Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Eine Einführung in das Graphdatenbankmodell

Seminararbeit

Seminar NoSQL-Datenbanken

vorgelegt von

Marcus Stuber

Master Informatik

Betreuer: Dr. Michael Hartung

Leipzig, 27. März 2012

Vorbemerkung

Diese Arbeit soll eine Einführung in das Feld der Graphdatenbanken geben. Der Fokus liegt hierbei auf den allgemeinen Eigenschaften dieses Modells. Auf die Vorstellung konkreter Systeme und deren Implementierung soll hingegen verzichtet werden, da dies Thema nachfolgender Arbeiten sein wird. Die nachfolgenden Ausführungen orientieren sich größtenteils an [1], [2] und [6].

Inhaltsverzeichnis

1 Motivation **1**
1.1 Einleitung 1
1.2 Relationale Datenbanken 2
1.3 Graphdatenbanken 3

2 Datenbankmodell **5**
2.1 Graphen 5
2.2 Property-Graphen 6
2.3 Property-Graph-Varianten 8
2.4 Realisierung 9

3 Traversierung **12**
3.1 Traversierungsoperationen 12
3.2 Traversierungsschema 14

4 Zusammenfassung **16**

Literaturverzeichnis **17**

1 Motivation

1.1 Einleitung

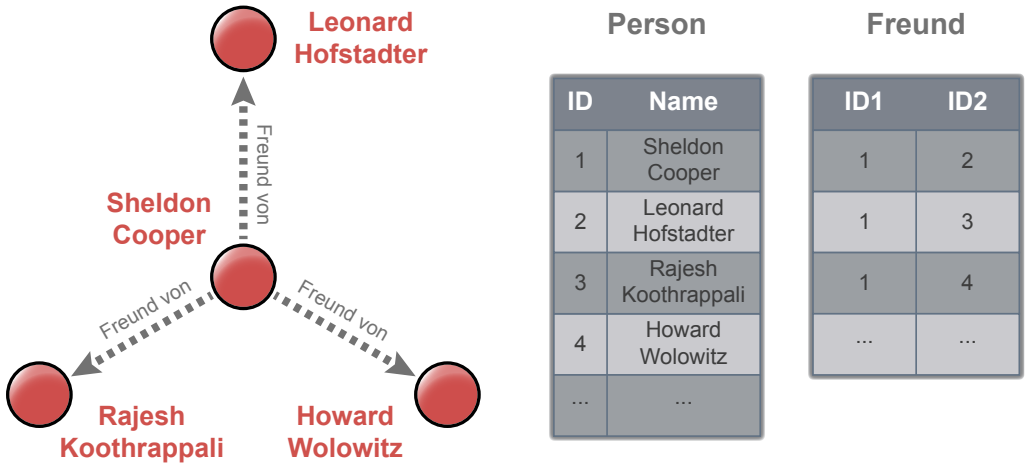
In den letzten Jahren hat das Interesse an komplexen, datenintensiven und verteilten Anwendungen stetig zugenommen, nicht zuletzt wegen der fortlaufenden Entwicklung des Internets und der Verbreitung mobiler Computer. Beispiele hierfür sind die Forschung auf den Gebieten des Semantic Web, der Wissensrepräsentation und der Bioinformatik, die Weiterentwicklung von Geoinformationssystemen, aber auch der Erfolg sozialer Netzwerke. Derartige Anwendungen zeichnen sich durch hochgradig vernetzte Daten aus. Der effiziente Umgang mit solchen Daten ist die große Stärke von Graphdatenbanken.

Graphdatenbanken sind keine vollkommen neue Technologie. Über Jahrzehnte hinweg wurden auf dem Gebiet graphbasierter Datenmodelle und der Verarbeitung großer verteilter Graphen umfangreiche Forschungen durchgeführt. Allerdings rückte Ende des 20. Jahrhunderts der Fokus zunächst auf spezialisierte Ansätze wie objektorientierte oder hierarchische Datenbankmodelle. Seit einigen Jahren gewinnen jedoch auch vollwertige Graphdatenbanken zunehmend an Bedeutung.

Ein Grund hierfür ist, dass sich Datensätze sehr vieler Anwendungsgebiete in natürlicher Weise auf Graphen abbilden lassen. Man denke etwa an die Hyperlink-Struktur des World Wide Web, Bekanntschaften in sozialen Netzwerken, geographische Informationen oder Verkehrsdaten. Darüber hinaus lassen sich auch verschiedenste Problemstellungen mit Hilfe wohlbekannter Graph-Algorithmen lösen. Beispiele hierfür sind das Internet-Routing, Fahr- und Flugplanoptimierung, die Ausbreitungsvorhersage von Krankheiten, Empfehlungssysteme sowie Verkehrsleitsysteme.

Ein weiterer Grund für die zunehmende Verbreitung von Graphdatenbanken ist der Bedarf an mehr Flexibilität sowie an neuartigen Verfahren zur Datenmodellierung, Abfrage und Speicherung. Dazu gehört vor allem:

- ⋆ effiziente Unterstützung semistrukturierter Datensätze
- ⋆ einfacher Umgang mit vernetzten Informationen innerhalb unterschiedlicher Datenquellen und Datenschemata
- ⋆ versionierte Datensätze und Datenschemata, um Änderungen und Erweiterungen der Anwendungen zu ermöglichen
- ⋆ effiziente Unterstützung graphorientierter Operationen wie beispielsweise Shortest-Path-Anfragen
- ⋆ bessere Integration in objektorientierte Programmiersprachen
- ⋆ einfache Skalierbarkeit bezüglich Anfragen und Datenvolumen durch automatische oder teilautomatische Replikation oder Partitionierung von Datensätzen auf mehrere unabhängige Datenbankinstanzen



(a) (b)

Abbildung 1.1: Ein einfacher Graph und dessen relationale Abbildung.

Das bisher vorherrschende relationale Datenbankmodell kann diese Anforderungen, wenn überhaupt, nur unzureichend erfüllen.

1.2 Relationale Datenbanken

Relationale Datenbanken basieren auf strengen Datenschemata. Selbst überschaubare Änderungen wie die Zusammenführung oder Erweiterung von Tabellen können schnell kompliziert und zeitaufwändig werden. Ein solches Modell ist denkbar ungeeignet für semistrukturierte, sich kontinuierlich weiterentwickelnde Datensätze und erschwert zusätzlich die Zusammenführung verschiedener Datenquellen. Ein weiteres Hindernis stellt die Integration relationaler Datenbanken in die heute sehr verbreiteten, objektorientierten Programmiersprachen dar, denn dies erfordert für gewöhnlich den Einsatz von komplizierten Hilfsmitteln wie objektrelationalen Mappern, beispielsweise Hibernate.

Obwohl all dies bereits die Unzulänglichkeiten des relationalen Datenbankmodells aufzeigt, gilt es noch eine weitere Schwachstelle zu nennen, die es mit anderen Modellen teilt. Die graphorientierte Traversierung vernetzter Daten ist sehr ineffizient. Dies lässt sich am besten anhand eines Beispiels verdeutlichen. Abbildung 1.1 zeigt einen Ausschnitt eines einfachen Freundesnetzwerks sowie eine mögliche Realisierung dieses Netzwerks innerhalb relationaler Datenstrukturen. Diese besteht aus genau zwei Tabellen, **Person** und **Freund**, die über Schlüsselwerte miteinander verbunden sind. Die Freund-Tabelle referenziert jeweils zwei befreundete Personen aus der Person-Tabelle. Beide Tabellen verfügen, wie in relationalen Datenbanken üblich, über eine Indexstruktur. Diese ermöglicht es, Tabellen effizient nach Zeilen mit bestimmten Spaltenwerten zu durchsuchen.

Eine verbreitete Klasse von Indexstrukturen stellen etwa Suchbäume dar, die es erlauben, eine Suche in einer Menge aus n Datensätzen in typischerweise $\mathcal{O}(\log n)$ Zeit zu bewerkstelligen.

Das Ziel sei nun, die Namen aller k Freunde von Sheldon Cooper zu ermitteln. Ein relationales Datenbanksystem mit n Personen müsste dann die folgenden Operationen ausführen, um diese Anfrage zu beantworten.

1. durchsuche **Person.Name** nach „Sheldon Cooper“
   $\mathcal{O}(\log n)$
2. ermittle **Person.ID** der Ergebniszeile
   $\mathcal{O}(1)$
3. durchsuche **Freund.ID1** nach dieser **Person.ID**
   $\mathcal{O}(\log n)$
4. ermittle **Freund.ID2** der k Ergebniszeilen
   $\mathcal{O}(k)$
5. durchsuche **Person.ID** nach diesen **Freund.ID2**
   $\mathcal{O}(k \log n)$
6. ermittle **Person.Name** der k Ergebniszeilen
   $\mathcal{O}(k)$

Die Operationen 3. und 5. realisieren hierbei die aus dem relationalen Datenbankmodell bekannten Join-Anweisungen. Die Ausführungszeit dieser Operationen, und damit die Traversierungszeit, steigt mit zunehmender Datenbankgröße. Auch wenn der Zusammenhang in der Regel nur logarithmisch ist, so stellt dies bei Millionen von Datensätzen doch einen nicht zu unterschätzenden Faktor dar. Hinzu kommt, dass selbst zur Beantwortung solch einfacher Anfragen oftmals eine ganze Reihe dieser relativ aufwändigen Operationen notwendig ist, eine Zahl, die mit steigender Komplexität schnell steigt. Aus diesem Grund eignet sich das relationale Datenbankmodell nur schlecht für derartige Suchvorgänge.

Zwar lassen sich einige der Schwachstellen dieses Modells beheben, etwa mit Hilfe proprietärer Spracherweiterungen, zusätzlicher Softwarekomponenten wie den bereits erwähnten objektrelationalen Mappern oder manuellen Partitionierungsalgorithmen, dies führt jedoch oftmals zu neuen Problemen. Beispielsweise wird es schwierig, im Falle partitionierter Datensätze Konsistenzkriterien einzuhalten. Auch die Skalierbarkeit wird beeinträchtigt, da die Auswirkungen zusätzlicher Software auf die Leistung einer Anwendung bei hoher Last und zunehmendem Datenumfang nur schwer abzuschätzen und zu optimieren sind. Letztendlich bieten Relationale Datenbanksysteme keine flexiblen und leistungsfähigen Standardlösungen.

1.3 Graphdatenbanken

Graphdatenbanken stellen hier eine Alternative dar. Sie spezialisieren sich auf die Speicherung vernetzter Informationen und deren effiziente Traversierung. Generell zeichnen sich Graphdatenbaken durch eine sehr gute und einfach abschätzbare Leistung aus, haben

allerdings noch weitere Vorzüge. So lässt sich beispielsweise das objektrelationale Abbildungsproblem bei der objektorientierten Programmierung vermeiden und Datenschemaänderungen für agile Softwareentwicklungsprozesse sind schnell und einfach realisierbar.

Es stellt sich allerdings die Frage, welche Eigenschaften eine Graphdatenbank tatsächlich definieren. Offensichtlich ist die bloße Fähigkeit Graphstrukturen darzustellen kein ausreichendes Kriterium. Wie der vorherige Abschnitt zeigt, können relationale Datenbanken problemlos Graphen abbilden. Auch andere Modelle wie Key-Value-Stores sind dazu in der Lage. Was Graphdatenbanken von diesen Systemen unterscheidet ist die Art und Weise, wie sie diese Strukturen realisieren. Standard-Modelle speichern Graphstrukturen implizit. Daten verweisen aufeinander über Referenzen, Verbindungen werden bei Anfragen dynamisch erzeugt, im Falle relationaler Systeme etwa über Join-Operationen. Graphdatenbanken hingegen bilden die Graphstruktur eines Datensatzes explizit ab und stellen entsprechende Operationen zur effizienten Durchsuchung und Manipulation dieser Struktur bereit. Die hierfür notwendigen Datenstrukturen und Funktionen sollen in den nachfolgenden Kapiteln vorgestellt werden.

2 Datenbankmodell

Graphdatenbanken verwenden graphorientierte Datenmodelle zur Darstellung von Informationen. Es gibt verschiedene Arten von Graphen, die sich hinsichtlich ihrer Ausdruckskraft und Komplexität zum Teil deutlich unterscheiden.

2.1 Graphen

Generell besteht ein Graph aus einer Reihe von Knoten und Kanten. Knoten stellen Objekte dar, während Kanten Beziehungen zwischen diesen Objekten repräsentieren. Einfache Graphen bestehen formal aus einem Tupel $G = (V, E)$ mit einer Knotenmenge V und einer Kantenmenge E. Man unterscheidet hierbei zwischen gerichteten ($E \subseteq V \times V$) und ungerichteten Graphen. Gerichtete Kanten symbolisieren einseitige, ungerichtete hingegen beidseitige Beziehungen.

Eine gebräuchliche Erweiterung dieses Modells stellen gewichtete Graphen dar. Ein gewichteter Graph $G = (V, E, w)$ mit $w\colon E \to \mathbb{R}$ weist jeder Kante e einen numerischen Wert $w(e)$ zu und eignet sich beispielsweise zur Beschreibung von Straßenkarten in Navigationssystemen. Abbildung 2.1 zeigt einen Ausschnitt einer solchen Karte. Knoten

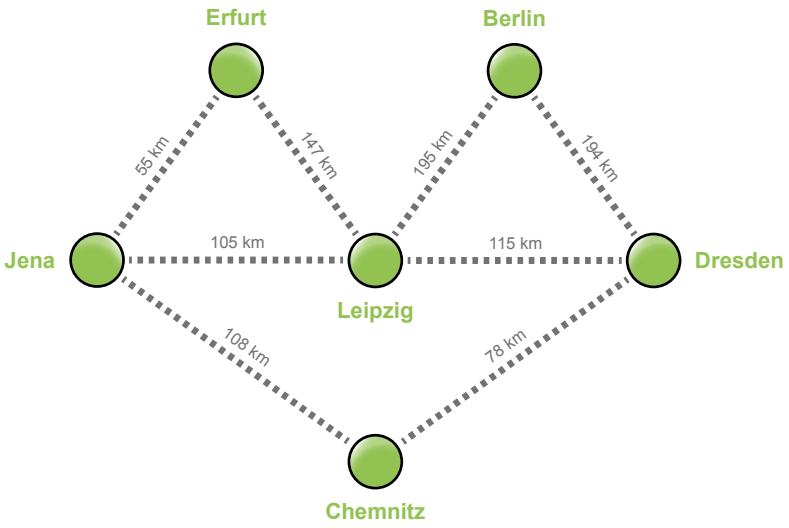
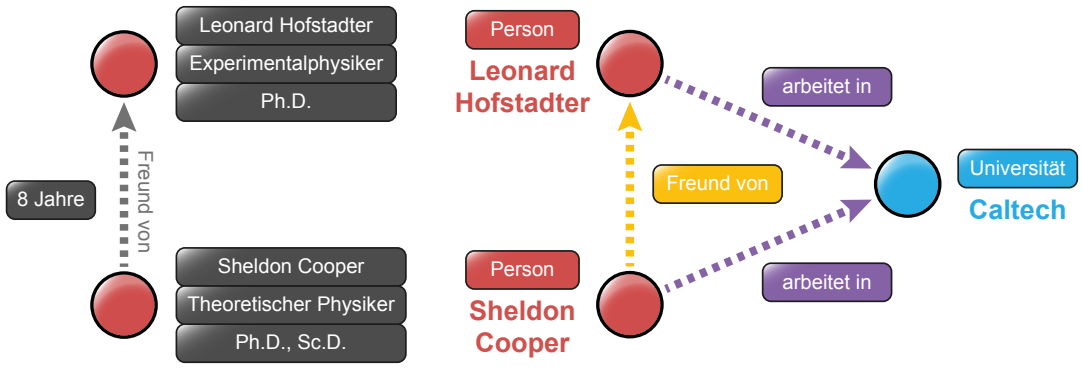


Abbildung 2.1: Ein gewichteter Graph.

symbolisieren Orte. Kanten symbolisieren Straßen, gewichtet durch die jeweilige Entfernung der Orte, die sie verbinden.



(a) (b)

Abbildung 2.2: Ein attributierter und ein beschrifteter Graph.

Das Prinzip gewichteter Graphen lässt sich leicht verallgemeinern. Während es für viele Anwendungsgebiete genügt, Kanten reelle Zahlen zuzuordnen, so kann es zur Abbildung komplexerer Zusammenhänge von Vorteil sein, Informationen beliebiger Art, etwa Zeichenketten oder Zahlenfolgen, darzustellen. Solche Informationen werden im Allgemeinen als Attribute bezeichnet. Natürlich lassen sich nicht nur Kanten, sondern auch Knoten Attribute zuweisen. Abbildung 2.2 (a) zeigt ein Beispiel für einen solchen attributierten Graphen.

Eine weitere verbreitete Variante stellen beschriftete Graphen dar. Beschriftete Graphen versehen Knoten beziehungsweise Kanten mit Bezeichnungen oder Labeln. Dies ermöglicht es etwa, die Bedeutung von Kanten explizit anzugeben und somit auch unterschiedliche Kantentypen zu definieren. Beschriftete Graphen können hierbei durchaus als ein Spezialfall attributierter Graphen angesehen werden. Ein Beispiel für einen beschrifteten Graphen ist in Abbildung 2.2 (b) zu sehen. Es sei an dieser Stelle noch erwähnt, dass die Namen in diesem Beispiel keinen Teil der Beschriftung darstellen. Sie dienen lediglich zur Bezeichnung der verschiedenen Knoteninstanzen.

Die Einbeziehung von Beschriftungen und Attributen erleichtert die Datenmodellierung erheblich. Beschriftete Graphen erlauben eine einfache Typisierung, während attributierte Graphen die Integration zusätzlicher Informationen ermöglichen. Es erscheint naheliegend, die Vorzüge dieser verschiedenen Graph-Varianten in einem Modell zu vereinen. Ein solches Modell stellen die sogenannten Property-Graphen [5] dar.

2.2 Property-Graphen

Ein Property-Graph ist ein gerichteter, beschrifteter, attributierter Graph. Label beschreiben verschiedene Knoten- und Kantentypen. Attribute, in diesem Kontext auch Propertys genannt, erweitern Knoten und Kanten um zusätzliche Eigenschaften in Form

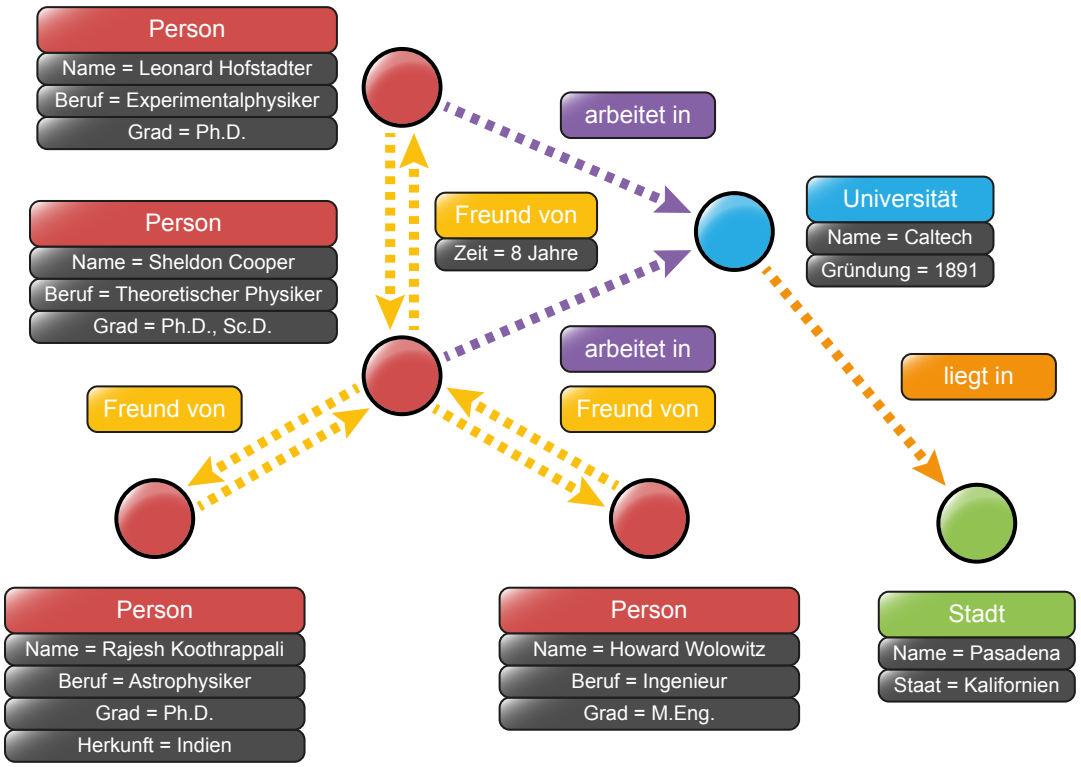


Abbildung 2.3: Ein Property-Graph.

von Schlüssel-Wert-Beziehungen. Mehrfachkanten sind in der Regel nur dann erlaubt, wenn diese unterschiedliche Kantentypen besitzen. Ein Beispiel ist in Abbildung 2.3 zu sehen. Der dargestellte Graph verfügt über eine Reihe verschiedener Knotentypen, die sowohl Personen als auch Orte beschreiben. Kanten repräsentieren verschiedene Beziehungen wie Freundschaften oder Arbeitsverhältnisse. Weiterhin verfügen Knoten wie auch Kanten über Attribute, deren Art und Anzahl variabel sind, teilweise sogar innerhalb ein und des selben Knoten- beziehungsweise Kantentyps.

Property-Graphen können allerdings auch mit einem festen Schema versehen werden. Schemata geben beispielsweise Schlüssel und Wertebereiche von Attributen vor oder beschränken Kantentypen auf bestimmte Kombinationen von Knotentypen. Es ist sogar möglich Typen innerhalb hierarchischer Systeme zu organisieren und somit etwa Konzepte wie Vererbung umzusetzen. Ein Schema besteht in der Regel aus einem Graphen, der Beziehungen zwischen Kanten- und Knotentypen herstellt sowie die zugehörigen Attribute definiert. Einige Graphdatenbanken verzichten jedoch auf derartige Funktionen, sind also schemalos.

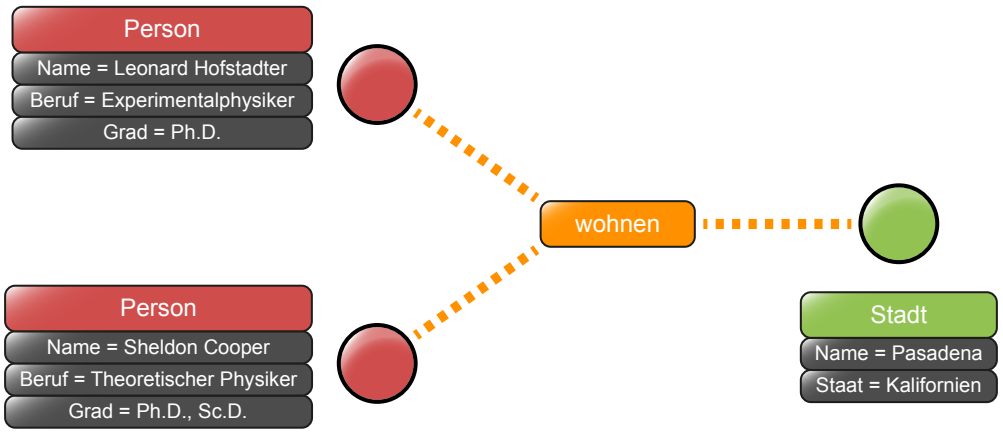


Abbildung 2.4: Ein Hypergraph.

Formal lassen sich Property-Graphen auf verschiedene Art und Weise definieren. Eine mögliche Darstellungsform hierfür ist ein Tupel $G = (V, E, S, T, \lambda_V, \lambda_E, \mu, \varnothing)$ mit

V	einer Menge von Knoten
$E \subseteq V \times V \times L$	einer Menge von Kanten
S	eine Menge möglicher Attribut-Schlüssel
T	eine Menge möglicher Attribut-Werte
$\lambda_V : V \to L$	einer Knoten-Beschriftung
$\lambda_E : E \to L$	einer Kanten-Beschriftung mit $\lambda_E(v_1, v_2, l) = l$
$\mu : (V \cup E) \times S \to (T \cup \{\varnothing\})$	einer Schlüssel-Wert-Zuordnung mit $\mu(o, s) = \varnothing$, genau dann, wenn o kein Attribut s besitzt

und L einer Menge von Labeln. Label sind hier Teil der Kantendefinition, um Kanten verschiedenen Typs formal unterscheiden zu können.

Die vorgestellte Definition ist sehr allgemein gehalten, obwohl sie keine Unterstützung für verschiedene Datenschemata bietet. Alternative Formulierungen hängen stark von den jeweils geforderten Eigenschaften ab und können sich deutlich von der hier präsentierten unterscheiden. Zwar bilden Property-Graphen heute die Basis nahezu aller Graphdatenbanken, dennoch existieren verschiedenste Varianten dieser Datenstruktur.

2.3 Property-Graph-Varianten

Eine Variante stellen sogenannte Hypergraphen dar. Diese erlauben es Kanten nicht nur zwei, sondern eine beliebige Anzahl von Knoten miteinander zu verbinden. Kanten eines Hypergraphs werden auch Hyperkanten genannt. Hyperkanten ermöglichen es, nichtbinäre Beziehungen intuitiv darzustellen. Ein Beispiel hierfür wäre etwa eine Reihe von Personen, die eine gemeinsame Gruppe bilden. Während traditionelle Hypergraphen

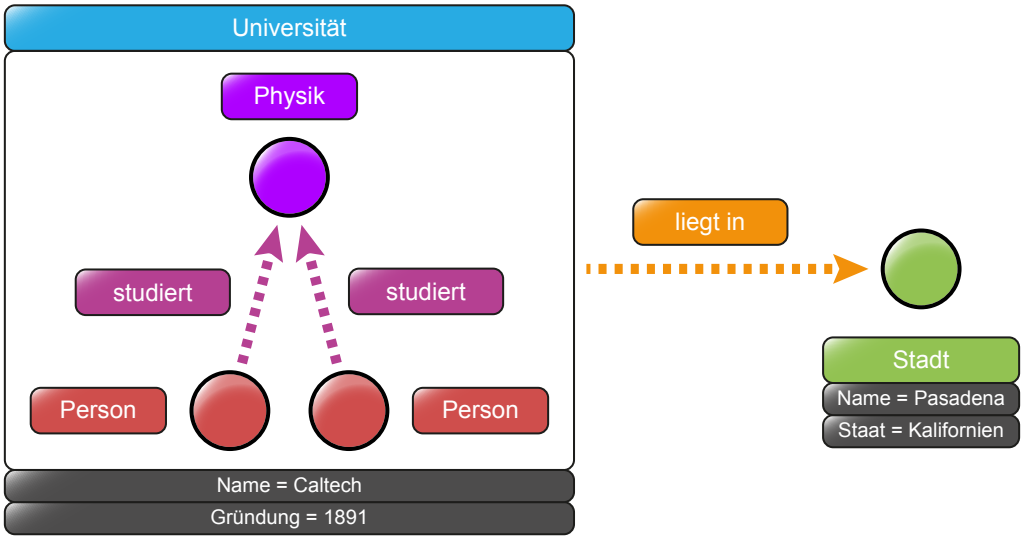


Abbildung 2.5: Ein verschachtelter Graph.

ungerichtet sind, verwenden erweiterte Property-Graphen in der Regel gerichtete Hyperkanten. Diese lassen sich im Wesentlichen auf zwei Arten [3] definieren, entweder über eine Reihe von Ausgangs- und Zielknoten oder aber durch eine Ordnung, also in gewissem Sinne eine Durchnummerierung, der verbundenen Knoten. Abbildung 2.4 zeigt ein Beispiel hierfür. Natürlich können auch Hyperkanten Attribute zugewiesen werden.

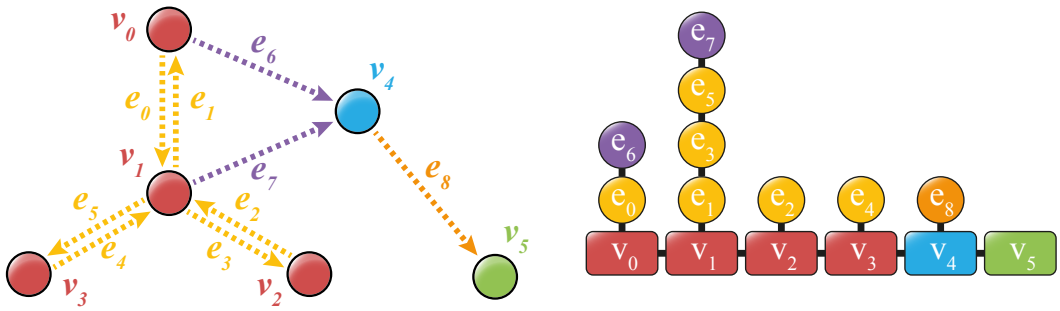
Eine andere Möglichkeit, Property-Graphen zu verallgemeinern, sind verschachtelte Graphen. Die Knoten eines verschachtelten Graphen, auch Hypernodes genannt, können selbst wiederum Graphen darstellen. Auf diese Weise lassen sich verschiedene Komplexitätsebenen erzeugen und Informationen kapseln. Ein Beispiel für einen verschachtelten Graphen ist in Abbildung 2.5 zu sehen.

Neben diesen Erweiterungen gibt es eine Reihe weiterer Unterscheidungsmerkmale der einzelnen Graph-Modelle. Die wesentlichen Kriterien sind hierbei:

- ⋆ Knotenattribute
- ⋆ Kantenattribute
- ⋆ Knotentypen (Knoten-Beschriftung)
- ⋆ Kantentypen (Kanten-Beschriftung)
- ⋆ gerichtete Kanten
- ⋆ ungerichtete Kanten
- ⋆ Hyperkanten
- ⋆ Hypernodes
- ⋆ Schleifen
- ⋆ Schemata

2.4 Realisierung

Neben der Wahl eines geeigneten Graph-Modells stellt die persistente Speicherung der so dargestellten Datensätze eine weitere Herausforderung dar. Von besonderer Bedeu-



(a) (b)

Abbildung 2.6: Ein Graph und seine Darstellung mit Hilfe einer Adjazenzliste.

tung ist hierbei die Fähigkeit, miteinander verbundene Kanten und Knoten identifizieren zu können. Dazu zählt vor allem die Bestimmung der von einem Knoten ausgehenden Kanten, und somit auch seiner Nachbarknoten. Diese Operation ist essenziell für die Traversierung eines Graphen und hat daher auch großen Einfluss auf die Suchleistung eines Datenbanksystems. Wie bereits in Abschnitt 1.2 dargelegt wurde, stellt dies einen Schwachpunkt relationaler Datenbanksysteme dar. Graphdatenbanken erfordern aus diesem Grund die Anwendung komplexerer Datenstrukturen. Es gibt verschiedene solcher Strukturen, die sich im Hinblick auf Zugriffszeit und Speicherbedarf unterscheiden.

Bekannte Verfahren stellen etwa Adjazenz- und Inzidenzmatrizen dar. Eine andere Möglichkeit sind Kantenlisten. All diese Verfahren sind jedoch nur bedingt für den Einsatz in Graphdatenbanken geeignet. Adjazenzmatrizen stellen nur für stark vernetzte Graphen[1] eine praktikable Lösung dar, Inzidenzmatrizen nur für schwach vernetze Graphen. Kantenlisten skalieren zwar gut mit dem Vernetzungsgrad, dennoch hängt die Geschwindigkeit einzelner Traversierungsoperationen wie schon bei relationalen Datenbanksystemen von der Gesamtgröße des Graphen ab. Eine Alternative zu diesen Verfahren stellen Adjazenzlisten dar.

Eine Adjazenzliste besteht im Wesentlichen aus einer Liste der abzubildenden Knoten. Dabei verwaltet jeder der enthaltenen Knoten, wie in Abbildung 2.6 zu sehen, wiederum eine Liste der von ihm ausgehenden Kanten. Der Speicherverbrauch einer Adjazenzliste ist linear, wenn auch höher als der einer Kantenliste, da ja eine Adjazenzliste ebenfalls die Knoten eines Graphen erfasst. Was dieses Verfahren allerdings von den vorher genannten unterscheidet, ist die Tatsache, dass die Bestimmung ausgehender Kanten ausschließlich vom Grad $\deg(v)$, also der Anzahl der ausgehenden Kanten, eines Knotens abhängt, nicht aber von der Gesamtgröße des Graphen. Das heißt, dass bei Traversierungsoperationen lediglich die lokalen Eigenschaften der durchlaufenen Knoten von Bedeutung sind. Dies ist der entscheidende Grund dafür, dass Adjazenzlisten in Graphdatenbanken das mit Abstand am häufigsten eingesetzte Verfahren darstellen.

[1] Unter einem stark vernetzten Graphen ist ein Graph mit einer hohen oder nahezu maximalen Kantenanzahl zu verstehen.

Die hier vorgestellte Variante hat natürlich den Nachteil, dass sie keine effiziente Traversierung eines Graphen entgegen der Kantenrichtung erlaubt, eine Operation, die durchaus von Nutzen sein kann. Diese Schwäche lässt sich jedoch einfach beheben, indem jeder Knoten zusätzlich zur Liste ausgehender Kanten auch eine Liste eingehender Kanten verwaltet, was allerdings auch den Speicherbedarf erhöht.

3 Traversierung

Ein wesentliches Designziel bei der Wahl eines geeigneten Graph-Modells und der zugrundeliegenden Speicherdarstellung ist natürlich die Gewährleistung einer effizienten Datenabfrage. Graphdatenbanken erlauben prinzipiell zwei Arten von Abfragen.

Eine Möglichkeit ist die gezielte Suche eines Knotens oder auch einer Kante mit einem bestimmten Attribut. Dies entspricht einer klassischen Suchanfrage, wie sie beispielsweise relationale Datenbanksysteme durchführen. Ebenso wie relationale verwenden auch graphorientierte Datenbanken hierbei oftmals Indexstrukturen über Attributen, um Suchvorgänge zu beschleunigen.

Die zweite Möglichkeit Daten abzufragen sind Graph-Traversierungen. Traversierungen umfassen das schrittweise Durchlaufen der Elemente, also der Knoten und Kanten, eines Graphen. Auf diese Weise können sowohl einfache Nachbarschaftsabfragen als auch umfangreiche Wegfindungsprobleme gelöst werden. Dazu ist es natürlich notwendig, den jeweiligen Ablauf eines Traverierungsvorgangs zu beschreiben.

3.1 Traversierungsoperationen

Traversierungen lassen sich aus einer Reihe von Einzelschrittoperationen zusammensetzen. Diese führen von einem Graphelement zum jeweils nächsten, etwa von einem Knoten zu einer anliegenden Kante. Für einen Property-Graphen $G = (V, E, S, T, \lambda_V, \lambda_E, \mu, \varnothing)$ lassen sich die folgenden Operationen definieren:

$e_{out} \colon \mathcal{P}(V) \to \mathcal{P}(E)$	traversiere zu den ausgehenden Kanten der Knoten
$e_{in} \colon \mathcal{P}(V) \to \mathcal{P}(E)$	traversiere zu den eingehenden Kanten der Knoten
$v_{out} \colon \mathcal{P}(E) \to \mathcal{P}(V)$	traversiere zu den Zielknoten der Kanten
$v_{in} \colon \mathcal{P}(E) \to \mathcal{P}(V)$	traversiere zu den Ausgangsknoten der Kanten
$p \colon \mathcal{P}(V \cup E) \times S \to \mathcal{P}(T)$	bestimme die Attribut-Werte für einen Schlüssel

Die Definition dieser Funktionen über Potenzmengen ermöglicht es, diese auf einfache Art und Weise zu verketten. Solch eine Verkettung beschreibt dann formal einen Traversierungsvorgang. In der Praxis können diese Operationen auch zusätzliche Informationen transportieren, etwa die zurückgelegten Wege oder auch deren Länge. Dies kann bei gewissen Problemen von Nutzen sein, etwa bei der Bestimmung des kürzesten Pfades zwischen zwei Knoten.

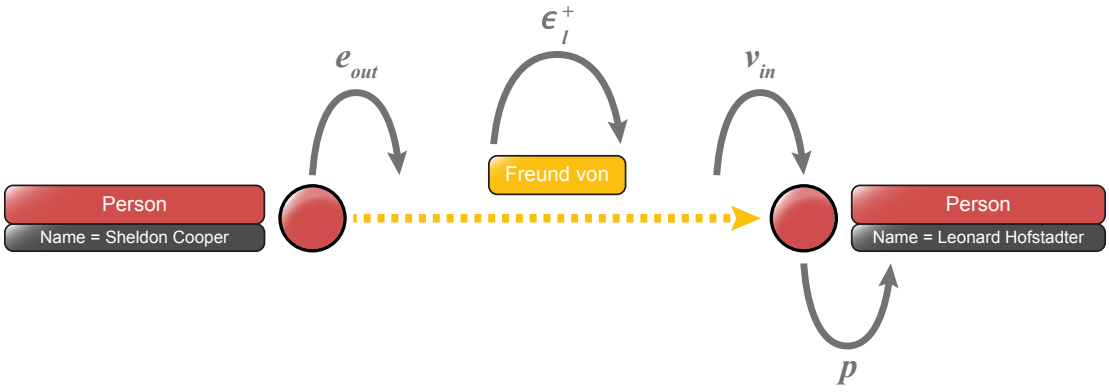


Abbildung 3.1: Die Suche aller Freunde von Sheldon Cooper.

Da Property-Graphen Label und Attribute besitzen, liegt es nahe, diese in die Traversierung mit einzubeziehen. Das erlaubt es, eine Suche auf Elemente mit bestimmten Labeln oder Attributen einzuschränken. Derartige Operationen werden Filter genannt und sehen wie folgt aus:

$\epsilon_l^+ : \mathcal{P}(V \cup E) \times L \to \mathcal{P}(V \cup E)$	behalte nur Elemente mit einem Label
$\epsilon_l^- : \mathcal{P}(V \cup E) \times L \to \mathcal{P}(V \cup E)$	filtere Elemente mit einem Label heraus
$\epsilon_p^+ : \mathcal{P}(V \cup E) \times S \times T \to \mathcal{P}(V \cup E)$	behalte nur Elemente mit einem Attribut
$\epsilon_p^- : \mathcal{P}(V \cup E) \times S \times T \to \mathcal{P}(V \cup E)$	filtere Elemente mit einem Attribut heraus
$\epsilon_\epsilon^+ : \mathcal{P}(V \cup E) \times (V \cup E) \to \mathcal{P}(V \cup E)$	behalte nur ein bestimmtes Element
$\epsilon_\epsilon^- : \mathcal{P}(V \cup E) \times (V \cup E) \to \mathcal{P}(V \cup E)$	filtere ein bestimmtes Element heraus

Aus diesen Operationen lassen sich nun beliebige Traversierungen zusammensetzen. Abbildung 3.1 zeigt ein Beispiel hierfür. Das Ziel sei, wie schon in Abschnitt 1.2, die Namen aller Freunde von Sheldon Cooper zu finden. Sei v_{sc} der Knoten, der Sheldon Cooper repräsentiert und

$$f : \mathcal{P}(V) \to \mathcal{P}(T) \qquad \text{mit} \qquad f(v) = p(\, v_{in}(\, \epsilon_l^+(\, e_{out}(v), \text{Freund}\,)\,), \text{Name}\,)$$

dann ist $f(v_{sc})$ die Menge der Namen aller gesuchten Freunde. Die Funktion f bestimmt zunächst die von v_{sc} ausgehenden Kanten, sucht anschließend diejenigen mit dem Label Freund heraus, findet deren Zielknoten und gibt die Attribute mit der Bezeichnung Name dieser Knoten zurück.

Ein Traversierungsvorgang lässt sich im Allgemeinen beliebig aus derartigen Einzeloperationen zusammensetzen, insbesondere ist er nicht auf Verkettungen fester Länge beschränkt. Im Grunde ist jede Verkettung möglich, die sich in irgendeiner Form als Algorithmus beschreiben lässt. Ein bekanntes Beispiel für einen Traversierungsvorgang variabler Länge stellt Dijkstras Algorithmus zur Bestimmung kürzester Pfade dar. In welchem Maße ein Anwender Kontrolle über die einzelnen Operationen eines Traversierungsvorgangs hat, hängt natürlich vom jeweiligen Datenbanksystem ab.

3.2 Traversierungsschema

Die Angabe der einzelnen Traversierungsschritte genügt allerdings noch nicht, um einen Traversierungsvorgang vollständig zu beschreiben. Die vorgestellten Funktionen wurden über Mengen definiert, deren Elemente die jeweils gleichen Operationen durchlaufen. In der Praxis können diese Operationen aber nicht für alle Elemente parallel ausgeführt werden. Es ist also notwendig, ein Schema zur Ausführung der einzelnen Traversierungsschritte festzulegen. Im Wesentlichen gibt es drei gebräuchliche Verfahren.

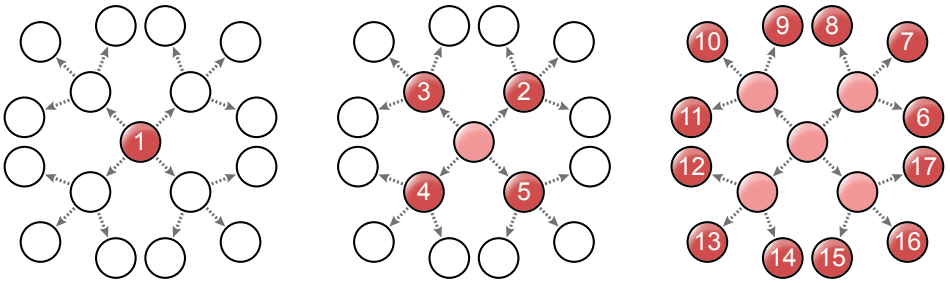


Abbildung 3.2: Bei einer Breitensuche werden zunächst alle Nachbarknoten durchsucht.

Die Breitensuche, dargestellt in Abbildung 3.2, beginnt an einem vorgegebenem Startknoten und besucht zunächst dessen Nachbarknoten. Anschließend wird dieser Vorgang für die jeweiligen Nachbarknoten wiederholt, dann für deren Nachbarknoten und so weiter. Es werden also zunächst alle Knoten einer bestimmten Tiefe durchlaufen, bevor die Suchtiefe erhöht wird. Auch werden Knoten nur dann besucht, wenn diese nicht zuvor schon einmal durchlaufen wurden. Der Nachteil der Breitensuche ist allerdings ihr recht hoher Speicherverbrauch.

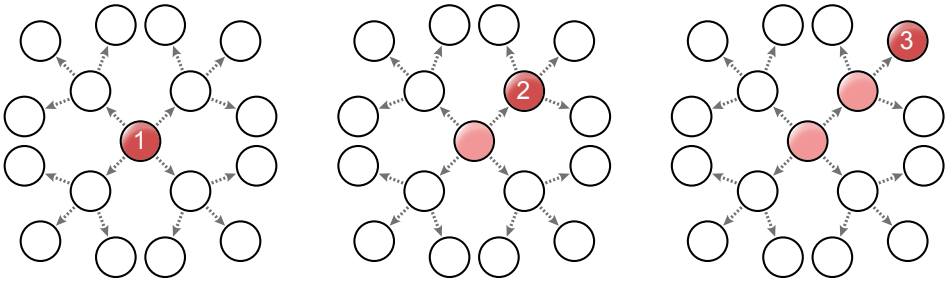


Abbildung 3.3: Bei einer Tiefensuche werden die nächsttieferen Knoten bevorzugt.

Im Gegensatz zur Breitensuche werden bei der Tiefensuche, wie in Abbildung 3.3 angedeutet, vorrangig die nächsttieferen Knoten untersucht, bevor die weiteren Nachbarn eines Knotens betrachtet werden. Dieses Verfahren ist in der Regel schneller und speichereffizienter als die Breitensuche, liefert jedoch bei bestimmten Problemstellungen nicht immer eine korrekte Lösung. Sucht man beispielsweise den kürzesten Pfad zwischen zwei

Knoten, so wird eine Tiefensuche immer den ersten gefundenen Pfad zurückgeben, dieser muss jedoch nicht minimal sein. Eine Lösung für dieses Problem stellt die iterative Tiefensuche dar, welche die Vorzüge von Breiten- und Tiefensuche weitestgehend in sich vereint.

Allerdings können einige Datensätze auch derart groß werden, dass die Untersuchung des gesamten Graphen nicht länger praktikabel ist. In solchen Fällen können randomisierte Suchverfahren verwendet werden. Diese weisen natürlich, abhängig vom jeweiligen Anwendungsfall, eine gewisse Fehlerwahrscheinlichkeit beziehungsweise Unvollständigkeit auf, erlauben aber eine schnelle Antwortzeit. Randomisierte Verfahren eignen sich vor allem für Anwendungen, in denen kein absolut korrektes Ergebnis notwendig ist, wie etwa bei Empfehlungssystemen im E-Commerce-Bereich.

4 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass Graphdatenbanken eine leistungsfähige Alternative darstellen, um dynamische, vernetzte und semistrukturierte Datensätze zu verarbeiten. Insbesondere lassen sich viele Anwendungsgebiete über Graphen repräsentieren und eine ganze Reihe von Problemen mit bekannten Graphalgorithmen lösen. In den vergangenen Kapiteln wurden die grundlegenden Aspekte von Graphdatenbanken vorgestellt. Dazu zählen Property-Graphen, die zur Speicherung verwendeten Datenstrukturen, aber auch die Beschreibung von Datenbanktraversierungen. Sicherlich ließen sich noch weitere Themen behandeln, etwa die Skalierung oder Partitionierung von Graphdatenbanken, jedoch sollen die hier präsentierten Inhalte als Einführung genügen.

Literaturverzeichnis

[1] Angles, Renzo ; Gutierrez, Claudio: Survey of graph database models. In: *ACM Compututing Surveys* 40 (2008), February

[2] Edlich, Stefan ; Friedland, Achim ; Hampe, Jens ; Brauer, Benjamin: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Hanser Fachbuchverlag, 2010

[3] Iordanov, Borislav: HyperGraphDB: A Generalized Graph Database. In: *Proceedings of the 2010 International Conference on Web-age Information Management*, Springer-Verlag, 2010

[4] Rodriguez, Marko A.: Mapping Semantic Networks to Undirected Networks. In: *International Journal of Applied Mathematics and Computer Sciences* (2008), Februar

[5] Rodriguez, Marko A. ; Neubauer, Peter: Constructions from dots and lines. In: *Bulletin of the American Society for Information Science and Technology* 36 (2010), September

[6] Rodriguez, Marko A. ; Neubauer, Peter: The Graph Traversal Pattern. In: *CoRR* (2010), April