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

Diplomarbeit

Entwurf eines Peer Data Management Systems mit Steuerungs- und Simulationskomponente

Martin Schweigert

31. Januar 2006

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

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Der Zugriff auf Informationen bestehender heterogener Quellen ist für viele Geschäftsprozesse ein wachsendes Problem. Der klassische Ansatz der Datenintegration, diese Quellen mit Hilfe eines globalen Datenschemas zu vereinen, stößt mit steigender Anzahl der Quellen schnell an seine Grenzen. Der Aufwand, ein globales Schema zu erzeugen und bei Änderungen zu warten, wird sehr groß. Auch ist es schwierig eine gemeinsame Semantik der zu integrierenden Schemata zu finden. Somit sind integrierende Systeme bei vielen Quellen meist unflexibel und teuer in der Wartung. Ebenso führt der Ausfall der integrierenden Komponente zum Ausfall des Gesamtsystems. Anders bei einem Peer Data Management System (PDMS), das aus einem Netzwerk mehrerer autonomer Quellen (Peers) besteht. Es besitzt keine globale Instanz und kein globales Schema. Weitere Peers werden leicht durch Angabe eines oder mehrerer Mappings zu bestehenden Peers Teil dieses Netzwerkes. Peer Data Management Systeme können somit schnell aufgebaut, kostengünstig gewartet werden und sind robust gegenüber dem Ausfall einzelner Komponenten. Ihr Einsatz bietet sich daher in Krisensituationen oder bei Firmenzusammenschlüssen an.

Ein Nutzer eines PDMS kann Anfragen an einen ihm bekannten Peer stellen. Dieser Peer beantwortet die Anfragen anhand seiner eigenen Daten und kann durch Ausnutzung der Mappings weitere Peers in die Anfragebearbeitung einbeziehen. Dabei muss dem Nutzer nicht bekannt sein, welche Mapping-Pfade und Datenquellen in einem PDMS zur Beantwortung seiner Anfrage genutzt werden. Dieser an sich positive Aspekt führt zu mehreren Problemen. Zum einen kann die Qualität der Anfrageergebnisse aufgrund der Verwendung sehr langer Mapping-Pfade sinken, zum anderen kann die Benutzung aller verfügbaren Pfade sehr ineffizient oder praktisch unmöglich sein. Der Einsatz von effizienten Anfragebearbeitungsstrategien wird somit notwendig.

Gegenstand dieser Arbeit ist die Entwicklung eines Peer Data Management Systems (*System P*). Mit dessen Hilfe soll ein mögliches Design für eine Implementierung eines PDMS aufgezeigt werden. Dabei wird auf den bestehenden Anfrageplaner eines PDMS-Simulators von Armin Roth zurückgegriffen. Im Gegensatz zu diesem Simulator, der den Anfrageplan erstellt, soll das *System P* Anfragen ausführen und dabei auf gespeicherte Daten einzelner Peers zugreifen. Des Weiteren sollen mit dem *System P* verschiedene Anfragebearbeitungsstrategien implementiert und analysiert werden.

Struktur der Arbeit
Im Kapitel 2 werden die wesentlichen Bestandteile und Begriffe eines PDMS erläutert. Anschließend folgt im Kapitel 3 eine Beschreibung der Implementierung des *System P*. Die wichtigste Komponente in einem PDMS bildet die Anfrageplanung. Sie wird im Kapitel 4 beschrieben. In Kapitel 5 wird deutlich, warum die vollständige Beantwortung einer Anfrage häufig problematisch ist und welche Strategien daher verfolgt werden können. Abschließend werden im Kapitel 6 mit Hilfe von Experimenten die Auswirkungen dieser Strategien im Hinblick auf Kosten und Qualität der Anfrageergebnisse untersucht.

Danksagung

An dieser Stelle, möchte ich Prof. Dr. Felix Naumann, Juniorprofessor für Informationsintegration an der Humboldt-Universität zu Berlin, sowie meinem Betreuer Dipl.-Ing. Armin Roth für die wertvollen Anregungen und Diskussionen danken. Ebenso gilt mein Dank meinem Kommilitonen und Freund Tobias Hübner vor allem für die gegenseitige Unterstützung während des gesamten Studiums und für die gute Zusammenarbeit an den 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.

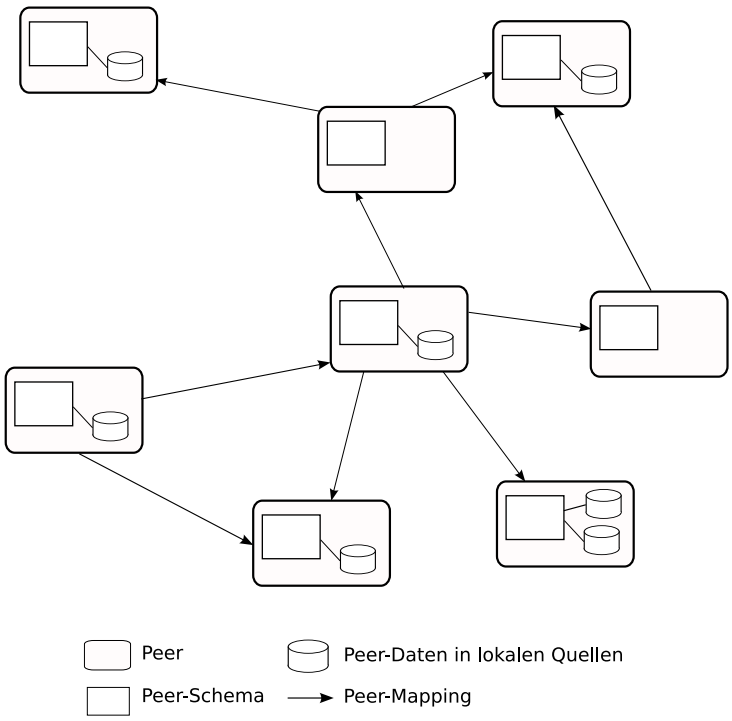

Abbildung 2.1: Ein Peer Data Management System

Dieses Kapitel wurde von Martin Schweigert und Tobias Hübner gemeinsam im Rahmen der Diplomarbeiten „Entwurf eines Peer Data Management Systems mit Steuerungs- und Simulationskomponente“ und „Entwicklung einer Testumgebung für ein Peer Data Management System“ [6] 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.

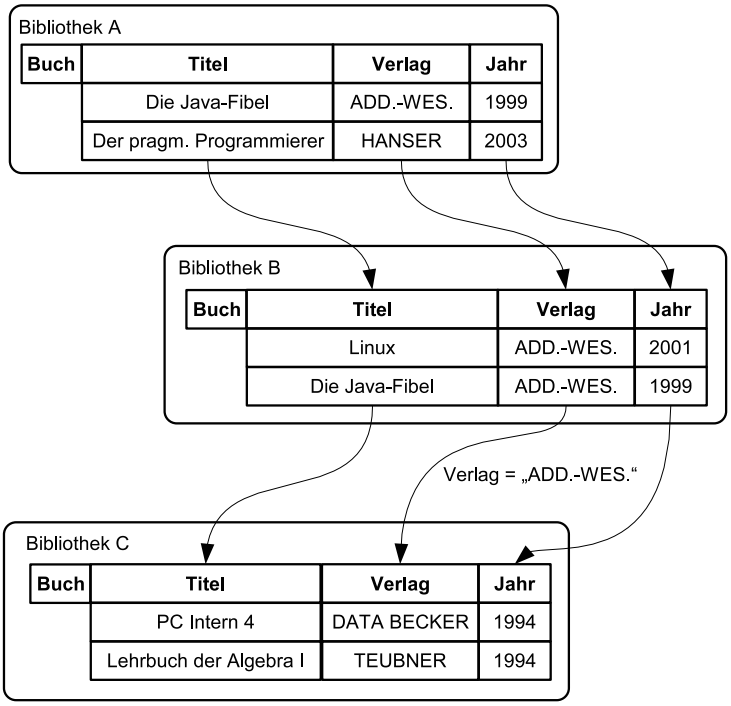

Abbildung 2.2: Informationsverlust durch Mappings

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.

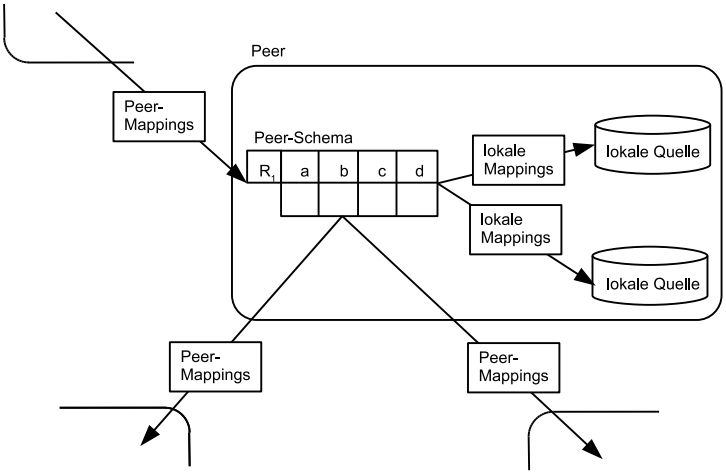

Abbildung 2.3: Aufbau eines Peers

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.*

Wie eingangs beschrieben, bildet dieses Kapitel die Grundlage für eine Diplomarbeit von Tobias Hübner [6] sowie für die vorliegende Arbeit. Während Tobias Hübner beschreibt wie eine PDMS-Instanz[1] aus einem gegebenen Schema automatisiert erzeugt werden kann, wird in dieser Arbeit auf die Implementierung eines PDMS und die verwendeten Strategien zur Anfragebearbeitung eingegangen.

[1] Es werden Peers, ihre lokalen Quellen, lokale und Peer-Mappings generiert.

3 Implementierung

Den ersten Teil dieser Diplomarbeit stellt die Implementierung eines Peer Data Management Systems dar. Dabei sollen die Peers sämtliche im vorherigen Kapitel beschriebenen Fähigkeiten besitzen. Insbesondere sollen sie Anfragen ausführen, auf Daten in den lokalen Quellen zugreifen können, Anfrageergebnisse liefern und die Möglichkeit bieten die Anfragebearbeitung im Hinblick auf die im Kapitel 5 beschriebenen Pruningstrategien zu analysieren. Dabei muss eine Verteilung der Peers auf mehrere Rechner möglich sein, um große PDMS-Instanzen[1] mit mehreren hundert Peers testen zu können. Diese PDMS-Instanzen sind unter anderem für die Experimente in Kapitel 6 notwendig.

In diesem Kapitel wird zunächst auf die Architektur der Implementierung und eines einzelnen Peers eingegangen. Anschließend wird beschrieben, wie die Peers miteinander kommunizieren und sich zu einem Peer-to-Peer (im Folgenden P2P)-Netzwerk zusammenschließen. Im Abschnitt 3.4 werden die Fähigkeiten erläutert, die eine Implementierung eines Peer besitzen muss, um die geforderten Funktionalitäten erfüllen zu können. Abschließend wird auf die Anbindung der lokalen Quellen und das Verwenden der bestehenden Funktionalität des Simulators sowie auf Testfälle der Implementierung eingegangen.

3.1 Verwendete Technologien

Das *System P* wurde komplett in Java implementiert. Die Vorteile von Java sind allgemein bekannt. Neben der Plattformunabhängigkeit besteht vor allem der Zugriff auf viele bereits vorhandene und größtenteils kostenlose Komponenten. So sind, neben der Bibliothek *JUnit*[2] zum Testen der Implementierung und Durchführung von automatisierten Experimenten, vor allem *JXTA*[3], *HSQLDB*[4] und *XStream*[5] hervorzuheben. *JXTA* bietet ein Framework für P2P-Anwendungen. Im Abschnitt 3.2.4 wird beschrieben, welche Funktionen davon im *System P* genutzt wurden.

HSQLDB

HSQLDB ist ein komplett in Java realisiertes relationales Datenbankmanagement-System. Es gibt verschiedene Arten, eine Verbindung zu einer HSQLDB-Datenbank aufzubauen. Interessant für die Verwendung im *System P* ist die Möglichkeit, die Datenbank innerhalb einer JVM[6] und komplett im Hauptspeicher laufen zu lassen. Dies bietet den Vorteil, dass die Datenbank sehr performant und denkbar einfach zu nutzen ist - sie startet in derselben JVM wie auch das *System P*.

---

1 Mit PDMS-Instanz ist der Zusammenschluss mehrerer Peers mit Hilfe eines Peer Data Management Systems gemeint (analog zu Datenbank und DBMS).

2 http://www.junit.org

3 http://www.jxta.org

4 http://hsqldb.org/

5 http://xstream.codehaus.org

6 Java virtual machine

Beispiel 3.1 *Aufbau einer HSQLDB-Datenbankverbindung*

```
java.sql.Connection con = java.sql.DriverManager.getConnection("jdbc:hsqldb:mem:DB_NAME", "sa", "");
```

XStream

Zum Serialisieren und Deserialisieren von Java Objekten in XML, wurde im *System P* die Bibliothek *XStream* verwendet. Das Serialisieren und Deserialisieren in einen UTF-8 kodierten String ist nötig, um Java-Objekte in JXTA-Nachrichten über ein Netzwerk transportieren zu können. Die Verwendung von XML bietet sich dafür an und erlaubt es, die übertragenen Daten beim Debugging lesbar darzustellen.

Beispiel 3.2 *Serialisieren und Deserialisieren mit XStream*

```
XStream xstream = new XStream();
String asXML = xstream.toXML(myObject);
Object myObjectClone = xstream.fromXML(asXML);
```

Im Beispiel 3.4 ist die serialisierte Form eines Java-Objektes zu sehen.

3.2 Architektur des *System P*

Grundsätzlich besteht das *System P* aus einer Menge gleichartiger Peers, welche die im vorherigen Kapitel beschriebenen Funktionen implementieren. Diese Art der Peers wird im Folgenden als PDMS-Peer bezeichnet. Da ein PDMS-Peer im *System P* weder eine grafische Benutzeroberfläche besitzt, noch die Möglichkeit der Interaktion über die Kommandozeile bietet, ist der im Abschnitt 3.2.2 beschriebene Monitor-Peer nötig. Er bietet Zugriff auf die Funktionen der PDMS-Peers. Somit ist der Monitor-Peer im *System P* zum Aufbau eines PDMS und zum Anfragen der Peers notwendig. Er ist aber nicht Teil des PDMS, da dieses auch ohne die Existenz des Monitor-Peers funktioniert, gäbe es eine direkte Interaktionsmöglichkeit mit den PDMS-Peers. Die Abbildung 3.1 zeigt einen groben Überblick über die Architektur des *System P*.

Im *System P* können innerhalb einer JVM mehrere PDMS-Peer-Instanzen laufen. Der Grund dafür wird in Abschnitt 3.2.4 erläutert. Dadurch kommt es zu verschiedenen Arten der Kommunikation der Peers untereinander. Peers, die in derselben JVM liegen, tauschen auf anderem Wege Daten aus, als Peers die in verschiedenen JVMs liegen, da hier eine Übertragung über das JXTA-Netzwerk erfolgen muss. In Abschnitt 3.4.1 wird auf die unterschiedlichen Übertragungswege eingegangen. Um die Architektur des *System P* genauer zu beschreiben, werden zunächst dessen Komponenten, die PDMS-Peers und Monitor-Peers, betrachtet.

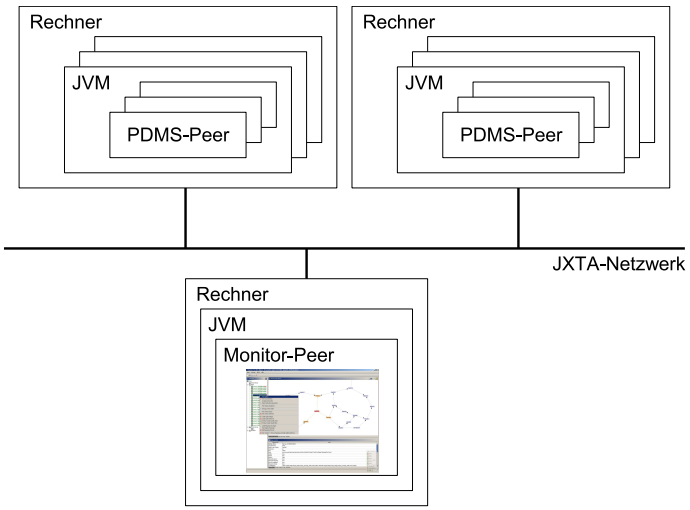

Abbildung 3.1: Architekturüberblick des *System P*

3.2.1 Architektur eines PDMS-Peers

Die Architektur eines PDMS-Peers folgt dem bereits im vorherigen Kapitel im Abschnitt 2.2 beschrieben Aufbau eines Peers. Die Abbildung 3.2 verdeutlicht die Einordnung und Funktion der einzelnen Module.

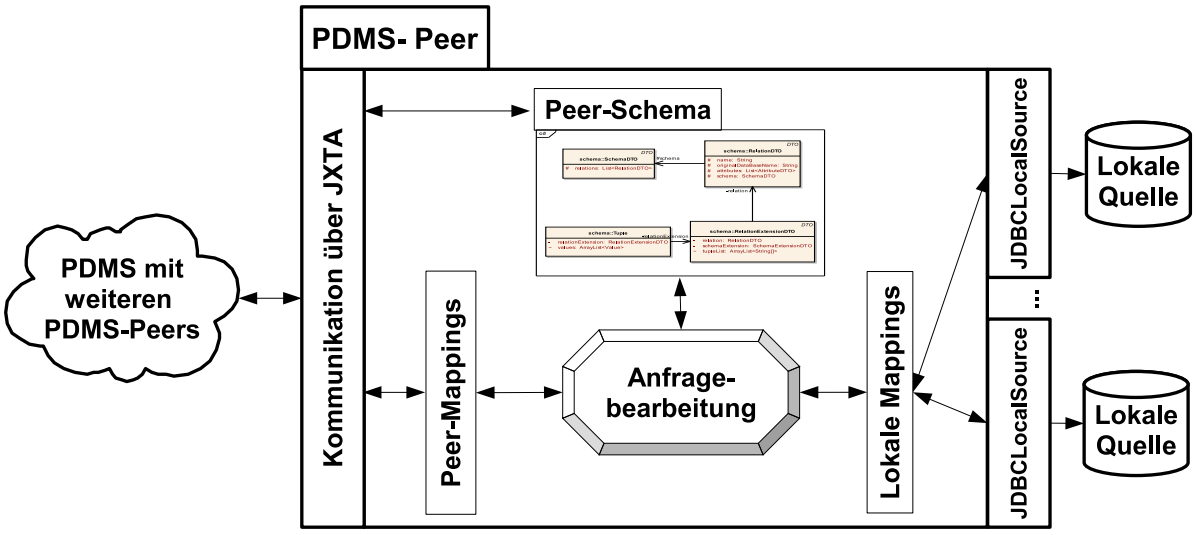

Abbildung 3.2: Peer im *System P*

Ein Peer besitzt die Möglichkeit zur Kommunikation mit weiteren PDMS-Peers in einem PDMS. Er kann Anfragen gegen sein Peer-Schema entgegennehmen. Diese Anfragen werden in der Komponente *Anfragebearbeitung* ausgeführt, welche über lokale Mappings auf lokale Quellen zugreifen kann. Ebenso können zur Anfragebearbeitung Peer-Mappings genutzt werden. Mit ihrer Hilfe werden Anfragen umformuliert und an weitere PDMS-Peers

gestellt. Anfragen, die durch die Anfrageumformulierung von ein Peer weiteren Peers stellt, werden im folgenden Peer-Anfragen genannt.

3.2.2 Monitor-Peer

Der Monitor-Peer ist ein spezieller Peer im *System P*. Er dient zur Verwaltung der PDMS-Peers, ermöglicht es Anfragen zu stellen und kann Informationen für Analyse der Anfragebearbeitung liefern. Dazu sucht der Monitor-Peer nach dem Start nach allen verfügbaren PDMS-Peers und stellt diese grafisch dar. Zusätzlich zeigt er Informationen einzelner Peers an. Abbildung 3.3 zeigt einen Ausschnitt der dargestellten Informationen.

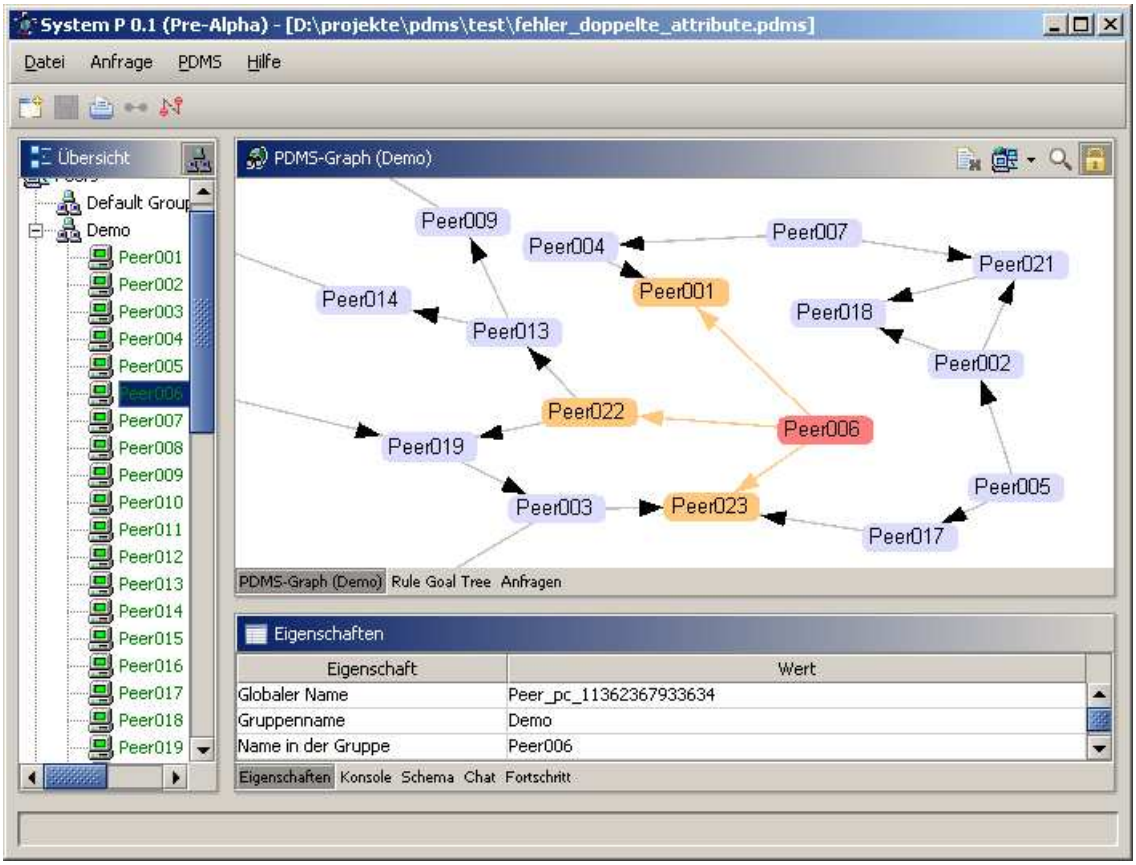

Abbildung 3.3: Monitor-Peer im *System P*

Im Monitor-Peer wird der Graph einer PDMS-Instanz dargestellt. Die Knoten stellen dabei die vorhandenen PDMS-Peers dar. Kanten zwischen den Peers deuten die Peer-Mappings an. Eine gerichtete Kante von einem *Peer X* zu einem *Peer Y* wird genau dann eingezeichnet, wenn *Peer X* ein Peer-Mapping besitzt, das *Peer Y* enthält und somit Anfragen an *Peer X* nach einer Umformulierung an *Peer Y* gestellt werden können. In der Abbildung ist dies zum Beispiel zwischen *Peer006* und *Peer023* der Fall.

Auf der linken Seite der Abbildung 3.3 des Monitor-Peers ist eine Übersicht aller verfügbaren Peers zu sehen. Bei der Wahl eines Peers in dieser Ansicht werden im unteren Bereich seine Eigenschaften dargestellt. Dies sind sein eindeutiger Name, sein Name innerhalb einer Gruppe (siehe folgender Abschnitt 3.2.3), Informationen über seinen Status (siehe Abschnitt 3.3.2), sein Peer-Schema und eine Auflistung seiner lokalen und Peer-Mappings.

Neben der Darstellung aller PDMS-Peers bietet der Monitor-Peer auch die Möglichkeit eine bestehende PDMS-Instanz anzupassen. Der Monitor-Peer bietet dazu den Zugriff auf das im Abschnitt 3.4 beschriebene Interface eines Peers. Dieses Interface bietet die folgenden Möglichkeiten:

- Anfragen an den Peer stellen
- Neue Peers anlegen, bestehende Peers herunterfahren
- Peer-Schema setzen, abfragen und löschen
- Mappings setzen, abfragen und löschen
- Lokale Quellen zuweisen und löschen

Weiterhin ermöglicht es der Monitor-Peer, die Anfragebearbeitung zu visualisieren und zu analysieren. Die Visualisierung der Anfragebearbeitung wird im Abschnitt 4.4 erläutert.

Der Monitor-Peer ist kein Bestandteil einer PDMS-Instanz. In einem *echten* kommerziellen Peer Data Management System würde es diese zentrale Komponente wahrscheinlich nicht geben, da sie dem P2P-Gedanken widerspricht und die Vorteile einer verteilten Architektur, wie sie im Abschnitt 2.1 beschrieben sind, zunichte macht. Im *System P* wird diese Komponente jedoch für den Aufbau einer PDMS-Instanz und die Anfrageanalyse benötigt.

3.2.3 Einteilung in Gruppen

Im *System P* ist es prinzipiell möglich, dass mehrere PDMS-Instanzen zeitgleich nebeneinander laufen. Die Peers eines PDMS bilden dann eine eigene Gruppe. Diese Gruppe zeichnet sich durch einen eindeutigen Namen aus, in der Peers einen neuen eindeutigen Namen erhalten können. Auf diesen Namen innerhalb einer Gruppe beziehen sich dann auch die Namen der Peer-Mappings. Es ist somit möglich, dass ein Peer „PeerX“ in einer Gruppe „PDMS1“ den Namen „Peer001“ annimmt, auf den sich dann alle Peer-Mappings innerhalb des „PDMS1“ beziehen. Dieses Vorgehen hat den Vorteil, dass der Peer global über seinen Namen „PeerXYZ“ innerhalb der grafischen Benutzeroberfläche des Monitor-Peers identifiziert werden kann und die Namen der Peers innerhalb eines PDMS frei wählbar bleiben. Ein Peer kann dabei immer nur einer Gruppe angehören.

Vor allem für automatisierten Experimente ist diese Funktionalität wichtig. Gäbe es nicht die Möglichkeit, den Namen eines Peers in einer PDMS-Instanz zu ändern, müsste beim

Start eines PDMS-Peers ein eindeutiger Name festgelegt werden, der auch in den Peer-Mappings verwendet wird. Dies würde den Aufbau eines PDMS und damit die Experimente komplizierter machen, da es eine zentrale Instanz geben müsste, die die Verteilung der eindeutigen Namen vornimmt. Durch die Möglichkeit einen Peer innerhalb einer Gruppe umzubenennen, kann beim Start des *System P* auf einem Rechner ein willkürlicher Name, zum Beispiel der Name des Rechners und ein Zeitstempel, verwendet werden. Dies stellt sicher, dass der Name des Peers im P2P-Netzwerk auch ohne eine zentrale Komponente eindeutig ist. Beim Zuweisen von lokalen Quellen, Schemata und Mappings auf die vorhandenen Peers kann deren Name dann an die Namen der Peer-Mappings angepasst werden.

3.2.4 Kommunikation mittels JXTA

Zur Kommunikation einzelner Peers untereinander wird die bereits erwähnte Bibliothek JXTA verwendet. Sie bietet eine Basis für P2P-Anwendungen. Es werden Methoden zum Aufbau eines P2P-Netzwerkes, zum Bereitstellen von Diensten und zur Kommunikation der Peers untereinander angeboten. Die Konzepte werden sehr gut auf der JXTA-Homepage[7] und auch von Gradecki [4] sowie Wilson [9] beschrieben.

Zwei Fähigkeiten von JXTA seien an dieser Stelle hervorgehoben. Zum einen können sich JXTA-Peers in einem lokalen Netz durch die Verwendung von Multicasts selbstständig finden, zum anderen bietet JXTA die Möglichkeit, dass zwei Peers selbst durch Firewalls und Proxy-Server miteinander kommunizieren können. Somit ist es leicht ein P2P-Netzwerk im lokalen Netz aufzubauen, und auch größere P2P-Netze über Netzwerkgrenzen hinweg sind mit Hilfe von JXTA denkbar.

Nach den ersten erfolgreichen Tests mit JXTA – einfache Peers haben sich im lokalen Netzwerk gefunden und konnten miteinander kommunizieren – stellte sich die Frage nach der Skalierbarkeit. Ziel ist es, ein möglichst großes PDMS mit mehreren hundert Peers auf einigen wenigen Rechnern zu erzeugen. Es stellte sich heraus, dass die maximale Anzahl der parallelen JVMs mit laufendem JXTA ungefähr bei 15 bis 20 lag[8]. Bei diesen Messungen sind noch nicht die Ressourcen einbezogen, die eine Anfragebearbeitung in einem Peer benötigt. Somit steht der Overhead, den eine JXTA-Instanz erzeugt, im Konflikt zu den Anforderungen an das *System P*.

Die Lösung des Problems sieht vor, die Anzahl der JXTA-Instanzen auf einem Rechner möglichst klein zu halten und trotzdem mehrere Peers auf diesem Rechner laufen zu lassen. Es ist daher nötig, dass in einer JXTA-Instanz mehrere Peers laufen. Somit entspricht ein „JXTA-Peer“ nicht mehr einem „*System P*-Peer“. Zur besseren Verständlichkeit, definieren wir die verwendeten Begriffe:

- Ein *System P-Peer* oder *PDMS-Peer* ist ein Peer im Peer Data Management System, wie in Kapitel 2.2 beschrieben. Wir nennen ihn im Folgenden *Peer* oder *PDMS-Peer*.

[7] http://www.jxta.org/white_papers.html

[8] Das verwendete System war ein AMD64 3500+ mit 1GB RAM

- Als *Monitor-Peer* werden die im Abschneitt 3.2.2 beschriebenen Instanzen bezeichnet, die zum Aufbau und zur Anfrageanalyse nötig sind.

- Ein *JXTA-Peer* ist ein Peer im JXTA Netzwerk. Um den Begriff „Peer“ nicht mehrfach zu verwenden und da ein JXTA-Peer immer einer eigenen virtual Machine-Instanz entspricht, nennen wir diese Art des Peers *JVM*.

Eine JVM kann somit einen oder mehrere Peers enthalten. Wichtig dabei ist, dass die Peers einer JVM keinen Zugriff auf die Daten der anderen Peers haben. Es darf keinen Unterschied machen, ob zwei Peers in einer JVM oder in zwei getrennten JVMs auf unterschiedlichen Rechnern laufen. Im Abschnitt 3.4.1 wird auf die Probleme und Lösungen mit mehreren Peers in einer JVM eingegangen.

3.3 Aufbau des P2P-Netzes

Der Aufbau eines P2P-Netzes besteht aus dem gegenseitigen Finden der JVMs. Im *System P* wird dazu das Discovery-Protokoll von JXTA genutzt. Vereinfacht dargestellt, bietet bei diesem Protokoll jede JVM Informationen über sich selbst an, nach denen andere JVMs gezielt suchen können.

3.3.1 StatusAdvertisement

Im *System P* liegen die angebotene Informationen einer JVM in einer XML-Nachricht, dem *StatusAdvertisement*. Das Beispiel 3.3 zeigt den Inhalt eines *StatusAdvertisements*.

Beispiel 3.3 *Beispiel für ein StatusAdvertisement*

```
<?xml version="1.0"?>
<!DOCTYPE StatusAdvertisement>
<StatusAdvertisement>
    <Type>StatusAdvertisement</Type>
    <JVMPeerId>urn:jxta:uuid-59616261646162614A78746150325033734 8C9C16FFC40F88AC3D69AAD51F34103</JVMPeerId>
    <TCPPort>20001</TCPPort>
    <JVMName>pc1134556091937</JVMName>
    <HostName>pc</HostName>
    <TimestampLastStatusChange>1134556101500</TimestampLastStatusChange>
    <Timestamp>1134557598234</Timestamp>
    <PeerNamesOfHostedPeersInGroup>|Peer1|</PeerNamesOfHostedPeersInGroup>
    <PipeAdvertisement>&lt;?xml version="1.0" encoding="UTF-8"?>
       &lt;!DOCTYPE jxta:PipeAdvertisement>
       &lt;jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
         &lt;Id>urn:jxta:uuid-29B8B05A503D4840B4BFA11D31062F69EB6DE95855934E93B8C68F957B8607D904&lt;/Id>
         &lt;Type>JxtaUnicast&lt;/Type>
         &lt;Name>Status Pipe&lt;/Name>
         &lt;Desc>Status Pipe for JVM: pc1134556091937&lt;/Desc>
       &lt;/jxta:PipeAdvertisement>
    </PipeAdvertisement>
</StatusAdvertisement>
```

- Der `Doctype`, sowie die beiden XML-Tags `StatusAdvertisement` und `Type` sind von JXTA vorgegeben. Die restlichen Tags sind spezifisch für das *System P*.

- Das Tag `JVMPeerId` enthält eine global eindeutige Bezeichnung (ID) für die JVM im JXTA-Netzwerk. Über diese ID kann sie identifiziert und angesprochen werden.

- Die Informationen in den Tags `TCPPort`, `JVMName` und `HostName` dienen nicht dem Aufbau des P2P-Netzes, sondern können einem Benutzer in der grafischen Benutzeroberfläche des Monitor Peers angeboten werden. Sie liefern Informationen auf welchem Rechner eine JVM läuft und welcher Ressource sie dort beansprucht.

- Im Tag `PeerNamesOfHostedPeersInGroup` ist eine Liste, der in der JVM laufenden Peers, zu finden. Diese Information kann von einem Peer genutzt werden, um gezielt nach Peers seiner Peer-Mappings zu suchen. Eine JVM *J1* nimmt somit nur Kontakt zu einer anderen JVM *J2* auf, wenn mindestens ein Peer von *J2* in einem Peer-Mapping von *J1* enthalten ist. Das heißt, *J1* benötigt nur dann Informationen von *J2*, wenn es die Möglichkeit gibt, dass *J1* bei einer Anfragebearbeitung *J1* Kontakt zu *J2* aufnehmen muss. Somit muss nicht jede JVM mit jeder anderen in Verbindung treten und es wird eine größere Skalierbarkeit des PDMS erreicht.

- Die beiden `Timestamp` Tags geben Auskunft über die Gültigkeit der angebotenen Informationen. Das JXTA-Framework verlangt, dass ein Advertisement eine begrenzte Lebensdauer hat. Der Grund dafür ist, dass die Advertisements von *Rendevouz-Peers* zwischengespeichert werden können. Hätten die Advertisements eine unendliche Lebensdauer, würden sie ewig in den Zwischenspeichern liegen, unabhängig davon, ob der Verfasser des Advertisements noch erreichbar ist. Andere JVMs würden daraufhin ungültige Informationen erhalten. Um dieses Risiko zu minimieren, müssen die Advertisements regelmäßig aktualisiert werden. In `Timestamp` wird der Zeitpunkt der Aktualisierung abgelegt. Sollte eine zweite JVM zwei Advertisements von einer JVM erhalten, kann sie entscheiden, welches der beiden *StatusAdvertisements* das aktuellere und zur Weiterverarbeitung von Nutzen ist. Der zweite Zeitstempel `TimestampLastStatusChange` zeigt die letzte Änderung des Status eines Peers in einer JVM an. Eine JVM, die das *StatusAdvertisement* erhält, kann anhand dieser Information entscheiden, ob sie die Absender-JVM über neue Statusinformation anfragt oder nicht. Eine Anfrage lohnt sich nur, wenn der Zeitstempel der letzten Statusänderung jünger ist, als der Zeitstempel bei der letzten Statusanfrage.

- Das letzte Tag `PipeAdvertisement` enthält die eigentliche Information, wie die JVM zu erreichen ist. Das Tag enthält ein JXTA *Pipeadvertisement* welches eine JXTA *Pipe* beschreibt. Pipes dienen in JXTA dazu, Informationen zwischen verschiedenen JVMs auszutauschen. Jeder Peer erzeugt vor dem Anbieten eines *StatusAdvertisements* eine Pipe und stellt die Informationen über die Pipe in das Advertisement. Anfragende Peers erhalten die Informationen über die Pipe und können sich anschließend mit dieser verbinden.

Die genaue Funktionsweise des Discovery-Protokolls und somit der Austausch von Advertisements wird sehr genau von Wilson [9] erläutert.

3.3.2 Status eines Peers

Nachdem geklärt ist, wie ein Peer Statusinformationen eines anderen Peers erhält, gehen wir der Frage nach, welche Informationen übertragen werden sollen. Für die eigentliche Anfragebearbeitung sind nur die Informationen wichtig, in welcher JVM ein Peer zu finden und ob dieser gerade „Online“, und somit aktiv ist. Aus dem StatusAdvertisement geht bereits hervor, in welcher JVM ein Peer zu erreichen ist. Die Information über die Erreichbarkeit des Peers fehlt dennoch. Weiterhin benötigt der im Abschnitt 3.2.2 beschriebene Monitor-Peer für die Darstellung in der grafischen Benutzeroberfläche zusätzliche Informationen.

Das Beispiel 3.4 zeigt eine XML-Nachricht, die die Statusinformationen von Peers einer JVM enthält. Diese XML-Nachricht wird über die oben erwähnte Pipe übertragen.

Beispiel 3.4 *PeerStatusContainer als XML-Nachricht*

```
<de.hu.mac.pdmsdipl.jxta.services.status.PeerStatusContainer>
  <peerStatusList>
    <de.hu.mac.pdmsdipl.jxta.services.status.PeerStatus>
      <peerGroupName>Default Group</peerGroupName>
      <peerNameInGroup>Peer1</peerNameInGroup>
      <peerName>Peer1</peerName>
      <status>
        <online>true</online>
        <isReady>true</isReady>
        <localSource>false</localSource>
        <monitor>false</monitor>
        <schema>false</schema>
        <localMappings>false</localMappings>
        <peerMappingsAsString/>
      </status>
    </de.hu.mac.pdmsdipl.jxta.services.status.PeerStatus>
  </peerStatusList>
</de.hu.mac.pdmsdipl.jxta.services.status.PeerStatusContainer>
```

Die Nachricht besteht aus einer Liste von *PeerStatus*-Informationen. Zu jedem Peer, der in der JVM läuft und der für den anfragenden Peer interessant ist, wird ein *PeerStatus* Eintrag erzeugt. Dieser enthält die folgende Informationen:

- `peerGroupName` enthält den Namen der Gruppe, in der sich der Peer gerade befindet. Welche Bedeutung die Gruppen haben, ist in Abschnitt 3.2.3 beschrieben.
- In `peerNameInGroup` ist der Name des Peers in seiner Gruppe abgelegt. Dieser Name entspricht den Namen in den Peer-Mappings.
- `peerName` ist der global eindeutige Name des Peers. Dieser Name bleibt bestehen, auch wenn der Peer seine Gruppenzugehörigkeit ändert.
- `online` zeigt an, ob dieser Peer in der JVM noch erreichbar ist oder nicht.
- Das Flag `isReady` zeigt an, ob ein Peer „bereit“ ist. „Bereit“ bedeutet, dass der Peer die Statusinformation über die Peers in seinen Peer-Mappings erhalten hat. Er kann somit Kontakt zu allen Peers aus seinen Peer-Mappings aufnehmen. Dies ist wichtig, damit die Anfragebearbeitung vollständig erfolgen kann.

- `schema` und `localMappings` sind gesetzt, wenn dieser Peer über ein Peerschema beziehungsweise über lokale Mappings verfügt.
- In `peerMappingsAsString` ist eine komplette Liste der Peer-Mappings des Peers abgelegt. Diese Liste ist nötig, damit zum Beispiel der Monitor-Peer in der Visualisierung des PDMS die Kanten zwischen den Peers setzen kann. Ein PDMS-Peer wertet diese Informationen nicht weiter aus.

Bis auf den Namen des Peers in seiner Gruppe und den Status, ob der Peer „bereit“ ist, sind die Status-Informationen nur für den Monitor-Peer und für die Darstellung in der grafischen Benutzeroberfläche nützlich und finden sich dort wieder (siehe Abbildung 3.3).

Statusanfrage in mehreren Schritten

In einer frühen Phase der Implementierung wurden sämtliche *PeerStatus* Informationen im StatusAdvertisement abgelegt. Der Nachteil dieser Methode ist, dass jede JVM die Statusinformationen über sämtliche in ihr laufenden Peers in das Advertisement legt und nicht gezielt deren Menge begrenzt, indem Statusinformationen nur auf Nachfrage gezielt übertragen werden. Es stellte sich heraus, dass ab einer größeren Anzahl von etwa fünfzig Peers in einer JVM der Aufbau der Statusadvertisements mit allen Statusinformationen zu lange dauerte und diese für das Discovery-Protokoll von JXTA zu groß wurden. Somit muss der oben beschriebene Ansatz verwendet werden. Anders als bei der Verwendung von Advertisements gibt es bei JXTA-Pipes keine Größenbeschränkungen beim Datenaustausch zwischen zwei Peers. Der komplette Ablauf der Übermittlung eines Peer-Status zwischen zwei JVMs ist in der Abbildung 3.4 festgehalten.

Zusammenfassend kann festgehalten werden, dass durch das verwendete Verfahren die Kommunikation zwischen den einzelnen JVMs auf ein Minimum reduziert wird. Eine JVM tritt nur in Kontakt mit einer anderen, wenn die Peers der anderen JVM in ihren Peer-Mappings vorkommen und somit bei einer Anfragebearbeitung erreicht werden müssen.

3.4 Peer Interface

Bis jetzt wurde beschrieben, wie sich Peers gegenseitig finden und Statusinformationen austauschen können. Ein Peer bietet darüber hinaus Methoden für andere Peers an. Sie umfassen das Verändern dieses Peers, wie das Setzen eines Peer-Schemas oder lokaler Quellen, sowie das Bearbeiten von Anfragen. Erst durch diese Methoden wird das P2P-Netzwerk zu einem PDMS.

Grundsätzlich sind alle Aufrufe von Methoden[9], die ein Peer für andere Peers im *System P* anbietet, asynchron. Das bedeutet, der Aufruf einer Methode blockiert nicht. Der Aufrufer muss daher ein „Callback“-Listener mitgeben, der bei Erfolg oder Misserfolg eines Aufrufes ausgelöst wird. In der Erfolgsmeldung ist dann ebenfalls ein eventueller Rückgabewert

[9] Außer den bereits beschriebenen Abfragen zu Statusinformationen.

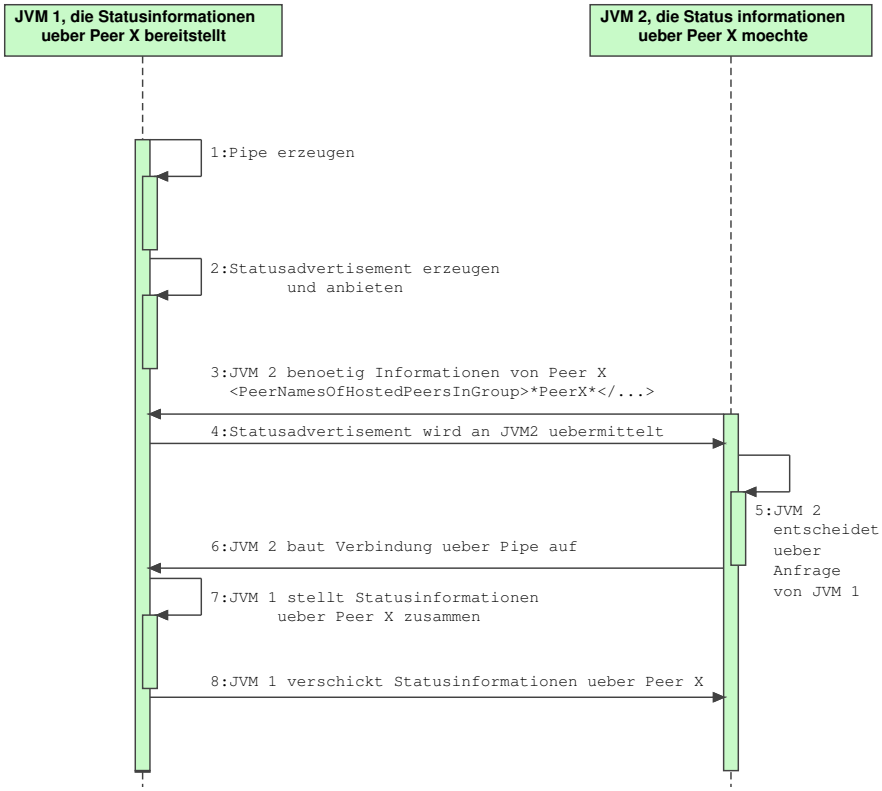

Abbildung 3.4: Statusübermittlung

der Methode enthalten. Ein Misserfolg kann entweder bedeuten, dass beim aufgerufenen Peer ein Fehler aufgetreten ist oder dieser Peer nicht mehr erreichbar ist. Für den letzten Fall kommt es dann zu einem vorher im Callback festgelegten Timeout. Es ist in jedem Fall garantiert, dass der aufrufende Peer eine Nachricht über den Ausgang seines Aufrufes erhält und nicht unendlich lange wartet.

Wie bereits erwähnt, gibt es neben dem PDMS-Peer noch den Monitor-Peer, der zum Erzeugen, Anpassen und Überwachen des PDMS und der Anfragebearbeitung zuständig ist. In Abbildung 3.5 sind die Hierarchie der Peer-Schnittstellen und die angebotenen Methoden dargestellt.

Die gemeinsam angebotenen Methoden von PDMS- und Monitor-Peer sind im Interface *Peer* zusammengefasst. Dies sind vor allem die Methoden, die keinen Callback benötigen, da sie aus den Statusinformation, die jede JVM liefert, beantwortet werden können. Diese gelieferten Informationen werden in der implementierenden Klasse zwischengespeichert.

Die beiden Methoden *splitPeer()* und *shutdown()* werden zum Auf- und Abbau eines PDMS verwendet. Mit Aufruf der Methode *splitPeer()* wird in der JVM, in der der aufgerufene Peer läuft, ein weiterer Peer erzeugt. Die PDMS-Instanz kann dadurch vergrößert werden. Der Aufruf von *shutdown()* setzt den Status der aufgerufenen Peers auf *offline*,

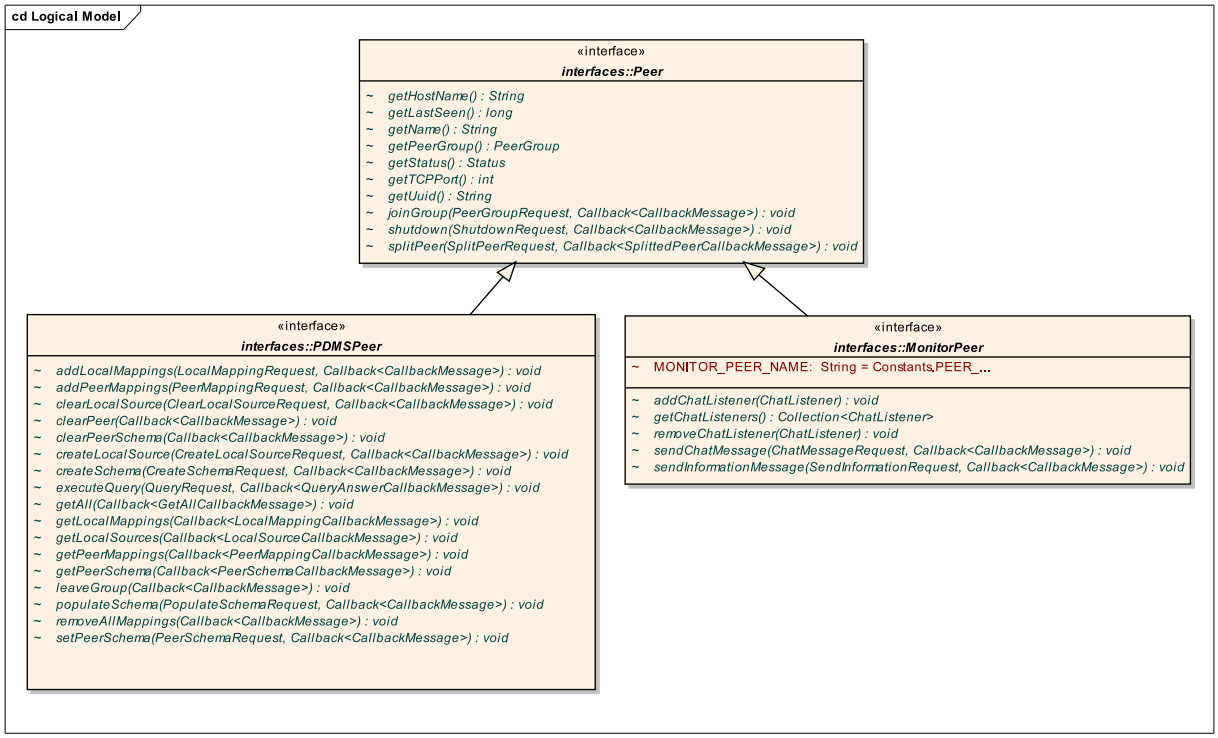

Abbildung 3.5: Peer-Interface

er ist somit nicht mehr für andere Peers zu erreichen.

Beim Interface für einen Monitor-Peer ist die Methode *sendInformationMessage()* hervorzuheben. Über diese Methode kann ein PDMS-Peer während der Anfragebearbeitung Rückmeldungen an einen Monitor-Peer geben, der diese dann parallel in einem Graphen darstellt (siehe Abschnitt 4.4). Die Rückmeldungen können auch zur Analyse der Anfragebearbeitung genutzt werden.

Ein PDMS-Peer hingegen bietet Methoden zum Setzen, Abfragen und Löschen seiner einzelnen Komponenten. So können zum Beispiel lokale Quellen angelegt (*createLocalSource()*), das Schema des Peers erfragt (*getPeerSchema()*) oder alle Mappings entfernt (*removeAllMappings()*) werden. Die wichtigste Methode ist jedoch *executeQuery()*, die die Anfragebearbeitung anstößt, die im nächsten Kapitel beschrieben wird.

Die Methode *joinGroup()* veranlasst das Wechseln eines Peers in eine neue Gruppe. Dabei kann dem Peer, wie im Abschnitt 3.2.3 beschrieben, innerhalb der Gruppe ein neuer Name zugewiesen werden.

3.4.1 Lokale und entfernte Peers

Mit der in Abbildung 3.5 definierten Schnittstellen ist festgelegt, welche Methoden ein PDMS-Peer implementieren muss. Dadurch, dass mehrere Peers in einer JVM liegen kön-

nen, ergeben sich zwei unterschiedliche Arten der Kommunikation zweier Peers. Die erste Art ist JVM-übergreifend. Die Peers liegen dabei in unterschiedlichen JVMs und zwischen Aufruf und Ausführung liegt eine Übertragung über ein Netzwerk. Im zweiten Fall befinden sich die Peers in der gleichen JVM und die Übertragung über das Netzwerk kann entfallen. Die Aufrufe sind dann Java-Methodenaufrufe.

In der Abbildung 3.6 sind die beiden nötigen Implementierungen dargestellt.

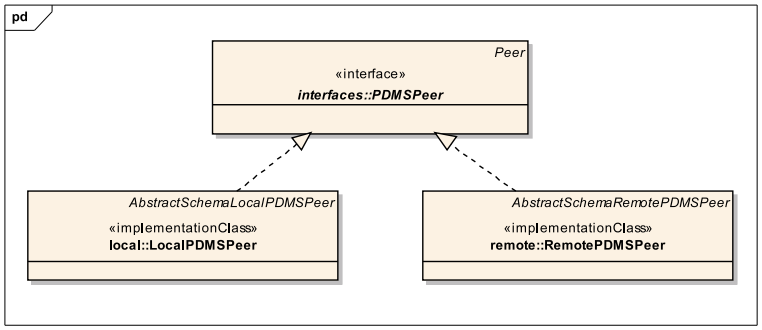

Abbildung 3.6: Implementierende Klassen der PDMS-Peer Schnittstelle

Die Klasse *LocalPDMSPeer* implementiert die Funktionen des lokalen JVM Aufrufs. Somit wird beim Aufruf der Methoden die tatsächliche Anwendungslogik, zum Beispiel eine Anfragebearbeitung, ausgeführt. Die Klasse *RemotePDMSPeer* enthält keine Anwendungslogik. Ein Aufruf ihrer Methoden führt zu einem entfernten Methodenaufruf (RPC[10]) in der JVM, in der der aufgerufene Peer liegt.

Durch die Verwendung des Interfaces *PDMSPeer* muss der aufrufende Peer nicht wissen, ob der aufgerufende Peer in derselben oder in einer entfernten JVM läuft. Die Schnittstelle ist in beiden Fällen dieselbe. Dies stellt auch sicher, dass der aufrufende Peer keinen Zugriff auf Funktionen eines Peers in derselben JVM erhält, den er bei einem entfernten Peer nicht hätte.

3.4.2 Entfernte Methodenaufrufe

Der Aufruf einer Methode eines Peers in einer entfernten JVM geschieht nach dem in Abbildung 3.7 dargestellten Muster. Die implementierende Klasse *RemotePDMSPeer* transformiert die Parameter der Methode in eine XML-Darstellung. Zusammen mit dem Namen der aufgerufenen Methode und dem Namen des aufgerufenen Peers wird diese XML-Nachricht an die JVM geschickt, die den aufgerufenen Peer enthält. Diese JVM wandelt die XML-Nachricht wieder in Java-Objekte, findet anhand des Peernamens die passende *LocalPDMSPeer* Instanz und führt die Methode anhand des übertragenen Namens aus. Ein eventueller Rückgabewert, Fehler oder das Gelingen des Aufrufes werden wieder in XML-Nachrichten serialisiert und zurück an die aufrufende JVM übertragen. Diese wandelt das XML-Dokument umgekehrt wieder in ein Java-Objekt oder eine Fehlermeldung

[10] remote procedure call

und übergibt dieses dem eigentlichen Aufrufer über die beim Aufruf registrierte Callback-Methode (siehe Abschnitt 3.4).

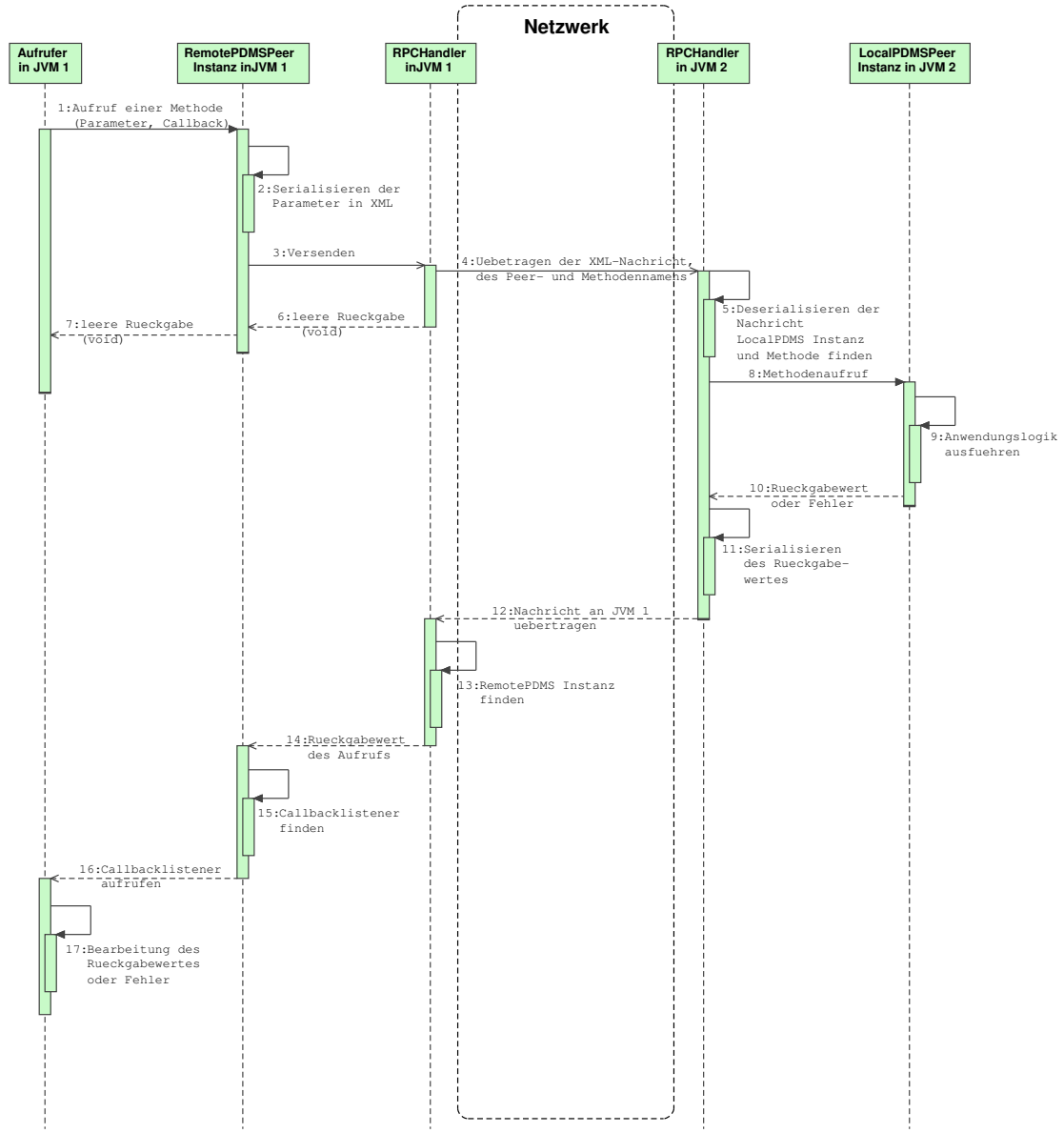

Abbildung 3.7: Entfernter Methodenaufruf

Zum Versenden der XML-Nachricht wird das „Peer-Resolver“ Protokoll von JXTA genutzt. Eine detaillierte Beschreibung dieses Protokolls findet sich in [9]. Bei diesem Protokoll stellt die fragende JVM ein *ResolverQuery*, das die antwortende JVM mit einem *ResolverResponse* beantwortet.

Das *ResolverQuery* ist von JXTA vorgegeben und hat folgendes Format:

```
<!DOCTYPE jxta:ResolverQuery>
   <jxta:ResolverQuery xmlns:jxta="http://jxta.org">
    <HandlerName>urn:jxta:uuid-49FB49CB1DC449318185DC71F0ED63CC05</HandlerName>
    <QueryID>0</QueryID>
    <HC>0</HC>
    <SrcPeerID>urn:jxta:uuid-59616261646162614A7874615032503385EA51B1B9AA4A4BA000E363E77105AE03</SrcPeerID>
    <SrcPeerRoute/>
    <Query>[...]</Query>
</jxta:ResolverQuery>
```

- Der *HandlerName* gibt die ID eines registrierten *Handlers* an. Ein Handler nimmt das empfangene Query an und bearbeitet es weiter. Im *System P* ist der Handler der „RPCHandler“, der die im *Query* übergebene XML Nachricht auswertet und auf einer *LocalPDMSPeer* Instanz ausführt.

- Jede Query benötigt eine eindeutige ID, die *QueryID*, die beim Versenden der Antwort mit angegeben wird. Mit Hilfe dieser kann der Empfänger einer Antwort den Aufrufer wiederfinden.

- *SrcPeerID* enthält die eindeutige JXTA-ID der fragenden JVM. An diese ID wird die Antwort des Aufrufs verschickt.

- *Query* enthält die eigentlich Anfrage. Im *System P* sind das die als XML serialisierten Objekte, der Name des Peers und der Name der aufzurufenden Methode.

Empfängt der *RPCHandler* im *System P* eine solche *ResolverQuery*, speichert er die *QueryID* und die *SrcPeerID* und deserialisiert die im *Query*-Tag abgelegte XML-Nachricht. Anschließend sucht er anhand des Peer-Namens die passende *LocalPDMSPeer*-Instanz und ruft auf dieser die Methode mit dem erhaltenen Methodennamen und den erhaltenen Parametern auf. Der Rückgabewert des Methodenaufrufs wird wieder in eine XML-Nachricht serialisiert und eine *ResolverResponse* aufgebaut, die an die gespeicherte *SrcPeerID* geschickt wird.

Die Struktur der *ResolverResponse* ist durch JXTA festgelegt:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:ResolverResponse>
  <jxta:ResolverResponse xmlns:jxta="http://jxta.org">
    <HandlerName>urn:jxta:uuid-49FB49CB1DC449318185DC71F0ED63CC05</HandlerName>
    <QueryID>0</QueryID>
    <Response>[...]</Response>
</jxta:ResolverResponse>\normalsize
```

- In *HandlerName* wird wieder die eindeutige ID des RPC-Handlers abgelegt.

- *QueryID* entspricht der ID aus der *ResolverQuery*

- *Response* enthält die XML-Nachricht mit dem serialisierten Rückgabewert des Methodenaufrufes.

Erhält die aufrufende JVM die Antwort in Form einer *ResolverResponse* kann sie den Rückgabewert aus dem übertragenen XML-Dokument deserialisieren und anhand der

QueryID den eigentlichen Aufrufer und dessen Callback-Listener finden und aufrufen. Der Aufruf einer Methode auf einem entfernten Peer ist damit abgeschlossen.

Ergänzend anzumerken ist, dass der Callback-Listener einen internen Timer besitzt, der nach einer einstellbaren Zeit eine *timeout()*-Methode aufruft. Damit ist sichergestellt, dass der Aufrufer auch im Falle eines Netzwerkausfalls oder sonstigen unerwarteten Fehlern eine Rückmeldung erhält.

Zusammengefasst hat die gewählte Methode den Vorteil, dass es für den Aufrufer keinen Unterschied macht, ob er einen lokalen Peer anspricht oder eine Kommunikation über ein Netzwerk zwischen dem Aufruf und der Ausführung der Methode stattfindet. Für die Implementierung bedeutet dies, dass sich neue Methoden leicht beim *RPCHandler* registrieren und sich somit die Funktionen eines PDMS-Peers einfach erweitern lassen.

3.5 Lokale Quellen

Neben der Komponente zum Kommunizieren bilden die lokalen Quellen einen wichtigen Bestandteil eines Peers. In ihnen sind die Daten abgelegt, die bei einer Anfrage als Ergebnis geliefert werden sollen. Die lokalen Quellen sind für einen Peer optional. Hat er keine lokalen Quellen, aber Peer-Mappings, nimmt er nur die Rolle eines Vermittlers (Mediator) ein.

Im *System P* können als lokale Quellen JDBC-Verbindungen[11] dienen. Genaueres dazu ist im Abschnitt 3.5.2 beschrieben. Als Datenmodell wird in den lokalen Quellen das relationale Modell verwendet. Da Anfragen innerhalb des *System P* in Datalog formuliert werden, muss es an mindestens einer Stelle eine Konvertierung von Datalog nach SQL geben. Im Abschnitt 3.5.4 werden diese Konvertierung und die damit verbundenen Probleme genauer erläutert. Zunächst sollen jedoch die verwendeten Datentypen betrachtet werden.

3.5.1 Relationales Modell in Java-Klassen

Grundsätzlich soll ein PDMS-Peer vom Monitor Peer mit einem Peer-Schema und mit Daten für seine lokalen Quellen versorgt werden können. Ebenso müssen Tupel aus dem Anfrageergebnis und auch Tupel von Zwischenergebnissen zwischen Peers übertragen werden. Dazu ist es nötig, Daten aus den lokalen Quellen auszulesen und in geeignete Datenstrukturen zu verpacken. Geeignet heißt dabei, dass diese Datenstrukturen den Transport über ein Netzwerk und Operationen auf den Daten ermöglichen.

Um die Funktionen der lokalen Quellen, die Implementierung der Join- und Union-Operatoren (Abschnitt 3.5.3), sowie die Datalog zu SQL Konvertierung (Abschnitt 3.5.4) genauer beschreiben zu können, folgt ein kurzer Überblick über die verwendeten Java-Klassen

[11] Java Database Connectivity

und ihre Beziehung zueinander. In Abbildung 3.8 sind diese Klassen und Beziehungen dargestellt. Das Suffix *DTO*[12] in den Klassennamen deutet an, dass diese Klassen als „Data TransferObject“ über ein Netzwerk übertragen werden können.

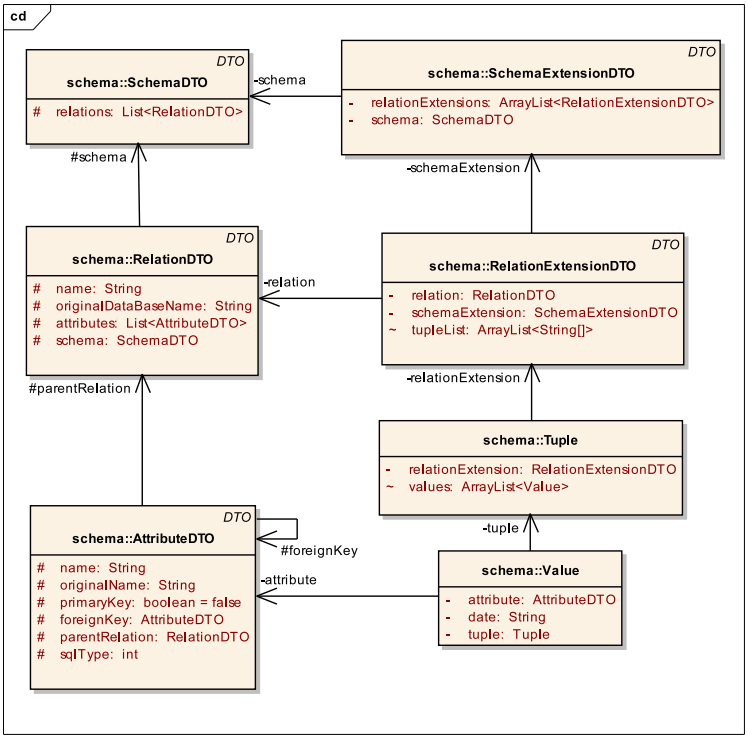

Abbildung 3.8: Klassendiagramm des relationalen Modells

Wie im relationalen Modell gibt es Relationen, die aus einem eindeutigen Namen und einer Menge von Attributen bestehen. In der Implementierung von *RelationDTO* werden die Attribute allerdings in einer Liste gespeichert, durch die sie in eine Reihenfolge gebracht sind. Dies ist für die spätere Konvertierung der Datalog-Anfrage in eine SQL-Anfrage von Bedeutung (siehe Abschnitt 3.5.4).

Die Attribute, die Instanzen von *AttributeDTO*, bestehen im Wesentlichen aus einem Namen und einem SQL-Typen. Die unterstützten Typen sind *String*, *Integer* und *Decimal*. Zusätzlich zeigt ein Flag *primaryKey* an, ob es sich bei dem Attribut um einen Primärschlüssel handelt. Das Attribut *foreignKey* zeigt im Falle, dass es sich bei der Instanz um einen Fremdschlüssel handelt, auf den referenzierten Primärschlüssel.

Mehrere Relationen können zu einer Instanz der Klasse *SchemaDTO* zusammengefasst und einem Peer als Peer-Schema zugewiesen werden.

[12] http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html

Für die Extension einer Relation dient die Klasse *RelationExtensionDTO*. Sie enthält neben der Referenz auf die Relation ein Array von Strings, das die String-Repräsentation der Daten der Relation enthält. Mehrere *RelationExtensionDTO* können zu einem *SchemExtensionDTO* zusammengefasst werden. Eine Instanz von *SchemExtensionDTO* entspricht zum Beispiel dem Inhalt einer lokalen Quelle.

Die Klassen *Tuple* und *Value* repräsentieren ein Tupel in einer Relation, beziehungsweise den Wert eines Attributes in einem Tupel. Die Instanzen von *Tupel* und *Value* werden aber nicht als *DTO* übertragen. Sie werden aus dem Array von Strings von *RelationExtensionDTO* bei Bedarf instanziiert. Dies bietet den komfortablen Zugriff auf einzelne Tupel und Werte über Java-Klassen. Gleichzeitig wird der Overhead beim Serialisieren der Daten und somit bei einer Übertragung übers Netzwerk verringert, da nur das String-Array und keine komplexen Java-Objekte übertragen werden müssen.

3.5.2 Anbindung lokaler Quellen

Als Datenquellen des *System P* können alle relationalen Datanbankmanagement-Systeme genutzt werden, die über einen JDBC-Treiber ansprechbar sind. Es wurde die Anbindung von HSQLDB (siehe Abschnitt 3.1) und PostgreSQL[13] verwendet und getestet.

Lokale Quellen werden über das in Abbildung 3.9 dargestellte Interface und nicht direkt über ihren JDBC-Treiber angesprochen. Unterschiede im SQL-Dialekt und der JDBC-Treiber verschiedener DBMS werden über dieses Interface verborgen. Die Verwendung dieses Interfaces ist keine direkte Anforderung an das *System P*. Sollte sich jedoch in Zukunft HSQLDB als zu langsam oder zu speicherintensiv herausstellen, oder eine bestehende Datenbank in einem anderen DBMS als lokale Quelle genutzt werden, so kann HSQLDB gegen dieses DBMS ausgetauscht werden.

Die wichtigsten Methoden des Interfaces sind dabei vor allem die Methoden *connect()* und *close()* zum Auf- und Abbau einer Datenbankverbindung, sowie die Methoden zum Setzen und Auslesen eines Schemas und einer Extension. Mit diesen Methoden kann ein Peer direkt Daten und Metadaten einer lokalen Quelle auslesen oder setzen. Eine wichtige Rolle spielt die Methode *selectProjectJoin()*. Sie ermöglicht es dem Peer, Anfragen an die lokale Quelle zu stellen. Dabei werden die angefragten Relationen als Instanz der Klasse *RelationDTO* übergeben. Ergebnis der Anfrage ist eine Instanz der Klasse *RelationExtensionDTO*. Somit kann der Peer allen Methoden einer Quelle nutzen, ohne deren spezielle Eigenheiten zu kennen.

Im *System P* gibt es, wie oben erwähnt, eine Implementierung des *JDBCLocalSource*-interfaces für HSLQDB. Soll ein weiteres RDBMS, wie zum Beispiel DB2, angebunden werden, so müssen nicht alle Funktionen erneut implementiert werden. Die Klasse *GenericJDBCLocalSource* fasst gemeinsam verwendete Methoden zusammen. Beim Erben

[13] http://www.postgresql.org/

Abbildung 3.9: JDBCLocalSource-Interface

von dieser Klasse müssen dann nur noch Methoden zum Auf- und Abbau einer Datenbankverbindung, sowie ein Mapping der vom *System P* unterstützten Datentypen (String, Integer, Double) auf die quellenspezifischen Datentypen angegeben werden.

3.5.3 Implementierung von Join und Union

In der später beschriebenen Anfragebearbeitung (siehe Abschnitt 4) ist es nötig, dass ein PDMS-Peer auf Daten aus verschiedenen Quellen die Operationen „Join“ und „Union“ ausführt. Quellen können dabei mehrere lokale Quellen oder das Ergebnis von Anfragen an andere Peers sein. Die Daten liegen in mehreren Instanzen von *RelationExtensionDTO* vor. Um keine komplizierte Analyse für die am besten geeignete Join-Strategie oder eine aufwendige Duplikaterkennung implementieren zu müssen, werden die Extensionen in eine HSQLDB-Datenbank geschrieben und die Operationen auf den Daten als SQL-Befehl ausgeführt. Die Ergebnis-Relation wird anschließend wieder in eine Instanz von *RelationExtensionDTO* geschrieben und kann als Zwischenergebnis oder als Anfrageergebnis weiterverarbeitet werden. Die Klasse *ExtensionUtils* enthält die Methoden *selectProjectJoin()* und *union()*, die die beschriebene Vorgehensweise implementieren. Durch die Verwendung des *JDBCLocalSource* Interface kann das in *ExtensionUtils* genutzte HSQLDB, sollte seine Leistung unzureichend sein, leicht gegen ein anderes DBMS ausgetauscht werden.

3.5.4 Konvertierung von Datalog nach SQL

Wie im Abschnitt 2.4 beschrieben, werden Anfragen an das *System P* in Datalog formuliert. Durch die Verwendung von relationalen Datenbanksystemen als lokale Quellen, werden Anfragen an lokale Quellen als SQL-Befehle ausgedrückt. Somit muss eine Konvertierung von Datalog nach SQL stattfinden.

Im Unterschied zum relationalen Modell kennt Datalog keine Namen von Attributen. Dafür haben Attribute eine feste Position innerhalb einer Relation.

Beispiel 3.5 *Es seien folgende Relationen gegeben*

```
Verlag(ID, Name)
Autor(ID, Name, Vorname)
Buch(ISBN, Name, Autor_ID, Verlag_ID)
```

Die Anfrage soll lauten:

„Gib mir von allen Büchern mit dem Titel 'Berlin' die ISBN, den Vornamen des Autors und den Namen des Verlages!“

In Datalog sieht die Anfrage folgendermaßen aus:

```
q(a,b,c):-buch(a,y1,x1,x2), autor(x1,y2,b), verlag(x2,c), y1 = 'Berlin'
```

Die gleiche Anfrage hat in SQL folgendes Format:

```
SELECT buch.ISBN, autor.vornamen, verlag.name
FROM buch, autor, verlag
WHERE buch.autor_id = autor.ID
AND buch.verlag_id = verlag.ID
AND buch.titel = 'Berlin'
```

Da die Namen der Attribute in Datalog keinen Einfluss auf die Anfrage haben, kommt es allein auf die Reihenfolge an. Joins werden bei Datalog, indem die Join-Variable (gleiche Bezeichnung) in mehreren Teilzielen verwendet wird. Dabei ist es wichtig zu wissen, an welcher Stelle die Attribute in den, am Join beteiligten, Relationen stehen.

Die Variable *y2* taucht im Anfrageergebnis nicht auf und ist an keiner Selektion beteiligt. Es muss jedoch bei dem im *System P* verwendeten Datalog explizit nach ihr gefragt werden. Eine angefragte Quelle würde somit auch Werte zu diesem Attribut liefern oder für den Fall, dass diese Quelle ein PDMS-Peer ist, weitere Quellen fragen. Alle Anfragen im Anfrageplan, die sich nur auf das *y2* entsprechende Attribut beziehen, könnten jedoch vermieden werden, da *y2* das Ergebnis der Anfrage nicht beeinflusst.

Da es im *System P* kein Mapping zwischen Attributnamen und der Position des Attributes in den Relationen gibt, muss die Reihenfolge der Attribute immer erhalten bleiben. Es ist zum einen wichtig, dass die Implementierung von *RelationDTO* eine Liste und keine Menge für die Attribute verwendet. Zum anderen müssen die angebundenen lokalen Quellen über die Schnittstelle *JDBCLocalSource* sicherstellen, dass die Reihenfolge der

Attribute bei jeder Anfrage gewahrt bleibt.

Der folgende Auflistung beschreibt das Umwandeln einfacher Datalog-Anfragen in eine SQL-Anfrage:

- In der *FROM-Clause* der SQL-Anfrage werden alle Relationen aufgenommen, die auch in der Datalog Anfrage enthalten sind.
- In der *SELECT-Clause* werden die Variablen der linken Seite der Datalog-Anfrage durch die Attribute der Relationen ersetzt. Dabei wird eine Relation ausgewählt, in der die Variable auftaucht und anhand der Position der Variablen in der Relation auf das Attribut geschlossen.
- Die *WHERE-Clause* besteht zum einen aus den Joins, bei denen die gleiche Variable in der Datalog-Anfrage in mehreren Relationen vorkommt. Dabei werden die Attribute hinter dieser Variable von je zwei Relationen durch den '='-Operator verknüpft.

  Den zweiten Teil der *WHERE-Clause* bilden die Selektionen. Dabei werden die Variablen in der Datalog-Notation durch ihre entsprechenden Attribute ersetzt. Entspricht eine Variable mehreren Attributen, ist es unerheblich, welches Attribute genommen wird, da über diesen Attributen zusätzlich ein (Equi-)Join ausgeführt wird.

3.5.5 Peer-Schema und Mappings

Um die Beschreibung der Module eines Peers, wie in Abbildung 3.2 des Abschnitts 3.2.1 dargestellt, zu vervollständigen, fehlt die Beschreibung zum Peer-Schema und den Mappings. Das Peer-Schema ist eine Instanz der Klasse *SchemaDTO*, wie sie in Abschnitt 3.5.1 beschrieben wird. Die Mappings, sowohl lokale als auch Peer-Mappings, werden als einfache Liste von *Strings* gespeichert. Es ist somit grundsätzlich möglich, dass ein Peer auch ohne lokale Quellen die Rolle eines Mediator-Peers einnehmen kann.

3.6 Anbindung des PDMS-Simulators

Wie bereits in der Einleitung beschrieben, wurde für die Anfragebearbeitung im *System P* auf den Code eines von Armin Roth entwickelten Simulators zurückgegriffen. An dieser Stelle soll kurz auf die dabei aufgetretenen Probleme und Lösungen eingegangen werden.

Die Hauptfunktion des Simulators ist die Anfragesimulation eines gegebenen PDMS. Dazu nutzt er die Informationen aller an einem PDMS beteiligten Peers, um aus einer Anfrage einen kompletten, so genannten Rule-Goal-Tree(RG-Tree)[14] (siehe Abschnitt 4.1) aufzubauen. Der Quellcode konnte zum Aufbau des lokalen RG-Trees eines Peers im *System P* komplett wiederverwendet werden. Dabei waren Anpassungen nötig. Einerseits musste der

[14] Ein RG-Tree entspricht im wesentlichen einem Anfrageplan

recht „monolithische“ Code in kleinere Module zerteilt werden. Die im Abschnitt 5.6 beschriebenen Ansätze zur Anfragebeaerbeitung wurden in einzelne Klassen verschoben, die von einem gemeinsamen Interface (*Approach*) erben. Somit wird das einfache Verändern und Hinzufügen neuer Anfragestrategien möglich. Andererseits verwendet der bestehende Simulator andere Datentypen zum Darstellen von lokalen Quellen, Mappings und Schemata. Diese Datentypen waren jedoch nicht für die Verwendung in einem System ausgelegt, das Anfragen nach dem Aufbau der RG-Trees auf Quellen mit tatsächlichen Daten ausführt und diese über ein Netzwerk überträgt. Es muss somit eine Konvertierung zwischen den Datentypen stattfinden, welche in der Klasse *ConverterUtil* implementiert ist.

Für den Code des Simulators gab es nur einige Testfälle in Form von manuell geschriebenen PDMS-Instanzen. Durch die Verwendung des Codes im *System P* und durch die in den Testfällen gezielte und in den Experimenten häufige und unterschiedliche Verwendung traten einige Fehler des Simulators zu Tage, die aber von Armin Roth schnell behoben werden konnten.

Die Wiederverwendung des Codes ermöglicht es, dass die Funktionen des Simulators erhalten bleiben und gleichzeitig derselbe Code im *System P* zum Einsatz kommt. Fehlerbeseitigungen und das Hinzufügen neuer Funktionen, wie zum Beispiel neuer Anfragebearbeitungsstrategien wirkt sich damit parallel auf beide Systeme aus.

3.7 Testfälle

Während der Entwicklung des *System P* wurde darauf geachtet, dass möglichst jede verwendete Komponente und die implementierten Funktionalitäten getestet werden. Das Schreiben der Testfälle bedeutet Aufwand, der aber durch das frühzeitige Finden von Fehlern auszahlt. Weiterhin lassen sich Fehler im Quellcode durch das Schreiben von Testfällen sehr leicht lokalisieren. [2] bietet einen guten Einstieg in das „Unit“-Testen mit *JUnit*.

In der Tabelle 3.7 sind die wichtigsten Testfälle des *System P* aufgelistet. Als Daten für die Testfälle werden einfache Mappings und relationale Datenbankschemata mit wenigen Tupeln verwendet.

Klasse	Name des Testfalls	Beschreibung
PDMSPeerTestCase	testPDMSPeerComplete()	Testet die Funktionen eines *LocalPDMSPeer*s, wie das Setzen der Mappings, des Peer-Schemas und lokalen Quellen und das Abfragen der Statusinformationen eines Peers.
ConverterUtilTestCase	testLocalMappingConversion()	Testet das Konvertieren der DTO Java Klassen zum Darstellen eines lokalen Mappings in die vom Simulator verwendete Struktur.
ConverterUtilTestCase	testPeerMappingConversion()	Testet das Konvertieren der DTO Java Klassen zum Darstellen eines Peer-Mappings in die vom Simulator verwendete Struktur.
DTOTestCase	testToStringAndDecodeString()	Testet das Serialisieren und Deserialisieren von Java-Objekten mit XStream.
ExtensionTestCase	testExtensionSerialisationDeserialisation()	Testet das Serialisieren und Deserialisieren einer *SchemaExtensionDTO* Instanz mit XStream.
ExtensionJDBCTestCase	testJDBC2Extension2JDBC2Extension()	Testet das Schreiben das anschließende Auslesen einer *SchemaExtensionDTO* Instanz in eine HSQLDB.
ExtensionUtilsTestCase	testEquiJoin(), testUnionSameRelation(), testUnionDifferentRelations(), testSelection()	Testet die Operationen *join()* und *union()* der Klasse *ExtensionUtils*.
HSQLDBTestCase	testHSQLDBSource()	Testet die Anbindung von *HSQLDB*. Es werden über eine aufgebaute HSQLDB-Verbindung SQL Insert- und Select-Anweisungen ausgeführt.
QueryProcessingTestCase	diverse	Testet die Anfragebearbeitung eines *LocalPDMSPeer*s. Die Testfälle werden im Abschnitt 4.3 beschrieben.

Tabelle 3.2: Testfälle der Implementierung im *System P*

4 Anfragebearbeitung

Die wichtigste Komponente eines Peers in einem PDMS ist die Anfragebearbeitung. Aufgabe dieser Komponente ist es, eine Anfrage unter Verwendung der lokalen und Peer-Mappings in einen Anfrageplan zu übersetzen. Der Anfrageplan eines Peers enthält die Anweisungen, welche Teil-Anfragen an lokale Quellen oder weitere Peers zu stellen sind und mit welchen Operatoren die Ergebnisse dieser Teil-Anfragen zum Gesamtergebnis zusammengeführt werden.

Die Anfragebearbeitung gliedert sich in die folgenden Schritte:

1. Parsen der Anfrage
2. Auswahl der nutzbaren[1] Mappings
3. Auswahl der zu benutzenden Mappings (siehe Kapitel 5)
4. Aufbau des Anfrageplanes
5. Optimierung des Anfrageplanes
6. Stellen der Anfragen des Anfrageplanes an lokale Quellen und weitere Peers
7. Zusammenführung der Ergebnisse der Anfragen
8. Rückgabe des Ergebnisses

Im *System P* werden die Punkte eins bis drei und Teile von Punkt vier durch den Anfrageplaner des bereits erwähnten PDMS-Simulators übernommen. Dieser zerlegt zunächst die Anfrage in ihre Teilziele. Mit dem Ersetzen der Teilziele durch nutzbare Mappings wird der *Rule-Goal Tree*(RG-Tree) der Anfrage aufgebaut. Dieser wird anschließend in den eigentlichen Anfrageplan übersetzt, optimiert und ausgeführt.

In diesem Kapitel wird zunächst der RG-Tree und sein Aufbau beschrieben. Anschließend werden die Transformation des RG-Trees in einen Anfrageplan und mögliche Optimierungen erklärt. Zum Abschluss wird auf die Visualisierung und mögliche Verbesserungen der Anfragebearbeitung im *System P* eingegangen.

[1] Die Bestimmung, ob ein Mapping nutzbar ist kann in Abschnitt 4.2 von [5] nachgelesen werden. Der Anfrageplaner des *System P* nutzt Global-as-View und Local-as-View-Anfrageumformulierungen.

4.1 Rule-Goal Tree

Die genaue Definition und der Algorithmus zum Erzeugen eines RG-Trees befindet sich im Abschnitt vier von [5]. An dieser Stelle soll nur ein kurzer Überblick gegeben werden, der zum Weiteren Verständnis der Anfragebearbeitung im *System P* nötig ist.

Ein Rule-Goal Tree ist ein Baum, dessen Wurzel-Knoten aus einem *Goal-Node* besteht. Ein Goal-Node steht für ein Teilziel, das erreicht werden soll. Kinder eines Goal-Nodes, sind *Rule-Nodes*. Sie bestehen aus Mappings, die für das Erreichen der Ziele der Goal-Nodes nutzbar sind. Kinder eines Rule-Nodes sind ausschließlich Goal-Nodes, deren Ziel jeweils ein Teilziel aus dem Kopf des Mappings des Rule-Nodes darstellt. Ein Pfad von der Wurzel eines RG-Trees bis zu den Blättern besteht somit immer abwechselnd aus Goal- und Rule-Nodes. Die Blätter des Rule-Goal Trees sind ausschließlich Goal-Nodes. Dabei symbolisiert ein Blatt eine Relation einer lokalen Quelle[2].

Der Inhalt der Wurzel eines RG-Trees, der erste Goal-Node und das einzige Kind der Wurzel, der erste Rule-Node, sind vorgegeben. In diesem Rule-Node ist die Anfrage selbst abgelegt, während der Wurzel-Goal-Node die Zielrelation dieser Anfrage angibt.
Das folgende Beispiel veranschaulicht den Aufbau eines RG-Trees.

Beispiel 4.1 *Beispiel eines RG-Trees zur Anfrage:*

Peer001.q(a, b, c, d, e, f) :- Peer001.R1(a, b, c, d, e), Peer001.R3(a, f)

```
[] Peer001.q(a, b, c, d, e, f)
  () Peer001.q(a, b, c, d, e, f) :- Peer001.R1(a, b, c, d, e), Peer001.R3(a, f)
    [] Peer001.R1(a, b, c, d, e)
      () L1.R1(c, b, a), L1.R2(c, d, e) :- Peer001.R1(a, b, c, d, e)
        [] L1.R1(c, b, a)
        [] L1.R2(c, d, e)
    [] Peer001.R3(a, f)
      () L1.R3(a, b) :- Peer001.R3(a, b)
         [] L1.R3(a, f)
```

`[]` = *Goal-Node*
`()` = *Rule-Node*

Die Wurzel und somit das Ziel des RG-Trees, ist die Menge aller Tupel der Relation q einer Anfrage an Peer001. Die Regel, die zur Beantwortung dieses Ziels führt, ist im ersten Rule-Node zu sehen. Es ist die Anfrage, die an Peer001 gestellt wurde.

Da der Rumpf der Anfrage aus den beiden Teilzielen „Peer001.R1(a, b, c, d, e)“ und „Peer001.R3(a, f)“ besteht, wird der ersten Rule-Node nur zu zwei Goal-Nodes mit diesen Teilzielen expandiert. Zur Beantwortung des ersten Teilziels wird nur das lokale Mapping „L1.R1(c, b, a), L1.R2(c, d, e) :- Peer001.R1(a, b, c, d, e)“ genutzt. Der Goal-Node wird

[2] Als Ausnahme kann es auch ein Teilziel sein, für das es kein nutzbares Mapping gibt. In diesem Fall kann das Blatt und der darüber liegende Rule-Node jedoch gelöscht werden.

zu genau einem Rule-Node expandiert. Da im Kopf des Mappings zwei Teilziele „L1.R1(c, b, a)“ und „L1.R2(c, d, e)“ vorkommen, besitzt dieser Rule-Node ebenfalls zwei Goal-Nodes als Kindknoten. Das verwendete Mapping ist ein lokales Mapping. Ziel der Goal-Nodes sind die Relationen „R1“ und „R2“ aus der lokalen Quelle „L1“. Diese Goal-Nodes werden nicht weiter expandiert.

Zur Beantwortung des zweiten Teilzieles „Peer001.R3(a, f)“ wird ein lokales Mapping „L1.R3(a, b) :- Peer001.R3(a, b)“ verwendet. Der Goal-Node „`[] Peer001.R3(a, f)`*“ wird daher nur zu einem Rule-Node expandiert. Das genutzte Mapping hat im Kopf ein Teilziel, so dass der Rule-Node ebenfalls nur einen Goal-Node „L1.R3(a, f)“ als Kind besitzt. Ziel dieses Goal-Nodes ist die Relation „R3“ in der lokalen Quelle „L1“, er wird ebenfalls nicht weiter expandiert.*

4.1.1 Aufbau des RG-Tree im *System P*

Der im Eingang erwähnte Anfrageplaner des Simulators verfügt über *globales Wissen*. Er kennt alle Peers, deren Schemata, Mappings und lokale Quellen. Im Simulator ist somit der Aufbau des kompletten RG-Trees für eine Anfrage an eine PDMS-Instanz möglich. Auch lässt sich die im späteren Abschnitt 5.2 beschriebene Erkennung für Kreise (aus Peer-Mappings) bei der Anfragebearbeitung leicht umsetzen.

Im *System P* hingegen verfügt keine Komponente einer PDMS-Instanz über globales Wissen. Es kann somit kein globaler RG-Tree berechnet werden. Jeder Peer erzeugt aus dem Wissen, das er in Form von lokalen und Peer-Mappings besitzt, einen lokalen RG-Tree. Dieser RG-Tree hat in Abhängigkeit der nutzbaren Mappings höchstens die Tiefe fünf. Der Wurzelknoten bildet die Zielrelation der gestellten Anfrage. Dieser Knoten besitzt als einziges Kind den Rule-Node mit der Anfrage. Die Kinder dieses Rule-Nodes sind die Goal-Nodes mit den Teilzielen der Anfrage. Diese Goal-Nodes nennen wir im Folgenden *Goal-Node der dritten Ebene*[3]. Jeder dieser Goal-Nodes kann mehrere Rule-Nodes haben. Die Anzahl der Rule-Nodes entspricht der Anzahl der nutzbaren lokalen und Peer-Mappings für das Ziel des Goal-Nodes. Diese Rule-Nodes werden im weiteren Verlauf als *Rule-Node der vierten Ebene* bezeichnet. Das Mapping eines dieser Rule-Nodes kann im Kopf wiederum mehrere Teilziele haben. Jedes dieser Teilziele wird zu einem Kind des Rule-Nodes in Form eines Goal-Node. Diese Goal-Nodes werden als *Goal-Node der fünften Ebene* bezeichnet werden. Die Blätter des lokalen RG-Trees können neben Relationen einer lokalen Quelle auch Relationen eines Peer-Schemas enthalten, sie beschreiben Anfragen, die ein Peer anderen PDMS-Peers stellen muss. Zur Verdeutlichung zeigt die Abbildung 4.1 zeigt die Bezeichnung der Goal- und Rule-Nodes aus dem Beispiel 4.1.

Der RG-Tree aus dem Beispiel 4.1 ist ein lokaler RG-Tree. Die Anfrage hat im Rumpf nur zwei Teilziele. Somit besitzt dieser RG-Tree auch zwei Goal-Nodes der dritten Ebene. Zur Expansion dieser Goal-Nodes stehen jeweils nur ein nutzbares (lokales) Mapping zur Verfügung, so dass beide nur einen Rule-Node der vierten Ebene als Kind besitzen.

[3] Gemeint ist die Ebene, die die Knoten im lokalen RG-Tree haben.

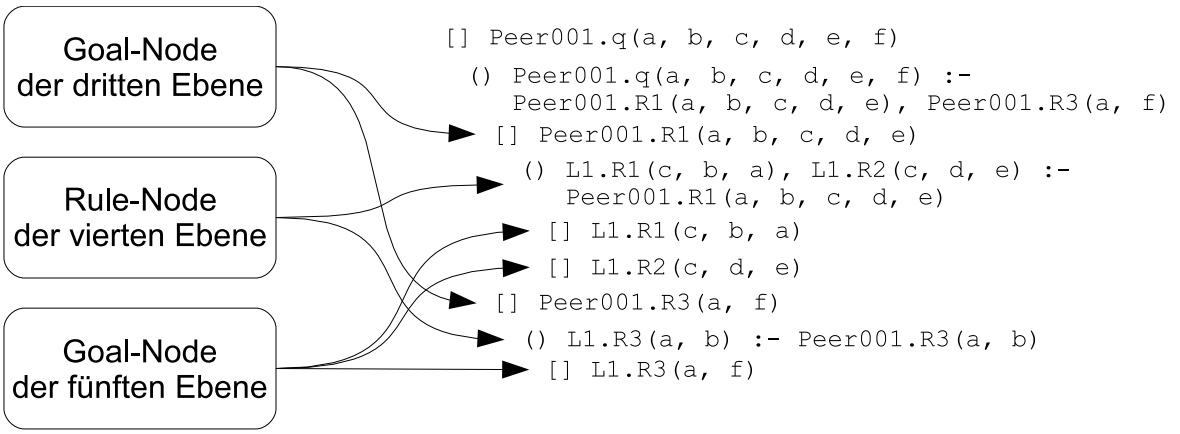

Abbildung 4.1: Ebenen im RG-Tree

Das verwendete Mapping des ersten Goal-Nodes hat im Kopf zwei Teilziele, so dass der zugehörige Rule-Node über zwei Goal-Nodes der fünften Ebene verfügt. Das genutzte Mapping des zweiten Teilziels der Anfrage besitzt nur ein Teilziel im Kopf, so dass der Rule-Node der vierten Ebene nur über einen Goal-Node der fünften Ebene verfügt. Alle Goal-Nodes der fünften Ebene des Beispiels zeigen auf Relationen in der lokalen Quelle *L1*. Dieser Peer muss somit keine weiteren Peers anfragen, um die Anfrage vollständig beantworten zu können.

4.1.2 Sonderfall LaV-Expansion

Der im vorherigen Abschnitt beschriebene Aufbau des RG-Trees bezieht sich auf die Verwendung von GaV-Mappings. Diese Mappings haben im Rumpf jeweils nur ein Ziel, das ein Teilziel der Anfrage ersetzt. Im Kopf können GaV-Mappings mehrere Ziele besitzen, die dann jeweils zu einem Kind eines Rule-Nodes der vierten Ebene werden.

Anders verhält es sich bei LaV-Mappings. Diese Mappings haben im Kopf jeweils nur ein Ziel, so dass es zu jedem benutzten LaV-Mapping nur einen Goal-Node der fünften Ebene gibt. Da LaV-Mappings mehrere Teilziele in ihrem Rumpf haben können, besteht die Möglichkeit, dass sie mehrere Teilziele der Anfrage abdecken. Das Beispiel 4.2 zeigt diesen Fall.

Beispiel 4.2 *Beispiel einer LaV Expansion zur Anfrage:*

Peer001.q(a, b, c, d) :- Peer001.t(c, d), Peer001.s(a, b, c)

```
[] Peer001.q(a, b, c, d)
  () Peer001.q(a, b, c, d) :- Peer001.t(c, d), Peer001.s(a, b, c)
    [] Peer001.t(c, d)
      () Peer2.t1(a, b, c, d) :- Peer001.s(a, b, c), Peer001.t(c, d)
        [] Peer2.t1(a, b, c, d)
           [unc] Peer001.s(a, b, c)
    [] Peer001.s(a, b, c)
```

`[]` = *Goal-Node*
`()` = *Rule-Node*
`[unc]` = *Uncle-Node*

Die Anfrage dieses Beispiels hat zwei Teilziele, so dass zwei Goal-Nodes der dritten Ebene im RG-Tree angelegt werden. Der erste dieser Goal-Nodes wird durch das LaV-Mapping abgedeckt und hat somit nur einen Rule-Node als Kind. Da es nur ein Ziel im Kopf des Mappings gibt, besitzt der Rule-Node ebenfalls nur einen Goal-Node, der in diesem Beispiel einer Anfrage an Peer2 nach der Relation t1 entspricht.
Der zweite Goal-Node der dritten Ebene und damit das zweite Teilziel der Anfrage wird ebenfalls durch das verwendete LaV-Mapping abgedeckt und deshalb nicht weiter expandiert. Der Uncle-Node deutet diese Abdeckung an.

Im Beispiel 4.2 werden zwei Teilziele der Anfrage durch ein LaV-Mapping und eine Anfrage an *Peer001* abgedeckt. Sie sind durch eine Uncle-Beziehung miteinander verknüpft. Für die Erzeugung des Anfrageplanes sind diese Uncle-Node Referenzen hinderlich. Das Umschreiben der RG-Trees in einen Anfrageplan, wie es im nächsten Abschnitt erklärt wird, ist mit diesen Referenzen nicht möglich. Sie werden daher vor der Erzeugung des Anfrageplanes aufgelöst. Der RG-Tree aus dem Beispiel wird in den folgenden RG-Tree umgeschrieben:

```
[] Peer001.q(a, b, c, d)
  () Peer001.q(a, b, c, d) :- Peer001.t(c, d), Peer001.s(a, b, c)
    [] Peer001.t(c, d)
      () Peer2.t1(a, b, c, d) :- Peer001.s(a, b, c), Peer001.t(c, d)
        [] Peer2.t1(a, b, c, d)
    [] Peer001.s(a, b, c)
      () Peer2.t1(a, b, c, d) :- Peer001.s(a, b, c), Peer001.t(c, d)
        [] Peer2.t1(a, b, c, d)
```

Nach dem Auflösen der Uncle-Nodes im RG-Tree gibt es zwei Goal-Nodes der fünften Ebene, die die gleiche Anfrage an *Peer2* darstellen. Die spätere Anfrageausführung kann

dies berücksichtigen, so dass diese Anfrage nur einmal ausgeführt wird. Im *System P* werden dazu die im Abschnitt 5.3 beschriebenen Zwischenspeicher verwendet.

Aus dem Ergebnis der Anfrage werden jeweils in den Goal-Nodes der dritten Ebene Attribute weggelassen. Die weggelassenen Attribute sind im Beispiel die Variablen a und b von *t1* beim ersten Goal-Node der dritten Ebene sowie d von *t1* beim zweiten Goal-Node. Anschließend werden diese Teilziele der Anfrage über gemeinsame Attribute wieder zusammengefügt (Join). Im Beispiel geschieht dies über die Variable c. Der umgeschriebene RG-Tree ist im Ergebnis der Anfrage äquivalent zum Ergebnis einer Anfrage des RG-Trees aus dem Beispiel 4.2, wenn die Join-Attribute im Mapping ein Schlüssel der Quell-Relation (hier *Peer2.t1*) sind und keine Projektion eines Join-Attributes im LaV-Mappings stattfindet. Der Beweis der Äquivalenz wird im Folgenden skizziert.

Sei

$$R_q(A_1, \ldots, A_n) : -R_1(A_1, ..., A_i), R_2(A_j, ..., A_n) \text{ mit } j < i$$

ein LaV-Mapping, wobei R_q eine Relation einer Quelle ist und R_1 und R_2 Relationen aus dem Peerschema[4]. A_1 bis A_n sind Attribute der Relationen, wobei ein Join über den Attributen A_j bis A_i erfolgt.

Sei nun

$$q(A_x, \ldots, A_y) : -R_1(A_1, ..., A_i), R_2(A_j, ..., A_n) \text{ mit } j < i \text{ und } x, y \in [1, \ldots, n]$$

eine Anfrage gegen das Peer-Schema.

Diese Anfrage kann auf zwei verschiedene Arten beantwortet werden. Die erste Art entspricht dem RG-Tree mit Uncle-Beziehungen. Dabei wird q folgendermaßen in q' umformuliert:

$$q'(A_x, \ldots, A_y) : -R_q(A_1, \ldots, A_n)$$

Die Anfrage q wird somit direkt aus der Relation R_q beantwortet. Bei der zweiten Variante wird die Anfrage q aus dem Join der beteiligten Relationen R_1 und R_2 berechnet. Dies ist der Fall, nachdem die Uncle-Beziehungen aufgelöst wurden. q wird zu q'' umformuliert:

$$q''(A_x, \ldots, A_y) : -\underbrace{\Pi_{A_1,...,A_i}(R_q)}_{R_1}, \underbrace{\Pi_{A_j,...,A_n}(R_q)}_{R_2}$$

Zu zeigen ist nun, dass für beliebige Extensionen von R_q gilt: $q' = q''$

$$\begin{aligned} & q'(A_x, \ldots, A_y) = q''(A_x, \ldots, A_y) \\ \Leftrightarrow & R_q(A_1, \ldots, A_n) = \Pi_{A_1,...,A_i}(R_q) \bowtie \Pi_{A_j,...,A_n}(R_q) \end{aligned}$$

[4] Zur Vereinfachung wurden im Mapping nur zwei Relationen aus dem Peer-Schema genommen, gleiches gilt auch für n-Relationen.

Der Join über den beiden Projektionen von R_q entspricht genau dann R_q selbst, $A_j, \ldots, A_i$ Schlüssel für R_q sind. □

Die Auflösung der Uncle-Beziehungen liefert ebenfalls ein korrektes Ergebnis, wenn das LaV-Mapping nicht alle Teilziele der Anfrage abdeckt. Eine Anfrage Q kann derart in zwei Anfragen zerlegt werden, dass sämtliche Teilziele der ersten Anfrage Q_1 durch des LaV-Mapping abgedeckt werden. Es gelten somit die obigen Betrachtungen. Die zweite Anfrage Q_2 enthält die restlichen Teilziele von Q, welche durche weitere Mappings abgedeckt sein müssen. Der Join von Q_1 und Q_2 liefert dann das Ergebnis der ursprünglichen Anfrage Q.

4.2 Anfrageplan und Anfrageausführung

Nach dem Aufbau des lokalen RG-Tree erfolgt die Übersetzung in einen lokalen Anfrageplan und dessen Ausführung. Dabei werden Ergebnisse benachbarter Goal-Nodes zusammengefügt (Join) und Ergebnisse benachbarter Rule-Nodes vereinigt (Union). Ein Rule-Node im RG-Tree nimmt im Anfrageplan somit die Rolle des Join-Operators ein. Ein Goal-Node repräsentiert den Union-Operator. Die Operatoren werden jeweils auf ihre Kindern angewandt.

Nach einer Optimierung des lokalen Anfrageplanes, wie sie im nächsten Abschnitt erläutert wird, erfolgt die Ausführung der Anfragen aus den Goal-Nodes der fünften Ebene. Nach dem Erhalt der Ergebnisse der Anfragen müssen diese zusammengefügt werden. Für den Fall, dass Goal-Nodes der fünften Ebene benachbart (verschwistert) sind, muss ein Join über den Ergebnissen ihrer Anfragen erfolgen. Die Ergebnisse der Anfragen oder Joins werden an die Rule-Nodes der vierten Ebene gereicht. Gibt es benachbarte Rule-Nodes der vierten Ebene (das ist genau dann der Fall, wenn für ein Teilziel der Anfrage mehrere Mappings verwendet wurden) müssen die Ergebnisse dieser Rule-Nodes durch den Union-Operator vereinigt werden. Die Ergebnisse der Rule-Nodes bzw. ihrer Vereinigungen werden an die Goal-Nodes der dritten Ebene übergeben. Diese Goal-Nodes stehen für die Teilziele der Anfrage, so dass wieder ein Join auf gleichen Attributen ihrer Ergebnisse erfolgen muss. Enthält die Anfrage Selektionen, so müssen diese auf das Resultat des Joins angewendet werden. Das Ergebnis dieser Selektion ist dann gleichzeitig das Ergebnis der Anfrage, die dem Peer gestellt wurde und wird an den Aufrufer zurückgegeben.

An dieser Stelle soll kurz der Unterschied zwischen lokalem und globalem RG-Tree sowie lokalem und globalem Anfrageplan verdeutlicht werden. Ein lokaler RG-Tree und ein lokaler Anfrageplan beziehen sich immer auf den aktuell betrachteten PDMS-Peers. Dieser ersetzt die Teilziele einer Anfrage durch die benutzten Mappings und erstellt somit den lokalen RG-Tree, welcher aus Rule- und Goal-Nodes besteht. Nach dem Auflösen der Uncle-Beziehungen erfolgt die Übersetzung und den lokalen Anfrageplan, der nur die Operatoren Union und Join enthält. Der globale oder vollständige Anfrageplan (im folgenden auch Anfragebaum genannt) ist der Anfrageplan, der entsteht, wenn alle lokalen Anfragepläne der an der Bearbeitung einer Anfrage beteiligten Peers zusammen betrach-

tet werden[5]. Der globale Anfrageplan kann neben dem Anfrageergenis im Monitor-Peer dargestellt werden. Die gleiche Betrachtung gilt für den globalen RG-Tree. Dieser setzt sich ebenfalls aus den lokalen RG-Trees zusammen. Er wird jedoch im *System P* nicht dargestellt.

4.2.1 Optimierungen des Anfrageplanes im *System P*

Der erzeugte Anfrageplan lässt sich optimieren. Durch das Verlagern der Join-Operatoren und Selektionen in die Quellen[6] kann die Anzahl der übertragenen Tupel verringert und somit die Ausführungszeit verkürzt werden.

Die im vorherigen Abschnitt erläuterte naive Ausführung des Anfrageplanes führt Joins im anfragenden Peer und nicht in den Quellen aus. Dies hat zur Folge, dass ein angefragter Peer die Relationen der Anfrage komplett übertragen muss. Im *System P* werden Joins, die durch benachbarte Goal-Nodes der fünften Ebene entstehen, in die Quellen verlagert. Eine Quelle wird dann nicht mehr nach einzelnen Relationen gefragt, sondern nach dem Join über diesen Relationen. Dies geschieht aber nur, wenn der Join über mindestens einem gemeinsamen Attribut erfolgt. Ansonsten ist das Ergebnis des Joins das Kreuzprodukt beteiligten Relationen. Es müssten mehr Daten übertragen werden, als bei der Übertragung der einzelnen Relationen.

Eine weitere Optimierung im *System P* ist, dass Selektionen nicht erst vor der Rückgabe des Ergebnisses und damit am Ende der Anfragebearbeitung ausgeführt werden, sondern den Anfragen an Quellen mitgegeben werden. Eine Quelle, liefert dann nur die für die Anfragebearbeitung relevanten Tupel. Für den Fall, dass die Quelle ein weiterer Peer ist, bedeutet dies, dass insgesamt weniger Daten übertragen werden müssen.

4.2.2 Komplettes Beispiel zur Anfragebearbeitung

Im folgenden Beispiel soll anhand einer Anfrage an *Peer001* der Aufbau des Anfrageplanes, seine Optimierung und seine Ausführung im *System P* erläutert werden.

Peer001 hat das Peer-Schema:
s(Attribute1, Attribute2, Attribute3)
t(Attribute4, Attribute5, Attribute6)

Er besitzt zu seiner lokalen Quelle *LS001* das lokale Mapping:
LS001.S1(a, b, c) :- Peer001.s(a, b, c)

Sowie die folgenden Peer-Mappings:
Peer003.S1(a, b), Peer003.T1(b, c) :- Peer001.s(a, b, c)
Peer002.T1(a, b, c) :- Peer001.t(a, b, c)
Peer004.T1(a, b, c) :- Peer001.t(a, b, c)

[5] Die einzelnen lokalen Anfragepläne unterschiedlicher Peers werden dabei über die Peer-Anfragen miteinander verknüpft.

[6] Mit Quellen sind lokale Quellen oder weitere Peers gemeint

Die Anfrage lautet:
Peer001.q(a, b, c, d, e) :- Peer001.s(a, b, c), Peer001.t(c, d, e), a > 1, d < 4, e = 4

Der erzeugte RG-Tree zur Anfrage lautet dann:

```
[] Peer001.q(a, b, c, d, e)
  () Peer001.q(a, b, c, d, e) :- a > 1, d < 4, e = 4, Peer001.s(a, b, c), Peer001.t(c, d, e)
    [] Peer001.s(a, b, c), a > 1
      () Peer003.S1(a, b), Peer003.T1(b, c) :- Peer001.s(a, b, c)
        [] Peer003.S1(a, b), a > 1.0
        [] Peer003.T1(b, c)
      () LS001.S1(a, b, c) :- Peer001.s(a, b, c)
        [] LS001.S1(a, b, c), a > 1.0
    [] Peer001.t(c, d, e), e = 4, d < 4
      () Peer002.T1(a, b, c) :- Peer001.t(a, b, c)
        [] Peer002.T1(c, d, e), e = 4.0, d < 4.0
      () Peer004.T1(a, b, c) :- Peer001.t(a, b, c)
        [] Peer004.T1(c, d, e), d < 4.0, e = 4.0
```

Da die Anfrage aus zwei Teilzielen besteht, gibt es zwei Goal-Nodes der dritten Ebene. Für jeden dieser Goal-Nodes können je zwei Mappings genutzt werden, so dass jeder dieser Goal-Nodes zwei Rule-Nodes der vierten Ebene als Kinder hat. Das Peer-Mapping zu *Peer003* ist ein GaV-Mapping mit zwei Teilzielen im Kopf. Der Rule-Node für dieses Mappings hat demzufolge auch zwei Goal-Nodes der fünften Ebene als Kinder. Die weiteren Mappings bestehen jeweils nur aus einem Teilziel in ihrem Kopf, so dass die restlichen Goal-Nodes der fünften Ebene keine Nachbarn (Geschwister) besitzen.

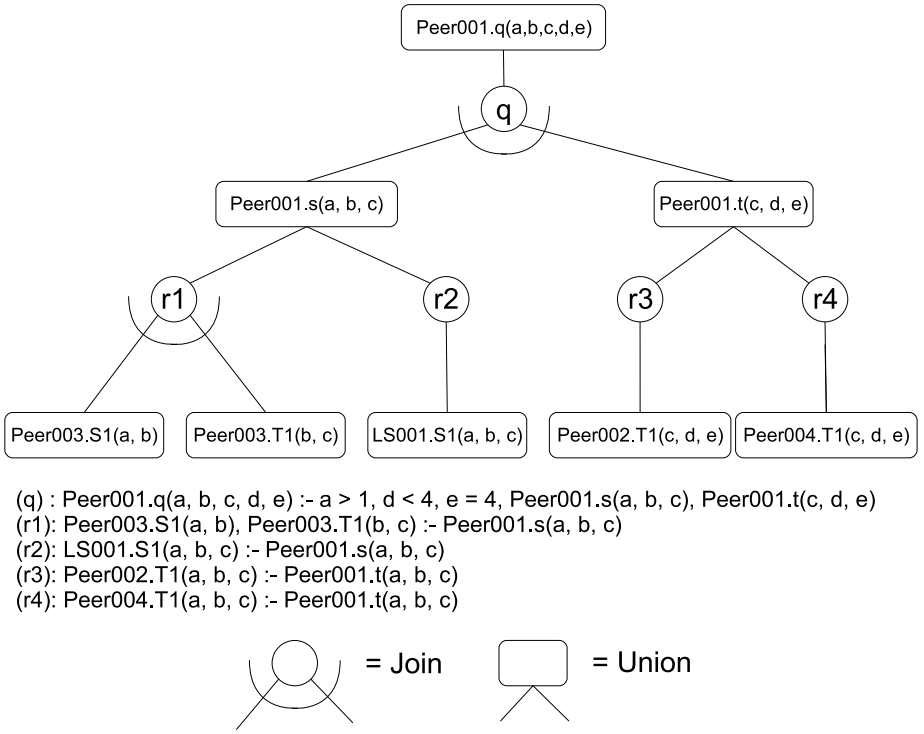

Abbildung 4.2: Nicht optimierter Anfrageplan

Die Abbildung 4.2 zeigt den aus dem RG-Tree gewonnenen, nicht optimierten Anfrageplan. Es sind insgesamt vier Anfragen an weitere Peers und eine Anfrage an eine lokale Quelle nötig. Diese Anfragen sind jeweils komplette *Table-Scans*, es muss der gesamte Inhalt der angefragten Tabellen übertragen werden.

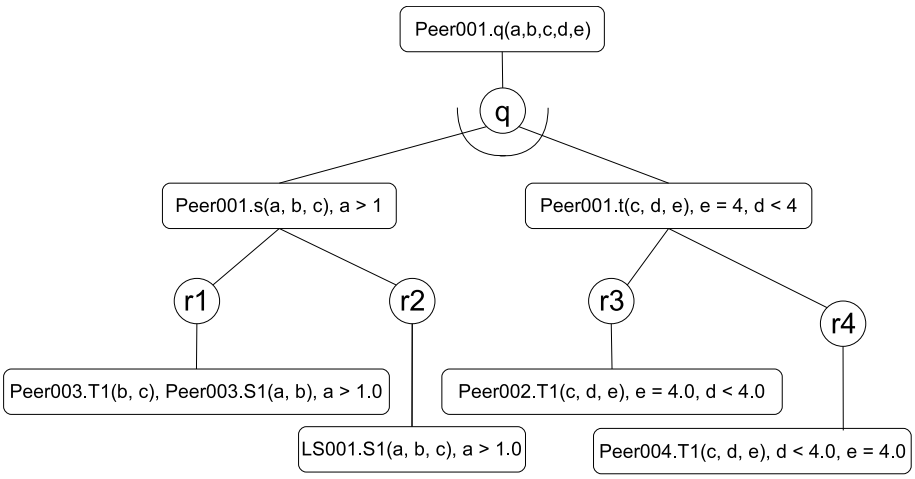

Abbildung 4.3: *System P* optimierter Anfrageplan
Legende wie in 4.2

Die Abbildung 4.3 zeigt den Anfrageplan nach den vom *System P* vorgenommenen Optimierungen. Der Join der Tabellen *S1* und *T1* von *Peer003* findet nicht mehr im angefragten *Peer001*, sondern in *Peer003* statt. Die Selektionen werden an die Quellen weiter gegeben, so dass möglichst keine kompletten Tabellen übertragen werden.

4.3 Testfälle zur Anfragebearbeitung

Um die Implementierung der Anfragebearbeitung im *System P* zu testen, wurden JUnit Testfälle geschrieben. Diese sind in der Klasse *QueryProcessingTestCase* zu finden. In der folgenden Auflistung der Testfälle ist beschrieben, was getestet wird und welche Daten verwendet werden.

- *Anfrage an Peer mit einfachem Schema, einer lokalen Quelle mit gleichem Schema und einem lokalen Mapping*
  Das Peerschema besteht aus einer einzigen Relation. Diese Relation ist auch Teil des Schemas der lokalen Quelle. Das lokale Mapping bildet die Relation des Peerschemas eins zu eins auf die Relation der lokalen Quelle ab. Die Anfrage fragt nach den Tupeln der Relation des Peerschemas und beinhaltet eine Selektion. Es wird erwartet, dass alle Tupel der lokalen Quelle, die die Selektionsbedingung erfüllen, geliefert werden.

- *Anfrage an Peer mit einfachem Schema, einer lokalen Quelle mit anderem Schema und einem lokalen Mapping*
  Das Peerschema besteht aus einer einzigen Relation. Das Schema der lokalen Quelle

besitzt zwei Relationen, wobei der Join über einem Attribut beider Relationen die Relation des Peerschemas ergibt. Das lokale Mapping ist ein GaV-Mapping.

- *Anfrage mit Join*
  Das Peerschema besteht aus zwei Relationen. Das Schema der lokalen Quelle besitzt drei Relationen. Es sind zwei lokale GaV-Mappings vorhanden. Die Anfrage besteht aus dem Join der beiden Relationen des Peer-Schemas.

- *Anfrage an Peer ohne lokale Quelle aber mit Peer-Mapping*
  Der angefragte Peer besitzt keine lokale Quelle, jedoch ein einfaches (GaV) Peer-Mapping zu einem zweiten Peer mit lokaler Quelle und gespeicherten Daten. Es wird erwartet, dass dieses Mapping genutzt wird. Der erste Peer fragt den zweiten Peer an und liefert als Ergebnis die Daten des zweiten Peers zurück.

- *Anfrage an Peer mit zwei lokalen Quellen*
  Das Peerschema besteht aus einer einzigen Relation. Diese Relation ist auch Teil der Schemata beider lokaler Quellen. Diese Quellen enthalten gemeinsame und auch unterschiedliche Tupel. Zu jeder Quelle gibt es ein lokales Mapping. Es wird erwartet, dass eine Anfrage an die Relationen des Peerschemas die Vereinigung der Tupel der lokalen Quellen liefert. Insbesondere sollen dabei keine Tupel doppelt geliefert werden.

- *Anfrage an Peer mit GaV-Peer-Mapping, das zwei verschiedene Peers enthält*
  Der angefragte Peer hat selbst keine lokalen Quellen aber ein Peer-Mapping, das zwei verschiedene Peers enthält, die selber lokale Quellen besitzen. Erwartet wird der Join der Tupel aus den lokalen Quellen der beiden Peers.

- *Anfrage an Peer mit lokalem LaV-Mapping*
  Das Peerschema besteht aus zwei Relationen. Die lokale Quelle besitzt nur eine Relation, die dem Join der beiden Relationen aus dem Peerschema entspricht. Ein lokales LaV-Mapping drückt diese Beziehung aus. Getestet wird das korrekte Auflösen der Uncle-Beziehung und die Rückgabe der erwarteten Tupel mit Hilfe einer Anfrage gegen das Peerschema, die aus dem Join der beiden Relationen besteht.

Es werden neben den relationalen Operatoren „Join“ und „Union“ auch die Zugriffe auf lokale Quellen, sowie Anfragen an weitere Peers getestet. Ebenfalls gibt es Testfälle für die unterschiedlichen Mapping-Arten LaV und GaV. Es kann mit Hilfe dieser JUnit-Testfälle gezeigt werden, dass die wichtigsten Funktionen der Anfragebearbeitung und damit des *System P* prinzipiell funktionieren.

4.4 Visualisierung der Anfragebearbeitung

Im *System P* kann die (sequentielle) Anfragebearbeitung während der Ausführung beobachtet werden. Dies ist nützlich, um grob die beteiligten Peers und die Größe des Anfrageplanes abzuschätzen. In Abbildung 4.4 ist diese Visualisierung dargestellt.

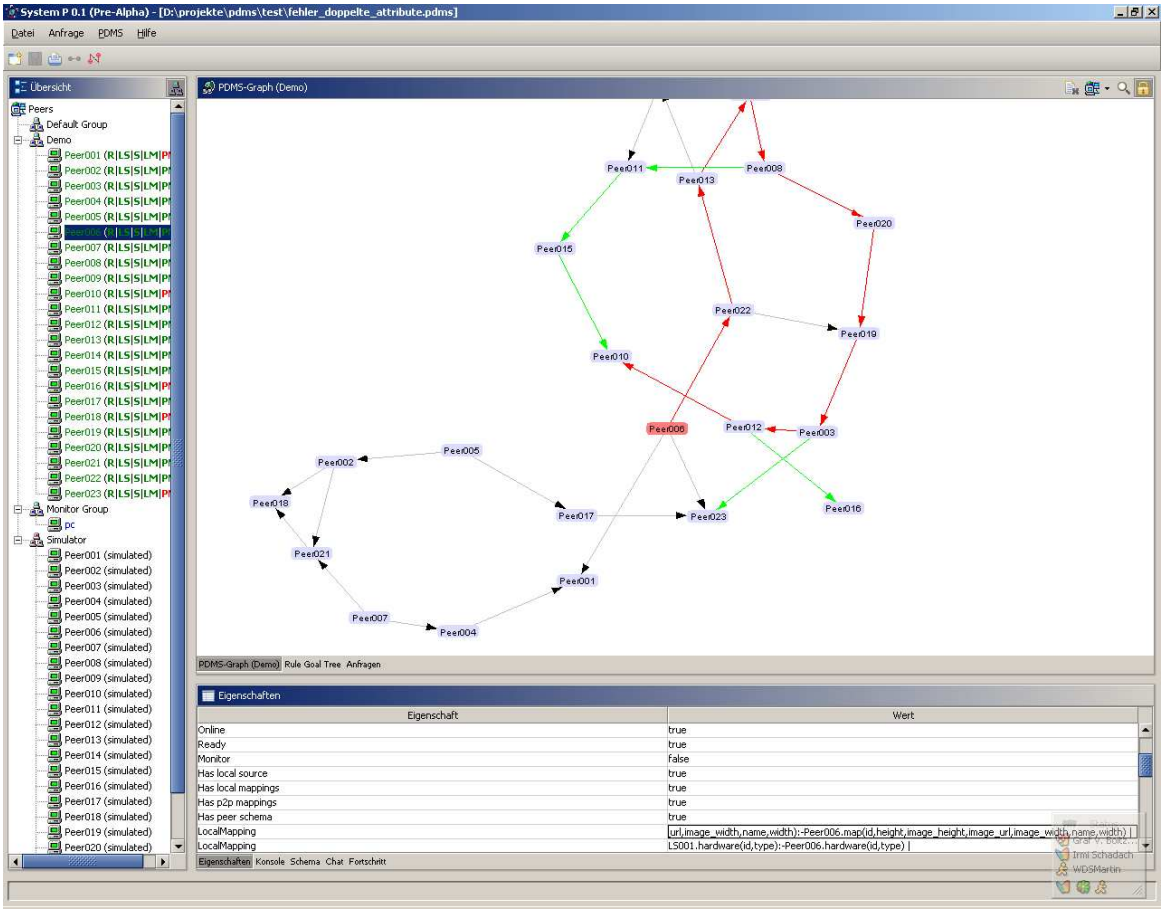

Abbildung 4.4: „Live“-Visualisierung der Anfragebearbeitung im *System P*
Grüne Linien stehen für gerade ausgeführte Anfragen, rote Linien symbolisieren beantwortete Anfragen.

Ebenso kann der vollständige Anfrageplan, der sich aus den lokalen Anfrageplänen der beteiligten Peers zusammensetzt, dargestellt werden. Die Abbildung 4.5 zeigt einen vollständigen Anfrageplan einer Anfrage. Es lassen sich die angefragten lokalen Quellen und Peers, sowie die verwendeten Operatoren erkennen.

4.5 Verbesserungsmöglichkeiten

Es sind mehrere Verbesserungsmöglichkeiten der Anfragebearbeitung im *System P* denkbar. Zum einen kann der erzeugte Anfrageplan eines Peers weiter optimiert, zum anderen kann die Art der Ausführung des Anfrageplanes durch Pipelining und Parallelisierung verbessert werden. Ebenso kann es für einen Benutzer wünschenswert sein, Auskunft über die Herkunft der Ergebnistupel zu erhalten.

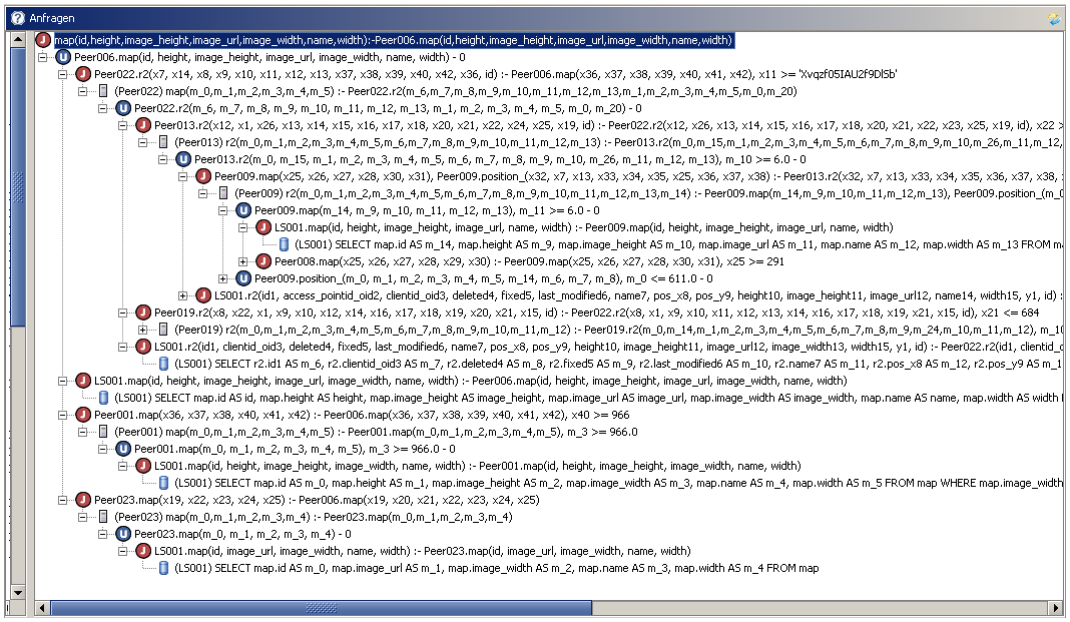

Abbildung 4.5: Vollständiger Anfrageplan

4.5.1 Verbesserungen bei der Optimierung des Anfrageplanes

Die derzeitige Implementierung des *System P* schöpft die Möglichkeiten der Optimierungen des Anfrageplanes nicht vollständig aus. Es werden nur Optimierungen auf den Goal-Nodes der fünften Ebene vorgenommen. Denkbar wären aber auch Optimierungen auf den Rule-Nodes der vierten Ebene und den Goal-Nodes der dritten Ebene, wenn in diesen dieselbe Quelle angefragt wird. Die Zahl der Anfragen und auch die Zahl der übertragenen Tupel kann so reduziert werden. Das folgende Beispiel 4.3 verdeutlicht diese Art der Optimierung.

Beispiel 4.3 *Beispiel einer Optimierung auf den Goal-Nodes der dritten Ebene des Anfrageplanes:*

Gegeben sei ein Peer „Peer001“. Dieser Peer besitzt die beiden Peer-Mappings:

Peer002.R1(a, b, c), Peer002.R2(c, d) :- Peer001.R1(a, b, c)
Peer002.R3(a, b) :- Peer001.R3(a, b)

Die RG-Tree zur Anfrage Peer001.q(a, b, c, d) :- Peer001.R1(a, b, c), Peer001.R3(c, d) *sieht folgendermaßen aus:*

```
[] Peer001.q(a, b, c, d)
  () Peer001.q(a, b, c, d) :- Peer001.R1(a, b, c), Peer001.R3(c, d)
    [] Peer001.R1(a, b, c)
      () Peer002.R1(a, b, c), Peer002.R2(c, d) :- Peer001.R1(a, b, c)
        [] Peer002.R1(a, b, c)
        [] Peer002.R2(c, d__0)
    [] Peer001.R3(c, d)
      () Peer002.R3(a, b) :- Peer001.R3(a, b)
        [] Peer002.R3(c, d)
```

[] = *Goal-Node*
() = *Rule-Node*

Nach der im System P vorgenommenen Optimierung würden zwei Anfragen an Peer002 gestellt werden:

q(a, b, c) :- Peer002.R1(a, b, c), Peer002.R2(c, d__0)
q(c, d) :- Peer002.R3(c, d)

Deren Ergebnistupel müssten im Peer001 wieder zusammengeführt werden (Join). Gäbe es eine weitere Optimierung auf den Goal-Nodes der dritten Ebene, könnten beide Anfragen zu einer zusammengefasst werden. Der Join könnte in der Quelle (Peer002) erfolgen und es wären insgesamt weniger Daten[7] zu übertragen. Die optimierte Anfrage von Peer001 an Peer002 würde lauten:

q(a, b, c, d) :- Peer002.R1(a, b, c), Peer002.R2(c, d__0), Peer002.R3(c, d)

Peer001 muss keine weiteren Operationen auf den Ergebnistupeln dieser Anfrage durchführen, sondern kann diese gleich als Ergebnis der an ihn gestellten Anfrage zurückliefern.

Als zweite Verbesserung in der Optimierung des Anfrageplanes kann bei verschiedenen Quellen über den Einsatz von Semi-Join-Strategien nachgedacht werden. Dadurch könnte die Anzahl der insgesamt über das Netzwerk zu übertragenen Daten weiter reduziert werden. Dies setzt allerdings voraus, dass die Quellen untereinander kommunizieren können. Im *System P* können die Quellen, weitere PDMS-Peers, nur dann miteinander kommunizieren, wenn das Ziel in den Peer-Mappings auftaucht. Dies ist in Abschnitt 3.3.2 beschrieben und entspricht in etwa der Situation in einem „echten“ PDMS, das über unterschiedliche Netzwerke verbunden ist. In solch einem PDMS ist nicht sicher gestellt, dass jeder Peer mit jedem anderen Peer kommunizieren kann.

[7] In Abhängigkeit der gespeicherten Daten in der Quelle Peer002

4.5.2 Verbesserungen bei der Anfrageausführung

Neben der logischen Optimierung des Anfrageplanes kann auch seine Ausführung verbessert werden. In der derzeitigen Implementierung liefert ein Peer erst eine Antwort auf eine Anfrage, wenn es diese vollständig ermittelt hat. Vor allem bei großen Peer-Data-Management-Systemen kann die Ermittlung der vollständigen Antwort sehr zeitaufwendig sein. Ein Nutzer möchte vielleicht lieber schnell eine zunächst unvollständige Antwort haben, als sehr lange auf eine vollständige Antwort warten zu müssen. Neben den im nächsten Kapitel erklärten Pruning-Strategien bietet sich Pipelining bei der Anfragebeaerbeitung an. Das heißt, ein Peer gibt jedes ermittelte Tupel der Ergebnismenge sofort an den Aufrufer zurück, anstatt diese zu sammeln und komplett zurückzugeben.

Für den Fall, dass ein Peer zur Anfragebearbeitung mehrere weitere Peers anfragen muss, sollten diese Anfragen parallel erfolgen. Die jetzige Implementierung stellt diese Anfragen sequentiell. Es wird immer auf das Ergebnis einer Anfrage gewartet bevor eine weitere Anfrage gestellt wird. Der Grund dafür ist eine einfachere Implementierung. Die parallele Ausführung bedeutet eine zusätzliche Komplexität, die eine schnellere Ausführung der Anfragen zur Folge hat. Um die Vorteile eines verteilten Datenbanksystems und einer verteilten Anfragebearbeitung ausnutzen und des beschriebenen Pipelinings umsetzen zu können, sollten Anfragen parallel ausgeführt werden.

4.5.3 Verbesserungen bei der Anzeige der Datenherkunft und Datenqualität

Neben der Optimierung der Anfrageausführung ist es für den Benutzer wünschenswert, Informationen über die Herkunft der Ergebnistupel (Data-Lineage[8]) zu erhalten. Nur wenn ein Benutzer die Qualität eines Anfrageergebnisses beurteilen kann, ist es für ihn nützlich. Im *System P* sind die Informationen über die Datenherkunft und -qualität nur begrenzt verfügbar. Zusätzlich zum Anfrageergebnis werden die Namen der beteiligten Peers, die Tiefe des Anfragebaumes, die Anzahl der ausgeführten Operationen (Join, Union) sowie die Anzahl der benutzten Mappings angezeigt. Auf Wunsch kann auch der komplette Anfragebaum einer Anfrage, in dem die verwendeten Mappings erkennbar sind, angezeigt werden. Diese Informationen reichen jedoch nicht aus, um die Quellen eines Ergebnis-Tupels eindeutig bestimmen zu können. Damit ein Benutzer die Qualität des Anfrageergebnisses genau ermitteln kann, ist es hilfreich dessen Data-Lineage zu kennen. Detaillierte Angaben aus welchen Quellen die Daten eines Ergebnistupels stammen und welche Operationen auf diesen ausgeführt wurden, kann nur die Anfragebearbeitung bereitstellen. Die Ansätze aus [3] lassen sich dabei auf ein PDMS übertragen. Pro Ergebnistupel müssen dazu die Pfade im Anfragebaum markiert werden, die auf die Quellen zeigen, aus denen Tupel zur Berechnung des Ergebnistupels verwendet wurden. Ein Problem dabei ist, dass die zu übertragenden Metainformationen bei größeren PDMS-Instanzen und großen Anfragebäumen sehr umfangreich werden.

[8] Beschreibung und Ansätze sind in [3] zu finden.

5 Pruningstrategien

Mit wachsender Größe eines PDMS wächst im Allgemeinen auch die Zahl der an einer Anfrage beteiligten Peers und somit die Größe des gesamten Anfrageplans (Anfragebaums)[1] von Anfragen. Schon in kleineren PDMS-Instanzen kann die vollständige Ausführung eines Anfragenplanes sehr viel Zeit und Ressourcen beanspruchen. In größeren PDMS-Instanzen wird sie praktisch unmöglich. Es ist daher wünschenswert oder notwendig, auf die Ausführung von Teilen des Anfrageplanes zu verzichten. Dieses Vorgehen wird als Pruning bezeichnet. Bildlich gesehen versteht man darunter das Abschneiden eines Astes im Anfragebaum. Anstatt die Anfrage komplett zu beantworten, wird nach bestimmten Kriterien oder Strategien entschieden, einzelne Quellen oder Mappings über bestimmte Anfragepfade nicht in die Anfragebearbeitung einzubeziehen. Dadurch werden weniger Ressourcen zur Beantwortung einer Anfrage benötigt. Der Preis dafür ist ein möglicherweise unvollständiges Anfrageergebnis. Ziel von Pruningstrategien sollte es sein, die Kosten der Anfragebearbeitung möglichst gering zu halten und gleichzeitig die Qualität der Anfrageergebnisse zu wahren. Alternativ können Pruningstrategien auch das Ziel verfolgen, bei einer vorgegebenen Begrenzung der Kosten, ein möglichst vollständiges Anfrageergebnis zu liefern. Diese Kosten(budget)-basierten Pruningstrategien werden in Abschnitt 5.6 beschrieben.

Zunächst soll ein einleitendes Beispiel zeigen, warum Pruning nötig ist. Anschließend werden in diesem Kapitel mögliche Pruningstrategien beschrieben und klassifiziert.

5.1 Motivation

Warum es sinnvoll und notwendig ist, Pruning in einem PDMS einzusetzen, verdeutlicht das Beispiel 5.1. Es soll einen Eindruck der Größe des kompletten Anfrageplanes bei einem gegebenen PDMS vermitteln. Die Details des PDMS sind dafür nicht nötig und werden deshalb ausgelassen. Die Berechnung des Anfrageplanes erfolgte mit dem *System P*.

Beispiel 5.1 *Größe des Anfrageplanes eines PDMS ohne Pruning*

Es wurde mit der in [6] beschriebenen Testumgebung ein PDMS erzeugt. Das eingesetzte Schema besteht aus 11 Relationen, die jeweils 3 bis 9 Attribute besitzen. Das erzeugte PDMS setzt sich aus 31 Peers zusammen. Jeder dieser Peers besitzt zwischen 5 und 15 Peer-Mappings. Dabei ist der Maximal-Grad[2] des PDMS gleich 5. In Abbildung 5.1 ist der Graph dieses PDMS dargestellt.

Die an Peer005 gestellte, einfache Anfrage nach den Tupel der Relation „client“ lautet:

$$client(id, hardwareid\_oid, mac, name, sessionid\_oid) :- Peer005.client(id, hardwareid\_oid, mac, name, sessionid\_oid)$$

[1] Der vollständige Anfragebaum ist der Baum, der sich aus allen lokalen Anfragebäumen der an der Anfragebearbeitung beteiligten Peers zusammensetzt.

[2] Der Grad eines Peers ist gleich die Anzahl der Peers, die dieser Peer kennt addiert mit der Anzahl der Peers, die diesen Peer kennen.

Die Ausführung dieser Anfrage erzeugte einen vollständigen Anfragebaum der Tiefe 21, ein Pfad des Anfragebaumes von der Wurzel bis zu den Blättern bestand somit aus maximal 21 Peers und einer lokalen Quelle. Insgesamt wurden zur Beantwortung der Anfrage 71006 Peer-Mappings und 88648 lokale Mappings genutzt[3]*. Dies führte zu 69773 weiteren Peer-Anfragen und zu 88648 Zugriffen auf lokale Quellen. Es waren 34378 Union- und 17035 Join-Operationen nötig. 30 Peers des PDMS waren an der Bearbeitung der Anfrage beteiligt. Die Anfragebearbeitung benötigte 4.156.123ms(ungefähr 1h). Dabei fand keine Kommunikation über ein Netzwerk statt, da alle Peers in einer JVM liefen. Es wurde nur der Anfrageplan erstellt und keine Ergebnistupel übertragen. Die Beantwortung der Anfrage und somit das Übertragen von Daten würde die Bearbeitungszeit weiter erhöhen.*

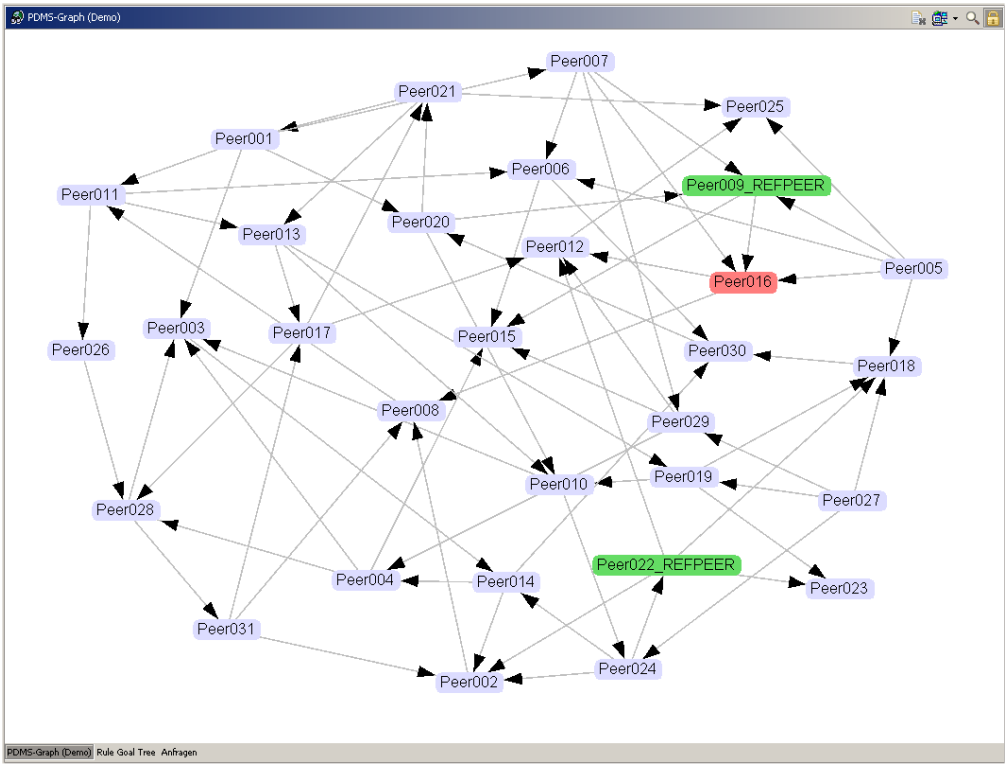

Abbildung 5.1: PDMS mit Maximalgrad fünf

5.2 Zyklenerkennung

In einem komplexen PDMS kann es vorkommen, dass bei der Anfragebearbeitung ein Peer, der weitere Peers anfragt, durch seine eigenen Anfragen ausgelöst, erneut angefragt wird. Im Anfragepfad erscheint dieser Peer mehrfach. Problematisch wird es, wenn dadurch ein unendlicher Zyklus entsteht. Dieser Zyklus muss entdeckt und aufgelöst werden, da andernfalls die Anfragebearbeitung nicht terminiert.

[3] Einzelne Mappings wurden mehrfach genutzt.

Abbildung 5.2 zeigt einen einfachen Fall, wie es zu einem unendlichen Zyklus in der Anfragebearbeitung kommen kann. *Peer1* und *Peer2* besitzen das gleiche Peer-Schema und je ein Peer-Mapping, das auf die Relation des jeweils anderen Peers zeigt. Bei einer Anfrage gegen das Schema eines der beiden Peers liefert dieser Peer die Daten seiner lokalen Quelle und führt eine Anfrage durch Nutzung seines Peer-Mappings an den zweiten Peer aus. Dieser liefert ebenfalls die Daten seiner lokalen Quelle und fragt erneut den ersten Peer an. Ohne ein Abbruch-Kriterium terminiert die Anfragebearbeitung in diesem Fall nicht.

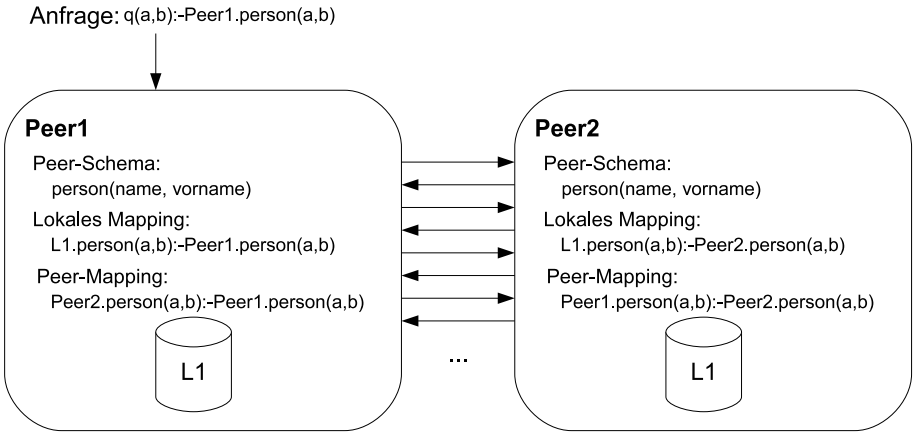

Abbildung 5.2: Kreis im Anfrageplan

Das Problem, alle Antworten in einem PDMS mit zyklischen Mappings[4] zu finden, ist nach Theorem 3.1 in [5] unentscheidbar. Ein allgemeines, korrektes Abbruchkriterium, das immer die vollständige Antwort für Anfragen liefert, die zu unendlichen Zyklen im Anfrageplan führen, kann somit nicht angegeben werden. Es kann somit nur versucht werden, unendliche Zyklen zu vermeiden. Dabei ist jedoch nicht sichergestellt, dass zu jeder Anfrage die vollständige Antwort gefunden wird.

Abbruchkriterien des Anfrageplaners zur Vermeidung unendlicher Zyklen

Im *System P* können zwei verschiedene Abbruchkriterien zur Vermeidung von unendlichen Zyklen während der Anfragebearbeitung angegeben werden. Das erste Abbruchkriterium, das Anfrage-Abbruchkriterium, bricht die Anfragebearbeitung ab, sobald ein Peer einem weiteren Peer mehr als einmal im Anfragepfad die gleiche Anfrage stellt. Die Abbildung 5.3 verdeutlicht diesen Fall.

Im Beispiel stellt $Peer1$ die Anfrage Q_2 an $Peer2$. Dieser stellt daraufhin eine Anfrage an $Peer3$, welcher wiederum $Peer1$ anfragt. $Peer1$ ist jetzt in der Situation, nochmal die Anfrage Q_2 an $Peer2$ zu stellen. Die Anfrage Q_2 wird bei Anwendung des Anfrage-Abbruchkriteriums nicht noch einmal gestellt. Wichtig dabei ist, dass das Kriterium nur

[4] Definition 3.1 in [5]

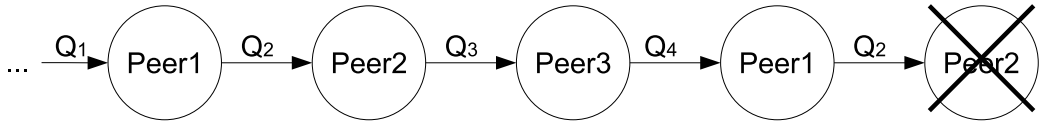

Abbildung 5.3: Erkennen von Zyklen anhand gleicher Anfragen in Anfragepfad

dann erfüllt ist, wenn exakt die gleiche Anfrage gestellt wird. Ändern sich durch die Verwendung von Peer-Mappings Selektionsattribute in Q_2, entsteht so die neue Anfrage Q'_2. Diese Anfrage würde dann von $Peer1$ an $Peer2$ gestellt werden. Dieser Fall ist in Abbildung 5.4 dargestellt.

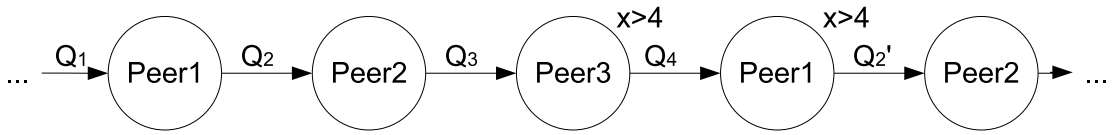

Abbildung 5.4: Veränderte Anfrage durch Selektionsprädikate in Peer-Mappings

Das verwendete Peer-Mapping von $Peer3$ schränkt die Variable x ein. Dadurch stellt $Peer1$ nun die veränderte Anfrage Q'_2 an $Peer2$. Das Abbruchkriterium ist nicht erfüllt und die Anfragebearbeitung wird fortgesetzt. Gleiches gilt für den Fall, dass sich zwei Anfragen Q und Q_p nur in der Projektion der Attribute im Ergebnis unterscheiden[5].

Das zweite Abbruchkriterium, das Mapping-Abbruchkriterium, verhindert die wiederholte Nutzung eines Peer-Mappings in einem Anfragepfad. Dabei ist die erlaubte Anzahl an Wiederholungen wählbar. Abbildung 5.5 zeigt die Wirkungsweise dieses Abbruchkriteriums für den Fall, dass ein Mapping nur einmal in einem Anfragepfad benutzt werden darf.

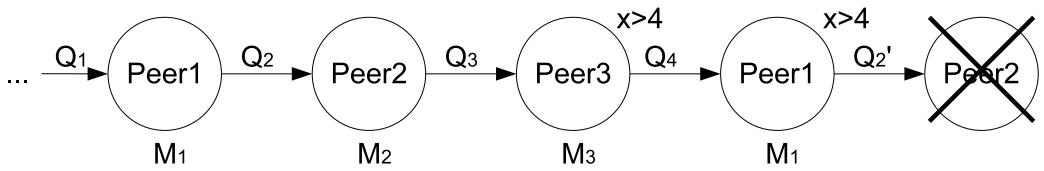

Abbildung 5.5: Verhinderung von Zyklen durch die Vermeidung mehrfacher Verwendung von Peer-Mappings

$Peer1$ erhält die Anfrage Q_1 und benutzt zur Beantwortung das Mapping M_1. Zur Bearbeitung der Anfrage Q_4, die im gleichen Anfragepfad an $Peer1$ gestellt wird, benutzt dieser das Mapping M_1 nicht erneut. Somit ist das Ende dieses Anfragepfades erreicht. Das Anfrage-Abbruchkriterium greift in diesem Fall nicht, da die Anfragen Q_2 und Q'_2

[5] zum Beispiel $\Pi_{\{D\}}Q = Q_p$ mit $D \neq \emptyset$ Menge von Attributen der Ergebnisrelation von Q_p

verschieden sind.

Im Gegensatz dazu ist in Abbildung 5.6 der Fall dargestellt, dass $Peer1$ zwei Mal die gleiche Anfrage Q_2 an $Peer2$ stellt. Die Anfragebearbeitung wird in diesem Fall nicht abgebrochen, da $Peer1$ unterschiedliche Mappings nutzt, die zur gleichen Anfrage führen[6]. Das Anfrage-Abbruchkriterium hingegen bricht die Anfrage in diesem Fall ab.

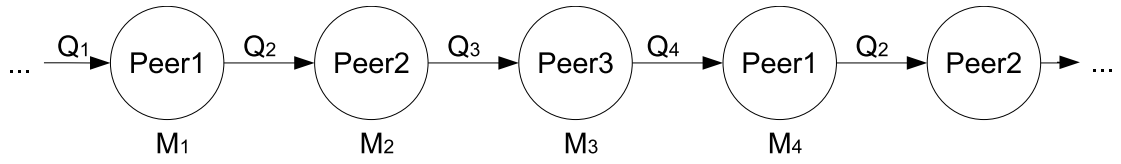

Abbildung 5.6: Fortsetzung des Anfrageplanes trotz gleicher Anfrage

Beide Abbruchbedingungen führen dazu, dass in besonderen Fällen nicht alle möglichen Tupel zu einer Anfrage geliefert werden. Das Beispiel 5.2 beschreibt diesen Fall.

Beispiel 5.2 *Beispiel einer Anfrage, für die beim Abbruch der Anfragebearbeitung nach dem Anfrage- und Mapping-Abbruchkriterium nicht alle möglichen Antworten gefunden werden.*

In Abbildung 5.7 ist die für dieses Beispiel verwendete PDMS-Instanz dargestellt. Die Peers dieser PDMS-Instanz besitzen jeweils die Relation $R1$ in ihrem Peerschema und ein zugehöriges Peer-Mapping PM, welches sich auf diese Relation bezieht. Die Relation $R2$ ist zusätzlich Teil des Peerschemas von $Peer1$. Zu ihr gehört ein lokales Mapping (LM), das auf die gleiche Relation in seiner lokalen Quelle LS verweist. $Peer2$ besitzt ebenfalls eine lokale Quelle LS mit zugehörigem lokalen Mapping LM.

Wird $Peer1$ die Anfrage Q gestellt, nutzt er zu deren Beantwortung sein Peer-Mapping PM, welches zur Anfrage Q_1 an $Peer2$ führt. $Peer2$ kann zum einem sein lokales Mapping LM nutzen, so dass die Tupel $(3, 8, 9)$ und $(6, 5, 4)$ Teil des Ergebnisses von Q_1 und somit auch von Q werden. Zum anderen kann das Peer-Mapping PM von $Peer3$ verwendet werden. Dies führt zur Anfrage Q_2 an $Peer3$, welcher sein Peer-Mapping PM[7] zur Beantwortung nutzen kann und $Peer1$ die Anfrage Q_3 stellt. Diese Anfrage ist ein Join zwischen den Relationen R_1 und R_2 des Peer-Schemas von $Peer1$. Für das erste Teilziel der Anfrage, die Relation $R2$, kommt das lokale Mapping LM zum Einsatz. Das zweite Teilziel kann mit der erneuten Verwendung des Peer-Mappings PM erreicht werden. Dessen Verwendung führt zur Anfrage Q_4, welche identisch mit der bereits gestellten Anfrage Q_1 ist. Beide Abbruchkriterien, sowohl das Anfrage- als auch das Mapping-Abbruchkriterium,

[6] Dass zwei Mappings zur gleichen Anfrage führen, ist ein eher theoretischer Fall, der aber durch Optimierungen auf dem lokalen Anfrageplan auch in der Praxis auftreten kann.

[7] Die Variablen s und t im Peer-Mapping sind nur Platzhalter und haben keinen Einfluss auf die Beantwortung von Q_2

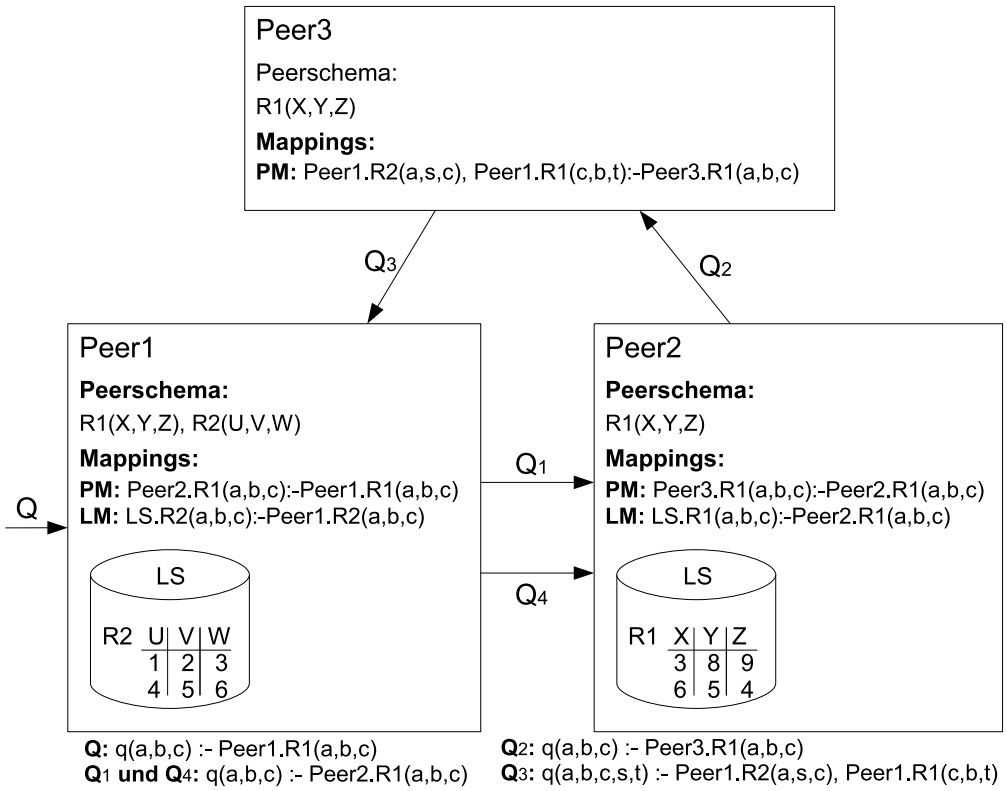

Abbildung 5.7: Verpasste Tupel bei Abbruch der Anfragebearbeitung

*sind erfüllt. Peer*1 *stellt somit nicht die Anfrage* Q_4*. Das Ergebnis des Joins zwischen den Relationen* R_1 *und* R_2 *und somit auch die Ergebnisse der Anfragen* Q_3 *und* Q_2 *sind leer. Es kommen keine weiteren Tupel zum Ergebnis der Anfrage* Q *hinzu.*
Würde die Anfrage Q_4 *jedoch ausgeführt werden, könnte Peer*2 *die Tupel* $(3, 8, 9)$ *und* $(6, 5, 4)$ *aus seiner lokalen Quelle an Peer*1 *liefern. Der Join in Peer*1 *aus den Tupeln der Relation R*2 *der lokalen Quelle und der Relation R*1*, die aus dem Ergebnis der Anfrage* Q_4 *wäre nicht leer, sondern bestünde aus den Tupeln* $(1, 8, 3)$ *und* $(4, 5, 6)$*. Diese beiden Tupel wären dann auch das Ergebnis der Anfragen* Q_3 *und* Q_2 *und kämen zum Ergebnis der Anfrage* Q_1 *und somit der Ergebnismenge von* Q *hinzu.*
Die Ergebnismenge der Anfrage ohne Anwendung der beiden Abbruchkriterien würde aus vier Tupeln bestehen. Mit Anwendung des Anfrage- oder des Mapping-Abbruchkriteriums besteht sie nur aus zwei Tupeln.

Das Beispiel 5.2 zeigt, dass das Anwenden der Abbruchkriterien dazu führen kann, dass nicht alle Tupel zu einer Anfrage gefunden werden. Dennoch ist es nötig ein Abbruchkriterium zu verwenden, da sonst die Anfragebearbeitung nicht in jedem Fall terminiert. Die Zyklenerkennung ist somit eine notwendige Pruningstrategie, die dazu führen kann, dass Anfragen nicht vollständig beantwortet werden.

Damit ein Peer entscheiden kann, ob er in einem Anfrageplan ein Mapping mehrfach genutzt oder eine Anfrage mehrfach gestellt hat, müssen zusätzliche Informationen ne-

ben der eigentlichen Anfrage übertragen werden. Diese Informationen können entweder die benutzten Mappings, die gestellten Anfragen selbst, oder IDs sein, anhand derer ein Peer eindeutig auf die bereits verwendeten Mappings und Anfragen schließen kann. Im *System P* erhält jede Anfrage eine eindeutige ID. Beim Stellen einer neuen Anfrage werden neben ihrer ID auch eine Liste der IDs aller Anfragen des aktuellen Anfragepfades übertragen. Ein Peer, der eine Anfrage bearbeitet, verknüpft die verwendeten Mappings oder gestellten Anfragen mit der eindeutigen ID der Anfrage. Erhält er eine weitere Anfrage, schaut er in der Liste der IDs des Anfragepfades nach, welche Mappings er bereits verwendet hat. Er kann so über einen Abbruch oder das Fortsetzen der Anfragebearbeitung entscheiden.

5.3 Verwendung von Zwischenspeichern

Eine weitere Pruningstrategie, die zur Verkleinerung des Anfragebaumes führt, ist das Verwenden von Zwischenspeichern (Caches). Ein Peer führt dabei eine Anfrage nur einmal aus und merkt sich deren Ergebnis. Soll die gleiche Anfrage noch einmal gestellt werden, wird diese nicht ausgeführt, sondern das bereits ermittelte Ergebnis verwendet. Im Anfrageplan zeigt die zweite Anfrage auf den Ast der ersten Anfrage. Somit kann der komplette Ast unter der zweiten Anfrage entfallen.

In einem statischen PDMS bei dem die Peers und ihre gespeicherten Daten konstant bleiben, liefert eine Anfrage zu jedem Zeitpunkt die gleiche Ergebnismenge. Die Verwendung von Zwischenspeichern verändert somit nicht das Anfrageergebnis. Ein variables PDMS in dem Peers verschwinden, neue hinzukommen und die Daten der Peers Veränderungen unterliegen, entspricht allerdings eher realen Bedingungen. Ein Vorteil von Peer Data Management System ist genau diese Flexibilität. In einem variablen PDMS kann die Verwendung von Zwischenspeichern unvollständige oder gar falsche Ergebnisse liefern. Für diesen Fall ist es sinnvoll, die Lebenszeit eines gespeicherten Anfrageergebnisses zu begrenzen. Die Dauer der Lebenszeit muss dabei an die gewünschte Genauigkeit bzw. Aktualität der Daten und dem geringeren Ressourcenverbrauch durch Nutzung der Zwischenspeicher sowie an die Geschwindigkeit der Änderungen in einer PDMS-Instanz angepasst werden.

Im *System P* sind zur Zeit zwei verschiedene Zwischenspeicher implementiert. Beide Varianten speichern die Ergebnisse von Anfragen, die ein Peer weiteren Peers stellt. Für den Fall, dass der Peer exakt dieselben Anfragen erneut stellen muss, kann er auf die in den Zwischenspeichern abgelegten Ergebnisse zugreifen, anstatt die Anfrage erneut auszuführen. Beide Zwischenspeicher im *System P* unterscheiden sich in der Dauer der Speicherung der Ergebnisse. Die erste Variante, der *Anfrage-Cache*, hält die zwischengespeicherten Ergebnisse von Anfragen in einem Peer nur während dieser Peer eine Anfrage bearbeitet vor. Ist die an ihn gestellte Anfrage beantwortet, werden die zwischengespeicherten Daten verworfen. Zusätzlich kann nur in der gerade laufenden Anfragebearbeitung auf die Daten im Cache zugegriffen werden. Der Anfrage-Cache ist für die im Abschnitt 4.1.2 beschriebene Sonderbehandlung der LaV-Expansion nützlich. Bei der Auflösung der Uncle-Beziehungen im RG-Tree entstehen gleiche Goal-Nodes der fünften Ebene, die zu

gleichen Anfragen führen. Diese Anfragen müssen bei der Verwendung dieses Caches nicht mehrfach ausgeführt werden. Ein Peer, der den Anfrage-Cache nutzen kann, hat dadurch weniger Kinder im Anfragebaum. Durch die geringe Lebenszeit der Daten im Cache ist es unwahrscheinlich, dass bei der Beantwortung einer Anfrage falsche oder veraltete Ergebnisse geliefert werden. Eine Veränderung in der PDMS-Instanz müsste derart erfolgen, dass ein Peer, der mehrfach kurz hintereinander oder parallel die gleiche Anfrage an einen weiteren Peer stellt, unterschiedliche Ergebnisse erhält. Somit verändert diese Art des Caches die Ergebnisse einer Anfrage im Allgemeinen nicht.

Der zweite im *System P* implementierte Cache ist der so genannte *globale Cache*. Er speichert die Ergebnisse einmal gestellter Anfragen eines Peers global sichtbar für alle laufenden und zukünftigen Anfragebearbeitungen dieses Peers. Die Ergebnisse von Anfragen, die zur Beantwortung einer bereits an ein Peer gestellten Anfrage ermittelt wurden, stehen zur Beantwortung weiterer Anfragen, die an diesen Peer gestellt werden, zur Verfügung. Sie können nur genutzt werden, wenn exakt die gleichen Anfragen erneut gestellt werden. Das heißt, die Anfrage muss aus den gleichen Teilzielen und den gleichen Selektionen bestehen. Die Lebenszeit der gespeicherten Anfrageergebnisse ist dabei unbegrenzt. Erhält ein Peer unter Nutzung des globalen Caches eine bereits gestellte Anfrage erneut, so wird diese allein aus den Daten seiner lokalen Quellen und den Einträgen des globalen Caches beantwortet. Der Peer muss keine weiteren Peers anfragen und wird somit zu einem Blatt im Anfragebaum. Die unendlich lange Lebenszeit der zwischengespeicherten Ergebnisse kann dazu führen, dass in einer PDMS-Instanz, in der sich die Daten und die Verfügbarkeit der Peers ändert, falsche Anfrageergebnisse geliefert werden. Dies ist nicht nur dann der Fall, wenn die gleiche Anfrage mehrfach gestellt wird, sondern auch wenn Peers zur Beantwortung einer neuen Anfrage auf bereits ermittelte Ergebnisse von vorherigen Anfragen zurückgreifen. Durch die unendliche Lebenszeit der Cache-Einträge und eine fehlende Verdrängungsstrategie der Implementierung eignet sich diese Art des Zwischenspeichers nicht für den „realen“ Einsatz. Im *System P* ist diese einfache Implementierung enthalten, um zu zeigen, welche Einsparungen sich durch die Verwendung von Zwischenspeichern prinzipiell ergeben. Im Abschnitt 6.5 werden die dazu durchgeführten Experimente näher erläutert.

Die Verwendung von Zwischenspeichern ist eine Pruningstrategie, die die Ausführung der Anfragebearbeitung beschleunigen kann. Ergeben sich während der Anfragebearbeitung keine Änderungen am PDMS, wird das Ergebnis einer Anfrage durch die Nutzung von Zwischenspeichern nicht verändert. Für die Zwischenspeicherung der Anfrageergebnisse ist es nötig, dass ein Peer Speicherplatz zur Verfügung stellt. Auch müssen die Dauer der Zwischenspeicherung und eventuelle Verdrängungsstrategien an die Geschwindigkeit, mit der sich eine PDMS-Instanz ändert, angepasst werden.

5.4 Anfragetiefe

Im Beispiel 5.1 hatte der Anfragebaum eine Tiefe von 21. Zwischen der eigentlichen Anfrage und dem letzten Peer im Anfragepfad wurden somit 20 Peer-Mappings genutzt. Da

Peer-Mappings Projektionen, Selektionen sowie Normalisierungen und Denormalisierungen (Joins) enthalten können, ist die Nutzung eines Peer-Mappings mit einem Verlust an Informationsqualität verbunden. Das Beispiel 5.3 verdeutlicht diesen Fall.

Beispiel 5.3 *Verlust von Informationsqualität durch Benutzung von Peer-Mappings.*

Die Abbildung 5.8 zeigt einen Ausschnitt einer PDMS-Instanz. Auf die Darstellung aller Peers, der lokalen Quellen und lokalen Mappings wird verzichtet.

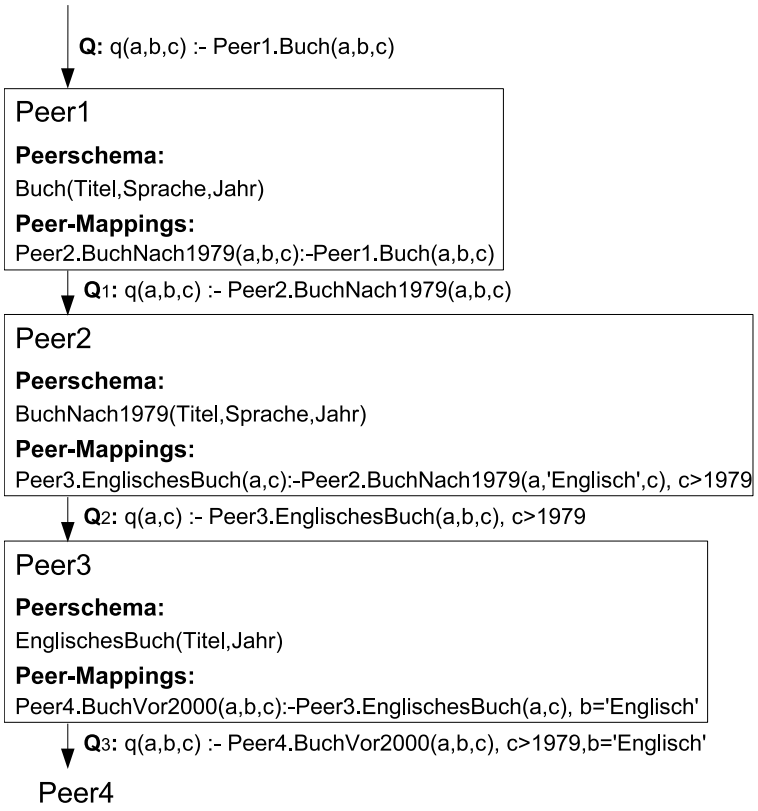

Abbildung 5.8: Ausschnitt einer PDMS-Instanz

*Peer*1 *kann in der Relation „Buch“ Angaben über Titel, Sprache und Erscheinungsdatum von Bücher machen. Die Anfrage Q kann somit als Frage: „Welches Buch wurde in welcher Sprache wann geschrieben?“ interpretiert werden. Peer*1 *nutzt zur Beantwortung dieser Anfrage ein Peer-Mapping, das auf die Relation „BuchNach1979“ im Peer-Schema von Peer*2 *verweist. Peer*2 *kennt in dieser aber nur Informationen über Bücher, die nach dem Jahr 1979 erschienen sind. Peer*1 *fragt Peer*2 *ebenfalls: „Welches Buch wurde in welcher Sprache wann geschrieben?“, erhält aber nur Antworten zu Büchern, die nach 1979 geschrieben wurden. Um die Anfrage* Q_1 *zu beantworten, kann Peer*2 *auf das Peer-Mapping zu Peer*3 *zugreifen. Die Relation „EnglischesBuch“ von Peer*3 *liefert jedoch nur englische Bücher und deren Erscheinungsdatum. Da Peer*2 *in seinem Schema nur Bücher*

anbietet, die nach 1979 erschienen sind, muss er Peer3 diese Selektion in einer Anfrage mitgeben. Die Anfrage Q_2 von Peer2 an Peer3 lautet daher: „Welches Buch wurde nach 1979 in welcher Sprache wann geschrieben?“. Die Antworten auf diese Frage können jedoch nur englische Bücher sein, die nach 1979 geschrieben wurden. In der Abbildung 5.8 ist im Peer3 ein Peer-Mapping dargestellt, das dieser zur Beantwortung der Anfrage Q_2 nutzen kann. Es verweist auf die Relation „BuchVor1980“ eines Peers Peer4. Dieser Peer kennt wiederum nur Bücher, die vor dem Jahr 1980 erschienen sind. Sollte Peer3 dieses Mapping verwenden stellt er die Anfrage Q_3 an Peer4. Er fragt Peer4, der nur Informationen über Bücher vor dem Jahr 1980 kennt, nach Büchern, die nach dem Jahr 1979 erschienen sind. Zusätzlich muss Peer3 in der Anfrage die Sprache der Bücher auf Englisch einschränken, da er selber nur englische Bücher über sein Peer-Schema anbietet.

Das Ergebnis der ursprünglichen Anfrage Q setzt sich neben Antworten auf die ursprüngliche Anfrage auch aus den Antworten der anderen Anfragen zusammen. Dabei wird die ursprüngliche Anfrage durch die Kumulation der Selektionen zunehmend eingeschränkt. Im Beispiel macht Peer3 in seinem Peerschema keine Einschränkungen, in welchem Jahr ein Buch erschienen ist. Jedoch enthält die an ihn gestellte Anfrage diese Einschränkung, da der Anfragepfad Peer2 enthält, welcher nur Informationen über Bücher nach 1979 anbietet. Diese Einschränkung wird an Peer4 weitergeben. Er kennt jedoch nur Bücher, die vor 1980 erschienen sind. Das Ergebnis der Anfrage Q_3 ist somit leer.

Das Beispiel verdeutlicht, wie die Fragestellung und damit auch die mögliche Größe der Ergebnismenge durch die Nutzung mehrerer geschachtelter Peer-Mappings eingeschränkt wird. Es kann sinnvoll sein, die Tiefe des Anfragebaumes und damit die maximale Anzahl hintereinander genutzter Peer-Mappings zu begrenzen. Der Aufwand der Bearbeitung einer Anfrage wird dadurch reduziert. Die Ergebnismenge ist jedoch nicht immer vollständig. Die Hoffnung ist, dass nur wenige Tupel mit geringem Informationsgehalt fehlen. Auswertungen von Experimenten zur Ermittlung des Informationsverlustes durch Begrenzung der Anfragetiefe sind in Abschnitt 6.6 beschrieben.

Durch die Begrenzung der Anfragetiefe ist auch in zyklischen PDMS-Instanzen sichergestellt, dass die Anfragebearbeitung terminiert. Es kann somit auf die im Abschnitt 5.2 beschriebene Zyklenvermeidung verzichtet werden. Somit ist es in einem zyklischen PDMS denkbar, dass die Begrenzung der Anfragetiefe auf einen festen Wert mehr Ergebnistupel liefert, als die Zyklenvermeidung. Während die Zyklenvermeidung den Anfrageplan aufgrund mehrfach genutzter Mappings oder mehrfach gestellter Anfragen in der Tiefe X abbrechen würde, könnte eine Begrenzung der Anfragetiefe auf $X + 1$ ohne Zyklenvermeidung mehr Ergebnistupel liefern. Das bereits erwähnte Beispiel 5.2 verdeutlicht diesen Fall. Die Zyklenvermeidung bricht mit der Anfragetiefe vier ab, eine Begrenzung der Anfragetiefe auf fünf würde jedoch zwei Ergebnistupel mehr liefern.

Die Begrenzung der Anfragetiefe ist eine Pruningstrategie, die in der Regel zu einer unvollständigen Ergebnismenge führt. Dabei hängt der auftretende Informationsverlust von der erlaubten Tiefe des Anfragebaumes und den verwendeten Peer-Mappings ab. Je geringer

die erlaubte Tiefe ist, desto wahrscheinlicher ist es, dass mehr Tupel im Anfrageergebnis fehlen. Enthalten die Peer-Mappings jedoch viele Selektionen, die sich wie im obigen Beispiel auch noch widersprechen, so kann der Informationsverlust durch eine Begrenzung der Anfragetiefe auch gering ausfallen.

Wie bei der Zyklenerkennung ist es bei dieser Pruningstrategie notwendig, neben der eigentlichen Anfrage weitere Informationen zu übertragen. Ein Peer muss wissen, in welcher Tiefe des Anfragebaumes sich die aktuelle Anfrage befindet und welches die maximal zulässige Tiefe ist. Nur so kann er entscheiden, ob die Anfragebearbeitung abgebrochen oder fortgesetzt werden soll.

Ein Nachteil dieser Methode ist, dass die eingesparten Kosten nicht vor einer Anfrage ermittelt werden können. Es werden alle Peer-Mappings bis zu einer angegebenen Tiefe des Anfragebaumes verfolgt. Dessen Struktur ist für einen Nutzer in der Regel jedoch unbekannt, so dass schwer abzuschätzen ist, auf welche Tiefe eine Anfrage begrenzt werden soll. Ein zusätzliches Problem dieser Methode ist, dass sämtliche Mappings gleich behandelt und alle Pfade bis zur angegebenen Tiefe verfolgt werden. Die nächste Pruningstrategie versucht hingegen nur „gute“ Pfade zu nutzen.

5.5 Qualitätsbasierte Pruningstrategien

Eine Voraussetzung für qualitätsbasierte Pruningstrategien ist ein Qualitätsmaß für Mappings. Als Qualitätsmaße bieten sich die erwartete Kardinalität der Ergebnismenge einer Anfrage bei Nutzung dieses Mappings, die Informationsqualität zum Beispiel die Dichte[8] der Tupel der Ergebnismenge, Antwortzeiten und Verfügbarkeit der Quelle hinter diesem Mapping sowie eine Kombination aller Eigenschaften an. Die Qualität des Mappings und seiner Quelle können Erfahrungen aus bereits gestellten Anfragen sein oder durch gezieltes Sampling gewonnen werden. Zur Erleichterung sollte jedes Qualitätsmaß eines Mappings auf Werte zwischen 0 und 1 normiert werden.

Die Qualität eines Mappings kann ein Peer bei der Mapping-Auswahl zur Beantwortung einer Anfrage nutzen. Eine Strategie könnte sein, nur qualitativ hochwertige Mappings oder nur Mappings zu nutzen, die ein gewisses Maß an Qualität nicht unterschreiten. Beide Varianten sind jedoch recht unflexibel und könnten auch durch das Löschen qualitativ minderwertiger Mappings erreicht werden. Interessanter ist der Ansatz, die Qualitäten der in einem Anfragepfad beteiligten Mappings insgesamt zu betrachten. Wie der im Beispiel 5.3 beschriebene Informationsverlust mit steigender Zahl der benutzten Mappings zunimmt, nimmt die Gesamtqualität des Anfragepfades ab. Als Qualität eines Anfragepfades wird die zusammengesetzte Qualität aller beteiligten Peer-Mappings dieses Pfades definiert. Dabei ist zu beachten, dass durch die Verwendung eines weiteren Mappings, die Qualität eines Anfragepfades nicht steigen kann. Bei einem „perfekten“ Mapping bleibt sie gleich, ansonsten nimmt sie entsprechend der Qualität des Mappings ab.

[8] Verhältnis von Null-Werten zu Nicht-Null-Werten im Anfrageergebnis

Die Idee ist nun, dass ein Benutzer neben der eigentlichen Anfrage eine untere Qualitätsschranke angibt[9], die ein Anfragepfad nicht unterschreiten darf. Ein Peer nutzt dann zur Beantwortung einer Anfrage nur die Mappings, deren Qualität zusammen mit der Qualität des Anfragepfades zu diesem Peer diese untere Qualitätsgrenze nicht unterschreitet. Das Ergebnis ist ein Anfrageplan der nur Anfragepfade enthält, die mindestens die angegebene Qualität besitzen. Die Hoffnung ist, dass trotz Pruning die Qualität des Anfrageergebnisses entsprechend hoch ausfällt. Sie hängt von der Genauigkeit der Qualitätskriterien der Mappings sowie von der eingestellten unteren Grenze ab. Dabei gilt, je niedriger diese Grenze gesetzt wird, desto vollständiger ist das Anfrageergebnis und umso geringer sind die eingesparten Kosten gegenüber der vollständigen Anfrage.

Um diese Pruningstrategie nutzen zu können, muss die Qualität der Mappings zuvor ermittelt werden. Die bedeutet einen zusätzlichen Aufwand. Ähnlich wie bei der Begrenzung der Anfragetiefe muss neben der Anfrage die untere Qualitätsgrenze und die Qualität des Anfragepfades an den aufgerufenen Peer übergeben werden.

Diese Methode lässt sich gegenüber der einfachen Begrenzung der Anfragetiefe viel feiner durch den Nutzer steuern. Sind bei der maximalen Anfragetiefe nur ganze Werte zwischen 0 und der maximalen Anfragetiefe[10] zulässig, können bei dieser Pruningstrategie reelle Werte im Bereich zwischen 0 und 1 verwendet werden. Nachteil dieser Methode ist ebenfalls, dass die Kosten, die eine Anfrage verursachen darf, nicht vom Nutzer vorgegeben werden können. Im Extremfall kann selbst die Angabe einer unteren Grenze von 100 Prozent zu einer vollständigen Anfrage führen, wenn alle Mappings des vollständigen Anfragebaumes eine optimale Qualität besitzen.

5.6 Budgetbasierte Pruningstrategien

Eine weitere Gruppe von Pruningstrategien bilden die budgetbasierten Strategien. Dabei erhält ein Peer zur Beantwortung einer Anfrage ein virtuelles Budget. Kosten, die mit dem Budget zu bezahlen sind, können dabei durch die Benutzung der CPU oder von Netzwerkressourcen entstehen oder Zeitverbrauch im Allgemeinen sein. Der anfragende Nutzer entscheidet[11], welches Budget er für die Beantwortung seiner Anfrage bereit ist auszugeben. Peers, die an der Bearbeitung der Anfrage beteiligt sind, können frei über ihr Budget entscheiden und dieses bei einer Anfragebearbeitung, nach Abzug der Kosten, auf die erzeugten Peer-Anfragen aufteilen. Sollte ein Teil des Budget übrig bleiben, kann dieses dem Aufrufer zurückgeben werden. Die Anfragebearbeitung terminiert, sobald das gesamte Budget aufgebraucht ist oder nicht weiter verwendet werden kann. Das verfügbare Budget eines Peers sollte so verteilt werden, dass das Ergebnis einer Anfrage möglichst

[9] Diese Schranke kann auch von System vorgegeben werden.

[10] Welche in den meisten Fällen unbekannt sein dürfte, aber praktische Werte zwischen 10 und 30 liegen.

[11] Das PDMS kann ihn bei seiner Entscheidung mit Hilfe von Erfahrungswerten unterstützen.

vollständig ist und die gelieferten Tupel möglichst „dicht“[12] sind. Dafür können die im vorherigen Abschnitt beschriebenen Qualitätsmerkmale genutzt werden.

Zwei mögliche budgetbasierte Strategien werden detailliert in [8] beschrieben und sind im Anfrageplaner des Simulators und somit auch im *System P* verfügbar. Das Kostenmodell beider Strategien sieht vor, dass die Nutzung eines Peer-Mappings eine Budget-Einheit kostet. Die erste Strategie, „Greedy“ genannt, wählt nur das qualitativ beste Peer-Mapping aus. Die durch dieses Peer-Mapping umformulierte Anfrage wird an den Peer (dieses Peer-Mappings) gestellt. Dabei wird ihm, nach Abzug der Kosten, das komplette Budget übergeben. Diese Strategie erzeugt einen eher tiefen Anfragebaum. Die zweite Strategie hingegen verteilt das gegebene Budget auf alle vorhandenen Peer-Mappings, gewichtet nach ihrer Qualität. Es wird dadurch ein breiter Anfragebaum erzeugt.

Der große Vorteil dieser Methode ist die Möglichkeit, dass die Kosten, die eine Anfragebearbeitung erzeugen darf, unabhängig von der Struktur der aktuellen PDMS-Instanz, festgelegt werden können. Außerdem kann auf eine Zyklenvermeidung verzichtet werden, da die Anfragebearbeitung bei endlichem Budget sicher terminiert. Für einen Nutzer ist es ohne weitere Unterstützung durch das PDMS schwierig, vor einer Anfrage abzuschätzen, wie viel Budget notwendig ist, damit seine Anfrage zufriedenstellend beantwortet wird. Gibt er zu viel Budget aus, benötigt die Anfragebearbeitung mehr Ressourcen als nötig. Gibt er zu wenig Budget an und ist somit die Ergebnismenge unzureichend, muss er die Anfrage mit einem größeren Budget wiederholen. Das Budget der ersten Anfrage ist verloren.

5.7 Vergleich der Pruningstrategien

Um die angegebenen Pruningstrategien vergleichen zu können, sollen die möglichen Einsparungen bei der Anfragebearbeitung, die erwartete Vollständigkeit des Anfrageergebnisses, der Aufwand der Pruningstrategie während der Anfragebearbeitung sowie deren Implementierung gegenübergestellt werden. Die Tabelle 5.1 fasst die jeweiligen Vor- und Nachteile der Pruningstrategien zusammen.

Zu erkennen ist, dass es möglichst hohe Einsparungen bei einer hohen Informationsqualität nicht umsonst gibt. Die Zwischenspeicher liefern in der Regel ein vollständiges Ergebnis und bieten große Einsparungen[13]. Der Aufwand für die Zwischenspeicherung der Anfrageergebnisse ist jedoch sehr hoch. Ebenso muss eine Strategie zur Verdrängung von veralteten oder nicht mehr benötigten Zwischenergebnissen implementiert werden. Insgesamt ist der Aufwand einer ausgereiften Implementierung ebenfalls sehr hoch[14]. Die Begrenzung der Anfragetiefe ist leicht zu implementieren und verursacht wenig Aufwand während der Anfragebearbeitung. Dafür bietet diese Strategie wenig Möglichkeiten, die Anfragebearbeitung durch einen Nutzer zu beeinflussen. Aufwendiger sind die qualitätsbasierten

[12] Möglichst ohne viele Null-Werte

[13] Vergleiche Ergebnisse aus den Experimenten in Abschnitt 6.5.

[14] Die Zwischenspeicher im *System P* nutzen keine Größenbeschränkung oder Verdrängungstrategien.

und budgetbasierten Ansätze zu implementieren. Sie lassen dafür feinere Einstellungen zu. Bei den bugdetbasierten Strategien lassen sich die Anfragekosten exakt festlegen. Die Algorithmen zur Verteilung des Budgets bringen eine zusätzliche Komplexität in die Anfragebearbeitung und erfordern ebenfalls eine gewisse Einschränkung an die Autonomie der Peers. Die beteiligten System müssen sich auf ein Protokoll zur Verwaltung des Budgets einigen.

Methode	Kostenersparnis	Vollständigkeit	Aufwand
Zwischenspeicher	Abhängig von der Struktur des PDMS und den gestellten Anfragen sowie der Dauer der Speicherung von Zwischenergebnissen.	Abhängig von der Geschwindigkeit von Änderungen in der Struktur und der Daten eines PDMS und der Länge der Speicherung von Zwischenergebnissen. In der Regel ist das Ergebnis vollständig.	Zwischenspeicher benötigt Ressourcen in Form von RAM oder Festplattenkapazität. Zusätzliche Logik für Verdrängungsstrategie nötig. Hoher Aufwand.
Begrenzung der Anfragetiefe	Abhängig von der Struktur (Tiefe, Verzweigung, Balance) des vollständigen Anfragebaumes, vor einer Anfrage nicht bekannt.	Vor einer Anfrage nicht abzuschätzen. Stark abhängig von der Struktur des vollständigen Anfragebaumes.	Neben der Anfrage sind die maximale Tiefe und die aktuelle Anfragetiefe nötig. Sehr geringer Aufwand.
Qualitätsbasiert	Abhängig vom gewählten Qualitätsmaß und den Qualitäten der Mappings, nicht vor einer Anfrage bekannt.	Abhängig vom Qualitätsmaß der Peer-Mappings. Bei geschätzten Qualitätsmaßen kann die Vollständigkeit stark schwanken. Vor einer Anfrage nicht bekannt.	Die Qualitätsmaße der Mappings müssen ermittelt werden (Erfahrung oder Sampling). Eine Qualitätsgrenze und die Qualität des aktuellen Anfragepfades müssen neben der Anfrage mitgegeben werden. Mittlerer Aufwand.
Budgetbasiert	Maximale Kosten können exakt festgelegt werden.	Ähnlich dem qualitätsbasierten Pruning, Kosten werden jedoch effektiver genutzt. Trotzdem bleibt die Schwierigkeit, das Budget richtig zu dosieren.	Wie beim qualitätsbasierten Pruning. Zusätzlich muss ein Algorithmus das vorhandene Budget aufteilen. Mittlerer Aufwand.

Tabelle 5.1: Vergleich der Pruningstrategien

5.8 Verbesserungsmöglichkeiten

Bei den beschriebenen und implementierten Pruningstrategien sind Verbesserungen denkbar. Die im *System P* verwendeten Zwischenspeicher kennen keine Verdrängungsstrategien. Es wird angenommen, dass genügend Speicher zur Verfügung steht. Dies führt beim

globalen Cache unweigerlich zu Problemen, sollten eine PDMS-Instanz mehrere verschiedene Anfragen gestellt werden. Auch liefern die Zwischenspeicher nur Ergebnisse für exakt die gleichen Anfragen zurück. Unterscheidet sich die Anfrage in Selektionen oder Projektionen von der zwischengespeicherten Anfrage, so werden das zwischengespeicherte Ergebnis nicht verwendet. Eine verbesserte Implementierung könnte auf diese Ergebnisse zurückgreifen und anschließend Selektionen und Projektionen ausführen.

Eine deutliche Steigerung der Effizienz der Pruningstrategien, dem Quotienten aus Qualität (Vollständigkeit) des Anfrageergebnisses und den benötigten Kosten für die Anfragebearbeitung, kann womöglich durch eine Kombination aller beschriebenen Strategien erzielt werden. Die Vorteile aller Strategien können so gemeinsam genutzt werden. Die Verwendung der Zwischenspeicher bringt eine Einsparung der Kosten, ohne das Ergebnis zu beeinflussen, durch eine Begrenzung der maximalen Anfragetiefe kann auf eine Zyklenvermeidung verzichtet werden, durch die Verwendung von qualitätsbasierten Strategien werden nur viel versprechende Anfragepfade genutzt und durch die Verwendung von budgetbasierten Strategien können die maximalen Kosten exakt festgelegt werden. Mit der Implementierung des *System P* können Zwischenspeicher und die Begrenzung der Anfragetiefe parallel zu einem qualitäts- oder budgetbasierten Ansatz genutzt werden.

6 Experimente

Neben den theoretischen Überlegungen zu Pruningstrategien und den Beschreibungen zur Implementierung ist vor allem deren praktischer Einsatz interessant. Zu diesem Zweck wurden automatisierte Experimente mit der Implementierung des *System P* durchgeführt[1]. In diesem Kapitel werden die Auswirkungen von Zwischenspeicher, der Begrenzung der Anfragetiefe und die Nutzung eines qualitätsbasierten Pruningansatzes untersucht. Abschließend wird aufgezeigt, wie sich die Anfragebearbeitung des *System P* bei der Verteilung auf mehrere Rechner verhält.

Bevor die Beschreibung der Experimente erfolgt, sollen zunächst die zur Anfragebearbeitung möglichen Einstellungen und die neben dem Anfrageergebnis gelieferten Informationen erläutert werden.

6.1 Parameter der Anfragebearbeitung

Für Experimente mit dem *System P* stehen eine Reihe von veränderbaren Parametern zur Verfügung. Zur Erzeugung einer PDMS-Instanz bietet der in [6] beschriebene PDMS-Generator unter anderem die Möglichkeit, die Anzahl der Peers, die Art des erzeugten Graphen[2], die Häufigkeit von Selektionen und Projektionen sowie die Größe der Extensionen der lokalen Quellen anzugeben. Daneben kann auch die Anfragebearbeitung durch verschiedene Einstellungen beeinflusst werden. Diese Einstellungen werden parallel zur eigentlichen Anfrage an einen Peer mitgegeben. Der angefragte Peer beachtete diese Einstellungen und gibt sie, sollte er zur Bearbeitung der Anfrage weitere Peers anfragen, an diese weiter. Die Parameter einer Anfrage werden dadurch global für alle an der Anfrage beteiligten Peers gesetzt. Die Abbildung 6.1 zeigt den Dialog im *System P* mit den möglichen Einstellungen zur Anfragebearbeitung. Für automatisierte Experimente können diese Einstellungen der Anfragebearbeitung in einer Instanz der Klasse *QuerySettingsDTO* übergeben werden.

Neben der eigentlichen Anfrage lassen sich in diesem Dialog weitere Einstellungen zur Anfragebearbeitung tätigen. So kann ausgewählt werden, ob ein kompletter Anfrageplan erzeugt werden soll, indem die lokalen Anfragepläne neben der Ergebnismenge übertragen werden. Diese Übertragung kann die Anfragebearbeitungsgeschwindigkeit senken und ist daher optional. Das Ergebnis ist ein kompletter Anfrageplan im Monitor Peer, wie er in Abbildung 4.5 dargestellt ist.

Die zweite Option im Dialog bietet die Möglichkeit, die Anfragebearbeitung zu visualisieren. Ist sie aktiviert, geben die PDMS-Peers während der Ausführung der Anfrage

[1] Die automatisierten Experimente sind als JUnit-Testfall implementiert und können jederzeit wiederholt werden.

[2] Gemeint ist der Graph, den eine PDMS-Instanz bildet, in dem Peers als Knoten und Peer-Mappings als Kanten dargestellt werden.

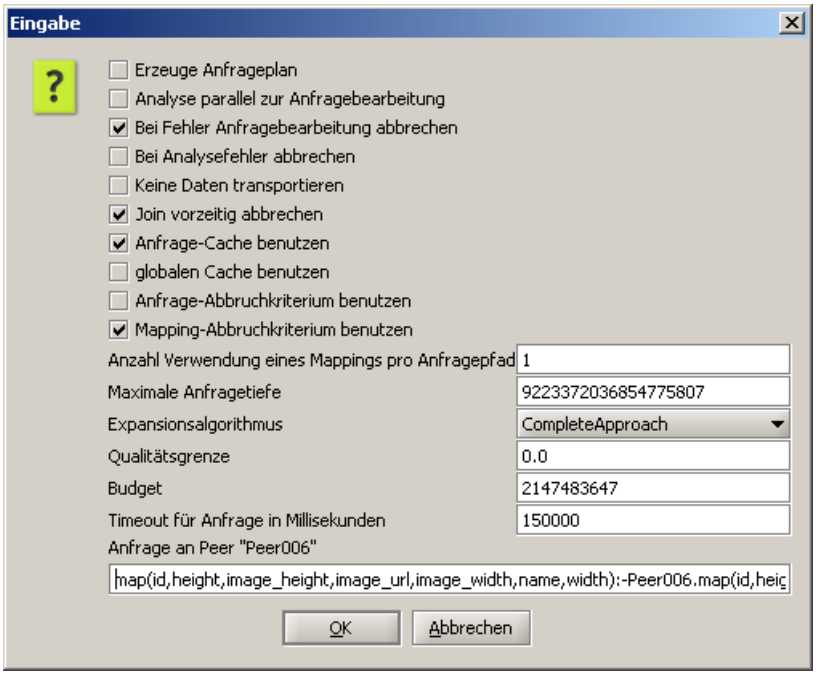

Abbildung 6.1: Anfragedialog im *System P*

Rückmeldungen an den Monitor-Peer. Diese Rückmeldungen beinhalten die Information, welcher Peer gerade welchen anderen Peer anfragt und welcher Peer bereits geantwortet hat. Dadurch wird die in Abbildung 4.4 dargestellte *Live*-Visualisierung der Anfragebearbeitung möglich. Das Übertragen der Rückmeldungen verlangsamt die Anfragebearbeitung, so dass diese Einstellung ebenfalls optional ist.

Während der Anfrageausführung können Fehler auftreten. So kann es, aufgrund von Störungen im Netzwerk dazu kommen, dass Peers nicht miteinander kommunizieren können oder Übertragungen abbrechen. Es ist auch denkbar, dass ein Peer zu wenig Speicher hat, um etwa einen Join lokal auszuführen. Mit der Einstellung „Bei Fehler Anfragebearbeitung abbrechen“ kann daher festgelegt werden, ob die Anfragebearbeitung trotz Fehler fortgesetzt oder abgebrochen werden soll. Im Falle eines Abbruchs wird kein Anfrageergebnis geliefert, auch wenn schon Teile berechnet wurden. Die Einstellung „Bei Analysefehler abbrechen“ legt fest, ob Fehler, die beim Übertragen der oben beschriebenen Rückmeldungen an den Monitor-Peer auftreten, ebenfalls zum Abbruch der Anfragebearbeitung führen sollen.

Mit der Option „Keine Daten transportieren“ ist es möglich, die Übertragung von Ergebnistupeln zu deaktivieren. Die Anfragebearbeitung wird dadurch wesentlich beschleunigt, da neben der Übertragung auch die Notwendigkeit der Ausführung von Join- und Union-Operationen entfällt. Diese Option kann gewählt werden, wenn nicht das konkrete Ergebnis der Anfrage, wohl aber die Größe des Anfragebaumes im Vordergrund der Experimente steht.

Mit der Einstellung „Join vorzeitig abbrechen“ wird bestimmt, ob die Daten für alle an einem Join beteiligten Relationen gesammelt werden, wenn bereits feststeht, dass das Ergebnis eines Joins leer ist, da eine beteiligte Relation keine Tupel enthält. Das folgende Beispiel verdeutlicht diesen Fall. *Peer1* soll den Join über den Relationen *Buch* von *Peer2* und *Autor* von *Peer3* bilden. Dabei liefert die Anfrage von *Peer1* an *Peer2* nach der Relation *Buch* keine Daten. Somit muss *Peer1* die Anfrage nach den Autoren nicht mehr an *Peer3* stellen, da das Ergebnis des Joins, die leere Menge, bereits feststeht. Das vorzeitige Abbrechen ist ebenfalls eine Pruningstrategie, da nicht der komplette Anfragebaum aufgebaut wird. Um den vollständigen Anfragebaum zu erzeugen, welcher unabhängig von den Daten der einzelnen Peers jederzeit ein vollständiges Ergebnis liefern soll, darf diese Einstellung nicht aktiviert sein.

Die folgenden Optionen bestimmen die verwendeten Pruningstrategien für die Anfrage. So lassen sich die im Abschnitt 5.3 beschriebenen beiden Arten von Zwischenspeichern aktivieren. Auch welche Art der Zyklenvermeidung (siehe Abschnitt 5.2) verwendet werden soll, lässt sich angeben. Wird das Mapping-Abbruchkriterium gewählt, kann die maximal zulässige Anzahl von Verwendungen eines Peer-Mappings je Anfragepfad eingeschränkt werden. Für eine Begrenzung der Anfragetiefe, welche im Abschnitt 5.4 erläutert ist, kann eine maximal zulässige Tiefe angeben werden. Die Standardeinstellung ist *Long.MAX_ VALUE*[3]. Dieser Wert ist so groß, dass er praktisch nie erreicht wird und somit für eine unbegrenzte Anfragetiefe steht.

Zusätzlich lässt sich der gewählte Expansionsalgorithmus zur Erzeugung der lokalen RG-Trees angeben. Es stehen der vollständige Ansatz (*CompleteApproach*), zwei budgetbasierte Ansätze und ein qualitätsbasierter Ansatz (*QualityApproach*) zur Verfügung. Der vollständige Ansatz verwendet zum Aufbau des RG-Trees jedes nutzbare Mapping. Die budgetbasierten Ansätze, welche im Abschnitt 5.6 beschrieben sind, verwenden ein einstellbares Budget, welches auf ausgewählte Peer-Mappings verteilt wird. Der qualitätsbasierte Ansatz nutzt Angaben zur Qualität von Mappings um zu entscheiden, welche Mappings Verwendung finden. Dazu lässt sich eine Qualitätsgrenze zwischen 0 und 1 setzen, die nicht unterschritten werden darf. Die genaue Funktionsweise dieser Strategie ist im Abschnitt 5.5 erklärt.

Der Dialog zur Anfrage an einen Peer bietet die Möglichkeit die im Kapitel 5 beschriebenen Pruningstrategien für eine Anfragebearbeitung im *System P* zu aktivieren und auch deren Kombination zu nutzen. So können Zwischenspeicher, eine Begrenzung der maximalen Anfragetiefe und ein beliebiger Expansionsalgorithmus gleichzeitig verwendet werden.

[3] Das entspricht in Java einem Wert von 9.223.372.036.854.775.807

6.2 Informationen zur Anfragebearbeitung

Nach erfolgter Anfragebearbeitung liefert das *System P* zu jeder Anfrage neben dem eigentlichen Anfrageergebnis weitere Informationen zurück. Diese werden zum Teil in den nachfolgenden Experimenten zur Auswertung der Pruningstrategien genutzt. Die folgende Liste fasst die wichtigsten Informationen zusammen.

- Liste aller Tupel des Anfrageergebnisses
- Informationen ob und welche Fehler bei der Anfragebearbeitung aufgetreten sind.
- Ausführungszeit - Liefert die Dauer der gesamten Anfragebearbeitung in Millisekunden.
- Berechnungszeit - Liefert die Ausführungszeit abzüglich der Zeit, die für Netzwerkübertragungen benötigt wurde.
- Operatorzeit - Liefert die Zeit in Millisekunden, die für Ausführung von Join- und Union-Operationen benötigt wurde. Die Operatorzeit ist in der Berechnungszeit enthalten.
- Anzahl entdeckter Kreise
- Anzahl der Tupel der Ergebnismenge
- Anzahl der Null-Werte in der Ergebnismenge
- Anzahl der Nicht-Null-Werte in der Ergebnismenge
- Anzahl der insgesamt übertragenen Tupel
- Anzahl der Cache-Treffer
- Anzahl der Join-Operationen
- Anzahl der Union-Operationen
- Anzahl der benutzten lokalen Mappings
- Anzahl der benutzten Peer-Mappings
- Anzahl der ausgeführten lokalen Anfragen
- Anzahl der ausgeführten Peer-Anfragen
- Anfragetiefe

Zur Analyse der Pruningstrategien werden die Vollständigkeit der Ergebnismenge und verursachten Kosten betrachtet. Das Kostenmodell soll im Folgenden die Anzahl der ausgeführten Peer-Anfragen sein. Jede Anfrage, die während der Beantwortung einer Anfrage von einem Peer an einen weiteren gestellt wird, verursacht eine „Kosteneinheit“. In Abbildung 6.2 ist beispielhaft eine Anfrage und deren verursachte Kosten dargestellt.

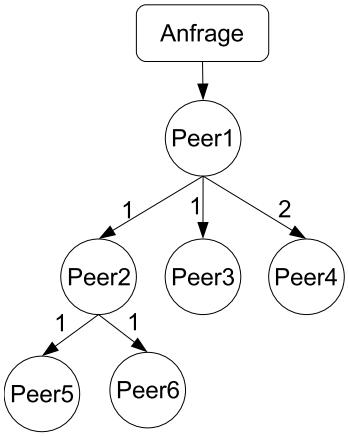

Abbildung 6.2: Beispiel für Kostenmodell

Zur Beantwortung einer Anfrage muss *Peer1* insgesamt drei weitere Peers fragen. Dabei stellt er *Peer2* und *Peer3* jeweils eine Anfrage und *Peer4* zwei Anfragen. *Peer2* muss seinerseits *Peer5* und *Peer6* jeweils einmal anfragen. Peer3, Peer4 und Peer5 beantworten die Anfrage ohne weitere Peeranfragen. Die verursachten Kosten dieser Anfrage betragen somit sechs Peeranfragen.

Neben den verursachten Kosten, ist die Vollständigkeit des Anfrageergebnisses wichtig für die Analyse von Pruningstrategien. Sie wird im nächsten Abschnitt genauer beschrieben.

Für weitere Untersuchungen sind andere Kostenmodelle und Qualitätskriterien denkbar. So könnten die Anzahl der insgesamt durch eine Anfrage übertragenen Tupel oder die Menge der Join- und Union-Operationen oder die Zahl der verwendeten Mappings als Kosten betrachtet werden. Als zusätzliches Qualitätskriterium bietet sich die Bearbeitungszeit einer Anfrage an. Das *System P* liefert diese Werte, so dass sie in weitere Analysen einbezogen werden können.

6.3 Vollständigkeit von Anfrageergebnissen

Wie bereits im Abschnitt 5.2 zur Zyklenvermeidung beschrieben, kann in einem PDMS mit zyklischen Mappings (zyklisches PDMS), kein allgemeines Abbruchkriterium angegeben werden, für das gilt, dass jedes Anfrageergebnis trotz Abbruch vollständig ist. Da im *System P* jede Anfragebearbeitung terminieren soll, ist die Angabe eines allgemeinen Abbruchkriteriums notwendig. Es wird daher für die folgenden Experimente das Ergebnis

des vollständigen Expansionsalgorithmus (CompleteApproach) mit aktiviertem Mapping-Abbruchkriterium[4] als *vollständiges Ergebnis* definiert. Diese Einschränkung liefert in einem PDMS ohne zyklische Mappings alle Ergebnistupel zu einer Anfrage. In einem PDMS mit zyklischen Mappings kann, wie das Beispiel 5.2 zeigt, dieses Ergebnis nicht alle möglichen Ergebnistupel enthalten.

6.4 PDMS-Typen

Für die nachfolgenden Experimente werden mit dem PDMS-Generator aus [6] eine Reihe von PDMS-Instanzen unterschiedlichen Typs erzeugt. Die Abbildung 6.3 zeigt die drei verwendeten PDMS-Typen.

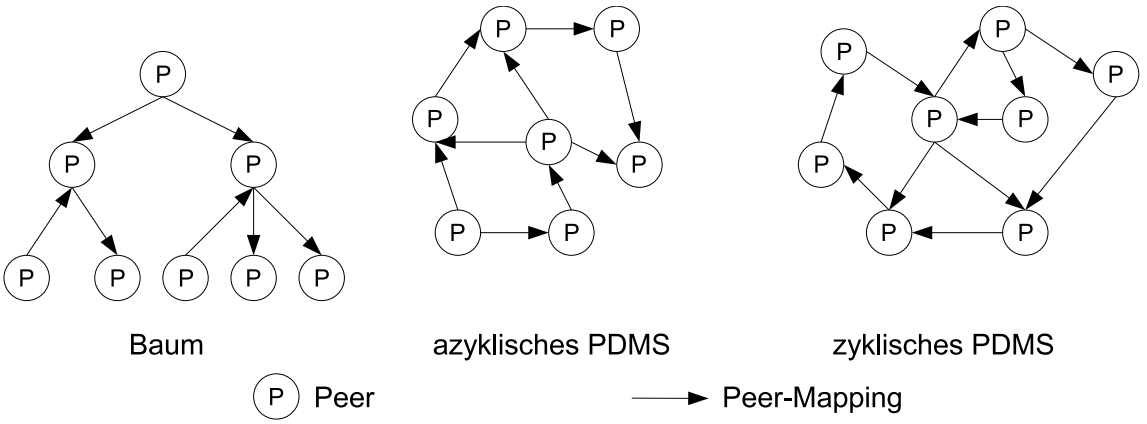

Abbildung 6.3: Erzeugte PDMS-Typen

Um den Typ einer PDMS-Instanz anzugeben, erfolgt die Betrachtung als Graph. Die Peers bilden dabei Knoten und die Peer-Mappings gerichtete Kanten. Der Typ *Baum* ist komplett kreisfrei die Richtung der Mappings ist zufällig. Ein *azyklisches PDMS* kann ungerichtete aber keine gerichteten Kreise enthalten, der Typ *zyklisches PDMS* darf gerichtete Kreise bilden. Alle drei Typen bestehen jeweils aus einer Komponente. Bei den ersten beiden Typen können bei der Anfragebearbeitung keine unendlich langen Anfragepfade entstehen. Die im Abschnitt 5.2 beschriebenen Strategien zur Zyklenvermeidung müssen daher nur beim Typ *zyklisches PDMS* angewendet werden.

Anwendungsbeispiele für den Graphentyp *Baum* sind hierarchisch organisierte Peer Data Management Systeme, welche zum Beispiel in Krisensituationen eingesetzt werden können. Die Blätter können Krankenhäuser, Polizei- oder Feuerwehrstationen symbolisieren, die Ebene darüber kann regionalen Einsatzstellen entsprechen und die Wurzel des Baumes bildet ein nationales Krisenzentrum. Die Knoten der beiden anderen PDMS-Typen können zum Beispiel Abteilungen eines Unternehmens repräsentieren. Bei gleichberechtigten Peers, die ähnliche Daten verwalten, kann die Verknüpfung der Peers mittels Peer-

[4] Jedes Mapping darf in jedem Pfad des Anfragebaums nur ein Mal verwendet werden.

Mappings wesentlich ausgeprägter ausfallen. Das Entstehen von gerichteten Mapping-Kreisen kann in der Praxis dabei schwer verhindert werden. Mappings werden paarweise zwischen autonomen Peers angelegt. Die Überprüfung, ob durch ein neues Mapping gerichtete Kreise entstehen, könnte nur eine „allwissende“ zentrale Instanz durchführen. Solch eine Instanz sollte es in einem PDMS in der Regel nicht geben. Somit dient der *azyklische PDMS-Typ* eher theoretischen Betrachtungen.

Bei der Erzeugung der PDMS-Instanzen mit Hilfe des PDMS-Generators wurde das Datenbank-Schema aus Abbildung 6.4 als globales Schema genutzt. Dieses Schema, das nur aus drei Relationen besteht, wird im PDMS-Generator durch Normalisierung, Denormalisierung und Projektionen in mehrere ähnliche Schemata zerlegt. Jeder Peer einer PDMS-Instanz erhält somit ein eigenes individuelles Schema.

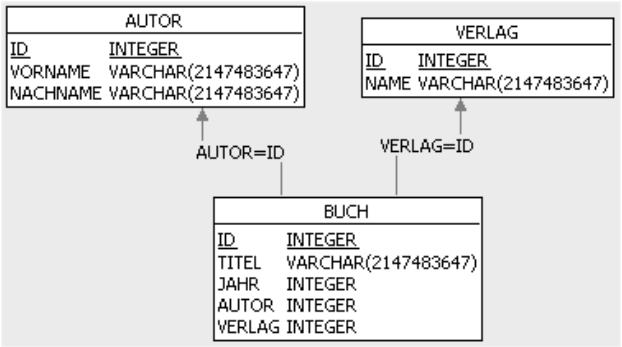

Abbildung 6.4: Einfaches relationales DB-Schema für Experimente

6.5 Auswirkungen von Zwischenspeichern

Mit dem ersten Experiment werden die Auswirkungen der im Abschnitt 5.3 beschriebenen Zwischenspeicher untersucht. Zu diesem Zweck wurden von jedem PDMS-Typ jeweils 50 Instanzen mit 5, 10, 15, 20 und 25 Peers erzeugt. Der Maximalgrad der *azyklischen* und *zyklischen Typen* wurde auf 6 gesetzt. Beim Typ *Baum* wurde eine Ordnung zwischen 1 und 5 gewählt, so dass der Grad der Peers ebenfalls zwischen 1 und 6 liegt. Der Grad im PDMS-Generator bezieht sich dabei nur auf die Kanten, die ein Peer im zu anderen Peers besitzt. Die Anzahl der Peer-Mappings zwischen diesen Peers wird dadurch nicht gesetzt. Eine Kante kann genau einem Peer-Mapping oder auch mehreren Peer-Mappings entsprechen. Bei dem gewählten Datenbank-Schema lag die Zahl der Peer-Mappings, die eine Kante symbolisieren zwischen 1 und 3.

Nach der Erzeugung wurde die PDMS-Instanz angefragt. Dazu wurde jeweils von einem zufällig ausgewählten Peer für eine zufällig gewählte Relation seines Peer-Schemas eine Anfrage ohne Joins und Selektionen erzeugt. Diese Anfragen wurden je PDMS-Instanz einmal ohne die Verwendung von Zwischenspeichern, einmal mit Verwendung des Anfrage-Caches und einmal unter Nutzung des globalen Caches ausgeführt. Da es sich um statische

PDMS-Instanzen handelt, deren Struktur und Daten sich nach der Erzeugung nicht mehr verändern, sollte das Ergebnis unabhängig vom gewählten Zwischenspeicher immer gleich sein. Um Fehler in der Implementierung der Zwischenspeicher auszuschließen, wurde die Gleichheit der Anfrageergebnisse geprüft. Bei allen Experimente waren die gelieferten Ergebnismengen immer gleich, so dass von einer korrekten Funktion der Implementierung ausgegangen werden kann.

Bei der Durchführung der Experimente stellte sich heraus, dass das Berechnen vollständiger Anfrageergebnisse von Anfragen an PDMS-Instanzen vom Typ *zyklisch* mit 20 und mehr Peers praktisch nicht durchführbar waren. Dies zeigt auch die Motivation für Pruningstrategien im Abschnitt 5.1. Der Grund hiefür dürfte die hohe Anzahl der Peer-Mappings in der erzeugten PDMS-Instanz sein. Der Maximalgrad des PDMS lag bei 6. Wie weiter oben beschrieben muss dieser Faktor für das verwendete DB-Schema mit einem Wert zwischen 1 und 3 multipliziert werden, um auf die Anzahl von Peer-Mappings kommen. Daher wurden vom *zyklischen PDMS-Typ* nur Instanzen mit maximal 15 Peers erzeugt. In nachfolgenden Arbeiten sollte der Zusammenhang zwischen der Anzahl der Peer-Mappings und der Durchführbarkeit von Experimenten näher untersucht werden.

Für die Analyse der Auswirkungen (Ersparnisse) von Zwischenspeichern werden die durch die Anfragebearbeitung einer Nutzer-Anfrage erzeugten Peer-Anfragen betrachtet. Die Anzahl der Peer-Anfragen bei einer Anfrage ohne Zwischenspeicher wird dabei als Referenzwert genommen. Das Benutzen von Zwischenspeicher reduziert in vielen Fällen diese Anzahl. Die verursachten Kosten einer Anfrage mit Cache liegen unterhalb der Kosten einer Anfrage ohne Cache. In Abbildung 6.5 sind die über alle angefragten PDMS-Instanzen gemittelten Kosten im Verhältnis zur Anfrage ohne Cache je Graphentyp dargestellt.

Wie erwartet liefert der globale Cache unabhängig vom Graphentyp durchschnittlich höhere Einsparungen als der Anfrage-Cache. Dies lässt sich dadurch erklären, dass der globale Cache durch das Speichern sämtlicher Anfrageergebnisse die Funktionen des Anfrage-Caches übernimmt und zusätzlich die gespeicherten Ergebnisse bei weiteren Peer-Anfragen genutzt werden können.

Interessant ist, dass der Typ eines Peer Data Management Systems starken Einfluss auf die Einsparungen von Peer-Anfragen mit Hilfe von Zwischenspeichern hat. Während beim Typ *Baum* die Kosten gegenüber der Anfrage ohne Cache zwischen 98 und 97 Prozent liegen, und damit nur zwischen zwei und drei Prozent der Anfragen eingespart werden können, brauchen beim Typ *azyklisch* je nach Cache acht bis 15 Prozent und beim den *zyklischen* Peer Data Management Systemen bis zu 30 Prozent der Peer-Anfragen nicht ausgeführt werden. Der Grund dürfte die größere Vernetzung der Peers untereinander durch Peer-Mappings sein. Der angefragte Peer wurde zufällig aus der PDMS-Instanz ausgewählt. Bei einer baumartigen Struktur kann die Wahl des Peers sehr entscheidend für die resultierende Anfragetiefe sein. Die Richtung der durch den PDMS-Generator erzeugten Mappings ist zufällig. Dadurch können gerichtete Pfade in einem Baum sehr kurz sein, während die Pfade in den *azyklischen* und *zyklischen Typen* durch alternative Peer-

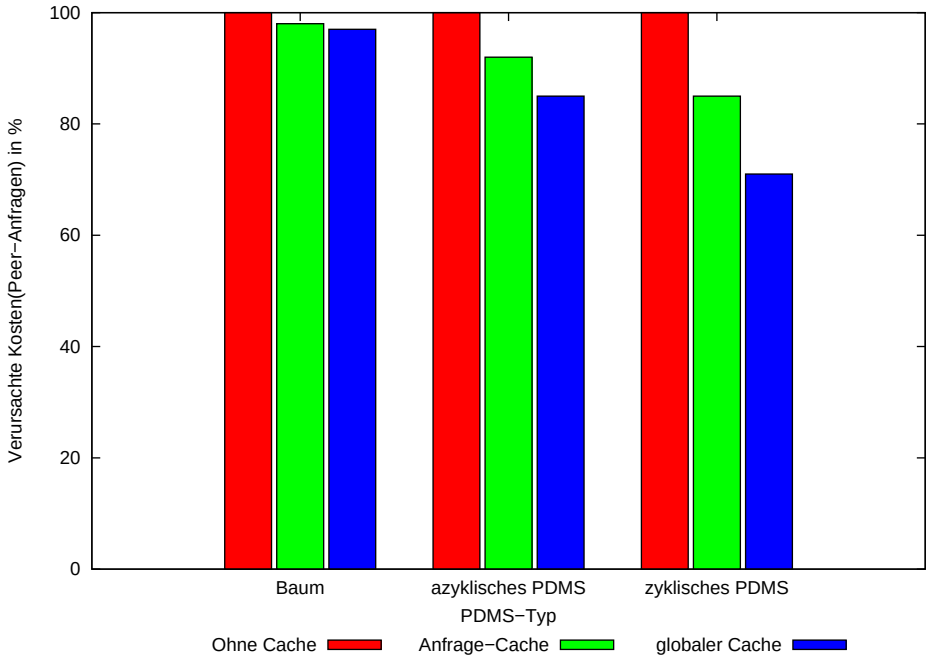

Abbildung 6.5: Durchschnittliche Kosten(Peer-Anfragen) unter Verwendung von Zwischenspeichern gegenüber Anfragen ohne Verwendung von Zwischenspeichern

Mappings tendenziell länger ausfallen. Die Länge dieser Peer-Mapping Pfade entspricht der möglichen Tiefe von Pfaden im globalen Anfrageplan. Mit der Tiefe des Anfrageplanes steigt auch die Zahl der Peer-Anfragen und somit die Zahl der Anfragen, die sich mit Hilfe eines Zwischenspeichers einsparen lassen. Dabei gilt, je höher im Anfragebaum eine Peer-Anfrage gespart werden kann und desto verzweigter der darunter liegenden Zweig des Anfragebaums insgesamt ist, desto größer ist die Ersparnis. Die Anfragebäume beim PDMS-Typ *Baum* fallen klein aus und besitzen weniger Verzweigungen. Dadurch ist das Einsparpotential im Gegensatz zu den beiden anderen Typen entsprechend gering. Die Anfragebäume von *zyklischen PDMS-Typen* fallen durch eine größere Anzahl an alternativen Mapping-Pfaden durchaus größer und verzweigter aus, als die Anfragebäume der *azyklischen* PDMS-Instanzen. Somit bieten sie ein entsprechend größeres Einsparpotential, das sich auch in den durchschnittlichen gesparten Peer-Anfragen zeigt.

Während der Unterschied zwischen dem Anfrage-Cache und dem globalen Cache in einem baumartigen PDMS eher gering ausfällt, können durch den globalen Cache bei den *azyklischen* und *zyklischen Typen* gegenüber dem Anfrage-Cache deutlich mehr Peer-Anfragen eingespart werden. Diese Beobachtung lässt sich dadurch erklären, dass die Wahrscheinlichkeit, dass ein Peer den globalen Cache nutzen kann, in den *zyklischen* und *azyklischen PDMS-Typen* höher ist. Der globale Cache hat gegenüber dem Anfrage-Cache genau dann einen Vorteil, wenn ein Peer mehr als einmal angefragt wird. Während die gespeicherten Ergebnisse beim Anfrage-Cache mit der Beantwortung einer Peer-Anfrage verworfen werden, bleiben sie beim globalen Cache bestehen. Wird nun ein Peer zur Beantwortung einer

Nutzer-Anfrage im Anfragebaum mehrfach angefragt und kann dabei dieselben Mappings nutzen, die zu gleichen Anfragen führen, müssen diese Anfragen bei der Nutzung des globalen Caches nicht ausgeführt werden. Ein Peer kann im Anfrageplan nur dann mehrfach angefragt werden, wenn es mehrere Peer-Mappings gibt, die auf ihn verweisen und der Peer auf verschiedenen Mapping-Pfaden angefragt wird. Dies ist durch die höhere Vernetzung durch Peer-Mappings bei den *zyklischen* und *azyklischen PDMS-Typen* häufiger als beim PDMS-Typ *Baum* der Fall.

Durch die Nutzung von Zwischenspeichern kann die Anfragebearbeitung durch Einsparung von Peer-Anfragen in Abhängigkeit von PDMS-Typ beschleunigt werden. Dabei konnten in den Experimenten für den *zyklischen PDMS-Typ* eine durchschnittliche Einsparungen von 30 Prozent erzielt werden, ohne dabei, wie die untersuchten Pruninigstrategien in den nächsten zwei Abschnitten, das Anfrageergebnis zu beeinflussen.

6.6 Informationsverlust durch Begrenzung der Anfragetiefe

Die Begrenzung der Anfragetiefe ist als Pruningstrategie bereits in Abschnitt 5.4 genauer beschrieben. In einem zweiten Experiment werden nun die Auswirkungen der Begrenzung der Anfragetiefe untersucht. Dazu werden genau wie beim vorherigen Experiment zu jedem PDMS-Typ jeweils 50 PDMS-Instanzen mit 5, 10, 15, 20 und 25 Peers erzeugt und angefragt. Auf die Erzeugung von *zyklischen* PDMS-Instanzen mit 20 und 25 Peers wird aufgrund der im vorherigen Abschnitt beschriebenen Probleme verzichtet.

Jede PDMS-Instanz wurde sechs Mal angefragt. Die erste Anfrage wurde ohne eine Begrenzung der Anfragetiefe ausgeführt. Das Anfrageergebnis ist somit vollständig. Neben der Anzahl der Tupel wurde die maximale Tiefe des Anfragebaumes ermittelt und als Basis für die weiteren Anfragen genommen. Für die zweite Anfrage wurde die Anfragetiefe auf zehn Prozent der Tiefe der vollständigen Anfrage gesetzt. Bei den weiteren Anfragen wurde die Tiefe auf 25, 50, 75 und 90 Prozent begrenzt[5]. Bei allen Anfragen kam der Anfrage-Cache zum Einsatz. Die bei *zyklischen PDMS-Typen* notwendige Zyklenvermeidung wurde jeweils nur bei der ersten Anfrage zur Ermittlung der vollständigen Antwort genutzt. Bei einer Begrenzung der Tiefe in den weiteren Anfragen ist die Zyklenvermeidung nicht notwendig[6] und wurde daher nicht verwendet.

6.6.1 Kostenersparnis

Als erstes sollen die Auswirkungen der Begrenzung der Anfragetiefe analysiert werden. Die Abbildung 6.6 zeigt die prozentualen Kosten von Peer-Anfragen gegenüber der vollständigen Anfrage. Zum Vergleich ist die vollständige Anfrage mit aufgeführt(100 Prozent).

[5] Je höher die Prozentzahl desto weniger weicht die begrenzte Anfragetiefe von der maximalen Anfragetiefe ab.

[6] siehe Abschnitt 5.4

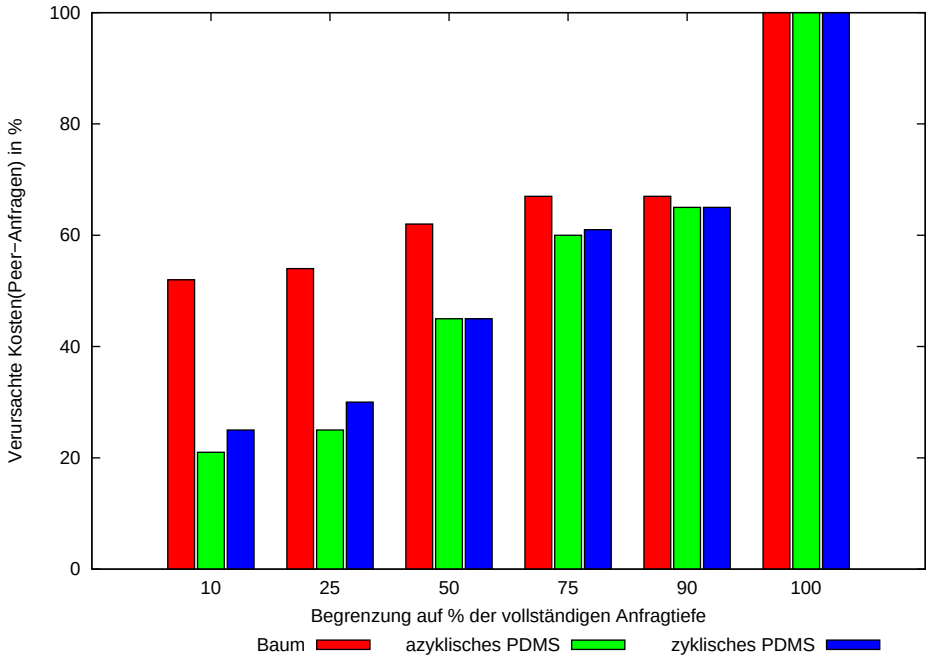

Abbildung 6.6: Durchschnittliche Kosten (Peer-Anfragen) durch Begrenzung der maximalen Anfragetiefe gegenüber der vollständigen Anfrage

Es ist zu erkennen, dass eine Begrenzung der Anfragetiefe auf zehn Prozent der Tiefe des vollständigen Anfragebaumes bei einem PDMS von Typ *Baum* deutlich weniger Einsparungen an Peer-Anfragen bietet als bei den beiden anderen Typen. Dies kann wie bei den Zwischenspeichern mit der geringen Tiefe und Verzweigung der Anfragebäume dieses Typs erklärt werden. Bei einer Begrenzung auf 90 Prozent werden vorwiegend die Blätter der kompletten Anfragebäume ausgelassen, so dass sich die Einsparungen der drei Typen angleichen.

Ebenso ist auffällig, dass die Einsparungen bei *zyklischen* PDMS tendenziell geringer ausfallen als bei den *azyklischen*. Die Ursache dieses Phänomens kann die notwendige Zyklenvermeidung bei den vollständigen Anfragen in *zyklischen* PDMS oder die im Durchschnitt geringere Peeranzahl[7] dieses PDMS-Typs sein. Beides führt zu einer geringeren Tiefe und Verzweigung der Anfragebäume und somit zu einem kleineren Einsparpotential.

6.6.2 Extensionale Vollständigkeit

Die Begrenzung der Anfragetiefe ist eine Pruningstrategie, welche im Allgemeinen dazu führt, dass das Anfrageergebnis nicht vollständig zurückgeliefert wird. Die Abbildung 6.7 zeigt die Vollständigkeit der Anfrageergebnisse je nach Begrenzung der Anfragetiefe. Die Vollständigkeit ist dabei der Quotient aus der Zahl der Ergebnistupel einer Anfrage mit begrenzter Anfragetiefe und Zahl der Ergebnistupel ohne Begrenzung. Sie beschreibt so-

[7] Es wurden nur Instanzen mit 5, 10 und 15 Peers erzeugt

mit, wie viel Prozent der Tupel der vollständigen Anfrage durch eine Begrenzung geliefert werden.

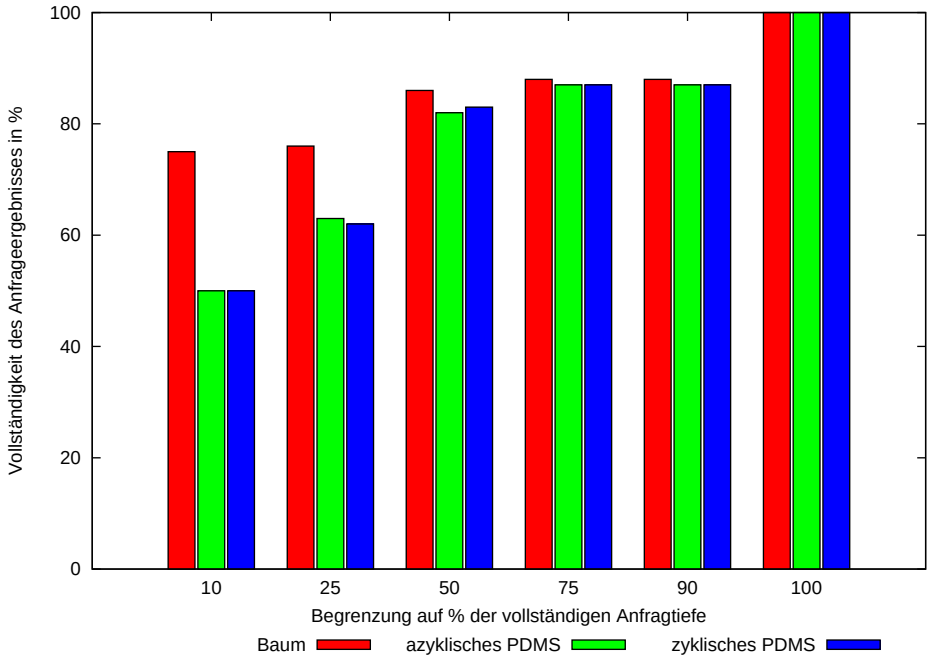

Abbildung 6.7: Durchschnittliche Vollständigkeit des Anfrageergebnisses durch Begrenzung der maximalen Anfragetiefe

In der Abbildung 6.7 ist zu Erkennen, dass die Vollständigkeit der Anfrageergebnisse, selbst bei großen Einschränkungen auf zehn Prozent der Anfragetiefe des vollständigen Anfragebaumes, im Durchschnitt 50 Prozent nicht unterschreitet. Auch erreicht die Vollständigkeit bei einer 50 prozentigen Beschränkung auf durchschnittliche Werte von 80 Prozent. Die für eine Anfrage relevanten Ergebnistupel setzen sich somit eher aus den Daten von Peers zusammen, die in der Nähe der Wurzel des Anfragebaumes und somit über wenige Peer-Mappings vom angefragten Peer erreichbar sind. Dies bestätigt die Annahmen aus Beispiel 5.3 im Abschnitt 5.4. Damit lässt sich auch die Beobachtung begründen, dass PDMS-Instanzen von Typ *Baum* trotz starker Begrenzung der Anfragetiefe relativ vollständig sind. Anfragebäume dieses Typs sind eher klein, so dass weniger Peer-Mappings eingespart werden können, wodurch aber auch die Vollständigkeit weniger leidet.

Abschließend kann zu diesem Experiment gesagt werden, dass die Begrenzung der Anfragetiefe auf die Hälfte der Tiefe der vollständigen Anfrage bei den erzeugten PDMS-Instanzen zu einer Einsparung von durchschnittlich 50 Prozent aller Peer-Anfragen führte. Dabei liegt die Vollständigkeit des Anfrageergebnisses im Schnitt bei 80 Prozent. Interessant ist die Tatsache, dass relativ viele Daten aus Peers der Blätter des Anfragebaumes kommen. Wie bereits oben erwähnt, werden bei einer Begrenzung auf 90 Prozent der voll-

ständigen Anfragetiefe vor allem die Blätter des vollständigen Anfragebaumes entfernt. Die Vollständigkeit der Anfragen bei dieser Begrenzung betrug im Schnitt 87 Prozent. Somit hängen 13 Prozent der Ergebnistupel direkt oder indirekt (über Joins) von den Daten der Peers aus den Blättern ab.

Problematisch bei dieser Pruningstrategie ist, dass die Tiefe des Anfragebaumes begrenzt werden muss, ohne das der vollständige Anfragebaum bekannt ist. Somit lässt sich nicht abschätzen, welche Ersparnis und welcher Grad der Vollständigkeit durch eine Begrenzung auf eine Tiefe erreicht wird. Dazu müssten das das vollständige Anfrageergebnis und der vollständige Anfragebaum gegeben sein. Auch kann bei dieser Pruningstrategie kein fester Wert für die maximal erlaubten Kosten festgelegt werden. Ein weiterer Nachteil ist, dass Mappings unabhängig ihrer Qualität gleich behandelt werden. Somit werden auch unattraktive Pfade im Anfragebaum verfolgt.

6.7 Qualitätsbasiertes Pruning

Die bereits im Abschnitt 5.5 erläuterte Pruningstrategie, welche die zu nutzenden Mappings anhand von Qualitätsmerkmalen auswählt, wird in diesem Abschnitt näher untersucht. Als ein sehr einfaches Qualitätsmerkmal eines Mappings soll dabei die *Selektivität* dienen. Der PDMS-Generator weist dazu jedem Selektionsprädikat eine Selektivität zwischen 0 und 1 zu, die angibt, wie viel Prozent der Tupel seiner lokalen Quelle ein Peer über dieses Peer-Mapping liefert[8]. Die betrachtete Qualität eines Mappings ist dann das Produkt aller Selektivitäten eines Mappings, wobei angenommen wird, dass die Selektionen unabhängig voneinander sind. Ein Peer-Mapping ohne Selektionen hat demnach die höchste Qualität 1. Analog dazu ergibt sich die Qualität (Selektivität) eines Anfragepfades aus dem Produkt der Selektivitäten aller auf diesem Pfad genutzten Peer-Mappings, wobei ebenfalls von einer Unabhängigkeit der Selektionen ausgegangen wird. Einer Anfrage wird eine untere Qualitätsgrenze mitgegeben, die ein Anfragepfad nicht unterschreiten darf. Ein Peer, der eine Anfrage beantwortet, nutzt demnach nur Peer-Mappings, deren Selektivität multipliziert mit der Selektivität des aktuellen Anfragepfades oberhalb der angegebenen Qualitätsgrenze liegen.

Für dieses Experiment wurde genau die gleiche Anzahl an PDMS-Instanzen der drei verschiedenen PDMS-Typen wie im vorherigen Experiment erzeugt. Die erste Anfrage wurde dabei wieder vollständig ausgeführt und als Referenz für die weiteren Anfragen, bei denen eine untere Qualitätsgrenze angegeben wurde, genommen.

6.7.1 Kostenersparnis

Die Ergebnisse der durchschnittlichen Kosten (Peer-Anfragen) bei unterschiedlichen unteren Qualitätsgrenzen gegenüber der vollständigen Anfrage sind in Abbildung 6.8 zusammengefasst. Die Qualitätsgrenze von 0 Prozent stellt die vollständige Anfrage dar.

[8] Die genaue Definition ist in [6] nachzulesen

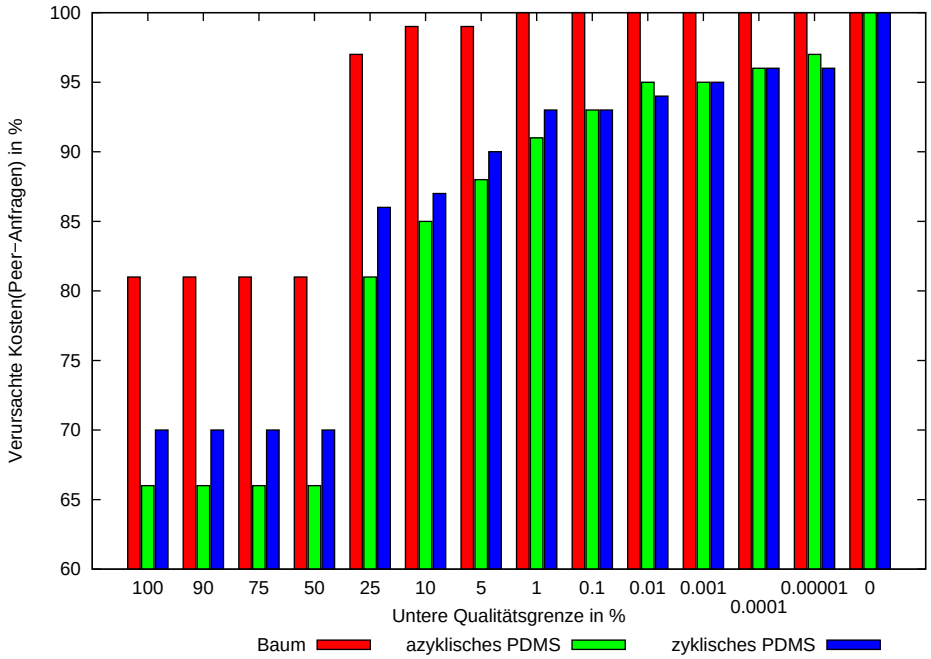

Abbildung 6.8: Durchschnittliche Kosten (Peer-Anfragen) durch Angabe einer unteren Qualitätsgrenze gegenüber der vollständigen Anfrage

Wieder gut zu erkennen ist das tendenziell niedrigere Einsparpotenzial bei PDMS-Instanzen vom Typ *Baum*. Der „Sprung“ bei den Kosten zwischen den Qualitätsgrenzen 50 Prozent und 25 Prozent lässt sich durch eine Einstellung für den PDMS-Generator erklären. Es wurden Peer-Mappings erzeugt, die mit einer 30 prozentigen Wahrscheinlichkeit eine Selektivität von ungefähr 50 Prozent haben. Somit werden bei einer Qualitätsgrenze zwischen 100 und 50 Prozent auf einem Anfragepfad keine zwei Peer-Mappings mit einer Selektion von 50 Prozent verwendet. Bei einer Grenze von 25 Prozent hingegen werden sie verwendet, so dass bei dieser Grenze die Kosten deutlich höher ausfallen. Bei den Grenzen unterhalb von 25 Prozent können dann mehr als zwei Mappings mit einer Selektivität von 50 Prozent in einem Anfragepfad genutzt werden. Dies scheint im vollständigen Anfrageplan jedoch seltener der Fall zu sein, wie sich an den jeweiligen Kosten ablesen lässt.

6.7.2 Extensionale Vollständigkeit

Neben den eingesparten Peer-Anfragen soll in diesem Experiment ebenfalls die Vollständigkeit der Anfragen mit einer unteren Qualitätsgrenze betrachtet werden. Die Abbildung 6.9[9] zeigt die relative Vollständigkeit zur vollständigen Anfrage über der jeweiligen unteren Grenze.

[9] Die y-Achse der Grafik beginnt nicht bei 0, um Unterschiede besser sichtbar zu machen.

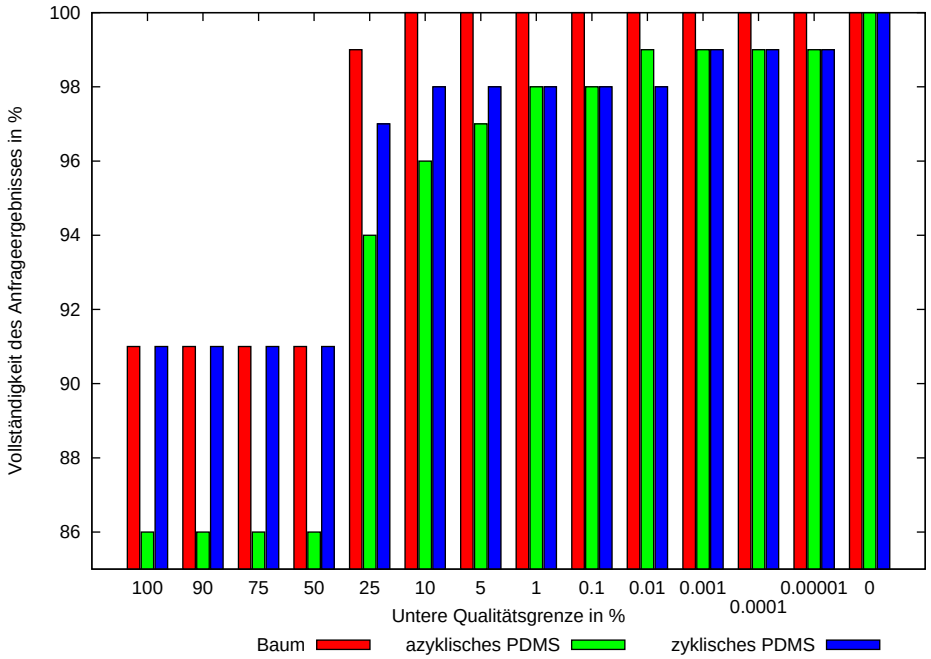

Abbildung 6.9: Durchschnittliche Vollständigkeit des Anfrageergebnisses durch Angabe einer unteren Qualitätsgrenze.

Ebenfalls erkennbar ist ein Sprung der relativen Vollständigkeit zwischen den unteren Grenzen 50 Prozent und 25 Prozent. Die Ursache hierfür ist die gleiche wie beim Diagramm zur Kostenersparnis. Ebenfalls gut zu erklären ist die mit abnehmender unteren Grenze schneller steigende Vollständigkeit des Typs *Baum* gegenüber den anderen Typen. Durch die tendenziell kleinere Anfragebäume und damit kürzeren Anfragepfade gibt es nur wenige Pfade die Selektivitäten unter zehn Prozent besitzen. Daher sind die Einsparungen sehr gering, die Vollständigkeit der Ergebnisse jedoch sehr hoch.

Allgemein lassen sich mit dieser Pruningstrategie die Einsparungen und Vollständigkeiten der Anfrageerbenisse feiner steuern als durch die strikte Begrenzung der Anfragetiefe. Es besteht jedoch ebenfalls das Problem, dass vor einer Anfrage an eine konkrete PDMS-Instanz nicht entschieden werden kann, welche Einsparungen und Vollständigkeiten zu erwarten sind.

6.7.3 Auswertung

Es ist nicht möglich zu sagen, dass eine der beiden Pruningstrategien bei einer PDMS-Instanz mit bestimmten Einstellungen immer X Prozent der Peer-Anfragen einspart und dabei im Durchschnitt eine Vollständigkeit von Y Prozent erreicht. Von daher können beide Pruningstrategien schlecht miteinander verglichen werden. Die Daten aus den Diagrammen sind nur gemittelte Werte. Sie schwanken zum Teil sehr stark und hängen von weiteren Faktoren, wie der Peer-Anzahl, der Anzahl von Selektionen und deren Selektivität von Peer-Mappings ab. Es zeigt sich jedoch, dass der qualitätsbasierte Pruning-

Ansatz feinere Einstellmöglichkeiten bietet. Während beim Reduzieren der Tiefe nur ganze Werte zwischen eins und der maximalen Anfragetiefe möglich sind, kann eine untere Qualitätsgrenze Werte zwischen 0 und 1 annehmen. Vor allem bei PDMS-Instanzen mit unterschiedlichen Selektivitäten der Peer-Mappings[10] sollten stetige Veränderungen der unteren Qualitätsgrenze zu stetigen Änderungen der Vollständigkeit und der Einsparung führen. Der beobachtete „Sprung“ tritt in der Regel nicht auf.

Ein großer Nachteil beider Methoden ist, dass ein Benutzer vor einer Anfrage nicht bestimmen kann, wie viel Kosten die Anfragebearbeitung erzeugen darf. Es ist schwer abschätzbar, wie Einschränkungen (Tiefe, Qualität) die Einsparungen von Peer-Anfragen und die Vollständigkeit des Ergebnisses beeinflussen. Beide Größen hängen stark von der aktuellen Struktur der PDMS-Instanz ab. Aufgrund der typischen Eigenschaften von Peer Data Management Systemen ändert sich die Struktur häufig und es gibt auch keine zentrale Instanz, die Auskunft über Änderungen geben könnte. Somit kennt ein Nutzer in der Regel die aktuelle Struktur der PDMS-Instanz nicht. Dies führt dazu, dass ein Nutzer „blind“ Einschränkungen zu seiner Anfrage treffen muss, über die Qualität und den gesparten Aufwand jedoch keine Aussagen machen kann. Zu restriktive Einschränken führen dazu, dass das Ergebnis zu stark eingeschränkt wird und der Benutzer die Anfrage möglicher weise erneut mit geringeren Einschränkungen stellen muss. Zu geringe Einschränkungen hingegen können zu sehr großen Anfragebäumen führen. Beide Fälle führen zu einem größerem Ressourcenverbrauch in einem PDMS als eigentlich nötig wäre.

Dieses Problem kann mit Hilfe der budgetbasierten Pruningstrategien[11] gelöst werden. Beim Einsatz dieser Strategien lassen sich Kosten vorgeben, die für eine Anfragebearbeitung höchstens verbraucht werden dürfen. Zusammen mit einem qualitätsbasierten Ansatz lässt sich das Budget auf Mappings entsprechend ihrer Qualität verteilen. Erste Experimente zu diesem viel versprechenden Ansatz gibt es in [7]. Experimente zu diesen Strategien können in zukünftigen Arbeiten mit dem *System P* durchgeführt werden. Die von Armin Roth bereits im Anfrageplaner implementierten budgetbasierten Strategien stehen im *System P* ebenfalls zur Verfügung[12].

6.8 Mehrere Rechner

Abschließend soll das *System P* verteilt auf mehreren Rechnern ausgeführt werden. Zum einen wird dadurch die Funktionsweise der Implementierung gezeigt und getestet, zum anderen kann untersucht werden, welche Komponenten wie viel Zeit bei der Bearbeitung einer Anfrage in Anspruch nehmen. Dabei ist zu erwarten, dass durch eine höhere Zahl beteiligter Rechner die gesamte Zeit, die zur Beantwortung einer Anfrage nötig ist, steigt. Die Ursache hierfür liegt in der nötigen Netzwerkübertragung der Ergebnisse der

[10] Leider waren in der für die Experimente genutzten Version des PDMS-Generators nur Selektionen mit einer Selektivität von 50 Prozent möglich.

[11] siehe Abschnitt 5.6

[12] Sie funktionierten für beliebige PDMS-Instanzen nicht zuverlässig genug und wurden daher nicht experimentell untersucht.

Peer-Anfragen und der fehlenden parallelen Anfragebearbeitung im *System P*. Zu einem Zeitpunkt ist immer nur ein Rechner mit der Anfragebearbeitung beschäftigt, die anderen Rechner sind entweder nicht beteiligt oder warten auf Zwischenergebnisse.

Für dieses Experiment wurde eine PDMS-Instanz mit zehn Peers von *Typ zyklisch* erzeugt. Eine Anfrage an einen der Peers führt zu weiteren Peer-Anfragen, die alle Peers mit in die Anfragebearbeitung einbezieht. Es standen insgesamt zehn Rechner, sechs Sun- und vier Linux-Rechner, zur Verfügung, welche nicht alle die gleiche Rechenleistung besaßen. Zunächst wurde die PDMS-Instanz auf nur einem Rechner in einer JVM ausgeführt. Alle zehn PDMS-Peers liefen somit in dieser JVM. Eine zweite JVM auf dem selben Rechner diente als Monitor-Peer, um die PDMS-Instanz anzufragen und die Ausführungszeit ermitteln zu können. Anschließend wurde auf den restlichen Rechnern nach und nach das *System P* in einer JVM gestartet und die PDMS-Peers der PDMS-Instanz auf alle verfügbaren JVMs verteilt und angefragt. Die Durchschnittswerte der Zeitmessungen von jeweils drei Anfragen sind im Diagramm 6.10 zusammengefasst.

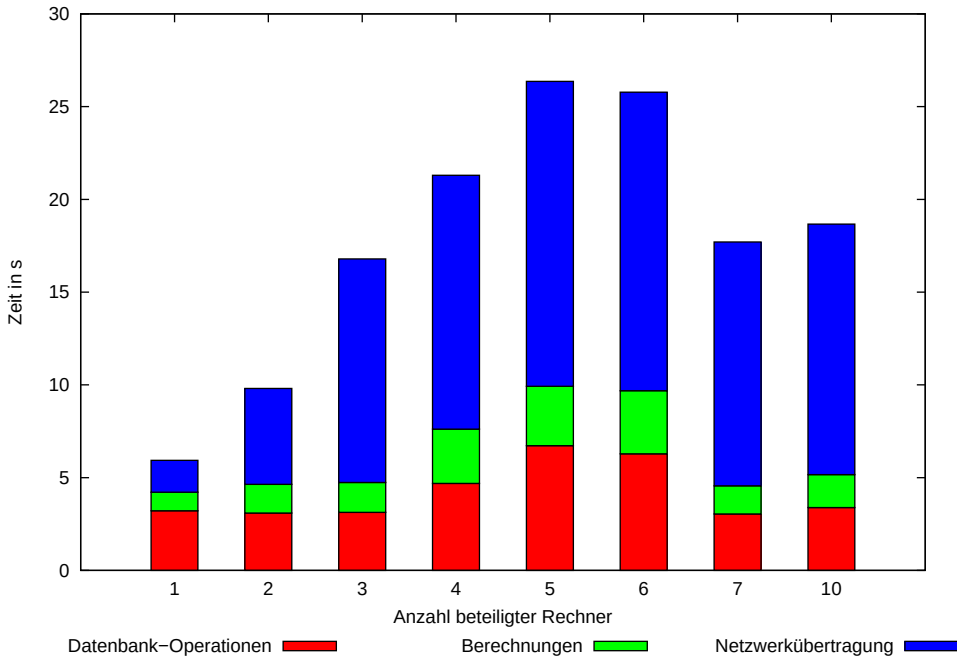

Abbildung 6.10: Anfragezeit in Abhängigkeit der Anzahl beteiligter Rechner

Die gesamte zur Beantwortung einer Anfrage benötigte Zeit wurde zur Auswertung in drei Bereiche unterteilt. Die Gesamtzeit setzt sich aus der Zeit für Datenbank-Operationen[13], Netzwerkübertragungen und sonstigen Berechnungen zusammen. Datenbankoperationen sind dabei alle Zugriffe auf lokale Quellen und die bei der Anfragebearbeitung notwendigen Join- und Union-Operationen. Die Zeit für die Netzwerkübertragungen setzt sich aus der Zeit für das Serialisieren und Deserialisieren der zu übertragenden Java-Objekte mit

[13] Es wurde nur HSQLDB verwendet.

XStream sowie der eigentlichen Übertragung durch das JXTA-Framework zusammen. Die restliche Zeit wird für zusätzliche Berechnungen, wie der Erzeugung des RG-Tree, dessen Übersetzung in einen optimierten Anfrageplan, sowie für Log-Ausgaben mit Log4j[14] verwendet.

Für die Auswertung sollen nur die ersten fünf Rechner betrachtet werden, da diese eine in etwa vergleichbare Rechenleistung besitzen. Läuft die PDMS-Instanz in nur einer JVM auf einem Rechner fällt auf, dass der Großteil der Zeit für Datenbankoperationen verwendet wird. Nur etwa 25 Prozent der Zeit wird für das Übertragen der Daten an den Monitor-Peer benötigt. Mit zunehmender Anzahl der beteiligten Rechner (JVMs) sinkt der Anteil der Datenbankoperationen an der Gesamtzeit. Dafür steigt der Anteil der Netzwerkübertragungen deutlich an. Dies spiegelt die Erwartungen wieder. Während durch die Übertragung der Zwischenergebnisse zwischen den Peers zusätzliche Zeit vergeht, kann insgesamt wegen der seriellen Ausführung der Peer-Anfragen keine Zeit eingespart werden. Die Zeiten für die Datenbankoperationen und sonstigen Berechnungen bleiben somit im wesentlichen gleich. Unterschiede können durch die leicht abweichenden Leistungen der Rechner erklärt werden.

Bei der Verwendung von zehn Rechnern wurden vier Linux-Rechner genutzt, deren Rechenleistung wesentlich höher war, als die der sechs Sun-Rechner. Da in der angefragten PDMS-Instanz nicht alle PDMS-Peers gleichmäßig an der Anfragebearbeitung beteiligt waren, ist es wahrscheinlich, dass ein oft angefragte PDMS-Peers auf den leistungsfähigeren Linux Rechner liefen. Somit lässt sich erklären, dass die Anfragebearbeitung bei sieben beteiligten Rechnern insgesamt schneller lief als bei sechs Rechnern.

Das Experiment sollte die Fähigkeit des *System P* demonstrieren, dass prinzipiell mehrere Rechner in ein PDMS integriert werden können. Aufgrund der seriellen Anfragebearbeitung erfolgt die Berechnung eines Anfrageergebnisses weniger effizient, als sie bei der parallelen Ausführung sein könnte. Es ist jedoch durchaus möglich, mehrere Anfragen gleichzeitig an verschieden Peers des *System P* zu stellen. In diesem Fall werden beide Anfragen parallel beantwortet.

[14] http://logging.apache.org/log4j/docs/

7 Fazit und Ausblick

Die vorliegende Arbeit beschreibt, wie Design eines Peer Data Management Systems und zeigt durch Implementierung im *System P* die praktische Umsetzung. Dabei wird auf die Kommunikation zwischen einzelnen Peers mit Hilfe des P2P-Frameworks JXTA eingegangen. Ebenso werden grundsätzliche Funktionen eines Peers erläutert, mit deren Hilfe eine PDMS-Instanz erzeugt und angefragt werden kann (Kapitel 3). Die wichtigste Funktionalität ist dabei die Anfragebearbeitung. Sie ist im Kapitel 4 näher beschrieben. Anfragen an komplexe Instanzen von Peer Data Management Systemen können bei der vollständigen Anfragebearbeitung eine sehr lange bis unendliche Laufzeit haben. Damit die Anfragebearbeitung jederzeit terminiert und praktisch durchführbar bleibt, sind Pruningstrategien nötig, welche den kompletten Anfrageplan einschränken. Eine Beschreibung und Charakterisierung der im *System P* verwendeten Strategien erfolgt im Kapitel 5. Im darauf folgenden Kapitel werden diese Strategien im Hinblick auf Kosten und Vollständigkeit untersucht. Die Experimente dazu konnten mit Hilfe des *System P* und des PDMS-Generators aus [6] automatisiert durchgeführt werden. Im abschließenden Experiment wird die Fähigkeit der Integration mehrerer Rechner mit Hilfe des *System P* gezeigt.

Durch die Experimente konnte die Funktionsweise ausgewählter Pruningstrategien und ihre Effizienz gezeigt werden. Es wurden Zwischenspeicher, eine Begrenzung der Anfragetiefe und eine qualitätsbasierte Pruningstrategie implementiert und untersucht. Als Ergebnis lässt sich feststellen, dass die Größe einer PDMS-Instanz im *System P* nicht durch die Anzahl der beteiligten Peers begrenzt wird, sondern durch die Größe der vollständigen Anfragepläne bestimmt ist (vergleiche Beispiel 5.1). Der Einsatz von Pruningstrategien ist daher nicht nur hilfreich, sondern mitunter notwendig.

Verbesserungen der Architektur

Derzeit ist ein Monitor-Peer nötig, um Anfragen an einzelne PDMS-Peers einer PDMS-Instanz zu stellen. In einer zukünftigen Version sollte es möglich sein, Anfragen über eine Kommandozeile oder GUI direkt an einen PDMS-Peer zu stellen. Ebenso wäre es wünschenswert die Anfragesprache von Datalog auf die bekanntere und häufig verwendete Sprache SQL umzustellen. Dadurch wird es einfacher, einen PDMS-Peer als Datenquelle in einem integrierenden System zu nutzen. Ziel kann ein JDBC-Treiber sein, über den ein *System P*-Peer angesprochen werden kann.

Verbesserungen der Anfragebearbeitung

Neben dem Aufbau des *System P* bietet die Anfragebearbeitung Raum für Verbesserungen. So erfolgt die Ausführung des Anfrageplanes derzeit seriell, obwohl eine parallele Ausführung die verfügbaren Ressourcen eines PDMS wesentlich effizienter nutzen kann. Ebenso wird das Ergebnis einer Anfrage erst komplett angezeigt, wenn alle Ergebnistupel berechnet sind. Eine Verbesserung wäre es, durch Pipelining die Tupel dem Nutzer zu präsentieren, sobald deren Berechnung erfolgt ist. Ebenso ist es erstrebenswert, wenn

zusätzlich zum Anfrageergbnis auch die Herkunft der Daten, die Data-Lineage, ersichtlich wird. Ein Benutzer kann so die Qualität einzelner Ergebnistupel abschätzen. Ausführlich werden diese Verbesserungen der Anfragebearbeitung am Ende von Kapitel 4 im Abschnitt 4.5 diskutiert.

Verbesserungen der Pruningstrategien

Neben zusätzlichen Experimenten zu den bestehenden Pruningstrategien, vor allem durch deren Kombination, sind viele Weiterentwicklungen der einzelnen Pruningstrategien denkbar (siehe auch Abschnitt 5.8). Die Implementierung der Zwischenspeicher ist sehr naiv und kann durch Begrenzung der Lebensdauer und Verdrängung der zwischengespeicherten Anfrageergebnisse erweitert werden. Die Qualitätsmerkmale der Peer-Mappings, nötig für die qualitäts- und budgetbasierten Pruningstrategien, bestehen zur Zeit aus einfachen, festen Selektivitäten, wobei von einer Unabhängigkeit der Selektionsattribute ausgegangen wird. Zukünftig sollte die Qualität eines Mappings mit Hilfe von Statistiken angepasst werden. Diese Statistiken lassen sich durch Sampling oder während der Anfragebearbeitung anhand der erhaltenen Ergebnisse erzeugen. Dabei können Histogramme ([1]) verwendet werden, um die Daten einer Quelle genauer zu charakterisieren.

Mit Hilfe des entwickelten *System P* können mehrere Rechner zu einem Peer Data Management System zusammengeschlossen werden. Die für die Anfragebearbeitung in diesem System genutzten Pruningstrategien wurden experimentell untersucht. Dabei wurden Grenzen der Anfragebearbeitung aufgezeigt, die die Entwicklung weiterer, ausgefeilterer Pruningstrategien notwendig machen. Diese Strategien können zukünftig in das *System P* integriert und getestet werden.

Literaturverzeichnis

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proc. of the ACM Int. Conf. on Management of Data (SIGMOD)*, 1999.

[2] David Thomas Andrew Hunt. *Pragmatic Unit Testing - In Java with JUnit*. Raleigh, 2003.

[3] Yingwei Cui and Jennifer Widom. Lineage Tracing for General Data Warehouse Transformations. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, 2001.

[4] Joseph D. Gradecki. *Mastering JXTA*. Wiley, 2002.

[5] 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.

[6] T. Hübner. *Entwicklung einer Testumgebung für ein Peer Data Management System.* Diplomarbeit, 2006.

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

[8] Armin Roth and Felix Naumann. Query answering in PDMS under limited resources Eingereicht zur Veröffentlichung, 2006.

[9] Brendon J. Wilson. *JXTA*. New Riders, 2002.

Selbständigkeitserklärung

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

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