UNIVERSITÄT
LEIPZIG

Institut für Informatik
Fakultät für Mathematik und Informatik
Abteilung Datenbanken

Relationale Speicherung und Verarbeitung für temporale Graphdaten

Masterarbeit

vorgelegt von:
Philip Fritzsche

Matrikelnummer:
3755781

Betreuer:
Prof. Dr. Erhard Rahm

Kurzzusammenfassung

In dieser Arbeit wird eine flexible Möglichkeit zur Abbildung von temporalen Graphen auf relationale Datenbanken gegeben. Dazu ist ein Graph-Schema definiert, mit dem ein Graph in logische Typen aufgeteilt wird. Jeder Typ als Tabelle in einer relationalen Datenbank abgebildet. Dabei bildet ein solcher Typ entweder nur Elementes einer bestimmten Art, also eines festen Labels, oder alle Knoten/Kanten ab. Properties sind wahlweise als Spalten oder als Einträge einer separaten Property-Tabelle für jeden Typ kodiert.

Die Konsistenz des Graphen in der Datenbank wird über Integritätsbedingungen garantiert. Im Gegensatz zu anderen Implementieren entfallen dadurch aufwendige Schritte zur Verifizierung der Ergebnisse.

Neben Property-Graphen wird die Möglichkeit der Abbildung von temporalen Property Graphen gegeben. Diese werden entsprechend auch auf eine temporale relationale Datenbank abgebildet. Durch Integritätsbedingungen wird wieder garantiert, dass auch dieses Modell eingehalten wird. Außer kann damit die aufwendige Verwaltung temporaler Daten, insbesondere die Speicherung und Anfrage historischer Graph-Zustände leicht umgesetzt werden. Allgemein werden Operatoren zur Verarbeitung von Graphen auf Operationen im relationalen Modell übersetzt. Dies gilt auch für Operatoren, welche speziell für temporale Graphen definiert sind.

In der Implementierung ist ein System gegeben, welches zur Definition und Anwendung der Abbildungen genutzt werden kann. Es werden APIs Implementiert, mit denen Anfragen auf dem Graph-Modell leicht anhand der Abbildungen in welche für das relationale Modell übersetzt werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

Die Arbeit stellt eine flexible Abbildung von *Property Graphen* auf das relationale Modell vor. Dabei werden insbesondere temporale Graphen auf temporale Datenbanken abgebildet. Eine Implementierung der Abbildung auf Basis von MariaDB wird vorgestellt.

1.1. Motivation

Das Graph-Modell bietet eine intuitive Möglichkeit der Darstellung realer Gegebenheiten. In der Welt kann quasi jede Situation als Menge von Beziehungen oder Interaktionen zwischen Objekten angesehen werden. Personen in einem sozialen Netzwerk interagieren miteinander, Banken wickeln finanzielle Transaktionen im Auftrag von Kund_innen ab, Firmen schließen neue Verträge ab und kündigen alte. Für eine Vielzahl von Anwendungsfällen biete sich die Verarbeitung von Daten in Form von Graphen an.

Dabei sind Objekten und Beziehungen stets bestimmte Attribute zuzuschreiben, die es abzubilden gilt. Ein entsprechendes Datenmodell bietet sich daher an. Hier wird ein Property Graph Modell [3] verwendet.

Da sich reale Gegebenheiten ständig ändern, werden abgebildete Daten auch permanent aktualisiert. Es ist allerdings nicht immer sinnvoll nur die aktuelle Sicht der Daten zu betrachten. Es kann auch sinnvoll sein, eine Möglichkeit zu geben, mit der historische Daten betrachtet und verglichen werden können. Für eine Bank, welche Transaktionen zwischen Kund_innen als Graph modelliert, kann es beispielsweise von Bedeutung sein, alle Transaktionen eines bestimmten Zeitraumes zu betrachten oder geschäftliche Beziehungen in der Vergangenheit zu analysieren.

Dafür ist ein temporales Modell für Graphen definiert, welches jedem Element Zeiten zuweist, in denen dieses gültig ist [20]. Ein Graph repräsentiert entsprechen nicht nur den aktuellen Zustand, sondern enthält auch alle Änderungen, wodurch alte Daten verarbeitet werden können.

Modelle für temporale Daten sind dabei keine Neuheit. Im Bereich relationaler Datenbanken finden sich dazu Konzepte die bereits Jahrzehnte alt sind, beispielsweise von Snodgrass aus dem Jahr 1999 [22]. In näherer Vergangenheit wurde der etablierte SQL Standard zu Datenbanken um Möglichkeiten zur Verarbeitung von temporalen Daten erweitert. [13]

Es finden sich Arbeiten, die zeigen, wie Graphen auf relationale Modelle abgebildet werden können. Zur Verarbeitung von temporalen Graphen liegt also die Idee nahe, dass diese auf temporale relationale Modelle übertragen werden können. Dabei lassen sich positive Eigenschaften des relationalen Modells ausnutzen, um beispielsweise ganz automatisch zu garantieren, dass abgebildete Graphen konsistent sind.

1.2. Ziel der Arbeit

Mit dieser Arbeit wird demonstriert, wie temporale (als auch nicht-temporale) Property Graphen auf ein relationales Modell abgebildet werden können. Es wird ein flexibles Schema definiert, anhand dessen die Abbildung gezeigt wird. Dieses kann je nach Anwendungsfall angepasst werden und verschmelzt dabei Konzepte aus anderen Arbeiten.

Durch die Nutzung einer temporalen, relationalen Datenbank ergeben sich manche Operationen für temporale Graphen, wie beispielsweise die Extraktion einen vergangenen Graph-Zustandes ganz automatisch. In Gradoop, als Referenz-Implementierung des temporalen Graph Modells TPGM [20] muss durch aufwendige Zusatzschritte garantiert werden, dass der resultierende Graph konsistent ist, also dem Modell entspricht. Datenbanken bieten allerdings die Möglichkeit so genannte Integritätsbedingungen zu definieren, welche Einschränken bzw. Eigenschaften der Daten beschreiben. Die Datenbank sorgt dafür, dass diese stets eingehalten werden [8]. Hier sind diese so zu definieren, dass Zusatzschritte zur Validierung von Graphen größtenteils eliminiert werden können.

Es sind eine Auswahl von Operatoren auf Graphen implementiert, werden in Operationen auf der Datenbank übersetzt und als diese ausgeführt werden.

1.3. Aufbau der Arbeit

Zu Beginn werden einige notwendige Grundlagen zu Graphen und relationalen Datenbanken gegeben. Das hier verwendete Graph-Modell wird definiert. Danach werden verwandte Arbeiten je kurz genannt. Im Anschluss, in Kapitel 4, wird das Konzept dieser Arbeit dargelegt. Es wird gezeigt, wie Graphen auf Relationen abgebildet werden können. Eine formale Definition wird hierfür gegeben. Für ein paar Operatoren wird demonstriert, wie diese auf Relationen ausgeführt werden können.

Kapitel 5 gibt anschließend einen groben Überblick über die Implementierung.

2. Grundlagen

2.1. Graphen

Formal handelt es sich bei Graphen um Mengen von Knoten und Kanten. Wobei jede Kante je 2 Knoten verbinden. Ein Graph kann damit als Paar $G = \langle V, E \rangle$ dargestellt werden. V ist eine Menge von Knoten und $E \subseteq [V]^2$ eine Menge von Kanten, wobei diese als Menge von je 2 Knoten dargestellt sind. [9] In Bildform werden für Graphen üblicherweise Knoten als Kreise oder Punkte, Kanten als Linien zwischen diesen dargestellt, beispielsweise wird in Abbildung 2.1 der Graph $G = \langle V, E \rangle$ mit $V = \{1, 2, 3, 4\}$ und $E = \{\{1, 2\}, \{2, 3\}\}$.

Der zuvor gezeigte Graph ist ein Beispiel eines *ungerichteten* Graphen, es werden mit jeder Kante je 2 Knoten verbunden, aber die Richtung der Kante ist nicht relevant. Die Kanten $\{1, 2\}$ und $\{2, 1\}$ stellen das gleiche Objekt dar. Alternativ dazu werden *gerichtete* Graphen definiert: $G = \langle V, E \rangle$, wobei V wieder eine Menge von Knoten und E eine Menge von Kanten ist. Die Mengen sind disjunkt. Es sind zwei Abbildungen $source : E \to V$ und $target : E \to V$ gegeben, die zu einer Kante je den *Quell-* (*source*) und Ziel-Knoten (*target*) zuordnen. In Diestel werden Knoten auch Ecken genannt, statt Quell-Knoten wird da die Bezeichnung Anfangsecke und statt Ziel-Knoten Endecke verwendet [9]. Die hier verwendeten Bezeichnungen sind aus den im Englischen üblichen *source* und *target* für Anfangs-/Quell- und End-/Ziel-Knoten abgeleitet. Es ist mit dieser Definition auch möglich, dass mehrere Kanten (in gleiche oder verschiedene Richtung) zwischen zwei Knoten existieren. Eine Beispiel-Darstellung eines gerichteten Graphen ist in Abbildung 2.2 gegeben.

Zu einem Graph kann ein Teilgraph bestimmt werden. Ein Teilgraph $G' = \langle V', E' \rangle$ ist ein Graph, dessen Knoten- und Kanten-Mengen je Teilmengen des ursprünglichen Graphen sind. Es gilt also $V' \subseteq V$ und $E' \subseteq E$. Weiter muss zu jeder Kante gerantiert sein, dass je Quell- und Zielknoten im Teilgraph enthalten sind:

$$e \in E' \implies source(e') \in V' \wedge target(e') \in V' \tag{2.1}$$

Als Beispiel wird in Abbildung 2.3 ein Teilgraph des vorherigen Beispiel-Graphen gezeigt.

2.1.1. Property Graphen

Das Graph-Modell soll genutzt werden, um Daten abzubilden, in denen Beziehungen zwischen Entitäten dargestellt sind. Ein reiner Graph ist dafür nicht aussagekräftig genug. Deshalb wird ein

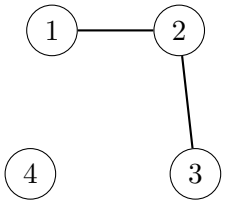

Abbildung 2.1.: Ein (ungerichteter) Graph

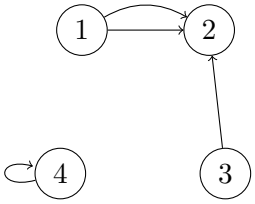

Abbildung 2.2.: Ein gerichteter Graph

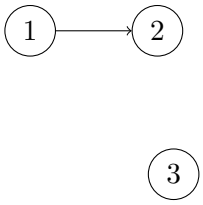

Abbildung 2.3.: Teilgraph des gerichteten Graph

Property Graph Modell verwendet, in dem zu jeden Element, also jedem Knoten und jeder Kanten, noch zusätzlich Attribute assoziiert sind. Nach Angles [3] ist ein *Property Graph* ein gerichteter Graph, bei dem wieder mehrere Kanten zwischen zwei Knoten zulässig sind. Zu jedem Knoten und Kante werden *Properties* zugewiesen, wobei Properties Paare aus Namen und Wert (*Key* und Wert) sind. Für eine Menge von Property-Namen kann jedem Element zu jedem Name ein Wert zugewiesen sein.

Weiter wird die Möglichkeit gegeben, Elemente in Kategorien zu unterteilen. Dazu je sind *Typ-Labels* (auch nur *Label* genannt), welche die Kategorie oder Art der Elemente identifizieren. Beispielsweise können allen Knoten, die für ein soziales Netzwerk eine Person darstellen sollen, mit dem Label *Person* versehen sein. Für Kanten kann das Label genutzt werden, um die Art der Beziehung zwischen zwei Entitäten zu beschreiben. Zwischen zwei Personen kann eine Kante zum Beispiel eine Bekanntschaft darstellen. Solch eine Kante ließe sich mit dem Label *knows* versehen.

Im folgenden wird formal ein Modell für *Property Graphen* definiert, welches in dieser Arbeit verwendet wird. Dieses orientiert sich am EPGM Modell in Gradoop von Junghanns et. al. [12], sowie dessen Erweiterung TPGM von Rost et. al. [18]. Wie in beiden Modellen, wird hier auch jedem Element eine ID zugewiesen, die dazu genutzt werden kann, um Elemente zu identifizieren. Dieser Wert sollte im Allgemeinen eindeutig sein, so kann beispielsweise jede *Person* über eine Sozialversicherungsnummer identifiziert werden. IDs können dabei auch beliebige, generierte Werte sein.

Definition 1 (Property Graph)**.** Ein *Property Graph* $\mathbb{G}$ wird als Tupel definiert durch

$$\mathbb{G} = \langle V, E, \mathcal{I}, \mathcal{L}, P, A, label, id, source, target, \kappa \rangle \tag{2.2}$$

Dabei sind V eine Menge von Knoten, E eine Menge von Kanten, $\mathcal{I}$ eine Menge von IDs, $\mathcal{L}$ eine Menge von Typ-Labels, P eine Menge von Property-Namen (*Property Keys*), A eine Menge von Property-Werten. *label* ist eine Funktion mit $label : V \cup E \to \mathcal{L}$ die jedem Element ein Typ-Label zuordnet. $id : V \cup E \to \mathcal{I}$ ordnet jedem Element eine ID zu. $source : E \to \mathcal{I}$ und $target : E \to \mathcal{I}$ sind Funktionen die jeder Kante einen Knoten zuordnen. Dabei weisen diese je die ID der entsprechenden

Quell- und Zielknoten zu. Der Quell-Knoten v_s einer Kante e ist dann der Knoten, für den $id(v_s) = source(e)$ gilt. Für den Ziel-Knoten v_t gilt entsprechend $id(v_t) = target(e)$. κ ist eine partielle Abbildung die jedem Element eine Menge von Properties zuordnet: $\kappa : (V \cup E) \times P \rightharpoonup A$. So kann für jedes Element zu jedem Property-Namen (Key) aus P ein Wert aus A zugeordnet werden.

Für Property Graphen sind einige Operatoren definiert, welche im weiteren in Kapitel 4 vorgestellt werden.

2.2. Relationale Datenbanken

Relationale Datenbanken bieten ein System zur Verwaltung von Daten in Form von Tabellen [8]. Ein Datenbanksystem bietet Methoden, um die verwalteten Daten mit neuen zu ergänzen (`INSERT`), zu aktualisieren (`UPDATE`), zu löschen (`DELETE`) und anzufragen (`SELECT`). Dafür wird die Anfragesprache SQL formalisiert, mit der solche Methoden ausgeführt werden können. Weiter wird SQL dazu genutzt um die Struktur der Daten, also die vorhandenen Tabellen und deren Aufbau, zu definieren. Die Grundlage hierfür bietet das von Codd vorgestellte [4] und später erweiterte [5] relationale Modell.

Zuerst wird hier ein Überblick über das relationale Datenbankmodell (auch nur Modell) gegeben, dass im weiteren Verlauf der Arbeit häufig referenziert wird.

2.2.1. Das relationale Modell

Im relationalen Modell (nach Codd [5]) werden Daten als Relationen dargestellt. Eine Relation setzt sich aus Komponenten zusammen. Die Elemente einer Relation entsprechend Tupeln.

Man nehme hierzu beliebige Mengen A, B, ... Dabei wird davon ausgegangen, dass jede dieser Mengen Elemente eines bestimmten Datentyps enthält, beispielsweise Zahlen, Zeichenketten oder Zeitpunkte. Eine Relation kann aus solchen Mengen über ein Kreuzprodukt $\times$ zusammengesetzt werden, wobei diese eine Teilmenge des Produktes enthält. So ist beispielsweise eine Relation

$$R \subseteq A \times B \times C \tag{2.3}$$

aus den Mengen A, B und C konstruiert. Diese sind die *Komponenten* der Relation. Elemente entsprechend Tupel, deren Einträge Elemente der Komponenten sind. Sind für A, B und C beispielsweise Zahlenmengen gewählt, dann könnte R ein Tupel

$$\langle 123, 0, 999 \rangle \tag{2.4}$$

enthalten. Die Komponenten sind $123 \in A$, $0 \in B$ und $999 \in C$. Die Reihenfolge ist dabei hier wichtig, damit klar ist, welcher Eintrag welcher Komponente entspricht.

Da die Zuordnung rein anhand von Reihenfolge recht unübersichtlich ist, wird jeder Komponente noch ein Name zugewiesen. Dann können diese über den Name referenziert werden. Codd nutzt dabei die Notation:

$$R(colA : A, b : B, columnC : C) \tag{2.5}$$

Der ersten Komponente wird der Name $colA$ zugewiesen. Diese enthält Elemente aus der Menge bzw. dem Datenbereich A (in Codd auch *Domain* genannt). Den weiteren Komponenten sind hier die Namen b und $columnC$ zugewiesen. Die Namen sind frei wählbar, müssen aber eindeutig sein. Mit diesen können nun Operationen auf Relationen gezeigt werden.

2.2.1.1. Zugriff auf Komponenten

Der Zugriff auf bestimmte Werte in einem Tupel wird geschrieben als:

$$R \ni \langle 1, 2, 3\rangle .colA = 1 \tag{2.6}$$

Komponenten werden also über ihren Namen ausgewählt.

2.2.1.2. Restriktion

Aus einer Relation kann eine Teilmenge bestimmt werden, für die eine festgelegte Bedingung gilt. Zum Beispiel können aus R alle Tupel gewählt werden, deren Komponente b größer als 3 ist:

$$R[b > 3] \tag{2.7}$$

Im allgemeinen werden in dieser Notation vergleiche einer Spalte mit einem Wert oder einer anderen Spalte ermöglicht.

$$R[colA > b] \tag{2.8}$$

meint die Teilrelation, in der die Werte der Komponente $colA$ größer als die von b sind.

2.2.1.3. Join

Hiermit können zwei Relationen verbunden werden, wobei Spalten aus den Relationen miteinander verglichen werden. Sind beispielsweise zwei Relationen $R(A : \mathbb{N}, B : \mathbb{N}, C : \mathbb{N})$ und $S(D : \mathbb{N}, E : \mathbb{N})$ gegeben, dann kann ein Join $R(A, B, C)[B > D]S(D, E)$ erfolgen durch:

$$\begin{pmatrix} A & B & C \\ 1 & 2 & 3 \\ 3 & 4 & 5 \end{pmatrix} [B > D] \begin{pmatrix} D & E \\ 3 & 5 \end{pmatrix} = \begin{pmatrix} A & B & C & D & E \\ 3 & 4 & 5 & 3 & 5 \end{pmatrix} \tag{2.9}$$

Die resultierende Relation enthält die Komponenten beider Eingangsrelationen und die Elemente setzten sich entsprechend zusammen. Dabei werden nur die Elemente ausgewählt, die die Bedingung (hier $B > D$) erfüllen. Neben diesem Join existieren noch weitere, wichtig ist hier der natürliche Join (*Natural Join*). Bei diesem werden je eine Komponente bestimmt und die neue Relation enthält

dann alle Tupel, bei denen die zugehörigen Werte gleich sind. Wenn dabei die gleiche Komponente in 2 Relationen vorhanden ist, dann wird die nun doppelte Komponente entfernt. Man schreibt hierfür $R(A, B, C)[C \cdot D]T(D, E)$. Dies entspricht $R(A, B, C)[C = D]T(D, E)$, wobei die Komponente D nicht Teil des Resultats ist, da die Werte ja laut Bedingung bereits gleich C, also doppelt vorhanden, sind.

2.2.1.4. Projektion

Aus einer Relation kann eine neue dadurch erzeugt werden, dass nur bestimmte Komponenten ausgewählt werden. Beispielsweise kann aus R eine Relation erzeugt werden, die nur die Komponenten A und B enthält:

$$R[A, B] = \begin{pmatrix} A & B \\ 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{2.10}$$

2.2.2. Relationale Datenbanken als Implementierung des relationalen Modells

Relationale Datenbanken implementieren das relationale Modell. Dabei wird eine Menge von Tabellen verwaltet, wobei Tabellen den konzeptionellen Relationen entsprechen. Tabellen bestehen aus Spalten, denen ein Datentyp und ein Name zugewiesen ist. Spalten entsprechend somit den Komponenten. Die Datenbank verwaltet neben den Daten auch Informationen darüber, welche Tabellen gespeichert sind, wie diese heißen und welche Spalten mit welchem Typ und Namen vorhanden sind. Diese Daten über den Aufbau der Daten werden als Katalog bezeichnet [8]. Hier wird auch die Bezeichnung Schema verwendet. Das Schema beschreibt, wie die Daten aufgebaut sind und wie Operationen darauf auszuführen sind [17]. Für relationale Datenbanken ist dabei irrelevant, wie die Daten konkret abgespeichert werden, der Zugriff erfolg immer nur aus relationaler Sicht. Dazu wird die Anfragesprache SQL verwendet. Im folgenden wird kurz gezeigt, wie die relationalen Operatoren in einer SQL Query deklariert sein können.

```
1  SELECT columnA, columnB, columnD
2  FROM TableA
3  INNER JOIN TableB ON TableA.columnB = TableB.columnE
4  WHERE columnB < 2
```

Listing 2.1: Eine Beispiel-Anfrage in SQL

In der Anfrage in Listing 2.1 finden sich alle Operatoren. In Zeile 2 wird angegeben, dass die Tabelle (Relation) $TableA$ angefragt werden sollen. Es wird ein Join mit $TableB$ mit Bedingung $columnB = columnE$ ausgeführt (Zeile 3). Eine Restriktion mit $columnB < 2$ wird ausgeführt (Zeile 4) und schließlich eine Projektion nach den Spalten $columnA$, $columnB$, $columnD$ (Zeile 1).

Der Zustand einer Tabelle in einer relationalen Datenbank kann als Relation, wie beschrieben, angesehen werden. Da sich die Datenbank ändern kann, stellt die Relation dabei immer nur den aktuell gültigen Zustand dar. Im folgenden Abschnitt werden temporale Datenbanken vorgestellt, welche es auch ermöglichen vergangene Zustände zu verarbeiten.

2.3. Temporale Datenbanken

Temporale relationale Datenbanken verwalten nicht nur den aktuellen Zustand der Daten, sondern ermöglichen auch die Verarbeitung vergangener Zustände. Die Beschreibungen hier beziehen sich auf den Entwurf von Snodgrass [22]. Wird eine Änderung an den Daten durchgeführt, dann wird der „alte“ Zustand gesichert.

Dazu kann zuerst zu jeder Tabelle eine Zeitdimension *transaction time* hinzugefügt werden. Diese repräsentiert, wann eine Zeile in der Datenbank gültig ist. Dabei wird je Start- und Endzeitpunkt gespeichert. Wenn eine neue Zeile eingefügt wird, dann wird als Start-Zeit die Zeit gesichert, zu der die `INSERT` Operation durchgeführt wurde. Wird die Zeile geändert, dann wird der End-Zeitpunkt auf die Zeit der `UPDATE` oder `DELETE` Operation gesetzt. Statt dabei den alten Zustand zu überschreiben, wird dieser kopiert und separat gespeichert. Mit jeder Änderung wird so eine Kopie der Zeile erzeugt. Mit den zugehörigen Spalten wird klar, wann welche Kopie gültig ist oder war. Diese Gültigkeits-Zeiträume dürfen sich für jede Zeile nicht überschneiden.

Neben *transaction time* wird auch eine weitere Zeitdimension eingeführt, die *valid time* oder *application time* genannt wird. Diese beschreibt nicht, wann Daten in der Datenbank gültig sind, sondern repräsentieren eine Gültigkeit in der realen Welt, beispielsweise die Lautzeit eines Vertrages, oder Ähnliches. Diese kann manuell gesetzt werden und historische Werte werden entsprechend nicht gespeichert. Sie verhält sich wie eine normale Property. Datenbanken mit diesen zwei Zeiten werden auch *bitemporal* genannt.

Im SQL Standard in Version von 2011 wurden dazu einige Konzepte für SQL definiert, die die Verarbeitung für solche Arten von Tabellen ermöglichen [13]. Nach dem Standard wir die *transaction time* als zwei Spalten gespeichert, die je einen Zeitpunkt-Datentyp haben. Für diese kann die *transaction time* als Zeitperiode definiert werden. Die Spalten werden dann mit jeder Änderung vom System aktualisiert. Eine manuelle Änderung ist *nicht* möglich. Damit nun noch Änderungen aufgezeichnet, wird eine Funktionalität, welche System-Versionierung genannt wird, aktiviert. Ältere Zustände werden dann separat gespeichert. Dabei kann je nur der aktuell gültige Zustand verändert werden. Historische Daten können nur angefragt werden. Dazu wird zu `SELECT` Anfragen ein neues Konzept eingeführt. Über eine `FOR SYSTEM_TIME` Klausel nach der Auswahl der anzufragenden Tabelle (siehe z.B. Zeile 2 in Listing 2.1) wird bestimmt, welcher Zustand der Tabelle angefragt werden soll. Dazu ist ein so genanntes temporales Prädikat anzugeben. Diese basieren auf den Zeit-Relationen von Allen [2]. Neben den Prädikaten kann auch ein konkreter Zeitpunkt angegeben sein. Die resultierende Tabelle enthält dann alle (ggf. historischen) Einträge, die entweder zum Prädikat passen oder, falls stattdessen ein Zeitpunkt gewählt wurde, genau zu einem bestimmten Zeitpunkt gültig sind oder waren. In Tabelle 2.1 sind in SQL:2011 definierte Prädikate aufgelistet. Neben `PRECEDES` und `SUCCEEDS` ist auch je `IMMEDIATELY PRECEDES` (für `SUCCEEDS` analog) unterstützt, wobei dann alle Elemente gewählt werden, die direkt vor Zeitraum gültig sind, wo die Zeiträume also direkt aufeinander folgen.

Prädikat	Beschreibung
`AS OF`	Elemente, die zu einem bestimmtem Zeitpunkt gültig sind
`CONTAINS`	Elemente, deren Gültigkeitszeitraum einen Zeitpunkt enthalten, also die zu diesem Zeitpunkt gültig sind
`OVERLAPS`	Elemente, deren Zeitraum sich mit einem anderen überschneidet, also die mindestens zu einem bestimmten gemeinsamen Zeitpunkt gültig sind
`EQUALS`	Elemente, deren Zeitraum genau einem anderen entspricht
`PRECEDES`	Elemente, die vor einem Zeitraum gültig sind
`SUCCEEDS`	Elemente, die nach einem Zeitraum gültig waren

Tabelle 2.1.: Temporale Prädikate in SQL:2011

2.4. Temporale Graphen

Das Konzept temporaler Datenbanken lässt sich auch auf Graphen übertragen. So kann jedem Graph-Element, also Knoten und Kanten eine oder beide Zeitdimensionen zugewiesen sein. Wieder wird *transaction time* dafür genutzt, um abzubilden, wann ein Graph-Element im Modell gültig ist. *Valid time* ist wieder effektiv eine Property, welche dazu genutzt werden kann, um Gültigkeiten in der realen Welt abzubilden. Das hier verwendete Graph-Modell orientiert sich an der Definition von Rost [19].

Die *transaction time* muss so abgebildet sein, dass der Graph zu jedem beliebigen Zeitraum konsistent ist. Dies meint, dass die Graph-Daten dem Modell entsprechen. So müssen zu jeder Kante immer die Quell- und Zielknoten so lang gültig sein, wie die Kante gültig ist. Andernfalls kann es Zeiten geben, an denen Kanten gültig sind, deren Quell- oder Zielknoten nicht existieren (also nicht gültig sind).

Hier wird nun eine Definition des Modells gegeben:

Definition 2 (Temporaler Property Graph)**.** Seien Ω_v und Ω_t Zeitdimensionen, also total geordnete Mengen von Zeitpunkten. Dann ist ein *temporaler Property Graph* $\mathbb{G}_t$ als Tupel definiert durch

$$\mathbb{G}_t = \langle V, E, \mathcal{I}, \mathcal{L}, P, A, label, id, source, target, \kappa, from_v, to_v, from_t, to_t \rangle \tag{2.11}$$

Dabei ist $\langle V, E, \mathcal{I}, \mathcal{L}, P, A, label, id, source, target, \kappa \rangle$ ein *Property Graph* gemäß Definition 1. $from_v, to_v : V \cup E \to \Omega_v$ sind Abbildungen, die jedem Element eine *valid time* zuordnen. Analog sind $from_t, to_t : V \cup E \to \Omega_t$ Abbildungen, die jedem Element die *transaction time* zuordnen. $from_v$ und $from_t$ bestimmen dabei je den Startzeitpunkt gemäß der Zeitdimensionen und to_v, to_t den Endzeitpunkt. Für jedes Element $e \in V \cup E$ müssen

$$from_v(e) < to_v(e) \text{ und } from_t(e) < to_t(e) \tag{2.12}$$

erfüllt sein. Für temporale Graphen ist es möglich, dass mehrere Elemente mit gleicher ID existieren, sofern diesen unterschiedliche *transaction times* zugewiesen sind. Diese Zeiträume für zwei Elemente

mit gleicher ID dürfen sich allerdings nicht überschneiden. Insbesondere lassen sich Elemente über ID und Endzeitpunkt der *transaction time* identifizieren. Für alle $e, f \in V \cup E$ gelten:

$$id(e) = id(f) \wedge to_t(e) = to_t(f) \implies e = f \tag{2.13}$$

$$e \neq f \implies [from_t(e), to_t(e)) \cap [from_t(f), to_t(f)) = \emptyset \tag{2.14}$$

Im Gegensatz zu nicht-temporalen Graphen müssen IDs also nicht eindeutig sein. Nur paare aus ID und *transaction time* identifizieren Elemente, beziehungsweise deren Zustände. Im Konzept wird dabei davon ausgegangen, dass Elemente verändert werden können (in der Implementierung ist hierfür eine API gegeben). Veränderungen von Elementen führen analog zu temporalen Datenbanken dazu, dass eine Kopie des Elementes erzeugt wird, welches dann separat als historisches Element gespeichert werden soll.

Zur Verarbeitung von temporalen Graphen genügt es, diese auf eine temporale, relationale Sicht abzubilden. Für diese existieren Implementierungen, welche Zeit-Informationen und historische Werte berücksichtigen. Es muss nur garantiert sein, dass die relationale Sicht immer einen gültigen Graph-Zustand repräsentiert. In dieser Arbeit werden dafür Integritätsbedingungen genutzt.

2.5. Integritätsbedingungen für Datenbanken

Integritätsbedingungen, auch *Constraints* genannt, sind Eigenschaften, die die Daten in einer Datenbank zu jedem beliebigen Zeitpunkt erfüllen müssen. Sie sind Teil des Schemas (Katalogs) der Datenbank. Wird die Datenbank verändert, dann wird zuerst geprüft, ob die Änderung den Bedingungen gerecht wird. Falls nicht, dann wird die Änderung abgelegt. [8]

Besonders wichtig für diese Arbeit sind Primärschlüssel- und Fremdschlüssel-Bedingungen (bzw. Constraints), welche bereits von Codd mit dem relationalen Modell eingeführt wurden [4, 5].

2.5.1. Primärschlüssel

Relationen sind Mengen von Tupeln, also kann es pro Relation auch keine zwei Tupel geben, die bezüglich aller Komponenten gleich sind. Falls für eine Relation ein Primärschlüssel definiert ist, dann wird zusätzlich garantiert, dass Tupel entsprechend dieses Schlüssels eindeutig sind. Dazu können für alle Relationen eine oder mehrere Komponenten als Schlüssel ausgewählt werden. Eine Relation kann dann keine zwei Tupel enthalten, deren als Schlüssel festgelegte Spalten die gleichen Werte tragen. In dieser Arbeit wird dies ausgenutzt, um Element zu identifizieren. Ein Graph-Element wird auf ein Tupel abgebildet und die ID des Elements wird auf eine Komponente abgebildet. Diese ist dann als Primärschlüssel definiert, wodurch garantiert wird, dass keine zwei Tupel existieren, die das gleiche Element repräsentieren.

2.5.2. Fremdschlüssel

Mit einer Fremdschlüssel-Beziehung können zwei Relationen „verknüpft“ werden. Eine oder mehrere Spalten einer Relation können dazu ausgewählt werden. Für diese wird eine Beziehung zu einer anderen Relation garantiert. Je eine Spalte referenziert eine Spalte einer anderen Relation (Ziel-Relation). Ist einer Relation eine Fremdschlüssel-Beziehung zugeordnet, dann kann für jedes Tupel der entsprechendes Schlüssel bestimmt werden. In der Ziel-Relation muss dann ein Tupel existieren, dessen passende Spalte genau diesen Wert trägt. Wird das Tupel der Ziel-Relation entfernt, dann ist die Beziehung entweder verletzt *oder* das passende Tupel in der Relation mit der Fremdschlüssel-Bedingung muss auch entfernt werden. Es wird also eine Referenz dargestellt, der immer gefolgt werden kann.

2.5.3. Zeit-Constraints

Für temporale Datenbanken können nur zusätzliche Integritätsbedingungen, welche auf den Zeitdimensionen basieren, definiert sein. [22]

So ist generell immer garantiert, dass sich für ein Tupel und dessen historische Kopien die *transaction time* Perioden nicht überschneiden. Das gleiche kann auch für *valid time* verlangt werden, dies ist allerdings optional.

3. Verwandte Arbeiten

Mit Gradoop, von Junghanns et. al. [11] existiert bereits ein System, mit dem *Property Graphen* analysiert werden können. Gradoop nutzt dabei, im Gegensatz zu dieser Arbeit, ein erweitertes Property Graph Modell, EPGM (Extended Property Graph Model). In diesem Modell existieren neben Knoten und Kanten, denen wie üblich Properties und Typ-Label zugeordnet werden, auch Teilgraph als Objekte des Modells. Diese werden mit den gleichen Attributen versehen. Mehrere solche Teilgraphen (da logische Graphen genannt), werden zu Graph-Mengen (*Graph Collections*) zusammengefasst. Elementen wird dabei über eine Abbildung je eine Menge von logischen Graphen zugeordnet. Es sind Operatoren definiert, die auf einem oder mehreren solchen Graphen agieren, beispielsweise um Schnittmengen, Vereinigungen oder Differenzen von Graphen zu bestimmen. Die Konzepte dieses Modells finden in dieser Arbeit keine Anwendung. Hier stellt die Datenbank nur einen Graph dar. Gradoop setzt das Modell im verteilten Verarbeitungssystem Apache Flink[1] um [12]. Elemente, bzw. das Graph-Modell, sind dabei als Java Objekte implementiert.

Einige Erweiterungen von Gradoop dienen als Inspiration für diese Arbeit. Dazu stellen Saalmann [21] und Adameit [1] je EPGM-kompatible Modelle und Implementierungen vor, welche auf relationalen Sichten basieren.

3.1. EPGM auf relationalen Sichten

Saalmann nutzt eine API von Apache Flink aus, welche eine relationale Verarbeitung von Daten ermöglicht [21]. Die Daten werden dabei immer noch auf einem verteilten System verarbeitet und *nicht* in einer Datenbank gespeichert. Saalmann definiert dafür auch mehrere relationale Sichten, bzw. Abbildungen. Adameit verwendet eine ähnliche Abstraktion des EPGM auf ein relationales Modell [1]. Ziel dieser Arbeit ist, im Gegensatz zu Saalmann, nicht die Verwendung der relationalen API von Apache Flink. An stelle dessen wird hier ein weiteres verteiltes Framework zur Verarbeitung von Daten eingesetzt: Apache Spark[2].

Beide Arbeiten stellen verschiedene Schemata zur Darstellung des EPGM vor. Diese sind hier besonders interessant und finden in abgewandelter Form, nämlich ohne die Abbildungen für logische Graphen und mit temporalen Erweiterungen, Anwendung in Konzeption dieser Arbeit.

3.1.1. GVE-Schema

Im GVE-Schema nach Saalmann [21] und Adameit [1]. Werden Knoten und Kanten durch je eine Tabelle repräsentiert. Zusätzlich wird eine Tabelle für logische Graphen eingeführt. Jede Tabelle speichert dabei ID, Label und Properties der Elemente. Die Kanten-Tabelle speichert Referenzen auf die Knoten-Tabelle über die passenden IDs der Quell- und Zielknoten. Knoten und Kanten werden je noch logische Graphen, in denen diese enthalten sind, als Referenz auf die Graph-Tabelle

[1] `https://flink.apache.org/`
[2] `https://spark.apache.org/`

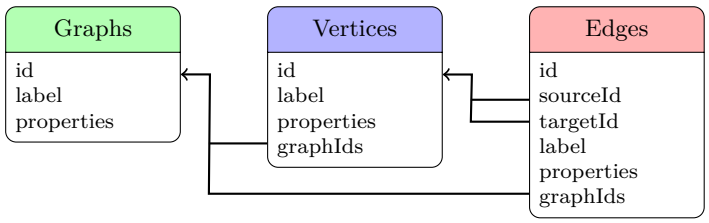

Abbildung 3.1.: GVE-Schema nach Saalmann

(Bild aus Adameit [1])

zugewiesen. In Abbildung 3.1 wird dies veranschaulicht. Adameit sieht den Vorteil dieses Schemas darin, dass Daten kompakt abgelegt sind. Von Nachteil ist die Tatsache, dass große Tabellen entstehen und außerdem alle Attribute in einer Zeile abgebildet sind. Im verteilten System muss diese immer komplett verarbeitet werden, auch wenn nicht alle Attribute zur Verarbeitung einer Operation benötigt werden.

3.1.2. Vertikales Schema

Um die besser aufzuteilen stellt Saalmann das vertikale Schema vor [21]. Dabei wird die Graph-Zugehörigkeit in eine Tabelle ausgelagert, die Elemente und logische Graphen referenziert. Properties werden ebenso in eine Tabelle ausgelagert. Ein ähnliches Schema wird in dieser Arbeit auch verwendet. In Abbildung 3.2 ist dieses veranschaulicht. Die Aufteilung auf mehrere Tabelle wurde dabei durchgeführt, damit Daten in einem verteilten System, auf dass die Arbeiten von Adameit und Saalmann aufbauen, entsprechend auch verteilt verarbeitet werden können. Saalmann stellt noch ein weiteres Schema vor, in dem jede einzelne Property in eine Tabelle ausgelagert wird [21]. In dieser Arbeit wird dieses Konzept ähnlich umgesetzt, indem einzelne Properties in Spalten der Element-Tabellen abgebildet werden.

3.1.3. Tables for Labels Schema

Adameit stellt mit dem Tables for Labels (kurz TFL, in dieser Arbeit auch Label-partitioniertes Schema) ein weiteres nützliches relationales Schema vor [1]. Hier sind Knoten- und Kanten-Typen je nach Label aufteilt. Properties sind in eine Tabelle ausgelagert, wobei diese als Objekte in Spalten gespeichert sind. Als Vorteil sieht Adameit die Möglichkeit des Verarbeitungssystems, nur bestimmte Tabellen-Sicht, also nur Teile der Daten zu verarbeiten, wenn ein Operator zum Beispiel nur Elemente bestimmter Label erwartet.

In dieser Arbeit wird auch ein relationales Schema definiert, auf das Graphen abgebildet werden. Die Graphzugehörigkeit, also das Konzept logischer Graphen wird hierbei nicht benötigt. Diese Arbeit vereint die Konzepte der TFL, GVE, horizontalen und vertikalen Schemas.

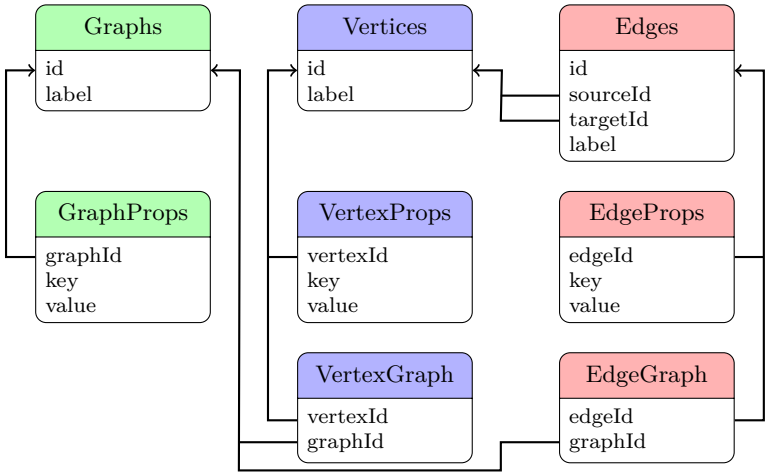

Abbildung 3.2.: Vertikales Schema nach Saalmann

(Bild aus Adameit [1])

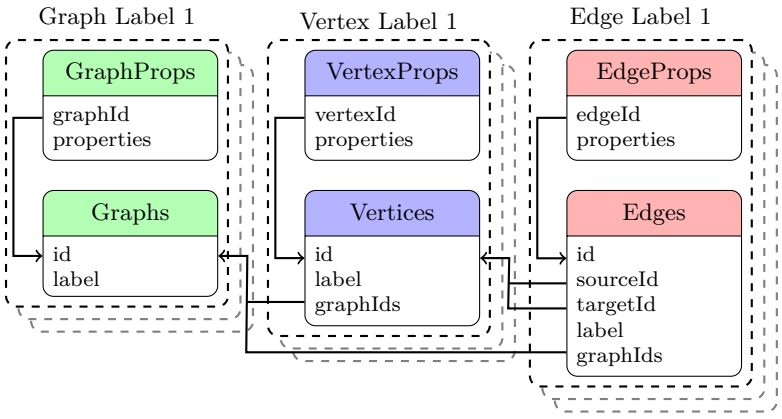

Abbildung 3.3.: Tables for Labels (TFL) Schema nach Adameit

(Bild aus Adameit [1])

3.2. Temporale Graphen

Rost et. al. stellen in [20] ein temporales Graph-Modell vor, welches das EPGM erweitert. Dieses *Temporal Property Graph Model* ordnet Elementen von Graphen Zeitattribute zu. Dabei werden die üblichen zwei Zeitdimensionen für *transaction time* und *valid time* genutzt. Die Abbildung erfolgt dabei über vier Zeitstempel, die zu beiden Dimensionen je Start- und Endzeitpunkt beschreiben.

Für das temporale Graph-Modell werden neue Operatoren zur Verarbeitung eingeführt. Der *Snapshot* Operator wird genutzt, um einen Graph-Zustand zu extrahieren. Dafür sind Prädikate als Funktionen definiert, die für jedes Element bestimmten, ob es entsprechend gültig war. Diese Funktionen richten sich wieder nach der Zeit-Algebra von Allen [2].

Neben dem *Snapshot* Operator wird auch ein *Difference* Operator vorgestellt, der dazu genutzt werden kann, um zwei Graph-Zustände (Snapshots) zu vergleichen. Es wird jedes Element, was hinzugefügt oder entfernt wurde entsprechend markiert.

Die Implementierung ist als Erweiterung von Gradoop erfolgt, nutzt damit wieder ein verteiltes System und keine relationale Datenbank. Trotzdem ist diese Modell hilfreich und wird zum Teil in dieser Arbeit adaptiert. Der *Snapshot* Operator wird implementiert.

3.3. Relationale Algebra für Graphen

In einer Arbeit von Cyganiak wird ein Verfahren vorgestellt, mit dem SPARQL-Anfragen in eine relationale Algebra übersetzt werden können [6]. SPARQL ist dabei eine Anfragesprache für das RDF-Modell, welches ebenso ein Graph-Modell darstellt, sich aber vom *Property Graph* Modell recht stark unterscheidet. In RDF werden Daten als Tripel aus Subjekt, Prädikat und Objekt modelliert. Dadurch lassen sich Kanten und Properties repräsentieren, da die Objekt-Komponente der Tripel entweder ein RDF-Objekt oder einen Wert (analog zu Property-Werten) modellieren kann.

SPAQL ist eine Anfragesprache für Graph-Strukturen in einem RDF-Graph (aus Subjekten, die über Tripel verbunden sind und denen auch über Tripel Properties zugewiesen sind). Cyganiak stellt dafür zuerst Abbildungen auf Relationen vor. Im einfachsten Fall werden Tripel auf Relationen mit drei Komponenten abgebildet. Die Tripel haben also im Modell und in Abbildung die gleiche Darstellung. Außerdem können einzelne Entitäten in Tabellen abgelegt sein, wobei in jeder Spalte das *Objekt* zu einem bestimmten *Prädikat* gespeichert ist. Diese Spalte kann also als eine Art Property-Spalte mit festem Key gesehen werden. Anfragen auf diesen Relationen-Arten sind wieder über die üblichen Selektionen, Projektionen und Joins [5] formalisiert.

Zur Definition der Tabellen-Strukturen, welche RDF-Graphen abbilden sollen existiert der Standard R2RML [7] von Cyganiak et. al..

Marton et. al. stellen in [14] eine Erweiterung der relationalen Algebra (aus Selektion, Projektion und Join nach Codd [4, 5]) vor. Hierbei dient die Anfragesprache openCypher als Basis. Ziel ist dabei nicht die Übersetzung von Anfragen in ein herkömmliches Modell (wie in dieser Arbeit),

sondern die Erweiterung des Modells um Konzepte, welche nötig sind, um openCypher Anfragen formal zu beschreiben. Ziel ist entsprechend eine formale Definition der Anfragen. Dies soll zukünftige Implementierungen der openCypher Sprache vereinfachen. Die Konzepte finden in dieser Arbeit keine Anwendung, da als das Ziel hier die Übersetzung in ein Modell mit existierenden Implementierungen (dem relationalen Modell, als Basis relationaler Datenbanken) gesetzt ist.

4. Konzepte

Dieses Kapitel stellt einige Konzepte vor, die in Entwurf und Entwicklung des Prototyps Anwendung finden. Hierzu wird zuerst das entwickelte Graph-Schema vorgestellt, anschließend gezeigt, wie Operationen auf Daten entsprechend dieses Schemas umgesetzt werden können.

4.1. Graph-Schema

Das Graph-Schema stellt eine Menge von Metadaten dar, die den Aufbau, bzw. die Struktur der verwalteten Daten beschreiben. Nach Pokorný [17] beschreibt ein *Graph Datenbank Schema* neben den zu Grunde liegenden Strukturen und Datentypen auch wie Operatoren auf diesen aufgeführt werden können. Weiter sind Bedingungen festzulegen, die eine Datenbank entsprechend des Schemas erfüllen muss. Diese werden als Integritätsbedingungen bezeichnet.

In dieser Arbeit wird ein Prototyp entworfen, mit dem Graph-Daten auf ein relationales Datenmodell abgebildet werden. Entsprechend meint Graph-Schema hier die vorhandenen Typen von Elementen, welche Attribute diese besitzen und wie sie auf ein relationales Schema abgebildet sind.

4.1.1. GVE-Schema

Ein einfache Möglichkeit der Verwaltung von Graph-Daten ergibt sich daraus, Elemente des Graphen nur nach Knoten und Kanten aufzuteilen. So unterteilt sich das Schema zuerst in genau diese zwei Datentypen. Das hier definierte Schema orientiert sich zum großen Teil am GVE-Schema nach Adameit [1]. Hierzu werden Knoten auf genau eine (relationale) Tabelle abgebildet, welche Attribute als Spalten speichert. Es wird eine *Vertices*-Tabelle verwaltet, wobei jeder Eintrag, also jede Zeile der Tabelle, genau einen Knoten des Graphen repräsentiert. Dazu sind Spalten definiert, die ID und Typ-Label der Knoten darstellen. Kanten werden ebenso in eine *Edges*-Tabelle abgebildet, die gleich aufgebaut ist, jedoch zusätzlich IDs der zugehörigen Quell- und Zielknoten in Spalten speichert. Dazu wird je eine Spalte hinzugefügt, welche die ID-Spalte der Knotentabelle referenziert.

Damit sichergestellt werden kann, dass die zu Grunde liegenden relationalen Daten immer einen Graphen ergeben, der dem verwendeten *Property Graph*-Modell entspricht, müssen noch folgende Integritätsbedingungen festgelegt werden: Jedem Element muss eine eindeutige ID zugewiesen sein, also muss die entsprechende Spalte in beiden Tabellen immer mit einem Wert versehen sein. Dieser Wert muss eindeutig sein und die Elemente identifizieren, also wird diese Spalte als Primärschlüssel verwendet. Weiter muss für jede Kante garantiert sein, dass entsprechende Quell- und Zielknoten existieren. Dazu genügt es festzulegen, dass die Attribute, welche die IDs beider Knoten speichern, gesetzt sind und auf existierende Elemente verweisen. Entsprechend werden je Integritätsbedingungen definiert, welche garantieren, dass die zugehörigen Spalten mit Werten versehen sind und auch auf einen existierenden Eintrag in der Knoten-Tabelle verweisen.

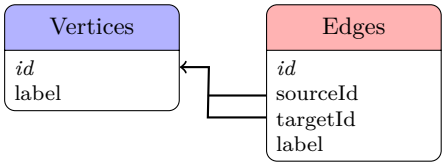

Abbildung 4.1.: Tabellen im GVE-basierten Schema

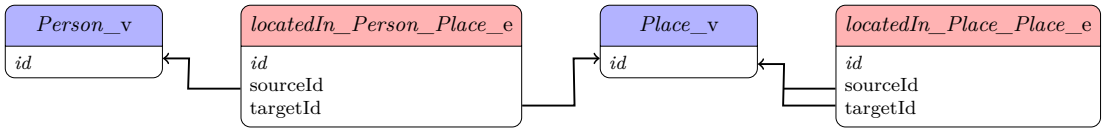

Abbildung 4.2.: Beispiel für Tabellen mit nach Label partitionierten Schema

Abbildung 4.1 zeigt die im GVE-Schema verwendeten Tabellen. Primärschlüssel sind *kursiv* und Fremdschlüsselbeziehungen mit Pfeilen dargestellt.

4.1.2. Nach Label partitioniertes Schema

Neben der simplen Trennung von Graph-Elementen nach Knoten und Kanten, ist es möglich, Typ-Label zur Partitionierung zu nutzen. Dieses Schema orientiert sich am *Tables for Labels*-Schema (*TFL*) von Adameit [1]. Für jedes Typ-Label von Knoten wird eine eigene Knoten-Tabelle erzeugt, in der jede Zeile wieder genau einen Knoten repräsentiert. Die Tabellen sind dabei so ähnlich aufgebaut wie die *Vertices*-Tabelle aus Unterabschnitt 4.1.1. Da hier jedoch das Typ-Label der Elemente implizit klar ist, da pro Tabelle nur ein einziges Typ-Label gespeichert ist, muss dieses nicht mehr als Spalte abgebildet werden. Allerdings muss das System eine Abbildung von Typ-Label auf Tabelle verwalten.

Ebenso können Kanten nach Typ-Label aufgeteilt werden. Wie auch im GVE-Schema soll die Konsistenz des Graphen in der Datenbank durch Integritätsbedingungen garantiert werden. Auch hier muss damit eine Fremdschlüsselbeziehung zwischen Kanten-Tabelle und zugehörigen Knoten-Tabellen bestehen, um die Existenz verbundener Knoten zu garantieren. Für die Spalten, welche die IDs der zugehörigen Quell- und Zielknoten speichern, existiert also eine Referenz auf *genau eine* Knoten-Tabelle. Da allerdings, wie zuvor erwähnt, das Typ-Label einer Knoten-Tabelle implizit fest ist, müssen die Typ-Label von verbundenen Knoten für jede Kanten-Tabelle ebenso fest sein. Eine Aufteilung von Kanten nur nach deren Typ-Label genügt also nicht, um alle Kanten abzubilden. Daher werden diese neben dem eigenen Typ-Label, auch nach denen der entsprechenden Quell- und Zielknoten aufgeteilt.

Abbildung 4.2 zeigt als Beispiel einen Ausschnitt eines solchen Schemas. Hier werden Personen als *Person*-Knoten dargestellt, diesen wird über einen *locatedIn*-Kantentyp ein Standort, z.B. eine Stadt zugeordnet. Diesen Orten ist wiederum mittels *locatedIn*-Kante ein Standort zugewiesen. So befinden sich beispielsweise Städte in Ländern, diese formen Kontinente, etc. Für beide Beziehungen wird das gleiche Kanten-Typ-Label ‚*locatedIn*‘ verwendet. Da dieser allerdings verschiedene Knoten-Typen verbindet, muss er auch in mehrere Tabellen abgebildet werden.

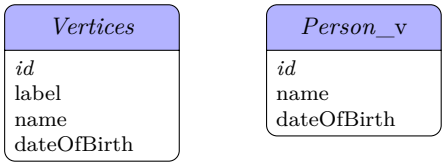

Abbildung 4.3.: Beispiel für Properties als Spalten im GVE- und Label-partitionierten Schema (v.l.n.r.)

4.1.3. Properties

Bislang wurde nur gezeigt, wie Graph-Elemente nach Art (Knoten oder Kante) und Typ-Label auf Tabellen aufgeteilt werden können. Entsprechend des verwendeten *Property Graph*-Modells kann für eine Menge von Property-Namen (*Keys*) P, den Mengen der Knoten V und Kanten E über eine partielle Abbildung ein Property-Wert aus einer Menge A zugewiesen werden:

$$(V \cup E) \times P \rightharpoonup A$$

Eine Möglichkeit der Darstellung von Properties im den zu Grunde liegenden relationalen Schemata ist die Speicherung dieser als zusätzliche Spalten. Dazu wird jedem Property-Key aus P ein Spaltenname zuordnet Typischerweise handelt es sich bei *Keys* um Zeichenketten (*Strings*), was die Zuordnung erleichtert. So können beispielsweise Personen, die wie im zuvor gegebenen Beispiel als *Person*-Knoten repräsentiert werden, Properties wie Namen, Geburtsdaten, usw. zugeordnet sein.

In Abbildung 4.3 wird für beide bisher vorgestellte Schemata gezeigt, wie das hinzufügen von Properties als Spalten das relationale Schema beeinflusst. Im nach Label partitionierten Schema wird der Tabelle, welche Knoten vom Typ *Person* speichert je eine Spalte für *name* und *dateOfBirth* Properties hinzugefügt. Da Personen im Allgemeinen ein Name und ein Geburtsdatum zugeordnet werden kann, ist diese kompakte Darstellung naheliegend. Im GVE-basierten Schema erscheint diese Art der Abbildung weniger sinnvoll, da die entsprechende Tabelle hier nicht nur *Person*-Knoten speichert, sondern Elemente beliebigen Labels. Für alle Knoten mit nicht gesetzter *dateOfBirth*-Property wird die entsprechende Spalte mit `null`-Werten gefüllt. Diese Darstellung erscheint damit, abhängig vom Anwendungsfall, weniger sinnvoll.

Ein weiterer Nachteil dieser Schema-Darstellung ergibt sich daraus, dass nur Properties gespeichert werden können, für die eine Spalte existiert und verwaltet wird. Solche Properties können nur gesetzt werden, wenn dafür das Schema verändert wird.

Um beide Nachteile zu umgehen, bietet sich die Möglichkeit Properties als separate Tabellen auszulagern. Hierfür wird eine Tabelle hinzugefügt, welche die ID des zugehörigen Elements, sowie Property-Key und -Wert enthält. Eine solche Tabelle wird für alle Element-Tabellen, also jede Knoten- oder Kanten-Tabelle, erstellt. Um jedes solche Tripel als ID, Key und Wert einem Element zuzuordnen, wird eine Fremdschlüsselbeziehung zwischen den ID-Spalten der Property-Tabelle und der Element-Tabelle definiert. Damit sichergestellt werden kann, dass zu jedem Property-Key nur höchstens ein Wert pro Element existiert, werden die Spalten für ID und Key als Primärschlüssel gewählt.

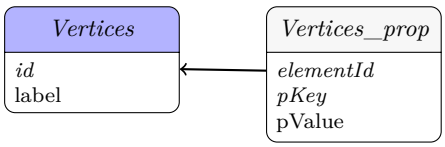

Abbildung 4.4.: Knoten-Tabelle mit entsprechender Property-Tabelle im GVE-Schema

Die resultierenden Tabellen werden in Abbildung 4.4 am Beispiel einer Knoten-Tabelle im GVE-Schema veranschaulicht.

4.1.4. Hybrider Schema-Ansatz

Bisher wurden zwei Möglichkeiten gezeigt, wie Graph-Element auf Relationen (Tabellen) aufgeteilt werden können:

Im GVE-Schema werden alle Knoten in einer, alle Kanten in einer weiteren Tabelle gespeichert. Alle Graph-Elemente werden auf diese beiden Tabellen verteilt. Dadurch ergibt sich ein kleines Schema, also eines mit wenigen Tabellen. Weiter lassen sich alle Elemente in dieses Schema abbilden, da für beide Tabellen keine Restriktion definiert sind. In diesem Schema finden sich somit wenige, aber dafür große Tabellen. Dies kann von Nachteil für Anfragen sein, da mit jedem Zugriff auf Knoten oder Kanten je nur auf eine große Tabelle zugegriffen wird. Es besteht die Vermutung dass dies einen negativen Einfluss auf die Performance mancher Operationen hat.

In einem nach Label partitionierten Schema können Daten besser aufgeteilt werden So lässt sich der Datenzugriff auf wenige Tabellen beschränken, wenn beispielsweise nur Element mit bestimmten Typ-Labels verarbeitet werden sollen. Allerdings wird das Schema mit steigender Anzahl von Typ-Labels in einem Graphen auch immer komplexer. Weiter besteht die Problematik, dass Kanten in Tabellen nicht nur nach eigenem Typ-Label, sondern auch nach denen verbundener Knoten separiert werden müssen. Schlimmstenfalls muss mit jedem neuen Label eine Tabelle für jede neue Kombination von Knoten-Labels bereitgestellt werden.

In dieser Arbeit wird nun ein Schema-Ansatz vorgestellt, der die Vorteile beider Schemata vereint, indem beide Möglichkeiten der Aufteilung gleichzeitig angeboten werden. Hierfür sind, abhängig vom Anwendungsfall, Mengen von Knoten-Labels und Tripel aus Kanten-Labels mit ihren entsprechenden verbundenen Knoten-Labels festzulegen. Für alle diese Typen werden separate Tabellen erzeugt. Für alle weiteren Elemente bietet sich die Nutzung von Tabellen wie im GVE-Schema an.

Mit diesem Ansatz lassen Elemente auf eine Mischung von Tabellen entsprechend der GVE- und Label-partitionierten Schemata abbilden. Um nun noch Properties zuzuordnen werden wieder beide Ansätze aus Unterabschnitt 4.1.3 vereint. Für jede Element-Tabelle, also für die generischen Knoten- und Kanten-Tabellen, sowie die nach Typ-Label aufgeteilten Tabellen, werden Mengen von Property-Keys festgelegt. Für die zugehörigen Properties wird je eine Spalte hinzugefügt, in der entsprechende Werte gespeichert werden. Für alle weiteren Properties kann zusätzlich eine Property-Tabelle bereitgestellt werden.

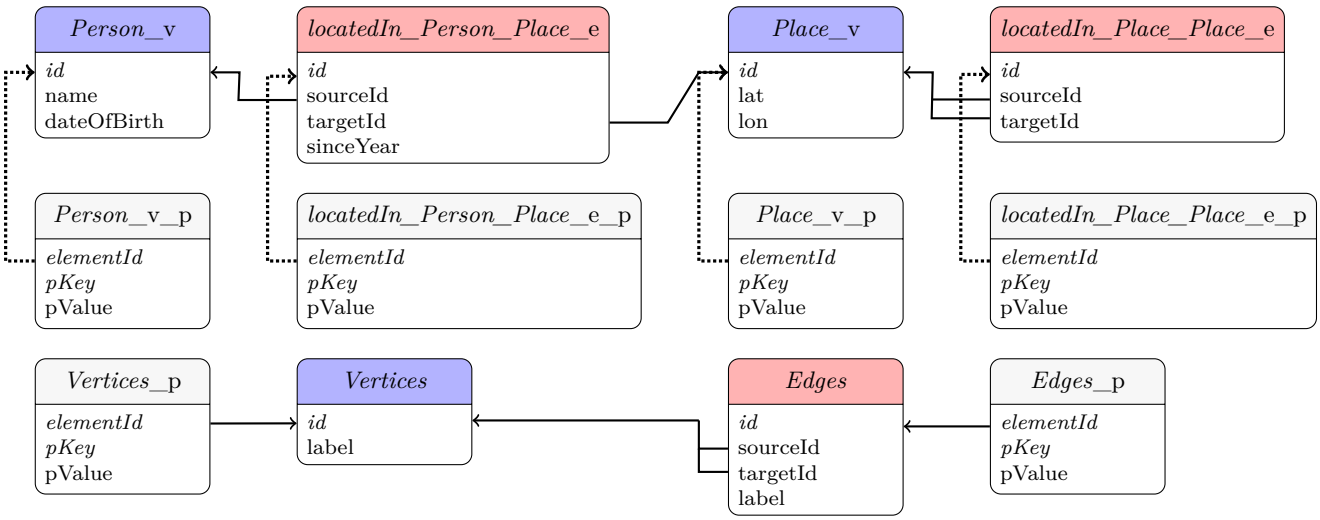

Abbildung 4.5.: Beispiel für Tabellen in einem Schema im hybriden Ansatz

In Abbildung 4.5 ist ein Beispiel eines solchen Schemas gegeben. Dieses vereint die bisher gezeigten Beispiele. Knoten-Tabellen sind rot, Kanten-Tabellen blau und Property-Tabellen grau eingefärbt. Knoten-Tabellen für Elemente eines bestimmten Typ-Labels tragen eben dieses als Tabellennamen, welcher mit Suffix *_v* ergänzt ist. Die generische Knoten-Tabelle trägt den Namen *Vertices*. Kanten-Tabellen tragen das Typ-Label der Kanten, sowie verbundener Knoten-Typen als Namen, welcher mit *_e* ergänzt ist. Die zu jeder Element-Tabelle gehörende Property-Tabelle trägt den entsprechend gleichen Namen, welcher noch mit dem Suffix *_p* ergänzt ist. Fremdschlüsselbeziehungen existieren wie zuvor und sind durch Pfeile dargestellt. In diesem Beispiel existieren separate Tabellen für die Knoten-Typen *Person* und *Place*, sowie die Kanten-Typen *locatedIn* zwischen diesen Knoten-Typen. Es sind je die Properties *name* und *dateOfBirth* (für *Person*), *lat* und *lon* (für *Place*), sowie *sinceYear* (für *locatedIn* zwischen *Person* und *Place*) als eigene Spalten abgebildet. Hier sei bemerkt, dass sowohl Spalten-, als auch Tabellennamen beliebig gewählt sein können. Die Abbildung von Typen auf Tabellen muss jedoch eindeutig sein. Da die Knoten- und Kanten-Typen, die in separate Tabellen abgebildet werden sollen, beliebig gewählt sein können, ergibt sich auch die Möglichkeit *keine* Typen zu separieren. Das GVE-basierte Schema kann also als Sonderfall den hybriden Schemas definiert werden. Gleiches gilt für das Label-partitionierte Schema. Sofern für jeden abzubildenden Typ ein separates Paar von Element- und Property-Tabellen existiert, werden generische *Vertices* und *Edges* Tabellen möglicherweise nicht benötigt.

4.1.5. Schema für temporale Daten

Im vorherigen Abschnitt wurde ein flexible Art der Abbildung von Knoten- und Kanten auf Relationen vorgestellt. Hier wird nun noch gezeigt, wie dies durch temporale Attribute ergänzt werden kann, um temporale Graphen abzubilden. Entsprechend des dargelegten relationalen Schemas wird jedes Element im Graph durch einen Eintrag in einer Element-Tabelle repräsentiert. Ändert sich das Element im Verlauf der Zeit, so ändern sich auch die Spaltenwerte der korrespondierenden Zeile. Sollen Elemente nun Gültigkeit als Zeitraum zugewiesen werden, so muss dieser mit der Zeile

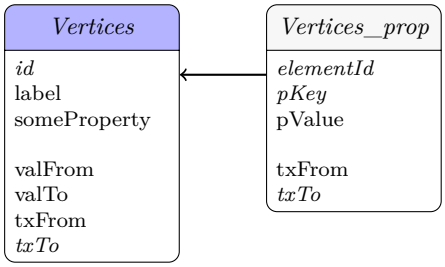

Abbildung 4.6.: Beispiel einer generischen Knoten-Tabelle mit Property-Tabelle und bitemporalen Attributen

in der Element-Tabelle gespeichert werden. Für bitemporale Tabellen existieren zwei Arten von „Gültigkeiten“ (Perioden) [22]:

Valid Time Der Zeitraum, in dem ein Element in der Realität gültig ist, beispielsweise die Laufzeit eines Arbeitsvertrages, die Dauer einer Freundschaftsbeziehung in einem sozialen Netzwerk, etc.

Transaction Time Der Zeitraum, in dem ein Fakt in einer Datenbank gültig ist. Dieser beginnt mit dem Einfügen eines Eintrages in die Datenbank und endet, sobald dieser entweder geändert oder gelöscht wird.

Wenn einem Element-Typ eine *valid time* Periode als Paar von Zeitpunkten zugewiesen ist, dann genügt es, diese zur Tabelle als zwei Spalten hinzuzufügen. Wie auch für Properties, die als Spalte der Element-Tabellen abgebildet sein können, werden die Start- und End-Zeitpunkte der *valid time* Periode in diesen Spalten abgelegt.

Da die *transaction time* eines Element nur die Zeit darstellt, in der das Element in der Datenbank gültig ist, kann diese nicht manuell gesetzt werden [13]. Um diese Zeit abzubilden, sind also auch wieder zwei Spalten hinzuzufügen, die allerdings von der Datenbank verwaltet werden. Diese Periode kann also nicht von Element auf Datenbank (bzw. Tabelle) abgebildet werden, sondern nur von Datenbank auf Element, sobald dieses einfügt/abgebildet wurde. Da sich mit Änderungen von Elementen auch Properties ändern können, die *nicht* als eigene Spalten gespeichert sind, müssen die gleichen Zeitspalten für *transaction time* auch für entsprechende Property-Tabellen bereitgestellt sein.

Da für temporale Tabellen mehrere Einträge mit gleichem Primärschlüssel, aber verschiedener Gültigkeitszeit existieren können, müssen Primärschlüssel ergänzt werden [22]. Hier wird der Zeitstempel, welcher das Ende der *transaction time* repräsentiert, zum Primärschlüssel hinzugefügt. Dieser ist immer eindeutig, da immer nur eine Version pro Element gültig, also in der Datenbank vorhanden, sein kann. Die Perioden können sich also nie überschneiden.

Abbildung 4.6 zeigt neu hinzugefügte temporale Attribute am Beispiel der generischen Knoten-Tabelle mit ihrer zugehörigen Property-Tabelle.

4.2. Operationen für Graphen

Zuvor wurde ein flexibles Schema vorgestellt, mit dem (temporale) Property-Graphen auf Relationen abgebildet werden können. In diesem Abschnitt wird nun beschrieben, wie damit einige Operationen auf Graphen umgesetzt werden. Diesen sollen auf dem zu Grunde liegenden relationalen Schema umgesetzt werden. Hierzu wird demonstriert, wie Operationen auf Graph-Daten entsprechend des Schemas auf relationale Operationen übersetzt werden.

Zuerst wird hierfür beschrieben, wie Graph-Elemente in der Datenbank gespeichert, gelesen, aktualisiert und gelöscht werden können. Anschließend werden noch komplexere Operatoren vorgestellt.

4.2.1. Abbildung von Graphen auf Relationen

4.2.1.1. Zuordnung von Tabellen zu Elementen

Um ein konkretes Element, also einen Knoten oder eine Kante einzufügen, muss zuerst bestimmt werden, welche Tabellen hierfür zu verwenden sind. Wie zuvor beschrieben, stellt das im Rahmen dieser Arbeit entwickelte Schema eine Abbildung von Knoten- und Kanten-Typen auf Relationen dar. Entsprechend des hybriden Schemas können diese entweder auf Tabellen für einen bestimmten Typ oder auf eine generische Knoten- und Kanten-Tabelle abgebildet werden. Hierfür wird zuerst eine Abbildung von Knoten- bzw. Kanten-Typ auf Tabellen definiert. Im folgenden wird die Bezeichnung *Kanten-Header* für Tripel aus Kanten-Label, sowie Knoten-Label der verbundenen Quell- und Zielknoten genutzt. Dies wird benötigt, um Kanten nicht nur nach Label, auch nach entsprechenden Knoten-Labels aufzuteilen. In Unterabschnitt 4.1.2 werden Gründe hierfür genannt.

Definition 3. Sei $\mathcal{L}$ eine Menge von Typ-Labels, $\mathcal{L}_V \subseteq \mathcal{L}$ eine Mengen von Knoten-Labels und $\mathcal{L}_E \subseteq \mathcal{L} \times \mathcal{L}_V \times \mathcal{L}_V$ eine Menge von Kanten-Header. Sei $\mathcal{R}$ eine Menge von Relationen und seien $R_V, R_E \in \mathcal{R}$ zwei feste weitere Relationen. Dann seien zwei injektive, partielle Abbildungen definiert, die einem Teil der Typen je eine Relation zuordnen:

$$\hat{T_V} : L_V \rightharpoonup \mathcal{R} \tag{4.1}$$

$$\hat{T_E} : L_E \rightharpoonup \mathcal{R} \tag{4.2}$$

Weiter seien zwei Abbildungen definiert, die jedem Typen eine Relation zuordnen:

$$T_V : L_V \to \mathcal{R} \tag{4.3}$$

$$T_E : L_E \to \mathcal{R} \cup \{\emptyset\} \tag{4.4}$$

mit

$$T_V : l \mapsto \begin{cases} \hat{T_V}(l) & l \in \operatorname{dom} \hat{T_V} \\ R_V & \text{sonst} \end{cases} \tag{4.5}$$

$$T_E : h \mapsto \begin{cases} \hat{T_E}(h) & h \in \operatorname{dom} \hat{T_E} \\ \emptyset & h \notin \operatorname{dom} \hat{T_E} \wedge \{l_s, l_t\} \cap dom\hat{T_V} \neq \emptyset \\ R_E & \text{sonst} \end{cases} \tag{4.6}$$

wobei $h = (l_e, l_s, l_t)$.

(4.7)

Nach Definition 3 gegebene Abbildungen T_V und T_E ordnen jedem Element-Typ, identifiziert durch *Label* (für Knoten) bzw. *Header* (für Kanten) genau eine Relation zu. Die Abbildungen $\hat{T_V}$ und $\hat{T_E}$ werden für Typen genutzt, die in separaten Tabellen gespeichert werden sollen. Diesen wird entsprechend eine eindeutige Relation zugeordnet, die nur Elemente diesen Typs repräsentiert. Jedoch muss nicht jedem Typ eine solche Relation zugewiesen sein. Typen für die diese Abbildungen nicht definiert ist, werden in die Relationen R_V und R_E abgebildet, welche den generischen Knoten- und Kanten-Tabellen entspricht.

Bemerkung 1. Nicht alle Kanten-Typen lassen sich abbilden. Denn in Gleichung 4.6 findet sich eine Ausnahme: Wenn $\hat{T_E}$ für einen bestimmten *Header* nicht definiert ist, $\hat{T_V}$ aber den entsprechenden Knoten-Labels der Quell- oder Ziel-Knoten eine Relation zuordnet, dann kann der durch diesen *Header* identifizierte Typ nicht abgebildet werden. Dies wird in diesem Fall durch eine leere Relation $\emptyset$ dargestellt. Diese Einschränkung ergibt sich daraus, dass im weiteren zwischen Kanten-Relationen und zugehörigen Knoten-Relationen Fremdschlüsselbeziehungen bestehen, nämlich zur Zuordnung von Quell- und Zielknoten zu jeder Kante, über deren ID (siehe Seite 17 zur Beschreibung der Integritätsbedingungen). Existiert keine separate Kanten-Relation für einen bestimmten Typ, dann wird die generische R_E verwendet. Diese kann aber nur R_V referenzieren, wobei in R_V genau nur die Elemente abgebildet sind, für die keine separate Knoten-Relation existiert. Für diese Kanten kann die Integritätsbedingung also nie erfüllt sein.

In Definition 3 wurde eine Abbildung eingeführt, die Typen von Graph-Elementen eine Relation zur Speicherung zuordnet. Im weiteren wird die Bezeichnung *logischer Typ* für eine Menge von Element-Typen verwendet, die auf eine Relation abgebildet werden. Die Abbildung ist damit von Typen von Graph-Elementen (nach Typ-Label) zu *logischen Typen* (Relationen) definiert.

Als nächstes wird definiert, wie diese Relationen aufgebaut sind, danach wie Elemente eines Graphen in diese abgebildet werden. Entsprechend des relationalen Modells von Codd [4, 5] sind Relationen als Mengen von Tupeln zu verstehen, wobei jede Komponente eines Tupels ein Element einer *Domain*, also einer Menge von Elementen eines bestimmten Datentyps, ist. Hier wird zur Anschaulichkeit direkt jeder Relation eine Bezeichnung zugeordnet. In einer Datenbank entspricht diese dem Name einer Tabelle. Im folgenden werden ID, $String$, PV und $Timestamp$ als Bezeichnung für verwendete Datentypen (*Domains* verwendet). Da sind ID eine Menge von IDs, *String*

eine Menge von Zeichenketten, PV eine Menge von Property-Werten und $Timestamp$ eine Menge von Zeitpunkten.

Relationen aus diesen Datentypen werden hier zur Abbildung von Graph-Elementen und deren Properties verwendet. So lässt eine Property analog zu den bereits vorgestellten Property-Tabellen als Tripel aus Element-ID, Property-Key und Property-Wert darstellen. Die zugehörige Relation ließe sich so repräsentieren:

$$R \subseteq R_1 \times R_2 \times R_3 = ID \times String \times PV = \{(i, p, v) : i \in ID, p \in String, v \in PV\} \tag{4.8}$$

Die Komponenten der Relation sind geordnet und lassen sich somit über ihren Index identifizieren. Hier wird jedoch die einfachere Art der Darstellung wie in [5] verwendet, in der Komponenten je eine Bezeichnung zugewiesen ist. Die gleiche Relation in Notation von Codd [5] kann also geschrieben werden als:

$$R(id : ID, key : String, value : PV) \tag{4.9}$$

Für das hybride Schema (Unterabschnitt 4.1.4) werden für jeden logischen Typ zwei Tabellen benötigt: eine Element-Tabelle, sowie eine Property-Tabelle. Die vorhandenen Spalten der Element-Tabelle hängen davon ab, welche Properties in dieser gespeichert sein sollen. Für jeden logischen Typ kann dabei eine andere Menge von Properties festgelegt sein.

Neben Properties müssen noch die folgenden weiteren Attribute der Elemente abgebildet werden: ID, IDs der Quell- und Zielknoten (nur für Kanten), Typ-Label (nur falls der Typ nicht bereits nur Elemente eines bestimmten Labels enthält), Start- und Endzeitpunkte für *valid time* und *transaction time* (falls vorhanden).

Definition 4 (Relationale Darstellung einer Element-Tabelle)**.** Die Element-Tabelle eines logischen Typen mit meiner Menge von Properties $P_f = \{property1, property2, \dots\}$ lässt sich als Relation R darstellen durch:

$$\begin{aligned}
&R(id : ID, && (4.10)\\
&\quad sourceId : ID, targetId : ID, && (4.11)\\
&\quad label : String, && (4.12)\\
&\quad property1 : PV, property2 : PV, \dots, && (4.13)\\
&\quad valFrom : Timestamp, valTo : Timestamp, && (4.14)\\
&\quad txFrom : Timestamp, txTo : Timestamp) && (4.15)
\end{aligned}$$

Dabei sind die Komponenten $sourceId$ und $targetId$ (4.11) nur für Kanten-Typen vorhanden, die Komponente *label* (4.12) nur falls der entsprechende logische Typ nicht bereits nur Elemente eines bestimmten Labels enthält. Komponenten für *valid time* (4.14) und *transaction time* (4.15) finden sich auch nur, falls diese abgebildet werden sollen.

Nach Definition 4 aufgebaute Relationen werden als Basis für Element-Tabellen genutzt. Es muss nun noch die zu jeder Element-Relation gehörende Property-Relation definiert werden:

Definition 5 (Relationale Darstellung einer Property-Tabelle)**.** Für eine Relation R, die einen logischen Typen repräsentiert, wird eine Relation R_p für Properties dargestellt als:

$$R_p(elementId : ID, key : String, value : PV) \tag{4.16}$$

Eine Zuordnung einer solchen Relation zu jeder Element-Relation wird gegeben durch:

Definition 6. Sei $\mathcal{R}$ eine Menge von Relationen. Dann sei eine injektive Abbildung definiert:

$$T_P : \mathcal{R} \to \mathcal{R} \tag{4.17}$$

Mit den in Definition 3 definierten Abbildungen T_V und T_E lassen sich dann Abbildungen definieren, die jedem logischen Typ eine Property-Relation zuordnen. Für Knoten-Typen ist eine Abbildung $T_{VP} : L_V \to \mathcal{R}$ definiert durch:

$$T_{VP} : l \mapsto T_P\left(T_V(l)\right) \tag{4.18}$$

Analog für Kanten $T_{EP} : L_E \to \mathcal{R} \cup \{\emptyset\}$ durch:

$$T_{EP} : h \mapsto \begin{cases} T_P\left(T_E(h)\right) & T_E(h) \neq \emptyset \\ \emptyset & \text{sonst} \end{cases} \tag{4.19}$$

Mit Definition 3 und Definition 6 lassen sich nun jedem logischen Typen eine Element-Relation und eine Property-Relation zuordnen. (Mit der in Bemerkung 1 beschriebenen Ausnahme.)

Nun lassen sich zu jedem Element Relationen (bzw. Tabellen) zuordnen. Für einen (temporalen) Property Graph $\mathbb{G}$ mit Knotenmenge V und Kantenmenge E sind jedem Knoten $v \in V$ und jeder Kante $e \in E$ Element-Relationen R_v und R_e zugeordnet mit

$$R_v = T_V(label(v)) \tag{4.20}$$
$$R_e = T_E\left((label(e), label(v_s), label(v_t))\right) \tag{4.21}$$

wobei $v_s, v_t \in V$ die Quell- und Zielknoten von e sind mit $id(v_s) = source(e)$ und $id(v_t) = target(e)$. Weiter sind je eine Property-Relation R_{vp} und R_{ep} zugeordnet mit

$$R_{vp} = T_{VP}(label(v)) \tag{4.22}$$
$$R_{ep} = T_{EP}\left((label(e), label(v_s), label(v_t))\right) \tag{4.23}$$

Wenn eine Kante nicht abgebildet werden kann, dann gilt $R_e = R_{ep} = \emptyset$.

4.2.1.2. Zuordnung von Attributen zu Spalten

Bisher wurde nur gezeigt, wie Abbildungen von Element-Typen auf Relationen definiert werden können und wie diese Relationen aufgebaut sind.

Hier wird zuerst vorausgesetzt, dass die Funktionen T_V und T_E je nur Relationen zuordnen, die eine für Elemente passende Struktur, wie in Definition 4, besitzen. Also Relationen der Form:

$$R(id, [sourceId, targetId], [label], [property1, \dots], [valFrom, valTo], [txFrom, txTo]) \tag{4.24}$$

Hier wurden Komponenten (bzw. Spalten), welche nicht für alle logischen Typen benötigt werden, in eckigen Klammern dargestellt. Die Komponenten $sourceId$ und $targetId$ finden sich nur für Kanten-Relationen aus T_E, *label* findet sich nur für die Relationen R_V und R_E, welche alle die Elemente enthalten, für deren Typ keine eigene Relation definiert wurde. Die Komponenten $property1, \dots$ repräsentieren Properties als Spalten. Namen und Anzahl dieser Komponenten hängen von der Definition ab. $valFrom, valTo, txFrom$ und $txTo$ repräsentieren die Start- und End-Zeitpunkte für *valid* und *transaction time*. Diese sind optional.

Weiter wird vorausgesetzt, dass die Funktion T_P je Relationen zuordnet, die eine für Properties passende Struktur, wir in Definition 5, besitzen. Also:

$$R_p(elementId : ID, key : String, value : PV) \tag{4.25}$$

Sei nun

$$\mathbb{G} = \langle V, E, \mathcal{I}, \mathcal{L}, P, A, label, id, source, target, \kappa' \rangle \tag{4.26}$$

ein Property Graph (siehe Definition auf Seite 4) oder ein temporaler Graph

$$\mathbb{G} = \langle V, E, \mathcal{I}, \mathcal{L}, P, A, label, id, source, target, \kappa', from_v, to_v, from_t, to_t \rangle \tag{4.27}$$

(nach Definition auf Seite 9). Dabei sei die Menge der IDs $\mathcal{I} = ID$, die Menge der Typ-Labels $\mathcal{L} = String$ und die Menge der Property-Namen (*keys*) $P = String$. Die Zeitdimensionen seien dabei je Mengen von Zeitpunkten $\Omega_v = \Omega_t = Timestamp$. Damit können nun Abbildungen von Elementen auf Relationen definiert werden. Es muss dabei jedes Element als Tupel in passender Struktur dargestellt werden. Im folgenden wird die partielle Abbildung $\kappa' : (V \cup E) \times P \rightharpoonup A$, welche Paaren aus Element und Property-Namen Werte zuordnet, zu einer „totalen“ Abbildung erweitert. Dazu sei $\texttt{null} \in PV$ ein spezieller Property-Wert, welcher verwendet wird, um nicht gesetzte Properties zu repräsentieren. Eine erweiterte Abbildung $\kappa : (V \cup E) \times P \to A$ ist dann definiert als

$$\langle e, p \rangle \mapsto \begin{cases} \kappa'(e, p) & \langle e, p \rangle \in \operatorname{dom} \kappa' \\ \texttt{null} & \text{sonst} \end{cases} \tag{4.28}$$

Wenn also für ein Element e eine Property mit *Key* p gesetzt ist, dann gibt diese Abbildung den Wert zurück. Andernfalls wird `null` als eine Art von Platzhalter verwendet.

Um zu ermitteln, welche Properties als Spalten in der Element-Relation R und welche als Elemente der Property-Relation R_p abgebildet werden, muss zuerst eine Funktion gegeben sein, die dies zuordnet. Hierfür wird zu jedem Typ eine Mengen von Properties zuordnet, welche dann in R gespeichert werden. Jeder dieser Property wird dann eine Spalte zugeordnet.

Definition 7 (Abbildung für Properties auf Spalten)**.** Seien wieder $\mathcal{L}_V \subseteq \mathcal{L}$ die Menge von Knoten-Labels und $\mathcal{L}_E \subseteq \mathcal{L} \times \mathcal{L}_V \times \mathcal{L}_V$ die Menge von Kanten-Header. Sei $\mathcal{C}$ eine total geordnete Mengen von Spaltennamen (Komponentennamen in Relationen). Sei P die Menge von Property-Namen. Dann seien Abbildungen

$$P'_V : \mathcal{L}_V \to \mathbb{P}(P) \tag{4.29}$$

und

$$P'_E : \mathcal{L}_E \to \mathbb{P}(P) \tag{4.30}$$

definiert, die jedem Typ (über Label bzw. Header) eine Menge Properties zuordnen. Weiter seien noch partielle Abbildungen, welche jedem Property-Namen eines Typs einen Spaltennamen zuordnet:

$$C_{\text{prop}} : (\mathcal{L}_V \cup \mathcal{L}_E) \times P \rightharpoonup \mathcal{C} \tag{4.31}$$

Damit lassen sich nun Elemente auf Element-Relationen zuordnen.

Definition 8 (Abbildung von Knoten auf Knoten-Relationen)**.** Sei $v \in V$ ein Knoten, $R_v \coloneqq T_V(label(v))$ die zugehörige Element-Relation für v. Seien $\{p_1, p_2, \ldots, p_n\} = P'_V(label(v))$ die *Keys* zu Properties, die für den Typ von v als Spalten der Element-Tabelle gespeichert werden sollen. Diese seien je auf Spaltennamen $\{prop\_1, prop\_2, \ldots, prop\_n\}$ abgebildet:

$$\{(label(v), p_1) \mapsto prop\_1, \ldots, (label(v), p_n) \mapsto prop\_n\} \subseteq \mathcal{C}_{\text{prop}} \tag{4.32}$$

Die Knoten-Relation R_v hat dann die Struktur:

$$R_v(id : ID, prop\_1 : PV, prop\_2 : PV, \ldots, prop\_n : PV) \tag{4.33}$$

Falls $R_v = R_V$ die generische Knoten-Relation ist, die nicht nur Knoten des Elementes $label(v)$ enthält, dann muss zusätzlich das Typ-Label Teil der Struktur sein:

$$R_v(id : ID, label : String, prop\_1 : PV, prop\_2 : PV, \ldots, prop\_n : PV) \tag{4.34}$$

Entsprechend sind für temporale Graphen noch die *Timestamp*-Spalten $valFrom$, $valTo$, $txFrom$, $txTo$ hinzuzufügen. Der Knoten v kann durch eine Abbildung $map_{R_v} : V \to R_v$ auf ein Tupel abgebildet werden:

$$v \mapsto \begin{cases} \langle id(v), \kappa(v, p_1), \ldots, \kappa(v, p_n) \rangle & R_v \neq R_V \\ \langle id(v), label(v), \kappa(v, p_1), \ldots, \kappa(v, p_n) \rangle & R_v = R_V \end{cases} \tag{4.35}$$

Definition 9 (Abbildung von Kanten auf Kanten-Relationen)**.** Die Abbildung für eine Kante $e \in E$ auf die zugehörige Kanten-Relation $R_e = T_E(\langle label(e), label(v_s), label(v_t) \rangle)$ (mit entsprechenden Quell- und Zielknoten v_s und v_t). Diese Abbildung kann dabei nur erfolgen, wenn R_e tatsächlich

existiert, e also abgebildet werden kann. Analog zur zuvor beschriebenen Abbildung für Knoten hat die Relation R_e die Struktur

$$R_e(id : ID, sourceId : ID, targetId : ID, prop\_1 : PV, \ldots, prop\_n : PV) \tag{4.36}$$

Für die generische Kanten-Relation R_E ist entsprechend wieder die $label : String$ Komponente zu ergänzen:

$$R_e(id : ID, sourceId : ID, targetId : ID, label : String, prop\_1 : PV, \ldots, prop\_n : PV) \tag{4.37}$$

Die Kante kann dann durch eine Abbildung $map_{R_e} : E \to R_e$ auf ein Tupel abgebildet werden:

$$e \mapsto \begin{cases} \langle id(e), \kappa(e, p_1), \ldots, \kappa(e, p_n) \rangle & R_e \neq R_E \\ \langle id(e), label(e), \kappa(e, p_1), \ldots, \kappa(e, p_n) \rangle & R_e = R_E \end{cases} \tag{4.38}$$

Damit sind nun zwei Abbildungen gegeben die Elemente auf ihre Element-Relation abbilden. Es verbleiben nun noch alle Properties abzubilden, die nicht in Spalten abgebildet werden.

Definition 10 (Abbildung von Properties auf Property-Relationen)**.** Sei $e \in V \cup E$ ein Element des Graphen und R_e die zugehörige Element-Relation mit $R_e \neq \emptyset$. Sei dann R_{ep} die durch $T_P(R_e)$ bestimmte Property-Relation. Diese hat die Struktur gemäß Definition 5:

$$R_{ep}(elementId : ID, key : String, value : PV) \tag{4.39}$$

Nun sei eine Abbildung definiert, welche das Element e auf eine Menge von Tupeln aus R_{ep} abbildet. Sei $\tilde{P} \subseteq P$ die Menge von Property-Namen, denen keine Komponenten in der Element-Relation zugeordnet sind, als $\tilde{P} \coloneqq P \setminus P'_V(label(e))$ für Knoten oder $\tilde{P} \coloneqq P \setminus P'_E(label(e), label(v_s), label(v_t))$ für Kanten.
Diese Abbildung $mapProperties_{R_{ep}} : V \cup E \to \mathbb{P}(R_{ep})$ ist gegeben durch:

$$e \mapsto \{\langle id(e), k, \kappa(e, k) \rangle : k \in \tilde{P} \wedge \kappa(e, k) \neq \texttt{null}\} \tag{4.40}$$

Diese Abbildung bildet also in eine Menge von Tripeln aus ID, Propery-Namen und Wert ab, wobei alle Properties berücksichtigt sind, die gesetzt sind (also deren Wert nicht `null` ist) und die nicht schon bereits in die Element-Tabelle abgebildet werden.

4.2.1.3. Beispiel

Um diese recht komplex wirkenden Abbildungen zu veranschaulichen, wird nun an einem Beispiel demonstriert, wie diese Definiert und genutzt werden können. Hierzu wird gezeigt, wie Element auf ein Schema wie in Abbildung 4.5 abgebildet werden.

In diesem Schema werden Knoten der Typen *Person* und *Place*, sowie Kanten der Typen *locatedIn*, je zwischen *Person* und *Place*, sowie *Place* und *Place*, als eigene logische Typen, damit auch separate Element-Relationen, abgebildet.

Art	Name	Komponenten
Knoten	$Person$	$id : ID, name : PV, dateOfBirth : PV$
Prop.	$Person\_p$	$elementId : ID, key : String, value : PV$
Knoten	$Place$	$id : ID, lon : PV, lat : PV$
Prop.	$Place\_p$	$elementId : ID, key : String, value : PV$
Kanten	$locatedIn\_Person\_Place$	$id : ID, sourceId : ID, targetId : ID, sinceYear : PV$
Prop.	$locatedIn\_Person\_Place\_p$	$elementId : ID, key : String, value : PV$
Kanten	$locatedIn\_Place\_Place$	$id : ID, sourceId : ID, targetId : ID$
Prop.	$locatedIn\_Place\_Place\_p$	$elementId : ID, key : String, value : PV$
Knoten	$Vertices$	$id : ID, label : String$
Prop.	$Vertices\_p$	$elementId : ID, key : String, value : PV$
Kanten	$Edges$	$id : ID, sourceId : ID, targetId : ID, label : String$
Prop.	$Edges\_p$	$elementId : ID, key : String, value : PV$

Tabelle 4.1.: Beispiel für Element- und Property-Relationen in einem Schema

In Tabelle 4.1 sind entsprechende Relationen dieses Schemas aufgelistet. Dabei sind zuerst alle logischen Typen gelistet, die nur Elemente eines bestimmten Labels enthalten, danach die generischen *Vertices* und *Edges* Typen für verbleibende Knoten und Kanten.

Hiermit lassen sich nun Abbildungen von Elementen auf dieses Schema definieren. Zuerst von Element-Typ auf logischen Typ:

$$\begin{aligned}\mathcal{R} \coloneqq \{&Person, Person\_p, Place, Place\_p, \\ &locatedIn\_Person\_Place, locatedIn\_Person\_Place\_p, \\ &locatedIn\_Place\_Place, locatedIn\_Place\_Place\_p, \\ &Vertices, Vertices\_p, Edges, Edges\_p\}\end{aligned} \tag{4.41}$$

$$R_V \coloneqq Vertices, R_E \coloneqq Edges \tag{4.42}$$

$$\hat{T_V} \coloneqq \left\{ \begin{array}{lcl} \text{‚Person‘} & \mapsto & Person \\ \text{‚Place‘} & \mapsto & Place \end{array} \right\} \tag{4.43}$$

$$\hat{T_E} \coloneqq \left\{ \begin{array}{lcl} \langle \text{‚locatedIn‘}, \text{‚Person‘}, \text{‚Place‘} \rangle & \mapsto & locatedIn\_Person\_Place \\ \langle \text{‚locatedIn‘}, \text{‚Place‘}, \text{‚Place‘} \rangle & \mapsto & locatedIn\_Place\_Place \end{array} \right\} \tag{4.44}$$

Jedem logischen Typ muss noch eine Property-Relation zugeordnet werden:

$$T_P \coloneqq \left\{ \begin{array}{lcl} Person & \mapsto & Person\_p \\ Place & \mapsto & Place\_p \\ locatedIn\_Person\_Place & \mapsto & locatedIn\_Person\_Place\_p \\ locatedIn\_Place\_Place & \mapsto & locatedIn\_Place\_Place\_p \\ Vertices & \mapsto & Vertices\_p \\ Edges & \mapsto & Edges\_p \end{array} \right\} \tag{4.45}$$

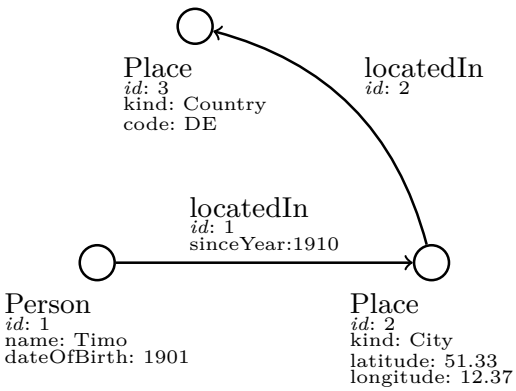

Abbildung 4.7.: Beispiel-Graph mit Person und Place Knoten-Typen

Als nächstes werden noch die Abbildungen für Properties angegeben:

$$P'_V(l) := \begin{cases} \{\text{,name‘, ,dateOfBirth‘}\} & l = \text{,Person‘} \\ \{\text{,longitude‘, ,latitude‘}\} & l = \text{,Place‘} \\ \emptyset & \text{sonst} \end{cases} \tag{4.46}$$

$$P'_E(h) := \begin{cases} \{\text{,sinceYear‘}\} & h = \langle \text{,locatedIn‘, ,Person‘, ,Place‘} \rangle \\ \emptyset & \text{sonst} \end{cases} \tag{4.47}$$

$$\mathcal{C}_{\text{prop}} := \left\{ \begin{array}{lcl} \langle \text{,Person‘, ,name‘} \rangle & \mapsto & name \\ \langle \text{,Person‘, ,dateOfBirth‘} \rangle & \mapsto & dateOfBirth \\ \langle \text{,Place‘, ,longitude‘} \rangle & \mapsto & lon \\ \langle \text{,Place‘, ,latitude‘} \rangle & \mapsto & lat \\ \langle \langle \text{,locatedIn‘, ,Person‘, ,Place‘} \rangle \text{, ,sinceYear‘} \rangle & \mapsto & sinceYear \end{array} \right\} \tag{4.48}$$

Anhand der zuvor definierten Funktionen und Relationen lassen sich nun Elemente auf Relationen überführen. Abbildung 4.7 zeigt dazu einen Beispiel-Graph, welcher aus den bereits eingeführten Typen besteht. Entsprechend der Abbildungen, ergeben sich beispielsweise für den *Place*-Knoten mit ID 2 (hier mit P_2 bezeichnet):

$$T_V(label(P_2)) = T_V(\text{,Place‘}) = \hat{T_V}(\text{,Place‘}) = Place \tag{4.49}$$

$$T_{VP}(label(P_2)) = T_P(Place) = Place\_P \tag{4.50}$$

Der Knoten wird also die Element-Relation *Place* und die Property-Relation *Place_P* abgebildet. Die entsprechenden Tupel ergeben sich durch:

$$\begin{aligned} map_{Place}(P_2) &= \langle id(P_2), \kappa(P_2, \text{,latitude‘}), \kappa(P_2, \text{,longitude‘}) \rangle \\ &= \langle 2, \texttt{12.37}, \texttt{51.33} \rangle \end{aligned} \tag{4.51}$$

$$mapProperties_{Place\_P}(P_2) = \{\langle id(P_2), \text{,kind‘}, \kappa(P_2, \text{,kind‘}) \rangle\} \tag{4.52}$$

$$= \{\langle 2, \text{,kind‘}, \texttt{City} \rangle\} \tag{4.53}$$

Die Properties *latitude* und *longitude* wurden in der Element-Relation gespeichert, da diese nach P'_V für dieses Label in der Element-Tabelle gespeichert werden sollen. Diese werden für alle Tupel sortiert, damit eine Zuordnung über Indizes möglich ist. Hier wird eine alphabetische Sortierung gewählt. Die verbleibende Property *kind* muss entsprechend in der Property-Relation abgebildet sein.

Der *Place*-Knoten mit ID 3 wird auf ähnliche Weise abgebildet. Da für diesen die Properties *latitude* und *longitude* nicht gesetzt sind, muss je der Platzhalter `null` verwendet werden.

$$\begin{aligned} map_{Place}(P_3) &= \langle id(P_3), \kappa(P_3, \text{,latitude`}), \kappa(P_3, \text{,longitude`}) \rangle \\ &= \langle 3, \texttt{null}, \texttt{null} \rangle \end{aligned} \tag{4.54}$$

Für die verbleibenden Properties *code* und *kind* werden je ein Tupel für die Property-Relation erzeugt:

$$mapProperties_{Place\_P}(P_3) = \{ \langle 3, \text{,code`}, \texttt{DE} \rangle , \tag{4.55}$$

$$\langle 3, \text{,kind`}, \texttt{Country} \rangle \} \tag{4.56}$$

Werden alle Elemente des Beispielgraphen abgebildet, dann ergeben sich folgende Relationen:

$$Person = \{\langle 1, \texttt{Timo}, \texttt{1901} \rangle\} \tag{4.57}$$

$$Person\_p = \emptyset \tag{4.58}$$

$$Place = \{\langle 2, \texttt{12.37}, \texttt{51.33} \rangle , \langle 3, \texttt{null}, \texttt{null} \rangle\} \tag{4.59}$$

$$Place\_p = \{\langle 3, \text{,code`}, \texttt{DE} \rangle , \langle 3, \text{,kind`}, \texttt{Country} \rangle\} \tag{4.60}$$

$$locatedIn\_Person\_Place = \{\langle 1, 2, 1, \texttt{1910} \rangle\} \tag{4.61}$$

$$locatedIn\_Person\_Place\_p = \emptyset \tag{4.62}$$

$$locatedIn\_Place\_Place = \{2, 2, 3\} \tag{4.63}$$

$$locatedIn\_Place\_Place = \emptyset \tag{4.64}$$

$$Vertices = Vertices\_p = Edges = Edges\_p = \emptyset \tag{4.65}$$

Eingangs wurde erwähnt, dass sich mit manchen Schemata nicht alle Kanten abbilden lassen (vgl. Bemerkung 1 auf Seite 24). Man betrachte dazu einen weiteren Beispielgraphen nach Abbildung 4.8. Der *Person* Knoten lässt sich, wie bereits gezeigt, auf die *Person* und *Person_p* Relationen abbilden. Für den Knoten mit Typ-Label *Food* existiert kein gesonderter Typ, dieser muss also in die generische *Vertices* Relation abgebildet werden. Es gilt nämlich ‚Food' $\notin \operatorname{dom} \hat{T_V}$. Der Knoten (hier F_6) wird also dargestellt als:

$$T_V(F_6) = Vertices \tag{4.66}$$

$$map_{Vertices}(F_6) = \langle id(F_6), label(F_6) \rangle = \langle 6, \text{,Food`} \rangle \tag{4.67}$$

$$T_{VP}(F_6) = T_P(Vertices) = Vertices\_P \tag{4.68}$$

$$mapProperties_{Vertices\_P}(F_6) = \{\langle 6, \text{,desc`}, \texttt{Mustard} \rangle , \langle 6, \text{,type`}, \texttt{Condiment} \rangle\} \tag{4.69}$$

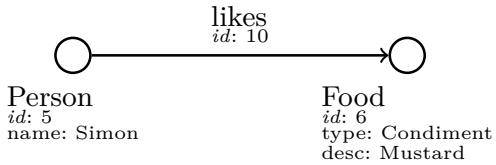

Abbildung 4.8.: Beispiel-Graph mit nicht abbildbaren Kanten-Typen

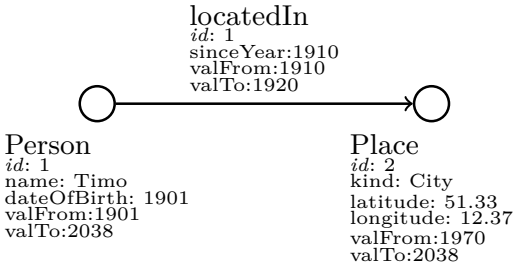

Abbildung 4.9.: Ausschnitt eines beispielhaften temporalen Graphen

Es bleibt die *likes*-Kante abzubilden. Diese besteht allerdings zwischen den logischen Typen *Person* und *Vertices*, wobei keine Relation existiert, die beide Relationen referenziert. Die Kante *müsste* in *Edges* abgebildet werden, diese Relation hat aber nur Fremdschlüsselbeziehungen zur Relation *Vertices*.

Dieser zweite Beispielgraph lässt sich damit nicht vollständig abbilden, was eine Limitierung des vorgestellten Schemas ist.

4.2.2. Abbildung von temporalen Graphen

Temporale Graphen lassen sich auf gleiche Weise wie „reguläre“ *Property Graphen* abbilden. Hier müssen lediglich noch temporale Attribute beachtet werden. Im temporalen Fall, werden noch allen Elementen zwei Zeitdimensionen zugewiesen. Die *valid time* kann manuell festgelegt sein, muss also auch abgebildet werden, die *transaction time* wird nur vom System verwaltet [22] und *kann nicht* manuell abgebildet werden. Man betrachte einen weiteren Beispielgraph in Abbildung 4.9. Dieser stellt einen Schnappschuss eines temporalen Graphen dar, wobei ein Ausschnitt eines vorherigen Beispielgraphen verwendet wurde, welcher nur noch um *valid time* ergänzt ist. *transaction time* wird hier nicht dargestellt, dies ist aus konzeptioneller Sicht nicht sinnvoll, da diese Zeitdimension nur repräsentiert, wann Elemente in der Datenbank gültig sind. Hier wird nur gezeigt, wie diese Elemente mit beiden Zeitdimensionen auf Tupel abgebildet werden. Weitere Details zur Verwaltung von *transaction time* finden sich in Kapitel zur Implementierung in dieser Arbeit.

Man betrachte den *Person* Knoten mit ID 1, diesem wurde die Zeitperiode $[1901, 2038)$ als *valid time* zugewiesen, was hier beispielsweise für den Lebenszeitraum der Person stehen könnte. Dabei ist 2038 hier nur ein Platzhalter für einen noch nicht gesetzten Wert. Der übliche Wert `null` kann hier nicht verwendet werden, da die Zeitpunkte vergleichbar und geordnet sein müssen. Die Wahl

des Wertes ist Detail der Implementierung und hier nicht relevant. Analog zu Definition 4 kann die *Person*-Relation mit temporalen Erweiterungen wie folgt strukturiert sein:

$$\begin{aligned} Person(&id : ID, name : PV, dateOfBirth : PV,\\ &validFrom : Timestamp, validTo : Timestamp,\\ &txFrom : Timestamp, txTo : Timestamp) \end{aligned} \tag{4.70}$$

Wie in der bisher definierten Funktion map_{R_V} zur Abbildung für Knoten, kann eine entsprechende temporale Version definiert werden. Dies wird nur am Beispiel des *Person* Knotens demonstriert. Sei P_1 also nun genau dieser Knoten. Die Abbildung des temporalen Elementes ergibt sich durch:

$$map^t_{Person}(P_1) = \langle id(P_1), \kappa(P_1, \text{,name`}), \kappa(P_1, \text{,dateOfBirth`}), \tag{4.71}$$

$$from_v(P_1), to_v(P_1), [null], [null]\rangle \tag{4.72}$$

$$= \langle 1, \texttt{Timo}, \texttt{1901}, 1901, 2038, [null], [null]\rangle \tag{4.73}$$

Die Abbildung funktioniert also auf gleiche Weise, jedoch wird anstelle den Zeitpunkten der *transaction time* ein spezieller Wert $[null]$ genutzt, welcher hier einem nicht gesetzten Wert entsprechen soll. In der Implementierung würde dieser Wert von der Datenbank ersetzt werden.

Die Abbildung für Properties erfolgt entsprechen auch durch:

$$mapProp^t_{Person\_p}(p) = \{\langle id(p), \text{,someKey`}, \texttt{someValue}, [null], [null]\rangle, \dots\} \tag{4.74}$$

4.2.3. Abbildungen von Relationen auf Graphen

Bisher wurde ausführlich gezeigt, wie Relationen und Abbildungen von Graphen auf Relationen definiert sein können. Dies wurde auch an einem Beispiel illustriert. Um nun die umgekehrte Richtung zu ermöglichen, werden wieder die gleichen Abbildungen wie zuvor benötigt. Über die Funktionen T_V und T_E lässt in umgekehrter Richtung jeder Element-Relation ein Typ-Label zuordnen. Jede Knoten-Relation ergibt damit eine Menge von Knoten, denen mit entsprechenden Property-Relationen weitere Properties zugewiesen sein können. Gleiches gilt für Kanten-Relationen. Der Graph lässt sich aus allen Relationen rekonstruieren.

Dies wird wieder nur an einem Beispiel gezeigt. Dazu seien alle Abbildungen wie zuvor auf Seite 30 gegeben.

So kann die Knoten-Relation *Person* beispielsweise folgendes Tupel enthalten:

$$\langle 20, \texttt{Max}, \texttt{1999}\rangle \in Person \tag{4.75}$$

Die zugehörige Property-Relation *Person_P*:

$$\langle 20, \text{,bornIn`}, \texttt{Berlin}\rangle \in Person\_P \tag{4.76}$$

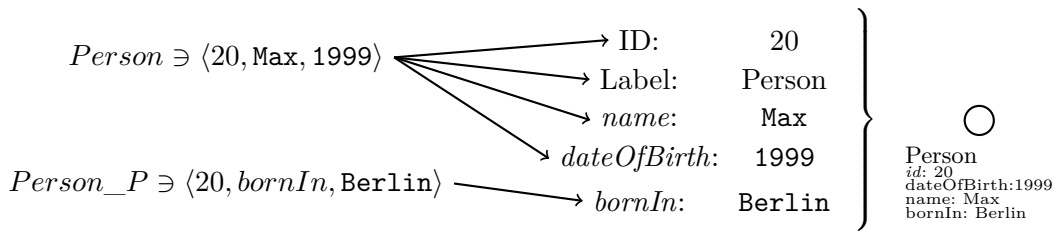

Abbildung 4.10.: Beispiel für Abbildung von Tupeln auf einen Knoten

Diese Tupel lassen sich in einen Knoten v_{20} übertragen, dabei wird zuerst Typ-Label bestimmt:

$$label(v_{20}) \coloneqq T_V^{-1}(Person) = \hat{T_V}^{-1}(Person) = \text{‚Person‘} \tag{4.77}$$

Die ID ist in der *id* Spalte der Element-Relation gespeichert, also

$$id(v_{20}) = \langle 20, \texttt{Max}, \texttt{1999} \rangle .id = 20 \tag{4.78}$$

Das Label ergibt sich daher, dass die Relation nur Knoten dieses Labels repräsentiert. Für die Properties wird zuerst ermittelt, welche in der Element-Relation abgebildet sind:

$$p_{v_{20}}^E = P_V'(label(v_{20})) = \{\text{‚dateOfBirth‘}, \text{‚name‘}\} \tag{4.79}$$

Entsprechende Werte ergeben sich dann auch aus den Spalten der Element-Relation:

$$\mathcal{C}_{prop}(\text{‚Person‘}, \text{‚dateOfBirth‘}) = dateOfBirth \tag{4.80}$$

Der Wert ist nun aus dem Tupel ablesbar mit

$$\langle 20, \texttt{Max}, \texttt{1999} \rangle .dateOfBirth = \texttt{1999} \tag{4.81}$$

Weitere Properties können aus der Property-Relation ermittelt werden, dafür werden alle Tupel mit passender ID gewählt und in Properties abgebildet. Für diesem Fall:

$$\kappa_{neu} = \kappa \cup \{\langle v_{20}, k, v \rangle : \langle i, k, v \rangle \in Person\_P \wedge i = id(v_{20})\} = \tag{4.82}$$

$$\kappa \cup \{\langle v_{20}, \text{‚bornIn‘}, \texttt{Berlin} \rangle\} \tag{4.83}$$

In Abbildung 4.10 wird die Abbildung der Tupel auf entsprechende Knoten-Attribute veranschaulicht.

4.2.4. Subgraph-Operator

Mit den bisher vorgestellten Konzepten lassen sich Graphen auf ein relationales Datenmodell abbilden. Hier werden nun noch Möglichkeiten vorgestellt, wie sich Operatoren auf diesem Modell umsetzten lassen. Dabei werden übliche Graph-Operatoren in Operationen auf Relationen übersetzt.

Der *Subgraph*-Operator bestimmten einen Teilgraphen eines Graphen. Es können zwei Prädikate definiert werden, welche je eine Teilmenge der Knoten- und Kantenmengen auswählen. Der resultierende Graph soll dabei konsistent sein. Wenn also durch das Kanten-Prädikat als Teil des Graphen ausgewählt wird, dann sollen auch die verbundenen Quell- und Zielknoten der Kante Teil des Graphen sein. Im folgenden wird für den Operator eine Notation ähnlich zu der von Junghanns et al. [12] verwendet. Als Parameter für den Operator dieses zwei Funktionen (Prädikate):

$$\varphi_V : V \to \{true, false\} \tag{4.84}$$

$$\varphi_E : E \to \{true, false\} \tag{4.85}$$

Dabei sind für die Funktionen folgende Möglichkeiten erlaubt:

Vergleich von Properties mit Konstanten Die Prädikate können so gewählt sein, dass diese bestimmte Properties der Elemente mit einer Konstante vergleichen. Dabei sind die üblichen Ordnungsrelationen erlaubt: $<$, $>$, $\leq$, $\geq$, $=$, daneben ist auch $\neq$ zulässig. So können beispielsweise alle Knoten ausgewählt werden, deren *age* Property einen Wert größer gleich 18 haben:

$$\varphi_V \coloneqq \begin{cases} true & v.age \geq 18 \\ false & \text{sonst} \end{cases} \tag{4.86}$$

Vergleich zweier Properties Neben den Vergleichsoperationen von Property-Werten mit Konstanten, können auch zwei Properties miteinander verglichen werden. Wieder werden die gleichen Vergleichsrelationen erlaubt. Beispielsweise können alle Knoten gewählt werden, deren *bornIn* und *residence* Properties nicht übereinstimmen.

$$\varphi_V \coloneqq \begin{cases} true & v.bornIn \neq v.residence \\ false & \text{sonst} \end{cases} \tag{4.87}$$

Logische verknüpfung mehrerer Prädikate Zwei Prädikate können mit den logischen *und* $\wedge$ und *oder* $\vee$ Operatoren zu einem neuen Prädikat verknüpft werden. Entsprechend werden dann alle Elemente ausgewählt, die von *beiden* (für *und*) oder *mindestens einem* (für *oder*) der Prädikate ausgewählt werden würden. Außerdem kann ein logisches *nicht* $\neg$ verwendet werden, um ein Prädikat zu invertieren. Dann werden genau alle Elemente gewählt, die vom ursprünglichen Prädikat nicht gewählt wurden.

Um einen Subgraph-Operator mit dem vorgestelltem relationalen Modell auszuführen, wird gezeigt, wie diese Arten von Prädikaten umgesetzt werden können. Allgemein werden im umgesetzten Subgraph-Operator bestimmten Teilmengen der Relationen so ausgewählt, dass wenn diese zurück in Elemente (bzw. einen Graphen) abgebildet werden, der Graph genau dem erwarteten Resultat des Operators entspricht.

Als Eingabe für den Operator dienen neben den zuvor genannten Möglichkeiten für Prädikate auch Mengen von Knoten-Labels und Kanten-Headern. Ausgabe sind dann alle Elemente die eines der Label tragen und für die das Prädikat φ_V den Wert *true* zurückgibt. Analog werden alle Kanten

ausgegeben, die eines der Header tragen, für die das Prädikat φ_E erfüllt ist und für die je Quell- und Zielknoten Teil des Resultats sind.

Dazu werden zuerst alle Element-Relationen ausgewählt die Elemente enthalten, deren Label (oder Header) in den Mengen der erwarteten Labels enthalten sind. Aus diesen Relationen werden dann genau die Tupel ausgewählt, deren zugehörige Elemente den Prädikaten entsprechen würden. Hierfür werden die Prädikate in Restriktionen nach Codd (da auch Theta-Select genannt) [5] übersetzt. Der Einfachheit halber werden logische Verknüpfungen direkt umgesetzt. Weiter wird der Vergleich zweier Spalten unterstützt.

Aus den gewählten Relationen werden dann je alle passenden Elemente ausgewählt. Im folgenden wird dazu gezeigt, wie jede Art von Prädikat übersetzt wird. Sei dafür l das Label eines zu verarbeitenden Typen und seien R_l die Element-Relation und R_{Pl} die Property-Relatioen für diesen Typen. Falls R_l dabei der generische Knoten- oder Kantentyp ist, dann müssen zuerst alle Elemente ausgewählt werden, die ein entsprechendes Label tragen. Seien also $l_1, \dots l_n$ die Typ-Label die ausgewählt werden sollen *und* im generischen Typ R_V oder R_E gespeichert sind. Dann muss zuerst eine Restriktion erfolgen mit:

$$R_V[label = l_1 \vee label = l_2 \vee \dots \vee label = l_n] \tag{4.88}$$

Im folgenden wird davon ausgegangen, dass dies bereits getan wurde. Um die Übersetzung in jedem Fall zu ermöglichen, wird die Element-Relation zuerst so zu einer neuen Relation erweitert, dass alle Properties, die im Prädikat referenziert werden, in dieser Relation abgebildet sind. Sei also

$$\varphi(e) = e.p_1 < c_1 \wedge e.p_2 > c_2 \vee \dots \tag{4.89}$$

ein beliebig komplexes Prädikat und seien $P = \{p_1, p_2, \dots\}$ alle Property-Namen, für die im Prädikat Vergleiche genutzt werden. Dann wird eine Relation erzeugt, die neben allen Spalten aus R_l auch Spalten für alle Properties aus P enthält. Zu jeder Property soll dann eine Spalte $c_{p_1}, c_{p_2}, \dots$ vorhanden sein.

Dazu wird für jede Property geprüft, ob diese bereits in der Element-Relation als Spalte abgebildet wird. Ist dies nicht der Fall, dann muss die zugehörige Property-Relation verwendet werden. Hierfür werden in R_{Pl} zuerst alle Tupel gewählt, die die Property p repräsentieren. Hierfür wird diese Restriktion verwendet:

$$R_{Pl}(elementId, key, value)[key = p] \tag{4.90}$$

Über einen Natural-Join ([5]) werden diese Tupel dann an die Element-Relation angefügt:

$$R'_l = R_l(id, \dots)[id \cdot elementId]R_{Pl}(elementId, key, value) \tag{4.91}$$

Anschließend wird noch die nicht benötigte *key* Spalte über eine Projektion entfernt:

$$R''_l = R'_l[\underbrace{id, \dots}_{R_l}, value] \tag{4.92}$$

Die Spalte wird dann noch entsprechend umbenannt und die resultierende Relation enthält dann alle Spalten der ursprünglichen Relation und alle Zeilen (Tupel), für die die Property p gesetzt ist (durch den Join $\cdot$ werden alle anderen Spalten entfernt). Außerdem enthält die Relation nun die Property p als weitere Spalte. Sollen Elemente *ohne* die Property p nicht entfernt werden, kann ein Outer Natural Join ($\odot$) an stelle des Natural Join verwendet werden. Statt Elemente ohne passende Properties zu entfernen, sind die Spalten dann mit `null`-Werten (ω bei Codd [5]) gefüllt. Ob dies gewünscht ist, hängt vom Anwendungsfall ab.

Dieses Verfahren wird für alle benötigten Properties $p_1, \ldots, p_n$ durchgeführt. Danach hat die Relation die Struktur:

$$R_l^* = (\underbrace{id, \ldots}_{R_l}, c_{p_1}, \ldots, c_{p_n}) \tag{4.93}$$

Das Prädikat φ lässt sich nun auf der erweiterten Relation „ausführen“. Dazu wird im Prädikat jedes Vorkommen einer Property p_i durch den zugehörigen Spaltennamen c_{p_i} ersetzt. Anschließend kann eine Restriktion auf der erweiterten Relation mit dem neuen Prädikat ausgeführt werden:

$$\tilde{R_l^*} = R_l^*(id, \ldots, c_{p_1}, \ldots, c_{p_n})[c_{p_1} < c_1 \wedge c_{p_2} > c_2 \vee \ldots] \tag{4.94}$$

Danach werden noch die verbleibenden Spalten über eine Projektion entfernt. Das Resultat ist eine Teilmenge der ursprünglichen Relation, wobei diese dann nur Tupel enthält, deren zugehörige Elemente dem Prädikat entsprechen. Diese können dann wie zuvor in Graph-Elemente abgebildet werden.

Um noch zu garantieren, dass die resultierenden Graphen konsistent sind, also Kanten nur enthalten sind, wenn beide verbundene Knoten auch im Resultat vorhanden sind, müssen Kanten-Relation noch weiter verarbeitet werden. Sei dazu

$$R_E^\varphi(id, sourceId, targetId, \ldots,) \tag{4.95}$$

eine Kanten-Relation, für die bereits eine Restriktion anhand des Kanten-Prädikats durchgeführt wurde. Seien dann R_s^φ und R_t^φ die Knoten-Relationen die je die Typen der Quell- und Ziel-Knoten des Kantentyps repräsentieren. Diese sollen bereits auch nach Knoten-Prädikat eingeschränkt sein. Dann kann über Joins garantiert werden, dass die referenzielle Integrität gewährleistet ist:

$$\tilde{R_E^\varphi} = (R_E^\varphi[sourceId \cdot id]R_s^\varphi)[targetId \cdot id]R_t^\varphi \tag{4.96}$$

Übrige Spalten können dann wieder über eine Projektion abgeschnitten werden. Im ersten Join über $sourceId$ und id wird garantiert, dass je der Quell-Knoten existiert, im zweiten über $targetId$ entsprechend je der Ziel-Knoten.

Dieses Verfahren wird für jeden logischen Typ separat durchgeführt. Danach lassen sich die entsprechenden Teil-Relationen in den gewünschten Teilgraphen abbilden.

4.2.5. Snapshot-Operator

Der Snapshot-Operator kann dazu genutzt werden, aus einem temporalen Graphen genau nur die Elemente auszuwählen, die zu einer bestimmten Zeit gültig sind. Der Operator bestimmt also den Zustand des Graphen zu einer bestimmten Zeit [20].

Temporale Graphen werden nach Unterabschnitt 4.2.2 wieder auf Relationen abgebildet, wobei die *transaction time* in zwei Spalten gespeichert ist, welche je den Start- und Endzeitpunkt festlegen. Diese sind vom System festgelegt und repräsentieren entsprechend des Konzepts temporaler Datenbanken [22, 13] ein Zeit-Interval, in dem das Element gültig ist.

Da die Abbildungen von Graphen auf Relationen so bestimmt sind, dass durch Integritätsbedingen immer sichergestellt ist, dass die Graph-Darstellung in der Datenbank auch einen konsistenten Graph-Zustand enthält, kann davon ausgegangen werden, dass die Datenbank zu jedem beliebigen Zeitpunkt in einen gültigen Graph-Zustand übersetzt werden kann. Diese Einschränkung wird ausgenutzt, um den Operator auf sehr einfache Weise umzusetzen:

Soll der Graph-Zustand zu einem bestimmtem Zeitpunkt t ermittelt werden, dann müssen nur alle Element- und Property-Relationen so eingeschränkt werden, dass diese nur Tupel enthalten, die zu diesem Zeitpunkt gültig sind. Werden diese dann auf einen Graphen abgebildet, dann entsteht garantiert ein gültiger Graph, der nur Elemente enthält, die zu diesem Zeitpunkt gültig sind. Für jede Relation R (also für jede Element- *und* Property-Relation) wird folgende Restriktion durchgeführt:

$$R[txFrom \leq t \wedge t < txTo] \tag{4.97}$$

Damit wird garantiert, dass der Zeitpunkt t im Gültigkeits-Intervall enthalten ist. Joins zwischen Knoten- und Kanten-Relationen, wie für den Subgraph-Operator, sind hier nicht nötig, da die Datenbank bereits zu jedem Zeitpunkt einen gültigen Graph enthält.

Der Snapshot-Operator lässt sich für *transaction time* also sehr leicht umsetzen. Für *valid time* sind keine entsprechenden Integritätsbedingungen vorhanden, trotzdem lässt sich der Operator leicht umsetzen. Dafür werden die zugehörigen Spalten $valFrom$ und $valTo$ wie Properties betrachtet. Ein Snapshot für *valid time* kann dann wie ein Subgraph-Operator mit dem Prädikat

$$\varphi(e) = e.valFrom \leq t \wedge t < e.valTo \tag{4.98}$$

ausgeführt werden.

4.2.6. Pattern Matching

Pattern Matching wird dazu genutzt, um alle Teilgraphen eines Graphen zu finden, die einem bestimmtem Muster entsprechen [10]. So sollen (Teil-)Graphen gefunden werden, in denen bestimmtem Knoten über bestimmte Kanten verbunden sind. Für alle Elemente können wieder Prädikate definiert werden.

Der umgesetzte Operator basiert auf einer Teilmenge der Anfragesprache *GDL* [10], mit der Einschränkung dass für jede Variable das Typ-Label fest sein muss. Das Vorgehen wird nun an einem Beispiel illustriert.

Man betrachte die Anfrage (*Query*) in Listing 4.1.

```
1  MATCH (p:Person)-[l:locatedIn]->(loc:Place)
2     (q:Person)-[m:locatedIn]->(loc)
3  WHERE p.name = 'Max M.' AND
4     p != q
```

Listing 4.1: Beispiel einer GDL-Anfrage

Entsprechend dieser Anfrage sollen alle Teil-Graphen mit 2 *Person*-Knoten und 1 *Place*-Knoten gefunden werden, wobei die beide *Person* über *locatedIn*-Kanten mit dem *Place*-Knoten verbunden sind (Zeile 1 und 2). Für den ersten *Person*-Knoten soll eine Property *name* auf den Wert `Max M.` gesetzt sein (Zeile 3). Außerdem sollen die beiden *Person*-Knoten nicht den gleichen Knoten darstellen (Zeile 4).

In der Anfrage ist jedem Element eine Variable zugeordnet. Zur Beantwortung der Anfrage wird jedem dieser Variablen ein Element des Graphen zugeordnet. Das allgemeine Vorgehen hier orientiert sich an der Implementierung in Gradoop von Junghanns [10], jedoch wird das Verfahren hier für das relationale Modell umgesetzt. Zuerst wird dafür für jede Variable ermittelt, aus welchem logischen Typ Elemente stammen müssen. Wie erwähnt wird in dieser Arbeit vorausgesetzt, dass jeder Variable schon ein Typ-Label zugewiesen wurde. Für Kanten ist daher auch implizit klar, welcher *Header* zugewiesen ist. Mit Hilfe der Abbildungen des Schemas (Unterabschnitt 4.2.1) kann somit auch jeder Variable eine Element-Relation zugeordnet werden.

Für dieses Beispiel ist die Zuweisung dann:

$$type := \left\{ \begin{array}{lcl} p & \mapsto & Person \\ l & \mapsto & locatedIn\_Person\_Place \\ loc & \mapsto & Place \\ q & \mapsto & Person \\ m & \mapsto & locatedIn\_Person\_Place \end{array} \right\} \tag{4.99}$$

Um die zusätzlichen Einschränkungen (Prädikate) in den Zeilen 3 und 4 berücksichtigen zu können, wird noch ermittelt, welche Spalten zur Beantwortung der Anfrage nötig sind. Allgemein ist für jede Relation *id*, *sourceId*, *targetId* und *label* (falls vorhanden) nötig. Zusätzlich ist für diese Anfrage noch die *name* Property für p relevant, anhand der Kandidaten für p ausgewählt werden sollen.

Ermittelt man also für jede Variable noch alle relevanten Spalten (bzw. Properties), dann ergibt sich folgende Zuweisung:

$$columns \coloneqq \left\{ \begin{array}{lcl} p & \mapsto & \{id, name\} \\ l & \mapsto & \{id, sourceId, targetId\} \\ loc & \mapsto & \{id\} \\ q & \mapsto & \{id\} \\ m & \mapsto & \{id, sourceId, targetId\} \end{array} \right\} \tag{4.100}$$

Falls eine Property *nicht* in der Element-Relation abgebildet ist, dann muss die entsprechende Relation um diesen Wert erweitert werden. In Unterabschnitt 4.2.4 wurde ein solches Verfahren gezeigt. Dabei wird für jede Property ein Join mit der Property-Relation durchgeführt, um entsprechende Werte hinzuzufügen. Der Einfachheit halber wird hier davon ausgegangen, dass dies bereits geschehen ist.

Die Anfrage kann nun in eine für das relationale Modell übersetzt werden. Für jede Variable v werden zuerst aus der entsprechenden Relation $type(v)$ alle Kandidaten ausgewählt. In diesem Fall ist nur $type(p)$ nach *name* Property eingeschränkt. Für diese Relation wird also eine Restriktion ausgeführt:

$$type(p)[name = \texttt{MaxM.}] = Person[name = \texttt{MaxM.}] \tag{4.101}$$

Anschließend wird für jede Relation garantiert, dass zugehörige Elemente entsprechend der Anfrage verbunden sind. So sollen in diesem Beispiel Elemente so ausgewählt sein, dass ein Kandidat für p der Quell-Knoten eines Kandidaten für l ist. Weiter muss loc dem Ziel-Knoten entsprechen. Diese Einschränkungen werden für alle Relationen durch Joins garantiert. p ist nämlich genau dann der Quell-Knoten von l, wenn die id-Spalte von p gerade der $sourceId$ Spalte von l entspricht, usw. Dazu wird zuerst das Kreuzprodukt ($\times$, [5]) für alle Relationen gebildet, danach werden alle Bedingungen für die Anfrage über Restriktionen garantiert:

$$R_1 = (type(p)[name = \texttt{MaxM.}]) \times type(l) \times type(loc) \times type(q) \times type(m) \tag{4.102}$$

$$R_2 = R_1[id_p = sourceId_l \wedge targetId_l = id_{loc} \wedge id_q = sourceId_m \wedge targetid_m = id_{loc}] \tag{4.103}$$

$$R_3 = R_2[id_p \neq id_q] \tag{4.104}$$

$$R_r = R_3[id_p, id_l, id_{loc}, id_q, id_m] \tag{4.105}$$

In (4.102) wird das Produkt aller Relationen gebildet. Die resultierende Relation enthält dann je die IDs aller Elemente und alle benötigten weiteren Spalten. In (4.103) wird dann garantiert, dass die Elemente entsprechend der Anfrage verbunden sind. Mit (4.104) werden dann noch die weiteren Bedingungen der Anfrage geprüft. In diesem Fallen sollen p und q nicht das gleiche Element darstellen, also werden nur die Elemente gewählt, wo entsprechende ID-Spalten ungleich sind. Mit dem finalen Schritt (4.105). Werden nun noch alle nicht benötigten Spalten entfernt. Jedes Tupel der Relation R_r enthält dann die IDs der Elemente, auf die die Variablen abgebildet sein können. Also entspricht jedes solche Tupel einem Teilgraph. Für dieses Beispiel würde ein Tupel $\langle 1, 2, 3, 4, 5 \rangle$ einer Abbildung von Variable auf Element-ID entsprechen mit: $p \mapsto 1, l \mapsto 2, \ldots$. Ein passender

Teilgraph lässt sich daraus bestimmen, dass für jede Variable nun ID und Typ-Label bekannt sind. Aus den Element-Relation können dann die Elemente ausgewählt werden, die genau diese IDs tragen. Der resultierende Graph enthält die Knoten $\{v_p, v_{loc}, v_q\}$ und die Kanten $\{e_l, e_m\}$. Mit $id(v_p) = 1, id(e_2) = 2, \ldots$.

4.3. Zusammenfassung

Es wurde ein flexibles Schema dargestellt, mit dem *Property Graphen* auf ein relationales Modell abgebildet werden können. Dazu wurde eine Reihe von Abbildungen eingeführt, die zuerst Element-Typen auf logische Typen abbilden. Element-Typen werden dabei über Typ-Labels identifiert. Für Kanten wird nicht nur das eigene Typ-Label, sondern auch die der verbundenen Knoten-Typen berücksichtigt. Die Typen von Kanten werden damit über so genannte *Header* bestimmt, welche Tripel der entsprechenden Knoten-, Quell-Knoten- und Ziel-Knoten-Labels darstellen.

Jeder Typ wird auf eine Element-Relation und eine Property-Relation abgebildet. Dabei können Typen festgelegt werden, die in eigenen Relationen gespeichert sein sollen. Alle weiteren Typen sind auf generische Knoten- und Kanten-Relationen abgebildet. Die zu einem Typ passenden Relationen werden als logische Typen bezeichnet. Für jeden solchen Typen können eine Menge von Property-Namen festgelegt werden, welche dann als Spalten der Element-Relationen abgebildet sind. Alle weiteren Properties werden die eine passende Property-Relation eingefügt.

Temporale Graphen können abgebildet werden, indem zu den Relationen je Zeitspalten hinzugefügt werden, die die Start- und Endzeitpunkte der *valid* und *transaction time* repräsentieren.

Es wurden einige Operatoren für dieses Modell vorgestellt. Ein Subgraph-Operator zur Bestimmung eines Teilgraphen kann umgesetzt werden, indem Prädikate für Graph-Elemente in Restriktionen auf Relationen (nach Codd [5]) übersetzt werden. Der Snapshot-Operator ergibt sich fast automatisch, da durch Integritätsbedingungen garantiert ist, dass die Datenbank zu jedem Zeitpunkt einen gültigen Graphen repräsentiert. Ein Verfahren zu Pattern Matching folgt den gleichen Konzepten, zuerst werden Kandidaten für Query-Variablen aus Element-Relationen über Restriktionen ausgewählt. Die gewünschte Graph-Struktur wird dann noch über Join (in dem Fall durch äquivalente Produkte und Restriktionen) garantiert.

Für alle Operatoren werden dabei gegebenenfalls zuerst alle betrachteten Element-Relationen mit Properties aus der Property-Relation erweitert.

Graph-Resultate sind dabei immer wieder über (Teilmengen von) Relationen dargestellt.

5. Implementierung

In diesem Kapitel wird beschrieben, wie die in Kapitel 4 vorgestellten Konzepte umgesetzt werden. Die Implementierung erfolgt in Java 8. MariaDB (in Version 10.5)[3] wird als relationale Datenbank eingesetzt, entsprechend wird SQL als Anfragesprache genutzt. Die Bibliothek SQLBuilder [15] (Version 3.0.1) wird genutzt, um die Verarbeitung von SQL Anfragen zu erleicherten. Die Kommunikation mit der Datenbank erfolgt über eine JDBC Schnittstelle [16].

5.1. Grundlagen

5.1.1. Graph-Modell

Property Graphen werden als `Graph` Interface repräsentiert. Dieses bietet Methoden an, um die Knoten- und Kanten-Mengen des Graphen zu erhalten und um Elemente hinzufügen. Knoten und Kanten werden je durch `Vertex` und `Edge` Interfaces dargestellt, welche von einem gemeinsamen `Element` Interface erben. Für Properties wird je `String` als Name (*Key*) und Wert verwendet. Properties anderen Typs (zum Beispiel Ganzzahlen/`Integer`) müssen entsprechend konvertiert werden. Als ID-Typ kann für alle Elemente eine beliebige Java-Klasse verwendet werden, wobei nur sortierbare Typen unterstützt sind. Diese müssen ebenso von der JDBC Schnittstelle unterstützt werden, also direkt in Typen der Datenbank umgewandelt werden können. In Tabelle 5.1 sind ID Typen (je in Java und SQL) aufgelistet, die getested wurden. In Listing 5.1 werden die Interface für Elemente vereinfacht aufgelistet. (Aus Platzgründen wurden je passende `set` Methoden entfernt.)

```
1  interface Properties<K, V> {
2      V get(K key);
3      void set(K key, V value);
4      Set<K> getKeys();
5      void remove(K key);
6  }
7  interface Element<I extends Comparable<I>> {
8      I getID();
9      String getLabel();
10     Properties<String, String> getProperties();
11     LocalDateTime getTxFrom();
12     LocalDateTime getTxTo();
13     LocalDateTime getValFrom();
14     LocalDateTime getValTo();
```

[3] `https://mariadb.com/kb/en/changes-improvements-in-mariadb-105/`

Java-Typ	SQL-Typ	Beschreibung
`Integer`	`INTEGER`	32-bit Ganzzahlen
`Long`	`BIGINT`	64-bit Ganzzahlen
`String`	`VARCHAR`	Zeichenketten

Tabelle 5.1.: Unterstützte IDs im Prototyp, je mit entsprechendem SQL Typ

```
15  }
16  interface Vertex<I extends Comparable<I>> extends Element<I> {}
17  class EdgeHeader {
18      String getLabel();
19      String getSourceLabel();
20      String getTargetLabel();
21  }
22  interface Edge<I extends Comparable<I>> extends Element<I> {
23      I getSourceId();
24      I getTargetId();
25      EdgeHeader getHeader();
26  }
```

Listing 5.1: Interfaces für Graph-Elemente (vereinfachter Ausschnitt)

Elemente stellen also einfache Objekte dar, alle Attribute, wie Label, ID, temporale Attribute und Properties werden über get- und set-Methoden zugänglich gemacht. `Graph`en nutzen dabei einfache `List`-Objekte von Elementen.

5.1.2. Graph-Schema

Das Graph-Schema ordnet entsprechend Kapitel 4 Elementen je einen logischen Typ zu und hält Informationen über diese Typen. Das Schema ist dabei in zwei Teile geteilt:

5.1.2.1. Graph-Schema aus Sicht des Graphen

Eine Sicht als Graph-Schema, wobei eine Mengen von logischen Typen, je für Knoten und Kanten, und eine Abbildung von Label oder Header auf Typ repräsentiert wird. Für jeden logischen Typen werden Informationen darüber gehalten, welche Properties in der Element-Tabelle gespeichert sind und welche in die Property-Tabelle ausgelagert werden sollen. Weiter wird zu jedem Attribut (Label, ID, usw.) eine Referenz zur passenden Spalte verwaltet.

Als Referenz wird dabei je nur Spalten- und Tabellen-Name gespeichert. In Listing 5.2 werden die Interfaces gezeigt, die Referenzen auf Tabellen und Spalten kodieren. Ein logischer Typ verwaltet eine Menge solcher Referenzen, je für jede Attribut. Dazu sind die Interfaces `VertexSchema` und `EdgeSchema` definiert, welche je einen logischen Knoten- und Kantentyp beschreiben. Diese erben vom gemeinsamen Typ `ElementSchema`. In Listing 5.3 wird letzteres aufgelistet. Mit *fixed property* sind dabei all die Properties gemeint, die als Spalte der Element-Tabelle abgebildet sind. *Dynamic properties* sind entsprechend die verbleibenden Properties, welche in der zugehörigen Property-Tabelle abgelegt sind. Es existieren dazu Funktionen, mit denen geprüft werden kann, welche Properties *fixed* sind (`getFixedPropertyKeys`) und in welcher Spalte diese abgelegt sind (`getColumnForFixedProperty`). Typen haben dabei entweder ein festes Label (*fixed label*), falls sie logische Typen darstellen, die nur Elemente eines bestimmten Typ-Labels abbilden *oder* eine Spalte, welche das Label speichert. Es darf nur eines der Fall sein, die je andere Funktion gibt `null` zurück.

```
interface TableReference {
    String getTableName();
}
interface ColumnReference {
    String getColumnName();
    TableReference getTable();
}
```

Listing 5.2: Interfaces für Referenzen auf Tabellen und Spalten

```
interface ElementSchema {
    ColumnReference getIdentifierColumn();
    boolean isFixedLabel();
    String getLabel();
    ColumnReference getLabelColumn();
    TemporalAttribute[] getTemporalAttributes();
    String[] getFixedPropertyKeys();
    ColumnReference getColumnForFixedProperty(String key);
    boolean hasDynamicProperties();
    ColumnReference getDynamicPropertiesForeignKey();
    ColumnReference getDynamicPropertyKeyColumn();
    ColumnReference getDynamicPropertyValueColumn();
}
```

Listing 5.3: Interface, welche logische Typen repräsentieren

Das `GraphSchema` Interface verwaltet eine Menge solcher logischer Typen und bietet Funktionen an, mit denen der zugehörige Typ zu einem Label gefunden werden kann.

```
interface GraphSchema {
    Collection<VertexSchema> getVertexTypes();
    Collection<EdgeSchema> getEdgeTypes();
    Optional<VertexSchema> getDefaultVertexType();
    Optional<EdgeSchema> getDefaultEdgeType();
    VertexSchema getVertexTypeByLabel(String label)
        throws NoSuchSchemaElementException;
    EdgeSchema getEdgeTypeByLabel(EdgeHeader label)
        throws NoSuchSchemaElementException;
}
```

Listing 5.4: GraphSchema Interface, zur Zuordnung logischer Typen zu Graph-Element-Typen

Es wird die Möglichkeit geboten, alle vorhandenen Typen anzufragen oder den zu einem Label passenden Typ zu finden. Die `getDefaultXType` Methoden geben dabei je die generischen Typen zurück, welche Elemente aufnehmen, für die kein eigener Typ vorhanden ist. Die `getXTypeByLabel` Funktionen geben entweder einen Typ mit festem Label oder den generischen Typ (in der Implementierung auch *default* Typ) zurück. Das Schema is dabei so flexibel, dass es möglich ist, die beiden *default* Typen nicht zu definieren. Als Sonderfall ist es sogar möglich Graph-Schemata zu definieren, welche *keine* Typen speichern.

Art	Beschreibung
`Key`	Primärschlüssel der Tabelle, genutzt für Element IDs
`KeyValue`	Spalte, welche in Property-Tabellen je Namen und Wert speichert
`Label`	Label eines Elementes (nur in generischen Element-Tabellen)
`Property`	Wert einer Property (mit festem Name)
`Reference`	Fremdschlüssel, genutzt für IDs der Quell-/Zielknoten und in Property-Tabelle
`Time`	Zeitstempel, für temporale Attribute
`EdgeHeader`	Label der Quell-/Zielknoten einer Kante (nur intern genutzt)

Tabelle 5.2.: Spalten-Arten in der relationalen Schema-Sicht

Im Allgemeinen können diese Klassen dazu verwenden werden, um zu Graph-Typen gehörende logische Typen zu finden und um die Struktur der entsprechenden Tabellen zu identifizieren. Dies wird in der Implementierung allgemein als Basis zur Generierung von Anfragen genutzt, dazu später mehr.

Die zweite Art der Repräsentation stellt die entsprechende Gegenrichtung dar:

5.1.2.2. Graph-Schema aus relationaler Sicht

Diese Sicht verwaltet das Schema als Menge von Tabellen und deren Aufbau. Zu jeder Tabelle sind folgende Informationen gespeichert:

Art der Tabelle Die Art der Tabelle bestimmt, wofür diese genutzt wird. Dabei ist zwischen Knoten-, Kanten- und Property-Tabellen unterschieden.

Name Der Name der Tabelle (bzw. Relation) in der Datenbank.

Typ-Label der Elemente Wenn es sich um eine Element-Tabelle handelt, dann *kann* gespeichert sein, welche Typen von Elementen (nach Typ-Label) gespeichert sind. Dies ist nur der Fall, wenn die Tabelle nicht die generischen (*default*) Typen repräsentiert.

Spalten Eine geordnete Liste von Spalten in der Tabelle. Dabei wird nicht nur der Name, sondern auch die *Art* der Spalte gespeichert. Damit wird festgelegt, welches Attribut in der Spalte gespeichert ist. Für jede Art ist eine Klasse definiert, die passende Metadaten abbildet. In Tabelle 5.2 sind implementierte Arten aufgelistet. Beispielsweise wird zu einer `Property` Spalte noch der Name der Property, zu `Time` die Zeitdimension und zu `Reference` die referenzierte Spalte gespeichert.

Diese Art der Darstellung wird nicht nur genutzt, um Metadaten über verwaltete Tabellen abzuspeichern, sondern auch, um die Struktur von Relationen zu speichern, die als Zwischenergebnisse für Operatoren erzeugt werden, abzubilden. Allgemein findet diese Sicht in der Abbildung von Relationen auf Elemente Verwendung. Anhand von Metadaten zu Spalten kann entschieden werden, welche Komponenten einer Relation auf welches Attribut eines Elementes abgebildet werden.

5.1.3. Definition von Graph-Schemata

Die bisher vorgestellten Klassen repräsentieren die Abbildungen von Graph-Schema auf relationales Schema. Diese sind über Interfaces definiert. Konkrete Implementierungen sind im Prototyp auch gegeben, zu diesen sind noch Hilfsklassen definiert, mit denen solche Abbildungen leicht erzeugt werden können. Dazu werden texttttBuilder-Klassen vorgestellt, wobei nur die beispielhafte Verwendung gezeigt wird, da diese Klassen sehr komplex sind. Sie garantieren unter anderem, dass das Schema konsistent ist, also dass Elemente immer korrekt abgebildet werden können, keine Typen doppelt definiert sind oder dass die Abbildung eindeutig ist. Die relationale Sicht des Schemas kann über die `TableBuilder` Klasse definiert werden. Diese wird in Listing 5.5 demonstriert.

```
1  Table t = new TableBuilder("Person_v", VERTICES)
2    .setLabel("Person")
3    .addKeyColumn("id")
4    .addPropertyColumn("name").addPropertyColumn("dateOfBirth")
5    .build();
```

Listing 5.5: Verwendung der TableBuilder Klasse (Beispiel)

Die Definition aus Graph-Sicht mittels `SchemaBuilder` wird anhand des Beispiels aus Unterabschnitt 4.1.4 gezeigt.

```
1  GraphSchema schema = new SchemaBuilder()
2    .withSystemVersioning()
3    .withApplicationTime()
4    .addVertexType()
5      .withLabel("Person")
6      .withProperty("name").withProperty("dateOfBirth")
7      .withPropertyTable()
8    .andThen()
9    .addVertexType()
10     .withLabel("Place")
11     .withProperty("longitude", "lon").withProperty("latitude", "lat")
12     .withPropertyTable()
13     .andThen()
14   .addVertexType().withLabelColumn().withPropertyTable().andThen()
15   .addEdgeType()
16     .withLabel("locatedIn", "Person", "Place")
17     .withProperty("sinceYear")
18     .withPropertyTable()
19     .andThen()
20   .addEdgeType()
21     .withLabel("locatedIn", "Place", "Place")
22     .withPropertyTable()
23     .andThen()
24   .addEdgeType().withPropertyTable().withLabelColumn()
25   .andThen().build();
```

Listing 5.6: Verwendung der SchemaBuilder Klasse (Beispiel)

Hier werden in Listing 5.6 zuerst Knoten-Typen mit festem Label definiert, welchen Properties als Spalten zugeordnet werden. Kanten-Typen werden danach definiert. Bei Nutzung dieser Klasse werden auch je Schemata aus Tabellen-Sicht definiert und validiert.

5.1.4. Schema-Serialisierung als XML

Das Graph-Schema lässt sich neben der Definition über die API auch als XML [23] definieren. Definierte Schemata können auch als XML exportiert werden. Dadurch wird ermöglicht, dass Metadaten für den Prototyp abgespeichert und wiederverwendet werden können. Zur Veranschaulichung wird hier nur gezeigt, wie die XML Darstellung des zuvor mit `SchemaBuilder` definierten Schemas aussieht. In Listing 5.7 ist ein Ausschnitt des generierten XML Dokumentes gezeigt.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE graph-schema SYSTEM "graph-schema-v01.dtd">
<graph-schema>
  <vertices table="null_v">
    <id>_id</id>
    <label>_label</label>
    <properties/>
    <timePeriod name="VALID_TIME" sysVersioned="false">
      <start>valStart</start>
      <end>valEnd</end>
    </timePeriod>
    <timePeriod name="SYSTEM_TIME" sysVersioned="true">
      <start>txStart</start>
    <end>txEnd</end>
    </timePeriod>
  </vertices>
  <edges label="locatedIn" sourceLabel="Person"
   table="locatedIn_Person_Place_e" targetLabel="Place">
    <id>_id</id>
    <sourceId>_sourceId</sourceId>
    <targetId>_targetId</targetId>
    <properties>
    <property key="sinceYear">sinceYear</property>
    </properties>
  </edges>
</graph-schema>
```

Listing 5.7: XML Darstellung des zuvor definierten Schemas (Ausschnitt)

5.2. Erzeugen des Tabellen-Schemas

Mit den bisher gezeigten Klassen ist es möglich das flexible Graph-Schema zu definieren. Hier wird nun noch kurz gezeigt, wie Tabellen tatsächlich erzeugt werden, dazu wird zu einer Tabelle

aus einem vorherigen Beispiel gezeigt, wie diese in SQL definert werden kann. Für den mittels Listing 5.6 definerten Typ *Person*, kann eine temporale Tabelle[4] so definiert sein:

```
1  CREATE TABLE locatedIn_Person_Place_e (
2    _id INTEGER,
3    _sourceId INTEGER,
4    _targetID INTEGER,
5    sinceYear VARCHAR(1000),
6    valStart TIMESTAMP(6),
7    valEnd TIMESTAMP(6),
8    txStart TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
9    txEnd TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
10   CONSTRAINT locatedIn_Person_Place_e PRIMARY KEY (_id),
11   CONSTRAINT fk_Person_source_idref FOREIGN KEY (_sourceId)
12     REFERENCES Person_v (_id) ON DELETE CASCADE ON UPDATE RESTRICT,
13   CONSTRAINT fk_Place_target_idref FOREIGN KEY (_targetId)
14     REFERENCES Place_v (_id) ON DELETE CASCADE ON UPDATE RESTRICT,
15   PERIOD FOR VALID_TIME(valStart, valEnd),
16   PERIOD FOR SYSTEM_TIME(txStart, txEnd)
17 ) WITH SYSTEM VERSIONING;
```

Listing 5.8: SQL Definition einer Tabelle

Dabei für die *locatedIn_Person_Place_e* Kantenrelation werden zuerst die Spalten für ID und IDs der Quell- und Zielknoten erzeugt, danach die Spalten für Properties, dann die für *valid time* und *transaction time*. Schließlich werden noch Contraints[5] definiert, die garantieren, dass zugehörige Knoten auch in der Datenbank existieren. Wird passender Quell- oder Zielknoten gelöscht, dann wird wegen der `ON DELETE CASCADE` für die Fremdschlüsselbeziehung zur Knotentabelle auch die zugehörige Kante gelöscht. Außerdem werden die Abbildungen der Zeitdimensionen für *valid time* und *transaction time* definiert. Am Schluss wird noch die System-Versionierung aktiviert, damit die Änderungen an der Tabelle aufgezeichnet und gespeichert werden. Dadurch können später vergangene Zustände der Tabelle angefragt werden.

Ähnliche Definitionen werden für alle Tabellen ausgeführt, zuerst für Knoten-Tabellen, dann für Kanten-Tabellen und schließlich für Property-Tabellen.

5.3. Abbildung von Elementen auf Relationen

Um Elemente auf Relationen abzubilden, wird zuerst über die `GraphSchema` Klasse der logische Typ ermittelt. Falls dieser existiert, wird dann mittels zugehöriger `ElementSchema` Klasse bestimmt, welche Properties als Spalten abgelegt werden. Außerdem werden die Spaltennamen der verbleibenden Attribute bestimmt. Anschließend kann eine `INSERT`[6] Anfrage erzeugt werden, die das Element in die Datenbank einfügt. Wurden noch nicht alle Properties des Elementes abgebildet, dann muss

[4] `https://mariadb.com/kb/en/system-versioned-tables/`
[5] `https://mariadb.com/kb/en/constraint/`
[6] `https://mariadb.com/kb/en/insert/`

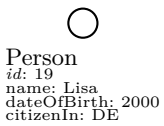

Abbildung 5.1.: Ein Beispiel-Knoten

noch für jede weitere Property eine `INSERT` Anfrage in die Property-Tabelle durchgeführt werden (solang der logische Typ auch eine Property-Tabelle besitzt).

Auf gleiche Weise können Elemente, die bereits abgebildet wurden, aktualisiert werden. Die Anfragen sind gleich aufgebaut, nutzen aber die MariaDB spezifische SQL-Erweiterung `REPLACE`[7]. Dabei werden existierende Einträge in den Tabellen überschrieben. Die Abbildung eines Elementes auf Relationen ist bereits in Unterabschnitt 4.2.1 erläutert wurden.

```
1  INSERT INTO Person_v (_id, name, dateOfBirth)
2    VALUES (19, 'Lisa', '2000');
3  INSERT INTO Person_v_p (elementId, _key, _value)
4    VALUES (19, 'citizenIn', 'DE');
```

Listing 5.9: INSERT Queries für einen Beispiel-Knoten

Um einen Knoten wie in Abbildung 5.1 einzufügen, müssen die Queries in Listing 5.9 ausgeführt werden. Die Properties *name* und *dateOfBirth* sind entsprechend des Schemas in die Knoten-Tabelle, alle weiteren in die Property-Tabelle einzufügen.

5.4. Anfragen von Elementen

Im Prototyp wird zur Anfrage von Elementen aus der Datenbank eine generische Methode gegeben, die Subgraph und Snapshot integriert. So kann beispielsweise der komplette Graph angefragt werden, indem als Prädikat $\varphi(x) = true$ gewählt wird. Bei jeder Anfrage kann zudem ein temporales Prädikat genutzt werden, um zusätzlich den Snapshot Operator zu emulieren.

Als Eingabe für die Anfrage von Elementen dient dabei eine Menge von Typ-Label für Knoten, eine Menge von Kanten-Headern, sowie je ein Prädikat. Der Implementierung geht dann wie folgt vor:

1. Bestimmen den entsprechenden Relationen:
   Mit Hilfe des `GraphSchema` wird zu jedem Typ-Label der passende Typ (`ElementSchema`) bestimmt.

2. Die Prädikate werden je analysiert und alle Attribute und Properties, die im Prädikat referenziert sind, werden extrahiert. Es ergeben sich zwei Mengen: Eine Menge von Attributen, welche in Kanten-Arten übersetzt werden (ID, Label, Start- oder Endzeitpunkt einer Zeitdimension, etc.), sowie eine Menge von Property-Namen. Diese Informationen werden in einer POJO-Klasse `TableAccessHint` kodiert.

[7] `https://mariadb.com/kb/en/replace/`

3. Über die in der `QuerySupport` Klasse, deren API auch welche nun auch Query-API genannt wird, wird für jeden logischen Typ eine `SELECT` Anfrage erzeugt, die genau diese Werte zurückgibt. Falls der Typ ein bestimmtes Attribut oder eine Property nicht speichern kann, wird dieser mit `null`-Werten aufgefüllt. Falls das Typ-Label angefragt wird, der Typ aber keine Label-Spalte besitzt, weil er nur Typen eines bestimmten Labels enthält, dann wird das entsprechende gespeicherte Label als Konstante zurückgegeben.

4. Die Query-API gibt neben einer `SELECT` Anfrage auch die Struktur des Resultats der Anfrage als `Table` Objekt zurück. Dadurch kann die Anfrage weiter verarbeitet werden, es ist klar in welcher Spalte welcher Wert abgebildet ist und es wird die Abbildung auf Elemente ermöglicht. (Dies wird später erklärt.)

5. Mit dem `Table`-Objekt, also der Struktur der generierten Anfrage, ist auch klar in welcher Spalte welche Property gespeichert ist. Entsprechend kann dann das Prädikat von Referenzen auf Property-Namen in welche auf Spaltennamen übersetzt werden. (Wie im Subgraph-Operator, siehe Unterabschnitt 4.2.4).

6. Die Query, an die nun noch die Prädikate angehangen werden, kann ausgeführt werden.

7. Nun wird noch gegebenenfalls je die Property-Tabelle angefragt, um verbleibende Properties zu erhalten.

```
1  class TableAccessHint {
2    List<String> getPropertyKeys();
3    Set<ColumnType> getAdditionalColumns();
4  }
5  class SqlQueryHolder<S> {
6    S getQuery();
7    Table getStructure();
8  }
9  class QuerySupport {
10   SqlQueryHolder<SelectQuery> attachPredicates(
11       SqlQueryHolder<SelectQuery>,
12       BooleanValue predicate);
13   SqlQueryHolder<SelectQuery> createSelectForVertexType(
14       VertexSchema type,
15       TableAccessHint accessedData);
16   SqlQueryHolder<SelectQuery> createSelectForEdgeType(
17       EdgeSchema type,
18       TableAccessHint accessedData);
19   SqlQueryHolder<SelectQuery> snapshot(
20       SqlQueryHolder<SelectQuery> query,
21       SnapshotDefinition snapshotDefinition)
22   void setSnapshot(SnapshotDefinition snapshot);
23 }
```

Listing 5.10: Teil der Query-API der Implementierung

In Listing 5.10 ist der hier relevante Ausschnitt der Query-API gezeigt. `BooleanValue` ist die kodierte Form der Prädikate. Die `createSelectForXType` Methoden gehen so vor wie im Konzept

zum Subgraph Operator (Unterabschnitt 4.2.4) beschrieben. Dabei wird anhand des jeweiligen `ElementSchema` bestimmt, welche Typen in der Datenbank in welcher Spalte angebildet sind. Für Properties, die nicht in der Element-Tabelle gespeichert sind, wird je ein Join mit der Property-Tabelle durchgeführt. Über die `setSnapshot` Funktion wird optional ein Snatshot festgelegt, der für jede `SELECT` Anfrage extrahiert wird. Dies ist in SQL:11 [13] mittels `FOR SYSTEM_TIME AS OF` nach einem Tabellennamen möglich. Alternativ kann die `snapshot` Funktion verwendet werden, um einer Query eine Snapshot Extraktion hinzuzufügen. Dies nutzt eine Möglichkeit von MariaDB, mit der für Sub-Queries ein Snapshot extrahiert werden kann.[8] Beide Anfragen in Listing 5.11 sind äquivalent.

```
1  SELECT _id, 'Person' as _label, name
2  FROM Person_v
3  FOR SYSTEM_TIME AS OF (NOW() - INTERVAL 1 MINUTE);
4
5  SELECT * FROM (
6    SELECT _id, 'Person' as _label, name
7    FROM Person_v
8  ) sub FOR SYSTEM_TIME AS OF (NOW() - INTERVAL 1 MINUTE);
```

Listing 5.11: Alternative Extraktion von Snapshots in MariaDB

5.5. Abbildung von Relationen auf Elemente

Wie erwähnt wird zu jeder generierten Anfrage die Struktur, als `Table` Objekt, also als relationale Schema-Sicht zurückgegeben. Diese kann dann genutzt werden, um die Relation in Elemente umzuwandeln. Diese Objekte enthalten wie auf Seite 46 beschrieben Informationen wie Art der Elemente (Knoten oder Kanten), Label (falls fest, andernfalls über Spalte) und enthaltene Spalten. Wird die Anfrage ausgeführt, dann kann jede Zeile des Resultats in ein Element umgewandelt werden:

1. Zuerst wird ein leeres Element (als Java Objekt) erzeugt. Dabei wird je nach Tabellen-Art Knoten oder Kante gewählt.
2. Falls der Tabelle ein festes Typ-Label zugewiesen ist, dann wird dieses als Label gesetzt.
3. Danach werden die Spalten der Tabelle iteriert. Diese sind geordnet, also kann der erste Wert der Zeile entsprechend der Art der ersten Spalte der `Table` verarbeitet werden. Dabei wird nach Typ unterschieden. Der Wert von `Label` Spalten wird mit `setLabel` gesetzt, usw. Der Wert zu jeder Spalte wird in das Element übertragen.

Wie auch in Unterabschnitt 4.2.3, werden noch Einträge der Property-Tabelle verarbeitet.

[8] `https://mariadb.com/kb/en/system-versioned-tables/`

5.6. Pattern Matching

In Unterabschnitt 4.2.6 wurde bereits demonstriert, die GDL Anfragen [10] auf das relationale Modell übersetzt werden können. Zur Umsetzen bietet es sich an, bereits eingeführte Funktionalitäten der Query-API wiederzuverwenden. Dabei nehme man wieder eine GDL-Query. Diese besteht aus zwei Teilen: Eine Beschreibung der erwarteten Struktur des Teilgraphen und einem Prädikat. Die Struktur ist durch eine Menge von Variablen beschreiben, wobei jeder Variable ein Typ-Label zugeordnet sein muss. Pfade sind im Übrigen entsprechend dieser Limitierung nicht unterstützt, da jedes Element in der Query bekannt sein muss, aber bei Pfaden nur Start- und Endpunkt festgelegt sind. Jede Variable repräsentiert dabei einen Knoten oder eine Kante und es ist kodiert, wie Kanten und Knoten verbunden. Zur Ausführung der Query wird diese erst in eine interne Darstellung überführt. Diese besteht aus: einer Menge von Variablen, eine Zuordnung von Variable zu Typ (Knoten oder Kante), eine Zuordnung von Variable zu Typ-Label (Typ-Header für Kanten ergibt sich daraus) und einer internen Darstellung des Prädikates. Die Query kann dann in folgenden Schnitten ausgeführt werden:

1. Wie bei Subgraph wird das Prädikat analysiert. Hier wird jedoch für jede Variable bestimmt, welche Properties und sonstige Attribute referenziert werden.
2. Wieder wird für jede Variable, dessen Typ bekannt ist, ein `TableAccessHint` erzeugt.
3. Mit der Query-API wird zu jeder Variable eine Anfrage (mit der Struktur, als Metadaten) erzeugt.
4. Über alle Anfragen kann wie im Konzept beschrieben eine Anfrage für das Produkt der Resultate konstruiert werden.
5. Mit Query-API wird das Prädikat übersetzt und an die Anfrage angehangen. Auch werden Join-Bedingungen zwischen und Knoten- und Kantentypen deklariert.
6. Aus dem Resultat der Anfrage wird je ein Tupel abgebildet, was jeder Variable eine Element-ID zuordnet. Mit dem bereits bekannten Typ, können dann die entsprechenden Elemente angefragt werden.

Die Rückgabe des Operator ist eine iterierbare Menge von Graphen, wobei zuerst nur leere Elemente erzeugt werden, die nur Label und ID enthalten. Erst tatsächlich auf einen Graphen zugegriffen wird, werden die Elemente angefragt. Da mehrere Graphen ein gleiches Element enthalten können, sollte verhindert werden, dass Elemente mehrfach angefragt werden. Dazu wird ein LRU-Cache (mit einer Implementierung von Apache Commons[9]) genutzt, der zuletzt referenziert Elemente speichert. Wird über einen Graphen iteriert, dann wird für jedes Element geprüft, ob dieses bereits angefragt wurde. In diesem Fall wird statt Anfrage der Cache genutzt.

Der Pattern Matching Operator ist also in zwei Teile geteilt: Anfrage der IDs der Elemente jedes passenden Teilgraphen und Anfrage der tatsächlichen Graphen. Letzteres geschieht erst, wenn

[9] `https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/map/LRUMap.html`

der auf einen resultierenden Graphen zugegriffen wird. Durch einen (optionalen) Cache werden Anfragen reduziert.

5.7. Weitere Bemerkungen zur Implementierung

An dieser Stelle seien noch kurz einige Funktionalitäten genannt, die zusätzlich implementiert wurden. Als Einstiegspunkt ist dabei meist die `GraphDataBase` Klasse anzusehen, über die die meisten Operatoren direkt aufgerufen werden können. Es ist beispielsweise ein Operator gegeben, mit dem Daten aus einem `GraphSchema` in ein anderes übertragen werden können. Dieser wird `SchemaMigration` genannt. Es finden sich auch eine Reihe von Hilfsoperatoren, die dazu genutzt werden können Daten aus anderen Formaten zu importieren. Insbesondere sind dabei Importe aus GDL Anfragen und CSV Dateien möglich. In Listing 5.12 wird kurz gezeigt, wie ein `GraphDataBase` Objekt erzeugt werden kann.

Es ist auch eine Konsolen-Anwendung enthalten, mit der manche Operationen direkt angesteuert werden können. Diese findet sich im `cli` Modul des Software-Projektes.

```
1  GraphSchema schema = new SchemaBuilder()....;
2  DataSource sqlDataSource = new MariaDbDataSource("jdbc:mariadb:...");
3  SqlExecutionHandler handler = new DefaultHandler(sqlDataSource, SqlWriter::new);
4  GraphDataBase<Long> db = new GraphDataBase<>(schema, handler,
5    new PgmElementFactory(DefaultTypes.BIGINT));
```

Listing 5.12: Initialisierung der Graph-Datenbank API des Prototyps

6. Auswertung

Zur Evaluation und Auswertung des implementierten Prototyps werden einige Anfragen durchgeführt und die Laufzeiten gemessen. Dazu werden Einfüge-Operationen und generische Graph-Anfragen (siehe Abschnitt 5.4) durchgeführt, welche Subgraph und Snapshot integrieren.

6.1. Versuchsaufbau

Es wird erwartet, dass die Art, wie das Graph-Schema definiert ist, die Performance des Systems beeinflusst. So wird erwartet, dass Anfragen über mehrere Tabellen langsamer sind als solche, die nur auf einzelne Tabellen zugreifen. Um dies zu untersuchen, wird zuerst beschrieben, welche Daten genutzt werden.

6.1.1. Daten

Zum messen einfacher Graph-Anfragen wird ein zufälliger Graph generiert. Dieser besteht aus 3 Knoten-Typen (Labels): a, b und c, sowie 18 Kanten-Typen mit Label e und f. Die erhöhte Zahl der Kanten-Typen ergibt sich daher, dass diese nach Typ-Header unterschieden werden. Da die Kanten zufällig Knoten mit drei verschiedenen Typ-Labels verbinden, ergeben sich $\#KantenLabels \cdot \#KnotenLabels^2 = 2 \cdot 3^2 = 18$ Typen. Jedem Typ werden drei Properties $p1$, $p2$ und $p3$ zugewiesen, wobei je ein zufälliger Zahlenwert von 0 bis 9 gewählt wird. Typ-Label und Property-Werte werden zufällig festgelegt. Es wird eine feste Anzahl von Knoten und Kanten erzeugt. Die Quell- und Ziel-Knoten zu jeder Kante werden frei bestimmt. Es kann erwartet werden, dass zufällige Graphen entsprechend der Typen annähernd gleich verteilt sind, sofern genügend Elemente erzeugt werden.

6.1.2. Graph-Schema

Es soll der Einfluss der Graph-Schemata gemessen werden. Hierfür werden 4 verschiedene Schemata definiert:

1. TFL-ähnliches Schema mit Properties in Spalten ($TFL + C$)
   Zu jedem Typ wird ein eigener logischer Typ definiert. Also wird jeder Typ in einer eigenen Tabelle abgelegt. Für jede Property wird eine eine Spalte angelegt, die Property-Tabelle wird nicht genutzt.
2. TFL-ähnliches Schema mit Property-Tabelle ($TFL + P$)
   Wieder wird je ein eigener logischer Typ definiert. Properties werden aber diesmal jeweils in der Property-Tabelle gespeichert, nicht in Spalten.

3. GVE-ähnliches Schema mit Properties in Spalten ($GVE + C$)
   Hier wird je nur der generische Knoten- und Kanten-Typ genutzt. Die Typen werden also alle in einen logischen Typen (einen für Knoten und einen für Kanten) abgebildet. Properties sind in passenden Property-Spalten abgelegt und die Property-Tabelle wird wieder nicht genutzt.

4. GVE-ähnliches Schema mit Property-Tabelle ($GVE + P$)
   Hier werden auch nur die generischen logischen Typen genutzt, jedoch sind diesmal Properties in der Property-Tabelle und nicht als Spalten repräsentiert.

Es kann also je überprüft werden, ob eine Property-Tabelle oder Property-Spalten und ob generischer Typ oder Typen für jedes Label präferiert sein sollten.

6.1.3. Anfragen

Der generische Anfrage Operator wird genutzt. Mit diesem wird je ein Teil-Graph anfragt. Wie in Abschnitt 5.4 genannt, dient dabei je optional eine Liste von Knoten- und Kanten-Typen und ein optionales Prädikat zur Auswahl bestimmter Knoten und Kanten. Es werden hier 4 verschiedene Anfragen ausgeführt:

1. Alle Elemente (All)
   Wenn keine Typen und kein Prädikat angegeben sind, dann wird der komplette Graph aus der Datenbank angefragt. Dies entspricht dem Prädikat

$$\varphi(e) = true \tag{6.1}$$

2. Elemente bestimmter Typen ($Some$)
   Es werden die Knoten-Typen a und b, sowie die Kanten-Typen $\langle e, a, b\rangle$ (Kanten-Typ e von Quell-Knoten a zu b) und $\langle f, b, d\rangle$ angefragt, wobei kein weiteres Prädikat angegeben ist. Hier sei bemerkt, dass Elemente des Typen $\langle f, b, d\rangle$ tatsächlich *nicht* Teil des Resultates sein können, da Knoten vom Typ d nicht Teil des angefragten Teilgraphen sind. Dies wird lediglich genutzt, um zu testen, dass der Prototyp dies beachtet.

3. Elemente aller Typen, mit Prädikaten ($All - Filter$)
   Es werden Elemente aller Typen ausgewählt. Dabei wird ein Prädikat definiert, wodurch dann nur ein Teilgraph über alle logischen Typen bestimmt wird. Hier wird je das Prädikat

$$\varphi(e) = \begin{cases} true & e.p1 \geq 3 \\ false & sonst \end{cases} \tag{6.2}$$

   genutzt.

4. Elemente bestimmter Typen, mit Prädikaten ($Some - Filter$)
   Als Kombination der bisherigen Anfragen werden wie bei $Some$ nur bestimmte Element-Typen ausgewählt. Zusätzlich wird das gleiche Prädikat wie für $All - Filter$ verwendet.

	$TFL+C$	$TFL+P$	$GVE+C$	$GVE+P$
`INSERT`	372622	1432085	365502	368797
All	3167	4608	7031	6236
(+temporal)	4379	6513	9136	9542
Some	567	949	3278	3242
(+temporal)	767	1296	5214	5334
$All - Filter$	1234	10745	1546	1542
(+temporal)	1998	13926	2540	2657
$Some - Filter$	267	1611	1056	1066
(+temporal)	351	2081	1793	1911

Tabelle 6.1.: Übersicht der Anfragezeiten (in Millisekunden)

6.1.4. Anfragen für temporale Daten

Um den Einfluss der Verwendung von temporalen Attributen zu prüfen, werden die gleichen Anfragen noch auf historischen Daten ausgeführt. Dazu werden, mit aktivierter System-Versionierung zuerst alle Elemente gelöscht. Die Datenbank wird dann mit einem neu generierten Graph gefüllt.

Anschließend werden die zuvor genannten Anfragen mit einer zusätzlichen `AS OF` Snapshot-Definition versehen, wodurch Elemente aus dem alten Graph-Zustand angefragt werden können.

6.1.5. Details zum System

Anfragen werden auf MariaDB 10.5.9 unter Oracle Linux 7 in einer virtuellen Maschine in Oracle Cloud[10] ausgeführt. Es kommt eine Maschine der Art `VM.Standard.E2.4` mit 4 virtuellen Prozessoren und 32 GB Arbeitsspeicher zum Einsatz. Die Datenbank ist auf einem *Block Volume* der Größe 300GB mit Performance `UHP (VPU:60)` abgelegt. Es wird eine Datendurchsatz Rate von $324 MB/s$ angegeben.

6.2. Ergebnisse

6.2.1. Anfragezeiten

In Tabelle 6.1 sind die Laufzeiten der Anfragen für alle Schemata aufgelistet. Als Eingabe wird ein Graph mit 100.000 Knoten und 300.000 Kanten generiert. Dazu wird auch je die Laufzeit mit Snapshot, also auf temporalen Tabellen aufgezeigt. Hier ist deutlich zu erkennen, dass das TFL-ähnliche Schema mit Properties als Spalten in jedem Fall am schnellsten ist. Die kann damit erklärt werden, dass Anfragen auf Teilmengen des Graphen auch nur einen Teil der Tabellen nutzen. Außerdem sind keine Joins mit Property-Tabellen nötig.

Einfügen ist für nahezu alle Schemata gleich schnell, wobei das TFL-Schema mit Property-Tabelle sehr negativ auffällt. Das Experiment wurde auch mit anderen Graph-Größen wiederholt, wo sich ähnlich schlechte Ergebnisse zeigen. Für dieses Schema ist die Operation immer circa vier mal

[10] `https://www.oracle.com/de/cloud/`

langsamer. Es wird vermutet, dass dies der Implementierung geschuldet ist. In dieser werden Einfügeoperationen mittels *Prepared Statements*[11] durchgeführt. Dieses Schema enthält die meisten Tabellen, daher auch die meisten Statements. Im Gegensatz zu anderen Schemata ist es schlechter möglich, zuerst alle `INSERT`s einer Tabelle gestaffelt auszuführen. Dies könnte ein Grund für das unerwartete Verhalten sein.

Anfrage auf temporalen Daten sind je immer ungefähr 30% langsamer als entsprechende Anfragen ohne Snapshot. Dies ist zu erwarten, da historische Daten separat behandelt werden und gegebenenfalls platzsparender abgelegt sind [22]. Datenstrukturen die Anfragen beschleunigen sind für diese Daten unter Umständen nicht notwendig.

6.2.2. Empfehlungen zur Wahl der Schemata

Anhand der Ergebnisse kann hier eine Empfehlung abgegeben werden, welche Art von Schema zu verwenden ist. Sind Aufbau der Daten, also vorhandene Knoten- und Kanten-Typen, sowie deren Attribute, vorhanden, dann bietet sich ein TFL-ähnliches Schema (also ein nach Labels aufgeteiltes) an. Für dieses kann erwartet werden, dass Anfragen besonders schnell verarbeitbar sind.

Ist die Datenstruktur nicht bekannt, dann ist ein GVE-ähnliches Schema mit Property-Tabelle zu empfehlen. Dieses kann jedes Element abbilden und verhält sich trotzdem verhältnismäßig performant. Zwar sind Anfragen im Vergleich zum zuvor empfohlenen Schema 2 bis 6 mal langsamer, jedoch kann dies durch die höchste Flexibilität diese Schemas gerechtfertigt werden.

[11] `https://mariadb.com/kb/en/prepared-statements/`

7. Zusammenfassung und Ausblick

Zum Abschluss dieser Arbeit werden vorgestellte Konzepte noch einmal kurz rekapituliert und es wird ein Ausblick auf potentielle weitere Entwicklung gegeben.

Mit dieser Arbeit wurde eine flexible Abbildung zwischen Graphen und Relationen formalisiert, sowie implementiert. Mit dieser hat sich allerdings eine Einschränkung ergeben. Kanten sind genau dann nicht abbildbar, wenn sie einen logischen Knoten-Typen zu einem festen Typ-Label mit einem ohne Einschränkung des Labels verbinden würden. In weiteren Arbeiten gilt es hierfür Lösungen zu finden. Es besteht zwar die Möglichkeit für jeden Kanten Typ auch jeden Knonte-Typ mit festem Label mit dem generischen Typ zu verbinden, jedoch führt dies dazu, dass das Schema mit dem Anzahl der Typ-Labels zu schnell an Komplexität gewinnt. Es gilt also elegantere Ansätze zur Lösung zu finden.

Diese Arbeit implementiert ein Property-Graph Modell auf relationalen Datenbanken. Es bietet sich nun noch an, die Erweiterungen in Form des EPGM und TPGM umzusetzen, wobei ersteres bereits für relationale Modelle abgebildet wurde. (Wenn auch weniger flexibel als hier demonstriert.)

In der Auswertung konnte gezeigt werden, dass ein Schema besonders langsame Beantwortung von Anfragen verursacht. Dies ist zu untersuchen. Insbesondere sollten einzelne Komponenten dieser Implementierung analysiert werden. Hier wurden nur Operatoren als ganzes geprüft. Es kann sich als sinnvoll herausstellen, zu untersuchen, welcher Teil zu diesem negativen Ergebnis führt. Gegebenenfalls ist, falls dies die Ursache ist, die anwendungsseitige Abbildung der Elemente auf Tabellen zu optimieren.

Weiter wurden hier in der Datenbank keine Optimierungen durchgeführt. Es bleibt zu untersuchen, ob beispielsweise die Definition von Index-Strukturen in der Datenbank zu besseren Ergebnissen führen kann.

Literaturverzeichnis

[1] Timo Adameit. Evaluation des EPGM auf Basis von Apache Spark. Masterarbeit, Universität Leipzig, 2020.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, nov 1983.

[3] Renzo Angles. The property graph database model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2018.

[4] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[5] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, December 1979.

[6] Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Laboratories Bristol, Bristol, UK, September 2005.

[7] Richard Cyganiak, Seema Sundara, and Souripriya Das. R2RML: RDB to RDF mapping language. W3C recommendation, W3C, September 2012. `https://www.w3.org/TR/2012/REC-r2rml-20120927/`.

[8] Christopher J. Date. *An Introduction to Database Systems*. Pearson Education, eigth edition, 2004.

[9] Reinhard Diestel. *Graphentheorie*. Springer Spektrum, 2017.

[10] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems - GRADES'17*. ACM Press, 2017.

[11] Martin Junghanns, André Petermann, Kevin Gómez, and Erhard Rahm. GRADOOP: Scalable graph data management and analytics with hadoop. Technical report, University of Leipzig & ScaDS Dresden/Leipzig, 2015.

[12] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and Erhard Rahm. Analyzing extended property graphs with apache flink. In *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics - NDA '16*. ACM Press, 2016.

[13] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *ACM SIGMOD Record*, 41(3):34–43, oct 2012.

[14] József Marton, Gábor Szárnyas, and Dániel Varró. Formalising opencypher graph queries in relational algebra.

[15] OpenHMS. SqlBuilder - builder style classes for creating SQL queries. `https://openhms.sourceforge.io/sqlbuilder/`, 2006-2021. Zugriff am 30.07.2021.

[16] Oracle. Java JDBC API. `https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/`, 1993,2021. Zugriff am 30.07.2021.

[17] Jaroslav Pokorný. Conceptual and database modelling of graph databases. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, page 370–377, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. Distributed temporal graph analytics with GRADOOP. *The VLDB Journal*, may 2021.

[19] Christopher Rost, Andreas Thor, Philip Fritzsche, Kevin Gomez, and Erhard Rahm. Evolution Analysis of Large Graphs with Gradoop. In P. Cellier and K. Driessens, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 1167 of *Communications in Computer and Information Science*, pages 402–408. ECML PKDD 2019, Springer, Cham, 2020.

[20] Christopher Rost, Andreas Thor, and Erhard Rahm. Temporal graph analysis using gradoop. In Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke, editors, *BTW 2019 – Workshopband*, pages 109–118. Gesellschaft für Informatik, Bonn, 2019.

[21] Elias Saalmann. Relationale abstraktion des EPGM unter verwendung der Apache FlinkTable-API. Masterarbeit, Universität Leipzig, 2019.

[22] Richard T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann series in data management systems. Morgan and Kaufmann, San Francisco, CA, 1999.

[23] Michael Sperberg-McQueen, Eve Maler, François Yergeau, Jean Paoli, and Tim Bray. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. https://www.w3.org/TR/2008/REC-xml-20081126/.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.
Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Leipzig, den 02.08.2021

Philip Fritzsche