Humboldt Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Diplomarbeit

Entwicklung einer Testumgebung für ein Peer Data Management System

Tobias Hübner

31. Januar 2006

betreut durch
Prof. Dr. Felix Naumann
Dipl.-Ing. Armin Roth

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die klassische Integration verschiedener heterogener Datenquellen bringt viele Probleme mit sich. Es ist ein globales Schema zu erstellen, welches die benötigte Semantik aller zu integrierenden Quellen erfasst, was oftmals schwierige und langwierige Einigungsprozesse erforderlich macht. Dies widerspricht dem Wunsch vorhandene Informationssysteme mit ähnlicher Semantik dynamisch integrieren zu können. Peer Data Management Systeme (PDMS) stellen einen erweiterten Ansatz dar. Es lassen sich bereits existierende Informationssysteme ad hoc vernetzen, beispielsweise bei Reorganisation oder Zusammenschluss mehrerer Firmen. Die Nutzer müssen nur das eigene Datenschema ihrer Zweigstelle kennen und können dennoch über alle Daten des gesamten PDMS verfügen. Ein PDMS besteht aus einer Menge autonomer Quellen, so genannten Peers, und benötigt keine globale integrierende Komponente. Peers lassen sich durch Mappings mit semantisch ähnlichen Peers verknüpfen. Auf diese Weise kann das Netzwerk durch Hinzufügen weiterer Peers und Mappings flexibel erweitert werden. Der Benutzer kann Anfragen an den für ihn verfügbaren Peer stellen. Der Peer kann die Anfrage aus seinen eigenen Daten beantworten, hat aber zusätzlich die Möglichkeit, die Anfrage umzuformulieren und die benachbarten, über Mappings angebundenen, Peers anzufragen. Auf diese Weise entstehen zahlreiche Einzelanfragen, von denen der Benutzer keine Kenntnis besitzt.
Ein PDMS hat leider nicht nur Vorteile. Durch die Verwendung von Mappings sinkt im Allgemeinen die Datenqualität. Des Weiteren wird zur Verkürzung der Antwortzeiten und Steigerung der Effizienz oft nur eine Teilmenge der verfügbaren Mappings verwendet, auf Kosten der Vollständigkeit des Anfrageergebnisses.

In [14] wurde ein PDMS namens *System P* implementiert, welches sich auf mehrere Rechner verteilen lässt. Es ist möglich, beliebig viele Peers zu erzeugen, mit diesen eine PDMS-Instanz zu bilden und Anfragen an einzelne Peers zu stellen. Somit können verschiedene Anfragestrategien getestet und optimiert werden. Das manuelle Konstruieren von Peer-Netzen mit über 100 heterogenen Peers ist jedoch sehr schwierig und zeitaufwendig. Neben verschiedenen Peer-Schemata müssen Peer-Mappings gebildet und Daten für die lokalen Quellen der Peers bereitgestellt werden.

Das Ziel dieser Diplomarbeit ist die Entwicklung einer Testumgebung für das *System P*. Sie soll das parametrisierte Generieren von Peer-Schemata und Peer-Mappings sowie die automatisierte Verteilung von Daten auf Peers ermöglichen. Des Weiteren sollen Peer-Netze nach einer vorgegebenen Topologie (zum Beispiel Kette, Kreis, Baum) erzeugbar sein und in einer Oberfläche dargestellt werden können. Dazu wird eine mögliche Schrittfolge zur Erstellung eines PDMS vorgestellt. Diese Schritte werden in ihrer Reihenfolge einzeln erarbeitet und bilden die Struktur dieser Arbeit.

Die Arbeit gliedert sich nach der Einleitung in neun weitere Kapitel. Das Kapitel 2 beschreibt, was ein Peer Data Management System ist, welche Vor- und Nachteile es gegenüber klassischen integrierten Informationssystemen hat und aus welchen Komponenten

es sich zusammensetzt. In Kapitel 3 werden die Grundfunktionen der entwickelten Testumgebung erläutert und die einzelnen Phasen der PDMS-Erzeugung in der gewählten Ablaufreihenfolge vorgestellt. In Kapitel 4 wird beschrieben, wie aus einem gegebenen relationalen Schema mit den Ansätzen der *Teilmengenbildung*, *Normalisierung* und *Denormalisierung* neue Schemata für die Verteilung auf Peers gewonnen werden. Das nachfolgende Kapitel 5 behandelt die Generierung von Schema-Mappings. Dazu werden die Begriffe *Query containment* und *Containment mapping* auf Anfragen an unterschiedliche Schemata übertragen und der Begriff des *schwachen Containment* eingeführt. Dieser ermöglicht das Modellieren von Projektionen in Peer-Mappings. In Kapitel 6 wird auf die Erzeugung von Peer-Graphen, welche die Topologie eines PDMS bestimmen, eingegangen. Grundlage hierfür sind die zuvor generierten Schemata und Mappings, die angeben, wie Knoten und Kanten gewählt werden können. Neben der Erzeugung von einfachen Graphen wie Kette, Kreis und Baum wird die Erzeugung von komplexeren Zufallsgraphen mit Hilfe von Markov-Ketten behandelt. Das Kapitel 7 beschreibt, wie aus Schemata, Mappings und Peer-Graph ein PDMS zusammengesetzt wird. Zusätzlich wird erläutert, wie die existierenden Peer-Mappings um Projektionen und Selektionen erweitert werden. Die Erzeugung von Extensionen für die lokalen Quellen der Peers ist ebenfalls Bestandteil dieses Kapitels. Das Kapitel 8 stellt die Java-Implementierung der Testumgebung vor und gibt Hinweise, an welchen Stellen diese sinnvoll erweitert werden kann. Des Weiteren wird vorgestellt, wie die Schnittstellen des *System P* zur Übertragung eines generierten PDMS auf konkrete Peers verwendet werden. Im vorletzten Kapitel 9 werden einige Experimente zu den vorgestellten Erzeugungsphasen durchgeführt. Diese umfassen die Untersuchung der Schema- und Mapping-Erzeugung, der Graph-Erzeugung mit Hilfe von Markov-Ketten und eine Laufzeitbetrachtung. Im letzten Kapitel 10 werden zukünftige Erweiterungen für die Testumgebung vorgeschlagen.

Danksagung

An dieser Stelle möchte ich Prof. Dr. Felix Naumann und Dipl.-Ing. Armin Roth für die Bereitstellung dieses interessanten Themas der Diplomarbeit und die sehr gute Betreuung danken. Bedanken möchte ich mich auch bei Dipl.-Inf. Michael Behrisch für wesentliche Impulse zur Graphentheorie und weitere Anregungen. Weiterhin bedanke ich mich bei Martin Schweigert für die gute Zusammenarbeit bei der Entwicklung der Schnittstellen unserer beiden Diplomarbeiten.

2 Peer Data Management Systeme

Peer Data Management Systeme (PDMS) besitzen eine dezentrale Architektur und bestehen aus einer Menge von Peers. Jeder Peer bildet ein autonomes, heterogenes, eventuell selbst wiederum integriertes Informationssystem mit einem Peer-Schema. Zusammenhänge zwischen Peer-Schemata werden mit Hilfe von Peer-Mappings modelliert. Nutzer eines PDMS können Anfragen gegen das Peer-Schema eines Peers stellen. Dieser Peer beantwortet die Anfrage anhand seiner lokal gespeicherten Daten. Zusätzlich kann er Peer-Mappings verwenden, um so benachbarte Peers in die Anfragebearbeitung einzubeziehen. Die angefragten Peers beantworten die an sie gestellte Anfrage nach demselben Muster. In Abbildung 2.1 sind die Komponenten eines PDMS dargestellt.

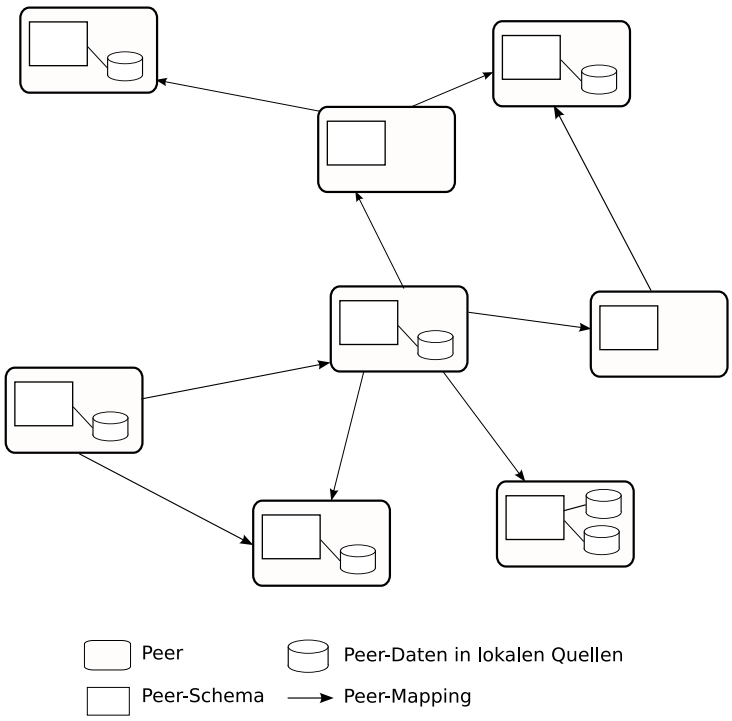


Abbildung 2.1: Ein Peer Data Management System

Dieses Kapitel wurde von Martin Schweigert und Tobias Hübner gemeinsam im Rahmen der Diplomarbeiten „Entwurf eines Peer Data Management Systems mit Steuerungs- und Simulationskomponente“ (siehe [14]) und „Entwicklung einer Testumgebung für ein Peer Data Management System“ verfasst. In beiden Arbeiten wird ein Peer Data Management System namens *System P* mit zugehöriger Testumgebung implementiert und vorgestellt.

Im Folgenden wird ein allgemeiner Überblick über die Vor- und Nachteile eines PDMS gegenüber herkömmlichen integrierenden Systemen gegeben und Unterschiede zu typischen P2P-Tauschbörsen herausgestellt. Zusätzlich werden die Komponenten der Peers

eines PDMS un der Aufbau von Mappings beschrieben. Abschließend wird die verwendete Datalog Notation für Anfragen an Peers vorgestellt.

2.1 Vor- und Nachteile eines PDMS

Durch den Verzicht auf ein globales Schema sind Peer Data Management Systeme gegenüber integrierten Informationssystemen einfach zu erweitern. Es muss keine Einigung auf eine Semantik für alle zur Verfügung stehenden Quellen erfolgen. Mappings werden nur zwischen Paaren von Peers angelegt. Ein neuer Peer kann in ein bereits bestehendes PDMS durch ein oder mehrere Mappings zu Peers mit ähnlichen Peer-Schemata einbezogen werden. Dadurch ist ein PDMS sehr flexibel und nahezu beliebig erweiterbar. Nutzer eines PDMS können Anfragen gegen das ihnen bekannte Peer-Schema stellen. Dennoch erhalten sie durch Nutzung von Peer-Mappings relevante Antworten weiterer Peers.

Durch die verteilte Architektur eines PDMS gibt es keine Komponente, deren Versagen zum Ausfall des kompletten PDMS führt. Selbst nach dem Ausfall einzelner Peers können funktionierende Peers weiterhin angefragt und bestehende Peer-Mappings durch diese genutzt werden. Im Gegensatz dazu führt in einem integrierten Informationssystem der Ausfall der integrierenden Komponente in der Regel zum Ausfall des Gesamtsystems.

Die Nutzung vieler Peer-Mappings zur vollständigen Anfragebearbeitung in einem PDMS führt zu zahlreichen weiteren Einzelanfragen (Peer-Anfragen). Dadurch wird die Anfragebearbeitung schnell ineffizient. Die Tatsache, dass ein Nutzer eines PDMS nicht alle Quellen kennen muss, diese jedoch Daten für seine Anfragen liefern können, ist zum einen positiv. Im Falle zweifelhafter Quellen kann diese jedoch auch ein Nachteil eines PDMS sein. Ebenso kann die kumulative Nutzung von Peer-Mappings zu einem Verlust von Semantik und einem Verlust an Informationsqualität führen. Das folgende Beispiel verdeutlicht einen Fall, in dem nicht alle möglichen Tupel einer Anfrage in einem PDMS gefunden werden.

Beispiel 2.1 *Verpasste Tupel bei der Anfragebeantwortung in einem PDMS*

In Abbildung 2.2 sind drei über Mappings verbundene Peers, in diesem Fall Bibliotheken, dargestellt. Sie besitzen Daten über ihre Bücher. Während Bibliothek A und Bibliothek C Daten zu Büchern unterschiedlicher Verlage besitzen, verfügt Bibliothek B nur über Bücher des Verlags „Addison-Wesley“. Wird Bibliothek A nach dem Buch „PC Intern 4“ angefragt, so kann dieser Peer aufgrund seines Mappings die Anfrage an Bibliothek B weiterleiten. Da Bibliothek B ein Mapping zu Bibliothek C besitzt, wird diese angefragt. Letzteres Mapping besitzt einen Filter, der die Übertragung von Büchern anderer Verlage als „Addison-Wesley“ unterbindet. Somit wird der gesuchte Datensatz von Bibliothek C nicht an Bibliothek A geliefert.

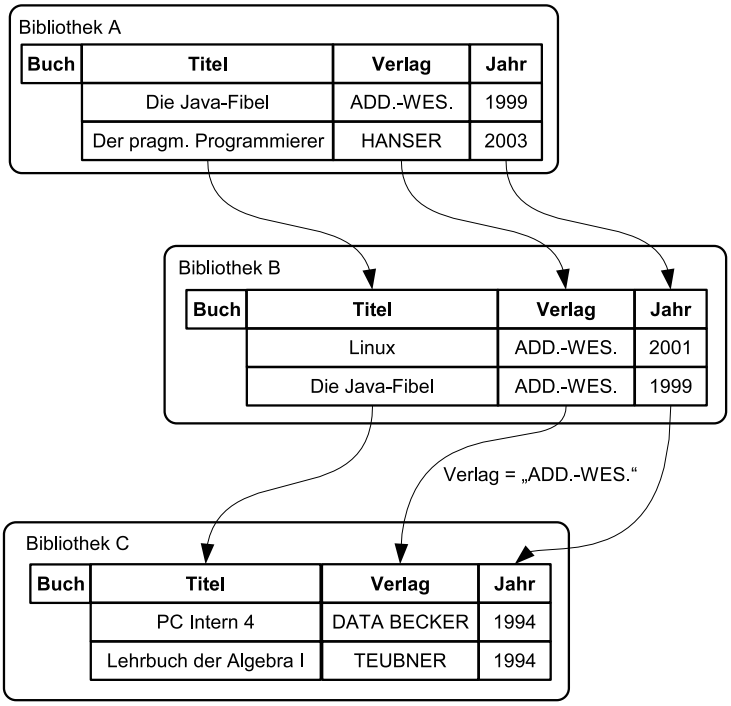


Abbildung 2.2: Informationsverlust durch Mappings

Unterschied zu P2P-Tauschbörsen

Obwohl Peer Data Management Systeme vergleichbar mit P2P-Tauschbörsen sind, gibt es einige entscheidende Unterschiede. Alle Peers einer Tauschbörse verfügen im Allgemeinen über dasselbe Schema. Dieses Schema besteht in der Regel aus einer einzigen Relation. In einem PDMS hingegen können Peers heterogene, komplexe Schemata besitzen. Ebenso sind die Möglichkeiten der Anfragesprache in P2P-Tauschbörsen meist sehr eingeschränkt. Im Unterschied dazu sind in einem PDMS komplexe Anfragesprachen wie zum Beispiel SQL möglich. Auch bestehen Unterschiede in der Datenübertragung zwischen Peers einer Tauschbörse und eines PDMS. Geschieht die Übertragung von Daten in einer Tauschbörse direkt zwischen zwei Peers, werden Daten in einem PDMS entlang der Mapping-Pfade übertragen. Dies kann die Effizienz des Datentransfers stark beeinträchtigen, ermöglicht jedoch den Datenaustausch zwischen Peers, die nicht direkt miteinander kommunizieren können.

Peer Data Management Systeme können im Krisenfall zur schnellen Vernetzung von öffentlichen Einrichtungen, wie Notrufzentralen, Krankenhäusern oder Feuerwehrstationen eingesetzt werden. Ein weiterer Anwendungsfall ist die Integration verschiedener Informationssysteme bei Zusammenschluss mehrerer Firmen. Prinzipiell ist die dynamische Vernetzung vieler heterogener Systeme ein Einsatzgebiet eines PDMS.

2.2 Peer

Jeder Peer eines PDMS kann mehrere Rollen einnehmen. Er kann als Datenquelle dienen, die Funktion eines Wrappers übernehmen, indem er Sichten auf seine lokalen Daten zur Verfügung stellt oder durch die Verwendung seiner *Peer-Mappings* $\mathcal{M}_P$ als Mediator agieren. Jeder Peer kann dabei selbst ein integriertes Informationssystem sein, das auf mehrere *lokale Datenquellen* $\mathcal{L}$ zugreift und diese mit Hilfe *lokaler Mappings* $\mathcal{M}_L$ in einem integrierten *Peer-Schema* S anbietet. Benutzer und weitere Peers des PDMS können Anfragen gegen dieses Peer-Schema stellen. Formal kann ein Peer somit durch das Tupel $P = (S, \mathcal{L}, \mathcal{M}_L, \mathcal{M}_P)$ repräsentiert werden. Der Aufbau eines Peers ist in Abbildung 2.3 dargestellt.

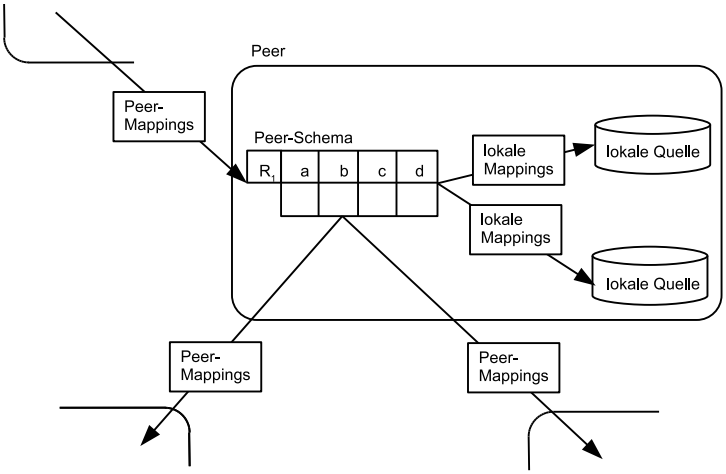


Abbildung 2.3: Aufbau eines Peers

2.3 Mappings

Über Mappings lassen sich zwei (unterschiedliche) Schemata in Beziehung setzen. In einem PDMS bilden lokale Mappings die Verknüpfung zwischen Schemata lokaler Quellen und dem Peer-Schema. Peer-Mappings stellen den Zusammenhang zwischen Schemata verschiedener Peers her.

Lokale Mappings haben die Form $Q_L(\mathcal{L}) \subseteq Q_S(S)$, wobei Q_L und Q_S konjunktive Anfragen darstellen. Die Anfrage $Q_L(\mathcal{L})$ referenziert Relationen (auch Teilziele genannt) aus den lokalen Quellen und $Q_S(S)$ Relationen aus dem Peer-Schema. Peer-Mappings haben die Form $Q_1(\mathcal{P}_1) \subseteq Q_2(\mathcal{P}_2)$. $\mathcal{P}_1$ und $\mathcal{P}_2$ sind Mengen von Peers, deren Schemata durch das Mapping aufeinander abgebildet werden. Die Anfragen Q_1 und Q_2 dürfen alle Relationen aus den Peer-Schemata referenzieren. Intuitiv bedeutet ein Peer-Mapping, dass Q_1 immer eine Teilmenge der von Q_2 berechneten Tupel zurückliefert. Daher darf eine gegebene Anfrage $Q_2(\mathcal{P}_2)$ nach $Q_1(\mathcal{P}_1)$ umformuliert werden. Eine Umformulierung in die andere

Richtung ist nicht möglich. Mappings sind demnach gerichtet. Aufgrund dieser Tatsache werden $Q_1(\mathcal{P}_1)$ als Kopf und $Q_2(\mathcal{P}_2)$ als Rumpf bezeichnet.
Der allgemeine Fall, in dem auf beiden Seiten des Mappings mehrere Relationen angefragt werden, wird Global-Local-As-View-Ansatz (GLaV) genannt. Eine besondere Bedeutung kommt folgenden Spezialfällen zu:

- $Q_1(\mathcal{P}_1) \subseteq P_2.R$ wird als Global-As-View-Ansatz (GaV) bezeichnet. Dabei besteht der Rumpf des Mappings nur aus einer Relation.
- $P.R_1 \subseteq Q_2(\mathcal{P}_2)$ wird Local-As-View-Ansatz (LaV) genannt. Der Kopf enthält bei diesem Typ Mapping nur eine Relation.

Im *System P* werden bisher ausschließlich GaV- und LaV-Mappings unterstützt, jedoch ist eine Erweiterung um GLaV-Mappings durch Anpassung des Anfrageplaners möglich. Bei einer Anfragebearbeitung entscheidet jeder Peer als autonomes System, welche Mappings er zur Beantwortung von Anfragen nutzt.

Mappings können unvollständig sein und müssen nicht alle Attribute der Peer-Schemata aufeinander abbilden. Derartige Projektionen treten häufig dann auf, wenn zwei Peer-Schemata zwar semantisch ähnlich sind, aber einen unterschiedlichen Detailgrad an Informationen aufweisen. Des Weiteren dürfen Mappings zur Filterung von Daten Selektionsprädikate besitzen. Dadurch kann zusätzlich implizites Wissen über ein Peer-Schema ausgedrückt werden. In den folgenden Beispielen zu Projektionen und Selektionen in Mappings wird die Datalog-Notation verwendet, welche im nachfolgenden Abschnitt über Anfragen erklärt wird.

Beispiel 2.2 *Projektionen in Peer-Mappings*

Es seien Peer1 und Peer2 gegeben mit den Relationen Buch, Autor *und* Verlag. *Während Peer1 in der Buch-Relation nur Informationen zu Autor, Verlag und Buchtitel speichert, besitzt Peer2 in der Buch-Relation das Attribut* Jahr. *Das Mapping*

$$\mathsf{Peer2.Buch}(\mathsf{ISBN}, \mathsf{Titel}, \mathsf{Verlag}, \mathsf{Jahr}) :- \mathsf{Peer1.Buch}(\mathsf{ISBN}, \mathsf{Titel}, \mathsf{Verlag})$$

enthält eine Projektion für das Attribut Jahr. *Da Peer2 im Mapping-Kopf und Peer1 im Mapping-Rumpf auftreten, steht das Mapping Peer1 zur Weiterleitung von Anfragen an Peer2 zu Verfügung. Peer1 kann die Werte für das* Jahr*-Attribut im eigenen Schema nicht aufnehmen und muss diese aus den Anfrageergebnissen von Peer2 streichen.*

Eine weitere Form der Projektion ist in dem Mapping

$$\mathsf{Peer1.Buch}(\mathsf{ISBN}, \mathsf{Titel}, \mathsf{Verlag}) :- \mathsf{Peer2.Buch}(\mathsf{ISBN}, \mathsf{Titel}, \mathsf{Verlag}, \mathsf{Jahr})$$

zu finden. Peer2 kann aufgrund dieses Mappings die Buch*-Relation von Peer1 anfragen. Die Relation enthält nicht das gewünschte Attribut* Jahr *des Peer-Schemas von Peer2. Somit müssen die Ergebnistupel von Peer1 um* NULL*-Werte ergänzt werden, damit diese zur Datenstruktur von Peer2 passen.*

Beispiel 2.3 *Selektionen in Peer-Mappings*

Durch Einfügen von Vergleichsprädikaten können Mappings als Filter eingesetzt werden. Das Mapping

$$\text{Peer4.Buch(ISBN, Titel, Jahr)} :- \text{Peer3.Buch(ISBN, Titel, Jahr)}, \text{Jahr} > 2001$$

enthält neben den Teilzielen aus den gegebenen Peer-Schemata das Vergleichsprädikat Jahr $>$ 2001*. Auf diese Weise lässt sich ausdrücken, dass Peer4 nur in der Lage ist, neue Bücher zu liefern. Alternativ ist ein vertragliches Abkommen zwischen den Betreibern von Peer3 und Peer4 denkbar, das nur die Lieferung von Daten zu Büchern mit* Jahr $>$ 2001 *umfasst.*

Wird Peer3 nach Büchern mit Jahr $<$ 1980 *gefragt, so kann dieser aufgrund der sich widersprechenden Prädikate* Jahr $<$ 1980 *und* Jahr $>$ 2001 *vom Gebrauch dieses Mappings absehen.*

2.4 Anfragen

Die Anfragesprache eines PDMS ist frei wählbar und kann prinzipiell von Peer zu Peer verschieden sein. Im *System P* werden Anfragen in Datalog formuliert. Gründe für die Wahl dieser Anfragesprache sind die leichte Handhabung und die einfache Integration einer bereits existierenden Komponente zur Anfrageplanung von Armin Roth.

Eine Datalog-Anfrage besteht aus einem Kopf und einem Rumpf. Der Kopf beschreibt die Struktur der Ergebnismenge, während der Rumpf sich aus einer Liste von Teilzielen (Relationen des Peer-Schemas) und Selektionsprädikaten zusammensetzt.

Das folgende Beispiel erläutert die verwendete Notation im *System P*.

Beispiel 2.4 *Anfrage an einen Peer*

Das Peer-Schema von Peer1 enthält drei Relationen Buch*,* Autor *und* Verlag*. Eine Anfrage soll die Titel aller Bücher aus dem Verlag „Pearson Studium“ und Vor- und Nachnamen der jeweiligen Autoren liefern. Diese Anfrage lautet in Datalog-Notation:*

```
q(Titel, Vorname, Name) :-  Peer1.Autor(AID, Vorname, Name),
                            Peer1.Buch(BID, AID, VID, Titel),
                            Peer1.Verlag(VID, VName), VName = 'Pearson Studium'.
```

Das Zeichen $:-$ *trennt den Kopf vom Rumpf der Anfrage. Der Rumpf enthält die Teilziele* Peer1.Autor(AID, Vorname, Name)*,* Peer1.Buch(BID, AID, VID, Titel) *und* Peer1.Verlag(VID, VName) *sowie das Selektionsprädikat* VName = 'Pearson Studium'*. Durch mehrmalige Verwendung von Variablennamen im Rumpf der Anfrage, werden Joins ausgedrückt. In diesem Fall finden Joins zwischen den Relationen* Autor *und* Buch *auf dem Attribut* AID *sowie zwischen* Buch *und* Verlag *auf dem Attribut* VID *statt.*

Während Martin Schweigert in seiner Diplomarbeit [14] die Implementierung eines PDMS und die verwendeten Strategien zur Anfragebearbeitung beschreibt, wird in den folgenden Kapiteln dieser Arbeit die Entwicklung einer Testumgebung für ein PDMS gezeigt. Nachdem ein Überblick über das Konzept und die grundlegende Funktionsweise der implementierten Anwendung gegeben wird, folgen detaillierte Beschreibungen der einzelnen Phasen für die Generierung eines PDMS. Den Abschluss bilden einige mit der Implementierung durchgeführten Experimente zur Schema- und Mapping-Generierung, Graph-Erzeugung und Laufzeit.

3 Konzept und Funktionsweise der Testumgebung

Ein großer Vorteil von Peer Data Management Systemen ist die Möglichkeit, ihnen beliebig viele neue Peers über Peer-Mappings hinzuzufügen. Im Laufe der Zeit kann ein einfaches PDMS zu einem sehr komplexen Netzwerk mit nicht-voraussehbarer Topologie wachsen. Die Aufgabe der in dieser Arbeit entwickelten Testumgebung ist es, derartige PDMS-Instanzen automatisiert erzeugen zu können. In den folgenden Abschnitten wird auf die Phasen der Erzeugung eines virtuellen PDMS eingegangen, erläutert, wie das reale PDMS aufgebaut wird und die Benutzeroberfläche kurz vorgestellt.

3.1 Erzeugung eines virtuellen PDMS

Die Algorithmen zur Erzeugung eines PDMS der hier vorgestellten Testumgebung lassen sich verschiedenen Phasen zuordnen und sind in Abbildung 3.1 schematisch dargestellt.

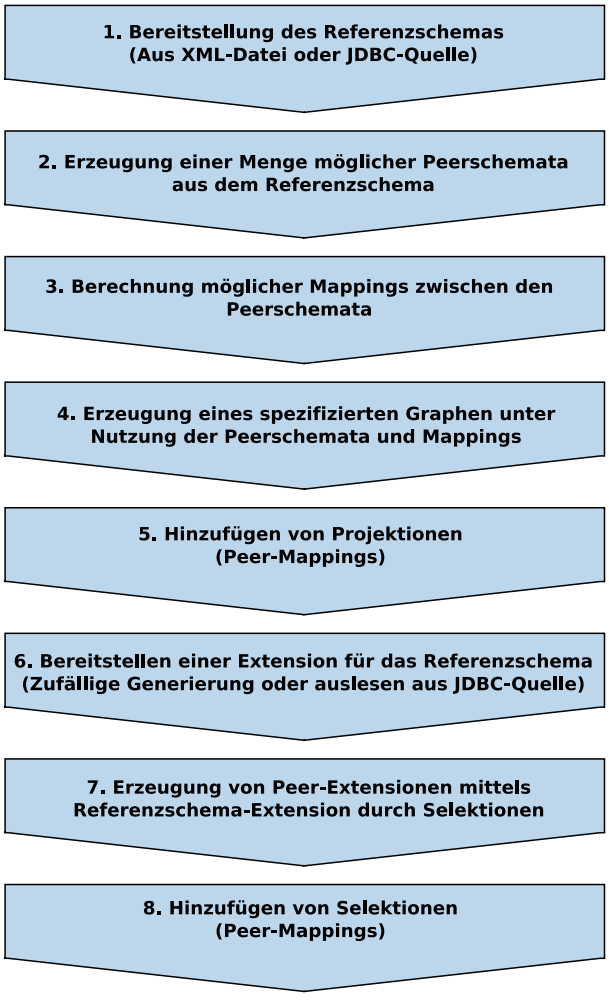


Abbildung 3.1: Schritte der PDMS-Erzeugung

Im ersten Schritt muss ein Schema zur Verfügung gestellt werden. Dies kann wahlweise aus einer XML-Datei gelesen oder aus einer JDBC-Quelle geladen werden. Aus diesem Schema, welches wir ab jetzt Referenzschema nennen wollen, werden im zweiten Schritt eine Menge Teilschemata erzeugt. Diese Menge bildet einen so genannten Schema-Pool und wird später für die Zuweisung der Peer-Schemata verwendet. Danach ermitteln wir in Schritt drei eine Menge von Mappings zwischen den erzeugen Schemata. Anhand dieser Menge steht fest, welche Peer-Mappings nach Zuteilung der Peer-Schemata möglich sind.

Im vierten Schritt wird ein Graph nach Vorgaben des Anwenders unter Nutzung eines Zufallszahlengenerators berechnet. Es stehen verschiedene Netz-Topologien bzw. Graphentypen mit einer Reihe von Parametern zur Auswahl. Die Knoten entsprechen den Peers und die Kanten den Mappings. Jedem Knoten wird ein Schema aus dem Pool als Peer-Schema zugewiesen. Beim Generieren des Graphen wird darauf geachtet, nur Kanten zwischen Knoten einzufügen, wenn ein Mapping zwischen den beteiligten Peer-Schemata im Pool existiert. Das Entnehmen der Schemata aus dem Pool erfolgt mit Zurücklegen, dass heißt dasselbe Schema kann mehrmals im Graphen als Peer-Schema auftauchen. Peer-Graphen lassen sich aufgrund des Schema-Pools und der Mappings einfach erstellen, da während des Erzeugungs-Prozesses die Information bereitsteht, welche Schemapaare für Mappings in Frage kommen und keine Schemata erzeugt werden.

Um die Schema-Heterogenität des PDMS zu steigern, werden im Schritt fünf einzelne Attribute aus den Peer-Schemata entfernt. So unterscheiden sich auch zwei Peer-Schemata voneinander, welche ursprünglich das gleiche Pool-Schema besaßen. In Folge dessen entstehen Projektionen, die bewirken, dass einzelne Peer-Mappings unvollständig sind und nicht mehr alle erforderlichen Attributwerte übertragen. In der Praxis ist dies dann der Fall, wenn ein Peer mit einem ausführlicheren Schema einen Peer mit einem weniger ausführlichen Schema anfragt. Die zurückgelieferten Tupel müssen um NULL-Werte ergänzt werden, um der Struktur des ausführlicheren Schemas zu genügen. Zusätzlich werden Projektionen durch Umbenennung einzelner Variablen in den Peer-Mappings erzeugt. Auf diese Weise können einige Mappings bestimmte Attributwerte nicht mehr übertragen, obwohl beide Schemata die passende Struktur aufweisen. Dies entspricht dem Szenario, in dem ein Peer den Werten bestimmter Attribute eines anderen Peers nicht traut und diese nicht zur Beantwortung der Anfrage nutzt.

Ab diesem Zeitpunkt stehen die Peer-Schemata fest und es können Extensionen für die Peers bereitgestellt werden. Hierzu wird eine Extension für das Referenzschema erzeugt und anschließend Teile davon auf die Peers übertragen. Die Referenz-Extension kann, wie das Referenz-Schema, aus einer JDBC-Quelle gelesen werden. Als Alternative wird auch die zufällige Generierung der Tupel angeboten. Der Vorteil dieser Methode ist, dass die Peers keine widersprüchlichen Daten aufweisen und das Problem der Konfliktbehebung nicht behandelt werden muss.

Im letzen Schritt haben die Peers bereits eine Extension erhalten, und es können Selektionsprädikate in den Peer-Mappings eingefügt werden. Damit wird modelliert, dass ein Peer nur eine Teilmenge seiner möglichen Daten an andere Peers weitergeben darf. Gründe

hierfür können unter anderem vertragliche Abkommen oder Vertrauen in die Korrektheit bestimmter Datenbereiche sein.

3.2 Übertragung in ein reales PDMS

Nach Ablauf der oben vorgestellten Phasen steht ein PDMS mit gewählter Netz-Topologie virtuell zur Verfügung. Die Testumgebung ist in der Lage, das erzeugte PDMS mit der in [14] vorgestellten Implementierung auf mehrere Rechner aufzusetzen. Sie kann, als so genannter Monitor-Peer, in einem Rechnernetz nach verfügbaren PDMS-Peers suchen und auf diesen das erzeugte System abbilden. Hierzu wird jeder gefundene reale Peer einem virtuellen Peer zugewiesen und mit dessen Schema, Mappings und Daten befüllt. In Abschnitt 8.2 wird dieser Vorgang ausführlicher beschrieben.

3.3 Benutzeroberfläche für Experimente

Wenn ein erzeugtes PDMS erfolgreich auf mehrere Rechner verteilt werden konnte, ist das System bereit für Anfragen. Für die Testumgebung wurde eine Benutzeroberfläche implementiert, mit der sich virtuelle PDMS-Instanzen erzeugen und auf reale Peers übertragen lassen. Einzelheiten zum Aufbau und zur Architektur der Oberfläche sind im Kapitel 8 zu finden.

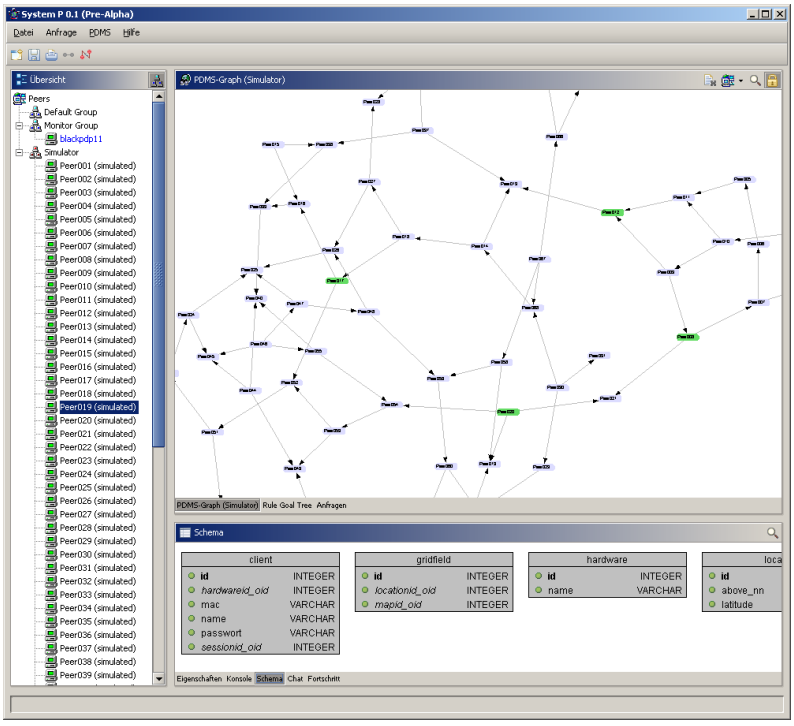


Abbildung 3.2: Benutzeroberfläche der implementierten Testumgebung

4 Erzeugung von Peer-Schemata

In einem PDMS haben die Peers im Allgemeinen unterschiedliche, unabhängig voneinander erzeugte Schemata. Die Aufgabe besteht zum einen darin, eine Menge Schemata zu erzeugen und zum anderen Mappings zwischen diesen zu finden. Der folgende Ansatz erzeugt mit verschiedenen Algorithmen, ausgehend von einem Referenzschema, einen Schema-Pool. Die im Schema-Pool enthaltenen Teilschemata können zu einem späteren Zeitpunkt zur Bildung von Peer-Mappings in Beziehung gesetzt werden, da für die Schemata Sichtdefinitionen bezüglich des Referenzschemas generiert werden (siehe Abschnitt 4.2). Die Erzeugung des Schema-Pools wird in diesem, die Generierung der Schema-Mappings im nächsten Kapitel beschrieben.

Nachfolgend wird erläutert, welche Einschränkungen wir an erzeugte Schemata stellen und wie die Sichtdefinitionen zwischen Teilschema und Referenzschema formuliert werden. Die anschließenden Abschnitte beschreiben die drei Schemaerzeugungs-Algorithmen *Teilmengenbildung*, *Normalisierung* und *Denormalisierung*. Diese konkreten Ansätze werden anhand eines beispielhaften Referenzschemas näher erläutert.

4.1 Überblick

Das Referenzschema in Abbildung 4.1 bildet dank der zwei Fremdschlüsselbeziehungen eine Zusammenhangskomponente und soll in diesem Kapitel als begleitendes Beispiel dienen. In einem Referenzschema dürfen keine Fremdschlüsselbeziehungen zwischen zwei Primärschlüsseln existieren, da die Extensions-Erzeugung, vorgestellt in Abschnitt 7.3, dadurch deutlich vereinfacht wird.

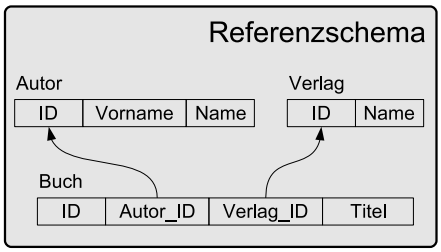


Abbildung 4.1: Ein Referenzschema

Bei der Schemaerzeugung sollen nur zusammenhängende Schemata generiert werden, da ein Peer mit n Schema-Komponenten sich durch n Peers mit jeweils einer Komponente ersetzen lässt. Das Teilschema aus Abbildung 4.2 wird beispielsweise nicht erzeugt, da kein Zusammenhang zwischen den Relationen Autor und Verlag über einen Fremdschlüssel besteht. Stattdessen werden zwei Teilschemata generiert, welche nur die Relation Autor und die Relation Verlag besitzen. Der Zusammenhang der Peer-Schemata muss über Fremdschlüsselbeziehungen sichergestellt sein, da so Relationen über Joins kombiniert werden

können und Relationen übergreifende Mappings möglich sind. Dies wird in Kapitel 5 deutlicher, wenn beschrieben wird, wie Mappings zwischen Schemata erzeugt werden.

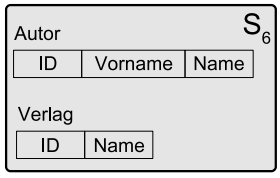


Abbildung 4.2: Ausgeschlossenes Teilschema

Neben der Forderung nach zusammenhängenden Schemata gibt es eine Begrenzung der Größe des Schema-Pools als Einschränkung. Die Algorithmen *Teilmengenbildung*, *Normalisierung* und *Denormalisierung* werden unabhängig voneinander auf das Referenzschema angewendet und nicht geschachtelt. So findet beispielsweise das Bilden von Teilschemata aus einem bereits denormalisierten Teilschema nicht statt. Gründe für dieses Vorgehen sind zum einen Schwierigkeiten bei der Generierung der Sichtdefinitionen (mehrfaches Umschreiben nötig) und zum anderen die Laufzeit des Algorithmus zur Erzeugung von Schema-Mappings (siehe Kapitel 5), welche von der Anzahl erzeugter Schemata abhängt. Eine zukünftige Erweiterung um eine Schachtelung der Algorithmen wäre jedoch denkbar.

Ein wichtiges Argument für obige Einschränkungen ist die Tatsache, dass Schemata des Pools mehrfach als Peer-Schema verwendet werden dürfen. Hat ein Schema-Pool beispielsweise eine Größe von 100 Schemata und ist ein PDMS mit 500 Peers zu erzeugen, so wird jedes Schema durchschnittlich 5 mal verwendet. In einem Nachbearbeitungsschritt (siehe Abschnitt 7.2 über Projektionen auf Seite 68) werden zugewiesene Peer-Schemata nachträglich verändert und somit die Schema-Vielfalt weiter erhöht.

4.2 Definierende Mappings

Ein wichtiger Aspekt bei der Erzeugung von Teilschemata ist die Bewahrung der Information über ihre Entstehung. Für die Erzeugung der Peer-Mappings und die Befüllung der Peers mit Daten werden bei der Schema-Erzeugung gleichzeitig auch Sichtdefinitionen (definierende Mappings) erzeugt. Sie beschreiben, wie eine Relation im generierten Schema aus dem Referenzschema gewonnen wird. Dabei bedienen wir uns der Datalog-Schreibweise.

Sei S_1 ein erzeugtes Schema und S das Referenzschema, dann könnte ein definierendes Mapping für die Relation R_1 aus S_1 wie folgt aussehen:

$$\mathsf{S_1.R_1(a,b,c,d,e)} :- \mathsf{S.R_2(a,b,c), S.R_3(c,d,e)}$$

Es handelt sich dabei um eine so genannte Datalog-Regel. Die linke Seite enthält die zu definierende Relation, die rechte Seite die verwendeten Relationen des Referenzschemas. Die kommaseparierten Buchstaben in den Klammern sind Variablen, welche die Attribute

der Relationen auf der linken und rechten Seite abbilden. Hierzu ist eine Festlegung der Attributreihenfolge in den Relationen nötig.
Werden in Relationen auf derselben Regelseite die gleichen Variablen verwendet, so bedeutet dies, dass ein Join auf dem abgebildeten Attribut stattfinden muss. In obigem Beispiel-Mapping wird beschrieben, dass die Relation $S_1.R_1$ aus einem Equi-Join der Relationen $S.R_2$ und $S.R_3$ hervorgeht. Die Join-Bedingung ist hierbei die Gleichheit des dritten Attributs der Relation $S.R_2$ mit dem ersten Attribut der Relation $S.R_3$.
Von der Datalog-Schreibweise wird im Kapitel 5 ausführlicher Gebrauch gemacht. Zum Verständnis der folgenden Abschnitte reichen jedoch obige Ausführungen aus.

4.3 Teilmengen

Sei ein relationales Datenbankschema S mit den Relationsschemata

$$S = \{R_1, R_2, \ldots, R_m\}$$

als Referenzschema gegeben. Von den $2^{|S|}$ möglichen Teilschemata der Potenzmenge $\mathfrak{P}(S)$ werden nur die Schemata generiert, welche aufgrund ihrer Fremdschlüsselbeziehungen eine Zusammenhangskomponente bilden. Die Relationen in den Teilschemata entsprechen denen im Referenzschema, besitzen demnach noch alle Attribute ihrer Ausgangsrelation. In einem späteren Schritt (siehe Abschnitt 7.2) werden aus den Relationen Attribute entfernt und die erzeugten Schemata somit weiter modifiziert. Folgende Überlegungen führten zu diesem Vorgehen:

- Die Anzahl der erzeugbaren Schemata würde unter Betrachtung der Attributteilmengen stark ansteigen. Der in Kapitel 5 vorgestellte Algorithmus zur Mapping-Suche liefe dann aufgrund seiner quadratischen Laufzeit erheblich länger.

- Eine Entfernung der Attribute, nachdem der Peer-Graph erzeugt wurde, hat den Vorteil, dass auch Peer-Schemata, die durch Normalisierung und Denormalisierung gewonnen wurden, unterschiedliche Attributmengen besitzen können. Andernfalls hätte dies in allen drei Algorithmen implementiert werden müssen.

- Bei der Erzeugung des Peer-Graphen wird mit Zurücklegen gearbeitet, ein Peer-Schema also im Allgemeinen mehrmals verwendet. Das Entfernen einiger Attribute findet jedoch nach der Graph-Erstellung statt. Somit gewinnen wir insgesamt eine höhere Anzahl unterschiedlicher Peer-Schemata.

Das Berechnen der Teilmengen wird auf naive Weise in Algorithmus 4.1 beschrieben. Die Funktion `FixForeignKeyReferences` prüft, ob im Teilschema Fremdschlüsselreferenzen zu ausgelassenen Relationen entfernt werden müssen.

Der Algorithmus 4.1 liefert für das Beispiel aus Abbildung 4.1 fünf Teilschemata S_1, S_2, S_3, S_4 und S_5, welche in Abbildung 4.3 zu sehen sind.

Input : Referenzschema $S = \{R_1, R_2, \ldots, R_n\}$
Output : Teilschemata $\mathcal{T} = \{S_1, \ldots, S_m\}$

$\mathcal{T} \leftarrow \emptyset$;
foreach $T \subset S$ **do**
  `/* Nur zusammenhängende Schemata zulassen */`
  if T *ist eine Zusammenhangskomponente* $\wedge\ T \neq \emptyset$ **then**
    `/* Fremdschlüsselintegrität herstellen */`
    `FixForeignKeyReferences` (T);
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$;
  end
end
return $\mathcal{T}$;

Algorithmus 4.1 : Teilschemata durch Teilmengenbildung berechnen

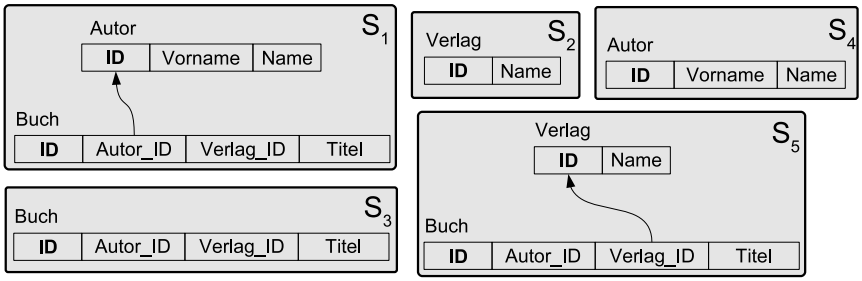


Abbildung 4.3: Durch den Teilmengenalgorithmus erzeugte Schemata

Für jedes erzeugte Teilschema T werden zeitgleich passende definierende Mappings angelegt. In diesem Fall sind die definierenden Mappings der Relationen einfache Datalog-Regeln, in denen jede Relation aus dem Teilschema einer Relation des Referenzschemas zugeordnet wird.

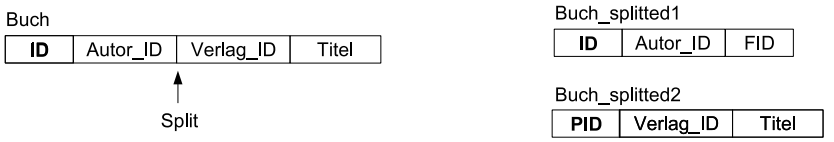
$$
\begin{aligned}
\mathsf{S_1.Autor(x,y,z)} &:\!- \mathsf{S.Autor(x,y,z)} \\
\mathsf{S_1.Buch(x,y,z,u)} &:\!- \mathsf{S.Buch(x,y,z,u)} \\
\mathsf{S_2.Verlag(x,y)} &:\!- \mathsf{S.Verlag(x,y)} \\
\mathsf{S_3.Buch(x,y,z,u)} &:\!- \mathsf{S.Buch(x,y,z,u)} \\
\mathsf{S_4.Autor(x,y,z)} &:\!- \mathsf{S.Autor(x,y,z)} \\
\mathsf{S_5.Verlag(x,y)} &:\!- \mathsf{S.Verlag(x,y)} \\
\mathsf{S_5.Buch(x,y,z,u)} &:\!- \mathsf{S.Buch(x,y,z,u)}
\end{aligned}
$$

Die in diesem Abschnitt erzeugten Teilschemata werden dem Schema-Pool hinzugefügt und stehen somit für die spätere Erzeugung von Mappings zur Verfügung. Im nächsten Abschnitt wird vorgestellt, wie weitere Schemata erzeugt werden.

4.4 Normalisierung

Unter Normalisierung versteht man für gewöhnlich die Analyse gegebener Relationsschemata auf der Grundlage ihrer funktionalen Abhängigkeiten und Primärschlüssel, um ein Schema mit gewünschten Eigenschaften zu erzeugen. Dabei werden nicht zufriedenen stellende Relationsschemata in kleinere Relationsschemata zerlegt [16].

Für die Normalisierung ist die Angabe von funktionalen Abhängigkeiten nötig, da diese sich nicht automatisch aus einem Schema oder einer Extension ableiten lassen. Aus diesem Grund wurde eine vereinfachte Normalisierung implementiert, die beliebige Relationen teilt. Das Zerteilen von Relationen mit weniger als drei Attributen ist nicht sinnvoll und wird daher unterbunden.



(a) Ausgangsrelation (b) Zwei neue Relationen

Abbildung 4.4: Teilen einer Relation

Wird aus einem Schema eine Relation entfernt und durch zwei neue ersetzt, so muss auf etwaige Fremdschlüsselbeziehungen geachtet werden. Existiert im Beispiel aus Abbildung 4.4a eine Relation mit Fremdschlüssel zu Buch.ID, so muss dieser auf Buch_splitted1.ID in Abbildung 4.4b umgesetzt werden. Unverändert darf dieser nicht bleiben, da die Relation Buch im neuen Schema nicht mehr existiert.

Die erste der neuen Relationen erhält ein zusätzliches Attribut und die zweite Relation ein neues Schlüsselattribut, einen so genannten Surrogatschlüssel (vgl. [16]). Dieser wird Primärschlüssel und von dem neuen Attribut der ersten Relation als Fremdschlüssel referenziert. Auf diese Weise wird sichergestellt, dass das resultierende Schema nur eine Zusammenhangskomponente darstellt. In Abbildung 4.4b ist Buch_splitted2.PID das neu eingefügte Surrogatschlüsselattribut und Buch_splitted1.FID das neue Attribut zur Verbindung der Relationen über einen Fremdschlüssel. Der Ablauf ist in Algorithmus 4.2 dargestellt.

Die Funktion `CalcComponents` errechnet die Zusammenhangskomponenten des Schemagraphen. Jede Komponente C ist eine Menge von Relationen, aus welchen in der nächsten Schleife jeweils alle Teilmengen T betrachtet werden. Für jede dieser Teilmengen wird ein neues Teilschema L angelegt, das alle Relationen der Komponente enthält, abzüglich derer, die Element der berechneten Teilmenge sind. Die Relationen in der Teilmenge T sind für die Normalisierung bestimmt, werden entfernt und durch jeweils zwei gesplittete Relationen ersetzt.

Input : Referenzschema $S = \{R_1, R_2, \ldots, R_n\}$
Output : Teilschemata $\mathcal{T} = \{S_1, \ldots, S_m\}$

$\mathcal{T} \leftarrow \emptyset$;
$\mathcal{C} \leftarrow$ `CalcComponents(`S`)`;
foreach *Komponente* $C \in \mathcal{C}$ **do**
  foreach $T \subseteq C$ **do**
    `/* Komponente kopieren und aus ihr neues Schema erzeugen */`
    $L \leftarrow$ `CopySchema(`C`)`;
    foreach *Relation* $R \in T$ *mit mehr als 2 Attributen* **do**
      `/* Teile die Relation in der Mitte */`
      `SplitRelation(`R, R_1, R_2`)`;
      `/* Ersetze im neuen Schema die Relation */`
      `/* durch die geteilten */`
      $L \leftarrow L \setminus \{R\} \cup \{R_1, R_2\}$;
    end
    `/* Fremdschlüsselreferenzen löschen bzw. umsetzen */`
    `FixForeignKeyReferences` (L);
    `/* Merke im Ergebnis */`
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{L\}$;
  end
end
return $\mathcal{T}$;

Algorithmus 4.2 : Einfache Normalisierung (Splitting)

Auch bei der Normalisierung entsteht für jedes erzeugte Teilschema eine Menge von definierenden Mappings. Jede Teilrelation, die aus einem Normalisierungsvorgang resultiert, bekommt dabei in ihrer Datalog-Regel nur die passenden Attribute der Ausgangsrelation zugewiesen. Es handelt sich dabei um eine einfache Projektion. Die für das Referenzschema aus Abbildung 4.1 erzeugten Teilschemata sind in Abbildung 4.5 dargestellt. Es besteht aus einer einzigen Komponente und hat nur zwei für das Normalisieren in Frage kommenden Relationen Autor und Buch. Somit werden die drei Schemata S_6, S_7 und S_8 erzeugt.

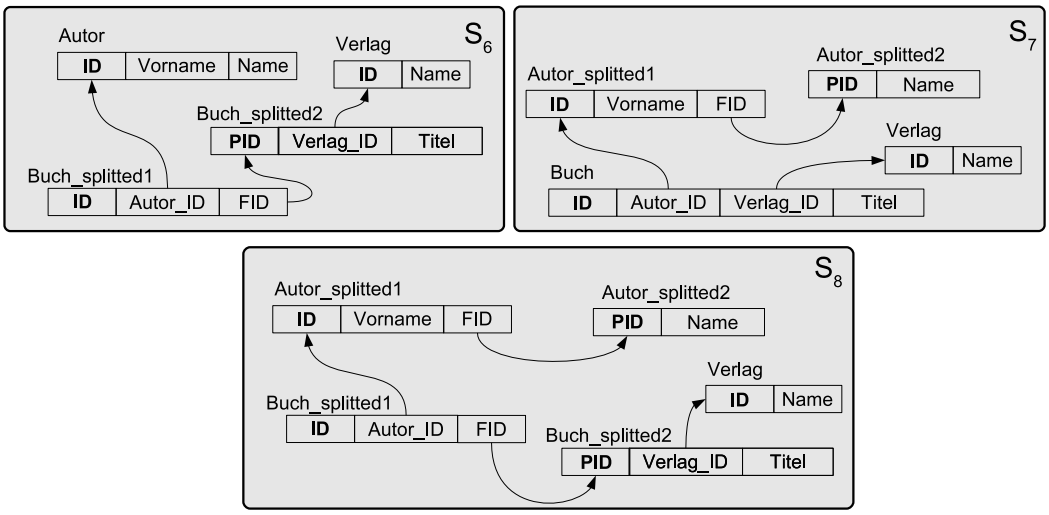


Abbildung 4.5: Durch den Normalisierungsalgorithmus erzeugte Schemata

Zu den in Abbildung 4.5 gezeigten Teilschemata wurden folgende definierende Mappings erzeugt:

$$
\begin{aligned}
S_6.\text{Buch\_splitted1}(x,y,fid) &:- S.\text{Buch}(x,y,z,u)\\
S_6.\text{Buch\_splitted2}(pid,z,u) &:- S.\text{Buch}(x,y,z,u)\\
S_6.\text{Verlag}(x,y) &:- S.\text{Verlag}(x,y)\\
S_6.\text{Autor}(x,y,z) &:- S.\text{Autor}(x,y,z)\\
S_7.\text{Verlag}(x,y) &:- S.\text{Verlag}(x,y)\\
S_7.\text{Buch}(x,y,z,u) &:- S.\text{Buch}(x,y,z,u)\\
S_7.\text{Autor\_splitted1}(x,y,fid) &:- S.\text{Autor}(x,y,z)\\
S_7.\text{Autor\_splitted2}(pid,z) &:- S.\text{Autor}(x,y,z)\\
S_8.\text{Autor\_splitted1}(x,y,fid) &:- S.\text{Autor}(x,y,z)\\
S_8.\text{Autor\_splitted2}(pid,z) &:- S.\text{Autor}(x,y,z)\\
S_8.\text{Buch\_splitted1}(x,y,fid) &:- S.\text{Buch}(x,y,z,u)\\
S_8.\text{Buch\_splitted2}(pid,z,u) &:- S.\text{Buch}(x,y,z,u)\\
S_8.\text{Verlag}(x,y) &:- S.\text{Verlag}(x,y)
\end{aligned}
$$

Die zerteilten Relationen enthalten künstlich eingefügte Attribute, welche nicht im Referenzschema auftauchen. Bei der Bereitstellung der Peer-Extensionen müssen die Werte dieser Attribute unter Beachtung der Zusammengehörigkeit der Tupel berechnet werden (siehe Abschnitt 7.3.2 auf Seite 72).

Der Schema-Pool enthält nach Ausführung dieses Algorithmus zusätzliche durch Normalisierung gewonnene Schemata. Der nächste Abschnitt beschreibt, wie durch Zusammenfassen mehrerer Relationen ebenfalls eine Menge Teilschemata gewonnen wird.

4.5 Denormalisierung

Eine weitere Möglichkeit, ein Schema zu modifizieren, ist das Denormalisieren einer Menge von Relationen. Mehrere über Fremdschlüsselbeziehungen verbundene Relationen werden zu einer größeren zusammengefasst. Wie der Name bereits sagt, handelt es sich dabei um einen der Normalisierung entgegenwirkenden Prozess.

Um herauszufinden, welche Relationen zusammengefasst werden können, betrachten wir den Relationengraph, dessen Knoten die Relationen und dessen Kanten die Fremdschlüsselbeziehungen repräsentieren. In Abbildung 4.6 ist ein komplexer Relationengraph für einen fiktives größeres Schema $S^* = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8\}$ dargestellt.

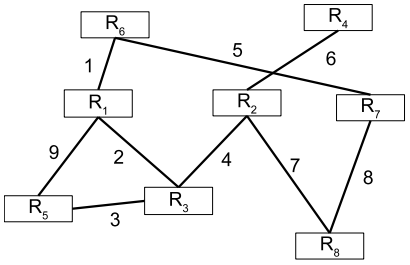


Abbildung 4.6: Ein komplexer Relationengraph

In dem ungerichteten Relationengraph werden zunächst alle Kanten identifiziert und aus diesen alle nicht-leeren Teilmengen gebildet. Für das Beispiel aus Abbildung 4.6 lautet die Kantenmenge $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Die Teilmenge $\{1, 2\}$ bedeutet beispielsweise, dass über die Join-Kette $R_6 \bowtie R_1 \bowtie R_3$ eine neue Relation gebildet wird. Die Kantenmengen dürfen keinen Kreis enthalten, da sonst eine Relation mehrmals in der Join-Kette auftreten würde. Die Kantenmenge $\{9, 2, 3\}$ beschreibt die Join-Kette $R_5 \bowtie R_1 \bowtie R_3 \bowtie R_5$. Die Relation R_5 bringt sich hier in die Ergebnistupel mehrmals, unter Umständen mit unterschiedlichen Werten, ein. Es ist nicht klar, welche der möglichen Werte von R_5 in die Ergebnisrelation gelangen sollen. Daher werden beim Denormalisieren alle Teilmengen unterbunden, die Zyklen enthalten.

Jede zyklenfreie Teilmenge der Kanten entspricht einer möglichen Denormalisierungsvariante. Die Mengen müssen keinen zusammenhängenden Teilgraphen beschreiben. Die Teilmenge $\{2, 8\}$ beschreibt beispielsweise, dass zwei denormalisierte Relationen aus $R_1 \bowtie R_3$ und $R_7 \bowtie R_8$ gebildet werden sollen. Die Anzahl der neuen Relationen pro Teilmenge ist gleich der Anzahl der Komponenten des durch die Teilmenge beschriebenen Teilgraphen.

In Algorithmus 4.3 ist der Vorgang im Pseudocode genauer beschrieben. Auch bei der Denormalisierung muss sichergestellt werden, dass die resultierenden Schemata zusammenhängend sind. Die Funktion `CalcVariants` berechnet daher nicht alle zyklenfreien Teilmengen der Kanten, sondern schließt jene Varianten aus, in denen Kanten zweier verschiedener Zusammenhangskomponenten (bezogen auf das Referenzschema) enthalten sind. Der Pseudocode hierzu ist in Algorithmus 4.4 nachzulesen. Da das Sche-

ma T durch die Funktion `CopySchema` kopiert wurde, enthält es noch Relationen die nicht zu der von `CalcVariants` ausgewählten Komponente gehören. Deshalb wird mit `EnsureOneComponent` sichergestellt, dass diese aus dem Schema T entfernt werden. Die Funktion `FixForeignKeyReferences` stellt die referenzielle Integrität des Schemas sicher. Einige Fremdschlüssel müssen nachträglich verändert werden, da eventuell Relationen aus dem Schema entfernt und durch neue ersetzt wurden.

Input : Referenzschema $S = \{R_1, R_2, \ldots, R_n\}$
Output : Teilschemata $\mathcal{T} = \{S_1, \ldots, S_m\}$

$\mathcal{T} \leftarrow \emptyset$;
`/* Alle Kanten im Schema suchen (Fremdschlüsselbeziehungen) */`
$E \leftarrow$ `GetEdges(`S`);`
`/* Alle Denormalisierungsvarianten berechnen */`
$\mathcal{V} \leftarrow$ `CalcVariants(`E`);`
foreach $V \in \mathcal{V}$ **do**
  `/* Referenzschema kopieren */`
  $T \leftarrow$ `CopySchema(`S`);`
  `/* Berechne Komponenten der Denormalisierungsvariante */`
  $\mathcal{C} \leftarrow$ `CalcSubGraphComponents(`V`);`
  foreach $C \in \mathcal{C}$ **do**
    `/* Neue Relation aus den Relationen in` C `bilden */`
    $R \leftarrow$ `BuildNewRelation(`C`);`
    `/* Aus` T `die Relationen aus` C `entfernen und die neue */`
    `/* Relation` R `einfügen */`
    $T \leftarrow (T \setminus C) \cup \{R\}$
  end
  `EnsureOneComponent`(T, V);
  `FixForeignKeyReferences(`T`);`
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{T\}$;
end
return $\mathcal{T}$;

Algorithmus 4.3 : Denormalisierung

Für das begleitende Beispiel aus Abbildung 4.1 ist der Relationengraph in Abbildung 4.7 dargestellt. Er besitzt die drei Relationen Autor, Buch und Verlag und hat wegen der Fremdschlüsselbeziehungen zwischen Buch und Autor sowie Buch und Verlag die zwei Kanten 1 und 2. Die nicht-leeren Teilmengen von $\{1, 2\}$: $\{1\}$, $\{2\}$ und $\{1, 2\}$ sind alle zyklenfrei und befinden sich in einer Komponente (der Ausgangsgraph ist zusammenhängend). Alle Teilgraphen, die durch die Kanten gebildet werden, sind ebenfalls zusammenhängend. Es wird somit für jede Teilmenge genau eine denormalisierte Relation erzeugt. Für $\{1\}$ ist dies $Autor \bowtie Buch$, für $\{2\}$ $Verlag \bowtie Buch$ und für $\{1, 2\}$ die Joinkette $Autor \bowtie Buch \bowtie Verlag$.

Die erzeugten denormalisierten Relationen setzen sich aus den Attributen der Relationen der Join-Ketten zusammen. Die Join-Attribute werden jeweils auf ein gemeinsames

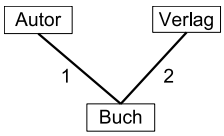


Abbildung 4.7: Relationengraph zum Referenzschema

Attribut abgebildet. Um Namenskonflikte zu vermeiden, müssen die ursprünglichen Attributnamen abgewandelt werden. Die neue Relation erhält ein neues Attribut mit einem synthetischen Primärschlüssel (Surrogatschlüssel). Die Schlüsseldaten werden bei der Berechnung der Extension automatisch generiert. In Abbildung 4.8 sieht man die vom Denormalisierungsalgorithmus erzeugten Schemata. Die Join-Attribute bekommen das Präfix ”y” und werden fortlaufend durchnummeriert.

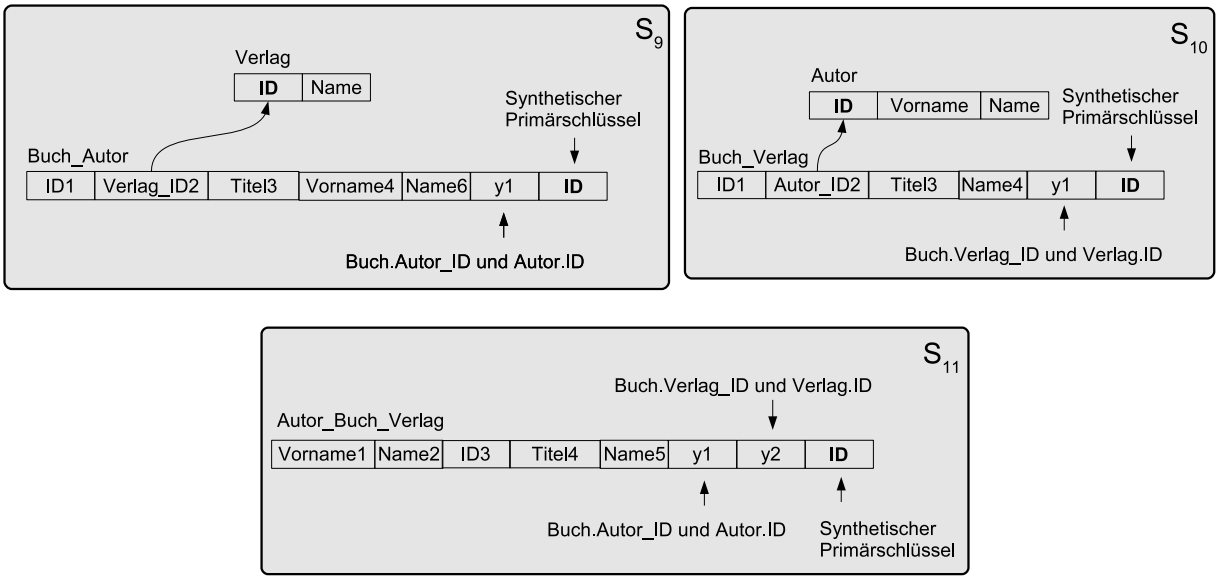


Abbildung 4.8: Durch den Denormalisierungsalgorithmus erzeugte Schemata

Die definierenden Mappings sind im Vergleich zu den vorherigen Algorithmen komplexer, da auf den rechten Seiten mehrere Relationen des Referenzschemas kombiniert werden. Das eingefügte Attribut id tritt nur auf den linken Seiten der Mappings auf, da dessen Werte bei Extensionsbestimmung errechnet und nicht aus dem Referenzschema abgeleitet werden. Dieser Fall trat bereits in Abschnitt 4.4 auf. Die Join-Attribute, die zuvor ein neues Attribut in der Zielrelation erhalten haben, müssen über Variablen auf der linken und rechten Seite der Datalog-Regel abgebildet werden.

$$
\begin{aligned}
\mathsf{S_9.Verlag(x,y)} &: -\mathsf{S.Verlag(x,y)} \\
\mathsf{S_9.Buch\_Autor}(x_1, x_2, x_3, x_4, x_5, y_1, \mathsf{id}) &: -\mathsf{S.Buch}(x_1, y_1, x_2, x_3), \\
&\quad\ \mathsf{S.Autor}(y_1, x_4, x_5) \\
\mathsf{S_{10}.Autor}(x, y, z) &: -\mathsf{S.Autor}(x, y, z) \\
\mathsf{S_{10}.Buch\_Verlag}(x_1, x_2, x_3, x_4, y_1, \mathsf{id}) &: -\mathsf{S.Buch}(x_1, x_2, y_1, x_3), \\
&\quad\ \mathsf{S.Verlag}(y_1, x_4) \\
\mathsf{S_{11}.Autor\_Buch\_Verlag}(x_1, x_2, x_3, x_4, x_5, y_1, y_2, \mathsf{id}) &: -\mathsf{S.Autor}(y_1, x_1, x_2), \\
&\quad\ \mathsf{S.Buch}(x_3, y_1, y_2, x_4), \\
&\quad\ \mathsf{S.Verlag}(y_2, x_5)
\end{aligned}
$$

Zur Verdeutlichung sind die definierenden Mappings für die Schemata S_9 und S_{11} in Abbildung 4.9 graphisch dargestelt.

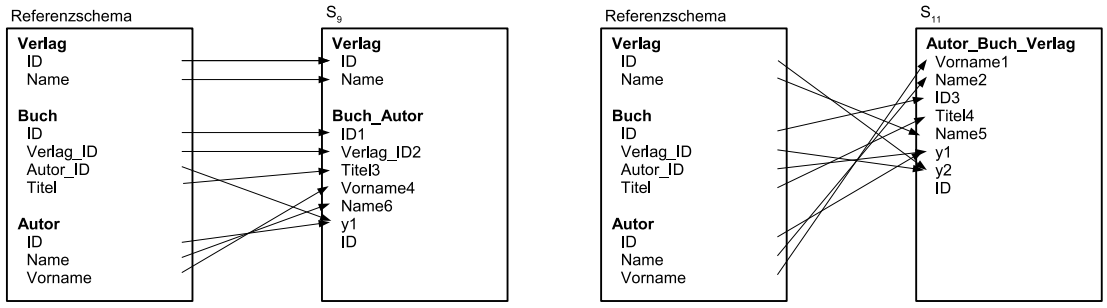


Abbildung 4.9: Visualisierung definierender Mappings des Denormalisierungsalgorithmus

Input : Kanten E
Output : Denormalisierungvarianten $\mathcal{V} \subseteq \mathfrak{P}(E)$

$\mathcal{V} \leftarrow \emptyset$;
`/* Für alle Elemente der Potenzmenge von` E `*/`
foreach $V \in \mathfrak{P}(E)$ **do**
  if $V \neq \emptyset \ \wedge$ `IsCycleFree(`V`)` $\wedge$ `EdgesAreInOneComponent(`V`)` **then**
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{V\}$;
  end
end
return $\mathcal{V}$;

Algorithmus 4.4 : Berechnung von Denormalisierungsvarianten

5 Erzeugung von Schema-Mappings

Ziel der in dieser Arbeit entwickelten Testumgebung ist es unter anderem, Peer-Mappings automatisch zu erzeugen. In diesem Kapitel wird vorgestellt, wie mit existierenden Ansätzen der Schemaintegration und den definierenden Mappings aus Kapitel 4 automatisiert Schema-Mappings erzeugt werden können. Im ersten Abschnitt werden einige Begriffe aus dem Bereich der Schema-Integration vorgestellt. Im Anschluss wird auf das Thema Anfragen und Sichten eingegangen. Der letzte Abschnitt beschäftigt sich mit der Übertragung dieser Begriffe auf unser Problem des automatischen Auffindens von Schema-Mappings und der Umsetzung.

5.1 Grundlegende Begriffe

Jeder Peer hat im Allgemeinen ein unabhängig erzeugtes Schema, welches sich von den Schemata der anderen Peers unterscheidet. In vielen Fällen lassen sich die Schemata zumindest teilweise durch Schematransformation ineinander überführen. Die im Kapitel 4 erzeugten Schemata wurden zwar unabhängig voneinander generiert, jedoch sind diese alle über eine Transformation des gegebenen Referenzschemas beschreibbar. Die Idee ist, Beschreibungen für Schematransformationen zwischen verschiedenen Peer-Schemata zu finden. Im Bereich der Informationsintegration existieren allgemeine Begriffe für die Beschreibung der Schemaintegration. Diese werden im Folgenden erläutert, um später damit Schema-Mappings charakterisieren zu können.

Definition 5.1 (Korrespondenz) Eine Korrespondenz ist die Zuordnung einer oder mehrerer Elemente eines Quellschemas zu einem oder mehreren Elementen eines Zielschemas. Sind Q die Elemente des Quellschemas und Z die Elemente des Zielschemas, so ist eine Korrespondenz eine Zuordnung z charakterisiert durch

$$z \in (\mathfrak{P}(Q) \setminus \emptyset) \times (\mathfrak{P}(Z) \setminus \emptyset).$$

Zur Korrespondenz gehört eine Transformationsfunktion t, die Daten und Metadaten des Quellschemas in Daten des Zielschemas umwandelt.

Als Schemaelemente kommen in dieser Arbeit ausschließlich Attribute von Relationen in Betracht. Zudem betrachten wir lediglich 1:1-Korrespondenzen, das heißt, es wird immer ein Attribut auf genau ein anderes Attribut abgebildet.

Beispiel 5.1 *Ein Beispiel für eine $n:1$-Korrespondenz ist*

$$z = (\{\mathsf{R}_1.\mathsf{Name}, \mathsf{R}_1.\mathsf{Vorname}\}, \{\mathsf{R}_2.\mathsf{Name}\}).$$

Hier werden zwei Attribute des Quellschemas einem Attribut des Zielschemas zugeordnet. Eine mögliche Transformationsfunktion ist die Konkatenation der beiden Quellattribute.

Definition 5.2 (High-Level Mapping) Ein High-Level Mapping ist eine Menge von Korrespondenzen zwischen zwei Schemata. Sind Q die Elemente des Quellschemas und Z die Elemente des Zielschemas, so ist ein Mapping M als Teilmenge

$$M \subseteq (\mathfrak{P}(Q) \setminus \emptyset) \times (\mathfrak{P}(Z) \setminus \emptyset)$$

auffassbar.

Ausführlichere Definitionen von Korrespondenzen und Mappings sind in [2] und [10] zu finden. Sind zwischen zwei Schemata High-Level Mappings definiert, so kann eine Datentransformationsregel erstellt werden, die beschreibt, wie Daten des Quellschemas in Daten mit der Struktur des Zielschemas überführt werden. Die Mappings sind demnach noch zu *interpretieren* und in eine Anfragesprache (beispielsweise SQL oder Datalog) zu übersetzen, welche diese Transformation leistet. Die Anfragesprache Datalog hat sich bewährt, um derartige Schematransformationen auf einfache Art zu beschreiben.

5.2 Anfragen und Sichten

In [6] beschäftigt sich Halevy mit dem Problem Anfragen unter Nutzung von Sichten zu beantworten und verwendet dabei Ansätze, die sich gut eignen, um Mappings zwischen Peers zu charakterisieren. Einige der Definitionen wollen wir hier aufgreifen und dabei die in Abschnitt 4.2 eingeführte Datalog-Notation näher erläutern.

Definition 5.3 (**Konjunktive Anfrage)** Eine konjunktive Anfrage (engl. *conjunctive query*) hat die Form

$$q(X) :- r_1(X_1), \ldots, r_n(X_n).$$

hierbei sind q und $r_1, \ldots, r_n$ Prädikatnamen. Das Element $q(X)$ wird als *Kopf* der Anfrage bezeichnet. Die Elemente $r_1(X_1), \ldots, r_n(X_n)$ bezeichnet man als *Teilziele*. $X, X_1, \ldots X_n$ sind Tupel, welche Variablen oder Konstanten enthalten können. Teilziele dürfen auch Vergleichsprädikate wie z.B. $<, \leq, =, \neq$ enthalten.

- Join-Prädikate in SQL werden bei Verwendung von konjunktiven Anfragen mit mehrfacher Verwendung von Variablennamen ausgedrückt.
- Vereinigungen in SQL lassen sich durch mehrere konjunktive Anfragen beschreiben, welche dasselbe Kopf-Prädikat verwenden.

Eine sehr ausführliche Definition von konjunktiven Anfragen findet man in [11]. Wie bereits in Abschnitt 2.3 kurz dargestellt, werden für die Formulierung von Schema- bzw. Peer-Mappings konjunktive Anfragen verwendet. Dies gilt auch für Anfragen, die der Nutzer an Peers stellt.

Beispiel 5.2 *Die SQL-Anfrage*

```
select  Autor.Name, Autor.Vorname, Buch.Titel
from    Autor, Buch, Verlag
where   Buch.Autor_ID = Autor.ID and Buch.Verlag_ID = Verlag.ID and
        Verlag.Name = "Pearson Studium".
```

würde in Datalog-Notation wie folgt dargestellt werden:

```
q(Name, Vorname, Titel) :-  Autor(AID, Vorname, Name), Buch(BID, AID, VID, Titel),
                            Verlag(VID, VName), VName = "Pearson Studium".
```

Wie in Abschnitt 2.3 dargestellt, existiert eine Teilmengenbeziehung zwischen den von Mapping-Kopf und Mapping-Rumpf errechneten Tupelmengen. Dieser Sachverhalt wird in der folgenden Definition näher charakterisiert.

Definition 5.4 (Containment und Äquivalenz) Seien Q_1 und Q_2 Anfragen an ein Schema S. Q_1 ist enthalten in einer Anfrage Q_2, geschrieben als $Q_1 \subseteq Q_2$, wenn für jede Datenbankinstanz D die Menge der durch Q_1 errechneten Tupel, Teilmenge der durch Q_2 errechneten Tupelmenge ist, also $Q_1(D) \subseteq Q_2(D)$ gilt. Zwei Anfragen sind äquivalent, wenn $Q_1 \subseteq Q_2$ und $Q_2 \subseteq Q_1$ erfüllt ist. Die Definition bezieht sich nicht auf die Syntax der Anfragen, sondern auf die erzeugten Tupelmengen. Zwei völlig unterschiedliche Anfragen mit verschiedener Syntax können trotzdem zueinander *contained* sein.

Die in Kapitel 4 erzeugten Datalog-Regeln sind auf der einen Seite als Transformationsregeln zu verstehen, können auf der anderen Seite aber auch als Sichtdefinitionen verstanden werden. Sind zwei Peers P_1 und P_2 gegeben und soll untersucht werden, ob

$$Q_1(P_1) \subseteq Q_2(P_2) \text{ bzw. } Q_2(P_2) \subseteq Q_1(P_1)$$

gilt, so ist dies zunächst nicht möglich, da P_1 und P_2 möglicherweise Peers mit unterschiedlichen Schemadefinitionen sind. Containment ist bisher nur über zwei Anfragen an dasselbe Schema definiert. $Q_1(P_1)$ und $Q_2(P_2)$ liefern im Allgemeinen Daten mit unterschiedlichen Strukturen, dennoch wollen wir Mappings zwischen Peers wie folgt charakterisieren:

Ein Peer-Mapping hat die Form $Q_1(\mathcal{P}_1) \subseteq Q_2(\mathcal{P}_2)$, wobei $\mathcal{P}_1$ und $\mathcal{P}_2$ Mengen von Peers (vgl. hierzu [7]) und Q_1 sowie Q_2 Anfragen an die Schemata dieser Peers sind. Die Schreibweise $Q_1(\mathcal{P}_1) \subseteq Q_2(\mathcal{P}_2)$ gibt das Verhältnis zwischen beiden Schemamengen an. Für jede Extension liefert die Anfrage Q_1 eine Teilmenge der Tupel der Anfrage Q_2. Solche Mappings werden GLaV-Mappings (Global-Local-As-View) genannt und sind gerichtet. Demzufolge werden $Q_1(\mathcal{P}_1)$ als Kopf (engl. *Head*) und $Q_2(\mathcal{P}_2)$ als Rumpf (engl. *Tail*) des Mappings bezeichnet.

Es wird davon ausgegangen, dass die Anfrage $Q_2(\mathcal{P}_2)$ umformuliert werden kann in die Anfrage $Q_1(\mathcal{P}_1)$. Durch die Containment-Beziehung beider Anfragen ist es demnach möglich, $Q_2(\mathcal{P}_2)$ durch $Q_1(\mathcal{P}_1)$ zu substituieren. Umgekehrt darf $Q_2(\mathcal{P}_2)$ jedoch nicht durch $Q_1(\mathcal{P}_1)$ ersetzt werden.

In dieser Arbeit beschäftigen wir uns nur mit dem praxisnahen Fall, dass $\mathcal{P}_1$ und $\mathcal{P}_2$ einelementige Mengen sind und die Mappings somit nur zwischen zwei Peers definiert werden. Weiterhin sollen nur diese beiden Spezialfälle aus Abschnitt 2.3 berücksichtigt werden:

- $Q_1(P_1) \subseteq P_2.R$ entspricht dem Global-As-View-Ansatz, in dem $Q_2 = P_2.R$ nur aus einem Teilziel besteht. Das Teilziel $P_2.R$ kann als Sicht auf die Teilziele der Anfrage $Q_1(P_1)$ aufgefasst werden.
- $P_1.R \subseteq Q_2(P_2)$ deckt den Spezialfall des Local-As-View-Ansatzes ab, bei dem die Quellschemata über Sichten auf das integrierte Schema definiert werden. In diesem Fall ist das Schema des Peers P_2 als integriertes und das Schema von P_1 als ein Quellschema anzusehen.

Für die Modellierung eines Mappings $Q_1(P_1) \subseteq Q_2(P_2)$ lassen sich die oben eingeführten konjunktiven Anfragen verwenden. Die Formulierung von $Q_1(P_1)$ und $Q_2(P_2)$ erfolgt dabei in Datalog-Notation. Die Teilziele der beiden Anfragen bilden Kopf und Rumpf des Mappings. Korrespondenzen zwischen den Peer-Schemata werden durch geeignete Wahl von Datalog-Variablen ausgedrückt.

Beispiel 5.3 *Gegeben sind zwei Anfragen q_1 und q_2 an die Peers P_1 und P_2:*

$$\begin{aligned} q_1(\text{Name, Vorname, Titel, AID, BID, VID}) :- \quad & P_1.\text{Autor(AID, Vorname, Name)}, \\ & P_1.\text{Buch(BID, AID, VID, Titel)} \\ q_2(\text{Name, Vorname, Titel, AID, BID, VID, ID}) :- \quad & P_2.\text{Autor\_Buch(BID, VID, Vorname,} \\ & \text{Name, Titel, AID, ID)} \end{aligned}$$

Peer P_1 hat das im Kapitel 4 illustrierte Schema S_1 (siehe Abbildung 4.3) und P_2 das Schema S_9 (siehe Abbildung 4.8). Unter Berücksichtigung der Korrespondenzen (vgl. Abbildung 4.9) zwischen beiden Schemata sind folgende Mappings denkbar:

$$\begin{aligned} P_1.\text{Autor(AID, Vorname, Name)}, \\ P_1.\text{Buch(BID, AID, VID, Titel)} \quad :- \quad & P_2.\text{Autor\_Buch(BID, VID, Vorname,} \\ & \text{Name, Titel, AID, ID)} \\ P_2.\text{Autor\_Buch(BID, VID, Vorname, Name,} \\ \text{Titel,AID,ID)} \quad :- \quad & P_1.\text{Autor(AID, Vorname, Name)}, \\ & P_1.\text{Buch(BID, AID, VID, Titel)} \end{aligned}$$

In einem PDMS darf jeder Peer seine Mappings nutzen, um an ihn gerichtete Anfragen zu beantworten. Die Anfrageumformulierung erfolgt immer von rechts nach links. Teilziele auf der Rumpf-Seite eines Mappings dürfen mit Teilzielen auf der Kopf-Seite beantwortet werden. In Beispiel 5.3 soll das erste Peer-Mapping betrachtet werden. Wird die Relation Autor_Buch in Peer P_2 angefragt, so darf dieser, aufgrund des ersten Mappings, Peer P_1

eine umformulierte Anfrage stellen und die gelieferten Tupel aus P_1 in die Antwort einbeziehen. Die Variable ID wird nicht auf der linken Seite des Mappings verwendet. Die von P_1 gelieferten Tupel müssen daher um ein ID-Attribut ergänzt und mit einem NULL-Wert aufgefüllt werden.

Betrachtet man im letzen Beispiel das zweite Mapping, so ergeben sich folgende Fälle: P_1 kann nur nach dem Teilziel Autor, nach dem Teilziel Buch oder nach beiden Teilzielen gleichzeitig angefragt werden. In allen Fällen darf P_2 aufgrund des Mappings P_1 anfragen, um das Ergebnis zu liefern. Nicht verwendete Attribute der Tupel werden entfernt und erscheinen im Ergebnis nicht mehr. Hierbei handelt es sich um Anfragebeantwortung unter Ausnutzung von Sichten.

5.3 Suche nach Mappings

In Kapitel 4 wurde ein Schema-Pool erzeugt, welcher im Folgenden um Schema-Mappings ergänzt werden soll. Für jedes Schema innerhalb dieses Pools existieren Sichtdefinitionen in Form von Datalog-Regeln, die das Verhältnis zum Referenzschema beschreiben. Sie wurden bei der Generierung der Schemata angelegt und können benutzt werden, um Mappings zu erzeugen. In den folgenden Unterabschnitten wird der grobe Ablauf beschrieben und die Containment-Begriffe auf Peer-Mappings übertragen. Anschließend wird gezeigt, wie die Suche nach Mappings konkret funktioniert.

5.3.1 Ablauf

Für jedes Schema-Paar (S_i, S_j) mit $i \neq j$ aus dem Schema-Pool wird geprüft, ob ein Mapping erzeugt werden kann. Jede Sichtdefinition (siehe Abschnitt 4.2) der beiden Schemata aus dem Paar wird entfaltet. Das Entfalten erfolgt, indem die Relationen im erzeugten Schema durch die korrespondierenden Teilziele aus dem Referenzschema ersetzt werden. Die beiden Sichten sind nun miteinander anhand des Referenzschemas vergleichbar und es kann geprüft werden, ob ein Mapping möglich ist. Nach dem Ablauf dieser Mappingsuche steht für jedes erzeugte Teilschema aus dem Pool eine Liste von Schema-Mappings zur Verfügung. In Abbildung 5.1 ist dieser Prozess graphisch dargestellt. Die roten Pfeile illustrieren die gesuchten Mappings zwischen den Schemata und die schwarzen Pfeile die Sichtdefinitionen anhand des Referenzschemas.

5.3.2 Übertragung der Containment-Definition auf Peer-Mappings

In Definition 5.4 wurde *Containment* anhand der Tupelmengen von Anfragen definiert. Eine Überpüfung der Eigenschaft $Q_1(P_1) \subseteq Q_2(P_2)$ für ein Peer-Mapping kann also erfolgen, indem man für alle möglichen Datenbankinstanzen feststellt, ob die Teilmengenbeziehung erfüllt ist. In [11] stellt Ulf Leser Ansätze vor, wie sich Containment zwischen Anfragen berechnen lässt.

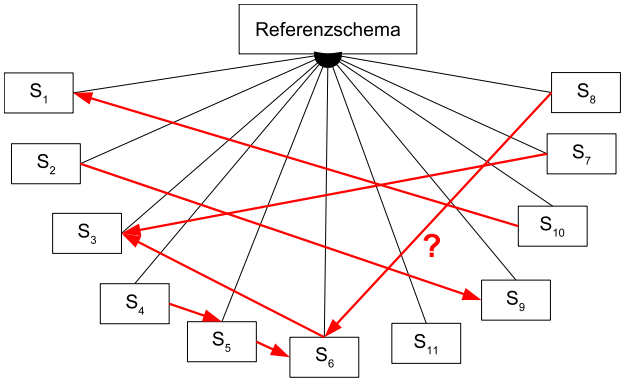


Abbildung 5.1: Erzeugte Schemata mit Sichten zum Referenzschema

Definition 5.5 (Containment-Mapping) Seien Q_1 und Q_2 zwei Anfragen. Ein *Containment-Mapping* h zwischen Q_2 und Q_1 ist eine Abbildung zwischen den Symbolmengen von Q_2 und Q_1, so dass

(CM1) Jede exportierte Variable v aus Q_2 wird durch h auf eine Variable in Q_1 abgebildet.

(CM2) Jede Konstante c aus Q_2 wird durch h auf sich selbst abgebildet ($h(c) = c$).

(CM3) Jede Relation in Q_2 wird auf mindestens eine Relation in Q_1 abgebildet.

(CM4) Die Bedingungen von Q_1 implizieren die Bedingungen von Q_2.

Satz 5.1 (Beziehung von Containment-Mappings zu Query Containment) *Seien Q_1 und Q_2 zwei Anfragen an das Schema S, dann*

$$Q_1 \subseteq Q_2 \Leftrightarrow \text{Es existiert ein } \textit{Containment-Mapping} \text{ von } Q_2 \text{ nach } Q_1.$$

Beweis: Der Beweis ist in [4] von A. K. Chandra und P. M. Merlin nachzulesen. □

Die obige Definition von *Containment-Mapping* ist zu scharf. Wenn Projektionen über Peer-Mappings modelliert werden sollen (siehe Abschnitt 7.2), können die Eigenschaften (CM1) und (CM2) nicht garantiert werden. Projektionen werden durch nicht zugeordnete Variablen modelliert und widersprechen den genannten Bedingungen. Solche Variablen werden in derartigen Fällen während der Anfragebearbeitung mit NULL-Werten aufgefüllt bzw. überflüssige, nicht angefragte Werte aus dem Ergebnis herausprojiziert.

Daher sind aus der Definition nur die Eigenschaften (CM3) und (CM4) für uns relevant, und wir kommen zu folgender neuen Definition:

Definition 5.6 (Schwaches Containment-Mapping) Seien Q_1 und Q_2 zwei Anfragen. Ein *schwaches Containment-Mapping* h zwischen Q_2 und Q_1 ist eine Abbildung zwischen den Symbolmengen von Q_2 und Q_1, so dass

(SCM1) Jede Relation in Q_2 wird auf mindestens eine Relation in Q_1 abgebildet.

(SCM2) Die Bedingungen von Q_1 implizieren die Bedingungen von Q_2, und Vergleichsprädikate in Q_2 müssen auf Variable definiert sein, welche auch in Q_1 vorkommen.

Mit Hilfe des *schwachen Containment-Mappings* kann das *Containment* zwischen zwei Anfragen, welche an zwei verschiedene Schemata gestellt werden, charakterisiert werden:

Definition 5.7 (Schwaches Containment von Anfragen an Teilschemata) Seien S_1 und S_2 zwei erzeugte Teilschemata und S das Referenzschema. Die Anfrage $Q_1(S_1)$ ist in $Q_2(S_2)$ enthalten, also

$$Q_1(S_1) \subseteq Q_2(S_2),$$

wenn

$$\exists \text{ ein } \textit{schwaches Containment-Mapping} \text{ von } Q_2' \text{ nach } Q_1'$$

erfüllt ist. Hierbei sind $Q_1'(S)$ und $Q_2'(S)$ die durch Sichtentfaltung umgeschriebenen Anfragen zu Q_1 und Q_2. Die Sichtentfaltung geschieht unter Verwendung der definierenden Mappings der Schemata S_1 und S_2.

Die neue Definition von *schwachem Containment* für Anfragen an unterschiedliche Schemata ist nötig, da nicht auf *Containment-Mappings* gemäß Definition 5.5 zurückgegriffen werden kann und somit der Satz 5.1 nicht greift. Durch modellierte Projektionen kann es passieren, dass von $Q_1'(S)$ Tupel geliefert werden, welche an bestimmten Attributen ausschließlich NULL-Werte enthalten. Die gleichen Tupel sind in $Q_2'(S)$ ebenfalls vorhanden, jedoch eventuell ohne NULL-Werte. Damit existieren Tupel in $Q_1'(S)$, welche in $Q_2'(S)$ nicht existieren, obwohl die Extensionen semantisch übereinstimmen.

Beispiel 5.4 *Sei S ein Referenzschema mit den Relationen $S = \{R_1(a,b,c), R_2(c,d,e)\}$. Zwischen $S.R_1.c$ und $S.R_2.c$ sei eine Fremdschlüsselbeziehung definiert. Gegeben seien auch $S_1 = \{R_3(a,b,c,d,e)\}$ und $S_2 = \{R_4(a,b,c)\}$ als aus S generierte Teilschemata mit folgenden Sichtdefinitionen (definierende Mappings):*

$$\begin{aligned} \mathsf{S_1.R_3(a,b,c,d,e)} &:\!- \quad \mathsf{S.R_1(a,b,c), S.R_2(c,d,e)} \\ \mathsf{S_2.R_4(a,b,c)} &:\!- \quad \mathsf{S.R_1(a,b,c)} \end{aligned}$$

Es soll, anhand der Definition von schwachem Containment, geprüft werden, ob das Schema-Mapping

$$\mathsf{S_1.R_3(a,b,c,d,e)} \subseteq \mathsf{S_2.R_4(a,b,x)}$$

gültig ist. Es enthält eine Projektion, die verhindert, dass Werte für $\mathsf{S_1.R_3.c}$ *nach* $\mathsf{S_2.R_4.c}$ *übertragen werden. Stattdessen erhalten die gelieferten Tupel an dieser Stelle einen* NULL*-Wert. In unserem System werden Projektionen durch unterschiedliche Variablennamen an den jeweiligen Attributpositionen modelliert. In der Literatur findet man auch alternativ die Notation* $\mathsf{S_1.R_3(a,b,c,d,e)} \subseteq \mathsf{S_2.R_4(a,b,-)}$ *vor. Nach der Sichtentfaltung ist zu prüfen, ob*

$$\mathsf{S.R_1(a,b,c), S.R_2(c,d,e)} \subseteq \mathsf{S.R_1(a,b,x)}$$

gilt. Nach der Definition 5.5 und Satz 5.1 ist diese Beziehung falsch, da das Symbol x *nicht auf der linken Seite des Mappings vorkommt (CM1). Es existiert kein Containment-Mapping, ein schwaches Containment-Mapping hingegen schon. Die Relation* $\mathsf{S.R_1}$ *auf der rechten Seite des Mappings hat ein Pendant auf der linken Seite (SCM1). Da in dem Mapping keine Selektionsbedingungen existieren (diese werden in Abschnitt 7.4 vorgestellt), ist auch (SCM2) erfüllt. Somit ist das Mapping* $\mathsf{S_1.R_3(a,b,c,d,e)} \subseteq \mathsf{S_2.R_4(a,b,x)}$ *zulässig.*

5.3.3 Automatisches Finden von Mappings

Sei $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ ein Schema-Pool. Für alle Paare $(S_i, S_j) \in \mathcal{S} \times \mathcal{S}, i \neq j$ sollen Mappings gefunden werden. Zwei ausgewählte Schemata S_i und S_j aus dem Pool seien fixiert und setzen sich aus den Relationen

$$S_i = \{V_1, V_2, \ldots, V_k\}$$
$$S_j = \{W_1, W_2, \ldots, W_l\}$$

zusammen. Für das geordnete Paar (S_i, S_j) sollen Local-As-View und Global-As-View Mappings generiert werden. Dies sind Mappings der Form

$$S_i.V_q \subseteq Q_2(S_j)$$

oder

$$Q_2(S_j) \subseteq S_i.V_q.$$

Die Relationen $V_1, V_2, \ldots, V_k$ und $W_1, W_2, \ldots, W_l$ sind Sichten auf das Referenzschema S in der Form

$$V_t(X) :- S.R_{a_1}(X_1), S.R_{a_2}(X_2), \ldots, S.R_{a_n}(X_n).$$

Dabei sind $S.R_{a_s}$ Relationen des Referenzschemas S und X_t Tupel mit Variablennamen (siehe Definition 5.3 über konjunktive Anfragen). Sei $\mathcal{R}_i$ die Menge aller Relationen $R \in S$, die Verwendung in mindestens einer Sichtdefinition aus S_i finden. Analog sei $\mathcal{R}_j$ zu S_j definiert. Eine notwendige Bedingung für die Existenz eines Mappings zwischen S_i und S_j ist, dass $\mathcal{R} = \mathcal{R}_i \cap \mathcal{R}_j \neq \emptyset$ gilt. Andernfalls wären die Schemata S_i und S_j disjunkt, da sie keine gemeinsame Relation aus dem Referenzschema zur Definition verwenden. Für ein Mapping kommen nur diejenigen Relationen aus S_i und S_j in Frage, deren Sichtdefinitionen mindestens eine Relation aus $\mathcal{R}$ enthalten. Die Menge $S_i^* \subseteq S_i$ sei definiert als Menge der Relationen aus S_i, welche Relationen aus $\mathcal{R}$ in ihren Sichtdefinitionen verwenden. Analog sei die Menge $S_j^* \subseteq S_j$ definiert. Für jedes Teilziel $V \in S_i^*$ werden diejenigen Teilziele $\mathcal{W} \subseteq S_j^*$ ausgewählt, welche mindestens eine Relation aus dem Referenzschema

für ihre Definition verwenden, die auch von V verwendet wird. Während $\mathcal{W}$ eine Menge von Relationen umfasst, steht V nur für eine einzige Relation. Bei V und $\mathcal{W}$ handelt es sich um genau die Teilziele, die in dem zu konstruierenden Schema-Mapping im Kopf bzw. Rumpf auftreten. Somit werden ausschließlich LaV- und GaV-Mappings generiert, da entweder der Kopf oder der Rumpf des Mappings einelementig ist.

Betrachten wir die Anfragen $Q_1(V)$ und $Q_2(\mathcal{W})$, so werden diese umgeschrieben in Anfragen $Q_1'(S)$ und $Q_2'(S)$ und ein *schwaches Containment-Mapping* zwischen diesen gesucht. Wegen fehlenden Vergleichsprädikaten in den Anfragen ist nur zu prüfen, ob die Relationen aus $Q_1'(S)$ in der Menge der Relation aus $Q_2'(S)$ enthalten sind oder umgekehrt. Ist eine der beiden Prüfungen wahr, so ist ein Mapping auf Grundlage des Teilziels V und derer in $\mathcal{W}$ möglich. In Algorithmus 5.1 ist das Verfahren in Pseudocode verfasst.

Input : Sichten S_i^* und S_j^*
Output : Kopf/Rumpf-Paare $\mathcal{M} = \{(H_1, T_1), (H_2, T_2), \ldots (H_m, T_m)\}$

$\mathcal{M} \leftarrow \emptyset$ **foreach** $V \in S_i^*$ **do**
  $A \leftarrow \{V\}$;
  $B \leftarrow \emptyset$;
  foreach $W \in S_j^*$ **do**
    if *V und W benutzen gleiche Relationen aus Referenzschema S* **then**
      $B \leftarrow B \cup \{W\}$;
    end
  end
  if `IsContained`(A,B) **then**
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(A, B)\}$;
  end
  if `IsContained`(B,A) **then**
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(B, A)\}$;
  end
end
return $\mathcal{M}$;

Algorithmus 5.1 : Berechnung der Kopf- und Rumpf-Seiten der Mappings

Die Funktion `IsContained` aus Algorithmus 5.2 besitzt die beiden Mengen A und B als Parameter. Deren Elemente sind Teilziele der erzeugten Schemata des Schema-Pools. Der Algorithmus bildet die Menge $\mathcal{R}_A$, welche alle Relationen aus dem Referenzschema enthält, die von Relationen aus A zur Sichtdefinition verwendet wurden. Analog wird die Menge $\mathcal{R}_B$ gebildet. Ist $\mathcal{R}_B \subseteq \mathcal{R}_A$, so können alle Relationen aus $\mathcal{R}_B$ auf die Relationen in $\mathcal{R}_A$ abgebildet werden. Es existiert ein schwaches Containment-Mapping von $\mathcal{R}_B$ nach $\mathcal{R}_A$. Betrachtet man nun A und B als Anfragen und nicht mehr als Menge von Relationen, so ist nach Definition 5.7 A in B enthalten. Intuitiv wird dies deutlich, wenn man die Anzahl der Joins in den entfalteten Anfragen vergleicht. Da $\mathcal{R}_B \subseteq \mathcal{R}_A$ gilt, werden

für die Definition der Teilziele aus A alle Joins verwendet, die auch zur Definition von B verwendet wurden und möglicherweise noch mehr. Da Joins über Fremdschlüssel die Ergebnismenge im Allgemeinen reduzieren, muss $\mathcal{R}_A$ als Anfrage aufgefasst weniger oder gleich viele Tupel liefern wie $\mathcal{R}_B$.

Function *IsContained()*

Input : Mengen mit Relationen A, B
Output : true oder false

`/* Funktion IsContained(`A, B`)` `*/`
$\mathcal{M} \leftarrow \emptyset$;
$\mathcal{R}_A \leftarrow$ Alle von A referenzierten Relationen aus dem Referenzschema;
$\mathcal{R}_B \leftarrow$ Alle von B referenzierten Relationen aus dem Referenzschema;
return $\mathcal{R}_B \subseteq \mathcal{R}_A$;

Algorithmus 5.2 : Prüfung der (SCM1)-Eigenschaft

Beispiel 5.5 *Es seien das Referenzschema*

$$\begin{aligned}\mathsf{S} := \{&\mathsf{Autor(ID,\ Vorname,\ Name)},\ \mathsf{Verlag(ID,\ Name)},\\ &\mathsf{Buch(ID,\ Autor\_ID,\ Verlag\_ID,\ Titel)}\}\end{aligned}$$

und zwei generierte Schemata

$$\begin{aligned}\mathsf{S_1} := \{&\mathsf{Autor(ID,\ Vorname,\ Name)},\\ &\mathsf{Buch(ID,\ Autor\_ID,\ Verlag\_ID,\ Titel)}\}\\ \mathsf{S_{10}} := \{&\mathsf{Autor(ID,\ Vorname,\ Name)},\\ &\mathsf{Buch\_Verlag(ID1,\ Autor\_ID2,\ Titel3,\ Name4,\ y1,\ ID)}\}\end{aligned}$$

gegeben (diese wurden in den Beispielen von Kapitel 4 bereits vorgestellt). Die Sichtdefinitionen für S_1 und S_{10} sind

$$\begin{aligned}\mathsf{S_1.Autor(ID,\ Vorname,\ Name)} &: -\mathsf{S.Autor(ID,\ Vorname,\ Name)}\\ \mathsf{S_1.Buch(ID,\ Autor\_ID,\ Verlag\_ID,\ Titel)} &: -\mathsf{S.Buch(ID,\ Autor\_ID,\ Verlag\_ID,\ Titel)}\\ \mathsf{S_{10}.Autor(ID,\ Vorname,\ Name)} &: -\mathsf{S.Autor(ID,\ Vorname,\ Name)}\\ \mathsf{S_{10}.Buch\_Verlag(x1,\ x2,\ x3,\ x4,\ y1,\ id)} &: -\mathsf{S.Buch(x1,\ x2,\ y1,\ x3)},\\ &\qquad\mathsf{S.Verlag(y1,\ x4)}\end{aligned}$$

Die benutzten Relationen des Referenzschemas sind

$$\begin{aligned}\mathcal{R}_1 &= \{\mathsf{S.Autor}, \mathsf{S.Buch}\}\\ \mathcal{R}_{10} &= \{\mathsf{S.Autor}, \mathsf{S.Buch}, \mathsf{S.Verlag}\}\end{aligned}$$

und die Schnittmenge $\mathcal{R} = \mathcal{R}_1 \cap \mathcal{R}_{10} = \{S.Autor, S.Buch\}$ ist nicht leer. Somit ist die notwendige Bedingung für ein Mapping erfüllt. Weiter ergeben sich die Mengen

$$\begin{aligned}\mathsf{S_1^*} &= \{\mathsf{S_1.Autor}, \mathsf{S_1.Buch}\}\\ \mathsf{S_{10}^*} &= \{\mathsf{S_{10}.Autor}, \mathsf{S_{10}.Buch\_Verlag}\}.\end{aligned}$$

V	$\mathcal{W}$	$\mathcal{R}_V$	$\mathcal{R}_\mathcal{W}$	Cont.
$\mathsf{S_1}$.Autor	$\mathsf{S_{10}}$.Autor	S.Autor	S.Autor	$\subseteq$ / $\supseteq$
$\mathsf{S_1}$.Buch	$\mathsf{S_{10}}$.Buch_Verlag	S.Buch	S.Buch, S.Verlag	$\supseteq$
$\mathsf{S_{10}}$.Autor	$\mathsf{S_1}$.Autor	S.Autor	S.Autor	$\subseteq$ / $\supseteq$
$\mathsf{S_{10}}$.Buch_Verlag	$\mathsf{S_1}$.Buch	S.Buch, S.Verlag	S.Buch	$\subseteq$

Tabelle 5.1: Ablauf Mappingsuche, erstes Beispiel

In Tabelle 5.1 wird dargestellt, wie für jedes Teilziel $V \in S_1^*$ nach Teilzielen $\mathcal{W} \subseteq S_{10}^*$ gesucht wird, welche mindestens eine Relation aus dem Referenzschema gemeinsam mit V bei ihrer Sichtdefinition benutzen. Die Menge $\mathcal{R}_V \subseteq \mathcal{R}$ ist die Menge der Relationen aus dem Referenzschema, welche bei der Sichtdefinition des Teilziels V verwendet wurden. Analog dazu enthält $\mathcal{R}_\mathcal{W}$ die Relationen aus $\mathcal{R}$, welche von den Teilzielen aus $\mathcal{W}$ zur Definition benutzt wurden. Ist $\mathcal{R}_V \subseteq \mathcal{R}_\mathcal{W}$, so existiert ein schwaches Containment-Mapping von $Q'_2(\mathcal{R}_V)$ in $Q'_1(\mathcal{R}_\mathcal{W})$, denn jede Relation aus Q'_2 kann abgebildet werden auf eine Relation in Q'_1. Die Anfragen Q'_1 und Q'_2 sind die entfalteten Anfragen $Q_1(\mathcal{W})$ und $Q_2(V)$. Somit gilt nach Definition 5.7, dass $Q_1(\mathcal{W}) \subseteq Q_2(V)$. Wir verifizieren dies genauer anhand der zweiten Zeile in Tabelle 5.1:

- $V = \mathsf{S_1}$.Buch
- $\mathcal{W} = \{\mathsf{S_{10}}$.Buch_Verlag$\}$
- $\mathcal{R}_V = \{$S.Buch$\}$
- $\mathcal{R}_\mathcal{W} = \{$S.Buch, S.Verlag$\}$

Es gilt $\mathcal{R}_V \subseteq \mathcal{R}_\mathcal{W}$. $Q'_2(\mathcal{R}_V)$ fragt die Relation S.Buch und $Q'_1(\mathcal{R}_\mathcal{W})$ die Relationen S.Buch und S.Verlag an. Es existiert ein schwaches Containment-Mapping von $Q'_2(\mathcal{R}_V)$ in $Q'_1(\mathcal{R}_\mathcal{W})$, da S.Buch in Q'_2 auf die Relation S.Buch in Q'_1 abgebildet werden kann und keine Vergleichsprädikate vorliegen. Somit ist $Q_1(\mathcal{W}) \subseteq Q_2(V)$ gemäß Definition 5.7 und das Mapping

$$\mathsf{S_{10}}\text{.Buch\_Verlag(x1,x2,x3,x4,y1,id)} : -\mathsf{S_1}\text{.Buch(x1,x2,y1,x3)}$$

kann gebildet werden. Auf diese Weise entsteht folgende Liste von Mappings:

$$\begin{aligned}
\mathsf{S_1}\text{.Autor(ID,Vorname,Name)} &: -\mathsf{S_{10}}\text{.Autor(ID,Vorname,Name)} \\
\mathsf{S_{10}}\text{.Autor(ID,Vorname,Name)} &: -\mathsf{S_1}\text{.Autor(ID,Vorname,Name)} \\
\mathsf{S_{10}}\text{.Buch_Verlag(x1,x2,x3,x4,y1,id)} &: -\mathsf{S_1}\text{.Buch(x1,x2,y1,x3)}
\end{aligned}$$

Wir wollen diese Mappings intuitiv deuten: Die ersten beiden Mappings sind so genannte Equality-Mappings und kommen somit in zweifacher Ausführung vor. Das letzte Mapping besagt, dass das Ergebnis der Anfrage $Q_1(S_{10}$.Buch_Verlag$)$ in dem Ergebnis der Anfrage $Q_2(S_1$.Buch$)$ enthalten ist, sofern diese sich auf die gleiche Extension des Referenzschemas

beziehen[1]*. Dies ist sofort ersichtlich, da die Relation* S_{10}.Buch_Verlag *über einen Equi-Join zwischen den Relationen* S.Buch *und* S.Verlag *definiert ist, während die Relation* S_1.Buch *lediglich* S.Buch *verwendet. Im Allgemeinen führt der Join mit einer weiteren Relation zu einer Reduzierung der Tupelmenge, da alle Tupel ohne Join-Partner aus dem Ergebnis entfernt werden*[2]*.*

Im Beispiel 5.5 ist ausführlich dargelegt, wie die Mappings zu berechnen sind. Eine Vertiefung bietet das Beispiel 5.6. Hier werden LaV- und GaV-Mappings mit mehreren Teilzielen in Kopf bzw. Rumpf erzeugt. Zusätzlich tritt der Fall auf, dass $\mathcal{R}_V \cap \mathcal{R}_\mathcal{W} \neq \emptyset$ gilt und trotzdem kein Mapping möglich ist.

Beispiel 5.6 *Gegeben sei das Referenzschema*

$$\mathsf{S} := \{\mathsf{R_1(a,b,c)}, \mathsf{R_2(c,d,e)}, \mathsf{R_3(e,f,g)}\}$$

und drei generierte Schemata

$$\begin{aligned}\mathsf{S_1} &:= \{\mathsf{R_4(a,b,c,d,e)}, \mathsf{R_5(e,f,g)}\}\\ \mathsf{S_2} &:= \{\mathsf{R_6(c,d,e,f,g)}\}\\ \mathsf{S_3} &:= \{\mathsf{R_7(a,b,c)}, \mathsf{R_8(c,d,e)}\}\end{aligned}$$

die definierenden Mappings lauten

$$\begin{aligned}\mathsf{S_1.R_4(a,b,c,d,e)} &:- \mathsf{S.R_1(a,b,c)}, \mathsf{S.R_2(c,d,e)}\\ \mathsf{S_1.R_5(e,f,g)} &:- \mathsf{S.R_3(e,f,g)}\\ \mathsf{S_2.R_6(c,d,e,f,g)} &:- \mathsf{S.R_2(c,d,e)}, \mathsf{S.R_3(e,f,g)}\\ \mathsf{S_3.R_7(a,b,c)} &:- \mathsf{S.R_1(a,b,c)}\\ \mathsf{S_3.R_8(c,d,e)} &:- \mathsf{S.R_2(c,d,e)}\end{aligned}$$

Der Mapping-Algorithmus sucht für jedes Schema-Paar nach möglichen Mappings und versucht dabei die rechten Seiten der Mappings so groß wie möglich zu wählen.

In Tabelle 5.2 sind alle Schritte des Algorithmus detailliert aufgeführt. In der Spalte V erscheinen alle Relationen aus den Schema S_1, S_2 *und* S_3 *und werden in Beziehung zu anderen Relationen gesetzt. Zum besseren Verständnis betrachten wir drei Fälle aus der Tabelle genauer:*

---

[1] Das ist nur theoretisch der Fall, in der Praxis hat jedes (Peer-)Schema eine eigene unabhängige Extension.

[2] Wir gehen nur von „sinnvollen“ Joins über Fremdschlüsselattribute aus, andere können über die definierenden Mappings auch nicht erzeugt werden.

V	$\mathcal{W}$	$\mathcal{R}_V$	$\mathcal{R}_\mathcal{W}$	Cont.
$\mathsf{S_1.R_4}$	$\mathsf{S_2.R_6}$	$\mathsf{S.R_1, S.R_2}$	$\mathsf{S.R_2, S.R_3}$	$\not\subseteq$ / $\not\supseteq$
$\mathsf{S_1.R_4}$	$\mathsf{S_3.R_7, S_3.R_8}$	$\mathsf{S.R_1, S.R_2}$	$\mathsf{S.R_1, S.R_2}$	$\subseteq$ / $\supseteq$
$\mathsf{S_1.R_5}$	$\mathsf{S_2.R_6}$	$\mathsf{S.R_3}$	$\mathsf{S.R_2, S.R_3}$	$\supseteq$
$\mathsf{S_2.R_6}$	$\mathsf{S_1.R_4, S_1.R_5}$	$\mathsf{S.R_2, S.R_3}$	$\mathsf{S.R_1, S.R_2, S.R_3}$	$\supseteq$
$\mathsf{S_2.R_6}$	$\mathsf{S_3.R_8}$	$\mathsf{S.R_2, S.R_3}$	$\mathsf{S.R_2}$	$\subseteq$
$\mathsf{S_3.R_7}$	$\mathsf{S_1.R_4}$	$\mathsf{S.R_1}$	$\mathsf{S.R_1, S.R_2}$	$\supseteq$
$\mathsf{S_3.R_8}$	$\mathsf{S_1.R_4}$	$\mathsf{S.R_2}$	$\mathsf{S.R_1, S.R_2}$	$\supseteq$
$\mathsf{S_3.R_8}$	$\mathsf{S_2.R_6}$	$\mathsf{S.R_2}$	$\mathsf{S.R_2, S.R_3}$	$\supseteq$

Tabelle 5.2: Ablauf Mappingsuche, zweites Beispiel

- *In der ersten Zeile haben wir das Teilziel $V = S_1.R_4$, welches zur Definition die Relationen $S.R_1, S.R_2$ aus dem Referenzschema verwendet. Für das Schema S_2 findet der Algorithmus somit $\mathcal{W} = \{S_2.R_6\}$, da diese Relation zur Definition unter anderem $S.R_2$ benutzt. Es besteht somit ein Zusammenhang zwischen $S_1.R_4$ und $S_2.R_6$, da diese Tupel aus der gleichen Relation des Referenzschemas beziehen. Ein Mapping zwischen $S_2.R_6$ und $S_1.R_4$ ist jedoch nicht möglich, da $S_2.R_6$ neben $S.R_2$ auch noch $S.R_3$ verwendet. Es kann kein schwaches Containment-Mapping für die entfalteten Anfragen an diese Relationen gefunden werden, und somit wird kein Mapping erzeugt. Intuitiv betrachtet: Entfaltet man beide Relationen aus S_2 und S_1, so steht der Vergleich zwischen $S.R_1 \bowtie S.R_2$ und $S.R_2 \bowtie S.R_3$ an. Selbst wenn man aus den resultierenden Relationen alle Attribute entfernt, welche nicht zu $S.R_2$ gehören, kann keine Teilmengenbeziehung angegeben werden, da die Semantik des Joins nicht bekannt ist.*

- *In der vierten Zeile wird $S_2.R_6$ betrachet und in Beziehung zu den Relationen $S_1.R_4$ und $S_1.R_5$ gebracht. Zwischen den entfalteten Anfragen $Q'_2(\mathcal{R}_\mathcal{W})$ und $Q'_1(\mathcal{R}_V)$ kann ein schwaches Containment-Mapping gefunden werden, da $\mathcal{R}_V \subseteq \mathcal{R}_\mathcal{W}$ gilt. Somit wird ein Global-As-View Mapping generiert.*

- *In der Zeile fünf wird $S_2.R_6$ mit $S_3.R_8$ in Beziehung gebracht. Man beachte, dass hier $\mathcal{R}_\mathcal{W} \subseteq \mathcal{R}_V$ ist und somit das schwache Containment-Mapping im Gegensatz zu den meisten anderen Fällen in eine andere Richtung definiert ist.*

5.4 Konstruktion der Datalog-Regeln

Bisher wurden lediglich die Teilziele (Relationen) der Kopf- und Rumpfseiten der Mappings durch Bildung von schwachen Containment-Mappings ermittelt. Der Output von Algorithmus 5.1 ist eine Menge von geordneten Paaren (H, T), wobei H der Kopfseite und T der Rumpfseite eines Mappings entsprechen. Aus diesen Informationen ist ein Schema-Mapping zu konstruieren, welches Attribute gleichen Ursprungs auf eine gemeinsame Variable abbildet. Hierzu wird für jedes Attribut $S.R_l.a_k$ aus dem Referenzschema

eine leere Attribut-Klasse $\mathcal{C}(S.R_l.a_k) := \emptyset$ erzeugt. Da die Relationen in H und T zu einem generierten Schema S_i bzw. S_j gehören und diese durch definierende Mappings einen Bezug zum Referenzschema haben, ist es möglich, jedes Attribut aus H und T der richtigen Klasse $\mathcal{C}$ zuzuordnen. Um Namenskonflikte zu vermeiden, werden den Attributen in H und T fortlaufende temporäre Variablennamen zugewiesen und diese dann den Klassen zugeordnet. In Abbildung 5.2 ist dies graphisch dargestellt:

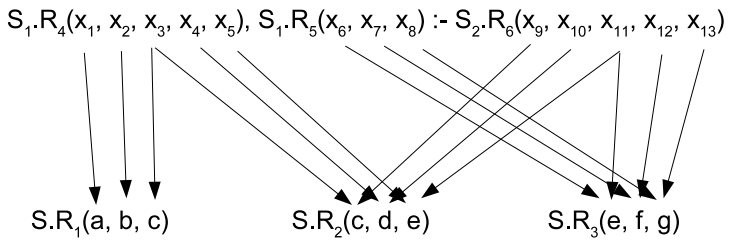


Abbildung 5.2: Zuweisung der Variablennamen anhand des Referenzschemas

Wurden alle Variablen mindestens einer Menge zugeordnet, werden alle nicht-disjunkten Mengen vereinigt, so dass am Ende nur noch disjunkte Klassen übrig bleiben. Alle Attribute innerhalb einer Klasse erhalten denselben Variablennamen in der erzeugten Datalog-Regel.

Beispiel 5.7 *Laut Abbildung 5.2 existieren folgende Mappings von Attributen auf Variablen:*

Attribut	Variablen
$S.R_{1.a}$	x_1
$S.R_{1.b}$	x_2
$S.R_{1.c}$	x_3
$S.R_{2.c}$	x_3, x_9
$S.R_{2.d}$	x_4, x_{10}
$S.R_{2.e}$	x_5, x_{11}
$S.R_{3.e}$	x_6, x_{11}
$S.R_{3.f}$	x_7, x_{12}
$S.R_{3.g}$	x_8, x_{13}

Tabelle 5.3: Attribut-Variable-Mapping

Alle nicht-disjunkten Variablenmengen-Paare werden vereinigt

$$\{x_3, x_9\} \cup \{x_3\} \text{ und } \{x_5, x_{11}\} \cup \{x_6, x_{11}\}.$$

und den gebildeten Klassen neue Variablennamen zugewiesen.

temporäre Variablen	neue Variable
x_1	y_1
x_2	y_2
x_3, x_9	y_3
x_4, x_{10}	y_4
x_5, x_6, x_{11}	y_5
x_7, x_{12}	y_6
x_8, x_{13}	y_7

Tabelle 5.4: Variablen-Variable-Mapping

Somit lautet die erzeugte Datalog-Regel:

$$S_1.R_4(y_1, y_2, y_3, y_4, y_5), S_1.R_5(y_5, y_6, y_7) :- S_2.R_6(y_3, y_4, y_5, y_6, y_7)$$

5.5 Zusammenfassung

Es wurden Mappings der Form $Q_1(S_1) \subseteq Q_2(S_2)$ generiert, welche dem LaV- oder GaV-Ansatz zuzuordnen sind. Auf der linken und rechten Seite eines Mappings stehen Anfragen an ein bestimmtes Schema. Diese Anfragen sind gemäß Definition 5.7 ineinander enthalten. Alle erzeugten Mappings existieren in Datalog-Notation und Attribute gleichen Ursprungs[3] sind über gleiche Variablennamen in Verbindung gebracht worden. So genannte *Equality-Mappings* $Q_1(S_1) = Q_2(S_2)$ resultieren in zwei Mappings $Q_1(S_1) \subseteq Q_2(S_2)$ und $Q_1(S_1) \supseteq Q_2(S_2)$. Die angelegten Mappings beziehen sich ausschließlich auf die Schemata des Schema-Pools und werden später gebraucht, um Peer-Graphen zu erzeugen. Erst dann werden die Mappings um Projektionen und Selektionen erweitert.

[3] das heißt, sie gehören zum gleichen Attribut aus dem Referenzschema

6 Erzeugung von Peer-Graphen

In der Praxis können Peer-Netze die unterschiedlichsten Topologien aufweisen. Es ist zu erwarten, dass sich gegebene Pruningstrategien (siehe [14]) für verschiedene Topologien unterschiedlich gut eignen. So liefert eventuell Strategie A für zyklische Graphen gute Ergebnisse und versagt bei baumartigen Graphen, bei denen Strategie B die besseren Resultate liefert. Die Testumgebung muss somit in der Lage sein, Graphen verschiedenen Typs zu generieren, um Pruningstrategien mit verschiedenen Graphen testen zu können. Nachdem ein Schema-Pool $\mathcal{S}$ (siehe Kapitel 4) und eine Menge $\mathcal{M}$ mit Schema-Mappings (siehe Kapitel 5) erzeugt wurden, wird in diesem Kapitel dargestellt, wie aus diesen Informationen ein Graph generiert werden kann. In Abbildung 6.1 wird verdeutlicht, wie, ausgehend von dem Schema-Pool und den Mappings, verschiedene Peer-Graphen erzeugt werden können.

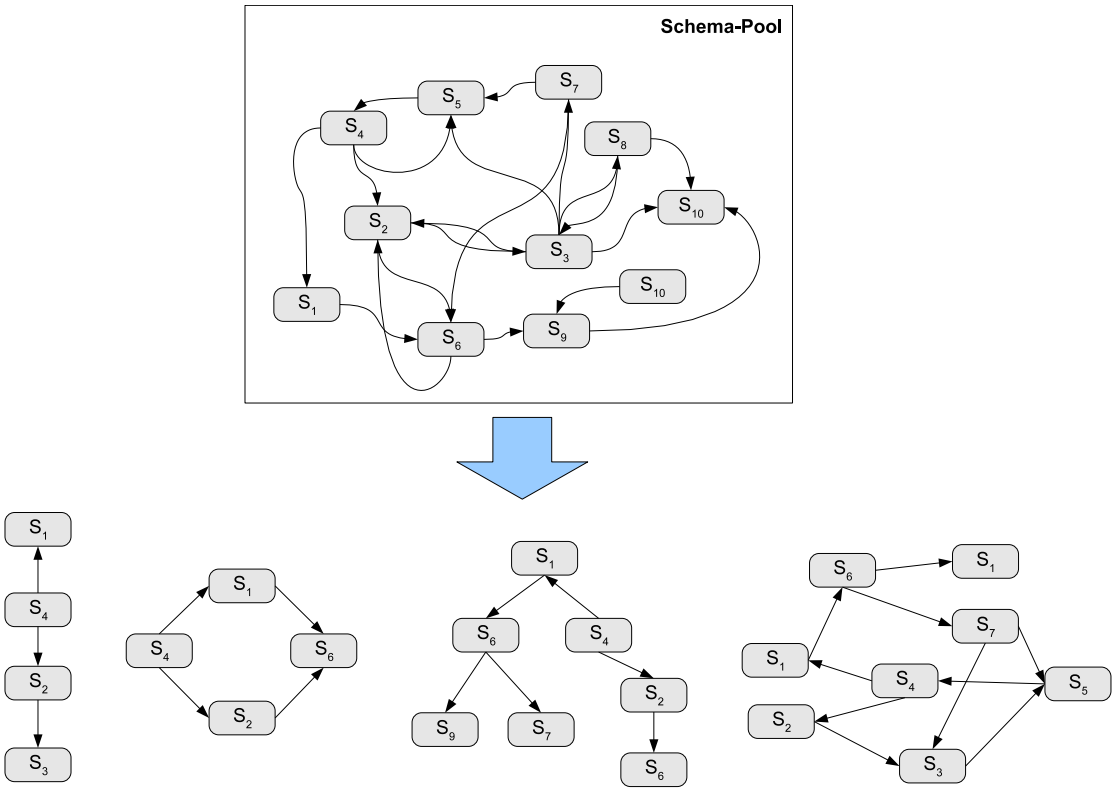


Abbildung 6.1: Graphische Darstellung des Generierungskonzeptes

Es werden verschiedene Graphentypen vorgestellt, Methoden, wie sich diese konstruieren lassen und welche Parameter hierfür notwendig sind.

6.1 Überblick

Ein endlicher gerichteter Graph ist ein Tupel $G = (V, E)$, wobei V eine endliche Menge und E eine Teilmenge der geordneten Paare $V \times V$ ist. Die Elemente $v \in V$ heißen die Knoten und die Elemente $e \in E$ die Kanten[1] von G. Ziel ist es, einen zusammenhängenden Graphen G zu erzeugen, dessen Knoten jeweils ein Schema aus dem Schema-Pool zugewiesen wird und dessen Kanten einem Schema-Mapping entsprechen. Nicht zusammenhängende Graphen sind für die gestellte Aufgabe nicht von Interesse, da zwischen den entstehenden Komponenten keine Mappings existieren.
Folgende Beziehungen existieren zwischen Schema-Pool $\mathcal{S}$, den generierten Mappings $\mathcal{M}$ und dem zu erzeugenden Graphen $G = (V, E)$:

- Eine Abbildung $h : V \to \mathcal{S}$ weist jedem Knoten $v \in V$ genau ein Schema $S \in \mathcal{S}$ des Schema-Pools zu.
- Eine Abbildung $f : E \to \mathfrak{P}(\mathcal{M})$ weist jeder Kante $e \in E$ eine Menge M von Mappings zu mit $M \subseteq \mathcal{M}$.
- In G dürfen nur Kanten $(v, u) \in E$ existieren, wenn zwischen den Schemata $S_v = h(v)$, $S_u = h(u)$ mit $S_v, S_u \in \mathcal{S}$ Mappings in $\mathcal{M}$ existieren.

Die Abbildungen h und f werden erst im Laufe der Graph-Erzeugung definiert. Wichtig ist, dass h im Allgemeinen keine Injektion ist und somit mehrere Knoten dasselbe Schema besitzen dürfen. Anders wäre eine Erzeugung großer Peer-Netze bei einem kleinen Schema-Pool nicht möglich. Die Abbildung f bildet auf die Potenzmenge von $\mathcal{M}$ ab, da zwischen zwei Peers im Allgemeinen mehrere Mappings definiert sind.

Es wurden Algorithmen für die Graphentypen Kette, Kreis, Baum und zufälliger Graph entwickelt. Jeder dieser Algorithmen hat den Schema-Pool $\mathcal{S}$, die Mappings $\mathcal{M}$ und einige Einschränkungen C für den Graphen als Eingabeparameter. Zu den Einschränkungen gehört für alle Graphenarten die Kardinalität der zu erzeugenden Knotenmenge. Je nach Graphenart kommen weitere geforderte Eigenschaften, wie zum Beispiel Kreisfreiheit und Maximalgrad hinzu. Resultat eines jeden Algorithmus sind stets ein endlicher, zusammenhängender und gerichteter Graph $G = (V, E)$ sowie die eindeutigen Abbildungen $h : V \to \mathcal{S}$ und $f : E \to \mathfrak{P}(\mathcal{M})$. Ausführliche Beschreibungen sowie weiterführende Definitionen zum Thema Graphen sind [17] und [15] zu entnehmen.

6.2 Standard-Graphen

Für spezielle Experimente können einfach strukturierte Graphen sinnvoll sein. In diesem Abschnitt werden Ansätze vorgestellt, wie derartige Standard-Graphen erzeugt werden können. Sämtliche hier vorgestellten Graph-Erzeugungs-Algorithmen sind nicht deterministisch, da sie für die Definition der Abbildung $h : V \to \mathcal{S}$ Gebrauch von einem Zufallsgenerator machen.

[1] Kanten mit gleichem Start- und Endknoten werden nicht betrachtet.

6.2.1 Kette

Ein Kettengraph oder auch Pfadgraph hat zwei ausgezeichnete Endknoten, ist kreisfrei und besitzt den Maximalgrad 2. Die beiden Endknoten haben den Grad 1. Wir unterscheiden zwei Varianten des Graphen: Die erste Variante (a) schränkt die Wahl der Kantenrichtungen nicht ein. Es muss somit kein Pfad in diesem gerichteten Graphen zwischen den Endknoten existieren. Die zweite Variante (b) garantiert die Erreichbarkeit des Endknotens von einem Startknoten aus. In Abbildung 6.2a ist die erste und in Abbildung 6.2b die zweite Variante dargestellt.

Anwendungsfall

Anhand eines Kettengraphen lassen sich Basisfunktionen der Anfragebearbeitung testen. Für den betrachteten *Pruning-Algorithmus* (siehe [14]) steht stets genau ein Mapping zur Auswahl. Somit können Budget-Berechnungen und Auswirkungen von Selektionen sowie Projektionen leichter untersucht werden als in komplexen Graphen, da die Berechnungen einfach nachzuvollziehen sind. Auch die maximale Anfragetiefe ist bei Kettengraphen durch die Anzahl der Knoten bestimmt.

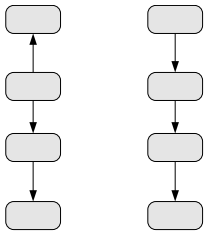


Abbildung 6.2: Ein gerichteter Kettengraph in zwei Varianten

Parameter

Die für die Erstellung eines Kettengraphen erforderlichen Parameter sind die Anzahl der zu erzeugenden Knoten und die Wahl zwischen den vorgestellten Varianten (a) und (b).

Algorithmus

Initial sind $V = \emptyset$ und $E = \emptyset$. Für die Erstellung des Graphen wird ein zufälliges Schema $S^* \in \mathcal{S}$ als Startknoten ausgewählt. Ausgehend von diesem wird versucht, eine Kette der geforderten Länge zu erzeugen. Für das Startschema wird ein Knoten $v \in V$ eingefügt und die Abbildung h erweitert durch $h(v) := S^*$. Anschließend wird nach Mappings $M \in \mathcal{M}$ gesucht, welche das Schema S^* enthalten. Je nachdem, welche Kettenvariante erzeugt werden soll, ist vorgegeben, auf welcher Seite des Mappings das Schema S^* existieren muss. Für Variante (a) spielt die Seite keine Rolle, für Variante (b) muss S^* auf der Kopfseite eines Mappings vorkommen. Aus der Menge der gefundenen Mappings für das Schema S^* wird eines zufällig gleichverteilt ausgewählt. An diesem Mapping ist neben

S^* ein anderes Schema S° beteiligt. Ein weiterer Knoten u wird in V eingefügt und die Abbildung h mit $h(u) := S^\circ$ erweitert. Je nach Richtung des Mappings wird eine Kante $e = (v, u)$ oder $e = (u, v)$ in die Menge E eingefügt. Der Vorgang wird nun rekursiv mit S° fortgesetzt und bricht ab, sobald der Graph die geforderte Anzahl von Knoten hat. Führt die Auswahl einer Kante dazu, dass keine Kette der geforderten Länge gefunden werden kann, so werden via *Back-Tracking* andere Lösungswege gesucht.

6.2.2 Kreis

Ein Kreisgraph hat den Minimal- und Maximalgrad 2 und ist zyklisch. Wir unterscheiden zwischen zwei Kreisarten. In der ersten Kreisvariante existiert ein gerichteter Weg, dessen Start- und Zielknoten identisch sind, in der zweiten Variante ist dies nicht der Fall.

Anwendungsfall

Zyklische Graphen eignen sich, um die Kreiserkennung während der Anfragebearbeitung zu testen. Besonders kontrolliert gelingt dies, wenn der Graph selbst ein Kreis bzw. Ring ist und keine Einflüsse anderer Mappings auftreten, welche aus dem Kreis hinausführen.

Parameter

Die Parameter für die Erzeugung eines Kreises sind identisch mit denen der Kette. Die Anzahl der Knoten und die Variante (siehe Abbildung 6.3) stehen auch hier zur Wahl.

Algorithmus

Der Algorithmus zum Erzeugen eines Kreises arbeitet analog zur Kettengraph-Erzeugung. Es wird eine Kette der geforderten Länge erzeugt und geprüft, ob sich beide Endknoten über eine Kante verbinden lassen. Dies ist genau dann der Fall, wenn in der Menge $\mathcal{M}$ ein Mapping zwischen den zugehörigen Schemata existiert.

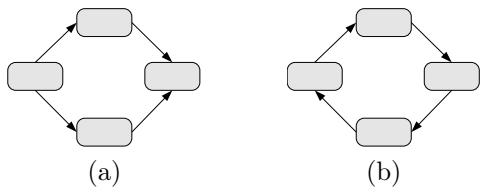


Abbildung 6.3: Ein gerichteter Kreis in zwei Varianten

Andernfalls wird die Rekursion abgebrochen und neue Pfade werden gesucht. Laufzeit und Ergebnis sind nicht deterministisch, der erzeugte Graph besitzt jedoch in jedem Fall die geforderten Eigenschaften.

Man beachte, dass in Variante (a) keine endlosen Wege existieren und nicht jeder Knoten von jedem anderen erreicht werden kann. Zum Testen der Zyklenerkennung (siehe [14]) während der Anfragebearbeitung eignet sich daher nur Variante (b).

6.2.3 Baum

Ein Baum ist ein azyklischer, zusammenhängender Graph. Der zu erzeugende Graph ist auch hier gerichtet. Es wird zwischen drei verschiedenen Baumtypen unterschieden:

1. Der Baum hat keine ausgezeichnete Wurzel und die Richtung der Kanten wird zufällig bestimmt.

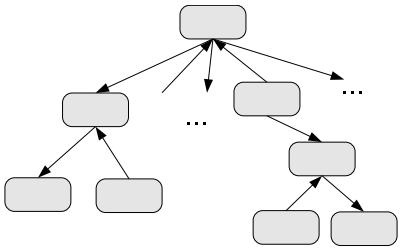
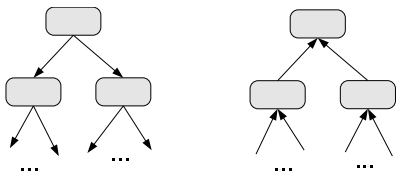


Abbildung 6.4: Gerichteter Baumgraph ohne ausgezeichnete Wurzel

2. Es ist ein ausgezeichneter Wurzelknoten vorhanden. Diese Baumart nennen wir *Wurzelbaum* und unterscheiden zwischen zwei Varianten:

   a) *Out-Trees* haben die Eigenschaft, dass von der Wurzel aus jeder Knoten durch genau einen gerichteten Pfad erreichbar ist. Die Kanten zeigen somit alle in Richtung der Blätter.

   b) *In-Trees* sind entgegengesetzt zu *Out-Trees* orientiert. Hier ist die Wurzel von jedem anderen Knoten durch genau einen gerichteten Pfad erreichbar. Die Kanten zeigen hier alle zur Wurzel hin.



(a) Out-Tree (b) In-Tree

Abbildung 6.5: Zwei spezielle Baumvarianten in gerichteten Graphen

Anwendungsfall

Über Bäume lassen sich Pruningstrategien unabhängig von der Kreiserkennungsproblematik testen. Für den Anfrageplaner stehen in Bäumen im Allgemeinen mehrere Mappings zur Auswahl. Wird die Implementierung in [14] zu einem späteren Zeitpunkt um die parallele Anfragebearbeitung erweitert, so kann diese anhand von Bäumen gut getestet werden. Jeder Peer fragt in diesem Fall seine Baumkinder zeitgleich an und wird selbst wegen der Zyklenfreiheit nur ein einziges Mal angefragt.

Parameter

Als Erzeugungsparameter sind die maximale Anzahl $|V|$ von Knoten, die minimale Ordnung *minOrd* die maximale Ordnung *maxOrd* und die Baumvariante gegeben. Die Ordnung gibt an, wie viele Verzweigungen ein Knoten haben darf, die Variante bezieht sich auf die oben vorgestellten Varianten 1) bzw. 2a) und 2b).

Algorithmus

Ein Schema $S^* \in \mathcal{S}$ wird dem Startknoten $v \in V$ zugewiesen. Es wird zufällig gleichverteilt eine Ordnung m zwischen *minOrd* und *maxOrd* gewählt und somit m Kinder für den Knoten v erzeugt. Die Kantenrichtungen werden der gewählten Baumvariante entsprechend gewählt. Der Vorgang wird für jedes erzeugte Kind wiederholt, wobei das Kind die Rolle einer Wurzel übernimmt und selbst zum Elternknoten wird. Der Aufbau des Baumes erfolgt in *Breadth-First* Reihenfolge und ist nicht rekursiv implementiert. Die Iteration bricht ab, sobald der Baum die geforderte Anzahl Knoten aufweist.
Die in Algorithmus 6.1 beschriebene Funktion *CreateTree()* wird initial mit den Parametern

- $\mathcal{S}$ Der Schema-Pool
- $\mathcal{M}$ Die Mappings
- $maxNodeCount$ Anzahl zu erzeugender Knoten
- *minOrd* minimale Ordnung und *maxOrd* maximale Ordnung zur Festlegung der Anzahl der Kinder eines inneren Knotens
- t Der Typ des Baums. In Frage kommen: Zufällig gerichtet, In-Tree und Out-Tree.

aufgerufen. Der Aufruf erzeugt zunächst die Baumwurzel und m Kinder mit $minOrd \leq m \leq maxOrd$. Die Funktion zur Erzeugung der Kinder ist in Algorithmus 6.2 dargestellt. Sind alle Kinder erzeugt, werden in einer Schleife für diese Kindknoten angelegt.

Function *CreateTree()*

Input : Schema-Pool $\mathcal{S}$, Mappings $\mathcal{M}$, Knotenanzahl c, min. Ord. $minOrd$, max. Ord. $maxOrd$, Baumtyp t
Output : Baum $G = (V, E)$

$w \leftarrow$ Neue Wurzel;
$V \leftarrow \{w\}$;
$E \leftarrow \emptyset$;
`/* Menge von Knoten, für die Kinder erzeugt werden sollen (Anfangs ist dies lediglich die oben erzeugte Wurzel) */`
$R \leftarrow \{w\}$;
$c \leftarrow (c - 1)$;
repeat
  $R' \leftarrow \emptyset$;
  foreach $v \in R$ **do**
    $childCount \leftarrow$ Zufälliger Wert zwischen $minOrd$ und $maxOrd$;
    if $childCount > c$ **then** $childCount \leftarrow c$;
    $C \leftarrow$ `CreateChildren(`$\mathcal{S}, \mathcal{M}, v, childCount, t$`)`;
    $V \leftarrow V \cup C$;
    $E \leftarrow E \cup \{$ Kanten zwischen v und den Knoten in $C\}$;
    `/* Merke erzeugte Kinder dieser Ebene */`
    $R' \leftarrow R' \cup C$;
    `/* Verringere Knotenbudget` c `*/`
    $c \leftarrow (c - |C|)$;
  end
  `/* In der nächsten Iteration werden für die Knoten aus` R' `Kinder erzeugt */`
  $R \leftarrow R'$;
until $c = 0$;
return (V, E);

Algorithmus 6.1 : Erzeugung eines Baumes

Function *CreateChildren()*

Input : Schema-Pool $\mathcal{S}$, Mappings $\mathcal{M}$, Wurzel w, Anzahl zu erzeugender Kinder c, Baumtyp t

Output : Menge von Kinderknoten C

```
C ← ∅;
for i = 1 to c do
    /* Liste mit möglichen Kindern                                    */
    P ← ∅;
    if t =Zufällig gerichtet then
        if RandomBool = true then
            /* Es werden alle Mappings in M durchsucht, welche auf der
               Kopfseite das dem Knoten w zugewiesene Schema besitzen.
               Aus diesen werden die Schemata der Rumpfseite
               zurückgegeben.                                         */
            P ←FindChildrenHeadMappings(S, M, w)
        else
            /* Wie FindChildrenWithHeadMappings, nur dass die Mappings
               auf der Rumpfseite das w zugeordnete Schema besitzen
               müssen und die Kopfschemata zurückgeliefert werden.    */
            P ← FindChildrenTailMappings(S, M, w)
        end
    else
        if t = Out-Tree then
            P ← FindChildrenTailMappings(S, M, w)
        else
            P ← FindChildrenHeadMappings(S, M, w)
        end
    end
    p ← zufälliges Element aus P;
    C ← C ∪ {p};
end
return C;
```

Algorithmus 6.2 : Erzeugung eines Baumes (Kindknoten)

6.3 Zufälliger Graph

Ein zufälliger Graph folgt keiner vorgegebenen Topologie und kann nur durch einige graphentypische Parameter spezifiziert werden. In diesem Abschnitt werden die Markov-Ketten vorgestellt und es wird gezeigt, wie sich anhand dieser Zufallsgraphen generieren lassen. In Kapitel 9 werden einige Experimente mit dem implementierten Algorithmus dargestellt.

6.3.1 Stochastische Prozesse und Markov-Ketten

Definition 6.1 (Stochastischer Prozess)
Ein stochastischer Prozess oder Zufallsprozess ist eine Folge von elementaren Zufallsereignissen $X_1, X_2, \ldots, X_i \in \Omega$, $i \in \mathbb{N}$ in einem Ereignisraum Ω. Auf diesem ist das Wahrscheinlichkeitsmaß $P : \Omega \to [0, 1]$ definiert.

Definition 6.2 (Markov-Kette)
Die möglichen Zufallswerte in einem stochastischen Prozess nennt man *Zustände* des Prozesses. Eine *Markov-Kette* ist ein stochastischer Prozess mit folgenden Eigenschaften:

- Eine endliche Markov-Kette besteht aus einer endlichen Anzahl von Zuständen und befindet sich zu jedem Zeitpunkt in einem dieser Zustände. Die Menge der Zustände sei hier mit $Z = \{z_1, \ldots, z_n\}$ angegeben.
- Zu jedem Zeitpunkt hängen die Wahrscheinlichkeiten aller zukünftigen Zustände nur vom momentanen Zustand ab. Diese Eigenschaft wird auch als *Markov-Eigenschaft* bezeichnet. Ein derartiger Prozess "vergisst" vorherige Ereignisse. Es existieren Übergangswahrscheinlichkeiten $p : Z \times Z \to [0, 1]$, die angeben, wie wahrscheinlich es ist, dass der Prozess vom Zustand $z_i \in Z$ in den Zustand $z_j \in Z$ wechselt.

Man findet in der Literatur (siehe [15]) oft noch die Angabe einer Startverteilung für Markov-Ketten, die festlegt, wie wahrscheinlich es ist, dass die Markov-Kette mit einem Zustand $z_i \in Z$ startet. Wir werden sehen, dass diese in unserem Fall trivial ist und vernachlässigt werden kann, da wir einen Startzustand vorgeben.

6.3.2 Konstruktion einer Markov-Kette

Wie in [5] wollen wir Markov-Ketten einsetzen, um einen Graphen sukzessiv zu verändern. Sei eine Menge C von geforderten Eigenschaften für den zu erzeugenden Graphen gegeben. Zudem sei ein Graph G_i vorhanden, der bereits die Anforderungen aus C erfüllt. Die Idee ist es, den Graphen G_i mit Hilfe eines kleinen Prozesses minimal zu verändern, so dass der resultierende Graph G_j immer noch die Eigenschaften aus C erfüllt. Der Prozess soll stochastisch sein, also eine zufällige Veränderung vornehmen und mehrmals angewendet werden können, um den Graphen stärker zu verändern. Entscheidend hierbei ist, dass

der Prozess als Eingabe einen Graphen erhält, der C genügt und als Ausgabe immer einen Graphen liefert, der C ebenfalls erfüllt. Die zugehörige Markov-Kette wird wie folgt modelliert:

- Es ist eine Menge C von Einschränkungen für den zu erzeugenden Graphen gegeben:
  - Es kann der Maximalgrad $2 \leq m < \infty$ für den Graphen bestimmt werden. Dabei erfolgt die Unterscheidung zwischen einem einheitlichen Maximalgrad, der für alle Knoten gilt und einem individuellen Grad für jeden Knoten. Bei der letzten Option wird für jeden Knoten anfangs ein Maximalgrad zwischen 2 und m zufällig und gleichverteilt gewählt.
  - Es können wahlweise gerichtete Zyklen erlaubt oder verboten werden. Sind Zyklen erlaubt, so heißt dies nicht zwingend, dass der resultierende Graph zyklisch ist.
  - Der erzeugte Graph muss eine Zusammenhangskomponente ergeben.
- Es ist ein einfacher Graph G_0 gegeben, der die Bedingungen aus C erfüllt.
- Ein stochastischer Prozess P wird definiert, welcher in der Lage ist, ausgehend vom Startgraphen G_0 eine endliche Menge $\mathcal{G} = \{G_0, \ldots, G_n\}$ von Graphen zu erzeugen. Diese Menge $\mathcal{G}$ entspricht der Zustandsmenge unserer Markov-Kette.
- Die Übergangswahrscheinlichkeiten zwischen den Zuständen aus $\mathcal{G}$ werden durch die Definition des Prozesses P bestimmt.

Unter gerichteten Zyklen verstehen wir solche, in denen endlose gerichtete Wege möglich sind (siehe Abbildung 6.3b). Weitere Parameter sind die geforderte Anzahl von Knoten $maxNodeCount$ und die Anzahl der Markov-Iterationen $maxIterationCount$. Als einfacher Startgraph kann zwischen Kette und Baum gewählt werden (siehe Abschnitte 6.2.1 und 6.2.3). Der Startgraph wird mit der geforderten Knotenzahl erzeugt. Da wir für den Maximalgrad $m \geq 2$ vorgeben und sowohl Kette als auch Baum kreisfrei sind, erfüllt der gewählte Startgraph alle Anforderungen. Der Prozess P zur Veränderung eines Graphen ist wie folgt definiert:

Definition 6.3 (Markov-Prozess zur Erzeugung von Zufallsgraphen)
Seien $v, u \in V$ zwei zufällig und gleichverteilt gewählte Knoten aus der Knotenmenge V des Graphen $G_i = (V, E_i)$ und die Funktion $h : V \to \mathcal{S}$ gegeben, dann:

(i) Wenn eine Kante $(u, v) \in E_i$ existiert und der Graph nach der Entfernung dieser Kante noch eine Zusammenhangskomponente bildet, dann gib $(V_i, E_i \setminus \{(u, v)\})$ als neuen Graphen zurück.

(ii) Wenn keine Kante $(u, v) \in E_i$ existiert, dann prüfe, ob ein Mapping zwischen den Schemata $h(u)$ und $h(v)$ in $\mathcal{M}$ existiert. Ist dies der Fall, so wird geprüft, ob bei Einfügen der Kante (u, v), der Maximalgrad m des Graphen bzw. die erlaubten Grade der

beteiligten Knoten nicht überschritten werden. Darf der Graph keine echten Zyklen enthalten, so muss auch diese Eigenschaft nachgeprüft werden. Sind die Prüfungen erfolgreich, so gib $(V_i, E_i \cup \{(u, v)\})$ zurück.

(iii) Wurden weder bei (i) noch bei (ii) Veränderungen am Graphen durchgeführt, so gib den unveränderten Graphen (V_i, E_i) zurück.

Bemerkung: Der hier vorgestellte Prozess verändert nicht die Knotenmenge V, sondern ausschließlich die Menge der Kanten E. Eine Überprüfung, ob der Graph die geforderte Anzahl Knoten besitzt, kann somit wegfallen.

Beispiel 6.1 *Es soll ein zusammenhängender, zufälliger Graph mit Maximalgrad $m = 4$ und $|V| = 7$ Knoten erzeugt werden. Der Graph darf zyklisch sein und der Markov-Prozess P wird 20 mal hintereinander ausgeführt. Der Startgraph G_0 ist eine Kette, wurde durch den in Abschnitt 6.2.1 vorgestellten Algorithmus erzeugt und ist in Abbildung 6.6a zu sehen. In den 20 Iterationen fanden insgesamt sieben Operationen statt:*

Abbildung	Knotenauswahl	Operation
(b)	$(v, u) = (S_1, S_4)$	Kante einfügen
(c)	$(v, u) = (S_1, S_2)$	Kante entfernen
(d)	$(v, u) = (S_6, S_3)$	Kante einfügen
(e)	$(v, u) = (S_4, S_5)$	Kante entfernen
(f)	$(v, u) = (S_3, S_5)$	Kante einfügen
(g)	$(v, u) = (S_5, S_1)$	Kante einfügen
(h)	$(v, u) = (S_5, S_7)$	Kante einfügen

Tabelle 6.1: Durchgeführte Operationen am Graphen

13 Iterationsschritte führten zu keiner Veränderung des Graphen, da entweder der Maximalgrad überschritten wurde, kein Mapping zwischen den Knotenschemata existiert oder der resultierende Graph keine einzige Zusammenhangskomponente mehr bilden würde. In Abbildung 6.6 sind die Veränderungen der verbleibenden Schritte am Graphen illustriert.

Bemerkung: Der letzte Graph (h) erfüllt die geforderten Eigenschaften (Maximalgrad $m = 4$) und weist einen echten Kreis auf ($S_1 \to S_4 \to S_3 \to S_5 \to S_1$).

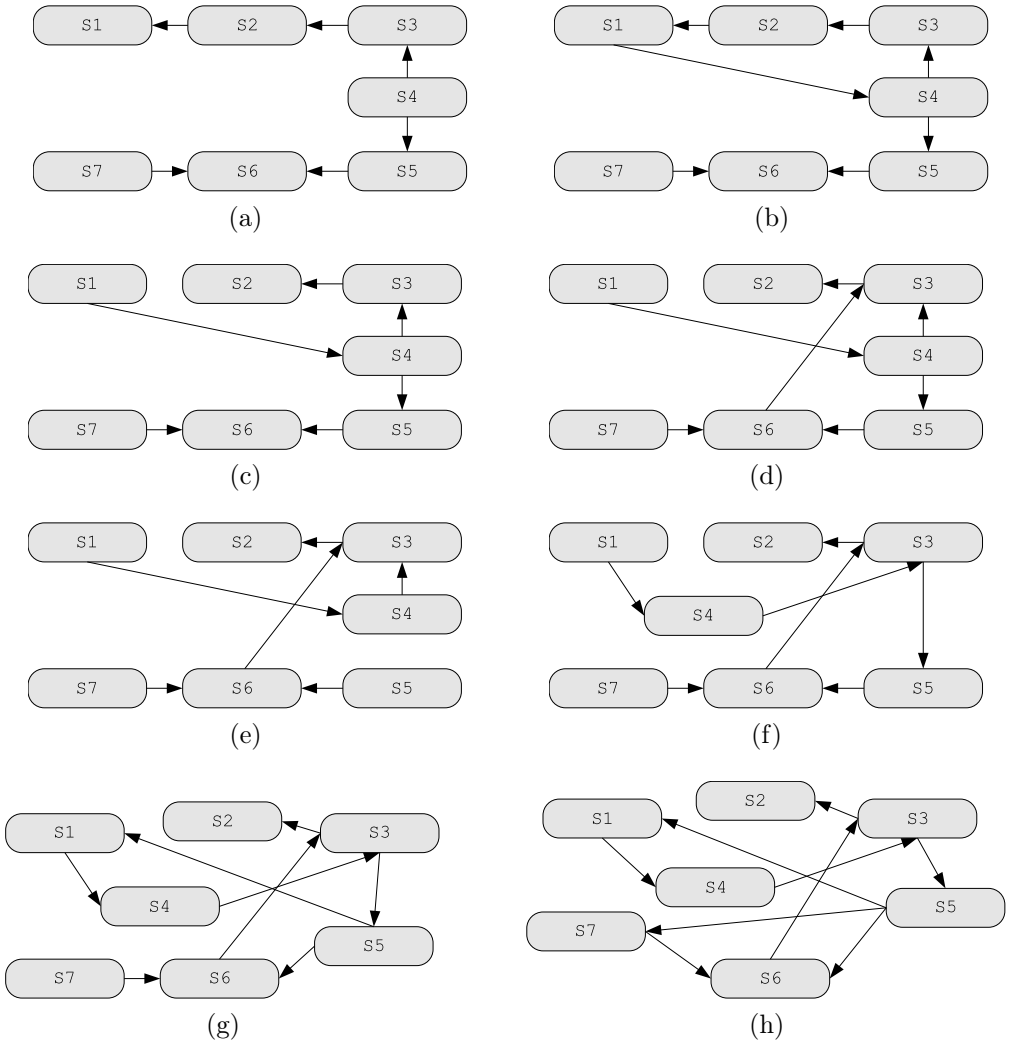


Abbildung 6.6: Verschiedene Graphen als Zustände des Markov-Prozesses

6.3.3 Eigenschaften der konstruierten Markov-Kette

Im vorherigen Abschnitt wurde eine Markov-Kette konstruiert, die anhand eines Schema-Pools $\mathcal{S}$ und eine Menge möglicher Mappings $\mathcal{M}$ einen zufälligen Graphen erzeugen kann. Dabei stellt sich die Frage, ob durch den Algorithmus alle erdenklichen Graphen, welche die geforderten Eigenschaften erfüllen, gebildet werden können. Im nachfolgenden Satz wird die Annahme getroffen, dass der Schema-Pool einen vollständigen Graphen repräsentiert. Wie das zugehörige Experiment aus Abschnitt 9.1 zeigt, trifft dies in der Praxis in den meisten Fällen zu.

Satz 6.1 (Menge erzeugbarer Graphen der konstruierten Markov-Kette)
Sei $\mathcal{S}$ ein vollständiger Schema-Pool und $\mathcal{M}$ die zugehörige Menge mit Schema-Mappings. Es seien weiter ein zusammenhängender Graph $G = (V, E)$, die Abbildungen $h : V \to \mathcal{S}$ und $f : E \to \mathcal{M}$ gegeben, welche die Bedingungen aus Abschnitt 6.1 erfüllen. Dann ist

der Graph G durch die Markov-Kette aus Definition 6.3 erzeugbar.

Beweis. Da G zusammenhängend ist, existiert ein spannender Baum $B \subseteq G$. Zu diesem Baum gelangt man durch sukzessives Entfernen einzelner Kanten. An dieser Stelle wird zwischen den beiden möglichen Startgraphen unterschieden:

(i) Der Startgraph ist ein Baum: Der spannende Baum B kann mit dem Algorithmus 6.1 erzeugt werden: Sei $w \in B$ als Wurzel gegeben, so können dessen Kinder zufällig gewählt worden sein, da laut Voraussetzung Mappings zwischen den Schemata möglich sind. Setzt man dies rekursiv fort, ist gezeigt, dass B erzeugbar ist.

(ii) Der Startgraph ist eine Kette: Wählt man aus B einen Knoten als Wurzel und hängt an das am weitesten links liegende Blatt schrittweise die übrigen Blätter an, so lässt sich der geforderte Startgraph konstruieren. Der Algorithmus hierzu wird exemplarisch in Beispiel 6.2 erklärt.

Es wurden mögliche Startgraphen (Baum, Kette), ausgehend von G, konstruiert und gezeigt, dass die Startgraphen immer erzeugbar sind. Wird das Löschen der Kanten aus G, um die Startgraphen zu konstruieren, rückgängig gemacht, so sind die dabei entstehenden Kanten-Einfüge-Operationen mögliche Zustandsübergänge des Markov-Prozesses. Der Graph ist zu jedem Zeitpunkt zusammenhängend, sofern erforderlich zyklenfrei (falls G zyklenfrei ist) und erfüllt alle Bedingungen an die Grade der Knoten. Somit ist gezeigt, dass der Markov-Prozess den Graphen G konstruieren kann, sofern der zufällige Prozess genau diese Kantenoperationen auswählt. □

In Kapitel 9 werden mit dem vorgestellten Markov-Prozess einige Experimente durchgeführt, die Aufschluss über die Beschaffenheit der erzeugten Graphen geben.

Beispiel 6.2 *In Abbildung 6.7a ist ein spannender Baum B gegeben. Aus diesem kann mit einem einfachen Algorithmus eine Kette konstruiert werden. Es werden zu jedem Zeitpunkt Eigenschaften, wie Zusammenhang des Graphen, Maximalgrad und Zyklizität beachtet.*

An das am weitesten links liegende Blatt l wird in jedem Schritt eines der übrigen Blätter b angehängt. Dazu wird zuerst eine Kante zwischen l und b eingefügt (grüne Kanten) und anschließend das Blatt b von seinem Elternknoten getrennt (rote Kanten). Der Zusammenhang des Graphen ist somit gewahrt. Da der erlaubte Maximalgrad für den Graphen und jeden einzelnen Knoten mindestens 2 beträgt und die Knoten l sowie b Blätter sind, führt das Einfügen einer Kante zwischen Knoten zu einem Grad von 2 für b und l. Somit werden auch etwaige Forderungen an die Grade der Knoten zu keinem Zeitpunkt verletzt. Durch Steuerung der Kantenrichtung kann auf diesem Wege auch gewährleistet werden, dass der Graph zu jedem Zeitpunkt azyklisch ist und somit keine gerichteten Kreise enthält. Nachdem b an das Blatt l angehängt wurde, übernimmt der Knoten b selbst die Rolle des ursprünglichen Blattes l. Der Vorgang wird solange wiederholt, bis der Graph eine Kette ist. Eine vollständige Umwandlung für einen Baum ist in Abbildung 6.7 zu sehen.

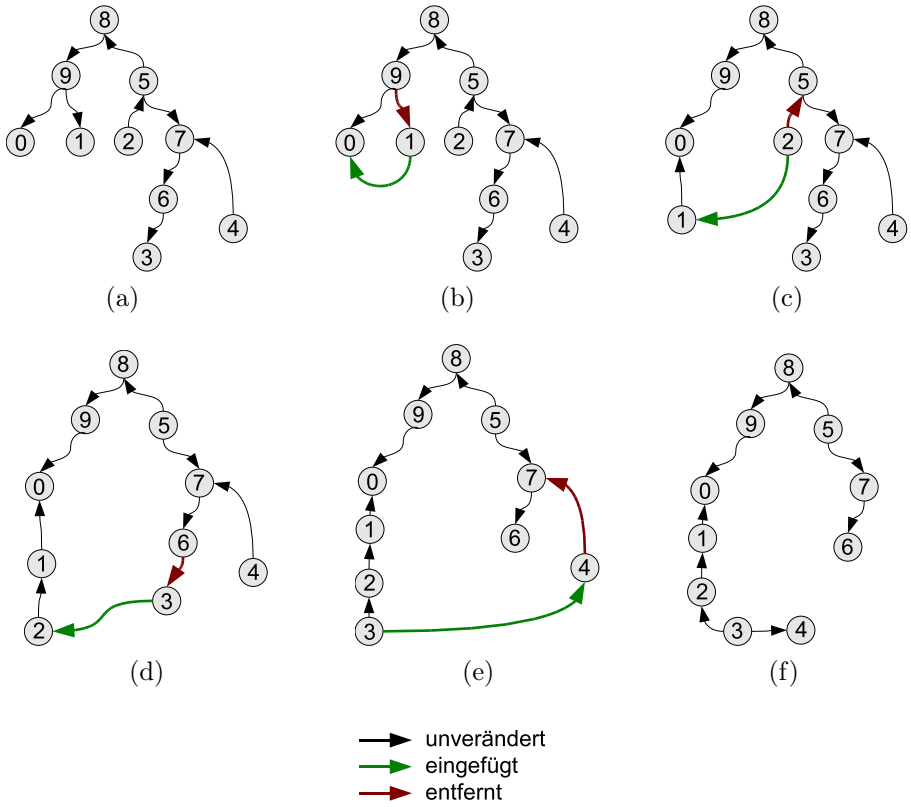


Abbildung 6.7: Umwandlung eines Baumes in eine Kette

Die Kette kann somit durch den Markov-Prozess aus Definition 6.3 in einen Baum überführt werden, sofern obige Kantenoperationen in umgekehrter Reihenfolge von dem verwendeten Zufallsgenerator erzeugt werden.

6.4 Weitere interessante Graphentypen

In diesem Abschnitt werden weitere Graphentypen vorgestellt, die leider in der Implementierung[2] für diese Arbeit nicht mehr berücksichtigt werden konnten.

6.4.1 Graphen mit Bottlenecks

In der Praxis haben viele Netzwerke oft so genannte Flaschenhälse (engl. *bottlenecks*). Gründe hierfür sind oftmals Einsparungen der Kosten beim Ausbau des Netzes oder unüberwindbare Hindernisse. Im Internet treten Bottlenecks (siehe [1]) beispielsweise bei Verbindungen zwischen verschiedenen ISPs[3] auf. Über derartige Verbindungen läuft ein

[2] Wie sich weitere Graphen in das System integrieren lassen, wird in Kapitel 8 detailliert anhand der Implementierung erklärt.

[3] Internet Service Provider, zu deutsch: Internetdienstanbieter

Großteil des Netzverkehrs ab, weshalb sich Ausfälle hier besonders stark bemerkbar machen.

Für den Graphen, der die Netzwerk-Topologie beschreibt, heißt das, dass er bei Ausfall eines derartig wichtigen Verbindungsknotens eventuell nicht mehr zusammenhängend ist. Knoten, bei deren Entfernung ein Graph in mehrere Komponenten zerfällt, heißen Artikulationsknoten. Ein Graph mit mehreren Artikulationsknoten ist in Abbildung 6.8 dargestellt.

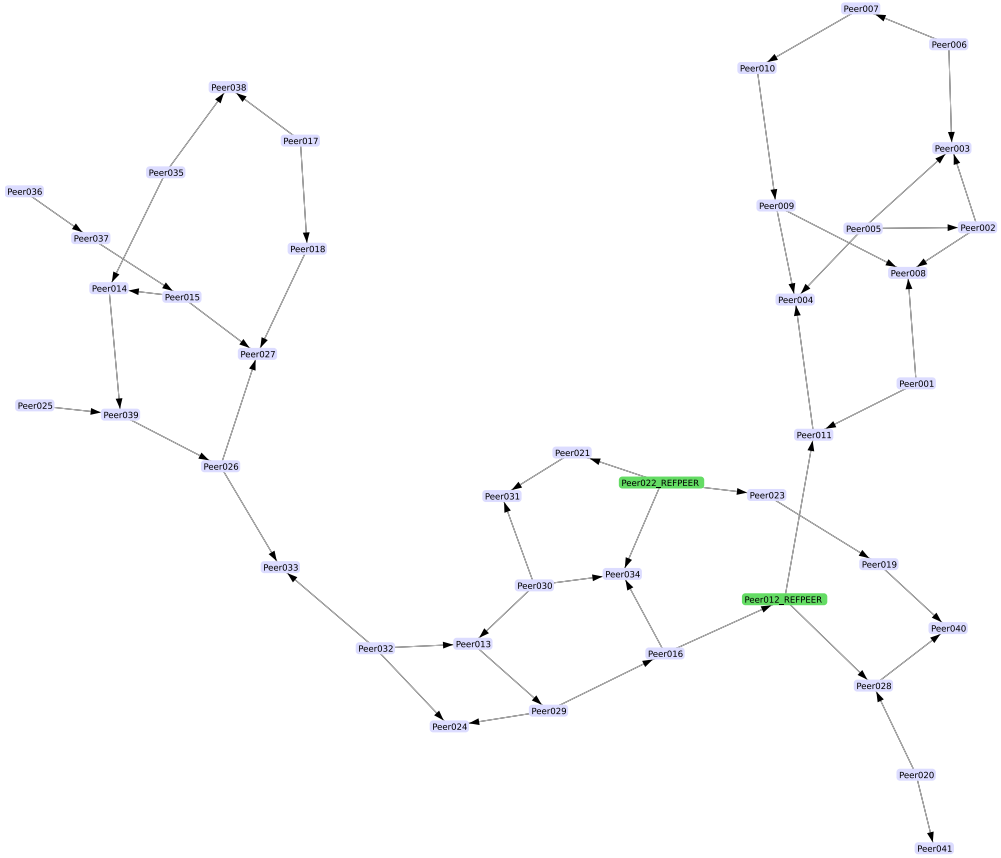


Abbildung 6.8: Graph mit mehreren Artikulationsknoten

Für ein PDMS ist eine Bottleneck-Topologie ebenfalls denkbar. Ein typisches Szenario ist, dass zwei bereits vorhandene Peer-Netze über einen vermittelnden Peer zusammengeschlossen werden. Im Laufe der Zeit kann sich auf diese Weise ein PDMS mit mehreren Bottlenecks oder sogar Artikulationsknoten herausbilden. Da ein Großteil der Kommunikation über die vermittelnden Peers (auch Mediator-Peers genannt) ablaufen wird, kann der Ausfall eines dieser ausgezeichneten Peers starke Auswirkungen auf die Anfragebeantwortung haben.

Die Erzeugung eines derartigen Graphen könnte auf folgende zwei Arten geschehen:

- Es werden, wie in der Realitität auch, unabhängig mehrere Peer-Netze generiert. Damit die Mappings möglich sind, müssen die Peer-Netze das gleiche Referenzsche-

ma als Grundlage besitzen. Zwischen den erzeugten Peer-Netzen wird dann nach möglichen Mappings gesucht und diese werden an geeigneten Stellen eingefügt.

- Man erzeugt einen Basisgraphen, der die Grundtopologie festlegt. Um jeden Knoten des Basisgraphen herum wird ein eigenständiges isoliertes PDMS erzeugt, welches nur mit dem zugehörigen Basisknoten verbunden ist. Dadurch entstehen ebenfalls mehrere unabhängige Peer-Netze, welche aber in gewählter Topologie vernetzt sind.

Anhand der oben vorgestellten Peer-Graphen lässt sich das Verhalten unterschiedlicher Pruning-Algorithmen bei Ausfall eines oder mehrerer Bottleneck-Peers analysieren. In Kapitel 9 wird der Markov-Prozess aus Abschnitt 6.3 auf Eignung zur Erzeugung von Artikulationsknoten untersucht.

6.4.2 Bow-Tie-Struktur des Web

Die Struktur des *World Wide Web* hat nach [3] (Broder et al.) die Form einer Fliege[4] aufgrund der Verlinkung seiner Dokumente. Das Web besteht aus einem stark verbundenen Kern (SCC - Strongly Connected Core), einem Bereich, der auf den Kern verweist (IN) und einem Bereich auf den vom Kern aus verwiesen wird (OUT). Durch so genannte Schläuche (eng. *tubes*) werden vereinzelt IN- und OUT-Bereiche miteinander verbunden. Des Weiteren existieren so genannte Ranken (eng. *tendrils*), die zwar eine Verbindung zu einem der drei großen Bereiche IN, SCC und OUT aufweisen, jedoch insgesamt stark isoliert sind. Zuletzt existieren noch Komponenten, die nicht mit einem der vier großen Bereiche verbunden sind. Die Struktur des Bow-Tie-Graphen ist in Abbildung 6.9 dargestellt.

Ein PDMS könnte eine ähnliche Topologie aufweisen, wenn man die nicht verbundenen Komponenten außer Betracht lässt. Für den SCC-Bereich (*strong connected core*) kann ein zufälliger Graph aus Abschnitt 6.3 verwendet werden. Der IN-Bereich besteht aus Knoten, welche über Mappings mit dem SCC-Teil verbunden sind. Im OUT-Bereich befinden sich Knoten, die eingehende Mappings aus dem SCC-Bereich haben und vereinzelt direkt mit Knoten aus dem IN-Bereich verbunden sind. Ranken werden analog zu IN- und OUT-Bereichen erzeugt.

Stellt man eine Anfrage an einen Peer im IN-Bereich, so wird sich die Anfragebearbeitung schnell in Richtung SCC-Bereich verlagern, dort vermutlich eine Weile andauern, um dann in den OUT-Bereich vorzustoßen. Für jeden dieser Bereiche wären höchstwahrscheinlich andere Pruningstrategien (siehe [14]) ideal. So kann es für das Verlassen des IN-Bereichs spezielle Bedingungen geben. Wird dieser Bereich verlassen, existiert aufgrund der Bow-Tie-Struktur kein Anfragepfad zurück. Daher ist es wichtig, den IN-Bereich nicht vorzeitig zu verlassen, um keine wertvollen Daten aus diesem Bereich zu verlieren. Analog verhält es sich mit dem Übergang zwischen SCC- und OUT-Bereich. Voraussetzung für derartige Pruningstrategien ist, dass jeder Peer weiß, in welchem Bereich des Graphen er selbst und die verbundenen Peers sich befinden. Alternativ könnte man den lokalen Anfrageplanern der Peers unterschiedliche Pruningstrategien zuteilen.

[4] gemeint ist das Kleidungsstück, engl. *bow tie*

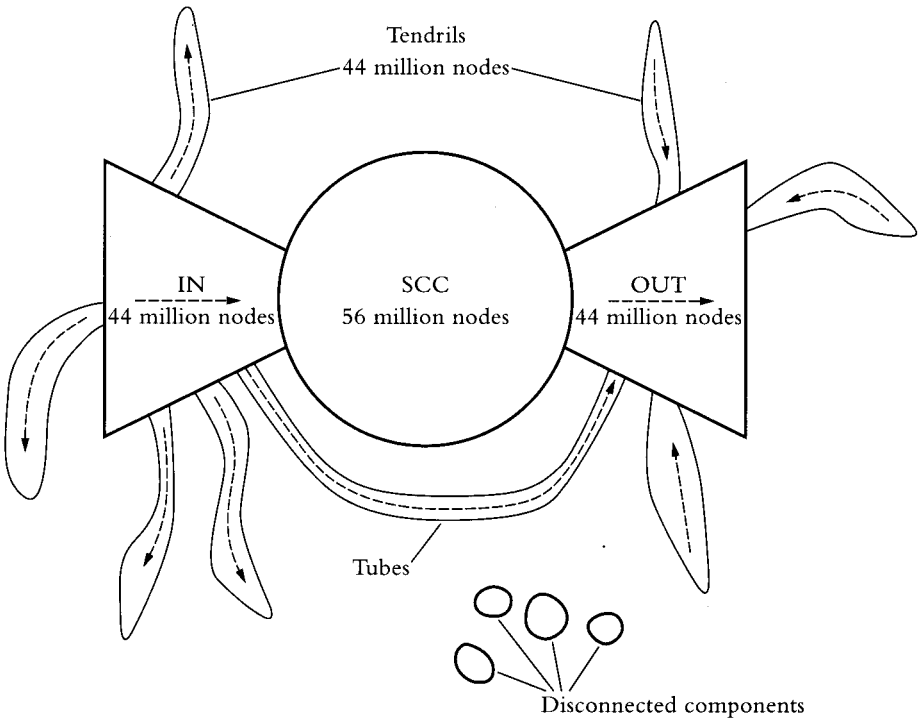


Abbildung 6.9: Bow-Tie-Struktur des Web (Broder et al. 2000)

7 Zusammensetzen des PDMS

In diesem Kapitel wird beschrieben, wie die Einzelergebnisse aus den Kapiteln 4 (Teilschemata), 5 (Schema-Mappings) und 6 (Peer-Graph) zusammengesetzt werden, um ein konkretes PDMS zu erzeugen. Der Vorgang ist in Abbildung 7.1 illustriert. Aus dem Graphen wird zunächst ein einfaches PDMS erzeugt, dessen Schemata noch durch Projektionen modifiziert und um Daten bereichert werden. Anschließend werden die Peer-Mappings um Selektionsprädikate erweitert und das PDMS ist bereit zur Übertragung auf konkrete Peers.

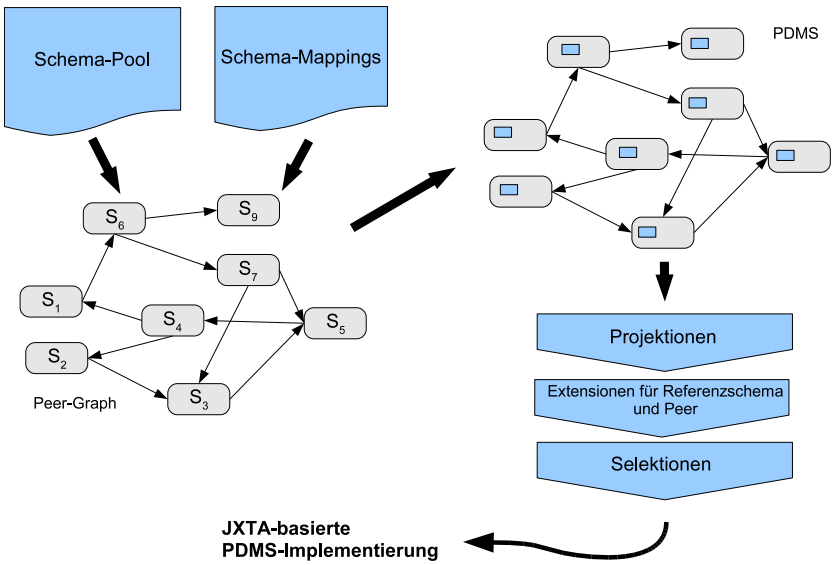


Abbildung 7.1: Zusammenfügen der bisherigen Ergebnisse zu einem PDMS

7.1 Vom Peer-Graphen zum PDMS

In Kapitel 6 wurde ein gerichteter zusammenhängender Graph $G = (V, E)$ konstruiert, der die Topologie des zu erzeugenden PDMS repräsentiert. Des Weiteren wurden Abbildungen $h : V \to \mathcal{S}$ und $f : E \to \mathfrak{P}(\mathcal{M})$ für die Zuweisung der Peer-Schema und Peer-Mappings definiert (siehe Abschnitt 6.1). Für jeden Knoten $v \in V$ wird ein Peer P_v erzeugt, welcher $h(v)$ als Peer-Schema aus dem Schema-Pool $\mathcal{S}$ erhält. Zusätzlich wird P_v ein eindeutiger Peername zugewiesen. Die Peer-Mappings werden aus den Kanten E des Graphen bezogen. Für jede Kante $e \in E$ werden die Mappings $f(e) \subseteq \mathcal{M}$ umgewandelt in Peer-Mappings. Dazu muss in den Datalog-Regeln die Schema-Bezeichnung S_i durch den Peernamen ersetzt werden. Eine mögliche Umschreibung der Datalog-Regel aus Beispiel 5.7 wäre z.B.:

$$\mathsf{Peer1.R_4}(\mathsf{y_1}, \mathsf{y_2}, \mathsf{y_3}, \mathsf{y_4}, \mathsf{y_5}), \mathsf{Peer1.R_5}(\mathsf{y_5}, \mathsf{y_6}, \mathsf{y_7}) :- \mathsf{Peer2.R_6}(\mathsf{y_3}, \mathsf{y_4}, \mathsf{y_5}, \mathsf{y_6}, \mathsf{y_7}).$$

Die umgeschriebenen Mappings enthalten bisher weder Selektionen noch Projektionen.

7.2 Projektionen

Mittels Projektionen lässt sich die Heterogenität der Schemata des PDMS steigern und unvollständige Mappings modellieren. Im ersten Fall werden Peer-Schemata nachträglich modifiziert und im zweiten Fall lediglich die Mappings. Ein Peer-Mapping beschreibt eine Projektion, wenn die Mengen der Variablen auf der Kopf- und der Rumpfseite nicht identisch sind. Es gibt verschiedene Varianten für Projektionen in einem Peer-Mapping $Q_1(P_1) \subseteq Q_2(P_2)$:

(a) Die Variablen aus $Q_2(P_2)$ sind echt enthalten in $Q_1(P_1)$. Beispiel:

$$\mathsf{Peer1.R_1(a, b, c, d)} :- \mathsf{Peer2.R_2(a, b, c)}$$

Dies ist der einfachste Fall einer Projektion. Die von Peer1 an Peer2 zurückgelieferten Daten enthalten zu viele Attribute. Peer2 entfernt die Spalten aus dem Ergebnis, die nicht in die eigene Datenstruktur passen.

(b) Die Variablen aus $Q_1(P_1)$ sind echt in der Menge der Variablen aus $Q_2(P_2)$ enthalten. Beispiel:

$$\mathsf{Peer1.R_1(a, b, c)} :- \mathsf{Peer2.R_2(a, b, c, d)}$$

Peer2 erhält von Peer1 Tupel der Form (x, y, z) als Anfrageergebnis. Das Peer-Schema von Peer2 verlangt aber ein viertes Attribut d für die Relation R_2. Die Lösung ist, dass Peer2 alle Tupel, die Peer1 liefert, um einen NULL-Wert ergänzt. Aus den Tupeln (x, y, z) werden Tupel (x, y, z, NULL). Derartige Projektionen führen dazu, dass kein Containment-Mapping von der Rumpf- zur Kopfseite der Regel möglich ist (CM1) und die Definition des schwachen Containment-Mappings nötig ist (siehe Abschnitt 5).

(c) Die Variablenmengen von $Q_1(P_1)$ und $Q_2(P_2)$ sind nicht disjunkt, enthalten einander aber auch nicht. Beispiel:

$$\mathsf{Peer1.R_1(a, b, c, d, e)} :- \mathsf{Peer2.R_2(a, b, c, f, g)}$$

Dieser Fall stellt die Kombination der beiden oben genannten Fälle dar. Es müssen die Spalten für d und e aus dem Anfrageergebnis von Peer1 entfernt und nicht vorhandene Werte für die Spalten f und g durch NULL-Werte ersetzt werden. Erst dann können die Tupel von Peer2 mit eventuell anderen Anfrageergebnissen (zum Beispiel von lokalen Quellen) vereinigt werden.

Im Allgemeinen unterscheiden sich die Peer-Schemata, weshalb die erwähnten Projektionsvarianten bereits in einigen Mappings vorkommen können. Damit das PDMS noch heterogener wird, wurden zwei Ansätze entwickelt, die im Folgenden näher vorgestellt werden.

7.2.1 Peer-Schemata verändern

Der erste Ansatz erzeugt Projektionen durch Veränderungen an den Peer-Schemata. Aus den Schemata werden zufällig gewählte Attribute entfernt, wodurch einige Änderungen an den Mappings erforderlich sind. Werden aus dem Peer-Schema von P_1 Attribute entfernt, so müssen

- die entsprechenden Variablen aus allen Peer-Mappings in denen P_1 vorkommt ebenfalls entfernt werden und
- da das Schema von Peer P_1 geändert wurde, müssen die Sichtdefinitionen (definierende Mappings) der Relationen angepasst werden. Dies erfolgt ebenfalls durch Entfernen von Variablen.

Als Parameter sind folgende Größen gegeben:

- *pSchema:* Wahrscheinlichkeit, dass in einem Schema Attribute entfernt werden
- *pRelation:* Wahrscheinlichkeit, dass aus einer Relation Attribute entfernt werden, wenn ein Schema für Projektionen ausgewählt wurde (siehe *pSchema*).
- *minAttr:* Minimale Anzahl zu entfernender Attribute pro Relation, falls Relationen für Projektion ausgewählt (siehe *pRelation* und *pSchema*).
- *maxAttr:* Maximale Anzahl zu entfernender Attribute pro Relation (analog *minAttr*)

Zunächst wird eine Menge Peer-Schemata zufällig mit der Wahrscheinlichkeit *pSchema* ausgewählt. Für jedes ausgewählte Peer-Schema wird wie folgt verfahren: Es werden alle Relationen aus dem Schema ermittelt, welche mindestens *minAttr* Attribute besitzen. Aus diesen werden mit der Wahrscheinlichkeit *pRelation* Relationen ausgewählt. Für jede dieser Relationen werden zufällig zwischen *minAttr* und *maxAttr* Attribute entfernt. Es werden nur Attribute entfernt, die nicht Teil einer Fremdschlüsselbeziehung sind. Wurde ein Schema abgearbeitet, werden die Variablen der Peer-Mappings und der definierenden Mappings, welche auf die entfernten Attribute Bezug nehmen, entfernt.

Durch diese Veränderungen der Peer-Schemata wird die schematische Heterogenität des PDMS erhöht, da es nun unwahrscheinlicher ist, dass zwei Peers exakt das gleiche Schema haben: Bei der Erzeugung des Peer-Graphen $G = (V, E)$ wurde die Abbildung $h : V \to \mathcal{S}$ definiert, welche nicht eineindeutig ist. Ein Schema $S \in \mathcal{S}$ kann somit von mehreren Knoten aus V verwendet werden. Da nicht das Schema aus dem Pool verändert wird, sondern das Peer-Schema[1], passiert es nun, dass Peers, welche vorher das gleiche Schema hatten, nach diesem Durchlauf geringfügig unterschiedliche Schemata besitzen.

Bei diesem Ansatz ist es nicht möglich, als Parameter die prozentuale Häufigkeit für Mappings mit Projektionen anzugeben, da Peer-Schemata modifiziert werden und dies Auswirkungen auf mehrere Peer-Mappings haben kann. Aus diesem Grund beziehen sich

[1] Das Peer-Schema ist eine Kopie des Schemas aus dem Schema-Pool und keine Referenz auf dieses.

die Parameter auf die Schemata und Relationen. Dieser Algorithmus kann alle drei Varianten der Projektion in Mappings erzeugen, da sich durch die Entfernung der Attribute die Variablenmengen von Kopf und Rumpf der Mappings zufällig verkleinern.

7.2.2 Peer-Mappings verändern

Der zweite Ansatz kommt ohne Änderung der Peer-Schemata aus und hat nur Auswirkungen auf die Peer-Mappings. Während im ersten Ansatz die resultierenden Mappings alle übertragbaren Informationen von einem Peer zum anderen liefern, wird in diesem Ansatz zusätzlicher Informationsverlust durch Variablenumbennung verursacht. Betrachten wir beispielsweise das Peer-Mapping

$$\mathsf{Peer1.R_1(a, b, c)} :- \mathsf{Peer2.R_2(a, b, c)}.$$

Sämtliche Attribute von $\mathsf{Peer1.R_1(a, b, c)}$ werden zu $\mathsf{Peer2.R_2(a, b, c)}$ übertragen. Wird das Peer-Mapping nun geändert durch Umbenennung einer Mapping-Variable,

$$\mathsf{Peer1.R_1(a, x, c)} :- \mathsf{Peer2.R_2(a, b, c)}$$

so gelangen keine Werte für $\mathsf{Peer1.R_1.b}$ zum Peer2, sondern ausschließlich NULL-Werte, obwohl Peer1 korrekte Daten liefern könnte. Ein mögliches Anwendungsszenario dieses Falls ist, wenn man einen Peer in ein PDMS einbinden möchte, die Korrektheit bestimmter Daten aber nicht sichergestellt ist. Hier sind NULL-Werte besser als falsche Daten, da diese Tupel eventuell subsummiert werden können, anstatt Datenkonflikte auf den falschen Attributen zu verursachen.

7.3 Daten und Datenverteilung

Nachdem Hinzufügen von Projektionen, erfolgen keine Änderungen mehr an den Peer-Schemata. In diesem Abschnitt wird vorgestellt, wie, ausgehend von einer Referenz-Extension, die Extensionen der Peers berechnet werden.

7.3.1 Extension für Referenzschema

Die Testumgebung soll in der Lage sein, bestehende relationale Datenbanken automatisiert über ein PDMS zu verteilen. Für den Fall, dass keine Datenbank verfügbar ist, sollen auch Daten zufällig für ein vorgegebenes Schema generiert werden können. Es existieren somit zwei mögliche Datenquellen für das PDMS:

- eine konkrete Datenbankinstanz, aus der Referenzschema und Extension gelesen werden,
- eine virtuelle Datenbankinstanz, welche ein vorgegebenes Schema besitzt (das Referenzschema), und dessen Extension zufällig berechnet wird.

Die Parameter für die Berechnung der virtuellen Extension sind hier aufgeführt:

- *refSchema:* Datenbankschema für welches eine Extension erzeugt werden soll. Neben der Angabe der Relationsnamen, Attribute und deren Typen sind auch Primärschlüssel und Fremdschlüsselbeziehungen Bestandteil des Schemas.
- *minTupel:* Minimale Tupelanzahl pro Relation
- *maxTupel:* Maximale Tupelanzahl pro Relation

Für jede Relation aus dem Referenzschema *refSchema* wird eine leere Extension angelegt und eine Tupelanzahl *tupelCount* zwischen *minTupel* und *maxTupel* zufällig bestimmt. Dann werden *tupelCount* Tupel für die Relation generiert. Dabei wird die Eindeutigkeit des Primärschlüssels garantiert, um keine Probleme bei der Übertragung in ein RDBMS[2] zu bekommen. Aus dem gleichen Grund werden zunächst keine Werte für Fremdschlüsselattribute generiert, da der jeweilige Wertebereich erst nach Generierung der Daten für den referenzierten Primärschlüssel feststeht. Hat ein Attribut $R_1.c$ eine Fremdschlüsseldefinition zu einem Attribut $R_2.c$, so muss sichergestellt werden, dass für jeden Wert von $R_1.c$ ein Tupel in R_2 existiert mit $R_2.c = R_1.c$. Das gelingt erst, nachdem die Extension R_2 berechnet wurde. Dadurch sind Fremdschlüsselbeziehungen über zwei Primärschlüssel nicht zulässig, wenn die Extension einer virtuellen Datenbankinstanz berechnet werden soll. Wäre für das Primärschlüssel-Attribut $R_2.c$ ebenfalls ein Fremdschlüssel definiert, wie in Abbildung 7.2 dargestellt, dann würden für $R_2.c$ nach der ersten Iteration noch keine Werte zur Verfügung stehen, da diese von $R_3.c$ abhängen.

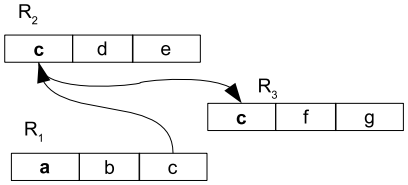


Abbildung 7.2: Nicht erlaubte Fremdschlüsselketten über Primärschlüssel hinweg

Um die Implementierung zu vereinfachen, sind Fremdschlüsselbeziehungen zwischen zwei Primärschlüsseln nicht zulässig: Es darf, ausgehend von einem Primärschlüssel-Attribut, kein Fremdschlüssel definiert werden, wenn Extensionen generiert werden müssen. Fremdschlüsselbeziehungen, welche ausgehend von einem Nicht-Primärschlüsselattribut zu einem Primärschlüsselattribut definiert werden, sind erlaubt. Durch derartig definierte Fremdschlüsselbeziehungen dürfen auch Zyklen im Schema vorhanden sein.

Nachdem für alle Relationen eine Extension erzeugt wurde, müssen in einem zweiten Schritt die fehlenden Fremdschlüsselwerte generiert werden. Wenn eine Fremdschlüsselbeziehung von $R_i.a_j$ zu $R_k.a_l$ definiert ist, dann werden in allen berechneten Tupeln aus der Extension von R_i, zufällige Einträge aus der Wertemenge von $R_k.a_j$ für das Attribut $R_i.a_j$ eingetragen. Nach dieser zweiten Phase ist die Extension für das Referenzschema

[2] relationales Datenbank Management System

vollständig und erfüllt sämtliche Einschränkungen, die durch die Primärschlüssel und Fremdschlüsselbeziehungen definiert wurden.

7.3.2 Peer-Extensionen und Mediator-Peers

Jeder Peer soll einen kleinen Teil des gesamten Wissens aus dem PDMS haben, um seinen Teil zur Anfragebeantwortung beitragen zu können. Eine Möglichkeit dies zu bewerkstelligen ist, anhand von Selektionsanfragen, den Peers Untermengen der Referenzschema-Extension zuzuweisen. Der Vorteil hierbei ist, dass Datenkonflikte aufgrund verschiedener Quellen vermieden werden und echte Duplikate entfernt werden können. Im Folgenden wird beschrieben, wie für jeden Peer verfahren wird. Es muss zwischen zwei Peer-Typen unterschieden werden.

Mediator-Peers

Unter Umständen ist es wünschenswert einigen Peers keine lokalen Quellen mit Daten zuzuweisen, sondern diese ausschließlich als Mediator-Peers fungieren zu lassen. Ein solcher Peer kann Anfragen nur durch Weiterleitung an andere Peers beantworten. Ein Peer darf nur dann als Mediator fungieren, wenn er sein Peer-Schema vollständig mit Daten aus anderen Peers füllen kann. Das bedeutet, dass für alle Relationen des Peer-Schemas ein ausgehendes Mapping zu anderen Peers existieren muss. Um Verweisungsringe zu vermeiden, welche ausschließlich aus Mediator-Peers bestehen, sind Mappings zwischen Mediator-Peers nicht zulässig. Dabei handelt es sich um eine starke Vereinfachung. Zukünftig sollten beispielsweise Hierarchien von Mediator-Peers gestattet sein.

Peers mit lokaler Quelle

Für jede Relation des Peers existieren Sichtdefinitionen, die beschreiben, wie Daten aus dem Referenzschema in das Peer-Schema zu überführen sind. Damit nicht alle Daten des Referenzschemas übernommen werden, müssen die Sichtdefinitionen um Selektionsprädikate erweitert werden. Dies sind einschränkende Vergleichsprädikate in den Datalog-Regeln, welche bestimmte Tupel herausfiltern. Ausgehend davon, dass Selektionen über alle Attribute möglich sind, müssen nur noch die Selektionsintervalle bestimmt werden. Dafür lässt sich die zuvor berechnete Extension des Referenzschemas verwenden.

Es wird für jede Peer-Schema-Relation zufällig und gleichverteilt ein Attribut ausgewählt und der vollständige Wertebereich anhand der Referenzschema-Extension berechnet. Zur Bestimmung eines Selektionsintervalls gibt es folgende Möglichkeiten:

- Es wird genau ein Wert `w` aus der Extension für das Attribut `a` zufällig ausgewählt. Genauso zufällig erfolgt die Wahl des Vergleichsoperators zwischen „$\leq$“ und „$\geq$“. Das resultierende Vergleichsprädikat hat somit die Form `a <= w` bzw. `a >= w`.
- Es werden zwei verschiedene Werte `w` und `v` ausgewählt. Ist $w < v$, so werden die Prädikate `a >= w`, `a <= v`, ist $w > v$ die Prädikate `a >=v`, `a<= w` erzeugt.

Die errechneten Selektionsprädikate dürfen den Sichtdefinitionen nicht bedenkenlos hinzugefügt werden. Soll eine Datenbankinstanz nach dem Löschen von Tupeln aus einer Relation noch die referenzielle Integrität bewahren, so sind aus allen Relationen Einträge mit ungültiger Fremdschlüsselreferenz zu entfernen. Da hierbei abermals Tupel aus Relationen entfernt werden, kann dieser Vorgang neue Inkonsistenzen hervorrufen und ist solange zu wiederholen, bis keine Änderungen mehr an den Extensionen durchgeführt werden müssen. Stellt man Konsistenz auf diese Weise sicher, können sehr kleine und auch leere Relationen auftreten. Dieses Problem könnte zukünftig gelöst werden, indem Peer-Relationen mit Tupeln schrittweise aufgefüllt werden.

Beispiel 7.1 *In Abbildung 7.3 ist ein Peer-Schema dargestellt, in dem zyklische Fremdschlüsselbeziehungen existieren. Dort sind niemals zwei Primärschlüssel an einer Fremdschlüsselbeziehung beteiligt, weshalb dieses Schema zulässig ist. Eine Entfernung des Tupels* $(4, 3)$ *aus der Relation* R_1 *löst folgende Kettenreaktion aus: Aus* R_3 *müssen* $(5, 4)$ *und* $(4, 4)$ *gelöscht werden, aus* R_2 *das Tupel* $(5, 4)$*. Dadurch wird aus* R_1 *das Tupel* $(2, 5)$ *entfernt, und in* R_3 *somit* $(9, 2)$*, was wiederum eine Löschung von* $(3, 9)$ *aus* R_2 *bewirkt. Da* $(4, 3)$ *bereits nicht mehr in* R_1 *existiert, bricht der Prozess an dieser Stelle ab.*

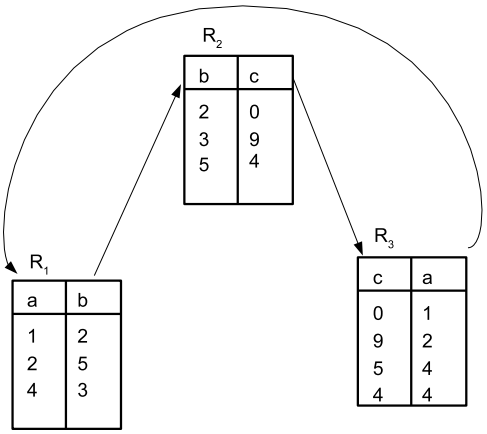


Abbildung 7.3: Kreise durch Fremdschlüsselbeziehungen

Das Beispiel 7.1 verdeutlicht, wie problematisch das Entfernen von Tupeln aus einer konsistenten Datenbankinstanz sein kann. In den Abschnitten 4.5 und 4.4 wurden zusätzliche Attribute in die Teilschemata eingefügt. Die Daten hiefür sind nicht im Referenzschema vorhanden, so dass die hinzugefügten Schlüsselattribute für jeden Peer separat erzeugt werden. Da die künstlichen Primärschlüssel keinem Attribut aus dem Referenzschema zugeordnet wurden, werden diese in Peer-Mappings niemals aufeinander abgebildet und können somit keine Datenkonflikte verursachen.

7.4 Selektionen und Selektivität

In einem PDMS kann es vorkommen, dass bestimmte Tupel über ein Mapping aufgrund eines Filters nicht übertragen werden können. Derartige Filter sind meistens Selektions-

prädikate, die den Wertebereich eines oder mehrerer Attribute einschränken. Im Folgenden wird aufgezeigt, wie sich Selektionen in Mappings erzeugen lassen und welche Probleme damit verbunden sind.

Das Anwenden einer Selektion σ auf die Tupelmenge einer Relation R reduziert die Tupelmenge mit einem Selektivitätsfaktor s, welcher in Prozent angegeben wird. Die Selektivität kann statistisch durch Testanfragen, so genanntes *Sampling* ermittelt werden und gibt an, wie viel Prozent der Tupel durch die Selektion erfasst werden.

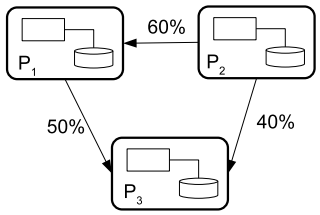


Abbildung 7.4: Mappings mit Selektivitätsfaktor

Auch in Peer-Mappings sind Selektionen sinnvoll. So können verschiedene Peers dieselbe Art von Daten modellieren, aber jeweils unterschiedliche Attributwerte aufweisen. In [13] beziehen A. Roth und F. Naumann die Selektivität eines Mappings auf das Mappingziel (den Kopfteil) und auf alle durch das Mapping erreichbaren Quellen. In der Abbildung 7.4 ist ein Mapping zwischen den Peers P_2 und P_1 mit einer Selektivität von 60% dargestellt. Fragt P_2 über dieses Mapping P_1, so werden nur 60% aller von P_1 erreichbaren Daten zurückgeliefert.

Für Experimente ist es wichtig, einer bestimmten Anzahl von Mappings Vergleichsprädikate hinzufügen zu können, so dass diese eine geforderte Selektivität aufweisen. Die Selektivität der Mappings wird von Pruning-Algorithmen während der Anfragebearbeitung genutzt, um Antwortzeiten zu verkürzen. Die Aufgabe ist es, für einen gegebenen Selektivitätsfaktor eine Menge von Vergleichsprädikaten für ein Mapping zu berechnen. Ein möglicher Ansatz ist, ein einzelnes Selektionsprädikat zufällig zu wählen und durch Testanfragen solange zu justieren, bis die gewünschte Selektivität erreicht wird. Dazu stellt man zuerst eine Testanfrage ohne Prädikat und vergleicht die Anzahl zurückgelieferter Tupel mit denen der anderen Testanfragen, welche Selektionsprädikate aufweisen. Zur Justierung des Vergleichsprädikats sind Ansätze wie zum Beispiel Intervallschachtelung denkbar.

Das Berechnen der Selektivitäten gemäß obiger Definition ist mit einigen Problemen verbunden:

- **Testanfragen:** Die gemessenen Selektivitäten hängen stark von der Werteverteilung innerhalb der Extension ab.

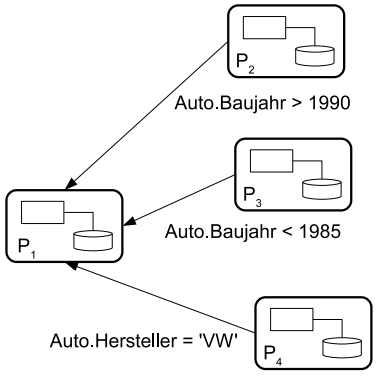


Abbildung 7.5: Widersprüchliche Mappings

- **Abhängigkeiten der Attribute:** Sind die Werte der Attribute nicht unabhängig voneinander, kann die Selektivität mehrerer Prädikate nicht mehr einheitlich berechnet werden. In der Regel werden Selektivitäten unabhängiger Attribute multipliziert.

- **Konsistenz:** Sollen die Vergleichsprädikate eingehender Peer-Mappings eine Aussage über die im Peer gespeicherten Daten machen, so sollten sich diese nicht widersprechen. Das in Abbildung 7.5 dargestellte PDMS ist ein Beispiel für ein inkonsistentes PDMS. Die widersprüchlichen Mappings ließen sich auch als vetragliche Abkommen oder Vertrauensverhältnisse auffassen. So kann es sein, dass P_2 von Peer P_1 aus vertraglichen Gründen nur Fahrzeuge mit einem Baujahr nach 1990 erfragen darf oder nur diese Daten als verlässlich ansieht. Es könnte sich zum Beispiel durch empirische Untersuchen gezeigt haben, dass die Daten älterer Fahrzeuge in P_1 eine hohe Fehlerquote aufweisen.

- **Gegenseitige Beeinflussung:** Die tatsächliche Selektivität eines Mappings hängt nicht nur von den Daten der Peers, welche es verbindet, sondern im Allgemeinen vom gesamten PDMS ab. Wird in einem Peer-Mapping ein Vergleichsprädikat geändert, hat dies Einfluss auf alle anderen Mappings deren tatsächliche Selektivität sich dadurch ändern kann.

Vereinfachung und Umsetzung

Das erste Problem kann umgangen werden, indem die Gleichverteilung der Werte in den Extensionen vorausgesetzt wird. Für die zufällig erzeugten Extensionen ist dies bereits der Fall. Das Problem der Abhängigkeiten zwischen den Attributen lösen wir, indem wir uns auf jeweils ein Vergleichsprädikat pro Peer-Mapping beschränken. Die Konsistenz der eingehenden Mappings eines Peers wird nicht beachtet. Somit ist es möglich, eine prozentuale Verteilung der Selektivitäten vorzugeben und jedem Mapping seine eigene Selektivität zuzuweisen. Andernfalls müssten alle eingehenden Mappings eines Peers sich nicht widersprechende Prädikate haben, was die Anpassung einzelner Mappings und

eine genaue Verteilung der Selektivitäten unmöglich macht. Der Einfluss eines Vergleichsprädikates auf andere Mappings verschwindet, wenn man Selektivität nicht anhand aller erreichbarer Quellen, sondern nur bezogen auf die lokalen Quellen definiert. Die Selektivität eines Mappings beschreibt dann nur noch, wie viel Prozent der Tupel der lokalen Quelle des Zielpeers durch das Mapping übertragen werden.

Der Algorithmus setzt ein Peer-Netz, bestehend aus Peers und Mappings ohne Selektionsprädikate, voraus. Ein Mapping darf nur dann Selektionsprädikate aufweisen, wenn der Peer aus dem Mapping-Kopf kein Mediator-Peer ist, also eine lokale Quelle bereitstellt. Dies ist nötig, da aufgrund obiger Einschränkung die Selektivität nur für lokale Quellen definiert wird.

Als Parameter ist eine Liste geordneter Paare (m_p, s_p) gegeben. Der Wert m_P gibt die Häufigkeit der Mappings und s_p die Selektivität an. Das Tupel $(50, 40)$ bedeutet, dass 50% aller Mappings eine Selektivität von 40% aufweisen sollen. Jedem Peer-Mapping, welches keinen Mediator-Peer als Ziel-Peer besitzt, wird eine Selektivität zugewiesen und ein Vergleichsprädikat errechnet, welches s_p Prozent der Tupel aus der lokalen Quelle abdeckt. Durch die Gleichverteilung kann der Wert aus der Extension gewonnen werden.

7.5 Zusammenfassung

Im ersten Abschnitt wird erklärt, wie aus einem Schema-Pool $\mathcal{S}$, einer Menge mit Mappings $\mathcal{M}$ und einem Peer-Graphen ein virtuelles PDMS erzeugt wird. Ein virtuelles PDMS existiert zunächst nur in Form von Datenstrukturen im Speicher der Testumgebung. Bei dessen Erzeugung wird jedem Peer ein Schema aus dem Schema-Pool als Peer-Schema zugewiesen. Eine Kante im Peer-Graphen entspricht einer Teilmenge der Mappings aus $\mathcal{M}$. Diese Mappings, welche sich ursprünglich auf die Schemata des Schema-Pools bezogen, werden zu Peer-Mappings umgeschrieben (siehe 7.1). Im Anschluss werden einige Peer-Schemata und Peer-Mappings nachträglich durch Projektionen modifiziert, um die Heterogenität der Peer-Schemata zu erhöhen und um unvollständige Mappings zu erzeugen. Nach dem Erzeugen der Projektionen stehen die Peer-Schemata fest und es können Extensionen für das Referenzschema und daraus die Daten für die lokalen Quellen der Peers generiert werden. Anhand der Extensionen werden einige der Mappings um Selektionsprädikate ergänzt, um die Auswirkung von Pruning-Algorithmen testen zu können, welche Selektivitäten beachten.

Im nächsten Kapitel über die Implementierung wird beschrieben, wie aus einem virtuellen PDMS über die Programm-Schnittstellen aus [14] ein konkretes, wirklich anfragbares PDMS erzeugt wird.

8 Implementierung

Die Testumgebung wurde als SWING[1]-Anwendung in Java implementiert. Sie ist in der Lage, ein virtuelles PDMS zu erzeugen und auf real laufende JXTA[2]-Peers zu übertragen. Der Nutzer kann erzeugte Peer-Netze speichern und laden. Es können Anfragen gestellt, die Anfragebearbeitung visualisiert sowie die Ergebnisse tabellarisch angezeigt werden.

In diesem Kapitel wird die Architektur der Anwendung und die Verknüpfung mit der in [14] vorgestellten PDMS-Implementierung gezeigt. Es wird beschrieben, wie ein virtuelles PDMS auf real laufende Peers übertragen wird und wie die Kommunikation zwischen der Testumgebung und den Peers abläuft. Im Anschluss werden die Möglichkeiten zur Erweiterung der Anwendung beschrieben.

8.1 Architektur der Testumgebung

Bei der Implementierung wurde das Model-View-Controller[3] Design-Pattern verwendet und Logik von Daten und graphischer Benutzeroberfläche getrennt. Dieser Ansatz ermöglicht den Austausch des graphischen Frontends und erhöht die Wiederverwendbarkeit der implementierten Applikationslogik. Es existieren

- ein Datenmodell, in dem der Zustand der Anwendung gespeichert wird,
- verschiedene Sichten (engl. *views*) die zur Visualisierung der Daten und zur Interaktion mit dem Anwender eingesetzt werden können
- und ein *Controller*, der auf Benutzerereignisse angemessen reagiert und die Logik der Anwendung verwaltet.

Dieser Ansatz wurde ursprünglich für das Smalltalk 80 Framework entwickelt (siehe [8]) und findet auch unter anderem in den SWING-Klassen von Java selbst Verwendung.

In Abbildung 8.1 befindet sich eine graphische Darstellung der Architektur. Die Testumgebung greift auf Schnittstellen des implementierten PDMS (siehe [14]) zurück, um sich als Monitor-Peer mit den PDMS-Peers verbinden zu können. Die dafür notwendige Komponente ist im oberen Teil dargestellt. Bindeglied zwischen Testumgebung und dem JXTA-Framework ist die Klasse *GUICoreController*. Dort wird abstrahiert von der allgemeinen *CoreController*-Klasse, welche von den PDMS-Peers verwendet wird, und eine Schnittstelle für die Testumgebung zur Verfügung gestellt. Von dieser wird die Anwendung über gesehene Peers und dessen Status-Informationen informiert. In der Modell-Klasse *PeerModel* werden alle Daten zu den gesehenen bzw. erzeugten Peers gespeichert. Sie dient als

[1] http://java.sun.com/products/jfc/index.jsp (Übersicht über SWING-Klassen)
[2] http://www.jxta.org
[3] http://www.enode.com/x/markup/tutorial/mvc.html

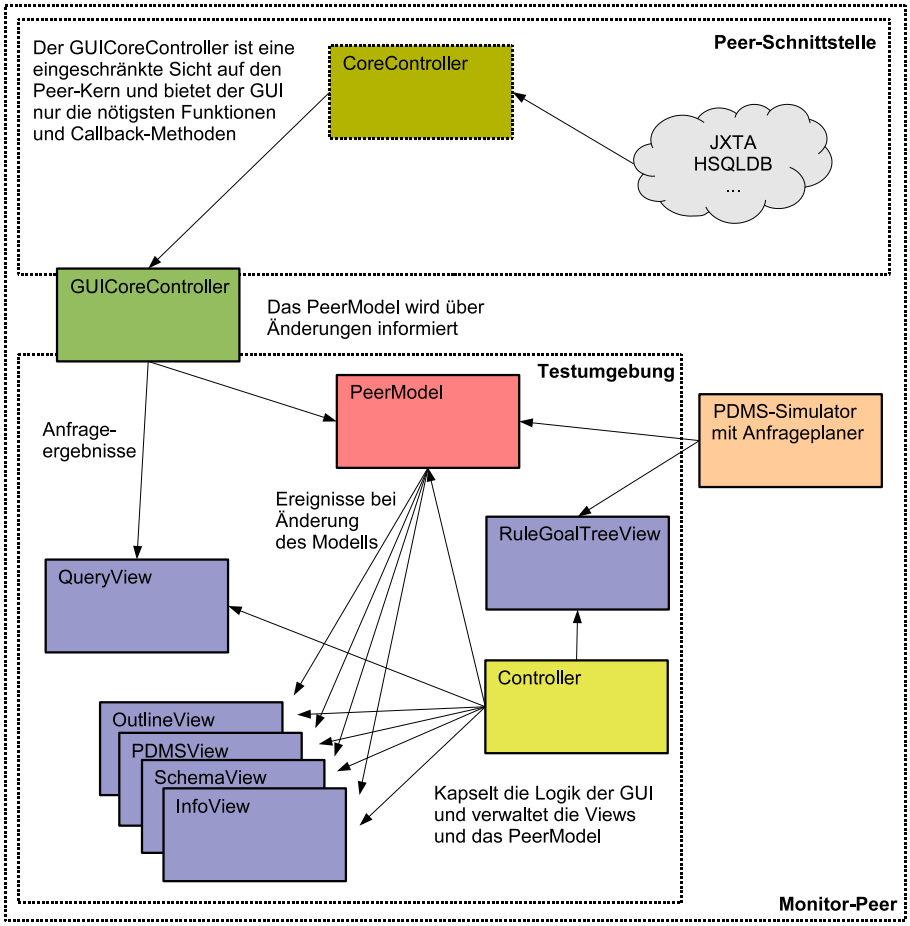


Abbildung 8.1: Architektur der Testumgebung mit Peer-Schnittstelle

GUI- und JXTA-unabhängige Speichermöglichkeit eines PDMS und übernimmt gleichzeitig die Funktion eines Zwischenspeichers. Daten zu einem Peer können somit direkt aus dem *PeerModel* gelesen werden und müssen nicht bei jedem Zugriff seitens der Anwendung über das Kommunikationsprotokoll abgefragt werden. Dies ist vor allem für die Visualisierung der Peer-Netze wichtig, da für eine flüssige Darstellung ein lokales Modell notwendig ist. Auf das *PeerModel* greifen verschiedene Sichten (*views*) zu, um die dort gespeicherten Daten zu visualisieren. So können dieselben Daten aus unterschiedlichen Perspektiven betrachtet werden.

8.1.1 Integration der alten PDMS-Simulator-Implementierung

Bevor das PDMS in [14] und diese Testumgebung implementiert wurden, gab es bereits eine Java-Implementierung, die in der Lage ist, ein PDMS lokal auf einem Rechner zu simulieren. Der vorhandene Quellcode des PDMS-Simulators wurde in die Testumgebung integriert. Die entsprechende Komponente ist in Abbildung 8.1 als *PDMS-Simulator* eingezeichnet. Erzeugt der Anwender ein PDMS, so ist dies zunächst ausschließlich im *Peer-*

Model und im *PDMS-Simulator* repräsentiert. Ein solches PDMS nennen wir „simuliertes PDMS“. Ist der Benutzer mit dem erzeugten PDMS zufrieden, kann er es auf konkrete PDMS-Peers übertragen, die zuvor über die Peer-Schnittstelle ermittelt wurden. Dabei entsteht ein „konkretes PDMS“, welches dann nochmals eine Repräsentation im *PeerModel* erhält.

Über den *PDMS-Simulator* können Anfragen an simulierte Peers gestellt werden. Dabei wird ein kompletter Rule-Goal-Tree erzeugt und im *RuleGoalTreeView* dargestellt. Hier werden keine Daten übertragen, sondern nur Berechnungen durchgeführt. Stellt man eine Anfrage an einen PDMS-Peer, so erhält man neben dem kompletten Anfrageplan und einigen Daten zum Verlauf der Anfragebearbeitung auch das Anfrageergebnis in Form einer Tupelmenge zurück.

8.1.2 Sichten

Die Benutzeroberfläche verfügt über einige Sichten (engl. *views*), die hier kurz erläutert werden sollen. Ziel des Entwurfs war, die im Datenmodell gespeicherten Informationen angemessen zu präsentieren.

PDMSView und OutlineView

Im PDMSView wird das aktuell gewählte[4] Peer-Netz graphisch dargestellt. Für die Visualisierung wurde das Java-Framework Prefuse[5] eingesetzt, welches eine gute Trennung zwischen Datenknoten und visuellen Elementen besitzt und verschiedene Layouts zur Ausrichtung von Graphen mitbringt. In Abbildung 8.2 ist ein mit Prefuse ausgegebener Graph zu sehen.

Die dargestellten Knoten entsprechen den PDMS-Peers, die gerichteten Kanten der Menge von Mappings, welche zwischen den Peers definiert wurden. Die Pfeile zeigen vom Peer aus dem Rumpf des Mappings in Richtung des Peers aus dem Kopf. Es werden nicht nur real betriebene Systeme angezeigt, sondern auch simulierte Peer-Netze. Die Topologie eines gerade erzeugten PDMS kann somit sofort visuell erfasst und kontrolliert werden, bevor die Daten auf die echten Peers übertragen werden. Die Ansicht OutlineView stellt das *PeerModel* baumartig dar und zeigt im Gegensatz zum PDMSView alle gefundenen Peers an und nicht nur die des selektierten PDMS.

SchemaView und InfoView

Klickt der Anwender auf einen Peer, so wird im SchemaView das Peer-Schema dargestellt. Mehr Informationen finden sich im InfoView wieder. Dort werden ausführliche Informationen sowohl über JXTA-Peers als auch simulierte Peers angezeigt. Dazu gehören eindeutige Namen, lokale Mappings und Peer-Mappings.

[4] Existieren im Netzwerk mehrere PDMS, so kann der Anwender eine Auswahl treffen.
[5] http://prefuse.sf.net

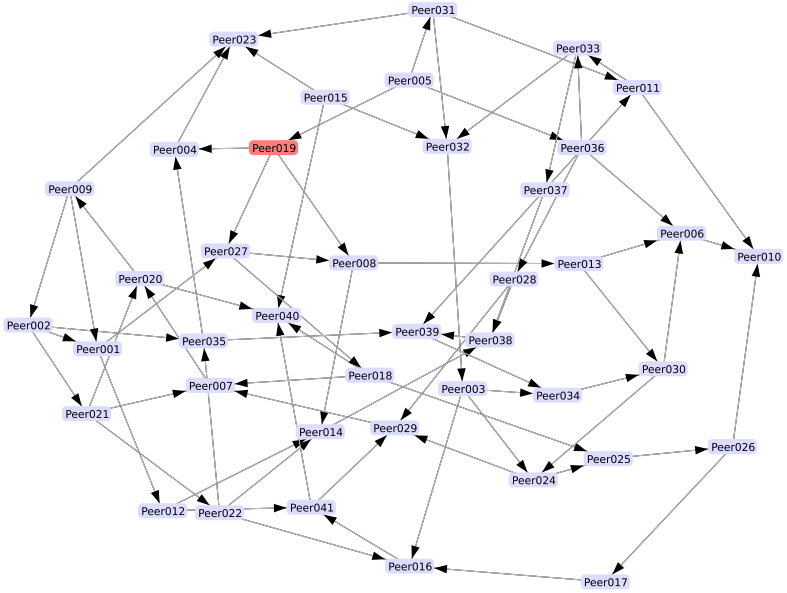


Abbildung 8.2: Visualisierung eines Peer-Netzes mit Prefuse

RuleGoalTreeView

Stellt der Anwender über den Simulator Anfragen an das simulierte PDMS, so kann der Rule-Goal-Tree in diesem View angezeigt werden. Dieser View wurde angelegt, um die Ausgaben des übernommenen PDMS-Simulator-Codes in die Oberfläche einzubetten und um eventuell später den vollständigen Rule-Goal-Tree einer Anfrage an ein konkretes PDMS anzuzeigen. In Abbildung 8.3 ist ein exemplarischer RG-Tree dargestellt.

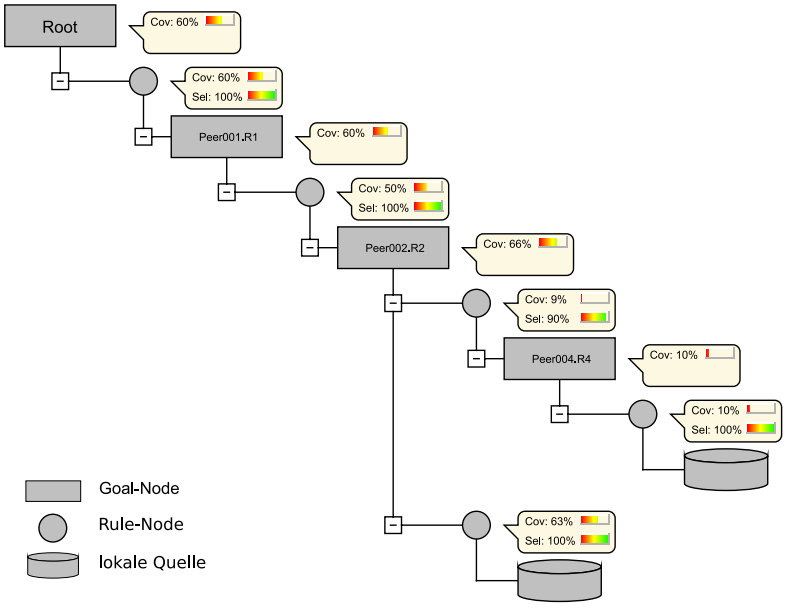


Abbildung 8.3: Visualisierung eines Rule-Goal-Tree

Neben den *Goal*- und *Rule*-Knoten des Baumes sind errechnete Werte für *Coverage* und *Selectivity* dargestellt. Die Werte sind nur exemplarisch und wurden aus dem vom Anfrageplaner errechneten Rule-Goal-Tree bezogen. Hier sind noch Freiräume für weitere Daten. Eine genauere Beschreibung des Rule-Goal-Trees ist in [14] und [13] zu finden.

QueryView

Stellt der Anwender Anfragen an konkrete Peers, werden die Anfrageergebnisse in diesem View angezeigt. Martin Schweigert hat diesen in [14] noch um die Ausgabe des Anfrageplans und um einige Zusatzinformationen erweitert.

8.1.3 Datenmodell

Das *PeerModel* ist das zentrale Datenmodell der Anwendung. In ihm sind alle Peers, Schemata und Mappings für die Verwendung der Testumgebung gespeichert. Es wird über Callback-Funktionen vom *GUICoreController* über Änderungen an den PDMS-Peers informiert, so dass stets der aktuelle Zustand des PDMS repräsentiert wird. Bei Änderungen am Modell werden automatisch alle am Modell registrierten Sichten (siehe Abbildung 8.1) aktualisiert. Umgekehrt kann das Modell indirekt durch die Sichten beeinflusst werden, wenn zum Beispiel der Benutzer einen Peer umbenennt oder seine Peer-Mappings entfernt.

Das *PeerModel* kann über das XStream-Framework[6] serialisiert und deserialisiert werden. Dadurch ist das Speichern und Laden eines simulierten PDMS unabhängig von vorhandenen konkreten Peers möglich. Ein einmalig generiertes PDMS kann somit für verschiedene Experimente wiederverwendet werden.

Im *PeerModel* sind eine Menge von Peers, Peer-Mappings und lokalen Mappings gespeichert, welche jeweils einer Gruppe zugehören. Eine Gruppe ist gleichbedeutend mit einem eigenständigen PDMS. In dem Modell können somit zeitgleich mehrere konkrete PDMS-Instanzen abgelegt sein. Für simulierte PDMS gibt es einen ausgezeichneten Gruppennamen, weshalb hiervon nur eines im *PeerModel* existieren darf. Im Wesentlichen orientieren sich die Modell-Klassen an den in [14] vorgestellten Interfaces bzw. an den PDMS-Simulator-Klassen.

8.2 Übertragung auf PDMS-Peers

Da sich die Testumgebung als JXTA-Peer im Netzwerk ausgibt, als so genannter *Monitor-Peer*, kann sie andere Peers im Netz ausfindig machen und mit diesen kommunizieren. Ein erzeugtes simuliertes PDMS kann auf diese Weise auf JXTA-Peers übertragen werden. Zunächst muss sichergestellt sein, dass genügend PDMS-Peers im Netz existieren, die das simulierte PDMS aufnehmen können. Für den Fall, dass zu wenige Peers vorhanden sind, lassen sich einzelne Peers teilen. Durch den mehrfachen Aufruf der entfernten *splitPeer()*-Methode über die Peer-Schnittstelle können, selbst bei Vorhandensein nur eines einzelnen

[6] http://xstream.codehaus.org

Peers, genügend PDMS-Peers nachträglich erzeugt werden. Eine Beschreibung der Schnittstellen und ausführliche Details zur Implementierung sind in [14] zu finden.

Nachdem sichergestellt wurde, dass genügend PDMS-Peers zur Aufnahme der Daten vorhanden sind, können alle Peers separat mit Schema, Daten und Mappings aufgebaut werden. Jeder PDMS-Peer verfügt über eine einheitliche Schnittstelle, über die sich lokale Schemata, lokale Mappings, Extension, Peer-Schema und Peer-Mappings setzen und auslesen lassen. In Abbildung 8.4 ist das zugehörige Java-Interface dargestellt.

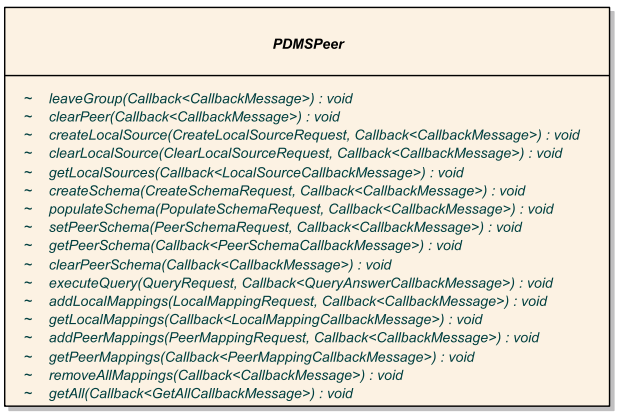


Abbildung 8.4: PDMS-Peer Interface

Die Methode *createLocalSource()* wird aufgerufen, um eine lokale Quelle anzulegen. Mit der Methode *createSchema()* wird das Schema der lokalen Quelle festgelegt und mit *populateSchema()* kann die Extension der lokalen Quelle gesetzt werden. Das Peer-Schema lässt sich über die Methode *setPeerSchema()* setzen. Lokale Mappings sowie Peer-Mappings werden über die Methoden *setLocalMappings()* und *setPeerMappings()* übertragen. Alle Methoden sind asynchron und geben ihren Status nach der Ausführung über eine Callback-Funktion zurück, was prinzipiell eine parallele Ausführung ermöglicht. Um den Prozess der Übertragung besser überwachen zu können, werden die Methoden beim Aufbau der Peers jedoch sequentiell aufgerufen. Das heißt, bevor der nächste Aufruf erfolgt, wird gewartet, bis die Statusmeldung des letzten Aufrufs eingetroffen ist.

In Abbildung 8.5 ist die Reihenfolge der synchronen Aufrufe zu sehen, mit der ein PDMS-Peer aufgebaut wird. Zunächst wird jedem Peer eine lokale Quelle mit Schema und Extension zugewiesen. Konnte die Extension fehlerfrei in die lokale Datenbank des Peers eingefügt werden, meldet der Peer dies über die Callback-Funktion zurück. Im Fehlerfall wird die Fehlerursache ebenfalls über eine Callback-Nachricht mitgeteilt. In diesem Fall bricht der gesamte Übertragungsvorgang ab. Ursache für einen Fehler kann sein, dass sich der Peer innerhalb einer vorgegebenen Zeitspanne nicht über die Callback-Funktion zurückgemeldet hat. Entweder dauert das Übertragen der Extension länger als eine konfigurierbare Zeitspanne (zum Beispiel 3 Minuten) oder der Peer ist nicht mehr im Netzwerk erreichbar. Sollten auf dem Peer unvorhergesehene *Java-Exceptions* auftreten, werden diese mit der

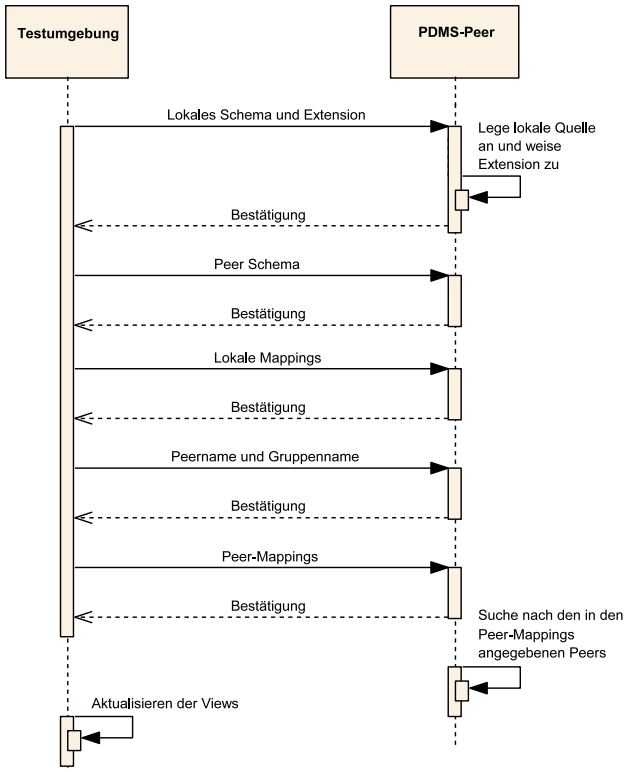


Abbildung 8.5: Übermittlung von Schema, Daten und Mappings an Peers

Callback-Funktion übertragen und in der Benutzeroberfläche der Testumgebung ausgegeben. Dies gilt auch für alle anderen Aufrufe. Ist die lokale Quelle mit Schema erfolgreich gesetzt worden, können Peer-Schema und lokale Mappings übertragen werden. Die beiden nächsten Schritte (Zuweisung der Peernamen und Gruppennamen sowie Peer-Mappings) wurden bewusst an die letzte Stelle gesetzt, da diese ein größeres Kommunikationsaufkommen zur Folge haben. Die via JXTA gefundenen PDMS-Peers haben initial möglicherweise noch nicht den korrekten Gruppen- und Peernamen gesetzt und bekommen im nächsten Aufruf die richtigen Werte zugewiesen. Der letzte Aufruf weist jedem Peer seine Peer-Mappings zu. An jedem Peer-Mapping sind zwei Peers beteiligt, von denen nur der auf der Rumpfseite einen Nutzen vom Mapping hat, weshalb nur diesem das Mapping zugewiesen wird. Das Mapping $Q_1(P_1) \subseteq Q_2(P_2)$ drückt aus, wie Peer P_2 den Peer P_1 anfragen darf, somit muss Peer P_1 keine Kenntnis von diesem Mapping haben. Das Setzen der Peer-Mappings bewirkt einen erhöhten Netzwerkverkehr, da jeder Peer versucht, die in seinen Mappings zitierten anderen Peers im Netzwerk zu finden. Dieser Prozess läuft im Hintergrund ab, nachdem der Peer seine Peer-Mappings erhalten hat.

Das PDMS ist erst einsatzbereit, wenn sich alle Peers gefunden und der Testumgebung den Status *ready* übermittelt haben.

8.3 Erweiterbarkeit

In diesem Unterabschnitt wird beschrieben, wie sich die vorliegende Implementierung erweitern lässt. Im ersten Abschnitt werden die Datenstrukturen zur Speicherung generierter Teilschemata und die bestehenden Algorithmen erläutert. Dann wird erklärt, wie das gemeinsame Interface zu implementieren ist, um die Testumgebung mit einem neuen Algorithmus zur Schema-Erzeugung zu erweitern. Im nachfolgenden Unterabschnitt wird vorgestellt, wie sich die Menge der unterstützten Graphen vergrößern lässt. Dabei werden ebenfalls die verwendeten Schnittstellen genauer beschrieben.

8.3.1 Zusätzliche Teilschemata erzeugen

Abbildung 8.6: Relevante Klassen der Schema-Erzeugung

Zur Repräsentation eines Schemas werden die Klassen *SchemaDTO*, *RelationDTO* und *AttributeDTO* von der Testumgebung und der PDMS-Implementierung *System P* gleichermaßen verwendet. Das passende UML-Diagram ist in [14] zu finden. Für die Schemagenerierung müssen zusätzlich Sichtdefinitionen bezüglich des Referenzschemas (siehe Kapitel 4) gespeichert und Methoden zum Klonen bereitgestellt werden. In Abbildung 8.6 befindet sich eine Übersicht der Klassen zur Speicherung generierter Teilschemata.

Die Klasse *GeneratedSchema* enthält das Quellschema als *SchemaDTO*-Objekt (im Allgemeinen handelt es sich dabei um das Referenzschema) und ein Zielschema. Unter einem Zielschema verstehen wir das erzeugte Teilschema, von dem aus Sichtdefinitionen (definierende Mappings) zum Quellschema erstellt wurden. Sichtdefinitionen werden repräsentiert durch die Klasse *GeneratedMapping* und sind in einer Liste Bestandteil der Klasse *GeneratedSchema*. Ein solches Mapping setzt sich aus *GeneratedSubGoal*- und *GeneratedPredicate*-Klassen zusammen.

Für die Schema-Erzeugung gibt es drei Algorithmen, die das gemeinsame, in Abbildung 8.7 gezeigte Interface *SchemaGenerationAlgorithm* implementieren. Die Klasse *SchemaPool* verwendet diese Algorithmen, um Teilschemata eines gegebenen Referenzschemas zu erzeugen. Soll ein neuer Algorithmus hinzugefügt werden, so ist dieses Interface zu verwenden.

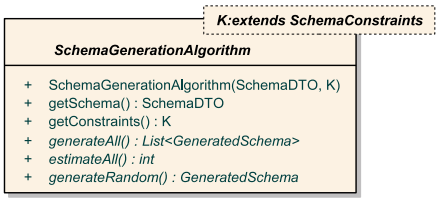


Abbildung 8.7: Interface der Algorithmen zur Schema-Erzeugung

Die abstrakte generische[7] Klasse *SchemaGenerationAlgorithm* erfordert den Parameter *K*, der festlegt, von welchem Typ die zugehörige *SchemaConstraints*-Klasse ist. In dieser werden Einschränkungen der zu erzeugenden Teilschemata beschrieben, die sich zwischen den verschiedenen Algorithmen unterscheiden können. In Abbildung 8.8 ist eine Übersicht der *SchemaConstraints*-basierten Klassen zu sehen. Eine neue, von *SchemaGenerationAlgorithm* abgeleitete Klasse sollte, analog zu den existierenden Algorithmen, eine eigene Constraints-Klasse besitzen, die von *BasicSchemaConstraints* erbt.

Ein neuer Algorithmus muss die in Abbildung 8.7 dargestellten Methoden implementieren. Die Methode *generateAll()* erzeugt alle möglichen Teilschemata zum gegebenen Referenzschema, während die Methode *generateRandom()* nur ein zufälliges Teilschema erzeugt. Die Methode *estimateAll()* gibt einen geschätzten Wert für die Anzahl der erzeugten Schemata von *generateAll()* zurück. Um die Laufzeit der Schema-Erzeugung gering zu halten,

[7] In Java gibt es seit Version 5.0 das Konzept der generischen Klassen.

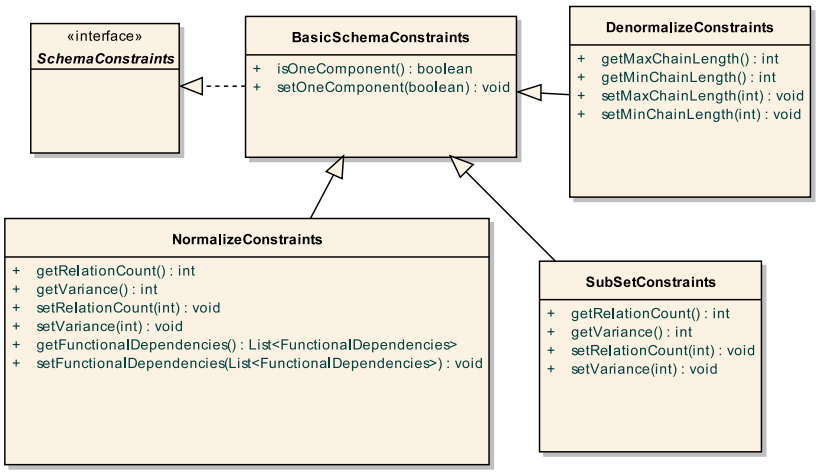


Abbildung 8.8: Interface der Algorithmen zur Schema-Erzeugung

wird vor dem Aufruf der Methode *generateAll()* via *estimateAll()* geprüft, wie viele Schemata wahrscheinlich generiert werden. Übersteigt diese Zahl einen gegebenen Grenzwert, wird statt *generateAll()* die Methode *generateRandom()* mehrfach aufgerufen. Diese Maßnahme war nötig, da die Anzahl generierter Schemata im Allgemeinen exponentiell mit der Anzahl der Relationen des Referenzschemas wächst (siehe hierzu Kapitel 9). Erst auf diese Weise ist es möglich, für Experimente in kurzer Zeit viele virtuelle PDMS zu generieren. Das Feature kann bei Bedarf deaktiviert werden.

Die in Kapitel 4 vorgestellten Algorithmen zur Schema-Generierung beachten keine Einschränkungen bezüglich der zu erzeugenden Schemata. Dies ist nicht immer sinnvoll, da beispielsweise der Denormalisierungs-Algorithmus sehr lange Join-Ketten erzeugen kann, womit einige DBMS Probleme haben. Für den Denormalisierungs-Algorithmus existieren daher Parameter zur Festlegung der minimalen und maximalen Länge von Join-Ketten. Analog besitzen die Constraints-Klassen des Normalisierungs- und Teilmengenalgorithmus Parameter zur Steuerung der Anzahl der betrachteten Relationen.

Stellt ein Algorithmus zur Schema-Erzeugung obige Methoden zur Verfügung, kann dieser in die Klasse *SchemaPool* analog zu den bestehenden Algorithmen eingebunden werden.

8.3.2 Unterstützung weiterer Graphentypen

Die Testumgebung verfügt bereits über eine Auswahl an Algorithmen zur Erzeugung von Peer-Graphen (siehe Kapitel 6). Möchte man Experimente an anderen interessanten Graphen (siehe hierzu Abschnitt 6.4) durchführen, so können eigene Algorithmen analog zu den bisherigen entwickelt und leicht integriert werden. In Abbildung 8.9 ist das Interface eines Graph-Erzeugungs-Algorithmus dargestellt. Ein Algorithmus zur Graph-Erzeugung benötigt eine von *GraphConstraints* erbende Klasse K zur Repräsentierung der Einschränkungen des Graphen (zum Beispiel Minimalgrad und Maximalgrad) und für die Methode

generate() eine Ergebnisklasse T, die von *MappingGraph* erbt. Der Methode *generate()* werden ein Schema-Pool und eine Instanz der Constraints-Klasse K übergeben. Ergebnis ist eine Klasse, die den Graphen repräsentiert und von dem Interface *MappingGraph* abgeleitet ist.

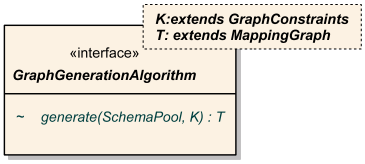


Abbildung 8.9: Interface der Algorithmen zur Graph-Erzeugung

Das Interface *MappingGraph* ist in Abbildung 8.10 dargestellt und besitzt grundlegende Methoden zur Verwaltung eines Graphen.

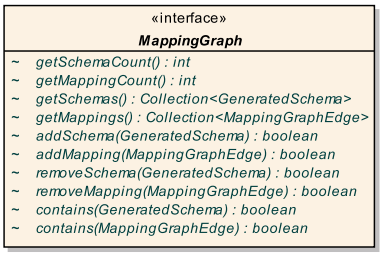


Abbildung 8.10: Interface eines Graphen

Über die Methoden *addSchema()* und *removeSchema()* können Schemata als Knoten hinzugefügt und entfernt werden. Für das Hinzufügen und Entfernen von Mappings sind ebenfalls Methoden bereitgestellt. Das theoretische Konstrukt einer Abbildung $h : V \rightarrow \mathcal{S}$ (siehe Kapitel 6) spiegelt sich hier dadurch wider, dass ein Knoten in den Graphen eingefügt wird, indem die Kopie eines Schemas des Schema-Pools über *addSchema()* eingefügt wird. Somit ist das kopierte Schema zugleich ein Knoten des Graphen. Gleiches gilt für die Mappings, welche in der Klasse *MappingGraphEdge* zu einer Kante zusammengefasst und mit Bezug auf die geklonten Schemata eingefügt werden. Die zur Erklärung der Graph-Erzeugung eingeführte Abbildung $f : E \rightarrow \mathfrak{P}(\mathcal{M})$ ist hier somit ebenfalls repräsentiert, da die Klasse *MappingGraphEdge* zwei Schemata aus dem Graphen referenzieren muss, zwischen diesen eine Kante repräsentiert und ihnen eine Menge von Mappings zuweist.

Eine neue Algorithmus-Klasse muss somit das Interface *GraphGenerationAlgorithm* implementieren und eine eigene Constraints-Klasse, basierend auf *GraphConstraints*, bereitstellen. Die von den vorhandenen Algorithmen verwendeten Constraints-Klassen sind in Abbildung 8.11 aufgeführt.

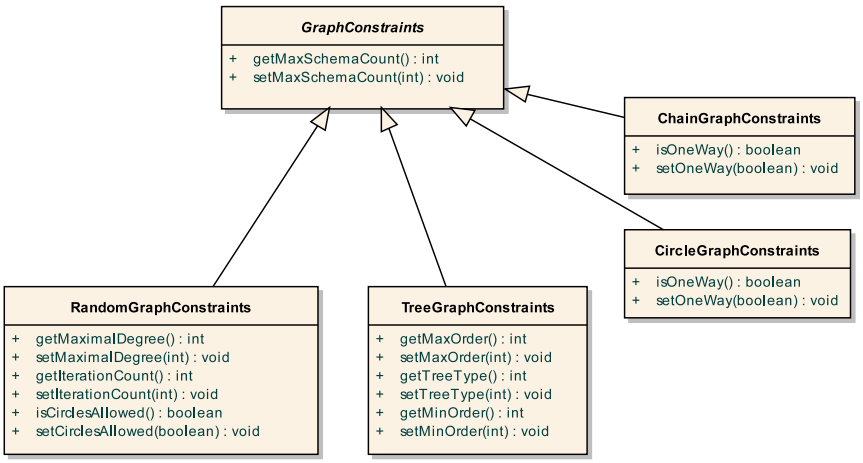


Abbildung 8.11: Constraints-Klassen für Graphen

Für das Interface *MappingGraph* existiert bereits die implementierende Klasse *DefaultMappingGraph*, die zur Speicherung eines Graphen verwendet werden kann. Es muss somit keine neue Klasse implementiert werden. Jedoch kann es je nach Graphentyp sinnvoll sein, eigene, davon abgeleitete Typen zu erzeugen. So lässt sich von einem *ChainMappingGraph* beispielsweise über die Methode *getStartSchema()* der Anfang der Kette oder von einem *TreeMappingGraph* die Wurzel via *getRoot()* ermitteln. In Abbildung 8.12 ist die Vererbungshierarchie der unterschiedlichen Mapping-Graphen illustriert.

Spezialisierte Graphenklassen für einzelne Graphentypen sind nicht zwingend nötig, erleichtern aber die Implementierung von Testfällen und eventuellen Experimenten, da der Zugriff auf ausgezeichnete Knoten, wie beispielsweise die Wurzel eines Baumes, ermöglicht wird.

Nachdem auf die Einzelheiten der Implementierung eingegangen wurde, beschäftigt sich das folgende Kapitel mit einer Reihe von Experimenten, welche mit dieser Implementierung durchgeführt wurden.

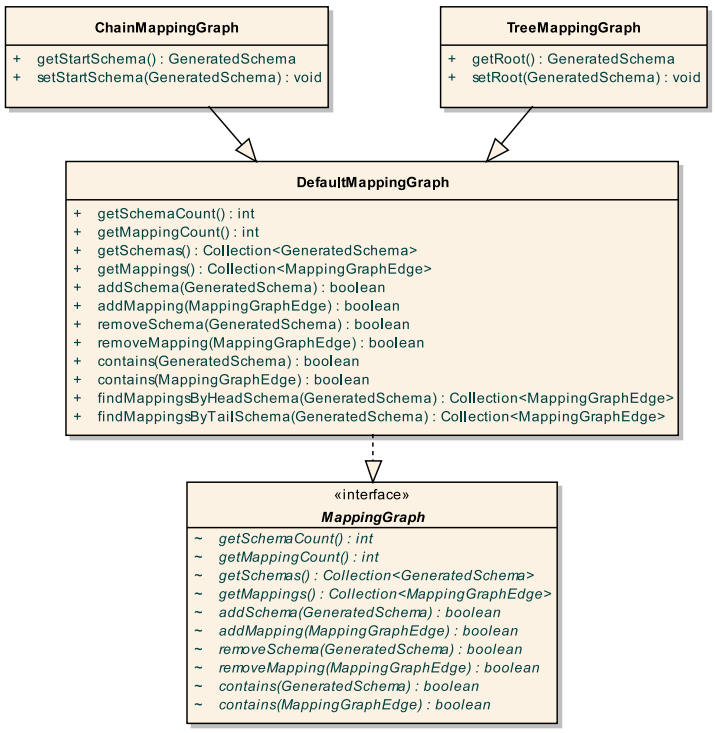


Abbildung 8.12: Klassen zur Kapselung von Graphen

9 Experimente

Die meisten der in dieser Arbeit vorgestellten Algorithmen sind nicht-deterministisch, da sie Gebrauch von einem Zufallszahlengenerator machen. Um die Auswirkungen einiger Parameter besser beurteilen zu können, wurden für dieses Kapitel eine Reihe von Experimenten durchgeführt. Der erste Abschnitt beschäftigt sich mit der Schema-Erzeugung und der Bildung von Mappings. Im darauf folgenden Abschnitt 9.2 werden Versuche mit der in Abschnitt 6.3.2 konstruierten Markov-Kette durchgeführt. Als Abschluss folgt eine Abschätzung der Laufzeit für die in den vorherigen Kapiteln vorgestellten Phasen zur Erzeugung eines PDMS.

9.1 Schema-Pool

Die Schema-Erzeugung erfolgt über die Anwendung der drei Algorithmen Teilmengenbildung, Normalisierung und Denormalisierung auf ein gegebenes Referenzschema. An dieser Stelle soll untersucht werden, wie sich die Anzahl der erzeugbaren Teilschemata mit der Anzahl der Relationen des Referenzschemas ändert. Hierzu wurden in jeweils 10 Läufen zufällige Referenzschemata mit einer Relationenanzahl $1 \leq m \leq 10$ erzeugt und aus diesen ein Schema-Pool berechnet. Für die erzeugten Teilschemata wurden folgende Einschränkungen (siehe Abschnitt 8.3.1) gesetzt:

- **Teilmengenbildung:** Schemata dürfen nur zwischen 1 und 5 Relationen enthalten.
- **Denormalisierung:** Join-Ketten dürfen maximal 5 Relationen umfassen.
- **Normalisierung:** Es werden maximal 5 Relationen des Referenzschemas zugleich normalisiert.

Diese Parameter sind die Grundlage aller hier vorgestellten Ergebnisse und sind als Standardwerte vorgegeben. Sie lassen sich jederzeit den Anforderungen entsprechend anpassen. Obige Einschränkungen waren nötig, da der Denormalisierungs-Algorithmus mitunter recht lange Join-Ketten in den Sichtdefinitionen und Peer-Mappings verursacht, welche dem verwendeten Datenbank Management System „HSQLDB“ Probleme bereiteten. Im Zuge dessen wurden die Parameter für die Teilmengenbildung und Normalisierung ebenfalls so gesetzt, dass nur eine Teilmenge der erzeugbaren Schemata generiert wird.

Die Abbildung 9.1 zeigt die Anzahl der erzeugten Schemata in Abhängigkeit der Relationenanzahl des Referenzschemas. Die verschiedenfarbigen Bereiche der Balken stellen die Anteile der jeweiligen Algorithmen an der Gesamtschemazahl dar. Gut zu erkennen ist der exponentielle Anstieg und die dabei dominierenden Anteile des Normalisierungs- und Denormalisierungs-Algorithmus. Der Teilmengen-Algorithmus liefert prinzipbedingt weniger Schemata, da die resultierenden Schemata zusammenhängend sein müssen und nicht zwischen allen Relationen des zufällig generierten Referenzschemas Fremdschlüsselbeziehungen bestehen.

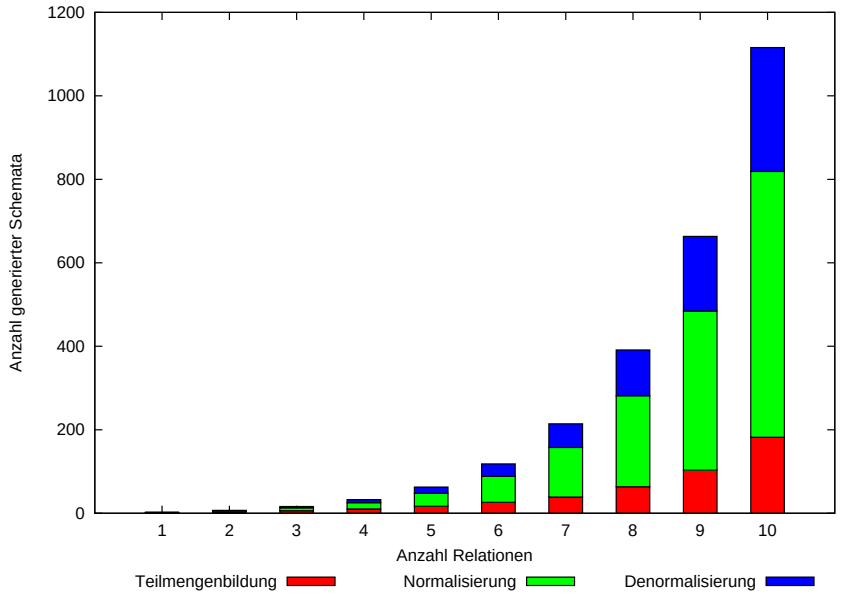


Abbildung 9.1: Anzahl generierter Schemata in Abhängigkeit vom Referenzschema

Die Schemata des Schema-Pools werden nach der Schema-Erzeugung durch Schema-Mappings miteinander verbunden. Während im vollständigen Graphen K_n zwischen allen Knotenpaaren eine Kante existiert, ist dies bei dem Schema-Pool nicht der Fall. Wie weit entfernt vom vollständigen Graphen der Schema-Pool ist, zeigt Abbildung 9.2.

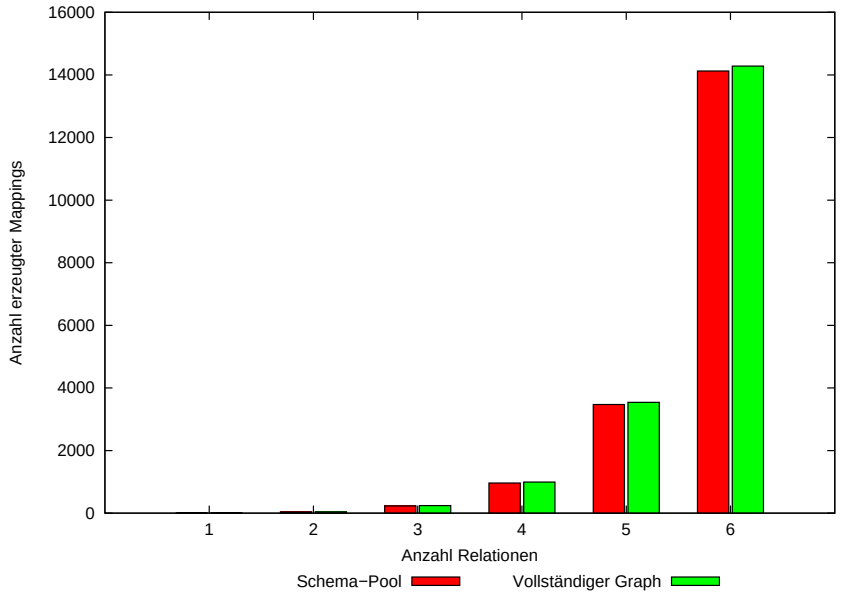


Abbildung 9.2: Anzahl generierter Mappings in Abhängigkeit vom Referenzschema

Die Berechnungen wurden für $1 \leq m \leq 6$ Relationen vorgenommen. Es ist gut zu erkennen, dass die Zahl der Kanten des Schema-Pools leicht unter der des K_n liegt. Die Abweichungen der Kantenanzahlen sind in Tabelle 9.1 dargestellt.

m	Abweichung
1	0%
2	4,8%
3	4,1%
4	3,0%
5	2,0%
6	1,2%

Tabelle 9.1: Abweichungen der Kantenanzahl vom K_n

Dabei ist m die Anzahl der Relationen des Referenzschemas. Die Anzahl der erzeugten Teilschemata ist n und entspricht der Anzahl der Knoten.

Die Anzahl der Schemata steigt exponentiell und somit auch die Laufzeit der Mapping-Suche. Der Speicherverbrauch für die vollständige Berechnung aller Mappings für $m = 7$ erfordert bereits über 768 MB Arbeitsspeicher und bricht nach 15 Minuten ab. Da die Größe des Schema-Pools mit der Anzahl der Relationen des Referenzschemas zu stark wächst, ist es nicht immer sinnvoll, alle möglichen Teilschemata zu berechnen. Aus diesem Grund wurde die Möglichkeit geschaffen, die maximale Größe des Pools zu begrenzen.

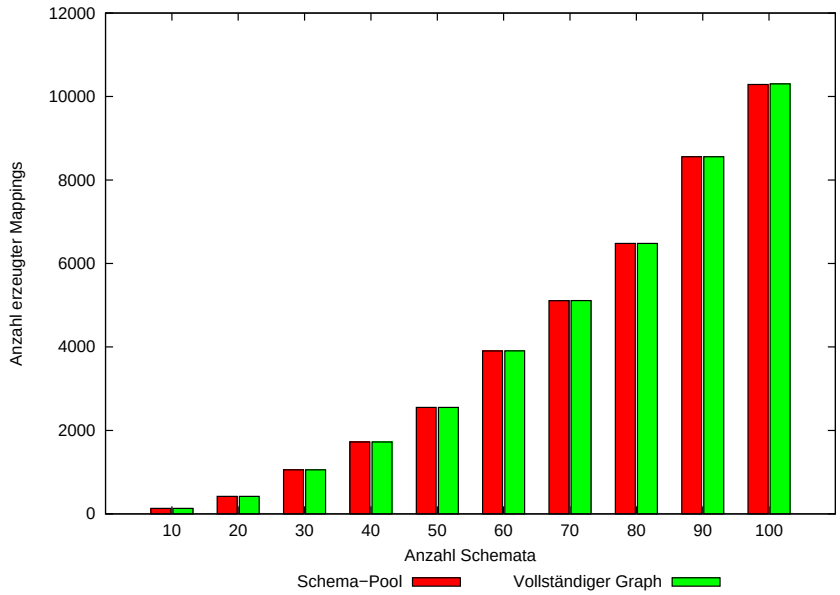


Abbildung 9.3: Anzahl generierter Mappings in Abhängigkeit von der Schema-Anzahl

Im nächsten Experiment wurde ermittelt, wie sich die Anzahl der Schema-Mappings für ein Referenzschema mit 10 Relationen mit der Größe des Schema-Pools entwickelt. Die

Größe des Pools wurde für jede Messung auf eine feste Anzahl Schemata festgelegt. Es erfolgt ebenfalls ein Vergleich mit dem vollständigen Graphen K_n. Das Ergebnis ist in Abbildung 9.3 dargestellt. Für jede Messung wurde die Größe des Schema-Pools gesetzt und zufällige Teilschemata aus dem gegebenen Referenzschema ermittelt. Es wird immer sichergestellt, dass die erzeugten Teilschemata zu gleichen Teilen mit dem Teilmengen-, dem Normalisierungs- und dem Denormalisierungs-Algorithmus erstellt wurden. Der Mapping-Graph des Schema-Pools ist in fast allen Fällen vollständig und entspricht dem K_n. Einzig bei $n = 100$ fehlen dem Schema-Pool 6 Kanten, was auf der Abbildung nicht mehr zu erkennen ist.

Für jedes Schemapaar aus dem Schema-Pool existiert demnach mit sehr hoher Wahrscheinlichkeit ein Schema-Mapping. Die in Abschnitt 6.3.3 getroffene Annahme, dass der Schema-Pool vollständig ist, erweist sich durch dieses Experiment als sehr realistisch.

9.2 Markov-Prozess zur Erzeugung von Zufallsgraphen

Die in Abschnitt 6.3.2 entwickelte Markov-Kette hat folgende Eingabeparameter:

- *maxIterationCount:* Anzahl auszuführender Markov-Schritte
- *maxNodeCount:*: Anzahl zu erzeugender Knoten
- *maxDegree:* Nicht zu überschreitender Maximalgrad
- *circlesAllowed:* Erlauben oder Verbieten von Zyklen

Der Markov-Algorithmus modifiziert eine Kette bzw. einen Baum mit *maxNodeCount* Knoten in *maxIterationCount* Schritten und beachtet dabei die Einhaltung der Parameter *maxDegree* und *circlesAllowed*. In den folgenden Unterabschnitten werden Experimente bezüglich der zu erwartenden Kantenanzahl, der Zyklizität und der Häufigkeit von Artikulationsknoten durchgeführt.

9.2.1 Untersuchung der Kantenanzahl

Die Grundoperationen des Algorithmus sind das Entfernen und Hinzufügen von Schema-Mapping-Kanten. Die Anzahl der Kanten wurde für verschiedene Maximalgrade erfasst. In Abbildung 9.4 ist ein typischer Verlauf einer Graph-Erzeugung dargestellt. Für *maxNodeCount* = 30, *maxDegree* = 3, *circleAllowed* = *true* und *maxIterationCount* = 10000 wurden pro Markov-Schritt die Anzahl der Kanten des aktuellen Graphen gezählt. Es ist zu erkennen, dass die Kantenanzahl, nach einem starken Anstieg von 30 auf ca. 90 Kanten, im Bereich zwischen 70 und 105 schwankt. Genau dieses Verhalten ist zu erwarten, denn unter den zufällig gewählten Knotenpaaren sind anfangs nur vergleichsweise wenige mit Kanten verbunden, weshalb zwischen ihnen neue eingefügt werden. Ab einem bestimmten Punkt stellt sich ein Gleichgewicht zwischen Knotenpaaren mit Kanten und solchen ohne Kanten ein. Die Anzahl der Kanten pendelt nach diesem Zeitpunkt um einen festen Wert.

Dieser Mittelwert ist der Erwartungswert des vorgestellten Zufallsexperiments. Es handelt sich um die zu erwartende Kantenanzahl für die vom Markov-Prozess erzeugten Graphen.

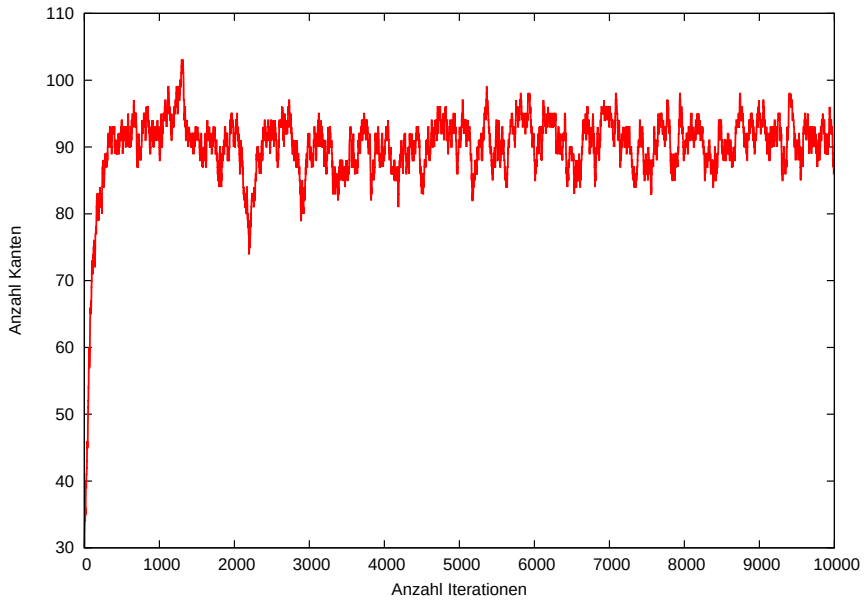


Abbildung 9.4: Typische Grapherzeugung. Anzahl Kanten gegen Anzahl Iterationen.

Die Abbildung 9.5 zeigt den Verlauf der Kantenanzahl für verschiedene Maximalgrade bei einer Knotenanzahl von 20.

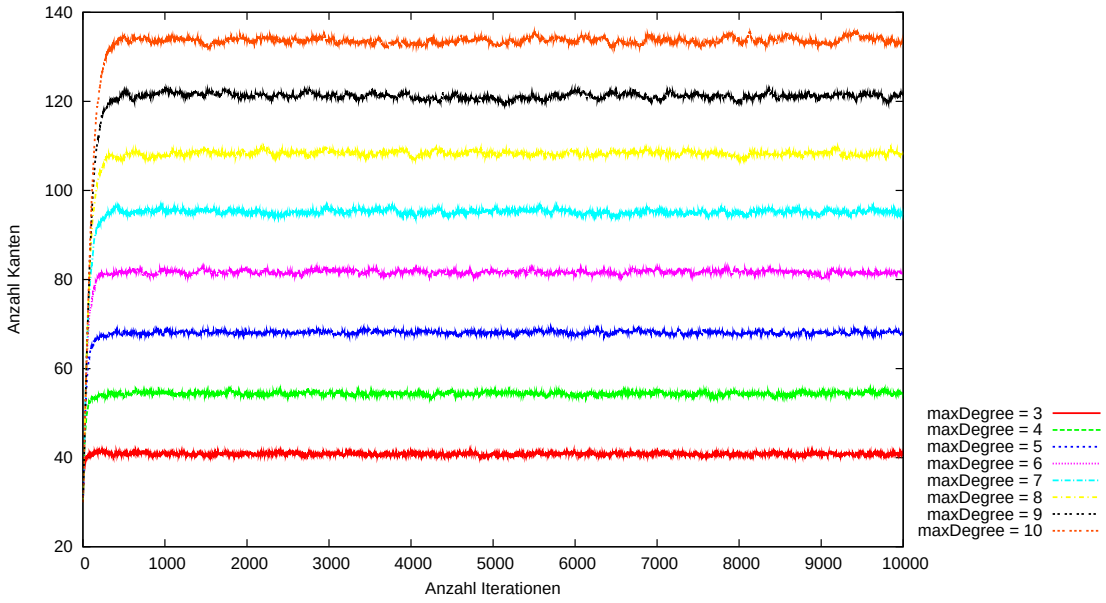


Abbildung 9.5: Anzahl Kanten gegen Anzahl Iterationen

Für die Graphik wurden Messwerte aus 50 Läufen pro Maximalgrad gemittelt. Es ist zu erkennen, dass sich der Mittelwert, um den die Kantenanzahl schwankt, mit dem Wert des Maximalgrades verändert. Die durchschnittliche Kantenanzahl über dem Maximalgrad ist in Abbildung 9.6 dargestellt. Die roten Balken zeigen die Messung für global vorgegebene Maximalgrade und die grünen Balken für lokal zugewiesene. Im letzteren Fall wird jedem Knoten im Startgraphen anfangs ein maximal erlaubter Grad zugewiesen. Der Durchnittswert wurde jeweils über die Kantenanzahlen zwischen dem 1000ten und 10000ten Iterationsschritt berechnet, um den anfänglich starken Zuwachs nicht zu berücksichtigen.

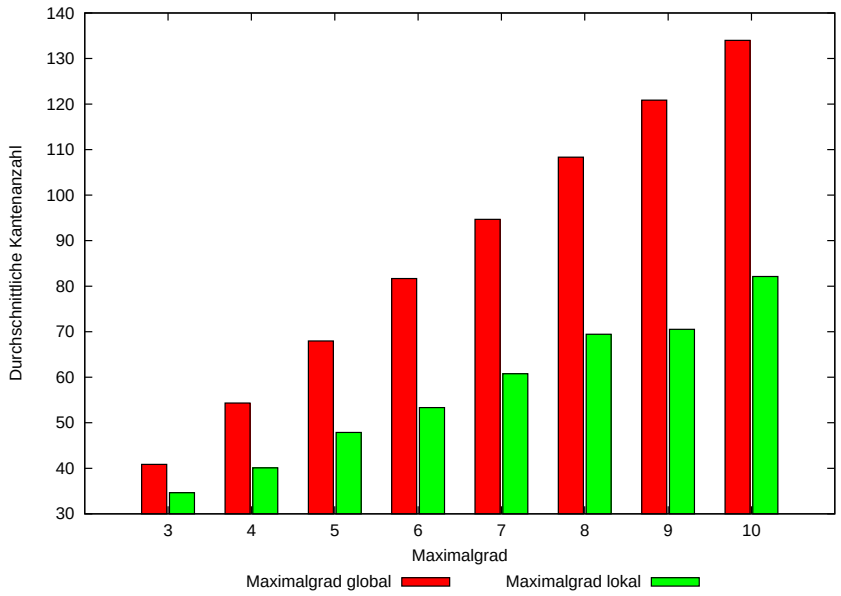


Abbildung 9.6: Anzahl Kanten für gegebene Maximalgrade

Anhand der Graphik erkennt man, dass die durchschnittliche Anzahl Kanten in Abhängigkeit zu dem globalen Maximalgrad linear wächst. Dies gilt somit auch für die Anzahl der Mappings im erzeugten PDMS. Weist man jedem Knoten des Startgraphen vor Beginn der Markov-Iterationen bereits einen maximal möglichen Grad zu, so steigt die Anzahl der Kanten flacher und unregelmäßiger an.

Dieses Experiment zeigt, dass die vom Markov-Prozess erzeugten Graphen in Abhängigkeit vom Maximalgrad und der Knotenanzahl eine bestimmte zu erwartende Kantenanzahl (Erwartungswert des Zufallsexperiments) aufweisen und Graphen mit wesentlich mehr oder weniger Kanten die Ausnahme darstellen.

9.2.2 Untersuchung auf Zyklizität

Nachdem der Einfluss des Maximalgrades auf die durchschnittliche Kantenanzahl untersucht wurde, soll nun der Zusammenhang von Maximalgrad und der Zyklizität der erzeugten Graphen ermittelt werden. Bei diesem Experiment werden nur gerichtete Kreise

entlang der Mapping-Pfade betrachtet. Es ist anzunehmen, dass bei Erhöhung des Maximalgrades die Wahrscheinlichkeit für die Entstehung von gerichteten Kreisen ebenfalls ansteigt, da die Größe der Kantenmenge mit dem Maximalgrad wächst. Das Ergebnis dieses Experiments ist in Abbildung 9.7 zu sehen.

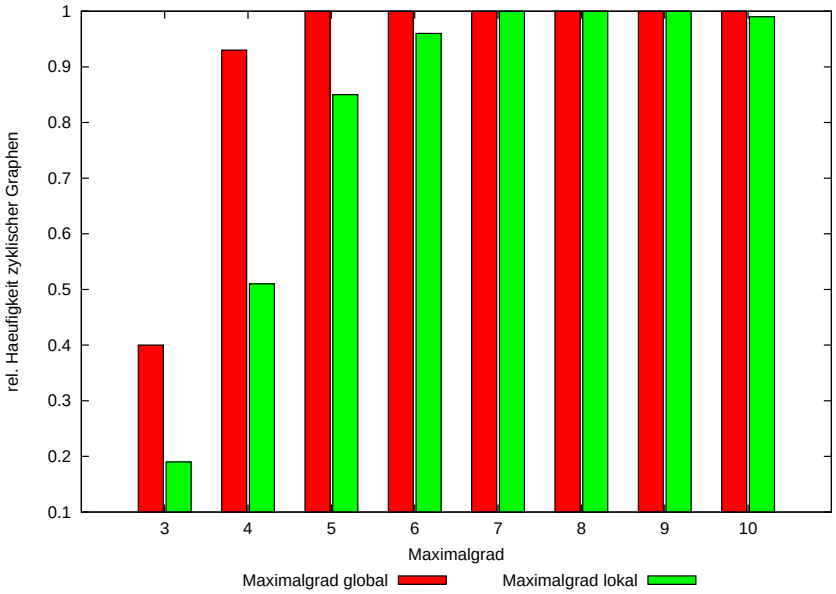


Abbildung 9.7: Häufigkeit zyklischer Graphen für verschiedene Maximalgrade

Es wurden für die globalen Maximalgrade 3 bis 10 jeweils 100 zufällige Graphen mit einer Knotenanzahl $maxNodeCount = 30$ und einer Iterationsanzahl von $maxIterationCount = 5000$ erzeugt und die Anzahl der Graphen mit gerichteten Kreisen gezählt. Die roten Balken geben für diesen Fall die Zyklen-Wahrscheinlichkeit im Interval $[0, 1]$ an. Es ist auffällig, dass bei einem globalen Maximalgrad von 3 die Wahrscheinlichkeit für einen gerichteten Kreis ca. 40% beträgt und diese bereits bei $maxDegree = 4$ auf über 90% ansteigt. Bei höheren globalen Maximalgraden liegen die Wahrscheinlichkeiten bei über 99% für einen zyklischen Graphen. Die grünen Balken repräsentieren die Messungen für den Fall, dass den Knoten vor Start des Markov-Prozesses lokale Maximalgrade zugewiesen werden. Laut Messung bewirkt dies eine verminderte Zyklenbildung bei gleichem Maximalgrad gegenüber der alternativen Methode.

9.2.3 Untersuchung auf Artikulationsknoten

In Kapitel 6 wurden so genannte Bottleneck-Graphen behandelt. Bottleneck-Knoten verbinden mehrere dicht vernetzte Bereiche miteinander und haben in einem Peer-Graphen somit eine hohe Netzwerklast zu tragen. Bei Entfernung eines derartigen Knotens kann der Graph in mehrere Zusammenhangskomponenten zerfallen. Man bezeichnet den Knoten dann als Artikulationsknoten.

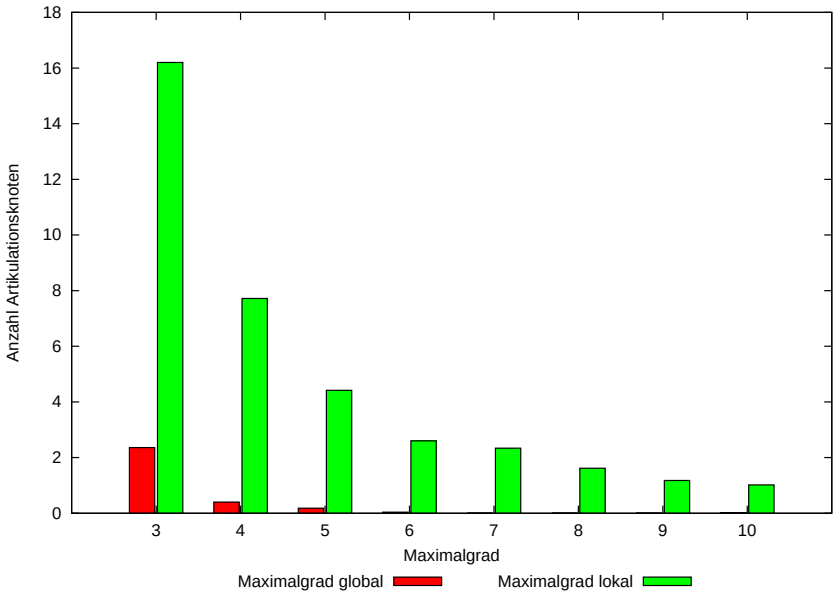


Abbildung 9.8: Anzahl Artikulationsknoten für verschiedene Maximalgrade

Im folgenden Experiment wird gemessen, wie viele Artikulationsknoten der vorgestellte Markov-Prozess in Abhängigkeit vom Maximalgrad durchschnittlich erzeugt. Es wurden in 100 Läufen jeweils Graphen mit 100 Knoten erzeugt. Das Ergebnis ist in Abbildung 9.8 dargestellt. Die roten Balken repräsentieren den Fall, dass ein globaler Maximalgrad vorgegeben wurde und die grünen Balken den Fall zufällig lokal zugewiesener Maximalgrade. Das Experiment zeigt, dass der implementierte Algorithmus zur Erzeugung beliebiger Zufallsgraphen durchaus verwendbar ist, um Graphen mit Artikulationsknoten zu erzeugen. Global zugewiesene Maximalgrade, die für alle Knoten einheitlich gelten, sorgen für eine geringe Anzahl von Artikulationsknoten und sind daher für deren Erzeugung ungeeignet. Wählt man den Maximalgrad für jeden Knoten individuell (lokal), so neigt der Algorithmus vermehrt dazu, Artikulationsknoten zu erzeugen.

9.3 Laufzeitmessungen

In diesem Abschnitt soll ein Gefühl dafür vermittelt werden, wie viel Zeit das Erzeugen verschieden großer Peer-Netze in Anspruch nimmt. Die Zeitmessungen des Generierungsprozesses erfolgen über folgende Phasen:

- **Schema-Erzeugung:** Ist die maximale Anzahl erzeugbarer Teilschemata kleiner als die geforderte Anzahl Peers, so hängt die Laufzeit nicht von der Anzahl der Peers ab, da alle Teilschemata erzeugt werden. Ist die Anzahl der möglichen Teilschemata größer als die geforderte Anzahl von Peers, werden nicht alle Teilschemata erzeugt, sondern nur genau so viele, wie benötigt werden. Dadurch ist hier eine lineare Laufzeit zur Anzahl der Peers zu erwarten.

- **Suche nach Mappings:** Da alle Schema-Paare aus dem Pool auf mögliche Mappings untersucht werden, ist die Laufzeit dieser Phase quadratisch abhängig von der Größe der Schema-Pools.
- **Graph-Erzeugung:** Die Laufzeit der Grapherzeugung hängt vom gewählten Algorithmus ab und ist im Vergleich zu den anderen Phasen stets sehr gering.
- **Zusammensetzen des PDMS:** In diese Phase fallen das Erzeugen von Projektionen, die Bereitstellung der Extensionen sowie das Generieren von Selektionen. Die Laufzeit hängt stark von der Extensionsgröße und der Beschaffenheit der Fremdschlüsselbeziehungen der Peer-Schemata ab. Eine genauere Abschätzung fällt hier schwer.
- **Übertragung auf reale Peers:** Eine Abschätzung kann hier nur experimentell ermittelt werden und hängt von den Skalierungsfähigkeiten des JXTA-Frameworks ab.

Die nachfolgenden Experimente wurden auf einem PC-System mit 1 GByte Arbeitsspeicher, AMD Athlon 64 3500+ Prozessor und Windows XP durchgeführt. Im ersten Experiment werden die Laufzeiten der PDMS-Erzeugung in Abhängigkeit von der geforderten Peer-Anzahl gemessen. Eine Übertragung auf real laufende Peers geht hier noch nicht in die Messung mit ein. In Abbildung 9.9 ist das Ergebnis dieses Experiments graphisch dargestellt. Der Zeitverbrauch der Suche nach Mappings für alle Schemapaare steigt quadratisch an, während der Zeitanteil für die Schema-Erzeugung nur linear wächst. Die Dauer der Extensionsberechnung variiert und ist abhängig von der zufällig gewählten Extensionsgröße der Peers, der benötigten Anzahl von Durchläufen, um die Peer-Extensionen konsistent zu machen (siehe Abschnitt 7.3.2) und von der Effizienz des verwendeten Datenbank Management Systems HSQLDB[1].

Die Messungen resultieren aus einmaligen Durchläufen und sollen nur einen Eindruck von der Laufzeit vermitteln. Für genauere Messungen dürfen keine Prozesse parallel laufen. Zusätzlich ist die mehrfache Wiederholung unter gleichen Bedingungen nötig. Dies war für die Messreihen nicht der Fall.

Das zweite Experiment schließt bei den Messungen den zeitlichen Aufwand für das Übertragen auf real laufende Peers mit ein. Es wurde parallel zum Testprogramm ein JXTA-Peer (siehe [14]) gestartet und auf diesem die nötigen PDMS-Peers erzeugt. Die Ergebnisse der Messung sind in Abbildung 9.10 dargestellt.

Während der Übertragungsvorgang für 250 PDMS-Peers auf eine Maschine noch erfolgreich verlief, wurden die Grenzen des oben erwähnten PCs bei 300 Peers bereits erreicht. Eine Übertragung war hier nicht mehr möglich.

[1] Projektionen, Joins und Selektionen werden über ein DBMS durchgeführt. Siehe [14] für weitere Informationen.

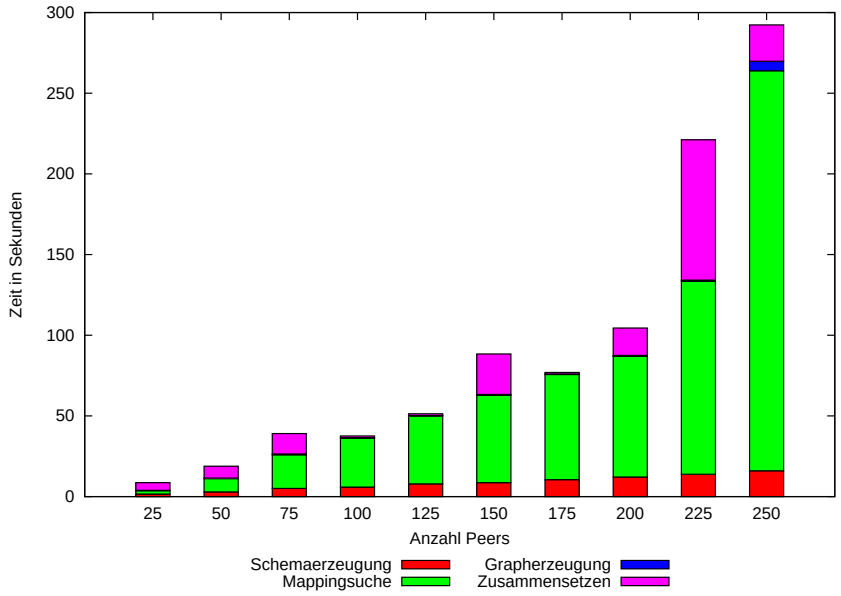


Abbildung 9.9: Zeitmessungen für die Erstellung von Peer-Graphen

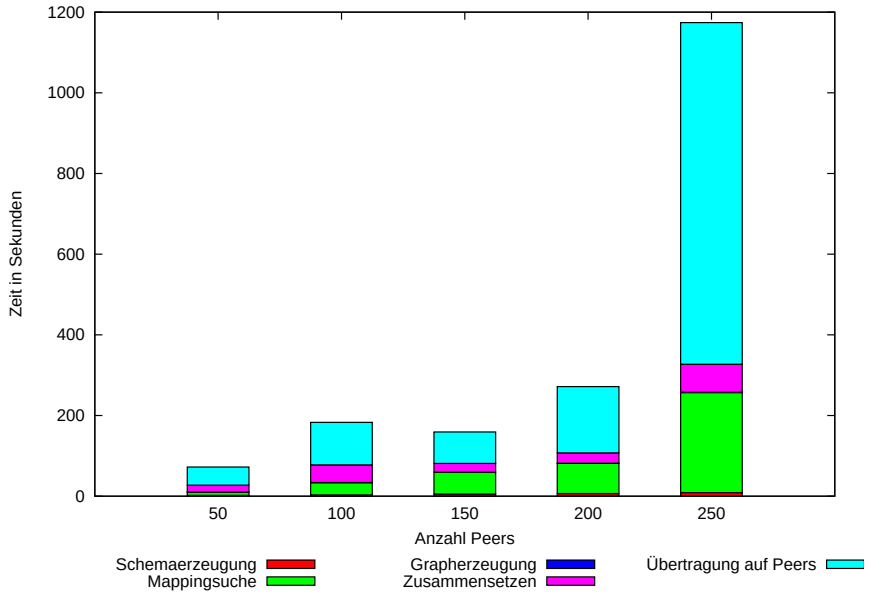


Abbildung 9.10: Zeitmessungen für die Erstellung verschieden großer PDMS

10 Fazit und Ausblick

In dieser Arbeit wurde eine Testumgebung entwickelt, mit der sich beliebige PDMS-Instanzen parametrisiert erzeugen und anfragen lassen. Es erfolgte eine Unterteilung der PDMS-Erzeugung in verschiedene Phasen. In Kapitel 4 wird dargestellt, wie aus einem relationalen Schema mit Hilfe der Algorithmen „Teilmengenbildung“, „Normalisierung“ und „Denormalisierung“ eine Menge neuer Schemata generiert werden kann. Dabei wird sichergestellt, dass erzeugte Schemata bestehende Fremdschlüsselbeziehungen des Referenzschemas beachten und eine Zusammenhangskomponente bilden. Zusätzlich wird beschrieben, wie Sichtdefinitionen zur Beschreibung der Schematransformationen zu bilden sind. Kapitel 5 zeigt auf, wie sich anhand dieser Sichtdefinitionen Schema-Mappings finden lassen. Dazu werden die Grundlagen von *Query containment* und *Containment mappings* aufgearbeitet und der Begriff des *schwachen Containments* eingeführt. Auf diese Weise gelingt es, Schema-Mappings mit Projektionen zu modellieren. Das anschließende Kapitel 6 behandelt die Erzeugung von Peer-Graphen aus einer Menge von Schemata und Schema-Mappings. Hierzu kommen unter anderem Markov-Ketten (zufällige stochastische Prozesse) zum Einsatz. In Kapitel 7 wird dargestellt, wie Peer-Graphen durch Hinzufügen von Projektionen und Selektionen in den Mappings sowie durch Erzeugung von Peer-Extensionen in eine konkrete PDMS-Instanz überführt werden. Im Folgenden werden einige Anregungen für zukünftige Erweiterungen der Testumgebung einschließlich des *System P* gegeben.

Verbesserungen der Schema-Erzeugung

Der Normalisierungsalgorithmus beachtet keine funktionalen Abhängigkeiten, sondern teilt Relationen des Referenzschemas in zwei Teilrelationen auf. Bessere Ergebnisse lassen sich durch Angabe funktionaler Abhängigkeiten erzielen. Damit lassen sich über den Normalisierungsalgorithmus zur Erzeugung der 3. Normalform (siehe [16]) intuitivere Zerlegungen der einzelnen Relationen finden. Des Weiteren sind Modellierungs-Unterschiede der Art „Relation vs. Attribut“, „Attribut vs. Wert“ oder „Relation vs. Wert“ zwischen Referenzschema und erzeugten Schemata eine neue Herausforderung. Mit den vorgestellten Datalog-Regeln für die Sichtdefinitionen kommt man in diesem Fall nicht jedoch mehr aus. Es sind Ansätze wie beispielsweise SchemaSQL (siehe [9]) notwendig. Bisher werden Attribute des Referenzschemas 1:1 auf Attribute der generierten Schemata abgebildet. Zusätzliche schematische Heterogenität lässt sich durch n:1- und 1:m-Korrespondenzen erreichen. Beispielsweise könnten die Attribute Name und Vorname eines Referenzschemas gemeinsam auf ein Attribut Name in einem generierten Schema durch Konkatenation abgebildet werden. Es sind somit Transformationsfunktionen nötig, die Daten eines Quellschemas in Daten des Zielschemas umwandeln können (siehe Abschnitt 5.1).

Verbesserungen der Mapping-Erzeugung

Die Mapping-Erzeugung in Kapitel 5 behandelt ausschließlich die beiden Spezialfälle der LaV- und GaV-Mappings und betrachtet Schemapaare. Hier ist zukünftig die Unterstüt-

zung von GLaV-Mappings wünschenswert. Diese lassen mehrere Teilziele sowohl im Kopf als auch im Rumpf des Mappings zu. Weiter sollte die Möglichkeit geschaffen werden, Mappings nicht nur zwischen Schemapaaren definieren zu können, sondern mehr als 2 beteiligte Peers zuzulassen. Werden, wie oben beschrieben, Transformationsfunktionen in den Sichtdefinitionen der generierten Mappings verwendet, so müssen auch Peer-Mappings um solche Funktionen erweitert werden. Das Gleiche gilt, wenn zwischen unterschiedlich modellierten Schemata Mappings ermöglicht werden sollen.

Verbesserungen der Graphen-Erzeugung

Neben der Schema- und Mapping-Erzeugung sind auch bezüglich der Graphen-Generierung zukünftige Arbeiten denkbar. In Kapitel 6 wurden, neben den bereits implementierten Graphen Kette, Kreis, Baum und „Zufälliger Graph“, Graphen mit Bottlenecks bzw. Graphen mit Bow-Tie-Struktur vorgestellt. Eine Implementierung dieser und weiterer Graphen wäre eine sinnvolle Ergänzung. Weitere Markov-Prozesse könnten neben der Kantenmenge auch die Menge der Knoten bzw. Schemata verändern. Die Kombination mehrerer Markov-Ketten, welche sich gegenseitig verwenden sobald die notwendigen Eigenschaften erfüllt sind, ist ebenfalls eine Variante. Des Weiteren ist nicht klar, wie oft der Markov-Prozess für einen zufällig gleichverteilt erzeugten Graphen ausgeführt werden muss. In [12] wird dazu ein möglicher Ansatz über die Verknüpfung von Markov-Ketten vorgestellt.

Verbesserungen der Daten- und Selektionsprädikat-Erzeugung

An letzter Stelle seien noch Anstöße für Erweiterungen bezüglich der Datenverteilung und der Generierung von Selektionsprädikaten gegeben. Der bisherige Ansatz zur Datenerzeugung verteilt eine gegebene Extension des Referenzschemas auf die erzeugten Peers. Da alle Daten aus einer gemeinsamen Quelle kommen, treten, bis auf durch Projektionen hervorgerufene `NULL`-Werte, keine Datenkonflikte auf. Durch den Einbau von Mechanismen zur Erzeugung verschmutzer Daten lassen sich weitere Probleme aus der Praxis, wie Konfliktlösung und Duplikaterkennung, in *System P* modellieren. Die Selektivität der Peer-Mappings ist bisher nur bezüglich der Tupel der lokalen Quelle eines Peers definiert. Des Weiteren enthalten Peer-Mappings stets genau ein Selektionsprädikat. Langfristig sollten mehrere Selektionsprädikate erlaubt werden und die eingeschränkte Selektivitätsdefinition dieser Arbeit verworfen werden. Die zu erwartende Vollständigkeit eines Mappings könnte durch Sampling und dem Aufbau von Histogrammen während der Lebenszeit des zugehörigen Peers gemessen werden.

Mit der in dieser Arbeit entwickelten Testumgebung lassen sich beliebig große Peer Data Management Systeme erzeugen und auf konkrete Peers des *System P* verteilen. Auf diese Weise können neue Pruningstrategien erprobt und die Auswirkungen verschiedener Netz-Topologien getestet werden.

Literaturverzeichnis

[1] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area internet bottlenecks, 2003.

[2] J. Bauckmann. *Schemaintegration auf der Grundlage von Schema-Mappings.* Diplomarbeit, 2005.

[3] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 309–320, 2000.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, 1977.

[5] Alain Denise, Marcio Vasconcellos, and Dominic J.A. Welsh. The random planar graph. *Congressus Numerantium*, 1996.

[6] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[7] A. Y. Halevy, Z. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, 2003.

[8] G.E. Krasner. A cookbook for using the modelview-controller user interface paradigm in smalltalk. *Journal of Object-Oriented Programming 1(3)*, 1988.

[9] Laks V. S. Lakshmanan, Fereidoon Sadri, and Subbu N. Subramanian. SchemaSQL: An extension to SQL for multidatabase interoperability. *Database Systems*, 2001.

[10] F. Legler. *Datentransformation mittels Schema Mapping.* Diplomarbeit, 2005.

[11] U. Leser. *Query Planning in Mediator Based Information Systems.* Dissertation, 2000.

[12] J. Propp and D. Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics, 1995.

[13] Armin Roth and Felix Naumann. Benefit and cost of query answering in PDMS. In *Proc. of the Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2005.

[14] M. Schweigert. *Entwurf eines Peer Data Management Systems mit Steuerungs- und Simulationskomponente.* Diplomarbeit, 2006.

[15] A. Taraz. *Zufällige diskrete Strukturen.* Vorlesungsskript, 2005.

[16] R. Elmasri und S. B. Navathe. *Grundlagen von Datenbanksystemen, 3. Auflage.* Pearson Studium, 2002.

[17] T. Emden-Weinert und S. Hougardy und B. Kreuter und H. J. Prömel und A. Steger. *Einführung in Graphen und Algorithmen.* Vorlesungsskript, 1996.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Kapitel 2 entstand in Zusammenarbeit mit Martin Schweigert.

Berlin, den 30.01.2006

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 30.01.2006