Fakultät für Mathematik und Informatik
Lehrgebiet Datenbanksysteme für neue Anwendungen
Prof. Dr. Ralf Hartmut Güting

Zwei Probleme der Anfrageoptimierung: Behandlung geschachtelter Relationen und Speicherzuteilung an Operatoren

Bachelorarbeit

Nikolai van Kempen
Hauptstraße 55
46446 Emmerich am Rhein
Matrikel-Nr. 6221467
November, 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Auf dem Gebiet der Datenbanken hat sich im Laufe der Jahre eine Anfragesprache für relationale Datenbanksysteme (RDMS)[1] etabliert, die Structured Query Language (SQL). SQL ermöglicht die einfache Formulierung von Anfragen auf dem relationalen Datenmodell. Ein Beispiel für eine Relation zeigt die Tabelle[2] 1.1[3]. Die Relation *Orte* besteht aus sechs Tupeln mit den Attributen *Kennzeichen, Ort, Vorwahl* und *BevT* für die Bevölkerung des Ortes angegeben in Tausend.

Tabelle 1.1: Die Relation *Orte*

Kennzeichen	Ort	Vorwahl	BevT
B	Berlin	030	3410
HA	Hagen	02331	208
BN	Bonn	0228	291
GE	Gelsenkirchen	0209	288
KLE	Kleve	02821	46
WES	Wesel	0281	62

Auf Basis der Relation *Orte* kann die SQL Anfrage 1.1[4] an das Datenbankmanagementsystem (DBMS) gestellt werden. Das Ergebnis dieser Anfrage wären die Orte in alphabetisch sortierter Reihenfolge, deren Bevölkerung die Anzahl von 100 000 Einwohnern übersteigt.

Listing 1.1: Beispielanfrage auf der Relation *Orte*

```
1 select  ort
2 from    orte
3 where   bevt >= 100
4 orderby ort
```

Das klassische relationale Modell nach E.F. Codd [Cod70] verlangt für Relationen, dass die dort enthaltenen Werte atomar sind und somit nicht weiter zerlegt werden können. Ein Ortsname

[1] Mehr zu den RDBMS im Abschnitt 2.1

[2] Tabellen werden üblicherweise verwendet um Relationen darzustellen, auch wenn es sich bei den beiden Begriffen nicht um dasselbe handelt.

[3] Die Daten enstammen der *opt* Datenbank die in dem Secondo DBMS enthalten ist.

[4] Hier wird schon im Vorgriff auf Abschnitt 2.3.2 auf Seite 10 der SQL-Dialekt des Secondo DBMS verwendet.

wie „Hagen“ genügt dieser Anforderung, aber ein Wert wie z.B. „58089 Hagen“ ist noch weiter zerlegbar, und zwar in die atomaren Werte „58089“ und „Hagen“. Dass diese Eigenschaft verlangt wird, ist Teil der *ersten Normalform* (first normal form). Wird diese Beschränkung aufgehoben, werden damit auch relationenwertige Attribute erlaubt, also Attribute, die Relationen enthalten. Relationen dieser Art sind geschachtelte Relationen.

Tabelle 1.2: Die Relation *OrteH*

BevH	SubRel			
	Kennzeichen	Ort	Vorwahl	BevT
34	B	Berlin	030	3410
2	HA	Hagen	02331	208
	BN	Bonn	0228	291
	GE	Gelsenkirchen	0209	288
0	KLE	Kleve	02821	46
	WES	Wesel	0281	62

Die in Tabelle 1.2 abgebildete geschachtelte Relation *OrteH* besteht auf der obersten Ebene aus zwei Attributen, *BevH* und *SubRel*. *SubRel* ist relationenwertiges Attribut, das eine Relation aus den Attributen *Kennzeichen, Ort, Vorwahl* und *BevT* darstellt. Die Relation *OrteH* beinhaltet dabei auf der obersten Ebene drei Datensätze. Weitere Datensätze finden sich aber in der eingebetten Relation *SubRel*. Für den zweiten Datensatz der Relation *OrteH* sind zum Beispiel drei weitere Datensätze in der Relation des Attributs *SubRel* vorhanden.

Die Anfrageoptimierung ist eine Komponente, die Bestandteil der meisten Datenbankmanagementsysteme (DBMS) ist. Aufgabe der Anfrageoptimierung ist es, für Anfragen, die an das DBMS gestellt werden, einen möglichst effizienten Ausführungsplan zu erzeugen. Ein Ausführungsplan ist dabei eine genau definierte Abfolge von Schritten, die vom DBMS ausgeführt werden, um das Ergebnis der Anfrage zu ermitteln. Anfragen an Datenbanksysteme werden dabei in der Regel mit Hilfe deklarativer Sprachen [Cha98], wie dem bekanntesten Vertreter, der Structured Query Language (SQL), formuliert. Charakteristisch für deklarative Sprachen ist die Eigenschaft, dass beschrieben wird, *was* das Ergebnis sein soll. Eine Vorgabe, wie dieses Ergebnis ermittelt werden soll, ist kein Bestandteil deklarativer Sprachen, sondern ist hier die Aufgabe des Anfrageoptimierers. Eine Anfrage kann dabei nicht nur in einen Ausführungsplan übersetzt werden, da es in Regel zu einer Anfrage viele mögliche Ausführungspläne geben wird, die aber immer das gleiche Ergebnis erzeugen. Der Anfrageoptimierer sollte daher in der Lage sein, unter all diesen möglichen Ausführungsplänen den besten Ausführungsplan möglichst effizient zu ermitteln. Der *beste* Ausführungsplan ist dabei der Ausführungsplan, der unter dem zugrundeliegenden Kostenmaß die geringsten Kosten aufweißt. In der Regel setzt sich das Kostenmaß aus der CPU Zeit und den I/O Kosten zusammen und modelliert damit die Ausführungszeit [Cha98].

Ein Ausführungsplan konkretisiert im Gegensatz zu einer Anfrage wie das Ergebnis ermittelt

werden soll. Dazu bedient sich der Anfrageoptimierer der Operatoren, die im DBMS implementiert sind. Ein wichtiger Punkt, der für diese Arbeit relevant ist, ist, wie ein Operator mit der Ressource Arbeitsspeicher umgeht. Ein Beispiel für einen solchen Operator ist der *sort* Operator, der für eine sortierte Ausgabe sorgt. Passen nicht alle Tupel in den Arbeitsspeicher, der dem Operator zur Verfügung steht, gibt es für den *sort* Operator die Möglichkeit trotzdem mit dem Arbeitsspeicher auszukommen, indem er Daten auf einen Sekundärspeicher auslagert. Speichermedien, die tiefer in der Speicherhierachie sind, sind dabei langsamer, bieten aber mehr Speicherplatz bei geringeren Kosten pro Byte [Sta10, Seite 107-109]. Es ist daher am vorteilhaftesten, wenn alle Tupel im Arbeitsspeicher sortiert werden können. Allerdings ist die Annahme, dass immer genug Arbeitsspeicher zur Verfügung steht, unrealistisch [ND98]. Wird ein Operator gezwungen, seine Daten auszulagern, erhöhen sich seine Kosten aufgrund der Verwendung der langsameren Sekundärspeicher. Da nun in einem Ausführungsplan in der Regel mehr als ein Operator an der Ausführung beteiligt ist, konkurrieren diese um den vorhanden Arbeitsspeicher. Wird einem Operator mehr Arbeitsspeicher gewährt, kann dieser Operator mit geringeren Kosten seine Aufgabe absolvieren, allerdings steht dann den anderen Operatoren wieder weniger Arbeitsspeicher zur Verfügung mit der Konsequenz, dass deren Kosten steigen, sofern nicht genügend Arbeitsspeicher für alle Operatoren zur Verfügung steht. Dabei unterscheiden sich die Operatoren nicht nur hinsichtlich ihrer Kosten, die vom zugewiesenen Arbeitsspeicher abhängen. Insbesondere unterscheiden sich die Operatoren auch darin, inwieweit diese von mehr zugewiesenem Arbeitsspeicher profitieren können. Beispielsweise kann ein Operator A seine Kosten um eine Sekunde pro mehr verfügbaren Megabyte an Arbeitsspeicher senken, ein Operator B aber um 2 Sekunden pro Megabyte. Angenommen, es steht nicht genug Arbeitsspeicher für beide Operatoren zur Verfügung, dann ist es in dem Bereich, in dem die Operatoren beide nicht genug Arbeitsspeicher zur Verfügung haben, am vorteilhaftesten zunächst nur dem Operator B mehr Arbeitsspeicher zu gewähren.

1.1 Ziele dieser Arbeit

SQL wurde als Abfragesprache für das relationale Datenmodell konstruiert. Dabei wurden verschiedene Erweiterungen vorgeschlagen, damit SQL mit geschachtelten Relationen arbeiten kann. Der erste Teil (Kapitel 3 bis 6) dieser Arbeit besteht darin, den im Datenbanksystem Secondo implementierten SQL-Dialekt um die bisher nicht vorhandene Möglichkeit zu erweitern, auch mit geschachtelten Relationen umgehen zu können.

Der zweite Teil (Kapitel 7 bis 12) beschäftigt sich damit, wie eine optimale Verteilung des Arbeitsspeichers für einen Ausführungsplan erreicht werden kann, um die Gesamtkosten zu minimieren. Bisher wurde hier von einem konkreten Ausführungsplan ausgegangen, der bezüglich eines Kostenmaßes der günstigste ist. In der Phase der Ausführungsplanerzeugung wurde also noch nicht die Ressource Arbeitsspeicher berücksichtigt. Daher wird ebenfalls untersucht, ob und welche Möglichkeiten bestehen, unter allen möglichen Ausführungsplänen denjenigen ermitteln zu können, der auch unter dem Gesichtspunkt der optimalen Arbeitsspeicherverteilung der günstigste Ausführungsplan ist.

1.2 Aufbau der Arbeit

Das zweite Kapitel behandelt die wichtigsten Grundlagen. Im Anschluss gliedert sich die Arbeit in die folgenden zwei voneinander unabhängigen Teile:

- Teil I (Kapitel 3 bis 6): Nachdem im Kapitel 3 der aktuelle Stand der Technik dargestellt worden ist, wird anschließend die *Behandlung der geschachtelten Relationen* genauer thematisiert. Im Kapitel *Entwurf* wird ein Lösungsansatz entworfen und es werden viele Beispiele zu den implementierten Möglichkeiten, um mit geschachtelten Relationen arbeiten zu können, aufgezeigt. Im Kapitel *Implementierung* werden die wichtigsten Punkte beschrieben, die am Optimierer geändert oder hinzugefügt wurden. Dieser Teil schließt ab mit den Ergebnissen, möglichen Erweiterungen und dem Fazit zu diesem Themenbereich. Das Fazit befindet sich am Ende dieses Teils, um es nicht aus dem Kontext zu reißen.
- Teil II (Kapitel 7 bis 12): Dieser Teil befasst sich mit dem Thema der *Arbeitsspeicherzuweisung an Operatoren*. Einleitend wird auf den aktuellen Stand der Technik eingegangen. Fortführend wird im Kapitel *Entwurf* das Problem analysiert und mögliche Strategien zur Lösung werden dargestellt. Im Kapitel *Implementierung* werden die Änderungen des Optimierers beschrieben. Im Anschluss werden im Kapitel *Evaluierung* die verschiedenen Strategien auf Basis von ausgewählten Experimenten analysiert und bewertet. Abschließend werden mögliche Erweiterungen dargelegt und ein Fazit gezogen.

Teilweise werden zu den deutschen Bezeichnungen auch die englischen Begriffe angegeben, um sich bei Bedarf besser in dem Quellcode orientieren zu können, denn die Bezeichnungen und Kommentare sind dort, wie im Secondo Quellcode üblich, in englischer Sprache verfasst.

Kapitel 2

Grundlagen

2.1 Datenbankmanagementsysteme

Ein Datenbankmanagementsystem ist im Kern, wie J.C. Date es in [Dat99, Seite 5] ausdrückt, ein computergestütztes System, dessen Gesamtziel es ist, Daten zu speichern, zu modifizieren und das seinem Anwender erlaubt, die Daten wieder auf Anfrage zur Verfügung zu stellen. Um dies zu leisten, wird vom DBMS in der Regel eine Abfragesprache wie SQL bereitgestellt. Das DBMS ist also ein Softwareprogramm. Es wird in seiner Begrifflichkeit noch abgegrenzt zur eigentlichen *Datenbank*, in der die Daten physisch und insbesondere persistent gespeichert sind. Unter einem Datenbanksystem (DBS) wird ein DBMS in Verbindung mit einer Datenbank verstanden. Auch wenn ein DBMS in seiner Hauptfunktion damit schnell skizziert werden kann, ist es doch ein komplexes Softwaresystem, das seinen Anwendern eine Vielzahl von Funktionen zur Verfügung stellt.

Die Sichtweise auf die Architektur eines DBMS kann in drei Ebenen aufgeteilt werden [Dat99]:

- Die externe Ebene: Hier wird in der Regel nur ein Teil der Datenbank beschrieben und zwar stellt der Ausschnitt der beschriebenen Daten nur den für einen Benutzer relevanten Teil dar. Es entspricht damit der Sichtweise eines Benutzers auf die Datenbank.
- Die konzeptuelle Ebene: Dies ist die logische Gesamtstruktur der Datenbank, in der demzufolge alle speziellen Sichtweisen der verschiedenen Benutzer enthalten sind. Insbesondere findet auf dieser Ebene die Modellierung der Situation aus der realen Welt in eine Datenbankstruktur statt.
- Die interne Ebene: Hier wird im Detail festgelegt, wie die Datenbank auf dem physischen Datenträgern organisiert ist.

Diese verschiedenen Sichtweisen erlauben jeweils eine gewisse Abstraktion von der tiefer liegenden Ebene. Diese Abstraktionen sind dabei eine herausragende Eigenschaft der DBMS. Die Benutzer der konzeptuellen Ebene werden größtenteils davon befreit, die Organisation der internen Ebene zu kennen. In einem DBS ist einzig und allein das DBMS für alle Operationen auf der Datenbank zuständig. Es gibt keine anderen Zugriffe auf die Datenbank als über das DMBS.

Die für diese Arbeit wichtigen Komponenten sind dabei die Abfragesprache SQL, die auf der externen und konzeptuellen Ebene operiert, und der Anfrageoptimierer. Auf den Anfrageoptimierer wird im Zusammenhang mit dem Secondo DBMS eingegangen.

2.2 Das relationale Datenmodell

Bisher wurde zwar immer von „den Daten“ gesprochen, aber erst ein konkretes Datenmodell legt fest, mit welchen Mitteln die Daten in ihrer Struktur beschrieben werden und auch welche Operationen auf den Daten ausgeführt werden können. Das relationel Modell wurde von Edgar F. Codd in [Cod70] vorgeschlagen und besitzt im Gegensatz zu den meisten anderen Modellen eine formale mathematische Basis. Es ist das am meisten verbreitete Datenmodell [Dat99]. Ein Grund, der dazu führte, dass das relationale Modell so erfolgreich wurde, ist vermutlich auch der Tatsache zuzurechnen, wie dieses Datenmodell repräsentiert werden kann. Relationen können als Tabellen dargestellt werden und Tabellen sind allgemein eine verständliche Struktur. Die Zeilen sind dabei die *Datensätze* und entsprechen den *Tupeln* in Relationen, während *Spalten* bei den Relationen die *Attribute* repräsentieren. Attribute besitzen dabei einen *Wertebereich* (*Domäne*). Die Werte in den Tupeln dürfen also nur aus der jeweiligen Domäne stammen. Bei Tabellen wird von einem Datentyp gesprochen. Allerdings sind Relationen und Tabellen im Sinne des Datenmodells nicht identisch. Relationen sind mathematisch beschrieben und beinhalten beispielsweise kein Tupel zweimal. Jedes Tupel ist also im Gegensatz zu den Datensätzen in einer Tabelle eindeutig. Die meisten DBMS arbeiten im Sinne von Tabellen, so dass identische Datensätze durchaus öfters vorkommen können.

Es sollen an dieser Stelle auch noch zwei der wichtigsten Operatoren der relationalen Algebra nach [Cod70], sowie weitere Operatoren des Secondo DBMS, die in den folgenden Kapiteln in den Ausführungsplänen wieder auftauchen, kurz beschrieben werden:

- Selektion: Es werden nur Tupel aus einer Relation ausgewählt, die eine bestimme Bedingung erfüllen. Wird die Beispielrelation Tabelle 1.1 auf Seite 1 zugrunde gelegt, könnte eine Selektion lauten: „Wähle alle Datensätze aus, die mehr als 100 000 Einwohner besitzen“. Im Secondo DBMS ist dieser Operator durch den Operator *filter* implementiert.

- Projektion: Aus der Eingaberelation werden nur bestimmte Attribute in Ausgaberelation übernommen. Beispiel: „Wähle aus der Relation *Orte* (Tabelle 1.1 auf Seite 1) nur das Attribut *Kennzeichen* aus.“. Dieser Operator wird im Secondo DMBS durch den Operator *project* realisiert.

- *remove*: Entfernt ein Attribut aus dem Tupelstrom.

- *extendstream*: Fügt dem Tupelstrom auf Basis eines Ausdrucks ein neues Attribut hinzu.

- *renameattr*: Ändert die Bezeichnung für ein Attribut.

- Symmetrischer Join (*symmjoin*): Dieser Operator verbindet zwei Relationen, indem das kartesische Produkt gebildet wird und als Ergebnis nur Tupel ausgegeben werden, die eine Bedingung erfüllen.

- *mergejoin*: Dieser Operator verbindet ebenfalls zwei Relationen anhand von zwei Attributen. Allerdings erwartert dieser, dass die erste Relation nach dem Attribut sortiert aus der Relation übergeben wird. Gegenüber des symmjoin kann dieser Operator wesentlich effizienter arbeiten.

- *sortmergejoin*: Der Operator arbeitet wie der mergejoin. Allerdings wird nicht erwartet, dass die erste Relation sortiert ist.

- *gracehashjoin, hybridhashjoin, itHashJoin, itSpatialJoin*: Diese Join Operatoren verbinden auch zwei Relationen wie die vorherigen Join Operatoren und unterscheiden sich nur darin, wie diese intern arbeiten.

Konkrete Details zu diesen Operatoren können beispielsweise [SUSER11, Seite 37] entnommen werden. Bei den Operatoren wurde hier teilweise davon gesprochen, als wären die Parameter der Operatoren die Relationen selbst. Dieses diente aber mehr der informalen Beschreibung, wie diese Operatoren arbeiten. Es sei aber darauf hingewiesen, dass viele DBMS Implementierungen so nicht arbeiten, denn es wird nicht immer erst eine Relation als Zwischenergebnis berechnet und anschließend der nächste Operator auf diesem Zwischenergebnis arbeiten. Dadurch würden zu große und vor allem unnötige Relationen als Zwischenergebnisse auftreten. Daher arbeiten die meisten DBMS, wie auch das Secondo DBMS, auf der ausführenden Ebene mit Strömen von Tupeln.

Genaueres zu dem relationalen Datenmodell kann beispielsweise [Cod70] oder auch [Dat99] entnommen werden.

2.3 Secondo

Das Secondo DBMS wurde am Lehrgebiet Datenbanksysteme für neue Anwendungen der FernUniversität in Hagen entwickelt. Zum einem zeichnet es sich dadurch aus, mit bewegten Objekten arbeiten zu können, zum anderem wurde besonderen Wert darauf gelegt, erweiterbar zu sein. Einen wesentlichen Beitrag zu dieser Erweiterbarkeit leistet das *Algebra Modul System*, welches erlaubt, zusätzliche Algebren zu implementieren. Eine Algebra implementiert dabei ein oder mehrere Datentypen und/oder Operatoren, die in den Ausführungsplänen (der Secondo-Textsyntax) verwendet werden können. Das Secondo DBMS ist auch nicht fest an ein Datenmodell gebunden. Es besteht die Möglichkeit, verschiedene Datenmodelle zu implementieren [SUSER11]. Ein Beispiel hierfür ist die *NestedRelation* Algebra. Diese erweitert gerade das im Secondo enthaltene relationale Modell um die geschachtelten Relationen.

Abbildung Abbildung 2.1 auf der nächsten Seite[1] zeigt einen Überblick über die verschiedenen Komponenten im Secondo DBMS. Anwender können auf das Secondo DMBS über eine graphische Oberfläche oder den Kommandozeileninterpretern zugreifen. Ist der Optimierer aktiv, können dabei auch SQL-Statements verwendet werden, die durch den Optimierer in einen entsprechenden Ausführungsplan übersetzt werden. Der Secondo Kernel mit seinen verschiedenen Komponenten ist für die Verarbeitung der Anweisungen in der Secondo-Textsyntax verantwortlich. Dies ist natürlich nur eine sehr grobe Beschreibung. Für tiefer gehende Informationen sei auf z.B. [Gü+04] oder [SDEV11] verwiesen.

2.3.1 Der Secondo Anfrageoptimierer

Der Secondo Anfrageoptimierer, im Folgenden nur noch als Optimierer bezeichnet, ist eine separate Komponente, die nicht zwingend verfügbar sein muss, um mit Secondo arbeiten zu können. Implementiert ist der Optimierer in der logischen Programmiersprache Prolog[2] und beschränkt sich im Kern auf die Optimierung von Anfragen auf dem relationalen Modell [Gü, Seite 3]. Die Aufgabe

[1] In Anlehnung an [SUSER11, Seite 2]

[2] Verwendet wird der SWI-Prolog-Interpreter: `http://www.swi-prolog.org`

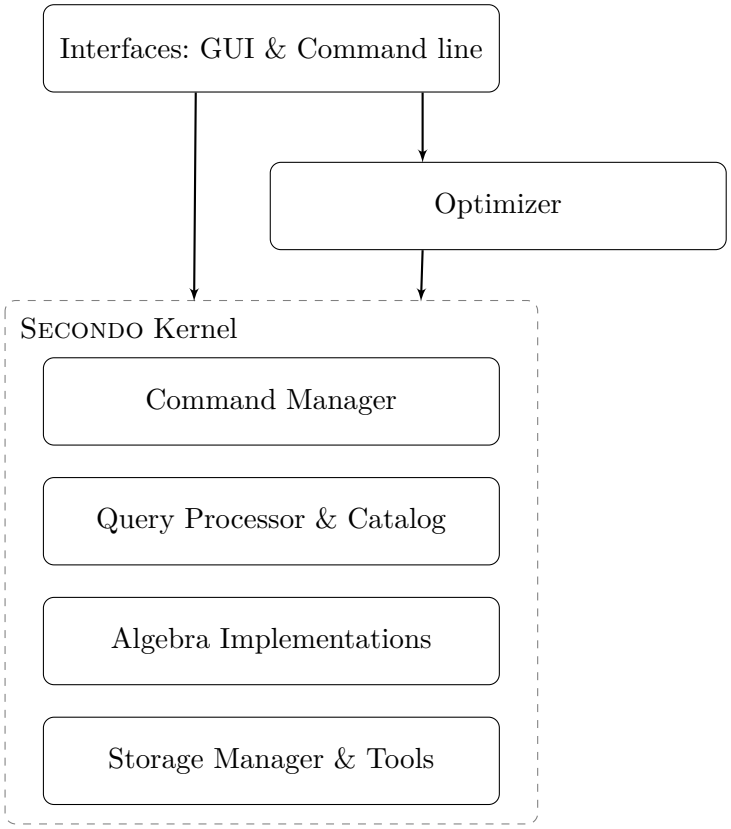

Abbildung 2.1: Grobe Architekturübersicht des SECONDO DBMS

des Optimierers ist, wie im Kapitel 1 schon aufgriffen, das Erzeugen eines effizienten Ausführungsplans.

Die Arbeitsweise des SECONDO Optimierers kann dabei in folgende Phasen aufgeteilt werden:

1. *Rewriting*: Diese erste Phase bekommt als Eingabe eine SQL-Anweisung in der Form, in der dieses Statement vom Anwender (oder eines Programmes) an den Optimierer übergeben wurde. Auf dieser Basis wird ein Umschreiben der Anfrage durchgeführt. Ein Beispiel für eine solche Umformung sind Makros. Bereits hier sind schon Optimierungsvorgänge möglich.

2. *Lookup*: In dieser Phase wird die Anfrage so umgeschrieben, dass die folgende Verarbeitung der Anfrage vereinfacht wird. Dies wird dadurch erreicht, dass die Anfrage in ein einheitliches Format gebracht wird. Auch werden zu diesem Zweck der Anfrage weitere Informationen hinzugefügt. Eine weitere, wichtige Funktion der Phase ist aber auch die Erkennung von Fehlern in der Anfrage. So wird beispielsweise überprüft, ob verwendete Attribute überhaupt existieren.

3. *Auswertungsplanerzeugung*: In dieser Hauptphase der Optimierung wird zum einem der Ausführungsplan selber erzeugt, aber auch die eigentliche Optimierung der Anfrage durchgeführt. Dazu verwendet der Optimierer ein Kostenmodell, um die verschiedenen möglichen Alternativen an Ausführungsplänen bezüglich ihrer Kosten zu bewerten. Um die Kosten aber über-

haupt schätzen zu können, verwaltet der Optimierer eigene Statistiken wie Selektivitäten und Ausführungszeiten zu WHERE-Prädikaten. Die Selektivität eines Wertes aus einem Attribut einer Relation gibt dabei an, wie oft dieser Wert in dem Attribut vorkommt. Legt man nochmal die Beispieltabelle 1.1 zugrunde, besitzt der Wert „KLE“ in der Spalte *Kennzeichen* die Selektivität $1/6 = 0,1\overline{6}$, da nur ein Tupel den Wert „KLE“ von insgesamt sechs Tupeln besitzt.

Im Gegensatz zu anderen Optimierern ist hier eine Phase aber nicht offensichtlich erkennbar. In der Regel liegt eine Anfrage anfangs immer nur als Zeichenkette vor. Diese muss als erstes dazu einmal durch Techniken des Übersetzerbaus in eine Form gebracht werden, die den weiteren Programmteilen einfacher zugänglich ist. Diese Phase fehlt hier auch nicht, allerdings wird dieser Schritt vom Optimierer dem Prolog-Interpreter überlassen. Dazu muss natürlich eine solche Anfrage insbesondere einen syntaktisch gültigen Prolog-Term[3] darstellen.

Im Folgenden soll noch darauf eingangen werden, mit welcher Methodik der SECONDO-Optimierer den aus seiner Sicht besten Ausführungsplan ermittelt. Mit „aus seiner Sicht“ ist dabei der Problematik geschuldet, dass der Anfrageoptimierer eigentlich nicht in der Lage ist zu garantieren den Plan mit den günstigsten Kosten zu finden. Die Hauptursache dafür ist der Mangel an Informationen. So besitzt ein Optimierer nie alle Selektiväten, da zum einen die Berechnung dafür zu aufwändig wäre und zum anderen die dann zustande kommende Datenmenge keine effiziente Optimierung erlauben würde. Zusätzlich ist der Suchraum (die Menge an möglichen Ausführungsplänen) für komplexere Anfragen zu groß, um diesen komplett abzusuchen. Dies soll jedoch an dieser Stelle nicht weiter vertieft werden, da das Thema wird im zweiten Teil dieser Arbeit aufgegriffen wird. Einzelheiten zu der Problematik werden in [Cha98] oder [Dat99] beschrieben.

Der *predicate order graph (POG)*

Anfragen, die Bedingungen in der WHERE-Klausel beinhalten, werden anders optimiert als Anfragen ohne Bedingungen. Anfragen ohne Bedingen werden nach festen Regeln in Ausführungspläne übersetzt, ohne deren Kosten zu betrachten. Im Folgenden sei n die Anzahl der Prädikate in der WHERE-Klausel. Zur Konstruktion des POG werden alle n WHERE-Bedingungen permutiert, also jede mögliche Reihenfolge betrachtet, in der die Bedingungen ausgewertet werden können. Aufgrund dieser möglichen Reihenfolgen wird ein gerichteter, azyklischer Graph mit $n \cdot 2^{n-1}$ Kanten und 2^n Knoten konstruiert, der *predicate order graph (POG)*. Der *predicate order graph* speichert mit den Knoten ab, welche Prädikate bisher ausgewertet worden sind, während die Kanten mit der Information versehen werden, welche Prädikate bereits ausgewertet sind. Im nächsten Schritt werden dann den Kanten alle möglichen[4] Anweisungsfolgen der ausführenden Ebene (der Algebren) zugeordnet. Als letzten vorbereitenden Schritt werden zu diesen Anweisungsfolgen deren Kosten berechnet und zugewiesen. Die eigentliche Optimierung erfolgt nun mit Hilfe eines Algorithmus zur Berechnung des kürzesten Weges (shortest path). Der Startknoten ist der Knoten, an dem noch keine Prädikate ausgewertet wurden. Der Zielknoten ist der Knoten, an dem alle Prädikate ausgewertet wurden. Der kürzeste Weg im *predicate order graph* vom Startkonten zum Zielknoten entspricht nun dem günstigsten Ausführungsplan. Durch die Verwendung des *predicate order graph* wurde also die Ermittlung des günstigsten Ausführungsplans auf die Ermittlung des kürzestens Weges im POG reduziert. Der Optimierer verwendet zur Ermittlung des kürzesten Weges den Algorithmus von Dijkstra

[3] Terme sind in Prolog die grundlegenden Datenstrukturen [SS94].
[4] In Sinne von allen möglichen im Optimierer bekannten Regeln.

[Dij59] in der Variante mit Fibonacci-Heap, die eine bessere Komplexitätsschranke vorweist. Hier ist noch ein entscheidendes Detail zu beachten: Die Ermittlung des kürzesten Pfades findet nicht direkt auf dem *predicate order graph* selbst statt. Genau genommen wird der Algorithmus auf dem Graphen angewendet, der implizit durch die zugewiesenen Kostenkanten im *predicate order graph* definiert ist. In einem ersten Schritt werden dazu den Kanten im POG mögliche Alternativen der ausführenden Ebene zugeordnet. In einem nächsten Schritt werden dann für diese Kanten die Kosten berechnet und als Kostenkanten abgespeichert. Im Folgenden wird dieser so definierte Graph auch als *Graph der Kostenkanten auf Basis des POG* bezeichnet. Dieser Graph der Kostenkanten kann wesentlich mehr als die $n!$ Pfade und auch mehr als $n \cdot 2^{n-1}$ Kanten enthalten. Alle Pfade vom Startknoten zum Endknoten im POG stellen damit alle möglichen Reihenfolgen dar, in der die Prädikate ausgewertet werden können. Alle möglichen Pfade des Graphen, der durch die Kostenkanten beschrieben ist, repräsentieren hingegen alle möglichen Ausführungspläne. In [Gü+04] wird der Algorithmus zur Konstruktion eines *predicate order graph* anhand eines Beispiels konkret vorgeführt.

2.3.2 Structured Query Language (SQL)

SQL ist die Standard Anfragesprache für das relationale Datenmodell. Obwohl es einen SQL-Standard in verschiedenen Versionen gibt, unterscheiden sich die Implementierungen vieler Hersteller zu diesem Standard. Dabei unterscheiden sich diese nicht nur in ihrer Grammatik. In vielen DBMS ist auch nur ein Teil des recht umfangreichen SQL-Standards implementiert [Arv12]. Vielfach werden aber auch Funktionen bereitgestellt, die nicht Bestandteil des SQL-Standards sind. Beispielsweise unterstützt das Secondo DBMS Datentypen für bewegte Objekte[5]. Auch der in Secondo implementierte SQL-Dialekt bildet hiervon keine Ausnahme und unterscheidet sich in vielen Punkten vom SQL-Standard.

Ein wenig näher betrachtet werden soll der Bereich der Abfragesprache. Dabei wird auch gleich die Syntax des Secondo SQL-Dialektes verwendet. Damit sollten Leser, die mit SQL bereits vertraut sind, keine größeren Probleme haben, die hier in der Arbeit verwendeten SQL-Statements verstehen zu können.

Eine Besonderheit gilt es immer bei den Secondo SQL-Statements zu beachten: Außer Textkonstanten sollte alles mit einem Kleinbuchstaben beginnen. Die Ursache dafür ist, dass Terme, die mit einem Großbuchstaben anfangen, in Prolog Variablen sind. Wenn ein Spaltenname oder ein Tabellenname in der Secondo Datenbank mit einem Großbuchstaben beginnt, können diese natürlich trotzdem verwendet werden. Dazu reicht es den ersten Buchstaben in einen Kleinbuchstaben umzuschreiben oder alternativ kann auch der ganze Name kleingeschrieben werden. In dieser Arbeit werden einheitlich nur Kleinbuchstaben in den SQL-Statements verwendet werden, ohne darauf immer explizit hinzuweisen. Das wohl wichtigste SQL-Statement ist ein SELECT-FROM-WHERE (SFW) Ausdruck wie im Beispiel 2.1.

In der FROM-Klausel wird die Datenbasis angegeben. Hier werden die Tabellen[6] angegeben, die der Anfrage zugrunde liegen.

Die SELECT-Klausel erlaubt das Angeben der Spalten der Ergebnistabelle des SQL-Statements. Entweder werden hier die Attribute direkt angegeben oder der Stern „*“ kann verwendet werden,

[5] Was aber nicht gleichbedeutend damit ist, dass Anwendungen für bewegte Objekte nicht mit relationalen Datenbanken realisiert werden können

[6] Im SQL-Jargon wird wiederum von Tabellen anstatt von Relationen usw. gesprochen [Vgl. 2.2].

Listing 2.1: SFW Beispiel

```
1 select *
2 from   orteh
3 where  [bevt >= 100, kennzeichen="B"]
```

um alle Attribute auszuwählen, die in den Tabellen der FROM-Klausel vorhanden sind. Möglich ist auch das Erstellen neuer Spalten durch z.B. Formeln auf Basis anderer Spalten.

Die WHERE-Klausel erlaubt es zum einen Bedingungen anzugeben, welche Datensätze der FROM-Klausel in die Ergebnistabelle übernommen werden, zum anderen können hier auch die Tabellenverknüpfungen spezifiziert werden, wenn mehrere Tabellen in der FROM-Klausel angegeben wurden. In der Beispielanfrage 2.1 werden durch die WHERE-Klausel-Bedingung, nur Datensätze ausgewählt, in denen die Bevölkerung in Tausend einen Wert größer oder gleich 100 aufweist und das Attribut *Kennzeichen* den Wert „B“ enthält.

Die eckigen Klammern „[“ und „]“ sind dabei im Secondo DBMS eine Besonderheit. Das liegt, wie bereits erwähnt, an der Verwendung von Prolog-Termen zum Definieren von SQL-Statements. Sobald für die Klauseln SELECT, FROM, WHERE, GROUPBY oder ORDERBY mehr als ein Element angegeben wird, müssen die Elemente in einer Prolog-Liste übergeben werden. Dazu werden die Elemete, durch Kommata getrennt, in den eckigen Klammern eingefasst. Mehrere Bedingungen in der WHERE-Klausel werden im Standard SQL mit den Schlüsselworten AND bzw. OR verknüft. In der WHERE-Klausel des Secondo SQL-Dialekts werden aber alle dort angegebenen Bedingungen als AND Bedingungen behandelt. OR-Bedingungen sind bisher nicht implementiert.

Aggregatsfunktionen können mit der GROUPBY-Klausel angewendet werden. Hier werden Funktionen gruppenweise angewendet, d.h. für alle Datensätze einer Gruppe wird die Funktion aufgerufen, aber nur einmal in die Ergebnisstabelle übernommen. Die Verwendung des in dem Zusammenhang im SQL-Standard möglichen HAVING ist hingegen ausgeschlossen, da dies nicht implementiert ist.

Folgende Aggregatsfunktionen sind im Secondo System verfügbar:

- count: Berechnet die Anzahl der Datensätze
- avg: Arithmetisches Mittel berechen
- max: Das Maximum ermitteln
- min: Das Minimum ermitteln
- sum: Die Summe bilden
- extract: Liefert nur den Wert eines Attributs des ersten Datensatzes zurück

Das DBMS Secondo erlaubt allerdings auch das Spezifizieren neuer Aggregatsfunktionen, wenn diese bestimmten Voraussetzungen genügen. Hierauf soll aber nicht weiter eingegangen und auf [SOM12] verwiesen werden.

Die ORDERBY-Klausel kann verwendet werden, um die Ergebnistabelle nach einer Reihe von Attributen zu sortieren. Dabei kann mit den Schlüsselwörtern *asc* oder *desc* angegeben werden, ob das Attribut aufsteigend oder absteigend sortiert werden soll.

Als letztes kann dem Statement ein *FIRST n* oder *LAST n* angehängt werden, wenn nur die ersten oder letzten n Datensätze in die Ergebnistabelle einfließen sollen.

Möglich ist auch das Vergeben neuer Namen für Attribute sowie das Bekanntmachen von Relationen unter einer Variablen, oft auch als Alias oder Label bezeichnet. Da die Methodik hier etwas anders als bei anderen DBMS ist, soll dies noch kurz erklärt werden. Das Vergeben neuer Spaltennamen funktioniert dabei wie bei anderen DBMS auch. Hierbei wird in der SELECT-Klausel dem Attribut ein „as Label“ angehängt, wobei „Label“ der neue Name ist. Für Relationen der FROM-Klausel funktioniert dieses in derselben Weise. Der erste Unterschied ist, dass, wenn Attribute dieser Relation verwendet werden, dieser Variablenname auch verwendet werden muss. Dies funktioniert dann, indem der Variablename mit einem Doppelpunkt dem Attributnamen voran gestellt wird. Dass hier der Doppelpunkt anstatt des Punktes wie sonst üblich verwendet wird, hat die Ursache, dass der Punkt in Prolog schon als Operator seine eigene Bedeutung besitzt. Beispiel 2.2 zeigt exemplarisch die Anwendung beider Fälle.

Listing 2.2: Beispiel: Umbenennungen

```
1 select [o:ort, o:bevt*1000 as einwohner]
2 from   orte as o
```

Im Secondo SQL-Dialekt führt die Verwendung einer Variablen dazu, dass ein Unterstrich gefolgt von dem Variablenname allen Attributen der Relation angehängt wird. Das Ergebnis der Anfrage aus Beispiel 2.2 wäre daher die Tabelle 2.1.

Tabelle 2.1: Ergebnis zur Anfrage aus Beispiel 2.2

Ort_o	Einwoher
Berlin	3 410 000
Hagen	208 000
Bonn	291 000
Gelsenkirchen	288 000
Kleve	46 000
Wesel	6200

Die vollständige Grammatik des Secondo SQL-Dialekts, inklusive der Änderungen dieser Arbeit, befindet sich im Anhang A.

2.3.3 Die Erweiterung des Secondo Optimierers um geschachtelte Anfragen

Diese Erweiterung wurde als Teil einer Abschlussarbeit [Pon12] implementiert und erweitert den Secondo SQL-Dialekt um die Möglichkeit geschachtelte Anfragen zu formulieren. Geschachtelte Anfragen erlauben es, in einer Anfrage der Form *select ... from ...* weitere Anfragen (Subqueries) in der SELECT-, FROM- oder WHERE-Klausel zu verwenden. Die implementierte Erweiterung in Secondo legt aber den Fokus auf die Verwendung von geschachtelten Anfragen in den WHERE-

Prädikaten. 2.3 zeigt ein Beispiel einer geschachtelten Anfrage. Die genaue Grammatik kann auch dem Anhang A oder [Pon12] entnommen werden.

Listing 2.3: Geschachtelte Anfrage

```
1 select kennzeichen
2 from   orte
3 where  exists(select *
4               from   orte as o
5               where  kennzeichen=o:kennzeichen)
```

Die Optimierung dieser geschachtelten Anfragen läuft dabei so ab, dass hier nicht eine Anfrage inklusive aller Subqueries in einem Schritt optimiert wird, sondern es wird dabei immer nur jeweils eine Anfrage ohne die Einbeziehung der geschachtelten Anfragen optimiert. Hier wird zuerst, falls eine WHERE-Bedingung vorhanden ist, die Anfrage der obersten Ebene optimiert. Nachdem nun ggf. die Optimierung durchgeführt wurde, wird der Ausführungsplan der Anfrage erzeugt. Die letzte Phase ist die Erzeugung des Ausführungsplans in der SECONDO-Textsyntax. Erst hier wird rekursiv in die Subqueries verzweigt und, soweit WHERE-Bedingungen involviert sind, die Optimierung aller geschachtelten Anfragen rekursiv durchgeführt.

2.3.4 Die NestedRelation Algebra des SECONDO DBMS

Im SECONDO DBMS sind die geschachtelten Relationen durch die dort implementierte *NestedRelationAlgebra* realisiert, die in [Teu12] beschrieben wird. Dort werden zwei neue Typen definiert. Relationen, die Relationen als Attribute enthalten dürfen, sind vom Typ *nrel* und Attribute, die eine Relation darstellen, sind vom Typ *arel*. An ein paar Stellen im weiteren Text ist in der Begrifflichkeit abzugrenzen zwischen Relation vom Typ *nrel* und geschachtelten Relationen. Eine *nrel* Relation kann sich auch in erster Normalform befinden, nämlich genau dann, wenn in der Relation keine Subrelation existiert. Um auf diesen Fall nicht immer explizit hinweisen zu müssen, wird von einer Relation des Typs *nrel* gesprochen.

Die Zusammenarbeit mit den anderen SECONDO Operatoren wird dadurch erreicht, dass die meisten Operatoren mit Strömen von Tupeln arbeiten. Daher arbeiten viele Operatoren ohne weiteres Zutun auch mit den Relationen vom Typ *nrel* und *arel*. Für das Erzeugen der Ströme von Tupeln aus *arel*-Relationen wird der *afeed* Operator verwendet und mit dem *aconsume* Operator kann ein Attribut vom Typ *arel* aus einem Tupelstrom erzeugt werden. Ausdrücklich erwähnt werden soll hier noch die Tatsache, dass die Operatoren in der Regel immer nur auf der obersten Ebene wirken. Eine *unnest* Operation kann beispielsweise keine zwei Ebenen tiefer liegende geschachtelte Relation auflösen. Eine Ausnahme hiervon stellt der *rename* Operator da. Dieser läuft rekursiv ab und ändert damit auch alle Attributbezeichnungen in allen Attributen, die vom Typ *arel* sind.

2.3.4.1 Der *unnest*-Operator

Das Entschachteln einer Subrelation wird durch den *unnest* Operator ermöglicht. Dabei bekommt der *unnest*-Operator als Parameter den Namen der Subrelation und verknüft jedes Tupel der Subrelation mit dem Tupel, in dem das Attribut enthalten ist. Beachtenswert ist dabei, dass die *unnest* Operation zwar invers zu der *nest* Operation ist, aber der *nest* Operator nicht in jedem Fall invers zum *unnest* Operator ist [SP82].

Wird der *unnest* Operator auf Basis der Relation *OrteH* (Tabelle 1.2 auf Seite 2) und dem Attribut *BevT* angewendet, ergibt sich die Tabelle 2.2.

Tabelle 2.2: Anwendung des *unnest* Operators

BevH	Kennzeichen	Ort	Vorwahl	BevT
34	B	Berlin	030	3410
2	HA	Hagen	02331	208
2	BN	Bonn	0228	291
2	GE	Gelsenkirchen	0209	288
0	KLE	Kleve	02821	46
0	WES	Wesel	0281	62

2.3.4.2 Der nest-Operator

Mit Hilfe des Operators *nest* wird die eine Relation in erster Normalform oder aber auch eine bereits geschachtelte Relation weiter geschachtelt. Der *nest*-Operator funktioniert dabei folgendermaßen: Als Parameter bekommt dieser eine Liste von Attributen und den Namen für ein neues Attribut, das die neu gebildete Relation nach der Operation beinhaltet. Für die Liste von Attributen werden alle eindeutigen Kombinationen gebildet und für jede dieser Kombinationen wird ein Tupel in der Ausgabe erstellt. Für alle nicht in der Parameterliste von Attributen aufgeführten Attribute werden die zugehörigen Tupel der eindeutigen Kombinationen in die geschachtelte Relation übernommen.

Die Implementierung des *nest* Operators hat eine Besonderheit, die von anderen Implementierungen abweicht. So muss der Eingabestrom von Tupeln nach den Attributen, anhand derer das *nest* durchgeführt wird, bereits sortiert sein.

Wird der *nest* Operator auf Basis der Tabelle 2.2 angewendet, ergibt sich wieder die Tabelle 1.2 auf Seite 2. Angewendet wird dazu der Operator auf dem Attribut *BevH* und als Name der neuen Subrelation ist *SubRel* anzugeben.

2.4 Nichtlineare Optimierung

Viele Optimierungsaufgaben sind derart, dass eine Funktion $f : S \to \mathbb{R}$ bezüglich dessen Funktionswertes über die Wahl der Parameter minimiert oder maximiert werden soll. Die Funktion, die dann minimiert (oder maximiert) werden soll, ist die Zielfunktion (objective function). Die Menge $S \subseteq \mathbb{R}^n$ beschreibt den zulässigen Bereich, in dem mögliche Lösungen liegen können. Die Menge S kann dabei auch weiteren Einschränkungen unterliegen, die durch Funktionen, den Restriktionen (constraints), beschrieben werden können. Von linearer Optimierung wird gesprochen, falls die Zielfunktion sowie die Restriktionen linear sind, andernfalls handelt es sich um die nichtlineare Optimierung. Die nichtlineare Optimierung gehört sogar immer noch zu den nicht vollständig gelösten Problemen der Mathematik. Es muss sich hier teilweise immer noch mit Spezialfällen benügt werden [MG00]. Insbesondere existiert für die nichtlineare Optimierung kein allgemeines Verfahren wie der Simplexalgorithmus aus der linearen Optimierung.

Die Darstellung eines nichtlinearen Optimierungsproblems im Falle eines Minimierungsproblems erfolgt oft in der folgenden Art:

$$\begin{aligned} &\min_{x \in S} f(x) \\ &unter \\ &fc_i(x) \leq 0,\ fc_i : S \to \mathbb{R},\ 1 \leq i \leq m \end{aligned} \tag{2.1}$$

Die fc_i stellen hier die Restriktionen in Form von Ungleichungen dar.

2.5 NLopt

NLopt[7] ist eine Open Source[8] Bibliothek, implementiert in der Sprache C, zur nichtlinearen Optimierung welche am Massachusetts Institute of Technology (MIT) entwickelt wurde. Dabei verhält sich NLopt als Schnittstelle, um verschiedene Algorithmen zur nichtlinearen Optimierung verwenden zu können. Die dort implementierten Algorithmen lassen sich in drei Kategorien einteilen:

- Globale Optimierung
- Lokale ableitungsfreie Optimierung
- Lokale gradientenbasierte Optimierung

Für diese Arbeit ist der Algorithmus *Method of Moving Asymptotes (MMA)* relevant, der in [Sva02] beschrieben ist. Dieser Algorithmus gehört zur Kategorie der lokalen gradientenbasierten Verfahren und unterstützt insbesondere eine beliebige Anzahl von Ungleichsrestriktionen der Form $g(x) \leq 0$.

[7] `http://ab-initio.mit.edu/wiki/index.php/NLopt`
[8] Lizenz: GNU LGPL

Teil I

Erweiterung des Secondo Optimierers: Behandlung geschachtelter Relationen

Kapitel 3

Stand der Technik

Geschachtelte Relationen wurden als erstes von A. Makinouchi [Mak77] vorgestellt. Er erkannte schon damals den Trend zu komplexeren Anwendungen, welche Datenstrukturen benötigen, die nicht zwingend in der ersten Normalform sind.

Die Operatoren *nest* und *unnest* wurden in [SP82] eingeführt, um aus einer geschachtelten Relation wieder eine Relationen in der ersten Normalform zu erhalten bzw. um Relationen, die sich in erster Normalform befinden, in eine geschachtelte Relation zu transformieren. Im Unterabschnitt 2.3.4 auf Seite 13 wurde bereits kurz die Arbeitsweise der beiden Operatoren, nämlich wie diese in der entsprechenden Algebra im SECONDO DBMS implementiert sind, informal beschrieben. Für eine etwas davon abweichende, aber formale Definition sei insbesondere auf [SP82] verwiesen.

In [KR89], [RKB87], [KHQ98] oder auch [PA86] werden verschiedene Vorschläge gemacht, wie die Integration in eine SQL ähnliche Abfragesprache aussehen könnte. Dabei unterscheiden sich die verschiedenen Vorschläge teilweise voneinander, allerdings eher in syntaktischer Hinsicht als in ihrer Mächtigkeit. Eine Ausnahme davon stellt [SP82] dar. Hier wird für die Operatoren *nest* und *unnest* nur die Anwendung auf der obersten Ebene betrachtet. Wie die übrige Schachtelung oder Entschachtelung in noch tiefer liegenden Relationen syntaktisch in deren SQL ähnlichen Abfragesprache integriert werden könnte, wurde offen gelassen. In [RKB87] wurde die Anwendung der *nest* oder *unnest* Operatoren über mehrere Ebenen dadurch ermöglicht, indem diese Anweisungen in die FROM-Klausel integriert wurden. Dies erfolgt, indem in der FROM-Klausel Subqueries erlaubt werden, auf welche dann eine *nest* oder *unnest* Anweisung folgt. Die Autoren zeigen auf, wie dies mit den Mitteln von SQL möglich ist, und stellen zusätzlich die an SQL angelehnte Sprache *SQL/NF* vor. *SQL/NF* besitzt dabei einige Erweiterungen gegenüber den SQL-Dialekten für Relationen in erster Normalform, anderseits sind viele Konstrukte aber auch hier eher syntaktische Vereinfachungen.

Die Operatoren *nest* und *unnest* werden bei all den genannten Vorschlägen verwendet. Gemeinsam ist den verschiedenen Arbeiten auch, dass SQL für mächtig genug gehalten wird, mit geschachtelten Relationen umgehen zu können, wobei sie fokussieren, möglichst wenige Änderungen in die SQL-Sprache neu einzubringen. In [PA86] sind die Abweichungen vom SQL-Standard am größten. Zum einen liegt es daran, dass hier nicht nur das *Nested Relation Model* unterstützt wird, sondern auch das *extended NF² Model (eNF²)*, zum anderen empfinden die Autoren ihren dort beschriebenen SQL-Dialekt konsistenter und einfacher zu verstehen [PA86, Seite 1]. Im eNF² Modell werden nicht nur relationenwertige Attribute erlaubt, sondern hier sind auch Listen und Multisets erlaubt. Listen sind hierbei als eine Menge von Elementen zu verstehen, die sortiert abgespeichert sind. Die Elemente von Multisets besitzen keine Reihenfolge und im Gegensatz zu Sets dürfen die Elemente beliebig oft enthalten sein. Letztlich, wenn von den spezielleren Teilen für das eNF² Modell abgesehen wird, sind die Änderungen aber auch hier eher syntaktischer Natur.

Auch sind im SQL-Standard seit der Version SQL:1999[1] objektrelationale Erweiterungen aufgenommen worden. Diese Erweiterungen beinhalten die Möglichkeiten, mit geschachtelten Relationen arbeiten zu können [Sch03]. Die dort eingeführten Neuerungen gehen aber auch weit über die Möglichkeiten der geschachtelten Relationen hinaus. Allerdings sind die dortigen Mechanismen für geschachtelte Relationen anders als die, die im SECONDO DBMS vorhanden sind. Es werden dort zuerst Attribute vom Typ *ROW* definiert, die ein Tupel beschreiben und auch nur ein Tupel abspeichern können. Die Möglichkeit dann mehrere Tupel in einem Attribut vom Typ *ROW* abzuspeichern wird dann durch das davon unabhängige Schlüsselwort *MULTISET* ermöglicht. In Punkten wie dem *unnest* oder dem *nest* gleichen sich die Funktionalitäten aber wieder. Allerdings gibt es hier keinen eigenen *nest* Operator. Diese Funktion wird in Kombination mit der *GROUP BY*-Klausel und einer Aggregatfunktion *SET*, mit dem die entsprechenden Tupel der Gruppe gesammelt werden, realisiert. Aufgrund der unterschiedlichen Mechanismen des SQL-Standards und den im SECONDO DMBS implementierten Mechanismen wurde sich mehr an den vorher genannten Arbeiten orientiert.

[1] Auch als SQL 3 bezeichnet.

Kapitel 4

Entwurf

Geschachtelte Relationen werden auch, da diese gegen die erste Normalform verstoßen, in [SP82] als *non first normal form*[1] (¬1NF, NFNF oder NF²) Relationen bezeichnet. Relationen der *ersten Normalform* können daher als Spezialfall von geschachtelten Relationen betrachtet werden, nämlich solche, die keine Relation als Attribut enthalten. Im folgenden Text wird eine Relation, die als Attribut in einer anderen Relation vorhanden ist, auch als relationenwertiges Attribut oder als Subrelation bezeichnet. Wird von der obersten Ebene einer geschachtelten Relation gesprochen, ist damit die Relation gemeint, die ggf. weitere Subrelationen enthält, aber selbst nicht als Subrelation in einer anderen Relation vorhanden ist. Eine Relation ohne Subrelation ist eine flache Relation, bzw. eine Relation in erster Normalform.

Wird noch einmal die Relation *OrteH* (Tabelle 1.2 auf Seite 2) betrachtet, ist zu erkennen, dass in den Datensätzen auf der obersten Ebene der Einwohnerzahl in Hundertausendeinwohner (das Attribute *BevH*) die Subrelation *SubRel* zugeordnet ist. Diese Subrelation beinhaltet die Orte, die in dem entsprechenden Bereich der Einwohneranzahl liegen, also beispielsweise „Hagen“, „Bonn“ und „Gelsenkirchen“ mit den Einwohnerzahlen zwischen 200 000 und 300 000. Eine alternative Möglichkeit zur Realisierung wäre es, die Daten auf zwei Tabellen aufzuteilen um der ersten Normalform zu genügen. Um aber wieder ein entsprechendes Ergebnis zu ermitteln, wären die beiden Tabellen dann wieder über eine Join-Operation zu verbinden. Es können also durch geschachtelte Relationen Join-Operationen eingespart werden aus dem Grund, dass die zugehörigen Datensätze bereits als Subrelation integriert gespeichert werden[2]. Dieses integrierte Abspeichern von Daten, die auch logisch zusammengehören, wird dabei gerade als ein Hauptvorteil der geschachtelten Relationen angesehen [Gar08, Seite 57]. Aus demselben Grund wird die erste Normalform oft als zu restriktiv angesehen, da diese keine Subrelationen erlaubt. Insbesondere sollen durch die geschachtelten Relationen auch komplexere Anwendungsgebiete besser unterstützt werden [SP82][PA86][KHQ98]. In dem Artikel [KHQ98] gehen die Autoren sogar einen Schritt weiter und sehen die geschachtelten Relationen als einen Zwischenschritt zu den objektorientierten Datenbanken.

Für den Anwender von geschachtelten Relationen bedeutet es zum Teil davon befreit zu werden, Kenntnisse über das Datenbankschema, vor allem über die dort enthaltenen Beziehungen, zu besitzen. Werden beispielsweise mit einer Anfrage auf der Relation *OrteH* (Tabelle 1.2 auf Seite 2) Datensätze selektiert, sind auch direkt die entsprechenden Daten der zugehörigen Orte in dem Attribut *SubRel* verfügbar.

Ausgehend von den im Abschnitt über den Stand der Technik (Kapitel 3 auf Seite 17) genannten Arbeiten und den derzeitigen SQL Möglichkeiten für flache Relationen, sind folgende Anforderungen aus Sicht des Anwenders an eine Erweiterung des Secondo SQL-Dialekts um die Behandlung geschachtelter Relationen zu stellen:

[1] Mangels eines präzisen deutschen Begriffes sei hier direkt der englische Ausdruck verwendet

[2] Was nicht zwingend erfordert, dass die Datensätze auch physisch zusammen abgespeichert sind.

1. Es sollten Relationen angelegt werden können, die Subrelationen enthalten.
2. Es sollten Tupel auch in Subrelationen gelöscht, modifiziert und geändert werden können in der Art, wie es auch für die Relationen der ersten Normalform möglich ist.
3. Attribute vom Typ *arel* sollen in der Attributliste vorkommen dürfen und bei der Ausgabe entsprechend dargestellt werden.
4. Es sollten neue *arel* Attribute konstruiert werden können. Dies kann auf Basis vorhandener *arel* Attribute oder neu zu konstruierender *arel* Attribute erfolgen.
5. Es sollten WHERE-Bedingungen auf die *arel* Attribute angewendet werden können.
6. Es sollte die Schachtelungs-Struktur verändert werden können.

Abgesehen von den Operatoren *unnest* und *nest*, die die Struktur transformieren, ist die Hauptänderung, die sich durch die Einbeziehung von geschachtelten Relationen ergibt, dass nun dort, wo vorher nur atomare Werte auftauchen konnten, nun auch Relationen stehen können, wobei dies dann nicht immer ein gültiger SQL-Ausdruck sein muss. Ein Vergleich in der WHERE-Klausel zwischen einer Relation und einer Zahl wird im Allgemeinen nicht erlaubt sein. Der nächste entscheidende logische Schritt, der auch in [KHQ98] angesprochen wird, ist dass sich die Abschlusseigenschaft[3] von SQL zunutze gemacht wird. Da das Ergebnis einer SQL-Anfrage eine Relation ist[4], soll dort wo eine Relation verwendet werden kann, auch ein SQL-Statement stehen dürfen. Durch das Ermöglichen dieser geschachtelten Anfragen, kann auch der vierte Punkt der Anforderungen realisiert werden. Die Anfrage 4.1 ist ein Beispiel für eine solche Anfrage auf Basis der Tabelle *OrteH* (Tabelle 1.2 auf Seite 2). Hier wird durch eine geschachtelte Anfrage nur ein Attribut der Subrelation *SubRel* ausgewählt. Dies erfolgt dadurch, dass eine neue Subrelation mit der Attributbezeichnung *newsubrel* auf Basis des ursprünglichen Attributs *SubRel* erzeugt wird, anstatt dass direkt das Attribut *SubRel* ausgewählt wird.

Listing 4.1: Erstellen einer Subrelation auf Basis einer Subrelation

```
1 select [o:bevth, (select ort
2                   from   o:subrel) as newsubrel]
3 from   orteh as o
```

Weiter können mit dieser Technik auch ganz neue geschachtelte Relationen aufgebaut werden wie Anfrage 4.2 zeigt. Da ein Ort mehrere Postleitzahlen besitzen kann, können die Orte selektiert werden, und zu jedem Ort werden alle zugehörigen Tupel der *plz* Relation in einer neu erzeugten Subrelation ausgegeben. Die Relation *plz* ist Teil der *opt* Datenbank die im SECONDO DMBS enthalten ist und beinhaltet die Attribute *PLZ* und *Ort*.

Listing 4.2: Erstellen einer Subrelation auf Basis einer flachen Relation

```
1 select [o:ort, (select *
2                 from   plz
3                 where  ort=o:ort) as newsubrel]
4 from   orte as o
```

[3] Das Ergebnis einer SQL-Abfrage ist wieder eine Relation.
[4] Das gilt im SECONDO DBMS nicht immer, mehr dazu in Unterabschnitt 5.2.5 auf Seite 38.

Ein anderer Fall von Anfragen liegt vor, wenn Tupel auf einer höheren Ebene nur davon abhängig ausgewählt werden sollen ob eine Subrelation bestimmten Bedingungen genügt. Ermöglicht werden kann auch dies durch eine geschachtelte Anfrage, die in der WHERE-Klausel auf der übergeordneten Ebene fomuliert wird. Die Anfrage 4.3 zeigt so einen Fall. Es sollen dort nur Tupel der Beispielrelation *OrteH* (Tabelle 1.2 auf Seite 2) selektiert werden, welche einen Ort besitzen, der mit dem Buchstaben „B“ anfängt.

Listing 4.3: Selektion auf einer höheren Ebene auf Basis der Tupel einer Subrelation

```
1 select *
2 from   orteh as o
3 where  [exists(select ort
4                from   o:subrel
5                where  ort starts "B")]
```

Dabei ist aber zu beachten, dass die so selektierten Tupel noch immer die Subrelation *SubRel* ohne Veränderung beinhalten. Diese beinhaltet dann allerdings noch Orte, die ggf. nicht mit dem Buchstaben „B“ beginnen. Sollen aber auch dort die Tupel entsprechend gefiltert werden, also die Ausgabe nur aus den Tupel bestehen, die Orte beinhalten, deren erster Buchstabe ein „B“ ist, kann dies durch die Kombination der Techniken aus Beispiel 4.1 und Anfrage 4.3 realisiert werden, wie Beispiel 4.4 zeigt.

Listing 4.4: Selektionen auf zwei Ebenen

```
1 select [o:bevth, (select ort
2                   from   o:subrel
3                   where  ort starts "B") as newsubrel]
4 from   orteh as o
5 where  [exists(   select ort
6                   from   o:subrel
7                   where  ort starts "B")]
```

In dem konstruierten Beispiel kam der Fall nicht vor, aber möglich sollte es auch sein, dass die Schachtelungshöhe mehr als Eins beträgt und auf einer Ebene auch mehrere Subrelationen enthalten sein können. Deutlich wurde aber schon, wie mächtig die Verwendung von geschachtelten Anfragen ist, um geschachtelte Relationen aufzubauen, anzupassen oder zu selektieren. Die Möglichkeiten dieser geschachtelten Anfragen reichen damit ausreichend, der vierten und fünften Anforderung zu genügen.

Durch die Verwendung von geschachtelten Anfragen ist es aber nicht möglich die Strukturtiefe zu erhöhen oder zu verringern. Dazu sind die Operatoren *nest* bzw. *unnest* der *NestedRelationAlgebra* zu verwenden. Anfangs wurde schon erwähnt, dass auch dies auf mehreren Ebenen möglich sein soll. Daher soll hier die FROM-Klausel entsprechend erweitert werden, wie es auch in [RKB87] vorgeschlagen wurde. Die Integration wird hier so erfolgen, dass die FROM-Klausel derartig erweitert wird, dass auf den dort angegebenen Relationen Transformationen zur Strukturänderung durchgeführt werden können. Anfrage 4.5 zeigt so einen Fall: Dort wird das in der Relation *OrteH* enthaltene Attribut *SubRel* entschachtelt.

Listing 4.5: Anwendung des *unnest* Operators

```
1 select *
2 from   orteh unnest(subrel)
```

Analog zum *unnest* soll auch ein *nest* durchgeführt werden können. Dazu soll den Relationen ein Term der Form *nest(Attrs, Label)* angehängt werden können. Der Parameter *Attrs* beinhaltet die Attribute, anhand derer die Subrelation erzeugt wird, wie in 2.3.4.2 beschrieben, und der Parameter *Label* definiert den Namen der dem neuen Attribut zugewiesen wird. Im Listing 4.6 wird auch dieser Fall gezeigt.

Listing 4.6: Anwendung des *nest* Operators

```
1 select *
2 from   orte nest(bevt, ortrel)
```

Mit den *unnest* und *nest* Operatoren wird es somit auch möglich, die Struktur zu verändern, also der Hauptanforderung Nr. 6 zu genügen. Auch hier sollen diese Operatoren auf jeder Ebene des SQL-Statements angewendet werden können, und es sollen die beiden Operatoren auf alle Arten von Relationen, die in der FROM-Klausel angegeben werden, angewendet werden können. Dies sind zum einen die regulären Relationen sowie die Subrelationen. Allerdings zeigte sich hier die Notwendigkeit, geschachtelte Anfragen in der FROM-Klausel ermöglichen zu müssen. Ansonsten wäre die Implementierung zur Behandlung geschachtelter Relationen des SQL-Dialekts nicht mächtig genug. Beispielsweise wäre es dann nicht möglich, über ein SQL-Statement aus der Tabelle *Orte* (Tabelle 1.1 auf Seite 1) ein Ergebnis wie die Tabelle *OrteH* (Tabelle 1.2 auf Seite 2) zu erzeugen.

Duch die Verwendung von geschachtelten SQL-Anfragen in der FROM-Klausel in Verbindung mit den Operatoren zu Strukturänderungen, wird dies dann aber möglich. Die Anfrage 4.7 zeigt diesen Fall. Dazu wird zuerst eine Relation mit einer geschachtelten Anfrage aufgebaut, die alle notwendigen Attribute beinhaltet. Insbesondere wird das Attribut erzeugt, das den Wert beinhaltet, anhand dessen die *nest* Operation durchgeführt wird. Als letztes wird auf dem neuen Attribut *bevth* die *nest* Operation angewendet.

Listing 4.7: Anwendung des *nest* Operators auf eine geschachtelte Anfrage

```
1 select *
2 from   (
3          select [bevt div 100 as bevth,
4                  bevt, ort, kennzeichen, vorwahl]
5          from   orte
6        ) nest(bevth, subRel)
```

Der dritte Punkt der Liste von Anforderungen ist eher trivial. Der Optimierer ist schon ohne weitere Änderungen in der Lage neue Typen zu behandeln. Daher beschränkt sich dieser Punkt auf die Implementeriung eines geeigneten Ausgabeformates für *arel* Attribute. Dieser Punkt wird nicht weiter ausgeführt. Das Format für die Textausgabe auf der Konsole im Optimierer lehnt sich dabei an das Ausgabeformat der bereits vorhandenen Ausgabeformatierung der C bzw. C++ Implementierung an.

Bisher wurde auf die Aspekte von SQL-Anfragen eingegangen, also auf die Punkte drei bis sechs der anfangs definierten Anforderungen. Bei den ersten beiden Anforderungen ist zumindest der Punkt eins realisierbar, also das Erstellen von geschachtelten Relationen mit einer *create table* Anweisung. Der Nutzen einer solchen Implementierung ist aber begrenzt, denn der zweite Punkt, also das Einfügen, Löschen und Modifizieren von Tupeln in Relationen vom Typ *nrel* oder *arel* ist derzeit in der *NestedRelationAlgebra* nicht implementiert. Daher wurden diese Aspekte für die Änderung des SECONDO SQL-Dialekts nicht weiter berücksichtigt.

Nicht betrachtet wurden dabei in dieser Arbeit mögliche Optimierungen der Ausführungspläne, die im Zusammenhang mit geschachtelten Relationen vom Anfrageoptimierer erstellt werden. Fokussiert wurde hier also, den SECONDO SQL-Dialekt in die Lage zu versetzen, mit geschachtelten Relationen überhaupt arbeiten zu können.

4.1 Spezifizieren der Grammatik

Hier soll zuerst die Klammerung in den SECONDO-SQL-Statements betrachtet werden, die durch die runden Klammen (und) erfolgt. Bisher wurde die Klammerung, wie in SQL auch üblich, für die geschachtelten Anfragen verwendet. Allerdings basiert die Klammerung nicht auf der eigentlichen Implementierung des SQL-Dialektes, sondern diese sind Bestandteil der Prolog-Grammatik. In Prolog dient die Klammerung unter anderem dazu, die Priorität der Operatoren oder deren Assoziativität zu durchbrechen, wie dies auch von den mathematischen Formeln bekannt ist [SS94]. Die Klammerung ist also nicht immer notwendig, auch wenn diese in der Grammatik mit angegeben wird. Andererseits wiederum könnten Klammern auch gesetzt werden ohne dass es nötig wäre. Hier in dieser Arbeit werden daher Klammern immer so gesetzt wie es von SQL her bekannt ist und es in der SECONDO SQL-Grammtik definiert ist. Es sollte aber im Hinterkopf behalten werden, dass die Klammerung nicht direkter Bestandteil der Implementierung ist. Daher ist nicht jede Klammerung der Grammatik in den Prolog-Prädikaten der Implementierung wiederzufinden.

Aus den Anforderungen und den in Kapitel 4 auf Seite 19 aufgezeigten Beispielen soll zuerst die Grammatik des SQL-Dialekts um die Anforderungen der geschachtelten Relationen erweitert werden. Es ist insofern sinnvoll, da sich auch die Prolog-Prädikate zum großen Teil in der Grammatik wiederfinden lassen.

Die relevanten Stellen, an denen Modifikationen durchgeführt werden müssen, sind nun konkreter zu fassen, woraus sich folgende Punkte ergeben:

- Die SELECT-Klausel erweitern, um mit geschachtelten SQL-Anfragen neue Subrelationen konstruieren zu können.
- Die FROM-Klausel um die Möglichkeit erweitern, auch hier geschachtelte Anfragen und Relationen vom Typ *nrel* und *arel* angeben zu können.
- Die WHERE-Klausel zur Selektion auf Basis von *nrel* und *arel* Relationen erweitern.

Die Erweiterung der SELECT-Klausel bedingt das Erlauben von Subqueries zum aufbauen neuer Subrelationen. Eine Regel dazu ist bereits vorhanden:

```
result      →  attr| attr-expr as newname | subquery-aggr as newname
```

Die Regel *result* beschreibt die Elemente der SELECT-Klausel. Die beiden Regeln *attr* und *attr-expr as newname* beschreiben Attribute aus Relationen oder Ausdrücke die dem Ergebnis der Anfrage neue Attribute hinzufügt. Die Regel *subquery-aggr* erlaubt es, dass auch geschachtelte Anfragen in der SELECT-Klausel verwendet werden dürfen. Diese Regel bleibt auch in dieser Form bestehen, allerdings muss diese Regel teilweise neu implementiert werden. Bisher ist die Forderung für geschachtelte Anfragen dieser Art, dass sie einen atomaren Wert liefern und insbesondere unabhängig sind. Es können hier also keine Attribute der äußeren SQL-Statements verwendet werden. Daher ist es hier die Aufgabe, diese so zu erweitern, dass ein relationenwertiges Attribut durch eine geschachtelte Anfrage konstruiert werden kann. Bisher war für Subqueries dieser Art die Strategie eine andere. In der Rewriting Phase wurde die unabhängige geschachtelte Anfrage direkt ausgeführt. Da die Anfrage bisher nur einen atomaren Wert als Ergebnis liefern durfte, wurde dieser Wert an Stelle der geschachtelten Anfrage eingesetzt [Pon12].

Bei der WHERE-Klausel sind zwar Änderungen in der Implementierung notwendig, aber die notwendigen Möglichkeiten auf der Ebende der Grammtik werden schon durch die vorhandenen Grammatikregeln abgedeckt. Hingegen sind die Änderungen der FROM-Klausel am deutlichsten. Um die im vorherigen Kapitel diskutierten Änderungen einzubringen ist die neue Regel für die Angabe von Relationen in der FROM-Klausel folgende:

```
rel     → relname asrule
        | subquery-table asrule
        | arelname asrule
        | outervar:arelname asrule
        | rel unnest (attr) asrule
        | rel nest (attr, newname) asrule
        | rel nest ([attr-list], newname) asrule
asrule  → as newname
        | ϵ
```

Outervar ist in dieser Regel eine Variable eines umschließenden SQL-Statements. Das Verwenden von *arel* Subrelationen ist in der FROM-Klausel nur in geschachtelten Anfragen möglich. Da die Relation vom Typ *arel* Attribut eines Tupels ist, ist das *arel*-Attribut zuerst in einer umgebenen Anfrage zu selektieren. Mit Hilfe der geschachtelten Anfragen kann dann auf die Tupel des *arel* Attributs des umgebenen SQL-Statements zugegriffen werden. Im Secondo DBMS ist es allerdings möglich, dass ein *arel* Objekt auch selbstständig in der Datenbank existiert, also gerade nicht ein Attribut eines Tupels ist. Hier wird aber dieser Fall nicht weiter betrachtet, da dies nicht der beabsichtigte Zweck der *arel* Objekte ist.

Damit sind die Möglichkeiten der geschachtelten Relationen auf der grammatikalischen Ebene beschrieben. Es sei noch darauf hingewiesen, dass geschachtelte Anfragen nicht aus mehreren SQL-Anfragen bestehen können, die per *union* oder *intersection* verknüft werden.

4.2 Strategie

Bisher stammten Attribute eines SQL-Statements entweder immer aus flachen Relationen oder aus einfachen[5] geschachtelten Anfragen der FROM-Klausel, sofern hier die Option *subqueryUnnesting* im Optimierer aktiviert war. Realisiert wurde dies, indem für geschachtelte Anfragen, die in der FROM-Klausel verwendet werden, eine temporäre Relation auf Basis dieser geschachtelten Anfrage erzeugt wurde. Auf dieser temporären Relation kann anschließend die umgebene Anfrage wieder wie für eine flache Relation arbeiten. Diese Strategie ist für die Subrelationen nicht mehr anwendbar. In der FROM-Klausel geschachtelter Anfragen können nun Subrelationen auftauchen, also Attribute der umgebenen Anfrage (Vgl. Anfrage 4.3 auf Seite 21). Diese Subrelationen sind insbesondere an den gerade aktuellen Datensatz gebunden, der verarbeitet wird. Dies ist die Ursache dafür, dass die angesprochene statische Materialisierung der geschachtelten Anfragen durch einen anderen Mechanismus ersetzt werden muss. In die Betrachtung muss weiterhin einbezogen werden dass auf den Elementen der FROM-Klausel auch noch Transformationen der Operatoren *unnest* und *nest* durchgeführt werden können.

Wird eine Anfrage an den Optimierer übergeben, folgt nach der rewriting Phase (Vgl. Abschnitt 2.3.1 auf Seite 7) die lookup Phase. Hier wird nun unter anderem nachgesehen, ob die in der Anfrage verwendeten Attribute auch in den jeweiligen Relationen existieren. Die korrekte Schreibweise wird ermittelt und später werden mit diesen Informationen weitere Informationen ermittelt, wie z.B. die Speichergrößen der Attribute. Dies bedingt, dass der Optimierer weiß, um welches Attribut es sich handelt, um in die Lage versetzt zu werden den Ausführungsplan zu erzeugen. Wie im vorherigen Absatz dargelegt, war der Fall bisher klar, Attribute kamen aus flachen Relationen. Nun können die Attribute aus verschiedenen Quellen stammen, allerdings zeigten sich genau hier im Laufe der Implementierung die größten Probleme. Der erste Ansatz bestand darin, sobald das Lookup eines Attributs durchgeführt werden soll, dies in der Art durchzuführen, wie es für die flache Relationen gemacht wurde, also in dem Moment des Lookups sich die nötigen Informationen zu beschaffen. Problematisch war hier, war dann die unterschiedlichen Fälle zu berücksichtigen und die bisherigen Informationen soweit zurückzugehen bis Informationen zum Attribut vorlagen. Dazu mussten nicht nur alle Umbenenungen zurückverfolgt werden. Inbesondere waren geschachtelte Anfragen und Restrukturierungen des *nest* und *unnest* wieder rückwärts zu durchlaufen. Letztendlich war es zwar möglich, diesen Ansatz zu verwenden, aber es war nur noch schwer nachvollziehbar, wie genau diese Rückverfolgung abläuft, insbesondere wenn alle Fälle kombiniert in einer Kette von geschachtelten Anfragen vorlangen. Das zeigte sich dann auch immer wieder bei der Fehleranalyse. Diese Problematik führte dazu einen anderen Lösungsansatz zu verfolgen.

Dazu soll die Lookup Phase genauer betrachtet werden. Von Interesse ist dabei, in welcher Reihenfolge die Beschreibung der Relationen (Relationenschemata) zustande kommen und verwendet werden. Beim Lookup einer Anfrage werden dabei zuerst die Elemente der FROM-Klausel betrachtet. Kommt dort eine weitere Anfrage vor, wird für deren FROM-Klausel direkt wieder das Lookup als erstes durchgeführt usw. Es wird dabei also rekursiv vorgegangen. Ist das Lookup der FROM-Klausel abgeschlossen, liegt die Datenbasis in Form einer oder mehrer Relationenbeschreibungen vor. Nachdem die Informationen der FROM-Klausel vorliegen, ist der Optimierer in der Lage für weitere Klauseln wie der SELECT- oder WHERE-Klausel das Lookup durchzuführen. In diesen Klauseln ist dann auch nur das Ergebnis der FROM-Klausel der jeweiligen Anfrage verwendbar. Wichtig ist hierbei, dass keine Daten in die geschachtelten Anfragen der FROM-Klauseln hinab-

[5] Diese Funktionalität war insbesondere bei den Join-Operationen eingeschränkt.

gereicht werden können, denn hier wird gerade erst die Datenbasis festgelegt. Anders hingegen bei geschachtelten Anfragen in der SELECT- oder WHERE-Klausel. Hier kann sich auf Werte bezogen werden, die aus umgebenen Anfragen stammen (Vgl. Anfrage 4.2 auf Seite 20). Im Falle von geschachtelten Relationen kann hier sogar in der FROM-Klausel ein Atttribut vom Typ *arel* stehen, das einer umgebenen Anfrage entstammt. Festzuhalten ist damit, dass Relationenbeschreibungen der FROM-Klauseln zur weiteren Verwendung im lookup „hochgereicht“ werden und in die anderen Klauseln „hinabgereicht“ werden.

Anhand der Anfrage 4.8 soll dieses verdeutlicht werden. Zuerst wird hier das Lookup der FROM-Klausel durchgeführt. Dies ist eine geschachtelte Anfrage mit einer anschließenden *nest* Operation. In diese geschachtelte Anfrage wird dann weiter hineinverzweigt und das Lookup dieser Anfrage durchgeführt. Da hier eine flache Relation in der FROM-Klausel vorhanden ist, wird nach dem Lookup dieser geschachtelten Anfrage zurück zum Lookup der Anfrage der obersten Ebene zurückgekehrt. Dabei wird durch die geschachtelte Anfrage auch eine Relationenbeschreibung definiert. Diese Relationenbeschreibung wird durch den *nest* Term wiederum einer Änderung entsprechend der Arbeitsweise des Operators unterzogen. Damit liegt nun die Datenbasis der Anfrage auf der obersten Ebene fest. Diese so fest gelegte Relationenbeschreibung kann dann in die beiden geschachtelten Anfragen der SELECT- bzw. WHERE-Klausel hinabgereicht werden. In der Lookup Phase dieser geschachtelten Anfragen wird dann die Beschreibung der Relation benötigt, die in Zeile 6-9 definiert wird.

Listing 4.8: Beispiel SQL zur Lookup-Arbeitsweise

```
1  select [o:bevth,
2          (select ort
3           from   o:subrel
4           where  ort starts "B"
5          ) as newsubrel]
6  from (select [bevt div 100 as bevth, bevt, ort,
7                kennzeichen, vorwahl]
8        from   orte
9       ) nest(bevth, subRel) as o
10 where [exists(select ort
11                from   o:subrel
12                where  ort starts "B"
13               )]
```

Um das Lookup der Attribute wesentlich zu vereinfachen und den vielen Fallunterscheidungen zu begegnen wird nun beim rekursiven Lookup der FROM-Klauseln direkt nach dem Lookup jeden Elementes der FROM-Klausel eine Beschreibung der Relation aufgebaut. Die Beschreibung soll alle nötigen Informationen beinhalten, die im Lookup und späteren Phasen benötigt werden. Die „nötigen“ Informationen, sind dabei die Informationen, die, wenn sie nicht zur Verfügung ständen, ein nochmaliges, rekursives Durchlaufen benötigen würden. Mit diesen so gesammelten Informationen kann das Lookup der Attribute nur mit dem Zugriff auf die Beschreibung der Elemente der aktuellen FROM-Klausel durchgeführt werden.

Kapitel 5

Implementierung

Im Folgenden sollen die wesentlichen Aspekte der Implementierung betrachtet werden, da nun Klarheit darüber besteht, was erreicht werden soll, und auch die Strategie dorthin klar ist.

5.1 Datenbankkatalog

Der Optimierer benötigt einige Informationen über die Datenbank wie die Relationenbeschreibungen, Anzahl der Tupel in den Relationen (Kardinalität), welche Indexstrukturen vorhanden sind etc. Ohne diese Informationen könnte der Optimierer einerseits ein SQL-Statement nicht in einen Ausführungsplan übersetzen, anderseits fehlen auch wichtige Informationen für die Optimierung. Da geschachtelte Relationen bisher hier nicht behandelt werden, muss auch diese Komponente entsprechend erweitert werden.

Bisher wurde für ein Attribut einer Relation zum Beispiel die genaue Bezeichnung als Prolog-Faktum wie folgt gespeichert:

```
1 storedSpell(opt,orte:bevt,bevT).
```

Es wird hier der Name der Relation *orte* vom Namen des Attributs *bevt* per Doppelpunkt getrennt. Als erstes steht im dem *storedSpell* Faktum der Datenbankname und als drittes die korrekte Schreibweise des Attributs, wobei die Schreibweise des ersten Buchstabens noch speziell gehandhabt wird. Bei den geschachtelten Relationen wurde diese Systematik dabei so weiter fortgeführt. Das Attribut *Kennzeichen* der Beispielrelation *OrteH* (Tabelle 1.2 auf Seite 2) würde also wie folgt gespeichert werden:

```
1 storedSpell(optext,orteh:subrel:kennzeichen,kennzeichen).
```

Es gilt hierbei, dass das erste Element der Name der Relation ist. Das letzte Element ist ein Name eines Attributs, das auch vom Typ *arel* sein kann, da auch hierfür die Schreibweisen bekannt sein müssen. Im mittleren Teil eines solchen Terms stehen nur Attribute die vom Typ *arel* sind. Um solche Terme zu ermöglichen war es nötig die Operatordeklaration des Doppelpunktes auf

```
1 :- op(200, xfy, :).
```

zu ändern.

Genaueres zu den Operatordeklarationen kann z.B. [SS94] entnommen werden. Geändert wurde hier die Assoziativität von *fx* auf *xfy*. Ein Term der Form a:b:c wird dadurch als a:(b:c) interpretiert. Dadurch wird es ermöglicht, das erste Element einfach abspalten zu können, analog zu den Listen in Prolog. Der Nachteil dieser Methode ist, dass Elemente nicht mehr in jedem Fall mit z.B. dem Prolog Term *A=B:C* verkettet werden können, wenn *B* auch einen Term mit einem Doppelpunkt beinhaltet. Daher wurde ein neues Prädikat *appendAttribute/3* implementiert um das Anhängen eines Attributs an einen Attributterm durchzuführen.

Bei den anderen Fakten, die Informationen über Attribute beinhalten, ist die Methodik mit der Verkettung durch Doppelpunkte identisch. Bei manchden Fakten wird nur der Name der Relation getrennt von dem restlichen Attributterm gespeichert.

Weiter erwähnenswert sind hier die Fakten *storedCard(+Database, +Relation, +Card)*. Für flache Relationen enthalten diese Fakten die Anzahl an Tupeln der jeweiligen Relation. Auch für Relationen von Typ *nrel* wird in dem entsprechenden Faktum die genaue Kardinalität festgehalten. Diese kann analog der Methode ermittelt werden, wie dies für die regulären Relationen erfolgt. Für *arel* Relationen ist die Sache aber komplizierter, denn es gibt hier nicht mehr die eine Kardinalität. Die Kardinalität ist dann abhängig vom jeweiligen Tupel, in dem das Attribut vorhanden ist, und wird daher in der Regel nicht immer gleich sein. Es kann den Tupeln an sich auch nicht angesehen werden, wieviele Tupel in der Subrelation vorhanden sind. In der Relation *OrteH* (Tabelle 1.2 auf Seite 2) könnte dies an den Werten in der Spalte *BevT* festgemacht werden. Betrachtet man dazu die Relation *CardExample* (Tabelle 5.1), ist zu erkennen, dass anhand des Tupels, in dem die Subrelation vorhanden ist, nicht erkannt werden kann ob die betrachtete *arel* Subrelation ein Tupel oder eine Million Tupel beinhaltet.

Tabelle 5.1: Die Relation *CardExample*

Wert	SubRel
	Wert2
1	1
1	1
	2
	...
	1 000 000

Für die Optimierung, in der die Kardinalität für die Kostenschätzung verwendet wird, kann nun nicht immer die korrekte Kardinalität zur Verfügung gestellt werden, da die Optimierung alle Tupel betrachtet. Wenn in der Optimierung die Kardinalitäten für Subrelationen, abhängig von dem Tupel, in dem das aktuell betrachtete *arel* Attribut enthalten ist, betrachtet werden würde, wäre das Ergebnis, dass je nach Kardinalität verschiedene Ausführungspläne erzeugt werden müssten. Zum einen ist völlig unklar, wie so eine Technik aussehen könnte, zum anderen wären aber umfangreiche Änderungen notwendig. Hinzu kommt hier, dass die Ermittlung der Kardinalität dann mit zusätzlichen Kosten verbunden ist. Auch ist nicht klar wie groß der Effekt einer solchen spezielleren Optimierung ist. Für eine solche Optimierung wäre es dann auch nötig, die *tupleid*, also der Wert, der ein Tupel eindeutig identifiziert, bereitzuhalten. Gerade im Beispiel der Tabelle *CardExample* war zu sehen, dass die Werte der Tabelle selbst keine Möglichkeit boten, auf die Kardinalität der Subrelation zu schließen, bzw. überhaupt die Subrelation zu identifizieren.

Implementiert wurde daher eine einfache Ermittlung des Durchschnittwertes für die Kardinalität eines *arel* Attributs. Dazu wird in den Fakten *storedCard(+Database, +Relation, +Card)* für *arel* Attribute die Gesamttupelzanzahl, über alle Tupel, in denen das *arel* Attribute enthalten ist, abgespeichert. Die Abfrage der Kardinalität für eine Subrelation erfolgt dann, wie für die anderen

Relationen, über das Prädikat *card(+Rel, ?Size)*. Bei dem Aufruf dieses Prädikates wird die Gesamtkardinalität dann umgerechnet auf die Durchschnittsanzahl der Tupel in einem *arel* Attribut für ein Tupel, in dem dieses *arel* Attribut enthalten ist. Bezogen auf die Beispielrelation *OrteH* wäre dies dann $6/3 = 2$, da insgesamt 6 Tupel in der Subrelation *subrel* vorhanden sind und drei Tupel in der Relation *OrteH* auf der obersten Ebene.

5.2 Optimierer: Die Lookup-Phase

Die umfangreichsten Änderungen für die Behandlung geschachtelter Relationen sind in der Lookup-Phase durchzuführen. Insbesondere ist hier auch die Strategie implementiert, die bereits in Abschnitt 4.2 auf Seite 25 beschrieben wurde. Nachdem zuerst die Änderungen der Subqueries-Erweiterung beschrieben werden, werden anschließend die Lookup-Prädikate für die Erweiterung der FROM-Klausel betrachtet, da diese die zentrale Komponente dieser Erweiterung sind. Danach werden die Lookup-Prädikate für Subqueries in der SELECT- und FROM-Klausel betrachtet. Als letztes wird das eigentliche Lookup für die Attribute betrachtet.

5.2.1 Die Erweiterungen der Subqueries-Erweiterung

Für die Realisierung der Behandlung geschachtelter Relationen, die nun insbesondere wesentlichen Gebrauch der Subqueries-Erweiterung macht, wurden einige Änderungen der Subqueries-Erweiterung notwendig.

Die wesentlichste Änderung ist, dass die geschachtelten Anfragen an sich nicht mehr identifiziert werden konnten und manche Informationen aus der Lookup Phase nachher nicht mehr zur Verfügung standen. Allerdings bestand bisher auch nicht die zwingende Notwendigkeit wie für die Behandlung der geschachtelten Relationen. In der Subquery Erweiterung konnten geschachtelte Anfragen unabhängig verarbeitet werden. Die einzige Verbindung zwischen den Subqueries waren dort verbindende Attribute in der WHERE-Klausel. Genau diese verbindenden Prädikate wurden vor der Verarbeitung entfernt. Dann wurde die Subquery verarbeitet, was insbesondere die Erstellung des Ausführungsplans für diese Subquery beinhaltet, abschließend wurden die entfernten Prädikate wieder dem Ausführungsplan hinzugefügt. Dies gestaltet sich für die Behandlung der geschachtelten Anfragen nicht mehr so einfach. Wird hierzu die Anfrage 5.1 betrachtet, ist zu erkennen, dass hier die Subquery nicht unabhängig analysiert werden kann. In dem Beispiel wären Informationen über die *o:subrel* Subrelation notwendig, die aus dem umgebenen SQL-Statement stammt.

Listing 5.1: Beispiel einer Anfrage, die nicht unabhängig analysiert werden kann

```
1 select *
2 from   orteh as o
3 where  [exists(select ort
4                from   o:subrel
5                where  ort starts "B")]
```

Eine weitere Hürde bei der Implementierung stellte die nicht vorhandene Trennung der Fakten dar, wenn das Lookup der Subqueries durchgeführt wurde. Bei dem Lookup einer Anfrage werden verschiedene Fakten der Prolog-Umgebung hinzugefügt. Für das Lookup von Suqueries wurden dann die Fakten immer weiter hinzugefügt, allerdings ohne die Informationen, zu welcher Subquery die Fakten überhaupt gehören. Dies funktionierte bisher, da einerseits die Verwendung einer

Relation eindeutig in der gesamten Anfrage inklusive Subqueries sein musste und da durch das rewriting der Query bisher ausgeschlossen war, dass ein Faktum *isStarQuery* nur hinzugefügt wurde, wenn die Anfrage auf der obersten Ebene der Form *select * from ...* war. Eigentlich müsste jedem Faktum das in der Lookup Phase hinzugefügt wird, die Information mitgegeben werden, zu welcher Subquery dieses gehört. Dann müssten aber alle diese Fakten im ganzen Optimierer geändert werden. Da diese Erweiterung aber eine zuschaltbare Erweiterung des Optimieres ist, wurde eine andere Strategie verwendet, die wesentlich weniger Änderungen in den Hauptkomponenten des Optimierers verursacht.

Um erst einmal die Subqueries identifizieren zu können, werden die geschachtelten Anfragen mit einer *Subquery ID (SQID)* nummeriert. Betrachtet man dazu eine SQL-Anfrage die mehrere Subqueries beinhaltet, ist leicht zu erkennen, dass eine Anfrage sich als Baum darstellen lässt, wenn die Wurzel die Anfrage auf der obersten Ebene darstellt und die Knoten die einzelnen geschachtelten Anfragen darstellen. Wird die Anfrage 4.8 auf Seite 26 als Baum bezüglich der Subquieres dargestellt, sieht der daraus resultierende Baum folgendermaßen aus:

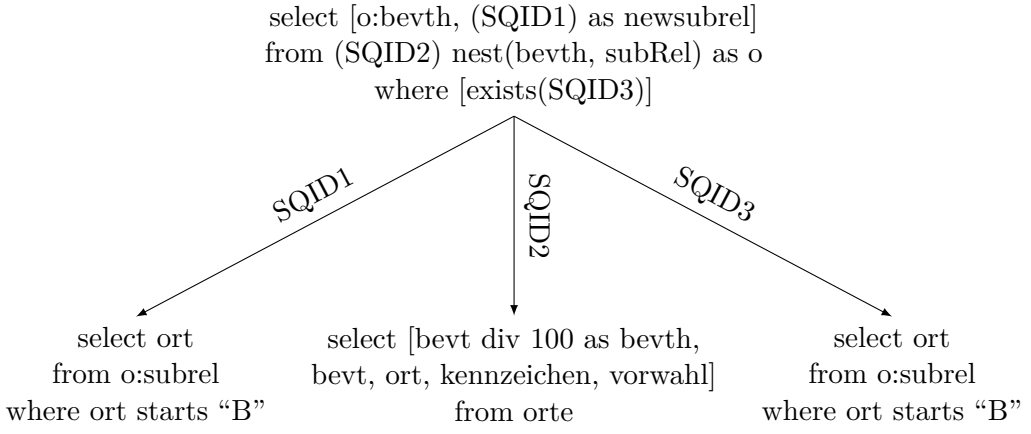

Abbildung 5.1: Baumdarstellung der Anfrage 4.8 bezüglich der geschachtelten Anfragen

Um die geschachtelten Anfragen zu nummerieren, wird folgendes Verfahren angewendet. Dem SQL-Statement der obersten Ebene wird immer die SQID 0 zugewiesen. Beim Eintritt in das Lookup der Subqueries wird zuerst das Prädikat *enterSubquery(+Type)* aufgerufen. Dieses Prädikat vergibt unter anderem der geschachtelten Anfrage eine neue SQID und speichert ein Faktum *subqueryDFSLabel(PrevSQID, SQID, Type)* ab, das auch die Vorgänger-*SQID* beinhaltet. Diese Fakten können daher auch als Kanten in dem Baum der geschachtelten Anfragen betrachtet werden. Zusätzlich wird auch noch der Typ festgehalten. Der Typ spezifiziert, ob die geschachtelte Anfrage in der SELECT-, FROM- oder WHERE-Klausel auftritt. Bedingt durch die Art der rekursiven Lookup-Aufrufe, realisiert dieses Verfahren eine Tiefensuche (Depth-first search). Zwei Eigenschaften dieser so implizit aufgebauten Baumstruktur sind für die weitere Verwendung relevant: zum einem, dass der Wurzelknoten eindeutig ist und zum anderen dass jeder Knoten bis auf den Wurzelknoten genau einen definierten Vorgängerknoten besitzt.

Mit Hilfe dieser Nummerierungsstrategie lässt sich nun das zweite Problem der nicht separierten Lookup-Umgebungen lösen. Dazu wurde das *enterSubquery* Prädikat erweitert. Hier wird nicht mehr nur eine neue SQID vergeben. Zudem wird vorher die aktuelle Lookup-Umgebung gesichert

und für das folgende Lookup eine leere Umgebung bereitgestellt. Beim Austritt des Lookups einer Subquery wird dann mit dem Prädikat *leaveSubquery/0* diese Umgebung der Subquery gesichert und die vorhergehende Umgebung wiederhergestellt. Damit besitzt nun jede Subquery seine eigene Lookup-Umgebung und später stehen auch weiterhin alle Umgebungen zur Verfügung. Insbesondere können diese dann für die Plangenerierung wiederhergestellt werden. Zum Wiederherstellen einer Lookup-Umgebung wurde das Prädikat *restoreSQID(+SQID)* implementiert. Die folgende Auflistung zeigt die genauen Fakten, aus denen die *Lookup-Umgebung* besteht:

- *variable/2*: Speichert Relationen der WHERE-Klausel, denen eine Variable zugewiesen wurde.
- *queryRel/2*: Speichert Relationen der WHERE-Klausel, denen keine Variable zugewiesen wurde.
- *queryAttr/1*: Fakten zu Attributen, die nicht aus einer Relation kommen sondern auf einem Ausdruck basieren.
- *usedAttr/2*: Speichert alle Attribute, die aus den Relationen der FROM-Klausel verwendet werden.
- *isStarQuery/0*: Dieses Faktum wird genau dann hinzugefügt, wenn das gerade betrachtete SQL-Statemtent der Form *select * from ...* ist. An gewissen Stellen in der Implementierung ist es wichtig zu beachten, dass für eine Anfrage der Form *select count(*) from ...* das Faktum *isStarQuery* nicht hinzugefügt wird.

Das Trennen der Lookup-Umgebungen ermöglicht dabei eine weitere Funktionalität. Bisher konnten in der WHERE-Bedingung nur Attribute erscheinen, die aus direkt umgebenen SQL-Anfragen stammen. Beispiel 5.2 zeigt so einen Fall: Hier wird in der geschachtelten Anfrage die Verbindung *ort=o:ort* hergestellt. Dabei kann das Attribut *o:ort* nur aus der direkt umschließenden Anfrage kommen.

Listing 5.2: Anfrage mit einem Zugriff auf ein Attribut der direkt umgebenen Anfrage

```
1 select [o:ort, (select *
2                 from   plz
3                 where  ort=o:ort) as newsubrel]
4 from   orte as o
```

Wünschenswert wäre hier natürlich, dass Attribute nicht mehr nur aus der direkt umschließenden Anfrage stammen. Realisierbar wäre dies auch mit der Subqueries-Erweiterung gewesen, da auf dem dort verwalteten Stack die Informationen vorhanden waren. Allerdings konnte durch die fehlende Trennung der Lookup-Umgebungen nicht zwischen nicht erreichbaren und erreichbaren Attributen unterschieden werden. Implementiert wurde diese Möglichkeit allerdings nur für die Verwendung von Subrelationen in der FROM-Klausel. Anfrage 5.3 auf der nächsten Seite zeigt so einen Fall. Hier wird sich in der Anfrage durch den Term *o:subrel* mit der Variable *o* auf ein Attribut bezogen, das nicht aus der direkt umgebenen Anfrage stammt. Das in dem Beispiel die Variable *o* verwendet wird, dient hier der Verdeutlichung woher das Attribut *subrel* stammt. Implementiert wurde auch die Möglichkeit die Variable wegzulassen und das Attribut nur über den Attributnamen anzusprechen.

Listing 5.3: Verwendung eines *arel* Attributs in einer zwei Ebenen tiefer liegenden Subquery

```
1 select (select (select *
2                 from o:subrel) as subrel2
3         from orte) as subrel1
4 from orteh as o
```

Die letzte, wesentliche Änderung an der Subquery-Erweiterung betrifft die Verwendung von geschachtelten Anfragen innerhalb der SELECT-Klausel, wie in der Anfrage 5.3. Bisher waren hier ausschließlich nicht verbindende Anfragen erlaubt, die als Ergebnis einen atomaren Wert liefern. Um nun die Verwendung von SQL-Statements in der SELECT-Klausel zu ermöglichen, die anstatt eines atomaren Wertes ein *arel* Attribut erzeugen, wurde ein neues Prädikat *subquery_plan_to_atom/2* implementiert. Im wesentlichen unterscheidet sich dieses Prädikat nicht von den anderen, die für die WHERE-Klausel implementiert wurden. Der wesentlichste Unterschied ist, dass die Auswertung nicht in einem Boolschen Wert resultiert wie für Prädikate der WHERE-Klausel, sondern in einem Attribut vom Typ *arel*.

5.2.2 Die Erweiterungen der Lookup-Phase im Detail

Im Kern geht es hier um die Erweiterungen der Lookup-Phase, nämlich um:

- Das Lookup der Relationen in der FROM-Klausel
- Das Lookup der Attribute der SELECT-Klausel
- Das Lookup der Attribute in den WHERE-Prädikaten

Die Problematik dieses Lookup von Attributen wurde in Kapitel 4 auf Seite 19 schon von der Idee her beschrieben. Im Folgenden wird daher thematisiert, wie die Strategie implementiert wurde. Besprochen wurde schon die Reihenfolge des Lookups von SQL-Anfragen. Da also die FROM-Klausel als erstes analysiert wird, soll damit begonnen werden. Das Lookup der einzelnen Terme der WHERE-Klausel erfolgt durch die Prädikate *lookupRel(+InTerm, -OutTerm)*. Dabei wird der Teil *InTerm* analysiert und in den *OutTerm* umgewandelt, der für die weitere Verarbeitung einfacher zu handhaben ist [SOM12]. Der eigentlich relevante Lookup-Vorgang erfolgt dann in den implementierten Prädikaten *nrLookupIRRel(+InTerm, -OutTerm)* für die folgenden Fälle, die auch in der Definition der Grammatik erkennbar waren:

- Relationenwertige Attribute von Typ *arel* in der FROM-Klausel.
- Geschachtelte Anfragen in der FROM-Klausel.
- Terme, gefolgt von einer *unnest* oder *nest* Anweisung in der FROM-Klausel.

Das Lookup der Relationen, die nicht in die obige Auflistung fallen, funktioniert dabei weiterhin ohne Änderungen. Spezieller ist der Fall, wenn eine Relation vom Typ *rel* oder *nrel* von einer *unnest* oder *nest* Transformation gefolgt wird. In diesem Fall ist es notwendig für die Relationen von Typ *rel* oder *nrel* das Lookup durch das *nrLookupIRRel(+InTerm, -OutTerm)* Prädikat durchzuführen. Der Grund dafür wird später erläutert.

Die Prädikate *nrLookupIRRel/2* erstellen in der Lookup-Phase Beschreibungen der sich ergebenen Relationen. Dazu werden *irrel* (Intermediate Result Relation) Terme aufgebaut, die alle

relevanten Informationen über eine Relation beinhalten, die später benötigt werden. Genau dies sind die Zwischenbeschreibungen, die in den späteren Lookup-Phasen benötigt werden. Die *irrel* Terme treten dabei an die Stelle der Relationsnamen in den *rel(Rel, Var)* Termen des Optimierers. Bisher konnten dort nur Namen für Relationen auftreten. Ein *irrel* Term ist wie folgt aufgebaut:

```
1 irrel(+Type, +Stream, +TOP, +Card, +SizeTerm, +AttrList, +TypeSpec)
```

Die Bedeutung der einzelnen Terme ist:

- *Type*: Identifiziert den Fall, durch den der *irrel* Term erzeugt wurde. Mögliche Fälle: *relation, arel, nest, unnest, query.*
- *Stream*: Der Ausführungsplan, der die so beschriebene Relation erzeugt.
- *TOP (Transformation Operator)*: Falls ein *unnest* oder *nest* auf den Tupelstrom *Stream* anzuwenden ist.
- *Card*: Die Kardinalität dieser Relation.
- *SizeTerm*: Beinhaltet den *sizeTerm* zu dieser Relation, in dem die Tupelgrößen angegeben sind.
- *AttrList*: Eine Liste mit den Beschreibungen der Attribute.
- *TypeSpec*: Kann erweiterte Informationen beinhalten, die abhängig vom Wert *Type* sind.

Die Attributliste *AttrList* besteht aus einer Liste von Attributbeschreibungen der Form:

```
1 [DCAttr, Spelling, Case, DCFQN, Type, ARelDesc, SizeTerm]
```

Die Bedeutung der einzelnen Terme ist:

- *DCAttr*: Der komplett kleingeschriebene Name des Attributs.
- *Spelling*: Die exakte Schreibweise, bis auf den ersten Buchstaben. Dieser wird durch den Wert *Case* angegeben. Die Ursache hierfür ist, dass dies der späteren Verwendung in dem Lookup von Attributen entgegen kommt. Dieser Wert wird bei Umbenennungen, die auf der zugehörigen Relationenbeschreibung durchgeführt wird, angepasst, so dass dieser Name genau dem Namen entspricht, wie er aktuell im Tupelstrom *Stream* zu verwenden ist.
- *DCFQN*: Dies ist der voll qualifizierte kleingeschriebene Name des Attributs. Damit kann später noch nachvollzogen werden, aus welcher Relation das Attribut ursprüglich stammt. Der Wert wird mit *fqn(no)* besetzt, falls dieses Attribut nicht aus einer Relation stammt[1].
- *Type*: Der Datentyp des Attributs.
- *ARelDesc*: Für ein *arel* Attribut ist hier eine weitere Liste enthalten, welche die Subrelation beschreibt.
- *SizeTerm*: Beinhaltet den *sizeTerm* zu diesem Attribut.

[1] Präferiert wäre gewesen dieses Feld nicht mit einer gebundenen Variablen zu besetzen. Dieser Wert wurde ersatzweise gewählt, da dieser nicht mit einem gültigen voll qualifizierten Attributnamen kollidiert und insbesondere auch die *ground/1* Tests in der *Subqueries* Erweiterung besteht.

Das Lookup von *arel* Relationen und Subqueries stellt nun eine *irrel* Struktur nach dem Lookup zur Verfügung. Im Fall von *nest* oder *unnest* Anweisungen wird dieser *irrel* Term dann modifiziert, und zwar genauso, wie diese beiden Operatoren auch den Tupelstrom transformieren würden. Hier findet sich auch der Grund, warum es ein Prädikat *nrLookupIRRel(+InTerm, -OutTerm)* gibt, das für Relationen des Typs *rel* oder *nrel* Anwendung findet. Dieses Prädikat erstellt für normale Relationen einen *irrel* Term, der dann von dem *nrLookupIRRel/2* Prädikat der Fälle *nest* oder *unnest* weiter modifziziert werden kann. Durch dieses Verfahren ist auch eine ganze Sequenz von *nest* und *unnest* Anwendungen möglich, wie Beispiel 5.4 es zeigt.

Listing 5.4: Sequenz von Transformationen

```
1 select *
2 from orteh unnest(subrel) nest(bevth, subrel2) unnest(subrel2)
```

Damit stehen am Ende des Lookups der FROM-Klausel eine Reihe von *rel(Rel, Var)* Termen zur Verfügung, in denen *Rel* entweder der Name einer Relation oder ein *irrel* Term ist. Im Falle der *irrel* Terme steht die Beschreibung in dem Term selbst. Im Falle einer flachen Relation ist diese Beschreibung einfach zu beschaffen.

Beschrieben wurde nun, wie in der Lookup-Phase der FROM-Klausel Beschreibungen aufgebaut wurden, die die dort angegebenen Relationen beschreiben. Als nächstes wird das Lookup der Attribute in der SELECT- und WHERE-Klausel betrachtet. Die Lookup Phase der SELECT-Klausel verwendet zwei Prädikate *lookupAttr/2*, um Attribute zu erkennen. Analog dazu funktioniert dies für das Lookup von Attributen der WHERE-Klausel. Hier wird dies von den zwei Prädikaten *lookupPred/4* erledigt. Jeweils ein Prädikat der beiden Fälle erkennt Attribute nur an deren Namen. Die anderen beiden Prädikate erkennen Attribute, in denen Attributnamen in Verbindung mit einer Variablen der Form *Var:Attribute* verwendet werden. Genau diese vier Prädikate wurden neu implementiert, um Attribute aus den verschiedenen Relationen der FROM-Klausel zu erkennen.

Die Basisarbeit für das Lookup der Attribute leisten die Prädikate *findAttrLists(+Mode, ?SQID, ?Var, -RelT, -AttrList)*. Diese Prädikate ermöglichen den Zugriff auf die erreichbaren Relationen mit ihren Attributlisten. Attribute können dabei nur aus diesen Beschreibungen stammen.

Der Parameter *Mode* bezeichnet dabei die Lookup Phase, aus der das Prädikat aufgerufen wird, beispielsweise wird der Wert *attribute* für den Aufruf aus den *lookupAttr/2* Prädikaten heraus übergeben. Dies dient der Steuerung welche Beschreibungen in welcher Lookup Phase erreichbar sind. Beispielsweise werden damit keine Beschreibungen in das Lookup einer geschachtelten Anfrage der FROM-Klausel hineingereicht. Verwendet wird das *findAttrLists/5* Prädikat in den Prädikaten *lookupAttr/2* bzw. *lookupPred/4* in der Art, dass die Parameter *Mode* und *Var* besetzt werden. Der Prolog-Interpreter wird nun durch das Backtracking alle erreichbaren Relationen und Attributlisten zur Verfügung stellen, in denen das jeweilig betrachtete Attribut enthalten sein kann. Wenn der Wert *Var* nicht mit dem Term * unifiziert werden kann, handelt es sich um einen Attributzugriff der Form *Var:Attribute*. Da hierbei die mit *Var* referenzierte Relation eindeutig bestimmt ist, wird das Prädikat *findAttrLists/5* umittelbar die richtige Attributliste dem aufrufenden Prädikat zurückgeben. Ist der *Var* Parameter mit dem * unifizierbar, werden nacheinander alle in Frage kommenden Attributlisten per Backtracking ausgegeben, auch beispielsweise die Attributlisten der umgebenen Anfragen. Damit wurde also durch die Vorarbeiten des Lookups in der FROM-Klausel, das Lookup der Attribute im Kern auf ein einfaches Nachschauen in den Attributlisten reduziert.

Angemerkt sei hier noch, dass Attribute, die in der ORDERBY und GROUPBY-Klausel verwendet werden, keine eigenen Prädikate zum Lookup von Attributen verwenden. Dafür werden auch die *lookupAttr/2* Prädikate verwendet. Daher waren dort keine weiteren Änderungen notwendig.

5.2.3 Optimierer: Die Ausführungsplanerzeugung

Wie schon erwähnt wird in den Ausführungsplänen mit Strömen von Tupeln gearbeitet. In der Regel werden dazu Relationen in einen Tupelstrom per *feed* Operator eingespeist und am Ende der Tupelstrom wieder in eine Relation mit dem *consume* Operator eingelesen. Bei Anfragen mit Aggregatsfunktionen ohne GROUPBY-Klausel, wie z.B. der Form *select count(*) from ...*, ist das Result allerdings ein atomarer Wert. Dieser Sachverhalt wird aber erst im Unterabschnitt 5.2.5 auf Seite 38 thematisiert. Bereits erwähnt wurde weiterhin, dass viele Operatoren nur auf Tupelströmen arbeiten und in der Regel hier nur auf der obersten Ebene wirken, also insbesondere nicht *arel* Attribute rekursiv bearbeiten. Diese beiden Tatsachen, also die Verwendung von Tupelströmen und die Arbeitsweise vieler Operatoren auf Strömen, sind dafür verantwortlich, dass viele Funktionen für geschachtelte Relationen wie für normale Relationen funktionieren. Für die relevanten Operatoren, die zur Ausführung von SQL-Anfragen verwendet werden, die auf den Tupelströmen arbeiten, ist es dabei nicht von Bedeutung, woher dieser Tupelstrom kommt, d.h. ob dieser aus einer Relation selbst kommt, aus einem *arel* Attribut per *afeed*, oder aus einer anderen Abfrage, desssen Ergebnis als Strom eingespeist wird. Relationen, die durch *irrel* Terme beschrieben sind, unterscheiden sich zu den flachen Relationen durch die Tatsache, dass die Optimierung hier keine weiteren Objekte wie insbesondere die Indexstrukturen zur Optimierung heranzieht. Zum einen existieren für Relationen vom Typ *arel* keine verwendbaren Indexstrukturen, zum anderen aber wurde für Relationen, die durch geschachtelte Anfragen definiert sind, dies nicht weiter vertieft. Möglicherweise bestehen hier noch sinnvolle Möglichkeiten durch das Erstellen von temporären Indexstrukturen eine Optimierung zu erreichen.

Die wenigen Änderungen in der Generierung der Ausführungspläne waren daher einerseits auf das richtige Anwenden der *afeed* und *aconsume* Operatoren beschränkt. Andererseits war dann noch das Einbringen der Ausführungspläne von *irrel* Zwischenergebnisströmen in die Ausführungspläne der umgebenen Anfrage zu realisieren. An zwei Stellen waren dazu weitere Prädikate im Optimierer zu integrieren. Als erstes wurden neue Übersetzungsregeln mit dem *=>/2* Operator für Anfragen die eine WHERE-Bedingung enthalten implementiert. Als zweites folgten die *makeStream(+Rel, -Stream)* Prädikate für alle anderen Fälle (also SQL-Anfragen, die keine WHERE-Bedingungen besitzen[2]). Da ein *irrel* Term aus der Lookup-Phase bereits den Ausführungsplan beinhaltet, der den Tuplestrom erzeugt, beschränkt sich dies auf wenige Schritte. Als erstes wird der Plan aus der *irrel* Struktur extrahiert. Dieser Plan wird dann, falls vorhanden, um Transformationen erweitert, die in der Lookup-Phase noch nicht angewendet wurden (*unnest* oder *nest* Anweisungen). Als letzten und dritten Schritt erfolgt die Anwendung der Operatoren *rename* und *project*. Die Anwendung des *project* Operators ist dabei erst hier möglich, da bei der Erstellung der *irrel* Struktur die dafür notwendigen Informationen noch gar nicht vorlagen. Anschließend ist dieser so konstruierte Plan, der einen Tupelstrom erzeugt, nur noch dem aufrufenden Prädikat zurückzugeben.

Angemerkt sei, dass, wie in Unterabschnitt 2.3.3 auf Seite 12 beschrieben, auch hier die Optimierung weiterhin mit der Strategie aus der Subqueries-Erweiterung erfolgt. Es wird daher jedes *select ... from ...* Konstrukt einer Anfrage unabhängig, also ohne Betrachtung der geschachtelten

[2] Wozu auch der Sonderfall der leeren WHERE-Bedingung in Form der leeren Liste [] gehört.

Anfragen, optimiert. Für die hier implementierten Subqueries, also die der SELECT- oder FROM-Klausel, ist die Reihenfolge anders. Hier findet die Optimierung schon in der Lookup Phase statt, anstatt erst bei der Erzeugung der SECONDO-Textsyntax durch die *plan_to_atom/2* Prädikate, wie es durch die Subqueries-Erweiterung erfolgt. Dabei ist es in beiden Fällen vom Design her nicht sehr elegant, dass die Ausführungspläne der eine Ebene tiefer liegenden Subqueries in einem Fall nach der Planerzeugungsphase, und im anderen Fall vor der Planerzeugungsphase erzeugt werden. In beiden Fällen erfolgt dies also nicht in der dafür eigentlich vorgesehen Phase der Planerzeugung. Dem entgegen zu halten ist, dass für ein *select ... from ...* Konstrukt, also wenn nicht die komplette Anfrage betrachtet wird, die Reihenfolge lookup, Planerzeugung, SECONDO-Textsyntax-Erzeugung sehr wohl in dieser Reihenfolge abläuft.

Der Grund, die Planerzeugung für *irrel* Terme in die Lookup Phase der direkt umgebenen Anfrage zu integrieren, ist, dass in diesem Fall die *irrel* Terme nicht mehr geändert werden müssen und keine zusätzliche Phase zwischen dem Lookup und der Planerzeugung der tiefer liegenden Subqueries eingeführt werden musste, um die Ausführungspläne der Subqueries vor der eigentlichen Planerzeugung bereitzustellen. Die Möglichkeit, die Ausführungspläne wie bei den Subqueries-Erweiterung erst in der SECONDO-Textsyntax-Erzeugung zu ermitteln, war hier nicht mehr möglich. Die Ausführungspläne der FROM-Klausel für Subqueries werden nämlich in der Planerzeugunsphase benötigt, insbesondere werden hier auch die Kardinalitäten, Tupelbreiten und Selektivitäten benötigt, um die Optimierung durchzuführen. Der Grund, warum die eigentliche Erzeugung des Plans für geschachtelte Anfragen der FROM-Klausel nicht strikt in der Phase der Planerzeugung erledigt wird, ist schnell erklärt. Die eigentliche Optimierung auf Basis des *predicate order graph* kann nicht rekursiv erfolgen, da aber genau dies der Fall für Subqueries der FROM-Klausel wäre, wurde die Entscheidung getroffen, dies in der Phase des Lookups mitzuerledigen.

An dieser Stelle soll eine weitere Beispielanfrage inklusive des dadurch erzeugten Ausführungsplans in der SECONDO-Textsyntax gezeigt werden. Der Anfrageoptimierer wird zu der Anfrage 5.5 den Ausführungsplan[3] erzeugen, der in Listing 5.6 auf Seite 37 zu sehen ist. Die Anfrage besteht in Zeile 1 bis 6 aus der schon bekannten Anfrage 4.7 auf Seite 22. Zusätzlich wurde in Zeile 7 bis 9 noch eine WHERE-Bedingung aufgenommen um auch den Fall einmal zu demonstrieren, wie die Subqueries-Erweiterung die geschachtelten Anfragen in einen Ausführungsplan umsetzt. In dem Ausführungsplan selbst wird damit begonnen mit dem Operator *feed* die Relation *Orte* auszulesen indem ein Tupelstrom erzeugt wird. In den Zeilen 2 bis 5 wird die Struktur der Ergebnisrelation aufgebaut[4]. Zeile 6 bis 13 ist die Umsetzung der geschachtelten Anfrage aus der WHERE-Bedingung durch die angepasste Subqueries-Erweiterung. Dazu wird eine Funktion durch das *fun* Schlüsselwort erzeugt, die an den *filter* Operator abgegeben wird. Die Definition der Funktion in Zeile 7 bis 11 bewirkt, dass für jedes Tupel des Tupelstroms die geschachtelte Relation *SubRel_o* gelesen wird (Zeile 8). In den Zeilen 8 bis 9 wird dann ermittelt, wie viele Tupel in dieser Relation vorhanden sind. Hierbei bestehen allerdings noch Verbesserungsmöglichkeiten in der Optimierung. So ist es hier nicht nötig alle Tupel zu zählen. Es reicht, sobald ein Tupel gefunden wird, der an den *count* Operator abgegeben wird, da nun die Bedingung *0 < ...* nicht mehr zu *false* ausgewertet werden kann.

[3] *predinfo* Terme wurden hier weggelassen, da diese nicht das Ergebnis beeinflussen.

[4] Der *project* Operator ist hier eigentlich nicht notwendig, da nur diese Attribute in dem Tupelstrom vorhanden sind. Eine derartige Optimierung wird im Optimierer allerdings nicht vorgenommen.

Listing 5.5: Anfrage

```
select *
from   (
         select [bevt div 100 as bevth,
                bevt, ort, kennzeichen, vorwahl]
         from   orte
       ) nest(bevth, subRel) as o
where  exists(select *
              from o:subrel
              where ort="Berlin")
```

Listing 5.6: Ausführungsplan zur Anfrage 5.5

```
Orte feed
extend[Bevth: (.BevT div 100)]
project[Bevth, BevT, Ort, Kennzeichen, Vorwahl]
sortby[Bevth]
nest[Bevth; SubRel]{o}
filter[fun(alias13: TUPLE) (
    0 <
      attr(alias13, SubRel_o) afeed
      project[Ort_o]
      filter[(.Ort_o = "Berlin")]
      count
  )
]
consume
```

5.2.4 Kostenfunktionen

Im Unterabschnitt 5.2.2 auf Seite 32 wurde erklärt wie aus den komplexeren Relationen der FROM-Klausel der zugehörige Ausführungsplan, der den Tupelstrom erzeugt und im *irrel* Term zur Verfügung gestellt wird, in den Plan des direkt umgebenen *select ... from ...*-Konstruktes bei der Optimierung eingebracht wird. Dieses Einbringen ganzer Ausführungspläne führt dazu, dass für Operatoren Kosten berechnet werden müssen, die vorher in der Optimierung nicht auftauchen konnten. Insbesondere tritt hiermit auch der Fall ein, dass Kosten für die Operatoren *unnest* und *nest* ermittelt werden müssen. Da der Aspekt der eigentlichen Optimierung von geschachtelten Relationen hier nicht mit behandelt werden konnte, werden die Kosten für diese Operatoren nicht betrachtet, sondern setzen sich aus den Kosten der Operatoren zusammen, für die Kostenfunktionen vorhanden sind. Relevant wäre dies für die Ermittlung der Gesamtkosten. Im Ergebnis der Planerzeugung könnte dies allerdings auch zu einer Änderung des optimalen Plans führen. Ein anderer Pfad würde dabei nicht aufgrund alternativer Übersetzungregeln gewählt, da es bisher nur eine Übersetzungsregel für *irrel* Terme gibt. Aber der optimale Pfad im *predicate order graph* könnte aufgrund der Abweichungen zwischen den wirklichen und den geschätzen Kosten ein anderer sein.

In der aktuellen Version vernachlässigt der Optimierer auch die Kosten für den *extend* Operator. In der Regel konnte davon ausgegangen werden, dass sich die Kosten hier im Rahmen halten, also

keine komplett andere Betrachtung der Kosten nötig wäre. Allerdings gilt dies bei der Verwendung der geschachtelten Relationen, oder allgemeiner, der Verwendung von geschachtelten Anfragen in der SELECT-Klausel nun nicht mehr. Wenn beispielsweise in den geschachtelten Anfragen komplexe Join-Operationen durchgeführt werden, kann in den wegelassenen Kosten der *extend*-Operation der Hauptanteil der Kosten verursacht werden. Angemerkt sei noch, dass diese Erweiterung der geschachtelten Relationen bisher allein mit dem Standardkostenmodell zusammenarbeitet.

5.2.5 Ergebnistypen geschachtelter Anfragen der SELECT- und FROM-Klausel

Im Unterabschnitt 5.2.1 auf Seite 29 über die Erweiterung der geschachtelten Relationen wurde bereits darauf eingangen, dass bisher geschachtelte Anfragen in der SELECT-Klausel atomare Werte als Ergebnis liefern. Neu ist nun bei der Verwendung der geschachtelten Relationen, dass nicht mehr nur atomare Werte erzeugt werden, sondern gerade Attribute, die eine Relation enthalten, also vom Typ *arel* sind.

Im Secondo DBMS wird für eine Anfrage, wie Beispiel 5.7 zeigt, der Ausführungsplan 5.8 erstellt.

Listing 5.7: Count-Anfrage ohne GROUPBY-Klausel

```
1 select count(*)
2 from   orte
```

Listing 5.8: Einfacher Ausführungsplan

```
1 Orte feed count
```

Der *count* Operator im Secondo DBMS liefert allerdings als Ergebnis einen Wert vom Typ *int*, also einen atomaren Wert und keine Relation. Auf der Kommandozeilenebene wird in der Ausgabe der Wert ausgegeben und mit diesem Ergebnis ist man in der Regel zufrieden. Der einzige, sichtbare Unterschied zu demselben Ergebnis in Form einer Relation besteht allein darin, dass für den Wert kein Attributname ausgegeben wird. Wird eine geschachtelte Anfrage aber in der SELECT- bzw. FROM-Klausel verwendet gibt es die beiden Fälle 5.9 und 5.10.

Listing 5.9: Subquery in der SELECT-Klausel mit einem atomaren Ergebnis

```
1 select (select count(*)
2         from   orte) as anzahl
3 from   orte
4 first  1
```

Anfragen der Art aus 5.9 führen üblicherweise in anderen DBMS, wie beispielsweie PostgreSQL oder dem IBM Informix Dynamic Server, dazu dass das Ergebnis der Anfrage ein atomarer Wert ist. Auch war dies bisher im Secondo DBMS durch die Subqueries-Erweiterung der Fall. Mit der Einführung der geschachtelten Relationen ist hier aber eine Definition nötig, wann ein atomarer Wert aus einer geschachtelten Anfrage entstammt und wann eine geschachtelte Relation konstruiert

Listing 5.10: Subquery in der FROM-Klausel mit einem atomaren Ergebnis

```
1 select *
2 from   (select count(*)
3         from   orte)
```

wird. Kritischer ist diese Problematik in der Anfrage 5.10. Hier wird als Datenbasis in der FROM-Klausel eine geschachtelte Anfrage angegeben, die ein atomares Ergebnis erzeugt; dies ist aber nicht möglich, da hier nur Relationen oder Anfragen, die einen Tupelstrom erzeugen, erscheinen dürfen. Diese Situation wurde nun wie folgt gelöst:

1. Anfragen der Formen
   - *select OP(_) from from-list*
   - *select OP(_) from from-list where where-list*

   mit $OP \in \{count, min, max, sum, avg, extract\}$ in der SELECT-Klausel liefern als Ergebnis einen atomaren Wert.
2. Alle anderen Anfragen der SELECT-Klausel erzeugen eine Relation vom Typ *arel*.
3. Anfragen der oberen Formen mit $OP = count$ innerhalb der FROM-Klausel erzeugen einen Tupelstrom mit einem Tupel und einem Attribut mit dem festen Namen *Count*, da für *count*-Anfragen kein Name mit *as* festgelegt werden kann.
4. Anfragen der oberen Formen mit $OP \in \{min, max, sum, avg, extract\}$ innerhalb der FROM-Klausel erzeugen einen Tupelstrom mit einem Tuple und einem Attribut mit dem festen Namen *Value*, da für diese Anfragen kein Name mit *as* festgelegt werden kann.
5. Alle anderen Anfragen der FROM-Klausel erzeugen einen Tupelstrom.

Zu beachten ist hier, dass Anfragen mit einer GROUPBY-Klausel *nicht* in die unter Punkt 1 genannten Anfrageformen fallen. Diese Anfragen erzeugen sowieso immer einen Tupelstrom, bzw. eine Relation.

Teilweise[5] kann das Ergebnis, das durch eine Anfrage erzeugt wird, auch durch das Umschreiben der Anfrage geändert werden. Würde man in der Anfrage 5.9 an einem Ergebnis interessiert sein, das eine Relation liefert, kann dies erzwungen werden, indem die Anfrage umgeschrieben wird. Beispiel 5.11 zeigt so einen Fall. Der zusätzliche umgebene *select * from (...)* Ausdruck erzwingt hier die Erzeugung einer Relation für ein atomares Ergebnis.

Andere DBMS, wie z.B. PostgreSQL, unterstützen auch geschachtelte Anfragen, die einen atomaren Wert liefern, in denen ohne weitere Kenntnisse der Datenbank der Anfrage aber nicht angesehen werden kann, ob das Ergebnis atomar ist. Beispiel 5.12 zeigt so einen Fall. Erst nach der Betrachtung der Daten, hier der Beispielrelation *Orte* (Tabelle 1.1 auf Seite 1), kann festgestellt werden, ob das Ergebnis auch als atomares Ergebnis interpretiert werden könnte.

[5] Durch die zusätzliche Subquery in der Anfrage sind Attribute ggf. nicht mehr erreichbar (Vgl. Unterabschnitt 5.2.1 auf Seite 29).

Listing 5.11: Relationenwertiges Ergebnis einer geschachtelten Anfrage erzwingen

```
select (select *
        from (select count(*)
              from orte)) as anzahl
from   orte
first  1
```

Listing 5.12: Beispiel einer geschachtelten Anfrage mit einer Anfrage, die ein möglicherweise atomares Ergebnis liefert.

```
select (select *
        from (select ort
              from   orte
              where  bevt>3000)) as subrel
from   orte as o
first  1
```

Hier ist das Ergebnis der innersten geschachtelten Anfrage ein Tupel mit einem Attribut. Das DMBS PostgreSQL, welches keine geschachtelten Relationen unterstützt, wird für so einen Fall einen Laufzeitfehler anzeigen, wenn die geschachtelte Anfrage nicht genau ein Tupel als Ergebnis liefert. Die in dieser Arbeit erfolgte Implementierung für das Secondo DMBS erzeugt für solche Fälle aber immer eine geschachtelte Relation als Ergebnis. Bei Bedarf kann über den *extract* Operator aber ein atomares Ergebnis erzwungen werden, wie im Beispiel 5.13 gezeigt wird. In dem Fall sollte man sich aber vergewissern, dass eine Anfrage nur ein Tupel liefert oder sich zumindest den Auswirkungen einer solchen Änderung bewusst ist. In dem Fall, dass hier mehrere Tupel aus der Anfrage zurückgegeben werden, wird nämlich keine Fehlermeldung zur Laufzeit erzeugt.

Listing 5.13: Beispiel einer geschachtelten Anfrage mit einer Anfrage, die ein atomares Ergebnis erzwingt.

```
select (select extract(ort)
        from  (select  ort
               from    orte
               where   bevt>3000)) as atomicresult
from   orte as o
first  1
```

5.2.6 Erweiterung: mpoint-Entschachtelungen

Im Zuge der Implementierung kam eine weitere Anforderung auf, die eine ähnliche Funktionsweise wie das *unnest* darstellt. Eine Komponente des Teils aus Secondo, der mit bewegten Objekten arbeitet, ist der Datentyp *moving point (mpoint)* [AGB06]. Dieser Datentyp ist dabei nicht atomar und beinhaltet eine variable Anzahl von zeitabhängigen Positionen. Diese zeitabhängigen Positionen sind vom Typ *upoint*.

Auf der Ausführungsebene kann ein *mpoint* Objekt mit dem Operator *units* in die einzelnen Werte vom Typ *upoint* zerlegt werden. Im Gegensatz zu einer geschachtelten Relation, die Tupel enthält, enthält ein *mpoint* Objekt ein oder mehrere *upoint* Objekte. Die Idee ist nun, die *unnest* Funktion des SECONDO SQL-Dialekts[6] nicht nur auf Attribute vom Typ *arel*, sondern auch für Attribute vom Typ *mpoint* verfügbar zu machen. Implementiert ist dies wie folgt: Das Lookup-Prädikat für *unnest* Terme erkennt dabei den Typ des im Parameter angegebenen Attributs auf dem die *unnest* Anweisung angewendet wird. Beim Anwenden der Transformationen wird die Änderung der Attributlisten in der *irrel* Struktur entsprechend verändert, analog wie dies beim *unnest* von *arel* Attributen erfolgt (allerdings in diesem Fall wesentlich vereinfacht). Die entsprechende Planerzeugung für diese Transformation erfolgt anschließend durch das Hinzufügen eines weiteren *addTransformationOperator(+StreamIn, +TOP, -StreamOut)* Prädikates.

Wird die Relation *Trains* mit einem Attribut *trip* vom Typ *mpoint* in den folgenden Beispielen[7] zugrunde gelegt, dann ist 5.14 ein Beispiel für eine Entschachtelung eines *mpoint* Attributs.

Listing 5.14: Die *unnest* Anweisung für ein *mpoint* Attribut

```
1 select *
2 from   trains unnest(trip)
```

Durch die neue Transformationsregel erzeugt der Optimierer den Ausführungplan, der in 5.15 zu sehen ist. Hier wird ein neues Attribut *Tempattr1* zur temporären Aufnahme der Zerlegung des *mpoint* Attributs in seine *upoint* Objekte erzeugt. Für jedes dieser *upoint* Werte des *mpoint* Attributs wird dann ein neues Tupel erzeugt, das aus dem *upoint* Wert selbst und den übrigen Attributen des Tupels besteht.

Listing 5.15: Ausführungsplan zum *unnest* auf einem *mpoint* Attribut

```
1 Trains feed
2 extendstream[Tempattr1: units(.Trip)]
3 remove[Trip]
4 renameattr[Trip: Tempattr1]
5 consume
```

In dem Ausführungsplan ist eine Besonderheit zu erkennen. Im Gegensatz zu den *arel* Attributen, besitzt ein *mpoint* keine eigene Strukturbeschreibung, in dem Attributnamen vorkommen. Daher wird hier der Name des ursprünglichen *mpoint* Attributs durch Umbenennungen der Attribute beibehalten. Eine anschließende Änderung des Attributenamens ist dabei natürlich weiterhin mit dem *as* Operator in der SELECT-Klausel möglich.

5.3 Test, Robustheit und Erweiterbarkeit

Die Komplexität bei der Implementierung zur Behandlung der geschachtelten Relationen hatte seinen Ursprung in erster Linie darin, dass Anpassungen an vielen verschiedenen Stellen des Optimierers nötig waren. Es existierten hier nur wenig Beschränkungen auf einen Teilbereich des

[6] Hier nicht zu verwechseln mit dem *unnest* Operator der *NestedRelationAlgebra*.

[7] Basis für dieses Beispiel ist die Relation *Trains* der Datenbank *berlintest*, die im SECONDO DBMS enthalten ist.

Optimierers, da im Prinzip überall, wo Attribute auftreten, der Fall eintreten kann, dass es sich um ein relationenwertiges Attribut handelt. Möglich ist hier sogar, dass ein entsprechendes *arel* Attribut bei der Ausführung der Anfrage noch gar nicht existierte, sondern dieses durch eine geschachtelte Anfrage erst aufgebaut wurde. Insgesamt bestand hier überhaupt nicht die Möglichkeit alle Kombinationen zu prüfen, da die Anzahl möglicher Kombinationen der verschiedensten Fälle viel zu groß ist. Im Zuge der Implementierung wurden allerdings zum Test der Erweiterungen mehr als 200 Testfälle geschrieben, um die relevantesten Konstellationen von Anfragen abzudecken. Erfahrungsgemäß zeigten diese Testfälle in einem Testlauf auch sofort Probleme auf, wenn Änderungen am Quellcode vorgenommen wurden, die nicht richtig waren.

In dem Quellcode dieser Erweiterung selbst ist die Strategie, dass hier Prädikate die nicht beweisbar sind, von denen erwartet wird, dass diese beweisbar sind, unmittelbar eine Ausnahme (Exception) auslösen. Ursache hierfür ist, dass die Fehlersuche im Optimierer, insbesondere bei der Verwendung von geschachtelten Anfragen, je nach Fehlerart sehr aufwendig sein kann. Der Analysevorgang kann sich dann auch noch wesentlich aufwendiger gestalten, wenn sich Fehler wesentlich später im Optimierungsvorgang durch das Backtracking auswirken. Um hier nach Möglichkeit aufwendiges Fehlersuchen zu vermeiden, wird möglichst frühzeitig eine Ausnahme ausgelöst: nämlich, wenn ein Prädikat nicht beweisbar war, von dem aber erwartet wurde beweisbar zu sein. Die vielen Prädikate, deren einziger Zweck es ist, die Ausnahmen auszulösen, machen den Quellcode zwar nicht eleganter, dafür sind sie eine wesentliche Hilfe in der Fehleranalyse.

Mögliche Erweiterungen sind insbesondere bei den *nest* und *unnest* Operationen zu erwarten. Beispielsweise könnte noch implementiert werden, die *unnest* Funktion des SQL-Dialekts auf Arrays anzuwenden oder auch die Möglichkeit die *nest* Funktionalität der *mpoint* Objekte zu implementieren. Solche Änderungen können leicht in die derzeitige Implementierung eingebracht werden. In der Regel sind dafür die Schritte, die in Unterabschnitt 5.2.6 auf Seite 40 angesprochen wurden, durchzuführen.

Kapitel 6

Fazit

Erreicht wurde durch die Implementierung der beschriebenen Änderungen, dass aus Sicht des Anwenders grundlegende Möglichkeiten bestehen, mit Hilfe des SECONDO SQL-Dialektes, mit geschachtelten Relationen arbeiten zu können. Wichtigstes Mittel dazu sind die geschachtelten Anfragen, mit denen neue geschachtelte Relationen aufgebaut oder vorhandene geschachtelte Relationen verändert werden können. Transformationen der Struktur können mit dem *unnest* oder *nest* Anweisungen erreicht werden. Damit die Ausdruckskraft für die Strukturänderung nicht auf eine Schachtelungsebene beschränkt ist, wurde die FROM-Klausel erweitert, so dass hier auch komplexe geschachtelte Anfragen erlaubt sind. Auch wurden im Kapitel Entwurf (Kapitel 4 auf Seite 19) viele Beispiele gezeigt, wie mit geschachtelten Relationen gearbeitet werden kann.

Insbesondere auf Seiten der Optimierung sind allerdings Punkte offen geblieben. Größter Teilpunkt sind hier die Selektivitäten für WHERE-Bedingungen, wenn Relationen durch *irrel* Terme beschrieben werden. Aber auch die veranschlagten Kardinalitäten der geschachtelten Relationen bedürfen noch einer weiteren Analyse. Insbesondere wurde die implementierte Strategie der Durchschnittswerte nicht evaluiert. Ein weiterer Punkt sind die Kostenfunktionen. Für eine Reihe von Operatoren werden bisher die Kosten mit 0 veranschlagt, da für die Operatoren noch keine Kostenfunktionen bekannt sind. Abgesehen vom zeitlichen Faktor ist dabei aufgrund der fehlenden Kostenfunktion fraglich, wie sinnvoll eine Einbeziehung der Optimierung gewesen wäre. Auf der Ebene des Anwenders hingegen fehlen noch die wichtigen Funktionen, Tupel in vorhandenen Relationen vom Typ *nrel* und *arel* einzufügen, modifizieren oder löschen zu können.

Teil II

Erweiterung des Secondo Optimierers: Speicherzuteilung an Operatoren

Kapitel 7

Stand der Technik

In dem Artikel [ND98] wurden verschiedene Algorithmen zur Optimierrung der Speicherverteilung auf die Operatoren in einem Ausführungsplan präsentiert. Die Kostenfunktionen konnten dabei entweder konstant oder linear sein. Die Durchführung der Arbeitsspeicheroptimierung wurde hier nach der Ermittlung des besten Ausführungsplans durchgeführt, also noch nicht in der eigentlichen Phase der Planermittlung mit einbezogen. Dementsprechend waren diese Algorithmen nicht in der Lage, den besten Ausführungsplan auch unter Berücksichtigung der Ressource Arbeitsspeicher zu ermitteln. Betrachtet wurden dort drei Algorithmen: ein einfach zu implementierender Algorithmus, der der derzeitigen Strategie im Secondo DBMS ähnelt, eine Heuristik, die schnell gute Ausführungspläne ermitteln soll und letztlich ein Algorithmus zur optimalen Lösung auf Basis der linearen Optimierung. In der Arbeit wurde dargestellt, dass sich die Laufzeit bei Änderung des Arbeitsspeichers linear verändert. Dort wurde allerdings nur der *HashJoin* Operator des Paradise DBMS betrachtet. Im Secondo DBMS sind aber auch Operatoren zu berücksichtigen die nichtlineare Kostenfunktionen in Abhängigkeit vom Arbeitsspeicher besitzen [Nid12]. Damit können die dort vorgeschlagenen Algorithmen so nicht auf das Secondo DBMS übertragen werden.

Der Artikel [HSS00] geht einen Schritt weiter als der Artikel [ND98] und betrachtet nicht nur die Optimierung eines Ausführungsplans, der vom Optimierer schon erzeugt wurde. Hier wird auch der Arbeitsspeicher schon bei der Betrachtung aller möglichen Ausführungspläne betrachtet. Beide Strategien wurden in dem Volcano DBMS implementiert. Nach den Autoren war dies die erste ihnen bekannte Arbeit, die auch die Ressource Arbeitsspeicher bereits bei der Betrachtung aller bekannten Ausführungspläne mit einbezieht. Im Gegensatz zu [ND98] werden hier aber auch nichtlineare Kostenfunktionen betrachtet. Allerdings werden diese nicht direkt in den implementierten Algorithmen verwendet, sondern auf Basis von Funktionen, die abschnittsweise linear sind. Das Ermitteln des besten Ausführungsplans im Volcano DBMS wird dabei auf einem spezielleren, gerichteten, azyklischen Graphen vorgenommen, allerdings unterscheidet sich dieser von dem POG des Secondo DBMS, weshalb die Strategie auch so nicht zu übernehmen ist. Für die in dem Artikel durchgeführten Tests ermittelten die Autoren, dass die Optimierung in einer Phase ca. 10 mal langsamer ist, als wenn nur eine Speicheroptimierung der Operatoren für einen gegebenen Ausführungsplan durchgeführt wird.

Kapitel 8

Entwurf

Die bisher im SECONDO DBMS[1] implementierte Strategie [SMEM11], den Operatoren eines Ausführungsplans Arbeitsspeicher zuzuweisen, ist eher einfach gehalten. Die Strategie ist, dass den Operatoren in einem Ausführungsplan manuell Arbeitsspeicher in MiB[2] Schritten zugewiesen werden kann. Die Zuweisung selbst erfolgt, indem hinter dem Operator das Schlüsselwort *memory*, gefolgt von der Menge an Arbeitsspeicher in MiB, in geschweiften Klammern angegeben wird. Folgender Ausführungsplan zeigt dabei für den Operator *sortmergejoin* eine Zuweisung von 512 MiB Arbeitsspeicher:

```
1 Roads feed{r}
2 Buildings feed{b}
3 sortmergejoin[No_r, No_b]{memory 512}
4 consume
```

Arbeitsspeicher kann in der Form aber nicht allen Operatoren zugewiesen werden. Dazu muss ein Operator eine dynamisch zugewiesene Menge an Arbeitsspeicher in seiner Implementierung unterstützen und der Operator muss dem SECONDO-System anzeigen dass dieser Arbeitsspeicher verwenden möchte[3]. Im Folgenden wird für solche Operatoren der Begriff *speicherverwendende Operatoren* verwendet.

Wird ein Ausführungsplan ausgeführt, dann wird nicht vergebener Arbeitsspeicher gleichmäßig auf die restlichen arbeitsspeicherverwendenden Operatoren aufgeteilt. Mindestens bekommt dabei ein Operator 16 MiB Arbeitsspeicher zugewiesen (*MinMemoryPerOperator*), auch wenn damit der maximal zur Verfügung stehende Arbeitsspeicher überschritten wird oder der Operator soviel Arbeitsspeicher gar nicht benötigt.

Problematisch war hier, sofern keine manuelle Zuteilung vorgenommen wurde, zum einen dass manchen Operatoren zu wenig Arbeitsspeicher zugewiesen wurde, während andere Operatoren mehr als nötigen Arbeitsspeicher bekommen haben. Die Strategie reagiert hier also nicht auf die Informationen, die der Optimierer zur Anfrageoptimierung schon zur Verfügung hat.

Der für diese Arbeit ausschlaggebende Nachteil ist, dass weder darauf Rücksicht genommen wurde, wieviel Arbeitsspeicher ein Operator genau für die Ausführung benötigt und wie sich der zugewiesene Arbeitsspeicher auf die Kosten auswirkt, wenn einem der Operatoren mehr oder weniger Arbeitsspeicher zugewiesen wird. Durch eine *geschicktere* Zuteilung des Arbeitsspeichers an die Operatoren kann für viele Fälle ein Plan existieren, der *besser* im Sinne seines Laufzeitverhaltens (gleich den Kosten im SECONDO DMBS) ist als der Plan, der momentan vom Optimierer ausge-

[1] der Version 3.2.1

[2] Einheiten zur Basis 2 sind nach der IEC 60027-2 Norm angegeben, also z.B. $1\,MiB = 2^{20}$ aber $1\,MB = 10^6$.

[3] Genauer sind dies Operatoren aus Algebra-Implementierungen, welche über die Funktion *SetUsesMemory()* den SECONDO Kernel darüber informieren, dass diese zusätzlichen Arbeitsspeicher verwenden wollen [SMEM11]

wählt wird. Was denn überhaupt eine geschicktere Zuteilung ist, und wie diese erreicht werden kann, sind die zentralen Fragen dieses Kapitels.

Eine effiziente Nutzung und damit eine bessere Zuteilung des Arbeitsspeichers ermöglicht für viele Algorithmen einen Leistungsgewinn. Ursache für den möglichen Leistungsgewinn sind dabei die unterschiedlichen Leistungen in den Ebenen der Speicherhierarchie, insbesondere zwischen dem Arbeitsspeicher eines Rechnersystems und dem Sekundärspeicher. Dieser ist in der Regel für DMBS eine Festplatte. Ausführliche Leistungsvergleiche finden sich dazu beispielsweise in [Jac09]. Demnach liegen die Zugriffzeiten für Festplatten üblicherweise im Millisekundenbereich und die für Arbeitsspeicher im Nanosekundenbereich[4]. Bei den Transferraten sind die Unterschiede nicht mehr so extrem, variieren aber stark. Hier kann zumindest von einem Faktor von 10 ausgegangen werden.

Zwei Tatsachen führen damit dazu, dass eine Optimierung der Speicherzuteilung der Operatoren sich in geringeren Gesamtkosten niederschlagen kann:

- Die Datenmengen die durch Anfragen verarbeitet werden, sind größer als der verfügbare Arbeitsspeicher, den die Operatoren benötigen würden, um ihre Arbeit ohne Verwendung des Sekundärspeichers auszuführen.
- Die Leistungen der Sekundärspeicher sind wesentlich geringer als die Leistungen des Arbeitsspeichers. Dies resultiert in eine Abhängigkeit der Kosten vom verwendeten Arbeitsspeicher.

8.1 Problemdefinition

Ein erstes Ziel ist, eine Möglichkeit zu finden, die Aufteilung des Arbeitsspeichers, der den Operatoren eines *gegebenen* Ausführungsplans zur Verfügung steht, so zu verbessern, dass die Gesamtkosten des Ausführungsplans minimiert werden. Gefragt ist konkret die Belegung der Speicherzuweisungen $m_1, ..., m_n$ für n Arbeitsspeicher verwendende Operatoren von m Gesamtoperatoren eines Ausführungsplans, so dass die Gesamtkostenfunktion $f(m_1, ..., m_n)$ des Ausführungsplans minimiert wird. Im folgenden wird diese nachträgliche Optimierung eines Ausführungsplans als Zwei-Phasen-Optimierung bezeichnet. In der ersten Phase wird der Ausführungsplan ohne Betrachtung des Arbeitsspeichers erstellt und erst in der zweiten Phase wird versucht diesen Ausführungsplan zu verbessern. Es kann hierbei also der Fall eintreten dass ein anderer Ausführungsplan nach der Arbeitsspeicheroptimierung besser gewesen wäre.

Das zweite Ziel ist die einphasige Optimierung. Hier geht es darum, zu evaluieren, wie unter all den möglichen Pfaden des Graphen der Kostenkanten auf Basis des *predicate order graphs* der Pfad, und damit der Ausführungsplan ermittelt werden kann, der auch optimal ist, wenn hier schon die Ressource Arbeitsspeicher berücksichtigt wird. Hierzu wurde in [ND98] festgestellt, dass dieses Problem, in dem das Ressourcenmanagement wie der Arbeitsspeicher schon einbezogen wird, in einer „Explosion des Suchraums“ resultiert. Eine kostengünstige, optimale Lösung für das Problem der einphasigen Optimierung ist daher nicht zu erwarten.

In dem Graphen der Kostenkanten auf Basis des POG ist es allerdings möglich, dass mehrere Pfade mit gleichen Kosten existieren. In dem Fall reicht es einen dieser Pfade zu ermitteln, da sie alle dem Kriterium genügen, einen optimalen Pfad darzustellen. Hier wird also eine rein kostenorientierte Bewertung vorgenommen. Es wird nicht nach einem Plan gesucht, der unter verschiedenen

[4] $0.000001ms = 1ns$

kostenidentischen Pfaden auch noch den geringsten Bedarf an Arbeitsspeicher besitzt oder auch weitere Kriterien berücksichtigt. Wenn ein Datenbanksystem parallele Anfragen ausführt, könnte es durchaus interessant sein, einen optimalen Pfad zu finden, der möglichst wenig Arbeitsspeicher benötigt. Allerdings ist hier fraglich, ob überhaupt oft genug kostenidentische Pfade existieren, um einen solchen Aufwand zu rechtfertigen. Allerdings wären hier ausgefeiltere Methoden möglich. Es könnte dann beispielsweise eine Abweichung in den Kosten um x Prozent toleriert werden mit dem Ziel, einen Pfad zu finden, der unwesentlich schlechtere Kosten aufweist, aber wesentlich weniger Arbeitsspeicher benötigt. Betrachtet wurde eine solche Erweiterung in dieser Arbeit aber nicht weiter, da sie den Rahmen sprengen würde.

8.2 Ein Beispiel

Ein Beispiel für eine lineare Kostenfunktion in Abhängigkeit vom Arbeitsspeicher wäre die Funktion $f : [16, 512] \to \mathbb{R}$ aus (8.1). Hier und für die folgenden Beispiele dieses Abschnitts wird dabei angenommen, dass die Operatoren jeweils bis 512 MiB Arbeitsspeicher sinnvoll verwenden können und auch maximal 512 MiB an Arbeitsspeicher zur Verfügung steht. Da der Operator hier nach Annahme nur 512 MiB an Arbeitsspeicher benötigt, ist ab dieser Schwelle für den Operator mit mehr Arbeitsspeicher keine Kostenverbesserung mehr zu erreichen. Daher werden die Funktionen hier nur auf dem Definitionsbereich betrachtet, in dem der Operator noch nicht genügend Arbeitsspeicher besitzt. Ab diesem Wert wird angenommen, dass sich die Kosten nicht mehr verändern und insbesondere nicht weiter reduziert werden können.

$$f(x) := 800 - 1.5 * x \tag{8.1}$$

Graphisch kann der Sachverhalt wie in Abbildung 8.1 dargestellt werden. In dem Diagramm ist zu erkennen wie die Kosten des Operators in Abhängigkeit vom Arbeitsspeicher fallen, bis der Punkt erreicht ist an dem der Operator genügend Arbeitsspeicher zur Verfügung hat.

Ein Beispiel für eine nichtlineare Kostenfunktion in Abhängigkeit vom Arbeitsspeicher wäre die Funktion $g : [16, 512] \to \mathbb{R}$ aus (8.2). Die Funktion ist in Abbildung 8.2 dargestellt.

$$g(x) := \frac{10000}{x} + 100 \tag{8.2}$$

Die Arbeitsspeicheroptimierung ist in diesen beiden Fällen ziemlich einfach. Die Optimierung der Arbeitsspeicherzuweisung besteht hier nur daraus, dem einzigen Arbeitsspeicherverwendendem Operator soviel Arbeitsspeicher zuzuweisen, wie er maximal benötigt oder maximal zur Verfügung steht, da die Kosten bis zu diesem Punkt immer weiter fallen.

Interessanter wird der Fall für zwei arbeitsspeicherverwendende Operatoren eines Ausführungsplans, der sich noch relativ gut graphisch darstellen läßt. Sind nun zwei Operatoren in einem Ausführungsplan vorhanden, welche die Operatorkostenfunktionen f und g besitzen, lauten die Gesamtkosten des Ausführungsplan $f(x) + g(y) + b$. b sind konstante Kosten aller anderen Operatoren in dem Ausführungsplan. Es wird also angenommen, dass keine weiteren speicherabhängigen Operatoren in diesem Plan vorhanden sind. Die Werte x und y geben die Arbeitsspeicherwerte für die Operatoren mit den Kostenfunktionen f und g an. Da die optimalen Arbeitsspeicherbelegungen gesucht sind, wird im Folgenden die Konstante b weggelassen. Da nun zwei Arbeitsspeicherparameter gesucht sind, ist auch nun eine weitere Dimension in dem Diagramm notwendig. Werden die

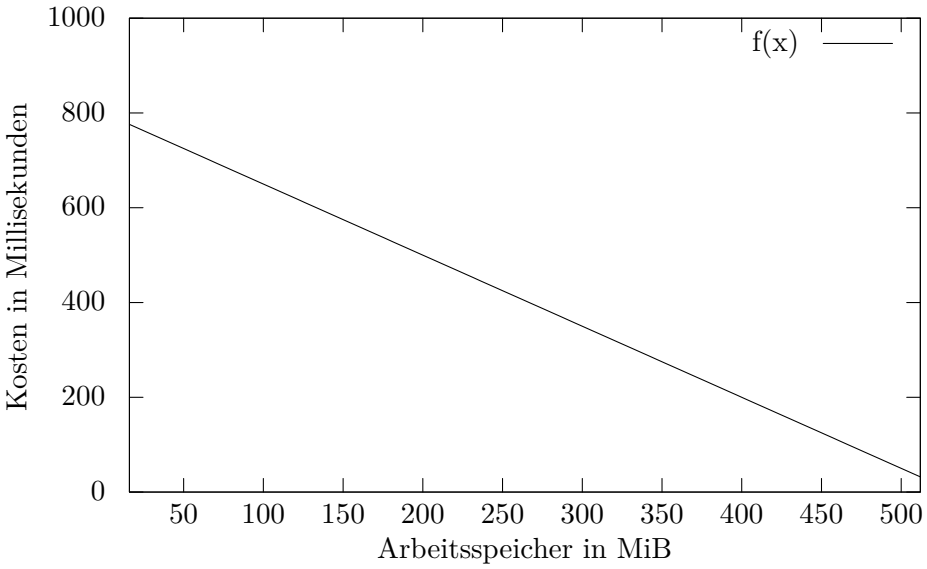

Abbildung 8.1: Lineare Operatorkosten in Abhängigkeit vom Arbeitsspeicher

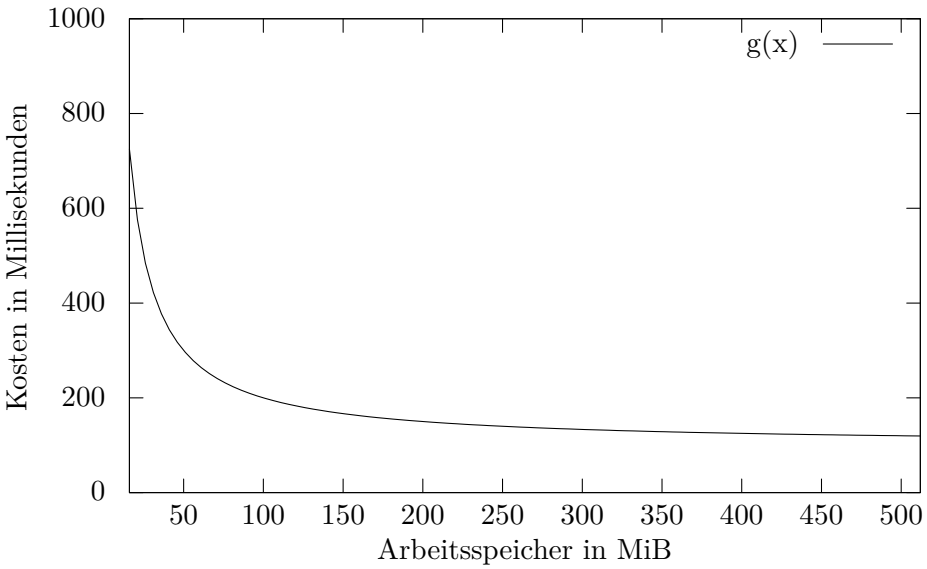

Abbildung 8.2: Nichtlineare Operatorkosten in Abhängigkeit vom Arbeitsspeicher

Kosten auf der z-Achse angezeigt, die x-Achse (y-Achse) gibt den gewählten Arbeitsspeicher für die Funktion g (f) an, stellt sich die Situation wie in Abbildung 8.3 dar.

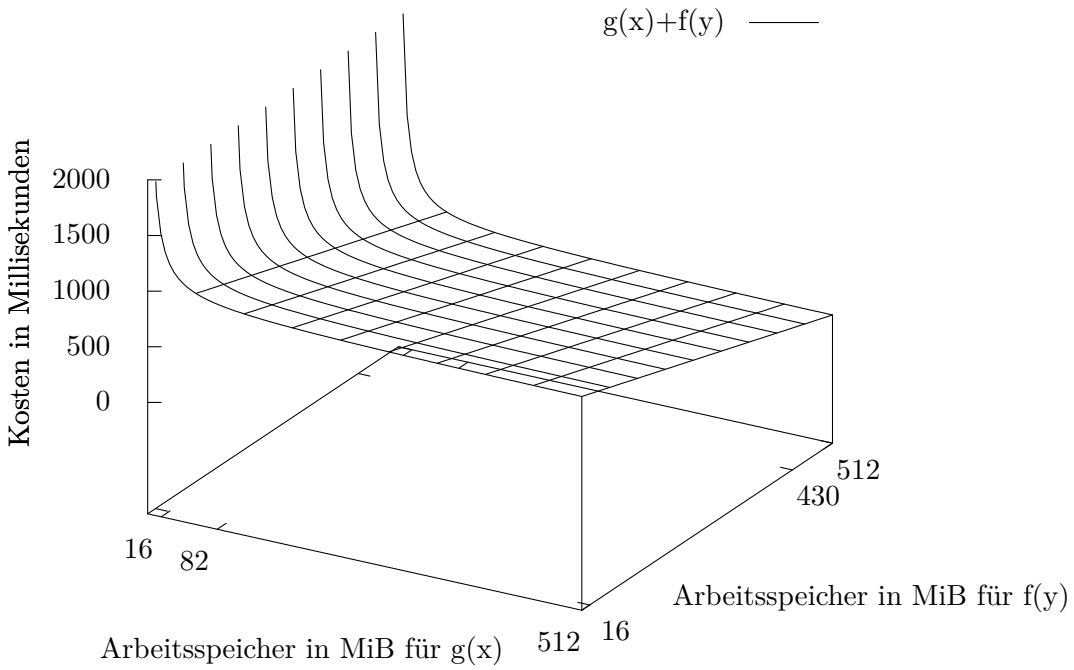

Abbildung 8.3: Gesamtkosten der zwei Operatoren in der dreidimensionalen Darstellung

Da nur maximal 512 MiB an Arbeitsspeicher zur Verfügung steht, ist nicht der ganze Bereich zulässig, sondern nur solange $x + y \leq 512$ gilt. Gesucht ist nun die x, y Kombination, die zu dem Wert $min\ f(x) + g(y)$ führt, denn diese Kombination besitzt die geringsten Gesamtkosten. Diese hier vorliegende Situation kann auch in einer zweidimensionalen Darstellung abgebildet werden. Dazu wird auf der x-Achse nach rechts hin der Wert von 0 MiB bis zu 512 MiB wachsen (untere Skala), um die Funktion $g(x)$ darzustellen. Die Funktion $f(x)$ wird aber von rechts nach links dargestellt (obere Skala) und besitzt ganz rechts den Wert 0 und wächst nach links bis zu 512 MiB. Die Gesamtkosten sind ergeben sich dann wieder durch Addition der beiden Funktionswerte. Die Situation ist in der Abbildung 8.4 auf der nächsten Seite dargestellt.

Die Werte für x und y, die zu den minimalen Gesamtkosten führen, können hier auf einen Blick ungefähr ermittelt werden, diese liegen bei ca. $x = 82\ MiB$ und $y = 430\ MiB$.

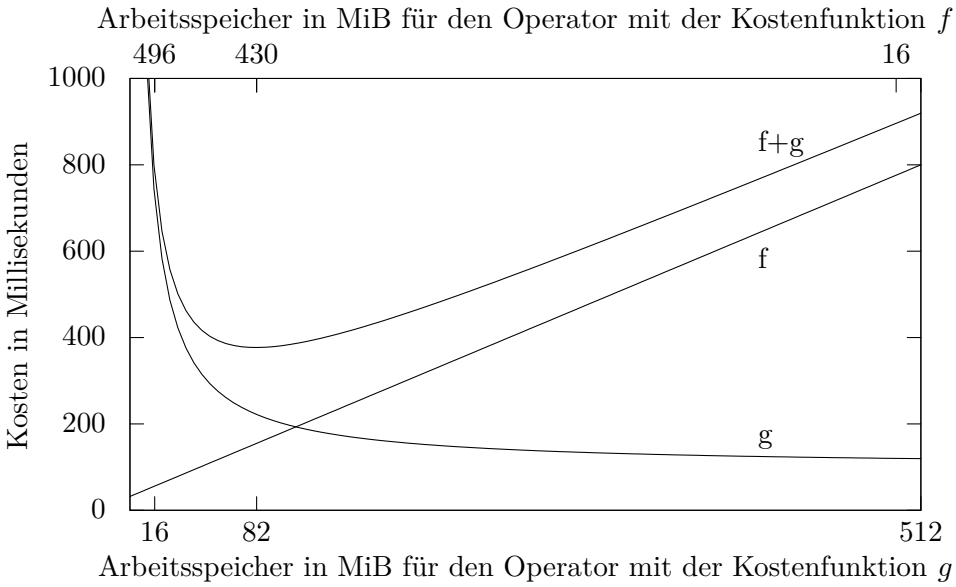

Abbildung 8.4: Zweidimensionale Einbettung des Optimierungsproblems der Gesamtkosten für zwei Operatoren

8.3 Operator-Kostenfunktionen und deren Arbeitsspeicherbedarf

Kosten repräsentieren für Operatoren des SECONDO DBMS die erwartete Laufzeit in Millisekunden. Im Folgenden wird daher hier auch nicht weiter zwischen Kosten und Laufzeit unterschieden, dies muss aber im Allgemeinen nicht der Fall sein. Die generelle Idee ist, die Kosten eines Operators in Abhängigkeit des vergebenen Arbeitsspeichers an diesen Operator darzustellen. Dies ist aber nicht Teil dieser Arbeit, sondern wurde in der Abschlussarbeit [Nid12] erarbeitet und implementiert. Im Zuge dessen wurde am Lehrgebiet eine Schnittstelle *CostEstimation* implementiert, die Operatoren ermöglicht, zu den Parametern der *Kardinalität* und den *Datensatzbreiten in Byte* die Kostenfunktion in Abhängigkeit vom Arbeitsspeicher zur Verfügung zu stellen. Dazu bereitgestellt wird auch die Menge an Arbeitsspeicher, die der Operator auf Basis der übergebenen Parameter für die Ausführung als optimal betrachtet (SufficientMemory). Ab diesem Wert besitzt der Operator genügend Arbeitsspeicher und kann keine Laufzeitverbesserung durch mehr zugewiesenen Arbeitsspeicher erreichen.

Unterschieden werden bisher folgende zwei Arten von Kostenfunktionen cf:

- Typ 1: Lineare Kostenfunktion der Form $cf(Memory) := y - (Memory * z)$
- Typ 2: Funktion der Form $cf(Memory) := \frac{A}{Memory} + B$

$Memory$ ist in diesen Formeln der zugewiesene Arbeitsspeicher in MiB und A und B sind operatorspezifische Konstanten, die von der *CostEstimation* Schnittstelle bereitgestellt werden und weiter unten beschrieben werden. Die Werte y und z für Typ 1 Funktionen hingegen werden erst noch aus den bereitgestellten Werten $SufficientMemory$, $TimeAt16MB$ und $TimeAtSuffMemory$ der

CostEstimation Schnittstelle ermittelt um die lineare Funktion zu konstruieren. Die lineare Funktion besitzt dazu für 16 MiB Arbeitsspeicher den Wert $TimeAt16MB$ und für $SufficientMemory$ den Wert $TimeAtSuffMemory$. Implementierungen der *CostEstimation* Schnittstelle besitzen momentan folgende Operatoren, die nach dem Funktionstyp gegliedert sind:

- Typ 1
  - mergejoin
  - sortmergejoin
  - symmjoin
  - gracehashjoin
  - hybridhashjoin
- Typ 2
  - itHashJoin
  - itSpatialJoin

Aus dem Optimierer können folgende Prädikate aufgerufen werden, um die Kostenfunktion in Abhängigkeit vom Arbeitsspeicher zu ermitteln:

```
1 getCostFun(+AlgId, +OpId, +FunId, +NoTuples1, +SizeOfTuple1,
2   -FunType, -DList)
3
4 getCostFun(+AlgId, +OpId, +FunId, +NoTuples1, +SizeOfTuple1,
5   +NoTuples2, +SizeOfTuple2, -FunType, -DList)
```

Das erste Prädikat ist für die Operatoren, die einen Eingabestrom verarbeiten und die zweite Version für Join-Operatoren, die zwei Eingabeströme verarbeiten. Es existieren in der Schnittstelle noch weitere Prädikate, welche die Kosten direkt berechnen oder eine lineare Funktion erzeugen. Allerdings werden sie für diese Erweiterung nicht benötigt. Die Optimierung basiert ausschließlich auf Funktionen, die über die *getCostFun* Prädikate bereitgestellt werden.

Um die obigen Prädikate aber aufrufen zu können, müssen die Parameter *AlgId (Algebra ID), OpId (Operator ID)* und *FunId (Function ID)* bekannt sein. Diese Parameter identifizieren einen Operator eindeutig im Secondo DBMS. Ermittelt werden können diese Werte mit dem Prädikat:

```
1 getOpIndexes(+OpName, +Signature, -ResultType, -AlgId, -OpId, -FunId)
```

Dazu ist dem Prädikat der Name des Operators und die Signatur der Parameter für den Operator zu übergeben.

Die *getCostFun* Prädikate liefern als Ausgabewerte den Wert *FunType* der den Typ der Funktion angibt, und den letzten Ausgabeparameter *DList*. Der Wert *DList* beinhaltet eine Liste, bestehend aus sieben Werten des C-Datentyp double[5] mit der folgenden Bedeutung:

1. *SufficientMemory*: Der Arbeitsspeicher in MiB, den der Operator als optimalen Wert für die Ausführung ansieht.

[5] In der Regel *64-bit IEEE floating point numbers*

2. *TimeAtSuffMemory*: Die geschätzten Kosten des Operators, wenn die optimale Menge an Arbeitsspeicher (SufficientMemory) vergeben wird.
3. *TimeAt16MB*[6]: Die geschätzten Kosten, wenn dem Operator 16 MiB Arbeitsspeicher zugewiesen wird.
4. *A*: Konstante, die in der Kostenfunktion vom Typ 2 verwendet wird.
5. *B*: Konstante, die in der Kostenfunktion vom Typ 2 verwendet wird.
6. *C*: Freie Konstante zur zukünftigen Verwendung.
7. *D*: Freie Konstante zur zukünftigen Verwendung.

Operatoren, die Arbeitsspeicher verwenden, um ihre Laufzeit zu verringern, sollten also, wenn ihnen mehr Arbeitsspeicher zugeteilt wird, schneller ihre Tätigkeit durchführen können und mehr Zeit benötigen, wenn ihnen weniger Arbeitsspeicher gewährt wird. Dies gilt solange, bis eine Obergrenze (SufficientMemory) erreicht ist, bei der ein Operator keinen weiteren Arbeitsspeicher mehr verwenden kann.

Im Weiteren wird davon ausgegangen werden, dass die speicherabhängigen Kostenfunktionen innerhalb des Intervalls [MinMemoryPerOperator, SufficientMemory] sowohl monoton fallend als auch stetig differenzierbar sind. Die Funktionen des Typ 1 und 2 genügen dabei diesen Bedingungen. Zukünftige Typen von Kostenfunktionen müssen diesen Bedingungen ebenso genügen, damit die getroffenen Annahmen in dieser Arbeit gültig bleiben. Auch wird davon ausgegangen, dass der Wert der Kostenfunktionen immer in diesem Intervall nicht negativ ist. Dies gilt da die Laufzeiten für Operatorfunktionen nicht negativ sein können.

8.4 Arbeitsspeicheroptimierung für einen gegebenen Ausführungsplan

Einerseits stehen nun die Kostenfunktionen zur Verfügung, die schon derzeit im Optimierer implementiert sind, anderseits stehen die Kostenfunktionen für die Operatoren aus Abschnitt 8.3 auf Seite 51 in Abhängigkeit vom gewährten Arbeitsspeicher zur Verfügung. Als letztes sind auch die optimalen Arbeitsspeicherwerte (SufficientMemory) bekannt. Im Optimierer selbst sind die Kardinalität der Zwischenergebnisse und die Datensatzbreiten bekannt[7]. Die letzte Eingabegröße, die für die Optimierung benötigt wird, ist die Gesamtmenge an Arbeitsspeicher, über die verfügt werden kann. Dies ist eine feste Größe, die aus der SECONDO Konfigurationsdatei *SecondoConfig.ini* kommt (GlobalMemory) und zur Laufzeit keiner Änderung unterliegt.

Die Kostenfunktionen, die hier betrachtet werden, sind additiv. Dies mag im Allgemeinen nicht immer der Fall sein, für das hier betrachtete Kostenmodell ist dies aber immer der Fall. Es gibt zwar den Operator *loopsel* für den die Kosten nicht additiv berechnet werden. Dieser wird allerdings aktuell nicht in Ausführungsplänen erzeugt, so dass zur Problemvereinfachung von additiven Kosten ausgegangen wird.

Generell ist die Optimierung solcher Funktionen möglich. In einem solchen Fall muss die Gesamtkostenfunktion aber aufwendiger konstruiert werden. Die Entscheidung, die Implementierung auf Basis der Annahme additiver Kosten durchzuführen, wurde getroffen, um die Ableitungen

[6] Die Variable ist so benannt, gemeint sind hier aber MiB.
[7] im Vorgriff auf den nächsten Abschnitt

einfach berechnen zu können. Zu beachten wäre für nicht additive Kosten, dass den Kanten im *predicate order graph* Teilpläne zugeordnet werden, die durch den kürzesten Pfad später zu dem ganzen Ausführungsplan zusammengesetzt werden. Für diese Teilpläne werden in der Phase der Optimierung die Kosten ermittelt. Sollen aber die Kosten dieser Teilpläne der Kanten nicht mehr additiv zu den Kosten des ganzen Ausführungsplans zusammengesetzt werden, dann ist auch der Dijkstra-Algorithmus zur Berechnung des kürzesten Pfades nicht mehr anwendbar[8].

Durch die additiv zusammengesetzten Kosten setzt sich die Gesamtkostenfunktion f eines Ausführungsplans bisher aus n einzelnen Kosten c_n zusammen:

$$f() := \sum_{i=1}^{n} c_i$$

Die Anzahl n bedeutet hier nicht, das auch n Operatoren im Ausführungsplan vorhanden sind, der Wert kann durchaus abweichen, beispielsweise für Operatoren, denen keine Kosten zugewiesen werden.

Wird die Integration der speicherabhängigen Kostenfunktionen in die Gesamtkostenfunktion für den Ausführungsplan vorgenommen, ändert sich die Gesamtkostenfunktion zu der Form:

$$f(x_1, ..., x_n) := \sum_{i=1}^{n} cf_i(x_i)$$

mit

$$cf_i(x_i) := \begin{cases} c_i & \text{falls die Operatorkosten } c_i \text{ konstant sind} \\ y_i - (x_i * z_i) & \text{falls die Operatorkosten vom Typ 1 sind} \\ \dfrac{A_i}{x_i} + B_i & \text{falls die Operatorkosten vom Typ 2 sind} \end{cases}$$

Die Werte für A_i, B_i, y_i und z_i sind dabei die Konstanten der Kostenfunktion des jeweiligen Operators mit dem Index i, wie diese in Abschnitt 8.3 auf Seite 51 beschrieben wurden, und die x_i sind die Arbeitsspeicherangaben für Funktionen vom Typ 1 oder 2.

Nun ist diese Kostenfunktion an Bedingungen geknüpft. Es kann nicht mehr als der verfügbare Arbeitsspeicher vergeben werden. Zudem existieren für jeden Operator bestimmte Grenzen, wieviel Speicher den Operatoren mindestens und höchstens zugewiesen wird. Als Minimalwert ist hier 16 MiB vorgegeben, als Maximalwert ist dabei der Wert an Arbeitsspeicher, den der Operator als optimal betrachtet (SufficentMemory). Der Maximalwert bezeichnet die Schwelle, ab der mit mehr Arbeitsspeicher keine Kostenverbesserung erreicht werden kann. Ab diesem Punkt muss sich von der Kostenfunktion gelöst werden. Rechnerisch ist auf Basis der bisher implementierten Kostenfunktionen, abgesehen vom Spezialfall der konstanten Kostenfunktionen, immer eine Verbesserung möglich. Allerdings sind diese Funktionen Approximationen, für die nur verlangt wird, auf dem Bereich zwischen dem Minimal- und Maximalwert des Arbeitsspeichers vernünftige Werte zu liefern (Vgl. auch [Nid12]).

Problemfälle dieser Art, in der die Parameterwerte $x_1, ..., x_n$ gesucht werden, die zur Minimierung (oder Maximierung) einer Zielfunktion führen, hier der Gesamtkostenfunktion, sind nun typische Optimierungsprobleme. Da unter anderem die Gesamtkostenfunktion selbst nicht mehr linear ist, wenn Operatorkostenfunktionen vom Typ 2 vorhanden sind, liegt hier der Fall eines nichtlinearen Optimierungsproblems vor, wie es in Abschnitt 2.4 auf Seite 14 beschrieben wurde.

[8] Der Dijkstra-Algorithmus geht von Kantenkosten aus, die additiv zusammenhängen [Dij59].

Die grundlegende Idee ist, die optimale Arbeitsspeicherverteilung eines gegebenen Ausführungsplans durch die Lösung des entsprechenden nichtlinearen Optimierungsproblems zu ermitteln. Die dadurch gewonnenen x_i lösen damit das Problem der zweiphasigen Arbeitsspeicheroptimierung.

8.5 Berücksichtigung blockierender Operatoren

Im vorherigen Abschnitt wurde davon ausgegangen, dass der gesamte, zur Verfügung stehende Arbeitsspeicher (GlobalMemory) auf alle Operatoren genau einmal aufgeteilt werden kann. Nun besitzen aber manche Operatoren die Eigenschaft blockierend zu arbeiten, d.h. diese Operatoren sind erst in der Lage ein Ergebnis auszugeben, wenn diese die gesamte Eingabe verarbeitet haben. Aus der Liste von Operatoren aus Abschnitt 8.3 auf Seite 51 arbeitet nur der *sortmergejoin* Operator blockierend. Da dieser Operator im Secondo DBMS mit Strömen von Tupeln arbeitet, bedeutet dies, dass der Operator erst in der Lage ist, das erste Tupel in den Tupelstrom auszugeben, wenn er alle Tupel des Eingabestroms verarbeitet hat.

Die Eigenschaft von Operatoren blockierend zu arbeiten kann sich in der Speicheroptimierung zunutze gemacht werden. Da also ein blockierender Operator seine erste Ausgabe erst nach vollständiger Abarbeitung der Eingabe beginnt, ist die Verarbeitung der Operatoren, die die Eingabe des blockierenden Operators erzeugen, vollständig abgeschlossen. Falls genau für diese Eingabe erzeugenden Operatoren Arbeitsspeicher vergeben wurde, steht dieser Arbeitsspeicher wieder zur Verfügung und kann ein weiteres Mal vergeben werden. Die Modellierung dieser Tatsache in das nichtlineare Optimierungsproblem bedingt, dass die Einschränkung auf den gesamten Arbeitsspeicher (GlobalMemory) ersetzt wird durch mehrere Restriktionen, in denen der gesamte Arbeitsspeicher nur auf bestimmte Operatoren angewendet wird.

Die Eigenschaft der Operatoren blockierend zu arbeiten ist dabei aber keine wünschenswerte Eigenschaft. Hier wird die Eigenschaft zwar ausgenutzt, um eine Verbesserung der Speicheroptimierung zu erreichen, aber generell arbeiten nicht blockierende Operatoren effizienter, da hier insbesondere auf die Technik der Stromverarbeitung zurückgegriffen werden kann. Die Eigenschaft blockierend zu arbeiten kann dabei für Operatoren im Secondo DBMS erzwungen werden indem ein Tupelstrom nach der Anwendung eines nicht blockierenden Operators zuerst mit *consume* in eine Relation geladen wird und dann per *feed* Operator wieder ein Tuplestrom erzeugt wird. Da der *consume* Operator blockierend arbeitet, würde an dieser Stelle im Ausführungsplan eine Blockierung auftreten. Durch Anwendung dieser Methode auf jeden speicherverwendenden Operator, der nicht ohnehin schon blockierend arbeitet, könnte daher jedem dieser Operatoren der maximale Arbeitsspeicher zugewiesen werden. Da aber davon auszugehen ist, dass die Methode im Allgemeinen keine Verbesserung der Gesamtkosten erreichen kann, wurde dies nicht weiter untersucht.

8.6 Die zweiphasige Arbeitsspeicheroptimierung

Da in der zweiphasigen Optimierung zuerst der beste Ausführungsplan, ohne Betrachtung der Ressource Arbeitsspeicher ermittelt wird, und danach erst die Speicheroptimierung durchgeführt wird, müssen die Kosten im Graphen der Kostenkanten auf Basis des POG fest sein, um einen Ausführungsplan ermitteln zu können. Dazu müssen also die Kosten mit einer bestimmten Menge an Arbeitsspeicher berechnet werden und dann den Teilplänen der Kanten zugewiesen werden. Für verschiedene Belegungen der Arbeitsspeicherzuweisungen ist es also möglich, unterschiedliche Ausführungspläne zu erhalten. Dies liegt daran dass der Dijkstra-Algorithmus dann jeweils unterschied-

liche kürzeste Pfade in dem Graphen der Kostenkanten auf Basis des POG ermittelt. Idealerweise würden diese so gewählten fixen Werte dazu führen, dass auch der Ausführungsplan ausgewählt würde, der dem optimalen Ausführungsplan entspricht, der auch in der einphasigen Optimierung ausgewählt würde. Da diese Belegung aber nicht bekannt ist, sollen vier verschiedene Strategien evaluiert werden:

- *static(MinMemoryPerOperator)*: Als erstes wird hier der Ausführungsplan anhand des Graphen der Kostenkanten auf Basis des POG ermittelt werden indem die Kosten für arbeitsspeicherverwendende Operatoren mit 16 MiB pro Operator (MinMemoryPerOperator) berechnet werden. In der zweiten Phase wird dann die Speicheroptimierung auf dem so ermittelten Ausführungsplan durchgeführt.

- *static(GlobalMemory)*: Dieser Fall ist ähnlich wie die *static(MinMemoryPerOperator)* Strategie. Hier wird auch wieder zuerst der Ausführungsplan mit fest vorgegebenen Arbeitsspeicherzuweisungen berechnet. Dies erfolgt hier mit dem gesamten verfügbaren Speicher (GlobalMemory) pro speicherverwendenden Operator. In der zweiten Phase wird dann der so ermittelte Ausführungsplan auch hier wieder der Arbeitsspeicheroptimierung unterzogen.

- *staticMaxDivOpCount(X)*: Diese Strategie soll die beiden erst genannten Strategien etwas verbessern, indem nicht für alle Operatoren ggf. zuwenig (*static(MinMemoryPerOperator)*) oder zuviel (*static(GlobalMemory)*) Arbeitsspeicher vergeben wird. Für wenige Operatoren ist die *static(MinMemoryPerOperator)* Strategie nämlich ungeeignet und für viele Operatoren trifft die *static(GlobalMemory)* Strategie eine schlechte Wahl. Daher wird hier der verfügbare Speicher durch den Wert X geteilt. Der Wert X wird zuvor berechnet und enthält die Anzahl der speicherverwendenden Operatoren auf dem Pfad mit den meisten speicherverwendenden Operatoren. Allerdings kann der Wert X nur durch Aufzählen aller Pfade berechnet werden. Daher ist diese Strategie auch nur für die Evaluierung interessant.

- *staticMaxDivPOG(X)*: Mit dieser Strategie wird versucht ein ähnliches Verhalten wie mit der *staticMaxDivOpCount(X)* Strategie zu realisieren. X ist hierbei die Anzahl der Prädikate des zugehörigen SQL-Statements. Im Gegensatz zu dem Wert X der *staticMaxDivOpCount(X)* Strategie kann dies ohne nennenswerten Aufwand berechnet werden. Dafür entspricht der Wert allerdings nicht immer exakt dem Wert X der *staticMaxDivOpCount(X)* Strategie. Die Verwendung der Anzahl der Prädikate als Divisor liegt in der Tatsache begründet, dass in der aktuellen Secondo Version nur Join-Operatoren speicherverwendende Operatoren sind und verbindende SQL-WHERE-Bedingungen einen solchen Join-Operator im entsprechenden Ausführungsplan erzeugen.

Gemeinsam ist den drei Möglichkeiten, dass diese mehr Arbeitsspeicher als verfügbar vergeben könnten, allerdings ist der Zweck auch nur, in der ersten Phase überhaupt einen Plan zu ermitteln. Die eigentliche Arbeitsspeicheroptimierung erfolgt dann in der zweiten Phase. Optimalerweise ist dieser mit der Strategie ausgewählte Pfad auch nach der Speicheroptimierung identisch mit dem Pfad, der auch in der einphasigen Speicheroptimierung ermittelt wird. Allerdings kann dies nicht garantiert werden, aber dafür führt diese Strategie die Speicheroptimierung in zwei Phasen durchzuführen zu einer Reduzierung der Komplexität und Laufzeit. Später wird versucht werden exemplarisch zu evaluieren, wie groß die Abweichung der zweiphasigen Optimierung zu der einphasigen Optimierung ist.

8.7 Die einphasige Arbeitsspeicheroptimierung

Komplizierter stellt sich die Sache dar, wie die Arbeitsspeicheroptimierung bereits in der Phase der Ermittlung des kostengünstigsten Ausführungsplans über den Graphen der Kostenkanten auf Basis des POG erfolgen kann. Die einfachste, aber auch ineffizienteste Möglichkeit ist, alle möglichen Pfade aufzuzählen und jeden dieser Pfade, die einen Ausführungsplan darstellen, der vorher diskutierten Arbeitsspeicheroptimierung zu unterziehen. Bei dem exponentiellen Wachstum der Anzahl möglicher Pfade ist diese Möglichkeit nur für eine Evaluierung interessant. Für den regulären Einsatz eignet sich diese Strategie aufgrund der dadurch hohen Kosten nicht. Abgesehen von der einfachen Implementierung ist der einzige Vorteil, dass hier garantiert wird, den optimalen Ausführungsplan, auch bezüglich der Ressource Arbeitsspeicher, zu finden.

Idealerweise wäre zur Lösung des Problems ein Algorithmus zum Berechnen eines kürzesten Pfades in einem zyklenfreien Graphen verfügbar, der folgenden Anforderungen genügt: Den Kostenkanten in dem Graphen auf Basis des POG können nichtlineare Kostenfunktionen in Abhängigkeit jeweils einer Variablen (hier dem Arbeitsspeicher) zugewiesen werden. Weiterhin muss für die Variablen das Angeben von Bedingungen möglich sein, um die minimalen und maximalen Werte angeben zu können. Damit kann dann der Arbeitsspeicher, der dem Operator minimal und maximal zugewiesen kann, eingeschränkt werden. Als letztes muss auch für den Pfad selber eine Bedingung angegeben werden können, um den maximalen Arbeitsspeicher, der vergeben werden kann, zu beschränken. Der berechnete, kürzeste Pfad ist dann der Pfad, der die geringsten Gesamtkosten für die Belegeung der Variablen unter Berücksichtigung der Bedingungen aufweist.

Die Ausgabe eines solchen Algorithmus wäre dann nicht nur der kürzeste Pfad, auch die Belegung der Variablen, die zu dem kürzesten Pfad geführt hätten, würden mit ausgegeben. Dann würde die Möglichkeit bestehen, das Speicheroptimierungsproblem in den Graphen der Kostenkanten auf Basis des POG zu integrieren, indem der Dijkstra-Algorithmus durch diesen Algorithmus ersetzt wird. Dabei existiert zur Ermittlung des kürzesten Pfades eine bemerkenswerte Anzahl unterschiedlichster Algorithmen, die verschiedenste Aspekte berücksichtigen. Es sind auch Algorithmen bekannt, welche die Anforderungen von oben teilweise erfüllen, allerdings konnte kein Algorithmus ermittelt werden, der alle Aspekte berücksichtigt. Hätte man allerdings so einen Algorithmus zur Verfügung, wäre noch der Punkt offen, wie die Integration der blockierenden Operatoren in so einen Algorithmus erfolgen könnte.

In Ermangelung der Verfügbarkeit eines solchen optimalen Algorithmus, wurde ein Verfahren implementiert, das auf eine Idee von Herrn Prof. Dr. Güting zurückgeht. Dazu wurde der Dijkstra-Algorithmus wie folgt angepasst: Wenn eine Kante bei der Durchführung des Algorithmus erstmalig betrachtet wird, wird der Ausführungsplan, zu dem aktuellen Pfad inklusive der gerade betrachteten Kante der Speicheroptimierung unterzogen. Die Kosten der neu betrachteten Kante sind dann die Kosten der Differenz zwischen den Kosten des bisherigen Pfades und den Kosten nach der Speicheroptimierung. Wichtig ist hier die Tatsache, dass diese Differenz immer einen Wert größer Null besitzt. Wäre dies nicht so, wäre der Dijkstra-Algorithmus nicht mehr geeignet, da dieser nicht in der Lage ist, negative Kantengewichte zu berücksichten. Im folgenden wird dieser Algorithmus als *modifizierter Dijkstra-Algorithmus* bezeichnet.

Dieser so angepasste Algorithmus ist allerdings nicht mehr korrekt. Es sei angenommen, dass zwei Pfade p und q existieren in einem gegebenen Graphen der Kostenkanten auf Basis des POG, die vom Startknoten zum Knoten n führen, der nicht der Endknoten ist, und die Kante e neu betrachtet wird. Weiterhin sei angenommen, dass die Kante e genau einen speicherverwendenden

Operator besitzt, der nicht mit genügend Arbeitsspeicher versehen werden kann. Der Pfad p ist dabei der bisherige kürzeste Pfad, der mindestens einen Operator mit speicherabhängigen Kosten[9] beinhaltet. q beinhaltet aber keinen Operator mit speicherabhängigen Kosten. Dann werden die Gesamtkosten des Pfades $q + e$ geringer sein als die Gesamtkosten des Pades $p + e$, sofern die Kostendifferenz der Pfade p und q kleiner ist, als die Differenz der Kosten der Kante e im Pfad $p + e$ und der Kante e im Pfad $q + e$. In diesem Fall wird dadurch, dass dem speicherverwendenden Operator in der Kante e mehr Arbeitsspeicher im Pfad $q + e$ zugewiesen werden kann als im Pfad $p + e$, der bisherige Kostenvorteil des Pfades p mehr als kompensiert. Damit ist der Pfad $p + e$ nicht mehr der kürzestes Pfad. Insbesondere hat der Dijkstra-Algorithmus von dem Pfad q hier keine Kenntnis. Um garantiert den günstigsten Pfad zu ermitteln, müssten daher eigentlich alle möglichen Pfade, die zum Knoten n führen, neu betrachtet werden. Dem Algorithmus liegt die Hoffnung zu Grunde, dass ein so sukzessiv aufgebauter Pfad auch weiterhin immer günstig bleibt, was, wie gezeigt, nicht immer der Fall sein kann. In dem Sinne ist es also eher eine Heuristik, deren Güte im Vergleich zur Optimallösung später versucht wird für ein paar Fälle zu bewerten.

Die Komplexität ist in der zweiphasigen Speicheroptimierung dabei wesentlich geringer, da dort nur eine nichtlineare Optimierung durchgeführt wird.

[9] Von dem Spezialfall, dass eine Kostenfunktion konstant sein könnte, sei hier abgesehen.

Kapitel 9

Implementierung

9.1 Vom Kostenmodell *improvedcosts* zu den *ma_improvedcosts*

Im Secondo DMBS sind mehrere Kostenmodelle implementiert. Das Standardkostenmodell (*standardcosts*) des Optimierers verwaltet zwar die Kardinalität, allerdings nicht die Datensatzbreiten. Daher wird als Basis für die Entwicklung des neuen Kostenmodells das Kostenmodell *improvedcosts* verwendet. Benötigt werden die Datensatzbreiten bei der Beschaffung der Kostenfunktionen über die *CostEstimation* Schnittstelle [Vgl. Abschnitt 8.3 auf Seite 51]. Auch wird ein Basiskostenmodell deswegen benötigt, da nicht für alle Operatoren eine Implementierung der *CostEstimation* Schnittstelle vorliegt. Für andere als in Abschnitt 8.3 auf Seite 51 genannten Operatoren werden daher die Kosten weiterhin durch die übernommenen Prädikate des Kostenmodells der *improvedcosts* kalkuliert.

Da nun Kosten durch zwei unterschiedliche Programme zusammengebracht werden, wurde deutlich darauf zu achten, dass die Kosten der *improvedcosts* sowie die Kosten, die über die *CostEstimation* Implementierungen berechnet werden, normiert sind. Beispielsweise darf nicht vorkommen, dass in den *improvedcosts* von sequentiellen Leseraten der Festplatte von 10 MiB/s ausgegangen wird, die *CostEstimation*-Implementierung aber mit sequentiellen Leseraten von 100 MiB/s rechnet. In diesem Fall wären die Kosten nicht miteinerander vergleichbar. Selbiges gilt natürlich auch für die verschiedenen *CostEstimation* Implementierungen. Dieser Aspekt wurde auch in der Abschlussarbeit [Nid12] kurz angesprochen. Letztlich existieren hier aber wohl noch Differenzen, die sich in der Optimierung auswirken können.

Um dieses angepasste Kostenmodell von den derzeitigen *improvedcosts* unabhängig implementieren zu können, erfolgte die Implementierung in der Datei „ma_improvedcosts.pl“. Dieses Kostenmodell wird zwingend verwendet, wenn im Optimierer diese Erweiterung aktiviert ist. Die Grundlage ist daher eine einfache Kopie der Datei „improvedcosts.pl“.

Für jeden Operator, der eine *CostEstimation* Implementierung bereitstellt, wurde das entsprechendes Kostenprädikat der *ma_improvedcosts* abgeändert. Die Ermittlung der Kosten wird darauf abgeändert, dass zuerst die Kostenfunktion des Operators über die *CostEstimation* Schnittstelle ermittelt wird. Auf Basis der ermittelten Kostenfunktion und dem diesem Operator zugeteilten Arbeitsspeicher werden die Kosten berechnet.

9.2 Optimierung eines gegebenen Ausführungsplans

Was sowohl für die einphasige als auch für die zweiphasige Speicheroptimierung benötigt wird, ist die Implementierung der Optimierung eines gegebenen Ausführungsplans (Vgl. Abschnitt 8.4 auf Seite 53). Auf Basis der speicherabhängigen Kostenfunktionen wird also ein nichtlineares Optimie-

rungsproblem definiert. Zur Vereinfachung werden dabei nur diese speicherabhängigen Kostenfunktionen mit in die Zielfunktion aufgenommen. Da die Kostenfunktionen der einzelnen Operatoren alle additiv zu der Gesamtkostenfunktion zusammengesetzt werden, ist die Lösung dieses vereinfachten Optimierungsproblems auch die Lösung für die Minimierung der Gesamtkostenfunktion.

Wenn n speicherabhängige Kostenfunktionen in dem gegebenen Ausführungsplan existieren, lautet das Optimierungsproblem wie folgt, wenn die Variablen x_i $(1 \leq i \leq n)$ die Arbeitsspeicherangabe für den jeweiligen Operator i darstellen:

$$\min f(x_1, x_2, ..., x_n)$$

mit

$$f(x_1, x_2, ..., x_n) := \sum_{i=1}^{n} cf_i(x_i)$$

$$cf_i(x_i) := \begin{cases} y_i - (x_i * z_i) & \text{falls die Operatorkosten vom Typ 1 sind} \\ \frac{A_i}{x_i} + B_i & \text{falls die Operatorkosten vom Typ 2 sind} \end{cases}$$

unter den Nebenbedingungen

$$\forall_{i \in \{1,...,n\}} \Big(x_i \geq MinMemoryPerOperator \Big)$$

$$\forall_{i \in \{1,...,n\}} \Big(x_i \leq max(MinMemoryPerOperator + 0,9MiB, SufficientMemory_i) \Big)$$

$$\sum_{i=1}^{n} x_i \leq GlobalMemory - n$$

Die Werte y_i, z_i, A_i, B_i, $SufficientMemory_i$ sind auch hier wieder operatorabhängige Konstanten, die über die *CostEstimation* Schnittstelle bereitgestellt werden und in Abschnitt 8.3 auf Seite 51 beschrieben wurden. Warum in der zweiten Bedingung der Wert n abgezogen wird, wird am Ende des Abschnitts genauer beschrieben. Der Allquantor dient dazu vereinfachend darzustellen, dass mehrere Restriktionen vorhanden sind. Beispielsweise werden durch $\forall_{i \in \{1,...,n\}}$ n Restriktionen mit den entsprechenden Variablenbindungen dem Optimierungsproblem hinzugefügt.

In der zweiten Restriktionsgleichung taucht eine Besonderheit auf. Der maximale Arbeitsspeicher für einen Operator ist auf $max(MinMemoryPerOperator, SufficientMemory_i)$ begrenzt. Allerdings wird durch die Addition von 0,9 MiB dafür gesorgt, dass der maximale Arbeitsspeicher immer mindestens um 0,9 MiB größer ist als der Minimal zu vergebene Arbeitsspeicher. Die Ursache hierfür ist technischer Natur. Vom Prinzip her sollte ein Operator, der mit 16 MiB auskommt auch nicht mehr als 16 MiB bekommen. Damit würde die Restriktion des Arbeitsspeichers für diesen Operator $16 \leq x_i \leq 16$ lauten und die Lösung sollte dann $x_i = 16$ lauten. Aufgrund des verwendeten Algorithmus bei der Lösung des Optimierungsproblems kann aber der Fall eintreten, dass das Optimierungsverfahren sehr lange laufend neue Werte für x_i erzeugt, die aber die Restriktion nicht erfüllen, da diese minimal von dem Wert 16 MiB abweichen. Daher wird durch die Addition von 0,9 MiB dafür gesorgt, dass hier der Optimierungsalgorithmus in der Lage ist, auch einen Wert für x_i zu erzeugen, der die Resktriktion erfüllt. Dass nicht der Wert von einem MiB verwendet wurde, hat die Ursache darin, dass am Ende die Ergebnisse aufgerundet werden und dann durch die Addition von einem MiB ggf. dazu führt ein MiB zuviel zu vergeben. Alternativ hätte ein Operator, der nicht

mehr als die minimale Menge an Arbeitsspeicher benötigt, nicht in dem Optimierungsproblem berücksichtigt werden können. Allerdings verliert sich dadurch die leicht nachvollziehbare Verbindung zwischen einem Operator in einem Ausführungsplan und dem Optimierungsproblem. Dies wäre für die Fehleranalyse oder für die Nachvollziehbarkeit durch Dritte nachteilig.

Als letztes sollen noch die blockierenden Operatoren in das nichtlineare Optimierungsproblem mit aufgenommen werden. Damit lautet die finale Version des Optimierungsproblems wie folgt:

$$\min f(x_1, x_2, ..., x_n)$$

mit

$$f(x_1, x_2, ..., x_n) := \sum_{i=1}^{n} cf_i(x_i)$$

$$cf_i(x_i) := \begin{cases} y_i - (x_i * z_i) & \text{falls die Operatorkosten vom Typ 1 sind} \\ \dfrac{A_i}{x_i} + B_i & \text{falls die Operatorkosten vom Typ 2 sind} \end{cases}$$

unter den Nebenbedingungen

$$\forall_{i \in \{1,...,n\}} \Big(\quad x_i \geq MinMemoryPerOperator \Big)$$

$$\forall_{i \in \{1,...,n\}} \Big(\quad x_i \leq max(MinMemoryPerOperator + 0,9MiB, SufficientMemory_i) \Big)$$

$$\forall_{j \in \{1,...,k\}} \Big(\sum_{i \in S_j} x_i \leq GlobalMemory - |S_j| \Big)$$

Die Rahmen S_j sind Mengen von Indizes der Operatoren, auf denen der gesamte Arbeitsspeicher jeweils aufgeteilt werden kann. Hier ist k die Anzahl der Rahmen und $|S_j|$ gibt die Kardinalität des Rahmens S_j an. Für die S_j gilt $S_j \subseteq \{1, ..., n\}$. Welche Bedingungen die S_i erfüllen müssen und wie diese Rahmen ermittelt werden, wird in Abschnitt 9.6 auf Seite 65 behandelt. In dem Fall, dass kein blockierender Operator vorhanden ist, gilt $k = 1$ und $S_1 = \{1, ..., n\}$. In diesem Fall ist der gesamte Arbeitsspeicher auf alle Operatoren aufzuteilen.

Damit die definierten Restriktionsfunktionen in der NLopt Bibliothek verwendet werden können, sind diese aber noch in die Form $g(x) \leq 0$ zu bringen. Diese triviale Umformung [MG00, Seite 3] wird hier nicht mehr explizit dargestellt.

Zum Lösen dieses nichtlinearen Optimierungsproblems wurden eine Reihe von C++ Funktionen (MemoryOptimization.cpp) implementiert, die die im Abschnitt 2.5 auf Seite 15 besprochene Bibliothek NLopt verwenden. Aus der Prolog-Umgebung wird die dort durchgeführte Optimierung über das Prädikat *memoryOptimization/5* aufgerufen. Dabei sind die in NLopt implementierten Algorithmen alle Näherungsverfahren. Daher wird auch ein Abbruchkriterium benötigt, um bei einer bestimmten Genauigkeit das Verfahren zu beenden. Gewählt wurde hierzu der Wert 0,25 MiB, da dieser für diese Zwecke genau genug ist und durch die Wahl eines zu kleinen Wertes die Laufzeit unnötig erhöht würde. Unter den verschiedenen, möglichen Abbruchmethoden wird der Wert von 0,25 MiB in dieser Implementierung so interpretiert: Ändert sich in allen Parametern der Zielfunktion der Wert um weniger als 0,25 MiB, wird die Suche beendet.

Der für die nichtlineare Optimierung verwendete Algorithmus ist *Method of Moving Asymptotes (MMA)* [Vgl. 2.5]. Ausgewählt wurde dieser Algorithmus, da er insbesondere eine beliebige Anzahl von Ungleichsbedingungen unterstützt. Da dieser Algorithmus gradientenbasiert ist, sind auch die

Ableitungen dem Algorithmus zur Verfügung zu stellen, die aber für die hier vorkommenden Fälle einfach zu berechnen sind. Allerdings ist dieser Algorithmus ein Verfahren zum Auffinden eines lokalen Optimums. Die hier verwendeten Funktionen vom Typ 1 und 2 sowie die Restriktionsfunktionen sind konvexe Funktionen und da die Addition von konvexen Funktionen wieder eine konvexe Funktion ergibt [MG00, Seite 5], gilt hier, dass ein lokales Minimum auch ein globales Minimum ist[1]. Letztlich würde aber auch die Möglichkeit bestehen, ein globales Verfahren einzusetzen, sofern der dann vewendete Algorithmus eine beliebige Anzahl von Restriktionsungleichungen unterstützt. Dazu wäre im Prinzip nur die Konstante zur Auswahl des Algorithmus auszutauschen. Allerdings wird in dem Fall dazu geraten, erst einen globalen Algorithmus zu verwenden um ein globales Minimum zu identifzieren und dann ein lokales Verfahren einzusetzen, um in diesem Bereich eine exaktere Lösung zu erhalten [Nlo]. Um nicht zwei Algorithmen einsetzen zu müssen, wurde von vornherein ein lokaler Algorithmus verwendet. Der MMA Algorithmus ist aber nicht auf die Ermittlung eines lokalen Minimums für konvexe Funktionen beschränkt. Wenn nicht konvexe Kostenfunktionen eingesetzt werden, ist nachzuweisen, dass ein so gefundenes lokales Minimum auch ein globales Minimum aus dem Definitionsbereich ist.

Auf folgende Besonderheit soll hier noch kurz eingegangen werden: Die Optimierung wird zur Berechnung der Lösung die Zielfunktion, die Restriktionsfunktionen sowie die Ableitungsfunktionen verwenden. Allerdings stehen diese nur in der Prolog-Umgebung zur Verfügung und werden auch nicht an die C++ Umgebung übergeben. Die Funktionsweise ist folgendermaßen: Prolog-Prädikate werden aus der C++ Umgebung aufgerufen, welche die jeweiligen Funktionswerte berechnen. Die Kosten für ein derartiges Verfahren sind dabei höher, als wenn diese direkt in der C++ Umgebung berechnet werden. Dass dies so implementiert wurde, liegt ursächlich darin, dass zu Beginn der Implementierung nicht klar war, wie und welche Funktionen über die *CostEstimation* Schnittstelle geliefert werden. Daher war die Idee, dass durch die Verwendung der Prolog-Prädikate zur Berechnung auch weit komplexere Funktionen einfach verwendet werden können. Die Übergabe von Funktionen, die in Prolog Terme sind, die mit dem *is/2* Operator ausgewertet werden können, an die C++ Umgebung wäre dann nicht mehr so einfach möglich gewesen.

Die beschriebene Optimierung eines Ausführungsplans wird von dem Prädikat *pathMemoryOptimization(+Path, -RPath)* durchgeführt. Dieses Prädikat arbeitet dabei noch mit dem Pfad im Graphen der Kostenkanten auf Basis des POG, der den Ausführungsplan beinhaltet. Indem mit dem entsprechenden Pfad gearbeitet wird, und nicht mit dem eigentlichen Ausführungsplan, kann dieses Prädikat auch in der Form für die einphasige Speicheroptimierung verwendet werden. Dazu wird im ersten Schritt der Pfad durch das Prädikat *analysePath(+Path)* analysiert. Dieses beinhaltet das Erstellen der Zielfunktion, der Restriktionen sowie der Ableitungen. Nach diesem Schritt wird die nichtlineare Optimierung des oben definierten Optimierungsproblems durchgeführt. Der letzte Schritt besteht darin, auf Basis der Lösung des Minimierungsproblems die Arbeitsspeicherzuteilungen durchzuführen. Am Ende wird der Pfad mit den neu berechneten Kosten dem aufrufenden Prädikat in dem Parameter *RPath* bereitgestellt.

9.3 Arbeitsspeicherzuteilung

Der Schritt der Arbeitsspeicherzuteilung weist den Operatoren nicht unmittelbar den Wert aus dem Ergebnis der nichtlinearen Optimierung zu, denn diese Werte sind vom C-Typ *double*. Die

[1] Der Definitionsbereich eines konvexen Optimierungsproblems muss eine konvexe Menge sein [MG00, Seite 5]. Da jedes x_i sich im Intervall [0, GlobalMemory] bewegt, ist dies auch hier der Fall.

Zuweisung in den Ausführungsplänen erlaubt nur ganzzahlige Werte der Einheit MiB. Diese Speicherzuteilung besteht dabei nicht nur aus einem einfachen Runden. Wenn ein Wert abgerundet würde, könnte es passieren, dass der ungerundete berechnete Arbeitsspeicherwert dazu geführt hätte, dass der Operator genug Arbeitsspeicher hätte, aber durch die Abrundung der Operatoren nicht mehr ausreichend Arbeitsspeicher besitzt, um seine Arbeit komplett im Arbeitsspeicher durchzuführen. Genau an dieser Stelle tritt ein Sprung in den tatsächlichen Kosten auf [Nid12, Abschnitt 6.2], die in den jeweiligen Approximationen, die zur Kostenfunktion, welche stetig ist, führten, nicht zu erkennen sind. Um nicht in die Situation zu geraten, dass im Extremfall ein Byte an Arbeitsspeicher fehlt, wird daher immer aufgerundet, soforn der Wert nicht schon ganzzahlig ist. Mit dieser Änderung ist es möglich, dass mehr als der zur Verfügung stehende Arbeitsspeicher vergeben wird. Um auch dies zu vermeiden, wird nicht der ganze verfügbare Arbeitsspeicher an die Optimierung übergeben, sondern der Wert, der sich ergibt, indem von dem verfügbaren Arbeitsspeicher die Anzahl der speicherverwendenden Operatoren abgezogen wird. Könnten also 512 MiB vergeben werden und in einem Ausführungsplan wären fünf speicherverwendende Operatoren enthalten, würden in der Optimierung selbst nur 507 MiB verwendet. Diese Anpassung findet sich daher in der dritten Restriktionsgleichung des Optimierungsproblems wieder. Nun wiederum kann der Fall eintreten, dass für die Speicherbelegung ein gerader Wert berechnet wird. In diesem Fall würde wiederum ein MiB ungenutzt bleiben. Daher wird in diesem Fall wieder ein MiB dazu addiert. Allerdings berechnet die nichtlineare Optimierung selten wirklich ganzzahlige Werte.

Genau genommen wäre es korrekter einen Algorithmus zur nichtlinearen gemischt-ganzzahligen Optimierung anzuwenden. Aber die Methoden auf diesem Forschungsgebiet sind nach [Hem+09] im Generellen noch nicht so weit wie für die ganzzahlige lineare Optimierung und das Problem ist noch größer als für die nichtlineare Optimierung. Da hier die Komplexität des Problems wesentlich ansteigen würde, wurde auf die Verwendung eines solchen Algorithmus verzichtet. Wenn davon ausgegangen wird, dass in der Optimierung 1 GiB Arbeitsspeicher zur Verfügung steht und in einem Ausführungsplan weniger als 10 speicherabhängigen Operatoren vorhanden sind[2], wird der dadurch zustande kommende Fehler aufgrund der stetigen Funktionen relativ klein bleiben.

In dem Fall, dass ausreichend Arbeitsspeicher vorhanden ist, also die Summe der $max(MinMemoryPerOperator, SufficientMemory_i)$ Werte kleiner oder gleich dem *GlobalMemory* ist, wird die Optimierung erst gar nicht ausgeführt. In diesem Fall wird den x_i unmittelbar der Wert $max(MinMemoryPerOperator, SufficientMemory_i)$ zugewiesen.

Ist bei der Zuweisung noch freier Arbeitsspeicher vorhanden, also der gesamte Arbeitsspeicher ($GlobalMemory$) noch nicht ausgeschöpft, findet eine gleichmäßige Verteilung des übrigen Arbeitsspeichers auf die einzelnen Operatoren statt. Die Verteilung des nicht verwendeten Arbeitsspeichers ist dabei momentan noch relativ wichtig. Aufgrund der konstanten Kostenfunktionen der Operatoren *symmjoin, mergejoin* und *sortmergejoin* wird für diese Operatoren immer nur die minimale Menge an Arbeitsspeicher vergeben. Um diesem Problem etwas entgegen zu wirken, wird der nicht verwendete Arbeitsspeicher auf alle speicherverwendenden Operatoren aufgeteilt. Dieses Verhalten kann mit dem Prädikat *setRestDistribution(+Mode)* beeinflusst werden. Wird als Parameter *Mode* der Wert *none* übergeben, wird keine Verteilung des nicht verwendeten Arbeitsspeicher durchgeführt. Der Wert *equal* ist das Standardverhalten und führt die gerade beschriebene gleichmäßige Verteilung durch.

Diese Verteilung nicht verwendeten Arbeitsspeichers wurde aber nur für den Fall implementiert, dass ein Rahmen ermittelt wurde, also kein blockierender Operator im Ausführungsplan vorhan-

[2] Wenn dies nicht mehr nur Join-Operatoren sind, wäre diese Annahme gegebenenfalls hinfällig.

den ist. Für den Fall, dass mehrere Rahmen vorhanden sind, ist diese Verteilung des nicht vergebenen Speichers nicht mehr so trivial. Dies liegt darin begründet, dass hier die Verteilung des ungenutzten Arbeitsspeichers eines Rahmens nicht unabhängig von den übrigen Rahmen erfolgen kann. Die Ursache hierfür wiederum ist, dass die blockierenden Operatoren nur eine Speicherzuweisung erhalten, diese aber in mehr als einem Rahmen vorhanden sind. Die Problematik könnte auch als ganzzahliges lineares Maximierungsproblem aufgefasst werden. Die Zielfunktion ist hier der vergebene Arbeitsspeicher der Operatoren und die Restriktionen sind Ungleichsfunktionen, die den vergebenen Speicher in einem Rahmen maximieren. Zusätzlich werden als untere Grenzen der Operatoren die schon durch nichtlineare Optimierung ermittelten Arbeitsspeicherwerte festgelegt. Ein offenes Problem ist hier aber, wie eine Gleichverteilung des nicht vergebenen Arbeitsspeichers erreicht werden kann. Um den Rahmen dieser Arbeit aber nicht zu sprengen, wurde auf eine Verteilung des nicht vergebenen Arbeitsspeichers für mehr als einen Rahmen verzichtet. Dies ist insofern momentan noch nicht kritisch, da der einzige blockierende Operator der *sortmergejoin* Operator ist. Dieser wird, wie später im Kapitel der Evalueriung (Kapitel 10 auf Seite 70) beispielhaft zu sehen ist, in der Regel nicht als Join-Operator ausgewählt, sofern der komplette Arbeitsspeicher nicht ausgeschöpft wird.

9.4 Die zweiphasige Arbeitsspeicheroptimierung

Mit der Optimierung eines gegebenen Ausführungsplans, wie es im vorherigen Abschnitt 9.2 beschrieben wurde, ist nun die Realisierung der zweiphasigen Optimierung ein Leichtes. Unverändert bleibt hierzu die Optimierung mit Hilfe des Dijkstra-Algorithmus auf dem Graph der Kostenkanten auf Basis des POG. Damit die Kosten überhaupt berechnet werden können, ist allerdings zuerst eine Wahl der Arbeitsspeicherzuweisungen zu treffen. Diese Wahl dieser initialen Arbeitsspeicherzuweisungen erfolgt mit einer der bereits in Abschnitt 8.6 auf Seite 55 beschriebenen Strategien. Mit diesen Werten werden die Kosten berechnet und anschließend wird der kürzeste Pfad ermittelt. Nachdem der Ausführungsplan in Form eines Pfades feststeht, kann in der zweiten Phase die Speicheroptimierung durch das Lösen des Optimierungsproblems, wie in Abschnitt 9.2 auf Seite 59, beschrieben, durchgeführt werden.

9.5 Die einphasige Arbeitsspeicheroptimierung

Diese Optimierung arbeitet direkt auf dem Graphen der Kostenkanten auf Basis des POG. Die hier zu implementierenden Algorithmen wurden in Abschnitt 8.7 auf Seite 57 diskutiert. Die Ermittlung des besten Pfades über das Aufzählen aller Pfade wird von dem Prädikat *bestPathByEnumeration/0* geleistet. Dort werden alle Pfade durch das Prädikat *enumeratePaths/3* aufgezählt und an das Prädikat *processResult/2* abgegeben, indem der aufgezählte Pfad verarbeitet wird. Dazu wird das Prädikat die Speicheroptimierung des Ausführungsplans, der durch den Pfad angegeben ist, ausführen [Vgl. Abschnitt 9.2 auf Seite 59]. Anschließend werden die Kosten des bisher besten Plans mit den Kosten des aktuell betrachteten Pfades verglichen. In dem Fall, dass die Kosten des betrachteten Pfades günstiger sind, wird der als bisher bester Plan gespeicherte Pfad durch diesen ersetzt. Am Ende liegt dann der kürzeste Pfad, der auch den Arbeitsspeicher einbezieht, in dem Faktum *currentShortestPath/3* vor.

Die Implementierung des modifizierten Dijkstra-Algorithmus befindet sich im Prädikat *bestPathByModifiedDijkstra(-Path, -Costs)*. Hier wird im Wesentlichen die Implementierung der

bereits im Optimierer vorhandenen Dijkstra-Implementierung verwendet. Die einzige und wesentliche Änderung ist, dass bei der Betrachtung neuer Kanten im Prädikat *successor/2* die Kosten der jeweiligen Kante, wie schon in Abschnitt 8.7 auf Seite 57 beschrieben, ermittelt wird.

9.6 Berücksichtigung blockierender Operatoren

In dem Artikel [ND98] wurden Bereiche getrennt, die durch blockierende Operatoren begrenzt werden, die dort *Shelfs* genannt wurden. Hier soll dafür der deutsche Begriff *Rahmen* verwendet werden. Innerhalb eines Rahmens kann dabei durch die Begrenzung von blockierenden Operatoren, bzw. dem Anfang oder Ende des Ausführungsplans, der gesamte Arbeitsspeicher aufgeteilt werden. Sind die Rahmen identifiziert, können damit, wie bereits im Abschnitt *Optimierung eines gegebenen Ausführungsplans* (Abschnitt 9.2 auf Seite 59) gezeigt wurde, die entsprechenden Restriktionsungleichungen aufgebaut werden.

Zum Ende eines Ausführungsplans wird immer entweder ein Tupelstrom in eine Relation ausgegeben oder ein atomarer Wert, wie dieser beispielsweise durch den *count*-Operator erzeugt wird. Um die Rahmen zu ermitteln wird von dem Ende eines Ausführungsplan dieser rückwärts analysiert. Dabei gehören alle Operatoren, die bis zum Auftauchen blockierender Operatoren oder dem Ende des Ausführungsplan vorkommen, zu einem Rahmen. Bei Join-Operatoren ist dabei zu beachten, dass in dem Fall in alle Eingabeströme verzweigt wird. Der blockierende Operator selbst gehört zu beiden Rahmen, da hier Arbeitsspeicher benötigt wird, der in beiden Rahmen existiert. Damit ist am Ende bekannt, zu welchen Rahmen welche Operatoren gehören. Zu beachten ist hier die Tatsache, dass einem blockierenden Operator, der die Schnittstelle zwischen zwei Rahmen bildet, auch nur ein Wert für den Arbeitsspeicher zugeteilt werden kann, der in beiden Rahmen identisch ist.

Die Implementierung der Prädikate zum Erstellen der Restriktionen für den zu verteilenden Arbeitsspeicher befinden sich in der Datei *blockingops.pl*. Das wesentlichste Prädikat ist *createGlobalMemoryConstraints(+_, +Path, -Constraints)*. Dieses Prädikat erstellt die Restriktionsungleichungen in der Form einer Liste als Ausgabeparameter *Constraints*. Die Elemente der Liste *Constraints* selbst sind auch wieder Listen mit $n + 1$ Elementen, wenn insgesamt in dem Ausführungsplan n Arbeitsspeicher verwendende Operatoren in diesem Plan existieren. Das erste Element beinhaltet dabei den gesamten Arbeitsspeicher, der für diesen Rahmen vergeben werden kann. Die nächsten n Elemente der Liste beinhalten für die i-te[3] Position, ob der Operator x_i des Optimierungsproblems in dem Rahmen enthalten ist. Der Wert 1 markiert hierbei, dass der Operator Speicher in diesem Rahmen benötigt. Eine 0 bedeutet, dass der Operator in dem Rahmen nicht vorhanden ist.

Im Übrigen soll noch hervorgehoben werden, dass jeder Operator zwingend in mindestens einer Restriktionsungleichung vorhanden sein muss und das in der Implementierung auch gewährleistet ist.

Für die Evaluierung existiert das Prädikat *setUseBlockingOpOptimization/1* das durch den Parameter *on* oder *off* die Berücksichtigung blockierender Operatoren aktiviert bzw. deaktiviert. Im Falle der Deaktivierung wird nur eine Restriktionsungleichung dem aufrufenden Prädikat zur Verfügung gestellt. Diese besagt, dass der gesamte Arbeitsspeicher auf alle Operatoren aufgeteilt werden muss.

Die Berücksichtigung der blockierenden Operatoren ist in der derzeitigen Implementierung nicht möglich in Verbindung mit dem modifizierten Dijkstra-Algorithmus. Da hier während der Ermitt-

[3] Das erste Element besitzt in der Liste den Index 0, im Optimierungsproblem wird erst ab 1 angefangen zu zählen.

lung des kürzestens Pfades im *predicate order graph* im Graphen der Kostenkanten nur Teile des vollständigen Ausführungsplans bekannt sind, ist auch nicht klar welche Rahmen für den Teilpfad aufzubauen sind, da der restliche Pfad gerade erst ermittelt wird. Diese Problematik wurde aber nicht weiter fokussiert. Diese benötigte Vorausschau macht noch mal deutlich, dass ein Greedy Algorithmus, wie Dijkstra's Algorithmus, der unmittelbar versucht, die lokal beste Entscheidung zu treffen, das Problem der Arbeitsspeicheroptimierung in einer Phase wohl nicht lösen kann. Sowohl die Verteilung des Arbeitsspeichers auf die einzelnen Operatoren als auch die Ermittlung der Rahmenbedingungen benötigen die Betrachtung des kompletten Pfades.

Um den Vorteil der blockierenden Operatoren in einem Ausführungsplan, der durch den modifizierten Dijkstra-Algorithmus ermittelt wurde, trotzdem noch integrieren zu können, wird, nachdem der kürzeste Pfad ermittelt wurde, noch einmal eine separate Optimierung des Pfades durchgeführt, diesmal aber mit der Berücksichtigung blockierender Operatoren, da nun der komplette Pfad bekannt ist.

9.7 Abweichungen zur Optimallösung

Durch das Aufzählen aller Pfade konnte, wenn auch mit hohen Kosten, zumindest garantiert werden, den besten Ausführungsplan, der auch die Ressource Arbeitsspeicher berücksichtigt, mit der Strategie *enumerate* ermitteln zu können. Dazu wurden auch ein paar kleine Abweichungen in Kauf genommen, die technischer Natur (1 MiB Schritte, Gleitkommaarithmetik) waren oder auch, wenn gute Gründe vorlagen (Aufrunden der Gleitkommazahlen aufs nächste volle MiB). Dieser so ermittelte Plan ist dann rechnerisch auf Basis aller Eingabeparameter auch in der Tat der beste Ausführungsplan. Allerdings gibt es einige Faktoren, die dazu führen, dass durch die Eingabewerte in die Arbeitsspeicheroptimierung der Operatoren am Ende nicht der wirklich beste Ausführungsplan ermittelt wird.

1. Selektivität: Auf Basis der Selektiviät wird im POG die Anzahl der Tupel geschätzt, die als Eingabe für die nächste Operation dienen. Je weiter die geschätzte Selektivität von der realen Selektivität abweicht, umso größer werden auch die Abweichungen der Kostenschätzung der Kostenfunktionen aus der *CostEstimation* Schnittstelle. Allerdings ist das Problem der Selektivtät ein eigenes, nicht triviales Themengebiet, das hier nicht weiter berücksichtigt werden kann.

2. Tupelgrößen dynamischer Datentypen: Die Kostenfunktion der *CostEstimation* Schnittstelle wird auch auf Basis der durchschnittlichen Tupelbreiten ermittelt. Allerdings ist die Tupelbreite nicht für jedes Tupel gleich, wenn Attribute mit dynamischen Datentypen vorhanden sind. Nach der Anwendung von WHERE-Bedingungen im POG in Form von *filter*- oder *join*-Operationen könnte daher die durchschnittliche Tupelgröße nicht mehr stimmen. Die Folge ist dann, dass die die Kostenfunktionen auf Basis zu kleiner oder zu großer Tupelgrößen ermittelt werden.

3. Kostenfunktionen: Die Kostenfunktionen an sich sind schon Approximationen des erwarteten realen Verhaltens der Operatoren und weichen daher auch von diesen ab. Diese Abweichung können dann auch dazu führen, dass die Optimierung der Arbeitsspeicherzuweisungen für die Operatoren andere Werte als optimal ansieht.

4. Konstanten der Kostenfunktionen:

   Die Kostenfunktion setzt sich unter anderem auch aus hardwarespezifischen Konstanten zusammen. Diese Konstanten der Kostenschätzung können mit dem Script *UpdateProgressConstants.sec* neu ermittelt werden. Die so ermittelten Konstanten werden dann durch dieses Skript in der Datei *ProgressConstants.csv* abgelegt. Auf Basis dieser hardwareabhängigen Konstanten werden die Kostenfunktionen erstellt. Für eine andere Hardware ist daher davon auszugehen, dass andere Ergebnisse zustande kommen. Allerdings sind diese dann auch für die entsprechende Hardware korrekter. Mehr zu dem Skript und den so ermittelten Konstanten kann [Nid12, Abschnit 5.6] entnommen werden.

   Im Laufe der Evaluierung in Kapitel 10 auf Seite 70 zeigten sich hier weitere Probleme, denn teilweise wurden Konstanten berechnet, die negativ sind. Dies ist insofern nicht richtig, da diese Werte Ausführungszeiten für bestimmte Operationen angeben. Diese negativen Werte führen dann dazu, dass Kostenfunktionen berechnet werden, die mit zunehmendem Arbeitsspeicher ansteigen. Sie modellieren damit nicht mehr das reale Verhalten der Operatoren. Daher führen hier falsche Werte oder auch Abweichungen zu anderen Ergebnissen in der Speicheroptimierung.

   Die in Secondo enthaltene Datei *ProgressConstants.csv* verursacht in der derzeiten Version auch noch Probleme. Bei bestimmten Konstellationen zu den Kardinalitäten und Tupelbreiten werden Kostenfunktionen in Abhängigkeit des Arbeitsspeichers erzeugt, die mit mehr zugewiesenem Arbeitsspeicher höhere Kosten aufweisen. Festgestellt werden konnte dies bisher zumindest für den *hybridhashjoin* Operator. Eine abschließende Klärung dieser Problematik war hier bis zur Abgabe aber nicht mehr möglich. Daher wurde in der Implementierung nach Ermitteln der Kostenfunktionen eine Änderung vorgenommen. Werden dabei Kostenfunktionen erkannt, die in Abhängigkeit des Arbeitsspeicher ansteigen, wird diese Kostenfunktion nicht übernommen. In diesem Fall wird auf die konstante Funktion zurückgegriffen, die immer den Wert für die Ausführungszeit bei 16 MiB zugewiesenem Arbeitsspeicher liefert (TimeAt16MB, Vgl. Abschnitt 8.3 auf Seite 51).

5. Runden der Ergebnisse der nichtlinearen Optimierung: Vgl. Abschnitt 9.2 auf Seite 59

6. Abweichungen in den Kostenmodellen: Vgl. Abschnitt 8.3 auf Seite 51

7. Nichtlineare gemischt-ganzzahlige Optimierung: Vgl. Abschnitt 8.4 auf Seite 53

8. Abbruchkriterium der nichtlinearen Optimierung: Vgl. Abschnitt 9.2 auf Seite 59

9. Die Vergabe von Arbeitsspeicher erfolgt in 1 MiB Schritten: Dies ist bedingt durch die Implemtierung der Zuweisung von Arbeitsspeicher an Operatoren im Secondo DBMS. Hierdurch können auch Abweichungen resultieren, da ggf. pro Operator bis zu einem MiB an Arbeitsspeicher ungenutzt bleibt. Allerdings kann davon ausgegangen werden aufgrund der stetigen Kostenfunktionen, dass die Abweichungen nicht groß sind.

10. Minimum von 16 MiB an Arbeitsspeicher, der einem Operator zugewiesen wird: Für Operatoren, die wesentlich weniger Arbeitsspeicher benötigen, wird hier bis zu 16 MiB zuviel an Arbeitsspeicher vergeben, der dann anderen Operatoren nicht mehr zur Verfügung steht. Allerdings birgt dieser Minimalwert einen Vorteil bei Fehlerschätzungen. Durch die Selektivitäten beispielsweise erhöhen sich die Kosten nicht um ein Vielfaches. Insbesondere modellieren

die linearen Kostenfunktionen nicht adäquat das Verhalten der Operatoren in Abhängigkeit vom Arbeitsspeicher, wenn sich dem Minimum von einem MiB angenähert wird, aber der Operator mehr als 16 MiB gebrauchen könnte.

9.8 Test und Robustheit

Die Implementierung der Arbeitsspeicheroptimierung ist im Gegensatz zu der Behandlung der geschachtelten Relationen ein wesentlich abgeschlosseneres System. Daher ist es insbesondere auch besser zu testen und verhält sich wesentlich robuster. Die Anzahl der Testfälle ist wesentlich geringer ausgefallen als für die Behandlung der geschachtelten Relationen. Solange die vom Optimierer erstellten Ausführungspläne nur Operatoren beinhalten, für die auch entsprechende *cost* Prädikate in der Datei *ma_improvedcosts.pl* vorhanden sind sowie die Kostenfunktion der *CostEstimation*, die die in Abschnitt 8.4 auf Seite 53 aufgestellten Vorraussetzungen erfüllen, wird diese Erweiterung einen Ausführungsplan erstellen, der nach dem dort eingestellten Algorithmus die Ressource Arbeitsspeicher einbezieht und aus Sicht des Algorithmus diesbezüglich optimal ist.

Für das Testen der nichtlinearen Optimierung wurden *testNLOPT/5* Fakten implementiert, die folgendenden Aufbau besitzen:

```
1 testNLOPT(+No, +Vars, +Formula, +Constraints, +SufficientMemory).
```

Die Bedeutung der einzelnen Werte:

- *No*: Eine beliebige Nummer, die ein *testNLOPT/5* Faktum eindeutig identifiziert.
- *Vars*: Eine Liste von Prolog-Variablen die anschließend in den Formeln verwendet werden können.
- *Formula*: Die Liste der einzelnen Kostenfunktionen. Es müssen dabei hier genauso viele Formeln angegeben werden wie Elemente in der Liste *Vars* vorhanden sind. Diese müssen auch in der gleichen Reihenfolge verwendet werden und in jeder Formel darf nur eine Variable verwendet werden. Diese eine Variable darf allerdings in einer Formel merhmals verwendet werden. In der Zielfunktion werden diese einzelnen Formeln dann additiv verknüpft.
- *Constraints*: Dies ist eine Liste von Rahmen. Das erste Element eines Rahmens beinhaltet dabei die Menge an Arbeitsspeicher, die in diesem Rahmen vergeben werden kann. Dann folgen soviele weitere 0 oder 1 Werte, wie es Variablen gibt. Dabei muss jeder Variablen hier zumindest einmal eine 1 zugewiesen werden. Im Wesentlichen werden die Rahmen hier spezifiziert, wie es in Abschnitt 9.6 auf Seite 65 beschrieben wurde.
- *SufficientMemory*: Eine Liste mit den Arbeitsspeicherwerten für die einzelnen Variablen, die höchstens zugewiesen werden, da ab hier keine Verbesserung mehr erreicht werden kann.

Ein Beispiel auf Basis der verwendeten Kostenfunktionen aus Abschnitt 8.2 auf Seite 48 für ein solches Faktum wäre:

```
1 testNLOPT(102, [X, Y], [800-1.5*X, (10000/Y)+100], [[512, 1, 1]], [512, ↩
    512]).
```

Durch den Aufruf des Prädikates *testNLOPT(+No)* kann anhand der Nummer die nichtlineare Optimierung durchgeführt werden. Für das obigen Beispiel wäre folgende Ausgabe das Resultat:

```
1 1 ?- testNLOPT(102).
2
3
4 Variables        : [_G377,_G380]
5 Formula          : 800-1.5*_G377+ (10000/_G380+100)
6 Constraints      : [[512,1,1]]
7 SufficientMemory: [512,512]
8
9 NLOPT-Result: [429.5852153130383,81.64968252308647]
10 Time         : 1ms
11 Total costs : 378.09662752414164
12 true.
```

In der neunten Zeile wurde durch das *testNLOPT/1* Prädikat die Lösung des Optimierungsproblems ausgegeben. Die Reihenfolge entspricht dabei der Reihenfolge der Variablen der *Vars* List des *testNLOPT/5* Faktums. Dadurch dass die nichtlineare Optimierung das Ergebniss nicht bis auf eine sehr große Genauigkeit berechnet, wird die Lösung in der Regel[4] schnell berechnet werden können. Selbst bei 20 zu ermittelnden Variablen liegt die nichtlineare Optimierung noch im Bereich von unter 100 Millisekunden.

[4] Garantiert werden kann dies aber nicht, denn die Laufzeit ist problemabhängig und hängt nicht nur von der Anzahl der Parameter ab.

Kapitel 10

Evaluierung

Im Abschnitt 9.7 auf Seite 66 wurde bereits erwähnt, dass rechnerisch für den Auszählungsalgorithmus gewährleistet ist, auch den besten Ausführungsplan auf Basis der prognostizierten Kosten finden zu können, der auch den verfügbaren Arbeitsspeicher berücksichtigt. Die anderen Algorithmen oder die Heuristik des modifizierten Dijkstra-Algorithmus können dies nicht, weshalb in diesem Abschnitt evaluiert werden soll, wie sich diese Methoden zueinander verhalten. Dazu gehört auch, in wie weit die Phase der Optimierung sich verteuert hat und welchen Effekt die implementierte Optimierung überhaupt im Gegensatz zur Anfrageoptimierung ohne Betrachtung des Arbeitsspeichers besitzt.

Die derzeitigen Kostenfunktionen der Operatoren *symmjoin, mergejoin* und *sortmergejoin* sind konstant. Die Operatoren liefern zwar die Menge an Arbeitsspeicher, die sie für ihre optimale Ausführung benötigen, aber dies spiegelt sich nicht in den Kostenfunktionen wieder, da gerade diese konstant sind. Insofern wird diesen Operatoren nur mehr Arbeitsspeicher als der minimale Arbeitsspeicher zugewiesen, wenn noch freier Arbeitsspeicher übrig bleibt, den kein anderer Operator mehr benötigt. Aus Sicht des nichtlinearen Optimierungsproblems besteht hier aufgrund der konstanten Kostenfunktionen kein Grund diesen Operatoren mehr Arbeitsspeicher zu gewähren. Dies liegt daran, dass die Gesamtkostenfunktion sich nur verringert, indem Operatoren mehr Arbeitsspeicher zugewiesen wird, bei denen sich die Kosten durch mehr zugewiesenen Arbeitsspeicher weiter senken lassen. Natürlich sinken die Kosten der Operatoren bei mehr verfügbarem Arbeitsspeicher bis zu einer gewissen Grenze, ansonsten gäbe es keine Veranlassung diesen überhaupt mehr Arbeitsspeicher zuzuweisen. Die Kostenfunktionen modellieren diesen Aspekt aber derzeit nicht adäquat. Für eine tiefergehende Betrachtung zu der Problematik der Kostenfunktionen sei auf [Nid12] verwiesen.

Der einzige blockierende Operator der eine Implementierung der *CostEstimation* Schnittstelle besitzt, ist der *sortmergejoin* Operator. Allerdings besitzt auch dieser nur konstante Kostenfunktionen in Abhängigkeit vom Arbeitsspeicher. Aufgrund der verhältnismäßig hohen geschätzten Kosten gegenüber den geschätzten Kosten der anderen Join-Operatoren wird dieser Operator in der Regel nicht in den erzeugten Ausführungsplänen des Optimierers auftauchen, sofern genug Arbeitsspeicher vorhanden ist. Durch die Optimierer-Option *noHashjoin* kann die Verwendung der *HashJoin* Operatoren ausgeschlossen werden. Allerdings hilft dies auch nicht, die Erweiterung um die Ausnutzung der Rahmen, die durch blockierende Operatoren angezeigt werden, zu evaluieren. Aufgrund der konstanten Kostenfunktionen, gibt es für die nichtlineare Optimierung keinen Grund dem Operator mehr Arbeitsspeicher zu gewähren und somit hier zu profitieren.

Der Optimierer berücksichtigte bisher nicht die Operatoren *gracehashjoin, hybridhashjoin* und *itHashJoin* für die möglichen Ausführungspläne. Da allerdings nur die Kostenfunktionen dieser Operatoren nicht konstante Kostenfunktionen bereitstellen, wurde der Optimierer um die Funkion erweitert, auch diese Operatoren bei der Optimierung zu berücksichtigen. Realisiert wurde dies

durch neue Übersetzungsregeln des Optimierers mit dem *=>/2* Operator. Implementiert wurde dies allerdings nicht für den *itSpatialJoin*, da für diesen Operator nicht ganz klar war, wie die entsprechenden Übersetzungsregeln zu implementieren sind. Sobald aber Ausführungspläne mit diesem Operator erzeugt würden, würde die Optimierung des Arbeitsspeichers diesen sofort berücksichtigen.

Da die Verwendung dieser drei Operatoren neu im Optimier ist, existieren hierfür keine implementierten Kostenfunktionen in den Kostenmodellen *improvedcosts* und *standardcosts*. Damit gestaltet sich ein Vergleich der Ausführungspläne zwischen den durch diese neue Erweiterung erzeugten Plänen, und den durch die beiden gerade genannten Kostenmodellen erzeugten Plänen schwierig, da nur das Kostenmodell der *ma_improvedcosts* die drei Operatoren *gracehashjoin, hybridhashjoin* und *itHashJoin* verwendet.

In Abschnitt 8.4 auf Seite 53 wurde bereits erwähnt, dass bisher der Arbeitsspeicher im SECONDO DBMS auf alle Operatoren eines Ausführungsplans, die Arbeitsspeicher verwenden, gleichmäßig aufgeteilt wurden. Diese Art der Speicheroptimierung wurde auch als eine mögliche Strategie *staticEqual* implementiert. Damit kann, wenn die Erweiterung der Arbeitsspeicheroptimierung aktiviert ist, ein Ausführungsplan ermittelt werden der das bisherige Verfahren simuliert, aber die Verwendung der Operatoren *gracehashjoin, hybridhashjoin* und *itHashJoin* erlaubt da in diesem Fall die Kosten über das Kostenmodell *ma_improvedcosts* ermittelt werden können. Der Ausführungsplan selbst, der auf Basis des POG ermittelt wird, wird in diesem Fall mit 16 MiB für jeden Operator ermittelt. Anschließend wird die Gleichverteilung des gesamten Arbeitsspeichers (*GlobalMemory*) auf alle Operatoren des Ausführungsplans vorgenommen. Der Sinn dieser Strategie liegt darin, in etwa abschätzen zu können, wie die Ergebnisse der Speicheroptimierung sich zu der bisherigen Strategie verhält. Denn einerseits sind Tests mit den Operatoren, die bisher im Optimierer verwendet werden (*sortmergejoin, mergejoin und symmjoin*) nicht möglich, da hier konstante Kostenfunktionen geliefert werden, und anderseits sind die interessanten Operatoren (*gracehashjoin, hybridhashjoin, itHashJoin*) nur in Verbindung mit dem neu implementierten Kostenmodell der *ma_improvedcosts* möglich.

Damit werden in den Testanfragen folgende neun Strategien evaluiert, die hier zusammenfassend knapp skizziert werden sollen:

1. *staticEqual*: Ermittelt auf Basis des POG mit 16 MiB pro Operator den besten Plan und teilt anschließend den verfügbaren Speicher (256 MiB) auf alle Operatoren gleichmäßig auf. Diese Strategie wird als Referenz verwendet, um die Ergebnisse der anderen Strategien zu bewerten.

2. *static 16 MiB*: Ermittelt auf Basis des POG mit 16 MiB pro Operator den besten Plan und anschließend wird auf dem so ermittelten Ausführungsplan die Speicheroptimierung durchgeführt (beschrieben in Abschnitt 8.6 auf Seite 55).

3. *static 256 MiB*: Ermittelt auf Basis des POG mit 256 MiB pro Operator den besten Plan und anschließend wird auf dem so ermittelten Ausführungsplan die Speicheroptimierung durchgeführt (beschrieben in Abschnitt 8.6 auf Seite 55).

4. *staticMaxDivOpCount(X) Y MiB*: Zuerst wird hier ermittelt, wieviele speicherabhängige Operatoren maximal in einem Ausführungsplan vorhanden sind. Diese Anzahl ist Wert X. Anschließend wird der verfügbare Arbeitsspeicher durch X geteilt (das ist der Wert Y) und auf Basis des POG mit Y MiB pro Operator der Ausführungsplan ermittelt. Als letztes wird auf

dem so ermittelten Ausführungsplan die Speicheroptimierung durchgeführt (beschrieben in Abschnitt 8.6 auf Seite 55).

5. *staticMaxDivPOG(X) Y MiB*: X ist hier die Anzahl der WHERE-Bedingungen in der SQL-Anfrage. Anschließend wird der verfügbare Arbeitsspeicher durch X geteilt (das ist dann der Wert Y) und auf Basis des POG mit Y MiB pro Operator der Ausführungsplan ermittelt. Als letztes wird auch hier wieder auf dem so ermittelten Ausführungsplan die Speicheroptimierung durchgeführt (beschrieben in Abschnitt 8.6 auf Seite 55).

6. *modifiedDijkstra*: Ermittelt den kürzesten Pfad im Graphen der Kostenkanten auf Basis des POG mit dem modifizierten Dijkstra-Algorithmus. Am Ende wird noch eine separate Optimierung des Ausführungsplan durchgeführt, um dann auch die blockierenden Operatoren berücksichtigen zu können (beschrieben in Abschnitt 8.6 auf Seite 55).

7. *enumerate*: Zählt alle möglichen Ausführungspläne auf und optimiert jeden dieser Ausführungspläne, um den günstigsten Pfad zu ermitteln (beschrieben in Abschnitt 8.6 auf Seite 55).

8. *improvedcosts*: Ermittelt für die Anfrage den Ausführungsplan, der mit dem bereits im Secondo implementieren Kostenmodell *improvedcosts* als bester Plan angesehen wird.

9. *standardcosts*: Ermittelt für die Anfrage den Ausführungsplan, der mit dem bereits im Secondo implementieren Kostenmodell *standardcosts* als bester Plan angesehen wird.

Damit ist es nun möglich in der Evaluierung folgende Ergebnisse gegenüberzustellen:

- Unterschiede der Ergebnisse zwischen den verschiedenen Algorithmen der Arbeitsspeicheroptimierung.
- Unterschiede der Ergebnisse zwischen den Ergebnissen der Optimierung und dem bisherigen Verhalten der Optimierung (*staticEqual*).
- Unterschiede der Ergebnisse zu der Lösung, die der Optimierer ohne diese Erweiterung bisher generieren würden. Aufgrund der fehlenden Möglichkeit der anderen Kostenmodelle die Operatoren *gracehashjoin, hybridhashjoin* und *itHashJoin* zu verwenden, werden hier aber ggf. ganz andere Ausführungspläne erzeugt. Ein fairer Vergleich ist hier zwar nicht möglich, allerdings ist die Information nicht uninteressant, so dass auch dieses Ergebnis ermittelt wird.
- Vergleich zwischen den prognostizierten Kosten der jeweiligen Strategie auf Basis des verwendeten Kostemodells und den tatsächlich gemessenen Kosten, also der Ausführungszeit selbst.

Die Testdatenbank

Basis der Testläufe ist eine Datenbank[1] der Geofabrik[2], die Geographische Daten des Bundeslandes Nordrhein-Westfalen beinhaltet. Die Datenbasis der Datenbank stammt aus dem OpenStreetMap Projekt[3]. Die Daten können unter dem Link `http://download.geofabrik.de/openstreetmap/`

[1] Stand vom 29.10.2012
[2] `http://www.geofabrik.de`
[3] `http://www.openstreetmap.org`

`europe/germany/nordrhein-westfalen.shp.zip` heruntergeladen werden und stehen unter der ODbL-Lizenz 1.0[4]. In das SECONDO DMBS wird die Datenbank durch ein von Herrn Prof. Dr. Güting bereitgestelltes Script eingeladen. Allerdings wurde das Script modifiziert, so dass die *counter* Attribute nicht mehr entfernt werden. Damit existiert in jeder der Relationen ein Attribut *No*, das im ersten Tupel den Wert 1 beinhaltet und im zweiten Tupel den Wert 2 usw. Dies ermöglicht nach einer Join-Operation weitere relativ große Ergebnisse zu erzeugen, die als Eingabe für weitere Join-Operatoren dienen. Beide Scripte sind im Anhang B zu finden. Tabelle 10.1 zeigt die dort vorhandenen Relationen und deren Datenmengen. Relationen die mit „00k“ enden, beinhalten nur einen Auschnitt der eigentlichen Relation. Also die Relation *Buildings500k* beinhaltet nur die ersten 500 000 Datensätze der Relation *Buildings*. Hierdurch können in den Testfällen Relationen mit verschiedenen Datenmengen verwendet werden und es treten keine Caching Effekte dadurch auf, dass Daten immer nur aus denselben Relationen stammen.

Tabelle 10.1: Datenmengen

Relation	Anzahl Tupel	Durchschnittliche Tupelbreite im Arbeitsspeicher in Byte	Datenmenge in MiB
Buildings	1 795 700	187	320
Buildings200k	200 000	187	36
Buildings300k	300 000	187	54
Buildings400k	400 000	187	71
Buildings500k	500 000	187	89
Natural	89 882	187	16
Points	295 819	141	40
Railways	45 313	254	11
Roads	1 122 040	283	303
Roads200k	200 000	276	53
Roads300k	300 000	277	79
Roads400k	400 000	278	106
Roads500k	500 000	279	133

Selektivitäten

Die Selektivitäten werden im SECONDO DMBS auf Basis kleiner Relationen, den sogenannten Samples, ermittelt. Für eine Relation stellt ein Sample einen Ausschnitt aus dieser Relation dar. Genaue Details hierzu können beispielsweise [SOM12] entnommen werden. Allerdings zeigte sich für diese hier erfolgten Testfälle, dass die auf dieser Basis ermittelten Selektivitäten nicht geeignet für weitere Analysen waren. Mit den Standardeinstellungen im Optimierer mit denen die Samples erstellt wurden, wurden so kleine Werte für die Selektivitäten gewählt, dass der Optimierer von annähernd leeren Zwischenergebnissen ausging. In der Speicheroptimierung führen falsche Selektivitäten allerdings zu noch größeren Abweichungen von der optimalen Lösung, da dann falsche optimale

[4] `http://opendatacommons.org/licenses/odbl`

Arbeitsspeicherangaben berechnet werden. Um die Evaluierung aber losgelöst von der Problematik der Selektivitäten durchzuführen, wurde ein Prädikat *maCreateTestSels/0* für den Test implementiert, das die korrekten Selektivitäten als *storedSel/3* Fakten hinzufügt. Der Grund dafür, dass hier in diesen Fällen die Selektivitäten über die Samples keine guten Werte lieferten, liegt daran, dass diese Samples auf zufällig ausgewählten Tupeln der Basisrelation basieren. Da hier aber die Nummeriung der Tupel alles andere als zufällig ist, sind die Samples zu klein dimensioniert um zumindest annähernd realistische Selektivitäten zu ermitteln.

Zuerst noch ein paar Rahmenbedingungen, unter denen die Tests ausgeführt wurden:

- Als maximal zur vergebener Arbeitsspeicher (GlobalMemory) wird 256 MiB gewählt.
- Die angegebenen Laufzeiten wurden über 4 Testläufe ermittelt und daraus das arithmetische Mittel berechnet. Vorher wurde ein separater Testlauf durchgeführt, der nicht in das arithmetische Mittel einfließt und nicht durch bisher nicht verwendete Puffer oder Caches die Ergebnisse verfälscht. Die Ausführungszeiten der Ausführungspläne sind auf zwei Stellen kaufmännisch gerundet in Sekunden angegeben. Diese Zeit entspricht dem Attribut *ElapsedTime* der *SEC2COMMANDS* Relation, in der im Secondo DBMS die Ausführungszeiten festgehalten werden.
- Konfigurationsmerkmale des Rechners, auf dem die Tests ausgeführt wurden:
  - Prozessor: 2x Intel(R) Core(TM) 2 Duo CPU E8335 @ 2.93GHz
  - Betriebssystem: Ubuntu 12.04.1 LTS
  - Festplatte: Western Digital WD6400AAKS 640 GB Caviar Blue 7200rpm 16MB SATA II
- Aufgrund in der Abschnitt 9.7 auf Seite 66 angesprochenen Problematik der Konstantenermittlung wurde für diese Tests die im Secondo DBMS enthaltete Datei *ProgressConstants.csv* verwendet. Allerdings beschränkt diese Einschränkung die Aussagekraft der Ergebnisse. Abweichungen der Konstanten vom realen Verhalten des Testsystems allerdings sollten in weniger korrekten Kostenfunktionen resulieren, so dass dadurch im Endeffekt (von andere Faktoren abgesehen wie die Selektivitäten etc.) nur schlechtere Ausführungspläne ausgewählt werden. Wenn hier also eine Verbesserung festgestellt wird, sollten die Ausführungspläne bei korrekten Konstanten mindestens so gut sein. Die verwendete Datei *ProgressConstants.csv* für diese Tests ist im Anhang B zu finden.

10.1 Fall 1

Als erstes wird eine Anfrage (10.1) betrachtet, in der die Operatoren mit dem zugewiesenen Arbeitsspeicher auskommen. Hier tritt daher keine Speicheroptimierung über die Lösung eines nichtlinearem Optimierungsproblems auf. Die Ergebnisse der Strategien dieser Anfrage sind in Tabelle 10.2 auf der nächsten Seite aufgelistet. Die von den einzelnen Operatoren angeforderten Arbeitsspeichermengen (Spalte A) werden zusammen mit den zugewiesenen Arbeitsspeicherwerten (Spalte Z) in Tabelle 10.3 auf Seite 76 aufgelistet da erstere Werte nicht mehr in den Ausführungsplänen zu erkennen sind. Für die Strategien *improvedcosts* und *standardcosts* sind nur die zugewiesen Arbeitsspeicherwerte angegeben, da für diese Strategien keine Werte für die optimalen

Arbeitsspeicherwerte bekannt sind. Die entsprechenden Ausführungspläne, die durch die jeweilige Strategie erzeugt wurden, sind im Listing 10.2 aufgeführt.

Listing 10.1: Anfrage auf zwei Relationen, ausreichender Arbeitsspeicher

```
1 select *
2 from   [points as p, railways as r]
3 where  [p:no=r:no]
```

Tabelle 10.2: Ergebnisse zur Anfrage 10.1

Strategie	Kosten in ms	Opt.-Zeit in ms	Ausführungs-zeit in s
staticEqual	8159	13	13,05
static 16 MiB	8159	26	12,69
static 256 MiB	8159	25	12,98
staticMaxDivOpCount(1) 256 MiB	8159	45	13,29
staticMaxDivPOG(1) 256 MiB	8159	27	12,82
modifiedDijkstra	8159	49	12,85
enumerate	8159	65	13,21
improvedcosts	28 364	5	13,38
standardcosts	1 062 786	5	13,13

Tabelle 10.3: Arbeitsspeicherwerte zur Anfrage 10.1

Strategie	Arbeitsspeicher in MiB Operator 1	
	A	Z
staticEqual	16	256
static 16 MiB	16	256
static 256 MiB	16	256
staticMaxDivOpCount(1) 256 MiB	16	256
staticMaxDivPOG(1) 256 MiB	16	256
modifiedDijkstra	16	256
enumerate	16	256
improvedcosts		256
standardcosts		256

Listing 10.2: Ausführungspläne zur Anfrage 10.1

```
1  # staticEqual:
2  Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
3  # static 16 MiB:
4  Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
5  # static 256 MiB:
6  Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
7  # staticMaxDivOpCount(1) 256 MiB:
8  Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
9  # staticMaxDivPOG(1) 256 MiB:
10 Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
11 # modifiedDijkstra:
12 Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
13 # enumerate:
14 Points  feed{p} Railways  feed{r} gracehashjoin[No_p, No_r, 99997]{memory
       256}{3.3770650296296046e-6, 1.5392e-5} consume
15 # improvedcosts:
16 Railways  feed{r} Points  feed{p} hashjoin[No_r, No_p,
      99997]{3.3770650296296046e-6, 1.5392e-5} consume
17 # standardcosts:
18 Points  feed{p} Railways  feed{r} hashjoin[No_p, No_r,
      99997]{3.3770650296296046e-6, 1.5392e-5} consume
```

Auswertung

Bis auf den Strategien *improvedcosts* und *standardcosts* erzeugen alle Strategien die gleichen Ausführungspläne. Dazu wählen diese für die Join-Operation den *gracehashjoin* Operator aus. Zugewiesen wird dem Operator in allen Fällen 256 MiB an Arbeitsspeicher, verlangt hat der Operator in allen Fällen 16 MiB an Arbeitsspeicher. Die Strategien *improvedcosts* und *standardcosts* verwenden den *hashjoin* Operator als effizientesten Join-Operator für die Join-Operation. Zeitlich bewegt sich die Ausführungszeit allerdings für alle Fälle in demselbem Rahmen. Diese Unterschiede sind dabei auch noch im Rahmen der Schwankungen der einzelnen Zeiten, so dass aus diesen Ergebnissen kein verwendbares Result gezogen werden kann. Allerdings zeigt die Zeit für die Erzeugung der Ausführungspläne schon Unterschiede. Zu erkennen ist, dass die Kosten für die hier implementieren Strategien höher ausfallen. Der Grund für die höheren Kosten ist darauf zurückzuführen, dass die Kosten aufwendiger über die *CostEstimation* Schnittstelle ermittelt werden und auch der Graph auf dem der kürzeste Pfad ermittelt wird, durch die neuen Übersetzungsregeln mehr Kostenkanten enthält.

Im Übrigen ist hier schon zu erkennen, dass die geschätzten Kosten in Millisekunden der drei verschiedenen Kostenmodelle sehr voneinander abweichen und zum Vergleich der Strategien zwischen den drei unterschiedlichen Kostenmodellen nicht geeignet sind. Dies bedeutet allerdings nicht zwangsläufig, dass die Ausführungspläne mit höher prognostizierten Kosten der Kostenmodelle *improvedcosts* und *standardcosts* schlechter sind (Vgl. Abschnitt 9.1 auf Seite 59).

10.2 Fall 2

Im Gegensatz zum ersten Fall soll hier die Anfrage (10.3) betrachtet werden die mit dem Arbeitsspeicher nicht auskommt. Hier wird zwar in diesem Fall eine nichtlineare Optimierung durchgeführt, in der nur ein speicherverwendender Operator vorhanden ist, aber hier beschränkt sich die Optimierung darauf dem Operator den Maximalwert *min(SufficientMemory, GlobalMemory)* an Arbeitsspeicher zu gewähren. Die Ergebnisse zu dieser Anfrage sind in Tabelle 10.4 auf der nächsten Seite aufgelistet. Tabelle 10.5 auf der nächsten Seite zeigt die Arbeitsspeicherwerte der Operatoren und die entsprechenden Ausführungspläne, die durch die jeweilige Strategie erzeugt wurden, sind im Listing 10.4 aufgeführt.

Listing 10.3: Anfrage auf zwei Relationen, nicht ausreichender Arbeitsspeicher

```
1 select *
2 from   [buildings as b, roads as r]
3 where  [b:no=r:no]
```

Tabelle 10.4: Ergebnisse zur Anfrage 10.3

Strategie	Kosten in ms	Opt.-Zeit in ms	Ausführungs-zeit in s
staticEqual	224 052	23	1108,06
static 16 MiB	224 052	42	1094,78
static 256 MiB	200 736	32	1088,82
staticMaxDivOpCount(1) 256 MiB	200 736	60	1080,31
staticMaxDivPOG(1) 256 MiB	200 736	38	1084,17
modifiedDijkstra	200 736	67	1084,79
enumerate	200 736	74	1068,23
improvedcosts	19 590 950	8	993,67
standardcosts	14 725 099	4	1007,61

Tabelle 10.5: Arbeitsspeicherwerte zur Anfrage 10.3

Strategie	Arbeitsspeicher in MiB Operator 1	
	A	Z
staticEqual	431	256
static 16 MiB	431	256
static 256 MiB	398	256
staticMaxDivOpCount(1) 256 MiB	398	256
staticMaxDivPOG(1) 256 MiB	398	256
modifiedDijkstra	398	256
enumerate	398	256
improvedcosts		256
standardcosts		256

Listing 10.4: Ausführungspläne zur Anfrage 10.3

```
# staticEqual:
Roads feed{r} Buildings feed{b} hybridhashjoin[No_r, No_b, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# static 16 MiB:
Roads feed{r} Buildings feed{b} hybridhashjoin[No_r, No_b, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# static 256 MiB:
Buildings feed{b} Roads feed{r} hybridhashjoin[No_b, No_r, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# staticMaxDivOpCount(1) 256 MiB:
Buildings feed{b} Roads feed{r} hybridhashjoin[No_b, No_r, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# staticMaxDivPOG(1) 256 MiB:
Buildings feed{b} Roads feed{r} hybridhashjoin[No_b, No_r, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# modifiedDijkstra:
Buildings feed{b} Roads feed{r} hybridhashjoin[No_b, No_r, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# enumerate:
Buildings feed{b} Roads feed{r} hybridhashjoin[No_b, No_r, 99997]{memory 256}{5.563290081862227e-7, 1.9056e-5} consume
# improvedcosts:
Buildings feed{b} Roads feed{r} sortmergejoin[No_b, No_r]{5.563290081862227e-7, 1.9056e-5} consume
# standardcosts:
Buildings feed{b} Roads feed{r} sortmergejoin[No_b, No_r]{5.563290081862227e-7, 1.9056e-5} consume
```

Auswertung

Der auffälligste Unterschied ist, dass im Kostenmodell *ma_improvedcosts* nun nicht mehr der *gracehashjoin* Operator bevorzugt wird, sondern der *hybridhashjoin*. Die anderen beiden Kostenmodelle *improvedcosts* und *standardcosts* wählen den *sortmergejoin* Operator aus und liegen damit in der Ausführungszeit sogar ca. 5 bis 10% besser als die übrigen Ausführungspläne mit dem *hybridhashjoin* Operator. Die Wahl der Strategien *static 256 MiB, staticMaxDivOpCount(1) 256 MiB, staticMaxDivPOG(1) 256 MiB, modifiedDijkstra* und *enumerate*, die kleinere Relation als die rechte Relation zu verwenden[5], ist dabei minimal besser, nämlich im Bereich von ca. 1-3% als die Wahl der Strategien *staticEqual* und *static 16 MiB*. Dass nicht der *sortmergejoin* Operator für die Strategien der Speicheroptimierung ausgewählt wird, liegt hier daran, dass die Kosten für den Operator ca. 6 mal so hoch angesetzt werden wie für den *hybridhashjoin* Operator. Die Ausführungszeit belegt hier aber, dass diese Schätzung das reale Verhalten unter den Testbedingungen nicht widerspiegelt.

Was diesem Fall weiterhin zu entnehmen ist, ist der Unterschied der Kosten für die Strategien auf Basis des Kostenmodells der *ma_improvedcosts* untereinander. Die ersten beiden Strategien schätzen hier Kosten von 224 052 ms für eine Ausführungszeit von jeweils um die 1000 ms. Die

[5] Über die rechte Relation wird die Hashtabelle aufgebaut.

nächsten fünf Strategien schätzen die Kosten auf 200 736 für eine Ausführungszeit von um die 1080ms. Während auf Basis der prognostizierten Kosten also durch die Arbeitsspeicheroptimierung ein Ausführungsplan ermittelt werden konnte der ca. 10% geringere Kosten aufweist, schmilzt dieser prognostizierte Unterschied bei der Ausführungszeit der Pläne auf ca. 1 bis 3%. Aus Sicht der Arbeitsspeicheroptimierung war das Unterfangen der Optimierung also wesentlich erfolgreicher als sich dies letztendlich in den Ausführungszeiten widerspiegelt. In den folgenden Fällen wird dieser Unterschied sogar noch erheblich größer ausfallen.

10.3 Fall 3

Fall 3 ist die Abfrage 10.5 und verbindet drei Tabellen miteinander. Hier tritt sogar nur fallweise eine Speicheroptimierung über die Lösung eines nichtlinearen Optimierungsproblems auf, je nachdem welcher Pfad im Graphen der Kostenkanten auf Basis des POG gewählt wurde. Die Ergebnisse zu dieser Anfrage sind in Tabelle 10.6 aufgelistet. Tabelle 10.7 auf der nächsten Seite zeigt die Arbeitsspeicherwerte der Operatoren und die entsprechenden Ausführungspläne die durch die jeweilige Strategie erzeugt wurden, sind im Listing 10.6 aufgeführt.

Listing 10.5: Anfrage auf drei Relationen

```
1 select *
2 from   [buildings as b, roads as r, points as p]
3 where  [b:no=r:no, r:no=p:no]
```

Tabelle 10.6: Ergebnisse zur Anfrage 10.5

Strategie	Kosten in ms	Opt.-Zeit in ms	Ausführungs-zeit in s
staticEqual	134 355	68	343,18
static 16 MiB	134 355	105	326,05
static 256 MiB	69 112	78	284,71
staticMaxDivOpCount(2) 128 MiB	69 112	232	287,18
staticMaxDivPOG(2) 128 MiB	69 112	78	286,37
modifiedDijkstra	69 112	125	288,98
enumerate	69 112	326	289,38
improvedcosts	1 629 147	12	317,90
standardcosts	16 721 764	5	373,27

Tabelle 10.7: Arbeitsspeicherwerte zur Anfrage 10.5

Strategie	Arbeitsspeicher in MiB			
	Operator 1		Operator 2	
	A	Z	A	Z
staticEqual	398	128	16	128
static 16 MiB	398	128	16	128
static 256 MiB	67	153	16	102
staticMaxDivOpCount(2) 128 MiB	67	153	16	102
staticMaxDivPOG(2) 128 MiB	67	153	16	102
modifiedDijkstra	67	153	16	102
enumerate	67	153	16	102
improvedcosts		128		128
standardcosts		128		128

Listing 10.6: Ausführungspläne zur Anfrage 10.5

```
1  # staticEqual:
2  Buildings feed{b} Points feed{p} Roads feed{r} hybridhashjoin[No_p,
       No_r, 99997]{memory 128}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 128}{7.992e-9, 6.0696e-5} consume
3  # static 16 MiB:
4  Buildings feed{b} Points feed{p} Roads feed{r} hybridhashjoin[No_p,
       No_r, 99997]{memory 128}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 128}{7.992e-9, 6.0696e-5} consume
5  # static 256 MiB:
6  Buildings feed{b} Roads feed{r} Points feed{p} gracehashjoin[No_r,
       No_p, 99997]{memory 153}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 102}{7.992e-9, 6.0696e-5} consume
7  # staticMaxDivOpCount(2) 128 MiB:
8  Buildings feed{b} Roads feed{r} Points feed{p} gracehashjoin[No_r,
       No_p, 99997]{memory 153}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 102}{7.992e-9, 6.0696e-5} consume
9  # staticMaxDivPOG(2) 128 MiB:
10 Buildings feed{b} Roads feed{r} Points feed{p} gracehashjoin[No_r,
       No_p, 99997]{memory 153}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 102}{7.992e-9, 6.0696e-5} consume
11 # modifiedDijkstra:
12 Buildings feed{b} Roads feed{r} Points feed{p} gracehashjoin[No_r,
       No_p, 99997]{memory 153}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 102}{7.992e-9, 6.0696e-5} consume
13 # enumerate:
14 Buildings feed{b} Roads feed{r} Points feed{p} gracehashjoin[No_r,
       No_p, 99997]{memory 153}{7.992e-9, 4.6096e-5} gracehashjoin[No_b,
       No_r, 99997]{memory 102}{7.992e-9, 6.0696e-5} consume
```

```
15 # improvedcosts:
16 Roads  feed{r} Points  feed{p} hashjoin[No_r, No_p, ↩
      99997]{8.903425902819864e-7, 1.412e-5} Buildings  feed{b} hashjoin[↩
      No_r, No_b, 99997]{5.563290081862227e-7, 2.0112e-5} consume
17 # standardcosts:
18 Buildings  feed{b} Roads  feed{r} Points  feed{p} sortmergejoin[No_r, ↩
      No_p]{8.903425902819864e-7, 1.412e-5} sortmergejoin[No_b, No_r↩
      ]{5.563290081862227e-7, 2.0112e-5} consume
```

Auswertung

Die ersten beiden Strategien ermitteln hier einen Ausführungsplan der mehr Arbeitsspeicher als vorhanden benötigt. Auch die Ausführungszeit der Pläne sind im Bereich von ca. 15% schlechter als die der nächsten fünf Strategien. Die Strategie *improvedcosts* setzt für die Ausführung auf den *hashjoin*-Operator und liegt damit etwa ca. 10% schlechter in der Ausführungszeit als die Zeiten der fünf mittleren Strategien. Die Strategie *standardcosts* verwendet zur Ausführung den *sortmergejoin* Operator und liegt damit sogar ca. 29% schlechter als die fünf mittleren Strategien.

In diesem Fall ist auch die schon in Abschnitt 9.7 auf Seite 66 erwähnte Problematik der Konstanten der Kostenschätzung zu erkennen. In der zweiten Strategie *static 16 MiB* wurde von dem *hybridhashjoin* Operator angegeben, dass dieser gerne 398 MiB für die Ausführung hätte. Dem zweiten Operator *gracehashjoin* in dem Ausführungsplan reichen aber 16 MiB. Demnach hätte diesem nur 16 MiB zugewiesen werden sollen und der *hybridhashjoin* Operator hätte den Rest der 256 MiB zugewiesen bekommen. Dies ist allerdings nicht so, da für die dort verwendete Konstellation der Kardinalitäten und Tupelbreiten auf Basis der Konstanten eine ungültige Kostenfunktion übergeben wurde, diese wurde dann verworfen und durch die konstante Funktion des *TimeAt16MB* Wertes ersetzt (Vgl. Abschnitt 9.7 auf Seite 66). Damit bekamen beide Operatoren erst einmal nur jeweils 16 MiB in der Optimierung zugewiesen und durch Verteilung des übrigen Speichers kamen dann beide auf jeweils 128 MiB letztendlich zugewiesenen Arbeitsspeicher.

10.4 Fall 4

Der vierte Fall mit der Anfrage 10.7 beinhaltet eine Anfrage über vier Tabellen. Hier tritt auch wie im vorherigen Fall nur fallweise eine Speicheroptimierung über die Lösung eines nichtlinearen Optimierungsproblems auf. Die Ergebnisse zu dieser Anfrage sind in Tabelle 10.8 auf der nächsten Seite aufgelistet. Tabelle 10.9 auf der nächsten Seite zeigt die Arbeitsspeicherwerte der Operatoren und die entsprechenden Ausführungspläne die durch die jeweilige Strategie erzeugt wurden, sind im Listing 10.8 aufgeführt.

Listing 10.7: Anfrage auf vier Relationen

```
1 select *
2 from   [buildings as b, roads as r, points as p, natural as n]
3 where  [b:no=r:no, r:no=p:no, p:no=n:no]
```

Tabelle 10.8: Ergebnisse zur Anfrage 10.7

Strategie	Kosten in ms	Optimierungs-zeit in ms	Ausführungs-zeit in s
staticEqual	248 845	127	327,88
static 16 MiB	248 589	200	326,02
static 256 MiB	68 873	191	216,85
staticMaxDivOpCount(3) 85 MiB	68 873	1751	209,26
staticMaxDivPOG(3) 85 MiB	68 873	123	213,71
modifiedDijkstra	68 873	333	212,52
enumerate	68 873	17 671	205,48
improvedcosts	667 115	27	231,50
standardcosts	11 834 531	6	213,63

Tabelle 10.9: Arbeitsspeicherwerte zur Anfrage 10.7

Strategie	Arbeitsspeicher in MiB					
	Operator 1		Operator 2		Operator 3	
	A	Z	A	Z	A	Z
staticEqual	67	86	398	86	431	86
static 16 MiB	67	85	398	85	431	85
static 256 MiB	21	60	42	81	74	113
staticMaxDivOpCount(3) 85 MiB	21	60	42	81	74	113
staticMaxDivPOG(3) 85 MiB	21	60	42	81	74	113
modifiedDijkstra	21	60	42	81	74	113
enumerate	21	60	42	81	74	113
improvedcosts		85		85		85
standardcosts		85		85		85

Listing 10.8: Ausführungspläne zur Anfrage 10.7

```
1  # staticEqual:
2  Natural feed{n} Points feed{p} hybridhashjoin[No_n, No_p, 99997]{memory
       85}{3.3770650296296046e-6, 1.5464e-5} Roads feed{r} hybridhashjoin[
     No_p, No_r, 99997]{memory 85}{8.903425902819864e-7, 1.4976e-5}
     Buildings feed{b} hybridhashjoin[No_r, No_b, 99997]{memory
     85}{5.563290081862227e-7, 1.9584e-5} consume
3  # static 16 MiB:
4  Natural feed{n} Points feed{p} hybridhashjoin[No_n, No_p, 99997]{memory
       85}{3.3770650296296046e-6, 1.5464e-5} Roads feed{r} hybridhashjoin[
     No_p, No_r, 99997]{memory 85}{8.903425902819864e-7, 1.4976e-5}
     Buildings feed{b} hybridhashjoin[No_r, No_b, 99997]{memory
     85}{5.563290081862227e-7, 1.9584e-5} consume
5  # static 256 MiB:
6  Buildings feed{b} Roads feed{r} Points feed{p} Natural feed{n}
     gracehashjoin[No_p, No_n, 99997]{memory 61}{3.3770650296296046e-6,
     1.5464e-5} gracehashjoin[No_r, No_p, 99997]{memory
     82}{8.903425902819864e-7, 1.4976e-5} gracehashjoin[No_b, No_r,
     99997]{memory 114}{5.563290081862227e-7, 1.9584e-5} consume
7  # staticMaxDivOpCount(3) 85 MiB:
8  Buildings feed{b} Roads feed{r} Points feed{p} Natural feed{n}
     gracehashjoin[No_p, No_n, 99997]{memory 61}{3.3770650296296046e-6,
     1.5464e-5} gracehashjoin[No_r, No_p, 99997]{memory
     82}{8.903425902819864e-7, 1.4976e-5} gracehashjoin[No_b, No_r,
     99997]{memory 114}{5.563290081862227e-7, 1.9584e-5} consume
9  # staticMaxDivPOG(3) 85 MiB:
10 Buildings feed{b} Roads feed{r} Points feed{p} Natural feed{n}
     gracehashjoin[No_p, No_n, 99997]{memory 61}{3.3770650296296046e-6,
     1.5464e-5} gracehashjoin[No_r, No_p, 99997]{memory
     82}{8.903425902819864e-7, 1.4976e-5} gracehashjoin[No_b, No_r,
     99997]{memory 114}{5.563290081862227e-7, 1.9584e-5} consume
11 # modifiedDijkstra:
12 Buildings feed{b} Roads feed{r} Points feed{p} Natural feed{n}
     gracehashjoin[No_p, No_n, 99997]{memory 61}{3.3770650296296046e-6,
     1.5464e-5} gracehashjoin[No_r, No_p, 99997]{memory
     82}{8.903425902819864e-7, 1.4976e-5} gracehashjoin[No_b, No_r,
     99997]{memory 114}{5.563290081862227e-7, 1.9584e-5} consume
13 # enumerate:
14 Buildings feed{b} Roads feed{r} Points feed{p} Natural feed{n}
     gracehashjoin[No_p, No_n, 99997]{memory 61}{3.3770650296296046e-6,
     1.5464e-5} gracehashjoin[No_r, No_p, 99997]{memory
     82}{8.903425902819864e-7, 1.4976e-5} gracehashjoin[No_b, No_r,
     99997]{memory 114}{5.563290081862227e-7, 1.9584e-5} consume
15 # improvedcosts:
16 Natural feed{n} Points feed{p} hashjoin[No_n, No_p,
     99997]{3.3770650296296046e-6, 1.5464e-5} Roads feed{r} hashjoin[No_p
     , No_r, 99997]{8.903425902819864e-7, 1.4976e-5} Buildings feed{b}
     hashjoin[No_r, No_b, 99997]{5.563290081862227e-7, 1.9584e-5} consume
17 # standardcosts:
```

```
18 Buildings feed{b} Roads feed{r} Points feed{p} Natural feed{n} ↩
     hashjoin[No_p, No_n, 99997]{3.3770650296296046e-6, 1.5464e-5} ↩
     hashjoin[No_r, No_p, 99997]{8.903425902819864e-7, 1.4976e-5} hashjoin ↩
     [No_b, No_r, 99997]{5.563290081862227e-7, 1.9584e-5} consume
```

Auswertung

Dieses vierte Experiment unterscheidet sich im Großen und Ganzen nicht wesentlich vom dritten Fall. Erwähnenswert sind hier zwei Punkte: Erstens sind aufgrund der schon angesprochenen Problematik der Kostenkonstanten für den *hybridhashjoin* Operator die Ausführungszeiten der unteren sieben Strategien um bis zu 35% besser als die der ersten zwei Strategien. Der zweite Punkt ist, dass hier die Zeiten für die Erstellung der Ausführungspläne der Strategien *staticMaxDivOpCount* und *enumerate* im Gegensatz zum dritten Fall jetzt größere Unterschiede zu den übrigen Strategien aufweisen.

10.5 Fall 5

Der fünfte Fall mit der Anfrage 10.9 beinhaltet eine Abfrage über fünf Relationen. Hier ist der Arbeitsspeicher für keine der gewählten Strategien ausreichend. Die Ergebnisse zu dieser Anfrage sind in Tabelle 10.10 auf der nächsten Seite aufgelistet. Tabelle 10.9 auf Seite 83 zeigt die Arbeitsspeicherwerte der Operatoren und die entsprechenden Ausführungspläne, die durch die jeweilige Strategie erzeugt wurden und im Listing 10.10 aufgeführt werden.

Listing 10.9: Anfrage auf fünf Relationen

```
1 select *
2 from   [points as r1, buildings200k as r2, roads200k as r3,
3         roads200k as r4, roads300k as r5]
4 where  [r1:no=r2:no, r2:no=r3:no, r3:no=r4:no, r4:no=r5:no]
```

Tabelle 10.10: Ergebnisse zur Anfrage 10.9

Strategie	Kosten in ms	Optimierungszeit in ms	Ausführungszeit in s
staticEqual	116 884	322	354,81
static 16 MiB	114 340	367	337,44
static 256 MiB	779 220	360	321,00
staticMaxDivOpCount(4) 64 MiB	97 757	106 699	306,51
staticMaxDivPOG(4) 64 MiB	97 757	347	289,72
modifiedDijkstra	97 752	632	285,90
enumerate	97 567	540 453	289,53
improvedcosts	484 243	53	331,21
standardcosts	7 399 167	16	375,48

Tabelle 10.11: Arbeitsspeicherwerte zur Anfrage 10.9

Strategie	Arbeitsspeicher in MiB							
	Operator 1		Op. 2		Op. 3		Op. 4	
	A	Z	A	Z	A	Z	A	Z
staticEqual	48	64	70	64	67	64	106	64
static 16 MiB	48	49	70	70	67	67	106	67
static 256 MiB	74	48	122	59	168	68	239	80
staticMaxDivOpCount(4) 64 MiB	70	70	70	71	48	48	93	64
staticMaxDivPOG(4) 64 MiB	70	70	70	71	48	48	93	64
modifiedDijkstra	48	49	70	71	70	71	93	65
enumerate	48	49	70	71	70	71	93	65
improvedcosts		64		64		64		64
standardcosts		64		64		64		64

Listing 10.10: Ausführungspläne zur Anfrage 10.9

```
# staticEqual:
Roads200k feed{r3} Buildings200k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 64}{4.9950000000000005e-6, 1.8792e-5} Roads200k feed{r4} hybridhashjoin[No_r3, No_r4, 99997]{memory 64}{4.9950000000000005e-6, 1.8768e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 64}{3.3770650296296046e-6, 1.536e-5} Roads300k feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 64}{3.33e-6, 1.8448e-5} consume
# static 16 MiB:
Roads200k feed{r3} Buildings200k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 49}{4.9950000000000005e-6, 1.8792e-5} Roads200k feed{r4} hybridhashjoin[No_r3, No_r4, 99997]{memory 70}{4.9950000000000005e-6, 1.8768e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 67}{3.3770650296296046e-6, 1.536e-5} Roads300k feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 67}{3.33e-6, 1.8448e-5} consume
# static 256 MiB:
Roads300k feed{r5} Roads200k feed{r4} Points feed{r1} Buildings200k feed{r2} Roads200k feed{r3} itHashJoin[No_r2, No_r3]{memory 48}{4.9950000000000005e-6, 1.8792e-5} itHashJoin[No_r1, No_r2]{memory 59}{3.3770650296296046e-6, 1.536e-5} itHashJoin[No_r4, No_r3]{memory 68}{4.9950000000000005e-6, 1.8768e-5} itHashJoin[No_r5, No_r4]{memory 80}{3.33e-6, 1.8448e-5} consume
# staticMaxDivOpCount(4) 64 MiB:
Roads300k feed{r5} Roads200k feed{r4} hybridhashjoin[No_r5, No_r4, 99997]{memory 70}{3.33e-6, 1.8448e-5} Roads200k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 71}{4.9950000000000005e-6, 1.8768e-5} Points feed{r1} Buildings200k feed{r2} gracehashjoin[No_r1, No_r2, 99997]{memory 48}{3.3770650296296046e-6, 1.536e-5} hybridhashjoin[No_r3, No_r2, 99997]{memory 64}{4.9950000000000005e-6, 1.8792e-5} consume
# staticMaxDivPOG(4) 64 MiB:
Roads300k feed{r5} Roads200k feed{r4} hybridhashjoin[No_r5, No_r4, 99997]{memory 70}{3.33e-6, 1.8448e-5} Roads200k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 71}{4.9950000000000005e-6, 1.8768e-5} Points feed{r1} Buildings200k feed{r2} gracehashjoin[No_r1, No_r2, 99997]{memory 48}{3.3770650296296046e-6, 1.536e-5} hybridhashjoin[No_r3, No_r2, 99997]{memory 64}{4.9950000000000005e-6, 1.8792e-5} consume
# modifiedDijkstra:
Roads300k feed{r5} Roads200k feed{r4} gracehashjoin[No_r5, No_r4, 99997]{memory 71}{3.33e-6, 1.8448e-5} Roads200k feed{r3} gracehashjoin[No_r4, No_r3, 99997]{memory 71}{4.9950000000000005e-6, 1.8768e-5} Points feed{r1} Buildings200k feed{r2} gracehashjoin[No_r1, No_r2, 99997]{memory 49}{3.3770650296296046e-6, 1.536e-5} gracehashjoin[No_r3, No_r2, 99997]{memory 65}{4.9950000000000005e-6, 1.8792e-5} consume
# enumerate:
```

```
14 Roads300k feed{r5} Roads200k feed{r4} hybridhashjoin[No_r5, No_r4, 99997]{memory 71}{3.33e-6, 1.8448e-5} Roads200k feed{r3} gracehashjoin[No_r4, No_r3, 99997]{memory 71}{4.9950000000000005e-6, 1.8768e-5} Points feed{r1} Buildings200k feed{r2} gracehashjoin[No_r1, No_r2, 99997]{memory 49}{3.3770650296296046e-6, 1.536e-5} hybridhashjoin[No_r3, No_r2, 99997]{memory 65}{4.9950000000000005e-6, 1.8792e-5} consume
15 # improvedcosts:
16 Roads200k feed{r4} Roads300k feed{r5} hashjoin[No_r4, No_r5, 99997]{3.33e-6, 1.8448e-5} Roads200k feed{r3} hashjoin[No_r4, No_r3, 99997]{4.9950000000000005e-6, 1.8768e-5} Buildings200k feed{r2} hashjoin[No_r3, No_r2, 99997]{4.9950000000000005e-6, 1.8792e-5} Points feed{r1} hashjoin[No_r2, No_r1, 99997]{3.3770650296296046e-6, 1.536e-5} consume
17 # standardcosts:
18 Points feed{r1} Buildings200k feed{r2} hashjoin[No_r1, No_r2, 99997]{3.3770650296296046e-6, 1.536e-5} Roads200k feed{r3} hashjoin[No_r2, No_r3, 99997]{4.9950000000000005e-6, 1.8792e-5} Roads200k feed{r4} hashjoin[No_r3, No_r4, 99997]{4.9950000000000005e-6, 1.8768e-5} Roads300k feed{r5} hashjoin[No_r4, No_r5, 99997]{3.33e-6, 1.8448e-5} consume
```

Auswertung

Hier ist nun erstmals der Fall zu sehen, dass alle Strategien unterschiedliche Ausführungspläne erzeugen. Die Strategien *staticMaxDivOpCount(4)* und *staticMaxDivPOG(4)* besitzen zwar die gleichen Kosten, aufgrund des gleichen dort ermittelten Divisors sind diese allerdings hier gleich zu bewerten. Auch tritt deutlicher hervor, was sich im vierten Fall angedeutet hatte: Die Strategien *staticMaxDivOpCount* und *enumerate* steigen exponentiell in ihren Kosten.

Zu sehen ist hier auch, dass der Laufzeitunterschied sich nun klarer darstellt zu der Referenzstrategie *staticEqual*, aber auch den Strategien *improvedcosts* und *standardcosts*. Die ersten drei und letzten beiden Strategien liegen in der Laufzeit grob zwischen 13 bis 30% schlechter als die übrigen Strategien. Weiterhin ist hier das erste Mal der Fall eingetreten, dass die Strategie *static 256 MiB* eine eher schlechte Wahl trifft.

Als letzte erwähnenswerte Tatsache dieses Falls ist festzustellen, dass hier die Heuristik *modified-Dijkstra* zum ersten Mal auf Basis der Kosten nicht mehr den besten Ausführungsplan ermittelt. Hier wirkt sich also aus, dass die Heuristik nicht garantieren kann, den besten Ausführungsplan ermitteln zu können (Vgl. Abschnitt 8.7 auf Seite 57). Allerdings ist der erzeugte Ausführungsplan immer noch ca. 19% schneller ausgeführt als der Ausführungsplan der verwendeten Referenzstrategie *staticEqual*.

10.6 Fall 6

Dieser Fall mit der Anfrage 10.11 ist ähnlich dem Fall 5, verarbeitet aber nochmals größere Datenmengen. Die Ergebnisse zu dieser Anfrage sind in Tabelle 10.12 auf der nächsten Seite aufgelistet. Tabelle 10.13 auf Seite 90 zeigt die Arbeitsspeicherwerte der Operatoren und die entsprechenden

Ausführungspläne, die durch die jeweilige Strategie erzeugt wurden, sind im Listing 10.12 aufgeführt.

Listing 10.11: Anfrage auf fünf Relationen

```
1 select *
2 from   [points as r1, buildings300k as r2, roads300k as r3,
3        roads300k as r4, buildings200k as r5]
4 where  [r1:no=r2:no, r2:no=r3:no, r3:no=r4:no, r4:no=r5:no]
```

Tabelle 10.12: Ergebnisse zur Anfrage 10.11

Strategie	Kosten in ms	Opt.-Zeit in ms	Ausführungszeit in s
staticEqual	141 856	322	361,55
static 16 MiB	141 440	334	376,49
static 256 MiB	636 327	335	359,25
staticMaxDivOpCount(4) 64 MiB	125 923	93 649	343,63
staticMaxDivPOG(4) 64 MiB	125 923	343	348,52
modifiedDijkstra	122 319	624	344,40
enumerate	121 541	575 011	355,47
improvedcosts	526 763	52	347,81
standardcosts	8 397 373	32	120,75

Tabelle 10.13: Arbeitsspeicherwerte zur Anfrage 10.11

Strategie	Arbeitsspeicher in MiB							
	Operator 1		Op. 2		Op. 3		Op. 4	
	A	Z	A	Z	A	Z	A	Z
staticEqual	106	64	106	64	72	64	67	64
static 16 MiB	106	57	106	57	72	72	67	67
static 256 MiB	48	17	122	66	193	82	241	91
staticMaxDivOpCount(4) 64 MiB	48	49	106	65	72	72	67	67
staticMaxDivPOG(4) 64 MiB	48	49	106	65	72	72	67	67
modifiedDijkstra	48	49	106	65	72	72	67	67
enumerate	48	50	122	123	72	17	67	66
improvedcosts		64		64		64		64
standardcosts		64		64		64		64

Listing 10.12: Ausführungspläne zur Anfrage 10.11

```
1 # staticEqual:
2 Buildings200k feed{r5} Roads300k feed{r4} hybridhashjoin[No_r5, No_r4, 99997]{memory 64}{3.33e-6, 1.6808e-5} Roads300k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 64}{3.33e-6, 3.012e-5} Buildings300k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 64}{3.33e-6, 1.9664e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 64}{3.33e-6, 1.6408e-5} consume
3 # static 16 MiB:
4 Buildings200k feed{r5} Roads300k feed{r4} hybridhashjoin[No_r5, No_r4, 99997]{memory 57}{3.33e-6, 1.6808e-5} Roads300k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 57}{3.33e-6, 3.012e-5} Buildings300k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 72}{3.33e-6, 1.9664e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 67}{3.33e-6, 1.6408e-5} consume
5 # static 256 MiB:
6 Points feed{r1} Buildings300k feed{r2} Roads300k feed{r3} Roads300k feed{r4} Buildings200k feed{r5} gracehashjoin[No_r4, No_r5, 99997]{memory 17}{3.33e-6, 1.6808e-5} itHashJoin[No_r3, No_r4]{memory 66}{3.33e-6, 3.012e-5} itHashJoin[No_r2, No_r3]{memory 82}{3.33e-6, 1.9664e-5} itHashJoin[No_r1, No_r2]{memory 91}{3.33e-6, 1.6408e-5} consume
7 # staticMaxDivOpCount(4) 64 MiB:
8 Roads300k feed{r4} Buildings200k feed{r5} gracehashjoin[No_r4, No_r5, 99997]{memory 49}{3.33e-6, 1.6808e-5} Roads300k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 65}{3.33e-6, 3.012e-5} Buildings300k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 72}{3.33e-6, 1.9664e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 67}{3.33e-6, 1.6408e-5} consume
```

```
# staticMaxDivPOG(4) 64 MiB:
Roads300k feed{r4} Buildings200k feed{r5} gracehashjoin[No_r4, No_r5, 99997]{memory 49}{3.33e-6, 1.6808e-5} Roads300k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 65}{3.33e-6, 3.012e-5} Buildings300k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 72}{3.33e-6, 1.9664e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 67}{3.33e-6, 1.6408e-5} consume
# modifiedDijkstra:
Roads300k feed{r3} Roads300k feed{r4} Buildings200k feed{r5} gracehashjoin[No_r4, No_r5, 99997]{memory 49}{3.33e-6, 1.6808e-5} itHashJoin[No_r3, No_r4]{memory 123}{3.33e-6, 3.012e-5} Buildings300k feed{r2} gracehashjoin[No_r3, No_r2, 99997]{memory 17}{3.33e-6, 1.9664e-5} Points feed{r1} gracehashjoin[No_r2, No_r1, 99997]{memory 67}{3.33e-6, 1.6408e-5} consume
# enumerate:
Roads300k feed{r3} Roads300k feed{r4} Buildings200k feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 50}{3.33e-6, 1.6808e-5} itHashJoin[No_r3, No_r4]{memory 123}{3.33e-6, 3.012e-5} Buildings300k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 17}{3.33e-6, 1.9664e-5} Points feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 66}{3.33e-6, 1.6408e-5} consume
# improvedcosts:
Roads300k feed{r4} Buildings200k feed{r5} hashjoin[No_r4, No_r5, 99997]{3.33e-6, 1.6808e-5} Roads300k feed{r3} hashjoin[No_r4, No_r3, 99997]{3.33e-6, 3.012e-5} Buildings300k feed{r2} hashjoin[No_r3, No_r2, 99997]{3.33e-6, 1.9664e-5} Points feed{r1} hashjoin[No_r2, No_r1, 99997]{3.33e-6, 1.6408e-5} consume
# standardcosts:
Points feed{r1} Buildings300k feed{r2} Roads300k feed{r3} Roads300k feed{r4} Buildings200k feed{r5} hashjoin[No_r4, No_r5, 99997]{3.33e-6, 1.6808e-5} hashjoin[No_r3, No_r4, 99997]{3.33e-6, 3.012e-5} hashjoin[No_r2, No_r3, 99997]{3.33e-6, 1.9664e-5} hashjoin[No_r1, No_r2, 99997]{3.33e-6, 1.6408e-5} consume
```

Auswertung

Dieser Fall birgt aus Sicht der Speicheroptimierung nichts wesentlich Neues. Die implementierten Strategien zur Arbeitsspeicheroptimierung sind hier nur leicht besser als die als Referenz herangezogene Strategie *staticEqual*. Interessant ist hier der Fall, dass die Strategie *standardcosts* einen wesentlich besseren Ausführungsplan erzeugt als die restlichen Strategien. Dies ist allerdings weniger auf die Auswahl der Join Operatoren zurückzuführen als auf die Reihenfolge der Ausführung und der Relationen.

10.7 Fall 7

Der letzte Fall soll nicht mehr in jedem Detail analysiert werden und in erster Linie einen weiteren Fall aufzeigen, in dem noch einmal die verarbeiteten Datenmengen wesentlich erhöht wurden. Die Experimente wurden hier auch nur noch zwei Mal durchgeführt, wovon nur die erste Zeit dann in

das Resultat übernommen wurde. Die Anfrage 10.13 ist wieder als erstes abgebildet, gefolgt von den Ergebnissen in Tabelle 10.14. Tabelle 10.15 auf der nächsten Seite zeigt die Arbeitsspeicherwerte der Operatoren und und am Ende stehen auch hier wieder die Ausführungspläne im Listing 10.14.

Listing 10.13: Anfrage auf sechs Relationen

```
1 select *
2 from   [roads300k as r1, roads400k as r2, roads500k as r3,
3         buildings300k as r4, buildings400k as r5, buildings500k as r6]
4 where  [r1:no=r2:no, r2:no=r3:no, r3:no=r4:no, r4:no=r5:no, r5:no=r6:no]
```

Tabelle 10.14: Ergebnisse zur Anfrage 10.13

Strategie	Kosten in ms	Optimierungs-zeit in ms	Ausführungs-zeit in s
staticEqual	261 110	892	1001,97
static 16 MiB	249 996	920	996,26
static 256 MiB	549 283	925	942,57
staticMaxDivOpCount(5) 51 MiB	226 775	798 052	982,24
staticMaxDivPOG(5) 51 MiB	226 775	965	987,91
modifiedDijkstra	222 429	2318	963,55
enumerate	222 429	1 234 656	942,84
improvedcosts	1 160 307	81	983,51
standardcosts	14 391 631	39	967,50

Tabelle 10.15: Arbeitsspeicherwerte zur Anfrage 10.13

Strategie	Arbeitsspeicher in MiB									
	Operator 1		Op. 2		Op. 3		Op. 4		Op. 5	
	A	Z	A	Z	A	Z	A	Z	A	Z
staticEqual	177	52	141	52	106	52	96	52	120	52
static 16 MiB	177	17	141	98	106	107	96	17	120	17
static 256 MiB	72	17	182	97	231	108	234	17	278	17
staticMaxDivOpCount(5) 51 MiB	72	73	141	59	106	89	96	18	120	17
staticMaxDivPOG(5) 51 MiB	72	73	141	59	106	89	96	18	120	17
modifiedDijkstra	106	107	72	73	178	42	96	17	120	17
enumerate	106	107	72	73	178	42	96	17	120	17
improvedcosts		52		52		52		52		52
standardcosts		52		52		52		52		52

Listing 10.14: Ausführungspläne zur Anfrage 10.13

```
1 # staticEqual:
2 Buildings300k feed{r4} Roads500k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 52}{1.9979999999999998e-6, 2.26e-5} Roads400k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 52}{1.9979999999999998e-6, 2.1248e-5} Roads300k feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 52}{2.4975000000000002e-6, 2.124e-5} Buildings400k feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 52}{2.4975000000000002e-6, 1.7536e-5} Buildings500k feed{r6} hybridhashjoin[No_r5, No_r6, 99997]{memory 52}{1.9979999999999998e-6, 1.868e-5} consume
3 # static 16 MiB:
4 Buildings300k feed{r4} Roads500k feed{r3} hybridhashjoin[No_r4, No_r3, 99997]{memory 17}{1.9979999999999998e-6, 2.26e-5} Roads400k feed{r2} hybridhashjoin[No_r3, No_r2, 99997]{memory 98}{1.9979999999999998e-6, 2.1248e-5} Roads300k feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 107}{2.4975000000000002e-6, 2.124e-5} Buildings400k feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 17}{2.4975000000000002e-6, 1.7536e-5} Buildings500k feed{r6} hybridhashjoin[No_r5, No_r6, 99997]{memory 17}{1.9979999999999998e-6, 1.868e-5} consume
5 # static 256 MiB:
6 Buildings500k feed{r6} Buildings400k feed{r5} Roads300k feed{r1} Roads400k feed{r2} Roads500k feed{r3} Buildings300k feed{r4} gracehashjoin[No_r3, No_r4, 99997]{memory 17}{1.9979999999999998e-6, 2.26e-5} itHashJoin[No_r2, No_r3]{memory 97}{1.9979999999999998e-6, 2.1248e-5} itHashJoin[No_r1, No_r2]{memory 108}{2.4975000000000002e-6, 2.124e-5} gracehashjoin[No_r5, No_r4, 99997]{memory 17}{2.4975000000000002e-6, 1.7536e-5} hybridhashjoin[No_r6, No_r5,
```

```
    99997]{memory 17}{1.9979999999999998e-6, 1.868e-5} consume
# staticMaxDivOpCount(5) 51 MiB:
Roads500k  feed{r3} Buildings300k  feed{r4} hybridhashjoin[No_r3, No_r4, 99997]{memory 73}{1.9979999999999998e-6, 2.26e-5} Roads400k  feed{r2}  hybridhashjoin[No_r3, No_r2, 99997]{memory 59}{1.9979999999999998e-6, 2.1248e-5} Roads300k  feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 89}{2.4975000000000002e-6, 2.124e-5} Buildings400k  feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 18}{2.4975000000000002e-6, 1.7536e-5} Buildings500k  feed{r6} hybridhashjoin[No_r5, No_r6, 99997]{memory 17}{1.9979999999999998e-6, 1.868e-5} consume
# staticMaxDivPOG(5) 51 MiB:
Roads500k  feed{r3} Buildings300k  feed{r4} hybridhashjoin[No_r3, No_r4, 99997]{memory 73}{1.9979999999999998e-6, 2.26e-5} Roads400k  feed{r2}  hybridhashjoin[No_r3, No_r2, 99997]{memory 59}{1.9979999999999998e-6, 2.1248e-5} Roads300k  feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 89}{2.4975000000000002e-6, 2.124e-5} Buildings400k  feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 18}{2.4975000000000002e-6, 1.7536e-5} Buildings500k  feed{r6} hybridhashjoin[No_r5, No_r6, 99997]{memory 17}{1.9979999999999998e-6, 1.868e-5} consume
# modifiedDijkstra:
Roads400k  feed{r2} Roads300k  feed{r1} hybridhashjoin[No_r2, No_r1, 99997]{memory 107}{2.4975000000000002e-6, 2.124e-5} Roads500k  feed{r3} Buildings300k  feed{r4} gracehashjoin[No_r3, No_r4, 99997]{memory 73}{1.9979999999999998e-6, 2.26e-5} hybridhashjoin[No_r2, No_r3, 99997]{memory 42}{1.9979999999999998e-6, 2.1248e-5} Buildings400k  feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 17}{2.4975000000000002e-6, 1.7536e-5} Buildings500k  feed{r6} hybridhashjoin[No_r5, No_r6, 99997]{memory 17}{1.9979999999999998e-6, 1.868e-5} consume
# enumerate:
Roads400k  feed{r2} Roads300k  feed{r1} gracehashjoin[No_r2, No_r1, 99997]{memory 107}{2.4975000000000002e-6, 2.124e-5} Roads500k  feed{r3} Buildings300k  feed{r4} hybridhashjoin[No_r3, No_r4, 99997]{memory 73}{1.9979999999999998e-6, 2.26e-5} hybridhashjoin[No_r2, No_r3, 99997]{memory 42}{1.9979999999999998e-6, 2.1248e-5} Buildings400k  feed{r5} hybridhashjoin[No_r4, No_r5, 99997]{memory 17}{2.4975000000000002e-6, 1.7536e-5} Buildings500k  feed{r6} hybridhashjoin[No_r5, No_r6, 99997]{memory 17}{1.9979999999999998e-6, 1.868e-5} consume
# improvedcosts:
Roads500k  feed{r3} Buildings300k  feed{r4} hashjoin[No_r3, No_r4, 99997]{1.9979999999999998e-6, 2.26e-5} Roads400k  feed{r2} hashjoin[No_r3, No_r2, 99997]{1.9979999999999998e-6, 2.1248e-5} Roads300k  feed{r1} hashjoin[No_r2, No_r1, 99997]{2.4975000000000002e-6, 2.124e-5} Buildings400k  feed{r5} hashjoin[No_r4, No_r5, 99997]{2.4975000000000002e-6, 1.7536e-5} Buildings500k  feed{r6} hashjoin[No_r5, No_r6, 99997]{1.9979999999999998e-6, 1.868e-5}
```

```
    consume
17 # standardcosts:
18 Roads300k  feed{r1} Roads400k  feed{r2} Roads500k  feed{r3} Buildings300k
     feed{r4} sortmergejoin[No_r3, No_r4]{1.9979999999999998e-6, 2.26e
   -5} sortmergejoin[No_r2, No_r3]{1.9979999999999998e-6, 2.1248e-5}
   hashjoin[No_r1, No_r2, 99997]{2.4975000000000002e-6, 2.124e-5}
   Buildings400k  feed{r5} hashjoin[No_r4, No_r5,
   99997]{2.4975000000000002e-6, 1.7536e-5} Buildings500k  feed{r6}
   hashjoin[No_r5, No_r6, 99997]{1.9979999999999998e-6, 1.868e-5}
   consume
```

Auswertung

Auch hier ist weiter die exponentielle Steigerung der Optimierungkosten zu beobachten. Die besten Ausführungspläne finden hier die beiden Strategien *modifiedDijkstra* und *enumerate*. Die Kostenverbesserungen dieser beiden Strategien zur Strategie *staticEqual* liegt im Rahmen von ca. 3,8% bis 5,9%.

10.8 Auswertung der Fälle

Die fallspezifischen Besonderheiten wurden bereits in den einzelnen Fällen diskutiert. Deutlich gezeigt hat sich besonders in den letzten Fällen, dass die Kosten für die Arbeitsspeicheroptimierung der Strategien *staticMaxDivOpCount* und *enumerate* mit zunehmender Größe des POG's zu stark steigen, um für den regulären Einsatz in Frage zu kommen. Auf Basis der Kosten allerdings fand nur die *enumerate* Strategie den Ausführungsplan mit den günstigen Gesamtkosten. Die anderen Strategien auf Basis der *ma_improvedcosts* lagen teilweise auf Basis der Kostenschätzung wesentlich höher (Fall 5), in den Ausführungszeiten sind die Unterschiede nicht mehr so groß, weichen aber trotzdem noch bis ca. 30% von den Ausführungsplänen der *enumerate* Strategie ab.

Die Strategie *staticMaxDivPOG* zeigte eigentlich gute Ergebnisse, ohne die hohen Kosten für die Optimierung wie durch die *staticMaxDivOpCount* Strategie aufzuweisen. Um die Probleme der Strategie zu zeigen wurden hier keine weiteren Testfälle definiert. Schwächen sind hier zu erwarten, wenn weitere Bedingungen in die WHERE-Klausel aufgenommen werden, die keine Join-Operatoren in den Ausführungsplänen erzeugen oder auch Join-Operationen in den Pfaden erscheinen, die keine speicherabhängingen Kostenfunktionen verwenden. Beispielsweise wäre dies für die regulären Operatoren *hashjoin* oder *spatialjoin* der Fall.

Die einfachen Strategien *static 16 MiB* und *static 256 MiB* stellen die beiden Extremfälle dar, indem zuerst jeder speicherverwendende Operator nur die minimale oder maximale Menge an Arbeitsspeicher erhält, um den Ausführungsplan zu ermitteln. Die anschließende Optimierung der so ermittelten Ausführungspläne hat dann nicht in jedem Fall mehr die Möglichkeit wesentlich besser abzuscheiden als die *staticEqual* Strategie Im Fall 5 und 6 versagte die Strategie *static 256 MiB* sogar. Im Fall 5 waren die hier ermittelten Kosten fast 7 mal schlechter als die Kosten der *staticEqual* Strategie. In der Ausführungszeit zeigt sich der Unterschied dabei dann aber nur im Bereich von wenigen Prozenten. Die Optimierung basiert dabei aber immer nur auf den geschätzten Kosten. Damit werden zwei Probleme dieses Falls deutlich: Erstens hat die Optimierung hier eine schlechte Wahl mit dieser Strategie getroffen und zweitens zeigte die Kostenschätzung viel zu hohe Kosten für

einen Ausführungsplan an, der sich, gemessen an den Kosten und Ausführungszeiten der anderen Strategien der *ma_improvedcosts*, als nicht so schlecht darstellte.

Die Heuristik *modifiedDijkstra* konnte zwar nicht immer den Ausführungsplan ermitteln, den auch die Strategie *enumerate* ermittelt hat (Fall 6), allerdings war die Strategie auf Basis der Kosten nah an den optimalen Kosten. In den Ausführungszeiten waren sogar keine Abweichungen feststellbar, die nicht im Rahmen der Messtoleranz lagen. Legitim wäre es hier zu vermuten, dass dies für die hier durchgeführten Experimente zumindest der Fall war, sich im Allgemeinen aber nicht bestätigt. Gezeigt wurde bereits in Abschnitt 8.7 auf Seite 57, dass die Heuristik nicht immer den besten Fall auf Basis der Kosten finden kann. Auch war dies im Fall 5 zu sehen. Was allerdings für die Heuristik spricht ist die durchaus bemerkenswerte Tatsache, dass sie unter den 157 464 möglichen Pfaden im Graphen der Kostenkanten auf Basis des POG diesen guten Ausführungsplan im Fall 5 ermittelt hat. Noch überraschender ist aber Fall 7, hier konnte die *modifiedDijkstra* Strategie denselben Plan ermitteln, den auch die Strategie *enumerate* ermittelt hat. Dies hier allerdings unter sogar 7 085 880 möglichen Pfaden im Graphen der Kostenkanten. Die Zeit für die Erstellung des Ausführungsplans ist allerdings teurer als für die anderen Strategien, die nicht auf das Aufzählen aller möglichen Pfade angewiesen sind. Dies liegt daran, dass hier mehrere nichtlineare Optimierungen durchgeführt werden müssen. Dazu erwähnt sei hier, dass die Optimierungszeit der Strategien *improvedcosts* und *standardcosts* nicht mit den anderen Strategien verglichen werden kann. Die Anzahl möglicher Pfade im Graphen der Kostenkanten auf Basis des POG ist dort wesentlich geringer. Im Fall 7 existieren hier beispielsweise 29 160 mögliche Pfade im Graph der Kostenkanten.

Ein Nachteil der Strategie *modifiedDijkstra* ist die Tatsache, dass der Algorithmus nicht die blockierenden Operatoren zur weiteren Verbesserung berücksichtigen kann (Vgl. Abschnitt 9.6 auf Seite 65). Allerdings konnten hierzu keine Experimente definiert werden, da der *sortmergejoin* Operator aufgrund der hohen Kosten[6] nicht berücksichtigt wurde. Hier könnte ggf. in einer späteren Evaluierung hervorgehen, dass die Unterschiede noch größer werden. Da bisher die Ausführungspläne der *modifiedDijkstra* Heuristik in den gezeigten Fällen immer nah an denen der *enumerate* Strategie waren und die Ermittlung der Ausführungspläne in akzeptabler Zeit durchgeführt werden konnte, ist diese Strategie damit die geeigneteste der evaluierten Strategien zur Arbeitsspeicheroptimierung der Operatoren. Aus diesem Grund wurde diese Strategie als Standardstrategie für die Erweiterung der Arbeitsspeicheroptimierung eingestellt.

In Tabelle 10.16 auf der nächsten Seite werden die Ausführungszeiten aller Experimente 1 bis 7 addiert und in der letzten Spalte wird immer die Abweichung in Prozent zur *staticEqual* Strategie angegeben. Natürlich ist die Auswahl der sieben Experimente beschränkt in der Repräsentativität zum allgemeinen Verhalten, dafür sind es zu wenige Fälle. Für das Testszenario soll aber trotzdem versucht werden die Gesamtleistung anzugeben. Im Vergleich zu der *staticEqual* Strategie sind die positiven Resultate der Speicheroptimierung zu erkennen. Zu erkennen ist allerdings auch, dass die *standardcosts* Strategie noch bessere Ergebnisse erzielt, dies sogar ohne Speicheroptimierung. Dieser Tatbestand liegt in dem verwendeten Kostenmodell begründet. Durch eine bessere Kostenschätzung der *ma_improvedcosts* sind die Werte aber auch zu erreichen und dann hätte eine Arbeitsspeicheroptimierung durchaus noch die Möglichkeit diese Ausführungszeiten zu unterbieten.

Eine Vermutung des Autors konnte allerdings im Rahmen der Evaluierung aufgrund von Zeit und Platzgründen nicht mehr gezeigt werden. Es geht hierbei darum, dass, wenn bei Arbeitsspeicheranforderungen aller Operatoren eines Ausführungsplans einer komplexeren SQL-Anfrage diese

[6] Gemeint sind die Kosten, die durch die Kostenfunktion angezeigt werden. Die realen Kosten sind oft nicht so hoch wie Beispielsweise Fall 2 zeigte.

Tabelle 10.16: Addierte Ergebnisse

Strategie	Summe der Ausführungszeiten in s	Abweichung in Prozent zur *staticEqual* Strategie
staticEqual	3489,92	0,00
static 16 MiB	3467,44	−0,64
static 256 MiB	3228,50	−7,49
staticMaxDivOpCount(5) 51 MiB	3223,82	−7,62
staticMaxDivPOG(5) 51 MiB	3230,60	−7,43
modifiedDijkstra	3196,63	−8,40
enumerate	3172,15	−9,11
improvedcosts	3222,21	−7,67
standardcosts	3071,01	−12,00

den verfügbaren Arbeitsspeicher um ein Vielfaches übersteigen, der Vorteil der Arbeitsspeicheroptimierung ins Gegenteil umschlägt und schlechtere Ausführungspläne erstellt werden als durch die Strategie *staticEqual*. Basis der Vermutung ist, dass der Fehler der verschiedenen Eingabeparameter in die Arbeitsspeicheroptimierung ab einem gewissen Punkt so groß wird, dass der Arbeitsspeicheroptimierung aufgrund dieser Parameter die Möglichkeit genommen wird, eine realistische Optimierung durchzuführen. In diesem Fall würde die *staticEqual* Strategie nicht den Fehler begehen, einem oder mehreren Operatoren viel zu wenig Arbeitsspeicher zuzuweisen. Allerdings ist dies weniger ein Problem der Arbeitsspeicheroptimierung. Bei richtigen Eingabeparametern ist aufgrund der gezeigten Fälle nicht davon auszugehen, dass ein ungünstiger Ausführungsplan erzeugt wird. Im Fall der *enumerate* Strategie ist dies sogar sichergestellt.

Kapitel 11

Zukünftige Erweiterungen

Eine interessante Erweiterung wäre die Integration des *sort* Operators in die Speicheroptimierung, da dieser häufig[1] verwendet wird und Kostenintensiv ist. Im Falle neuer *CostEstimation*-Implementierungen von vorhandenen oder neuen Operatoren sind diese aber leicht in die Speicheroptimierung zu integrieren. Hierzu muss erstens die Operatorsignatur in die Datei *madata.pl* mit aufgenommen werden und zum anderen muss für den Operator ein entsprechendes *cost* Prädikat für den Operator in der Datei *ma_improvedcosts.pl* erstellt werden das für die Ermittlung der Kosten über die *CostEstimation* Schnittstelle sorgt.

Eine weitere mögliche Erweiterung wäre dann die direkte Implementierung eines Algorithmus zum Lösen eines nichtlinearen Optimierungproblems in Prolog. Dies hätte den Vorteil weniger abhängig von einer Bibliothek zu sein. Aufgrund der Schwierigkeit als Teil dieser Arbeit auch noch einen numerisch stabilen Algorithmus zu implementieren, wurde hier darauf verzichtet und die Entscheidung fiel zu Gunsten der NLopt Bibliothek.

Darüber hinaus wäre eine andere Erweiterung in Bezug auf die Verarbeitung von parallelen Anfragen denkbar. Hier wäre der zur Verfügung stehende Arbeitsspeicher auf mehere Anfragen nach einem Algorithmus aufzuteilen. Bisher wurde nur die Optimierung einer einzelnen Anfrage betrachtet, die den ganzen Arbeitsspeicher verwenden durfte. Komplexer ist hier die Frage, wie gut dies mit dem derzeitigen Verfahren gelöst werden kann, da die Zuweisung von Arbeitsspeicher an die Operatoren, nachdem der Ausführungsplan einmal zur Ausführung abgesetzt wurde, fest ist. Eine Änderung ist bei neu eingehenden Anfragen nicht mehr möglich. Den Operatoren kann dann nicht wieder so einfach Arbeitsspeicher entzogen werden. Allerdings wäre es denkbar, dass dem Operator noch nachträglich bisher nicht verwendeter Arbeitsspeicher entzogen wird, weil dieser besser in einem ganz anderen Ausführungsplan benötigt wird und damit die Anfrage zwar teurer wird, aber beide Anfragen in der Gesamtbetrachtung so die geringsten Kosten aufweisen. Der Vorteil eines solchen dynamischeren Systems wäre auch noch ein anderer im Zusammenhang mit der Problematik der Kostenschätzung, dass dort aus dem Stand heraus die Kosten zu schätzen sind (Vgl. auch [Nid12]). So könnten Fehlschätzungen in der Speicherzuweisung teilweise korrigiert werden.

Speziell für die verschiedenen *HashJoin* Operatoren, wäre noch eine Ergänzung der *CostEstimation* Schnittstelle wichtig. Der Optimierer vergibt in den Ausführungsplänen für die *HashJoin* Operatoren immer die feste Anzahl von 99 997 Behältern (Buckets). Für kleine Eingabemengen ist dies zuviel und für größere Eingabemengen ggf. zu wenig. Die Anzahl der Behälter hat aber eine entscheidende Bedeutung für die Effizienz der Hashing basierten Operatoren, da die Anzahl der Kollisionen ansteigt, wenn die Anzahl der Behälter nicht groß genug ist. Daher wäre es sinnvoll, wenn der Operator über die *CostEstimation* Schnittstelle zukünftig auch die optimale

[1] Wenn die TPC-H Anfragen, die in SECONDO enthalten sind, als Grundlage der Behauptung dienen.

Anzahl von Behältern anzeigen würde. Die Arbeit [Nid12, Seite 100] geht hier sogar noch weiter und zeigt sogar einen Zusammenhang zwischen dem Arbeitsspeicher und der Anzahl der Behälter auf. Damit ist die optimale Anzahl der Behälter abhängig vom gewählten Arbeitsspeicher. Das Einbringen dieses Sachverhaltes in das nichtlineare Optimierungsproblems würde dann dazu führen, dass die Lösung des Optimierungsproblem nicht nur den optimalen Wert an Arbeitsspeicher für diesen Operator anzeigt, auch würde dieser den Wert für die optimale Anzahl an Behältern liefern. Ähnliches gilt für den Informationsfluss in umgekehrter Richtung. So wurde in der Arbeit [Nid12, Seite 139] angemerkt, dass es für die Kostenschätzung wünschenswert wäre, auch Informationen, wie beispielsweise die Selektivität und die Anzahl der Attribute, an die Kostenschätzung weiterzugeben.

Für die nichtlineare Optimierung selbst sind auch Erweiterungen denkbar, die sich auf die Effizienz auswirken. Angesichts der Schnelligkeit, in der bisher die Lösungen ermittelt werden, ist dies im Gegensatz zu den vorher genannten Erweiterungen weniger interessant. Als erstes könnte im Falle, dass nur lineare Kostenfunktionen vorliegen, auf die lineare Optimierung zurückgegriffen werden, anstatt die teurere, nichtlineare Optimierung durchzuführen. Auch könnte hier dann die ganzzahlige, lineare Optimierung Anwendung finden (Vgl. dazu auch Abschnitt 8.4 auf Seite 53). Als zweites könnte dann die Wahl des Startpunktes verbessert werden. Derzeit wird als Startpunkt für jeden Operator der Minimalwert an Arbeitsspeicher verwendet, der einem Operator zugewiesen werden kann. Wenn von den konstanten Kostenfunktionen abgesehen wird, ist dies, unter den hier gemachten Annahmen der Kostenfunktionen, sogar die ungünstigste Wahl, da dieser Punkt dann immer am weitesten von der optimalen Lösung entfernt ist. In der Regel führt dies dann zu mehr Iterationen des verwendeten Algorithmus, welches sich letztendlich in der Effizienz niederschlägt. Daher ist es möglich auch für die Wahl des Startpunktes einen Algorithmus einzusetzen der eine bessere erste Wahl für einen Startpunkt liefert. Ein möglicher Algorithmus dafür ist beispielsweise [LW08].

Weiterhin ist für die einphasige Optimierung immer noch ein Algorithmus von Interesse, der das Problem des kürzesten Pfades unter Beachtung der Arbeitsspeicherverteilung optimal löst ohne die Kosten der Aufzählungsstrategie zu verursachen. Auf die Problematik wurde bereits in Abschnitt 8.7 auf Seite 57 eingegangen. Es steht mit dem modifizierten Dijkstra-Algorithmus zwar eine gute Alternative der einphasigen Arbeitsspeicheroptimierung zur Verfügung. Trotzdem kann nicht ausgeschlossen werden, dass diese Strategie in manchen Situationen versagt.

Kapitel 12

Fazit

Für die Optimierung der Zuweisung von Arbeitsspeicher an Operatoren wurden verschiedene Strategien evaluiert. Basis für alle Strategien ist, dass ein nichtlineares Optimierungsproblem gelöst wird und für eine Arbeitsspeicherverteilung an die Operatoren sorgt, die einen kompletten oder auch einen teilweise vorhandenen Ausführungsplan bezüglich der Kosten minimiert. In der Evaluierung zeigt dann die Heuristik *modifiedDijkstra* gute Ergebnisse bei akzeptablen Zeiten für die Ermittlung des Ausführungsplans. Letztlich kann diese Heuristik aber nicht garantieren, den besten Ausführungsplan unter Berücksichtigung der Arbeitsspeicheroptimierung zu ermitteln. Die durchgeführten Experimente gaben allerdings keine Hinweise darauf, dass hier größere Abweichungen zum optimalen Ausführungsplan zustande kommen können. Damit wurde diese Strategie auch als Standardmethode für die Arbeitsspeicheroptimierung der Operatoren ausgewählt. In dem Testszenario (Kapitel 10) konnte immerhin eine Verbesserung im Mittel von ca. 8,4% im Gegensatz zur Referenzstrategie festgestellt weden.

Berücksichtigt wurden auch die blockierenden Operatoren, die es ermöglichen mehr Speicher zuzuteilen als eigentlich vorhanden ist. Dies ermöglicht für Ausführungspläne mit blockierenden Operatoren die Gesamtkosten noch weiter zu minimieren. Was nicht gelang, war dies in Experimenten zu zeigen, da auf Basis der Kostenfunktion der bisher einzige blockierende Operator, der *sortmergejoin*, aufgrund der Kosten nicht als Join-Operator für die Ausführungspläne berücksichtigt wurde.

Aufgetan haben sich insbesondere noch Schwächen in dem Kostenmodell und dessen Kostenschätzungen. Rechnerisch ist die Optimierung der Arbeitsspeicherzuweisung in der Lage die beste Zuteilung, zumindest mit der sehr kostenintensiven Aufzählungsstrategie, ermitteln zu können. Allerdings bedingt dies korrekte Eingabedaten, wie insbesondere die Kostenfunktionen in Abhängigkeit vom Arbeitsspeicher und den Selektivitäten. Im Abschnitt 9.7 wurden auch weitere Punkte aufgeführt, die Abweichungen vom realen optimalen Ausführungsplan bewirken können. Was die Evaluierung aber zumindest für das verwendete Testszenario zeigte, ist, dass trotzdem eine sinnvolle Arbeitsspeicheroptimierung durchgeführt werden kann. Wenn die in der Evaluierung diskutierten Probleme der konstanten Kostenfunktionen und der Konstantenermittlung der Kostenschätzung teilweise gelöst oder zumindest verbessert werden, könnte sich in einer erneuten Evaluierung herausstellen, dass sich die prozentualen Verbesserungen in der Ausführungszeit weiter erhöhen.

Anhang A

Grammatik des SECONDO SQL-Dialektes

Auf Basis der Grammatik aus [Pon12].

`sql-clause`	→	**let** `objectname mquery.` | **let**(`objectname, mquery, secondo-rest-query`). | **sql** `mquery.` | **sql**(`mquery, secondo-rest-query`).
`aggr`	→	`groupattr` | `groupattr` **as** `newname` | `aggr2`
`aggr2`	→	**count**(`distinct-clause *`) **as** `newname` | aggrop(`ext-attr-expr`) **as** `newname` | **aggregate**(`ext-attr-expr, aggrfun, datatype, datatype-`*`constant`*`)` **as** `newname`
`aggrop`	→	**min** | **max** | **sum** | **avg** | **extract** | **count**
`aggr-clause`	→	`aggr` | `[aggr, aggr-list]`
`aggr-fun`	→	$(*)\|(+)$ | **union_new** | **intersection_new** | ... % any name *fun* of a binary SECONDO-operator or function object with syntax $fun : T \times T \to T$ which should be associative and commutative. Infix-operators must be enclosed in round parentheses.
`aggr-list`	→	`aggr` | `aggr, aggr-list`
`all-expr`	→	`attr-`*`expr`* $<$ **all** `(subquery)` | `attr-`*`expr`* $<=$ **all** `(subquery)` | `attr-`*`expr`* $=>$ **all** `(subquery)` | `attr-`*`expr`* $>$ **all** `(subquery)`
`any-expr`	→	`attr-`*`expr`* $<$ **any** `(subquery)` | `attr-`*`expr`* $<=$ **any** `(subquery)` | `attr-`*`expr`* $=$ **any** `(subquery)`

		| attr-*expr* => **any** (subquery)
		| attr-*expr* > **any** (subquery)
attr	→	attrname | var:attrname
attr-list	→	attr | attr, attr-list
attrname	→	id
column	→	newname : datatype
columnlist	→	column | column, column-list
createquery	→	**create table** newname **colums** [columnlist]
		| **create index on** newname **columns** index-clause
datatype	→	**int** | **real** | **bool** | **string** | **line** | **points** |mpoint| **uregion**
		| **arel(** columnlist **)**
		| ...
		% any name of a SECONDO-datatype
deletequery	→	**delete from** rel-clause where-clause
distinct-clause	→	**all** | **distinct** | ϵ
dropquery	→	**drop table** relname
		| **drop index** indexname
		| **drop index on** relname indexclause
exists-expr	→	**exists** (subquery)
exists-not-expr	→	**not(exists** (subquery))
ext-attr	→	distinct-clause attr
ext-attr-expr	→	distinct-clause attr-*expr*
first-clause	→	**first** int-*constant* | **last** int-*constant* | ϵ
groupattr	→	attr
groupattr-list	→	groupattr | groupattr, groupattr-list | ϵ
groupby-clause	→	**groupby** [groupattr-list]|**groupby** groupattr
id	→	% any valid Prolog constant-identifier without an underscore-character
in-expr	→	attr-*expr* **in** (value-list) | attr-*expr* **in** (subquery)
indexname	→	id
indextype	→	**btree** | **rtree** | **hash** | ...
		% any name of a logical index type
index-clause	→	attrname | attrname **indextype** indextype
insertquery	→	**insert into** rel **values** value-list|**insert into** rel query
mquery	→	query
		| insertquery
		| deletequery

| updatequery
| createquery
| dropquery
| **union** [query-list]
| **intersection** [query-list]

newname → id
% where id is not already defined within the database or the current query

not-in-expr → attr-*expr* **not in** (value-list)

orderattr → attrname | attrname **asc** | attrname **desc** | **distance** (id, id)

orderattr-list → orderattr | orderattr, orderattr-list

orderby-clause → **orderby** [orderattr-list] | **orderby** orderattr | ϵ

pred → attr-*boolexpr*
| all-expr
| any-expr
| exists-expr
| exists-not-expr
| in-expr
| not-in-expr

pred-list → pred | pred, pred-list

query → **select** distinct-clause sel-clause **from** rel-clause where-clause orderby-clause first-clause
| **select** aggr-clause **from** rel-clause where-clause groupby-clause orderby-clause first-clause

query-list → query | query, query-list

rel → relname asrule
| subquery-table asrule
| arelname asrule
| outervar:arelname asrule
| rel **unnest** (attr) asrule
| rel **nest** (attr, newname) asrule
| rel **nest** ([attr-list], newname) asrule

asrule → **as** newname
| ϵ

rel-clause → rel | [rel-list]

rel-list → rel | rel, rel-list

```
relname              → id
result               → attr | attr-expr as newname | subquery-aggr as newname
result-list          → result | result, result-list
secondo-rest-query   → 'text'
                       % any valid subexpression in SECONDO executable language
sel-clause           → *
                       | result
                       | [result-list]
                       | count (distinct-clause *)
                       | aggrop(ext-attr-expr)
                       | aggregate (ext-attr-expr, aggrfun, datatype,
                       datatype-constant)
subquery             → subquery-aggr
                       | subquery-correlated
                       | subquery-simple
subquery-aggr        → % Subqueries extension: any query with aggr-clause but
                       without first-clause, groupby-clause and orderby-clause
                       MemoryAllocation extension: any query without union or
                       dinstinct terms.
subquery-correlated  → % any query without first-clause, groupby-clause
                       and orderby-clause which uses an attribute of a relation of an
                       outer query block
subquery-simple      → % any query without aggr-clause, first-clause,
                       groupby-clause and orderby-clause with a single result column
subquery-table       → % Subqueries extension: any query without aggr-clause, first-clause,
                       groupby-clause and orderby-clause
                       MemoryAllocation extension: any query without union or
                       dinstinct terms.
text                 → % any sequence of characters, that completes the optimized
                       query to a valid expression in SECONDO executable language
transform            → attrname = update-expression
transform-clause     → transform | [transform-list]
transform-list       → transform | transform, transform-list
update-expression    → % a fixed value, or an operation calculating a value
updatequery          → update rel set transform-clause where-clause
var                  → id
value                → % an integer, boolean or string value in prolog
```

`value-list` → `value` | `value, value-list`

`where-clause` → **where** `[pred-list]` | **where** `pred` | ϵ

Anhang B

Quellcode und Onlinequellen auf CD

Die CD befindet sich auf letzten Seite dieser Arbeit und beinhaltet folgende Teile:

- Quellcode der Implementierung
- Verwendete Literatur, die ausschließlich online verfügbar war
- Die verwendete NRW-Datenbank für die Evaluierung und die zugehörigen Skripte zum Laden der Daten in eine Secondo Datenbank
- Die verwendete Datei *ProgressConstants.csv* der Kostenschätzung
- Die vorliegende Arbeit als PDF-Datei

Literaturverzeichnis

[AGB06] Victor Teixeira de Almeida, Ralf Hartmut Guting und Thomas Behr. „Querying Moving Objects in SECONDO“. In: *Proceedings of the 7th International Conference on Mobile Data Management*. MDM '06. Washington, DC, USA: IEEE Computer Society, 2006, Seiten 47–. ISBN: 0-7695-2526-1. DOI: 10.1109/MDM.2006.133. URL: http://dx.doi.org/10.1109/MDM.2006.133 (siehe Seite 40).

[Arv12] Troels Arvin. *Comparison of different SQL implementations*. 31. Juli 2012. URL: http://troels.arvin.dk/db/rdbms/ (besucht am 11. 11. 2012) (siehe Seite 10).

[Cha98] Surajit Chaudhuri. „An Overview of Query Optimization in Relational Systems“. In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*. Herausgegeben von Alberto O. Mendelzon und Jan Paredaens. ACM Press, 1998, Seiten 34–43. ISBN: 0-89791-996-3 (siehe Seiten 2, 9).

[Cod70] E. F. Codd. „A relational model of data for large shared data banks“. In: *Commun. ACM* 13.6 (Juni 1970), Seiten 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: http://doi.acm.org/10.1145/362384.362685 (siehe Seiten 1, 6, 7).

[Dat99] C. J. Date. *An introduction to database systems (7th ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-38590-2 (siehe Seiten 5–7, 9).

[Dij59] E. W. Dijkstra. „A Note on Two Problems in Connexion with Graphs“. In: *NUMERISCHE MATHEMATIK* 1.1 (1959), Seiten 269–271 (siehe Seiten 10, 54).

[Gar08] Georgia Garani. „Nest and Unnest Operators in Nested Relations“. In: *Data Science Journal* 7 (2008), Seiten 57–64 (siehe Seite 19).

[Gü] Ralf Hartmut Güting. *Secondo: A Short Guide to Writing Executable Queries in Secondo*. Technischer Bericht (siehe Seite 7).

[Gü+04] Ralf Hartmut Güting u. a. *SECONDO: An extensible DBMS architecture and prototype*. Technischer Bericht. FernUniversität Hagen, 2004 (siehe Seiten 7, 10).

[Hem+09] Raymond Hemmecke u. a. *Nonlinear Integer Programming*. Technischer Bericht arXiv:0906.5171. Juni 2009 (siehe Seite 63).

[HSS00] A. Hulgeri, S. Seshadri und S. Sudarshan. „Memory cognizant query optimization“. In: *Proc. of COMAD* (2000), Seiten 181–193 (siehe Seite 45).

[Jac09] Adam Jacobs. „The pathologies of big data“. In: *Commun. ACM* 52.8 (Aug. 2009), Seiten 36–44. ISSN: 0001-0782. DOI: 10.1145/1536616.1536632. URL: http://doi.acm.org/10.1145/1536616.1536632 (siehe Seite 47).

[KHQ98] Gy. Kovacs, Cs. Hajas und I. Quilio. „Representations and query languages of nested relations“. In: *Annales Univ. Sci. Budapest, Sect. Comp.* 17 (1998), Seiten 235–252 (siehe Seiten 17, 19, 20).

[KR89] Henry F. Korth und Mark A. Roth. „Query Languages for Nested Relational Databases“. In: *Papers from the Workshop “Theory and Applications of Nested Relations and Complex Objects’ on Nested Relations and Complex Objects*. London, UK, UK: Springer-Verlag, 1989, Seiten 190–204. ISBN: 3-540-51171-7. URL: http://dl.acm.org/citation.cfm?id=648072.748524 (siehe Seite 17).

[LW08] Soomin Lee und Benjamin Wah. „Finding Good Starting Points for Solving Nonlinear Constrained Optimization Problems by Parallel Decomposition“. In: *Proceedings of the 7th Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*. MICAI ’08. Atizapán de Zaragoza, Mexico: Springer-Verlag, 2008, Seiten 65–76. ISBN: 978-3-540-88635-8. DOI: 10.1007/978-3-540-88636-5_6. URL: http://dx.doi.org/10.1007/978-3-540-88636-5_6 (siehe Seite 99).

[Mak77] Akifumi Makinouchi. „A consideration on normal form of not-necessarily-normalized relation in the relational data model“. In: *Proceedings of the third international conference on Very large data bases - Volume 3*. VLDB ’77. Tokyo, Japan: VLDB Endowment, 1977, Seiten 447–453. URL: http://dl.acm.org/citation.cfm?id=1286580.1286627 (siehe Seite 17).

[MG00] K. Marti und D. Gröger. *Einführung in die lineare und nichtlineare Optimierung*. Physica-Lehrbuch. Physica-Verlag, 2000. ISBN: 9783790812978 (siehe Seiten 14, 61, 62).

[ND98] Biswadeep Nag und David J. DeWitt. „Memory allocation strategies for complex decision support queries“. In: *Proceedings of the seventh international conference on Information and knowledge management*. CIKM ’98. Bethesda, Maryland, United States: ACM, 1998, Seiten 116–123. ISBN: 1-58113-061-9. DOI: 10.1145/288627.288647. URL: http://doi.acm.org/10.1145/288627.288647 (siehe Seiten 3, 45, 47, 65).

[Nid12] Jan Kristof Nidzwetzki. „Operator-Kostenmodelle für Fortschrittschätzung und Optimierung in Datenbanksystemen“. Abschlussarbeit. FernUniversität Hagen, 2012 (siehe Seiten 45, 51, 54, 59, 63, 67, 70, 98, 99).

[Nlo] *NLopt Algorithms*. 27. Sep. 2012. URL: http://ab-initio.mit.edu/wiki/index.php/NLopt_Algorithms (besucht am 11. 11. 2012) (siehe Seite 62).

[PA86] Peter Pistor und F. Andersen. „Designing A Generalized NF2 Model with an SQL-Type Language Interface“. In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB ’86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, Seiten 278–285. ISBN: 0-934613-18-4. URL: http://dl.acm.org/citation.cfm?id=645913.671466 (siehe Seiten 17, 19).

[Pon12] Burkart Poneleit. „Optimierung geschachtelter Abfragen im Datenbanksystem SECONDO“. Abschlussarbeit. FernUniversität Hagen, 2012 (siehe Seiten 12, 13, 24, 101).

[RKB87] Mark A. Roth, Henry F. Korth und Don S. Batory. „SQL/NF: a query language for ¬1NF relational databases“. In: *Inf. Syst.* 12.1 (Jan. 1987), Seiten 99–114. ISSN: 0306-4379. DOI: 10.1016/0306-4379(87)90021-4. URL: http://dx.doi.org/10.1016/0306-4379(87)90021-4 (siehe Seiten 17, 21).

[Sch03] Prof. Marc H. Scholl. *Vorlesungsfolien zum Kurs information systems, Kapitel 4, SQL:1999 - Objektrelationale Datenbanken.* 2003. URL: http://www.inf.uni-konstanz.de/dbis/teaching/ws0304/information-systems/K4.pdf (besucht am 11. 11. 2012) (siehe Seite 18).

[SDEV11] Ralf Hartmut Güting u. a. *Secondo Programmer's Guide Version 3.1.* FernUniversität Hagen, Faculty for Mathematics und Computer Science, Database System for New Applications, 2011. URL: http://dna.fernuni-hagen.de/Secondo.html/content_downloads.html#documentation (siehe Seite 7).

[SMEM11] Ralf Hartmut Güting. *Memory management in Secondo.* FernUniversität Hagen, Faculty for Mathematics und Computer Science, Database System for New Applications, 2011. URL: http://dna.fernuni-hagen.de/Secondo.html/content_downloads.html#documentation (siehe Seite 46).

[SOM12] Ralf Hartmut Güting. *Secondo Optimizer Manual.* FernUniversität Hagen, Faculty for Mathematics und Computer Science, Database System for New Applications, 2012 (siehe Seiten 11, 32, 73).

[SP82] Hans-Jörg Schek und Peter Pistor. „Data Structures for an Integrated Data Base Management and Information Retrieval System“. In: *Proceedings of the 8th International Conference on Very Large Data Bases.* VLDB '82. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1982, Seiten 197–207. ISBN: 0-934613-14-1. URL: http://dl.acm.org/citation.cfm?id=645910.673598 (siehe Seiten 13, 17, 19).

[SS94] Leon Sterling und Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques.* Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-19338-8 (siehe Seiten 9, 23, 27).

[Sta10] W. Stallings. *Computer Organization and Architecture: Designing for Performance.* Prentice Hall, 2010. ISBN: 9780136073734 (siehe Seite 3).

[SUSER11] Ralf Hartmut Güting u. a. *Secondo User Manual Version 3.1.* FernUniversität Hagen, Faculty for Mathematics und Computer Science, Database System for New Applications, 2011. URL: http://dna.fernuni-hagen.de/Secondo.html/content_downloads.html#documentation (siehe Seite 7).

[Sva02] Krister Svanberg. „A Class of Globally Convergent Optimization Methods Based on Conservative Convex Separable Approximations“. In: *SIAM J. on Optimization* 12.2 (Feb. 2002), Seiten 555–573. ISSN: 1052-6234. DOI: 10.1137/S1052623499362822. URL: http://dx.doi.org/10.1137/S1052623499362822 (siehe Seite 15).

[Teu12] Klaus Teufel. „Implementierung geschachtelter Relationen im Datenbanksystem Secondo“. Abschlussarbeit. FernUniversität Hagen, 2012 (siehe Seite 13).