLab* nutzen zu können, muss es in Java implementiert sein. Frameworks, die dieses Kriterium nicht erfüllen, wurden nicht betrachtet.

A.2.2. Hibernate

Produkt	Hibernate
Hersteller	JBoss
Homepage	`http://www.hibernate.org/`
Aktuelle Version	3.3.2 GA
Implementierungssprache	Java
Unterstützte Plattformen	durch Java alle
Unterstützte Datenspeicher	PostgreSQL, MySQL, Firebird, Oracle, DB2, Sybase, MS SQL Server, HypersonicSQL, Mckoi SQL, SAP DB, Interbase, Pointbase, Progress, FrontBase, Ingres, Informix
Abfrage über	HQL, SQL, Criteria Query, JPQL
Lizenz	LGPL
Status	aktiv

Charakterisierung. *Hibernate* ist das bekannteste *ORM-Framework für Java*. Es ermöglicht die Persistierung von und den Zugriff auf Attributbelegungen und Identitäten gewöhnlicher Ob-

jekte[2], die in relationalen oder objektrelationalen Datenbanken abgebildet sind. Beziehungen zwischen Objekten werden ebenfalls auf entsprechende Relationen abgebildet.
Bei der Arbeit mit Hibernate erscheint es dem Benutzer, als würden ganze Objekte aus der Datenbank geladen und wieder zurück gespeichert werden. Die manuelle Entwicklung der Transformation von Ergebnismengen in objektorientierte Strukturen entfällt.
Im Gegenzug muss eine *Beschreibung des Modells der abzubildenden Daten per XML oder Annotationen der Java-Klassen* vorgenommen werden. Dabei kann aus der Beschreibung ein relationales Datenbankschema erzeugt oder das Datenmodell auf ein bereits vorhandenes Datenbankschema abgebildet werden. Für Zugriffe auf die Daten erzeugt Hibernate anhand der Beschreibung und abhängig vom SQL-Dialekt der verwendeten Datenbank die nötigen SQL-Anweisungen. Somit muss für diese Zugriffe kein SQL explizit programmiert werden, und die *Applikation bleibt von der gewählten Datenbank unabhängig*.
Zum Abrufen der Objekte aus der Datenbank bietet Hibernate ursprünglich nur eine eigene Abfragesprache, die *Hibernate Query Language* (kurz HQL). Mittlerweile implementiert Hibernate *JPA 1.0* und bringt somit zusätzlich die *Java Persistence Query Language* mit [Mic10].

Historie. Hibernate wurde ursprünglich im Jahr 2001 von einer Gruppe von Entwicklern um Gavin King begonnen. Die Firma *JBoss* stellte später die Hauptentwickler des Projekts ein, um die Integration von Hibernate in eigene Produkte voranzutreiben und sorgt seitdem für die Weiterentwicklung von Hibernate.

Zukunftssicherheit. Die Zukunft von Hibernate kann als *sicher* angesehen werden, da Hibernate aktiv weiterentwickelt und in vielen freien und kommerziellen Lösungen eingesetzt wird.

Dokumentationsgrad. Der Dokumentationsgrad von Hibernate kann als *sehr gut* bewertet werden. Neben einem *Referenzhandbuch* stehen *FAQs*, *API-Beschreibungen* und *Tutorials* auf der Homepage zur Verfügung. Weitere Fragen können in der aktiven Community in *Foren* und *Mailinglisten* gestellt werden. Zudem sind im Buchhandel eine Reihe von Büchern zu Hibernate erhältlich. Keines davon ist online verfügbar.

Lizenz. Hibernate steht unter der *Lesser General Public License* zur Verfügung. Eine unter LGPL lizenzierte Software darf, wie bei der GPL, nur zusammen mit ihrem Quelltext vertrieben werden. Im Gegensatz zur GPL dürfen alle Programme, welche die LGPL-lizenzierte Software nur extern benutzen, unter einer eigenen Lizenz stehen. Somit eignet sich Hibernate auch für kommerzielle Produkte.

[2]Häufig *POJOs (Plain Old Java Objects)* genannt.

Clientseitiges Caching. Hibernate bringt ein *zweistufiges Cache-System* mit, welches konfiguriert werden kann. Es kann aus verschieden Cache-Implementierungen (*EH Cache, Swarm Cache, OS Cache, JBossTreeCache*) gewählt oder eine eigene eingesetzt werden. Die zu cachenden Objekte können festgelegt und auf bereits zwischengespeicherte Objekte kann während der Laufzeit Einfluss genommen werden.
Zusätzlich kommt ein *Query-Cache* zum Einsatz. Dieser speichert neben den bereits erstellten SQL-Anfragen auch die Ergebnismengen.

A.2.3. DataNucleus Access Plattform

Produkt	DataNucleus Access Plattform
Hersteller	DataNucleus
Homepage	`http://www.datanucleus.org/`
Aktuelle Version	2.0
Implementierungssprache	Java
Unterstützte Plattformen	durch Java alle
Unterstützte Datenspeicher	MySQL, PostgreSQL, db4o, Firebird, Apache Derby, MaxDB, DB2, Informix, HSQL, H2, McKoi, PostGIS, Pointbase, MS SQL Server, Oracle, Sybase, NeoDatis, LDAP, Excel, ODF, XML, JSON, Amazon S3, Google BigTable, Hadoop HBase
Abfrage über	JPQL, JDQL, SQL
Lizenz	Apache License 2.0
Status	aktiv

Charakterisierung. *DataNucleus Access Plattform* ist ein *Persistenz-Framework* und implementiert die Persistenzmodelle *JPA 2* und *JDO 2.3* vollständig. Es beschränkt sich somit nicht nur auf das Persistieren von Objekten in relationalen und objektrelationalen Datenbanken, sondern unterstützt eine Vielzahl weiterer Datenspeicher. So werden zusätzlich objektorientierte und postrelationale Datenbanken sowie einige Dateiformate unterstützt.
Die Persistenzdefinition kann, wie mit *Hibernate*, wahlweise über eine XML-Beschreibung oder Annotationen vorgenommen werden.
Im Gegensatz zu *Hibernate*[3] werden die Funktionalitäten von *DataNucleus* über eine Vielzahl von *Plugins* zur Verfügung gestellt. Die eingesetzte Konfiguration der Lösung kann somit kompakt gehalten werden.

[3] welches einen Großteil seiner Funktionalitäten in einem Paket liefert

Historie. DataNucleus hat seinen Ursprung im Open-Source-Projekt *TriActive JDO*[4], welches ebenfalls ein Persistenz-Framework ist. Es implementiert bis heute lediglich das Persistenzmodell *JDO 1.0*. Im Jahr 2003 wurde das Projekt *Java Persistent Objects* (kurz *JPOX*), von einer Gruppe von Entwicklern begonnen, die den Funktionsumfang von *TriActive JDO* erweitern wollten. *JPOX* wurde schließlich von *Sun Microsystems* als Referenzimplementierung für *JDO 2* ausgewählt. Darüber hinaus implementiert es *JPA 1*. Im Jahr 2008 wurde die Weiterentwicklung von *JPOX* eingestellt und schließlich ging das Projekt *DataNucleus* daraus hervor, welches abermals eine Erweiterung der Funktionen des ursprünglichen Frameworks zum Ziel hat.

Zukunftssicherheit. Die Zukunft von DataNucleus kann als *relativ sicher* angesehen werden. Es befindet sich aktiv in der Weiterentwicklung und wird in einigen freien und kommerziellen Lösungen genutzt.

Dokumentationsgrad. Der Dokumentationsgrad von DataNucleus kann als *zufriedenstellend* bewertet werden. Auf der Homepage des Frameworks wird eine umfangreiche Dokumentation in Form eines *Wikis* angeboten. Aufgeführt werden *Tutorials*, *FAQs* und mehrere *API-Beschreibungen*. Zusätzlich gibt es ein *Benutzerforum*.

Lizenz. DataNucleus Access Plattform steht unter der *Apache License 2.0*. Software, die unter dieser Lizenz steht, darf in jedem Umfeld frei verwendet, modifiziert und verteilt werden. Programme, die unter Apache-Lizenz stehende Quelltexte verwenden, brauchen selbst nicht unter dieser Lizenz zu stehen.

Clientseitiges Caching. Auch DataNucleus bringt ein *zweistufiges Cache-System* mit, welches konfiguriert werden kann. Es kann ebenfalls aus verschiedenen Cache-Implementierungen gewählt oder eine eigene eingesetzt werden. Auf bereits zwischengespeicherte Objekte kann während der Laufzeit Einfluss genommen werden. Wie bei Hibernate gibt es ebenfalls einen *Query Cache*.

A.2.4. iBATIS

Produkt	iBATIS
Hersteller	Apache Software Foundation
Homepage	`http://ibatis.apache.org/`
Aktuelle Version	2.3.4
Implementierungssprache	Java, .NET
Unterstützte Plattformen	durch Java alle, mit .NET nur Windows

[4] http://tjdo.sourceforge.net/, abgerufen am 15. September 2010

Unterstützte Datenspeicher	MySQL 4, PostgreSQL 7/8, Firebird, HSQLDB, Oracle, DB2, MS SQL Server, MS Access 97/2000
Abfrage über	SQL
Lizenz	Apache License 2.0
Status	aktiv

Charakterisierung. *iBATIS* ist für die Trennung von Datenbankzugriffscode und Applikationscode zuständig. Es ist *kein ORM-Framework*, denn die Abbildung von objektorientierten Klassen auf Relationen muss vom Entwickler selbst vorgenommen werden. Dazu werden der Applikation *Data-Access-Objects* (kurz *DAOs*) zur Verfügung gestellt und die SQL-Anweisungen in XML-Dateien, sogenannte SQL-Maps, ausgelagert. Damit wird die Zuordnung von Tabellen zu Klassen von der restlichen Programmlogik entkoppelt.
Im Gegensatz zu *Hibernate* oder *DataNucleus Access Plattform* ist die automatische Erstellung eines Datenbankschemas aus dem Datenmodell nicht möglich.

Historie. Das Projekt wurde 2001 von Clinton Begin begonnen. Ursprünglich ging es dabei um die Entwicklung von Kryptografie-Software. Anfang 2002 behauptete Microsoft in einer Veröffentlichung, dass *.NET* erheblich schneller und produktiver als *J2EE* sei. Um dies zu widerlegen, entwickelte das iBATIS-Projekt, anhand der Anforderungen aus der Microsoft-Veröffentlichung, das System *JPetStore*, welches am 1. Juli 2002 veröffentlicht wurde. Das heute unter dem Name iBATIS bekannte Persistenz-Framework stammt ursprünglich aus JPetStore.

Zukunftssicherheit. Auch die Zukunft von iBATIS kann als *relativ sicher* angesehen werden. Es befindet sich aktiv in der Weiterentwicklung und wird in einigen freien und kommerziellen Lösungen genutzt.

Dokumentationsgrad. Der Dokumentationsgrad zu iBATIS kann als *zufriedenstellend* bewertet werden. Über die Homepage werden ein kurzes *mehrsprachiges Referenzhandbuch*, *Tutorial*, *Wiki*, *FAQ* und mehrere *API-Beschreibungen* angeboten. Die Community kann über eine *Mailingliste* zu weiteren Aspekten befragt werden.

Lizenz. iBATIS steht unter der *Apache License 2.0* zur Verfügung. Software, die unter dieser Lizenz steht, darf in jedem Umfeld frei verwendet, modifiziert und verteilt werden. Programme, die unter Apache-Lizenz stehende Quelltexte verwenden, brauchen selbst nicht unter dieser Lizenz zu stehen.

Clientseitiges Caching. iBATIS bringt einen *einstufigen Cache* mit.

A.2.5. Weitere Persistenzframeworks

Mittlerweile existiert eine Vielzahl an Persistenz-Frameworks für den freien und kommerziellen Einsatz. Eine Liste von Lösungen für Java kann bei [JS10] eingesehen werden.

B. Abbildung der Reihenfolge von Graphelementen

Eine der zentralen Anforderungen an die Persistenzlösung lautet, dass alle Eigenschaften des TGraphs in der Datenbank erhalten bleiben müssen. Darunter fällt unter anderem die Anordnung der Knoten und Kanten zueinander, die durch die Reihenfolge der Inzidenzen in den Inzidenzlisten der Knoten realisiert wird. Zudem muss die Reihenfolge der Knoten und Kanten in den globalen Knoten- und Kantensequenzen des Graphen erhalten bleiben.
Zur Abbildung der Reihenfolge der Elemente in der Knotensequenz, Kantensequenz und den Inzidenzlisten werden zwei Lösungsansätze unterschieden. Der erste Ansatz setzt auf eine *Verkettung* und der zweite auf eine *Nummerierung* der Elemente.

B.1. Verkettung

Die Verkettung der Elemente orientiert sich an der klassischen Implementation von *JGraLab*. Sie realisiert Knoten- und Kantensequenzen sowie Inzidenzlisten mit *doppelt verketteten Listen*. Jedes Element einer doppelt verketteten Liste besitzt eine Referenz auf seinen Vorgänger und eine auf seinen Nachfolger. Dadurch kann die Liste in beide Richtungen durchlaufen werden, vorausgesetzt, man hat einen Einstiegspunkt in die Liste.
Ein Graph besitzt jeweils Referenzen auf das erste und das letzte Element der Knoten- und Kantensequenzen. Analog dazu besitzt ein Knoten jeweils eine Referenz auf die erste und letzte Inzidenz. Besteht eine doppelt verkettete Liste aus nur einem Element, so ist es gleichzeitig das erste und letzte Element dieser Liste.
Die Verkettung der Elemente kann auch in einer Datenbank erhalten bleiben. Dazu müssen entsprechende Anpassungen am generische relationalen Schema vorgenommen werden.

B.1.1. Angepasstes generisches relationales Schema

Um die Verkettung abzubilden, müssen die Referenzen auf die Einstiegselemente und für jedes Element die Referenzen auf Vorgänger und Nachfolger in der Datenbank persistiert werden.
Um die Einstiegspunkte für Kanten- und Knotensequenzen zu persistieren, wird die Relation `Graph` aus dem generischen relationalen Schema um die Attribute `firstVertex` und `firstEdge` erweitert. Sie enthalten den Fremdschlüssel eines Elements aus den Relationen

`Vertex` und `Edge`, welches jeweils das erste Element der Knoten- und Kantensequenz darstellt.
Die Referenzen auf die letzten Elemente der Sequenzen werden nicht in eigenen Attributen abgelegt. Sie *können durch SQL-Abfragen ermittelt werden*.

Die Relation `Graph` sieht dann wie folgt aus:

```
Graph(
  gId:Integer,
  id:String,
  version:Long,
  firstVertex:Integer,
  vSeqVersion:Long,
  firstEdge:Integer,
  eSeqVersion:Long,
  type:Integer
)
```

Für die Kantensequenz wird die Referenz auf den Nachfolger in der Relation `Edge` durch das Attribut `nextEdge`, welches den Fremdschlüssel zu einer weiteren Kante aus `Edge` enthält, aufgenommen.
Auf das Ablegen der Referenz auf das Vorgängerelement kann zur Redundanzvermeidung auch hier verzichtet werden, *da es durch eine Abfrage ermittelt werden kann*. Zudem werden die Listen in der Regel vorwärts durchlaufen.

Die Relation sieht dann wie folgt aus:

```
Edge(
  eId:Integer, gId:Integer,
  type:Integer,
  nextEdge:Integer
)
```

Analog zu `Edge` wird die Relation `Vertex` mit dem Attribut `nextVertex` erweitert. Für die Referenz auf das erste Element der Inzidenzliste wird die Relation analog zu `Graph` mit dem Attribut `firstIncidence` erweitert.

Die Relation sieht dann wie folgt aus:

```
Vertex(
  vId:Integer, gId:Integer,
  type:Integer,
  lambdaSeqVersion:Long,
  nextVertex:Integer,
  firstIncidence:Integer
)
```

Auf die gleiche Weise wird mit der Inzidenzliste verfahren. Die Relation `Incidence` wird dazu um das Attribut `nextIncidence` ergänzt.

Die Relation sieht dann wie folgt aus:

```
Incidence(
  eId:Integer, gId:Integer, direction:Direction,
  vId:Integer,
  nextIncidence:Integer
)
```

B.1.2. Resultierende Abfragen

Aus dem relationalen Schema lässt sich nun ableiten, wie eine SQL-Abfrage zum Ermitteln und Abrufen des ersten Knotens in *Vseq* des Graphen `foo` formuliert werden muss:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE id = 'foo' AND vId = firstVertex
```

Die SQL-Abfrage zum Ermitteln und Abrufen des letzten Knotens in *Vseq* des gleichen Graphen kann folgendermaßen formuliert werden:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE id = 'foo' AND nextVertex = NULL
```

Zum Ermitteln und Abrufen des Vorgängers in der Knotensequenz des Knotens 34 aus dem gleichen Graph kann folgende SQL-Abfrage formuliert werden:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE nextVertex = 34 AND id = 'foo'
```

Der Nachfolger in der Knotensequenz des Knotens 34 ist im Attribut `nextVertex` hinterlegt und soll hier der Knoten 87 sein. Er muss nicht ermittelt werden und kann direkt abgerufen werden. Die SQL-Abfrage zum Abrufen des Nachfolgeknotens lautet dann:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE vId = 87 AND id = 'foo'
```

Analog dazu werden die SQL-Abfragen für Elemente aus der Kantenliste und den Inzidenzlisten formuliert.

B.1.3. Bewertung

Die grundsätzliche Stärke der Verkettung liegt darin, dass Änderungen an den Sequenzen (durch Einfügen, Löschen und Verschieben) einen konstanten Aufwand bedeuten. Zudem ist der Zugriff auf die Elemente beim Vorwärtsdurchlaufen einer Sequenz über die Referenzen auf das erste Element und die Nachfolger sehr einfach, denn das nächste Element muss nicht ermittelt werden, da es bereits bekannt ist.
Demgegenüber steht ein im Vergleich dazu uneffizientes Rückwärtsdurchlaufen der Sequenzen, da der Vorgänger immer ermittelt werden muss. Dieser Nachteil kommt besonders dann zum Tragen, wenn festgestellt werden soll, ob sich ein Element vor einem anderen gegebenen Element in der gleichen Sequenz befindet.

B.2. Nummerierung

Im zweiten Ansatz wird die Reihenfolge der Elemente in den Knoten- und Kantensequenzen sowie in den Inzidenzlisten durch eine Nummerierung der Elemente ausgedrückt. Dabei wird das erste Element einer Liste mit der kleinsten und das letzte Element mit der größten Nummer versehen.

B.2.1. Generisches relationales Schema

Im Gegensatz zur Verkettung kann auf ein Attribut zum Referenzieren des ersten Elements einer Liste verzichtet werden. Die Relationen `Edge`, `Vertex` und `Edge` kommen dann jeweils

mit nur einem zusätzlichen Attribut `sequenceNumber` aus. Die Relation `Graph` benötigt kein zusätzliches Attribut.

Die relevanten Relationen des generische relationalen Schemas lauten:

```
Vertex(
  vId:Integer, gId:Integer,
  type:Integer,
  sequenceNumber:Long,
  lambdaSeqVersion:Long
)

Edge(
  eId:Integer, gId:Integer,
  type:Integer,
  sequenceNumber:Long
)

Incidence(
  eId:Integer, gId:Integer, direction:Direction,
  vId:Integer,
  sequenceNumber:Long
)
```

B.2.2. Resultierende Abfragen

Die SQL-Abfragen zur Ermittlung des ersten und letzten Elements sowie des Vorgängers und Nachfolgers werden auch hier für die Relation `Vertex` exemplarisch aufgeführt, da sie für Kanten und Inzidenzen analog formuliert werden können.

Das erste Element der Knotensequenz des Graphen `foo` kann über folgende SQL-Abfrage ermittelt werden:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE id = 'foo'
ORDER BY sequenceNumber ASC
LIMIT 1
```

Das letzte Element der Knotensequenz des gleichen Graphen kann über die folgende SQL-Abfrage ermittelt werden:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE id = 'foo'
ORDER BY sequenceNumber DESC
LIMIT 1
```

Der Vorgänger in der Knotensequenz des Knotens 32 mit Sequenznummer 642 kann über folgende SQL-Abfrage ermittelt werden:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex
WHERE id = 'foo' AND sequenceNumber < 642
ORDER BY sequenceNumber DESC
LIMIT 1
```

Der Nachfolger des gleichen Knotens kann über folgende SQL-Abfrage ermittelt werden:

```
SELECT vId, typeId, sequenceNumber, lambdaSeqVersion FROM Vertex, Graph
WHERE id = 'foo' AND sequenceNumber > 642
ORDER BY sequenceNumber ASC
LIMIT 1
```

B.2.3. Bewertung

Grundsätzlich hat dieser Lösungsansatz den Nachteil, dass ein hoher Aufwand beim Einfügen und Verschieben von Elementen in einer Sequenz entsteht, da die Nummern aller nachfolgenden Elemente entsprechend angepasst werden müssen, damit die Reihenfolge erhalten bleibt. Nachteilig wirkt sich auch aus, dass beim Vorwärts- und Rückwärtsdurchlaufen einer Sequenz jedes Element ermittelt werden muss, da keines direkt referenziert wird. Beim Vorwärtsdurchlauf einer Sequenz ist die Verkettung somit im Vorteil, da dort jeder Nachfolger über seinen Primärschlüssel bekannt ist.

B.2.4. Optimierungen

Durch Lücken in Nummerierung mit der Größe 2^k kann verhindert werden, dass dies bei jedem Einfügen und Verschieben geschehen muss. Dazu bekommt bereits beim Aufbau des Graphen jedes Element, welches einer Sequenz angefügt wird, eine Nummer, die um den Wert 2^k größer ist, als die Nummer des unmittelbaren Vorgängers. Zwar können somit zwischen zwei

Elementen 2^k - 1 Elemente Platz finden, jedoch teilt jedes eingefügte Element die Lücke in zwei gleich große Teile, so dass im Mittel wesentlich weniger Einfügungen vorgenommen werden können, bevor eine neue Lücke geschaffen werden muss. Im ungünstigsten Fall können genau k Einfügungen vorgenommen werden.
Wählt man `BIGINT` (2^{64}) als Datentyp für `seqNumber` und eine Lückengröße von 2^k, so können einer Sequenz maximal 2^{64-k} Elemente angehören, bevor die Lückenbreite verkleinert werden muss.

Die Zugriffsgeschwindigkeit des Lösungsansatzes der Nummerierung der Elemente kann auf das Niveau der Zugriffsgeschwindigkeit der Verkettung und teilweise sogar darüber hinaus angehoben werden. Dazu muss die Reihenfolge der Elemente in den Sequenzen bereits beim Laden des Graphen aus der Datenbank ermittelt werden. Es kann eine SQL-Anfrage formuliert werden, die alle Primärschlüssel der Elemente einer Sequenz nach ihrer Nummerierung sortiert zum Ergebnis hat. Exemplarisch sieht diese Anfrage für die Knotensequenz *Vseq* wie folgt aus:

```
SELECT vId, sequenceNumber FROM Vertex, Graph
WHERE id = 'foo'
ORDER BY sequenceNumber ASC
```

Das Ergebnis kann in eine `Collection` aufgenommen werden und *steht dann clientseitig zur Verfügung*. Auf diese Weise können alle Elemente über ihren Primärschlüssel angesprochen werden und die umständlichen SQL-Abfragen aus Abschnitt B.2.2 können vermieden werden. Bereits ohne Optimierung hat dieser Lösungsansatz gegenüber der Verkettung den Vorteil, dass mit weit weniger Aufwand ermittelt werden kann, ob sich ein Element in einer Sequenz vor oder nach einem gegebenen Element befindet.

B.3. Vergleich

Beim Zugriff ist der optimierte Ansatz der Nummerierung dem der Verkettung ebenbürtig bis teilweise überlegen.
Nur beim Verschieben und Einfügen von Elementen in die Sequenzen behält der Ansatz der Verkettung mit seinem konstanten Aufwand langfristig die Nase vorn. Über die Optimierung durch Lücken ist der Ansatz der Nummerierung dem Ansatz der Verkettung nur so lange überlegen, bis eine Lücke geschaffen werden muss; im ungünstigsten Fall also nach k Einfügungen oder Verschiebungen. Änderungsoperationen am Graphen kommen in der Regel aber nicht sehr häufig vor.

C. Traversierungsalgorithmen

In diesem Kapitel werden Java-Implementationen der Graph-Traversierungs-Algorithmen aufgeführt, die zur Leistungsbewertung herangezogen wurde. Sie ermöglichen es in einem *Java5-Graphen* ausgehend von einem beliebigen Knoten nach einem Knoten `Identifier` mit gegebenem *Namen* zu suchen.

C.1. Depth First Search

```
public Vertex dfs(Vertex start, String targetName) throws
    NoSuchFieldException{
  BooleanGraphMarker marker = new BooleanGraphMarker(start.getGraph());
  Stack<Vertex> stack = new Stack<Vertex>();
  stack.push(start);
  while(!stack.isEmpty()){
    Vertex currentVertex = stack.pop();
    marker.mark(currentVertex);
    if(isTarget(currentVertex, targetName))
      return currentVertex;
    Edge edge = currentVertex.getFirstEdge(EdgeDirection.IN);
    while (edge != null){
      Vertex otherEnd = edge.getThat();
      if (!marker.isMarked(otherEnd))
        stack.push(otherEnd);
      edge = edge.getNextEdge(EdgeDirection.IN);
    }
  }
  return null;
}

private boolean isTarget(Vertex vertex, String targetName) throws
    NoSuchFieldException{
  return vertex instanceof Identifier &&
      ((Identifier)vertex).get_name().equals(targetName);
}
```

C.2. Breadth First Search

```
public Vertex bfs(Vertex start, String targetName) throws
    NoSuchFieldException{
  BooleanGraphMarker marker = new BooleanGraphMarker(start.getGraph());
  LinkedList<Vertex> queue = new LinkedList<Vertex>();
  queue.add(start);
  while(!queue.isEmpty()){
    Vertex currentVertex = queue.pop();
    marker.mark(currentVertex);
    if(this.isTarget(currentVertex, targetName))
      return currentVertex;
    Edge edge = currentVertex.getFirstEdge(EdgeDirection.IN);
    while(edge != null){
      Vertex otherEnd = edge.getThat();
      if (!marker.isMarked(otherEnd))
        queue.add(otherEnd);
      edge = edge.getNextEdge(EdgeDirection.IN);
    }
  }
  return null;
}

private boolean isTarget(Vertex vertex, String targetName) throws
    NoSuchFieldException{
  return vertex instanceof Identifier &&
     ((Identifier)vertex).get_name().equals(targetName);
}
```

D. CD

Auf der beigefügten CD befinden sich alle vom Autor im Rahmen dieser Arbeit erstellten elektronischen Artefakte.

Enthaltene Quelltexte entsprechen jenen im Repository des Instituts für Software-Technik der Universität Koblenz-Landau:

- Subversion-Repository:
  `https://svn.uni-koblenz.de/ist`
- Unterverzeichnis:
  `/projects/jgralab/branches/db-persistence/`