Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Datenbanken und Informationssysteme

Anfragebearbeitung in integrierten Datenbanken nach dem Local-as-View-Paradigma

Masterarbeit

im Studiengang Informatik

von

Timo Hüther

Prüfer: **Prof. Dr. Udo Lipeck**
Zweitprüfer: **Dr. Hans Hermann Brüggemann**

Hannover, 07.11.2012

Zusammenfassung

Datenintegrationssysteme, die die Daten der zu integrierenden Quellen in einer Datenbank materialisieren, haben den Nachteil, dass die gespeicherten Informationen veralten können und es hohen Aufwand erfordert, einzelne Quellen der Integration hinzuzufügen oder sie aus ihr zu entfernen. Virtuell integrierte Datenbanken überwinden diese Probleme, indem sie selbst keine Daten gespeichert halten und erst zum Zeitpunkt der Anfragebearbeitung die angefragten Informationen zusammentragen. Diese virtuelle Integration hat dafür eine höhere Komplexität der Anfragebearbeitung zur Folge. Es gibt verschiedene Paradigmen, nach denen eine virtuelle Integration vollzogen werden kann. In dieser Arbeit wird das Local-as-View-Paradigma angewendet, welches die Datenquellen als Sichten auf ein ideales Gesamtschema annimmt.
Ziel der Arbeit ist es, integrierte Datenbanken praktisch umzusetzen und anzuwenden. Dafür werden bekannte Algorithmen und Verfahren, insbesondere der Bucket-Algorithmus und der MiniCon-Algorithmus, untersucht, implementiert und eingesetzt.
Eine Anfrageschnittstelle wird entwickelt, die nach dem Local-as-View-Paradigma Anfragen zur Laufzeit in Anfragepläne umschreibt, diese übersetzt und ausführt. Die mit Hilfe der Schnittstelle durchgeführten praktischen Versuche an integrierten Datenbanken werden beschrieben. Es wird gezeigt, dass die Umsetzung von virtuell integrierten Datenbanken zwar mit hohem Aufwand verbunden ist, dafür aber flexibel auf Veränderungen in der Quellenstruktur reagiert und ständige Aktualität der Ergebnisse gewährleistet werden kann. Anfragen können auch bei vielen Quellen und großen Datenmengen noch innerhalb weniger Sekunden beantwortet werden.
Insgesamt wird das Konzept von virtuell integrierten Datenbanken als sinnvolle Alternative zu bekannten Integrationsmethoden, insbesondere der Materialisierung von Daten, eingeschätzt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Privatleute, Unternehmen, Vereine und Behörden stellen seit einigen Jahren vermehrt Informationen, zum Beispiel im Rahmen von Open Data Bewegungen, zur freien Verfügung bereit. Neben freien Quellen gibt es auch viele proprietäre Quellen, zum Beispiel aus Wissenschaft und Wirtschaft.

Die stetig wachsende Anzahl an Informationen und der damit einhergehenden Vielzahl an Datenquellen machen es nötig, homogene Zugriffsmöglichkeiten auf möglichst viele relevante Informationen gleichzeitig anzubieten. Bekannte Lösungen, wie das Integrieren von mehreren Quellen zu einer großen materialisierten Gesamtdatenbank, sind häufig nicht durchführbar, weil viele Quellen nur eingeschränkte Zugriffsrechte auf die Datenmengen zulassen. Zudem veralten die gespeicherten Daten, da sich diese in den Quellen jederzeit ändern können. Das Hinzufügen und Entfernen von einzelnen Quellen zieht häufig viele Änderungen in der Integration nach sich.

Eine Möglichkeit, einheitliche Schnittstellen für viele heterogene Datenquellen anzubieten, aber dennoch keine Materialisierung vorzunehmen, stellen virtuell integrierte Datenbanken dar. Dabei werden Anfragen so formuliert, als seien sie nur an eine einzige Datenbank (bzw. ein einziges Datenbankschema) gerichtet. Zur Beantwortung werden dann jedoch viele verschiedene Quellen verwendet.

Bei virtuell integrierten Datenbanken gibt es verschiedene Vorgehensweisen, um Quellen zur Integration zusammenzufassen. Ein Konzept ist das Local-as-View-Paradigma. Dabei werden einzelne Quellen als Sichten auf ein ideales Gesamtschema angenommen. Anfragen werden immer an ein globales Schema gestellt und dann so umgeschrieben, dass sie von den Quellen bestmöglich beantwortet werden können.

Der Ablauf der Anfragebearbeitung in integrierten Datenbanken hat dabei eine erstaunlich hohe Komplexität. Alleine die Umformulierung von einer Anfrage an ein Schema in eine semantisch äquivalente Anfrage an mehrere heterogene Quellen, die sogenannte Anfrageplanung, ist mit großem Aufwand verbunden. Die anschließende Bereinigung und Vereinheitlichung der Ergebnisse der einzelnen Quellen ist ebenfalls sehr vielschichtig.

Intensive Forschung zur Integration komplexer Systeme wird erst seit den 1990er Jahren betrieben. Erforschte Konzepte und Algorithmen sind daher noch sehr jung und in der Praxis wenig eingesetzt. Erfahrungen mit dieser Art der Integration sind deshalb sehr rar.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, das Konzept von virtuell integrierten Datenbanken praktisch umzusetzen und anzuwenden. Dabei konzentriert sich diese Arbeit auf die Anfragebearbeitung nach dem Local-as-View-Paradigma, welches besagt, dass eine Anfrage zur Laufzeit dynamisch so umgeschrieben wird, dass es von einer Menge gegebener Quellen beantwortet werden kann. Die Quellen werden dabei als Sichten auf ein gedachtes globales Schema angenommen. Bestehende Algorithmen und Methoden sollen gefunden und analysiert werden. Vor- und Nachteile sollen erkannt und abgewogen werden.

Die untersuchten Konzepte und Algorithmen sollen anschließend in einer Anfrageschnittstelle implementiert werden. Diese soll Anfragen an eine virtuell integrierte Datenbank ermöglichen und sie nach dem Local-as-View-Paradigma durch mehrere verschiedene Datenquellen beantworten lassen.

Die erstellte Anfrageschnittstelle soll schließlich verwendet werden, um das Konzept der virtuell integrierten Datenbanken nach dem Local-as-View Paradigma durch Experimente zu bewerten. Dabei soll überprüft werden, ob die Anfragebearbeitung auch bei großen Datenmengen und komplexen Strukturen noch effizient bewältigt werden kann.

Zuletzt soll mit Hilfe der gewonnenen Erfahrung eingeschätzt werden, wieviel Aufwand im Vergleich zum Nutzen betrieben werden muss, um virtuell integrierte Datenbanken nach dem Local-as-View-Paradigma zu realisieren. Schließlich soll versucht werden, eine Wertung darüber abzugeben, ob das Konzept von virtuell integrierten Datenbanken praxistauglich ist.

1.3 Struktur der Arbeit

Diese Arbeit besteht aus insgesamt sechs Kapiteln. Nachdem Kapitel 1 eine Einleitung in das Thema geboten hat, werden in Kapitel 2 die Grundlagen dieser Arbeit vorgestellt und erläutert. Darin wird zunächst das Konzept der integrierten Datenbanken erklärt und zu anderen Datenbankarchitekturen abgegrenzt; die einzelnen Schritte der Anfragebearbeitung werden behandelt. Anschließend werden die in der Arbeit verwendeten Datenbanksprachen SQL und Datalog behandelt. Das Kapitel schließt mit der Beschreibung des Local-as-View-Paradigmas und dessen Alternativen.

Im 3. Kapitel wird ein intensiver Blick auf Algorithmen geworfen, die in der Anfragebearbeitung eingesetzt werden. Dabei werden mehrere alternative Algorithmen vorgestellt und diskutiert, die den jeweils angegebenen Quellen nachempfunden sind, aber teilweise geändert und optimiert wurden. Am Ende des Kapitels erfolgt eine Be-

wertung, die die praktikabelsten Algorithmen benennt. Diese werden im späteren Verlauf der Arbeit weiter verfolgt.

Kapitel 4 behandelt den praktischen Teil der Arbeit, in dem der Entwurf und die Implementierung der Anfrageschnittstelle beschrieben werden. Dort wird erklärt, wie die Konzepte und Algorithmen in der Praxis umgesetzt werden können und welche Einschränkungen dabei vorgenommen werden müssen.

Anschließend wird die implementierte Anfrageschnittstelle in Kapitel 5 verwendet, um sie an Beispielszenarien auf Tauglichkeit zu überprüfen. Dabei wird untersucht, wie sich verschiedene Algorithmen und Verfahren auf die Laufzeit und die Vollständigkeit auswirken.

Am Ende dieser Arbeit steht mit Kapitel 6 ein Fazit des Erarbeiteten. Weiterhin wird ein Ausblick auf weitere Arbeiten und mögliche Erweiterungen der Anfrageschnittstelle gegeben.

Kapitel 2

Grundlagen

Das vorliegende Kapitel behandelt die Grundlagen dieser Arbeit. Im ersten Abschnitt werden integrierte Datenbanken behandelt. Dabei werden sie zu anderen Datenbankarchitekturen abgegrenzt und die Vor- und Nachteile aufgezeigt. Im weiteren Verlauf des Abschnitts werden die einzelnen Schritte bei der Anfragebearbeitung in integrierten Datenbanken erklärt.

Im zweiten Abschnitt werden die beiden Datenbankanfragesprachen SQL und Datalog eingeführt, die in dieser Arbeit Verwendung finden.

Da man bei der Anfragebearbeitung zwischen verschiedenen Paradigmen zur Zusammenführung von Schemata unterscheidet, beschäftigt sich der letzte Abschnitt dieses Kapitels mit den beiden Paradigmen Local-as-View und Global-as-View und grenzt sie voneinander ab.

2.1 Anfragebearbeitung in integrierten Datenbanken

Grundlage eines monolithischen Datenbanksystems ist immer nur ein einzelner Datenspeicher, auf den zugegriffen wird. Entsprechend einfach gestaltet sich die Anfragebearbeitung.

Eine verteilte Datenbank besteht aus mehreren Datenbanken, die über ein Netzwerk verteilt, aber logisch zusammenhängend sind [ÖV99]. Sie sind in der Regel wenig heterogen und nicht autonom, da sie zentral entworfen und administriert werden. Bei verteilten Datenbanken ist die Verteilung explizit hergestellt worden und somit gewollt. Die Anfragebearbeitung ist zwar etwas komplizierter als bei monolithischen Datenbanksystemen, aber immer noch relativ leicht zu bewältigen.

Föderierte Datenbanken integrieren mehrere heterogene Datenquellen auf Schemaebene. Ein globales Schema ist dabei stabiler und zentraler Bezug für alle globalen Anfragen. Man unterscheidet zwischen materialisierten Systemen und virtuell integrierten Systemen. Bei materialisierten Systemen stammen die Daten zwar aus autonomen Quellen, werden aber in das föderierte Datenbanksystem kopiert und immer zu einem bestimmten Update-Intervall aktualisiert. Die Anfragebearbeitung ist dann

ähnlich einfach wie bei monolithischen Datenbanken. Die Daten in einem solchen System können aber immer nur so aktuell sein, wie es das Update-Intervall zulässt. Da das Auffrischen der Daten viel Netzwerk- und Prozessorlast verursachen kann, wird ein Update in der Regel eher selten vorgenommen, zum Beispiel einmal am Tag.

Gegenstand dieser Arbeit sind virtuell integrierte Datenbanken. Bei Anfragen an eine solche Datenbank werden die angefragten Informationen erst zur Laufzeit zusammengetragen, was den Vorteil ständiger Aktualität hat. Die Anfragebearbeitung lässt sich in die fünf wesentlichen Schritte einteilen: Anfrageplanung, -optimierung, -übersetzung, -ausführung und Ergebnisintegration [LN07]. Die Komplexität der Anfragebearbeitung, insbesondere der Anfrageplanung, ist um einiges höher als bei Datenbanken, die zu den anderen vorgestellten Datenbankarchitekturen gehören. Sie hängt von zwei wesentlichen Aspekten ab: Zum einen von der Ausdrucksmächtigkeit der Anfragesprache, zum anderen von dem gewählten Paradigma zur Zusammenführung von unterschiedlichen Quellen.

Die Anfragesprache dieser Arbeit ist Datalog. Sie hat ihren Ursprung bei logischen Datenbanken, kann aber auch für das relationale Modell verwendet werden. Dafür muss man den Sprachumfang allerdings einschränken, da mit Datalog Negationen und Rekursionen möglich sind, die äußerst ausdrucksstark sind. Die Sprache hat den Vorteil, dass sich Anfragen sehr kompakt formulieren lassen, was ausgesprochen nützlich ist, da in der Anfrageplanung die Anfrage selbst analysiert werden soll und nicht etwa das Ergebnis. Datalog mit notwendigen Einschränkungen und damit einhergehender Reduzierung der Ausdrucksmächtigkeit wird im Abschnitt 2.2.2 beschrieben.

Bei der Anfrageplanung in integrierten Datenbanken unterscheidet man zwischen den beiden grundlegenden Paradigmen Global-as-View (GaV) und Local-as-View (LaV). Bei GaV werden die Relationen des globalen Schemas als Sichten (Views) auf die lokalen Schemata der Quellen beschrieben. Bei LaV ist es umgekehrt. Die Relationenschemata der Quellen werden als Sichten auf ein (gedachtes) ideales Gesamtschema ausgedrückt. In dieser Arbeit wird die Anfragebearbeitung nach dem Local-as-View-Paradigma erfolgen. Es wird im Abschnitt 2.3.2 genau erklärt und von GaV abgegrenzt.

In Abbildung 2.1 ist der Ablauf der Anfragebearbeitung in integrierten Datenbanken nach dem Local-as-View-Paradigma dargestellt. Eine globale Anfrage wird an das globale Schema gestellt. Für die Anfrageplanung werden die Anfrage und alle gegebenen Sichten der Datenquellen benötigt. Das Ergebnis der Anfrageplanung sind ein oder mehrere Anfragepläne. Ein Anfrageplan ist eine Möglichkeit, die globale Anfrage mit Hilfe von lokalen Anfragen an die Quellen ganz oder teilweise zu beantworten.

Nach der Erzeugung müssen die einzelnen Anfragepläne von der globalen Anfragesprache in die Anfragesprache der Quellen übersetzt werden. In dieser Arbeit werden nur relationale Datenbanken, sowohl für die gobale Anfrage, als auch für die Quellen verwendet. Die Anfragesprache der integrierten Datenbank in dieser Arbeit ist Datalog; alle Quellen werden hier mit SQL angesprochen. Die Anfrageübersetzung ist also immer von Datalog zu SQL.

Die übersetzten Pläne können nun optimiert werden, wobei die Art der Optimierung vom jeweiligen Szenario abhängig ist. Hierbei wird unter anderem entschieden,

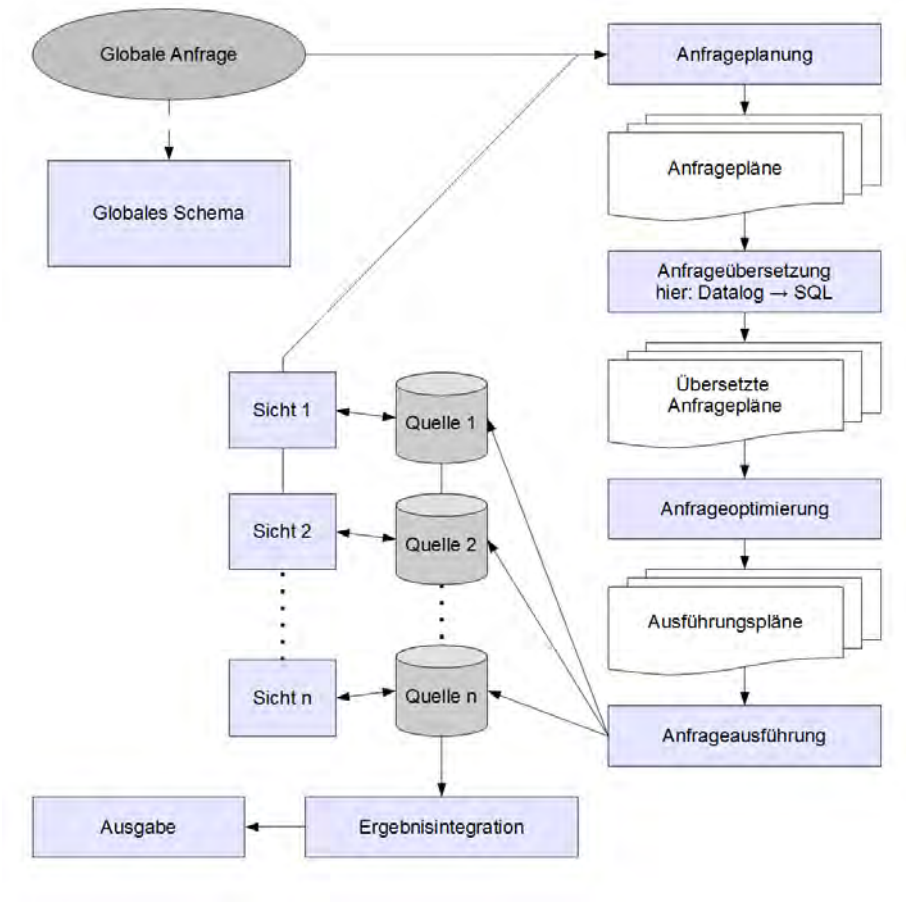


Abbildung 2.1: Anfragebearbeitung in integrierten Datenbanken nach dem Local-as-View-Paradigma

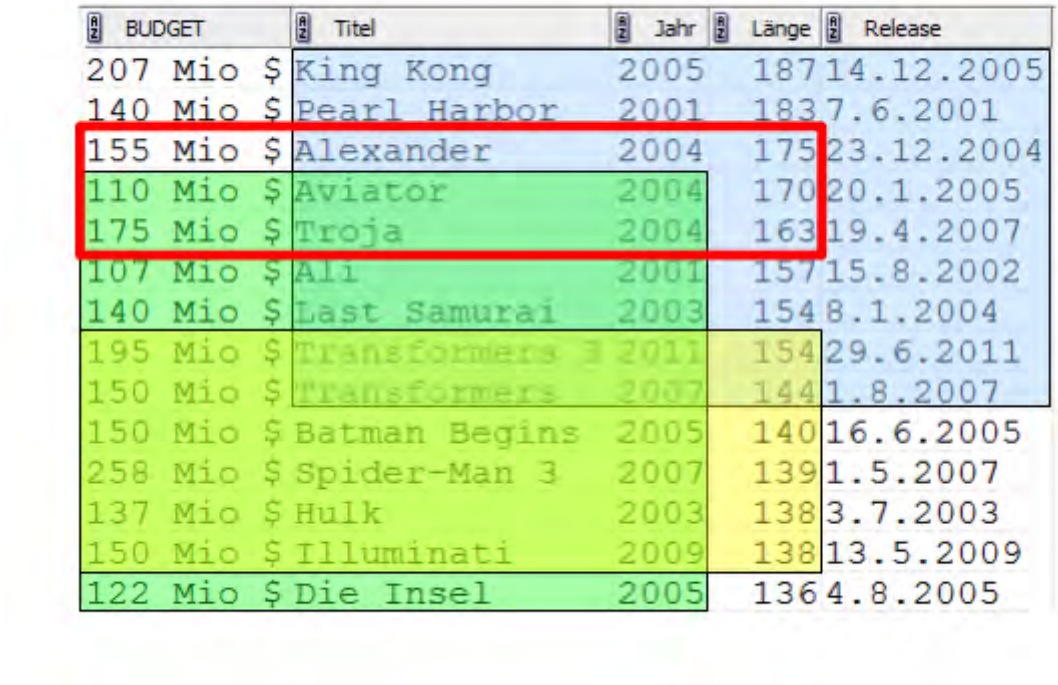
in welcher Reihenfolge die Teilanfragen ausgeführt werden bzw. ob und an welchen Quellen Joins durchgeführt werden. Die Anfrageoptimierung liefert für jeden Anfrageplan einen Ausführungsplan.

Danach werden die übersetzten und optimierten Anfragen ausgeführt, also zu den einzelnen Quellen zur Bearbeitung geschickt. Diese optimieren die Anfragen noch einmal lokal, führen sie anschließend nach ihren lokalen Gegebenheiten aus und senden ihre individuellen Ergebnisse an das integrierte Datenbanksystem zurück. Dieses muss im letzten Schritt die einzelnen Ergebnisse zu einem einheitlichen, homogenen Ergebnis integrieren und schließlich ausgeben.

Die einzelnen in den folgenden Abschnitten detaillierter erklärten einzelnen Schritte der Anfragebearbeitung stammen aus [LN07].

2.1.1 Anfrageplanung

Quellen können entweder eine Anfrage komplett, einen Teil der Anfrage oder überhaupt keine Anfrageteile beantworten. Um zu verbildlichen, was das bedeutet, ist in Abbildung 2.2 eine ideale Gesamtdatenbank über Kinofilme mit den Attributen Budget, Titel, Jahr, Länge und Releasedatum zu sehen. In den Farben Blau, Gelb und Grün eingefärbt sind die existierenden Beispielquellen. Rot umrandet ist das gewünschte Ergebnis einer globalen Anfrage („Alle Kinofilme mit Titel, Budget, Jahr und Länge aus dem Jahr 2004“).



BUDGET	Titel	Jahr	Länge	Release
207 Mio $	King Kong	2005	187	14.12.2005
140 Mio $	Pearl Harbor	2001	183	7.6.2001
155 Mio $	Alexander	2004	175	23.12.2004
110 Mio $	Aviator	2004	170	20.1.2005
175 Mio $	Troja	2004	163	19.4.2007
107 Mio $	Ali	2001	157	15.8.2002
140 Mio $	Last Samurai	2003	154	8.1.2004
195 Mio $	Transformers 3	2011	154	29.6.2011
150 Mio $	Transformers	2007	144	1.8.2007
150 Mio $	Batman Begins	2005	140	16.6.2005
258 Mio $	Spider-Man 3	2007	139	1.5.2007
137 Mio $	Hulk	2003	138	3.7.2003
150 Mio $	Illuminati	2009	138	13.5.2009
122 Mio $	Die Insel	2005	136	4.8.2005

Abbildung 2.2: Ideale Gesamtdatenbank mit drei Quellen und einer Anfrage

Die gelbe Quelle kann die Anfrage intensional vollständig beantworten. Extensional kann sie allerdings keine Tupel zum Ergebnis beitragen, da sie keine Filme aus dem Jahr 2004 enthält. Dass sie ein leeres Ergebnis liefern wird, stört zunächst nicht, schließlich könnte dies auch bei Anfragen an jede andere Datenbank passieren, wenn keine passenden Inhalte gespeichert sind.

Die blaue und die grüne Quelle können die Anfrage teilweise beantworten. Die blaue Quelle hat Titel, Jahr, Länge und Releasedatum von Filmen gespeichert und die grüne Quelle enthält Budget, Titel und Jahr von Filmen.

Die Aufgabe der Anfrageplanung ist es, mit Hilfe der gegebenen Quellen die Benutzeranfrage in einen oder mehrere Anfragepläne umzuschreiben. In einem Anfrageplan

wird festgelegt, welche Quelle welchen Teil der Anfrage beantworten soll und wie die einzelnen Teilanfragen verbunden werden. In diesem Beispiel würde die Anfrageplanung also zwei Anfragepläne erstellen: Einen mit der Quelle Gelb allein und einen mit den Quellen Blau und Grün zusammen. Der letztere Anfrageplan müsste noch um einen Join über das Attribut Titel ergänzt werden.

Grundsätzlich geht man bei integrierten Datenbanken von der Annahme aus, dass die Quellen extensional unvollständig sind, also wie im obigen Beispiel gezeigt, nicht alle existierenden Objekte der idealen Welt tatsächlich beinhalten. Die blaue Quelle beispielsweise enthält nur Filme mit einer Länge größer als 140. Außerdem können verschiedene Pläne auch für gleiche Objekte der realen Welt unterschiedliche Ergebnisse erzeugen. Dies tritt insbesondere dann auf, wenn zwei Quellen widersprüchliche Informationen gespeichert haben. Bei integrierten Datenbanken kommt das häufig vor, da man in der Regel nur autonome Quellen verwendet; Tipp- bzw. Rechtschreibfehler und verschiedene Schreibweisen (z.B. von Namen oder Adressen) sind keine Seltenheit. Daher können unterschiedliche Anfragepläne, wie im Beispiel zu sehen, auch zu unterschiedlichen Ergebnissen führen: Anfrageplan Gelb liefert ein leeres Ergebnis, Anfrageplan Blau-Grün liefert die Filme „Aviator“ und „Troja“. Zum Film „Alexander“ ist in keiner Quelle ein Budget gespeichert, der Film wird bei der gegebenen Anfrage also nicht gefunden[1], obwohl er zur gedachten idealen Datenbank gehört.

Wenn im Folgenden davon die Rede ist, dass eine Quelle eine Anfrage komplett oder teilweise beantworten kann, ist damit immer die intensionale Beantwortbarkeit gemeint. Mit Unvollständigkeit einer Quelle ist nachfolgend immer die extensionale Unvollständigkeit gemeint.

Damit dieser Schritt, das Erzeugen der Anfragepläne, automatisiert ausgeführt werden kann, gibt es verschiedene Anfrageplanungsalgorithmen. Das 3. Kapitel dieser Arbeit beschäftigt sich mit diesen Algorithmen. Dabei werden insbesondere der Bucket-Algorithmus (in verschiedenen Varianten) und der MiniCon-Algorithmus erklärt und bewertet.

Sollte ein Algorithmus mehrere alternative Anfragepläne finden, ist es auch Aufgabe der Anfrageplanung zu entscheiden, welche Pläne davon tatsächlich ausgeführt werden sollen. Da es aber nicht trivial ist zu entscheiden, welcher Plan der Beste ist, werden in der Regel alle Pläne benutzt und ihre Ergebnisse integriert (siehe Abschnitt 2.1.5). Aus Gründen der Performance kann es aber sinnvoll sein, nur bestimmte Pläne auszuwählen. Hier vermischt sich die Anfrageplanung mit der Optimierung.

2.1.2 Anfrageübersetzung

Ein in der Anfrageplanung erzeugter Anfrageplan kann aus mehreren Teilanfragen bestehen, die jeweils an genau eine Quelle gestellt werden. Sie sind in der Sprache der Anfrageschnittstelle formuliert. Die Teilanfragen müssen aus der Sprache der Anfrageschnittstelle in die Sprache übersetzt werden, in der die jeweilige Quelle angespro-

[1] Hier wird von (Inner) Joins ausgegangen. Würde man einen Outer Join verwenden, wäre der Film mit einem NULL-Wert bei Budget in der Ergebnismenge.

chen werden kann. Handelt es sich in beiden Fällen um relationale Datenbanken mit gleicher Anfragesprache, müssen für die Übersetzung meistens nur rein syntaktische Maßnahmen ausgeführt werden: Datentypen müssen angepasst, Datums- und Zeitangaben harmonisiert und Zeichen für Dezimalkommata angeglichen werden.

Handelt es sich um eine Übersetzung in andere Datenmodelle, wie zum Beispiel XML[2], ist die Übersetzung um einiges komplexer. Die Anfragen müssen erst in eine semantisch äquivalente Anfrage übersetzt werden und auch die Ergebnisse müssen wieder in relationale Form umgewandelt werden. Eine Anfrageübersetzung dieser Art wird meistens mit separaten Programmen gelöst, sogenannten Wrappern.

In dieser Arbeit werden Anfragen an die integrierte Datenbank in Datalog gestellt und alle Beispielquellen werden mit SQL angesprochen. Die Anfrageübersetzung ist also immer von Datalog zu SQL. Beide Anfragesprachen werden im Abschnitt 2.2 beschrieben.

2.1.3 Anfrageoptimierung

Das Ergebnis der Anfrageübersetzung sind Anfragepläne in übersetzter Form. Die Aufgabe der Anfrageoptimierung ist es, diese so anzupassen, dass physische Ausführungspläne entstehen. Die Ausführungspläne sollen möglichst wenig Kosten verursachen. Dabei ist der Kostenbegriff bei integrierten Datenbanken je nach Ziel der Anwendung differenziert zu betrachten. Das Ziel der Optimierung muss daher festgelegt werden, damit die Kosten bezüglich des Ziels minimiert werden können. Mögliche Ziele der Anfrageoptimierung könnten zum Beispiel die Minimierung der Ausführungszeit, die Minimierung des Netzwerkverkehrs oder die Minimierung tatsächlicher Kosten (zum Beispiel bei Anfragen an kommerzielle Systeme) sein.

Die Minimierung der Ausführungszeit kann man mit gleichen Techniken wie bei der Anfrageoptimierung in zentralen Datenbanksystemen erreichen. Selektionen können zum Beispiel möglichst nah an der Relation ausgeführt werden, damit die Anzahl der Tupel frühzeitig reduziert wird.

Möchte man Anfragen an ein kommerzielles System stellen und muss pro Anfrage einen Betrag bezahlen, kann man die Kosten minimieren, indem man möglichst selten Anfragen stellt und dafür möglichst viele Informationen auf einmal abfragt. Anders muss man Kosten minimieren, wenn das kommerzielle System pro Kilobyte oder pro Tupel abrechnet. Hier kommt es darauf an, möglichst nur die tatsächlich benötigten Daten abzufragen.

Im Folgenden werden zwei Elemente der Anfrageoptimierung in integrierten Datenbanken, die Ausführungsreihenfolge der verschiedenen Teilpläne und Caching, kurz vorgestellt.

[2]XML: Extensible Markup Language, wird spezifiziert durch das World Wide Web Consortium (Aktuelle Spezifikation: http://www.w3.org/TR/xml/).

Ausführungsreihenfolge

Die verschiedenen Teilpläne eines Anfrageplans sind in der Regel verbunden durch Joins. Das impliziert, dass Daten zwischen den Quellen transferiert werden. Eine naheliegende Strategie der Optimierung ist es, die Teilpläne in der Reihenfolge auszuführen, in der die Größe der Zwischenergebnisse möglichst gering ist. Da Zwischenergebnisse zwischen den Datenquellen verschickt werden, machen ihre Größe und die gewählte Netzwerkverbindung[3] für den Transfer die größte Zeit aus, um eine Anfrage zu beantworten.

Caching

Beim Caching wird die Tatsache ausgenutzt, dass häufig verschiedene Anfragepläne die gleichen Teilanfragen enthalten und somit einzelne Quellen mehrfach angefragt werden. Diese Redundanz kann durch einen globalen Blick auf alle Anfragen erkannt und verhindert werden. Solche Teilanfragen können, anstatt bei jedem Anfrageplan einzeln ausgeführt zu werden, global nur einmal ausgeführt und zwischengespeichert werden. Sollte ein Anfrageplan dann die gleiche Teilanfrage stellen, wird auf das gecachte Ergebnis zurückgegriffen, anstatt die Quelle erneut anzusprechen.

Der Cache darf nur bis zum Ende der Ausführung existieren, sonst könnte bei einer erneuten Anfrage nicht mehr die volle Aktualität gewährleistet werden, da sich die Quellen zu jeder Zeit ändern können. Würde der Cache länger bestehen, würde sich das System von einer rein virtuell integrierten Datenbank zu einer materialisierten Datenbank wandeln.

2.1.4 Anfrageausführung

Die einzelnen Teilanfragen der Ausführungspläne werden anschließend ausgeführt, also an die Datenquellen versandt. Dafür müssen Verbindungen aufgebaut, überwacht und abgebaut werden. Die Zwischenergebnisse der Datenquellen müssen im integrierten Datenbanksystem gespeichert werden.

Gleichzeitig muss die Ausführung überwacht werden. Die Netzwerkverbindungen zwischen dem Anfragesystem und der Datenquelle können abbrechen oder nicht reagieren. Solche Probleme sollten erkannt und wenn möglich, durch ein erneutes Stellen der Anfrage abgefangen werden.

Wenn alle Zwischenergebnisse vorliegen, müssen noch alle Operationen, die nicht in den jeweiligen Quellen ausgeführt werden konnten, durchgeführt werden. Hierbei handelt es sich meistens um Joins. Es kann sich aber auch um Selektionen handeln, die nicht direkt auf der Quelle erlaubt sind.

Sollte es bei der Ausführung der Anfrage zu Problemen kommen, muss in diesem Schritt reagiert werden. Ist eine Quelle zum Beispiel mehrmals nicht erreichbar, muss die Anfrageplanung unter Ausschluss der defekten Quelle wiederholt werden. Wenn dies notwendig ist, kann es sein, dass keine Anfragepläne mehr gefunden werden; die

[3] Typischerweise das Internet

Anfrage wäre dann mit dem aktuellen Zustand der integrierten Datenbank nicht beantwortbar.

Das Resultat der Anfrageausführung ist für jeden Ausführungsplan eine Menge von Ergebnistupeln. Diese Tupel müssen im nächsten und gleichzeitig letzten Schritt der Anfragebearbeitung noch zu einem einheitlichen Ergebnis integriert werden.

2.1.5 Ergebnisintegration

Pro Ausführungsplan gibt es eine Menge von Ergebnistupeln, die für das globale Ergebnis relevant sind. Die Aufgabe der Ergebnisintegration ist es, die Ergebnistupel zu vereinigen. Beim Zusammenfügen der Tupel ist zu beachten, dass die Ergebnisse häufig uneinheitlich oder redundant sein können. Die Auflösung von Widersprüchen und die Erkennung und Eliminierung von Duplikaten gehören somit ebenso zur Ergebnisintegration.

Die Reinigung von Daten ist in den meisten Fällen aber so aufwendig, dass sie nicht innerhalb von wenigen Sekunden durchgeführt werden könnte. Bei einem materialisierten System nimmt man diese Zeit in Kauf, da eine Integration nur zu einem bestimmten Update-Intervall stattfindet. Bei einem virtuell integrierten System, wie es Gegenstand dieser Arbeit ist, ist die Datenbereinigung nicht im Zuge der Anfragebearbeitung zu bewältigen. Außerdem ist manchmal domänenspezifisches Wissen für die Datenbereinigung erforderlich. Somit beschränkt sich die Ergebnisintegration in dieser Arbeit auf die Zusammenführung der einzelnen Ergebnistupel.

Statt der Reinigung werden in dieser Arbeit die ausgegebenen Ergebnisse mit einer Quellenangabe versehen, damit der Benutzer eventuelle Duplikate besser beurteilen kann.

2.2 Anfragesprachen

In diesem Abschnitt geht es um Anfragesprachen für relationale Datenbanken. Als erstes wird SQL, als die wichtigste und weitgehend standardisierte relationale Datenbanksprache, eingeführt. Da SQL allgemein eine große Bekanntheit genießt, wird hier nur kurz auf die für diese Arbeit relevanten Aspekte der Sprache eingegangen.

Im zweiten Teil wird die logische Anfragesprache Datalog erläutert, die sich durch eine sehr kompakte Schreibweise auszeichnet.

2.2.1 SQL

SQL[4] ist die am weitesten verbreitete Datenbanksprache für relationale Datenbanken [SMW04]. Sie ist semantisch an die englische Sprache angelehnt. Jede SQL-Anfrage ist nach dem Gerüst

SELECT ... **FROM** ... **WHERE** ...

[4]SQL ist definiert durch die ISO/IEC Standards 9075 und 13249.

aufgebaut, wobei die WHERE-Klausel nur benötigt wird, wenn die Anfrage an eine Bedingung gebunden ist (z. B. Jahr < 1960) oder mehrere Tabellen verbunden werden sollen. Im SELECT-Teil stehen diejenigen Spalten, die ausgegeben werden sollen. Im FROM-Element gibt man alle Tabellen an, an die die Anfrage gerichtet ist. Diese Teilmenge des Sprachumfangs nennt man auch Selektion-Projektion-Join-Anfragen (SPJ-Anfragen).

Dieses Grundgerüst kann durch weitere Sprachelemente, wie zum Beispiel ORDER BY, das das Ergebnis nach bestimmten Spalten sortiert oder GROUP BY, welches Zeilen gruppiert, erweitert werden. Außerdem können SQL-Anfragen Aggregierungsfunktionen benutzen und Unteranfragen enthalten. Es handelt sich also um eine sehr ausdrucksstarke Sprache, mit der man mächtige Anfragen an ein Datenbanksystem stellen kann.

Eine Anfrage über zwei Relationen an eine Filmdatenbank könnte zum Beispiel so formuliert werden:

```
SELECT titel,jahr,genre,adresse
FROM film,studio
WHERE film.studio = studio.name
AND jahr < 2000;
```

SQL stellt viele weitere Befehle bereit, die für die Manipulation von Daten gebraucht werden, wie INSERT, UPDATE und DELETE. Für die Strukturierung der Daten und Administration des Datenbanksystems gibt es nochmal zahlreiche Befehle, wie zum Beispiel CREATE, ALTER und DROP.

Da man aber bei integrierten Datenbanken in der Regel weder Daten einfügen noch ändern, sondern meistens nur lesend auf die Quellen zugreifen kann, beschränkt sich der für diese Arbeit benötigte Sprachumfang auf SPJ-Anfragen, wie oben erklärt. Alle Quellen, die im praktischen Teil dieser Arbeit benutzt werden, werden mit SQL angefragt.

Wenn man sich beim obigen Beispiel annimmt, dass noch weitere Attribute angefragt werden sollen, dafür mehr Tabellen angesprochen werden müssen und eventuell noch Bedingungen formuliert werden, hat man sehr schnell eine sehr lange und sperrige Anfrage. Aus diesem Grund wird im nächsten Abschnitt die sehr viel kompaktere Datenbanksprache Datalog erläutert.

2.2.2 Datalog

Datalog ist eine syntaktisch an Prolog angelehnte Datenbanksprache für deduktive Datenbanken[5] [KE06]. Datalog-Regeln bestehen aus Literalen, auch atomare Formeln genannt. Ein Literal ist von der Form:

[5]Deduktive Datenbanken sind eine Erweiterung des relationalen Datenmodells um die Logik erster Stufe, beinhalten also Tabellen und Regeln.

$$q(A_1, ..., A_m).$$

q bezeichnet dabei den Namen einer Relation oder ein eingebautes Prädikat ($\neq$, $=$, $\leq$, $<$, $>$, $\geq$). Nachfolgend wird bei eingebauten Vergleichsoperationen jedoch statt der Präfixnotation $<(X, Y)$ die Infixnotation $X < Y$ verwendet; sie werden im Weiteren Bedingungen genannt. Eine Datalog-Regel besteht aus einem Kopf und einem Rumpf, die durch :- getrennt sind, und ist von der Form:

$$p(X_1, ..., X_m) :- q_1(A_{11}, ..., A_{1m_1}), ..., q_n(A_{n1}, ..., A_{nm_n})$$

wobei jedes $q_j(...)$ eine atomare Formel ist. $X_1, ..., X_m$ sind Variablen, die mindestens einmal im Rumpf, also als A_{ij} (mit $1 \leq i \leq n$ und $1 \leq j \leq m$) vorkommen müssen. Die obige Datalog-Regel entspricht der folgenden Logikregel:

$$p(...) \vee \neg q_1(...) \vee ... \vee \neg q_n(...)$$

Diese Regel lässt sich mittels der logischen Implikation in die Form einer Horn-Klausel[6] umformen:

$$q_1(...) \wedge ... \wedge q_n(...) \Rightarrow p(...)$$

Eine solche Formel besagt, dass p dann wahr ist, wenn alle q_i (mit $1 \leq i \leq n$) wahr sind. Eine Datalog-Regel ist nichts anderes als eine Horn-Klausel. Das Ergebnis der Ausführung einer Datalog-Regel sind alle Tupel, die die Regel erfüllen, sie also wahr machen.

Wie im Abschnitt 2.2.1 beschrieben, sind für diese Arbeit nur SPJ-Anfragen bedeutend. Im Folgenden wird erklärt, wie Selektion, Projektion und Join in Datalog umgesetzt werden.

Eine Selektion in Datalog funktioniert ähnlich einfach wie in SQL. Die Anfrage

$$q(titel, regisseur, jahr, genre) :- film(titel, regisseur, jahr, genre), jahr \geq 2010;$$

gibt nur diejenigen Filme mit $titel$, $regisseur$, $jahr$ und $genre$ aus, die im Jahr 2010 oder später veröffentlicht wurden.

Durch das Entfernen von Variablen aus dem Kopf, die im Rumpf vorkommen, erreicht man in Datalog sehr einfach eine Projektion:

$$q(genre) :- film(titel, regisseur, jahr, genre);$$

Mit dieser Anfrage werden aus der Relation $film$ nur noch die Attribute $genre$ ausgegeben.

Einen Join in Datalog kann man bilden, indem man mehrere Relationen in den Rumpf der Anfrage schreibt und sie über ein Attribut verknüpft. Die SQL-Beispielan-

[6] Benannt nach dem Mathematiker Alfred Horn.

frage aus Abschnitt 2.2.1 könnte wie folgt umgesetzt werden:

$$q(titel, jahr, genre, adresse) :- film(titel, regisseur, jahr, genre, studio),$$
$$studio(name, adresse), studio = name, jahr < 2000;$$

Da Attribute nicht wie bei SQL über ihren Namen, sondern über ihre Position im Literal identifizert werden, können die Namen der einzelnen Variablen beliebig geändert werden. Diese Eigenschaft kann man nutzen, um einen Join kürzer zu formulieren:

$$q(titel, jahr, genre, adresse) :- film(titel, regisseur, jahr, genre, studioID),$$
$$studio(studioID, adresse), jahr < 2000;$$

Ein Join erfolgt also über die Verwendung der gleichen Variablen; hier wird über den Namen des Studios gejoint, der in $studioID$ umbenannt wurde. Nachfolgend wird die Möglichkeit der Umbenennung stets genutzt, um lange Attributnamen abzukürzen.

Für Datalog-Anfragen werden jetzt einige Begriffe definiert, die in der weiteren Arbeit, insbesondere in Kapitel 3, gebraucht werden:

Definition 1. *Eine Datalog-Anfrage p ist von der Form:*

$$p(X_1, ..., X_m) :- q_1(A_{11}, ..., A_{1m_1}), ..., q_n(A_{n1}, ..., A_{nm_n}), c_1, ..., c_l$$

Dann ist:

- $exp(p) = \{X_1, ..., X_m\}$*, die Menge der exportierten Variablen von* p*,*
- $var(p)$ *die Menge der Variablen von* p*,*
- $const(p)$ *die Menge der Konstanten von* p*,*
- $cond(p) = \{c_1, ..., c_l\}$*, die Menge der Bedingungen von* p*,*
- $literale(p) = \{q_1, ..., q_n\}$*, die Menge der Literale von* p*,*
- $|p| = n$*, die Länge von* p *(Anzahl der Literale).*

Datalog, welches Negationen im Rumpf zulässt, kann noch weitere Operationen, wie etwa Vereinigung und Mengendifferenz ausdrücken und ist damit äquivalent zur Relationenalgebra [KE06]. Wenn man die Möglichkeit zur Rekursion nicht ausschließt, ist es sogar noch ausdrucksstärker. Je ausdrucksstärker jedoch die Anfragesprache ist, desto komplexer wird die ohnehin schon sehr komplexe Anfrageplanung. Daher werden in dieser Arbeit weder Rekursionen noch Negationen oder andere Operationen als Joins erlaubt. Zugelassen sind also nur SPJ-Anfragen. Diese Teilmenge von Datalog wird auch als konjunktive Anfragen bezeichnet [LN07].

2.3 Paradigmen zur Zusammenführung von Schemata

Integrierte Datenbanken stellen ein einziges (gedachtes) Schema zur Verfügung, an das alle Anfragen gestellt werden [Ull97]. Eine wichtige Entwurfsentscheidung ist, wie die Quellen so zusammengeführt werden, dass sie einheitlich angesprochen werden können. Dafür gibt es die zwei grundsätzlichen Herangehensweisen Local-as-View (LaV) und Global-as-View (GaV) [Len02], die in den nachfolgenden zwei Abschnitten erklärt werden.

Im dritten Abschnitt wird noch kurz auf eine Mischform der beiden Paradigmen, Global-Local-as-View (GLaV), eingegangen.

2.3.1 Global-as-View

Bei GaV wird jede Relation des globalen Schemas als Sicht auf die Datenquellen definiert. Gibt es zum Beispiel die zwei globalen Relationen

```
Hersteller:  Herstellernummer, Ort
Fahrzeug:    Herstellernummer, Name, Baujahr
```

in einem globalen Schema, sowie die drei Quellen

```
Q1:  Herstellernummer, Name, Ort
Q2:  Name, Herstellernummer, Baujahr
Q3:  Herstellernummer, Baujahr, Motor
```

so ist die Quelle `Q1` die einzige Quelle, die Informationen der globalen Relation `Hersteller` enthalten kann. Die Sicht für `Hersteller` sähe dann wie folgt aus:

```
CREATE VIEW Hersteller AS
SELECT Herstellernummer, Ort
FROM Q1;
```

Es werden die `Herstellernummer` und der `Ort` aus der Quelle selektiert. Der `Name` wird hier nicht benötigt, da er in der globalen Relation `Hersteller` nicht vorkommt.

Die Relation `Fahrzeug` kann durch `Q2` komplett oder aus einer Kombination von `Q1` und `Q3` berechnet werden. Eine GaV-Sicht für die Relation würde also wie folgt definiert werden:

```
CREATE VIEW Fahrzeug AS
SELECT Herstellernummer, Name, Baujahr
FROM Q2
UNION
SELECT Q1.Herstellernummer, Q1.Name, Q3.Baujahr
FROM Q1,Q3
WHERE Q1.Herstellernummer = Q3.Herstellernummer;
```

Die gesamte Sicht ergibt sich also aus der Vereinigung aller relevanten Quellen, wobei wieder nur die benötigten Attribute, also `Herstellernummer`, `Name` und `Baujahr`, projiziert werden.

Die Anfragebearbeitung bei GaV ist im Vergleich zu LaV sehr einfach. Wird eine Anfrage an das globale Schema gestellt, kann für jede Relation, die in der Anfrage vorkommt, die erstellte Sicht der Relation substituiert werden. Substitutionen und spätere Vereinfachungen (siehe unten) werden üblicherweise im relationen-algebraischen Term und nicht in SQL durchgeführt. Im folgenden Beispiel wird allerdings auf SQL zurückgegriffen. Die Beispielanfrage lautet „Welche Fahrzeuge (Herstellernummer, Name) haben einen Namen der mit M oder einem im Alphabet später vorkommenden Buchstaben beginnt“ und wird in SQL wie folgt formuliert:

```
SELECT Herstellernummer, Name FROM Fahrzeug WHERE Name > 'M';
```

In der Beispielanfrage kommt nur die Relation `Fahrzeug` vor. Die oben definierte Sicht der Relation kann nun direkt, also eins zu eins eingesetzt werden:

```
SELECT Herstellernummer, Name
FROM (
   SELECT Herstellernummer, Name, Baujahr
   FROM Q2
   UNION
   SELECT Q1.Herstellernummer, Q1.Name, Q3.Baujahr
   FROM Q1, Q3
   WHERE Q1.Herstellernummer = Q3.Herstellernummer
) AS Fahrzeug
WHERE Name > 'M';
```

Die Bearbeitung erfolgt konzeptionell von innen nach außen. Dabei werden die Zwischenergebnisse in temporären Tabellen gespeichert. Tatsächlich besteht hier aber ein großes Optimierungspotential durch Umschreiben der Anfrage, um so kleinere Teilergebnisse zu erhalten. Denkbar sind Verschiebungen von Selektionen und Projektionen, Entschachtelung der Unteranfragen und Änderungen bei Joins.

Im Beispiel oben könnte durch das Verschieben der Projektion nach Innen das Attribut `Baujahr` entfernt werden, da es nicht selektiert wird. Somit kann der Join zwischen `Q2` und `Q3` eingespart werden, da die Quelle 3 nur in der Sicht enthalten ist, um das `Baujahr` zu ergänzen. Die Selektion kann ebenfalls direkt an den Quellen durchgeführt werden, um weniger Tupel anzufragen. Die optimierte GaV-Anfrage sähe in SQL wie folgt aus:

```
SELECT Herstellernummer, Name
FROM Q2
WHERE Name > 'M'
UNION
SELECT Herstellernummer, Name
FROM Q1
WHERE Name > 'M';
```

Ohne Optimierungen ist die GaV-Anfragebearbeitung sehr einfach, da einmal erstellte Sichten nur noch substituiert werden müssen. Schwieriger wird es, wenn Optimierungstechniken, wie oben gezeigt, angewendet werden sollen. Aber dies ist nicht der einzige Nachteil: Da es meistens eine Vielzahl von Quellen gibt, werden die Sichten schnell sehr groß; kommt eine Quelle hinzu, müssen unter Umständen alle Sichten geändert werden, um die Quelle sinnvoll der bestehenden Integration hinzuzufügen. Wird eine Quelle entfernt, müssen ebenfalls alle Sichten daraufhin geprüft werden, ob sie die Quelle benutzen. Ist dies der Fall, müssen alle betroffenen Sichten umgeschrieben werden.

Außerdem kann es mit Nebenbedingungen zu Problemen kommen. Existieren Nebenbedingungen im globalen Schema, kann das dazu führen, dass Quellen aus der Integration ausgeschlossen werden, wenn sich die Bedingung nicht überprüfen lässt. Dies ist dann der Fall, wenn sich die Bedingung auf ein Attribut bezieht, welches von der Quelle nicht exportiert wird.

In dieser Arbeit wird die Anfragebearbeitung nach dem Local-as-View-Paradigma durchgeführt, welches sehr viel flexibler und robuster ist, wenn es um das Hinzufügen und Entfernen von Quellen geht - auf Kosten der Einfachheit bei der Anfragebearbeitung.

2.3.2 Local-as-View

Local-as-View (LaV)[7] betrachtet die Quellen als Sichten auf ein (gedachtes) globales Schema. Diese Definition hat ihre Berechtigung durch die Tatsache, dass eine Quelle als ein Ausschnitt aus einer idealen integrierten Gesamtdatenbank angesehen werden kann. Bildlich kann man sich LaV klar machen, wenn man als Ausgangspunkt eine vollständige Datenbank annimmt, von der es mehrere materialisierte Sichten gibt. Wenn man sich nun vorstellt, dass die Datenbank verloren geht, etwa durch einen Systemabsturz, und nur die Sichten übrig bleiben, ist die es Aufgabe der Anfragebearbeitung, nur mit Hilfe der noch verbliebenden Sichten, eine gegebene Anfrage so gut wie möglich zu beantworten. Dabei kommt es unter Umständen zu unvollständigen Ergebnissen (vgl. Abschnitt 2.1.1).

Es seien wieder das gleiche globale Schema mit den Relationen `Hersteller` und `Fahrzeug` wie oben gegeben. Ebenfalls gegeben seien die zum obigen Beispiel leicht

[7] Der hier vorgestellte Teil von LaV, das Umschreiben der Anfrage mit Hilfe gegebener Sichten, ist auch bekannt unter dem Namen „Answering Queries Using Views“ (AQUV) [Hal01].

modifizierten Quellen `Q1`, `Q2` und `Q3`:

```
Hersteller:  Herstellernummer, Ort
Fahrzeug:    Herstellernummer, Name, Baujahr
Q1:          Herstellernummer, Name, Ort
Q2:          Name, Herstellernummer, Baujahr
Q3:          Herstellernummer, Baujahr
```

Im Unterschied zu GaV wird jetzt für jede Quelle eine einzelne Sicht erstellt, und nicht für jede globale Relation. Für die Quelle `Q1` ist eine Sicht über die beiden Relationen `Fahrzeug` und `Hersteller` zu erstellen, die über das Attribut `Herstellernummer` gejoint werden:

```
CREATE VIEW S1 AS
SELECT F.Herstellernummer, F.Name, H.Ort
FROM Fahrzeug F, Hersteller H
WHERE F.Herstellernummer = H.Herstellernummer;
```

Die Quelle 2 beinhaltet nur Attribute aus der Relation `Fahrzeug` und ist daher sehr einfach aufgebaut:

```
CREATE VIEW S2 AS
SELECT Name, Herstellernummer, Baujahr
FROM Fahrzeug;
```

Die Sicht zu Quelle 3 ist ähnlich zu erstellen wie `S2`, doch sei hier noch eine Nebenbedingung der Quelle gegeben, die direkt in die Sicht übernommen werden kann. `Q3` enthalte hier keine Oldtimer (z. B. Baujahr > 1990), was sich wie folgt auf die Sicht auswirkt:

```
CREATE VIEW S3 AS
SELECT Herstellernummer, Baujahr
FROM Fahrzeug
WHERE Baujahr > 1990;
```

Die Anfragebearbeitung gestaltet sich bei LaV um einiges komplexer als bei GaV. Erstes Zwischenziel ist es, alle Sichten zu finden, die für eine globale Anfrage relevant sind. Relevant ist eine Sicht genau dann, wenn sie die Anfrage intensional gesehen komplett oder zum Teil beantworten kann.

Sei die globale Anfrage

```
SELECT F.Baujahr, H.Ort
FROM Fahrzeug F, Hersteller H
WHERE F.Herstellernummer=H.Herstellernummer;
```

gegeben. Offensichtlich kann keine Quelle diese Anfrage allein beantworten, da alle Quellen entweder das Attribut `Baujahr` oder das Attribut `Ort` enthalten. Die Anfrageplanung muss jetzt also diejenigen Quellen finden, die zumindest einen Teil der Anfrage beantworten können und sich mit einer anderen Quelle verbinden (joinen) lassen. In diesem Beispiel gibt es zwei Möglichkeiten, die Sichten so zu kombinieren, dass die Anfrage beantwortet werden kann. `Q1` enthält als einzige Quelle den `Ort`, muss also in Kombination jeweils mit `Q2` und `Q3` vorkommen. Die Verknüpfung der Sichten ist in diesem Beispiel problemlos über das Attribut `Herstellernummer` möglich. Die Anfrage wird also in die zwei folgenden Anfragen umgeschrieben:

```
-- Anfrage 1: S1, S2
SELECT S2.Baujahr, S1.Ort
FROM S1, S2
WHERE S1.Herstellernummer=S2.Herstellernummer;

-- Anfrage 2: S1, S3
SELECT S3.Baujahr, S1.Ort
FROM S1, S3
WHERE S1.Herstellernummer=S3.Herstellernummer;
```

Die Ergebnisse der beiden Anfragen müssen nach dem Ausführen noch vereinigt werden, damit das Gesamtergebnis der Ausgangsanfrage entspricht.

Leicht vorstellbar ist, dass es bei komplexeren Anfragen über mehr Relationen als zwei und einer größeren Anzahl zur Verfügung stehender Quellen, sehr schnell zu einer sehr großen Menge an Kombinationsmöglichkeiten kommen kann. Da in der Regel davon ausgegangen werden kann, dass alle Quellen unvollständig sind, reicht es nicht aus, nur eine Kombination zu finden. Erst durch die Vereinigung von möglichst vielen Kombinationen erhält man eine größtmögliche Vollständigkeit des Ergebnisses.

Dieser Schritt, das Umschreiben der Anfrage, soll automatisiert zur Laufzeit der Anfragebearbeitung vollzogen werden können. Dafür existieren unterschiedliche Algorithmen, die die Menge der relevanten Quellen sehr geschickt eingrenzen und somit die Zahl der Kombinationsmöglichkeiten früh reduzieren. Diesen Algorithmen widmet sich das Kapitel 3.

2.3.3 Global-Local-as-View

Global-Local-as-View (GLaV), auch Both-as-View genannt, ist eine Mischform der beiden oben erläuterten Paradigmen. GLaV führt die Anfrageplanung wie bei LaV durch, die Anfrageausführung wird hingegen nach GaV realisiert. Gegeben seien wieder die gleichen globalen Relationen aus den vorherigen Abschnitten. Zur Vereinfachung sei jetzt aber nur eine einzige Quelle `Q` gegeben:

```
Hersteller:  Herstellernummer, Ort
Fahrzeug:    Herstellernummer, Name, Baujahr
Q:           Herstellernummer, Baujahr, Motor
```

Die Anfrage, die an die integrierte Datenbank gestellt wird, lautet:

```
SELECT Herstellernummer, Baujahr FROM Fahrzeug;
```

Nun muss es für `Q` zwei Sichten geben: Eine GaV-Sicht, die die Relation `Fahrzeug` mittels `Q` umsetzt und eine LaV-Sicht, die `Q` als Sicht auf das globale Schema beschreibt. Die GaV-Sicht projiziert die benötigten Attribute aus der Quelle. In diesem Beispiel fällt das Attribut `Motor` weg. Die Sicht sieht dann wie folgt aus:

```
CREATE VIEW QQ_GaV AS
SELECT Herstellernummer, Baujahr
FROM Q;
```

Die zweite Sicht geht von der Quelle `Q` auf das globale Schema, wie bei LaV. Es sei wieder gegeben, dass die Quelle nur junge Fahrzeuge (Baujahr > 1990) enthalte:

```
CREATE VIEW QQ_LaV AS
SELECT Herstellernummer, Baujahr
FROM Fahrzeug
WHERE Baujahr > 1990;
```

Der große Vorteil von GLaV ist, dass sowohl globale als auch lokale Nebenbedingungen (wie zum Beispiel Baujahr > 1990) möglich sind.

Kapitel 3

Algorithmen zur Anfrageplanung

Wie in Abschnitt 2.1.1 beschrieben, ist es die Aufgabe eines Anfrageplanungsalgorithmus, automatisiert aus einer globalen Anfrage und gegebenen Sichten, einen oder mehrere Anfragepläne zu erstellen. Dieses Problem wird auch Answering Queries Using Views (AQUV) genannt [Hal01]. Dabei werden Anfragen so umgeschrieben, dass sie nicht mehr von den Relationen des globalen Schemas, sondern von den gegebenen Sichten bestmöglich beantwortet werden.

In diesem Kapitel werden verschiedene Anfrageplanungsalgorithmen vorgestellt. Zuerst werden anhand des Generate-and-Test-Algorithmus die zum Verständnis erforderlichen Begriffe des Containment Mappings und des minimalen Anfrageplans eingeführt und es wird ein naiver Lösungsansatz aufgezeigt.

In den danach folgenden Abschnitten werden wesentlich effizientere Algorithmen, die den Suchraum stark reduzieren und somit Geschwindigkeitsvorteile haben, vorgestellt. Alle Algorithmen entspringen immer den angegebenen Quellen, werden aber teilweise in modifizierter und optimierter Form angegeben.

Das Kapitel endet mit dem Vergleich und der Bewertung der vorgestellten Algorithmen. Es werden Algorithmen genannt, die im weiteren Verlauf der Arbeit eingesetzt werden.

3.1 Generate-and-Test-Algorithmus

Die Beschreibung des hier vorgestellten Generate-and-Test-Algorithmus (GTA) ist inspiriert durch [Les00]. Es handelt sich dabei um einen einfachen Algorithmus, der viele mögliche Anfragepläne aufzählt und für jeden testet, ob er die Anfrage beantworten kann.

Um diesen und die weiteren Anfrageplanungsalgorithmen zu verstehen, werden in den ersten beiden Teilen dieses Abschnitts der Begriff des Containment Mappings und minimale Anfragepläne definiert. Im letzten Abschnitt wird der GTA informell beschrieben und bewertet.

3.1.1 Containment Mapping

Eine Sicht kommt immer dann für die Beantwortung einer Anfrage benutzt werden, wenn ein Containment Mapping (siehe Definition 2) von der Anfrage zur Sicht existiert.

Definition 2. *(Containment Mapping) Ein Containment Mapping ist ein Mapping $h : q_1 \rightarrow q_2$, mit q_1, q_2 sind konjunktive Anfragen, bei dem Symbole aufeinander abgebildet werden und folgende Bedingungen gelten:*

1. $\forall c \in const(q_1) : h(c) = c$
   Alle Konstanten müssen auf sich selbst abbgebildet werden, also jede Konstante in q_1 muss auch in q_2 vorkommen.
2. $\forall l \in q_1 : h(l) \in q_2$
   Jedes Literal in q_1 muss auch in q_2 vorhanden sein. Umgekehrt gilt dies aber nicht: q_2 darf auch noch weitere Literale enthalten, die nicht in q_1 vorkommen.
3. $\forall v \in exp(q_1) : h(v) \in exp(q_2)$
   Alle exportierten Variablen von q_1 müssen auf eine exportierte Variable von q_2 projiziert werden.
4. $cond(q_2) \Rightarrow cond(h(q_1))$
   Die Bedingungen von q_2 müssen die Bedingungen von q_1 implizieren.

Anhand des folgenden Beispiels wird gezeigt, wie ein Containment Mapping konkret aussehen kann. Es sei die Anfrage `q` und die Quelle `s` gegeben (Variablen, die nur einmal in einer Anfrage vorkommen, werden durch _ ausgelassen) [LN07]:

```
s(T, S, K) :- film(T, _, R, _), spielt(T, S, 'Hauptrolle', K);
              q(X, Y, Z) :- spielt(X, Y, W, Z);
```

Für die Quelle `s` gilt $exp(s) = \{T, S, K\}$, $const(s) = \{'Hauptrolle'\}$ und $cond(s) = \{R =' Hauptrolle'\}$. Für die Anfrage `q` gilt $exp(q) = \{X, Y, Z\}$ und $const(q) = cond(q) = \emptyset$. Nachfolgend werden die einzelnen Punkte überprüft, die zutreffen müssen, damit ein Containment Mapping vorliegt:

1. Offensichtlich erfüllt, da in `q` keine Konstanten vorkommen.

2. Das einzige Literal, dass in `q` vorkommt ist `spielt`, welches ebenfalls in `s` vorhanden ist. Daher ist die zweite Bedingung ebenfalls erfüllt.

3. Diese Bedingung ist erfüllt mit dem Mapping `h: X` $\rightarrow$ `T, Y` $\rightarrow$ `S` und `Z` $\rightarrow$ `K`.

4. Trivialerweise auch erfüllt, da `q` keine Bedingungen enthält.

Bei diesem Beispiel liegt also eine Containment Mapping von der Anfrage `s` zur Quelle `q` vor. Die Quelle kann diese Anfrage also beantworten.

3.1.2 Minimale Anfragepläne

Wenn eine Quelle für eine Anfrage verwendet werden kann, ändert sich nichts am Ergebnis, wenn diese Quelle beliebig oft in einer Anfrage enthalten ist. Ein naiver Algorithmus könnte somit theoretisch unendlich viele Pläne erzeugen. Die Anfrageplanung, also alle korrekten Pläne zu finden, wäre dann aber nicht entscheidbar, da unendlich viele Pläne gefunden werden könnten. Außerdem wäre es nicht im Interesse der Anfrageausführung, an die Quellen unendlich viele Anfragen zu stellen. Eine sinnvolle Einschränkung ist die Länge der Anfragepläne. Für diese Einschränkung werden in Definition 3 minimale Anfragepläne definiert.

Definition 3. *(Minimaler Anfrageplan) Ein Anfrageplan p aus Sichten $V = \{v_1, ..., v_n\}$ heißt minimal, wenn es keinen Plan p' gibt mit Sichten V' für den gilt, dass $V' \subset V$ und $result(p) \subseteq result(p')$.*

Ein Plan ist genau dann minimal, wenn es keinen anderen Plan gibt, der weniger Sichten verwendet und trotzdem ein genauso vollständiges Ergebnis liefert. Ein minimaler Plan für eine Anfrage q enthält dabei maximal so viele Sichten, wie q Literale hat. Das liegt daran, dass es für einen Plan p und eine Anfrage q immer ein Containment Mapping h von q nach p gibt. Jedes Literal aus q hat dabei in h genau ein Zielliteral. Somit kann es höchstens $|q|$ verschiedene Zielliterale in p geben.

Alle Pläne, die mehr Sichten enthalten als q Literale hat, berechnen nur Tupel, die auch von einem kürzeren Plan berechnet werden könnten. Somit kommen nur noch endlich viele Pläne in Frage. Ziel der Anfrageplanungsalgorithmen ist also offensichtlich nur minimale Pläne zu erzeugen oder wenigstens nur Pläne zu finden, die höchstens so viele Sichten beinhalten, wie die Anfrage Literale hat.

3.1.3 Beschreibung und Bewertung

Das Vorgehen des GTA ist nun wie folgt: In einer ersten Phase werden zu einer gegebenen Anfrage q und einer gegebenen Menge Sichten V alle möglichen Kombinationen $p = \{v_1, ...\}$, mit $v_i \in V$ und $|p| \leq |q|$ gebildet. Dabei muss überprüft werden, ob die Joins aus q auch von p ausgeführt werden können. Das bedeutet, dass alle Literale der Anfrage sich in den Sichten eines Planes wiederfinden müssen und die Variablen über die gejoint wird, müssen von den Sichten exportiert werden. Anschließend wird in einer zweiten Phase für jedes p getestet, ob ein Containment Mapping von q nach p vorliegt.

Ist zum Beispiel $|q| = 3$ und $V = \{v_1, v_2, v_3\}$, dann werden die folgenden Pläne erstellt:

$$P = \{(v_1), (v_2), (v_3), (v_1, v_1), (v_1, v_2), ...(v_3, v_3), (v_1, v_1, v_1), (v_1, v_1, v_2), ...(v_3, v_3, v_3)\}$$

Die erstellten Pläne werden anschließend daraufhin untersucht, ob sie die Joins aus q ausführen und ob ein Containment Mapping existiert.

Dieses Vorgehen ist sehr teuer, da sehr viele überflüssige Kombinationen überprüft werden müssen. Es werden $|V|^{|q|}$ Pläne erstellt und für jeden Plan muss überprüft werden, ob ein Containment Mapping möglich ist. Diese Prüfung hat eine Komplexität von $\mathcal{O}(|p|^{|q|})$ [LN07].

Nachfolgend werden verschiedene Algorithmen erläutert, die die Komplexität reduzieren, indem sie nur Sichten betrachten, die überhaupt für die Anfrage geeignet sind. Dabei unterschreitet keiner der Algorithmen die exponentielle Komplexitätsklasse; alle Algorithmen haben aber wesentlich bessere Average-Case Laufzeiten als der GTA.

3.2 Bucket-Algorithmus

Der Bucket-Algorithmus (BA) [LRO96] war der erste Algorithmus zur Anfrageplanung. Seine Grundidee ist es, nur in Frage kommende Sichten überhaupt zu betrachten. Die Zahl der möglichen Kombinationen reduziert sich dadurch wesentlich. Er ist in zwei Phasen aufgeteilt. In der ersten Phase wird zu jedem Literal der Anfrage ein leerer Bucket erstellt. Dieser wird mit Sichten gefüllt, die für die Beantwortung der Anfrage relevant sein könnten. Im zweiten Schritt des Algorithmus werden die Inhalte der einzelnen Buckets kombiniert. Dabei wird überprüft, ob die dadurch entstehenden Teilmappings zueinander kompatibel sind. Ist dies nicht der Fall, können sie verworfen werden und es kann mit der nächsten Kombination fortgefahren werden.

Im ersten Teil dieses Abschnitts wird der BA formal erklärt und anhand von Pseudo-Codes angegeben. Danach folgen zwei Beispiele, die die Anwendung des Algorithmus zeigen. Dabei wird das zweite Beispiel einige Schwachpunkte des BA offenbaren. Im letzten Teil wird der Algorithmus auf Vollständigkeit, Korrektheit und Komplexität untersucht und bewertet.

3.2.1 Formale Beschreibung

In der ersten Phase des BA wird für jedes Literal der Anfrage ein leerer Bucket erstellt. Dieser wird dann mit in Frage kommenden Sichten gefüllt. Wenn eine Sicht das Literal des Buckets enthält und es ein Containment Mapping von der Anfrage auf die Sicht gibt, kommt die Sicht mit dem Mapping in den Bucket. Dabei müssen alle Variablennamen mit neuen Variablen belegt werden, damit sie eindeutig unterschieden werden können. Falls eine Sicht mehrmals das gleiche Literal beinhaltet, kommt die Sicht mit dem jeweils passenden Mapping auch mehrmals in den Bucket. Dieser erste Teil des BA, die Erstellung der Buckets, ist im Algorithmus 1 dargestellt.

In Zeile 7 des angegebenen Algorithmus werden zwei Fakten überprüft, die erfüllt sein müssen, damit eine Sicht in den Bucket kommt. Zum einen muss das aktuell betrachtete Literal der Sicht das gleiche Literal sein, für dass der Bucket erstellt wird. Zum anderen müssen die Bedingungen der Sicht mit den Bedingungen der Anfrage kompatibel sein. Ist dies nicht der Fall, kann die Sicht keine Tupel zum Gesamtergebnis beitragen.

Algorithmus 1 Phase 1 des Bucket-Algorithmus: Buckets erstellen

input: Eine Menge von konjunktiven Quellendefinitionen $\mathcal{V}$.
input: Eine konjunktive Anfrage Q.
output: Eine Menge von Buckets $\mathcal{B}$

```
1:  function CREATEBUCKETS(V, Q)
2:      B ← ∅
3:      for Literale l ∈ Q do
4:          Bucket ← ∅
5:          for all V ∈ V do
6:              for Literale k ∈ V do
7:                  if l = k ∧ cond(V) kompatibel mit cond(Q) then
8:                      if ein partielles Mapping h der Variablen aus l → V existiert then
9:                          Benutze für h unbenutzte Variablen
10:                         Bucket ← Bucket ∪ (V, h)
11:                     end if
12:                 end if
13:             end for
14:         end for
15:         B ← B ∪ Bucket
16:     end for
17:     return B
18: end function
```

Die Zeile 8 überprüft, ob ein partielles Mapping von den Variablen des Literals der Anfrage zur Sicht existiert. Dies kann nur der Fall sein, wenn die Sicht mindestens die Variablen exportiert, die von Q exportiert werden und diese im aktuell betrachteten Literal von Q enthalten sind.

Sind diese Bedingungen erfüllt, wird die Sicht mit dem erstellten Mapping in den Bucket übernommen. Für die Variablen in der Sicht, werden dabei für das Mapping unbenutzte Variablennamen vergeben, um eine Verwechslung mit existierenden Variablen auszuschließen. Wenn die Anfrage immer unterschiedliche Variablen benutzen würde als die Sichten, wäre diese Umbenennung überflüssig. Da dies aber nicht vorausgesetzt wird, erfolgt diese Umbenennung.

Die zweite Phase des BA erstellt die Anfragepläne mit Hilfe der zuvor generierten Buckets. Dafür werden die Inhalte der Buckets kombiniert, sodass aus jedem Bucket genau eine Sicht verwendet wird. Bei zum Beispiel 2 Buckets mit einmal 3 und einmal 4 Inhalten ergeben sich $3 \cdot 4 = 12$ mögliche Anfragepläne, die auf ihre Kompatibilität überprüft werden.

Die Inhalte der Buckets bestehen, wie in der ersten Phase beschrieben, immer aus einer relevanten Sicht v und einem passenden Mapping h. Immer wenn (v, h) aus einem Bucket einem Teilplan $P_{gesamt} = (V_{gesamt}, h_{gesamt})$ hinzugefügt werden soll, wird überprüft, ob die beiden Mappings kompatibel sind. Kompatibel sind zwei Mappings

dann, wenn keine Variable auf mehrere unterschiedliche Variablen gemappt wird.

Ist (v, h) mit P_{gesamt} kompatibel, kann $V_{gesamt} = V_{gesamt} \cup \{v\}$ und $h_{gesamt} = h_{gesamt} \cup \{h\}$ gebildet werden und es kann mit dem nächsten (v, h) aus dem nächsten Bucket fortgefahren werden.

Häufig ist die Kompatibilität zwischen dem aktuell betrachteten (v, h) und dem bestehenden Plan P_{gesamt} nicht sofort erfüllt. Sehr oft kommt es zu einer solchen Situation: Seien z.B. die zwei Mappings $\{A \rightarrow Z, B \rightarrow Y\}$ und $\{A \rightarrow M, C \rightarrow X\}$ gegeben, die auf Kompatibilität überprüft werden. Zunächst gibt es den Widerspruch, dass A einmal auf Z und einmal auf M gemappt werden soll. Dieser Konflikt kann gelöst werden, indem die Bedingung $Z = M$ dem Plan hinzugefügt wird. Damit diese Bedingung in der späteren Ausführung auch überprüft werden kann, müssen allerdings beide Variablen in ihren Sichten exportiert werden. Ist dies der Fall, können V_{gesamt} und h_{gesamt}, wie oben, gebildet werden.

Ein Plan P ist also eine heterogene Menge bestehend aus Sichten, dazugehörigen Mappings und Verbundbedingungen und wird notiert als

$$P = \{(v_1, h_1), (v_2, h_2), ..., X_1 = Y_1, ...\}.$$

Die in Algorithmus 2 angegebene zweite Phase des Bucket-Algorithmus arbeitet leicht unterschiedlich zu der obigen Beschreibung. Anstatt jeden Plan nacheinander zusammenzusetzen, werden in jeder Iteration alle Pläne gleichzeitig um ein (v, h) erweitert. Tritt bereits am Anfang eines Planes eine Inkompatibilität auf, wird er aus der Menge der möglichen Pläne entfernt und nachfolgende Kombinationsmöglichkeiten werden nicht mehr betrachtet.

3.2.2 Beispiel Vorlesungsdatenbank

Das folgende Beispiel ist angelehnt an [Hal01] und zeigt wie der BA arbeitet. Das nachstehende globale Schema mit den Relationen `lehrt`, `eingeschrieben` und `kurs` einer Vorlesungsdatenbank sei gegeben:

`lehrt`(P, K, S)	Ein Professor[1] P lehrt einen Kurs K in dem Semester S
`eingeschrieben`(H, K, S)	Ein Hochschüler H ist im Semester S im Kurs K eingeschrieben
`kurs`(K, T)	Der Kurs mit der Id K hat den Titel T

Außerdem gegeben seien die folgenden vier Sichten:

[1] Die in dieser Arbeit aus Gründen der besseren Lesbarkeit gewählte männliche Form schließt die weibliche immer mit ein.

Algorithmus 2 Phase 2 des Bucket-Algorithmus: Buckets kombinieren

input: Eine Menge von Buckets $\mathcal{B}$ erzeugt von der ersten Phase des Algorithmus
output: Eine Menge von korrekten Anfrageplänen $\mathcal{P}$

```
1:  function COMBINEBUCKETS(B)
2:      P ← ∅
3:      for all B ∈ B do
4:          if P = ∅ then
5:              for all (v, h) ∈ B do
6:                  P ← P ∪ (v, h)
7:              end for
8:          else
9:              for all P = (V_gesamt, h_gesamt) ∈ P do
10:                 for all (v, h) ∈ B do
11:                     if (v, h) kompatibel mit P then
12:                         P ← P ∪ {(v, h)}
13:                         P ← P ∪ P
14:                     else if inkompatibel wegen der exportierten Variablen
15:                             X ∈ v und Y ∈ V_gesamt then
16:                         P ← P ∪ {(v, h)} ∪ {X = Y}
17:                         P ← P ∪ P
18:                     else
19:                         P ← P \ P
20:                     end if
21:                 end for
22:             end for
23:         end if
24:         if P = ∅ then
25:             return ∅
26:         end if
27:     end for
28:     return P
29: end function
```

$$
\begin{array}{lll}
\texttt{v1}(\texttt{H},\texttt{T},\texttt{S},\texttt{K}) & \texttt{:-} & \texttt{eingeschrieben}(\texttt{H},\texttt{K},\texttt{S}),\texttt{kurs}(\texttt{K},\texttt{T}),\\
 & & \texttt{K} \geq 500, \texttt{S} \geq \texttt{WS98};\\
\texttt{v2}(\texttt{H},\texttt{P},\texttt{S},\texttt{K}) & \texttt{:-} & \texttt{eingeschrieben}(\texttt{H},\texttt{K},\texttt{S}),\texttt{lehrt}(\texttt{P},\texttt{K},\texttt{S}),\\
 & & \texttt{K} \leq 400;\\
\texttt{v3}(\texttt{H},\texttt{K}) & \texttt{:-} & \texttt{eingeschrieben}(\texttt{H},\texttt{K},\texttt{S}),\texttt{S} \leq \texttt{WS94};\\
\texttt{v4}(\texttt{P},\texttt{K},\texttt{T},\texttt{S}) & \texttt{:-} & \texttt{lehrt}(\texttt{P},\texttt{K},\texttt{S}),\texttt{kurs}(\texttt{K},\texttt{T}),\\
 & & \texttt{eingeschrieben}(\texttt{H},\texttt{K},\texttt{S}),\texttt{S} \leq \texttt{WS97};
\end{array}
$$

Die Anfrage, die an die integrierte Datenbank gestellt werden soll, lautet: „Welche Hochschüler H waren im Kurs mit der Id K ($\geq$ 300) beim Lehrenden P im oder nach dem WS95 eingeschrieben?“ und wird wie folgt in Datalog formuliert:

$$\texttt{q}(\texttt{H},\texttt{K},\texttt{P}) : -\texttt{lehrt}(\texttt{P},\texttt{K},\texttt{S}),\texttt{eingeschrieben}(\texttt{H},\texttt{K},\texttt{S}),\texttt{kurs}(\texttt{K},\texttt{T}),\texttt{S} \geq \texttt{WS95},\texttt{K} \geq 300;$$

In seiner ersten Phase erstellt der BA jetzt drei Buckets, für die drei Literale `lehrt`, `eingeschrieben` und `kurs` in der Anfrage. Dabei kommen in die Buckets genau die Sichten mit Mapping, die daselbe Literal enthalten und bei denen die zugehörigen Bedingungen zu keinem Konflikt führen. Die partiellen Mappings sind dabei immer von dem jeweils betrachteten Literal der Anfrage auf die Sicht. Beim Literal `lehrt` müssen also die Variablen $\texttt{P},\texttt{K}$ und $\texttt{S}$ auf die Variablen der Sichten gemappt werden können. Die drei erzeugten Buckets sind:

- $Bucket_1$ (`lehrt`): (`v2`, $\texttt{P} \to \texttt{P}_1, \texttt{K} \to \texttt{K}_1, \texttt{S} \to \texttt{S}_1$), (`v4`, $\texttt{P} \to \texttt{P}_1, \texttt{K} \to \texttt{K}_1, \texttt{S} \to \texttt{S}_1$)
  `v1` und `v3` kommen hier nicht in den Bucket, weil sie kein Literal `lehrt` enthalten.

- $Bucket_2$ (`eingeschrieben`): (`v1`, $\texttt{H} \to \texttt{H}_2, \texttt{K} \to \texttt{K}_2, \texttt{S} \to \texttt{S}_2$), (`v2`, $\texttt{H} \to \texttt{H}_2, \texttt{K} \to \texttt{K}_2, \texttt{S} \to \texttt{S}_2$)
  Bei der Sicht `v3` gibt es einen Konflikt mit der Bedingung für `S`: Die Anfrage fordert $\texttt{S} \geq \texttt{WS95}$, die Sicht kann nur $\texttt{S} \leq \texttt{WS94}$ liefern. Da es keine extensionale Überschneidung gibt, ist die Sicht unbrauchbar, um die Anfrage zu beantworten.
  Die Sicht `v4` ist ebenfalls nicht im Bucket. Sie exportiert `H` nicht und ist daher nicht in der Lage, das Literal `eingeschrieben` zu beantworten.

- $Bucket_3$ (`kurs`): (`v1`, $\texttt{K} \to \texttt{K}_3, \texttt{T} \to \texttt{T}_3$), (`v4`, $\texttt{K} \to \texttt{K}_3, \texttt{T} \to \texttt{T}_3$)
  In diesen Bucket kommen nur `v1` und `v4`, weil die anderen beiden Sichten kein Literal `kurs` besitzen.

Nachdem die erste Phase des BA abgeschlossen ist, bekommt die zweite Phase die erzeugten Buckets übergeben und erstellt daraus Anfragepläne. Dafür werden die Inhalte der Buckets miteinander iterativ kombiniert. Immer wenn eine neue Sicht mit Mapping zu einem bestehenden Teilplan hinzugefügt werden soll, wird überprüft, ob sie kompatibel sind. Ist dies nicht der Fall und sie können auch nicht kompatibel gemacht werden, kann die Kombination abgebrochen und mit der nächsten fortgefahren werden. In diesem Beispiel würde also etwa mit der Kombination von `v2`, `v1`, `v1` begonnen werden und es müssten $2^3 = 8$ Kombinationen überprüft werden, da es 3 Buckets gibt und in jedem 2 Elemente sind.

Bei der Kombination von v2, v1, v1 werden zuerst (v2, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1) und (v1, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2) auf Kompatibilität überprüft. Offensichtlich sind die beiden Mappings nicht kompatibel, da es einen Konflikt mit K gibt. Einmal wird K $\rightarrow$ K_1 und einmal K $\rightarrow$ K_2 gemappt. Die gleiche Situation tritt bei S auf. Doch dieses Problem wäre noch reparabel, da sowohl K als auch S in beiden Sichten exportierte Variablen sind. Man könnte also die Bedingung $S_1 = S_2$ mit in das Mapping aufnehmen. Doch bei der Bedingung $K_1 = K_2$ scheitert dieses Vorhaben, da $K_1 \geq 500$ und $K_2 < 400$ in den Sichten festgelegt ist. Die Quellen enthalten also gar keine überschneidenden Daten. Diese Kombination ist daher nicht möglich und es kann mit der nächsten fortgefahren werden.

Die Kombination v2, v1, v4 braucht gar nicht erst überprüft werden, da sie wie die erste mit den Sichten v2 und v1 beginnt und aus den gleichen Gründen scheitern würde.

Die nächste Kombination ist v2, v2, v1. Auch hier kommen wieder v2 und v1 vor und man kann prädizieren, dass diese Kombination wieder aus dem selben Grund nicht möglich sein wird. Der BA würde jedoch zu erst probieren (v2, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1) und (v2, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2) zu kombinieren. Hier ist zunächst eine Inkompatibilität gegeben, da K einmal auf K_1 und einmal auf K_2 gemappt wird, ebenso wie mit S. K und S werden aber exportiert und diesmal gibt es auch keine Probleme mit den Bedingungen, da es sich um die selbe Quelle handelt, sind natürlich auch die Daten die selben. Es würde nun also der Teilplan {(v2, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1), (v2, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2), $K_2 = K_1$, $S_2 = S_1$} erstellt werden.
Jetzt erst würde v1 hinzugenommen werden und die Kombination würde wegen der Bedingungen $K_2 < 400$ und $K_3 \geq 500$ verworfen werden.

Nun wird die Kombination v2, v2, v4 betrachtet. Wie bei der letzten Kombination auch, werden v2 und v2 erfolgreich zusammengefasst und man erhält den Teilplan {(v2, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1), (v2, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2), $K_2 = K_1$, $S_2 = S_1$}. Jetzt würde (v4, K $\rightarrow$ K_3, T $\rightarrow$ T_3) dazukommen. Offensichtlich ist diese Kombination wieder nicht direkt kompatibel, aber K wird exportiert und es kann daher $K_3 = K_2 = K_1$ sichergestellt werden. Diese Kombination ist die Erste, die funktioniert. Der Anfrageplan sähe dann wie folgt aus: {(v2, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1), (v2, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2), (v4, K $\rightarrow$ K_3, T $\rightarrow$ T_3), $K_3 = K_2 = K_1$, $S_2 = S_1$}.

Anschließend ist die Kombination v4, v1, v1 an der Reihe. Auch hier tritt ein ähnliches Problem auf wie bei den ersten drei Kombinationen: Es scheitert an den Bedingungen der Sichten. v4 enthält nur Daten mit S $\leq$ WS97, während v1 nur mit S $\geq$ WS98 dienen kann. Diese beiden Sichten sind daher nicht kompatibel. Aus dem selben Grund wird auch die danach kommende Kombination v4, v1, v4 scheitern.

Jetzt betrachtet der BA die vorletzte Kombination v4, v2, v1. (v4, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1) und (v2, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2) können kompatibel gemacht werden mit $K_1 = K_2$ und $S_1 = S_2$. Es kann also der Teilplan {(v4, P $\rightarrow$ P_1, K $\rightarrow$ K_1, S $\rightarrow$ S_1), (v2, H $\rightarrow$ H_2, K $\rightarrow$ K_2, S $\rightarrow$ S_2)} erstellt werden. Allerdings kann v1 nicht hinzugefügt werden, weil wieder wie oben die Bedingungen für K widersprüchlich sind.

Die letzte Kombination $\texttt{v4}, \texttt{v2}, \texttt{v4}$ ist die zweite und letzte kompatible Kombination. Wie bei der Vorangegangenen wird $\{(\texttt{v4}, \texttt{P} \rightarrow \texttt{P}_1, \texttt{K} \rightarrow \texttt{K}_1, \texttt{S} \rightarrow \texttt{S}_1), (\texttt{v2}, \texttt{H} \rightarrow \texttt{H}_2, \texttt{K} \rightarrow \texttt{K}_2, \texttt{S} \rightarrow \texttt{S}_2)\}$ gebildet. Jetzt soll $(\texttt{v4}, \texttt{K} \rightarrow \texttt{K}_3, \texttt{T} \rightarrow \texttt{T}_3)$ dazu kommen, was mit $\texttt{K}_1 = \texttt{K}_2 = \texttt{K}_3$ auch funktioniert. Der gebildete Gesamtplan ist dann $\{(\texttt{v4}, \texttt{P} \rightarrow \texttt{P}_1, \texttt{K} \rightarrow \texttt{K}_1, \texttt{S} \rightarrow \texttt{S}_1), (\texttt{v2}, \texttt{H} \rightarrow \texttt{H}_2, \texttt{K} \rightarrow \texttt{K}_2, \texttt{S} \rightarrow \texttt{S}_2), (\texttt{v4}, \texttt{K} \rightarrow \texttt{K}_3, \texttt{T} \rightarrow \texttt{T}_3), \texttt{K}_3 = \texttt{K}_2 = \texttt{K}_1, \texttt{S}_2 = \texttt{S}_1\}$.

Damit ist die zweite Phase des BA abgeschlossen. Zur Erinnerung noch einmal die Anfrage, die an das globale Schema gestellt wurde:

$$\texttt{q}(\texttt{H}, \texttt{K}, \texttt{P}) : -\texttt{lehrt}(\texttt{P}, \texttt{K}, \texttt{S}), \texttt{eingeschrieben}(\texttt{H}, \texttt{K}, \texttt{S}), \texttt{kurs}(\texttt{K}, \texttt{T}), \texttt{S} \geq \texttt{WS95}, \texttt{K} \geq 300;$$

Die Anfrage muss noch umgeschrieben werden, damit sie von den Quellen beantwortet werden kann. Aus der kompatiblen gefundenen Kombination $\texttt{v2}, \texttt{v2}, \texttt{v4}$ wird die Anfrage:

$$\begin{aligned}\texttt{q}'(\texttt{H}_2, \texttt{K}_1, \texttt{P}_1) \texttt{ :- } & \texttt{v2}(\_, \texttt{P}_1, \texttt{S}_1, \texttt{K}_1), \texttt{v2}(\texttt{H}_2, \_, \texttt{S}_2, \texttt{K}_2), \texttt{v4}(\_, \texttt{K}_3, \texttt{T}_3, \_), \\ & \texttt{K}_3 = \texttt{K}_2 = \texttt{K}_1, \texttt{S}_2 = \texttt{S}_1, \texttt{S}_1 \geq \texttt{WS95}, \texttt{K}_1 \geq 300;\end{aligned}$$

Die zweite gefundene Kombination $\texttt{v4}, \texttt{v2}, \texttt{v4}$ wird zur Anfrage:

$$\begin{aligned}\texttt{q}''(\texttt{H}_2, \texttt{K}_1, \texttt{P}_1) \texttt{ :- } & \texttt{v4}(\texttt{P}_1, \texttt{K}_1, \_, \texttt{S}_1), \texttt{v2}(\texttt{H}_2, \_, \texttt{S}_2, \texttt{K}_2), \texttt{v4}(\_, \texttt{K}_3, \texttt{T}_3, \_), \\ & \texttt{K}_3 = \texttt{K}_2 = \texttt{K}_1, \texttt{S}_2 = \texttt{S}_1, \texttt{S}_1 \geq \texttt{WS95}, \texttt{K}_1 \geq 300;\end{aligned}$$

Die beiden erzeugten Anfragen sind sehr ähnlich, aber nicht gleich. Bei $\texttt{q}'$ wird der Professor `P` aus der Sicht `v2` abgefragt, die Semesterangabe `S` in `v4` wird ignoriert. Bei $\texttt{q}''$ wird der Professor aus der Sicht `v4` abgerufen.

Keine der beiden Anfragen ist besser oder schlechter als die andere. Da man grundsätzlich von heterogenen Datenbeständen ausgeht, kann man nur ein möglichst vollständiges Ergebnis erhalten, wenn man in der Phase der Anfrageausführung beide Anfragen ausführt und in der Phase der Ergebnisintegration beide Ergebnisse vereint.

Eine sinnvolle und naheliegende Optimierung des BA ist es, Sichten, die mehrmals in einem Anfrageplan vorkommen, auf eins zu beschränken. Dadurch würden Quellen nicht doppelt angefragt werden. Diese Optimierung wird der MiniCon-Algorithmus, der in Abschnitt 3.4 eingeführt wird, beinhalten; der BA in der hier vorgestellten Form wird die Anzahl doppelter Sichten nicht reduzieren.

3.2.3 Beispiel Literaturdatenbank

Zum BA wird jetzt ein zweites Beispiel angegeben. Es ist aus [PH01] übernommen und führt vor, welche Schwächen der BA hat. Das folgende globale Schema mit den Relationen `cites` und `sameTopic` einer Literaturdatenbank sei gegeben:

$\texttt{cites}(\texttt{X}, \texttt{Y})$	Literatur `X` zitiert Literatur `Y`
$\texttt{sameTopic}(\texttt{A}, \texttt{B})$	Literatur `A` und Literatur `B` haben das selbe Thema

Des Weiteren seien die folgenden Sichten gegeben:

$$\texttt{v1}(\texttt{A}) : -\texttt{cites}(\texttt{A},\texttt{B}), \texttt{cites}(\texttt{B},\texttt{A});$$
$$\texttt{v2}(\texttt{C},\texttt{D}) : -\texttt{sameTopic}(\texttt{C},\texttt{D});$$
$$\texttt{v3}(\texttt{F},\texttt{H}) : -\texttt{cites}(\texttt{F},\texttt{G}), \texttt{cites}(\texttt{G},\texttt{H}), \texttt{sameTopic}(\texttt{F},\texttt{G});$$

Die zu untersuchende Anfrage an das globale Schema lautet „Zu welcher Literatur X gibt es eine Literatur Y, die sich gegenseitig zitieren und dasselbe Thema haben?“:

$$\texttt{q}(\texttt{X}) : -\texttt{cites}(\texttt{X},\texttt{Y}), \texttt{cites}(\texttt{Y},\texttt{X}), \texttt{sameTopic}(\texttt{X},\texttt{Y});$$

Der BA erstellt in seiner ersten Phase die drei folgenden Buckets:

- $Bucket_1$ ($\texttt{cites}_1$): $(\texttt{v1}, \texttt{X} \to \texttt{A}_1, \texttt{Y} \to \texttt{B}_1), (\texttt{v3}, \texttt{X} \to \texttt{F}_1, \texttt{Y} \to \texttt{G}_1)$
- $Bucket_2$ ($\texttt{cites}_2$): $(\texttt{v1}, \texttt{Y} \to \texttt{B}_2, \texttt{X} \to \texttt{A}_2), (\texttt{v3}, \texttt{Y} \to \texttt{G}_2, \texttt{X} \to \texttt{H}_2)$
- $Bucket_3$ (sameTopic): $(\texttt{v2}, \texttt{X} \to \texttt{C}_3, \texttt{Y} \to \texttt{D}_3), (\texttt{v3}, \texttt{X} \to \texttt{F}_3, \texttt{Y} \to \texttt{G}_3)$

In der zweiten Phase stellt der BA $2^3 = 8$ Kombinationen auf und überprüft diese. Die erste Kombination ist v1, v1, v2: Der BA beginnt mit dem ersten Vergleich (v1, $\texttt{X} \to \texttt{A}_1$, $\texttt{Y} \to \texttt{B}_1$) und $(\texttt{v1}, \texttt{Y} \to \texttt{B}_2, \texttt{X} \to \texttt{A}_2)$. Diese beiden Sichten sind nicht kompatibel, B in v1 nicht exportiert wird und $\texttt{B}_1 = \texttt{B}_2$ somit nicht sichergestellt werden kann. An dieser Stelle muss die Kombination nicht weiter verfolgt werden und es kann mit der nächsten fortgefahren werden.

Doch auch bei den weiteren Kombinationen wird der BA daran scheitern, dass die Variablen B in v1 und G in v3 nicht exportiert werden und somit als Joinvariablen nicht in Frage kommen. In diesem Beispiel findet er also keinen Anfrageplan und das, obwohl die dritte Sicht offensichtlich identisch mit der Anfrage ist. Mit dem Aufruf

$$\texttt{q}'(\texttt{X}) : -\texttt{v3}(\texttt{X},\texttt{X})$$

könnte die Sicht die Anfrage alleine beantworten.

3.2.4 Bewertung

Der BA ist korrekt: Jeder gefundene Plan ist semantisch richtig, obwohl an keiner Stelle explizit überprüft wird, ob ein Containment Mapping vorliegt. Jede der vier Voraussetzungen für ein Containment Mapping (vgl. Definition 2) wird im Algorithmus implizit berücksichtigt. Mit anderen Worten: Pläne, die an irgendeiner Stelle der Definition eines Containment Mappings widersprechen, werden verworfen.

Der BA ist aber nicht vollständig: Es gibt Anfragepläne, die er nicht findet. Das Problem ist, dass jeder vom BA erzeugte Plan genauso viele Sichten enthält wie die Anfrage Literale. Da es aber kürzere Pläne geben kann, die korrekt, aber nicht in einem längeren Plan enthalten sind, findet er nicht in jedem Fall alle Anfragepläne. Dies tritt insbesondere bei einem Join über eine nicht-exportierte Variable auf [Les00], wie im Beispiel der Literaturdatenbank gezeigt wurde.

Im Vergleich mit dem GTA ist der BA weitaus effizienter. Dieser hätte keinerlei Vorauswahl der Sichten getroffen und das Kreuzprodukt über alle Sichten aufgestellt.

3.3 Varianten des Bucket-Algorithmus

Wie im vorherigen Abschnitt gezeigt, ist der Bucket-Algorithmus in seiner vorgestellten Form nicht optimal. Zwar sind die gefundenen Anfragepläne immer korrekt; es gibt aber Pläne die ebenfalls korrekt wären und vom BA nicht gefunden werden. Dies liegt daran, dass die Länge der vom BA gefundenen Anfragepläne immer gleich der Länge der Anfrage ist, es aber kürzere Pläne geben kann, die nicht in einem längeren Plan enthalten sind. Ein vollständiger Algorithmus muss diese Spezialfälle also mit einbeziehen.

In diesem Abschnitt werden die zwei Varianten des BA, der Improved-Bucket-Algorithmus (IBA) und der Shared-Variable-Bucket-Algorithmus (SVB) kurz vorgestellt.

3.3.1 Improved-Bucket-Algorithmus

Der Vorgänger des IBA wurde im Jahr 1998 veröffentlicht [Les98]. Der IBA selbst wurde im Jahr 2000 dokumentiert [Les00]. Im Kern arbeitet er wie der BA mit zwei Phasen, erweitert den Algorithmus aber um die Eigenschaft der Vollständigkeit durch die Tatsache, dass auch kürzere Pläne, als die Anfrage Literale hat, möglich sind. Weiterhin optimiert er die Kombinationsphase durch Sortierung.

Die erste Phase des IBA läuft analog zur ersten Phase des BA. Am Anfang der zweiten Phase (Kombination) werden die Buckets aber einmalig nach ihrer Größe sortiert. Eine solche Sortierung führt eine Reduzierung der Vergleiche herbei. Sind beispielsweise drei Buckets gegeben mit $|B_1| = 1$, $|B_2| = 3$ und $|B_3| = 2$ so ergibt sich ohne Sortierung eine Kombinationsstruktur wie in Abbildung 3.1 (a) gezeigt. Sortiert man die Buckets nach ihrer Größe, ergibt sich der Kombinationsbaum (b) (selbe Abbildung).

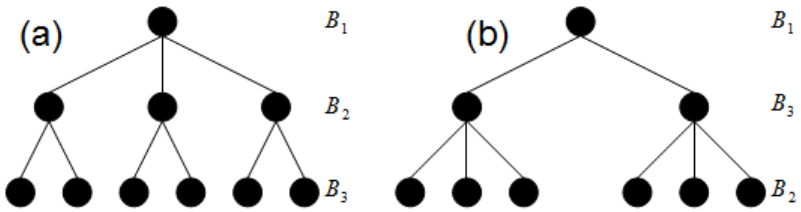


Abbildung 3.1: Ein Baum ohne Sortierung (a) und ein Baum mit Sortierung (b)

Der IBA erweitert einen bestehenden Teilplan immer um ein Element aus dem nächsten Bucket. Dabei werden ähnlich wie beim BA Vergleiche fällig, ob ein hinzukommendes Element zu einem bestehenden Plan kompatibel ist. Jede Kante im Kombinationsbaum entspricht einer solchen Operation. Im Fall von (a) müsste der Algorithmus 9 Vergleiche anstellen. Im sortierten Fall (b) sind es nur 8.

Nach der Sortierung werden die Pläne aufgezählt. Dabei muss ein Plan nicht zwingend so lang werden wie die Anzahl der Literale in der Anfrage (beim BA immer der Fall). Ist bereits bei einem Teilplan ein vollständiges Mapping zur Anfrage gegeben,

kann der Teilplan schon als Anfrageplan verwendet werden. Dadurch findet der IBA auch Anfragepläne, die eventuell in einem längeren Plan nicht enthalten wären. Er ist somit vollständig.

3.3.2 Shared-Variable-Bucket-Algorithmus

Der Shared-Variable-Bucket-Algorithmus (SVB) [Mit99] arbeitet sehr ähnlich zum BA mit seinen zwei Phasen. Der signifikante Unterschied ist, dass es strengere Regeln gibt, ob eine Sicht in einen Bucket kommt oder nicht.

Dabei unterscheidet der Algorithmus zwischen Single-Subgoal-Buckets, in die Sichten kommen, wenn sie genau ein Literal der Anfrage abdecken, und Shared-Variable-Buckets, in denen Sichten aufgesammelt werden, die mehrere Literale abdecken. Der SVB versucht so die Buckets minimal, also für möglichst wenige Literale zu erstellen. Das stellt sicher, dass alle möglichen Kombinationen von Sichten auch tatsächlich gefunden werden.

In der zweiten Phase des Algorithmus werden die erstellten Buckets ähnlich wie beim BA kombiniert. Durch die Tatsache, dass es Buckets geben kann, die mehrere Literale abdecken, schließt der SVB die Vollständigkeitslücke des BA.

Der SVB ist also eine gute Alternative zum BA, wird an dieser Stelle aber nicht formal beschrieben. Der nun folgende MiniCon-Algorithmus berücksichtigt ebenso wie der SVB die Eigenschaft, dass Sichten mehrere Literale einer Anfrage abdecken können, reizt die Möglichkeiten aber noch weiter aus und verwirft Sichten, die niemals zum Ergebnis beitragen können, bereits in der ersten Phase.

3.4 MiniCon-Algorithmus

Die meisten Probleme hat der BA mit den Joins in der Benutzeranfrage. Denn diese funktionieren nur, wenn die notwendigen Variablen von den Sichten exportiert werden. Diese Eigenschaft nutzt der MiniCon-Algorithmus (MCA) [PH01]. Er hat eine ähnliche Grundidee wie der BA. Jedoch probiert er stets kleinere Buckets zu erzeugen und kann manche Sichten früh ausschließen. Dadurch zählt er in der Regel weniger potentielle Pläne auf, als die bisher vorgestellten Algorithmen.

Zuerst wird in diesem Abschnitt der MCA formal beschrieben und an Hand von Pseudocodes angegeben. Danach folgen die gleichen zwei Beispiele wie im Abschnitt über den BA, die die unterschiedliche Vorgehensweise deutlich machen. Im letzten Abschnitt wird der MCA bewertet.

3.4.1 Formale Beschreibung

Der MCA ist ebenso wie der BA in zwei Phasen aufgeteilt. In der ersten Phase werden pro Sicht Buckets erstellt, nicht wie beim BA ein Bucket für jedes Literal der Anfrage. Ein solcher Bucket wird MiniCon Description (MCD) genannt und ist komplexer als ein Bucket beim BA. Zusätzlich zum Mapping wird noch ein Homomorphismus der

Kopfvariablen und die Teilmenge der überdeckten Literale berechnet. Der Homomorphismus wird benötigt, um Variablen angleichen zu können. Die Teilmenge der überdeckten Literale dient zur einfacheren Kombination der MCDs in der zweiten Phase des Algorithmus. Das partielle Mapping, das beim BA mit h bezeichnet wurde, bekommt beim MCA das Zeichen φ, weil der Homomorphismus h genannt wird. MiniCon Descriptions sind wie folgt definiert:

Definition 4. *(MiniCon Descriptions) Ein MCD C für eine Anfrage Q und eine Sicht V ist ein Tupel der Form $(h, V(\bar{Y}), \varphi, G)$ mit:*

- *h ein Homomorphismus, also ein Mapping von $exp(V) \rightarrow exp(V)$,*
- *$V(\bar{Y})$ ist die Sicht V nach Umbenennung der Kopfvariablen gemäß h, mit $\bar{Y} = h(exp(V))$,*
- *φ ist ein partielles Mapping von $Var(Q)$ nach $Var(V)$,*
- *G ist eine Teilmenge von den Literalen von Q, welche von mindestens einem Literal in $h(V)$ mit dem Mapping φ benutzt werden.*

Ein Homomorphismus von Kopfvariablen entsteht bei der Unifikation von Anfrage und Sicht sowie aus Kombination der partiellen Mappings für die Literale der Anfrage. Um dieses Prinzip zu verdeutlichen sei eine Anfrage

$$q(x) : -cites(X, Y), sameTopic(Y, X)$$

und eine Sicht

$$v(A, C) : -cites(A, B), sameTopic(B, C)$$

gegeben. Die möglichen partiellen Mappings für die Literale der Anfrage sind $X \rightarrow A$, $Y \rightarrow B$ und $X \rightarrow C$. Offensichtlich wird die Variable X auf zwei unterschiedliche Variablen gemappt und muss daher angeglichen werden. Es muss also $A = C$ gelten. Daraus lässt sich ableiten, dass es zwei Homomorphismen der Kopfvariablen der Sicht v gibt: $\{A \rightarrow A, C \rightarrow A\}$ und $\{A \rightarrow C, C \rightarrow C\}$. Im MCA ist es nicht nötig, alle möglichen Kopfhomomorphismen zu berechnen, es wird sich auf einen beschränkt.

$V(\bar{Y})$ ergibt sich aus dem Kopf der Sicht und dem Homomorphismus h aus einer MCD. Für die obige Sicht $v(A, C)$ mit $h = \{A \rightarrow A, C \rightarrow A\}$ ist $V(\bar{Y}) = V(A, A)$. Zu beachten ist, dass $V(\bar{Y})$ nicht extra berechnet wird, sondern sich aus den anderen Elementen der MCD ergibt und somit nicht im unten angegebenen Algorithmus vorkommt.

φ ist ein partielles Mapping von der Menge der Variablen, die in den von der MCD überdeckten Literalen von Q vorkommen zu einer Menge von Variablen aus V.

Die Menge G enthält die Literale der Anfrage, die von der Sicht überdeckt werden. G muss nicht zwingend alle Literale enthalten.

Neben Definition 4 ist noch eine weitere Definition nötig, um den Algorithmus formal anzugeben. Da eine MCD für eine Sicht V nur unter sehr viel eingeschränkteren

Bedingungen als beim Bucket-Algorithmus erstellt wird, wird in Definition 5 eine Eigenschaft eingeführt, die jede MCD erfüllen muss. Im unten angegebenen Algorithmus wird diese Eigenschaft dann verwendet.

Definition 5. *(MCD-Eigenschaft) Sei C eine MCD für eine Anfrage Q und eine Sicht V. Dann darf C in einem Anfrageplan von Q nur benutzt werden, wenn gilt:*

1. *Für jede Variable x von Q, die in φ vorkommt, ist $\varphi(x)$ eine Kopfvariable in $h(V)$.*
2. *Falls $\varphi(x)$ keine Kopfvariable in $h(V)$ ist, muss für jedes Literal l von Q, welches die Variable x enthält, gelten: (1) Alle Variablen in l kommen in φ vor; und (2) $\varphi(l) \in h(V)$.*
3. *Für jede Konstante a in Q muss $\varphi(a)$ eine exportierte Variable sein oder $\varphi(a)$ muss die Konstante a selbst sein.*

Der erste Punkt der Eigenschaft kommt so auch im BA vor: Variablen, die im Mapping vorkommen, müssen auch exportiert werden, damit sie überhaupt gemappt werden können.

Der MCA erweitert die Bedingung des BA noch um einen zweiten Punkt: Wenn eine Variable x ein Teil eines Joins ist, der von der Sicht selbst nicht durchgeführt wird, dann muss x eine Kopfvariable in der Sicht sein, damit der Join ausgeführt werden kann.

Der dritte Punkt deckt den Fall ab, dass die Anfrage Konstanten enthält. Ist dies der Fall, muss die Konstante auf eine Variable, die von der Sicht exportiert wird oder auf eine gleiche Konstante gemappt werden. Dies ist nötig, damit der Wert der Konstante überprüft werden kann.

Die erste Phase des MCA, das Erzeugen der MCDs mittels einer Anfrage und einer Menge von Sichten, wird im Algorithmus 3 gezeigt.

Die im Vergleich zum BA kompliziertere Methode des MCA die MCDs zu erstellen, zahlt sich in der zweiten Phase des Algorithmus aus. Diese kombiniert die MCDs, die die Literale der Anfrage paarweise disjunkt überdecken. Dabei wendet der MCA eine Kombinationseigenschaft an, die in Definition 6 angegeben ist.

Definition 6. *(Kombinationseigenschaft des MCA) Gegeben sei eine Anfrage Q, eine Menge von Sichten $\mathcal{V}$ sowie eine Menge von MCDs $\mathcal{C}$ für Q. Die einzigen Kombinationen von MCDs, die zu vollständigen Anfrageplänen führen, sind von der Form $C_1, ..., C_n$, für die gilt (mit G_{C_i} die Teilmenge der überdeckten Literale G aus der MCD C_i):*

1. *$G_{C_1} \cup G_{C_2} \cup ... \cup G_{C_n} =$ Literale(Q), und*
2. *$\forall i, j$ mit $i \neq j$ gilt: $G_{C_i} \cap G_{C_j} = \emptyset$.*
3. *Für alle beteiligten MCDs sind die Bedingungen der Sichten aus $\mathcal{V}$ kompatibel.*

Ein Anfrageplan muss also aus nicht-überlappenden MCDs bestehen, was zu einer zusätzlichen Einsparung beim Aufzählen von Kandidaten führt. Dafür werden die Teilmengen von Literalen G aus den MCDs benutzt. Wenn eine MCD alle Literale von Q

Algorithmus 3 Phase 1 des MiniCon-Algorithmus: MCDs erstellen

input: Eine Menge von konjunktiven Quellendefinitionen $\mathcal{V}$
input: Eine konjunktive Anfrage Q
output: Eine Menge von MiniCon Descriptions $\mathcal{C}$

```
1:  function FORMMCDS(V, Q)
2:      C ← ∅
3:      for Literale l ∈ Q do
4:          for all V ∈ V do
5:              if cond(V) kompatibel zu cond(Q) then
6:                  for Literale k ∈ V do
7:                      Erstelle einen Homomophismus h auf V,
8:                      so dass ein Mapping φ existiert bei dem gilt, dass φ(l) = h(k)
9:                      if h und φ existieren und die MCD Eigenschaft (s.o.) erfüllt ist then
10:                         if MCD C für Sicht V existiert und kann erweitert werden then
11:                             Erweitere C um h und φ
12:                         else
13:                             Erstelle eine neue MCD C = (h_C, V_C, φ_C, G_C) mit:
14:                             h_C ← h
15:                             φ_C ← φ
16:                             G_C ist die minimale Menge von Literalen von Q, die von C
17:                                 überdeckt werden
18:                             C ← C ∪ C
19:                         end if
20:                     end if
21:                 end for
22:             end if
23:         end for
24:     end for
25:     return C
26: end function
```

enthält, so ist diese MCD direkt ein Anfrageplan. Enthält sie nur eine Teilmenge der Literale, so müssen weitere MCDs gefunden werden, die die fehlenden Literale beitragen können.

Der dritte Punkt der Definition besagt, dass nur Sichten kombiniert werden, deren Bedingungen kompatibel sind. Das stellt sicher, dass kein Anfrageplan erstellt wird, der in jedem Fall ein leeres Ergebnis berechnen würde [2].

Der letzte Schritt ist es, jeden Anfrageplan auf Sichten zu überprüfen, die mehrmals im Anfrageplan vorkommen. Redundante Sichten können entfernt werden, ohne etwas

[2] Diese Eigenschaft ist im originalen MCA nicht vorgesehen. Sie ist aber sinnvoll, um überflüssige Kombinationen direkt in der Phase der Anfrageplanung auszuschließen. Dies reduziert den Aufwand in den späteren Phasen, insbesondere in der Anfrageoptimierung und -ausführung.

am Ergebnis zu ändern (siehe Abschnitt 3.1.2 über minimale Anfragepläne).

Die zweite Phase des MCA ist in Algorithmus 4 gezeigt.

Algorithmus 4 Phase 2 des MiniCon-Algorithmus: MCDs kombinieren

input: Eine Menge von MCDs $\mathcal{C}$, erzeugt von der ersten Phase des Algorithmus
output: Eine Menge von umgeschriebenen Anfragen $\mathcal{R}$

```
1:  function COMBINEMCDS(C)
2:      R ← ∅
3:      for all C_1, ..., C_n ∈ C so dass die Kombinationseigenschaft (s.o.) erfüllt ist do
4:          Erstelle ein Mapping Ψ_i auf Ȳ_i
5:          for all y ∈ h_C do
6:              if x ∈ Q existiert mit φ_i(x) = y then
7:                  Ψ_i(y) ← x
8:              else
9:                  Ψ_i(y) ist neue Kopie von y
10:             end if
11:         end for
12:         Schreibe die Anfrage Q mit Hilfe des Mappings Ψ zu Q' um
13:         Entferne dabei doppelte Sichten
14:         R ← R ∪ Q'
15:     end for
16:     return R
17: end function
```

3.4.2 Beispiel Vorlesungsdatenbank

Gegeben seien die gleichen globalen Relationen `lehrt`, `eingeschrieben` und `kurs`, wie in Abschnitt 3.2.2:

$\texttt{lehrt}(\texttt{P}, \texttt{K}, \texttt{S})$	Ein Professor P lehrt einen Kurs K in dem Semester S
$\texttt{eingeschrieben}(\texttt{H}, \texttt{K}, \texttt{S})$	Ein Hochschüler H ist im Semester S im Kurs K eingeschrieben
$\texttt{kurs}(\texttt{K}, \texttt{T})$	Der Kurs mit der Id K hat den Titel T

und die gleichen vier Sichten:

$$\begin{array}{lll}
\texttt{v1}(\texttt{H}, \texttt{T}, \texttt{S}, \texttt{K}) & \texttt{:-} & \texttt{eingeschrieben}(\texttt{H}, \texttt{K}, \texttt{S}), \texttt{kurs}(\texttt{K}, \texttt{T}), \texttt{K} \geq 500, \texttt{S} \geq \texttt{WS98}; \\
\texttt{v2}(\texttt{H}, \texttt{P}, \texttt{S}, \texttt{K}) & \texttt{:-} & \texttt{eingeschrieben}(\texttt{H}, \texttt{K}, \texttt{S}), \texttt{lehrt}(\texttt{P}, \texttt{K}, \texttt{S}), \texttt{K} \leq 400; \\
\texttt{v3}(\texttt{H}, \texttt{K}) & \texttt{:-} & \texttt{eingeschrieben}(\texttt{H}, \texttt{K}, \texttt{S}), \texttt{S} \leq \texttt{WS94}; \\
\texttt{v4}(\texttt{P}, \texttt{K}, \texttt{T}, \texttt{S}) & \texttt{:-} & \texttt{lehrt}(\texttt{P}, \texttt{K}, \texttt{S}), \texttt{kurs}(\texttt{K}, \texttt{T}), \texttt{eingeschrieben}(\texttt{H}, \texttt{K}, \texttt{S}), \texttt{S} \leq \texttt{WS97};
\end{array}$$

Die Anfrage, die an die integrierte Datenbank gestellt werden soll, lautet wieder: „Welche Hochschüler H waren im Kurs mit der Id K (≥ 300) beim Lehrenden P im oder nach dem WS95 eingeschrieben?“ und wird wie folgt in Datalog formuliert:

$$\texttt{q}(\texttt{H},\texttt{K},\texttt{P}) :- \texttt{lehrt}(\texttt{P},\texttt{K},\texttt{S}), \texttt{eingeschrieben}(\texttt{H},\texttt{K},\texttt{S}), \texttt{kurs}(\texttt{K},\texttt{T}), \texttt{S} \geq \texttt{WS95}, \texttt{K} \geq 300;$$

Der MCA versucht jetzt für jede Sicht eine MCD zu erstellen. Genau wie beim BA fällt auch hier die Sicht v3 raus, weil die Bedingung $\texttt{S} \leq \texttt{WS94}$ nicht mit der Bedingung in der Anfrage $\texttt{S} \geq \texttt{WS95}$ kompatibel ist. Für die drei anderen Sichten werden jeweils zwei MCDs erstellt, diese sind in Tabelle 3.1 dargestellt.

C	V($\bar{Y}$)	h	φ	G
1	v1(H,_,S,K)	φ	$\texttt{H} \to \texttt{H}, \texttt{K} \to \texttt{K}, \texttt{S} \to \texttt{S}$	eingeschrieben
2	v1(_,T,_,K)	φ	$\texttt{K} \to \texttt{K}, \texttt{T} \to \texttt{T}$	kurs
3	v2(_,P,S,K)	φ	$\texttt{K} \to \texttt{K}, \texttt{P} \to \texttt{P}, \texttt{S} \to \texttt{S}$	lehrt
4	v2(H,_,S,K)	φ	$\texttt{H} \to \texttt{H}, \texttt{K} \to \texttt{K}, \texttt{S} \to \texttt{S}$	eingeschrieben
5	v4(P,K,_,S)	φ	$\texttt{K} \to \texttt{K}, \texttt{P} \to \texttt{P}, \texttt{S} \to \texttt{S}$	lehrt
6	v4(_,K,T,_)	φ	$\texttt{K} \to \texttt{K}, \texttt{T} \to \texttt{T}$	kurs

Tabelle 3.1: MCDs im Beispiel der Vorlesungsdatenbank

An diesem Beispiel sieht man, dass der Homomorphismus nicht immer benötigt wird. Da jede Sicht fast alle Variablen auch exportiert und keine Variable auf zwei unterschiedliche Variablen gemappt wird, müssen keine Angleichungen der Kopfvariablen vorgenommen werden, das Mapping entspricht also in diesem Beispiel genau dem Homomorphismus. Dies ist nur der Fall, weil die Variablen die gleichen Namen haben. Bei einer Umbenennung der Variablen, beispielsweise in der Anfrage, würden die beiden Mappings unterschiedliche Variablennamen enthalten.

Nach dem Erstellen der MCDs werden diese in der zweiten Phase kombiniert. Dabei muss die Kombinationseigenschaft (Definition 6) erfüllt sein. In diesem Beispiel müssen also immer genau drei MCDs ausgewählt werden. Die Kombinationen, die entstehen, sind dabei die gleichen wie beim BA:

1. v2, v2, v4
   Hier gilt: $G_{C_3} \cup G_{C_4} \cup G_{C_6} = Literale(q)$
2. v4, v2, v4
   Hier gilt: $G_{C_5} \cup G_{C_4} \cup G_{C_6} = Literale(q)$

Bei allen anderen Kombinationen ist mindestens ein Punkt der Kombinationseigenschaft verletzt. Sie werden daher, genau wie beim BA, verworfen. Alle Kombinationen, die die Sichten v1 und v2 enthalten, werden ein leeres Ergebnis liefern, weil $\texttt{K} \geq 500$ und $\texttt{K} \leq 400$ keine gemeinsamen K ermöglichen. Bei Kombinationen, die v1 und v4 enthalten, wird ebenfalls eine leere Ergebnismenge berechnet, da $\texttt{S} \geq \texttt{WS98}$ von der Sicht v1 keine Überschneidung mit $\texttt{S} \leq \texttt{WS97}$ von der Sicht v4 hat.

Die zwei gefundenen Kombinationen ergeben die folgenden Anfragepläne:

$$\texttt{q}'(\texttt{H},\texttt{K},\texttt{P}) :- \texttt{v2}(\_,\texttt{P},\texttt{S},\texttt{K}), \texttt{v2}(\texttt{H},\_,\texttt{S},\texttt{K}), \texttt{v4}(\_,\texttt{K},\texttt{T},\_), \texttt{S} \geq \texttt{WS95}, \texttt{K} \geq 300;$$

$$q''(H, K, P) :- v4(P, K, \_, S), v2(H, \_, S, K), v4(\_, K, T, \_), S \geq WS95, K \geq 300;$$

Im letzten Schritt des MCA werden Sichten, die in einem Anfrageplan mehrmals auftreten, entfernt, so dass jede Sicht maximal einmal in einem Anfrageplan vorkommt. Die gekürzten Anfragepläne lauten wie folgt:

$$q'(H, K, P) :- v2(H, P, S, K), v4(\_, K, T, \_), S \geq WS95, K \geq 300;$$

$$q''(H, K, P) :- v2(H, \_, S, K), v4(P, K, T, S), S \geq WS95, K \geq 300;$$

Die erzeugten Anfragepläne entsprechen den gleichen Anfrageplänen, die auch der BA gefunden hat. Dieser hat allerdings keine Reduzierung von mehrfach auftretenden Sichten vorgenommen. In diesem Beispiel arbeiten die beiden Algorithmen also nahezu gleich und würden in einer tatsächlichen Ausführung keine spürbaren Geschwindigkeitsunterschiede aufweisen. Das nun folgende Beispiel stellt die Vorteile des MCA gegenüber dem BA heraus.

3.4.3 Beispiel Literaturdatenbank

Mit dem Beispiel einer Literaturdatenbank hatte der BA einige Probleme: Er hatte relativ viele Kombinationen zu vergleichen, hat schlussendlich aber keinen Anfrageplan gefunden. Das gleiche Beispiel wird nun mit dem MCA gezeigt, der sehr viel besser damit umgehen kann.

Gegeben seien wieder die globalen Relationen `cites` und `sameTopic`, die drei Sichten `v1`, `v2` und `v3` und die Anfrage `q`:

$$\begin{aligned}
&cites(X, Y)\\
&sameTopic(A, B)\\
&v1(A) : -cites(A, B), cites(B, A);\\
&v2(C, D) : -sameTopic(C, D);\\
&v3(F, H) : -cites(F, G), cites(G, H), sameTopic(F, G);\\
&q(X) : -cites(X, Y), cites(Y, X), sameTopic(X, Y);
\end{aligned}$$

Der MCA erstellt in seiner ersten Phase die MCDs, die in Tabelle 3.2 gezeigt sind. Sofort sieht man einen großen Unterschied zum BA: Die Sicht `v1`, bekommt keinen MCD, weil sie `B` nicht exportiert und `v1` mit `sameTopic` gejoint sein müsste. `v1` enthält aber kein Literal `sameTopic`. Somit wird `v1` vom MCA schon in der ersten Phase als unbrauchbar erkannt und komplett ausgeschlossen. Dadurch spart sich der Algorithmus später viele überflüßige Vergleiche.

Der Homomorphismus h von `v2` kommt wie folgt zu stande: Da es nur ein mögliches Mapping φ mit $X \to C, Y \to D$ gibt, braucht keine Unifikation gemacht zu werden und damit besteht h aus den Variablen die auf sich selbst abbilden (es ist keine Angleichung erforderlich). Bei `v3` gibt es zwei mögliche Mappings (da man die Reihenfolge der Literale vertauschen kann):

$V(\bar{Y})$	h	φ	G
$\mathtt{v2}(\mathtt{C},\mathtt{D})$	$\mathtt{C}\to\mathtt{C},\mathtt{D}\to\mathtt{D}$	$\mathtt{X}\to\mathtt{C},\mathtt{Y}\to\mathtt{D}$	$\mathtt{sameTopic}$
$\mathtt{v3}(\mathtt{F},\mathtt{F})$	$\mathtt{F}\to\mathtt{F},\mathtt{H}\to\mathtt{F}$	$\mathtt{X}\to\mathtt{F},\mathtt{Y}\to\mathtt{F}$	$\mathtt{cites}_1,\mathtt{cites}_2,\mathtt{sameTopic}$

Tabelle 3.2: MCDs im Beispiel der Literaturdatenbank

- $\mathtt{X}\to\mathtt{F},\mathtt{Y}\to\mathtt{G},\mathtt{X}\to\mathtt{H}$
  Bei diesem Mapping wird X auf zwei verschiedene Variablen gemappt. Diese müssen daher angeglichen werden. Hier muss also $\mathtt{F}=\mathtt{H}$ gelten.
- $\mathtt{X}\to\mathtt{G},\mathtt{X}\to\mathtt{F},\mathtt{Y}\to\mathtt{G},\mathtt{Y}\to\mathtt{F},\mathtt{Y}\to\mathtt{H}$
  Bei dieser zweiten Möglichkeit werden sowohl X, als auch Y auf verschiedene Variablen gemappt. Für X ergibt sich die Angleichung $\mathtt{G}=\mathtt{F}$ und für Y muss $\mathtt{G}=\mathtt{F}=\mathtt{H}$ festgelegt werden. Da G keine Kopfvariable ist, ist sie für den Homomorphismus uninteressant und es muss nur $\mathtt{F}=\mathtt{H}$ betrachtet werden.

Bei beiden Mappings kommt also heraus, dass die Kopfvariablen F und H angeglichen werden müssen. Genau dafür ist jetzt der Homorphismus zuständig, der mittels $\mathtt{H}\to\mathtt{F}$ aus H die Variable F macht.

Die zweite Phase des MCA kombiniert die MCDs und stellt beim Überprüfen der Kombinationen fest, dass v2 nur zusammen mit v3 benutzt werden kann. Da v3 aber alle Literale und v2 nur ein Literal abdeckt, wird v2 verworfen und der einzige erzeugte Anfrageplan lautet:

$$\mathtt{q}'(\mathtt{X}) : -\mathtt{v3}(\mathtt{X},\mathtt{X})$$

Bei diesem Beispiel findet der MCA also einen Anfrageplan, wohingegen der BA gescheitert ist und keinen Plan gefunden hat.

3.4.4 Bewertung

Der MiniCon-Algorithmus gilt als derzeit schnellster Algorithmus für die Local-as-View Anfrageplanung. Durch seine strengen Regeln beim Anlegen der MiniCon Descriptions hat er in der teuren Phase, dem Kombinieren, in vielen Fällen weniger Vergleiche zu tätigen und dadurch einen Geschwindigkeitsvorteil.

Der MCA ist sowohl korrekt als auch vollständig [PH01]. Dadurch ist er ein guter Kandidat, trotz komplizierterer Implementierung, um in einer integrierten Datenbank zur Anfrageplanung verwendet zu werden.

3.5 Inverse-Rules-Algorithmus

Einen vollkommen anderen Ansatz verfolgt der Inverse-Rules-Algorithmus (IRA) [GKD97]. Er dreht die Sichten zu Regeln um. Für jedes Literal einer Sicht wird eine

eigene Regel erzeugt. Ein Problem gibt es dabei mit nicht-exportierten Variablen. Wenn diese nicht in Joins verwendet werden, sind sie unerheblich. Wenn sie in Joins verwendet werden, soll diese Information erhalten bleiben. Man setzt dafür eine Skolemfunktion ein, die als reiner Platzhalter fungiert, also nie ausgerechnet wird. Nach Berechnung der Formeln dürfen gleiche Skolemfunktionen gejoint werden.

Eventuelle Nebenbedingungen der Sichten, die > oder < enthalten, können beim Erstellen der Regeln nicht übernommen werden und gehen verloren. Die Erzeugung der Regeln ist in Algorithmus 5 dargestellt.

Algorithmus 5 Inverse-Rules-Algorithmus: Regeln erstellen

input: Eine Menge von konjunktiven Quellendefinitionen $\mathcal{V}$.
output: Eine Menge von Datalog Regeln $\mathcal{R}$

```
1:  function CREATERULES(V)
2:      R ← ∅
3:      for all V ∈ V do
4:          if V enthält nicht-exportierte Variablen then
5:              Erstelle neue Skolemfunktion f mit exp(V) im Funktionskopf
6:          end if
7:          for all Literal l ∈ V do
8:              Erzeuge neue Regel R für Literal l mit V
9:              Ersetze ggf. in R existierende nicht-exportierten Variablen durch f
10:             R ← R ∪ R
11:         end for
12:     end for
13:     return R
14: end function
```

Gegeben sei wieder das Beispiel einer Literaturdatenbank mit den drei Sichten `v1`, `v2` und `v3`. Diese Sichten werden vom IRA zunächst in die folgenden sechs Regeln umgeformt:

- R1: `cites(A, f1(A)) :- v1(A);`
- R2: `cites(f1(A), A) :- v1(A);`
- R3: `sameTopic(C, D) :- v2(C, D);`
- R4: `cites(F, f2(F,H)) :- v3(F, H);`
- R5: `cites(f2(F,H), H) :- v3(F, H);`
- R6: `sameTopic(F, f2(F,H)) :- v3(F, H);`

Diese Regeln können nun als Datalog-Regeln aufgefasst werden. Durch das Ausführen einer Regel wird sie berechnet. Die Anfrage q muss nicht mehr umgeschrieben werden, wie es bei den oben vorgestellten Algorithmen der Fall war. Sie kann einfach auf den Regeln ausgeführt werden.

Das Beispiel fortgesetzt, werden folgende $4 \cdot 4 \cdot 2 = 32$ Ableitungen erzeugt und durchgegangen:

- R1, R1, R3
- R1, R1, R6
- R1, R2, R3
- R1, R2, R6
- ...
- R5, R5, R6

Allerdings fallen viele Kombinationen frühzeitig raus, wenn keine Unifikation möglich ist.

Der große Vorteil dieses Algorithmus ist seine Einfachheit. Die Regeln sind leicht zu erstellen, das komplizierte Umschreiben der Anfrage entfällt vollständig. Das Problem ist die Durchführung: Datalog-Systeme sind eher theoretischer oder didaktischer[3] Natur und spielen in realen Systemen quasi keine Rolle. Die anderen Algorithmen arbeiten zwar ebenfalls mit Datalog als Anfragesprache, die erzeugten Anfragepläne werden jedoch in einem weiteren Schritt in SQL übersetzt. Diese Übersetzung kann beim IRA nicht vorgenommen werden, da er die Anfrage direkt mittels der Regeln ausführt.

3.6 Vergleich der Algorithmen

Der BA war der erste Algorithmus zur Anfrageplanung. Im Vergleich zum MCA hat er eine höhere Komplexität für den Average Case [PH01]. In der Realität sind die meisten Buckets aber eher schwach besetzt. Dadurch sind die Kombinationsmöglichkeiten gering und es kommt in vielen Fällen sogar zu linearen Anfragepläne [Les00].

Die Eigenschaft, dass der BA unvollständig ist, wird durch die Varianten IBA und SVB behoben, die auch kürzere Anfragepläne finden, als die globale Anfrage Literale enthält.

Der MCA gilt als der derzeit schnellste Algorithmus zur Anfrageplanung. Es können signifikante Geschwindigkeitsvorteile für den Average Case gezeigt werden [PH01]. Problematisch ist nur die Bestimmung eines Average Case. Wenn alle Quellen alle Attribute exportieren, arbeitet der MCA äquivalent zum BA. Werden nur wenige Attribute von den Quellen exportiert, hat der MCA Vorteile in der (komplexen) Kombinationsphase und der BA hat unter Umständen Probleme alle Anfragepläne zu finden.

Der IRA setzt sich von den anderen Algorithmen dadurch ab, dass er auch mit rekursiven Sichten funktioniert (die in dieser Arbeit aber ausgeschlossen sind). Er kann allerdings nicht mit Bedingungen umgehen, die $>$ oder $<$ enthalten. Diese Informationen gehen beim Erzeugen der Regeln verloren. Des Weiteren koppelt er die Planung bereits mit der Ausführung: Es werden Anfragen ausgeführt, obwohl die Tupel unter Umständen nicht gebraucht werden. Da Regeln in verschiedenen Ästen des Ableitungsbaums mehrmals ausgeführt werden, sollte zudem ein Caching integriert werden.

Einen entscheidenden Schnelligkeitsvorteil für den worst-case hat allerdings keiner der Algorithmen, da das Problem von AQUV NP-vollständig ist. Die Zeitkomplexität ist bei allen vorgestellten Algorithmen gleich. Sie beträgt $\mathcal{O}(nmM)^n$, mit n Anzahl von

[3] Wie zum Beispiel das Datalog Educational System (Web: http://sourceforge.net/projects/des/).

Literalen der Anfrage, m die maximale Anzahl an Literalen, die in einer Sicht vorkommen, und M die Gesamtanzahl der Sichten. [PH01]

Trotz der exponentiellen Zeitkomplexität lassen sich die Algorithmen im praktischen Einsatz verwenden, da sowohl m, aber vor allem n meistens kleine Zahlen sind. Möglich ist auch eine explizite Beschränkung von n auf eine Konstante c: Zum Beispiel können Anfragen bei denen $n > c$ gilt, von einer integrierten Datenbank abgelehnt werden, um die Ausführungszeit der Anfragebearbeitung in Grenzen zu halten.

Kapitel 4

Anfrageschnittstelle für integrierte Datenbanken

Um das Local-as-View-Paradigma und die Anfrageplanungsalgorithmen auf Praxistauglichkeit zu untersuchen, wurde im Rahmen dieser Arbeit eine Anfrageschnittstelle entwickelt, die Anfragen an integrierte Datenbanken möglich macht.

In diesem Kapitel sind die Anforderungen, der Entwurf, die Implementierung und der Test der entwickelten Anfrageschnittstelle beschrieben. Die Anforderungen entspringen dabei zu großen Teilen dem Kapitel 2, welches die Grundlagen dieser Arbeit beschreibt.

4.1 Anforderungen

Der erste Abschnitt dieses Kapitels beschäftigt sich mit den zu Grunde liegenden Anforderungen an die Anfrageschnittstelle. Dabei wird unterschieden in funktionale Anforderungen, technische Anforderungen, Anforderungen an die Benutzerschnittstelle und Qualitätsanforderungen.

4.1.1 Funktionale Anforderungen

Die Anfrageschnittstelle soll Anfragen in der Datenbanksprache Datalog entgegennehmen und die Anfragebearbeitung nach dem Local-as-View-Paradigma durchführen. Dies bedeutet, dass die Anfrage mit einem Anfrageplanungsalgorithmus in einen oder mehrere Anfragepläne umgeschrieben werden muss. Es sollen der Bucket-Algorithmus und der MiniCon-Algorithmus implementiert werden (siehe Kapitel 3). Der gewünschte Algorithmus soll vor dem Absetzen einer neuen Anfrage gewählt werden können.

Nach LaV vorzugehen heißt im nächsten Schritt, die Anfragepläne von Datalog in SQL zu übersetzen und sie zu optimieren. Auf eine Optimierung wird in dieser Arbeit verzichtet, da von keinen wesentlichen Geschwindigkeitsvorteilen auszugehen ist, da alle Datenquellen im Rahmen dieser Arbeit immer in der selben Datenbank liegen.

Die optimierten Anfragepläne sollen dann ausgeführt werden. Wenn es der zeitliche Rahmen zulässt, soll zur Optimierung ein Caching implementiert werden, damit Quellen nicht doppelt angefragt werden, wenn es sich vermeiden lässt. Die gesamte Ausführung soll von der Anfrageschnittstelle überwacht werden: Kann eine Quelle mehrmals nicht erreicht werden (etwa wegen eines Serverausfalls), soll darauf reagiert werden, indem die Anfrageplanung unter Ausschluss der defekten Quelle erneut ausgeführt wird.

Nach der Ausführung folgt die Ergebnisintegration. Diese soll sich hier auf das Vereinigen der Ergebnistupel beschränken, wobei die Ergebnisse mit ihren jeweiligen Quellen gekennzeichnet werden sollen, damit für den Benutzer der Anfrageschnittstelle erkenntlich ist, aus welcher Quelle welche Ergebnisse stammen. Diese Einschränkung wird festgelegt, da es in virtuell integrierten Datenbanken in der Regel nicht möglich ist, im Rahmen der Anfragebearbeitung eine vollwertige Ergebnisbereinigung durchzuführen (vgl. Abschnitt 2.1.5). Die Ergebnisse sollen in Tabellenform ausgegeben werden.

Da integrierte Datenbanken aus verteilten Quellen bestehen, muss die Anfrageschnittstelle eine Möglichkeit zum Hinzufügen, Ändern und Löschen von Datenbanken bereitstellen. Einmal hinzugefügte Schemata der Datenbanken mit Verbindungsinformationen der Quellen (Server, Benutzername, Passwort) sollen persistent in Dateien oder einer internen Datenbank gespeichert werden.

Damit der komplexe Ablauf bei integrierten Datenbanken nach LaV verfolgt werden kann, soll es einen anzeigbaren Log geben, der Abläufe im Hintergrund aussagekräftig dokumentiert.

4.1.2 Technische Anforderungen

Die Anfrageschnittstelle soll in der objektorientierten Programmiersprache Java geschrieben sein. Java ist plattformunabhängig und damit nicht auf ein Betriebssystem festgelegt, sondern kann ohne Änderungen des Quellcodes auf allen gängigen Systemen eingesetzt werden, für die es Laufzeitumgebungen gibt.

Die Quellen, die zu einer Gesamtdatenbank integriert werden, sollen in der Datenbanksprache SQL angefragt werden. SQL ist die am weitesten verbreitete und weitgehend standardisierte Anfragesprache für relationale Datenbanken (vgl. Abschnitt 2.2.1).

4.1.3 Anforderungen an die Benutzerschnittstelle

Die Anfrageschnittstelle soll eine grafische Benutzerschnittstelle (GUI, engl. Graphical User Interface) haben. Wichtig ist, dass die Ein- und Ausgabe dem Benutzer vertraut sind, also ähnlich zu bekannten Datenbankmanagementsystemen aussieht. Dafür soll im oberen Drittel ein großes Eingabefeld für die Datenbankanfrage und darunter eine Tabelle mit dem Ergebnis sein. In Abbildung 4.1 ist ein Mockup der zu entwickelnden Anfrageschnittstelle gezeigt, welches im folgenden genau erklärt wird.

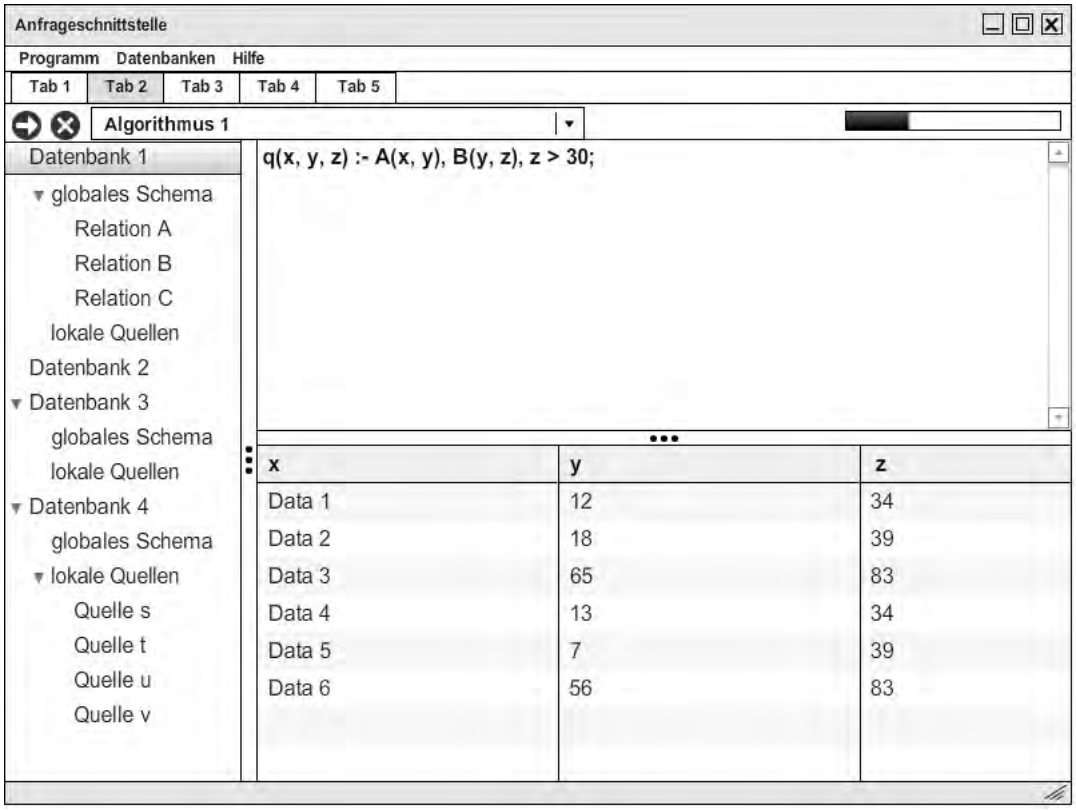


Abbildung 4.1: Mockup der grafischen Benutzerschnittstelle

Ein Menü soll sich am oberen Rand des Schnittstellenfensters befinden, mit dem man zum Beispiel die integrierten Datenbanken verwalten oder den Log anzeigen kann.

Um mehrere Anfragen parallel analysieren zu können, ohne die Anfrageschnittstelle in mehreren Fenster öffnen zu müssen, soll es die Möglichkeit geben, beliebig viele Registerkarten (Tabs) zu öffnen. Dafür soll sich unter dem Menü eine Tableiste befinden, in der beliebig viele Tabs geöffnet werden können.

Jedes Tab soll aus einer horizontalen Symbolleiste bestehen, die die Möglichkeiten zum Absetzen einer Anfrage (Pfeil nach rechts), Abbrechen einer laufenden Anfrage (Kreuz), Auswahl eines Anfrageplanungsalgorithmus (Auswahlliste) und die Anzeige der Progression einer abgesetzten Anfrage (Fortschrittsbalken) zur Verfügung stellen soll.

Unter der Symbolleiste soll sich in jedem Tab eine dreigeteilte Ansicht befinden. Auf der linken Seite sollen alle gespeicherten Datenbanken mit Relationen aufgelistet werden und es soll eine Datenbank ausgewählt werden können, an die die nächste Anfrage gestellt wird (im Mockup grau unterlegt). Rechts daneben befinden sich übereinander das Eingabefeld für die Anfrage des Benutzers sowie die Tabelle mit den letzten Ergebnistupeln. Diese drei Elemente der GUI sollen in ihrer Größe frei veränderbar sein, um zum Beispiel der Ergebnistabelle mehr Platz zu geben; dies deuten im Mockup die

Trennleisten mit den drei Punkten an.

Wie aus den funktionalen Anforderungen hervorgeht, lassen sich mit der in Abbildung 4.1 gezeigten Oberfläche nicht alle benötigten Funktionen durchführen. Im Menü soll es daher die Möglichkeit geben, zwei weitere Fenster zu öffnen. Das eine soll einen Log enthalten, der die Abläufe im Hintergrund nach dem Absetzen einer Anfrage dokumentiert. Das andere Fenster soll eine Möglichkeit zur Verwaltung von integrierten Datenbanken liefern. In Abbildung 4.2 sind die beiden benötigten Fenster dargestellt.

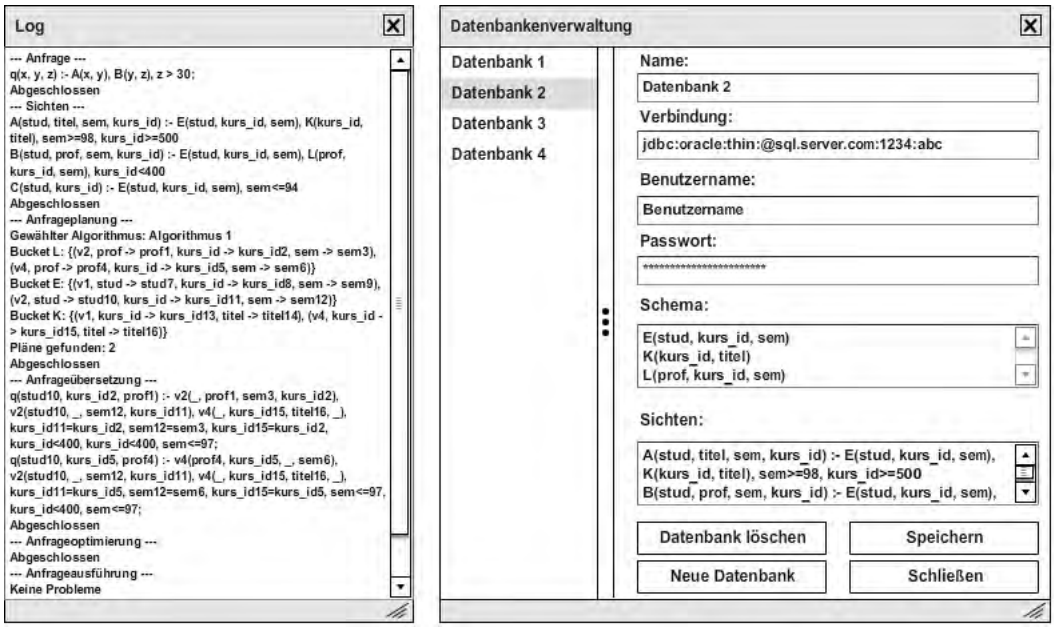


Abbildung 4.2: Mockup des Logs und der Datenbankverwaltung

Das Log (in der Abbildung links) soll ein Textfeld enthalten, aus dem man ablesen kann, was im Hintergrund der Anfragebearbeitung passiert.

Das Fenster zur Verwaltung der integrierten Datenbanken (in der Abbildung rechts) ist hingegen etwas komplexer: Auf der linken Seite soll eine Liste mit allen gespeicherten Datenbankverbindungen angezeigt werden. Wenn man eine Datenbank auswählt, sollen die gespeicherten Daten auf der rechten Seite in editierbaren Eingabefeldern angezeigt werden. Im unteren rechten Teil des Fensters sollen die verschieden Optionen für den Benutzer in Form von Buttons sein: Die aktuell ausgewählte Datenbank muss gelöscht oder gespeichert werden können, eine neue Datenbank muss hinzugefügt werden können und das Fenster soll ohne zu speichern geschlossen werden können.

4.1.4 Qualitätsanforderungen

Die wichtigste Qualitätsanforderung ist die Korrektheit des Programms, also die Korrektheit der Ergebnisse. Damit ist nicht gemeint, dass die Ergebnisse der realen Welt entsprechen, sondern dass die Ergebnisse so vollständig und so korrekt sind, wie es

die Datenquellen zulassen. Haben die Quellen eine schlechte Qualität, ist es nicht Aufgabe der Anfrageschnittstelle dies festzustellen und die Ergebnisse zu verbessern.

Des Weiteren ist es wichtig, dass die Anfrageschnittstelle effizient arbeitet. Nach dem Absetzen einer Anfrage an eine Datenbank sollen die Ergebnisse innerhalb kurzer Zeit berechnet werden. Natürlich hängt die tatsächlich benötigte Zeit immer von der jeweiligen Anfrage, den Extensionen der Quellen und der Bandbreite der Netzwerkverbindung ab. Enthält ein Ergebnis beispielsweise mehrere Gigabyte an Daten, so wird auch eine längere Antwortzeit in Kauf genommen, da alleine die Zeit, um eine solche Datenmenge über ein Netzwerk zu übertragen, einige Sekunden in Anspruch nehmen kann.

Die letzte sehr wichtige Qualitätsanforderung ist die Wartbarkeit des Systems. Es soll nach der Fertigstellung möglich sein, die Anfrageschnittstelle sinnvoll zu erweitern. Eine Erweiterung könnte zum Beispiel sein, einen zusätzlichen Algorithmus zur Anfrageplanung zu implementieren. Auch wenn zu einem späteren Zeitpunkt etwa die Anfrageübersetzung oder Ergebnisintegration verbessert werden soll, soll der Quellcode so gestaltet sein, dass dies mit einfachen Möglichkeiten realisiert werden kann. Vorstellbar ist auch, dass die Quellen einer Gesamtdatenbank nicht nur in SQL angesprochen werden sollen, sondern in späteren Versionen auch die Möglichkeit zur Anfrage an Quellen die XQuery[1] oder Datalog als Sprache verwenden. Die Anfrageübersetzung muss daher ebenfalls gut wartbar sein.

4.2 Entwurf

Aus dem vorherigen Abschnitt gehen eine Reihe von Anforderungen hervor, die die Entwurfsentscheidungen beeinflusst haben. Im Folgenden wird im ersten Abschnitt die gewählte Architektur der Software erläutert. Danach folgen einige Überlegungen, die dem Feinentwurf zuzuordnen sind.

Im letzten Abschnitt geht es um den Entwurf persistener Klassen. Damit ist die Speicherung der Verwaltungsdaten, um auf die einzelnen Quellen zugreifen zu können, gemeint. Diese Daten sollen in Klassen persistent gespeichert werden um sie über die Laufzeit eines Programmaufrufs hinweg benutzen zu können.

4.2.1 Architektur

Die für die Architektur gewählten Entwurfsmuster gehen aus den Anforderungen hervor, dass die Anfrageschnittstelle zum einen eine grafische Benutzeroberfläche haben und zum anderen nach dem Local-as-View-Paradigma vorgehen soll. Daraus ergeben sich zwei gute Kandidaten, die für diese Art von Problemen bestens geeignet sind. Zum einen das Model-View-Controller-Pattern, welches das Zusammenspiel von Benutzeroberfläche und Programmlogik organisiert und zum anderen das Pipes-and-Filters-

[1] XQuery: XML Query Language, wird spezifiziert durch das World Wide Web Consortium (Aktuelle Spezifikation: http://www.w3.org/TR/xquery/)

Pattern, welches sehr gut Systeme strukturiert, die einen Datenstrom bearbeiten. Beide Architekturmuster werden nachfolgend vorgestellt.

Model-View-Controller

Da die Anfrageschnittstelle eine GUI für die Interaktion mit dem Benutzer bereitstellen soll, bietet sich das Model-View-Controller-Pattern (MVC) an [KP88]. Dieses Architekturmuster unterteilt die Softwareentwicklung in drei Einheiten: das Datenmodell (model), die Präsentation (view) und die Programmsteuerung (controller). Das Ziel von MVC ist es, die drei Komponenten voneinander abzugrenzen, um so eine flexiblere Programmentwicklung, die eine spätere Änderung und Erweiterung ermöglicht, zu erreichen. MVC gilt als De-facto-Standard für den Softwareentwurf mit grafischen Benutzeroberflächen und wird deswegen an dieser Stelle nicht weiter erläutert.

Pipes-and-Filters

Die Anfragebearbeitung bei integrierten Datenbanken nach LaV entspricht einem Datenstrom: Nach der Eingabe einer Anfrage wird diese in mehreren Schritten bearbeitet, bis am Ende die Ergebnisse ausgegeben werden. Aus der Qualitätsanforderung der Wartbarkeit geht eine möglichst einfache Erweiterung der einzelnen Systemkomponenten hervor. Für diese Art von Problemen ist das Pipes-and-Filters-Pattern (PF) bestens geeignet. Es unterteilt ein System in mehrere sequentielle Verarbeitungsschritte. Diese sind durch den Datenfluss durch das System miteinander verbunden. Die Ausgabe eines Schrittes ist gleichzeitig die Eingabe für den nächsten. [BMR+98]

Filter-Komponenten sind dabei die Verarbeitungseinheiten einer Pipeline. Ein Filter ergänzt, transformiert oder verändert Eingabedaten (oder alles drei) und gibt diese weiter. Pipe-Komponenten stellen die Verbindung zwischen einer Datenquelle, den anschließenden Filtern und einer Datensenke her. In der Regel macht eine Pipe nichts anderes als die Ergebnisse von einem Filter an den nächsten weiterzureichen. Man unterscheidet zwischen aktiven und passiven Filtern. Während aktive Filter versuchen, Daten aus der jeweils vorgeschalteten Pipe zu „ziehen“, warten passive Filter bis die Pipe ihnen Daten zur Weiterverarbeitung gibt. Bei dieser Arbeit sollen alle Filter-Komponenten passiv sein. In Abbildung 4.3 sind die benötigten Pipe- und Filter-Komponenten abgebildet. Der Datenfluss geht von der Datenquelle bis zur -senke.

Die Linien in Abbildung 4.3 repräsentieren jeweils eine Pipe-Komponente. Bei doppelten Linien kann bereits ein Teilergebnis des Filters an den nächsten Filter weitergereicht werden. Zwischen Filtern, die im Bild mit einer einfachen Linie verbunden sind, macht dies dagegen wenig Sinn.

Die Datenquelle wird das Eingabefeld für die Benutzeranfrage, sowie die gewählte integrierte Datenbank sein, an die die Anfrage gestellt werden soll. Der Verarbeitungsprozess wird angestoßen, wenn der Benutzer die Anfrage ausführen möchte. Dann erhält der erste Filter, die Parsing-Komponente, die Eingabe und formt die Benutzeranfrage, die als Zeichenkette vorliegt, in die Datenstruktur für eine Anfrage um. Die geparste Anfrage kann dann von der Planungs-Komponente verwendet werden. In ihr

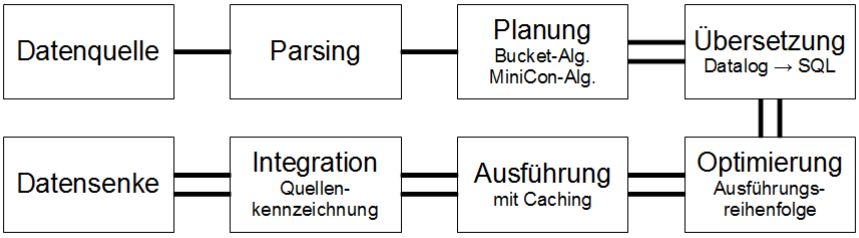


Abbildung 4.3: Benötigte Pipe- und Filter-Komponenten

wird mittels eines gewählten Anfrageplanungsalgorithmus die Anfrage umgeschrieben. So werden nach und nach alle Verarbeitungsschritte durchgegangen, bis die Daten in der Datensenke angelangen. Die Datensenke ist in diesem Fall die Ausgabe in der Benutzerschnittstelle.

Entscheidender Vorteil dieser Architektur ist, dass einzelne Komponenten gekapselt sind. Dies erzeugt eine hohe Kohäsion und eine geringe Kopplung. Im Implementierungsprozess lassen sich so außerdem zunächst sehr einfache Filter entwickeln, die später verbessert werden. Ein Beispiel wäre die Integrations-Komponente, die bei einer ersten Implementierung nur die Ergebnisse vereinigt, in einer späteren Version noch zusätzlich die Ergebnisse mit den Quellen kennzeichnet. Für den Planungs-Filter ist vorstellbar, dass zuerst nur ein Algorithmus entwickelt wird. Später kann ein neuer Planungs-Filter einen anderen Algorithmus implementieren. Bei der Wahl des gewünschten Algorithmus durch den Benutzer wird dann nur noch festgelegt, welche Planungs-Komponente für die Anfragebearbeitung verwendet wird.

4.2.2 Feinentwurf

Dieser Abschnitt beschäftigt sich mit konkreten Fragestellungen der Implementierung und legt fest, wie und nach welchen Mustern sie umgesetzt werden sollen. Dafür ist dieser Abschnitt in mehrere Unterabschnitte gegliedert. Der erste Unterabschnitt legt fest, mit welchen Hilfsmitteln die Benutzeroberfläche in Java implementiert werden soll.

Danach folgt ein Überblick über die Organisation der Klassen in Programm-Pakete (Packages). Auf die wichtigsten Packages wird dann in den Abschnitten „Datenmodell“ und „Algorithmen“ eingegangen.

Der Abschnitt schließt mit den Entwurfsentscheidungen für die Datenbankkommunikation.

Benutzeroberfläche

Für die Benutzeroberfläche soll die Standard Widget Toolkit (SWT)[2] Bibliothek in der Anwendung verwendet werden. Diese Open Source Bibliothek hat den Vorteil, dass sie reaktionsschneller und robuster ist als beispielsweise Swing[3]. Die Bedienelemente entstammen bei SWT dem Betriebssystem und müssen nicht emuliert werden. [Dau05]

Die Anforderungen an die GUI (siehe Abbildungen 4.1 und 4.2) lassen sich mit SWT vollständig umsetzen.

Da SWT kein fester Bestandteil der Java-Laufzeitumgebung ist, muss es manuell in die Anwendung eingebunden werden.

Package-Struktur

Die Klassen der Anfrageschnittstelle werden, wie bei der objektorientierten Programmierung unter Java üblich, in Packages organisiert. Klassen die thematisch zusammengehören oder das gleiche Problem behandeln, werden dabei im selben Package abgelegt. In Tabelle 4.1 ist eine Übersicht über die zu erstellenden Packages gezeigt.

Package	Beschreibung
controller	Klassen, die durch die Interaktion des Benutzers angesprochen werden
global	Klassen, die die Anwendung global beeinflussen, beispielsweise eine Konfigurations-Klasse
model	Klassen des eigentlichen Datenmodells, welche in mehreren Unterpackages verwendet werden
model.bucketalgorithm	Klassen, die zusätzlich zu den Modellklassen vom Bucket-Algorithmus benötigt werden
model.miniconalgorithm	Klassen, die zusätzlich zu den Modellklassen vom MiniCon-Algorithmus benötigt werden
model.pipesfilters	Klassen und Schnittstellen, die die Pipes-and-Filters-Architektur organisieren. Insbesondere finden sich hier die implementierten Komponenten der LaV-Anfragebearbeitung
view	Klassen, die zur grafischen Anzeige benötigt werden

Tabelle 4.1: Packagestruktur des Quellcodes

In den folgenden zwei Unterabschnitten wird auf die allgemeinen Modellklassen (Package model) und auf die spezifischen Modellklassen für die Algorithmen (Packages model.bucketalgorithm und model.miniconalgoritm) genauer eingegangen.

[2]SWT wird von der Eclipse Foundation entwickelt (URL: http://www.eclipse.org/swt/).

[3]Swing ist eine Grafikbibliothek, die fester Bestandteil der Java-Laufzeitumgebung ist.

Das Package model.pipesfilters wird sechs Schnittstellen (in Abbildung 4.3 zu sehen, außer Datenquelle und -senke) enthalten. Jede dieser Schnittstellen wird dann durch mindestens eine Klasse implementiert.

Datenmodell

Das model-Package enthält Klassen, die das zu Grunde liegende Datenmodell der Anfrageschnittstelle beschreiben. Die wichtigsten Klassen des Datenmodells sind in Abbildung 4.4 in einem UML Klassendiagramm[4] gezeigt. Eine vollständige Übersicht über alle Klassen befindet sich im Anhang B.

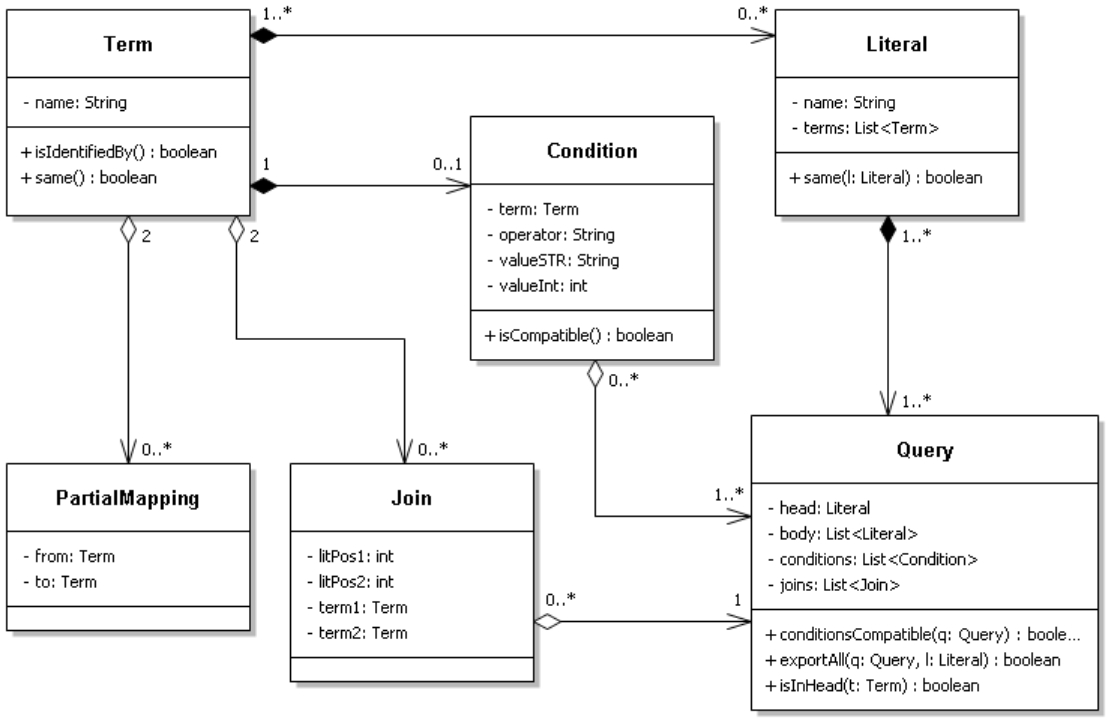


Abbildung 4.4: UML Diagramm der wichtigsten Klassen

Um eine konjunktive Anfrage zu beschreiben, die zum einen für die Formulierung von Benutzeranfragen und zum anderen zur Definition der Quellsichten gebraucht wird, gibt es die Klasse `Query`. Ein `Query` besteht dabei aus einem Kopfliteral und mindestens einem (meist mehreren) Rumpfliteralen. Des Weiteren kann sie Bedingungen und Verbunde enthalten.

Die Literale sind in einer eigenen Klasse `Literal` organisiert. Literale bestehen aus einem Namen und einer nicht leeren Menge von Termen. Bedingungen und Verbunde

[4]UML: Unified Modeling Language, wird spezifiziert durch die Object Management Group (Aktuelle Spezifikation: http://www.omg.org/spec/UML/).

sind ebenso eine eigene Klasse: Die Klasse für Bedingungen heißt `Condition` und enthält einen Term, einen Operator (=, <, $\leq$, $\geq$, >) und einen Wert, der entweder eine Ganzzahl oder eine Zeichenkette sein kann. Vereinfachend wird festgelegt, dass keine anderen Datentypen unterstützt werden, und bei Zeichketten nur der Gleichheitsoperator erlaubt ist.

Die Klasse für Verbunde heißt `Join` und enthält immer genau zwei Terme und deren absolute Position im Literal, um sie eindeutig zuordnen zu können. Terme wiederum sind in der Klasse `Term` organisiert.

Die letzte wichtige Klasse im Datenmodell ist die des `PartialMapping`. Da Terme auf andere Terme abgebildet (gemappt) werden können, ist diese Klasse dafür da, eine solche Verbindung zu beinhalten.

Datenbankkommunikation

Damit die Anwendung relativ unabhängig von den verwendeten Datenbankmanagementsystemen der Quellen arbeiten kann, bietet sich die standardisierte Datenbankschnittstelle JDBC[5] an. Sie verwendet SQL als Datenbanksprache, überwindet aber durch eine Abstraktion die Unterschiede zwischen verschiedenen SQL-Dialekten. Darüber hinaus ist es die Aufgabe von JDBC, Datenbankverbindungen aufzubauen, die SQL-Anfragen an die angesprochene Datenbank weiterzugeben und die Ergebnisse in für Java nutzbare Form zu bringen. [KE06]

Für jedes unterschiedliche Datenbanksystem, das in einer Java-Anwendung benutzt werden soll, sind eigene Treiber erforderlich, die die JDBC-Spezifikation implementieren. Die Hersteller der meisten Datenbank-Systeme liefern solche Treiber. In dieser Arbeit werden die Quellen auf Oracle Datenbanken betrieben. Um diese anzusprechen gibt es den Oracle JDBC Treiber, der kostenlos heruntergeladen und in eine Java-Anwendung eingebunden werden kann.

4.2.3 Algorithmenentwurf

Der Entwurf der zu implementierenden Algorithmen ergibt sich aus den formellen Beschreibungen des 3. Kapitels. Der Bucket-Algorithmus wird durch die Algorithmen 1 und 2 festgelegt, der MiniCon-Algorithmus durch die Algorithmen 3 und 4. Die dort formal angegebenen Algorithmen sind abstrakt formuliert, um sie kompakter anzugeben und verständlicher zu machen. In einer Implementierung werden jedoch konkrete Methoden und Vorgänge benötigt. Diese sind in den nachfolgenden Abschnitten erläutert.

Die Algorithmen verwenden dabei das oben erläuterte Datenmodell. Darüber hinaus werden für die beiden Algorithmen noch einige weitere Klassen benötigt, die nachfolgend erläutert werden.

[5] JDBC: Java Database Connectivity

Entwurf Bucket-Algorithmus

Für den Bucket-Algorithmus sind dies die Klassen `Bucket`, `BucketElement` und `Plan`. In Abbildung 4.5 sind deren Abhängigkeiten gezeigt.

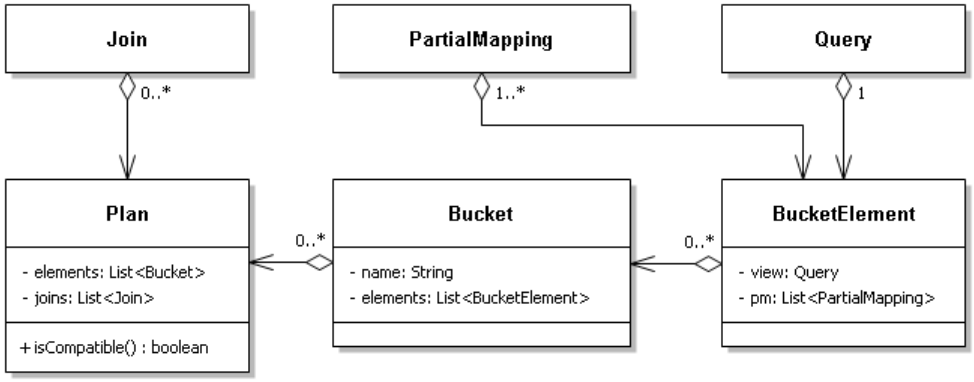


Abbildung 4.5: UML Diagramm der Klassen im Package model.bucketalgorithm

Die drei Klassen `Join`, `PartialMapping` und `Query` aus Abbildung 4.5 entstammen dem Package `model` (siehe oben). Ein Objekt der Klasse `Bucket` modelliert einen Bucket im BA, enthält also eine Menge von Sichten mit zugehörigen Mappings. Eine Sicht mit den zugehörigen partiellen Mappings wird in der Klasse `BucketElement` gekapselt. Wie in Abschnitt 3.2 gezeigt, werden die einzelnen Buckets miteinander kombiniert, so dass nach und nach ein Plan entsteht.

In der ersten Phase des Algorithmus werden für jedes Literal der Anfrage Buckets erstellt und mit Sichten gefüllt, die folgende Bedingungen erfüllen müssen:

- Sie müssen das gleiche Literal mindestens einmal beinhalten (`Literal.same(Literal l)`).
- Die Bedingungen der Sicht müssen mit den Bedingungen der Anfrage kompatibel sein (`Query.conditionsCompatible(Query q)`).
- Die Sicht muss alle Terme exportieren, die die Anfrage im Kopf und im aktuell betrachteten Literal enthält (`Query.exportAll(Query q, Literal l)`.

Für jeden dieser drei Punkte wird jeweils eine Methode benötigt, die in den Klammern angegeben ist. Im Erfolgsfall geben sie `true` und im Misserfolgsfall `false` zurück. Nur wenn eine Sicht alle drei Überprüfungen besteht, kommt sie in den Bucket.

In der zweiten Phase werden die Inhalte der Buckets zu Anfrageplänen kombiniert. Ein solcher (abstrakter) Anfrageplan ist als Klasse `Plan` modelliert und enthält eine Menge von Bucket-Elementen und eine Menge von Joins. Bei jeder Iteration wird dem Plan ein neues Element hinzugefügt. Anschließend muss die Methode `isCompatible`

aufgerufen werden, die `false` liefert, wenn ein Plan nicht kompatibel ist und verworfen werden kann. Die Methode gibt `true` zurück, wenn ein Plan kompatibel ist oder kompatibel gemacht werden kann. Tritt der letztere Fall ein, dass ein Plan nicht direkt kompatibel ist, aber kompatibel gemacht werden kann, fügt die Methode den oder die benötigten Joins in die Liste der Join-Elemente hinzu.

Zuletzt müssen die abstrakten Anfragepläne, die als `Plan`-Objekte vorliegen, noch zu richtigen Anfragen (Klasse `Query`) umformuliert werden. Dafür werden die Kopfliterale aller beteiligten Sichten zu Literalen im Rumpf der Anfrage. Dabei werden die Terme ersetzt, wie es das Mapping vorschreibt. Zusätzlich müssen noch die Bedingungen der Anfrage angehängt werden. Der Kopf des Anfrageplans ergibt sich aus dem Kopf der Benutzeranfrage, nur dass die Terme hier ebenfalls gemappt werden müssen.

Entwurf MiniCon-Algorithmus

Da der MiniCon-Algorithmus etwas anders arbeitet als der BA, werden für ihn die Klassen `Bucket`, `BucketElement` und `Plan` nicht benötigt. Stattdessen arbeitet der Algorithmus auf der etwas komplexeren Datenstruktur der MCDs (vgl. Definition 4 in Abschnitt 3.4). Außerdem gibt es die Klasse `Combination`, die ähnlich wie die Klasse `Plan` vom BA die Kombinationen strukturiert. Ein Objekt der Klasse `Combination` steht immer für eine Kombination von MCDs, also für einen potentiellen Anfrageplan.

Wie MCD- und Kombinations-Klassen in das bestehende Datenmodell eingebunden werden, ist im Klassendiagramm in Abbildung 4.6 dargestellt.

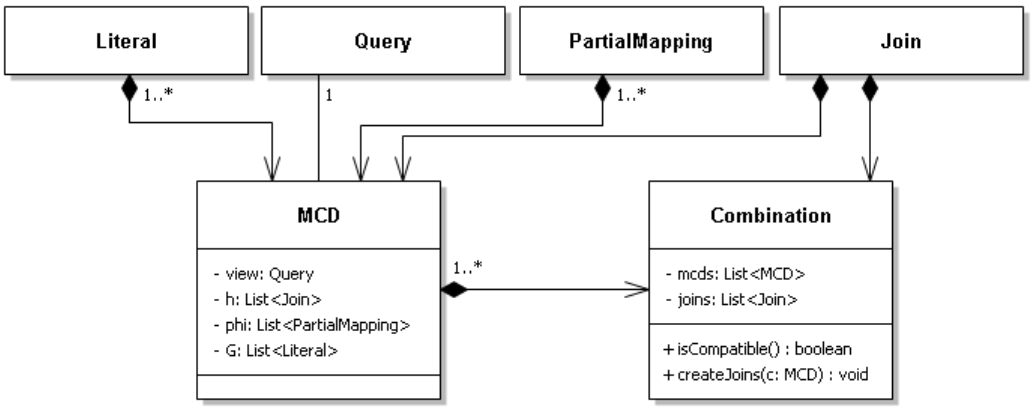


Abbildung 4.6: UML Diagramm der Klassen im Package model.miniconalgorithm

Aus dem Grundmodell des vorherigen Abschnitts werden die Klassen `Literal`, `Query`, `PartialMapping` und `Join` benötigt. Ein Objekt der Klasse `MCD` enthält genau ein Objekt der Klasse `Query`, da jede MCD für jeweils eine Sicht erstellt wird. Weiterhin wird eine Menge von partiellen Mappings für das Mapping φ benötigt. Der Homomorphismus der Kopfvariablen wird aus Gründen der einfacheren Implementierung mittels `Join`-Klasse organisiert. Ein Join steht hierbei immer für die Angleichung

von zwei Variablen. Jede MCD speichert weiterhin die Literale der Anfrage, die es überdeckt. Daher erhält jede MCD eine Menge von Literalen G.

Immer wenn ein neuer MCD zu einer bestehenden Kombination hinzugefügt wird, muss die Funktion `isCompatible` aufgerufen werden. Diese Funktion überprüft, ob die Bedingungen der Sichten der MCDs zueinander kompatibel sind.

4.2.4 Entwurf persistenter Klassen

Dieser Entwurf dient der Speicherung der Verwaltungsdaten der integrierten Datenbanken. In der internen Datenbank sollen keine integrierten Daten gespeichert werden. Jede integrierte Datenbank soll mit ihrem Namen, der JDBC-Adresse und den Zugangsdaten, bestehend aus Benutzernamen und Passwort, gespeichert werden.

Für das Mapping von Objekten innerhalb des Java-Programms in ein relationales Datenbankmodell gibt es die Java Persistence API (JPA), die es vereinfacht, Laufzeit-Objekte einer Java-Anwendung über eine einzelne Sitzung hinaus zu speichern (persistent zu halten).

Dabei werden Klassen, die beständig gespeichert werden sollen, im Quellcode mit Entity-Annotationen versehen. Jede Klasse die per Annotation als Entity gekennzeichnet ist, entspricht dabei üblicherweise einer einzelnen Tabelle in der relationalen Datenbank. Objekte einer Klasse sind die Zeilen in der Tabelle.

Um komplexe Beziehungen zwischen den Klassen zu kennzeichnen, zum Beispiel kann ein Objekt mehrere weitere Objekte enthalten, gibt es zusätzliche Annotationen. Die JPA übersetzt solche Beziehungen in gegenfalls benötigte weitere Tabellen und Attribute.

In Abbildung 4.7 sind die Klassen und deren Beziehungen zu sehen, die persistent gespeichert werden müssen.

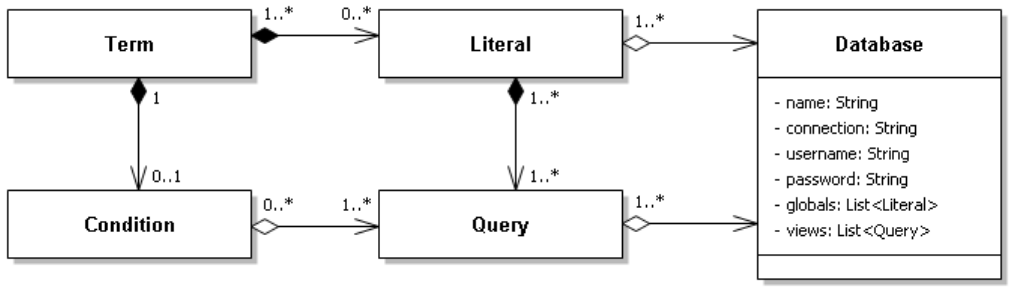


Abbildung 4.7: UML Diagramm der persistenten Klassen

Die Klassen `Term`, `Literal`, `Condition` und `Query` sind dabei schon aus den vorhergehenden Abschnitten bekannt. Die Klasse `Database` kapselt immer genau eine integrierte Datenbank. Die Attribute der Klasse sind die für die Verwaltung benötigten Informationen `name`, `connection`, `username` und `password`. Zusätzlich hat jede

Datenbank eine Menge von Literalen, die das globale Schema bilden und eine Menge von Sichten (`Query`-Objekte), die den Quellen entsprechen.

4.3 Implementierung

Die Implementierung folgt sehr stark dem Entwurf. Im ersten Abschnitt wird ein Überblick über die entwickelten Filter und die Umsetzung des Pipes-and-Filters-Pattern gegeben. In den danach folgenden Abschnitten wird die Implementierung der einzelnen Filter mit notwendigen Einschränkungen und Vereinfachungen detailliert beschrieben. Der letzte Abschnitt erläutert, wie die Anfrageschnittstelle getestet wurde.

4.3.1 Umsetzung des Pipes-and-Filters-Pattern

Die Vorteile des Pipes-and-Filters-Pattern wurden im Abschnit 4.2.1 erläutert. In diesem Abschnitt wird beschrieben, wie die Umsetzung erfolgte.

Um die Implementierung flexibel zu halten, wurde für jeden zu entwickelnden Filter eine Schnittstelle (Interface) geschrieben. Ein Filter implementiert also immer ein Filterinterface. Pipes wurden nicht verwendet. Jeder Filter ruft seinen Nachfolgefilter direkt auf. Jede Filtermethode erhält dafür eine Methode `write`, mit der Daten in den Filter geschrieben werden. Der Filter wird daraufhin aktiv, setzt seine Programmlogik um und ruft den nächsten Filter auf. In Tabelle 4.2 sind alle implementierten Filter aufgelistet. Diese befinden sich im Programmcode im Package `model.pipesfilters`.

Interface	Implementierte Filter
`IFilterParser`	`Parser`
`IFilterPlanner`	`PlannerBA`
	`PlannerMCA`
`IFilterTranslator`	`Translator`
`IFilterOptimizer`	`UndevelopedOptimizer`
`IFilterExecutor`	`SimpleExecutor`
`IFilterIntegrator`	`SimpleIntegrator`
	`Integrator`

Tabelle 4.2: Übersicht über implementierte Filter

Die Anfrageplanungsfilter können zur Laufzeit der Anfrageschnittstelle dynamisch gewählt werden. Alle anderen Filter, zu denen es mehrere Implementierungen gibt, können in der Konfigurationsklasse vor dem Kompilieren des Programms eingestellt werden.

Alle implementierten Filter werden in den nachfolgenden Abschnitten genau erklärt.

4.3.2 Parsing von Dataloganfragen

Um die Benutzereingabe in die Datenstruktur eines `Query`-Objektes zu bekommen, muss die eingegebene Zeichenkette geparst werden. Hierfür wurde die Klasse `Parser` implementiert, die die `IFilterParser`-Schnittstelle umsetzt. Dabei handelt es sich um eine Eigenentwicklung, die mit den Stringoperationen von Java die Eingabe zerlegt und in ein Objekt vom Typ `Query` umwandelt. Aus Gründen der schnelleren Implementierung wurde darauf verzichtet, einen Parser-Generator wie etwa JavaCC[6] oder Xtext[7] zu verwenden. Nachfolgend wird der grobe Ablauf des Parsing-Filters beschrieben.

Zuerst entfernt der Parser jeglichen Leerraum (Whitespace) aus der Benutzereingabe, der sich außerhalb von einfachen Anführungszeichen befindet. Alles, was in Anführungszeichen geschrieben ist, stellt eine Zeichenkette dar, die unverändert übernommen werden soll. Anschließend wird die von Leerraum befreite Eingabe anhand des Zeichens `:-` in einen Kopf und einen Rumpf geteilt.

Der Kopf besteht nur aus einem einzigen Literal und kann direkt als solches weiterverarbeitet werden. Der Rumpf besteht unter Umständen aus mehreren Literalen und Bedingungen. Diese müssen erst zu einzelnen Zeichenketten zertrennt werden und können dann zu `Literal`- und `Condition`-Objekten geparst werden.

Beschränkung der Ausdrucksstärke der Datalog-Anfragen

Wie in Abschnitt 2.2.2 beschrieben, können Datalog Anfragen sehr ausdruckstark sein. Dort wird aufgezeigt, dass Datalog mit Rekursionen und Negationen ausdrucksstärker als die Relationenalgebra ist. In dieser Arbeit wird sich daher nur auf konjunktive Anfragen beschränkt.

Es seien nur Bedingungen in den Anfragen von folgender Form erlaubt:

[Variable] [Vergleichsoperator] [Konstante]

wobei die Variable im Rumpf der Anfrage vorkommen muss. Der Vergleichsoperator muss aus dem Zeichenvorrat $\{=, <, >, <=, >=\}$ sein und die Konstante muss entweder eine ganze Zahl oder eine Zeichenkette in einfachen Anführungszeichen sein. Sollte die Konstante eine Zeichenkette sein, ist als Vergleichsoperator nur das Gleichheitszeichen zulässig.

Vergleiche zwischen zwei Variablen sind nicht möglich. Dies würde auch Probleme mit den Algorithmen nach sich ziehen, da diese immer davon ausgehen, dass Vergleiche nur mit Konstanten gemacht werden. Eine korrekte Bedingung könnte zum Beispiel

$$jahr > 2000$$

lauten, oder

[6] http://javacc.java.net/
[7] http://www.eclipse.org/Xtext/

$$name = \text{'Meyer'}.$$

Diese Beschränkung der Konstante auf lediglich zwei Datentypen ist sinnvoll, da jeder weitere unterstützte Datentyp viel Implementierungsaufwand mit sich bringen würde. Eine Erweiterung der Datentypen um beispielsweise Gleitkommazahlen oder Datumsangaben ist allerdings noch relativ einfach möglich, ist aber aus oben genannten Gründen in dieser Arbeit nicht vorgesehen.

4.3.3 Anfrageplanung mit dem Bucket-Algorithmus

Die Implementierung des Bucket-Algorithmus läuft wie folgt ab: Die Klasse `PlanerBA`, welche das Interface `IPlanner` implementiert, bekommt ein Objekt der Klasse `Database` und ein Objekt der Klasse `Query` vom Parser übergeben. Das Objekt `Query` ist die Benutzeranfrage. Im Objekt `Database` befinden sich alle Sichten, das globale Schema sowie die Daten, die für die Verbindung mit der Datenbank benötigt werden (u.a. Benutzername und Passwort).

Der BA benötigt allerdings nur die Anfrage des Benutzers und die Sichten aus dem Datenbank-Objekt. Der Algorithmus enthält die drei Hauptmethoden `createBuckets`, `createPlans` und `rewriteQueries`. Nachfolgend wird die Implementierung dieser drei Methoden genauer erläutert.

createBuckets

Diese Methode bekommt die Benutzeranfrage und alle Sichten (in Form einer Liste) übergeben. Über drei geschachtelte Schleifen werden dann die Buckets erzeugt und mit Sichten und Mappings gefüllt. Die äußerste Schleife iteriert über alle Literale in der Anfrage und erstellt für jedes Literal ein neues `Bucket`-Objekt.

Der BA soll für jede Sicht überprüfen, ob sie einen Teil zur Lösung der Anfrage beitragen kann. Dafür werden die nächsten beiden Schleifen gebraucht, die alle Sichten iterieren und jede Sicht wird widerum über alle Rumpfliterale iteriert.

Für jedes Literal jeder Sicht wird dann überprüft, ob es sich um das aktuelle Literal der Anfrage (äußerste Schleife) handelt. Außerdem muss für die aktuell betrachtete Sicht überprüft werden, ob die Bedingungen zur Anfrage kompatibel sind. Widersprechen sich die Bedingungen, hat die Anfrage zum Beispiel $x > 100$ und die Sicht $x < 50$ als Bedingung, kann die Sicht verworfen werden. Weiterhin wird überprüft, ob die Sicht alle relevanten Terme auch exportiert. Relevante Terme sind die, die im Kopf von der Anfrage und im Literal vorkommen.

Sind alle Anforderungen an die Sicht erfüllt, kann ein partielles Mapping von dem betrachteten Literal der Anfrage zur Sicht erstellt werden. Wenn es ein solches Mapping gibt, wird ein neues Objekt der Klasse `BucketElement` erstellt, welches eine Sicht und ein partielles Mapping beinhaltet. Das erzeugte Objekt wird dann dem Bucket hinzugefügt.

Alle erzeugten Buckets werden einer Liste hinzugefügt und am Ende der Methode zurückgegeben.

createPlans

Der implementierte Algorithmus hat mit `createBuckets` eine Liste von Buckets erzeugt. Diese werden der nun erläuterten Funktion `createPlans` übergeben, die daraus eine Liste von Objekten der Klasse `Plan` erzeugt. Dieser Schritt entspricht der zweiten Phase des BA. Er kombiniert die Inhalte der Buckets und überprüft jede entstehende Kombination auf Kompatibilität und Ausführbarkeit.

Die Funktion iteriert über alle Buckets. In der ersten Iteration gibt es noch keine `Plan`-Objekte. Für alle Elemente des ersten Buckets wird daher ein neuer möglicher Anfrageplan erstellt.

In der zweiten Iteration wird jeder Plan dann um alle Elemente des aktuell betrachteten Buckets erweitert. Jeder Plan wird dann auf Kompatibilität überprüft: Nur wenn alle Bedingungen der beteiligten Sichten sich nicht widersprechen und alle nötigen Joins ausgeführt werden können, wird der Plan weiter verfolgt. Ansonsten wird der Plan gelöscht.

Sollte am Ende einer Iteration kein Plan mehr übrig sein, kann die Funktion an der Stelle direkt abbrechen. Sind noch Pläne übrig nachdem alle Buckets betrachtet wurden, gibt die Funktion diese in einer Liste zurück. Die so erstellten Pläne stellen nur eine Sammlung von Sichten, Mappings und Joins dar und können noch nicht ausgeführt werden. Dafür müssen sie zu einer richtigen Anfrage umgeschrieben werden.

rewriteQueries

Für das Umschreiben der `Plan`-Objekte in `Query`-Objekte, die die späteren Schritte der Anfragebearbeitung verarbeiten können, ist die Funktion `rewriteQueries` zuständig. Diese Funktion bekommt eine Liste mit Plänen und die Benutzeranfrage übergeben.

Die Funktion iteriert über alle Pläne und erzeugt aus jedem eine Anfrage. Innerhalb jeden Plans wird wiederum über die enthaltenen `BucketElement`-Objekte iteriert. Dabei entspricht jedes Element im `Plan`-Objekt einem Literal, dass in der Anfrage enthalten sein muss. Wenn ein Literal der Anfrage hinzugefügt wird, muss immer auch das zugehörige Mapping eingesetzt werden. Ist eine Variable, die in einer Sicht vorkommt, nicht im Mapping enthalten, wird sie auf einen leeren Term gemappt (Symbol _). Während der Iteration über alle Elemente des Plans, werden alle partiellen Mappings gespeichert. Sie werden ganz am Ende benötigt, um die Terme im Kopf der Anfrage ebenfalls zu mappen.

4.3.4 Anfrageplanung mit dem MiniCon-Algorithmus

Da die Klasse `PlannerMCA` ebenso wie die Klasse `PlannerBA` das Interface `IPlanner` implementiert, bekommt sie ebenso ein Objekt der Klasse `Database` und ein Objekt der Klasse `Query` vom Parser übergeben.

Implementiert wird der MCA durch die zwei Methoden `formMCDs`, die der ersten Phase des Algorithmus entspricht und `combineMCDs`, die die zweite Phase organisiert.

Außerdem werden noch eine Reihe von Hilfsfunktionen benötigt. Nachfolgend werden die implementierten Methoden genauer erklärt.

formMCDs

Die Methode `formMCDs` bekommt die Benutzeranfrage und eine Liste von Sichten übergeben und erstellt daraus eine Liste von MCDs und gibt diese zurück.

Die äußerste Iteration geht über alle Sichten. Für jede Sicht wird überprüft, ob ihre Bedingungen mit den Bedingungen der Anfrage kompatibel sind (gleiche Funktion wie beim BA). Ist eine Sicht kompatibel, wird für jedes Literal der Anfrage überprüft, ob die Sicht das Literal überdeckt; ob es in der Sicht vorkommt. Kann die Sicht das Literal beantworten, wird versucht, ein partielles Mapping zu bilden. Wenn ein Mapping existiert, wird die MCD-Eigenschaft (Definition 5, Abschnitt 3.4.1) überprüft. Diese besagt, dass alle Variablen des Mappings exportiert werden müssen, oder wenn sie nicht exportiert wird und es sich um eine Join-Variable handelt, muss die Sicht alle beteiligten Joins ausführen.

Der Algorithmus überprüft also zunächst, ob es nicht-exportierte Variablen gibt. Wenn dies nicht der Fall ist, kann an dieser Stelle sofort eine MCD für die Sicht erstellt werden, wobei φ dem erstellten partiellen Mapping und die Menge der überdeckten Literale G genau dem aktuell betrachteten Literal entspricht. (Ein Homomorphismus muss nicht berechnet werden, da alle wichtigen Variablen exportiert werden).

Gibt es nicht-exportierte Variablen, wird für jede Variable kontrolliert, ob sie eine Join-Variable ist. Handelt es sich nicht um eine Join-Variable, braucht sie nicht weiter beachtet werden. Wenn die Variable aber an einem Join beteiligt ist, überprüft der Algorithmus, ob alle beteiligten Literale in der Sicht enthalten sind und die Joins ausgeführt werden können. Ist dies der Fall, wird eine MCD für die Sicht erstellt, wobei G die Menge aller überdeckten Literale ist.

Alle erstellen MCDs werden schließlich zu einer Liste zusammengefügt und von der Methode zurückgegeben.

combineMCDs

Die erzeugten MCDs müssen in der zweiten Phase des MCA zu Anfrageplänen umgeschrieben werden. Dafür ist die Funktion `combineMCDs` zuständig. Sie erhält eine Liste von MCDs und formt daraus `Query`-Objekte.

Das Finden von Kombinationen wurde durch eine rekursive Hilfsfunktion `findCombinationsFor` realisiert. Diese bekommt die Literale der Anfrage, die aktuell betrachtete Kombination (am Anfang ein neues `Combination`-Objekt) und die MCDs übergeben. Die Abbruchbedingung der Rekursion ist die Liste der Literale. Diese Liste enthält die Literale der Anfrage, die durch die aktuell betrachtete Kombination noch nicht abgedeckt sind. Ist diese Liste leer, kann die Funktion abbrechen und somit den rekursiven Aufruf stoppen.

Sind noch Literale in der Liste, wird über alle MCDs iteriert, die in der Kombination noch nicht vorkommen. Für jede MCD, die das erste Literal der Liste überdeckt und

ansonsten nur Literale überdeckt die noch in der Liste sind, wird die Funktion erneut aufgerufen.

Sei zum Beispiel eine Anfrage mit drei Literalen und vier MCDs gegeben. Wobei die MCDs die Literale der Anfrage wie folgt überdecken:

$$G_{C_1} = \{1\}, G_{C_2} = \{2,3\}, G_{C_3} = \{1,2,3\}, G_{C_4} = \{2\}$$

Wobei $G_{C_2} = \{2,3\}$ bedeutet, dass die MCD C_2 das zweite und dritte Literal der Anfrage überdeckt (beantworten kann). In Tabelle 4.3 sind die rekursiven Aufrufe der Funktion `findCombinationsFor` aufgelistet.

#	Literale	Kombination	MCDs	Beschreibung
0	1,2,3	$\emptyset$	C_1, C_2, C_3, C_4	Initiale Belegung
1	2,3	C_1	C_2, C_3, C_4	C_1 kann Literal 1 überdecken. Aufruf nur noch für die anderen Literale und MCDs
1	$\emptyset$	C_3	C_1, C_2, C_4	C_3 überdeckt alle Literale. Es sind keine Literale mehr übrig $\rightarrow$ Kombination gefunden
2	$\emptyset$	C_1, C_2	C_3, C_4	Die Kombination überdeckt alle Literale $\rightarrow$ Kombination gefunden
2	3	C_1, C_4	C_2, C_3	Es gibt keine MCD, die nur Literal 3 überdeckt $\rightarrow$ Abbruch, keine Kombination gefunden

Tabelle 4.3: Beispielaufrufe der Funktion findCombinationsFor

Initial sind beim Funktionsaufruf alle Literale der Anfrage und alle MCDs enthalten. Das erste Literal der Anfrage können die zwei MCDs C_1 und C_3 überdecken. Für beide wird jeweils die Funktion erneut aufgerufen, wobei die Anzahl der Literale um die Menge der von den MCDs überdeckten Literale reduziert wird. Die Kombination mit C_3 ist an dieser Stelle schon fertig, da es alle Literale überdeckt. Die Kombination C_1 ist noch unvollständig. Es werden beim nächsten Aufruf MCDs gesucht, die das Literal 2 beantworten können. In diesem Beispiel sind das die MCDs C_2 und C_4. Es entstehen also die Kombinationen $\{C_1, C_2\}$ und $\{C_1, C_4\}$. Die erste Kombination überdeckt nun alle Literale der Anfrage und ist somit fertig. Für die zweite Kombination fehlt noch das Literal 3, welches noch nicht überdeckt ist. Die Funktion wird also nochmal aufgerufen, dann wird allerdings keine MCD gefunden, die nur das Literal 3 beantwortet und die Kombination wird verworfen.

Immer wenn eine MCD zu einer bestehenden Kombination hinzugefügt wird, überprüft der Algorithmus, ob sich die Bedingungen der enthaltenen MCDs nicht widersprechen.

Die entstehenden Kombinationen sind Objekte der Klasse `Combination`. Ähnlich wie beim Bucket-Algorithmus müssen auch beim MiniCon-Algorithmus diese ab-

strakten Kombinationen noch zu richtigen Anfrageplänen umgeschrieben werden. Dafür ruft die Funktion `combineMCDs` für jedes `Combination`-Objekt die Hilfsfunktion `rewrite` auf, die aus einer Kombination einen Anfrageplan (`Query`-Objekt) formt.

Dabei geht die Funktion ähnlich vor wie die Funktion `rewriteQueries` beim BA und wird daher an dieser Stelle nicht genauer beschrieben.

Weitere Funktionen

Der MCA verwendet in den oben vorgestellten Methoden eine Reihe von Hilfsfunktionen. Die Wichtigsten werden nachfolgend kurz erläutert.

Die Funktion `getNotExportedPMs` bekommt eine Liste von partiellen Mappings, eine Sicht und die Benutzeranfrage übergeben. Sie gibt die relevanten Mappings zurück, die von der Sicht nicht exportiert werden. Ein Mapping ist dann relevant, wenn der zugehörige Term in der Anfrage exportiert wird oder er Teil eines Joins ist. Diese Funktion wird benötigt, um festzustellen, ob eine MCD einer Sicht nur ein Literal abdeckt oder mehrere. Gibt die Funktion mindestens ein partielles Mapping zurück, das relevant ist aber nicht exportiert wird, kann die MCD das Literal nicht alleine beantworten.

Tritt der Fall ein, dass eine MCD ein Literal alleine nicht abdeckt, wird die Funktion `getCoveredLiterals` aufgerufen, um alle überdeckten Literale zu finden. Dafür sucht die Funktion nach Literalen der Anfrage, die den übergebenden Join-Term enthalten. Werden mehr Literale gefunden, als die Sicht Literale hat, kann die Funktion abgebrochen werden. Ansonsten wird für alle beteiligten Literale der Anfrage nach passenden Literalen in der Sicht gesucht, die den Join ausführen. Am Ende werden alle gefundenen, überdeckten Literale zurückgegeben. Diese entsprechen dann dem G der MCD.

Wenn eine MCD mehrere Literale abdeckt, muss das Mapping φ neu berechnet werden und ggf. müssen die Kopfvariablen angeglichen werden (Homomorphismus). Dafür wurde die Funktion `generatePhiAndH` in der Klasse `MCD` implementiert. Diese stellt fest, wenn ein Mapping einen Term auf zwei unterschiedliche Terme mappen möchte und gleicht die Terme an.

4.3.5 Übersetzung von Dataloganfragen nach SQL

Wie in Abschnitt 2.2 erläutert, werden in dieser Arbeit Dataloganfragen auf Projektion, Selektion und Verbunde beschränkt. Die Anfrageübersetzung von Dataloganfragen zu SQL ist dadurch relativ einfach möglich.

In Abbildung 4.8 ist eine Beispielanfrage in Datalog gegeben, sowie eine äquivalente SQL-Anfrage. Zwischen den beiden Anfragen sind durch Pfeile die Beziehungen eingezeichnet. So werden die Terme im Kopf der Datalog-Anfrage zum `SELECT`-Statement in der SQL-Anfrage. Die Literale im Rumpf der Datalog-Anfrage kommen in den `FROM`-Teil bei SQL. Der Join der beiden Tabellen (über `studio=name`) sowie die Bedingung (`jahr<2000`) kommen in den `WHERE`-Teil und werden durch ein `AND` getrennt.

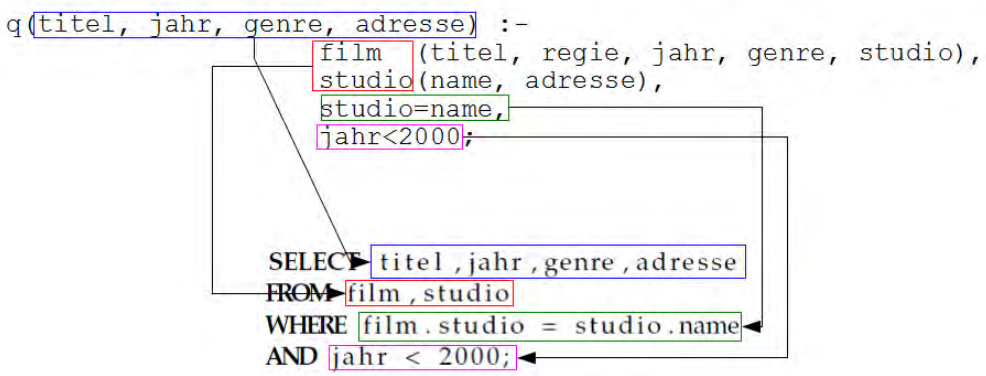


Abbildung 4.8: Zusammenhänge zwischen Datalog- und SQL-Anfragen

An diesem Beispiel sieht man, dass eine Anfrageübersetzung einfach möglich ist, wenn die entsprechenden Datenstrukturen vorliegen. In dieser Arbeit wurde das Interface `IFilterTranslator` durch die Klasse `Translator` implementiert. Diese bekommt ein `Database`-Objekt, welches, wie oben beschrieben, eine Sammlung von Quellen und eine Menge von Datalog-Anfragen (`Query`-Objekte) kapselt und formt daraus ausführbare SQL-Statements.

4.3.6 Optimierung

Wie in den Anforderungen beschrieben, wird in dieser Arbeit auf eine Optimierung der Anfragepläne verzichtet. Diese würde vermutlich keinen wesentlichen Geschwindigkeitsvorteil bringen, da alle Quellen immer auf der selben Datenbank liegen.

Dennoch wurde die simple Klasse `UndevelopedOptimizer` erstellt, die das Interface `IFilterOptimizer` enthält. Diese Klasse wurde nach dem PF-Pattern auch im Programmfluss eingegliedert. Sie macht aber nichts anderes, als die Ergebnisse der Anfrageübersetzung unverändert an die Ausführung weiterzureichen.

Sollte zu einem späteren Zeitpunkt eine Optimierung der Anfrageschnittstelle hinzugefügt werden, müsste eine solche Komponente lediglich die `IFilterOptimizer`-Schnittselle implementieren und es müsste anstatt des `UndevelopedOptimizer` zu den Filtern der Anfragebearbeitung hinzugefügt werden.

4.3.7 Anfrageausführung

Für die Anfrageausführung wurde die Klasse `SimpleExecutor` entwickelt. Dieser Filter erhält ein `Database`-Objekt und die übersetzten SQL-Anfragen vom Optimierer übergeben. Jedes SQL-Statement eines Anfrageplans wird von der Anfrageausführung zu einer großen SQL-Anfrage zusammengefasst. Dafür werden die Anfragen an die einzelnen Sichten in Inline-Views gesetzt und mit einem temporären Namen versehen, um sie eindeutig ansprechen zu können.

Die erzeugten Anfragen werden anschließend mit Hilfe der Zugangsdaten aus dem `Database`-Objekt nacheinander ausgeführt. Die Ergebnisse beim Ausführen einer Anfrage werden bei Java in `ResultSet`-Objekten gesammelt, über die man iterieren kann. Alle erzeugten `ResultSet`-Objekte werden in einer Liste an den nächsten Filter, die Ergebnisintegration, weitergegeben.

Diese Art der Anfrageausführung funktioniert nur, weil in dieser Arbeit davon ausgegangen wird, dass alle Quellen immer auf der selben Datenbank liegen. Wenn in einer späteren Version der Anfrageschnittstelle diese Vorraussetzung nicht mehr erfüllt ist, brauchen lediglich die `Database`-Klasse, die die Datenbankzugangsinformationen enthält und dieser Filter geändert werden (bzw. ein neuer Filter implementiert werden).

4.3.8 Ergebnisintegration

Wie in den Anforderungen beschrieben, beschränkt sich die Ergebnisintegration in dieser Arbeit auf das Vereinigen der Ergebnisse ohne eine Bereinigung der Daten vorzunehmen. Um die Herkunft der Daten für den Benutzer sichtbar zu machen, soll jedoch jeder Eintrag mit einem kurzen Quellenverweis gekennzeichnet werden.

Damit schnell eine lauffähige Version des Programms fertig gestellt werden konnte, wurde zunächst eine einfache Ergebnisintegration in Form der Klasse `SimpleIntegrator` implementiert. Die Integration bekommt die Benutzeranfrage und eine Liste von `ResultSets` von der Anfrageausführung übergeben. Die Benutzeranfrage wird dabei nur benötigt, um die Spaltentitel zu ermitteln. Die Spaltentitel sollen die Namen der Kopfvariablen der Anfrage sein.

Die Ergebnisintegration iteriert über alle erhaltenen `ResultSet`-Objekte und innerhalb dieser wiederum über alle Tupel und fügt sie in ein zweidimensionales Array vom Typ `String`. Eine Dimension des Array entspricht der Anzahl der Spalten, die andere Dimension ist ein in der Konfigurationsklasse festzulegender Wert. Gibt es mehr Tupel als dieser Wert groß ist, wird die Schleife abgebrochen und es werden nur die bis dahin erreichten Tupel übernommen. Dadurch wird die Anwendung vor zu großen Datenmengen und damit einhergehenden Programmabstürzen geschützt.

Am Ende der Ergebnisintegration werden die Spaltentitel in Form eines einfachen Arrays und die Inhalte der Ergebnistabelle in Form eines zweidimensionalen Arrays an die Datensenke übergeben. Diese sorgt dafür, dass die Daten in der Tabelle der Anfrageschnittstelle ausgegeben werden.

4.3.9 Test

Die Anfrageschnittstelle wurde einer Reihe Tests unterzogen. Dafür wurde das unter Java weit verbreitete Test-Framework JUnit[8] verwendet. JUnit ermöglicht das automatische, wiederholbare Testen von Units (Klassen oder Methoden). Dabei gibt es für jeden Test nur zwei mögliche Ergebnisse: Entweder der Test gelingt oder er schlägt fehl. Um

[8] http://www.junit.org/

JUnit zu verwenden, muss die JUnit-Programmbibliothek in die Java-Anwendung eingebunden werden. In dieser Arbeit wird JUnit eingesetzt, um wichtige, komplizierte und fehleranfällige Klassen und Methoden zu testen.

Alle Testfälle sind auf der beiligenden CD (siehe Anhang A), im selben Projektordner wie die Anfrageschnittstelle enthalten. Daher wird an dieser Stelle darauf verzichtet die Testfälle zu erläutern.

Kapitel 5

Experimente

In diesem Kapitel wird die entwickelte Anfrageschnittstelle praktisch eingesetzt. Dafür wird im ersten Abschnitt dieses Kapitels die zu Grunde liegende Versuchsumgebung, also das vewendete globale Schema und die Quellen, erläutert.

Anschließend folgen drei Abschnitte mit Experimenten. Im ersten Experiment werden die implementierten Algorithmen in ihrer Ausführung auf Schnelligkeit untersucht und verglichen. Im dritten Experiment wird die Zeit, die für die gesamte Anfragebearbeitung benötigt wird, untersucht. Im dritten und letzten Experiment wird die Vollständigkeit der Algorithmen analysiert.

5.1 Versuchsumgebung

In allen Experimenten werden das selbe globale Schema einer Filmdatenbank und die selben Quellen verwendet.[1] Das globale Schema ist in Tabelle 5.1 gezeigt.

Relation	Attribute
`Production`	`production, title, year, runtime`
`Budget`	`production, budget, currency`
`Genre`	`production, genre`
`Movie`	`production, type`
`Rating`	`production, rating`
`Person`	`person, name, sex, birth_country`
`Works`	`production, person, job, credit_pos`

Tabelle 5.1: Globales Schema der verwendeten Filmdatenbank

Bei den Attributen `production` und `person` handelt es sich um Schlüssel, die eine Produktion bzw. eine Person eindeutig identifizieren.

[1] Die Daten stammen aus der Internet Movie Database (IMDb) (Web: http://www.imdb.com/). Sie sind nur für den 2. Versuch relevant.

Die Tabelle `Movie` enthält den Typ des Films. Es handelt sich entweder um Kino-, Fernseh- oder Videofilme. In der Relation `Rating` sind Produktionen mit ihrer aktuellen Bewertung (0-10) enthalten.

In `Person` sind Personen gespeichert, die an einem Film mitgewirkt haben (Schauspieler, Regisseure, Drehbuchautoren...). Die Relation `Works` verbindet Personen mit Filmproduktionen. Sie enthält dabei die Aufgabe (`job`) der beteiligten Person und (wenn vorhanden) die Position der Nennung im Abspann.

Die Quellen, die in den Experimenten gegeben sind, sind in Tabelle 5.2 aufgelistet. In der Tabelle stehen in der ersten Spalte die Namen der Quellen. In der zweiten Spalte stehen die exportierten Variablen. Die dritte Spalte enthält die beteiligten Relationen des globalen Schemas und in der letzten Spalte stehen eventuell vorhandene Bedingungen.

Alle Quellen joinen die beteiligten Relationen jeweils über die Attribute `person` bzw. `production`. Außerdem wurden für diese Attribute bei jeder Quelle Indizes zum schnelleren Zugriff angelegt (sofern vorhanden). Die genauen SQL-Befehle, die beim Anlegen der Quellen verwendet wurden, sind auf der beiliegenden CD (siehe auch Anhang A) zu finden.

Jeder der folgenden Versuche wurde auf einem Notebook mit einem Intel Pentium Prozessor T4200 (2 GHz) und 4 GB RAM unter Microsoft Windows 7 Professional durchgeführt. Die Java SE Runtime Eviroment hatte die Version 1.7.0_03. Der JDBC-Treiber wurde in der Version 11.2.0.3.0 verwendet. Die Quellen befanden sich auf einem Oracle Database 11g Enterprise Edition Server in der Version 11.2.0.2.0 - 64bit.

Für die Zeitmessung wurde die Java-Funktion `System.nanoTime()` verwendet, welche die aktuelle Systemzeit in Nanosekunden zurückgibt. Durch die Differenz zwischen zwei gemessenen Zeitpunkten ergibt sich die Laufzeit.

5.2 Geschwindigkeitsexperiment Anfrageplanung

In diesem Versuch sollten der Bucket-Algorithmus und der MiniCon-Algorithmus, die in dieser Arbeit zur Anfrageplanung implementiert wurden, in ihrer Ausführungsgeschwindigkeit verglichen werden. Dafür wurde die oben beschriebene Filmdatenbank verwendet. Jeder Algorithmus führte die selbe Anfrage 10 mal aus. Die Zeit, die die Algorithmen dafür brauchten, wurde in Nanosekunden gemessen. Der Durchschnitt über die gemessenen Werte wurde als Ergebnis angesehen.

Heutige Systeme sind so performant, dass keine wesentlichen Geschwindigkeitsunterschiede zu erwarten waren. Um dennoch wenigstens einen Unterschied messen zu können, wurde die folgende Anfrage verwendet, die 6 globale Relationen joint und 4 Bedingungen enthält:

Name	Exp. Attribute	Relationen	Bedingungen
Actionfilm	production, title, year, runtime	Production, Genre	genre='Action'
Horrorfilm	production, title, genre, budget	Production, Genre, Budget	genre='Horror', currency='USD'
Kinofilm	production, title, year, runtime	Production, Movie	type='cinema'
Kurzfilm	production, title, runtime, budget	Production, Budget	runtime<60, currency='USD'
Gutethriller	production, rating	Genre, Rating	genre='Thriller' rating>=7
Kinoflops	production, rating, budget	Budget, Movie, Rating	budget>5M, currency='USD', type='cinema', rating<=6
Schauspieler	person, name, sex	Person, Works	job='actor'
SciFiPerson	person, name, sex, job	Person, Works, Genre	genre='Sci-Fi'
Regisseur	person, name, birth_country	Person, Works	job='director'
Arbeitet	production, person, job, credit_pos	Works	-
SpieltGut	production, person, credit_pos	Rating, Movie, Works	job='actor', type='cinema', rating>=7, credit_pos<=10

Tabelle 5.2: Verwendete Quellen/Sichten

```
q(title, budget, genre, rating, name, sex, job) :-
        Production(production, title, year, runtime),
        Budget(production, budget, currency),
        Genre(production, genre),
        Rating(production, rating),
        Works(production, person, job, credit_pos),
        Person(person, name, sex, birth_country),
        sex='m', genre='Horror', budget<10000000, rating<4
```

Die Anfrage sucht nach allen Filmen und beteiligten männlichen Personen (mit Titel, Budget, Genre, Bewertung, Name der Person, Geschlecht und Aufgabe) aus dem Horror-Genre mit einem niedrigen Budget (<10.000.000) und schlechter Bewertung (<4). Beide Algorithmen finden die gleichen 9 Anfragepläne.

Die gemessenen Werte sind in Tabelle 5.3 zu sehen. Der BA war in diesem Experiment knapp 4 ms langsamer als der MCA.

	BA	MCA
1	23,34 ms	22,72 ms
2	21,52 ms	20,63 ms
3	22,16 ms	21,43 ms
4	29,01 ms	21,43 ms
5	30,54 ms	19,77 ms
6	20,52 ms	17,55 ms
7	27,98 ms	14,99 ms
8	20,06 ms	22,10 ms
9	24,94 ms	22,66 ms
10	22,76 ms	21,94 ms
Durchschnitt	24,28 ms	20,52 ms

Tabelle 5.3: Messergebnisse Geschwindigkeitsexperiment

Es bestätigen sich also, wenn auch nur knapp, die Ergebnisse aus [PH01], in denen gezeigt wird, dass der MCA performanter als der BA ist.

Im Experiment dieser Arbeit ist das Ergebnis dennoch nicht sicher gewesen, da die Quellen alle Variablen exportieren, über die in der Anfrage gejoint wird (`production` und `person`), kann der MCA an dieser Stelle seine Vorteile nicht voll ausspielen. Beide Algorithmen stellen die gleiche Anzahl an Kombinationen auf. Es ist wahrscheinlich, dass die Kombinationsphase, die für den BA iterativ und für den MCA rekursiv implementiert wurde hier den Unterschied ausmacht. Iterative Algorithmen gelten im Allgemeinen als zwar performanter, im Einzelfall kommt es aber immer auf die jeweilige Umsetzung an. Diese ist in dieser Arbeit für den MCA effizienter gelungen.

5.3 Geschwindigkeitsexperiment gesamte Anfragebearbeitung

Wie im vorherigen Versuch gezeigt wurde, ist die Zeit, die für die Anfrageplanung gebraucht wird bei 11 Quellen und einer Anfrage über 6 Relationen vernachlässigbar gering. Dieser Abschnitt untersucht nun stichprobenartig die gesamte Zeit, die zur Bearbeitung einer Anfrage aufgewendet wird. Dafür wird die selbe Anfrage wie oben 10 mal ausgeführt. Für die Anfrageplanung wurde der etwas schnellere MiniCon-Algorithmus eingesetzt. Das System, auf dem die Anfrageschnittstelle lief, war das selbe, wie in der Versuchsumgebung beschrieben. Die Quellen lagen auf einem Server, der über das Internet angesprochen wurde. Der Client verfügte über eine Bandbreite von 9.415 kbit/s (Download) bzw. 952 kbit/s (Upload). Die Durchschnittswerte der einzelnen Schritte sind in Tabelle 5.4 zu sehen. Alle gemessenen Werte sind im Anhang C dargestellt.

Schritt	Durchschnittszeit	Anteil
Parser	17,23 ms	0,387%
Planung	18,44 ms	0,414%
Übersetzung	6,73 ms	0,151%
Optimierung	0,15 ms	0,003%
Ausführung	2906,27 ms	65,252%
Integration	841,58 ms	18,895%
Ausgabe	663,52 ms	14,898%
Gesamt	4453,92 ms	100,00%

Tabelle 5.4: Durchschnittswerte der Schritte bei der Anfragebearbeitung

Die gesamte Anfragebearbeitung vom Absetzen der Anfrage bis zur Anzeige des Ergebnisses dauerte bei diesem Experiment durchschnittlich 4,45 Sekunden. Den Hauptanteil mit 65% hat dabei die Anfrageausführung ausgemacht. Bei dieser Anfrage werden von der Anfrageschnittstelle 9 Anfragepläne erstellt, die nacheinander ausgeführt werden. Jeder der Anfragepläne enthält dabei wiederum eine Reihe von Unteranfragen. Dies verlangsamt die Anfrageausführung merklich. Außerdem sind die Quellen teilweise sehr groß: Die Tabelle `Arbeitet`, die in jedem Anfrageplan verwendet wird, enthält zum Beispiel über 10 Millionen Zeilen. Die genauen Tupelanzahlen der verwendeten Quellen sind im Anhang C aufgelistet.

Außerdem besteht noch Potential für die Optimierung der Quellen. Bei der Testdurchführung waren nur Indizes auf den Attributen `person` und `production` vorhanden. Mit weiteren Indizes könnte es zu einer Beschleunigung der Anfragen kommen.

Insgesamt hat dieses Experiment gezeigt, dass die Anfragebearbeitung mit integrierten Datenbanken, auch bei einer relativ komplexen Anfrage und relativ vielen Quellen, innerhalb weniger Sekunden durchgeführt werden kann.

5.4 Vollständigkeitsexperiment

Der BA findet nicht immer alle möglichen Anfragepläne und ist somit unvollständig. Der MCA hingegen ist ein vollständiger Algorithmus und sollte immer alle korrekten Anfragepläne finden. In diesem Versuch wurde überprüft, ob diese Voraussetzungen auch für die in dieser Arbeit getätigten Implementierungen zutrifft. Dafür wurden zwei verschiedene Anfragen an die Algorithmen gestellt und analysiert.

5.4.1 1. Versuch

Um die Unvollständigkeit des BA und die Vollständigkeit des MCA zu zeigen, bekommen beide Algorithmen folgende Anfrage gestellt:

```
q(person, name, sex, job) :-
        Person(person, name, sex, birth_country),
        Works(production, person, job, credit_pos),
        Genre(production, genre), genre='Sci-Fi'
```

Der BA findet für den Bucket `Person` die beiden Quellen `Schauspieler` und `SciFiPerson` und für den Bucket `Works` die Quellen `SciFiPerson` und `Arbeitet`. Für den Bucket `Genre` findet er keine Quelle, die mit der Bedingung `genre='Sci-Fi'` kompatibel ist und das benötigte Attribut `production` zum Joinen mit einer anderen Quelle exportiert. Wenn ein Bucket leer ist, kann der BA keine Kombinationen aufstellen. Er findet in diesem Versuch keinen Anfrageplan.

Der MCA hingegen stellt die folgenden vier MCDs auf, die in Tabelle 5.5 zu sehen sind (ohne h und φ).

V	G
`Schauspieler`	`Person`
`SciFiPerson`	`Person`
`SciFiPerson`	`Works, Genre`
`Arbeitet`	`Works`

Tabelle 5.5: Erzeugte MCDs im Vollständigkeitsexperiment

Die Relation `Genre` kann nur durch die Quelle `SciFiPerson` beantwortet werden. Da die zugehörige MCD auch gleichzeitig die Relation `Works` abdeckt, wird die Quelle `Arbeitet` für keinen Anfrageplan benötigt. Aus diesen MCDs kann der MCA also nur folgende zwei Kombinationen bilden, die zu folgenden Anfrageplänen werden:

- `q(person, name, sex, job) :- Schauspieler(person, name, sex), SciFiPerson(person, _, _, job)`
- `q(person, name, sex, job) :- SciFiPerson(person, name, sex, _), SciFiPerson(person, _, _, job)`

Dieses Ergebnis entspricht den Erwartungen. Da die Quelle `SciFiPerson` über eine Variable (nämlich `production`) joint, die sie nicht exportiert, kann der BA sie nicht benutzen. Der MCA hingegen kann mit solchen Joins umgehen und verwendet die Quelle in seinen Anfragepläne.

5.4.2 2. Versuch

Im zweiten Test wurde den Algorithmen die folgende Anfrage gestellt:

```
q(person, name, sex, birth_country) :- Person(person, name,
                    sex, birth_country)
```

Diese Anfrage entspricht der vollständigen Ausgabe der `Person`-Relation im globalen Schema. Keine Quelle kann diese Anfrage alleine beantworten, da keine Quelle alle vier Kopfvariablen der Anfrage exportiert. Der BA erzeugt daher einen leeren Bucket für die Relation `Person` und der MCA findet für keine Quelle eine MCD.

Die Algorithmen verhalten sich zwar wie erwartet, dieser Versuch zeigt jedoch, dass die Anfrageplanung mit den existierenden Algorithmen nicht optimal ist. Würden die Algorithmen die Möglichkeit in Betracht ziehen, dass eine Relation des globalen Schemas durch zwei Quellen beantwortet werden könnte, würden sie Anfragepläne finden können.

Zwei mögliche Anfragepläne für die oben gestellte Anfrage, wären:

- `q'(person, name, sex, birth_country) :- Schauspieler(person, name, sex), Regisseur(person, name, birth_country)`
- `q"(person, name, sex, birth_country) :- SciFiPerson(person, name, sex, _), Regisseur(person, name, birth_country)`

Die Quellen `Schauspieler` und `SciFiPerson` können jeweils die Attribute `person`, `name` und `sex` zum Ergebnis beitragen und die Quelle `Regisseur` kann das noch fehlende Attribut `birth_country` ergänzen. Gejoint werden können die Quellen zum Beispiel über `person` und/oder `name`. In den oben angegebenen möglichen Anfragepläne wird über beide Attribute gejoint, was aber nicht zwingend nötig ist. Joint man beispielsweise nur über `person`, können die Namen der Personen auch aus unterschiedlichen Quellen kommen.

Die bekannten Algorithmen finden diese Anfragepläne aber nicht, weil sie immer höchstens so lange Pläne erstellen, wie die Anfrage Literale hat. Dieses Vorgehen ist nach diesem Experiment in Frage zu stellen.

Ein wirklich vollständiger Algorithmus darf nicht nur darauf achten, ob eine Sicht ein Literal (oder mehrere) beantworten kann, sondern muss auch überprüfen, ob ein Literal eventuell von mehreren Sichten beantwortet werden kann.

Kapitel 6

Fazit und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde die Anfragebearbeitung in integrierten Datenbanken nach dem Local-as-View-Paradigma untersucht, umgesetzt und angewendet. Dafür wurde das Konzept hinter integrierten Datenbanken erläutert und das Local-as-View-Paradigma beschrieben. Die einzelnen Schritte der Anfragebearbeitung wurden erklärt. Die Anfrageplanung, insbesondere der Bucket- und der MiniCon-Algorithmus, wurden dabei sehr intensiv beschrieben und diskutiert. Diese Algorithmen wurden im späteren Verlauf der Arbeit implementiert.

Im Rahmen der Arbeit entstand eine Anfrageschnittstelle mit grafischer Benutzeroberfläche für integrierte Datenbanken in der Programmiersprache Java. Diese kann Anfragen in Datalog entgegennehmen und versucht sie, nach dem Local-as-View-Paradigm, mit einer Menge gegebener Quellen, möglichst optimal zu beantworten. Dafür erstellt sie mit einem gewählten Algorithmus Anfragepläne, übersetzt diese in SQL, führt sie an den Datenquellen aus, vereint die Ergebnisse und gibt sie schließlich in der Anfrageschnittstelle in tabellarischer Form aus. Das Programm kann mehrere integrierte Datenbanken speichern und verwalten. Über einen Log können die Berechnungen der Algorithmen und Verfahren, die im Hintergrund geschehen, angezeigt werden.

Die Implementierung wurde anschließend verwendet, um stichprobenartig einen Geschwindigkeitsvergleich zwischen dem Bucket- und dem MiniCon-Algorithmus zu erzeugen. Bei dem Vergleich war der MiniCon-Algorithmus knapp schneller. Ein zweites Geschwindigkeitsexperiment über die gesamte Anfragebearbeitung zeigte, dass integrierte Datenbanken Anfragen auch bei relativ komplexen Anfragen und relativ vielen Quellen innerhalb weniger Sekunden beantworten können. Außerdem wurde, in einem dritten Versuch, praktisch gezeigt, dass der Bucket-Algorithmus unvollständig ist.

Mit dem letzten Experiment wurde außerdem eine Schwachstelle der Anfrageplanung deutlich: Kein Algorithmus versucht eine Relation durch mehrere Quellen zu beantworten, wenn dies durch eine Quelle allein nicht möglich ist.

6.2 Fazit

Die Hauptmotivation für diese Arbeit war es, das Local-as-View-Paradigma und die zugehörigen Anfrageplanungsalgorithmen praktisch anzuwenden. Verständnisschwierigkeiten waren dabei vor allem beim MiniCon-Algorithmus vorhanden, deren Originalliteratur ([PH01]) sehr abstrakt und kompakt formuliert ist. Sekundärliteratur über den MiniCon-Algorithmus gibt es kaum. Jede gefundene Quelle stammte dabei aus dem Internet und verwendete die selben Beispiele, die auch in der Originalquelle zu finden sind. Erst sehr spät wurde der Algorithmus vollständig erfasst und konnte somit erst sehr spät korrekt implementiert werden. Da der Bucket-Algorithmus älter und verständlicher als der MiniCon-Algorithmus ist, gab es zu diesem mehr Quellen und die Implementierung war um einiges einfacher.

Insgesamt lässt sich feststellen, dass integrierte Datenbanken nach dem Local-as-View-Paradigma eine sehr sinnvolle aber auch komplexe Art der Integration darstellen. Sinnvoll deshalb, weil das Hinzufügen und Entfernen von Quellen zu einer integrierten Datenbank ohne großen Aufwand möglich ist und die Ergebnisse immer aktuell sind, da sie erst zur Laufzeit der Anfragebearbeitung aus den Quellen angefragt werden. Komplex sind integrierte Datenbanken deshalb, weil der Vorteil der einfachen Quellenänderung auf Kosten der Anfrageplanung geht. Diese muss flexibel aus dem aktuellen Zustand der Datenbank Anfragepläne erzeugen können und ist somit sehr vielschichtig.

Es war zwar möglich, alle wichtigen Funktionen in das Programm zu implementieren, die Anfrageschnittstelle kommt aber trotzdem nicht über einen prototypischen Stand hinaus. Optimierungen der Anfragepläne und ein robusteres Parsing konnten aus zeitlichen Gründen nicht umgesetzt werden.

6.3 Ausblick

Die in dieser Arbeit implementierte Anfrageschnittstelle ist funktionsfähig und macht Anfragen an eine integrierte Datenbank möglich. Dabei sind zwar alle Schritte der Anfragebearbeitung enthalten, diese sind aber unterschiedlich weit implementiert. Eine Optimierung der Anfragepläne findet beispielsweise überhaupt nicht statt: Der Optimierer gibt die ihm übergebenen Anfragepläne unverändert an die nächste Filterkomponente weiter. Wenn die Anfrageausführung an einem Anfrageplan scheitert, zum Beispiel weil eine Quelle nicht angesprochen werden kann, wird nur ein Fehlereintrag im Log vorgenommen, aber nichts unternommen. Denkbar wäre, dass die Anfrageplanung unter Ausschluss der defekten Quelle wiederholt wird. Da alle Schritte der Anfragebearbeitung in separaten, austauschbaren Komponenten gekapselt sind, lässt sich die Anfrageschnittstelle leicht erweitern und verbessern.

Integrierte Datenbanken machen auch eine Integration mit beschränkten Datenquellen möglich. Dies ist ein weiterer interessanter Aspekt, der in dieser Arbeit kein Thema war.

Die Schnittstelle lässt sich auch auf technischer Ebene noch verbessern. Derzeit werden die Schritte der Anfrageplanung seriell ausgeführt. Bei den meisten Schritten ist aber auch eine parallele Bearbeitung denkbar oder es ist möglich, bereits ein Teilergebnis dem nächsten Schritt zur Verarbeitung zu übergeben. Die aktuelle Implementierung arbeitet jeden Schritt bis zum Ende ab und ruft erst dann die nächste Komponente auf.

Nachfolgende Arbeiten könnten die einzelnen Schritte der Anfragebearbeitung behandeln, die in dieser Arbeit nicht im Fokus standen. Hierzu gehören insbesondere die Optimierung und die Ergebnisintegration. Beide Schritte sind in integrierten Datenbanken nach dem Local-as-View-Paradigma sehr komplex und facettenreich.

Auch können in nachfolgenden Arbeiten die bekannten Anfrageplanungsalgorithmen noch einmal genauer untersucht werden. Zum einen hat das letzte Experiment (s. Abschnitt 5.4.2) gezeigt, dass die bestehenden Algorithmen niemals mehrere Quellen verwenden um eine Relation zu beantworten (was, wie gezeigt wurde, sinnvoll sein kann). Zum anderen besteht die (theoretische) Möglichkeit, dass es noch einen schnelleren Algorithmus als den MiniCon-Algorithmus gibt.

Bedarf an der Integration von Daten wird vermutlich auch in Zukunft weiterhin bestehen, mit steigender Tendenz. Die Informationsmenge der Menschheit insgesamt und mit ihr auch die Anzahl an freien und proprietären Datenquellen, die für die Datenintegration nutzbar sind, wird in den nächsten Jahren weiter steigen. Integrierte Datenbanken haben dabei den Vorteil ständiger Aktualität und flexibler Zusammenstellung der Quellen. Sie sind somit ein guter Kandidat, um in der Zukunft verstärkt zum Einsatz zu kommen.

Literaturverzeichnis

[BMR+98] Buschmann, F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern-orientierte Softwarearchitektur, Addison-Wesley, 2. Auflage 1998.

[BSSV06] Boisson F., Scholl M., Sebei I., Vodislav D.: Query rewriting for open XML Data Integration Systems, IADIS International Conference WWW/Internet, 2006.

[Dau05] Daum B.: Java-Entwicklung mit Eclipse 3, dpunkt Verlag, 2005.

[GKD97] Genesereth M.R., Keller A.M., Duschka O.M.: Infomaster: An Information Integration System. ACM SIGMOD Int. Conference on Management of Data 1997, S. 539-542, Tuscon, Arizona, 1997.

[Hal01] Halevy A.Y.: Answering queries using views: a survey. VLDB Journal 10: 270–294, 2001.

[KE06] Kemper A., Eickler A.: Datenbanksysteme - Eine Einführung, Oldenbourg, 6. Auflage 2006.

[KP88] Krasner G.E., Pope S.T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80, Journal of Object Oriented Programming 1 (3), 26-49, 1988.

[Len02] Lenzerini M.: Data Integration: A Theoretical Perspective, 21st ACM Symposium on Principles of Database Systems (PODS) S. 233-246, 2002.

[Les98] Leser U.: Combining heterogenous data sources through query correspondence assertions. In Proc. of Workshop on Web Information and Data Management (WIDM98), 1998.

[Les00] Leser U.: Query Planning in Mediator Based Information Systems. Dissertation, Technische Universität Berlin, 2000.

[LN07] Leser U., Naumann F.: Informationsintegration, dpunkt Verlag, 2007.

[LRO96] Levy A.Y., Rajaraman A., Ordille J.J.: Querying Heterogeneous Information Sources Using Source Descriptions. VLDB S. 251-262, 1996.

[Mit99] Mitra P.: An algorithm for answering queries efficiently using views, Infolab Stanford University, 1999.

[ÖV99] Özsu M.T., Valduriez P.: Principles of distributed database systems, Prentice-Hall, 2. Auflage 1999.

[PH01] Pottinger R., Halevy A.: MiniCon: A scalable algorithm for answering queries using views. VLDB Journal (2001) 10: 182-198, 2001.

[SMW04] Skulschus M., Michaelis S., Wiederstein M.: Oracle 10g Programmierhandbuch, Galileo Computing, 1. Auflage 2004.

[Ull97] Ullman J.D.: Information integration using logical views, Proc. of ICDT. S. 19-40, Springer, 1997.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AQUV	Answering Queries Using Views
BA	Bucket-Algorithmus
GaV	Global-as-View
GLaV	Global-Local-as-View
GTA	Generate-and-Test-Algorithmus
GUI	Graphical User Interface (Grafische Benutzerschnittstelle)
IBA	Improved-Bucket-Algorithmus
IRA	Inverse-Rules-Algorithmus
JDBC	Java Database Connectivity
JPA	Java Persistence API
LaV	Local-as-View
MCA	MiniCon-Algorithmus
MCD	MiniCon Description
MVC	Model-View-Controller
PF	Pipes-and-Filters
SVB	Shared-Variable-Bucket-Algorithmus
SWT	Standard Widget Toolkit
XML	Extensible Markup Language

Anhang A

Inhalt der beiliegenden CD

Dieser Arbeit liegt eine CD bei, auf der sich dieses Dokument in digitaler Form sowie die entwickelte Anfrageschnittstelle als Quellcode mit Dokumentation befindet. In der unten stehenden Tabelle ist eine Erklärung zu den Verzeichnissen zu sehen.

Verzeichnis	Inhalt
Arbeit	Diese Arbeit im pdf-Format in zwei Versionen: Eine mit anklickbaren Verweisen (z.B. im Inhaltsverzeichnis) und eine ohne anklickbare Verweise.
ArbeitQuelletext	Diese Arbeit als Latex-Quelltext mit allen Abbildungen und SQL-Befehlen.
Dokumentation	Die technische Dokumentation der Anfrageschnittstelle
Experimente	Die Definitionen der Quellen, die bei den Experimenten verwendet wurden und alle Messergebnisse im OpenDocument Tabellendokument
Programm	Der Quellcode der entwickelten Anfrageschnittstelle

Anhang B

Klassen der Anfrageschnittstelle

In den folgenden Abbildungen sind alle Klassen der erstellten Anfrageschnittstelle in UML Klassendiagrammen dargestellt. Dabei entspricht jede Abbildung genau einem Package. Assoziationen zwischen den Packages sind nicht dargestellt. Es sind keine Getter-, Setter-, toString- und Konstrukter-Methoden angegeben.

Abbildung B.1: UML Klassendiagramm des Package controller

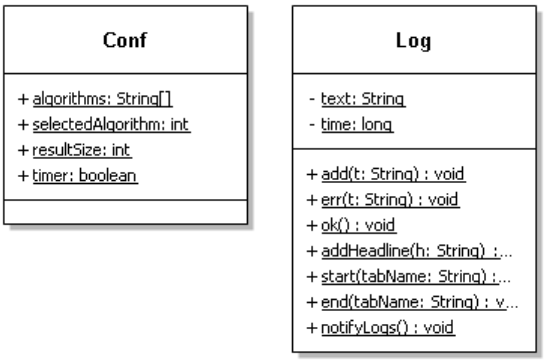


Abbildung B.2: UML Klassendiagramm des Package global

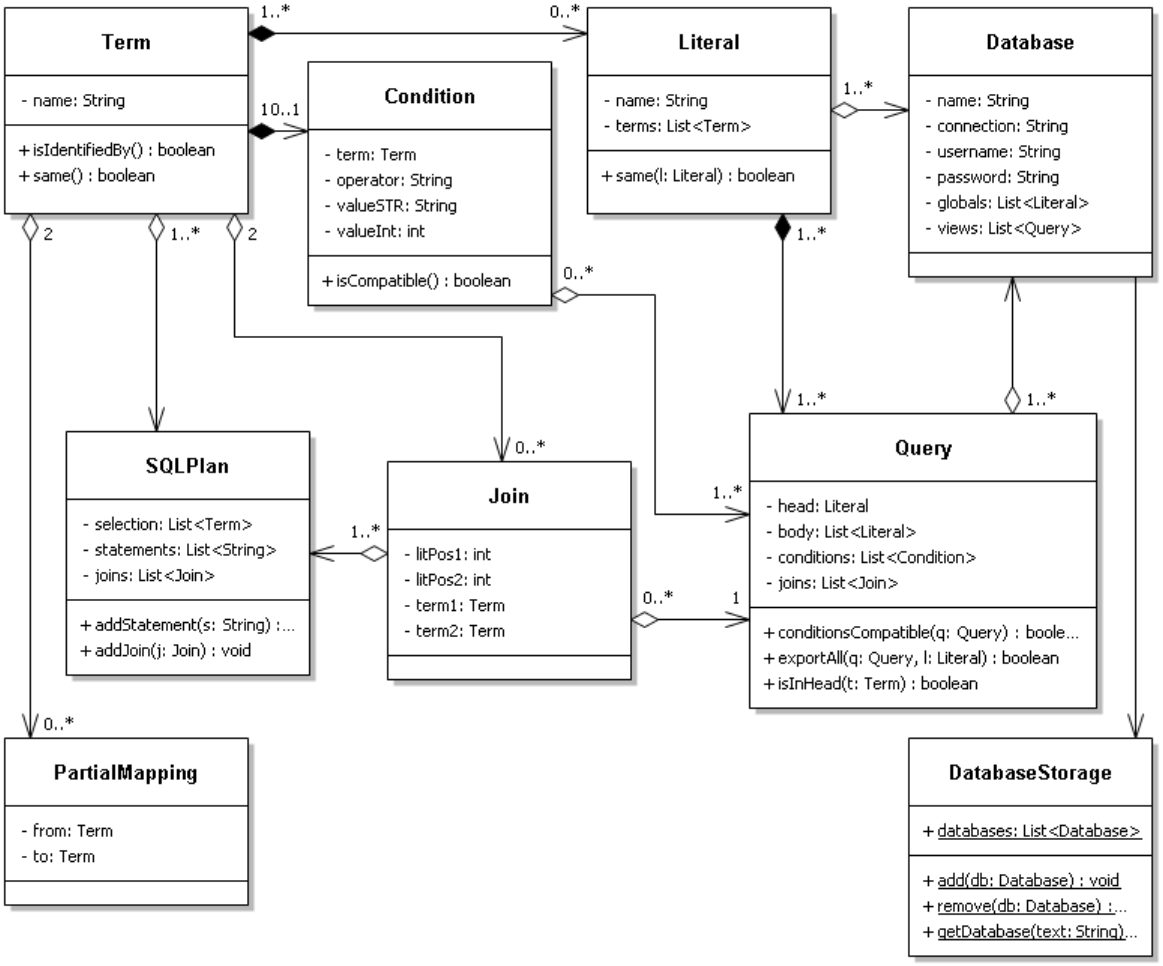


Abbildung B.3: UML Klassendiagramm des Package model

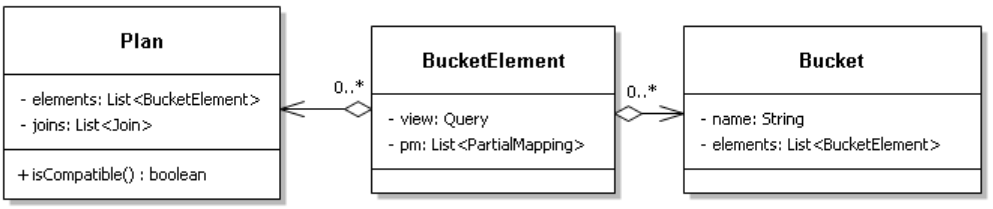


Abbildung B.4: UML Klassendiagramm des Package model.bucketalgorithm

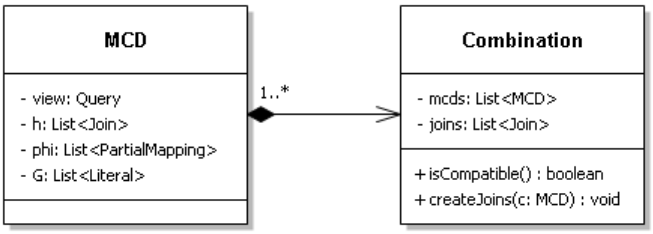


Abbildung B.5: UML Klassendiagramm des Package model.miniconalgorithm

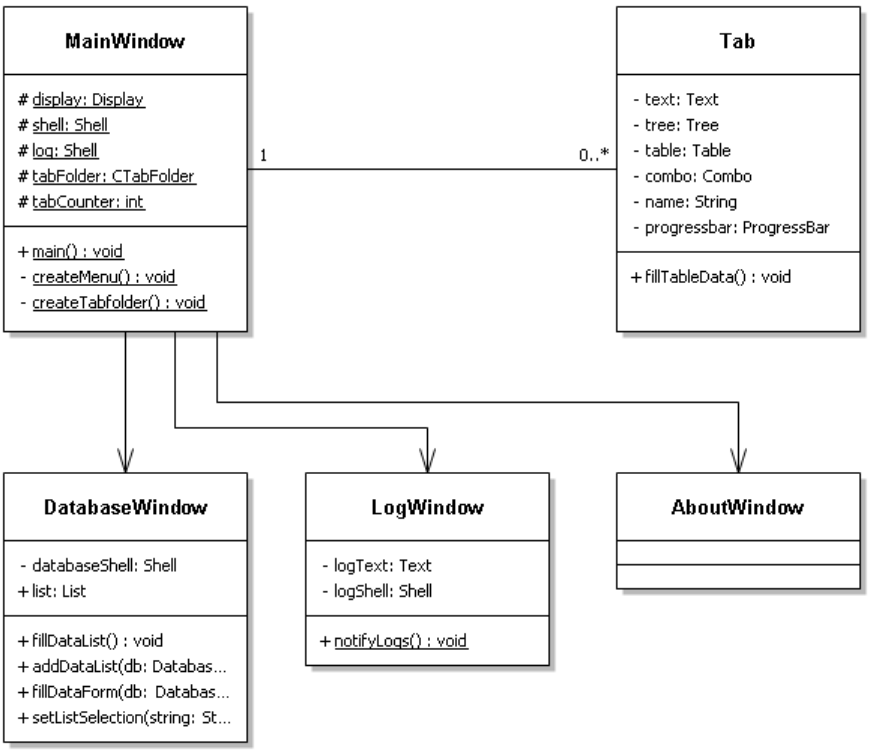


Abbildung B.6: UML Klassendiagramm des Package view

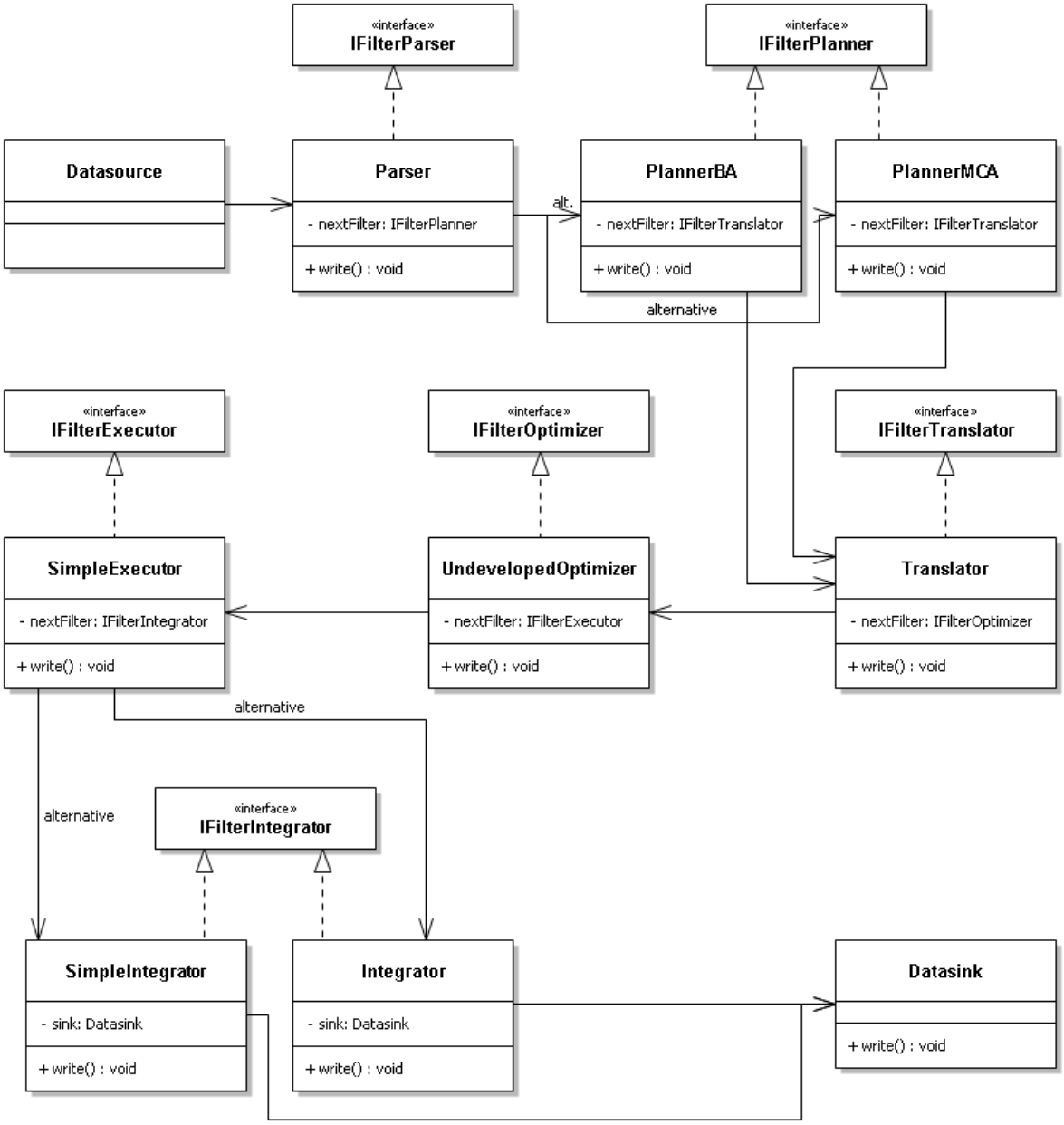


Abbildung B.7: UML Klassendiagramm des Package model.pipesfilters

Anhang C

Messwerte und Quellengröße

In der Tabelle C.1 sind alle gemessenen Werte des Geschwindigkeitsexperiments aufgeführt. In der untersten Zeile stehen die Durchschnittswerte, die sich aus den Messungen ergeben. In der Tabelle C.2 sind alle verwendeten Quellen mit ihren Anzahl Tupeln aufgelistet.

	Parser	Planung	Übers.	Opti.	Ausführ.	Integra.	Ausgabe	Gesamt
1	17,05	20,27	6,60	0,15	2985,47	821,88	661,55	4512,97
2	19,90	14,79	6,39	0,14	2907,47	834,20	671,05	4453,94
3	17,32	21,01	7,94	0,15	2807,01	835,57	666,22	4355,22
4	16,81	21,09	7,26	0,15	2869,13	832,00	664,99	4411,43
5	17,14	19,00	6,65	0,16	2893,89	847,98	667,88	4452,70
6	16,65	18,93	6,50	0,15	2894,00	849,22	637,88	4423,33
7	17,10	15,04	6,47	0,15	3058,70	834,57	655,84	4587,87
8	17,01	17,98	6,42	0,15	2923,51	848,09	673,04	4486,20
9	16,56	18,82	6,53	0,15	2894,41	879,97	679,10	4495,54
10	16,74	17,44	6,51	0,15	2829,12	832,35	657,67	4359,98
	17,23	18,44	6,73	0,15	2906,27	841,58	663,52	4453,92

Tabelle C.1: Messwerte (in ms) des Geschwindigkeitsexperiments mit allen Schritten der Anfragebearbeitung

Quelle	Tupel	Quelle	Tupel	Quelle	Tupel
`Actionfilm`	3.593	`Schauspieler`	133.592	`Kinoflops`	1.692
`Horrorfilm`	1.114	`SciFiPerson`	39.070	`Gutethriller`	939
`Kinofilm`	17.684	`Regisseur`	16.370	`SpieltGut`	31.294
`Kurzfilm`	55	`Arbeitet`	10.823.736		

Tabelle C.2: Größe der Quellen aus dem Experiment

Erklärung der Selbstständigkeit

Ich versichere, die vorliegende Masterarbeit selbstständig, ohne fremde Hilfe und nur unter Verwendung der von mir aufgeführten Quellen und Hilfsmittel angefertigt zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem Prüfungsamt vorgelegen.

Hannover, 07. November 2012

_____

Timo Hüther